

**جامعة العلوم والتكنولوجيا الاردنية**

**ورقة بحثية - مشروع تخرج 2**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **البرنامج**  **تنافس** | **البريد الإلكتروني**  **mmaltamimi20@cit.just.edu.jo** | | **الرقم الجامعي**  **144754** | **اسم الطالب**  **محمد مهند عبدالحفيظ التميمي** |
| **التاريخ**  **19-6-2024** | **الكليـة**  **تكنولوجيا المعلومات** | | **القسم**  **علم الحاسوب** | **التخصص**  **علم الحاسوب** |
| **توقيع المشرف الرئيس** | | **البريد الالكتروني للمشرف الرئيس**  **qmbanibaker@just.edu.jo** | | **اسم المشرف الرئيس**  **د/قانتة بني بكر** |
|  | | **البريد الالكتروني للمشرف المشارك** | | **اسم المشرف المشارك** |
| **I. Title of the Proposed Project**  Disease diagnosis in task-oriented dialogue systems using Deep Reinforcement Learning and Auto Encoder | | | | |
| **II. Introduction and Hypothesis of the Study**  In the context of task-oriented dialogue systems using DRL, there are many challenges that need to be addressed in order to perform well such as: reward sparsity and huge action space and hyperparameter sensitivity as well as categorical data, in this research we are doing extensive experiments to overcome these challenges in order to increase the performance. Many experiments have been done but the most significant result was the Auto Encoder. | | | | |
| **III. Aims of the study:**  we implemented a robust and insensitive to hyperparameter model by using Auto Encoder, while keeping the extinsive, prohibitive time consumbtion to lowest, by using the latest OpenAI packages. | | | | |

**Index**

1. Introduction and Background
   1. Reinforcement learning components.
   2. Why reinforcement learning?
   3. Algorithms and Background
      1. Reinforcement learning algorithms background :
      2. Proximal Policy Optimization Agent
      3. User Simulator
      4. Dialogue Manager
      5. State Tracker
      6. Disease Classifier and Dual Actor
      7. Embeddings
      8. GAN
      9. Expert Trajectories
   4. Reinforcement learning and dialogue systems challenges
      1. Huge Action Space
      2. Reward Sparsity
      3. Categorical Data
      4. Hyperparameter sensitivity
      5. Action and state representation
      6. Exploration vs exploitation dilemma
   5. Why Reinforcement learning?
   6. Data Information
2. Related Work
   1. Hierarchical Reinforcement Learning (HRL).
   2. Adversarial networks.
3. Methodology
   1. Our Framework
   2. Experiments and Results
   3. Proposed implementation (Auto Encoder)
4. Discussion and Results
5. References
6. Appendix

**1-Introduction**

The progression from traditional to automated systems in disease diagnosis represents a critical evolution in medical technology. Traditionally, as Ledley and Lusted outlined in 1959, diagnosis depended on a threefold approach: extensive medical knowledge, the patient's reported symptoms, and the clinician's expertise culminating in a diagnosis. Typically, this process starts with the patient describing their symptoms, followed by the doctor's inquiries for additional symptom details, eventually leading to a diagnosis through a series of questions and answers.

This process of disease diagnosis is often a time-intensive task for doctors. It involves a detailed exploration of a patient's symptoms to pinpoint the positive indicators of specific conditions. Automating this procedure is not only essential to save time but also requires meticulous execution to ensure high performance and a positive user experience. The goal is to achieve accurate diagnoses without burdening the patient with excessive questioning about their symptoms. This balance is crucial in developing an efficient and patient-friendly diagnostic system. Recently, Reinforcement Learning-(RL) became popular in achieving this goal.

Task-oriented dialogue systems are specialized conversational agents designed to assist users in completing specific tasks. Unlike open-domain dialogue systems, which can handle a wide range of topics and conversations, task-oriented systems are focused on achieving particular goals, such as booking a flight, making a restaurant reservation, or similar to our field of context, diagnosing medical conditions. The primary purpose of task-oriented dialogue systems is to facilitate efficient and effective communication between users and the system to accomplish predefined tasks. [33]

In Rule-based systems they use predefined rules and logic to manage the dialogue. These rules dictate how the system should respond to specific user inputs and situations. However, in reinforcement learning, agent learns, and updates strategies based on interactions. This dynamic approach allows for greater flexibility and adaptability. [34]

(RL) is a type of machine learning that allows machines to improve their performance depending on prior experience. RL is based on an 'agent' that learns to make decisions by interacting with the environment. The agent, performs different 'actions' in response to the 'state' of the environment.

Unlike other machine learning systems, which use explicit instructions to assist the learning process, RL uses a trial-and-error methodology. The agent gains knowledge from the results of its activities, receiving 'rewards' or 'penalties' based on the success of its judgments.

In order to maximize rewards, the agent eventually learns to increase the frequency of profitable behaviors.

A medical informational chart

Description automatically generated with medium confidenceThis approach is similar to how people learn from their experiences, which makes it an effective tool for situations where decision-making processes must change and adapt on the fly, like in conversation systems for medical diagnosis.

Figure 1: A photo shows how the process of disease diagnosis works.

Our contribution to the field of disease diagnosis in dialogue systems by tackling issues and challenges we implemented a robust and insensitive to hyperparameter model by using Auto Encoder, while keeping the extinsive, prohibitive time consumbtion to lowest, by using the latest OpenAI packages.

**Reinforcement learning component:**

**Actions:** are the specific responses or decisions the agent makes. In our case, it is asking a patient about specific symptoms or diagnosing a disease.

**States:** represents the current situation or status within the environment. In this context, a state is the current set of symptoms that is known to agent.

**Rewards**: are feedback given to the agent based on the effectiveness of its actions, a reward system is critical to the efficiency of the RL agent. Rewards can be positive or negative i.e. ‘penalty’.

**Policy:** The policy is the strategy or set of rules that the agent follows to decide its actions in each state. It's essentially the decision-making guideline for the agent. The goal of the agent is to learn an optimal policy (π) which maps state to most appropriate action, i.e., a\* = *argmaxi* π\* (ai |si) [35]

**Algorithms and Background**

1. **Reinforcement learning algorithms background :**

On-policy algorithms, such as REINFORCE and Proximal Policy Optimization (PPO) which we will be implementing later, learn a value function or policy based on the current policy's experience. This means they need constant access to fresh data collected from the environment using the current policy. In other words, the policy being evaluated and improved is the same one used to make decisions during interaction.

In contrast, off-policy algorithms like Deep Q-learning (DQN) learn from data collected from a different policy, which could be an older version of the same policy or an entirely different one. For example, DQN uses two policy networks: a target network for learning and a behavior policy for exploration.

Another important distinction in RL is between value-based and policy-based algorithms. Value-based algorithms aim to find or approximate the optimal value function, which maps actions to values indicating the expected return of taking a particular action in a given state. In policy-based algorithms, the focus is on directly learning the policy, which maps states to probabilities of selecting each action, with the goal of adjusting these probabilities to maximize rewards.

Table 1: Different algorithms categorization

|  |  |  |
| --- | --- | --- |
| Learning strategy  Method | On-policy | Off-policy |
| Value-based | * SARSA (State-Action-Reward-State-Action) | * Q-Learning * Deep Q-Networks (DQN) |
| Policy-based | * REINFORCE * Proximal Policy Optimization (PPO) |  |

**Steps for Policy-based and Value-based Algorithms**

|  |  |  |
| --- | --- | --- |
| **Steps** | **Value-based** | **Policy-based** |
| 1 | Initialize value function (action or state function) randomly. | Initialize the policy parameters θ randomly, a probability distribution over action parameterized by θ. |
| 2 | Derive a policy π from the current value function. | Generate episodes by following the current policy π\_θ. Each episode consists of states, actions taken, and rewards received. |
| 3 | Interact with the environment by following an exploratory policy to collect experience (in experience replay). | For each episode, compute the gradient of the expected return with respect to the policy parameters. |
| 4 | Update the value function based on the collected experiences. | Update the policy parameters in the direction of the computed gradient, scaled by a learning rate α. |
| 5 | Implicitly improve the policy by improving the accuracy of the value function estimates. | Repeat these steps until the policy converges. |
| 6 | Repeat this process until the value function converges. |  |

1. **Proximal Policy Optimization Agent:**

The goal of the dialogue agent is to have a policy , that maximizes the expected discounted reward over time. Given sum of discounted return, then is the goal. [24]

To achieve this in value based we need to maximize these 2 functions, state value function and action value function, the state value function is the expected reward given a state and following a policy , and denotes as: , while action value function is expected reward given a state and an action and then following a policy . This approach is within the broader category of policy gradient methods in reinforcement learning, and it is closely connected to the actor-critic framework. PPO incorporates a clipped surrogate objective function, which restricts the magnitude of policy updates to ensure stability and prevent destructive policy changes. The loss for the actor’s network is calculated as the following: [24]

Where:

* θ represents the parameters of the policy.
* Et​ denotes the expectation over time steps.
* Rt​(θ) is the probability ratio between the new and old policies, given by:
* π​(at​∣st​) is the new policy.
* π​old(at​∣st​) ​is the old policy.
* At​ is the advantage estimate at time step t. The advantage function is the difference between the actual rewards and the estimated value function.
* ϵ is a small hyperparameter that defines the clipping range.

This loss function ensures that the policy updates are constrained within a trust region, preventing large, potentially destabilizing updates.

Whereas the Value network’s loss is computed as the following:

It is simply the Mean Square Error, between the value network and the actual reward obtained multiplied by a discount factor.

The overall objective of the value network loss function is to reduce the discrepancy between the predicted value and the actual return, thereby improving the accuracy of value estimates used for policy updates.[24]

The PPO agent is designed to optimize its policy by storing and utilizing trajectories effectively. The agent interacts with the environment, collecting trajectories in the form of state-action-reward tuples. These tuples are stored in the agent’s memory. After accumulating trajectories over a predefined number of steps (N\_steps), the agent initiates the learning phase. During this phase, the agent samples minibatches from its memory and computes the loss for each minibatch. This loss is then used to perform backpropagation, adjusting the network's gradients to improve the policy. The learning process runs for a specified number of epochs (n\_epochs), ensuring the policy is refined iteratively based on the collected experiences.

