Program Guided Agent:

`https://openreview.net/pdf?id=BkxUvnEYDH`

Paper Summary Notes

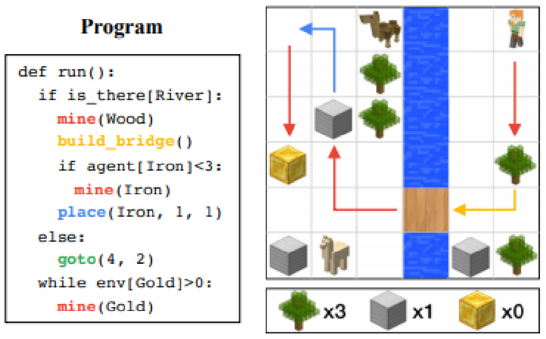
1 Main Ideas, High Level Assumptions

Question: can we *guide* a reinforcement learning agent so that we have implicit control of the policy we would like the agent to follow?

- many complex mdp's and stochastic reinforcement learning algorithms can be naturally decomposed into hierarchy with subtasks that need to be completed depending on some high level ideas
- (e.g. Getting to work can be accomplished by finding house door, exiting door, finding car door, getting in car, driving to train station, finding train door, getting in train door, ...); we can see finding a specific door is a repetitive subroutine which may be learned independently
- what if we have some ideas about the high level logic that should be followed, and we want to guide the agent through our logic?
- believe that natural language instructions are too ambiguous, so instead we should encode our instructions into a very high level simple program
- **assumption**: mdp can be decomposed into subtasks, which can be learned conditioned on goals we specify via the provided instructional program
- **approach**: consider a family of simple but precise computer programs, which allow for a rich prior over policies, paired with a multi-task learning framework that acts accordingly to goals extracted from the code's control flow
- **novelty**: when learning from expert behaviour is too expensive, and learning from natural language is too poor, we can achieve zero shot generalization to complex instructional; programs, and efficient end-to-end learning compared to other s.o.a

2 Approach and Formulation

2.1 Defining the Family of DSL (domain specific language) Programs



Program

```
def run():
  if is_there[River]:
    mine(Wood)
    build_bridge()
    if agent[Iron]<3:
      mine(Iron)
    place(Iron, 1, 1)
  else:
    goto(4, 2)
  while env[Gold]>0:
    mine(Gold)
```

Figure 1: An illustration of the proposed problem. We are interested in learning to fulfill tasks specified by written programs. A program consists of control flows (*e.g.* `if`, `while`), branching conditions (*e.g.* `is_there[River]`), and subtasks (*e.g.* `mine(Wood)`).

Program p := def run() : s

Statement s := while(c) : (s) | b | loop(i) : (s)
| if(c) : (s) | elseif(c) : (s) | else : (s)

Item t := Gold | Wood | Iron

Terrain u := Bridge | River | Merchant | Wall | Flat

Operators o := $> \; \geq \; == \; < \; \leq$

Numbers i := A positive integer or zero

Perception h := agent[t] | env[t] | is_there[t] | is_there[u]

Behavior b := mine(t) | goto(i, i)
| place(t, i, i) | build_bridge() | sell(t)

Conditions c := $h[t] \; o \; i$ | $h[u] \; o \; i$

Figure 2: The domain-specific language (DSL) for constructing programs. Each program is composed of domain dependent perception, subtasks, and control flows.

2.1.1 Perception Primitive

- circumstances in the environment that can be perceived by the agent
- (e.g. is-there(River))

2.1.2 Action Primitive

- defines a subtask for a certain behaviour (goal)
- (e.g. go-to(1-1))

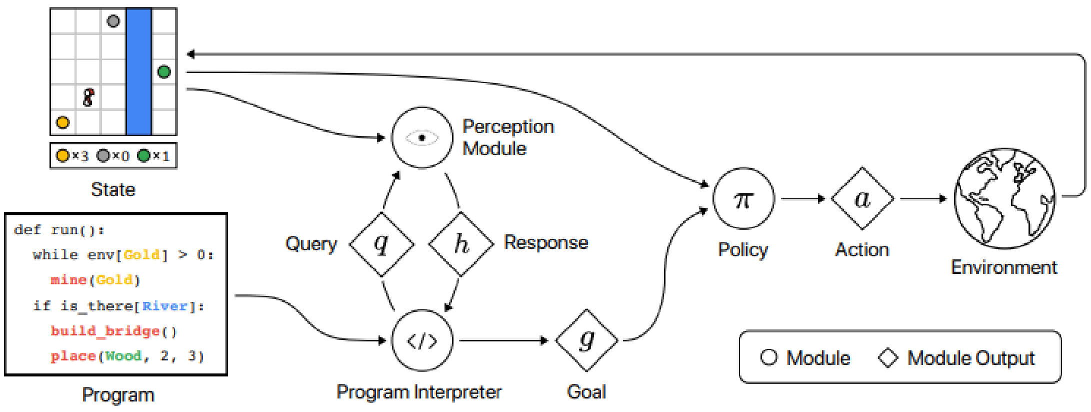
2.1.3 Control flow

- if/else statements, loops, boolean values

2.2 Program Definition

- A program p is a deterministic function that outputs a desired behavior (i.e.subtask) given a history of states $o_t = p(H_j)$, where $H_j = \{s_1, ..., s_t\}$ is a state history with $s \in S$ denoting a state of the environment, and $o \in O$ denotes an instructed behavior (subtask)

