Graph Classification using Structural Attention

John Boaz Lee
Worcester Polytechnic Institute
Massachusetts, USA
jtlee@wpi.edu

Ryan Rossi
Adobe Research
California, USA
rrossi@adobe.com

Xiangnan Kong
Worcester Polytechnic Institute
Massachusetts, USA
xkong@wpi.edu

ABSTRACT

Graph classification is a problem with practical applications in many different domains. To solve this problem, one usually calculates certain graph statistics (*i.e.*, graph features) that help discriminate between graphs of different classes. When calculating such features, most existing approaches process the entire graph. In a graphlet-based approach, for instance, the entire graph is processed to get the total count of different graphlets or subgraphs. In many real-world applications, however, graphs can be noisy with discriminative patterns confined to certain regions in the graph only. In this work, we study the problem of *attention-based graph classification*. The use of attention allows us to focus on small but informative parts of the graph, avoiding noise in the rest of the graph. We present a novel RNN model, called the *Graph Attention Model* (GAM), that processes only a portion of the graph by adaptively selecting a sequence of "informative" nodes. Experimental results on multiple real-world datasets show that the proposed method is competitive against various well-known methods in graph classification even though our method is limited to only a portion of the graph.

CCS CONCEPTS

• **Information systems** → **Data mining**; • **Mathematics of computing** → *Graph algorithms*; • **Theory of computation** → *Reinforcement learning*;

KEYWORDS

Attentional processing, graph mining, reinforcement learning, deep learning

ACM Reference Format:
John Boaz Lee, Ryan Rossi, and Xiangnan Kong. 2018. Graph Classification using Structural Attention. In *KDD 2018: 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3219819.3219980

1 INTRODUCTION

Graph classification, or the problem of identifying the class labels of graphs in a dataset, is an important problem with practical applications in a diverse set of fields. Data from bioinformatics [5], chemoinformatics [11], social network analysis [2], urban computing [3], and cyber-security [6] can all be naturally represented as labeled graphs. In chemoinformatics, for instance, molecules can be represented as graphs where nodes correspond to atoms, and edges signify the presence of chemical bonds between pairs of atoms. The task then is to predict the class label of each graph – for instance, the anti-cancer activity, solubility, or toxicity of a molecule.

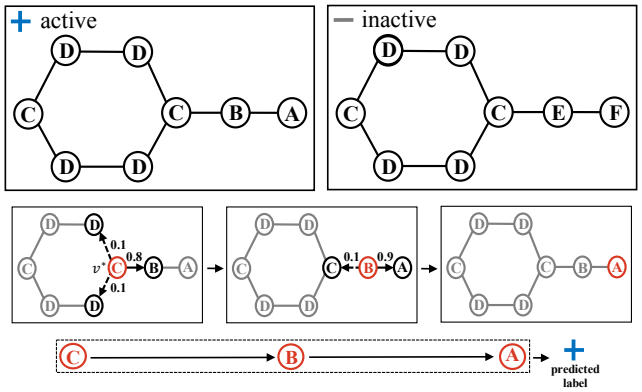

Figure 1: Attention-based graph classification. Given a starting node v^* and a budget $T = 3$ nodes to select for graph classification, attention is used to guide the walk towards more informative parts of the graph.

To solve this problem, the usual strategy is to calculate certain graph statistics (*i.e.*, graph features) on the entire graph. A popular technique is the graphlet kernel [26] which counts the occurrences of various graphlets (*i.e.*, subgraphs) on a graph. Graphs that share a lot of common graphlets are then considered similar. The Morgan algorithm [23] is another method for calculating graph features. It uses an iterative process which updates each node's attribute vector by hashing a concatenation of all the attributes in the node's local neighborhood. The graph feature is then computed from the final attributes of all the nodes in the graph. In more recent years, the focus has shifted towards learning data-driven graph features [11, 19] which is to say task-relevant features are learned automatically from the graphs in a given dataset. Since we can expect graphs belonging to a particular class to exhibit some common pattern that is not typically observed among the other graphs, we can then use the calculated graph features for classification.

However, graphs in the real-world can be both large and noisy [21, 35]; this introduces some challenges when the entire graph has to be processed to calculate graph features. When a graph is noisy, the significant subgraph patterns can be sparse and confined only to small neighborhoods within the graph. For instance, when studying the interaction networks of complex diseases, researchers have identified specific subnetworks that are associated with the disease [8]. In this case, processing the entire graph can inadvertently cause

KDD 2018, August 19–23, 2018, London, United Kingdom

ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3219980

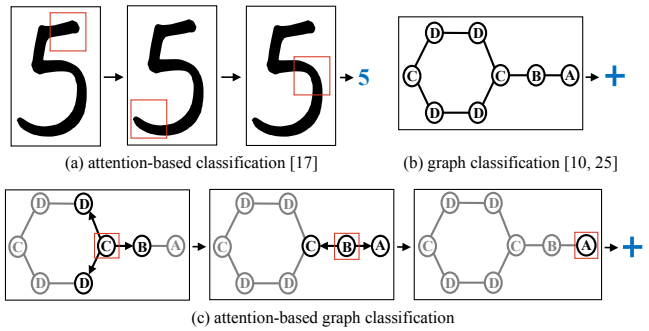

Figure 2: Comparison of different classification problems. Traditional glimpse-based attentional processing in (a) takes several glimpses of the input before classification. On the other hand, graph classification without attention in (b) takes the entire input graph and uses it for classification. The problem we study in (c) uses attention to process a small but relevant part of the graph for classification while obeying graph structure.

noise to be introduced into the calculated feature as most of the graph does not contain anything informative. Furthermore, it is usually costly if not infeasible, to compute representations of large real-world graphs [24].

To address the issues mentioned above, we study the problem of *attention-based graph classification* which we formally define below. We begin by giving the definition of traditional graph classification.

Definition 1. *Graph Classification*: Given a set of attributed graphs $\mathcal{D} = \{(\mathcal{G}_1, \ell_1), (\mathcal{G}_2, \ell_2), \cdots, (\mathcal{G}_n, \ell_n)\}$, the goal is to learn a function $f : \mathbb{G} \rightarrow \mathcal{L}$, where $\mathbb{G}$ is the input space of graphs and $\mathcal{L}$ is the set of graph labels. Here each graph $\mathcal{G}_i = (\mathbf{A}_{\mathcal{G}_i}, \mathbf{D}_{\mathcal{G}_i})$ is comprised of an adjacency matrix $\mathbf{A}_{\mathcal{G}_i} \in \{0, 1\}^{N_i \times N_i}$, and an attribute matrix $\mathbf{D}_{\mathcal{G}_i} \in \mathbb{R}^{N_i \times D}$, where N_i is the number of nodes in the i-th graph and D is the number of attributes. Each graph also has a corresponding label ℓ_i.

