NORTHWESTERN UNIVERSITY

Integrating Machine Learning and Symbolic Reasoning: Learning to Generate Symbolic Representations from Weak Supervision

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Chen Liang

EVANSTON, ILLINOIS

December 2018

ABSTRACT

Integrating Machine Learning and Symbolic Reasoning:
Learning to Generate Symbolic Representations
from Weak Supervision

Chen Liang

Machine learning and symbolic reasoning have been two main approaches to build intelligent systems. Symbolic reasoning has been used in many applications by making use of expressive symbolic representations to encode prior knowledge, conduct complex reasoning and provide explanations. Recently, machine learning has enabled various successful applications by learning from large amount of noisy data. In this thesis, I propose to integrate these two approaches to build more expressive, efficient and interpretable learning systems. The main idea is, instead of training a model to predict the output directly, training a model to generate symbolic representations and then predict the output based on the generated symbolic representations. Incorporating symbolic representations into machine learning helps the model conduct complex reasoning, leverage external knowledge sources and learn more efficiently with better inductive biases. The main challenge is that the symbolic representations are usually hard to collect because it requires expertise, so we propose to induce them from weak supervision, which is much easier to collect. We analyze the challenges when learning from

weak supervision and propose several novel techniques in reinforcement learning and latent structured prediction to overcome the problems. The proposed approach is investigated in two settings. In the first setting, to estimate the similarity between two relational structures, we use the structural alignment between them as the symbolic representation, which is then fed into a classifier to estimate their similarity. Experiments have shown that, with the inductive bias from structural alignment, the learned model achieves results competitive to state-of-the-art on paraphrase identification and knowledge base completion benchmarks while being much simpler or using orders of magnitude less data. In the second setting, we use compositional programs as the symbolic representations, which can be executed against a knowledge base or database tables to answer open-domain questions. By generating programs, the model can leverage existing knowledge and operations to perform complex reasoning compositionally. To our knowledge, this is the first end-to-end model without feature engineering that significantly outperforms previous state-of-the-art results on two very competitive semantic parsing benchmarks. Besides, I will also show that the generated symbolic representations, e.g., the structural alignment and the programs, can be inspected and verified by the users, which makes the model more interpretable and easier to debug.

Acknowledgments

This thesis is not possible without the help from my advisor, Ken Forbus. Ken has been an amazing advisor throughout my PhD career. Thank you for bringing me into the area of AI and providing countless suggestions on my experiments, (often last minute) drafts, and presentations. Your support and guidance gave me the strength to pursue my research goal, and your dedication to AI research has always been an inspiration for me.

This thesis is a result of close collaboration with Ni Lao. Thanks for mentoring me during my Google internships and for collaborating on several projects. Among many other things, you taught me how to be bold and pragmatic at the same time. I enjoyed our discussions on research and life. I really appreciate the chance to also collaborate with Quoc Le, Mohammad Norouzi, and Jonathan Berant on these exciting projects, which made me a much better researcher.

I feel really grateful for having Doug Downey in my committee. The collaboration with you and your lab provided me the opportunity to obtain a firm understanding of machine learning and NLP.

I also hope to thank Dedre Gentner for showing me how intriguing and exciting cognitive science is, Chris Riesbeck for teaching me solid skills on AI programming, and Larry Birnbaum for giving me the first lesson on entrepreneurship.

I would like to thank Karl Hermann, Wang Ling, and the rest of the language team in DeepMind, as well as Praveen Paritosh, Ka Wong, Vinodh Kumar, and the rest of the

Queriosity team in Google Research for having me for internship. I have learned a lot from you and also had a lot of fun working together.

Many thanks to Thanapon Noraset, Subu Kandaswamy and Chandra Bhagavatula for the fun conversations about AI and PhD life, and for spending the late nights working for paper deadlines together. Without you, my PhD life would be much less fun. I also would like to thank my lab mates, Maria Chang, Joe Blass, CJ McFate, Matt McLure, Tom Hinrichs, Madeline Usher for all the help and all the fun.

Finally, I want to thank my wife, Chenchen Pan, for always supporting me and being with me through the best and worst times, and my parents, Yuxin Liang and Jingti Chen for the love, support and encouragement throughout my life.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Overview

Learning and reasoning are fundamental elements of intelligence. Machine learning and symbolic reasoning have been two main approaches to build intelligent systems [**114**]. Recently, machine learning has enabled various successful applications by using statistical models, such as deep neural networks (DNN) [**67**] and support vector machines (SVM) [**23**], to learn from large amounts of data. Symbolic reasoning, on the other hand, uses expressive symbolic representations to encode prior knowledge, conduct complex reasoning and provide explanations [**92, 16, 37**]. In this thesis, I propose to integrate these two approaches by learning to generate symbolic representations from weak supervision.

Machine learning aims to learn a mapping from input x to output y: $y = f(x)$. The common approach is to represent the function f as a deep neural network or SVM with feature vectors and then learn the parameters from data, which has been successful in many applications. To extend this success to more high-level tasks, for example, in natural language understanding, two problems need to be solved: (1) if the mapping from x to y requires complex reasoning and prior knowledge, it can be hard or inefficient to represent using neural networks or SVMs with feature vectors; (2) without a good inductive bias, training these models successfully often requires large amounts of data. Symbolic representations can be used to address these problems.

To incorporate symbolic representations into machine learning methods, we introduce the symbolic representation as a latent variance z, which is usually a structured object (e.g., a program) defined by a grammar G. Instead of learning a direct mapping from input x to output y, the proposed approach first learns to generate the latent symbolic representation z as $z = h(x)$ and then predicts the final output using both x and z as $y = g(z, x) = g(h(x), x)$. This helps address the aforementioned problems: (1) the symbolic representations z can be used to leverage prior knowledge and conduct complex reasoning, which enables more expressive models. For example, it is hard to train a neural network that maps questions about the Olympics like "how many more gold medals did USA won than the Great Britain in 2008 Olympics?" directly to its answer "7", because the model needs to acquire the knowledge about the Olympics and at the same time learn arithmetic reasoning, which is hard and inefficient to represent and learn using a neural network. But if the model is augmented with a knowledge base (KB) of Olympics and arithmetic functions, it is easier and more generalizable to learn to generate a program z that queries the knowledge base and computes the answer. In this case, the symbolic representation (the executable program) is used to represent both the query to KB and the arithmetic reasoning process. (2) we can introduce effective inductive bias through the symbolic representations z to learn with less data. For example, when learning to estimate similarity between two relational structures, if we let the model generate alignment z between the two structures first and use this alignment to make final prediction, the model would pay more attention to the matches and mismatches between the two rather than just looking at each one independently. This inductive bias helps it avoid overfitting to specific inputs and generalize better. In addition, the symbolic representation z can also be inspected by human to understand the model and verify its behavior. For example, we could look at the generated alignment to see what matches or mismatches make

the model draw the final conclusion. We can also check the generated program to see the reasoning process and the knowledge used by the model to answer a certain question. This way, it makes the model more interpretable and easier to debug.

In the proposed approach, the symbolic representation is latent and is inferred from weak supervision. In other words, instead of learning from data annotated with the symbolic representations $\{(x_i, z_i, y_i)\}$, the proposed approach learns from only input-output pairs $\{(x_i, y_i)\}$ and induces the intermediate symbolic representations z_i. Learning from weak supervision is preferred because it is usually expensive and slow to annotate the correct symbolic representations z for large amount of data, but it is relatively easy to collect input-output pairs or feedback from downstream applications as weak supervision. For example, expertise is needed to annotate each question with the reasoning process and required prior knowledge in a formal language, but it is relatively simple and natural to let a human answer a question or evaluate the quality of an answer.

1.2. Outline and Contributions

The thesis is organized as follows:

Chapter 2: Background Chapter 2 covers the machine learning methods that the proposed approach is built upon. We first review the structured perceptron algorithm to train log-linear models for structured prediction. We show how it can be extended to work with latent variables, which is adapted and applied to text similarity in Chapter 3. Then we review the policy gradient methods in reinforcement learning and show how it can be applied to train recurrent neural networks for weakly-supervised sequence generation. In Chapter 4, we propose novel policy gradient methods to enable efficient and robust deep reinforcement learning in large search space with sparse rewards and applied it to semantic parsing.

Chapter 3: Learning Similarity Estimation with Structural Alignment This chapter applies the proposed approach to estimate similarity between relational structures. The input x are two relational structures and the output y is the similarity score. Inspired by the structure-mapping theory of human similarity judgment, we use structural alignment as the intermediate symbolic representation z. We evaluate this approach in two tasks. The first task is knowledge base completion or link prediction. To decide whether two entities e_1 and e_2 has a certain relation r or not, we compare the relational structures S_1 between e_1 and e_2 with a prototypical structure S_r of entities with the relation r and use their similarity to make the prediction. The second task is text similarity. To estimate how similar two sentences are, we compare the attributed relational graphs constructed using the linguistic annotations of the sentences and compute their similarity. In both tasks, by generating the structural alignment z, the model benefits from a inductive bias that focuses on the matches and mismatches between the two structures and considers both structural similarity and local similarity. The module that generates the structural alignment and the module that estimates the similarity are learned jointly by tying parameters or through an iterative training process of a latent structured prediction model. As evaluation, the proposed approach is shown to achieve state-of-the-art results while being much simpler or uses much less data on a knowledge base completion and a paraphrase identification benchmark. The generated alignments can be used to justify the similarity score.

Chapter 4: Learning to Generate Programs from Natural Language This chapter applies the proposed approach to semantic parsing and question answering. The input x is a natural language question and the output y is its answer. We use compositional programs as the symbolic representation z, which can be executed against a knowledge base or database tables to compute the answer. By generating programs, the model can query a KB for existing

knowledge and use external functions compositionally to perform complex reasoning. We apply this approach to integrate deep neural networks with symbolic reasoning and propose the Neural Symbolic Machines (NSM) framework. By using a neural sequence-to-sequence model as the "neural programmer" and a Lisp interpreter as the "symbolic computer", the Neural Symbolic Machines learns to generate programs from weak supervision (a reward signal) indicating if the desired goal is achieved (for example, if the correct answer is returned). Reinforcement learning [**133**] is applied to learn from weak supervision (question-answer pairs). To train NSM successfully, we propose novel techniques to improve the efficiency and robustness of the policy gradient methods in large search space with sparse rewards. We evaluate this framework on two competitive semantic parsing tasks. The first one is to answer open-domain questions using Freebase [**12**]; the second one is to answer open-domain questions using tables from Wikipedia [**98**]. Both of the datasets have been used intensively to benchmark different methods. To our knowledge, NSM is the first end-to-end model without feature engineering that significantly outperforms previous state-of-the-art results on both benchmarks [**70**]. Besides, the generated programs can be inspected and verified by the users, which makes the model more interpretable and easier to debug.

Conclusion and Future Directions This chapter summarizes the proposed approach and the experiments, and discusses future directions.

CHAPTER 2

Background

When applying the proposed approach to a task, two questions are: (1) what symbolic representation should be used for the task; (2) what machine learning method should be used to learn to generate structured objects (the symbolic representations) from weak supervision. This thesis builds upon previous literature that investigated these two questions. We leave the background on the symbolic representations to the following chapters when we apply the proposed approach to specific tasks. In this chapter, we focus on reviewing the types of machine learning methods our work builds upon.

The proposed approach turns the problem of learning $y = f(x)$ into learning $z = h(x)$ and $y = g(z, x)$. z is the symbolic representation, so learning h is a structured prediction problem. When the gold symbolic representation is not available, and both h and g need to be learned from the end task, it is considered a weakly supervised structured prediction problem. To solve this problem, we investigate structured prediction methods with log-linear models and reinforcement learning with neural networks and propose new techniques to improve them. We will briefly review both topics in this chapter.

2.1. Structured Prediction with Structured Perceptron

Structured prediction refers to a set of machine learning methods that predicts structured output such as a sequence, rather than a scalar of continuous or discrete value as in regression or classification. It is widely applied in natural language processing [**1**] and computer vision [**96**]. Given the structure in the symbolic representations, learning to generate symbolic representations can be naturally viewed as a structured prediction problem. A lot of methods have been proposed for structured prediction. In this section, we focus on a simple approach,

log-linear models trained with structured perceptron algorithm, which is fast, effective and can be extended to include latent variables.

2.1.1. Log-Linear Models

Log-linear models assign the joint probability of input x and output y as:

$$P_\theta(x,y) = \frac{\exp \theta \cdot \Phi(x,y)}{\sum_{x',y'} \exp \theta \cdot \Phi(x',y')} = \frac{\exp \theta \cdot \Phi(x,y)}{Z(\theta)} \tag{2.1}$$

Note that the input x and the output y can be arbitrary structures. θ is a $\mathbb{R}^d$ vector of feature weights that parameterizes the model. Φ is a function that maps pairs of input and output (x,y) to a $\mathbb{R}^d$ feature vector. $Z(\theta)$ is the partition function to normalize the probability to $[0,1]$. The conditional probability of the output given a certain input is represented as

$$P_\theta(y|x) = \frac{\exp \theta \cdot \Phi(x,y)}{\sum_{y'} \exp \theta \cdot \Phi(x,y')} \tag{2.2}$$

The inference or decoding, which generates an output for a given input, is done through

$$y = \underset{y' \in \mathcal{Y}}{\operatorname{argmax}} P_\theta(y|x) = \underset{y' \in \mathcal{Y}}{\operatorname{argmax}} \theta \cdot \Phi(x,y') \tag{2.3}$$

$\mathcal{Y}$ is the space of all possible values for y, which is usually exponentially large. When $\Phi(x,y)$ can be decomposed, the argmax might be solved efficiently through dynamic programming. In general, it needs to be approximated by beam search or greedy decoding. Given a dataset $\{(x_i,y_i)\}$, the conditional log-likelihood objective can be written as

$$\begin{aligned} J(\theta) =& \frac{1}{N} \log \prod_i P_\theta(y_i|x_i) \\ =& \frac{1}{N} \sum_i \log P_\theta(y_i|x_i) \\ =& \frac{1}{N} \sum_i (\Phi(x_i,y_i) \cdot \theta - \log Z(\theta, x_i)) \end{aligned} \tag{2.4}$$

in which

$$Z(\theta, x_i)) = \sum_{y \in \mathcal{Y}} \exp(\Phi(x_i, y) \cdot \theta) \tag{2.5}$$

So the gradient of the log-likelihood w.r.t the parameters θ is

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_i (\Phi(x_i, y_i) - \mathbb{E}_{P_\theta(y|x_i)}[\Phi(x_i, y)]) \tag{2.6}$$

If we introduce the latent variable z into the model, the joint probability becomes

$$P_\theta(x, z, y) = \frac{\exp \theta \cdot \Phi(x, z, y)}{\sum_{x,z,y} \exp \theta \cdot \Phi(x, z, y)} \tag{2.7}$$

The conditional log-likelihood of the output given input needs to marginalize over the latent variable z.

$$P_\theta(y|x) = \frac{\sum_{z'} \exp \theta \cdot \Phi(x, z^{'}, y)}{\sum_{z',y'} \exp \theta \cdot \Phi(x, z', y^{'})} \tag{2.8}$$

Then the log-likelihood objective becomes

$$\begin{aligned} J(\theta) =& \frac{1}{N} \log \prod_i P_\theta(y_i|x_i) \\ =& \frac{1}{N} \sum_i \log P_\theta(y_i|x_i) \\ =& \frac{1}{N} \sum_i (\log Z(\theta, x_i, y_i) - \log Z(\theta, x_i)) \end{aligned} \tag{2.9}$$

in which

$$\begin{aligned} Z(\theta, x_i) =& \sum_{y,z} \exp(\Phi(x_i, z, y) \cdot \theta) \\ Z(\theta, x_i, y_i)) =& \sum_z \exp(\Phi(x_i, z, y_i) \cdot \theta) \end{aligned} \tag{2.10}$$

And the gradient of the objective becomes

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_i (\mathbb{E}_{P_\theta(z|x_i,y_i)}[\Phi(x_i, z, y_i)] - \mathbb{E}_{P_\theta(z,y|x_i)}[\Phi(x_i, z, y)]) \tag{2.11}$$

We can use gradient-based methods to train the model, but computation of the gradient requires computing the expectations. In some special case, the expectations can be computed efficiently using dynamic programming, but in general it needs to be approximated because the space of the structured objects are usually exponentially large. In the next section, we review a fast and effective training algorithm, which can viewed as an approximation to the stochastic gradient descent training.

2.1.2. Structured Perceptron Algorithm

There are several methods to train a log-linear model. We focus on the structured perceptron algorithm [**21, 22**], which is simple and fast, and achieves good results in a lot of NLP applications such as POS tagging, parsing, machine translation, etc.

The algorithm is shown in Algorithm 1. It adapts the perceptron algorithm for binary classification to structured outputs. To generalize better, the average of the weights from different training steps are usually used as the final parameters [**21**].

Algorithm 1 Structured Perceptron

Input: training data $\mathbb{D} = \{x_i, y_i\}$, maximum number of iterations T
Initialize: parameters $\theta_1 = 0$
Procedure:
for t in $1...T$, i in $1...N$ **do**
  $y_i^{'} = \text{argmax}_{y \in \mathcal{Y}} \theta \cdot \Phi(x_i, y)$
  if $y_i^{'} \neq y_i$ **then**
    $\theta \leftarrow \theta + \Phi(x_i, y_i) - \Phi(x_i, y_i^{'})$
Output: parameters θ
Inference: $y = h_\theta(x) = \text{argmax}_{y \in \mathcal{Y}} \theta \cdot \Phi(x, y)$

It is proven to converge when the examples are separable or inseparable by a small margin [**21**]. We can also view it as stochastic gradient descent training with an approximation of the gradient in Equation 2.11. It approximates the expectation using the value with the maximum probability.

$$\mathbb{E}_{P_\theta(y|x_i)}[\Phi(x_i, y)] \approx \Phi(x_i, y_i^{'}) \tag{2.12}$$

in which

$$y_i^{'} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} P_\theta(y|x_i) = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \theta \cdot \Phi(x_i, y) \tag{2.13}$$

2.1.3. Latent Structured Perceptron Algorithm

Following [**74, 130**], the structured perceptron algorithm can be adapted to include latent variables. Similarly to Equation 2.12, the gradient of J can be approximated as

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_i (\mathbb{E}_{P_\theta(z|y_i,x_i)}[\Phi(x_i, z, y_i)] - \mathbb{E}_{P_\theta(y,z|x_i)}[\Phi(x_i, z, y)]) \\ &\approx \sum_i (\Phi(x_i, z_i^*, y_i) - \Phi(x_i, z_i^{'}, y_i^{'})) \end{aligned} \tag{2.14}$$

in which

$$(y_i^{'}, z_i^{'}) = \underset{(y,z)}{\operatorname{argmax}} \theta \cdot \Phi(x_i, z, y) \tag{2.15}$$

$$z_i^* = \underset{z}{\operatorname{argmax}} \theta \cdot \Phi(x_i, z, y_i) \tag{2.16}$$

Applying stochastic gradient descent with the approximation in Equation 2.14, we can get the Latent Structured Perceptron in Algorithm 2.

Algorithm 2 Latent Structured Perceptron

Input: training data $\mathbb{D} = \{x_i, y_i\}$, maximum number of iterations T
Initialize: parameters $\theta = 0$
Procedure:
for t in $1...T$, i in $1...N$ **do**
  $(y_i^{'}, z_i^{'}) = \operatorname{argmax}_{y \in \mathcal{Y}, z \in \mathcal{Z}} \theta \cdot \Phi(x_i, z, y)$
  $z_i^* = \operatorname{argmax}_{z \in \mathcal{Z}} \theta \cdot \Phi(x_i, z, y_i)$
  if $(y_i^{'}, z_i^{'}) \neq (y_i, z_i^*)$ **then**
    $\theta \leftarrow \theta + \Phi(x_i, z_i^*, y_i) - \Phi(x_i, z_i^{'}, y_i^{'})$
Output: parameters θ
Inference: $y = f(x) = \operatorname{argmax}_{y' \in \mathcal{Y}} \max_{z' \in \mathcal{Z}} \theta \cdot \Phi(x, z^{'}, y^{'})$

The intuition of Algorithm 2 is to "hallucinate" a gold value z_i^* given (x_i, y_i) and use it to update the model. When learning to generate symbolic representations from weak supervision, the gold symbolic representation is also not given, so we need to model it as a latent variable z. In Chapter 3, we apply a similar iterative training process by "hallucinating"

the gold value for the latent variable z to jointly learn the parameters for structural alignment and similarity estimation.

2.2. Structured Prediction with Deep Reinforcement Learning

Deep reinforcement learning refers to reinforcement learning methods with function approximation using neural networks. It has been successfully applied in several applications such as Go [**124**], computer games [**83**], robotics [**68**], etc. By sequentializing the generation process of the structured objects (for example, the symbolic representations), we can turn a structured prediction problem into a sequence generation problem. Recurrent neural network is commonly used to define a generative model over variable-length sequences and has been the state-of-the-art in many sequence generation tasks. Using reinforcement learning, we can apply RNNs to the weakly supervised sequence generation / structured prediction problem.

2.2.1. Sequence Generation with Recurrent Neural Networks

Recurrent neural networks (RNN) are a type of neural networks for processing sequential data [**44**]. One typical use of RNN is to define a generative model over sequences, which can be used, for example, to generate text. It is shown to achieve state-of-the-art results on many sequence generation tasks such as language modeling, speech recognition, machine translation, etc.

