# Analysis of Information Propagation in Ethereum Network Using Combined Graph Attention Network and Reinforcement Learning to Optimize Network Efficiency and Scalability

Stefan Behfar
University of Cambridge
Cambridge, UK
skb67@cam.ac.uk

Richard Mortier
University of Cambridge
Cambridge, UK
rmm1002@cam.ac.uk

Jon Crowcroft
University of Cambridge
Cambridge, UK
jon.crowcroft@cl.cam.ac.uk

## Abstract

Blockchain technology has revolutionized the way information is propagated in decentralized networks. Ethereum, as a major blockchain platform, plays a pivotal role in facilitating smart contracts and decentralized applications. Modeling information propagation dynamics in Ethereum is crucial for ensuring network efficiency, security, and scalability. In this study, we introduce three innovative theorems, aiming to use Graph Attention Network (GAT) to analyze the information propagation patterns; while our major contribution is to develop a combined GAT and Reinforcement Learning (RL) method to enhance the network efficiency and scalability by optimizing the gas limits for block processing. It learns the best actions to take in various network states, ultimately leading to improved Ethereum network efficiency and throughput and optimize gas limits for block processing. Additionally, we explore methods for effectively aggregating transaction data by capturing graph structures and updating node embeddings for transaction pattern prediction. To evaluate scalability, we implement and compare three Graph Neural Network (GNN) models—GraphConv, GraphSAGE, and GAT—comparing their performance at scale.

**ACM Reference Format:**
Stefan Behfar, Richard Mortier, and Jon Crowcroft. 2025. Analysis of Information Propagation in Ethereum Network Using Combined Graph Attention Network and Reinforcement Learning to Optimize Network Efficiency and Scalability. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3721146.3721965


*EuroMLSys '25, March 30–April 3, 2025, Rotterdam, Netherlands*

ACM ISBN 979-8-4007-1538-9/25/03
https://doi.org/10.1145/3721146.3721965

## 1 Introduction

The advent of blockchain technology has revolutionized the world of cryptocurrencies, with Ethereum being one of the most prominent platforms. Ethereum's decentralized nature and smart contract functionality have resulted in a massive amount of transactional data generated daily. As a result, efficient pattern recognition methods are essential for extracting meaningful insights and addressing security issues within the network.

A graph convolution algorithm based on deep learning techniques has been proposed to automatically generate features using a graph algorithm. Kipf and Welling (2017) and Li et al. (2018) used Graph Convolutional Network (GCN) model and its derivatives that apply semi-supervised classification using neural network models to graph data. Chen et al. (2020) suggested a GCN-based phishing detection model which samples subgraphs by random walk and applies node embeddings and a model to incorporate spatial structures and node features. Lin et al. (2020) investigated modeling the Ethereum transaction network as a weighted temporal graph and a temporal weighted multigraph embedding to incorporate temporal and weighted transaction edges. Moreover, there are some studies on modeling smart contract interactions as graph structures (Chen et al. 2020, Torres and Camino 2021), but it is challenging to accurately attribute the performance load of specific smart contract operations to the edges and nodes of the graph (Zhuang et al., 2020).

While prior studies have explored various aspects of blockchain analytics, few have addressed the challenges of optimizing network efficiency, particularly in the context of gas cost management, and determining optimal gas limits remains an open problem, as it requires dynamically adapting to network conditions while ensuring fairness and security. To tackle this, we introduce a novel Graph Attention Network and Reinforcement Learning (GAT-RL) framework that leverages graph-based learning to model transaction dependencies and reinforcement learning to optimize gas allocation strategies.

Our approach primarily leverages graph convolutional layers to facilitate information propagation while effectively aggregating data from neighboring nodes and dynamically updating graph structures. In the next section, we first outline our model design, explain theorems on gas limit optimization and maximizing network throughput, then introduce a novel integration of GAT and RL for network optimization. We apply our method to Ethereum by extracting network data for a specific block range, developing an algorithm for gas limit optimization, and demonstrating how GAT-RL outperforms RL-only approaches at scale.

## 2 Model Design

To enable information propagation across multiple layers, the graph convolution operation is performed iteratively through multiple graph convolutional layers. The output of one layer serves as the input to the next layer, allowing the propagation of information through the network. The node representations are updated layer by layer, allowing information from neighbors and their neighbors to be incorporated into the node features. This process facilitates the learning of meaningful representations that capture the dependencies and patterns in the blockchain network. We aim to use this for enhancing the network efficiency and scalability by optimizing the gas limits for block processing.

