Multilayer Perceptron Ensembles trained with Early Stopping: `mlp-ese-1`

Carl Edward Rasmussen

May 6, 1996

Early stopping is a technique for avoiding overfitting in models that use iterative learning procedures. One uses a model which is expected to have a too large capacity and through early stopping one ensures that the model does not overfit. Consequently this approach should be helpful in overcomming the bias/variance dilemma, since only an upper bound on model capacity is required rather than a good estimate of the optimal capacity.

A fraction of the training examples are held out for validation, and performance on this set is monitored while the iterative learning procedure is applied. Typically, the validation error will initially decrease as the model fits the data better and better, but later on, when the model begins to overfit, the validation error starts rising. The idea is to stop learning as soon as this minimum in validation error is achieved. For neural networks, one of the great advantages of early stopping is that the difficult question of how long to train the model vanishes.

A number of details have to be specified to make the method practically applicable. This section contains a discussion of the issues involved and the decisions that I have made for how to implement the method. I should stress that I am not claiming my choices to be optimal in any sense, rather I try to define a method which cannot in any obvious (and well documented) way be improved upon.

I use networks with a single hidden layer of hyperbolic tangent units and linear output units. All units have biases. The network is fully connected (including direct connections from inputs to outputs). The number of hidden units is chosen to be the smallest such that the number of weights is at least a large as the total number of training cases (after removal of the validation cases). We need to decide how large a fraction of the training cases to use for validation. There are two obvious conflicting interests; we want a large validation set to achieve a good estimate of performance and we want a large training set to be able to train a complex model, namely as complex a model as could be supported by the entire training data set. There seem to be no helpful theoretical results about this tradeoff (for non-linear models and finite amounts of data). I chose to use one third of the training examples (rounded down if necessary) for validation and the rest for training.

Another question is the exact criterion for when the training should stop, since we do not want to stop merely when small fluctuations in the validation error occur. A number of more or less complicated stop criteria have been proposed in the literature, but none has convincingly been shown to outperform others. Here I will use the very simple scheme of training until the iteration with smallest validation error lies 33% backwards in the run. In this way I will have trained for 50% more epochs than 'necessary' and can thus be fairly confident that I have found a good minimum. I will do a validation after every line-search in the training run (training with conjugate gradients). Since the validation set is only half the size of the training set and validation only requires a forward pass, validation will typically slow down training by about 20% (since typically the line search involved in each iteration requires on average about 1.3 forward and backward passes), which I do not think is an alarming slowdown. More computer efficient strategies could be employed by validating only at longer intervals, at the risk of finding worse stopping times. If a simpler method of steepest descent with a fixed step-size was used as an optimisation technique, it would probably be sufficient to validate at longer intervals, but the conjugate gradient technique I use involves line-searches and can potentially take very large steps.

The idea of including the validation set and continuing training is appealing, since it is expected that better performance can be achieved with larger training sets. However, the problem of when to stop this additional training presents itself. One suggestion could be to train on until the same training error per example is achieved. From a theoretical point of view, this idea seems reasonable in an average case analysis, since it is expected that validation examples will have a slightly larger error initially than the training examples. However, there are practical problems; what if the validation error is already lower (per case) than the training error? And what if it is impossible to achieve the same training error using all the training data? Even if scenarios as bad as these do not occur frequently, it is still possible that the generalisation performance will often be decreased by using this procedure.

Instead of including all the training data and retraining, I will effectively use all the data by training an ensemble of networks, each of which is trained using a different validation set (chosen at random). This is a convenient way of combining the benefits of averaging by ensembles with the ability to use all examples for training. Using different validation sets for each net has the side effect that one cannot choose between networks and only include the ones with best validation error, since the variation in validation error might be due to the differing validation sets. Consequently I will include all trained nets in the final ensemble. Since the networks have many weights, they are probably not too prone to bad local minima, and selecting networks for inclusion in the ensemble is probably not necessary. A more complicated hybrid scheme with a few training iterations for each validation set could be contemplated, but for simplicity this will not be pursued further.

