IBM-MIT Sponsored Research Agreement
Status Report (TD-Gammon Project)

Final Report

1 Work Done on Rollout-Gammon

As was jointly decided early into the execution of this contract, the original and more ambitious plan for a rewrite of the TD-Gammon code into a modular, easily reconfigurable platform for research and experimentation with learning methodologies was abandoned for the shorter path of folding other separate developments to TD-Gammon into the parallel version to make a working demonstration system with those improved features. It was subsequently decided that due to the significant divergence between the code bases, the best way of achieving this goal was to parallelize from scratch the version with the new features. This goal has been accomplished as planned.

1.1 Overview of Functionality

The new system features parallel truncated rollouts and equity estimation, faster evaluation of boards for some of the players, an improved and parallelized doubling algorithm, and the ability to have several weight sets coexistent allowing the use of different networks for different purposes. However, this system is not modular or easily reconfigurable; using the different weight sets or networks is only allowed where I explicitly put in those features and nowhere else, and it would not be easy to for instance install a different move evaluator or play two arbitrarily configured players against each other.

As has been implemented in the other parallel versions, the new system includes decisionmaking by parallel rollout on any number of nodes available, maximizing expected equity. Decisions in the rollout games are made by one neural network, and the rollouts are truncated after a specifiable number of plies by a potentially different network (even with a potentially different network structure). The now-familiar dynamic confidence bound computation to allow the specification of a certainty that the best move is being selected to within some expected equity point loss has been implemented as well. As usual, this and all other players may be switched in and out of the system for benchmarking purposes.

One major motivation for the development of this new demonstration system was to correct what was identified as the main weakness of the prior systems, match play; with the TD(λ) learning and the rollout system added to that, game move decisions or the so-called "money play" had achieved satisfactorily strong performance. A significant component of that functionality is the performance in doubling situations. An improved doubling algorithm had been developed, but seemed to suffer from incorrect estimates of the true win and loss probability and equity estimates. That doubling algorithm was integrated into the parallel system including parallel equity estimation, to providing statistically accurate and unbiased estimates for use in the algorithm. This was implemented for the initiation of doubling as well as for accepting or declining an opponent's proposed doubling; both the amount of time spent computing each of these and the networks used to truncate the rollouts for each are separately adjustable.

1.2 Technical Detail

The implemented system can be found in `/farm/grg/rollout-demo/`. As described above, that code is a parallelization of the provided TD-Gammon version including the improved doubling formula.

The files which have been added or changed are: `bg.h`, `debug.c`, `dec.h`, `defines.h`, `endtbl15.c`, `enginit.c`, `eval.c`, `evaltd.c`, `getbbdbl.c`, `getbbmv.c`, `getsbdbl.c`, `getsbmv.c`, `gjtmain.c`, `mpiprof.c`, `net.h`, `ppm.c`, `randmove.c`, `stddef.h`, and `resign.c`. To be complete, the source files which have not changed are: `getmvs.c`, `evaltd3p.c`, and `evalpub.c`.

To make system configuration easier, most of the parameters controlling the computation (both parameters used as data and switches controlling what functionality is compiled into the code) have been grouped together into the header file `defines.h`. These parameters are described below; similar documentation will be put in the source code. Where this document disagrees with the source code, consider the source more current.

1. Parameters Guiding Computation:

   `ROLLOUT_SKILL` Skill level for rollout player (a number, 1-6: 1-3 are pubeval with decreasing noise added; 4 is TD 1-ply, 5 is TD 2-ply, and 6 is TD 3-ply). As in TD-Gammon.

   `TRUNCATE_MAXPLY <n>` Plies before truncating during all parallel rollout activity; set to `MAXINT` for no truncation. 1 ply is one move by the player to move next. Lower is more efficient but less accurate.

   `MINGAMES <n>` Minimum number of games to be played for each candidate move (before choosing or discarding any); used to make it more likely that any given situation (i.e., dice roll) is considered at least one (but makes no guarantees about the distribution of the `n` situtations examined). Lower is more efficient, but more risky. This basically second-guesses the confidence bound computation, overriding it when it tries to kill an option before the specified number of games have been played for it.

   `MPD_PASSES <n>` Number of passes to make adjusting maxPointDrop (0 for no adjustments, meaning when any move is eliminated it is never considered again); this is an efficiency parameter.

   `USE_BEAROFF_TABLE` If this symbol be defined, rollout games are terminated using a precomputed bearoff table. This makes computation faster and should be more accurate, but curiously seems to degrade the accuracy.

