Structure Recognition with Graph Neural Networks

Final Report for the Lab-Course Advanced Projects in Computational Physics 2

Benedikt Wenzel

January 29, 2025

Abstract

The project lies at the intersection of Machine Learning and solid-state physics. A common task in solid-state physics is the classification of atomic structures, such as those found in crystals. Machine Learning, on the other hand, is well-known for its ability to handle classification tasks. Combining these two worlds provides a powerful tool for classifying different crystal structures. As crystal structures are described by the so-called Bravais lattices, which closely relate to graphs, we will need machine learning tools capable of properly handling graph-like data. Fortunately, in the last few years, a new type of neural networks, exactly designed for this kind of data, emerged: Graph Neural Networks. The overall goal of this project is to get familiar with GNNs and apply them to classification tasks of Bravais lattices. Furthermore, the phenomenon of percolation will be explored.

Contents

1 Background

This introductory section lays the theoretical foundation of Bravais lattices and GNNs. We start with a short recap of lattice structures in two and three dimensions and then continue with the most fundamental background of Neural Networks (NNs) and Graph Neural Networks (GNNs). Results on Bravais lattices can be found in many textbooks. Subsection 1.1 follows [10] and [8]. An introduction to NNs and GNNs can be found for example in [5], [11] and in [4] which are also the main reference for section 1.2 and section 1.3.

1.1 Bravais Lattices

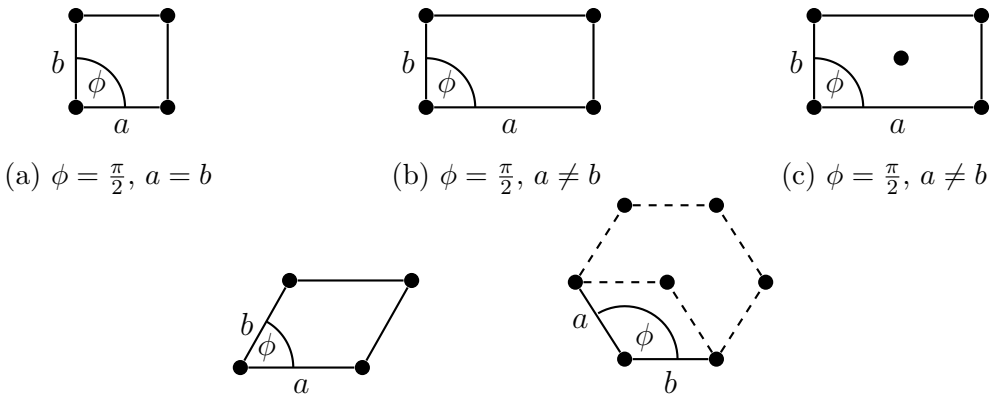
Let $d \in \mathbb{N}$ and $\{b_i\}_{i=1,\dots,d} \subset \mathbb{R}^d$ a basis of $\mathbb{R}^d$. The set

$$\Omega \coloneqq \left\{ \sum_{i=1}^{d} z_i b_i : z_i \in \mathbb{Z} \, \forall i \in \{1, \dots, d\} \right\}$$

is called a d-dimensional lattice. Given any subset $S \subset \mathbb{R}^d$, we define its point group G_S to be

$$G_S \coloneqq \{M \in O(d) : MS = S\} \subset O(d).$$

G_S is obviously a subgroup of $O(d)$. We say that two d-dimensional lattices $\Omega_1, \Omega_2 \subset \mathbb{R}^d$ are of the same Bravais type if there exists $g \in GL_n(\mathbb{Z})$ such that $G_{\Omega_1} = gG_{\Omega_2}g^{-1}$ and $\Omega_1 = g\Omega_2$. Being of the same Bravais type introduces an equivalence relation on the set of all d-dimensional lattices. We refer to the equivalence classes as Bravais classes. A natural question arising is about the total number of Bravais classes. Despite being a very interesting and challenging problem, we will leave this question to the mathematicians. For us, the result is more important than the actual proof. One obtains the following result: For $d = 2$ there are 5 Bravais classes and for $d = 3$ there are 14 Bravais classes. Roughly speaking, they can be distinguished by the relative lengths of the basis vectors b_i and the angles between them. Visualizations of all Bravais classes for $d = 2$ can be found in figure 1. Visualizations for $d = 3$ and further notes on each Bravais class can be found for example in [7].



(a) $\phi = \frac{\pi}{2}$, $a = b$ (b) $\phi = \frac{\pi}{2}$, $a \neq b$ (c) $\phi = \frac{\pi}{2}$, $a \neq b$

(d) $\phi \neq \frac{\pi}{2}$, $a \neq b$ (e) $\phi = \frac{2\pi}{3}$, $a = b$

Figure 1: All different Bravais classes in two dimensions. They are called (a) square, (b) rectangle, (c) centered rectangle, (d) oblique, (e) hexagonal.

1.2 Introduction to Neural Networks

Stating a precise definition of „Neural Network“ is quite involved. Hence, we will only speak about the most fundamental type of neural networks, the so-called feed forward neural networks. Even for these simple networks, a precise definition is rather lengthy. We have to introduce some notation first: Let $l \in \mathbb{N}_{\geq 2}$, $n_0, n_1 \ldots, n_l \in \mathbb{N}$ and for each $i \in \mathbb{N}, i \leq l$ choose an affine linear map $T^{(i)} : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$ and a smooth map $a^{(i)} : \mathbb{R} \to \mathbb{R}$. Denoting with $f^{(i)} : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$ the function acting element wise by $a^{(i)}$ (i.e $f^{(i)}(x_1, \ldots, x_{n_i}) = (a^{(i)}(x_1), \ldots, a^{(i)}(x_{n_i}))$), we define the composition

$$\tilde{F} \coloneqq T^{(l)} \circ f^{(l-1)} \circ T^{(l-1)} \circ \cdots \circ f^{(2)} \circ T^{(2)} \circ f^{(1)} \circ T^{(1)} : \mathbb{R}^{n_0} \to \mathbb{R}^{n_l}.$$

Recall that for any affine linear map $L : \mathbb{R}^n \to \mathbb{R}^m$ there exist $b \in \mathbb{R}^m$ and $W \in \mathrm{Mat}(n \times m, \mathbb{R})$ such that $Lx = W^T x + b$ for all $x \in \mathbb{R}^n$ (the transposition is just for our convenience later on). Hence, to each $T^{(i)}$ corresponds a matrix $W^{(i)} \in \mathrm{Mat}(n_{i-1} \times n_i, \mathbb{R})$ called the weight-matrix and a vector $b^{(i)} \in \mathbb{R}^{n_i}$ called bias such that $T^{(i)}x = \left(W^{(i)}\right)^T x + b^{(i)}$. We introduce further notation: Obviously, we can think of $W^{(i)}$ as an element in $\mathbb{R}^{n_{i-1}n_i}$. Let $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(l)}, b^{(l)}) \in \mathbb{R}^{n_0 n_1} \times \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_{l-1}n_l} \times \mathbb{R}^l$. Clearly, $\tilde{F}$ depends on the choice of $T^{(l)}$ and hence on the choice of θ. Correspondingly, for each such θ we can build a map $\tilde{F} = F_\theta$. With this notation in mind, we are ready to define what a neural network should be: The map

$$F : \mathbb{R}^{n_0 n_1} \times \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_{l-1}n_l} \times \mathbb{R}^{n_l} \to C(\mathbb{R}^{n_0}, \mathbb{R}^{n_l}), \quad \theta \mapsto F_\theta$$

is called (feed forward) neural network with l layers and activation function $a^{(i)}$ in layer i.

