Using Graph Neural Networks in Reinforcement Learning with application to Monte Carlo simulations in Power System Reliability Analysis

Øystein Rognes Solheim[1,2,*], Boye Annfelt Høverstad[1,3], and Magnus Korpås[2]

[1]Statnett SF, Norway
[2]Department of Electric Energy, NTNU, Trondheim, Norway
[3]Department of Computer Science, NTNU, Trondheim, Norway
[*]Corresponding author: oysterso@ntnu.no

***Abstract*—This paper presents a novel method for using Graph Neural Networks in combination with reinforcement learning for power reliability studies. Monte Carlo methods are the backbone of such probabilistic power system reliability analyses. Recent efforts from the authors have shown that it is possible to replace Optimal Power Flow solvers with the policies of deep reinforcement learning agents and obtain significant speedups of Monte Carlo simulations while retaining close to optimal accuracies. However, a limitation of this reinforcement learning approach is that the training of the agent is tightly connected to the specific case being analyzed, and the agent cannot be used as is in new, unseen cases. In this paper, we seek to overcome these issues. First, we represent the state and actions in the power reliability environment by features in a graph, where the adjacency matrix can vary from time step to time step. Second, we train the agent by applying a message passing graph neural network architecture to an integrated variant of an actor-critic algorithm. This implies that the agent can solve the problem independently of the power system grid structure. Third, we show that the agent can solve small extensions of a test case without having seen the new parts of the power system during training. In all of our reliability Monte Carlo simulations using this graph neural network agent, the simulation time is competitive with that based on optimal power flow, while still retaining close to optimal accuracy.**

***Index Terms*—Reinforcement learning, power systems, graph neural networks, reliability, Monte Carlo simulation**

I. Introduction

When doing large scale probabilistic power system reliability studies, Monte Carlo (MC) simulations are an indispensable tool. One approach to these types of studies, as we do in this paper, is to do a time-sequential simulation of the power system with stochastic component failures and repair times. At each time step, we then compute the remedial actions needed to continuously reestablish a safe state of the power system. The costs incurred by these control actions can in severe outage scenarios include the cost of energy not served to consumers.

After a component failure has occurred in these time-sequential simulations, it has traditionally been useful to employ optimization methods to find a new and safe state of operation. These methods work by minimizing system costs while finding a configuration of the power generator outputs together with any necessary load shedding, in order to comply with the constraints in the system [1]. Despite their widespread use, typical optimization methods such as the classical Optimal Power Flow (OPF) solver, have several disadvantages. Computationally, solving the full AC-OPF problem is numerically expensive.

A. Related work

Machine learning has gained attention in the power system research community as an alternative to traditional optimization methods, see for example [2]. For reliability, a comprehensive overview is presented in [3]. For example, there have been vast and successful efforts in learning the OPF-solutions from single, independent power system states. In dynamic environments, where both the constraints of the system and the power system states are connected in time as described above, a promising approach is to formulate the problem as a Sequential Decision Problem (SDP), and then use reinforcement learning (RL) to find approximate solutions [4]. Within a power system environment with varying loads and stochastic component outages of lines, generators and transformers, an RL-trained agent would be able to act rationally and provide realistic, remedial actions. This agent could then be incorporated into the MC simulations of power system reliability. We recently proposed a method for doing this, using an RL-trained agent in an MC simulation setting [5].

However, an issue with the approach in [5] is that the agent was trained to act in a specific power grid with few or no possibilities of transferring learning to new grid topologies or altogether different training environments. Ideally, one would like to have an agent that can be applied to any power reliability environment, independent of the number of buses and lines in the power system at hand. A first step in this direction is to train an agent that can be used in a grid expansion planning scenario, where one or a few new lines are added to an existing power grid. This is difficult to achieve

using traditional RL, because the state and actions vectors are dependent on the number of buses and power lines.

In this paper, we address this shortcoming by using Graph Neural Networks (GNNs) [6] within the RL algorithm. GNNs are becoming increasingly popular in machine learning and are applicable when the data structure can be represented by a graph. Some of the most popular variants include Graph Convolutional networks (GCN) [7], Graph Attention Networks (GAT) [8], [9] and Gated Graph Neural Network (GGNN) [10].

Combining reinforcement learning with graph neural networks has also gained considerable momentum in recent years, with applications including traffic flow management, molecular generation and dynamic toll collection [11], [12], [13], [14], [15], [16]. We refer to [17] and [18] for additional examples and references.

B. Research contributions of this paper

In this paper, we present a graph-based extension of the framework developed in [5]. By combining an actor-critic method of reinforcement learning with graph neural networks, we train an agent that can be applied to a power grid irrespective of the grid structure, such as topology and the number of connected loads and generators. Thus, the agent behaves graph-wise, giving individual actions to each node, where the nodes and edges of the graph correspond to the buses and branches in the power system.

An additional benefit of using GNNs is their generalization capabilities. For example, the control actions needed in a power system with thermally overloaded lines constitute a learnable pattern. In real life, the transmission system operator often knows from experience and training what to do when a thermal overload in the power grid occurs, namely changing the injections in areas somewhere near each end of the line, irrespective of the location of the overload. In this respect, graph neural networks seem to be a promising tool for learning such location-independent action patterns.

To this end, we propose an encoder-decoder message passing algorithm [19], where information is distributed iteratively throughout the network. In addition, we integrate the actor and critic with extensive parameter sharing, as in [20]. To express the action of the agent and the critic value, we use an attention based approach as described in [8] and [9], where we include not only the node features but also the edge and graph features.

This paper's three main contributions are thus:

- Establish a graph-based reinforcement learning environment for power system reliability analysis.
- Combine a variant of the TD3 [21] actor-critic method with an integrated attention-based message passing graph convolutional network.
- Demonstrate the speed and accuracy of the method and that it can be successfully applied to incrementally changed grid topologies that are unseen during training.

The remainder of this paper is organized as follows. In Section II, we provide a brief summary of reinforcement learning basics and the power reliability environment. In Section III, we present our model based on an integrated actor-critic with sequential graph network blocks (GN-blocks) [22] and an encoder-decoder architecture built around recurrent attention-based message passing layers. In Section IV, we present our experiment setup together with the results for both the IEEE 24 bus standard reliability case and an extended version with an extra bus, to illustrate the relevance for an expansion planning scenario. Finally, in Section V we sum up the results and discuss possible future approaches.

II. Reinforcement Learning and Power System Reliability

A. Reinforcement learning fundamentals

We model the agent and its behavior in the power system as a fully observable Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, and $\mathcal{R}$ is the reward space. $r_t \in \mathcal{R}$ is the reward the agent receives when taking action $a_t \in \mathcal{A}$ at time t and the environment transitions stochastically from state $s_t \in \mathcal{S}$ to state $s_{t+1} \in \mathcal{S}$ with probability $p(s_{t+1}, s_t, a_t) = p(s_{t+1}|s_t, a_t)$. The parameter γ, $0 \leq \gamma \leq 1$, is the *discount factor*. The return, G_t, is defined as the sum of discounted rewards, $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. The goal of the agent is to maximize the expected return by finding a policy $\pi(s, a) = \pi(a|s)$ where $\pi(s, a)$ is the probability of taking action $a_t = a$ if $s_t = s$. For a deterministic policy, $\pi(s, a) = \pi(s)$.

B. The Power Reliability Environment

In this paper, we build on the power system reliability environment established in [5]. Referring to Fig. 1, the agent interacts with the environment to train its policy, π. At each time step t, the state s_t is fed to the agent from the environment. The policy π calculates an action a_t based on the state s_t. In this power system reliability environment, the agent can a) adjust production up or down at a node and b) adjust load curtailment up or down at a node. A single action a_t contains the set of all adjustments the agent makes at time t.

