12-2018

Improving Deep Reinforcement Learning Using Graph Convolution and Visual Domain Transfer

Sufeng Niu
Clemson University, sufengniu@gmail.com

Improving Deep Reinforcement Learning Using Graph Convolution and Visual Domain Transfer

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

by
Sufeng Niu
December 2018

Abstract

Recent developments in Deep Reinforcement Learning (DRL) have shown tremendous progress in robotics control, Atari games, board games such as Go, etc. However, model free DRL still has limited use cases due to its poor sampling efficiency and generalization on a variety of tasks. In this thesis, two particular drawbacks of DRL are investigated: 1) the poor generalization abilities of model free DRL. More specifically, how to generalize an agent's policy to unseen environments and generalize to task performance on different data representations (e.g. image based or graph based) 2) The reality gap issue in DRL. That is, how to effectively transfer a policy learned in a simulator to the real world.

This thesis makes several novel contributions to the field of DRL which are outlined sequentially in the following. Among these contributions is the generalized value iteration network (GVIN) algorithm, which is an end-to-end neural network planning module extending the work of Value Iteration Networks (VIN). GVIN emulates the value iteration algorithm by using a novel graph convolution operator, which enables GVIN to learn and plan on irregular spatial graphs. Additionally, this thesis proposes three novel, differentiable kernels as graph convolution operators and shows that the embedding-based kernel achieves the best performance. Furthermore, an improvement upon traditional n-step Q-learning that stabilizes training for VIN and GVIN is demonstrated. Additionally, the equivalence between GVIN and graph neural networks is outlined and shown that GVIN can be further extended to address both control and inference problems. The final subject which falls under the graph domain that is studied in this thesis is graph embeddings. Specifically, this work studies a general graph embedding framework GEM-F that unifies most of the previous graph embedding algorithms. Based on the contributions made during the analysis of GEM-F, a novel algorithm called WarpMap which outperforms DeepWalk and node2vec in the unsupervised learning settings is proposed.

The aforementioned reality gap in DRL prohibits a significant portion of research from reaching the real world setting. The latter part of this work studies and analyzes domain transfer techniques in an effort to bridge this gap. Typically, domain transfer in RL consists of representation transfer and policy transfer. In this work, the focus is on representation transfer for vision based applications. More specifically, aligning the feature representation from source domain to target domain in an unsupervised fashion. In this approach, a linear mapping function is considered to fuse modules that are trained in different domains. Proposed are two improved adversarial learning methods to enhance the training quality of the mapping function. Finally, the thesis demonstrates the effectiveness of domain alignment among different weather conditions in the CARLA autonomous driving simulator.

Dedication

This dissertation is dedicated to my wife, for always encouraging me and inspiring me, my parents, for their support and love, my dog and cat, for being the joy of my life. I also dedicate this dissertation to my academic advisor, committee members, and all of the teachers that shaped my research and career.

Acknowledgments

I would like to give my great thanks to many people who helped me along my path to completing this thesis.

Special thanks to my advisor Dr. Melissa C. Smith who helped me finish this long journey at Clemson. She encouraged me to explore my research interests, giving me high degree of freedom to explore creative ideas and innovations with many researchers. Her guidance and insights were key in shaping this thesis. I would like to thank my committee members for reviewing my dissertation work. I would also like to thank Dr. Smith, Dr. Luo for providing me with several opportunities to represent Clemson at top-tier conferences including (Association for the Advancement of Artificial Intelligence) AAAI, (Neural Information Processing System) NIPS, and IEEE Big Data. Many thanks to the Department of Defense (DoD) for their support with the research grant (DoD HFEHRI), and resources provided by Palmetto Cluster. Special thanks to the Future Computing Technologies (FCTLab) group here at Clemson University for all the pleasant and fun-filled discussions.

Finally, I would like to thank my wife, Jiajing Niu, who sacrificed plenty and tolerant my frustration much of the time, my parents who were always there to provide support and encouragement when I hit rock bottom. Without their unconditional love and patience, none of this would have been possible. Thanks to all of you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past decade, the digital revolution has enabled vast amounts of data to be accessible. Massive amounts of data created by social media and scientific/engineering applications open new doors for analyzing economic value, unearthing scientific truth, and improving customer services. Data processing and data analysis have become a crucial challenge. Research in academia and industry is mainly concentrated on how to extract knowledge from big data using techniques including Machine Learning, data mining, operational research, etc [3, 96, 93].

Machine Learning is the study of how to design a machine that can learn from data. Industry leaders have invested significant effort into mining latent information from available data via machine learning techniques. However, traditional machine learning capabilities have reached an upper bound, even after increasing the scale of datasets. In general, Machine Learning is categorized as Supervised Learning, Unsupervised Learning, and Reinforcement Learning (RL). Supervised Learning algorithms analyze the training data and infers a function mapping to predict unseen instances. Unsupervised Learning finds hidden structure in unlabeled data, such as clustering, dimensional reduction, feature extraction and so on. RL is concerned with how an agent should react in an environment so that the agent can maximize accumulated rewards.

A methodology known as Deep Learning has emerged as a promising research area that outperforms state-of-the-art published results and approaches human-level performance in several applications. Deep learning is positioned for a promising future in Big Data analytics, allowing computational models composed of a multitude of processing layers to learn representations of data through several levels of abstraction. Deep learning has achieved tremendous success in numerous supervised learning tasks, including speech recognition [5, 106], object recognition [51, 127, 72], semantic segmentation [98], natural language processing [122, 55, 73] and much more.

Reinforcement learning (RL) is a learning technique that solves sequential decision making problems that lack explicit rules and labels [124]. Recent developments in Deep Reinforcement Learning (DRL) have lead to enormous progress in autonomous driving [13], innovation in robot control [78], and human-level performance in both Atari games [87, 48] and the board game Go [115]. Given a reinforcement learning task, the agent explores the underlying Markov Decision Process (MDP) [11, 12] and attempts to learn a mapping of high-dimensional state space data to an optimal policy that maximizes the expected return.

In this thesis, we are specially interested in the Deep Reinforcement Learning (DRL) technique, which introduces Deep Learning into Reinforcement Learning settings. Conventional RL use tabular representation to hold the model information, whereas DRL uses neural networks as a function approximator to backup the policy information. Under the theory of RL, one of the important factors is whether the agent has access to (or learns) a model of the environment. By the model, we mean the function of state transitions and rewards. Algorithms that use a model are called model-based RL and those that do not are called model-free RL. If the agent knows the model beforehand, the agent could plan ahead, which means see what would happen in the future and decide among multiple choices. However, in practice, the true model of the environment is not available to the agent. In this case, the agent must learn the model from the sampled data, which is also called system identification. The biggest drawback is that the learned model often is often biased or inaccurate, resulting an agent policy that is optimized on the learned model and behaves sub-optimally.

On the other hand, model-free RL is more popular in the DRL community since they are easier to implement and do not encounter the model compounded error issue found in model-based RL. The samples from policy are unbiased true experiences, which are used to update the policy network. Typically, there are three approaches to train the agent by using model-free RL: policy-based, value-based, and actor-critic. The policy-based and actor-critic methods are usually on-policy learning, meaning that the sampled data is directly used to train the current policy function. Value-based methods, such as Q-learning or Deep Deterministic Policy Gradient (DDPG), are usually off-policy learning, that is learning current policy by using the samples from other policies, such as old policy [88] or expert's policy [56]. In practice, on-policy learning normally provides more stable training, whereas off-policy reuse data more effectively, offering better sample efficiency.

1.1 Dissertation Research

Even though model-free RL has been popular in the research community and shows great success in variety of applications, recent researchers [61, 82] show that the current DRL approach is not a path towards Artifical General Intelligence (AGI). In the following, we list limitations from a variety perspectives:

- **Model-free RL shows poor sample efficiency**: the task of RL aims to maximize the accumulated rewards. In most applications, reward signals are very sparse, such that when a policy search is conducted on the parameter space, most of time the agent receives no feedback at all. In other words, it is hard to track any gradient information during optimization since the policy is searching a Kronecker delta function. Therefore, the agent spends most of the sampled trajectories doing random search, which is highly inefficient. Additionally, DRL becomes worse when employing a neural network as the function approximator: current Deep Learning is inherently data hungry, that is deep learning requires large-scale datasets with high-quality labels even in supervised learning settings.

- **Model-free RL is hard to train**: When using a neural network as a non-linear function approximator, the convergence of a model-free RL algorithm cannot be guaranteed theoretically. A comparison is shown in Table 1.1. The main reason that RL is difficult to train is that it is a multi-level optimization problem. The inner loop optimizes the policy to maximize the total returns and the outer loop optimize the parameters of the policy network to minimize the gradient loss.

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Table 1.1: Algorithm Convergence [125]

- **Model-free RL shows poor generalization**: [63] shows that even a tiny visual differences or environment variations could lead to total failure of a state-of-the-art model-free DRL. The model-free DRL makes no assumption about the domain structures since the deep neural network system is simply memorizing the mapping relationships between inputs and outputs, thus it is capable of highly limited generalization, reasoning, and planning capabilities.

- **Model-based RL has inherent problem of modeling errors**: The advantage of model-based RL is that it improves the sample efficiency compared with a model-free RL. However, the model-based method requires system identification step to map the observations to a dynamics model. Then, a learned model is used to solve the policy. In many applications, such as robotic manipulation and locomotion, accurate system identification is difficult, and modelling errors can severely degrade the policy performance.
- **DRL has a reality gap issue**: The pre-assumption of RL is that the agent is designed to interact with the environment, and in many business applications, it is too expensive or even impossible for an agent to interact with hypothetical trial and error scenarios. For example, training an autonomous car in the real world might damage the car or hit the pedestrian. The central question is how to solve this reality gap issue. That is, how to effectively transfer a policy learned in a simulator to the real world.

The aforementioned limitations are also current active research topics. Methods such as Hindsight Experience Replay (HER) [6], model-based RL [116], and planning algorithms [94] are proposed to solve the sample efficiency. The combination of model-based RL and model-free RL [145, 116] could further improve the compound error of learned environment dynamics. Meta learning [34] and planning [94, 128] could effectively generalize to environment variants. Also, transfer learning and meta learning provide the possibility to transfer the learned knowledge from a simulator to the real world.

1.2 Methods of Study

Among these limitations and their corresponding research proposals, one question remains: what is the fundamental problem behind these drawbacks? And is there an universal approach to solving the above problems that shares the same properties? Further, we think this is also one of the key factors towards AGI. The central theme among the above problems is that state-of-the-art deep learning models lack, ironically, a 'model' from which to learn: when humans begin solve a complicated task, he or she often first builds a world knowledge or 'model' in his or her mind, then performs inference or planning upon that model. The model can be naturally represented as a graph. The graph is a discrete data structure model of the instance relationships. Unlike current deep learning systems, which learn the function mapping between two variables, a graphical model takes consideration of the dependencies among latent variables, the relationships among the latent variables essentially forms the conditional probability or potential function, which is already widely used in the probabilistic graphical model (PGM) [69]. However, the inference in PGM is commonly intractable, using different approximation or reduction techniques. On the other hand, a neural network introduces a fully differentiable method to do inference based on backpropagation in a convenient way. Then, it is natural to combine the graphical model and neural network together to model the dependencies, while using a neural network for inference task.

Additionally, the reality gap is a typical problem of domain transfer/adaption. Ideally, we are interested in learning adaption with a set of unlabeled source examples and a set of unlabeled target examples. The goal is to find or construct a common representation space for the two domains. One possible way is by using adversarial learning methods [45], which learn feature representations from samples in different domains that are indistinguishable.

According to this analysis, our research can be abstracted as three sub-tasks:

- **Dynamic graph planning based on neural networks**: Here dynamic means the graph contains unknown dynamics, for example, traffic on the map, or network bandwidth state on network routing. The graph can be viewed as a semi-model and the model transition probability remains unknown. Therefore, we explore a RL model between model-free and model-based.
- **Graph embedding**: In the previous bullet, we assume that the graph structure features are given a priori. Our research investigates how to obtain the structure features from the raw graph for a variety of tasks including control, classification, clustering, etc.
- **Domain adaption between different environments**: How to bridge the gap between RL training in a simulator to a real world environment without any supervision signal. Also, for practical reason, our interests also lie on the domain transfer with light computational cost.

1.2.1 Dynamic graph planning based on neural networks

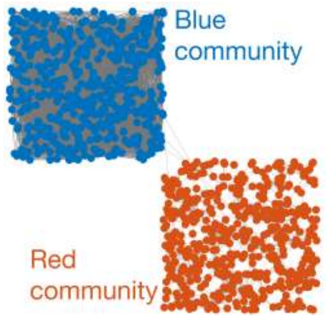
RL can be categorized as model-free [80, 86, 87] and model-based approaches [124, 30, 108]. Model-free approaches learn the policy directly by trial-and-error and attempt to avoid bias caused by a suboptimal environment model [124]. Model-based approaches, on the other hand, allow for an agent to explicitly learn the mechanisms of an environment, which can lead to strong generalization abilities. A recent work, the value iteration networks (VIN) [128] combines recurrent convolutional neural networks and max-pooling to emulate the process of value iteration [11, 12]. As VIN learns an environment, it can plan shortest paths for unseen mazes.

The input data fed into deep learning systems is usually associated with regular structures. For example, speech signals and natural language have an underlying 1D sequential structure; images have an underlying 2D lattice structure. To take advantage of this regularly structured data, deep learning uses a series of basic operations defined for the regular domain, such as convolution and uniform pooling. However, not all data is contained in regular structures. In urban science, traffic information is associated with road networks; in neuroscience, brain activity is associated with brain connectivity networks; in social sciences, users' profile information is associated with social networks. To learn from data with irregular structure, some recent works have extended the lattice structure to general graphs [29, 68] and redefined convolution and pooling operations on graphs; however, most works only evaluate data that has both a fixed and given graph. In addition, most lack the ability to generalize to new, unseen environments.

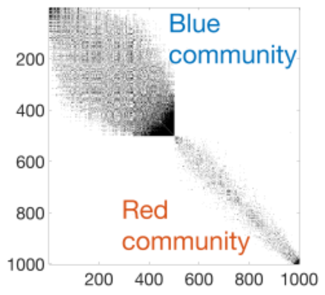
In this subsection of research, we aim to enable an agent to self-learn and plan the optimal path in new, unseen spatial graphs by using model-based DRL and graph-based techniques. This task is relevant to many real-world applications, such as route planning of self-driving cars and web crawling/navigation. The proposed method is more general than classical DRL, extending for irregular structures. Furthermore, the proposed method is scalable (computational complexity is proportional to the number of edges in the testing graph), handles various edge weight settings and adaptively learns the environment model. Note that the optimal path can be self-defined, and is not necessarily the shortest one. Additionally, the proposed work differs from conventional planning algorithms; for example, Dijkstra's algorithm requires a known model, while our proposed method aims to learn a general model via trial and error, then apply said model to new, unseen irregular graphs.

1.2.2 Graph embedding

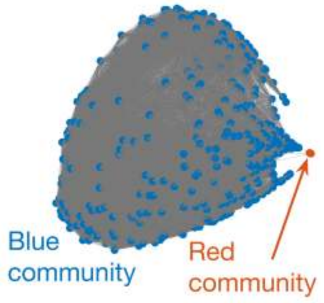
A graph embedding is a collection of feature vectors associated with nodes in a graph; each feature vector describes the overall role of the corresponding node. Through a graph embedding as shown in Figure 1.1, we are able to visualize a graph in a 2D/3D space and transform problems from a non-Euclidean space to a Euclidean space, where numerous machine learning and data mining tools can be applied. The applications of graph embedding include graph visualization [54], graph clustering [141], node classification [111, 151], link prediction [79, 8], recommendation [148], anomaly detection [4], and many others. A desirable graph embedding algorithm should (1) work for various types of graphs, including directed, undirected, weighted, unweighted and bipartite; (2) preserve the symmetry between the graph node domain and the graph embedding domain, that is, when two nodes are close/far away in the graph node domain, their corresponding embeddings are close/far away in the graph embedding domain; (3) be scalable to large graphs; and (4) be interpretable, that is, we understand the role of each building block in the graph embedding algorithm.



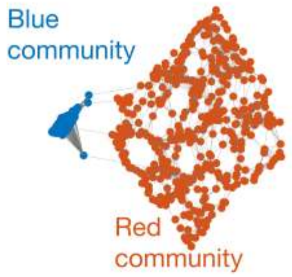
(a) Input graph.



(b) Adjacency matrix.



(c) Linear embedding through SVD.



(d) WarpMap.

Figure 1.1: Our proposed graph embedding algorithm WarpMap introduces a warping function, which significantly improves the quality of a graph embedding. We expect the nodes in the blue community should be close in proximity in the graph embedding domain because their connections are strong, which is shown by WarpMap.

1.2.3 Domain adaption between different environments

Most DRL [87, 80, 74, 35] studies are conducted in a simulated environment from scratch. Training an agent in a real world environment is often too expensive or even unrealistic. An alternative is to apply the model learned from the simulator to the real world. However, directly adopting the model trained from the simulator to a real physical system commonly leads to poor performance. How to perform transfer from simulation to the real world plays a key role in robotics since the simulator can be a source of practically infinite cheap data with flawless labels.

Since multiple factors could contribute to performance degradation from the simulator to the real world, we summarize from the macro-view as two main points: 1) the mismatch of perception or sensory feature representation and 2) the policy bias that is caused by mismatched dynamics between the simulator and real environment.

The model of the agent can be partitioned as two parts: the representation module and the policy module. The representation module performs feature extraction, and the policy module could be either model-free or model-based [124]. Many recent efforts in RL domain transfer [129] focus on the policy transfer, where their applications simplify state by using low dimensional sensor input data such as IMU. On the other hand, most of the work regarding representation transfer [136, 14] in RL settings attempts to learn the generator function to adapt the images from the source domain to appear as possible to those sampled from the target domain. However, this method is relatively cumbersome and it is typically specific to robotics.

As [113] demonstrated, good representation allows a trained policy to be quickly adapted and recovered even when the policy network is obliterated. The robust representation greatly eases the learning difficulties when performing domain transfer in RL problems. In this research, we propose to develop an adversarial learning for the feature-level domain transfer, in which the features from the source domain can be aligned to the target domain without any labels. Particularly, our work mainly focuses on image-based input for the agent since the real world environment largely involves image-based state and the computer vision community already provides large amounts of data and advanced neural network based models for image classification [51], object detection [101, 51], and segmentation [22].

1.3 Contributions and Outline

The reminder of this dissertation is organized as follows: Chapter 2 provides background information related to Machine Learning, deep generative model, and RL. Chapter 3 presents the details and design architectures of the aforementioned three sub-taskes including GVIN, WarpMap graph embedding, and domain alignment based on adversarial learning. Chapter 4 presents literature review and related state-of-the-art work. Chapter 5 presents experiments and results, which we studies the empirical results and offers the corresponding analysis. Note that this chapter is also divided into three parts, each of them corresponding to one task in Chapter 3. Finally, conclusions and future work are discussed in Chapter 6.

Chapter 2

Background

In this chapter, we will cover the background knowledge used in this thesis. Much of the work in this thesis is based on Deep Learning, and hence we introduce Deep Learning techniques that are widely used in supervised learning. We will also review the concept of Deep Generative Model and its extensions, which is used in the proposed domain alignment algorithms for visual based input (Section 3.3). Additionally, we will go through the reinforcement learning basics and related state-of-the-art techniques, which is used in GVIN (Section 3.1). Finally, we will talk about the graph convolutional neural network (GCNN) and its extensions that are used later in Sections 3.1 and 3.2

2.1 Deep Learning

Deep Learning or deep neural networks (DNN) have shown great success in a variety of applications including image processing [72], speech processing [58], natural language processing [140], control [116] problems and others. These tasks can be modeled as either supervised learning or unsupervised learning. In this subsection, we will discuss popular supervised learning techniques, a variety of neural network architectures, and the auto differentiable system such as Tensorflow [3] which has lead to the explosive growth of Deep Learning applications.

2.1.1 Supervised Learning

From the point of view of supervised learning, many practical problems can be simply formulated as a computing mapping function $f\colon X \to Y$, where X is in the input space and Y is in the output space. Here we list several popular applications that fall into this mapping function framework:

- **Image recognition** in image recognition, X is the space of image and Y represents object probability
- **Machine translation** for machine translation, X is source language (ex: English) and Y is the target language (ex: German)
- **Image captioning** in image captioning, X is query image and Y is corresponding caption of that image
- **Speech recognition** in speech recognition, X represents the audio wave data and Y indicates the transcripts of the corresponding audio wave
- **Question answering** in question answering application, X denotes the questions and stories and Y is the answer

In supervised learning, all of the above tasks become searching for a function that can exactly or approximately map from X to Y. Concretely, we assume that we have a training set with n samples $\{< x_1, y_1 >, ..., < x_n, y_n >\}$ where $x_{1...n} \in X$ and $y_{1..g.n} \in Y$, and each sample is independent and identically distributed (i.i.d.). Then, the learning process aims to search over the mapping function f so that f maintains the consistent mapping among all $< x_i, y_i >$ pairs. Hence, we define an objective/loss function $\mathcal{L}(\hat{y}_i, y_i)$ to measure how consistent it is among the training set, where $\hat{y}_i$ represents the prediction from f, and y_i denotes the ground truth value. In the general case, the loss function can be defined as:

$$f^* = \arg\min_f \mathbb{E}_{<x,y>} \mathcal{L}(f(x), y) \tag{2.1}$$

where f^* is the optimal function configuration; $\mathcal{L}(\cdot, \cdot)$ represents the loss function; and $\hat{y} = f(x)$ denotes the prediction. Equation 2.1 indicates that the goal is to find the configuration of f to minimize the expected loss over the training sample. Across the machine learning community, f has a variety of hypotheses. For example, in the linear model, $f = x^\intercal \theta + \epsilon$ and the configuration is parameter θ. In the decision tree model, f is a tree, and the configuration indicates how the tree is split based on the loss criteria. In a neural network, f is a multi-layer perceptron and the parameters of the perceptron represent the configuration.

Searching the parameters of the function f is an optimization problem. Unfortunately, optimizing on Equation 2.1 often yields poor performance on different datasets, such as the testing set versus real world deployment. In other words, the optimized f does not *generalize* to all $< x, y >$ pairs, which is often referred to as overfitting. In order to choose a generalized solution from the set of optimized f configurations, regularization techniques are introduced to overcome overfitting. Therefore, the Equation 2.1 is written as:

$$f^* = \arg\min_f \mathbb{E}_{<x,y>}[\mathcal{L}(f(x), y)] + R(f) \tag{2.2}$$

where $R(f)$ is a scalar value added to the original loss function. Regularization follows the principle of Occam's razor: "Suppose there exist two explanations for an occurrence. In this case, the one that requires the least speculation is usually better". In Equation 2.2, $R(f)$ measures the model complexity. During the optimization, the loss function balances the objective loss and model complexity.

Nevertheless, directly optimizing Equation 2.2 is not tractable since the optimization needs to operate on all sample data. Therefore, stochastic gradient descent (SGD) is introduced to make the optimization scalable. Then, Equation 2.3 is more practical for ML applications:

$$f^* = \arg\min_f \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i), y_i) + R(f) \tag{2.3}$$

Several popular optimization methods for ML algorithms are provided in the following discussion.

Linear Regression Assuming we have a regression task such that the target label is a continuous value $Y \in \mathbb{R}$ and input features are m dimension $X \in \mathbb{R}^m$. The loss function uses the mean squared error (MSE) between the predicted value $\hat{y}$ and the target value y, where $\hat{y} = \theta^\top x + \theta_0$. Assuming the model follows a Gaussian, which is L2 regularization, putting each piece together, we have:

$$\theta^* = \arg\min_\theta \underbrace{\frac{1}{N} \sum_{i=1}^{N} ||\theta^\top x_i + \theta_0 - y_i||^2}_{\text{training objective}} + \lambda \underbrace{\sum_{j=0}^{m} ||\theta_j||^2}_{\text{regularization}} \tag{2.4}$$

λ is the coefficient that determines how much regularization effect should be added. Equation 2.4 is also called the ridge regression. When the feature exhibits the sparsity property, the model can use the Laplace prior, which is L1 regularization, and we could replace the term $\sum_{j=0}^{m} ||\theta_j||^2$ with $\sum_{j=0}^{m} ||\theta_j||$. The optimization of linear regression has a closed form solution and can be solved in one-step by $\theta^* = (X^\top X)^{-1} X^\top y$.

Logistic Regression Logistic regression is a classical model for solving the binary classification problem. Consider Y to be a discrete value, specifically in Logistic regression it is a binary value $\{0, 1\}$. The hypothesis is $\hat{y} = \frac{1}{1+e^{-\phi(x)})}$ and $\phi(x) = \theta^\top x + \theta_0$. To optimize the parameters of logistic regression, the loss function contains binary cross-entropy as the objective function and L2 norm as regularization:

$$\theta^* = \arg\min_\theta \underbrace{\frac{1}{N}\sum_{i=1}^{N}(-y_i * \log \hat{y}_i - (1-y_i) * \log(1-\hat{y}_i))}_{\text{training objective}} + \underbrace{\lambda \sum_{j=0}^{m} ||\theta_j||^2}_{\text{regularization}} \tag{2.5}$$

The optimization upon θ, which we call training in later sections, uses SGD to update the parameter by $\theta_{new} = \theta_{old} - \alpha \nabla \mathcal{L}(\theta_{old})$, where α is a hyper-parameter called the learning rate that defines the gradient step size in each iteration.

Support Vector Machine Classification Consider a binary classification problem with labels y and features x. We will use $y \in \{-1, 1\}$ to denote the class labels. Given each training sample $< x_i, y_i >$, SVM aims to find the optimal or robust decision boundary that maximize the geometric margins. Assuming the parameter $\theta = \{w, b\}$, the loss function becomes a constrained optimization:

$$\begin{aligned} &\max_{w,b} \frac{1}{2}||w||^2 \\ &\text{s.t. } y_i(w^\top x_i + b) \geq 1, i = 1, ..., m \end{aligned} \tag{2.6}$$

Typically, by using Lagrange duality, the above quadratic programming (Equation 2.6) can be eventually constructed as follows:

$$\mathcal{L}(w, b) = \underbrace{\frac{1}{2}||w||^2}_{\text{regularization}} + \underbrace{C \sum_{i=1}^{m} \max(0, 1 - y_i(w^\top x_i + b))}_{\text{training objective}} \tag{2.7}$$

The above loss function (Equation 2.7) is also called hinge loss. However, the operator $\max(\cdot, \cdot)$ is not differentiable. The sub-gradient is computed to optimize the objective function.

Neural Network Classification Neural networks extend logistic regression to a more complex hypothesis. Logistic regression could be viewed as a special case of neural networks that contains one neuron output with a single layer. Neural networks stack multiple layers and we are free to choose from a variety of non-linearity activation functions including sigmoid, hyperbolic tangent tanh, rectifier relu, etc. The non-linearity enables neural networks with more powerful model expressiveness. [114] shows that a single layer of sigmoid or tanh functions is Turing-complete, which means the neural network can approximate arbitrary function mapping. Similarly, the loss function of the neural network could also be formed as Equation 2.3. However, with direct multi-layer optimization it is difficult to search the optimal parameters via SGD. The

Backpropagation algorithm allows the gradient to be passed back to the previous layer based on the chain rule, updating the neural network weights layer by layer. Therefore, Backpropagation enables the research community to train very large scale networks for a variety of applications, such as image processing, speech processing, text processing, etc.

2.1.2 Different Neural Architectures

Here we briefly show three basic and commonly used neural network architectures in different applications.

2.1.2.1 Multi-layer Neural Network

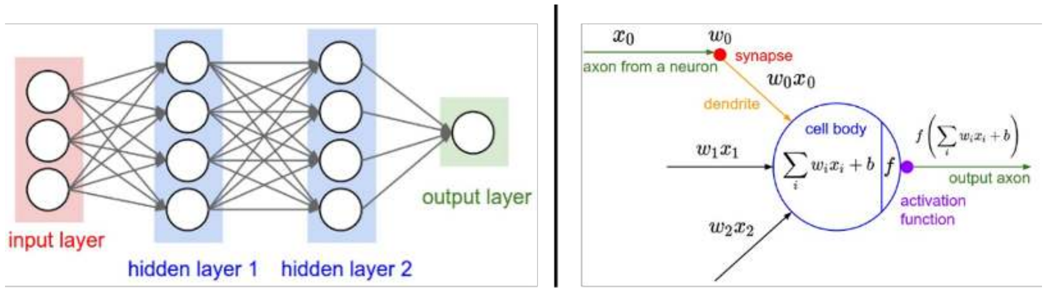


Figure 2.1: Multi-layer neural network model [1]

Multi-layer neural networks receive the input feature and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where the neurons in a single layer are completely independent (do not share connections) as shown in Figure 2.1. Each neuron is a linear transformation of the input features with non-linear activation. In a conventional multi-layer perceptron, the sigmoid is a popular activation function but recent models show that the sigmoid experiences a gradient vanishing problem and is therefore replaced with a rectifier function.

