ABSTRACT

Title of dissertation: SEQUENTIAL DECISIONS AND PREDICTIONS IN NATURAL LANGUAGE PROCESSING

He He, Doctor of Philosophy, 2016

Dissertation directed by: Professor Hal Daumé III
Department of Computer Science

Natural language processing has achieved great success in a wide range of applications, producing both commercial language services and open-source language tools. However, most methods take a static or batch approach, assuming that the model has all information it needs and makes a one-time prediction. In this dissertation, we study dynamic problems where the input comes in a sequence instead of all at once, and the output must be produced while the input is arriving. In these problems, predictions are often made based only on partial information. We see this dynamic setting in many real-time, interactive applications. These problems usually involve a trade-off between the amount of input received (cost) and the quality of the output prediction (accuracy). Therefore, the evaluation considers both objectives (e.g., plotting a Pareto curve).

Our goal is to develop a formal understanding of sequential prediction and decision-making problems in natural language processing and to propose efficient solutions. Toward this end, we present meta-algorithms that take an existent batch model and produce a dynamic model to handle sequential inputs and outputs. We

build our framework upon theories of Markov Decision Process (MDP), which allows learning to trade off competing objectives in a principled way. The main machine learning techniques we use are from imitation learning and reinforcement learning, and we advance current techniques to tackle problems arising in our settings. We evaluate our algorithm on a variety of applications, including dependency parsing, machine translation, and question answering. We show that our approach achieves a better cost-accuracy trade-off than the batch approach and heuristic-based decision-making approaches.

We first propose a general framework for cost-sensitive prediction, where different parts of the input come at different costs. We formulate a decision-making process that selects pieces of the input sequentially, and the selection is adaptive to each instance. Our approach is evaluated on both standard classification tasks and a structured prediction task (dependency parsing). We show that it achieves similar prediction quality to methods that use all input, while inducing a much smaller cost. Next, we extend the framework to problems where the input is revealed incrementally in a fixed order. We study two applications: simultaneous machine translation and quiz bowl (incremental text classification). We discuss challenges in this setting and show that adding domain knowledge eases the decision-making problem. A central theme throughout the chapters is an MDP formulation of a challenging problem with sequential input/output and trade-off decisions, accompanied by a learning algorithm that solves the MDP.

SEQUENTIAL DECISIONS AND PREDICTIONS IN NATURAL LANGUAGE PROCESSING

by

He He

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Advisory Committee:
Professor Hal Daumé III, Chair/Advisor
Professor Jordan Boyd-Graber, Co-Advisor
Professor Philip Resnik
Professor Ramani Duraiswami
Professor Joelle Pineau

Dedication

To my loving parents.

Acknowledgments

I am very fortunate to have spent five lovely years at UMD. Throughout my journey in graduate school, many people have helped me mature in both academics and life. This dissertation would not have been possible without them.

First, I thank my advisors, Hal Daumé III and Jordan Boyd-Graber, for their patience and guidance. I not only benefited from insightful technical conversations with them, but also learned a lot through osmosis: They have continuously demonstrated how to discover interesting problems, how to bake ideas and how to ruthlessly question a work, especially that of oneself. Hal usually filled me with "crazy ideas" in each meeting, and I can only appreciate his foresight after weeks of delving into the topic. Jordan taught me critical thinking: one should not stop at the numbers but should go deep inside the model and expose what it is doing. I am grateful to Jason Eisner, with whom I closely worked in the first half of my graduate school. His academic rigor, enthusiasm, and precise writing gave me invaluable perspectives on research in my early days. Thank you, Jason, for pushing every piece of my work closer to perfection.

My committee members—the above-mentioned advisors, Philip Resnik, Ramani Duraiswami and Joelle Pineau—have been extremely supportive and provided insightful comments to make this dissertation deeper and more coherent. I want to especially thank Philip for establishing new connections to my work. My thanks also go to all other members of the CLIP faculty. I thank Marine Carpuat for giving me valuable career advice, and Naomi Feldman for organizing reading groups and paper

clinics. Outside of UMD, I learned a great deal from Lihong Li, Jason Williams, Paul Mineiro and Nikos Karampatziakis during my internships at Microsoft. I am also grateful for the chance of collaborating with John Langford and Kai-Wei Chang.

I am fortunate to be surrounded by a diverse and inspiring group of (former and current) students at CLIP. Many ideas in this dissertation were shaped and improved through conversations with my colleagues. I enjoyed intellectual discussions with Jiarong Jiang, Mohit Iyyer, Hua He, Naho Orita, Alvin Grissom II and John Morgan. I thank Greg Sanders, Abhishek Kumar, Jagadeesh Jagarlamudi, Amit Goyal, Arvind Agarwal, Taesun Moon, Dan Goldwasser, Sudha Rao and Hadi Amiri for the camaraderie. I found pleasure in spending time in the lab with Viet-An Nguyen, Thang Nguyen, Ning Gao, Mossaab Bagdouri, Yogarshi Vyas, Jinfeng Rao and Weiwei Yang.

Now, I would like to thank some dear friends, who aside from being colleagues—thus vital inspirations for this thesis—were also guides, confidants, comrades, and fellow travellers outside of work for these last five years: Anupam Guha, who pulled me out of my shell with candor, wisdom, and vibrant discussions on history and politics; Wu Ke, epitome of reason, with whom I enjoyed delicious Korean noodles and intriguing conversations; Yulu Wang, who showed me the art of empathy and the power of kindness; Snigdha Chaturvedi, who joined CLIP and Halfolks along with me and has been a source of constant support since then; Yuening Hu, Ke Zhai, Xi Chen and Chang Liu, who made my summer in Redmond full of laughter and adventure. In addition, thanks to Yue Dong, Xia Hu, and Rongkang Deng for being caring and considerate roommates.

Life at UMD would have been much more difficult without members of the administrative and technical staff. Jennifer Story has always been patient and helpful beyond the call of duty. Joe Webster has made resources available just as we needed it. Janice Perrone and Arlene Schenk have minimized bureaucracy when it comes to money. Thank you all.

Last but in no way the least, I thank my family who have given me unconditional love throughout my life and provided me the soil to grow. My parents, to whom my academic career owes greatly, have been supporting me wholeheartedly for my pursuit of dreams, even if that meant a distance of thousands of miles for many years. Words are not enough to express my gratitude for them. Finally, I thank Sheng Zha, who has always been there for me as an inspiring partner. Thank you for keeping my sanity in the ebb and flow of this journey, and more journeys to come. My harshest critic and my most faithful supporter, I thank you for steering me to a better self.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

NLP	Natural Language Processing
MDP	Markov Decision Process
RL	Reinforcement Learning
DAGGER	Dataset Aggregation
AGGREVATE	Aggregate Values to Imitate
MaxEnt	Maximum Entropy
PTB	Penn Treebank
MSTParser	Minimum-Spanning Tree Parser
POS	Part-of-Speech
MT	Machine Translation

Chapter 1: Introduction

Natural language processing (NLP) focuses on "static" problems where the model has all information it needs and makes a one-time prediction. For example, classifying a document after reading everything in it or translating a sentence after seeing all words of it. This thesis tackles "dynamic" problems where the input arrives sequentially, and outputs are required before all pieces of the input are in. This is important for two reasons. First, real-time applications often have a budget (time or money constraint), and a dynamic solver that works with a variable amount of information enables a cost-accuracy trade-off. Second, interactive applications require systems that respond in real time even with partial information. For instance, in simultaneous interpretation, translations must be generated before an utterance is finished; in robotic navigation, localization may be needed before the environment is fully sensed. In these situations, we must make decisions online adaptive to any information obtained so far.

We present a general framework based on Markov Decision Process (MDP) for dynamic problems in NLP. Unlike most prior statistical language learning methods, the contribution of this thesis is a *meta-algorithm* that works with various models developed for different task—e.g., dependency parsing, machine translation or

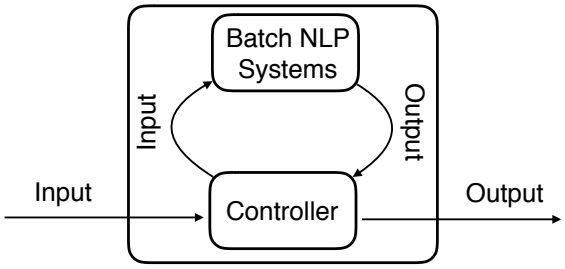


Figure 1.1: Overview of the dynamic system.

question answering— rather than a new model or a new estimation method. An overview of our approach is shown in Figure 1.1. Given an existing batch system, we learn a controller that interacts with it sequentially. The controller decides which piece of the input to feed the batch model at each step and when to generate output from the batch model as a prediction. Its novelty lies in the formulation of a sequential decision-making framework (Section 1.3) that unrolls a single, static prediction into a sequence of partial predictions, without modifying existing models. This additional dimension in time allows us to learn a trade-off between cost and accuracy.

1.1 Dynamism in Natural Language Processing

Recent years have witnessed great success of statistical NLP in a wide range of areas, e.g., speech recognition, machine translation, and information extraction, mostly due to the availability of large speech and text data. At its core, a statistical system maps an input (e.g., a sentence, a document) to an output (e.g., a syntactic structure, a document category). In a typical NLP setting, this mapping is performed offline, meaning that the inputs are prepared in a batch and are

processed in complete units. However, real-time applications usually have stringent time-constraints and/or operate in an interactive environment, where such batch processing can be impractical.

Consider a chatbot that relies on core NLP techniques. To fully understand a talking human, many sub-tasks need to be solved. For example, speech recognition, parsing, sentiment analysis, sarcasm detection, and even facial expression and gesture classification if visual systems are enabled. Performing all of these tasks are expensive, especially when the human input becomes long. Fortunately, not all tasks are necessary at all times. For example, if the human asks "How is the weather today?", which is a common, unambiguous question, we do not need nuanced analysis like sarcasm detection. Knowing the right information to seek at the right time requires the bot to evaluate the current situation (e.g., a clear or vague message) and make decisions adaptively—what additional information would be most helpful for making a response *at the moment*.

In addition, when humans interact with a computer, their input is sequential—we speak/type word by word—and sometimes dynamic—we pause, interrupt and correct what we have said. Batch methods which wait for the entire input before processing can cause unnecessary delay. They are also less responsive when the input is changing (e.g., correction, topic diversion) because they passively wait for the human to hit "Enter". Therefore, we need dynamic methods that can process sequentially revealed data and maintain a (partial) output throughout the inputting period. In fact, commercial search engines such as Google search (Figure 1.2) have adopted the same dynamic paradigm. As soon as you start typing in the input box,

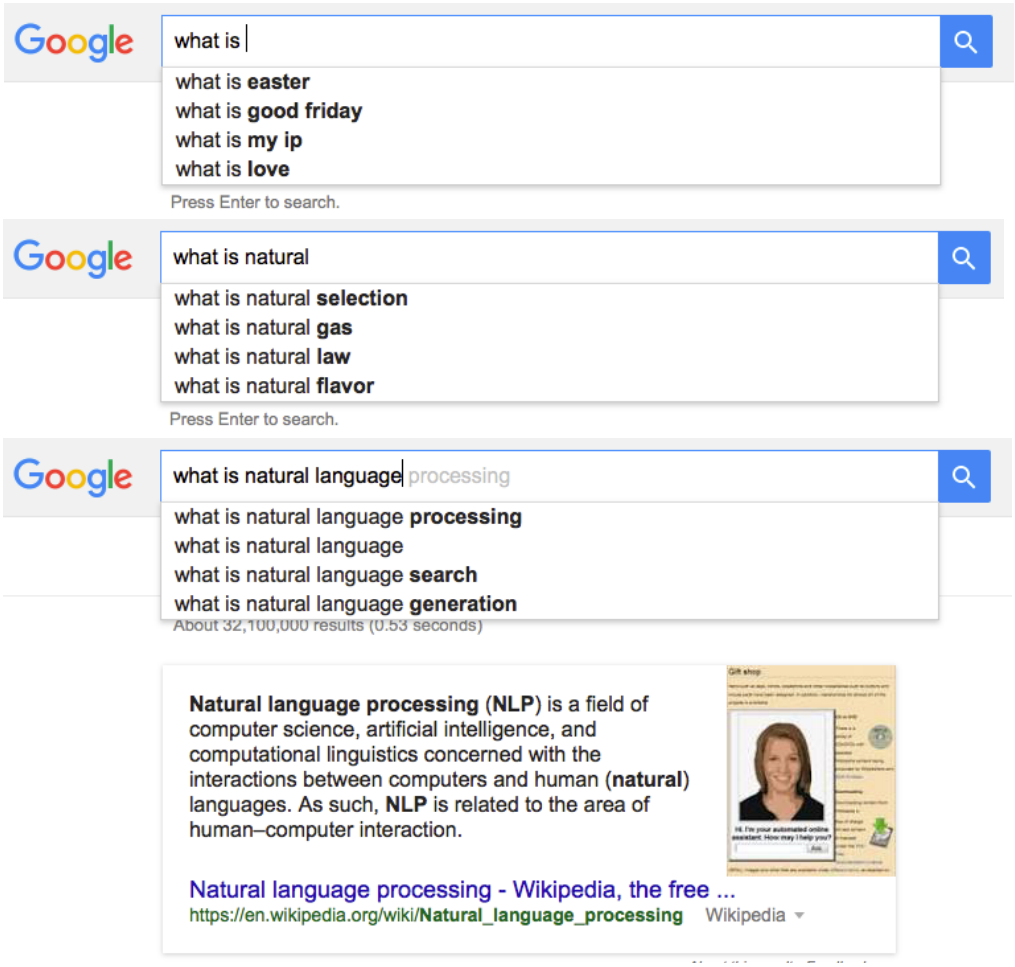


Figure 1.2: An example of dynamic information retrieval by Google.

it auto-completes the partial query and shows you the search result; this process is iterated whenever the input is updated.

Several lines (Dulac-Arnold, 2014; Benbouzid et al., 2012; Gao and Koller, 2011; Xu et al., 2013) of machine learning research have focused on tasks with dynamic input. Despite the advances on speed-accuracy trade-off in the machine learning community, there is much less work in dynamic, sequential settings in NLP. Such problems in NLP are more challenging because they are often structured and need complex decoding compared to multi-class classification addressed in related machine learning work. Given the burgeoning high-performance batch NLP tools (e.g., Moses, [1] Stanford Core NLP tools, [2] Illinois NLP tools, [3]), we are ready to bridge the gap by taking them to the dynamic setting.

1.2 Sequential Acquire vs. Sequential Reveal

Sequential problems roughly fall into two categories depending on how the input sequence is generated: (a) inputs come at different costs and are selected sequentially for cost-sensitive prediction; (b) inputs come in sequentially in a fixed order and early prediction is preferred despite insufficient information. A common challenge in both categories is the conflict between information *cost* and prediction *quality*. We characterize and give examples of the two types of problems below.

[1] http://www.statmt.org/moses/index.php?n=Main.HomePage
[2] http://nlp.stanford.edu/software/
[3] http://cogcomp.cs.illinois.edu/page/software/

1.2.1 Sequential Acquire

Prediction relies on information such as sensor measurements, lab reports, and retrieved knowledge. These pieces of information do not come cheap. In addition, because of varying levels of difficulty or ambiguity among instances, different instances of a problem often require a different amount of information to solve. To reduce cost and adapt to various instances, a system must dynamically acquire the information they need for decisions based on goals and current knowledge. Given a time or expense budget, the system must also balance a trade-off between the cost of acquiring more information (and reasoning about it) and the quality of the result.

In this setting, we have a set of available information to exploit. Our goal is to learn a decision-maker that decides which information to acquire at each step and when to stop (output the final prediction). Intuitively, recalling the chatbot example, we want to use cheap information for easy instances and additional expensive information only for hard instances; and we want to stop as soon as enough information has been obtained. One NLP problem with costly information is dependency parsing.

Dependency Parsing The goal of dependency parsing is to find the syntactic structure of a sentence in the form of a dependency tree. Figure 1.3 shows the parse tree for a sentence. Each arc of the tree represents a head-modifier relationship between two words of the sentence. For example, in Figure 1.3, "This" is a modifier of "time". Similar to other structured prediction methods, statistical dependency

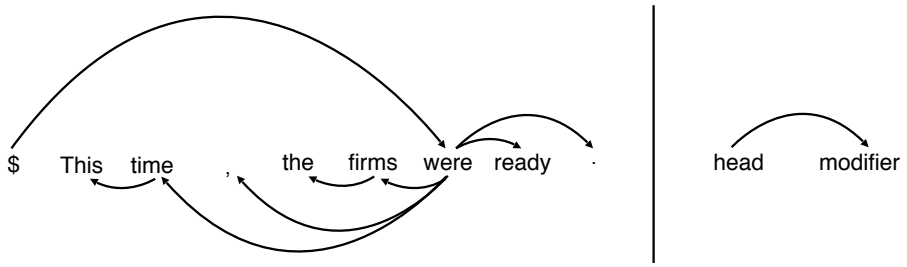


Figure 1.3: Dependency parse tree of an example sentence ($ represents the root).

parsing algorithms first compute a score for each potential parse tree and then output the highest-scoring tree. Often myriads of features are needed to evaluate the goodness of a potential parse tree. For example, given the sentence "I saw a bird", to decide whether "saw" is the head of "bird", we can include features such as their part-of-speech tags, word forms, morphology, distance, and various combinations of these basic components. Most computation is spent on getting feature values (e.g., running a part-of-speech tagger, running a morphology analyzer, and matching regular expressions) and hashing the feature values (often strings) to feature indices. However, some word pairs have an obvious head-modifier relationship, e.g., a determiner and a noun close to each other like "a" and "bird" in the example sentence. If we select features to compute only when needed at test time, we can resolve the easy cases without spending time on more complex features.

1.2.2 Sequential Reveal

In some applications, the input is revealed incrementally, and we do not have control over its arrival. Here the cost is time spent on waiting for more input, because an early prediction is often desired (this will become clear in the examples

below). Therefore, the model needs to decide when to output its prediction based on the partial input. It may seem that the sequential-revealing task is a simpler version of the above sequential-acquiring task, because we need not worry about what to acquire and only need to decide when to output. However, without the flexibility of selecting the piece of input most in need, we face more difficulty when predicting with scarce information: instead of selecting a few informative features, we may be given a few unhelpful ones. This results in a trade-off unbalanced towards accuracy. We discuss such challenges in simultaneous interpretation and quiz bowl below.

Simultaneous Interpretation In a standard machine translation setting, the system gets a complete sentence in the source language as input and produces its translation in the target language. However, when two people speak with each other, such batch translation would result in undesirable delay and hinder communication, because waiting for an utterance to finish may take a long time. A better way is to translate *while* the speaker is talking—the so-called simultaneous interpretation. It was invented during the Nuremberg Trials and has become a standard for international meetings since then. Given a stream of speech, a simultaneous machine translation system must decide when to start/resume translating and when to pause and wait for more input. The goal is to produce coherent translation while minimizing translation latency.

Since translation must be produced given a partial input sentence, one problem arises when translating between two languages with divergent word orders: The system is not able to produce a legitimate translation due to missing syntactic

German	Ich bin mit dem Zug nach Ulm gefahren
Gloss	I am with the train to Ulm traveled
English	I (.....waiting.....) traveled by train to Ulm

Figure 1.4: Large delay due to divergent word order in De-En translation.

constituents, and the translation has to be delayed until the missing constituents are available. One example is translating from a head-final language (SOV) to a head-initial (SVO) language, e.g., from German or Japanese to English. As shown in Figure 1.4, the verb "traveled" comes at the end of the source German sentence, but the target English sentence needs a verb immediately after the subject "I". In such cases, the translator has to wait for the necessary constituents which may appear at the very end. In Chapter 5, we discuss how to alleviate this problem by predicting the missing content and by reordering the target sentence.

Quiz Bowl Quiz bowl is a trivia game widely played in English-speaking countries between schools, with tournaments held most weekends. It is usually played between two teams. Similar to Jeopardy!, a moderator reads the questions to players, and they score points by buzzing in first (often before the question is finished) and answering the question correctly.[4] To test the depth of one's knowledge on a subject, a question usually starts with obscure information and reveals more and more obvious clues towards the end. One example question and its answer are shown in Figure 1.5. Therefore, players face a speed-accuracy trade-off: while buzzing later increases one's chance of answering correctly, it also increases the risk of losing the chance to answer to the opponent.

[4] A buzzer is used in quiz bowl to interrupt the question.

With Leo Szilard, he invented a doubly-eponymous refrigerator with no moving parts. He did not take interaction with neighbors into account when formulating his theory of heat capacity, so Debye adjusted the theory for low temperatures. His summation convention automatically sums repeated indices in tensor products. His name is attached to the A and B coefficients for spontaneous and stimulated emission, the subject of one of his multiple groundbreaking 1905 papers. He further developed the model of statistics sent to him by Bose to describe particles with integer spin. For 10 points, who is this German physicist best known for formulating the special and general theories of relativity? *Answer: Albert Einstein*

Figure 1.5: A quiz bowl question.

A quiz bowl bot should consist of a question-answering model that predicts the answer given incremental text input, and a decision-making model that decides when to buzz. Normally, we would train the decision-making model to buzz when the question-answering model is most confident about the correct answer. However, due to uncontrollable input order we have a similar challenge here: how to improve performance when the question-answering model becomes confident only towards the end of a question, in which cases a naive model is forced to answer very late. One distinct characteristic of this problem is that decisions are made in a multiagent environment: the opponents are also actively making decisions to compete with us. In Chapter 6 we discuss adaptive strategies to exploit the opponent's behavior.

1.3 A Dynamic Solver

We have seen a recurring pattern in sequential problems with dynamic input from examples in the previous section: a sequence of decisions leading to a sequence of predictions that incrementally builds up the final output. Toward this end, the framework proposed in this thesis aims to address the following questions: (a) how

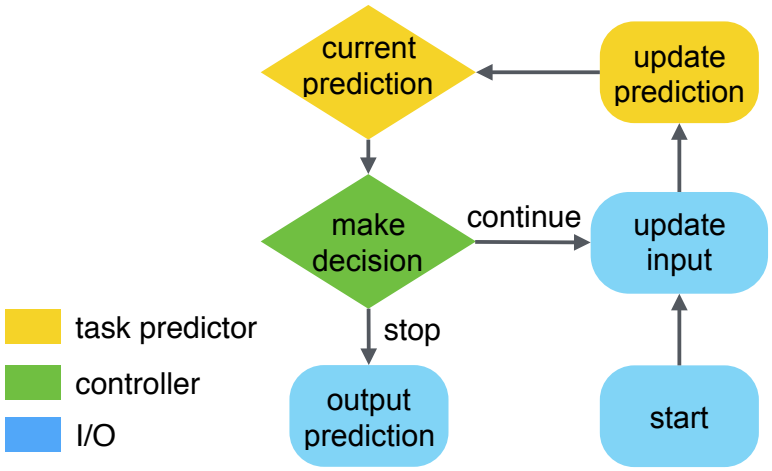


Figure 1.6: Interaction between the task predictor and the controller.

to make the intermediate and final predictions; (b) what decisions to make and how they affect the following input and prediction; (c) how to learn decision-making in an interactive environment. This section provides high-level answers to these questions and gives and overview of our dynamic solver.

Building a dynamic solver requires three components, each responding to one question raised above:

- a *task predictor* that outputs predictions given (partial) inputs;
- a *search space* that defines how the problem is solved in a sequential manner;
- a *controller* that reacts to any change of the input and defines a path in the search space leading to a solution.

For all tasks addressed in this thesis, there exists a solver for the static version of the problem, and it is used as an off-the-shelf task predictor. The task predictor does not necessarily deal with the fact that the input may be partial; it is used only as a black box. The complication due to partial inputs is instead handled by the search

space and the controller. The main advantage of separating this complication from the task predictor is better compatibility of the framework and more freedom in designing the search space. Our framework works with any task predictor without being constrained by its design, since the interaction happens only at the I/O level.

We learn the solver by actively interacting with the environment (input/output). Figure 1.6 shows the paradigm of algorithms proposed in this thesis. The input is updated iteratively. After each update, the task predictor makes a new intermediate prediction. The controller then makes a decision about whether to terminate the process; if not, it decides how to update the input based on past intermediate predictions. During this decision-making process, both the task predictor and the controller make predictions which are dependent on each other. It is useful to differentiate between the two types of predictions: We refer to the output of the task predictor as a (intermediate) *prediction*, and output of the controller as a *decision* (e.g., continue or stop).

We assume that the task predictor is given or pre-trained. There is no constraint on the task predictor, thus its choice is problem-dependent. For example, a convolutional neural network can be used for object recognition where predictions are object classes; a maximum entropy classifier can be used for text classification where predictions are text labels such as topics or sentiment. The only requirement is that the task predictor must be able to work with partial inputs. Handling incomplete data is an area with extensive literature, e.g., feature imputation. We do not address this (orthogonal) problem in particular. Instead, the task predictor considers all inputs complete and operates in the same way as it does in standard

settings.

The main tool we use to learn the controller is reduction-based imitation learning, which we describe in Chapter 2. The high-level idea is to have an expert demonstrate the desired decisions to make in various situations and learn to mimic the expert. The advantage of this approach is that given supervision from the expert, we can reduce it to a supervised learning problem and exploit recent advances in this area. The disadvantage, of course, is that it requires an expert throughout the training process. In our setting, the desired behavior is better, faster prediction. Fortunately, in many NLP applications such an expert is easy to obtain as we will explain in later chapters. In addition, recent imitation learning algorithms are able to work with sub-optimal experts by better exploration (Chang et al., 2015).

1.4 Contribution

The primary contribution of this thesis is a meta-algorithm for converting an existent batch model to a dynamic model that solves problems with sequential input. Unlike prior approaches that rely on a particular prediction model or are crafted for specific problems, we have a decision model dedicated to handling complication resulted from incomplete inputs, and we keep the batch predictor untouched. This separation enables better generalization and larger flexibility: there is much less constraint when designing the search space (i.e., the sequential solution). Otherwise, we usually need to modify the task predictor (e.g., a decoder) that is specifically designed for a problem. In addition, we can quickly swap in a more advanced task

predictor in future without changing the whole system.

We demonstrate the effectiveness of our framework on a variety of tasks, including standard multiclass classification, structured prediction (dependency parsing, simultaneous machine translation), and games (quiz bowl). We show that our framework is flexible enough to adapt to the needs of different problems. New functionality can be incorporated as an action to form a more complex search space. New objectives can be encoded as the reward function, where users can specify a desired trade-off between cost and accuracy. Performance can be further improved by using batch models that are trained or fine-tuned to handle partial inputs.

In addition, we have investigated several previously under-explored areas. For example, most work on dependency parsing (or more generally, on structured prediction) focuses on better and faster decoding methods, but few studied the cost of feature computation. Similarly, machine translation has largely focused on batch translation at the sentence level. Although there exists work on translation at the sub-sentence level based on speech segmentation, little work exploits linguistic knowledge and strategies of human interpreters. Finally, we first explored human-computer question-answering in a competitive setting—one step towards building strategic dialog agent (e.g., for negotiation).

1.5 Beyond Natural Language Processing

Our work has applications in areas other than NLP as well. We briefly describe some directions below.

Generally speaking, sequential information selection is relevant to any task where the input is costly. In cloud computing, we may need to collect logs from each host to inspect operation of the distributed system (e.g., for debugging). These logs are often huge structured files that are hard to analyze. Instead of tediously examining everything in each log, a dynamic model can sequentially process from coarse information to finer details, thus efficiently identifies dubious parts where the problem may reside. Similarly, in the Internet of Things, the central system needs to frequently collect sensor data from its nodes to examine the current status and send commands. Instead of periodically collect data from all nodes, an energy-efficient system would automatically decide when to pull the information in need and which node to pull from given past information.

Another potential application is online education. With the success of massive open online courses, there is an increasing need to develop personalized courses for individual students. Given a large amount of data from user record, it is possible to adapt the teaching plan to each student based on their recent feedback. For example, we can learn a dynamic planner to decide which lecture or how much of the lecture to show to a student given one's current progress in the course. Besides learning a good planner, another challenge in this problem is human-computer interaction: how can we design an effective interface to communicate with students and collect their feedback for training the planner.

A related sequential search problem in the field of programming languages is program synthesis. We take a user specification as input and aim to transform that into executable code. This is usually formulated as a constraint satisfaction problem:

A candidate program is proposed and then tested by a verifier; given feedback from the verifier, a new candidate program is generated and the process repeats until a program passes the verifier. In our framework, this can be naturally framed as a sequential decision-making problem. Instead of proposing the next program based on search heuristics, we can learn to search in the program space more efficiently, for example, it is possible to identify which part of the program to modify based on the feedback, or to quickly prune unpromising search areas.

1.6 Roadmap

We begin by introducing machine learning background needed to understand the remainder of this thesis (Chapter 2), followed by applications with sequentially acquired and sequentially revealed inputs (Chapter 3 to 6). To coherently present the thesis, we think it is helpful to discuss prior work related to an application in its own chapter, instead of putting them all in a single chapter. Therefore, we include a section of related work at the end of each application chapter. Chapter 7 concludes with a summary and future work. The thesis proceeds as follows.

Chapter 2 introduces relevant background from machine learning. We start with the MDP formulation of sequential decision-making problems and introduce two main approaches for solving MDPs: reinforcement learning and imitation learning. The focus of this chapter is a family of interactive imitation learning algorithm based on learning reduction—DAgger (Dataset Aggregation) and its variant AggreVaTe (Aggregate Values to Imitate), which forms the core

learning algorithms for our model. We also describe the theoretical guarantee of these algorithms.

Chapter 3 describes our sequential prediction framework in the simplest case, where the model selects pieces of information sequentially to complete a supervised classification task. We present results of dynamic feature selection on common multiclass classification datasets. We also propose a variant of DAgger that tackles the learning problem when the expert is too good to mimic. This chapter forms the basis of our method, which is adapted to more complex problems in later chapters.