After executing the action, an intermediate power system state is obtained by solving the power flow equations. The reward r_t is calculated from this intermediate state. The power system is then stepped forward in time, applying any changes to the demand, generator, and component status. The power flow equations are then solved again and the state s_{t+1} is given to the agent. These collections of environment experiences $\{s_t, a_t, r_t, s_{t+1}\}$ are then stored in the replay buffer from which a batch of samples is drawn to improve the policy such that $\pi \rightarrow \pi'$. For simplicity, we have not included the terminal status of the episode in the above.

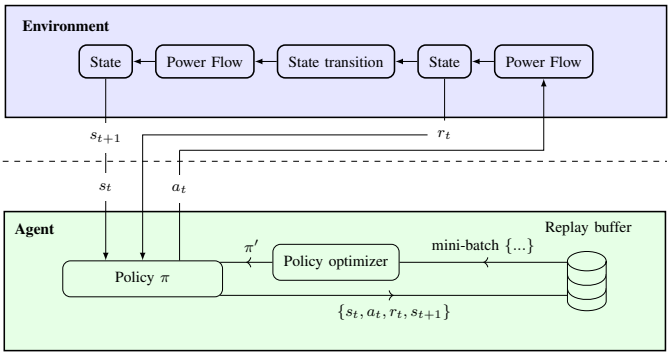

Fig. 1: The interaction of the agent with the environment. Reprinted from [5].

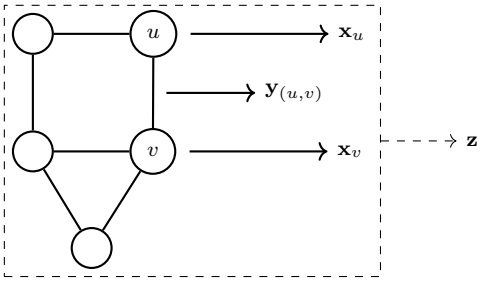

Fig. 2: Node features $\mathbf{X}$, edge features $\mathbf{Y}$ and graph features $\mathbf{z}$.

The power reliability environment is episodic, and each episode is defined by interweaving the RL environment with a Monte Carlo simulation of the power system as follows. Each component (generators, lines, and transformers) has reliability characteristics defined by the mean time to failure and the mean time to repair. Episodes are defined by letting the RL agent operate in a Monte Carlo simulation that runs continuously throughout the RL experiment. In this MC simulation, the failure and repair rates are used in an exponential distribution, from which the state of each component is drawn at each time step.

To the MC simulation into episodes, we define the power system as being in a *safe state of operation* when all the following conditions are met:

- There are no thermal overloads in the system.
- There are no load curtailments in the system.
- The swing bus generator output is within its physical nominal limits.

The first RL episode starts at the beginning of the MC simulation. Subsequently, a new episode starts as soon as the power system returns to a safe state of operation after the end of the previous episode.

Each episode consists of two phases. The second phase starts when the MC simulation brings the system to an unsafe state of operation, either because a) the number of components on outage increases, or b) the available generation capacity is insufficient to cover the system demand. The goal of the agent in phase two is to bring the system back to a safe state of operation as quickly and cost efficiently as possible. The episode ends when this is achieved or when a predefined maximum number of time steps is reached.

The reward structure of the environment is designed to minimize the socioeconomic costs associated with power system outages. We want the agent to curtail the cheapest loads and keep outages as short as possible. The agent receives a negative reward at each time step according to the cost of the curtailed load and generator output. We also add penalties for thermal overloading of transmission lines and swing bus constraint violations. See [5] for further details. Following [5], we make the usual DC-approximations, assuming negligible losses and small angle differences between neighboring buses and unity voltages.

III. Graph Neural Networks

The core contribution of this paper is the use of graph-based representations to express the state of the reinforcement learning environment (i.e. the power system) and actions of the agent. We use these representations in an actor-critic algorithm, formulating the actor and critic as graph neural networks that are trained with a message passing [23] algorithm where information is exchanged between neighboring nodes and updated through neural networks.

A. Actor-Critic Graph Model

Following the exposition in [23], we state that the main idea in graph neural networks is to generate representations that depend on both the structure of the graph and the data the graph may contain. Traditional convolutional neural networks, such as those used in image processing, are based on grid-structured inputs. Recurrent neural networks are specialized for handling structured sequences. However, graph neural networks are based on being permutation invariant, meaning that the arbitrary ordering of the nodes in a graph is without significance.

We represent the power system as a graph, where each bus in the power system is represented by a vertex and the branches correspond to edges between the vertices. Feature vectors on the vertices and edges contain information about the physical power system, such as demand, production and load curtailments (buses), and flows, thermal limits and susceptance (branches), see the appendix for details. In addition, there are "global" graph variables relevant to the graph representation of the entire power system, such as the overall balance between load and production in the system.

Referring to Fig. 2, we consider a simple symmetric directed graph $\mathcal{G}_{\text{STATE}} = (\mathcal{V}, \mathcal{E})$ with a set of nodes $\mathcal{V}$ with cardinality $|\mathcal{V}|$ and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ with cardinality $|\mathcal{E}|$. The *adjacency* matrix $A \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|} = \{a_{ij}\}$ is defined by $a_{ij} = 1$ if $(u_i, u_j) \in \mathcal{E}$, and 0 otherwise. We denote the node features by the column vectors $x_u \in \mathbb{R}^{d_n}$, $\forall u \in \mathcal{V}$, and arrange these into the matrix $X = [... \, x_u \, ...] \in \mathbb{R}^{d_n \times |\mathcal{V}|}$. We denote the edge features by the column vectors $y_{(u,v)} \in \mathbb{R}^{d_e}$, $\forall (u, v) \in \mathcal{E}$,

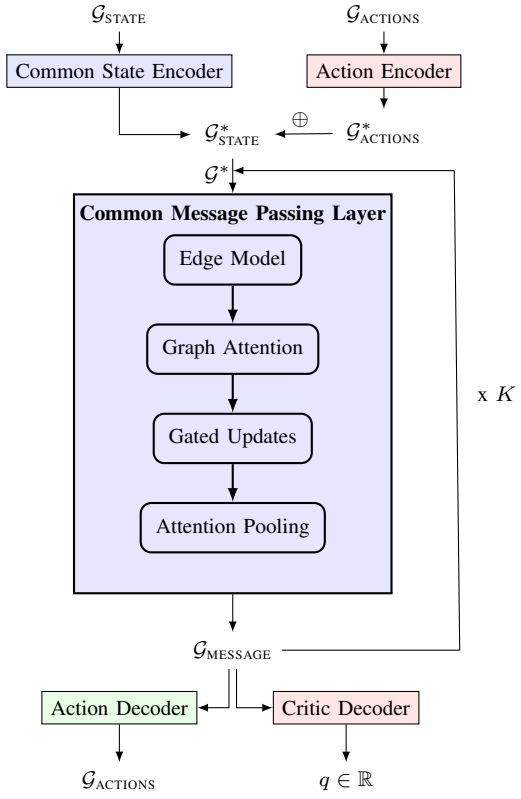

Fig. 3: Integrated Actor-Critic network architecture

which are collected into the matrix $Y = [... \; y_{(u,v)} \; ...] \in \mathbb{R}^{d_e \times |\mathcal{E}|}$. We denote the graph features by $z \in \mathbb{R}^{d_g}$. In addition, we specifically define the action graph $\mathcal{G}_{\text{ACTIONS}}$ as the graph with the same adjacency matrix A but with node features $X_{\text{ACTIONS}} \in \mathbb{R}^{d_a \times |\mathcal{V}|}$ (no edge or graph features).

