Topics in Machine Learning: Course Project

Neural Logic Reinforcement Learning

Team Pikachu: Aishwarya S. (20171046), Kritika P. (20161039), Pooja S. (20171403)
Link to Github Repository: https://github.com/Kritikalcoder/NLRL

Overview

Introduction

Problem

Policies learnt by Deep Reinforcement Learning algorithms are not interpretable or generalizable to similar environments.

“Learn programs (procedures) from examples using back propagation, given program structure”

Motivation

Most deep reinforcement learning (DRL) algorithms suffer a problem of generalising the learned policy, which makes the policy performance largely affected even by minor modifications of the training environment. DRL algorithms also use deep neural networks making the learned policies hard to interpret.

Neural Logic Reinforcement Learning (NLRL) represents a neural program synthesis method to learn the policies in reinforcement learning by first-order logic, thus addressing the challenges faced by DRL algorithms.

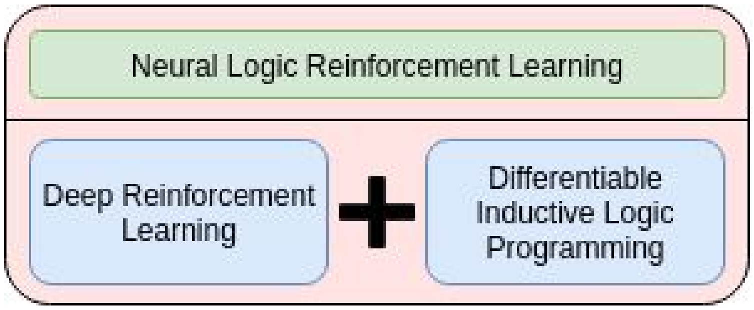
- **Symbolic Program Synthesis**
  Input: Examples of input-output pairs of the desired program
  Output: An explicit human-readable program that provably satisfies a given high-level formal specification by generating (finding) the output program using symbolic search procedures
- **Neural Program Induction**
  Input: Examples of input-output pairs of the desired program
  Output: An implicit general procedure to map inputs to outputs using a latent representation of the program
- **Neural Program Synthesis**
  Input: Examples of input-output pairs of the desired program
  Output: An explicit human-readable program by generating (finding) it using optimization procedures

Desirable Qualities	Symbolic Program Synthesis	Neural Program Induction	Neural Program Synthesis
Data Efficiency	Yes	No	Yes
Interpretability	Yes	No	Yes
Generalizability	Yes	Sometimes	Yes
Robust to mislabelled data	No	Yes	Yes
Robust to ambiguous data	No	Yes	Yes

Solution Approach

A new RL method: Neural Logic Reinforcement Learning

- Compatible with policy gradients
- Combines Deep RL with Differential Inductive Logic Programming
- Learns near optimal policies in training
- Much more interpretable and generalizable
- Neural Program Synthesis


Neural Logic Reinforcement Learning
Deep Reinforcement Learning
Differentiable Inductive Logic Programming

Preliminaries

First Order Logic Programming

Logic programming can be used to express knowledge in a way that does not depend on the implementation, making programs more flexible, compressed and understandable. It enables knowledge to be separated from use, ie the machine architecture can be changed without changing programs or their underlying code.

First Order / Predicate Logic = Propositional Logic + Quantifiers
The language constructs of **Datalog** is used for this project

No	Element	Symbol	Definition
1.	Variable	V	Variable V coming from a fixed countably infinite set of symbols, representing objects in the domain
2.	Constant	C	Constant C - Represent objects in the domain
3.	Term	T	Term T := V | C
4.	Predicate	P	Predicate P := { DC_1 DC_2 . . . DC_n } | { $ɣ_1$ $ɣ_2$ . . . $ɣ_n$ } Maps one or more objects onto truth values. They could be of knowledge or action type.