Definition 2. *Attention-based Graph Classification*: Given a set of attributed graphs $\mathcal{D} = \{(\mathcal{G}_1, \ell_1), (\mathcal{G}_2, \ell_2), \cdots, (\mathcal{G}_n, \ell_n)\}$, and a budget T, the goal is to learn a composite function $f' \circ g : \mathbb{G} \rightarrow \mathcal{L}$. Here, $\mathcal{G}_i$, $\mathbb{G}$, and $\mathcal{L}$ still retain the same definition while $\circ$ is the function composition operator. The function $g : \mathbb{G} \rightarrow \mathbb{R}^{T \times D}$ selects T nodes and returns their corresponding attributes. In other words, given a graph $\mathcal{G}_i$, we are limited to selecting T nodes and can only use the corresponding information (*i.e.* node attributes) to make a prediction on the graph label.

We summarize the two main challenges of this problem as follows: (1) graph structure is important; and (2) graphs can be noisy with task-relevant patterns confined to small regions within the graph. A useful model has to take these two factors into consideration which makes the problem challenging. For instance, we cannot just choose a random set of nodes since this does not capture the graph's structure and is unlikely to yield task-relevant information.

To address these challenges, we propose a solution based on attention-guided walks. Instead of choosing T random nodes, we use a walk on the graph to sample T nodes. A walk allows us to capture the structure of the traversed region. On the other hand, an attention mechanism is used to guide the walk. This allows us to focus on relevant parts of the graph while ignoring the noise in the rest of the graph which results in graph features that provide better predictive performance. Furthermore, our attention-guided walk is designed to rely only on local information from the graph which has the added benefit of keeping computation costs (space in particular, in this case) low since there is no need to load the entire graph into memory. We provide an illustration of this in Figure 1. On the other hand, Figure 2 highlights the difference between attention-based graph classification and the two related problems of (1) graph classification and (2) attentional processing on non-graph data.

Inspired by the recent success of Recurrent Neural Networks (RNN) with attention on vision-related tasks [18], we propose an RNN model with a built-in attention mechanism for graphs. The attention mechanism in our model is trained using reinforcement learning to actively select informative regions in the graph to process. We also introduce a memory component which allows the model to integrate information gathered from different parts of a graph. The main contributions of our paper can be summarized as follows:

(1) We propose to study the problem of attention-based graph classification and introduce GAM, a general framework that uses attentional processing to learn data-driven features for graphs. The model uses attention to selectively process informative portions of the graph. To the best of our knowledge, this is the first model that uses attention to learn representations for general attributed graphs.
(2) We introduce a model that does not require global information of the graph. Instead, the attention mechanism processes only a node's local neighborhood. In addition, the model is easily parallelizable.
(3) We show through empirical evaluation on multiple real-world datasets that the model can outperform various established baselines.
(4) We demonstrate the effectiveness of attention by comparing against modified baselines that utilize random attention.

The rest of the paper is organized as follows. We start by reviewing related work in the next section. We then present the proposed method in section 3. Experimental setup and discussion of results are provided in section 4. Finally, we conclude the paper and give some directions for future work in the last section.

2 RELATED WORK

Many different techniques have been proposed to solve the graph classification problem. One popular approach is to use a graph kernel to measure similarity between different graphs [20]. This similarity can be measured by considering various structural properties like the shortest paths between nodes [4], the occurrence of certain graphlets or subgraphs [26], and even the structure of the graph at different scales [15].

Recently, several new methods which generalize over previous approaches, have been introduced. These methods use a deep learning framework to learn data-driven representations of graphs [11, 19, 34]. In [11], a method is introduced that generalizes the Weisfeiler-Lehman (WL) algorithm by learning to encode only relevant features from a node's neighborhood during each iteration.

Interestingly, [19] proposes a method that processes a section of the input graph using a convolutional neural network. However, for this to work for graphs of arbitrary sizes the method relies on a labeling step that ranks all the nodes in the graph which means it still processes the entire graph initially.

One thing common among all the above-mentioned approaches is that the entire graph is processed to compute the graph's final representation. In contrast, we study a model that only processes a portion of the graph with attention used to determine the parts of the graph to focus on.

Recent studies have shown that deep learning frameworks with attentional processing can perform well on a variety of tasks. In [17], attention was used to allow the model to attend to a subset of the source words in the language translation task. Meanwhile, [33] used attention to help a model fix its gaze on salient objects for image captioning and [18] applied attention to the image classification task. The work in [7], on the other hand, used attention to guide a CNN to focus on relevant objects for the visual question answering task. Although attentional processing has been applied successfully to many problems, most of the existing work lie in the computer vision or natural language processing domains. In this work, we focus on graphs, which have less well-behaved structure when compared to images/videos (grids) or text (sequences). Because of this, traditional models for attention cannot be directly applied to graph data.

A few work have also begun to investigate the use of attention on graphs [9, 28]. Choi et al. studied a model which used attentional processing on medical ontology graphs [9]. However, our work is significantly different from that of [9] as their model is specifically designed for medical ontologies and work only on directed acyclic graphs (DAG) while we explore an attention mechanism on more general (un)directed attributed graphs. In another work, attentional processing was used to solve the problem of node representation learning [28] which aims to learn embeddings for nodes in a graph. This is quite different from the task of learning representations for graphs – and not nodes – in a dataset which is the problem studied here. To the best of our knowledge, this is the first work that utilizes attention to learn data-driven features for graphs.

Finally, we also experiment with an architecture that has a simple external memory to allow multiple agents to integrate information from various parts of the graph. In a sense, this is conceptually similar to the memory networks of [22, 27].

3 GRAPH ATTENTION MODEL

To simplify the discussion, we begin by describing a basic attention model. In subsequent discussion, we introduce a variant with more refined attention and external memory.

In this work, we formulate the problem of applying attention on graph-structured data as a decision process of a goal-directed agent traversing along an input attributed graph. The agent starts at a random node on the graph and, at each time step, moves to a neighboring node. The information available to the agent is limited to the node it chooses to explore. Since global information about the graph is unavailable, the agent needs to integrate information over time to help it determine the parts of the graph to explore further. The ultimate goal of the agent is to collect enough information that will allow it to make a correct prediction on the label of the graph.

The agent will only explore a small portion of the graph with the attention mechanism guiding it in its exploration. If the graph is large, we can also initialize multiple agents at different nodes in the graph and run them in parallel. Deploying multiple agents can help improve the performance of the model since each agent can explore a different part of the graph with attention helping to steer each agent's exploration along the local neighborhood. This allows us to use the model on large graphs that may be difficult or impossible to load into memory.

3.1 Proposed Model

Our proposed model has an RNN at its core, as shown in Figure 3. At each time step, the core network processes new information from the step that was just taken and integrates this into its internal representation together with information retained from previous steps. It uses this information to predict the label of the input graph and to decide which areas of the graph to prioritize for further exploration in the next time step.