RNNs can be used to define a probability distribution over possible sequences $(y_0...y_T)$. In the discrete case, y_i is a token from a vocabulary $\mathbf{V}$, which can be represented as a one-hot vector x_i, in which only one of dimensions has the value 1 and all others are 0. A RNN summarizes the history of a sequence in a hidden state h. Starting from a initial state h_0. The hidden state at each timestep is computed as

$$\begin{aligned}\mathbf{e_t} &= \mathbf{E_x x_{t-1}} \\ \mathbf{a_t} &= \mathbf{W_{hh} h_{t-1}} + \mathbf{W_{hx} e_t} + \mathbf{b_h} \\ \mathbf{h_t} &= \tanh(\mathbf{a_t})\end{aligned} \tag{2.17}$$

$\mathbf{x_t}$ is the input at step t. $\mathbf{E_x}$, $\mathbf{W_{hh}}$, $\mathbf{W_h x}$ and $\mathbf{b_h}$ are parameters of the RNN, which is shared among different timesteps. In the discrete case, the hidden state of the RNN can be used to compute a probability distribution over the next token y_t with softmax function

$$\begin{aligned} P_\theta(y_t|y_0, y_1, ..., y_T) =& P_\theta(y_t|\mathbf{x_0}, ..., \mathbf{x_t}) \\ =& P_\theta(y_t|\mathbf{h_t}) \\ =& f^*(y_t, h_t) \\ =& \frac{\exp(\mathbf{w_s^{y_t} h_t} + \mathbf{b_s^{y_t}})}{\sum_y \exp(\mathbf{w_s^y h_t} + \mathbf{b_s^y})} \end{aligned} \tag{2.18}$$

So the joint probability of a sequence of tokens $(y_0, y_1, ..., y_T)$ can be written as:

$$\begin{aligned} P_\theta(y_0, y_1, ..., y_T) =& \prod_{t=1}^{T} P_\theta(y_t|y_0, ..., y_{t-1}) \\ =& \prod_{t=1}^{T} P_\theta(y_t|\mathbf{h_t}) \end{aligned} \tag{2.19}$$

In supervised learning, where the gold sequence is given, we can train the model by maximizing the log-likelihood

$$J(\theta) = \sum_i \sum_t \log P_\theta(y_t|y_0, ..., y_{t-1}) \tag{2.20}$$

We can use backpropagation through time (BPTT) to compute the gradient $\nabla_\theta J(\theta)$ and train the model through stochastic gradient descent. When there is only weak supervision available, we can use reinforcement learning to train the model, which we cover in the next section.

There are many variants of RNNs. The long-short term memory network (LSTM) [**53**] addresses the vanishing and exploding gradient problem and enables learning relatively long-term dependencies in a sequence. The sequence-to-sequence (seq2seq) models [**132**] uses two RNNs, one as an encoder to generate a hidden representation for the source sequence and the other as a decoder to generate the target sequence based on the output of the encoder. It

has been successfully applied to many NLP tasks such as machine translation [**146**], dialogue generation, question answering, etc.

In Chapter 4, we augment a RNN with an external memory to model the compositionality in a program and use it as a decoder in a seq2seq model that translates a natural language utterance into a program.

2.2.2. Policy Gradient methods

Reinforcement learning concerns an agent needing to learn to take actions in an environment so as to maximize the cumulative reward [**133**]. Many reinforcement learning problems can be defined as a markov decision process with $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}$ is the transition probability, $\mathcal{R}$ is the reward function and γ is the discount factor. Assuming the episodic or finite-horizon setting, the return of an episode R is the accumulated rewards.

$$R = \sum_{t=0}^{T} \gamma^t r_t \tag{2.21}$$

The goal is to learn a policy $\pi_\theta(a|s)$, which picks an action given a state, that maximizes the expected return.

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R] = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T} \gamma^t r_t] \tag{2.22}$$

θ parameterizes the policy.

Policy gradient methods are one type of RL algorithms that optimizes the expected return objective directly using gradient descent. The gradient of the objective J is

$$\begin{aligned} \nabla_\theta J(\theta) =& \nabla_\theta \mathbb{E}_{\pi_\theta}[R] \\ =& \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s,a)] \end{aligned} \tag{2.23}$$

$Q_{\pi_\theta}(s,a)$ is the state-action value function which is defined recursively as:

$$Q_{\pi_\theta}(s,a) = r(s,a) + \sum_{s'} \sum_{a'} \gamma P(s'|s,a) \pi_\theta(a'|s') Q_{\pi_\theta}(s',a') \tag{2.24}$$

There are many approaches to estimate the gradient defined in Equation 2.23. In the episodic or finite horizon setting, one commonly used algorithm REINFORCE [**143**] uses Monte Carlo estimation for Equation 2.23. It uses the final return of an episode as the estimate for Q_{π_θ}, which is unbiased, but has high variance.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) R] \tag{2.25}$$

One common technique to reduce the variance of the gradient estimator without introducing any bias is to subtract a constant from the gradient.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)(R - b)] \tag{2.26}$$

A common choice for baseline, which is close to optimal, is the expected return under the current policy.

$$b = \mathbb{E}_{\pi_\theta}[R] \tag{2.27}$$

There is a long line of literatures on policy gradient, which proposed methods such as actor-critic algorithm [**64, 105, 82**], natural policy gradient [**61**], trust region policy optimization [**119**] and poximal policy optimization [**121**], to improve the convergence speed and robustness. Off-policy methods are also used to improve its sample efficiency [**141, 33**].

We apply policy gradient methods to the structured prediction problem, where, in each episode, the policy constructs a structured output. For example, in Chapter 4, a policy is trained to generate programs, where each action is to output one valid token in the programming language. It is known that the policy gradient methods has bad sample efficiency and robustness given sparse rewards in a large search space. In Chapter 4, we propose new policy gradient methods to address these problem to successfully apply the proposed approach, which learns from weak supervision to generate programs given natural language utterances.

CHAPTER 3

Learning Similarity Estimation with Structural Alignment

Similarity estimation is an important component of human intelligence [**43**]. The common approach for similarity estimation assumes the inputs are vectors or can be flattened into vectors and apply dot product or cosine similarity. However, when the input data is relational (for example, facts from a knowledge base or syntactic structures of a sentences), flattening them into a vector does not make effective use of the structural information. Evidence from cognitive science has shown that structural alignment, which aligns the two relational structures to be compared and considers both structural similarity and local similarity, is key to similarity estimation. In this chapter, I investigate the proposed approach to generate structural alignments as the latent symbolic representations to help similarity estimation. We use knowledge base completion and text similarity benchmarks as evaluation.

3.1. Background: Structure-Mapping Engine

Using structural alignment as the symbolic representation to help estimate similarity is largely inspired by the structure-mapping theory (SMT) [**41**], which studies how humans make analogy and similarity judgment, and its computational model, the structure mapping engine (SME) [**34**]. SME has been used to model a variety of psychological phenomena [**38**]. It is also the foundation of the cognitive models of analogical retrieval [**39**] and analogical generalization [**79**].

SME takes as input two structured representations, a base and target, and produces one or more mappings. Each mapping consists of a set of matches or correspondences (i.e. what goes with what), a structural evaluation score that provides an overall estimate of the match quality, and candidate inferences. The similarity score of a mapping is usually normalized to [0,1] by dividing the raw score by the mean of the self-scores of the base and target. Forward

candidate inferences go from base to target, reverse candidate inferences go from target to base, which represent new facts based on the mapping.

SME ensures structural consistency in the mapping, which is defined by two constraints: (1) the one-to-one mapping constraint, which requires that each item in the base maps to at most one item in the target and vice versa. (2) the parallel connectivity constraint, which requires that if a correspondence between two expressions is included in a mapping, then correspondences between their arguments must also be included. To find the best mapping, SME follows the principle of systematicity, which prefers mappings with systems of predicates that contain higher-order relations, rather than to map isolated predicates.

SME has shown that structural alignment captures similarity over deep semantic representations. In this chapter, we apply structural alignment to shallow and noisy structures collected from crowd-sourcing and automatic extraction, for example, triplets from a knowledge graph or the dependency parse of a sentence, and combine it with machine learning methods to tackle two real world tasks, knowledge base completion (link prediction) and paraphrase identification.

3.2. Knowledge Base Completion

3.2.1. Motivation

Intelligent systems need knowledge and the ability to reason with it. Hand-coding knowledge and inference rules is not a scalable solution. KBs like Freebase [**14**], WordNet [**81**], and YAGO [**129**] are potential sources to use and have accumulated considerable structured data, which encodes knowledge about different domains. They are already used to support applications like question answering and information retrieval. These KBs continue to grow rapidly. However, using these KBs brings up two concerns. First, because the fact in the KB can be extracted from text or collected by crowd-sourcing, they are often incomplete and noisy. Traditional logical inference may not be sufficiently robust to reason over them. Second, unlike images, auditory data or raw text, this data is inherently structured. Although traditional statistical machine learning methods can handle the noise well, most such methods

are designed to work over feature vectors, and cannot exploit the relational structure in the data effectively.

One testbed for reasoning with these KBs is the knowledge base completion or link prediction task. In KBs like Freebase and WordNet, knowledge is stored in the triplet format: "entity relation entity". Since there are only binary relations, it can be seen as a labeled, directed graph. Each entity is a node, and each triplet between two entities is an edge labeled with the relation. In the knowledge base completion task, the model should learn to distinguish correct triplets like "Obama nationality USA" from incorrect ones like "Obama nationality Kenya", which can be viewed as a binary classification problem.

3.2.2. Method

The main idea is that the systems learns a parameterized template for each relation, and then use the estimated similarity score between the facts about two entities $e1$ and $e2$ with the template of the relation r to decide whether the relation holds for the two entities.

There are three main steps in the method as outlined in figure 3.1: (1) generate structured representations for each triplet as preprocessing; (2) discover the templates that include the important facts; (3) integrate SME and logistic regression to jointly learn the parameters (weights of each fact in the template) for generating the structural alignment and estimating the similarity.

3.2.2.1. Data Preprocessing through Path-finding. First, we perform case construction by path-finding, which finds the paths between two nodes in a graph. A case contains structured information about a particular triplet. The idea is to include enough information to enable a system to distinguish correct from incorrect triplets, while limiting the size of the cases for the sake of efficiency and to reduce the number of irrelevant matches.

A KB with only triplets can be viewed as a graph. We use path-finding as a heuristic to pick relevant facts for case construction. For example, when we want to create a case for a triplet with e_1 and e_2 as head and tail entity, we use depth-first-search to find paths between them and put all the facts along the paths into the case. In a large scale and highly connected

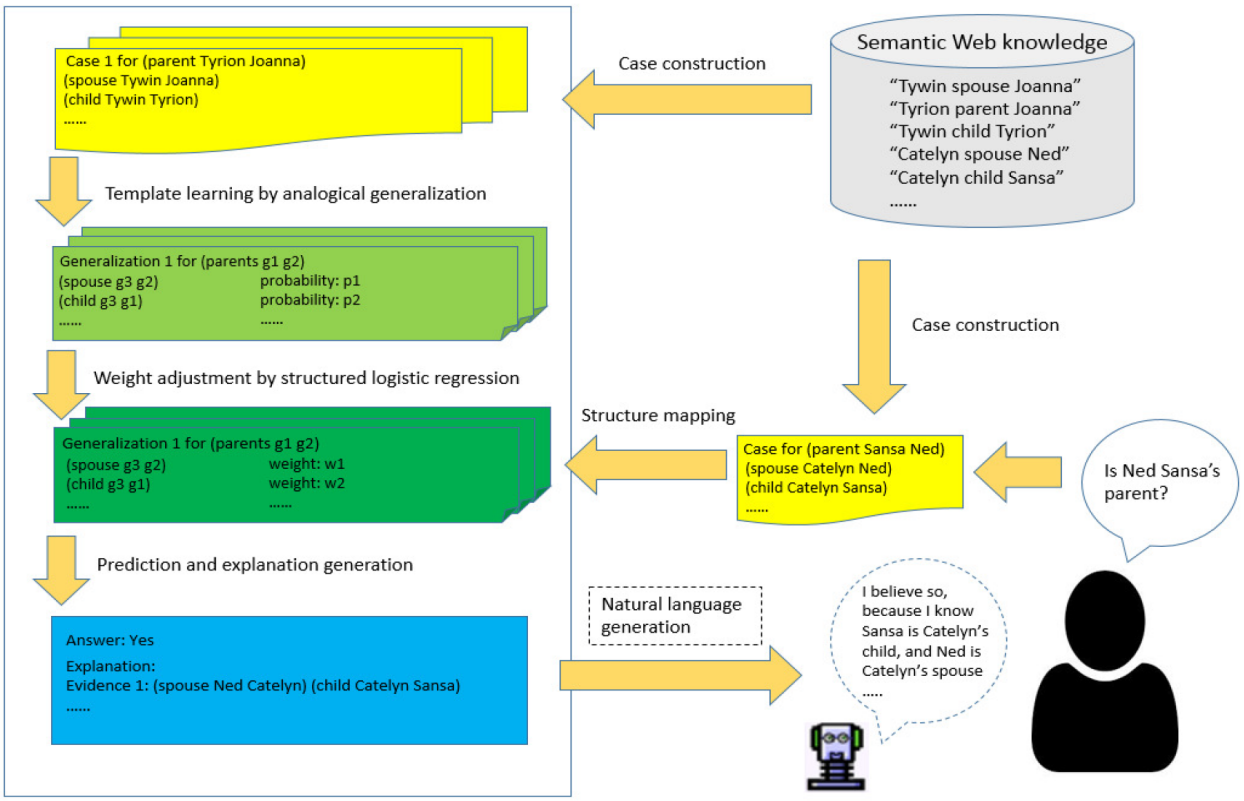

Figure 3.1. SLogAn's workflow for a possible use case: the learning system gets input from a knowledge base, does template learning and parameter learning with the data, and, with the learned generalizations, it is able to provide answers and explanations to the users queries.

knowledge base, an exhaustive search will be prohibitive, so we use limits on branching factor and search depth to randomly select parts of the search tree to explore.

Since there are only positive examples in the original dataset, we corrupt the correct triplets by changing their tail entities to wrong ones to get negative examples, which is similar to [**126**]. We treat the training set as our knowledge base, and create cases from it. With just a few cases, our method is able to learn plausible inferences. For the knowledge base completion task, we used only 10 positive and 10 negative examples for each relation.

Second, to decide what to include in the inference, we apply SAGE (Sequential Analogical Generalization Engine) [**79**] to learn a template for each relation. SAGE can create generalizations by comparing examples and compressing them into one prototype. Here, a generalization works as a template for inferring a certain relation. It is trivial to compare

feature vectors using cosine similarity because they all share the same dimensions, but it is non-trivial to find the best way to align two structured representations. SAGE applies SME to find the best structural alignment between the examples, and uses these alignments to compress the examples into one structurally consistent template. We also require the head entity and tail entity of one triplet to be respectively matched to those of the other triplet to make sure they are the focus of the template. Note that, unlike the way SAGE is usually used, we first create generalizations with positive examples only, and then add negative examples to these generalizations so that they contain facts from positive as well as negative examples. In this way, some facts could contribute negatively to the target relation. For example, if I know that e_2 is e_1's parent, then e_1 cannot be e_2's parent. Although that fact never appears in a positive example, it is still a critical fact to consider in the inference.

3.2.2.2. Structural Alignment and Plausible Inference. In this step, given the training examples and the templates, the model learns the parameters using an integration of SME and logistic regression.

To represent how much each expression in the template supports or contradicts the target relation, each expression E_i is associated with a weight w_i measuring its support for the relation to hold. Note that the weight w_i will be negative if E_i contradicts with the relation and positive if E_i supports the relation. Given an example, we compare it to the template using SME. We use the absolute value of the weight $|w_i|$ of an expression E_i as its weight when applying SME. In this way, SME will prefer mappings with matches of expressions that have larger contributions to the final decision (measured by $|w_i|$). With the resulting structural alignment, we compute the probability of the relation being true as

$$p = \frac{1}{1 + e^s} \tag{3.1}$$

$$s = \sum_{E_i \in \mathcal{M}} w_i \tag{3.2}$$

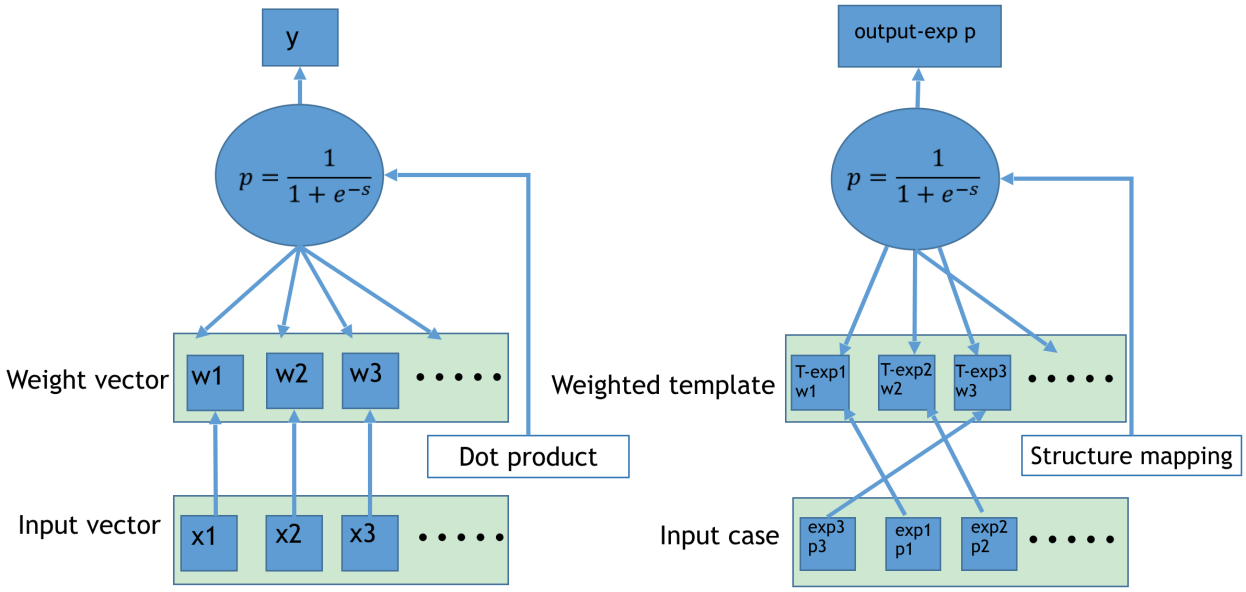

Figure 3.2. A comparison of the standard logistic regression (on the left), which assumes vector representation and applies dot product, and the structured logistic regression (on the right), which assumes structured representation and uses structure mapping to compare the template and the input.

$\mathcal{M}$ is the set of expressions that get matched in the given alignment. The score s is computed as the sum of the weights of the matched expressions. The higher the score, the higher the probability of the triplet to be true. Let θ be the parameter vector that consists of the weights for the expressions. A comparison of the standard logistic regression with the current model is in Figure 3.2.

We use cross entropy loss to define the training objective $J(\theta)$ and add L1 penalty to promote sparsity in weights as regularization and to improve interpretability:

$$J(\theta) = -\sum_{i=0}^{n}[(1-y_i)\log(1-p_i) + y_i \log(p_i)] + \alpha \sum_{j=1}^{m}|w_i| \tag{3.3}$$

n is the number of examples in the training set. m is the number of expressions in the template. y_i and p_i are the label (1 for positive, 0 for negative) and predicted probability of the i-th example. α controls the degree of the regularization. We can then compute the gradient $\nabla_\theta J(\theta)$ and train the parameters θ through gradient descent. The prediction accuracy on the validation set is used to decide early stopping.

To evaluate the method, we compare its accuracy to the state-of-the-art on the knowledge base completion task. We use the datasets collected by [**126**]. One dataset contains triplets from Freebase consisting of 13 relations, the other dataset, from WordNet, consisting of 11 relations. More details about the datasets is shown in Table 3.1.

Table 3.2 shows the accuracy on test set. Using only 20 examples, our model has achieved results competitive to state-of-the-art. It is 2nd on WN11 and FB13, and no method is better than ours on both of the datasets.

Model	#Relations	#Entities	#Train	#Valid	#Test
WordNet11	11	38,696	112,581	2,609	10,544
FreeBase13	13	75,043	316,232	5,908	23,733

Table 3.1. Statistics of the WordNet11 and Freebase13 datasets

Model	WordNet11	FreeBase13
Distance Model	53.0 %	75.2 %
Hadamard Model	70.0%	63.7 %
Single Layer Model	69.9%	85.3 %
Bilinear Model	73.8%	84.3%
Neural Tensor Network	70.4%	87.1%
TransH	78.8%	83.3%
SLogAn (Our model)	75.3%	85.3%

Table 3.2. Knowledge base completion accuracy (%)

3.2.3. Discussion

In summary, our model learned plausible inferences from the KB with only a few examples and achieved results competitive to the state-of-the-art on the knowledge base completion or link prediction task. Moreover, it can provide explanations for its inference based the structural alignment. For example, when asked about "is Taufa'ahau Tupou IVs ethnicity Tongan?", even though the system does not know this fact, it can learn from a few examples how to infer the "ethnicity" relation and make an accurate prediction. If the users are only provided with the answer, they have to decide whether to believe it or not based on their

trust of the system. However, by examining what expressions are matched in the structural alignment, the system can add, "I believe Taufa'ahau Tupou IVs ethnicity is Tongan, because I know his parents' ethnicity is Tongan, and I remember a person with the same nationality whose ethnicity is also Tongan". This makes the system more interpretable to the users, and can alert them when the system makes an invalid inference.

