



CHALMERS
UNIVERSITY OF TECHNOLOGY



Using Graph Neural Networks to Learn Fire and Smoke Behaviour in a Physics-Based Simulator

Master's thesis in Physics

EDVIN LAM

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

MASTER'S THESIS 2021

Using Graph Neural Networks to Learn Fire and Smoke Behaviour in a Physics-Based Simulator

Edvin Lam



Department of Physics
Division of Subatomic, High Energy and Plasma Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Using Graph Neural Networks to Learn Fire and Smoke Behaviour in a Physics-Based Simulator

Edvin Lam

© Edvin Lam, 2021.

Supervisor: Olof Mogren, RISE Research Institutes of Sweden

Examiner: Andreas Ekström, Department of Physics

Master's Thesis 2021

Department of Physics

Division of Subatomic, High Energy and Plasma Physics

Chalmers University of Technology

SE-412 96 Gothenburg

Telephone: +46 31 772 1000

Typeset in L^AT_EX

Printed by Chalmers Reproservice

Gothenburg, Sweden 2021

Using Graph Neural Networks to Learn Fire and Smoke Behaviour in a Physics-Based Simulator

Edvin Lam

Department of Physics

Chalmers University of Technology

Abstract

Deep learning is prominent in computer vision as a means of learning spatial features and processing them in useful ways, such as for image classification and image segmentation. An application of this in physics is to use videos of fire to predict how the flames and smoke evolve over time. This thesis explores a simplified version of this in an attempt to lay the groundwork for future studies. In particular, simulations of fire and smoke were made using the open source software Fire Dynamics Simulator by NIST. From these simulations, frame by frame states of the physical system could be extracted. A graph neural network using the MeshGraphNets framework was trained to predict future states of the fire given two past states as input. Some of the resulting neural networks were able to reasonably predict over 50 frames into the future and also showed some competence when faced with test sets with a different physical geometry. The performances of the trained models varied greatly depending on the loss functions used in training, suggesting that more hyperparameter optimisation could be done in this regard.

Keywords: graph neural network, fire dynamics simulator, deep learning, physics simulation.

Acknowledgements

I would first and foremost like to thank my supervisor Olof Mogren, who provided me with the guidance needed to complete this thesis. He helped push me towards the right direction and took the time to arrange for meetings with many other helpful colleagues at RISE. I would like to thank the colleagues, too, for without their insights my work wouldn't have proceeded as well and as quickly as it did. Finally, I would also like to thank my examiner Andreas Ekström, who helped me with the logistics of planning and working on a master's thesis.

Edvin Lam, Gothenburg, June 2021

Table of Contents

1	Introduction	1
1.1	Specification	1
1.2	Related Work	1
1.3	Ethical Concerns	2
2	Background	5
2.1	Machine Learning Concepts	5
2.2	Neural Networks and Deep Learning	6
2.2.1	Graph Neural Networks	8
2.3	Mesh-based Physical Systems	10
2.3.1	Next State Calculation	10
3	Methods	13
3.1	MeshGraphNets	13
3.1.1	The GNN Architecture	14
3.1.2	Targets, Rollouts and Loss Functions	16
3.2	Implementation Details	18
3.2.1	GNN Details	18
3.2.2	The FDS Datasets	19
3.2.3	Training	22
3.3	Metrics for Model Evaluation	23
3.4	Baseline Comparison with UNet	25
3.5	Test Cases for Generalisation	25
4	Results	27
4.1	SmallSet	27
4.2	BigSet	32
4.3	Domain Generalisation	37
4.4	GNN Performance	39
4.5	Incorrect Data	40
5	Discussion	43
5.1	Conclusion	45
	References	47

1. Introduction

Computer vision has come a long way, with deep learning at its core. By using deep neural networks, spatial features can be computed for images and be processed for many different purposes, such as image classification and image segmentation. Conversely, images can also be generated from an input prompt with the help of generative models.

An interesting question is whether neural networks can be used to process and generate time series images—that is, videos—of physical systems evolving over time, using a short video as input. Generating videos of complex physical systems can be a computationally costly endeavour for traditional, numerical approaches, so having neural networks that can do it faster is of particular interest. Graph neural networks have in recent studies (such as Pfaff et al., 2021, Sanchez-Gonzalez et al., 2018, Watters et al., 2017 and Battaglia et al., 2016) been deployed to learn simulated physical systems, with much success. The limitations of these studies resulted in a compromise between either reducing the physical system to a small number of interacting particles, or having more feature channels in the video input than just the typical RGB colour channels. The goal of this thesis is to explore the latter compromise, with the prospect of possibly bridging the gap in the future.

1.1 Specification

This thesis will explore using more detailed features than just RGB video to learn a complex physical system. A system of fire and smoke will be studied, using the open source software Fire Dynamics Simulator developed by NIST (2020) to generate time series data of evolving fire and smoke. This software can export many different quantities from the state of the system, such as temperature, gas density and gas velocity, at every point in the simulation space. These features will be used as training data to teach a graph neural network the dynamics of the system, using a few consecutive states as inputs in order to predict a future state. The network’s generalisation capabilities will also be tested on systems with geometries different from that of the training data. Transitioning into using RGB video inputs can be done with the complementary software Smokeview, which is able to export the FDS simulations as images; this is however outside the scope of the study.

1.2 Related Work

Using visual input to predict physical systems has been done with the Visual Interaction Network, or VIN for short. The VIN is a neural network developed Watters

et al. (2017) that can predict the movement of balls that interact with each other, either due to gravity or due to some other physical processes like elastic collision. VIN uses convolutional layers to encode sequences of images of balls into state codes for each individual ball. It then uses Interaction Networks—a network architecture introduced by Battaglia et al. (2016) that interprets objects as nodes in a graph and their interactions with each other as edges between nodes—to predict the future state codes of the balls, which then gets decoded into their positions and velocities. This method was successful, but the approach only works for physical systems that consist of a fixed number of interacting particles.

The MeshGraphNets framework by Pfaff et al. (2021) allows for a graph neural network to learn a physical system modelled as a field, where every point in space is assigned a vector of quantities such as temperature and velocity. Unlike the VIN, which uses a series of images to discern individual particles, the MeshGraphNets framework uses a graph representing a discretised version of the field as input. This allows for learning macroscopic systems that consist of a huge number of particles, and it works better than more conventional convolutional networks such as the UNet.

Fire Dynamics Simulator has been a subject of deep learning research before. Hodges et al. (2019) used a transpose convolutional neural network to spatially model temperature and gas flow velocities of fires in compartmentalised rooms, given general information about the rooms’ geometries, ventilations, and source fire intensity and location. The network did however not learn to predict future time steps; instead, it was trained to predict the future time-averaged spatial features.

1.3 Ethical Concerns

Since this work researches application of machine learning, there are always ethical concerns to be aware of. Although using simulations to train a neural network is essentially risk-free—the only concerns are environmental, since simulating data and training the network requires a lot of computing power—rolling out the network to real world use cases can be problematic.

Trained neural networks are essentially black boxes, meaning that they seem to work without us knowing exactly how they work. In this case, the trained model might be able to predict fire and smoke patterns in many scenarios, but fail catastrophically in edge cases that weren’t in the training data. If the model is used for e.g. evaluating fire safety, this risk of failure is unacceptable. Using black boxes like this is therefore grounds for ethical concerns in itself. One must thus very thoroughly evaluate and train the model before use.

This leads to another concern, which is where the training data comes from. Simulations are approximations of the real world, so in order to have a well-trained neural network, it is advisable for it to have been trained on real data. One way of gathering such data is to intentionally start controlled fires, but one again runs the risk of not testing every possible scenario. The other option is using footage of

real fires for training. This option is ethically concerning, especially if the fires have caused any fatalities.

2. Background

This chapter will describe machine learning and some necessary mathematics behind modelling a physical system as a mesh.

2.1 Machine Learning Concepts

This section will give a brief explanation of important concepts in machine learning in the context of neural networks and deep learning. The content is based largely on *Deep Learning* by Goodfellow et al. (2016) and *Machine learning with neural networks: An introduction for scientists and engineers* by Mehlig (2021).

In the most general case, a neural network is a function $f(\mathbf{x}, \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$ that takes an input \mathbf{x} that is designed to approximate a target function $f^*(\mathbf{x})$. Not all machine learning methods utilise parametrised functions, but this thesis will be limited to only such cases. The individual elements in \mathbf{x} are called features, and are quantities that describe whatever is relevant to the problem at hand. In a single particle system, the features in \mathbf{x} could for example be the particle’s position, velocity and mass. Training a neural network means finding the right $\boldsymbol{\theta}$ so that the network $f(\mathbf{x}, \boldsymbol{\theta})$ approximates the target function $f^*(\mathbf{x})$ well enough. This target function could range from being a function that correctly predicts tomorrow’s temperature given today’s weather conditions \mathbf{x} , to a function that correctly identifies whether an image \mathbf{x} depicts a cat or a dog. A trained neural network is also called a model.

Training a model generally involves using a set of pairs of training inputs \mathbf{x}^i and their corresponding targets $\mathbf{y}^i = f^*(\mathbf{x}^i)$. An instance of these pairs $(\mathbf{x}^i, \mathbf{y}^i)$ is called a training example, and the set of these pairs is called a training set, $\mathbb{X} = \{(\mathbf{x}^i, \mathbf{y}^i)\}$. The training is performed by minimising a loss function $J(\boldsymbol{\theta})$ that depends on the model’s parameters and implicitly on the training set \mathbb{X} itself. The loss function is a measure of how well a model f with parameters $\boldsymbol{\theta}$ approximates f^* for a given set of examples \mathbb{X} , and can come in many forms. A commonly used loss function for regression is the MSE loss, or the mean squared error loss,

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_i \left(\mathbf{y}^i - f(\mathbf{x}^i, \boldsymbol{\theta}) \right)^2, \quad (2.1)$$

where N is the number of training examples. The loss is also a scalar, so the per-feature mean of the squared errors is also taken. One can minimise $J(\boldsymbol{\theta})$ with gradient descent, which is an algorithm—also called an optimiser—where $\boldsymbol{\theta}$ is successively moved towards the opposite direction of the loss function’s gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

This leads to the parameter update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \quad (2.2)$$

for some positive, usually small, learning rate η . Although this can find a local minimum, it's not guaranteed to find the global minimum.

Using gradient descent on the whole set of data at once is called batch gradient descent. This method of training the model is very expensive for large data sets, since the whole batch of data might not fit into the memory. A way to alleviate this problem is by using stochastic gradient descent, or SGD. This samples the training examples one by one and performs a gradient descent for every such sample. A similar algorithm is the minibatch SGD, which samples minibatches of data and updates the parameters accordingly. When the parameters have been updated for every minibatch in the training set, one *epoch* of training has been completed.

This kind of training can be done over many epochs. One wants to arrive at parameters $\boldsymbol{\theta}$ that neither *underfits* nor *overfits* the data. In general terms, underfitting is when the model hasn't learned enough while overfitting is when the model has learned the training set well but can't generalise to other inputs. A so-called validation set or test set \mathbb{X}_{val} can be used to measure how well a model generalises. The validation set is similar to the training set, but consists of an independent set of examples that aren't used to train the model. By keeping track of both the loss of the training set and the loss of the validation set during training, one can get an indication of how the training is going; if both the training and validation losses are still decreasing, the current model is underfitted and can still improve, but if the training loss is decreasing while the validation loss is not, the model is probably overfitted.

There are many other kinds of optimisers based on the SGD optimiser. The Adam optimiser (Kingma et al., 2017) adapts the learning rate for individual parameters over the course of the training which can result in better performance for some datasets. AdamW (Loshchilov et al., 2019) is a modification of Adam which penalises large values of $\boldsymbol{\theta}$ in an attempt to reduce overfitting, and has been shown to perform better than Adam.

2.2 Neural Networks and Deep Learning

Neural networks is a family of functions inspired by neurons in the brain. One of the simplest kind of neural networks are the fully connected neural networks, also called multilayer perceptrons, or MLPs. These are a class of neural networks that consist of an input layer, some number of hidden layers, and an output layer, in that order. A layer can be seen as a vector whose elements are an affine transformation of all the elements in the previous layer, composed with an activation function. The name fully connected neural network comes from this affine transformation, since every element in a layer is connected to all the elements in the previous layer. This

connectedness is shown in figure 2.1, which is a diagram over an MLP with only a single hidden layer. Such a network takes a vector input \mathbf{x} with N elements at the input layer, transforms the input to a vector \mathbf{V} with H elements in the hidden layer and then transforms it into an output \mathbf{O} with M elements at the output layer. While N and M depends on the input and output of the target function $f^*(\mathbf{x})$ that the network is to approximate, the hidden layer width H is a model hyperparameter that can freely be chosen. The individual elements in the vectors are also referred to as nodes, so the hidden layer has H nodes. Within the transformations from \mathbf{x} to \mathbf{V} to \mathbf{O} are weights and biases that together make up the model's learnable parameters θ .

Input Layer Hidden Layer Output Layer

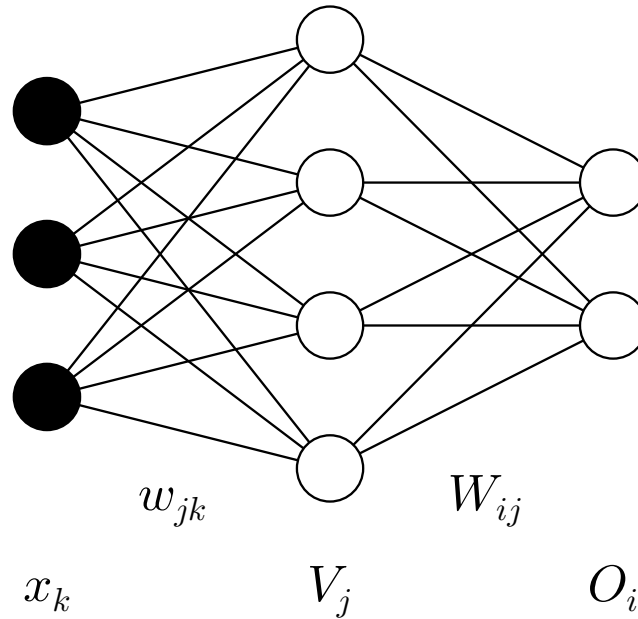


Figure 2.1: An MLP with a single hidden layer. The input layer has $N = 3$ nodes x_k , the hidden layer has $H = 4$ nodes V_j and the output layer has $M = 2$ nodes O_i . The edges connecting the nodes to each other are the weights w_{jk} and W_{ij} , which determine the affine transformation between layers. Not shown are the biases and activation functions.