Though this definition might seem technical, it bears a visual explanation: It is best explained with figure 2. We can view $F_\theta : \mathbb{R}^{n_0} \to \mathbb{R}^{n_l}$ as a graph-like structure. Each coordinate x_i of an input vector $x \in \mathbb{R}^{n_0}$ can be thought of as a node. Applying $f^{(1)} \circ T^{(1)}$ to x yields a new vector $y^1 = f^{(1)} \circ T^{(1)}(x) \in \mathbb{R}^{n_1}$. The coordinates of y^1 can again be interpreted as nodes. All these nodes (coordinates) together constitute the new layer of the network. Applying $f^{(2)} \circ T^{(2)}$ to y^1 yields a new layer y^2 and so on. The last layer y^l is then given by $F_\theta(x)$. Going from layer $i-1$ to the layer i can be visualized as follows: According to the above definitions we have

$$\begin{aligned} y^i_j &= a^{(i)}\left(T^{(i)}\left(y^{i-1}\right)\right)_j \\ &= a^{(i)}\left(\sum_{k=1}^{n_{i-1}} \left(W^{(i)}\right)^T_{jk} y^{i-1}_k + b^{(i)}_j\right) \\ &= a^{(i)}\left(\sum_{k=1}^{n_{i-1}} W^{(i)}_{kj} y^{i-1}_k + b^{(i)}_j\right). \end{aligned}$$

Up to application of $a^{(i)}$, the node y^i_j is a weighted sum of all nodes y^{i-1}_k in the previous layer (in addition to a constant bias value $b^{(i)}_j$). The matrix element $W^{(i)}_{kj}$ describes how much the node y^i_j is influenced by node y^{i-1}_k. We picture an edge

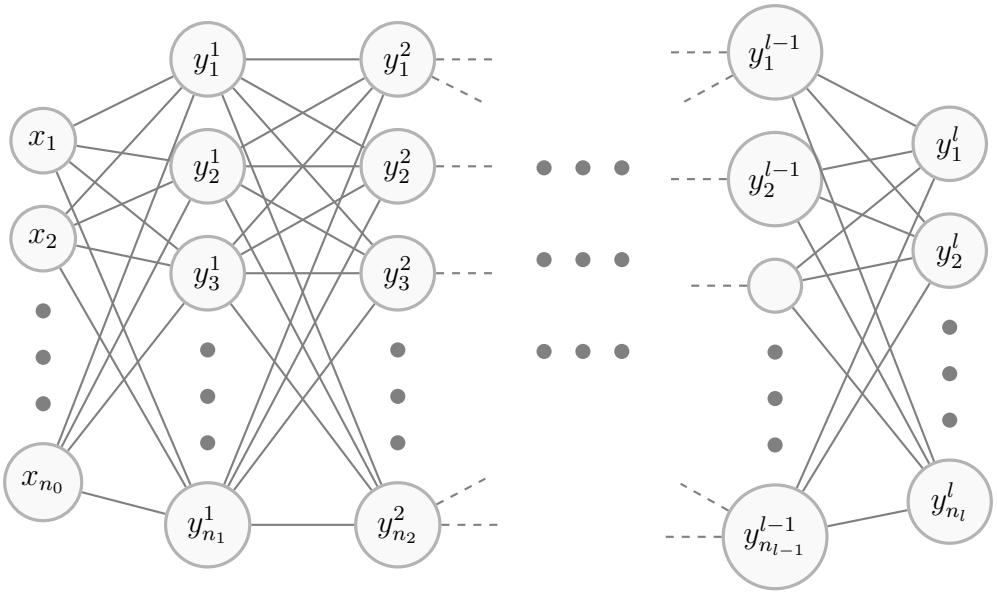


Figure 2: Visualization of a feed forward neural network.

from node y_k^{i-1} to node y_j^i which has the weight $W_{kj}^{(i)}$. Consequently, the matrix $W^{(i)}$ is called weight-matrix.

With the visuals in mind, we can explain what neural networks are used for and how they are being used. The purpose of neural networks lies in the approximation of unknown functions. Suppose we were given a function $G : \mathbb{R}^n \to \mathbb{R}^m$ from which we only know values at D-many ($D \in \mathbb{N}$) points $x_1, \dots, x_D \in \mathbb{R}^n$, i.e. only the values $y_1 = G(x_1), \dots, y_D = G(x_D)$ are known. Choose $l \in \mathbb{N}$, set $n_0 = n$, $n_l = m$ and consider the neural network

$$F : \mathbb{R}^{n_0 n_1} \times \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_{l-1} n_l} \times \mathbb{R}^{n_l} \to C(\mathbb{R}^{n_0}, \mathbb{R}^{n_l}), \quad \theta \mapsto F_\theta.$$

We can then try to find an „optimal“ θ such that $F_\theta \approx G$. To do so, we have to introduce a measure for how much F_θ differs from G. This is commonly called a cost/loss function, i.e. a function $C : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, that maps to a pair $(F_\theta(x_i), y_i)$ a real value $C(F_\theta(x_i), y_i)$ that can be interpreted as a distance between the true value y_i and the predicted value $F_\theta(x_i)$. Finding an optimal θ is achieved by minimizing the map $\theta \to C(F_\theta(x_i), y_i)$ for all $i = 1, \dots, D$. This optimization is called „training the neural network“. There is a whole branch of mathematics about these kinds of optimization of problems. A common way to do this is by means of the gradient descent method. Roughly speaking, the idea is the following: Calculate the gradient of the map $\theta \mapsto C(F_\theta(x_i), y_i)$ for a fixed $i = 1, \dots, D$, and then change θ slightly in the opposite direction of the gradient. How much θ is changed, depends on the so-called learning rate $\eta \in \mathbb{R}_+$. As F_θ is a large composition, computing the gradient basically comes down to repeated application of the chain rule. An efficient algorithm that does exactly this and is widely used is called backpropagation. We can repeat this step until we found a θ such that $C(F\theta(x_i), y_i)$ is sufficiently small for all $i = 1, \dots, d$.

Furthermore, there are many more techniques and tools (like optimizers, batching, ...) to make the training more efficient, faster converging and less time-consuming. However, this section was intended to give only a rough overview.

Hence, we will not go into more detail. For additional information, there is a vast amount of literature on these topics, see for example [5], [11] and [1].

1.3 Fundamentals of Graphs and Graph Neural Networks

To talk about graphs, we first have to agree on some notation: Let V be a set and $E \subset V \times V$. The tuple $G = (V, E)$ is called a graph. Furthermore, we call an element $x \in V$ a node and a tuple $(x, y) \in E$ a directed edge from x to y. Suppose there is a directed edge from x to y, then we say that x is a neighbor of y. In case $(x, y) \in E$ implies $(y, x) \in E$, we speak of an undirected graph. In that case, we can think of elements in E as unordered tuples $\{x, y\}$ instead of ordered ones. We still call $\{x, y\}$ an edge between x and y. For $y \in V$ we define the neighborhood N_y of y to be the set of all neighbors, i.e.

$$N_y \coloneqq \{x \in V : (x, y) \in E\} \subset V. \tag{1}$$

Furthermore, we assign to each node $x \in V$ a vector $v_x \in \mathbb{R}^n$ called node feature and to each edge $(x, y) \in E$ a vector $e_{x,y} \in \mathbb{R}^m$ called edge feature. A graph together with all its node and edge features is called attributed graph.

Roughly, a GNN takes an attributed graph as an input and manipulates the edge and node features in each step. More than that, a GNN can transform the structure of the graph itself, e.g. by introducing new nodes or edges. However, we will not go into detail about this possibility and stick to the simpler case of manipulating only node and edge-features. Furthermore, we restrict ourselves to the case where the GNN does not alter the edge features. Let us make these ideas a bit more rigorous (see figure 1.3 for an illustration): Let (V, E) be a graph with node features $\{v_x \in \mathbb{R}^n : x \in V\}$ and edge features $\{e_{x,y} \in \mathbb{R}^m : (x, y) \in E\}$. For each $c \in V$ a new node feature $\widehat{v_c}$ is calculated according to the following rule

$$\widehat{v_c} = \phi_{upd}\left(v_c, \phi_{agg}\left(\{\phi_{mes}\left(v_y, v_c, e_{y,c}\right) : y \in N_c\}\right)\right), \tag{2}$$

where $\phi_{upd}, \phi_{agg}, \phi_{mes}$ denote differentiable functions. These three functions are commonly interpreted as follows: ϕ_{mes} takes as inputs the node value v_c and the node value v_y of one neighbor y of c as well as the value $e_{y,c}$ of the edge (y, c). Depending on these inputs, it then computes a value, which can be thought of as a message originating from node y which is sent to node c. ϕ_{agg} collects all these messages to node c and aggregates them in some way, so that the output can be thought of as one overall message to node c. ϕ_{upd} takes this overall message as well as the value of node c and computes how the value of node c is altered. Unsurprisingly, this scheme is called message passing layer. Given a specific problem that is required to be solved by a GNN, the challenge is to make appropriate choices for $\phi_{upd}, \phi_{agg}, \phi_{mes}$ that suit the problem at hand. Furthermore, one has to decide how many iterations of the above procedure are suitable.