Chapter 4 extends the dynamic feature selection method introduced in the previous Chapter to structured prediction. As mentioned in Section 1.2.1, feature computation is expensive in dependency parsing. We present how dynamic feature selection can be done jointly with inference in minimum-spanning tree dependency parsing. We show a speedup of parsing time across seven languages. This work is among the first to apply feature selection to structured problems for test-time efficiency.

Chapter 5 begins applications where the input is sequentially revealed. This chapter focuses on simultaneous machine translation. We first describe challenges specific to this domain due to divergent word order (e.g., see Section 1.2.2). We then present two methods to alleviate the problem, including prediction of future content and target-side word reordering. This work is the first to combine machine learning innovation with linguistic knowledge and human strategies for machine interpretation.

Chapter 6 presents another application with sequentially revealed input—quiz bowl. It is both an incremental text classification task and a two-player zero-sum game which involves interaction with another human opponent. Given the two perspectives, we experiment with two approaches: imitation learning and reinforcement learning with opponent modeling. This is the first work that directly learns to compete with humans in a text-based game.

Chapter 7 summarizes the results presented in this thesis and proposes future directions for these methods.

Chapter 2: Machine Learning Foundations for Sequential Decision-Making

Sequential prediction is challenging because the training data are dynamic. The current decision results from previous decisions and will affect future inputs. Since data are generated online, one big challenge in learning in a dynamic, sequential process is to make sure that distributions of the training data and the test data stay as close as possible. This chapter covers machine learning background necessary for understanding the problem formulation and learning algorithms in this thesis.

We start by introducing the basics of Markov decision process (MDP), a typical model for sequential decision-making. We use it to formulate the sequential process in many of our algorithms. Next, we describe common algorithms for solving MDPs with a focus on imitation learning (or learning from demonstrations). Finally, we present the main learning method we use: a family of imitation learning algorithms based on interaction with the environment and a teacher.

2.1 Markov Decision Process

The Markov decision process (MDP) is a concise mathematical model describing an environment that responds stochastically to actions conducted by a decision-

maker. MDPs were first studied at least as early as the 1950s (Bellman, 1957) and have been widely used in robotic systems.

An MDP is defined by a 5-tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$. $\mathcal{S}$ is the set of all possible *states* ($s \in \mathcal{S}$) of the environment, and $\mathcal{A}$ is the set of *actions* ($a \in \mathcal{A}$) that an agent may take to interact with the environment. The transition function T defines the dynamics of the environment: $T\colon \mathcal{S} \times \mathcal{A} \to Pr(\mathcal{S})$. We use $T(s, a, s')$ to denote the probability of transitioning to state s' after executing action a in state s. The transition function assumes the Markov property: the conditional probability distribution of future states depends only on the current state. The reward function R quantifies the goodness of an action in a certain state. $R(s, a, s')$ denotes the immediate reward of taking a in s and moving into s'; we use r_t as a shorthand for $R(s_t, a_t, s_{t+1})$.

For example, our problem of sequential information acquisition can be formulated as an MDP. Suppose we want to select features adaptively at test time to reduce computation cost. The state contains selected features and past predictions, and the action space is the set of all unused features. Once a new feature is selected, we transit to the next state deterministically: the new feature is added to the state and the intermediate prediction is updated.

An agent in an MDP observes the state and interacts with the environment by taking actions and receiving rewards until a terminal state is reached. A *policy* π defines how an agent acts in an MDP, mapping a state to an action: $\pi\colon \mathcal{S} \to \mathcal{A}$. The policy can be either a deterministic function of the state, where $a = \pi(s)$, or a stochastic distribution over actions, where $a \sim \pi(\cdot|s)$. We consider only determinis-

tic policies in this thesis.

Given an MDP, our goal is to find a policy that maximizes the agent's expected discounted cumulative reward over a potentially infinite task horizon. The expected reward is defined by $\mathbb{E}_{a_t \sim \pi, s_{t+1} \sim T}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})\right]$, where $\gamma \in [0, 1)$ is the discount factor so that future rewards are weighted down accordingly. In this thesis, all of our tasks have a finite horizon. In addition, to simplify terminology, we refer to the expected discounted cumulative reward as future reward thereafter.

Most algorithms for solving MDPs are based on estimating *value functions*, which evaluate how desirable it is for an agent to be in a given state. Here "desirable" is defined in terms of future reward. Formally, the value of state s under policy π is

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right]. \tag{2.1}$$

We can further define the value of taking action a in state s following policy π:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right]. \tag{2.2}$$

We call Q^{π} the *action-value function* or *Q-function*. One important property of these value functions is that they can be defined recursively, which is given by the Bellman Equation (Bellman, 1957):

$$Q^{\pi}(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi}(s')\right] \tag{2.3}$$

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi}\left[Q^{\pi}(s, a)\right] \tag{2.4}$$

In addition, the optimal value functions satisfy:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a') \right] \tag{2.5}$$

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a) \tag{2.6}$$

and the optimal policy is:

$$\pi^*(s) = \arg\min_{a \in \mathcal{A}} Q^*(s,a) \tag{2.7}$$

Suppose we know the transition function T and the reward function R, and there are finitely many states and actions, we can solve the MDP via linear programming or dynamic programming based on the above recursive relationships, e.g., value iteration (Bellman, 1957), policy iteration (Howard, 1960) and so on. However, in real-world applications, the model dynamics are often unknown and we often face large state and action spaces. In these cases, the exact methods are no longer feasible. Instead, we turn to approximate methods based on *experience* (sampled trajectories) rather than an MDP model. These methods (e.g., Q-learning (Watkins and Dayan, 1992; Sutton and Barto, 1998), policy gradient (Sutton et al., 2000)) use techniques of Monte Carlo estimation (sampling), bootstrapping, and/or function approximation. We have introduced basic ideas behind reinforcement learning (RL) algorithms; in the next section, we describe imitation learning, a more efficient approach that takes advantage of expert demonstration.

2.2 Imitation Learning

RL algorithms aim to learn a policy that maximizes the future reward; learning is often done by trial-and-error style interaction with the environment. On the other hand, imitation learning learns a policy that mimics an expert's behavior, which is also called apprenticeship learning or learning from demonstrations. Imitation learning assumes access to a human expert or a reference policy that demonstrates the desired behavior during training. Instead of directly optimizing the reward, the agent aims to imitate the expert who executes an optimal or near-optimal policy implicitly. Therefore, imitation learning is suitable for problems where a reward function is not obvious or exploration is expensive, while expert demonstration is easy to obtain. One such example is robot control, such as navigation, manipulation, and locomotion, where imitation learning has gained success (Coates et al., 2008; Pieter Abbeel and Ng, 2008; Ratliff et al., 2006).

Imitation learning is attractive to us for two reasons. First, the NLP problems we are interested in are all supervised; therefore, we can often easily compute the optimal action sequence from the ground truth, or at least derive some form of weak supervision from it. For example, in quiz bowl, the expert should buzz if the current answer prediction is correct and wait otherwise. The expert supervision greatly reduces our search space, because the expert guidance constrains the search area to be close the expert's path. Second, imitation learning is closely related to well-studied supervised learning problems such as multiclass classification. Therefore, we can use powerful existing learning tools in areas of, for example, online learning

and deep learning. Below we introduce basic notations used in imitation learning.

Notation We consider a sequential task with a T-step time horizon.[1] The state distribution under policy π at time t is d_π^t. The average state distribution over T steps is $d_\pi = \frac{1}{T}\sum_{t=1}^{T} d_\pi^t$. To be consistent with terminology from supervised learning, we use the notion of loss rather than reward. Formally, the immediate loss of policy π is $L(s, \pi(s))$, and we use $L(s, \pi)$ as a shorthand. We define the *task loss* as the T-step expected loss of π: $J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{s\sim d_\pi^t}[L(s, \pi(s)] = T\mathbb{E}_{s\sim d_\pi}[L(s, \pi(s))]$. This is the loss that we are interested in minimizing.

For some tasks, however, we may not know the loss function L that accurately penalizes bad behavior, or we can assign a loss only in the terminal state, in which case the lack of immediate loss creates difficulty for RL. For example, when teaching a robot to move a cup from one table to another, we do not necessarily know how to quantify the goodness of a move except when the cup is dropped or placed as expected. The delayed reward introduces large variance in future reward estimation and makes learning harder. On the other hand, in imitation learning we observe demonstrations from an expert who is assumed to minimize $J(\pi)$, and we aim to mimic the expert's behavior, e.g., cloning a human's hand trajectory of moving a cup. Since we have supervision (the expert action) in each state along the trajectory, the problem becomes easier and can be reduced to classification, as we explain in the next section.

[1] We overload the symbol T here to refer to the sequence length instead of the transition function introduced in the previous section.

2.2.1 Reduction to Classification

In imitation learning, examples of state-action pairs generated by the expert are used as supervision. Instead of minimizing the task loss, we minimize a surrogate loss $\ell(s, \pi, \pi^*)$ that measures the difference between the learned policy and the expert policy. This reduces imitation learning to a classification problem because we essentially learn to recreate the expert's action in each state. For example, if we have a continuous action space, we may use squared loss, where $\ell(s, \pi, \pi^*) = (\pi(s) - \pi^*(s))^2$ and π is a regressor; if we have a discrete action space, we may use negative log likelihood, where $\ell(s, \pi, \pi^*) = -\log \frac{\exp^{\pi(s,a^*)}}{\sum_{a \in \mathcal{A}} \exp^{\pi(s,a)}}$ ($a^* = \pi^*(s)$ and $\pi(s, a)$ is the probability of choosing action a given by π), and π is a maximum entropy (MaxEnt) classifier.

Therefore, a straightforward approach is to use the expert's trajectories as supervised data and learn a multiclass classifier that predicts the expert's action. At each time step t, we collect a training example $(\phi(s_t), \pi^*(s_t))$, where ϕ maps a state to a feature vector, and $\pi^*(s_t)$ is the class label (expert's action) of this example. Then using any standard supervised learning algorithm, we can learn a policy

$$\hat{\pi} = \arg\min_{\pi \in \Pi} \mathbb{E}_{s \sim d_{\pi^*}}[\ell(s, \pi, \pi^*)], \tag{2.8}$$

where Π is the policy space and d_{π^*} is the distribution of states generated by executing the oracle policy.

Imitation learning algorithms are often analyzed in terms of regret to the

expert performance. To prove that mimicking the expert indeed leads us to the true goal, we need to bound the difference between $J(\pi)$ and $J^*(\pi)$ based on ℓ. Ross and Bagnell (2010) show:

Theorem 1. *Assume ℓ upper bounds the 0-1 loss, and L is bounded in $[0, L_{\max}]$. Let $\mathbb{E}_{s\sim d_{\pi^*}}[\ell(s, \pi, \pi^*)] = \epsilon$, then:*

$$J(\pi) \leq J(\pi^*) + L_{\max} T^2 \epsilon.$$

This bound shows that performance of the learned policy is upper bounded by the expert's, and in the worst case the difference increases quadratically with time. For detailed proof, see Theorem 3.3.2 of (Ross, 2013). Similar bounds on the supervised learning approach can also be found in Kääriäinen (2006) and Syed and Schapire (2011).

The main reason for the quadratically increasing loss is the mismatch between training and test state distribution. During training, the learner observes state-action pairs generated by the expert only. However, at test time, the learner may go to a bad state that the expert never visits due to prediction error. As a result, it can be trapped in the bad state forever because the expert has not demonstrated what to do in such a state. Taking quiz bowl as an example, the expert may always answer within the first sentence; at test time, if the agent goes beyond the first sentence, it may never answer. Therefore, it is possible for the learner to achieve a small surrogate loss during training *on the expert's state distribution*, but still does

poorly at test time. To alleviate this problem, we need to explore states that may be encountered at test time. In the next section, we describe an iterative learning method that tackles the data mismatch problem.

2.3 Iterative Imitation Learning

The key idea of the iterative approach to imitation learning is to train the policy under states visited by both the expert and the learner. To explore states induced by the learner, we learn policies iteratively and generate states using previous learned policies. Many algorithms have been proposed along this line, including SEARN (Daumé III et al., 2009), DAGGER (Ross et al., 2011), AGGRAVATE (Ross and Bagnell, 2014) and LOLS (Chang et al., 2015). We mainly use DAGGER in this thesis, which is perhaps the most efficient one among the others. We also present its variant AGGRAVATE in this section.

In its simplest form, the Dataset Aggregation (DAGGER) algorithm (Ross et al., 2011) works as follows. In the first iteration, we collect a training set $\mathcal{D}_1 = \{(s, \pi^*(s))\}$ where s is induced by the expert policy ($\hat{\pi}_1 = \pi^*$); then learn a policy $\hat{\pi}_2$ using $\mathcal{D}_1$, same as the supervised learning approach. In iteration i, we collect trajectories by executing the previous policy $\hat{\pi}_i$, and form the training set $\mathcal{D}_i$ by labeling $s \sim d_{\pi_i}$ with expert actions $\pi^*(s)$. At the end of iteration i we learn a new policy $\hat{\pi}_{i+1}$ on all examples collected so far, $\mathcal{D}_0 \bigcup \ldots \mathcal{D}_i$, hence the name "Dataset Aggregation". Intuitively, this enables the learner to learn how to recover from its mistakes when it is off the expert's trajectory, such that the policy performs well

Algorithm 1 DAGGER

1: Initialize $\mathcal{D} \leftarrow \emptyset$, $\pi_1 \leftarrow$ any policy in Π.
2: **for** $i = 1$ to N **do**
3: Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_{i-1}$. ▶ Using stochastic policy
4: Sample T-step trajectories using π_i where $s \sim \pi_i$.
5: Collect dataset $\mathcal{D}_i = \{(s, \pi^*(s)\}$ from the trajectories.
6: Aggregate datasets $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
7: Train policy $\hat{\pi}_{i+1}$ on $\mathcal{D}$. ▶ For example, a linear SVM
8: **end for**
9: **Return** best $\hat{\pi}_i$ evaluated on the validation set.

under states induced by itself.

We show the complete DAGGER algorithm in Algorithm 1. To learn from states more likely to be induced by the learned policy at test time, we explore by a mixture of π^* and the last learned policy $\hat{\pi}_{i-1}$, as shown in Line 3 of Algorithm 1. In practice, we often set $\beta_1 = 1$ so that no initial policy needs to be specified before learning starts. In later iterations, we schedule β to decrease over time such that the state distribution becomes closer to one induced by a learned policy. For example, we can choose $\beta_i = (1 - \alpha)^{i-1}$ where α is a small constant, such that the probability of taking an expert's action decays exponentially in the number of iterations.[2] Such stochastic mixing policies are first proposed in SEARN to gradually move away from the expert's trajectory. In this thesis, we use the hyperparameter-free version where $\beta_1 = 1$ and $\beta_i = 0$ for $i > 1$ unless stated otherwise.

At the end of each iteration, a new policy $\hat{\pi}_i$ is learned on the aggregated dataset. The procedure of training the policy is the same as training a multiclass classifier or a regressor (in which case the action space is continuous). For example, we can use standard machine learning packages such as LIBSVM (Chang and Lin,

[2] In theory, DAGGER requires $\frac{1}{N}\sum_{i=1}^{N} \beta_i \to 0$ as $N \to \infty$.

2011), LIBLINEAR (Fan et al., 2008) and Scikit-learn (Pedregosa et al., 2011). Furthermore, if there is a large amount of data, we can use an online learner and update after each new example is collected, e.g. using Vowpal Wabbit (Langford et al., 2007). At the end, we return the policy that performs best on the validation set; in practice, one can also return the last policy.

Analysis The theory of this family of iterative imitation learning algorithms largely relies on no-regret online learning. The key is to view each iteration as one step in the online learning setting: The adversary picks a loss function in each iteration depending on the state distribution, and the learner chooses the best policy in hindsight, much as in the Follow-the-Leader algorithm. Here we only present related theoretical results, as the proof is not relevant to understanding the rest of the thesis. Interested readers are referred to Chapter 3 of (Ross, 2013).

We define several key variables used in the theoretical guarantee. Let $Q^*_t(s, \pi)$ be the t-step cost in L of executing π in in state s and following π^* thereafter for t steps. If the expert policy satisfies $Q^*_{T-t+1}(s, \pi) - Q^*_{T-t+1}(s, \pi^*) \leq u$ for any t, s and π, where u is a constant, then we call it a u-robust expert policy. This condition requires that the expert is capable of recovering from a mistake at some step without incurring additional loss larger than u. In many situations, it is reasonable to assume u is small. For example, when learning to drive a car, if the learner steers the car to a wrong direction, a human expert can often steer it back with a little detour. In our sequential information acquisition case, if a non-informative or noisy feature is selected by mistake, we can add a discriminate feature in the next step to correct

the prediction. Further, we use ϵ_{class} to denote the minimum loss we can achieve in the policy space Π such that $\epsilon_{class} = \min_{\pi \in \Pi} \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{s \sim d_{\pi_i}}[\ell(s, \pi, \pi^*)]$; and we denote the sequence of learned policies $\pi_1, \pi_2, \ldots, \pi_N$ by $\pi_{1:N}$.

Given the above definitions, we can show that DAGGER results in a relative loss linear in T compared to the supervised learning approach:

Theorem 2. *(Ross et al., 2011) Assume ℓ upper bounds L, π^* is u-robust, and N is $O(uT \log T)$, then there exists a policy $\pi \in \pi_{1:N}$ such that:*

$$J(\pi) \leq J(\pi^*) + uT\epsilon_{class} + O(1). \tag{2.9}$$

The main reason of DAGGER's success is that it explores states likely to be encountered at test time, and learns to recover from mistakes. However, it has a couple of limitations: (a) it requires an expert who knows what to do in *any* state and is available throughout training, which may be very expensive if the expert is a human; (b) it treats all mistakes equally, whereas some mistakes can be more costly than others, e.g. hitting the car in front vs. slow acceleration. The first limitation is not a problem in our case since our expert actions are easy to obtain, which will become clear in later chapters. The second limitation can pose a problem to some applications, and we present learning algorithms that consider action costs next.

2.4 Cost-sensitive Imitation Learning

In the previous section, we have assumed that we do not know the actual loss function L and minimize the surrogate loss ℓ instead. However, sometimes we do have additional information on L which should be considered during training. For example, we can often assign a loss in the terminal state based on the result of a task even though we may not know the cost of intermediate actions. Knowledge on L helps us to identify which imitation errors are tolerable and which are serious. In this section, we present methods that consider the cost of each action during policy learning.

We define the cost of an action as its expected future loss (cost-to-go) following the expert policy after the current state. Formally, if the learner takes action a in state s at time t, the expert cost-to-go of a is $Q^*_{T-t+1}(s, a)$. It is straightforward to extend DAGGER with the cost information. In Algorithm 1, when sampling trajectories using π_i, we roll out random actions to get their cost-to-go under the expert policy. Then we collect examples augmented with cost: (s, a, Q^*) (we use Q^* as a short hand for $Q^*_{T-t+1}(s, a)$). The imitation problem is now reduced to a *cost-sensitive* classification problem, which again can be solved efficiently using standard algorithms (Langford and Beygelzimer, 2005; Beygelzimer et al., 2009).

In Algorithm 2, we show training procedure of the extension of DAGGER, AGGRAVATE (Aggregate Values to Imitate). Comparing to DAGGER (Algorithm 1), the main difference is that instead of using the expert action $\pi^*(s)$ as the supervision signal, we use the expert cost-to-go $Q^*(s, a)$. This allows us to measure the good-

Algorithm 2 AGGRAVATE Algorithm (DAGGER with cost)

1: Initialize $\mathcal{D} \leftarrow \emptyset$, $\pi_1 \leftarrow$ any policy in Π.
2: **for** $i = 1$ to N **do**
3: Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_{i-1}$. ▶ Using stochastic policy
4: **for** $j = 1$ to M **do**
5: Sample t uniformly from $\{1, \ldots, T\}$.
6: Roll in with π_i until t.
7: Execute exploration action $a \in \mathcal{A}$ in current state s_t.
8: Roll out with π^* until terminal state and record the cost-to-go Q^*.
9: **end for**
10: Collect dataset $\mathcal{D}_i = \{(s, a, Q^*\}$ from the trajectories.
11: Aggregate datasets $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
12: Train cost-sensitive classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
13: **end for**

ness of each action in a given state and apply cost-sensitive classification. Similar rollout-based methods include SEARN and LOLS.

In standard cost-sensitive classification settings, a cost is assigned to each label. In AGGRAVATE, however, we only know the cost of sampled actions (Line 7–8). Given only partial cost information, we can learn a regressor to predict the cost-to-go, or treat the unobserved action cost as zero. In addition, instead of exploring actions randomly, we may want to use an exploration strategy to select actions that are most helpful to learning. The action selection problem can be formulated as a contextual bandit problem (Auer et al., 2002; McMahan and Streeter, 2009; Beygelzimer et al., 2011). In this thesis, we take a simple approach of rolling out all available actions, same as SEARN's exploration. The cost of each action is then normalized by subtracting the minimum cost-to-go from the state: $Q^*(s, a) \leftarrow Q^*(s, a) - \min_{a' \in \mathcal{A}} Q^*(s, a')$. Computing the full cost information is not as expensive as one would think if proper memoization is applied (Chang et al., 2014).

We can obtain a similar theoretical guarantee for AGGRAVATE. Let $\epsilon_{class} = \min_{\pi \in \Pi} \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{t \sim U(1,T), s \sim d^t_{\pi_i}} \left[Q^*_{T-t+1}(s, \pi_i) - \min_a Q^*_{T-t+1}(s, a) \right]$, which is the minimum expected relative cost achieved by policies in Π over N iterations. We have

Theorem 3. *(Ross and Bagnell, 2014) Assume Q^* is non-negative and is bounded by $Q^*_{\max}$, and $\beta_i \leq (1-\alpha)^{i-1}$ for some constant α. If N is $O(\frac{Q^*_{\max}}{\alpha} T \log T)$, then*

$$J(\pi) \leq J(\pi^*) + T\epsilon_{class} + O(1). \tag{2.10}$$

In practice, rollouts can be expensive to execute, especially if T is large. To reduce computational cost, one can also directly assign a heuristic cost to each action based on domain knowledge of the task. Generally speaking, we have the following choices for rollouts:

- **No rollout** (DAGGER or using heuristic costs). This is the most efficient choice and often provides decent performance in practice.
- **Expert rollout** (SEARN, AGGRAVATE). It provides additional information at the cost of more computation. However, there are various ways to speedup the rollout process, many of which has been implemented in the Vowpal Wabbit L2S package, for example, memoization and early termination. Additionally, in structured prediction expert cost-to-go is often easy to compute/simulate without actually rolling out a policy (Daumé III et al., 2009).
- **Learner rollout** (RL). This resembles Monte Carlo methods in RL, or more

specifically, classification-based RL (Lagoudakis and Parr, 2003; Lazaric et al., 2010). It is perhaps the most inefficient option but can be helpful when the expert is sub-optimal (Chang et al., 2015).

2.5 Summary

We have introduced the basics of MDP, which is a typical mathematical formulation for sequential decision-making problems. We then described two approaches for solving MDPs, reinforcement learning, and imitation learning. These are the main machine learning tools we will apply in the following chapters.

Imitation learning is usually more efficient when the expert policy is easy to construct, which is often true in our problems. On the other hand, reinforcement learning is needed when no good expert policy is available or is cheap to query. Specifically, we design greedy selection experts for sequential acquiring problems and use DAGGER for the first two applications: dynamic feature selection for multiclass classification (Chapter 3) and dependency parsing (Chapter 4). For simultaneous interpretation (Chapter 5), we propose a novel reward function that considers both translation quality and speed, and we use imitation learning with cost (negative reward) to solve it. Finally, we use reinforcement learning (Q-learning) for quiz bowl (Chapter 6) so that we can better model interaction with multiple agents.

Chapter 3: Dynamic Feature Selection

This chapter describes joint work with Hal Daumé III and Jason Eisner (He et al., 2012) published in NIPS 2012.

Before going into sequential problems in NLP, we first focus on a sequential acquisition problem in the setting of cost-sensitive classification. In a practical machine learning task, features are usually acquired at a cost with unknown discriminative powers. In many cases, expensive features often imply better performance. For example, in medical diagnosis, some tests can be very informative (e.g., X-ray, electrocardiogram) but are expensive to run or have side-effects on the human body. While at training time we can devote large amounts of time and resources to collecting data and building models, at test time we often cannot afford to obtain a complete set of features for all instances. This leaves us a cost-accuracy trade-off problem. We propose to reduce feature cost by selecting features dynamically on an instance basis at test time.

This chapter introduces a sequential decision-making framework for dynamic feature selection. In the next chapter, we show how this approach can be adapted to dependency parsing, a structured prediction problem in NLP. The central learning method we use is the DAgger algorithm introduced in the previous chapter.

Besides showing its effectiveness when applying to non-typical imitation scenarios, we discuss a practical issue often arising in our setting: the expert behavior is too good to learn for the learner. We further propose an adaptation of DAgger which alleviates this problem.

3.1 Sequential Acquisition of Features with Cost

We now define the problem of dynamic feature selection. We consider the classification setting where each feature is obtained at a known cost. The precise definition of cost is problem-dependent, for instance, the computation time or the expense of running an experiment. We assume that we are provided with a classifier that has been trained to work well on instances for which all feature values are known, hence we are only concerned with the *test time* classification performance in terms of both accuracy and cost. We refer to the pre-trained classifier as the *task predictor* below.

Unlike typical feature selection methods that use a fixed set of pre-selected features for all instances, we select a (different) set of features for *each instance* adaptively, such that hard instances get more budget and easy instances get less budget. This can be naturally framed as a sequential decision-making problem. At each step, given selected features and current prediction/belief of the instance, we decide whether to stop acquiring more features and output a prediction, and if not, which feature(s) to purchase next. We allow the agent to select more than one feature at a time. A selectable bundle of one or more features is called a *factor*; such

a bundle might be defined by a feature template, for example, or by a procedure that acquires several features at once.

3.2 An Imitation Learning Approach

We represent the feature selection process as an MDP. The state includes the set of factors selected so far and intermediate predictions based on these factors. Thus given D factors, we have a state space of size 2^D. The action set includes all factors that have not been selected yet, as well as the termination action STOP (stop adding more features and output a prediction). An agent follows an acquisition policy π that determines which action to take in a state. After a new factor is added, we run the task predictor to update the current prediction with the new features, thus transit to the next state. When classifying with an incomplete feature set, we set values of non-selected features to be zero.[1] Next, we define the loss function and the expert used in imitation learning.

Loss Function The loss L should consider both the prediction quality and the feature cost. We use *classification margin* as a measure of prediction quality. Assuming that the task predictor h computes a score for each class (which is true for most classification algorithms) given any set of features, the classification margin is defined as the difference between the score of the true label and the highest score among those of other labels. Higher margin indicates a more confident prediction of the correct class. In general, we can use other measures depending on the specific

[1] Classification with missing features is an extensively researched area, e.g. feature imputation. Here we use this simple method which is shown to be effective in our case.

application as well, e.g. log likelihood of the true label. We can compute the loss of a state s as a simple linear combination of these two objectives:

$$L(s) = \alpha \cdot \text{cost}(s) - \text{margin}(s, h), \tag{3.1}$$

where $\text{cost}(s)$ and $\text{margin}(s, h)$ denote the total cost of features in s and the classification margin given by h using all selected features, and α is a user-specified parameter to trade-off cost and accuracy. In general, this loss function is only defined for the terminal state (after the STOP action is taken), since the cost function and the classification margin do not necessarily decompose over each factor. In the simplest case where costs of each feature are independent and the task predictor h is a linear classifier, we can define the immediate loss as the cost of a newly added factor and the classification margin using only the new factor.

Given the delayed loss and the exponential state space (2^D), we take the imitation learning approach. Below We describe how we obtain an expert efficiently.

Expert Ideally, the expert should find a subset of features achieving the minimum terminal loss for each instance. However, we have too large a state space (exponential) to search for the optimal subset of features exhaustively. In addition, given a state, the expert action may not be unique since the optimal subset of features does not have to be selected in a fixed order. Therefore, we use a (sub-optimal) greedy forward-selection expert. At time step t, the expert iterates through all features in the action space A_t and calculates a one-step lookahead cost for each action $C(s_t, a)$

($a \in A_t$); it then chooses the action with the minimum cost. The action cost is defined based on L. Let $s_{t+1}|a$ be the tentative next state assuming action a is taken, then $C(s_t, a) = L(s_{t+1}|a)$ is the loss of the next state. Note that the STOP action does not change the current state (i.e. $s_t = s_{t+1}$), therefore the expert terminates if no factor improves the current loss $L(s_t)$. Even though the greedy expert finds a sub-optimal feature set, we find it often achieves high accuracy with a rather small cost in our experiments.

Given the above expert, we are ready to apply imitation learning to learn a feature selection policy. Recall that the core learning part of DAGGER reduces to a supervised learning algorithm. While the expert provides labels in each state, we still need to define features describing the state.

State Features We include two types of features: *task features* and *meta-features*. Task feature are simply features selected so far and they are defined by the specific classification problem, e.g. different tests performed to diagnose a patient. Meta-features are computed based on feature costs and past predictions given by the task predictor. They are used to evaluate confidence of the current prediction. Task features are useful to decide the next feature to add, and meta-features are useful to decide when to stop.

We define the state feature $\phi(s)$ as a concatenation of task features and meta-features. Our meta-features look at current scores/cost given by h and the change in scores/cost compared to the last time step. More specifically, let the confidence score of h be score of the predicted class, and we have the following meta-features:

confidence score, change in confidence score after adding the last factor, a boolean bit indicating whether the prediction changed after adding the last factor, cost of the current feature set, change in cost after adding the last factor, the cost-efficient index (cost divided by confidence score), and the current prediction.

Now that we have all components needed for learning a policy, we can run DAgger with our favorite supervised learning algorithm. However, we find in the initial experiments that a large gap exists between the performance of the expert policy and the learned policy. We investigate this phenomenon in the next section.

