Lab 1 — Supervised Learning in Feed-forward Neural Networks

0.1 Introduction

In this lab you will experience the joy of training perceptrons and multilayer perceptrons with gradient-descent learning algorithms. The particular focus is on getting the best possible generalization performance, and how network complexity and training set size affects generalization performance. The tasks you will look at are very simple. The goal is to get a notion of how these models learn and generalize.

0.2 Tasks

There are three problems which you will work on. The first task is a 2-dimensional classification problem called Toy. Because it is 2-dimensional, it is possible to view the decision boundary and see what the neural network is doing. The second problem is a regression problem (the desired outputs are numbers). The third problem involves classification of simple images. .

Note: in what follows, I first describe what must be done. In the subsequent sections I describe how to actually do it.

0.2.1 Task 1: Toy data

The first task is a 2-dimensional binary classification task. There are 300 examples in the dataset. Your goal is to first investigate the effect of the network complexity on the generalization performance. Try a perceptron, a simple MLP, and a (small) range of more complex MLPs in order to bracket the complexity required to get good generalization performance. Measure the generalization errors, but also examine the decision boundaries to see how that is affected by the network complexity.

Boston Housing Data

This is a regression task rather than a classification task. The goal is to predict the value of a house based on a number of numerical attributes concerning the area in Boston in

which it is located. This is a standard benchmark problem in machine learning and statistics. The output is a number between about 8 and 50 representing the median value (in $1000's) of owner-occupied homes in areas of Boston in the the 1970s. The input attributes represent properties of the area, including pollution indices, crime rate, and so on. (For a brief description of the meaning of the variables, see

`http://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html`

There are 506 examples, 13 input attributes, and one output attribute.

Find and record the approach which gives the best generalization performance. This will include the network architecture, the regularization method, and the partitioning of the training set. Questions you need to deal with:

1. How to deal with the fact that the variables lie in arbitrary ranges (e.g. not between 0 and 1).

2. What is the appropriate error measure? How should you determine the quality of the network performance?

0.2.2 How Many Stripes

The third task is a classification task on images. The images consist of stripes of varying angles and varying frequency. The task is to distinguish those which have three or more stripes in the image. I.e. the desired network output is 1 if there are three or more stripes and 0 otherwise. (I.e. the system has to distinguish high frequency from low frequency.) Find an approach which generalizes with less than 10% error.

Question to think about:

1. Can a perceptron do this task? You might be able to figure this out by thought alone.

2. How many possible patterns are there, assuming that each frequency and angle of stripe is possible?

3. Does your network generalize? I.e. can you get good out-of-sample performance?

4. Examine the weights using the image visualization tools discussed in the final section. Can you see how the network is solving the problem?

0.3 Resources — Supervised Learning in Matlab

0.3.1 Representation

The inputs, outputs, and weights are represented in matlab as described in the appendix B of the pre-course work. A set of weights and biases are associated with each layer of neurons. Although the weights and biases are taken care of by high level commands, I reiterate the representation details here in case you need them.

Inputs: An N component pattern is a column of N numbers. A pattern set consisting of Q patterns is N $\times$ Q matrix.

Weights and Biases: The weights for a node with R inputs is a row of R numbers. A layer of S neurons requires an S $\times$ R matrix for the weights, and a S $\times$ 1 column for the biases. For a multilayer perceptron, the following convention is used: w1 and b1 denote the weights and biases from input to the first hidden layer, w2 and b2 denote the weights and biases on the next layer, and so forth. Matlab routines can create and train networks with one, two, or three layers of neurons.

Desired Output: The desired output is an M $\times$ Q matrix, where M is the number of output nodes. For the problems here, M is 1.

0.3.2 Getting the First Dataset, Examining and Partitioning It

To get the **Toy** data, the command is

```
load toy
```

The input patterns are in `toy.P` and the desired outputs are in `toy.T`.

The data is two-dimensional, so you can plot it. A command to do this is

```
plotpv(toy.P,toy.T)
```

will do this.

