



October 2021
asundail@cern.ch

CERN Summer Student Project Report

Albert Sund Aillet

Supervisors: Roman Stoklasa & Luigi Serio

(ATS-TE-CRG), Data and Image Analysis Group (DIAG)

Keywords: Graph Neural Networks, Deep learning, Physics simulations, Image processing

Summary

This summer student project consisted of three subprojects. First, two introductory subprojects were conducted as an introduction to the Tensorflow environment, the graph generation, graph batching and the training loop. Then followed a subproject that aimed to model a physics simulation of connected vessels. Finally an image processing subproject was conducted where the goal was to detect cracks from shortest-paths tree graphs generated from images of road cracks. In all of these subprojects, a so-called EncodeProcessDecode model from the Graph Nets Library was used with different numbers of processing steps. The first of the two introductory subprojects showed that the Graph Neural Network (GNN) could correctly sort the values represented by a fully connected graph. The mistakes made by the GNN occurred when two of the node values were close to each other and they were being predicted as being connected when they should not. The second introductory subproject showed that the GNN could learn to highlight the shortest path in a graph with weighted edges, but as in the first introductory subproject, the GNN sometimes highlighted two paths with similar lengths. The physics simulation subproject was performed by multiple experiments concluding that a GNN can reliably approximate a simulation of a physical system without any prior knowledge of it, simply by being trained on observations of the system. The crack detection subproject was performed by different runs of input data and target data of images of road cracks and resulted in the finding that it is possible for the GNN to highlight the underlying cracks in the shortest-tree graph of parts of the image. The summer student project was successful in showing that GNN methods have potential to be of use in a wide range of problems and can be useful for various tasks at CERN.

Contents

1	Introduction	3
2	Graph Neural Network Theory	3
3	Graph Nets model properties	4
4	Introductory Subprojects	6
4.1	Sorting Problem	6
4.1.1	Problem Formulation	6
4.1.2	Results	6
4.2	Shortest Path Problem	8
4.2.1	Problem Formulation	8
4.2.2	Results	9
5	Physics Simulation - Communicating Vessels	10
5.1	Problem Formulation	10
5.2	Experiments and Results	11
5.2.1	First Experiment	11
5.2.2	Second Experiment	12
5.2.3	Third Experiment	12
5.2.4	Fourth Experiment	13
5.2.5	Fifth Experiment	15
5.2.6	Sixth Experiment	16
5.2.7	Seventh Experiment	17
5.3	Subproject Conclusion	18
6	Image processing - Crack detection	18
6.1	Problem Formulation and design	18
6.2	Results	19
6.3	Subproject Conclusion	23
7	Conclusion	24
8	Acknowledgements	25
9	References	25
A	Appendix	26

1 Introduction

This summer student project consisted of three subprojects. First, two introductory subprojects were conducted as an introduction to the Tensorflow environment, the graph generation, graph batching and the training loop. After that, a Graph Neural Network (GNN) model was used to approximate a simple physics simulation consisting of communicating vessels. Thirdly a GNN was trained to detect cracks from shortest-paths tree graphs generated from images of road cracks. Before the subprojects are presented a short description of the GNN theory as well as the model architecture is provided as a background to the work.

2 Graph Neural Network Theory

Constructing new inferences, predictions, and behaviors from known building blocks, also called combinatorial generalization is an important priority for artificial intelligence to achieve human-like abilities [3].

Using relational structures can facilitate learning about entities, relations, and rules for composing them. This motivates the need for a deep learning component which operates on an arbitrary relational structure [3]. Graphs are representations that support an arbitrary pairwise relational structure.

An inductive bias is a choice in the design of the learning algorithm that allows it to prioritize one solution over another. For example a convolution in a Convolutional Neural Network (CNN) operates by sliding a kernel over a grid and calculating the scalar product between the kernel and different parts of the grid. A convolutional layer of a CNN is made up by several of these kernels. This building block imposes locality and translation invariance. Recurrent layers is Recurrent Neural Networks (RNNs) are implemented over a sequence of steps where the same function is reused across different processing steps. This building block imposes the relational inductive bias of temporal invariance.

Computations over graphs afford a strong relational inductive bias beyond that which convolutional and recurrent layers can provide since they can impose any arbitrary pairwise relational inductive bias. Graph Neural Networks (GNNs) are neural networks that operate on graphs and can be designed to classify whole graphs or edges or nodes of a graph.

There are two important properties of GNNs that especially are worth mentioning. The first is that the same model can be trained and used to predict graphs with a variable amount of nodes and vertices. The second one is that some important computations in the model are required to be permutation invariant or equivariant in relation to node and edge ordering so that the way in which the graph is stored does not matter, and the permutation invariant relation between the individual nodes and all their connected edges is preserved.

3 Graph Nets model properties

The model used in the subprojects of this report is based on the Graph Network (GN) block introduced in the article *Relational inductive biases, deep learning, and graph networks* by Battaglia et al. [3]. The GN block, pictured in figure 1, performs the message passing between the nodes and the edges of the input graph. These message passing steps are for GNNs what convolutions are for CNNs. They allow information to flow according to the structure of the graph, meaning that the information in a certain vertex propagates to its direct neighbors in each message passing step, allowing the GNN to learn the patterns present in the graph.

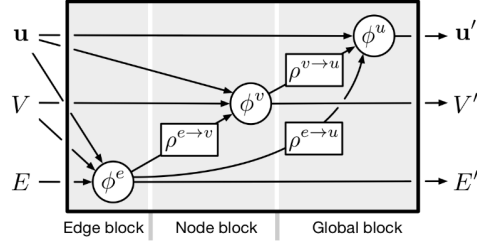


Figure 1: Message Passing GN Block [3].

$$\begin{aligned}
 \mathbf{e}'_k &= \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) & \bar{\mathbf{e}}'_i &= \rho^{e \rightarrow v}(E'_i) \\
 \mathbf{v}'_i &= \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}) & \bar{\mathbf{e}}' &= \rho^{e \rightarrow u}(E') \\
 \mathbf{u}' &= \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}) & \bar{\mathbf{v}}' &= \rho^{v \rightarrow u}(V')
 \end{aligned}$$

\mathbf{v}_i : the attributes of vertex i

\mathbf{e}_k : the attributes of edge k

\mathbf{u} : the global attributes

N^v : the number of vertices

N^e : the number of edges

$E'_i = \{(\bar{\mathbf{e}}'_i, r_k, s_k)\}_{r_k=i, k=1:N^e}$: the neighboring edges of the vertex v_i

$V' = \{\mathbf{v}'_i\}_{i=1:N^v}$: the set of all vertices

$E' = \{(\mathbf{e}, r_k, s_k)\}_{k=1:N^e}$: is the set of all edges

The prime ($'$) notation refers to the updated values after the update function and the bar ($\bar{}$) notation refers to aggregated attributes after the aggregation function.

The aggregation functions ρ summarize attributes of the same type using permutation invariant operations as for example sum, max, min or mean. The update functions ϕ summarize different attribute types and do not require to be permutation invariant. Therefore

the update functions often contain learnable parameters of the model. These two functions decide how the message-passing is performed.

The model architecture used in the subprojects can be seen in part c) of figure 2 and includes three components. The first component is an encoder GN block that independently encodes the edge, node and global attributes using an independent GN block pictured in figure 3. The core consists of a sequence of M message-passing GN blocks and can be seen in part a) of figure 2. Finally the third component is a decoder GN block that independently decodes the edge, node and global attribute of each message passing step to the output of the neural network that is to be compared with the target. This model architecture is referred to as **EncodeProcessDecode** in the code [1] [2].

In the implementation used in the subprojects, the three update functions ϕ in the message-passing GN block are three different Multi Layer Perceptrons (MLP) with 2 latent layers with 16 nodes in each layer and the aggregation functions ρ are all the sum operation. In the encoder and decoder independent GN blocks the three update functions ϕ are also MLPs with 2 latent layers with 16 nodes in each layer.

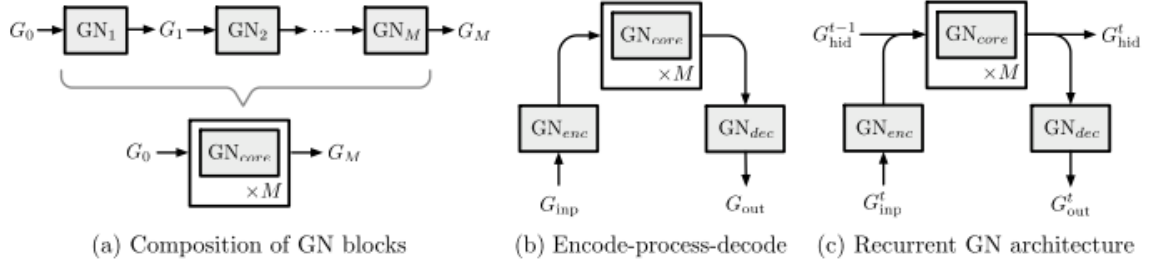


Figure 2: Recurrent GN Block.

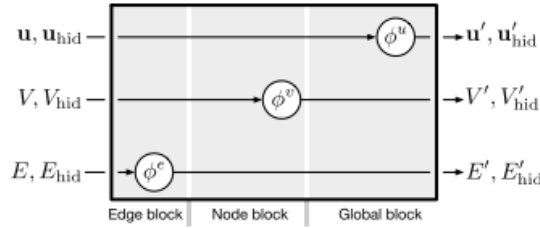


Figure 3: Independent GN Block.

4 Introductory Subprojects

The first part of this project includes a couple of introductory subprojects as an introduction to the graph generation, graph batching and the training loop, replicating and modifying available demos from the Graph Nets Library [2].

4.1 Sorting Problem

4.1.1 Problem Formulation

This subproject was directly inspired and used code from the sorting demo from the Graph Nets Library [2].

In this subproject, the input graphs are fully connected graphs where the nodes have a single feature of a value between 0 and 1. The target graphs are graphs with the same structure as their corresponding input graphs, where the directed edges and the nodes both have a two-dimensional feature vector. A directed edge of the graph is labelled $[1,0]$ if it is part of the directed path between the smallest and the largest value and labelled $[0,1]$ otherwise. The node with the smallest value is labelled $[1,0]$ and all other nodes are labelled $[0,1]$.

In figure 4 a visualization of a sample input and target graph is shown, alongside the target graph where only the path is presented.