3 Architecture



```
def run():
  while env[Gold] > 0:
    mine(Gold)
  if is_there[River]:
    build_bridge()
    place(Wood, 2, 3)
```

Figure 3: Program Guided Agent. The proposed modular framework comprehends and fulfills a desired task specified by a program. The program interpreter executes the program by altering between querying the perception module with a query q when an environment condition is encountered (*e.g.* `env[Gold]>0`, `is_there[River]`) and instructing a policy when it needs to fulfill a goal/subtask g (*e.g.* `mine(Gold)`, `build_bridge()`). The perception module produces a response h to answer the query, determining which paths in the program should be chosen. The policy takes a sequence of low-level actions a (*e.g.* `moveUp`, `moveLeft`, `Pickup`) interacting with the environment to accomplish the given subtask (*e.g.* `mine(Gold)`).

3.1 Program Interpreter

A compiler that reads a DSL program and executes it by querying a perception module with environment conditions (e.g. env[Gold]¿0) and instructing the policy with subtasks (e.g. mine(Gold)).

- *not* learned
- rule based compiler creates parser tree given DSL langauge outline

3.2 Perception Module

responds to perception queries (e.g. env[Gold] ≥ 0) by examining the observation and predicting responses (e.g. true)

- learned
- Determining which paths should be chosen when executing a program requires grounding a symbolically represented query (e.g. is-there[River]) and perceiving the environment.
- perception module Θ that learns to map a query q and current observation s to a boolean response:

  $$h = \Theta(q, s)$$

  where h denotes the corresponding perception output (e.g. true/false)

3.3 Action Module

performs low-level actions (e.g. moveUp, moveLeft, pickUp) to fulfill the symbolically represented subtasks (e.g. mine(Gold)) provided by the program interpreter

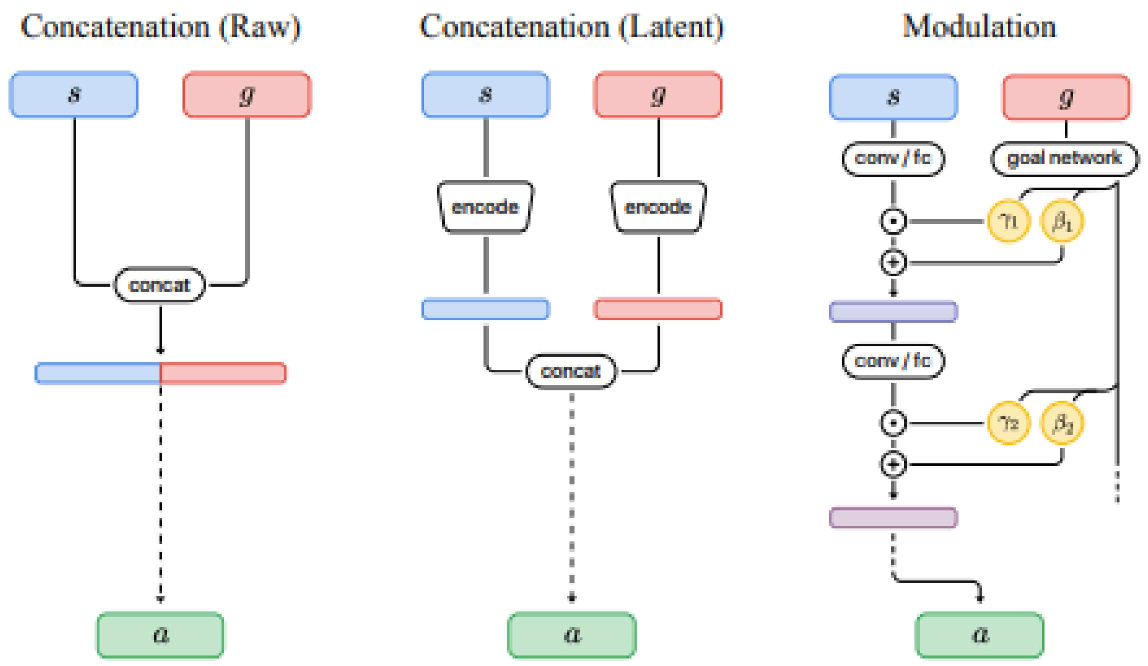
- learned
- When program execution reaches a subtask/leaf node (e.g. mine(Gold)), the agent is required to take a sequence of low-level actions (e.g. moveUp, moveLeft, Pickup) to interact with the environment to fulfill it
- multitask policy π which is instructed by a symbolic goal (e.g. mine(Gold)) provided by the program interpreter indicating the details of the corresponding subtask.
- train the policy using actor-critic reinforcement learning, which takes a goal vector g and an environment state s and outputs a probabilistic distribution a for low-level actions