### 2.1 Optimizing Gas Limits for Block Processing

Optimizing gas limits in a blockchain network is a challenging problem. Traditional methods often rely on heuristics or fixed rules, which may not adapt well to changing network conditions. This can result in suboptimal throughput and resource allocation. Reinforcement Learning (RL) provides a framework for agents to learn and adapt their decision-making based on interactions with an environment. In the context of Ethereum, an RL agent can learn to adjust gas limits dynamically to maximize throughput while minimizing processing time and congestion. We aim to use RL in the Ethereum network to optimize gas costs in general, where in Theorem 1, we attempt to prove it mathematically.

**Theorem 1:** In an Ethereum network where each transaction takes a fixed time to process, the optimal gas limit that maximizes throughput is when $\frac{G}{T} > N$, where N is the total number of transactions in the block, G is the gas limit, and T is the time taken to process a single transaction.

**Proof:** To prove the theorem, let's consider a simplified Ethereum network with a fixed set of transactions and a single miner. The objective is to find the optimal gas limit that maximizes throughput, assuming each transaction takes a fixed time to process and gas usage is linearly related to time. Let N be the total number of transactions in the block, G be the gas limit (maximum amount of gas that can be used in the block), and T be the time taken to process a single transaction (assumed fixed).

The total time taken to process all transactions in the block can be represented as: Total processing time = $N \times T$. To maximize throughput, we need to maximize the number of transactions processed in the block while respecting the gas limit. If the total gas used exceeds the gas limit, then the block will be full, and the number of transactions processed will be limited by the gas limit:

$$\text{Transactions processed} = \min(N, \frac{G}{T}) \tag{1}$$

To maximize throughput, we need to find the gas limit $G$ that maximizes the number of transactions processed:

$$\begin{aligned} \text{maximize Transactions processed} &= \min(N, \frac{G}{T}) \\ \text{subject to } G &\geq 0 \end{aligned} \tag{2}$$

Since the number of transactions processed is a linear function of $G$, we can differentiate it with respect to $G$ and set the derivative to zero to find the maximum:

$$\frac{d}{dG}\left[\min(N, \frac{G}{T})\right] = 0 \tag{3}$$

To handle the "min" operation, we consider two cases:
**Case 1:** When $\frac{G}{T} > N$, the maximum number of transactions processed will be $N$.

$$\frac{d}{dG}[N] = 0 \tag{4}$$

**Case 2:** When $\frac{G}{T} \leq N$, the maximum number of transactions processed will be $\frac{G}{T}$.

$$\frac{d}{dG}\left[\frac{G}{T}\right] = 0 \tag{5}$$

Solving for $G$ in Case 2: $1/T = 0$

Since in Case 2, when $\frac{G}{T} \leq N$, we cannot find a valid solution for $G$ that maximizes the number of transactions processed. Therefore, the maximum throughput is achieved when $\frac{G}{T} > N$. In conclusion, in a Ethereum network where each transaction takes a fixed time to process and gas usage is linearly related to time, the optimal gas limit that maximizes throughput is when $\frac{G}{T} > N$. This means that the gas limit should be set high enough to allow the processing of all transactions in the block without reaching the gas limit constraint.

**Theorem 2:** In the Ethereum network, prove how the objective function

$$F = \sum(TransactionPriority \cdot \frac{GasCost}{TransactionComplexity}) \tag{6}$$

maximizes the network throughput, where: *Transaction Priority* represents the urgency or importance of a transaction. *Transaction Complexity* is a measure of the computational complexity of a transaction.

**Proof:** The optimization objective is to determine the gas limit for each block to maximize network throughput,

defined as the total number of successfully processed transactions within a given time frame, while minimizing block processing time and congestion. We initially make some definitions:

1. **Gas Usage Model:** Gas usage is not strictly linear with time. It depends on the complexity of the smart contracts involved in each transaction. The total gas used in the block cannot exceed the gas limit ($G$).

$$\sum(\text{Gas Cost}) \leq G \tag{7}$$

2. **Block Size Limit:** There is a maximum block size limit in terms of gas. Blocks cannot exceed this gas limit, and transactions must be prioritized accordingly. The total computational time for transactions in the block must not exceed the block processing time ($T_{\text{block}}$).

$$\sum(\text{Transaction Complexity} \cdot \text{Processing Time}) \leq T_{block} \tag{8}$$