Additionally, we must take care of the cases where early stopping seems to stop too early or too late. In the initial phases of training it is conceivable that the validation error fluctuates quite wildly and this may cause the stopping criteria to be fulfilled (since the extra 50% epochs may be very few and thus not be a reliable indicator of local minima). To rule out this scenario, I require that each training run uses at least 2% of the total available training time and that at least 250 iterations have been done (unless the minimization algorithm converges). This will limit the number of possible members of the final ensemble to 50, which is probably adequate for getting most of the benefit from averaging. On the other hand, it may be that the validation error keeps decreasing and the stopping criteria is never satisfied. Recall that the *expected* behaviour of the validation error is that it should increase when overfitting sets in, but there is no guarantee that this will happen in any particular run. On the other hand, it may be that it simply takes a large number of epochs to train the network. To resolve this dilemma, I will terminate a training if it uses more than 33% of the total computational resources available, such that the final ensemble will contain at least 3 networks. Technically, some of the above criteria can be conflicting; in the actual implementation the conditions for deciding to stop the training of a particular net in order of precedence are: if the network has converged; OR if more than 33% of the allowed cpu time has been used on training this net; OR if more than 250 iterations of training has been completed AND the minimum validation error was achieved at least 33% backwards in the run AND we've used at least 2% of the total available cpu time on the current net.

This method can now be run automatically without any further specifications, except that a limit on the allowed amount of cpu time has to be specified. As a default I have chosen to let the allowed cpu time scale linearly with the number of training cases. The default allows 1.875 seconds per training case; this means that net net with 1024 training cases will be allowed to train for 32 minutes.

Implementation

The network is trained for a single iteration at a time using the conjugent gradient function `conj()`. Between iterations the error on the validation set is computed and if performance is better than the previously best then the iteration number and performance `best_iter` and `best_perf` are updated and the weights of the current net saved in `best_w`. This is repeated until we run out of time or till `iter` is more than `esmargin` (early stopping margin) in excess of `best_iter` and at least the fraction `1/MaxEnsembleSize` of the total available time and at least `MinIteration` iterations have been expended on this net. If we did not run out of time or we have trained less that `MinEnsebleSize`, the best net is saved to disk. More nets are trained while the amount of used time is less than the allowed `length`.

Each new net gets random weights drawn from a zero mean normal distribution with a standard deviation set to 0.1 divided by the square root of the fan-in of the unit which the weight projects to. This should ensure that the initial weights keep the activities of the sigmoid units close to their linear region, which is thought to be beneficial for early stopping.

⟨main 3⟩ ≡

```
main(int argc, char **argv)
{
  ⟨define variables 6c⟩
  ⟨parse command line 4a⟩
  ⟨load and partition cases 5b⟩
  ⟨open weights file 6a⟩
  ⟨create network 6b⟩
  ensembleSize = 0;
  fprintf(stderr, "net  train-err valid-err  best-iter  max-iter\n");
  while (used_time < length && ensembleSize < MaxEnsembleSize) {
    ⟨generate random train/validation sets 5a⟩
    randomizeNet(0.1);
    initTimer(&timer, min(length/MinEnsembleSize, length-used_time));
    best_perf = cost(valid, 0);                /* initialize best_perf and... */
    for (i=0; i<no_wts; i++) best_w[i] = w[i];   /* best_w to the random net */
    iter = best_iter = 0; restart = 1;   /* conj() should restart its search */
    do {
      succ = conj((i=1, &i), (j=0, &j), &restart, &junk); iter++;
      if ((perf=cost(valid, 0)) < best_perf) {
        best_iter = iter; best_perf = perf;          /* update "best" values */
        for (i=0; i<no_wts; i++) best_w[i] = w[i];           /* save this net */
```

```
      }
    } while ((succ) && (!TIMEOUT) && ((iter < EarlyStopMargin*best_iter) ||
     (iter < MinIteration) || (MaxEnsembleSize*elapsedTime(&timer) < length)));
    used_time += elapsedTime(&timer);
    fwrite(best_w, no_wts*sizeof(real), 1, wgtf);      /* append net to wgtf */
    if (used_time<length || ensembleSize<MinEnsembleSize) {   /* add if time */
      for (i=0; i<no_wts; i++) w[i] = best_w[i];
      fprintf(stderr, "%3d %9.4f %9.4f %7d    %7d\n", ++ensembleSize,
                                 cost(train, 0), best_perf, best_iter, iter);
    }
  }
  ⟨make predictions and write them to file 4b⟩
} ◇
```

