Alien

Alien is a family of three reinforcement learning algorithms: Slow Alien, Improved Alien and Humanized Alien. This document describes Slow Alien and Improved Alien.

Slow Alien - highly general reinforcement learning algorithm that works only in theory due to being too computationally expensive and slow learning rate.

Improved Alien - improved version of Slow alien, still highly general but better in terms of computational expensiveness and learning rate. Improved Alien can learn tasks like: playing FrozenLake, playing chess, machine translation. Improved Alien wouldn't be able to learn (at least in short time) human-like tasks like: visual recognition, answering questions about distant past.

Humanized Alien (invention in progress) - improved version of Improved Alien such that the algorithm is better at human-like tasks. Humanized Alien can learn what Improved Alien is able to learn and can also learn human-like tasks like: visual recognition, answering questions about distant past. Humanized Alien can be used for creating human-level artificial intelligence.

The current progress is that:

1. Slow Alien has been invented but hasn't been implemented (and never will be).
2. Improved Alien has been invented (although it will never be invented in 100% because you can always improve it) and has been partially implemented.
3. Humanized Alien has been partially invented and hasn't been implemented.

Below, I am describing the characteristics of each algorithm. When I use the words "enough", "sufficient", "is not too X", "doesn't require too much X", I mean "enough, given that we want to use this algorithm for creating human-level artificial intelligence", "sufficient, given that we want to use this algorithm for creating human-level artificial intelligence" or "is not too X, given that we want to use this algorithm for creating human-level artificial intelligence". The meaning of these characteristics and why these characteristics are important can be found in "Plan for creating human-level artificial intelligence".

Slow Alien has the following characteristics:

1. The algorithm is general enough.
2. The model that the algorithm uses can represent every possible policy.
3. The algorithm can represent and learn loop or recurrence.

Slow Alien lacks the following characteristics:

1. The algorithm doesn't require too much human supervision.
2. The algorithm is good enough at transfer learning.
3. The algorithm is not too computationally expensive.

Improved Alien has the following characteristics:

1. The algorithm is general enough.
2. The model that the algorithm uses can represent every possible policy.
3. The algorithm is good enough at transfer learning.
4. The algorithm can represent and learn loop or recurrence.
5. The algorithm is not too computationally expensive.

Improved Alien lacks the following characteristics:

1. The algorithm doesn't require too much human supervision.

Humanized Alien (invention in progress) has the following characteristics:

1. The algorithm is general enough.
2. The model that the algorithm uses can represent every possible policy.
3. The algorithm doesn't require too much human supervision.
4. The algorithm is good enough at transfer learning.
5. The algorithm can represent and learn loop or recurrence.
6. The algorithm is not too computationally expensive.

Humanized Alien (invention in progress) doesn't lack any characteristics.

Facts about Alien

Input of Alien algorithm must be an array of integers (it can be a multidimensional array).

Alien algorithm treats the input as categorical data. But it can be modified to treat the input as relative values.

Alien algorithm works with discrete actions space. But it can be modified to work with continious actions space.

Alien algorithm is a reinforcement learning algorithm. But it can be modified to be a supervised learning algorithm.

Alien algorithm is a reward-to-go algorithm (it has a finite horizon after which it doesn't care how much reward it gets). But it can modified so that it's not a reward-to-go algorithm.

Alien algorithm is a model-free reinforcement learning algorithm. But it can be modified to be a model-based reinforcement learning algorithm.

Slow Alien

Slow Alien is an algorithm that works only in theory. In practice, it's too computationally expensive and requires too much human supervision to be practically usable.

The purpose of the existence of this algorithm is to make explaining Improved Alien algorithm easier (it's easier to understand Improved Alien if you understand Slow Alien first).

Slow Alien doesn't need to be fast, this will be later improved in Improved Alien algorithm, so ignore the fact that it's slow.

Internal state

Agent in Alien algorithm uses internal state. Internal state consists of tapes. Tape is an array of integers (from 0 to some number) with a pointer (or pointers) that indicates a cell in an array. Some tapes can consist of only one cell and they don't have a pointer because the pointer is useless in this case.

There are different kinds of tapes:

1. Memory tape - a one-dimensional tape that serves as a working memory. The values in this tape can be modified by the agent. At the beginning all values in the tape are 0.
2. Observation tape - a tape that contains the observation of the agent. The shape of the observation tape is equal to the shape of observation.
3. Location tape - a tape that relates to another tape and represents the position of the pointer in the another tape, value on a location tape should be treated as a relative value, not categorical data.

The agent can execute external actions and internal actions. External actions are normal actions from the action space defined in the Markov decision process. Internal actions are additional actions for altering the internal state. External actions are the actions that have the impact on the world. Internal actions are the actions that doesn't have any impact on the external world, but they alter the state of the mind of the agent (internal state).

In order to better understand the concept, you can give analogy to human mind. External actions of a human mind are actions like: moving a hand, make a sound with your mouth etc. The internal actions of a human are actions like: think a thought, imagine something, remind something... External actions are those actions that have the impact on the external world. Internal actions doesn't have any impact on the state of the world, they only modify the state of the mind but they

are also needed for the purpose of figuring out the right external action to take.

On default, the agent has the following internal actions:

1. Actions for moving the tape pointers in each possible direction.
2. Actions for incrementing and decrementing the value indicated by the pointer on a memory tape (or alternatively: actions for setting each possible value in the cell indicated by the pointer).
3. Actions for moving the pointer to a starting point - the first cell on the tape (or sometimes to the first or last cell in a given axis, in case of observation tape; starting points are customizable and they can depend on the task and what observation represents).
4. Actions for dynamically creating a new memory tape (with moving the pointer into the new memory tape) and for removing it (and putting the pointer back to where it was on the previous tape).

In each tour, the agent perceives only the internal observation - an array consisting of the values indicated by the pointers. The values in internal observation are always in the same order, that is internal_observation[0] always represtent the value indicated by the observation tape pointer, internal_observation[1] always represent the value indicated by the first memory tape, internal_observation[2] always represent the value indicated by the second memory tape etc.

The agent doesn't have a direct acccess to the external observation (the actual observation), it takes as an input the internal observation and outputs internal or external action in each tour.

The above information can be summarized with the following diagram:

The agent can have more than one input/sense and the number of observation tapes is equal to how many inputs/senses it has. For example if the agent can see and hear (two senses), then it can have two observation tapes: one for visual input and one for audio input.

For many tasks it's better to have many 1-cell observation tapes, one tape for each integer from the observation array, than having one observation tape with the entire observation. The agent doesn't have to learn how to use the tape then, so it will learn things quicker. For example, if you use Alien algorithm for playing chess on a 8x8 board, it would probably make more sense to have 64 observation tapes with only one cell representing the state of each position on the board, instead of having one observation tape with shape (8,8) or (64,). If you have 64 tapes with one cell, then the agent doesn't have to learn using the tape which would probably require a lot of time.

In the context of Alien algorithm, we have internal tours and external tours. Tour is one episode of the agent accepting observation and outputing an action. External tour is one episode of the agent accepting external observation (the actual observation) and outputing one of the external action.

Internal tour is one episode of the agent accepting internal observation (the observation of the values on tapes indicated by the pointers) and outputing one of the external or internal actions.

Memory sections

In Alien algorithm, the agent has the possibility to create new memory tapes dynamically. There is no fixed number of memory tapes, the agent can create new memory tapes and remove them.

The following is the exact description of how it works. There are 2 (or more if you want) memory sections. A memory section is a container for memory tapes. The number of memory sections remains the same all the time. At the beginning there is only one memory tape in both sections. The agents is equiped with internal actions for creating a new memory tape in both sections (there is a seperate action for creating a new memory tape in the section 1 and a separate action for creating a new memory tape in the section 2), as well as internal actions for removing the last tape. The attention of the agent (the cell indicated on the memory section by the pointer and what the agent perceives) is always at the last created tape in the section 1 and the last created tape in the section 2. If the agent creates a new memory tape, the beginning values in each cell are equal 0 and the pointer is on the first cell. If the agent removes a memory tape, the pointer moves back to the previous tape and the pointer moves back to the position where it was on that memory tape before creating that new memory tape. For example, if the agent has 2 memory tapes in the section 2 and the pointer is on 5th cell of the 2nd memory tape, then if agent creates a new memory tape, executes some actions on it and removes the new tape, then the pointer moves back to the 5th cell on the 2nd memory tape.

What is it for? Why does the agent is equiped with the possibility to create new memory tapes? Alien is a machine learning algorithm - it's an algorithm that learns an algorithm. Many algorithms that the agent needs to learn consists of other subalgorithms. For example, exponentation relies on multiplication which relies on addition which relies on incrementation. In order to multiply numbers 3 and 4, the agent needs to execute the procedure for adding number 4, 3 times. If the agent can't dynamically create tapes, then when doing addition in order to do multiplication, the agent must be careful not to mess up with the information that it stores in the memory for multiplication. In order to do that it has to go down with the pointer to a place where there is no important information and then move back to where it was so that it can do the rest. This going down and back could be difficult to learn. If the agent can dynamically create new tapes, then it can do multiplication which needs to save some information in memory and when it needs to add two numbers, then it can create a new memory tape, add two numbers, save the result to the memory tape from the other section and move back to the previous memory tape. After doing so, the state of the memory will be exactly the same as it was before addition but with the difference that on the second section, you will have the result of addition. Maybe that explanation of why it's needed doesn't have too much sense as far, but possibly it will make more sense after you learn how the algorithm works (i.e. how the agent chooses the action to take).

Internal trajectory

In reinforment learning, trajectory means a sequence of states and actions. In the context of Alien algorithm, internal trajectory means a sequence of internal observations and actions (that can be internal or external).

Internal trajectory can be represented as a two-dimensional array with shape `(length_of_internal_observation + 1, internal_tours_count)`, where length_of_internal_observation is the length of internal observation and internal_tours_count is the number of past internal tours. "+ 1" in the above shape comes from the fact that we also need to store the actions, not only internal observation.

Example:

Internal_trajectory[0][n] equals the value indicated on the input tape in the n-th internal tour.
Internal_trajectory[1][n] equals the value indicated on the memory tape 1 in the n-th internal tour.
Internal_trajectory[2][n] equals the value indicated on the memory tape 2 in the n-th internal tour.
Internal_trajectory[3][n] equals the id of the action taken in the n-th internal tour.

This is assuming that we have 3 tapes in the internal state: one observation tape and two memory tapes, but the tapes might be different than this.