1.4 Percolation

Besides Bravais classes, neural networks and graph neural networks, the fourth ingredient for this report is the notion of percolating graphs. We start with an undirected graph $G = (V, E)$ and want to define what paths and cycles are. Pick

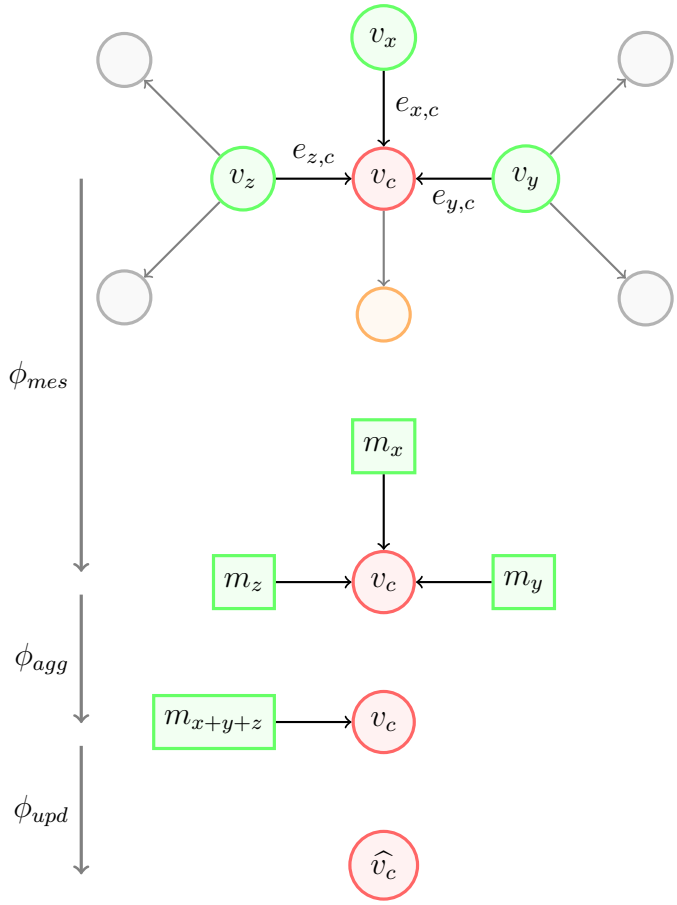


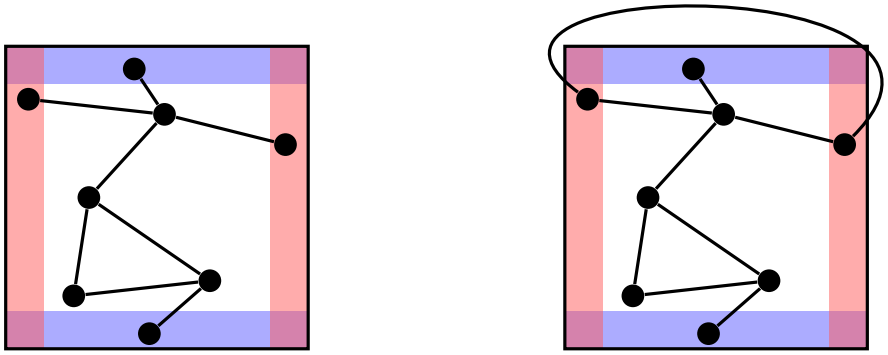
Figure 3: Illustration of the message passing procedure according to equation (2). The neighbors of node c (red) are the nodes x, y, z (green). Attention: According to equation (1) the orange node is not regarded as a neighbor of c as the edge points in the wrong direction. For each neighbor (x, y, z), ϕ_{mes} computes messages (m_x, m_y, m_z) which are sent to c. Then, ϕ_{agg} aggregates these three messages and outputs one overall message m_{x+y+z} that is sent to c. In the last step, ϕ_{upd} updates the node value v_c to a new value $\widehat{v_c}$.

two nodes $n_1, n_2 \in V$. We say that n_1 and n_2 are connected if there are nodes $m_0, m_1, \ldots, m_N \in V$ such that $m_0 = n_1$, $m_N = n_2$ and for each $i \in \{0, \ldots, N-1\}$ there is an edge $\{m_i, m_{i+1}\} \in E$. In this case, the tuple $(m_0, m_1, ..., m_N)$ is called a path of length N from n_1 to n_2. A path $(m_0, m_1, ..., m_N)$ is called a cycle if $m_0 = m_N$ and $m_i \neq m_j$ for $i \neq j, 0 \leq i, j < N$ (i.e. no node besides the first one is visited twice).

Knowing what paths and cycles are, everything is set up to define what a percolating graph is. Suppose $G = (V, E)$ is an undirected graph, where each node $n \in V$ has a position $(n_x, n_y) \in [0, 1] \times [0, 1]$. Visually, we think of G as a graph inside the unit square. Fix $0 < r < \frac{1}{2}$. The graph G is called percolating, if there are nodes $n, m \in V$ such that

1. there is a cycle containing n and m,
2. n and m are connected by an edge, i.e. $\{n, m\} \in E$,
3. either $n_x < r$ and $m_x > 1 - r$ or $n_y < r$ and $m_y > 1 - r$.

Figure 4 gives an example of percolating and non-percolating graphs. Properties 1. and 2. are easily understood. It is worth mentioning, that 1. and 2. imply, that there is a cycle containing n,m as well as the edge $\{n, m\}$. Property 3. needs a bit more explanation: We can picture small strips of width r at the sides of the unit square (see figure 4 where the strips are colored red and blue). Requiring $n_x < r$ and $m_x > 1 - r$ means that n is located in the strip on the left side of the unit square while m is located in the strip on the right side. Accordingly, $n_y < r$



(a) non-percolating graph (b) percolating graph

Figure 4: Illustration of percolating and non-percolating graphs.

and $m_y > 1 - r$ requires n to be on the bottom and m to be on the top of the unit square. In any of these two cases, n and m are in strips of the same color but on opposite sides. The restriction to $r < \frac{1}{2}$ guarantees that strips of the same color do not overlap.

2 Goals and Implementation

The project can be split into two parts. The goal of the first part is to build a GNN that is capable of assigning a 2- or 3-dimensional lattice its Bravais class. For this instance, different lattices need to be created programmatically. This will be covered in the first part of this chapter. The second part describes the actual implementation of the GNN. In the second part of the project, we will take a look at more complex structures. The goal is to distinguish between percolating and non-percolating graphs.

2.1 Lattice Creation

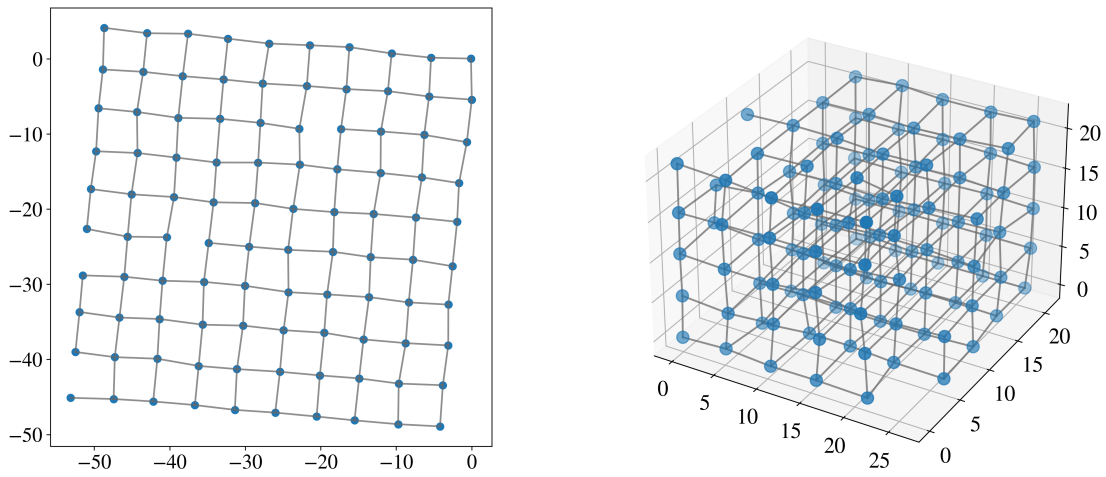
To be precise, we will not create lattices according to the definition given in section 1.1, as this would require creating sets with infinitely many elements. Clearly, a GNN can only deal with finitely many nodes. Hence, we will only create finite subsets of lattice but, for sake of simplicity, we will still call them lattices.