	Intensional P		P defined by a set of definitive clauses
	Extensional P		P wholly defined by a set of ground atoms
5.	Atom	α	Atom α := P (T_1 T_2 . . . T_n), where P is an n-ary predicate,
	Ground Atom	γ	An atom is said to be a ground atom when every term is a constant.
6.	Atoms Set	**A**	Set of all atoms: **A**
	Ground Atoms Set	**G**	Set of all ground atoms: **G**
7.	Definite Clause	DC	Rule of the form DC := $\alpha \leftarrow \{ \alpha_1 \alpha_2 \ldots \alpha_n \}$, where $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n \Rightarrow \alpha$ (conjunction) Head Atom := α, Body: = $\alpha_1 \alpha_2 \ldots \alpha_n$, n > 0 Exactly one positive literal, any number of negative literals
8.	Language	L	Language L is defined as a set of predicates L := (P_e , P_i , arity, C) P_e : Set of extensional predicates P_a : Set of auxiliary intensional predicates P_i : Set of intensional predicates, $P_i = P_a \cup$ {target} Arity_e gives the arity of each predicate C: set of constants
9.	Substitution	$\alpha[\theta]$	Applying substitution θ (variables) to atom α
10.	Function	f	A relation among objects, of arity greater than 0, that maps one object onto another.
11.	Language Frame	$\mathscr{L}$	$\mathscr{L}$:= (Target, P_e , arity_e , C) Target: Target intensional predicate P_e : Set of extensional predicates Arity_e gives the arity of each predicate C: set of constants
12.	Rule Template	τ	τ := (v, int) pair v = number of existentially quantified variables allowed in the clause int $\in$ {0, 1} = intensional predicate flag Describes a range of clauses that can be generated

13.	Program Template	Π	Π : = (P_a , $arity_a$, rules, T) P_a : Set of auxiliary intensional invented predicates $Arity_a$: gives arity of each auxiliary predicate Rules: map from each intensional predicate p to a pair of rule templates T : max number of steps of forward chaining inference
14.	Ground Rule		A clause in which all variables are substituted by constants
15.	Immediate Consequence	$cn_R(X)$	$cn_R(X) = X \cup \{\gamma \mid \gamma \leftarrow \gamma_1 \ldots \gamma_m \in ground(R), \wedge \gamma_i \in X\}$ $cn_R(X) = X \cup \{\alpha[\theta] \mid \alpha \leftarrow \alpha_1 \ldots \alpha_m \in R, \wedge \alpha_i[\theta] \in X\}$ Immediately apply rules in R on X R: Set of clauses ground(R): Ground rules of R X: Set of ground atoms

Truth Value

A clause containing variables is true in an interpretation if it is true for all variable assignments (universally quantified in the scope of the clause)

Forward Chaining (Modus Ponens)

A set **R** of clauses is said to entail a ground atom γ if every model satisfying **R** also satisfies γ.
Represented as: $\mathbf{R} \models \gamma$
$\gamma \in cn_\infty(R)$
This is known as forward chaining. It is a bottom-up approach of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.

ILP: Inductive Logic Programming

Inductive logic programming (ILP) is a logical approach to machine learning. From a database of facts and expected results, which are divided into positive and negative examples, an ILP system tries to derive a logic program that proves all the positive and none of the negative examples. More formally, ILP systems develop predicate descriptions from examples and background knowledge. The examples, background knowledge and final descriptions are all described as logic programs.

Given

Target predicate P_T

ILP Problem

Inductive Logic Programming (ILP) problem is defined as
Tuple ***< $\mathcal{L}$, B, P, N >*** of ground atoms, where:

- $\mathcal{L}$ is a language frame
- ***B***: Set of background assumptions (ground atoms formed from the predicates in P_e and the constants in C)
- ***P***: Set of positive instances (ground atoms formed from the target predicate and the constants in C) with truth value = True
- ***N***: Set of negative instances taken outside the extension of the target predicate (ground atoms formed from the target predicate and the constants in C) with truth value = False

Aim
To construct a logic program that explains the set of positive instances ***P*** and rejects the set of negative instances ***N*** (return a set of intensional predicates).

Solution
Find the set **R** of definite clauses such that:

- **B, R** $\models \gamma$, for every $\gamma \in$ **P**
- **B, R** $\not\models \gamma$, for every $\gamma \in$ **N**

Inference
The central idea behind the differentiable implementation of inference is that each clause c induces a differentiable function on ground action atom valuations that implements a single step of forward chaining inference using c.

Inductive / Learning Bias
The inductive bias of a learning algorithm is the set of assumptions that the learner uses to predict outputs given inputs that it has not encountered.
Induction: Finding a set of rules R such that when they are applied deductively to B, they produce the desirable consequences.

