
# Investigation of transformer architectures for geometrical graph structures and their application to two-dimensional spin systems

## Bachelor Thesis

submitted by

Jonas Kell

on 7th October 2022

*Augsburg University*
*Faculty of Applied Computer Science*
*Institute of Computer Science*
*Chair for Machine Learning & Computer Vision*

1st Corrector: Prof. Dr. Rainer Lienhart
2nd Corrector: Prof. Dr. Markus Heyl

# Table of Contents

# List of Abbreviations

| Abbreviation | Meaning |
|---|---|
| AI | Artificial Intelligence |
| GPU | Graphics Processing Unit |
| TPU | Tensor Processing Unit |
| NQS | Neural Quantum State |
| VMC | Variational Monte Carlo |
| DMC | Diffusion Monte Carlo |
| nn | nearest neighbor |
| nnn | next nearest neighbor |
| sgd | stochastic gradient descent |
| ReLU | Rectifier Linear Unit |
| RBM | Restricted Boltzmann Machine |
| fcl | fully connected layer |
| MLP | Multi Layer Perceptron |
| rgb | red green blue |
| sdp attention | scaled dot product attention |
| swin | shifted window |
| XLA | Accelerated Linear Algebra |
| pe | positional encoding |
| sr | single real |
| sc | single complex |
| tr | two real |
| spc | split complex |
| NaN | Not a Number |

The abbreviations of the different metaformer names are collected in Table 3.2.

# 1 Introduction

AI architectures are rapidly evolving. With the surging availability of data due to cheap storage [1] and high resolution sensors in every device, the need for processing said information follows this trend. As every single person is producing data at an unprecedented rate [2], the growing interconnectedness due to advances in social media and the internet fuels the need to not only process the data one self is producing, but also the data others are generating. Various large and small corporations are depending on high throughput and high accuracy data extraction to allow them to provide their services [3]. At the same time, individuals call for faster and more accurate processing of their voice and images by computational intelligences.
Due to the likewise surging developments in GPU and TPU computing [4], larger and more elaborate neural networks can be evaluated. Usually this comes along with huge requirements in hardware and power, which often times can only be efficiently met in datacenter-scale computations.
This however harshly conflicts with the population's demand for privacy. Processing of highly confidential data should best only be performed offline and locally on the user's device.
In order to meet these demands for high performance and low footprint networks, very specialized network-architectures have been developed over the past years [5, 6]. These employ highly technical inductive biases to allow for calculations, that are precisely tailored to the individual task - therefore becoming more performant while remaining or even becoming less computationally expensive [7].
Numerous challenges exist in the domain of many body physics to come up with better and more efficient ways to compute the behavior of quantum systems. With the aid of computers, exact solutions could "simply" be calculated numerically. However as the exponential nature of the tasks at hand is still vastly overshadowing the advances in computational availability [8], the focus was shifted to more involved ways of solving this problem.
The *neural quantum state* as a method to combat the problem with the use neural networks is by now firmly established as a more than capable approach [9]. However the models used in these computations tend to be of relatively modest and seemingly "outdated" nature in comparison to the bleeding edge of computational AI.
This thesis is supposed to take a look at the feasibility and usefulness of employing highly specialized, modern neural network architectures, with specifically tailored inductive biases, to enrich the current computational methods. The target is to compute the *ground state* of a system described by the *transverse field Ising model* [10] in 1D as well as 2D lattices. In the course of this, the focus lies on the *metaformer* architecture [6], *graph networks* [11] and the unification of the two concepts.

# 2 Theory

## 2.1 From the Perspective of Physics

In the beginning the fundamental physical concepts will be defined. The goal is to introduce the physical theory in a way that explains the basics, but doesn't sacrifice the accuracy. By defining the used models and methods, while giving a basic insight into the more advanced notation, the hope is to introduce computer scientists with their knowledge in neural network architectures to the pending problems of many body quantum physics.
In general, physical equations are derived from fundamental differential equations. The classical *equations of motion* for example can be derived from the fundamental differential equations $\vec{v} = \frac{d\vec{r}}{dt}$ and $\vec{F} = m \cdot \frac{d\vec{v}}{dt}$ [12]. These can generally be solved (not always analytically) if the boundary conditions are known.
Quantum systems are also described by fundamental differential equations, the *Schrödinger equation* probably being the most important one. Apart from the obstacle that in quantum mechanics the boundary conditions generally cannot be measured exactly, a lot of problems can be boiled down to finding a specific solution to the Schrödinger equation.

