Using Graph Convolutional Network and Message Passing Neural Networks for Solving Unit Commitment and Economic Dispatch in a day ahead Energy Trading Market based on ERCOT Nodal Model.

by

Pradnya Sanjay Gaikwad

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2020

Acknowledgments

I would like to express my sincere gratitude to my supervising professor, Dr. Ramez Elmasri who has been a great motivating factor and a constant source of encouragement throughout my masters' research. Without his guidance and excellent foresight this thesis would have only remained a great idea.

I would like to thank my colleague, Yashodhan Kumthekar, who helped me with the research and kept pushing forward to achieve even more results. I am sincerely thankful to Dr. Leonidas Fegaras and Prof. David Levine for giving valuable suggestions and serving on my committee. I am also thankful to Dr. Upa Gupta, who directed us in the initial phases of this project.

My heartfelt thanks to my parents, for their constant support and inspiration. They are the reason that this was possible in the first. I am also thankful for my friends who in some or other way have always had my back.

May 17th, 2020

Abstract

Using Graph Convolutional Network and Message Passing Neural Networks for Solving Unit Commitment and Economic Dispatch in a day ahead Energy Trading Market based on ERCOT Nodal Model.

Pradnya Sanjay Gaikwad, MS

The University of Texas at Arlington, 2020

Supervising Professor: Dr. Ramez Elmasri

Various machine learning applications will pre-process graphical representations into a vector of real values which in turn loses information regarding graph structure. Graph Neural Networks (GNNs) are a combination of an information diffusion mechanism and neural networks, which represent a set of transition functions and a set of output functions. Graph Convolution Network (GCN) is based on the optimized variant of CNN which operates on graph and is a scalable approach for semi-supervised learning on structured graph data. Message Passing Neural Networks (MPNNs) summaries the cohesions between many of the existing Neural Network models for structured graph data. This thesis proves the viability of semi-supervised learning GCN model and supervised learning MPNNs to solve the crucial problems like the Unit Commitment (UC) and Economic Dispatch (ED) for the energy market. Power System Optimizer (PSO), a MILP based solution which simulates energy market accurately, but is extremely reluctant to scale in both time and compute. This thesis aims at representing the complex structure of the energy

network using GNN and training the models to simulate the market with increased flexibility to scale in time and compute.

Table of Contents

Table of Figures

1. Introduction

1.1. Energy trading market

Energy trading markets are exactly like any other commodity markets. Wholesale energy is traded as commodities. Energy prices are driven by the market supply and demand of the energy. Higher demand higher the prices, higher the supply lowers the prices.

Several external factors also affect the supply and demand of the energy. For example, any faults or outages in transmission lines can affect the supply of energy. Consider weather, when temperature is hot more energy is consumed for air conditioning, thus increasing the demand, and decreasing the supply due to transmission losses.

1.2. Role of ERCOT and Power System Optimizer (PSO)

The Electric Reliability Council of Texas (ERCOT) manages the flow of electric power to Texas. It represents about 90% of Texas electric load. It provides a central place where generation is dispatched onto the grid. ERCOT's primary responsibility is Reliability i.e. to match generation with demand and operate transmission system within established limits.

Figure 1. shows the ERCOT's Nodal market design for Texas state. The Nodal market is responsible for dispatching energy to follow the system demand. It also ensures sufficient Capacity is on-line to meet the forecasted demand. ERCOT's congestion management system keeps transmission system operating within limits and avoiding congestion on transmission lines.

As the independent system operator for the region, ERCOT schedules power on an electric grid that connects more than 46,500 miles of transmission lines and 650+ generation units.

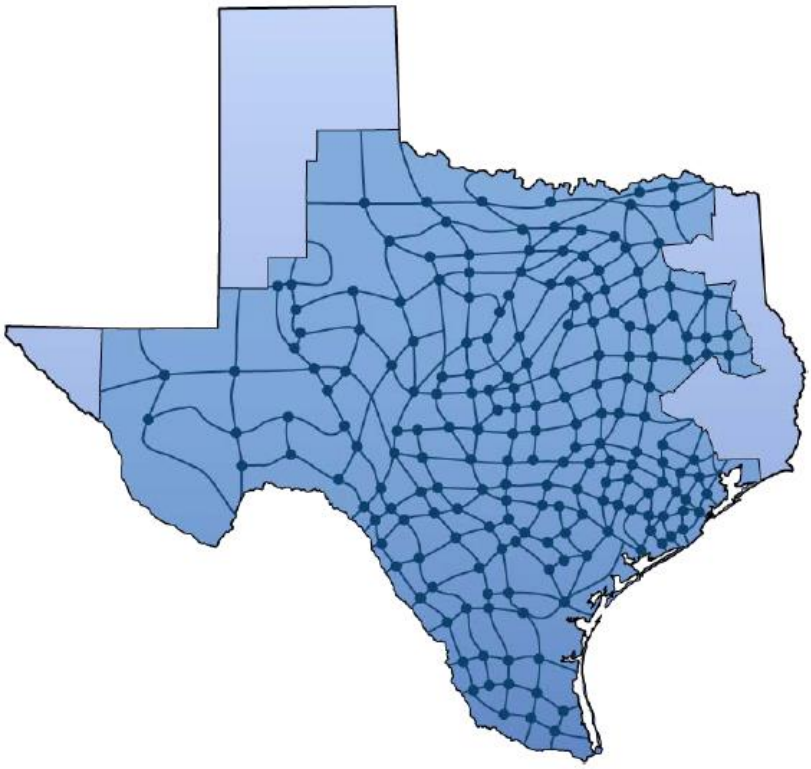

Figure 1: ERCOT Nodal Market Design for Texas

The main problem that ERCOT or any other energy market faces is Economic Dispatch and Unit Commitment. Unit commitment (UC) aims at optimizing the total cost of power generation in a specific period, by forming an adequate scheduling of the generating units. The economic dispatch (ED) problem concerns with finding how much power each unit should generate for a given demand, by minimizing the total operational costs. Unit Commitment and Economic Dispatch are crucial to meet generation

requirements and satisfy the real-time system demand and Day-Ahead market system demand.

ERCOT uses Power System Optimizer (PSO) simulator for solving the above problem.

Power System Optimizer (PSO) is a simulator that takes variables that defines supply and demand of energy and derives (simulates) the market prices for every hour by solving a mixed integer linear problem. It simulates the production cost and supports the modeling of multi-level, nested time intervals that simultaneously optimize energy and ancillary services dispatch and can simulate uncertainties.

PSO's modeling approach is based on the use of Mixed Integer Programming (MIP) algorithms. We can give forecasted values for these variables (weather forecasts, transmission outages prediction) and simulate the market to understand the behavior of market in the future. Energy can be traded at an auction a day before (Day-Ahead Market) or on the very day it is generated and consumed (Real-Time spot market).

The disadvantage of PSO that it is not time efficient and hence cannot be used to simulate market in real time. To simulate next 2 weeks of markets, PSO needs 90 minutes approximately. Another disadvantage of PSO is that it does not scale out. We need to scale out the computation manually meaning, we can ask the simulator to simulate 1 year of data in biweekly partition. Simulator in return runs the 26 biweekly market simulation as the year contains 26 bi-weeks and combines the result. Then, 26 outputs are merged to give final output. It will take approximately 150 mins to finish simulation of 1-year timeframe. We can also run the simulator to simulate 1 year of data without any partition where

simulator will run PSO in 1 big single run which will take 20 hours to finish and sometimes fail due to memory constraints.

1.3. Proposed Work

Graph Neural Networks (GNNs) have developed as a fascinating application to a variety of problems. This thesis focuses on extending the application of GNN to simulate the unregulated energy market and solve the UC ED problem. The UC ED problem is heavily dependent on the structured graph data and GNN gives the ability to process the structured graph data and its spatial properties.

Out of the various models available for GNN, this thesis experiments with supervised and semi-supervised models i.e. Message Passing Neural Network (MPNNs) and Graph Convolution Networks (GCNs). MPNN are famous for considering edge constraints along with node constraints as part of its convolution. CNNs work by extracting local properties of the data with use of local features which are shared across the complete graph. We will be leveraging above features of these models on our dataset to solve the UC ED problem and give better accuracy than the traditional MILP solution.

To represent the energy market problem, we will be using the 7_Bus model that is provided to us by Vistra energy. It is an accurate, but small representation of what Vistra uses, and which is compatible with the PSO software. We are training our GNN models based on 1 month of data generated by PSO on the 7_Bus Model. The accuracy of our GNN models will be computed by calculating the Mean Squared Error (MSE) between PSO output and GNN outputs.

2. Motivation and Background

We saw in chapter 1 the vastness of the energy trading market. There are multiple factors which affect the supply and demand of energy every day. It is thus a challenge to constantly balance out demand and supply of the energy. This section will focus on problems faced by ERCOT and the existing solutions for them.

2.1. Unit Commitment (UC)

One of the most important problem faced in the energy sector is Unit Commitment (UC). UC refers to finding optimal schedule and a production level for power system's each generating unit over a given time period, subject to a given load forecast and spinning reserve constraints [1]. Altogether UC aims to balance demand with supply while optimizing costs.

The total energy generation cost consists of start-up costs, shutdown costs, ramp-up cost, ramp-down cost, fuel costs. Solving UC determines the best possible commitment status for an injector, the start-up/shutdown sequences, and the power dispatch for all available units. The UC optimization problem has the following form [1]:

Total production costs = Fuel cost + Start-up cost + Shutdown cost + Maintenance cost

2.2. Economic Dispatch (ED)

Economic dispatch (ED), playing an important role in the power system operation and planning, has received significant attention in recent years. The purpose of ED is to schedule the committed generating unit outputs. ED also minimizes the operating

cost and meet the load demand of a power system and satisfy all the constraints [2]. The conventional solution methods for ED includes linear programming, Lagrange relaxation, nonlinear programming, quadratic programming, dynamic programming, etc.

As discussed in ERCOT use a market simulator called as PSO, which provides an accurate solution to UC and ED problems. PSO helps to simulate market very accurately, however it fails to scale efficiently. Which makes it un-suitable for Real-Time trading. It takes longer to run, requires to be scaled out manually or even fail due to memory constraints.

3. ERCOT 7-Bus Model and its Components

We introduced ERCOT and ISO in 1.2. Role of ERCOT and Power System Optimizer (PSO). This chapter will describe in detail the 7-Bus Model which we used in our thesis provided by Vistra Energy. The main components of the models include the Areas, Injectors, Branches, Buses, etc. The following diagram show the 7-Bus Model.

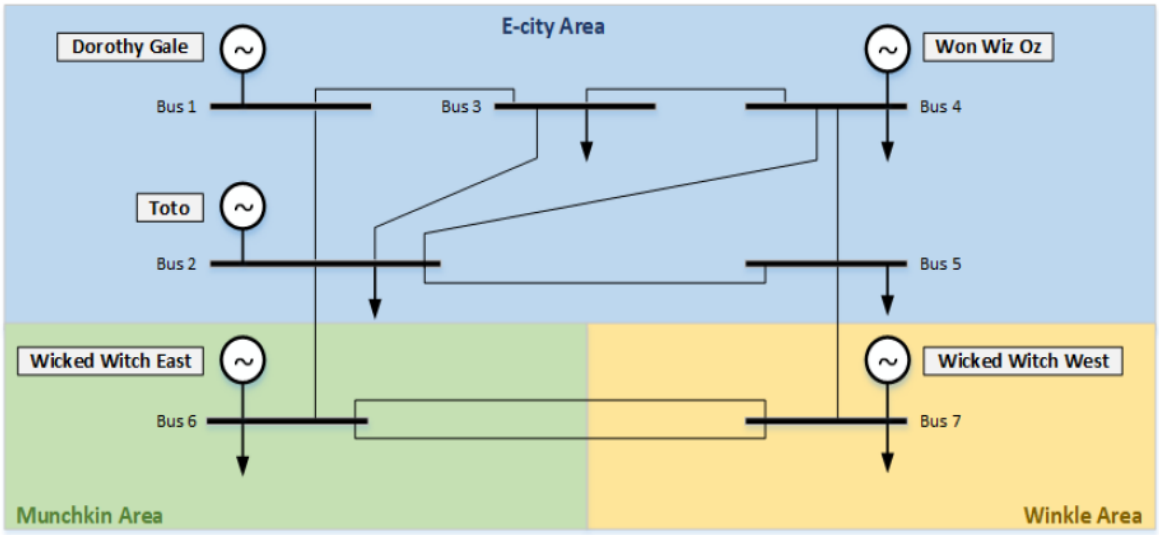

Figure 2 ERCOT 7-Bus Model and its Components

3.1. Areas

End users in the system are consumers that consume the energy produced and paying for it to maintain the balance. There can be millions and millions of consumers that are involved in the system. So, for abstraction these are grouped by geographical regions and each of this region is called “Area”. These can also be classified based on demand into “Load zones”.

An area is a geographical area with preferably non-commercial establishments and households. These are considered as a separate entity as the energy required here varies vastly depending on various constraints like weather, time of the day, events, festivals etc. Thus, these require different set of constraints than Load Zones. As shown in Figure 2 the model has 3 areas:

a. E-City Area

b. Munchkin Area

c. Winkle Area

All the above areas belong to a parent area which is the complete system in this case. The parent area is labelled as 0.

3.2. Injectors

The cycle starts from generators which generate energy to supply to the grid. These come in all shapes and sizes, require different fuels to run, differ in capacities, require to be maintained. Basically, there are a multitude of constraints that can be associated with each generation unit. These are also referred to as injectors since they are injecting the energy into the grid. As shown in Figure 2 the model has 5 injectors:

a. Dorothy Gale

b. Toto

c. Won Wiz Oz

d. Wicked Witch East

e. Wicked Witch West

The constraints for all the injectors are explained in the chapter 4

3.3. Nodes/Buses

Node identifies physical connection to a transmission network. Every Node belongs to an Area. Injector dispatches power to its connected Node. A generator/injector can have a power-balance “Area” different from the area of its Node. An injector is unavailable if Node is not specified. Electrical Nodes are mapped to substations to support data management.

Electrical Nodes are valid only when mapped to branches. Nodes are mapped when used to identify a branch “FrEnode” and “ToEnode”.

3.4. Substation

Substations are used to map Electrical Nodes. Substations belong to a given area and are mapped to a Node at a given time.

3.5. Branches

Branches are the transmission lines which connects one Node to another. As transmission lines have certain constraints similarly branches also have few constraints which are explained in chapter 4.1. Input Data Resistance is used to calculate losses, and Reactance is used to calculate sensitivities of branch flows to changes in power injections and phase-shifter angles. Branch limits are assumed to be physical limits that apply to flow in either direction

4. Data description

4.1. Input Data

4.1.1. AREA ID (ARA_ID)

Field	Type	Description
Area	char	{areas}
Name	char	Long name
ParentArea	char	{areas}
Balance	bit	Power balance constraint enforced for area
External	bit	External-area logic enforced

Notes:

1. Primary Key: (Area). ParentArea passed as text string.
2. The default "system" area (Area = '0') is always defined. It is enforced as a balancing area if Balance is not flagged for any area.
3. Additional areas are always sub-areas of the system area "0". Sub-areas can have power-balance constraints enforced instead of or in addition to balancing constraints on the system area. Balance is automatically enforced for system area if not identified for any sub areas.
4. When an area is flagged as External, it is modeled as having only aggregate load and generation. Injectors mapped to the area are ignored, and generation is the sum of the area load schedule (ARA_SCH_LOD) and interchange (IPR_ID). Note that external generators scheduled to meet internal loads (sometimes referred to as "dynamically scheduled") should be mapped to the internal balancing area that they serve.

5. All areas must be a balance areas or child of a balance area, even if flagged as External.

4.1.2. BRANCH ID (BRN_ID)

Field	Type	Description
Branch	char	{branches}
Name	char	Long name
FrEnode	char	{ electrical nodes } starting node of branch (“from”)
ToEnode	char	{ electrical nodes } ending node of branch (“to”)
Circuit	char	Circuit identifier for multi-circuit branches
Voltage	float	(Kilovolt) Nominal voltage of branch
Resistance	float	(per unit) Branch resistance R
Reactance	float	(per unit) Branch reactance X
NormalLimit	float	(MW) Normal bi-directional limit
CtgLimit	float	(MW) Contingency bi-directional limit
Solve	bit	Branch shift-factors solved and saved in case file
Enforce	bit	Branch constraint is enforced in all periods
Monitor	bit	Branch constraint is monitored when running feasibility analysis
Switchable	bit	Identifies branch that may be opened
Penalty	float	($/MWh) Penalty for flow violation
AngleLimit	float	(rad) Maximum difference in open-circuit voltage angles

HVDC	bit	Branch is high voltage direct current line
CID	char	{ Constraint IDs } name used to add path to set of Constraint IDs

Notes:

1. Primary Key: (Branch). FrEnode and ToEnode passed as text strings.
2. Circuit is optional field used to identify branches of multi-circuit lines (i.e., "parallel" branches). PSO automatically identifies multi-circuit lines as those mapped to the same FrEnode and ToEnode, or mapped to nodes that are electrically equivalent. Branch must be a unique key, even when representing different circuits of the same line.
3. Because Resistance and Reactance are "per unit" values, Voltage is not used. Resistance is used to calculate losses, and Reactance is used to calculate sensitivities of branch flows to changes in power injections and phase-shifter angles.
4. Modeling marginal losses (with option "AddAreaLosses") can have a significant impact on computation performance. Because losses are separately calculated for each line with positive Resistance, performance can be improved by limiting the number of lines with losses (i.e., by using Resistance = 0 for low-voltage lines).
5. Branch limits are assumed to be physical limits that apply to flow in either direction. Limits that differ based on flow direction (such as when used to enforce voltage stability constraints or agreements between neighboring BAs) should be specified using paths.
6. NormalLimit identifies default non-contingency limit (aka long-term or steady-state limit), and CtgLimit identifies default contingency limit (aka short-term or emergency limit). The

default CtgLimit cannot be more restrictive than the default NormalLimit but, when appropriate, more restrictive limits can be specified for specific contingency constraints (PTH_CTG). If NormalLimit = 0, limits are ignored. If CtgLimit = 0, NormalLimit is used as contingency limit.

