Structure Recognition with Graph Neural Networks

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

Benedikt Wenzel

January 28, 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, for example in crystals. Machine Learning on the other hand is well known for its ability in 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.

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 2d and 3d and then continue with the most fundamental background of Graph-Neural Networks (GNN). Results on Bravais lattices can be found in many textbooks. Subsection 1.1 follows [6] and [5]. Introduction to Graph Neural Networks can be found for example in [3] which is also the main reference for section 1.3.

1.1 Bravais lattice

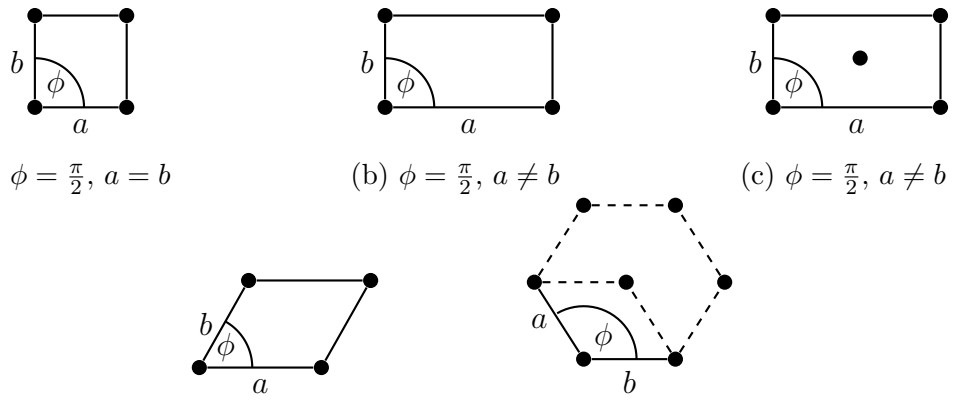
Let $d \in \mathbb{N}$ and $\{b_i\}_{i=1,\dots,d} \subset \mathbb{R}^d$ a basis of $\mathbb{R}^d$. The set

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

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 $g \in GL_n(\mathbb{R})$ 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 [4].



(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 a bit 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 $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(l)}, b^{(l)}) \in \mathbb{R}^{n_0 n_1} \times \mathbb{R}^{n_0} \times \cdots \times \mathbb{R}^{n_{l-1}n_l} \times \mathbb{R}^l$. Clearly, $\tilde{F}$ depends on the choice of $T^{(l)}$ and hence of 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

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

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 if as a node. Applying $f^{(1)} \circ T^{(2)}$ to x yields a new vector $y^1 = f^{(1)} \circ T^{(2)}(x) \in \mathbb{R}^{n_1}$. 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 definitinos we have

$$\begin{aligned} y_j^i &= a^{(i)}\left(T^{(i)}\left(y^{i-1}\right)\right)_j \\ &= a^{(i)}\left(\sum_{k=1}^{n_{i-1}} \left(W^{(i)}\right)_{jk}^T y_k^{i-1} + b_j^{(i)}\right) \\ &= a^{(i)}\left(\sum_{k=1}^{n_{i-1}} W_{kj}^{(i)} y_k^{i-1} + b_j^{(i)}\right). \end{aligned}$$

Up to application of $a^{(i)}$, the node y_j^i is a weighted sum of all nodes y_k^{i-1} in the previous layer (in addition to a constant bias value $b_j^{(i)}$). The matrix element $W_{kj}^{(i)}$ describes how much the node y_j^i is influenced by node y_k^{i-1}. We picture an edge