For the mathematical representation of how Reinforcement Learning (RL) can maximize throughput in gas optimization, let's define some key elements: **States (S):** The state space represents network conditions, and can be represented as a vector of state variables, $S = [S_1, S_2, ..., S_n]$, where $n$ is the number of state variables. **Actions (A):** The action space represents different gas price choices for a transaction. Actions can be represented as a vector of possible gas price levels, $A = [A_1, A_2, ..., A_m]$, where $m$ is the number of gas price levels. **Rewards (R):** The reward function represents the efficiency of a gas pricing decision. In this context, rewards could be based on transaction confirmation time, fee income, or other relevant metrics. We define the reward function as $R(s, a)$, where $s$ is the current state, and $a$ is the chosen action. **Policy ($\pi$):** The policy is a mapping from states to actions, $\pi : S \rightarrow A$, that defines which gas price to choose in each network state. The goal is to find an optimal policy $\pi^*$. **Value Function (V):** The value function represents the expected cumulative reward when following a policy $\pi$ from a given state. It can be defined as $V(s) = \sum(\gamma^t \times R(s_t, a_t))$ where $t$ is the time steps, $\gamma$ is the discount factor, $s_t$ is the state at time step $t$, and $a_t$ is the action (gas price) chosen at time step $t$.

The objective is to find the policy $\pi^*$ that maximizes the expected cumulative reward over time while accounting for the discount factor $\gamma$. Mathematically, this can be represented as:

$$\pi^* = \arg\max \left[ \sum(\gamma^t \times R(s_t, a_t)) \right] \tag{9}$$

Now, let's express the throughput:

$$\text{Throughput} = \frac{\text{Transactions processed}}{\text{Block processing time}} \tag{10}$$

The RL training process involves adjusting the policy based on past experiences to maximize the expected cumulative reward. Over time, the agent refines its policy, converging toward an optimal policy $\pi^*$ that maximizes throughput in Eq. (10) in the Ethereum network. To align this with the reward function, we can consider a case where the reward is solely based on Gas Cost:

$$R(s, a) = \frac{\text{Gas Cost}}{\text{Transaction Complexity}} \tag{11}$$

Now the optimal policy that maximizes the expected cumulative reward becomes:

$$\pi^* = \arg\max \left[ \sum \gamma^t \times \frac{\text{Gas Cost}}{\text{Transaction Complexity}} \right] \quad subject to \sum(\text{Gas Cost}) \leq G \tag{12}$$

The RL agent aims to optimize its policy to maximize the sum of rewards. Since the reward function involves the ratio of Gas Cost to Transaction Complexity, and ($\gamma^t$) can be approximated for Transaction Priority, the agent needs to optimize the objective function given in the theorem to reach the efficient processing of transactions.

### 2.2 GAT and RL to Optimize Network Efficiency

In blockchain networks, the efficient allocation of resources is crucial for scalability and user experience. By combining GCN/GAT and RL, where GCN/GAT can capture complex relationships between blocks and transactions, while RL agents can learn optimal strategies for gas limit adjustments, we offer a solution that can adapt to changing network conditions and improve overall efficiency. Blockchain data can be represented as a graph, with blocks and transactions as nodes and edges. This graph structure provides a suitable basis for applying graph-based techniques to analyze and model network dynamics.

**Theorem 3:** Combining GAT with RL in modeling Ethereum network efficiency out-performs RL alone.

Let $G = (V, E)$ represent the Ethereum network graph, where $V$ is the set of nodes (miners/validators) and $E$ the set of edges (communication links). Let $S_t \in \mathbb{R}^{|V| \times d}$ denote the global state at time $t$, where $S_{t,v}$ is the local state vector of node $v$. Consider a reinforcement learning agent that selects actions to maximize a cumulative reward $R = \sum_{t=0}^{T} \gamma^t r_t$, where $r_t$ is the reward at time $t$.

Compare two agents:

- A *vanilla RL agent* with policy $\pi_t = \pi(S_t; \theta)$ directly based on the raw state.
- A *GAT-enhanced RL agent* with policy $\pi_t = \pi(\tilde{S}_t; \theta)$, where $\tilde{S}_t$ is the graph-processed state produced by a graph attention network (GAT).