7. When Solve is flagged, shift factors are solved and stored in a “case file”. This allows all shift factors for the “core topology” to be solved up front where needed in the current or a future model. Branches need shift factors when reported, enforced, monitored, switched, associated with phase-shifter, or used in definition of path, contingency or losses. Once solved, these shift factors can be used in future runs without re-solving. When a branch is not in the case file but is flagged as Solve or is used where shift factors are needed, the case file is resolved.
8. When Enforce is flagged, branch limit is enforced and flows in excess of the limit are penalized. When Monitor is flagged, flows are not calculated and limit is not enforced unless feasibility analysis is used (CYC_SAI). When “RunAnalysis” is flagged, branch flows are calculated and reported, and flows in excess of the limit are reported as violations without penalty. When neither Enforce nor Monitor is flagged, branch violations are not reported.
9. The Switchable flag identifies branches that may be switched to define new topologies. Switchable branches are also identified when associated with an open-branch schedule (SCN_BRN_OPN or SCN_FCP_OPN) or optimized switch branch (SCN_FCP_OPT) in the PS library.

10. Use of Enforce, Monitor and Switchable flags should be minimized as these increase data processing and memory requirements needed to calculate additional sensitivity data for each topology. The Solve flag also increases requirements needed to load case-file data, but has a much smaller impact.
11. A non-zero Penalty identifies a branch-specific penalty cost.
12. A non-zero AngleLimit identifies a branch-specific "MaxPhaseAngleDifference" when Switchable branches are modeled using Flow Canceling Phase Shifters (FCPS). The default value of 1.5 radians should be sufficiently large, but higher angles may be needed for branches in weakly-connected areas, typically with lower voltage. However, AngleLimit should not be any larger than necessary because smaller AngleLimit can improve solver performance.
13. When HVDC branch resistance is identified, associated losses are included in the power-balance constraints (i.e., added to area loads). The impact of HVDC losses cannot be captured using the penalty-factor model.
14. When HVDC branches are added to a model that does not have congestion, flows can vary without having an impact of costs (i.e., solution is "degenerate"). When losses are modeled and HVDC flows can reverse direction, the linearized "loss factor" model can cause HVDC flows to be optimized to create "negative losses". This can be avoided by constraining HVDC flows to be unidirectional. When solving an iterative model (i.e., using "MaxIterations" in table CYC_ID) with HVDC losses, PSO will fix HVDC flows to zero when flow reverses between first and second iterations.

15. When CID is specified, results for branch will be included in all reports that use “cid” as index. Results for branches and paths provided by other reports (that use “pth” as index) only include those flagged as “Enforce” or “Monitor” or that are HVDC. CID can be used to identify a subset of these paths or to specify additional paths that are not enforced or monitored.

4.1.3. CYCLE ID (CYC_ID)

Field	Type	Description
Cycle	char	{ cycles }
Name	char	Long name of cycle
DeltaTime	int	(TimeUnit) Time between start of each horizon of the cycle
LeadTime	int	(TimeUnit) Time between start of the cycle and model StartDate
DecisionTime	int	(TimeUnit) Time in advance of each horizon to begin analysis
OrderTime	int	(TimeUnit) Time in advance of each horizon to announce controls
MipGap	int	MIP convergence tolerance

Notes:

1. Primary Key: (Cycle)
2. In the absence of an explicit cycle model, the default cycle (Cycle = ‘0’) is automatically defined with a single horizon.
3. Each cycle defines a chronological rolling horizon model when a positive value is specified for DeltaTime.

4. LeadTime identifies the amount of time that the start of a cycle (i.e., beginning of the first horizon) leads the model “StartDate” (MDL_ID). LeadTime allows users to stagger the start of cycles when they do not coincide.
5. Horizons of all cycles are solved chronologically based on the “decision time” of each horizon, defined by the starting time of each horizon minus DecisionTime. DecisionTime is also used by the Sequence Model (TS library) to identify schedule data used as the forecast for solving each horizon. Larger values for DecisionTime will reduce access to more-accurate forecasts available closer to real-time operations, but can also give longer time to respond to control orders (e.g., for generators with longer startup time).
6. Except when used to define the solution order of horizons, DecisionTime is rounded to the nearest integer multiple of IntervalLength (MDL_ID) so that the decision time of each horizon can be associated with a time point in the model calendar.
7. OrderTime is the time when control decisions are announced prior to the start of each horizon. This reflects the amount of time following DecisionTime required to process data and issue orders. For example, this is used to identify the earliest period in a horizon that units can startup based on startup time requirements. OrderTime cannot be greater than DecisionTime. If not identified, OrderTime is assumed to be the same as DecisionTime.
8. If an injector’s startup time exceeds available notification time in a cycle, the injector will have restricted availability for startup the cycle’s horizons. If startup time exceeds the sum of notification time and the cycle’s DeltaTime, then the injector cannot be started in the cycle. Insufficient notification time to start an injector can be addressed by applying: (1) a must-run schedule, (2) Stage1 commitment in an earlier cycle, or (3) "LastPeriod" logic

(INJ_STG_CMT) so that commitment decisions later in the horizons can be enforced as Stage1 in the current cycle.

9. Cycles are ordered from longest to shortest DeltaTime (e.g., a day-ahead cycle with 24-hour DeltaTime comes before an hour-ahead cycle with shorter DeltaTime). Cycles with the same DeltaTime are ordered from longest to shortest LeadTime, DecisionTime, and OrderTime. Though cycles may have identical values for DeltaTime, DecisionTime, and OrderTime, all values must be ordered from longest to shortest consistent with the cycle order.
10. The number of horizons solved in each cycle will be sufficient to solve all intervals in the solution horizon as periods in the DeltaTime of a horizon.
11. The values of LeadTime and DecisionTime must be consistent with the amount of time between the first interval (or "MinDate") and the "StartInterval" (or "StartDate") specified for the model (MDL_ID).
12. The definition of cycles must be consistent with the amount of time between the "StopInterval" (or "StopDate") and the "MaxInterval (or "MaxDate") specified for the model (MDL_ID). Additional time is required to solve intervals that are beyond the "DeltaTime" of the last horizons of each cycle. Additional time is also required when the set of solution intervals is not an even multiple of intervals in the DeltaTime of each cycle.
13. Cycle-specific MipGap identifies override of convergence tolerance for mixed-integer programming (MIP) optimization problems specified by option "MipGap". Negative values can be used to set MipGap to zero.

14. MaxSolveTime identifies the maximum solver time to solve each horizon (or to solve each iteration of a horizon when MaxIterations > 0). If not specified, solver time is not constrained and will continue until required convergence tolerance is reached (i.e., “MipGap” for mixed integer models).
15. By default, a horizon is solved once before progressing to the next horizon. This can provide sufficient solution accuracy when prior cycles or earlier horizons of the current cycle identify monitored constraints that should be enforced, power-flow for loss-factor estimates, or coupling FCPS. However, additional iterations may be used to identify (1) additional enforced constraints, (2) more accurate linearized loss factors, (3) more accurate power flow in contingency topologies when using FCPS, or (4) co-optimization of unit commitment and optimized switching.
16. MaxIterations identifies the maximum number of times a horizon is solved. Iterations are halted earlier if not needed. If not specified, MaxIterations is 1.

4.1.4. CYCLE PERIOD ID (CYC_PRD_ID)

Field	Type	Description
Cycle	char	{ cycles }
Period	int	{ Periods }
Length	int	(TimeUnit) Period length

Notes:

1. Primary Key: (Cycle, Period)

2. The definition of periods is mandatory when solving rolling-horizon models. The total length of all periods defines the horizon length of the associated cycle.
3. Periods must be labeled with consecutive integers starting with ‘1’ for each cycle.
4. Periods must be integer multiples of IntervalLength and can be used to aggregate intervals into coarser time steps. This reduces the size of a model and improves the computational performance of optimization problems. When periods are not defined, IntervalLength is used.
5. Within the “DeltaTime” of a cycle (CYC_ID), periods must have an equal Length, and DeltaTime must be an integer multiple of this “characteristic period length”. If the beginning of a cycle were allowed to fall in the middle of a period from a previous horizon, interpolation of previous results could lead to inaccurate and confusing results.
6. The value Length should not decrease with increasing period must be an integer multiple of the characteristic period length.
7. Solved values are mapped by interval using period definitions associated with the horizon. Solutions are passed to subsequent horizons by aggregating these interval values using period mapping associated with each horizon. When solved values exist for some but not all intervals of a period, the values are ignored and the associated period variables are free. Solved values for these intervals will be overwritten with the new solution. Period lengths should be structured so that periods in one horizon are fully contained in periods of the previous horizon where they overlap.

4.1.5. FUEL ID (FUE_ID)

Field	Type	Description
FuelType	char	{ fuel types } alias: fuel
Name	char	Long name
Quantity	char	Fuel quantity (e.g., tons, Mcf, barrels, MMBtu)
Heat	float	(MMBtu/FuelQuantity) Fuel heat content
Cost	float	($/FuelQuantity) Default cost of fuel

Notes:

1. Primary Key: (FuelType)
2. Fuel types may be used to represent different fuels or be used to represent the same fuel used for different purposes (e.g., for different injectors, different locations, etc.).
3. Any unit can be used for Quantity, but it must be used consistently to define all data associated with the fuel type.
4. For compressible fuels (i.e., gas), Quantity and associated Heat and Cost should be based on standard temperature and pressure.
5. Default Cost of fuels is used when unit-specific or pool-specific cost is not specified.

4.1.6. FUEL THERMAL UNIT (FUE_UTH)

Field	Type	Description
FuelType	char	{ fuel types }
ThermalUnit	char	{ thermal units } alias: thermalunit
FuelFactor	float	Adjustment factor applied to total required fuel
Cost	float	($/FuelQuantity) Unit-specific cost of fuel
CostAdder	float	($/FuelQuantity) Addition component added fuel cost

Notes:

1. Primary Key: (FuelType, ThermalUnit).
2. FuelFactor identifies fuel or fuel-mix used by unit. May also be used to calibrate a unit's fuel consumption, but value (or sum of values) should be close to 1. FuelFactor does not change a unit's heat rate but, instead, it scales total fuel consumption required to provide a quantity of heat to identify a unit-specific heat-content for fuel.
3. Unit-specific Cost of fuels overrides default and pool-specific costs if specified.
4. CostAdder is added to Cost or default costs (FUE_ID or SCN_FUE_CST). This cost is added as an independent component even when time varying costs are specified (SCN_FUE_UTH_CST).

4.1.7. HEATCURVE ATT (HCV_ATT)

Field	Type	Description
HeatCurve	char	{ heat curves } alias: curve
TotalHeat	bit	Curve is defined by total heat-rate data (MMBtu/hour)
BaseHeat	float	(MMBtu/hour) Base heat added incremental heat-rate curve
NonConvex	bit	Curve known to be non-convex

Notes:

1. Primary Key: (HeatCurve)
2. Implementation of curve logic (and validation to produce convex curves) parallels implementation for cost curves except than MMBtu is used in place of $. See CV library for details.
3. When used with an incremental heat-rate curve, BaseHeat is added to total heat before identifying incremental-heat-rate points. BaseHeat is ignored when a total heat-rate curve is used; total heat-rate data should already include BaseHeat. “BaseHeat” specified by table INJ_HEA is always added so it can be used as an injector-specific adder.

4.1.8. HEATCURVE POINT (HCV_PNT)

Field	Type	Description
HeatCurve	char	{ heat curves } alias: curve
Point	int	{ points } Vertices of the heat curve
X	float	(MW) Generation level in MW
Y	float	(MMBtu/MWh or MMBtu/hour) Incremental or total heat rate

Notes:

1. Primary Key: (HeatCurve, Point)
2. If "TotalHeat" curve (HCV_ATT), Point = 0 may be specified with X = 0 MW. This will be interpreted as "BaseHeat" for curve.

4.1.9. INJECTOR COMMIT (INJ_CMT)

Field	Type	Description
Injector	char	{ injectors }
MinDispatch	float	(MW) Minimum dispatch when committed
MinOn	float	(hours) Minimum time committed after startup
MinOff	float	(hours) Minimum time de-committed after shutdown
BaseCost	float	($/hour) Cost rate when committed
HotStartCost	float	($) Cost to start unit from hot status
WarmStartCost	float	($) Cost to start unit from warm status
ColdStartCost	float	($) Cost to start unit from cold status
TimeToWarm	float	(hours) Time after shutdown to go from hot to warm status
TimeToCold	float	(hours) Time after shutdown to go from hot to cold status
HotUpTime	float	(hours) Time from startup notification to on-line when initially hot
WarmUpTime	float	(hours) Time from startup notification to on-line when initially warm
ColdUpTime	float	(hours) Time from startup notification to on-line when initially cold

Notes:

1. Primary Key: (Injector)

2. Startup is deemed to occur when an injector is synchronized with the grid (breaker closed), and shutdown when it is de-synchronized (breaker opened). Thus, the time used for ramping between 0 MW and MinDispatch counts towards MinOn requirements and does not count towards MinOff.
3. Commitment constraints are enforced only if non-zero values are specified for MinDispatch or BaseCost; otherwise, there is no reason to ever turn a unit off once committed.
4. Startup and shutdown ramping is based on normal dispatch ramp rates (INJ_ID). Startup and shutdown ramp rates cannot vary by scenario, and are currently based on the lowest non-zero values of each injector.
5. BaseCost is cost when injector is committed at 0 MW. It should include costs at low dispatch levels where these costs are greater than the mimimum incremental cost. BaseCost is added to any other costs identified by a cost curve or heat requirements. When used in combination with heat model (HD library), BaseCost may be used to account for maintenance costs.
6. When commitment constraints are not enforced for specified injector, BaseCost is added to incremental costs by prorating over scheduled "MaxMw" (INJ_ID and/or SCN_INJ_MAX). Costs are ignored when option "IgnoreBaseCostWhenRelaxed" is flagged and commitment constraints are relaxed.
7. Startup costs (HotStartCost, WarmStartCost , ColdStartCost) should include costs associated with subsequent shutdown. Once a unit is started, shutdown costs are "sunk" costs that cannot be avoided.

8. When commitment constraints are relaxed, startup costs are ignored unless option "EnforceStartCostWhenRelaxed" is flagged.
9. The timing of startup orders is based on OrderTime (CYC_ID). Injectors cannot be started until startup-time requirements (HotUpTime, WarmUpTime, ColdUpTime) have been satisfied. An injector's status (hot, warm, or cold) is based on the amount of time since the last shutdown at the time startup orders are issued.
10. Startup-time requirements constrain only the first startup in a horizon. Subsequent startups in the same horizon are constrained by MinOff requirements.
11. ColdUpTime also constrains the amount of "non-spinning" reserves that can be assigned to an injector when not committed. ColdUpTime identifies time lost before an inject can begin ramping from 0 MW, reducing the remaining "activation time" to supply reserves (RSV_ID).

4.1.10. INJECTOR HEATCURVE (INJ_HCV)

Field	Type	Description
Injector	char	{ injectors }
HeatCurve	char	{ heat curves }

Notes:

1. Primary Key: (Injector). HeatCurve passed as text string.
2. "IncHeat" and "BaseHeat" (INJ_HEA) are enforced even when an injector is mapped to a curve (i.e., costs are additive).

4.1.11. INJECTOR ID (INJ_ID)

Field	Type	Description
Injector	char	{ injectors }
Name	char	Long name
Area	char	{ areas } Area used to identify power balance
LoadFlag	bit	Identifies injector that withdraws power
Link	bit	Identifies injector with source and sink at different locations
MaxMw	float	(MW) Max dispatch under any condition (nameplate capacity)
MinMw	float	(MW) Min negative dispatch under any condition
RaiseRR	float	(MW/minute) Maximum rate allowed for increasing power
LowerRR	float	(MW/minute) Maximum rate allowed for decreasing power
RampCapOnly	bit	Ramp rates enforced only on capacity during startup & shutdown
EnergyCost	float	($/MWh) Cost of energy
CostAdder	float	($/MWh) Cost component added to EnergyCost
RampUpCost	float	($/MW) Cost of increasing dispatch (“mileage” cost)
RampDnCost	float	($/MW) Cost of decreasing dispatch (“mileage” cost)

Notes:

1. Primary Key: (Injector). Area passed as text strings.
2. An injector must be a load (non-negative withdrawal of power) or a generator (non-negative injection of power). This also applies to specialized models (such as for energy storage) which must also be classified as load or generator based on an appropriate convention (e.g., determined by source-data convention).
3. Area identifies balancing areas that include the injector. All injectors (except for paired injectors discussed later) must be mapped to a balancing area or the sub-area of a balancing area. When mapped to a sub area, the injector's balancing area is inferred by the parent/child relationships between areas. By default, all injectors are mapped to the default area (Area = '0') which represents the entire system. Power-balance is enforced at the system level if not enforced in sub areas.
4. LoadFlag identifies injectors that are loads and adjusts interpretation of capacity limits, costs, ramp-rates, and other parameters. When a load increases its dispatch, it withdraws additional power from the grid.
5. Link identifies injectors with source and sink at different locations. There are two injector types: (1) "Paired Injectors" (IPR_ID) that identify flows of power from one balance area to another, and (2) "Point-to-Point bids" (PTP_ID) that identify flow of power from one node to another node in the power grid.
6. When an injector is flagged as a load, MaxMw and MinMw are limits on power withdrawals, and RaiseRR and LowerRR are limits on ramping.

7. The value of MaxMw must be positive. If modeling loads that withdraw power from the grid, LoadFlag should be used.
8. The value of MinMw should be zero or negative. Note that the minimum dispatch when committed (and not starting up or shutting down) is a positive number and is enforced by the Unit Commitment Model.
9. A variety of models may require negative MinMw: Storage units (SRG_ID) and models bases on "paired injectors" (IPR_ID), including area interchange, transactions, and financial transmission rights (FTRs). For these models, LoadFlag should not be used. When LoadFlag is combined with negative MinMw, this is interpreted as representing "behind the meter" distributed generation.
10. For loads, RaiseRR is the rate at which the load can increase, and LowerRR is the rate at which the load can decrease. Thus, the RaiseRR capacity of a load contributes to down reserves, and the LowerRR capacity of a load contributes to up reserves (see Reserve Requirements section). Zero values are ignored (i.e., no ramp limit).
11. Ramp rate constraints can significantly increase optimization time. To improve computational performance the flag RampCapOnly identifies that ramp limits are enforced as a reduced MaxMw limit in periods after startup and before shutdown. In many models, the biggest impact of ramp-rate constraints is to limit the capacity immediately available after startup and before shutdown. By replacing constraints that are enforced in every period with a smaller number of constraints, the RampCapOnly flag may improve computational performance while maintaining the most important impacts of ramp-rate constraints.