### 2.1.1 The Ground State

This section introduces the fundamentals of (many body) quantum physics. The state of a quantum mechanical system is described by the wavefunction $\Psi$. The wavefunction encodes this state while evaluating at a wide possible range of parameters (position in space, shape of the underlying potential, angular momentum, spin, etc.). The goal is to find a representation, that can encode the quantum state with large precision, while requiring as little inputs and parameters as possible. The choice of the general form of the function used for representing the wavefunction is called the *ansatz*. If a wrong ansatz is chosen, the quantum state cannot be encoded efficiently or at all. This will come into play in subsequent chapters.
A wavefunction is typically written in the *bra-ket notation* (see Equation 2.1). The advantage of this notation is, that it is *independent* of the chosen *base system* [13]. This is used to simplify later notation.

$$\Psi(x) \rightarrow |\Psi\rangle \qquad \Psi_n(x) \rightarrow |n\rangle \tag{2.1}$$

In this case $x$ denotes an arbitrary coordinate. $|\Psi\rangle$ is an arbitrary wavefunction. $|n\rangle$ is a system of $N, 0 \leq n < N$ wavefunctions. Depending on the parameterization, there is a finite or infinite amount of different $|n\rangle$. In *subsection 2.1.2 The Ising Model* the base system used in this application will be introduced. In this thesis only systems with finite $N$ will be discussed.
The following properties will be important:

The representation in a different base (Equation 2.2), where $c_n$ is a complex number that can be obtained via Equation 2.3.

$$|\Psi\rangle = \sum^{n} c_n \cdot |n\rangle \tag{2.2}$$

$$c_n = \langle n|\Psi\rangle = \Psi(n) \tag{2.3}$$

For an *orthogonal base* Equation 2.4 is true, for an *orthonormal base* even the stricter Equation 2.5. $\delta_{n,m}$ is called *Kronecker-δ* [13].

$$\langle m|n\rangle = \begin{cases} = 0 & : n \neq m \\ \neq 0 & : n = m \end{cases} \tag{2.4}$$

$$\langle m|n\rangle = \begin{cases} = 0 & : n \neq m \\ = 1 & : n = m \end{cases} = \delta_{n,m} \tag{2.5}$$

A wavefunction is the correct representation for a quantum system, if it satisfies the differential equation of the problem, the so called *(time-independent) Schrödinger equation* (Equation 2.6), where $\mathcal{H}$ is called the *hamiltonian* of the system. The hamiltonian basically "encodes" everything relevant to the problem. It must be seen as an operator, that performs a transformation of $|\Psi\rangle$, so $\mathcal{H} = E$ is *not* the simplified version of Equation 2.6.

$$\mathcal{H}\,|\Psi\rangle = E \cdot |\Psi\rangle \tag{2.6}$$

$E$ is the energy of the system. As can bee seen here, the Schrödinger equation is a linear differential equation and the correct $|\Psi\rangle$ needs to be chosen, in order for it to be fulfilled. Later this will be transformed into a matrix-vector equation. In that case, $|\Psi\rangle$ needs to be an *eigenvector* to the matrix $\mathcal{H}$.
The challenge - that will be taken on in this thesis - is not only to find any valid $|\Psi\rangle$, but to find the so called *ground state*. The ground state is the eigenstate with the smallest eigenvalue, speaking the smallest *energy* $E = E_0$. This is very interesting for the physical application, because as we know from physics, a system always tries to transition into the state with the smallest energy. A ball will not stay on a slope, but roll down into the valley, where it can sit stable. The quantum system (generally) also tries to exist in the ground state. Therefore, the shape of the ground state tells a lot about the "default" behavior of the system.