3.2.4. Related work

Many vector embedding based methods has been tested on the knowledge base completion tasks. [**126**] and [**140**] are the most recent and have the best accuracy. Their models are quite different from ours. They learn vector representations for the entities that implicitly encodes knowledge about them and use translation of the vector or tensor product for inferring new relations. During training, they created negative examples for every triplet in the training set and train on all of them. In contrast, our method only randomly selects a few triplets and creates cases and corresponding negative examples for them. Instead of implicit encoding of knowledge with vector embedding, we use the structured knowledge directly and learn templates that explicitly encode valid inferences. This makes our model more interpretable. Path-finding for relational learning is used by a lot of previous work. [**123**] uses it to learn valid inference chains. They are using plausible inference patterns and type information of the relations to find correct inference, while we are learning purely from ground facts. Their way of using the existing knowledge would be a possible improvement to our method. [**110**] uses path-finding to find candidate clauses for learning first-order rules, but it is computationally expensive. [**66**] uses limited length path-finding in the NELL knowledge base to create features with the paths found and do logistic regression with them. Our model can use facts that doesn't form a complete path in the inference. Also note that path-finding is just one heuristic we use for case construction, analogical generalization and structured logistic regression can be combined with other case construction methods like dynamic case construction [**86**] or spreading activation as long as they generate structured representations. Structural logistic regression [**106**] generates features by propositionalizing

first-order rules learned by inductive logic programming, and uses logistic regression with these features for classification. Relational logistic regression [**62**] uses logistic regression to learn weights for first-order formulas in defining the conditional probability of a new relation given those formulas. Their ways of adding the counts of certain facts as features are possible improvements for the current model. Although the knowledge base completion or link prediction task only deals with binary relations, our model has the ability to deal with first-order and high-order relations because it builds on SME and SAGE, which can handle both of these types of relations.

3.2.5. Conclusion

In this work, we integrate SME with logistic regression to learn to estimate the similarity of the facts about two entities, e_1 and e_2, with a template of a relation, r, using structural alignments. The similarity score is used to predict whether the triplet (e_1, r, e_2) is true or not. Our model achieved state-of-the-art performance on the knowledge base completion task with orders of magnitude less data and can provide explanations for its inference based on the structural alignment.

3.3. Text Similarity

3.3.1. Motivation

Semantic similarity of texts is used in many tasks like paraphrase identification [**30**], textual entailment [**24**], and question answering [**137**]. Although simple bag-of-words models work well for large documents, short texts are challenging because those simple models suffer from sparsity.

The semantics of text is often decomposed into two important parts. The first part is the local information, i.e., semantic of lexical units. Recently there has been a lot of improvements in semantic tasks using word embeddings learned from a large corpus [**8, 80**]. The second part is the structural information, i.e., syntactic and semantic structure. For example, a sentence pair from recently introduced Stanford NLI corpus [**15**] “A man wearing

padded arm protection is being bitten by a German shepherd dog / A man bit a dog" is wrongly predicted as having entailment relation by both a lexicalized classifier and a sentence embedding model. If explicit syntactic and semantic structures are used, the difference between the two sentences' meanings will be obvious. The development of natural language analysis tools [**78**] makes it easy to efficiently generate a variety of syntactic and semantic annotations like dependency parses, POS tags, entity mentions as well as semantic role labeling and open relation extraction.

Despite the progress on both sides, how to effectively combine the local and structural information is still an open question. Our system jointly learns to generate structural alignments and estimate semantic similarity. It uses a hybrid representation, attributed relational graphs [**116, 159**], to encode lexical, syntactic and semantic information together. In order to exploit the relational structure, we use structural alignment as an intermediate symbolic representation to support similarity estimation. Different from word alignments, structural alignment forms consistent correspondences between both words and syntactic and semantic structures of two pieces of text. To get better alignment, we introduce structural constraints to utilize the predicate-argument structure encoded in the graphs. These structural constraints are inspired by the structure-mapping theory [**42**], a cognitive theory of similarity and analogy and its computational model, structure-mapping engine. More details can be found in Section 3.1.

3.3.2. Related Work

A lot of different approaches have been proposed for text similarity. We focus on two main categories most related to ours.

Neural network models extend the idea of word embedding to larger constructions like phrases, sentences and paragraphs. A variety of architectures have been explored. The recursive neural network in [**127**] used a constituency tree and recursive autoencoder to learn composition functions of word embeddings to phrase embeddings and eventually sentence embeddings. Tree-LSTM [**134**] generalizes LSTM from sequences to tree structures. Other

recent approaches include the Siamese architecture with syntax-aware multi-sense embeddings [**19**] and convolutional neural networks [**50, 151**]. Our approach also uses word embeddings, but instead of trying to compress all the information into one fixed length vector, we use word embeddings together with other linguistic structures explicitly encoded in the attributed relational graph.

Some approaches explore syntactic structures of two sentences. Tree kernels and graph kernels with SVM have been used on syntactic structures extended with relational links between similar words [**35**] to predict relations between texts. Quasi-synchronous grammar was used in [**26**] to model divergence of syntactic structure between paraphrases. [**11**] used probabilistic soft logic to combine logical and distributional representations through weighted inference rules. Our method took a similar hybrid approach, but used attributed relational graphs as a extensible representation to encode various different kinds of lexical, syntactic and semantic information.

Alignment has been used as an intermediate step in several NLP tasks. For example, [**115**] used alignment of structured annotations from different resources for textual entailment. Word alignment as latent variable was used in machine translation [**75**]. Neural attention models in machine translation [**6**] and textual entailment [**111**] can be seen as jointly learning soft alignments between words in source and target sentences or words in text and hypothesis. The problem that alignments are latent in data is a common challenge for these alignment-based approaches. Latent variable model and joint learning is a commonly used solution. We take a similar approach here. However, different from word alignment, our alignment is carried on both words and linguistic structures. By using structural constraints, it utilizes the predicate-argument structure in the text, which is generally ignored in other works.

Structural alignment as an intermediate step to support similarity estimation also has support from cognitive science. Structure Mapping Theory (SMT) [**42**] states that similarity judgment and analogy is done through structural alignment of mental representations. In the alignment process, humans prefer structurally consistent alignment of deep nested structure, which is called structural consistency and systematicity principle. This theory and its

computational model Structure Mapping Engine (SME) [**34**] has been proven to fit human performance in various different experiments [**36**]. A variant of the algorithm was used in the IBM Watson Jeopardy system for evaluating candidate answers [**88**]. In the previous section, we introduced the work [**72**] that combined SME with statistical learning and showed state-of-the-art performance on knowledge base completion task using orders of magnitude less training data than other approaches. Structural alignment is a major component of our approach, and the structural constraints we introduced to our alignment model is inspired by SMT's structural consistency principle.

3.3.3. Method

The problem of semantic similarity is, given two pieces of texts, the system must produce a score indicating the degree that their meanings are equivalent. We solve this problem with a pipeline of three components similar to RATER [**115**].

(1) **Graph extractor**: Given two pieces of texts, it uses word embeddings and a set of automatic annotators to extract the tokens, syntactic relations, POS tags and entity mentions to generate the attributed relational graph.

(2) **Structural aligner**: Given two attributed relational graphs, the structural aligner generates an alignment. An alignment of two attributed relational graphs is a set of matches, and each match is a correspondence between two nodes or edges.

(3) **Similarity estimator**: Given an alignment, the similarity estimator produces a similarity score between the two graphs or a label indicating whether they are similar enough to be considered equivalent.

3.3.3.1. Data Representation and Graph Extraction. Our method uses a hybrid representation, attributed relational graphs (directed graphs with attributes attached to the nodes and edges). See figure 3.3 for an example. The attributes store local information about a unit/node or a relation/edge, which will later be used to extract features for each match between two nodes or two edges. These features are used to estimate the similarity and importance of the match. In our experiment, for fair comparison with other methods,

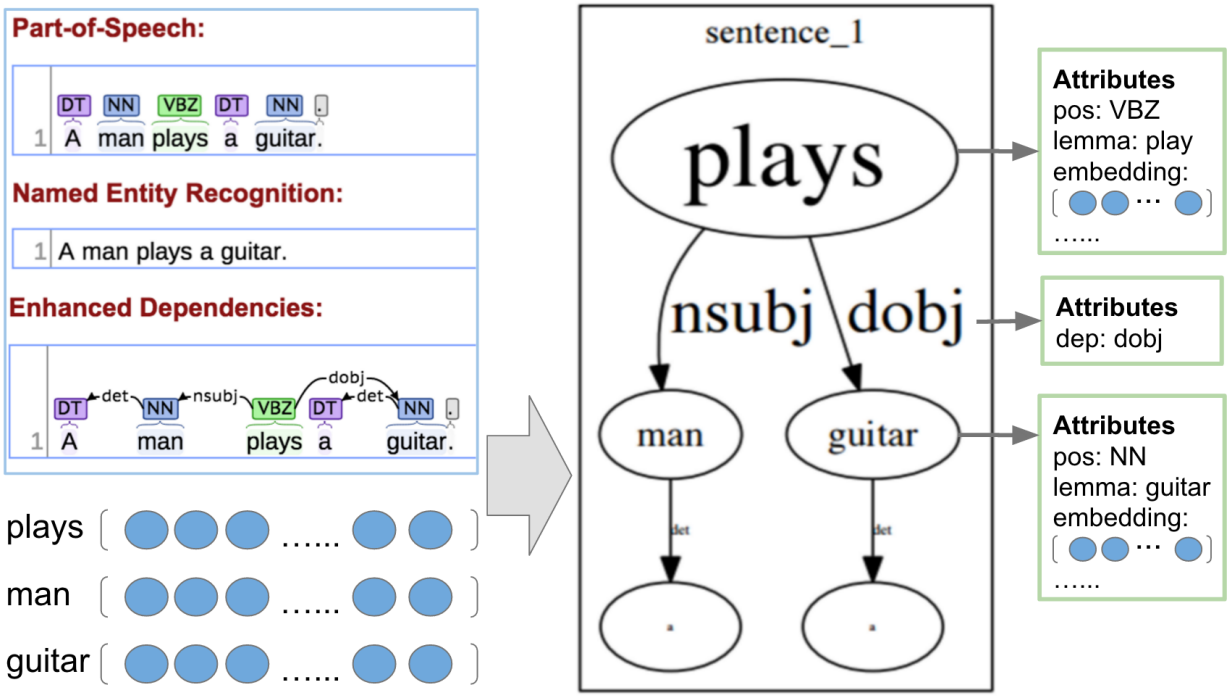

Figure 3.3. A example of annotations turned into an attributed relational graph, structural information such as syntactic structure is encoded in the graph structure, and local information such as word embedding or dependency label is encoded in the attributes attached to nodes and edges

we just use tokens as units/nodes and dependency arcs as relations/edges. For attributes, we used dependency label, token, lemma, POS tag, NER tag and word embedding. These annotations can all be obtained easily through standard natural language analysis tools. Here we used Stanford CoreNLP and pretrained Word2Vec word embeddings[1] [**80**]. We refer to this preprocessor as the graph extractor.

3.3.3.2. Structural Alignment and Similarity Estimation. The two core components of our approach are the *structural aligner* and *similarity estimator*. Given two input graphs, the structural aligner finds the best alignment between them, which can be seen as a structured prediction problem. Based on the best alignment, the similarity estimator produces a score indicating the degree of similarity or a binary output indicating whether the two sentences

[1]https://code.google.com/archive/p/word2vec/

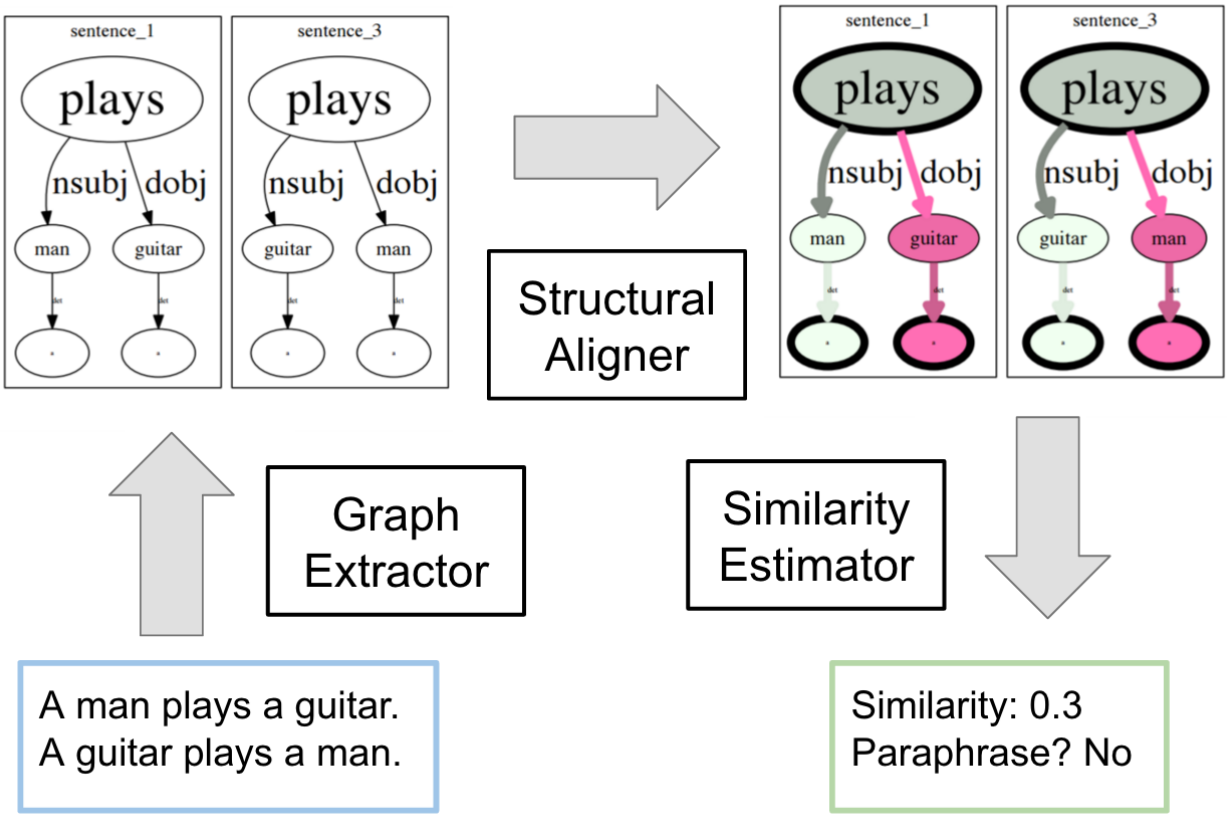

Figure 3.4. An overview of the full pipeline consisting of three main components to turn raw texts into attributed relational graphs, structurally align them and estimate their similarity. The color indicates how the nodes and edges from two graphs match with each other in the alignment.

are semantically equivalent or not, which can be seen as a regression or classification task depending on which output is produced.

An alignment is a set of matches. Each match is a pair of nodes or edges from the two graphs. The structural alignment has two steps. First, given two graphs, the structural aligner generates all the possible matches that pass some criteria. In this work, the criteria we used are that (1) two matched dependency arcs must have the same dependency label; (2) the cosine similarity between word embeddings of two matched tokens must be greater than 0.4. This value is chosen from pilot experiments with a subset of the data. The result is not sensitive to it because most of the matches that passed this threshold have similarities much

higher than it. Second, it selects the subset of matches that optimizes an objective defined in Equation 3.4.

Formalized as a structured prediction problem, the input x is a pair of graphs and the output a is an alignment. Φ_a(x, a) is a function mapping input graphs and a candidate alignment to a feature vector. Details about the features can be found in the next section. Let w_a be the set of parameters for the structural aligner, and A be all the possible subsets of all the possible matches. Finding the best alignment is solving the following problem:

$$a_{pred} = \underset{a \in A}{\operatorname{argmax}}\, w_a \cdot \Phi_a(x, a) \tag{3.4}$$

This argmax problem is intractable, so we use beam search or greedy search to find an approximate solution.

Formalized as a regression or classification problem, the similarity estimator takes the pair of graphs and the predicted alignment as input, and uses another feature function $\Phi_s(x, a_{pred})$ to map them to a feature vector. Let w_s be the set of parameters for the similarity estimator. Here we consider the paraphrase identification task. Since the correct similarity label y_{true} is usually given in training data, this is a supervised binary classification problem. Using a linear classifier, the similarity label y_{pred} is predicted as:

$$y_{pred} = sgn(w_s \cdot \Phi_s(x, a_{pred})) \tag{3.5}$$

In this work, we use a SVM to learn w_s.

Learning w_a is more challenging because the true alignment a_{true} is latent. We address this problem using two approaches, alignment as feature extraction and alignment as latent variable.

In the first approach, we consider the structural aligner as a feature extractor and w_s as hyperparameters, and use grid search over a validation set to select the best parameters. But the problem is that the number of runs needed in grid search grows exponentially with the number of hyperparameters, So we have to restrict Φ_s to include just a small set of features.

To overcome these problems and utilize more features in the structural aligner, the second approach jointly trains the structural aligner and the similarity estimator through an iterative EM-like process, as follows. First, we initialize the parameters to some value w_a^0, for example using the values obtained from the first approach. Then we repeat two steps:

(1) Keep w_a fixed, for each input x^i, assume the alignments produced by the structural aligner is "correct" and learn w_s from examples $\{(x^i, a^i_{pred}, y^i_{true}), i = 1, 2 \ldots, n\}$.

(2) Keep w_s fixed, for each input x^i, hallucinate the "correct" alignment a^i_* by solving

$$a^i_* = \underset{a \in A^i}{\operatorname{argmax}}\, w_a \cdot \Phi_a(x^i, a) - C \cdot Loss(y_{true}, w_s \cdot \Phi_s(x^i, a)) \tag{3.6}$$

where C is a hyperparameter that represents the confidence in the classifier. Since we use SVM to learn w_s, the *Loss* here is hinge loss. In the experiment, we set C to be very large (10^6). Thus, this argmax solves for the alignment that causes the lowest prediction error and, if there is a tie in the error, has the highest alignment score $w_a \cdot \Phi_a(x^i, a)$. Then, examples $\{(x^i, a^i_*), i = 1, 2 \ldots, n\}$ are used to train the aligner using the averaged structured perceptron algorithm [**21**].

Because this process will usually overfit the training data, we use the performance on a validation set to decide when to stop.

3.3.4. Features

In this section, we discuss the features used in feature functions Φ_a and Φ_s, and focus on how the pairwise features are used to encode the structural constraints. We first describe a feature function Φ, then show how Φ_a and Φ_s are defined using Φ.

The (global) feature vector $\Phi(x, a)$ is a concatenation of global unary feature vector $\Phi_u(x, a)$ and global pairwise feature vector $\Phi_p(x, a)$.

$$\Phi(x, a) = [\Phi_u(x, a);\ \Phi_p(x, b)] \tag{3.7}$$

The global unary and pairwise feature vectors are aggregations of local unary and pairwise feature vectors. In other words, a global feature vector is just a sum of local feature vectors.

Recall that an alignment a is just a set of individual matches $\{m_i\}$. The local unary feature vector $\phi_u(x, m_i)$ is computed for each individual match m_i, and the local pairwise feature vector $\phi_p(x, m_i, m_j)$ is computed for each pair of matches (m_i, m_j).

$$\Phi_u(x, a) = \sum_i \phi_u(x, m_i) \tag{3.8}$$

$$\Phi_p(x, a) = \sum_{i,j} \phi_p(x, m_i, m_j),\ i \neq j \tag{3.9}$$

For the structural aligner, only the ranking of candidate alignments matters, because it just produces the one with highest score as the output. In this case, the aggregation of local features Φ suffices to be a good feature vector. So Φ_a is simply defined as:

$$\Phi_a(x, a) = \Phi(x, a) \tag{3.10}$$

For the similarity estimator, however, the value of $w_s \cdot \Phi_s(x, a_{pred})$ matters because its sign decides the prediction and its absolute value roughly represents the confidence in the prediction. So the feature vector needs to be normalized.

Here we use the self alignments to normalize. Note that the input x is just a pair of attributed relational graphs (g_1, g_2) extracted from the pair of sentences. The self alignment feature vector is defined as:

$$\Phi_{self}(x) = \frac{\Phi(x^1_{self}, a^1_{self}) + \Phi(x^2_{self}, a^2_{self})}{2} \tag{3.11}$$

where $x^1_{self} = (g_1, g_1)$ and $x^2_{self} = (g_2, g_2)$. a^1_{self} and a^2_{self} are the self alignments of g_1 and g_2, in which each node and edge just matches to itself.

Then Φ_s is defined as concatenation of three vectors:

$$\Phi_s(x, a) = [\Phi(x, a);\ \Phi_{self}(x);\ \frac{\Phi_{self}(x) - \Phi(x, a)}{\Phi_{self}(x) + \delta}] \tag{3.12}$$

where δ is a very small smoothing term. The third vector is the normalized difference between self alignment's aggregation feature vector and the current alignment a's aggregation feature

vector. Because each dimension i of the vector corresponds to one feature and for most of the features[2] we used, $0 < \Phi(x, a)_i < \Phi_{self}(x)_i$. So each dimension of the third term is bounded between 0 and 1. We still include the raw aggregation feature vector $\Phi(x, a)$ and self alignment aggregation feature vector $\Phi_{self}(x)$ in the final feature vector because the raw values of these features are also informative and were shown to improve the performance in the experiments.