DILP: Differentiable Inductive Logic Programming

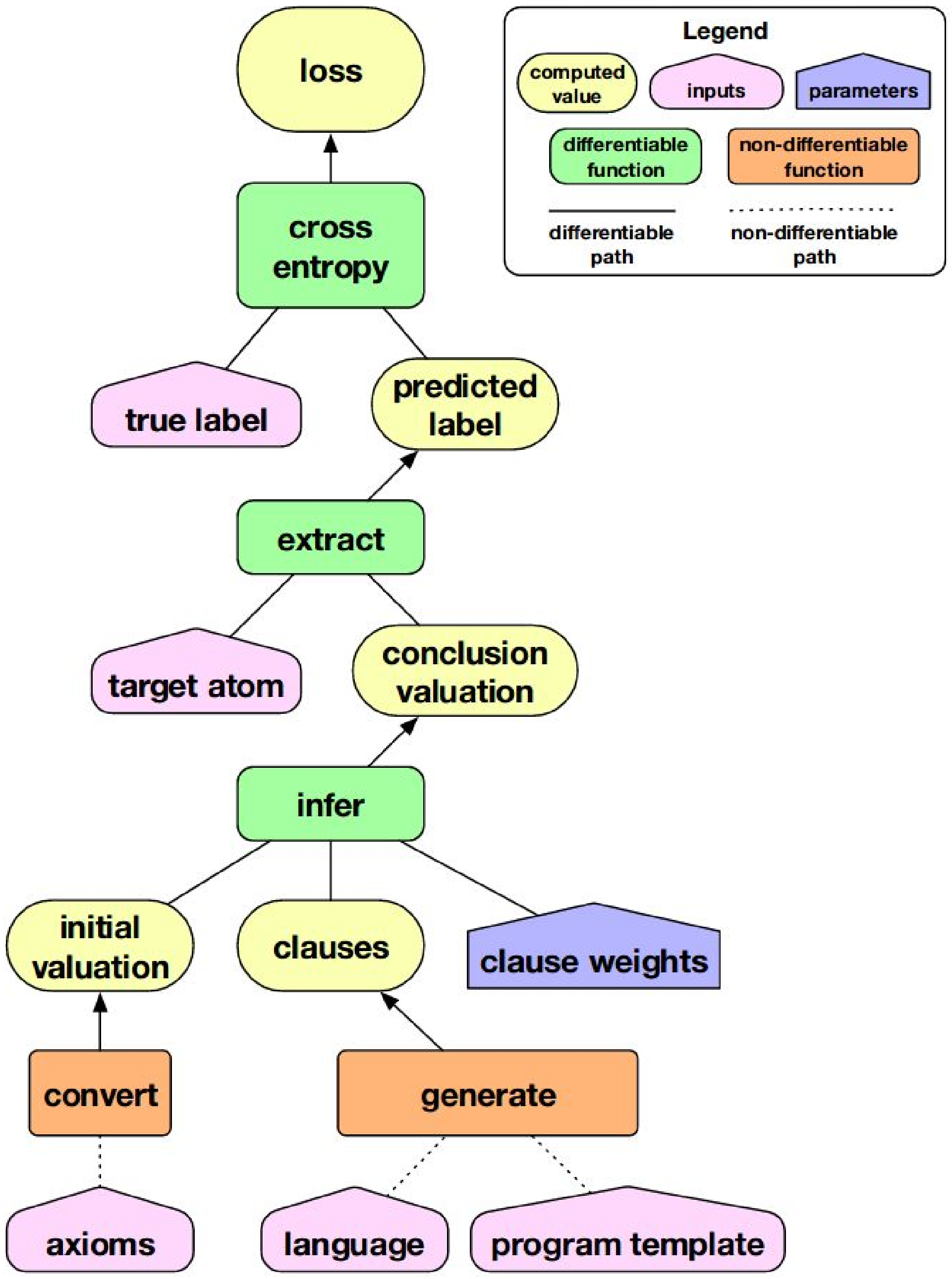
Re-implementation of ILP in an end-to-end differentiable architecture
→ Data-efficient induction system
→ Learns explicit human-readable symbolic rules
→ Robust to noisy and ambiguous data
→ Does not deteriorate when applied to unseen test data

- differentiable implementation of deduction through forward chaining on definite clauses
- reinterpret the ILP task as a binary classification problem, and we minimise cross-entropy loss with regard to ground-truth boolean labels during training
- it requires significant memory resources

→ Robust connectionist approach, differentiable turing machine, input can be raw

∂ILP: DILP Architecture

∂ILP is an attempt to combine ILP with differentiable programming. It uses forward chaining inference. For each predicate, ∂ILP generates a series of potential clauses combinations in advance based on rules templates. The weights give us the probability distribution over clauses. It minimizes log loss using stochastic gradient descent and finally extracts readable programs from the weights.