### 2.1.2 The Ising Model

In this thesis, a special case of quantum system will be discussed: The *Ising model*. The model describes multiple *spins* that are locked in place. A spin is a very special kind of angular momentum that is inherent to quantum mechanical particles. It helps to imagine the spin as a tiny magnet, that is locked in place, but can freely rotate to react to external/neighboring magnetic fields. This is a very harsh oversimplification, but enough for our purposes.

The state of the *many body wavefunction* can solely be described by the direction that each of the spins is pointing in. In this simplified case, only *spin-*$\frac{1}{2}$ particles are used. That means each spin at each position can only either be pointing *up* ↑ or *down* ↓ (in relation to the z-axis).
That means, that a system with $n = 3$ spins (in arbitrary, but fixed position) can be represented like in Equation 2.7. $\vec{s}_m$ is the direction of the spin vector, that can (in this case) be simplified to the z-axis contribution $s_m^z$ of the vector. The values can only be $\frac{1}{2}$ (↑) or $-\frac{1}{2}$ (↓). In the given example the spins at the positions 1 and 2 point up, the one at 3 points down.

$$\left|\vec{s}_1, \vec{s}_2, \vec{s}_3\right\rangle \rightarrow |s_1^z, s_2^z, s_3^z\rangle = \left|\tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}\right\rangle = |\uparrow, \uparrow, \downarrow\rangle \tag{2.7}$$

The hamiltonian for the system is defined in Equation 2.8. $J_{i,j}$ is the *coupling constant* between the lattice sites $i$ and $j$. $h_i$ is a measure for the external magnetic field (in x-direction) at the lattice site $i$. Often the terms each get multiplied with $-1$ (sing flip of $J$ and $h$) as per convention. This was not done in this case in order for the equation to reflect the code. The $\hat{s}_i^a$ are *operators*, that measure the $s_i^a$ number of the wavefunction (possibly altering its state in the process). $a$ in this case denotes the direction: $a \in \{x,y,z\}$.

$$\mathcal{H} = \sum_{i,j} J_{i,j} \cdot \hat{s}_i^z \hat{s}_j^z + \sum_i h_i \cdot \hat{s}_i^x \tag{2.8}$$

Equation 2.9 is a simplification, that assumes homogeneous interaction constants $J$ and $h$ for all sites, as well as only allowing nearest neighbors ($\langle i, j\rangle$) to interact.

$$\mathcal{H} = J \cdot \sum_{\langle i,j\rangle} \hat{s}_i^z \hat{s}_j^z + h \cdot \sum_i \hat{s}_i^x \tag{2.9}$$

In this form, a $J < 0$ leads to so-called *ferromagnetic* interaction. With a $J > 0$ the interaction is called *anti-ferromagnetic*. When $h \neq 0$, the model is called *transverse field Ising model* [10].
The model was first solved for a 1D chain by Ernst Ising in 1924 [14]. He solved the ferromagnetic case without a transverse field (however with an additional field parallel to the z-direction). His solution was analytically derived, but already not trivial to come up with. With the introduction of a transverse field and a more complex lattice structure than a linear chain, an analytic solution to even the most simplified hamiltonian (Equation 2.9) does not exist.
In order to be able to use the hamiltonian, a few rules about the interaction of the used operators and the wavefunction will be introduced. They represent only a small subset of the mathematics and focus only on the bare minimum needed to understand the problem as someone that is unfamiliar with quantum mechanics.
The spin operators can be written in terms of the *Pauli spin matrices* ($\sigma^a, a \in \{x,y,z\}$ in Equation 2.10) [13]. This is only mentioned, because it reflects the implementation in the code. Note that in the code, the $\frac{1}{2}$ factors in Equation 2.10 are omitted for simplification. $i$ stands for the imaginary unit.