Step module: At each time step, the step module considers the one-hop neighborhood of the current node c_{t-1} and picks a neighbor c_t to take a step towards. The step module is biased towards picking neighbors whose types or labels have higher rankings in the rank vector $\mathbf{r}_{t-1}$. The attribute vector of the chosen node is then extracted and fed together with $\mathbf{r}_{t-1}$ to produce the step representation $\mathbf{s}_t = f_s(\mathbf{d}_{c_t}, \mathbf{r}_{t-1}; \theta_s)$ (see Figure 3a). The step representation $\mathbf{s}_t$ is the new information available to the core LSTM network at each time step. The step algorithm is summarized in Algorithm 1.

Node type: The way we label or assign types to nodes allows us to bias the exploration towards certain nodes at different stages of the exploration. Depending on the application, the node type can be a simple discrete value (*e.g.*, type of atom in a molecular graph) or it can be something more elaborate like a category derived from log-binning several attributes that capture the local structure of the node [1]. We give a simple example of the latter case. Suppose the agent wants to visit one of two Carbon nodes adjacent to it, it cannot differentiate between the nodes under the first node typing strategy. In the second method, the node type may be calculated based on the statistics encoded in the k-hop neighborhood of each node and this allows us to differentiate between the two Carbon nodes. Using more complex node typing strategies [1] may, however, increase the number of node types substantially and one may have to look into reinforcement learning strategies that work well when the discrete action space is large [10].

History: The core LSTM network maintains a history vector which is a summary of all the information obtained by the agent in its exploration of the graph thus far. At each time step, as new information becomes available in the form of $\mathbf{s}_t$ from the step we just took, the history vector is updated via $\mathbf{h}_t = f_h(\mathbf{s}_t, \mathbf{h}_{t-1}; \theta_h)$. This allows the core network to integrate information over time.

We use an LSTM in our architecture as it is superior to simple RNNs in capturing long-range dependencies. Even though LSTMs have a more sophisticated memory model when compared to simple RNNs, it has been shown that they still have trouble remembering information that was inputted too far in the past [30]. Because of

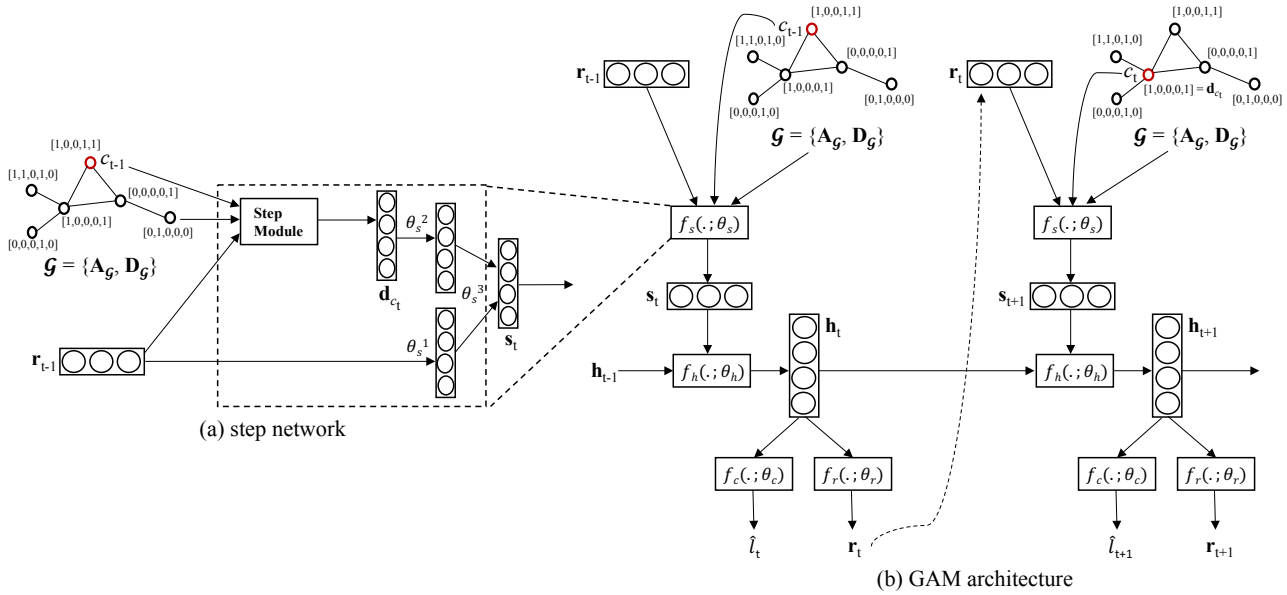

Figure 3: (a) Step network: Given a labeled graph $\mathcal{G}$ (composed of the adjacency matrix $\mathrm{A}_{\mathcal{G}}$, and the attribute matrix $\mathrm{D}_{\mathcal{G}}$), a current node c_{t-1}, and a stochastic rank vector r_{t-1}, the step module takes a step from the current node c_{t-1} to one of its neighbors c_t, prioritizing those whose type (*i.e.*, node label) have higher rank in r_{t-1}. The attribute vector of c_t, d_{c_t}, is extracted and mapped to a hidden space using the linear layer parameterized by θ_s^2. Similarly, r_{t-1} is mapped using another linear layer parameterized by θ_s^1. Information from these two sources are then combined using a linear layer parameterized by θ_s^3 to produce s_t, or the step embedding vector which represents information captured from the current step we took. (b) GAM architecture: We use an RNN as the core component of the model; in particular, we use the Long Short-Term Memory (LSTM) variant [12]. At each time step, the core network $f_h(.;\theta_h)$ takes the step embedding s_t and the internal representation of the model's history from the previous step h_{t-1} as input, and produces the new history vector h_t. The history vector h_t can be thought of as a representation or summary of the information we've aggregated from our exploration of the graph thus far. The rank network $f_r(.;\theta_r)$ uses this to decide which types of nodes are more "interesting" and should thus be prioritized in future exploration. Likewise, the classification network $f_c(.;\theta_c)$ uses h_t to make a prediction on the graph label.

this, on large graphs, it may be better to deploy multiple agents with each agent exploring a relatively small neighborhood rather than having one agent traverse the graph for a long period. To integrate information, we can augment the architecture with a shared external memory [27]. Additionally, a network conditioned on the current history vector can be trained to allow the model to selectively save information to memory. This will allow the model to store information that is useful for graph classification (*e.g.*, discriminative subgraphs).