The PPO has 2 different implementations either with a shared network layer, and the loss of both actor’s and critic’s is backpropagated to it, or it combines both losses together to edit the actor’s loss.

1. **User Simulator**

The agent requires an interactive context, typically an 'environment', from which it derives its information;[39] this is where the role of the user simulator becomes crucial to train the agent, it-chooses a sample user goal from the dataset, each sample consists of 4 parts, a) *disease\_tag* b) *explicit\_symptoms* c) *implicit\_symptoms* d) *request\_slot* (See Figure 7)[40][22]. Initially the user simulator informs the agent about the explicit symptoms, that’s because it is considered a (self-

A screenshot of a computer

Description automatically generatedreported symptoms), as well as requests a disease.

Figure 2: Example of a user goal.

The user randomly chooses a user goal from dataset, makes a state vector that includes all the explicit symptoms to inform, since the explicit symptoms are the symptoms that the patient has given the doctor at the beginning of the dialogue and are said to be self-reported symptoms. The user holds a symptom to index dictionary, to handle any further symptom inquiry. The state has an initial state value, it then updates the symptoms that are classified as explicit then adds them to the state vector. When an agent requests a new symptom, the user should change the state of the new symptom according to the symptoms that are classified as implicit, and then assign a value based on the symptom presence. If the symptom is not classified as implicit, then a value not mentioned is assigned. [40]

The state the user returns depends on the methods we use for encoding the response and whether a state tracker is used or not, such differences will be discussed later in the report.

1. **Dialogue Manager**

The manager class wraps every component of the system together, instantiates objects from each class, assign the hyperparameters values, contains the main method for starting the conversation between agent and user, aka, rollout.

1. **State Tracker**

The state tracker importance lies in the concept that the agent does not know what he asked at last turn at what the user responded that states seemed unrelated and individualistic.

Traditionally, dialogue states were encoded as static label-encoded vectors that did not account for past interactions, leading to a lack of continuity and context in the dialogue, with state tracker the last turns are added to the state so the agent can give actions relative to the past actions.

So, a state tracker role is to keep track of the current states of the dialogue, and the new state representation for the agent is the state of the last 2 turns of the dialogue and following the state tracker used by [41][42]. The state tracker featured three primary methods: reset(), update(), and get\_state(). The update method modified the state with each interaction, while get\_state provided the current state to the agent for decision-making. This setup ensured continuous state updates reflecting the dialogue's progression. The code can be seen in the appendix.

1. **Disease Classifier and Dual Actor**

**Dual-Network Approach (disease classifier)**

To further refine action selection, we implemented new architectures to divide the action space and reduce dimensionality. Thus, tackle the issue of *huge action space*. The first architecture introduced a secondary network with the same input from a shared layer, outputting the disease list. This network's loss function, based on Mean Squared Error (MSE) from the final disease tag, influenced the shared layer, with actions chosen based on the higher probability between the actor and disease classifier.

**Dual-Actor Model**

To align the disease classifier with reinforcement learning principles, we modified its loss function to mirror the actor's clipped surrogate loss, effectively creating two actors: one for requests and one for informs. Actions were chosen based on the confidence threshold of the disease classifier.

1. **Embeddings**

One clear disadvantage of one-hot encoding is that the distance between one-hot encoded vectors carries minimal information [36]. To tackle this issue of categorical data, embeddings are used in task-oriented dialogue systems, especially in medical diagnosis, as vector representations of the current state. Let S be the set of distinct values of a variable characterized as categorical data. An entity embedding e is a mapping e: S→Rd that transforms elements of S into vectors of real numbers. These vectors enable the system to process and understand these elements efficiently [36]. The primary purpose of embeddings in this context is to handle various discrete elements (like symptoms) in a continuous vector space, facilitating more effective learning and decision-making by the dialogue management model. [36].

1. **Auto Encoders**

Other way to tackle the issue of categorical data, autoencoders are neural networks designed to learn efficient data representations by compressing input data into a latent-space and then reconstructing it. They consist of an encoder, which compresses the data, and a decoder, which reconstructs the original input from the latent representation. [36]

Encoder: Compresses high-dimensional input data into a lower-dimensional latent-space representation, capturing essential features.

Latent-Space Representation: A compact representation containing the most relevant information of the input data.

Decoder: Reconstructs the input data from the latent representation.

1. **GAN**

It is used to tackle the issue of reward sparsity. Adversarial networks has 2 components, a generator and a discriminator. The former role is to generate state-action pairs (or sequence of symptoms) for the latter to differentiate from the real ones. In this process the two modules improve each other in an adversarial way.[44] Over time, the discriminator will have the capability to distinguish real sequence of symptoms from fake ones. To be used as a reward function to train the dialogue agent, along with other reward, such as penalties for asking user simulator more than times. This reward model is then incorporated into a PPO or REINFORCE, simultaneously or similar to [15] into DQN algorithm to train the dialogue agent sequentially.

1. **Expert Trajectories**

Imitation learning or Behavior cloning, it is a concept in RL is used to speed up training speed, by making some trajectories as expert (optimal tuples of state-action-reward) and make the agent learn on them, this method has been used to tackle the issue of time-extensive training of PPO agent.

**Reinforcement learning and dialogue systems challenges:**

1. **Huge Action Space**

managing a high action space becomes a critical challenge. The complexity and variability of medical conditions require the system to handle a vast array of possible actions. This complexity stems from the need to inquire about numerous symptoms and make accurate diagnostic decisions. Assume the action space A is {1, . . . , m} ∪ {m + 1, . . . , m +n}. We say that an action is a feature acquisition action(choosing a symptom) if it is less than or equal to m; otherwise, it is a classification action, i.e. the agent choose from disease set. [14][5][10]

1. **Reward Sparsity**

the reward sparsity is a huge problem in the context of disease diagnosis, since the number of valid symptoms for agent to request or informing a disease, are small considering the huge action space of symptoms and disease, the feedback is infrequent and inadequate as it is only given to the agent at the end of the dialogue, as we will discuss later we tried to solve this challenge with different approaches. Assume in some medical dataset D, X = {0,1}m where X is the set of all symptoms, m is the number of symptom, value 1 for present symptom and 0 for absent symptom, x+ = {j : xj = 1} and x- = {j : xj = 0}, then we say D is sparse if |x+| << m for X in D. Hence, identifying pivotal features amidst a sea of patient data is challenging, This sparsity of 'useful' features leads to a demanding and time-consuming learning process for the RL agent. [14][7]

1. **Categorical Data**

Categorical data often represented through one-hot encoding, results in high-dimensional and sparse vectors. This sparsity leads to less informative gradients during optimization, making it challenging for algorithms to learn effective representations. Thus, exacerbating the curse of dimensionality. This issue makes it harder for algorithms to explore the feature space efficiently and learn meaningful patterns.[36]

1. **Hyperparameter sensitivity**

The PPO algorithm's performance is highly sensitive to its hyperparameters. Even slight changes can lead to significant variations in the model's performance. This sensitivity is particularly pronounced in our context, which is fraught with local minima due to the previously mentioned challenges. The fine-tuning of hyperparameters such as the policy learning rate, the value function coefficient, and the entropy coefficient is essential to ensure that the model does not get stuck in local minima. Proper tuning helps the agent navigate the complex action space more effectively, avoiding premature convergence and improving overall performance.[24][43]

**Clipping Parameter (ε) Sensitivity:**

The clipping parameter in PPO is designed to prevent large updates that could destabilize training. If this parameter is set too tightly, it restricts the policy updates too much, leading to slow progress and potentially trapping the agent in local optima. If set too loosely, it allows for large, destabilizing updates that can cause the training process to become unstable and diverge. Due to the categorical data challenges said in previous section, the model was rarely taking the loss from his own and not from the clipping range, this showed that the were indeed very crucial to adjust.[24][43]

**Entropy Coefficient Sensitivity:**

The entropy coefficient encourages exploration by preventing the policy from becoming too deterministic. A high entropy coefficient fosters exploration, which is essential in the early stages of training to discover diverse strategies. However, if set too high, it can hinder convergence by continuously pushing the policy to explore, even when a near-optimal strategy has been found. Conversely, a low entropy coefficient may lead to premature convergence to suboptimal policies due to insufficient exploration.[24][43]

1. **Action and state representation**

In reinforcement learning, the choice of action space representation significantly impacts the learning stability and performance of algorithms. Proximal Policy Optimization (PPO) benefits from continuous action spaces as they allow for smooth policy updates, reducing abrupt changes that can destabilize learning [24]

1. **Exploration vs exploitation dilemma**

These are not the only challenges that RL faces, the exploration vs. exploitation dilemma is important to shed the light on, balancing the exploration of uncharted state-action pairs against the exploitation of known, rewarding paths is crucial for efficient learning and accurate diagnosis. Some known exploration algorithms are ϵ-greedy and VIME.[39]

**Why reinforcement learning?**

Unlike statistical methods that rely on predefined models, or LLMs that generate output based on vast pre-trained *language* datasets, RL has one significant advantage, its ability to continuously learn and adapt, where dataset might change or new diseases might show up, RL can continuously learn contrasts with many machine learning models that may require retraining or adjustment when the underlying data distribution changes.

Furthermore, RL can maximize long-term goals through sequential decisions. While supervised learning focuses on immediate accuracy or error minimization, RL takes into account the cumulative effect of decisions, including learning algorithms that may forego short-term advantages in exchange for larger long-term rewards. As well as supervised learning only consider the explicit symptoms.

|  |  |
| --- | --- |
| Experiment | results |
| XGBoost knowing only the explicit symptoms | ~63% |
| XGBoost assuming he knows both explicit and implicit symptoms. | ~77% |

The experiment shows that the model needs to have full knowledge before getting good accuracies.

Lastly, the trial-and-error learning approach by RL makes it particularly suitable for tasks that require a degree of cognitive-like reasoning or decision-making.