The actor-critic class of algorithms for reinforcement learning provides us with a method of obtaining a) the expected discounted sum of rewards when taking a specific action in a given state (critic value) and b) the action that maximizes this value. In our graph neural network approach, this critic value corresponds to a global evaluation of the current state of the graph given a specific action. Similarly, the action of the agent is expressed as a graph where the output node features are directly related to what is enforced in the physical system. Whenever the electrical connectivity in the system changes due to a line failure, for example, the adjacency matrix and the state and action graph also change.

Fig. 3 shows the overall architecture of our message passing model. More specifically, our model combines an encoder-decoder structure with a parameter sharing architecture for the actor and critic graph neural networks. This integrated actor-critic architecture is inspired by what is done in [20]. In this integrated actor-critic network architecture, the state encoder and message passing layer are shared by the actor and the critic. When the actor-critic model is given a graph state $\mathcal{G}_{\text{STATE}}$ as input (upper left part of figure), the model outputs a graph representation of the action, $\mathcal{G}_{\text{ACTIONS}}$. When the model is given both a graph state $\mathcal{G}_{\text{STATE}}$ and action $\mathcal{G}_{\text{ACTIONS}}$ as inputs, the model outputs a critic value $q \in \mathbb{R}$. In this latter evaluation, the node features of $\mathcal{G}^*_{\text{ACTIONS}}$ are added elementwise to the node features of the state encoded graph, $\mathcal{G}^*_{\text{STATE}}$. The resulting graph $\mathcal{G}^*$, with edge and graph features unchanged from $\mathcal{G}^*_{\text{STATE}}$, is then passed on to the K common message passing layers. The arrows within the common message passing layer do not represent the data flow but illustrate the sequential nature of the operations.

In the following we describe the different elements of the model in more detail.

B. Actor-Critic Encoder

Referring to Fig. 3, the actor and critic share a common state encoder. We define the common state encoder multilayer perceptrons (MLPs) as $\phi^*_{\text{n}} : \mathbb{R}^{d_n} \to \mathbb{R}^{d^*_n}$, $\phi^*_{\text{e}} : \mathbb{R}^{d_e} \to \mathbb{R}^{d^*_e}$ and $\phi^*_{\text{g}} : \mathbb{R}^{d_g} \to \mathbb{R}^{d^*_g}$ such that

$$\begin{aligned} X^*_{\text{STATE}} &= \phi^*_{\text{n}}(X), \\ Y^*_{\text{STATE}} &= \phi^*_{\text{e}}(Y), \\ z^*_{\text{STATE}} &= \phi^*_{\text{g}}(z). \end{aligned} \tag{1}$$

Together with the adjacency matrix, A, this defines $\mathcal{G}^*_{\text{STATE}}$, which is used as the input to the common message passing layer for the actor.

We want the encoded embedding of X to be of the same dimension as that of the encoded embedding of X_{ACTIONS}. Thus, for the action encoder, we define the MLP $\phi^*_{\text{a}} : \mathbb{R}^{d_a} \to \mathbb{R}^{d^*_n}$ such that

$$X^*_{\text{ACTIONS}} = \phi^*_{\text{a}}(X_{\text{ACTIONS}}), \tag{2}$$

which, together with A, defines $\mathcal{G}^*_{\text{ACTIONS}}$. Note that X^*_{STATE} and X^*_{ACTIONS} are of the same size. We define the graph $\mathcal{G}^*$ by the adjacency matrix A and features

$$\begin{aligned} X^* &= X^*_{\text{STATE}} \oplus X^*_{\text{ACTIONS}}, \\ Y^* &= Y^*_{\text{STATE}}, \\ z^* &= z^*_{\text{STATE}}, \end{aligned} \tag{3}$$

where $\oplus$ denotes elementwise addition.

C. Common message passing layer

A key idea in many graph convolutional algorithms, is to update a hidden embedding $\mathbf{h}^k_u$ for each node $u \in \mathcal{V}$ with information from the graph neighborhood of u, $\mathcal{N}(u)$, at each iteration $k = 1, \ldots, K$ of the message passing scheme. This can be formalized as [23]

$$\begin{aligned} \mathbf{h}^k_u &= \text{UPDATE}^k \left(\mathbf{h}^{k-1}_u, \text{AGGR}^k(\{\mathbf{h}^{k-1}_v, \forall v \in \mathcal{N}(u)\}) \right) \\ &= \text{UPDATE}^k \left(\mathbf{h}^{k-1}_u, \mathbf{m}^k_{\mathcal{N}(u)} \right). \end{aligned} \tag{4}$$

Here, UPDATE^k and AGGR^k are arbitrary differentiable functions, and $\mathbf{m}^k_{\mathcal{N}(u)}$ is defined as the *message function* that represents the aggregation of neighborhood information at iteration step k. At each step, the information from the previous step's embedding $\mathbf{h}^{k-1}_u$ is updated by the message $\mathbf{m}^k_{\mathcal{N}(u)}$ to form the new embedding $\mathbf{h}^k_u$. The aggregation function AGGR is defined on a set, that is, a neighborhood $\mathcal{N}(u)$ of node u, and is thus

permutation invariant and independent of node ordering. This function could be a summation, max, or mean, but it could also be a more complex aggregation based on, for example, an attention mechanism.

Because essential information about the power system is related to its branch features, we would like the graph neural networks in our models to also include edge features in the message passing scheme. To accomplish this, we use a popular neural network technique, the attention mechanism [24], which was first developed for graphs in [8] and further improved in [9]. Instead of weighting each neighbor equally, as in mean aggregation, the attention mechanism makes it possible to assign individual weights to each of the neighborhood nodes based on the feature values. In the present formulation, we also include the edge feature $\mathbf{h}_{(u,v)}$ between nodes u and v. However, the graph attention mechanism with edge features can not include self loops because the edge features are not defined on such constructs. Thus, updating the node features in (15) cannot include the node itself. To overcome this, we also include a recurrent step.

The inclusion of recurrence has an additional benefit. As stated in [23], one problem that might arise after several rounds of message passing is *over-smoothing* [23] which means that all representations tend to approach each other as the message passing iterator k increases. One solution to this problem is to introduce skip-connections where the embedding of an iteration is made up of a concatenation of the previous embedding and the current output. This method can be further refined, as we do in this paper, by using a recurrent network, where implicitly not only the last step is used for updating the embeddings, but the model learns how much it should bring forward from all previous message passing steps. This approach was described in [10], where a GRU cell [25] was used. We follow this recurrent path but use a simple recurrent cell instead of a GRU cell.

Furthermore, our model architecture is inspired by the generalized updating scheme in [23] and the use of GN-blocks, as in [22]. In line with both these approaches, we define the following sequential updating of edge $\mathbf{h}^k_{(u,v)}$, node $\mathbf{h}^k_u$ and graph features $\mathbf{h}^k_{\mathcal{G}}$ in the common message passing layer:

$$\mathbf{h}^k_{(u,v)} = \text{UPDATE}_{\text{edge}}(\mathbf{h}^{k-1}_{(u,v)}, \mathbf{h}^{k-1}_u, \mathbf{h}^{k-1}_v, \mathbf{h}^{k-1}_{\mathcal{G}}) \tag{5}$$

$$\begin{aligned} \mathbf{m}^k_{\mathcal{N}(u)} &= \text{AGGR}_{\text{node}}\left(\mathbf{h}^{k-1}_v, \mathbf{h}^k_{(u,v)}, \mathbf{h}^{k-1}_{\mathcal{G}} \quad \forall v \in \mathcal{N}(u)\right) \\ \mathbf{h}^k_u &= \text{UPDATE}_{\text{node}}(\mathbf{h}^{k-1}_u, \mathbf{m}^k_{\mathcal{N}(u)}) \end{aligned} \tag{6}$$

$$\mathbf{h}^k_{\mathcal{G}} = \text{UPDATE}_{\text{graph}}(\mathbf{h}^k_u, \mathbf{h}^k_{(u,v)}, \mathbf{h}^{k-1}_{\mathcal{G}} \ \forall u \in \mathcal{V}, (u,v) \in \mathcal{E}) \tag{7}$$

We initialize the hidden features of the recurrent cells using the features of graph $\mathcal{G}$ where we for simplicity use $\mathcal{G}$ for both $\mathcal{G}^*_{\text{STATE}}$ and $\mathcal{G}^*$:

$$\begin{aligned} \mathbf{h}^0_u &= x_u, \ \forall u \in \mathcal{V}, \\ \mathbf{h}^0_{(u,v)} &= y_{(u,v)}, \ \forall (u,v) \in \mathcal{E}, \\ \mathbf{h}^0_{\mathcal{G}} &= z. \end{aligned} \tag{8}$$

In the following, we describe the steps in (5), (6), and (7).

