Journal of Machine Learning Research 23 (2025) 1-8 Submitted 05/07; Revised 05/07; Published 05/07

FeUdal Networks for Hierarchical Reinforcement Learning

Luke Reynolds L.REYNOLDS@YALE.EDU
Department of Statistics
Yale University
New Havent, CT 06520, USA

Editor: S&DS 685 Staff

Abstract

Throughout this course we have focused monolithic algorithms for RL. That is- a single, end-to-end policy that directly maps states to actions (DQN, PPO, etc). To try something new, I would like to recreate the architecture/training method used at the original paper FeUdal Networks for Hierarchical Reinforcement Learning from scratch. This alternative technique creates two models: a "manager" model that sets vague goals at some large interval, and a smaller "worker" model that makes actions every time step to achieve the goal that the manager model set. It was found in the original paper that such models perform better especially when the task requires large timescale credit assignment and memorization tasks. I aim to mainly recreate these results on MiniGrid tasks.

Keywords: Reinforcement Learning, Hierarchical Reinforcement Learning, FeUdal Networks, MiniGrid, Sparse Rewards

1 Architecture Details

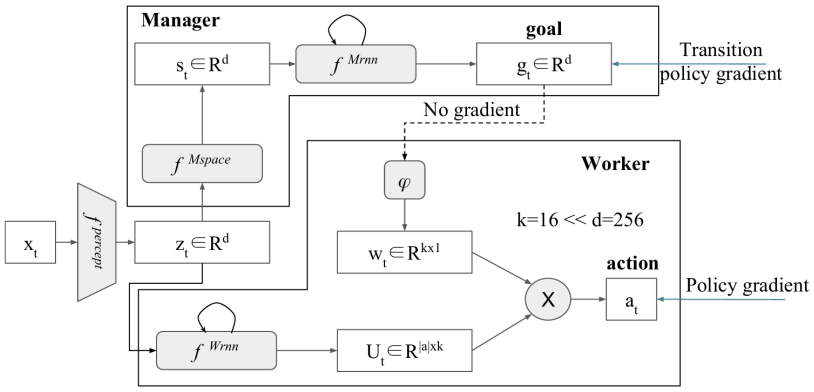

Figure 1: Schematic illustration of FuN from the original paper

The general structure of FeUdal Networks (FuN) has two main parts, as shown above. The first part is the manager, which operates at a more zoomed out time scale and produces sub-goals (encoded as a goal vector g_t) for the worker to carry out. The manager receives this sub-goal and tries produce an action that brings the agent closer to achieving

this sub-goal. Both the manager and the worker share a perceptual model which converts the raw pixel values of the environment to a more useful embedding.

1.1 Perception Module

The shared perception module is quite simple. In my experiments I mainly copied the original paper by placing a two layer CNN with ReLU activations followed by a two layer fully connected network, eventually projecting to a d-dimensional embedding space. As I will discuss later, I also experimented with simplifying the observations x_t to be the raw $7x7x3$ observation rather than a fully expanded screen along with a much simpler perception module to see if this would improve convergence on more difficult tasks.

1.2 Manager

The manager receives the perception model's output $z_t \in \mathbb{R}^d$ and maps it to an internal representation of the state $s_t \in \mathbb{R}^d$. The reason that it does not simply use z_t directly is because this state s_t will later be used to asses if the worker was successful in achieving the goal. Thus s_t should only information of the environment relevant to sub-goals.

After projecting to s_t, the manager applies an RNN module f^{Mrnn} (M standing for manager). In order to ensure that the manager operates at a zoomed-out time scale, the RNN that is chosen here is a dilated LSTM (dLSTM), whose architecture I will cover later. The output of the dLSTM is the goal vector $g_t \in \mathbb{R}^d$, which is then fed into the worker module.

Notice that g_t and s_t are both in $\mathbb{R}^d$. This is of key importance, because the goal vector g_t is interpreted as a desired *direction in the state space*. That is, if at time t_0 the state vector is s_{t_0} and the goal vector produced is g_{t_0}, then the goal g_{t_0} is said to be achieved at time t_1 if the cosine similarity between $s_{t_1} - s_{t_0}$ and g_{t_0} is large. Later when we discuss training, this cosine similarity will be used in the loss function of the worker to encourage the worker to actually achieve the goals set by the manager.

The original paper chose to use directions in the state space rather than positions themselves because it is more reasonable to assume that arbitrary *directions* in the state space are achievable rather than arbitrary *positions.*

1.2.1 The Dilated LSTM

The dLSTM architecture is a new architecture proposed in the paper based on dilated convolutional networks. For a dilation radius r, they define the full state of the network to be $h = \{\hat{h}^i\}_{i=1}^r$. That is, it is composed of r separate groups of sub-states.