The GAT-enhanced agent achieves at least the same, and typically a strictly higher, expected cumulative reward:

$$J(\pi_{\text{gat}}) \geq J(\pi_{\text{rl}}) \tag{13}$$

where:

$$J(\pi) = \mathbb{E}\left[ \sum_{t=0}^{T} \gamma^t r_t \mid \pi \right] \tag{14}$$

with strict inequality for sufficiently large graphs with non-trivial communication patterns.

**Proof:**

The processed state for the vanilla RL agent at time $t$ is:

$$S_{\text{processed}}(t) = S_t W_s \tag{15}$$

where $W_s$ is a learnable transformation matrix.

The policy and value function are computed as:

$$\pi_t = \text{softmax}(S_{\text{processed}}(t)W_\pi), \quad V_t = S_{\text{processed}}(t)W_v \tag{16}$$

The reward $r_t$ depends on $S_{\text{processed}}(t)$ and the chosen action.

—

For the GAT-enhanced agent, the processed state incorporates graph attention:

$$\tilde{S}_t = \text{GAT}(G, S_t; W_s) \tag{17}$$

This reflects local structural dependencies through attention-weighted aggregation:

$$\tilde{S}_v = \sum_{u \in \mathcal{N}(v)} \alpha_{vu}(S_u W_s) \tag{18}$$

where $\alpha_{vu}$ is the learned attention weight for edge $(v, u)$.

The policy and value function become:

$$\pi_t = \text{softmax}(\tilde{S}_t W_\pi), \quad V_t = \tilde{S}_t W_v \tag{19}$$

The reward function depends on $\tilde{S}_t$ instead of $S_t$.

—

**Inductive Argument.** At $t = 0$, both agents start from the same initial state $S_0$, but the GAT agent has richer state processing:

$$\tilde{S}_0 = \text{GAT}(G, S_0; W_s) \tag{20}$$

The GAT term incorporates structural information from the graph, which the vanilla agent ignores. This yields a better-informed policy:

$$\pi_{\text{gat},0} = \pi(\tilde{S}_0; \theta) \tag{21}$$

leading to:

$$\mathbb{E}[r_0 \mid \pi_{\text{gat}}] \geq \mathbb{E}[r_0 \mid \pi_{\text{rl}}] \tag{22}$$

**Inductive Step.** Assume for all $t \leq k$:

$$\mathbb{E}[r_t \mid \pi_{\text{gat}}] \geq \mathbb{E}[r_t \mid \pi_{\text{rl}}] \tag{23}$$

Then, at step $k + 1$, the state transitions to $S_{k+1}$ based on the previous state and action. The GAT agent processes this into:

$$\tilde{S}_{k+1} = \text{GAT}(G, S_{k+1}; W_s) \tag{24}$$

and the vanilla agent uses:

$$S_{\text{processed}}(k + 1) = S_{k+1} W_s \tag{25}$$

Because $\tilde{S}_{k+1}$ retains more structural information than $S_{\text{processed}}(k+1)$, the GAT policy has access to richer features, leading to:

$$\mathbb{E}[r_{k+1} \mid \pi_{\text{gat}}] \geq \mathbb{E}[r_{k+1} \mid \pi_{\text{rl}}] \tag{26}$$

This completes the induction, showing:

$$J(\pi_{\text{gat}}) \geq J(\pi_{\text{rl}}) \tag{27}$$

While Theorem 3 claims that GAT-enhanced reinforcement learning (GAT-RL) is more effective than vanilla RL, this claim relies on some implicit assumptions that yields a better-informed policy. In the next section, we provide empirical evidence comparing the performance of GAT-RL and vanilla RL on a well-defined Ethereum network benchmark.

## 3 Empirical Analysis

### 3.1 Dataset Collection

This section discusses the publicly available Ethereum datasets and how we obtain them. Creating a complete transaction graph for all Ethereum blocks would be a computationally intensive task, as it would involve processing and storing a large amount of data. Additionally, the Ethereum blockchain is constantly growing, so the graph would continuously expand as new blocks are added. However, we provide an algorithm to generate a transaction history graph for a range of blocks to create a graph of transactions between Ethereum addresses within a specified block range.

1. Place the URL of the Ethereum node obtained from Infura/Alchemy website in order to access the Ethereum Mainnet.
2. Set the `start_block` and `end_block` variables to specify the block range to create the transaction history graph; one can create a descending range (`latest_block, 0, -1`).
3. Create history of the transactions adding graph nodes and edges.

We ran the models on a MacBook Pro equipped with an Intel Core i9 processor, featuring 8 cores, speed of up to 4.8 GHz, and 30 GB of RAM.