2.1.2.2 Convolutional Neural Network

Convolution neural networks (CNN) specifically handle the data that consists of the spatial invariant property, such as image, video, etc. Figure 2.2 shows an example of image data to illustrate its mechanism. The input color image is of size $128 \times 128 \times 3$ (length and height are 128, 3 color channels for rgb). The input image is convolved with CNN learned filters and each channel is independently computed. However, directly applying a fully connected neural network requires a very large number of parameters. Instead, CNN require

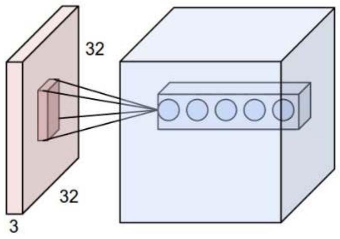


Figure 2.2: Convolutional neural network model [1]

two assumptions: 1) image data has local connectivity, that is the current pixel is only related to its local region of the input, and 2) the parameters in the sliding window is shared among the entire image. These two assumptions allow for a dramatic reduction in the number of parameters for CNN.

2.1.2.3 Recurrent Neural Network

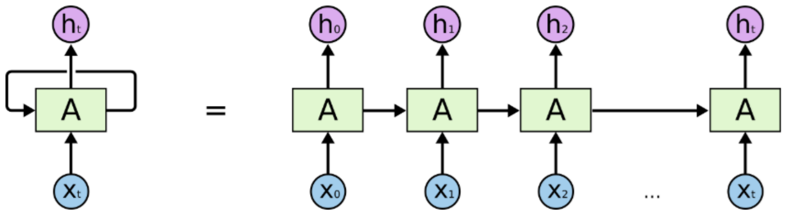


Figure 2.3: Recurrent neural network model [2]

Many applications contain a sequence of data, such as time series, natural language, speech recognition, etc. A recurrent neural network (RNN) shown in Figure 2.3 accepts input sequence $\{x_1, ...x_T\}$ and has the feedback loop using a recurrence formula $h_t = f(h_{t-1}, x_t; \theta)$. The parameter θ is shared at every time step, allowing the network to process arbitrary sequence lengths. As shown in Figure 2.3, RNNs can be unfolded as a chained feed-forward neural network, which could potentially cause the gradient vanishing problem. In order to overcome the gradient drawbacks, long short term memory (LSTM) [59] and gated recurrent units (GRU) [25] utilize gating and memory to address the limitations. LSTM and GRU model are summarized as:

$$
\begin{array}{ll}
\text{LSTM} & \text{GRU} \\
f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) & \\
i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) & z_t = \sigma(W_z[h_{t-1}, x_t]) \\
\bar{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) & r_t = \sigma(W_r[h_{t-1}, x_t]) \\
C_t = f_t * C_{t-1} + i_t * \bar{C}_t & \bar{h}_t = \tanh(W[r_t * h_{t-1}, x_t]) \\
o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) & h_t = (1 - z_t) * h_{t-1} + z_t * \bar{h}_t \\
h_t = o_t * \tanh(C_t) &
\end{array}
$$

2.1.3 Auto Differentiable System

Neural networks provide an elegant way to train the deep network by the chain rule. It is natural to think about the neural network as a computational graph, which lead the research community to redefine the artificial neural network. The graph is not necessary limited to the bipartite graph, but rather it could be any directed acyclic graph (DAG) of operations: input feature vectors flow along edges and vertices which are differentiable operators, output flows out to another differentiable module or loss function. In the implementation, the vertices objects implement forward() and backward(·) function, where forward(·) function propagates the graph in topological order producing the prediction based on the network parameters, and backward() function passes the gradient from the next layer to the previous layer in reverse order. When vertices among the computational graph are connected, each gradient in the neuron are chained and training can be fully automated.

The computational graph concept is one of biggest reasons why Deep Learning has been such a significant achievement: 1) it allows developers to define neural networks in a modular fashion, where each module can be highly customized to specific problems; and 2) the developer does not need to derive backpropagation from scratch, many auto differentiable systems such as Tensorflow [3] and Pytorch [96] automate the backpropagation process for users.

2.2 Deep Generative Model

Often typical generative models consist of two types of variables: observed/visible variable x and latent/hidden variables z. x is commonly pre-given during the training phase, where variable z is used to describe the structure in x. z depends on the model parameters that are trained through the learning algorithms.

Generative models directly learn the data distribution which can be used in variety of scenarios. Figure 2.4 shows the result from a few samples generated from a generative adversarial network (GAN) [45]; note that every picture is generated via the neural network and the input is random noise.

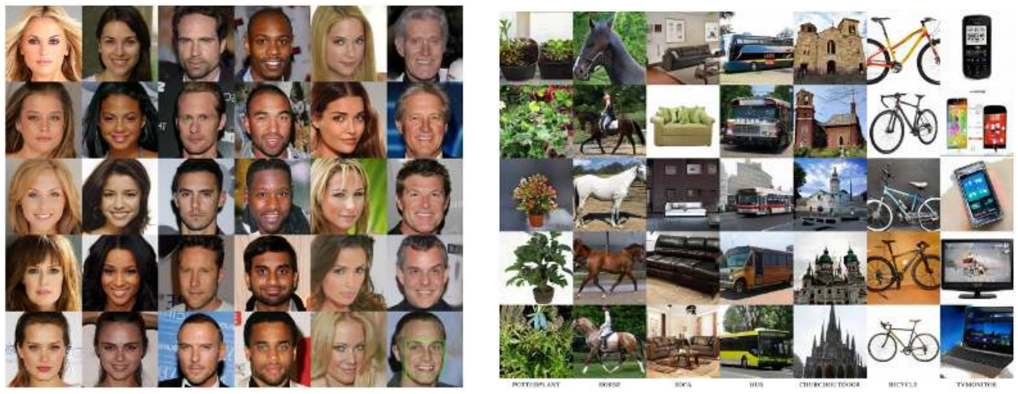


Figure 2.4: Generative adverserial network (GAN) results from [64]. Note that both samples above are generated samples from a neural network, that is to say, none of the figures actually exist in the real world. All pictures are "created" or drawn via a Deep Generative Model, which is a GAN in this case.

We use the following notations:

- x represents an observed random variable
- $p(x)$ represents the probability over the variable x
- $X \sim p(x)$ denotes a value X sampled from the probability distribution $p(x)$; symbol ~ denotes sampling
- $P(x)$ represents the density function of the distribution of x

According to the Bayesian rule:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} = \frac{p(x,z)}{\int p(x,z)dz} \tag{2.8}$$

where $p(x|z)$ is the likelihood, $p(z)$ is the prior probability, $p(z|x)$ is the posterior probability, and $p(x)$ is the evidence. Equation 2.8 represents how to perform an inference over the graphical model. $\int p(x,z)dz$ integrates all possible configurations over the latent variables, which is a NP-hard problem. To overcome the computational limitations, the research community proposes exact inference and approximate inference. Since we are mainly focused on the approximate inference approach, we omit background information for exact inference research.

Approximate inference could be divided into two types: Monte Carlo Markov Chain (MCMC) and variational inference. Instead of computing the true posterior distribution, MCMC constructs a markov chain to obtain samples of the desired distribution after a number of steps. While variational inference provides an approximation to the posterior based on known distribution. Since our research utilizes deep implicit density estimation (ex: GAN), we only discuss the variational inference and variational autoencoder that is currently popular among the Deep Learning domains.

2.2.1 Variational Inference

The posterior distribution $p(z|x)$ is often computationally intractable in many models and thereby we use a parametric distribution $q_\phi(z|x)$ such as a Gaussian to approximate the true distribution $p(z|x)$. Performing inference on the known distribution is computationally tractable, then our task has converted inference as an optimization problem where we train the parameters ϕ so that q is as close to p as possible.

To measure the distance between q and p, variational inference uses the reverse KL Divergence:

$$KL(q||p) = \sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z|x)} \tag{2.9}$$

where $p(z|x) = \frac{p(x,z)}{p(x)}$ is from Equation 2.8, then:

$$\begin{aligned} KL(q||p) &= \sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)p(x)}{p(z,x)} \\ &= \sum_z q_\phi(z|x)(\log \frac{q_\phi(z|x)}{p(z,x)} + \log p(z)) \\ &= \left(\sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z,x)}\right) + \left(\sum_z \log p(x) q_\phi(z|x)\right) \\ &= \left(\sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z,x)}\right) + \left(\log p(x) \sum_z q_\phi(z|x)\right) \\ &= \left(\sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z,x)}\right) + \log p(x) \end{aligned} \tag{2.10}$$

Therefore, in order to minimize $KL(q||p)$ over parameters ϕ, the objective is to minimize the term

$\sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z,x)}$ since $\log p(x)$ is fixed. Therefore, the objective function becomes:

$$\begin{aligned}\sum_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z,x)} &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log \frac{q_\phi(z|x)}{p(z,x)}] \\ &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log q_\phi(z|x) - \log p(z,x)] \\ &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log q_\phi(z|x) - \log p(x|z) - \log p(z)] \end{aligned} \tag{2.11}$$

Here, minimizing the above equation is equivalent to maximizing:

$$\begin{aligned}\max \mathcal{L} &= \mathbb{E}_{z\sim q_\phi(z|x)}[-\log q_\phi(z|x) + \log p(x|z) + \log p(z)] \\ &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x|z) + \log \frac{p(z)}{q_\phi(z|x)}] \end{aligned} \tag{2.12}$$

where $\mathcal{L}$ is variational evidence lower bound (ELBO), which is computationally feasible since we can evaluate $p(x|z)$, $p(z)$ and $q(z|x)$. The ELBO could be reformulated as:

$$\begin{aligned}\mathcal{L} &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x|z) + \log \frac{p(z)}{q_\phi(z|x)}] \\ &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x|z)] - KL(Q(z|x)||P(z)) \end{aligned} \tag{2.13}$$

In another point of view, when substituting $\mathcal{L}$ back to Equation 2.9, we have:

$$\begin{aligned} KL(q||p) &= \log p(x) - \mathcal{L} \\ \log p(x) &= \mathcal{L} + KL(q||p) \end{aligned} \tag{2.14}$$

where $\log p(x)$ is the log likelihood of true data distribution and it remains constant over the parameters ϕ. Therefore, to minimize the KL divergence $KL(q||p)$ as $KL(q||p) \geq 0$ is equivalent to maximizing $\mathcal{L}$. In summary, the variational inference intends to optimize the following objective function:

$$\mathcal{L} = \mathbb{E}_{z\sim q_\phi(z|x)}[\log p(x|z)] - KL(Q(z|x)||P(z)) \tag{2.15}$$

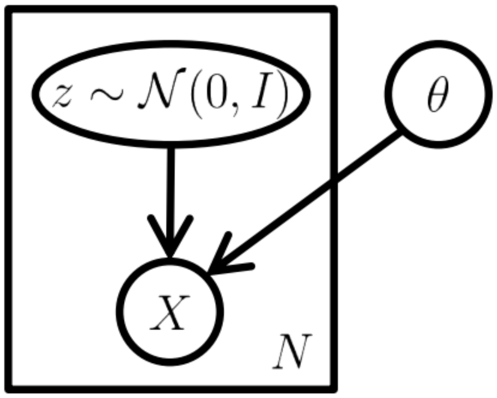


Figure 2.5: Graphical model of the generative process in VAE [32]

2.2.2 Variational Auto-Encoder

Variational auto encoder (VAE) [67] provides an elegant amortized inference approach extended from the variational inference. VAE use two neural networks to parameterize the variational parameters ϕ and generative process θ, where the graphical model is shown in Figure 2.5. Therefore, we rewrite the Equation 2.15 as follows for the optimization objective function:

$$\mathcal{L}(x; \theta, \phi) = \mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)] - KL(Q_\phi(z|x)||P_\theta(z)) \tag{2.16}$$

The first term $\mathbb{E}_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)]$ denotes the log likelihood that the Auto-Encoder can be used for reconstruction loss, where the second term is the regularization term. It describes that by using an arbitrarily high capacity model for $q(z|x)$, we adjust $q(z|x)$ to actually match $p(z|x)$; the KL-divergence term becomes zero when the perfect match.

The $KL(Q_\phi(z|x)||P_\theta(z)$ term is not analytically formed for computation. In practice, we assume both $Q_\phi(z|x)$ and $P_\theta(z)$ are two Gaussian distributions. Then the problem becomes a density estimation to infer the Gaussian parameters. Neural networks can be viewed as a powerful function approximator, therefore we can combine a simple distribution with a learned neural network to emulate an arbitrary distribution.

Therefore, the prior distribution could be set as $\mathcal{N}(0, I)$, where I is the identity matrix. For each latent Gaussian, the latent dimension could be in k dimension. To simplify the assumption, VAE assumes that each latent dimension is independent, where the covariance matrix is a diagonal matrix.

Therefore, the KL-divergence between two multivariate Gaussian distributions can be computed in closed form as follows:

$$KL[Q_\phi(z|x)||P_\theta(z)] = \int q_\phi(z|x)(\log p_\theta(z) - \log q_\phi(z|x))dz$$
$$= \frac{1}{2}\Sigma_{j=1}^{J}(1 + \log((\sigma_j)^2) - (\mu_j)^2 - (\sigma_j)^2) \tag{2.17}$$

where written in matrix form as follows:

$$KL[\mathcal{N}(\mu(x), \Sigma(x))||\mathcal{N}(0, I)] = \frac{1}{2}\left(\mathrm{tr}(\Sigma(x) + (\mu(x))^\top(\mu(x))) - k - \log\det(\Sigma(x))\right) \tag{2.18}$$

After we have the analytic computation form (Equation 2.18), the loss function in Equation 2.16 can be iteratively trained via the stochastic gradient descent algorithm. And VAE can fit into the Auto-encoder framework, where q_ϕ is the encoder and p_θ is the decoder.

However, one typical problem in VAE is that the sampling process is non-differentiable. To overcome the gradient estimation issue, [67] introduces a reparameterization trick: converting the sampling process to a differentiable operator as follows:

$$z = \mu(x) + \epsilon * \Sigma(x) \text{ where } \epsilon \sim \mathcal{N}(0, I) \tag{2.19}$$

Equation 2.19 is equivalent to $z \sim \mathcal{N}(\mu(x), \Sigma(x))$ and is widely used for continuous random variables. In the case of categorical distribution, Gumbel-softmax [62] can be adopted, which shares a similar idea as continuous variables. Assuming that each categorical variable probability is π_i, to draw the sample z, the reparameterization trick becomes:

$$z = \text{ one\_hot } (\arg\max_i[g_i + \log\pi_i]) \tag{2.20}$$

where g_i is the noise $g_i \sim$ Gumbel $(0, 1)$. Then, the VAE is fully differentiable in both continuous variables and discrete variables, and thereby could be directly trained based on the backpropagation algorithm.

2.2.3 Generative Adversarial Learning

VAE is an explicit density estimation where the latent distribution is a Gaussian distribution. Generative adversarial network (GAN) [45] is a recently proposed implicit density estimation method. Unlike the variational inference, which uses another distribution (ex: Gaussian) to approximate the posterior, GAN does not contain any explicit probability distributions, but instead, using one neural network to mimic any data distribution while another one to evaluate the distribution quality. The generator defines a process to simulate data and does not require tractable densities.

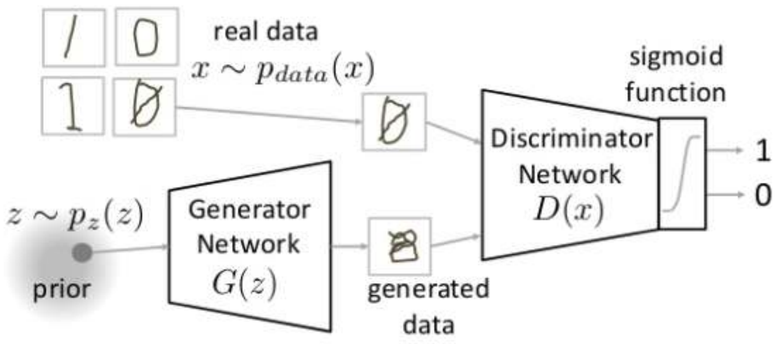


Figure 2.6: Generative Adversarial Network model

As shown in Figure 2.6, the GAN model consists of two neural networks: generator and discriminator. The input to the generator is from a simple prior, which could be a high dimensional vector generated from a random uniform distribution. To evaluate the generated samples, instead of using likelihood to evaluate the generator, GAN utilizes another neural network (discriminator) to measure how good the sample is. This becomes a two player game for implicit density estimation, where two networks are optimizing opposite objectives. The overall loss function is:

$$\min_G \max_D V(D,G) \tag{2.21}$$

where:

$$V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x;\theta_D)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z;\theta_G);\theta_D))] \tag{2.22}$$

$p_z(z)$ is the prior distribution, which could be set as $\mathcal{N}(0,I)$. Equation 2.22 is a standard binary cross entropy loss when training a standard binary classifier with a sigmoid output. However, compared with conventional classifier, the framework takes two minibatches of data where one is from the true distribution and the other one is from the generator. The intuition is that the discriminator tries to distinguish between the real data and generated one, whereas the generator tries to fool the discriminator that it cannot distinguish between them. The cost used for the discriminator is:

$$J^{(D)}(\theta_D, \theta_G) = -\frac{1}{2}\mathbb{E}_{x \sim p_{data}(x)}[\log D(x; \theta_D)] - \frac{1}{2}\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z; \theta_G); \theta_D))] \tag{2.23}$$

The generator and discriminator are parameterized through θ_G and θ_D. During the training phase, the generator and discriminator are learned iteratively. At each step, a minibatch of x values from the true dataset and a minibatch of a z values drawn from the generator's prior over latent variables are sampled, and then two stochastic gradient descent (SGD) steps are applied: one updating θ_D by minimizing $J^{(D)}$ and the other updating θ_G by maximizing $J^{(D)}$.

However, the Equation 2.23 is mainly used for theoretical analysis. In practical applications, the gradient of the generator could quickly vanish when the discriminator rejects the fake samples. One common improvement modifies the generator loss function as follows:

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{z \sim p_z(z)}[\log D(G(z; \theta_G); \theta_D)] \tag{2.24}$$

A further improvement is to do maximum likelihood learning with GANs in practice as follows:

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{z \sim p_z(z)}[e^{\sigma^{-1}(D(G(z; \theta_G); \theta_D))}] \tag{2.25}$$

where σ denotes the logistic sigmoid function.

State-of-the-art GAN research [7, 47, 64, 18] mainly focuses on two points: 1) how to produce the sample as realistic as possible and 2) how to stabilize the GAN training process and also avoid the mode collapse issue. Mode collapse represents the generator collapses, which produce limited modes and sample variety. We will discuss state-of-the-art approaches that overcome these drawbacks in Section 4.

Compared with results from VAE, which often produce blurry samples, GAN generates more sharp and realistic samples. Many researchers believe that the effect comes from the VAE minimizing the asymmetric divergence (ex: KL divergence) between the data and the model, whereas the GAN minimizes the Jensen-Shannon (JS) divergence, which is symmetric. Some clues [44] show that even using different divergence, the model still encounters the sample quality, training stability, and mode collapse problems. Therefore, specifically why GAN produces more realistic samples is still not clear to the current research community.

2.3 Reinforcement Learning

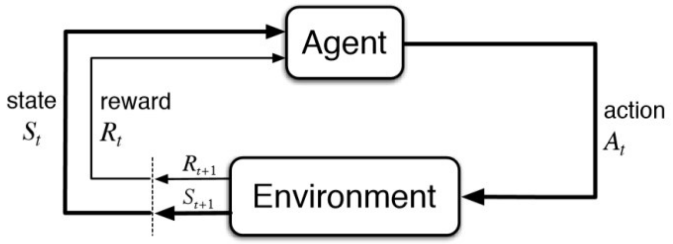


Figure 2.7: RL environment settings

Reinforcement learning (RL) is primarily used to solve the sequential decision making problem, which is also a typical control problem. Compared with supervised learning or unsupervised learning that collects large amounts of data for training, RL (shown in Figure 2.7) requires an interactive query in the environment to learn the optimal policy, which is maximizing the accumulated rewards. The learning process was originally inspired from neuro-psychology.

Unlike conventional control methodology which uses domain knowledge to build a mathematical model of the system dynamics, RL is a data-driven approach to learn the optimal policy. In the conventional optimal control, we have:

$$\begin{aligned} &\min \mathbb{E}_e[\Sigma_{t=1}^T C_t(x_t, u_t)] \\ &\text{s.t. } x_{t+1} = f_t(x_t, u_t, e_t), \\ &u_t = \pi_t(\tau_t) \end{aligned} \tag{2.26}$$

where $C_t(x_t, u_t)$ is the user defined cost, f_t denotes the state transition function, $\tau_t = (u_1, ..., u_{t-1}, x_0, ..., x_t)$ represents an observed trajectory, e_t is the noise process, and $\pi_t(\tau_t)$ is the optimization decision variable. Typically, the state transition is unknown and the system identification is employed to learn continuous system dynamics.

In RL settings, the system dynamics lie upon discrete configurations: x_t represents state at time t,

u_t is the action at each step, and $C_t(x_t, u_t)$ denotes the reward function. Unlike traditional control methods, which use a model driven approach, RL is a data driven method that learns optimal policy from the data. RL follows the Markov Decision Process (MDP) or Partially Observable Markov Decision Process (POMDP). In this research, we mainly focus on problems that are modeled by MDP, therefore we omit discussion of POMDP.

2.3.1 Markov Decision Process

We consider an environment defined as an MDP $< S, A, \mathrm{P}_{s',s,a}, \mathrm{R}_{s,a}, \gamma >$ that contains a set of states $s \in S$, a set of actions $a \in A$, a reward function $\mathrm{R}_{s,a}$, and a series of transition probabilities $\mathrm{P}_{s',s,a}$, the probability of moving from the current state s to the next state s' given an action a. The goal of an MDP is to find a policy that maximizes the expected return (accumulated rewards):

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.27}$$

where r_{t+k} is the immediate reward at the $(t+k)$th time stamp and $\gamma \in (0,1]$ is the discount rate. A policy $\pi_{a,s}$ is the probability of taking action a when in state s. The value of state s under a policy π, $\mathbf{v}_s^{\pi}$, is the expected return when starting in s and following π; that is, $\mathbf{v}_s^{\pi} = \mathbb{E}[R_t | S_t = s]$. The value of taking action a in state s under a policy π, $\mathbf{q}^{\pi}{}_s^{(a)}$, is the expected return when starting in s, taking the action a and following π; that is, $\mathbf{q}^{\pi}{}_s^{(a)} = \mathbb{E}[R_t | S_t = s, A_t = a]$. A policy is defined as:

$$\pi(s) : S \to A \tag{2.28}$$

The value-function is used to estimate how good it is at state s_t. If the agent uses a given policy π, the corresponding value-function can be written as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[\sum_{k=0}^{T-1} \gamma^k r_k] \tag{2.29}$$

Among all possible value-functions, there exists an optimal value function that is higher than others for all states:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \tag{2.30}$$

and the optimal policy is:

$$\pi^*(s) = \arg\max_{\pi} V^{\pi}(s) \tag{2.31}$$

RL also introduces an additional value function (action-value function) to evaluate at specific state s, the value of agent applying specific action. The action-value function, also called Q function, is a function of the state-action paired with real values, where mathematically Q: $S \times A \rightarrow \mathbb{R}$. The optimal Q function, which is defined as $Q^*(s,a)$, has the following relationship with optimal value function $V^*(s)$:

$$V^*(s) = \max_{a} Q^*(s,a) \tag{2.32}$$

and is similar to the value function. If we know the optimal Q function, the optimal policy chooses the action that gives maximum $Q^*(s,a)$:

$$\pi^*(s) = \arg\max_{a} Q^{\pi}(s,a) \tag{2.33}$$

2.3.2 Dynamic Programming

The typical way to update the Q function and V function is based on the Bellman Equation, which uses dynamic programming to recursively update the value function.

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \tag{2.34}$$

where:

$$V^*(s) = \max_{a} Q^*(s,a) \tag{2.35}$$

In summary, for the value-based RL, there is always at least one policy that is better than or equal to all other policies, called an optimal policy π^*; that is, the optimal policy is $\pi^* = \arg\max_{\pi} \mathbf{v}_s^{\pi}$, the optimal state-value function is $\mathbf{v}_s^* = \max_{\pi} \mathbf{v}_s^{\pi}$, and the optimal action-value function is $\mathbf{q^*}_s^{(a)} = \max_{\pi} \mathbf{q^{\pi}}_s^{(a)}$. To obtain π^* and $\mathbf{v}^*$, we usually consider solving the Equations 2.34 and 2.35.

Policy iteration and Value iteration are two popular dynamic programming algorithms to solve the Bellman equation in the discrete state space.

Value iteration The algorithm initializes $V(s)$ to random values, then we iteratively compute $\mathbf{v}_s \leftarrow \max_a \sum_{s'} \mathrm{P}_{s',s,a} \left(\mathrm{R}_{s,a} + \gamma \mathbf{v}_{s'}\right)$ until the result converges. The algorithm is guaranteed to converge to the

Algorithm 1 Value Iteration

```
1: Initialize $V(s)$ with random value
2: repeat
3:     for all $s \in S$ do
4:         for all $a \in A$ do
5:             $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V(s')$
6:         end for
7:         $V(s) \leftarrow \max_a Q(s,a)$
8:     end for
9: until $V(s)$ converge
```

optimal values [125] and the algorithm is given in Algorithm 1:

Policy iteration The goal of the agent is to obtain the optimal policy. Therefore, instead of repeatedly improving the value-function, policy iteration updates policy in each epoch and the value is re-computed upon the new policy. The policy iteration is also guaranteed to converge to the optimal policy and normally takes less iteration than the value iteration algorithm. The detailed algorithm in shown in Algorithm 2

Algorithm 2 Policy Iteration

```
1: Initialize policy $\pi'$ randomly
2: repeat
3:     $\pi \leftarrow \pi'$
4:     Improve values:
5:         $V^{\pi}(s) \leftarrow R(s,\pi(s)) + \gamma \sum_{s' \in S} p(s'|s,\pi(s))V^{\pi}(s')$
6:     Improve policy:
7:         $\pi'(s) \leftarrow \arg\max_a (R(s,a) + \gamma \sum_{s' \in S} p(s'|s,a)V^{\pi}(s'))$
8: until $\pi = \pi'$
```

The challenge to use policy iteration and value iteration is that both algorithms assume the agent knows the environment dynamics ($p(s'|s,a)$) beforehand. However, this is often not true in most applications. One possible approach to this issue is to use system identification, which is similar to traditional control methods. Another alternative is to use model-free RL techniques. Model-free RL can be categorized as three types: value-based, policy-based, and actor-critic.

2.3.3 Value-based Model-free RL

Q-learning is to approximate the state-action pairs Q-function from the samples that the agent observes during the interaction with environment. Since Equation 2.34 still requires the model dynamics $p(s'|s,a)$, Q-learning uses the Bellman equation to learn directly the Q values from the data. The update rule can be written as:

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a') \tag{2.36}$$

Therefore the policy becomes $a = \arg\max_a Q(s,a)$. Nevertheless, the agent should not greedily choose the optimal policy at the initial stage, but rather explore the environment. Hence, the ϵ-greedy algorithm balances the exploration and exploitation during training. The trade-off between exploration and exploitation belongs to the study of the bandit problem. We do not discuss the more advanced methods such as the Thompson Sampling and Upper Confidence Bound (UCB) since they are out of scope of this research.

Deep Q Network (DQN) [88] uses a neural network to approximate the Q function. However, using a nonlinear approximator for Q-learning does not guarantee convergence. To stabilize the training, DQN introduces the experience replay buffer and target network. The loss function is defined as:

$$\mathcal{L}(\theta) = \frac{1}{2}\sum_i \|R + \gamma \max_{a'} Q_\theta(s_i', a_i') - Q_\theta(s_i, a_i)\|_2^2 \tag{2.37}$$

where each $< s_i, a_i, s_i', r_i >$ are sampled from the experience replay buffer. The experience replay buffer collects history policy of the neural network. In other words, the current policy is updated by old policy or another policy. Then, this is a typical off-policy learning. The experience replay buffer provides better sample efficiency, that is, the agent repeatedly samples from historical data to correct the current Q function. Also, the experience replay buffer could be flexible and filled with other policies, for example, the optimal trajectory or high quality policy.

DQN also introduces the target network. In Equation 2.37, the term $R + \gamma \max_a' Q_\theta(s_i', a_i')$ shifts in every iteration, which falls into the feedback loop between the target and estimated Q-value. In other words, the value estimations can easily spiral out of control. The target network employs the second Q network that fixes the network's weights and only periodically or slowly updates it from the primary Q-network. Therefore, the new loss function can be written as:

$$\mathcal{L}(\theta) = \frac{1}{2}\sum_i \|R + \gamma \max_{a'} Q_{\hat{\theta}}(s_i', a_i') - Q_\theta(s_i, a_i)\|_2^2 \tag{2.38}$$

where $Q_{\hat{\theta}}$ denotes the target network. More recent proposed extensions are discussed in Chapter 4.