3.3.4.1. Unary Features. Unary features are used to estimate how similar the two matched tokens or dependency arcs are, and also how important they are in their sentence. These features are used to compute how much this match will contribute to the alignment score or overall similarity. Listed below are the features we used as unary features in this work.

(1) **Lexical similarity**: cosine similarity between word embeddings of the matched two tokens.

(2) **Lexical features**: word features for words that appeared at least twice, lemma features and an indicator feature for whether the two matched tokens have the same lemma.

(3) **Syntactic features**: POS tag features, and an indicator feature of whether the matched two token has the same POS tag; dependency label features, and an indicator feature of whether the matched two dependency arcs have the same dependency label.

(4) **NER features**: NER tag features, and two indicator features, one for whether the matched two tokens has the same NER tag, and another one for whether the matched two token has the same normalized entity name.

(5) **Position difference feature**: the difference between the positions of the matched two tokens in the their sentences.

3.3.4.2. Pairwise Features. Pairwise features are introduced to improve alignment by encoding the structural constraints between matches. These structural constraints ensure that the final alignment is structurally consistent. They are inspired by the structural consistency

[2]The only feature that violates this is the position difference feature, so we don't include it in this normalized term and just keep its raw values.

principle of Structure Mapping theory. The principle states that two constraints are used by human when aligning predicate argument structures: (1) one-to-one mapping that one entity or predicates should only match to one entity or predicate; (2) parallel connectivity that if a predicate matches another predicate, their roles and arguments should also match correspondingly.

In this work, we adapted these constraints to work on tokens and syntactic relations. The one-to-one mapping is encoded as a hard constraint in the structural aligner so that alignments that matches one token or relation to more than one other token or relation would be filtered out. The parallel connectivity constraint is adapted by considering dependency tree as an approximate predicate argument structure. So if the heads of two dependency arcs matches, the two dependency arcs should also be more likely to match. If two dependency arcs matches, the tail of the dependency arcs should be more likely to match as well.

These two constraints are implemented as two pairwise indicator features. For a match m_i between two tokens t_a and t_b, and a match m_j between two dependency arcs d_a and d_b.

$$\phi_p(x, m_i, m_j)_1 = \begin{cases} 1, & \text{if } t_a = head(d_a) \ \& \ t_b = head(d_b) \\ 0, & \text{otherwise} \end{cases} \tag{3.13}$$

$$\phi_p(x, m_i, m_j)_2 = \begin{cases} 1, & \text{if } t_a = tail(d_a) \ \& \ t_b = tail(d_b) \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

Note that in this work, for simplicity, we just used pairwise features to demonstrate the utility of structural constraints. Structural constraints involving more matches can be modeled using higher-order features, and more fine-grained constraints can make use of the attributes of nodes and edges.

Model	Accuracy	F1
Das and Smith (2009)	76.1%	82.7%
Wan et al. (2006)	75.6%	83.0%
Socher et al. (2011)	76.8%	83.6%
Madnani et al. (2012)	77.4%	84.1%
He et al. (2015)	78.6%	84.7%
Cheng and Kartsaklis (2015)*	78.6%	85.3%
Ji and Eisenstein (2013)*	80.4%	85.9%
This work		
local similarity	76.9%	83.9%
+structural constraints	77.4%	84.2%
+syntactic features	78.2%	84.7%
+latent variable model	78.3%	84.9%
This work	78.3%	84.9%

Table 3.3. Test results on MSRP paraphrase dataset

3.3.5. Results & Analysis

We compared our model to other state-of-the-art models[3]. The result is shown in Table 3.3. Despite its simplicity, our model is competitive to the state-of-the-art. The two results labeled with * used extra data besides the training set. [**56**] used matrix factorization on both training and test set to extract distributional features, and [**19**] used PPDB [**40**], which is several orders of magnitude larger, as training data. [**35**] used trees as structured representations of text, and lexical matching was also an important component in their method. They achieved better results with more complex features and a thorough comparison and combination of different graph and tree kernels, while our much simpler alignment-based method showed comparable performance.

For different experiment settings, we kept the features used in the similarity estimator fixed, and varied the features used in the aligner. For the first three settings, the parameters for the aligner are decided using grid search over a validation set.

The baseline similarity estimator uses SVM with a set of simple features: (1) cosine similarity between average word embeddings of the two sentences; (2) simple number features,

[3] `http://aclweb.org/aclwiki/index.php?title=Paraphrase_Identification_(State_of_the_art)`

percentage of words in the other sentence and sentence length difference used in [**127**]; (3) BLEU1 through BLEU4 as separate features.

The **local similarity** setting adds lexical similarity and same dependency label features. In other words, it uses only local similarity of the matched two tokens or dependency arcs to find the best alignment. Surprisingly, this simple approach already outperforms the baseline and three other sophisticated models. This showed that word embeddings and shallow features are not good enough, but the combination of word embeddings with alignment and rich features is surprisingly effective.

Using only local similarity, the alignment contains separating matches that do not connect with each other. The **+structural constraints** setting added the two pairwise indicator features to promote structural consistency. With these two features as structural constraints, the aligner prefers a set of consistent matches between two connected syntactic tree structure over matches between scattered pieces. This improved the F1 score by another 0.3 percent.

Since the dependency tree is used as an approximation to the predicate-argument structure, this approximation makes more sense for some dependency relations, such as *nsubj*, *nsubjpass* and *dobj*, and some words, such as verbs and nouns, than others. Using this heuristic, the **+syntactic features** setting added verb and noun POS tag features and features for those three dependency labels. This would help the aligner focus on the words and dependency relations that capture the predicate-argument structure better during the search for a set of structurally consistent matches. This increased the F1 score by another 0.4 percent.

The **+latent variable model** setting explored the full parameter space for the aligner using the iterative training described in Section 3.2. It takes the best parameters from last setting as initialization. We run averaged structured perceptron for 10 epochs. The averaged parameters after each epoch is stored as $\{w_i, i = 1, 2, ..., 10\}$, and we used a validation set to decide which one to use as the final parameters. We also used the validation set to decide when to stop the iterative training. This further improved the F1 score by another 0.2 percent.

We used 5-fold cross validation on the whole dataset to check the statistical significance of the differences between settings. We found that the improvement of local similarity setting to the baseline and the improvement of the full model to the local similarity setting are both statistically significant. But the differences within the three structural alignment settings are more subtle and not statistically significant.

The alignment and rich features enabled the system to learn which part of the sentences are more important to its semantic rather than treating them all the same. This eliminates many false positives caused by misleading lexical overlap. For example, "Gyorgy Heizler, head of the local disaster unit, said the coach had been carrying 38 passengers.", and "The head of the local disaster unit, Gyorgy Heizler, said the coach driver had failed to heed red stop lights." was classified wrongly as paraphrase by the baseline because of high lexical overlap, but using the local similarity setting, it made the right prediction.

Structural alignment further eliminates false positives because it helps constrain the lexical matches. For example, "a dog bites a man" and "a man bites a dog" have a perfect alignment in local similarity setting, but will not be recognized as similar by the full model, because the syntactic structures lead the aligner to match "dog" with "man" and "man" with "dog", which are not similar.

3.3.6. Conclusion

In this work, we showed how to jointly learn to generate structural alignment and estimate the semantic similarity of texts and evaluated it on the paraphrase identification task. We used a hybrid representation, attributed relational graphs, to encode local and structural information. This enables us to integrate structural alignment and similarity estimation through two approaches: alignment as feature extraction and alignment as latent variable. In the experiment, our approach achieved results competitive with state-of-the-art models on the MSRP corpus.

CHAPTER 4

Learning to Generate Programs from Natural Language

Understanding natural language is a fundamental goal for artificial intelligence. It requires modeling the compositionality in natural language, making use of prior knowledge and performing complex reasoning, which makes it hard for current machine learning methods, such as neural networks. On the other hand, programs naturally handle compositionlity through the use of memory, and can make use of external knowledge and modules by calling predefined functions. In this chapter, I apply the proposed approach that learns to generate programs as the latent symbolic representation from weak supervision (question-answer pairs) to answer open-domain compositional questions using Freebase and tables.

4.1. Background: Compositional Semantics

Semantic parsing is the process of translating natural language utterance into a formal semantic representation. Montague semantics [**85, 84**] is one paradigm in compositional semantics that relies on higher-order functional models such as lambda calculus to provide denotations for the semantic representations. Combinatory categorial grammar (CCG) [**128**] is a lexicalized grammar used by many semantic parsers [**156, 157, 158**]. The grammar is defined using a lexicon that maps words to categories with semantic interpretations and a set of combinatory rules like functional application, functional composition and type-raising. Dependency-based compositional semantics (DCS) [**73**] and its variants are recently proposed semantic representations for semantic parsing. It relies on lambda calculus and the derivation of the semantic representation is usually parallel to the syntactic dependency tree. It is used in recent work on learning semantic parsers from weak supervision [**12, 99**].

We draw inspirations from these compositional semantics formalisms and take a more practical approach by using domain specific languages implemented in Lisp as semantic

representations. Similar to modern programming languages, the program is generated from a bottom up manner so that the results of early expressions are saved to variables and reused by later expressions. This representation has the same expressive power as other compositional semantics, but easier to generate by interacting with a Lisp interpreter and easier to extend by defining more functions in Lisp. The Lisp interpreter can also help prune the search space by constraining the programs to be syntactically and semantically correct.

4.2. Semantic Parsing

4.2.1. Motivation

Deep neural networks have achieved impressive performance in supervised classification and structured prediction tasks such as speech recognition [**52**], machine translation [**6, 146**] and more. However, training neural networks for semantic parsing [**155, 156, 73**] or program synthesis, where language is mapped to a symbolic representation that is executed by an executor, through weak supervision remains challenging. This is because the model must interact with a symbolic executor through non-differentiable operations to search over a large program space.

In semantic parsing, recent work [**31, 57**] handled this by training from manually annotated programs and avoiding program execution at training time. However, annotating programs is known to be expensive and scales poorly. In program induction, attempts to address this problem [**45, 109, 60, 47, 4**] either utilized low-level memory [**154**], or required memory to be differentiable [**91, 150**] so that the model can be trained with backpropagation. This makes it difficult to use the efficient discrete operations and memory of a traditional computer, and limited the application to synthetic or small knowledge bases.

4.2.2. Method

In this work, we propose to utilize the memory and discrete operations of a traditional computer in a novel Manager-Programmer-Computer (MPC) framework for neural program induction, which integrates three components:

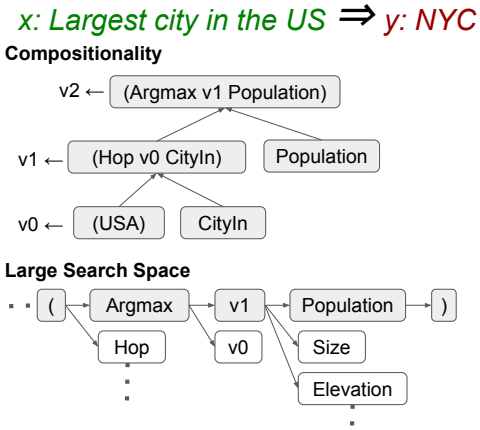

Figure 4.1. The main challenges of training a semantic parser from weak supervision: (a) *compositionality*: we use variables (v_0, v_1, v_2) to store execution results of intermediate generated programs. (b) *search*: we prune the search space and augment REINFORCE with pseudo-gold programs.

(1) A *manager* that provides weak supervision (e.g., 'NYC' in Figure 4.1) through a reward indicating how well a task is accomplished. Unlike full supervision, weak supervision is easy to obtain at scale (Section 4.2.3.1).

(2) A *programmer* that takes natural language as input and generates a program that is a sequence of tokens (Figure 4.2). The programmer learns from the reward and must overcome the hard search problem of finding correct programs (Section 4.2.2.2).

(3) A *computer* that executes programs in a high level programming language. Its non-differentiable memory enables *abstract*, *scalable* and *precise* operations, but makes training more challenging (Section 4.2.2.3). To help the "programmer" prune the search space, it provides a friendly *neural computer interface*, which detects and eliminates invalid choices (Section 4.2.2.1).

Within this framework, we introduce the Neural Symbolic Machine (NSM) and apply it to semantic parsing. NSM contains a "computer", a non-differentiable Lisp interpreter, which executes programs against a large KB and provides code assistance (Section 4.2.2.1), and a seq2seq model ("programmer"), which supports compositionality using a key-variable memory to save and reuse intermediate results (Section 4.2.2.2). We also propose a training

procedure that is based on REINFORCE, but is augmented with pseudo-gold programs found by an iterative ML training procedure (Section 4.2.2.3).

Before diving into details, we define the *semantic parsing* task: given a knowledge base $\mathbb{K}$, and a question $x = (w_1, w_2, ..., w_m)$, produce a program or logical form z that when executed against $\mathbb{K}$ generates the right answer y. Let $\mathcal{E}$ denote a set of entities (e.g., AbeLincoln),[1] and let $\mathcal{P}$ denote a set of properties (e.g., PlaceOfBirth). A knowledge base $\mathbb{K}$ is a set of assertions or triples $(e_1, p, e_2) \in \mathcal{E} \times \mathcal{P} \times \mathcal{E}$, such as (AbeLincoln, PlaceOfBirth, Hodgenville).

4.2.2.1. Computer: Lisp Interpreter with Code Assistance. Semantic parsing typically requires using a set of operations to query the knowledge base and process the results. Operations learned with neural networks such as addition and sorting do not perfectly generalize to inputs that are larger than the ones observed in the training data [**45, 109**]. In contrast, operations implemented in high level programming languages are *abstract*, *scalable*, and *precise*, thus generalizes perfectly to inputs of arbitrary size. Based on this observation, we implement operations necessary for semantic parsing with an ordinary programming language instead of trying to learn them with a neural network.

$(\ \mathit{Hop}\ r\ p\ ) \Rightarrow \{e_2 \vert e_1 \in r, (e_1, p, e_2) \in \mathbb{K}\}$
$(\ \mathit{ArgMax}\ r\ p\ ) \Rightarrow \{e_1 \vert e_1 \in r, \exists e_2 \in \mathcal{E} : (e_1, p, e_2) \in \mathbb{K}, \forall e : (e_1, p, e) \in \mathbb{K}, e_2 \geq e\}$
$(\ \mathit{ArgMin}\ r\ p\ ) \Rightarrow \{e_1 \vert e_1 \in r, \exists e_2 \in \mathcal{E} : (e_1, p, e_2) \in \mathbb{K}, \forall e : (e_1, p, e) \in \mathbb{K}, e_2 \leq e\}$
$(\ \mathit{Filter}\ r_1\ r_2\ p\ ) \Rightarrow \{e_1 \vert e_1 \in r_1, \exists e_2 \in r_2 : (e_1, p, e_2) \in \mathbb{K}\}$

Table 4.1. Interpreter functions. r represents a variable, p a property in Freebase. $\geq$ and $\leq$ are defined on numbers and dates.

We adopt a Lisp interpreter as the "computer". A program C is a list of expressions $(c_1...c_N)$, where each expression is either a special token "*Return*" indicating the end of the program, or a list of tokens enclosed by parentheses "$(FA_1...A_K)$". F is a function, which takes as input K arguments of specific types. Table 4.8 defines the semantics of each function and the types of its arguments (either a property p or a variable r). When a function is

[1]We also consider numbers (e.g., "1.33") and date-times (e.g., "1999-1-1") as entities.

executed, it returns an entity list that is the expression's denotation in $\mathbb{K}$, and save it to a new variable.

By introducing variables that save the intermediate results of execution, the program naturally models *language compositionality* and describes from left to right a bottom-up derivation of the full meaning of the natural language input, which is convenient in a seq2seq model (Figure 4.1). This is reminiscent of the floating parser [**139, 99**], where a derivation tree that is not grounded in the input is incrementally constructed.

The set of programs defined by our functions is equivalent to the subset of λ-calculus presented in [**148**]. We did not use full Lisp programming language here, because constructs like control flow and loops are unnecessary for most current semantic parsing tasks, and it is simple to add more functions to the model when necessary.

To create a friendly "neural computer interface", the interpreter provides code assistance to the programmer by producing a list of valid tokens at each step. First, a valid token should not cause a syntax error: e.g., if the previous token is "(", the next token must be a function name, and if the previous token is "*Hop*", the next token must be a variable. More importantly, a valid token should not cause a semantic (run-time) error: this is detected using the denotation saved in the variables. For example, if the previously generated tokens were "(*Hop* r", the next available token is restricted to properties $\{p \mid \exists e, e' : e \in r, (e, p, e') \in \mathbb{K}\}$ that are reachable from entities in r in the KB. These checks are enabled by the variables and can be derived from the definition of the functions in Table 4.8. The interpreter prunes the "programmer"'s search space by orders of magnitude, and enables learning from weak supervision on a large KB.

4.2.2.2. Programmer: Seq2seq Model with Key-Variable Memory. Given the "computer", the "programmer" needs to map natural language into a program, which is a sequence of tokens that reference operations and values in the "computer". We base our programmer on a standard seq2seq model with attention, but extend it with a key-variable memory that allows the model to learn to represent and refer to program variables (Figure 4.2).

Sequence-to-sequence models consist of two RNNs, an encoder and a decoder. We used a 1-layer GRU [**20**] for both the encoder and decoder. Given a sequence of words $w_1, w_2...w_m$, each word w_t is mapped to an embedding q_t (embedding details are in Section 4.3.5). Then, the encoder reads these embeddings and updates its hidden state step by step using $h_{t+1} = GRU(h_t, q_t, \theta_{Encoder})$, where $\theta_{Encoder}$ are the GRU parameters. The decoder updates its hidden states u_t by $u_{t+1} = GRU(u_t, c_{t-1}, \theta_{Decoder})$, where c_{t-1} is the embedding of last step's output token a_{t-1}, and $\theta_{Decoder}$ are the GRU parameters. The last hidden state of the encoder h_T is used as the decoder's initial state. We also adopt a dot-product attention similar to [**31**]. The tokens of the program $a_1, a_2...a_n$ are generated one by one using a softmax over the vocabulary of valid tokens at each step, as provided by the "computer" (Section 4.2.2.1).

To achieve compositionality, the decoder must learn to represent and refer to intermediate variables whose value was saved in the "computer" after execution. Therefore, we augment the model with a **key-variable memory**, where each entry has two components: a continuous embedding key v_i, and a corresponding variable token R_i referencing the value in the "computer" (see Figure 4.2). During encoding, we use an entity linker to link text spans (e.g., *"US"*) to KB entities. For each linked entity we add a memory entry where the key is the average of GRU hidden states over the entity span, and the variable token (R_1) is the name of a variable in the computer holding the linked entity (*m.USA*) as its value. During decoding, when a full expression is generated (i.e., the decoder generates ")"), it gets executed, and the result is stored as the value of a new variable in the "computer". This variable is keyed by the GRU hidden state at that step. When a new variable R_1 with key embedding v_1 is added into the key-variable memory, the token R_1 is added into the decoder vocabulary with v_1 as its embedding. The final answer returned by the "programmer" is the value of the last computed variable.

Similar to pointer networks [**136**], the key embeddings for variables are dynamically generated for each example. During training, the model learns to represent variables by backpropagating gradients from a time step where a variable is selected by the decoder,

through the key-variable memory, to an earlier time step when the key embedding was computed. Thus, the encoder/decoder learns to generate representations for variables such that they can be used at the right time to construct the correct program.

While the key embeddings are differentiable, the values referenced by the variables (lists of entities), stored in the "computer", are symbolic and non-differentiable. This distinguishes the key-variable memory from other memory-augmented neural networks that use continuous differentiable embeddings as the values of memory entries [**142, 46**].

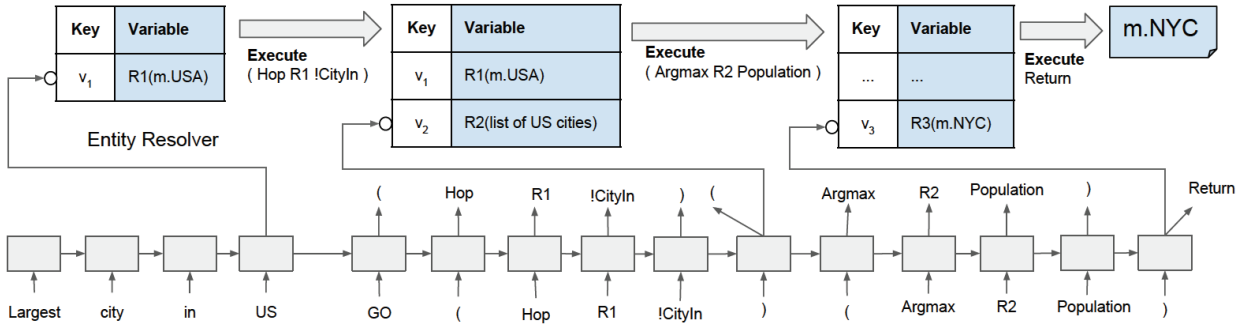

Figure 4.2. Semantic Parsing with NSM. The key embeddings of the key-variable memory are the output of the sequence model at certain encoding or decoding steps. For illustration purposes, we also show the values of the variables in parentheses, but the sequence model never sees these values, and only references them with the name of the variable ("R_1"). A special token "*GO*" indicates the start of decoding, and "*Return*" indicates the end of decoding.

