# Learning Goal-Oriented Dialog Strategies using Deep Reinforcement Learning

Shishir Narayan
Berlin

October 25, 2018

# Declaration

I hereby declare that this thesis was composed solely by myself and that the work contained herein is my own except where specific references are made to the work of others or explicitly stated otherwise in the text. Further, I declare that this work has not been submitted, in whole or in part, for any other degree or professional qualification.

This thesis contains fewer than 30,000 words including the bibliography, footnotes, tables and equations.

Shishir Narayan
October 25, 2018

# Confidentiality Statement

This master thesis is provided with a blocking notice. Access to this thesis is not allowed to unauthorized persons or organizations. The master thesis may not be made accessible to the third party, with the expectation of the first and secondary referee as well as the members of the examination board of the SRH Hochschule Heidelberg neither completely nor in parts. Further, any publication, duplication or the passing on of the master thesis in parts are not permitted without the permission of the company *botconnect GmbH*[1]

Shishir Narayan
October 25, 2018

[1] www.botconnect.ai

# Acknowledgments

# Abstract

In this work we focus on applying reinforcement learning techniques to develop simple goal oriented dialog systems. The design and development of dialog systems is known to be a challenging task, with developers devoting large amounts of effort and time on foreseeing prospective user interactions to develop appropriate system responses. This mapping across predicted user inputs and the correct system outputs is called a dialog strategy or in terms of reinforcement learning, a dialog policy.

Here we model the design and development of this strategy as a sequential decision-making task. This is done by employing a framework known as Markovian Decision Processes(MDP) that helps to model decision-making tasks. Our task is goal-oriented or domain-specific, i.e. to assist the user in helping to sell a product to a customer. Thus our goal oriented dialog system, commonly known as a chatbot, helps the user in achieving this predefined goal by having a dialog with the user which consists of making suggestions or asking questions about the customer. According to user response and previous interactions, the system decides what to say next to help the user. The success of the system is directly dependent on the quality of the learnt policy, which is dependent on the complexity of the environment and user simulation. We use Deep Reinforcement Learning techniques which use the MDP model to learn the required dialog policies by interacting with a simulated user environment.

In this thesis we also detail the probabilistic model of the environment simulation, as well as the intricacies in casting our dialog domain as a MDP. Further, we delve in detail about the various reinforcement learning techniques and formulations that can be used to optimize dialog policies. Finally, we give a summary of deep learning before examining two reinforcement learning algorithms, Advantage Actor Critic and REINFORCE and their formulation using neural networks in detail. This is then proceeded by a documentation of the results obtained during the training of these algorithms and their comparisons with appropriate classic reinforcement learning baselines.

# Contents

# List of Figures



# List of Tables



# 1 Introduction

## 1.1 Overview

Automating speech and dialog is generally regarded as one of the more difficult challenges to solve in Machine Learning, especially in the long term. As we know Language is a unique

marker of intelligence in humans, in fact, theorists have suggested that crucial evolutionary developments that led to the burst in creativity and productivity in the Neolithic era were the emergence of human language [1]. Thus, it seems our very notion of intelligence is deeply connected to language and speech. Alan Turing, in his *Imitation Game* [2] first attempted to define the standard of intelligence for a machine by testing whether a human participant was able to successfully differentiate between a conversation with it from a human conversation. It was he who first suggested that any sufficiently intelligent machine could simulate the process of learning of an animal or even a human child. This idea is central to our approach of using Reinforcement Learning to build a dialog system. Ultimately, the goal is to simulate the inception of a truly intelligent conversation machine, which, according to Ray Kurzweil [3] is only a matter of time.

Formally, Dialog Systems are computer programs designed with the purpose of conversing with humans. These systems are generally built with a pipeline like structure in order to understand user utterances and have coherent and meaningful conversations. The primary component of a Dialog System is the Dialog Manager. This component decides what to say next to the user. The process of *deciding* can involve external queries to knowledge resources such as databases and APIs and estimating the various states of the dialog. Once we have a Dialog Manager, we have a plethora of options to communicate with the end user such as speech, text or gestures. One of the earliest successful Dialog Systems was created at the MIT AI Laboratory by Joseph Weizenbaum in 1966. This early program was called ELIZA [4] and used large extended chains of simple word re-orderings to produce dialog which could imitate the language and behavior of a psychotherapist. With recent advancements in theoretical and practical applications of AI and Computing, we can confidently say that natural, meaningful and useful conversations with machines is now fully plausible. In this thesis we focus on text-based Dialog Systems, conventionally known as *chatbots*. Chatbots can be broadly categorized into two groups depending on the nature of the dialog. These are *Chat-Oriented* or *Open Domain* chatbots and *Goal/Task-Oriented* or *Closed Domain* chatbots. The former type aims to generally converse with users providing reasonably contextually aware responses where the chatbot is expected to talk about anything i.e. it is an open domain setting. Goal-Oriented chatbots on the other hand are designed to help users achieve a certain task e.g. Making a restaurant reservation, rather than having a general conversation. Therefore, Goal-Oriented chatbots have narrower conversations limited a particular task or domain setting. In this work we focus on designing Goal Oriented chatbots to help as an assistant during sales calls.

## 1.2 Motivation

Since its inception, humanity has been leveraging technology to augment its workers with the goal of maximizing efficiency and productivity. This precept is adopted by botconnect, a startup based in Berlin that works on applying Machine Learning algorithms to automate processes in large enterprises to help empower their users with increased speed, efficiency and productivity. One field where such a project is highly valuable is Enterprise Sales.

Creating individual value for customers is essential for lead generation and customer retention and in turn successful sales, which all depend on a fast decision-making process by the human sales agent. Massive amounts of data that is stored in multiple complex systems such as Distributed Databases and Customer Relationship Management (CRM) Systems make this problem increasingly difficult to solve as the organization grows. In order to generate value for a present or prospective customer, a sales-agent often has to decide what to suggest to a customer and how to do it. The varied sources, sizes and types of data often make it inefficient for the sales-agent to analyze while simultaneously trying to interpret customer behavior and preferences in a short span of time such as a phone call. Botconnect's recommendation engine, built applying Deep Learning with more traditional filtering techniques [5], can model customer preferences once trained on enterprise customer-sales data. The service is now being applied to large enterprises primarily by customer and sales agents whose daily job involves a high amount of interaction with customers and potential customers. When queried with a customers identity, the service uses its trained model to generate a ranked list of recommended products for the customer, which the sales agent then uses to create offers during their interactions. Even though this solves the problem of having to dive through data manually for recommendations, it still leaves open the issue of having to come up with a reasonably good dialog strategy to successfully convert the conversation to a sale.

The entire cycle of a sales call, from greeting to making a sale is what we propose to automate in a manner that is clear, interactive and effortless for the agent to use [6]. We aim to accomplish this by having a dialog [7] between the sales-agent and a dialog system that augments the powers of the agent. Thus we propose a second dialog between the agent and sales bot which would occur simultaneously as the agent interacts with the customer. This Dialog System is built using best practices in handling sales dialogues and trained with a large amount of agent-customer interactions with the help of a simulated agent to come up with a good dialog strategy for the current situation. This Dialog System combined the recommendation service provides auxiliary support to the agent, augmenting their capabilities and maximizing the chance of a successful sale.

Classic Spoken Dialog Systems are in general formed by components such as a Dialog manager, modules to understand and generate spoken language as well as a speech synthe-

sizer for the audio and an analyzer for text-to-speech. In our case we focus on a text based dialog system, this is a simpler system consisting only of a User Simulator, a Dialog Manager and a chatbot based User Interface. Thus the primary motivation and aim of this project is to design and implement a Dialog Manager trained with Deep Reinforcement Learning Techniques that can learn a good internal dialog policy based on interactions with a Simulated User and with Input from the recommendation Service. This policy is a sales dialog strategy that the system uses to help the agent make successful sales. This learned policy is then used by the system to determine its response at each dialog turn.

## 1.3 State of the Art

In the past three decades the design of Goal Oriented systems have been under constant development and has changed drastically with the introduction of new techniques such as deep learning. Due to these changes there have been several new frameworks proposed for Dialog Management. One of the first commercially successful Goal oriented systems was JUPITER [8]. Created in the 2000s, this system was designed with a hand-crafted approach to modeling dialog states with the purpose of helping users retrieve worldwide weather forecast information via dialog over the telephone. Another milestone in this field was the introduction of the Information State modeling paradigm [9]. In this work the authors proposed a complete framework for Dialog State Tracking. Recent important attempts in creating such machines is documented below. These systems can broadly be categorized as following.

### 1.3.1 Dialog Policy trained with Reinforcement Learning

Applying Reinforcement Learning to Dialog Systems has recently seen an explosion in research activities with the arrival of Deep Learning coinciding with wide availability of cheap and powerful computing resources. In RL typically the main objective is to learn a good approximation of a function mapping the states of an environment to actions and deep neural networks have been shown to be excellent approximators for this purpose. In this field the concept of a Markov Decision Process (MDP) is at the core. The details of an MDP and how to design and solve it are discussed in later chapters. An MDP is a mathematical framework that facilitates the learning of an optimal mapping between situations that arise in a dialog (otherwise known as states of a system) and actions(or system responses)[10]. Levin, Pieraccini and Eckert back in 1998 were the first to formally compose the problem of dialog management as an MDP [11] and in turn, laid the foundation for the future.

Once we have the MDP of a dialog, it is simply a matter of applying one of the many Reinforcement Learning Algorithms to learn the most optimal mappings between states and

actions and thus select the best action of a given state. This was the focus of the work done in this field in the next decade [12][13][14]. This however solved only the problem of dialog policy estimation. But to build a complete dialog system, as research in the past decade has shown, a complex structure with multiple components that are able to comprehend human input and extract the state of the dialog, select an action or dialog act as a response to this state and finally translate this into human language. Formally these components are known as a Natural Language Understanding Unit (NLU) and a Natural Language Generation Unit, Dialog State Tracker and Dialog Policy Learner. A novel application of RL using this architecture for the purpose of Information Retrieval was shown in [15], where the authors built a dialog system to help users search knowledge bases for movies with simple queries. In the following year Cuayhuitl designed and implemented a robust dialog system for the restaurant reservation and booking domain [16], where the author successfully uses simple noisy text input from the user and the previous history of the dialog to represent a so-called *belief-state* that is an estimation of the true dialog state. Recently, the authors in [17] designed a complex dialog system to work in the movie booking domain. This robust system worked with raw user input and thus used a NLU to process inputs and build an internal belief state representing its estimate of the true dialog state. The NLU they used is a commonly one that is proposed in [18]. The NLG unit for their Dialog System was implemented from [19]. Both the NLU and the NLG use Long Short-Term memory units which form a type of Recurrent Neural Network [20]. For training the dialog policy the authors also used a Simulated User with a hidden agenda and goal [21][22]. The dialog policy was trained using Deep Q Networks [23].

### 1.3.2 Dialog Policy trained with End-to-End Supervised Learning

Even a decade ago the use of deep neural networks was restricted to the academia and veterans of the machine learning industry with access to a large amount of resources. Now, with the arrival of deep learning and consequent theoretical breakthroughs, wide availability of cheap computing resource companies such as Google building tools like Tensorflow[24] making it easily accessible to anybody with a computer. It is being applied in every possible field of human endeavor. This also means that neural networks have also been used to design dialog systems. In this case, it is not in the sense of representing a policy or an NLU, but completely from scratch. It involves coming up with a neural network based dialog system that once trained, does everything from understanding the user input semantics, deciding the state and converting the semantics of the desired output to speech or text. This is what is meant by End-to-End. One pioneer in this field was Sutskever, who in 2014 proposed the *Sutskever Model* [25]. The author used recurrent neural networks(RNN) to first learn sequences of user and system responses from a large amount of training data. This model

has also come to be known as the *encoder-decoder model*. The architecture is made up of two separate components built with RNNs: the first analyses the input sequence and encodes it into a feature vector, the other, then decodes the feature vector to output the predicted semantics of the sequence. Using this, the dialog manager is able to predict the latent dialog state. The disadvantage in this case, is the large amount of time and resources needed for training a large neural network and a painstaking preprocessing, labeling and restructuring of the training data.

RNNs are important in this field of dialog systems and are used widely. This is mainly due to their ability to perform very well on sequence and temporal processing tasks [26]. This is because RNNs have a feedback loop that functions as a very small memory storage and allows it to capture time based information about the data flowing through it by rolling out the time layers as individual feed forward layers. This feedback loop can also become arbitrarily large and lead to the *Vanishing Gradient Problem*[27] which can be overcome by using LSTMs[20]. Another landmark project was proposed by Bordes and Weston, who recently used the *Sutskever Model* but replaced RNNs with End-to-End Memory Networks[28]. They are a modified form of RNN combined with external memory that have the ability to manage very long term dependencies, forget unimportant information and can be trained end-to-end. This was a comprehensive goal-oriented chatbot and built using the BabI tasks[29] for the domain of restaurant reservation[30]. This advanced system demonstrates powerful abilities such as being able to interpret, remember and update past user preferences and decisions.

### 1.3.3 Dialog Policy trained with a Combined Approach

One of the primary disadvantages in using reinforcement learning to learn sophisticated dialog is that it involves the training of not just the dialog policy but auxiliary components such as the NLU and NLG. This is very compute intensive and is difficult to optimize the model. Further, this optimization is typically often be done on simulated users with developer designed user responses because using real users is often impractical and costly, therefore the trained model may not represent the nuances and sophistication when dealing with real user responses. Although some of these could be overcome with the use of data driven user simulation, this approach is generally less data intensive. On the other hand, supervised learning typically requires a very large amount of training data. This training data must have sequences of real human to human interactions from which the system could learn. For goal oriented chatbots, which are often designed for a specific domain, this kind of training data is seldom available. To overcome these limitations, a third branch of dialog system optimization was born. This method, also known as a hybrid approach to dialog policy optimization. It combines both of the above approaches to arrive at dialog systems trained with the sophistication and nuance of real human dialog along with the granular component

training offered by RL. In their work to produce a robust dialog system for the domain of restaurant information and booking, the authors in [31] propose a multi-phase approach to training the dialog policy. The design begins with the building of a *Policy Network*, which is a neural network with a single hidden layer. The output of this layer is the actions that the policy can take and the input is the belief state given to it by the DST. Once they built this, the authors used the Wizard-Of-Oz corpus to train the policy net in a fully supervised fashion, this was phase one. In the second phase, the authors then used a variant[32] of a RL based algorithm known as Policy Gradients [33]. This was combined with the concept of Experience Replay[34]. This process reuses previous experiences in a supervised manner to train the policy net to maximize the expected cumulative reward and further explore the dialog space. Recently, another such system was introduced by Williams, Asadi and Zweig. They proposed the concept of Hybrid Code Networks[35], which also address the drawbacks of the above approaches. The authors, again describe two phases of learning for the dialog policy. In the first, they train an RNN with domain specific, hand-crafted feature templates comprised of system actions. Once the model is trained, they then put the dialog system online and use a vanilla policy gradient[33] approach to fine-tune the model with real user interactions.

# 2 Dialog Systems

## 2.1 Overview

In this section we want to give the reader a brief introduction to the semantics, components and architecture commonly found in the field of dialog systems. From literature, it can be seen that depending on the type of application domain, user utterances and required system responses vary significantly. The two most commonly found varieties are speech based dialog systems, known as spoken dialog systems and text based dialog systems, known as chatbots. As we know, these systems can be further grouped into Goal Oriented (GO) or Open Domain chatbots. In this work we primarily focus on Text Based GO Dialog Systems. Here, it is a good idea to discuss what exactly we mean by dialog and some other assumptions that we make in the process of designing a Dialog Manager. A dialog, in our case, is a process of sequential and turn-based exchange of information between only two participants, using only a word or a string of words (a dialog act) at each turn. Another assumption we make is that the above described dialog can, in principle, be represented as a Markov Decision Process [11]. This is further elaborated in Section 3.3. Further in this chapter we elaborate on the general architecture of a dialog system followed by focusing on prospects of Dialog Management and User Simulation and end with some popular applications of such systems.

## 2.2 System Architecture

It is well known and can be seen from the literature [36] that there is no standard architecture for designing dialog systems, but in general they can be said to be made up of two distinct components. The first is an internal dialog management unit which, depending on the approach used (RL or Supervised), may or may not consist of components to process and generate natural language. But this unit must include a dialog policy learner and a dialog state tracker, although these could be the same component, i.e. a single neural network. Another component is the user or a simulated representation of a user for the system to interact with. This idea was first conceived by Pieraccini [37] and most systems follow the described pipeline based system. The pipeline architecture for GO text based dialog systems can be seen in Figure 1 along with the flow of information. The dialog starts with a user utterance, this is then processed by a Natural Language Understanding (NLU) Component that extracts the perceived semantics of the input. These semantics represent the intent of the user and the current context which includes the previous history of the dialog. The user utterance semantics then flow into the central decision-making module called the Dialog Manager(DM). The DM interprets the input from the NLU Component in the context of the entire history of the conversation and estimates the current dialog state. This process

of tracking the state according to user input is done by the Dialog State Tracker(DST) component of the DM [38]. The other sub-component of the DM is the Dialog Policy, which takes the state from the DST as input and uses it to select the most appropriate action or system response. The semantics of this chosen action is then fed into the NLG Unit, which generates an appropriate sentence for the end user. This is the entire process of the commonly used Pipeline Architecture. In the next sections, we go into further details of the Dialog Management and User Simulation component from the perspective of the current working requirements.

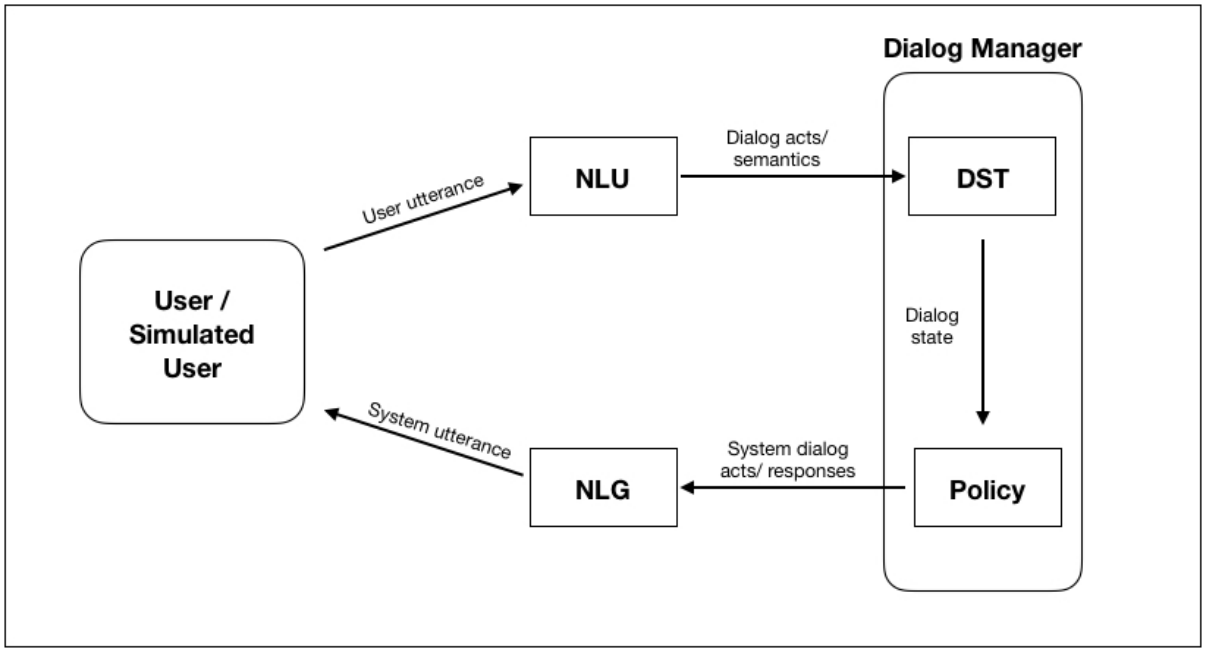


Figure 1: Text Based Dialog System Architecture

## 2.3 Dialog Management

The DM can generally be seen as the central decision making module of the entire system. This central component is responsible for the tracking and estimation of the current dialog state as well as choosing the best response for the user, based on the dialog state. The former is performed by a sub-component known as the Dialog State Tracker (DST)

### 2.3.1 Dialog State Tracking (DST)

Dialog is a sequence of turn-based interactions. This implies that the probability of a specific utterance is conditionally dependent on the sequence of previous utterances or the dialog history and the uncertainty in the mapping between the perceived input signal to its semantics. Thus, to infer the current dialog state, it is important for us to consider the previous sequence of utterances and states of the system. We can extend this to what is known as a Markovian representation of the dialog [10][11], where the dialog state is fully described by the previous history of dialog states and utterances. This is a simplification of the dialog, where we have to reduce the entire state space to a unique finite set with discrete states for every possible conversation sequence. This set of discrete states can also be represented as nodes in a conversation graph. A dialog sequence is a specific path through these nodes. This process of state factorization depends on how closely the observable raw user input represents the true observable state of the dialog and it is the main function of the DST. This discrepancy can arise practically due to various factors such as noise, ambiguous user responses or changes in user goals. A few commonly found methods of state factorization for DSTs are discussed below.

#### 2.3.1.1 Information State DST

These trackers are also known as rule-based and are generally found in finite state systems, where there exists a state transition network whose nodes represent different dialog states. Here the next state is a function of the latest user response and the previous state. Thus the dialog state is managed by traversing this finite state network [39]. Larson and Traum in [9] made major contributions to this are of research with the introduction of the *Information State Paradigm*, which formalized the representation of a state as a discrete random variable representing its position in the network to build a complex dialog system. This approach is convenient since no training data is required, as the state to state transition probabilities, domain information and dialog actions that is to be learnt from the dialog data are built into the hand crafted state transition network. However, a drawback is that the system does not sufficiently consider the intricacies present in natural language.

#### 2.3.1.2 Generative DST

Other attempts to solve the discrepancy problem involved the use of so called *generative models*. Such models are commonly found in the field of Statistical Modeling and consist of using a Dynamic Bayesian Network (DBN) to estimate the joint probability distributions between the current observation of the user utterance and the dialog state [40]. Here the

tracker works by representing user utterance $u$ and the true dialog state $s$ as unobserved noisy random variables. Bayes Theorem is then applied to get a distribution over $s$ dependent on the system action $a$ and the noisy user utterance $u$ and the previous state. Thus, with the DBN we can then infer a probability distribution over the state and user utterance, i.e. if $a'$ is the current system action, and $s$ is the previous dialog state. Then, the model tries to estimate the joint distribution $s' = P(a', s)$, where $a'$ is again dependent on the previous user utterance $u$. Early methods used an information state approach with discrete dialog states[41] which became intractable with very large state spaces. To overcome these, a number of methods were put forth to approximate the dialog state [42][43]. This also allowed the developer to optimize using domain specific feature parameters, which could be trained with dialog data.

#### 2.3.1.3 Discriminative DST

Generative models, though advantageous and relatively easy to implement, have a few drawbacks. They use a BDN, which tries to model all complex relationships present in the input features and hence they face some difficulty in exploiting useful contextual parts of the dialog history as they are not included in the feature parameters of the DBN. This and some other important disadvantages can be found in [44]. Discriminative models try to overcome these drawbacks by trying to directly model a distribution over the dialog states, given input features which are semantic representations of the dialog history. The parameters of this modeling are generally optimized using machine learning techniques such as using a Conditional Random Field (CRF). A linear-chain CRF could be utilized for learning the distribution and tracking the sequential dialog as shown by the authors of [45]. Recently, RNNs have shown to be very effective in dealing with sequential user input to dialog state approximation [46][47]. It can accurately model a conditional distribution given even a noisy semantic dialog history. This is made possible due its ability to perform well on tasks involving sequential data. This approach is also known as a word-based DST and can be applied directly to noisy user input semantics and uses the n-gram approach combined with slot-value pairs with generic or *delexicalized* features to come up with the semantics of user input after which the RNN is applied to distinguish different states[48]. This DST has a few major advantages such as avoiding the need for designing hand-crafted features for the input while also omitting the need for a separate NLU component and hence is widely used in the field.

#### 2.3.1.4 Our Approach

In this work we use a Information State model for the DST Component. In our domain, the dialog is required to be structured as a conversational graph with nodes representing various states and the connections between them representing actions. This means that all our user responses are constrained by the given generic response actions, for e.g. *yes*, *no*, *not sure*. Since all the nodes are unique states, the path to any one state is unique. This is the same as the State Transition Network described in [39]. Therefore our model uses the Information State Paradigm i.e. employs a full MDP [9]. Here, any state with the latest user action contains all the information required to infer the next state. This representation has a few key advantages in our case. The first being that it allows us to model the dialog as a full MDP, which means we can easily apply RL Planning and Control algorithms to solve it [39]. The second key advantage is that with this representation we do not need an explicit NLU, as there exists a one to one mapping between the observable feature vectors (user input and observed dialog state) and the true dialog state. Since we use a deep neural network to model the policy, using this representation lets us easily generalize between similar states in large state-action spaces. This approach to DST design also requires very less data to train, which lets us effectively add, change and test domain specific features quickly and without changes to the dialog system.