Example of an internal trajectory (represented as a variable int_tra):

```
| Cell         | Value     | Meaning                         |
|--------------|-----------|---------------------------------|
| int_tra[0]   | 1 2 3 4 5 | Observation tape indicated values |
| int_tra[1]   | 2 3 4 1 2 | Memory tape 1 indicated values |
| int_tra[2]   | 3 2 2 2 0 | Memory tape 2 indicated values |
| int_tra[3]   | 2 3 3 3 5 | Actions taken                   |
```

When I later say "the current internal trajectory", I mean the internal trajectory which is a sequence of internal observations that the agent received as an input and actions that the agent took. For example, if we had 3 internal tours as far, and the internal observations were consecutively: [1, 2, 3], [2, 1, 3], [0, 1, 2] and the actions were consecutively: action 1, action 2, action 3; then current internal trajectory is: [1, 2, 3], action 1, [2, 1, 3], action 2, [0, 1, 2], action 3.

Alien notation

Alien notation is a way to represent a policy or part of it. Alien notation policy or part of policy is a sequence of instructions from the following set of possible instructions:

1. Execute one out of the internal or the external actions.
2. If the indicated value on some tape is equal to some value, then execute some sequence of instructions.
3. While the indicated value on some tape is equal to some value, then execute some sequence of instructions.

Example of an algorithm/policy represented in alien notation:

```
[2 3 4 8 1 if(2, 7, [1 2 3 4 while(1, 2, [1 2])])]
```

Where:

1. `[x y ... z]` means a sequence of actions `x`, `y`, some other actions and `z` (actions can be internal or external). We give an id to every action (internal and external) and the number represents the id of the action.
2. `if(a, b, c)` means if the indicated value of the tape with number `a` equals `b`, then execute the sequence of instructions `c`.
3. `while(a, b, c)` means while (as long as) the indicated value of the tape with number `a` equals `b`, execute the sequence of instructions `c`.

Using only these 3 types of instructions that were mentioned above we can represent every possible policy that can be represented with imperative programming language (including policies that save something in memory between outputing actions), assuming that we have right internal state system (internal state system = internal state mechanism + internal actions to manage it). The internal state and set of internal actions proposed above are sufficient for the above statement to be true. // proof needed

For each existing reinforcement learning task, there exist an optimal policy that can be represented with imperative programing language (// proof needed). Therefore, alien notation can represent optimal policy for each reinforcement learning task (by reinforcement learning task I mean MDP that ends at some point).

Storithm

Storithm is basically a policy or part of it represented in alien notation. The name comes from connecting the words "state" and "algorithm" and I needed to introduce a new word for that as saying "algorithm" can be confusing because if I say "algorithm", then you don't know if I mean a storithm or the Alien algorithm. Also, there is a difference between storithm and algorithm in alien notation, but that difference will become clear later.

We can say that storithm S occurs in an internal trajectory T at the positions from A to B, when all of the actions that were taken in the internal trajectory T at positions from A to B could be

generated by following the storithm S in such way that we go inside each conditional statement and loop (at least one time).

For example, if we have a slice of internal trajectory like that:

```
| Cell         | Value     | Meaning                         |
|--------------|-----------|---------------------------------|
| int_tra[0] | 1 2 3 4 5 | Observation tape indicated values    |
| int_tra[1] | 2 3 4 1 2 | Memory tape 1 indicated values |
| int_tra[2] | 3 2 2 2 0 | Memory tape 2 indicated values |
| int_tra[3] | 2 3 3 3 5 | Actions taken                   |
```

Then we have a storithm like that occuring in this slice:

`[2 if(0, 1, [3]) 3 3 5]`

The above storithm means:

1. Execute action 2
2. After that if the value indicated on tape no. 0 (the observation tape) equals 1 then execute 3.
3. After that execute actions 3, 3 and 5.

Other examples of storithms that occur in this slice of internal trajectory:

[2 3 3 3 5]
[2 3 while(2, 2, [3]) 5]

Example of a storithm that doesn't occur in the above slice:

[2 3 if(2, 8, [4 3 5]) 3 3 5]

The above storithm doesn't occur in the slice because there is a conditional statement with the condition that hasn't been met (if the value on the tape 2 equals 8 then execute 4 3 5). Storithm occurs in the slice if we can generate the actions taken in that slice by following the storithm **in such way that we go inside the body of each conditional statement and loop (at least one time)**. In case of this storithm, there is a conditional statement `if(2, 8, [4 3 5])` where the condition is not met, therefore this storithm doesn't occur in this slice.

The name "storithm" comes from connecting the word "state" and "algorithm". It can be clear why "algorithm" but it can be unclear why "state". This is because storithm can represent some part of state of the world (feature). For example, you can have a storithm that recognizes an eye, i.e. occurrs only if there is an eye in the input. This storithm can consist of execution of the algorithm for recognizing and eye and putting 1 in a cell of the memory if there is an eye or 0 if there is not.

At the end this storithm would have if condition with empty body which causes this storithm to occurr only if there is an eye in the input. Maybe that name should be connection between the words "feature" and "algorithm" instead of "state" and "algorithm", but the name "forithm" or "featorithm" would sound even more ridiculous.

// storithm can have complete_while uncomplete_while

Fusion

Fusion is a special type of storithm that consists of a set of other storithms.

We can say that storithm F of the type fusion occurs in an internal trajectory T at position from A to B, if and only if one of the storithms that it consists of occurs at this position (from A to B) in the internal trajectory T.

Fusion storithm allows us to represent every possible alien notation policy. I.e. for each alien notation policy or part of policy P, there exists a fusion storithm F that will occur at the position from A to B in internal trajectory T, if and only if the agent acted according to the policy or part of policy P from the position A to B in the internal trajectory T.

Interpretation and interpreting

Interpretation is a set of storithm occurrences.

Storithm occurrence is a tuple of:
a) storithm - a storithm that occurs,
b) start - an integer representing position in internal trajectory at which the storithm occurrence starts.
c) end - an integer representing position in internal trajectory at which the storithm occurrence ends.

Length of the storithm occurrence is defined as:

length(occurrence) = occurrence.end - occurrence.start + 1.

Intepreting is a procedure that finds occurrences of storithms (either all storithms or only some set of storithms) in the current internal trajectory and adds them to the interpretation stored in the memory of the program.

In Alien algorithm, we store the current internal trajectory and its interpretation in the memory of the program.

Estimator and predictors

In the algorithm, we use an estimator - something that predicts some value based on some predictors.

Put simply, estimator is a class with two methods:
a) fit(predictors, true_value, sample_weight=1)
b) predict(predictors): float

What do we use estimator for in the algorithm and what do we try to predict? We use it to predict on-policy action-value - the return that we will get if we choose an action. The predictors based on which the estimator makes that prediction is if some storithms occured or not. In other words, the algorithm estimates action-value based on the storithms occurrences that we have found in the internal trajectory. I will explain that in more detail later.

In normal linear regression predictors can have different activations. In the context of Alien algorithm, predictors doesn't have activation, their activation is always 0 or 1 (because the storithm can either occur or not). If a predictor has activation 0, then you simply don't pass it to the fit() method, so `predictors` argument is an array of instances of class Predictor. All of these predictors are assumed to have activation = 1 in the given sample.

In Alien algorithm, I currently use Gradient Descent as estimator and in case of this estimator, Predictor class have only one attribute - coefficient. But we can use also some other estimator and then it can have different attributes.

// coefficients are zeros at the beginning

// decresing learning rate if there are a lot of predictors

Storithm repository

Storithm repository is a hash table in which the key is a storithm and value is a tuple of two arrays of predictors (with the index starting from 0 and the length is equal to horizon).

// tell about what hash function we can use

`storithm_repository[storithm].internal[i]` stores a predictor which is used to predict the return that we will get after `i` internal tours after the occurrence of the storithm. If the coefficient of that predictor is high, it means that if that storithm occurs, the return starting from `i` internal tours after that occurrence is likely to be high. If the coefficient of that predictor is low, it means that if that storithm occurs, then the return starting from `i` internal tours after that occurrence is

likely to be low.

`storithm_repository[storithm].internal` is an arrray with keys from 0 to MAX_INTERNAL, where MAX_INTERNAL is a hyperparameter of the algorithm.

`storithm_repository[storithm].external[i]` stores a predictor which is used to predict the return that we will get after `i` external tours after the occurrence of the storithm. If the coefficient of that predictor is high, it means that if that storithm occurs, the return starting from `i` external tours after that occurrence is likely to be high. If the coefficient of that predictor is low, it means that if that storithm occurs, then the return starting from `i` external tours after that occurrence is likely to be low.

`storithm_repository[storithm].external` is an arrray with keys from 0 to MAX_EXTERNAL, where MAX_EXTERNAL is a hyperparamet of the algorithm.

When a storithm S occurs at the position `a` in the external trajectory T and `storithm_repository[S][i] = p`, then we say that predictor `p` occurs at the position `a + i` in the external trajectory T.

Predictors occurring at the position `i` are used to estimate how much return we will get starting from the position `i`.

Steps of the algorithm

After introducing some concepts that are important for the algorithm, we can finally introduce the algorithm itself.

The steps of the Slow Alien algorithm are as follows. In every internal tour, the algorithm does consecutively:

1. Interpret - find all occurrences of storithms in the current internal trajectory.
2. Predict - estimate action-values for all actions (both external and internal).
3. Choose - sample one action with the probabilities generated by softmax function accepting action-values as input and execute that action.
4. Fit - update coefficients of some predictors.
5. Merge - create new fusion storithms.

Now, let's move through those steps in greater detail.

Interpret

In "interpret" part, we take the current internal trajectory and find all storithms that occur in this

internal trajctory (at any positions) and we also find occurrences of all created fusion storithms (they get created in "Merge" part). We use brutforce algorithm to achieve that. The number of storithms that we need to find is finite. The number of slices of the current internal trajectory at which storithms can occur is finite as well. Given a slice and a storithm, we can verify if the storithm occurs at this slice of the current internal trajectory, therefore brutforce algorithm can do that.

Don't worry that the algorithm is slow because in Improved Alien, we will use a different algorithm and, as stated before, Slow Alien is only for making explanation of Improved Alien easier.

When we run interpreting procedure, we only need to find storithms that have their end in the last internal tour and add them to the previously found storithms (in previous tours), we don't need to look for storithms that end before the last internal tour. This is because we run interpreting procedure in every internal tour, so if in every tour we run this procedure and it finds only storithms ending in the last tour, then that with previously found storithms will consitute all storithms that can be found in the current internal trajectory.