4.2.2.3. Training NSM with Weak Supervision. NSM executes non-differentiable operations against a KB, and thus end-to-end backpropagation is not possible. Therefore, we base our training procedure on REINFORCE [**143, 94**]. When the reward signal is sparse and the search space is large, it is common to utilize some full supervision to pre-train REINFORCE [**124**]. To train from weak supervision, we suggest an iterative ML procedure for finding pseudo-gold programs that will bootstrap REINFORCE.

REINFORCE We can formulate training as a reinforcement learning problem: given a question x, the state, action and reward at each time step $t \in \{0, 1, ..., T\}$ are (s_t, a_t, r_t). Since the environment is deterministic, the state is defined by the question x and the action sequence: $s_t = (x, a_{0:t-1})$, where $a_{0:t-1} = (a_0, ..., a_{t-1})$ is the history of actions at time t. A valid action at time t is $a_t \in A(s_t)$, where $A(s_t)$ is the set of valid tokens given by the

"computer". Since each action corresponds to a token, the full history $a_{0:T}$ corresponds to a program. The reward $r_t = I[t = T] \cdot F_1(x, a_{0:T})$ is non-zero only at the last step of decoding, and is the F_1 score computed comparing the gold answer and the answer generated by executing the program $a_{0:T}$. Thus, the cumulative reward of a program $a_{0:T}$ is

$$R(x, a_{0:T}) = \sum_t r_t = F_1(x, a_{0:T}).$$

The agent's decision making procedure at each time is defined by a policy, $\pi_\theta(s, a) = P_\theta(a_t = a|x, a_{0:t-1})$, where θ are the model parameters. Since the environment is deterministic, the probability of generating a program $a_{0:T}$ is

$$P_\theta(a_{0:T}|x) = \prod_t P_\theta(a_t \mid x, a_{0:t-1}).$$

We can define our objective to be the expected cumulative reward and use policy gradient methods such as REINFORCE for training. The objective and gradient are:

$$J^{RL}(\theta) = \sum_x \mathbb{E}_{P_\theta(a_{0:T}|x)}[R(x, a_{0:T})] \tag{4.1}$$

$$\nabla_\theta J^{RL}(\theta) = \sum_x \sum_{a_{0:T}} P_\theta(a_{0:T} \mid x) \cdot [R(x, a_{0:T}) - B(x)] \cdot \nabla_\theta \log P_\theta(a_{0:T} \mid x) \tag{4.2}$$

where $B(x) = \sum_{a_{0:T}} P_\theta(a_{0:T} \mid x) R(x, a_{0:T})$ is a baseline that reduces the variance of the gradient estimation without introducing bias. Having a separate network to predict the baseline is an interesting future direction.

While REINFORCE assumes a stochastic policy, we use beam search for gradient estimation. Thus, in contrast with common practice of approximating the gradient by sampling from the model, we use the top-k action sequences (programs) in the beam with normalized probabilities. This allows training to focus on sequences with high probability, which are on the decision boundaries, and reduces the variance of the gradient.

Empirically (and in line with prior work), REINFORCE converged slowly and often got stuck in local optima (see Section 4.3.5). The difficulty of training resulted from the sparse reward signal in the large search space, which caused model probabilities for programs with

non-zero reward to be very small at the beginning. If the beam size k is small, good programs fall off the beam, leading to zero gradients for all programs in the beam. If the beam size k is large, training is very slow, and the normalized probabilities of good programs when the model is untrained are still very small, leading to (1) near zero baselines, thus near zero gradients on "bad" programs (2) near zero gradients on good programs due to the low probability $P_\theta(a_{0:T} \mid x)$. To combat this, we present an alternative training strategy based on maximum-likelihood.

Iterative ML If we had gold programs, we could directly optimize their likelihood. Since we do not have gold programs, we can perform an iterative procedure (similar to hard Expectation-Maximization (EM)), where we search for good programs given fixed parameters, and then optimize the probability of the best program found so far. We do decoding on an example with a large beam size and declare $a^{best}_{0:T}(x)$ to be the pseudo-gold program, which achieved highest reward with shortest length among the programs decoded on x in all previous iterations. Then, we can optimize the ML objective:

$$J^{ML}(\theta) = \sum_x \log P_\theta(a^{best}_{0:T}(x) \mid x) \tag{4.3}$$

A question x is not included if we did not find any program with positive reward.

Training with iterative ML is fast because there is at most one program per example and the gradient is not weighted by model probability. while decoding with a large beam size is slow, we could train for multiple epochs after each decoding. This iterative process has a bootstrapping effect that a better model leads to a better program $a^{best}_{0:T}(x)$ through decoding, and a better program $a^{best}_{0:T}(x)$ leads to a better model through training.

Even with a large beam size, some programs are hard to find because of the large search space. A common solution to this problem is to use curriculum learning [**154, 109**]. The size of the search space is controlled by both the set of functions used in the program and the program length. We apply curriculum learning by gradually increasing both these quantities (see details in Section 4.3.5) when performing iterative ML.

Nevertheless, iterative ML uses only pseudo-gold programs and does not directly optimize the objective we truly care about. This has two adverse effects: (1) The best program $a_{0:T}^{best}(x)$ could be a spurious program that accidentally produces the correct answer (e.g., using the property PlaceOfBirth instead of PlaceOfDeath when the two places are the same), and thus does not generalize to other questions. (2) Because training does not observe full negative programs, the model often fails to distinguish between tokens that are related to one another. For example, differentiating ParentsOf vs. SiblingsOf vs. ChildrenOf can be challenging. We now present learning where we combine iterative ML with REINFORCE.

Augmented REINFORCE To bootstrap REINFORCE, we can use iterative ML to find pseudo-gold programs, and then add these programs to the beam with a reasonably large probability. This is similar to methods from imitation learning [**112, 58**] that define a proposal distribution by linearly interpolating the model distribution and an oracle.

Algorithm 3 IML-REINFORCE

Input: question-answer pairs $\mathbb{D} = \{(x_i, y_i)\}$, mix ratio α, reward function $R(\cdot)$, training iterations N_{ML}, N_{RL}, and beam sizes B_{ML}, B_{RL}.
Procedure:
Initialize $C_x^* = \emptyset$ the best program so far for x
Initialize model θ randomly ▷ Iterative ML
for $n = 1$ to N_{ML} **do**
  for (x, y) in D **do**
    $\mathbb{C} \leftarrow$ Decode B_{ML} programs given x
    for j in 1...$|\mathbb{C}|$ **do**
      if $R_{x,y}(C_j) > R_{x,y}(C_x^*)$ **then** $C_x^* \leftarrow C_j$
  $\theta \leftarrow$ ML training with $\mathbb{D}_{ML} = \{(x, C_x^*)\}$
Initialize model θ randomly ▷ REINFORCE
for $n = 1$ to N_{RL} **do**
  $\mathbb{D}_{RL} \leftarrow \emptyset$ is the RL training set
  for (x, y) in D **do**
    $\mathbb{C} \leftarrow$ Decode B_{RL} programs from x
    for j in 1...$|\mathbb{C}|$ **do**
      if $R_{x,y}(C_j) > R_{x,y}(C_x^*)$ **then** $C_x^* \leftarrow C_j$
    $\mathbb{C} \leftarrow \mathbb{C} \cup \{C_x^*\}$
    for j in 1...$|\mathbb{C}|$ **do**
      $\hat{p}_j \leftarrow (1 - \alpha) \cdot \frac{p_j}{\sum_{j'} p_{j'}}$ where $p_j = P_\theta(C_j \mid x)$
      if $C_j = C_x^*$ **then** $\hat{p}_j \leftarrow \hat{p}_j + \alpha$
      $\mathbb{D}_{RL} \leftarrow \mathbb{D}_{RL} \cup \{(x, C_j, \hat{p}_j)\}$
  $\theta \leftarrow$ REINFORCE training with $\mathbb{D}_{RL}$

Algorithm 3 describes our overall training procedure. We first run iterative ML for N_{ML} iterations and record the best program found for every example x_i. Then, we run REINFORCE, where we normalize the probabilities of the programs in beam to sum to $(1-\alpha)$ and add α to the probability of the best found program $C^*(x_i)$. Consequently, the model always puts a reasonable amount of probability on a program with high reward during training. Note that we randomly initialized the parameters for REINFORCE, since initializing from the final ML parameters seems to get stuck in a local optimum and produced worse results.

On top of imitation learning, our approach is related to the common practice in reinforcement learning [**117**] to replay rare successful experiences to reduce the training variance and improve training efficiency. This is also similar to recent developments [**146**] in machine translation, where ML and RL objectives are linearly combined, because anchoring the model to some high-reward outputs stabilizes training.

4.2.3. Experiments and Analysis

We now empirically show that NSM can learn a semantic parser from weak supervision over a large KB. We evaluate on WebQuestionsSP, a challenging semantic parsing dataset with strong baselines. Experiments show that NSM achieves new state-of-the-art performance on WebQuestionsSP with weak supervision, and significantly closes the gap between weak and full supervision for this task.

4.2.3.1. The WebQuestionsSP dataset. The WebQuestionsSP dataset [**149**] contains full semantic parses for a subset of the questions from WebQuestions [**12**], because 18.5% of the original dataset were found to be "not answerable". It consists of 3,098 question-answer pairs for training and 1,639 for testing, which were collected using Google Suggest API, and the answers were originally obtained using Amazon Mechanical Turk workers. They were updated in [**149**] by annotators who were familiar with the design of Freebase and added semantic parses. We further separated out 620 questions from the training set as a validation set. For query pre-processing, we used Google's internal named entity linking system to find

the entities in a question. The quality of the entity linker is similar to that of [**148**] at 94% of the gold root entities being included. Similar to [**31**], we replaced named entity tokens with a special token "*ENT*". For example, the question "*who plays meg in family guy*" is changed to "*who plays ENT in ENT ENT*". This helps reduce overfitting, because instead of memorizing the correct program for a specific entity, the model has to focus on other context words in the sentence, which improves generalization.

Following [**148**] we used the last publicly available snapshot of Freebase [**14**]. Since NSM training requires random access to Freebase during decoding, we preprocessed Freebase by removing predicates that are not related to world knowledge (starting with "*/common/*", "*/type/*", "*/freebase/*"),[2] and removing all text valued predicates, which are rarely the answer. Out of all 27K relations, 434 relations are removed during preprocessing. This results in a graph that fits in memory with 23K relations, 82M nodes, and 417M edges.

4.2.3.2. Model Details. For pre-trained word embeddings, we used the 300 dimension GloVe word embeddings trained on 840B tokens [**103**]. On the encoder side, we added a projection matrix to transform the embeddings into 50 dimensions. On the decoder side, we used the same GloVe embeddings to construct an embedding for each property using its Freebase id, and also added a projection matrix to transform this embedding to 50 dimensions. A Freebase id contains three parts: domain, type, and property. For example, the Freebase id for ParentsOf is *"/people/person/parents"*. *"people"* is the domain, *"person"* is the type and *"parents"* is the property. The embedding is constructed by concatenating the average of word embeddings in the domain and type name to the average of word embeddings in the property name. For example, if the embedding dimension is 300, the embedding dimension for *"/people/person/parents"* will be 600. The first 300 dimensions will be the average of the embeddings for *"people"* and *"person"*, and the second 300 dimensions will be the embedding for *"parents"*.

The dimension of encoder hidden state, decoder hidden state and key embeddings are all 50. The embeddings for the functions and special tokens (e.g., "*UNK*", "*GO*") are

[2]We kept "*/common/topic/notable_types*".

randomly initialized by a truncated normal distribution with mean=0.0 and stddev=0.1. All the weight matrices are initialized with a uniform distribution in $[-\frac{\sqrt{3}}{d}, \frac{\sqrt{3}}{d}]$ where d is the input dimension. Dropout rate is set to 0.5, and we see a clear tendency for larger dropout rate to produce better performance, indicating overfitting is a major problem for learning.

4.2.3.3. Training Details. In iterative ML training, the decoder uses a beam of size $k = 100$ to update the pseudo-gold programs and the model is trained for 20 epochs after each decoding step. We use the Adam optimizer [**63**] with initial learning rate 0.001. In our experiment, this process usually converges after a few (5-8) iterations.

For REINFORCE training, the best hyperparameters are chosen using the validation set. We use a beam of size $k = 5$ for decoding, and α is set to 0.1. Because the dataset is small and some relations are only used once in the whole training set, we train the model on the entire training set for 200 iterations with the best hyperparameters. Then we train the model with learning rate decay until convergence. Learning rate is decayed as $g_t = g_0 \times \beta^{\frac{\max(0, t-t_s)}{m}}$, where $g_0 = 0.001$, $\beta = 0.5$ $m = 1000$, and t_s is the number of training steps at the end of iteration 200.

Since decoding needs to query the knowledge base (KB) constantly, the speed bottleneck for training is decoding. We address this problem in our implementation by partitioning the dataset, and using multiple decoders in parallel to handle each partition. We use 100 decoders, which queries 50 KG servers, and one trainer. The neural network model is implemented in TensorFlow. Since the model is small, we didn't see a significant speedup by using GPU, so all the decoders and the trainer are using CPU only.

Inspired by the staged generation process in [**148**], curriculum learning includes two steps. We first run iterative ML for 10 iterations with programs constrained to only use the "*Hop*" function and the maximum number of expressions is 2. Then, we run iterative ML again, but use both "*Hop*" and "*Filter*". The maximum number of expressions is 3, and the relations used by "*Hop*" are restricted to those that appeared in $a^{best}_{0:T}(q)$ in the first step.

4.2.3.4. Results and discussion. We evaluate performance using the offical evaluation script for WebQuestionsSP. Because the answer to a question may contain multiple entities

or values, precision, recall and F1 are computed based on the output of each individual question, and average F1 is reported as the main evaluation metric. Accuracy measures the proportion of questions that are answered exactly.

A comparison to STAGG, the previous state-of-the-art model [**149, 148**], is shown in Table 4.2. Our model beats STAGG with weak supervision by a significant margin on all metrics, while relying on no feature engineering or hand-crafted rules. When STAGG is trained with strong supervision it obtains an F1 of 71.7, and thus NSM closes half the gap between training with weak and full supervision.

Model	Prec.	Rec.	F1	Acc.
STAGG	67.3	73.1	66.8	58.8
NSM	70.8	76.0	**69.0**	59.5

Table 4.2. Results on the test set. Average F1 is the main evaluation metric and NSM outperforms STAGG with no domain-specific knowledge or feature engineering.

Four key ingredients lead to the final performance of NSM. The first one is the neural computer interface that provides code assistance by checking for syntax and semantic errors. We find that semantic checks are very effective for open-domain KBs with a large number of properties. For our task, the average number of choices is reduced from 23K per step (all properties) to less than 100 (the average number of properties connected to an entity).

The second ingredient is augmented REINFORCE training. Table 4.3 compares augmented REINFORCE, REINFORCE, and iterative ML on the validation set. REINFORCE gets stuck in local optimum and performs poorly. Iterative ML training is not directly optimizing the F1 measure, and achieves sub-optimal results. In contrast, augmented REINFORCE is able to bootstrap using pseudo-gold programs found by iterative ML and achieves the best performance on both the training and validation set.

The third ingredient is curriculum learning during iterative ML. We compare the performance of the best programs found with and without curriculum learning in Table 4.4. We find that the best programs found with curriculum learning are substantially better than those found without curriculum learning by a large margin on every metric.

Settings	Train F1	Valid F1
Iterative ML	68.6	60.1
REINFORCE	55.1	47.8
Augmented REINFORCE	83.0	**67.2**

Table 4.3. Average F1 on the validation set for augmented REINFORCE, REINFORCE, and iterative ML.

Settings	Prec.	Rec.	F1	Acc.
No curriculum	79.1	91.1	78.5	67.2
Curriculum	88.6	96.1	89.5	79.8

Table 4.4. Evaluation of the programs with the highest F1 score in the beam ($a_{0:t}^{best}$) with and without curriculum learning.

The last important ingredient is reducing overfitting. Given the small size of the dataset, overfitting is a major problem for training neural network models. We show the contributions of different techniques for controlling overfitting in Table 4.5. Note that after all the techniques have been applied, the model is still overfitting with training F1@1=83.0% and validation F1@1=67.2%.

Settings	Δ F1@1
$-$*Pretrained word embeddings*	-5.5
$-$*Pretrained property embeddings*	-2.7
$-$*Dropout on GRU input and output*	-2.4
$-$*Dropout on softmax*	-1.1
$-$*Anonymize entity tokens*	-2.0

Table 4.5. Contributions of different overfitting techniques on the validation set.

#Expressions	0	1	2	3
Percentage	0.4%	62.9%	29.8%	6.9%
F1	0.0	73.5	59.9	70.3

Table 4.6. Percentage and performance of model generated programs with different complexity (number of expressions).

Among the programs generated by the model, a significant portion (36.7%) uses more than one expression. From Table 4.6, we can see that the performance doesn't decrease

much as the compositional depth increases, indicating that the model is effective at capturing compositionality. We observe that programs with three expressions use a more limited set of properties, mainly focusing on answering a few types of questions such as "who plays meg in family guy", "what college did jeff corwin go to" and "which countries does russia border". In contrast, programs with two expressions use a more diverse set of properties, which could explain the lower performance compared to programs with three expressions.

Error analysis on the validation set shows two main sources of errors:

(1) **Search failure**: Programs with high reward are not found during search for pseudo-gold programs, either because the beam size is not large enough, or because the set of functions implemented by the interpreter is insufficient. The 89.5% F1 score in Table 4.4 indicates that at least 10% of the questions are of this kind.

(2) **Ranking failure**: Programs with high reward exist in the beam, but are not ranked at the top during decoding. Because the training error is low, this is largely due to overfitting or spurious programs. The 67.2% F1 score in Table 4.3 indicates that about 20% of the questions are of this kind.

4.2.4. Related work

Among deep learning models for program induction, Reinforcement Learning Neural Turing Machines (RL-NTMs) [**154**] are the most similar to NSM, as a non-differentiable machine is controlled by a sequence model. Therefore, both models rely on REINFORCE for training. The main difference between the two is the abstraction level of the programming language. RL-NTM uses lower level operations such as memory address manipulation and byte reading/writing, while NSM uses a high level programming language over a large knowledge base that includes operations such as following properties from entities, or sorting based on a property, which is more suitable for representing semantics. Earlier works such as OOPS [**118**] has desirable characteristics, for example, the ability to define new functions. These remain to be future improvements for NSM.

We formulate NSM training as an instance of reinforcement learning [**133**] in order to directly optimize the task reward of the structured prediction problem [**94, 69, 152**]. Compared to imitation learning methods [**27, 112**] that interpolate a model distribution with an oracle, NSM needs to solve a challenging search problem of training from weak supervisions in a large search space. Our solution employs two techniques (a) a symbolic "computer" helps find good programs by pruning the search space (b) an iterative ML training process, where beam search is used to find pseudo-gold programs. Wiseman and Rush [**144**] proposed a max-margin approach to train a sequence-to-sequence scorer. However, their training procedure is more involved, and we did not implement it in this work. MIXER [**108**] also proposed to combine ML training and REINFORCE, but they only considered tasks with full supervisions. Berant and Liang [**13**] applied imitation learning to semantic parsing, but still requires hand crafted grammars and features.

NSM is similar to Neural Programmer [**91**] and Dynamic Neural Module Network [**4**] in that they all solve the problem of semantic parsing from structured data, and generate programs using similar semantics. The main difference between these approaches is how an intermediate result (the memory) is represented. Neural Programmer and Dynamic-NMN chose to represent results as vectors of weights (row selectors and attention vectors), which enables backpropagation and search through all possible programs in parallel. However, their strategy is not applicable to a large KB such as Freebase, which contains about 100M entities, and more than 20k properties. Instead, NSM chooses a more scalable approach, where the "computer" saves intermediate results, and the neural network only refers to them with variable names (e.g., "R_1" for all cities in the US).

NSM is similar to the Path Ranking Algorithm (PRA) [**66**] in that semantics is encoded as a sequence of actions, and denotations are used to prune the search space during learning. NSM is more powerful than PRA by 1) allowing more complex semantics to be composed through the use of a key-variable memory; 2) controlling the search procedure with a trained neural network, while PRA only samples actions uniformly; 3) allowing input questions to express complex relations, and then dynamically generating action sequences. PRA can

combine multiple semantic representations to produce the final prediction, which remains to be future work for NSM.

4.2.5. Conclusion

We proposed the Manager-Programmer-Computer framework for neural program induction. It integrates neural networks with a symbolic *non-differentiable* computer to support *abstract*, *scalable* and *precise* operations through a friendly *neural computer interface*. Within this framework, we introduce the Neural Symbolic Machine, which integrates a neural sequence-to-sequence "programmer" with key-variable memory, and a symbolic Lisp interpreter with code assistance. Because the interpreter is non-differentiable and to directly optimize the task reward, we apply REINFORCE and use pseudo-gold programs found by an iterative ML training process to bootstrap training. NSM achieves new state-of-the-art results on a challenging semantic parsing dataset with weak supervision, and significantly closes the gap between weak and full supervision. It is trained end-to-end, and does not require any feature engineering or domain-specific knowledge.