3.3 Learning when the Expert is Too Good

Recall that the guarantee of DAgger (Theorem 2) depends on the imitation error ϵ_{class} in the supervised learning step. If ϵ_{class} is small, DAgger can always find a good policy. However, sometimes it can be hard to obtain a good classifier that has low training error using a typical supervised learning algorithm. This occurs when the learner's policy space is not able to approximate the expert's policy.

An oracle can be hard to imitate in two ways. First, the learning policy space is not able to approximate the expert's policy, meaning that the learner has limited learning ability. For example, the actual loss function can be highly non-convex, which suggests a linear predictor is not enough and a better class of predictors is needed. Second, information known by the expert cannot be sufficiently inferred from the state features, meaning that the learner has limited learning resources. For example, in dynamic feature selection, the expert knows the ground truth. Thus,

it can evaluate each action by the true loss, while the learner has to infer this information from the state features.

To address this problem, we propose to *coach* the learner with easy-to-learn actions that gradually approach the expert actions. Intuitively, this allows the learner to move towards a better action without much effort, instead of aiming at an impractical goal from the very beginning. To better instruct the learner, a coach should demonstrate actions that are not much worse than the expert action but are easier to achieve within the learner's policy space. The lower an action's cost is, the closer it is to the expert action. The higher an action is ranked by the learner's current policy, the more it is preferred by the learner, thus easier to learn. Therefore, similar to (Chiang et al., 2008), we define a *hope action* that combines the action cost and the preference of the learner's current policy. We use $\tilde{\pi}_i$ to denote the coach in place of the expert in iteration i. Let $\text{score}_{\pi_i}(s, a)$ be a measure of how likely π_i chooses action a in state s. We define $\tilde{\pi}_i$ as

$$\tilde{\pi}_i(s) = \arg\max_{a \in A} \lambda_i \cdot \text{score}_{\pi_i}(s, a) - C(s, a) \tag{3.2}$$

where λ_i is a nonnegative parameter specifying how close the coach is to the oracle. We set $\lambda_1 = 0$ as the learner has not learned any model yet in the first iteration. The coaching algorithm is exactly the same as DAGGER, except that we use $\tilde{\pi}_i$ instead of π^* in each iteration. We refer to the proposed algorithm as Coaching below.

Theoretical Analysis We analyze the coaching algorithm under the theoretical framework of DAGGER (Section 2.3). We show that it achieves the same relative bound as DAGGER but with a possibly smaller ϵ_{class}. Let $\tilde{\ell}_i(\pi) = \mathbb{E}_{s\sim d_{\pi_i}}[\ell(s, \pi, \tilde{\pi}_i(s))]$ denote the expected surrogate loss with respect to $\tilde{\pi}_i$. We define the minimum loss of the best policy in hindsight with respect to hope actions as

$$\tilde{\epsilon}_{class} = \frac{1}{N} \min_{\pi\in\Pi} \sum_{i=1}^{N} \tilde{\ell}_i(\pi). \tag{3.3}$$

Our main result is the following theorem:

Theorem 4. *Assume ℓ upper bounds L, π^* is u-robust, and N is $O(uT \log T)$, then there exists a policy $\pi \in \pi_{1:N}$ such that:*

$$J(\pi) \leq J(\pi^*) + uT\tilde{\epsilon}_{class} + O(1). \tag{3.4}$$

Both the DAgger theorem and the coaching theorem provide a relative guarantee. They depend on whether we can find a policy that has small training error in each iteration. The distinction is that our guarantee depends on $\tilde{\epsilon}_{class}$, which is resulted from learning the easier coaching actions, while DAGGER may fail to find a policy with small ϵ_{class}. Through coaching, we can always adjust λ to create a more learnable expert policy space to enable low training error, at the price of running a few more iterations. Proof of the theorem can be found in Appendix A.

3.4 Experiments

Setup We perform experiments on three UCI datasets (radar signal, digit recognition, image segmentation) and assign random costs to features. For all datasets, the data classifier are trained using MegaM (Daumé III, 2004). However, since we assume the provided classifier is to be used at test time, using it at training time may cause a difference in the distribution of training and test data for feature selection. For example, the confidence level in $\phi(s)$ during training can be much higher that that during testing. Therefore, similar to cross validation, we split the training data into 10 folds. We collect trajectories on each fold using a data classifier trained on the other 9 folds. This provides a better simulation of the environment at test time. For the digit dataset, we split the 16×16 image into non-overlapping 4×4 blocks and each factor contains the 16 pixel values in a block. For the other two datasets, each factor contains one feature.

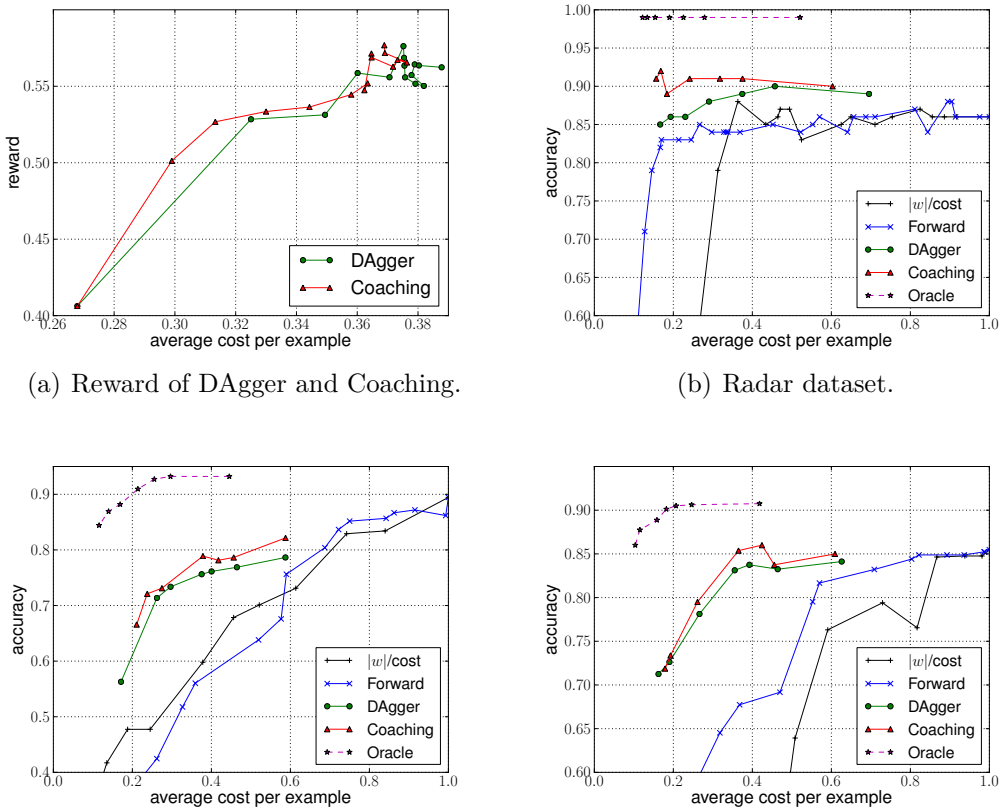
For dynamic feature selection, we use DAgger and coaching for policy learning. We run both algorithms for 15 iterations and take the best policy evaluated on the development set. For coaching, we schedule λ with exponential decay, such that $\lambda_1 = 0$ and $\lambda_i = e^{-i+2}$ for $i > 1$ where i is the iteration number. We use LIBSVM to learn the multiclass classifier.

Evaluation We compare with two static feature selection baselines that sequentially add features according to a ranked list. The first baseline (Forward) ranks factors according to the standard forward feature selection algorithm without cost.

The second baseline ($|w|$/cost) ranks the factors by the ratio $|w|$/cost, where $|w|$ denotes the sum of absolute weights of features in a factor. Features with high ratios are expected to be more cost-efficient.

Since we have two competing objectives, reducing feature cost and improving prediction quality, we need a measurement for comparing how much we can gain in one objective without making the other objective worse. Therefore, we evaluate the algorithms by their Pareto curves that show different trade-off results on a plane formed by axes of both objectives. More specifically, for our dynamic feature selection algorithms, we control the parameter α to obtain a set of results at increasing costs. We experiment with α values from $\{0.0, 0.1, 0.25, 0.5, 1.0, 1.5, 2.0\}$. For the two baseline algorithms, we take the performance after adding each factor to get accuracy at different costs.

Result We show the Pareto curves of different algorithms in Figure 3.1. First, we can see that dynamically selecting features for each instance significantly improves the accuracy at a small cost. Sometimes, it even achieves higher accuracy than using all features. Second, we notice that there is a substantial gap between performance of the learned policy and the expert policy, however, in almost all settings Coaching dominates DAgger, i.e. achieving higher accuracy at a lower cost. From Figure 3.1(a), we see that coaching reduces the gap by taking small steps towards the oracle. Nevertheless, the learned policy hardly imitates the expert policy, as coaching is still inherently limited by the insufficient policy space. We expect better performance with predictors of higher capacity, e.g. neural networks.



(a) Reward of DAgger and Coaching.

(b) Radar dataset.

(c) Digit dataset.

(d) Segmentation dataset.

Figure 3.1: (a) reward (negative loss) vs. cost of DAGGER and Coaching over 15 iterations on the digit dataset with $\alpha = 0.5$. (c) to (d) show accuracy vs. cost of each of the three datasets. For DAGGER and Coaching, we show results with $\alpha \in \{0, 0.1, 0.25, 0.5, 1.0, 1.5, 2\}$.

3.5 Related Work

As the scale of machine learning problems become larger, learning information gathering policies has received more attention in cost-sensitive classification. However, many of the proposals in this space use general MDP techniques, which are sufficient but perhaps not necessary given the constrained, deterministic world of sequential selection.

Our work is closest to Dulac-Arnold (2014), where a datum-wise representation is learned by selecting features for each instance adaptively in a sequential decision-making process. There are a couple of distinctions. First, their primary goal is to learn a sparse representation per data point and to approximate the L_0 regularization, while we aim to learn a cost-accuracy trade-off. Therefore, our loss function explicitly quantifies the trade-off. The second distinction is mainly technical. They take the reinforcement learning approach and use a single policy for both the task predictor and the feature selector. This formulation has a larger search space compared to ours and does not leverage pre-training of the task predictor. In addition, it might face difficulty in complex domains where the predictor and the selector need different function classes.

Our work is also related to Benbouzid et al. (2012). We both formulate classification as sequential decision-making and solve it under the MDP framework. The difference is that here features are pre-ranked; at test time, the policy takes features from the ranked list and choose to add or skip the next feature. However, in our work and Dulac-Arnold (2014), the policy can choose *any* feature. The advantage of free selection is that the model enjoys larger flexibility. The disadvantage is that it increases the action space, thus implies more a more difficult learning problem. In the next chapter, we will also use a pre-ranked list of features due to a more complex problem setting.

Other recent feature selection or budgeted learning methods include classifier ensemble methods which adaptively select a subset of classifiers to evaluate, including tree-based algorithms (Xu et al., 2013; Kusner et al., 2014), boosting style

algorithms (Grubb and Bagnell, 2012; Reyzin, 2011), and probabilistic approaches based on value-of-information (Gao and Koller, 2011). In addition, cascading approaches incorporate test-time cost constraint by a coarse-to-fine classifier cascade and an early-stop strategy (Chen et al., 2012; Weiss and Taskar, 2010).

3.6 Conclusion

In this chapter, we showed how to formulate the task of sequential feature selection for cost-sensitive classification inside the imitation learning paradigm. We proposed a computationally simple reference policy (that has access to the training labels) and used imitation learning to compete with it, avoiding the difficulties of more general reinforcement learning techniques. We also proposed a loss function that explicitly balances the trade-off between the task loss and the cost of features.

More generally, the method introduced in this chapter demonstrates the effectiveness of our dynamic solver on sequential acquisition problems. In addition, it shows that imitation learning is a promising tool for solving problems where an expert policy can be constructed at training time. Fruitful directions include adapting and extending the ideas we presented to more complex domains: In the next chapter, we extend this method to structured prediction with more sophisticated feature space and output space.

Chapter 4: Graph-based Dependency Parsing

This chapter describes joint work with Hal Daumé III and Jason Eisner (He et al., 2013) in EMNLP 2013.

In the last chapter, we have seen how classification can be solved sequentially to trade off cost and accuracy. However, applying the framework to structured problems is not straightforward, as the input space and the output space now consist of exponentially many local substructures. In this chapter, we focus on a structured prediction problem, dependency parsing. Structured prediction is defined by an input space $\mathcal{X}$ and an output space $\mathcal{Y}$. Unlike in multiclass classification, where $\mathcal{Y}$ is a set of discrete values, here it is a set of discrete structures. Therefore, a structured prediction model must make a joint prediction over mutually dependent output variables that form a structure in $\mathcal{Y}$. In NLP, input $\mathcal{X}$ is usually a piece of text (e.g. sentences, documents), and $\mathcal{Y}$ is a linguistic structure that we are interested in, (e.g. a parse tree, word alignment between two sentences), or text of certain relation to the input text (e.g. summarization, translation).

Compared to settings in the previous chapter, we have two complexities in structured prediction. First, features are defined on independent parts (substructures) of the input. Second, predicting the output requires solving a combinatorial

optimization problem over $\mathcal{Y}$. In this chapter, we propose a method that answers the following questions in the context of dependency parsing: (a) how to arrange a large number of features so that they can be selected by groups; (b) how to adaptively select features for each part of the structured input space while keeping the overall decision overhead low; (c) how to adaptively make predictions for each part of the structured output space while considering all parts jointly.

We begin by introducing common algorithms for graph-based dependency parsing, and then describe our dynamic feature selection algorithm that speeds up the process.

4.1 Graph-based Dependency Parsing

In Section 1.2.1, we briefly introduced the task of dependency parsing; in this section, we give a more detailed description of the minimum-spanning tree (MST) parsing algorithm (McDonald et al., 2005a; McDonald and Pereira, 2006; McDonald et al., 2005b). We will use the MST parser as our task (batch) predictor and aim to speed it up by reducing features computation cost.

In a typical structured prediction problem, searching for the best structure in the exponentially large space $\mathcal{Y}$ is often intractable. Therefore, the output space is decomposed into substructures. The problem is usually solved in two steps. First, each structure is scored by taking the sum of scores of its substructures, which are typically computed by a linear model based on features describing a substructure. Second, the highest-scoring structure can be found by dynamic programming thanks

to the decomposition (Eisner, 1996; McDonald, 2007; Jurafsky and Martin, 2000) or by integer linear programming (Gillick and Favre, 2009; Martins et al., 2009; Roth and tau Yih, 2005). The second step is often called decoding or inference.

In graph-based dependency parsing of an n-word input sentence, we must construct a tree $\boldsymbol{y}$ whose vertices $0, 1, \ldots n$ correspond to the root node (namely 0) and the ordered words of the sentence. Each directed edge of this tree points from a head (parent) to one of its modifiers (child).

We follow a common approach to structured prediction problems. In the scoring stage, we score all possible edges (or other small substructures) using a learned function; in the decoding stage, we use dynamic programming to find the dependency tree with the highest *total* score. We describe the two steps in detail and examine their empirical computation cost below.

Scoring The score of a tree $\boldsymbol{y}$ is defined as a sum of local scores. That is, $s_{\boldsymbol{\theta}}(\boldsymbol{y}) = \boldsymbol{\theta} \cdot \sum_{E \in \boldsymbol{y}} \phi(E) = \sum_{E \in \boldsymbol{y}} \boldsymbol{\theta} \cdot \phi(E)$, where E ranges over small connected subgraphs of $\boldsymbol{y}$ that can be scored individually. Here $\phi(E)$ extracts a high-dimensional feature vector from E together with the input sentence, and $\boldsymbol{\theta}$ denotes a weight vector that has typically been learned from data. The first-order model decomposes the tree into edges E of the form $\langle h, m \rangle$, where $h \in [0, n]$ and $m \in [1, n]$ (with $h \neq m$) are a head token and one of its modifiers. Finding the best tree requires first computing $\boldsymbol{\theta} \cdot \phi(E)$ for each of the n^2 possible edges.

The features are computed according to feature templates. In Table 4.1, we show example feature templates for a potential dependency edge. A typical parser

Edge	Feature Template	Feature Instantiation
This ← time	head word	time
	mod word	This
	head word+mod word	time+This
	head pos	NN
	mod pos	DT
	head pos+mod pos	NN+DT
	head word+mod word+head pos+mod pos	time+This+NN+DT
	edge direction	right

Table 4.1: Example feature templates and instantiations. In feature templates, "mod" means modifier and "pos" means part-of-speech tag.

usually uses hundreds of feature templates (e.g. 268 for our first-order parser). The number of features (template instantiations) can be millions (e.g. 76 million for our first-order parser on PTB).

Since scoring the edges independently in this way restricts the parser to a local view of the dependency structure, higher-order models can achieve better accuracy. For example, in the second-order model of McDonald and Pereira (2006), each local subgraph E is a triple that includes the head and two modifiers of the head, which are adjacent to each other. Other methods that use triples include grandparent-parent-child triples (Koo and Collins, 2010), or non-adjacent siblings (Carreras, 2007). Third-order models (Koo and Collins, 2010) use quadruples, employing grand-sibling and tri-sibling information.

Decoding The usual inference problem is to find the highest scoring tree for the input sentence. Note that in a valid tree, each token $1, \ldots, n$ must be attached to exactly one parent (either another token or the root 0). We can further require the tree to be projective, meaning that edges are not allowed to cross each other. It

is well known that dynamic programming can be used to find the best projective dependency tree in $O(n^3)$ time, much as in CKY, for first-order models and some higher-order models (Eisner, 1996; McDonald and Pereira, 2006). When the projectivity restriction is lifted, McDonald et al. (2005b) pointed out that the best tree can be found in $O(n^2)$ time using a minimum directed spanning tree algorithm (Chu and Liu, 1965; Edmonds, 1967; Tarjan, 1977), though only for first-order models.[1] We will make use of this fast non-projective algorithm as a subroutine in early stages of our system.

Analyzing Time Usage We have assumed that feature computation is bottleneck in dependency parsing. Therefore, we first testify the assumption by investigating time usage during parsing. Specifically, we take the MSTParser[2] and measure the wall-clock time spent on scoring and decoding. We use the development set, section 22 of the Penn Treebank (PTB) (Marcus et al., 1993) for profiling.

In Figure 4.1, we show the average scoring and decoding time of sentences at different lengths, and we measure for both first-order (projective and non-projective) and second-order (projective) parsing. We observe that (a) feature computation took more than 80% of the total time; (b) even though non-projective decoding time grows quadratically in terms of the sentence length, in practice, it is almost negligible (0.23 ms on average) compared to the projective decoding time; (c) the second-order projective model is significantly slower due to higher time complexity

[1] The non-projective parsing problem becomes NP-hard for higher-order models. One approximate solution (McDonald and Pereira, 2006) works by doing projective parsing and then rearranging edges.

[2] Available at http://www.seas.upenn.edu/~strctlrn/MSTParser/MSTParser.html

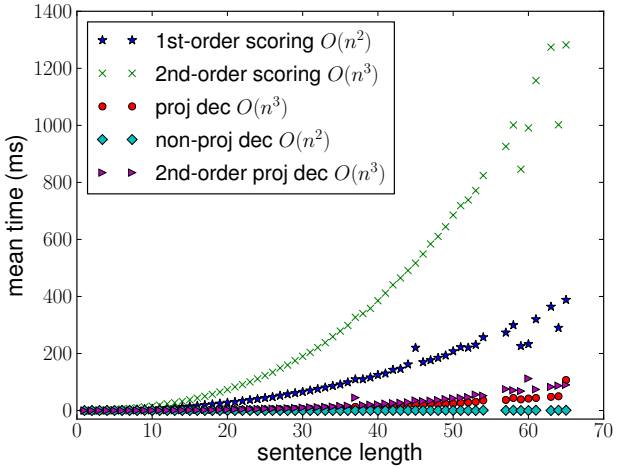


Figure 4.1: Comparison of scoring time and decoding time on English PTB section 22 using multiple parsers.

in both scoring and decoding.

The actual time usage depends largely on how the parser is implemented.[3] However, we believe feature extraction is the major computation block since it grows with the number and complexity of features, which can be a large hidden constant in the time complexity. In addition, large time consumption of feature extraction has been reported in other work (Bohnet, 2010) as well. In the next section, we discuss how to adapt the dynamic feature selection framework described in the last chapter to dependency parsing, specifically the MSTParser.

4.2 Dynamic Feature Selection for Parsing

Using the analogy of feature selection for classification, here each edge is an example to be classified as either a true edge in the ground truth parse tree or a false

[3]We changed the feature computation part in the original MSTParser to make to more efficient. Specifically, instead of hashing a feature string, we hash an integer by encoding the features into a sequence of bytes similar to Bohnet (2010).

edge, and we select feature templates to instantiate. There are two main challenges. First, given hundreds of templates, it is not efficient to select one template at a time considering the decision overhead between two steps: running the policy and the decoder (for intermediate prediction). Second, making decisions and maintaining a different set of features for each edge (n^2 in total) is expensive. Therefore, we make two simplifications to reduce the number of decisions needed. First, we group and rank all features beforehand, such that we only make decisions about whether to add more features or to stop. Second, instead of making separate decisions for each edge, we make decisions for important edges only and use structural constraints to decide for other edges. Recall that in dynamic feature selection we make intermediate predictions after new features are added. The key idea is to make decisions for edges in the intermediate parse tree only: these are the important edges which are likely to appear in the final parse tree. In the following, we give an overview of our parsing algorithm followed by a detailed description.

4.2.1 Overview

We show the parsing process for one short sentence in Figure 4.2. The first step is to parse using the first feature group. We use the fast first-order non-projective parser for intermediate prediction since given observations (b) and (c) from the time usage analysis, we cannot afford to run projective parsing multiple times. The single resulting tree (blue and red edges in Figure 4.2) has only n edges, and we use a classifier to decide which of these edges are reliable enough that we should "lock"

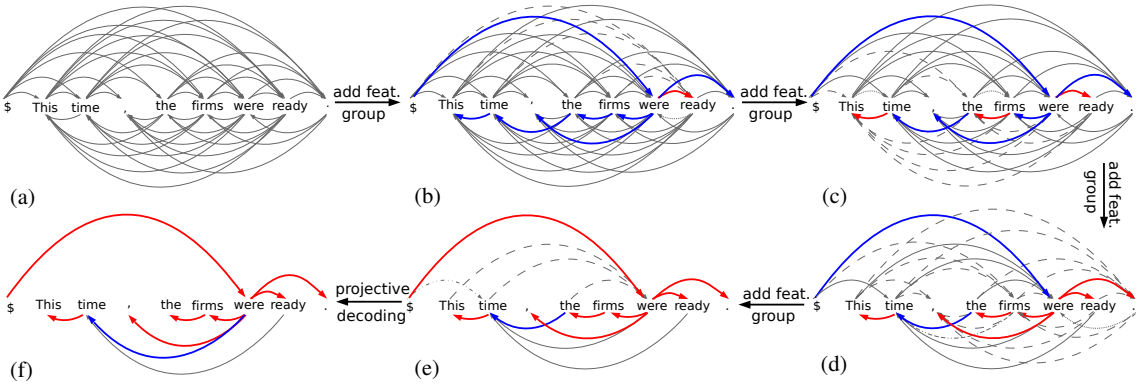


Figure 4.2: Overview of dynamic feature selection for dependency parsing. (a) Start with all possible edges except those filtered by the length dictionary. (b) – (e) Add the next group of feature templates and parse using the non-projective parser. Predicted trees are shown as blue and red edges, where red indicates the edges that we then decide to lock. Dashed edges are pruned because of having the same child as a locked edge; 2-dot-3-dash edges are pruned because of crossing with a locked edge; fine-dashed edges are pruned because of forming a cycle with a locked edge, and 2-dot-1-dash edges are pruned since the root has already been locked with one child. (f) Final projective parsing.

them—i.e., commit to including them in the final tree. This is the only decision that our policy π must make. Locked (red) edges are definitely in the final tree. We then do constraint propagation to make decisions for other edges: we rule out all edges that conflict with the locked edges, barring them from appearing in the final tree.[4] Conflicts are defined as violation of the projective parsing constraints:

- Each word has exactly one parent;
- Edges cannot cross each other;[5]
- The directed graph is non-cyclic;

[4]Constraint propagation also automatically locks an edge when all other edges with the same child have been ruled out.

[5]Naively, the cost of finding edges that cross a locked edge is $O(n^2)$. But at most n edges will be locked during the entire algorithm, for a total $O(n^3)$ runtime—the same as *one* call to projective parsing, and far faster in practice. With cleverness, this can even be reduced to $O(n^2 \log n)$ by using a k-d tree to store edges as two-dimensional ranges.

- Only one word is attached to the root.

Once constraint propagation has finished, we visit all edges (gray) whose fate is still unknown and update their scores in parallel by adding the next group of features. The process is repeated unless all edges are determined or no more feature is left.

As a result, most edges will be locked in or ruled out without needing to look up all of their features. Some edges may still remain uncertain even after including all features. If so, a final iteration (Figure 4.2 (f)) uses the slower projective parser to resolve the status of these maximally uncertain edges. In our example, the parser does not figure out the correct parent of *time* until this final step. This final, accurate parser can use its own set of weighted features, including higher-order features, as well as the projectivity constraint. But since it only needs to resolve the few uncertain edges, both scoring and decoding are fast.

If we wanted our parser to be able to produce non-projective trees, then we would skip this final step or have it use a higher-order non-projective parser, and we would not prune edges crossing the locked edges.

4.2.2 The Full Algorithm

We now describe our parsing algorithm in details.

pre-trained Parsers Same as in the classification setting, we use pre-trained task predictors to generate intermediate and final outputs. We assume that we are given three increasingly accurate but increasingly slow parsers that can be called as subroutines: a first-order non-projective parser, a first-order projective parser, and

a second-order projective parser. In all cases, their feature weights have already been trained using the *full* set of features, and we will not change these weights. In general, we will return the output of one of the projective parsers. But at early iterations, the non-projective parser helps us rapidly consider interactions among edges that may be relevant to our dynamic decisions.

Feature Template Ranking We first rank the 268 first-order feature templates by forward selection. We start with an empty list of feature templates, and at each step, we greedily add the one whose addition most improves the parsing accuracy on a development set. Since some features may be slower than others (for example, the "between" feature templates require checking all tokens in-between the head and the modifier), we could instead select the feature template with the highest ratio of accuracy improvement to runtime. However, for simplicity we do not consider this: after grouping (see below), minor changes of the ranks within a group have no effect. The accuracy is evaluated by running the first-order non-projective parser since we will use it to make most of the decisions. The 112 second-order feature templates are then ranked by adding them in a similar greedy fashion (given that all first-order features have already been added), evaluating with the second-order projective parser.

We then divide this ordered list of feature templates into K groups: $T_1, \ldots, T_K$. Our parser adds an entire group of feature templates at each step, since adding one template at a time would require too many decisions and obviate speedups. The simplest grouping method would be to put an equal number of feature templates in

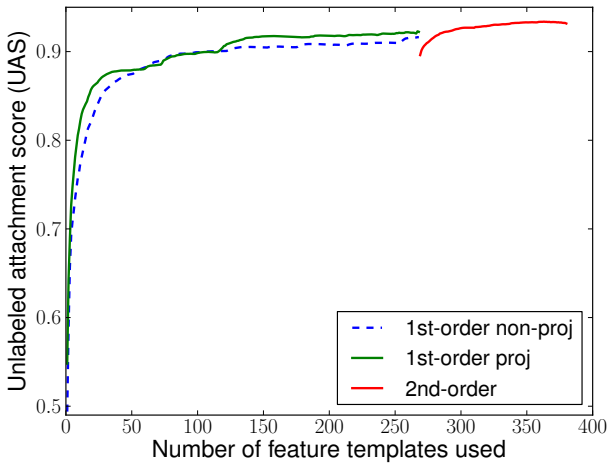


Figure 4.3: Forward feature selection result using the non-projective model on English PTB section 22.

each group. From Figure 4.3 we can see that the accuracy increases significantly with the first few templates and gradually levels off as we add less valuable templates. Thus, a more cost-efficient method is to split the ranked list into several groups so that the accuracy increases by roughly the same amount after each group is added. In this case, earlier stages are fast because they tend to have many fewer feature templates than later stages. For example, for English, we use 7 groups of first-order feature templates and 4 groups of second-order feature templates. The sequence of group sizes is 1, 4, 10, 12, 47, 33, 161 and 35, 29, 31, 17 for first- and second-order parsing respectively.

Sequential Feature Selection and Edge Pruning Our test time parsing procedure is shown in Algorithm 3.

