Homework # 3

15-486/782: Artificial Neural Networks

Exercise created by Dean Pomerleau and Dave Touretzky

Due Wednesday, September 27. This homework examines the use of the backpropagation learning rule on the difficult real world task of autonomous road following. You will train and test an MLP that maps simulated road images to the correct steering direction for that image, and examine the representations the network develops.

Make sure you've done all the readings relevant to backpropagation learning and the ALVINN system before beginning this assignment. **Answer all questions appearing in the text. Hand in plots where requested. You do not need to hand in any color figures of weights. If your plots are not in color, then use different line types (solid, dashed, dotted) for multiple curves on the same plot.**

1. First we'll take a look at the datasets for this problem. Use the `load` command to load the file `alvinn_data.mat` from the `matlab/alvinn` class directory. (Note that this is a binary file, so if you FTP it to another computer, be sure to use binary mode for the file transfer.) Use the `whos` command to see the variables that were loaded. The training set consists of the variables `Patterns` (250 actual road images taken from the Navlab's camera, each 30×32 pixels, encoded as a 960 element vector), `Desired` (250 desired output vectors, each consisting of 30 elements, depicting a gaussian bump centered on the correct steering direction), and `Positions`, a vector of 250 values giving the correct steering direction for each road image as a real number between 1 and 30. There are other variables as well, which will be explained later.

   Also included in the `alvinn` directory are some Matlab routines you will want to copy. To "take a tour" through the training data, type: `tour(Patterns,0,Desired)`. You can stop the tour by hitting `^C`. Note that the window displaying the road images has two buttons marked "+" and "−" in the top right corner; these can be used at any time to scroll through the road images.

   There is also a "cross-validation" dataset (actually just a test set, since we won't be doing true cross-validation in this problem) which we'll use to determine when to stop training the network. Take a tour through this dataset by typing `tour(CVPatterns,0,CVDesired)`.

   Questions: What kind of road is used in the training and cross validation datasets? What prominent road feature does the cross-validation dataset contain that is almost entirely absent from the training dataset?

2. Now we will assemble the core routines for training the backprop network. Start with the Matlab scripts `approx`, `approxhelper`, and `bp_innerloop` in the class `matlab/bp` directory. Copy the scripts to your directory and rename the first two to `alvinn` and `alvinnhelper`.

   (a) Modify your `alvinn` routine to make it a script rather than a function. Fix the number of hidden units at 4. Set `DerivIncr` to 0.1 and `Momentum` to 0.9.

   (b) The `alvinn` script sets up a "learning rate schedule" for backprop, which is then used by `alvinnhelper`. When training large networks, it is advantageous to start off with a slow learning rate for the first 50 or so epochs so the hidden units can sort themselves out and get moving in the right direction before taking any large steps in weight space. Then the learning rate can be safely increased. It may help to reduce the learning rate later in training to fine tune the weights. For the ALVINN problem, set your learning rate schedule to run for 50 epochs at 0.001, then 100 epochs at 0.007, and finally 50 epochs at 0.004.

(c) Another trick to provide better initial conditions for backprop learning is to make sure the magnitudes of the initial random weights are small enough that the hidden units are not driven to the flat spots on the sigmoid, yet large enough that the units are effectively differentiated. For large networks such as ALVINN, a unit's initial random weights should be scaled by (i.e., divided by) the square root of the fan-in. Make this change to your `alvinn` routine.

(d) As we saw in class, a unit's learning rate should also be scaled by its fan-in. Make this change to the `bp_innerloop` script.

(e) Because we will be comparing performance of the network on datasets of different sizes, we should normalize the total sum-squared-error (called `TSS`) by dividing by the number of patterns. Modify the `bp_innerloop` script to do this.

(f) Another common backprop trick is called "weight decay". At each time step, after the weights are updated, the magnitudes of all weights are reduced a tiny amount by multiplying by a constant slightly less than one. With repeated reductions, the weights of "unimportant" connections will decay to zero, while for the important connections, i.e., the ones that contribute significantly to solving the problem, the learning algorithm will cause the weights to grow, counteracting the decay effect. Modify `bp_innerlooop` to multiply the newly computed weights by `(1-WeightDecay)`. Typical values for this parameter range from 0 (for no decay) up to 0.05.

