CS 545 Project
ParamNN: A Parameter Prediction Neural Network

Dave Matthews

December 16, 2009

Contents

1 Introduction

In this project, we use a neural network [1] to predict the best parameters for a reinforcement learning (RL) algorithm [2]. The reinforcement learning algorithm incorporates a neural network to solve a simple problem and employs several parameters that affect the performance of the algorithm. Predicting these parameters becomes a second problem to solve.

Section 2 introduces the reinforcement learning problem used in the experiments. Section 3 describes the ParamNN approach to algorithm parameter prediction. Section 4 highlights the results obtained using the ParamNN approach. Section 5 summarizes the findings

2 Reinforcement Learning Problem

The reinforcement learning problem involves pushing a marble to a given location on a one-dimensional track [3]. The position on the track is bounded by 1 on the left and 10 on the right. The goal is to reach a certain position on the track. We will use a goal of 5 for these experiments.

The state of the marble is given by its position on the track and its velocity. The initial position is chosen randomly and the initial velocity is zero. One of three actions may be applied to the marble in each time interval, a push to the right, a push to the left, or no push. The reinforcement learning algorithm determines the appropriate action for each state based on a series of samples used to update the neural network.

```
epsilonRate <- exp(log(finalEpsilon)/nReps)
net <- makeNN(ni,nh,length(actions),matrix(c(1,10,-5,5),2,2))

for (reps in 1:nReps) {
  samples <- getSamples(net,nSteps,epsilon)
  net <- updateNN(samples$X, samples$R, samples$Q, samples$Y, samples$AI,
                  net, gamma=gamma, lambda=lambda, fPrec=1e-8, nIter=maxIterations)
  epsilon <- epsilonRate * epsilon
}
```

Several parameters control the behavior of the algorithm and the updates to the neural network. These parameters include discrete values (nh, nReps, nSteps, maxIterations) and continuous values(lambda, gamma, finalEpsilon).

- nh, the number of hidden units,
- nReps, the number of times the main loop is repeated,
- nSteps, the number of samples (time steps) to collect each time through the main loop,
- maxIter, the maximum number of iterations of the Scaled Conjugate Gradient algorithm allowed,
- lambda, the factor on the weight penalty in the hidden layer, with a value of 0 for no penalty,
- gamma, the discount factor on the sum of future reinforcements, greater than 0 and less than or equal to 1.
- epsilon, the final probability of taking a random action, between 0 and 1.

Tuning or predicting the algorithm parameters becomes an optimization problem. In many cases the algorithm parameters are determined using an ad hoc, manual approach based on experience. In other cases empirical studies or parameter tuning algorithms are employed [4].

The use of discrete and continuous parameters complicates the tuning effort. Discrete parameter values alone create a combinatorial problem. The continuous parameter values create an infinite number of parameter combinations. Many methods work only with discrete parameters. Continuous parameters are converted to discrete values by sampling over a range for use in these methods.

In this paper we will employ a neural network to identify promising values for both the discrete and continuous algorithm parameters for our reinforcement learning problem.

3 ParamNN

ParamNN uses a simple non-linear regression neural network to tune algorithm parameters for a given problem. The `tuneParameters` function uses information gathered from previous executions of the algorithm to predict parameters that may produce a better result. The results from the previous executions are the input values to the neural network, while the corresponding parameters are the output values from the neural network. The ParamNN neural network attempts to learn a function to predict the parameter values from the results they produced.

The results may contain one or more values for each parameter set and represent values to be minimized, such as root mean squared error. The parameters may contain both discrete and continuous values. The function returns the set of parameters predicted for results that are better than the current best or minimum results. The returned parameters are continuous values. The calling routine must round discrete parameters and check the bounds on all parameters.

```
defFactor      <- 0.95   # amount of improvement to target
defHidden      <- 4      # extra hidden units
defIterations <- 50      # maximum gradient descent iterations (assumes SCG)
defLambda      <- 1e-6   # some penalty to encourage smaller parameters
defFprecision <- 1e-6
defXprecision <- 1e-6

tuneParameters <- function( parameters,
                            results,
                            factor = defFactor,
                            hidden = ncol(parameters)+ncol(results)+defHidden,
                            nIterations = defIterations,
                            lambda = defLambda,
                            fPrecision = defFprecision,
                            xPrecision = defXprecision)
{
  ### construct a net from the previous results and corresponding parameters
  nn <- makeNNR(results,parameters,
    hidden,lambda,
    fPrecision=fPrecision,xPrecision=xPrecision,nIterations=nIterations)
  ### determine parameters for results slightly better than previous best
  bestsofar <- matrix(apply(results,2,min),1,ncol(results))
  newParm <- useNNR(nn,bestsofar*factor)
  newParm$Y
}
```