At time t, the network is defined by $\hat{h}_t^{t\%r}, g_t = LSTM(s_t, \hat{h}_{t-1}^{t\%r})$ where % denotes the modulo operation. Thus at each time t, only one of the groups is used in the calcula-

tion/updated. In the computation of the goal, we take the sum of the last r outputs of the goal vector to smooth out changes in the goal vector.

By only updating small parts of the hidden state at once, we allow the dLSTM to preserve memories for long periods and also learn from every input experience. The original paper found that using dLSTM (rather than traditional LSTM) was essential to get good convergence on certain tasks.

1.3 Worker

The worker module does not need its own internal representation and directly uses the $z_t \in \mathbb{R}^d$ from the perception module. The worker thus immediately applies an RNN module f^{Wrnn} (W standing for worker), which in this case is able to be a simple LSTM.

The worker also receives the goal vector g_t from the manager and projects with a simple linear map φ to a much lower dimensional space $\mathbb{R}^k$ with $k << n$. In order to smooth out the variation in the goal (especially since the dLSTM in the manager only outputs one of its hidden states groups at a time), the worker actually sums over the last c values of g_t where c is the dilation of f^{Mrnn}. That is, $w_t = \varphi\left(\sum_{i=t-c}^{t} g_i\right)$.

The output of the LSTM is reshaped to a $|\mathcal{A}| \times k$ matrix U_t, which essentially has a k dimensional embedding for each of the actions. Then by computing the matrix multiplication $U_t w_t$, the result is a vector in $\mathbb{R}^{|\mathcal{A}|}$ which measures how closely each action aligns with the goal. By running a softmax over this vector, a policy is derived.

It is crucial that the linear map φ has no bias term, as this forces w_t to not be able to be a nonzero constant with respect to $\{g_i\}$. This is needed to ensure the worker actually follows what the manager instructs.

1.4 Summary Equations

The following key equations summarise the model as a whole:

$$z_t = f^{\text{percept}}(x_t) \tag{1}$$

$$s_t = f^{\text{Mspace}}(z_t) \tag{2}$$

$$h_t^M, \hat{g}_t = f^{\text{Mrnn}}(s_t, h_{t-1}^M); \quad g_t = \hat{g}_t / \|\hat{g}_t\| \tag{3}$$

$$w_t = \varphi\left(\sum_{i=t-c}^{t} g_i\right) \tag{4}$$

$$h_t^W, U_t = f^{\text{Wrnn}}(z_t, h_{t-1}^W) \tag{5}$$

$$\pi_t = \text{SoftMax}(U_t w_t) \tag{6}$$

2 Training

Both the worker and the manager are trained using a variant of advantage actor critic.

We begin with training the worker. Note that the goal of an agent is to maximize the discounted return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ for some $\gamma \in [0, 1]$. Note that both the worker and the manager can have different values of γ.. Often it makes sense to have the γ value for the worker to be smaller than the value for the manager, since the worker should greedily try to achieve the goal direction in the short term.

We want to reward the worker when it make movements that in the direction of the goal, so we define the intrinsic reward as

$$r_k^I = \frac{1}{c} \sum_{t=1}^{c} d_{\cos}(s_t - s_{t-i}, g_{t-i})$$

where $d_{\cos}$ is the cosine similarity. It was found in the original paper that it helps to also let the worker see the environment extrinsic reward r_t so in practice the worker is trained on $R_t + \alpha R_t^I$ where α controls the strength of the internal reward. Then we train the worker using advantage actor critic:

$$\nabla \pi_t = A_t^D \nabla_\theta \log \pi(a_t | x_t; \theta)$$

where the Advantage function $A_t^D = (R_t + \alpha R_t^I - V_t^D(x_t; \theta))$ is calculated using an internal critic that estimates the value functions for the combined reward.

The manager is trained similarly with a gradient 'based update that estimates the "true" gradient

$$\nabla g_t = A_t^M \nabla_\theta \log \pi(a_t | x; \theta)$$

where $A_t^M = R_t - V_t^M(x_t; \theta)$ is the manager's advantage function computed from an internal critic. The real algorithm, however, uses a proxy for this gradient assuming that the direction in state space $s_{t+c} - s_t$ follows a von Mises-Fisher distribution. In this case, we would have $p(s_{t+c} | s_t, o_t) \propto e^{d_{\cos}(s_{t+c} - s_t, g_t)}$. Thus we actually use

$$\nabla g_t = A_t^M \nabla_\theta d_{\cos}(s_{t+c} - s_t, g_t(\theta)).$$

This is called the "transition policy gradient," and is what allows for the fast training of FuN models. The intrinsic reward of the worker actively encourages this von Mises-Fisher distribution on trajectories in state space, so as time goes on this assumption becomes more and more likely to hold true.