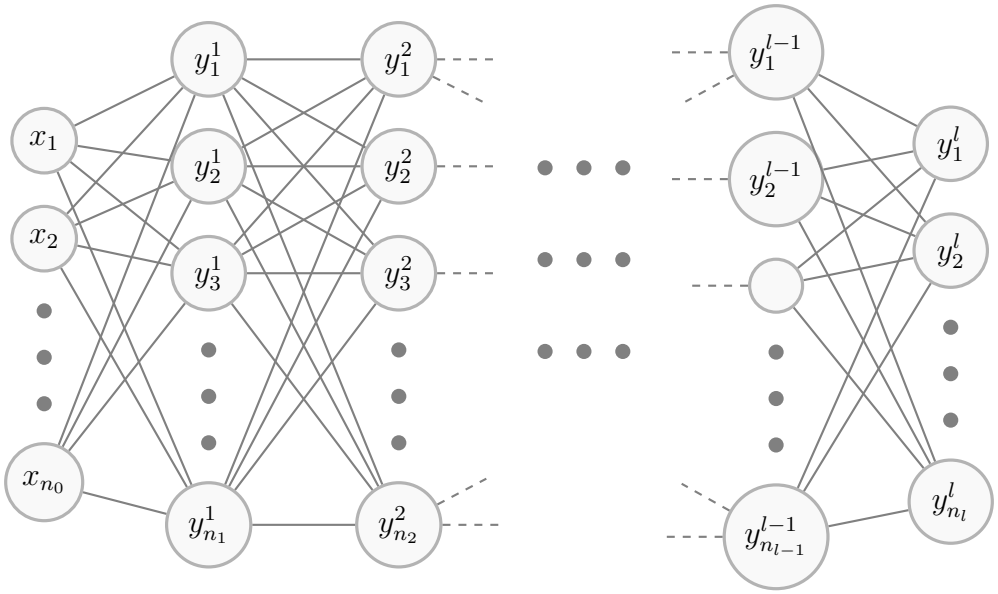


Figure 2: Visualization of a feed forward neural network. [warum können nicht einfach alle nodes gleich groß sein?!]

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 network 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 the 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

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

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 functions, 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 theory about optimization of real valued functions. A common way to do this 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. As F_θ is a large composition, computing 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 $i = 1, \dots, d$.

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, and 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.

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

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.

Roughly, a GNN takes a graph with all its nodes and edge features as an input and manipulates these 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

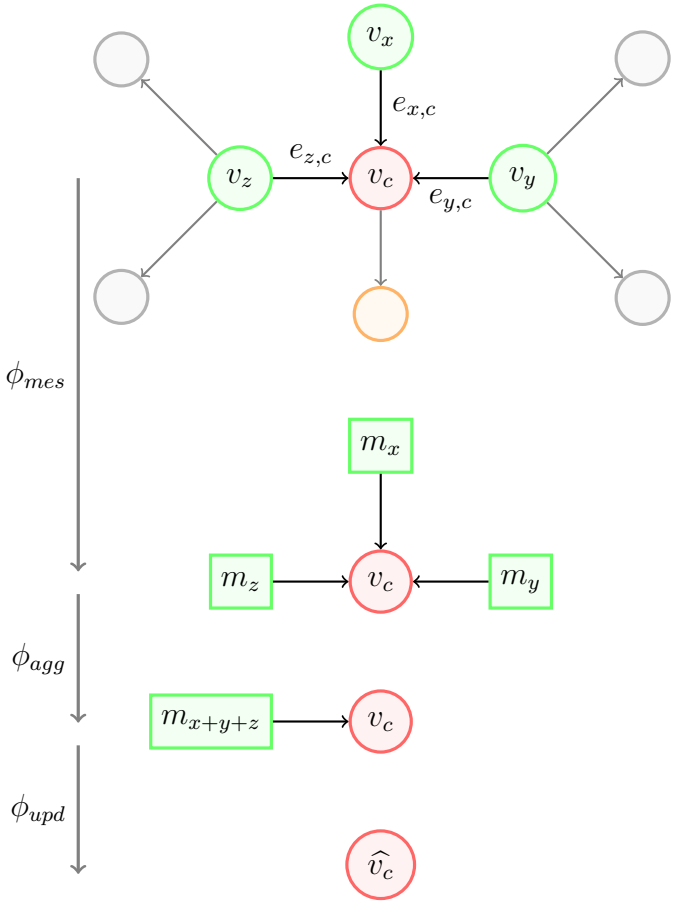


Figure 3: Illustration of the message passing procedure according to equation (2). The neighbors of node c (red) are the nodes x, y, z (green). Attention: According to equation (1) the orange node is not regarded as a neighbor of c as the edge points in the wrong direction. For each neighbor (x, y, z), 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 $\widehat{v_c}$.