The hidden layer vector \mathbf{V} is computed through

$$V_j = g \left(\sum_{k=1}^N w_{jk} x_k + b_j \right), \quad (2.3)$$

where b_j are the biases, w_{jk} are the weights and g is an activation function. Likewise, the output \mathbf{O} is then computed with

$$O_i = g' \left(\sum_{j=1}^H W_{ij} V_j + B_i \right) \quad (2.4)$$

with its own set of weights W_{ij} , biases B_i and activation function g' . The whole vector of parameters θ thus contains \mathbf{w} , \mathbf{W} , \mathbf{b} and \mathbf{B} .

The activation functions, g and g' in the above example, are chosen per layer. These are usually non-linear functions, the most popular being the ReLU function $g(x) = \max(0, x)$. Models for regressive tasks, that is, where the outputs of the target function f^* are on a continuous interval in the real number line, often use a linear activation function $g(x) = x$ at the output layer.

MLPs aren't restricted to a single hidden layer. Multiple hidden layers, each with their own number of nodes and activation functions, can be used to obtain a deep MLP. The depth of a neural network generally refers to the number of hidden layers it has, and the associated training and usage of networks with more than one hidden layer is called deep learning.

There are other modifications one can make to the MLPs. One is the residual connection, or skip connection. Instead of purely forwarding an input \mathbf{x} through a series of hidden layers $h(\mathbf{x})$, one can define a residual connection $f(\mathbf{x}) = h(\mathbf{x}) + \mathbf{x}$. This has empirically been shown to be easier to train than standard MLPs, likely due to how the gradients are calculated.

Another modification is to introduce normalisation in an MLP. Layer normalisation is one such method, proposed by Ba et al. (2016). It works by normalising the nodes in a layer to zero mean and unit variance, and has been shown to improve training speed and performance.

2.2.1 Graph Neural Networks

A particular class of neural network architectures is the graph neural network, or GNN. A graph $G = (V, E)$ is an object with nodes V and edges E that connects nodes with each other. A node $i \in V$ contains the features \mathbf{v}_i while an edge $(i, j) \in E$ going from the sender node j to the receiver node i can contain features \mathbf{e}_{ij} . Graph neural networks take graphs G as input and compute an output that depends on the task at hand. These tasks could range from a per node or per edge regression to a full graph regression. In the layers of a graph neural network, messages get passed between the nodes according to their edges in order to facilitate learning relations that are dependent on the graph topology. This could be seen as a generalised convolutional neural network that works for non-Euclidean geometry. A comprehensive review of graph neural networks with many of their use cases is given by Zhou et al. (2021).

A kind of GNN of particular interest for this work is the Interaction Network, or IN, as described by Battaglia et al. (2016). It characterises a physical system into a graph with the nodes being the physical objects and the edges being the objects' interaction with each other. The IN was then generalised by Sanchez-Gonzalez et al. (2018) to a so-called Graph network, GN, which takes as input and outputs a graph

G along with global features \mathbf{x}_g . This architecture is summarised in figure 2.2.

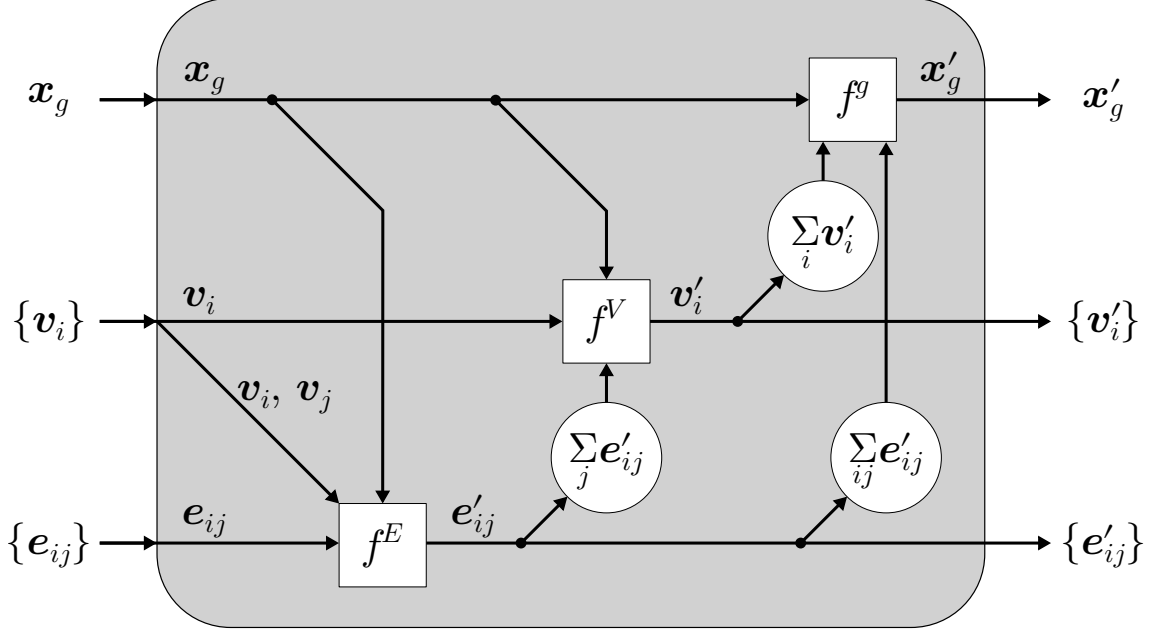


Figure 2.2: The architecture of the Graph network, with the input nodes $\{v_i\}$, input edges $\{e_{ij}\}$, input global features \mathbf{x}_g , output nodes $\{v'_i\}$, output edges $\{e'_{ij}\}$ and output global features \mathbf{x}'_g . The message passing with this architecture is performed with the MLPs f^E and f^V , while the f^g is an MLP that updates the global features.

From the GN's input (\mathbf{x}_g, G) , the global features \mathbf{x}_g , an edge e_{ij} and its respective sender and receiver nodes v_i and v_j get concatenated and passed through an MLP f^E that outputs a new edge e'_{ij} . This is done for every edge in the graph with the same MLP f^E . After updating the edges, the GN takes the global features \mathbf{x}_g , a node v_i and an aggregation of every edge the node receives, $\hat{e}_i = \sum_j e'_{ij}$, and pass the concatenation of these through an MLP f^V that outputs a new node v'_i . This process is repeated for every node using an identical f^V . Finally, the global features gets updated with an MLP f^g that takes a concatenation of \mathbf{x}_g , $\hat{e} = \sum_{ij} e'_{ij}$ and $\hat{v} = \sum_i v'_i$ as input. These updates to the features can be summarised as

$$e'_{ij} \leftarrow f^E(e_{ij}, v_i, v_j), \quad v'_i \leftarrow f^V\left(v_i, \sum_j e'_{ij}\right), \quad \mathbf{x}'_g \leftarrow f^g\left(\mathbf{x}_g, \sum_{ij} e'_{ij}, \sum_i v'_i\right). \quad (2.5)$$

The whole GN input to output pipeline is detailed in algorithm 1.

A GN in a network can be treated as a single layer. Several GNs can be composed together to form a deep graph neural network with multiple layers. Since a single GN layer has messages passing between neighbouring nodes, a network with n GN

layers can pass messages up to n nodes away.

Algorithm 1: Graph network

Input: $G = (\{\mathbf{v}_i\}, \{\mathbf{e}_{ij}\})$, \mathbf{x}_g

for each edge \mathbf{e}_{ij} **do**

 Gather receiver and sender nodes \mathbf{v}_i and \mathbf{v}_j ;

 Compute output edges $\mathbf{e}'_{ij} = f^E(\mathbf{x}_g, \mathbf{e}_{ij}, \mathbf{v}_i, \mathbf{v}_j)$;

end

for each node \mathbf{v}_i **do**

 Aggregate edges with \mathbf{v}_i as receiver, $\hat{\mathbf{e}}_i = \sum_j \mathbf{e}'_{ij}$;

 Compute output nodes $\mathbf{v}'_i = f^V(\mathbf{x}_g, \mathbf{v}_i, \hat{\mathbf{e}}_i)$;

end

Aggregate all edges and nodes, $\hat{\mathbf{e}} = \sum_{ij} \mathbf{e}'_{ij}$, $\hat{\mathbf{v}} = \sum_i \mathbf{v}'_i$;

Compute output global features, $\mathbf{x}'_g = f^g(\mathbf{x}_g, \hat{\mathbf{e}}, \hat{\mathbf{v}})$;

Output: $G' = (\{\mathbf{v}_i\}', \{\mathbf{e}'_{ij}\})$, \mathbf{x}'_g

2.3 Mesh-based Physical Systems

A physical system can be modelled as a vector valued continuous field, $\mathbf{q}(\mathbf{r}, \tau)$, where every point \mathbf{r} in space at some time τ has an assigned vector that describes some quantities of the system. This can be spatially discretised into a mesh, where each cell i contains a sample of the field, or feature vector, $\mathbf{q}^i(\tau) = \mathbf{q}(\mathbf{r}_i, \tau)$ and where \mathbf{r}_i is the location of the cell. The complete state of the mesh at time τ is then defined as $\mathbf{s}(\tau) = \{\mathbf{q}^i(\tau) \mid \forall i \in V\}$, with V being the set of all cells in the mesh.

2.3.1 Next State Calculation

Assuming that the physical system is continuous in time up to the second order, the Taylor expansion of the state $\mathbf{s}(\tau)$ around the time $\tau = t$ is

$$\mathbf{s}(\tau) \approx \mathbf{s}(t) + (\tau - t)\dot{\mathbf{s}}(t) + \frac{(\tau - t)^2}{2}\ddot{\mathbf{s}}(t), \quad (2.6)$$

with $\dot{\mathbf{s}}(t)$ and $\ddot{\mathbf{s}}(t)$ being the first and second time derivative, respectively ¹. Evaluating $\mathbf{s}(t + 1)$ then yields

$$\mathbf{s}(t + 1) \approx \mathbf{s}(t) + \dot{\mathbf{s}}(t) + \frac{1}{2}\ddot{\mathbf{s}}(t). \quad (2.7)$$

This approximation is valid if the time unit is small enough. The first time derivative in equation 2.7 can be approximated with $\dot{\mathbf{s}}(t) \approx \mathbf{s}(t) - \mathbf{s}(t - 1) = \Delta\mathbf{s}(t)$, resulting in

$$\mathbf{s}(t + 1) \approx \mathbf{s}(t) + \Delta\mathbf{s}(t) + \frac{1}{2}\ddot{\mathbf{s}}(t). \quad (2.8)$$

¹The addition used here is performed element-wise between feature vectors belonging to the same cell, e.g. $\mathbf{s}(t) + \dot{\mathbf{s}}(t) = \{\mathbf{q}^i(t) + \dot{\mathbf{q}}^i(t) \mid \forall i \in V\}$.

In a time-discrete notation, this can be expressed as

$$\mathbf{s}_{t+1} \approx \mathbf{s}_t + \Delta \mathbf{s}_t + \frac{1}{2} \ddot{\mathbf{s}}_t. \quad (2.9)$$

From this, it is apparent that one can calculate the state in a future time given the current state, the past state and the current state acceleration as long as the time step is small enough. This kind of state update was used by Pfaff et al. (2021) in their MeshGraphNets framework.

3. Methods

This chapter will introduce the MeshGraphNets framework, which will be used to model and train a graph neural network to predict future states in a physical system given two past states. The training data is obtained from FDS simulations, which will also be detailed. The data will be split into multiple episodes, with each episode corresponding to a single, completed simulation. In turn, each episode contains multiple training examples for the neural network.

3.1 MeshGraphNets

MeshGraphNets is a framework for using a graph neural network to learn a mesh-based physical system, developed by Pfaff et al. (2021). It represents the state of a mesh-based physical system \mathbf{s}_t , its velocity mesh $\Delta\mathbf{s}_t$ and its geometry mesh $\mathcal{G}(\mathbf{s})$ as a graph G_t , where every node i corresponds to a cell in the mesh with some mesh-space coordinate \mathbf{u}_i and are connected to its neighbours with edges. The framework allows for the mesh to represent a Lagrangian system, where the mesh can be a deforming surface with real world coordinates \mathbf{x}_i that differ from the mesh-space coordinates \mathbf{u}_i . This work will however only consider an Eulerian system, where the mesh is fixed in real world coordinates. In tandem with that, this work won't consider dynamically remeshing the system.

The features in the nodes are the mesh's feature vector \mathbf{q}_t^i , the velocity $\Delta\mathbf{q}_t^i = \mathbf{q}_t^i - \mathbf{q}_{t-1}^i$ and a one-hot geometry vector \mathbf{g}^i that indicates whether there is any solid material in cell i , defined as

$$\mathbf{g}^i = \begin{cases} (1, 0) & \text{if } i \text{ contains solid} \\ (0, 1) & \text{otherwise.} \end{cases} \quad (3.1)$$

This one-hot encoding with two elements is redundant, but allows for future extensions with more granular geometry segmentation, such as an element indicating whether a cell is a burner. Meanwhile, the features in an edge between node i and j were their relative mesh-space coordinates and the length of that coordinate, $\mathbf{e}_{ij} = (\mathbf{u}_i - \mathbf{u}_j, |\mathbf{u}_i - \mathbf{u}_j|)$. Since two time steps are need to calculate the features for the nodes, the states needed to calculate the whole graph at a time t are \mathbf{s}_{t-1} and \mathbf{s}_t .

The MeshGraphNets framework uses a GNN with the graph G_t as input to predict the future state \mathbf{s}_{t+1} . Specifically, the output \mathbf{p} of the GNN is trained to approximate $\frac{1}{2}\dot{\mathbf{s}}_t$. This in turn can be used to calculate the next state by using equation 2.9, where

$\mathbf{s}_{t+1} = \mathbf{s}_t + \Delta \mathbf{s}_t + \mathbf{p}$. Figure 3.1 shows the pipeline of calculating features for the graph, using those features in the GNN and predicting the next state.

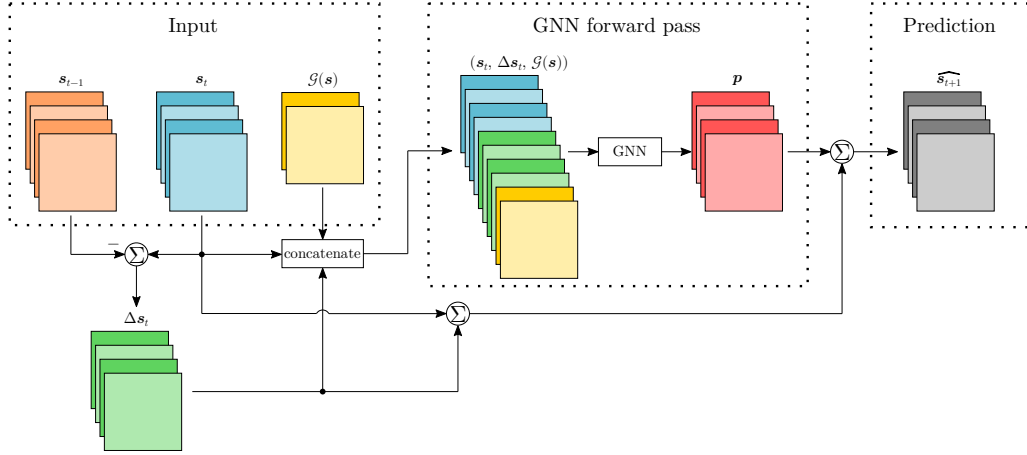


Figure 3.1: A schematic overview of how the next state \mathbf{s}_{t+1} is predicted from an input consisting of \mathbf{s}_t , \mathbf{s}_{t-1} and $\mathcal{G}(\mathbf{s})$ using the MeshGraphNets framework. The approximate state velocity $\Delta \mathbf{s}_t$ is calculated from the two input states. The feature-wise concatenation of \mathbf{s}_t , $\Delta \mathbf{s}_t$ and $\mathcal{G}(\mathbf{s})$ is then turned into a graph G_t at the input to GNN, which in turns outputs the predicted acceleration \mathbf{p} of the system. Summation of this predicted acceleration, the state \mathbf{s}_t and the velocity $\Delta \mathbf{s}_t$ yields a prediction of the next state $\widehat{\mathbf{s}_{t+1}}$.