Similar to the length dictionary filter of Rush and Petrov (2012), for each test sentence, we first deterministically remove edges longer than the maximum length of

Algorithm 3 DynFS($\mathcal{G}(\mathcal{V}, \mathcal{E})$, π)

1: $\mathcal{E} \leftarrow \{\langle h, m\rangle \colon |h - m| \leq \text{lenDict}(h, m)\}$
2: Add T_1 to all edges in $\mathcal{E}$
3: $\hat{\boldsymbol{y}} \leftarrow$ non-projective decoding
4: **for** $i = 2$ to K **do**
5: $\quad \mathcal{E}_{\text{sort}} \leftarrow$ sort unlocked edges $\{E : E \in \hat{\boldsymbol{y}}\}$ in descending order of their scores
6: $\quad$ **for** $\langle h, m\rangle \in \mathcal{E}_{\text{sort}}$ **do**
7: $\quad\quad$ **if** $\pi(\psi(\langle h, m\rangle)) == \mathit{lock}$ **then**
8: $\quad\quad\quad \mathcal{E} \leftarrow \mathcal{E} \setminus \{\{\langle h', m\rangle \in \mathcal{E} \colon h' \neq h\} \cup \{\langle h', m'\rangle \in \mathcal{E} \colon \text{ cross } \langle h, m\rangle\} \cup \{\langle h', m'\rangle \in \mathcal{E} \colon \text{ cycle with } \langle h, m\rangle\}\}$
9: $\quad\quad\quad$ **if** $h == 0$ **then**
10: $\quad\quad\quad\quad \mathcal{E} \leftarrow \mathcal{E} \setminus \{\langle 0, m'\rangle \in \mathcal{E} \colon m' \neq m\}$
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **else**
13: $\quad\quad\quad$ Add T_i to $\{\langle h', m'\rangle \in \mathcal{E} \colon m' == m\}$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: $\quad$ **if** $i == K$ **then**
17: $\quad\quad \hat{\boldsymbol{y}} \leftarrow$ projective decoding
18: $\quad$ **else if** $i \neq K$ **or** FAIL **then**
19: $\quad\quad \hat{\boldsymbol{y}} \leftarrow$ non-projective decoding
20: $\quad$ **end if**
21: **end for**
22: **Return** $\hat{\boldsymbol{y}}$

edges in the training set that have the same head part-of-speech (POS) tag, modifier POS tag, and direction (Algorithm 3, Line 1). This simple step prunes around 40% of the non-gold edges in our PTB development set at a cost of less than 0.1% in accuracy.

Given a test sentence of length n, we start with a complete directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{E} = \{\langle h, m\rangle \colon h \in [0, n], m \in [1, n]\}$. After the length dictionary pruning step, we compute T_1 for all remaining edges to obtain a pruned weighted directed graph. We predict a parse tree using the features so far (other features are treated as absent, with value 0). Then for each edge in this intermediate tree, we use a policy π (binary linear classifier) to choose between two actions: $A = \{\mathit{lock}, \mathit{add}\}$.

The *lock* action ensures that $\langle h, m \rangle$ appears in the final parse tree by pruning edges that conflict with $\langle h, m \rangle$ (Algorithm 3, Line 8).[6] If the classifier is not confident enough about the parent of m, it decides to *add* to gather more information. The *add* action computes the next group of features for $\langle h, m \rangle$ and all other competing edges with child m. Since we classify the edges one at a time, decisions on one edge may affect later edges. To improve efficiency and reduce cascaded error, we sort the edges in the predicted tree and process them as above in descending order of their scores (Algorithm 3, Line 5).

Now we can continue with the second iteration of parsing. Overall, our method runs up to $K = K_1 + K_2$ iterations on a given sentence, where we have K_1 groups of first-order features and K_2 groups of second-order features. We run $K_1 - 1$ iterations of non-projective first-order parsing (adding groups $T_1, \ldots, T_{K_1-1}$), then 1 iteration of *projective* first-order parsing (adding group T_{K_1}), and finally K_2 iterations of projective second-order parsing (adding groups $T_{K_1+1}, \ldots T_K$).

Before each iteration, we use the result of the previous iteration (as explained above) to prune some edges and add a new group of features to the rest. We then run the relevant parser. Each of the three parsers has a different set of feature weights, so when we switch parsers on rounds K_1 and $K_1 + 1$, we must also *change* the weights of the previously added features to those specified by the new parsing model.

In practice, we can stop as soon as the fate of all edges is known. Also, if no

[6] If the conflicting edge is in the current predicted parse tree (which can happen because of non-projectivity), we forbid the model to prune it. Otherwise, the non-projective parser at the next stage may fail to find a tree in rare cases.

projective parse tree can be constructed at round K_1 using the available unpruned edges, then we immediately fall back to returning the non-projective parse tree from round $K_1 - 1$ (Algorithm 3, Line 18–20). This FAIL case rarely occurs in our experiments (fewer than 1% of sentences).

We have defined the search space and decisions the policy must make. Next, we describe how to learn such a policy using imitation learning.

4.2.3 Policy Learning

We follow the same learning algorithm introduced in Chapter 3. In this section, we describe components required to run DAGGER.

Expert In our case, the expert's decision is rather straightforward. Replace the policy π in Algorithm 3 by an expert. If the edge under consideration is a gold edge, it executes *lock*; otherwise, it executes *add*. Basically the expert "cheats" by knowing the true tree and always making the right decision. On our PTB dev set, it can get 96.47% accuracy[7] with only 2.9% of the first-order features. This is an upper bound on our performance.

pre-trained Parsers We obtain the pre-trained parsers in the same way as in the last chapter. We partition the training sentences $\mathcal{T}$ into N folds $\mathcal{T}^1, \mathcal{T}^2, \ldots, \mathcal{T}^N$. To simulate parsing results at test time, when collecting examples on $\mathcal{T}^i$, using parsers trained on $\mathcal{T} \setminus \mathcal{T}^i$, much as in cross-validation.

[7] The imperfect performance is because the accuracy is measured with respect to the gold parse trees. The expert only makes optimal pruning decisions but the performance depends on the pre-trained parser as well.

State Features We use a feature map ψ that concatenates all previously acquired parsing features together with meta-features that reflect confidence in current prediction of the edge. The meta-features are based on scores of each edge given features added so far. Since each modifier can have only one head, it makes sense to normalize scores of competing edges which have the same modifier. We standardize the edge scores by a sigmoid function. Formally, let $\dot{s}$ denote the normalized score, defined by $\dot{s}_{\boldsymbol{\theta}}(\langle h, m\rangle) = 1/(1 + \exp\{-s_{\boldsymbol{\theta}}(\langle h, m\rangle)\})$. Our meta-features for $\langle h, m\rangle$ include:

- current normalized score, and normalized score before adding the current feature group;
- margin to the highest scoring competing edges, i.e., $\dot{s}(\boldsymbol{w}, \langle h, m\rangle) - \max_{h'} \dot{s}(\boldsymbol{w}, \langle h', m\rangle)$ where $h' \in [0, n]$ and $h' \neq h$;
- index of the next feature group to be added.

We also tried more complex meta-features, for example, mean and variance of the scores of competing edges, and structured features such as whether the head of e is locked and how many locked children it currently has. It turns out that *given all the parsing features*, the margin is the most discriminative meta-feature. We believe it is because the margin is the most direct indication of how confident a prediction is after we normalize scores of competing edges to a probability distribution. When it is present, other meta-features we added do not help much. Thus, we do not include them in our experiments due to overhead.

4.3 Experiments

Setup We generate dependency structures from the PTB constituency trees using the head rules of Yamada and Matsumoto (2003). Following convention, we use sections 02–21 for training, section 22 for development and section 23 for testing. We also report results on six languages from the CoNLL-X shared task (Buchholz and Marsi, 2006) as suggested in (Rush and Petrov, 2012), which cover a variety of language families. We follow the standard training/test split specified in the CoNLL-X data and tune parameters by cross validation when training the classifiers (policies). The PTB test data is tagged by a Stanford POS tagger (Toutanova et al., 2003) trained on sections 02–21. We use the provided gold POS tags for the CoNLL test data. All results are evaluated by the unlabeled attachment score (UAS). For a fair comparison with previous work, punctuation is included when computing parsing accuracy of all CoNLL-X languages but not English (PTB).

For policy training, we train a linear SVM classifier using LIBLINEAR (Fan et al., 2008). For all languages, we run DAGGER for 20 iterations and select the best policy evaluated on the development set among the 20 policies obtained from each iteration.

Evaluation Our baseline parser is the publicly available implementation of MST-Parser[8] (with modifications to the feature computation) and its default settings, so the feature weights of the projective and non-projective parsers are trained by the

[8] http://www.seas.upenn.edu/~strctlrn/MSTParser/MSTParser.html

MIRA algorithm (Crammer and Singer, 2003; Crammer et al., 2006).

The parsing features contains most features proposed in the literature (McDonald et al., 2005a; Koo and Collins, 2010). The basic feature components include lexical features (token, prefix, suffix), POS features (coarse and fine), edge length and direction. The feature templates consist of different conjunctions of these components. Other than features on the head word and the child word, we include features on in-between words and surrounding words as well. For PTB, our first-order model has 268 feature templates and 76,287,848 features; the second-order model has 380 feature templates and 95,796,140 features. The accuracy of our full-feature models is comparable or superior to previous results.

We compare with the baseline parser and the vine pruning cascade parser (Rush and Petrov, 2012) in terms of speedup in wall-clock time and parsing accuracy. The vine pruning cascade parser gains speedup by coarse-to-fine projective parsing cascades (Charniak et al., 2006), where they take multiple passes of parsing, and prune edges in each pass. While we focus on reducing feature computation, they focus on reducing decoding time.

Result In Table 4.2, we compare the dynamic parsing models with the full-feature models and the vine pruning cascade models for first-order and second-order parsing. The *speedup* for each language is defined as the speed relative to its full-feature baseline model. We take results reported by Rush and Petrov (2012) for the vine pruning model. As speed comparison for parsing largely relies on implementation, we also report the percentage of feature templates chosen for each sentence. The *cost*

column shows the average number of feature templates computed for each sentence, expressed as a percentage of the number of feature templates if we had only pruned using the length dictionary filter.

Language	Method	First-order				Second-order			
		Speedup	Cost(%)	UAS(D)	UAS(F)	Speedup	Cost(%)	UAS(D)	UAS(F)
Bulgarian	DynFS	3.44	34.6	91.1	91.3	4.73	16.3	91.6	92.0
	VineP	3.25	-	90.5	90.7	7.91	-	91.6	92.0
Chinese	DynFS	2.12	42.7	91.0	91.3	2.36	31.6	91.6	91.9
	VineP	1.02	-	89.3	89.5	2.03	-	90.3	90.5
English	DynFS	5.58	24.8	91.7	91.9	5.27	49.1	92.5	92.7
	VineP	5.23	-	91.0	91.2	11.88	-	92.2	92.4
German	DynFS	4.71	21.0	89.2	89.3	6.02	36.6	89.7	89.9
	VineP	3.37	-	89.0	89.2	7.38	-	90.1	90.3
Japanese	DynFS	4.80	15.6	93.7	93.6	8.49	7.53	93.9	93.9
	VineP	4.60	-	91.7	92.0	14.90	-	92.1	92.0
Portuguese	DynFS	4.36	32.9	87.3	87.1	6.84	40.4	88.0	88.2
	VineP	4.47	-	90.0	90.1	12.32	-	90.9	91.2
Swedish	DynFS	3.60	37.8	88.8	89.0	5.04	22.1	89.5	89.8
	VineP	4.64	-	88.3	88.5	13.89	-	89.4	89.7

Table 4.2: Comparison of speedup and accuracy with the vine pruning cascade approach for six languages. In the setup, DynFS means our dynamic feature selection model, VineP means the vine pruning cascade model, UAS(D) and UAS(F) refer to the unlabeled attachment score of the dynamic model (D) and the full-feature model (F) respectively. For each language, the speedup is relative to its corresponding first- or second-order model using the full set of features. Results for the vine pruning cascade model are taken from Rush and Petrov (2012). The cost is the percentage of feature templates used per sentence on edges that are *not pruned by the dictionary filter*.

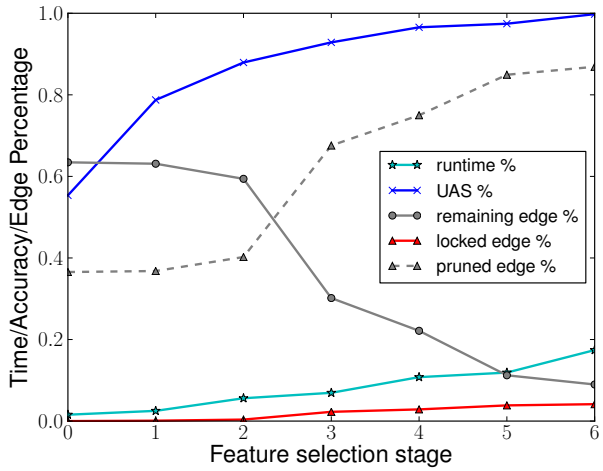


Figure 4.4: System dynamics on English PTB section 23. Time and accuracy are relative to those of the baseline model using full features. Red (locked), gray (undecided), dashed gray (pruned) lines correspond to edges shown in Figure 4.2.

From the table we notice that our first-order model's performance is comparable or superior to the vine pruning model, both in terms of speedup and accuracy. In some cases, the model with fewer features even achieves higher accuracy than the model with full features. The second-order model, however, does not work as well. In our experiments, the second-order model is more sensitive to false negatives, i.e. pruning of gold edges, due to larger error propagation than the first-order model. Therefore, to maintain parsing accuracy, the policy must make high-precision pruning decisions and becomes conservative. We could mitigate this by training the original parsing feature weights in conjunction with our policy feature weights. In addition, there is larger overhead during when checking non-projective edges and cycles.

We demonstrate the dynamics of our system in Figure 4.4 on PTB section 23. We show how the runtime, accuracy, number of locked edges and undecided edges

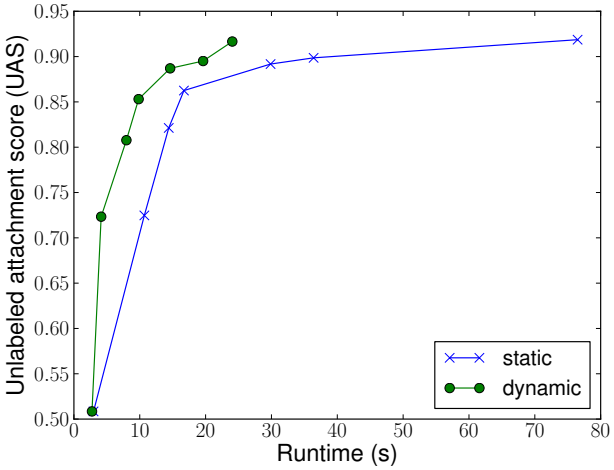


Figure 4.5: Pareto curves for the dynamic and static approaches on English PTB section 23.

change over the iterations in our first-order dynamic projective parsing. From iterations 1 to 6, we obtain parsing results from the non-projective parser; in iteration 7, we run the projective parser. The plot shows relative numbers (percentage) to the baseline model with full features. The number of remaining edges drops quickly after the second iteration. From Figure 4.3, however, we notice that even with the first feature group which only contains one feature template, the non-projective parser can almost achieve 50% accuracy. Thus, ideally, our policy should have locked that many edges after the first iteration. The learned policy does not imitate the expert perfectly, either because our policy features are not discriminative enough, or because a linear classifier is not powerful enough for this task.

Finally, to show the advantage of making dynamic decisions that consider the interaction among edges on the given input sentence, we compare our results with a static feature selection approach on PTB section 23. The static algorithm does no pruning except by the length dictionary at the start. In each iteration, instead of

running a fast parser and making decisions online, it simply adds the next group of feature templates to all edges. By forcing both algorithms to stop after each stage, we get the Pareto curves shown in Figure 4.5. For a given level of high accuracy, our dynamic approach (black) is much faster than its static counterpart (blue).

4.4 Related Work

In dependency parsing, the problem of feature computation is less studied. Instead, previous work has made much progress on the complementary problem: speeding up *decoding* by pruning the search space of tree structures. In Roark and Hollingshead (2008) and the follow-up work by Bodenstab (2012), pruning decisions are made locally for each cell in the CKY algorithm as a preprocessing step. At test time, the parser works on a fixed pruned graph. In the vine pruning approach (Rush and Petrov, 2012), edge pruning is done via a coarse-to-fine projective parsing cascade, using the framework of structured prediction cascades (Weiss and Taskar, 2010). These approaches do not directly tackle the feature selection problem. However, it is important to note that the feature selection approach and the graph pruning approach are not exclusive. During feature selection, we lock and prune edges, which reduces work of the decoder. On the other hand, pruned edges do not require further feature computation, which also reduces the scoring time. In addition, the two approaches both have limitations due to decision overhead. In feature selection, we have to reduce the number of decisions for edges and features. In graph pruning, the pruning step must itself compute high-dimensional features

just to decide which edges to prune, thus pruning has to rely on simpler features for efficiency (Rush and Petrov, 2012).

From the machine learning perspective, much work has focused on adaptive feature selection for supervised learning (See Section 3.5), however, very few is applicable to structured prediction. The most related work is Weiss et al. (2013), where models are arranged in order of increasing complexity (in terms of the feature used), and a selector is learned to decide whether to use a more complex model. The difference from our work is that they make the same decision for all substructures, while we consider each substructure—at least the important ones—separately.

4.5 Conclusion

In this chapter, we extended the dynamic feature selection framework proposed in the last chapter to structured prediction. We proposed a dynamic feature selection algorithm for graph-based dependency parsing, which successfully reduces feature computation time and avoids decision overhead. We evaluated our method across 7 languages. Our dynamic parser achieved accuracies comparable or even superior to parsers using a full set of features, while computing fewer than 30% of the feature templates.

This chapter concludes the first part of methods focusing on sequential acquisition problems. We have demonstrated the effectiveness of dynamically acquiring information only when needed on two applications. Starting from the next chapter, we will focus on problems with sequentially revealed input.

Chapter 5: Simultaneous Interpretation

This chapter describes joint work with Alvin Grissom II, John Morgan, Jordan Boyd-Graber and Hal Daumé III (He et al., 2015; Grissom II et al., 2014) in EMNLP 2014 and 2015.

To this point, we have seen that a cost-accuracy trade-off can be achieved by selecting *features* adaptively. From this chapter on, we start a new type of sequential problem where the *input* is revealed incrementally. In Chapter 1 we introduced two sequential revealing problems in NLP: simultaneous interpretation and quiz bowl (sequential question answering). The focus of this chapter is simultaneous interpretation, a less studied area in machine translation. In the rest of this chapter, we assume that speech input/output has been recognized or transcribed into text and we work with textual data only. Therefore, we will use the two terms—simultaneous interpretation and simultaneous translation—interchangeably.

Similar to the dynamic feature selection setting, we assume that we are given a pre-trained translator, and we need to adapt it to changing input and achieve a trade-off between cost and prediction quality. The difference here is that we cannot select which part of the input we would like to see, instead, we are given parts of the input in a fixed order. In simultaneous translation, the given input is a sequence of

words coming in one by one. The cost in sequential revealing problems is the time spent on waiting for more input. Therefore, our goal is to translate (speak) as soon as possible once words are input (uttered).

We first show how to learn a translation policy in a framework similar to that introduced in the previous two chapters. We then discuss challenges for simultaneous translation when we are not able to select the information most needed, e.g., the verb required for translation comes late in the input. Finally, we present two linguistically inspired methods to alleviate problems due to divergent word orders of the source language (input) and the target language (output).

5.1 Simultaneous Interpretation

Unlike translating a book, interpretation happens during real-time conversation or speech. To make sure information flows smoothly between the speaker and the listener, the interpreter must translate under a stringent time constraint. There is no time for weighing the appropriateness of similar words or looking up an unfamiliar expression. There are two major modes of interpretation: consecutive and simultaneous. In consecutive interpretation, the speaker stops after finishing a complete thought and waits for the interpreter to translate. This is similar to the normal translation setting, where complete sentences are seen before translation starts. In simultaneous interpretation, the interpreter translates *while* the speaker is talking. One of the first noted uses of simultaneous interpretation was the Nuremberg trials (Gaiba, 1998) after the Second World War, where a trade-off between translation

accuracy and latency was required to achieve "fair and expeditious trials". Since the speaker does not reserve time for translation, the interpreter must decide when to translate and when to wait for more information. In the rest of the section, we formally define the problem of simultaneous machine translation and our learning objective.

5.1.1 Problem Formulation

In batch machine translation (MT), we are given an input sentence x in the source language, and the system needs to output its translation y in the target language. We are interested in simultaneous machine translation, where the input comes in word by word.[1] Therefore, our input is a sequence of words $x_1, \ldots, x_T$, where x_t is the input word at time t and T is the length of the input. Correspondingly, the system generates partial translations $y_1, \ldots, y_T$ at each time step. The output y_t can be empty, in which case the system decides to wait for more input words. Next, we describe evaluation metrics for batch MT and simultaneous MT.

5.1.2 Objective and Evaluation Metric

Good simultaneous translations must optimize two objectives that are often at odds: producing high-quality translations and producing them expeditiously. All else being equal, maximizing either goal in isolation is trivial: for maximally accurate translation, use a batch translator and wait until the sentence is complete, then

[1] In this chapter, we deal with textual data on sentence level instead of continuous speech as in real interpretation. We simulate speaker's utterance by revealing one word of the source sentence at a time.

translate it all at once; for maximally expeditious translation, create glossed translations as words appear. The former is essentially consecutive interpretation, and we refer to it as *batch* translation. The latter is translated in the source sentence's word order, which can result in awkward translations of distant language pairs; we refer to it as *monotone* translation (Tillmann et al., 1997; Pytlik and Yarowsky, 2006). In the ideal case, we can imagine a *psychic* translator that maximizes both accuracy and speed: it predicts what the speaker is going to say and translate the entire (imagined) sentence as soon as one word is uttered. Obviously, this system is unrealistic, however, it helps us to identify a spectrum of system performance. We propose an evaluation metric for simultaneous MT that considers both translation latency and accuracy. Our metric is based on the BLEU metric for batch MT.

BLEU BLEU (Papineni et al., 2002b) is a widely used metric for measuring performance of a MT system, which has been shown to have high correlation with human judgments of quality (Papineni et al., 2002b; Papineni et al., 2002a). Given a set of reference translations $\mathcal{R}$ and a set of candidate translations $\mathcal{Y}$ (generated by a MT system), BLEU measures the similarity between these two translations by computing the average n-gram precision for paired $y \in \mathcal{Y}$ and $r \in \mathcal{R}$ (i.e., the percentage of n-grams in y that also appear in r). Formally, we define the n-gram precision as

$$p_n(\mathcal{Y}, \mathcal{R}) = \frac{\sum_{y \in \mathcal{Y}} \sum_{x \in f_n(y)} \mathbf{1}_{x \in f_n(r)}}{\sum_{y \in \mathcal{Y}} \sum_{x \in f_n(y)} 1} \tag{5.1}$$

where **1** is the indicator function and f_n denotes the n-gram extractor; for example, f_1 returns all words in a sentence and f_2 returns all bigrams.

If we only consider precision, it is possible for a system to get 100% score by simply generating one word in each reference translation. Therefore, BLEU also includes a brevity penalty (BP). Let m be the total length of all reference translations and n be the length of corresponding candidate translations, we have

$$\text{BP} = \begin{cases} 1 & n > m \\ e^{1-m/n} & \text{otherwise} \end{cases} \tag{5.2}$$

Finally, the BLEU score is defined as

$$\text{BLEU}(\mathcal{Y}, \mathcal{R}) = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} p_n(\mathcal{Y}, \mathcal{R})\right), \tag{5.3}$$

where N is usually set to 4.

The above BLEU score is defined on the entire corpus. However, since we will be learning policy on an instance basis, we need a sentence-level BLEU. The naive way to compute BLEU score for a sentence is to simple treat it as a corpus having only one sentence.

Latency-BLEU We now describe our metric, latency-BLEU (LBLEU) for simultaneous MT. Instead of a single candidate translation in batch MT, we have a sequence of partial translations $y_1, \ldots, y_T$ for each input sentence. We consider these partial translations in a two-dimensional space, with time (quantized by the number of

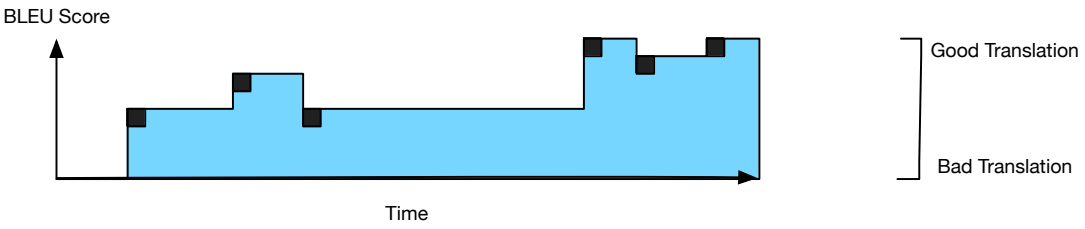


Figure 5.1: Latency-BLEU: integral of BLEU score over time.

source words seen) on the x-axis and the BLEU score on the y-axis. At each point in time, the system may generate a partial translation or nothing (wait). If y_t is empty, we use the BLEU score of the last partial translation as the score at time t. The visualization of LBLEU is shown in Figure 5.1, as the integral area of BLEU over time. A good system will be high and to the left, optimizing the area under the curve: the psychic system would produce points as high as possible immediately.

Formally, let $y_{1:t}$ be a shorthand for $\{y_1, \ldots, y_T\}$; we define LBLEU on a sentence basis:

$$\text{LBLEU}(y_{1:T}, r) = \frac{1}{T} \sum_t \text{BLEU}(y_t, r) + T \cdot \text{BLEU}(y_T, r). \tag{5.4}$$

The score of a simultaneous translation is the sum of the scores of all individual segments that contribute to the overall translation. We multiply the final BLEU score by T to ensure good final translations in learned systems.[2] This effectively gives equal weight to intermediate translations and the final translation.

Now that we have defined our objective, in the following section we describe what kind of decisions are needed to achieve such a trade-off between latency and

[2]One could replace T with a parameter, β, to bias towards different kinds of simultaneous translations. As $\beta \to \infty$, we recover batch translation.

accuracy.

5.2 Decision Process for Interpretation

A simultaneous MT system needs two modules: a translator that produces translations for input (partial) sentences, and a decision-maker that decides when to translate and when to wait for more information. We pretrain a translator using an off-the-shelf batch MT system and use it as a callable function to provide us translations of any input text. We learn a decision-maker in the MDP framework using imitation learning. This section focus on the formulation of the MDP (Section 2.1) for simultaneous translation.

In the MDP context, our state contains words seen so far $x_{1:t}$, and their translations $y_{1:t}$. The action set includes *wait* and *commit*. Waiting is the simplest action. It produces no output and allows the system to receive more input, biding its time, so that when it does choose to translate, the translation is based on more information. Committing sends received words to the translator and produces a translation. We can recreate a batch translation system by a sequence of wait actions until all input is observed, followed by a commit action to generate the complete translation. Similarly, we can recreate the monotone translator by committing at every time step. The reward in the final state is LBLEU score defined in the last section, and the optimal action sequence that maximizes LBLEU can be found by beam search.

Combining Partial Translations While we assume an underlying batch translator, we do not assume it can automatically combine a sequence of partial trans-

lations. The straightforward way is to send consecutive segments of text to the translator, translate them separately and concatenate the output segment translations. However, this way the translator does not have the previous context of each segment, and the translation quality may degrade. A slightly more complex way is to send all words seen so far (including those already translated) to the translator. To avoid possible conflict translation with previously committed translations, we only append translation of new words to the current output.[3]

We have described all essential components required to run an imitation learning algorithm. However, we have ignored an important problem that may limit the performance of our system: divergent word orders between the source and the target language. We detail this problem in the next section.

5.3 Challenge: Delayed Information

The translation process described above is limited by unavoidable word reordering between languages with drastically different word orders. Performing real-time translation is especially difficult when information that comes early in the target language comes late in the source language. A common example is when translating from a verb-final (SOV) language (e.g., German or Japanese) to a verb-medial (SVO) language, (e.g., English). For instance, in the example in Figure 5.2, the main verb of the sentence (in **bold**) appears at the end of the German sentence. However, the target English sentence requires a verb immediately after the subject

[3]We which part of the translation is for new input words since the output has alignment information, assuming a phrase-based MT system.

ich	bin	mit	dem	Zug	nach	Ulm	**gefahren**
I	am	with	the	train	to	Ulm	**traveled**
I		*(. . . . . waiting. . . . .)*					**traveled** by train to Ulm

Figure 5.2: An example of translating from a verb-final language to English. The verb, in **bold**, appears at the end of the sentence, preventing coherent translations until the final source word is revealed.

"I". In this case, a dynamic translator would behave the same as a batch translator at best, since it is forced to wait until the end of the sentence for the final verb. Waiting for such missing constituents required early in the target sentence can cause significant delay.

To address the above problem of divergent word orders, we propose two approaches. Both improves the speed at a little cost of translation quality. Although the two approaches focus on translating from SOV languages to SVO languages, some techniques apply to other language pairs as well.

The first approach is to predict upcoming words, especially the required but late syntactic constituent such as the verb. In reality, predicting future content is also an important skill of human interpreters (Hönig, 1997; Camayd-Freixas, 2011). They usually talk with the speaker or review the speech outline beforehand, so as to have a better idea of what the speaker is going to say next during interpretation. If the prediction is accurate, the system can start to translate before actually sees the syntactic component in need. However, our prediction may not always be reasonable, thus we need to decide when to trust the prediction and when to ignore it. In Section 5.4 we show how to incorporate future content prediction as another action and make decisions about it.

The second approach is to reorder the target sentence to postpone the need

for late constituents. Thanks to the flexibility of natural languages, sometimes we can arrange the words in a different way such that the unseen part is not required to produce a grammatical partial translation. Essentially, we want to paraphrase the target sentence so that it has a word order similar to that of the source sentence. While this is possible by adding "paraphrasing" actions to the search space, we will end up with a more complex action set and need to edit possibly inaccurate translations given by the batch translator. To avoid the complication, we modify the batch translator instead. In Section 5.5 we describe rewriting rules to make the target translation more monotone.

5.4 Predicting Future Content

In this section, we present a prediction method for alleviating the problem of divergent word orders. Our goal is to predict upcoming words (especially the late constituent in need) in the source sentence and use it as extra information for translating the current partial input. To see the advantage of predicting future words, we show an example in Figure 5.3. The reward is shown in dark blue for different systems. The monotone system translates anyway without the necessary constituent, and the batch system has to wait until the end for the verb. By correctly predicting the verb "gegangen" (to go), we achieve a better overall translation more quickly. In the following, we first describe how this can be combined with our current translation system, then show how to build such predictors.