Predict

In Predict part, the algorithm tries to predict how much return we would get if we choose each action.

We iterate through all possible actions (internal and external) and for each action we do the following things consecutively:
a) We temporarily* add the iterated action at the end of the current internal trajectory.
b) We run interpreting procedure - we find storithms that occur in the current internal trajectory supposing that we took the given action. We add these storithm occurrences to the interpretation temporarily* . As we add storithm occurrences, we also add predictor occurrences (temporarily*) - we have two arrays with internal and external predictors that represent what predictor occur at which position and we update that array with the predictors of a storithm when we add the storithm occurrence.
c) Using estimator, we predict the return that we will get if we choose the given action. We estimate that based on predictors that occur in the current internal tour and in the current external tour supposing that we took this action.
d) We clear all storithm occurrences and predictor occurrences that were added temporarily. We do this because we added them to interpretation only to see what will happen if we take that action.

`*` Temporarily means that we add them in such way that we mark them as temporary so that we can remove them later.

Put simply, what the algorithm does is that for each possible action, it simulates that it takes this

action, then it runs interpreting procedure to see what storithms it will have if it chooses this action and then it estimates what the return will be if it chooses this action based on the predictors that would occur in the last tour. After that we clear every storithm occurrence and predictor occurrence that the interpreting procedure has added in this iteration because it added them only temporarily to simulate what will happen and estimate the return.

This code can help with understanding:

```
def _predict(self):
    actions = self._internal_actions + self._external_actions
    action_values = {}
    for action in actions:
        self._interpretation.internal_trajectory.\
            add_at_the_end_temporarily(action)
        self._interpretation.interpret_temporarily()
        self._interpretation_cache[action] = self._interpretation.\
            temporary_occurrences
        internal_predictors = self._interpretation.\
            predictors_in_internal_tour(self._current_internal_tour)
        external_predictors = self._interpretation.\
            predictors_in_external_tour(self._current_external_tour)
        predictors = internal_predictors + external_predictors
        action_values[action] = Estimator.predict(predictors)
        self._interpretation.clear_temporary()
    return action_values
```

As you can see in the above code, we can save the temporary interpreted occurrences of storithms in memory because if we later choose that action then we don't have to do interpretation again, we can reuse the occurrences that we have found when doing the simulation.

If we know the model of the environment, i.e. we know what the observation will be after each action (like when playing chess where we know the rules and we can predict the observation after each action), then in the first step of the process that was described above, we can add not only the given action at the end but also the observation that will be after taking the action. The idea is that we simulate taking the action and then we estimate the expected return.

Choose

Choose part chooses one action that the agent will take. It chooses that based on the on-policy action-values (how much return we will get if we choose an action) that are calculated in Predict part.

This part of the algorithm is very simple - the algorithm samples one action with the probabilities generated by softmax function (based on action-values) and executes that action.

This code can help understanding:

```
def _choose(self, action_values):
    actions = list(action_values.keys())
    values = list(action_values.values())
    probabilities = self._softmax(values)
    action = actions[choice(range(len(action_values)), p=probabilities)]
    return action

def _softmax(self, logits):
    logits = [logit * self._softmax_temperature for logit in logits]
    max_ = max(logits)
    exponentials = [exp(logit - max_) for logit in logits]
    exponentials_sum = sum(exponentials)
    return [exponential / exponentials_sum for exponential in exponentials]
```

There are two options how to set softmax temperature. The first option is that softmax temperature is a hyperparameter that is set by the user of the algorithm. The other, recommended option is that the user of the algorithm provides two more information to the algorithm (besides observation and reward) and these are target return and antitarget return and the algorithm sets softmax temperature based on these two information. Target return is the maximal return that the agent can achieve from that point. Antitarget return is the minimal return that the agent can achieve from that point. Based on these two information, it's possible to automate setting right softmax temperature. Assuming that the user of the algorithm is the one that chooses how much reward to give, the user should know the value of target return and antitarget return. If it doesn't know, they can choose some estimation, target return and antitarget return doesn't need to be set perfectly accurrate.

Fit

The agent learns with the latency equal to the horizon. That means that if the current external tour is 20 and the horizon is 5, then in Fit part of the algorithm, the algorithm learns about the choice that it made 5 external tours ago. This is because if the horizon is 5, then you know the return after 5 tours and only then you can completely see if your choice of the action was good or not.

It's probably possible to modify the algorithm so that it learns without this latency but this would add additional confusion to the explanation of the algorithm.

We execute the Fit part of the algorithm only after the agent took an external action (when we get some reward). If the last action that agent took was an internal action, then we don't receive any reward and we have nothing to learn.

In Predict part, the algorithm has made predictions about what the return will be if we choose each

action. It has chosen one action and after horizon number of external tours, the return is finally known. Now, it's time to learn. So the algorithm gets back to the predictions that it has made in the tour before horizon number of tours, it takes the predictors that it then used to make the prediction (in Predict part), the return is also known and the algorithm simply fits that using estimator which adjusts the predictors (its coefficients) so that the next time it will make a better prediction.

This is what we do in fit() part:

```
def _fit(self):
    if self._current_external_tour <= self._horizon + 1:
        return None
    start = self._external_actions_positions[-self._horizon - 2] + 1
    end = self._external_actions_positions[-self._horizon - 1]
    return_ = self._returns[-1]
    for internal_tour in range(start, end + 1):
        external_tour = self._external_for_internal_tour[internal_tour]
        internal_predictors = self._interpretation.\
            predictors_in_internal_tour(internal_tour)
        external_predictors = self._interpretation.\
            predictors_in_external_tour(external_tour)
        predictors = internal_predictors + external_predictors
        Estimator.fit(predictors, return_, self._sample_weight)
```

Why is there a for loop from start to end? Because there are many internal tours where we tried to predict the last return. It might get clear at the below example.

Let's suppose the agent took the following actions:

iiieiiiieieiiie

Where:
i - means an internal action,
e - means an external action.

We took the external actions in the following internal tours: 3, 8, 10, 14 (assuming that we count from 0).

The current internal tour is 15. The current external tour is 4.

Let's suppose that we have received the following rewards consecutively after each external action: 2, 3, 1, 4.

Let's suppose that the horizon = 2 (we count from 0). That means that we take care of only 3 last actions to calculate the last return.

The last return = 3 + 1 + 4 = 8

Now, the internal tours where we made the prediction of what will be that return are the internal tours that are from 4 to 8. In these tours we made a prediction about what will be the return after 3 (horizon + 1) external tours. We are now 3 external tours after and we can fit the true value into the predictors. That's why we iterate through tours from 4 to 8 and we execute fit method with predictors from the given tour and the return.

You can also see that we pass sample weight to the fit method. What is the value of that sample weight? In the first tour it is 1 and in every internal tour we multiply it by hyperparameter TIME_IMPORTANCE_FACTOR. We do this because the recent samples should be taken more into account than the samples far from the past. Why is that? For two reasons: // this should be in "why does it work"

1. Predictors are used for predicting on-policy action-values. On-policy action-value is an expected return that the agent will get if it chooses an action and it acts according to a policy forever after. And when we estimate on-policy action-values in predict part, then what policy do we mean? We mean the current policy of the agent. And the current policy changes as the algorithm learns. For this reason we should value more the recent samples than the samples from the distant past because they tell us more about what the on-policy action-value is, given the current policy. If we have an old sample, then our policy could greatly change from that moment so that sample is not that much important now.
2. The task can change as well and for this reason we value recent samples more than the samples from the distant past.

Merge

In every tour, we sample two random storithms from the set of all storithms that we have added to storithm repository as far and we create (add to the storithm repository) a new fusion storithm that consists of these two sampled storithms. We do this MERGE_COUNT number of times, where MERGE_COUNT is a hyperparamter. Later, in the "interpret" part we find all storithms that we have created.

The coefficients of predictors of newly created storithm are always equal 0 at the beginning.

How and why does it work?

After understanding what steps the algorithm takes, it's not clear and obvious why it should work, so that requires a bit of an explanation.

It's difficult to prove that it works because it's difficult to define what it means that a reinforcement

learning algorithm works and I can't think of any good definition. So instead, I will show you an example of how this algorithm can learn to be good at some reinforcement learning task. It can become good at every other reinforcement learning task in an analogical way.

Firstly, I'll give you my definition of a reinforcement learning task.

Reinforcement learning task - a tuple of:

1. Input set - a set of all valid inputs (observations). Each input is an array of integers. The array can be multi-dimensional.
2. Ouput set - a set of all valid outputs (actions to take). Each ouput is an integer.
3. Next input function - accepts a sequence of valid inputs and outputs (ending with output) as an argument and returns probability distribution of what is the next input and the probability that the task will end at this point.
4. Reward function - accepts a sequence of valid inputs and outputs (ending with output) as an argument and returns probability distribution of reward received by the agent after given trajectory.

Additionaly, a valid reinforcement learning task must end at some point. If the Next input function is such that the task can last infinitely, then it's not considered to be a reinforcement learning task.

In other words, reinforcment learning task is almost like MDP, but with some differences:

1. It doesn't need to have the Markov property.
2. Input/observation can be only an array of integers.
3. Output/action can be only an integer.
4. It must end at some point.

Imagine that you have a following task:

In every tour, the observation of RL agent are two random numbers. The RL agent is rewarded with 1 point, when it outputs the action corresponding to the sum of these two numbers. Otherwise, it doesn't get any reward.

Using this task as an example, I'll show you what happens if you apply the Alien aglorithm to a reinforcement learning task repeatedly. "Repeatedly" means: if the task ends, then you reset the task and start the task once again, but the RL agent continues with what it has learned in previous tasks. You repeat the task infinitely.

At the beginning, the agent will output random actions as it hasn't learned anything yet.

For each reinforcement learning task, there exists an algorithm that is an optimal policy for this task (see Lemma 1 below for the proof of that).

The task of adding number is not an exception - there exists an algorithm being an optimal policy for the task of adding two numbers. This is the algorithm that adds two numbers and outputs the action that corresponds to the sum of them.