2.3.4 Policy Optimization

In the policy gradient approach, instead of using the value function to estimate expected returns from some policy, the agent directly uses the function approximator to map input state to action. Recall that the goal of RL is $\max \mathbb{E}_{\pi}[R]$, where $R = r_0 + r_1 + ... + r_{T-1}$, the agent must solve the gradient of the objectives. Consider a general expectation $\mathbb{E}_{x \sim p(x|\theta)}[f(x)]$, the gradient is:

$$\begin{aligned}
\nabla_{\theta}\mathcal{L}(\theta) &= \nabla_{\theta}\mathbb{E}_{x \sim p(x|\theta)}[f(x)] \\
&= \nabla_{\theta}\int p(x|\theta)f(x)dx \\
&= \int \nabla_{\theta}p(x|\theta)f(x)dx \\
&= \int p(x|\theta)\frac{\nabla_{\theta}p(x|\theta)}{p(x|\theta)}f(x)dx \\
&= \int p(x|\theta)\nabla_{\theta}\log p(x|\theta)f(x)dx \\
&= \mathbb{E}_{x \sim p(x|\theta)}[\nabla_{\theta}\log p(x|\theta)f(x)]
\end{aligned} \tag{2.39}$$

we replace the $f(x)$ to be the accumulated rewards R, and $p(x|\theta)$ to be the policy network $\pi(a_t|s_t,\theta)$. Then, the policy gradient can be written as:

$$\nabla_{\theta}\mathbb{E}_{\pi}[R] = \mathbb{E}_{\pi}[\sum_{t=0}^{T-1}\nabla_{\theta}\log\pi(a_t|s_t,\theta)\sum_{t'=t}^{T-1}\gamma^{t'-t}r_{t'}] \tag{2.40}$$

Equation 2.40 is the famous REINFORCE algorithm [126]. However, the vanilla REINFORCE is a high variance and unbiased gradient estimator, which could take many long iterations for the algorithm to converge. One typical approach is to add the baseline term $b(s) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} ... + \gamma^{T-1-t}r_{T-1}]$ as:

$$\nabla_{\theta}\mathbb{E}_{\pi}[R] = \mathbb{E}_{\pi}[\sum_{t=0}^{T-1}\nabla_{\theta}\log\pi(a_t|s_t,\theta)(\sum_{t'=t}^{T-1}\gamma^{t'-t}r_{t'} - b(s_t))] \tag{2.41}$$

since expectation of $b(s)$ is also unbiased, that is $\mathbb{E}_{\pi}[\sum_{t=0}^{T-1}\nabla_{\theta}\log\pi(a_t|s_t,\theta)b(s)] = 0$, the conclusion is easy to prove and thereby we omit it here. For simplicity, the term $\hat{A}_t = (R_t - b(s_t))$ is also called the advantage function, where $R_t = \sum_{t'=t}^{T-1}\gamma^{t'-t}r_{t'}$ and Equation 2.41 can be written as:

$$\nabla_{\theta}\mathbb{E}_{\pi}[R] = \mathbb{E}_{\pi}[\sum_{t=0}^{T-1}\nabla_{\theta}\log\pi(a_t|s_t,\theta)\hat{A}_t] \tag{2.42}$$

Equation 2.42 is the standard policy gradient optimization objective function in current RL. We will discuss recently proposed and more advanced algorithms in Chapter 4

2.3.5 Actor-Critic Model-free RL

Actor-Critic combines the idea of value-based and policy gradient: instead of using high variance accumulated return in each episode, Actor-Critic uses the value function to represent the advantage function, which can be written as:

$$\nabla_\theta \mathbb{E}_\pi[R] = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) Q(s,a)] \tag{2.43}$$

The $Q(s,a)$ can be updated via MSE in DQN (Equation 2.38). However, Equation 2.43 offers low variance biased gradient estimation. To reduce the bias, Actor-Critic adds an extra baseline function, which commonly chooses the value function $V(s)$. Therefore, the advantage function becomes $A(s,a) = Q(s,a) - V(s)$, training in a temporal difference (TD) way. Nevertheless, the new advantage function still requires two additional networks to represent Q value and V value. On the other hand, the true value function $V(s)$ is an unbiased estimate of the advantage function, and we define TD error as: δ^π:

$$\delta^\pi = r + \gamma V^\pi(s') - V^\pi(s) \tag{2.44}$$

Since $\mathbb{E}_\pi[\delta^\pi|s,a] = Q^\pi(s,a) - V^\pi(s) = A^\pi(s,a)$, the standard Actor-Critic algorithm updates the policy gradient by:

$$\nabla_\theta \mathbb{E}_\pi[R] = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) \delta^\pi] \tag{2.45}$$

Similar to Q learning, the value network $V(s)$ is also trained with regression based on MSE loss. Assuming the value network is parameterized by ψ, the loss function for value network is:

$$\mathcal{L}(\psi) = \frac{1}{2}||r + \gamma V_{\hat{\psi}}(s') - V_\psi(s)||_2^2 \tag{2.46}$$

Equation 2.45 and Equation 2.46 together defined the Actor-Critic algorithm. Since π_θ and $\mathcal{V}_\psi$ are fully differentiable neural networks, the agent could be easily trained with the backpropagation algorithm.

2.4 Graph Convolutional Neural Network

A graph embedding is a collection of feature vectors associated with nodes in a graph; each feature vector describes the overall role of the corresponding node. Through a graph embedding, we are able to visualize a graph in a 2D/3D space and transform problems from a non-Euclidean space to a Euclidean space, where numerous machine learning and data mining tools can be applied. The applications of graph embedding include graph visualization [54], graph clustering [141], node classification [111, 151], link prediction [79, 8], recommendation [148], anomaly detection [4], and many others.

Graph Convolutional Neural Networks (GCNNs) are a new recently proposed approach that introduces neural networks into graph embedding problems. The goal of GCNN is to learn a function of features $G = (\mathcal{V}, \mathcal{E})$ that takes input from node features x_i and the adjacency matrix A, producing new synthesized feature representations H for each node. The final graph representation H could be used for any tasks such as clustering, community detection, and others.

Unlike images, which can be viewed as a lattice graph, a typical graph such as social network might have a varying number of neighbors. Hence, the graph filter is difficult to share among neighbors. [68] is one typical GCNN model that uses a layer-wise propagation rule, which is used in Section 3.1.2. For each layer:

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{2.47}$$

where $\hat{A} = A + I$, I denotes the identity matrix, $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$, σ represents the neural network activation function, and $W^{(l)}$ is the parameter in l-th layer. Note that the entire propagation rule is also fully differentiable, which can be trained by backpropagation.

2.5 Summary

In this chapter, we have reviewed basic Machine Learning techniques relevant to this research from a general optimization view. We also discussed the deep generative model, specifically VAE and GAN that learns the data distribution by employing a neural network. Additionally, we have described the concept of RL and listed several key RL algorithms including model-free RL and planning. Finally, we briefly reviewed graph embedding and the GCNN concept, a typical GCNN approach is also discussed. The proposed research in Section 3.1 requires backgrounds on planning, RL and GCNN; the prerequisit of Section 3.2 is Machine Learning basics and graph embedding; Section 3.3 uses the background knowledge of the deep generative model.

Chapter 3

Research Design and Methods

In this chapter, we present our research design and methods. First we introduce the Generalized Value Iteration Network (GVIN) [92], which is a differentiable planning module on graph; GVIN relies on graph embedding as vertices features. To learn the low dimensional dense vertices features, we propose WarpMap [23], which is an unified framework for graph embedding. Finally, to resolve the domain transfer in reinforcement learning environment, we demonstrate feature alignment for visual based state input that leverages the agent transfer knowledge from the simulator environment to the real world. The corresponding results are presented in Chapter 5.

3.1 Generalized Value Iteration Network

To create GVIN, we generalize VIN in two aspects. First, to work for irregular graphs, we propose a graph convolution operator that generalizes the original 2D convolution operator. With the new graph convolution operator, the proposed network captures the basic concepts in spatial graphs, such as direction, distance and edge weight. It also is able to transfer knowledge learned from one graph to others. Second, to improve reinforcement learning on irregular graphs, we propose a reinforcement learning algorithm, episodic Q-learning, which stabilizes the training for VIN and GVIN. The original VIN is trained through either imitation learning, which requires a large number of ground-truth labels, or reinforcement learning, whose performance is relatively poor. With the proposed episodic Q-learning, the new network performs significantly better than VIN in the reinforcement learning mode. Since the proposed network generalizes the original VIN model, we call it the generalized value iteration network (GVIN).

GVIN takes a general graph with a starting node and a goal node as inputs and outputs the designed plan. The goal of GVIN is to learn an underlying MDP that summarizes the optimal planning policy applied for arbitrary graphs, which requires GVIN to capture general knowledge about planning, that is structure and transition invariant. A key component of an MDP is the transition matrix, which is needed to solve the Bellman equation. To train a general transition matrix that works for arbitrary graphs, similar to the VIN [128], we treat it as a graph convolution operator and parameterize it by using graph-based kernel functions, each of which represents a unique action pattern. We train the parameters in GVIN by using episodic Q-learning, which makes reinforcement learning on irregular graphs practical.

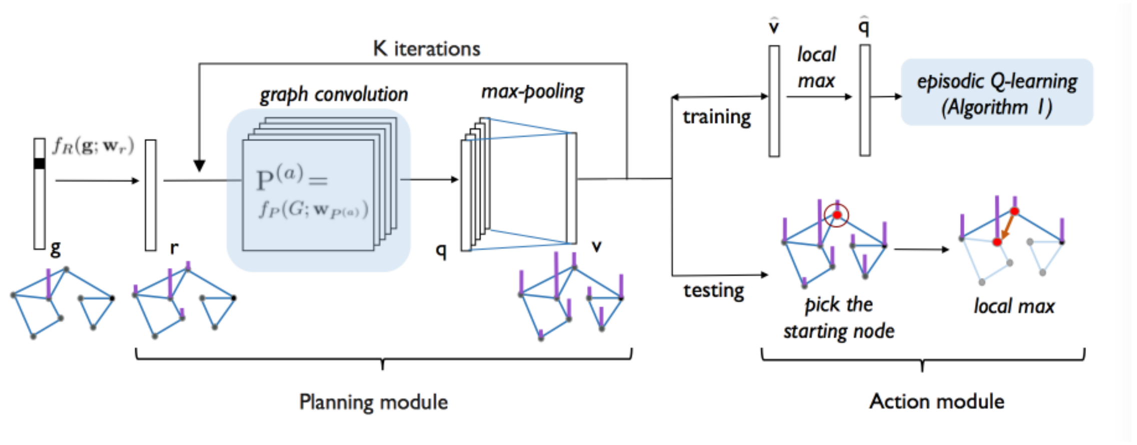


Figure 3.1: Architecture of GVIN. The left module emulates value iteration and obtains the state values; the right module is responsible for selecting an action based on an ϵ-greedy policy (for training) or a greed policy (for testing). We emphasize our contributions, including graph convolution operator and episodic Q-learning, in the blue blocks.

3.1.1 GVIN Framework

The input of GVIN is a graph with a starting node and a goal node. In the training phase, GVIN trains the parameters by trial-and-error on various graphs. During the testing phase, GVIN plans the optimal path based on the trained parameters. The framework includes the planning module (left) and the action module (right), shown in Figure 3.1. The planning module emulates value iteration by iteratively operating the graph convolution and max-pooling. The action module takes the greedy action according to the value function.

Mathematically, we consider a directed, weighted spatial graph $G = (\mathcal{V}, \mathrm{X}, \mathcal{E}, \mathrm{A})$, where $\mathcal{V} = \{v_1, ..., v_N\}$ is the node set, $\mathrm{X} \in \mathcal{R}^{N \times 2}$ are the node embeddings with the ith row $\mathrm{X}_i \in \mathcal{R}^2$ being the embedding of the ith node (here we consider 2D spatial graphs, but the method is generalizable), $\mathcal{E} = \{e_1, ..., e_M\}$ is the edge set, and $\mathrm{A} \in \mathbb{R}^{N \times N}$ is the adjacency matrix, with the (i,j)th element $\mathrm{A}_{i,j}$ representing the edge weight between the ith and jth nodes. We consider a graph signal as a mapping from the nodes to real values. We use a graph signal $\mathbf{g} \in \{0,1\}^N$ to encode the goal node, where $\mathbf{g}$ is one-sparse and only activates the goal node. Let $\mathbf{r} \in \mathcal{R}^N$, $\mathbf{v} \in \mathcal{R}^N$, and $\mathbf{q} \in \mathcal{R}^N$ be the reward graph signal, the state-value graph signal, and the action-value graph signal, respectively. We represent the entire process in a matrix-vector form as follows,

$$\mathbf{r} = f_R(\mathbf{g}; \mathbf{w}_{\mathbf{r}}), \tag{3.1}$$

$$\mathrm{P}^{(a)} = f_P(G; \mathbf{w}_{\mathrm{P}^{(a)}}), \tag{3.2}$$

$$\mathbf{q}_{n+1}^{(a)} = \mathrm{P}^{(a)} \left(\mathbf{r} + \gamma \mathbf{v}_n \right), \tag{3.3}$$

$$\mathbf{v}_{n+1} = \max_a \mathbf{q}_{n+1}^{(a)}. \tag{3.4}$$

In the feature extraction step, Equation (3.1), $\mathbf{g}$ is encoded to become the robust reward $\mathbf{r}$ via the feature-extract function $f_R(\cdot)$, which is a convolutional neural network in the case of regular graphs, but is the identity function when operating on irregular graphs. In Equation (3.2), where $\mathrm{P}^{(a)}$ is the graph convolution operator in the ath channel, a set of graph convolution operators is trained based on the graph G, which is further described in Section 3.1.2; in (3.3) and (3.4), the value iteration is emulated by using graph convolution to obtain the action-value graph signal $\mathbf{q}^{(a)}$ in the ath channel and max-pooling to obtain the state-value graph signal $\mathbf{v}$. The training parameters $\mathbf{w}_{\mathbf{r}}$ and $\mathbf{w}_{\mathrm{P}^{(a)}}$ are used to parameterize $\mathbf{r}$ and $\mathrm{P}^{(a)}$, respectively. As shown in Figure 3.1, we repeat the graph convolution operation Equation (3.3) and max-pooling Equation (3.4) for K iterations to obtain the final state-value graph signal $\widehat{\mathbf{v}}$. When G is a 2D lattice, the planning module of GVIN degenerates to VIN.

In the training phase, we feed the final state-value graph signal $\widehat{\mathbf{v}}$ to the action module. The original VIN extracts the action values from Equation (3.3) and trains the final action probabilities for eight directions. However, this is problematic for irregular graphs, as the number of actions (neighbors) at each node varies. To solve this, we consider converting $\widehat{\mathbf{v}}$ to a pseudo action-value graph signal, $\widehat{\mathbf{q}} \in \mathcal{R}^N$, whose sth element is $\widehat{\mathbf{q}}_s = \max_{s' \in \mathrm{Nei}(s)} \widehat{\mathbf{v}}_{s'}$, representing the action value moving from s to one of its neighbors. The advantages of this approach come from the following three aspects: (1) the final state value of each node is obtained by using the maximum action values across all the channels, which is robust to small variations; (2) the pseudo action-value graph signal considers a unique action for each node and does not depend on the number of actions; that is, at each node, the agent queries the state values of its neighbors and always moves to the one with the highest value; and (3) the pseudo action-value graph signal considers local graph structure, because the next state is always chosen from one of the neighbors of the current state.

The pseudo action-value graph signal is used through episodic Q-learning, which learns from trial-and-error experience and backpropagates to update all of the training parameters. In episodic Q-learning, each episode is obtained as follows: for each given starting node s_0, the agent will move sequentially from s_t to s_{t+1} by the ϵ-greedy strategy; that is, with probability $(1-\epsilon)$, $s_{t+1} = \arg\max_{s' \in \mathrm{Nei}(s_t)} \widehat{\mathbf{v}}_{s'}$ and with probability ϵ, s_{t+1} is randomly selected from one of the neighbors of s_t. An episode terminates when s_{t+1} is the goal state or the maximum step threshold is reached. For each episode, we consider the loss function as, $L(\mathbf{w}) = \sum_{t=1}^{T} \left(R_t - \widehat{\mathbf{q}}_{s_t}\right)^2$, where $\widehat{\mathbf{q}}_{s_t}$ is a function of the training parameters $\mathbf{w} = [\mathbf{w}_{\mathbf{r}}, \mathbf{w}_{\mathrm{P}^{(a)}}]$ in GVIN, T is the episode length and R_t is the expected return at time stamp t, defined as $R_t = (r_{t+1} + \gamma R_{t+1})$, where γ is the discount factor, and r_t is the immediate return at time stamp t. Additional details of the algorithm will be discussed in Section 3.1.3. In the testing phase, we obtain the action by greedily selecting the maximal state value as shown in Equation (3.5)

$$s_{t+1} = \arg \max_{s' \in \mathrm{Nei}(s_t)} \widehat{\mathbf{v}}_{s'} \tag{3.5}$$

3.1.2 Graph Convolution

The conventional CNN takes an image as input, which is a 2D lattice graph. Each node is a pixel and has the same local structure, sitting on a grid and connecting to its eight neighbors. In this case, the convolution operator is easy to obtain. In irregular graphs, however, nodes form diverse local structures, making it challenging to obtain a structured and translation invariant operator that transfers knowledge from one graph to another. The fundamental problem here is to find a convolution operator that works for arbitrary local structures. We solve this through learning a 2D spatial kernel function that provides a transition probability distribution in the 2D space and we evaluate the weight of each edge to obtain a graph convolution operator.

The 2D spatial kernel function assigns a value to each position in the 2D space, which reflects the possibility to transit to the corresponding position. Mathematically, the transition probability from a starting position $\mathbf{x} \in \mathcal{R}^2$ to another position $\mathbf{y} \in \mathcal{R}^2$ is $K(\mathbf{x}, \mathbf{y})$, where $K(\cdot, \cdot)$ is a 2D spatial kernel function, which is specified in Definition 1 below.

Definition 1. A 2D spatial kernel function $K(\cdot, \cdot)$ is shift invariant when it satisfies $K(\mathbf{x}, \mathbf{y}) = K(\mathbf{x}+\mathbf{t}, \mathbf{y}+\mathbf{t})$, for all $\mathbf{x}, \mathbf{y}, \mathbf{t} \in \mathcal{R}^2$.

The shift invariance requires that the transition probability depends on the relative position, which is the key for transfer learning. In other words, no matter where the starting position is, the transition probability distribution is invariant. Based on a shift-invariant 2D spatial kernel function and the graph adjacency matrix, we obtain the graph convolution operator $\mathrm{P} = f_P(G; \mathbf{w}_\mathrm{P}) \in \mathcal{R}^{N \times N}$, where each element is defined as $\mathrm{P}_{i,j} = \mathrm{A}_{i,j} \cdot K_{\mathbf{w}_\mathrm{P}}(\mathrm{X}_i, \mathrm{X}_j)$, where the kernel function $K_{\mathbf{w}_\mathrm{P}}(\cdot,\cdot)$ is parameterized by $\mathbf{w}_\mathrm{P}$, and $\mathrm{X}_i, \mathrm{X}_j \in \mathcal{R}^2$ are the embeddings of the ith and jth node. The graph convolution operator follows from (1) graph connectivity and (2) 2D spatial kernel function. With the shift-invariant property, the 2D spatial kernel function leads to the same local transition distribution at each node such that the graph adjacency matrix works as a modulator to select activations in the graph convolution operator. When there is no edge between i and j, we have $\mathrm{A}_{i,j} = 0$ and $\mathrm{P}_{i,j} = 0$; when there is an edge between i and j, $\mathrm{P}_{i,j}$ is high when $K_{\mathbf{w}_\mathrm{P}}(\mathrm{X}_i, \mathrm{X}_j)$ is high. In other words, when the transition probability from the ith node to the jth node is higher, the edge weight $\mathrm{P}_{i,j}$ is high and the influence from the ith node to the jth node is larger during the graph convolution. Note that P is a sparse matrix and its sparsity pattern is the same with its corresponding adjacency matrix, which ensures cheap computation.

As shown in Equation (3.2), the graph convolution is a matrix-vector multiplication between the graph convolution operator P and the graph signal $\mathbf{r} + \gamma \mathbf{v}_n$ (see Figure 3.2). Note that when we work with a lattice graph and an appropriate kernel function, this graph convolution operator P is simply a matrix representation of the conventional convolution [75]; in other words, VIN is a special case of GVIN when the underlying graph is a 2D lattice.

We consider three types of shift-invariant 2D spatial kernel functions: the directional kernel, the spatial kernel, and the embedding kernel.

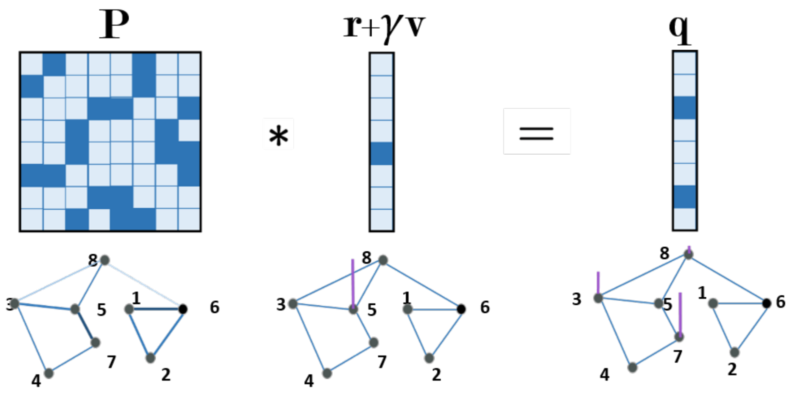


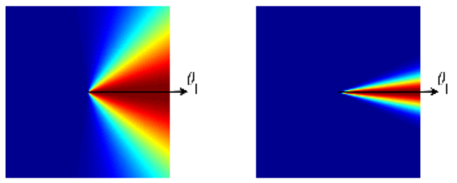
Figure 3.2: Matrix-vector multiplication as graph convolution. Through a graph convolution operator P, $\mathbf{r} + \gamma \mathbf{v}$ diffuses over the graph to obtain the action-value graph signal $\mathbf{q}$.

Directional Kernel. We first consider direction. When we face several roads at an intersection, it is straightforward to pick the one whose direction points to the goal. We aim to use the directional kernel to capture edge direction and parameterize the graph convolution operation.

The (i,j)th element in the graph convolution operator models the probability of following the edge from i to j ; that is,

$$\begin{aligned} \mathrm{P}_{i,j} &= \mathrm{A}_{i,j} \cdot \sum_{\ell=1}^{L} w_\ell K_{\mathrm{d}}^{(t,\theta_\ell)}(\theta_{ij}), \\ \text{where } K_{\mathrm{d}}^{(t,\theta_\ell)}(\theta) &= \left(\frac{1+\cos(\theta-\theta_\ell)}{2}\right)^t, \end{aligned} \tag{3.6}$$

where w_ℓ is the kernel coefficient, θ_{ij} is the direction of the edge connecting the ith and the$j$th nodes, which can be computed through the node embeddings $\mathrm{X}_i \in \mathcal{R}^2$ and $\mathrm{X}_j \in \mathcal{R}^2$, and $K_{\mathrm{d}}^{(t,\theta_\ell)}(\theta)$ is the directional kernel with order t and reference direction θ_ℓ, reflecting the center of the activation. The hyperparameters include the number of directional kernels L and the order t, reflecting the directional resolution (a larger t indicates more focus in one direction, see Figure 3.3). The kernel coefficient w_ℓ and the reference direction θ_ℓ are the training parameters, which is $\mathbf{w}_{\mathrm{P}}$ in Equation (3.2). Note that the graph convolution operator $\mathrm{P} \in \mathcal{R}^{N\times N}$ is a sparse matrix and its sparsity pattern matches the input adjacency matrix, which ensures that the computation is cheap.



(a) $t = 5$ (b) $t = 100$

Figure 3.3: The directional kernel function activates the areas around the reference direction θ_ℓ in the 2D spatial domain. The activated area is more concentrated when t increases.

The intuition behind Equation (3.6) is that each graph convolution operator represents a unique direction pattern. An edge is a 2D vector sampled from the 2D spatial plane. When the direction of the edge connecting i and j matches one or some of the L reference directions, we have a higher probability to follow the edge from i to j. In GVIN, we consider several channels. In each channel, we obtain an action value for each node, which represents the matching coefficient of a direction pattern. The max-pooling operation then selects the best available matching direction pattern for each node.

Spatial Kernel. We next consider both direction and distance. When all of the roads at the current intersection are in the opposite of the goal, it is straightforward to try the shortest one first. We thus include the edge length into the consideration. The (i, j)th element in the graph convolution operator is then,

$$\begin{aligned} \mathrm{P}_{i,j} &= \mathrm{A}_{i,j} \cdot \sum_{\ell=1}^{L} w_\ell K_{\mathrm{s}}^{(d_\ell, t, \theta_\ell)}\left(d_{ij}, \theta_{ij}\right), \\ \text{where } K_{\mathrm{s}}^{(d_\ell, t, \theta_\ell)}(d, \theta) &= \mathrm{I}_{|d-d_\ell| \le \epsilon} \left(\frac{1 + \cos(\theta - \theta_\ell)}{2} \right)^t, \end{aligned} \tag{3.7}$$

where d_{ij} is the distance between the ith and the jth nodes, which can be computed through the node embeddings $\mathrm{X}_i \in \mathcal{R}^2$ and $\mathrm{X}_j \in \mathcal{R}^2$, $K_{\mathrm{s}}^{(d_\ell, t, \theta_\ell)}(d, \theta)$ is the spatial kernel with reference distance d_ℓ and reference direction θ_ℓ, and the indicator function $\mathrm{I}_{|d-d_\ell| \le \epsilon} = 1$ when $|d - d_\ell| \le \epsilon$ and 0, otherwise. The hyperparameters include the number of directional kernels L, the order t, the reference distance d_ℓ, and the distance threshold ϵ. The kernel coefficient w_ℓ and the reference direction θ_ℓ are training parameters, which is $\mathbf{w}_{\mathrm{P}}$ in Equation (3.2).

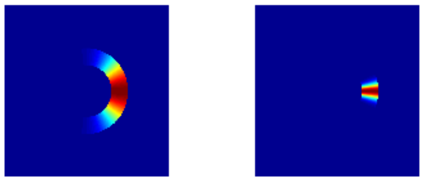

(a) $t = 5$. (b) $t = 100$.

Figure 3.4: The spatial kernel function activates the areas around the reference direction θ_ℓ and reference distance d_ℓ in the 2D spatial domain.

Compared to the directional kernel, the spatial kernel adds the distance as extra dimension of information (see Figure 3.4); in other words, the directional kernel is a special case of the spatial kernel when we ignore the distance. Each spatial kernel activates a localized area in the direction-distance plane. With the spatial kernel, the graph convolution operator in Equation (3.7) represents a unique direction-distance pattern; that is, if the direction/distance of the edge connecting i and j matches one or some of the L reference directions and distances, we have a higher probability to follow the edge from i to j.

Embedding-based Kernel. In the directional and spatial kernels, we manually design the kernel and provides hints to GVIN to learn useful direction-distance patterns. Now with the embedding-based kernel we directly feed the node embeddings and allow GVIN to automatically learn implicit hidden factors for general planning. The (i, j)th element in the graph convolution operator is then,

$$\mathrm{P}_{i,j} = \frac{(\mathrm{I}_{i=j} + \mathrm{A}_{i,j})}{\sqrt{\sum_k (1 + \mathrm{A}_{k,j}) \sum_k (1 + \mathrm{A}_{i,k})}} \cdot K_{\mathrm{emb}}(\mathrm{X}_i, \mathrm{X}_j), \tag{3.8}$$

where the indicator function $\mathrm{I}_{i=j} = 1$ when $i = j$, and 0, otherwise, and the embedding-based kernel function is $K_{\mathrm{emb}}(\mathrm{X}_i, \mathrm{X}_j) = \mathrm{mnnet}([\mathrm{A}_{ij}, \mathrm{X}_i - \mathrm{X}_j])$, where mnnet$(\cdot)$ is a standard multi-layer neural network. The training parameters $\mathbf{w}_{\mathrm{P}}$ in Equation (3.2) are the weights in the multi-layer neural network. Note that the graph convolution operator $\mathrm{P} \in \mathcal{R}^{N \times N}$ is still a sparse matrix and its sparsity pattern matches that of the input adjacency matrix plus the identity matrix.

In the directional kernel and spatial kernel, we implicitly discretize the space based on the reference direction and distance; that is, for each input pair of given direction and distance, the kernel function outputs the response based on its closed reference direction and distance. In the embedding-based kernel, we do not set the reference direction and distance to discretize the space; instead, we use the multi-layer neural network to directly regress from an arbitrary edge (with edge weight and embedding representation) to a response value. The embedding-based kernel is thus more flexible than the directional kernel and spatial kernel and may learn hidden factors.

Theorem 1. The proposed three kernel functions, the directional kernel, the spatial kernel and the embedding-based kernel, are shift invariant.

The proof follows from the fact that those kernels use only the direction, distance, and the difference between two node embeddings, which only depends on the relative position.