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 a percolation. We start with an undirected graph $G = (V, E)$ and want to define what paths and cycles are. Pick 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 is $(m_0, m_1, ..., m_N)$ is called a cycle if $m_i \neq m_j$ for $i \neq j$ (i.e. no node is visited twice).

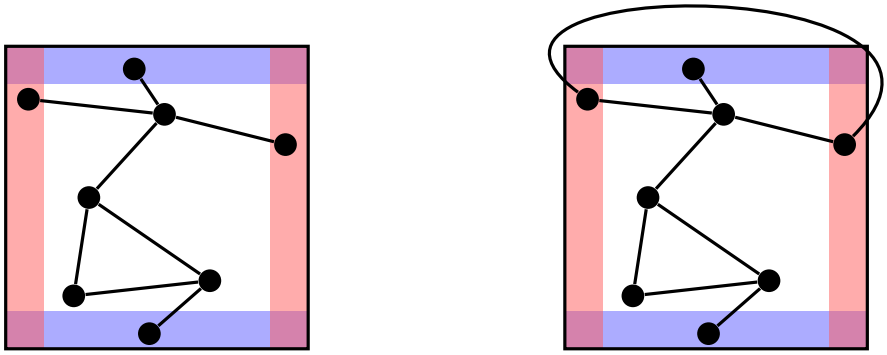
Knowing what paths and cycles are, everything is set up to introduce percolating graph. Suppose $G = (V, E)$ is an undirected graph, where each node $n \in V$ has a position $p_n = (n_x, n_y) \in [0, 1] \times [0, 1]$. Visually, we think of G as a graph inside the unit square. Fix $\frac{1}{2} > r >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

1. there is a cycle containing n and m,
2. $\{n, m\} \in E$ and
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 trips 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 on the left side of the unit square while m is located on the right side. Accordingly, $n_y < r$ 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 sites. 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.



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

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

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

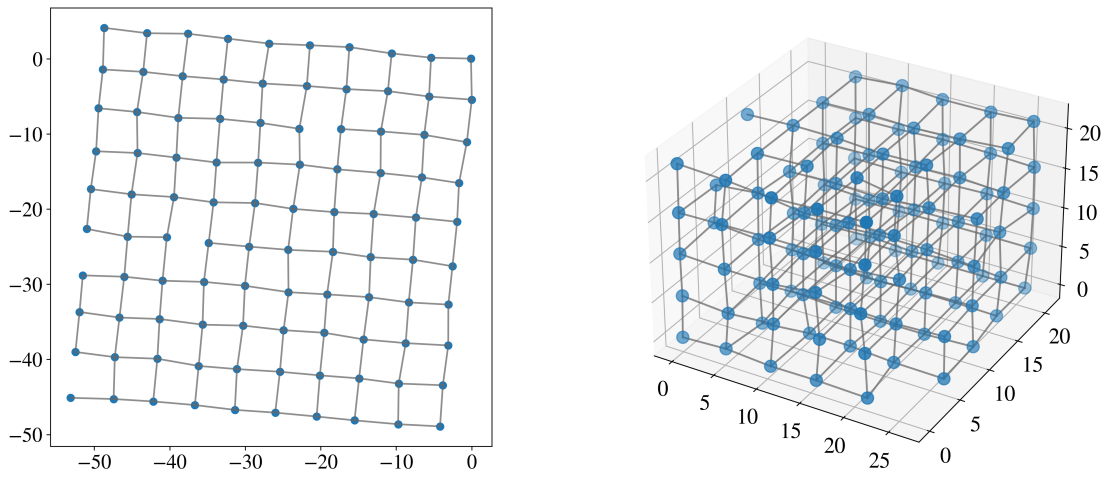
To be precise, we will not create lattice according to the definition given in sectio 1.1, as this would require the creation of sets with infinite elements. Clearly, a GNN can only deal with finitely many nodes. Hence, we will only create finite subsets of lattice but, for sake of simplicity, we will 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 $e_x = (0, a)^T$, $e_y = (b\cos(\phi), b\sin(\phi))^T$ (for visulaization ogf a, b and θ, see figure 1). 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 Ω. Hence, we choose $N_x, N_y \in \mathbb{N}$ and set $\Omega = \mathbb{N}_{\leq N_x} e_x \times \mathbb{N}_{\leq N_y} e_y \subset \tilde{\Omega}$. This lattice now has $N_x N_y$ elements. Next, we add some noise to the elements in Ω and restrict their coordinates to a certain range. For this instance, let $\mu, \sigma, s \in \mathbb{R}_+$ and 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 (component-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 simply 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. 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).