1) Edge updates via MLPs: To update the edge features, we create a simple edge model, as in [26], by defining a candidate edge feature $\hat{\mathbf{h}}^k_{(u,v)}$ by

$$\hat{\mathbf{h}}^k_{(u,v)} = \phi^k_e\left(\mathbf{h}^{k-1}_u \parallel \mathbf{h}^{k-1}_v \parallel \mathbf{h}^{k-1}_{(u,v)} \parallel \mathbf{h}^{k-1}_{\mathcal{G}}\right), \tag{9}$$

where ϕ^k_e is an MLP and $\parallel$ is the concatenation operator. To obtain the final edge representation, we apply a simple recurrent cell:

$$\mathbf{h}^k_{(u,v)} = \text{RNN}(\mathbf{h}^{k-1}_{(u,v)}, \hat{\mathbf{h}}^k_{(u,v)}), \tag{10}$$

where the hidden state $\mathbf{h}^{k-1}_{(u,v)}$ of the recurrent cell is the output of the previous step. Generally, $\mathbf{h}^k_{(u,v)}$ is not equal to $\mathbf{h}^k_{(v,u)}$.

2) Node updates via multiheaded graph attention: The attention score $e^{i,k}_{u,v}$ between node u and v for attention head i at iteration level k is calculated by a scoring function

$$e^{i,k} : \mathbb{R}^{d^{k-1}_n} \times \mathbb{R}^{d^{k-1}_n} \times \mathbb{R}^{d^k_e} \to \mathbb{R} \tag{11}$$

defined by

$$e^{i,k}_{u,v} = e^{i,k}(\tilde{\mathbf{h}}^{k-1}_u, \tilde{\mathbf{h}}^{k-1}_v, \mathbf{h}^k_{(u,v)}) = \mathbf{a}^T \cdot \mathbf{w}^k, \tag{12}$$

where $\mathbf{a}$ is a trainable vector, T means transposition and $\mathbf{w}^k$ is defined by

$$\mathbf{w}^k = \text{LeakyReLU}([W^{i,k}_{\text{r}}\tilde{\mathbf{h}}^{k-1}_u \parallel W^{i,k}_{\text{s}}\tilde{\mathbf{h}}^{k-1}_v \parallel W^{i,k}_{\text{e}}\mathbf{h}^k_{(u,v)}]). \tag{13}$$

Here, d^{k-1}_{n} and d^k_{e} are the dimensions of the node and edge features at levels $k-1$ and k respectively, $\tilde{\mathbf{h}}^{k-1}_u = \mathbf{h}^{k-1}_u \parallel \mathbf{h}^{k-1}_{\mathcal{G}}$ is the concatenation of node features with graph features at bus u, LeakyReLU is the elementwise activation function as defined in [27], and $W^{i,k}_{\text{r}}$, $W^{i,k}_{\text{s}}$ and $W^{i,k}_{\text{e}}$ are trainable weight matrices associated with attention head i, iteration level k, and receiver and sender nodes, and edges, respectively.

These attention scores are then normalized using the softmax function, providing the attention coefficients $\alpha^{i,k}_{u,v}$:

$$\alpha^{i,k}_{u,v} = \text{softmax}_v(e^{i,k}_{u,v}) = \frac{\exp(e^{i,k}_{u,v})}{\sum_{v' \in \mathcal{N}(u)} \exp(e^{i,k}_{u,v'})}. \tag{14}$$

The $\text{AGGR}_{\text{node}}$ function in (6) is then defined through the multi-head graph attention mechanism

$$\mathbf{m}^k_{\mathcal{N}(u)} = \Big\Vert_{i=1}^{N^k_{\text{h}}} \sigma\left(\sum_{v \in \mathcal{N}(u)} \alpha^{i,k}_{u,v} W^{i,k}_{\text{s}} \tilde{\mathbf{h}}^{k-1}_v\right), \tag{15}$$

where $\sigma(x)$ is an elementwise nonlinear activation function, and $\|_{i=1}^{N^k_{\text{h}}}$ denotes the concatenation of the attention heads $i = 1, \ldots, N^k_{\text{h}}$.

Note that in 13, we use the recently updated edge features $\mathbf{h}^k_{(u,v)}$ when calculating the coefficients. The model also includes trainable bias vectors. For simplicity, they are excluded from this exposition.

We finally define the node updating function in (6) by

$$\mathbf{h}^k_u = \text{RNN}(\mathbf{h}^{k-1}_u, \mathbf{m}^k_{\mathcal{N}(u)}), \tag{16}$$

where the hidden state $\mathbf{h}^{k-1}_u$ of the recurrent cell is the output of the previous step.

3) Graph updates via attention pooling: To construct an embedding for the graph features for each message passing step k we use a soft attention mechanism, as in [10], for both the node and edge features. First, we define a pooling over the nodes by defining

$$\mathbf{h}^k_{\mathcal{G},\mathrm{n}} = \sum_{u\in\mathcal{V}} \tilde{\alpha}^k_u \, \mathbf{h}^k_u, \tag{17}$$

where the attention coefficients $\tilde{\alpha}^k_u$ are calculated by first applying the real-valued scoring function defined by an MLP $\tilde{\phi}^k_{\mathrm{n}} : \mathbb{R}^{d^k_{\mathrm{n}}} \to \mathbb{R}$ and then normalizing the coefficients using the softmax function:

$$\tilde{\alpha}^k_u = \frac{\exp(\tilde{\phi}^k_{\mathrm{n}}(\mathbf{h}^k_u))}{\sum_{v\in\mathcal{V}} \exp(\tilde{\phi}^k_{\mathrm{n}}(\mathbf{h}^k_v))}. \tag{18}$$

Similarly, we define a graph attention mechanism for the edge features:

$$\mathbf{h}^k_{\mathcal{G},\mathrm{e}} = \sum_{(u,v)\in\mathcal{E}} \tilde{\alpha}^k_{(u,v)} \mathbf{h}^k_{(u,v)}, \tag{19}$$

where the coefficients $\tilde{\alpha}^k_{(u,v)}$ for the edge features are given by

$$\tilde{\alpha}^k_{(u,v)} = \frac{\exp(\tilde{\phi}^k_{\mathrm{e}}(\mathbf{h}^k_{(u,v)}))}{\sum_{(u',v')\in\mathcal{E}} \exp(\tilde{\phi}^k_{\mathrm{e}}(\mathbf{h}^k_{(u',v')}))}, \tag{20}$$

and $\tilde{\phi}^k_{\mathrm{e}} : \mathbb{R}^{d^k_{\mathrm{e}}} \to \mathbb{R}$ is an MLP with input size d^k_{e}.