We chose to use a simple error function as the result for this example. The error function computes the root mean squared error of the final positions of the marble based on *ntrips* initial positions distributed evenly along the track $x0$ with no velocity. The final position is determined by applying the policy learned by the neural network for *nsteps* time steps.

```
error <- function(net, nsteps=500, ntrips=50, goal=5) {
  err <- NULL
  x0 <- seq(1,10,length=ntrips)
  for (trips in 1:ntrips) {
    s <- c(x0[trips],0)
    for (step in 1:nsteps) {
      eg <- epsilonGreedy(net,s,epsilon=0)
      a <- eg$a
      s <- nextState(s,a,goal)
    } #step
    err <- rbind(err, abs(s[1]-goal))
  } #trips
  sqrt(mean(err^2))
}
```

3.1 Randomly Generated Seed Parameters

The ParamNN approach requires a sample of previous results which are obtained from executions using randomly generated parameters. The following R code generates $nRand = 50$ sets of random parameters and captures the resulting error for each parameter combination. The parameter ranges are fairly broad to ensure coverage of the parameter space. We fixed the total number of samples at $nSamples = 200000$ and generate a random value for the number of repetitions. We then determine the number of steps per repetition.

```
parm <- NULL
rmse <- NULL

### executions with randomly generated parameters to seed parameter tuning
for (samp in 1:nRand)
{
  # random parameters
  nh                <- floor(runif(1,2,21))
  nReps           <- floor(runif(1,50,4001))
  nSteps          <- floor(nSamples/nReps)
  N               <- rep(nSteps,nReps)
  maxIterations   <- floor(runif(1,5,51))
  lambda          <- runif(1,0,1e-2)
  gamma           <- runif(1,0.5,1)
  finalEpsilon    <- runif(1,0,0.5)
  # train & test
  net   <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  err   <- error(net, 1000)
  # capture results and corresponding parameters
  parm  <- rbind(parm, c(nh,nReps,nSteps,maxIterations,lambda,gamma,finalEpsilon))
  rmse  <- rbind(rmse, err)
}
```

3.2 Predicted Parameters

In the next phase, we applied the `tuneParameters` function $nTune = 25$ times to iteratively predict better parameters based on the previous results, both random and tuned. After tuning the parameters, we employed a jitter factor of 5% to prevent premature convergence of the parameters and bounded the parameters to a valid range. We excluded the *nSteps* parameter from tuning since it is determined from the *nReps* parameter. For this study we chose to tune parameters based on the 10 best results from previous random and tuning executions.

```
jf <- 0.05 # jitter factor
for (samp in 1:nTune)
{
  best <- order(rmse)[1:10]                # best ten
  # tune parameters based on previous best results
  np            <- tuneParameters(parm[best,-3,drop=FALSE],rmse[best,,drop=FALSE])
  # jitter and bound parameters
  nh            <- boundParameter(round(jitter(np[1],amount=abs(np[1])*jf)),2,20)
  nReps         <- boundParameter(round(jitter(np[2],amount=abs(np[2])*jf)),50,4000)
  nSteps        <- floor(nSamples/nReps)
  N             <- rep(nSteps,nReps)
  maxIterations <- boundParameter(round(jitter(np[3],amount=abs(np[3])*jf)),5,50)
  lambda        <- boundParameter(jitter(np[4],amount=abs(np[4])*jf),0,1e-2)
  gamma         <- boundParameter(jitter(np[5],amount=abs(np[5])*jf),0,1)
  finalEpsilon  <- boundParameter(jitter(np[6],amount=abs(np[6])*jf),0,1)
  #train & test
  net   <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
  err   <-  error(net, 1000)
  #capture results
  parm  <- rbind(parm, c(nh,nReps,nSteps,maxIterations,lambda,gamma,finalEpsilon))
  rmse  <- rbind(rmse, err)
}
```

3.3 Best Predicted Parameters

Then we tested the parameter combination that produced the best error result, applying it $nTest = 25$ times to explore the variation of results with a single parameter set and determine if the results for that parameter set are consistently better.

```
best <- which.min(rmse[1:(nRand+nTune)])
for(samp in 1:nTest)
{
  #best parameter combination
  nh              <- parm[best,1]
  nReps           <- parm[best,2]
  nSteps          <- parm[best,3]
  N               <- rep(nSteps,nReps)
  maxIterations   <- parm[best,4]
```

```
    lambda            <- parm[best,5]
    gamma             <- parm[best,6]
    finalEpsilon      <- parm[best,7]
    #train&test
    net    <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
    err    <- error(net,1000)
    #capture results
    parm   <- rbind(parm, c(nh,nReps,nSteps,maxIterations,lambda,gamma,finalEpsilon))
    rmse   <- rbind(rmse, err)
  }
```