If the internal state system (internal state system = internal state mechanism + internal actions to manage it) is right (i.e. it contains the internal actions that I have defined above and works in a way I have described above), then we can represent every possible policy using alien notation. I.e. if the internal state system is right, then for each reinforcement learning task T, for each policy P that applies to this task, there exists alien notation policy R such that for each possible input of the task T, policies P and R output exactly the same actions (see Lemma 2 below for the proof of that). // examples to each paragraph please

For example, there exists a policy that takes two numbers and outputs the actions that corresponds to the sum of these two numbers. We can represent that policy in alien notation. If the internal state system is right, then we can represent every possible algorithm using alien notation. In alien notation, we don't have a operator for adding numbers, but we can add two numbers, let's suppose A and B, by a series of internal actions that increment the number A, B number of times. And in order to do that B number of times, we can decrement B as long as it's not equal to 0. Alien notation can't represent "inequal" but it can do something if a value is inequal to 0 by assigning 1 to another cell only if it equals 0 and then doing that something if that cell still equals 0 (which means that the previous value is inequal 0) assuming that the value in that cell was 0 at the beginning (which can be achieved by creating a new tape which will have 0 in every cell at the beginning). You don't need to understand exactly what I have written in this paragraph, but the point is: if the internal state system is right (and the one that I have described above is right), then alien notation can represent every possible algorithm that is possible to be represented using imperative programming language.

For each alien notation policy or part of policy, there exists a storithm that represent that policy or part of policy. I.e. for each alien notation policy or part of policy P, there exists a storithm S that will occur at the positions from A to B in internal trajectory T, if and only if the agent acted according to the policy or part of policy P at the positions from A to B in the internal trajectory T (see Lemma 3 below for the proof of that).

For example, if there is a policy that adds two numbers provided as input (observation) and outputs the action corresponding to the sum of them, then there is also a storithm that will only occur if what agent did is adding two numbers provided as input and outputing the sum of it and it will occur at the position in the internal trajectory where the agent did that.

As stated above, every policy can be represented with alien notation and every alien notation policy can be represented with storithm. Therefore, every policy can be represented with a storithm.

For each reinforcement learning task there exists an optimal policy. Therefore for each

reinforcement learning task, there exists a storithm that can represent optimal policy (because for each policy there exists a storithm that can represent it).

If we execute the Alien algorithm infinetely for a task, then every possible storithm will get created at some point (see Lemma 4 below for the proof of that). Precisely speaking, every possible storithm that applies to this task, for example if a storithm outputs an action with id = 54 and there are only 30 actions (including internal and external actions) in the task, then we don't count that storithm when we say "every possible storithm". When I say "every possible storithm", I mean every possible storithm that makes sense for the task that we run the algorithm on.

Therefore, the storithm representing an optimal policy will get created at some point.

For example, the storithm that represents optimal policy for the task of adding two numbers will get created at some point.

If the task is to add two numbers and the agent is rewarded with 1 point when giving the right answer, then every time when storithm representing optimal policy occurrs (let's denote it with S), the agent will get 1 point.

Some people might say that the occurrence of the storithm that represents the optimal policy will not always lead to getting 1 point because if that storithm will lead to the optimal action or not depends on what the state of the memory is when starting the occurrence of the storithm. For example, if the state of the memory is clean (every cell contains 0 and the pointer is at the beginning), then the storithm representing optimal policy will lead to the optimal action. But if the state of the memory has been changed, then if we execute the optimal policy after that, then it won't necessarily lead to the optimal action and getting one point. But if that occurrence doesn't always lead to the optimal action, then it means that it's not the storithm representing optimal policy. But there exists also another storithm that starts with creating a new memory tape and then execution of the optimal policy and that storithm will always lead to the optimal actions because if you create a new memory tape then you have a clean state and that's the actual storithm representing optimal policy.

The coefficient of the predictor `storithm_repository[S].internal[0]` will grow because this predictor occurrs every time right after the occurence of the storithm S (the storithm representing the optimal policy). And every time when that occurrence happens, the agent takes optmal action and gets 1 point.

This will lead to the situation that whenever the storithm S has a chance to occur, then the agent will expect to get about 1 point of return more than when choosing every other action that doesn't lead to the occurrence of the storithm S. Therefore, after some time, if the agent is in the situation when it is one action from completing the occurrence of the storithm S, the agent will be more likely to choose the action that will complete the storithm S because the action-value will be higher for that action.

This will lead to a situation that whenever the agent is two actions away from completing the storithm S, it will choose the action that will lead to completing this storithm. This is because there will be also another storithm T that will be identical to storithm S but without the last action. Every time when the agent completes the storithm T, the agent will be very likely (assuming that the softmax temperature is high enough) to choose the action that leads to the storithm S (for the reason described in the previous paragraph). After some time, `storithm_repository[T].internal[0]` will grow because every time when the agent completes the storithm T, it is very likely to choose the action that leads to completing the storithm S which leads to getting 1 point of reward. The agent, when having an opportunity to complete the storithm T, will be more and more likely to do this as `storithm_repository[T].internal[0]` will grow more and more.

This will lead to a situation that whenever the agent is three actions away from completeing the storithm S, it will be very likely to choose the action that will lead to completing this storithm (for the same reason why it started to be likely to choose the right action two actions away from completing the storithm S). This process will continue with 4 actions away, 5 actions away and eventually the agent will learn to execute the entire process.

Some people might say:

"ok, but this will work if the optimal policy is for example like that:
[1 2 3 4]. What about if it contains some conditional statements or loops like that: [1 2 3 if(1, 0, [4 5]) 4 5] or that: [1 2 3 while(1, 0, [1 if(2, 0, [1 2 3]) 3]) 1 2 3]?".

When it comes to conditional statements, it will be same thing as I have described above. For example, if the optimal policy is like that: [1 2 3 if(1, 0, [4 5]) 4 5], then it will firstly learn that the occurrence of the storithm [1 2 3 if(1, 0, [4 5]) 4 5] leads to high return (by accidently executing this storithm through exploration), later it will learn that the storithm [1 2 3 if(1, 0, [4 5]) 4] leads to a high return, later it will learn that the storithm [1 2 3 if(1, 0, [4 5])] leads to a high return, later [1 2 3 if(1, 0, [4])]. And at that point when it is in a situation that it has already executed [1 2 3], it will also execute the action 4 after that if the condition (indicated value on tape 1 is equal 0) is met because this will lead to the occurrence of [1 2 3 if(1, 0, [4])] which already has a high coefficient. If the condition is not met, then it won't execute it (or at least not with that same probability as if the condition was met) because if it executes action 4 when the condition is not met, that won't create the storithm [1 2 3 if(1, 0, [4])]. Later it will learn that the storithm [1 2 3] leads to a high return etc...

With loops the situation is a bit more difficult. Let's suppose that the optimal storithm is like this: [1 2 3 while(1, 0, [1 if(2, 0, [1 2 3]) 3]) 1 2 3]. It will learn in an analogical way that the storithm [1 2 3 while(1, 0, [1 if(2, 0, [1 2 3]) 3])] leads to high return. At this point, when it's in the situation that it has executed [1 2 3 while(1, 0, [1 if(2, 0, [1 2 3])], it won't necesarily learn that executing action 3 after that leads to high return because if it leads to high return or not depends on if it follows that loop or not in its next actions. If it continues acting according to that loop, then it will receive

This way the agent will learn to add two numbers.

Obviously, before learning the optimal policy, the agent will learn less perfect policies as well.

The policies that lead to a solution are not the only important ones. For example, there might be a storithm that represent an algorithm that checks if a value indicated by the pointer is not equal to some value. That algorithm is useful in many tasks, so I expect that the coefficients of the predictors of this storithm are more likely to grow higher after applying the agent to many tasks. There are also storithms that consist of only one internal action. The storithms of the internal actions that should be used more often are more likely to get high value of a coefficient than others, so the algorithm can also learn after many tasks what actions are more probable to constitute a good policy. It can also learn that one action is often useful after another action, or that some action is useful in some situation.

"This will take insane amount of time" argument

Some people after reading the above explanation will be like:

"Ok, maybe it can theoretically learn adding numbers this way, but this will take insane amount of time to learn it this way. Let's say that the optimal policy for adding numbers (the policy that represents the algorithm for adding two numbers and outputting the sum of them) consists of 50 instructions in alien notation. Let's say that there is 20 possible actions (internal and external). In order to create a storithm that represents this optimal policy in the way that was described above you need to execute on average at least 20^50 operations. It's similar to trying to hack the 50 characters long password with bruteforce - it's not possible with the computational power that we have as far as I know."

There are a few things that I need to say in the answer to that argument.

Firstly, as I stated above, the Slow Alien algorithm is the foundation for Improved Alien algorithm which would learn that much quicker than Slow Alien. But Improved Alien still would take insane amount of time to realize the algorithm for adding numbers, given the task of accepting two numbers and being rewarded for outputing the sum of them. So it doesn't answer this problem. So keep reading.

Secondly, if in this task there is a small number of possible inputs (for example the numbers are always from 0 to 9), then the Improved Alien algorithm would learn to be good at this task quickly, not by actually realizing the algorithm for adding numbers, but by memorizing the good answers for each input. So the optimal policy that it would learn would be "if the input is 3 and 4, then execute the action that corresponds to 7, if the input is 2 and 1, then execute the action that corresponds to 3 etc...". That optimal policy is quick to learn because in order to represent that policy, the Alien algorithm needs to create 50 storithms (the number of all possible inputs) of the

length of one or two instructions (depends on if we count conditional statement as one instruction or two). And then learn how the occurrences of these storithms relate to the recieved return using Gradient Descent. This is totally possible to learn for Alien in quick time. "Ok, but this is not learning to add numbers, but actually just memorizing the right answers". Yes, you're right, so keep reading.

Thirdly, you can add an internal action to the Alien algorithm that will add two numbers and put the sum of it to some other cell in the memory. This way, the optimal policy will be much shorter because the agent will need to execute only one action and then it will be able to learn that policy in a quick time. But this is not really learning to add numbers, it's putting the calculator inside the mind of the agent. If the task would be different, for example to learn how to program instead of how to add numbers, then what will you do? Will you put a programmer inside the mind of the agent? Adding internal action like that can still be useful though, if you want to use the algorithm only for some arithmetic operations. But this doesn't answer the above argument, so keep reading.

Fourthly, the reason why it would take so much time for the Improved Alien algorithm to realize the optimal policy for this task (the algorithm that adds two numbers) is because it's impossible to realize this in a quick time. To realize the algorithm for adding numbers is just a very difficult task. This is for two reasons. Firstly, you have a task where you get two numbers and you need to output the sum of them and let's say that the agent has had 1000 tours as far, so it has 1000 examples that it can learn from. You need to find the function/algorithm that generates the output based on the input and fits into these examples. There are many functions/algorithms that could fit into these examples, that's the first problem. The second problem is that the only way how you can find that algorithm/function is by trying different algorithms/function or something of similar computational expensiveness. In other words, learning to add numbers (without just memorizing it) is a difficult task that requires insane amount of human supervision and computational power to learn **without any previous knowledge**.