**Data information**

The dataset collected is from both [23] where they made the MZ dataset. As well as [3] where they made a new dataset called DX, which is more challenging retaining original self-reports and conversational data between patients and doctors. Another source of data is SymCAT, which is a website that generates synthetic dataset. As well as a medical dialogue dataset (MDD) similar that has been used in [9] Other paper uses dataset that are not related to the context of medical diagnosis. Such as movie booking dataset (DSTC2). And datasets related to multi-domain context such as MultiWOZ[31].

The specification of the dataset MZ and DX used in our research are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | # of user goals | # of diseases | Avg. # of imp.[[1]](#footnote-1) symptoms | # of symptoms |
| MZ-4 | 1733 | 4 | 5.46 | 230 |
| DXy | 527 | 5 | 1.67 | 41 |
| SymCAT | 30,000 | 90 | 2.60 | 266 |

**2-Related Work**

Recent advancements in task-completion dialogue systems have seen a variety of approaches, each addressing specific challenges in the field. In [2] shows that Bayesian inference and decision trees have been proposed. [25] Used a Bayesian based symptom-screening algorithm getting an overall accuracy of ~92%. However, they assumed that the symptom status is only negative or positive and the system extracts negative and positive symptoms from self-reports, ignoring “not-mentioned” status.

The use of Deep RL as an optimization has been used by [1] with a hybrid manner using supervised learning. In [6] they trained an RNN with supervised and reinforcement learning for dialogue control. Prior work showed that a pre-trained supervised policy or a weak rule-based policy can significantly improve the efficiency of exploration [4, 11].

In [3] a hybrid model is used which merges RL with graph reasoning to guide policy learning by prior medical knowledge. However, this method is computationally expensive since it uses a graph for medical domain knowledge to link every symptom in a relational matrix. Additionally, other models used Deep Q network (DQN)[22] [7] agents typically explore via the ϵ-greedy heuristic, this strategy tends to fail due to the reward sparsity and huge action space problems. So, [5] explores via Thompson sampling and proposes a Bayesian exploration strategy called BBQN that encourages a dialogue agent to explore state-action regions, proving that it outperforms the DQN using ϵ-greedy. In [14] they used their novel REFUEL algorithm that does a (reward shaping and feature rebuilding techniques) to alleviate the problems of dialogue system, by giving the agent an additional reward to guide the search towards better directions in the sparse feature space which solves the large action space problem, and a Feature rebuilding enables the agent to discover the key symptoms by learning a better representation. However, they used and evaluated on synthetic/simulated dataset, that cannot reflect what happens in real-world scenarios.

Unlike other popular diagnosis approaches, which prioritize both symptom inquiry and disease diagnosis equally, [29] their method only allows for disease diagnosis after evaluating all possible future outcomes; only when the candidate symptom can cause alternative diagnosis results will it be investigated and only when the most valuable inquiry has no impact on future diagnosis will the projected disease be informed.

**Hierarchical reinforcement learning (HRL):**

The motivation of (HRL) is that RL suffers from curse of dimensionality in complex problems [26], the goal of HRL is to divide the complex problem into smaller ones. In [27] they proposed a hierarchical reinforcement learning for symptom checking, by dividing the problem space as anatomical parts of the human body and each part has its own symptoms then trained an individual policy for each group. Introducing a latent layer called ‘master’ agent, that cannot directly perform inquiry and prediction actions, instead the ‘master’ agent's responsibility is to select one of the anatomical part models at each time step, treating the other models as subroutines. Moreover, they added contextual information about the patient. They showed that these enhancements performed better than their previous model [7]. However, the suggested approach looks at symptoms in accordance with the chosen policy after activating group policies using a rule-based framework. When there are numerous diseases or groups with significantly overlapping symptoms, a rule-based system like this will take a lot of time and human labor.

The dialogue agent proposed by [28], has 4 main components, LSTM based NLU (Natural language understanding), a ‘global’ state tracker, a dialogue policy using the hierarchical schema, and a NLG (Natural Language Generation module), they formulated their approach in the *options* framework proposed by [32], which is known as semi-MDP. The agent consists of 3 components, a top-level (Master) dialogue policy learning, a low-level dialogue policy learning, and an internal critic, which gives an intrinsic reward adjusting the low-level dialogue policy based on how likely a particular subtask is completed, later the user will give an extrinsic reward for the high-level dialogue policy, and a new state. As shown In Figure 2.

A diagram of a dialog

Description automatically generated

Figure 3: The hierarchy of a composite dialogue agent presented by [28]

The top-level receives a state and selects a subtask(output), where the low-level receives state and a subtask as input then outputs a primitive action as output.

Following the same formulation of semi-MDP, [8] Continued with the hierarchical approach by adding another component called disease classifier and editing the low-level dialogue policy, the low-level dialogue policy now consists of group of workers, where at each turn, the master chooses whether to activate a worker, or the disease classifier. Activating a worker causes the activated worker to interact with user multiple turns until the subtask is terminated. Each worker is responsible for a specific group of symptoms to collect from the user. As shown on Figure 4.

A diagram of a diagram

Description automatically generated

*Figure 4: The actions of Master and Worker.*

In [9], they added multiple more components to improve the actions of the Master and the Workers, by adding a Worker Confidence Critic (WCC), a PCM (potential candidate module), a learner critic, and a Hierarchical Disease Classifier (HDC), the (WCC) serves as a additional immediate reward to motivate the master to choose an appropriate worker, while the PCM receives a candidate disease from (HDC), then sends a potential symptoms to learner critic that gives additional guidance to the agent. This approach using the PCM module with DQN showed a ~81% accuracy.

**Adversarial networks in RL:**

As explained before on Actor-critic method, one’s can notice the similarity between GANs and Actor-critic. However, the main difference between both, is that in Actor-critic the improvement or learning, is collaborative between the 2 networks, but in GANs, it is adversarial.

The generator in GANs tries to fool the discriminator by creating data as realistic as possible.

A diagram of a program

Description automatically generatedIn [12] the researchers proposed a new method for dialogue policy, using the Actor-critic and inspired by GAN, they made an adversarial advantage actor-critic method, trained a discriminator to distinguish between dialogue agents' responses (or actions) and those of human experts. Next, the output of the discriminator acts as an extra intrinsic reward for the agent to explore the regions that acts similar to human experts, the generator in this case is an actor that selects actions, this method still requires prior knowledge of the user’s goal. Figure 5 shows the proposed method.

Figure 5: Illustration for the proposed method by [12]

In [17] they used adversarial reward as the only source of reward for policy optimization.

The problem of the two methods proposed, is that the discriminator's reward is only available after the entire dialogue has been formed, so that the guiding reward signal is still sparse while the generator is training, this may result in dialogues that lack humanlike qualities.

In [21] to avoid this problem they made 2 discriminators, turn-based(local) and a dialogue-based(global) discriminator, allowing the generator to update its policy efficiently. The generator is local cascaded attentive, that interacts with the user to fulfill a complete dialogue, it takes utterance of the user, the key slot values and the response of the agent at previous turn, then it outputs key slot values and the response at current turn. Note that these methods are done on non-medical dataset, but for task-oriented dialogues.

Another approach [16] that relies on concept of Inverse reinforcement learning (IRL), and to give a brief background for the concept of IRL, it essentially observes the policy then it deduce its reward function that the policy is following, assuming the policy is the optimal one(or near optimal), after learning the reward function, a value-based or policy based algorithm is used to make the policy. In [16], different from the IRL where reward function is learnt before the policy, they used adversarial networks, to learn the reward function and the policy simultaneously in an alternate way, thus improving each other during training.

Due to limitation on the use of adversarial networks to policy-gradient based algorithms (i.e. on-policy) as mentioned before, [15] provided a new approach that decompose the adversarial learning into 2 sequential steps, allowing the approach to be applicable to both on-policy and off-policy. By training a discriminator using an auxiliary generator before integrating a derived reward function into a normal RL model.

While all these papers are not in the domain of automatic diagnosis, [18] proposed a method that integrates GANs in RL, using policy-gradient framework.

In GAMP [18] the discriminator’s job is to estimate a sequence of symptoms to be either ‘real’(asked by real doctors) or ‘fake’. While the generator’s job is to generate a sequence of symptoms for the discriminator. The architecture of the framework consists also of an inference engine, that infer possible diseases at a given state, the network of the inference engine is same as the discriminator, except that the discriminator uses a binary classification as the output layer, and the inference engine outputs a disease probability distribution.

Moreover, the architecture includes a Mutual information step, which computes the mutual information of the current disease distribution and the adjacent next state disease distribution, then the reward from the discriminator is the action-value function which approximates the value when take an action at a current state, along with the Mutual information reward, are used to update the policy.

*A diagram of a network

Description automatically generated*

*Figure 6: Illustration of GAMP[18] framework*

In [19], since the action space and the problem they are working in is multi-domain dialogue dataset, they structured a multi-level discriminator and generator, where the learned discriminator can give reward in terms of domain, act and slot. In their framework, the generator (policy) and discriminator (reward estimator) are trained alternatively.

**3- Methodology**

In this section, we would like to divide it into 3 subsections, where in each subsection we will talk about and discuss the reasoning behind different part of our project, in the first section we will discuss the overall methodology and the framework we are using, while in the second subsection we will discuss further our different experiments with their results, and finally we will discuss our final implementation.

**1-Our Framework:**

A single rollout (or episode) is initiated within the Manager class, it starts with the user \_reset\_() method, that takes the first goal from the dataset, initiate the user for the dialogue, by modifying the current state of the user with the explicit symptoms, the user also reserve the implicit symptoms and the disease tag for future need as well as holds a symptom to index dictionary. He then returns the observation (initially it is a torch vector label encoded with the explicit symptoms). Assume the symptom set S = [0: ‘cough’, 1: ‘vomit’, 2: ‘headache’, 3:’runny nose’, 4: ‘anorxia’, 5: ‘expectoration’, 6: ‘difficulty breathing’, …] and the

user\_goal G = {

‘**request\_slot’** : {‘*disease’*:’UNK’},

**‘explicit\_inform\_slots’**: {‘*anorxia’*: True, ‘*cough’*: True, ‘*vomit’*: False}, ‘**implicit\_inform\_slots’**: {‘*expectoration’* : True, ‘*headache’*: False},

**‘disease\_tag’**: pneumonia }

Thus, the observation initially is the following: [3, 2, 0, 0, 3, 0 … ] where 3 is True(present) symptoms 2 is False(absent) symptoms, and 1 not-mentioned, 0 is the initial state.