12. EnergyCost is the cost of power consumption. In general, this can be interpreted as the variable operating and maintenance cost (VOM) when not included separately as fuel cost). EnergyCost is applied even if a cost-curve is also associated with an injector. EnergyCost can be used in combination with other cost models (e.g., heat and fuel costs) to represent other operating costs and benefits (e.g., emissions cost or a production tax credit for generation from renewable sources). Negative price are allowed.
13. CostAdder is added to EnergyCost, and can be used to separately report components of injector costs. This is useful when costs are calibrated (e.g., to match historical data), and users want to preserve original EnergyCost. This is also useful even if energy costs are accurate but offers are different from costs (i.e., not equal to VOM) in market models.
14. RampUpCost and RampDnCost identify cycling costs. Ramp cost is cost per MW/minute of ramping accumulated over time: (ramp cost) x (ramp rate) x (ramp time). For example, $1/MW x (1 MW/minute) x (1 hour) = $60.

4.1.12. INJECTOR INITIAL MW (INJ_INI)

Field	Type	Description
Injector	char	{ injectors }
Mw	float	(MW) initial MW injection (+) or withdrawal (-) by injector
EnforceMw	bit	Identifies that MW=0 is enforced.
TimeInStatus	float	(hours) Time in current commitment status

Notes:

1. Primary Key: (Injector)
2. Mw is used to enforce ramp constraints at beginning of first horizon. If zero and EnforceMw not flagged, initial MW not defined (i.e., initial ramp constraint not enforced).
3. TimeInStatus is used to enforce “MinOn” and “MinOff” (INJ_CMT) at beginning of first horizon. Positive values identify that injector is committed at beginning of first horizon and negative values identify that injector is not committed. If zero, initial status is not defined and does not constrain subsequent status.
4. Initial values are used at the beginning of the first cycle and, in subsequent cycles, initial values are taken from prior cycle. After the first horizon of a cycle, initial values are determined by results from previous horizon of the same cycle.

4.1.13. INJECTOR NETWORK (INJ_NET)

Field	Type	Description
Injector	char	{ injectors }
Node	char	{ nodes } Electrical location (electrical node or aggregate node)
PhysicalArea	char	{ areas } Physical area if different from balance area
LossFactor	exp	(MW/MW) Change in system losses with change in dispatch
IgnoreLoss	bit	Resistive losses ignored in penalty-factor model

Notes:

1. Primary Key: (Injector). PhysicalArea and Node passed as text strings.
2. Node identifies physical connection to a transmission network. Note that a generator can have a power-balance "Area" (INJ_ID) different from the area of its Node (aka "dynamically scheduled" generators). An injector is unavailable if Node is not specified and option "IgnoreUnmappedInjectors" is set.
3. When an injector has no Node mapping, power-flow impact is based on generation or load distribution (PF library) of the injector's "Area" (INJ_ID) or, if specified, PhysicalArea. For example, area mapping can be used to evaluate general power-flow impacts of new generation or load without a detailed interconnection specification.
4. LossFactor is user specified the incremental chance in net system losses with each MW change in injector dispatch (i.e., where is system loss and is injector dispatch) and is used to define penalty factor applied as multiplier of costs of delivered energy and services (i.e., for dispatch, commitment, heat, fuel and reserves).
5. Penalty-factor models assume system losses are already included in area load forecasts and need not be separately calculated (this is typical case where power utilities have better metering of generation than load), and penalty factors are used to reflect the impact that losses should have on dispatch (e.g., "merit order" of generator dispatch). Thus, LossFactor does not affect power and heat output (e.g., maximum limits) or power balance (i.e., the increase in generator output needed to compensate for losses).
6. When also modeling resistive losses (BRN_ID), LossFactor is applied as additional multiplier of costs. Care should be taken to avoid double counting when both are used.

7. IgnoreLoss identifies that resistive losses are ignored when calculating penalty factors. This can be used to override calculated injector loss factors (based on branch resistance and prior power flow) to apply user-specified or to set loss factors to zero for specified injectors. LossFactor and IgnoreLoss do not impact locational marginal prices (LMPs).

4.1.14. INJECTOR STAGE1 COMMITMENT (INJ_STG_CMT)

Field	Type	Description
Injector	char	{ injectors }
Cycle	char	{ cycles } Cycle in which commitment is Stage1 decision
LastPeriod	int	Last period of Stage1 decisions

Notes:

1. Primary Key: (Injector). Cycle is passed as text string.
2. An injector's Cycle is the cycle in which commitment decisions are finalized. In any subsequent cycles, commitment is assumed fixed at results from the identified Cycle (however, results can be overridden by forced outages and failed starts; see CYC_INJ_FOP and INJ_FSP).
3. When any injector is mapped to the null cycle ('0'), validation will be applied to verify that all commit-able injectors are mapped to a cycle.
4. When dispatch decisions after DeltaTime should be assigned as Stage1 decisions, LastPeriod identifies the last period that is Stage1 in each horizon. Prior periods are also

Stage1 decisions or are solved in earlier horizons as Stage1 decisions. Not needed when Stage1 decisions coincide with DeltaTime.

5. Uses standard implementation for stage specification. See INJ_STG_DSP for further description.

4.1.15. LIBRARY REPORT SUPRESS (LIB_RPT)

Field	Type	Description
Library	char	{ libraries } alias: modellibrary
NoInterval	bit	Supress interval results
NoAggHour	bit	Supress aggregate hour results
NoAggDay	bit	Supress aggregate day results
NoAggMonth	bit	Supress aggregate month results
NoAggYear	bit	Supress aggregate year results

Notes:

1. Primary Key: (Library)
2. Suppresses default reporting of results (i.e., “Default” results identified by PSO Results Reports). NoInterval, NoAggHour, NoAggDay, NoAggMonth, and NoAggYear identify results that should not be reported. NoInterval suppresses interval results (i.e., the most-

granular results of a model) of identified Library. NoAggHour, NoAggDay, NoAggMonth, and NoAggYear suppress aggregate results with identified granularity.

3. NoInterval suppresses interval results regardless of interval length. For example, if interval length is = 1 hour, Hourly results are suppressed even if NoAggHour is not flagged.
4. Libraries are identified by their two-character acronym by lower-case letters (e.g., “ed”, “uc”, etc.). Library = 0 can be used to identify all libraries.
5. “Default” results controlled by LIB_RPT. In addition, “standard” are always reported and cannot be suppressed, and “detailed” results are not reported unless requested (see PSO Results Reports).
6. Specific results (including “detailed” reports) are reported by adding report names to the list of options (i.e., “configuration parameters”). For example, “ED_Inj” added to the list of options causes this default report to be written, even if LIB_RPT identifies that “ed” results are suppressed.
7. When writing tabular reports, each field is written to a separate CSV file or Excel tab. Adding “ED_Inj” added to the list of options causes all fields of the table to be written while adding “ED_InjP” causes only the “P” field to be written. Names of table and field reports are case sensitive.
8. Specific aggregate results can be reported by adding aggregate report names to the list of options (e.g., “ED_Inj_d” or “ED_InjP_d”).

4.1.16. MODEL ID (MDL_ID)

Field	Type	Description
Name	char	Long name for model
MajorRelease	int	Major release number
MinorRelease	int	Minor release number
BranchRelease	int	Branch number
TimeUnit	char	Units associated with interval length
IntervalLength	int	(TimeUnit) Length of each interval
MaxInterval	int	Last interval used to specify model data
StartInterval	int	Start interval of model's solution horizon
StopInterval	int	Stop interval of model's solution horizon
MinDate	date	First date used to specify model data
MaxDate	date	Last date used to specify model data
StartDate	date	Start date of model's solution horizon
StopDate	date	Stop date of model's solution horizon

Notes:

1. Primary Key: not applicable (scalar data)
2. Version data is used to ensure model changes and data structures are synchronized and also allows software to be backwards compatible (i.e., it can run older data models). Major and minor releases are identified by MajorRelease and MinorRelease. Previously defined tables

cannot be re-organized without a change in MinorRelease. BranchRelease identifies “beta” releases with prototype or custom implementations.

3. Valid TimeUnit values are “minute”, “hour”, “day” or “week”. Must be specified. Longer values (e.g., month or year) can be approximated by using IntervalLength.
4. IntervalLength is the shortest time increment used by model time step and input data specified by interval or date. To minimize memory and data-processing requirements, IntervalLength should be as large as possible based on model and data. Currently, it is assumed that IntervalLength should not be less than one minute can be specified as an integer multiple of time units.
5. Time data must align with intervals, which can be difficult when using intervals longer than 1 day. Months vary from 28 to 31 days, and years can be 365 or 366 days. As a result, the beginning and end of months and years will not be included in the set of valid time points when multi-day or weekly intervals are used. Where values are specified in years (e.g., “DiscountRate” in UE library), it is assumed that 1 year = 365.25 days or 8766 hours. Nevertheless, aggregate values are based on calendar length of each month and year (i.e., considering varying length).
6. Time-varying data must be modeled using either interval data or time points. If interval data is used, then all mapping to time (e.g., by schedules) must be by interval. If time-point data is used, then all mapping to time must use dates. In general, interval data is more appropriate for simple or abstract models with a limited number of intervals. Time-point data is more appropriate for models of real systems, particularly when integrating data from different sources.

7. When interval data is used, intervals are identified (i.e., labeled) by consecutive integers from ‘1’ to MaxInterval, and intervals that are outside of this range are not recognized.
8. When interval data is used, StartInterval and StopInterval identify the first and last interval of the “solution horizon”. Intervals before the StartInterval may be included in the model definition when control decisions are made in advance (i.e., “LeadTime” in table CYC_ID). Intervals after the StopInterval may also be included in the model definition to avoid myopic decisions.
9. If StartInterval and/or StopInterval are not identified, StartInterval is set to 1 and StopInterval is set to MaxInterval.
10. When time-point data is used, dates are identified by MinDate and MaxDate. Dates outside this range are not recognized.
11. When time-point data is used, the maximum interval is calculated based on TimeUnit, IntervalLength, and the difference between MinDate and MaxDate.
12. When time-point data is used, StartDate and StopDate identify the beginning and end of the solution horizon.
13. MinDate identifies the date at the beginning of the first interval even when interval data is used. If not specified, DefaultMinDate is used.

4.1.17. ELECTRICAL NODE ID (NDE_ID)

Field	Type	Description
Enode	char	{ electrical nodes } alias: enode
Name	char	Long name
Busbar	char	{ electrical nodes }
Substation	char	{ substations }
ReportNode	bit	Nodal LMPs and other results should be calculated and reported

Notes:

1. Primary Key: (Enode). Busbar, Substation and Area passed as text strings.
2. Electrical nodes are valid only when mapped to branches. Nodes are mapped when used to identify a branch “FrEnode” and “ToEnode” (BRN_ID) or when indirectly mapped by Busbar.
3. Mapping electrical nodes to same Busbar identifies inoperable zero-impedance connections. This avoids the need to use dummy branches or jumpers to establish connectivity. The combined use of both bus bars and electrical nodes allows users to distinguish between electrical and physical connections. This can also be useful when combining data from datasets that use inconsistent naming or mapping conventions. This can also be useful to track power-flow values of different injectors connected to the same bus.
4. Any node from among a group of electrically-identical nodes may be a Busbar. In general, it will be easier to manage data if bus bars are electrical nodes used as branch terminals

(i.e., branch “FrEnode” or “ToEnode”) except when breakers or transformers are also modeled.

5. Default logic calculates and reports locational marginal prices (LMPs) for injectors but not electrical nodes. The flag ReportNode identifies additional LMPs that are reported.

4.1.18. SCHEDULE TIMEPOINT (SCH_TMP)

Field	Type	Description
Schedule	char	{ schedules }
Time	date	{ time points }
Value	exp	Schedule value associate with time point
Enforce	bit	Identifies that Value=0 should be enforced.

Notes:

1. Primary Key: (Schedule, Time)
2. Time-point schedules are converted to interval schedules using linear interpolation unless “StepChange” is flagged (SCH_ATT). By definition, time points are always at the boundary between intervals (since an interval is the smallest time increment in the model). Thus, every interval is fully contained between time points.
3. A time-point schedule is not defined before the first time point or after the last time point (i.e., the value of the time point values are not extrapolated).
4. When Enforce flag is not used, zero values are ignored. If the Enforce flag is used, it must be specified for all other values in the same schedule that are to be enforced.

5. For tabular data, tabs in Excel are labeled SCH_TMP_TV for Value and SCH_TMP_TE for Enforce and have the following structure:
    a. Columns = Schedule
    b. Rows = Time
    c. Data = Value, Enforce
6. Multiple Excel tabs can be used to specify input data when the option "NumberOfExcelTabs" is identified. Tab names are identified as SCH_TMP1, SCH_TMP2, SCH_TMP3 and so on (or SCH_TMP_TV1, SCH_TMP_TE1).

4.1.19. SCENARIO AREA LOAD (SCN_ARA_LOD)

Field	Type	Description
Scenario	char	{ scenarios }
Area	char	{ areas }
Load	float	(MW) Static fixed load
Enforce	bit	Identifies that Load = 0 should be enforced.
ScaleFactor	float	Factor used to scale Schedule and Sequence
Schedule	char	{ schedules }
Sequence	char	{ sequences }

Notes:

1. Primary Key: (Scenario, Area). Schedule and Sequence passed as text strings.
2. Table identifies fixed active-power loads that are not associated with a specific device and location. When power-flow impacts need to be considered, area loads are distributed based on weighs applied to specified electrical nodes (see STE_NDE).
3. When parent-area mapping exists, sub-area load schedules are aggregated to establish parent area load schedules when parent-area schedules are not identified.
4. All loads must be mapped to a balancing area or the sub-area of a balancing area.

General notes for scenario data:

1. When flagged, Enforce will cause all intervals to have an enforced value. Time-varying values will be used where enforced by Schedule and Sequence. The static scenario value (i.e., Load) is enforced in all other intervals. Time varying Schedule and Sequence values are enforced only in periods which have enforced values in all intervals of the period.
2. The optional parameter ScaleFactor is applied to Schedule and Sequence to define time-varying values when they differ only by a constant scaling factor. ScaleFactor is an attribute of the Schedule and Sequence mapping: When assigned to the default scenario ‘0’, it is applied only to schedules and sequences also associated with the default scenario. Non-default scenarios that do not have a ScaleFactor will be assigned a value of 1.
3. If ScaleFactor is not assigned a value or is assigned the value of 0, then ScaleFactor = 1. If it is desired that ScaleFactor = 0 be used, this can be achieved by setting Enforce = 1 with default value of 0 or null and with no Schedule or Sequence specified.

4. Load (or other appropriate static value), Enforce, ScaleFactor, Schedule or Sequence mapped to the default scenario (Scenario = ‘0’) is used as default data for all scenarios without explicit mapping. When multiple methods are used to map static and/or time-varying data, the following priority identifies data associated with each period:

   Sequence, Schedule, and static data (including any mapping to the default scenario). For example, when both static and time-varying data are specified, Load and/or Enforce identify default values for periods without enforced Schedule or Sequence values. Priority order is as follows:

   a. Sequence of non-default scenario
   b. Sequence of default scenario
   c. Schedule of non-default scenario
   d. Schedule of default scenario
   e. Static value (e.g., Load) of non-default scenario
   f. Static value (e.g., Load) of default scenario
   g. Static default value

   Area load does not have a static default value. Static default values exist for other types of scenario data (e.g., SCN_INJ_MAX).

4.1.20. SCENARIO CYCLE (SCN_CYC)

Field	Type	Description
Scenario	char	{ scenarios }
Cycle	char	{ cycles } Cycle to which scenario is applied
Weight	float	Weight (or probability) of scenario
Reference	bit	Scenario used to define deviations of other scenarios

Notes:

1. Primary Key: (Scenario). Cycle passed as text string.
2. Used to identify scenarios and map scenarios to cycles. Each scenario can be mapped to only one cycle and each cycle must have at least one scenario. A stochastic cycle is defined when multiple scenarios are mapped to the same cycle.
3. The default scenario (‘0’) should not be mapped to any cycle. The default scenario is always defined and is used define shared data applicable to all scenarios. Data is mapped to other scenarios is applied as overrides of default data.
4. The Weight of scenarios mapped to each cycle should sum to one (100%). Scenarios with zero Weight have no direct contribution to the objective function except through constraint violations (e.g., load shedding) and associated penalties. A zero-weight scenario is a contingency scenario required only for reliability.
5. The Reference flag is used to define values of other scenarios of a stochastic cycle as positive or negative deviations. When stochastic reserves are procured explicitly

(CYC_RSV), the reference scenario identifies deviations that must be met by “Up” reserves or “Down” reserves.

4.1.21. SUBSTATION ID (SUB_ID)

Field	Type	Description
Substation	char	{ locations } alias: station
Name	char	Long name
Area	int	{ areas }

Notes:

1. Primary Key: (Substation). Area passed as text string.
2. Optional data used to associate electrical nodes with areas. Physical area mapping can be used to define physical location of injectors, such as when used to identify “dynamically scheduled” generators.
3. Area mapping is used only to organize substations and does not impact area power balance except where used to define injector mapping to area (INJ_ID) or to assign branch losses to areas.

4.2. Output Data

4.2.1. Scenario Area Interval (ED_Ara)

Record	Type	Description
scn	char	{ scenarios }
ara	char	{ areas }
int	int	{ intervals }
Load	float	(MW) fixed area load
Loss	float	(MW) resistive losses
P	float	(MW) dispatch
NetIC	float	(MW) net interchange: (+) export, (-) import (i.e., exports minus imports)
Violation	float	(MW) violation of area power balance constraint
Penalty	float	($Unit) cost of violation
SP	float	($/MWh) shadow price of area power-balance constraint
BalancePrice	float	($/MWh) total shadow price including parent areas
LoadPrice	float	($/MWh) weighted LMP of fixed-load
LoadCost	float	($Unit) fixed-load cost (product of load, price and period length)
SourcePrice	float	($/MWh) weighted LMP of injector MW
SourceRevenue	float	($Unit) injector revenue (product of MW, price and period length)
NetIcCost	float	($Unit) net-interchange cost: (+) export, (-) import

NetIcRevenue	float	($Unit) net-interchange revenue: (+) export, (-) import

Notes:

1. Primary Key: (scn, ara , int)
2. Load is fixed load specified by input data and not affected by Violation (e.g., value does not change with load shedding).
3. P is total dispatch of area injectors (generation minus load). P is zero in external areas.
4. NetIC is sum of flow on paired injectors with source or sink in area, including sub-areas.
5. Loss is additional load added to power-balance constraints when option "AddAreaLosses" is flagged and/or when modeling HVDC branches.
6. SP and BalancePrice are determined by shadow price of power-balance constraints. These are based on the marginal cost to provide an additional MWh from marginal injectors. When interchange transactions are modeled (i.e., using paired injectors), marginal injectors can be outside the area.
7. The additional MWh provided by marginal injectors is balanced by the slack bus and, thus, SP and BalancePrice include the locational impacts of wheeling costs, losses, or binding area power-balance constraints. As a result, SP and BalancePrice can depend on the location of the slack bus. Locational marginal price (LMP) at the reference bus is equal to SP and BalancePrice of area '0' (i.e., the entire system) and at other areas if there are no wheeling costs, losses, or binding area power-balance constraints. SP and BalancePrice do not include impact of binding transmission constraints resulting from flow over the

physical network (i.e., “congestion” from constraints in the PN, PC, and NC libraries), except as included in the slack-bus LMP.