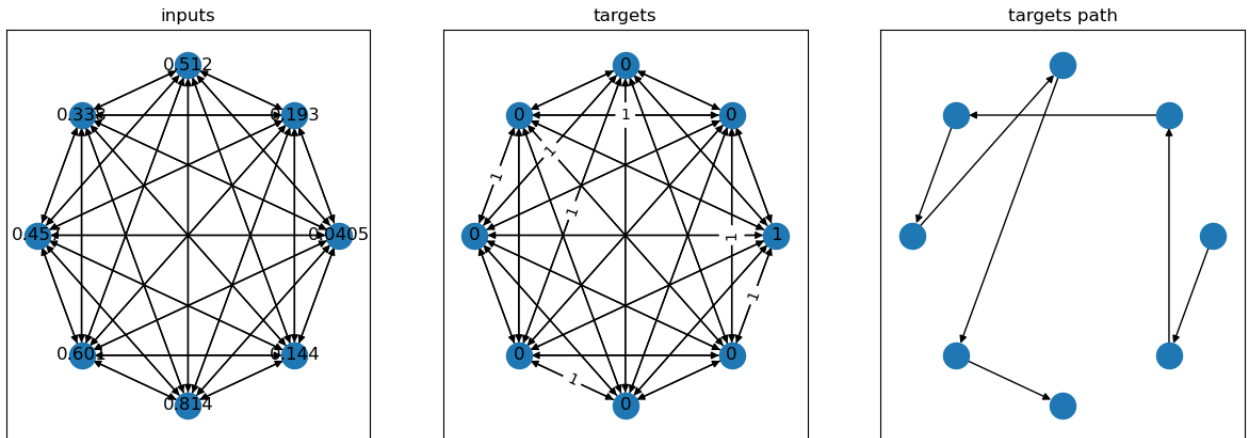


Figure 4: Visualization of input and target data.

The goal is to train a GNN to correctly identify the starting nodes and the path from the smallest to the largest values.

Since these graphs can be generated with their corresponding labels, it is possible to compare the output of the GNN to the ground truth.

4.1.2 Results

The model is trained on graphs with 8 to 16 nodes (inclusive) and then tested for generalization on graphs with 16 to 32 nodes (inclusive). New graphs for both training and

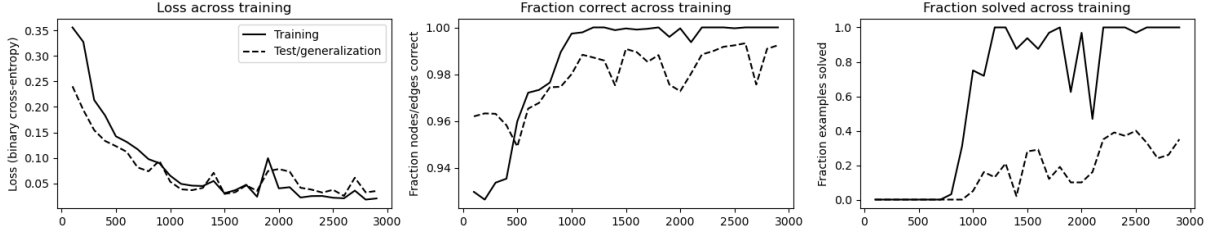


Figure 5: Loss and fraction of correctly predicted label and fraction of solved graphs for every training iteration.

generalization are generated for every training iteration. The code is available on GitLab [1].

The calculated loss is the cross entropy loss on both the node and edge labels. Figure 5 presents how the GNNs learns to predict the sorted path, as the loss decreases and the fraction of correctly labelled edges and nodes approaches 1. The fraction of solved examples is unstable, since if a single edge or node of a graph is misclassified the whole graph is counted as not solved.

Figure 6 is a plot of the predicted edge features on a generalization graph with 30 nodes (the GNN was only trained on graphs with max 17 nodes). The entries in the plot are sorted, therefore the ground truth is the diagonal shifted by one position towards the right side. The predicted edges are always between nodes that are close to each other in value, but as can be seen in the last column of figure 6, the GNN classifies some of the close values as being connected when they should not. Most of the time, these falsely connected nodes have values that are very close to each other. To get a more appropriate idea of how well the model performs, it could be of interest to implement a "weighted" failure score, that would take into account how close the values of the incorrectly connected nodes are. If two nodes have values that are very close to each other, they should be penalized less in this weighted failure score.

Another way to get a better representation of the performance of the model would be to include some post-processing step that converts the output graph into a sorted list of values. This could be done by converting the output graph into a weighted graph (for example taking the inverse of the prediction probability as weight for the edges) and finding the shortest path in the graph that includes all values.

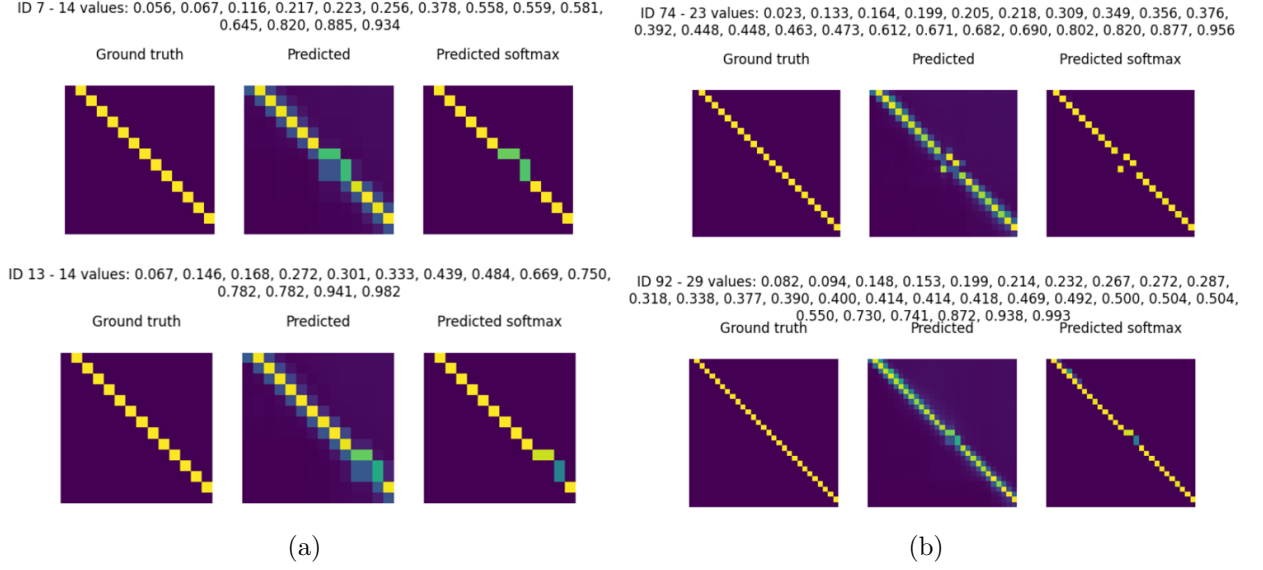


Figure 6: Plot of predicted edge labels where the GNN makes mistakes on (a) training set and (b) generalization set. The values are the values present in the graph to be sorted. In the graph with ID 7 the nodes that have been incorrectly connected have values 0.558 and 0.559, in the graph with ID 13 they have values 0.782 up to three decimal places, in the graph with ID 74 they have values 0.448 up to three decimal places and in the graph with ID 92 they have values 0.414 up to three decimal places. The full images can be found on [GitLab \[1\]](#).

4.2 Shortest Path Problem

4.2.1 Problem Formulation

This subproject was directly inspired from the shortest path demo from the Graph Nets Library [2] but was designed a bit differently.

The input graphs are directed weighted graphs with random structures, where the average rank of a node is 3. The directed edges are randomly weighted with values between 0 and 1, uniformly sampled. The graph is constructed in a way such that if there is an edge from node u to node v , there is also an edge from node v to node u , but they can have different weights. The nodes are labelled with a single node feature. One node is marked with a value of -1, representing the fact that it is the starting node. Another node is marked with a value of 1, representing the fact that it is the end node. All other nodes have a value of 0.

The target graphs are graphs with the same structure as their corresponding input graphs, where the directed edges and the nodes both have a two-dimensional feature vector. A directed edge of the graph is labelled [1,0] if it is part of the shortest path between the start and end node and labelled [0,1] otherwise.

The goal is to train a GNN to correctly identify the shortest path in the input graph between the start and end node.

4.2.2 Results

The training scripts are available on GitLab [1]. As in the sorting example, the graphs are generated “on the fly” for every iteration, meaning that the training data is refreshed for every training iteration. The loss is calculated using cross entropy loss on the predicted edge labels and the predicted node labels are discarded.

Figure 7 shows that the GNN learns to predict the shortest path, as the loss decreases and the fraction of correctly labelled edges and nodes approaches 1. The fraction of solved examples is unstable, since if a single edge or node of a graph is misclassified the graph is counted as not solved. A relatively small part of the graphs is solved even though the fraction of correct labels is more than 93%. Why this is the case is investigated by studying two examples, one graph part of the training set and another from the generalization set. In the output graphs, probabilities larger than 0.1 are shown and those smaller than 0.1 are not shown.

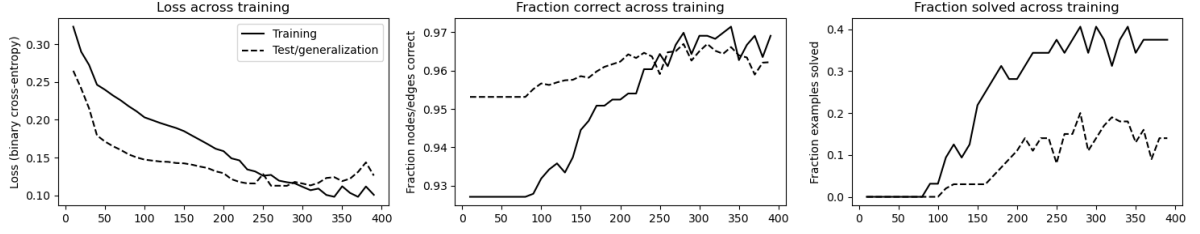


Figure 7: Loss and fraction of correctly predicted label and fraction of solved graphs for every training iteration.

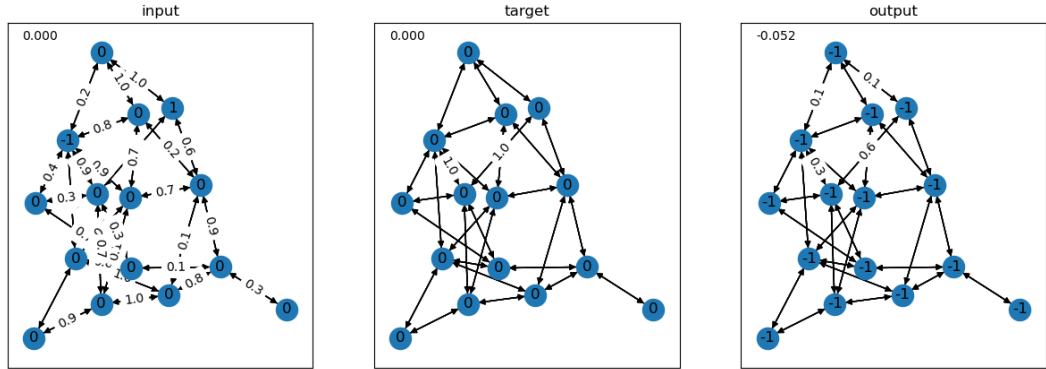


Figure 8: Visualization of input, target and output graph of a training data sample.

