1 Ideas/Issues

Even if I train the obseravtion model and the controls at the same time succesfully, how do I motivate the controls to do what I want?

Use the built in reward (+1 for every time frame where the pole is "up")? Add an extra term to loss that depends on both time and controls?

Devise a better loss function. The current loss function is $\mathcal{L}(\phi) = \sum_{i=1}^{N} \log p_{SS}(z_{1:T_i}|\Theta_{1:T_i})$

Maybe use unscented Kalman Filter??

Multiply the loss term by a factor of $1/t$ or $1/t^2$. Something along those lines to get implicit reliance on time component. Suggests that a system that learns to stay upright longer has less error/smaller gradient than a system that falls over quickly.

Run x trials with the current model, train the model on those trials with the loss multiplier. Repeat with the updated parameters.

Need a separate optimization problem for learning the control? Maybe learn mapping from observations to control directly with nn.
Design a policy function/mapping from observations to controls

1.1 Training

The considerations for training are:

- What/when I'm training (observation model, control,
- what the goal of that training is(learn to control the system, or just explore, i.e. learn to predict the next state)
- How to get training samples(online/offline)
- LSTM fixed/variable length
- Inputs to LSTM(physical specifications, previous outputs, etc.)

1.2 Thoughts

- Train transition model with LQR, hope that the model learns how to linearize given any particular observation. Then separately train the learned transition/control models to put the pole into any place of our choosing. Loss function: euclidean distance from our desired state? Only works if we have knowledge of what the states mean

- Suppose that I include the previous state as input to the LSTM, the system becomes auto-regressive, do the assumptions leading to the kalman filter, and the log-likelihood function still hold?

2 Assumptions

We have the linear dynamical system:

$$\begin{array}{rcll} l_{t+1} & = & A_t l_t + B_t u_t + g_t \epsilon_t & \epsilon_t \sim \mathcal{N}(0,1) \\ Z_t & = & C_t l_t + D_t u_t + \sigma_t \varepsilon_t & \varepsilon_t \sim \mathcal{N}(0,1) \end{array}$$

where $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{m \times r}, C \in \mathbb{R}^{z \times m}, D \in \mathbb{R}^{z \times r}, l \in \mathbb{R}^m, u \in \mathbb{R}^r, Z \in \mathbb{R}^z, \epsilon \in \mathbb{R}^m, \varepsilon \in \mathbb{R}^z$ However, it simplifies if, as in our situation, C = I, and D = 0.

3 Potential mistakes

In computing the likelihood: $\mathcal{N}(z_t | \mu_t, \Sigma_t)$ Compare the paper's calculation:

$$\begin{array}{rcl} \mu_1 & = & a_1^T \mu_0 \\ \Sigma_1 & = & a_1^T \Sigma_0 a_1 + \sigma_1^2 \\ \mu_t & = & a_t F_t f_{t-1} \\ \Sigma_t & = & a_t^T (F_t S_t F_t^T + g_t g_t^T) a_t + \sigma_t^2 \end{array}$$

with my control version:

```
mu_1 = tf.matmul(trans(self.C)[0], self.mu_0)
mu = tf.matmul(C, tf.add(tf.matmul(A,l_filtered), tf.matmul(B,u)))

Sigma_1 = tf.matmul(tf.matmul(trans(self.C)[0],
    tf.linalg.diag(tf.squeeze(self.Sigma_0))),trans(self.C)[0],
    transpose_b=True)+tf.square(trans(self.sigma)[0])

temp = tf.matmul(tf.matmul(A, P_filtered), A, transpose_b=True) +
    tf.matmul(g, g, transpose_b=True)
Sigma = tf.matmul(tf.matmul(C, temp), C, transpose_b=True) +
    tf.square(sigma)
```