Macro referenced in scrap 6d.

The usage is: `mlp-ese-1 instance-number [[-]cpu-time [seed]]`. The `instance-number` refers to the suffix of **Delve** data files. The `time` is the amount of cpu time (in seconds) allowed by the training procedure. Negative values are interpreted as times per training case. The default time is 1.875 seconds per training case. A seed for the random number generator may be specified; if omited a seed based on time and date will be used.

⟨parse command line 4a⟩ ≡

```
length = -1875;          /* default length is 1.875 secs per training case */
time(&seed);                                      /* chose a seed based on time */
no_inp = no_tar = -1;    /* defaults for "unknown" in case of no arguments */
if (argc < 2 || argc > 4 ||
    argc > 2 && ((length = 1000*atof(argv[2])) == 0.0) ||  /* length in ms */
    argc==4 && ((seed = atoi(argv[3])) <= 0)) {
  fprintf(stderr, "Usage: %s instance-number [[-]cpu-time [seed]]\n", argv[0]);
  exit(-1);
}
srand48(seed); ◇
```

Macro referenced in scrap 3.

The predictions are made by averaging outputs from the ensemble of trained networks. We allocate space for the predictions and accumulate predictions from all networks in the `wgtf` file and divide by the number of nets to get the average. Then the prediction file is opened and the predictions dumped.

⟨make predictions and write them to file 4b⟩ ≡

```
rewind(wgtf);                                  /* compute ensemble predictions */
pred = createMatrix(no_tar, test.num);
for (k=0; k<no_tar; k++) for (i=0; i<test.num; i++) pred[k][i] = 0.0;

for (j=0; j<ensembleSize; j++) {
  fread(w, (long) no_wts*sizeof(real), 1, wgtf);
  for (i=0; i<test.num; i++) {
```

```
    net(test.inp[i], output);
    for (k=0; k<no_tar; k++) pred[k][i] += output[k];
  }
}

for (k=0; k<no_tar; k++)                              /* use mean prediction */
  for (i=0; i<test.num; i++) pred[k][i] /= ensembleSize;
fclose(wgtf);

pred_file = openPredFile("cguess.%s", argv[1]);   /* point prediction file */
for (i=0; i<test.num; i++) {                           /* write predictions */
  for (k=0; k<no_tar; k++) fprintf(pred_file, "%f ", pred[k][i]);
  fprintf(pred_file, "\n");
}
fclose(pred_file); free(pred[0]); free(pred);
◇
```

Macro referenced in scrap 3.

Random splits of data into training and validation sets is done by randomly reordering the examples and using a leading fraction for training and the remainder for validation (**valid.num** already contains the desired set size). Random reordering is achieved by swapping each example in turn with a random other example. Since the examples are stored in a pointer to array of pointers structure, we only need to manipulate the pointers. Both inputs and targets must be swapped in the same way.

⟨generate random train/validation sets 5a⟩ ≡

```
for (i=0; i<train.num+valid.num; i++) {
  j = drand48()*train.num;
    tmp = train.inp[j]; train.inp[j] = train.inp[i]; train.inp[i] = tmp;
    tmp = train.tar[j]; train.tar[j] = train.tar[i]; train.tar[i] = tmp;
  }
◇
```

Macro referenced in scrap 3.

Training examples are loaded from `train.`n files and test inputs from `test.`n, where n is the instance number. The number of cases used for validation is computed and the validation set is set to point to this part of the training examples.