Figure 8 shows the input and target graph of a training data sample as well as the output graph from the GNN. In the output, two paths connecting the start node and the end node are highlighted with probabilities larger than 0.1. These paths are similar in length, and the model still seems to choose the correct one, as the correct path is estimated with probabilities 0.3 and 0.6 while the incorrect one is estimated with probabilities 0.1 and 0.1.

Figure 9 shows that the output graph correctly labels the two edges that are part of the shortest path in the target graph, but also labels neighbouring edges that are not part of

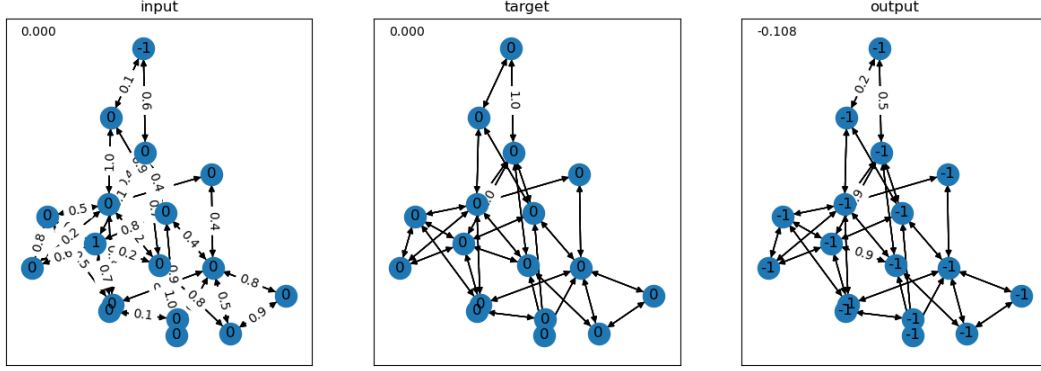


Figure 9: Visualization of input, target and output graph of a generalization data sample.

the path incorrectly with a high probability of 0.9. This means that the GNN makes critical mistakes in some of the graphs.

A possible improvement to this experiment would be to label the nodes in the target graph as being part of the shortest path to get a stronger feedback/training “signal”.

5 Physics Simulation - Communicating Vessels

5.1 Problem Formulation

This subproject is somewhat inspired by the physics demo from the Graph Nets Library [2] and is designed to show the performance of GNNs when it comes to model a physics simulation that can also be simulated analytically, specifically a system of communicating vessels. An example of a system with two vessels connected with a pipe can be seen in figure 10.

The vessels are represented as nodes in the graph and have the following features:

- **base_area** : the area of the base of the vessel.
- **base_altitude** : the altitude of the vessel base measured from a global zero level.
- **water_level** : the level of water above the base of the vessel.

The pipes are represented as edges in the graph and are given the following features:

- **altitude** : the altitude of the pipe from a global zero level.
- **diameter** : the diameter of the circular cross section of the pipe.
- **length** : the length of the pipe.

Some parameters of the simulation are included as global features of the graph:

- **g** : the gravitational constant used in the simulations.
- **viscosity** : the viscosity of the simulated fluid.

- **density** : the density of the fluid.

The full description as well as an explanation of the physics behind the simulation can be found on [GitLab \[1\]](#). The GNN is trained to predict the state of the system for the next time step. The model's next-step predictions can be fed back as input to create the following step. This means that the trained GNN can be used to simulate the system evolution instead of using a simulator.

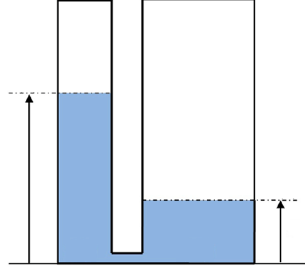


Figure 10: Diagram of vessels physics problem.

5.2 Experiments and Results

Multiple different experiments were conducted to be able to train a model that could reliably simulate the system.

5.2.1 First Experiment

The goal of the first experiment was to prove the performance of the system on a very restricted example, to see that the training loop was buildt correctly and that the model had the capacity to learn a simple time step. This experiment uses only 1 time step of a specific system with a specific water level state as training data, and the model is trained on this single time step for 2400 iterations.

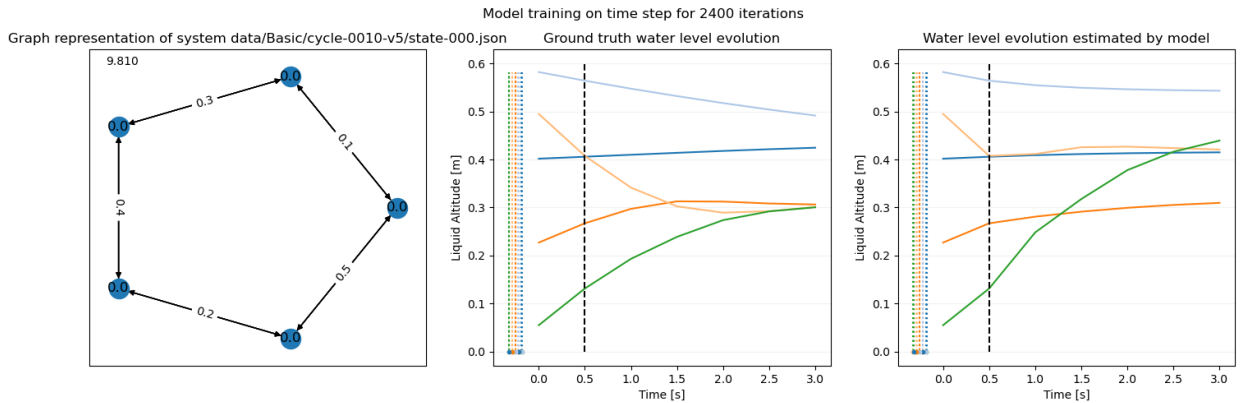


Figure 11: Results of the first experiment.

In figure 11 it can clearly be seen that the model is able to learn the single step that was provided. As expected, the model does not generalize to more time steps other than the one trained on.

5.2.2 Second Experiment

As the performance of the model was shown on a single time step, the second experiment is conducted to expand the training data to include more time steps so that the model can learn more general patterns in the simulation, without adding too much variability by restricting the training data to a single system with a specific water state. In the second experiment, the model is trained on 100 succeeding time steps of a specific system, with a specific water level state. The model is trained for 4000 training iterations on this restricted data. The results can be seen in figure 12. The first row is the system present in the training data and the two other rows are the same system but with different initial water levels.

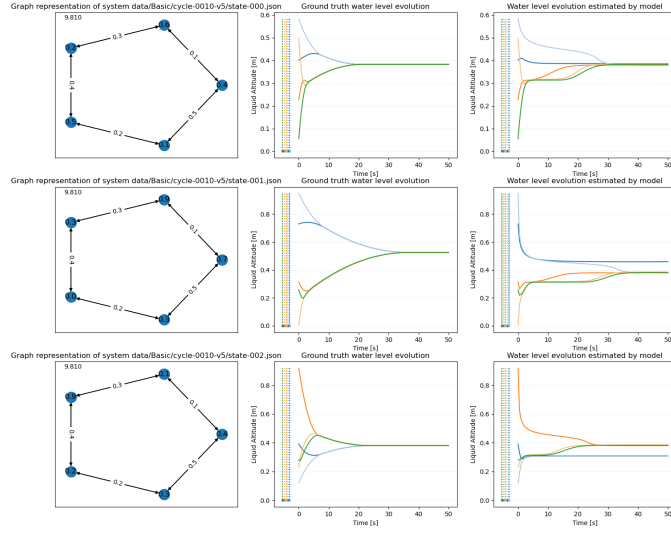


Figure 12: Results of the second experiment.

As can be seen by the water level evolutions, the model does not generalize well to systems with different starting conditions and is bad at even predicting the exact system that it was trained on.

The poor results of this experiment show that training on succeeding time steps is not a good idea. It can be theorized that this is due to the training data comprising many similar examples and that many of them include the system in an equilibrium state. This would result in the model not generalizing well since it has only been trained on a small number of significantly different systems.

5.2.3 Third Experiment

Since the second experiment showed that training on succeeding time steps resulted in poor results, this experiment instead uses non-succeeding time steps and different initial water states to provide the model with more variability in the data. In this experiment, the

training is instead done on 100 non-succeeding time steps where the initial water levels are randomly sampled. The same system of vessels and pipes is used and all vessel base areas and pipe lengths are set to a fix value. The model is trained for 200 training iterations on this restricted data. To produce the results, the model simulates 10 time steps by running the system through the model in 10 succeeding steps.

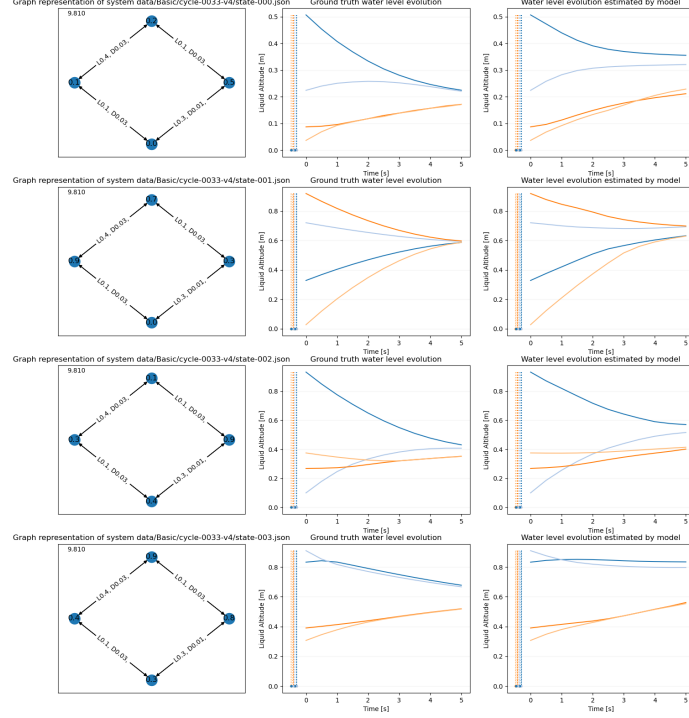


Figure 13: Results of the third experiment.

Figure 13 shows the predictions of the model on systems with the same structure (meaning arrangement of vessels and pipes) as the ones in the training set, but with randomly generated water levels. It is apparent that even if the model makes some small mistakes when simulating for a longer time, the estimated water level evolution is close to the ground truth simulated one.

5.2.4 Fourth Experiment

As the performance of the model on a very restricted case has been shown, the next few experiments add complexity to the training data and assess the model performance. It is important to note that the systems that are shown in figures 14-17 are part of the test set, meaning that they are unknown to the model. In the experiments four to seven the model is trained on non-succeeding time steps of 250 different vessel systems that are each sampled 100 times with random water levels, for a total of 25 000 example training samples. The training and testing samples are of a large variety of topologies and number of vessels. The generated graph can either be a cycle graph (meaning that the edges form a cycle), a line graph (meaning that the edges form a line) or a star graph (meaning that all nodes are connected to one central node). The graph can also be generated by walking randomly to

nodes and connecting them (called *snake* graph in the code) or be completely randomly generated (called *random* in the code).