$$\hat{s}^x = \frac{1}{2} \cdot \sigma^x = \frac{1}{2} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \hat{s}^y = \frac{1}{2} \cdot \sigma^y = \frac{1}{2} \cdot \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \hat{s}^z = \frac{1}{2} \cdot \sigma^z = \frac{1}{2} \cdot \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.10}$$

With the corresponding Pauli representation of the spin vectors in Equation 2.11, the interaction of the operators can be directly computed with standard matrix multiplication. The important cases are calculated in Equation 2.12.

$$|\uparrow\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |\downarrow\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.11}$$

$$\sigma^z |\uparrow\rangle = +1 \cdot |\uparrow\rangle \quad \sigma^z |\downarrow\rangle = -1 \cdot |\downarrow\rangle \quad \sigma^x |\uparrow\rangle = +1 \cdot |\downarrow\rangle \quad \sigma^x |\downarrow\rangle = +1 \cdot |\uparrow\rangle \tag{2.12}$$

The last important piece of information is, that operators denoted by indices only interact with the spin at the corresponding lattice site. Three examples are given in Equation 2.13. Multiple operators acting on one state can be computed by only evaluating the effect of the rightmost operator according to the mentioned rules and then repeating until finished. Complex numbers can always be commutatively swapped with operators and bra-kets.

$$\sigma_2^z |\downarrow,\uparrow,\downarrow\rangle = +1 \cdot |\downarrow,\uparrow,\downarrow\rangle \quad \sigma_1^z |\downarrow,\uparrow,\downarrow\rangle = -1 \cdot |\downarrow,\uparrow,\downarrow\rangle \quad \sigma_3^z |\downarrow,\downarrow,\downarrow\rangle = +1 \cdot |\downarrow,\downarrow,\uparrow\rangle \tag{2.13}$$

### 2.1.3 Solution through Exact Diagonalization

With the combined knowledge of the two last sections, the ground state (and the ground state energy) can be calculated. On the one hand it will become evident, that the calculation is really simple to do and program. On the other hand it will be shown that it is not feasable to compute sufficiently large lattices, because of the exponential nature of the problem.

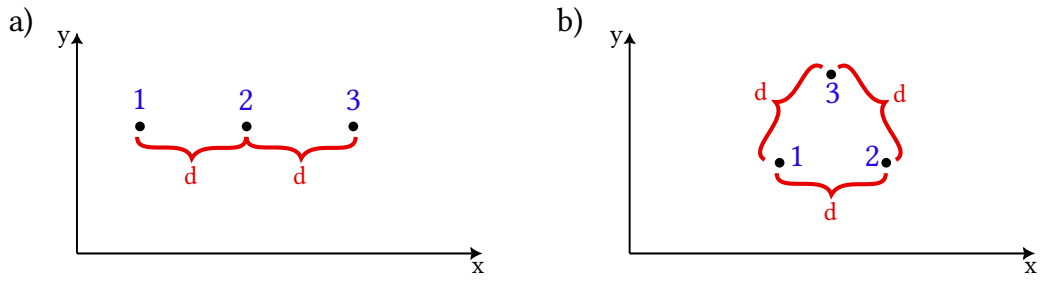


**Figure 2.1**: A representation of two simple latices. The spins are fixed at the point of the numbered locations. All sites have the same distance $d$ from their neighbors. Note, that a spin up $\uparrow$ is directed parallel to the z-axis. That means in this case it would point out of the paper in the direction of the viewer.

In the beginning one trivial example will be discussed. The lattice is visualized in Figure 2.1a. This corresponds to a linear lattice in 1D. Let the hamiltonian be the one in Equation 2.14.