Let us start with the creation of two-dimensional lattices. Choose $a, b \in \mathbb{R}_+$, $\phi \in (0, \pi)$ and set $e_x = (0, a)^T$ as well as $e_y = (b\cos(\phi), b\sin(\phi))^T$ (see figure 1 for a visualization of a, b and θ). Then, the set $\tilde{\Omega} = \mathbb{Z}e_x \times \mathbb{Z}e_y \subset \mathbb{R}^2$ is a two-dimensional lattice. As mentioned, we only work with finite subsets of $\tilde{\Omega}$. Hence, we choose $N_x, N_y \in \mathbb{N}$ and set $\Omega = \mathbb{N}_{\leq N_x}e_x \times \mathbb{N}_{\leq N_y}e_y \subset \tilde{\Omega}$. Now, this lattice has N_xN_y elements. Next, we add some noise to the elements in Ω and restrict their coordinates to a certain range. For this instance, let $\mu, \sigma, s \in \mathbb{R}_+$. Firstly, draw random samples from a normal distribution with mean value μ and standard deviation σ. Secondly, we scale these random numbers by the factor s and add the scaled noise to all elements in Ω (coordinate-wise, i.e. add noise to the first coordinate of the elements in Ω as well as to the second coordinate). Restricting all coordinates to a certain range is easily done. Let $x_{max}, y_{max} \in \mathbb{R}_+$ and replace each $(x, y) \in \Omega$ with $(x \mod x_{max}, y \mod y_{max})$ Next, we want to turn our lattice Ω into a graph (V, E). Obviously, we can set $V = \Omega$ and what remains is the choice of edges. For this, we take $r \in \mathbb{R}_+$ and set

$$E = \{(v, w) \in V \times V : \|v - w\| < r\},$$

where $\|\cdot\|$ denotes the standard norm in R^d, i.e. nodes that are close together will be connected. As $\|v - w\| = \|w - v\|$, the graph (V, E) is undirected. Lastly, we choose $p_n, p_e \in [0, 1]$ and randomly delete nodes with probability p_n (and edges with probability p_e respectively). This step finishes the creation of 2d lattices with artificial artifacts. The creation of three-dimensional lattices goes analogously. The only difference is that we have to pick three basis vectors e_x, e_y, e_z which have length $a, b, c \in \mathbb{R}_+$ and enclose angles $\phi = \angle(e_x, e_y)$, $\psi = \angle(e_x, e_z)$, $\chi = \angle(e_y, e_z)$. As an example, figure 5 shows two different lattices created by this scheme.

According to the above procedure, we are able to generate training and test datasets for our GNN, both in 2d and in 3d. We start with a description of the 2d-dataset. We created graphs with 100 nodes ($N_x = N_y = 10$). Recall that depending on the specific values of a, b and θ, different graphs for different Bravais classes can be generated (take again a look at figure 1 as a reminder, which values for a, b and θ result in which Bravais class). Furthermore, recall that some Bravais classes require $\theta = \frac{\pi}{2}$. In order to introduce further artificial



(a) 2d (b) 3d

Figure 5: Examples of lattices created by the procedure described in section 2.1.

noise, we allow for values in the range $(0.99\frac{\pi}{2}, 1.01\frac{\pi}{2})$. Accordingly, when the side lengths are required to fulfill $a = b$, we allow for deviations of 1%, i.e. such that $0.99 \leq \frac{a}{b} \leq 1.01$. Furthermore, we set the minimal side length to be 0.1 and the maximal side length to 10. The parameters σ, μ and s determining the above-mentioned Gaussian noise were chosen to be $\sigma = 0.5$ and $\mu = 0$. The amplitude s of the noise was taken to be $s = 0.07a$ in x-direction (depending on the side-length a) as well as $s = 0.07b$ in y-direction. Next, we restricted all coordinates to the range $(x_{max}, y_{max}) = (N_x a, N_y b) = (10a, 10b)$. The probabilities p_n, p_e for randomly dropping nodes/edges were taken to be $p_n = p_e = 0.01$. According to these settings, we created 10000 graphs per Bravais class. Since there are 5 Bravais classes in total, the 2d-dataset consists of 50000 graphs, 20% were taken to be the test set and the remaining 80% constitute the training set.

Creating the 3d-dataset completely goes the same way. The noise, the maximal side-lengths as well as the probabilities for removing nodes and edges were chosen identically. In order to obtain roughly the same number of nodes per graph as in the 2d case, we chose $N_x = N_y = N_z = 5$, which leads to 125 nodes per graph. The only difference between the 2d- and the 3d-dataset lies in the number of graphs per Bravais class. We were aiming for the same total number of graphs (50000), but, in contrast to the 2d case, we now have 14 Bravais classes. This leads to 3500 graphs per Bravais class.

2.2 Implementing the GNN

Once we have a sufficient amount of training data at hand, we need to determine the optimal structure of the GNN as well as a training procedure, that suits the problem of classifying graphs into Bravais classes. This amounts to finding the right functions $\phi_{upd}, \phi_{mes}, \phi_{agg}$ (cf. equation 2) and a proper number of message passing layers. More than that, one has to decide, which edge/node features do best, as well as which hyperparameters and loss function to use during training.

As we do not have extensive computational resources available, we were unable to vary all of these parameters. We fixed the hyperparameters, the loss function

as well as the node/edge features as follows: Both in the 2d and in the 3d case the edge n, m between nodes n and m carries the difference of the positions of nodes n and m as edge feature $e_{n,m}$. The nodes n and m do not carry any specific node feature. Each node got the number 1 assigned as a feature, i.e. there are basically no node features. For training the GNN, the standard NAdam optimizer with all its standard settings implemented in the PyTorch library was used. In particular, the standard setting for the learning rate is $\eta = 0.002$. Each training consisted of 30 epochs. The training datasets were split in batches of size 32. Since we are interested in classification tasks, we one-hot encoded the Bravais classes and used cross entropy loss as our loss function.

Furthermore, in all the following experiments, we fixed ϕ_{agg} to be the function that sums up all its inputs. All other parameters mentioned at the beginning of this section were varied in the following way: The function ϕ_{mes} was implemented as a general feed forward neural network (cf. section 1.2) with depth d_m and uniform width w_m. By depth, we mean the number of hidden layers and by uniform width we mean the number of nodes in each hidden layer, which was chosen to be uniform over all hidden layers. Likewise, the function ϕ_{upd} was taken to be a general feed forward neural network width depth d_u and uniform width w_u. In principle, as mentioned above, there is a fifth parameter that needs to be optimized, namely the number d_G of message passing layers. However, five parameters are computationally too expensive to handle. Therefore, all further experiments were conducted with $d_G = 2$. Via grid search, we looked through all possible combinations of d_m, w_m, d_u, w_u in the following ranges

$$d_m \in \{1, 2, 3\}, \tag{3}$$
$$w_m \in \{10, 20, 30\}, \tag{4}$$
$$d_u \in \{1, 2\}, \tag{5}$$
$$w_u \in \{5, 10\}. \tag{6}$$

In total, this amounts to finding an optimum within 36 parameters. Once we found an optimum on the 2d dataset, we used these optimal settings and trained on the 3d dataset. There are to question we are aiming to answer: First, can we find any correlations between the widths and depths of our GNN and the training accuracy. Second, whether the settings that did best in the 2d case also lead to good results in the 3d case.

A few words on the actual implementation: The implementation is based on PyTorch and PyTorch Geometric (abbreviated PyG), see [9] and [2]. Luckily, one does not have to implement the whole message passing scheme. Instead, the PyG-package is equipped with a base class called MessagePassing. This class is build in a way such that the function $\phi_{upd}, \phi_{mes}, \phi_{agg}$ can be freely implemented and everything else works under the hood. Hence, it is sufficient to state how these three functions were implemented. It is not necessary to go into detail about the implementation of the whole message passing scheme.