Actions: Given the new history vector that captures what the agent has seen so far, the agent performs two actions at each time step. First, it predicts the label of the input graph $\hat{l}_t = \arg\max_i P(y = i|f_c(\mathbf{h}_t;\theta_c))$ from the softmax output of the classification network conditioned on $\mathbf{h}_t$. Second, it uses the rank network to generate the rank vector $\mathbf{r}_t = f_r(\mathbf{h}_t;\theta_r)$ that will help "steer" exploration in the next step by ranking the importance of different types of nodes.

Primarily, the rank vector's job is to encode the importance of different types of nodes. However, we can augment it to include additional actions such as one for deciding when to stop further exploration if the agent is confident it has enough information to classify the graph correctly. Another possible action is the one that allows the model to transfer its current internal information to a memory component.

1: **procedure** STEP($\mathbf{r}_{t-1} \in \mathbb{R}^R, \mathbf{A} \in \mathbb{N}^{N\times N}, \mathbf{D} \in \mathbb{R}^{N\times D}, c_{t-1}$)
2: $\quad\mathbf{a} \leftarrow \mathbf{A}[c_{t-1}, :]$
3: $\quad\mathbf{T} \leftarrow \tau(\mathbf{D})$ ▷ $\mathbf{T} \in \mathbb{R}^{N\times R}$ is a matrix of one-hot row vectors indicating node types; we assume that type can be derived from node attributes.
4: $\quad\mathbf{p} \leftarrow (\mathbf{T}\,\mathbf{r}_{t-1})^\top$
5: $\quad\mathbf{p} \leftarrow \mathbf{p} \odot \mathbf{a}$
6: $\quad d \leftarrow \sum_i p_i$
7: $\quad\mathbf{p} \leftarrow \mathbf{p} \odot \frac{1}{d}$
8: $\quad c_t \sim \text{Multinomial}(\pi = \mathbf{p})$ ▷ Sample a neighbor from multinomial distribution parameterized by $\mathbf{p}$.
9: $\quad$**return** $\mathbf{D}[c_t, :], c_t$
10: **end procedure**

Algorithm 1: Procedure to pick a neighbor to move to. The algorithm is biased towards picking neighbors whose types have higher ranks in r_{t-1}. Here, $\odot$ represents element-wise multiplication and ▷ denotes the start of a comment.

Reward: In the typical reinforcement learning setting, the agent receives new information x_{t+1} from the environment and a reward signal r_{t+1} after taking an action at each time step t. The goal of the agent is to maximize the reward it receives which is usually quite sparse and delayed: $R = \sum_{t=1}^{T} r_t$. In our setting, $\mathbf{x}_{t+1} = \mathbf{d}_{c_{t+1}}$

and the reward is given only at the end, where $r_T = 1$ if the model classified the graph correctly and $r_T = -1$ otherwise. Hence $R = r_T$.

Under this formulation, we have what can be considered a Partially Observable Markov Decision Process (POMDP). In this setting, we only obtain partial information about the graph or our environment through our interactions with it at each time step. As in [18], our goal is to learn a policy $\pi((\mathbf{r}_t, \hat{l}_t | s_{1:t}; \theta))$ with parameters θ that maps the sequence of our past interactions with the environment $s_{1:t} = \mathbf{x}_1, \mathbf{r}_1, \hat{l}_1, \cdots, \mathbf{x}_{t-1}, \mathbf{r}_{t-1}, \hat{l}_{t-1}, \mathbf{x}_t$ to a distribution over actions for the current time step t. In other words, given the history of past interactions as summarized in the history vector $\mathbf{h}_t$, the classification network $f_c(.; \theta_c)$ and the rank network $f_r(.; \theta_r)$ – or our policy networks – learn to generate actions that maximize reward.

3.2 Training

Together, the core LSTM network, the step network, and the rank network work in conjunction with each other to form the policy of the agent. We learn the parameters $\theta = \{\theta_h, \theta_s, \theta_r\}$ of these networks to maximize the total reward the agent can expect to obtain. Since each specific policy for the agent induces a distribution over the possible interaction sequences $s_{1:T}$, we want to train our policy to maximize the reward under the generated distribution: $J(\theta) = \mathbb{E}_{P(s_{1:T};\theta)}[R]$.

It is a non-trivial task to maximize J exactly as we are dealing with a very large, and possibly infinite, number of possible interaction sequences. However, since we frame the problem as a POMDP, we are able to obtain a sample approximation of the gradient of J by using the technique introduced by [32] as shown in [18]. This is given by

$$\nabla_\theta J \approx \frac{1}{M} \sum_{i=1}^{M} \sum_{t=1}^{T-1} \nabla_\theta \log \pi(\mathbf{r}_t^i[\tau(c_{t+1}^i)] | s_{1:t}^i; \theta) \gamma^{T-t} R^i \qquad (1)$$

where the s^i's are the interaction sequences from running the agent under the current policy for $i = 1, \cdots, M$ episodes, $\gamma \in (0, 1]$ is a discount factor that allows us to attribute more significance to actions performed closer to time T or when the prediction was made, and $\tau(c_{t+1}^i)$ is a function that maps a node to its type. The intuition behind equation 1, which is also known as the REINFORCE rule, is as follows. We run the agent with the current policy to obtain samples of interaction sequences. The parameters θ are then adjusted to increase the log-probability or rank of the type of nodes that were frequently selected during episodes that resulted in a correct prediction. Training the policy this way allows us to increase the chance that the agent will choose to take a step towards a particular type of node the next time it finds itself in a similar state. To compute $\nabla_\theta \log \pi(\mathbf{r}_t^i[\tau(c_{t+1}^i)] | s_{1:t}^i; \theta)$, we simply compute the gradient of our network at each time step, this can be done using standard backpropagation [31]. Note that we only adjust the log-probabilities for $t = 1, \cdots, T-1$ since the rank vector $\mathbf{r}_t$ in the last step is no longer used.

Since the gradient estimate in Equation 1 may exhibit high variance, one may choose to estimate $\nabla_\theta J$ via

$$\frac{1}{M} \sum_{i=1}^{M} \sum_{t=1}^{T-1} \nabla_\theta \log \pi(\mathbf{r}_t^i[\tau(c_{t+1}^i)] | s_{1:t}^i; \theta)(\gamma^{T-t} R^i - b_t^i) \qquad (2)$$

instead. This provides us with an estimate that is equal in expectation to the original formulation but with possibly lower variance [18]. Here $b_t^i = f_b(s_{1:t}^i; \theta_b) = f_b(h_t^i; \theta_b)$ captures the cumulative reward we can expect to receive for a state h_t^i. The term $(\gamma^{T-t} R^i - b_t^i)$, or the advantage of choosing an action, allows us to increase the log-probability of actions that resulted in a much larger expected cumulative reward and to decrease the log-probability of actions that resulted in the reverse. We can train the parameter θ_b of f_b by reducing the mean squared error of $R^i - b_t^i$.