DRLM: Differentiable Recurrent Logic Machine (Improved ∂ILP)

Set of ground action atoms, **G** = { α_1 , α_2 , … , α_n}
Number of ground action atoms, |**G**| = n
Valuation vector (valuation of each atom α_i) **E** = $[0,1]^n$ = [$e_1\, e_2 \ldots e_n$]
Value e_i : Confidence that ground atom α_i is **True**

Differentiable Recurrent Logic Machine is a mapping f_θ : E → E which produces deductions of the facts e_0, using weights w associated with possible clauses.

$$f_{\boldsymbol{\theta}}^t(\boldsymbol{e}_0) = \begin{cases} g_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}^{t-1}(\boldsymbol{e}_0)), & \text{if } t > 0. \\ \boldsymbol{e}_0, & \text{if } t = 0. \end{cases},$$

Here, t is the deduction step, g_θ implements one step deduction of all possible clauses weighted by their confidence.
The probabilistic sum is denoted as ⊕, and

$$\boldsymbol{a} \oplus \boldsymbol{b} = \boldsymbol{a} + \boldsymbol{b} - \boldsymbol{a} \odot \boldsymbol{b},$$

where $\boldsymbol{a} \in E, \boldsymbol{b} \in E$. $g_{\boldsymbol{\theta}}$ can then be expressed as

$$g_{\boldsymbol{\theta}}(\boldsymbol{e}) = \left(\sum_n^{\oplus} \sum_j w_{n,j} h_{n,j}(\boldsymbol{e}) \right) + \boldsymbol{e}_0,$$

Here, $h_{n,j}$(e) implements one-step deduction of the valuation vector e using jth possible definition of *n*th clause. The weights w_c is the vector of weights associated with predicate c, and the sum of w_c for a single clause is limited to 1 by taking w_c = softmax($\boldsymbol{\theta}_C$).
The predicates are more flexible in DRLM, in comparison to ∂ILP as the weights are directly associated with clauses and not weights of clauses.

MDP with Logic Interpretation

Formulating an MDP with logic interpretation and solving MDPs with a combination of Policy Gradient and DILP.
An MDP with logic interpretation is a triplet (M, p_S, p_A):

- M = (S, A, T, R) is a finite horizon MDP
  S is the finite set of states, A is the finite set of actions, T is the transition function and R is the reward.
- p_S : S -> 2^G is the state encoder
  maps each state to a set of extensional atoms including both information of the current state and background knowledge

- $p_A : [0,1]^{|D|} \to [0,1]^{|A|}$ is the action decoder
  maps the valuation of a set of atoms D to probability of actions. It should be differentiable as we train the system with policy gradient methods on both discrete and stochastic.
  $$p_A(a|e) \; = \; \{(l(e,a)/\sigma \; , \; \sigma \geq 1\}$$
  , where, $p_A(a|e)$ is the probability of choosing action a given the valuations e
- Action determination: adding a set of action predicates to the architecture, which in turn uses depends on the evaluation of these action atoms
- p_S, p_A : handcrafted or are modelled as neural network architectures, that can be trained together using DILP architecture.
- The advantage of using neural network architecture to represent p_A is that, the decision making process is much more flexible. If the optimal policy cannot be expressed as first-order logic then, even when the output is deterministic, the final choice of action may depend on more than just the action atoms when NNA is used.

DILP Example: Even Numbers

Task
To learn which natural numbers are even

Given
Minimal background description of natural numbers

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3), ...\}$$

zero(X) = True when X = 0 (unary predicate)
succ(X,X+1) = True for all X (successor relation)

Positive and negative examples of even predicate are:

$$\begin{aligned} \mathcal{P} &= \{\text{even}(0), \text{even}(2), \text{even}(4), \text{even}(6), ...\} \\ \mathcal{N} &= \{\text{even}(1), \text{even}(3), \text{even}(5), \text{even}(7), ...\} \end{aligned}$$

Solution
Set of Clauses R :=

$$\begin{aligned} even(X) &\leftarrow zero(X) \\ even(X) &\leftarrow even(Y), succ2(Y,X) \\ succ2(X,Y) &\leftarrow succ(X,Z), succ(Z,Y) \end{aligned}$$

Here, the target predicate is even(), and learnt auxiliary predicate is succ2().
Through this, we are able to exhibit recursion and predicate invention (succ2 is an auxiliary learnt predicate).