3.4 Jittered Best Predicted Parameters

Finally we explored the area near the best parameter combination, applying a 1% jitter factor $nJitr = 25$ times. The smaller discrete values (hidden units and maximum gradient descent iterations) remained constant but the other parameters varied slightly.

```
  jf <- 0.01
  for(samp in 1:nJitr)
  {
    # jitter the best parameter combination
    nh              <- boundParameter(round(jitter(parm[best,1],amount=abs(parm[best,1])*jf)
         ),2,20)
    nReps           <- boundParameter(round(jitter(parm[best,2],amount=abs(parm[best,2])*jf)
         ),50,4000)
    nSteps          <- floor(nSamples/nReps)
    N               <- rep(nSteps,nReps)
    maxIterations   <- boundParameter(round(jitter(parm[best,4],amount=abs(parm[best,4])*jf)
         ),5,50)
    lambda          <- boundParameter(jitter(parm[best,5],amount=abs(parm[best,5])*jf),0,1e
         -2)
    gamma           <- boundParameter(jitter(parm[best,6],amount=abs(parm[best,6])*jf),0,1)
    finalEpsilon    <- boundParameter(jitter(parm[best,7],amount=abs(parm[best,7])*jf),0,1)
    #train&test
    net    <- trainIt(nh,nReps,N,maxIterations,lambda,gamma,finalEpsilon)
    err    <- error(net,1000)
    #capture results
    parm   <- rbind(parm, c(nh,nReps,nSteps,maxIterations,lambda,gamma,finalEpsilon))
    rmse   <- rbind(rmse, err)
  }
```

The results of these tests appear in the next section.

4 Results

A maximum RMSE of 5.0 occurs when the learned policy pushes the marble completely to the right resulting in a final position of 10. An RMSE of 4.0 occurs when the learned policy pushes the marble completely to the left resulting in a final position of 1. Table 1 shows the 10 best results obtained across the 4 different groups of trials. A pair of random results appear in this list, one of which is near some of the parameters for the best parameters found. The tuning process contributed two sets of parameters to the list, including the best found during all of the trials. The resulting trials using the best and jittered best parameters did not improve upon the parameters found during tuning. There appears to be a close correlation of the parameters across most of these results.

Figure 1 shows the RMSE results of the individual trials and means for the four different groups of trials: randomly generated parameters, tuned parameters, best parameters, and jittered best parameters. The top graph uses a normal vertical scale, the bottom graph uses a logarithmic vertical scale to highlight the differences between the smaller error values.

The randomly chosen parameters consistently produced poor results with a few exceptions that were worth exploring. We initially used all of the random trials to train the neural network, but found it only

group	trial	rmse	nh	nReps	nSteps	maxIter	λ	γ	ϵ
tuned	68	0.016961	6	1289	155	7	0.00029587	0.87368	0.19973
best	97	0.030167	6	1289	155	7	0.00029587	0.87368	0.19973
random	25	0.036914	5	1339	149	6	0.00045731	0.89752	0.27850
jittered	104	0.054504	6	1295	154	7	0.00029686	0.88211	0.19994
best	85	0.056382	6	1289	155	7	0.00029587	0.87368	0.19973
best	91	0.056743	6	1289	155	7	0.00029587	0.87368	0.19973
jittered	121	0.074697	6	1293	154	7	0.00029436	0.87327	0.19798
tuned	53	0.101039	9	1010	198	15	0.00032490	0.88994	0.18420
random	43	0.101637	14	350	571	47	0.00004915	0.95715	0.21234
best	83	0.105229	6	1289	155	7	0.00029587	0.87368	0.19973

Table 1: Ten best results obtained across all trials ordered by root mean squared error.

	mean	median	standard deviation
random	2.88	3.20	1.53
tuned	1.18	0.76	1.15
best	0.50	0.31	0.59
best jittered	0.58	0.42	0.57

Table 2: Mean, median, and standard deviation for each group of trials.

slightly improved results. A second attempt using only parameters that produced results better than the mean did somewhat better. We finally settled on the approach using the parameters associated with the top ten results to tune the parameters which performed much better.

The tuned parameters in the second group were jittered by 5% to prevent premature convergence on simpler problems. This approach may have a detrimental affect on the tuning in this case. In the future we may wish to reduce the parameter jitter, make it conditional on convergence, or eliminate it altogether.

The best parameters in the third group show the effect of the stochastic algorithm. Twenty five runs with the same parameters produced a range of results, none of which improved upon the original results. Other than a few outliers, the results were consistently better than those produced by the random parameters or during the tuning process.