To combine the pooling embeddings with the graph features, we define another MLP $\tilde{\phi}^k_{\mathcal{G}} : \mathbb{R}^{d^k_{\mathrm{n}}+d^k_{\mathrm{e}}+d^{k-1}_{\mathrm{g}}} \to \mathbb{R}^{d^k_{\mathrm{g}}}$ such that the $\mathrm{UPDATE}_{\mathrm{graph}}$-function in (7) is defined through the candidate representation $\hat{\mathbf{h}}^k_{\mathcal{G}}$

$$\hat{\mathbf{h}}^k_{\mathcal{G}} = \tilde{\phi}^k_{\mathcal{G}}(\mathbf{h}^k_{\mathcal{G},\mathrm{n}} \parallel \mathbf{h}^k_{\mathcal{G},\mathrm{e}} \parallel \mathbf{h}^{k-1}_{\mathcal{G}}) \tag{21}$$

and the final recurrent output

$$\mathbf{h}^k_{\mathcal{G}} = \mathrm{RNN}(\mathbf{h}^{k-1}_{\mathcal{G}}, \hat{\mathbf{h}}^k_{\mathcal{G}}). \tag{22}$$

D. Actor-Critic decoder

As illustrated in Fig. 3, both the actor and critic parts of the model have an MLP decoder after the graph convolutional layers to produce the final outputs.

The final output of the actor network is a graph $\mathcal{G}_{\mathrm{ACTIONS}}$ with a structure similar to $\mathcal{G}_{\mathrm{STATE}}$ and with node features $\mathbf{x}^{\mathrm{a}}_u \in \mathbb{R}^{d_{\mathrm{a}}}$ where d_{a} is the number of actions per node. Thus, we define the output of the actor by an MLP ϕ_{a} that takes the node features of the final convolutional layer K as the input, such that

$$\mathbf{x}^{\mathrm{a}}_u = \phi_{\mathrm{a}}(\mathbf{h}^K_u) \tag{23}$$

and $X_{\mathrm{ACTIONS}} = [...\mathbf{x}^{\mathrm{a}}_u...] \in \mathbb{R}^{d_{\mathrm{a}}\times|\mathcal{V}|}$.

Analogously, we define the output of the critic as a value $q \in \mathbb{R}$ resulting from an MLP ϕ_{c} that uses the graph features of the final convolutional layer as an input:

$$q = \phi_{\mathrm{c}}(\mathbf{h}^K_{\mathcal{G}}). \tag{24}$$

E. Dual critic, delayed updated and loss function

Let $\theta_{\mathrm{se}}, \theta_{\mathrm{ae}}, \theta_{\mathrm{mpl}}, \theta_{\mathrm{ad}}, \theta_{\mathrm{cd}}$ be the neural network parameters associated with the state encoder, action encoder, message passing layer, action decoder, and critic decoder, respectively. The actor $\pi_{\theta_{\mathrm{a}}}$ has the parameters $\theta_{\mathrm{a}} = \theta_{\mathrm{se}} \cup \theta_{\mathrm{mpl}} \cup \theta_{\mathrm{ad}}$. As in the TD3 algorithm, we define a dual critic network by defining two separate pairs of action encoders and critic decoders. Each pair of action encoder and critic decoder work together with the common state encoder and common message passing layer to give two different critic value outputs $q1, q2 \in \mathbb{R}$. Thus,

$$\begin{aligned} \mathcal{G}_{\mathrm{ACTIONS}} &= \pi_{\theta_{\mathrm{a}}}(\mathcal{G}_{\mathrm{STATE}}), \\ q_j &= Q_{\theta^j_{\mathrm{c}}}(\mathcal{G}_{\mathrm{STATE}}, \mathcal{G}_{\mathrm{ACTIONS}}) \quad j = 1, 2, \end{aligned} \tag{25}$$

where $\theta^i_{\mathrm{c}} = \theta_{\mathrm{se}} \cup \theta^i_{\mathrm{ae}} \cup \theta_{\mathrm{mpl}} \cup \theta^i_{\mathrm{cd}},\ i = 1, 2$. We denote the integrated actor-critic network IAC, as in [20], by IAC_θ where $\theta = \theta_{\mathrm{a}} \cup \theta^1_{\mathrm{c}} \cup \theta^2_{\mathrm{c}}$. As is standard in these types of training methods, we define a target network $\mathrm{IAC}_{\theta'}$ to stabilize training, where θ' are slowly updated copies of θ using Polyak updating [28].

When training the integrated actor-critic neural networks, we have only one training step, as we do not train the actor and critic separately. To train the IAC we sample tuples $\{g_i, a_i, r_i, t_i, g_{i+1}\}$ with mini-batch size N from a replay buffer $\mathcal{D}$, where g_i and g_{i+1} are the graph states $\mathcal{G}_{\mathrm{STATE}}$ before and after an action $a_i = \mathcal{G}_{\mathrm{ACTIONS}i}$, r_i is the reward, t_i is the episode terminal indicator and i is the corresponding replay buffer index.

For each tuple in the sampled minibatch, we define action $\hat{a}_{i+1}$ as the target model output from state g_{i+1} as follows:

$$\hat{a}_{i+1} = \pi_{\theta'_{\mathrm{a}}}(g_{i+1}). \tag{26}$$

We let

$$\hat{q}_i = \min(Q_{\theta^{1\prime}_{\mathrm{c}}}(g_{i+1}, \hat{a}_{i+1}), Q_{\theta^{2\prime}_{\mathrm{c}}}(g_{i+1}, \hat{a}_{i+1})), \tag{27}$$

and we define the target $\hat{y}_i$ as

$$\hat{y}_i = r_i + \gamma(1 - t_i)\hat{q}_i, \tag{28}$$

where γ is the discount factor. We also define the target loss δ_i as

$$\delta_i = (Q_{\theta^1_{\mathrm{c}}}(g_i, a_i) - \hat{y}_i)^2 + (Q_{\theta^2_{\mathrm{c}}}(g_i, a_i) - \hat{y}_i)^2 \tag{29}$$

and the total critic loss as $L(\theta) = \mathbb{E}_{\mathcal{D}}[\delta_i]$.

We further define the actor loss through the expected return:

$$J(\theta) = -\mathbb{E}_{\mathcal{D}}[Q_{\theta'}(g_i, \pi_\theta(g_i))], \tag{30}$$

where we have used the target critic network $Q_{\theta'_{\mathrm{c}}}$ to stabilize training.

Following [20], we define the combined actor and critic loss, $Z(\theta)$, using an adaptive variable $\lambda \in [0, 1]$ as a measure of the accuracy of the critic and controlling the update of the actor:

$$Z(\theta) = L(\theta) + \lambda J(\theta), \tag{31}$$

where λ is updated by

$$\lambda \leftarrow \tau \exp(-L(\theta)^2) + (1 - \tau)\lambda, \tag{32}$$

and $\tau \in (0,1)$ is a hyperparameter. This adaptive updating of the actor replaces the delayed training schedule of the actor, as in TD3. Contrary to [20], we did not find it necessary to include a separate regularizing term for the actor.

F. Prioritized replay buffer

To further improve the training, we use a prioritized replay buffer $\mathcal{P}$ [29]. This means that the sampled tuple from the replay buffer also contains a priority value $p_i = (\delta_i + 1.0e^{-10})^\beta$ which gives a relative priority to the tuple being sampled, with default values for previously unsampled tuples being set to a predefined value p_{def}. $\beta \in [0,1]$ is a hyperparameter defining the strength of the priority mechanism, with $\beta = 0$ indicating uniform sampling. Following [30], we define the new critic loss $\hat{L}(\theta)$ as

$$\hat{L}(\theta) = \frac{\mathbf{w} \cdot \delta}{N}, \tag{33}$$

where δ is the vector $\{\delta_i\}$ and the bias-correcting weights $\mathbf{w} = \{w_i\}$ are calculated by

$$\hat{w}_i = \left(\frac{1}{p_i + 1.0e^{-10}} \right)^\beta, \tag{34}$$

and $w_i = \hat{w}_i / \max_i \hat{w}_i$.