4.3. Program Synthesis with Generalization

In the previous section, we applied Neural Symbolic Machines to semantic parsing over Freebase. However, the main challenge in WebQuestionsSP is to match natural language with the correct predicates in a large schema, and the length of the programs and the set of functions used in the programs are limited [**100**]. It remains challenging to apply NSM and RL to complex program synthesis or compositional semantic parsing tasks [**100, 90**], which need to generate (given a natural language utterance) longer programs that use a large set of functions. Previous work under the RL paradigm [**48, 90**] has shown that simply applying on-policy RL methods like REINFORCE is usually not enough, so successfully applying NSM to complex program synthesis and compositional semantic parsing requires more efficient RL methods.

In this section, we propose Memory Augmented Policy Optimization (MAPO): a novel policy optimization formulation that incorporates a memory buffer of promising trajectories to reduce the variance of policy gradient estimates for deterministic environments with discrete actions. The formulation expresses the expected return objective as a weighted sum of two terms: an expectation over a memory of trajectories with high rewards, and a separate expectation over the trajectories outside the memory. We propose 3 techniques to make an efficient training algorithm for MAPO: (1) distributed sampling from inside and outside memory with an actor-learner architecture; (2) a marginal likelihood constraint over the memory to accelerate training; (3) systematic exploration to discover high reward trajectories. MAPO improves the sample efficiency and robustness of policy gradient, especially on tasks with a sparse reward. We evaluate MAPO on *weakly supervised* program synthesis from *natural language* with an emphasis on *generalization*. On the WikiTableQuestions benchmark we improve the state-of-the-art by 2.5%, achieving an accuracy of 46.2%, and on the WikiSQL benchmark, MAPO achieves an accuracy of 74.9% with only weak supervision, outperforming several strong baselines with full supervision. Our code is open sourced at `https://github.com/crazydonkey200/neural-symbolic-machines`.

4.3.1. Introduction

There has been a recent surge of interest in applying policy gradient methods to various application domains including program synthesis [**71, 48, 161, 18**], dialogue generation [**69, 25**], architecture search [**162, 164**], game [**125, 82**] and continuous control [**104, 120**]. Simple policy gradient methods like REINFORCE [**143**] use Monte Carlo samples from the current policy to perform an *on-policy* optimization of the expected return. This often leads to unstable learning dynamics and poor sample efficiency, sometimes underperforming random search [**77**].

The difficulty of gradient based policy optimization stems from a few sources: (1) policy gradient estimates have a large *variance*; (2) samples from a randomly initialized policy often attain small rewards, resulting in a slow training progress in the initial phase; (3) random

policy samples do not explore the search space efficiently because many samples can be repeated. These issues can be especially prohibitive in applications such as program synthesis and robotics [**5**], which involve a large search space with sparse rewards. In such tasks, a high reward is achieved only after a long sequence of *correct* actions. For instance, in program synthesis, only a few programs in the large program space lead to the correct functional form. Unfortunately, this often leads to forgetting a high reward trajectory unless it is re-sampled frequently [**71, 3**].

This paper presents MAPO: a novel formulation of policy optimization for deterministic environments with discrete actions, which incorporates a memory buffer of promising trajectories within the policy gradient framework. It estimates the policy gradient as a weighted sum of an expectation over the trajectories inside the memory and a separate expectation over those outside the memory. The gradient estimates are unbiased and attain lower variance provided that trajectories in the memory buffer have non-negligible probability. Because high-reward trajectories remain in the memory, it is almost impossible to forget them. To make an efficient algorithm for MAPO, we propose 3 techniques: (1) distributed sampling from inside and outside memory buffer in an actor-learner architecture; (2) a constraint on the marginal likelihood of the high-reward trajectories in the memory buffer to accelerate training at the cost of introducing some bias at the initial training stage;(3) systematic exploration of the search space to efficiently discover the high-reward trajectories.

We assess the effectiveness of MAPO on *weakly supervised* program synthesis from *natural language* (see Section 4.3.2). Program synthesis presents a unique opportunity to study *generalization* in the context of policy optimization, while having an impact on a real world application. On the challenging WikiTableQuestions [**100**] benchmark, MAPO achieves an accuracy of 46.2% on the test set, significantly outperforming the previous state-of-the-art of 43.7% [**160**]. Interestingly, on the WikiSQL [**161**] benchmark, MAPO achieves an accuracy of 74.9% without the supervision of gold programs, outperforming several strong *fully supervised* baselines.

Year	Venue	Position	Event	Time
2001	Hungary	2nd	400m	47.12
2003	Finland	1st	400m	46.69
2005	Germany	11th	400m	46.62
2007	Thailand	1st	relay	182.05
2008	China	7th	relay	180.32

Table 4.7. **x**: Where did the last 1st place finish occur? **y**: Thailand

4.3.2. The Problem of Weakly Supervised Contextual Program Synthesis

Consider the problem of learning to map a natural language question $\mathbf{x}$ to a structured query $\mathbf{a}$ in a programming language such as SQL (*e.g.*, [**161**]), or converting a textual problem description into a piece of source code, *e.g.*, in python, as in programming competitions (*e.g.*, [**7**]). We call this family of problems *contextual program synthesis* and aim at tackling such problems in a weakly supervised setting – i.e., no correct action sequence $\mathbf{a}$ is given as part of the training data, and training needs to solve the hard problem of exploring a large program space. Table 4.7 shows an example question-answer pair. The model needs to first discover the programs that can generate the correct answer in a given context, and then learn to generalize to new contexts.

We formulate the problem of *weakly supervised* contextual program synthesis as follows: to generate a program by using a parametric mapping function, $\hat{\mathbf{a}} = f(\mathbf{x}; \theta)$, where θ denotes the model parameters. The quality of a generated program $\hat{\mathbf{a}}$ is measured in terms of a scoring or *reward* function $R(\hat{\mathbf{a}} \mid \mathbf{x}, \mathbf{y})$. The reward function may evaluate a program by executing it on a real environment and comparing the emitted output against the correct answer. For example, it is natural to define a binary reward that is 1 when the output equals the answer and 0 otherwise. We assume that the context $\mathbf{x}$ includes both a natural language input and an environment, for example an interpreter or a database, on which the program will be executed. Given a dataset of context-answer pairs, $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$, the goal is to find an optimal parameter θ^* that parameterizes a mapping of $\mathbf{x} \rightarrow \mathbf{a}$ with maximum empirical return on a *heldout test set*.

One can think of the problem of contextual program synthesis as an instance of *reinforcement learning (RL)* with *sparse terminal rewards* and *deterministic dynamics*, for which *generalization* plays a key role. There has been some recent attempts in the RL community to study generalization to unseen initial conditions (*e.g.* [**107, 93**]). However, most prior work aims to maximize empirical return on the training environment [**9, 17**]. The problem of contextual program synthesis presents a natural application of RL for which generalization is the main concern.

4.3.3. Optimization of Expected Return via Policy Gradients

To learn a mapping of (context $\mathbf{x}$) $\rightarrow$ (program $\mathbf{a}$), we optimize the parameters of a conditional distribution $\pi_\theta(\mathbf{a} \mid \mathbf{x})$ that assigns a probability to each program given the context. That is, π_θ is a distribution over the *countable* set of all possible programs, denoted $\mathcal{A}$. Thus $\forall \mathbf{a} \in \mathcal{A}: \ \pi_\theta(\mathbf{a} \mid \mathbf{x}) \geq 0$ and $\sum_{\mathbf{a}\in\mathcal{A}} \pi_\theta(\mathbf{a} \mid \mathbf{x}) = 1$. Then, to synthesize a program for a novel context, one finds the most likely program under the distribution π_θ via exact or approximate inference $\hat{\mathbf{a}} \approx \text{argmax}_{\mathbf{a}\in\mathcal{A}} \pi_\theta(\mathbf{a} \mid \mathbf{x})$.

Autoregressive models present a tractable family of distributions that estimates the probability of a sequence of tokens, one token at a time, often from left to right. To handle variable sequence length, one includes a special *end-of-sequence* token at the end of the sequences. We express the probability of a program $\mathbf{a}$ given $\mathbf{x}$ as $\pi_\theta(\mathbf{a} \mid \mathbf{x}) \equiv \prod_{i=t}^{|\mathbf{a}|} \pi_\theta(a_t \mid \mathbf{a}_{<t}, \mathbf{x})$,where $\mathbf{a}_{<t} \equiv (a_1, \ldots, a_{t-1})$ denotes a prefix of the program $\mathbf{a}$. One often uses a recurrent neural network (*e.g.* [**53**]) to predict the probability of each token given the prefix and the context.

In the absence of ground truth programs, policy gradient techniques present a way to optimize the parameters of a stochastic policy π_θ via optimization of *expected return*. Given a training dataset of context-answer pairs, $\mathcal{D} \equiv \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, the objective is expressed as $\mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{x})} R(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$. The reward function $R(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$ evaluates a complete program $\mathbf{a}$, based on the context $\mathbf{x}$ and the correct answer $\mathbf{y}$. These assumptions characterize the problem of program synthesis well, but they also apply to many other discrete optimization domains.

Note that these assumptions are required for the exact unbiased gradients, but we expect the MAPO algorithm to benefit many discrete RL benchmarks on which these assumptions apply to a big extent.

Simplified notation. In what follows, we simplify the notation by dropping the dependence of the policy and the reward on $\mathbf{x}$ and $\mathbf{y}$. We use a notation of $\pi_\theta(\mathbf{a})$ instead of $\pi_\theta(\mathbf{a} \mid \mathbf{x})$ and $R(\mathbf{a})$ instead of $R(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$, to make the formulation less cluttered, but the equations hold in the general case.

We express the expected return objective in the simplified notation as,

$$\mathcal{O}_{\mathrm{ER}}(\theta) \;=\; \sum_{\mathbf{a}\in\mathcal{A}} \pi_\theta(\mathbf{a})\, R(\mathbf{a}) \;=\; \mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a})}\, R(\mathbf{a}) \;. \tag{4.4}$$

The REINFORCE [**143**] algorithm presents an elegant and convenient way to estimate the gradient of the expected return (4.4) using Monte Carlo (MC) samples. Using K trajectories sampled *i.i.d.* from the current policy π_θ, denoted $\{\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(K)}\}$, the gradient estimate can be expressed as,

$$\nabla_\theta \mathcal{O}_{\mathrm{ER}}(\theta) \;\approx\; \frac{1}{K} \sum_{k=1}^{K} \nabla \log \pi_\theta(\mathbf{a}^{(k)}) \,[R(\mathbf{a}^{(k)}) - b] \;, \tag{4.5}$$

where a baseline b is subtracted from the returns to reduce the variance of gradient estimates. A close-to-optimal form of a baseline is the on-policy average of the returns $\mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a})} R(\mathbf{a})$, often approximated empirically as $b = \sum_k R(\mathbf{a}^{(k)})/K$. This formulation enables direct optimization of $\mathcal{O}_{\mathrm{ER}}$ via MC sampling from an unknown search space, which also serves the purpose of exploration. To improve such exploration behavior, one often includes the entropy of the policy as an additional term inside the objective to prevent early convergence. However, the key limitation of the formulation stems from the difficulty of estimating the gradients accurately only using a few *fresh* samples.

There has been various attempts to incorporate off-policy samples within the policy gradient framework to improve the sample efficiency of the REINFORCE and actor-critic algorithms (*e.g.*, [**28, 141, 122, 33**]). Most of these approaches sample from an old policy and resort to truncated importance correction to obtain a low variance, but *biased* estimate of

the gradients. Previous work has aimed to incorporate a replay buffer into policy gradient in the general RL setting of stochastic dynamics and possibly continous actions. By contrast, we focus on deterministic environments with discrete actions and develop a formulation resulting in an *unbiased* policy gradient estimate.

4.3.4. MAPO: Memory Augmented Policy Optimization

We consider a RL environment with a finite number of discrete actions, deterministic dynamics, and deterministic terminal returns. In other words, the set of all possible action trajectories $\mathcal{A}$ is countable, even though possibly infinite, and re-evaluating the return of a trajectory $R(\mathbf{a})$ twice results in the same value. These assumptions characterize the problem of program synthesis, but also apply to many combinatorial optimization domains (*e.g.*, [**10**]).

Our goal is to optimize the expected return objective (4.4) via gradient ascent. We assume the access to a memory buffer of trajectories and their corresponding returns denoted $\mathcal{B} \equiv \{(\mathbf{a}^{(i)}, r^{(i)})\}_{i=1}^{M}$, where $r^{(i)} = R(\mathbf{a}^{(i)})$. Let $\mathcal{B}_a$ and $\mathcal{B}_r$ denote the action trajectories and the rewards stored in the memory independently, *i.e.* $\mathcal{B}_a \equiv \{(\mathbf{a}^{(i)})\}_{i=1}^{M}$ and $\mathcal{B}_r \equiv \{(r^{(i)})\}_{i=1}^{M}$. Our key observation is that one can re-express the expected return objective in terms of a sum of two terms: an enumeration over the memory buffer elements, and a separate enumeration over the unknown trajectories,

$$\mathcal{O}_{\mathrm{ER}}(\theta) \;=\; \sum_{(\mathbf{a},r)\in\mathcal{B}} \pi_\theta(\mathbf{a})\, r \;+\; \sum_{\mathbf{a}\in(\mathcal{A}-\mathcal{B}_a)} \pi_\theta(\mathbf{a})\, R(\mathbf{a}), \tag{4.6}$$

where $\mathcal{A} - \mathcal{B}_a$ denotes the set of action trajectories not included in the memory. The significance of this decomposition hinges on the fact that one can enumerate the trajectories in the buffer and compute their exact expected reward and its gradient with no variance, and use the budget of MC samples to compute the expectation only in the unexplored part of the space.

Let $\pi_{\mathcal{B}} = \sum_{\mathbf{a}\in\mathcal{B}_a} \pi_\theta(\mathbf{a})$ denote the total probability of the trajectories in the buffer, and let $\pi_\theta^+(\mathbf{a})$ and $\pi_\theta^-(\mathbf{a})$ denote a normalized probability distribution inside and outside of the

buffer,

$$\pi_\theta^+(\mathbf{a}) = \begin{cases} \pi_\theta(\mathbf{a})/\pi_\mathcal{B} & \text{if } \mathbf{a} \in \mathcal{B}_a \\ 0 & \text{if } \mathbf{a} \notin \mathcal{B}_a \end{cases}, \; \pi_\theta^-(\mathbf{a}) = \begin{cases} 0 & \text{if } \mathbf{a} \in \mathcal{B}_a \\ \pi_\theta(\mathbf{a})/(1-\pi_\mathcal{B}) & \text{if } \mathbf{a} \notin \mathcal{B}_a \end{cases} . \tag{4.7}$$

Then, one can re-express the expected return objective as the linear combination of two expectations,

$$\mathcal{O}_{\text{ER}}(\theta) \;=\; \pi_\mathcal{B} \underbrace{\mathbb{E}_{\mathbf{a}\sim\pi_\theta^+(\mathbf{a})} R(\mathbf{a})}_{\text{enumeration inside } \mathcal{B}} + (1-\pi_\mathcal{B}) \underbrace{\mathbb{E}_{\mathbf{a}\sim\pi_\theta^-(\mathbf{a})} R(\mathbf{a})}_{\text{sampling outside } \mathcal{B}} . \tag{4.8}$$

The key intuition of the MAPO algorithm is to use enumeration to evaluate the former expectation and $\pi_\mathcal{B}$, and MC sampling to compute the latter expectation. To sample from $\pi_\theta^-(\mathbf{a})$, one can resort to rejection sampling by sampling from $\pi_\theta(\mathbf{a})$ and rejecting the sample if $\mathbf{a} \in \mathcal{B}_a$. We get an exact estimate of the first expectation while sampling from a smaller stochastic space of size $(1-\pi_\mathcal{B})$ to get an estimate of the latter expectation.

Based on (4.8), using K trajectories $\{\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(K)}\}$ sampled *i.i.d.* from current π_θ^-, *i.e.* the unexplored region of the space, we formulate an unbiased estimator of the policy gradients as,

$$\nabla_\theta \mathcal{O}_{\text{ER}}(\theta) \;\approx\; \sum_{(\mathbf{a},r)\in\mathcal{B}} \nabla\pi_\theta(\mathbf{a})\,[r-b] \;+\; \frac{1-\pi_\mathcal{B}}{K} \sum_{k=1}^{K} \nabla \log \pi_\theta(\mathbf{a}^{(k)})\,[R(\mathbf{a}^{(k)}) - b] \; . \tag{4.9}$$

Assuming that $\pi_\mathcal{B} > 0$, using a full enumeration of the buffer and a budget of K MC samples, the variance of the estimator in (4.9) is lower than the estimator in (4.5) because it has less stochasticity. For practical applications in which sample evaluation is expensive, we expect the MAPO estimator in (4.9) to significantly outperform policy gradients. Note that even if we use sampling to approximate the first term, this can be viewed as a stratified sampling estimator. The variance reduction still holds as long as the trajectories inside and outside $\mathcal{B}$ have different distributions, which is usually true.

In the following we present 3 techniques to create an efficient algorithm. An overview of the MAPO training algorithm is also shown in Figure 4.3.

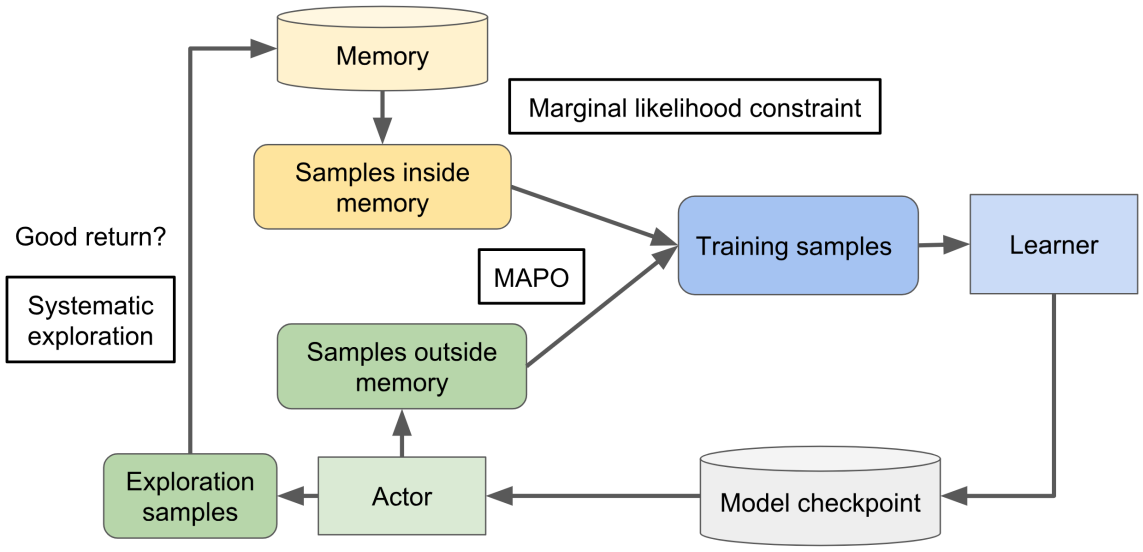

Figure 4.3. Overview of the MAPO algorithm with systematic exploration and marginal likelihood constraint.

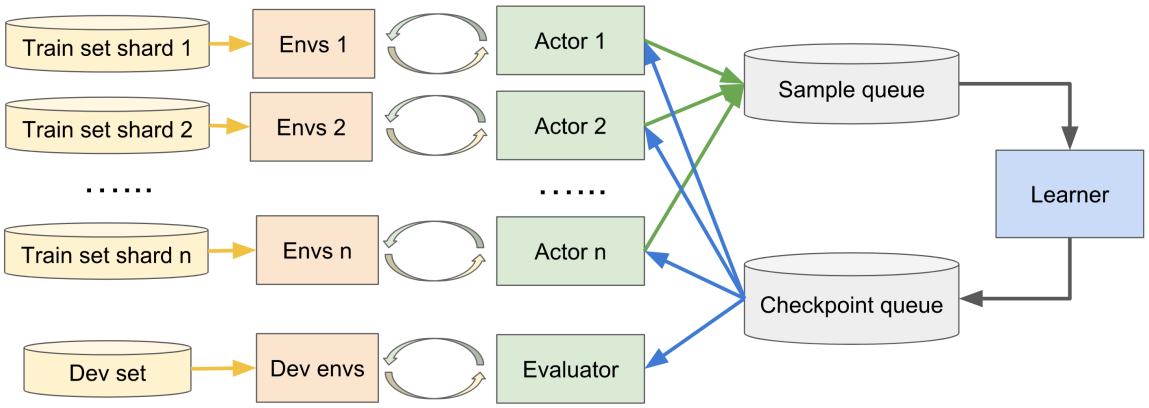

Figure 4.4. Distributed actor-learner architecture.

4.3.4.1. Distributed Sampling. An exact computation of the first term on the RHS of (4.9) involves an enumeration over $\mathcal{B}$, which is efficient only when $|\mathcal{B}|$ is small. If $|\mathcal{B}|$ is large, enumeration can become prohibitive even though it involves no fresh samples. One solution is to simply truncate the memory buffer to keep only a few trajectories with the highest probabilities. However, this simple approach may lead to suboptimal result, because one often benefits from keeping all of the promising programs in the buffer to let model select the ones that generalize well. For example, in program synthesis, there can sometimes be many

programs that compute the correct answer for a certain question, but only a handful will generalize and others are spurious (getting the correct answer because of luck, e.g., using $2 + 2$ to answer the question "what is two times two"). The generalizable programs could be discarded during the truncation of the memory buffer, making it hard for the model to overcome the spurious programs.