8. LoadPrice is area locational marginal price (LMP) calculated using area load distribution (see PF library) and is not affected by reference-bus location (except for impact on loss factors). This is BalancePrice adjusted for cost of congestion to flow power from slack bus to area loads. LoadCost is product of Load and LoadPrice.
9. SourcePrice is area LMP calculated using distribution of solved injector MW. SourceRevenue is product of injector MW and SourcePrice.
10. NetIcCost is product of SourcePrice, period length, and total MW of paired injectors with source or sink in current area, including sub-areas. Area of SourcePrice should not be confused with “SourceArea” (IPR_ID). SourcePrice is weighted LMP of area injectors and is used as an estimate of marginal generation cost to increase generation for export, and to decrease generation with imports.
11. NetIcRevenue identifies revenues associated with paired-injector imports and exports. Used weighted LMP of injectors in area of origin (for imports) and destination area (for exports). If an external area with no injectors, area distribution of “GenMw” (STE_NDE) is used. Difference in NetIcCost and NetIcRevenue identifies total “profit” from interchange.
12. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
13. Tabular reports are identified by scn and have the following structure:
    a. Columns = ara

b. Rows = int

14. Sparse fields: Violation, Penalty

4.2.2. Scenario Injector Interval (ED_Inj)

Record	Type	Description
scn	char	{ scenarios }
inj	char	{ injectors }
int	int	{ intervals }
Cap	float	(MW) installed dispatch capacity
Max	float	(MW) maximum dispatch capacity
Min	float	(MW) minimum dispatch capacity
P	float	(MW) dispatch
RC	float	($/MWh) change in objective cost for increase in dispatch
Marginal	bit	Injector may be marginal
LMP	float	($/MWh) locational marginal price
LoadPrice	float	($/MWh) load-weighted locational marginal price of balance area
BalancePrice	float	($/MWh) area energy price from power-balance constraints
CostTotal	float	($Unit) Cost from all libraries
CostOfEnergy	float	($Unit) Product of energy and EnergyCost
CostOfAdder	float	($Unit) Product of energy and CostAdder
CostOfRamp	float	($Unit) Product of ramp MW and RampCost

Revenue	float	($Unit) ED “revenue” (product of dispatch, price and period length)
Mileage	float	(MW) total movement over aggregate period
ViolationRR	float	(MW) violation of ramp-rate limit
PenaltyRR	float	($Unit) cost of ramp violation
Up	bit	injector is up (committed or dispatch > 0) (for aggregate reports)
UpCap	float	(MW) installed dispatch capacity; zero if not up (for aggregate reports)
UpMax	float	(MW) maximum dispatch capacity; zero if not up (for aggregate reports)

Notes:

1. Primary Key: (scn, inj, int)
2. Cap is “MaxMw” (INJ_ID), considering only the impact of “InstallDate” and “RetireDate” (INJ_INS) and deration by season (INJ_MAX).
3. Max and Min identify dispatch limits. They are de-rated by fixed-dispatch (SCN_INJ_DSP) or dispatch limits (SCN_INJ_MAX), and are set to zero when off due to scheduled outage, forced outage, failed startup, or unavailable for commitment (un-committed status after Stage1 cycle, unsatisfied “MinDown” requirements, startup time requirements, scheduled de-commitment, failed startup, and maintenance outages).
4. MaxMw identifies capacity associated fixed-dispatch (SCN_INJ_DSP), dispatch limit (SCN_INJ_MAX), or post-Stage1 commitment status (0 if not committed). In contrast to

values reported by ED_Inj, MaxMw identified available dispatch before impact of forced outage (FO) or scheduled outage (SO).

5. When an outage begins or ends in the middle of a period, a de-rated value is identified for MaxMw (to accurately represent available energy). CapMw is de-rated only when installation or retirement occurs in the middle of a period.
6. Marginal identifies injectors that may be marginal-cost resources. Injectors can be marginal when their dispatch is variable (e.g., not on fixed schedule and not after stage-1 dispatch), and whose reduced cost (RC) is zero. This does not guarantee injectors are marginal, but logic should identify all potentially marginal injectors.
7. LoadPrice is load weighted LMP for all load nodes in an injector's balance area (i.e., not smaller area if mapped to a more granular area). BalancePrice is the shadow price of the power-balance constraints. Prices can vary by area when solving a multi-area model (with power-balance constraints for each) or when including power network (modelled by DC power-flow model using PN and PC libraries or by shift-factors using NC library). BalancePrice will be the same for all areas unless solving a multi-area model.
8. LMP, LoadPrice and BalancePrice do not exist for paired injectors and are not reported. Paired-injector prices are based on difference in source-area and sink-area prices, adjusted for congestion cost. Area prices are based on marginal-injector costs. Though similar to area load prices (ED_Ara), changes in paired-injector flow are generally matched by changes in dispatch of marginal injectors and not by changes in distributed area load. Identification and weighting of marginal injectors can be difficult, but the net impact of prices, congestion and wheeling cost is identified by the paired-injector RC.

9. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
10. Tabular reports are identified by scn and have the following structure:
    a. Columns = inj
    b. Rows = int
11. Sparse fields: ViolationRR, PenaltyRR.

4.2.3. Scenario Fuel Area Interval (FD_Ara)

Record	Type	Description
scn	char	{ scenarios }
fue	char	{ fuel types }
ara	char	{ areas }
int	int	{ intervals }
Fuel	float	(FuelQuantity) fuel use
Cost	float	($Unit) cost of fuel
CostOfScaling	float	($Unit) cost added by scaling
CostOfAdder	float	($Unit) cost added by CostAdder

Notes:

1. Primary Key: (scn, fue, ara, int).
2. Tabular reports are identified by scn and fue, and have the following structure:
    a. Columns = ara

b. Rows = int

3. Compound tabular reports are identified by scn and have the following structure:
    a. Columns = (fue, ara)
    b. Rows = int

4.2.4. Scenario Fuel Unit Interval (FD_FueUth)

Record	Type	Description
scn	char	{ scenarios }
fue	char	{ fuel types }
uth	char	{ thermal units }
int	int	{ intervals }
Fuel	float	(FuelQuantity) fuel use
Cost	float	($Unit) cost of fuel
CostOfScaling	float	($Unit) cost added by scaling
CostOfAdder	float	($Unit) cost added by CostAdder

Notes:

1. Primary Key: (scn, fue, uth, int).
2. Current logic allows only one fuel type to be mapped to each injector (i.e., fuel-mix modeling not supported, except through the definition of fuels).
3. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.

4. Tabular reports are identified by scn and fue, and have the following structure:
    a. Columns = uth
    b. Rows = int
5. Compound tabular reports are identified by scn and have the following structure:
    a. Columns = (fue, uth)
    b. Rows = int

4.2.5. Scenario Unit Interval (HD_Uth)

Record	Type	Description
scn	char	{ scenarios }
uth	char	{ thermal units } heat units and heat-based injectors
int	int	{ intervals }
Rate	float	(MMBtu / hour) heat rate (base heat + incremental heat)
Start	float	(MMBtu) startup heat
Received	float	(MMBtu / hour) received heat from upstream units
Exhaust	float	(MMBtu / hour) exhaust heat
Cost	float	($Unit) cost of heat

Notes:

1. Primary Key: (scn, uth, int).
2. For thermal units that are injectors, Exhaust is reported only when used by downstream unit (i.e., in combined-cycle model).

3. For heat units, Received or Exhaust is reported, depending on LoadFlag status.
4. Tabular reports are identified by scn and have the following structure:
    a. Columns = inj
    b. Rows = int

4.2.6. Scenario (MC_Hrzn)

Record	Type	Description
scn	char	{ scenarios }
hrzn	char	{ horizons }
FirstInterval	int	{ intervals } First interval of horizon
Cost	float	($Unit) Real cost included in model results
Noncost	float	($Unit) Virtual cost included in model results
Penalty	float	($Unit) Penalty cost included in model results
DeltaCost	float	($Unit) Real cost from periods in DeltaTime of horizon
DeltaNoncost	float	($Unit) Virtual cost from periods in DeltaTime of horizon
DeltaPenalty	float	($Unit) Penalty cost from periods in DeltaTime of horizon
AllCost	float	($Unit) Real cost from all periods of horizon
AllNoncost	float	($Unit) Virtual cost from all periods of horizon
AllPenalty	float	($Unit) Penalty cost from all periods of horizon

Notes:

1. Primary Key: (scn, hrzn)

2. Results are reported for each horizon, including those solved before “StartDate”.
3. Periods after a horizon’s “DeltaTime” of a horizon are re-solved in subsequent horizons. DeltaCost, DeltaNoncost and DeltaPenalty identify costs accruing in “DeltaTime”. AllCost, AllNoncost and AllPenalty identify costs from all periods of horizon, including those after “DeltaTime” that are re-solved by subsequent horizons.
4. DeltaCost, DeltaNoncost and DeltaPenalty is same as Cost, Noncost and Penalty identify costs except when “StartDate” and “StopDate” fall within the horizon’s “DeltaTime”.
5. Tabular reports not defined.

4.2.7. Library Scenario Horizon (MC_Lib)

Record	Type	Description
scn	char	{ scenarios }
lib	char	{ libraries }
Cost	float	($Unit) Real cost included in model results
Noncost	float	($Unit) Virtual cost included in model results
Penalty	float	($Unit) Penalty cost included in model results

Notes:

1. Primary Key: (scn, lib)
2. Identifies by library total costs and penalties between the model “StartDate” and “StopDate” (MDL_ID), excluding costs associated results used to establish appropriate boundary conditions (i.e., those associated with “LeadTime” or after “StopDate”).

3. Tabular reports not defined.

4.2.8. Library Scenario Horizon (MC_LibHrzn)

Record	Type	Description
scn	char	{ scenarios }
lib	char	{ libraries }
hrzn	char	{ horizons }
FirstInterval	int	{ intervals } First interval of horizon
Cost	float	($Unit) Real cost included in model results
Noncost	float	($Unit) Virtual cost included in model results
Penalty	float	($Unit) Penalty cost included in model results
DeltaCost	float	($Unit) Real cost from periods in DeltaTime of horizon
DeltaNoncost	float	($Unit) Virtual cost from periods in DeltaTime of horizon
DeltaPenalty	float	($Unit) Penalty cost from periods in DeltaTime of horizon
AllCost	float	($Unit) Real cost from all periods of horizon
AllNoncost	float	($Unit) Virtual cost from all periods of horizon
AllPenalty	float	($Unit) Penalty cost from all periods of horizon

Notes:

1. Primary Key: (scn, lib, hrzn)
2. Tabular reports have the following structure:
    a. Columns = lib
    b. Rows = hrzn, FirstInterval

4.2.9. Solution (MC_Solution)

Record	Type	Description
slv	char	{ solves } Chronological order in which solution was solved
cyc	char	{ cycles }
hrzn	char	{ horizons }
iter	char	{ iterations }
FirstInterval	int	{ intervals } First interval of horizon
LastInterval	int	{ intervals } Last interval of horizon
#Constraints	int	Number of individual constraints
#Var	int	Number of individual variables
#IntVar	int	Number of individual integer variables
#NonZeros	int	Number of non-zeros
GenTime	float	(second) CPU time to generate math problem
SolveTime	float	(second) CPU time to solve math problem
ElapsedTime	float	(second) Clock-time elapsed

#Iterations	int	Number of solver iterations
#Nodes	int	Number of nodes evaluated in MIP tree
Memory	float	(Mb) Current memory use
Status	char	Solution status
Objective	float	($Unit) Value of objective function
AllCost	float	($Unit) Real cost from all periods of horizon
AllNoncost	float	($Unit) Virtual costs from all periods of horizon
AllPenalty	float	($Unit) Penalty cost from all periods of horizon

Notes:

1. Primary Key: (slv)
2. Reports solver performance and other characteristics of each solver iteration. Always reported using vector format, even when tabular reports are generation for other results.
3. The index slv identifies the order in which each iteration of each horizon was solved. This can be used to identify the value of GUI option “LastSolution” when a model run should be halted after solving the specified solution.
4. Tabular reports not defined.

4.2.10. Interval (MS_Int)

Record	Type	Description
int	int	{ intervals }
StartTime	date	{ time points } Time at start of interval
StopTime	date	{ time points } Time at end of interval

Notes:

1. Primary Key: (int)

4.2.11. Scenario Area Interval (PC_Ara)

Record	Type	Description
scn	char	{ scenarios }
ara	char	{ areas }
int	int	{ intervals }
BalancePrice	float	($/MWh) energy component of LMP adjusted by reference location
LoadFlowPrice	float	($/MWh) distribution-weighted LMP congestion component of fixed-load
LoadLossPrice	float	($/MWh) distribution-weighted LMP loss component of fixed-load
LoadPrice	float	($/MWh) distribution-weighted LMP of fixed-load

SourceFlowPrice	float	($/MWh) distribution-weighted LMP congestion component of injector MW
SourceLossPrice	float	($/MWh) distribution-weighted LMP loss component of injector MW
SourcePrice	float	($/MWh) distribution-weighted LMP of injector MW
Loss	exp	(MW) resistive losses approximated by linearized solution
LossUpdate	exp	(MW) resistive losses from quadratic (I2R) calculation
LossError	exp	(MW) difference between loss and update

Notes:

1. Primary Key: (scn, ara , int).
2. Components of LMP are based on reference node ("ReferenceNodeName") or load-distributed reference (ARA_REF) which may be different from the slack bus. Components of locational marginal price (LMP) can be identified as (1) energy, (2) losses and (3) congestion. BalancePrice identifies energy component, LoadFlowPrice and SourceFlowPrice identify congestion components, LoadLossPrice and SourceLossPrice identify loss components, and LoadPrice and SourcePrice identify resulting LMPs.
3. Loss factors and losses are calculated using shift-factors of the base topology and do not include impact of open branches modeled by FCPS.
4. Loss, LossUpdate and LossError are based on all branches with resistance, even when penalty factor model is used and losses are already included in area loads (i.e., regardless of option "AddAreaLosses"). Values are reported only for power-balance areas.

5. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
6. Tabular reports are identified by scn and have the following structure:
    a. Columns = ara
    b. Rows = int

4.2.12. Scenario Branch Interval (PC_Brn)

Record	Type	Description
scn	char	{ scenarios }
brn	char	{ branches }
int	int	{ intervals }
Mw	float	(MW) flow
Loss	exp	(MW) resistive losses approximated by linearized solution
LossUpdate	exp	(MW) resistive losses from quadratic (I2R) calculation
LossError	exp	(MW) difference between loss and update

Notes:

1. Primary Key: (scn, brn, int).
2. Reports losses on branches with non-zero resistance. Losses on AC branches are added to loads in power-balance equations when option “AddAreaLosses” selected (in PN library). When using penalty-factor model, losses are not added as they should already be included in area load requirements. Losses of DC branches are always added to power-balance

equations (DC flows are controllable and impact of losses on DC branches cannot be captured by penalty-factor model).

3. Reported losses are based on slack bus ("SlackBusName") used to solve the model, and are not adjusted if a different Reference is used (i.e., "ReferenceNodeName" or ARA_REF data). However, reported shift factors, loss factors, area prices and congestion components of Locational Marginal Prices (LMPs) are adjusted based on the Reference.
4. Loss, LossUpdate and LossError are reported when using either "AddAreaLosses" or penalty-factor model.
5. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
6. Tabular reports are identified by scn and have the following structure:
    a. Columns = brn
    b. Rows = int

4.2.13. Scenario Injector Interval (PC_Inj)

Record	Type	Description
scn	char	{ scenarios }
inj	char	{ injectors } Excludes "NetFlag" injectors
int	int	{ intervals }
LMP	float	($/MWh) locational marginal price
BalancePrice	float	($/MWh) energy component of LMP
FlowPrice	float	($/MWh) congestion component of LMP

LossPrice	float	($/MWh) loss component of LMP
LossFactor	exp	loss factor used to solve power flow
LfUpdate	exp	loss factor updated using power flow solution
LfError	exp	difference between loss factor and update

Notes:

1. Primary Key: (scn, inj, int). Excludes paired injectors and point-to-point injectors.
2. Reported values are based on reference node or load-distributed reference which may be different from the slack bus. Loss factors are calculated using the base topology (branches opened by SCN_BRN_OPN) and do not include impact of open branches modeled by FCPS (SCN_FCP_OPN or SCN_FCP_OPT).
3. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
4. Tabular reports are identified by scn and have the following structure:
    a. Columns = inj
    b. Rows = int

4.2.14. Scenario Node Interval (PC_Nd)

Record	Type	Description
scn	char	{ scenarios }
nd	char	{ nodes } electrical nodes and aggregate nodes
int	int	{ intervals }

LMP	float	($/MWh) locational marginal price
BalancePrice	float	($/MWh) energy component of LMP
FlowPrice	float	($/MWh) congestion component of LMP
LossPrice	float	($/MWh) loss component of LMP
LossFactor	exp	loss factor used to solve power flow
LfUpdate	exp	loss factor updated using power flow solution
LfError	exp	difference between loss factor and update

Notes:

1. Primary Key: (scn, nd, int).
2. Reported nodes must be identified by "ReportNode" (NDE_ID or NDA_ATT), except for node labeled "Reference" that reports LMPs of reference node or load-distributed reference.
3. Reported values are based on reference node or load-distributed reference which may be different from the slack bus. Loss factors are calculated using the base topology (branches opened by SCN_BRN_OPN) and do not include impact of open branches modeled by FCPS (SCN_FCP_OPN or SCN_FCP_OPT).
4. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
5. Tabular reports are identified by scn and have the following structure:
    a. Columns = nd
    b. Rows = int

4.2.15. Topology Map (PC_Top)

Record	Type	Description
scn	char	{ scenarios }
int	int	{ intervals }
Topology	int	{ topologies }

Notes:

1. Primary Key: (scn, int).
2. Tabular report is labeled PC_Top and has the following structure:
    a. Columns = scn
    b. Rows = int

4.2.16. Scenario Path Interval (PN_Pth)

Record	Type	Description
scn	char	{ scenarios }
pth	char	{ paths } monitored paths
int	int	{ intervals }
Mw	float	(MW) flow
Min	float	(MW) minimum flow limit
Max	float	(MW) maximum flow limit
Violation	float	(MW) violation of flow limit
MinEnforced	bit	Minimum limit enforced in solution
MaxEnforced	bit	Maximum limit enforced in solution
Binding	bit	Binding constraint
SAC	bit	Security analysis constraint
Penalty	float	($Unit) cost of violation
SP	float	($/MWh) shadow price of MW limit
Revenue	float	($Unit) value (product of Mw and SP)

Notes:

1. Primary Key: (scn, pth, int).
2. Reports all enforced or monitored paths. Additional paths can be reported using “cid” tables by providing the path with a “ConstraintID” (PTH_ID).