3.1.1 The GNN Architecture

The GNN used in the MeshGraphNets framework operates with an encode-process-decode architecture. That is, the GNN encodes the input graph into a latent graph with abstract node and edge features while preserving the graph topology. The latent graph is then processed through a series of GN layers. The features of the nodes of the graph at the final GN output are then decoded to the GNN’s output \mathbf{p} . Notably, the GNs used in this framework neither take as input nor give as output any global features \mathbf{x}_g . The GNN used is detailed in algorithm 2. A schematic of the modified GN layers is illustrated in figure 3.2, and a schematic of the whole GNN is shown in figure 3.3.

At the encoder, the node and edge features in the graph are encoded into latent features with two MLPs ϵ^V and ϵ^E , respectively. The same ϵ^V are used for every node while the same ϵ^E are used for every edge, meaning that the encoding process is spatially invariant.

The processor contains L number of GN layers, each with identical architecture but different learnable parameters. Since no global features were used, the update algorithm from equation 2.5 is instead just

$$\mathbf{e}_{ij}^l \leftarrow f_l^E(\mathbf{e}_{ij}^{l-1}, \mathbf{v}_i^{l-1}, \mathbf{v}_j^{l-1}), \quad \mathbf{v}_i^l \leftarrow f_l^V(\mathbf{v}_i^{l-1}, \sum_j \mathbf{e}_{ij}^{l-1}) \quad (3.2)$$

at the l th layer in the processor.

The decoder simply takes the node features from the processed graph and applies an MLP δ node-wise to decode the latent features and then reconstruct it to a mesh \mathbf{p} with the same dimensions as a state \mathbf{s}_t .

Algorithm 2: GNN used in MeshGraphNets

Input: $G = (\{\mathbf{v}_i\}, \{\mathbf{e}_{ij}\})$

```

/* Encoder                                                                    */
for each edge  $\mathbf{e}_{ij}$  do
    | Encode edges  $\mathbf{e}_{ij}^0 = \epsilon^E(\mathbf{e}_{ij})$ ;
end
for each node  $\mathbf{v}_i$  do
    | Encode nodes  $\mathbf{v}_i^0 = \epsilon^V(\mathbf{v}_i)$ ;
end

/* Processor                                                                    */
for  $l$  from 1 to and including  $L$  do
    for each edge  $\mathbf{e}_{ij}^{l-1}$  do
        | Gather receiver and sender nodes  $\mathbf{v}_i^{l-1}$  and  $\mathbf{v}_j^{l-1}$ ;
        | Process edges  $\mathbf{e}_{ij}^l = f_l^E(\mathbf{e}_{ij}^{l-1}, \mathbf{v}_i^{l-1}, \mathbf{v}_j^{l-1})$ ;
    end
    for each node  $\mathbf{v}_i^{l-1}$  do
        | Aggregate processed edges with  $\mathbf{v}_i^{l-1}$  as receiver,  $\hat{\mathbf{e}}_i = \sum_j \mathbf{e}_{ij}^l$ ;
        | Process nodes  $\mathbf{v}_i^l = f_l^V(\mathbf{v}_i^{l-1}, \hat{\mathbf{e}}_i)$ ;
    end
end

/* Decoder                                                                    */
for each node  $\mathbf{v}_i^L$  do
    | Decode nodes  $\mathbf{p}_i = \delta(\mathbf{v}_i^L)$ ;
end

Output:  $\mathbf{p} = \{\mathbf{p}_i\}$ 

```

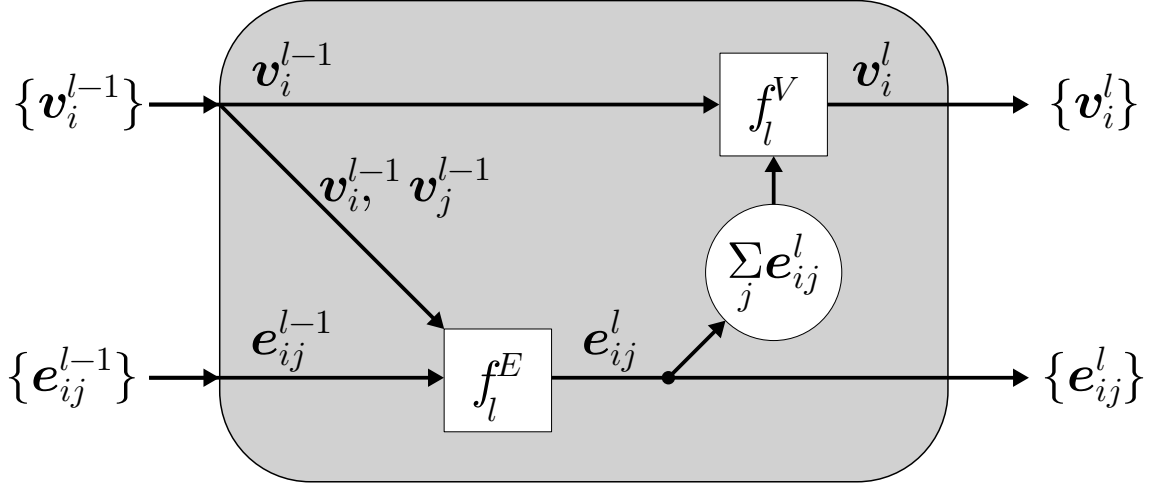


Figure 3.2: The GN layers used by the MeshGraphNets framework. Unlike the GN described by Sanchez-Gonzalez et al. (2018), this version doesn’t use global features \mathbf{x}_g .

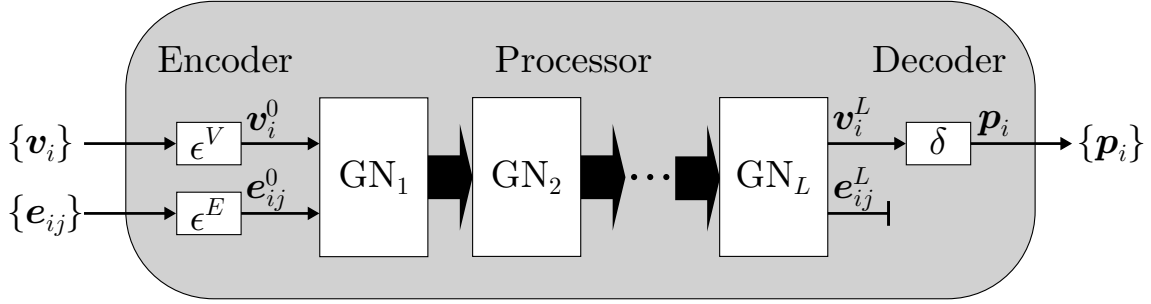


Figure 3.3: The architecture of the GNN used in the MeshGraphNets framework. At the encoder, the MLPs ϵ^V and ϵ^E encode every node \mathbf{v}_i and edge \mathbf{e}_{ij} into latent nodes and edges. These then get processed through L GN layers before the nodes are decoded by the MLP δ into node-wise predictions \mathbf{p}_i .

3.1.2 Targets, Rollouts and Loss Functions

The goal is to have the GNN learn to output a prediction of the current acceleration, $\mathbf{p} = \frac{1}{2}\hat{\mathbf{s}}_t$, given an input $(\mathbf{s}_t, \Delta\mathbf{s}_t, \mathcal{G}(\mathbf{s}))$. By using equation (2.9) one obtains a prediction of the future state $\widehat{\mathbf{s}}_{t+1} = \mathbf{p} + \mathbf{s}_t + \Delta\mathbf{s}_t$. A desired capability is for the model to be able to predict many states into the future. A prediction like this can be done with the same GNN model; the predicted state $\widehat{\mathbf{s}}_{t+1}$ can be reused as an input $(\widehat{\mathbf{s}}_{t+1}, \widehat{\Delta\mathbf{s}}_{t+1}, \mathcal{G}(\mathbf{s}))$ to predict $\frac{1}{2}\widehat{\mathbf{s}}_{t+1}$, which in turn can be reused to predict yet another future state, et cetera. A series of consecutive predictions n time steps into the future is called a rollout of length n . Algorithm 3 details how a rollout is calculated given an initial input $(\mathbf{s}_t, \Delta\mathbf{s}_t, \mathcal{G}(\mathbf{s}))$ to the model f .

Two different kinds of loss functions were devised that used rollouts as training targets, taking inspiration from the work of Watters et al. (2017). They were the

n-step rollout loss and the *n*-step corrective rollout loss. With rollouts as targets, each episode of states is divided into multiple, overlapping training examples. Every training example consists of two input states (so that the velocity $\Delta \mathbf{s}_t$ can be calculated), the geometry and the *n* rollout states. The rollout states were used to calculate *n* partial targets. The MSE between the *n* predictions \mathbf{p}_k and the *n* partial targets were then computed and aggregated into a single loss. An episode with N_E states is thus divided into $N_E - 2 - n$ overlapping training examples, with each training example having *n* partial targets. This approach to targets and loss functions is different from that used by Pfaff et al., who only used a 1-step rollout loss.

Algorithm 3: *n*-step rollout

Input: $f, (\mathbf{s}_t, \Delta \mathbf{s}_t, \mathcal{G}(\mathbf{s}))$

Calculate current acceleration, $\mathbf{p}_0 = f(\mathbf{s}_t, \Delta \mathbf{s}_t, \mathcal{G}(\mathbf{s}))$;

Calculate future state, $\widehat{\mathbf{s}}_{t+1} = \mathbf{p}_0 + \mathbf{s}_t + \Delta \mathbf{s}_t$;

Calculate future velocity, $\widehat{\Delta \mathbf{s}}_{t+1} = \widehat{\mathbf{s}}_{t+1} - \mathbf{s}_t$;

for *k* from time 1 to and including *n*−1 **do**

 Calculate current acceleration, $\mathbf{p}_k = f(\widehat{\mathbf{s}}_{t+k}, \widehat{\Delta \mathbf{s}}_{t+k}, \mathcal{G}(\mathbf{s}))$;

 Calculate future state, $\widehat{\mathbf{s}}_{t+k+1} = \mathbf{p}_k + \widehat{\mathbf{s}}_{t+k} + \widehat{\Delta \mathbf{s}}_{t+k}$;

 Calculate future velocity, $\widehat{\Delta \mathbf{s}}_{t+k+1} = \widehat{\mathbf{s}}_{t+k+1} - \widehat{\mathbf{s}}_{t+k}$;

end

Output: $\{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_{n-1}\}, \{\widehat{\mathbf{s}}_{t+1}, \widehat{\mathbf{s}}_{t+2}, \dots, \widehat{\mathbf{s}}_{t+n}\}$

***n*-step rollout loss** The *n*-step rollout loss uses *n* subsequent ground truth accelerations as partial targets. A training input $(\mathbf{s}_t, \Delta \mathbf{s}_t, \mathcal{G}(\mathbf{s}))$ therefore has the target $\{\frac{1}{2}\ddot{\mathbf{s}}_t, \frac{1}{2}\ddot{\mathbf{s}}_{t+1}, \dots, \frac{1}{2}\ddot{\mathbf{s}}_{t+n-1}\}$, where each individual acceleration at the time τ is a partial target. These ground truth accelerations are calculated with

$$\frac{1}{2}\ddot{\mathbf{s}}_\tau = \mathbf{s}_{\tau+1} - \mathbf{s}_\tau - \Delta \mathbf{s}_\tau, \quad (3.3)$$

where the $\mathbf{s}_{\tau+1}$, \mathbf{s}_τ and $\Delta \mathbf{s}_\tau$ are given directly by the FDS simulation. For each training input, the model outputs a rollout of length *n*. The partial losses of this loss function is then the individual MSE losses between the *k*th prediction \mathbf{p}_k and *k*th partial target $\frac{1}{2}\ddot{\mathbf{s}}_{t+k}$. The loss function is a weighted sum of the partial losses, defined as

$$J_n(\boldsymbol{\theta}) = \frac{\sum_{k=0}^{n-1} w^k \left(\mathbf{p}_k - \frac{1}{2}\ddot{\mathbf{s}}_{t+k} \right)^2}{\sum_{k=0}^{n-1} w^k} \quad (3.4)$$

with a tunable hyperparameter *w* that impacts how much weight will be given to the partial loss at the *k*th rollout according to an exponential decay. The mean over all nodes *i* and over all features is also taken, so $J_n(\boldsymbol{\theta})$ is a scalar.

***n*-step corrective rollout loss** Instead of using the ground truth accelerations as partial targets for every partial loss, the *n*-step corrective rollout loss uses a

corrective acceleration \mathbf{a}_k as partial target for $k \geq 1$. This acceleration is defined as

$$\mathbf{a}_k = \mathbf{s}_{t+k+1} - \widehat{\mathbf{s}}_{t+k} - \widehat{\Delta \mathbf{s}}_{t+k}, \quad (3.5)$$

where $\widehat{\mathbf{s}}_{t+k}$ and $\widehat{\Delta \mathbf{s}}_{t+k}$ are predictions of the state and velocity obtained from the model itself during a rollout. The total loss function then becomes

$$J_{n, \text{corr}}(\boldsymbol{\theta}) = \frac{\left(\mathbf{p}_0 - \frac{1}{2}\ddot{\mathbf{s}}_t\right)^2 + \sum_{k=1}^{n-1} w^k (\mathbf{p}_k - \mathbf{a}_k)^2}{\sum_{k=0}^{n-1} w^k} \quad (3.6)$$

with the same weight w as the above loss. This is designed to incentivise the model to output accelerations that correct an erroneous current state $\widehat{\mathbf{s}}_{t+k}$ into a ground truth future state \mathbf{s}_{t+k+1} , as it was observed during preliminary test training that a model could have a low n -step rollout loss but still have compounding errors that made the states further into a long rollout diverge from ground truth. This is akin to forcing the model to learn de-noising if the errors introduced by the predictions are interpreted as noise.