The agent and the user interacts in a while loop until the *done* value is True that indicates the episode is over, the observation is then sended to the agent, for him to choose an action, with choose\_action() method, it returns the action, value from the value network and the probabilities for the loss computation. Then the chosen action is wrapped up as dictionary and sent to the user using the \_step\_() method which handles the action in 2 ways:

* 1. If request action is sended, it takes the requested slot and changes the state vector. Assume the action as follows: {‘action’: ‘request’, ‘request\_slots’: {‘runny nose’:UNK},’disease’:’UNK’} then the new state vector(observation) is [3, 2, 0, ***1,*** 3, 0 … ] which indicates the symptom “runny nose”is not-mentioned in the implicit symptoms.
  2. If inform action is sended, the user checks if the informed disease is equel to the disease\_tag from the user goal, if it equels then it sends the dialogue\_status as True, and False otherwise. And point the done to True indicating the end of the episode.

At the end of the \_step\_() method depending on the actions taken, the user returns a reward or a penality, the state-action-reward tuple is then saved in a PPOMemory.

After N steps, where each step is the single call of the \_step\_() method, the learn() method from the agent is called, which does the following:

1. **Data Preparation**:
   * It retrieves stored trajectories (state-action-reward tuples) from the agent's memory.
   * These trajectories are divided into state, action, old probability, value, reward, and done arrays, along with mini-batches for training.
2. **Advantage Calculation**:
   * An advantage estimate is computed using a discounted sum of future rewards adjusted by the value function, with the discount factor (γ-gamma) and GAE lambda.
   * This is done iteratively for each time step, resulting in an advantage tensor.
3. **Batch Processing**:
   * For each batch, states, actions, and old probabilities are extracted and converted to tensors on the appropriate device.
   * The actor network produces a distribution over actions, and the critic network provides value estimates for the given states.
4. **Loss Computation**:
   * **Actor Loss**:
     + New probabilities are calculated from the action distribution.
     + The probability ratio between new and old actions is computed.
     + The loss is calculated using both the original and clipped probability ratios to ensure stable policy updates.
   * **Critic Loss**:
     + The critic loss is computed as the mean squared error between the estimated returns (advantage plus values) and the predicted values.
   * **Entropy**:
     + The entropy of the action distribution is calculated to encourage exploration.
5. **Backpropagation**:
   * The total loss, combining the actor loss, critic loss, and entropy term, is used for backpropagation.
   * Gradients are calculated and applied to update the network parameters.
6. **Memory Management**:
   * The memory is cleared to prepare for the next cycle of interaction and learning.

The run on all episodes is then repeated for number of epochs, the number of epochs is generally large, ranging between 1000 and 5000. Here is a psuodocode for the algorithm and the mentioned functions for the agent.

**Algorithm: Reinforcement Learning with PPO**

**Initialize** environment, agent, memory, parameters

**For** epoch = 1 to n\_epochs:

**For** episode =1 to train\_episodes:

1. N\_steps = 0’
2. Hits = 0
3. **Initialize** user with a new goal
4. observation ← initial state
5. **While** done is False:
   * action, value, prob ← agent.choose\_action(observation)
   * next\_observation, reward, done, dialogue\_status ← user.step(action)
   * memory.store(observation, action, reward, prob, value, done)
   * observation ← next\_observation
   * If N\_steps % N == 0:
     + agent.learn()
6. if dialoge\_status is True:
   * hits +=1

**Function: agent.choose\_action(observation, turn, max\_turn)**

1. action← PolicyNetwork(observation)
2. value ← ValueNetwork(observation)
3. prob ← action distribution
4. Return action, value, prob

#### **Function: agent.learn()**

1. trajectories ← memory.retrieve()
2. **Data Preparation:** states, actions, rewards, values, probs
3. **Advantage Calculation:** estimate using GAE
4. **For** each batch in trajectories:
   * **Compute Actor Loss:**
     + new\_probs ← PolicyNetwork(states)
     + ratio ← new\_probs / old\_probs
     + actor\_loss ← -min(ratio \* advantage, clipped\_ratio \* advantage)
   * **Compute Critic Loss:**
     + values ← ValueNetwork(states)
     + critic\_loss ← MSE(advantage + rewards, values)
   * **Compute Entropy:**
     + entropy ← -sum(new\_probs \* log(new\_probs))
   * **Total Loss:**
     + total\_loss ← actor\_loss + vf\_coeff \* critic\_loss - entropy\_weight \* entropy
   * **Backpropagation:**
     + Update network parameters using gradients
   * **Reset Memory**

#### **Function: user.reset()**

1. **Initialize** user with the first goal from the dataset:
   * goal ← get\_goal()
   * explicit\_symptoms ← goal.explicit\_inform\_slots
   * implicit\_symptoms ← goal.implicit\_inform\_slots
   * disease\_tag ← goal.disease\_tag
   * symptom\_to\_index ← [for each idx : symptom in enumerate(symptom\_list)]
2. **Modify** current state with explicit symptoms:
   * state ← initialize\_state\_vector()
   * **For each** symptom, value in explicit\_symptoms:
     + state[symptom\_to\_index[symptom]] ← value
3. **Return** observation

#### **Function: user.step(agent\_action)**

1. **If** action['action'] == 'request':
   * requested\_slot ← action['request\_slots']
   * **If** requested\_slot in implicit\_symptoms:
     + state[symptom\_to\_index[requested\_slot]] ← value\_of\_symptom
   * **Else**:
     + state[symptom\_to\_index[requested\_slot]] ← 1
2. **If** action['action'] == 'inform':
   * informed\_disease ← action['disease']
   * **If** informed\_disease == disease\_tag:
     + dialogue\_status ← True
     + reward ← positive\_reward
   * **Else**:
     + dialogue\_status ← False
     + reward ← Negative\_reward
3. done = True
4. **Return** state\_vector, reward, done, dialogue\_status:
   1. **The Flowchart of the system can be shown in the Appendices.**

**2-Experiments and Results:**

We implemented baseline PPO agent from scratch (similar to framework of the previous section), for easy control and flexibility of the code, we evaluate on = , and divided the dataset for 20% testing set(104 episodes) and 80% training. However, unexpectedly the results were poor, getting only ~33% accuracy. The model exhibited rapid convergence, predominantly selecting the "inform" action to guess diseases rather than requesting more symptom information. This behavior indicated that the model quickly settled into a local optimum, prioritizing immediate guesses from the disease set over more informative symptom inquiries. Show table 1.

Then we headed towards multiple experiments and walkarounds to enhance the performance of the PPO agent before going to the next step.

**1. Addressing Local Optima in High Dimensional Action Spaces**

**Experiment 1: Redefining the Reward Function**

To address this, we redefined the reward function to provide additional fixed rewards for each turn. The agent received positive rewards for requesting relevant, non-redundant symptoms and penalties otherwise. This change led the model to request symptoms more frequently, sometimes reaching the maximum number of turns without making an "inform" action, suggesting a new local minimum focused on symptom requests.

**Experiment 2: Enforcing Inform Actions**

To mitigate the new issue, we forced the model to take an "inform" action at the maximum turn. The slot chosen was the most probable disease at that turn. While this ensured an "inform" action, the model's probabilities remained skewed towards requests, rendering the forced "inform" actions meaningless and ineffective in altering the model's state-action beliefs.

**Experiment 3: Dynamic Inform Action Threshold**

Recognizing the importance of the inform action choice, we introduced a confidence threshold for disease certainty. The agent would inform the most probable disease once this threshold was met, at any point during the dialogue. Although this approach balanced dialogue length, the model either acted too early or delayed excessively before informing a disease.

**Function: agent.choose\_action(observation, turn, max\_turn)**

1. action\_probs ← PolicyNetwork(observation)
2. disease\_action\_prob ← max(action\_probs[first\_disease\_index:end])
3. **If** disease\_action\_prob ≥ threshold **or** turn == max\_turn:
   * action ← disease\_action
4. **Else:**
   * action ← PolicyNetwork.sample\_action(action\_probs)
5. act\_as\_dict = wrap\_action(action)
6. value ← ValueNetwork(observation)
7. prob ← action distribution
8. Return act\_as\_dict, action, value, prob

The results after implementing these changes enhanced the performance getting ~39% accuracy. However, further model improvements is required.

**2. Architectural Enhancements**

To further refine action selection, and to tackle the issue of huge action space, we implemented the following techniques.

* **Dual-Network Approach**

We implemented new architectures to divide the action space and reduce dimensionality. The first architecture introduced a secondary network with the same input from a shared layer, outputting the disease list. This network's loss function, based on Mean Squared Error (MSE) from the final disease tag, influenced the shared layer, with actions chosen based on the higher probability between the actor and disease classifier. Although this increased the accuracy to 55% but it was not actually reinforcement learning but supervised learning, since the agent was always taking action from the disease classifier and extra information about the disease must be known.

* **Reinforced Dual-Actor Model**

To align the disease classifier with reinforcement learning principles, we modified its loss function to mirror the actor's clipped surrogate loss, effectively creating two actors: one for requests and one for informs. Actions were chosen based on the confidence threshold of the disease classifier, balancing reinforcement learning and supervised learning elements. However, this approach introduced additional hyperparameters, maintaining the challenge of hyperparameter sensitivity.

**2. Enhancing Input State Representation**

* **Initial State Encoding**

Our initial input state utilized a multi-value encoded vector and a one-hot encoded vector, representing symptoms as unasked (0), not mentioned (1), absent (2), or present (3)., and for further experiments we then changes the state into one hot encoded vector, this allowed for flexibility in activation function experimentation such as using TanH().

* **Feature Engineering and State Tracking**

We extended the input state with engineered features derived from the dataset, linking diseases to their dependent symptoms. The disease score was calculated using the formula:

Additionally, a state tracker component was introduced to maintain a history of user and agent actions throughout the dialogue. This component tracked requested and informed slots and incorporated the last actions into the subsequent state vector, enhancing the informativeness of the state at the cost of increased dimensionality.

