Lent Update: Week 1

Alex Darch
Supervisor: Dr Glenn Vinnicombe

January 27, 2019

This is just an update report to read through before the next meeting since I couldn't fit everything onto the Technical Milestone Report - I also managed to solve a couple of major problems the morning before the TMR hand-in, but the code takes a couple of days to run fully so the results couldn't be added to the report. They are however, shown below with questions and thoughts too - just as a bit of a guide during the meeting, thought it might help me consolidate a couple of things and make the meeting more productive if you knew what was going on beforehand?

All graphs below are produced from the same setting:

- 18MCTS recursions at each step.
- The first 15 steps of training episodes are sampled from the MCTS-Improved action probability for exploration - greedy actions are then taken (this is done by suragnair: not sure if applicable to control?).
- 20 training episodes
- 15 greedy/test episodes, the mean loss is then compared with the mean loss of the best current set of test episodes by the current best policy - if mean*0.95 ¿ best mean then update.
- Neural Network Losses are trained in mini-batches of 8.

1 Training Example Graphs

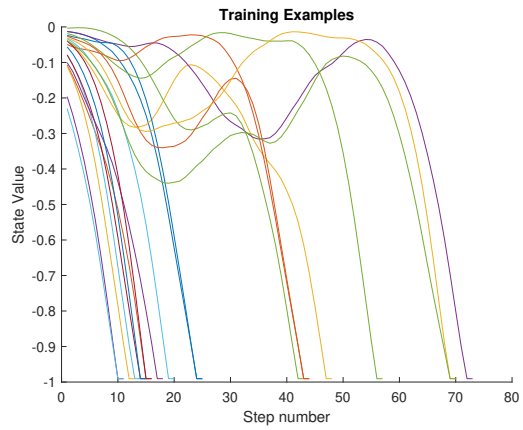

Figure 1: First round of training examples with untrained nnet.

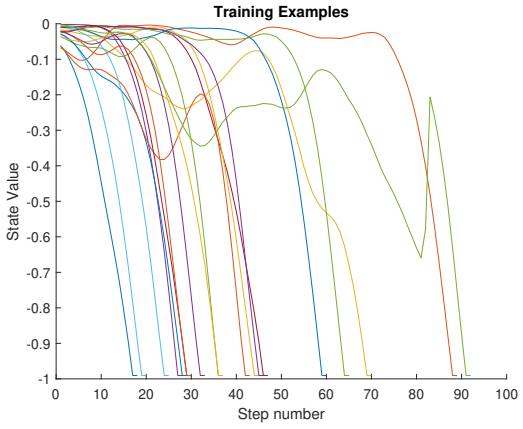

Figure 2: Second round of training examples with nnet trained on those in fig 1.

Clearly the nnet is learning as the average number of steps to failure increases. Looking at the first 15 steps, in fig 1 most just lose balance and fall, whereas in fig 2 a lot of the episodes still go down in those first few seconds, but they manage to pick themselves back up. In general it is only the ones that randomly stay up for the first 15 that go on to survive for 50+ steps - it seems to be very difficult to

regain balance for the nnet. This should be minimised with the chosen number of steps to "look ahead" for the value function of 16? Or perhaps it is the 18MCTS recursions stated that allow it to look forwards 15 or so steps, and after that it falls over.

The value and loss function are still defined as follows:

$$v_t = \frac{\sum_{t'=t,\tau=0}^{t+k,k} \gamma^\tau \mathcal{L}_{t'}}{\sum_\tau^k \gamma_\tau}, \qquad \text{where } \gamma^k < \frac{1}{20} \tag{1}$$

$$\mathcal{L}_t = -\frac{1}{2}\left[\left(\frac{x_t}{x_{max}}\right)^2 + \left(\frac{\theta_t}{\theta_{max}}\right)^2\right] \tag{2}$$

We can see that the MCTS is doing something by looking at a typical change in MCTS action-probs from the first set of training examples and the second set:

First Set

$[0.529, 0.470], [0.52, 0.48], [0.5, 0.5], [0.5, 0.5], [0.515, 0.484], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5]$

Second set

$[0.588, 0.411], [0.481, 0.518], [0.612, 0.38], [0.689, 0.310], [0.675, 0.324], [0.758, 0.241], [0.692, 0.307], [0.758, 0.241], [0.666, 0.333], [0.64, 0.36]$

Initially it is outputting a 50/50 left/right probabiltiy. It makes sense that this is 50/50 as the net hasn't been trained on what a good or bad state is yet, or whether it should go left/right in each position. The improvement is more likely due to the improved value function rather than improved action probabilities from the nnet - though I need to output the nnet actions from each to see how they differ (will need another 4 days!). A good thing to note here is that the nnet/MCTS isn't over-training and outputting that it should push left every time.