### 2.3.2 Dialog Policy

Previously we saw how given the history of the dialog we could infer the current state. This state is representational of the current context of the conversation, previous user and system responses and in some cases the user's needs and intentions. Intuitively, the next step is to choose how to respond to the users latest utterance with the most appropriate dialog act. This is the job of the dialog policy, the *brain* of any dialog system. The behavior of this module is of crucial importance to the success of the entire system as it plays an intermediary role between the system and user, thus playing a role in the quality of the user experience. This behavior is controlled by the semantic mapping between the dialog states and system actions. This mapping in essence, has to be learnt and the best policy would be one where the user reaches his goal in the least amount of time, which can be achieved efficiently using Reinforcement Learning.

In RL we work by modeling the abstract way animals behave in nature. This means that we have some agent acting in an environment. The agent can observe a certain state at a time denoted by $s_t$. The agent, depending upon the observed state then chooses an action $a_t$ to perform in the environment, this chosen action is based on a certain policy it has learned from past experience. The nature of the experience whether positive or negative, is determined by a reward $r_t$. Finally, the action taken changes the environment which is now in state $s_{t+1}$. In this manner, the job of the DM is to learn the most optimal policy for

orchestrating efficient and useful dialog. This means, in terms of RL, the dialog policy with the maximum cumulative future reward received. Thus, we make the agent maximize the cumulative future reward. The theory of RL [39] offers many methods to achieve this[49], which will be discussed further in the coming chapter.

For management of the dialog interaction, one approach, as discussed before is to represent the dialog using the information state technique as a dialog tree. Other approaches generally involve the use of probabilistic models to model the noise in the observations and map them onto a distribution over actions. These are generally used in systems that require inter-domain functionality[50]. The advantage of using these techniques is that whichever method we use to model the DST, the function approximation technique we use to represent the policy can remain the same. This allows us flexibility in testing different hyper-parameters for optimizing the policy without modification to the rest of the system.

#### 2.3.2.1 Our Approach

In this work when modeling the dialog policy using RL, the ultimate goal of the policy is to predict the best sequence of system actions given the current state (from user input) to maximize the future reward. Here, the reward is determined by the success of the dialog, whether the dialog, in this instance, actually helped the sales-agent to make a sale. In our case the first inference from the DST, directly represents the dialog state input to the policy, hence the problem can be represented by an appropriate MDP [11]. This is advantageous as we can feed the output of the DST, which represents the dialog state, directly into the policy net without further modifications.

## 2.4 User Simulation

In the process of designing dialog systems, it feels intuitive that we should use real users for testing, training and optimization. This would give the best benchmark for its quality. While it is true that this is the most ideal scenario, unfortunately for practical, real-world use cases the learning of a good policy often required thousands of dialog iterations. This makes learning from only real user interaction highly impractical mainly due to large financial, legal and time costs associated to employing human participants. Another common situation that arises when designing dialog systems, is the system designer might want to try out different representations of the dialog state. This is a problem when using a corpus of dialog data, as the entire dataset often has to be annotated again for training, which is generally a long and time-consuming process. A good solution to overcome these issues is to use a simulated user or a dialog simulation. Once built, these simulations can then be used to produce a potentially unlimited amount of training episodes or conversations, thus the dialog policy

has a lot of data to learn from. Further, they can also be modified easily to experiment with several different action and state factorizations. Simulation of dialog according to literature, can be performed at different levels of detail. Generally these levels can be classified as raw speech signals or at the acoustic level [51], at the word level and at the dialog act or Intention level. Generally, this simulation is done at the dialog act level to train goal oriented dialog systems. This is due to the fact that dialog acts can be easily abstracted into its semantics representing the dialog state, which in turn makes it very efficient to train RL agents. There exist many different techniques of simulating a user for dialog policy optimization [52]. We discuss a few important and relevant techniques below

### 2.4.1 Rule Based Simulation

These techniques are straightforward as they are simply a set of rules that dictate how the simulated user acts. This type of deterministic behavior can be inferred from the correctly annotated dialog corpus or explicitly hard coded by a developer. Some previous work that successfully applied explicit rule based models, to generate simulated dialog at a low level of granularity, can be seen in [53][54]. Here the authors used a heuristic model to generate responses at word and intention level respectively.

### 2.4.2 Probabilistic Model Based Simulation

A crucial drawback in the rule based methods described above, is that they are inflexible and rigid with respect to simulating the uncertainty and noise generally displayed by real users. The hard coded rules generally tend to produce a high amount of variance between real and simulated user behavior and this results in a sub-optimal dialog policy with a lot of room for improvement. To overcome some of these drawbacks, a probabilistic model was proposed. In these models we would use a probability distribution to select the user responses. Early versions of this was called the bi-gram model and it was represented by the conditional probability $P$ of user utterance $u_t$, given that the current state of the system was $s_t$ is given by $P\big(u_t|s_t\big)$. This model can be widely found in literature [55][56][57]. This model was later extended to n-grams with $n$ varying from two to five in [58], further the authors also proposed producing simulation based on a combination of linear feature vectors representing the user dialog state $s$ with user action $a$ with the probability $P\big(a|s\big)$. Other attempts in this field include the use of Hidden Markov Models (HMM) in [59] where the probability of an utterance is given by $P\big(u_t|q_t, s_t\big)$ where $q_t$ are hidden states in the weights of the HMM and the current state of the system is $s_t$. Another extension of this model was to add the concept of a user goal so that, user responses can now follow a user goal. This was proposed in [60] to introduce a pre-defined user goal $g$, which is generally represented

as a collection of slot-value pairs. This was later modified to include a user agenda and a goal with the hidden user agenda represented by HMMs [61].

### 2.4.3 Our Approach

In this work, we use a vanilla probabilistic simulation model that is similar to the bi-gram model, but also combined with the the finite state-network representation of states based roughly on [60]. In practice these distributions are spread across state-action combinations. This gets converted to a collection of state-state transition probabilities, when producing simulated responses.

## 2.5 Applications

Goal Oriented text based dialog systems are widespread and used in many practical applications and industries. These dialog systems can be found everywhere from call centers to answer simple questions [62], tools to navigate the web [63], personalized interactive games[64] to the modern, intelligent personal assistants that we all know and love such as Amazon's Alexa[2], Google Assistant[3] and Apple's Siri[4]. One important legacy system that laid the foundation for the current state of the field was known as ELVIS (EmaiL Voice Interactive System). It was designed to help users access their email over the phone[65]. This system used Q-learning for training its dialog policy using interactions with real users. Using the foundation laid by the authors of ELVIS, others built applications to help users access weather information [8] and to help make travel and restaurant bookings [66]. Other important legacy systems that needs to be mentioned here are TRAIN[67] and TRIPS[68] systems developed at the University of Rochester. TRAINS is a planning assistant for managing freight processes. TRIPS was another intelligent problem solving agent that collaborated with humans to help make evacuation plans for an island.

[2] developer.amazon.com/alexa
[3] assistant.google.com/
[4] apple.com/siri/

# 3 Technical Background

## 3.1 Overview

As we have seen, the basic premise behind the dialog systems we discussed is that dialog can be considered to be a decision-making task and that the dialog policy is at the core of controlling these decisions. A series of these decisions, when made to achieve a certain goal is called a dialog strategy. Thus these dialog strategies regulate the behavior of dialog systems. From Section 3.3 we have seen that various approaches exist to design these strategies. The simplest being a rule-based or information-state approach, generative and discriminative approaches. While all these avenues have their advantages and drawbacks in various domains of application they do not automatically optimize the dialog strategy, but simply follow it. Further, it is known that there is no standard principles guiding the design of these strategies. All these issues motivated the authors in [69] to cast the problem of developing dialog strategies as a *optimization problem*. This concept dictates that if we have a collection of dialog states and a set of system actions, we can produce an objective loss function that describes the various nuances in the quality of the dialog. Now we can define an optimal dialog strategy as one which minimizes the loss function by selecting actions that produce the least loss for every possible dialog state. Thus, the loss function ascribes a certain price for taking a particular action in a particular state. The entire process can be viewed as a finite (since our dialog ends after a certain goal is reached) sequence of states, chosen actions and costs associated with taking that action $\{s_0, a_0, c_1, s_1, a_1..., c_{t-1}, s_t\}$. With this representation we can optimize action selection to come up with the best strategy. The authors in [69][11] were early pioneers in applying the Reinforcement Learning to solve this problem.

In this chapter we further explore the application of RL paradigms to dialog management, introduce the mathematical formulation of decision making processes and describe approaches to optimize such processes.

## 3.2 Reinforcement Learning (RL)

Reinforcement Learning, is a computational framework that helps in developing programs that can learn by interacting with an environment. RL algorithms form a subset of all Machine Learning algorithms. Here at any point in time an *agent* makes an *observation* of the the *environment*. Using this observation it decides what *action* to perform and consequently receives some feedback about the quality of the decision made in the form of some *reward*. This process is repeated until some terminal state and can be seen in Figure 2. Further, the goal of this agent is to identify the sequence of actions it must take to maximize the total expected reward. With this representation we can intuitively model good dialog policy with

the RL paradigm as the agent can learn how to make good replies(take actions) by interacting with an environment(user utterances) and observing its state which is the current dialog state. Technically, this paradigm generally consists of four components a reward function, a policy, a value function and a model of the environment(not essential). The reward function can be considered to have as inputs the current state and the action chosen, the function then returns the reward for the given state-action pair. This reward is generally a discrete numerical value and represents the quality of the immediately selected action. As discussed earlier the job of the RL agent is to maximize this rewards returned from this function for the long term considering current and all consequent next states.

A policy defines the strategy the agent uses to select actions and thus represents its general behavior. In essence it is simply a mapping between states of the environment and the actions that the agent takes. A policy can be represented by simple tables, decision tress and even neural networks. The value function keeps track of what is best for the long term. The general value function can be considered to have as input the current observable state and outputs a value that represent how good is it to be in that state in the long term. This value is calculated as the total future rewards previously received after starting in that observed state. Thus the agent's strategy to select actions takes into account the value associated to taking an action that leads to a certain state and not just the numerical reward. The model forms the fourth component and is optional is the RL paradigm. The model is a simulated representation of the real environment. This model is used to produce the next observable environment state given the current state and the selected action. Algorithms that use such a model is said to be "model-based" and otherwise "model-free". Another important assertion is that this environment can be represented by a Markov Decision Process (MDP), which we discuss in the following section.

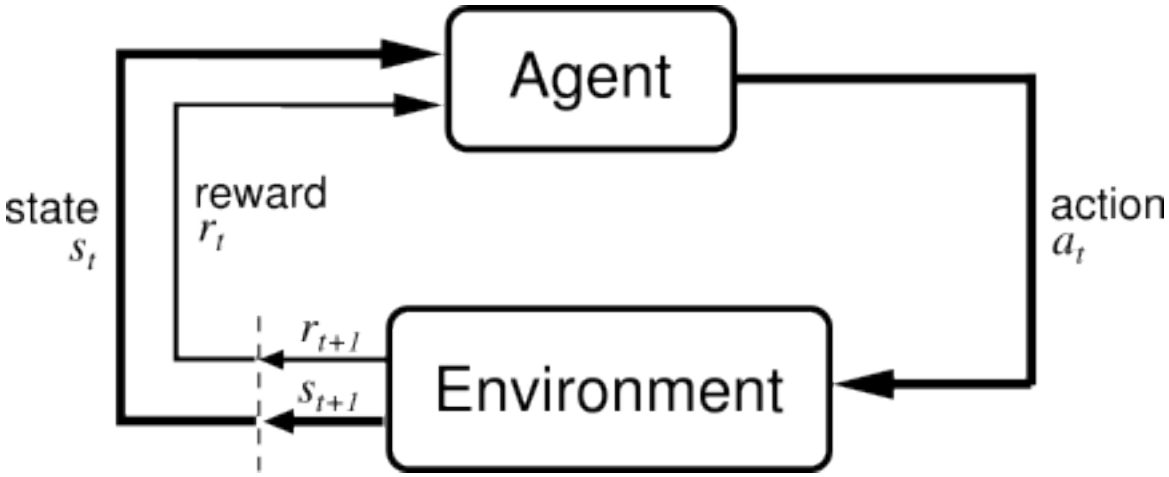


Figure 2: Agent-Environment Interface [39]

## 3.3 Markov Decision Processes (MDP)

MDPs are a mathematical framework that facilitates the learning of an optimal mapping between situations(or states) and actions[39]. A *policy*, referred to by $\pi$ is often the name given to this mapping. Policy learning often occurs through the learning of a *action-value function* or a *state-value function* or both. Formally an MDP is defined by a tuple $\{S, A, P, R, \gamma\}$.

- Here $A$ is a the discrete action space and represents the set of actions that can be performed in the environment and is given by $A = \{a_0, a_1..., a_m\}$.
- $S$ is the state space that represents the set of all possible observable states of the environment and is given by $S = \{s_0, s_1, ..., s_n\}$ and $s_t$ is the state at time step $t$. In an MDP all the states are directly observable and can be used to represent the different configurations the environment is in. In non-continuous tasks, such as dialog, each episode ends with a terminal state.
- $P$ represents the a function that performs the transition between states. Its job is to return the next state $s_{t+1}$ when we input the presently observed state $s_t$ and the action selected $a_t$ at time $t$. These transitions are modeled by a conditional distribution $P\big(s_{t+1}|s_t, a_t\big)$.
- $R$ is the reward function and it returns the immediate reward at time $t$, denoted $r_t$ for the action $a_{t-1}$ and the consequent state transition $s_{t-1} \rightarrow s_t$.
- $\gamma \in \big[0, 1\big]$ is called the discount factor that prioritizes short-term rewards.

At each time step $t$, a particular state $s_t$ characterizes the environment. The agent has to now choose an action $a_t$ according to policy, $\pi : S \rightarrow A$. Due to this interaction with the state, it changes to $s_{t+1}$ according to the transition probabilities and this change could lead to a feedback to the agent as the reward, $r_t = R\big(s_t, a_t, s_{t+1}\big)$. The goal of the agent is to then find a policy which maximizes the expected discounted cumulative reward. Simply, this quantity can be defined as:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T \tag{3.1}$$

Where $T$ is the final time-step. To deal with the case of infinite time-steps, we use the concept of *discounting*. Here, the agent tried to select actions so that the sum of the discounted rewards is maximized. The *discount rate*, $0 \leq \gamma \leq 1$ determines the *present value of future rewards*[39]

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

Further, (3.2) can be simplified as successive returns are related to each other.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \ldots \\ &= R_{t+1} + \gamma\big(R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \ldots\big) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.3}$$

Here $G_t$ is expected discounted cumulative reward and $t$ defines the current time-step. Now we can define the *value* of a state $s$ given a policy $\pi$ is the total expected return when starting in $s$ and henceforth following $\pi$.

$$v_\pi(s) = E_\pi\big[G_t | S_t = s\big] = E_\pi\left[\sum_{t+1}^{T} \gamma^{t+1} R_{t+1} \middle| S_t = s\right],\ for\ all\ s \in S, \tag{3.4}$$

where $E_\pi\big[.\big]$ is the expected value of a given state if the agent follows policy $\pi$ and $t$ is any time-step. $v_\pi$ is known as the *state-value function* for *policy* $\pi$. Likewise, we can define the value of taking an action $a$ in state $s$ given a policy $\pi$, as the expected return starting from $s$, taking action $a$ and thereafter following policy $\pi$:

$$q_\pi(s, a) = \mathbb{E}_\pi\big[G_t | S_t = s,\ A_t = a\big] = E_\pi\left[\sum_{t+1}^{T} \gamma^{t+1} R_{t+1} \middle| S_t = s,\ A_t = a\right] \tag{3.5}$$

We call the $q_\pi(s, a)$the *action-value function* for *policy* $\pi$. An important and useful property of value functions used throughout Reinforcement Learning is that they satisfy recursive

relationships similar to that of Equation 3.3. For any given policy $\pi$ and any state $s$, the following condition holds between the value of $s$ and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\big[G_t|S_t = s\big] \\ &= \mathbb{E}_\pi\big[R_{t+1} + \gamma G_{t+1}|S_t = s\big] \\ &= \sum_a \pi\big(a|s\big) \sum_{s'} \sum_r p\big(s', r|s, a\big)\Big[r + \gamma \mathbb{E}_\pi\big[G_{t+1}|S_{t+1} = s'\big]\Big] \\ &= \sum_a \pi\big(a|s\big) \sum_{s',r} p\big(s', r|s, a\big)\Big[r + \gamma v_\pi\big(s'\big)\Big], \text{for all s} \in \text{S} \end{aligned} \tag{3.6}$$

Equation 3.6 is called the *Bellman equation* for $v_\pi$. It shows the relationship between the value of a state with respect to the value of its successor states. Similarly the Bellman Equation for $q_\pi$ is given by :

$$q_\pi\big(s, a\big) = \sum_{s' \in S} p\big(s', r|s, a\big)\Bigg(r + \gamma \sum_{a' \in A} \pi\big(s', a'\big) q_\pi\big(s', a'\big)\Bigg), \tag{3.7}$$
$$\text{for all s} \in \text{S and a} \in \text{A}$$

### 3.3.1 Optimality

In essence solving the Reinforcement Learning task aims to find a policy that selects actions in a way that maximizes the future rewards[39]. In the case of finite MDPs, a policy $\pi$ is said to be better than or equal to a policy $\pi'$ if it's expected return is greater than or equal to that of $\pi'$ for all states. So $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. This implies that there is always at least one policy that is better than or equal to all other policies[39]. This is denoted by $\pi_*$, and it's corresponding state-value function as $v_*$. and is shared by all optimal policies. Thus

$$v_*(s) = \max_\pi v_\pi(s), \;\; for\ all\ s \in S \tag{3.8}$$

The *optimal action-value function* is also shared between optimal policies and is denoted by $q_*$

$$q_*(s, a) = \max_\pi q_\pi(s, a), \;\; for\ all\ s \in S\ and\ a \in A \tag{3.9}$$

Now for any pair of states and actions, given by $\big(s, a\big)$, equation 3.9 can give the expected return for taking action $a$ in state $s$ and henceforth following an optimal policy. So from Equations 3.3,3.4 and 3.5, we can define $q_*$ in terms of $v_*$,

$$q_*(s, a) = E\big[R_{t+1} + \gamma v_*\big(S_{t+1}\big)|S_t = s, A_t = a\big] \tag{3.10}$$

Now since $v_*$ is the value function for a policy it must satisfy the condition given by Equation 3.6. But, since it tis the optimal value function, it can be defined without reference to any specific policy. This is known as the Bellman Equation for $v_*$ or the *Bellman optimality equation*. Intuitively, it can be seen that when working with an optimal policy the return for the optimal action in that state must be the same as the value of that state.[39]

$$\begin{aligned}
v_*(s) &= \max_{a \in A(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}\big[G_t | S_t = s, A_t = a\big] \\
&= \max_a \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\big] \\
&= \max_a \mathbb{E}\big[R_{t+1} + \gamma v_*\big(S_{t+1}\big) | S_t = s, A_t = a\big] \\
&= \max_a \mathbb{E}\big[R_{t+1} + \gamma v_*\big(S_{t+1}\big) | S_t = s, A_t = a\big] \\
&= \sum_{s',r} p\big(s', r | s, a\big)\Big[r + \gamma v_*\big(s'\big)\Big]
\end{aligned} \tag{3.11}$$

The Bellman Optimality Equation for $q_*$ is given by,

$$\begin{aligned}
q_*\big(s,a\big) &= \mathbb{E}\big[R_{t+1} + \gamma \max_{a'} q_*\big(S_{t+1}, a\big) | S_t = s, A_t = a\big] \\
&= \sum_{s',r} p\big(s', r | s, a\big)\Big[r + \gamma \max_{a'} q_*\big(s', a'\big)\Big]
\end{aligned} \tag{3.12}$$

Equations 3.11 and 3.12 are the Bellman Optimality Equations for $v_*$ and $q_*$ respectively.

## 3.4 Composing Dialog Management as an MDP

Dialog Management (DM) [12] is the primary task of any dialog system. The job of the dialog manager is to predict best system action or utterance to the user provided information such as the user utterance, dialog history ,knowledge of possible systems actions and other domain-specific information or features. Collective this can be called the dialog context. Thus the DM has to take the best actions i.e. make good *decisions* based on often times incomplete and highly variant *contexts* in order to eventually achieve a *reward*. This learning of an optimal mapping between situations and actions otherwise is called a *policy*. This definition of the problem allows us to cast dialog management as a sequential decision making problem. First done by Pieraccini et. al [11] who cast the DM problem as a Markov Decision Process (MDP) [10]. Since the MDP framework comes from optimal control theory we mainly consider the

decision making process as a *control* problem. To start of, some probabilistic description of the problem use case is made which requires knowledge of some of its parameters. If some of these parameters are unknown the Reinforcement Learning(RL) paradigm [39] provides approximate solutions for optimization of policies. In this context the DM is seen as an agent which has to interact with its environment (the human user) to maximize some expected reward. This reward is ideally returned only when the user is satisfied or a domain related goal is reached.

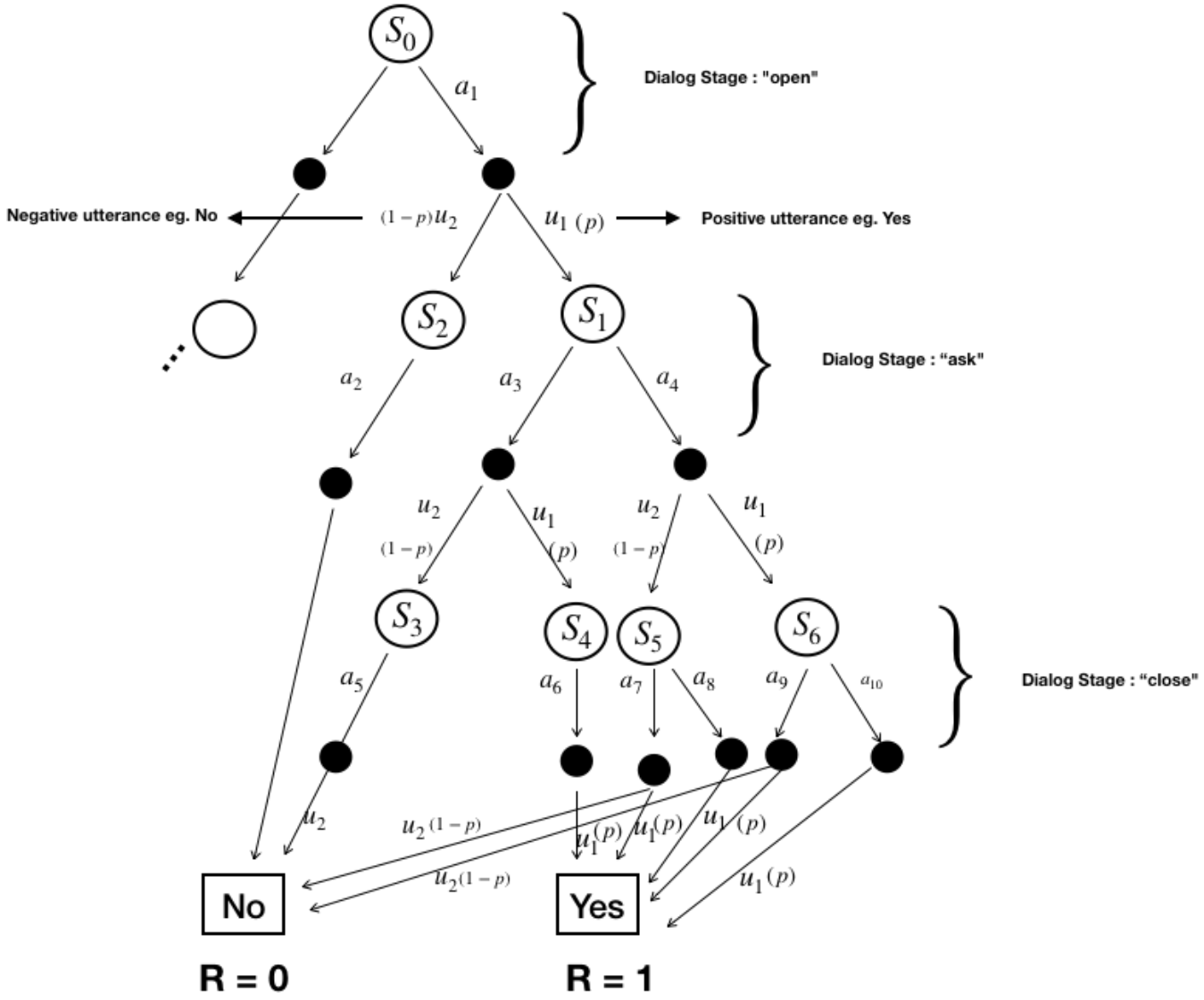


Figure 3: Dialog represented as a Markov Decision Process

Reinforcement Learning(RL) can be applied in a fairly straightforward manner to conversational models casted as MDPs. The Reinforcement Learning Agent's(RLA) Policy, observes a discrete state at time step $t$ represented by the current position of the dialog in the dialog MDP shown in Figure 3. Thus the current state represents the sequence or path given by the arcs between all previous user input utterances, dialog states and system actions. The Dialog MDP otherwise called the *conversation graph* is given by nodes that

represent the actions/utterances from the RL Agent or utterances that the user or environment inputs. When the edges originate from a dialog state such as $\{S_0, S_1...\}$ of the RL Agent, they represent the agents actions such as $\{a_0, a_1...\}$. When the edges originate from the hidden user states they are utterances that the user inputs, represented by $\{u_0, u_1...\}$. Thus each observable state of the Agent is a function of the history of the dialog and the previous user utterance. This state is also called the *dialog state*. This modeling of the dialog state is based on the *Information State Paradigm*[9], where the dialog environment is modeled as a discrete state space with a unique state number for each state otherwise known as finite state models. Our dialog system is task-oriented with the objective being the successful conversion of a service call to sell a product. Here the the objective can be summarized as making the best actions for a given dialog state to increase the chances of getting a positive response from the user.