- `TRUNCNET_MOVE <net>` Name of a net to use for making move decisions during rollouts for ordinary play (top-level move decisions). The `<net>` can currently be `nn40`, `nn80`, or `nn160` corresponding to the 40-, 80-, or 160-hidden unit neural network weight sets currently being read into the program.
- `TRUNCNET_DOUBLE <net>` Name of a net for playing during rollouts to determine equity for doubling decisions.
- `TRUNCNET_ACCEPT <net>` Name of a net for playing during rollouts to determine equity for accepting opponent's double.
- `TRUNCNET_RESIGN <net>` Name of a net for playing during rollouts to determine equity for resigning.
- `TIME_DOUBLE <secs>` Seconds to spend computing an equity when deciding whether to double.
- `TIME_ACCEPT <secs>` Seconds to spend computing an equity when deciding whether to either accept or resign.

2. Parameters Controlling Printing:

- `SHOWMOVES` Define to see the plays being made by the rollout player "at the top level," i.e., none of the internals of the rollout system are exposed. (used during `AUTOPLAY`)
- `MONITOR_BEST` To print out the evolution of the rollout's best move as it changes its estimation of the best.
- `SHOW_EVERYTHING` To see *every* move as it's made during rollout; produces on the order of tens to hundreds of megabytes of output for a single "top-level" game. Not recommended except for serious debugging.
- `SHOW_ALIVE_MOVES <n>` If defined, shows the number of moves still under consideration at a periodic interval (each global communication between nodes). If a number is specified as an argument, also shows at the same time all moves still under consideration if the total number of moves still under consideration is less than the argument.
- `PRINT_END_STATS` At the end of the rollout decision for a move, shows various statistics gathered from the rollouts on all top-level moves. Probably a little too much output for casual use.
- `PRINT_INT_STATS` Prints information like in `PRINT_END_STATS`, but does so at various points during the rollout. Only has effect if `MONITOR_BEST` is also defined. Use when debugging or examining a rollout in more detail.
- `GREGS_PRINT` To show an ASCII graphics picture of the board state any time a board is displayed.
- `PRINT_BGPROB` To show backgammon probability numbers (for debugging).

3. Other Parameters:

`FAST_SIGMOID` Use a fast (RS6000) version of the sigmoid function for a neural net (uses a lookup table).

`MPI_PROFILE` Time (selected) MPI routines.

`CHECK_NAN` To insert checks to see if any numbers are invalid. Slows code down, but very occasionally useful for debugging serious math problems.

Note that in contrast to the modular system as designed, it isn't easy in this system to swap in other players (it's set up to allow any of the above weight sets to be replaced with other weight sets independently, but the *player* (e.g., whatever uses those weight sets to make decisions) cannot be easily changed. Also, other reconfigurations aside from the functions mentioned above are nontrivial.

1.3 Parameter Tuning

As we were always aware, the optimal parameter settings are a function of the system architecture and of the demands of the system.

Here are presented a set of parameters which have been experimentally determined to work well for a range of system configurations, from about 32 to 128 dedicated nodes on an SP2. (These will certainly work for other machine configurations, such as 16 nodes of an SP2, but the farther you get from the designed system, the less optimal these settings will be.)

It has been determined that a good set of parameters to use is:

- Use a linear net with the full feature set as the player in the rollouts
- Use the standard 80 hidden unit neural net weight set for all truncation
- Truncate rollouts at 7 ply
- Take 4 or 5 steps in the point drop progression
- Use a 'confidence parameter' setting of around 0.01
- Set `MINGAMES` to 18
- Allow a maximum of 10,000 games to be rolled out
- Take 1 or 2 seconds to decide whether to double, 5 to decide to accept

1.4 Running the System

Within the past year, it seems that the SP2 systems have standardized on using a new IBM product, LoadLeveler, to manage jobs. With this tool, most of the complexity of running a program is pushed into a single configuration file (see Fig. 1 for an example). As a quick primer to this system, LoadLeveler pays attention only to the lines which begin with the characters "`# @`" (so an easy way to comment out a line is to remove the `@`). Once this file represents your wishes, you run it via `llsubmit <config-file>`. Other interesting commands are `llq` to see the queue of jobs submitted, and `llstatus` to see what the hardware is doing. Apparently this does not support interactive jobs, and the correct way to do that is to use the old POE entry point as before.