The jittered best parameters in the fourth group explores parameter combinations near the best parameter combination. The results for the jittered best parameters exhibit a slight decrease in performance. While the difference is statistically significant, it is unclear if it is a result of the parameter values or the sample size of 25.

Table 2 shows the mean, median, and standard deviation of the four different groups of trials. The tuned and best parameter values show significant improvement in the mean, median and standard deviations of the results. The jittered best parameter values show a slight decrease in performance compared to the best parameters, but the same standard deviation. The random median was greater than the mean, suggesting the best results may be outliers. The other medians were less than the mean, suggesting the worst results may be outliers. The box and whisker plot in Figure 2 confirms this and suggest the tuned and best parameters are more robust.

The following sections study the relationship of the individual parameters to the root mean squared error. The default parameters for the parameter study in assignment 7 [3] are also considered.

4.1 Hidden Unit

The default value for hidden units in the parameter study was 6. Figure 3 displays the root mean squared error for all trials by the number of hidden units on both normal and logarithmic scales. The randomly generated values range from 2 to 20 hidden units for the seed trials. The parameter tuning trials narrowed this range from 5 to 11 hidden units. The best and jittered best trials used 6 hidden units. While the normal

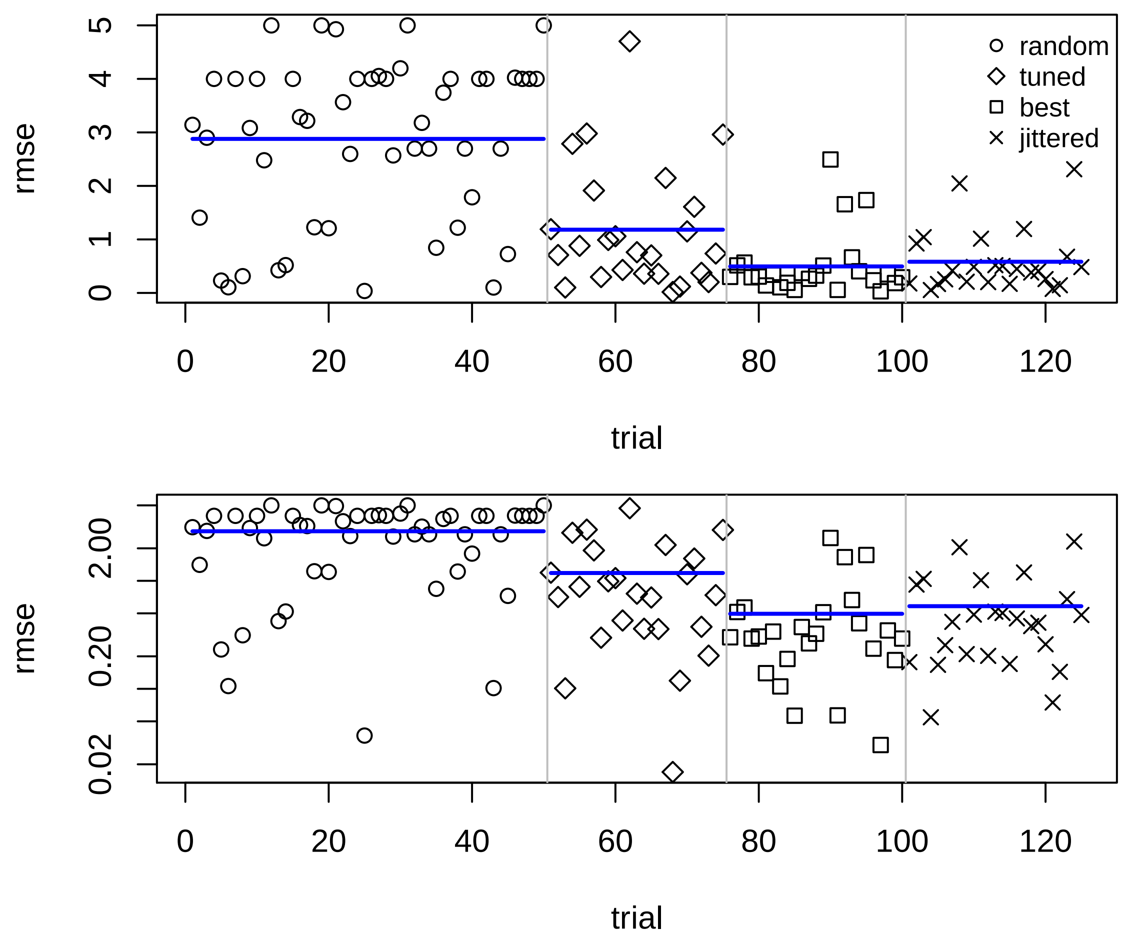


Figure 1: RMSE trial results and means obtained from random, tuned, best, and jittered best parameters on normal and logarithmic scales.