The variation introduced in this experiment is the large number of different graph topologies, but to reduce variability from other sources of the simulator, many of the parameters are set to constant values. In the fourth experiment, all vessel base areas and pipe lengths and diameters are set to a fix value. The altitudes of both the vessels and the pipes are set to zero, meaning that there can always be flow between connected vessels. The model is trained for 3000 training iterations on the training data.

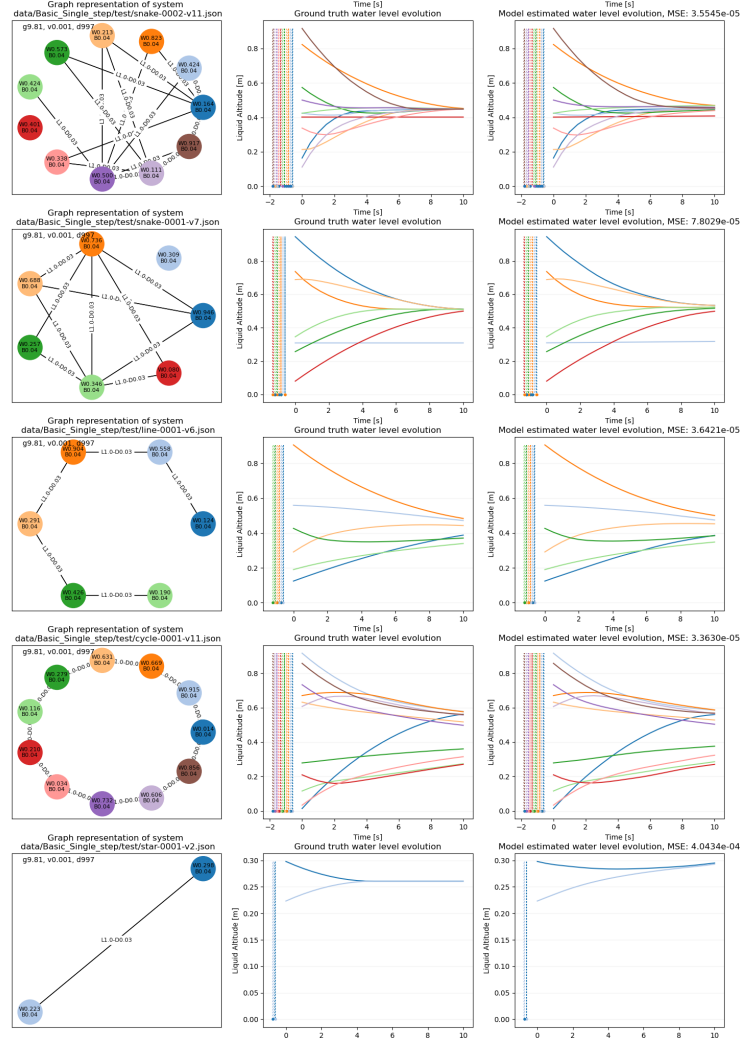


Figure 14: Results on test set of the fourth experiment.

Figure 14 show that the model is able to simulate a system and replicate the ground truth evolution. In the first and second rows, the estimated evolution of the vessels without any connected pipes shows that the model understands to keep the water level of non connected vessels constant. On the other hand, as seen in the last row of figure 14 the model may make wrong predictions on systems with a low number of vessels.

5.2.5 Fifth Experiment

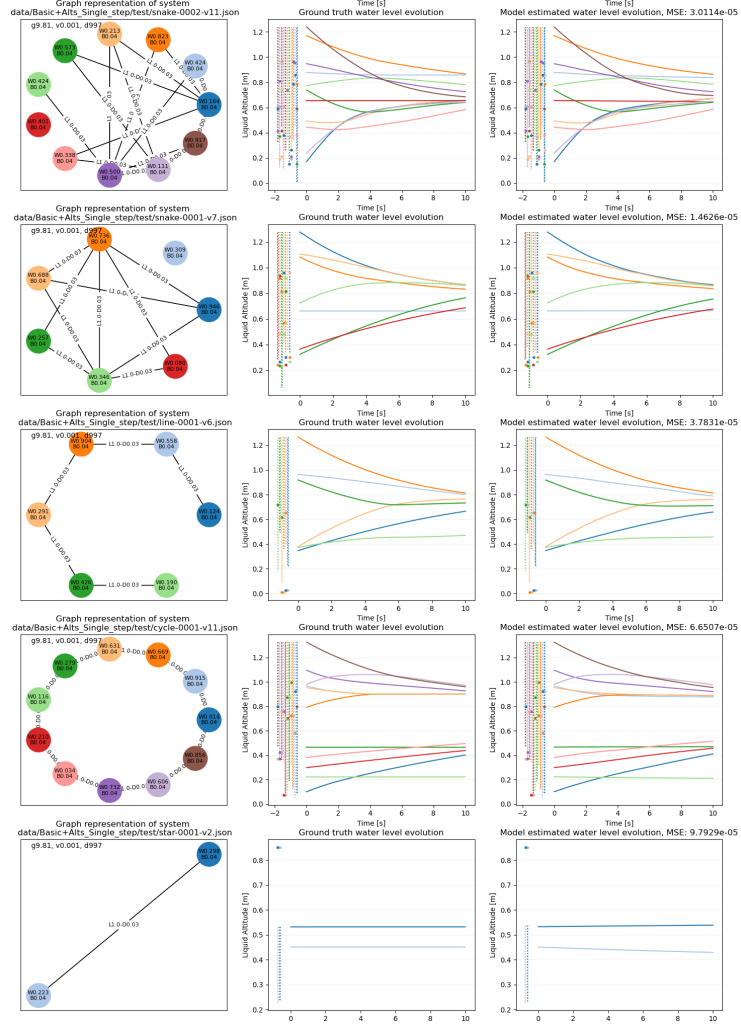


Figure 15: Results on test set of the fifth experiment.

The variation introduced in this experiment is the altitudes of the pipes, which means that the model now has to learn that sometimes there cannot be flow between vessels since the pipe is placed higher than the water level of the vessels.

In the fifth experiment all vessel base areas and pipe lengths and diameters are also set to a fix value. The altitudes of the vessels are set to zero while the altitudes of the pipes can be above the ground. The pipe altitude can take values between 0 and 0.25 m. The model is trained for 3000 training iterations on the training data.

In the third and fourth rows of figure 15 it is apparent that the model has learned that water should not flow through pipes that are placed higher than the water level.

5.2.6 Sixth Experiment

In this experiment, variation is introduced as the altitudes of the vessels, which means that the model now has to learn that sometimes there cannot be flow between vessels since the pipe is placed higher than the water level of the vessel, relative to the original zero level.

In the sixth experiment all vessel base areas and pipe lengths and diameters are still set to a fix value. The altitudes of both the vessels and the pipes can be above the ground. The vessel base altitude can take values between 0 and 0.5 m and the pipe altitude can take values between 0 and 0.25 m. The model is trained for 3000 training iterations on the training data.

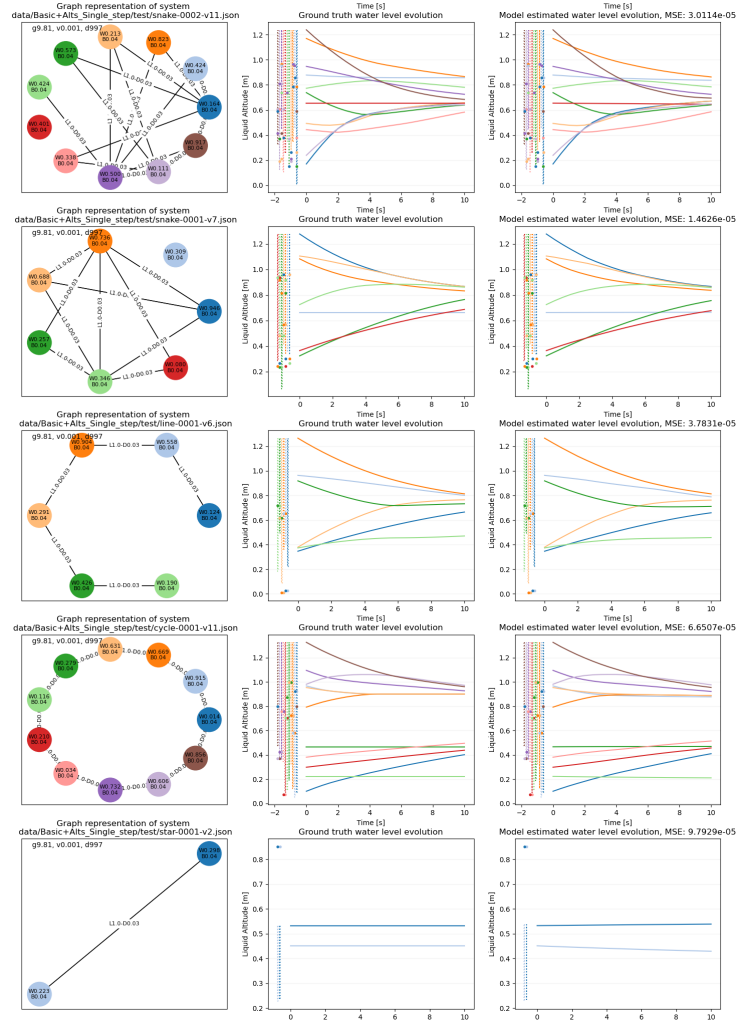


Figure 16: Results on test set of the sixth experiment.

In figure 16 it is apparent that the model approximates the ground truth simulated system well. This implies that in some way it has learned to sum up the altitude of the vessel and the water level to be able to compute the resulting flow.

5.2.7 Seventh Experiment

In this last experiment, almost all of the possible system parameters are varied and to find out if the model can learn the impact of all these parameters, simply by being exposed to simulation step examples. The vessel base areas can take values between 0.01 and 0.05 m². The pipe length can take values between 0.5 and 1 m. The pipe diameters can take values between 0.01 and 0.025 m². The vessel base altitude can take values between 0 and 0.5 m. The pipe altitude can take values between 0 and 0.25 m. The model is trained for 3000 training iterations on the training data.