You need to partition the data into training, validation, and test sets. Since part of your task is to explore different network complexities, the validation set is required to compare them. The test set is used at the end to estimate the true error of the final network.

I have made a tool to do the partitioning. The command is

```
[training,testing,validation]=partitionTool(toy);
```

This creates a training set with inputs called `training.P` and desired outputs called `training.T` and likewise for testing and validation. When given the choice of "contiguous" or "random" sampling, choose "random". This shuffles the order of the patterns before partitioning. This is required because in the original data, all patterns of a given class are together.

0.3.3 Supervised Learning in Matlab Neural Network Toolbox Version 3.0

Overview

The basic scheme for creating and training neural networks in Matlab works like this,

1. Create the training (and validation, etc.) data. In Matlab, the data is put into two matrices — one containing the input patterns and one containing the desired output or labels.

2. Create a new network. The command is of the form `net = newXX` where XX is different for different types of networks. A network is stored is a variable which is a structure, like a structure in C.

3. Implement learning. There are two forms of training algorithms. On-line learning is implemented using `adapt`; batch learning is implemented using `train`. For example,

   ```
   net = train(net,p,t);
   ```

   trains network `net` on input patterns `p` with desired outputs `t`. Since `net` has been updated with the trained version, so to continue training this network, simply run the command again.

4. To re-initialize a network, the command is `init`, e.g.

   ```
   net=init(net);
   ```

5. To simulate a network, i.e. calculate the network outputs, the command is `sim`. E.g.

   ```
   a = sim(net,p);
   ```

   produces the outputs of `net` on input patterns `p`.

6. To save a neural network, the command is

   ```
   save fname net
   ```

   which saves the network called `net` to a file called `fname.mat`.

   ```
   load fname
   ```

   loads it in again. See `help save` for more details.

These steps are described for particular neural networks below.

Creating and Initializing a Perceptron

To create a perceptron, the command is

```
net = newp(inputrange,numbernodes);
```

where

inputrange is a matrix with the same number of rows as the network has inputs. Each row contains the minimum and maximum value of the patterns on that input. This tells matlab how many input attributes there are and also an appropriate scale for initializing the weights.

numbernodes is the number of (output) nodes.

For example, to use create a perceptron called `perc` with two inputs which will receive binary input, use,

```
perc = newp([0 1;0 1],1);
```

since there are two inputs, each in the range 01, and 1 output.

In general, the command for determining the input range would be,

```
inputrange = [min(toy.P')' max(toy.P')'];
```

and use `inputrange` for all of the new commands. However, this is quite tedious, so I have made a command to do it called `getRange`,

```
inputrange=getRange(toy.P);
```

So,

```
perc=newp(getRange(toy.P),1);
```

would work.

Creating and Initializing an MLP

To create a general feedforward neural network, including an MLP, the command is

```
net=newff(inputrange,[number1,number2],{transfer1,transfer2},trainingalgorithm)
```

Here

inputrange is as in the perceptron, a matrix of ranges the inputs take,

[number1,number2] is a list of the number of units in each layer. Examples: [50,1] would denote 50 units in a hidden layer, 1 output; [1] would denote 1 output and no hidden units, [3,4,2] would denote 3 units in the first hidden layer, 4 units in the second hidden layer, and 2 outputs. (Remember, it figures the number of inputs from `inputrange`; that number is not in the list.)

transfer1,transfer2 is a list of transfer functions for each layer. These are strings. We will typically use sigmoid functions, which are called 'logsig' in Matlab, or hard-limiting functions, which are called 'hardlim' in Matlab. Another useful transfer function is 'purelin' in which the output is just the weighted sum of inputs.

trainingalgorithm is described by a string. Matlab includes many: simple gradient descent, gradient descent with adaptive learning rate (stepsize), conjugate gradient descent, and others. A complete list is appended.

For example,

```
net=newff([0 1;0 1],[2,1],{'logsig','logsig'},'traincgf');
```

would create a 2-2-1 network of sigmoid neurons which will receive binary input and will be trained using a conjugate gradient descent method.