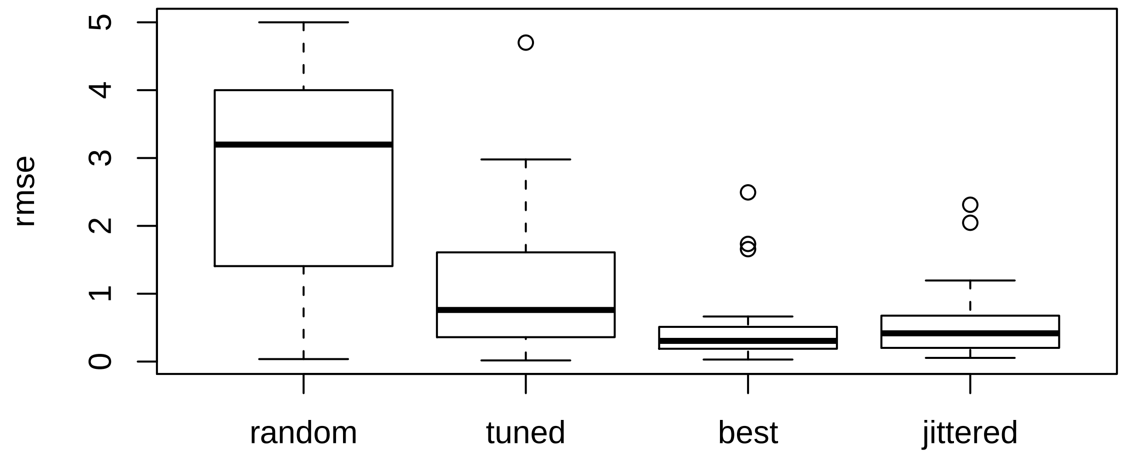
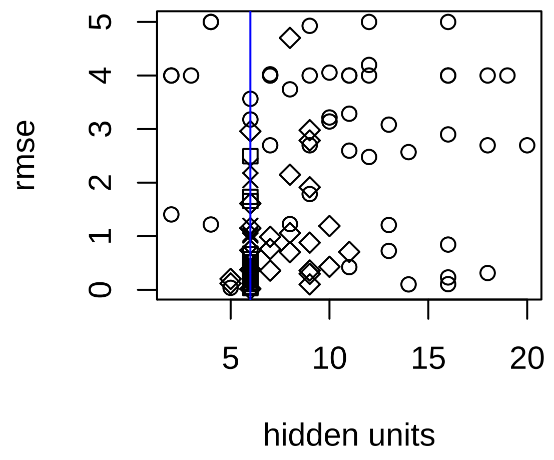


Figure 2: Box and whisker plot showing results for the four different groups of trials.



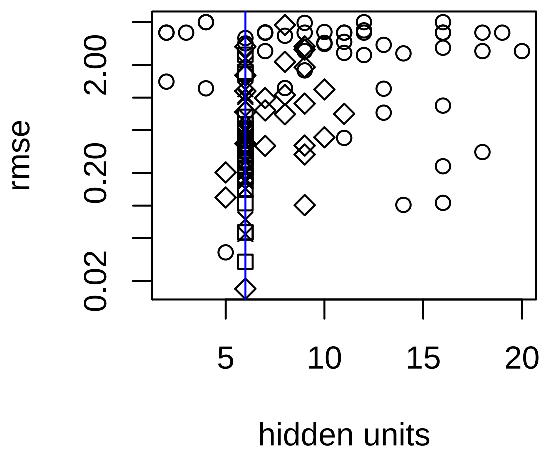


Figure 3: RMSE versus hidden unit parameters in trials on normal and logarithmic scales. The blue line is the value recommended in the assignment.

scale suggests a broader range of effective values for hidden units, the logarithmic scale clearly highlights 6 hidden units as the best parameter value.

4.2 Repetitions and Steps

The default value for repetitions of the main reinforcement learning loop was 100 and the default number of sample steps in each repetition was 2000 in the assignment. In these experiments we varied the number of repetitions and determined the number of steps using the formula $steps = 200000/repetitions$. This allowed us to keep the overall number of samples $repetitions * steps$ fairly constant, aside from rounding differences. Figure 4 displays the root mean squared error for all trials by the number of repetitions. We focus on the repetitions graph since it is the randomly selected or tuned parameters.

The randomly generated values ranged from 179 to 3987 repetitions for the seed trials, resulting in step values ranging from 50 to 1142 The parameter tuning trials narrowed this range from 722 to 1529 repetitions and 130 to 277 steps. The best trials used 1289 repetitions with 155 steps and jittered best trials ranged from 1277 to 1301 repetitions with 153 to 156 steps. The logarithmic scale graph suggests diminishing returns as the number of repetitions deviates from this range. It appears this problem benefits from more repetitions with fewer steps per repetition than the defaults in the assignment.