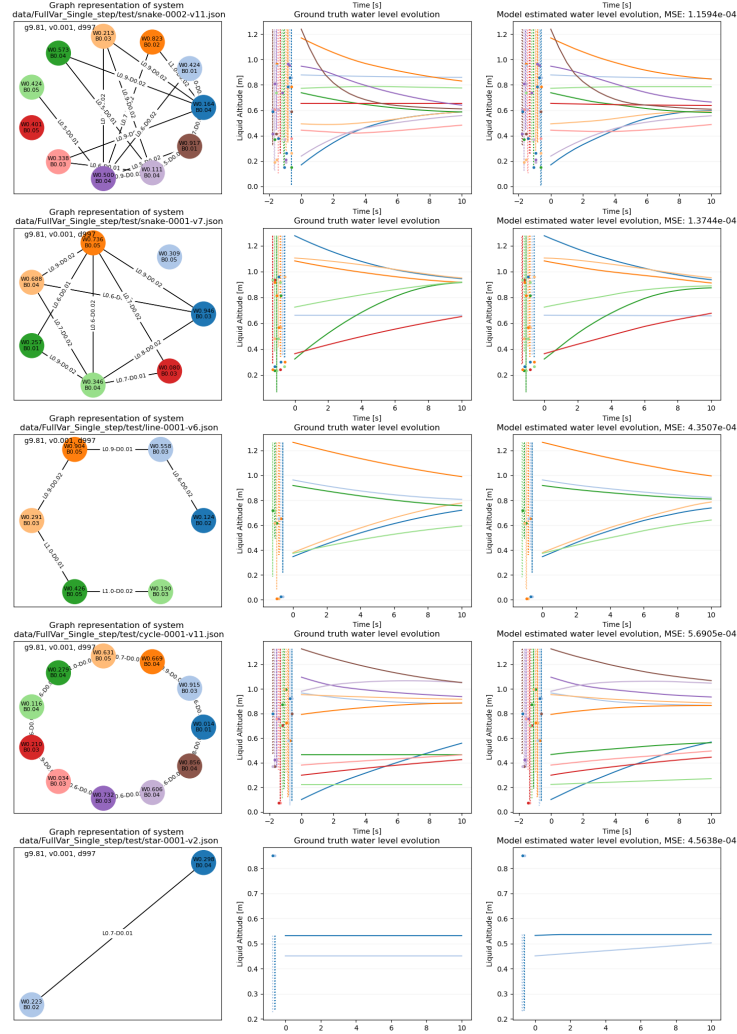


Figure 17: Results on test set of the seventh experiment.

In figure 17 the model is shown to very well approximate the ground truth simulated system even though all this complexity is introduced. It still seems that model has the hardest time to approximate the system with two vessels and one pipe, since it may be a case that did not appear many times in the training data.

5.3 Subproject Conclusion

To conclude this subproject, it has been shown that a GNN can reliably approximate a simulation of a physical system without any prior knowledge of it. The only thing required for the model to be appropriately trained is a large set of diverse simulation examples.

6 Image processing - Crack detection

6.1 Problem Formulation and design

The last conducted subproject is about crack detection and segmentation in images of cracks, and uses a dataset called CrackForest also used by Shi et al. [6]. The dataset consists of 480x300 pixel PNG images of cracks as well as segmentation maps (one for each of the 118 images).

Certain sampled pixels in these images are used as seed points for a shortest-paths tree graph generation algorithm similar to those used by Chen et al. [4] or Kaul et al. [5]. The graph and image crop seen in figure 18 show an example of a graph generated with this method and the image from which it was generated. The idea of the algorithm is to generate the tree path along the crack part of the image (meaning the darkest parts of the image). Certain nodes of these generated graphs are then removed to exclude any parts of the image that should not be considered, for example when the algorithm continues generating the graph outside the image, as can be seen in figure 19, this part of the graph has to be removed.

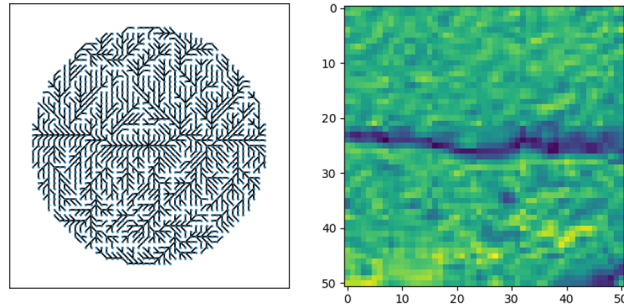


Figure 18: A shortest-paths tree graph created from seed pixel (226,196) in image 002.png. The shortest-paths tree has horizontal paths that clearly follow the horizontal crack, making it apparent from the tree structure.

These graphs, as well as the data generated from the shortest-paths tree algorithm are then fed as input to the GNN and the task for the model is to highlight the path in the graph that makes up the crack.

The ground truth map used is a cropped version of the original segmentation ground truth, that only highlights one path through the graph that follows the crack. This means that the task for the model is, similarly as in the shortest path and sorting task, to find a path through the given graph that in this case follows the crack in the underlying image.

The dataset consists of 118 images from which seed pixels are sampled and graphs created. The samples from images 1-100 are used as the training data, the samples from images 101-

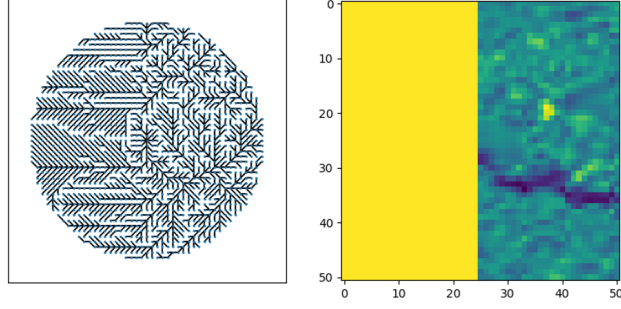


Figure 19: A shortest-paths tree graph created from seed pixel (001,164) in image 001.png, where a part of the graph has been created outside the image borders and needs to be removed because it does not represent a real solution.

110 are used as the validation data and the samples from images 111-118 are used for the test set.

6.2 Results

Seven different experiments were completed, where the input data and the target data varied between the different runs (run description available on GitLab [1], under `runs/run_descriptions.md`)

The last run is the one that seems to produce the best results and is the one presented in further detail. In this run, the input graph does not only include the edges from the shortest-paths tree algorithm, but also includes edges between nodes that are next to each other in the original image. In figure 20 the same graph with and without these image edges is depicted. To make them distinguishable to the GNN the edges of the shortest-paths tree are given a feature 1 and the ones that represent the pixel neighbourhood are given a feature 0.

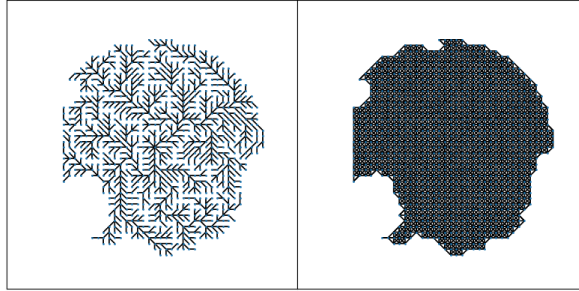


Figure 20: Graph without (left) and with (right) image edges.

The performance of the model is assessed using precision, recall and F1-score. Precision gives the fraction between the number of true positives and the number of positives in the prediction:

$$Precision := \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \quad (1)$$

Recall gives the fraction between the number of true positives and the number of positives in the ground truth:

$$Recall := \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \quad (2)$$

The F1-score is the harmonic mean of precision and recall:

$$\text{F1-score} := 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)$$

The model is trained for a total of 500 epochs, but the model with the lowest validation loss, from epoch 490 is kept. The training took around two days to complete. In appendix figure 26 a few samples from the validation set and their prediction from the model are presented.

To get a binary map from these predictions, a relevant threshold to decide what values are considered part of the crack has to be chosen. To find this, two methods are used. First, the Precision-Recall (PR) curve and F1-curve of every sample in the validation set is computed. The PR-curve plots the Precision and Recall (for which the formula is given in equation (1) and (2)) for different values of the threshold. In figure 21 such a PR curve is shown, and it is apparent that a compromise between precision and recall has to be made. If a too low threshold is chosen, pixels that are not part of the crack will be predicted as positive, while if the threshold is too high, some pixels that should be predicted as positive would be missed. In figure 21 at the top of the sample there is a part of the crack that is only very faintly highlighted by the GNN, but if the threshold is set too low to include this, it may also include other pixels that do not form a crack.

The PR-curve and F1-curve of some representative samples of the validation set is presented in appendix figure 27. An interesting observation to note is that even though the ground truth is not perfect in many of the cases and may contain breaks in the path, the model seems to still find a continuous path in the graph. Then all the F1-curves of the individual samples are averaged to compute a mean F1-score, presented in figure 22. The formula of the mean F1-score over the validation set is given by

$$\text{Mean F1-score} = \sum_{\substack{\text{all samples } i \\ \text{in validation set}}} \frac{\text{Precision}_{\text{sample } i} \cdot \text{Recall}_{\text{sample } i}}{\text{Precision}_{\text{sample } i} + \text{Recall}_{\text{sample } i}} \quad (4)$$

Secondly, all pixels (or nodes) of all samples in the validation set are considered together and the total Recall, Precision and F1-score can be computed for every threshold. The total PR-curve and F1-curve of the validation set can be seen in figure 23. The formula of the total precision, recall and F1-score over the validation set is given by

$$\text{Precision}_{\text{Total}} := \frac{\sum_{\substack{\text{all samples } i \\ \text{in validation set}}} \text{True Positives}_{\text{sample } i}}{\sum_{\substack{\text{all samples } i \\ \text{in validation set}}} \text{True Positives}_{\text{sample } i} + \text{False Positives}_{\text{sample } i}} \quad (5)$$

$$\text{Recall}_{\text{Total}} := \frac{\sum_{\substack{\text{all samples } i \\ \text{in validation set}}} \text{True Positives}_{\text{sample } i}}{\sum_{\substack{\text{all samples } i \\ \text{in validation set}}} \text{True Positives}_{\text{sample } i} + \text{False Negatives}_{\text{sample } i}} \quad (6)$$

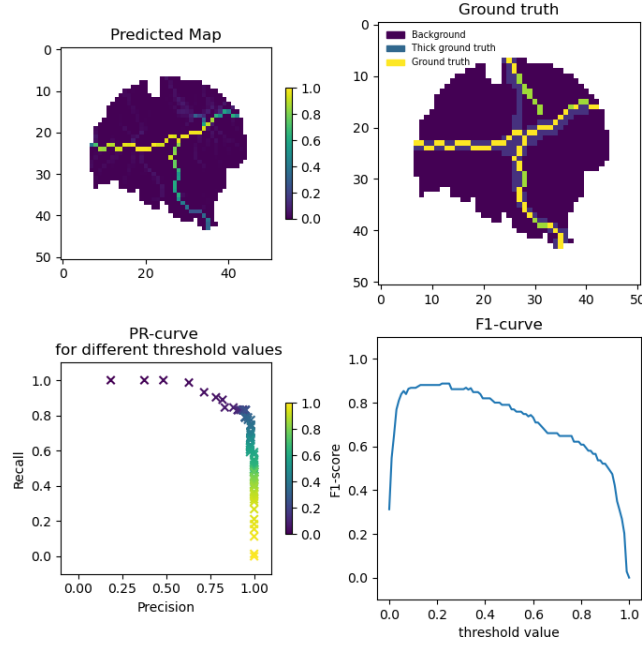


Figure 21: A sample from the validation, its corresponding ground truth as well as the accompanying PR and F1-curve.