Alternatively, we can approximate the expectations over the trajectories both inside and outside $\mathcal{B}$ using sampling. As mentioned above, this can be viewed as *stratified sampling* from inside and outside the memory buffer and variance reduction still holds. A key insight is that the cost of sampling can be distributed through an actor-learner architecture depicted in Figure 4.4 inspired by [**33**]. The actors uses a stale model checkpoint to sample trajectories from inside the memory buffer through renormalization, and uses rejection sampling to pick trajectories from outside the memory. It also computes the weights for these trajectories using the stale model. These trajectories and their weights are then pushed to a queue of samples. A learner fetches samples from the queue and use them to compute gradient estimates to update the parameters.

Algorithm 4 Systematic Exploration

Input: context $\mathbf{x}$, policy π, exploration buffer of fully explored sub-sequences $\mathcal{B}^e$, memory buffer of high-reward sequences $\mathcal{B}$

Initialize: empty sequence $a_{0:0}$

while true **do**

  $V = \{a \mid a_{0:t-1} \| a \notin B^e\}$

  if $V == \emptyset$ **then**

    $\mathcal{B}^e \leftarrow \mathcal{B}^e \cup a_{0:t-1}$

    break

  sample $a_t \sim \pi^V(a|a_{0:t-1})$

  $a_{0:t} \leftarrow a_{0:t-1} \| a_t$

  if $a_t ==$ EOS **then**

    if $R(a_{0:t}) > 0$ **then**

      $\mathcal{B} \leftarrow \mathcal{B} \cup a_{0:t}$

    $\mathcal{B}^e \leftarrow \mathcal{B}^e \cup a_{0:t}$

    break

4.3.4.2. Marginal Likelihood Constraint. Policy gradient methods usually suffer from a cold start problem if the model is randomly initialized. When supervised data in the form of demonstrations $\mathcal{D}$ is available, a reasonable approach (*e.g.*, [**146**]) incorporates such data by combining the expected return objective with a conditional log likelihood objective via,

$$\mathcal{O}_{\text{AUG}}(\theta) = \lambda \mathcal{O}_{\text{ER}}(\theta) + (1 - \lambda) \sum_{\mathbf{a} \in \mathcal{D}} \log \pi_\theta(\mathbf{a}) \tag{4.10}$$

In our setup, we do not have access to demonstrations, but through exploration we accumulate the memory buffer with various high-reward trajectories. Instead of optimizing $\mathcal{O}_{\text{AUG}}(\theta)$, which introduces some bias, we adopt a clipping mechanism that ensures that the buffer probability is greater or equal to α, *i.e.* $\pi_{\mathcal{B}} \geq \alpha$, otherwise clips it to α. Then, the policy gradient estimates are forced to pay a certain amount of attention to the high-reward

trajectories in the buffer effectively optimizing

$$\mathcal{O}_{\text{MML}}(\theta) = \frac{1}{N} \sum_i \log \sum_{\mathbf{a} \in \mathcal{B}_i} \pi_\theta(\mathbf{a}) = \frac{1}{N} \log \prod_i \sum_{\mathbf{a} \in \mathcal{B}_i} \pi_\theta(\mathbf{a}). \tag{4.11}$$

At the beginning of training, the clipping will be active and it will introduce a bias, but will facilitate much faster training. Once the policy is off the ground, the buffer probabilities are almost never clipped given that they are naturally larger than α and the gradients are not biased any more.

4.3.4.3. Systematic Exploration. To avoid repeated samples during exploration, we propose to use systematic exploration. More specifically we keep a set $\mathcal{B}^e$ of fully explored partial sequences, which can be efficiently implemented using a bloom filter. Then, we use it to enforce a policy to only take actions that lead to unexplored sequences. Using a bloom filter we can store billions of sequences in $\mathcal{B}^e$ with only several gigabytes of memory. The pseudo code for this approach is shown in Algorithm 4. We warm start the memory buffer using systematic exploration from random policy as it can be trivially parallelized. In parallel to training, we continue the systematic exploration with the current policy to discover new high reward trajectories.

Algorithm 5 MAPO

Input: data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, memory and exploration buffer $\{(\mathcal{B}_i, \mathcal{B}_i^e)\}_{i=1}^N$, constants α, ϵ, M

repeat ▷ **for all actors**

Initialize training batch $D \leftarrow \emptyset$

Get a batch of inputs C

for $(\mathbf{x}_i, \mathbf{y}_i, \mathcal{B}_i^e, \mathcal{B}_i) \in C$ **do**

SystematicExploration$(\mathbf{x}_i, \pi_\theta^{old}, \mathcal{B}_i^e, \mathcal{B}_i)$

Sample $\mathbf{a}_i^+ \sim \pi_\theta^{old}$ over $\mathcal{B}_i$

$w_i^+ \leftarrow \max(\pi_\theta^{old}(\mathcal{B}_i), \alpha)$

$D \leftarrow D \cup (\mathbf{a}_i^+, R(\mathbf{a}_i^+), w_i^+)$

Sample $\mathbf{a}_i \sim \pi_\theta^{old}$

if $\mathbf{a}_i \notin \mathcal{B}_i$ **then**

$w_i \leftarrow (1 - w_i^+)$

$D \leftarrow D \cup (\mathbf{a}_i, R(\mathbf{a}_i), w_i)$

Push D to training queue

until converge

repeat ▷ **for the learner**

Get a batch D from training queue

for $(\mathbf{a}_i, R(\mathbf{a}_i), w_i) \in D$ **do**

$\mathrm{d}\theta \leftarrow \mathrm{d}\theta + w_i \, R(\mathbf{a}_i) \, \nabla \log \pi_\theta(\mathbf{a}_i)$

update θ using $\mathrm{d}\theta$

$\pi_\theta^{old} \leftarrow \pi_\theta$ ▷ once every M batches

until converge or a certain number of steps is reached

Output: final parameters θ

4.3.4.4. Final Algorithm. Putting things together, the final training procedure is summarized in Algorithm 5. As mentioned above, we adopt the actor-learner architecture for distributed training. It uses multiple actors to collect training samples asynchronously and

one learner for updating the parameters based on the training samples. Each actor interacts with a set of environments to generate new trajectories. For efficiency, an actor uses a stale policy (π_θ^{old}), which is often a few steps behind the policy of the learner and will be synchronized periodically. To apply MAPO, each actor also maintains a memory buffer $\mathcal{B}_i$ to save the high-reward trajectories. To prepare training samples for the learner, the actor picks n_b samples from inside $\mathcal{B}_i$ and also performs rejection sampling with n_o on-policy samples, both according to the actor's policy π_θ^{old}. We then use the actor policy to compute a weight $max(\pi_\theta(\mathcal{B}), \alpha)$ for the samples in the memory buffer, and use $1 - max(\pi_\theta(\mathcal{B}), \alpha)$ for samples outside of the buffer. These samples are pushed to a queue and the learner reads data from the queue to compute gradients and update the parameters.

4.3.5. Experiments

We evaluate MAPO on two program synthesis from natural language (also known as *semantic parsing*) benchmarks, WikiTableQuestions and WikiSQL, which requires generating programs to query and process data from tables to answer natural language questions. We first compare MAPO to four common baselines to show the advantage of combining samples from inside and outside the memory buffer, and ablate systematic exploration and marginal likelihood constraint to show their utility. Then we compare MAPO to the state-of-the-art on these two benchmarks. On WikiTableQuestions, MAPO is the first RL-based approach that significantly outperforms the previous state-of-the-art. On WikiSQL, MAPO trained with weak supervision (question-answer pairs) outperforms several strong models trained with full supervision (question-program pairs).

4.3.5.1. Experimental setup. In this section, we discuss the details of the experiments, including the datasets, the model architecture and the training details.

Datasets. WikiTableQuestions [**100**] contains tables extracted from Wikipedia and question-answer pairs about the tables. See Table 4.7 as an example. There are 2,108 tables and 18,496 question-answer pairs. We follow the construction in [**100**] for converting a table into a directed graph that can be queried, where rows and cells are converted to graph nodes

while column names become labeled directed edges. For the questions, we use string match to identify phrases that appear in the table. We also identify numbers and dates using the CoreNLP annotation released with the dataset. The task is challenging in several aspects. First, the tables are taken from Wikipedia and cover a wide range of topics. Second, at test time, new tables that contain unseen column names appear. Third, the table contents are not normalized as in knowledge-bases like Freebase, so there are noises and ambiguities in the table annotation. Last, the semantics are more complex comparing to previous datasets like WebQuestionsSP [**149**]. It requires multiple-step reasoning using a large set of functions, including comparisons, superlatives, aggregations, and arithmetic operations [**100**].

WikiSQL [**161**] is a recently introduced large scale dataset on learning natural language interfaces for databases. The dataset also uses tables extracted from Wikipedia, but is much larger and is annotated with programs (SQL). There are 24,241 tables and 80,654 question-program pairs in the dataset. Comparing to WikiTableQuestions, the semantics are simpler because the SQLs use fewer operators (column selection, aggregation, and conditions). We perform similar preprocessing as for WikiTableQuestions. Most of the state-of-the-art models relies on question-program pairs for supervised training, while we only use the question-answer pairs for weakly supervised training.

Model architecture. We adopt the Neural Symbolic Machines framework[**71**], which combines (1) a neural "programmer", which is a seq2seq model augmented by a key-variable memory that can translate a natural language utterance to a program as a sequence of tokens, and (2) a symbolic "computer", which is an Lisp interpreter that implements a domain specific language with built-in functions and provides code assistance by eliminating syntactically or semantically invalid choices.

For the Lisp interpreter, we adopt a domain specific language with certain built-in functions. A program C is a list of expressions $(c_1...c_N)$, where each expression is either a special token "*EOS*" indicating the end of the program, or a list of tokens enclosed by parentheses "$(FA_1...A_K)$". F is a function, which takes as input K arguments of specific types. Table 4.8 defines the arguments, return value and semantics of each function. In the

table domain, there are rows and columns. The value of the table cells can be number, date time or string, so we also categorize the columns into number columns, date time columns and string columns depending on the type of the cell values in the column.

Function	Arguments	Returns	Description
(**hop** v p)	**v**: a list of rows. **p**: a column.	a list of cells.	Select the given column of the given rows.
(**argmax** v p) (**argmin** v p)	**v**: a list of rows. **p**: a number or date column.	a list of rows.	From the given rows, select the ones with the largest / smallest value in the given column.
(**filter**$_{>}$ v q p) (**filter**$_{\geq}$ v q p) (**filter**$_{<}$ v q p) (**filter**$_{\leq}$ v q p) (**filter**$_{=}$ v q p) (**filter**$_{\neq}$ v q p)	**v**: a list of rows. **q**: a number or date. **p**: a number or date column.	a list of rows.	From the given rows, select the ones whose given column has certain order relation with the given value.
(**filter**$_{in}$ v q p) (**filter**$_{!in}$ v q p)	**v**: a list of rows. **q**: a string. **p**: a string column.	a list of rows.	From the given rows, select the ones whose given column contain / do not contain the given string.
(**first** v) (**last** v)	**v**: a list of rows.	a row.	From the given rows, select the one with the smallest / largest index.
(**previous** v) (**next** v)	**v**: a row.	a row.	Select the row that is above / below the given row.
(**count** v)	**v**: a list of rows.	a number.	Count the number of given rows.
(**max** v p) (**min** v p) (**average** v p) (**sum** v p)	**v**: a list of rows. **p**: a number column.	a number.	Compute the maximum / minimum / average / sum of the given column in the given rows.
(**mode** v p)	**v**: a list of rows. **p**: a column.	a cell.	Get the most common value of the given column in the given rows.
(**same_as** v p)	**v**: a row. **p**: a column.	a list of rows.	Get the rows whose given column is the same as the given row.
(**diff** v0 v1 p)	**v0**: a row. **v1**: a row. **p**: a number column.	a number.	Compute the difference in the given column of the given two rows.

Table 4.8. Functions used in the experiments.

In the WikiTableQuestions experiments, we used all the functions in the table. In the WikiSQL experiments, because the semantics of the questions are simpler, we used a subset of the functions (hop, filter$_=$, filter$_{in}$, filter$_>$, filter$_<$, count, maximum, minimum, average and sum). We created the functions according to [**160, 90**].[3]

We implemented the seq2seq model augmented with key-variable memory from [**71**] in TensorFlow. (1) we used a bi-directional LSTM for the encoder; (2) we used two-layer LSTM with skip-connections in both the encoder and decoder. GloVe [**103**] embeddings are used for the embedding layer in the encoder and also to create embeddings for column names by averaging the embeddings of the words in a name. Following [**90, 65**], we also add a binary feature in each step of the encoder, indicating whether this word is found in the table, and an integer feature for a column name counting how many of the words in the column name appear in the question. For the WikiTableQuestions dataset, we use the CoreNLP annotation of numbers and dates released with the dataset. For the WikiSQL dataset, only numbers are used, so we use a simple parser to identify and parse the numbers in the questions, and the tables are already preprocessed. The tokens of the numbers and dates are anonymized as two special tokens <NUM> and <DATE>. The hidden size of the encoder and decoder LSTM is 200. We keep the GloVe embeddings fixed during training, but project it to 200 dimensions using a trainable linear transformation. The same architecture is used for both WikiTableQuestions and WikiSQL.

Training Details. We first apply systematic exploration using a random policy to discover high-reward programs to warm start the memory buffer of each example. For WikiTableQuestions, we generated 10k programs per example using systematic exploration with pruning rules inspired by the grammars from [**160**]. We apply 0.2 dropout on both encoder and decoder, and 0.01 entropy regularization [**82**]. Each batch includes samples from 25 examples. For experiments on WikiSQL, we generated 1k programs per example due to computational constraint. Because the dataset is much larger, we don't use any

[3]The only function we have added to capture some complex semantics is the same_as function, but it only appears in 1.2% of the generated programs (among which 0.6% are correct and the other 0.6% are incorrect), so even if we remove it, the significance of the difference in Table 4.10 will not change.

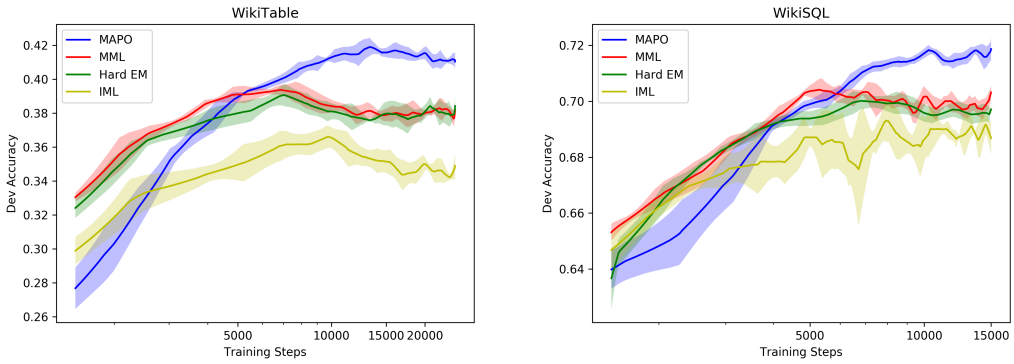

Figure 4.5. Comparison of MAPO and 3 baselines' dev set accuracy curves. Results on WikiTableQuestions is on the left and results on WikiSQL is on the right. The plot is average of 5 runs with a bar of one standard deviation. The horizontal coordinate (training steps) is in log scale.

regularization. Each batch includes samples from 125 questions. We use distributed sampling for WikiTableQuestions. For WikiSQL, due to computational constraints, we use the simple approach to truncate each memory buffer to top 5 and then enumerate all 5 programs for training. For both experiments, the samples outside memory buffer are drawn using rejection sampling from 1 on policy sample per example. At inference time, we apply beam search of size 5. We evaluate the model periodically on the dev set to select the best model.

We apply a distributed actor-learner architecture for training. The actors use CPUs to generate new trajectories and push the samples into a queue for training. The learner reads batches of data from the queue and uses GPU to accelerate training. We use Adam optimizer for training and the learning rate is 10^{-3}. All the hyperparameters are tuned on the dev set.

4.3.5.2. Comparison to baselines implemented within our framework. To assess the effectiveness of MAPO, we compare against the following baselines in a controlled manner by using the same neural architecture and experimental framework.

► **REINFORCE:** We use on-policy samples to estimate the gradient of expected return as in (4.5), not utilizing any form of memory buffer.

► **MML:** Maximum Marginal Likelihood maximizes the marginal probability of the memory buffer as in (4.11). Assuming binary rewards and assuming that the memory buffer contains almost all of the trajectories with a reward of 1, MML optimizes the marginal probability of generating a rewarding program. Note that under these assumptions, expected return can be

Method	WikiTable	WikiSQL
REINFORCE	< 10	< 10
MML (Soft EM)	39.7 ± 0.3	70.7 ± 0.1
Hard EM	39.3 ± 0.6	70.2 ± 0.3
IML	36.8 ± 0.5	70.1 ± 0.2
MAPO	$\mathbf{42.3} \pm 0.3$	$\mathbf{72.2} \pm 0.2$
No Systematic Exploration	< 10	< 10
No Marginal Likelihood	< 10	< 10

Table 4.9. Ablation study on both datasets. We report mean accuracy % ± standard deviation on dev sets based on 5 runs.

Method	E.S.	Dev.	Test
Pasupat & Liang (2015) [**100**]	-	37.0	37.1
Neelakantan *et al.* (2017) [**90**]	1	34.1	34.2
Neelakantan *et al.* (2017) [**90**]	15	37.5	37.7
Haug *et al.* (2017) [**49**]	1	-	34.8
Haug *et al.* (2017) [**49**]	15	-	38.7
Zhang *et al.* (2017) [**160**]	-	40.4	43.7
MAPO	1	**42.3**	43.8
MAPO (ensembled)	5	-	**46.2**

Table 4.10. Results on WikiTableQuestions. E.S. is the ensemble size, when applicable.

expressed as $\mathcal{O}_{\text{ER}}(\theta) \approx \frac{1}{N} \sum_i \sum_{\mathbf{a} \in \mathcal{B}_i} \pi_\theta(\mathbf{a})$. Comparing the two objectives, we can see that MML maximizes the product of marginal probabilities, whereas expected return maximizes the sum. More discussion of these two types of objectives can be found in [**48, 95, 113**].

► **Hard EM:** Expectation-Maximization algorithm is commonly used to optimize the marginal likelihood in the presence of latent variables. Hard EM uses the samples with the highest probability to approximate the gradient to $\mathcal{O}_{\text{MML}}$.

► **IML:** Iterative Maximum Likelihood training [**71**] uniformly maximizes the likelihood of all the trajectories with the highest rewards $\mathcal{O}_{\text{CLL}}(\theta) = \sum_{\mathbf{a} \in \mathcal{B}} \log \pi_\theta(\mathbf{a})$. Because the memory buffer is too large to enumerate, we use samples from the buffer to approximate the gradient for MML and IML, and uses samples with highest $\pi_\theta(\mathbf{a})$ for Hard EM.

Method	Dev.	Test
Fully supervised		
Zhong *et al.* (2017) [**161**]	60.8	59.4
Wang *et al.* (2017) [**138**]	67.1	66.8
Xu *et al.* (2017) [**147**]	69.8	68.0
Huang *et al.* (2018) [**55**]	68.3	68.0
Yu *et al.* (2018) [**153**]	74.5	73.5
Sun *et al.* (2018) [**131**]	75.1	74.6
Dong & Lapata (2018) [**32**]	**79.0**	**78.5**
Weakly supervised		
MAPO	**72.2**	72.6
MAPO (ensemble of 5)	-	**74.9**

Table 4.11. Results on WikiSQL. Unlike other methods, MAPO only uses weak supervision.

The results are summarized in Table 4.9. We show the accuracy curves on the dev set in Figure 4.5. Removing systematic exploration or the marginal likelihood constraint significantly weaken MAPO possibly because high-reward trajectories are not found or easily forgotton. REINFORCE barely learns anything useful because starting from a random policy, most samples result in a reward of zero. MML and Hard EM converge faster, but the learned models underperform MAPO, which suggests that the expected return is a better objective. IML runs faster than MML and MAPO because it randomly samples from the buffer, but the objective is prone to spurious programs.

4.3.5.3. Comparison to state-of-the-art. On WikiTableQuestions (Table 4.10), MAPO is the first RL-based approach that significantly outperforms the previous state-of-the-art by 2.5%. Unlike previous work, MAPO does not require manual feature engineering or additional human annotation[4]. On WikiSQL (Table 4.11), even though MAPO does not exploit ground truth programs and only uses weak supervision, it is able to outperform many strong baselines trained using programs as full supervision. The techniques introduced in other models can be incorporated to further improve the result of MAPO, but we leave that

[4]Krishnamurthy *et al.* [**65**] achieved 45.9 accuracy when trained on the data collected with dynamic programming and pruned with more human annotations [**101, 87**].

as future work. We also qualitatively analyzed a trained model and see that it can generate fairly complex programs. Table 4.12 shows examples of several types of programs generated by a trained model.