IV. Experiments

In this section, we 1) apply the presented Graph RL algorithm described in Section III to train agents in the power reliability environment described in Section II and [5], and 2) use these agents in Monte Carlo simulations of two different power system test cases. The first test case is the standard IEEE Reliability 24 bus test system (RTS) [31]. All the training of the RL agents was based on this power system case. The second power system test case used in the simulations is an extension of the RTS, where we include an extra bus with a new load attached to it. None of the trained agents experienced this test case topology during training. We compare all the agent-based simulation results with OPF-based Monte Carlo simulations. The goal of this section is to demonstrate the speed, accuracy and generalization capabilities of the trained Graph RL agents. As in [5], we use the Expected Energy Not Served (EENS) reliability index to compare the results.

To study the importance of model size on accuracy and speed, we defined three different agent configurations with respect to the size of the graph neural network model. These models are denoted as small (S), medium (M) and large (L); the specific neural network configurations are listed in Table I.

Ultimately, we want the trained agent to behave rationally in a power system with a topology that is not experienced during training. We hypothesize that the RTS is not large enough and does not contain enough variability to produce enough data to generalize from. Hence, to improve the generalization capabilities of the agent, we additionally trained the agent within an environment with increased variability in load, curtailment costs and maximum generator output. This can be considered as a form of data augmentation [32], and the two different network configurations are denoted as BASE and AUG. This means that in total we trained six different agents using the RTS: Three RL agent model sizes, and for each of these model sizes, we trained the agent with and without additional variability.

As in [5], the implementation was done in the Julia programming language [33], using *GraphNeuralNetworks.jl* [34] as the main library for the graph neural networks and *PowerModels.jl* [35] as the power system framework. All agents were trained on 16 cores of Intel 6148 Xeon processors with 256 GB memory on the Idun HPC cluster [36] using a single V100 NVidia 32 GB GPU card. The MC simulations in Sections IV-D1 and IV-D2 were done using a single core on an AMD Ryzen 7 5800X 8-Core Processor for both the OPF and the agents. GPUs were not used in the MC simulations.

A. Training specifications

We used the Adam optimizer [37] with a learning rate of 0.001 which decays linearly to 0.0001 over 2 000 000 training steps from step 500 000. The β-parameters for the Adam optimizer were kept at default values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$, but we set the epsilon parameter to $\epsilon = 0.01$. Similar to the default parameters of the TD3-algorithm, we modified the target actor $\hat{a}_i$ in (26) by adding normally distributed noise with mean $\mu = 0.0$ and standard variation $\rho = 0.2$ but limited to the range $[-0.5, 0.5]$. In addition, we added target noise with mean $\mu = 0.0$ and standard variation $\rho = 0.05$ to the node, edge, and graph features of the graph states g_{i+1}. We used a discount factor of $\gamma = 0.5$ and the target network soft update parameter was set to 0.995.

The prioritized experience replay buffer was implemented as a circular buffer with a capacity of 1 000 000 and default priorities p_{def} set to 100. To generate training samples, we used a multienvironment configuration with 32 parallel environments, each with its own random seed. Referring to [38], we use a replay ratio of 0.25, that is, for every 32 environment interactions, we perform 8 actor-critic updates, using a mini-batch size of 64.

The interactions with the power reliability environment were started with 50 000 random actions, and subsequently we continued to draw random actions with probability 0.1. As is common for making the agent explorative, we define a linearly decaying normalized Gaussian noise added to the actions, with a final value of $\rho = 0.2$ standard deviations after 950 000 environment interactions. When not drawing random actions, we instead draw noisy actions with a probability of 0.9, and noise-free (greedy) actions with a probability of 0.1. The agent model is saved every 10 000 environment steps, and, as in [5], we run a continuous Monte Carlo simulation with 1 000 simulation years in parallel using the latest saved version of the agent with no exploration noise to evaluate the agent's performance. This means that when reporting the results below, the models used for testing are not necessarily the latest.

For the neural networks, we used the Rectified Linear Unit (ReLU) activation function for the hidden layers, and we used

dropout after each hidden layer. We use the $\tanh$ activation function for all the output layers, except for the critic decoder (24) and the scoring functions in the pooling layers (18) and (20) where we use a linear layer. Table I lists the neural network parameters used for the three different model sizes.

B. Additional noise in environment during training

In order to train the model labelled "AUG", we configured an additional training environment where we did a random permutation of the curtailment costs among the buses. Hence, at each point in time when the load or system state changes, we redistributed the curtailment costs by permuting over the buses. We also added Gaussian noise with $\mu = 0.0$ and $\sigma = 0.1$ to these costs, with a cut-off at 2.0 times the noise-free nominal cost. We did this to ensure that, for any given MW, it was always more expensive to have a swing bus imbalance or thermal overload than curtailment costs. To further increase the variability in line flows and load/production-patterns, we randomized each generator in the power system such that the maximum generation capacity had Gaussian noise added to it with $\mu = 0.0$ and $\sigma = 0.2$. Finally, we performed the same random permutation of the bus loads and add Gaussian noise with $\mu = 0.0$ and $\sigma = 0.2$.

C. Rule-based post-processing of actions

In our initial simulations, we observed that the agents were not always able to balance the load and production in the power system. More specifically, the production at the swing bus sometimes exceeded its physical limits. However, in a DC-system such as this, it is possible to make a rule-based adjustment to agent actions after these actions have been enforced but before the first load flow is solved, see Fig. 1. In a power adequacy reliability setting, we are interested in determining whether the current load can safely be covered by the available generators.

Hence, in this revised environment, we adjust the production upward uniformly at all generators not producing at maximum until the swing bus imbalance (production deviation at swing bus from its physical limits) is zero. If this is not sufficient to remove imbalances, we further adjust the load curtailments uniformly until the swing bus imbalance is zero. The rewards are still calculated after the first load flow is solved, and the agent is given a penalty proportional to the original imbalances in order to ensure that the agent learns to balance the system. In addition, the agent learns how this rule works to avoid possible thermal overloads resulting from the rule itself.

D. Results and Discussion

Having trained the six different agents, we did MC simulations as in [5]. In this section, we compare the results between the RL-based and OPF-based simulations. The simulations were configured with 1500 simulated years with a 5-minute time step. The generator and line failure and repair rates, were as described in the original case description [31].

1) RTS base case: The comparison of the OPF solution with the six agent-based solutions for the RTS base case is summarized in Table II.

We note that the smallest model with no data augmentation, denoted "RL BASE S", has the best performance. Its simulation time was only 60 % of the OPF time, and its system costs was only 1.1 % above the OPF. We also note that this agent can select the cheapest loads to curtail according to their cost rank, as indicated in the curtailment cost column. Comparing these results with those reported in [5] (Table V), the results presented here are more accurate, with the speed being approximately equal to the OPF. One of the largest models, denoted "RL BASE L", is closer to the OPF model in absolute terms when it comes to system sum EENS. This model is also only slightly less accurate when it comes to the total system cost, but it does not choose the loads to curtail according to their rank. For example, this large agent often prefers curtailing load at bus 19 to bus 14, even though the load curtailment at bus 19 is slightly more expensive. The computation times of the two largest models were at or slightly above the computation time of the OPF. In the original RTS, the models with augmented data do not produce faster or more accurate results than the BASE-models. However, as the results in the tables show, the augmented data models always select the least expensive load to curtail.