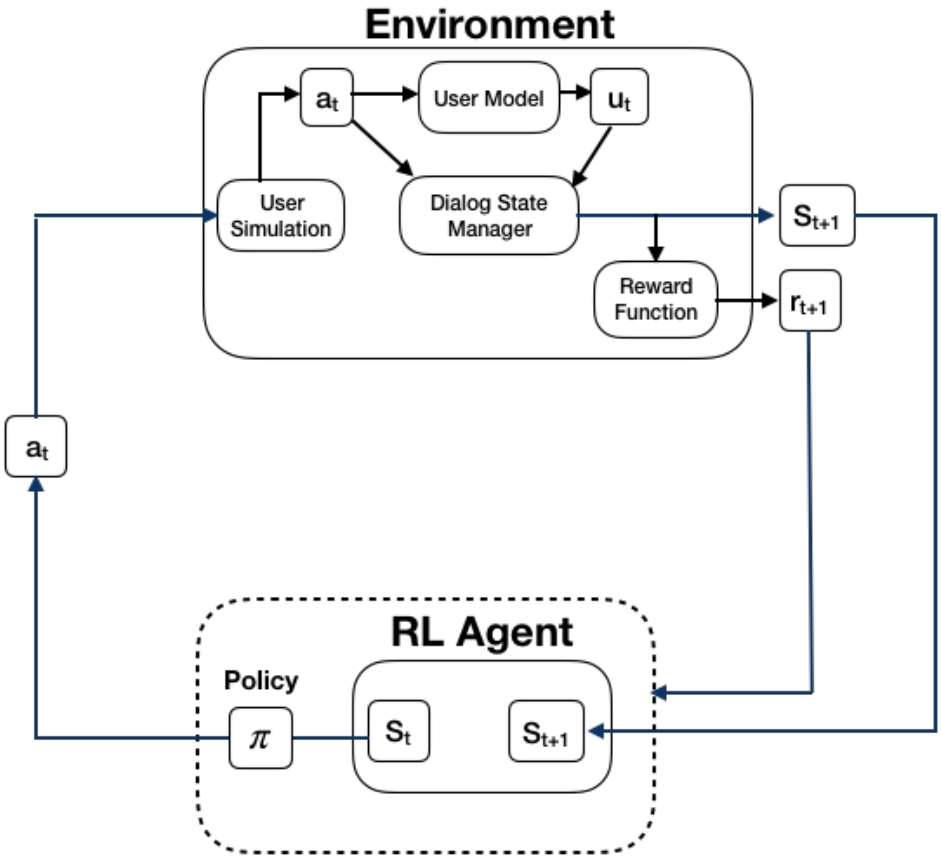


Figure 4: Agent-Environment Interaction

To carry out a Dialog, the Agent may greet the user and recommend a product and opening question for their customer. If this receives a positive response, the agent suggests a follow up question that leads to a sale, if this is positive then the system closes the sale. These acts or dialog moves form the action set. With regard to real users the transition probabilities are unknown but can be sampled by letting the user interact with the agent. However, for reasons of practicality, we first train the agent on simulated users created with transition probabilities according to broad categories of demographic data of actual users. For both real and Simulated users a reward is given to the agent upon successfully selling a product to a customer i.e. If the customer shows interests, which means in the model that certain affirmative terminal states are reached, a reward is generated, e. g. $R = 1$. In conversation with variable length we can consider to penalize the agent by assigning a small negative reward to every round, e. g. $R = -0.1$. The interaction between the Reinforcement Learning Agent (RL Agent) and the Environment, which includes the Simulated User is shown in Figure 4.

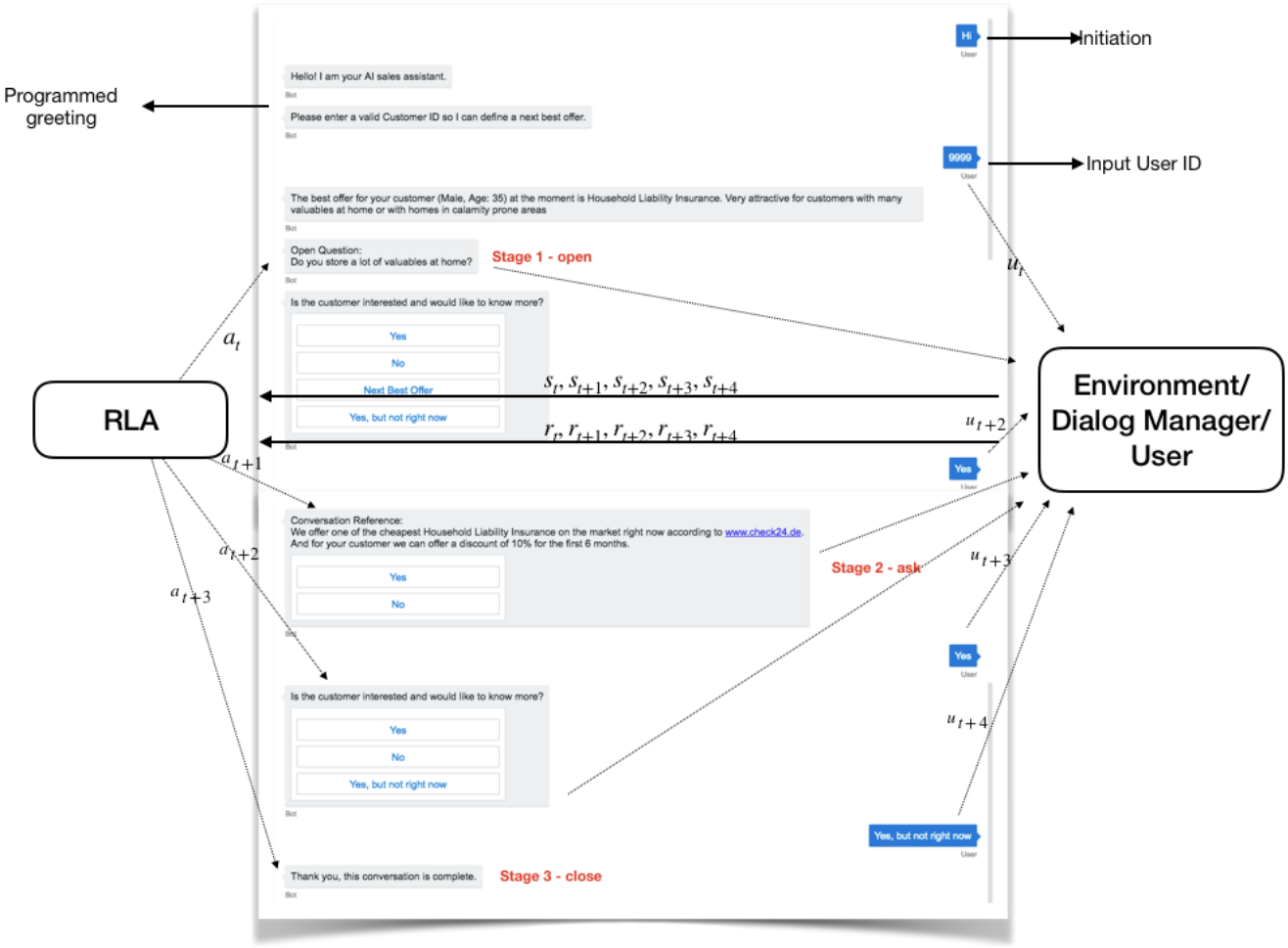


Figure 5: Agent-Environment Interaction in the form of a chatbot application

Here $S_t$ and $S_{t+1}$ represents the current and next observable state of the environment. $a_t$ is the currently chosen action from the policy $\pi$ approximated by the Reinforcement Learning Agent(RLA). $u_t$ represents the utterance/response from a simulated user to the current action(statement/question) $a_t$ from the RLA. This utterance $u_t$ along with the current action $a_t$ is fed into the Dialog State Manager(DSM) which manages the next state of the environment. The DSM then consults the reward function, which generates the reward(if any) for the current state. The Reward and the new State are then given to the RLA, which takes the new state as the current state and repeats the process. This process is expressed more clearly in the view of the chat-bot implementation in Figure 5. Here we see the User

Input Utterance (In Blue) as well as the the most recent action from the agent is fed to the environment which then returns the next state and the reward for the current step to the Agent.

## 3.5 Dialog Policy Optimization

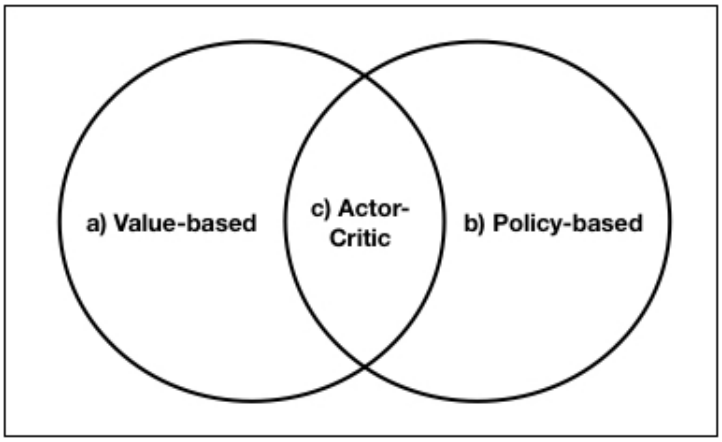


Figure 6: Dialog Policy Optimization Methods

The RL paradigm offers us many varieties and flavors of algorithms[70] to help us optimize policies and to consequently solve MDPs. Generally these can be broadly categorized as methods that :

- Iteratively improve a value function which help find a good policy by learning better state or action values. Known as Value-Based Methods.
- Directly parametrize the policy and iteratively improve it without using any value function. Known as Policy-Based Methods
- Iteratively improve both value function and policy. Known as Actor-Critic Methods

These methods are discussed in further below. In this work, we assume our dialogs to be episodic and thus considered to have a finite number of steps T. In order to find the best policy we must train our policy $\pi$ to maximize the discounted cumulative reward per episode. Thus considering Equation 3.2 with final time step as $T$ and current time step as $t$, we get,

$$R_t = \sum_{t}^{T} \gamma^t R_{t+1}, \text{while following policy } \pi$$

This is the value that has to be maximized and we first see how to do this by improving the value function.

### 3.5.1 Value Iterative Methods

As we have seen in Section 3.3.1, one can represent the expected cumulative reward at a state $s$ following a specific policy $\pi$ by using a state-value function, which is a semantic mapping between the state and the expected cumulative reward i.e. $V : s \rightarrow R$, this is represented by Equation 3.4. Similarly we have the action-value function which is semantically a mapping between state, action pairs and total expected cumulative reward i.e. $Q : \big(S, A\big) \rightarrow R$, this is represented by Equation 3.5. Now Equations 3.4 and 3.5 together with *Bellman's Optimality Condition* 3.8 [39], gives us the Optimal state-value function, represented by Equation 3.11, and the optimal action-value function, represented by Equation 3.12. Finally, if $q_*\big(s, a\big)$ is the optimal action-value function for a policy $\pi$, this implies that the optimal policy $\pi_*$ behavior is controlled by

$$\pi_*\big(s\big) = arg\max_a q_*\big(s, a\big) \tag{3.13}$$

And in the above Equation if we substitute the value of $q_*\big(s, a\big)$ with Equation 3.11, we get

$$\pi_*\big(s\big) = arg\max_a \sum_{s',r} p\big(s', r|s, a\big)\Big[r + \gamma v_*\big(s'\big)\Big] \tag{3.14}$$

Thus we can see from Equations 3.13 and 3.14 that we can arrive at the optimal policy $\pi_*$ by selecting actions that maximize the state-value function or the action-value function (Q-function). In the perspective of dialog systems this would imply that by selecting the system responses that maximizes cumulative reward for every conceivable dialog state and consequently follow a dialog policy $\pi$, we can iteratively arrive at the best possible policy $\pi_*$ that represents the best sequence of responses.

#### 3.5.1.1 Dynamic Programming

These techniques primarily depend on using the value functions given by Bellman Optimality Equations 3.11 and 3.12. Dynamic programming is a process of solving MDP's i.e. finding the optimal value function, given by Equations 3.8 and 3.9. This is achieved by recursively computing the value os states and/or actions using some known model of the MDP with the Bellman Optimality Equation as an update rule. This process is also known as basic value iteration and the value of a state-action pair can be recursively updated by

the rule [39] :

$$q_{t+1}(s,a) = \sum_{s,s' \in S} p(s', r|s, a)[r + \gamma \max_{a'} q_t(s', a')] \tag{3.15}$$

Another common DP method is known as policy iteration here we first evaluate the value of a specific policy $\pi$ using Equation 3.14. From this value computation, a policy improvement step is performed that creates a new policy by behaving *greedily* with respect to this new learned value. This recursive process repeats until the optimal policy or a policy very close to it is found[39]. Policy iterations, thus consists of two interacting processes, one that updates the value making it match the current policy(evaluation) and the other taking greedy policy actions with respect to the learned value(improvement). Generally these processes alternate with one completing before the other starts. Generalized Policy Iteration(GPI) is the process of letting policy evaluation and improvement interact independent of the other details of these processes[39] i.e. in GPI the change between evaluation and improvement is made before the current process is complete.

#### 3.5.1.2 Monte Carlo Learning

These methods play upon the ideas of GPI. As we know GPI consists of twi steps, policy evaluation and improvement. In MC techniques the value is estimated by completely rolling out the current policy on the system i.e. executing the policy based actions step by step across a full episode of learning. This process accumulates the reward returned over the entire episode with respect to the various states that the system encountered on the way, which is then used to compute the value function. This is followed by usual policy improvement by acting greedy with respect to current value function and generating a new better policy. Following these two steps repeatedly can be shown to converge to the optimal policy and value function. Though these methods are straightforward to understand and implement, in practice they require a very large number of iterations to converge and further show a high amount of variance in the estimated values.

#### 3.5.1.3 Temporal Difference(TD) Learning

The combination of the above approaches of Monte Carlo and Dynamic Programming give us TD learning. Since its invention in the 1980's by Richard Sutton [71] it has come to be a central idea in RL. Typically in TD learning like DP, the computed estimates are updated with respect to other learned estimates but without waiting for any one process to converge before starting the other. An important concept here is the TD error $\delta$. $\delta$ is defined as the

error or discrepancy between the expected return and the currently received return and thus can be represented as the update:

$$\delta_t = r_{t+1} + \gamma v\big(s_{t+1}\big) - v\big(s_t\big) \tag{3.16}$$

This the TD error ae any time $t$ is discrepancy found in the value estimate at that time. Since this error depends on the following state and reward, $\delta_t$ is not available until time $t+1$. According to Sutton's TD algorithm that incorporates policy iteration, the value function for a policy can be estimated by incrementally following the update[71] :

$$v\big(s_t\big) = v\big(s_t\big) + \alpha_t \delta_{t+1} \tag{3.17}$$

Where $\alpha_t \in (0, 1]$ is the learning rate at the time $t$ and is designed to be weighting that sets the importance of considering the old and new estimate of the value function. The learning rate controls the convergence properties of the update process describes in Equation 3.17 as the update process convergence only when the learning rate satisfies what is known as the *Robbins-Monro* convergence theorems. This is given by :

1. $\sum_{t=0}^{\infty} \alpha_t = \infty$
2. $\sum_{t=0}^{\infty} \alpha_t^2 < +\infty$
3. $\alpha_t \geq 0, \forall t$

from this it also follows that all state-action pairs would be been visited nearly infinitely many times before convergence. These principle have been applied for decades in RL with huge success. The most well known algorithms that emerge from this framework are *Q-learning* and *Sarsa*. Sarsa works *on-policy* which means that it works iteratively to estimate the action-value function of the currently followed policy by following the update:

$$q\big(s_t, a_t\big) = q\big(s_t, a_t\big) + \alpha_t \big[r_t + \gamma q\big(s_{t+1}, a_{t+1}\big) - q\big(s_t, a_t\big)\big] \tag{3.18}$$

Thus the policy evaluation step uses the TD error for computing the action-value, the complete algorithm can be found below. *Off-policy* methods are complimentary to the above in the sense that these methods attempt to learn a value function for a policy that differs from the policy that was used to choose the current actions, i.e. the target policy differs from current used policy. Watkin's in 1989 invented the famous Q-learning algorithm[72]. This algorithm is similar to Sarsa but works off-policy and uses a different update rule. Here the update rule is defined as

$$q\big(s_t, a_t\big) = q\big(s_t, a_t\big) + \alpha_t \big[r_t + \gamma \max_a q\big(s_{t+1}, a_t\big) - q\big(s_t, a_t\big)\big] \tag{3.19}$$

The complete Q-learning Algorithm can be found below:

**Algorithm 1:** SARSA

**Initialize:** $q(s,a)$ randomly for all $s \in S$ and $a \in A(s)$

**repeat**
  Observe initial state $s_1$
  Choose action $a_1$ using policy derived from $Q$ (by $\epsilon$-greedy behavior)
  **for** $t = 1,...,T$ **do**
    Take action $a_t$
    Observe reward $r_t$ and new state $s_{t+1}$
    Choose next action $a_{t+1}$ using policy derived from $Q$ (by $\epsilon$-greedy behavoir)
    Update Q function by
    $q\big(s_t, a_t\big) = q\big(s_t, a_t\big) + \alpha_t\big[r_{t+1} + \gamma q\big(s_{t+1}, a_{t+1}\big) - q\big(s_t, a_t\big)\big]$
    $s_t \leftarrow s_{t+1}$
    $a_t \leftarrow a_{t+1}$
  **end**
**until** *terminated*

#### 3.5.1.4 Sample Efficiency

All the TD techniques we saw above works by individually using each same a single time for updating the value. It is found that this learning can be accelerated by using multiple samples efficiently. A common way to achieve this is by using *Eligibility Traces*, here the trace $e\colon S \times A \to [0, \infty)$ propagates the current reward $r_t$ to update the values of recently encountered states. The update rule of the above algorithms with eligibility traces become:

$$q\big(s_t, a_t\big) = q\big(s_t, a_t\big) + \alpha_t \delta_t e_t, \forall s, a \text{ and } e_0 = 0 \tag{3.20}$$

Here every state has an additional variable associated to it known as its trace, $e_t(s,a) \in \mathbb{R}$. This trace represents the discrepancy in the value of each state-action pair at every time step and is dependent on the number of individual encounters with that state was followed by taking that action. These traces are updated at each time step using the update rule:

$$e_t(s,a) = \begin{cases} \lambda\gamma e_{t-1}\big(s,a\big) & \text{if } \big(s,a\big) \neq \big(s_t, a_t\big) \\ 1 & \text{if } \big(s,a\big) = \big(s_t, a_t\big) \end{cases} \tag{3.21}$$

Where $\lambda$ is called the *trace-decay*. Another important method which enables faster learning and greater sample efficiency is by recording the transition history e.g. $(s_t, a_t, s_{t+1}, a_{t+1}...)$ and using them at the ends of each episode to update the value, thus replaying the transitions

**Algorithm 2:** Q-Learning

**Initialize:** $q(s, a)$ randomly for all $s \in S$ and $a \in A(s)$

**repeat**
  Observe initial state $s_1$
  **for** $t$ = *1,...,T* **do**
    Choose action $a_t$ using policy derived from $Q$ (by $\epsilon$-greedy behavior)
    Take action $a_t$
    Observe reward $r_t$ and new state $s_{t+1}$
    Update Q function by
    $$q(s_t, a_t) = q(s_t, a_t) + \alpha_t \left[ r_t + \gamma \max_a q(s_{t+1}, a_t) - q(s_t, a_t) \right]$$
    $s_t \leftarrow s_{t+1}$
  **end**
**until** *terminated*

in the update rule. This is known as *Experience Replay* and has been shown to be very successful in accelerating learning[73].

#### 3.5.1.5 Function Approximation

In the previous section we just discussed how we could find the optimal state-value or action-value function and thus the optimal policy by iteratively selecting actions that maximize our estimate of these functions. The methods described above generally estimate the value functions using some static data structure such as a look-up table. This generally functions well when the environment and domain has a small and manageable state or state-action space. When this space is very large of the order $10^5$ or greater, then this solution becomes very slow and intractable due to the large number dimensions, this is commonly known as the curse of dimensionality[74]. A good solution is to employ function approximation techniques that substitutes the above mentioned look-up table with a representation such as a decision tree, linear functions or most popularly, neural networks. Neural Networks are known be excellent Universal Function Approximators [75] which means they can model any continuous function. Modeling our value functions as neural networks is advantageous in very large and continuous state-action spaces as it makes the solution tractable and we can represent the function as a linear combination of features which represent different portions of the large state space. This lets us generalize well from observed to unobserved states as any update to the parameters will also update the value function which affects the *value* of all states. Thus we can represent the complex and large state space as a set of features

$\{f_1, f_2, ..., f_n\}$, here $n$ is the number of features. Now if we consider our actions as a factored set of state-oriented allowed actions we can represent the action-value function (and any Utility function in general) as:

$$Q^a\big(s, a\big) = \theta_1^a f_1 + \cdots + \theta_n^a f_n \tag{3.22}$$

Where $a$ is each action and $\theta = \{\theta_1, ..., \theta_n\}$ are the parameters of the neural network. Now a RL agent can learn the values of $\theta$ to approximate the value-function.