3. MinEnforced and MaxEnforced identifies constraints enforced in solution. A path can have a non-zero Violation when enforced or identified by security analysis (SAC).
4. Binding identifies enforced paths with non-zero shadow price or paths with flow equal to their limit. A path can be binding only when enforced. Binding constraints include enforced paths with non-zero Violation.
5. SAC identifies violated, binding or near-binding constraints identified by security analysis, which are enforced only if identified prior to the last iteration of the solution horizon. When multiple iterations are solved, SAC identifies constraints identified by any iteration.
6. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
7. Tabular reports are identified by scn and have the following structure:
    a. Columns = pth
    b. Rows = int
8. Sparse fields: Violation, Penalty.

4.2.17. Scenario Injector Interval (UC_Inj)

Record	Type	Description
scn	char	{ scenarios }
inj	char	{ injectors } committed injectors
int	int	{ intervals }
Commit	bit	commitment status (1=on, 0=off)
SU	bit	first commitment interval
SD	bit	first de-commitment interval
Hot	bit	Startup from hot status
Warm	bit	Startup from warm status
Cold	bit	Startup from cold status
Failed	bit	Failed startup (identified after solving commitment)
StatusViolation	exp	violation of on status (+) or off status (-)
TimeViolation	exp	(hour) violation of MinOn (+) or MinOff (-)
BaseCost	float	($Unit) Cost associated with “BaseCost” (not including cost curve)
StartCost	float	($Unit) Cost associated with “StartCost” (not including cost curve)
Penalty	float	($Unit) cost of commitment-constraint violations

Notes:

1. Primary Key: (scn, inj, int).
2. Aggregate reports are identified for first interval of aggregate periods that are fully defined by the solution horizon.
3. Tabular reports are identified by scn and have the following structure:
    a. Columns = inj
    b. Rows = int
4. Sparse fields: StatusViolation, TimeViolation, Penalty.

4.2.18. Scenario Injector Status (UC_Status)

Record	Type	Description
scn	char	{ scenarios }
inj	char	{ injectors } committed injectors
int	int	{ intervals }
Change	int	change in commitment status (actual or attempted)
StatusViolation	exp	violation of on status (+) or off status (-)
TimeViolation	exp	(hour) violation of MinOn (+) or MinOff (-)

Notes:

1. Primary Key: (scn, inj, int).
2. Intervals are reported only when a change in commitment status exists. Failed startups are also identified as an attempted change in status.

3. The value of Change is interpreted as follows:

   Startup from hot = 3

   Startup from warm = 2

   Startup from cold = 1

   Shutdown = -1

4. Tabular reports are labeled UC_Change(<scn>), UC_StatusViolation (<scn>) and UC_TimeViolation (<scn>), and have the following structure:

   a. Columns = inj

   b. Rows = int

5. Sparse fields: Change, StatusViolation, TimeViolation.

4.2.19. Scenario Injector Status by Time Point (UC_TmpStatus)

Record	Type	Description
scn	char	{ scenarios }
inj	char	{ injectors } committed injectors
tmp	date	{ time points }
Change	int	change in commitment status (actual or attempted)

Notes:

1. Primary Key: (scn, inj, tmp).

2. Time-point results are identified so that Change can used to define binary schedules (SCH_BIN) used as input data in other models.

3. Tabular reports are labeled UC_TmpChange(<scn>) and have the following structure:
    a. Columns = inj
    b. Rows = tmp
4. Sparse fields: Change.

5. Tools

This section will cover the tools that we used to develop and perform the experiments in detail. These are the tools that matched exactly or to most levels as to what we were expecting out of them.

5.1. Power System Optimizer (PSO)

PSO is a production power market cost simulator developed by Polaris Optimization Systems. An industry wide used tool that can precisely simulate the energy market. It is developed using the MILP approach of solving the SCUCED problem instead of heuristics and is consistent with the methods used by many ISOs. It supports vastly dynamic modelling of inputs, supports day ahead as well as intra-hour estimation capabilities, uncertainty forecasting using historical data and is extremely adaptable to changing system conditions.

We used PSO in our experiments to generate the data to train our models. We did so by using the option to feed in historical data and used the forecasting feature to generate new system states. Our models are built using these outputs, which are extremely precise to actual system conditions for the given conditions would be.

5.2. PyTorch

PyTorch is a robust, scalable and production ready deep learning framework for Python. It contains a plethora of utils for deep learning with cuda and distributed training support.

It makes developing ML models extremely easy by using its utilities. Neural Networks is easy with torch.nn, as it has a wide range of pre-developed as well as pre-trained models which you can extend to create new models tailored to your needs. It also comes with easy ways to visualize your experiments and almost all of the graphs in this have been developed using this.

5.3. PyTorch-Geometric (PyG)

PyG is an extension to PyTorch and is tailored specifically for developing and Graph Neural Networks. Much like PyTorch, this has pre-developed algorithms for implementing various GNNs. These can be easily extended to create custom GNN models for specific use cases.

It can be used together with PyTorch to benefit from features from PyTorch. We used PyG for implementing our GNN layers.

5.4. Google Collab

Google Collab is a Google service that provisions users with a Python Notebook runtime, which can be tailored to install any libraries that the project needs. It provides with a safe environment to install libraries and run scripts for quick prototyping. It runs on clouds with GPU support, which means faster testing rounds and easier updates.

6. Graph Neural Network (GNN)

6.1. Understanding GNN

Understanding GNN Graphs have been known to be one of the most popular manner to represent data that is structured and highly related. Graph Neural Networks provide ways of modelling associations between nodes in a graph and hence has been a hot topic between researchers. It has changed how data analysis can be performed on graphs with fast and accurate prediction ability. GNNs are type of Neural Networks that can be applied directly on graphs. Applications range from image classification, gene identification, brain connectomes and more.

Each node in a graph represents a set of features which are identified by the node. Each node is associated with a set of labels. GNN is then used to train weights which can be used to predict labels for new nodes. The graphs in the question can be directed or un-directed graphs or graphs with edge weights, GNNs are flexible to accommodate these edge features to accommodate any constraints that edges may have.

GNNs work by representing the states of the nodes in a recurrent manner and applying a Feed forward NN. This NN learns the biases and embeddings, GNNs do MessagePassing. This is basically Neighborhood Aggregation in which the nodes push their embeddings to their neighbor nodes through the edges. Different NN can also be used to model which

can be used to capture the spatial relationships between nodes. These embeddings are then summed together to get new graph representation.

Much like how traditional NNs introduced Convolutional Neural Networks, which help to speed up the learning and increase accuracy with hierarchical processing of data units, Graph Convolutional Networks (GCN) exist to do the same but on graph data. GCNs are extremely powerful NNs and have proven ability to perform extremely well with minimum training.

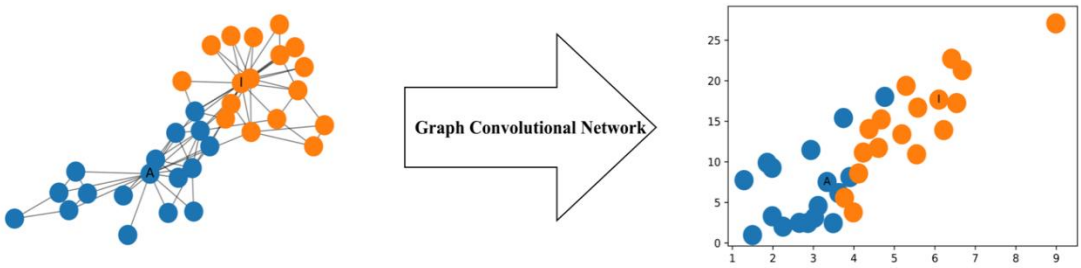

Figure 3: Representation of learning with GCN

Figure 3 demonstrates how a simple GCN can preserve the relationships between the nodes with minimal training. This [2] paper comprises of a recent comprehensive review of methods, applications and efficiency of GCN across different domains.

6.2. Constructing GNN Based on ERCOT 7-Bus Model

Chapter 3 covers in detail about how the 7-Bus model is and the constraints of each component of the system. This can be visualized as a classic graph model with different components of the system representing the nodes of the graph and the connecting transmission lines as edges connecting the nodes.

Each node of this system graph encapsulates its constraints and properties and can be represented as feature matrix for the node in the GNN. Each edge, since it is a transmission line, it comes with its set of constraints which represent the edge feature matrix which is used to model relationships between the nodes.

We represented the injectors, buses, and areas as nodes. The injector node contains all the properties and constraints mentioned in Nodes/Buses. The bus (substations) which are directly connected with the Injectors are represented as nodes and provide a layer of abstraction between the network and the injector. These nodes are responsible for storing/transferring the energy from one point to another and do so by using transmission lines. Bus nodes relate to area nodes which comprise of the consuming areas of the system and to another substation nodes with transmission lines. The transmission lines are representing the edges connecting the nodes. Since they carry energy, they are subject to certain constraints (Branches) and limits which are encapsulated within the edges as attributes. The area nodes are the ones that create the demand, this demand is associated with each area. These nodes contain the demand for a given time.

For a given point time, the complete state of the network can thus be defined as a graph with all the required relations preserved. Different states of systems for each timestamp can be represented by multiple graphs in similar fashion.

6.3. Creating own Dataset in PyTorch

We have all the information we have but the next problem is how can we easily create graphs and generate a graph dataset that can be loaded to train the model. That is where

NetworkX and PyTorch come to rescue. NetworkX is a utility that is developed to easily create descriptive knowledge graphs with support for edge attributes as well. We used NetworkX to generate aforementioned graphs, with a simple adapter from NetworkX we created graphs which could then be added to a dataset. PyTorch amongst everything also provides a scalable method to implement custom Datasets, these datasets can be dynamically created and updated at scale. The main advantage of this is these Datasets don't need to be loaded into memory completely to train the model, but rather can be loaded into memory in batches which is how we require these when training a neural network. These datasets can also be used in a distributed environment when training is distributed to multiple compute systems. PyTorch also provides easy framework to create the datasets and store them. The datasets can be downloaded from any cloud blob storages and processed ones can be stored back into blob storages as well. We used a combination of these two libraries to create dataset of graphs. We however did not experiment with cloud-based blob storages, or with very large datasets, but people have praised these two to be the best ones currently.

7. Implementing GNN models on 7 Bus system

This chapter will cover how the 2 GNN models were implemented on the 7 bus models to do deep learning and how the results were generated to verify the viability of the models for this specific problem.

- Working of GNN:

  GNNs are a combination of an information diffusion mechanism and neural networks, representing a set of transition functions and a set of output functions.

  The information diffusion mechanism is defined by nodes updating their states and exchanging information by passing “messages” to their neighboring nodes until they reach a stable equilibrium.

  The process involves first a transition function that takes as input the features of each node, the edge features of each node, the neighboring nodes’ state, and the neighboring nodes’ features and outputs the nodes’ new state.

  Based on this understanding of GNN we have implemented 2 models which will be discussed in following chapters.

  7.1. GCNConv: Graph Convolutional Network Model

  7.2. NNConv: Message Passing Neural Networks

7.1. GCNConv: Graph Convolutional Network Model

This model is referred from the [3]

7.1.1. Concept

Many approaches for semi-supervised learning using graph representations have been proposed in recent years, most of which fall into two broad categories:

a. Graph Laplacian regularization – Graph Laplacian regularization means to get a better understanding on the role of normalization of the graph Laplacian matrix as well as impact of dimension reduction in graph learning [4]. It includes propagation of labels, manifold regularization, and deep semi-supervised embedding.

b. Graph embedding-based approaches - Graph embeddings can be learned by the skip-gram model. DeepWalk learns embeddings via the prediction of the local neighborhood of nodes, sampled from random walks on the graph. LINE and node2vec extend DeepWalk with more sophisticated random walk or breadth-first search schemes [3].

For all the above methods, a multistep pipeline including random walk generation and semi-supervised training is necessary. Each step of these models must be optimized separately. This can be alleviated by injecting label information in the process of learning embeddings.

7.1.1.1. Semi-Supervised Node Classification

The paper [3] considers a two-layer GCN for semi-supervised node classification on a graph with a symmetric adjacency matrix A (binary or weighted). The forward model then takes the simple form:

$$Z = f(X, A) = \mathrm{softmax}\Big(\hat{A}\ \mathrm{ReLU}\Big(\hat{A}XW^{(0)}\Big)\,W^{(1)}\Big)\ .$$

Figure 4: GCN Model Formula [3]

Where W(0) is an input-to-hidden weight matrix for a hidden layer with H feature maps. W(1) is a hidden-to-output weight matrix. The softmax activation function is applied row-wise. For semi-supervised multiclass classification, cross-entropy error over all labels is evaluated.

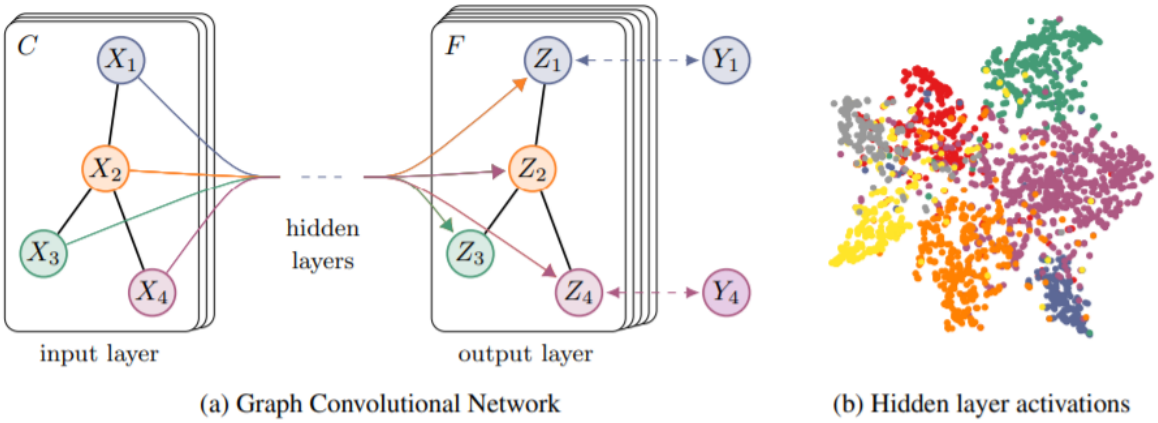

Figure 5: Multilayer GCN for semi-supervised learning with C input channels and F feature maps in the output layer.[3]

7.1.2. Model Setup

We will be using PyTorch for its readily available implementations of GNN layers, I decided on using the GCNConv layer from the Pytorch-Geometric suite. This layer is implementation of the exact model for semi-supervised classification mentioned in [3]. We also used TOPKPooling operator from [5]for pooling layers and performed global mean pooling.

We used 1-month data output from PSO as our ground truth data. Transformed the PSO outputs to graph data, where each timestamp represents a complete graph state. Similarly collecting each timestamp data, created a dataset to load into the model. Train and test dataset was creating using 70/30 split.

We experimented with a 6-layer network with 2 GCNConv layers, 2 Pooling layers, 3 Linear layers and 1 dropout layer to get the final output. RELU activation function was used for each of the GCNConv and Linear layers.

For training we used the Adam approximation loss function to calculate the training loss for each batch and for back propagation.

7.1.3. Hyperparameter Optimization

Parameter optimization was performed using exhaustive search over set limits for each parameter and best results were selected for each group. The GCNConv model is defined as:

GCNConv(in_channels, out_channels, improved=False, cached=False, bias=True, normalize=True, **kwargs)

The GCNConv model has below parameters:

- in_channels (int) – Size of each input sample.
- out_channels (int) – Size of each output sample.
- improved (bool, optional) – If set to True, the layer computes the transpose by adding 2*identity matrix. (default: False)
- cached (bool, optional) – If set to True, the layer will cache the computation on first execution, and will use the cached version for further executions. (default: False)
- bias (bool, optional) – If set to False, the layer will not learn an additive bias. (default: True)
- normalize (bool, optional) – Whether to add self-loops and apply symmetric normalization. (default: True)
- **kwargs (optional) – Additional arguments of torch_geometric.nn.conv.MessagePassing.

From the above parameter's list, we can control the model performance based on the out_channels for hidden layer connectivity, improved, bias and normalize.

Model was tested against multiple of each of these parameters and grouped to get the best one per group.

For bias, normalize and improved the model was tested with hidden connectivity ranging from 7 till 98 with an interval of 7. From outputs, best connectivity was chosen based on min MSE.

The created dataset was used in three different ways to predict different parameters.

a. In the First experiment the model was trained to predict only commit status which can be one of 1 and 0.

b. In the Second experiment model was trained to predict the dispatch for ED.

c. Lastly, the model was fed with the input data and tried to predict UC and ED params together.

Input data consisted of 31 days of network data for the 7-bus model simulated using PSO. The data was split between 30 days to train the model and 1 day to test the model. The error was measured using mean squared error method with ground truth being labels predicted by PSO.

7.1.4. Results/Observations

All the below experiments are made using the below parameters combinations:

Normalize	Improved	Bias
True	True	True
True	True	False
True	False	True
True	False	False
False	True	True
False	True	False
False	False	True
False	False	False

For all the below experiments the hidden layer channel size is varied from 7 to 100 in steps of 7. Epochs are in the range of 1 to 20.