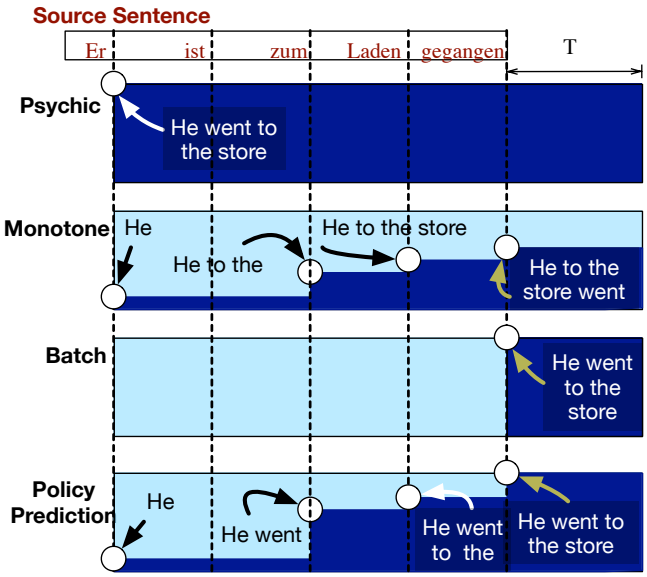


Figure 5.3: Comparison of LBLEU for an impossible psychic system, a traditional batch system, a monotone (German word order) system, and our prediction-based system.

Richer State and Action Space We incorporate prediction as an action in our MDP framework. Specifically, we add two actions: *next-word* and *verb* The next-word action takes a prediction of the next source word and produces an updated translation based on that prediction, i.e. appending the predicted word to the words seen so far and translating them as a single sentence. The verb action predicts the source sentence's final verb (since it is a verb-final language). The system uses the predicted verb in the same way as the next word by appending it to the current prefix.

With the two additional action, our state space becomes richer too. Besides the words and translations so far, we also include prediction of the next word and the final verb. Specifically, our state features are as follows.

- **Input.** We use a bag-of-words representation of the words seen so far. To

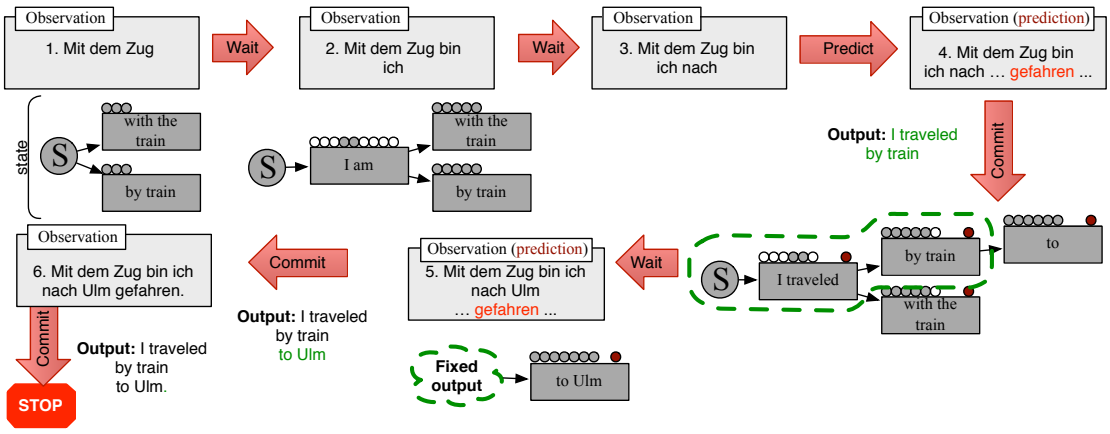


Figure 5.4: A simultaneous translation from source (German) to target (English). The agent chooses to wait until after (3). At this point, it is sufficiently confident to predict the final verb of the sentence (4). Given this additional information, it can now begin translating the sentence into English, constraining future translations (5). As the rest of the sentence is revealed, the system can translate the remainder of the sentence.

encode time/position of the state, we include the length of the current input as well as the most recent word and bigram.

- **Translation.** We use a bag-of-words representation of the target translation so far. We also have meta-features including the score of the current translation, and the difference between the current and previous translation scores.

- **Prediction.** We include the predicted verb and next word as well as their probabilities (details of the predictor are described later in this section).

We show the whole simultaneous translation process in Figure 5.4, using an example translation from German to English. The gray boxes represent the underlying batch translator, and the red arrows represent the decision-maker and its actions. Once the system commits, the new output is shown in green (the previous

output is fixed once committed and is shown in black). In the example, for the first few source words, the translator lacks the confidence to produce any output due to insufficient information at the state. However, after State 3, the state shows high confidence in the predicted verb "gefahren". Combined with previous German input it has observed, the system is sufficiently confident to act on that prediction to produce English translation.

Incremental Word Predictor The prediction of the next word in the source language sentence is modeled with a left-to-right language model. This is analogous to how a human translator might use his own "language model" to guess upcoming words to gain some speed by completing, for example, collocations before they are uttered. We use a simple bigram language model for next-word prediction (Heafield et al., 2013).

For verb prediction, we use a generative model that combines the prior probability of a particular verb v, $p(v)$, with the likelihood of the source context at time t given that verb (namely, $p(x_{1:t}|v)$), as estimated by a smoothed Kneser-Ney language model (Kneser and Ney, 1995). The prior probability $p(v)$ is estimated by simple relative frequency estimation. The context, $x_{1:t}$, consists of all words observed. We model $p(x_{1:t}|v)$ with verb-specific n-gram language models. The predicted verb at time t is then:

$$\hat{v}^{(t)} = \arg\max_{v} p(v) \prod_{i=1}^{t} p(x_i|v, x_{i-n+1:i-1}) \tag{5.5}$$

where $x_{i-n+1:i-1}$ is the $n-1$-gram context. To narrow the search space, we consider only the 100 most frequent final verbs, where a "final verb" is defined as the sequence

of verbs at the end of a sentence and particles as detected by a German POS tagger.[4]

5.4.1 Experiments

Setup We focus on translating from German (SOV) to English (SVO). We use data from the German-English Parallel "de-news" corpus of radio broadcast news (Koehn, 2000), which we lower-cased and stripped of punctuation. A total of 48,601 sentence pairs are randomly selected for building our system. Of these, we use 70% (34,528 pairs) for training word alignments.

We used 1 million words of news text from the Leipzig Wortschatz (Quasthoff et al., 2006) German corpus to train 5-gram language models to predict a verb from the 100 most frequent verbs. For training the translation policy, we restrict ourselves to sentences that end with one of the 100 most frequent verbs. This results in a data set of 4401 training sentences and 1832 test sentences from the de-news data. We did this to narrow the search space (from thousands of possible but mostly very infrequent, verbs).

For next-word prediction, we use the 18,345 most frequent German bigrams from the training set to provide a set of candidates in a language model trained on the same set. We use frequent bigrams to reduce the computational cost of finding the completion probability of the next word.

In Figure 5.5, we show performance of the optimal policy vs. the learned policy, as well as the two baseline policies: the batch policy and the monotone

[4]This has the obvious disadvantage of ignoring morphology and occasionally creating duplicates of common verbs that have may be associated with multiple particles; nevertheless, it provides a straightforward verb to predict.

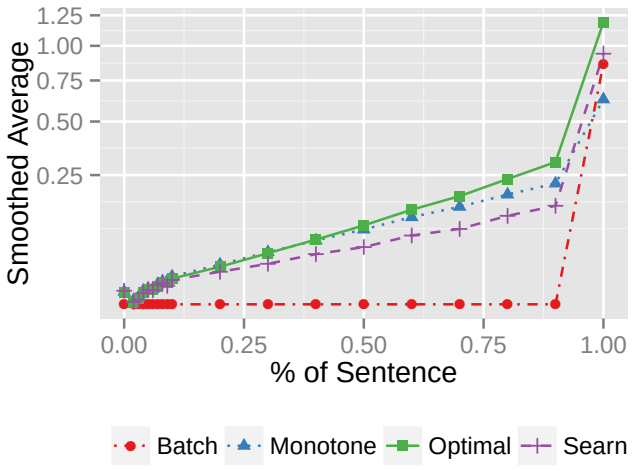


Figure 5.5: The final reward of policies on German data. Our policy outperforms all baselines by the end of the sentence.

policy. The x-axis is the percentage of the source sentence seen by the model, and the y-axis is a smoothed average of the reward as a function of the percentage of the sentence revealed. The monotone policy's performance is close to the optimal policy for the first half of the sentence, as German and English have similar word order, though they diverge toward the end. Our learned policy outperforms the monotone policy toward the end and of course outperforms the batch policy throughout the sentence. Figure 5.6 shows counts of actions taken by each policy. The batch policy always commits at the end. The monotone policy commits at each position. Our learned policy has an action distribution similar to that of the optimal policy but is slightly more cautious.

Discussion The word prediction method we use here is very simple. In fact, the low percentage of the verb and next-word action is because that these predictions are often inaccurate. A more realistic predictor would take context into account when

Figure 5.6: Histogram of actions taken by the policies.

it is available. For example, for TED talks we usually have the title and abstract of the talk beforehand, which can be used as additional features for prediction. Since the prediction approach can suffer from low prediction accuracy due to limited knowledge of the context, we propose an orthogonal paraphrasing approach in the next section.

5.5 Word Reordering

We now turn to a different approach to handling divergent word orders during simultaneous translation. We have seen from Figure 5.2 that the batch translator does not work well for simultaneous translation as it expects complete information. Therefore, we modify training references of the batch translator such that it learns to produce more monotone translation using only the available information. The

Source:	我々は	政府の	構造	や	組織を	変更	すべき	だ
	We-TOP	**government**-GEN	structure	and	composition-ACC	**change**	should	COP
Batch:	我々は ‖	政府の構造や組織を変更すべき だ						
	We	should **change** the structure and composition **of the government**						
Monotone:	我々は ‖	政府の ‖	構造や組織を	‖	変更すべきだ			
	the government's	structure and composition	should **be changed** by us					

Figure 5.7: Divergent word order between language pairs can cause long delays in simultaneous translation: Segments (||) mark the portions of the sentence that can be translated together. (Case markers: topic (TOP), genitive (GEN), accusative (ACC), copula (COP).)

motivation is that a sentence can often be paraphrased into a different word order with the meaning kept unchanged. We apply syntactic transformations to target references to make their word orders closer to the source language word order.

We show an example of Japanese-English translation in Figure 5.7. Consider the batch translation: in English, the verb "change" comes immediately after the subject "We", whereas in Japanese it comes at the end of the sentence; therefore, to produce an intelligible English sentence, we must translate the object after the final verb is observed, resulting in one large and painfully delayed segment. However, in the monotone translation, by passivizing the English sentence, we can cache the subject and begin translating before observing the final verb. Furthermore, by using the English possessive, we mimic the order of the Japanese genitive construction. These transformations enable us to divide the input into shorter segments, thus reducing translation delay.

We propose to *rewrite the reference translation* in a way that uses the original lexicon, obeys standard grammar rules of the target language, preserves the original

semantics, and yields more monotonic translations. As it is hard to define rewriting as a small set of actions, we instead learn to rewrite when training the underlying translator. We first rewrite reference translations (as a preprocessing step) in an order similar to the source language word order, then train the MT system with the rewritten references so that it learns how to produce low-latency translations from the data. A data-driven approach to learning these rewriting rules is hampered by the dearth of parallel data: we have few examples of text that have been both interpreted and translated. Therefore, we design syntactic transformation rules based on linguistic analysis of the source and the target languages. We apply these rules to parsed text and decide whether to accept the rewritten sentence based on the amount of delay reduction. In this section, we focus on Japanese to English translation, because (i) Japanese and English have significantly different word orders; and consequently, (ii) the syntactic constituents required earlier by an English sentence often come late in the corresponding Japanese sentence. Next, we describe our rewriting rules and how they are applied to the original references.

5.5.1 Transformation Rules

We design a variety of syntactic transformation rules for Japanese-English translation motivated by their structural differences. Our rules cover verb, noun, and clause reordering. While we specifically focus on Japanese to English, many rules are broadly applicable to SOV to SVO languages.

5.5.1.1 Verb Phrases

The most significant difference between Japanese and English is that the head of a verb phrase comes at the end of Japanese sentences. In English, it occupies one of the initial positions. We now introduce rules that can postpone a head verb.

Passivization and Activization In Japanese, the standard structure of a sentence is $\texttt{NP}_1$ $\texttt{NP}_2$ `verb`, where case markers following the verb indicate the voice of the sentence. However, in English, we have $\texttt{NP}_1$ `verb` $\texttt{NP}_2$, where the form of the verb indicates its voice. Changing the voice is particularly useful when $\texttt{NP}_2$ (object in an active-voice sentence and subject in a passive-voice sentence) is long. By reversing positions of `verb` and $\texttt{NP}_2$, we are not held back by the upcoming verb and can start to translate $\texttt{NP}_2$ immediately. Figure 5.7 shows an example in which passive voice can help make the target and source word orders more compatible, but it is not the case that passivizing every sentence would be a good idea; sometimes making a passive sentence active makes the word orders more compatible if the objects are relatively short:

O: The talk was **denied** by the boycott group spokesman.

R: The boycott group spokesman **denied** the talk.

Quotative Verbs *Quotative* verbs are verbs that, syntactically and semantically, resemble *said* and often start an independent clause. Such verbs are frequent, especially in news, and can be moved to the end of a sentence:

O: **They announced** that the president will restructure the division.

R: The president will restructure the division, **they announced**.

In addition to quotative verbs, candidates typically include factive (e.g., *know*, *realize*, *observe*), factive-like (e.g., *announce*, *determine*), belief (e.g., *believe*, *think*, *suspect*), and anti-factive (e.g., *doubt*, *deny*) verbs. When these verbs are followed by a clause (`S` or `SBAR`), we move the verb and its subject to the end of the clause.

While some exploratory work automatically extracts factive verbs, to our knowledge, an exhaustive list does not exist. To obtain a list with reasonable coverage, we exploit the fact that Japanese has an unambiguous quotative particle, *to*, that precedes such verbs.[5] We identify all of the verbs in the Kyoto corpus (Neubig, 2011) marked by the quotative particle and translate them into English. We then use these as our quotative verbs.[6]

5.5.1.2 Noun Phrases

Another difference between Japanese and English lies in the order of adjectives and the nouns they modify. We identify two situations where we can take advantage of the flexibility of English grammar to favor sentence structures consistent with positions of nouns in Japanese.

Genitive Reordering In Japanese, genitive constructions always occur in the form of X *no* Y, where Y belongs to X. In English, however, the order may be

[5] We use a morphological analyzer to distinguish between the conjunction and quotative particles. Examples of words marked by this particle include 見られる (*expect*), 言う (*say*), 思われる (*seem*), する (*assume*), 信じる (*believe*) and so on.

[6] We also include the phrase *It looks like*.

reversed through the *of* construction. Therefore, we transform constructions $\texttt{NP}_1$ of $\texttt{NP}_2$ to possessives using the apostrophe-s, $\texttt{NP}_2$'(s) $\texttt{NP}_1$ (Figure 5.7). We use simple heuristics to decide if such a transformation is valid. For example, when X / Y contains proper nouns (e.g., *the City of New York*), numbers (e.g., *seven pounds of sugar*), or pronouns (e.g., *most of them*), changing them to the possessive case is not legal.

***that* Clause** In English, clauses are often modified through a pleonastic pronoun. E.g., *It is* ADJP *to/that* SBAR/S. In Japanese, however, the subject (clause) is usually put at the beginning. To be consistent with the Japanese word order, we move the modified clause to the start of the sentence: *To* S/SBAR *is* ADJP. The rewritten English sentence is still grammatical, although its structure is less frequent in common English usage. For example,

O: **It is important** to remain watchful.

R: To remain watchful **is important**.

5.5.1.3 Conjunction Clause

In Japanese, clausal conjunctions are often marked at the end of the initial clause of a compound sentence. In English, however, the order of clauses is more flexible. Therefore, we can reduce delay by reordering the English clauses to mirror how they typically appear in Japanese. Below we describe rules reversing the order of clauses connected by these conjunctions:

- Clausal conjunctions: *because (of)*, *in order to*
- Contrastive conjunctions: *despite*, *even though*, *although*
- Conditionals: *(even) if*, *as a result (of)*
- Misc: *according to*

In standard Japanese, such conjunctions include *no de*, *kara*, *de mo* and so on. The sentence often appears in the form of $\texttt{S}_2$ `conj`, $\texttt{S}_1$. In English, however, two common constructions are

$\texttt{S}_1$ `conj` $\texttt{S}_2$: We should march **because** winter is coming.

`conj` $\texttt{S}_2$, $\texttt{S}_1$: **Because** winter is coming, we should march.

To follow the Japanese clause order, we adapt the above two constructions to

$\texttt{S}_2$, `conj'` $\texttt{S}_1$: Winter is coming, **because of this**, we should march.

Here `conj'` represents the original conjunction word appended with simple pronouns/phrases to refer to $\texttt{S}_2$. For example, *because* $\rightarrow$ *because of this*, *even if* $\rightarrow$ *even if this is the case*.

5.5.2 Sentence Rewriting Process

We now turn our attention to the implementation of the syntactic transformation rules described above. Applying a transformation consists of three steps:

1. **Detection:** Identify nodes in the parse tree for which the transformation is applicable;

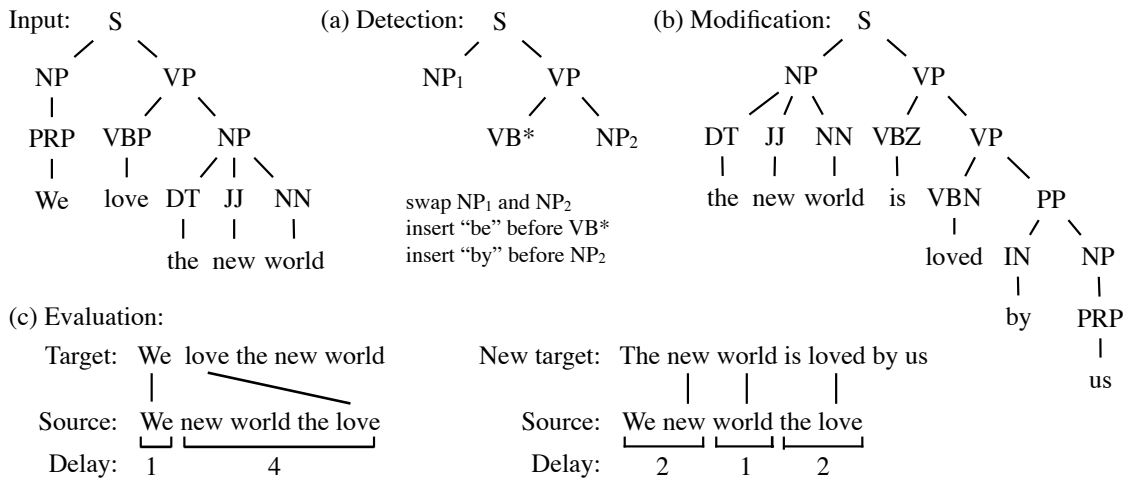


Figure 5.8: An example of applying the passivization rule to create a translation reference that is more monotonic.

2. **Modification:** Transform nodes and labels;

3. **Evaluation:** Compute delay reduction, and decide whether to accept the rewritten sentence.

Figure 5.8 illustrates the process using passivization as an example. In the detection step, we find the subtree that satisfies the condition of applying a rule. In this case, we look for an `S` node whose children include an `NP` (denoted by $\texttt{NP}_1$), the subject, and a `VP` to its right, such that the `VP` node has a leaf `VB`*, the main verb,[7] followed by another `NP` (denoted by $\texttt{NP}_2$), the object. We allow the parent nodes (`S` and `VP`) to have additional children besides the matched ones. They are not affected by the transformation. In the modification step, we swap the subject node and object node; we add the verb *be* in its correct form by checking the tense of the verb and the form of $\texttt{NP}_2$;[8]and we add the preposition *by* before the subject. The process is

[7]The main verb excludes *be* and *have* when it indicates tense (e.g., *have done*).

[8]We use the Nodebox linguistic library (https://www.nodebox.net/code) to detect and modify word forms.

executed recursively throughout the parse tree.

Although our rules are designed to minimize long range reordering, there are exceptions.[9] Thus applying a rule does not always reduce delay. In the evaluation step, we compare translation delay before and after applying the rule. We accept a rewritten sentence if its delay is reduced; otherwise, we revert to the input sentence. Since we do not segment sentences during rewriting, we must estimate the delay.

To estimate the delay, we use word alignments. Figure 5.8c shows the source Japanese sentence in its word-for-word English translation and alignments from the target words to the source words. The first English word, "We", is aligned to the first Japanese word; it can thus be treated as an independent segment and translated immediately. The second English word, "love", is aligned to the last Japanese word, which means the system cannot start to translate until four more Japanese words are revealed. Therefore, this alignment forms a segment with a delay of four words/seg. Alignments of the following words come before the source word aligned to "love"; hence, they are already translated in the previous segment and we do not double count their delay. In this example, the delay of the original sentence is 2.5 word/seg; after rewriting, it is reduced to 1.7 word/seg. Therefore, we accept the rewritten sentence. However, when the subject phrase is long and the object phrase is short, a swap may not reduce delay.

We can now formally define the **delay**. Let e_i be the ith target word in the input sentence x and a_i be the maximum index among indices of source words that

[9]For example, in clause transformation, the Japanese conjunction *moshi*, which is clause initial, may appear at the beginning of a sentence to emphasize conditionals, although its appearance is relatively rare.

e_i aligned to. We define the delay of e_i as $d_i = \max(0, a_i - \max_{j<i} a_j)$. The delay of x is then $\sum_{i=1}^{N} d_i/N$, where the sum is over all aligned words except punctuation and stopwords.

Given a set of rules, we need to decide which rules to apply and in what order. Fortunately, our rules have little interaction with each other, and the order of application has a negligible effect. We apply the rules, roughly, sequentially in order of complexity: if the output of current rule is not accepted, the sentence is reverted to the last accepted version.

5.5.3 Experiments

We evaluate our method on the Reuters Japanese-English corpus of news articles (Utiyama and Isahara, 2003). For training the MT system, we also include the EIJIRO dictionary entries and the accompanying example sentences.[10] The rewritten translation is generally slightly longer than the gold translation because our rewriting often involves inserting pronouns (e.g., "it", "this") for antecedents.

We use the `TreebankWordTokenizer` from NLTK (Bird et al., 2009) to tokenize English sentences and Kuromoji Japanese morphological analyzer[11] to tokenize Japanese sentences. Our phrase-based MT system is trained by Moses (Koehn et al., 2003) with standard parameters settings. We use GIZA++ (Och and Ney, 2003) for word alignment and k-best batch MIRA (Cherry and Foster, 2012) for tuning. The translation quality is evaluated by BLEU (Papineni et al., 2002b) and

[10] Available at http://eijiro.jp
[11] Available at http://www.atilika.org/

	verb	voice	noun	conj.
Applicable %	39.9	50.0	26.4	4.8
Accepted %	22.5	24.0	51.2	38.4

Table 5.1: Percentage of sentences that each rule category can be applied to (Applicable) and the percentage of sentences for which the rule results in a more monotonic sentence (Accepted).

RIBES (Isozaki et al., 2010).[12] To obtain the parse trees for English sentences, we use the Stanford Parser (Klein and Manning, 2003) and the included English model.

5.5.3.1 Quality of Rewritten Translations

After applying the rewriting rules, Table 5.1 shows the percentage of sentences that are candidates and how many rewrites are accepted. The most generalizable rules are passivization and delaying quotative verbs. We rewrite 32.2% of sentences, reducing the delay from 9.9 words/seg to 6.3 words/seg per segment for rewritten sentences and from 7.8 words/seg to 6.7 words/seg overall.

We evaluate the quality of our rewritten sentences from two perspectives: grammaticality and preserved semantics. To examine how close the rewritten sentences are to standard English, we train a 5-gram language model using the English data from the Europarl corpus, consisting of 46 million words, and use it to compute perplexity. Rewriting references increase the perplexity under the language model only slightly: from 332.0 to 335.4. To ensure that rewrites leave meaning unchanged, we use the SEMAFOR semantic role labeler (Das et al., 2014) on the original and modified sentence; for each role-labeled token in the reference sen-

[12] In contrast to BLEU, RIBES is an order-sensitive metric commonly used for translation between Japanese and English.

tence, we examine its corresponding role in the rewritten sentence and calculate the average accuracy across all sentences. Even ignoring benign lexical changes—for example, "he" becoming "him" in a passivized sentence—95.5% of the words retain their semantic roles in the rewritten sentences.

Although our rules are conservative to minimize corruption, some errors are unavoidable due to propagation of parser errors. For example, the sentence *the London Stock Exchange closes at 1230 GMT today* is parsed as:[13]

(S (NP *the London Stock Exchange*)
(VP (VBZ *closes*)
(PP *at 1230*)
(NP *GMT today*)))

GMT today is separated from the PP as an NP and is mistaken as the object. The passive version is then *GMT today is closed at 1230 by the London Stock Exchange*. Such errors could be reduced by skipping nodes with low inside/outside scores given by the parser, or skipping low-frequency patterns. However, we leave this for future work.

5.5.3.2 Segmentation

At test time, we use right probability (Fujita et al., 2013, RP) to decide when to start translating a sentence. As we read in the source Japanese sentence, if the input segment matches an entry in the learned phrase table, we query the RP of the Japanese/English phrase pair. A higher RP indicates that the English translation of

[13]For simplicity we show the shallow parse only.

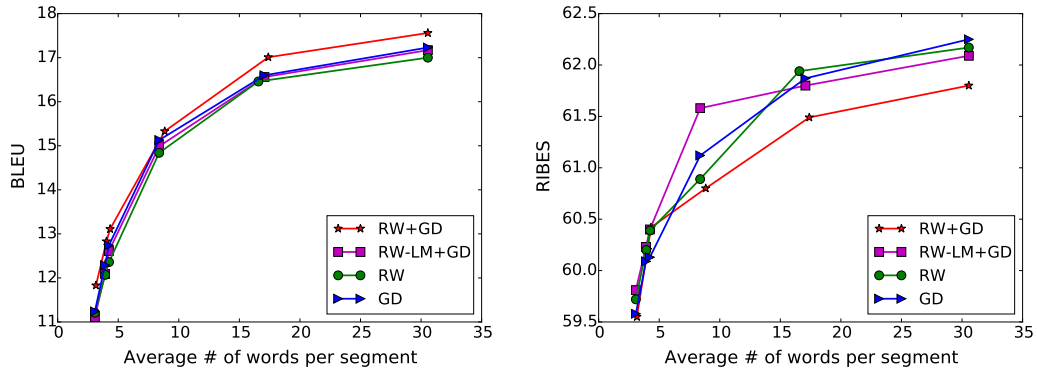
this Japanese phrase will likely be followed by the translation of the next Japanese phrase. In other words, translation of the two consecutive Japanese phrases is monotonic, thus, we can begin translating immediately. Following (Fujita et al., 2013), if the RP of the current phrase is lower than a fixed threshold, we cache the current phrase and wait for more words from the source sentence; otherwise, we translate all cached phrases. Finally, translations of segments are concatenated to form a complete translation of the input sentence.

5.5.3.3 Speed/Accuracy Trade-off

To show the effect of rewritten references, we compare the following MT systems:

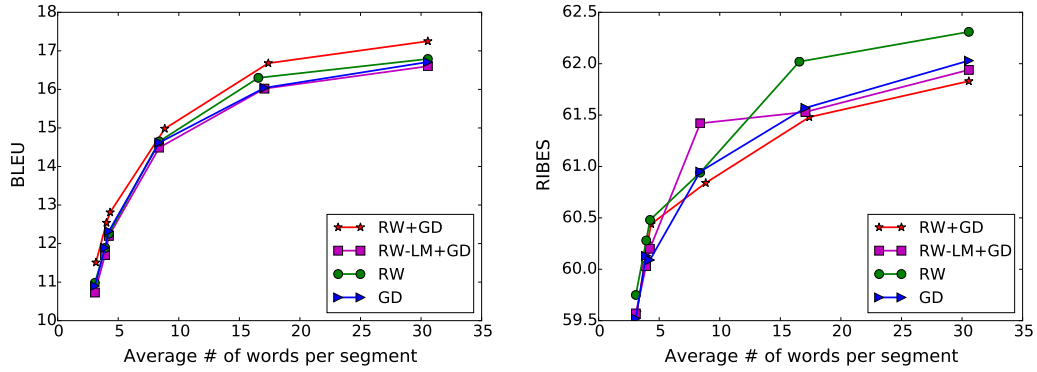
- **gd:** only gold reference translations;
- **rw:** only rewritten reference translations;
- **rw+gd:** both gold and the rewritten references; and
- **rw-lm+gd:** using gold reference translations but using the rewritten references for training the LM and for tuning.

For RW+GD and RW-LM+GD, we interpolate the language models of GD and RW. The interpolating weight is tuned with the rewritten sentences. For RW+GD, we combine the translation models (phrase tables and reordering tables) of RW and GD by fill-up combination (Bisazza et al., 2011), where all entries in the tables of RW are preserved and entries from the tables of GD are added if new.



(a) BLEU w.r.t. gold ref

(b) RIBES w.r.t. gold ref



(c) BLEU w.r.t. rewritten ref

(d) RIBES w.r.t. rewritten ref

Figure 5.9: Speed/accuracy trade-off curves: BLEU (left) / RIBES (right) versus translation delay (average number of words per segment).

Increasing the RP threshold increases interpretation delay but improves the quality of the translation. We set the RP threshold at 0.0, 0.2, 0.4, 0.8 and finally 1.0 (equivalent to batch translation). Figure 5.9 shows the BLEU/RIBES scores vs. the number of words per segment as we increase the threshold. Rewritten sentences alone do not significantly improve over the baseline. We suspect this is because the transformation rules sometimes generate ungrammatical sentences due to parsing errors, which impairs learning. However, combining RW and GD results in a better speed-accuracy trade-off: the RW+GD curve completely dominates other

curves in Figure 5.9(a), 5.9(c). Thus, using more monotone translations improves simultaneous machine translation, and because RW-LM+GD is about the same as GD, the major improvement likely comes from the translation model from rewritten sentences.