* **State Tracker Methods**

The state tracker helps the agent relate to the state in the overall context of the dialogue, and treat the states as part of the dialogue rather than individual, as well as, previously when agent asks about an implicit symptom that is not mentioned, in turn 0 and, let’s say turn 12, the agent doesn’t know which symptom he asked first, or at which turn.

The state tracker featured three primary methods: reset(), update(), and get\_state(). The update method modified the state with each interaction, while get\_state provided the current state to the agent for decision-making. This setup ensured continuous state updates reflecting the dialogue's progression.

**reset**(): only reset the states, most importantly the history list, and other components for the new dialogue. Here's the pseudocode for the state tracker component with the update(), and get\_state() methods:

### Function: update(action)

1. **If** user\_action:
   * **For each** slot, value in action['inform\_slots']:
     + current\_slots['inform\_slots'][slot] ← value
   * current\_slots['disease'] ← action['disease']
   * history.append(action)
   * turn\_count += 1
2. **If** agent\_action:
   * history.append(action)
   * turn\_count += 1
   * **If** agent\_action['action'] == 'inform':
     + current\_slots['disease'] ← action['disease']

**Function: get\_state()**

1. disease\_vector ← self.calculate\_disease\_vector(current\_slots['inform\_slots'])
2. represent\_current\_slots ← [0] \* len(self.symptoms\_list)
3. **For each** slot, value in current\_slots['inform\_slots']:
   * represent\_current\_slots[self.slot\_to\_index[slot]] ← value
4. **If** history and last agent action is 'inform':
   * last\_agent\_action ← [1]
5. **Else if** history and last agent action is not 'inform':
   * last\_agent\_action ← [2]
6. **Else**:
   * last\_agent\_action ← [0]
7. **If** history and last user action is 'inform':
   * last\_user\_action ← [1]
8. **Else if** history and last user action is not 'inform':
   * last\_user\_action ← [2]
9. **Else**:
   * last\_user\_action ← [0]
10. represent\_user\_slots ← [0] \* len(self.symptoms\_list)
11. **If** last user action:
    * **For each** slot, value in last user action['inform\_slots']:
      + represent\_user\_slots[self.slot\_to\_index[slot]] ← value
12. represent\_agent\_slots ← [0] \* len(self.symptoms\_list)
13. **If** previous agent action:
    * **For each** slot in previous agent action['request\_slots']:
      + represent\_agent\_slots[self.slot\_to\_index[slot]] ← 1
14. final\_representation ← concatenate(represent\_current\_slots, represent\_user\_slots, represent\_agent\_slots, disease\_vector, last\_agent\_action, last\_user\_action)
15. **Return**: final\_representation

The results after adding the state tracker improved to ~45%, this shows that this component helped the overall performance of the agent. However, at some epochs the model was starting at bad random spots which made him stick in local optima or taking long time to learn.

* **Expert Trajectories**

To address the rapid convergence to a local minimum, we employed expert trajectories to enhance the model's learning efficiency. Expert trajectories consist of optimal actions that the agent should take in specific states to achieve the highest possible reward. By following these trajectories, the agent can bypass the inefficiencies of random initialization and learn more effectively in the early stages of training.

In our implementation, we used expert learning for 50 episodes, equivalent to 10% of the DXY dataset. During these episodes, the agent was supervised to follow the optimal paths, maximizing rewards and building a robust policy foundation quickly. This method aimed to provide the agent with a solid starting point. Expert learning increased the speed of learning but did not increase the performance much, and only up to 47%.

After these observations we decided the necessity of doing hyperparameter tuning to test the impact the different hyperparameter has on the model and to enhance the performance. However, the *computational complexity* was **prohibitive**, the model took about 2 days to compute 100 trails on **50 epochs** only. Then we implemented DQN since it is less sensitive to hyperparameters as well as it has less hyperparameters to tune, but it was also computationally prohibitive. All the trainings happened on GPU GTX 1660-ti.

* **Using OpenAI package:**

Until now we have been doing experiments on a completely from scratch code, which gave us flexibility and control over the code and the structure of the code. However, it was computationally extensive. The complexity of doing hyperparameters tuning was also computationally prohibitive and did not allow us to train on big number of epochs, which is a crucial part in reinforcement learning.

Then we implemented the Stable Baselines package. This package offers the flexibility to use various reinforcement learning algorithms and, more importantly, facilitates hyperparameter tuning over shorter periods and across more epochs. A key feature of Stable Baselines that significantly enhances training efficiency is the use of a ***Vectorized Environment.*** [46]

Vectorized Environment enables multiple instances of an environment to run in parallel, significantly accelerating the training process for reinforcement learning agents. By allowing parallel execution, it facilitates the simultaneous collection of experiences or trajectories from multiple environments, thereby providing the agent with more diverse experiences in a shorter time. This diversity helps the agent learn more efficiently and enhances its performance across different scenarios. Additionally, the ability to run multiple environments concurrently supports extensive hyperparameter tuning. Tools like the Stable Baselines package can explore a wider range of hyperparameter values and configurations more quickly, leading to more optimal settings for the agent. The increased number of epochs available for training due to parallel execution ensures that the agent has ample opportunities to refine its policy and value functions, ultimately improving its overall performance.

A custom environment for reinforcement learning should be made when using Stable Baseline package, It inherits from gym.Env and initializes with lists of symptoms and diseases. The observation space consists of multiple discrete values representing the state of each symptom (True, False, or Unknown), while the action space allows the agent to either inquire about a symptom or predict a disease.

* **Initialization**: The environment is initialized with the given symptoms and diseases, defining the observation and action spaces. It also sets the maximum number of steps per episode to 20.
* **reset()**: This method resets the environment at the start of each episode, obtaining a new example from the dataset and initializing the state with unknown symptoms.
* **step(action)**: The step function processes the agent's actions. If the action corresponds to asking about a symptom, it updates the state based on the simulated patient's response and applies a small reward. If the action is a disease prediction, it evaluates the correctness, assigns an appropriate reward, and ends the episode.
* **render()**: This method is used to visualize the result, indicating whether the agent's disease prediction was correct.

**Function: step(action)**

1. **If** action < num\_symptoms:
   * symptom\_idx ← action
   * state[symptom\_idx] ← dataClass.evaluate\_symptom(symptom\_idx)
   * reward ← asking\_reward
   * done ← False
2. **Else**:
   * disease\_idx ← action - num\_symptoms
   * is\_correct ← dataClass.evaluate\_disease(disease\_idx)
   * reward ← positive\_reward if is\_correct else negative\_reward
   * done ← True
3. current\_step += 1
4. **If** current\_step ≥ max\_steps:
   * done ← True
5. **Return** state, reward, done

Moreover, a Dataset class has been made, to easily manage data for the environment. The environment then must be registered for the PPO agent to interact with.

After implementing the PPO agent using the package the results were better achieving 61 hits for 104 test episodes, that is 58% accuracy, as well as the acceleration in training was noticeable than our implementation with about 1000 epochs. This acceleration enables us to make more experiments to enhance the performance by doing hyperparameter tuning.

Therefore, after doing hyperparameter tuning, to the Stable Baseline package the accuracy of the model achieved 68% on 100 trail and on 1000 epoch, the model took 16 hours training.

**Embeddings**

Moreover, we implemented Embeddings to alleviate the problems discussed earlier[36], in particular the experiment called Triplet loss, as explained in [47], the Triplets has been one of the most popular loss functions for supervised similarity or metric learning ever since. In its simplest explanation, Triplet Loss encourages that dissimilar pairs be distant from any similar pairs by at least a certain margin value. Mathematically, the loss value can be calculated as where:

anchor (): A reference point in the dataset.

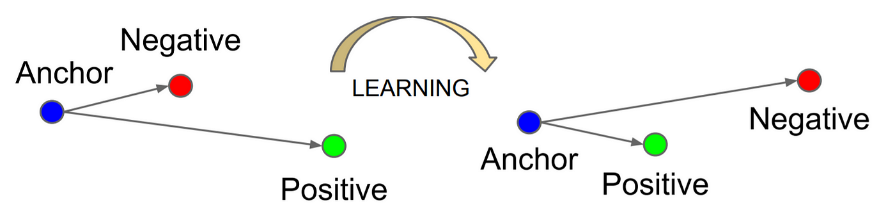
- , i.e., positive, is a sample that has the same label as ,

- , i.e., negative, is another sample that has a label different from ,

- is a function to measure the distance between these three samples,

- is a margin value to keep negative samples far apart.

Figure 7 the triplets before and after learning



The triplet loss will bring the positives closer to anchor and push out the negatives, in our implementation we applied triplet anchors in 2 different contexts:

* 1. We trained the model to learn embeddings that differentiate between contextually related and unrelated dialogues. In other words, the positive points will come closer if they are in the same dialogue, negatives will be pushed far back.



* 1. We trained the model to learn embeddings that differentiate between similar diseases dialogues. In other words, the positive points will come closer if they are in the same disease, while other diseases will be pushed far off.



The results for both methods were interestingly not performing much better than the baseline PPO, grouping by dialogue and grouping by disease got 62% and 64% accuracy, respectively.

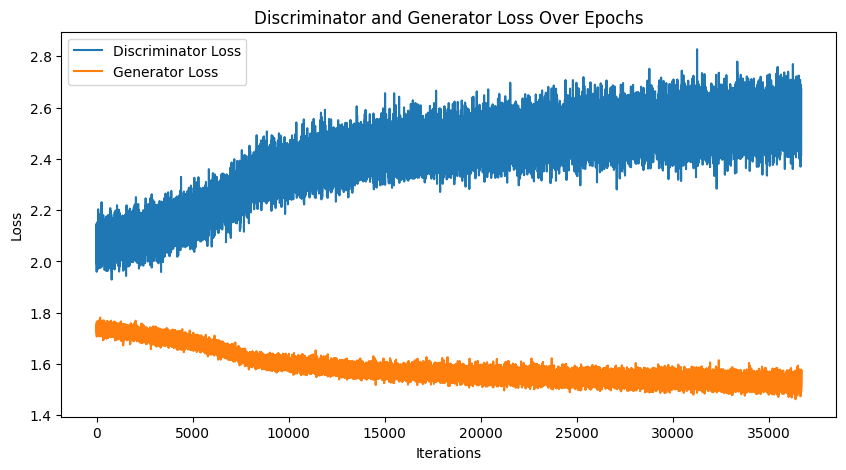
**General Adversarial Networks**

In this subsection we experimented GAN architecture isolating it from the PPO agent to test the performance, the adversarial network requires making a perfect agent actions dataset. The generator's goal is to generate realistic actions based on the given states, outputs actions, while the discriminator's goal is to distinguish between the real expert actions with a given state and the actions generated by the generator by the given state, and outputs a value ranges between 0 and 1, indicates how real he thinks the action is. we made it as label encoded with Embeddings, the loss for the generator is calculated as follows:

Where , indicating the generator's goal to fool the discriminator, and , ensuring the generator produces actions close to the real actions.