7.1.4.1. Injector Commitment Prediction:

Input Variables: 8 Output Variables: 1

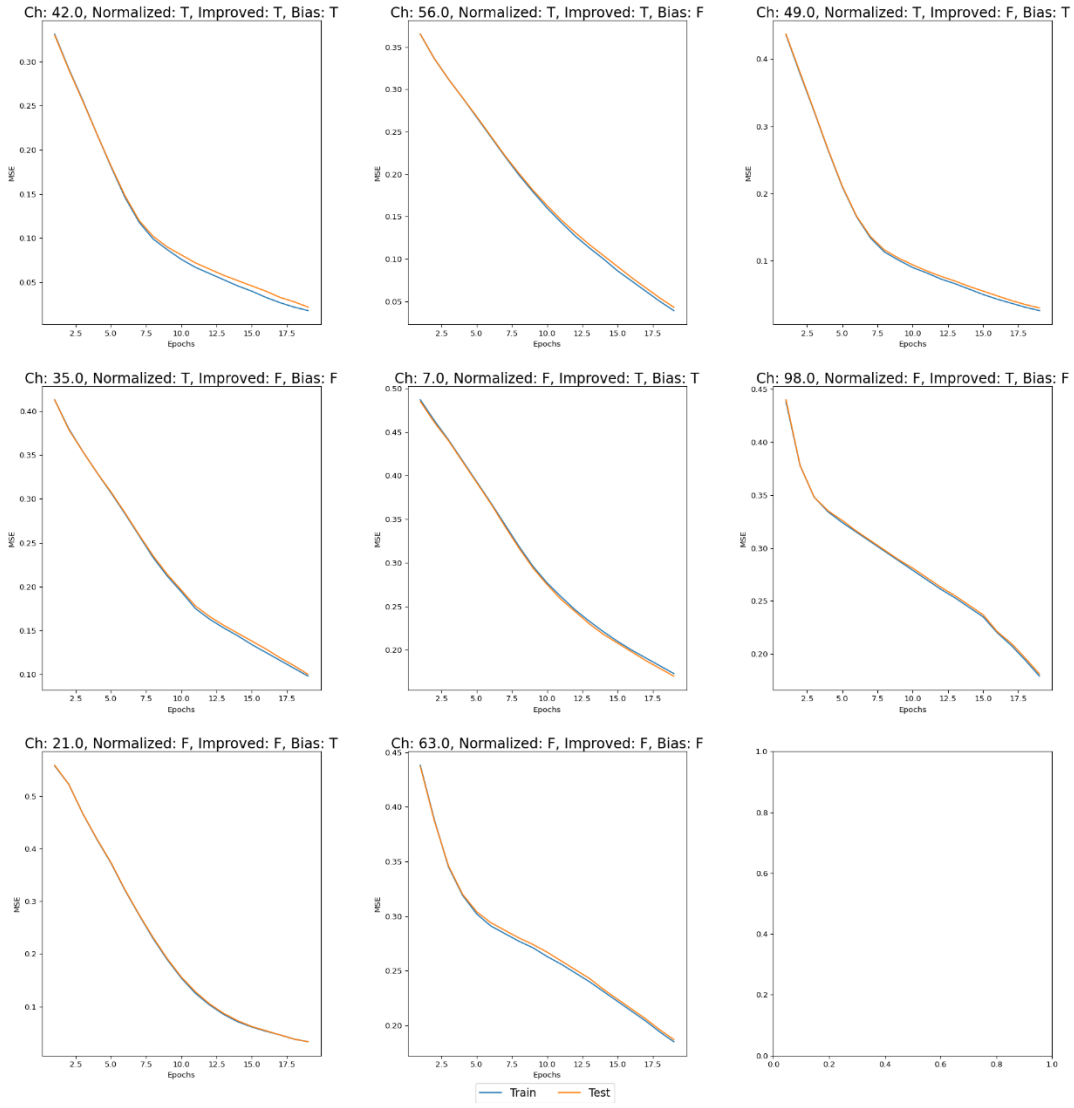

Figure 6: Injector Commitment Prediction Outputs

Observations: The MSE is significantly reduced in case of Normalized – False, Improved – False and Bias – False with hidden channels as 63. The lowest MSE is obtained by keeping Bias as True and others as False but the train and test MSE are almost same for 20epochs, which might be the result of Overfitting.

7.1.4.2. Injector Dispatch Prediction:

Input Variables: 8 Output Variables: 1

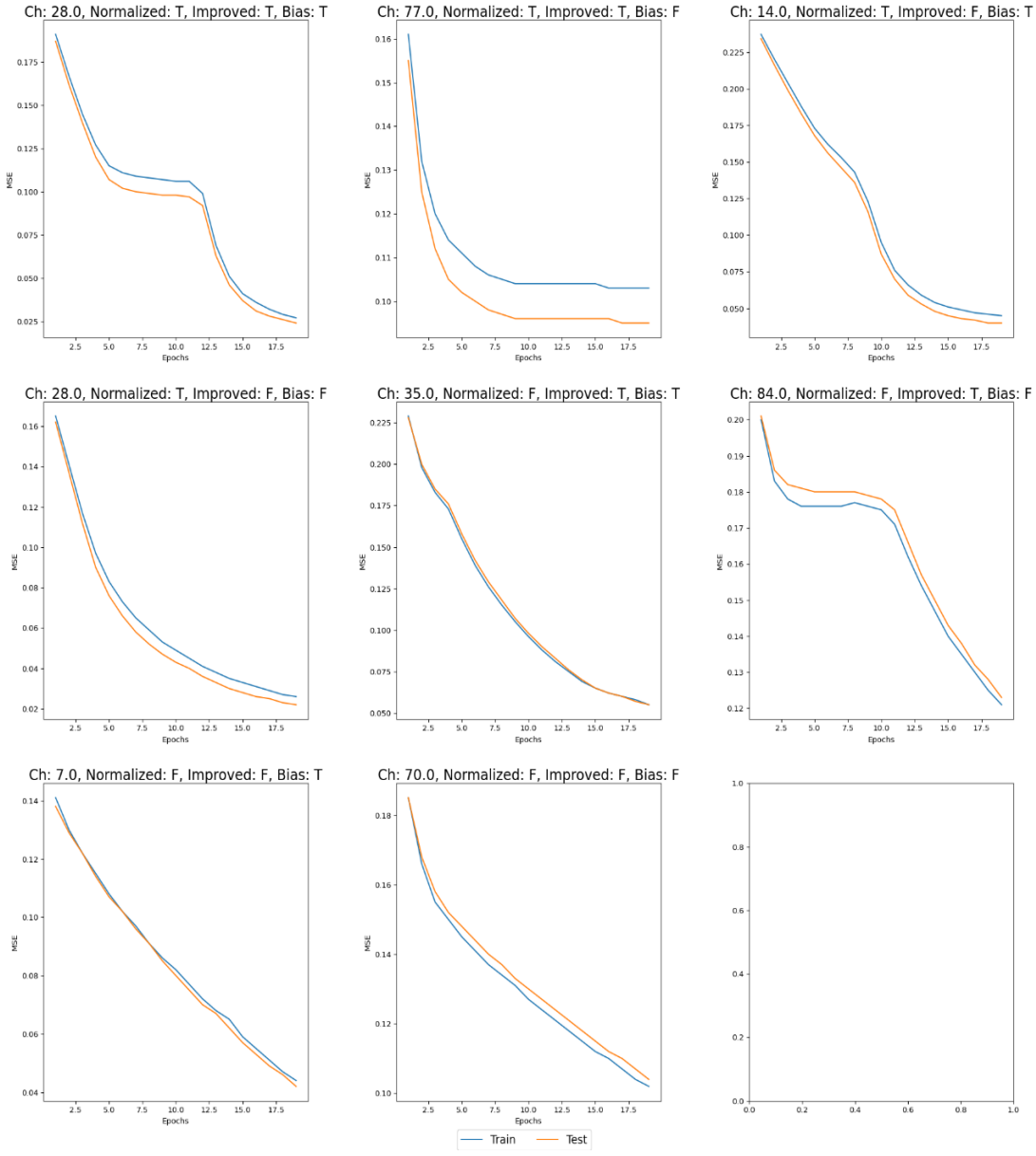

Figure 7: Injector Dispatch Prediction Results

Observations: By keeping Normalized - False, improved – False, Bias – False and hidden channels – 70 the model performance was substantial and MSE was reduced significantly. With the above parameters the model performance was good even for predicting Injector commitment.

For channels – 77, Normalized - True, improved – True, Bias – False, the model's performance was worst.

7.1.4.3 Predicting UCED Together:

Input Variables: 8 Output Variables: 6

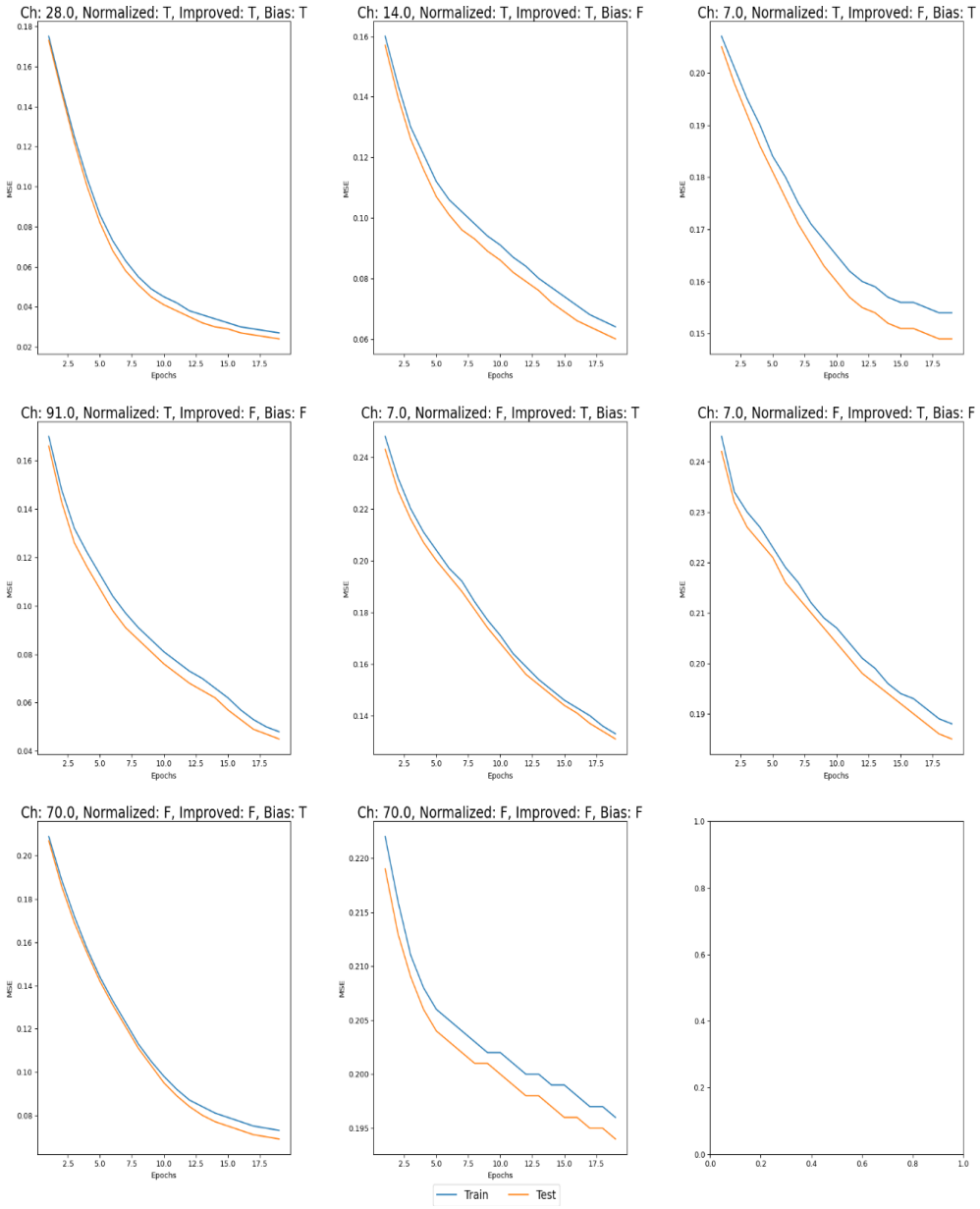

Figure 8: Predicting UCED together

Observations:

The GCNconv model performed the best for predicting UC-ED together. No overfitting is observed for this experiment. The model performed best for when all the parameters were true and hidden channels were 28. However, this model performed worst when all parameters were False as compared to predicting UC and ED individually.

7.2. NNConv: Message Passing Neural Networks

This model is referred from the paper [6]

7.2.1. Concept

Neural network architectures that operate directly on graph-valued inputs have been developed allowing “end-to-end” learning on molecular space. This approach is based on models that simultaneously learn both how to extract appropriate features as well as how to use these features to make accurate predictions. End-to-end learning techniques have supplanted traditional methods in image recognition and computer translation [7]. A number of approaches for end-to-end learning on molecules have recently been unified into a single theoretical framework known as Message Passing Neural Networks (MPNNs) and even more recently as graph networks [7]. We can learn edge features in MPNN by introducing hidden states for all edges in the graph.

In MPNNs, predictions are generated from input graphs with node and edge features. The network comprises a sequence of layers, including a number of message passing layers and a readout layer.

7.2.1.1 Message Passing Layer:

In the message passing layers, node-level state vectors are updated according to the graph’s connectivity and the current states of neighboring nodes. The message passing phase runs for T time steps and is defined in terms of message functions and vertex update functions [6]. During the message passing phase, hidden states at each node in the graph are updated based on messages according to below equation:

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw})$$
$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1})$$

Figure 9: MPNN Equations [6]

7.2.1.2. The Readout Phase

Following a number of message passing layers, the readout layer generates a single graph-level vector from node-level states [7]. The readout phase computes a feature vector for the whole graph using some readout function R [6]. The message functions, vertex update functions, and readout function R are all learned differentiable functions. R operates on the set of node states and must be invariant to permutations of the node states for the MPNN to be invariant to graph isomorphism [6].

7.2.2 Model Setup

As described in chapter 7.1.2, we will be using PyTorch for its readily available implementations of GNN layers, I decided on using the NNConv layer from the Pytorch-Geometric suite. This layer is implementation of the continuous kernel-based convolutional operator from [6]. We also used a Sequential Neural Network which maps the edge features to shape

[-1, in_channels * out_channels] and performed global mean pooling.

We used 1-month data output from PSO as our ground truth data. Transformed the PSO outputs to graph data, where each timestamp represents a complete graph state.

Similarly collecting each timestamp data, created a dataset to load into the model. Train and test dataset were creating using 70/30 split. I experimented with a 6-layer network with 2 NNConv layers, 2 Pooling layers, 3 Linear layers and 1 dropout layer to get the final output. RELU activation function was used for each of the NNConv and Linear layers. For training we used the Adam approximation loss function to calculate the training loss for each batch and for back propagation.

7.2.3. Hyperparameter Optimization

Parameter optimization was performed using exhaustive search over set limits for each parameter and best results were selected for each group.

The NNConv is defined as:

NNConv(in_channels, out_channels, nn, aggr='add', root_weight=True, bias=True, **kwargs)

The parameters for the NNConv are as follows:

- in_channels (int) – Size of each input sample.
- out_channels (int) – Size of each output sample.
- nn (torch.nn.Module) – A neural network that maps edge features edge_attr of shape [-1, num_edge_features] to shape [-1, in_channels * out_channels]
- aggr (string, optional) – The aggregation scheme to use ("add", "mean", "max"). (default: "add")
- root_weight (bool, optional) – If set to False, the layer will not add the transformed root node features to the output. (default: True)

- bias (bool, optional) – If set to False, the layer will not learn an additive bias. (default: True)
- **kwargs (optional) – Additional arguments of torch_geometric.nn.conv.MessagePassing.

From the above parameter's list, we can control the model performance based on the out_channels for hidden layer connectivity, aggr, bias and root_weight. Model was tested against multiple of each of these parameters and grouped to get the best one per group.

For aggr, bias and root_weight the model was tested with hidden connectivity ranging from 7 till 98 with an interval of 7. From outputs, best connectivity was chosen based on min MSE.

The created dataset was used in three different ways to predict different parameters.

a. In the First experiment the model was trained to predict only commit status which can be one of 1 and 0.

b. In the Second experiment model was trained to predict the dispatch for ED.

c. Lastly, the model was fed with the input data and tried to predict UC and ED params together.

Input data consisted of 31 days of network data for the 7-bus model simulated using PSO. The data was split between 30 days to train the model and 1 day to test the model. The error was measured using mean squared error method with ground truth being labels predicted by PSO.

7.2.4. Results

All the below experiments are made using the below parameters combinations:

Root Weight	Bias
True	True
True	False
False	True
False	False

For all the below experiments the hidden layer channel size is varied from 7 to 100 in steps of 7. Epochs are in the range of 1 to 20.

Also, aggregation feature of the following values was used to propagate the calculated features to the hidden layers.

Aggregation
Add
Mean
Max

7.2.4.1. Injector Commitment Prediction:

Input Variables: 8 Output Variables: 1

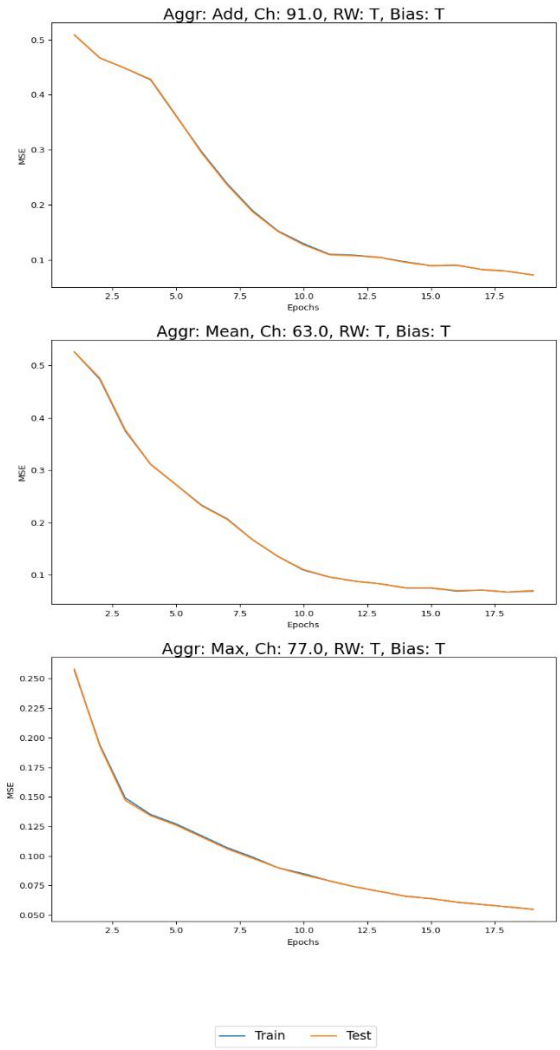

Figure 10: Predicting injector commitment with root weight and bias

Observations: For Root Weight - True and Bias - True, the lowest MSE is obtained by keeping Aggregation as Max and hidden channels as 77.