$$\mathcal{H} = \sigma_1^z \sigma_2^z + \sigma_2^z \sigma_3^z \tag{2.14}$$

This hamiltonian is anti-ferromagnetic ($J = +1$), therefore the energy $E$ in Equation 2.6 becomes minimal (largest magnitude with negative sign) when the spins always point into the opposite direction as their neighbors: $|\uparrow,\downarrow,\uparrow\rangle$ or $|\downarrow,\uparrow,\downarrow\rangle$. This is quite a crucial step, so validating this yourself is advised for someone without prior quantum mechanical experience. The ground

state therefore will be a *superposition* consisting of equal parts $|\uparrow,\downarrow,\uparrow\rangle$ and $|\downarrow,\uparrow,\downarrow\rangle$ (Equation 2.15). The factor $\frac{1}{\sqrt{2}}$ instead of the seemingly more obvious $\frac{1}{2}$ makes the wavefunction be *normalized*. If it wasn't for the square root, Equation 2.5 would not be satisfied. This way it is satisfied, which can be tested.

$$\left|\Psi_{\text{gnd}}\right\rangle = \frac{1}{\sqrt{2}} \cdot |\uparrow,\downarrow,\uparrow\rangle + \frac{1}{\sqrt{2}} \cdot |\downarrow,\uparrow,\downarrow\rangle \tag{2.15}$$

The second example, visualized in Figure 2.1b, however is not as simple. The corresponding hamiltonian can be seen in Equation 2.16.

$$\mathcal{H} = \sigma_1^z\sigma_2^z + \sigma_2^z\sigma_3^z + \sigma_1^z\sigma_3^z \tag{2.16}$$

It has one more interaction than the previous one. Even though it also is anti-symmetric, it is *not* possible to align all three spins in a way, so that they stand anti-parallel to all their neighbors. This phenomenon is called *frustration* [15].
Already this minimal problem has a complex superposition of eight base states as a ground state. The introduction of the transverse field will only make this more complex. Therefore a systematic way of finding the solution will be presented.
For this purpose, the example from Figure 2.1b will be used once more, but this time with the hamiltonian in Equation 2.17.

$$\mathcal{H} = J \cdot \sigma_1^z\sigma_2^z + J \cdot \sigma_2^z\sigma_3^z + J \cdot \sigma_1^z\sigma_3^z + h\sigma_1^x + h\sigma_2^x + h\sigma_3^x \tag{2.17}$$

The goal is now, to write the hamiltonian $\mathcal{H}$ as a matrix. The elements of the matrix should be $\langle m|\,\mathcal{H}\,|n\rangle$. Where $|m\rangle$ and $|n\rangle$ are two states out of the chosen basis. In this case the eight base states are the following (*canonical spin basis*):

| | | | |
|---|---|---|---|
| 1: $\|\uparrow,\uparrow,\uparrow\rangle$ | 2: $\|\uparrow,\uparrow,\downarrow\rangle$ | 3: $\|\uparrow,\downarrow,\uparrow\rangle$ | 4: $\|\uparrow,\downarrow,\downarrow\rangle$ |
| 5: $\|\downarrow,\uparrow,\uparrow\rangle$ | 6: $\|\downarrow,\uparrow,\downarrow\rangle$ | 7: $\|\downarrow,\downarrow,\uparrow\rangle$ | 8: $\|\downarrow,\downarrow,\downarrow\rangle$ |

In Equation 2.18 an example calculation of $\langle 2|\,\mathcal{H}\,|4\rangle$ is presented.

$$\langle\uparrow,\uparrow,\downarrow|\,\mathcal{H}\,|\uparrow,\downarrow,\downarrow\rangle =$$

$$\begin{aligned}&\langle\uparrow,\uparrow,\downarrow|\,J\cdot\sigma_1^z\sigma_2^z\,|\uparrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,J\cdot\sigma_2^z\sigma_3^z\,|\uparrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,J\cdot\sigma_1^z\sigma_3^z\,|\uparrow,\downarrow,\downarrow\rangle+\\&\langle\uparrow,\uparrow,\downarrow|\,h\cdot\sigma_1^x\,|\uparrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,h\cdot\sigma_2^x\,|\uparrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,h\cdot\sigma_3^x\,|\uparrow,\downarrow,\downarrow\rangle \overset{2.12,2.13}{=}\end{aligned}$$