Viewing the Network

If you type,

```
>> net
```

you will see all of the fields of the structure which defines the network. Unfortunately, since the structure is devised to represent a wide range of different type of neural networks, it is very complex and not terribly illuminating. To look at a particular field, put a dot between the name of the variable and the field. E.g.

```
net.adaptParam
```

will show you the adaptation parameters. This is a structure as well; it has fields. For example,

```
net.adaptParam.passes
```

which gives the number of presentations of all patterns before `adapt` terminates. The learning parameters used by the `train` command are in `net.trainParam`. This is rather difficult to deal with, so I have made a command to show it,

```
net=getParameters(net);
```

This shows and allows you to change the parameters. It shows both training parameters (which batch algorithms such as gradient descent learning use) and adapt parameters (which on-line algorithms such as perceptron learning algorithm use), even though the algorithm will only use one of this. (I am not clever enough to make it figure out which is used.)

For a perceptron, the weights are in `net.IW{1,1}` and the bias in `net.b{1}`. (The IW{1,1} means input weights from the first set of inputs to the first layer of units. Denotes an object in Matlab which is an array of matrices, called a "cell array".) For an MLP, there will be weights in `net.LW{2,1}` which denotes the matrix of layer weights from layer 1 to layer 2. The biases will be in `net.b{2}`. So, `net.IW{1,1}(i,j)` contains the weight from the jth input to the ith hidden unit.

Training the Network

Perceptron learning algorithm is an one-line learning algorithm, so for perceptrons use `adapt`. The command is

```
net=adapt(net,p,t);
```

This runs the perceptron learning algorithm for a number of passes set by `net.adaptParam.passes` on the input patterns `p` with desired outputs `t`. I have produced an algorithm which runs `adapt` for multiple steps and also plots the error and the error on a validation set. It is called `multistepadapt`. The command is

```
net = multistepadapt(net,nsteps,p,t,vp,vt,nskip);
```

Here nsteps is the number steps `adapt` is to be run, `vp` and `vt` are the validation patterns and desired outputs, and nskip is the number of steps to skip between updating the plot. The latter three are optional.

Most of the fast gradient descent algorithms are batch learning algorithms, so `train` is used to train MLP's typically. The command is

```
net=train(net,p,t);
```

This will train the network using as training data `p` and `t` using the learning algorithm specified when `net` was created using `new`. You should see several lines of text output, such as:

```
TRAINGD, Epoch 50/300, MSE 0.298671/0, Gradient 0.114085/1e-10
```

This describes, from left-to-right: the training method, the current epoch (i.e. training cycle) and the maximum epoch, the current mean-squared error (sum-squared error divided by the number of patterns) of the training set data and the goal error, the current size of the gradient and the goal gradient. Training terminates when any of the number of epochs, the mean-squared error, or the gradient size reaches the goal value. There is also a graph displayed, which shows the training set mean-squared error against time.

The learning command `train` can be called with a validation set, either to monitor the validation error during training or to implement early stopping. The validation data must be in a structure with `v.P` containing the inputs and `v.T` containing the desired outputs. It is in that form for the first two data set. If it isn't and `vp` is the input patterns and `vt` is the desired outputs of the validation set, then

```
v.P=vp;
v.T=vt;
```

will put it into that format.

The command for early stopping is

```
net=train(net,p,t,[],[],v);
```

which will stop training when the validation error starts to rise. This can happen quite early and could just be a fluctuation, so you might want to try restarting learning after using this.

To simply monitor the validation error, but not implement early stopping, use the following,

```
net=train(net,p,t,[],[],[],v);
```

In either case, the `[]` are placeholders for other inputs which we will not use in these labs.

Testing the Network

You can see the results of the input data acting on the network by using the `sim` function:

```
a = sim(net,p);
```

which will return a matrix of the outputs of the network using `p` as inputs. The matrix `a` has as many rows as the network has outputs and as many columns as there are patterns. Each column represents the output for one input pattern.

To get the sum-squared error or mean-squared error, use `sse` and `sse` respectively,

```
sse(a-t)
```

```
mse(a-t)
```

where `a` is the network outputs computed using `sim` and `t` is the desired output.