3 Experiments

I originally wanted to run this algorithm on MiniGrid-MultiRoom-NxN and compare to something like PPO, but found was in the end not able to solve very complicated tasks. I will write up here what problems I *was* able to solve, and what I tried to resolve the more complicated tasks.

3.1 Solving MiniGrid-Empty-Random-5x5-v0

One of the simplest non-trivial problems in the MiniGrid environment collection is MiniGrid-Empty-Random-5x5-v0. In this problem, the agent is spawned in a random location in a 5x5 grid and must navigate to the green square in the corner of the room. The model can only observe a restricted view of the tiles in front of it, which depends on the models position and orientation. A reward of $1 - 0.9 \cdot (\text{step\_count}/\text{max\_steps})$ is given for success, and 0 for failure. If the model is unable to find the square in under 100 steps, the episode is ended automatically.

I created a FuN agent with a simple perception unit (CNN with 64 output channels channels, ReLU, CNN with 32 channels, a flatten layer, a linear layer projecting up to 128 dimensions and then back down to 64), a manager with dilation given by 10, and a worker with goal embedding dimension $k = 16$. The model was trained with the loss functions described above and Adam with early stopping when the model was able to find the goal in less than 30 time-steps for all of the last 100 runs of the model. The model was able to learn the task in about 1200 epochs as shown in Figure 2. Note that the model performance

Figure 2: Training over time for FuN. I plot the average number of time-steps in the episode grouped into bins of 10 episodes, as this was a more intuitive measure of performance than the reward itself.

sometimes was very good at the very start, but this is mostly due to random change. If the model spawns in right next to the green square, it is trivial to solve the task. However after a large number of epochs, the model is consistently able to find the door after a reasonable number of time-steps ($\approx 5 - 20$). This represents a true ability to find the square. There are still scenarios where the model fails to solve the task, which is shown by large spikes to 100 (the maximum number of time-steps.) I loaded this saved model and ran 5 episodes, plotting each frame of the models trajectory. These were the first 5 trajectories, they are not cherry picked:

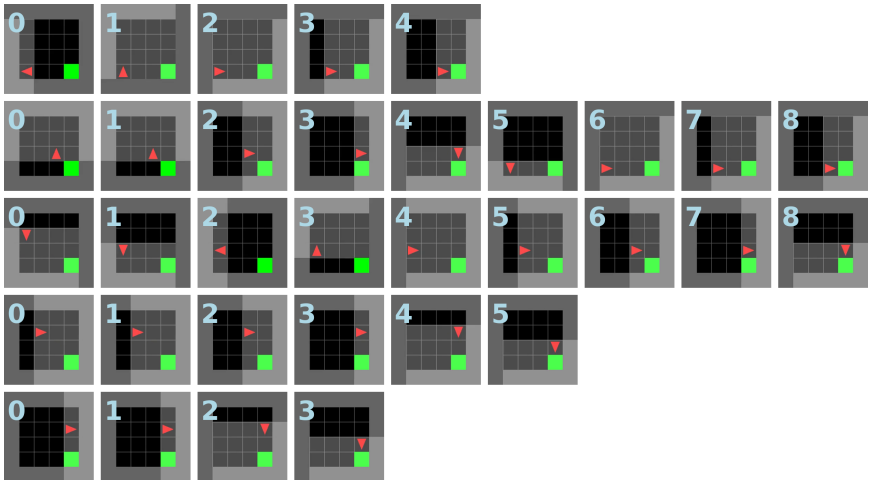

Figure 3: Five random trajectories of the model. Each row is a trajectory. The model is not perfect, but it demonstrates advanced knowledge about the environment!

However, a basic implementation of PPO was unable to solve the task in a similar amount of time, see Figure 4.

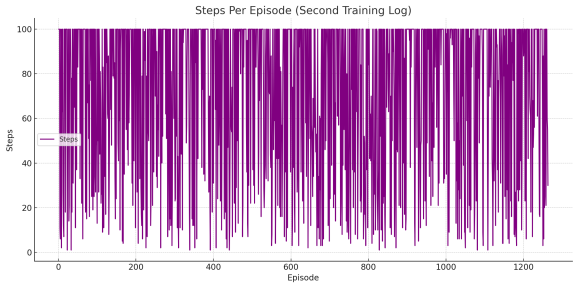

Figure 4: Training over time for PPO. Again, I plot the average number of time-steps in the epsiode over time. The model fails to learn with PPO despite a similar amount of training time.