### 3.5.2 Policy Iterative Methods

Previously we saw how a value function could represent the most optimal policy $\pi_*$. Another solution to finding the optimal policy is to directly parametrize the policy $\pi$ itself. These techniques aim at iteratively optimizing the policy directly. Here we work in the policy space and not in the state-action space like before. A function parameterized with respect to $\theta$ is chosen to represent the policy $\pi_\theta\big(a|s\big)$. The reward function to be maximized depends on this policy and is given by:

$$J\big(\theta\big) = \sum_{s \in S} d_\pi\big(s\big) v_\pi\big(s\big) = \sum_{s \in S} d_\pi\big(s\big) \sum_{a \in A} \pi_\theta\big(a|s\big) q_\pi\big(s, a\big) \tag{3.23}$$

Now consider traveling along a Markov Process's states forever, once all states are explored and seen multiple times, eventually, the probability of assuming any single state becomes stable and unchanged - this is called the *Stationary Probability* for $\pi_\theta$ and is represented by $d_\pi$. Now to improve the objective function we need to compute its gradient, $\nabla J\big(\theta\big)$ The Policy Gradient Theorem now gives us an expression an expression for the gradient of the objective function states that for the Objective Function $J\big(\theta\big)$, generally this is a problem as it is dependent on the action-selection behavior and the state-wise stationary distribution we spoke about earlier. Generally since the environment is unknown, this computation is intractable. Here is where the Policy Gradient Theorem comes in and gives us an expression for the gradient that *does not* depend on the derivative of the distribution $d_\pi\big(s\big)$. Thus the PG theorem shows that :

$$\nabla J\big(\theta\big) \propto \sum_{s \in S} d_\pi\big(s\big) \sum_{a \in A} q_\pi\big(s, a\big) \nabla \pi\big(a|s, \theta\big) \tag{3.24}$$

To arrive at this relation, we follow the conventions in [39] and start by considering the

partial derivative of the state-value function $v_\pi(s)$:

$$\nabla_\theta v_\pi(s) = \nabla_\theta \Big( \sum_{a \in A} \pi_\theta(a|s) q_\pi(s,a) \Big), \qquad \text{From Eq. 3.6}$$

$$= \sum_{a \in A} \Big( \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) + \pi_\theta(a|s) \nabla_\theta q_\pi(s,a) \Big),$$

Using the product rule,

$$= \sum_{a \in A} \Big( \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} p(s',r|s,a) \Big[ r + \gamma v_\pi(s') \Big] \Big), \qquad \text{From Eq. 3.11}$$

Now to simplify we consider $\gamma = 1$ and since $r$ does not depend on $\theta$ we can eliminate it. Also we know that:

$$P(s'|s,a) = \sum_r P(s',r|s,a)$$

This lets us simplify the above as

$$\nabla_\theta v_\pi(s) = \sum_{a \in A} \Big( \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) + \pi_\theta(a|s) \sum_{s'} p(s'|s,a) \nabla_\theta v_\pi(s') \Big) \tag{3.25}$$

Now if we notice the term on the LHS and the final term in the RHS, we can see that the above Equation 3.25 is recursive. This means that we can *role out* the final term to

$$\nabla_\theta v_\pi(s) = \sum_{a \in A} \Big( \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) + \pi_\theta(a|s) \sum_{s'} p(s'|s,a) \sum_{a' \in A} \Big[ \nabla_\theta \pi_\theta(a'|s') q_\pi(s',a') + \pi_\theta(a'|s') \sum_{s''} p(s''|s',a') \nabla_\theta v_\pi(s'') \Big] \Big) \qquad \text{From Eq. 3.25} \tag{3.26}$$

Now to simplify this recursive equation we generalize the probability of moving from any state $s$ to any other state $x$ in $k$ steps while following policy $\pi_\theta$ as $Pr(s \to x, k, \pi_\theta)$. This lets us rewrite the above equation in a generalized form as

$$\nabla_\theta v_\pi(s) = \sum_{x \in S} \sum_{k=0}^{\infty} Pr(s \to x, k, \pi_\theta) \sum_{a \in A} \nabla_\theta \pi_\theta(a|x) q_\pi(x,a) \tag{3.27}$$

Finally this generalized equation lets us exclude the derivative of the action value function $\nabla_\theta q_\pi(s,a)$ by plugging it back into the objective function $J(\theta)$. Considering that we start

from any random state $s_0$ we get:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta v_\pi(s_0) \\ &= \sum_s \left( \sum_{k=0}^{\infty} Pr(s_0 \to s, k, \pi_\theta) \right) \sum_a \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) \end{aligned} \tag{3.28}$$

Let $\eta(s) = \sum_{k=0}^{\infty} Pr(s_0 \to s, k, \pi_\theta)$. Now we get

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_s \eta(s) \sum_a \nabla_\theta \pi_\theta(a|s) a_\pi(s,a) \\ &= \left( \sum_s \eta(s) \right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \sum_a \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) \qquad \text{by normalizing } \eta(s)\end{aligned}$$

Now since $\sum_s \eta(s)$ is a constant and considering $d_\pi(s) = \frac{\eta(s)}{\sum_s \eta(s)}$, we get:

$$\nabla_\theta J(\theta) \propto \sum_s d_\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) \tag{3.29}$$

This is known as the Policy Gradient Theorem [39][33]. Now if we consider the *episodic* case, where each episode of training has a distinct starting and terminal state, the proportionality constant $\sum_s \eta(s)$ is defined as the average length of an episode. Thus the gradient can be further simplified as:

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \sum_s d_\pi(s) \sum_a q_\pi(s,a) \nabla_\theta \pi_\theta(a|s) \\ &= \sum_s d_\pi(s) \sum_a \pi_\theta(a|s) q_\pi(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \mathbb{E}_\pi \left[ q_\pi(s,a) \nabla_\theta \ln \pi_\theta(a|s) \right] \end{aligned} \tag{3.30}$$

This is known as the *Vanilla* Policy Gradient update and lays the theoretical foundation foundation for the various policy gradient algorithms found in Section 5

### 3.5.3 Actor-Critic Methods

In this chapter we will introduce and derive the Actor-Critic framework. These methods are known for combining both the techniques we saw above i.e. Value based optimization which is known as the Critic as intuitively it *judges* the quality of the action or *act* chosen

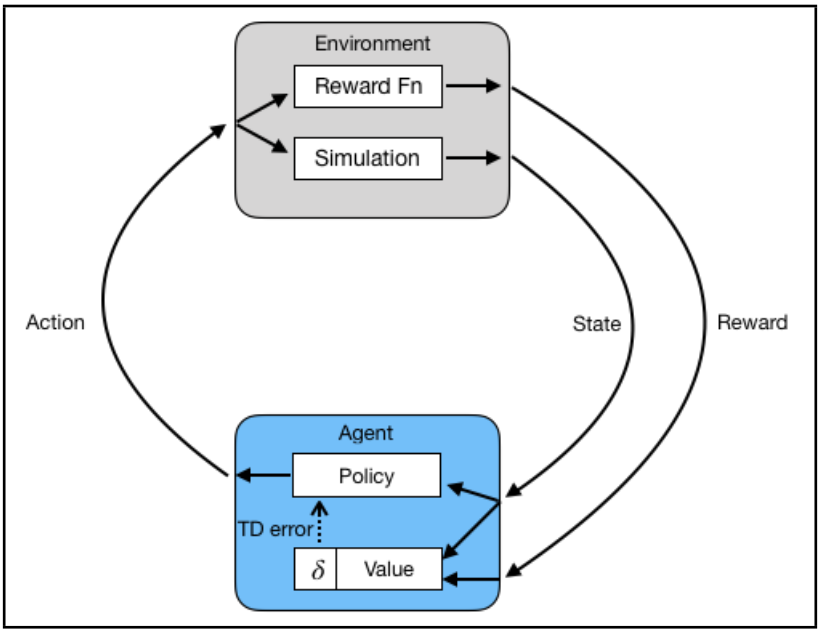


Figure 7: Actor-Critic Environment Interaction

by the actor Policy. The complete functioning of this process can be seen in Figure 7. In the previous section we saw the PG theorem[39][33] which helped us estimate the gradient of the objection function for a parameterized policy i.e. the *policy gradient*. According to this, the policy gradient for any MDP is given by:

$$\nabla_\theta J(\theta) = \sum_s d_\pi(s) \sum_a \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) \tag{3.31}$$

Here a great advantage is that there is no gradient of the steady state distribution as discussed in the previous section. This means that if we obtain training samples by following a policy $\pi_\theta$, then $\sum_a \nabla_\theta \pi_\theta(a|s) q_\pi(s,a)$ represents an unbiased estimate of $\nabla_\theta J(\theta)$ and also a gives us a relation between the policy gradient and value function or critic. The PG theorem also lets us parameterize the value function. Consider $f_w(s,a)\colon S \times A \to R$ as an approximation of the value function $q_\pi(s,a)$ where $w$ represents the weights of parameterization. Generally since function approximators such as neural networks try to minimize the discrepancy between the true value function $q_\pi(s,a)$ and its approximation $f_w(s,a)$, we get an update rule for $w$ which is of the form $\nabla w \propto \nabla_w [q_\pi(s,a) - f_w(s,a)] \propto [q_\pi(s,a) - f_w(s,a)] \nabla_w [f_w(s,a)]$. Using this in Equation 3.31 we get convergence if we successfully approximate $q_\pi(s,a)$ as:

$$\sum_s d_\pi(s) \sum_a \pi_\theta(a|s) [q_\pi(s,a) - f_w(s,a)] \nabla_w [f_w(s,a)] = 0 \tag{3.32}$$

Now when $f_w(s,a)$ satisfies the above, we get the following relation from Equation 3.30 :

$$\nabla_w[f_w(s,a)] = \nabla_\theta[\ln \pi_\theta\big(a|s\big)] \tag{3.33}$$

Now by plugging in the RHS of Equation 3.33 into Equation 3.32 we get:

$$\sum_s d_\pi\big(s\big) \sum_a \nabla_\theta \pi_\theta\big(a|s\big)[q_\pi(s,a) - f_w(s,a)] = 0 \tag{3.34}$$

Thus we can see that the error in approximating the value function is perpendicular to the policy gradient. Since the above equation is equal to 0, we can subtract Equation 3.34 from Equation 3.31 which gives :

$$\nabla_\theta J\big(\theta\big) = \sum_s d_\pi\big(s\big) \sum_a \nabla_\theta \pi_\theta\big(a|s\big) f_w\big(s,a\big) \tag{3.35}$$

Thus when the value approximation satisfies Equation 3.32 and 3.33, we get a reliable estimate of the policy gradient from Equation 3.35. Now since $f_w$ has 0 mean for any particular state i.e. invariant to state or action distribution, we have:

$$\sum_a \pi_\theta\big(a|s\big) f_w\big(s,a\big) = 0, \qquad \forall s \in S \tag{3.36}$$

By this we can think of $f_w$ as some *advantage function*: $A_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$ [76][33]. Thus we can introduce what is known as a baseline function $b$: $S \to R$[76] without changing the results of Equation 3.35. This baseline $b(s)$ can have a great influence on the performance of the gradient approximators [33]. Now if we have a parameterized baseline $b_{w_2}$ with parameter $w_2$ and a parameterized value approximation $f_{w_1}$ with parameters $w_1$ we can get the value function approximation $q_w$ as:

$$q_w = f_{w_1}\big(s,a\big) + b_{w_2}\big(s\big) \tag{3.37}$$

where $w = [w_1^T w_2^T]^T$. We will further explore this method in Section 5.3 but an important property here is that t he convergence to some policy is guaranteed given that the value function approximate follows the above conditions.

## 3.6 Evaluation and Reward Estimation

Typically evaluating large dialog systems is known to be a complex and difficult task. Primarily, this is due to the long term interactions and history between the dialog system and the human user. The fact that a typical large scale dialog system includes sub components

such as a NLU and NLG, make things even more complicated and even though these systems can be individually evaluated, the joint evaluation of the entire system is still considered to be very difficult. The easiest and most intuitive method of evaluating a dialog system is also generally the most expensive, that is using real human interaction to judge the dialog, but this too is susceptible to human generated subjective biases. In cases where we develop end-to-end open-domain dialog systems with a corpus of dialogs that the systems needs to learn to mimic, certain similarity metrics such as *word-overlap* can be used[77], but this too in known to be highly variant when compared to human evaluation of the same systems[78]. Since in this work we deal with task-oriented dialog, we can use task-completion as a straightforward and automatic measurement that makes sure the learning strategy is valid. When applying RL techniques to develop dialog systems these task-completion signals can be used as a high-level objective during learning, this is also known as the *reward* in colloquial RL terminology.

### 3.6.1 Heuristic Rewards

The reward signal plays a major role in the MDP framework as it directly defines the desired policy or behavior of the system. Generally a positive reward signal is associated with situations related to task success for e.g. in chess and Atari game the end goal is often very clear, it can be to win the game or gain the most points/score as possible. But this becomes more complicated in dialog systems as its 'win' is not as clearly defined as say, chess as it is dependent on the responses and feedback of unknown human users who were not part of its learning. Further, in dialog systems assigning a reward for every dialog step or turn becomes very expensive so only some sparse signals can be assigned to desirable situations. General in task oriented dialog literature the definition of a 'win' is based on some previously constructed task that is given to users and the success of which is given by whether all the required subtasks or slots described are individually completed [79][49][80].

### 3.6.2 The Paradise Framework

The **PARA**digm for **DI**alogue **S**ystem **E**valuation or the PARADISE framework [81] is a dialog system evaluation process that uses the weighted sum of three main expressions to determine the user satisfaction. These are the dialog success or completion of primary dialog task, dialog costs such as duration or redundancy and the amount of times the system has to explicitly confirm some information with the user. The use of PARADISE is advantageous as a comparison between systems built for different domains is possible due to the normalized measurement and the effect of the dialog costs in different domain can be measured by the learnt weights. It should also be noted that task completion is rarely viable in a practical

situation with real users and certain reservations about this framework's validity have been raised [82]

### 3.6.3 Reward Shaping

A big issue with Reinforcement Learning is called the *Credit Assignment Problem*. This is the fact in RL and specifically MDPs with large and fine-grained state-action spaces, rewards can occur very late. For example in the game of chess, the agent will have to make many moves through its state-action where the immediate rewards of zero but gets a large positive reward at the very end of the game, if the agent happens to win. Since this reward signal occurs very late, it only weakly affects the temporally distant state that preceding the final state where the agent received a reward for winning. This means that the influence of a reward gets further and further diluted and thus leads to unreliable convergence properties of the RL process and a slow rate of learning. One way to address this problem add an additional reward signal $F$ that encodes the expert domain knowledge that is complimentary to the normal environmental reward R [83]. Making the final reward take the form $\hat{R} = R + F$. Since the reward is representative of the learning objective as discussed above, changing it may change the original task and if designed badly could even degrade learning. To overcome this [84] proposed some formal conditions such as the difference between some potential functions $\phi$ on successive states $s$ and $s'$ that guarantees the preservation of the optimality of the policies. This means that:

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \tag{3.38}$$

where $\gamma$ is teh same learning rate used to train the policy. In this way $\phi$ lets us encode expert or domain knowledge into the reward signal without changing the original task. Thus the total reward is given by:

$$\begin{aligned}
&\mathbb{E}[\hat{r}_0 + \gamma\hat{r}_1 + \gamma^2\hat{r}_2 + ...] \\
&= \mathbb{E}[\big(r_0 + \gamma\phi(s_1) - \phi(s_0\big) + \gamma\big(r_1 + \gamma\phi(s_2) - \phi(s_1\big) + ...] \\
&= \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + ... - \phi(s_0)]
\end{aligned} \tag{3.39}$$

This changes the value function and advantage as:

$$\begin{aligned}
&\hat{q}_\pi\big(s, a\big) = q_\pi\big(s, a\big) - \phi(s) \\
&\hat{v}_\pi\big(s\big) = v_\pi\big(s\big) - \phi(s) \\
&\hat{A}_\pi\big(s, a\big) = \Big[[q_\pi\big(s, a\big) - \phi(s)] - [v_\pi\big(s\big) - \phi(s)]\Big] = A_\pi\big(s, a\big)
\end{aligned} \tag{3.40}$$

Thus we can see that the optimal policy using any RL approach is invariant to reward shaping given this property from Equation 3.38.

## 3.7 Auxiliary RL terminology

Previously we saw the various techniques used to find the optimal policy, these maybe value based, policy based or a combination of the two. However, this terminology of three primary methods do not show the intricacies and complexity of the various different dimensions which can also characterize the above familiar methods. Some these auxiliary approaches are shown here.

### 3.7.1 Model-based and Model-free

The first we will explore is the concept of a model of an environment ir more specifically an MDP. Having a full model of the MDP would mean a complete definition of {S,A,P,R} i.e. the entire possible Action Set $A$, the entire state space consisting of all possible states $S$, complete knowledge of the reward function $R$, and most importantly a complete definition of the state to state transition probabilities $P$. This is generally not possible in real world or practical situations. Thus *model-based* methods are methods which require the knowledge of the complete model, these include dynamic programming algorithms such as policy or value iteration[39] we saw in Section 3.5.1.1. These methods generally work by exploiting the Bellman Optimality Equation as we discussed before. On the other hand *model-free* methods need no prior knowledge of the internal workings of an MDP. They try to maximize the reward by inferring this unknown reward structure by directly interacting with the environment and collecting rewards and thus letting the agent learn the hidden structure and working of its environment.

### 3.7.2 Episodic and Continuous Tasks

As discussed previously we know that the goal of the RL agent is to find a policy that maximizes the expectation of discounted rewards starting at some time-step in the environment. Here we can distinguish between two types of tasks. One where this process is divided into several episodes of finite length which each end with some terminal state at time $T$ and starting again with some initial state $t$, this is called *episodic*. Another where the entire task consists of a single infinitely long episode with no terminal state and thus no terminal time step $T$, this is called *continuous*. A Goal Oriented dialog is a good example of an episodic task where each dialog occurs from a starting state to some terminal state when the dialog is complete. An example of continuous task might be the control of the spinning momentum of the blades of an helicopter to keep it flying.

### 3.7.3 On-line and Off-line

in RL the approaches that, during training, updates the agents policy based on the return from every sample or every set of $\{s_t, a_t, r_{t+1}\}$ encountered are said to be *Online*. Most MC algorithms such as SARSA are part of this category. On the contrary RL methods that update the policy with a batch of training samples (generally collected from one full episode) as a single parameter update are known as being *Offline*, Q-learning and some Policy Gradient methods can be part of this category.

### 3.7.4 On Policy and Off Policy

When thinking about RL policies, we can consider them to be one of two separate concepts, a behavioral policy $\sigma$ that is used to generate actions and hence play out episodes and the optimal target policy $\pi$, which the agent has to find. Now *On Policy* methods assume that the behavioral policy is the target policy i.e. $\sigma = \pi$ and work by optimizing it. Whereas *Off Policy* methods aim to update the current policy $\pi$ with samples produced from older behavioral policies $\sigma$. These older policies could be from an earlier stage in the agent's training or from a corpus of dialog.

### 3.7.5 Exploration and Exploitation

In RL when working with policies, the sort of actions we take can be distinguished into two distinct types, one is selecting the best action according to the policy, or the optimal action for that state. Another is to take a non-optimal or even random action. Choosing the former is called *exploitation* as we exploit the knowledge we gained until now which is represented in policy or value function. While choosing the latter is known as an *exploratory* action as it lets us explore and collect more information from the environment thus letting us reach a truly optimal policy (as where might be states we have not yet seen because the current "optimal" action does not take us there). The problem of deciding what sort of action to take is known as the exploration/exploitation problem and one we way solve it is by using what is known as the $\epsilon$-greedy method. Here we select exploitation with the probability $1-\epsilon$ and some random exploratory action with probability $\epsilon$.

# 4 Summary of Deep Learning

## 4.1 Overview

Previously we saw theoretical foundations of RL and how they can be applied to the domain of Dialog Management. This chapter aims to explore how different RL algorithms based on the above described strategies for dialog optimization can be used in developing simplistic dialog strategies. The effects that the parameters of the learning framework has on the resulting strategies are also elaborated. Specifically we see how the simulated environment, hyper-parameters, reward functions and states affect the length of training time and what strategy is learned..... to be continued.

## 4.2 Deep Feedforward Networks

In Section 3.5.1.5, we explored the concept of function approximation and introduced the idea of Neural Networks to achieve this. Next we delve into an important type of Neural Network called the Multi-Layer Perceptrons (MLP) or more commonly the Deep Feedforward Networks. MLPs are one of the foundational models that stimulated the inception of the Deep Learning field. In essence, the primary aim of an MLP is to approximate some function $f^*$[85].

As an example consider a simple function that can classify inputs into different known categorizers, $y = f^*(x)$ which maps the given input $x$ to some category $y$. Now we can create an MLP that tries to approximate this mapping, but with an additional parameter $\theta$, $y = f(x|\theta)$. Once this is established the network aims to learn the best values for the parameter $\theta$ that would generate the best approximation for the given function. Here we use the term *feedforward* because of how the information flows unidirectionally from input to output with no mechanism for feedback. The information always flows from the input, $x$ through the intermediary computations which represent $f$, and then to the output $y$. MLPs are generally represented by the chaining of many different functions. This structure can be associated with a directed acyclic graph that can describe how the functions are chained together. Consider three functions $f^1, f^2 \text{and} f^3$ chained to form $f(x) = f^3(f^2(f1(x)))$, this is exactly the type of structures that MLPs use. Here $f^1$ is generally known as the *first layer* of the MLP and $f^2$, the second and so on until the final layer which is referred to as the *output layer*. In fact, the overall number of layers or the length of the chain is decribed as the *depth* of the model and this is where the *deep* in deep learning originates.

The method of computing the best values for $\theta$ to generate the best approximation of $f^*$ is known as *training* the MLP. During this process the approximate function $f(x)$ is repeatedly optimized and pushed to match $f^*(x)$. This process required what is known

as *training data*. This data is a collection of appropriate examples of $f^*(x)$ that properly represent the required properties of $f^*$. Each example is this collection, $x$ is further tabulated with its *label* which represents what category it is $y \approx f^*(x)$. In this way the training data directly specifies what is required from the output layer for each point in the input space $x$. An important consideration here is that the behavior of the middle layers (in between the input and output) is not directly given in the training data and it is up to the learning algorithm used, to decide how these layers should be used to generate the required output and thus best approximate $f^*$. Since the training data does not specify the behavior for these individual layers, they are known as *hidden layers*.

In neural network models each hidden layer is generally vector valued and the dimensions of these hidden layers describes what is known as the *width* of the model. Each element of this vector is said to represent the analogous role a neuron plays in our brains and this is the origin of *neural* in neural networks. We can think of each of these layers as composed of many such elements called *units*, that when acting in parallel can represent a vector-to-scalar or vector-to-vector function. This idea of a multi layered chain each consisting of vector-valued representation of input semantics is also closely associated to neuro-scientific findings about the brain. Further when successive layers of a network have all their neurons or units connected to one another, they are known as *fully-connected* layers. Fully connected layers and their research formed the first generation of neural networks. Further research on the sparsity of these interconnected layers lead to what is known as *convolutional networks*[86], where specific features *convolve* in a batch of examples, thus making the network invariant to input translations. However in this work, we always assume this full connectivity when we refer to neural networks. We can see a representation of a fully connected neural network in figure 8[5], where each unit of the input layer is fully connected to each unit of the next layer and so on until the output layer.

## 4.3 Loss Functions

Deep learning involves the process of minimizing or maximizing some function $f(x)$ by varying $x$[85], this is known as optimization and the function we want to optimize is known as the objective functions, cost function or loss function. Typically, our model consists of a distribution $P(y|x, \theta)$ and we use what is known as the *principle of maximum likelihood* to arrive at the discrepancy, also known as the *cross-entropy* between the training data and the approximated function's predictions, as a representation of the loss function. This is elaborated further below

[5] http://neuralnetworksanddeeplearning.com/chap5.html

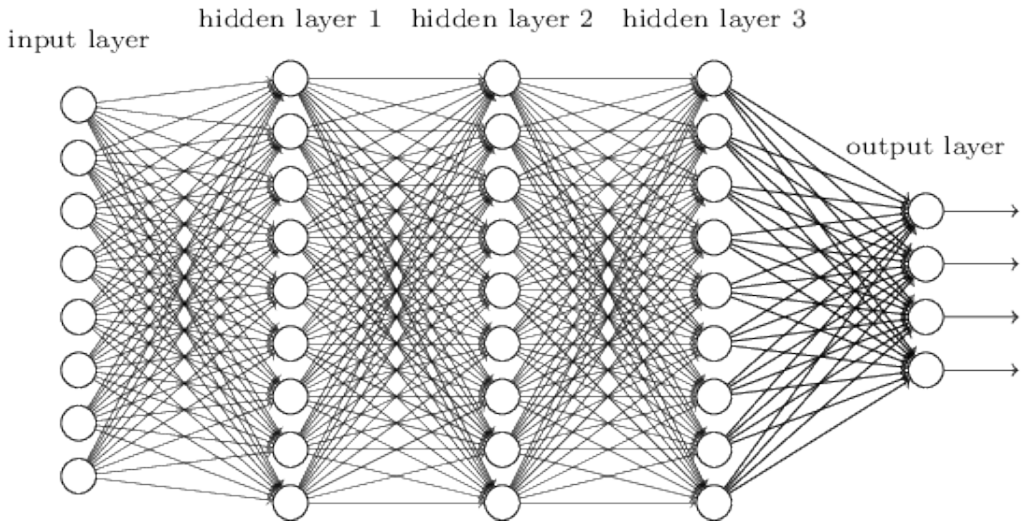


Figure 8: Fully Connected Neural Network

### 4.3.1 Maximum Likelihood Estimation

This method was derived from wanting to have some principle from which we could generate specific loss functions that work for different models. Consider a a collection of training data that consists of $m$ separate examples, $\mathbb{X} = \{x_1, x_2, ..., x_m\}$. Since $\mathbb{X}$ is a representation of the true distribution $P_{data}$, we can assume that each $x_i$ is valid independently of each other. Let $P_{model}(x|\theta)$ be a parameterized probability distribution over the same space but now indexed by $\theta$, then $P_{model}(x|\theta)$ will map any input $x$ to a real number that estimated the true probability $P_{data}(x)$. The maximum likelihood parameter $\theta$ is then:

$$\begin{aligned} \theta_{ML} &= \arg\max_{\theta} P_{model}\big(\mathbb{X}|\theta\big) \\ &= \arg\max_{\theta} \prod_{i=1}^{m} P_{model}\big(x_i|\theta\big) \end{aligned} \tag{4.1}$$

To simplify the above we take the logarithm of the likelihood as it does not alter the arg max but only transform the product to a sum:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log P_{model}\big(x_i|\theta\big) \tag{4.2}$$

Since the arg max is invariant to rescaling, we further divide by $m$ to obtain a general loss function that is represented by the *expectation* with respect to the true distribution $P_{data}$

that is given by the training data:

$$\theta_{ML} = \arg\max_{\theta} \mathbb{E}_{x\sim P_{data}} \big[ \log P_{model}\big(x|\theta\big) \big] \tag{4.3}$$

The above equation can now be interpreted as the process of minimizing the discrepancy between the true data $P_{data}$, defined by the training data and model distribution $P_{model}$, defined by the predictions of the model. The degree of this dissimilarity is called the *Kullback Leibler* divergence or more commonly as KL divergence. This is given by :

$$D_{KL}\big(P_{data}||P_{model}\big) = \mathbb{E}_{x\sim P_{data}} \big[ \log P_{data}\big(x\big) - \log P_{model}\big(x\big) \big] \tag{4.4}$$

Since the term $\log P_{data}\big(x\big)$ is not a function of the model and is known, to minimize the KL divergence means that we need to only minimize:

$$-\mathbb{E}_{x\sim P_{data}} \big[ \log P_{model}\big(x\big) \big] \tag{4.5}$$

Thus minimizing the KL divergence is analogous to minimizing the cross-entropy between the distributions.