1.1 MCTS and State Representations

When MCTS is being performed there are two representations that are used: The 2D state and the 1D state. The 2D state is fed to the neural network to predict the action-probability and value for the state, and the 1D state is used to keep a record of what states have previously been visited - this was where I was going wrong before the TMR (I was just flattening the 2D state, however due to float comparison problems MCTS was never recognising states that had previously been visited). An updated function, shown in figure 3 is now being used and seems to be working well. I tried adding norm.cdf to give more credence to states closer to the centre as suggested in the TMR, but this increased the time to complete and episode by 6x... Possible solutions are either to do as the block of comments suggest, or just leave it as linear binning. I have not tested this yet.

```
def get_rounded_observation(self):
    # get the values to be within +-1
    obs = [self.state[0]/self.x_threshold,
           self.state[1]/self.x_threshold,
           self.state[2]/self.theta_threshold_radians,
           self.state[3]/self.theta_threshold_radians
           ]
    # what about calculating a list of bin edges at the start using inverse cfd (norm.ppf),
    # and then running each thing through until we get to said list idx then returning the index as the
    # norm cfd'd output? would remove the whole astype(int) things too so maybe even faster if small enough?

    # obs = norm.cdf(obs, scale=1/3)  # want +-1 to lie on the +-3std point -> scale down by 1/3
    # obs = np.rint(obs * self.mcts_bins).astype(int)  # norm.cdf returns np.array(), if don't round then 3.8 -> 3
    # return tuple(obs.tolist())

    obs = [int(round(elm*self.mcts_bins)) for elm in obs]
    return tuple(obs)
```

Figure 3: The current function to calculate the 1D state for MCTS. The number of bins is 25. It attempts to get each component of $[x, \dot{x}, \theta, \dot{\theta}]$ into a (-1, 1) range. Obviously the velocity components can go outside that, but the idea is just to keep the majority of observations within this bracket so not too many states are made, and multiply by $self.mcts\_bins$ to get the states are represented by integers so we can compare them easily in python.

2 Neural Network Training

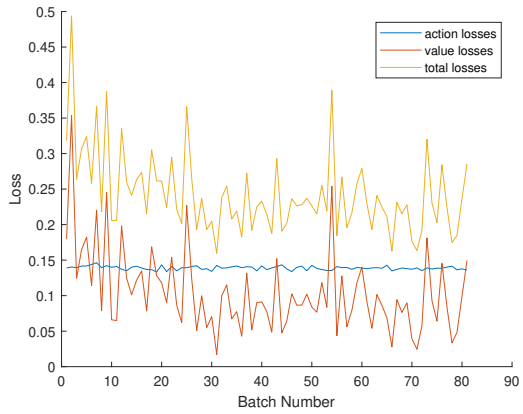

Figure 4: Loss from Neural Network Training1

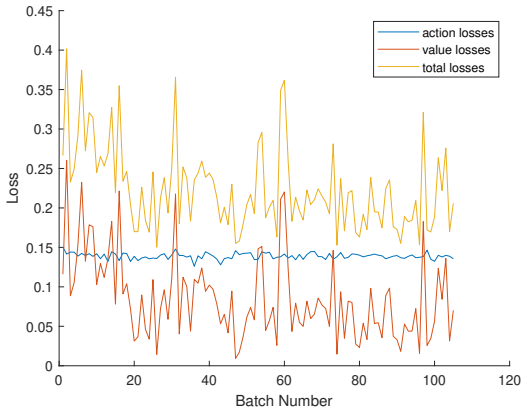

Figure 5: Loss from Neural Network Training2, trained on the examples in 2

Training the neural network seems only to improve the value-function loss even though they are roughly the same size? I tried playing with the pareto constant as suggested earlier and this seems to do almost nothing. One of the positives is that the network starts at a lower loss value than previously and ends up with a lower total loss.