Statement	Comment
Superlative	
nt-13901: the most points were scored by which player?	
(argmax all_rows r.points-num)	Sort all rows by column 'points' and take the first row.
(hop v0 r.player-str)	Output the value of column 'player' for the rows in v0.
Difference	
nt-457: how many more passengers flew to los angeles than to saskatoon?	
(filter$_{in}$ all_rows ['saskatoon'] r.city-str)	Find the row with 'saskatoon' matched in column 'city'.
(filter$_{in}$ all_rows ['los angeles'] r.city-str)	Find the row with 'los angeles' matched in column 'city'.
(diff v1 v0 r.passengers-num)	Calculate the difference of the values in the column 'passenger' of v0 and v1.
Before / After	
nt-10832: which nation is before peru?	
(filter$_{in}$ all_rows ['peru'] r.nation-str)	Find the row with 'peru' matched in 'nation' column.
(previous v0)	Find the row before v0.
(hop v1 r.nation-str)	Output the value of column 'nation' of v1.
Compare & Count	
nt-647: in how many games did sri lanka score at least 2 goals?	
(filter$_{\geq}$ all_rows [2] r.score-num)	Select the rows whose value in the 'score' column ¿= 2.
(count v0)	Count the number of rows in v0.
Exclusion	
nt-1133: other than william stuart price, which other businessman was born in tulsa?	
(filter$_{in}$ all_rows ['tulsa'] r.hometown-str)	Find rows with 'tulsa' matched in column 'hometown'.
(filter$_{!in}$ v0 ['william stuart price'] r.name-str)	Drop rows with 'william stuart price' matched in the value of column 'name'.
(hop v1 r.name-str)	Output the value of column 'name' of v1.

Table 4.12. Example programs generated by a trained model.

4.3.6. Related work

Program synthesis & semantic parsing. There has been a surge of recent interest in applying reinforcement learning to program synthesis [**18, 2, 154, 89**] and combinatorial optimization [**163, 10**]. Different from these efforts, we focus on the contextualized program synthesis where generalization to new contexts is important. Semantic parsing [**155, 156, 73**]

maps natural language to executable symbolic representations. Training semantic parsers through weak supervision is challenging because the model must interact with a symbolic interpreter through non-differentiable operations to search over a large space of programs [**12, 71**]. Previous work [**48, 90**] reports negative results when applying simple policy gradient methods like REINFORCE [**143**], which highlights the difficulty of exploration and optimization when applying RL techniques to program synthesis or semantic parsing. MAPO takes advantage of discrete and deterministic nature of program synthesis and significantly improves upon REINFORCE.

Experience replay. An experience replay buffer [**76**] enables storage and usage of past experiences to improve the sample efficiency of RL algorithms. Prioritized experience replay [**117**] prioritizes replays based on temporal-difference error for more efficient optimization. Hindsight experience replay [**5**] incorporates goals into replays to deal with sparse rewards. MAPO also uses past experiences to tackle sparse reward problems, but by storing and reusing high-reward trajectories, similar to [**71, 97**]. Previous work[**71**] assigns a fixed weight to the trajectories, which introduces bias into the policy gradient estimates. More importantly, the policy is often trained equally on the trajectories that have the same reward, which is prone to reinforcing spurious programs. By contrast, MAPO uses the trajectories in a principled way to obtain an unbiased low variance gradient estimate.

Variance reduction. Policy optimization via gradient descent is challenging because of: (1) large *variance* in gradient estimates; (2) small gradients in the initial phase of training. Prior variance reduction approaches [**145, 143**] mainly relied on control variate techniques by introducing a critic model [**64, 82, 122**]. MAPO takes a different approach to reformulate the gradient as a combination of expectations inside and outside a memory buffer. Standard solutions to the small gradient problem involves supervised pretraining [**124, 51, 108**] or using supervised data to generate rewarding samples [**95, 29**], which cannot be applied when supervised data are not available. MAPO solves this problem by a soft constraint on the marginal likelihood of the memory buffer, which accelerates training at the beginning and becomes unbiased once the constraint is satisfied.

Exploration. Recently there has been a lot of work on improving exploration [**102, 135, 54**] by introducing additional reward based on information gain or pseudo count. For program synthesis [**7, 90, 18**], the search spaces are enumerable and deterministic. Therefore, we propose to conduct systematic exploration, which ensures that only novel trajectories are generated.

4.3.6.1. Conclusion. We present memory augmented policy optimization (MAPO) that incorporates a memory buffer of promising action trajectories to reduce the variance of policy gradients. We propose 3 techniques to enable an efficient training algorithm for MAPO. (1) distributed sampling from inside and outside memory buffer in an actor-learner architecture; (2) a constraint over the marginal likelihood of the trajectories in the memory buffer to accelerate training; (3) systematic exploration to efficiently discover high-reward trajectories. MAPO is evaluated on real world program synthesis from natural language tasks. On WikiTableQuestions, MAPO is the first RL approach that significantly outperforms previous state-of-the-art; on WikiSQL, MAPO trained with only weak supervision outperforms several strong baselines trained with full supervision.

CHAPTER 5

Conclusions

This thesis shows how to integrate machine learning with symbolic reasoning by learning to generate symbolic representations from weak supervision. Given the problem of learning a mapping from input x to output y, instead of directly learning the function $y = f(x)$, the proposed approach decomposes f into two functions: the first one generates a latent symbolic representation z, which is usually a structured object defined by a grammar G, as $z = h(x)$; the second one uses the input and the symbolic representation together to predict the final output as $y = g(z, x) = g(h(x), x)$. Both h and g are learned in an end-to-end manner from weak supervision, i.e., input-output pairs $\{(x_i, y_i)\}$. The approach has three main advantages:

(1) **Efficiency** By selecting the right type of symbolic representations for the task, for example, using structural alignment for similarity estimation, we can introduce an effective inductive bias to the model, which will make the model learn faster and require less data to train.

(2) **Expressiveness** The expressive symbolic representations can be used to represent complex reasoning, for example, multi-step reasoning using different operations (i.e., arithmetics, sorting or querying external knowledge base) can be represented efficiently as compositional programs, which is usually hard or inefficient to represent in a purely statistical model.

(3) **Intepretability** The generated symbolic representations, for example, alignment or programs, can be inspected and verified by the users, which makes the model more interpretable and easier to debug.

We then evaluated this approach in two settings:

(1) **Learning Similarity Estimation with Structural Alignment** The goal is to learn a model that, given two structured or relational representations as the input x, can

estimate their similarity score as the output y. By using the structural alignment as the latent symbolic representation z, we can introduce an inductive bias into the model to let it focus on how one structure aligns with the other structure to consider both structural and local similarity, which makes learning much more sample-efficient. When applied to knowledge base completion / link prediction task, this enables a model to achieve state-of-the-art accuracy using orders of magnitude less data. On the paraphrase identification task, a simple model using structural alignment achieved competitive results to other state-of-the-art models that used more data or more sophisticated learning methods. Besides, the generated alignment can be used to explain the similarity estimation by showing the contribution of each match or mismatch.

(2) **Learning to Generate Programs from Natural Language** The goal is to learn a model that, given a natural language utterance as input x, can produce the answer as output y. By using compositional programs as the latent symbolic representations z, the model can represent complex multi-step reasoning using different operations like arithmetics, sorting and querying external knowledge base. It is the first end-to-end model that significantly outperforms the previous state-of-the-art without feature engineering on two challenging tasks: answering open-domain question using Freebase and answering compositional questions about tables. The generated programs can also be used to verify or debug the reasoning process of the model.

To successfully apply the proposed approach, the weakly supervised structured prediction problem needs to be solved. In the first setting, we model the symbolic representation as a latent variable and use a EM-like process to jointly train structural alignment using structured perceptron and similarity estimation using SVM. In the second setting, we formalize the problem of generating programs as reinforcement learning. To apply policy gradient methods to this large search space with sparse rewards, we proposed novel techniques to improve its the sample efficiency and robustness by using a replay buffer of high-reward trajectories and systematic exploration strategy.

The proposed approach is most applicable in high-level tasks that deals with noisy input, i.e., natural language utterances or images, and requires explicit reasoning. For low-level tasks in perception and motor control, for example, image segmentation or controlling a robot arm, it might be better to apply purely statistical learning methods. And for reasoning tasks with perfect input like solving logic puzzles or theorem proving, it is better to apply purely symbolic reasoning methods.

Although we focus on applications in natural language understanding in this thesis, the proposed approach is generally applicable to many tasks that requires high-level reasoning. Some high-level vision tasks [**59**] also requires multi-step reasoning, for example, given a image, to answer question like "How many cyan things are right of the gray cube or left of the small cube". The designing process of a neural network model can also be viewed as generating a symbolic representation that defines the model architecture [**163**], and the training signal is the performance of the generated model.

To conclude, the proposed approach presents a new way of incorporating symbolic representations into machine learning, which fits into a long line of research that attempts to integrate the two branches in AI. It is generally applicable to many high-level tasks where both learning and reasoning are required, and achieved promising results in several real world applications.

References

[1] *Practical structured learning techniques for natural language processing*. University of Southern California, 2006.

[2] Daniel A Abolafia, Mohammad Norouzi, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

[3] Daniel A. Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V. Le. Neural program synthesis with priority queue training. *arXiv:1801.03526*, 2018.

[4] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *arXiv:1601.01705*, 2016.

[5] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *NIPS*, 2017.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[7] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2017.

[8] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *ACL*, 2014.

[9] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *JMLR*, 2013.

[10] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv:1611.09940*, 2016.

[11] Islam Beltagy, Katrin Erk, and Raymond J. Mooney. Probabilistic soft logic for semantic textual similarity. In *ACL*, 2014.

[12] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. *EMNLP*, 2(5):6, 2013.

[13] Jonathan Berant and Percy Liang. Imitation learning of agenda-based semantic parsers. *TACL*, 2015.

[14] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. *International Conference on Management of Data (SIGMOD)*, pages 1247–1250, 2008.

[15] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *EMNLP*, 2015.

[16] Ronald J Brachman, Hector J Levesque, and Raymond Reiter. *Knowledge representation*. MIT press, 1992.

[17] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv:1606.01540*, 2016.

[18] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

[19] Jianpeng Cheng and Dimitri Kartsaklis. Syntax-aware multi-sense word embeddings for deep compositional models of meaning. In *EMNLP*, 2015.

[20] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *EMNLP*, 2014.

[21] Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 1–8. Association for Computational Linguistics, 2002.

[22] Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 111. Association for Computational Linguistics, 2004.

[23] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[24] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *MLCW*, 2005.

[25] Abhishek Das, Satwik Kottur, José MF Moura, Stefan Lee, and Dhruv Batra. Learning cooperative visual dialog agents with deep reinforcement learning. *arXiv:1703.06585*, 2017.

[26] Dipanjan Das and Noah A. Smith. Paraphrase identification as probabilistic quasi-synchronous recognition. In *ACL*, 2009.

[27] H. Daume, J. Langford, and D. Marcu. Search-based structured prediction. *Machine Learning*, 75:297–325, 2009.

[28] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *ICML*, 2012.

[29] Nan Ding and Radu Soricut. Cold-start reinforcement learning with softmax policy gradient. In *Advances in Neural Information Processing Systems*, pages 2817–2826, 2017.

[30] William B Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proc. of IWP*, 2005.

[31] Li Dong and Mirella Lapata. Language to logical form with neural attention. *ACL*, 2016.

[32] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. *CoRR*, abs/1805.04793, 2018.

[33] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv:1802.01561*, 2018.

[34] Brian Falkenhainer, Kenneth D Forbus, and Dedre Gentner. The structure-mapping engine: Algorithm and examples. *Artificial intelligence*, 41(1):1–63, 1989.

[35] Simone Filice, Giovanni Da San Martino, and Alessandro Moschitti. Structural representations for learning relations between pairs of texts. In *ACL*, 2015.

[36] K Forbus. *Exploring analogy in the large*. MIT Press, 2001.

[37] Kenneth D Forbus and Johan De Kleer. *Building problem solvers*, volume 1. MIT press, 1993.

[38] Kenneth D Forbus, Ronald W Ferguson, Andrew Lovett, and Dedre Gentner. Extending sme to handle large-scale cognitive modeling. *Cognitive Science*, 41(5):1152–1201, 2017.

[39] Kenneth D Forbus, Dedre Gentner, and Keith Law. Mac/fac: A model of similarity-based retrieval. *Cognitive science*, 19(2):141–205, 1995.

[40] Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. Ppdb: The paraphrase database. In *NAACL*, 2013.

[41] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive science*, 7(2):155–170, 1983.

[42] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7:155–170, 1983.

[43] Dedre Gentner. analogical learning. *Similarity and analogical reasoning*, page 199, 1989.

[44] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[45] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv:1410.5401*, 2014.

[46] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio G. Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, AdriÃ P. Badia, Karl M. Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

[47] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

[48] Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *ACL*, 2017.

[49] Till Haug, Octavian-Eugen Ganea, and Paulina Grnarova. Neural multi-step reasoning for question answering on semi-structured tables. In *ECIR*, 2018.

[50] Hua He, Kevin Gimpel, and Jimmy Lin. Multi-perspective sentence similarity modeling with convolutional neural networks. In *EMNLP*, 2015.

[51] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations. *AAAI*, 2018.

[52] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[53] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 1997.

[54] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[55] Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen tau Yih, and Xiaodong He. Natural language to structured query generation via meta-learning. *CoRR*, abs/1803.02400, 2018.

[56] Yangfeng Ji and Jacob Eisenstein. Discriminative improvements to distributional sentence similarity. In *EMNLP*, 2013.

[57] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *ACL*, 2016.

[58] J. Jiang, A. Teichert, J. Eisner, and H. Daume. Learned prioritization for trading off accuracy and speed. *NIPS*, 2012.

[59] Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David A Shamma, Michael Bernstein, and Li Fei-Fei. Image retrieval using scene graphs. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[60] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv:1511.08228*, 2015.

[61] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[62] Seyed Mehran Kazemi, David Buchman, Kristian Kersting, Sriraam Natarajan, and David Poole. Relational logistic regression. In *KR*, 2014.

[63] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[64] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[65] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. *EMNLP*, 2017.

[66] Ni Lao, Tom Mitchell, and William W Cohen. Random walk inference and learning in a large scale knowledge base. *EMNLP*, 2011.

[67] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[68] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[69] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv:1606.01541*, 2016.

[70] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*, 2016.

[71] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *ACL*, 2017.

[72] Chen Liang and Kenneth D. Forbus. Learning plausible inferences from semantic web knowledge by combining analogical generalization with structured logistic regression. In *AAAI*, 2015.

[73] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. *ACL*, 2011.

[74] Percy Liang, Alexandre Bouchard-Côté, Dan Klein, and Ben Taskar. An end-to-end discriminative approach to machine translation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 761–768. Association for Computational Linguistics, 2006.

[75] Percy Liang, Alexandre Bouchard-Côté, Dan Klein, and Benjamin Taskar. An end-to-end discriminative approach to machine translation. In *ACL*, 2006.

[76] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[77] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[78] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL*, 2014.

[79] Matthew D McLure, Scott E Friedman, and Kenneth D Forbus. Extending analogical generalization with near-misses. In *AAAI*, pages 565–571, 2015.

[80] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[81] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[82] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[83] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[84] Richard Montague. English as a formal language. 1970.

[85] Richard Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

[86] Thomas Mostek, Kenneth D Forbus, and Cara Meverden. Dynamic case creation and expansion for analogical reasoning. In *AAAI/IAAI*, pages 323–329, 2000.

[87] Pramod Kaushik Mudrakarta, Ankur Taly, Mukund Sundararajan, and Kedar Dhamdhere. It was the training data pruning too! *arXiv:1803.04579*, 2018.

[88] J. William Murdock. Structure mapping for jeopardy! clues. In *ICCBR*, 2011.

[89] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2775–2785, 2017.

[90] Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew D McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. *arXiv:1611.08945*, 2016.

[91] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

[92] Allen Newell and Herbert A Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[93] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv:1804.03720*, 2018.

[94] Mohammad Norouzi, Samy Bengio, zhifeng Chen, Navdeep Jaitly, Mike Schuster, Yonghui Wu, and Dale Schuurmans. Reward augmented maximum likelihood for neural structured prediction. *NIPS*, 2016.

[95] Mohammad Norouzi, Samy Bengio, Navdeep Jaitly, Mike Schuster, Yonghui Wu, Dale Schuurmans, et al. Reward augmented maximum likelihood for neural structured prediction. In *Advances In Neural Information Processing Systems*, pages 1723–1731, 2016.

[96] Sebastian Nowozin, Christoph H Lampert, et al. Structured learning and prediction in computer vision. *Foundations and Trends® in Computer Graphics and Vision*, 6(3–4):185–365, 2011.

[97] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. *ICML*, 2018.

[98] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*, 2015.

[99] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *ACL*, 2015.

[100] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *ACL*, 2015.

[101] Panupong Pasupat and Percy Liang. Inferring logical forms from denotations. *ACL*, 2016.

[102] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

[103] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. *EMNLP*, 2014.

[104] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. *IROS*, 2006.

[105] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008.

[106] Alexandrin Popescul and Lyle H Ungar. Structural logistic regression for link analysis. *Departmental Papers (CIS)*, page 133, 2003.

[107] Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. *NIPS*, 2017.

[108] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *ICLR*, 2016.

[109] Scott Reed and Nando de Freitas. Neural programmer-interpreters. *ICLR*, 2016.

[110] Bradley L Richards and Raymond J Mooney. *Learning relations by pathfinding*. Artificial Intelligence Laboratory, University of Texas at Austin, 1992.

[111] Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomá s Kociský, and Phil Blunsom. Reasoning about entailment with neural attention. *CoRR*, abs/1509.06664, 2015.

[112] S. Ross, G. Gordon, and A. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *Artificial Intelligence and Statistics (AISTATS)*, 2011.

[113] Nicolas Le Roux. Tighter bounds lead to improved classifiers. *ICLR*, 2017.

[114] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2010.

[115] Mark Sammons, V. G. Vinod Vydiswaran, Tim Vieira, Nikhil Johri, Ming-Wei Chang, Dan Goldwasser, Vivek Srikumar, Gourab Kundu, Yuancheng Tu, Kevin Small, Joshua Rule, Quang Do, and Dan Roth. Relation alignment for textual entailment recognition. In *TAC*, 2009.

[116] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:353–362, 1983.

[117] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *ICLR*, 2016.

[118] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

[119] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

[120] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. *ICML*, 2015.

[121] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[122] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[123] Abhishek B Sharma and Kenneth D Forbus. Graph-based reasoning and reinforcement learning for improving q/a performance in large knowledge-based systems. In *AAAI Fall Symposium: Commonsense Knowledge*, 2010.

[124] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[125] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 2017.

[126] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934, 2013.

[127] Richard Socher, Eric H. Huang, Jeffrey Pennington, Andrew Y. Ng, and Christopher D. Manning. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *NIPS*, 2011.

[128] Mark Steedman. *The syntactic process*, volume 24. MIT Press.

[129] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

[130] Xu Sun, Takuya Matsuzaki, and Daisuke Okanohara. Latent variable perceptron algorithm for structured classification.

[131] Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. Semantic parsing with syntax-and table-aware sql generation. *arXiv:1804.08338*, 2018.

[132] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *NIPS*, 2014.

[133] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[134] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, 2015.

[135] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2753–2762. Curran Associates, Inc., 2017.

[136] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *NIPS*, 2015.

[137] Ellen M. Voorhees. The TREC-8 question answering track report. In *TREC*, 1999.

[138] Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. Pointing out SQL queries from text. *ICLR*, 2018.

[139] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. *ACL*, 2015.

[140] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. 2014.

[141] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *ICLR*, 2017.

[142] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv:1410.3916*, 2014.

[143] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, pages 229–256, 1992.

[144] Sam Wiseman and Alexander M. Rush. Sequence-to-sequence learning as beam-search optimization. *arXiv:1606.02960*, 2016.

[145] Cathy Wu, Aravind Rajeswaran, Yan Duan, Vikash Kumar, Alexandre M Bayen, Sham Kakade, Igor Mordatch, and Pieter Abbeel. Variance reduction for policy gradient with action-dependent factorized baselines. *ICLR*, 2018.

[146] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144*, 2016.

[147] Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating structured queries from natural language without reinforcement learning. *ICLR*, 2018.

[148] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. *ACL*, 2015.

[149] Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. The value of semantic parse labeling for knowledge base question answering. *ACL*, 2016.

[150] Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. Neural enquirer: Learning to query tables. *arXiv:1512.00965*, 2015.

[151] Wenpeng Yin and Hinrich Schütze. Convolutional neural network for paraphrase identification. In *NAACL*, 2015.

[152] Adam Yu, Hongrae Lee, and Quoc Le. Learning to skim text. *ACL*, 2017.

[153] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv:1804.09769*, 2018.

[154] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *arXiv:1505.00521*, 2015.

[155] M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1050–1055, 1996.

[156] L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *Uncertainty in Artificial Intelligence (UAI)*, pages 658–666, 2005.

[157] L. S. Zettlemoyer and M. Collins. Online learning of relaxed CCG grammars for parsing to logical form. *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*, pages 678–687, 2007.

[158] Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*, 2012.

[159] DongQing Zhang and Shih-Fu Chang. Detecting image near-duplicate by stochastic attributed relational graph matching with learning. In *MM*, 2004.

[160] Yuchen Zhang, Panupong Pasupat, and Percy Liang. Macro grammars and holistic triggering for efficient semantic parsing. *ACL*, 2017.

[161] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103*, 2017.

[162] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2016.

[163] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv:1611.01578*, 2016.

[164] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv:1707.07012*, 2017.