The right two plots recapitulate the evaluation with the RIBES metric. This result is less clear, as MT systems are optimized for BLEU and RIBES penalizes word reordering, making it difficult to compare systems that intentionally change word order. Nevertheless, RW is comparable to GD on gold references and superior to the baseline on rewritten references.

5.5.3.4 Effect on Verbs

Rewriting training data not only creates lower latency simultaneous translations, but it also improves *batch* translation. One reason is that SOV to SVO translation often drops the verb because of long range reordering. (We see this for Japanese here, but this is also true for German.) Similar word orders in the source and target results in less reordering and improves phrase-based MT (Collins et al., 2005; Xu et al., 2009). Table 5.2 shows the number of verbs in the translations of the test sentences produced by GD, RW, RW+GD, as well as the number in the gold reference translation. Both RW and RW+GD produce more verbs (a statistically significant result), although RW+GD captures the most verbs.

	Translation			
	GD	RW	RW+GD	Gold ref
# of verbs	1971	2050	**2224**	2731

Table 5.2: Number of verbs in the test set translation produced by different models and the gold reference translation. Boldface indicates the number is significantly larger than others (excluding the gold ref) according to two-sample t-tests with $p < 0.001$.

Ref	**he also said** that the real dangers for the euro lay in the potential for divergences in the domestic policy needs among the various participating nations of the single currency.
GD	**he** also for the euro, is a real danger to launch a single currency in many different countries and domestic policies on the need for the possibility of a difference.
RW	**he** also for the euro is a real danger to launch a single currency in many different countries and domestic policies to the needs of the possibility of a difference, **he said**.

Table 5.3: Example of translation produced by GD and RW.

5.5.3.5 Error Analysis

Table 5.3 compares translations by GD and RW. RW correctly puts the verb *said* at the end, while GD drops the final verb. However, RW still produces *he* at the beginning (also the first word in the Japanese source sentence). It is because our current segmentation strategy does not preserve words for *later* translation This can be addressed by adding a stack to reorder segments instead of monotonically translating all segments.

5.6 Related Work

Most previous work on simultaneous machine translation focuses on learning a segmentation strategy, essentially a decision model with two actions: wait and commit. The input is segmented into small chunks which are then translated independently. Most segmentation strategies are based on heuristics, such as pauses in speech (Fügen et al., 2007; Bangalore et al., 2009), comma prediction (Sridhar et al., 2013) and phrase reordering probability (Fujita et al., 2013). Learning-based methods have also been proposed. Oda et al. (2014) find segmentations that maximize the BLEU score of the final concatenated translation by dynamic programming. Compared to prior approaches, our work provides a nice decision-making framework that allows for richer actions other than wait and commit.

Smarter segmentation method can achieve a better speed-accuracy trade-off to some extent, however, their gain can still be restricted by natural word reordering between the source and the target sentences.

One approach is to predict the necessary component that has not been revealed. To our knowledge, the only attempt to specifically predict verbs or any late-occurring terms is Matsubara et al. (2000). They used pattern matching on what would today be considered a small data set to predict English verbs for Japanese to English simultaneous MT. More recently, Oda et al. (2015) predict syntactic constituents for a syntax-based MT system to process incremental input.

Another approach is to train with interpreter-generated references, so that the MT system can learn delay reduction strategies implicitly from the interpreters (Paulik

and Waibel, 2009; Paulik and Waibel, 2010; Shimizu et al., 2013). However, existing parallel simultaneous interpretation corpora (Shimizu et al., 2014; Matsubara et al., 2002; Bendazzoli and Sandrelli, 2005) are often small, and collecting new data is expensive due to the inherent costs of recording and transcribing speeches (Paulik and Waibel, 2010). In addition, due to the intense time pressure during interpretation, human interpretation has the disadvantage of simpler, less precise diction (Camayd-Freixas, 2011; Al-Khanji et al., 2000) compared to human translations done at the translator's leisure, allowing for more introspection and precise word choice. Our paraphrasing approach addresses the data scarcity problem and combines translators' lexical precision and interpreters' syntactic flexibility. In addition, it can be combined with any decision module.

There are many work in machine translation (Collins et al., 2005; Galley and Manning, 2008) on source-side reordering to reduce long-distance word reordering during decoding. However, in interpretation, we do not have control over the source sentence. Since reordering happens at the target side, we also need to make sure that the rewritten sentence is grammatical—a constraint not present in source-side reordering. The idea of rewriting as a preprocessing step is also related to text simplification for NLP systems. Chandrasekar et al. (1996) design rules to break long sentences into multiple shorter, simpler sentences before send them to an NLP system (e.g., a parser, a translator), since long and complicated sentences are challenging to process for both humans and NLP models.

5.7 Conclusion

In this chapter, we propose a sequential decision-making framework for simultaneous machine translation, a sequential revealing problem. The proposed framework is under the same paradigm as the sequential acquisition algorithms introduced in the previous two chapters. However, naively applying the framework may face difficulties when the needed information is not available. In simultaneous translation, this occurs when a syntactic constituent required by the target sentence comes late in the source sentence. We further proposed two complementary methods to address this issue. The prediction approach demonstrates the flexibility of our framework to handle new actions suitable to different applications. Besides this new learning framework for simultaneous MT, we also develop linguistically inspired paraphrasing rules to reduce translation delay.

Chapter 6: Opponent Modeling

This chapter describes joint work with Jordan Boyd-Graber, Kevin Kwok and Hal Daumé III (He et al., 2016) in ICML 2016.

Till now the sequential problems we have considered all have a single agent (the system itself) interacting with the environment, e.g., a selector acquiring features, a controller starting and pausing translation. In practice, however, a system may need to interact with other systems or human beings. In this chapter, we aim to answer the question: how can we take into consideration other agents who are actively affecting the environment during decision-making?

To answer this question, we focus on quiz bowl, a question answering game that involves two parties. It has similar characteristics to simultaneous translation: the input is revealed incrementally, and an early answer is preferred which calls for a speed-accuracy trade-off. More importantly, the multi-player nature of this game provides us a concrete setting to investigate policy learning with multiple agents. Given possibly unknown agents (aside from the system itself) in the environment, sometimes it can be hard to define an expert, as we will see later in this chapter. Instead, we take the reinforcement learning approach and show how the recent deep reinforcement learning technique enables us to incorporate behavior of other agents.

6.1 Motivation and Overview

An intelligent agent working in strategic settings (e.g., collaborative or competitive tasks) must predict the action of other agents and reason about their intentions. This is important because all active agents affect the state of the world. For example, a multi-player game AI can exploit suboptimal players if it can predict their bad moves; a negotiating agent can reach an agreement faster if it knows the other party's bottom line; a self-driving car must avoid accidents by predicting where cars and pedestrians are going. Two critical questions in opponent modeling are what variable(s) to model and how to use the predicted information. However, the answers depend much on the specific application, and most previous work (Billings et al., 1998a; Southey et al., 2005; Ganzfried and Sandholm, 2011) focuses exclusively on poker games which require substantial domain knowledge.

In this chapter, we aim to build a general opponent modeling framework for sequential decision-making, which enables the agent to exploit idiosyncrasies of various opponents. First, to account for the changing behavior, we must model uncertainty in the opponent's strategy instead of classifying it into one of the player stereotypes. Second, domain knowledge is often required when the prediction of opponents are separated from learning the dynamics of the world. Given the above two considerations, we model the opponent probabilistically and learn the policy and the opponent model jointly. Our models in this chapter are based on reinforcement learning, more specifically, Q-learning, rather than imitation learning in the previous chapters. We suggest the reader to review concepts defined in Section 2.1. In the next section, we

first introduce the game of quiz bowl and its multiagent setting.

6.2 Quiz Bowl: an Incremental Classification Game

A real-life setting where humans classify documents incrementally and compete with an opponent is quiz bowl, an academic competition between schools in English-speaking countries; hundreds of teams compete in dozens of tournaments each year (Jennings, 2006). Note the distinction between quiz bowl and Jeopardy, a recent application area (Ferrucci et al., 2010). While Jeopardy also uses signaling devices, these are only usable *after a question is completed* (interrupting Jeopardy's questions would make for bad television). Thus, Jeopardy is batch classification followed by a race—among those who know the answer—to punch a button first.

Two teams listen to the same question.[1] In this context, a question is a series of clues (features) referring to the same entity (for an example question, see Figure 6.1). We assume a fixed feature ordering for a test sequence (i.e., you cannot request specific features). Teams interrupt the question at any point by "buzzing in"; if the answer is correct, the team gets points and the next question is read. Otherwise, the team loses points and the other team can answer.

A successful quiz bowl player needs two things: a content model to predict answers given (partial) questions and a buzzing model to decide when to buzz in.

[1] Called a "starter" (UK) or "tossup" (US) in the lingo, as it often is followed by a "bonus" given to the team that answers the starter; here we only concern ourselves with tossups answerable by both teams.

After losing a race for the Senate, this politician edited the Omaha World-Herald. This man resigned ◇ from one of his posts when the President sent a letter to Germany protesting the Lusitania ◇ sinking, and ◇ he advocated ◇ coining ◇ silver at a 16 ◇ to 1 ◇◇ rate ◇ compared to ◇ gold. He was the ◇ three-time Democratic ◇ Party ◇◇◇ nominee for ◇ President ◇ but ◇◇◇ lost to McKinley twice ◇◇ and then Taft, although he served as Secretary of State ◇◇ under Woodrow Wilson, ◇ and he later argued ◇ against Clarence Darrow ◇ in the Scopes ◇◇ Monkey Trial. For ten points, name this ◇ man who famously declared that "we shall not be crucified on a Cross of ◇ Gold". ◇

Figure 6.1: Quiz bowl question on William Jennings Bryan, a late nineteenth century American politician; obscure clues are at the beginning while more accessible clues are at the end. Words (excluding stop words) are shaded based on the number of times the word triggered a buzz from any player who answered the question (darker means more buzzes; buzzes contribute to the shading of the previous five words). Diamonds (◇) indicate buzz positions of humans.

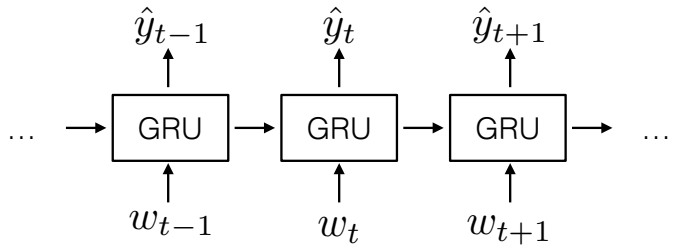


Figure 6.2: GRU for incremental text classification. Words are revealed one by one. w_t represents the t-th revealed word, and $\hat{y}_t$ represents the predicted answer at time t.

Content Model We model the question answering part as an incremental text-classification problem. Therefore, our content model is a classifier that takes in a (partial) question and outputs the answer. A human player would have a changing guess of the answer while the question is being revealed. To model a sequence of guesses, we use a recurrent neural network with gated recurrent units (GRU) (Cho et al., 2014) as our content model. It reads in the question sequentially and outputs a distribution over answers at each word given past information encoded in the hidden states. The model is shown in Figure 6.2.

Buzzing Model To test the depth of one's knowledge on a subject, the question usually starts with obscure information and reveals more and more obvious clues towards the end. For example, in Figure 6.1, more human buzzes (marked by diamonds) occur towards the end of the question. Therefore, the buzzing model faces a speed-accuracy trade-off: while buzzing later increases one's chance of answering correctly, it also increases the risk of losing the chance to answer. A safe strategy is to always buzz as soon as the content model is confident enough. A smarter strategy, however, is to adapt to different opponents: if the opponent is likely to buzz late on a question, wait for more clues; otherwise, buzz more aggressively.

Following the imitation approach as in previous chapters, we need an expert. If we take the safe strategy, the expert behavior is straightforward: always buzzing when the current prediction is correct. The agent may learn to buzz at different confidence levels that adapt to different states.[2] However, it cannot learn to adapt to different opponents. An expert policy considering the opponent behavior is less obvious. Therefore, we take a RL approach to learning a buzzing policy. The state includes words revealed and predictions from the content model, and the actions are *buzz* and *wait*. Upon buzzing, the content model outputs the most likely answer at the current position. An episode terminates when one player buzzes and answers the question correctly. A correct answer is worth 10 points and a wrong answer is -5. The complete payoff matrix is shown in Table 6.1.

Our model is built upon a basic RL algorithm, Q-learning. In the next section,

[2]For example, if an answer has a small number of training examples (questions), prediction of it tends to have lower confidence in general, thus the policy should learn to buzz at a lower confidence level for questions from the minority class.

	Computer	Human	Payoff
1	first and wrong	right	−15
2	—	first and correct	−10
3	first and wrong	wrong	−5
4	first and correct	—	+10
5	wrong	first and wrong	+5
6	right	first and wrong	+15

Table 6.1: Payoff matrix (from the computer's perspective) for when agents "buzz" during a question. To focus on incremental classification, we exclude instances where the human interrupts with an *incorrect* answer, as after an opponent eliminates themselves, the answering reduces to standard classification.

we review Q-learning and its recent variant based on deep neural networks.

6.3 Deep Q-Learning

Reinforcement learning is commonly used for solving Markov-decision processes (MDP). Unlike imitation learning, an expert is not available and the agent has to find the optimal action in each state by exploration. In Section 2.1, given the optimal Q-values of each action in a state, the optimal policy always chooses the action with the highest Q-value (Equation 2.7). Therefore, if we can find the optimal Q-values of each state, the decision problem is solved.

Q-learning (Watkins and Dayan, 1992; Sutton and Barto, 1998) is a popular model-free method for finding the optimal Q-values that does not require knowledge of $\mathcal{T}$. Given observed transitions (s, a, s', r) (also called an experience), the Q-values are updated recursively:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right].$$

Algorithm 4 Q-Learning

1: Initialize $Q(s, a)$ for all $a \in \mathcal{A}$, $s \in \mathcal{S}$
2: **for** $i = 1$ to N **do**
3: Initialize s randomly
4: **while** s is not terminal **do**
5: Choose a using an exploration policy derived from Q
6: Take action a, observe s', r
7: Update $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
8: $s \leftarrow s'$
9: **end while**
10: **end for**

The transition probabilities are thus modeled implicitly through stochastic updates. Given discrete state and actions (e.g., in a grid world), we can update $Q(s, a)$ for each state iteratively. We show the complete learning algorithm in Algorithm 4. During training, we follow an exploration policy (Line 5, Algorithm 4) to explore off-policy actions. A common strategy is to use ϵ-greedy exploration. With probability ϵ, it chooses a random action; with probability $1 - \epsilon$, it chooses the action with the highest Q-value. It has been shown that if each action is executed in each state an infinite number of times and the learning rate α is decayed appropriately, the Q-values will converge with probability 1 to the optimal values Q^* (Jaakkola et al., 1994; Tsitsiklis, 1994). Therefore, at test time, we execute the greedy policy that chooses the action with the maximum Q-value in any state.

Algorithm 4 works effectively for a small, discrete state and action space. For complex problems with continuous states, the Q-function cannot be expressed as a lookup table, requiring a continuous approximation. For example, in linear approximation models:

$$Q(s, a) = \boldsymbol{w} \cdot \phi(s, a), \tag{6.1}$$

where ϕ is a feature map that returns a d-dimensional vector representation of the state and action: $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$. The weight vector $\boldsymbol{w}$ is then updated as in stochastic gradient descent for linear regression. Line 7 of Algorithm 4 becomes

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \cdot \phi(s, a) \tag{6.2}$$

Deep RL uses a (deep) neural network to approximate the Q-function. Recently, the deep Q-Network (DQN) (Mnih et al., 2015) has shown great success in a variety of problems, including Atari games, Go (Silver et al., 2016) and text-based games (Narasimhan et al., 2015). Inspired by the success of DQN, we use it as our basic learning framework. The DQN algorithm is essentially Q-learning with experience replay. It draws samples (s, a, s', r) from a replay memory M, and the neural network predicts Q^* by minimizing squared loss at iteration i:

$$L^i(\theta^i) = \mathbb{E}_{(s,a,s',r)\sim U(M)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^{i-1}) - Q(s, a; \theta^i) \right)^2 \right], \tag{6.3}$$

where $U(M)$ is a uniform distribution over replay memory. We will build our opponent model upon the DQN framework.

6.4 Deep Reinforcement Opponent Network

In a multi-agent setting, the environment is affected by the *joint* action of all agents. From the perspective of one agent, the outcome of an action in a given state is no longer stable but is dependent on actions of other agents. In this sec-

tion, we first analyze the effect of multiple agents on the Q-learning framework; then we present our model Deep Reinforcement Opponent Network (DRON) and its multitasking variation.

6.4.1 Q-Learning with Opponents

In MDP terms, the joint action space is defined by $\mathcal{A}^M = \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n$ where n is the total number of agents. We use a to denote the action of the agent we control (the primary agent) and o to denote the joint action of all other agents (secondary agents), such that $(a, o) \in \mathcal{A}^M$. Similarly, the transition probability becomes $\mathcal{T}^M(s, a, o, s') = Pr(s'|s, a, o)$, and the new reward function is $\mathcal{R}^M(s, a, o, s')$. Our goal is to learn an optimal policy for the primary agent given interactions with the joint policy π^o of the secondary agents.[3]

If π^o is stationary, then the multi-agent MDP reduces to a single-agent MDP: the opponents can be considered as part of the world. Thus, they redefine the transition function and the reward function:

$$\begin{aligned} \mathcal{T}(s, a, s') &= \sum_o \pi^o(o|s)\mathcal{T}^M(s, a, o, s') \\ \mathcal{R}(s, a, s') &= \sum_o \pi^o(o|s)\mathcal{R}^M(s, a, o, s') \end{aligned}$$

Therefore, the agent need not be aware of the existence of other agents, and standard Q-learning should suffice.

Nevertheless, it is often unrealistic to assume that the opponent uses a fixed

[3]While a joint policy defines the distribution of joint actions, the opponents may be controlled by independent policies.

policy. Other agents may also be learning or adapting to maximize their rewards through time. For example, in strategy games, players may disguise their true strategies at the beginning to fool the opponents; a player in a winning situation tends to play more defensively and one in a losing situation may play more aggressively. In these situations, we face opponents with an unknown policy π_t^o that changes over time.

Considering the effects of other agents, the definition of an optimal policy no longer applies—the goodness of a policy now depends on policies of secondary agents. Therefore, we define the optimal Q-function relative to the joint policy of opponents: $Q^{*|\pi^o} = \max_\pi Q^{\pi|\pi^o}(s,a)\ \forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$. The recurrent relation between Q-values still holds:

$$Q^*_{\pi|\pi^o}(s_t, a_t) = \sum_{o_t} \pi_t^o(o_t|s_t) \sum_{s_{t+1}} \mathcal{T}(s_t, a_t, o_t, s_{t+1}) \left[\mathcal{R}(s_t, a_t, o_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q^*_{\pi|\pi^o}(s_{t+1}, a_{t+1})\right]. \tag{6.4}$$

6.4.2 DQN with Opponent Modeling

Given Equation 6.4, we can continue applying Q-learning and estimate both the transition function and the opponents' policy by stochastic updates. However, treating opponents as part of the world can result in slow response to an adaptive opponent (Uther and Veloso, 2003), since the dynamics of the world perceived by an agent is in fact changing.

To encode opponent behavior explicitly, we propose the Deep Reinforcement

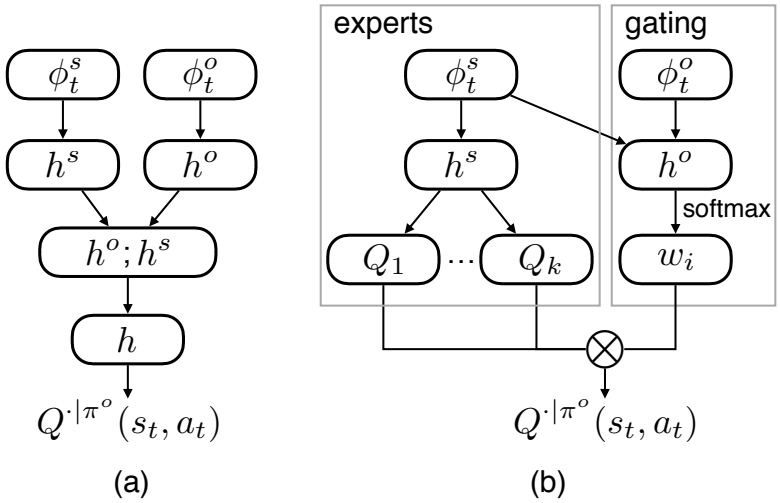


Figure 6.3: Diagram of the DRON architecture. (a) DRON-concat: the opponent representation is concatenated with the state representation. (b) DRON-MoE: Q-values predicted by K experts are combined linearly by weights from the gating network.

Opponent Network (DRON) that models $Q^{\cdot|\pi^o}$ and π^o jointly. DRON consists of a Q-Network (N_Q) that evaluates actions for a state and an opponent network (N_o) that learns a representation of π^o. Now the key questions are how to combine the two networks and what supervision signal to use. To answer the first question, we investigate two network architectures: DRON-concat that concatenates N_Q and N_o, and DRON-MOE that applies a Mixture-of-Experts model. To answer the second question, we consider two settings: (a) predicting Q-values only, as our goal is to maximize reward instead of building an accurate simulator of the opponent; and (b) also predicting extra information about the opponent when it is available, e.g., the type of their strategy.

DRON-concat N_Q and N_o embed the state and the opponent in separate hidden spaces (h^s and h^o) via standard mappings, e.g., linear layers with rectification or

convolutional neural networks. To incorporate knowledge of π^o into the Q-Network, we concatenate representations of the state and the opponent (Figure 6.3a). The concatenation then jointly predicts the Q-value. Therefore, the last layer(s) of the neural network is responsible for understanding the interaction between opponents and Q-values. Since there is only one Q-Network, the model requires a more discriminative representation of the opponents to learn an adaptive policy. To alleviate this, our second model encodes a stronger prior of the relation between opponents' actions and Q-values based on Equation 6.4.

DRON-MoE The right part of Equation 6.4 can be written as $\sum_{o_t} \pi_t^o(o_t|s_t)Q^\pi(s_t, a_t, o_t)$, an expectation over different opponent behavior. We use a Mixture-of-Experts network (Jacobs et al., 1991) to explicitly model the opponent action as a hidden variable and marginalize over it (Figure 6.3b). The expected Q-value is obtained by combining predictions from multiple *expert networks*:

$$\begin{aligned} Q(s_t, a_t; \theta) &= \sum_{i=1}^{K} w_i Q_i(h^s, a_t) \\ Q_i(h^s, \cdot) &= f(W_i^s h^s + b_i^s). \end{aligned}$$

Each expert network responds to a possible reward given the opponent's move. The combination weights (distribution over experts) are computed by a *gating network*:

$$w = \text{softmax}\left(f(W^o h^o + b^o)\right).$$

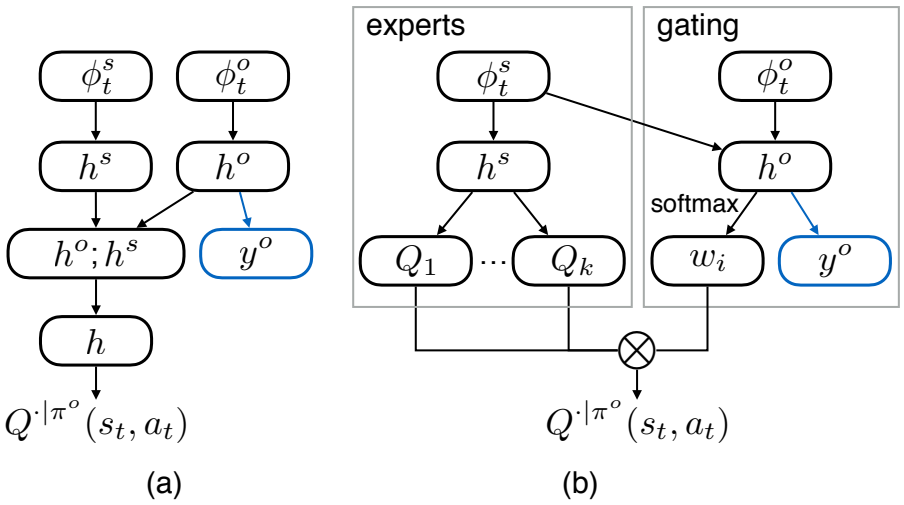


Figure 6.4: Diagram of the DRON with multitasking. The blue part shows that the supervision signal from the opponent affects the Q-learning network by changing the opponent features.

Here $f(\cdot)$ is a nonlinear activation function (we use ReLU for all experiments), W represents the linear transformation matrix and b is the bias term.

Unlike DRON-concat which is ignorant of the relation with opponents and solely relies on the data to learn it, DRON-MOE has the advantage of knowing that Q-values have different distributions depending on ϕ^o, and having each expert network manage one type of opponent strategy.

Multitasking with dron As shown in Figure 6.3, the previous two models aim to predict only Q-values, thus, the opponent representation is learned indirectly through feedback from the Q-value. If extra information about the opponent is available, we may use them as extra supervision for the opponent network to learn more discriminative representations. Fortunately, many games reveal additional information besides the final reward at the end of a game. At the very least the agent has observed actions taken by the opponents in past states; sometimes their

private information such as the hidden cards in poker. More high-level information includes abstracted plans or strategies after some analysis. Such information reflects characteristics of opponents and can aid policy learning.

Unlike previous work that learns a separate model to predict this information about the opponent (Davidson, 1999; Ganzfried and Sandholm, 2011; Schadd et al., 2007), we apply multitask learning and use the observation as extra supervision signals to learn a *shared* opponent representation h^o. Figure 6.4 shows the architecture of multitask DRON, where the supervision signal is denoted by y^o in blue. The advantage of multitasking over explicit opponent modeling is that it leverages high-level knowledge of the game and the opponent, while being robust to insufficient opponent data and modeling error due to the main supervision from Q-values. We evaluate multitasking DRON with two types of supervision signals, future action and overall strategy of the opponent.

We have described variants of DQN that incorporate an opponent model. Next, we introduce our problem, quiz bowl, and we will see the role of opponent modeling in interactive, multi-agent sequential problems.

6.5 Experiments

In this section, we first evaluate our models on a simulated soccer game, then on quiz bowl using real data. Both tasks have two players against each other and the opponent presents varying behavior. We compare DRON models with DQN and analyze their response against different types of opponents.

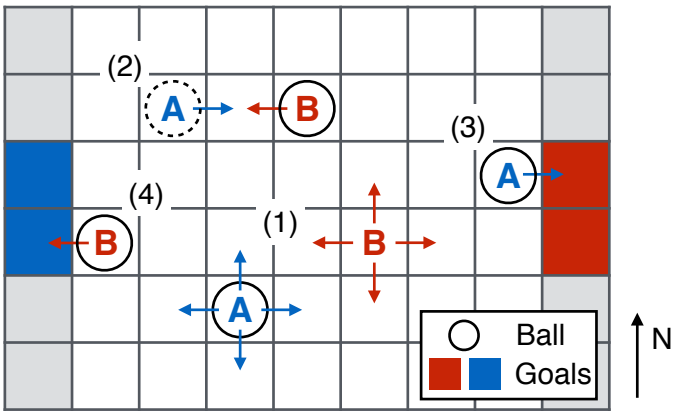


Figure 6.5: Illustration of the soccer game. Two players A and B compete to move the ball to the opponent's goal. Arrows show players' moving directions.

All systems are trained under the same Q-learning framework. Unless stated otherwise, the experiments have the following configuration: discount factor γ is 0.9, parameters are optimized by AdaGrad (Duchi et al., 2011) with a learning rate of 0.0005, and the mini-batch size is 64. We use ϵ-greedy exploration during training, starting with an exploration rate of 0.3 that linearly decays to 0.1 within 500,000 steps. We train all models for fifty epochs and select the one with the highest average reward on the development set for testing.

6.5.1 Soccer

Setup Our first testbed is a soccer variant following previous work on multi-player games (Littman, 1994; Collins, 2007; Uther and Veloso, 2003). The game is played on a 6×9 grid (Figure 6.5) by two players, A and B.[4] The game starts with A and B standing in a randomly selected square in the left and right half (except the goals), and the ball going to one of them randomly. Players choose from five actions:

[4]Although the game is played in a grid world, we do not represent the Q-function in tabular form as in previous work. Therefore, it can be generalized to more complex pixel-based settings.

`move N, S, W, E` or `stand` still (Figure 6.5(1)). An action is invalid if it takes the player to a shaded square or outside of the border. If two players move to the same square, the player who possesses the ball before the move loses it to the opponent (Figure 6.5(2)), and the move does not take place. Therefore, a good strategy to intercept the ball is to move to where the opponent will be. A player scores one point if they take the ball to the opponent's goal (Figure 6.5(3), (4)) and the game ends. If neither player gets a goal within one hundred steps, the game ends with a tie and both players get zero points.

	Defensive	Offensive
w/ ball	Avoid opponent	Advance to goal
w/o ball	Defend goal	Intercept the ball

Table 6.2: Strategies of the hand-crafted rule-based agent.

Implementation We design a rule-based agent as the opponent. It has a defensive mode and an offensive mode. We show its strategies under different conditions in Table 6.2. The offensive agent always prioritizes attacking over defending. In 5000 games against a random agent, it wins 99.86% of the time and the average episode length is 10.46. The defensive agent only focuses on defending its own goal. As a result, it wins 31.80% of the games and ties 58.40% of them; the average episode length is 81.70. To simulate a varying strategy, we let the agent randomly choose between the two modes in each game.