(g) Modify `alvinnhelper` to display its progress during learning, as follows. On every iteration it should print the epoch number and the TSS error. On every fifth iteration, it should select figure 1 and call

```
PlotAlvinn(epoch,Patterns,Result2,Desired)
```

and select figure 2 and call `PlotAlvinnWeights(Weights1,Weights2)`.

(h) Now you are ready to try training an ALVINN network. Watch the hidden unit weights as the network learns. What effect does the weight decay parameter have on learning?

3. A network trained to minimum sum-squared error on the training set may not produce the best possible results on a new test set. The reason is that in a net with many parameters and a modest sized training set, overfitting can occur. With overfitting the network learns weights reflecting not just the function we intended it to learn, but also any statistical irregularities (noise) in the training set. The specific peculiarities of the training set will not exist in the test set (which of course has peculiarities of its own), so the network does not do as well as it could if it had not tried to model the training set noise. Therefore, we use performance on a second dataset, the cross-validation dataset, to determine when to stop training. Generally, the training set error will continue to decrease as training progresses, but the cross-validation error will decrease to some minimum value and then begin to increase.

Note: although the network is always trying to move downhill on the error surface, if the learning rate is set to an aggressive (high) value, the sum-squared error on the *training* set can increase on some trials, when the network takes too big a step in weight space. It will eventually go down again. **If the error does not get down to around 2.0, the network has not really learned this dataset; there may be something wrong with your initial weights, fan-in correction, or learning schedule.**

(a) Modify `alvinn` to create `CVInputs1` from `CVPatterns`, just as `Inputs1` was created from `Patterns`.

(b) Write a function `forwprop` that peforms just the feed-forward computation part of an MLP network. You can extract this code from `bp_innerloop`. Your function should take the following form:

```
[Result1,Result2,TSS] = forwprop(Inputs1,Desired,Weights1,Weights2)
```

Note that since `forwprop` is a function, variables such as `Result2` and `TSS` are local to it, and do not conflict with the global variables of the same name used by your `alvinn` script.

(c) Modify `alvinnhelper` to run a forward propagation step on the cross-validation dataset with each epoch of training, storing the return values in variables `CVResult1`, `CVResult2`, and `CVTSS`.

(d) Modify `alvinnhelper` to record the `TSS` and `CVTSS` values for each epoch in an array. It should also keep track of the best weights seen so far for each case, i.e., if the current weights produce the lowest `TSS` value seen so far then set `Best_Weights = {Weights1 Weights2}`. (The brace notation is a "cell array"; see the Matlab manual for an explanation.) Store the best weights seen so far on the cross-validation set in the variable `Best_CV_Weights`.

(e) Modify `alvinnhelper` to print the cross-validation error as well as the training error for each epoch. It should also graphically display the record of both error measures, as follows. On every fifth epoch, in addition to displaying the network output in figure 1 and the current weights in figure 2, select figure 3 and plot the complete history (so far) of `TSS` and `CVTSS`, using lines of different colors. In addition, plot an open circle symbol of the appropriate color at the location of the lowest value for each line.

Graphics hints: use `hold on` so you can plot multiple things in the same figure. Use `axis([0 200 0 25])` to fix ths ranges of the graph so it doesn't get rescaled each time you add new points. Use `legend` to label the lines of the graph.

(f) Compare the weight vector that gives the best generalization on the cross-validation set with the weight vector that gives the lowest error on the training set. Describe in a sentence or two how they differ. You can plot the best cross-validation weight vector by typing: `PlotAlvinnWeights(Best_CV_Weights{1},Best_CV_Weights{2})`.

(g) For the rest of this exercise we will want to stick with the weights that give the best generalization on novel inputs, so use `forwprop` with the best CV weights to find the network's best output on the cross-validation set.

4. If we want to use our neural net to drive a car, we must extract a single steering direction value from the network's output.

(a) Write a function `[dir,err]=extractdir(Result,Gaussians)` that takes as input a 30 element vector (could be one column of `Result2` or `CVResult2`) and a set of model gaussians, and returns two numbers: a number `dir` between 1 and 30 indicating the steering direction, and the sum-squared error `err` between the result vector and the best-matching gaussian.