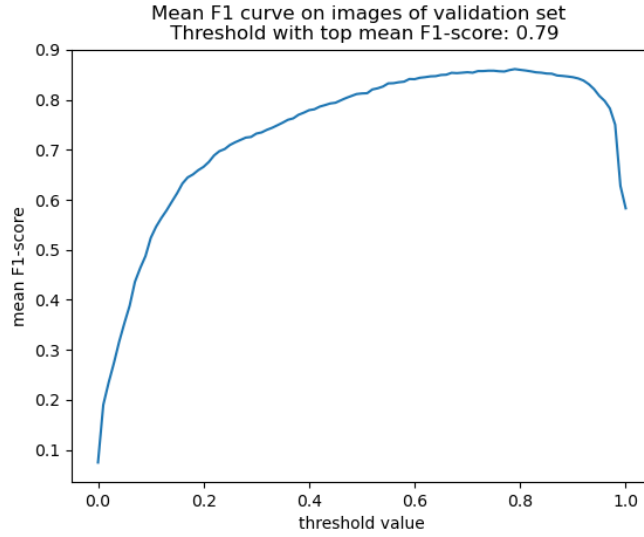


Figure 22: Mean F1-curve for validation set, taking the average of all individual F1-curves.

$$\text{Total F1-score} = \frac{\text{Precision}_{\text{Total}} \cdot \text{Recall}_{\text{Total}}}{\text{Precision}_{\text{Total}} + \text{Recall}_{\text{Total}}} \quad (7)$$

A threshold of 0.61 that maximizes the F1-score of all the samples in the validation set is chosen. The final total metrics of the test set for the threshold 0.61 are:

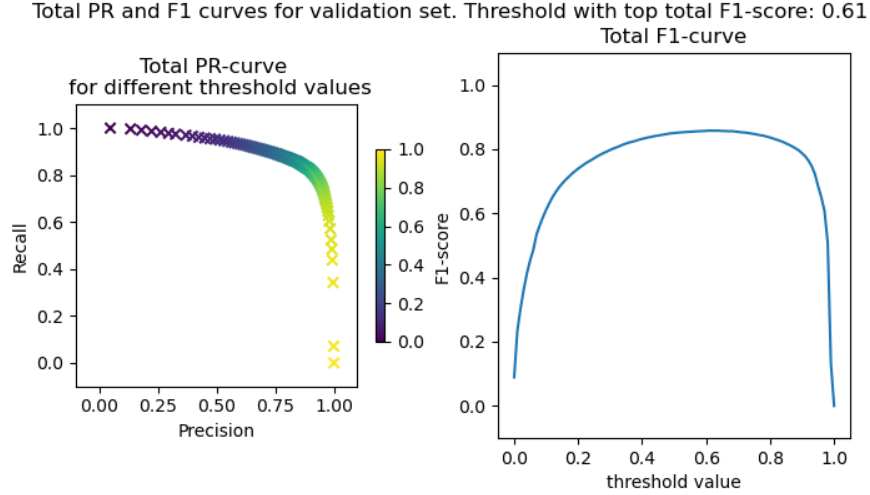


Figure 23: Total PR and F1-curve for validation set, when considering every node/pixel individually.

Precision : 0.937

Recall : 0.846

F1-score : 0.889

To visually inspect the results, parts of the original image can be reconstructed by combining the predictions of the individual samples. The reconstruction of image 115 can be seen in figure 24. The voting map is the reconstructed predictions, gtMap is the reconstructed thin ground truth used as target data for the GNN, gtMap_thick is the reconstructed original image segmentation from the dataset and coverage_map shows where the samples used for the reconstruction are located. The highlighted yellow areas in the voting_map are the model thresholded predictions, where the areas that are more yellow (have a larger value than 1) show where there are predictions from multiple samples. This means that anything with at least 1 vote would be highlighted by the model and the path closely follow the one from the thin ground truth.

Reconstructed Image 115 from samples, for threshold 0.61

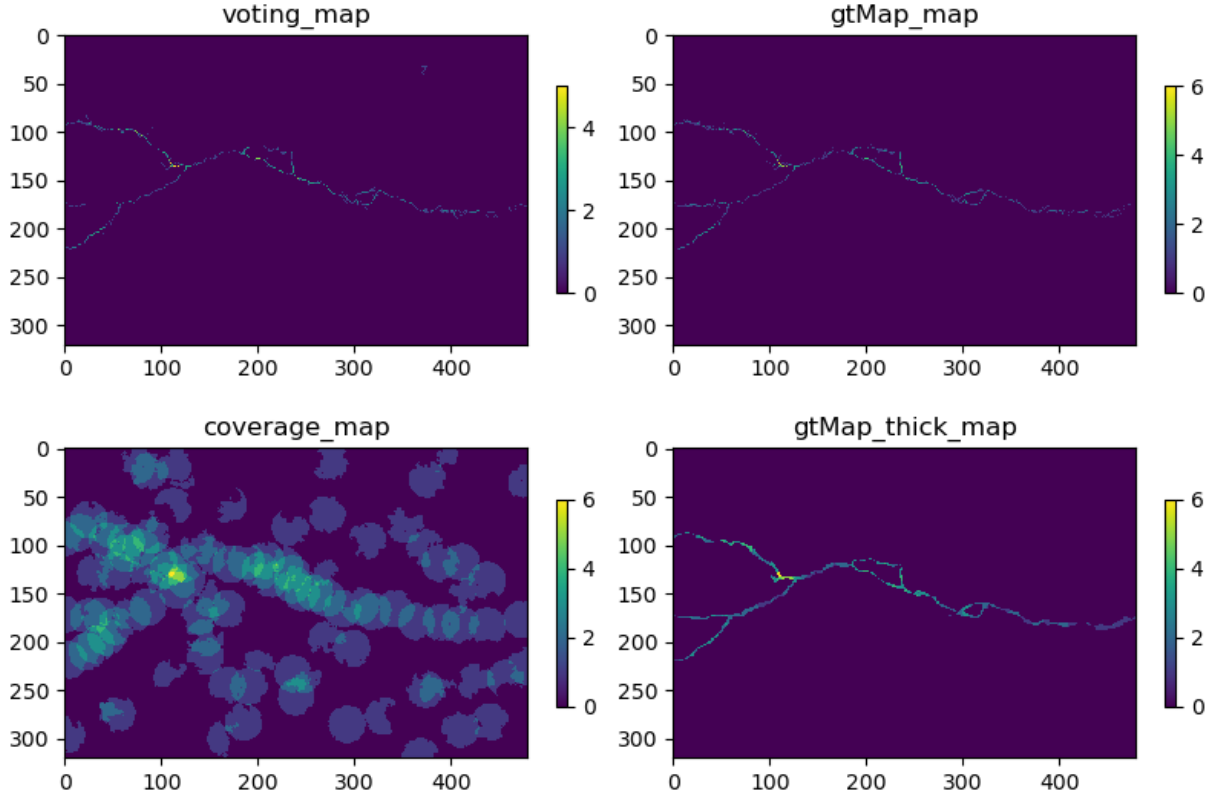


Figure 24: Reconstruction of image 115 from the CrackForest dataset.

6.3 Subproject Conclusion

The GNN is able to highlight the path in the shortest-paths tree graph with good precision and recall. As a next step, this algorithm and GNN should be implemented recursively so that the whole crack in an image can be found by iteratively tracking the crack. This would work by starting at a point of the crack and generating the shortest-paths tree graph at that point of the image. Then the trained model would be used to predict the crack path for that sample and find what pixels of the sample the shortest-paths algorithm should be used next to reveal more of the crack path. This would reveal the whole crack in an image and such a algorithm could be used in for example defectoscopy, or detecting cracks in the CERN tunnels.

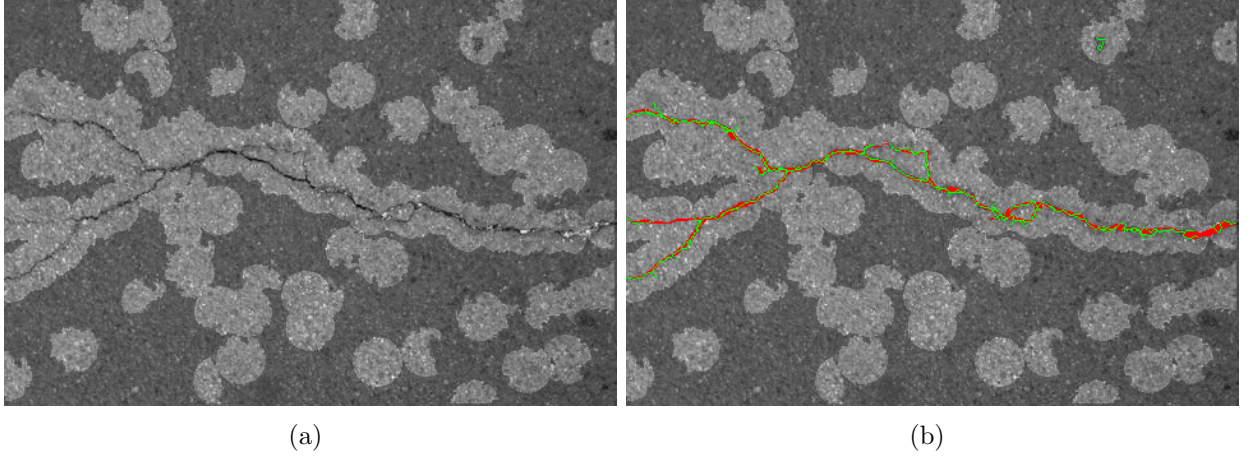


Figure 25: Reconstruction of image 115 overlaid on top of the original PNG image. Subfigure (a) is the original PNG image overlaid with the coverage_map, meaning that the light areas of the image are the ones that have been predicted at least once by the GNN. Subfigure (b) is the original PNG image overlaid with the coverage_map, the original crack segmentation (red) and the predicted voting_map (green). Overlays of the whole testing are available in appendix figure 28.

7 Conclusion

In conclusion, this summer student project consisted of several different parts. First, two introductory subprojects using Graph Neural Networks (GNNs) were implemented to familiarize with the Tensorflow environment, the graph generation, graph batching and the training loop, and they showed great potential for what was to come. After that, a GNN model was used to approximate a simple physics simulation consisting of communicating vessels. Lastly the GNN was trained on crack detection from shortest-paths tree graph generated from images of road cracks. The summer student project was successful in showing that GNN methods have potential to be of use in a wide range of problems that are encountered at CERN.