The input state is a 1×15 vector representing coordinates of the agent, the opponent, the axis limits of the field, positions of the goal areas and possession of

Model	Basic	Multitask +action	Multitask +type
	Max R		
DRON-concat	0.682	0.695*	0.690*
DRON-MOE	**0.699***	0.697*	0.686*
DQN-world	0.664	-	-
	Mean R		
DRON-concat	0.660	0.672	0.669
DRON-MOE	0.675	0.664	0.672
DQN-world	0.616	-	-

Table 6.3: Rewards of DQN and DRON models on Soccer. We report the maximum test reward ever achieved (Max R) and the average reward of the last 10 epochs (Mean R). Statistically significant ($p < 0.05$ in two-tailed pairwise t-tests) improvement is shown for the DQN (*) and all other models (**bold**). DRON models achieve higher rewards in both measures.

the ball. We define a player's move by five cases: approaching the agent, avoiding the agent, approaching the agent's goal, approaching self goal and standing still. The opponent features include frequencies of observed opponent moves, its most recent move and action, and the frequency of losing the ball to the opponent.

The baseline DQN has two hidden layers, both with 50 hidden units. We call this model DQN-world, meaning treating the opponents as part of the world. The hidden layer of the opponent network in DRON also has 50 hidden units. For multitasking, we experiment with two supervision signals, opponent action in the current state (+action) and the opponent mode (+type). We use cross entropy as the loss function in both settings.

Results In Table 6.3, we compare rewards of DRON models, their multitasking variations, and DQN-world. After each epoch, we evaluate the policy with 5000

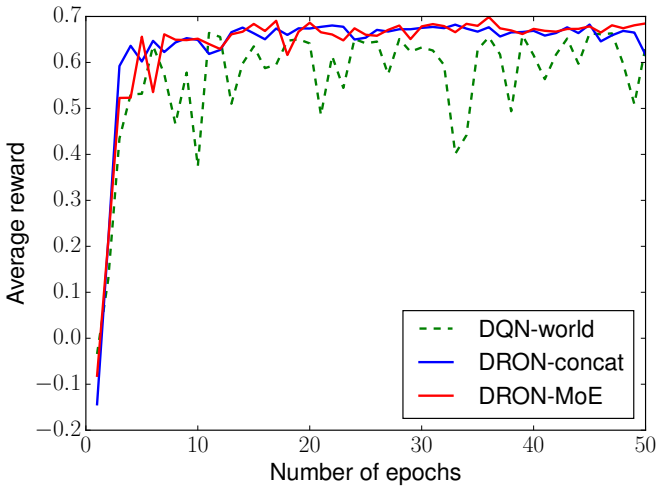


Figure 6.6: Learning curves on Soccer over fifty epochs. DRON models are more stable than DQN.

randomly generated games and compute the average reward. We report the mean test reward after the model stabilizes and the maximum test reward ever achieved. The DRON models outperform the DQNbaseline. Our model also has much smaller variance (Figure 6.6).

From the "Multitask" column we see that adding additional supervision signals improves DRON-concat but not DRON-MOE. We suspect this is because DRON-concat does not explicitly learn different strategies for different types of opponents, therefore more discriminative opponent representation helps it model the relation between opponent behavior and Q-values better. However, for DRON-MOE, while better opponent representation is still desirable, the supervision signal may not be aligned with "classification" of the opponents learned from the Q-values.

To investigate how the learned policies adapt to different opponents, we let the agents play against a defensive opponent and an offensive opponent. Furthermore, to understand the best an agent can do in these two settings, we train two DQN

	DQN		DQN	DRON	DRON
	O only	D only	-world	-concat	-MOE
O	**0.897**	-0.272	0.811	0.875	0.870
D	0.480	**0.504**	0.498	0.493	0.486

Table 6.4: Average rewards of DQN and DRON models when playing against different types of opponents. Offensive and defensive agents are represented by O and D. "O only" and "D only" means training against O and D agents only. Upper bounds of rewards are in bold. DRON achieves rewards close to the upper bounds against both types of opponents.

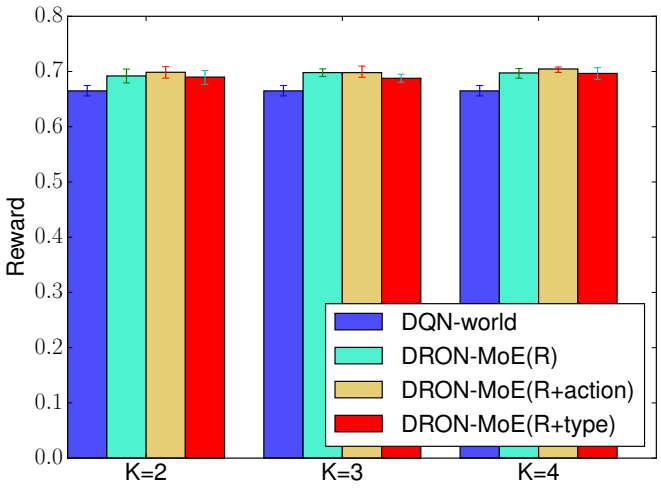


Figure 6.7: Effect of varying the number experts (2–4) and multitasking on Soccer. The error bars show the 90% confidence interval. DRON-MOE consistently improves over DQN regardless of the number of mixture components. Adding extra supervision does not obviously improve the results.

agents against the offensive and defensive opponents. The rest of the agents are trained against the random opponent. Table 6.4 shows their average rewards and upper bounds (in bold). DQN-world is confused by the defensive behavior and significantly sacrifices its performance against the offensive opponent; DRON achieves a much better trade-off, retaining rewards close to the upper bounds against a varying opponent.

Finally, we examine how the number of experts in DRON-MOE affects the result.

From Figure 6.7, we see no significant difference in varying the number of experts, and DRON-MOE consistently performs better than DQN across all K. Similar to the results in Table 6.3, multitasking is not obviously helpful here.

6.5.2 Quiz Bowl

Setup We collect question/answer pairs and log user buzzes from Protobowl, an online multi-player quizbowl application.[5] Additionally, we include data from Boyd-Graber et al. (2012). After removing answers with fewer than 20 questions and users who played fewer than twenty games, we end up with 1045 answers and 37.7k questions. We divide all questions into two non-overlapping sets: one for training the content model and one for training the buzzing policy. There are clearly two clusters of players (Figure 6.8(a)): aggressive players who buzz early with varying accuracies and cautious players who buzz late but maintain higher accuracy. Our GRU content model (Figure 6.8(b)) is more accurate with more input words—a behavior similar to human players.

Implementation Our input state must represent information from the content model and the opponents. Information from the content model takes the form of a *belief vector*: a vector (1×1045) that represents the current estimate of each possible guess being the correct answer of the question given our current input represented as log probabilities. As described in Section 3.2, we use both task features from the content model and meta-features. Specifically, we concatenate the belief vector

[5] http://protobowl.com

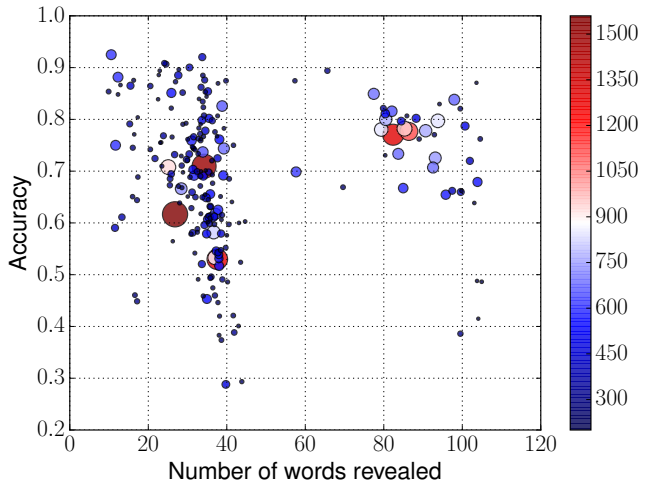
from the previous time step, which allows the model to capture sudden shifts in certainty, which are often good opportunities to buzz. In addition, we include the number of words seen and whether a wrong buzz has happened.

The opponent features include the number of questions the opponent has answered, the average buzz position and the error rate. The basic DQN has two hidden layers, both with 128 hidden units. The hidden layer for the opponent has ten hidden units. Similar to the soccer game, we experiment with two settings for multitasking: (a) predicting how likely the opponent will buzz; (b) predicting the type of the opponent. We approximate the ground truth for (a) by $\min(1, t/\text{buzz position})$ and use the mean square error as the loss function. The ground truth for (b) is based on dividing players into four groups according to their buzz positions—the percentage of question revealed—and cross entropy is used as the loss function.

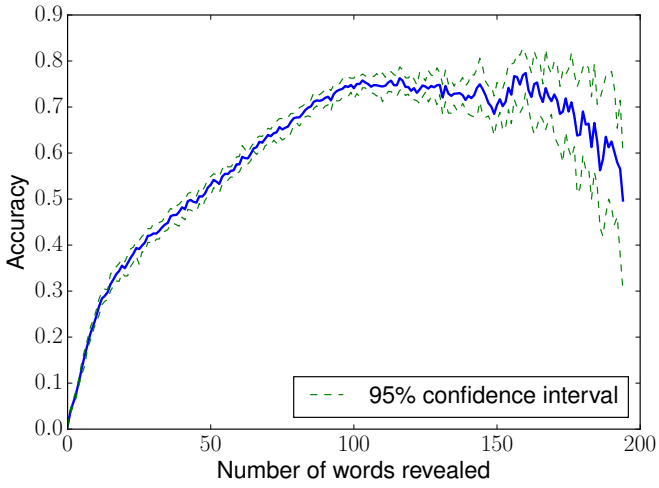
Results In addition to DQN-world, we also compare with DQN-self, a baseline without interaction with opponents at all. DQN-self is ignorant of the opponents and learns to play the safe strategy: answer as soon as the content model is confident. During training, when the answer prediction is correct, it receives a reward of 10 for `buzz` and -10 for `wait`. When the answer prediction is incorrect, it receives a reward of -15 for `buzz` and 15 for `wait`. Since all rewards are immediate, we set γ to 0 for DQN-self. With data of the opponents' responses, DRON and DQN-world use the game payoff (from the perspective of the computer) as the reward.

First, we compare the average rewards on the test set of our models, DRON-concat and DRON-MOE (with 3 experts), and the baseline models, DQN-self and DQN-

world. From the first column in Table 6.5, our models achieve statistically significant improvements over the DQN baselines and DRON-MOE outperforms DRON-concat. In addition, the DRON models have much less variance compared to DQN-world as the learning curves show in Figure 6.11.



(a)



(b)

Figure 6.8: Accuracy vs. the number of words revealed. (a) Real-time user performance. Each dot represents one user; dot size and color correspond to the number of questions the user answered. (b) Content model performance. Accuracy is measured based on predictions at each word. Accuracy improves as more words are revealed.

Model	Basic	Multitask +action	Multitask +type	Basic vs. opponents buzzing at different positions (%revealed (#episodes)) $0-25\%$ (4.8k)			$25-50\%$ (18k)			$50-75\%$ (0.7k)			$75-100\%$ (1.3k)		
	$R\uparrow$			$R\uparrow$	rush$\downarrow$	miss$\downarrow$	$R\uparrow$	rush$\downarrow$	miss$\downarrow$	$R\uparrow$	rush$\downarrow$	miss$\downarrow$	$R\uparrow$	rush$\downarrow$	miss$\downarrow$
DRON-concat	1.04	**1.34***	**1.25**	-0.86	0.06	0.15	1.65	0.10	0.11	-1.35	0.13	0.18	0.81	0.19	0.12
DRON-MOE	**1.29***	1.00	**1.29***	-0.46	0.06	0.15	1.92	0.10	0.11	-1.44	0.18	0.16	0.56	0.22	0.10
DQN-world	0.95	-	-	-0.72	0.04	0.16	1.67	0.09	0.12	-2.33	0.23	0.15	-1.01	0.30	0.09
DQN-self	0.80	-	-	-0.46	0.09	0.12	1.48	0.14	0.10	-2.76	0.30	0.12	-1.97	0.38	0.07

Table 6.5: Comparison between DRON and DQN models. The left column shows the average reward of each model on the test set. The right column shows performance of the basic models against different types of players, including the average reward (R), the rate of buzzing incorrectly (rush) and the rate of missing the chance to buzz correctly (miss). $\uparrow$ means higher is better and $\downarrow$ means lower is better. In the left column, we indicate statistically significant results ($p < 0.05$ in two-tailed pairwise t-tests) with boldface for vertical comparison and * for horizontal comparison.

To investigate strategies learned by these models, we show their performance against different types of players (as defined at the end of "Implementation") in the right columns in Table 6.5. We compare three measures of performance, the average reward (R), percentage of early and incorrect buzzes (rush), and percentage of missing the chance to buzz correctly before the opponent (miss). All models beat Type 2 players, mainly because they are the majority in our dataset. As expected, DQN-self learns a safe strategy that tends to buzz early. It performs the best against Type 1 players who answers early. However, it has very high rush rate against cautious players, resulting in much lower rewards against Type 3 and Type 4 players. Without opponent modeling, DQN-world is biased towards the majority player, thus having the same problem as DQN-self when playing against players who buzz late. Both DRON models successfully learn to exploit the cautious players while maintaining a competent performance against the aggressive players. Furthermore, DRON-MOE matches DQN-self's performance on the Type 1 players, demonstrating that it discovers different buzzing strategies.

In Figure 6.9, we show an example question with buzz positions labeled. The DRON agents demonstrate dynamic behavior against different players; DRON-MOE almost always buzzes right before the opponent in this example. In addition, when the player buzzes wrong and the game continues, DRON-MOE learns to wait longer since the opponent is gone, while the other agents are still in a rush.

Next, we show results with multitasking with two supervision signals, next action (+action) and type (+type) of the opponent. Similar to what we observed in Soccer, adding extra supervision does not yield better results over DRON-MOE

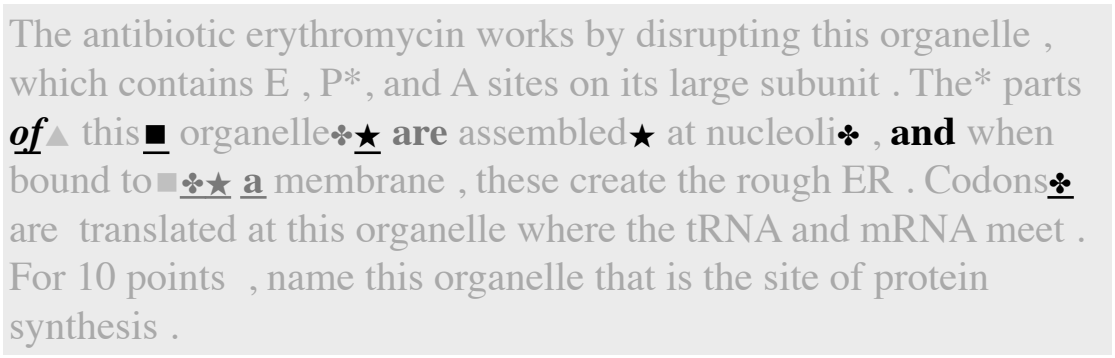


Figure 6.9: Buzz positions of human players and agents on one science question whose answer is "ribosome". Words where a player buzzes are displayed in a combination of color and underline unique to the player; a wrong buzz is shown in *italic*; a buzz with a star (∗) indicates a fast one before all machine buzzes. Words where an agent buzzes is subscripted by a symbol unique to the agent; the format of the symbol corresponds to the player it is playing against. The lightest gray color (of DQN-self on one DQN-world) means that the buzz position of the agent does not depend on its opponent. DRON agents adjust their buzz positions according to the opponent's buzz position and correctness.

(Table 6.5) but significantly improves DRON-concat. Figure 6.10 varies the number of experts in DRON-MOE (K) from two to four. Using a mixture model for the opponents consistently improves over the DQN baseline, and using three experts gives better performance on this task. For multitasking, adding the action supervision does not help at all. However, the more high-level type supervision yields competent results, especially with four experts, mostly because the number of experts matches the number of types.

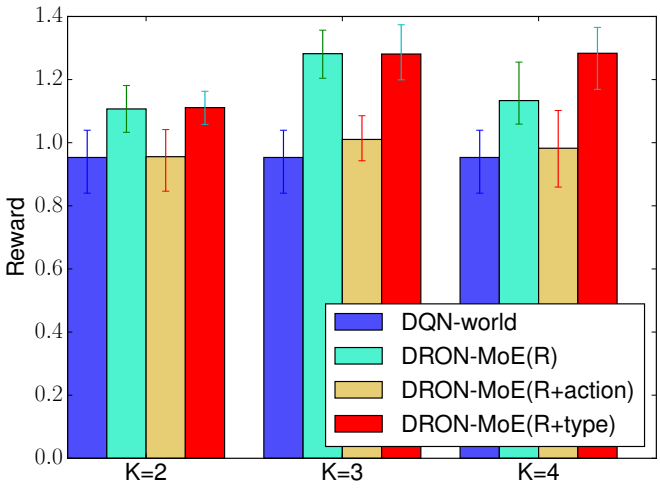


Figure 6.10: Effect of varying the number experts (2–4) and multitasking on quiz bowl. The error bars show the 90% confidence interval. DRON-MOE consistently improves over DQN regardless of the number of mixture components. Supervision of the opponent type is more helpful than the specific action taken.

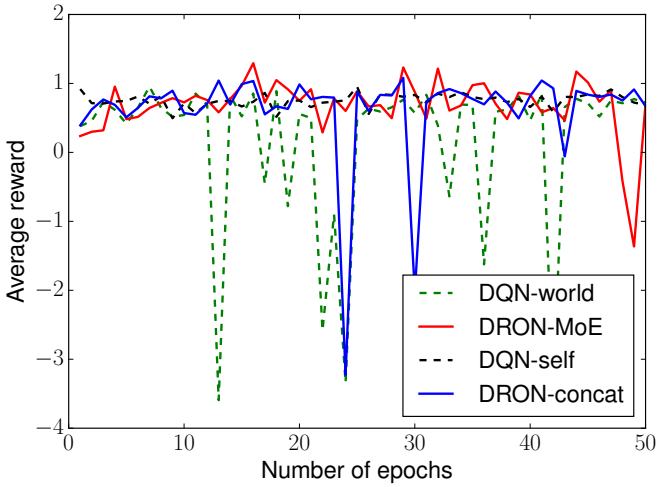


Figure 6.11: Learning curves on Quizbowl over fifty epochs. DRON models are more stable than DQN.

6.6 Related Work

Implicit vs. Explicit opponent modeling Opponent modeling has been studied extensively in games. Most existing approaches fall into the category of explicit modeling, where a model (e.g., decision trees, neural networks, Bayesian models) directly predicts parameters of the opponent, e.g., actions (Uther and Veloso, 2003; Ganzfried and Sandholm, 2011), private information (Billings et al., 1998b; Richards and Amir, 2007), or domain-specific strategies (Schadd et al., 2007; Southey et al., 2005). Here one difficulty is that the model may need a prohibitive number of examples before producing anything useful. Another is that as the opponent behavior is modeled separately from dynamics of the world, it is not always clear how to incorporate these predictions robustly into policy learning. The results on multitasking DRON also suggest that improvement from explicit modeling is limited. However, it is better suited to games of incomplete information, where it is clear what information needs to be predicted to achieve a higher reward.

Our work is closely related to implicit opponent modeling. Since the agent aims to maximize its own expected reward without having to identify the opponent's strategy, this approach does not have the difficulty of incorporating prediction of the opponent's parameters. In Rubin and Watson (2011) and Bard et al. (2013), first a portfolio of strategies are constructed offline based on domain knowledge or past experience, then strategies are selected online using multi-arm bandit algorithms. Both models focus on heads-up limit Texas hold'em poker. Our approach does not have a clear online/offline distinction. Instead, we learn strategies and their

combiner jointly in a simpler way, thus we require less domain knowledge of the problem. Nevertheless, offline initialization can be easily enabled in our model by initializing expert networks with DQN models pre-trained against a particular type of opponents.

Neural network opponent models Davidson (1999) applies neural networks to opponent modeling, where a simple multi-layer perceptron is trained as a classifier to predict opponent actions given game logs. Lockett et al. (2007) propose an architecture similar to DRON-concat that aims to identify the type of an opponent. However, instead of learning a hidden representation, they learn a mixture weights over a pre-specified set of cardinal opponents; and they use the neural network as a standalone solver without the reinforcement learning setting, which may not be suitable for more complex problems. Foerster et al. (2016) use modern neural networks to learn a group of parameter-sharing agents that solve a coordination task, where each agent is controlled by a deep recurrent Q-Network (Hausknecht and Stone, 2015). Our setting is different in that we control only one agent and the policy space of other agents is unknown. Opponent modeling with neural networks remains understudied with ample room for improvement.

6.7 Conclusion

This chapter considers sequential decision-making when more than one active agents are acting in the environment, which is common in collaborative and competitive settings. Our method takes advantage of the recent deep Q-learning and

learns the representation of opponents which affects the rewards. The model is also flexible enough to include supervision for parameters of the opponents, much as in explicit modeling.

In general, opponent modeling methods can be applied to NLP systems that interact with humans. In fact, the prediction approach for simultaneous translation introduced in the last chapter can be considered as a form of opponent modeling. If we consider the speaker as another agent in the environment, then the system is trying to prediction the opponent action (next word, final verb). Another example would be building user models for persuasion or negotiation dialogs.

Chapter 7: Conclusion

7.1 Summary

In this thesis, we address problems with sequential predictions and decisions in NLP, and answer the following questions:

- When to solve a problem sequentially?
- How to formulate the sequential decision-making process?
- How to extend the framework to different domains?

To answer these questions, we focus on two scenarios: when the input information is costly, we select parts of it sequentially to make predictions at a lower cost; when the input is revealed incrementally, we decide when to make an early prediction based on the amount of information we have. The common characteristics of these two types of problems are (a) both takes a sequence of inputs and produces a sequence of predictions; (b) each prediction in the sequence has some cost; (b) there is a cost-quality trade-off in deciding when to output/commit to the current prediction.

Specifically, we have looked at dynamic feature selection for both classification and dependency parsing, simultaneous translation, and quiz bowl (incremental text

classification). We showed basic MDP components in each of these problems. In general, the state contains inputs so far and intermediate predictions; the actions are the decisions the system needs to make; the loss/reward encodes the desired trade-off between information cost and prediction quality.

In addition, we described multiple ways to extend the sequential decision-making framework. The most straightforward one is to change the action set and state space. For example, in sequential acquisition problems, we can either select from all available features or decide whether to take a feature from a ranked list. In simultaneous translation, we can add richer actions such as prediction the next word and the final verb. Another way is to modify the baseline predictor (e.g. the classifier trained with complete information, the batch translator). For example, we train the batch translator with paraphrased inputs so that it produces more monotone translations, fitting better to the sequential setting.

The main machine learning tools we use are imitation learning and reinforcement learning algorithms. They are two common approaches to solving sequential decision-making problems using the MDP formulation. Besides demonstrating their effectiveness in solving NLP problems, we further proposed adaptations of the standard algorithms, including using coaching in imitation learning when the expert is too good, and modeling opponents when multiple agents are affecting the environment in reinforcement learning.

7.2 Future Directions

While this thesis has focused on sequential acquisition and sequential revealing problems, there are several other directions worth future pursuit, in both learning method and application.

7.2.1 Joint Learning of Predictor and Policy

In most of our applications, we separated the task predictor and the policy. The task predictor is pre-trained in the static or batch stetting where no decision is involved, and the policy is trained given a fixed task predictor. This design has one potential problem that the task predictor is not prepared for handling partial inputs. Dulac-Arnold (2014) considers possible predictions as different actions in addition to actions taken by the controller. The disadvantage of this approach is that learning can be hard if the prediction space is large; also, the task predictor and the policy may require different kinds of features.

Another way to alleviate the problem is to give the task predictor randomly sampled partial inputs during pretraining. For example, we can train a classifier with randomly selected features. To adapt the task predictor to partial examples likely to form at test time, we can fine-tune the pre-trained predictor during policy learning: feeding it partial examples produced by the current learned policy.

Alternatively, we can take a multitasking approach to learning the task predictor and the policy: they can be two modules with shared parameters and representation. Since outputs of the two modules are interdependent, it is reasonable to have

overlap between them. This approach avoids disadvantages of either considering them as a single predictor or completely separate predictors.

7.2.2 Context Representation

One important aspect in sequential prediction is context representation. The standard MDP assumes the Markov property for state transition. This is sometimes a simplified model for real applications. In fact, state features often include features from previous states in practice. Context also plays a central role in language understanding. For example, a dialog system needs to maintain a flow of topics to fully understand the user. If a user says "What about tomorrow", they may be asking about the weather tomorrow, or proposing a meeting time. The actual answer depends much on previous interaction.

Recent work (Zhang et al., 2016; Mnih et al., 2014) has used an LSTM to encode a sequence of states to incorporate history information in reinforcement learning. When the sequence becomes longer, it is interesting to consider a hierarchical representation that captures both global and local context. For example, in simultaneous translation of TED talks, we can model the talk title/abstract as the global context, which can be combined with the local word sequence to predict future content.

It would also be helpful for the system to have an external memory to save information that might be useful in future. The external memory can be updated in each step by rewriting some memory slots. Therefore, the system needs to make de-

cisions about which memories are out-dated and should be removed, which memories can be combined to form a higher-level concept, and whether the new information should be written for future use.

7.2.3 Extensions of Sequential Selecting and Revealing Problems

We have seen dependency parsing as a sequential selection problem, as well as simultaneous interpretation and quiz bowl as sequential revealing problems. Being able to select needed information dynamically and produce an early prediction for streaming data are key to efficient systems given a plethora of information, and they have wider application than what we have worked on. Below we detail two other directions.

NLP with World Knowledge Currently most NLP systems work as standalone applications without a context of the world. However, world knowledge is essential in language understanding. For example, consider the following narrative:

"Alice met Bob on Friday afternoon. They had some fun time. In the end, Bob proposed to watch a show together the next day. Alice agreed."

To answer the question "which day will Alice and Bob watch a show", one must know that the day following Friday is Saturday. Such knowledge is considered common sense in human communication, thus it cannot be inferred from the input narrative alone, and a knowledge base is required. Unfortunately, considering world knowledge is expensive in practice. A knowledge base usually has millions of entries and it takes time to identify relevant information. This is where dynamic selection

can help: we only need to query the knowledge base when ambiguity is present or the reasoning is broken (e.g., due to missing knowledge). In our sequential selection framework, the knowledge base is considered as another piece of costly input and we decide whether to use it or not depending on specific instances.

Customer Service Our work on quiz bowl is relevant to incremental text classification. Early prediction of streaming text is important in time-sensitive applications such as high-frequency trading. Another area that needs fast text processing is customer service call center. At least half of the cost in call centers is spent on labor, thus reducing communication time is crucial to reduce cost. A good customer service chat bot should direct the user to a corresponding agent as fast as possible, otherwise, it may block other customers and lower satisfaction rate. We can improve processing speed from two aspects. First, the system can infer user intention before one finishes talking, much as in quiz bowl. Second, the system needs to decide the next question to ask so as to finish the task in a minimum number of steps. This is also related to goal-oriented dialogs, which we explain in Section 7.2.5.

7.2.4 Interpretable Prediction and Decision

Although machine learning models are often used as a black-box in practice, in some domains it is important to understand reasons behind these predictions. For example, when making medical, business or government decisions, a wrong prediction can have serious consequences, and a desirable model is desirable. Sequential prediction can help expose a model's "decision process" by examining its reaction

> The film that unfolds from these beginnings is in many ways a conventional one, **but** it unfolds with so much wit, panache, and visual ingenuity that it outstrips many a more high-concept movie.

Figure 7.1: Example movie review. Words in red indicate negative opinion while words in blue indicate positive opinion.

to different information.

We show a snippet of an example movie review in Figure 7.1. Suppose we have a sentiment analysis model and we want to interpret its prediction for the snippet. One approach is to use the model to produce a sequence of predictions at each word. By looking at changes in the output, we can identify specific parts of the input leading to the final prediction. In the example, a reasonable model would predict negative opinion after the word "convention", become less confident at the contrastive conjunction "but", and predict positive opinion after seeing "wit, panache, and visual ingenuity".

The above method is mainly used to inspect a trained model, nevertheless, we can also directly frame prediction as a sequential decision-making process that reflects effects of different pieces of the input. One direction is to equip the model with memory. As we scan the input sequentially, we decide whether to add a piece to the memory, to delete a piece from the memory or to combine with a piece of the memory. Such composition reveals what the model is "thinking" during prediction. We use the example in Figure 7.1 for illustration again. Here "convention" is an important word that should be added to the memory first, however, after seeing "but" we can delete it as this conjunction emphasize content after it. Finally, the model should mainly focus on "wit, panache, and visual ingenuity" in the memory.

7.2.5 Interaction with Humans

In Chapter 6 we briefly explored systems that interact with humans. Instead of statically receiving user input and producing an output independent of any context, we believe an NLP system can benefit from considering user interaction as a sequential process. Below we detail some directions along this line.

Information Retrieval When searching for information about an event, a user is likely to issue a sequence of related queries regarding the event; in addition, later queries may depend on new information found in earlier results. Therefore, considering the context in a query sequence can help disambiguation. For instance, if a user first searches "GOP nomination", followed by "the wall", then it is much more likely that "the wall" means the Trump Wall other than the album by Pink Floyd. In this setting, the system does not necessarily make any explicit decision, however, the context given by previous queries and outputs should be considered in the current state and affect ranking of current search results.

Dialog Dialog is probably the most typical sequential NLP problem, which can be broadly classified as goal-oriented dialogs and open-ended dialogs. Open-ended dialog system focuses on building a chat bot. We are more interested in goal-oriented dialog systems, which complete a task specified by the user through conversation, for example, personal assistants such as Siri by Apple, Echo by Amazon, Cortana by Microsoft and Google Now. These dialogs are often formulated as a partially observable MDP (Young, 2006), where intention (state) of the user is modeled probabilistically.