And the discriminator’s loss is computed as follows:

Where , Binary cross-entropy loss comparing predictions to a tensor of ones (indicating real data), and Binary cross-entropy loss comparing predictions to a tensor of zeros (indicating fake data).



The GAN is currently experiencing an imbalance where the generator is improving, but the discriminator is not keeping up, as indicated by its increasing loss. By adjusting learning rates, improving the discriminator's training, and ensuring balanced training, you can work towards achieving the optimal balance where both losses stabilize, typically around 0.5 for the discriminator and a stable, lower value for the generator. [see figure 8]

Figure 8 Generator and Discriminator loss over epochs

In a scenario where the dataset is small, the discriminator has a limited amount of data to learn from. Given sufficient training, the discriminator can become exceptionally good at distinguishing real data from the generator’s fake data.

As a result, the discriminator quickly learns the nuances of the small dataset, making it nearly impossible for the generator to fool it. The discriminator’s accuracy becomes so high that it provides little to no constructive feedback to the generator.

The generator relies on the discriminator's feedback to improve. If the discriminator is too powerful and confident (due to overfitting on the small dataset), it will consistently classify the generator's outputs as fake with high certainty.

**Auto Encoder**

As mentioned earlier, Autoencoders offer a powerful solution for handling categorical data by transforming it into a latent space. This transformation involves encoding the categorical data into a numerical format that captures its essential characteristics while reducing dimensionality. The key advantages of using autoencoders for categorical data include:

Autoencoders reduce the high-dimensional categorical data into a lower-dimensional latent space, which helps in mitigating the curse of dimensionality and improves model efficiency.

As well as the autoencoders made the PPO agent less sensitive to hyperparameter tuning, by making him more stable and robust with changes in hyperparameters and less susceptible local minima. The results showed good results in autoencoder PPO.

They automatically learn and extract relevant features from the data, capturing intricate relationships and patterns that might be lost with traditional encoding methods like one-hot encoding. The flow of the Autoencoders is as follows:

**Input Data (Categorical) Encoder Latent Space Decoder Output Data (Reconstructed)**

The results were the best results we got so far, and they are as follows, the autoencoders without hyperparameter tuning achieved ~71%, and with hyperparameter tuning achieved (~81%)

**Experiments results summary**

In this subsection we will show how hyperparameters affect the performance of the PPO agent model, The default parameters are.[see table 3]

All the codes were running in the default parameters, and they are as follows [table 2]:

| **Hyperparameter** | **Default Value** |
| --- | --- |
| Learning Rate () | 3e-4 |
| Number of Epochs (n\_epochs) | 10 |
| Batch Size (batch\_size) | 64 |
| Gamma () | 0.99 |
| Clip Range (clip\_range) | 0.2 |
| Value Function Coefficient (vf\_coef) | 0.5 |
| Entropy Coefficient (ent\_coef) | 0.01 |
| GAE Lambda (gae\_lambda) | 0.95 |
| Discount factor | 0.99 |

Table 2 The default hyperparameter values used for PPO

|  |  |  |
| --- | --- | --- |
| **Hyperparameter** | **Stable Baseline PPO** | **Auto Encoder** |
| Learning rate = 0.002 | ~67 % | ~**69%** |
| Learning rate = 0.004 | **~**0.19% | **~71.9** |
| Learning rate = 0.0003 | ~68% | **~72%** |
| Learning rate = 0.00002 | **~**23% | **~55%** |
| Value function coefficient = 0.2 | ~63% | **~73%** |
| Value function coefficient = 0.4 | ~62% | **~77%** |
| Entropy coefficient =0.005 | ~62% | **~68%** |
| Entropy coefficient =0.05 | ~64% | **~68%** |
| GAE lambda =0.9 | ~64% | **~75%** |
| GAE lambda = 0.75 | ~59% | **~71%** |

Table 3 Different values for hyperparameters and how they affect performance

The results clearly show that the slightest change in hyperparameter, the baseline PPO performance changes drastically, while the Auto Encoder PPO is less susceptible to that change.

**3- Proposed implementation**

Our proposed implementation with the highest accuracy is the PPO agent using the Auto Encoder, the advantage of this implementation exceeds the increase of performance, but also the stability and robustness of the model to hyperparameters as well as the tackling issues like categorical data, this approach outperforms the hyper tuned baseline PPO agent accuracy, while other approaches failed to do so.

We defined the initial state with the define\_initial\_state function, categorizing symptoms as "explicit\_available," "explicit\_unavailable," or "default" and encoding them into a state vector. The state is dynamically updated using the change\_state\_based\_on\_feedback function, which modifies a symptom's status to "implicit\_available," "implicit\_unavailable," or "implicit\_unknown" based on feedback, provided the symptom was initially "default."

Then we created a dataset class for the autoencoder to handle the data inputted into the encoder, the state is randomly generated.

The AutoEncoder can be trained on a randomly generated dataset, because the main goal is to transform any state into a lower dimensional latent space. As opposed to GAN, where the discriminator has to learn on expert dataset, where each state must be from an expert path.

The network shape of the encoder is as follows:

We defined empirically a sequence\_length = 41.

The architecture of both Encoder and Decoder can be seen in the appendix.

The training function of the autoencoder, we use Adam optimizer. Here is a psueodcode of the training function:  
**Function: Train encoder(iterator):**

1. **For** **each** *epoch* from 1 to num\_epochs:
   1. Set epoch\_loss to 0
   2. **For** **each** batch (src, label) **in** iterator: # iterator provides batches of (src, label)
      1. Zero the gradients
      2. Compute hidden states from encoder
      3. Compute output from decoder
      4. If first batch, run assertions on output
      5. Flatten output and label
      6. Generate random predictions
      7. Compute loss and random loss
      8. Backpropagate loss
      9. Update model parameters
      10. Accumulate batch loss
   3. Update learning rate scheduler
   4. Get current learning rate
   5. Append epoch\_loss to losses
   6. Log epoch loss with plotter
2. **Return** average\_epoch\_loss

The system flow can be represented as follows:

The process starts by taking an example from the dataset and defining the initial state, which is then converted to a one-hot encoded vector of size 42x6. This vector is fed into the encoder, producing a latent space representation of size 20. The latent representation is then passed to the agent, which selects an action. The state is updated based on the feedback, and if the action is a disease prediction, it is evaluated. The updated state is again converted to a one-hot encoded vector, passed through the encoder to generate a new latent space representation, and then fed back to the agent. This cycle repeats continuously.

Representation of the flow can be seen in the Appendix.

**4) Discussion and results**

We conducted a series of experiments to improve the performance of our PPO agent for the dialogue system in a medical diagnosis setting. The initial implementation served as a baseline, and subsequent experiments aimed to address issues related to local optima, action space size, reward sparsity, and hyperparameter sensitivity. Below is a detailed discussion and analysis of these experiments.

### Baseline PPO Agent and Initial Reward Function Adjustments

The initial implementation of the PPO agent achieved an accuracy of approximately 33%. This poor performance is attributed to the model's tendency to rapidly converge to a local optimum, predominantly selecting the "inform" action without requesting sufficient symptom information. This behavior underscores the challenge of reward sparsity and the difficulty in balancing immediate rewards with the need for informative actions.

To address this, we conducted three related experiments focused on adjusting the reward function and action strategies:

1. **Redefining Reward**: We provided additional fixed rewards for each turn to encourage the agent to request more symptoms, resulting in a slight improvement to 35%.
2. **Enforcing Inform**: We forced the agent to take an "inform" action at the maximum turn, increasing accuracy to 38%. This ensured that the agent eventually made a diagnosis but did not significantly change its skewed action probabilities.
3. **Dynamic Inform Threshold**: Introducing a confidence threshold for the "inform" action marginally improved accuracy to 39%. This method aimed to balance dialogue length and decision-making timing but highlighted the challenge of setting an optimal threshold.

### Dual-Network Approach

Implementing a dual-network approach, which incorporated a supervised classifier, resulted in a significant accuracy increase to 55%. This approach reduced the action space dimensionality but shifted towards a supervised learning paradigm, which deviates from the reinforcement learning principles and required additional disease information.

### Reinforced Dual-Actor Model

Modifying the dual-network approach to align with reinforcement learning principles resulted in a drop in accuracy to 40%. This approach introduced additional hyperparameters, exacerbating the sensitivity issue and complicating the training process.

### Feature Engineering & State Tracking

Enhancing the input state representation with engineered features and a state tracker improved accuracy to 45%. The state tracker added contextual relevance to the agent's decisions but increased the state space dimensionality, complicating the learning process.

### PPO with Everything

Combining all enhancements into the PPO agent achieved an accuracy of 47%. This experiment underscored the incremental benefits of individual improvements but highlighted the need for effective hyperparameter tuning.

### Hyperparameter Tuning on PPO with Everything

Hyperparameter tuning on the fully enhanced PPO agent further increased accuracy to 53%. However, the computational complexity was prohibitive, taking 2 days for 100 trials over 50 epochs, demonstrating the challenge of optimizing hyperparameters in reinforcement learning.

### PPO Stable Baselines

Using the Stable Baselines package with a Vectorized Environment achieved an accuracy of 58% in approximately 15 minutes for 1000 epochs. This approach leveraged parallel execution, significantly accelerating training and enabling extensive hyperparameter tuning. The use of Stable Baselines made the model less sensitive to hyperparameter tuning, as the package's built-in tools facilitated more efficient and comprehensive optimization.