The creation of three-dimensional lattices is completely analogously. The only difference is that we have to pick three basis vectors e_x, e_y, e_z which have length



(a) 2d (b) 3d

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

$a, b, c \in \mathbb{R}_+$ and enclose angles $\phi = \angle(e_x, e_y)$, $\psi = \angle(e_x, e_z)$, $\chi = \angle(e_y, e_z)$.

Figure 5 shows two different lattices created by the above procedure.

According to the above procdeure, we are able, to generate training and test datasets for our GNN, both in 2d and in 3d. We start with a quick describtion of the 2d-dataset. First, epending 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 of a, b, θ result in which Bravais class). Recall, that in some cases $\theta = \frac{\pi}{2}$ is required. In order to introduce a bit artifical 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 fullfill $a = b$, 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 be 10. The parameters determining the above mentioned noise, where chosen to be $\sigma = 0.5, \mu = 0$. The amplitude of the noise in x-direction was taken to be $s = 0.07a$ (depeneding on the side-length a) as well as $s = 0.07b$ in y-direction. The probabilities p_n, p_e for randomly droping nodes/edges were taken to be $p_n = p_e = 0.01$ According to these rules, 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%4 constitute the training set.

[3d Datensatz]

2.2 Implementing the GNN

Once we have a good amount of training data at hand, we need to determine the optimal structure of the GNN. This amounts to finding the right functions $\phi_{upd}, \phi_{mes}, \phi_{agg}$ in equation 2, and proper number of message passing layers. More than that, one has to decide, which edge-/node-features do best, as well as which hyperparamters to use during training.

As we do not have many computatinoal resources available, we were unable to vary all of these parameters. We fixed the hyperparamters as well as the node-/edge features as follows: Both in the 2d and in the 3d case the each $e_{n,m}$ between

nodes n, m has the difference of the positions of nodes n and m as edge feature. The nodes n and m do not carry any specific node feature. Each node got the number one 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 was used. Each training consisted of 30 epochs with a batch size of 32.

All other paramters mentioned at the beginning of this section were varied in the following way: In all the following experiments, for computational convenience, ϕ_{agg} was chosen to be the function that sums up all its inputs. 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 fith parameter that needs to be optimized, namely the number d_G of message passing layers. However, five paramters are computatinoally to 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

$$\begin{aligned} d_m &\in \{1, 2, 3\} \\ w_m &\in \{10, 20, 30\} \\ d_u &\in \{1, 2\} \\ w_u &\in \{5, 10\}. \end{aligned}$$

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. The question we are trying to answer, is whether the settings that did best in the 2d case also work in the 3d case.

A few words on the actual implementation: Luckily, one does not have to implement the whole message passing scheme. Instead, the PyG-package ([1]) comes equipped with a base class called MessagePassing. This class in 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 functinos were implemented. It is not necessary to go into detail about the implemetation of the whole message passing scheme.

2.3 On the Problem of Percolation

I was asked to experiment with the so called TopKPooling layer. What follows is a rough overview over the principles of this layer. For additional information see [2], 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 a graph. Recall from section 1.3, that 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