3.1.3 Training via Reinforcement Learning

We train GVIN through episodic Q-learning, a modified version of n-step Q-learning. The difference between episodic Q-learning and the n-step Q-learning is that the n-step Q-learning has a fixed episode duration and updates the training weights after n steps. In episodic Q-learning, each episodic terminates when the agent reaches the goal or the maximum step threshold is reached and we update the trainable weights after the entire episode. During experiments, we found that for both regular and irregular graphs, the policy planned by the original Q-learning keeps changing and does not converge due to the frequent updates. Similar to the Monte Carlo algorithms [124], episodic Q-learning first selects actions by using its exploration policy until the goal is reached. Afterwards, we accumulate the gradients during the entire episode and then update the trainable weights, allowing the agent to use a stable plan to complete an entire episode. This simple change greatly improves the performance (see Section 5.1.1). The pseudocode for the algorithm is presented in Algorithm 3 (Section 3.1.4).

Recall original Deep Q-learning, the agent follows MDP introduced in Chapter 2, and the goal is to maximize the total expected discounted reward:

$$G_t = \mathbb{E}_{s_t,a_t,r_t}[\sum_{t=0}^{T} \gamma^t r_t]. \tag{3.9}$$

Where the expectation is over policy $\pi(a_t|s_t)$ with unknown environment dynamics $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$, discount factor γ, and rewards function $r_t \sim r(s_t, a_t)$. To effectively learn a policy, DQN uses the history trajectory data to estimate Q-function $Q(s, a)$, where it follows Bellman equations which is sufficient to obtain optimal Q^*:

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t,a_t)}[\max_{a_{t+1} \in \mathcal{A}} Q(s_{t+1}, a_{t+1})] \tag{3.10}$$

Then, the optimal policy is defined as $\pi(a_t|s_t) = \arg\max_{a_t \in \mathcal{A}} Q(s_t, a_t)$. To update the $Q(s_t, a_t)$ value, normally DQN uses a neural network as a nonlinear function approximator to represent the Q value. The update rule is defined as:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha\big(r(s_t, a_t) + \gamma \max_{a_{t+1} \in \mathcal{A}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big) \tag{3.11}$$

where α is the learning rate. We can re-write Equation 3.11 as a loss function by using gradient descent to update the parameters as:

$$\mathcal{L}_\theta = \mathbb{E}_{s_t,a_t}\Big[\big(r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t,a_t)} \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big)^2\Big] \tag{3.12}$$

By using training set sampled from the experience replay buffer [88], the gradient is:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \big(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big)\frac{\partial Q(s_t, a_t; \theta)}{\partial \theta} \tag{3.13}$$

The original DQN uses Temporal Difference learning (TD(0)), that is training network parameters in every step. In each step, the agent samples the trajectory from the experience replay buffer [88], updating the network parameters via Equation 3.12. [86] proposed asynchronous n-step Q-learning. Similar to

Asynchronous Advantage Actor Critic (A3C) algorithm, the n-step Q-learning use multiple asynchronous threads to run different policy on each corresponding environment, then it becomes on-policy since it avoids using experience replay buffer to sample the low quality policy at the initial training stage.

3.1.4 Episodic Q-learning

In GVIN, we train the agent in an episodic way to stabilize the training process, which is similar to Monte Carlo control (ex: REINFORCE). We highlight the differences between episodic Q-learning and the original n-step Q-learning in blue, including the initial expected return, the termination condition, and the timing of updating the gradient.

Algorithm 3 Episodic Q-learning

```
1:  input graph $G$ and the goal $s_g$
2:  initialize global step counter $T = 0$
3:  initialize GVIN parameters $\mathbf{w} = [\mathbf{w_r}, \mathbf{w}_{P^{(a)}}]$
4:  initialize parameter gradients $\Delta\mathbf{w}$
5:  repeat (one episode)
6:      clear gradients $\Delta\mathbf{w} \leftarrow 0$
7:      $t = 0$
8:      randomly pick a start node $s_t$
9:      repeat (one action)
10:         take action $a_t$ according to the $\epsilon$-greedy policy based on $\mathbf{q}_{s_t}^{(a)}$
11:         receive reward $r_t$ and new state $s_{t+1}$
12:         $t \leftarrow t + 1$
13:     until terminal $s_t = s_g$ or $t > t_{\max}$
14:     $R = 0$
15:     for $i = t : -1 : 0$ do
16:         $R \leftarrow r_i + \gamma R$
17:         accumulate gradients wrt $\mathbf{w}$ : $\Delta\mathbf{w} \leftarrow \Delta\mathbf{w} + \frac{\partial(R - \mathbf{q}_{s_t}^{(a)})^2}{\partial\mathbf{w}}$
18:     end for
19:     $\mathbf{w} \leftarrow \mathbf{w} - \Delta\mathbf{w}$
20:     $T = T + 1$
21: until $T > T_{\max}$
```

3.1.5 Computational Complexity

Let the input graph G have $|\mathcal{V}|$ nodes and $|\mathcal{E}|$ edges. In the testing phase, since the input graph is commonly sparse, the computational complexities of (3.1), (3.3), and (3.4) are $O(|\mathcal{V}|)$, $O(|\mathcal{E}|)$, and $O(|\mathcal{V}|)$ based on sparse computation, respectively. Therefore, the total computational complexity is $O(|\mathcal{V}| + K(|\mathcal{E}| + |\mathcal{V}|))$, where K is number of iterations. For a spatial graph, the number of edges is usually proportional to the number of nodes; thus the computational complexity is $O(K|\mathcal{V}|)$, which is scalable to large graphs.

3.1.6 Connections Between GVIN and Graph Convolutional Neural Network

Here we highlight the relation between GVIN and GCNN [9] which is recently proposed. GCNN defined as 3-tuple $G = (u, V, E)$ is a new type of neural network architecture that follows graph input/output interface. The general GCNN block consists of *update* function ϕ and *aggregation* function ρ for edges E, vertices V and global variables u, respectively. In GVIN, the architecture can be decomposed as graph convolution operator and kernel functions, where graph convolution operator can be viewed as aggregation function ρ, and kernel function could be viewed as update function ϕ. Since GVIN does not contain any global variables and edge embeddings, GVIN is a special case of the GCNN.

Current GCNN are widely used in the inference problem [147]. GVIN, which is a subset configuration of GCNN, bridges the gap between control problem and inference, showing that planning problem has equivalence to inference problem. This conclusion has also been derived by [77], and our GVIN verify the findings empirically. The conclusion potentially push Machine Learning community to rethink a general framework for both control tasks and inference tasks. Additionally, the employment of GCNN leverages auto differentiable system such as neural network to classical inference methodologies.

3.2 Unified Framework for Graph Embeddings

One question not answered from GVIN is, "where does the graph embedding come from?" In this subsection, we proposed a unified framework for graph embedding.

A desirable graph embedding algorithm should (1) work for various types of graphs, including directed, undirected, weighted, unweighted, and bipartite; (2) preserve the symmetry between the graph node domain and the graph embedding domain; that is, when two nodes are close/far away in the graph node domain, their corresponding embeddings are close/far away in the graph embedding domain; (3) be scalable to large graphs; and (4) be interpretable, that is, we understand the role of each building block in the graph embedding algorithm. On the heels of these outstanding contributions, can we ask:

Q1. Which factor in the previous embeddings matters?

Q2. Can we find an even better embedding?

Q3. Can we summarize the design of graph embedding in a unifying framework?

To answer the first question, we propose a unifying graph embedding framework, called Graph EMbedding

GEM-F	**LapEigs [10]**	**DiffMaps [27]**	**LINE [131]**	**DeepWalk [97]**	**node2vec [46]**	**GrapRep [21]**	**WarpMap**
Loss $d(\cdot,\cdot)$	Frobenius	Frobenius	KL	KL	KL	Frobenius	Frobenius
Proximity $h(\cdot)$	Laplacian	transition	transition	FST	FSMT	FST	FST
Warping $g(\cdot)$	linear	linear	sigmoid	exponential	exponential	exponential	exponential
Closed form	✓	✓				✓	✓
High-order		✓	✓	✓	✓	✓	✓
Warped			✓	✓	✓	✓	✓
Explanation	✓	✓					✓
Scability			✓	✓	✓		✓

Table 3.1: Unifying power of GEM-F. Many graph embeddings, including LapEigs, DiffMaps, LINE, DeepWalk, node2vec, GrapRep are specific cases. WarpMap wins the competition.

Framework (GEM-F), which decomposes a graph embedding algorithm into three building blocks: node proximity function, warping function, and loss function. GEM-F clarifies the functionality of each building block and subsumes many previous graph embedding algorithms as special cases. Based on GEM-F, we distinguish previous embedding designs. To answer the second question, based on the insights GEM-F brings and extensive experimental validations, we propose a graph embedding algorithm, WarpMap, which inherits advantages of previous algorithms; it is general enough to handle diverse types of graphs, has a closed-form solution, and is interpretable. To answer the third question, we propose a scalable version of WarpMap, which performs 6 times faster than DeepWalk.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathrm{A})$ be a directed and weighted graph, where $\mathcal{V} = \{v_i\}_{i=1}^N$ is the set of nodes, $\mathcal{E}$ is the set of weighted edges, and $\mathrm{A} \in \mathcal{R}^{N \times N}$ is the weighted adjacency matrix[1], whose element $\mathrm{A}_{i,j}$ measures the direct relation between the ith and jth nodes. Let $\mathrm{D} \in \mathcal{R}^{N \times N}$ be a diagonal degree matrix, where $\mathrm{D}_{i,i} = \sum_{j \in \mathcal{V}} \mathrm{A}_{i,j}$. To represent the pairwise node proximity in the graph node domain, we define a proximity matrix as $\Pi = h(\mathrm{A}) \in \mathcal{R}^{N \times N}$, where A is the adjacency matrix and $h(\cdot)$ is a node proximity function we will discuss in Section 3.2.1.

Here we answer the first question, by providing a unifying framework, and showing that all the above algorithms are special cases. Later, we answer the second question, proposing WarpMap, which ties or outperforms top competitors on accuracy and is up to 6 times faster.

Let $\mathrm{F}, \widehat{\mathrm{F}} \in \mathcal{R}^{N \times K} (K \ll N)$ be a pair of dual embedding matrices with their ith rows, $\mathbf{f}_i, \widehat{\mathbf{f}}_i \in \mathcal{R}^K$, the hidden features of the ith node. F and $\widehat{\mathrm{F}}$ capture in-edge and out-edge behaviors, respectively. The final graph embedding is a concatenation of these two. In the graph embedding domain, we use the inner product of $\mathbf{f}_i$ and $\widehat{\mathbf{f}}_j$ to capture the pairwise proximity between the ith node and jth nodes, that is, we define $g(\mathrm{F}\widehat{\mathrm{F}}^T) \in \mathcal{R}^{N \times N}$

[1]The proposed framework and algorithm can be easily extended to bipartite graphs.

as the proximity matrix in the graph embedding domain. The (i, j)th element $g(\mathbf{f}_i^T \widehat{\mathbf{f}}_j)$ measures the proximity between the ith and jth nodes in the graph embedding domain, with the warping function $g(\cdot)$, a predesigned element-wise monotonically increasing function, which we will discuss further in Section 3.2.2.

Our goal is to find a pair of low-dimensional graph embeddings such that the node proximities in both domains are close, that is, $\Pi_{i,j}$ is close to $g(\mathbf{f}_i^T \widehat{\mathbf{f}}_j)$ for each pair of i, j. We thus consider a general formulation, GEM-F, as

$$\mathrm{F}^*, \widehat{\mathrm{F}}^* \;=\; \arg \min_{\mathrm{F}, \widehat{\mathrm{F}} \in \mathcal{R}^{N \times K}} d(h(\mathrm{A}), g(\mathrm{F}\,\widehat{\mathrm{F}}^T)), \tag{3.14}$$

where $d(\cdot, \cdot)$ is a loss function that measures the difference; we discuss it further in Section 3.2.3. $g(\mathrm{F}\,\widehat{\mathrm{F}}^T)$ is generated from a low-dimensional space as $\mathrm{F}^*, \widehat{\mathrm{F}}^* \in \mathcal{R}^{N \times K}$, which is used to approximate an input Π. We minimize the loss function (3.14) to control the approximation error. GEM-F involves the following three important building blocks, denoted by GEM-F$[h(\cdot), g(\cdot), d(\cdot, \cdot)]$:

- *node proximity function* $h(\cdot) : \mathcal{R}^{N \times N} \to \mathcal{R}^{N \times N}$;
- *warping function* $g(\cdot) : \mathcal{R} \to \mathcal{R}$ (when $g(\cdot)$ applies to a matrix, it is element-wise); and
- *loss function* $d(\cdot, \cdot) : \mathcal{R}^{N \times N} \times \mathcal{R}^{N \times N} \to \mathcal{R}$.

The general framework GEM-F makes it clear that all information comes from the graph adjacency matrix A. Based on the properties of graphs, we adaptively choose three building blocks and solve the corresponding optimization problem to obtain a desirable graph embedding. We consider GEM-F at a general and abstract level to demystify the constitution of a graph embedding algorithm and analyze the function of each building block.

3.2.1 Node Proximity Function $h(\cdot)$

The proximity matrix is a function of the adjacency matrix, $\Pi = h(\mathrm{A})$; in other words, all information comes from given pairwise connections. The node proximity function extracts useful features and removes useless noise for subsequent analysis; for example, the node proximity function can normalize the effect from hubs and capture the high-order proximity information. The node proximity function is implicitly used in DeepWalk and node2vec, where the proximity matrix is obtained via random-walk simulation. We consider an explicit matrix form, so that we are able to analyze the asymptotic behavior. We now consider some choices for the proximity matrix Π.

1. *Laplacian matrix.* The Laplacian matrix $\mathcal{L} \in \mathcal{R}^{N\times N}$ is defined as $\mathcal{L} = \mathrm{D} - \mathrm{A}$; this is the matrix used in LapEigs [10].

2. *Transition matrix.* The transition matrix $\mathrm{P} \in \mathcal{R}^{N\times N}$ is defined as $\mathrm{P} = \mathrm{D}^{-1}\,\mathrm{A}$. The element $\mathrm{P}_{i,j}$ is the transition probability of a random walker at the ith node visiting the jth node in a single step; this is the matrix used in diffusion maps [27] and LINE [131].

   Both the Laplacian matrix and the transition matrix normalize the original adjacency matrix based on the node degrees, which reduces the effect from hubs.

3. *Finite-step transition matrix.* The finite-step transition (FST) matrix is defined as a finite-order polynomial in P (transition matrix), $\Pi^{(L)} = \sum_{\ell=1}^{L} \mathrm{P}^{\ell}$, where $L \geq 1$ is a finite number of steps. The FST matrix models the random walker walking L steps randomly and sequentially. When a random walker starts at the ith node, the expected number of times to visit the jth node is $\Pi^{(L)}_{i,j}$; this is the matrix implicitly used in DeepWalk [97].

4. *Infinite-step transition matrix.* The infinite-step transition (IST) matrix is defined as an infinite-order polynomial in P (transition matrix),

$$\Pi^{(\alpha)} = \sum_{\ell=1}^{\infty} \alpha^{\ell-1}\,\mathrm{P}^{\ell} = \frac{1}{\alpha}\left(\mathcal{P} - \mathrm{I}\right),$$

   where $\mathcal{P} = \left(\mathrm{I} - \alpha\,\mathrm{P}\right)^{-1}$ and $0 < \alpha < 1$. The IST matrix models the random walker stopping after each step with probability $1 - \alpha$. When a random walker starts at the ith node, the expected number of times to visit the jth node is $\Pi^{(\alpha)}_{i,j}$; this is the matrix used in PageRank [17] and personalized PageRank [50].

   Both FST and IST matrices consider not only direct relations but high-order, long-range node proximities.

5. *Finite-step walk-memory-modulated transition matrix.* The finite-step walk-memory-modulated transition (FSMT) matrix is a generalized version of the FST matrix, which also takes the walk memory into account. The walk memory factor is introduced in node2vec [46] without an explicit matrix form. Here we derive its closed form. The walk memory factor records each previous step and influences the walking direction, leading to a biased random walk, which is a trade-off between breadth-first search

(BFS) and depth-first search (DFS). The walk memory matrix $\mathcal{M} \in \mathcal{R}^{N \times N}$ is defined as:

$$\mathcal{M}_{i,k} = \begin{cases} 1/p, & i = k; \\ 1, & i \text{ and } k \text{ are adjacent}; \\ 1/q, & \text{geodesic distance between } i \text{ and } k \text{ is } 2; \\ 0, & \text{otherwise}. \end{cases}$$

The walk memory factors p and q control the walking direction: when p is small, the walker tends to do BFS and when q is small, the walker tends to do DFS. The FSMT matrix is defined as:

$$\Pi^{(L,p,q)} = \Phi \left(\sum_{\ell=0}^{L-1} \mathrm{W}^{\ell} \right) \mathrm{Q}^T,$$

where $L \geq 1$ is a finite number of walk length and the initial probability matrix is $\mathrm{Q} = \begin{bmatrix} \mathrm{Q}_1 & \mathrm{Q}_2 & \cdots & \mathrm{Q}_N \end{bmatrix} \in \mathcal{R}^{N \times N^2}$, where $\mathrm{Q}_k \in \mathcal{R}^{N \times N}$ with $(\mathrm{Q}_k)_{m,\ell} = \mathrm{P}_{k,m} \mathcal{I}(\ell = m)$ (P is the transition matrix and $\mathcal{I}(\cdot)$ is the indicator function), the merging matrix is $\Phi = \mathrm{I}_N \otimes \mathbf{1}_N \in \mathcal{R}^{N \times N^2}$, where $\mathrm{I}_N \in \mathcal{R}^{N \times N}$ is an identity matrix, $\mathbf{1}_N \in \mathcal{R}^N$ is an all-one vector, and the expanded transition matrix is

$$\mathrm{W} = \begin{bmatrix} \mathrm{W}_{1,1} & \mathrm{W}_{1,2} & \cdots & \mathrm{W}_{1,N} \\ \mathrm{W}_{2,1} & \mathrm{W}_{2,2} & \cdots & \mathrm{W}_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{W}_{N,1} & \mathrm{W}_{N,2} & \cdots & \mathrm{W}_{N,N} \end{bmatrix} \in \mathcal{R}^{N^2 \times N^2},$$

where $\mathrm{W}_{i,k} \in \mathcal{R}^{N \times N}$ whose elements are

$$(\mathrm{W}_{i,k})_{j,\ell} = \frac{\mathrm{A}_{i,k} \mathcal{M}_{i,\ell}}{\sum_v \mathrm{A}_{v,k} \mathcal{M}_{v,\ell}} \mathcal{I}(j = k).$$

The FSMT matrix is modulated by the walk memory matrix $\mathcal{M}$. When a walk-memory-modulated random walker starts at the ith node and walks L steps sequentially, the expected number of times to visit the jth node is $\Pi_{i,j}^{(L,p,q)}$. When $p = q = 1$, the FSMT matrix is the FST proximity; that is, $\Pi^{(L,1,1)} = \Pi^{(L)}$. This is the matrix implicitly used in node2vec [46].

3.2.2 Warping Function $g(\cdot)$

The warping function warps the inner products of graph embeddings and normalizes the distribution of elements in $g(\mathrm{F}\,\widehat{\mathrm{F}}^T)$, providing an appropriate scale to link the proximity matrices in both node and embedding domains. The warping function is applied element-wise and monotonically increasing.

We here consider the inverse Box-Cox transformation (IBC) as the warping function, defined as:

$$g(x) = \mathrm{IBC}(\gamma, x) = \begin{cases} (1+\gamma x)^{\frac{1}{\gamma}}, & \gamma \neq 0; \\ \exp(x), & \gamma = 0, \end{cases}$$

where γ measures the nonlinearity. When $\gamma = 1$, $g(x) = 1 + x$; this linear function is used in LapEigs [10], diffusion maps [27], and the exponential function in SNE [57], DeepWalk [97], and node2vec [46].

3.2.3 Loss Function $d(\cdot, \cdot)$

The loss function quantifies the differences between the proximity matrices in two domains. Since $g(\cdot)$ is monotonic, when the rank of $g^{-1}(\Pi)$ is K, we can find a pair of $\mathrm{F}, \widehat{\mathrm{F}} \in \mathcal{R}^{N\times K}$ satisfying $\mathrm{F}\,\widehat{\mathrm{F}}^T = g^{-1}(\Pi)$ or $g(\mathrm{F}\,\widehat{\mathrm{F}}^T) = \Pi$. In this case, any valid loss function $d(\Pi, g(\mathrm{F}\,\widehat{\mathrm{F}}^T))$ should be zero, that is, the problem does not depend on the choice of loss function. When the rank of $g^{-1}(\Pi)$ is far-away from K, however, an appropriate loss function can still be beneficial to the overall performance.

1. *KL divergence:* We can use the KL-divergence based loss function to quantify the difference between Π and $g(\mathrm{F}\,\widehat{\mathrm{F}}^T)$. To simplify the notation, let $\mathrm{Y} = g(\mathrm{F}\,\widehat{\mathrm{F}}^T) \in \mathcal{R}^{N\times N}$. We normalize the sum of each row in both Π and Y to be 1 and use the KL divergence to measure the difference between two corresponding probability distributions. The total loss between Π and Y is:

$$\begin{aligned} d_{\mathrm{KL}}(\Pi, \mathrm{Y}) &= -\sum_{i\in\mathcal{V}} \mathrm{KL}(\Pi_{i,:} \| \mathrm{Y}_{i,:}) \\ &= -\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} \left(\frac{\Pi_{i,j}}{\sum_k \Pi_{i,k}}\right) \log\left(\frac{\Pi_{i,j}}{\sum_k \Pi_{i,k}} / \frac{\mathrm{Y}_{i,j}}{\sum_k \mathrm{Y}_{i,k}}\right), \end{aligned}$$

where $\Pi_{i,:}$ and $\mathrm{Y}_{i,:}$ denote the ith rows of Π and Y, respectively. We then solve the following optimization problem:

$$\mathrm{F}^*, \widehat{\mathrm{F}}^* = \arg \min_{\mathrm{F}, \widehat{\mathrm{F}} \in \mathcal{R}^{N\times K}} d_{\mathrm{KL}}(\Pi, \mathrm{Y}). \tag{3.15}$$

We use the gradient descent to optimize (3.15). A stochastic algorithm can be used to accelerate the optimization, such as negative sampling, edge sampling, or asynchronized stochastic gradient descent [83, 84, 131].

2. *Warped Frobenius norm.* The warped-Frobenius-norm based loss function is defined as:

$$d_{\mathrm{wf}}(\Pi, g(\mathrm{F}\,\widehat{\mathrm{F}}^T) = \left\| \mathrm{F}\,\widehat{\mathrm{F}}^T - g^{-1}(\Pi) \right\|_F^2, \tag{3.16}$$

where $g^{-1}(\cdot)$ is the inverse function of $g(\cdot)$. We thus solve the following optimization problem:

$$\mathrm{F}^*, \widehat{\mathrm{F}}^* = \arg \min_{\mathrm{F}, \widehat{\mathrm{F}} \in \mathcal{R}^{N \times K}} d^2_{\mathrm{wf}}(\Pi, g(\mathrm{F}\,\widehat{\mathrm{F}}^T). \tag{3.17}$$

The closed-form solution of (3.17) can be efficiently obtained from the truncated singular value decomposition as follows:

- $\mathrm{U}_{[N\times K]}, \Sigma_{[K\times K]}\,\mathrm{V}_{[N\times K]} \leftarrow \mathrm{SVD}(g^{-1}\,(\Pi)\,, K);$
- $\text{Obtain } \mathrm{F}_{[N\times K]} \leftarrow \mathrm{U}\,\Sigma^{\frac{1}{2}};$
- $\text{Obtain } \widehat{\mathrm{F}}_{[N\times K]} \leftarrow \mathrm{V}\,\Sigma^{\frac{1}{2}}.$

3.2.4 Specific Cases of GEM-F

We articulate the unifying power of GEM-F through the special cases that it encompasses.

Lemma 1. GEM-F includes DeepWalk as a special case with corresponding triple GEM-F$[\Pi^{(L)}, \exp(x), d_{\mathrm{KL}}(\cdot,\cdot)]$. DeepWalk [97] is a randomized implementation of GEM-F when the proximity matrix is the FST matrix, the warping function is the exponential function and the loss function is the KL divergence.

Lemma 2. GEM-F includes node2vec as a special case with the corresponding triple GEM-F$[\Pi^{(L,p,q)}$, $\exp(x), d_{\mathrm{KL}}(\cdot,\cdot)]$. node2vec [46] is a randomized implementation of GEM-F when the proximity matrix is the FSMT matrix, the warping function is the exponential function and the loss function is the KL divergence.

In the original DeepWalk and node2vec, the authors did not specify the closed-form proximity matrices. They estimated the proximity matrices through random-walk simulations. Here we provide their closed forms, which are equivalent to doing infinitely many random walk simulations.

Lemma 3. GEM-F includes LINE as a special case with the corresponding triple GEM-F$[\mathrm{P}, \frac{1}{1+\exp(-x)}$, $d_{\mathrm{KL}}(\cdot,\cdot)]$ (first-order proximities) and GEM-F$[\mathrm{P}, \exp(x), d_{\mathrm{KL}}(\cdot,\cdot)]$ (second-order proximities) . LINE [131] is an implementation of GEM-F when the proximity matrix is the transition matrix, the warping function is the sigmoid/exponential function and the loss function is the KL divergence.

Lemma 4. GEM-F includes matrix factorization as a special case with the corresponding triple GEM-F$[\mathrm{A}, x, d_{\mathrm{wf}}(\cdot,\cdot)]$. Matrix factorization [70] is an implementation of GEM-F when the proximity matrix is the adjacency matrix, the warping function is the linear function and the loss function is the warped Frobenius norm.

3.2.5 Proposed Design: WarpMap

Based on the GEM-F framework and our upcoming experiments in Section 5.2, we observe that (1) the transition matrix with high-order proximity matters, while the walk memory factor does not (at least not as much); (2) the warping function is the key contributor to the performance and the exponential function usually performs the best; and (3) once the node proximity function and warping function are properly selected, the distance function does not matter. We thus choose a triple: the FST matrix, exponential function and warped Frobenius norm to propose $\boxed{\text{WarpMap = GEM-F}[\Pi^{(L)}, \exp(x), d_{\mathrm{wf}}(\cdot,\cdot)]}$, where $d_{\mathrm{wf}}(\cdot,\cdot)$ is defined in (3.16).

3.2.6 Closed Form

The WarpMap algorithm solves

$$\mathrm{F}^*, \widehat{\mathrm{F}}^* = \arg \min_{\mathrm{F}, \widehat{\mathrm{F}} \in \mathcal{R}^{N \times K}} \left\| \mathrm{F}\,\widehat{\mathrm{F}}^T - \log\left(\Pi^{(L)}\right) \right\|_F^2,$$

where L is the diameter of the given graph (based on our experiments, we set our default value to be 7). To ensure numerical correctness, during the computation, we take the log of non-zero elements and replace others with $-c$ (the default value for c is 100), that is, we set

$$\mathrm{Z} = \log(\Pi^{(L)})\mathcal{I}(\Pi^{(L)} \neq 0) - c\mathcal{I}(\Pi^{(L)} = 0) \in \mathcal{R}^{N \times N}, \tag{3.19}$$

where $\mathcal{I}(\cdot)$ is the element-wise indicator function. The solution is obtained from the truncated singular value decomposition of Z. WarpMap has the closed-form solution; the choice of each building block in WarpMapis based on an insightful understanding of GEM-F. We find that $\log(\Pi^{(7)})$ maximizes the empirical performance by normalizing the distribution of elements in $\log(\Pi^{(7)})$. Specifically, $L = 7$ is usually the diameter of a social network and $\Pi^{(7)}$ properly diffuses information to all nodes without introducing echo and redundancy. Empirically, we observe that the elements in $\Pi^{(7)}$ approximately follow a log-normal distribution and SVD can only preserve high-magnitude elements. The logarithmic factor aids in normalizing the distribution, such that SVD preserves more information in $\Pi^{(7)}$ (see Figure 5.11).

3.2.7 Scalable Implementation

Since $\Pi^{(L)}$ is not necessarily a sparse matrix, when the graph is large, we cannot afford to compute the closed form of $\log(\Pi^{(L)})$, which is order of $O(N^2)$; instead, we estimate it via random-walk simulation. Specifically, to estimate the (i,j)th element $\Pi^{(L)}_{i,j}$, we run L-step random walks from the ith node for m independent trials and record the number of times that the jth node has been visited, denoted by $\mathrm{S}_{i,j}$. In each trial, we start from the ith node and walk L steps randomly and sequentially. The estimate of $\Pi^{(L)}_{i,j}$ is then $\widetilde{\Pi}^{(L)}_{i,j} = \mathrm{S}_{i,j}\,/m$. To work with a sparse matrix, we further obtain a proxy matrix $\widetilde{\mathrm{Z}} \in \mathcal{R}^{N\times N}$ defined as

$$\widetilde{\mathrm{Z}} = \left(\log(\widetilde{\Pi}^{(L)}) + c\right)\mathcal{I}(\widetilde{\Pi}^{(L)} \neq 0) \in \mathcal{R}^{N\times N}.$$

When $m \to +\infty$, $\widetilde{\Pi}^{(L)}_{i,j} \to \Pi^{(L)}_{i,j}$ and $\widetilde{\mathrm{Z}} \to \mathrm{Z} + c\mathbf{1}_N\mathbf{1}_N^T$, where Z follows from Equation (3.19)w, we obtain the solution by the singular value decomposition of the proxy matrix $\widetilde{\mathrm{Z}}$. Since $\widetilde{\Pi}^{(L)}$ is sparse, $\widetilde{\mathrm{Z}}$ is sparse and the computation of singular value decomposition is cheap; we call this approach scalable WarpMap. We add constant c to all the elements in $\log(\widetilde{\Pi}^{(L)})$, which will introduce a constant singular vector and does not influence the shape of the graph embedding.