  $$a \sim \pi(s_t, g_t|\theta)$$

- value estimator used for our policy optimization is also goal-conditioned:

  $$V_\pi s_t, g_t) = \mathbb{E}[\sum_t \gamma^t R_t | s_t, g_t]$$

3.4 State Modulation Mechanism for Multi-task learning



added mechanism for feedings goal-conditioned state observation into policy network in a way that accounts for different task-specific observation significance

- common way to feed a state and goal to a policy parameters by a neural network is to concatenate them in a raw space or a latent space:

$$\{s, g\} \longrightarrow \phi([s, g]) \; \textit{(raw concat)}$$

$$\{s, g\} \longrightarrow \phi([\psi(s), \psi(g)]) \; \textit{(latent concat)}$$

- less effective when the policy has to learn a diverse set of tasks.
- we employ a goal network $\mathcal{G} : g \to \{\Gamma, \beta\}$ to encode the goal and compute affine transform parameters Γ and β, which are used to modulate state features to $e'_s = \Gamma \cdot e_s + \beta$.
- With the modulation mechanism, the goal network learns to activate state features related to the current goal and deactivate others.
- the modulated features are sent to action module and treated as typical observation in R.L. alg

4 Learning

4.1 Perception Module

- supervised learning task given tuples of (query q, state s, ground truth perception h_{gt}), we train a neural network Φ to predict the perception output h by optimizing the binary cross-entropy loss:

$$LCE = -h_{gt} \cdot log(h) - (1 - h_{gt}) \cdot log(1 - h)$$

.

  - A query such as is-there[River] is represented as a sequence of symbols extracted via the compiler

4.2 Action Module

- (A2C) with discrete action spaces:

$$A_t \nabla_\theta log \pi_\theta a_t | s_t, g_t \; A_t = R_t - V(s_t, g_t)$$

giving us the entropy-regularized update

$$\theta \longleftarrow \theta + \alpha(A_t \nabla_\theta log \pi_\theta (a_t | s_t, g_t) + \beta \nabla_\theta H_{\pi_\theta}$$

5 Experiments

Research Questions:

1. Can our proposed framework learn to perform tasks specified by programs?
2. Can our modular framework generalize better to more complex tasks compared to end-to-end learning models?
3. How well can a variety of end-to-end learning models (e.g. LSTM, Tree-RNN, Transformer) learn from programs and natural language instructions?
4. Is the proposed learned modulation more efficient to learn a multitask (multi-goal) policy than simply concatenating a state and goal?

5.1 Task Completion

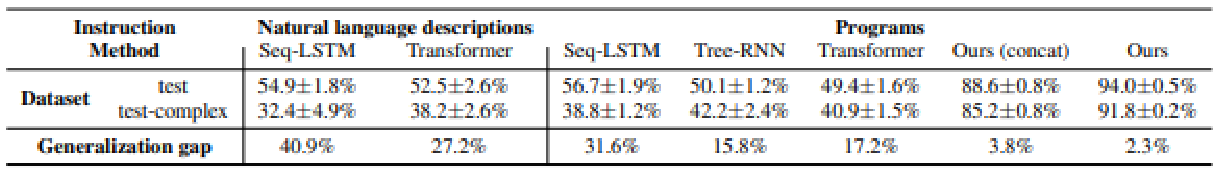


Table 1: Task completion rate. For each method, we iterate over all the programs in a testing set by randomly sampling ten initial environment states and running three models trained using different random seeds for this method. The averaged task completion rates and their standard deviations are reported. Note that all the end-to-end learning models learning from natural language descriptions and programs suffer from a significant performance drop when evaluated on the more complex testing set.