### 4.3.2 Approximating Conditional Distributions with Maximum Likelihood

As we saw is the previous section the maximum likelihood is the negative log-likelihood, also known as the cross-entropy between the training data and model predictions. Typically neural networks are training using this principle. This implies that a general loss function can be represented as :

$$J\big(\theta\big) = -\mathbb{E}_{x,y\sim P_{data}} \log P_{model}\big(y|x\big) \tag{4.6}$$

The specific form can change between different models and depend largely on the structure of $\log P_{model}$. Further it can be shown that the loss function is invariant to terms that do not depend on the model parameters and thus can be discarded. That is, if $P_{model}(x|y) = F(y, f(x,\theta), I)$ where $I$ represents the extra terms, then we arrive at the mean squared error cost:

$$J\big(\theta\big) = \frac{1}{2}\mathbb{E}_{x,y\sim P_{data}} [\![ y - f\big(x,\theta\big) ]\!]^2 + \text{const} \tag{4.7}$$

Thus we can see that the equivalence between the maximum likelihood loss and the minimization of mean squared error holds for not only linear models, but for any $f(x,\theta)$. A major advantage with this method is that we can avoid the effort needed to generate cost functions for each different model, letting us automatically generate a loss function $\log P(x|y)$ for any model $P(x|y)$

## 4.4 Output Units

In the case of neural networks there seems to be a close relationship between the type of loss function chosen and the type of units used in the output layer of the network. Generally all types of neural network units can be used anywhere in the network and not only the output, in this case we focus on the use of these units as outputs. Consider a complex set of training data that includes some *hidden* features described by $h = f(x, \theta)$. The function of the output is then to handle these features, generally by additional transformation from the features to the labels, to arrive at good predictions.

### 4.4.1 Linear Units

An affine transformation is one where points, straight lines and planes are preserved. One of the simplest types of units are called linear units and are based on affine transformations without any non-linearity. Given that we have features $h$, a layer of this type of unit generates a vector $\hat{y} = W^T h + b$. These output layers are generally employed to generate the mean of a conditional distribution $P(y|x) = \aleph(y|\hat{y}, I)$, then maximizing the log-likelihood is analogous to minimizing the mean squared error.

### 4.4.2 Sigmoid Units

It is common to find machine learning tasks that have a need to predict a binary variable $y$. The approach using maximum likelihood involves the definition of a Bernoulli Distribution over $y$ depending on $x$. This distribution is defined by a number and the neural network has to only predict whether $P(y = 1|x)$ and for this to be a valid probability it must lie in the interval $\big[0, 1\big]$. Due to these constraints we use an approach that combines the sigmoid unit with the maximum likelihood. Thus a sigmoid output unit is defined by

$$\hat{y} = \sigma\big(w^T h + b\big) \tag{4.8}$$

where $\sigma$ is the logistic sigmoid, defined by $\sigma(x) = \frac{1}{1+e^{-x}}$. We can interpret this as the output unit having two distinct components. First a linear layer is used to compute $\big(w^T h + b\big)$. Then, the logistic sigmoid is used to convert this into a probability.

### 4.4.3 Softmax Units

The softmax output is used in situations where a probability distribution over a discrete variable with $n$ different possible values needs to be modeled. For example, in the case of representing a mapping between actions and states of an RL agent where there are $n$ different states. Thus softmax functions are generally employed as outputs of classifiers to represent

the distribution over $n$ different classes. These type of outputs can also be considered as a generalization of the above sigmoid function. With the sigmoid, we only had to 2 classes of outputs 0 and 1, to generalize this to the case of $n$ different values we require a vector $\hat{y}$ such that $\hat{y}_i = P\big(y = i|x\big)$. Not only is each element of $\hat{y}_i$ to be between 0 and 1, but the entire vector $\hat{y}$ should also sum to 1, in order to make it represent a valid probability. To do this we first consider a linear layer that predicts raw log probabilities:

$$z = W^T h + b \tag{4.9}$$

where $z_i = \log \tilde{P}(y = i|x)$, here $\tilde{P}$ represents an unnormalized probability distribution. We then exponentiate and normalize $z$ to get the desired $\hat{y}$. Thus :

$$\text{softmax}\big(z\big)_i = \frac{\exp\big(z_i\big)}{\sum_j \exp\big(z_i\big)} \tag{4.10}$$

Now we need to maximize $\log P(y = i; z) = \log \text{softmax}(z)_i$ Applying log-likelihood we get:

$$\log \text{softmax}\big(z\big)_i = z_i - \log \sum_j \exp\big(z_j\big) \tag{4.11}$$

Thus the softmax units helps us to handle cases where there are $n$ separate output classes.

## 4.5 Hidden Units

Previously we saw the various design choices for MLPs that are actually common to most ML models that use gradient based training methods. Now we consider the types of units used in hidden layers which is unique to MLPs. Typically, hidden units can be represented as some vector of inputs $x$ that has a linear affine transformation applied on it $z = W^T x + b$. To this transformation some element-wise non-linear function $g(z)$ is applied. Hidden units only tend to vary the activation function $g(z)$.

### 4.5.1 Rectified Linear Units(ReLU)

Earlier we discussed the linear unit in section 4.4.1. The difference between a rectified linear unit and a linear unit is that for half of its domain the ReLU will give an output of zero. Thus, the output activation function that ReLU's use is of the form $g(z) = \max\{0, z\}$. From this function we can intuitively see that the derivatives through a ReLU remain large while the unit is active. Further, the second derivative of this operation outputs zero everywhere.

As discussed, the ReLUs are generally used after an affine linear transformation. Thus we have

$$h = g\big(W^T x + b\big) \tag{4.12}$$

When using ReLUs, we generally set the parameters of the transformation such that all elements of $b$ and the weights hold a small positive value. This ensures that the ReLU is active for most inputs in the training data and allows the derivatives to pass. The plot of the rectifier $g(z)$ can be seen in figure 9

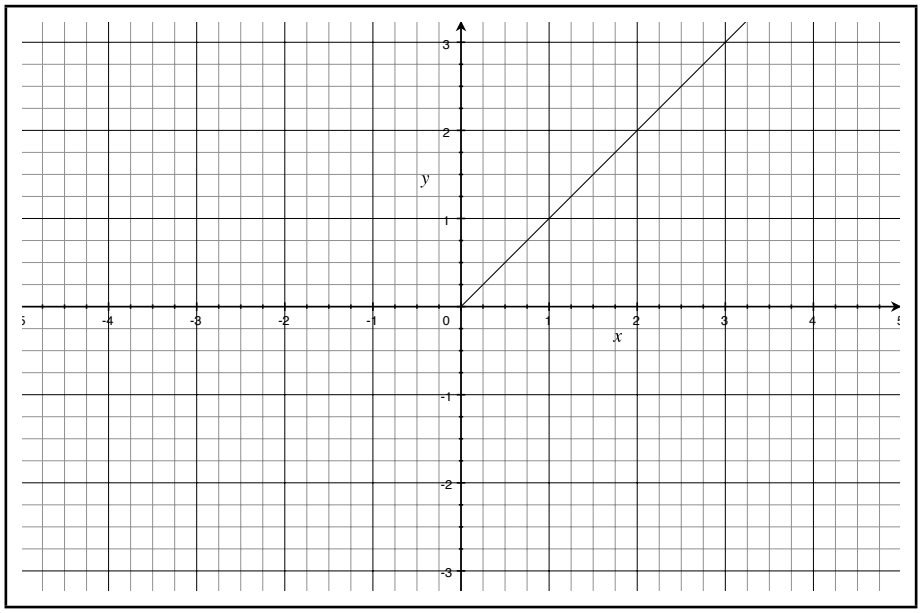


Figure 9: Rectified Linear Unit

One major drawback of ReLUs is that if the examples in the training set cause its activation to be pushed to zero, then no learning is performed. Some generalizations of ReLUs were later designed to ensure a gradient everywhere. These generalizations of ReLUs, whose performance is generally comparable to that of the vanilla ReLU are typically based on a non-zero slope $\alpha_i$ when $z_i < 0$. This gives us $h_i = g(z_i, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)$. A popular generalization of ReLUs that is found to be very effective with image recognition tasks is called the *Absolute Value Rectifier* [87]. It pushes $g(z) = |z|$ by fixing $\alpha_i = -1$. Other typical generalizations of ReLUs are widely used across various task domains. These include the *Leaky* ReLU that sets $\alpha_i$ to a small positive value like 0.01[88]. Further the Parametric ReLU lets $\alpha_i$ be learnt during training[89].

### 4.5.2 The Hyperbolic Tangent and Logistic Sigmoid

The ReLU we learnt about in the previous section, is actually in use only recently. Prior to their invention, most neural networks and other ML algorithms that use gradients for optimization used the logistic sigmoid or the hyperbolic tangent activation functions described below. The sigmoid activation function is of the form,

$$g(z) = \sigma(z) \tag{4.13}$$

And hyperbolic tangent activation function is represented by,

$$g(z) = \tanh(z) \tag{4.14}$$

Further the above functions are closely related due to the relation

$$\tanh(z) = 2\sigma(2z) - 1$$

Previously, we discussed sigmoid units used as outputs to predict binary classification results. Sigmoidal units, unlike element-wise linear units saturate to a high value when $z$ is very positive and go to a low value when $z$ is negative, the plot of the sigmoid function can be seen in figure 10. When $z$ is near 0, the sigmoidal becomes strongly sensitive to the input. Due to this saturation, they often make gradient based learning difficult and thus their use as hidden units is generally discouraged in the field.

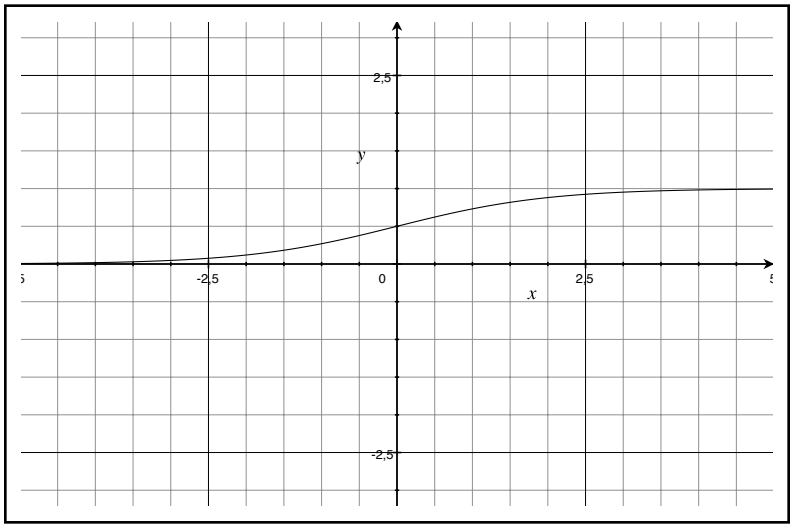


Figure 10: Logistic Sigmoid Unit

Typically the hyperbolic tangent activation performs better than the sigmoid. Due to the fact that $\tanh(0) = 0$ while $\sigma(0) = 0.5$, the hyperbolic tangent resembles the identity function closely as we approach 0, this can be seen in Figure 11. Training a network $\hat{y} = w^T \tanh(U^T \tanh(V^T x))$ is similar to training a simple linear model $\hat{y} = w^T U^T V^T x$ given that the activations of the network can be kept small and manageable, this makes training with tanh more efficient.

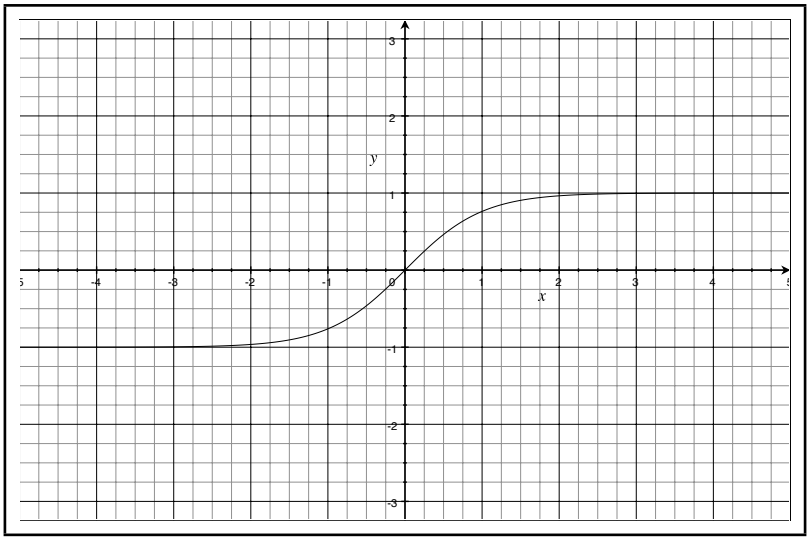


Figure 11: Hyperbolic Tangent Unit

## 4.6 Architecture

Previously we saw the internal workings of the MLP, the design and working of its internal components and cost functions. Now we focus on two main questions, the solution of which forms the architecture of a MLP. These are typically how many units should it have(the *width* of the network) in how many successive layers(the *depth* of the network). Here layers are groups of units or neurons which are typically arranged into a chain like structure with each successive layer being fed the output of the previous layer or, each layer is a function of the preceding layer. In these chain like structures the index gives the position of the layer it belongs to, such that

$$h^{(1)} = g^{(1)}(W^{(1)T} x + b^{(1)})$$

represents the first layer and

$$h^{(2)} = g^{(2)}(W^{(2)T} x + b^{(2)})$$

represents the second[85] and so on. This is illustrated in figure 12, where we can see a neural network with a depth of 3 consisting of a single output layer with 4 units, a single hidden layer with 3 units and an output layer with 1 unit.

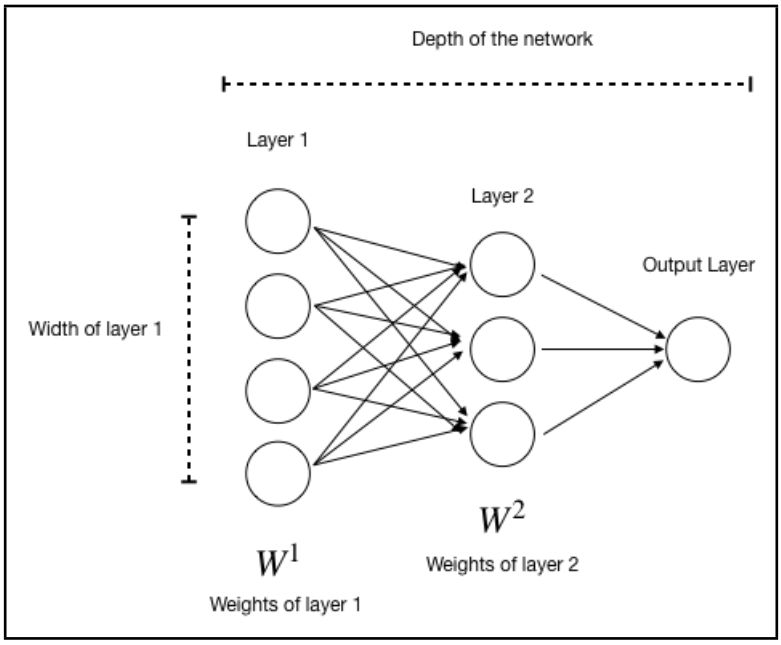


Figure 12: General Architecture of a Neural Network

## 4.7 Gradient Based Learning

Typically machine learning algorithms involve the task of optimizing, which means the process of maximizing and minimizing a function (generally a loss function) $f(x)$ by repeatedly altering $x$[85]. Consider some function $y = f(x)$, with $x, y \in \mathbb{R}$. We know that the derivative of this function is given by $f'(x)$ or $\frac{dy}{dx}$ and represents the slope of $f(x)$ at a certain point $x$. In essence, this tells us how by making a small change in $x$ we can get a corresponding change in $y$ or how the output can be pushed in a certain direction according to changes in the input: $f(x + \epsilon) \simeq f(x) + \epsilon f'(x)$. Thus, the derivative is important in the process of optimization, as it tells us how to improve $y$ by changing $x$ allowing us to reduce $f(x)$ by moving $x$ in small steps with the opposite sign of the computed derivative. Invented by the mathematician Cauchy in 1847 [90], this process, illustrated in figure 13[6], is fundamental to many machine learning algorithms and central to deep learning where it is known as *gradient*

[6]http://www.deeplearningbook.org/contents/numerical.html, Page 81

*descent.*

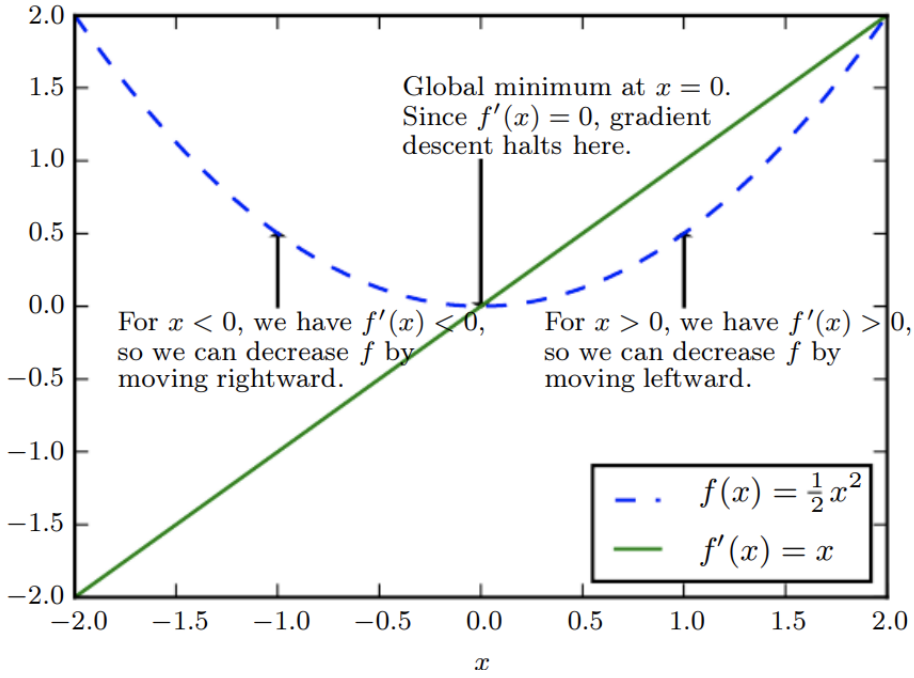


Figure 13: The Process of Gradient Descent[85]

### 4.7.1 Back-Propagation

As we now know a feedforward neural network produces an output $\hat{y}$ for an input $x$, this process involves a flow of information through the layers of the network. The information provided by $x$ is *propagated* to the units at each layer and consequently generates $\hat{y}$. This process is known as *forward propagation*. This propagation continues until finally a scalar loss $J(\theta)$ is produced (by comparing y and $\hat{y}$). Now in order to apply gradient descent or any other gradient based learning algorithm, we need to compute the gradient of this loss. This is achieved by letting the information flow from the loss, backward through the network. This process is known as *back-propagation*[91] and is illustrated in Figure 14. Thus, the back-propagation algorithm is only the technique used to compute the gradient. Some other algorithm such as stochastic gradient descent uses this gradient to perform learning.

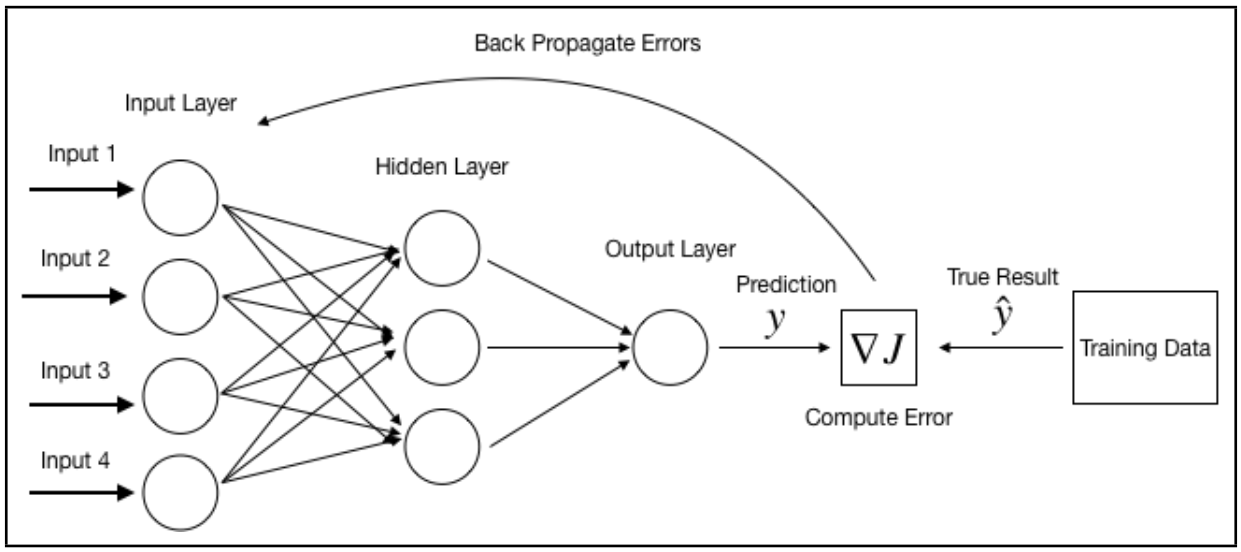


Figure 14: Back Propagating Errors Through a Neural Network

Here we formally express the process of computing the gradient $\nabla_x f(x, y)$ for some function $f$, with $x$ as a set of variables whose derivatives we need and $y$ as a set of inputs to the function. In the case of learning algorithms, we use the gradient of the loss function we discussed above, based on the weights of the network i.e. $\nabla_\theta J(\theta)$. Now we consider forward and back propagation for the layers of a fully connected MLP. In Algorithm 3 we see the process of forward-propagation. This process performs a mapping between a single training example (Input $= x$, Target $= y$). Here the loss $L(\hat{y}, y)$ depends directly upon the output of the final layer $\hat{y}$ and the label of the current input $y$. Typically, a regularizer $\Omega(\theta)$ is added before computing the total loss $J$, where the parameter $\theta$ describes the weights and biases

of the network.

**Algorithm 3:** Forward Propagation through a deep neural network and the loss computation

**Require:** Network depth, $l$
**Require:** $W^{(i)}, i \in \{1, ..., l\}$, the weight vectors of the network
**Require:** $b^{(i)}, i \in \{1, ..., l\}$, the bias parameters of the network
**Require:** $x$, the input to process
**Require:** $y$, the target output

$h^{(0)} = x$
**for** $k = 1, ..., l$ **do**
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(k)} = f(a^{(k)})$
**end**
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Above we saw the process used to propagate information in the forward direction though a deep neural network and the consequent computation of the loss. Now to perform gradient-based learning, we need to compute the gradient of this loss $\nabla J$ with respect to the neural network parameters $W$ and $b$, this back-propagation process can be seen in Algorithm 4. The aim is to generate the gradients on the activations $a^{(k)}$ for every layer $k$, but this time going backwards, starting from the output layer and going toward the input layer. From the computed gradients, we can get a picture of how the parameters of each layer should be varied in order to move the predicted result $y$ closer to the true target $\hat{y}$.