4.3 Maximum SCG Iterations

The default value for the maximum number of SCG iterations was 10 in the assignment in an attempt to prevent the neural network from overfitting the data. Figure 5 displays the root mean squared error for all trials by the maximum number of SCG iterations used to update the neural network. The randomly generated values ranged from 5 to 49 maximum SCG iterations for the seed trials. The parameter tuning trials narrowed this range from 5 to 18 iterations. The best and jittered best trials used a maximum of 7 SCG iterations to tune the neural network. While the logarithm scale graph shows this value superior to larger values, it performed poorly when tuned manually in the assignment, possibly due to other parameter settings.

4.4 Hidden Layer Weight Penalty λ

The default value for the hidden layer weight penalty λ in the neural network was 0.0 in the assignment. Figure 6 displays the root mean squared error for all trials by the λ value. The randomly generated values for λ ranged from $2.4e-05$ to $9.4e-04$ in the seed trials. The parameter tuning trials narrowed this range

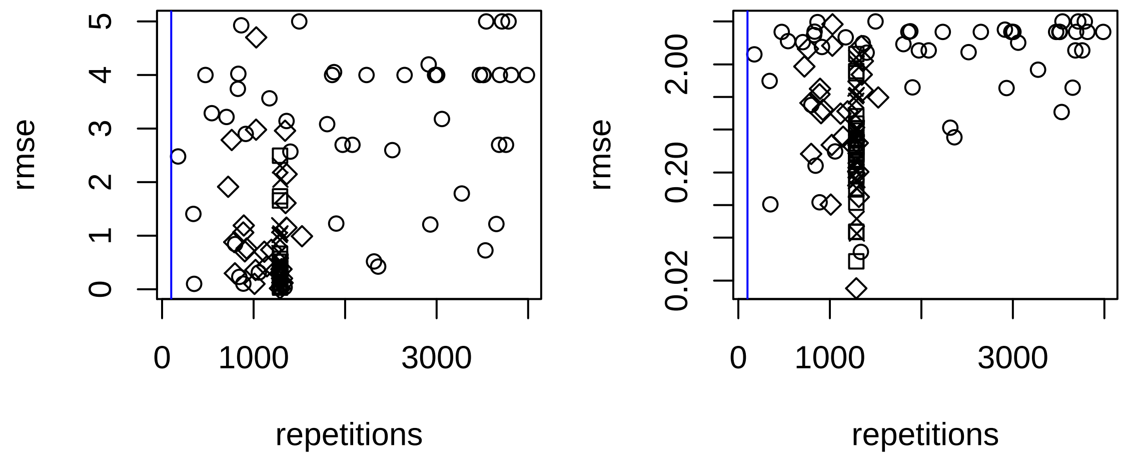


Figure 4: RMSE versus repetitions parameter in trials on normal and logarithmic scales. The blue line is the value recommended in the assignment.

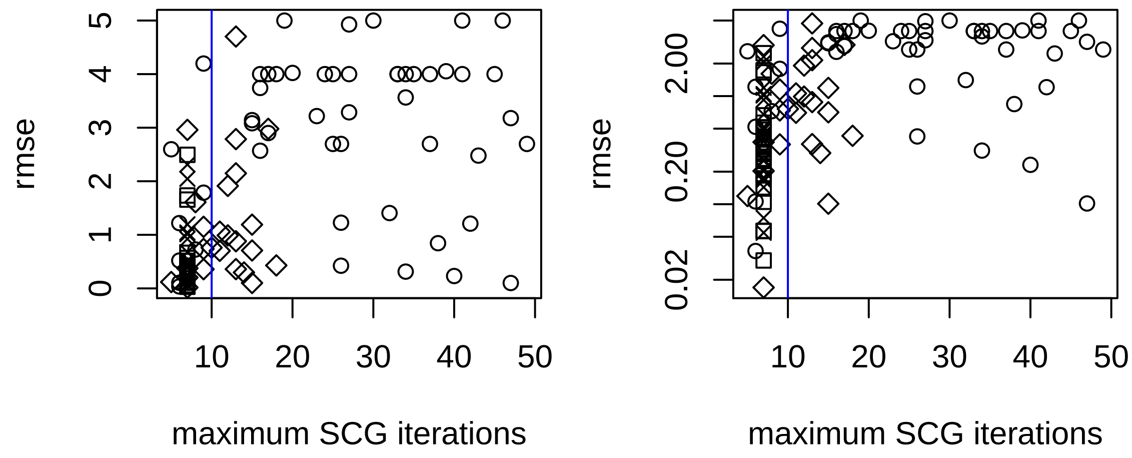
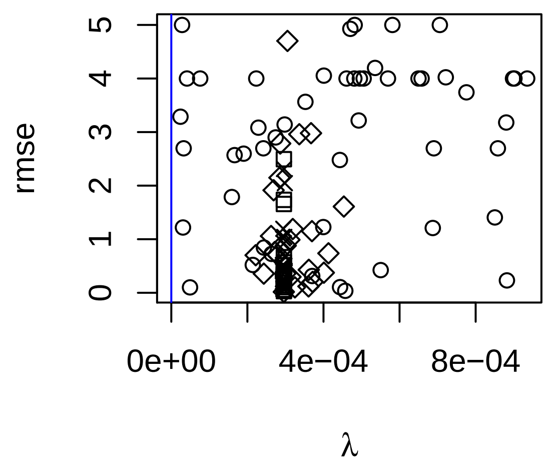