⟨load and partition cases 5b⟩ ≡

```
sprintf(df, "test.%s", argv[1]);                         /* read test inputs */
test.num = train.num = no_inp = no_tar = -1;      /* default for "unknown" */
loadExamples(&test, &no_inp, (i=0, &i), df, NULL);
sprintf(df, "train.%s", argv[1]);
loadExamples(&train, &no_inp, &no_tar, df, NULL);
if (length < 0) length *= -train.num;  /* training time was given per case */
valid.num = train.num/ValidateFraction;
train.num -= valid.num;         /* now train on only the training fraction */
valid.inp = &train.inp[train.num]; valid.tar = &train.tar[train.num];
◇
```

Macro referenced in scrap 3.

Open a binary file called **wgts**.n, where n is the instance number. This file will contain the weights of all the nets in the ensemble.

⟨open weights file 6a⟩ ≡

```
sprintf(df, "wgts.%s", argv[1]);
if ((wgtf = fopen(df, "w+b")) == NULL) {
  fprintf(stderr, "Could not open weight file %s for writing ...bye!\n", df);
  exit(-1);
} ◇
```

Macro referenced in scrap 3.

The number of hidden units in the net is chosen to be the smallest possible such that the network will have at least as many free parameters as there are training cases (when the cases for validation has been removed).

⟨create network 6b⟩ ≡

```
no_hid = ceil((train.num-(no_inp+1.0)*no_tar)/(no_inp+no_tar+1.0));
no_wts = (no_inp+1)*no_hid+(no_inp+no_hid+1)*no_tar;
fprintf(stderr, "Using nets with %3d hidden units and %5d weights.\n",
        no_hid, no_wts);
createNet();
best_w = (real *) malloc((size_t) no_wts*sizeof(real)); ◇
```

Macro referenced in scrap 3.

⟨define variables 6c⟩ ≡

```
char df[50];                                                /* file name */
long seed,
     length, used_time = 0;                   /* times are in milliseconds */
int  restart,    /* used with conj(); if 1 then conj uses steepest descent */
     i, j, k, l,
     iter, best_iter,     /* current and best iteration for early stopping */
     ensembleSize,                 /* counts number of nets trained so far */
     succ;
real *tmp,
     junk,
     perf, best_perf,   /* current and best performance for early stopping */
     **pred,                                      /* array of predictions */
     *best_w;            /* array of weights of best net in early stopping */
struct exampleset valid, test;
extern int TIMEOUT;  /* set to 1 when SIGVTALRM is caught; when time is up */
FILE *wgtf,                 /* file for storing ensemble of network weights */
     *pred_file;                   /* file to which predictions are dumped */
◇
```

Macro referenced in scrap 3.

"mlp-ese-1.c" 6d ≡

```
#include <stdio.h>
#include <math.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include "util.h"

#define MinEnsembleSize   3
#define MaxEnsembleSize  50
#define ValidateFraction  3
#define EarlyStopMargin   1.5
#define MinIteration    250
#define min(a,b) (((a)<(b))?(a):(b))

void createNet();
void destroyNet();
void randomizeNet(real magnitude);
real prior(real *dw) { return 0.0; }        /* we do not have a weight prior */
extern void initTimer(struct itimerval *timer, int time);
extern long elapsedTime(struct itimerval *timer);
real cost(struct exampleset ex, int wc);
void net(real *inp, real *out);
extern int conj(int *iter, int *epoch, int *restart, real *costvalue);

struct itimerval timer;
struct exampleset train;                             /* the training examples */

real *w,                              /* array [no_wts] of all network weights */
     **wi,   /* array [no_hid] of arrays [no_inp] of input-to-hidden weights */
     **wd,                          /* array of direct input-to-output weights */
     **wo,                                /* array of hidden-to-output weights */
     *output;                                      /* array of network outputs */
int  no_hid, no_inp, no_tar,        /* no. of hidden, input and target units */
     no_wts;                                           /* total no. of weights */
```

⟨main 3⟩
◇