To exhibit the performance on ambiguous data, a cited work even learns to identify even numbers from raw pixel images

---

NLRL: Neural Logic Reinforcement Learning

Intuition

The agent must learn auxiliary invented predicates by themselves, together with the action predicates. Weights are not assigned directly to the whole policy. The parameters to be trained are involved in the deduction process. Trains the parameterised rule-based policy using policy gradient. Each action is represented as an atom. Scalable approach: Number of parameters is significantly smaller than that in an enumeration of all policies.

Step 1: Make exhaustive list of ground atoms built on all predicates.
Valuation vector is of this length

Step 2: Building all clause combinations for each intensional predicate
Assumption: Every clause will have 2 atoms in its body and one at its head
Every clause with a ground atom at its head will have 2 ground atoms in its body.
Target atoms/predicates are not part of the body
By definition, head atom is not part of the body of a clause

Step 3: Making a deduction matrix
For each clause we obtain list of satisfying pairs of ground atoms for body, given a ground head atom.

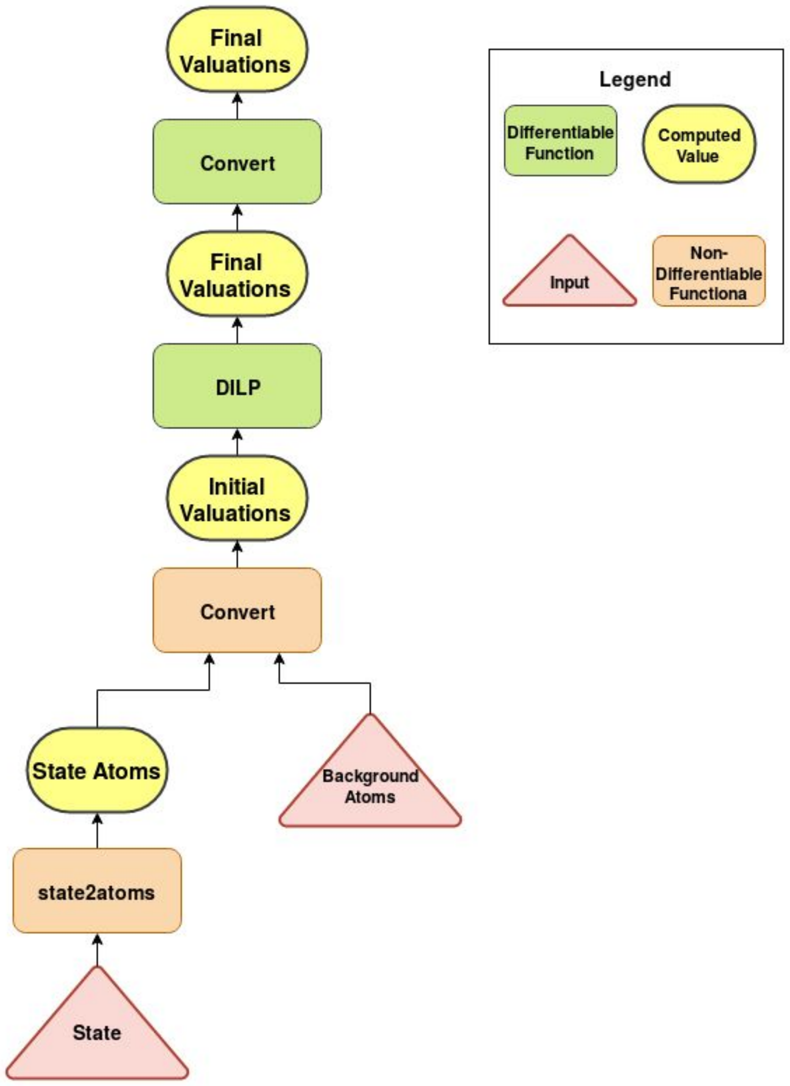
Step 4: For each predicate
Each predicate induces a deduction function f_θ (DRLM)
automatically generate, from each predicate, a differentiable function f_c on valuations that implements a single step of forward chaining inference using c.