Lemma 5. WarpMap is scalable. The time and space complexities of scalable WarpMap are at most $O(K\max(|\mathcal{E}|,\ NLm))$ and $O(\max(|\mathcal{E}|,\ NLm))$, respectively, where N is the size of graph, K is the dimension of graph embedding, L is the walk length and m is the number of random-walk trials.

Proof. Each random walk from a given node can visit at most L unique nodes and m such walks would give non-zero proximity for a node to at most Lm neighbors. Thus, the space requirement is $O(\max(|\mathcal{E}|, NLm))$. The complexity of the truncated singular value decomposition of $\widetilde{\mathrm{Z}}$ is then $O(K\max(|\mathcal{E}|,\ NLm))$. ■

In practice, a few nodes are visited more frequently than others; thus, the sparsity of the proxy matrix $\widetilde{\mathrm{Z}}$ is much smaller than $O(NLm)$.

3.2.8 Acceleration via Data Splitting

Instead of using all of the random walks to estimate $\Pi^{(L)}$ and then taking the logarithm operator, we run T threads in parallel, estimate $\log(\Pi^{(L)})$ in each thread, and finally average the results in all threads to obtain a more accurate estimation of $\log(\Pi^{(L)})$. The advantages are two-fold: (1) we parallelize the procedure; (2) we focus on estimating $\log(\Pi^{(L)})$, not $\Pi^{(L)}$. As mentioned earlier, the empirical observations show that each element in $\log(\Pi^{(L)})$ approximately follows the normal distribution. The following lemma ensures that the data splitting procedure provides the maximum likelihood estimator of $\log(\Pi^{(L)})$.

Lemma 6. Average provides the maximum likelihood estimator. Denote the (i, j)th element of $\widetilde{\Pi}^{(L)}$ in the tth thread as x_t. Let $\log(x_1)$, $\log(x_2)$, $\cdots$, $\log(x_T)$ be independent and identically distributed random variables, which follow the normal distribution with mean $\log(\Pi_{i,j})$ and variance σ^2. Then, the maximum likelihood estimator of $\log(\Pi_{i,j})$ is $\sum_{t=1}^{T} \log(x_t)/T$.

Proof. The proof follows from the maximum likelihood estimator of the log-normal distribution. ■

Guided by Lemma 6 and considering numerical issues, we obtain the proxy matrix $\widetilde{Z}_t$ in the tth thread, average the proxy matrices in all the threads, and obtain $\widetilde{Z}_{\text{avg}} = \sum_{t=1}^{T} \widetilde{Z}_t/T$, which estimates $\log(\Pi^{(L)}) + c$. Finally, we calculate the singular value decomposition of $\widetilde{Z}_{\text{avg}}$.

3.3 Domain Alignment for Visual-Based Applications

In this subsection, we focus on the reality gap issue of reinforcement learning, that is how to align the features in different domains so that the agent could directly adopt trained policy from the simulator environment to the real world. Most previous related work investigates the image generation and classification from source domain to target domain at the pixel level, while this work is mainly focused on feature alignment at embedding level in control scenarios, such as reinforcement learning.

3.3.1 Problem Setup

We consider that the domain transfer in RL settings can be decomposed as representation transfer and policy transfer. In this work, the focus is on the representation transfer, and the policy transfer is left for future work. Particularly, each agent could be partitioned as a representation module $V(z; \theta)$ and a policy module $A(x; \phi)$ as shown in Figure 3.5, where $V(z; \theta)$ produces the better feature representation x, conditioned

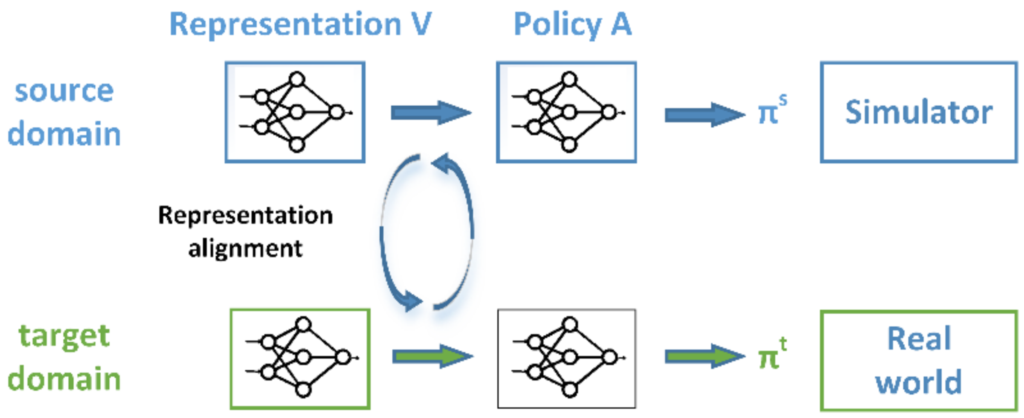


Figure 3.5: Problem setup illustration. The network with the blue bounding box represents the network trained in the source domain, while the green one represents the network running in the target domain. In our setup, the representation/perception module in target domain is pre-trained, while the parameters of policy module are unknown.

on input state z, and $A(x;\phi)$ generates an action conditioned on x. θ and ϕ denote the module weights, respectively. In deep reinforcement learning, V and A are both neural networks, which are pipelined and trained via backpropagation.

The proposed study handles image-based domain transfer from domain $\mathcal{S}$ to domain $\mathcal{T}$. We assume that the representation network V_s and policy network A_s in the source domain $\mathcal{S}$ is fully trained to perform the same task in target domain $\mathcal{T}$, whereas in domain $\mathcal{T}$ the representation module V_t is pre-trained and the policy module is not. This is often true in a real world environment, where the computer vision community has already collected large amounts of real world image data [31, 150] and investigated a variety of neural network architectures.

We aim to duplicate the policy network parameters ϕ_s trained in $\mathcal{S}$ as ϕ_t which applies in $\mathcal{T}$, so that π_s in domain $\mathcal{T}$ is still functional with much less policy parameter tuning or even without any adaptions. This is a typical domain transfer problem that trains the model from source domain and generalizes to target domain. Prior works in RL are mainly grouped into two types: feature-level and pixel-level adaption. The feature-level adaption makes the features extracted from both domains similar, while pixel-level re-renders the original source domain image to make it look close to the target domain image. Our work focuses on the feature-level category, since direct feature alignment is not only general to many other perception tasks but also computationally efficient.

3.3.2 Fusion Layer based on Adversarial Learning

We introduce an additional fusion layer inserted between the perception module and the policy module to perform feature alignment as shown in Figure 3.6. A major difficulty in constructing the fusion layer is to align features without any paired labels or dictionary. The hypothesis is that domain $\mathcal{S}$ and domain $\mathcal{T}$ should share a similar feature structure in latent space. Assuming both source and target feature vectors are in d-dimensional feature space, similar to cross-lingual word embeddings [28, 85], in which a linear mapping function W is used between the source and target domain, we define the fusion layer as a linear transformation:

$$W^{*} = \arg\min_{W} ||WX - Y||_{F} \tag{3.20}$$

where $W \in \mathbb{R}^{d \times d}$ denotes mapping parameters. In the case of supervised learning, which a paired dictionary $\{x_i, y_i\}_{i \in \{1,n\}}$ is given ($x_i \in \mathbb{R}^d$ is in source space and $y_i \in \mathbb{R}^d$ is in target space), then the optimization of the mapping function becomes the Procrustes problem where W is orthogonal. The advantages of orthogonal Procrustes yields a closed form solution as:

$$W^{*} = UV^{T}, \text{ where } U\Sigma V^{T} = \text{ SVD } (YX^{T}) \tag{3.21}$$

In most cases, the pairwise dictionary is challenging to obtain. We use adversarial learning to train the mapping function without any supervision. In the case of language (MUSE [28]), which learns the alignment between two cross-lingual word embeddings, it pre-assumes that the structure of the target space and source space are the same or highly similar. However, in the perception-based applications, the image embedding holds the following differences compared to word embedding: 1) each word is a symbol that has its paired word in the target domain, while the image does not since it is in high-dimensional continuous space, and thereby two domains are not guaranteed to have the same structures; 2) the structure of word embedding is fixed in one language, whereas the image embedding is domain-specific and varies case by case. As we will show in the experiment, directly applying MUSE yields poor performance. In the next section, we will describe how we enhance the model to train the fusion layer in adversarial settings effectively (Figure 3.6).

3.3.3 Adversarial Learning

For each batch iteration, let X $\in \mathbb{R}^{n \times d}$ and Y $\in \mathbb{R}^{n \times d}$ be two sets of image embeddings randomly sampled from the source and the target domain, where n refers to batch size. We define a discriminator that

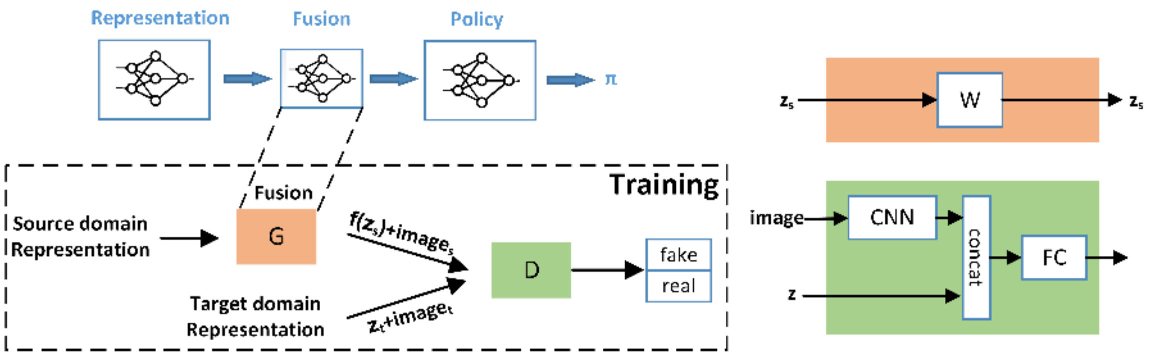


Figure 3.6: Fusion layer architecture and training. The fusion layer, which is a linear transformation, is inserted between the representation module and the policy module; the lower box shows the adversarial training setup for the fusion layer, described in Section 3.3.3; the right hand side of the figure shows the network architecture of the discriminator and mapping function (fusion layer). Implementation details are described in Section 5.3

maximizes the ability to distinguish whether the embedding comes from the term W X or Y, whereas the goal of the mapping function is to maximize the probability that the discriminator makes a mistake.

The goal of the feature alignment is to aligned each paired features between two different domains. In other words, we interest in using minimum cost of effort in converting the data distribution $\mathbb{P}_g$ to the data distribution $\mathbb{P}_t$, which is conceptually similar to Wasserstein distance. Instead of using the vanilla Generative Adverserial Network (GAN) [45], which minimize the Jensen Shannon (JS) divergence between generated data distribution and real data distribution, we use Wasserstein GAN [7] training metrics since the Wasserstein distance inherently defines the feature alignment.

Concretely, we assume that the target data distribution $\mathbb{P}_t$ admits a density and $\mathbb{P}_g$ a parameterized density, which in our case is the density function of the mapped embedding. In order to measure the similarity of the aligned embedding and target embedding, the choice of distance or divergence $\rho(\mathbb{P}_t, \mathbb{P}_g)$ should also imply the cost of the mapping transformation. Therefore, we define the mapping cost as the minimum cost of transporting mass in order to transform the distribution $\mathbb{P}_g$ to distribution $\mathbb{P}_t$, where the Earth-Mover distance (Wasserstein-1 distance) $W(\mathbb{P}_t, \mathbb{P}_g)$ is defined as:

$$W(\mathbb{P}_t, \mathbb{P}_g) = \inf_{\gamma \in (\mathbb{P}_t, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{3.22}$$

The Earth-Mover distance is adopted as the cost to measure the feature alignment in adversarial learning, which leads our adversarial learning as Wasserstein GAN [7], and we observed better performance compared with vanilla GAN used in MUSE settings (see Section 5.3.1). Our adversarial learning value function is constructed as:

$$\min_{W} \max_{D \in \mathcal{D}} \mathop{\mathbb{E}}_{y \sim \mathbb{P}_t} [D(y)] - \mathop{\mathbb{E}}_{\hat{x} \sim \mathbb{P}_g} [D(\hat{x})] \tag{3.23}$$

where $\mathcal{D}$ denotes 1-Lipschitz functions, and $\hat{x}$ is defined as Wx. In this min-max game, the discriminator distinguishes between the mapped feature embeddings and target feature embeddings. It turns out that high-dimensional abstracted features are already hard for the discriminator to critique (see Figure 5.16), so that the mapping function can easily fool the discriminator by approximating the mapped embedding structure (ex: Euclidean distance/dot product) to the target embedding structure without learning the paired semantic meanings, which causes *mode collapse* issues. We propose two methods to prevent *mode collapse*: pre-clustering and feature diversity.

• **Pre-clustering**. Clustering reflects the inherent structure of data by discretizing latent space into small partitions. Instead of learning the mapping on the original embedding data, our first approach learns clustered embedding that leverages the intrinsic data structure via K-means clustering, where other approaches could be applied. The K-means algorithm clusters k centroids as embedding representations. Then, we apply adversarial learning only on these k centroids. In every training epoch, we reset the K-means centroids for better generalization. Pre-clustering reinforces the mapping function on the coarse structure of the data while ignoring the fine-grain details. However, the vanilla adversarial learning and pre-clustering version are both sensitive to the embedding structure, and the algorithm may fail (see Section 5.3.2) if the source domain and target domain have significantly different structures.

• **Feature Diversity** In a conventional GAN, the discriminator often critiques on image-based (or pixel-level) samples. The convolutional neural network (CNN) in the discriminator provides abstracted features to improve the network distinction ability, while in our adversarial learning, the mapped embedding only represents the task-specific features. As shown in Section 5.3, direct critics on the feature space is difficult for the discriminator. Thus, our proposed work involves extra image features for the discriminator, which is method similar to pixel-level domain adaption [15]. Our second approach fetches both feature embeddings and its corresponding domain images into the discriminator (shown in Figure 3.6). This allows the CNN in the discriminator to capture extra features to distinguish between mapped embedding and target embedding, which encourages feature diversity for the discriminator. In addition, the concatenated features allow the discriminator to learn the correlation between the aligned embedding and corresponding domain image. In other words, the feature diversity method is a combined way of pixel-level and feature-level domain adaption.

Learning Algorithm During each iteration, assuming the discriminator parameter as θ, the discriminator loss is defined as:

$$\mathcal{L}_\theta = \mathbb{E}_{z_s \sim \mathbb{P}_s}[D_\theta([Wx, z_s])] - \mathbb{E}_{z_t \sim \mathbb{P}_t}[D_\theta([y, z_t])] + \lambda \mathbb{E}_{z_s \sim \mathbb{P}_{\hat{z}}}[(\|\nabla_{[Wx,z_s]} D_\theta([Wx, z_s])\|_2 - 1)^2] \quad (3.24)$$

where $\mathbb{E}_{z_s \sim \mathbb{P}_{\hat{z}}}[(\|\nabla_{[Wx,z_s]} D_\theta([Wx, z_s])\|_2 - 1)^2]$ denotes gradient penalty term [47] to stabilize the training process, $\mathbb{P}_s$ refers to the source domain data distribution, $\mathbb{P}_t$ refers to the target domain data distribution, and $\mathbb{P}_{\hat{z}}$ implicitly indicates uniform sampling among paired points from $\mathbb{P}_s$ and $\mathbb{P}_t$. $[\cdot]$ represents the concatenation operation.

The mapping function is trained to generate aligned feature, and the loss can be written as:

$$\mathcal{L}_W = -\frac{1}{n} \sum_{i=1}^{n} D_\theta([Wx, z_s]) \quad (3.25)$$

The training process follows the same procedure used in the original GAN [45]. The detailed training procedures and parameters are described in Section 5.3.

Without regularization on the source domain, the inherent structure might collapse in each optimization iteration. To guarantee that we preserve the manifold in original embedding space after every parameter update, we perform orthogonality constraints [89] as secondary update rule through $W \leftarrow (1 + \beta)W - \beta(WW^T)W$, where we set $\beta = 0.01$.

Refinement We also consider the reliable anchor points for fine tuning after adversarial learning. We use the Cross-domain similarity local scaling (CSLS) [28] to find the anchor points. CSLS uses a bi-partite neighborhood graph to define the pairing rules, so that anchor points in the source and target domains are adjusted accordingly. To apply this to the image domain, we simply change the similarity score to:

$$\text{CSLS}(Wx, y) = \frac{2}{\|Wx - y\|_2} - r_T(Wx) - r_S(y) \quad (3.26)$$

where $r_T(Wx)$ is defined as the mean similarity from source embedding x to its target embedding, and it has the form $\frac{1}{K} \sum_{y \in \text{Nei(Wx)}} \frac{1}{\|Wx - y\|_2}$, where K denotes the number of nearest neighbors ($K = 10$ in our experiment); $r_S(y)$ is defined similarly. As we will show in Section 5.3, the refinement may hurt performance when two embeddings have different structures.

3.4 Summary

In this chapter, we discussed GVIN, a differentiable, novel planning module capable of both regular and irregular graph navigation and impressive scale generalization. Based on GVIN architecture, we introduced episodic Q-learning that designed to stabilize the training process of VIN and GVIN. Additionally, we propose a unified framework GEM-F, subsumes LapEigs, DeepWalk and node2vec as special case, providing insights into the asymptotic behaviors of DeepWalk and node2vec. Among our studies, we also developed WarpMap algorithm built upon GEM-F is faster than state-of-the-art and scales linearly with input edges. Finally, we proposed a fusion layer that performs linear mapping for domain transfer combining feature-level and pixel-level transfer. To capture additional semantic information of image, we developed pre-clustering and feature diversity techniques.

Chapter 4

Related Work

In this chapter we review recent related work surrounding the focus of this dissertation. Section 4.1 reviews the studies of RL and planning, the primary focus of the proposed GVIN introduced in this research. We also examine different graph embedding approaches that are influential in this research in Section 4.2, including factorization based- and neural network based-embedding. Section 4.3 highlights domain transfer in RL and control tasks, which is relevant to our domain alignment research. Although there are a large number of related works, here we only show the significant related work that is relevant to this thesis.

4.1 RL and Planning

Model-free RL learns policy from real experience. Standard model-free algorithms include value-based methods, such as Q-learning and SARSA, and policy-based methods, such as TRPO [109], and actor-critic such as A3C [86]. We will briefly discuss model-free methods since this thesis mainly focuses on planning, then we review state-of-the-art planning approaches based on neural networks.

Model-free value-based RL In value-based model-free DRL, the action-value function is parameterized by nonlinear functions with a large number of parameters that are trained with deep neural networks. The policy is then derived from the action-value function (ex: ϵ-greedy algorithm). DQN [87], which has been discussed in Section 2.3.3, introduces the experience replay buffer and target network to stabilize the training process. Deep Deterministic Policy Gradient (DDPG) [80] extends DQN to continuous actions by using an extra policy network, which is trained using the gradient of Q function. One problem in DDPG is that the learned Q function begins to dramatically overestimate the true Q-value. Twin Delayed DDPG (TD3) [41] uses clipped double-Q learning [139]. To improve the training efficiency, [107] enhances the replay buffer to be a priority queue, such that higher priority represents where there is a big difference between the prediction and the TD target. Dueling network [144] separates the estimator of the value function and advantage function since that in many cases the value function is already informative enough to make decisions.

Model-free policy gradient based RL In policy-based model-free DRL, a policy is directly approximated by deep neural networks. Asynchronous Advantage Actor Critic (A3C) [86] proposes an on-policy learning actor critic algorithm that utilizes multiple independent agents to learn simultaneously. A3C has a master network that updates gradients with workers in an asynchronous way. The research community found that asynchronicity does not necessarly lead to improved performance and that Advantage Actor Critic (A2C) without asynchronous is preferred since it is a better computational fit for GPUs. Proximal Policy Optimization (PPO) [110] is a recently developed policy gradient algorithm that minimizes the cost function while constraining the size of a policy update. Compared with Trusted Region Policy Optimization (TRPO), PPO does not require second order derivative information such as the conjugate gradient descent, and is easy to tune and implement.

Value Iteration Network (VIN) Although most recent works focus on the model-free approach, some recent model-based works also show promising results. VIN employs an embedded differentiable planning architecture, trained end-to-end via imitation learning [128]. VIN recovers the underlying MDP by using the standard value iteration algorithm. The Bellman equation is encoded within the convolutional neural network framework, and thereby, the policy can be obtained through backpropagation. VIN demonstrates the ability to diffuse the value function throughout a dynamic model that is position-invariant. However, VIN has some shortcomings: it is limited to regular lattice space, it requires imitation learning for maximum performance, and it is trained separately with a reactive policy.

Predictron A different model-based related work, Predictron, uses a learning and planning model that simulates a Markov reward process [117]. The architecture unrolls the "imagined" plan via a predictron core, where each core (multi-layer convolutional neural network) represents one step in a MDP. Similar to VIN, the predictron also approximates the true value function through end-to-end training; however, Predictron is limited to the tasks of value prediction and is relatively computationally expensive when compared to VIN.

Value Prediction Network (VPN) [94] integrates model-free and model-based RL methods into one architecture. VPN, which extends Predictron [117] to MDP problems, decomposes the Q value into rollout trajectories, and each rollout contains a prediction of the model dynamics (ex: rewards, state transition, and discount factor). To improve the trainability, VPN is trained by an on-policy asynchronous approach and auxiliary loss from model dynamics prediction. VPN is more general than the VIN architecture since it does not assume the local topological structure, such as an image. However, VPN is not capable of addressing continuous actions. Additionally, in many cases the environment contains the topological structure, which VPN does not take advantage of these assumptions.

AlphaGo AlphaGo [115, 116] shows great success on the game of Go, which previously it was thought that a machine could not beat a human player because of the very large search space. AlphaGo combines deep neural networks with Monte-Carlo Tree Search (MCTS) based on pure RL settings. In order to better approximate the value of leaf node states during the tree search, AlphaGo introduces a value network that directly estimates the value of the state in the game of Go. AlphaGo also employs rollout-policy which shares the same idea as VPN [94], I2A [145], Dyna-Q [123], etc.

Universal Planning Networks (UPN) [119] embeds differentiable planning within a goal-directed policy. The planning computation unrolls a forward model in a latent space and infers an optimal action plan through gradient descent trajectory optimization. Unlike the aforementioned differentiable planning methods, which plan the future actions based on rollout Q functions, UPN infers an optimal action plan through gradient descent upon the action variables.

4.2 Graph Embedding

Numerous embedding algorithms have been proposed in data mining, machine learning, and signal processing communities. Our literature review mainly covers graph embedding with manifold, factorization based, and neural network based methods.

Graph embedding with manifold IsoMAP [134], LLE [102], LapEigs [10], and diffusion maps [27] estimate the intrinsic manifold based on the distribution of data points; also, SNE [57], t-SNE [138], and LargeVis [130], which visualize high-dimensional data in a 2D/3D scatter plot, share a similar idea, that is computing a similarity structure of the data points or k-nearest neighbor graph and then projecting them into a low dimensional space with the structure preserved.

Factorization based graph embedding DeepWalk [97], LINE [131], node2vec [46], HOPE [95] and structural deep network embedding [142] learn feature vectors for nodes to handle prediction tasks in large-scale networks. Word2vec [83, 84] learns feature vectors for words in documents based on negative sampling. Instead of using a text corpus composed of sentences in natural languages, DeepWalk [97] first considers the vertices paths traversed by random walks over the network as the sentences for network embeddings, leveraging skip-gram for learning latent representations. Similar to DeepWalk, node2vec [46] incrementally enhances DeepWalk based on 2nd-order random walks. In addition to random walk, LINE [131] proposes to preserve both the 1st-order and the 2nd-order proximity, efficiently trained by using asynchronous stochastic gradient algorithm (ASGD) via edge sampling. However, the aforementioned work is still limited to considering microscopic structures such as 1st-order or 2nd-order proximity, while community-based graph embeddings [143, 149] further preserve macroscopic information. For example, Modularized Nonnegative Matrix Factorization (M-NMF) [143] that considers extra community information jointly optimizes the node representation (microscopic) and an auxiliary community representation matrix (macroscopic).

Deep Learning based graph embedding A number of recent works in deep learning use neural networks to handle signals supported on graphs [91, 37, 53]. The principal idea is to generalize basic operations in the regular domain, such as filtering and pooling, to the graph domain based on spectral graph theory. For example, [20, 53] introduce hierarchical clustering on graphs and the spectrum of the graph Laplacian to neural networks. [29] generalizes classical convolutional neural networks by using graph coarsening and localized convolutional graph filtering. [68] considers semi-supervised learning with graphs by using graph-based convolutional neural networks. [112] generalizes the classical recurrent neural networks by combining graph-based convolutional neural networks and recurrent neural networks to find dynamic patterns. A recent overview is provided in [19]. All of the mentioned related works consider signals supported on a fixed and known graph, consequentially preventing the training parameters to be transferable to other graphs.

4.3 Domain Transfer for visual input

This research is mainly focused on domain transfer in RL settings. Typical domain transfer in RL consists of: (1) policy transfer, meaning that the source domain distinguishes dynamics or actions from the target domain; (2) task transfer, where the source domain differentiates the rewards function from the target domain; and (3) observation transfer, where observations are different between the source domain and target domain. Since policy transfer and task transfer are well studied by meta-RL [100, 52, 36], hierarchy RL [40, 121], etc., our research mainly focuses on observation transfer, which is still highly challenging and thereby we will specifically review the transfer learning on observations.

Pixel-Level Domain Transfer Pixel-Level domain transfer commonly refers to Generative Adversarial Network (GAN)-based models that generate source domain image(s) that look like target domain image(s) [105, 14, 15]. CAR2RL [105] proposed a vision based RL that relies on a high quality simulator that produces realistic rendered images. Another domain transfer work [14] trains a GraspGAN using synthetic data for a grasping task in both pixel-level and feature-level space. Additionally, [15] studies an image transfer by adding extra target loss in the GAN model, in which it decouples the domain adaption process and task process. [137] adapts robotic perception from simulator to real-world environments, but they presume a small dataset that pairs the synthetic with real image from the same scene, which is often False or impossible in real cases. [104] uses layer-wise adapters from progressive networks [103] to transfer the simulator policy to the real world utilizing lateral connections. Most of these works are either task-specific, making it difficult to transfer to other RL problems, or the model relies on high quality synthetic images, which does not scale to different physical world environments. Also, the progressive network and pixel-level domain transfer are generally cumbersome, which requires an extra large model to generate images for the agent during prediction.

Feature-Level Domain Transfer Feature-level domain transfer aims to learn domain-invariant features, either by pre-trained transformation between source domain and target domain [104, 43, 136], or a domain invariant feature extractor [43]. One similar work, Domain Adversarial Neural Network (DANN) [43] uses adversarial learning for domain transfer at the feature level. DANN's approach attempts to make the extracted features from both domains look similar, but the DANN targets image classification whereas the goal of our approach mainly focuses on RL settings. Another work [136] aligns the source domain to the target domain in feature space via pairwise loss and confusion loss. However, the algorithm still requires extra supervision as constraints for feature alignment, while our proposed domain adaption is in a pure unsupervised fashion. [104] proposes a framework to transfer the low level visual features to a new task by enriching new network input features layer-by-layer. [42] proposes a gradient reversal layer in deep architectures for classification tasks. To our knowledge, all of these methods either require paired supervision between synthetic images and real images, or tuning neural network parameters with classification label information. Our proposed methods, which combine the pixel-level and feature-level methods, preserve the pre-trained knowledge as much as possible, only aligning the feature representation by a simple linear transformation without any labels.

Generative Adversarial Network The basic GAN [45] model and concept is covered in Section 2.2.3. In order to obtain higher resolution generated images with stabilized training, Deep Convolutional Generative Adversarial Networks (DCGAN) [99] use pure de-convolution layers in the generator with Adam optimizer [66]. By using curriculum learning, [64] proposed a layer-wise training method to produce high resolution samples through a progressive growing manner, which speeds up the training and improves the training stability. [81] proposes Least Squares Generative Adversarial Networks (LSGAN) which simply tweak the discriminator objective function from binary cross-entropy to be the least mean square, which is equal to minimizing Pearson χ^2 divergence. Recent advanced methods [7] introduce Wasserstein distance, showing that compared with the original Jensen-Shannon Divergence, Wasserstein distance helps to avoid the mode collapse issue and make network training more robust. To improve the Wasserstein GAN trainability, [47] employs the gradient penalty terms to regularize the objective function. Multilingual Unsupervised or Supervised word Embeddings (MUSE) [28] inspired the initial idea of our work: MUSE [28] aims to align word embeddings from two different languages by using adversarial learning, where the generator is a simple linear orthogonal transformation since it has a closed form solution and orthogonal property.