2.3 Percolation and Top-K Pooling

For classification of graphs into percolating and non-percolating ones, we were asked to experiment with a mixture of the GNN described in the previous section

(which layers are message passing layers) and the so called Top-K Pooling layer. How exactly this mixture looks like depends on the experiment we did and will be explained once we come the specific experiments. What follows is a rough overview over the working principles of the Top-K Pooling layer. For additional information see [3], where this layer was originally proposed. The following explanation of the pooling procedure might become more clear with an illustration at hand. See figure 6. Suppose $G = (V, E)$ is an attributed graph. Recall from section 1.3, that

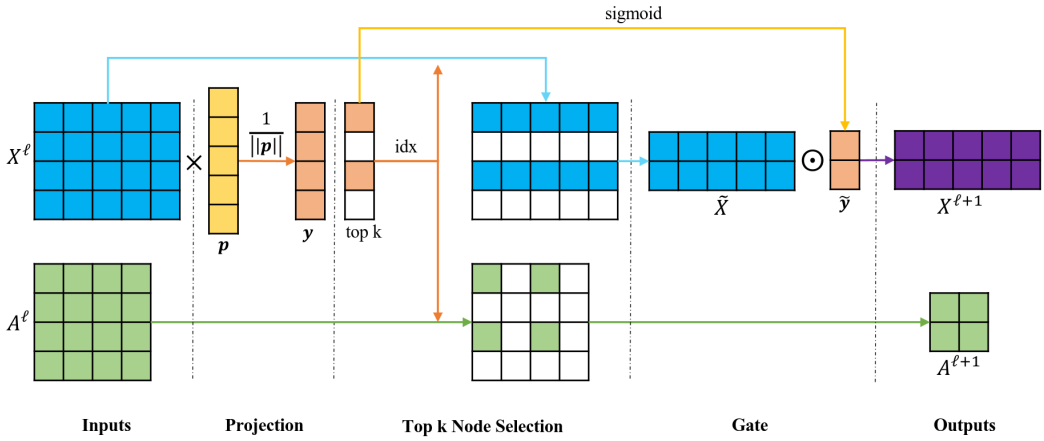


Figure 6: Illustration of the Top-K Pooling layer. See section 2.3 for an explanation. Taken from [3].

each node $x \in V$ has a node feature $v_x \in \mathbb{R}^n$. We can organize all node features in a matrix $X^l \in \mathrm{Mat}(|V| \times n, \mathbb{R})$ given by

$$\left(X^l\right)_{x,j} = \left(v_x\right)_j, \quad x \in V, 1 \leq j \leq n. \tag{7}$$

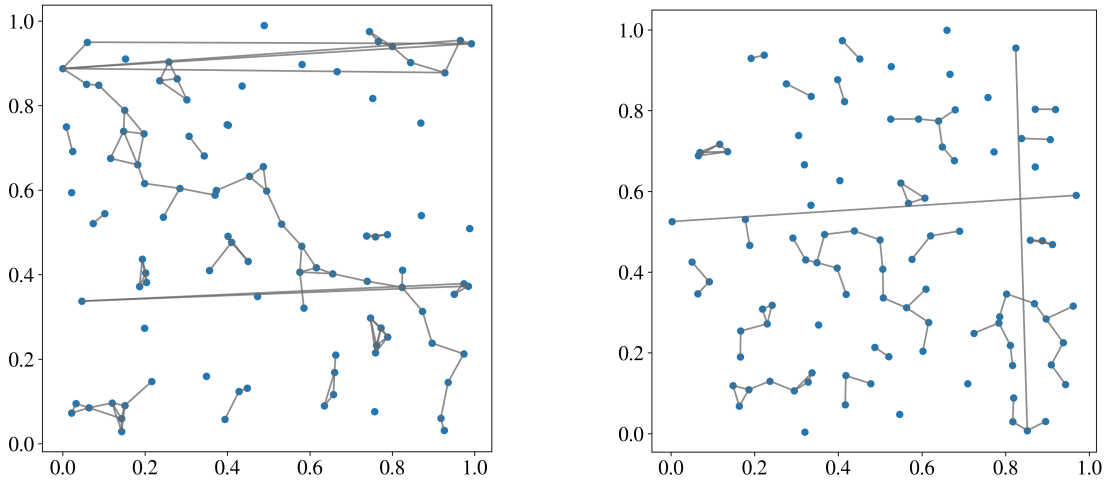
Furthermore, we can define the so-called adjacency matrix $A^l \in \mathrm{Mat}(|V| \times |V|, \mathbb{R})$ to be

$$\left(A^l\right)_{x,y} = \begin{cases} 1 & \text{if } (x,y) \in E, \\ 0 & \text{if } (x,y) \notin E \end{cases}, \quad x, y \in V. \tag{8}$$

For $p \in \mathbb{R}^n$, called the projection vector, we define $y = \frac{X^l p}{\|p\|}$. Next, choose $k < |N|$ and select the indices of the k highest entries in y and denote the resulting list of indices as $idx \in \mathbb{N}^k$. To each index i in idx, there is a corresponding node $x_i \in V$. Therefore, we can equivalently think of idx as a subset $\tilde{V} \subset V$. These are the nodes that will not be deleted during the pooling process. All nodes in $V \backslash \tilde{V}$ will be deleted. The deletion is done as follows: Define $\tilde{X} \in \mathrm{Mat}(k \times n, \mathbb{R})$ to be the matrix consisting of all node feature v_x for $x \in \tilde{V}$, analogues to equation 7. Likewise, define $A^{l+1} \in \mathrm{Mat}(k \times k, \mathbb{R})$ analogously to equation 8 to be the adjacency matrix corresponding to the nodes in $\tilde{V}$. Next, compute the elementwise product of $\tilde{y}$ and $\tilde{X}$ which leads to a new matrix $X^{l+1} \in \mathrm{Mat}(k \times n, \mathbb{R})$. This step is called „gate operation“. The new node feature matrix X^{l+1} together with the new adjacency matrix A^{l+1} define a new graph G^{l+1} which has $k < |V|$ nodes. Effectively, we have reduced the original graph G with $|V|$ nodes to a smaller graph. Depending on the choice of the projection vector p, we can achieve different output graphs G^{l+1}. Given a specific problem, the goal of the Top-K Pooling layer is to learn a suitable projection vector p. The gate operation step ensures, that the projection

vector is indeed learnable via standard backpropagation. The precise argument why the gate operation is necessary for learning p is a bit technical, we refer to [3]. Again, one do not have to implement this layer from scratch, as the PyG library already provides an implementation, where one just have to choose the desired k.

Obviously, the dataset created in section 2.1 is not appropriate for the percolation problem. Instead, the dataset for the percolation problem was created by Jonas Buba and his colleges. The dataset consists in total of 1614 planar graphs, 822 of which are not percolating, whereas the remaining 792 are percolating. See figure 7 for examples of graphs that are in the dataset. The whole set was again partitioned in a training set (80% of the total number of graphs) and a test set (20%). The nodes have their position in the unit cube as attributes, whereas the edges are attributed with their length.



(a) Percolating (b) Non-Percolating

Figure 7: Example of graphs that are in the dataset used for the percolation problem (here $r = 0.08$, see again section 1.4 for an interpretation of r).

3 Results and Discussion

This chapter presents the results of the problems mentioned in chapter 2. Furthermore, we are going to analyze these results. As in chapter 2, we start by looking at the classification tasks in 2 and 3 dimensions and after that, we will proceed with the percolation problem.

3.1 Classifying Graphs into Bravais Classes

At first, we start by analyzing the results of the training on the 2d-dataset and then continue with the 3d case.