This summer student project improved my Python programming abilities considerably and introduced me to many new technologies and libraries such as use of ssh, version control via GitLab, how to run training scripts with Sonnet and Tensorflow, training monitoring using Tensorboard and made me efficient at creating Matplotlib plots.

8 Acknowledgements

The biggest possible thank you to my supervisor Dr. Roman Stoklasa for providing me with what seemed like endless support during this project. A big thank you also to Dr. Luigi Serio for arranging this project and for our insightful and encouraging meetings. Thank you to Prof. Karl Meinke and Prof. Mats Wallin at the KTH Royal Institute of Technology in Stockholm for recommending me for the CERN Summer Student programme. Finally I would like to thank my parents for patiently supporting me throughout my work at CERN. This summer student project would not have been possible without the help of all of the aforementioned people.

9 References

- [1] Albert Aillet and Roman Stoklasa. *Summer Student Project Repository*. 2021.
- [2] Peter W. Battaglia, Jessica B. Hamrick and Victor Bapst. *Graph Nets Github repository*. 2018.
- [3] Peter W. Battaglia, Jessica B. Hamrick and Victor Bapst. “Relational inductive biases, deep learning, and graph networks”. In: (2018). arXiv: [1806.01261](https://arxiv.org/abs/1806.01261). URL: <http://arxiv.org/abs/1806.01261>.
- [4] Yang Chen et al. “Curve-Like Structure Extraction Using Minimal Path Propagation With Backtracking”. In: *IEEE Transactions on Image Processing* 25.2 (2016), pp. 988–1003. DOI: [10.1109/TIP.2015.2496279](https://doi.org/10.1109/TIP.2015.2496279).
- [5] Vivek Kaul, Anthony Yezzi and Yichang Tsai. “Detecting Curves with Unknown Endpoints and Arbitrary Topology Using Minimal Paths”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.10 (2012), pp. 1952–1965. DOI: [10.1109/TPAMI.2011.267](https://doi.org/10.1109/TPAMI.2011.267).
- [6] Yong Shi et al. “Automatic road crack detection using random structured forests”. In: *IEEE Transactions on Intelligent Transportation Systems* 17.12 (2016), pp. 3434–3445.

A Appendix

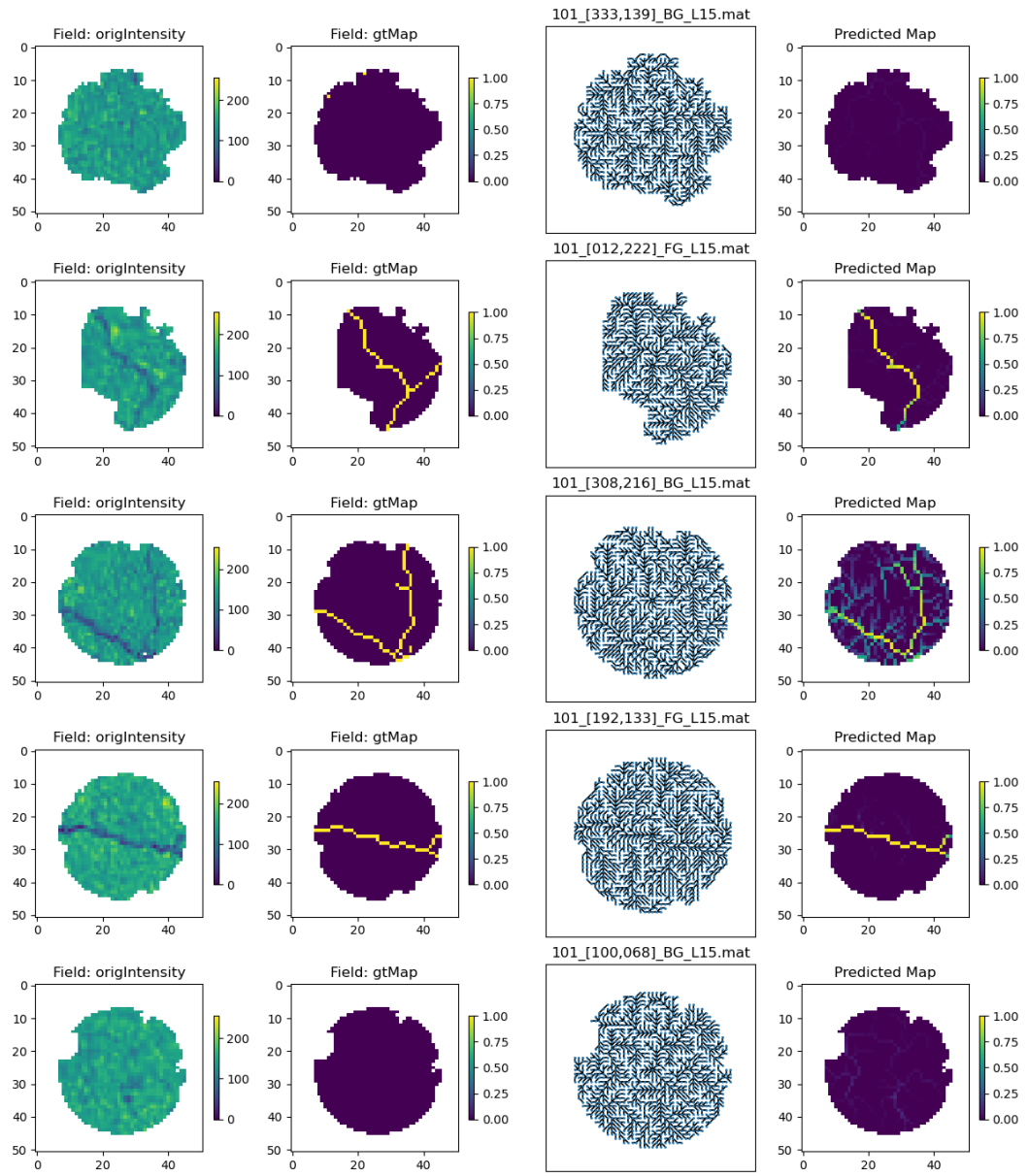


Figure 26: Sample Results from the validation set.

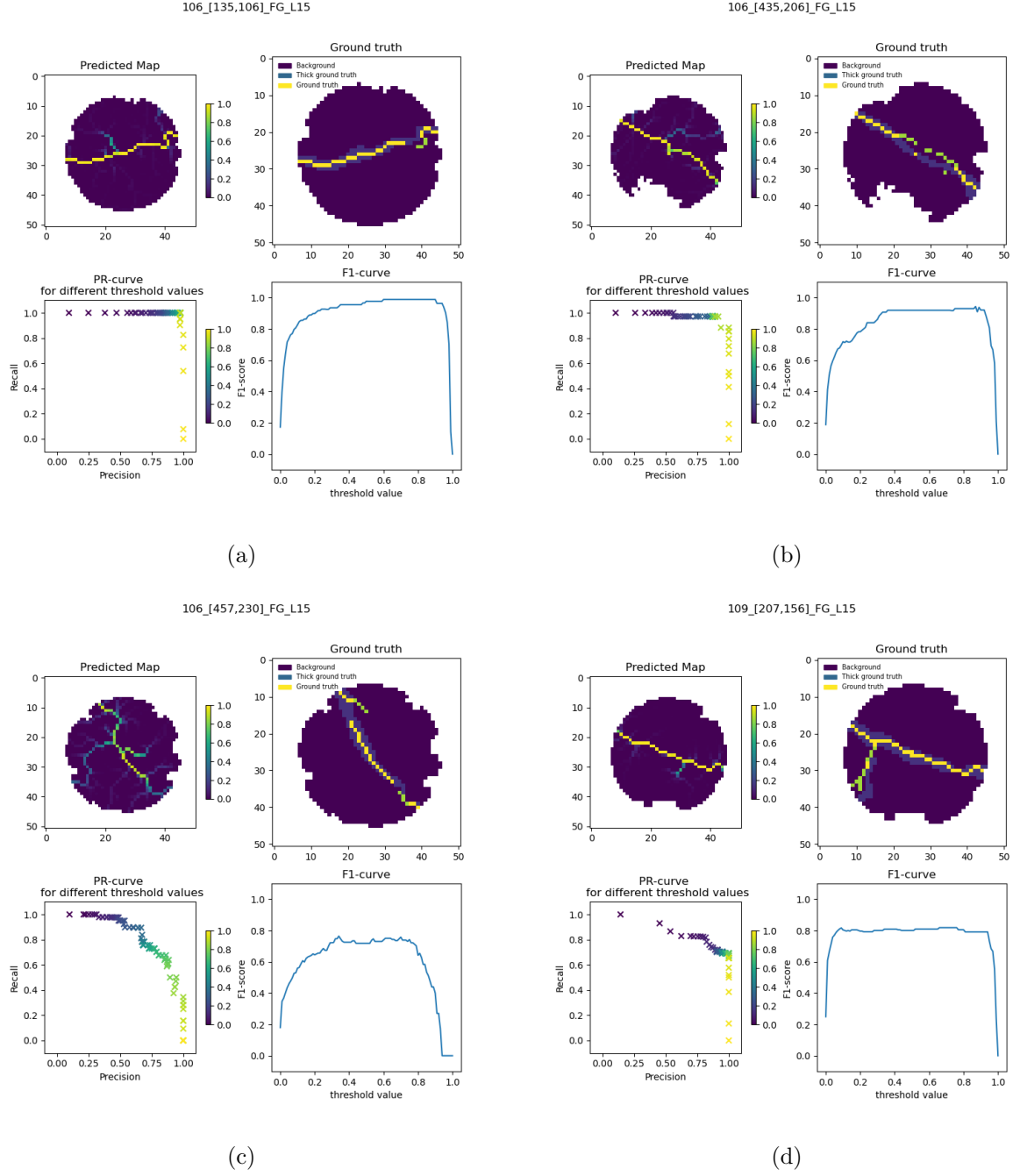
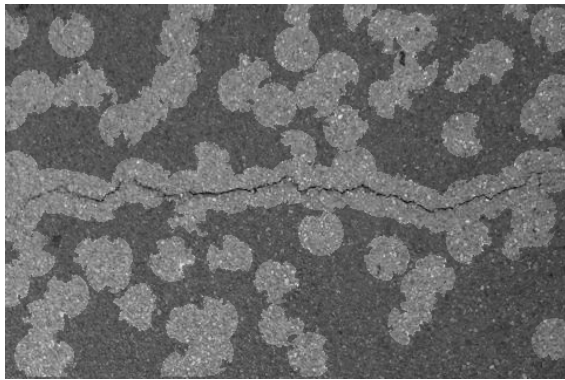
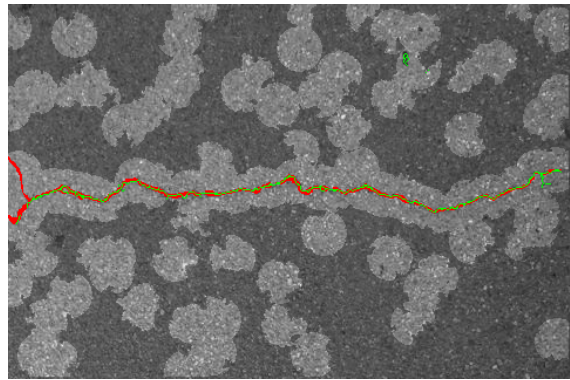


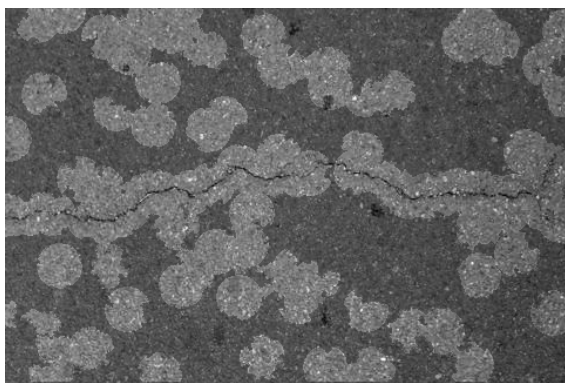
Figure 27: PR-curves and F1-Curves for four representative samples of the validation set.