Finally, we use cross entropy loss to train the classification network $f_c(.; \theta_c)$ by maximizing $\log \pi(l_T | s_{1:T}; \theta_c)$, where l_T is the true label of the input graph $\mathcal{G}$. As in [18], we use this hybrid loss formulation where the rank network f_r is trained at each time step using REINFORCE and the classification network f_c and the baseline network f_b are trained using the classical approach from supervised learning.

3.3 Space Complexity

Let $\triangle_{\mathcal{G}}$ be the max node degree for graph $\mathcal{G}$ and D be the dimension of the node attribute vector. Since the agent only moves to one of the current node's neighbors at each time step, we only need to store a $\triangle_{\mathcal{G}} \times D$ matrix containing the attributes of neighboring nodes at any given time. After taking a step to a new node, the attribute matrix for the new set of neighbors can be fetched from disk. Ignoring the space needed to store $\mathbf{r}$, $\mathbf{s}$, $\mathbf{h}$, c, and the parameters of our model, which are constant and negligible, our model has a space complexity of $O(\triangle_{\mathcal{G}} D)$ which is quite small in practice.

3.4 Initialization

For each new instance, we initialize the start vertex c_0 by selecting a random node in the input graph and the rank vector $\mathbf{r}_0$ is initialized to the uniform distribution.

3.5 Attention with Memory

When predicting the label of an input graph, one may choose to average the softmax output of several runs by initializing multiple agents at different starting locations in the graph. In this case, we can view each agent as one classifier in an ensemble where we predict by voting. While averaging the predictions of several agents can certainly improve classification performance, our model is still at a disadvantage against methods that integrate information from the entire graph. This is because each agent makes a prediction independently, using only the information it gathered from a local area within the graph.

To remedy this, we introduce a variant of our model with a shared external memory component that can store information from multiple agents. In this architecture, each agent i for $i = 1, \cdots, n$ stores information in a local memory component $\mathbf{p}_i$, these are then combined to form the shared memory $\mathbf{m}$ that the classification network uses to make a single prediction. In the simplest case, $\mathbf{p}_i = \mathbf{h}_T^i$, which means we use the final history vector as each agent's local memory. However, not all parts of an agent's walk through the graph may yield equally important information. To allow the model to retain only information useful to the task we set $\mathbf{p}_i = \sum_{j=1}^{T} u_j^i \mathbf{h}_j^i$, where the u_j^i's are the softmaxed output of $f_u(h_j^i; \theta_u)$ which decides

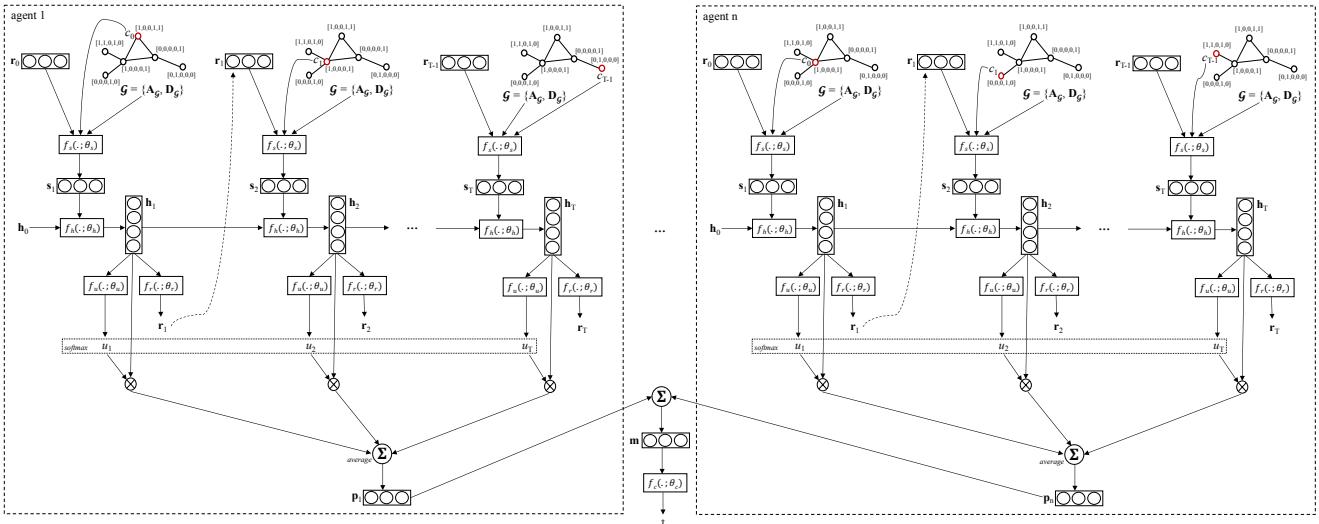

Figure 4: The attention model with memory. Multiple agents can be initialized (in parallel) at different starting nodes allowing each agent to explore a different part of the graph. We use weighted pooling to combine each agent's history vectors, giving us an agent's local memory. The outputs of $f_u(.;\theta_u)$, or the u_i's, are normalized using softmax and multiplied with their corresponding history vectors. The intuition here is to train $f_u(.;\theta_u)$ to assign more importance to useful information. To integrate information, we average the local memories of all the agents to get the shared memory: $\mathbf{m} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{p}_i$ – which is used to make a prediction on the graph label. For brevity, we omit the superscripts for $\mathbf{h}$, $\mathbf{u}$, $\mathbf{r}$, $\mathbf{s}$, and c.

how useful a particular "piece of memory" is. In other words, we do weighted pooling to obtain our local memory. This can be viewed as another form of attention. Finally, to integrate information from multiple agents, we simply set $\mathbf{m} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{p}_i$. This modification allows us to integrate information from various regions in the graph and is especially helpful if the graph is large and we only take a small number of steps T. Note that each agent's exploration is still guided by the attention mechanism proposed earlier. The architecture of attention with memory is shown in Figure 4.

Various modifications can be made to this architecture. For instance, we can choose to condition the output of the rank network on the local memory or even the shared external memory. Additional actions can also be introduced to allow the model to modify or rewrite the shared memory; also, it isn't difficult to imagine including an action that allows the agent to stop exploration if it has already gained enough information to make a prediction. In this work, however, we deliberately choose to test on the simplest version because our goal is to show the usefulness of attention.

4 EXPERIMENTS

4.1 Motivating Example

Before we consider the details of our main experimental setup, we introduce a simple motivating example that shows how attention can be used to guide an agent towards more relevant regions in the graph. For this toy example, we generated a small dataset of random graphs. We embedded several patterns or subgraphs in the generated graphs, two of which were the 3-paths $A-B-C-D$, and $A-B-E-D$. The former pattern was embedded primarily onto positive samples while the latter was included in negative samples. In Figure 5, we show the output of the rank network, over time, when it is given the history vector $\mathbf{h_1}$ capturing the initial step onto the node of type B. It is interesting to note that, initially, the rank network assigns more or less equal importance to the five types of nodes. However, after some time, it learns to prioritize the nodes of types C or E. This guarantees that the agent will prioritize exploration in the right direction, giving the model enough information to classify the graphs correctly in a small number of steps.