Results for training on the 2d dataset As mentioned in section 2.2 we tested in total 36 different models, each with a unique combination of values d_m, w_m, d_u, w_u. For better statistics, each model was trained five times in a row. All training curves shown below are the averages of these five individual trainings. Before going into detail which combination led to the best results, we start with a more high-level overview:

The average test loss and test accuracy over all 36 combinations can be found in figure 8. Achieving an accuracy of approximately 90%, the problem can be

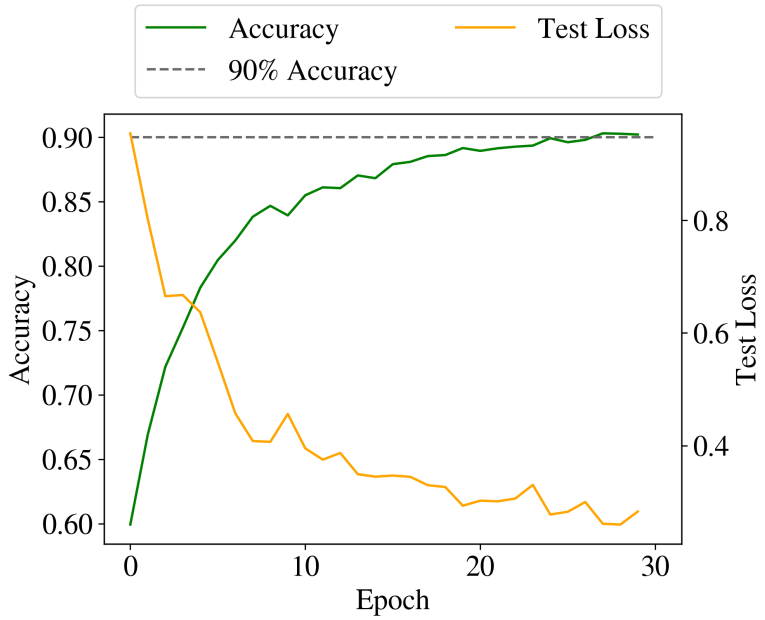
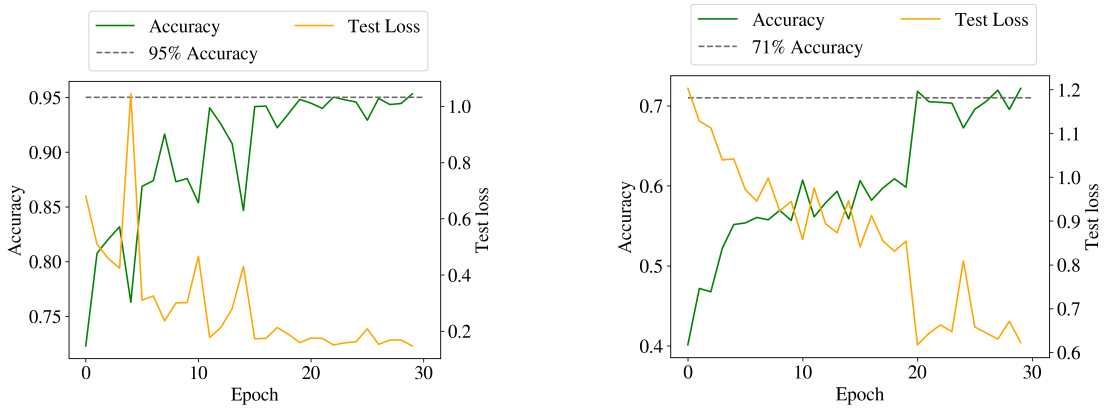


Figure 8: Averaged performance over all 36 different models.

considered solved. However, there are non-negligible difference between different models. The model that performed best achieved an accuracy of almost 95%, whereas the model that performed worst only managed to get about 70% accuracy. The training process of both of these models are depicted in figure 9. Hence, different values for d_m, w_m, d_u, w_u may lead to drastically different outcomes.

Now that we have a rough overview, we are going to take a closer look at possible correlations between different combinations and their resulting performance. A problem arising is that we can only plot two-dimensional data, but we are varying 4 parameters. To overcome this issue, we considered the following type of



(a) Best performing model ($(d_m, w_m, d_u, w_u) = (2, 30, 1, 10)$).

(b) Worst performing model ($(d_m, w_m, d_u, w_u) = (3, 30, 2, 5)$)

Figure 9: Performance of models with the highest, respectively lowest, accuracy.

diagram: Each axis corresponds to one out of the four parameters d_m, w_m, d_u, w_u. Since there are only two axes, there are still 2 parameters left that need to be accounted for. To see, how we dealt with that problem, take figure 10 as an example. The x-axis takes care of the parameter w_m, the y-axis shows w_u. To each specific combination (w_m, w_u) there are six possible combinations of the remaining parameters (d_m, d_u), namely $(d_m, d_u) \in \{(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)\}$ (recall all possible settings from equations 3-6). Each of these six combinations is shown as an individual point inside a rectangle centered at the location (w_m, w_u). The points are then colored according to the test accuracy of the model they represent. Note that only points with an accuracy above 80% are shown. Based on figure 10 we can draw conclusions about the impact of the parameters w_m, w_u on the accuracy of the model. Obviously, when $(w_m, w_u) = (10, 5)$ (lower left rectangle), the performance was always below 90%, no matter the choice of d_m and d_u. We can conclude, that $(w_m, w_u) = (10, 5)$ is the worst choice for the problem at hand. On the contrary, the setting $(w_m, w_u) = (30, 10)$ seems to lead to the best performance. The four other settings (i.e. the four remaining rectangles in figure 10) can not be distinguished. We interpret this as follows: For the accuracy it does not matter if w_u is low and w_m is high, or if w_u is high and w_m is low. Hence, one can state a reasonable hypothesis: Settings, for which the product $w_m w_u$ is relatively low, lead to bad performing models. In opposition to that, relatively high products $w_m w_u$ lead to well performing models. However, this can only be seen as an unproven proposition. Much more statistics and data is needed to really substantiate this claim.

Next, we want to investigate the relation between the width w_u and the depth d_u. To this end, we consider the same type of visualization as in figure 10. The result can be found in figure 11. Keep in mind, that figure 11 does not provide any additional information that figure 10 would not carry as well. It is just a matter of visualization, that makes finding patterns easier for us. First, recall that only points with an accuracy above 80% are shown. To put it differently, missing nodes correspond to settings leading to very bad performing models. Hence, the choice $(d_u, w_u) = (2, 1)$ (lower right rectangle) leads to overall low accuracies, as two points inside the rectangle are missing and the present points are mostly below 90%

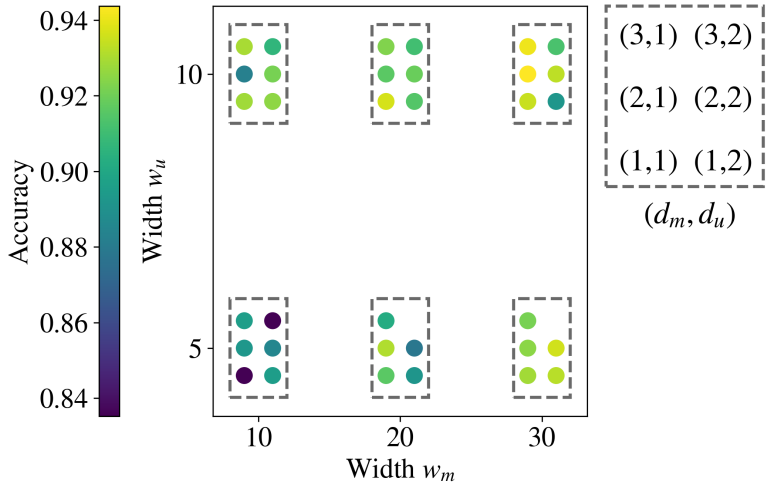


Figure 10: Correlation between the width w_m of the neural network ϕ_{mes} and the width w_u of the neural ϕ_{upd} (see section 2.2 for a precise explanation of w_m and w_u). Each point corresponds to one specific setting (d_m, w_m, d_u, m_u). Points inside the same rectangle have the same (w_m, w_u)-setting and differ in the (d_m, d_u)-setting. Which position inside the rectangle corresponds to which (d_m, d_u)-setting, can be seen by the rectangle on the right-hand side of the plot. The color of each point encodes the test accuracy. Note, that only points with an accuracy above 80% are shown (otherwise, the color map would be too dense and details would no longer be distinguishable).

accuracy. The best accuracies are achieved within the upper left rectangle (i.e. the setting $(d_u, w_u) = (1, 10)$). One can claim that small depths d_u in combination with high widths w_u are preferably for the problem of classifying Bravais classes. As above, regarding the small amount of data, this can only be seen as a hypothesis. One more point regarding figure 11 one can make is the following: We want to draw the attention to the lower left rectangle (i.e. the setting $(d_u, w_u) = (1, 5)$). The points of the bottom row inside this rectangle all have an equally low accuracy. This row corresponds to the setting $(w_m, w_u) = (10, 5)$. Hence, the low accuracy within this row is consistent with the conclusions we draw from figure 10 above.