3.2 Implementation Details

Many different combinations of GNN architecture hyperparameters, such as graph layer depth, MLP depth, MLP size were tested, as well as different optimisers and loss function hyperparameters. This section will detail some of the more successful combinations of hyperparameters, as well as how the training data were generated.

3.2.1 GNN Details

The graph neural network used in this work had 6 graph net layers in the processor, each of which had identical MLP designs f_l^E and f_l^V . They consisted of an input layer, two hidden layers and an output layer. Every layer had a ReLU activation function, including the output layers. At the end, the outputs got layer normalised. All layers had 128 nodes except for the input layer, which had 384 nodes for f_l^E and 256 nodes for f_l^V due to them concatenating several feature vectors at their inputs. They also had residual connections between the input and output activation but before the layer normalisation LN, so that $f(\mathbf{x}) = \text{LN}(\mathbf{x} + h(\mathbf{x}))$ for both f_l^E and f_l^V , and where h is the feedforward from input to output.

The two encoder MLPs ϵ^E and ϵ^V were also the same, except at the input layer which took the 3 features of an edge \mathbf{e}_{ij} or the 18 features of a node \mathbf{v}_i as input, respectively. The decoder δ took the output of f_6^V as input, had two hidden layers with 128 nodes each and then an output with size 8, corresponding to the features in the acceleration $\ddot{\mathbf{s}}$. While the other layers also used the ReLU activation, the output layer didn't use an activation function at all. No layer normalisation nor residual connection was used for δ either.

3.2.2 The FDS Datasets

FDS was used to generate training data by simulating many different episodes of fires, where an episode is the set of all the states in a whole simulation, $\{\mathbf{s}_t \mid t = 0, 1, 2, \dots\}$. The simulations created were all 20 seconds long and started from random initial conditions before any fire has started. The simulation domain used in this work was a cuboid with a width of 1 m along the x -axis, a height of 1 m along the z -axis and a depth of 0.1 m along the y -axis. The walls in the yz -plane and the ceiling were ventilated, allowing air to flow through them, while the floor and the xz -plane walls were solid. This is illustrated in figure 3.4, with the ventilated walls shaded grey. FDS calculates the dynamics in a 3D grid. The domains used in the simulations had a grid resolution that were equivalent to 3 cells along the y -axis and either 10 or 16 cells in the x and z axes, depending on which of the two dataset scenarios were considered.

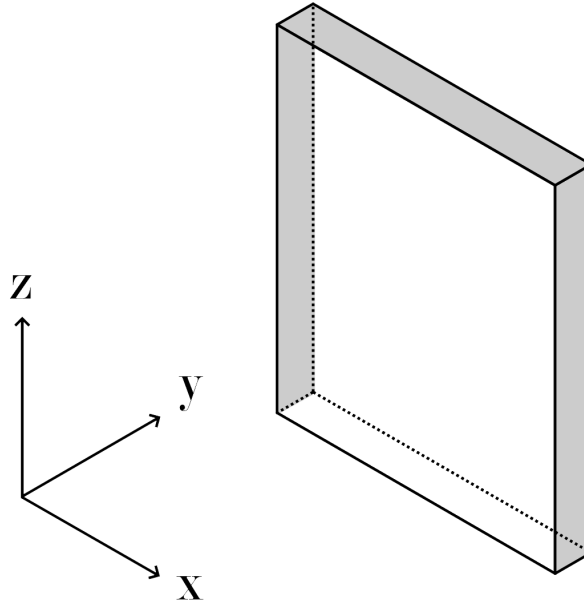


Figure 3.4: The simulation domain, not to scale. The shaded walls and ceiling are ventilated surfaces, allowing for air to pass through, while the floor and the two unshaded walls are solid.

Fire was generated by placing a rectangular burner at the bottom of the simulation domain. The burner’s top reacts with the oxygen in the air and burns, creating generalised soot particles in the process. Optionally, an obstruction could be placed in the domain in order to obstruct the path of the smoke. The initial conditions for every simulation episode is such that the fire hasn’t started in the initial frame. Figure 3.5 shows an example frame from an episode, visualised with Smokeview.

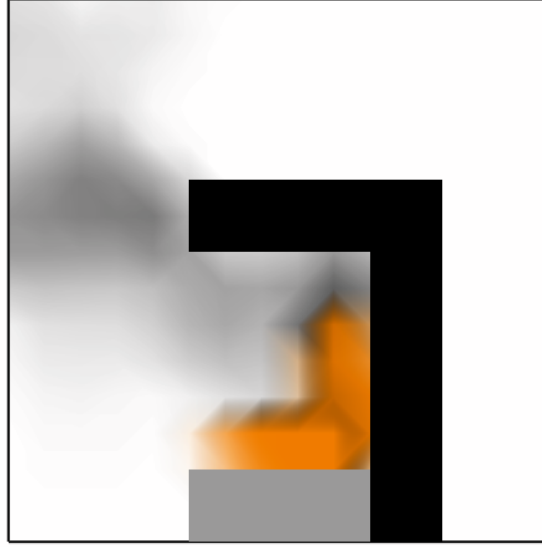


Figure 3.5: The front view of a single frame from an FDS simulation visualised with Smokeview. The grey rectangle is the burner and the solid black regions are obstructions through which gas can't flow. The grey smoke is the soot and the fire visualisation is calculated from the heat reaction rate per unit volume.

Out of the three layers of cells along the y -axis, the center layer was chosen as the 2D mesh of our system. That is, the state \mathbf{s}_t was a slice of all the features in the centremost xz -plane. The feature vectors \mathbf{q}_t^i in the mesh contained the mean temperature, HRRPUV (heat reaction rate per unit volume), oxygen density, oxygen velocity along x and z , soot density, and soot velocity along x and z , of cell i at time t . A sample of a state \mathbf{s}_t is shown in figure 3.6. Since the internal framerate in FDS is variable, each episode was interpolated to a uniform 3 frames per second, resulting in 61 states per episode.

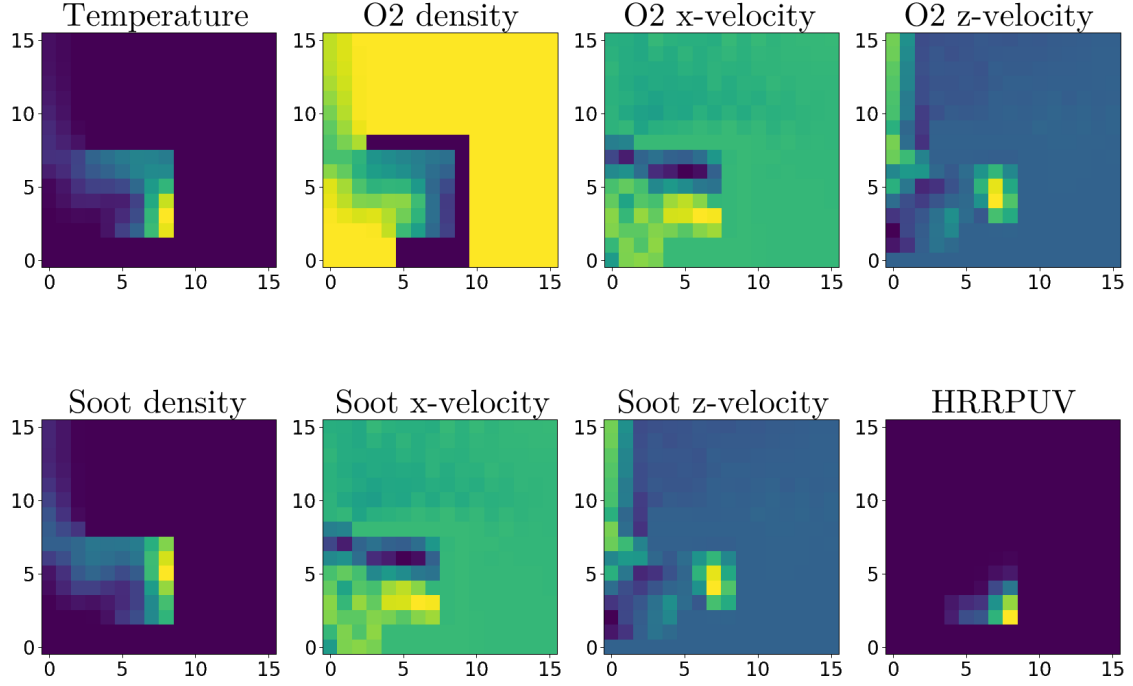


Figure 3.6: The 2D mesh representation of the system, visualised as a set of heatmaps, each belonging to a feature. A pixel in a heatmap represents the cell $i \in V$, and the value of the pixel represents the quantity of the feature at the location of the cell. The values corresponding to pixel i in all the heatmaps together form the feature vector \mathbf{q}_t^i . A state \mathbf{s}_t is obtained by considering every pixel in the heatmaps. The state shown in this example is similar but not identical to the state in figure 3.5.

Two datasets were created from different scenarios: a simple, low resolution dataset called SmallSet and a slightly more complex, higher resolution scenario called BigSet.

SmallSet The simple SmallSet dataset had a 10×10 grid. A burner was placed in the middle of the floor of the domain. No obstructions were used. The only variation in the initial conditions between different episodes was the width of the burner. The variation in width was smaller than the output grid resolution, meaning that the initial conditions looked nearly identical. 2900 episodes were used in the training set and 100 episodes were used in the validation set.

BigSet BigSet had a finer 16×16 grid, this time with obstructions. The obstructions were as shown in figure 3.5, wherein a wall was connected to either the right side or the left side of the burner, with a roof attached at the top on the same side as the burner. The distributions of the burner’s width, the coordinate of the burner’s centre, the wall’s height and the roof’s length are shown in figure 3.7. The burner had to be at least 0.2 m from the edge of the domain, while the roof had to be at least 0.125 m from the edge. The minimum gap between the burner and the ceiling was 0.25 m. The thicknesses of the burner, the wall and the roof were all 0.125 m.

The wall could be on either the left or the right side with equal probability. As in the previous scenario, 2900 episodes were used in the training set and 100 episodes were used in the validation set.

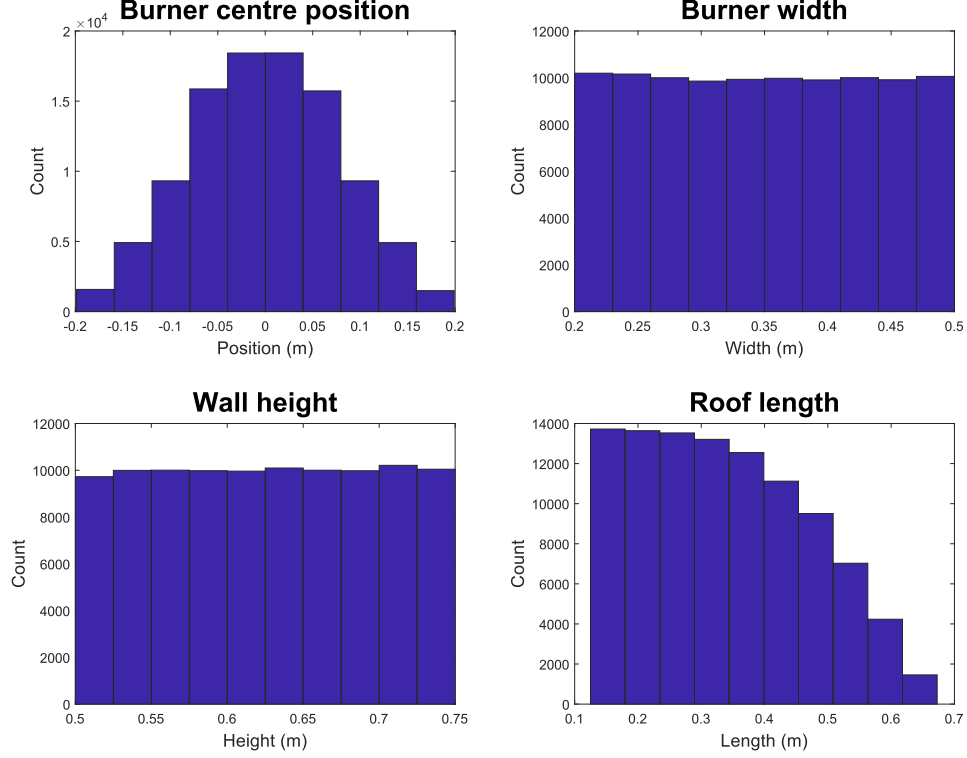


Figure 3.7: The distribution of the position of the burner’s centre, the burner’s width, the wall’s height and the roof’s length from 100 000 samples.

The datasets were standardised, meaning that the input states \mathbf{s}_t and velocities $\Delta \mathbf{s}_t$ of all the training examples had zero mean and unit variance in a per-feature basis (with feature referring to the distinct features in a node; SmallSet and BigSet had the same amount of features even though one was a 10×10 mesh and the other was 16×16). This was obtained by subtracting the mean and then dividing by the standard deviation of every feature in the training sets. The two different datasets were standardised independently. The geometry input was, however, not standardised. Similarly, the partial targets—both $\frac{1}{2}\ddot{\mathbf{s}}_{t+k}$ and \mathbf{a}_k alike—were standardised by subtracting the mean and dividing by the standard deviation of the first step rollout accelerations $\frac{1}{2}\ddot{\mathbf{s}}_t$ in their respective training set. Using the GNN to predict anything thus requires the input and the output to be standardised and unstandardised according to the training set.

3.2.3 Training

Multiple MeshGraphNets models with a GNN architecture as described earlier were independently trained on the two FDS training sets in order to test different hyper-

parameter configurations. They all used the AdamW optimiser, but with varying learning rates that decayed for every training epoch by multiplying with the hyperparameter $\gamma < 1$. Some of the models trained on the simple FDS data didn't use the geometry features.

Different models were trained with the 1-step rollout loss, the 8-step rollout loss and the 8-step corrective rollout loss. For the 8-step rollout cases, the hyperparameter w was increased over the training epochs with

$$w = c(1 - \exp(-hn)), \quad (3.7)$$

where n is the n th training epoch ($n = 1, 2, \dots$), c is a constant determining the maximum value of w and h is a constant determining the speed at which w converges to c .

The trainings lasted for different number of training epochs. With few exceptions, the models were trained for a number of epochs that was equivalent to training on approximately $\frac{1\,000\,000}{\text{rollout length}}$ episodes worth of partial targets in order to have a comparable training speed. Since each episode has 61 frames, the training conducted with the 1-step rollout loss had in total 59 partial targets per episode, whereas the 8-step rollout trainings had $52 \times 8 = 416$ partial targets per episode, due to overlapping training examples in different rollouts. With a training set consisting of 2900 episodes, the two cases were equivalent to training for 344 and 43 epochs, respectively, when using the aforementioned criterion.