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

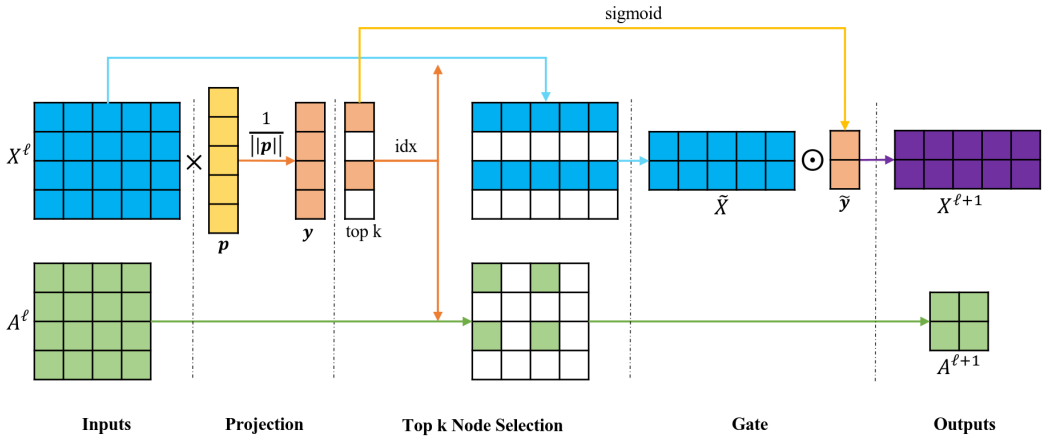


Figure 6: Illustration of the TopKPooling layer. Taken from [2].

Furthermore, we can define the so called adjancecy matrix $A^l \in \mathrm{Mat}(|V| \times |V|, \mathbb{R})$ given by

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

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 $V \setminus \tilde{V}$ 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 $x \in \tilde{V}$, analouguls to equation 2.3. And define analogously to equation 2.3 $A^{l+1} \in \mathrm{Mat}(k \times k, \mathbb{R})$ to be the adjancecy matrix corresponding to the nodes in $\tilde{V}$. Next, compute the elementwise product of $\tilde{y}$ and $\tilde{X}$ which leads to new matrix $X^{l+1} \in \mathrm{Mat}(k \times n, \mathbb{R})$. This step is called „gate operation“. The matrices node feature matrix X^{l+1} together with the adjancency matrix A^{l+1} define a new graph which has $k < |V|$ nodes. Effectivvely, 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 $\tilde{G}$. Given a specific problem, the goal of the TopKPooling layer is to learn a suitable projection vector p. The gate operation steps 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 [2])

Obviously, the dataset created in section 2.1 is not feasable for the percolation problem. Instead, the dataset for the percolation problem was created by Jonas Buba and his collegeaus. The dataset consists of 1614 planar graphs in total, 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 as above) and a test set (20%). The nodes have their position in the unit cube as attributes, whereas the graphs do not have any attributes.

(a) Percolating (b) Non-Percolating

Figure 7: Example of graphs that are in the percolation dataset.[TODO]

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 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 training on the 2d-dataset and then go on to the 3d case.

Results on the 2d dataset As mentioned in section 2.2 we tested in total 36 different combinations of values for d_m, w_m, d_u, m_u. 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. With an accuracy of approximately 90%, the problem can be considered

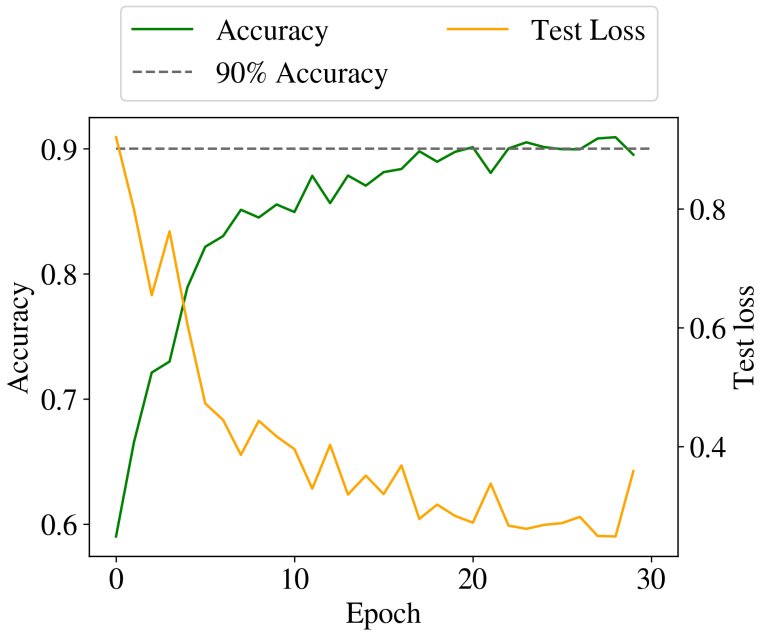
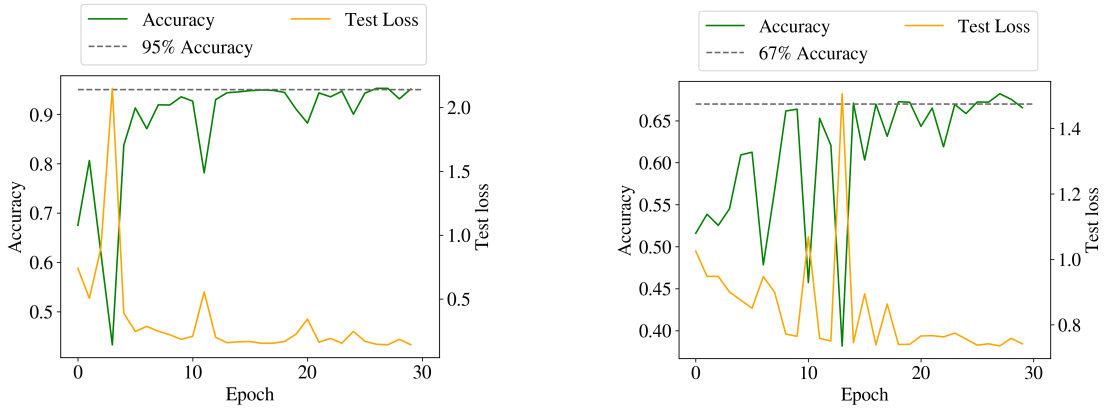