Step 5: DILP agent
We initialize the agent with 0-1 valuation for base predicates and random weights to all clauses for an intensional predicate.

Step 6: Forward chaining
For each step in forward chaining, we first get value of all the clauses for all combinations of constants using deduction matrix. Then, each intensional atom's value is updated according to deduction function f_θ which corresponds to the atoms predicate.

Step 7: Choose action
Get value of action atoms from the ground atoms. According to the values choose an action.

Step 8: Update θ
Calculate loss for each state as a function of the discounted reward received and log probability. Update the weights.



Algorithm

Steps of Algorithm

```
1. forward_n <- number of times we chain
2. C <- constants
3. $P_e$ <- extensional predicates      // predicates whose values are initially known
4. $P_t$ <- target predicates           // action predicates we want to predict values of
5. $P_a$ <- auxiliary predicates        // extra predicates which help us in better
   learning
6. $P_i$ <- $P_t + P_a$                 // set of intensional atoms
7. P <- $P_e + P_i$                     // set of all predicates
8. G <- get_ground_atoms()
9. base_valuation <- array, where base_valuation[atom] = 1 if atom is True independent
   of the state, 0 otherwise $\forall$ atom $\in$ G
10. clauses <- dict{pred, create_clauses(pred)} $\forall$ pred $\in P_i$
11. weights <- dict{pred, dict{clause, w}}, $\forall$ clause $\in$ clauses[pred], $\forall$ atom $\in P_i$,
    sum(w)=1
12. batch <- create_batch(num_episodes)
13. train(itr)
```

```
def get_ground_atoms():
    G = []
    for predicate in P:
        G.extend(create_atoms(predicate))
    return G
```

```
def get_atoms(clause):
    // X, Y $\in$ C
    return list of pair of ground atoms which should be true for clause(X, Y) to be true
```

```
def create_atoms(predicate):
    // atom predicate + constants, number of constants = arity of predicate
    Return list of all unique atoms for the predicate
```

```
def create_clauses(current_atom):
    // clause combination of any 2 atoms which can be used to get the current_atom
    if atom.allows_intensional:
        // atoms in G can also be used to get current atom
        returns list of all unique clauses
    // atoms in $P_e$ can also be used to get current atom
    returns list of all unique clauses for the atom
```

```
def create_batch(episodes):
    episodes = []
    state = env.reset()
    for _ in 0, episodes:
        valuations = []
        rewards = []
        actions = []
        done = False
        while not done:
            valuation <- state2valuation(state)
            final_valuation <- forward_chaining(valuation)
            action_probs <- get_action_prob(list(final_valuation[atom] $\forall$atom $\in P_t$))
            action <- choose action according to action_probs
            reward, next_state, done <- env.step(action)
            valuations.append(valuation)
            rewards.append(discount(reward))
            actions.append(action)
        episodes.append(Episode(valuations, rewards, actions, sum(rewards)))
```

```
def state2valuation(state):
    list_atoms = state2atoms(state)
    for atom in list_atoms:
        valuation[atom] = 1
    // all atom values which are true are set to 1
    return valuation + base_valuation
```

```
def state2atoms(state):
    return list of extensional atoms True for that state
```

```
def get_action_prob(action_valuations):
    return probability of actions by normalizing the sum of action_valuations to 1
```

```
def forward_chaining(valuation):
    for _ in 0, forward_n:
        valuation <- infer(valuation)
    return valuation
```

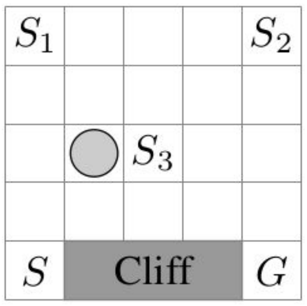
```
def infer(valuation):
    for pred in P_i:
        for clause in clauses[pred]:
            valuation[atom] <- (valuation[atom] + weights[pred][clause] * clause_valuation(atom, clause) ∀ atom ∈ G where atom.predicate == pred)
```

```
def clause_valuation(atom, clause):
    return atom1 * atom2, where atom1 and atom2 make the clause for atom
```

```
def train(itr):
    for _ in 0, itr:
        mini_batch <- sample(batch)
        train_weights(mini_batch)
```

```
def train_weights(minu_batch):
    for valuations, rewards, actions, final_reward in mini_batch:
        action_probs <- get_action_prob(list(final_valuation[atom] ∀ atom ∈ P_t))[actions]
        l <- reward*-sum(log(action_probs))
        optimize_weights(l) // we use RMSPropOptimizer
```