Training was performed partly on a GTX 1080 Ti with 11 GB memory and partly on a notebook GTX 1070 with 8 GB memory. The batch size for the minibatches were chosen to be 64 or however many training examples could fit into the GPU memory, whichever was smallest. The smallest batch size used was 16 for the 8-step corrective rollout loss on BigSet, since the implementation required the whole rollout to be stored in memory during training.

3.3 Metrics for Model Evaluation

The models were trained on the accelerations in either 1-step rollouts or 8-step rollouts. Since we want to predict the states of the system, it is necessary to consider the MSE of the states instead of the accelerations. Also, because some models were trained on 8-step rollouts, it will be of use to measure the MSE between the predicted states and the ground truth states for every step in an 8-step rollout in the validation dataset. The step-wise MSEs were averaged over the $100 \times 51 = 5100$ 8-step rollouts in the validation set, with 51 being the number of examples per episode.

Long term prediction is also a point of interest. It's therefore necessary to also consider rollouts that are as long as whole episodes. One way of gauging this is to simply look at the MSE between the predicted states and the ground truth states, with the mean taken over every frame of every episode, in what we'll call the episode

MSE. A caveat to this measure is that chaotic physical systems will deviate over time even for small initial differences. Thus, even for good models, it is possible that small errors in the predictions will compound in a longer rollout and result in states that are physically plausible yet very different from the ground truth.

Quantifying the plausibility of a predicted state isn't straightforward. A naive way of doing so is to, in a whole episode rollout, compare the predicted state in some particular episode n_e at time t with the ground truth states in every episode at time t . Specifically, the features averaged over all the cells $\hat{\mathbf{q}}_{t,\text{mean}}^{n_e}$ of a predicted state $\hat{\mathbf{s}}_t$ is compared to the cell and episode mean of the ground truth features $\mathbf{q}_{t,\text{mean}}$. These two quantities are calculated through

$$\hat{\mathbf{q}}_{t,\text{mean}}^{n_e} = \frac{1}{|V|} \sum_i \hat{\mathbf{q}}_t^{i,n_e} \quad (3.8)$$

and

$$\mathbf{q}_{t,\text{mean}} = \frac{1}{N_E|V|} \sum_{n_e=1}^{N_E} \sum_i \mathbf{q}_t^{i,n_e}, \quad (3.9)$$

where $\hat{\mathbf{q}}_t^{i,n_e}$ is the predicted feature vector at cell i in time t of episode n_e and $|V|$ is the total number of cells. \mathbf{q}_t^{i,n_e} is the ground truth feature vector and N_E is the total number of episodes. As can be seen in the equations, $\mathbf{q}_{t,\text{mean}}$ is the average over all episodes as well as all cells, while $\hat{\mathbf{q}}_{t,\text{mean}}^{n_e}$ is just the average over all cells in a single episode n_e . These two quantities are compared using MSE over all episodes, where we define \mathbf{Q}^t as

$$\mathbf{Q}^t = \frac{1}{N_E} \sum_{n_e=1}^{N_E} \left(\hat{\mathbf{q}}_{t,\text{mean}}^{n_e} - \mathbf{q}_{t,\text{mean}} \right)^2. \quad (3.10)$$

The elements in \mathbf{Q}^t are the feature-wise MSEs, so the mean of all the elements ξ_t is computed in order to obtain a single measure. This is done with

$$\xi_t = \frac{1}{|\mathbf{Q}^t|} \sum_j Q_j^t, \quad (3.11)$$

with Q_j^t being the j th feature in \mathbf{Q}^t . The features are standardised, i.e. with zero mean and unit variance, so that averaging over the different features is meaningful. The difference between calculating ξ and calculating the state MSE as usual is that ξ is a statistic that measures the predictions' deviation from the dataset's mean. It can be called a measure of plausibility since plausible predictions should be close to the dataset's mean. However, it is a naive measure since implausible predictions can also result in small ξ . An example is a network that always outputs the mean of the ground truths.

A final, and easy, way of evaluating the models is through visual inspection. For a physical system that is somewhat intuitive like rising smoke, it is easy to tell whether a series of predicted states is plausible. The downside is that complex patterns, such as turbulence, can be hard to get an intuition for, and the manual labour can be tedious. It could function as an indicator as to which models should be tested more, however.

3.4 Baseline Comparison with UNet

The UNet architecture as implemented by Thuerey et al. (2020) was also tested as a comparison. UNet is a deep convolutional network that utilises many recurrent connections between its layers. The specific architecture used expected a 256×256 sized input and output, so the input and output meshes were upsampled and downsampled bicubically, respectively. Previous MeshGraphNets results showed that UNet performed worse (Pfaff et al., 2021), but it might not necessarily be the case with the datasets used in this work.

3.5 Test Cases for Generalisation

Three test sets were created to test the generalising capabilities of any successful models trained on BigSet. The three test sets each contain 100 episodes with 61 states. They are a set with a taller simulation domain called TallSet, a set with taller roofs as well as a taller simulation domain called TallRoofSet, and finally a set with only a burner called BurnerSet. The tall sets had grid resolutions of 32×16 while BurnerSet used the same 16×16 resolution.

4. Results

This chapter details the results from training the neural networks. Also described is a bug discovered regarding the computation of the systems' geometries.

4.1 SmallSet

Several GNN models with and without geometry features were trained on SmallSet, with different training hyperparameters. Table 4.1 lists them along with their MSEs for their first predicted states in an 8-step rollout, the MSEs for their eight predicted states in an 8-step rollout as well as their mean state MSE for rollouts lasting for a whole episode. The whole episode rollouts used only the first two frames of an episode as input, while the 8-step rollouts could use any two consecutive frames as input (except the last 8 frames). Figure 4.1 shows several performance measures over the course of an episode rollout for the models listed in table 4.1. The temperature, oxygen density and soot density from a single episode rollout for some of these models are shown in figure 4.2, figure 4.3 and figure 4.4, respectively.

Table 4.1: The performances of different models with different training hyperparameters trained on SmallSet. The performance measured are the 1st predicted state MSE in an 8-step rollout, the 8th predicted state MSE in an 8-step rollout and the mean state MSE for a whole episode rollout. The predicted states were compared with the ground truth states. All the predictions are made on the validation set.

Name	S1	S8	S1g	S8g	S8gc
Neural network	GNN	GNN	GNN	GNN	GNN
Geometry features	No	No	Yes	Yes	Yes
Rollout steps	1	8	1	8	8
Corrective loss	-	No	-	No	Yes
Epochs trained	344	43	344	43	43
GPU	1070	1080 Ti	1070	1080 Ti	1080 Ti
Time to train	≈ 53 h	≈ 26 h	≈ 44 h	≈ 26 h	≈ 26 h
γ	$e^{\frac{\ln 0.3}{344}}$	$e^{\frac{\ln 0.3}{43}}$	$e^{\frac{\ln 0.3}{344}}$	$e^{\frac{\ln 0.3}{43}}$	$e^{\frac{\ln 0.3}{43}}$
c	-	0.5	-	0.5	0.5
h	-	5/43	-	5/43	5/8
1st state MSE	0.43	0.51	0.43	0.51	0.50
8th state MSE	1.0	0.37	0.79	0.44	2.6
Episode MSE	1.1	29	1.1	27.0	21

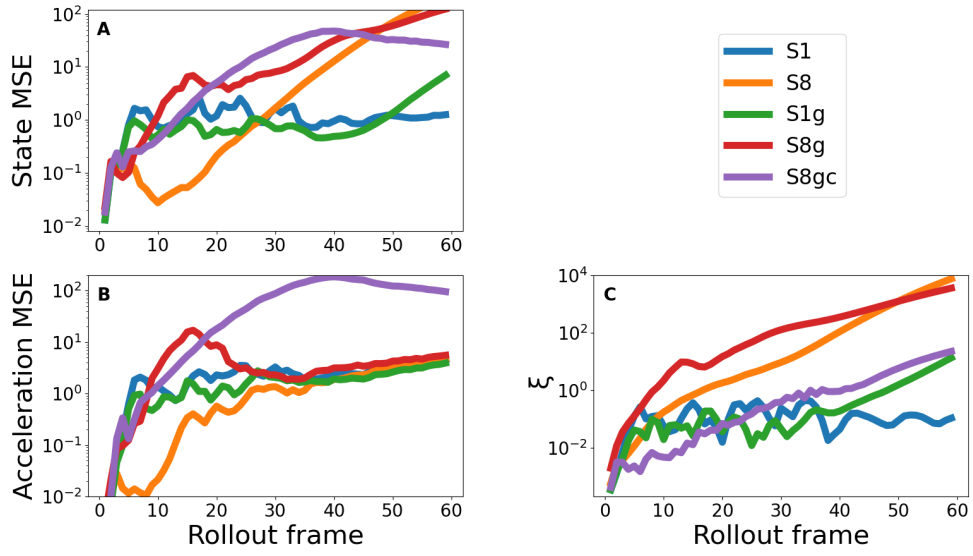
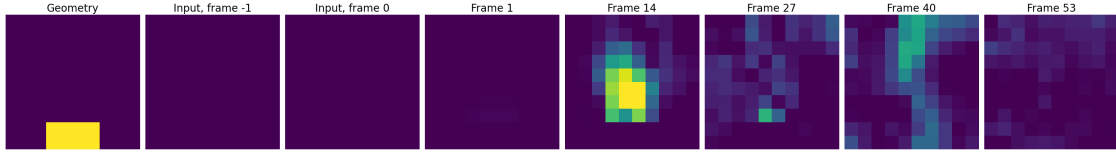
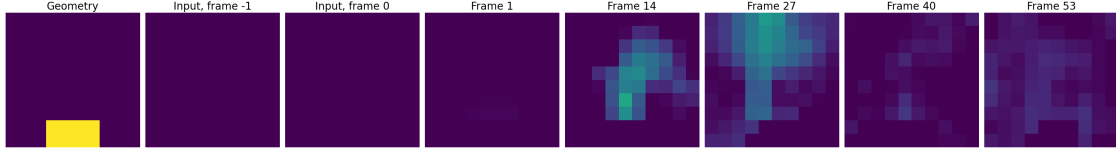


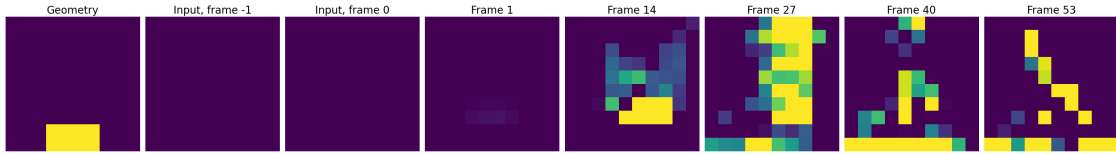
Figure 4.1: Three different measures of how well the models performed on the validation set of SmallSet. The quantities are the episodic averages, plotted over whole episode rollouts. Plot **A** and **B** shows the state and acceleration MSEs, and are calculated with the respective ground truth quantities. **C** illustrates how ξ , as defined in equation (3.11), varies over an episode rollout.



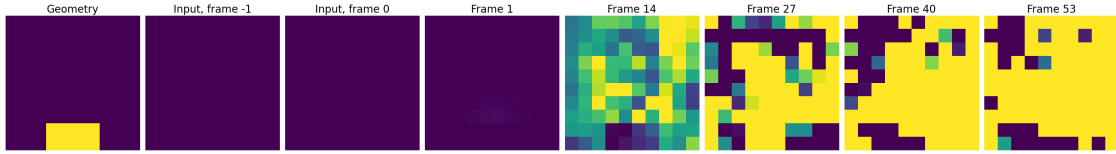
(a) Temperatures in a sample episode rollout for S1.



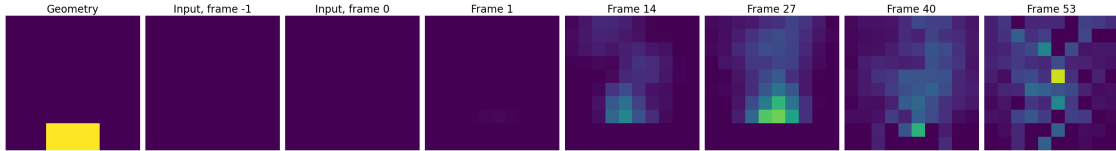
(b) Temperatures in a sample episode rollout for S1g.



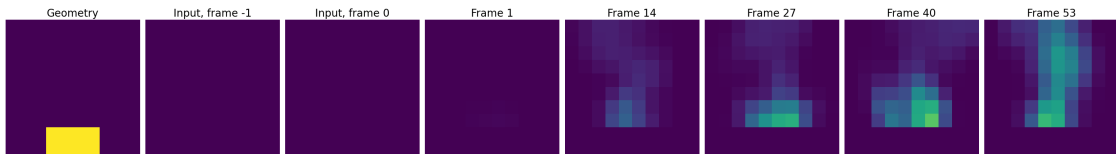
(c) Temperatures in a sample episode rollout for S8.



(d) Temperatures in a sample episode rollout for S8g.

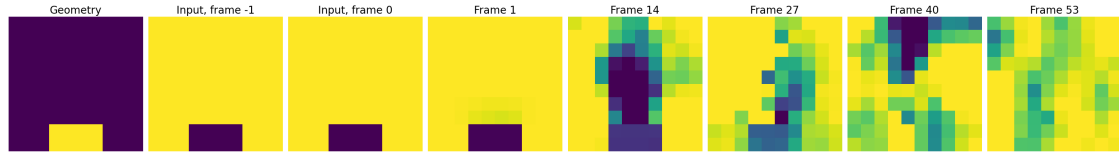


(e) Temperatures in a sample episode rollout for S8gc.

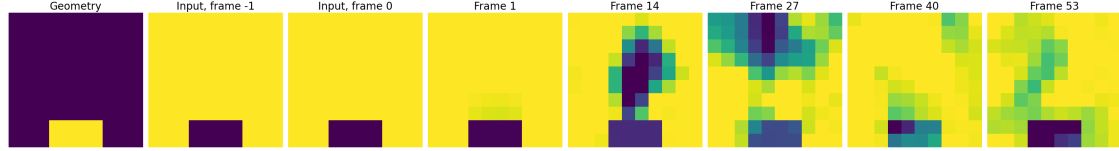


(f) The corresponding ground truth temperatures.

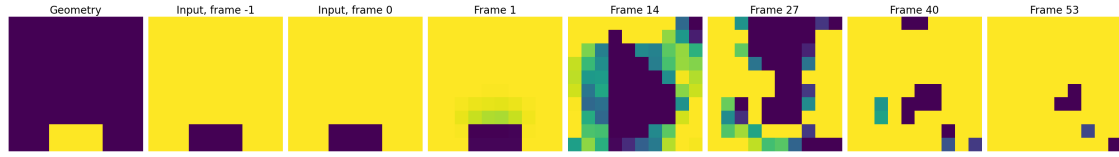
Figure 4.2: Temperatures in a whole episode rollout for different models, together with the ground truth. The only ground truth used as input for the predictions are the first two frames. The heatmap ranges are normalised after the ground truth rollout.