4.2 Experimental Setup

4.2.1 Data. We evaluated our proposed method on the binary classification task using five real-world graph datasets: HIV, NCI-1, NCI-33, NCI-83, and NCI-123 [16]. These datasets have been made publicly available by the National Cancer Institute (NCI)[1] [34]. Since the molecular structures in the datasets were encoded using the SMILES format [29], we used the RDKit[2] package to convert each string into its corresponding graph. We used the same package to extract the following information for each node (*i*.e. atom) to use as node attributes: atom element, node degree, total number of attached hydrogens, the implicit valence, and atom aromaticity. Atom element was used to label or assign types to the nodes. The graph class labels indicate the anti-cancer property (active or negative) of each molecule. The datasets are all highly imbalanced with far more negative samples than positive ones. We follow the methodology in previous work [16, 34] and randomly extract a balanced subset (500) for each dataset.

4.2.2 Compared Methods. In order to demonstrate the effectiveness of our proposed approach, we compare it against several baseline methods, all of which utilize the entire graph for feature extraction. To the best of our knowledge, this is the first work on attention with graphs so we compare against baselines that observe the entire graph. We would like to emphasize that our proposed

[1] https://www.cancer.gov/
[2] http://www.rdkit.org/

Table 1: Summary of experimental results: "average accuracy ± SD (rank)". The "ave. rank" column shows the average rank of each method. The lower the average rank, the better the overall performance of the method.

method	dataset					ave. rank
	HIV	NCI-1	NCI-33	NCI-83	NCI-123	
Agg-Attr	69.58 ± 0.03 (4)	64.79 ± 0.04 (4)	61.25 ± 0.03 (6)	58.75 ± 0.05 (6)	60.00 ± 0.02 (6)	5.2
Agg-WL	69.37 ± 0.03 (6)	62.71 ± 0.04 (6)	67.08 ± 0.04 (5)	60.62 ± 0.02 (4)	62.08 ± 0.03 (5)	5.2
Kernel-SP	69.58 ± 0.04 (4)	65.83 ± 0.05 (3)	71.46 ± 0.03 (1)	60.42 ± 0.04 (5)	62.92 ± 0.07 (4)	3.4
Kernel-Gr	71.88 ± 0.05 (3)	67.71 ± 0.06 (1)	69.17 ± 0.03 (3)	66.04 ± 0.03 (3)	65.21 ± 0.05 (2)	2.4
GAM	74.79 ± 0.02 (2)	64.17 ± 0.05 (5)	67.29 ± 0.02 (4)	67.71 ± 0.03 (2)	64.79 ± 0.02 (3)	3.2
GAM-mem	78.54 ± 0.04 (1)	67.71 ± 0.04 (1)	69.58 ± 0.02 (2)	70.42 ± 0.03 (1)	67.08 ± 0.03 (1)	1.2

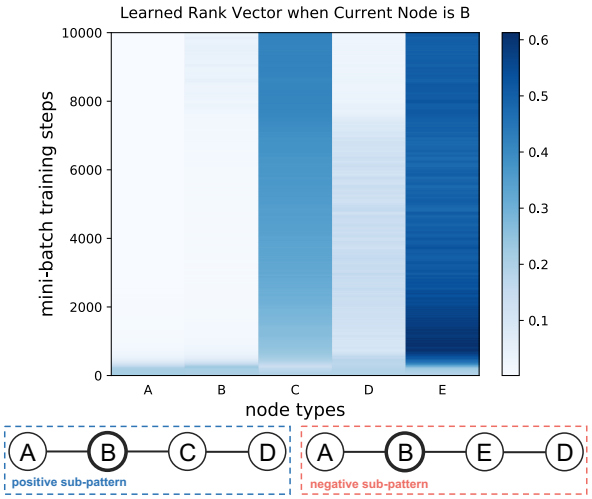

Figure 5: Rank values, over time, in the generated rank vector r_1 when the rank network is given h_1 encoding information from an initial step onto node B. Higher rank value signifies more importance.

model (GAM) uses attention to explore only a portion of the input graph, this puts our model at a disadvantage since it only has partial observability. Since the main goal is to show the viability of using attention, we limit the architecture of our tested models to simple ones (more detail below). The compared methods are summarized below.

- **Agg-Attr**: Given an attributed graph, one simple way to construct a feature vector is to get the component-wise average of the attribute vectors of all the nodes in the graph.
- **Agg-WL**: The first approach captures information from node attributes. However, it completely ignores the graph's structural information. The second method uses the Weisfeiler-Lehman (WL) algorithm [25] to calculate new node attributes that capture the local neighborhood of each node. The algorithm works by iteratively assigning a new attribute to each node by computing a hash of the attributes of neighboring nodes. We simply average the new attributes after running the WL algorithm to use as feature vector used for prediction.
- **Kernel-SP**: As in [34], we compare against the shortest path (SP) kernel which measures the similarity of a pair of graphs by comparing the distance of the shortest paths between nodes in the graphs. Since we use attributed graphs, we label the nodes in the graph by concatenating the categorical attributes.
- **Kernel-Gr**: As in [34], we also compare against the graphlet kernel which measures graph similarity by counting the number of different graphlets. Here, we evaluate against the 3-graphlet kernel and nodes are labeled as above.
- **GAM**: Our proposed approach which uses attention to steer the walk of an agent on an input graph.
- **GAM-mem**: Proposed approach with external memory. Note that given a budget of T for GAM , GAM-mem with n agents has access to same amount of information if each agent is constrained to take $\frac{T}{n}$ steps.

We used a logistic regression (LR) classifier with the first two baselines. To reduce overfitting, we applied ℓ_1 and ℓ_2 regularization and used a grid search over $\{0.01, 0.1, 1.0\}$ to select the ideal regularization penalty. Furthermore, we also did a grid search over the number of iterations for the WL algorithm, we tested over $\{2, 3, 4\}$. For a fair comparison, we limited the classification network for both our methods to a single softmax layer to make it equivalent to LR. We also limited the number of hidden layers in all other networks of our model to a single layer, whenever possible. For the graph-kernel based approaches, we used an SVM classifier using the precomputed kernel generated by each approach. Here, we did a grid search over $C = \{0.01, 0.1, 1.0\}$. We used a vector in $\mathbb{R}^{200}$ for Agg-WL and limited the size of the LSTM history vector to this size as well. In particular, we tried size = $\{156, 200\}$. We also tried the following sizes for the first and second hidden layers, respectively, of the step network: $(128, 164)$, and $(64, 128)$.