4.4 Summary

In this chapter, we discussed some of the recent developments in RL, graph embedding, and domain transfer. While a large proportion of RL focuses on model-free RL, we have shown that pure model-free RL has poor generalization. In this chapter, we reviewed several recent key studies of model-based RL, planning modules that coupled with model-free approaches, and combined model-based with model-free RL algorithms. Our work can be viewed as embedding a differentiable planning module into the model-free approach. Additionally, we discussed the domain transfer in RL for image based input, which is also studied by the computer vision community. Similar to existing work that uses adversarial learning to tune the target domain network or align the feature representation, our approach is computationally lightweight and modular. The proposed method is a general approach for aligning features across different neural network modules.

In addition to RL studies, we also discussed several graph embedding research efforts. Our research builds upon the factorization based method and summarizes most of previous work as special cases of the proposed more general formalization. The next chapter provides additional details on the performance comparison between previous baseline results and our results.

Chapter 5

Experimental Results

In this chapter, we present our results in three sections that corresponds to Chapter 3: 1) generalized value iteration network, our experiments demonstrates that the agent can plan and generalize to a variety of input structures (regular or irregular) and environment that have never been seen before; 2) warped graph embedding, which we show the proposed algorithm outperforms the baseline model in both computation time and prediction accuracy; and 3) domain alignment for visual based input, we successfully demonstrate the transferred network parameters could be directly used without re-training.

5.1 Generalized Value Iteration Network

In this section, we evaluate the proposed method on three types of graphs: 2D mazes, synthesized irregular graphs and real road networks. We first validate that the proposed GVIN is comparable to the original VIN for 2D mazes, which have regular lattice structure. We next show that the proposed GVIN automatically learns the concepts of direction and distance in synthesized irregular graphs through the reinforcement learning setting (without using any ground-truth labels). Finally, we use the pre-trained GVIN model to plan paths for the Minnesota road network and Manhattan street network.

Our implementation is based on Tensorflow with a GPU-enabled platform. All experiments use the standard centered RMSProp algorithm as the optimizer with learning rate η = 0.001 [135]. All reinforcement learning experiments use a discount of γ = 0.99, RMSProp decay factor of α = 0.999, and exploration rate ϵ annealed linearly from 0.2 to 0.001 over the first 200 epochs.

5.1.1 Revisting 2D Mazes

Given a starting point and a goal location, we consider planning the shortest paths for 2D mazes; see Figure 5.1(a) as an example. We generate $22,467$ 2D mazes (16×16) using the same scripts[1] that VIN used. We use the same configuration as VIN ($6/7$ data for training and $1/7$ data for testing). Here we consider four comparisons: VIN vs. GVIN, action-value based imitating learning vs. state-value based imitating learning, direction-guided GVIN vs. unguided GVIN, and reinforcement learning.

We consider the rules as follows: the agent receives a $+1$ reward when reaching the goal, receives a -1 reward when hitting an obstacle, and each movement gets a -0.01 reward. To preprocess the input data, we use the same two-layer CNN for both VIN and GVIN, where the first layer involves 150 kernels with size 3×3 and the second layer involves a kernel with size 3×3 for output. The transition probability matrix is parameterized by 10 convolution kernels with size 3×3 in both VIN and GVIN. In GVIN, we use the directional kernel based method as shown in Equations 3.6 and 3.7 and we set $\ell = 8$ to represent the eight reference directions. We consider two approaches to initialize the directions θ_ℓ. In the direction-aware approach, we fix θ_ℓ as $[0, \pi/4, \pi/2, ..., 7\pi/4]$. In the direction-unaware approach, we set θ_ℓ to be weights and train them via backpropagation. We set the recurrence K in GVIN to be 20 for 16×16 2D mazes. In the regular domain, we set the kernel order $t = 100$ to be the default.

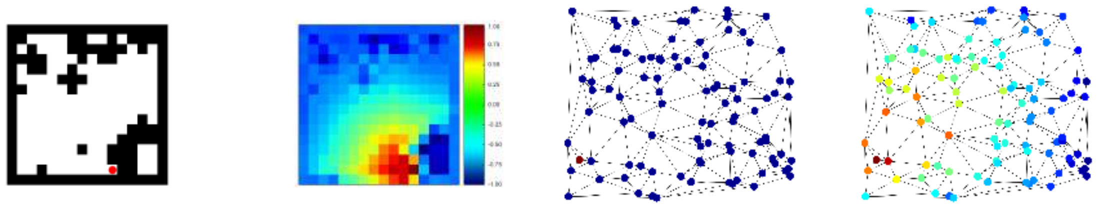
Four metrics are used to quantify the planning performance, including prediction accuracy—the probability of taking the ground-truth action at each state (higher means better); success rate—the probability of successfully arriving at the goal from the start state without hitting any obstacles (higher means better); path difference—the average length difference between the predicted path and the ground-truth path (lower means better); and expected reward—the average accumulated reward (higher means better). The overall testing results are summarized in Table 5.1.

	VIN		**GVIN**			
	Action-value	**State-value**	**Action-value**		**State-value**	
			dir-aware	unaware	dir-aware	unaware
Prediction accuracy	95.00%	95.00%	**95.20%**	92.90%	94.40%	94.80%
Success rate	99.30%	99.78%	**99.91%**	98.60%	99.57%	99.68%
Path difference	0.089	0.010	**0.004**	0.019	0.013	0.015
Expected reward	0.963	0.962	**0.965**	0.939	0.958	0.960

Table 5.1: 2D Maze performance comparison for VIN and GVIN. GVIN achieves similar performance with VIN for 2D mazes (16×16); state-value imitation learning achieves similar performance with action-value imitation learning.

[1]https://github.com/avivt/VIN

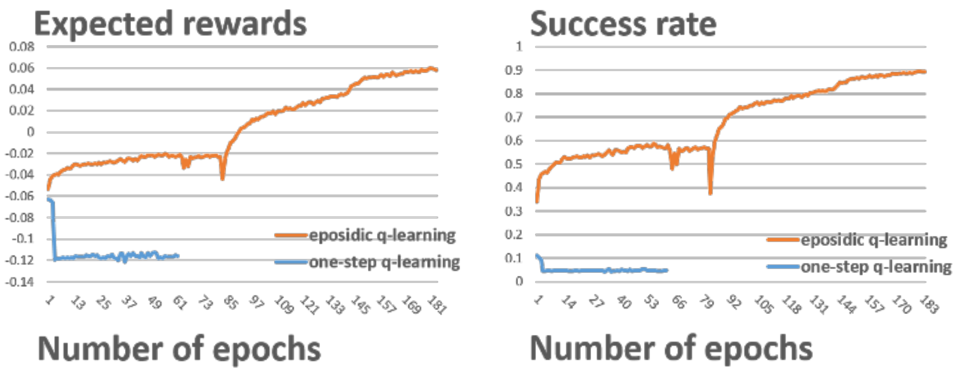
VIN vs. GVIN. GVIN performs competitively with VIN (Table 5.1), especially when GVIN uses direction-aware action-value based imitation learning (4th column in Table 5.1), which outperforms the others for all four metrics. Figure 5.1(b) shows the value map learned from GVIN with direction-unaware state-value based imitation learning. We see negative values (in blue) at obstacles and positive values (in red) around the goal, which is similar to the value map that VIN reported in [128].



(a) Input map (2D mazes). (b) Value map (2D mazes). (c) Input map (irregular graph). (d) Value map (irregular graph).

Figure 5.1: Value map visualization on regular and irregular graph.

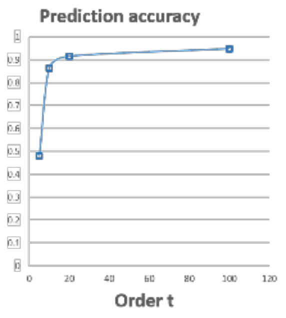
Action-value vs. State-value. VIN with action-value imitation learning slightly outperforms VIN with state-value imitation learning. Similarly, GVIN with action-value based imitation learning slightly outperforms GVIN with state-value based imitation learning. The results suggest that our action approximation method (Section 3.1.1) does not impact the performance while maintaining the ability to be extended to irregular graphs.



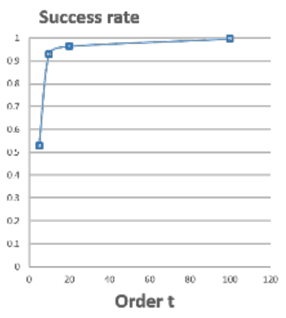
(a) Expected rewards. (b) Success rate.

Figure 5.2: Q- vs. episodic Q-learning on 16×16 Maze.

Direction-aware GVIN vs. Unaware GVIN. Direction-aware GVIN slightly outperforms direction-unaware GVIN, which is reasonable because the fixed eight directions are ground truth for regular 2D mazes.



(a) Prediction accuracy
in 2D mazes.



(b) Success rate
in 2D mazes.

Figure 5.3: Kernel direction order influences the planning performance in regular graphs (images).

It remains encouraging that the GVIN is able to find the ground-truth directions through imitation learning. As shown later, direction-unaware GVIN outperforms direction-aware GVIN in irregular graphs. Figures 5.3 show that the planning performance improves as the kernel exponential t in Equation (3.6) increases due to the resolution in the reference direction being low when t is small. Figure 3.3 compares the kernel with the same reference direction, but two different kernel orders. When $t = 5$, the kernel activates wide-range directions; when $t = 100$, the kernel focuses on a small-range directions and has a higher resolution.

Reinforcement Learning. We also examine the performance of episodic Q-learning (Section 3.1.3) in VIN. Table 5.2 shows that the episodic Q-learning algorithm outperforms the training method used in VIN (TRPO + curriculum learning). For the results reported in Table 5.2, we were able to train the VIN using our algorithm (episodic Q-learning) in just 200 epochs, while TRPO and curriculum learning took 1000 epochs to train VIN, as reported in [128] (both algorithms used the same settings). As shown in Figure 5.2, the episodic Q-learning algorithm shows faster convergence and better overall performance when compared with Q-learning.

5.1.2 Exploring Irregular Graphs

We consider four comparisons in the following experiments: Directional kernel vs. Spatial kernel vs. Embedding-based kernel, direction-aware vs. direction-unaware, scale generalization, and reinforcement learning vs. imitation learning. We use the same performance metrics as the previously discussed in the 2D maze experiments.

We evaluate our proposed methods in Section 3.1.2 for the irregular domain. Our experimental domain is synthetic data that consists of $N = 10000$ irregular graphs, in which each graph contains 100 nodes.

	TRPO	**EQL^2**
Success rate	82.50%	98.67%
No. of Epochs	1000	200
Path difference	N/A	0.1617
Expected reward	N/A	0.9451

Table 5.2: Performance comparison using different training algorithms on the VIN model. The first column is VIN trained by TRPO with curriculum learning reported in [128], the second column is VIN trained by episodic Q-learning.

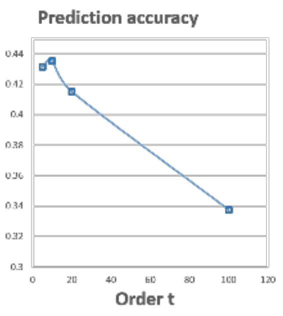
	VIN	**MACN**	**Directional Kernel**		**Spatial Kernel**		**Embedding-based Kernel**		
		(36 nodes)	dir-aware	unaware	dir-aware	unaware	train 100 (IL)	train 10 (IL)	train 10 (RL)
Prediction acc.	26.57%	78%	41.50%	41.51%	57.45%	57.90%	**58.90%**	56.14%	50.90%
Success rate	10.29%	89.4%	34.75%	65.30%	96.56%	97.17%	97.34%	6.73%	**100%**
Path diff.	0.992	-	0.175	0.141	0.082	0.082	**0.079**	0.041	0.148
Expected reward	−0.905	-	0.266	0.599	0.911	0.917	0.922	−0.03	**0.943**

Table 5.3: The performance comparison among VIN and three different kernels of GVIN. All experiments except MACN [65] are tested on 100-node irregular graphs. The last column is trained using episodic Q-learning. IL and RL stands for imitate learning and reinforcement learning, respectively. Under similar experimental settings, MACN achieves an 89.4% success rate for 36-node graphs, while GVIN achieves a 97.34% success rate for 100-node graphs.

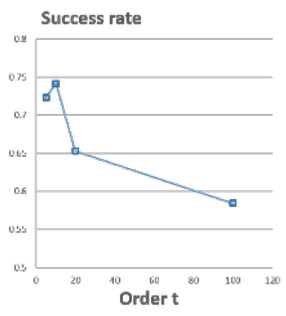
We follow the standard rules of random geometric graphs to generate our irregular graphs. Specifically, we generate $|\mathcal{V}|$ vertices with random coordinates in the box $[0, 1]^2$ and connect a pair of two vertices with an edge when the distance between two vertices is smaller than a certain threshold. For each node in the graph, we define the coordinates that represent its spatial position ranged between 0 and 1. The dataset is split into 7672 graphs for training and 1428 graphs for testing. Additionally, to examine whether GVIN could handle weighted graphs, we also generated a synthetic dataset consisting of 100 node graphs and partitioned 42857 graphs for training and 7143 graphs for testing.

For the directional kernel and spatial kernel, we set the number of reference directions to $\ell = 8$ and kernel order $t = 20$ to be default values for Equations 3.6 and 3.7. We also set θ_ℓ to be 0 to 2π with an interval of $\pi/4$ for *direction-aware* mode and we set θ_ℓ to be trainable weights for *direction-unaware* mode. For the spatial kernel function, we set the number of bins d_ℓ to be 10 in Equation 3.7. In the embedding-based kernel, we use three layers of fully connected neural networks (32-64-1), where each layer uses $ReLU(\cdot) = \max(0, \cdot)$ as its activation function. The neural networks are initialized with zero-mean and 0.01 derivation. For all three kernel methods, we set the graph convolution channel number to be 10 and $K = 40$ for recurrence.

Directional Kernel vs. Spatial Kernel vs. Embedding-based Kernel. We first train the GVIN via imitation learning. Table 5.3 shows that the embedding-based kernel outperforms the other kernel methods in terms of both action prediction and path difference (*Embedding-based Kernel* column in Table 5.3), indicating that the embedding-based kernel captures the edge weight information (distance) within the neural network weights better than the other methods. The spatial kernel demonstrates higher accuracy and success rate when compared with the directional kernel, which suggests the effectiveness of using bin sampling. The *direction-unaware* method shows slightly better results for the spatial kernel, but has a larger success rate gain for the directional kernel. Figure 5.1(d) shows the visualization of the learned value maps which shares similar properties with the regular graph value map. We also train VIN (1st column) by converting the graph to a 2D image.



(a) Prediction accuracy in irregular graphs



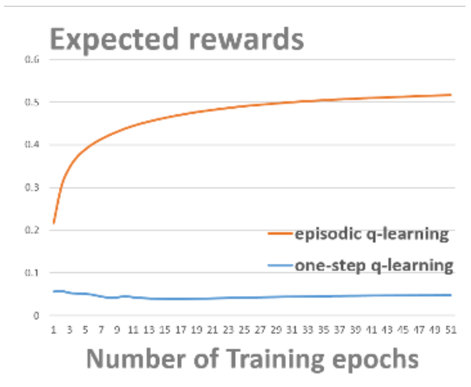
(b) Success rate in irregular graphs.

Figure 5.4: Kernel direction order influences the planning performance in irregular graphs.

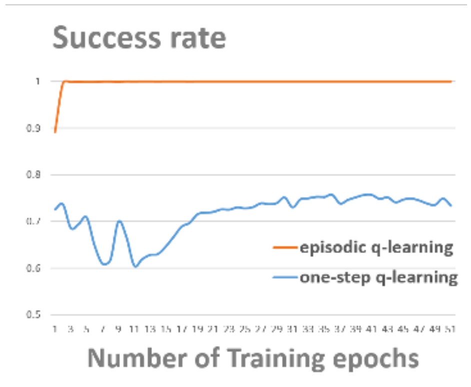
Training VIN on Irregular Graphs. To show the strong generalization of GVIN, we evaluate VIN on irregular graphs by converting the graph data to a 2D image. Each testing set contains the reward map and obstacle map of size 100 × 100 pixels. We use pre-trained weights from 28 × 28 maze with all parameters tuned to be highest performance. To make the training and testing consistent, we set the rewards map and obstacle map the same settings as the 2D maze: vertices and edges are marked as free path (value set to be 0), while the other areas are marked as obstacles (value set to be 1). The edge path is generated via Bresenham's line algorithm [16]. We set recurrence K to be 200 so that the value iteration could cover the whole map.

Figures 5.4(c) and (d) show the planning performance for the irregular domain as the kernel order t in Equation 3.6 increases. The results show that a larger t in the irregular domain has the opposite effect when compared with the regular domain. The observation is reasonable: in the irregular domain, the direction of each neighbor is extremely variable and a larger kernel order creates a narrower direction range (as seen in Figure 3.3), thus resulting in information loss.

Reinforcement Learning. We then train GVIN using episodic Q-learning to compare with imitation



(a) Expected rewards



(b) Success rate

Figure 5.5: Q- vs. Episodic Q-learning on irregular graphs.

learning. As a baseline, we also train GVIN by using standard deep Q-learning techniques, including using an experience replay buffer and a target network. Both networks use the same kernel function (embedding-based kernel) and configurations. Figure 5.5 shows the comparison of the two algorithms' success rate and expected rewards during the training. Clearly, episodic Q-learning converges to both a high success rate and a high expected rewards, but the standard deep Q-learning techniques fail to achieve reasonable results.

Scale Generalization. We also examine scale generalization by training on 10-node graphs and then testing on 100-node graphs using the embedding-based kernel. When GVIN is trained on 10-node graphs via imitation learning, the performance is significantly hindered as shown in Table 5.3 (2nd column in *Embedding-based Kernel*). When GVIN is trained using episodic Q-learning, Table 5.3 (3rd column in *Embedding-based Kernel*) shows excellent generalization abilities that outperform all imitation learning based results for success rate and expected rewards. Compared with imitation learning, we also observe that the performance decreases for path differences and action prediction.

	Imitation Learning		**Reinforcement Learning**	
	w edge weight	**w/o edge weight**	**w edge weight**	**w/o edge weight**
Prediction accuracy	62.24%	43.12%	**50.38%**	50.14%
Success rate	3.00%	97.60%	**100%**	100%
Path difference	0.020	0.763	**0.146**	0.151
Expected reward	−0.612	0.771	**0.943**	0.940

Table 5.4: Performance comparison for testing weighted graphs. Imitation learning is trained on 100-node irregular graphs while reinforcement learning is trained on 10-node irregular graphs.

Graph with Edge Weights. We also test how GVIN handles edge weights. We set the true weighted shortest path to be $\frac{\mathrm{X}_i - \mathrm{X}_j}{W_{ij}}$, where $\mathrm{X}_i - \mathrm{X}_j$ is the distance between two nodes and W_{ij} is the edge weight. As shown in Table 5.4, imitation learning is trained on 100-node graphs, while reinforcement learning is trained on 10-node. We also examine GVIN by excluding edge weights from the input to see if there are any effects on performance. Table 5.4 shows that for reinforcement learning, edge weights slightly help the agent find a more suitable policy; for imitation learning, input edge weights makes GVIN overfitting, causing a significant failure.

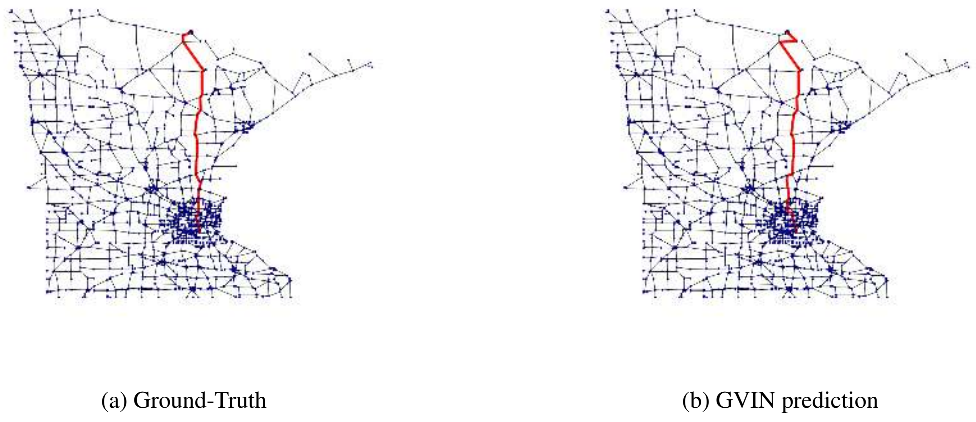

(a) Ground-Truth (b) GVIN prediction

Figure 5.6: Sample planning trajectories on Minnesota highway map.

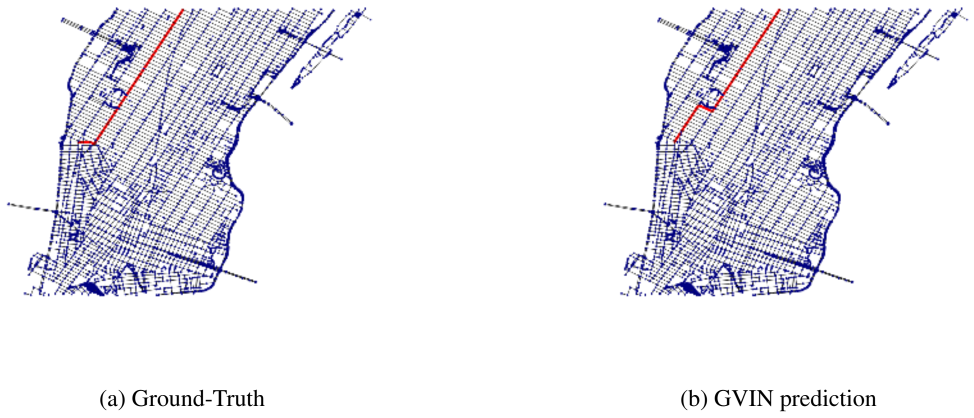

(a) Ground-Truth (b) GVIN prediction

Figure 5.7: Sample planning trajectories on New York City street map.

5.1.3 Validating Real Road Networks

To demonstrate the generalization capabilities of GVIN, we evaluate two real-world maps: the Minnesota highway map, which contains 2642 nodes representing intersections and 6606 edges representing roads, and the New York City street map, which contains 5069 nodes representing intersections and 13368 edges representing roads [24]. We use the same models trained on the graphs containing $|\mathcal{V}| = 100$ and $|\mathcal{V}| = 10$ nodes with the embedding-based kernel and using episodic Q-learning in Section 5.1.2, separately. We normalize the data coordinates between 0 and 1, and we set recurrence parameter to $K = 200$. We randomly pick starting points and goal points 1000 different times. We use the A* [49] algorithm as a baseline. Table 5.5 shows that both $|\mathcal{V}| = 100$ and $|\mathcal{V}| = 10$ generalize well on large scale data. The policy could reach the goal position with 100% in the experiments. One sample planned path is shown in Figures 5.6 and 5.7.

	Minnesota			**New York City**		
	Optimal	$\lvert\mathcal{V}\rvert = 100$	$\lvert\mathcal{V}\rvert = 10$	**Optimal**	$\lvert\mathcal{V}\rvert = 100$	$\lvert\mathcal{V}\rvert = 10$
Prediction Accuracy	100%	78.37%	78.15%	100%	78.66%	79.11%
Success rate	100%	100%	100%	100%	100%	100%
Path difference	0.0000	0.1069	0.1025	0.0000	0.03540	0.0353
Expected reward	0.96043	0.95063	0.95069	0.97279	0.97110	0.97136

Table 5.5: Performance comparison on Minnesota and New York City street map data using GVIN. $|\mathcal{V}| = 100$ is trained on 100-node graphs and $|\mathcal{V}| = 10$ is trained on 10-node graphs.

5.2 Graph Embeddings

In this section, we demonstrate the experiments and results for proposed GEM-F framework and WarpMap algorithm (see Section 3.2).

The main goal here is to empirically demystify how each building block of GEM-F influences the overall performance. We plugin various proximity matrices and warping functions to GEM-F and test their performance. To avoid the randomness and suboptimal solutions, we adopt the warped Frobenius norm and its corresponding SVD solution, which is unique and deterministic.

We focus on the task of node classification: we observe a graph where each node has one or more labels, we randomly sample the labels of a certain fraction (labeling ratio) of nodes and we aim to predict the labels of the remaining nodes. We repeat this process for 20 times and report the average results in terms of Macro and Micro F1 scores [132]. For all of the graph embeddings, we choose dimension size $K = 64$ and implement the classification by using a one-vs-rest logistic regression of LibLinear [39].

5.2.1 Datasets

We use the following datasets:

- `Kaggle` 1968[3]. This is a social network of Facebook users. The labels are the social circles among the users. The network has 277 nodes, $2,321$ edges and 14 labels.

- `BlogCatalog`[4] [132]. This is a network of social relationships provided by bloggers on the BlogCatalog website. The labels are the topic categories provided by the bloggers. The network has $10,312$ nodes, $333,983$ edges and 39 labels.

- `Flickr`[5] [133]. This is a social network of Flickr users who share photos. The labels are the interest groups of users. The network has $80,513$ nodes, $5,899,882$ edges and 195 labels.

- `U.S. Patent 1975-1999`[6] [76]. U.S. patent dataset is maintained by the National Bureau of Economic Research. This is a citation network among the patents granted between 1975 and 1999. The network has $3,774,768$ nodes, $16,518,948$ edges.

5.2.2 Which factor matters most?

We want to investigate the impact of each of three building blocks: proximity matrix, warping function, and loss function. We first consider the construction of the proximity matrix from two aspects.

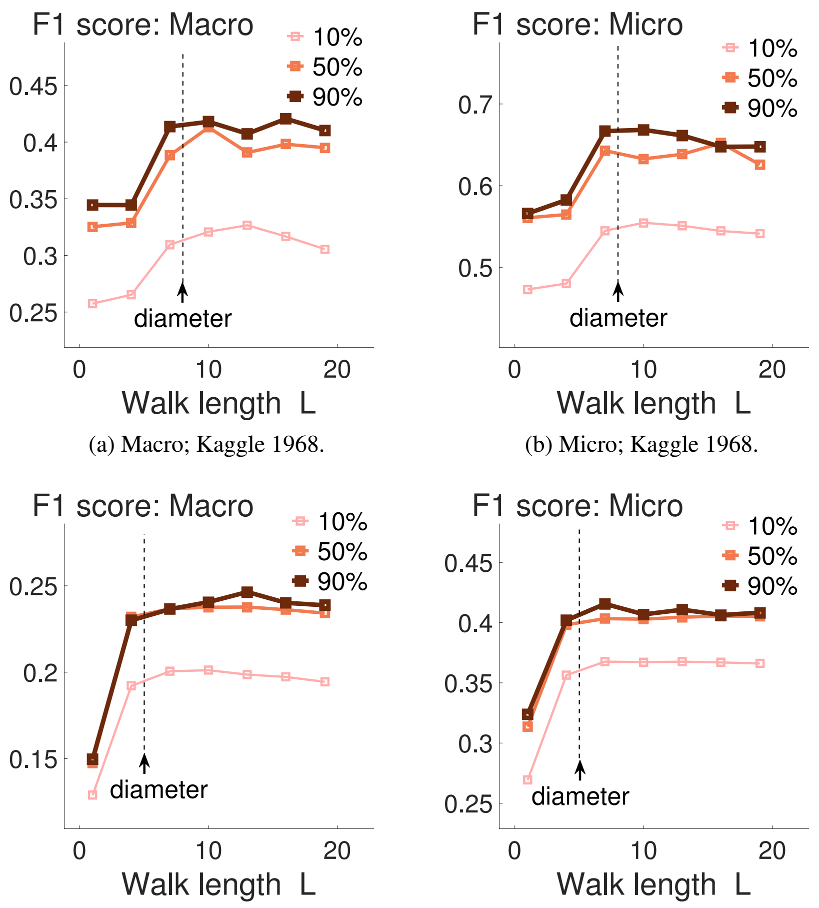
5.2.2.1 Proximity: Walk length

In the FST matrix, a tuning parameter is the walk length, which controls the range of random walks. We consider the closed-form WarpMap with the FST matrix $\Pi^{(L)}$, varying L from 1 to 20. The goal is to understand how the walk length influences the overall performance.

We compare the classification performance on Kaggle 1968 and Blogcatalog [132] in Figure 5.8, where the x-axis is the walk length and the y-axis is the F1 score. In each plot, light, medium and dark red curves show the performance under the labeling ratios of 10%, 50% and 90%, respectively.