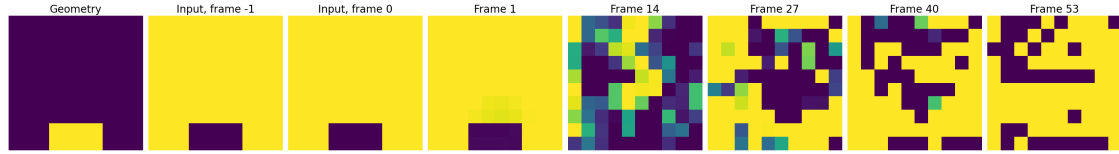
(a) Oxygen densities in a sample episode rollout for S1.



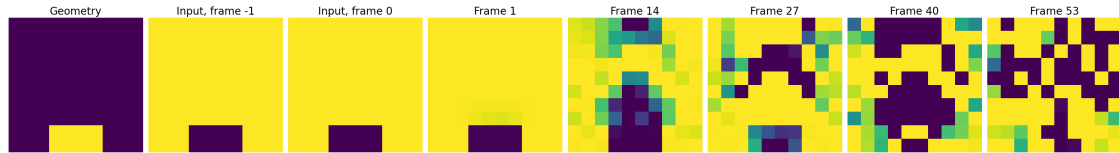
(b) Oxygen densities in a sample episode rollout for S1g.



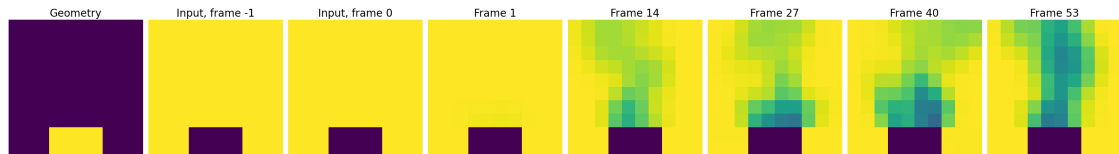
(c) Oxygen densities in a sample episode rollout for S8.



(d) Oxygen densities in a sample episode rollout for S8g.

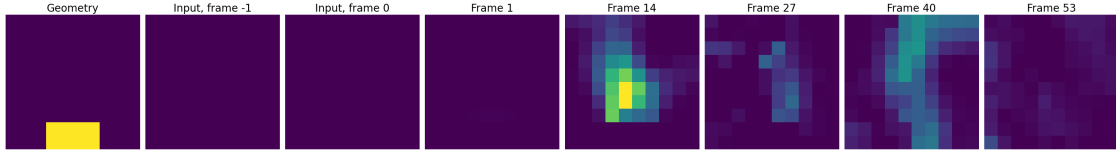


(e) Oxygen densities in a sample episode rollout for S8gc.

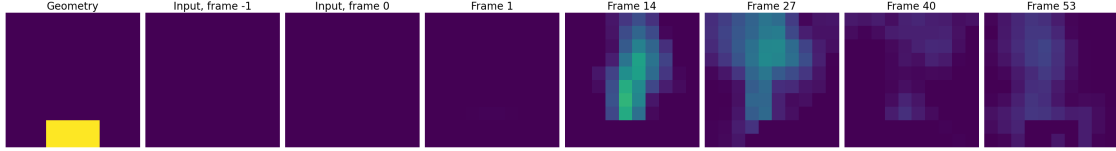


(f) The corresponding ground truth oxygen densities.

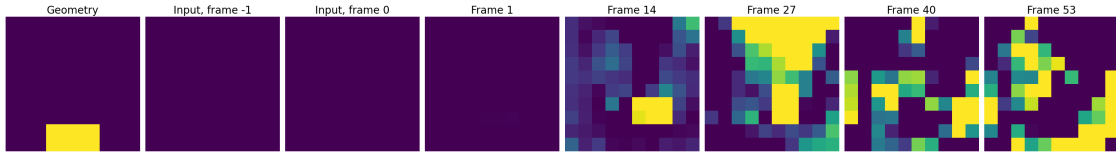
Figure 4.3: Oxygen densities in a whole episode rollout for different models, together with the ground truth. The only ground truth used as input for the predictions are the first two frames. The heatmap ranges are normalised after the ground truth rollout.



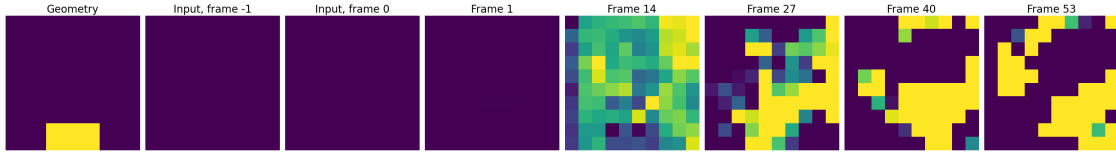
(a) Soot densities in a sample episode rollout for S1.



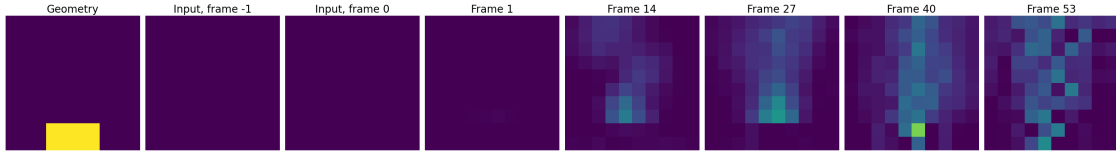
(b) Soot densities in a sample episode rollout for S1g.



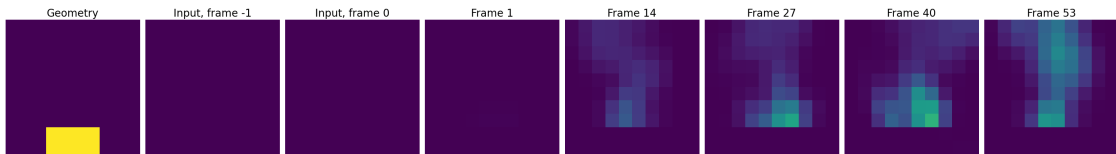
(c) Soot densities in a sample episode rollout for S8.



(d) Soot densities in a sample episode rollout for S8g.



(e) Soot densities in a sample episode rollout for S8gc.



(f) The corresponding ground truth soot densities.

Figure 4.4: Soot densities in a whole episode rollout for different models, together with the ground truth. The only ground truth used as input for the predictions are the first two frames. The heatmap ranges are normalised after the ground truth rollout.

4.2 BigSet

Several GNN and UNet models were trained on the complex dataset, with different training hyperparameters. Table 4.2 lists them along with their MSEs for their first predicted states in an 8-step rollout, the MSEs for their eight predicted states in an 8-step rollout as well as their mean state MSE for rollouts lasting for a whole episode. Figure 4.5 shows several performance measures over the course of an episode rollout for the models listed in table 4.2. The temperature, oxygen density and soot density from a single episode rollout for some of these models are shown in figure 4.6, figure 4.7 and figure 4.8, respectively.

Table 4.2: The performances of different models with different training hyperparameters trained on BigSet. The performance measured are the 1st predicted state MSE in an 8-step rollout, the 8th predicted state MSE in an 8-step rollout and the mean state MSE for a whole episode rollout. The predicted states were compared with the ground truth states. All the predictions are made on the validation set.

Name	C1	C8Long	C8Short	C8Flat	U8Long	U8Short
Neural network	GNN	GNN	GNN	GNN	UNet	UNet
Geometry features	Yes	Yes	Yes	Yes	Yes	Yes
Rollout steps	1	8	8	8	8	8
Corrective loss	-	Yes	Yes	Yes	Yes	Yes
Epochs trained	344	43	8	43	43	8
GPU	1070	1080 Ti	1080 Ti	1080 Ti	1080 Ti	1080 Ti
Time to train	≈ 138 h	≈ 90 h	≈ 17 h	≈ 90 h	≈ 41 h	≈ 8 h
γ	$e^{-\frac{\ln 0.3}{344}}$	$e^{-\frac{\ln 0.3}{43}}$	$e^{-\frac{\ln 0.3}{8}}$	$e^{-\frac{\ln 0.3}{43}}$	$e^{-\frac{\ln 0.3}{43}}$	$e^{-\frac{\ln 0.3}{8}}$
c	-	0.5	0.5	0.5	0.5	0.5
h	-	5/43	5/8	∞	5/43	5/8
1st state MSE	0.27	0.28	0.25	0.27	0.32	0.25
8th state MSE	0.76	0.11	0.078	0.081	39000	0.078
Episode MSE	0.91	2.1	0.044	0.041	∞	0.16

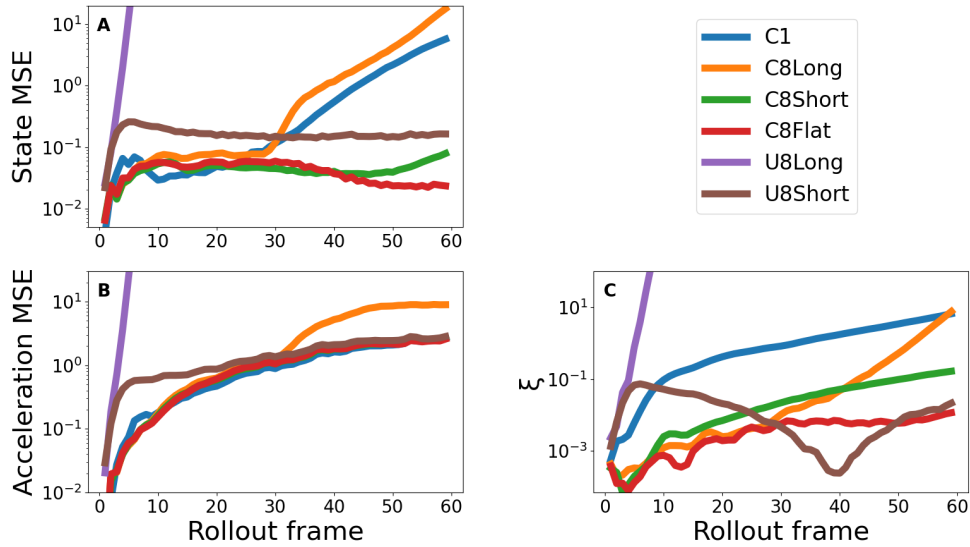
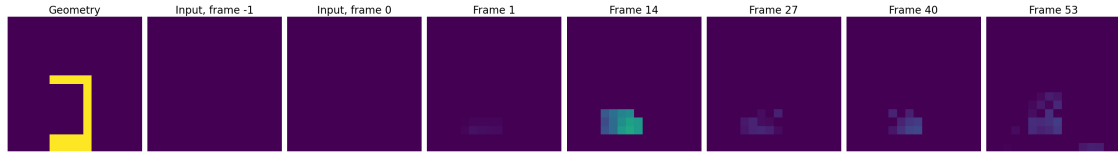
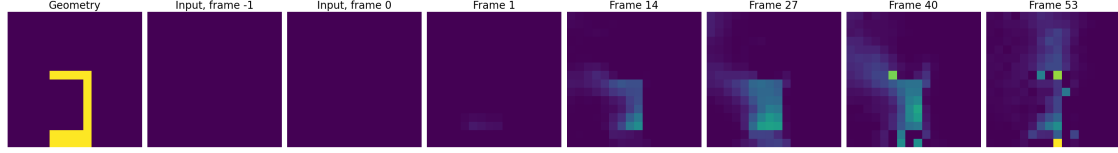


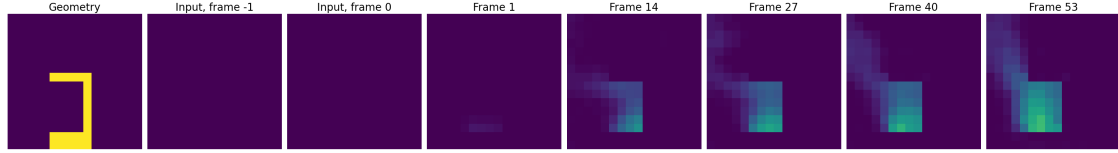
Figure 4.5: Three different measures of how well the models performed on the validation set of BigSet. The quantities are the episodic averages, plotted over whole episode rollouts. Plot **A** and **B** shows the state and acceleration MSEs, and are calculated with the respective ground truth quantities. **C** illustrates how ξ , as defined in equation (3.11), varies over an episode rollout.



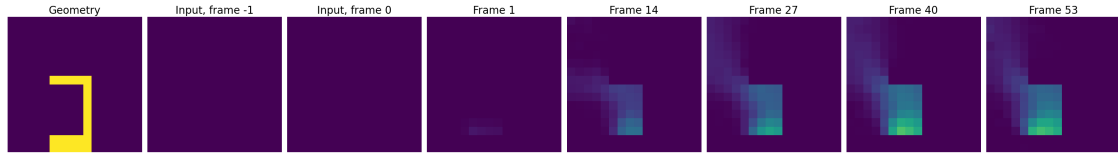
(a) Temperatures in a sample episode rollout for C1.



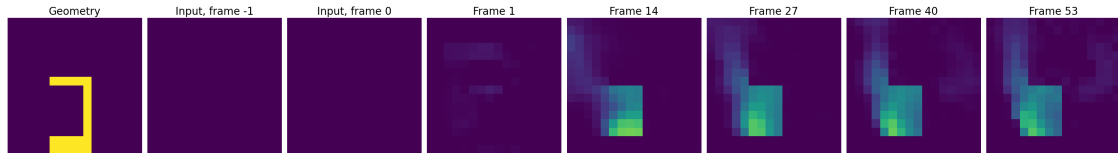
(b) Temperatures in a sample episode rollout for C8Long.



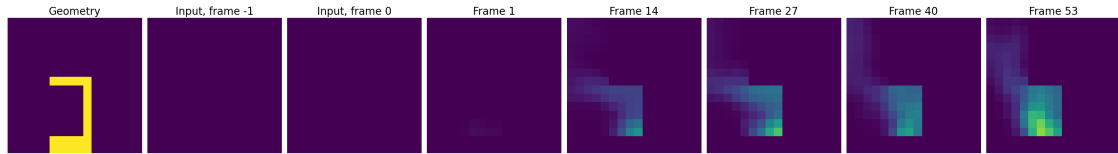
(c) Temperatures in a sample episode rollout for C8Short.



(d) Temperatures in a sample episode rollout for C8Flat.