I am sure that with enough hyperparameter tuning/training time, PPO *could* solve this task. However, the sparse rewards and random, unpredictable initialization made make PPO perform very poorly out of the box.

3.2 Attempting to Solve MiniGrid-Unlock-v0

Given the models success in solving MiniGrid-Empty-Random-5x5-v0, I was fairly confident my model setup was correct. I switched the model environment to be MiniGrid-Unlock-v0, which is a much more complicated environment where the model must...

1. Find the key

2. Pick up the key
3. Find the door
4. Open the door

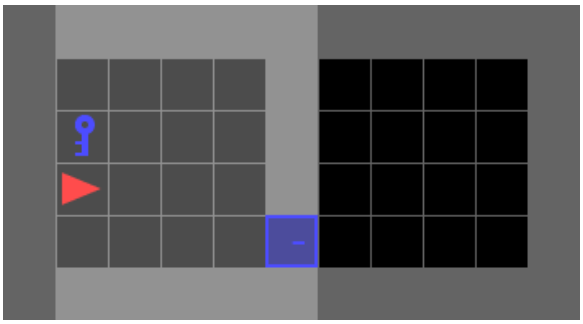

Figure 5: A visualization of a random level from the MiniGrid Unlock task. The player position, key position, door position, and color of key/door are randomized at each run.

only when all four actions were completed correctly does the model receive a reward. A reward of $1 - 0.9 \cdot (\text{step\_count}/\text{max\_steps})$ is given for success, and 0 for failure. The epsiode is automatically ended after 255 steps. Thus, this is a *very* sparsely rewarded task, but it seemed to be a task especially well suited for the FuN model due to the clear existence of sub-goals. I ran the same code as in the previous experiment from scratch overnight, but the model was unable to learn. Although the model was occasionally (approximately once ever 100 epochs) able to unlock the door, the rewards were too sparse for the model to learn.

I tried a variety of things to improve performance, but I was unable to progress meaningfully on this problem. I attribute this mainly due to the fact that I needed to wait over 6 hours for each experiment to run. In the original experiment, the model was only able to start to progress on Montezuma revenge (a similarly sparse task) after about $50,000,000$ observed actions in the environment. Here is a summary of the things I tried:

1. Implementing multiple workers in the environment setup so that the model was able to run 16 versions of the game in parallel.
2. Implementing a replay buffer for the last several episodes to train the value function with less noise
3. Changing the architecture of the perception module to instead accept a 7x7x3 image which included all the information in a simpler format. I also tried one-hot encoding the different colors to get an even more accessible observation.
4. Changing the dilation to be smaller (≈ 5) to give the manager more opportunities to direct to the worker

5. Implementing more advanced reward shaping by giving the model a reward of +0.5 when the model picked up the key. This in particular seemed to marginally help, and after implementing this the model was able to find the key consistently.

None of these attempts allowed the model to fully solve the environment.

As a final test to see if there was a glaring issue in my code, I used a popular repo with an implementation of the FuN paper from here[1]. I implemented a wrapper for MiniGrid that allowed their version of FeUdal Networks to run, and found that even this version (which is almost certainty correctly implemented) failed to learn after leaving it running overnight. Thus, I believe this issue to related to the problem itself (with its extremely sparse rewards) as well as perhaps failures of the FuN architecture itself. While there may be some selection of hyperparameter that allow the model to learn this task, I was not able to solve them.

4 Conclusion + Disclosures

Although I did not achieve everything I initially planned to with this project, I learned a lot about hierarchical reinforcement learning. The model was able to learn the task of MiniGrid-Empty-Random-5x5-v0 fairly quickly, and visualizations of the trajectory confirm that the model in a sense "understands" how to locate and walk towards the goal. My attempts to solve the MiniGrid-Unlock-v0 were ultimately unsuccessful, but I still learned a lot about different ways to improve stability of reinforcement learning.

I originally wrote all the code for the FuN training + architecture myself (without using ChatGPT nor the github repo I mentioned earlier). However, the model failed to learn, and when I asked ChatGPT it pointed out several errors (such as forgetting to attach/detach gradients in key places) that destroyed my implimentation of the FuN architecture. Thus ChatGPT was key in my debugging and completion of the project. I also asked ChatGPT to reorgnize my code about halfway through because I was working within the a notebook and wanted a clean final version. ChatGPT also generated Figure 2 and Figure 3 given the data I had collected. Lastly, I used ChatGPT to implement PPO for gridworld as this was not the focus of my project.

1. Note: I did not know this repo existed until yesterday, all the code up until this point was generated by me (as well as ChatGPT, see disclosures section).