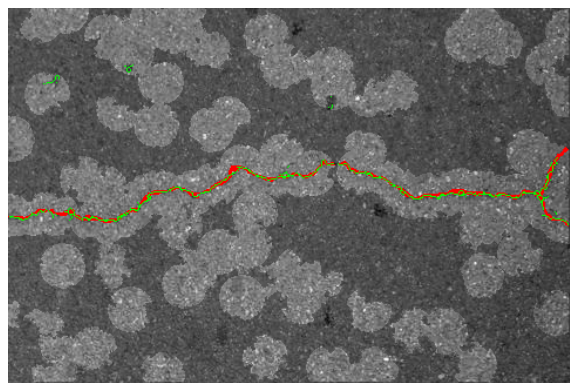
(a)



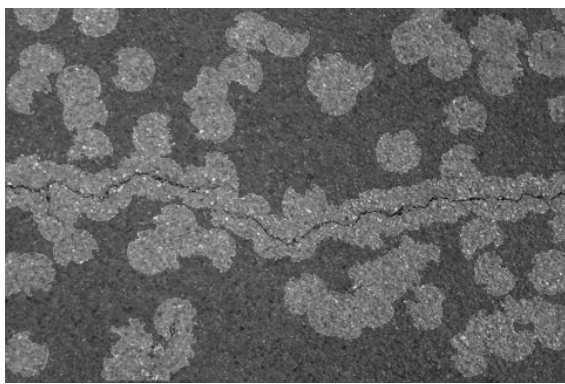
(b)



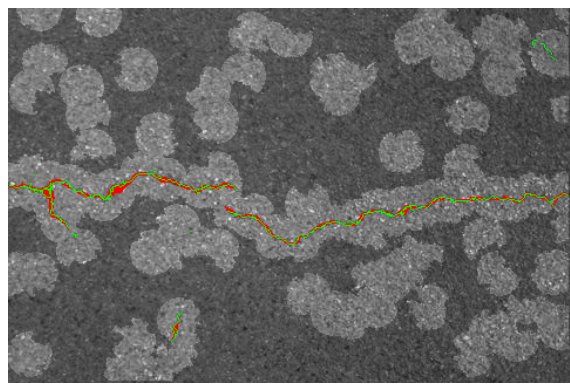
(c)



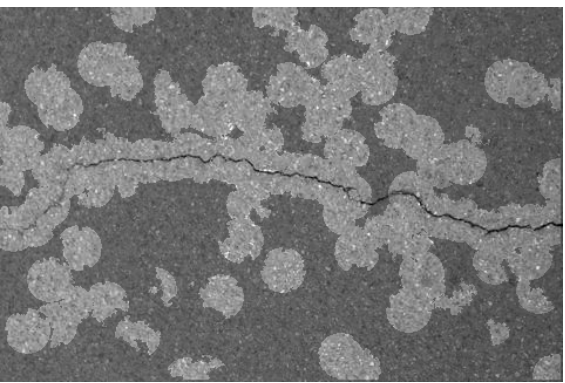
(d)



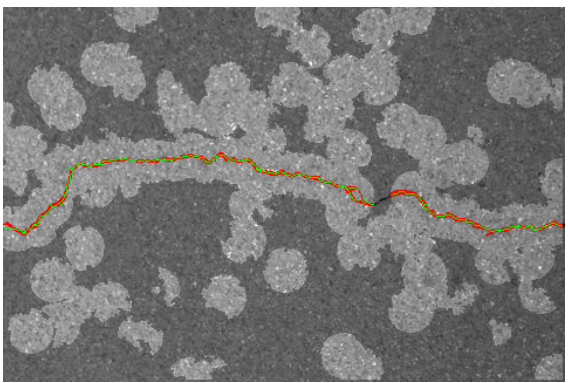
(e)



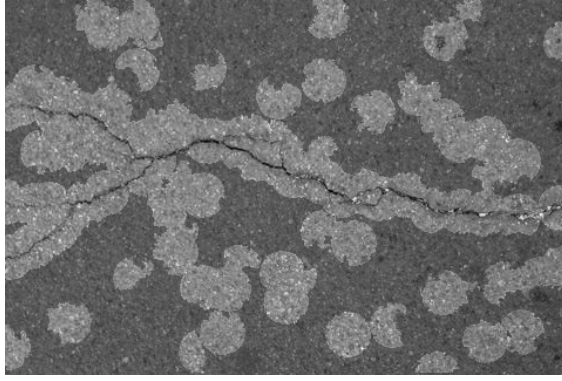
(f)



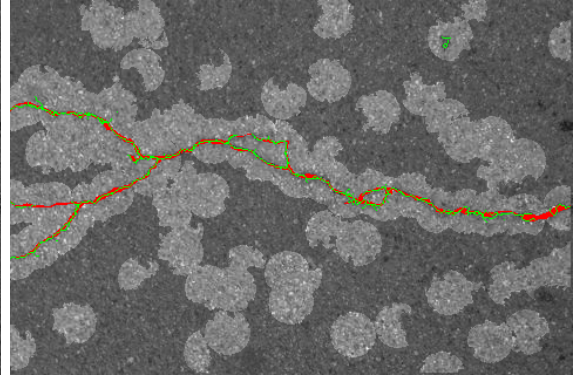
(g)



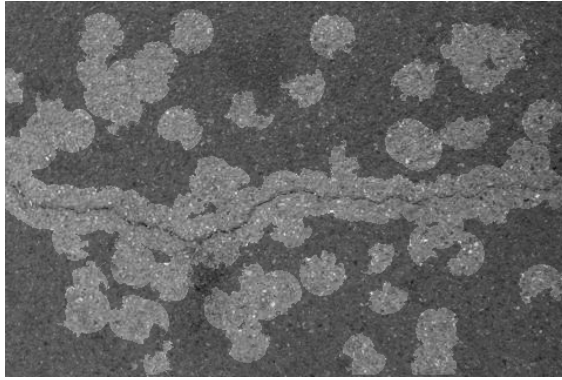
(h)



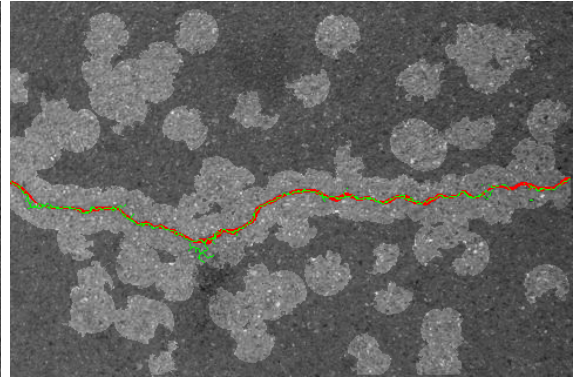
(i)



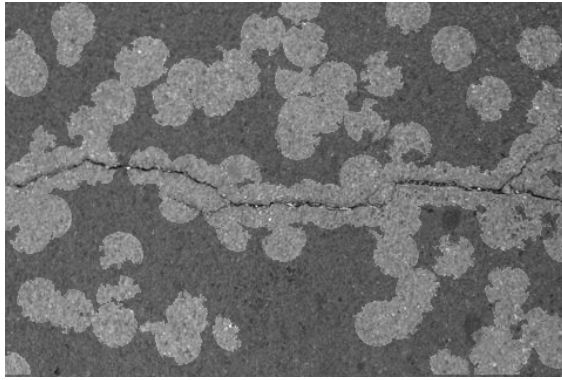
(j)



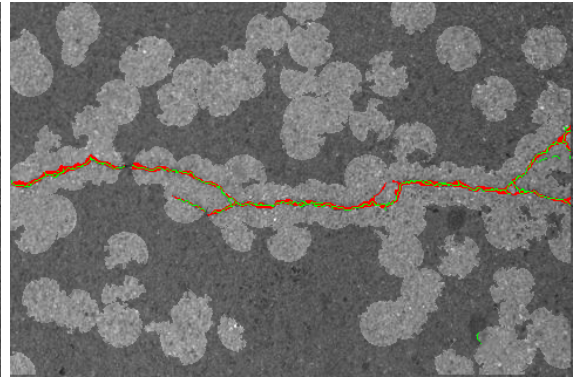
(k)



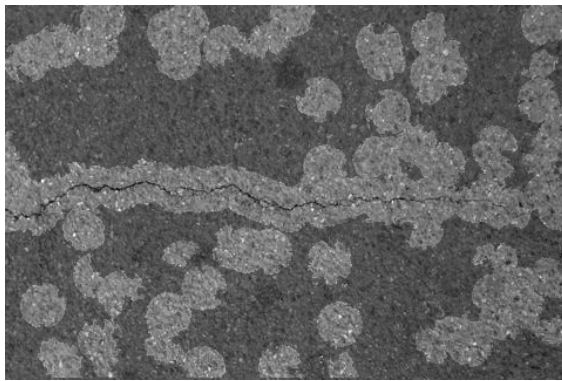
(l)



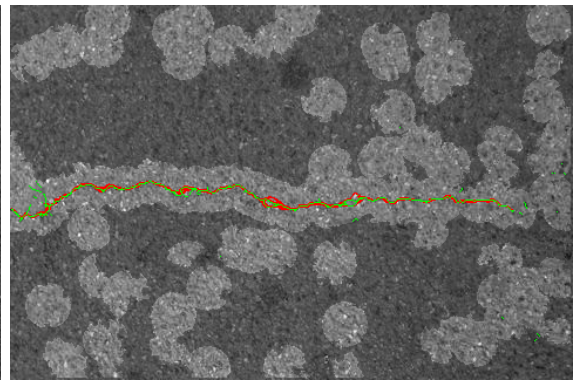
(m)



(n)



(o)



(p)

Figure 28: Original PNG images overlaid with original segmentation map (red), coverage map and the model prediction map of all image in the testing set (111.png to 118.png).