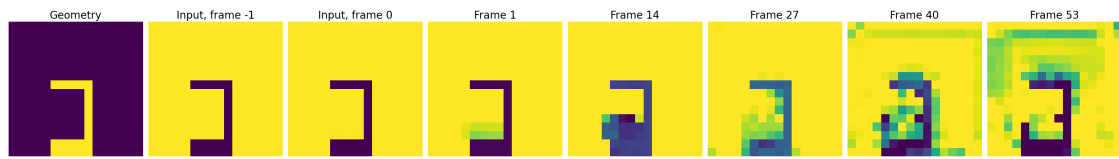


(e) Temperatures in a sample episode rollout for U8Short.

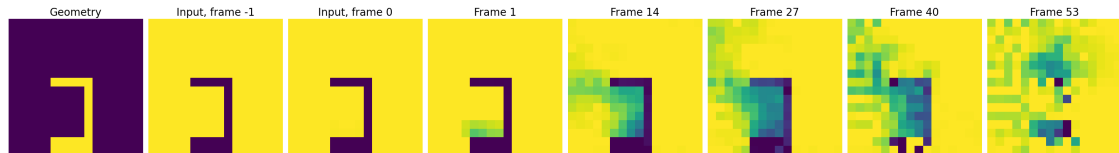


(f) The corresponding ground truth temperatures.

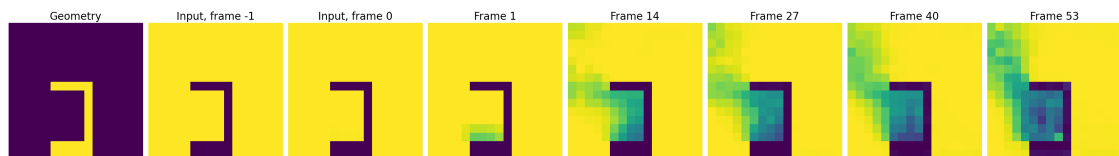
Figure 4.6: Temperatures in a whole episode rollout for different models, together with the ground truth. The only ground truth used as input for the predictions are the first two frames. The heatmap ranges are normalised after the ground truth rollout.



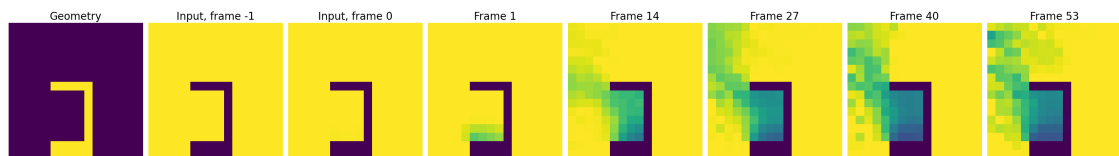
(a) Oxygen densities in a sample episode rollout for C1.



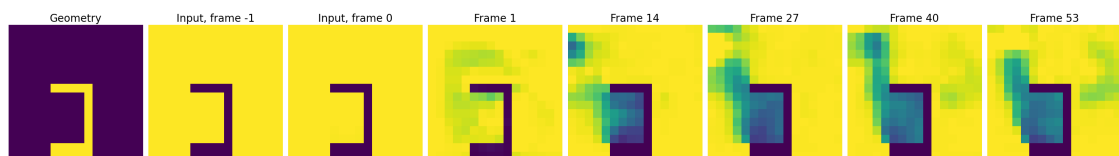
(b) Oxygen densities in a sample episode rollout for C8Long.



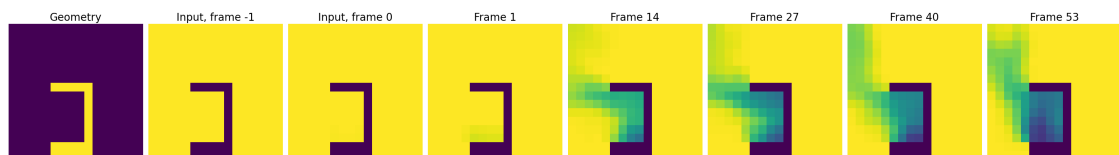
(c) Oxygen densities in a sample episode rollout for C8Short.



(d) Oxygen densities in a sample episode rollout for C8Flat.

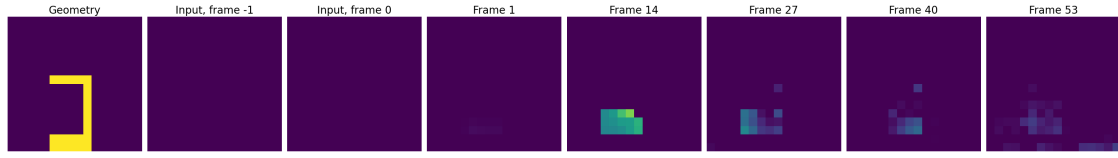


(e) Oxygen densities in a sample episode rollout for U8Short.

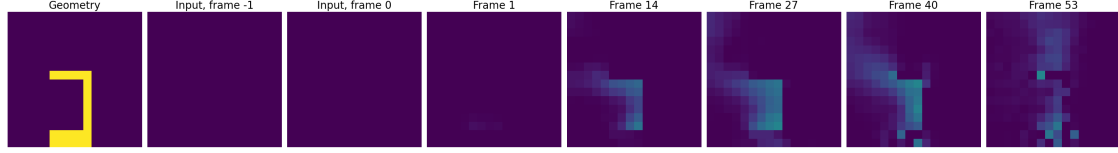


(f) The corresponding ground truth oxygen densities.

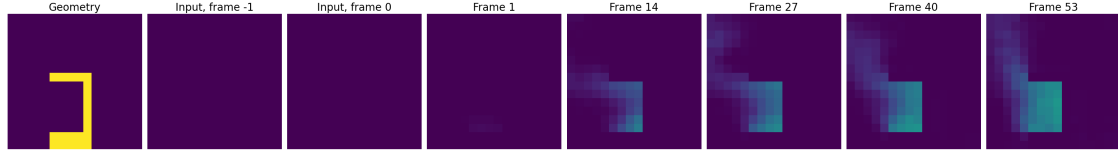
Figure 4.7: Oxygen densities in a whole episode rollout for different models, together with the ground truth. The only ground truth used as input for the predictions are the first two frames. The heatmap ranges are normalised after the ground truth rollout.



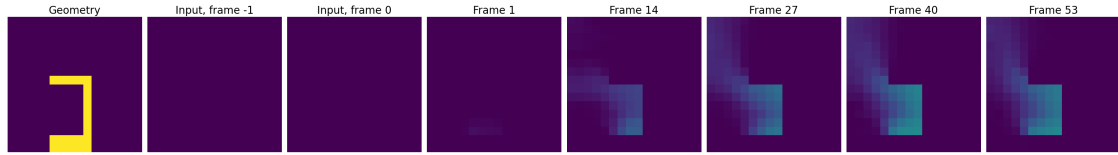
(a) Soot densities in a sample episode rollout for C1.



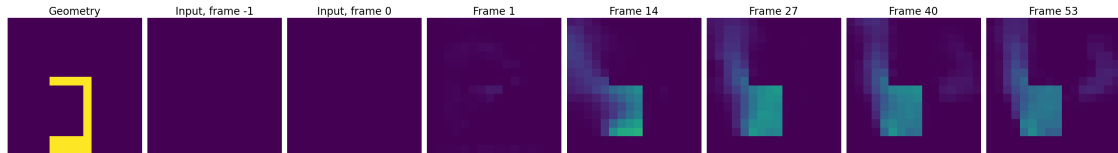
(b) Soot densities in a sample episode rollout for C8Long.



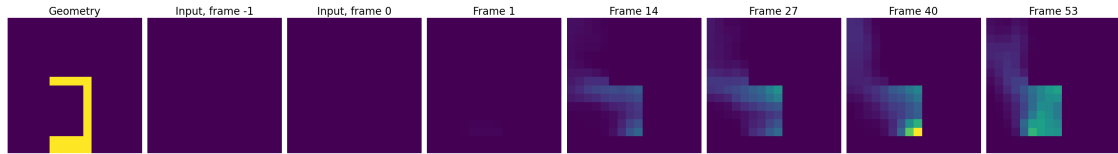
(c) Soot densities in a sample episode rollout for C8Short.



(d) Soot densities in a sample episode rollout for C8Flat.



(e) Soot densities in a sample episode rollout for U8Short.



(f) The corresponding ground truth soot densities.

Figure 4.8: Soot densities in a whole episode rollout for different models, together with the ground truth. The only ground truth used as input for the predictions are the first two frames. The heatmap ranges are normalised after the ground truth rollout.

4.3 Domain Generalisation

C8Short, which was trained on BigSet, was tested on the generalising datasets. Table 4.3 lists the 1st & 8th state MSE and the episode MSE on the different datasets. The model’s performance over the course of an episodic rollout is shown in figure 4.9. Rollouts of an episode from the set with a taller domain, with a taller domain and roof, and without roofs and walls are illustrated in figure 4.10, 4.11 and 4.12, respectively.

Table 4.3: The 1st and 8th state MSE in an 8-step rollout, as well as the whole episode MSE for an episodic rollout, calculated for the C8Short model. C8Short was trained on BigSet. The datasets used were the validation set of BigSet, TallSet, TallRoofSet, and finally BurnerSet.

	1st state MSE	8th state MSE	Episode MSE
BigSet	0.25	0.078	0.044
TallSet	0.31	0.11	0.036
TallRoofSet	0.31	0.088	0.035
BurnerSet	0.45	0.068	0.025

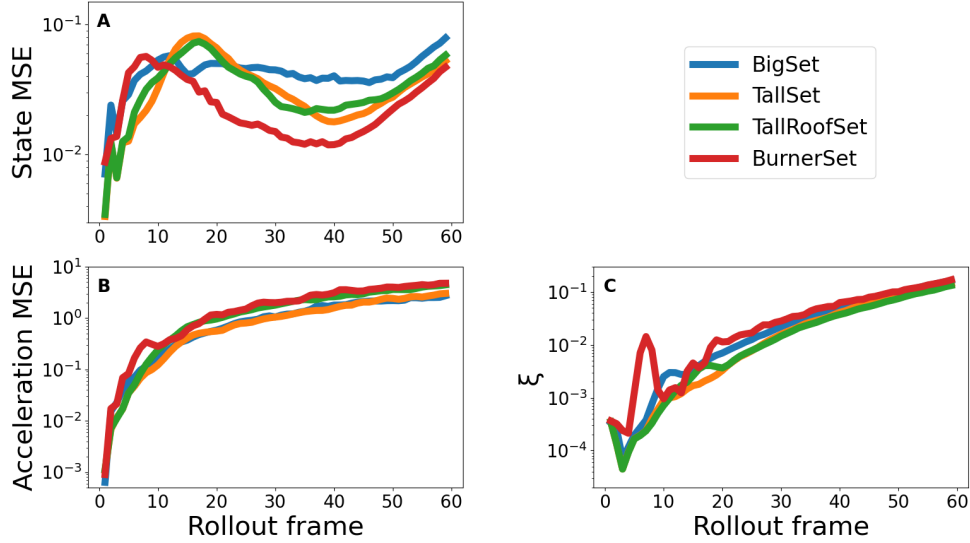
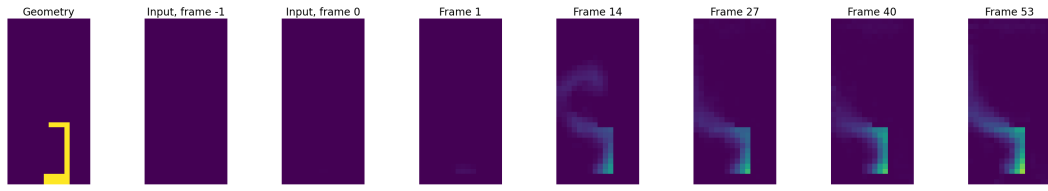
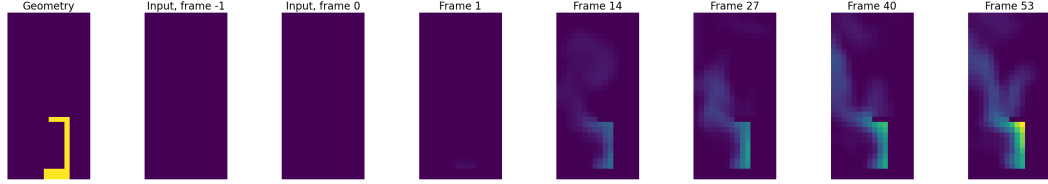


Figure 4.9: Three different measures of how well the C8Short performed on BigSet as well as the additional test sets. The quantities are the episodic averages, plotted over whole episode rollouts. Plot **A** and **B** shows the state and acceleration MSEs, and are calculated with the respective ground truth quantities. **C** illustrates how ξ , as defined in equation (3.11), varies over an episode rollout.

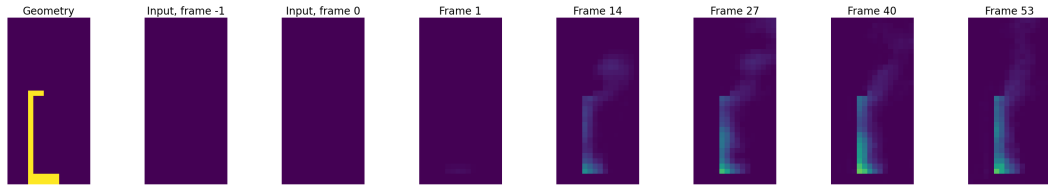


(a) Sample rollout soot density as predicted by C8Short.

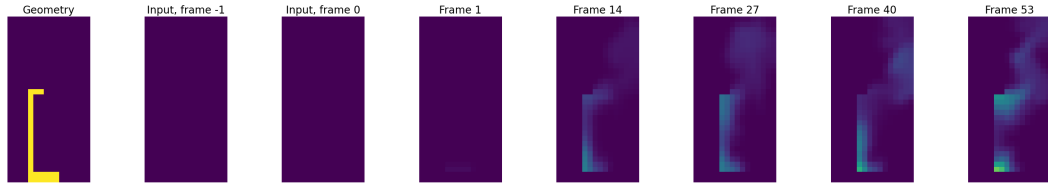


(b) Sample ground truth soot density of the predictions in (a).

Figure 4.10: Sample soot density rollouts from TallSet. The rollout spans a whole episode.

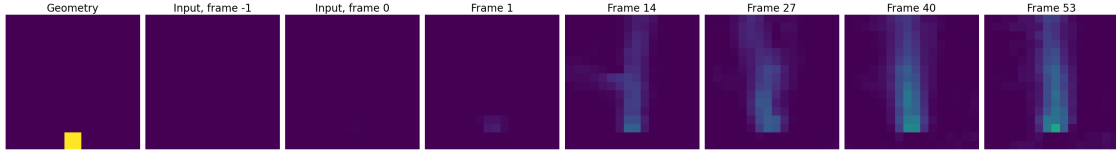


(a) Sample rollout soot density as predicted by C8Short.