Since we did not find any noticeable change in the performance of GAM when increasing the following parameters, we fixed their values. We set the number of steps $T = 12$ and the number of samples $M = 20$. M is also the number of agents we run on each graph for prediction. For GAM-mem, we did a grid search over $T = \{12, 25\}$, and $M = \{5, 10\}$. We use the Adam algorithm for optimization [14] and fix the initial and final learning rates to 10^{-3} and 10^{-6}, respectively. We also did not use discounted reward as there was no noticeable gain, setting $\gamma = 1$. Finally, we limit the training of our methods to 200 epochs and applied early stopping using a validation set.

4.3 Classification Results

Table 1 shows the average classification accuracy, over 5-fold cross-validation, of the compared methods. From the results, we can see

Table 2: Performance of the baselines when we restrict their setting to that of GAM where they are given 20 randomly selected partial snapshots of each graph and have to predict by voting. The column "full" indicates the performance when the entire graph is seen and "partial" shows the performance when only parts of the graph is seen. "Diff." is the difference in performance, a ↓ means that performance deteriorated when only partial information is available and ↑ shows increase in performance.

method	dataset														
	HIV			NCI-1			NCI-33			NCI-83			NCI-123		
	full	partial	diff.	full	partial	diff.	full	partial	diff.	full	partial	diff.	full	partial	diff.
Agg-Attr	69.58	64.17	05.41 (↓)	64.79	59.58	05.21 (↓)	61.25	58.54	02.71 (↓)	58.75	62.71	03.96 (↑)	60.00	57.50	02.50 (↓)
Agg-WL	69.37	56.04	13.33 (↓)	62.71	51.46	11.25 (↓)	67.08	49.79	17.29 (↓)	60.62	51.46	09.16 (↓)	62.08	52.29	09.79 (↓)
GAM	-	74.79	-	-	64.17	-	-	67.29	-	-	67.71	-	-	64.79	-

that our proposed model is always among the top-2 in terms of performance on all tested datasets. In particular, the attention model with memory performs the best on four of the five datasets and comes in at second on the fifth dataset (NCI-33). In every single case, GAM-mem outperforms GAM which shows that adding an external memory to integrate information from various locations is beneficial. However, we find that GAM still performs respectably against the compared baselines and in fact comes in second on two of the tested datasets. We also find that GAM outperforms Agg-Attr and Agg-WL in almost every single case, which is remarkable since each agent in GAM only has access to a portion of the graph while the latter two have access to the entire graph. In our experiments, we find that the first two baselines perform the worst, almost always performing the worst on all the datasets. The kernel-based approaches are better, with the graphlet-based approach being superior. It is able to outperform GAM slightly. However, GAM-mem is consistently the best performer on all the datasets that were tested.

4.3.1 Applying Random Attention. Our experiments show that the attention model is competitive against baselines that observe the entire graph while our model is limited to seeing a portion of the graph. To demonstrate the effectiveness of attention further, we ran another experiment where we restrict the first two baselines to the setting of GAM. It is a straightforward modification since the methods also use the graph attribute vectors. However, the baselines do not have a concept of attention, so we use random attention where we sample 20 subgraphs from each graph using a random-walk of length 12. This limits the information available to the baselines to that which is available to GAM since we fixed $M = 20$ and $T = 12$.

Table 2 shows the result of the baselines when they only observe a random portion of each graph. It is clear that the performance deteriorates for both methods, with Agg-WL showing a more marked difference in performance. This is with the exception of Agg-Attr on NCI-83. In fact, we can see that the performance of Agg-WL drops so drastically that it performs almost no better than random guessing on four of the five datasets (NCI-1, NCI-33, NCI-83, and NCI-123). This shows that attention can help us examine parts of the graph that are relevant.

4.4 Parameter Study

We study the effect of varying step sizes T on performance of both GAM and GAM-mem. For each of the 5 datasets, we fixed all other parameters to the ones that yielded the best results and varied $T = \{1, 3, \cdots, 15, 18\}$. In both cases, accuracy increased as we increased the number of steps with $T = 12$ giving fairly good performance on all datasets on both methods. Surprisingly, we found that both models already performed relatively well when $T \geq 3$, in some cases being only 5-6% worse than the best accuracy. This may be because molecular graphs are fairly small in size. We found that GAM-mem, in general, benefits more from an increase in the size of T which may be due to the fact that we are using weighted pooling of the history vectors so the model can support longer walks.

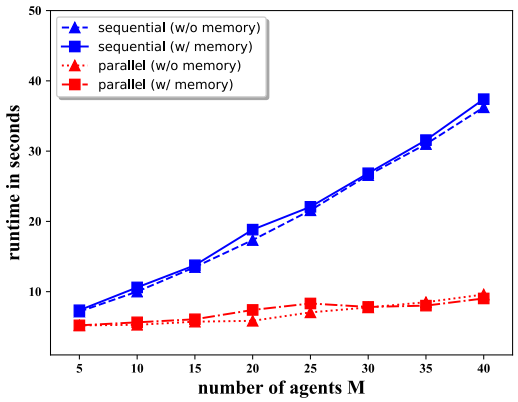

Figure 6: Average runtime when agents are run in parallel versus sequentially. Here we show the runtime for doing prediction on a mini-batch of 32 graphs with $T = 100$.

4.5 Parallel Execution of Agents

One advantage of our model is the ability to execute multiple agents in parallel during prediction time. This is particularly useful when the graph is large and we need multiple agents to explore different parts of the graph. For instance, in the task of malware classification on function-call graphs the graphs have been known to contain up to ~37,000 nodes [13]. Also, recall that in the proposed model the agents are not required to have access to the entire graph. In fact, at any time-point t, the model only needs access the current node c_t and its neighbors along with their attributes. If the graph is too large to load into memory, this information can be accessed on the fly (*e.g.*, from a database).

Since each agent in GAM can make an independent prediction, the agents can be run in parallel. The only step that needs to be done in the end is to average the predictions of all the agents. For GAM-mem, since the history vectors of all the agents are combined, we wait for all the agents to take T steps before we combine the information. However, the individual agents in GAM-mem can still explore the graph in parallel. Figure 6 shows the difference in runtime of our method when multiple agents are executed in

parallel versus in a single process. The experiments are conducted on a Linux machine with 48 CPU cores and 160GB of RAM. We see here that the parallelized version of GAM-mem is slightly slower. This can be expected since each agent's local memory needs to be combined to form the shared memory. However, it is clear that parallelization helps keep the runtime close to constant for both methods as the bottleneck is in the attention-guided walk.