### 4.7.2 Stochastic Gradient Descent(SGD)

One of the most widely used optimization algorithms in this field is SGD. Earlier we saw the process of gradient descent where we used a derivative of a function to descend downhill by making small changes in the input. This process is strengthened by using SGD to follow the gradient of small batches of data downhill to reduce the loss. For this algorithm, we use small batches of data known as mini-batches, which let us compute the estimate of the gradient by averaging over a mini-batch of $m$ examples. An important hyper-parameter for SGD is the *learning rate*. Practically, when training a neural network it is required to gradually decrease the learning rate as time passes, this is performed by describing the learning rate as $\epsilon_k$ for the $k$-th iteration. The reason behind this learning rate *decay* is that the noise generated

**Algorithm 4:** Back Propagation

After the forward propagation, compute the gradient on the output layer
$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$
**for** $k = l, l-1, ..., 1$ **do**
  Convert the grdient on the layer's output into a gradeint on the activation function
  $g \leftarrow \nabla_{a^k} J = g \odot f'(a^k)$
  Compute gradients on weights and biases:
  $\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$
  $\nabla_{W^{(k)}} J = g h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t the next lower-level hidden layer's activations:
  $g \leftarrow \nabla_{h^{(k-1)}} J = W^{(k)T} g$
**end**

**Algorithm 5:** Stochastic Gradient Descent Algorithm

**Require:** Learning Rate Schedule, $\epsilon_1, \epsilon_2, \ldots$
**Require:** Initial Parameter $\theta$
$k \leftarrow 1$
**while** *Not Stopping Criteria* **do**
  Sample $m$ examples from training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with respective targets $y^{(i)}$
  Compute Gradient: $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
  Apply Update: $\theta \leftarrow \theta - \epsilon_k \hat{g}$
  Increment $k$: $k \leftarrow k + 1$
**end**

from randomly sampling $m$ examples at a time (using mini-batches) stays active even when we successfully minimize the loss. Algorithm 5 shows the process of SGD:

Further, a sufficient condition to ensure the convergence os SGD is :

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \text{ and} \tag{4.15}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty \tag{4.16}$$

# 5 Neural Dialog Management with Deep Reinforcement Learning

## 5.1 Overview

Previously we saw how we can develop dialog strategies by applying Policy and Value Iteration Methods. In this chapter we will see the implementation of Algorithms from these methods, specifically the Advantage Actor Critic Algorithm applied to learn simple dialog strategies. What we mean by Neural Dialog Management. What are actor critic methods (refer section 4.5.3). How is it conceptually applied to dm, like what does the critic do actor do with respect to dialog and what does the critic do with respect to dialog.

## 5.2 REINFORCE

In Section 3.5.2 we learnt about finding the gradient of the objective loss of a parameterized policy i.e. the policy gradient. We now use the policy gradient we derived in Equation 3.30 to build a PG learning algorithm. This is known as the REINFORCE algorithm or Monte Carlo Policy Gradient algorithm[92]. For the parameterization we use a deep neural network to approximate the policy $\pi_\theta$ where $\theta$ represent the weights of the network. The PG theorem gives us an expression (Equation 3.30) that is exactly equal to the gradient of the loss or performance measure of the parameterized policy. To use this method for actual learning we need a way to sample the episodes of the environment such as the expectation of the samples for each episodes is approximately equal to this expression i.e RHS of Equation 3.30. With a little further investigation we can also see that the RHS of the PG theorem is a sum over the states weighted by the recurrence rate of these states i.e how often we see these states under the required policy $\pi$. Thus we have

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \sum_s d_\pi(s) \sum_a q_\pi(s,a) \nabla_\theta \pi_\theta(a|s) \\ &= \mathbb{E}_\pi\big[\sum_a q_\pi(s_t,a) \nabla_\theta \pi_\theta(a|s_t)\big] \end{aligned} \tag{5.1}$$

Now in the remaining part of the expectation above is a sum over actions i.e. $\sum_a \nabla_\theta \pi_\theta(a|s)$. To make it such that the action in the above equation $a$ represents the sample of actions $a_t$ we must make sure that each term above is weighted by the probability of selecting those actions i.e. by $\pi_\theta(a_t|s_t)$.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\Big[\sum_a \pi_\theta(a|s_t) q_\pi(s_t,a) \frac{\nabla_\theta \pi_\theta(a|s_t)}{\pi_\theta(a|s_t)}\Big] \tag{5.2}$$

Now we can replace $a$ by sample action $a_t$

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi\left[q_\pi(s_t, a_t)\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right] \\ &= \mathbb{E}_\pi\left[G_t\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right], \text{ Since } \mathbb{E}_\pi\big[G_t|s_t, a_t\big] = q_\pi(s_t, a_t)\end{aligned} \tag{5.3}$$

Here $G_t$ is the total return for one episode. This final expression is a quantity that can be sampled on for the steps of each episode and whose expectation is equal to the policy gradient. Thus we can define an update rule for the parameterized weights as:

$$\theta_{t+1} = \theta_t + \alpha G_t\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} \tag{5.4}$$

This is called the REINFORCE algorithm and the above expression is its update with $\alpha$ being its learning rate. Intuitively here we can see that each increment to the weights is directly proportional to the product of the return and the gradient of the probability of taking the action that was taken divided by the probability of taking that very action. Thus this update pushes this second term which is a vector in the direction proportional to the return and inversely proportional to the probability of taking that action. We do this as we can then move the parameter in the direction which produce the action that yield the highest possible return and because we can give lesser importance to the most frequently selected actions as they may not lead to the highest returns. Since this algorithm uses the total return from that starting time step of an episode $t$ which includes all rewards until the end of the episode i.e. performs a full backup, this is also known as Monte Carlo Policy Gradient

### 5.2.1 REINFORCE Algorithm

Using the update rule we derived in the previous section i.e. Equation 5.4 we can build the full REINFORCE algorithm and process as shown in Algorithm 6

### 5.2.2 Policy Network

The Policy Network approximates the policy $\pi_\theta(a|s)$ with parameter $\theta$ can be found in the above figure. It is a Neural Network with the following characteristics:

- Input Layer : The Input layer consists units with equal number to that of the dimensions of the state space. The Input is the observable state $S$. The Input Layer has a

**Algorithm 6:** REINFORCE

**Input :** Differentiable Policy Parameterization $\pi(a|s, \theta)$
**Require:** Initialize Policy Parameter $\theta$
**while** *S is not Terminal* **do**
  Generate an episode on policy $\pi_\theta$: $s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t$
  **for** *Each Step of the episode* $t = 0, ..., T$ **do**
    $G_t \leftarrow$ Return at time $t$
    Update Policy Parameters : $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(a_t|s_t)$
  **end**
**end**

Rectified Linear Unit as its activation. Further the Input Layer is initialized with zero values for all its weights and biases.

- Hidden Layers : The two Hidden layers consist of 32 and 16 neurons respectively. The hidden layers are fed from the output of th ReLU from the Input layers. Further each hidden layer has another ReLU as its activation. Both Hidden Layers are initialized with zero values for their weights and biases.

- Ouput Layer: The Output Layer consists of units with the same number as the dimension of the action space. The role of the output layer is to produce a probability distribution across the various possible actions so that we can choose the action with the highest probability. Thus we use a Softmax as the activation function from Section 4.4.3 which can generate the required probability distribution across the $n$ actions in the action space. The Output Layer is also initialized with zero values for all its weights and biases.

- Loss Function: here we use the Mean Squared Error loss function, from Equation (4.7).

- Optimizer: For the learning algorithm we have used the Adaptive Moments Optimizer (Adam)[93] which is a special form of the SGD Algorithm where the learning rate adapts to the magnitude of the partial derivative of the cost function.

## 5.3 Advantage Actor Critic

The Advantage Actor Critic Algorithm is a hybrid method that combines both policy and value learning. This method involves the use of two neural networks. The first is called

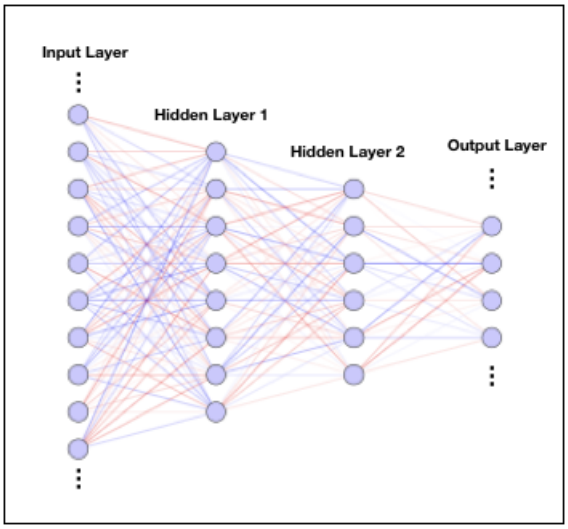


Figure 15: Policy Neural Network

an *Actor*: $\pi(a|s,\theta)$, this control how the RL agent behaves, in other works maps semantic states to actions. The other network is called a *Critic*: $\hat{q}(s,a,w)$, which measures the value or quality of the action taken in the present state. Here $\theta$ and $w$ represents the parameters of their respective networks. This technique overcomes the drawback of the previous *Vanilla* Policy Gradients such as a high degree of variance and the slow learning and time to convergence. This is done by introducing a better scoring function [33]. A good candidate for this function is the learned value function or Critic as we have seen in Section 3.5.3. Typically the Critic is implemented using the Temporal Difference update algorithm and the Actor using the vanilla Policy Gradient update with the value function in its expectation from Equation 3.30 $q_\pi(s,a)$ by the Critic Network $\hat{q}_\pi(s,a,w)$ where $w$ represents the weights of the network and $\hat{q}$ represents the approximated value function. This method has a major drawback which is that value function approximation have a high degree of variance which is addressed by the *Advantage function*. An overview of the entire Advantage Actor Critic process can be seen in Figure 16.

### 5.3.1 Advantage Function

To address the drawbacks of using value approximation directly in learning, the Advantage function was introduced [33]. From Policy methods in Section 3.5.2 we know that the objective is to find a parameterized policy $\pi_\theta(a|s)$ that can maximize the objective function, in this case the total expected discounted cumulative reward $J(\theta)$ starting from a certain

dialog state over all possible dialog paths. Further, from the *Policy Gradient Theorem*[39] we have the gradient of the objective function as:

$$\nabla_\theta J\big(\theta\big) = \mathbb{E}_\pi\big[q_\pi\big(s,a\big)\nabla_\theta \ln \pi_\theta\big(a|s\big)\big] \qquad \big(\text{From Eq. } 3.30\big).$$

This representation of the gradient implies a potentially high degree of variance. This can be handled with the introduction of a comparison of the action-value to a baseline while not affecting the estimated gradient [39]. The value function is generally a good choice for this and thus Equation 3.30 becomes:

$$\nabla_\theta J\big(\theta\big) = \mathbb{E}_\pi\big[\nabla_\theta \ln \pi_\theta\big(a|s\big)\big(q_\pi\big(s,a\big) - v_\pi\big(s\big)\big)\big]. \tag{5.5}$$

Here the expression $\big(\hat{q}_\pi\big(s,a\big) - \hat{v}_\pi\big(s\big)\big)$ is known as the *Advantage function* $A(s,a)$ and tells us the improvement in the action taken in a particular state compared to the average for that state. Thus if $A(s,a) > 0$ then our gradient of the objective function $\nabla_\theta J(\theta)$ is returning a value higher than the expected average for that state and thus is sent further in the same direction. If $A(s,a) < 0$, this means that our action returned a reward that was worse than the average for that state and thus the gradient is sent backward in the opposite direction. To implement this algorithm we need two value functions $\hat{q}_\pi(s,a)$ and $\hat{v}_\pi(s)$, this can be simplified by using the expression from Equation 3.10.

$$\hat{A}_\pi(s_t,a,w) = R_{t+1} + \gamma\hat{v}_\pi\big(s_{t+1},w\big) - \hat{v}_\pi\big(s_t,w\big) \tag{5.6}$$

where $s$ is the observed state of the dialog, $a$ is the chosen action, $t$ is the current time-step and $w$ is the parameters of the neural network (or any other function approximators)

### 5.3.2 A2C Algorithm

Typically in Parameterized Actor Critic methods, we seek to maximize the performance such that each update to the objective function $J(\theta)$ approximates a gradient *ascent* in $J$, this is given by :

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J\big(\theta_t\big)} \tag{5.7}$$

Where $\nabla J\big(\theta_t\big)$ represents a stochastic estimate of the performance gradient with respect to its parameters $\theta_t$, and $\alpha$ is the learning rate of the optimizer. Using the above along with Equation 3.30 and unfolding to represent the update at each time-step $t$ we get :

$$\theta_{t+1} \doteq \theta_t + \alpha q_\pi\big(s,a\big)\nabla_\theta \ln \pi_\theta\big(a|s\big) \tag{5.8}$$

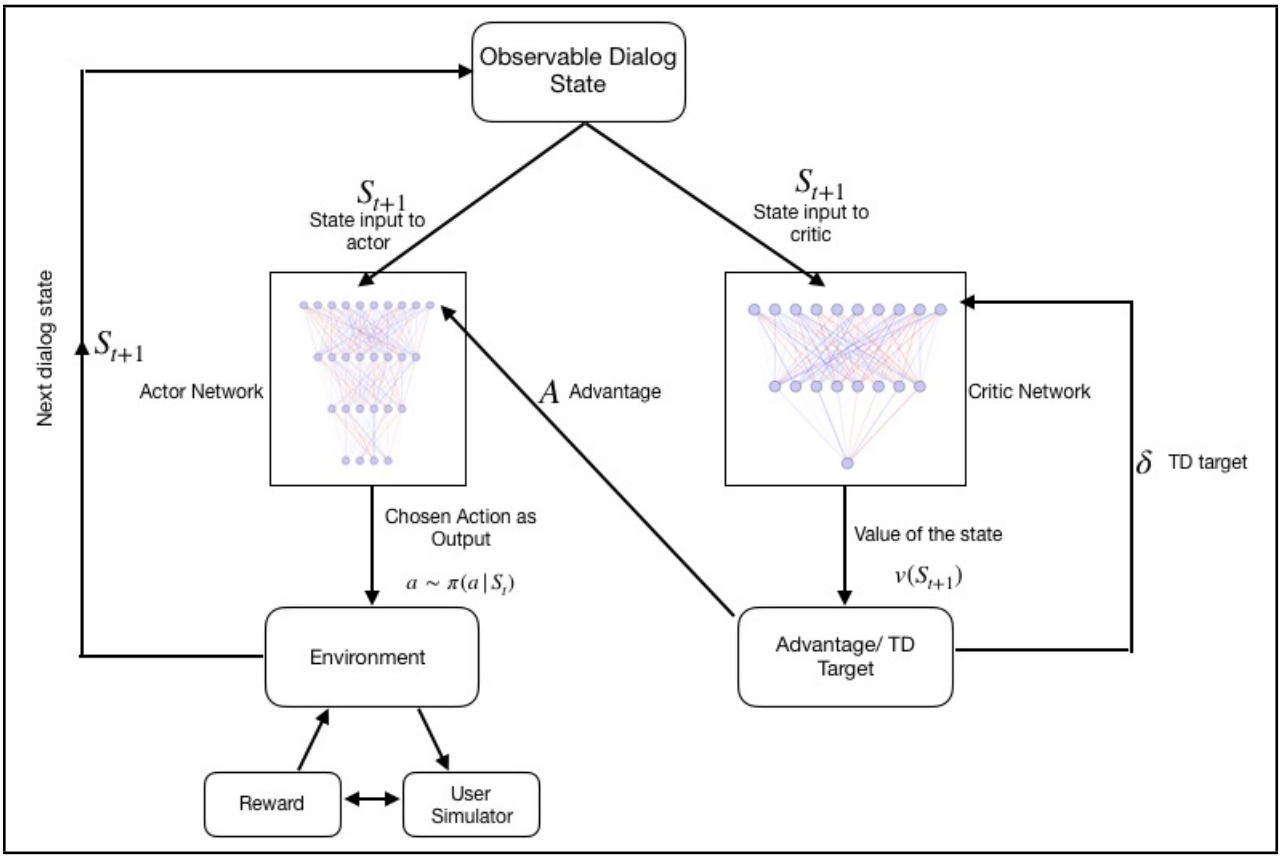


Figure 16: Actor-Critic Environment Interaction

This is known as the Monte Carlo Policy Gradient or the REINFORCE update [92]. In the case of Advantage Actor-Critic we replace $q_\pi$ in the above equation with the approximation $\hat{q}_\pi$. We follow this by replacing this state-value approximation with the Advantage function. Thus from Equation 5.5 and 5.6 we get our required Policy Network Parameter update:

$$\theta_{t+1} \doteq \theta_t + \alpha^\theta A_\pi\big(s, a, \theta\big) \nabla_\theta \ln \pi_\theta\big(a|s\big) \tag{5.9}$$

Similarly the Update for the Value or Critic Network is got by using the The Temporal Difference Target along with the definition of the objective function for the episodic case given by Equation 3.28[39]

$$w_{t+1} \doteq w_t + \alpha^w\big(R_{t+1} + \gamma \nabla_w v_\pi\big) \tag{5.10}$$

The entire process of the Advantage Actor Critic Method is described in the Algorithm found below,

**Algorithm 7:** Advantage Actor Critic Algorithm

**Input :** Differentiable Policy Parameterization $\pi(a|s,\theta)$
**Input :** Differentiable State-Value Parameterization $\hat{v}(s,w)$
**Require:** Initialize Starting Start $s$ of Episode
**Require:** Random Initialization $\theta$
**Require:** Random Initialization $w$
**Sample :** Action $a \sim \pi_\theta$
**while** $S$ *is not Terminal* **do**
  Take Action $a$
  Sample Reward $R$
  Sample Next State $S'$
  Compute Advantage $A_\pi \leftarrow R + \gamma\hat{v}_\pi\big(S', w\big) - \hat{v}_\pi\big(S, w\big)$
  Update Policy(Actor) Weights: $\theta \leftarrow \theta + \alpha^\theta A_\pi \nabla_\theta \ln \pi_\theta\big(a|S\big)$
  Update Value(Critic) Weights : $w \leftarrow w + \alpha^w\big(R_{t+1} + \gamma\nabla_w\hat{v}_\pi\big)$
  Get Next Action: $a \leftarrow a'$
  Get Next State: $S \leftarrow S'$
**end**

### 5.3.3 Actor Network

The Actor Network which approximate the policy $\pi_\theta(S)$ with parameter $\theta$ can be found in the above figure. It is a Neural Network with the following characteristics:

- Input Layer : The Input layer consists units with equal number to that of the dimensions of the state space. The Input is the observable state $S$. The Input Layer has a Rectified Linear Unit as its activation. Further the Input Layer is initialized with zero values for all its weights and biases.

- Hidden Layers : The two Hidden layers consist of 32 and 16 neurons respectively. The hidden layers are fed from the output of th ReLU from the Input layers. Further each hidden layer has another ReLU as its activation. Both Hidden Layers are initialized with zero values for their weights and biases.

- Ouput Layer: The Output Layer consists of units with the same number as the dimension of the action space. The role of the output layer is to produce a probability distribution across the various possible actions so that we can choose the action with the highest probability. Thus we use a Softmax as the activation function which can

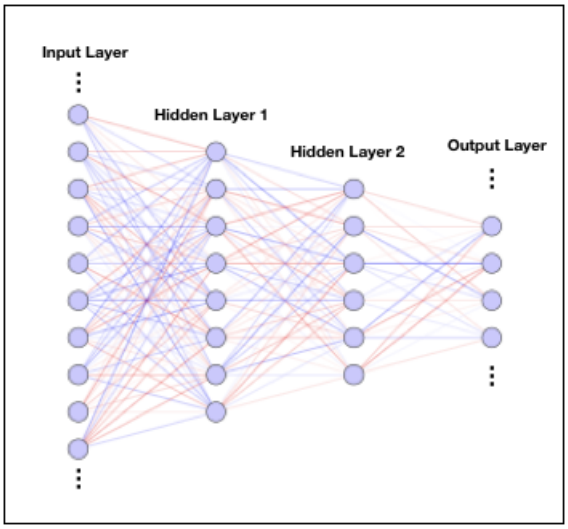


Figure 17: Actor Network

generate the required distribution across the $n$ actions in the action space. The Output Layer is also initialized with zero values for all its weights and biases.

- Loss Function: here we use the Mean Squared Error loss function, from Equation (4.7).
- Optimizer: For the learning algorithm we have used the Adaptive Moments Optimizer (Adam)[93] which is a special form of the SGD Algorithm where the learning rate adapts to the magnitude of the partial derivative of the cost function.

### 5.3.4 Critic Network

The Critic Network is a neural network that works to approximate the state-value function $\hat{v}_w(S)$ using parameters $w$ which represent its weights and biases. This can be found visually represented in the above figure. The Critic Network consists of Layers and Neurons, each with its own properties and functions. This is described below:

- Input Layer : The function of the Input layer here is to accept the current state $s_t$ of the dialog and feed it into the network. This layer consists of the same number of units as the dimension of the state space with the Input being the directly observable state. The Input Layer uses a Rectified Linear Unit for its activation and like the Actor Network, is initialized with zero values for all its weights and biases.

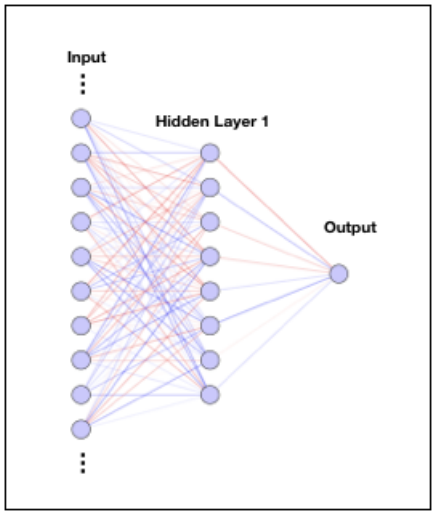


Figure 18: Critic Network

- Hidden Layer : The critic network since its job is simpler than the actor, uses only a single hidden layer with 16 Units. These units receive information from the Input layer. This single Layer has another ReLU as its activation and as before has all its parameters initialized with zeros.
- Output Layer : The Output layer of the critic network, since it has to produce a single value for the given state requires only a single neuron unit. The final magnitude that is produced by this isngle unit itself represents the *value* of the given state and thus requires no activation. Further, this layer is also initialized with zeros.
- Loss Function : In the Critic Network we again use the Mean Squared Error loss.
- Optimizer : The Critic Network learns by using the previously discussed SGD Algorithm from Section 4.7.2.

# 6 Experiments and Results

In this section we show how the implementation of the above discussed algorithms can be practical in developing simple dialog strategies. Further we discuss how these learning algorithms can be used together with a dialog MDP and a Simulated User to learn simple dialog strategies for the customer-support assistance domain. We also explore how the variation of learning parameters such as the discount rate affect the resultant learned strategy. The purpose of these experiments is to show that it is possible to develop intelligent dialog systems in this domain and to reach an understanding of how these dialog strategies are affected by the learning framework. If we can establish this foundation then we could in the future extend this more realistic and practical situations with extremely large or continuous state-action spaces.

## 6.1 Overview

## 6.2 Learning Simple Dialog Strategies

All learning algorithms and dialog environment/user simulation were all done using Python 3.6.6[94]. Further we used Google Tensorflow[24] and the Keras framework[95] to implement the neural network Approximators. These experiments have the following primary goals :

- To explore whether the implementation of the specific learning algorithms can indeed be useful and practical in its application to learning dialog strategies in this domain. Generally in the application of application of these algorithms the goal is generally easy to define but this is not the case with dialogs
- To explore how a variation in some learning parameters has an effect in the learning process, specifically the time to convergence on the learning curve and the quality of the resultant dialog strategy
- To compare the learning behavior of the various above algorithms compared to a baseline and compared to each other.