Instruction		Natural language descriptions		Programs				
Method		Seq-LSTM	Transformer	Seq-LSTM	Tree-RNN	Transformer	Ours (concat)	Ours
Dataset	test	54.9±1.8%	52.5±2.6%	56.7±1.9%	50.1±1.2%	49.4±1.6%	88.6±0.8%	94.0±0.5%
	test-complex	32.4±4.9%	38.2±2.6%	38.8±1.2%	42.2±2.4%	40.9±1.5%	85.2±0.8%	91.8±0.2%
Generalization gap		40.9%	27.2%	31.6%	15.8%	17.2%	3.8%	2.3%

- framework achieves a satisfactory test performance and only suffers a negligible drop when it is evaluated on test-complex set due to the modular design, which explicitly utilizes the structure and grammar of programs
- all the end-to-end learning models suffer a significant performance drop between test and test-complex sets

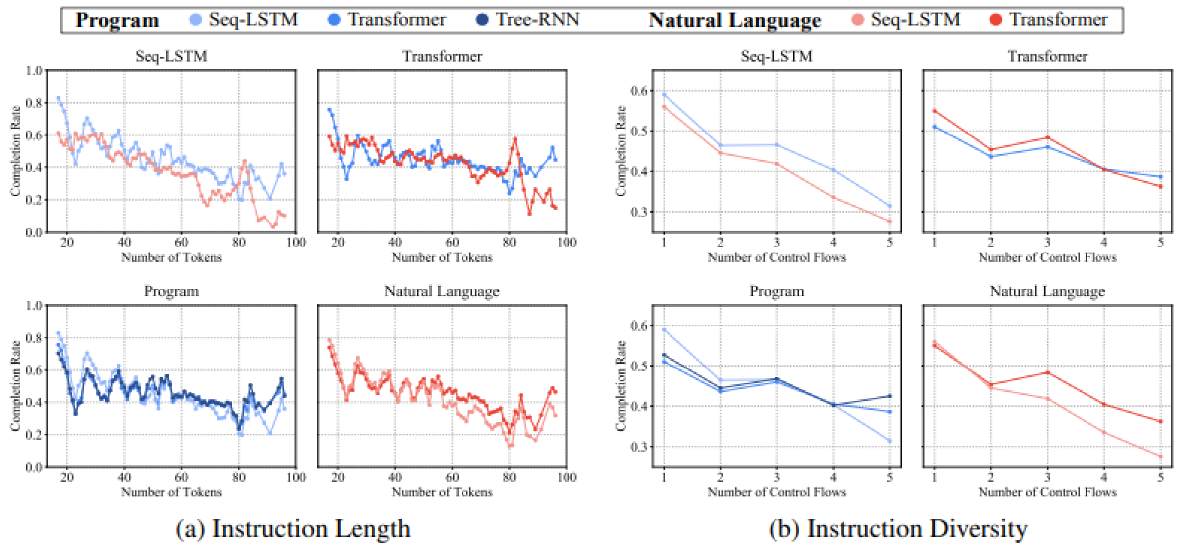
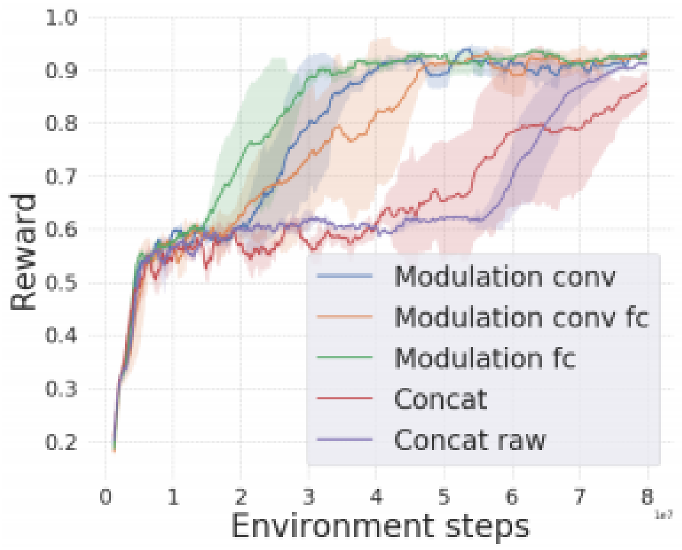


Figure 5: **Analysis on end-to-end learning models:** (a) Models learning from programs generalize better to longer instructions. Transformer is more robust to longer instructions (Upper). Tree-RNN exploiting the program structure generalizes the best, but performs worst for shorter programs (Lower). (b) Seq-LSTM learning from both instructions performs worse as the diversity increases. Transformer learns better from natural language when the instructions are less diverse (Upper). Transformer and Tree-RNN learning from programs are more consistent as the diversity increases, yet perform worse on less diverse instructions (Lower).

5.2 Policy Modulation



- concatenating a state and goal in a raw space (Concat raw) or a latent space (Concat) as baseline
- modulating convolution or modulating fc is more sample efficient

6 Applications and Limitations

6.1 Applications

- put risk algo in DSL format, alongside some general heuristics for an algo
- generalized agents across a multitude of tasks and policy behaviours

6.2 Limitations

- need a way to map natural language to DSL format
- requires efficient and effective pre-training the perception module, which needs to be retrained if the Domain is modified or if we want to expand the allowable perceptions
- May become intractable if the output of perception module is extended beyond true/false
- modulation beyond navigation tasks is not studied