This is also difficult for a human, because imagine that you go to an island with newly discovered people. These people don't know numbers. You show them two numbers and you give them a task to show the number that is the sum of these two numbers. You reward them for the good answer. How much time do you think it would take them to learn this task? They would learn that after some time a bit, but by memorizng the correct answer for each pair of numbers, not by actually understanding the math of it because the algorithm for adding numbers is complicated (and the Alien algorithm would behave in exactly the same way).

The following question arises (and in the answer to this question lies the final answer to the argument above):

If realizing the algorithm for adding numbers is so complicated and it's so computationally expensive, then how is that possible that a human can learn that? How is that possible that we are able to add numbers?

Because human learns adding numbers through transfer learning. It firstly learns to recognize objects, then it learns counting objects, then it learns the concept of number, then it learns comparing numbers, then it learns adding numbers (showed on example of adding candies on the table), then it learns multiplying numbers, then it learns exponentation. If you try to teach a new born infant without any knowledge of the world to add numbers, you will most likely fail or it will take insane amount of time. Human is able to learn adding numbers because human brain is constructed in such way that it reuses the policies that it has learned before and give preference to (i.e. will realize faster) the policies that base on the policies it has learned before. For example, if human learns how to compare numbers, then it will quicker realize how to sort numbers (which relies on comparing numbers).

The same is true about Improved Alien algorithm - if you try to teach this algorithm to add numbers (not by just memorizing the answers) without any previous knowledge of the world, then you will most likely fail or it will take insane amount of time (just like with an infant). But if you use transfer learning and teach the agent to understand the concept of number first, after that teach the agent to count objects and after that you teach it to add numbers, then the Improved Alien algorithm should be able to learn that in non-insane time. This is why transfer learning capability is an important condition that the algorithm must meet. For this reason, the Alien algorithm has been created with the intention to be good at transfer learning. In the section "Analyzing the criteria", I will describe why I expect Improved Alien to be good at transfer learning.

Is Improved Alien algorithm able to learn adding two numbers in non-insane amount of time? Yes, it is. But you need to teach the agent to add numbers in the same way that human learns - by transfer learning. You can't expect this algorithm to learn adding numbers in quick time without any previous knowledge, just like you can't expect that an infant can learn adding numbers without any previous knowledge of the world.

// the more difficult situation when it needs to store sometihng in memory...

Improved Alien

Improved Alien is Slow Alien algorithm with improvements that make the algorithm practically usable.

Improved Alien algorithm has been successfuly applied to FrozenLake environment from OpenAI gym, but it should be also able to do more difficult things like tasks playing chess, playing go (it needs to be used as a model-based algorithm for this purpose) or machine translation (it needs to be used as a supervised learning algorithm for this puprose).

A short summary of the improvements:

1. Procedures - we represent a storithm using other storithms.
2. Create - instead of creating (adding to the storithm repository) all storithms occurring in the current internal trajectory, we create random storithms that base on existing storithms occurring in the current internal trajectory.
3. Interpreting the smart way - we interpret the current internal trajectory using a smart, quick algorithm instead of bruteforce.
4. Importance - we store only the most important storithms and predictors so that we don't loose memory and computational power on something that is not important.
5. Small improvements - we reward taking external actions faster so that algorithm doesn't think too long and we store less predictors for a storithm.
6. Computational complexity - we do everything in such way that the algorithm doesn't have insane computational complexity and can be practically usable with the amount of computational power that we have.
7. Merging - merging works in a bit different way than in Slow Alien algorithm.

The following improvements are propositions that need to be investigated yet:

7. Importance based sampling - in the process of interpreting, we don't find all storithms that we have in storithm repository but we interpret in such way that we find important storithms with greater probability.
8. Getting more learning out of one sample - we use some tricks to greatly improve how many samples (and human supervision, if reward is given by human) the agent needs in order to learn what it needs to learn.

General map of the algorithm

Below, I show you the map of the Improved Alien algorithm (the most important functions in the algorithm) to give you a general idea of the steps of the algorithm.

1. init(hyperparameters)
2. act(observation, reward)
    i. prepare(observation, reward)
    ii. learn(reward)
        a. create(reward)
        b. merge(reward)
        c. fit(reward)
    iii. act_internally()
        a. predict()
            a. interpret()
        b. choose(action_values)

The two functions at the top (init and act) can be called by the user of the algorithm. The rest of

them are private functions that shouldn't be called by the user.

Explanation of the functions:

init(hyperparameters) - initializes the agent with the given hyperparameters.

act(observation, reward) - accepts observation and reward, returns an external action that the agent chooses to take.

prepare(observation, reward) - updates the variables that are used both by learn() and act_internally() functions, like: the current internal trajectory, the current return, the current internal tour, the current external tour and some others...

learn(reward) - learns, all it does is calling create(), fit() and merge() functions.

create(reward) - creates new storithms (adds them to the storithm repository) and predictors.

merge(reward) - creates new storithms of the type "Fusion".

fit(reward) - updates the predictors of the storithms that have occurred, it can also remove some storithms and predictors if they turn out to be unimportant.

act_internally() - returns an internal or external action that the agent chooses to take.

predict() - predicts on-policy action-values for each possible action to take and the current policy of the agent, returns the on-policy action-values.

interpret() - executes the procedure of interpreting on the current internal trajectory.

choose() - chooses an action to take, based on on-policy action-values, returns the chosen action.

Improvements

Now, I'm going to describe all improvements in greater detail.

Procedures

We represent storithms using other storithms. We don't represent a storithm as a sequence of instructions as we did in Slow Alien, but we represent it as a sequence of other storithms (sometimes it's a set of other storithms or some other combination of storithms, it depends of the storithm type).

This improvement is called "procedures" because a storithm is simply a procedure that calls other procedures (storithms).

For example, if we have a storithm that represents an algorithm for sorting numbers, then it can consist of a storithm representing an algorithm for comparing two numbers, as the procedure that sorts numbers makes use of a procedure that compares two numbers. Analogically, we can have a storithm that recognizes a face that can consist of another storithm that recognizes an eye, as recognizing a face requires recognizing an eye. Another example: a procedure for exponentation can use a procedure for multiplication which can use a procedure for addition which can use an internal action for incrementation.

If a storithm A is part of the sequence of storithm B, then we say that storithm A is a child of storithm B and storithm B is a parent of A. Storithm can have more than one parent.

All storithms that we have in the storithm repository constitute a directed graph. In this graph a single node represents one storithm. There is an edge from a node A to node B, if the storithm represented by node B is a child of the storithm represented by node A.

There is a special type of storithm called atom. Atom is a storithm that doesn't have any children. It has attributes instead. For example, action atom represents taking one action and the attribute of action atom is the action that it represents.

Storithms are stored in the memory as a sequence (or set or some other data structure) of other storithms, but every storithm can be converted to its string representation that can look more or less like that:

```
[1 2 3 if(2, 3, [1 2 3]) while(1, 3, [1 2 3])]
```

Storithms are stored in a storithm repository which is a hash table in which the key is a storithm and value is all needed information about the storithm. Hash function for storithms is calculated based on the string representation of a storithm. Equality of two storithms is defined by their string representation as well - two storithms are equal if they have identical string representation.

What are the benefits of the "Procedures" improvement:

1. We use significantly less memory because we don't have to store the entire sequence of instructions (string representation) but only the references to another storithms. If one storithm is contained in many other storithms, then these all storithms store the reference to this storithm, instead of repreating the whole sequence of instructions (string representation) in every storithm of these many storithms.
2. We use significantly less computational power because representing storithms in this way enables the algorithm to do many things in a smarter, quicker way.
3. In combination with other improvements, it improves transferring knowledge capability.

Types of storithms

There are the following types of storithms.

Action atom

It's a storithm consisting of one internal/external action. An action atom consisting of action A is considered to occur at the position P, if and only if the agent took action A in the P internal tour of the current internal trajectory.

This storithm doesn't have any children. The length of the occurrence of this type of storithm is always 1.

State atom

It's a storithm consisting of one state, that is: a value indicated on tape T equals V. State atom has two attributes: tape id and value. A state atom in which tape id equals T and value equals V is considered to occur at the position P, if and only if the value indicated on the tape T equals V in the P internal tour of the current internal trajectory.

This storithm doesn't have any children. The length of the occurrence of this type of storithm is always 1.

Procedure

It's a storithm consisting of a sequence of two storithms.

Children of a procedure can't be a state atom. It can be all of the rest: an action atom, condition, procedure, conditional statement or loop.

A procedure is considered to occur at the position from P (start of the occurrence) to R (end of the occurrence), if and only if the below conditions are met:

1. There is an occurrence of the first child of the procedure starting at the position P (I will refer to this occurrence as first_child_occurrence).
2. There is an occurrence of the second child of the procedure starting right after the occurrence of the first child (i.e. at the position first_child_occurrence.end + 1) and ending at the position R.

Sometimes two procedures that have two different children are really the same. For example if you have a procedure A with the following children consecutively: [1 2 3] and [4 5] and you have procedure B with the following children consecutively: [1 2] and [3 4 5], then procedure A and B represents really the same thing because both of them represent sequence of instructions [1 2 3 4

5]. For this reason, children of a procedure are represented in the program as a set of tuples of storithms (the length of a tuple is 2). Each tuple represents one group of children and every pair/tuple/group has different children but represents really the same sequence of instructions as every other pair/tuple/group from that procedure but using other children. We store children as a set of tuples so that the algorithm doesn't think of a procedure A and B as two different procedures (A and B are the same procedures but having different children) but knows that's really the same thing. If procedure B gets created when procedure A already exists, then the algorithm won't create a new storithm, instead it will merge with procedure A. Finding out if that procedure already exists is not computationally expensive, despite the fact that they have different children, because storithms are stored in a hash table and hash is generated based on the string representation of the storithm. The string representation is the same for procedures A and B, even if they have different children. Equality of two procedures is also defined based on string representation. Therefore, the algorithm can quickly see if there is an identical storithm already created, even if those two storithms are represented with different children. To be fair, it's actually not that quick because it requires calculating hash for a storithm which requires creating string representation of the storithm which can be slow if the storithm is very long. But with caching hashes at the right places, it's quick (you can find more about that in "time complexity challenges" section).