1.5 Considerations/Difficulties

We seem to be past the Age of Great SP2s. In the latter half of 1997, two of the largest machines accessible all but disappeared. The Argonne National Labs machine (run by the Mathematics and Computer Science Division), a 128-node SP1/SP2 was in fact decommissioned in September 1997. We have used many cycles on that machine for large-scale development and testing. Even more of a tragedy is that at the same time, the Cornell Theory Center's 512-node SP2 was stripped down to 160 nodes with little or no notice or explanation when the NSF Supercomputer Centers Program was discontinued. Not only was our official allocation immediately reduced due to the lowered availability of cycles, the machine (which previously had almost half the nodes free at any given time) became significantly overloaded. Finally, later in 1998 the "home machine" SP2 on which we did nearly all our development, `titan` a.k.a. `curie`, was taken off-line for dedicated usage (and all but 4 non-interactive nodes are still off-line as of this writing).

While we had hoped that the end of the era of large SP2s were accompanied by an exodus of users away from the existing machines, leaving more cycles for our usage, in fact it seems that like us, too many remaining users clung to the SP2 platform (probably not wanting to abandon the significant development effort which had gone into making an application "tuned" for maximum performance on their chosen platform). As a result, most of the SP2s remain fairly inaccessible for significant computation.

2 Retraining on Rollout Results

2.1 Motivation

I initially developed the rollout system while investigate techniques for using parallel systems to improve the training time of neural nets; I was looking for some way around the month and a half training time for each neural network. The rollout system as developed requires a (set of) fully trained neural network(s) and does not modify them. However, it does spend tremendous effort to estimate the values of states (with respect to the policy defined by the rollout player) – which is exactly what the neural nets are attempting to estimate, meaning the rollout results form a high-quality training set for the very neural nets used to generate them. This allows the possibility of retraining the nets based on the rollout results.

Of course, the goal of such retraining would be to improve the prediction ability of the nets, and in the end improve the play of the system. The theoretical justification for the plausibility of this is that we would be performing an explicit step of policy iteration to improve the underlying policy. Note though that while policy iteration is guaranteed to improve the (worst case) error on every step, this guarantee requires that no information about the policy is lost; but with a neural network representing the policy, there is no

```
# Running rollout under LoadLeveler on the SP2
# template last modified GRG FEB99
#
# Run this using "llsubmit <this-file>"

########################## Things I might want to change:

## node min,max
# @ node = 4

## default directory is current working directory
# @ arguments = rollout-demo -euilib "us"
# @ input = input-autoplay

## note: it overwrites these files
# @ output = run.log
# @ error  = error.log

## staying out of others' hair
# make sure it's dead, jim.  hh:mm:ss.
# @ wall_clock_limit = 03:00:00

########################## Things which probably don't change:

# @ job_type = parallel
# @ job_name = rollout
# @ tasks_per_node = 1

## might share the adapter? try IP?
# @ network.MPI = css0,not_shared,US

## when to send me email
# @ notification = error
#  notification = complete

## not_shared for time-critical things (demos)
# @ node_usage = shared

# @ checkpoint = no
# @ restart = no
# @ executable = /usr/bin/poe
# @ queue
```

Figure 1: Sample job configuration file for LoadLeveler

guarantee that information is preserved (and indeed there is almost a guarantee that it is not).

2.2 Description, Results

The rollout system, in the course of making a move decision, estimates the equity of every possible move by playing many games out for each. Statistics are gathered about the results for each, including the expectation (mean equity) and a confidence/variance measure (expected bounds on the error). The expectation values are used as training labels for the position rolled out; that is, in retraining the neural net is adjusted to try to make its output closer to the rolled out equity value. To make the policy iteration more explicit (and make organizing the computation simpler), the complete set of expectation values is collected first, and then training is performed.

2.2.1 Linear Net Experiments

Proof-of-concept experiments were performed using a simple system: a linear (no hidden layer) neural net as both the player for the rollouts and the net to retrain. This choice was made in consideration of the amount of computational effort needed to perform this task. For reference, the nets we use are typically trained on 100 million positions, taking a month. Each rollout game uses thousands of position evaluations (and thus thousands of network evaluations), and each position should be rolled out 100 - 1000 times. Clearly computation limits the number of positions we can examine, but the hope is that since each has a high-precision equity for a training value (as opposed to the win/loss binary result from a reinforcement learning sequence in standard training) many fewer positions would be needed for training.

A data set of 1,750,000 positions (generated in 2,000 hours of computation, involving on the order of 100 billion position evaluations) was generated by playing games and rolling out each candidate position 1000 times. Using this database, the linear neural net (with only a couple hundred free parameters) was retrained.[1] Unfortunately, no noticeable improvement in the performance of that neural net was observed; instead, a slight degradation of quality was observed.