Next, one can continue trying to find patterns within the remaining four plots (these are the plots $(w_m, d_m), (w_m, d_u), (d_m, d_u), (d_m, w_u)$). However, we were not able to find any more interesting patterns.

Results for training on the 3d dataset As mentioned in section 2.2 we chose the model that performed best in the 2d case and tested in on the 3d dataset. According to the last paragraph, the settings $(d_m, w_m, d_u, w_u) = (2, 30, 1, 10)$ led to the best results in the 2d case. Without any further tweaks of these parameters or changes in the training procedure, the model achieved a test accuracy of 95% on the 3d dataset (see figure 12). As in the 2d case, we trained the model five times in a row and averaged over these five trainings in order to achieve better statistics. We can conclude, that the best performing model in the 2d case did very well in the 3d case too. Out of interest, we trained the worst performing model (in the 2d case) on the 3d dataset too. Interestingly, it performed equally bad (see figure 13) and just reached an accuracy of 74%.

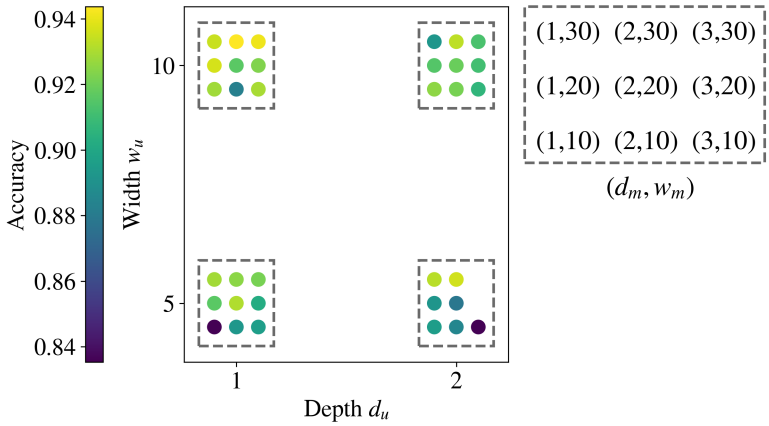


Figure 11: wu vs du

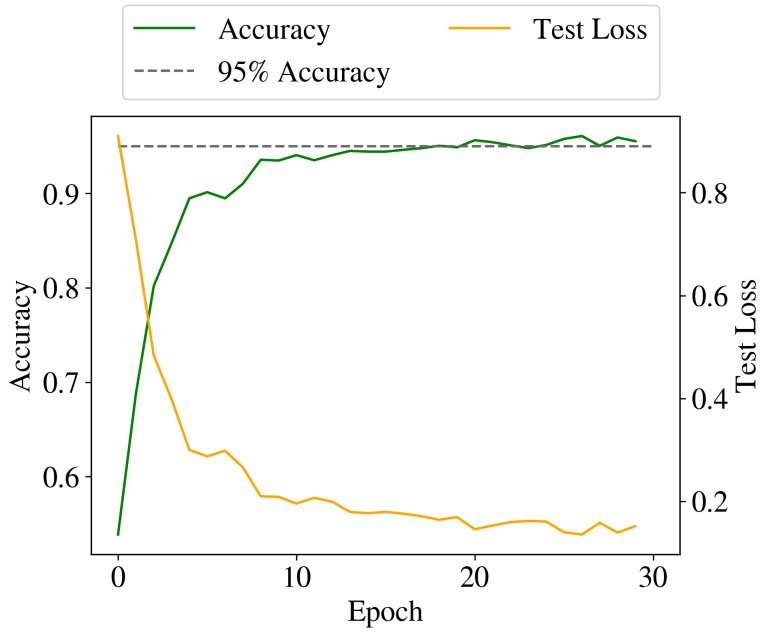


Figure 12: Training process of the model $(d_m, w_m, d_u, w_u) = (2, 30, 1, 10)$ on the 3d datatset.

3.2 On the Problem of Percolation

As mentioned in 2.3, we were asked to experiment with the Top-K Pooling layer as well with message passing layers to tackle the problem of classifying graphs into percolating and non-percolating ones. However, before presenting the results on how well different mixtures of these layers performed, we will give an argument, why the percolation problem cannot be solved completely by a pure message passing GNN, no matter how clever it might be designed. Suppose there is a Message Passing GNN, that can solve the percolation problem, i.e. given any graph with nodes positioned inside the unit square, it can determine whether the graph is percolating or not. Now, take any graph G you like and choose two nodes n, m, that are not connected by an edge (i.e. $\{n, m\} \notin E$). Assign each node different from n, m a position inside $(r, 1-r) \times (r, 1-r)$, place node n inside the strip $[0, 1] \times [0, r]$ and m inside $[0, 1] \times [1-r, 1]$. Furthermore, add the new edge $\{n, m\}$,

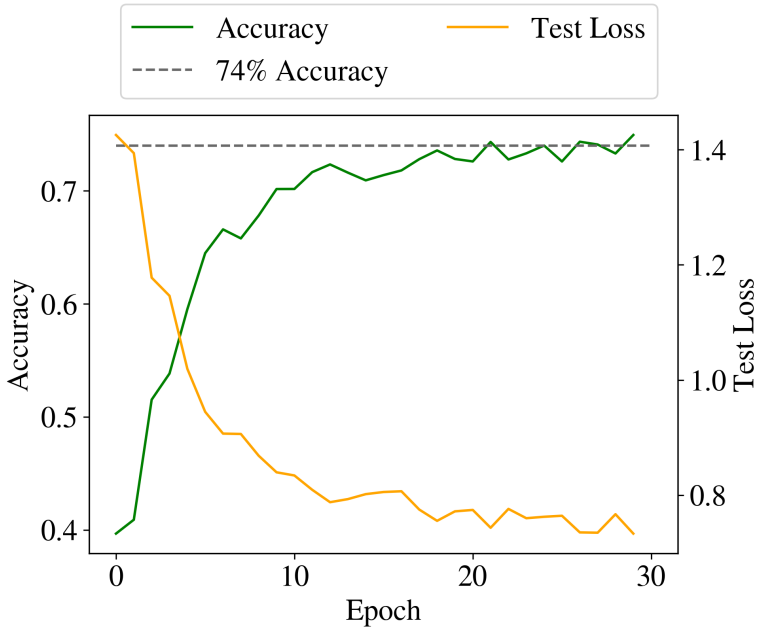


Figure 13: Training process of the model $(d_m, w_m, d_u, w_u) = (3, 30, 2, 5)$ on the 3d dataset.

Figure 14: Illustration, why no Message Passing GNN can detect whether two nodes are in the same connected component. Suppose such a GNN exists and that it has l-layers. Consider the graph with $2l + 2$ nodes depicted above. Since each node can only share information with its l-neighrest neighbors, node 0 can only share information with nodes $0, \cdots, l$ and node $2l+1$ can only receive information from nodes $l + 1, \cdots, 2l + 1$. Hence, there is no possibility for node 0 to know about node $2l + 1$ and therefore, the GNN cannot detect, whether they are in the same connected component or not.

which gives a new graph $\tilde{G}$ that has the same nodes as G and the same edges plus the one additionally added. Next, run the GNN on the graph $\tilde{G}$. Either the GNN outputs that $\tilde{G}$ is not percolating or it outputs that $\tilde{G}$ is percolating. If the graph was percolating, the nodes n and m were already connected in G. In case $\tilde{G}$ was not percolating, n and m were not connected in G. In total, we can use our GNN to detect, whether two randomly chosen nodes (n and m) in a randomly chosen graph (G) are connected via a path or not. However, such a GNN can clearly not exist (suppose there were such a GNN, then consider the graph shown in figure 14, which leads to a contradiction). Hence, a GNN that is capable of solving the percolation problem can not exist too. Besides being impossible to solve this problem with message passing layers, we were asked to present a training process nonetheless. The results can be found in figure 15. The model used the settings d_m, d_u, w_m, w_u that were found out do work best in the Bravais class classification as well as the same hyperparameters, but with 10 message passing layers instead of just 2 (it seemed reasonable to use a deeper GNN for this task). Unsurprisingly, the GNN was not able to solve the percolation problem. The ac-

curacy seemingly fluctuates at random between 65% and 75%. One might expect to have an accuracy of about 50%. However, there are graphs which can be fairly easy classified into percolating and non-percolating. To give an example, consider a graph that does not have any edge between a node on the left side and a node on the right side of the unit square (or a node on the bottom and one on the top). By definition, this graph cannot be percolating. In this case, the GNN just have to detect, whether such an edge exists or not. Detecting if such an edge exits is just a matter of measuring lengths of edges, which is easily done, because each edge is already attributed with its length. Furthermore, as the GNN is relatively deep, it is possible, that the network indeed learned to detect percolating graphs, but only when the occurring cycles have a length that is less than the GNNs depth.