We see that the walk length has a significant impact on the classification performance; it increases considerably when the walk length increases in the beginning and then drops slowly when the number of walks

[3] `https://www.kaggle.com/c/learning-social-circles/data`
[4] `http://socialcomputing.asu.edu/datasets/BlogCatalog3`
[5] `http://socialcomputing.asu.edu/datasets/Flickr`
[6] `http://snap.stanford.edu/data/cit-Patents.html`



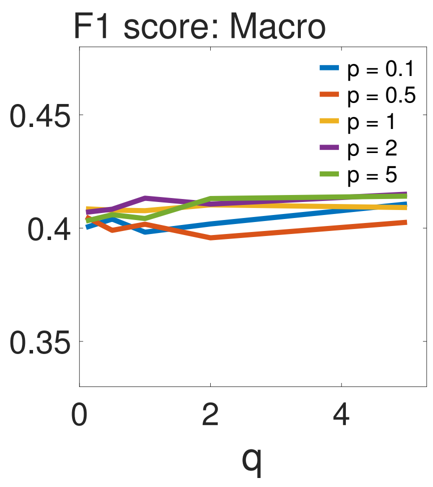
(a) Macro; Kaggle 1968. (b) Micro; Kaggle 1968.

(c) Macro; BlogCatalog. (d) Micro; BlogCatalog.

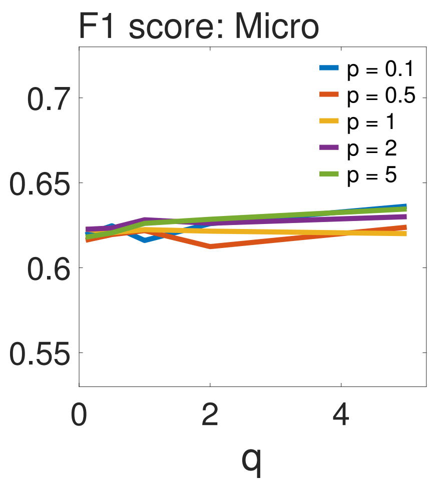
Figure 5.8: The defaulted setting for the walk length is the diameter of input graph. The walk length influences the classification performance. An empirical choice of walk length is the diameter of input graph (Kaggle 1968 and BlogCatalog [132]).

increases even further. The intuition is as follows: when the walk length is too small, only low-order proximity information is obtained and the community-wise information is hard to capture; when the walk length is too large, high-order proximity information is over-exposed. That is, the proximity matrix tends to be an all-one matrix and does not contain any structure information. Empirically, we find that the sweet spot of the walk length is approximately the diameter of a graph, which is usually around 7 for a social network, as shown in Figure 5.8. Note that the diameters of Kaggle 1968 and BlogCatalog are 8 and 5, respectively.

Summary. The walk length matters; the choice should be the diameter of the graph.



(a) Macro at labeling ratio 50%.



(b) Micro at labeling ratio 50%.

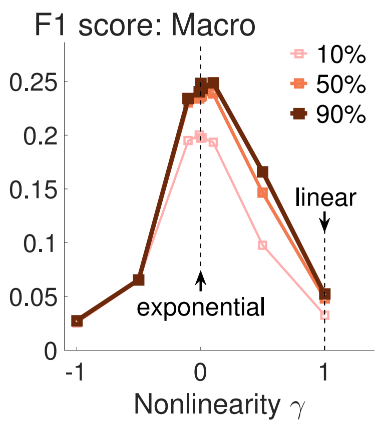
Figure 5.9: Walk memory does not matter. Walk memory factors do not significantly influence the classification performance (Kaggle 1968).

5.2.2.2 Proximity: No walk memory vs. walk memory

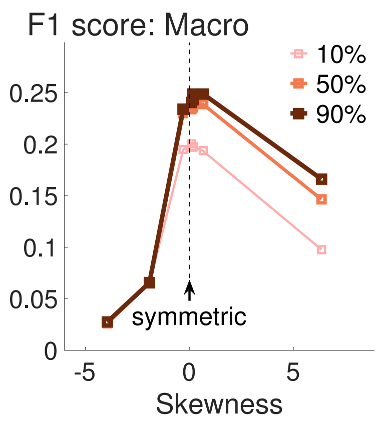
In node2vec, the walk memory term is adopted to design a biased random walk, which explores the graph structure in a controlled manner. The empirical results also show the superiority of node2vec [46] over DeepWalk [97] with appropriate walk memory parameters; however, the comparison suffers from the suboptimality of solving a non-convex optimization problem and randomness of obtaining biased random walks. Here we tune the walk memory parameters in various settings and obtain the corresponding closed-form solutions via singular value decomposition, which help us understand how the walk memory term influences the overall performance. We consider GEM-F with the warping function $g(x) = \exp(x)$ and the FSMT matrix $\Pi^{(L,p,q)}$. We set $L = 7$ and vary p, q on the 2-D grid of $[0.1, 0.5, 1, 2, 5]$.

We compare the classification performances on Kaggle 1968 in Figure 5.9, where the x-axis is q and the y-axis is the F1 score. In each plot, five curves show the performance under various p. We set the labeling ratio to 50%. We see that the walk memory parameters influence the classification performance only slightly: under various settings of p and q, the classification performances do not vary too much. The results show that the walk memory term can provide more flexibility and improves the overall performance, but it is not as significant as the walk length and nonlinearity.

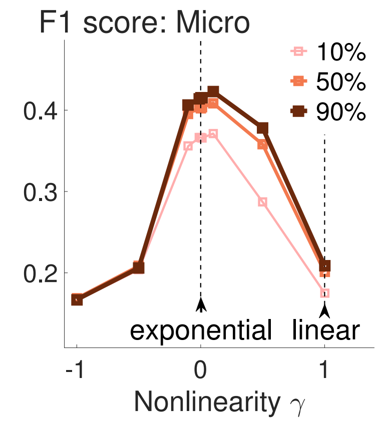
Summary. The walk memory factor does not matter much.



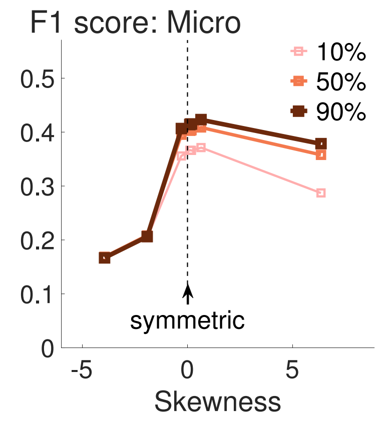
(a) Nonlinearity; Macro.



(b) Skewness; Macro.



(c) Nonlinearity; Micro.



(d) Skewness; Micro.

Figure 5.10: Warping matters. Nonlinearity significantly influences the F1 score (Blogcatalog). The warping function normalizes the distribution of all elements in the proximity matrix; when the distribution is symmetric (skewness is zero), the best performance is achieved.

5.2.2.3 Warping function

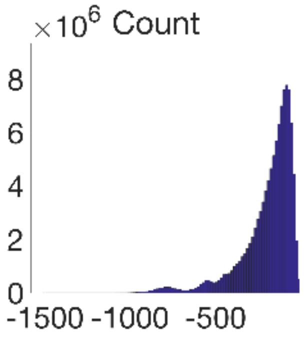
Many previous embedding methods, such as SNE [57], DeepWalk [97] and node2vec [46], adopt the softmax model, which corresponds to an exponential warping function $g(x) = \exp(x)$. How does it differ from a linear warping function $g(x) = x$? We consider GEM-F with the FST matrix $\Pi^{(L)}$ with $L = 7$ and vary the nonlinear warping function $g(x) = (1 + \gamma x)^{\frac{1}{\gamma}}$ from -1 to 1; note that $g(x) = \exp(x)$ when $\gamma = 0$.

We compare the classification performances on Blogcatalog in Figure 5.10 (a) and (c), where the x-axis is the nonlinearity parameter γ and the y-axis is the F1 score. In each plot, light, medium and dark red curves show the performance under the labeling ratios of 10%, 50% and 90%, respectively.

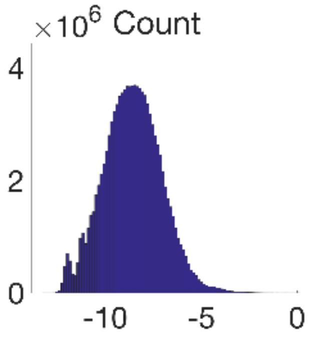
We see that the nonlinearity has a significant impact on the classification performance. A linear function ($\gamma = 1$) results in poor performance. The classification accuracy increases significantly as more nonlinearity is introduced (γ decreases). The exponential function ($\gamma = 0$) almost always provides the best performance. After that, however, adding more nonlinearity deteriorates the classification performance. These results validate the superiority of the (nonlinear) exponential model over the linear model.

How should we then choose the nonlinearity parameter γ? Empirical evidence shows that symmetry matters. Let x be a random variable. We define $\mathrm{skewness} = \mathbf{E}(x-\mu)^3/\sigma^3$, where μ and σ are the mean and standard deviation of x. Figure 5.10 (b) and (d) show the relationship between skewness and classification performance; we see that the classification performance reaches its highest value when the skewness is zero. To illustrate the intuition, Figure 5.11 shows the histogram of elements in the proximity matrix $g^{-1}(\Pi^{(7)})$ under various γ, where x-axis is the value and y-axis counts the number of elements associated with the value; when $\gamma = 0.5$, slight nonlinearity is introduced and, as shown in Figure 5.11 (c), the distribution is highly left-skewed, meaning most pairwise relationships are weak. This is common in social networks due to the power-law phenomenon [38]. The logarithmic factor rescales the distribution to make it symmetric. An empirical choice of the optimal nonlinearity could be to randomly sample a subset of elements in Π and choose the γ that makes the distribution symmetric.

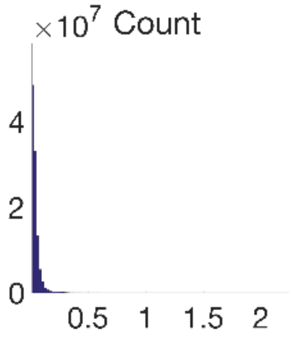
Summary. The warping function matters; the choice should be the one that makes the distribution of elements in the proximity matrix symmetric.



(a) $\gamma = -0.5$, skewness -1.92.

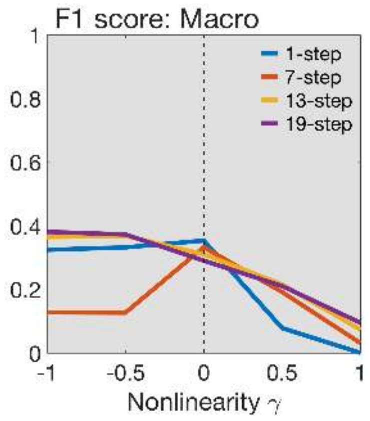


(b) $\gamma = 0$, skewness 0.15.

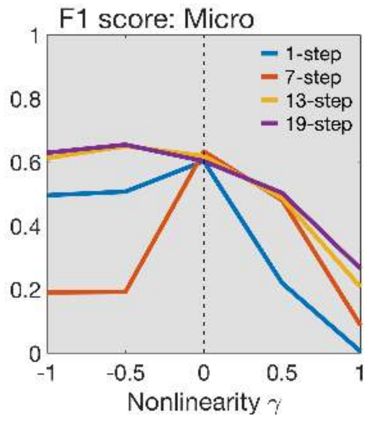


(c) $\gamma = 0.5$, skewness 6.35.

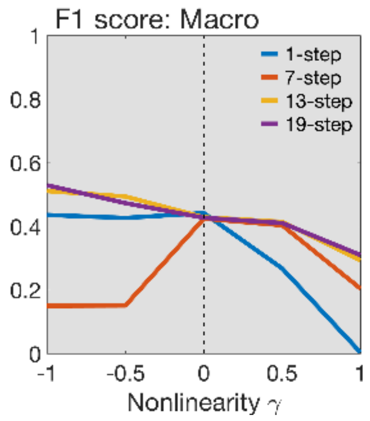
Figure 5.11: Symmetry matters. A symmetric distribution of elements in $g^{-1}(\Pi)$ optimizes the empirical performance (BlogCatalog).



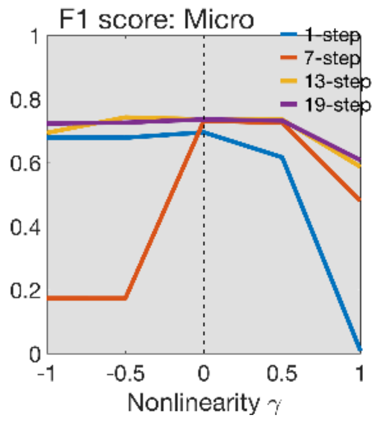
(a) Macro at labeling ratio 10%.



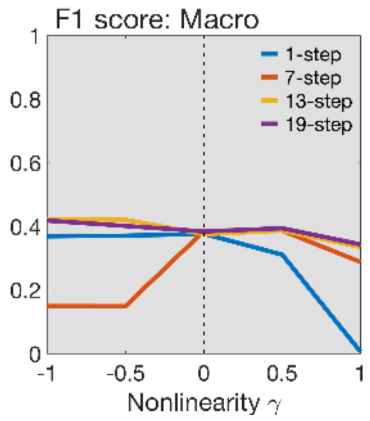
(b) Micro at labeling ratio 10%.



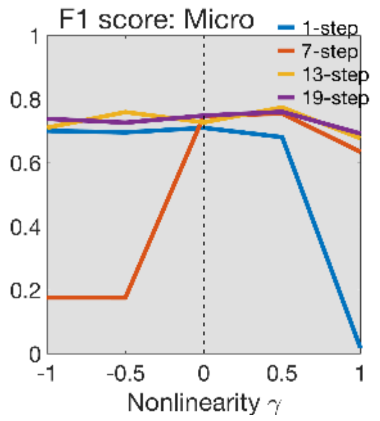
(c) Macro at labeling ratio 50%.



(d) Micro at labeling ratio 50%.



(e) Macro at labeling ratio 90%.



(f) Micro at labeling ratio 90%.

Figure 5.12: In the dataset of Kaggle 1968, the number of steps and nonlinear significantly influence the classification performance.

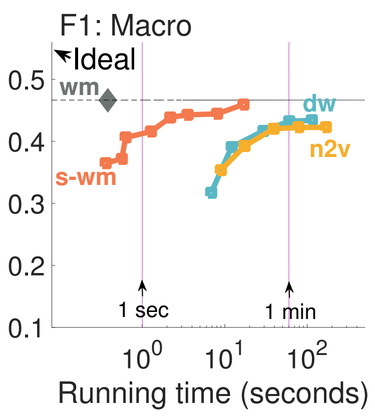
5.2.2.4 Loss function

In Section 3.2.4, we have shown that DeepWalk and node2vec are the implementations of GEM-F with the KL divergence as the loss function. Here we compare DeepWalk and node2vec with WarpMap, whose loss function is the warped Frobenius norm.

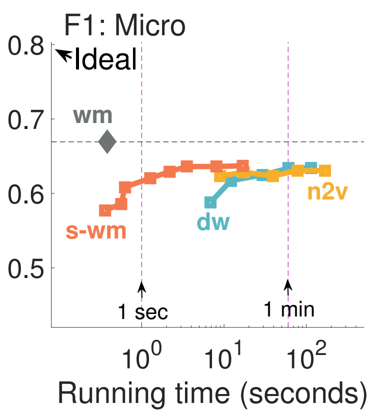
We compare the classification performance of DeepWalk, node2vec, the closed form of WarpMap and scalable WarpMap on three datasets: Kaggle 1968, BlogCatalog and Flickr. For DeepWalk, we set the walk length to be 7; for node2vec, we set the walk length to be 7 and the walk memory factors $p = 0.25, q = 0.25$, as recommended in [46]. We set the labeling ratio to 50%; that is, a half of the nodes are used for training and the rest for testing. The classification performance is shown in Figure 5.13. In each plot, the x-axis is the running time and the y-axis is the F1 score (either Macro or Micro). The blue, yellow and red curves are DeepWalk (dw), node2vec (n2v) and scalable WarpMap (s-uw), which all rise as the running time grows; black diamonds denotes the closed-form of WarpMap (uw).

We see that (1) as the closed-form solution, WarpMap provides the most accurate classification performance; with increasing running time, DeepWalk, node2vec and scalable WarpMap converge to WarpMap, which validates Lemma 1; (2) with increasing size of the input graph, the closed-form WarpMap takes much longer in terms of running time; (3) at a similar level of classification performance, the proposed scalable WarpMap runs faster than DeepWalk and node2vec.

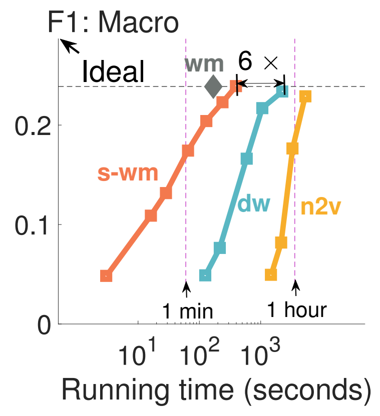
Summary. Randomness does not improve accuracy, but makes the algorithm scalable.



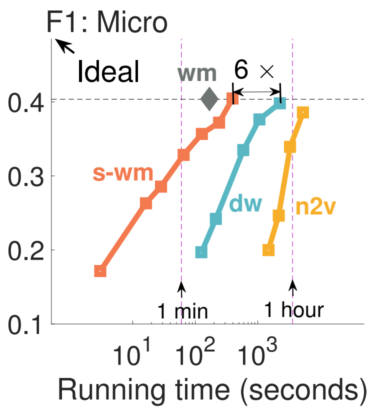
(a) Macro in Kaggle 1968.



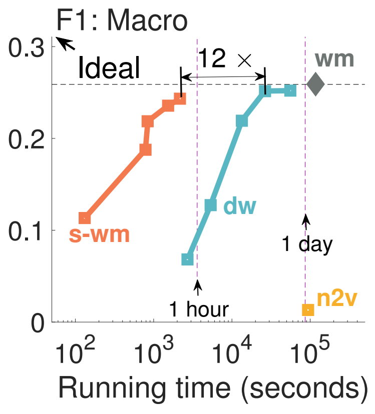
(b) Micro in Kaggle 1968.



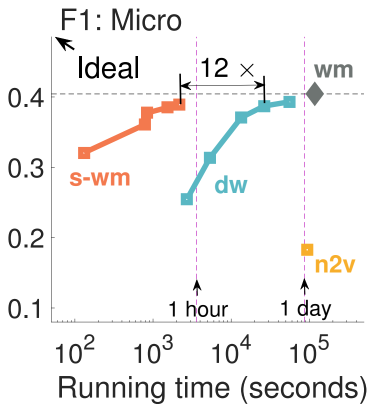
(c) Macro in BlogCatalog.



(d) Micro in BlogCatalog.

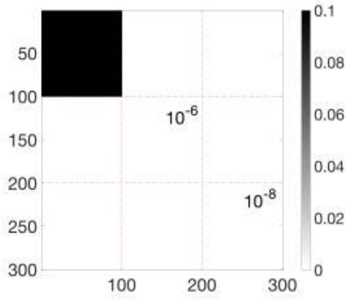


(e) Macro in Flickr.

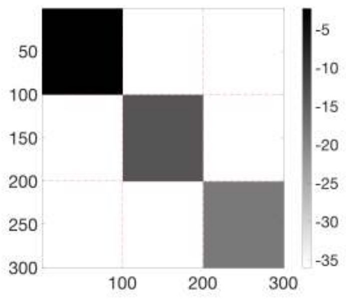


(f) Micro in Flickr.

Figure 5.13: WarpMap outperforms competition. The closed-form WarpMap (uw, grey diamond) is the most accurate; scalable WarpMap (s-uw, red) is much faster than DeepWalk (dw, blue) and node2vec (n2v, yellow).



(a) Π.



(b) $\log \Pi$.

Figure 5.14: The $\log$ factor amplifies small discrepancies. Plot (a) considers a linear warping function $g(x) = g^{-1}(x) = x$; Plot (b) considers an exponential warping function $g(x) = \exp(x)$ and $g^{-1}(x) = \log(x)$.

5.2.3 Why Warping matters?

In WarpMap, we consider factorizing $\log(\Pi)$. The logarithm is introduced to rescale the distribution of Π and amplify small structural discrepancies. On the other hand, when we introduce more nonlinearity than the logarithm, all the values tend to be the same.

Let us see the mechanism of nonlinearity from a toy example. Figure 5.14 (a) shows a 300×300 proximity matrix, which can be divided into three diagonal blocks and indicates three communities. The values of three diagonal blocks are $10^{-1}, 10^{-6} and 10^{-8}$, respectively; however, we cannot distinguish their little difference from the figure. When we factorize this proximity matrix, the resulting embedding barely reflects the structure information. When we adopt the exponential warping function, leading to the inverse warping function $g^{-1}(x) = \log(x)$, Figure 5.14 (b) shows the logarithm version of the same proximity matrix. After a logarithm operation, it is much easier to distinguish their difference. When we factorize this logarithm version, the resulting graph embedding reflects the structure information well.

5.2.4 Scalability

We test scalable WarpMap on the U.S. patent 1975–1999 data. We use the chronological order to construct a growing citation network (as new nodes and edges are added). We compute the corresponding graph embedding by using scalable WarpMap. We set the number of random walks $m = 50$. Figure 5.15 shows its running time (y-axis) versus the number of input edges (x-axis). Red dots depict the graph at different years. The blue dotted line is not a linear fit, but shows the linear trend. Scalable WarpMap scales linearly with the number of input edges and embeds a 14-million edge graph in under 40 minutes.

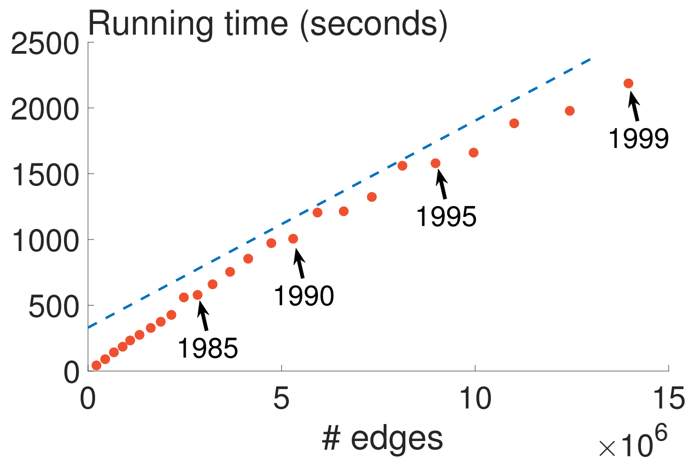


Figure 5.15: WarpMap scales linearly. The proposed algorithm WarpMap scales well, linearly with the number of edges. (U.S. patent 1975–1999 dataset; see Lemma 5.)

5.3 Domain Alignment for Visual Input

In this section, we demonstrate two experiments for visual domain alignment. The first one uses CIFAR-10 and shows our algorithm effectiveness when two domains have very similar embedding structure. The second experiment uses the CARLA autonomous driving environment, in which two domains have mismatched embedding structure.

5.3.1 CIFAR-10 feature alignment

Experiment Setup. In CIFAR-10 experiment, we use two neural networks to generate different features for the same image. Specifically, we use CIFAR-10 image as input, VGG-19 [118] and resnet-18 [51] as two neural networks that are trained from scratch to generate source domain features and target domain features. Both networks are fully trained such that accuracy converges to 91.64% and 94.37%, respectively. We split CIFAR-10 data into $50,000$ images for alignment and 10,000 validation images for model selection. We choose the model based on paired euclidean distance. In both networks, we use the second last layer as the feature layer. We use RMSprop as an optimizer with learning rate $\eta = 0.00005$ for both the mapping function and discriminator. The discriminator is a 3-layer neural network with Leaky RELU [146] as the activation function and dropout [120] to be 0.8 for better generalization.

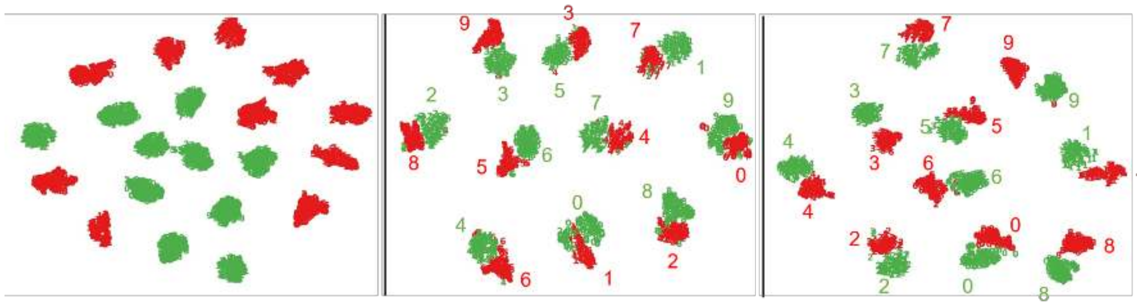


Figure 5.16: t-SNE visualization on CIFAR-10 for different training senarios. the red is from source domain, and the green one comes from target domain. **Left**: the visualization of source domain feature and target domain feature before alignment. **Middle**: the aligned feature embedding use Vanilla AL. We mark the corresponding CIFAR label on each cluster group based on most appeared classes. **Right**: the aligned feature embedding use WAL+CNN.

CIFAR-10 [71] dataset is a typical image recognition dataset for testing baseline models in computer vision. CIFAR-10 consists of $60,000$ 32×32 color images in 10 classes, where each class contains $60,000$ images. Typically, there are $50,000$ training images and $10,000$ test images.

For naming convenience, we denote WAL as the mapping function trained using the Wasserstein GAN settings; we refer to the adversarial learning used in MUSE as Vanilla AL; we denote WAL+PC as pre-clustering methods described in Section 3.3.3 (k is set to be 200); and we refer to WAL+CNN as the feature diversity method described in Section 3.3.3.

The performance comparison is shown in the Table 5.6, where we report evaluation via two metrics. 1) Label similarity is calculated as $\mathbb{1}_{l_t = l_s}$ normalized by the number of samples, where $\mathbb{1}$ is an indicator function, l_t is the label of target domain features, l_s represents the label of $\hat{x}$, where $\hat{x} \in \mathrm{Nei}(Wx)$ is the 1-nearest neighbor of aligned features after the mapping function. 2) We also use euclidean distance as an evaluation metric to measure how close the paired feature embedding is after the mapping function.

Vanilla AL vs. WAL vs. WAL+PC. Table 5.6 shows that WAL is more effective than Vanilla AL by using the Earth-Mover distance as discussed in Section 3.3.3. WAL+PC outperforms both Vanilla AL and WAL. The results suggest that pre-clustering, which relies on intrinsic embedding structure, makes training more effective and robust.

Supervised vs. Unsupervised. Supervised learning outperforms all adversarial learning methods except WAL+CNN on the label similarity. Since supervised learning is solved as a Procrustes problem, the mean l2 norm distance shows the worst result to meet the orthogonality constraints. WAL+CNN outperforms supervised learning in both metrics, indicating that WAL+CNN helps to find a balance between source

Method	Label Simimlarity	l2 distance
Supervised	87.6%	8.09
Vanilla AL	1.71%	7.69
WAL	11.86%	7.55
WAL+PC	64.74%	7.46
WAL+CNN	90.44%	7.36
WAL+CNN+refine	**91.09%**	**7.32**

Table 5.6: The performance comparison on CIFAR-10 using label similarity (higher is better) and L2 distance (lower is better)

embedding structure and closeness of mapping function.

WAL vs. WAL+CNN. Among all results in Table 5.6, WAL+CNN outperforms all other methods significantly, which suggests the effectiveness by using proposed feature diversity method. Figure 5.16 also compared the differences of aligned features by using Vanilla AL and WAL+CNN. As shown in the figure, even though the source embeddings are aligned similar to target embeddings in Vanilla AL, the corresponding labels are almost all mismatched. While in WAL+CNN, the source and target image label are all matched. This result demonstrates that by the means of CNN in the discriminator, WAL+CNN could avoid mode collapse problems.

Refinement In the CIFAR-10 experiment, where two embedding structures are roughly the same, we observe marginal performance boost compared with pure unsupervised learning after using CSLS [28] through anchor points for fine tuning.

CARLA Experiment To demonstrate how feature alignment affects the policy, we evaluate our algorithms in the CARLA [33] environment. CARLA is a simulation environment developed to support training and validation of the autonomous urban driving system. It supplies 14 different kinds of weather and lighting conditions with two driving maps.

Our experiment procedures are as follows: we adopt imitation learning with pre-collected data to train multiple networks in different environments. In CARLA, we use different environment pairs to emulate the source domain and target domain. Then, we benchmark the models by pipelining the representation module from the target domain and policy module from the source domain. The hypothesis is that mismatched feature embeddings from the source domain and target domain will fail the agent behaviors, while adding the trained fusion layer, the agent could recover the task successful rate.

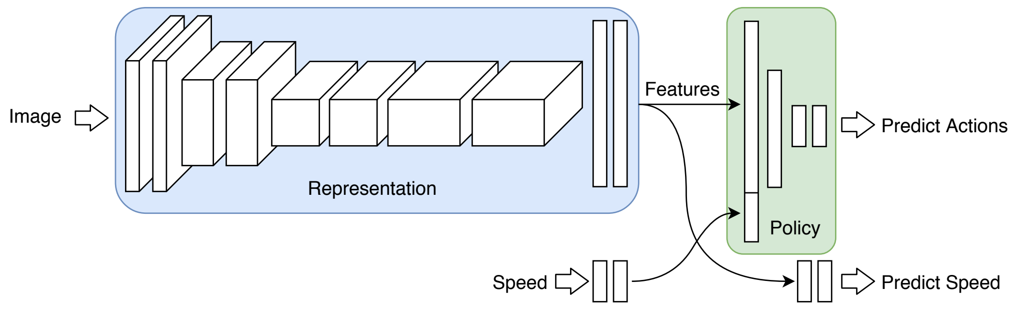


Figure 5.17: New imitation learning network in CARLA. The image module consists of 8 CNN and 2 FCN. The speed feature is generated from 2 FCN. The policy module consists of 3 FCN that process the feature concatenated from image and speed.