$$\begin{aligned}&\langle\uparrow,\uparrow,\downarrow|\,J\cdot 1\cdot(-1)\,|\uparrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,J\cdot(-1)\cdot(-1)\,|\uparrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,J\cdot 1\cdot(-1)\,|\uparrow,\downarrow,\downarrow\rangle+\\&\langle\uparrow,\uparrow,\downarrow|\,h\cdot 1\cdot|\downarrow,\downarrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,h\cdot 1\cdot|\uparrow,\uparrow,\downarrow\rangle+\langle\uparrow,\uparrow,\downarrow|\,h\cdot 1\cdot|\uparrow,\downarrow,\uparrow\rangle =\end{aligned} \tag{2.18}$$

$$\begin{aligned}&(-J)\cdot\langle\uparrow,\uparrow,\downarrow|\uparrow,\downarrow,\downarrow\rangle+J\cdot\langle\uparrow,\uparrow,\downarrow|\uparrow,\downarrow,\downarrow\rangle+(-J)\cdot\langle\uparrow,\uparrow,\downarrow|\uparrow,\downarrow,\downarrow\rangle+\\&h\cdot\langle\uparrow,\uparrow,\downarrow|\downarrow,\downarrow,\downarrow\rangle+h\cdot\langle\uparrow,\uparrow,\downarrow|\uparrow,\uparrow,\downarrow\rangle+h\cdot\langle\uparrow,\uparrow,\downarrow|\uparrow,\downarrow,\uparrow\rangle \overset{2.5}{=}\end{aligned}$$

$$(-J)\cdot 0+J\cdot 0+(-J)\cdot 0+h\cdot 0+h\cdot 1+h\cdot 0=h$$

Placing them inside a matrix at the corresponding location yields the matrix representation for $\langle m|\,\mathcal{H}\,|n\rangle$ that can be seen in Equation 2.19. Because of Equation 2.3, each wavefunction can be written in terms of the chosen base. This is used in Equation 2.20. By representing the wavefunction as a column vector, filled with the scaling factors, the transformation is complete. The calculation of the hamiltonian $\mathcal{H}$ acting on a wavefunction $\Psi$ as in Equation 2.6 can now be computed analogously by performing the matrix multiplication $\langle m|\,\mathcal{H}\,|n\rangle\cdot\Psi_{\text{vec}}$.

$$\begin{array}{c} \\ \langle\uparrow,\uparrow,\uparrow| \\ \langle\uparrow,\uparrow,\downarrow| \\ \langle\uparrow,\downarrow,\uparrow| \\ \langle\uparrow,\downarrow,\downarrow| \\ \langle\downarrow,\uparrow,\uparrow| \\ \langle\downarrow,\uparrow,\downarrow| \\ \langle\downarrow,\downarrow,\uparrow| \\ \langle\downarrow,\downarrow,\downarrow| \end{array} \begin{array}{c} \begin{array}{cccccccc} |\uparrow,\uparrow,\uparrow\rangle & |\uparrow,\uparrow,\downarrow\rangle & |\uparrow,\downarrow,\uparrow\rangle & |\uparrow,\downarrow,\downarrow\rangle & |\downarrow,\uparrow,\uparrow\rangle & |\downarrow,\uparrow,\downarrow\rangle & |\downarrow,\downarrow,\uparrow\rangle & |\downarrow,\downarrow,\downarrow\rangle \end{array} \\ \begin{pmatrix} 3\cdot J & h & h & 0 & h & 0 & 0 & 0 \\ h & -1\cdot J & 0 & h & 0 & h & 0 & 0 \\ h & 0 & -1\cdot J & h & 0 & 0 & h & 0 \\ 0 & h & h & -1\cdot J & 0 & 0 & 0 & h \\ h & 0 & 0 & 0 & -1\cdot J & h & h & 0 \\ 0 & h & 0 & 0 & h & -1\cdot J & 0 & h \\ 0 & 0 & h & 0 & h & 0 & -1\cdot J & h \\ 0 & 0 & 0 & h & 0 & h & h & 3\cdot J \end{pmatrix} \end{array} \tag{2.19}$$