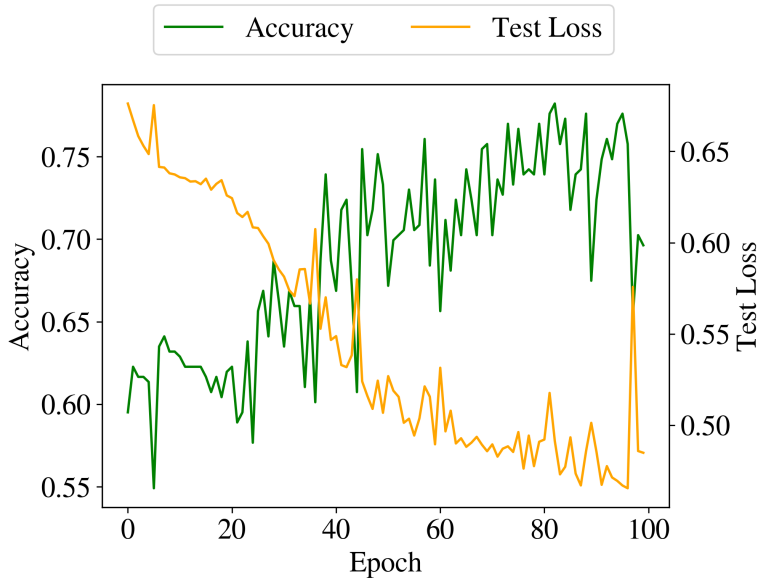


Figure 15: Training of a pure message passing GNN on the percolation problem.

As we cannot hope to solve the percolation problem with pure message passing layers, a natural idea is to try using layers that do not depend on message passing. Furthermore, we learned from figure 14 that the problem lies in graphs that are „too long“, or, putting it differently, that have too many nodes. Hence, we are looking for layers, that reduce the number of nodes and do not depend on message passing. That is the reason why we were asked to work with Top-K Pooling layers. However, there is a good reason, why, even with Top-K Pooling, we cannot hope to achieve anything better than in figure 15. Recall, that the Top-K Pooling procedure deletes nodes with all its edges and does not create new edges. In particular, it does not preserve connected components. To illustrate the point a bit more, consider the following situation: Suppose there is a percolating graph G. After going through the Top-K Pooling layer some nodes might be deleted, so that the resulting graph is not percolating anymore. Hence, instead of looking at pooling layers that do not respect connected components, we have to look at pooling layers, that preserve connected components. Despite being an interesting challenge, this goes beyond the scope of this project. However, there are promising approaches and already existing pooling layers, that preserve structures to some extent (for an overview, see for example [6]). Unfortunately, these layers are not

yet implemented in PyG and implementing these layers goes well beyond this project. To conclude, Top-K Pooling layers will not help with the percolation problem. Nonetheless, we were again asked to present some training results. We used the same model as above with the same hyperparameters but introduced a Top-K Pooling layer after the first message passing layer. The Top-K Pooling layer was configured so that it reduced the number of nodes by 50%. The training results can be found in figure 16. At a first sight, the training seems a bit more stable than without the Top-K Pooling layer. However, the random fluctuations between 65% and 75% are still present. Overall, the training does not show any improvements regarding the accuracy compared to figure 15.

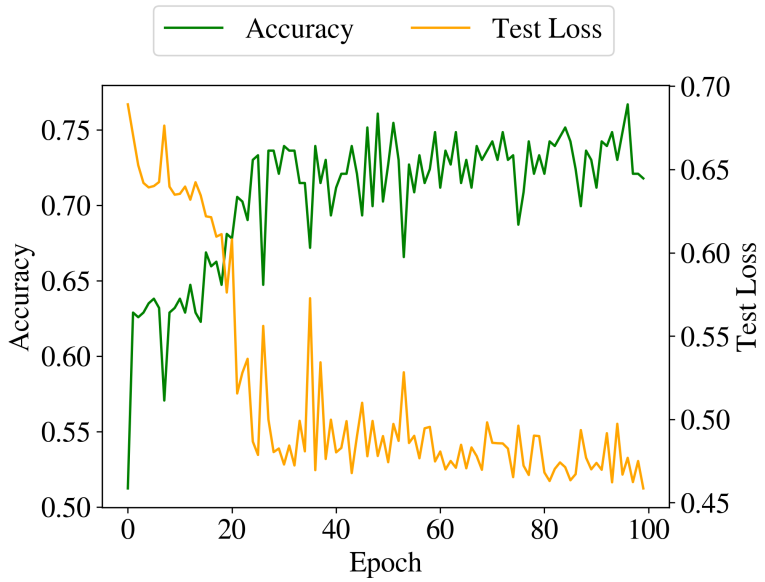


Figure 16: Training of a message passing GNN additionally equipped with a Top-K Pooling layer on the percolation problem.

4 Conclusion, Outlook and Code Availability

At this point, we presented all our results and want to reiterate what the goals of this project were, which goals we successfully accomplished and which problems we could not solve. The primary goal was to get familiar with Graph Neural Networks. We successfully programmed and trained GNNs, understood how message passing works, so that we definitely can say, we took our first steps into the GNN-world. Secondly, we wanted to build a GNN capable of assigning 2-dimensional and 3-dimensional lattices their Bravais class. We build a GNN that can solve this task with an accuracy of 95%, both in 2d and in 3d. Furthermore, we investigated the influence of different settings in the message passing procedure. The third goal, was to build a GNN than can detect percolation. This problem could not be solved. We learned, that there are fundamental limitations of message passing GNNs, like detecting whether two nodes are in the same connected component.

What can be done next? Clearly, for the Bravais classification task one can try to vary node and edge features and try out different training procedures. However, the more interesting part is probably the percolation problem and related tasks. We already tried to overcome the limitations of message passing layers with the usage of a Top-K Pooling layer. However, this layer is not suitable for the problem at hand. But we have seen that there are layers which might be more suitable. Implementing these layers would be a very interesting future challenge.

The code for this project is made public on GitHub. It is available at `https://github.com/BenWen0/ComputationalPhysics2`.

References

[1] Bernd Bischl et al. *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges*. 2021. arXiv: 2107.05847 [stat.ML]. URL: https://arxiv.org/abs/2107.05847.

[2] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[3] Hongyang Gao and Shuiwang Ji. "Graph U-Nets". In: *CoRR* abs/1905.05178 (2019). arXiv: 1905.05178. URL: http://arxiv.org/abs/1905.05178.

[4] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. 2017. arXiv: 1704.01212 [cs.LG]. URL: https://arxiv.org/abs/1704.01212.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[6] Daniele Grattarola et al. "Understanding Pooling in Graph Neural Networks". In: *CoRR* abs/2110.05292 (2021). arXiv: 2110.05292. URL: https://arxiv.org/abs/2110.05292.

[7] Charles Kittel. *Introduction to Solid State Physics*. 8. ed. Wiley, 2005.

[8] Willard Miller. *Symmetry Groups and Their Applications*. Academic Press, 1972.

[9] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *CoRR* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[10] R. L. E. Schwarzenberger. "Classification of crystal lattices". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 72.3 (1972), pp. 325–349. DOI: 10.1017/S0305004100047162.

[11] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From Theory to Algorithms*. http://www.deeplearningbook.org. Cambridge University Press, 2014.