### 3.2 Optimal Gas Limit Design

Using RL without GAT, Algorithm 1 (The code is given in the supplementary material) presents a model designed to determine the optimal gas limit, based on the network congestion, and transactions processed according to theorem 1.

The plotted results in Figure 1 show the learning progress of the RL agent as it attempts to optimize gas limits for block processing in the Ethereum network. The x-axis represents the blocks, while the y-axis represents the throughput (number of transactions processed) for each block. Here, we explain the results:

1. **Line Plots:** The line plot represents the throughput achieved by the RL agent in different episodes and block numbers. The agent starts with random gas limits and gradually learns to adjust the gas limits to maximize throughput, which proves our algorithm efficacy.
2. **Learning Progress:** As the block number increases, the graph shows how the agent's throughput improves over time. Initially, the agent explores various gas limit choices, leading to fluctuations in throughput, then the throughput values stabilize and tend to increase.
3. **Convergence:** The convergence of the RL agent can be observed when the lines start to converge to a more

Algorithm 1: Optimal gas limit design

**1. Initialization**

**2. Iterative Process:**

Simulate Block Processing:

*Calculate the expected block processing time ($E(T_{block})$) based on the average transaction processing time ($T_{transaction}$) and the number of transactions included in the block ($N_{transactions}$):*

$$E(T_{\text{block}}) = N_{\text{transactions}} \cdot T_{\text{transaction}}$$

*Determine the congestion level (congestion_level) based on the gas limit (gas_limit) and the number of transactions with higher gas fees:*

$$congestion\_level = \frac{N_{\text{high\_gas\_fee\_transactions}}}{gas\_limit}$$

Check Convergence:

*If $E(T_{block})$ is close to target time and congestion_level is below congestion threshold, the algorithm terminates.*

Adjust Gas Limit:

*If $E(T_{block})$ is too high or congestion_level is too high, reduce gas_limit by gas_limit_increment: gas_limit = gas_limit − gas_limit_increment*

*If $E(T_{block})$ is too low or congestion_level is too low, increase gas_limit by gas_limit_increment: gas_limit = gas_limit + gas_limit_increment*

**3. Algorithm Convergence:**

The algorithm repeats the iterative process until it converges or reaches a predefined maximum number of iterations.

stable throughput pattern. This indicates that the agent has learned optimal gas limits for maximizing throughput.

4. **Exploration-Exploitation Trade-off:** The lines might show some exploration and exploitation behavior. Initially, the agent explores different gas limit options to learn about their effects on throughput. As the agent gains knowledge, it starts exploiting its learned Q-values to choose better actions and converges to a higher throughput. The randomness in the lines is due to the inherent stochasticity in the simulation of the environment and RL updates. The final throughput achieved after the RL agent's convergence is the most critical outcome of the learning process. A higher final throughput indicates that the RL agent has successfully optimized gas limits to maximize transaction processing.

In Figure 2, we illustrate the gas limit optimization achieved by the RL agent across all episodes and plot the average gas limit over the learning rate, and exploration probabilities. This provides a representation of the RL agent's learning progress and performance in maximizing throughput. As the exploration probability and learning rate increases, the averaged gas limit decreases, which shows optimizing gas limit by the RL agent.