### 6.2.1 Experimental Setup

As discussed in Section 3.4 our process of modeling the *environment* is described as MDP which can be seen in 3. In dialogs modeled as a MDP, the observable state is directly the input received by the RL agent's *Policy*. From Figure 3 we can see that the entire system is composed of discrete states $S$, system actions $a$ and user utterances or responses $u$. The initial state of the system is always the same starting state that is automatically set

every time the environment is reset. From here the system can perform actions as described below. These actions are then fed into the Environment Simulation which includes the user. According to this action the environment then queries the simulated user for a response. The joint probabilities of the current state $S_t$ and the previous action $a_t$ give the probability of transition in to the next dialog state which is fed back to the RL agent. The state transition model can be seen in Figure 19. The simulation model is based on a simple state transition function of a probabilistic simulation environment similar to ones described in [13], [96] and [41]. Thus in our framework both the model of state transition and the simulation environment are described by a single transition probability. Here the probability of the next state is given by the joint probability of the observation $o_t$, action $a_t$ at time $t$ and the state $s_{t+1}$ at time $t$. The observation here is noisy user input. Thus we have as the general state transition :

$$P\big(s_{t+1}, o_t|s_t\big) = P\big(s_{t+1}|o_t, a_t, s_t\big) \cdot P\big(o_t|a_t, s_t\big) \cdot P\big(a_t|s_t\big) \tag{6.1}$$

Here we make the following assumptions

- We do not take into account noise that directly affects the choice of the action chosen. Thus the main job of the simulation becomes to evaluate the first and second term on the RHS of Eq 6.1 while the last term depends on the learnt policy of the system.
- We assume that the output of the simulated environment and observation are directly representative of the state transitions

Thus we have :

$$P\big(s_{t+1}|o_t, s_t\big) = P\big(s_{t+1}|a_t, s_t\big) = P^a_{s,s'} \tag{6.2}$$

Thus the probability encodes the transition probabilities of the underlying MDP that represents the entire dialog and we do not need a complex dialog task model interpretation that is usually seen in literature, where this is job of some function approximator, usually a Recurrent Neural Network that takes as input the observation and returns an internal *belief* state. This representation is also known as the Information State Paradigm [9] which is characteristic of working with a full MDP. Our environment as described above contains all components of the dialog system except the dialog management and learning module itself. This includes a probabilistic user model. Our simulated user is represented as a probabilistic model based on the recent interaction history. Thus the probability of generating user utterance $u_t$ at time $t$ is given by

$$\big(u_t|H_t\big) \tag{6.3}$$

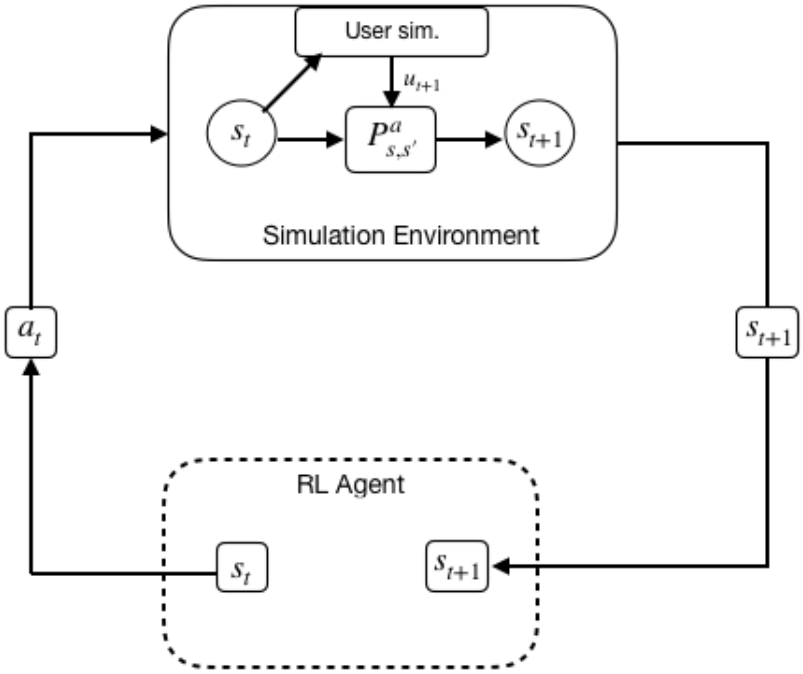


Figure 19: State Transition Model

Where $H_t$ represents the complete previous history of the dialog process. Here the history is given by the the previous system action and the previous dialog state which is representative of the chain of system actions and user utterances that occurred before the previous time step. Further we can assimilate the observation $o_t$ at time $t$ to be representative of this user response $u_t$ Thus our $P(o_t)$ from Eq. 6.2 becomes

$$P\big(o_t|a_t, s_t\big) = P\big(o_t|a_t, s_t, u_t\big).P\big(u_t|a_t, s_t\big) \tag{6.4}$$

Thus Equation 6.2 becomes:

$$\begin{aligned} P\big(s_{t+1}|o_t, s_t\big).P\big(o_t|a_t, s_t, u_t\big).P\big(u_t|a_t, s_t\big) &= P\big(s_{t+1}, o_t, u_t|a_t, s_t\big) \\ &= P\big(s_{t+1}|a_t, s_t\big) \end{aligned} \tag{6.5}$$

Finally the above equation 6.5 represents our Dialog System state transition function as well as the transition probabilities of the underlying MDP. Further we can see an example of the probabilities in Table 1. Here the topmost row represents the various discrete actions that the system is allowed to perform and the leftmost row represents the various states that the system can be in. The value for every state-action pair gives the probability of a *positive* user response. Thus $P(\bar{a}, s)$ gives the probability of a *negative* user response.

Table 1: User Simulation Model

| | a1 | a2 | a3 | a4 | a5 | b1 | b2 | b3 | b4 | b5 | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| s1 | 0.9 | 0.1 | 0.5 | 0.1 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| s2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.1 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| s3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.1 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| s4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.1 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| s5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.1 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| s6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.1 | 0.9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| s7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.2 | 0.9 |
| s8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.2 | 0.9 |
| s9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.2 | 0.9 |
| s10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.2 | 0.9 |
| s11 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.5 | 0.9 | 0.1 | 0.2 | 0.9 |
| $s_{Tp}$ | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| $s_{Tn}$ | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |

The purpose of this first learned strategy will be to to offer the user a product recommendation(for a customer) and a following opening question for the user to ask said customer, according to this the system then suggests a offer or deal for the customer following which the system confirms the customers interest and closes the dialog, in this order. Thus the actions of the system or the *Dialog Acts* can be in three separate classes corresponding to the above described stages of the conversation i.e. *open*, *offer* and *close*. This can be seen in Table 2. In the environment the actions are restricted to these three types i.e. *opening_question*, *offer_deal* and *close_deal*. The first action greets the user and offers a product to be recommended for his/her customer. If the customer responds positively then the system offers a deal for that customer, generally this would involve the querying of multiple knowledge sources. If this deal is accepted then the user can use a closing statement to confirm the deal and escalate the transaction to the billing department for example. The leftmost column in the table describes the type of action and the second column describes the number of action under this type. The actions are deigned to be favorable for different classes of users and the system is to learn to best sequences of actions to pick for a particular user class and recommendation. Further, as can be seen in Figure 5 the number of possible responses from the user is restricted to two i.e. a positive and negative utterance to keep the system simple and to convey the needs of the user to the system. To represent the end of the dialog we have implemented Initial and Terminal States apart from the dialog states associated with the actual conversation. From Figure 3 we can see that the system is always initialized to a starting state $S_0$ when the dialog starts and correspondingly the dialog ends when one of the two terminal states (represented by the bottom two boxes) are reached. These two terminal states represent respectively the situation where the dialog ended positively i.e. customer was interested in the offer and closed the deal or where the dialog ended negatively i.e. the customer was not interested in the recommendation or lost interest during the course of the dialog with the user. These two terminal states are represented by $S_{Tp}$ for a positive end to the episode and $S_{Tn}$ for a negative end. The final landing of the system in one these two terminal states also controls the rewards associated with each dialog episode. Here $T$ represents the final time-step of an episode i.e. each episode consists of time-steps $1, 2, 3, \ldots T$ Unless mentioned otherwise all experiments have the reward function described as:

$$R = \begin{cases} 1 & \text{if } S_T = S_{Tp} \\ 0 & \text{if } S_T = S_{Tn} \end{cases} \tag{6.6}$$

Thus the system get a positive unit reward if the final state is positive and no reward otherwise. Keeping the reward simple allows us to accurately define and measure the rate of task completion and thus optimize on this completion rate only which allows us to converge on the best sequence of system actions to which most users would respond positively. Here the task completion rate is the ratio of dialogs ending with $R = 1$.

Table 2: System Actions

| System Actions | # | Description | Example |
|---|---|---|---|
| opening_question | 5 | Opens the sales conversation by asking a product related sales question | Do you store many valuables at home? |
| offer_deal | 5 | Offers a customer and product specific deal | We offer one of the cheapest Household Liability Insurance on the market currently, you can offer a discount of 15% for the first 1 year. |
| closing_deal | 5 | Closes the deal and gets an idea of the escalation direction. Could also start a new dialog for another product | Is the customer interested in going forward or would he like to know more? |

### 6.2.2 Learning with A2C

The goal of this first learned policy is, as discussed above to greet and recommend a product and opening question to the user, followed by presenting a offer/deal and ending the dialog by closing said offer. A single experiment lasts 5000 episodes and the resulting rewards for each episode are collected and smoothed by taking the mean and displayed for comparison. This forms the learning curve for that algorithm. In order to clearly comprehend the various aspects of the A2C learning algorithm a comparison is made with two standard algorithms which we use as baselines. These are SARSA and Q-learning [39] and were chosen as they are the most commonly applied RL algorithms. Here the Q-learning algorithm is generally off-policy but can be used with either offline and online approaches. The SARSA algorithm is representative of an on-policy and online technique. The representation of the Q-function (state-value function) in this baseline implementation is linear and non-parameterized i.e. we do not use any function approximator such as a neural network but a simple table to hold Q-values. Please check Section 5.3 for a detailed view on the Advantage Actor Critic formulation.

#### 6.2.2.1 Results

The learning curves are formed by performing at least 5000 episodes and learning is evalu-

ated by taking the average return over at least 5 experiments. There are many parameters to be initialized, these are given below:

1. SARSA Hyperparameters:
    - $\epsilon$-Greedy value = 0.1
    - Discount Factor $\gamma_{SARSA} = 0.98$
    - Learning Rate $\alpha_{SARSA} = 0.01$
2. Q-learning Hyperparameters:
    - $\epsilon$-Greedy value = 0.1
    - Discount Factor $\gamma_{Q-learning} = 0.98$
    - Learning Rate $\alpha_{Q-learning} = 0.01$
3. A2C Hyperparameters:
    - Discount Factor $\gamma_{A2C} = 0.98$
    - Actor Network Learning Rate $\alpha_{Actor} = 0.01$
    - Critic Network Learning Rate $\alpha_{Critic} = 0.01$
    - Actor Network Layers : 3 Hidden layers with 32, 64 and 32 units respectively.
    - Critic Network Layers : 1 Hidden layer with 64 units.

These parameters were defined as the result of a large amount of tuning, these parameters sets are based on two criteria, the system must reach highest possible returns and second the speed of learning. The above described parameter sets were found to be the best for the above described (Section 6.2.1) experimental setup.

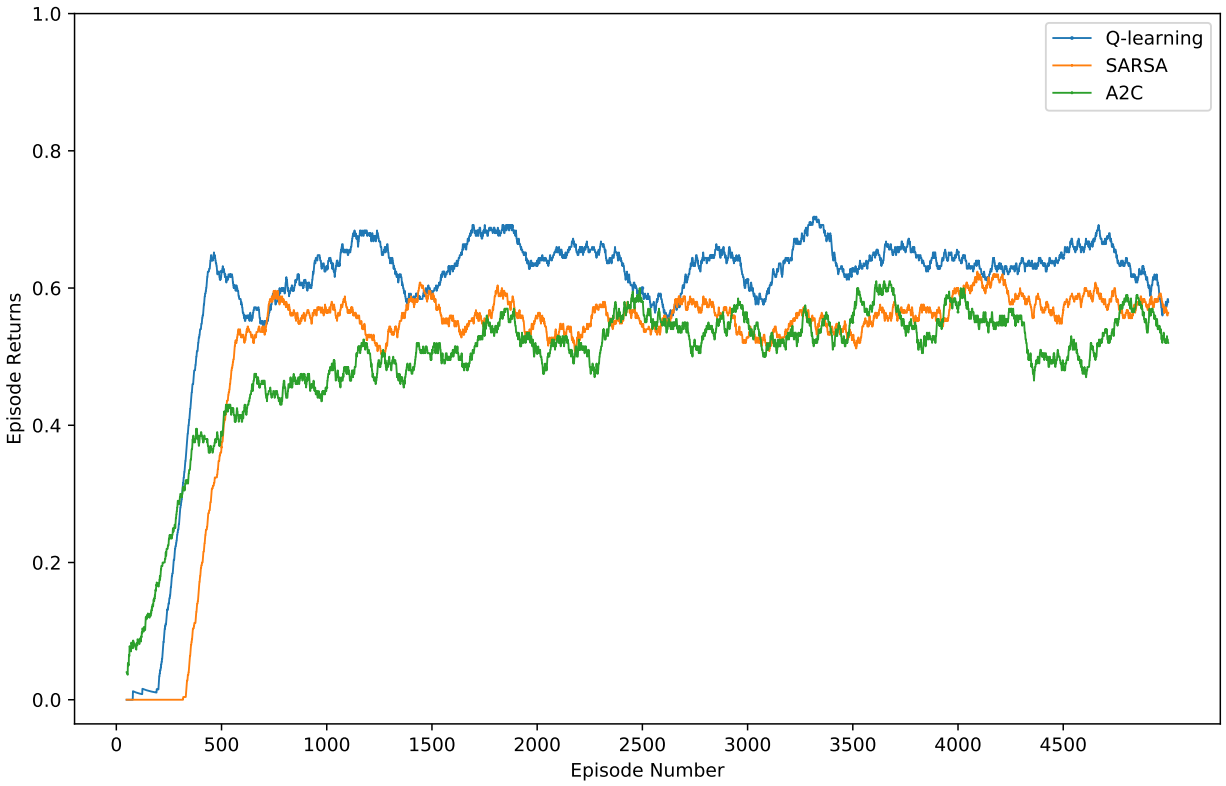


Figure 20: A2C Learning Curve

The learning curve for this experiment can be see in Figure 20. Here the the individual curves are smoothed by the method of moving averages with a rolling window of 200 data points. We can infer from the plot that the A2C algorithm successfully learns the required dialog strategy and converges to the maximum return path or optimal policy after around 500 episodes. Compared to the two baseline algorithms, it seems that A2C converges faster with fewer training episodes. First, from the graph we can see that the A2C algorithm is the first to hit the maximum return for an episode after just around 50 episodes and its curve immediately starts to improve as the returns from these steps are incorporated into the policy and value networks. In contrast we can see that Q-Learning and SARSA take at least 250 - 750 episodes to start improving, presumably this could be due to the $\epsilon$-greedy

behavior in these algorithms that restrict the exploration phase and extend the exploitation phase. The returns during these first 500 episodes for SARSA and Q-learning, would have been 0 but the system learnt that this must be most optimal. Due to this and the low rate of $\epsilon$ most greedy actions did not result in an improvement of the returns. This changed when $\epsilon$ finally allowed the system to explore the paths with most return around episode 500. This problem was avoided with A2C as it, according to the input state, chooses actions directly from a probability distribution across the different actions i.e. softmax output from the Actor/Policy Network and does not follow $\epsilon$-greedy behavior. Further, the fluctuations that can be seen in SARSA and Q-learning where the curve takes a periodic dip in the return can also be attributed to the forced exploration that $\epsilon$ controls which would disturb the policy from picking the action. Further, we can infer from the graph that for relatively small state-action spaces, tabular methods such as SARSA and Q-Learning perform almost 15 - 20% better after 1000 episodes than a parameterized policy method such as A2C. The sudden increase in performance of SARSA and Q-learning between 250 to 500 episodes shows that value based methods are indeed more sample efficient than policy methods.

#### 6.2.2.2 Effect of the Discount Factor

For the A2C Algorithm and typically in RL we have an addition hyperparameter called the *discount factor* denoted by $\gamma$. The system is simulated using a few variants of $\gamma$ while keeping the other parameters such as the actor learning rate $\alpha_{Actor}$, critic learning rate $\alpha_{Critic}$ and the number of layers in both networks constant.

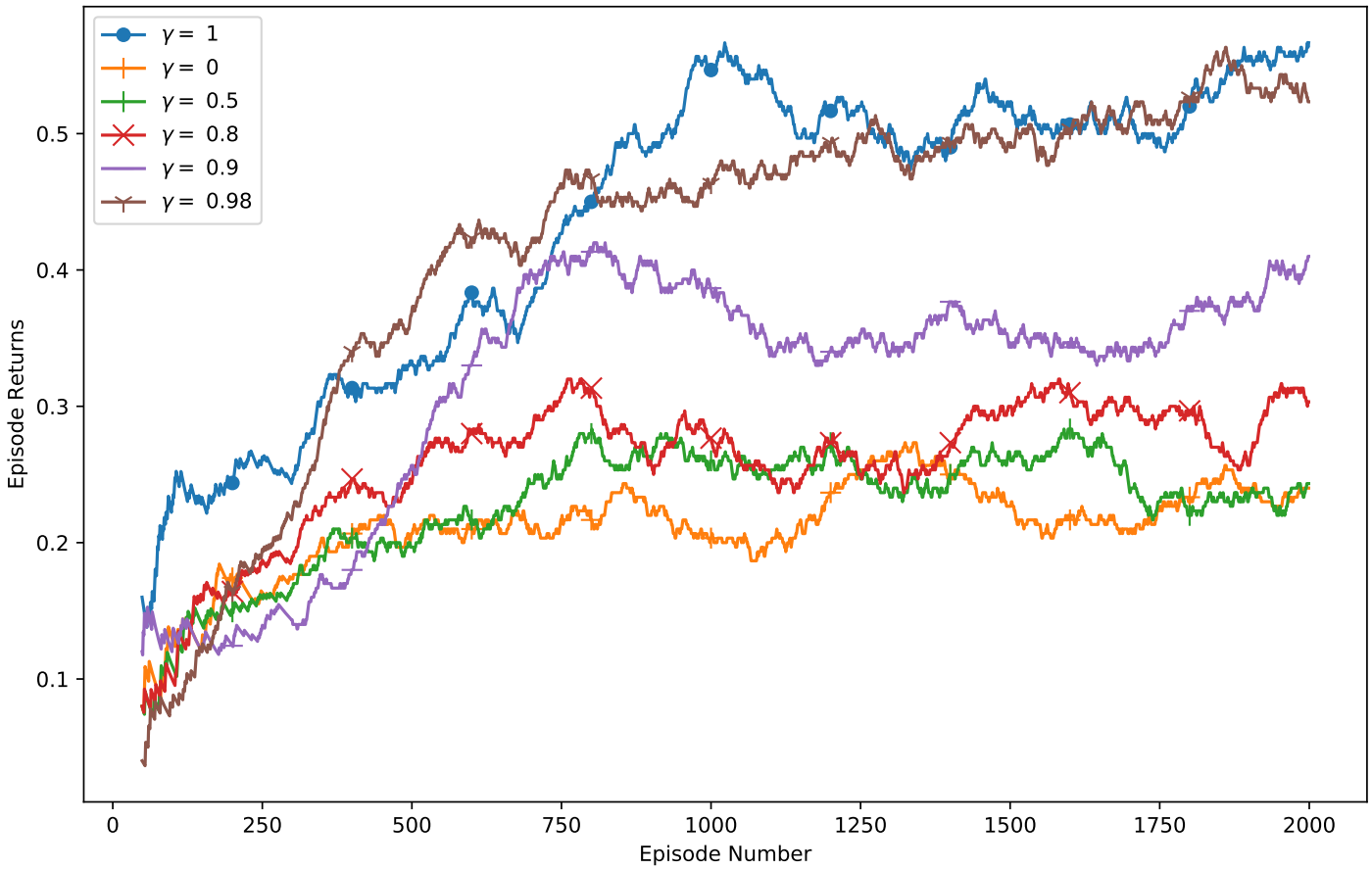


Figure 21: Effect of the Discount Factor on A2C Learning

The influence of the discount factor also known as the forgetting factor $\gamma$ can be seen in Figure 21 where the smoothed average episodic return is plotted against the number of episodes. The amount of information in each update of the Actor and Critic Network weights is directly influenced by the discount factor $\gamma$. Intuitively this means the agents behavior is affected by making it prefer long-term rewards than short-term or even immediate rewards. This becomes important in our scenario where the agent only receives a reward at the final step of the episode and so must optimize the preceding steps so as to reach the correct final

step. If $\gamma = 0$ or close to 0 then this means that the agent is very *short-sighted* and does not consider any rewards other than the most immediate one, in contrast if $\gamma = 1$ then the agent considers all rewards that it may receive in the following steps. This is observed in the given diagram, where the performance beings to degrade as we bring the discount factor closer to 0. We can also notice that the discount factor does not seem to have a large affect on the speed of convergence since the gradient of the best performing curve ($\gamma > 0.9$) converges after 1000 episodes so does the worst performing curve ($\gamma \leq 0.5$) around the same episode. Another interesting phenomenon is that the discount rate seems to critically effect a rise in performance only on the upper bound close to 1 or $\gamma > 0.9$, this can be seen the curves for $\gamma \leq 0.8$ which seem to show a low but similar performance with respect to each other. As we move $\gamma$ upward by just 0.1 we get almost a two fold increase in performance. This again can be attributed to the fact that we have very delayed rewards in our environment and only at the last step, this would require the agent to be very long-sighted to be able to push the policy gradient toward this maximum return region that would only occur 3 or 4 time-steps following the starting state if following an optimal path.

#### 6.2.2.3 Effect of Reward Magnitude

This third set of experiments is to examine how varying the absolute magnitude of the reward $R$. The system was simulated using the same reward structure i.e. one positive final reward signal at the end of each successful episode but varying the absolute magnitude of this reward unit by a factor of 10. Thus we tested the learning curve for $R = 1, 10, 10^2, 10^3$ and $10^4$.To do this we kept all other hyperparameters such as the actor learning rate $\alpha_{Actor}$, critic learning rate $\alpha_{Critic}$ , discount factor $\gamma$, $\epsilon$ and the number of layers in both networks as constant. The effects of varying the reward in this way can be seen in Figure 22 where we plot the absolute magnitude in the log scale on the y axis vs the number of episodes on x. The curves have been smoothed by using the method of moving averages with a rolling window of 200 data points. Each experiment ran for 5000 episodes and the resultant learning curve was built from the average of 5 repetitions.

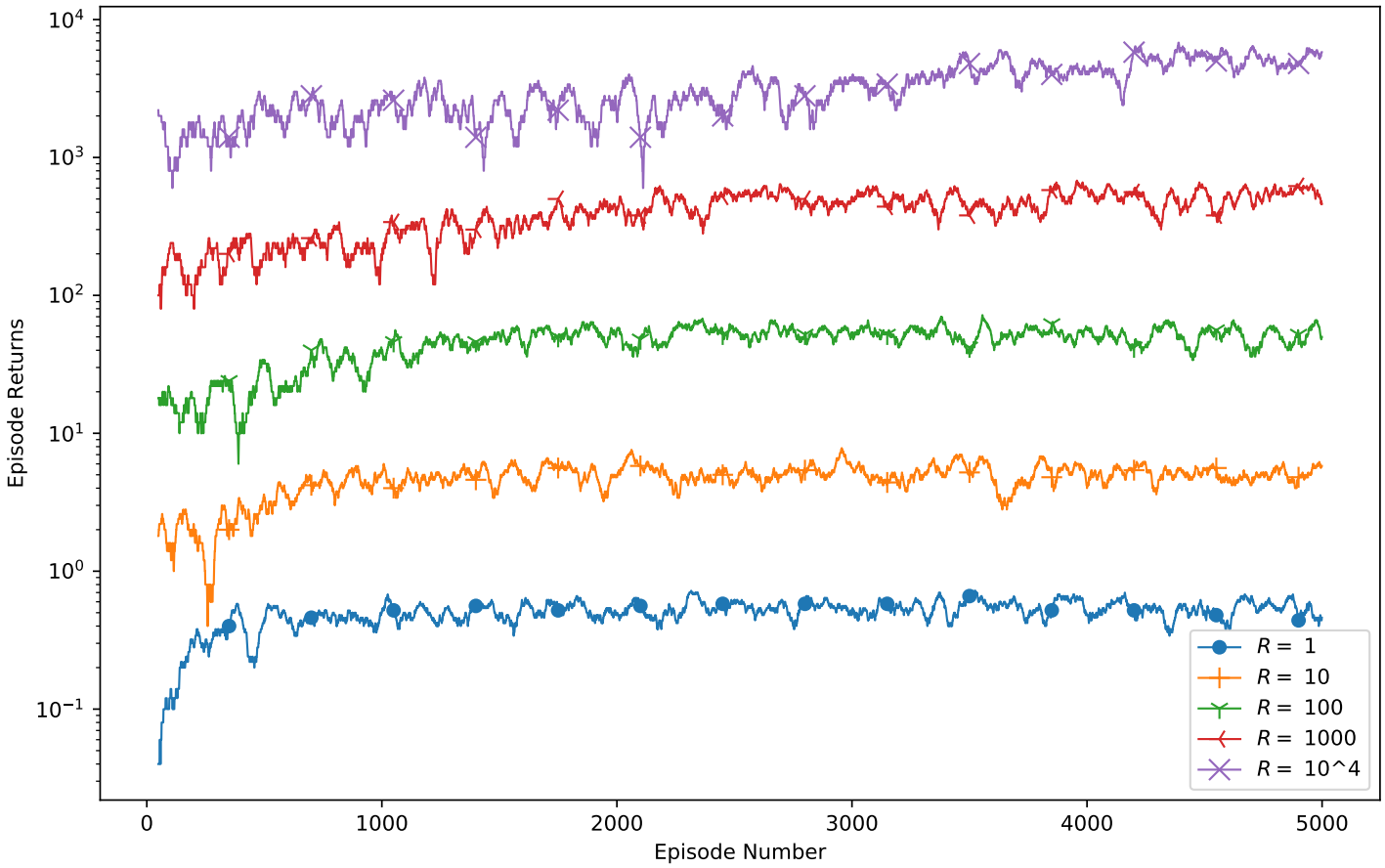


Figure 22: Effect of the Absolute Reward Magnitude on A2C Learning

According to the learning curves it is observed that reward magnitude does not significant seem to affect the performance of the final learnt dialog but does seem to have a small effect on the learning speed. As we can see the gradient of the curve for $R = 1$ converges after about 400 episodes, but as we increase the magnitude of the reward 10 fold we can see that, for the same performance the convergence now requires around 900 episodes. Similar behavior can be seen in the other curves with $R \geq 100$ where we now need atleast $1000 - 1200$ episodes to converge. This shows that that lower rewards are a little bit more sensitive to updates,

but this could also be due to statistical fluctuations.