### Hyperparameter Tuning with Stable Baselines

Hyperparameter tuning with Stable Baselines further improved accuracy to 68%. This result underscores the importance of efficient hyperparameter tuning and the benefits of using established reinforcement learning frameworks. The process was markedly more efficient compared to the custom implementation, highlighting the advantage of using a well-optimized package.

### Triplet Loss Experiments

Implementing triplet loss for dialogue grouping and disease grouping achieved accuracies of 62% and 63%, respectively. These approaches aimed to improve the agent's ability to distinguish between contextually related and unrelated dialogues or diseases, but the improvements were modest. The integration of triplet loss demonstrated a method to address the issue of categorical data handling and reward sparsity, making the model less dependent on hyperparameter tuning by providing a more structured learning process

.

### PPO Auto Encoder

Using an AutoEncoder to reduce the latent space achieved an accuracy of 71% in approximately 20 minutes. This approach effectively reduced the state space dimensionality, improving the agent's performance. The AutoEncoder significantly addressed the problem of large action spaces, leading to more stable learning and reduced sensitivity to hyperparameter variations as well, as well as ~81% with hyperparameter tuning.

| **Experiment** | **Hyperparameter** | **Description** | **Accuracy** | **Training Time** |
| --- | --- | --- | --- | --- |
| Baseline PPO Agent | No | Initial implementation, default reward function | ~33% | - |
| Experiment 1: Redefining Reward | No | Additional rewards for requesting relevant symptoms, penalties otherwise | ~35% | - |
| Experiment 2: Enforcing Inform | No | Forced "inform" action at max turn | ~38% | - |
| Experiment 3: Dynamic Inform Threshold | No | Confidence threshold for disease certainty before informing | ~39% | - |
| Dual-Network Approach | NO | Use of supervised classifier | ~55% | - |
| Reinforced Dual-Actor Model | No | Dual actors for requests and informs, actions based on confidence threshold | ~40% | - |
| Feature Engineering & State Tracking | No | Engineered features linking diseases to symptoms, state tracker added | ~45% | - |
| PPO with everything | No | PPO agent with full additions | ~47% | 9 hours 1000 epochs |
| PPO with everything | YES | Hyperparameter tuning on PPO agent with full additions | ~53% | 2 days for 100 trials, 50 epochs |
| PPO Stable baseline | NO | Using Stable Baselines package with Vectorized Environment | ~58% | ~15 min, 1000 epochs |
| PPO (Stable Baselines) | YES | Hyperparameter tuning with Stable Baselines package | ~68% | 16 hours for 100 trail and 1000 epochs |
| Triplet Loss (Dialogue Grouping) | No | Embeddings with triplet loss for dialogue grouping | ~62% | - |
| Triplet Loss (Disease Grouping) | No | Embeddings with triplet loss for disease grouping | ~63.7% | - |
| **PPO AutoEncoder** | **No** | **AutoEncoder(latent space)** | **~71%** | **Approx. 20 min** |
| **PPO AutoEncoder** | Yes | AutoEncoder(latent space) | **~81%** | 16 hours for 100 trail and 10K step and 1000 epoch |

Table 4 Summary of performance and time taken for different experiments

**6-Reference**

[1] End-to-End Optimization of Task-Oriented Dialogue Model with Deep Reinforcement Learning

[2] Machine learning for medical diagnosis: history, state of the art and perspective

[3] End-to-End Knowledge-Routed Relational Dialogue System for Automatic Diagnosis

[4] Continuously learning neural dialogue management

[5] BBQ-Networks: Efficient Exploration in Deep Reinforcement Learning for Task-Oriented Dialogue Systems

[6] Hybrid Code Networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning

[7] Inquire and Diagnose: Neural Symptom Checking Ensemble using Deep Reinforcement Learning

[8] Task-oriented Dialogue System for Automatic Disease Diagnosis via Hierarchical Reinforcement Learning

[9] A knowledge infused context driven dialogue agent for disease diagnosis using hierarchical reinforcement learning.

[10] Hierarchical reinforcement learning for automatic disease diagnosis.

[11] Efficient exploration for dialogue policy learning with BBQ networks & replay buffer spiking

[12] ADVERSARIAL ADVANTAGE ACTOR-CRITIC MODEL FOR TASK-COMPLETION DIALOGUE POLICY LEARNING

[14] REFUEL: Exploring Sparse Features in Deep Reinforcement Learning for Fast Disease diagnosis.

[15] Guided Dialogue Policy Learning without Adversarial Learning in the Loop

[16] Guided Dialog Policy Learning: Reward Estimation for Multi-Domain Task-Oriented Dialog

[17] Adversarial learning of task-oriented neural dialog models

[18] Generative Adversarial Regularized Mutual Information Policy Gradient Framework for Automatic Diagnosis

[19] Imperfect also deserves reward: Multi-level and sequential reward modeling for better dialog management.

[20] Amalgamating Knowledge from Two Teachers for Task-oriented Dialogue System with Adversarial Training

[21] HaGAN: Hierarchical Attentive Adversarial Learning for Task-Oriented Dialogue System

[22] Task-oriented Dialogue System for Automatic Diagnosis

[23] Ronald J Williams. 1992. Simple statistical gradient following algorithms for connectionist reinforcement learning

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms

[25] Bayesian-Based Symptom Screening for Medical Dialogue Diagnosis Zhong Cao, Yuchun Guo∗ , Yishuai Chen, Daoqin Lin School of Electronic and Information Engineering Beijing Jiaotong University

[26] A.G. Barto, S. Mahadevan, Recent advances in hierarchical reinforcement learning, Discrete Event Dyn. Syst. 13 (2003) 41–77.

[27] Context-Aware Symptom Checking for Disease Diagnosis Using Hierarchical Reinforcement Learning by Hao-Cheng Kao, Kai-Fu Tang, Edward Y. Chang, 2017

[28] Composite Task-Completion Dialogue Policy Learning via Hierarchical Deep Reinforcement Learning, Baolin Peng, Xiujun Li, Lihong Li, Jianfeng Gao, Asli Celikyilmaz, Sungjin Lee, Kam-Fai Wong.2018

[29] TOWARDS A RELIABLE AND ROBUST DIALOGUE SYSTEM FOR MEDICAL AUTOMATIC DIAGNOSIS

[30] POMDP-Based Statistical Spoken Dialog Systems: A Review, Steve Young; Milica Gašić; Blaise Thomson; Jason D. Williams

[31] Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Inigo Casanueva, Stefan Ultes, Osman Ra- ˜ madan, and Milica Gasic. 2018. Multiwoz-a largescale multi-domain wizard-of-oz dataset for taskoriented dialogue modelling.

[32] The Logical Options Framework; Brandon Araki, Xiao Li, Kiran Vodrahalli, Jonathan DeCastro, Micah J. Fry, Daniela Rus

[33] MultiWOZ - A Large-Scale Multi-Domain Wizard-of-Oz Dataset for Task-Oriented Dialogue Modelling

[34] Deep Reinforcement Learning for Dialogue Generation Jiwei Li, Will Monroe, Alan Ritter, Dan Jurafsky, Michel Galley, Jianfeng Gao

[35] Playing Atari with Deep Reinforcement Learning Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller

[36] Survey on categorical data for neural networks Open Access John T. Hancock\* and Taghi M. Khoshgoftaar

[37] CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING Timothy P. Lillicrap∗ , Jonathan J. Hunt∗ , Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra

[38] VIME: Variational Information Maximizing Exploration Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, Pieter Abbeel

[39] How to Build User Simulators to Train RL-based Dialog Systems Weiyan Shi⇤1, Kun Qian⇤1, Xuewei Wang2 and Zhou Yu1

[40] A User Simulator for Task-Completion Dialogues Xiujun Li, Zachary C. Lipton, Bhuwan Dhingra, Lihong Li, Jianfeng Gao, Yun-Nung Chen

[41] End-to-End Task-Completion Neural Dialogue Systems Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, Asli Celikyilmaz

[42] Github repository : <https://github.com/MiuLab/TC-Bot>

[43] Tuning Proximal Policy Optimization Algorithm in Maze Solving with ML-Agents

[44] Generative Adversarial Networks Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio

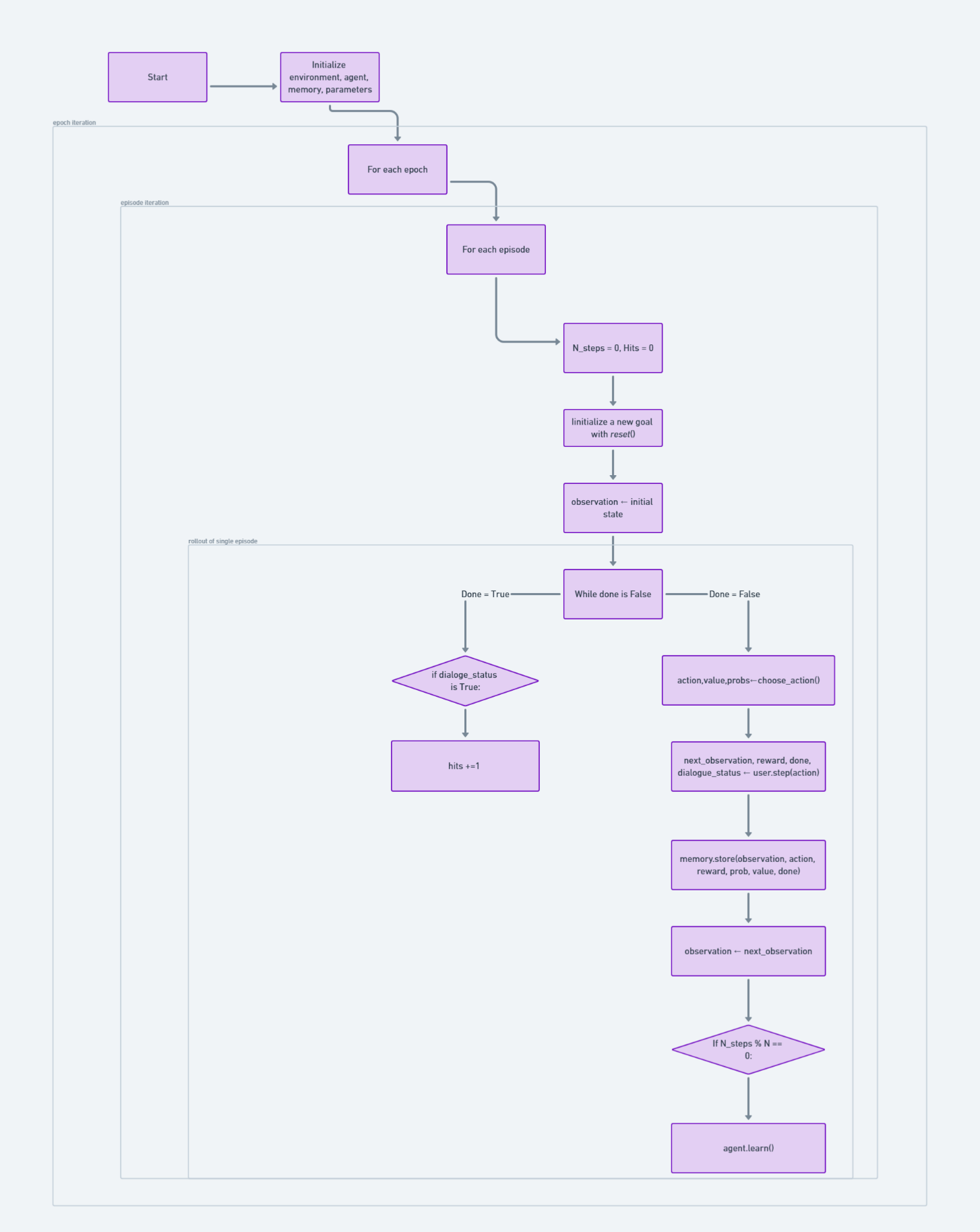
[45] Behavioral Cloning from Observation Faraz Torabi, Garrett Warnell, Peter Stone

[46] <https://stable-baselines.readthedocs.io/en/master/guide/vec_envs.html>

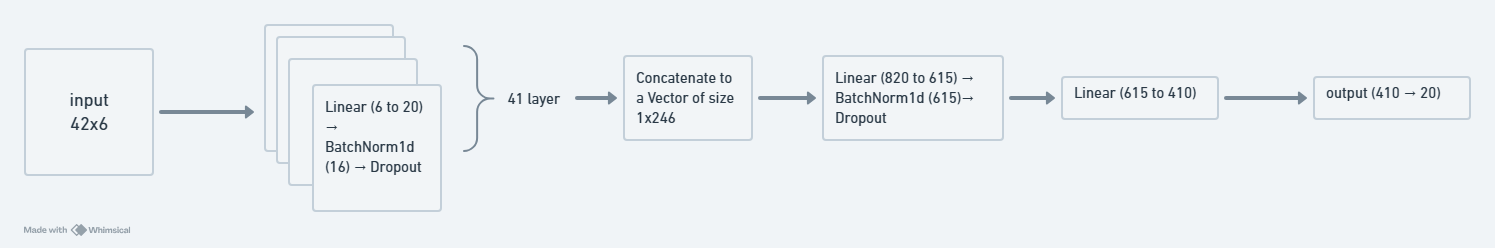
[47] FaceNet: A Unified Embedding for Face Recognition and Clustering

**Appendix**

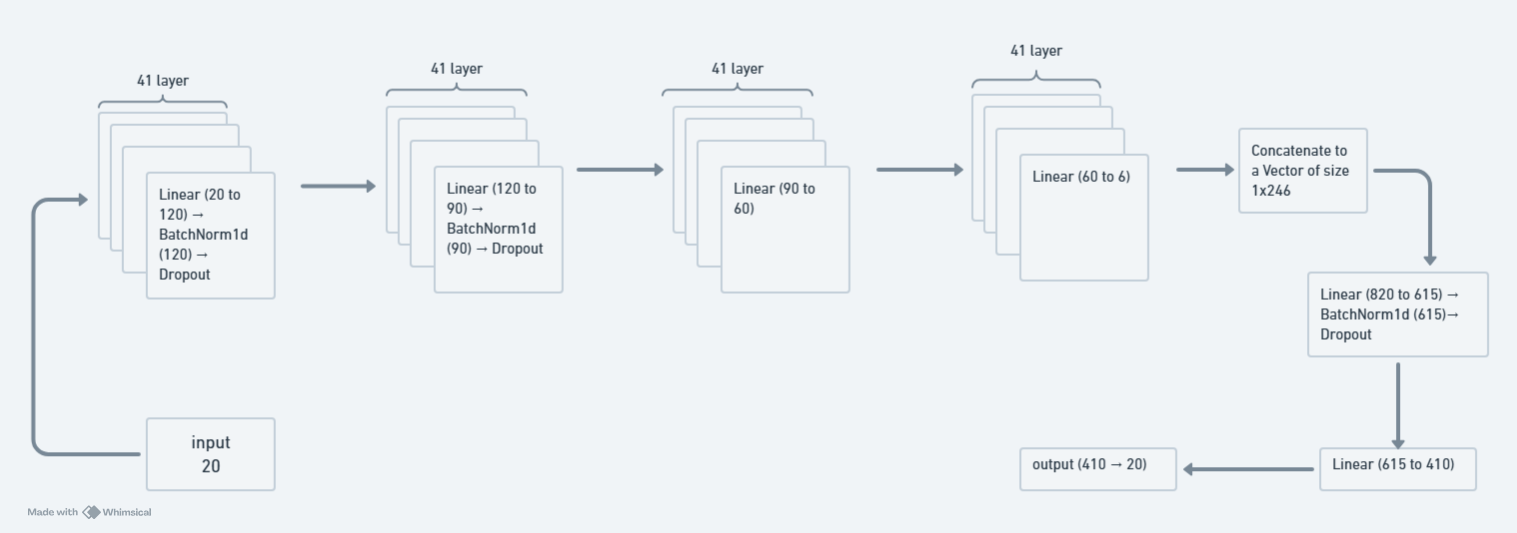
**Agent- user interaction flowchart**

****

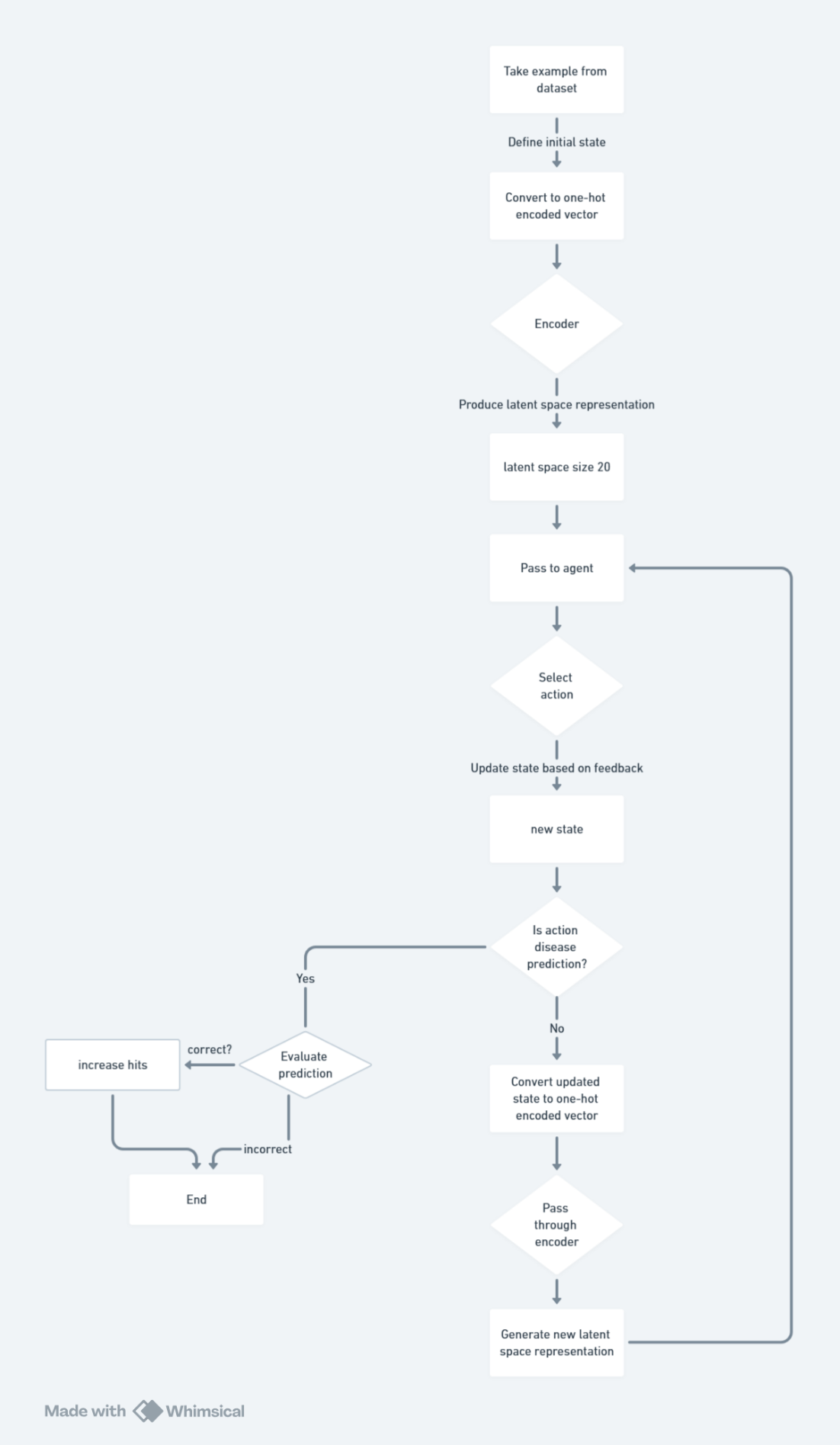
**Encoder architecture:**

****

**Decoder architecture:**

****

**PPO with autoencoder flow**

****

1. Imp= implicit symptoms. [↑](#footnote-ref-1)