5 CONCLUSION

In this work, we propose to study the problem of attention-based graph classification and introduced GAM, a general RNN-based framework that uses attention to learn data-driven features for attributed graphs. The model uses attention to process task-relevant parts of the graph (partial observability) and has an external memory to integrate information from various parts of the graph. The attention mechanism in our model is easily parallelizable. Empirical results show that the method can outperform methods that observe the entire graph.

There are a lot of interesting directions for future work. We intend to study the model using more expressive node typing strategies. We would also like to experiment with an extension of the model with a more sophisticated external memory (*e.g.*, making memory rewritable, and using memory to condition the output of the rank network). Finally, it would be interesting to test more flexible architectures for LSTM like Tree-LSTMs that seem more natural for graphs.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants IIS-1718310, MRI-1626236.

REFERENCES

[1] Nesreen K. Ahmed, Ryan A. Rossi, Rong Zhou, John Boaz Lee, Xiangnan Kong, Theodore L. Willke, and Hoda Eldardiry. 2018. Learning Role-based Graph Embeddings. In *arXiv preprint arXiv:1802.02896*.

[2] Lars Backstrom and Jure Leskovec. 2011. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the Fourth International Conference on Web Search and Web Data Mining*. 635–644.

[3] Jie Bao, Tianfu He, Sijie Ruan, Yanhua Li, and Yu Zheng. 2017. Planning Bike Lanes based on Sharing-Bikes' Trajectories. In *Proceedings of the Twenty-Second ACM SigKDD International Conference on Knowledge Discovery and Data Mining*.

[4] Karsten M. Borgwardt and Hans-Peter Kriegel. 2005. Shortest-Path Kernels on Graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining*. 74–81.

[5] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schonauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. 2005. Protein function prediction via graph kernels. *Bioinformatics* 21, 1 (2005), i47–i56.

[6] Duen Horng Chau, Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. 2011. Polonium: Tera-Scale Graph Mining for Malware Detection. In *Proceedings of the Eleventh SIAM International Conference on Data Mining*. 131–142.

[7] Kan Chen, Jiang Wang, Liang-Chieh Chen, Haoyuan Gao, Wei Xu, and Ram Nevatia. 2015. ABC-CNN: An Attention Based Convolutional Neural Network for Visual Question Answering. In *arXiv preprint arXiv:1511.05960v2*.

[8] Dong-Yeon Cho, Yoo-Ah Kim, and Teresa M. Przytycka. 2012. Chapter 5: Network Biology Approach to Complex Diseases. *PLOS Computational Biology* 8 (2012), e1002820.

[9] Edward Choi, Mohammad Taha Bahadori, Le Song, Walter F. Stewart, and Jimeng Sun. 2017. GRAM: Graph-based Attention Model for Healthcare Representation Learning. In *Proceedings of the Twenty-Third ACM SigKDD International Conference on Knowledge Discovery and Data Mining*. 787–795.

[10] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. 2015. Deep reinforcement learning in large discrete action spaces. In *arXiv preprint arXiv:1512.07679*.

[11] David K. Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *Proceedings of the Twenty-Eight Annual Conference on Neural Information Processing Systems*. 2224–2232.

[12] Felix Gers, Jurgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: continual prediction with LSTM. In *Proceedings of the Tenth International Conference on Artificial Neural Networks*. 850–855.

[13] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the Sixteenth ACM Conference on Computer and Communications Security*. 611–620.

[14] Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the Third International Conference on Learning Representations*.

[15] Risi Kondor and Horace Pan. 2016. The Multiscale Laplacian Graph Kernel. In *Proceedings of the Twenty-Ninth Annual Conference on Neural Information Processing Systems*. 2982–2990.

[16] Xiangnan Kong, Wei Fang, and Philip S. Yu. 2011. Dual active feature and sample selection for graph classification. In *Proceedings of the Seventeenth ACM SigKDD International Conference on Knowledge Discovery and Data Mining*. 654–662.

[17] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the Thirteenth Conference on Empirical Methods in Natural Language Processing*. 1412–1421.

[18] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent models of visual attention. In *Proceedings of the Twenty-Seventh Annual Conference on Neural Information Processing Systems*. 2204–2212.

[19] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning Convolutional Neural Networks for Graphs. In *Proceedings of the Thirty-Third International Conference on Machine Learning*. 2014–2023.

[20] Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. 2017. Matching Node Embeddings for Graph Similarity. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 2429–2435.

[21] Shirui Pan, Jia Wu, Xingquan Zhu, and Chengqi Zhang. 2015. Graph Ensemble Boosting for Imbalanced Noisy Graph Stream Classification. *IEEE Transactions on Cybernetics* 45, 5 (2015), 940–954.

[22] Aaditya Prakash, Siyuan Zhao, Sadid A. Hasan, Vivek Datla, Kathy Lee, Ashequl Qadir, Joey Liu, and Oladimeji Farri. 2017. Condensed Memory Networks for Clinical Diagnostic Inferencing. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 3274–3280.

[23] David Rogers and Mathew Hahn. 2010. Extended-connectivity fingerprints. *Journal of Chemical Information and Modeling* 50, 5 (2010), 742–754.

[24] Ryan A. Rossi, Rong Zhou, and Nesreen K. Ahmed. 2017. Estimation of graphlet statistics. In *arXiv preprint arXiv:1701.01772*.

[25] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research* 12 (2011), 2539–2561.

[26] Nino Shervashidze, S. V. N. Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M. Borgwardt. 2009. Efficient graphlet kernels for large graph comparison. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*. 488–495.

[27] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. 2015. End-To-End memory networks. In *Proceedings of the Twenty-Eight Annual Conference on Neural Information Processing Systems*. 2440–2448.

[28] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph Attention Networks. In *arXiv preprint arXiv:1710.10903v2*.

[29] David Weininger. 1988. SMILES, a chemical language and information system. *Journal of Chemical Information and Modeling* 28 (1988), 31–36.

[30] Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory Networks. In *Proceedings of the Second International Conference on Learning Representations*.

[31] Daan Wierstra, Alexander Forster, Jan Peters, and Jurgen Schmidhuber. 2007. Solving deep memory POMDPs with recurrent policy gradients. In *Proceedings of the Seventeenth International Conference on Artificial Neural Networks*. 697–706.

[32] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (1992), 229–256.

[33] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of the Thirty-Second International Conference on Machine Learning*. 2048–2057.

[34] Pinar Yanardag and S. V. N. Vishwanathan. 2015. Deep Graph Kernels. In *Proceedings of the Twenty-First ACM SigKDD International Conference on Knowledge Discovery and Data Mining*. 1365–1374.

[35] Jingyuan Zhang, Bokai Cao, Sihong Xie, Chun-Ta Lu, Philip S. Yu, and Ann B. Ragin. 2016. Identifying Connectivity Patterns for Brain Diseases via Multi-side-view Guided Deep Architectures. In *Proceedings of the Sixteenth SIAM International Conference on Data Mining*. 36–44.