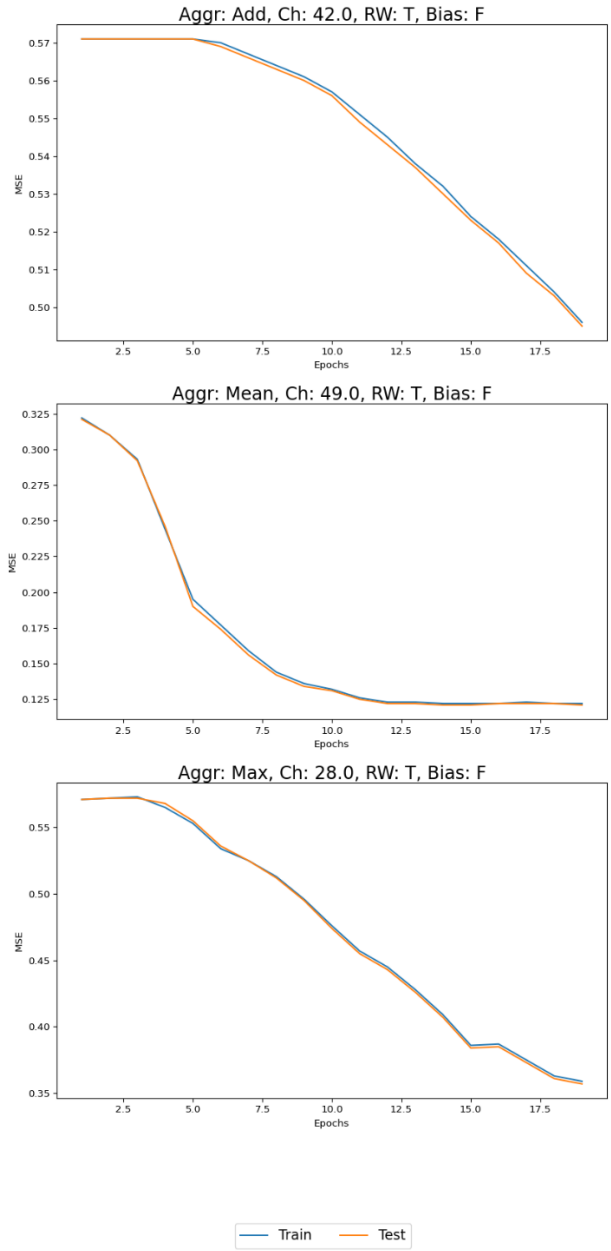

Figure 11: Predicting injector commitment with root weight and no bias

Observations: For Root Weight- True and Bias – False, the lowest MSE is obtained by keeping Aggregation as Mean and hidden channels as 49

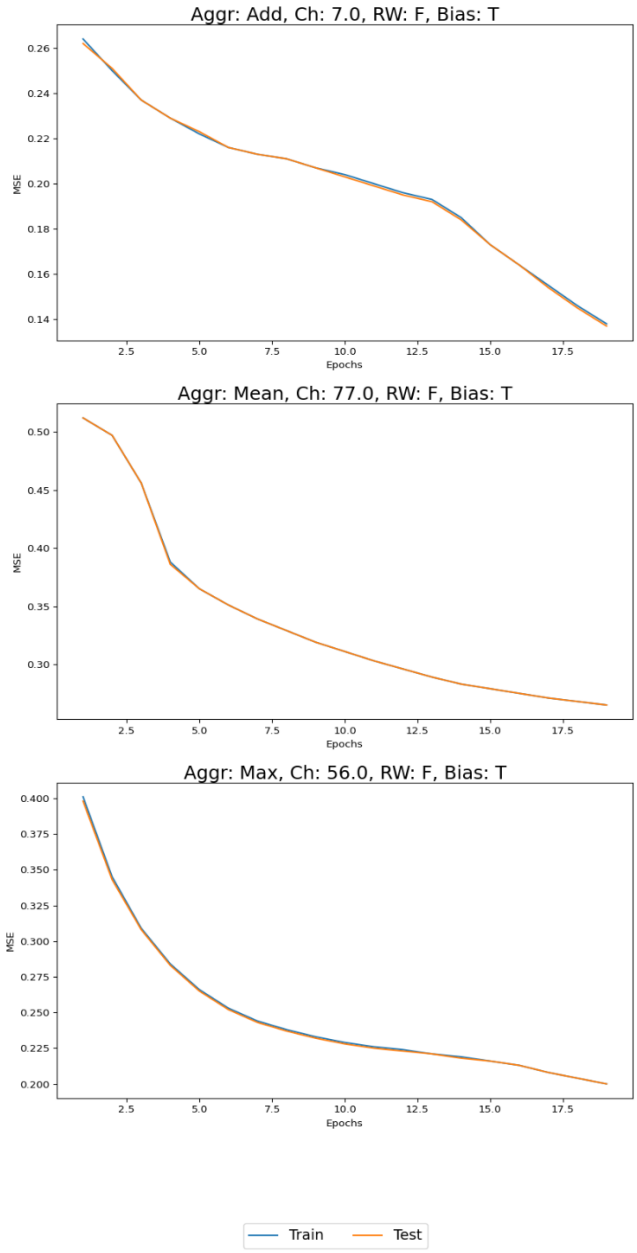

Figure 12: Predicting injector commitment without root weight but with bias

Observations: For Root Weight- False and Bias – True, the lowest MSE is obtained by keeping Aggregation as Add and hidden channels as 7.

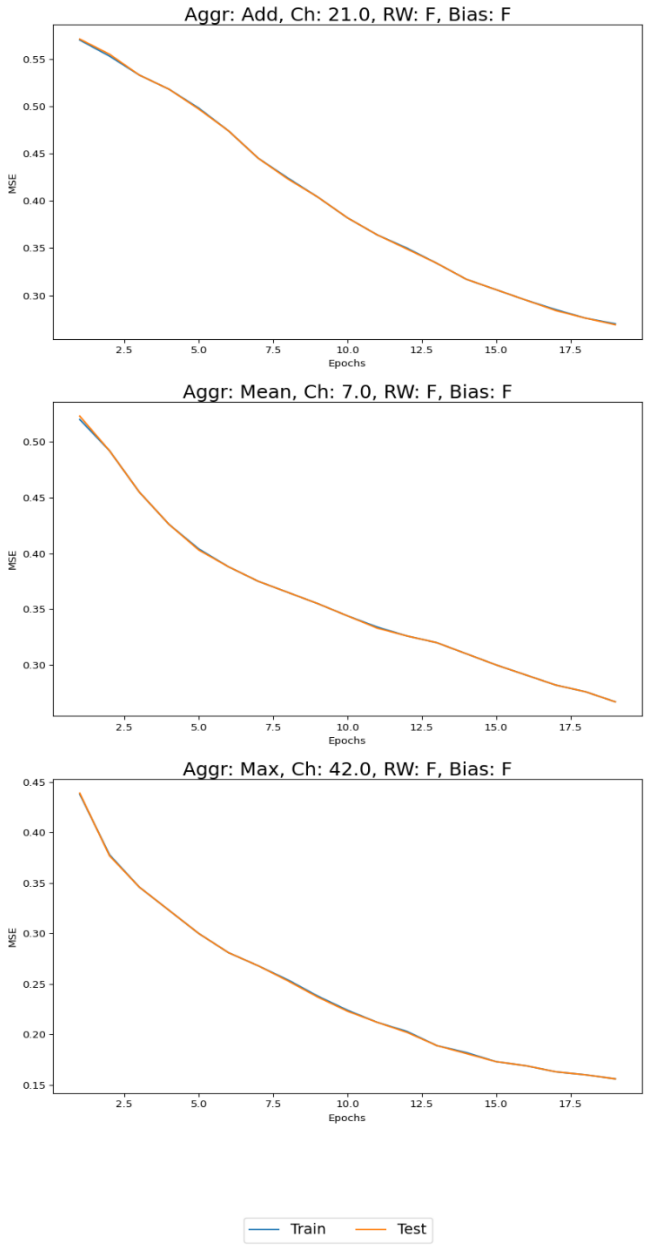

Figure 13: Predicting injector commitment without root weight and no bias

Observations: For Root Weight- False and Bias – False, the lowest MSE is obtained by keeping Aggregation as Max and hidden channels as 42

Overall injector commitment observations: We observe the least MSE with aggregation as Max, hidden channels as 77 and including root weight and bias. Removing the bias while adding the root weight seems to decrease the overfitting. Adding more epochs do help in increasing the accuracy but at the same time, it overfits the data as both the train and test accuracy seems to be similar.

7.2.4.2. Injector Dispatch Prediction:

Input Variables: 8 Output Variables: 1

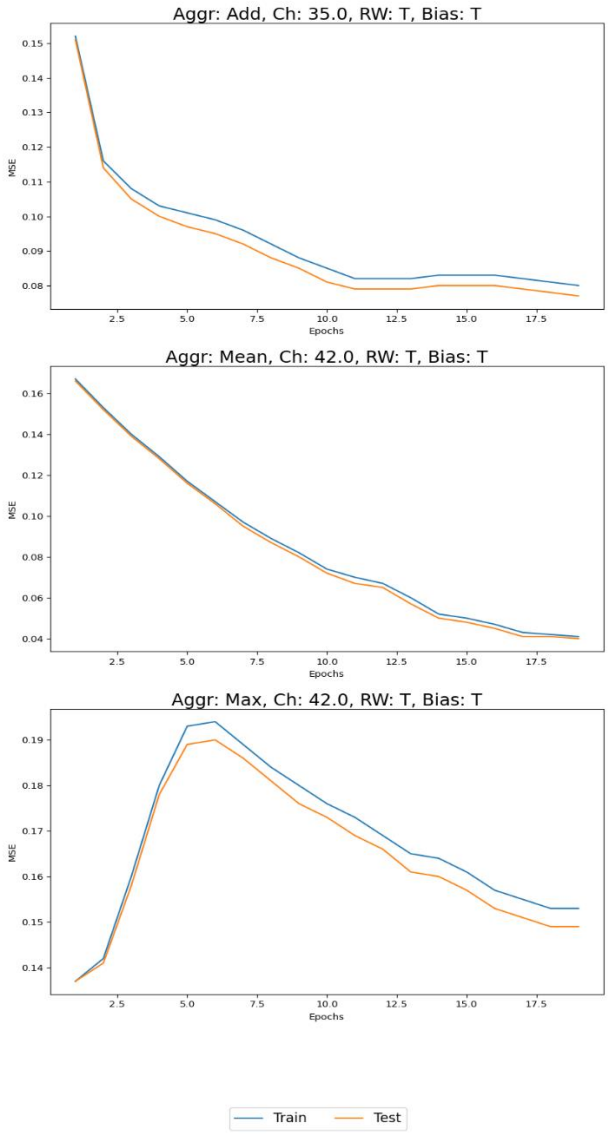

Figure 14: Predicting injector dispatch with root weight and bias

Observations: For Root Weight- True and Bias – True, the lowest MSE is obtained by keeping Aggregation as Mean and hidden channels as 42

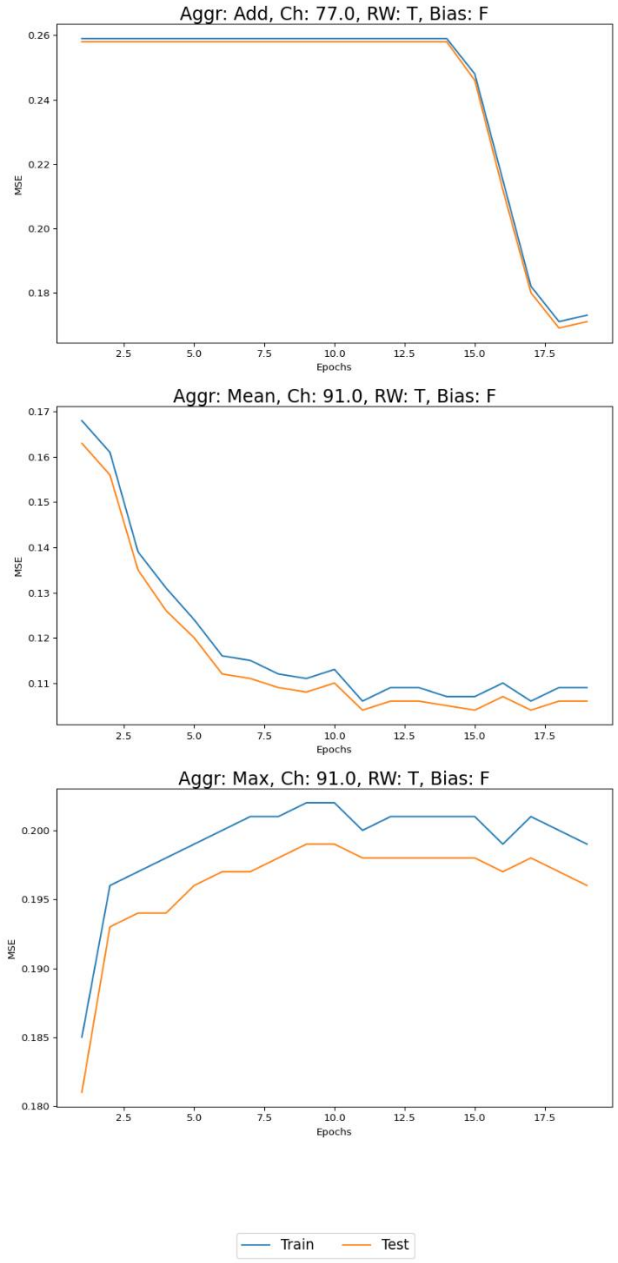

Figure 15: Predicting injector dispatch with root weight and no bias

Observations: For Root Weight- True and Bias – False, the lowest MSE is obtained by keeping Aggregation as Mean and hidden channels as 91.

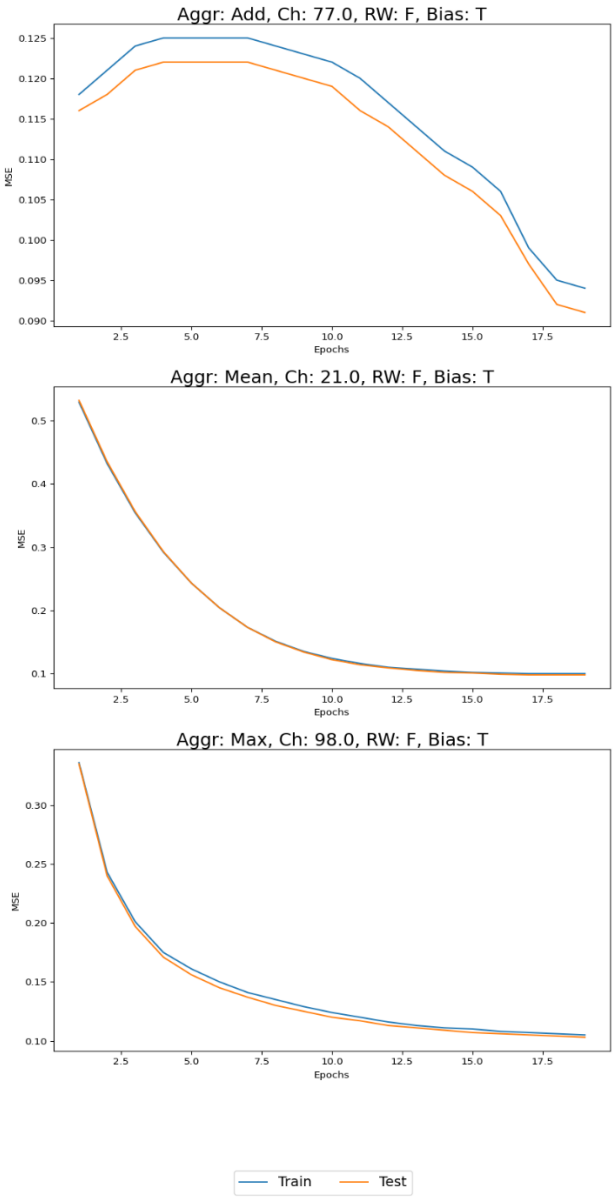

Figure 16: Predicting injector dispatch without root weight but with bias

Observations: For Root Weight- False and Bias – True, the lowest MSE is obtained by keeping Aggregation as Add and hidden channels as 77.