Here's how to compute the answer. The global variable `Gaussians` from the `alvinn_data` file is a 30×291 matrix containing 291 gaussian bumps with peaks at locations 1 to 30 in steps of 0.1. Given a vector of output unit values, compute the sum-squared distance between that vector and each of the 291 gaussians. If i is the index of the minimum-distance gaussian, then the corresponding steering direction is $1 + (i - 1)/10$.

As a test: `[dir,err]=extractdir(Desired(:,1),Gaussians)` should return a steering direction of 12.6 and a sum-squared error of 0.0037. Note that `Positions(1)` equals 12.6610.

(b) The correct steering direction for each pattern is given in the variable `Positions` or `CVPositions`. Write a function `cmpbump(imageno,result,Gaussians)` that takes a road image number, a matrix of output patterns such as `CVResult2`, and a set of model gaussians as inputs, and plots both the actual output pattern for that road image and the curve of the best-matching gaussian. Find an image for which the match is very close, and an image for which the match is poor. Hand in plots for both images.

5. An extremely important function that any MLP applied to a real world problem must address is confidence estimation. The network needs to be able to determine when it is confused, to avoid making catastrophic mistakes (e.g., steering into a ditch when it comes to a confusing stretch of roadway). One approach thats allows the network to estimate its own confidence is a simple extension to the gaussian matching technique you implemented above. This approach is called Output Appearance Reliability Estimation (OARE).

   OARE is based on the premise that if the network is trained to output a gaussian peak of activation with a particular shape, and the network's actual output doesn't have this shape (either because there are multiple peaks in the output, or no strong peak at all), then the input pattern probably depicts a situation unlike anything in the training set, and the network's response is likely to be incorrect.

   (a) Use the cross-validation images for this task. For each cross-validation pattern, find the best matching gaussian output vector and the sum squared error of the match using the `extractdir` procedure from the previous question. This sum squared difference is called the output appearance error. The lower the output appearance error, the more closely the shape of the network's output matches the ideal gaussian shape, and the more confident the network should be of its output. Normalize the vector of output appearance errors on the set of cross-validation patterns by dividing by the maximum value.

   (b) For each cross-validation pattern, compute the steering error as the square of the difference between the estimated steering direction (as returned by `extractdir`) and the correct direction given in `CVPositions`. Normalize the vector of steering errors by dividing by the maximum value.

(c) Write a function `PlotOARE(result,positions,Gaussians)` that plots the OARE and squared steering error values for a set of output patterns.

(d) If your network has been trained properly, you should see three regions containing peaks in the OARE error for the cross validation dataset. Two of these regions have corresponding peaks in the steering error. Thus, OARE seems to be a fairly good predictor of steering error. Question: what is happening in the road images where these two peaks occur?

6. A *sensitivity analysis* of ALVINN's hidden units helps to reveal how they respond to roads at different shifts and orientations. In this portion of the exercise you will conduct a simplified sensitivity analysis of your trained network, looking only at shifts.

   (a) Write a function to construct a simulated 30×32 road image with a trapezoidal shape. The road should be 10 pixels wide at the top of the image and 24 pixels wide at the bottom. Road pixels should have values of 0.6, and non-road pixels should have values of -0.6.

   (b) Write a function to generate a set of road images shifted left or right of center by varying amounts. (Since we're looking only at shifts, not rotations, this is trivial to compute.) At the furthest left or right shift, no road is visible. At intermediate values of shift, only part of the road may be visible. Plot two of your simulated road images with different shift values and hand them in as part of your answer to this problem.

   (c) Use your `forwprop` function to run the trained network on your sequence of synthetic road images. The return value `Result1` holds the hidden unit activations. Plot the hidden unit activation as a function of road shift; use a different color line for each hidden unit. (Note: the `plot` function can plot multiple lines at the same time if given a matrix argument, and each line will automatically be assigned a different color. Or you can specify colors explicitly if you prefer) Use `legend` to label your lines with the corresponding hidden unit number.

   (d) What correlation do you see between the locations of peaks and valleys in the sensitivity plot and the positive and negative values of the hidden-to-output weights? (Give a qualitative description; you don't need to compute correlation coefficients.)

7. How well does your trained network handle multi-lane roads? Try running it on the images in `TwoLanePatterns`. What is your assessment of its performance?

Hand in the following on September 27:

- all the code you wrote
- a printout of a sample training run showing the training set error and CV error for each epoch
- the plots you generated
- short answers to the various questions asked in the text above.