2) RTS extended topology: Motivated by an expansion planning scenario, we constructed a new extended variation of the RTS by inserting a new bus between buses 21 and 22 with a load equal to the load at bus 19 and with line characteristics equal to the line between buses 21 and 22. The curtailment cost was set to \$4000/MWh. This means that only bus 9 had lower curtailment costs. We reiterate that neither agent experienced this topology during training.

From Table III, we see that the medium-sized model trained on augmented data, denoted "RL AUG M", has the best performance in terms of deviation from the OPF with respect to system curtailment costs. Its simulation time was 65 % of the OPF, and it distributes the load curtailments in accordance to their rank and cost. In addition, we observe that the agents trained on RTS without augmented data do not perform well in this new power system case. For example, none of the agents seem able to assign an adequate amount of load curtailment to the new cheap bus 25. The smallest models are also far from approaching the overall system curtailment of the OPF. We also note that the smallest models have only two message passing layers, which means that their ability to transfer information within the network is limited. This suggests that to be able to generalize and perform well on the RTS extended case, the capacity of the models (both in terms of the number of parameters and architecture) must be of a certain order.

The simulation results also indicate that to be able to generalize to new topologies, the agents must be trained in an environment with sufficient variation in the data. The synthetic variations in load, curtailment cost, and production capacity from one hour to another in the augmented data models provide one such enhanced signalling input from which the

TABLE I: Neural network parameters

Model size	S	M	L
Number of message passing layers K	2	4	4
Number of attention heads N_h^k	1	4	4
Encoder size d_{n}^*, d_{e}^*, d_{g}^*, d_{n}^k, d_{e}^k, d_{g}^k	32	64	128
Hidden layer size in encoder modelsϕ_n^*, ϕ_e^*, ϕ_g^* and ϕ_a^*	32	64	128
Number of hidden layers in ϕ_n^*, ϕ_e^*, ϕ_g^* and ϕ_a^*	2	2	2
Hidden layer size in edge models ϕ_e^k	128	256	512
Number of hidden layers in ϕ_e^k	2	2	2
Hidden layer size in node scoring function $\tilde{\phi}_{\mathrm{n}}^k$	32	64	128
Number of hidden layers in node scoring function $\tilde{\phi}_{\mathrm{n}}^k$	1	1	1
Hidden layer size in edge scoring function $\tilde{\phi}_{\mathrm{e}}^k$	32	64	128
Number of hidden layers in edge scoring function $\tilde{\phi}_{\mathrm{e}}^k$	1	1	1
Hidden layer size in graph attention update function $\tilde{\phi}_{\mathcal{G}}^k$	96	192	384
Number of hidden layers in graph attention update function $\tilde{\phi}_{\mathcal{G}}^k$	1	1	1
Hidden layer size in actor and critic encoder ϕ_{a}, ϕ_{a}, ϕ_{c}	64	256	1024
Number of hidden layers in actor and critic encoders ϕ_{a}, ϕ_{c}	2	2	2
Initial value of adaptive variable λ	1.0	1.0	1.0
Parameter controlling actor update τ	0.01	0.01	0.01
Total number of parameters	139 752	1 596 876	13 189 004

TABLE II: Monte Carlo simulation RTS base case, 1500 simulations for evaluation

Bus	Curt. cost [$/MWh]	OPF EENS [MWh/y]	RL BASE S [MWh/y]	RL AUG S [MWh/y]	RL BASE M [MWh/y]	RL AUG M [MWh/y]	RL BASE L [MWh/y]	RL AUG L [MWh/y]
1	8980	0.0	0.0	0.0	0.0	0.1	0.0	0.0
2	7360	0.0	0.0	0.0	0.0	0.4	0.0	0.0
3	5900	0.0	0.6	0.0	0.0	0.6	0.1	0.1
4	9600	0.4	0.4	0.4	0.4	0.4	0.4	0.4
5	9230	0.0	0.0	0.0	0.0	0.1	0.0	0.0
6	6520	0.0	0.0	0.0	0.1	0.1	0.1	0.0
7	7030	256.7	256.7	256.7	256.7	256.9	257.8	256.7
8	7770	0.0	0.1	0.7	0.0	0.2	0.1	0.0
9	3660	797.0	808.7	797.7	814.2	826.2	805.1	831.7
10	5190	7.3	7.1	7.7	7.6	7.6	7.6	8.3
13	7280	0.0	0.1	0.1	0.1	0.2	0.1	0.0
14	4370	226.3	226.9	282.2	54.4	237.5	53.2	234.2
15	5970	0.0	0.2	0.1	0.1	0.3	0.1	0.0
16	7230	0.0	0.0	0.0	0.0	0.1	0.0	0.0
18	5610	1.2	0.1	1.8	0.9	1.3	0.8	1.1
19	4540	44.7	46.0	52.6	222.5	48.1	220.0	45.1
20	5680	0.1	2.4	0.2	0.6	0.2	0.8	0.3
	Sum system	1333.8	1349.4	1400.2	1357.5	1380.2	1346.2	1378.0
	Overloads [MWh/y]	-	0.1	0.2	0.4	0.1	0.6	0.6
	System curtailment costs [$]	$5.96 * 10^6$	$6.03 * 10^6$ (+1.1%)	$6.26 * 10^6$ (+5.0%)	$6.09 * 10^6$ (+2.1%)	$6.15 * 10^6$ (+3.2%)	$6.05 * 10^6$ (+1.4%)	$6.13 * 10^6$ (+2.8%)
	Model training time [h]		155	133	75	132	112	413
	Simulation time [s]	6516	3904	3911	4352	4344	6426	6960

agent can learn general principles. It is our opinion that the models trained on the augmented data perform well in both the RTS base case and the extended case. Noticeably, agents trained without augmented data are not able to transfer their learning, meaning that they might not be learning what we want them to after all. Additionally, larger neural networks are not necessarily better. In our experience, they are more difficult to train, and the forward evaluation step is more computationally demanding.

Our data augmentation approach is motivated by the fact that the RTS is relatively small. Therefore, a natural extension of the current work would be to train agents in larger power systems more similar in size to real life systems. It is our belief that using a larger power system would reduce the need for data augmentation, as they naturally have a larger inherent variation. It would also be useful to study the model's capacity for transfer learning. The combination of a long initial learning on a general base case with a relatively short retraining on a specific power system also seems to be a viable direction for further investigation.

V. Conclusions and future work

In this paper, we formulated a graph-based actor-critic method to train a reinforcement learning agent within a power system reliability environment. The rationale behind this graph-based approach is twofold. First, we wanted to construct a framework in which there are no *technical barriers* in training an agent in one environment with a specific power

TABLE III: Monte Carlo simulation RTS extended case, 1500 simulations for evaluation