5.3.2 CARLA Autonomous Driving

Imitation Learning The original imitation learning on CARLA [26] uses both image-based state and extra commands to derive the driving policy. The extra command is a given inner state used to guide the action. Our experiment aims to verify the policy sensitivity with the environment changing, and thereby we use unconditional imitation learning that uses pure image as state. It turns out that generalization becomes more challenging than original the CARLA experiment since we only use one policy network to predict action without extra commands.

Our network model is shown in Figure 5.17, which includes image representation module and policy module. The input state contains current frame of the observed image and car speed. The representation module consists of 8 convolution network layers and 2 fully connected network layers, with each layer followed by batch normalization [60]. The output of the policy module contains steer, brake, and accelerator, which are all continues variables. Additionally, we feed the policy module with the image module output to get predicted speed, using predicted speed as an auxiliary loss to capture vehicle dynamics. The detailed network architecture and model parameters are shown in Table 5.7:

Data Generation In CARLA simulator, we mix 11 straight trails and 5 one-curve trails from Town01 as one episode. Every trail has a certain start point and end point. We selected two similar kinds of weather as a group and the data was collected with the autopilot mode. We log all of the actions using the currently observed RGB images (we resize the image as 88×200) and current speed to build the datasets. In order to obtain a diverse state, we also applied a random triangular noise on the steer input. We selected three sets of environments for examining domain transfer: sunny noon, rainy noon, and sunset. The samples are shown in Figure 5.18. The agent is trained on sunny noon environment and transfered to rainy and sunset environments.

layer	filter size	strides	input size	output size
Conv	$5 \times 5 \times 32$	2	$88 \times 200 \times 3$	$44 \times 100 \times 32$
Conv	$3 \times 3 \times 32$	1	$44 \times 100 \times 32$	$44 \times 100 \times 32$
Conv	$3 \times 3 \times 64$	2	$44 \times 100 \times 32$	$22 \times 50 \times 64$
Conv	$3 \times 3 \times 64$	1	$22 \times 50 \times 64$	$22 \times 50 \times 64$
Conv	$3 \times 3 \times 128$	2	$22 \times 50 \times 64$	$11 \times 25 \times 128$
Conv	$3 \times 3 \times 128$	1	$11 \times 25 \times 128$	$11 \times 25 \times 128$
Conv	$3 \times 3 \times 256$	1	$11 \times 25 \times 128$	$11 \times 25 \times 256$
Conv	$3 \times 3 \times 256$	1	$11 \times 25 \times 256$	$11 \times 25 \times 256$
FC	512	-	$1,024$	512
FC	512	-	512	512
FC	3	-	512	3

Table 5.7: Model parameters for CARLA experiments

It is a challenging task since the rainy environment consists of many water reflection effects and the sunset contains glare sunshine, and the agent model is not trained on any of these environment conditions.

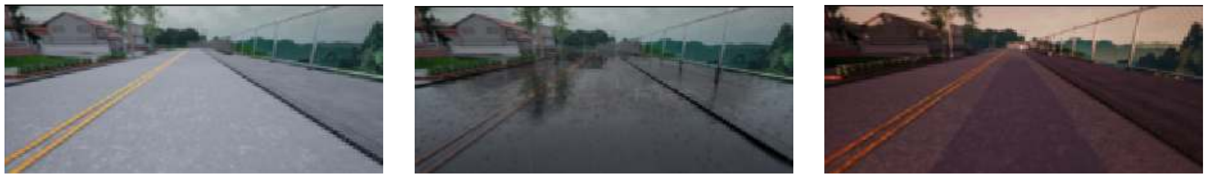

(a) Sunny noon (a) Rainy noon (a) Sunset

Figure 5.18: Environment samples from three different domains in CARLA

Environment Domain Transfer Evaluation We used the sunset environment as the source domain and rainy environment as the target domain. To train the mapping function via adversarial learning for two different domains, we randomly selected 50,000 images and submitted them to the pre-trained network to obtain its corresponding image embeddings. We evaluated different learning algorithms for the mapping function as shown in Table 5.8.

Performance Analysis As seen in the 1st row of Table 5.8, when the representation module and policy module are merged without feature alignment, the agent significantly fails due to the mismatched feature representation. WAL+CNN outperforms all other methods, which suggests that the algorithm is effective when using both embedding structure and extra feature information from the CNN in the discriminator. We also find that the fine-tuning step using CSLS degrades the performance.

In order to compare the performance between supervised learning and adversarial learning, we also use a robot policy to collect image embeddings in two different domains, since the policy of the environment robot does not depend on the visual input. The collected data has one-to-one mapping correspondences, and we build a paired dictionary for supervised learning. The WAL+CNN still outperforms supervised learning in a marginal way.

We observe that the WAL+PC method failed all the evaluation scenarios. To investigate the what causes the WAL+PC to fail, we visualize the t-SNE plot before and after WAL+PC alignment as shown in Figure 5.19. The results show that the feature alignment effectively matches the embedding structure from the two domains, but miss-classifies the semantic meaning of both embeddings. This observation suggests that the extra CNN feature is necessary when two domain embeddings have different structures.

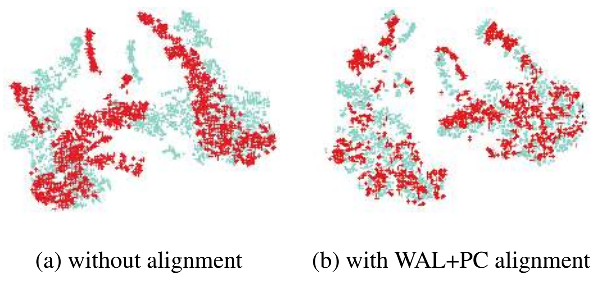


Figure 5.19: The mapping from source domain to taret domain in CARLA.

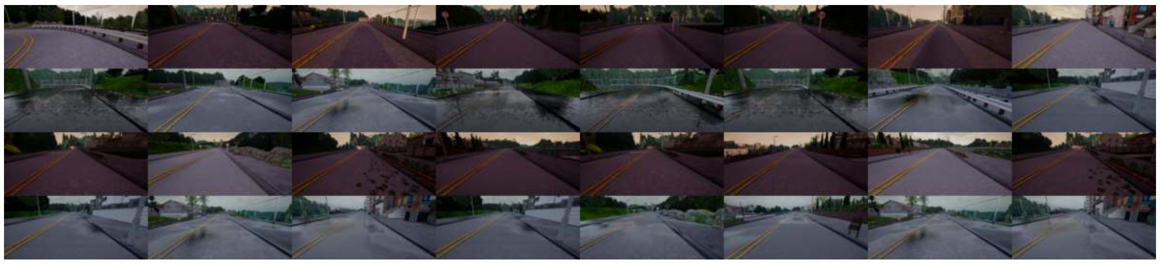


Figure 5.20: Source domain image samples and its corresponding aligned target domain images. The first and third rows are sunset environment images (source domain), the second and fourth rows are nearest neighbor images in the rainy environment (target domain).

To visualize the effectiveness of WAL+CNN, we plot the sample images from the mapped source domain to its target domain as shown in Figure 5.20. The result shows that the mapped images at the target domain is almost the same as the source domain but in different weather conditions, and the result suggest that the trained fusion layer finds the relatively good mapping function that align features from different space.

5.4 Summary

In this chapter, we first demonstrated the experiment results of the GVIN. We show that GVIN generalizes both regular and irregular graph structures, and episodic Q learning stabilizes the training process of GVIN with faster convergence compared with TD learning. Also, we demonstrated good scale generalization of GVIN, that is the training and testing graph has different number of vertices and edges.

Policy	Model	% of success fully reaching the goal	% of distance finishing the trail	% of distance before outside the road
source	without alignment	0.00%	28.71%	6.00%
source	WAL	40.91%	69.58%	57.33%
source	WAL+PC	0.00%	52.82%	6.69%
source	**WAL+CNN**	**54.55%**	**71.43%**	**100%**
source	WAL+CNN+refine	31.82%	63.93%	60.56%
robot	WAL+CNN	45.45%	59.96%	19.99%
robot	supervised	40.91%	53.54%	39.74%
target	without alignment	100%	100%	100%

Table 5.8: Performance Comparison on CARLA Environment

We also presented how design factors of GEM-F affect the classification performance. Our study showed that the warping function significantly influences the overall performance. The walk length in the node proximity function significantly influences the overall performance, and the walk memory factor slightly influences the performance while introducing extra computational cost; see Figure 5.9.

Finally, we evaluate the domain alignment in CIFAR-10 with different networks, our results suggest that when two domain has a similar structure, pre-clustering and feature diversity can both effectively capture the domain mapping relations. In addition, we demonstrate domain alignment in CARLA experiment, our results show that the feature diversity maintains consistent performance while pre-clustering failed, which indicates that adding extra CNN in discriminator is necessary.

Chapter 6

Conclusions and Discussion

In this final chapter, we summarize the dissertation investigations, highlight the important contributions, and provide future research directions. The chapter is organized as follows: Section 1 provides chapter-wise summaries with key dissertation research findings. We then discuss the connection between state-of-the-art GCNN architectures and GVIN, where we highlight the GVIN contributions with its pros and cons. The primary contributions and research outcomes are listed in Section 3. Finally, Section 4 closes with directions for future work and unsolved open questions.

6.1 Dissertation Summary

The research presented in this doctoral manuscript aims to address two fundamental challenges facing the Machine Learning community today: 1) How can we incorporate graph information with features for control problems and classifications; and 2) How to transfer domain features automatically by using light-weight computations. Chapter 1 initializes the research with observations from DRL, listing several DRL limitations and current proposals. Based on these limitations, we summarize that current methods lack a graph representation for the model, and we propose to combine the neural network with graph representation for learning, inference, and planning.

Chapter 2 introduces the background knowledge regarding Machine Learning, DL, generative models, and DRL. In Machine Learning, several fundamentals such as supervised learning, unsupervised learning are discussed, and we show a general Machine Learning perspective that converts learning as an optimization problem. Then we discuss some popular DL architectures used by different data representations. Also, we review the deep generative model and specifically discuss VAE and GAN. Additionally, we go over basic RL concepts and compare its differences with control theory, and three types of model-free RL are listed. Lastly, we review the recent proposed graph convolution.

Chapter 3 demonstrates the main contributions of this thesis: 1) our first work investigates the GVIN architecture based on graph convolution that is capable of both regular and irregular graph navigation and impressive scale generalization. We then develop three novel convolution kernels that are fully differentiable. We employ the Monte-Carlo control, episodic Q-learning, to improve the performance of RL. 2) we design a general graph embedding framework GEM-F that decomposes the graph embedding problem into three functions: warping function, loss function, and node proximity function. Based on GEM-F, we propose the WarpMap algorithm which is up to 6x faster than DeepWalk and Node2Vec, and the algorithm scales linearly. 3) we propose a fusion layer that aligns two domain feature representations into the same space. The fusion layer is trained with adversarial learning, for which we propose two methods to enhance the training procedure.

In Chapter 5, we list three experiment sections that correspond to the sections in Chapter 3. For GVIN, our first experiment uses the same settings as VIN, which is a 2D maze navigation. This experiment shows that GVIN also generalizes for image based inputs. The second experiment demonstrates planning on an irregular graph, which verifies that GVIN can plan on an unseen irregular structure. The final experiment demonstrates the impressive scale generalization, where we train the agent on randomly generated 10-vertices graphs and execute planning on real world street maps. The agent successfully demonstrates good generalization.

In graph embedding, our experiments examine different scale datasets including small scale graphs (277 nodes) to very large graphs ($3,774,768$ nodes). Our experiments verify the effectiveness of each of the design factors, showing that the walk length and non-linearity of the warping function play key roles for overall performance gain (accuracy and computational efficiency).

In domain alignment for transfer learning, our first experiment is tested upon the CIFAR-10 image dataset by aligning two different feature spaces from the VGG-19 and ResNet-18 networks, showing that both feature diversity and pre-clustering enhancement are more robust compared with the original GAN. Our second experiment demonstrates that using domain alignment in a controlled setting with the CARLA simulator, we can successfully transfer the policy between different simulated environments.

6.2 Contributions and Outcomes

With the preceding discussions as a summary, the key objectives addressed by this dissertation research can summarized as:

1. Development of GVIN that involves a graph convolution operator for planning. We show that graph convolution is not only flexible for solving the inference problem, but also could be used for the planning problem.

2. Development of three different graph convolution kernel functions, and demonstrating that the embedding based kernel outperforms other methods marginally, see Table 5.3.

3. Indicating the connection between GVIN and GCNN. We highlight correspondences of each components between GVIN and GCNN, showing that GVIN is a subset configuration of GCNN.

4. Demonstration of the effectiveness of training the planning module episodically, see Figure 5.2

5. Development of the GEM-F framework, where previous embedding algorithms can be viewed each as a special case, see Table 3.1.

6. Development of the WarpMap algorithm based on GEM-F; the algorithm is computationally efficient and scales linearly (Figure 5.15).

7. Theoretical and empirical analysis of the graph embedding design factors. The warping function and walk length in the node proximity function significantly influence the overall performance; the exponential warping function consistently re-scales to the distribution of the proximity matrix to be symmetric and optimizes the performance (see Figure 5.10). The walk memory factor slightly influences the performance while introducing extra computational cost (see Figure 5.9).

8. Development of adversarial learning for domain alignment based on Wasserstein distance; proposing two variations to improve the model robustness and avoid model collapse issues.

9. Demonstration of domain alignment, showing that feature diversity can both effectively capture the semantic meaning of the image, offering significant performance gain (see Table 5.6).

6.3 Further Work

The research presented in this thesis also opens several potential future research topics which are discussed below:

1. *Graph Generative Model* — Even though GCNN shows impressive reasoning and generalization on different tasks, one typical question is, where does the graph come from? In our experimental settings, we assume that the graph is pre-given, such as social network or map information. However, in real world sensor data such as images, audio, text, etc., it is still an open question of how to convert these high-dimensional non-structured datasets to a graph. In PGM, one possible approach is the use of structure learning, which treats the graph structure as a random variable, then the graph structure can be obtained via inference. Nevertheless, structure learning has two drawbacks: 1) the derived graph is commonly meaningless, lacking insights into the information structure; and 2) structure learning does not generalize to different tasks, where each task has its own graph representation. Future work includes a graph generative model that adaptively generates graphs conditioned upon incoming information. Based on the generated graph, the future work should also address the interpretablility of the prediction results.

2. *General Framework for Reinforcement Learning and Inference* — According to the discussion in Subsection 3.1.6, GCNN could be potentially used for both inference and RL tasks. Suggested future work includes proposing a general framework that takes advantage of the differentiable property in neural networks and apply it to diverse applications, e.g., natural language understanding. Furthermore, potential future work should explore combining the GCNN with Bayesian inference, allowing the model to make decisions based on uncertainty instead of point estimation. Finally, future work should also conduct investigations regarding the causality between variables instead of solely conventional statistics correlation, and this could be further extended to generate a casual graph for inference.

3. *Hierarchy Graph Embedding* — Our world knowledge is commonly represented in a hierarchical way, but most graph embedding algorithms miss considerations on hierarchical embedding. One possible method [90] maps the information into hyperbolic space instead of Euclidean space. Hyperbolic geometry can be intuitively viewed as a continues version of the tree structure, but in many cases, knowledge may share properties such that it is not limited to the tree structure. Further study should cover the multi-relation among entities, and the study can be potentially further extended to GCNN for

hierarchy RL and continual learning (also known as the catastrophic forgetting problem).

Bibliography

[1] Convolution neural network. `http://cs231n.github.io/convolutional-networks/`. Accessed: 2018-11-06.

[2] MS Windows understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2018-10-30.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In OSDI, volume 16, pages 265–283, 2016.

[4] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. Data Min. Knowl. Discov., 29(3):626–688, 2015.

[5] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. arXiv preprint arXiv:1512.02595, 2015.

[6] 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. In Advances in Neural Information Processing Systems, pages 5048–5058, 2017.

[7] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. arXiv preprint arXiv:1701.07875, 2017.

[8] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In Proceedings of the Forth International Conference on Web Search and Web Data Mining, WSDM, pages 635–644, Hong Kong, China, 2011.

[9] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261, 2018.

[10] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. Neural Computation, 15(6):1373–1396, 2003.

[11] RICHARD Bellman. Dynamic programming. Princeton, USA: Princeton University Press, 1(2):3, 1957.

[12] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. Dynamic programming and optimal control, volume 1. Athena Scientific Belmont, MA, 1995.

[13] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316, 2016.

[14] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. arXiv preprint arXiv:1709.07857, 2017.

[15] Konstantinos Bousmalis, Nathan Silberman, David Dohan, Dumitru Erhan, and Dilip Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks.

[16] Jack Bresenham. A linear algorithm for incremental digital display of circular arcs. Communications of the ACM, 20(2):100–106, 1977.

[17] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. Computer Networks, 30(1-7):107–117, 1998.

[18] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. arXiv preprint arXiv:1809.11096, 2018.

[19] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. arXiv preprint arXiv:1611.08097, 2016.

[20] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. arXiv preprint arXiv:1312.6203, 2013.

[21] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, pages 891–900. ACM, 2015.

[22] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE transactions on pattern analysis and machine intelligence, 40(4):834–848, 2018.

[23] Siheng Chen, Sufeng Niu, Leman Akoglu, Jelena Kovačević, and Christos Faloutsos. Fast, warped graph embedding: Unifying framework and one-click algorithm. arXiv preprint arXiv:1702.05764, 2017.

[24] Siheng Chen, Yaoqing Yang, Jelena Kovacevic, and Christos Faloutsos. Monitoring manhattan's traffic from 5 cameras?

[25] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555, 2014.

[26] Felipe Codevilla, Matthias Müller, Alexey Dosovitskiy, Antonio López, and Vladlen Koltun. End-to-end driving via conditional imitation learning. arXiv preprint arXiv:1710.02410, 2017.

[27] R. R. Coifman and S. Lafon. Diffusion maps. Appl. Comput. Harmon. Anal., 21:5–30, Jul. 2006.

[28] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. arXiv preprint arXiv:1710.04087, 2017.

[29] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In Advances in Neural Information Processing Systems, pages 3837–3845, 2016.

[30] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In Proceedings of the 28th International Conference on machine learning (ICML-11), pages 465–472, 2011.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pages 248–255. IEEE, 2009.

[32] Carl Doersch. Tutorial on variational autoencoders. arXiv preprint arXiv:1606.05908, 2016.

[33] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In Proceedings of the 1st Annual Conference on Robot Learning, pages 1–16, 2017.

[34] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In Advances in neural information processing systems, pages 1087–1098, 2017.

[35] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In International Conference on Machine Learning, pages 1329–1338, 2016.

[36] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. arXiv preprint arXiv:1611.02779, 2016.

[37] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In Advances in neural information processing systems, pages 2224–2232, 2015.

[38] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In SIGCOMM, pages 251–262, 1999.

[39] R-E Fan, K-W Chang, C-J Hsieh, X-R Wang, and C-J Lin. LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research, 9:1871–1874, 2008.

[40] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. arXiv preprint arXiv:1710.09767, 2017.

[41] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. arXiv preprint arXiv:1802.09477, 2018.

[42] Yaroslav Ganin and Victor Lempitsky. Unsupervised domain adaptation by backpropagation. arXiv preprint arXiv:1409.7495, 2014.

[43] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, Franccois Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. The Journal of Machine Learning Research, 17(1):2096–2030, 2016.

[44] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. arXiv preprint arXiv:1701.00160, 2016.

[45] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Advances in neural information processing systems, pages 2672–2680, 2014.

[46] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 855–864, San Francisco, CA, 2016.

[47] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In Advances in Neural Information Processing Systems, pages 5769–5779, 2017.

[48] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In Advances in neural information processing systems, pages 3338–3346, 2014.

[49] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics, 4(2):100–107, 1968.

[50] T. H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. IEEE Trans. Knowl. Data Eng., 15(4):784–796, 2003.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 770–778, 2016.

[52] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. arXiv preprint arXiv:1512.04455, 2015.

[53] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. arXiv preprint arXiv:1506.05163, 2015.

[54] I. Herman, G. Melancon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. IEEE Trans. Vis. Comput. Graph., 6(1):24–43, 2000.

[55] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In Advances in Neural Information Processing Systems, pages 1693–1701, 2015.

[56] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, et al. Deep q-learning from demonstrations. arXiv preprint arXiv:1704.03732, 2017.

[57] G. E. Hinton and S. T. Roweis. Stochastic neighbor embedding. In Advances in Neural Information Processing Systems NIPS, pages 833–840, Vancouver, British Columbia, 2002.

[58] 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, 29(6):82–97, 2012.

[59] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[60] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

[61] Alex Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[62] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. arXiv preprint arXiv:1611.01144, 2016.

[63] Ken Kansky, Tom Silver, David A Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. arXiv preprint arXiv:1706.04317, 2017.

[64] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196, 2017.

[65] Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar, and Daniel D Lee. Memory augmented control networks. arXiv preprint arXiv:1709.05706, 2017.

[66] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[67] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114, 2013.

[68] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

[69] Daphne Koller and Nir Friedman. Probabilistic graphical models: principles and techniques. MIT press, 2009.

[70] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. IEEE Computer, 42(8):30–37, 2009.

[71] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[73] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. CoRR, abs/1506.07285, 2015.

[74] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In AAAI, pages 2140–2146, 2017.

[75] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series.

[76] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 177–187, Chicago, Illinois, August 2005.

[77] Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. arXiv preprint arXiv:1805.00909, 2018.

[78] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. arXiv preprint arXiv:1504.00702, 2015.

[79] D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. JASIST, 58(7):1019–1031, 2007.

[80] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[81] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In Computer Vision (ICCV), 2017 IEEE International Conference on, pages 2813–2821. IEEE, 2017.

[82] Gary Marcus. Deep learning: A critical appraisal. arXiv preprint arXiv:1801.00631, 2018.

[83] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. CoRR, abs/1301.3781, 2013.

[84] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In Advances in Neural Information Processing Systems, pages 3111–3119, Lake Tahoe, Nevada, Dec. 2013.

[85] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. arXiv preprint arXiv:1309.4168, 2013.

[86] 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.

[87] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[88] 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, 518(7540):529–533, 2015.

[89] Cisse Moustapha, Bojanowski Piotr, Grave Edouard, Dauphin Yann, and Usunier Nicolas. Parseval networks: Improving robustness to adversarial examples. arXiv preprint arXiv:1704.08847, 2017.

[90] Maximillian Nickel and Douwe Kiela. Poincaré embeddings for learning hierarchical representations. In Advances in neural information processing systems, pages 6338–6347, 2017.

[91] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In Proceedings of the 33rd annual international conference on machine learning. ACM, 2016.

[92] Sufeng Niu, Siheng Chen, Hanyu Guo, Colin Targonski, Melissa C Smith, and Jelena Kovačević. Generalized value iteration networks: Life beyond lattices. arXiv preprint arXiv:1706.02416, 2017.

[93] Sufeng Niu, Guangyu Yang, Nilim Sarma, Pengfei Xuan, Melissa C Smith, Pradip Srimani, and Feng Luo. Combining hadoop and gpu to preprocess large affymetrix microarray data. In Big Data (Big Data), 2014 IEEE International Conference on, pages 692–700. IEEE, 2014.

[94] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In Advances in Neural Information Processing Systems, pages 6118–6128, 2017.

[95] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1105–1114, San Francisco, CA, Aug. 2016.

[96] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[97] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, pages 701–710, New York, NY, 2014.

[98] Pedro O Pinheiro, Ronan Collobert, and Piotr Dollar. Learning to segment object candidates. In Advances in Neural Information Processing Systems, pages 1990–1998, 2015.

[99] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434, 2015.

[100] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.

[101] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 91–99, 2015.

[102] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. Science, 290:2323–2326, Dec. 2000.

[103] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. arXiv preprint arXiv:1606.04671, 2016.

[104] Andrei A Rusu, Matej Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. arXiv preprint arXiv:1610.04286, 2016.

[105] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. arXiv preprint arXiv:1611.04201, 2016.

[106] Hacsim Sak, Andrew Senior, Kanishka Rao, and Franccoise Beaufays. Fast and accurate recurrent neural network acoustic models for speech recognition. arXiv preprint arXiv:1507.06947, 2015.

[107] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

[108] Jürgen Schmidhuber. An on-line algorithm for dynamic reinforcement learning and planning in reactive environments. In Neural Networks, 1990., 1990 IJCNN International Joint Conference on, pages 253–258. IEEE, 1990.

[109] 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.

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

[111] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. AI Magazine, 29(3):93–106, 2008.

[112] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. arXiv preprint arXiv:1612.07659, 2016.

[113] Evan Shelhamer, Parsa Mahmoudieh, Max Argus, and Trevor Darrell. Loss is its own reward: Self-supervision for reinforcement learning. arXiv preprint arXiv:1612.07307, 2016.

[114] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. Journal of computer and system sciences, 50(1):132–150, 1995.

[115] 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.

[116] 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, 550(7676):354, 2017.

[117] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, et al. The predictron: End-to-end learning and planning. arXiv preprint arXiv:1612.08810, 2016.

[118] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[119] Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Universal planning networks. arXiv preprint arXiv:1804.00645, 2018.

[120] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1):1929–1958, 2014.

[121] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, et al. Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game. arXiv preprint arXiv:1809.07193, 2018.

[122] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, pages 3104–3112, 2014.

[123] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. ACM SIGART Bulletin, 2(4):160–163, 1991.

[124] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction, volume 1. MIT press Cambridge, 1998.

[125] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2018.

[126] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Advances in neural information processing systems, pages 1057–1063, 2000.

[127] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1–9, 2015.

[128] Aviv Tamar, Sergey Levine, Pieter Abbeel, YI WU, and Garrett Thomas. Value iteration networks. In Advances in Neural Information Processing Systems, pages 2146–2154, 2016.

[129] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. arXiv preprint arXiv:1804.10332, 2018.

[130] J. Tang, J. Liu, M. Zhang, and Q. Mei. Visualizing large-scale and high-dimensional data. In Proceedings of the 25th International Conference on World Wide Web, WWW, pages 287–297, Montreal, Canada, 2016.

[131] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In Proceedings of the 24th International Conference on World Wide Web, pages 1067–1077, Florence, Italy, 2015.

[132] L. Tang and H. Liu. Relational learning via latent social dimensions. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 817–826, Paris, France, 2009.

[133] L. Tang and H. Liu. Scalable learning of collective behavior based on sparse social dimensions. In Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM, pages 1107–1116, Hong Kong, China, 2009.

[134] J. B. Tenenbaum, V. De Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. Science, 290:2319–2323, Dec. 2000.

[135] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2), 2012.

[136] Eric Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Pieter Abbeel, Sergey Levine, Kate Saenko, and Trevor Darrell. Adapting deep visuomotor representations with weak pairwise constraints. arXiv preprint arXiv:1511.07111, 2015.

[137] Eric Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Xingchao Peng, Sergey Levine, Kate Saenko, and Trevor Darrell. Towards adapting deep visuomotor representations from simulated to real environments. CoRR, abs/1511.07111, 2015.

[138] L. van der Maaten and G. E. Hinton. Visualizing high-dimensional data using t-sne. Journal of Machine Learning Research, 9:2579–2605, 2008.

[139] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2016.

[140] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems, pages 5998–6008, 2017.

[141] U. von Luxburg. A tutorial on spectral clustering. Statistics and Computing, 17(4):395–416, 2007.

[142] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1225–1234, San Francisco, CA, 2016.

[143] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang. Community preserving network embedding. In AAAI, San Francisco, CA, 2017.

[144] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581, 2015.

[145] Théophane Weber, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. arXiv preprint arXiv:1707.06203, 2017.

[146] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. arXiv preprint arXiv:1505.00853, 2015.

[147] KiJung Yoon, Renjie Liao, Yuwen Xiong, Lisa Zhang, Ethan Fetaya, Raquel Urtasun, Richard Zemel, and Xaq Pitkow. Inference in probabilistic graphical models by graph neural networks. arXiv preprint arXiv:1803.07710, 2018.

[148] X. Yu, X. Ren, Y. Sun, Q. Gu, B. Sturt, U. Khandelwal, B. Norick, and J. Han. Personalized entity recommendation: a heterogeneous information network approach. In Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, pages 283–292, New York, NY, 2014.

[149] V. W. Zheng, S. Cavallari, H. Cai, K. C-C. Chang, and E. Cambria. From node embedding to community embedding. CoRR, abs/1610.09950, 2016.

[150] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Semantic understanding of scenes through the ade20k dataset. arXiv preprint arXiv:1608.05442, 2016.

[151] X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. In Proc. ICML, pages 912–919, 2003.