Condition

It's a storithm that consists of a set of state atoms. Children of these storithm can be only state atoms.

A condition is considered to occur at the position P, if and only if all children of this storithm occur at the position P.

The length of the occurrence of this type of storithm is always 1.

Conditional statement

It's a storithm that consists of condition and body. It has two children: condition child and body child. The condition child must be a storithm of the type "condition". The body child can be everything except state atom.

A conditional statement is considered to occur at the position from P (start of the occurrence) to R (end of the occurrence), if and only if the condition child occurs at the position P and the body child occurs at the position from P to R.

Loop

It's a storithm that consists of a condition and a body. It has two children: condition child and body child. The condition child must be a storithm of the type "condition". The body child can be everything except state atom.

A loop is considered to occur at the position from P (start of the occurrence) to R (end of the occurrence), if and only if:

1. There is an occurence of the condition child at the position P and there is an occurrence of the body child (I will refer to that occurrence with first_body_child_occurrence) starting at the position P.
2. Later, if there is an occurrence of the condition child at the position first_body_child_occurrence.end + 1 (I will refer to that occurrence with second_body_child_occurrence), then there must be another occurrence of the body child starting at this position.
3. Later, if there is an occurrence of the condition child at the position second_body_child_occurrence.end + 1, then there must be another occurrence of the body child starting at this position.
   ...
4. And you continue like that until condition child is not present.
5. And if that loop ends at the position of R, then the last body child occurrence must end at the position R.

Fusion

It's a storithm consisting of a set of two storithms.

Children of a fusion can't be a state atom or a condition. It can be all of the rest: an action atom, procedure, conditional statement or loop.

A fusion is considered to occur at the position from P (start of the occurrence) to R (end of the occurrence), if and only if one of its children occurs at the position from P to R.

Create

In create part of the algorithm, the algorithm creates new storithms and predictors.

In Slow Alien algorithm, we create (create storithm = add it to the storithm repository) all storithms that occur in the current internal trajectory. And for each storithm we have two arrays of predictors.

In Improved Alien, we have a hyperparameter CREATE_STORITHM_TYPES which defines how many storithm of each type we want to create (and in what order). More precisely, how many attempts of creating a new storithm we want to make. We make as many attempts of creating a storithm of each type as defined.

We create storithms by sampling a position in the current internal trajectory / interpretation and checking if at this position there is any storithm of the type that we want to create.

The algorithm for creating a storithm is slightly different for each type of storithm but the idea is similar.

For example the algorithm for creating a new storithm of the type "procedure" is as follows:

1. Sample position (internal tour) in the current internal trajectory (the closer it is to the current tour, the higher probability of choosing that position). Assign that position to the variable `position`.
2. Sample one storithm occurrence from the set of all the storithm occurrences from interpretation that meet the two following conditions:
   a) The type of the storithm is action atom, conditional statement, loop or procedure.
   b) They end at the position `position - 1`.
   The storithm of which occurrence we sample here is meant to be the first child of a new procedure that we're creating. If there's no storithm that meets these conditions, then the attempt of creating new procedure has failed, simply do nothing and move on with your life.
3. Sample one storithm occurrence from the set of all the storithm occurrences from interpretation that meet the two following conditions:
   a) The type of the storithm is action atom, conditional statement, loop or procedure.
   b) They start at the position `position`.
   The storithm of which occurrence we sample here is meant to be the second child of a new procedure that we're creating. If there's no storithm that meets these conditions, then the attempt of creating new procedure has failed, simply do nothing and move on with your life.
4. Return a new procedure with the following children:
   a) The storithm that the occurrence of we sampled in point 2.
   b) The storithm that the occurrence of we sampled in point 3.

Sampling storithms occurrences in point 2 and 3 might seem to be computationally expensive because we need to prepare the set from which we sample. But in fact, it's not computationally expensive because if we store storithm occurrences in interpretation in a way that is prepared for this sampling, then we don't have to prepare this set, it's already prepared. We simply segregate the storithm occurrences in interpretation by the type and position at which they starts and ends. And we do this at the moment when we add an occurrence to the interpretation. This allows us later to sample one storithm that meets the conditions with constant time complexity*.

`*` In fact, it's not constant time complexity, it's almost constant. The reason for this is how set is implemented internally - it is represented using a hash table. But we can't sample an element from a hash table, so we represent it also as a list. When we remove an element from a set, we need to remove it from the list as well. But if we do it in a normal way, then it results in a linear time complexity because we have to move every element that is after the element that we are removing. Instead, we need to put a None value in place of the removed element. Later, when we sample an element from that set (so internally we sample it from the list), then if we're unlucky we can hit the None value and then we need to sample one more time. For this reason, the worst-case time complexity of sampling is infinite because we can always hit the None value, so there is no

limit of how many operations it can take. But in practice, this won't take many operations assuming there are not too many removed elements. The sets that we're talking about here have many removed elements only when action space is big (the reason being that in "Predict" part of the algorithm, we add occurrences to the interpretation temporarily and then remove them later which results in a lot of removed elements, we do this for each possible action to take, so the more possible actions, the more removed elements we will have). In this case, we can do a clean up and remove the None values from the list (with linear time complexity, but we don't have to do that often), if we notice that there has been many removed elements in the set. Thanks to this, we can sample elements quicker because there is a smaller probability that we hit the None value.

For other types of storithms like: conditional statement, loop and condition the algorithm of creating them is similar - we sample some position in the current internal trajectory and then we sample some storithm that starts or ends at the position when they need to start or end and we sample the type of storithm that we need in order to create the type of storithm that we need. For example, for conditional statement we sample one storithm of the type condition that starts at the sampled position and one storithm of the type action atom, procedure conditional statement or loop that also starts at the sampled position. We create the new conditional statement out of those two sampled storithms.

Why is it better to create new storithms out of the storithms that has occurred in the internal trajectory and not some two random storithms? For example, when we create procedure, we sample some position in the internal trajectory and try to see if there is some procedure there by sampling one storithm that ends at this position and sampling one storithm that starts right after that. Would it be simpler to sample two random storithms of the right type from the storithm repository? It would, but it's better to create storithms from what occurred in the internal trajectory because then the probability that we create a storithm is proportional to how often this storithm occurs in the current internal trajectory which is a good thing because the more it occurs, the more we will be able to use it (the information about how it affects reward, its predictors). Also, there are probably some other reasons that I don't remember. // it improves transfer learning because by creating base on existing algorithms we give preference to the storithm that base on existing storithms.

Obviously, before creating a new storithm, we have to see if it already exists. Storithms are stored in a storithm repository which is a hash table where the key is a storithm and value is information about the storithm. If we want to see if a storithm already exists, we can simply see if there is a key with this storithm in a hash table.

What are the benefits of "create" improvement:

1. We use less computational power comparing to the Slow Alien algorithm because we don't create all occurring storithms.
2. It improves transferring knowledge capability because we create new storithms out of existing storithms and thanks to that the algorithm will faster learn the policies that base on what it has

already learned. For example, if the algorithm has already created storithms for solving the task of finding the smallest number in a sequence, then it will need less time to create storithms for sorting numbers because these storithms can contain the storithms for finding the smallest number in it.

When creating new storithms, the most computationally expensive operation is the operation of calculating the hash for a storithm. Storithm repository is a hash table in which key is storithm and we need to calculate hash for two reasons: 1. after we create a new storithm we need to see if it exists in the storithm repository (which is a hash table); 2. if it exists, then we need to add it in the right place. The storithm hash is calculated based on its string representation and time complexity of calculating hash (including constructing this string representation) is O(n), where n is the length of the string representation. If the storithm is short, then it's ok, but when it's not then it can become a bottleneck. We can improve that with caching at two places:

1. After calculating hash, we assign the result to the attribute `Storithm.hash_cache` and the next time we simply return the cached hash.
2. When we create new storithms, the algorithm will often try to create new storithms for the storithms that it has already created. For example, it will create a procedure A that consists of storithms B and C and later it is likely to happen that it will try to create that storithm many times again (each time failing because there is already a storithm like that). Each time when it happens, it will calculate hash for this storithm. We can avoid that by caching the information if the storithm already exists (or the hash) based on the children out of which it tries to create a storithm and the type of storithm. We can do that by having a hash table in which the key is a tuple of the type of a storithm and the children ids. The value is boolean if the storithm already exists (or hash if it exists).

The quick algorithm for interpreting

In the Slow Alien algorithm, the interpreting part simply finds all storithms that occur in the current internal trajectory using bruteforce algorithm. In Improved Alien we use a smarter algorithm that is described below.

The purpose of this algorithm is to find all occurrences of the storithms that are in the storithm repository in the current internal trajctory.

Problem specification

Given a set of storithms and an internal trajectory, find all occurrences of these storithms in that internal trajectory.

The storithms are represented as we described it in the "Procedures" improvement - as a sequence (or set or some other combination) of other storithms (except atoms which are storithms

that doesn't consist of any storithms).

We can also assume that if a storithm is in the set of the storithms that we need to find, then all of its children are in this set too. This is because the set of storithms that we need to find is all storithms that are in the storithm repository and if we represent a storithm using other storithms, then those storithms have to be in the storithm repository as well.

We can also assume that if a storithm is in the set of the storithms that we need to find, then all of its parents are also in this set. This is because the set of storithms that we need to find is all storithms that are in the storithm repository and if we remove a storithm from storithm repository (when we do this will be described in "importance" improvement) then we also remove the parent connection from all its children. So we don't need to worry that the storithm will have a parent that doesn't exist in the set of the storithms that we need to find (storithm repository).

Additionally, in Improved Alien algorithm we call "interpret" method in each internal tour, so we can assume that all storithms that end before the last cell of the current internal trajectory has been already found in the previous tours. So we need to find only those storithm occurrences that have their end in the last cell. If in each internal tour, we find only the storithm occurrences that end in the last cell and add them to the interpretation, then we always have all storithm occurrences in the interpretation because the current internal trajectory is extended with only one cell at the end in each internal tour.

We can also assume that we have already found all atom occurrences of the atoms that are in the set of the storithms that we need to find and that we can access them without any problem. We can assumme that because it's super easy to find atom occurrences. There are two types of atoms: action atoms and state atoms. An action atom corresponds to an action and occurs at the position at which that action was taken. A state atom corresponds to the indicated value on a tape being equal to some value. In order to find all action atoms, we can simply put the occurrence of the action atom that corresponds to the action in the last cell of the current internal trajectory. In order to find all state atoms, we can iterate through the whole internal observation in the last cell of the current internal trajectory and put the corresponding state atoms to the values indicated on the tapes.