$$\begin{aligned}\Psi =&c_1\,|\uparrow,\uparrow,\uparrow\rangle+c_2\,|\uparrow,\uparrow,\downarrow\rangle+c_3\,|\uparrow,\downarrow,\uparrow\rangle+c_4\,|\uparrow,\downarrow,\downarrow\rangle+\\&c_5\,|\downarrow,\uparrow,\uparrow\rangle+c_6\,|\downarrow,\uparrow,\downarrow\rangle+c_7\,|\downarrow,\downarrow,\uparrow\rangle+c_8\,|\downarrow,\downarrow,\downarrow\rangle\end{aligned}$$

$$\longrightarrow \tag{2.20}$$

$$\Psi_{\text{vec}} = \begin{array}{c} \begin{array}{cccccccc} |\uparrow,\uparrow,\uparrow\rangle & |\uparrow,\uparrow,\downarrow\rangle & |\uparrow,\downarrow,\uparrow\rangle & |\uparrow,\downarrow,\downarrow\rangle & |\downarrow,\uparrow,\uparrow\rangle & |\downarrow,\uparrow,\downarrow\rangle & |\downarrow,\downarrow,\uparrow\rangle & |\downarrow,\downarrow,\downarrow\rangle \end{array} \\ \begin{pmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \end{pmatrix}^{\mathrm{T}} \end{array}$$

Rewriting Equation 2.6 in terms of the newly defined matrix operators, results in Equation 2.21.

$$\langle m|\,\mathcal{H}\,|n\rangle\cdot\Psi_{\text{vec}} = E\cdot\Psi_{\text{vec}} \tag{2.21}$$

This representation is quite useful, because now the problem of finding the eigenstates has been turned into a problem of finding the eigenvectors for a sparse (hermitian) matrix. A problem, that is already solved by a lot of algorithms and software [16]. It is now clear, that the smallest eigenvalue of the matrix *is* the ground state energy and the corresponding eigenvector *is* a representation for the ground state wavefunction.
As mentioned, the complexity of the problem does not lie in terms of how to compute this value, but in how to compute it in reasonable time. If $L$ is the size of the lattice, it is obvious, that the canonical base contains $2^L$ states ($L$ sites, each can be $\uparrow$ or $\downarrow$). Therefore the matrix whose eigenvalues are required has $2^{2 \cdot L}$ entries, of which $2^L \cdot m$ are non-zero ($m$ being constant and dependent on the form of the hamiltonian, as well as the number of neighbors to one site in the lattice).
So even *if* an algorithm was found, to compute the eigenvalues of such a matrix in linear time (in comparison to the number of non-zero entries in the matrix), the time complexity for solving the problem would still scale in $\mathcal{O}(2^L)$ with the number of lattice sites $L$.
To compute macroscopic crystals, the number of lattice sites would have to be in the same range of magnitude as *Avogadros constant* ($N_\mathrm{A} \approx 6.022 \times 10^{23}$ 1/mol [17]). Current hardware can not go reasonably into the range of hundreds.
[18]: `/computation/exact_diagonalization.py` shows an implementation of this naive algorithm. On moderately modern hardware it takes about 2.5 min to compute the ground state energy of a $L = 13$ lattice. $L = 16$ already takes hours.

### 2.1.4 Solutions with Neural Quantum States

As a direct numerical approach was not sufficient, more advanced methods needed to be developed. *Iterative* approaches have shown to be quite effective. In these kind of methods, the goal is not to compute a full solution directly, but to start with a random solution and perform iterative updates to it. If a good update strategy is chosen, the solution may converge towards the desired solution.
Because of the iterative nature, it is not necessary to do a complete computation at once. As learned in subsection 2.1.3, the memory requirement for storing a complete hamiltonian matrix representation is also exponential. So even with unlimited time, a diagonalization calculation could be impossible because of limited available memory. The iterative methods get around this, by being able to chose a data structure with much higher information density than the full hamiltonian. For example if only the ground state is wanted, it is unnecessary to store anything but information regarding it. The hamiltonian though encodes a lot of "unnecessary" information about other states. This structure can then be iteratively refined.
In practice a so called *neural quantum state* (NQS) is used. This basically is nothing but a neural network, taken from established machine learning tasks [19]. The function is parametrized by $M$ variables $\theta_i \in \mathbb{R}, 0 \leq i < M$ and takes a spin configuration as an input. A spin configuration for $N$ lattice sites can be described by $\sigma_j \in \{\uparrow, \downarrow\}, 0 \leq j < N$. The function will be written like in Equation 2.22.