Bus	Curt. cost [$/MWh]	OPF EENS [MWh/y]	RL BASE S [MWh/y]	RL AUG S [MWh/y]	RL BASE M [MWh/y]	RL AUG M [MWh/y]	RL BASE L [MWh/y]	RL AUG L [MWh/y]
1	8980	0.0	0.0	0.1	2.0	1.9	0.5	0.2
2	7360	0.0	0.1	0.1	1.8	2.0	0.5	0.2
3	5900	0.0	0.3	0.1	3.4	3.7	0.9	0.4
4	9600	0.2	0.2	0.2	1.5	1.5	0.5	0.3
5	9230	0.0	0.0	0.0	1.3	1.3	0.3	0.1
6	6520	0.4	0.4	0.4	2.9	2.7	1.0	0.6
7	7030	253.5	253.5	253.6	255.9	255.7	261.1	253.7
8	7770	0.1	0.2	0.4	3.3	3.1	0.9	0.4
9	3660	2893.3	2782.1	2803.0	2932.3	2850.9	2907.9	2874.0
10	5190	8.0	9.3	9.2	28.3	10.8	46.7	12.8
13	7280	0.0	0.1	0.2	4.9	4.7	1.3	0.5
14	4370	278.9	573.6	334.7	269.4	281.6	272.0	280.7
15	5970	0.0	0.1	0.2	5.9	5.6	1.5	0.6
16	7230	0.0	0.0	0.1	1.9	1.8	0.5	0.3
18	5610	0.8	0.1	1.1	7.3	6.8	2.4	0.7
19	4540	47.7	114.8	58.3	1003.2	51.4	1026.1	44.5
20	5680	0.0	3.2	0.2	4.8	2.3	4.8	1.2
25	4000	1033.7	2039.3	1194.6	38.5	1118.6	7.5	1218.6
	Sum system	4516.5	5777.3	4656.4	4568.7	4606.1	4536.4	4689.8
	Overloads [MWh/y]	-	1.4	0.1	1.6	0.1	2.4	0.1
	System curtailment costs [$]	$1.80 * 10^7$	$2.32 * 10^7$ (+29.1%)	$2.20 * 10^7$ (+22.2%)	$1.88 * 10^7$ (+4.7%)	$1.85 * 10^7$ (+2.7%)	$1.87 * 10^7$ (+3.9%)	$1.87 * 10^7$ (+4.0%)
	Model training time [h]		155	133	75	132	112	413
	Simulation time [s]	7251	4251	4097	4773	4744	7486	7968

system and then use this agent as is in a new unseen power system. Second, the agent trained in one system must be able to successfully *apply acquired learning* in another unseen system. As the simulation results show, we obtained both goals. In addition, our proposed method produced faster results than the OPF, while retaining close to optimal results.

This paper presented an algorithm and architecture for using graph neural networks. It has been a challenging task to strike a balance between tuning the hyperparameters and choosing between different model architectures. We do not exclude the possibility that there might be other architectures and model formulations that are more effective and simpler than those presented in this paper. Further study of the role of the different parts and building blocks of the current algorithm would therefore be fruitful. As also noted in [5], the relative strength of the reinforcement learning technique increases with the complexity of the problem. Including the AC-formulation from this perspective would in our opinion be worthwhile, possibly also in conjunction with a model-based RL approach.

APPENDIX

The feature vectors of the nodes, edges and graph are defined as follows.

1) Node features: The node features $x_u \in \mathbb{R}^{12}$ at bus $u \in \mathcal{V}$ consist of the following normalized values

- Capacity-weighted generator production cost
- Ranking of production cost
- Load curtailment cost
- Ranking of load curtailment cost
- 1 or 0, where 1 indicates a generator bus
- 1 or 0, where 1 indicates a load bus
- 1 or 0, where 1 indicates the swing bus
- Sum generator output
- Available generator capacity
- Sum loads
- Sum load curtailments
- Voltage level

2) Edge features: The edge features $y_{(u,v)} \in \mathbb{R}^5$ for $(u, v) \in \mathcal{E}$ consists of the following normalized values:

- Power flow
- Line rating
- Transformer ratio (1 if no transformer)
- 1 or 0, where 1 indicates branch being a transformer
- Line susceptance

3) Graph features: The graph features $z \in \mathbb{R}^9$ consists of the following normalized values

- Total system load curtailment
- Total system generator output
- Total system available generator capacity
- Total system load
- Maximum thermal overload ratio in the system
- Power reliability environment phase
- Absolute value of swing bus production exceeding physical limits
- Swing bus production
- Swing bus available capacity

In addition, as in [5], we increase the failure and repair rates during training to make the agent learn faster by experiencing failures more often.

REFERENCES

[1] Allen J Wood and Bruce F Wollenberg. *Power Generation, Operation, and Control*. Wiley, 1995.

[2] Xiang Pan, Tianyu Zhao, Minghua Chen, and Shengyu Zhang. Deepopf: A deep neural network approach for security-constrained dc optimal power flow, 2020.

[3] Laurine Duchesne, Efthymios Karangelos, and Louis Wehenkel. Recent developments in machine learning for energy systems reliability management. *Proceedings of the IEEE*, 108(9):1656–1676, 2020.

[4] Antoine Marot, Benjamin Donnot, Gabriel Dulac-Arnold, Adrian Kelly, Aïdan O'Sullivan, Jan Viebahn, Mariette Awad, Isabelle Guyon, Patrick Panciatici, and Camilo Romero. Learning to run a power network challenge: a retrospective analysis, 2021.

[5] Øystein Rognes Solheim, Boye Annfelt Høverstad, and Magnus Korpås. Deep reinforcement learning applied to Monte Carlo power system reliability analysis. In *2023 IEEE Belgrade PowerTech*, 2023.

[6] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[7] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[8] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[9] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2022.

[10] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks, 2017.

[11] Francois-Xavier Devailly, Denis Larocque, and Laurent Charlin. IG-RL: Inductive graph reinforcement learning for massive-scale traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, 23(7):7496–7507, jul 2022.

[12] Amit Ranjan, Hritik Kumar, Deepshikha Kumari, Archit Anand, and Rajiv Misra. Molecule generation toward target protein (sars-cov-2) using reinforcement learning-based graph neural network via knowledge graph. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 12, 01 2023.

[13] Paul Almasan, José Suárez-Varela, Krzysztof Rusek, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case. *Computer Communications*, 196:184–194, dec 2022.

[14] Wei Qiu, Haipeng Chen, and Bo An. Dynamic Electronic Toll Collection via Multi-Agent Deep Reinforcement Learning with Edge-Based Graph Convolutional Networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 4568–4574. International Joint Conferences on Artificial Intelligence Organization, 2019.

[15] Jiechuan Jiang, Chen Dun, Tiejun Huang, and Zongqing Lu. Graph convolutional reinforcement learning, 2020.

[16] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation, 2019.

[17] Weiwei Jiang, Jiayun Luo, Miao He, and Weixi Gu. Graph neural network for traffic forecasting: The research progress. *ISPRS International Journal of Geo-Information*, 12(3):100, Feb 2023.

[18] Mingshuo Nie, Dongming Chen, and Dongqi Wang. Reinforcement learning on graphs: A survey, 2023.

[19] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017.

[20] Jiaohao Zheng, Mehmet Necip Kurt, and Xiaodong Wang. Integrated actor-critic for deep reinforcement learning. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 505–518, Cham, 2021. Springer International Publishing.

[21] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[22] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[23] William L Hamilton. *Graph Representation Learning*. Morgan and Claypool, 2020.

[24] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

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

[26] Chi Chen, Weike Ye, Yunxing Zuo, Chen Zheng, and Shyue Ping Ong. Graph networks as a universal machine learning framework for molecules and crystals. *Chemistry of Materials*, 31(9):3564–3572, apr 2019.

[27] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30 th International Conference on Machine Learning*, 2013.

[28] 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, 2019.

[29] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, and Google Deepmind. PRIORITIZED EXPERIENCE REPLAY.

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

[31] IEEE Committee Report. IEEE Reliability Test System. In *IEEE Trans. on Power Apparatus and Systems, PAS-98*, 1979.

[32] Connor Shorten and Taghi Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6, 07 2019.

[33] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing, 2015.

[34] Carlo; Lucibello. GraphNeuralNetworks.jl: A geometric deep learning library for the Julia programming language.

[35] Carleton Coffrin, Russell Bent, Kaarthik Sundar, Yeesian Ng, and Miles Lubin. Powermodels.jl: An open-source framework for exploring power flow formulations, 2018.

[36] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[38] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay, 2020.