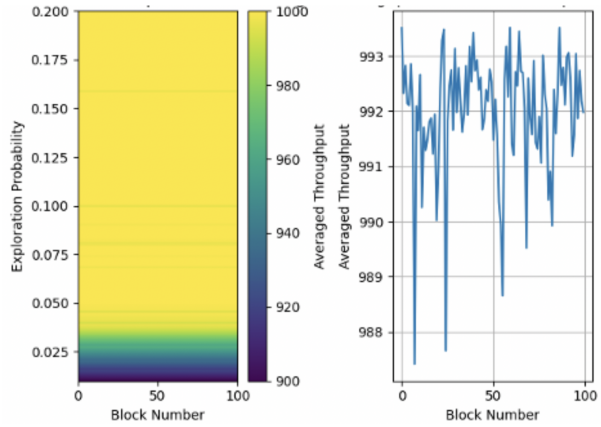


**Figure 1.** The average throughput achieved by the RL agent w.r.t. block number and exploration probability.

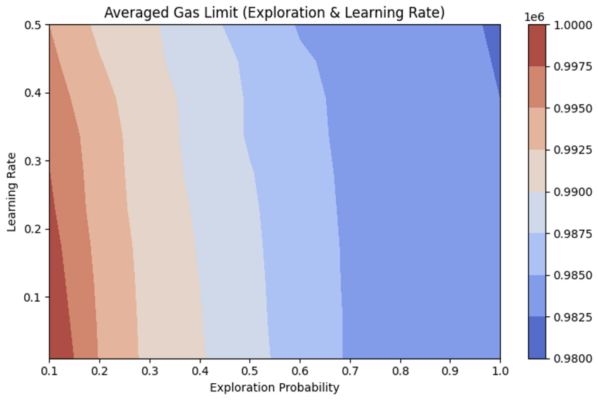


**Figure 2.** Optimizing gas limit achieved by the RL agent: average gas limit w.r.t. the learning rate, and exploration probability.

### 3.3 GAT-RL and Throughput Maximization

Our GAT-RL method is defined with Proximal Policy Optimization (PPO), RL and sparse tensor support. The code algorithm given in Algorithm 2 (see theorem 2,3) trains the GAT-RL on the Ethereum dataset and measures its performance after each epoch for about 10 epochs, as seen in Figure 3; we train GAT and GAT-RL separately with num_epochs = 1000 though.

In Figure 3, we present the Loss for both GAT and GAT-RL. The GAT Loss reflects how well the Graph Attention Network captures and understands the relationships and patterns in the graph-structured data. A lower GAT Loss indicates that the GAT is effectively learning meaningful representations from the graph. The GAT-RL Loss corresponds to the case where GAT is combined with the Proximal Policy Optimization (PPO) algorithm. It represents how well the policy is being updated to maximize cumulative rewards.

Algorithm 2: GAT-RL model design

**Steps:**

1. For each epoch in range(num_epochs):
   *Create subgraphs for inductive learning*
   *Define and initialize GAT model*
   *Train GAT model on the training data*
   *Compute and store the GAT loss*
2. Define the Ethereum optimization RL environment:
   *Set initial node features and masks (train/test)*
   *Define apply_resource_allocation method*
   *Define calculate_reward method using normalized rewards*
3. Define PPO policy and value function networks:
   *Initialize policy and value function networks*
   *Define policy optimizer and value function optimizer*
4. Train PPO agent using PPO algorithm:
   *for each epoch in range(num_epochs):*
   *Initialize environment state*
   *while not done:*
   *Sample action from policy*
   *Apply action to environment*
   *Observe next state and compute reward*
   *Store state, action, log-probability, reward*
   *Compute advantages using stored rewards*
   *Normalize advantages*
   *Update policy and value function using PPO loss*
5. Train GAT-RL in a combined manner:
   *for each epoch in range(num_epochs):*
   *Train GAT model*
   *Update RL environment with GAT-enhanced state*
   *Train RL agent using PPO*
   *Combine GAT loss and PPO loss for joint optimization*


**Figure 3.** GAT and GAT-RL (PPO) loss over 1000 blocks: x-Axis is Epoch and y-Axis is Loss

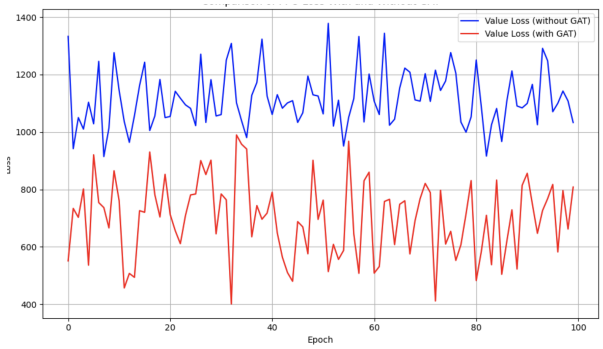


**Figure 4.** Comparison: Value Loss with and without GAT over 1000 blocks: x-Axis is Epoch and y-Axis is Loss

A lower GAT-RL Loss suggests that the policy updates are successfully improving the policy's performance.

**3.3.1 Ablation study:** In the ablation study we conducted a value loss comparison between the GAT-RL method and RL-only method within the latest 1000 blocks range sample (a good representative of the Ethereum network activity), two key components were examined. Among the baseline RL-only and the full GAT-RL method, the latter incorporates the Graph Attention Network for enhanced gas optimization, while the baseline RL-only relies solely on reinforcement learning mechanism. Figure 4 showing value loss with GAT (GAT-RL) and without GAT (RL) reveals that the GAT-RL method demonstrates notable improvements in performance compared to the RL-only baseline although very volatile. The result underscores the impactful contribution of the Graph Attention Network in the GAT-RL method, showcasing its effectiveness in optimizing gas usage and enhancing network efficiency in Ethereum, which establishes the GAT-RL method as a superior approach for gas optimization in Ethereum network.