Algorithm description

The algorithm for interpreting is as follows.

The algorithm that I will describe here assumes that we have already found all atom occurrences of the atoms that are in the set of the storithms that we need to find.

These are the steps of the algorithm:

1. Add all atom occurrences (of the atoms that are in the set of the storithms that we need to

find) to a queue. This is a queue that contains occurrences that we have already found. Later, if we find any other occurrences we will add them to this queue as well.

2. While (as long as) the queue is not empty, do the following:
    i. We take one element from the queue (and remove it). The element is a storithm occurrence.
    ii. For each of the parents of the storithm from this element (remember: storithm occurrence is a tuple of storithm, start and end), we do:
        a. See if all of the rest of the children of this parent occur at the right place for the parent to occurr. If so, then it means that the parent occurs at this place.
        b. If the parent occurs at this place, then:
            a. We add the occurrence of the parent to the interpretation.
            b. We add the occurrence of the parent to the queue.

The below code can help understanding. The below code is the definition of interpret() method of a class Interpretation in the Improved Alien algorithm implementation. The below function assumes that we have already found all atom occurrences of the atoms that are in the set of the storithms that we need to find and that they can be accessed by `self._storithm_occurrences_ending[-1].values()`. Know that `self` in the below code is an instance of class Interpretation. The responsibility of this class is to store interpretation - storithm occurrences that occur in an internal trajectory. The below function finds these occurrences based on the atoms occurrences that are already added. When it finds a storithm occurrence, then it adds the storithm occurrence using `self.add(occurrence)`.

```
def interpret(self, temporarily=False):
    if len(self) < 1:
        return None

    queue = deque()
    occurrences_at_the_end = self._storithm_occurrences_ending[-1].values()
    for occurrence in occurrences_at_the_end:
        queue.append(occurrence)

    while queue:
        occurrence = queue.popleft()
        for parent_pointer in occurrence.storithm.parent_pointers:
            parent_occurrence = parent_pointer.check_parent_occurrence(
                self,
                occurrence
            )
            if parent_occurrence:
                self.add(parent_occurrence, temporarily)
                queue.append(parent_occurrence)
```

In the above algorithm we iterate through parents of a storithm (precisely speaking, through parent pointers of a storithm). This requires that when we store storithms in the memory, we need to store

not only the children of a storithm but also its parents (and the position at which the storithm exists in the parent storithm - for example, if it's the first child or the second child). There is no problem with doing that. That indication on the parent and the position in the parent sequence of children is represented by `ParentPointer` class. The variable `parent_pointer` in the above code is an instance of this class.

The "see if all of the rest of the children of this parent occur at the right place" part (the definition of check_parent_occurrence() method that is called in the above code) is slightly different for each storithm type but based on a similar idea. Each storithm type implements method check_occurrence() which accepts interpretation, child_occurrence (the storithm occurrence that trigerred checking if the parent occurs at this place) and position (for example, if the storithm occurrence that we have found is an occurrence of the first child or the second child in the sequence of children). The check_occurrence() method returns storithm occurrence of the parent, if other children (brothers of the child_occurrence) occur in the interpretation at the right place for the parent to occurr. If they are not at the right place, then the check_occurrence() method returns None.

The result of the above Interpreting algorithm will be that we have all storithm occurrences from a given set added to the interpretation which is what the algorithm is meant to do. This fact might be confusing because we don't even get that "given set" as an input of the algorithm, so how can this algorithm find all storithm occurrences of the storithms from this set? Does the algorithm have clairvoyant abilities and can predict the input? Here's the answer to this confusion. Firstly, notice that I said before: "the algorithm that I will describe here assumes that we have already found all atom occurrences of the atoms that are in the set of the storithms that we need to find" (that finding atom occurrences part is super easy). Secondly, notice that I also said before: "we can also assume that if a storithm is in the set of the storithms that we need to find, then all of its children are in this set too". Thirdly, notice also that every non-atom storithm consists of atoms at the end because every non-atom storithm consists of other storithms and these storithms consist of other storithms and they eventually have to consist of atoms (that are storithms without any children). Fourthly, if this algorithm finds all children occurrences of a storithm occurrence, then the algorithm will also find that storithm occurrence. If you connect all of the above facts together, then you can make a conclusion that the algorithm will find all storithm occurrences from the given set. The fact that we have all atom occurrences from that set in the queue when we start is enough for finding all storithm occurrences from the given set because if there is a storithm that occurs in the current internal trajectory, then at least one atom that is a descendant of that storithm must occur in the current internal trajectory as well (because for every type of storithm, occurrence of that type of storithm is defined such that it occurs only if at least one of its children occurs). We now know, that the algorithm will find all storithm occurrences of the storithms that are in the given set, but can the algorithm find also some additional occurrences of storithms that are not in the given set, if it starts from atoms? No, it can't. This can be implied from the following statement that I said before: "we can also assume that if a storithm is in the set of the storithms that we need to find, then all of its parents are also in this set".

I said before that all storithms from the storithm repository can be represented as a directed graph. If you imagine the set of storithms from storithm repository as a graph, then you can notice similary of this algorithm of interpreting to the BFS algorithm for traversing graph data structures. It's not identical, but it's similar. Noticing that similarity can help to understand why this algorithm works.

The time complexity of this algorithm is O(n + m), where n is the number of all occurrences that we find and m is the sum of the number of all parents of the storithms of the occurrences for each storithm occurrence that we find. This is assuming that calling check_parent_occurrence method is O(1). This assumption is wrong, but it's close to being true.

Importance

In the algorithm, we store some storithms in the storithm repository and we store some predictors for these storithms. "Importance" improvement is basically about the realisation that not every storithm is equally important (how much an existence of a storithm in storithm repository helps us to achieve what we want is not equal for every storithm). In fact, the vast majority of the storithms is totally useless. If we don't store some unimportant storithms in the memory, then we need significantly less memory and we need signitificantly less computational power for finding these storithms in Interpreting part of the algorithm. The same is true for predictors. // and also there is less mess with predictors, so it's easier to do Gradient Descent, if there is less predictors, then Gradient Descent will quickly find that predictors that actually affect the return

How do we know which storithm or predictor is important? And what does "important" mean? The goal of a reinforcement learning algorithm is to maximize the return. Therefore, the importance of a storithm or predictor should be equal to the difference of the expected return that we would get if we keep the storithm/predictor and the expected return that we would get if we remove the storithm/predictor. Knowing exactly that difference would be complicated and probably computionally expensive. Therefore we don't aim for having a very precise estimation of the importance. We aim for having something that is simple and quick to calculate but still works to a sufficient degree. I will describe how the importance of a predictor/storithm is calculated in the following subsections.

The algorithm has hyperparameters MAX_STORITHMS_COUNT and MAX_PREDICTORS_COUNT which represent the highest number of predictors and storithms that we can have. If the number of storithms or predictors is exceeded, then we remove the least important storithm/predictor. How do we handle knowing which storithm/predictor is the least important without adding a lot of computational overhead (especially taking into account that storithms/predictors change their importance)? We use a data structure called limited set (the term "limited set" has been introduced for the purpose of this algorithm). Limited set is a set with a limit of elements. If we want to add a new element, but the limit of elements is exceeded, then we remove the lowest element from a set (or don't add a new element at all, if the new element is the lowest). Internally, limited set is represented using heap (and a hash table for storing the position

in the heap for each element). Heap is basically a data structure that is very good at knowing what the lowest element in the data structure is. We have two limited sets in the algorithm - the first limited set stores all storithms (or references to them) and the second one stores all predictors (or references to them). In these limited sets, we judge if a storithm/predictor is higher or lower than other storithm/predictor based on their importance. When we create a new storithm/predictor, we simply add this to the limited set and if the limit is exceeded, then `LimitedSet.add` method will return the storithm/predictor that we need to remove (the least important one). If a storithm/predictor is modified in a way that changes its importance, then we call `LimitedSet.update()` method. Thanks to using heap and a hash table internally, limited set has good enough time complexity for all operations that we need to use it for - adding a new element (with removing the lowest element), updating an element (and the importance of it), removing the lowest element, seeing if an element exists in the set.

Predictor importance

How is importance of a predictor defined? Defining importance of a predictor is responsibility of the estimator (i.e. if we use one estimator, then it can be defined in a different way than when we use an other estimator). If we use Gradient Descent, then importance of a predictor is equal to the absolute value of its coefficient multiplied by the frequency of occurrence of the predictor. Why? As I said before, the importance of the predictor generally should be close to how much we loose if we remove the predictor. Absolute value of the coefficient is stronly related to how much we loose if we remove the predictor because predictors with high absolute of coefficient affect our expectation of the return (when we calculate on-policy action-values) higher than the predictors with low absolute of coefficient. Frequency of occurrence of the predictor is important too because if a predictor occurr often, then it's important for calculating the action-values, if it occurs rarely then we don't have that much use of that predictor. Therefore, if we want to calculate how important the predictor is and how much we loose if we remove it, then we multiply the coefficient with the frequency of the occurrence of the predictor.

There is a problem with storing frequency of the occurrence of the predictor because if we want to store it then we need to update this value for all predictors in every tour because this value changes in every tour. This produces unacceptable time complexity (O(n) where n is the number of all predictors in the program at all). The solution to this problem is that instead of storing frequency of the occurrence of the predictor, you store the number of how many times the predictor has occurred in the past. The frequency of the occurrence of the predictor is the number of how many times it has occurred in the past divided by the number of all past tours. Therefore, if we calculate the importance of a predictor based on the number of the occurrences instead of frequency, then if we want to get actual importance then we need to divide it by the current tour (number of all past tours). In order to know the number of all occurrences of a predictor, we simply increment that number when a predictor occurrs. This produces much better time complexity (O(n), but n is now the number of all predictors that has occurred in the tour, not all predictors in the program at all which makes a big difference). But the problem with that is that at the beginning,

when we create a new predictor, then we don't know how many times it has occurred in the past. The solution is that when we create a new predictor, we put into its number of occurrences an expected value of how many times it has occurred before. We calculate that expected value simply by tracking the average frequency of all predictors (more or less) and we multiply it by the number of all tours - this value will more or less of how many times the predictor has occurred (not exactly, but it doesn't need to be exact).

Storithm Importance

How should the importance of a storithm be measured?