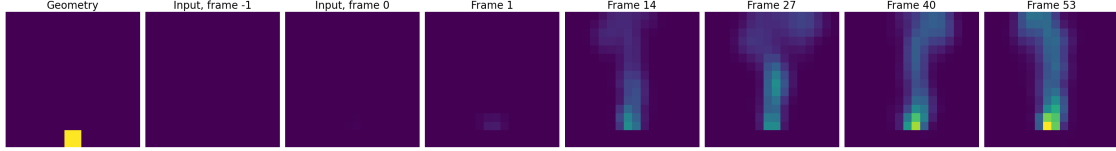


(b) Sample ground truth soot density of the predictions in (a).

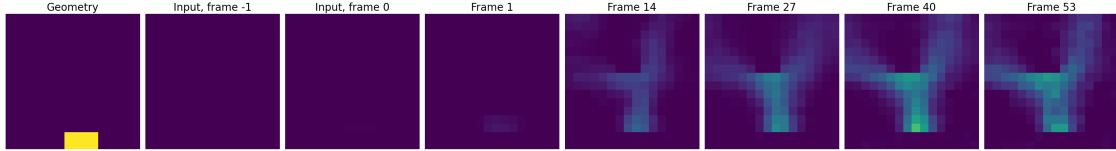
Figure 4.11: Sample soot density rollouts from TallRoofSet. The rollout spans a whole episode.



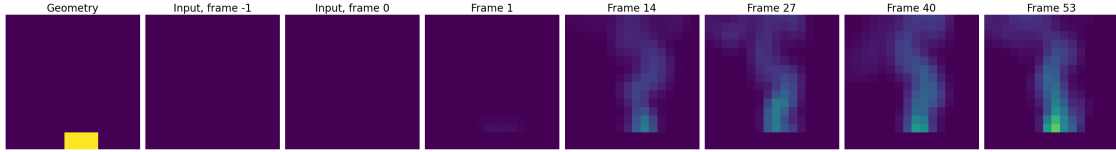
(a) Sample rollout soot density as predicted by C8Short.



(b) Sample ground truth soot density of the predictions in (a).



(c) Sample rollout soot density as predicted by C8Short. The burner size and placement is different from (a).



(d) Sample ground truth soot density of the predictions in (c).

Figure 4.12: Sample soot density rollouts from BurnerSet. The rollout spans a whole episode. (a) and (c) shows the prediction of two different episodes while (b) and (d) are their respective ground truths.

4.4 GNN Performance

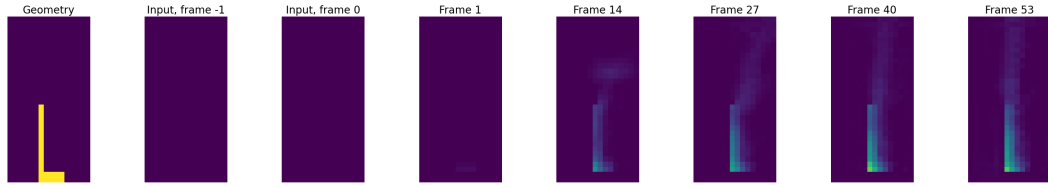
The time taken to generate 100 episodes of rollouts with the graph neural networks was compared to the time it took to generate the FDS simulations. The results are shown in table 4.4.

Table 4.4: The time it took for different hardware to compute 100 episode rollouts on either SmallSet or BigSet with a GNN, and the time it took for a Ryzen 5 5600X CPU to simulate a single episode with FDS. The 100 GNN rollouts were calculated in parallel.

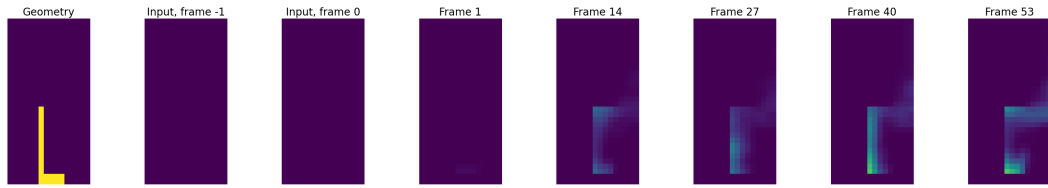
	R5 5600X (CPU)	1080 Ti (GPU)	1070 Notebook (GPU)
GNN, SmallSet	65 s	3.6 s	8.7 s
GNN, BigSet	190 s	9.0 s	18 s
FDS, SmallSet	≈ 13 s	-	-
FDS, BigSet	≈ 12 s	-	-

4.5 Incorrect Data

During testing, some of the data were found to have incorrect geometry \mathcal{G} , where either a roof, a wall or possibly both were missing. \mathcal{G} is calculated in the post-processing of the simulations, which is where the error arose. This means that the ground truth data weren't affected, but the inputs to the networks were wrong. The frequency of these errors is unknown. Two examples of this are shown in figure 4.13 and 4.14.

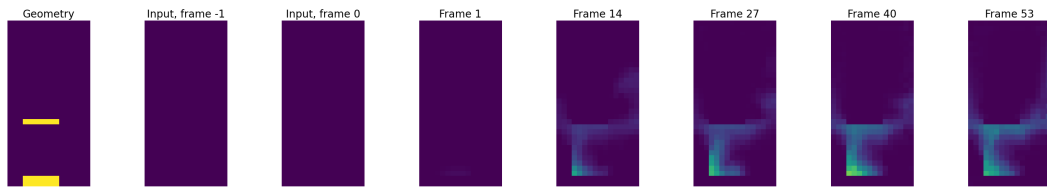


(a) The soot density rollout as predicted by C8Short, with an incorrect, roof-less geometry input.

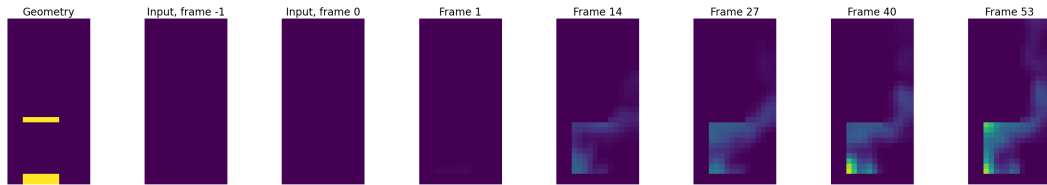


(b) The ground truth soot density corresponding to the predictions in (a). Note that the geometry feature is incorrect.

Figure 4.13: Sample soot density rollout from TallRoofSet, where the geometry has been incorrectly calculated to not have a roof.



(a) The soot density rollout as predicted by C8Short, with an incorrect, wall-less geometry input.



(b) The ground truth soot density corresponding to the predictions in (a). Note that the geometry feature is incorrect.

Figure 4.14: Sample soot density rollout from TallRoofSet, where the geometry has been incorrectly calculated to not have a wall.

5. Discussion

The S8gc model, which was the model that trained with an 8 corrective rollout loss on SmallSet, was the best performing model trained on SmallSet according to visual inspection. Figure 4.2 shows that S8gc predicted very reasonable temperatures up to frame 14, while no other model did well. The 27th frame also seems reasonable. The same can be said for the predicted soot densities in figure 4.4, but the rollouts of the oxygen densities in figure 4.3 show bad performance for all models. The otherwise good results of S8gc are a bit peculiar when compared to the state and acceleration MSE; figure 4.1 A and B show that S8gc had the among the highest state and acceleration MSEs around these frames. However, S8gc had the lowest ξ in that range of frames according to figure 4.1 C. It could be that some parts of the predictions were somehow smeared out, which didn't affect ξ but negatively affected the state MSE. There is however no compelling evidence for that. It's also a possibility that the particular episodes that were inspected just happened to show a well-performing S8gc. Whatever the case is, the discrepancy observed indicates that the state and acceleration MSEs aren't necessarily absolute indicators of model performance.

Table 4.1 shows that S1 and S1g had the lowest episode MSEs. This can be attributed to the latter frames in the predicted oxygen densities, which for these models were less extreme than for S8, S8g and S8gc. The velocities and the HRRPUV weren't considered during the visual inspection, however, so there might be patterns in those features that are more revealing.

Among the models trained on BigSet, C8Short performed the best, followed closely by C8Flat and U8Short. Figure 4.7 shows that, while the general distribution of the oxygen matches the ground truth, U8Short predicted a region of lower oxygen density disjointed from the fire and smoke. C8Flat also appears to predict very noisy regions of low oxygen density, unlike ground truth and C8Short. These three models all had state and episode MSEs that were a couple of orders of magnitudes smaller than the best models trained on SmallSet. Their ξ s were also low, so both these measures correctly identified the best models. Neither of these indicated that C8Short was the best model, however.

Although all the models trained on BigSet except C1 used the 8-step corrective rollout loss, C8Long and U8Long performed much worse than the other three. The difference between these and their "short" versions was due to different h and γ , and due to having been trained for more epochs. They had a less aggressive ramping up of w due to the smaller h , and a slower learning rate decay due to the smaller γ . The small h resulted in models that were trained on low w for a longer time. As indicated

by the results, this slow ramping up of w is worse than quickly letting w converge to 0.5, or, as in the case of C8Flat, letting $w = 0.5$ for the whole duration of the training. The wildly differing results from changing the h and γ hyperparameters indicate that a finer hyperparameter tuning could increase the performance of the networks. Varying c could also potentially improve the networks, but this wasn't tested.

It's not clear why the networks were able to learn the BigSet scenarios better than the SmallSet scenarios. One possibility is that the grid resolution was a major deciding factor, while another possibility is that the greater variety in the initial conditions of BigSet resulted in a more robust training.

C8Short was tested on the generalising test sets. One can see from table 4.3 and figure 4.9 that its performance on the test sets was similar to its performance on BigSet. An inspection on some rollouts indicate that the model could generalise rather well; figure 4.10 and figure 4.11 show that reasonable soot patterns were predicted in TallSet and TallRoofSet. The predictions weren't perfect—in particular, the model didn't predict any build-up of soot in the corner of the ceilings.

C8Short had some trouble with BurnerSet, which didn't have any obstructions. The soot patterns in figure 4.12 (c) shows that the model inferred that there was a roof where there shouldn't have been one. This is possibly also visible in frame 14 of figure 4.12 (a), where some soot seem to branch off perpendicularly to the main smoke plume. This kind of error was a bit surprising, since the geometry of the domain was part of the input data. However, it was discovered that the geometries calculated for some episodes had missing roofs or walls, as shown in figure 4.13 and figure 4.14. The FDS simulations were done with the correct geometry, but the geometry inputs to the GNN were wrong. Assuming this also occurred in the training data, the model might've learned that roofless geometries would still have smoke that interacts with an invisible roof. This is a likely cause of the invisible roof in figure 4.12 (c).

The ξ of the BurnerSet was a bit erratic for the first 10 frames. It's not apparent if this correlates with the invisible roofs. In fact, ξ has been hard to interpret between all models and scenarios, which casts some doubt whether it is a good measure at all. Although better models generally had a lower ξ , they also had a lower state MSE. In the end, visual inspection was still the best way to evaluate the models in this study; after all, the fact that the rollouts on the BurnerSet had some glaring errors was only discovered through visual inspection.

The rollout speeds shown in table 4.4 indicates that using a GNN is faster than simulating with FDS. An advantage with using a GNN is that multiple rollouts can be parallelised and be computed on GPUs, while FDS simulations were restricted to CPUs. Interestingly, calculating at different grid resolutions using FDS didn't change the computing time. This is likely due to the FDS using a different internal resolution during simulation before exporting the data at the desired resolution.

If more time had been available, a more thorough hyperparameter tuning would've been made with the current results in mind. Scenarios with even more different geometries could also have been tested, both as test sets and as part of the training sets. This would also have helped in understanding why training on BigSet resulted in better learning than training on SmallSet.

5.1 Conclusion

In this work, a graph neural network was implemented according to the MeshGraphNets framework to learn the behaviour of fire and smoke generated by Fire Dynamics Simulator. This was successful with the 8-step corrective rollout loss, which was a loss function developed in this work, but requires some hyperparameter tuning for optimal performance. Several methods for evaluating the models' performances were used. The state MSE and a measure of plausibility ξ seemed to be good albeit rough predictors of model performance, but models that were close in terms of state MSE had to be evaluated through manual visual inspection of their outputs.

The trained models had some generalising capabilities, and could generally predict smoke patterns well when faced with test data outside of the training data's domain, but fell short in some specific cases due to bugs in the training data. Fixing these bugs and tuning the hyperparameters even more would've likely resulted in better models. Small improvements like these notwithstanding, the method seems to work well enough for future work to study whether the networks can learn to predict states using RGB video input. An approach similar to the one used by Watters et al. (2017) can be tested, where a convolutional network encodes the RGB video into a state code before it gets processed further.

References

- Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). *Layer Normalization*. arXiv: 1607.06450 [stat.ML].
- Battaglia, Peter W., Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu (2016). *Interaction Networks for Learning about Objects, Relations and Physics*. arXiv: 1612.00222 [cs.AI].
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press.
- Hodges, Jonathan L., Brian Y. Lattimer, and Kray D. Luxbacher (2019). “Compartment fire predictions using transpose convolutional neural networks”. In: *Fire Safety Journal* 108, p. 102854. ISSN: 0379-7112. DOI: <https://doi.org/10.1016/j.firesaf.2019.102854>. URL: <https://www.sciencedirect.com/science/article/pii/S0379711218304922>.
- Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].
- Loshchilov, Ilya and Frank Hutter (2019). *Decoupled Weight Decay Regularization*. arXiv: 1711.05101 [cs.LG].
- Mehlig, B. (2021). *Machine learning with neural networks*. arXiv: 1901.05639 [cs.LG].
- National Institute of Standards and Technology (2020-08-21). *Fire Dynamics Simulator and Smokeview*. Version 6.7.5, 6.7.15. URL: <https://pages.nist.gov/fds-smv/>.
- Pfaff, Tobias, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia (2021). *Learning Mesh-Based Simulation with Graph Networks*. arXiv: 2010.03409 [cs.LG].
- Sanchez-Gonzalez, Alvaro et al. (2018). *Graph networks as learnable physics engines for inference and control*. arXiv: 1806.01242 [cs.LG].
- Thuerey, Nils, Konstantin Weßenow, Lukas Prantl, and Xiangyu Hu (2020-01). “Deep Learning Methods for Reynolds-Averaged Navier–Stokes Simulations of Airfoil Flows”. In: *AIAA Journal* 58.1, pp. 25–36. ISSN: 1533-385X. DOI: 10.2514/1.j058291. URL: <http://dx.doi.org/10.2514/1.j058291>.
- Watters, Nicholas, Andrea Tacchetti, Theophane Weber, Razvan Pascanu, Peter Battaglia, and Daniel Zoran (2017). *Visual Interaction Networks*. arXiv: 1706.01433 [cs.CV].
- Zhou, Jie et al. (2021). *Graph Neural Networks: A Review of Methods and Applications*. arXiv: 1812.08434 [cs.LG].