### 6.2.3 Learning with REINFORCE

Similar to the experiments conducted with the A2C Algorithm, here we attempt to use the REINFORCE algorithm to learn the dialog strategy discussed in Section 6.2.1. Here each experimental run lasted 5000 episodes with the total returns from each episode being collected to form the learning curve shown below. To gain a better understanding of learning with REINFORCE we again use SARSA and Q-learning[39] as baselines. As we know the SARSA is online while Q-learning is offline and uses a simple table to hold the Q values. Please see Section 5.2.1 for a detailed explanation of the REINFORCE process. it should be noted here that for these set of experiments we modify the reward function from Equation 6.6 to

$$R = \begin{cases} 10 & \text{if } S_T = S_{Tp} \\ 0 & \text{if } S_T = S_{Tn} \end{cases} \tag{6.7}$$

#### 6.2.3.1 Results

The Hyperparameters for the experiments were initialized as shown below.

1. SARSA Hyperparameters:
   - $\epsilon$-Greedy value = 0.2
   - Discount Factor $\gamma_{SARSA} = 0.9$
   - Learning Rate $\alpha_{SARSA} = 0.01$
2. Q-learning Hyperparameters:
   - $\epsilon$-Greedy value = 0.2
   - Discount Factor $\gamma_{Q-learning} = 0.9$
   - Learning Rate $\alpha_{Q-learning} = 0.01$
3. REINFORCE Hyperparameters:
   - Discount Factor $\gamma_{reinforce} = 0.9$
   - Policy Network Learning Rate $\alpha_{Actor} = 0.01$
   - Policy Network Layers : 2 Hidden layers with 10 and 10 units respectively.

Based on a large number of training iterations and consequent tuning, the above parameters were found to produce the highest returns and fastest training times. Thus they were found to be the best set of parameters for the above described experimental setup (Section 6.2.1).

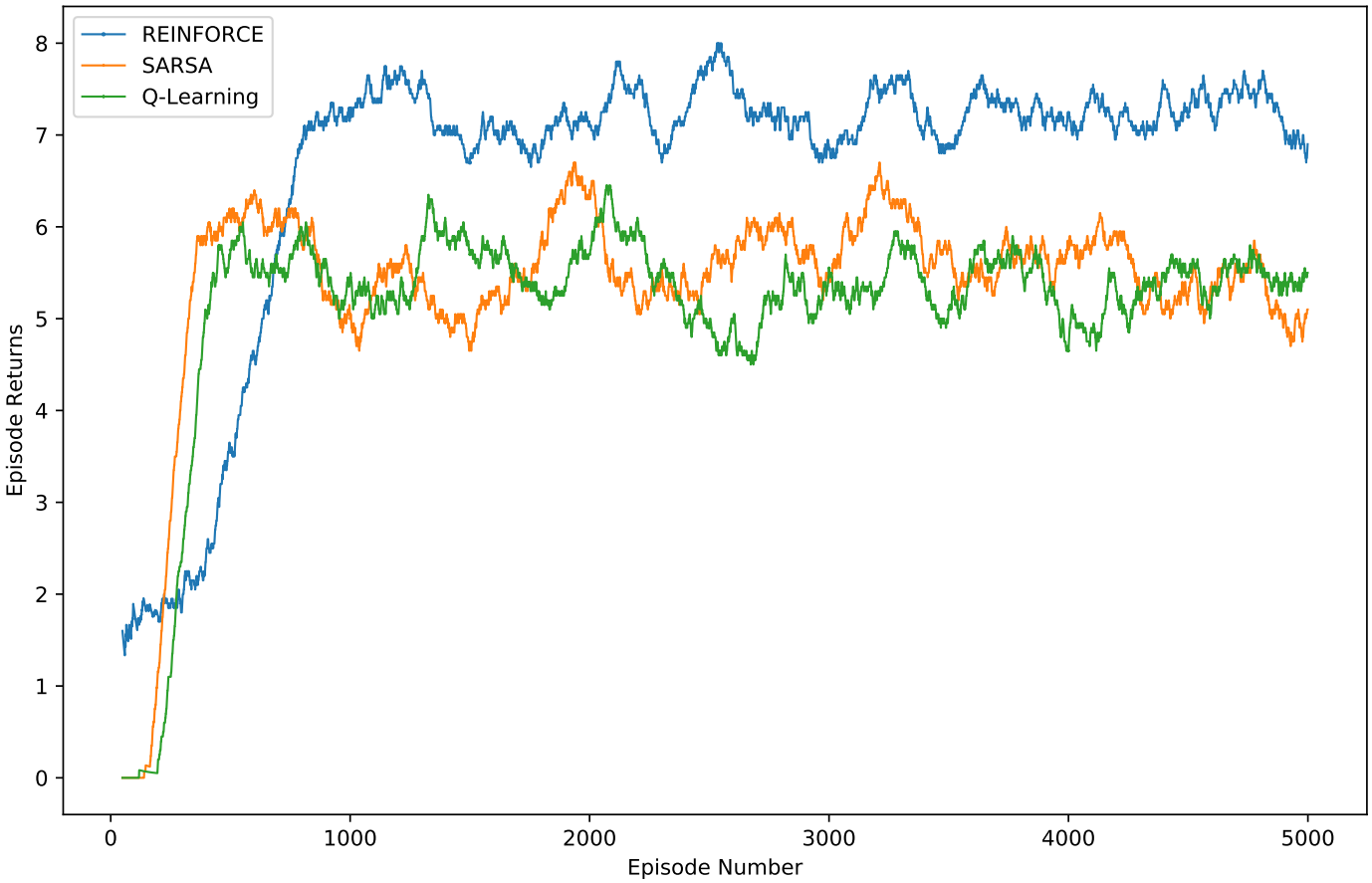


Figure 23: REINFORCE Learning Curve

The learning curves are plotted in Figure 23. Here each curve was smoothed by the method of moving averages with a rolling window of 200 data points. From first glance we can see that REINFORCE does indeed converge to a policy with highest returns after 800

episodes. When we compare it to the two baselines we can see that it shows a much higher performance but slower training times. We can also see that REINFORCE is the last to converge while Q learning and SARSA converge faster, taking only about 300 episodes to do so. This is because *vanilla* Policy Gradient methods such as REINFORCE, suffer from a high degree of variance. This variance stems from the fact that the policy update defined by $\nabla_\theta J(\theta)$ varies depending on the trajectories or path the system takes through the MDP. From Equation 5.1 we have $\nabla_\theta J(\theta) = \mathbb{E}_\pi\big[\sum_a q_\pi\big(s_t, a\big)\nabla_\theta \pi_\theta\big(a|s_t\big)\big]$. Here we can see that both the terms are dependent on the *current* action and state from the path that the system is following. In REINFORCE recall that we compute this gradient by sampling trajectories from the policy i.e. $a \sim \pi_\theta(s)$ and then average over them to get the expectation (for each episode). Thus the gradient is dependent on the current actions and states. This causes high variance as these trajectories can take very different paths which means that the two terms in the above equation $q_\pi\big(s_t, a\big)$ and $\nabla_\theta \pi_\theta\big(a|s_t\big)$ can take very different values according to the path it follows. This makes it is difficult for this algorithm to find the value of its states and affects the performance. Once it has visited states of relatively high value(local optima) it repeatedly optimizes the same policy without further exploration of the state-action space, this is the cause behind the better performance of REINFORCE. SARSA and Q learning on the other hand could depend on the values they assigned to the state action pairs as these algorithms optimize the value function itself and not the policy as REINFORCE. The $\epsilon$-greedy process let the baselines to explore further and reach the states with associated with the highest returns after about 500 episodes. Once these states were assigned high values, due to the high return, they were easily able to converge to the optimal path by repeatedly visiting those states. Thus we can infer from the plot that for small state-action spaces and without an approach to reduce the variance for REINFORCE such as using an intrinsic baseline (such as A2C), tabular methods such as SARSA and Q-learning converge faster but REINFORCE shows a much higher performance for the same task.

## 6.3 Summary

In summary we can say confidently that both the REINFORCE and Advantage Actor-Critic Algorithms can be used to learn dialog strategies for our domain, where our environment is modeled as a full MDP. In these experiments we saw that the learned policies shows behaviors that are fully in conjunction with our simple simulated environment and user. Thus these experiments have been fundamental in helping us understand and answer questions of what can be done with the employed algorithms and how we could go about using and applying them to domain specific features and goals. However, in this setup the state-action space is very small and limited. Now we need to scale up to more realistic scenarios that require a Partially Observable MDP and much larger state-action spaces.

# 7 Conclusions

## 7.1 Conclusion

In this work we have seen that RL provides an excellent framework for general decision making tasks. Further, we have seen that this is useful for the development of automated dialog policy learning systems. In this thesis we have shown that these two algorithms can address the challenge of learning dialog that is designed as complete Markov Decision Process. In general with this work we have also explored in-depth the concept of Domain Specific or Goal-Oriented dialog systems which are based on Reinforcement Learning. Further we examine the fundamental concepts involved in RL such as a MDP and the various basic methods and techniques to solve them. We have also studied and shown the required mathematical tools and structures such as MLPs, loss functions and deep learning in general to help us overcome the challenges present in developing dialog systems. Further, We show that the Policy Based RL methods such as REINFORCE and Advantage Actor-Critic can indeed be used in the developing of Goal-Oriented Dialog Systems in the domain of sales assistance and explore in-depth the working behind these algorithms and how we would implement these algorithms using deep neural networks. With repeated experimentation we also give the best set of hyperparameters that encourage learning of optimal dialog policies using these algorithms. Finally we show the advantages in using these algorithms and the difference in learning behaviors compared to standard baselines such as SARSA and Q-Learning.

## 7.2 Future Work

Based on the work done in this thesis and the consequent conclusions above we discuss a few further possibilities and future extensions :

- Extend the environment to include noisy real user input. This would introduce partial observability of our currently full MDP and also a Natural Language Understanding Unit due to the need to recognize and classify input speech dialog acts which would also introduce speech recognition errors due to the noise in user input and the quality of the NLU module. From here one would have to model the environment as a POMDP[39]. This stems from the fact that the dialog manager (which includes the NLU) may not be able fully understand what is input by a user, and thus the dialog states must be made more complex, perhaps with a so called *confidence score* for different dialog state features to differentiate the multitude of states. This is also describes above as Discriminative dialog state tracking.

- Since our User simulation was on based a simple joint probability based on the history

of the dialog, future work should extend this to include multiple complex users each with the their own goals and agenda. Perhaps using a complex probabilistic model based on Hidden Markov Models.

- Extend to a more complex reward function that better encodes the true rewards in more realistic human machine interaction situations. For tasks such as our which aim to assist the user with some goal the reward function can be extended to be more appropriate learning the required policies, step rewards for positive user utterances, punishing extended dialog with a lot of turns etc. Further the complex reward modeling for these tasks has received relatively less research attention and is thus remains a big challenge for developing robust dialog systems. Thus it may be important to continue research in the direction of complex reward modeling.

- Testing the use of further sophisticated RL approaches such as Deep Q Networks, Generative Adversarial Networks and Deep Deterministic Policy Gradients in the development of robust dialog systems

# 8 Bibliography

# References

[1] B. Macwhinney, "Language evolution and human development," 12 2008.

[2] A. M. TURING, "I.computing machinery and intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, 1950.

[3] R. Kurzweil, *The Singularity Is Near: When Humans Transcend Biology*. Penguin (Non-Classics) 2006, 2006.

[4] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," *Commun. ACM*, vol. 9, pp. 36–45, Jan. 1966.

[5] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, "Neural collaborative filtering," *CoRR*, vol. abs/1708.05031, 2017.

[6] C. B. Frey and M. A. Osborne, "The future of employment: how susceptible are jobs to computerisation?," *Technological forecasting and social change*, vol. 114, pp. 254–280, 2017.

[7] R. Porzel and M. Baudis, "The tao of chi: Towards effective human-computer interaction," in *HLT-NAACL*, 2004.

[8] V. Zue, S. Seneff, J. R. Glass, J. Polifroni, C. Pao, T. J. Hazen, and L. Hetherington, "Juplter: a telephone-based conversational interface for weather information," *IEEE Transactions on Speech and Audio Processing*, vol. 8, pp. 85–96, Jan 2000.

[9] S. Larsson and D. R. Traum, "Information state and dialogue management in the trindi dialogue move engine toolkit," *Nat. Lang. Eng.*, vol. 6, pp. 323–340, Sept. 2000.

[10] R. Bellman, "A markovian decision process," vol. 6, p. 15, 04 1957.

[11] E. Levin, R. Pieraccini, and W. Eckert, "Learning dialogue strategies within the markov decision process framework," pp. 72 – 79, 01 1998.

[12] S. Young, "Probabilistic methods in spoken dialogue systems," *Philosophical Transactions of the Royal Society (Series A*, vol. 358, pp. 1389–1402, 1999.

[13] S. P. Singh, M. J. Kearns, D. J. Litman, and M. A. Walker, "Reinforcement learning for spoken dialogue systems," in *Advances in Neural Information Processing Systems 12* (S. A. Solla, T. K. Leen, and K. Müller, eds.), pp. 956–962, MIT Press, 2000.

[14] O. Pietquin and T. Dutoit, "A probabilistic framework for dialog simulation and optimal strategy learning," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 14, pp. 589–599, March 2006.

[15] B. Dhingra, L. Li, X. Li, J. Gao, Y.-N. Chen, F. Ahmed, and L. Deng, "End-to-end reinforcement learning of dialogue agents for information access," *CoRR*, vol. abs/1609.00777, 2016.

[16] H. Cuayáhuitl, "Simpleds: A simple deep reinforcement learning dialogue system," in *IWSDS*, 2016.

[17] X. Li, Y. Chen, L. Li, and J. Gao, "End-to-end task-completion neural dialogue systems," *CoRR*, vol. abs/1703.01008, 2017.

[18] D. Z. Hakkani-Tür, G. Tür, A. elikyilmaz, Y.-N. Chen, J. Gao, L. Deng, and Y.-Y. Wang, "Multi-domain joint semantic frame parsing using bi-directional rnn-lstm," in *INTERSPEECH*, 2016.

[19] T. Wen, M. Gasic, N. Mrksic, P. Su, D. Vandyke, and S. J. Young, "Semantically conditioned lstm-based natural language generation for spoken dialogue systems," *CoRR*, vol. abs/1508.01745, 2015.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] J. Schatzmann and S. Young, "The hidden agenda user simulation model," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 17, pp. 733–747, May 2009.

[22] X. Li, Z. C. Lipton, B. Dhingra, L. Li, J. Gao, and Y. Chen, "A user simulator for task-completion dialogues," *CoRR*, vol. abs/1612.05688, 2016.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray,

B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014.

[26] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," 2015.

[27] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[28] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "Weakly supervised memory networks," *CoRR*, vol. abs/1503.08895, 2015.

[29] facebook research, "The babi project," *BABL*, 2017.

[30] A. Bordes and J. Weston, "Learning end-to-end goal-oriented dialog," *CoRR*, vol. abs/1605.07683, 2016.

[31] P. Su, M. Gasic, N. Mrksic, L. M. Rojas-Barahona, S. Ultes, D. Vandyke, T. Wen, and S. J. Young, "Continuously learning neural dialogue management," *CoRR*, vol. abs/1606.02689, 2016.

[32] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015.

[33] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, (Cambridge, MA, USA), pp. 1057–1063, MIT Press, 1999.

[34] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, pp. 293–321, May 1992.

[35] J. D. Williams, K. Asadi, and G. Zweig, "Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning," *CoRR*, vol. abs/1702.03274, 2017.

[36] H. Chen, X. Liu, D. Yin, and J. Tang, "A survey on dialogue systems: Recent advances and new frontiers," *CoRR*, vol. abs/1711.01731, 2017.

[37] R. Pieraccini and J. M. Huerta, "Where do we go from here ? research and commercial spoken dialog systems," 2006.

[38] M. Henderson, "Machine learning for dialog state tracking: A review," in *Proceedings of The First International Workshop on Machine Learning in Spoken Language Processing*, 2015.

[39] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[40] S. J. Young, J. Schatzmann, K. Weilhammer, and H. Ye, "The hidden information state approach to dialog management," *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, vol. 4, pp. IV–149–IV–152, 2007.

[41] N. Roy, J. Pineau, and S. Thrun, "Spoken dialogue management using probabilistic reasoning," in *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pp. 93–100, Association for Computational Linguistics, 2000.

[42] J. Henderson and O. Lemon, "Mixture model pomdps for efficient handling of uncertainty in dialogue management," in *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, (Stroudsburg, PA, USA), pp. 73–76, Association for Computational Linguistics, 2008.

[43] S. Young, M. Gašić, S. Keizer, F. Mairesse, J. Schatzmann, B. Thomson, and K. Yu, "The hidden information state model: A practical framework for pomdp-based spoken dialogue management," *Comput. Speech Lang.*, vol. 24, pp. 150–174, Apr. 2010.

[44] J. D. Williams, "A critical analysis of two statistical spoken dialog systems in public use," in *2012 IEEE Spoken Language Technology Workshop (SLT)*, pp. 55–60, Dec 2012.

[45] S. Lee, "Structured discriminative model for dialog state tracking," in *SIGDIAL Conference*, 2013.

[46] M. Henderson, B. Thomson, and S. J. Young, "Word-based dialog state tracking with recurrent neural networks," in *SIGDIAL Conference*, 2014.

[47] M. Henderson, B. Thomson, and S. Young, "Robust dialog state tracking using delexicalised recurrent neural networks and unsupervised adaptation," 12 2014.

[48] N. Mrksic, D. Ó. Séaghdha, T. Wen, B. Thomson, and S. J. Young, "Neural belief tracker: Data-driven dialogue state tracking," *CoRR*, vol. abs/1606.03777, 2016.

[49] H. Chen, X. Liu, D. Yin, and J. Tang, "A survey on dialogue systems: Recent advances and new frontiers," *CoRR*, vol. abs/1711.01731, 2017.

[50] M. Gai, D. Kim, P. Tsiakoulis, C. Breslin, M. Henderson, M. Szummer, B. Thomson, and S. Young, "Incremental on-line adaptation of pomdp-based dialogue managers to extended domains," pp. 140–144, 01 2014.

[51] R. Lpez-Czar, . Torre, J. Segura, A. Rubio, and V. Sanchez, "Testing dialogue systems by means of automatic generation of conversations," vol. 14, 10 2002.

[52] J. Schatzmann, K. Weilhammer, M. Stuttle, and S. Young, "A survey of statistical user simulation techniques for reinforcement-learning of dialogue management strategies," *Knowl. Eng. Rev.*, vol. 21, pp. 97–126, June 2006.

[53] G. Chung, "Developing a flexible spoken dialog system using simulation," in *Proceedings of the 42Nd Annual Meeting on Association for Computational Linguistics*, ACL '04, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2004.

[54] R. López-Cózar, Z. Callejas, and M. Mctear, "Testing the performance of spoken dialogue systems by means of an artificially simulated user," *Artif. Intell. Rev.*, vol. 26, pp. 291–323, Dec. 2006.

[55] H. Cuayáhuitl, S. Renals, O. Lemon, and H. Shimodaira, "Learning multi-goal dialogue strategies using reinforcement learning with reduced state-action spaces," in *INTERSPEECH*, 2006.

[56] E. Levin, R. Pieraccini, and W. Eckert, "A stochastic model of human-machine interaction for learning dialog strategies," *IEEE Transactions on Speech and Audio Processing*, vol. 8, pp. 11–23, Jan 2000.

[57] J. D. Williams, "A method for evaluating and comparing user simulations: The cramr-von mises divergence," in *2007 IEEE Workshop on Automatic Speech Recognition Understanding (ASRU)*, pp. 508–513, Dec 2007.

[58] K. Georgila, J. Henderson, and O. Lemon, "User simulation for spoken dialogue systems: learning and evaluation," in *INTERSPEECH*, 2006.

[59] H. Cuayáhuitl, S. Renals, O. Lemon, and H. Shimodaira, "Human-computer dialogue simulation using hidden markov models," *IEEE Workshop on Automatic Speech Recognition and Understanding, 2005.*, pp. 290–295, 2005.

[60] K. Scheffler and S. Young, "Corpus-based dialogue simulation for automatic strategy learning and evaluation," pp. 64–70, 07 2001.

[61] J. Schatzmann, B. Thomson, and S. Young, "Statistical user simulation with a hidden agenda," 2007.

[62] K. Georgila, A. Tsopanoglou, N. Fakotakis, and G. Kokkinakis, "An integrated dialogue system for the automation of call centre services," in *Fifth International Conference on Spoken Language Processing*, 1998.

[63] J. Y. Chai, N. Kambhatla, and W. Zadrozny, "Natural language sales assistant-a web-based dialog system for online sales.,"

[64] P.-h. Su, Y.-B. Wang, T.-h. Yu, and L.-s. Lee, "A dialogue game framework with personalized training using reinforcement learning for computer-assisted language learning," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8213–8217, IEEE, 2013.

[65] M. A. Walker, "An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email," *CoRR*, vol. abs/1106.0241, 2011.

[66] S. Shriver, A. Toth, X. Zhu, A. Rudnicky, and R. Rosenfeld, "A unified design for human-machine voice interaction," in *CHI '01 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '01, (New York, NY, USA), pp. 247–248, ACM, 2001.

[67] J. F. Allen, B. W. Miller, E. K. Ringger, and T. Sikorski, "A robust system for natural spoken dialogue," *CoRR*, vol. cmp-lg/9606023, 1996.

[68] G. Ferguson, J. F. Allen, *et al.*, "Trips: An integrated intelligent problem-solving assistant,"

[69] E. Levin and R. Pieraccini, "A stochastic model of computer-human interaction for learning dialogue strategies," in *In EUROSPEECH 97*, pp. 1883–1886, 1997.

[70] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[71] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[72] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[73] S. Adam, L. Busoniu, and R. Babuska, "Experience replay for real-time reinforcement learning control," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 2, pp. 201–212, 2012.

[74] M. Verleysen and D. François, "The curse of dimensionality in data mining and time series prediction," in *International Work-Conference on Artificial Neural Networks*, pp. 758–770, Springer, 2005.

[75] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[76] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *European Conference on Machine Learning*, pp. 280–291, Springer, 2005.

[77] A. Bordes, Y.-L. Boureau, and J. Weston, "Learning end-to-end goal-oriented dialog," *arXiv preprint arXiv:1605.07683*, 2016.

[78] C.-W. Liu, R. Lowe, I. V. Serban, M. Noseworthy, L. Charlin, and J. Pineau, "How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation," *arXiv preprint arXiv:1603.08023*, 2016.

[79] J. D. Williams and S. Young, "Partially observable markov decision processes for spoken dialog systems," *Comput. Speech Lang.*, vol. 21, pp. 393–422, Apr. 2007.

[80] O. Lemon, K. Georgila, and J. Henderson, "Evaluating effectiveness and portability of reinforcement learned dialogue strategies with real users: the talk towninfo evaluation," in *Spoken Language Technology Workshop, 2006. IEEE*, pp. 178–181, IEEE, 2006.

[81] M. A. Walker, D. J. Litman, C. A. Kamm, and A. Abella, "Paradise: A framework for evaluating spoken dialogue agents," in *Proceedings of the eighth conference on European chapter of the Association for Computational Linguistics*, pp. 271–280, Association for Computational Linguistics, 1997.

[82] L. B. Larsen, "Issues in the evaluation of spoken dialogue systems using objective and subjective measures," in *Automatic Speech Recognition and Understanding, 2003. ASRU'03. 2003 IEEE Workshop on*, pp. 209–214, IEEE, 2003.

[83] M. J. Mataric, "Reward functions for accelerated learning," in *Machine Learning Proceedings 1994*, pp. 181–189, Elsevier, 1994.

[84] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *ICML*, vol. 99, pp. 278–287, 1999.

[85] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[86] Y. LeCun and Y. Bengio, "The handbook of brain theory and neural networks," ch. Convolutional Networks for Images, Speech, and Time Series, pp. 255–258, Cambridge, MA, USA: MIT Press, 1998.

[87] K. Jarrett, K. Kavukcuoglu, Y. LeCun, *et al.*, "What is the best multi-stage architecture for object recognition?," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2146–2153, IEEE, 2009.

[88] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, p. 3, 2013.

[89] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[90] A. Cauchy, "Méthode générale pour la résolution des systemes déquations simultanées,"

[91] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, p. 533, 1986.

[92] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Machine Learning*, pp. 229–256, 1992.

[93] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[94] Python Core Team, *Python: A dynamic, open source programming language*. Python Software Foundation, 2015.

[95] F. Chollet, "keras." `https://github.com/fchollet/keras`, 2015.

[96] M. A. Walker, L. Hirschman, and J. S. Aberdeen, "Evaluation for darpa communicator spoken dialogue systems.," in *LREC*, Citeseer, 2000.