Figure 8: avg bravais 2d

solved. However, there are non-negligtable difference between models. The model that performed best achieved an accuracy about 95%, whereas the model that performed worst only managed to get about 67% accuracy. The training process of both models are dpeicted in figure 9. Hence, different values for d_m, w_m, d_u, m_u may lead to drastically different outcomes.



(a) best (b) worst

Figure 9: bravais 2d

Results 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. As explained in the settings $d_m =, w_m, = d_u =, m_u =$ led to the best results in the 2d case. Without any further tweaks of these paramters or changes in the training procedure, the model achieved a test accuracy of $xy\%$ on the 3d dataset (see figure 10). Out of

Figure 10: best bravais 3d

interest, we trained to worst performing model in the 2d dataset on the 3d dataset too. Interestingly, it performed equally bad (see figure 11).

Figure 11: worst bravais 3d

3.2 On the Problem of Percolation

As mentioned in 2.3 I was asked to experiment with the TopKPooling layer. However, before presenting the results on how well this layer performed, we will give an argument, why the percolation problem cannot be solved by a MessagePassing 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 or not the graph is percolating. 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 $\{n, m\}$, which gives a new graph $\tilde{G}$ that has the same nodes as G and the same edges plus the one additinal more. 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 $\tilde{G}$ is not percolating, n and m are 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

Figure 12: No Message Paassing GNN can detect whether two nodes are in the same connected coomponent. 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 possibilty 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.

Figure 13: bla bla bla

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 12, which leads to a contradiction). Hence, a GNN that is capable of solving the percolation problem can not exist too. Besides beeing provable impossible to solve this problem, I was asked to present some training processes nonetheless. The resulst can be seen in figure 13. Unsurprisingly, the GNN was not able to solve the percolation problem.

As we cannot hope to overcome the percolation problem with pure message passing layers, a natural idea is to try using layers that do not depend in message passing. That is the reason why I was asked work with the TopKPooling-Layer. However, there is a good reason, why, even with the TopKPolling, we cannot hope to achieve anything better than in 13. Recall, that the TopKPooling procedure deletes nodes with all its edges and does not create new edges. In particular, it does not presever connected components. To illistrate the point a bit more, consider the follwoing situation: Suppose there is a percolating graph G. After going through the TopKPolling layer some node might be deletet, so that the resulting graph is not percolating anymore. Hence, instead of looking at Pooling layers that do not respect cnnected components, we have to look at pooling layers, that preserve connected components. Depsite beeing a interesting challenge, this goes beyond the scope of this project. Hence, a TopKPolling layer will not help with the percolatino problem. Nonetheless, I was again asked to present some training results. They can be found in figure 14. As expected, the training does not show any improvements compared to figure 13.

Figure 14: bla bla bla

4 Conclusion and Outlook

References

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

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

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

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

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

[6] 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.