### 3.4 Relative Scaling of Large Ethereum Networks

Applying GCN to a larger scale Ethereum network to address scalability and efficiency challenges demands careful consideration of the data representation and its architecture, where its implementation for a large-scale Ethereum network requires optimizations like using sparse matrices and distributed computing to handle the massive amount of data efficiently. Additionally, one might need to design custom GCN architectures to achieve better performance for Ethereum network analysis. To handle such scenarios, one common approach is to use GraphSAGE (Graph Sample and Aggregated) (Hamilton et al., 2017), which enables better scalability and captures more complex relationships in the data. We implement sparse matrices using PyTorch Geometric for network scaling, then compare the three methods for different numbers of Ethereum blocks. This is more theoretical comparison with relative scaling due to our computational constraints. Scaling GCNs using sparse matrices involves modifying the conventional GCN equations to efficiently handle sparse adjacency matrices. The formulations are as follows:

**Table 1.** Accuracy/Performance for scaling with GraphConv, GraphSAGE, and GAT, where accuracy is quantified using a train and test mask split of 50%, applied over 1,000 epochs. Performance is calculated in terms of block processing time.

| Accuracy / Performance(sec) | GraphConv | GraphSAGE | GAT |
|---|---|---|---|
| #blocks = 1000 | 0.73 / 0.8577 | 0.99 / 0.635 | 0.99 / 1.215 |
| #blocks = 3000 | 0.99 / 0.8524 | 0.99 / 0.558 | 0.99 / 1.989 |
| #blocks = 9000 | 0.811 / 17.289 | 0.297 / 3.870 | 0.934 / 24.467 |

**3.4.1 GraphConv Neighbor Sampling.** The GraphConv layer performs aggregation over the neighbors of each node, which involves sampling a fixed-size neighborhood for each node and aggregating the features of the sampled neighbors. Given an input feature matrix $X$ of shape $(N, D)$, where $N$ is the number of nodes, $D$ is the number of input features per node, and the sparse adjacency matrix $A$ of shape $(N, N)$, the output feature matrix $H$ at each GraphConv layer is calculated as $H = \sigma(A@X@W)$; where @ represents matrix multiplication, $\sigma$ is the activation function typically ReLU or another non-linear function, and $W$ is the weight matrix of the layer which needs to be learned during training.

**3.4.2 GraphSAGE Neighbor Sampling.** To handle large graphs efficiently, GraphSAGE employs neighbor sampling, which samples a fixed number ($K$) of neighbors for each node. This helps reduce memory consumption during training and speeds up computation. The equation for neighbor sampling involves selecting a random subset of neighbors for each node $i$ based on the adjacency matrix $A$:

$$N_i = \text{sample\_neighbors}(A, i, K) \tag{28}$$

sample_neighbors$(A, i, K)$ is a function that returns a set of $K$ sampled neighbors for node $i$ from adjacency matrix $A$. The sampled neighbor nodes' features are aggregated using mean or sum pooling to create the feature representation for each node's neighborhood:

$$H_{N_i} = \text{sum\_pooling}(H[N_i])/K \tag{29}$$

where $H[N_i])$ represents the feature matrix of the sampled neighbors $N_i$, and sum_pooling and mean_pooling are functions that calculate the sum or mean of the rows of a matrix.

The output feature matrix $H$ is updated with the aggregated neighborhood features to create the final node representations for the next layer. By handling the adjacency matrix as a sparse tensor, we can save memory and computation time when dealing with large graphs. We use PyTorch's sparse_coo_tensor to create a sparse matrix and measure performance and scalability. PyTorch's sparse_coo_tensor is used to create a sparse matrix in CSC format. This format represents a sparse matrix by storing only the non-zero elements and their corresponding row and column indices. Using sparse matrices can significantly reduce memory consumption and speed up computations.