Performance results reported are all comparisons over 100,000 games against Gerry's public domain program "pubeval." The players all have absolutely no backgammon features; the input vector is a raw board. The initial linear net (not pubeval, but td-trained) scores 0.001 ppg (points per game) over pubeval with a standard deviation of 0.0008 ppg, so the initial linear net and pubeval should be considered to be at roughly the same strength (note also that all other ppg numbers reported here have a similar stdev). Retraining the same linear net on the rollout database scores -0.092 ppg against pubeval. A slight degradation is reasonable because even 1.75 million positions is a small number compared to the number of total positions, so we only have a very sparsely covered input space.

[1] *Retraining* refers to training a new weight set using the given weight set as an initial condition. This is done to bias the resulting weight set to be closer to the original weight set in hopes of facilitating the "information transfer" between the stages of policy iteration mentioned earlier.

	Network	ppg vs. pubeval
Linear Rollouts	base linear player	0.001
	base linear player retrained on database	-0.092
	new linear player trained on database	-0.129
	5HU player trained on database	-0.014
Nonlinear Rollouts	base 5HU player	0.185
	base 5HU player retrained on database	0.184

Figure 2: Performance results playing the players reported on here against pubeval. Numbers quoted are in points per game. Each number is the average over 100,000 games, yielding a standard deviation on each figure of 0.0008.

Since is it possible that the lack of improvement results from the network weight set already being stuck in a local optimum, I trained a new net from scratch (randomly initialized) on the same database. The results are slightly worse, at -0.129 ppg, indicating that in fact we do benefit from *re*training the initial network, and that "information transfer" between these steps of policy iteration is taking place.

One possible reason for the observed lack of improvement could be that the linear neural net was the limitation in the system, rather than the training; i.e., the local maxima of the simple linear system are roughly equivalent in quality, and there is no room for improvement.

To address this possibility, I also trained a 5 hidden unit neural network on the same database. The results here are still worse than the original player, scoring -0.014 ppg against pubeval. This experiment of course does not benefit from policy iteration's "information transfer" (the nonlinear net had no information about the linear net used in the rollouts), and there is a potential mismatch of architecture because we are training a 5 hidden unit nonlinear net to model the value function of a linear network's policy. The logical next step is to train it on the value function of a nonlinear network's policy.

2.2.2 Nonlinear Net Experiments

In the next phase, a 5 hidden layer neural network was td-trained (from scratch, not using anything from the above mentioned 5 hidden unit network) on around 100 million positions. The resulting player was used in a similar procedure as above, though with considerably more computational effort. A database of 250,000 positions was generated (this smaller set represents more than twice as much net computational effort as the set generated by the linear rollouts, and its smaller size is due only to computational constraints), and the net retrained on this database.

The starting player scores a quite respectable 0.185 ppg against pubeval. After training on the database, the net achieves 0.184 ppg, again showing a slight degradation in playing ability after retraining (although since these figures are about a standard deviation away from each other, they should be considered equivalent; still, no improvement was achieved).

2.2.3 Future Work

The next step to pursue in this work would be to burn huge numbers of cycles and attempt the same on a larger neural network, e.g., a single-hidden-layer neural net with 20 or 40 hidden units. Alternately, expectation values from archives of previously rolled out positions could be milked for training data.[2]

2.2.4 Discussion

As mentioned above, there could be several possible explanations for the degradation in playing ability. First, the training sets may be too small. Second, the function approximators (neural nets) may not have enough representational power to model the value function well; and the training procedure only seeks to minimize error on the value function over the database, without regard to the resulting playing ability (while the td-training is much more focused on the outcome). Further, the training set is necessarily biased towards the positions which the td-trained player wants to play (and all their immediate successors). The rest of the state space is not represented at all here; there is no feedback mechanism here for discovering and correcting mistakes learned by the net in the course of training (whereas in td-training mistakes are often immediately exploited and thus corrected).

While these practical arguments against the procedure attempt to explain the experimental results, the arguments presented as Motivation still hold, and there may well be gains to be had if the procedure could be made to work; though the evidence is of course discouraging. Moreover, the computational effort expended so far is substantial, and no improvement has been achieved. For instance, it actually took about twice as much time to train the 5 hidden unit network to convergence on the (comparatively small) database than it did to td-train the network from scratch, not counting the enormous amount of computation required to generate the database in the interim (which dwarfs both by a couple orders of magnitude) – and both the database generation and retraining time will scale linearly in the size of the data set, which would probably benefit from being an order of magnitude larger. While it is clear that we will not achieve faster training using this method, the computational effort would be worth it if the ending result were a better player; but at this time there is no evidence that such would result.

[2]Such an archive is on the machine `hicl1.watson.ibm.com` in the directory `/usr/gammon/`, which has been unavailable to me for now several months and which it seems may be down for the count.