In order to answer this question, we should think of what we use storithms for. We use storithms for the following purposes:

1. To estimate on-policy action-values based on the predictors of the storithm (in "predict" process).
2. To find other storithms that the storithm is a child of (in "interpret" process).
3. To create other storithms that the storithm will be the child of (in "create" process).

This gives us three kinds of storithms importance that correspond to each of the above purpose:

1. Storithm predictor importance.
2. Storithm parent importance.
3. Storithm potential importance.

The overall storithm importance is the sum of these three importances.

Storithm predictor importance is the sum of importances of all predictors that the storithm contains.

Storithm parent importance is the sum of all overall importances of all storithms that the given storithm is child of. Updating this importance is a bit complicated to do with acceptable time complexity, but it can be done (I will not describe how because I don't want to go into that level of detail).

Storithm potential importance is equal to ((predictor importance + parent importance) * POTENTIAL_IMPORTANCE_MULTIPLIER / age of the predictor). Age of the predictor is how many tours the predictor exists. If it exists 0 tours, then we put some expected value at the beginning instead of predictor importance and parent importance. We assume that if in the past there was tendency that the predictor and parent importance of the storithm was growing quickly, then there are higher chances that it will grow in the future. // needs correction

Small Improvements

There are two small improvements.

The first small improvement is that we reward the agent for taking an external action quickly. If we don't do this then the agent should take a lot of internal actions before taking any external action. Why? Because why not - it's rewarded for taking right external actions, not for taking them quickly, so it's in the interest of the agent to think long before taking any action. If that becomes a problem, then we can easily resolve it by rewarding the agent for taking external action quickly.

The second small improvement is the answer to the problem that if we have a storithm and we have arrays with its predictors, then if the MDP that we run the algorithm on requires long-term thinking, then that array needs to be very long and can require a lot of memory. This is not needed because the coefficient of `storithm_repository[S].external[64]` will be for example very similar to the coefficient of `storithm_repository[S].external[65]`. This is because the first one represents the predictor that is used to estimate the return after 64 external tours after the occurence of the storithm S and the second one is used to estimate the return after 65 external tours after the occurrence of the storithm S. Let's suppose that the horizon is 200. Both predictors are trained on the the same training set. So the coefficient for estimating the return after 64 external tours should be very similar to the coefficient for estimating the return after 65 external tours. Therefore, we can save memory if instead storing the predictor for each tour, we store predictors that are used for estimating the return both after 64 external tours and 65 external tours. Therefore, instead having predictors `storithm_repository[S].external[64]` and `storithm_repository[S].external[65]`, we have for example `storithm_repository[S].external[60-65]` and that predictor occurrs at each position from 60 tours after the occurrence of the storithm S to 65 tours after the occurrence of the storithm S.

Merging

Merging works in a bit different way than in the Slow Alien algorithm. Likewise in Slow Alien, we create new storithms of the type "Fusion" by sampling two storithms and the new "Fusion" storithm has these two storithms as children. But there are some changes. These changes are complicated and I will not explain them for now.

// But the problem in Improved Alien is that we introduce "Importance" improvement which // predictors are 0 so importance will be zero, so the storithm will not be added - solution: borrow coefficient from other predictors, we merge only conditional statements and loops that can't occurr at the same time (have opposite condition, so that we can properly borrow coefficient).

Importance based sampling

This improvement is a modification to the algorithm of interpreting that was described previously.

This improvement is complicated so I will only summarize it.

The "importance based sampling" improvement is basically as follows. In the algorithm of interpreting, we add all occurrences that we find to a queue. For each occurrence that we find, we iterate through parents of the storithm of that occurrence and we have a look if there is an occurrence of the parent at this place. Instead of iterating through all parents of the storithm at that stage of the interpreting algorithm (as described previously), sample one parent with probabilities equal to what the expected usefulness of finding each parent is. Repeat that many times. After this improvement, interpreting procedure won't find all storithms that we have in the storithm repository but will be more probable to find the important storithms. This will allow the algorithm to store more storithms in the storithm repository because after that improvement, we don't have to find all storithms that we have in the storithm repository (before that improvement the number of storithms that we can store in the storithm repository was limited by the computational expensiveness of finding all occurrences of the storithms, after that improvement how much computational power is required to find all occurrences is not limited by how many storithms we have in the storithm repository, so we can store more storithms). This will allow more diversity when creating storithms. Before that improvement, the algorithm will often try to create the same storithms. After that improvement the diversity of the storithms that it will try to create will be greater. Thanks to that, it will create more important storithms with less attempts.

Getting more out of one sample

The problem with this algorithm is that it doesn't take a lot of learning for one sample comparing to how much it's possible to take. In order to explain what the problem is, I will give an analogy to eating a chicken. Imagine you eat a chicken that is divided to pieces. You take the first piece of chicken, you take one bite and then throw it away and take next piece of chicken, take one bite and then throw it away, then take next piece of chicken, take one bite and then throw it away and after some time you realize there is no more pieces of chicken and you're still hungry because you haven't eaten as much as you could, you have thrown away all pieces after taking only one bite. Analogically, this algorithm takes one sample (one tour), takes one bite of learning and throws the sample away, then takes another sample, takes only one bite of learning and at some point you realize that it hasn't learned too much but there are no more samples to learn from. This is not a problem if the rewards are given programatically (because then you can generate many samples/tours), but it is a problem when rewards are given by human because in this case human has to give a lot of rewards and it requires a lot of human supervision. It is also a problem when the algorithm is used for supervised learning and there is a small amount of samples (but in this case the problem can be easily solved by simply repeating the training many times on the same training samples).

Why doesn't it learn as much as it could? Why is there potential for more? Let's suppose that the agent took internal actions 2, 1 and 4 and it led to taking the external action 8. The reward that the agent received after that action was 5. If the agent took internal actions: 5, 2, 4, 1 and 3 and after

that it would take external action 8, then it would get 5 reward as well because only external actions affect how many reward the agent gets, not internal actions. Because internal actions are not visible in the external world. And the external action (8) was the same in both cases. But if the agent takes actions: 2, 1, 4 and then 8, it only learns that taking actions 2, 1, 4 and 8 leads to getting 5 reward. The learning that taking action 5, 2, 4, 1, 3 and then 8 also leads to getting 5 reward is that additional meat that the agent doesn't eat and throws away. That's what I meant when I gave the analogy to eating the chicken.

How can we make the agent get more learning out of one sample? We can generate more sequences of internal actions that lead to the same external action, interpret that and learn from that. But in order to learn from that, we need to move back to the state of the observation and memory that was before, then execute the actions and then you can learn from that. This should be possible to do without making computational expensiveness of the algorithm too big, but it's very complicated to implement. But how to generate more sequences of internal actions that lead to the same external action? It's easy - you can generate random sequence of internal actions and add the taken external action at the end. But this is not ideal because the sequences of internal actions that the agent would be likely to take are more valuable than random ones. And random sequences are not very valuable comparing to the ones that the agent would be likely to take. Therefore, how to generate a sequence of actions that the agent would be likely to take? One way is to simply move back to the state of the observation and memory that was before and use Alien to act again until it generates some external action. But we need to generate a sequence of actions that ends with that one external action that we can learn from, so we need to throw away every sequence of actions that doesn't end at that external action. That is not a problem when the action space is small, for example if we have only 5 possible external actions because then we throw away only 4 out of 5 sequences. But this is a problem when the action space is big, for example if we have 10000 possible external actions then we throw away 9999 out of 10000 sequences. In this case, a better way to generate that sequence of internal actions would be to use a recurrent neural network that would generate a sequence of internal actions that the agent would be likely to take but in the reverse order, starting from the external action that we need to have at the end.

That improvement needs investigation yet and the above described mechanism is only a proposition. Also, this improvement is crucial because it enables the agent to learn by demonstration/impersonation. For example, let's say that you have a task where you have some question and have 100000 possible answers. It will take a lot of time for the agent to learn that because in order to learn that, it has to accidently give the right answers many times so that it can find out what the optimal policy for generating the answer is. That time can be greatly reduced using impersonation - you impersonate the agent and give the right answers, you choose the actions instead of the agent but the agent still learns from this experience. This way you can reduce the time of the agent accidently giving many right answers at the beginning. This impersonation is not possible in Alien algorithm on default. It's only possible after that improvement.

Examples

In this section I will give an example how Alien algorithm can be used to conquer some challenges (games, environments, tasks).

Frozenlake

Frozenlake is an environment from OpenAI gym. You can find the description of it here: https://gym.openai.com/envs/FrozenLake-v0/ . It's a game in which the goal is to move from one point to another, but you can move into hole and then you start from the beginning. The observation of the agent is one number which represents the position where the agent is. The agent can execute for actions: move left, move right, move up, move down.

I have applied the Alien algorithm to this game and the algorithm is clearly able to become good at this simple game.

Here are the results of the Alien agent after being trained with 10000 tours (each time it was trained seperately): 30, 34, 7, 34, 1, 40, 76, 3, 3, 44, 83, 38, 3, 29, 83. Average: 33.9.

Here are the result of the random agent (taking random action at each tour): 4, 2, 3, 1, 1, 4, 0, 4, 2, 2, 4, 1, 4, 1, 0. Average: 2.2.

This is for FrozenLake with is_slippery=False. The implementation of the algorithm that it was tested on was incomplete, therefore it's not important how quickly this algorithm learned anything because this will be greatly improved. What's important is that it learns at all.

Since this game is very simple and quite easy, you don't need to use internal state for the agent to become good at this. Also, the only storithm types that the algorithm needs to use are as follows: state atom, action atom, condition, conditional statement. The algorithm doesn't need to use procedure and loop for this task.

How can Alien algorithm learn to be good at this game? Looking inside the mind of the Alien after being trained to be good at this game, we can see that the predictors with highest coefficient were the predictors of the following storithms:

1. The storithm consisting of one action for moving down - therefore Alien learned that it's good to go down in general (the same situation was for going right).
2. The storithms like for example: "if observation = 15 then move right".

Copy

Copy is another environment from OpenAI gym. You can find the description of it here: https://gym.openai.com/envs/Copy-v0/ . The goal is to copy the text from a tape. The observation of the agent is the letter that is currently indicated on the tape. This environment seems to be easy, but it's a bit more difficult than it seems because it's about learning to manage the tape.

How can Alien become good at this task? The optimal policy for this task is for example like that:

Ensure that done = 0 ('done' can be given as part of the observation). Ensure means "if"
If you observe empty value, then move right.
Then if you observe empty value, then move right.
Then if you observe the letter "A", then write it to the output tape and move right.
Then if

Chess

Sorting numbers