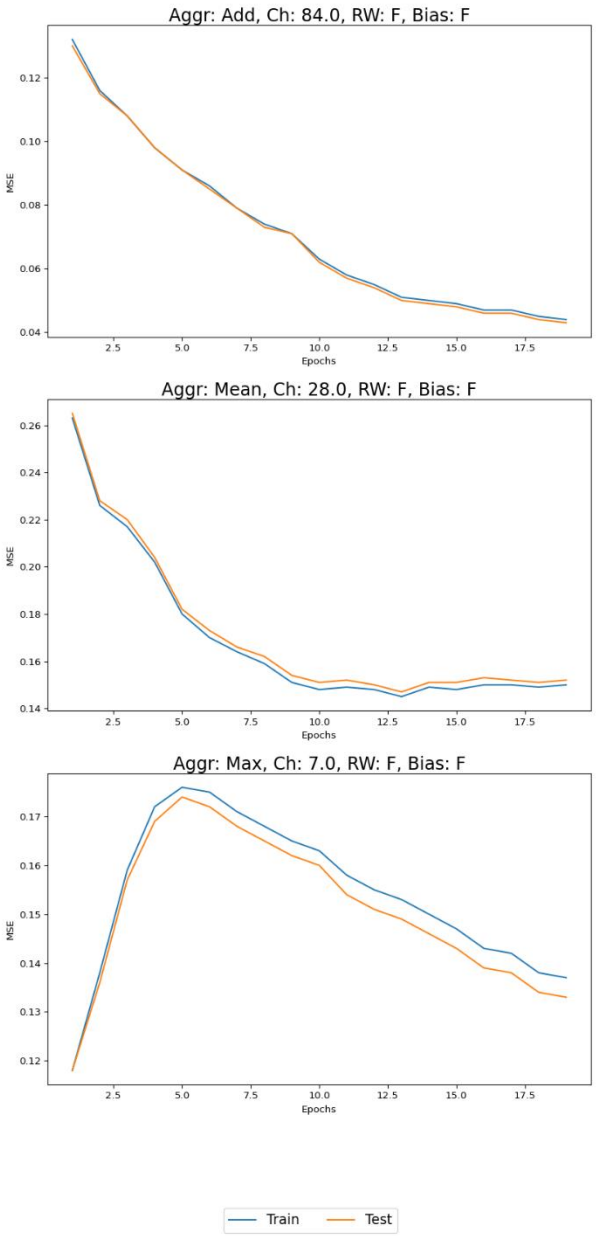

Figure 17: Predicting injector dispatch without root weight and bias

Observations: For Root Weight- False and Bias – False, the lowest MSE is obtained by keeping Aggregation as Add and hidden channels as 84.

Overall injector dispatch observations: We observe the least MSE with aggregation as Add, hidden channels as 84 and excluding root weight and bias. When both root weight and bias are not added, the test accuracy seems to be enhanced as evident from the corresponding mean aggregation model. This is the only case where we do not see the model overfitting. As the epochs increase, the accuracy seems to be decreasing in some cases but at the same time, continuing to increase the epochs, do increase the accuracy. This proves that more forward and backward propagation is helping tune the weights of the nodes.

7.2.4.3 Predicting UCED Together:

Input Variables: 8 Output Variables: 6

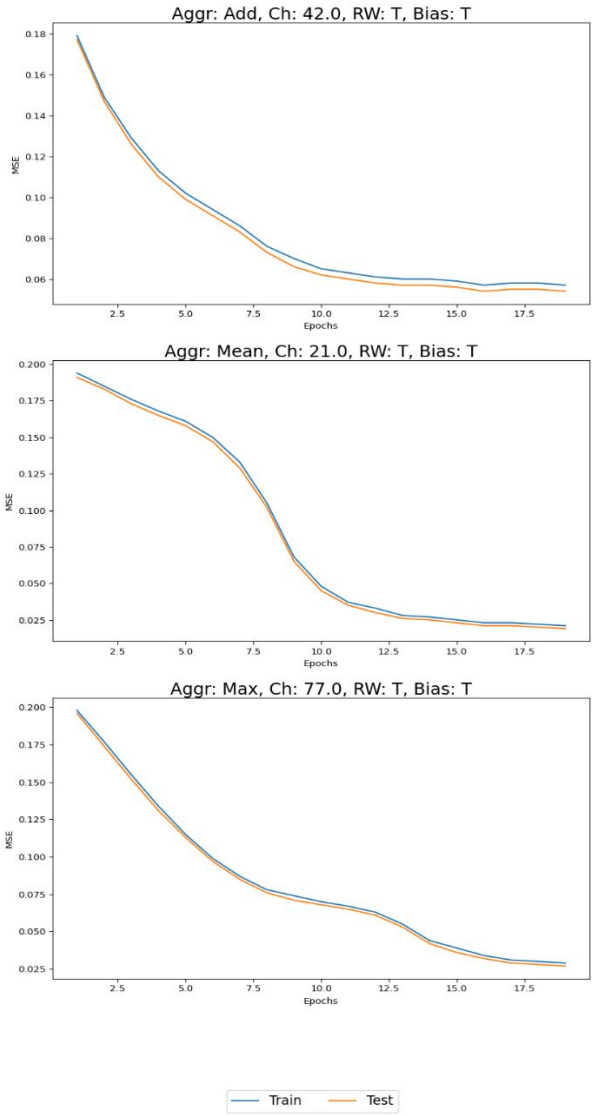

Figure 18: Predicting UCED with root weight and bias

Observations: For Root Weight- True and Bias – True, the lowest MSE is obtained by keeping Aggregation as Mean and hidden channels as 21.

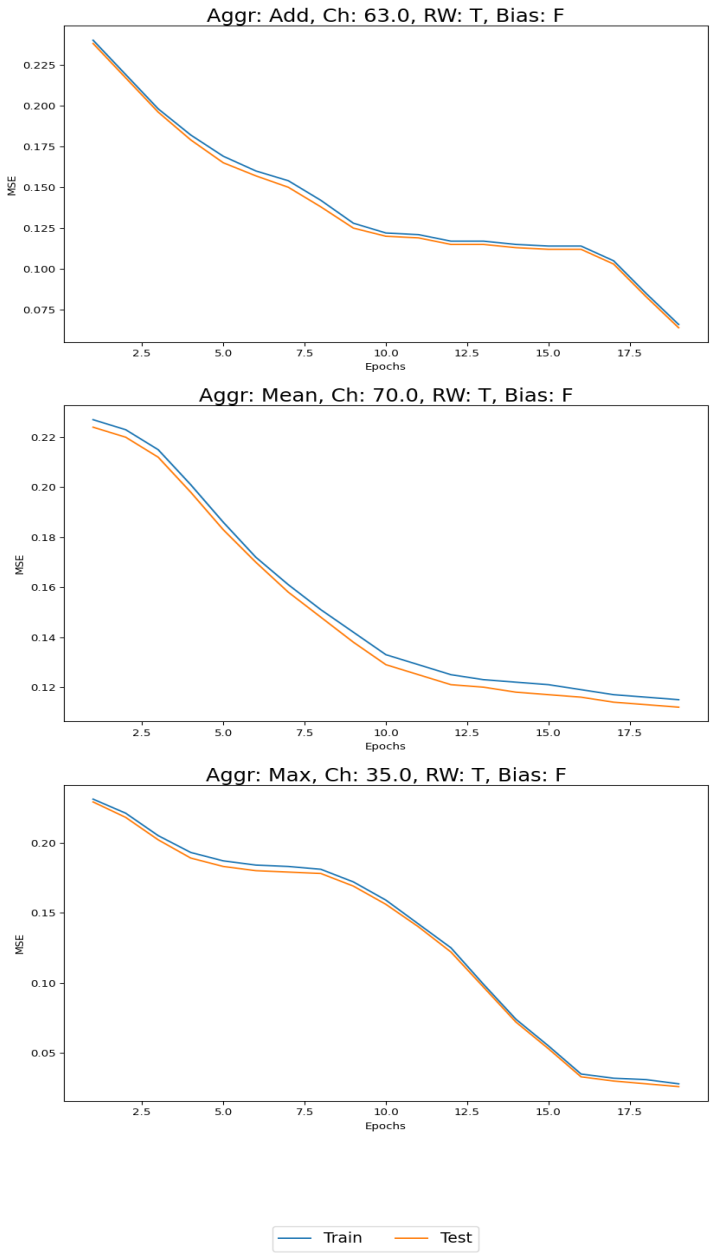

Figure 19: Predicting UCED with root weight and no bias

Observations: For Root Weight- True and Bias – False, the lowest MSE is obtained by keeping Aggregation as Max and hidden channels as 35.

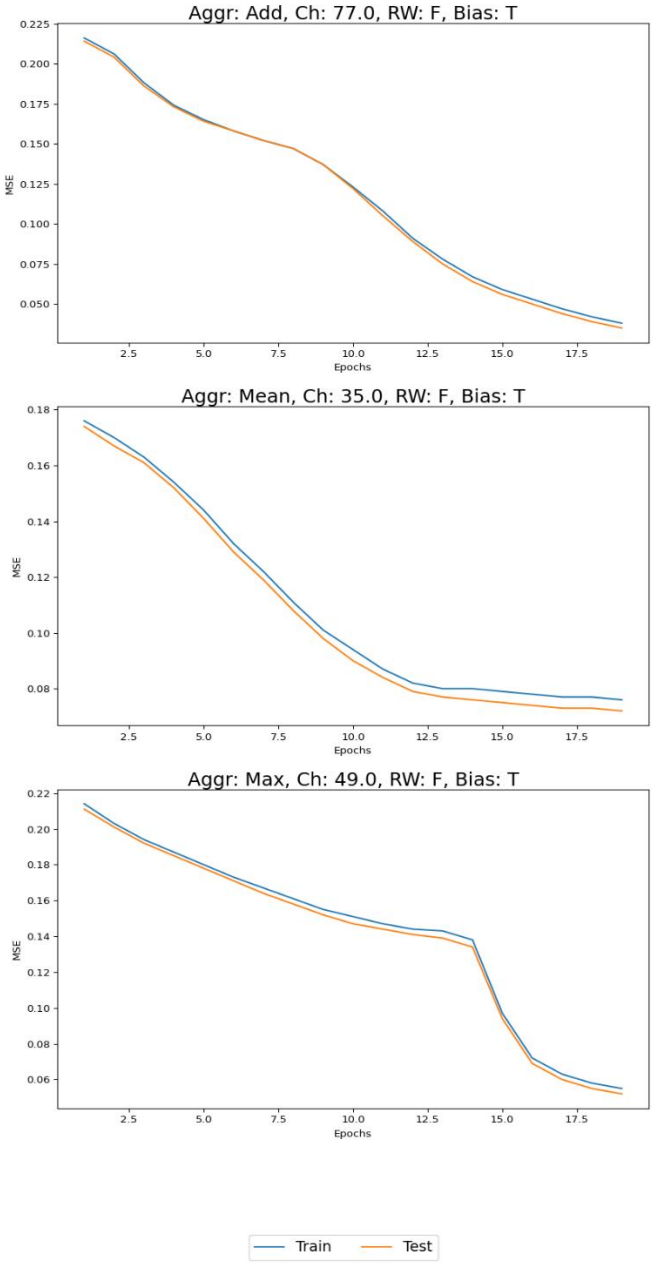

Figure 20: Predicting UCED without root weight but with bias

Observations: For Root Weight- False and Bias – True, the lowest MSE is obtained by keeping Aggregation as Add and hidden channels as 77.

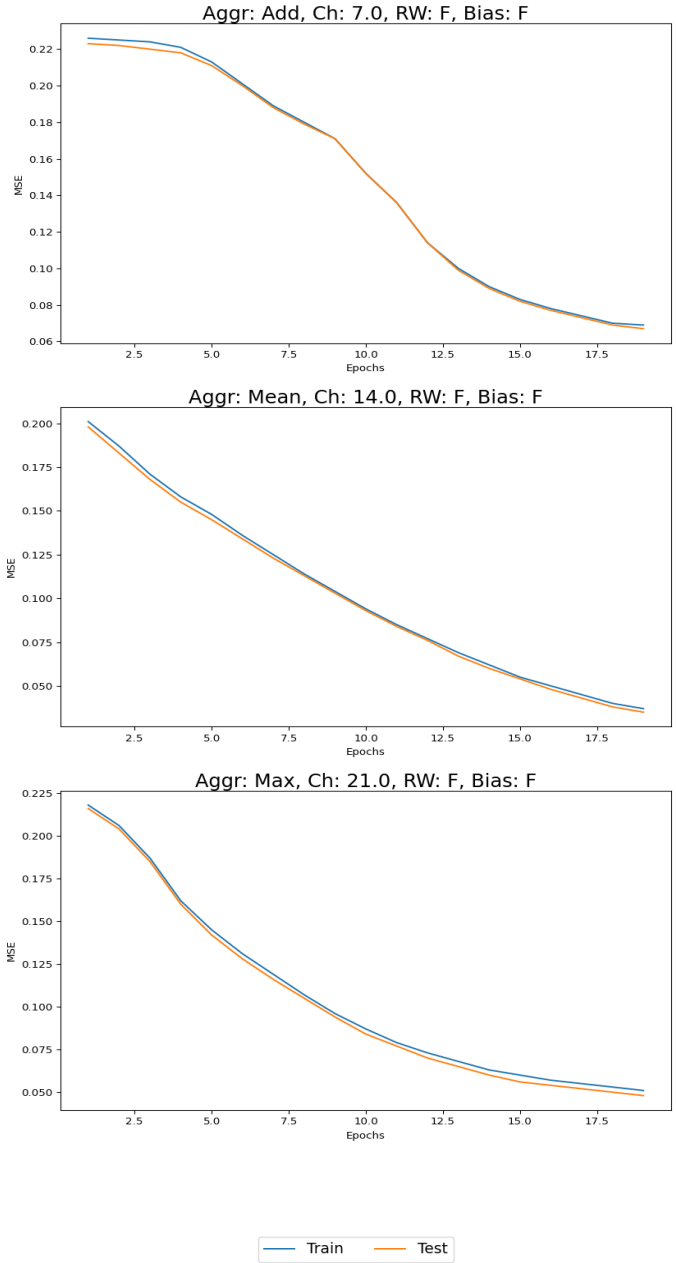

Figure 21: Predicting UCED without root weight and bias

Observations: For Root Weight- False and Bias – True, the lowest MSE is obtained by keeping Aggregation as Max and hidden channels as 21.

Overall UCED observations: We observe the least MSE with aggregation as Max, hidden channels as 77 and including root weight and bias. Root weight and bias does not seem to be enhancing the model as the results are almost similar for 20 epochs in every case. There is slight overfitting in almost all models as the train and test accuracy are almost similar and correspondingly test accuracy seems to be always better than the train accuracy.

8. Related Work

Unit Commitment and Economic dispatch has been extensively researched and numerous solutions have been proposed to solve this problem. [1] provides a comprehensive review of all the proposed studies to solve UCED problem till date. It lists a list of solutions with all of its advantages and weakness.

This chapter will summarize a few of the solutions that were studied to get a better understanding of each of the solutions to come up with this thesis.

8.1. Mixed Integer Linear Programming (MILP)

MILP is an mathematical optimization technique where some of the variable are constrained to be integers while others can be non-integers and the objective functions and the constraints are linear. [8] provides with a novel approach for solving the UC ED problem using MILP. It allows for topology definition and has the ability to support constraints as well. It is an extremely powerful modeling tool with an ability to reach a globally optimal solution. However, it is extremely inefficient and slow compared to methods like heuristics.

8.2. Genetics Algorithm

Genetics algorithm is a search heuristics-based algorithm that takes inspiration from Darwin's theory of natural evolution. It comprises of 5 different phases. Population, in this all candidates are populated. Fitness, in this phase each of the candidates from the population phase are passed through a fitness function. Selection, in this phase, best candidates are chosen based on their scores from the fitness test. Crossover, in this phase, two pairs of fit candidates are selected, and an offspring is created and added to the

population. Off springs inherit genes from parents. Mutation, in this phase certain probabilities are applied over the offspring to generate diversity within population to avoid premature convergence. The algorithm terminates when the population has converged and provides with a set of solutions.

[9] provides a genetic algorithm-based solution for the UC ED problem. The proposed solution in [9] implements a hybrid genetics algorithm comprised of integrating tabu search into genetics algorithm. They have achieved results for applying their algorithm to the UC ED problem and gives global optimal solution with lesser computation time.

8.3. Artificial Neural Network

Artificial Neural Networks have been widely used for a large domain of problems. They are famous for their ability to learn from its mistakes and that makes it extremely adaptable to wide range of problems. [10] proposes a ANN based solution for solving the UC ED problem. The proposed solution generates optimal generation schedule for generators using operational and load constraints. It utilizes B-Coefficients for evaluating the transmission losses in the system. The methodology is tested with six thermal power plants and the results have been found accurate when compared with classical methods.

ANN has proved capable of dealing with the variation of the data that the system may have and can be extremely flexible with the noise. However, the computation time augments exponentially for larger size problems

9. Conclusion and Future Work

9.1. Conclusion

GCN models like GCNConv: Graph Convolutional Network and NNConv MPNN was used to solve the Unit Commitment and Economic Dispatch problem. We represented the complex electric grid using a graph structure to retain all the nodes and edge constrains. We used the generated graph and applied two models on it. We conducted three experiments for which the results are as follows:

1. Predicting Injector Commitment:

GCNConv model performed better in this experiment compared to NNConv. The MSE error was reduced from 0.35 to 0.05. Running more epochs on NNConv model will help improving its performance.

2. Predicting Injector Dispatch:

NNConv model showed almost no overfitting for this experiment. However, GCNConv performed better comparatively.The MSE error was reduced from 0.16 to 0.02.

3. Predicting UC-ED together:

GCNConv performed the best for UC-ED predictions. However, the test MSE error is always less compared to train MSE error in NNConv.

The models learned on the electric grid data was able to perform surprisingly better and generate results with some error as compared with the PSO results. Both of the models are fast and scalable as compared to the MILP based PSO solver.

9.2. Future Work

All the above experiments were performed on a 7_Bus model. In Future we can perform the similar experiments on Real-Time Energy Market and observe the performance. It is expected that NNConv(MPNN) should give better results as it considers edge features as well as node features to perform convolution. Large number of Nodes and edges can be used to test the model behavior. Experiments can be made with different GNN models with proper parameter tuning.

10. References

[1] I. Abdou and M. Tkiouat, “Unit commitment problem in electrical power system: A literature review,” *Int. J. Electr. Comput. Eng.*, vol. 8, no. 3, pp. 1357–1372, 2018, doi: 10.11591/ijece.v8i3.pp1357-1372.

[2] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, “Graph convolutional networks : a comprehensive review,” *Comput. Soc. Networks*, 2019, doi: 10.1186/s40649-019-0069-y.

[3] T. N. Kipf and M. Welling, “SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS,” *Iclr*, pp. 1–14, 2017.

[4] R. K. Ando and T. Zhang, “Learning on graph with Laplacian regularization,” *Adv. Neural Inf. Process. Syst.*, pp. 25–32, 2007, doi: 10.7551/mitpress/7503.003.0009.

[5] H. Gao and S. Ji, “Graph U-nets,” *36th Int. Conf. Mach. Learn. ICML 2019*, vol. 2019-June, pp. 3651–3660, 2019.

[6] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural Message Passing for Quantum Chemistry,” *34th Int. Conf. Mach. Learn. ICML 2017*, vol. 3, pp. 2053–2070, 2017.

[7] P. C. St John *et al.*, “Message-passing neural networks for high-throughput polymer screening,” *J. Chem. Phys.*, vol. 150, no. 23, 2019, doi: 10.1063/1.5099132.