One interesting area is diplomatic dialogs, e.g., negotiation, persuasion, and argumentation. Here uncertainty of the user intention is not only due to the limitation of speech recognition and language understanding but also due to strategy of the user (e.g. faking an intention). Identifying the user's strategy is related to opponent modeling. In addition to that, the system needs a counter strategy to achieve a cooperative or competitive goal, which can be modeled a decision-making process. Finally, the action needs to be rendered into natural language understandable by users.

Human-In-the-Loop Learning In Chapter 3 we have seen how a system can acquire pieces of input adaptively. Taking this framework one step further, we can consider human as a costly subroutine available during prediction. Of course, we do not want to directly ask a person to label the example for us. Instead, we can decompose the task to smaller, simpler subtasks which can be answered by a person without too much effort. For example, We can ask a human to translate part of a sentence (e.g. an unknown word, a phrase) that we are not sure about. This way, even though the system cannot completely replace a human expert, it can lower the requirement for an expert.

Appendix A: Proof for DAgger with Coaching

The theorem is as follows:

Theorem 5. *Assume ℓ upper bounds L, π^* is u-robust, and N is $O(uT \log T)$, then there exists a policy $\pi \in \pi_{1:N}$ such that:*

$$J(\pi) \leq J(\pi^*) + uT\tilde{\epsilon}_{class} + O(1).$$

We consider a policy π parameterized by a vector $\boldsymbol{w} \in \mathbb{R}^d$. Let $\phi \colon S \times A \to \mathbb{R}^d$ be a *feature map* describing the state. The predicted action is

$$\hat{a}_{\pi,s} = \arg\max_{a \in A} \boldsymbol{w}^T \phi(s, a) \tag{A.1}$$

and the hope action is

$$\tilde{a}_{\pi,s} = \arg\max_{a \in A} \lambda \boldsymbol{w}^T \phi(s, a) - L(s, a). \tag{A.2}$$

We assume that the loss function $\ell \colon \mathbb{R}^d \to \mathbb{R}$ is a convex upper bound of the 0-1 loss. Further, it can be written as $\ell(s, \pi, \pi^*(s)) = f(\boldsymbol{w}^T \phi(s, \pi(s)), \pi^*(s))$ for a function $f \colon \mathbb{R} \to \mathbb{R}$ and a feature vector $\|\phi(s, a)\|_2 \leq R$. We assume that f is twice

differentiable and convex in $\boldsymbol{w}^T\phi(s,\pi(s))$, which is common for most loss functions used by supervised classification methods.

It has been shown that given a strongly convex loss function ℓ, *Follow-The-Leader* has $O(\log N)$ regret (Hazan et al., 2006; Kakade and Shalev-shwartz, 2008). More specifically, given the above assumptions we have:

Theorem 6. *Let* $D = \max_{\boldsymbol{w}_1,\boldsymbol{w}_2\in\mathbb{R}^d} \|\boldsymbol{w}_1-\boldsymbol{w}_2\|_2$ *be the diameter of the convex set* $\mathbb{R}^d$. *For some* $b, m > 0$, *assume that for all* $\boldsymbol{w} \in \mathbb{R}^d$, *we have* $|f'(\boldsymbol{w}^T\phi(s,a))| \leq b$ *and* $|f''(\boldsymbol{w}^T\phi(s,a))| \geq m$. *Then* Follow-The-Leader *on functions* ℓ *have the following regret:*

$$\begin{aligned}\sum_{i=1}^{N}\ell_i(\pi_i) - \min_{\pi\in\Pi}\sum_{i=1}^{N}\ell_i(\pi) &\leq \sum_{i=1}^{N}\ell_i(\pi_i) - \sum_{i=1}^{N}\ell_i(\pi_{i+1}) \\ &\leq \frac{2nb^2}{m}\left[\log\left(\frac{DRmN}{b}\right)+1\right] \qquad \text{(A.3)}\end{aligned}$$

To analyze the regret using surrogate loss with respect to hope actions, we use the following lemma:

Lemma 1. $\sum_{i=1}^{N}\ell_i(\pi_i) - \min_{\pi\in\Pi}\sum_{i=1}^{N}\tilde{\ell}_i(\pi) \leq \sum_{i=1}^{N}\ell_i(\pi_i) - \sum_{i=1}^{N}\tilde{\ell}_i(\pi_{i+1})$.

Proof. We prove inductively that $\sum_{i=1}^{N}\tilde{\ell}_i(\pi_{i+1}) \leq \min_{\pi\in\Pi}\sum_{i=1}^{N}\tilde{\ell}_i(\pi)$.

When $N = 1$, by *Follow-The-Leader* we have $\pi_2 = \arg\min_{\pi\in\Pi}\tilde{\ell}_1(\pi)$, thus $\tilde{\ell}_1(\pi_2) = \min_{\pi\in\Pi}\tilde{\ell}_1(\pi)$.

Assume correctness for $N-1$, then

$$\begin{aligned}\sum_{i=1}^{N}\tilde{\ell}_i(\pi_{i+1}) &\leq \min_{\pi\in\Pi}\sum_{i=1}^{N-1}\tilde{\ell}_i(\pi)+\tilde{\ell}_N(\pi_{N+1}) \quad \text{(inductive assumption)}\\ &\leq \sum_{i=1}^{N-1}\tilde{\ell}_i(\pi_{N+1})+\tilde{\ell}_N(\pi_{N+1}) \;=\; \min_{\pi\in\Pi}\sum_{i=1}^{N}\tilde{\ell}_i(\pi)\end{aligned}$$

The last equality is due to the fact that $\pi_{N+1}=\arg\min_{\pi\in\Pi}\sum_{i=1}^{N}\tilde{\ell}_i(\pi)$. □

To see how learning from $\tilde{\pi}_i$ allows us to approaching π^*, we derive the regret bound of $\sum_{i=1}^{N}\ell_i(\pi_i)-\min_{\pi\in\Pi}\sum_{i=1}^{N}\tilde{\ell}_i(\pi)$.

Theorem 7. *Assume that $\boldsymbol{w}_i$ is upper bounded by C, i.e. for all i $\|\boldsymbol{w}_i\|_2\leq C$, $\|\phi(s,a)\|_2\leq R$ and $|L(s,a)-L(s,a')|\geq\epsilon$ for some action $a,a'\in A$. Assume λ_i is non-increasing and define n_λ as the largest $n<N$ such that $\lambda_i\geq\frac{\epsilon}{2RC}$. Let $\ell_{\max}$ be an upper bound on the loss, i.e. for all i, $\ell_i(s,\pi_i,\pi^*(s))\leq\ell_{\max}$. We have*

$$\sum_{i=1}^{N}\ell_i(\pi_i)-\min_{\pi\in\Pi}\sum_{i=1}^{N}\tilde{\ell}_i(\pi)\leq 2\ell_{\max}n_\lambda+\frac{2nb^2}{m}\left[\log\left(\frac{DRmN}{b}\right)+1\right] \tag{A.4}$$

Proof. Given Lemma 1, we only need to bound the RHS, which can be written as

$$\left(\sum_{i=1}^{N}\ell_i(\pi_i)-\tilde{\ell}_i(\pi_i)\right)+\left(\sum_{i=1}^{N}\tilde{\ell}_i(\pi_i)-\tilde{\ell}_i(\pi_{i+1})\right). \tag{A.5}$$

To bound the first term, we consider a binary action space $A=\{1,-1\}$ for clarity. The proof can be extended to the general case in a straightforward manner.

Note that in states where $a_s^*=\tilde{a}_{\pi,s}$, $\ell(s,\pi,\pi^*(s))=\ell(s,\pi,\tilde{\pi}(s))$. Thus we only

need to consider situations where $a_s^* \neq \tilde{a}_{\pi,s}$:

$$\begin{aligned} & \ell_i(\pi_i) - \tilde{\ell}_i(\pi_i) \\ = \; & \mathbb{E}_{s\sim d_{\pi_i}}\left[\left(\ell_i(s,\pi_i,-1) - \ell_i(s,\pi_i,1)\right)\mathbf{1}_{\{s:\,\tilde{a}_{\pi_i,s}=1, a_s^*=-1\}}\right] \\ & + \mathbb{E}_{s\sim d_{\pi_i}}\left[\left(\ell_i(s,\pi_i,1) - \ell_i(s,\pi_i,-1)\right)\mathbf{1}_{\{s:\tilde{a}_{\pi_i,s}=-1, a_s^*=1\}}\right] \end{aligned} \tag{A.6}$$

In the binary case, we define $\Delta L(s) = L(s,1) - L(s,-1)$ and $\Delta\phi(s) = \phi(s,1) - \phi(s,-1)$.

Case 1 $\tilde{a}_{\pi_i,s} = 1$ and $a_s^* = -1$.

$\tilde{a}_{\pi_i,s} = 1$ implies $\lambda_i \boldsymbol{w}_i^T \Delta\phi(s) \geq \Delta L(s)$ and $a_s^* = -1$ implies $\Delta L(s) > 0$. Together we have $\Delta L(s) \in (0, \lambda_i \boldsymbol{w}_i^T \Delta\phi(s)]$. From this we know that $\boldsymbol{w}_i^T \Delta\phi(s) \geq 0$ since $\lambda_i > 0$, which implies $\hat{a}_{\pi_i} = 1$. Therefore we have

$$\begin{aligned} & p(a_s^* = -1, \tilde{a}_{\pi_i,s} = 1, \hat{a}_{\pi_i,s} = 1) \\ = \; & p(\tilde{a}_{\pi_i,s} = 1 | a_s^* = -1, \hat{a}_{\pi_i,s} = 1) p(\hat{a}\pi_i, s = 1) p(a_s^* = -1) \\ = \; & p\left(\lambda_i \geq \frac{\Delta L(s)}{\boldsymbol{w}_i^T \Delta\phi(s)}\right) \cdot p(\boldsymbol{w}_i^T \Delta\phi(s) \geq 0) \cdot p(\Delta L(s) > 0) \\ \leq \; & p\left(\lambda_i \geq \frac{\epsilon}{2RC}\right) \cdot 1 \cdot 1 \;=\; p\left(\lambda_i \geq \frac{\epsilon}{2RC}\right) \end{aligned}$$

Let n_λ be the largest $n < N$ such that $\lambda_i \geq \dfrac{\epsilon}{2RC}$, we have

$$\sum_{i=1}^{N} \mathbb{E}_{s\sim d_{\pi_i}}\left[\left(\ell_i(s,\pi_i,-1) - \ell_i(s,\pi_i,1)\right)\mathbf{1}_{\{s:\,\tilde{a}_{\pi_i,s}=1, a_s^*=-1\}}\right] \leq \ell_{\max} n_\lambda \tag{A.7}$$

For example, let λ_i decrease exponentially, e.g., $\lambda_i = \lambda_0 e^{-i}$. If $\lambda_0 < \dfrac{\epsilon e^N}{2RC}$, Then

$n_\lambda = \lceil \log \frac{2\lambda_0 RC}{\epsilon} \rceil$.

Case 2 $\tilde{a}_{\pi_i,s} = -1$ and $a^*_s = 1$. This is symmetrical to Case 1. Similar arguments yield the same bound.

In the online learning setting, imitating the coach is to obsearve the loss $\tilde{\ell}_i(\pi_i)$ and learn a policy $\pi_{i+1} = \arg\min_{\pi \in \Pi} \sum_{j=1}^{i} \tilde{\ell}_j(\pi)$ at iteration i. This is indeed equivalent to *Follow-The-Leader* except that we replaced the loss function. Thus Theorem 6 gives the bound of the second term.

Equation A.5 is then bounded by $2\ell_{\max} n_\lambda + \frac{2nb^2}{m}\left[\log\left(\frac{DRmN}{b}\right) + 1\right]$. □

Now we can prove Theorem 4. Consider the best policy in $\pi_{1:N}$, we have

$$\begin{aligned}
\min_{\pi \in \pi_{1:N}} \mathbb{E}_{s\sim d_\pi}[\ell(s, \pi, \pi^*(s))] &\leq \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}_{s\sim d_{\pi_i}}[\ell(s, \pi_i, \pi^*(s))] \\
&\leq \tilde{\epsilon}_N + \frac{2\ell_{\max} n_\lambda}{N} + \frac{2nb^2}{mN}\left[\log\left(\frac{DRmN}{b}\right) + 1\right]
\end{aligned}$$

When N is $O(T \log T)$, the regret is $O(1/T)$. Applying Theorem 2 completes the proof.

Bibliography

Raja Al-Khanji, Said El-Shiyab, and Riyadh Hussein. 2000. On the use of compensatory strategies in simultaneous interpretation. *Journal des Traducteurs*, 45(3):548–577.

Peter Auer, Nicolò Cesa-Bianchi, and Yoav Freund. 2002. The nonstochastic multiarmed bandit problem. *SIAM Journal of Computing*, 32(1):48–77.

Srinivas Bangalore, Vivek Kumar Rangarajan Sridhar, Prakash Kolan, Ladan Golipour, and Aura Jimene. 2009. Real-time incremental speech-to-speech translation of dialogs. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Nolan Bard, Michael Johanson, Neil Burch, and Michael Bowling. 2013. Online implicit agent modelling. In *Proceedings of International Conference on Autonomous Agents and Multiagent Systems*.

Richard E. Bellman. 1957. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684.

Djalel Benbouzid, Róbert Busa-Fekete, and Balázs Kégl. 2012. Fast classification using sparse decision DAGs. In *Proceedings of the International Conference of Machine Learning*.

Claudio Bendazzoli and Annalisa Sandrelli. 2005. An approach to corpus-based interpreting studies: Developing EPIC (european parliament interpreting corpus). In *Proceedings of Challenges of Multidimensional Translation*.

Alina Beygelzimer, John Langford, and Pradeep Ravikumar. 2009. Error-correcting tournaments. *Algorithmic Learning Theory*, 5809:247–262.

Alina Beygelzimer, John Langford, Lihong Li, Lev Reyzin, and Robert E. Schapire. 2011. Contextual bandit algorithms with supervised learning guarantees. In *Proceedings of Artificial Intelligence and Statistics (AISTATS)*.

Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. 1998a. Opponent modeling in poker. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. 1998b. Opponent modeling in poker. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media.

Arianna Bisazza, Nick Ruiz, and Marcello Federico. 2011. Fill-up versus interpolation methods for phrase-based SMT adaptation. In *Proceedings of International Workshop on Spoken Language Translation (IWSLT)*.

Nathan Bodenstab. 2012. *Prioritization and Pruning: Efficient Inference with Weighted Context-Free Grammars*. Ph.D. thesis, Oregon Health and Science University, August.

Bernd Bohnet. 2010. Very high accuracy and fast dependency parsing is not a contradiction. In *Proceedings of International Conference on Computational Linguistics*.

Jordan Boyd-Graber, Brianna Satinoff, He He, and Hal Daumé III. 2012. Besting the quiz master: Crowdsourcing incremental classification games. In *Empirical Methods in Natural Language Processing*.

S. Buchholz and E. Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Conference on Computational Natural Language Learning*.

Erik Camayd-Freixas. 2011. Cognitive theory of simultaneous interpreting and training. In *Proceedings of the 52nd Conference of the American Translators Association*.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *Proceedings of Empirical Methods in Natural Language Processing*.

Raman Chandrasekar, Christine Doran, and Srinivas Bangalore. 1996. Motivations and methods for text simplification. In *Proceedings of International Conference on Computational Linguistics*.

Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Kai-Wei Chang, Hal Daumé III, John Langford, and Stéphane Ross. 2014. Efficient programmable learning to search. In *arXiv*.

Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, Hal Daumé III, and John Langford. 2015. Learning to search better than your teacher. In *Proceedings of the International Conference of Machine Learning*.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-fine PCFG parsing. In *Conference of the North American Chapter of the Association for Computational Linguistics*.

Minmin Chen, Kilian Q Weinberger, Olivier Chapelle, Dor Kedem, and Zhixiang Xu. 2012. Classifier cascade for minimizing feature evaluation cost. In *Proceedings of Artificial Intelligence and Statistics (AISTATS)*.

Colin Cherry and George Foster. 2012. Batch tuning strategies for statistical machine translation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

D. Chiang, Y. Marton, and P. Resnik. 2008. Online large-margin training of syntactic and structural translation features. In *Proceedings of Empirical Methods in Natural Language Processing*.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder–decoder approaches. *arXiv:1409.1259*.

Y. J. Chu and T. H. Liu. 1965. On the shortest arborescence of a directed graph. *Science Sinica*, 14.

Adam Coates, Pieter Abbeel, and Andrew Y. Ng. 2008. Learning for control from multiple demonstrations. In *Proceedings of the International Conference of Machine Learning*.

Michael Collins, Philipp Koehn, and Ivona Kučerová. 2005. Clause restructuring for statistical machine translation. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*.

Brian Collins. 2007. Combining opponent modeling and model-based reinforcement learning in a two-player competitive game. Master's thesis, School of Informatics, University of Edinburgh.

Koby Crammer and Yoram Singer. 2003. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585.

Dipanjan Das, Desai Chen, André F. T. Martins, Nathan Schneider, , and Noah A. Smith. 2014. Frame-semantic parsing. *Computational Linguistics*, 40(1).

Hal Daumé III, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine Learning Journal (MLJ)*.

Hal Daumé III. 2004. Notes on cg and lm-bfgs optimization of logistic regression. Software available at http://www.cs.utah.edu/~hal/megam/.

Aaron Davidson. 1999. Using artifical neural networks to model opponents in texas hold'em. CMPUT 499 - Research Project Review.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*.

Gabriel Dulac-Arnold. 2014. *A General Sequential Model for Constrained Classification*. Ph.D. thesis, Université Pierre & Marie Curie.

J. Edmonds. 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B(4):233–240.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: an exploration. In *Proceedings of International Conference on Computational Linguistics*.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. 2010. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3).

Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. 2016. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *Arxiv:1602.02672*.

Christian Fügen, Alex Waibel, , and Muntsin Kolss. 2007. Simultaneous translation of lectures and speeches. *Machine Translation*, 21(4):209–252.

Tomoki Fujita, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2013. Simple, lexicalized choice of translation timing for simultaneous speech translation. In *Proceedings of Interspeech*.

Francesca Gaiba. 1998. *The origins of simultaneous interpretation: The Nuremberg Trial*. University of Ottawa Press.

Michel Galley and Christopher D. Manning. 2008. A simple and effective hierarchical phrase reordering model. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*.

Sam Ganzfried and Tuomas Sandholm. 2011. Game theory-based opponent modeling in large imperfect-information games. In *Proceedings of International Conference on Autonomous Agents and Multiagent Systems*.

Tianshi Gao and Daphne Koller. 2011. Active classification based on value of classifier. In *Proceedings of Advances in Neural Information Processing Systems*.

Dan Gillick and Benoit Favre. 2009. A scalable global model for summarization. In *Proceedings of the Workshop on Integer Linear Programming for Natural Language Processing*.

Alvin Grissom II, He He, John Morgan, Jordan Boyd-Graber, and Hal Daum'e III. 2014. Don't until the final verb wait: Reinforcement learning for simultaneous machine translation. In *Proceedings of Empirical Methods in Natural Language Processing*.

Alexander Grubb and J. Andrew Bagnell. 2012. Speedboost: Anytime prediction with uniform near-optimality. In *Proceedings of Artificial Intelligence and Statistics (AISTATS)*.

Matthew Hausknecht and Peter Stone. 2015. Deep recurrent q-learning for partially observable MDPs. *Arxiv:1507.06527*.

Elad Hazan, Adam Kalai, Satyen Kale, and Amit Agarwal. 2006. Logarithmic regret algorithms for online convex optimization. In *Proceedings of Conference on Learning Theory*, pages 499–513.

He He, Hal Daum'e III, and Jason Eisner. 2012. Imitation learning by coaching. In *Neural Information Processing Systems (NIPS)*.

He He, Hal Daum'e III, and Jason Eisner. 2013. Dynamic feature selection for dependency parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*.

He He, Alvin Grissom II, John Morgan, Jordan Boyd-Graber, and Hal Daum'e III. 2015. Syntax-based rewriting for simultaneous machine translation. In *Proceedings of Empirical Methods in Natural Language Processing*.

He He, Jordan Boyd-Graber, and Hal Daum'e III. 2016. Opponent modeling in deep reinforcement learning. In *Proceedings of the International Conference of Machine Learning*.

Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. 2013. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Hans G. Hönig. 1997. Using text mappings in teaching consecutive interpreting. *Translation and Translation Theory*, pages 19–34.

Ronald A. Howard. 1960. *Dynamic Programming and Markov Processes*. The MIT Press, 1st edition, June.

Hideki Isozaki, Tsutomu Hirao, Kevin Duh, Katsuhito Sudoh, and Hajime Tsukada. 2010. Automatic evaluation of translation quality for distant language pairs. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*.

Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. 1994. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6).

Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. 1991. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87.

Ken Jennings. 2006. *Brainiac: adventures in the curious, competitive, compulsive world of trivia buffs*. Villard.

Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing*. Prentice Hall, Upper Saddle River, NJ.

Matti Kääriäinen. 2006. Lower bounds for reductions. Talk at the Atomic Learning Workshop (TTI-C), March.

Sham M. Kakade and Shai Shalev-shwartz. 2008. Mind the duality gap: Logarithmic regret algorithms for online optimization. In *Proceedings of Advances in Neural Information Processing Systems*.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for n-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*. IEEE.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Brooke Cowan Nicola Bertoldi, Wade Shen, Richard Zens Christine Moran, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbs. 2003. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*.

Philipp Koehn. 2000. German-english parallel corpus "de-news".

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Matt J Kusner, Wenlin Chen, Quan Zhou, Zhixiang Eddie Xu, Kilian Q Weinberger, and Yixin Chen. 2014. Feature-cost sensitive learning with submodular trees of classifiers. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1939–1945.

Michail G. Lagoudakis and Ronald Parr. 2003. Reinforcement learning as classification: Leveraging modern classifiers. In *Proceedings of the International Conference of Machine Learning*.

John Langford and Alina Beygelzimer. 2005. Sensitive error correcting output codes. In *Proceedings of Conference on Learning Theory*.

John Langford, Lihong Li, and Alex Strehl. 2007. Vowpal Wabbit online learning project. http://hunch.net/~vw.

Alessandro Lazaric, Mohmammad Ghavamzadeh, and Rémi Munos. 2010. Analysis of a classification-based policy iteration algorithm. In *Proceedings of the International Conference of Machine Learning*.

Michael L. Littman. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the International Conference of Machine Learning*.

Alan J. Lockett, Charles L. Chen, and Risto Miikkulainen. 2007. Evolving explicit opponent models in game playing. In *Proceeedings of the Genetic and Evolutionary Computation Conference*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

André F. T. Martins, Noah A. Smith, and Eric P. Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Shigeki Matsubara, Keiichi Iwashima, Nobuo Kawaguchi, Katsuhiko Toyama, and Yasuyoshi Inagaki. 2000. Simultaneous Japanese-English interpretation based on early predition of English verb. In *Symposium on Natural Language Processing*.

Shigeki Matsubara, Akira Takagi, Nobuo Kawaguchi, and Yasuyoshi Inagaki. 2002. Bilingual spoken monologue corpus for simultaneous machine interpretation research. In *Proceedings of the Language Resources and Evaluation Conference (LREC)*.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of the European Chapter of the Association for Computational Linguistics*, pages 81–88.

Ryan McDonald, K. Crammer, and Fernando Pereira. 2005a. Online large-margin training of dependency parsers. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Empirical Methods in Natural Language Processing*.

Ryan McDonald. 2007. *A Study of Global Inference Algorithms in Multi-document Summarization*. Springer.

Brendan McMahan and Matthew Streeter. 2009. Tighter bounds for multi-armed bandits with expert advice. In *Proceedings of Conference on Learning Theory*.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent models of visual attention. In *Proceedings of Advances in Neural Information Processing Systems*.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02.

Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. 2015. Language understanding for text-based games using deep reinforcement learning. In *Proceedings of Empirical Methods in Natural Language Processing*.

Graham Neubig. 2011. The Kyoto free translation task. *Available online at http://www. phontron. com/kftt*.

Franz Josef Och and Hermann Ney. 2003. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51.

Yusuke Oda, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2014. Optimizing segmentation strategies for simultaneous speech translation. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*.

Yusuke Oda, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Syntax-based simultaneous translation through prediction of unseen syntactic constituents. In *The 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, Beijing, China, July.

Kishore Papineni, Salim Roukos, Todd Ward, John Henderson, and Florence Reeder. 2002a. Corpus-based comprehensive and diagnostic MT evaluation: Initial arabic, chinese, french, and spanish results. In *Proceedings of Human Language Technology*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. 2002b. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Matthias Paulik and Alex Waibel. 2009. Automatic translation from parallel speech: Simultaneous interpretation as mt training data. In *Proceedings of the Automatic Speech Recognition and Understanding*.

Matthias Paulik and Alex Waibel. 2010. Spoken language translation from parallel speech audio: Simultaneous interpretation as slt training data. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Timothy Hunter Pieter Abbeel, Adam Coates and Andrew Y. Ng. 2008. Autonomous autorotation of an rc helicopter. In *11th International Symposium on Experimental Robotics*.

Brock Pytlik and David Yarowsky. 2006. Machine translation for languages lacking bitext via multilingual gloss transduction. In *5th Conference of the Association for Machine Translation in the Americas (AMTA)*, August.

Uwe Quasthoff, Matthias Richter, and Christian Biemann. 2006. Corpus portal for search in monolingual corpora. In *International Language Resources and Evaluation*, pages 1799–1802.

Nathan D Ratliff, J Andrew Bagnell, and Martin A Zinkevich. 2006. Maximum margin planning. In *Proceedings of the International Conference of Machine Learning*.

Lev Reyzin. 2011. Boosting on a budget: Sampling for feature-efficient prediction. In *Proceedings of the International Conference of Machine Learning*.

Mark Richards and Eyal Amir. 2007. Opponent modeling in Scrabble. In *International Joint Conference on Artificial Intelligence*.

B. Roark and K. Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of International Conference on Computational Linguistics*.

Stéphane Ross and J. Andrew Bagnell. 2010. Efficient reductions for imitation learning. In *Proceedings of Artificial Intelligence and Statistics (AISTATS)*.

Stéphane Ross and Andrew Bagnell. 2014. Reinforcement and imitation learning via interactive no-regret learning. *arXiv:1406.5979*.

Stéphane Ross, Geoff J. Gordon, and J. Andrew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of Artificial Intelligence and Statistics (AISTATS)*.

Stéphane Ross. 2013. *Interactive Learning for Sequential Decisions and Predictions*. Ph.D. thesis, Carnegie Mellon University.

Dan Roth and Wen tau Yih. 2005. Integer linear programming inference for conditional random fields. In *Proceedings of the International Conference of Machine Learning*, pages 737–744.

Jonathan Rubin and Ian Watson. 2011. On combining decisions from multiple expert imitators for performance. In *International Joint Conference on Artificial Intelligence*.

Alexander Rush and Slav Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Conference of the North American Chapter of the Association for Computational Linguistics*.

Frederik Schadd, Er Bakkes, and Pieter Spronck. 2007. Opponent modeling in real-time strategy games. In *Proceedings of the GAME-ON 2007*, pages 61–68.

Hiroaki Shimizu, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2013. Constructing a speech translation system using simultaneous interpretation data. In *Proceedings of International Workshop on Spoken Language Translation (IWSLT)*.

Hiroaki Shimizu, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2014. Collection of a simultaneous translation corpus for comparative analysis. In *Proceedings of the Language Resources and Evaluation Conference (LREC)*.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. 2005. Bayes' bluff: Opponent modelling in poker. In *Proceedings of Uncertainty in Artificial Intelligence*.

Vivek Kumar Rangarajan Sridhar, John Chen, Srinivas Bangalore Andrej Ljolje, and Rathinavelu Chengalvarayan. 2013. Segmentation strategies for streaming speech translation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*, volume 1. MIT Press Cambridge.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of Advances in Neural Information Processing Systems*.

Umar Syed and Robert E. Schapire. 2011. A reduction from apprenticeship learning to classification. In *Proceedings of Advances in Neural Information Processing Systems*.

R. E. Tarjan. 1977. Finding optimum branchings. *Networks*, 7(1):25–35.

Christoph Tillmann, Stephan Vogel, Hermann Ney, and Alex Zubiaga. 1997. A dp-based search using monotone alignments in statistical translation. In *Proceedings of International Conference on Computational Linguistics*.

Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Conference of the North American Chapter of the Association for Computational Linguistics*.

John N. Tsitsiklis. 1994. Asynchronous stochastic approximation and q-learning. *Machine Learning*, 16(3).

William Uther and Manuela Veloso. 2003. Adversarial reinforcement learning. Technical Report CMU-CS-03-107, School of Computer Science, Carnegie Mellon University.

Masao Utiyama and Hitoshi Isahara. 2003. Reliable measures for aligning japanese-english news articles and sentences. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*.

Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning*, 8(3-4):279–292.

David Weiss and Ben Taskar. 2010. Structured prediction cascades. In *Proceedings of Artificial Intelligence and Statistics (AISTATS)*.

David Weiss, Ben Sapp, and Ben Taskar. 2013. Dynamic structured model selection. In *International Conference on Computer Vision*.

Peng Xu, Jaeho Kang, Michael Ringgaard, , and Franz Och. 2009. Using a dependency parser to improve SMT for subject-object-verb language. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Zhixiang Xu, Matt J Kusner, Kilian Q Weinberger, and Minmin Chen. 2013. Cost-sensitive tree of classifiers. In *Proceedings of the International Conference of Machine Learning*.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with Support Vector Machines. In *Proceedings of IWPT*.

Steve Young. 2006. Using POMDPs for dialog management. In *Proceedings of the Workshop on Spoken Language Technologies*.

Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. 2016. Learning deep neural network policies with continuous memory states. In *Proceedings of the IEEE International Conference on Robotics and Automation*.