Figure 5: RMSE versus maximum number of SCG iterations parameter in trials on normal and logarithmic scales. The blue line is the value recommended in the assignment.



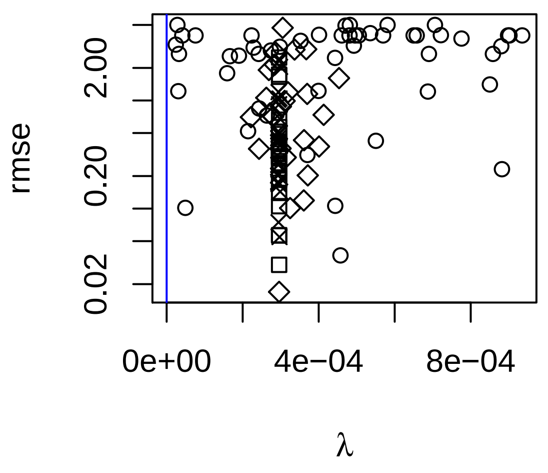


Figure 6: RMSE versus λ parameter in trials on normal and logarithmic scales. The blue line is the value recommended in the assignment.

from $2.2e-04$ to $4.5e-04$. The best and jittered best trials used λ values around $2.96e-4$. The logarithmic scale graph shows that performance drops off as the value deviates from the tuned value. Smaller values approaching zero did not perform as well when the other parameters were tuned.

4.5 Reinforcement Discount Factor γ

The default value for the reinforcement discount factor γ was 0.9 in the assignment. Figure 7 displays the root mean squared error for all trials by the γ value. The randomly generated values for γ ranged from 0.501 to 0.997 in the seed trials. The tuning trials narrowed this range from 0.837 to 0.960. The best and jittered best trials used a value near 0.874. The logarithmic scale graphs shows that performance drops as the value deviates from the tuned value.

4.6 Final Random Action Probability ϵ

The default value for the final random action probability ϵ is 0.001 in the assignment. Figure 8 shows the root mean squared error for all trials by the ϵ parameter value. The randomly generated values for ϵ ranged from 0.031 to 0.492 in the seed trials. The tuning trials narrowed this from 0.151 to 0.256. The best and jittered best trials used a value near 0.200. The logarithmic scale graph shows that performance drops as the value deviates from the tuned value. This value is significantly larger than the one used in the assignment, providing more exploration during the reinforcement learning process.

5 Conclusions

This project suggests a possible method for tuning algorithms with discrete and continuous parameters. We tuned six parameters (3 discrete and 3 continuous) for a reinforcement learning algorithm using our parameter tuning method. A seventh parameter one derived from one of the six. We were able to start with fairly wide ranges on the parameters which can be beneficial when tuning algorithms and problems where there is little prior experience. The best of the tuned parameters consistently produced better results than randomly selected parameters or parameters near the best. However, the results do show it is possible to randomly choose parameters that are near the best. We did not test them to determine how well repeated use of these parameters would perform.

The algorithm predicted some parameters significantly different than those recommended for the assignment, particular the number of repetitions/steps, λ and ϵ. The recommended parameters may account for