NLRL Example: Cliff Walking



state: $current(1,2)$

background: $zero(0)$, $last(4)$, $succ(0,1)$, $succ(1,2)$, $succ(2,3)$, $succ(3,4)$

List of Types of Predicates defined:

- 4 Auxiliary Predicates

  a) 3 intensional : invented2 (arity = 2), invented3 (arity = 1), invented4 (arity = 2)

  b) 1 intensional: invented1 (arity = 1) (depends only on extensional)

- 4 Action (Target) Predicates
  All of them are intensional. All have arity = 0.
  Predicates = {Up, Down, Left, Right}
- 4 Base Predicates
  All of them are extensional
  a) zero (arity = 1)
  b) succ (arity = 2)
  c) current (arity = 2)
  d) last (arity = 1)

Domain of values = {0,1,2,3,4} as grid is of size 5x5
Known Truth Values of extensional predicates:
a) zero(0) = True, False otherwise
b) succ(x,x+1) = True for x in {0,1,2,3}, False otherwise
c) last(4) = True, False otherwise
d) curr(x,y) = True if agent's location is (x,y), False otherwise

Learnt Policy

$$0.990 : right() \leftarrow current(X,Y), succ(Z,Y)$$
$$0.561 : down() \leftarrow pred(X), last(X)$$
$$0.411 : down() \leftarrow current(X,Y), last(X)$$
$$0.988 : pred(X) \leftarrow zero(Y), current(X,Z)$$
$$0.653 : left() \leftarrow current(X,Y), succ(X,X)$$
$$0.982 : up() \leftarrow current(X,Y), zero(Y)$$

Results

Our Contribution

- Implemented Hybrid architecture: DILP + NN (learnt state encoding, to be able to make sense from raw sensory input data)
- Cartpole environment created with specifications matching requirements of NLRL agent in order to test the performance of the agent in a continuous state space domain
- A reinforcement learning agent to compare NLRL's performance to a reinforcement learning environment when we don't have direct access to the MDP tuple elements.
- Extensive Report and Presentation of the work (besides debugging the original code)

Experiments

Setup

1. Same rules template used for all the subtasks of an environment.
2. Each atom is made up of a maximum of 2 other atoms.

Environments / Domains

Environments vary with different initial states and problem sizes.

→ **Block Manipulation**

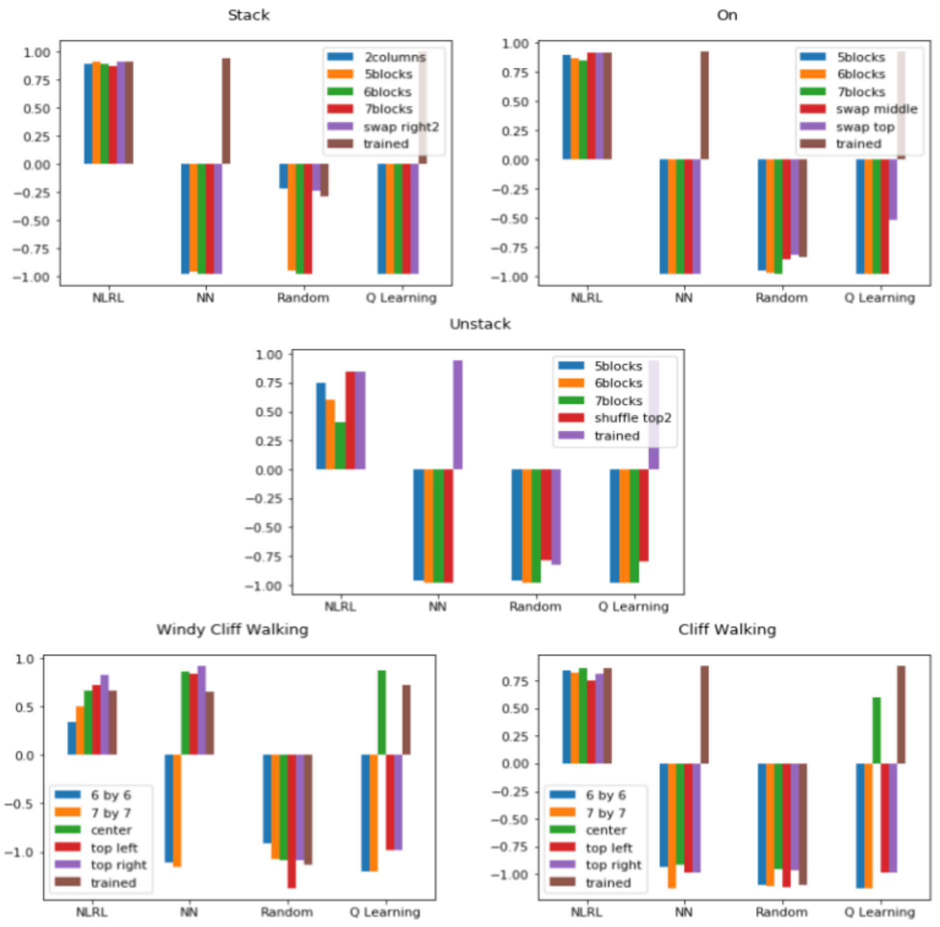
- Performance examined on 3 subtasks: STACK, UNSTACK and ON
- For training, 5 entities: 4 blocks and a floor
- Ground Atoms:
    - on(X,Y): block X is on entity Y
    - top(X): X is on top of the column
- Auxiliary predicates: 4
- Background knowledge:
    - Common to all tasks: isFloor(floor)
    - For ON task: goalOn(X,Y)
- Termination:
    - reaching goal state
    - Taking 50 steps
- States are represented using tuples of tuples.
- Action atoms(25): move(X,Y), move X on top of Y.
- move(X,Y) is valid only if both X and Y are on top of the stack or when Y is floor and X is on top of the stack.
- Variations: swap top 2, swap right 2, swap middle 2, 2 columns, 4 blocks, 5 blocks, 6 blocks, 7 blocks