**3.4.3 Scaling GCNs using attention mechanism.** The GAT model (Velickovic et al., 2017) introduces an "attention" mechanism that assigns varying weights to different nodes, enhancing the model's adaptability to the unique structure of heterogeneous networks. In GAT, attention coefficients ($\alpha_{ij}$) are computed to capture the importance of neighboring nodes in aggregating node features. These coefficients are calculated using an attention mechanism as:

$$e_{ij} = \text{LeakyReLU}(a^T [W \cdot h_i || W \cdot h_j]) \tag{30}$$

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})} \tag{31}$$

where $h_i$ and $h_j$ are the feature representations of nodes i and j, W is a learnable weight matrix, and a is a shared parameter. After computing the attention coefficients, $h_i$ are aggregated based on their neighbors' importance using the attention coefficients ($\alpha_{ij}$), and is then used for subsequent computations by taking the weighted sum of its neighboring nodes' features. To scale using sparse matrices, we utilize PyTorch Geometric and its sparse tensor capabilities.
We show accuracy and performance between GraphConv, GraphSAGE, and GAT in Table 1 for different numbers of blocks. Accuracy is quantified using a train and test mask split of 50%, applied over a comprehensive series of 1,000 epochs. Performance is calculated in terms of block processing time. As observed in the Table, our GAT-based method appears to outperform other sampling algorithms in terms of combined accuracy and performance (in seconds) when number of blocks increase from 1000 to 9000.

## 4 Conclusion

This study aimed to investigate information propagation of the Ethereum network using a combined GAT-RL method where we focused on the research gap in utilizing graph attention network to facilitate information propagation, while using reinforcement learning to optimize gas limit in Ethereum network. We introduced 3 theorems to investigate the optimal gas limit that maximizes the throughput. We performed an ablation study showing that the GAT-RL method demonstrates notable improvements in performance compared to the RL-only method, which underscores the impactful contribution of the GAT in the combined GAT-RL method for producing an embedding where RL agent could learn the best actions to take in various network states, ultimately improve the throughput. Applying GAT-RL to large-scale

blockchain network presents scalability challenges; in due course, we compared accuracy and performance among three GNN approaches: GraphConv, GraphSAGE, and GAT. The results show that GAT outperforms others in terms of accuracy/performance in producing the embedding, which is the basis for RL calculation in our combined GAT-RL method.

## References


**[1]** Kipf, T.N., and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. *ICLR-17*.

**[2]** Li, Q., Han, Z., and Wu, X-M. 2018. Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning. *AAAI-18*.

**[3]** Chen, L., Peng, J., Liu, Y., Li, J., Xie, F., and Zheng, Z. 2020. Phishing Scams Detection in Ethereum Transaction Network. *ACM Trans. Internet Technol. 21, 1, Article 10*.

**[4]** Chen, T., Li, Z., Zhu, Y., Chen, J., Luo, X., Lui, J. C. S., ... and Zhang, X. (2020). Understanding Ethereum via graph analysis. ACM Transactions on Internet Technology (TOIT), 20(2), 1-32.

**[5]** Lin, D., Wu, J., Yuan, Q., and Zheng, Z. 2020. T-edge: Temporal weighted multidigraph embedding for Ethereum transaction network analysis. *Frontiers in Physics 8, 204*.

**[6]** Hamilton, W., Ying, Z., and Leskovec, J. 2017. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems 30*.

**[7]** Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. 2017. Graph attention networks. *Stat 1050, 20*.

**[8]** Kanezashi, H., Suzumura, T., Liu, X., and Hirofuchi, T. 2022. Ethereum Fraud Detection with Heterogeneous Graph Neural Networks. *ACM-22*.

**[9]** Luu, L., Chu, D-H, Olickel, H., Saxena, P., and Hobor, A. 2016. Making smart contracts smarter. In *Conference on Computer and Communications Security*, pages 254–269. *ACM-16*.

**[10]** Torres, C. F., and Camino, R. (2021). Frontrunner Jones and the Raiders of the Dark Forest: An empirical study of frontrunning on the Ethereum blockchain. In 30th USENIX Security Symposium (USENIX Security 21), 1343-1359.

**[11]** Tsankov, P., Dan, A., Drachsler-Cohen, A., Gervais, A., Buenzli, F., and Vechev, M. 2018. Securify: Practical security analysis of smart contracts. In *Conference on Computer and Communications Security*, pages 67–82. *ACM-18*.

**[12]** Zhuang, Y., Liu1, Z., Qian, P., Liu, Q., Wang, X., and He, Q. 2020. Smart Contract Vulnerability Detection Using Graph Neural Networks. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)*.