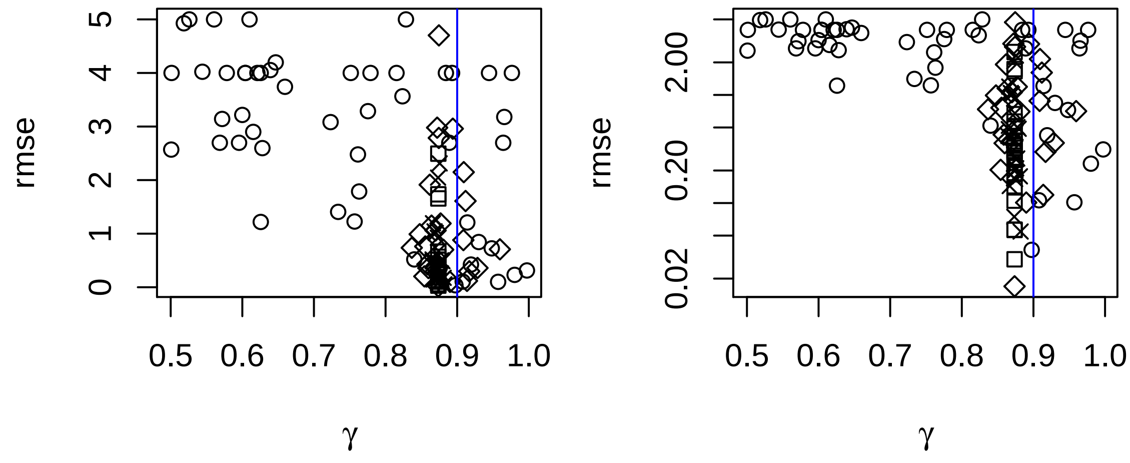


Figure 7: RMSE versus γ parameter in trials on normal and logarithmic scales.

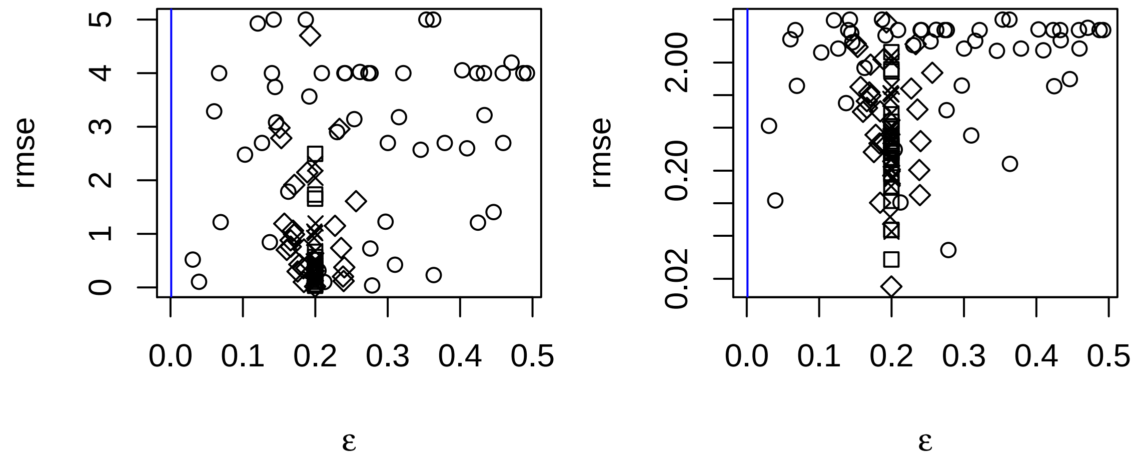


Figure 8: RMSE versus final ϵ parameter in trials on normal and logarithmic scales. The blue line is the value recommended in the assignment.

the variability in the results achieved on the assignment. It might be interesting to repeat the parameter study in the assignment using the predicted parameters as a starting point.

While the initial results are promising, further study of this parameter tuning method is necessary to determine its validity across a range of problems. For the problem we studied in this paper we need to increase the trial sizes to eliminate any issues of sampling size and coverage of the parameter ranges. We could also use additional result metrics and vary the goal.

We did not study the parameters associated with the neural network used in the `tuneParameters` function in this project. This becomes a second order tuning problem and warrants further investigation.

Some quick experiments with problems from the Machine Learning Repository [5] were also performed with promising results, but are not covered here. We need to thoroughly explore tuning parameters for other types of problems and other types of algorithms to understand the full utility of this method.

References

[1] Bishop, C.M., *Pattern Recognition and Machine Learning*, Springer, 2006.

[2] Sutton, R.S. and Barto, A.G., *Reinforcement Learning: An Introduction*, MIT Press, 1998.

[3] Anderson, C., *CS 545 Assignment 7*, `http://www.cs.colostate.edu/~anderson/cs545/assignments/assignment7.html`, Colorado State University, 2009.

[4] Hutter, F., Hoos, H.H., Leyton-Brown, K., and Stutzle, T., *ParamILS: An Automatic Algorithm Configuration Framework*, Journal of Artificial Intelligence Research, October 2009.

[5] Asuncion, A., Newman, D.J., *UCI Machine Learning Repository*, `http://www.ics.uci.edu/~mlearn/MLRepository.html`. Irvine, CA: University of California, School of Information and Computer Science, 2007.