→ **Cliff-walking**

- Performance examined on 2 subtasks: cliff-walking, windy cliff-walking
- For training, 5 by 5 grid
- Ground Atoms:
    - current(X,Y): X, Y represent coordinates of agent on the grid
- Auxiliary predicates: 4
- Rewards:
    - Cliff position, -1
    - Goal position, +1
    - For each step: -0.02
- Termination:
    - reaching cliff or goal state
    - Taking 50 steps

- State representation: current(X,Y)
- Action atoms:up(), down(), left() and right()
- Variations: top left, top right, center, 5 by 5, 6 by 6, 7 by 7

We present the average peaks of evaluation:



Analysis

NLRL achieves a near-optimal performance in all the training environments and also successfully adapts to all the variations of the environments. In most of the generalization tests, the agents manage to keep the performance in the near optimal level even if they have never experienced these new environments before.

The neural network agents learns near-optimal policy in the training environments. However, the neural network agent appears to only remember the best routes in the training environment rather than learn the general approaches to solving the problems. The overwhelming trend is, in varied environments, the neural networks perform even worse than a random player.

The Q Learning agent learns optimal policy in all the training environments for all subtasks. However, the Q Learning agent appears to only to perform well for states it has already visited in the training environment but is unable to perform for unvisited state i.e. it is unable to learn a general approach for the environments. This can be observed in the case of cliff walking where the Q Learning agent in one of the variations starts at the center and is still able to perform decently.

In conclusion, though NLRL learns near-optimal policies, they have superior interpretability and generalizability.

Work Division

Aishwarya: Building State Encoding, Built baseline for comparison
Kritika: Existing code Debugging, Built demo example
Pooja: Building New Environment, Built demo example
Together: Report, Presentation, Experimentation

References

1. Neural Logic Reinforcement Learning, link to paper: https://arxiv.org/pdf/1904.10729.pdf
2. Author's Source code: https://github.com/ZhengyaoJiang/NLRL
3. Learning Explanatory Rules from Noisy Data: https://arxiv.org/pdf/1711.04574.pdf
4. Tutorial: https://www.youtube.com/watch?v=_wuFBF_Cgm0
5. Inductive Bias: https://en.wikipedia.org/wiki/Inductive_bias
6. First Order Logic: https://en.wikipedia.org/wiki/First-order_logic
7. https://math.stackexchange.com/questions/9554/whats-the-difference-between-predicate-and-propositional-logic
8. https://www.doc.ic.ac.uk/~cclw05/topics1/summary.html