For classification problem mean-squared error might not be what you want. I have produced a function called `test_classifier`. The format is

```
test_classifier(class_rep,t,a);
```

The first argument is a matrix of columns showing the desired output of class1, class2, ... For example, suppose the problem has two classes, and the network has two outputs, and the desired output for class 1 is `[1;0]` and for class 2 is `[0;1]`. The command would be

```
test_classifier([1 0;0 1],t,a);
```

The output of `test_classifier` is a confusion matrix. The rows are the true classes, the columns are the classes picked by the network. The first column corresponds to those patterns unclassified by the network (the output cannot be interpreted as one of the classes); the first row corresponds to those patterns which are not given valid classification labels, there will typically be none of these. For example

```
0     0     0
1    67     7
0     4    22
```

tells us that 67 of the class 1 patterns were correctly classified by the network as being class 1, 7 of the class 1 patterns were incorrectly classified as class 2, 4 of the class 2 patterns were incorrectly classified as class 1, and 22 of the class 2 patterns were correctly classified as class 2. The first column contains a 1 in row 2. This means that one pattern from class 1 was unclassified by the network; its output was neither `[1;0]` or `[0;1]`.

For MLPs with sigmoid functions, the outputs will never be exactly 0 or 1. You have to decide how to get a precise class from the network. The simplest way is to round the network outputs, e.g.

```
a = round(sim(net,p));
```

For the toy data (only) there are other tools which can be used.

```
NewDecision2d(perc,toy.P,toy.T)
```

will attempt to draw the decision boundary. This is a difficult computation, especially for MLPs, so it may not get it perfectly right. However, it is very interesting to see how MLPs can give more complex decision boundaries. Obviously, use the appropriate names for the network and the datasets. A similar tool is

```
plotClassification2d(toy.P,toy.T,a)
```

This plots the data, and colors those points which are misclassified.

Improving Generalization

There are three ways to get better generalization performance for a given size of training set. The first is the reduce the number of adjustable weights, e.g. by reducing the number of hidden units.

The second way is the use early stopping. This can be implemented using a validation set in the training commands described above.

The third way is to use weight-decay regularization. This is implemented in one of the training algorithms called `trainbr`. This shows on a graph the effective number of adjustable parameters, in addition to the sum-squared error and the sum-squared weights. This requires sufficient computation to run very slowly indeed, unfortunately.

0.3.4 Getting the Other Data Sets

Boston Housing Data: To get this data, run the command

```
load housing
```

The training patterns are in `housing.P` and the desired outputs are in `housing.T`.

Stripes Patterns: To make a set of patterns, the command is

```
[p,t]=stripes(100);
```

This makes 100 patterns; the number is determine by the number input to the function. These are 16×16 black and white (0 and 1) images.

Since the patterns are two-dimensional images, it is useful to view them as such. I have made three routines which allow you to do this: **showpattern2d**, **showpatterns2d**, and **showpatternset2d**. The first one takes a single image, so it would be used like this,

```
showpattern2d(p(:,1),16,16);
```

to view the first pattern, if p is a pattern set created as above. The two 16's tell it that the vectors are to be viewed as 16 by 16 images. The second two are used to view the entire set,

```
showpatterns2d(p,16,16);
showpatternset2d(p,16,16,10,10);
```

The first displays each image in p one at a time, The second displays all patterns in p as a 10 by 10 array.

These commands can also be used to display the weight vectors going into the first layer of hidden units. This could be useful in attempting to determine what the hidden units are doing. Since the weights are row vectors, it must be transposed before being displayed with these commands (' is the transpose operator). As discussed previously, the weight vector from input layer to hidden layer for a network called 'net' would be called 'net.IW1,1'. If there were 9 hidden units, the command

```
showpatternset2d(net.IW{1,1}',16,16,3,3);
```

would be an appropriate way to view these weights. Remember: to produce a new figure window, so as to have more than one plot on the screen simultaneously, type the command **figure**.