Currently the 2D state representation is 50x50 and spaced linearly - as it had the same performance issues as the 1D state when using a gaussian c.d.f. I don't understand why this is happening, but even so it doesn't seem to be affecting the performance of the MCTS.

Another problem that I keep getting is the *very* unstable nature of the loss - which I think should be a smooth curve. This could possibly be caused by not having enough training examples, or it is a problem with the current 2D state representation. I tried increasing the mini-batch size to get it to be smoother and shuffled the examples, but neither of these helped. I have no real clue as to why this is happening and how to improve it, any ideas?

I am thinking of changing the 2D state to be something like the 1D state representation: Represent x vs θ on the 2D plane as a +1 and the $\dot{x}$ *vs* $\dot{\theta}$ as a +2/-1? and all other entries as 0. Images are normally represented as an array of 1-256 integer values, and in the AlphaZero model they represent each piece with integers rather than floats. A problem with this method is that the velocity values would be difficult to bin as they go from -inf to +inf, maybe binning them gaussian-ly will work. But in order to do this I can't use the norm.cdf method in SciPy, I'll need to write my own function as suggested in the comment on figure 3 so as not to get massive performance issues.

3 Greedy Episode Execution and Comparison

Unfortunately I overwrote the first greedy episode execution, so only the two below are shown. After the second training of the neural net, the model decreases in performance, possible over-fitting is occuring and causing the sharp drop in state loss at around 150. The variability in the episodes does decrease though.

On the bright side, clearly the code is working - episodes lasting to 900 steps is amazing for one pass, the data that I lost only had greedy episodes lasting up until 150 steps ish. Obviously this is part of the reason that running the code took so long, I haven't written the termination for each episode correctly yet.

4 Code Performance

4 days to run the code really annoying, this is my next priority. I have found a couple of solutions that I will try out over the next couple of days. The first of which is to use a module called cprofile to profile my code and work out what functions are taking the longest to run, then hopefully improve these.

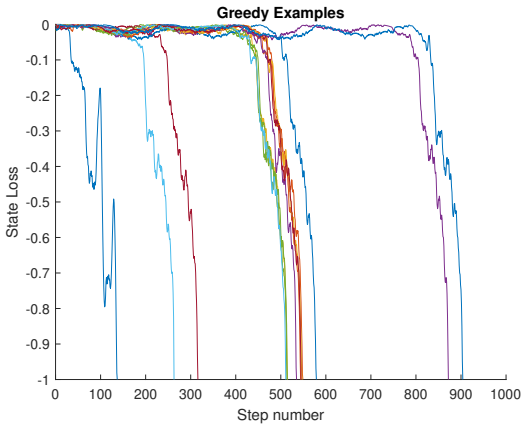

Figure 6: Greedy Episodes after the neural network had been trained once (fig 4).

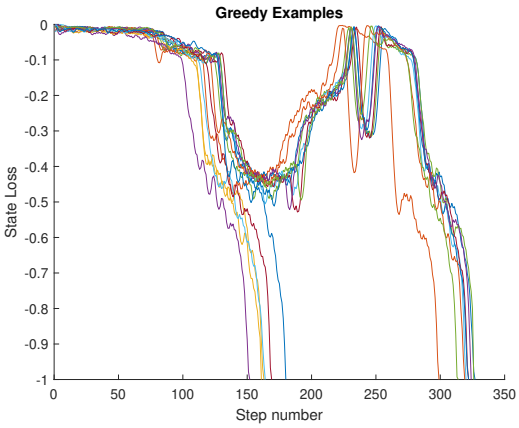

Figure 7: Greedy Episodes after the second training of the neural network (fig 5).

The next bit - which I had never heard of - was suggested by a friend. There are things around to speed up your code by turning the python interpreter into a C compiler. Cpython is the largest of these, but this requires rewriting a lot of code. PyPy is another, but that is a whole new compiler and it doesn't support pytorch from the looks of it. The best option that I found is a module called "Numba", this allows you to declare functions to be compiled ahead of time, it then works out what type (int/float) all your variables are, compiles it and then just lets to program run apparently improving the speed by up to 150x. I'll have a go trying this out over the next couple of days and come back with the results whenever we meet.

Also, GPU for pytorch isn't working - it's complaining about memory overloads so I need to work out why that is happening.