$$\left|\Psi_{\theta_0,\ldots,\theta_{M-1}}(\sigma_0,\ldots,\sigma_{N-1})\right\rangle \equiv |\Psi_\theta\rangle \tag{2.22}$$

In order to compute a helpful representation, the identities in Equation 2.23 and Equation 2.24 are necessary [13]. $^*$ represents the *complex conjugation* operation. Equation 2.23 applies to *complete orthonormal* bases only.

$$\sum_s |s\rangle\langle s| = \mathbb{1} \tag{2.23}$$

$$\langle\Psi|\Psi\rangle = \Psi^* \cdot \Psi = |\Psi|^2 \in \mathbb{R} \tag{2.24}$$

The following method is described in [20].
To calculate the *expectation value* of a quantum mechanical operator $\hat{O}$ for the wavefunction $|\Psi\rangle$, the expression $\frac{\langle\Psi|\hat{O}|\Psi\rangle}{\langle\Psi|\Psi\rangle}$ is used [13]. The denominator assures, that the calculation is correct for *non-normalized* wavefunctions (as the NQS is initialized randomly it very likely is not normalized).
Applying this to the problem at hand, the target is to measure the observable corresponding to $\hat{O}$ in the parametrized neural quantum state $|\Psi_\theta\rangle$.

$$\begin{aligned}
\frac{\langle\Psi_\theta|\hat{O}|\Psi_\theta\rangle}{\langle\Psi_\theta|\Psi_\theta\rangle} &\overset{2.23}{=} \frac{\langle\Psi_\theta|\left(\sum_s |s\rangle\langle s|\right)\hat{O}\left(\sum_{s'}|s'\rangle\langle s'|\right)|\Psi_\theta\rangle}{\langle\Psi_\theta|\left(\sum_s |s\rangle\langle s|\right)|\Psi_\theta\rangle} \\
&= \frac{\sum_{s,s'}\langle\Psi_\theta|s\rangle\langle s|\hat{O}|s'\rangle\langle s'|\Psi_\theta\rangle}{\sum_s \langle\Psi_\theta|s\rangle\langle s|\Psi_\theta\rangle} = \frac{\sum_s \langle\Psi_\theta|s\rangle\frac{\langle s|\Psi_\theta\rangle}{\langle s|\Psi_\theta\rangle}\sum_{s'}\langle s|\hat{O}|s'\rangle\langle s'|\Psi_\theta\rangle}{\sum_s \langle\Psi_\theta|s\rangle\langle s|\Psi_\theta\rangle} \\
&= \sum_s \frac{\langle\Psi_\theta|s\rangle\langle s|\Psi_\theta\rangle\sum_{s'}\langle s|\hat{O}|s'\rangle\frac{\langle s'|\Psi_\theta\rangle}{\langle s|\Psi_\theta\rangle}}{\langle\Psi_\theta|s\rangle\langle s|\Psi_\theta\rangle} \overset{2.26}{=} \sum_s \frac{\langle\Psi_\theta|s\rangle\langle s|\Psi_\theta\rangle\hat{O}^\theta_{\text{loc}}(s)}{\langle\Psi_\theta|s\rangle\langle s|\Psi_\theta\rangle} \\
&\overset{2.3}{=} \sum_s \frac{\Psi_\theta(s)^*\cdot\Psi_\theta(s)\cdot\hat{O}^\theta_{\text{loc}}(s)}{\Psi_\theta(s)^*\cdot\Psi_\theta(s)} \overset{2.24}{=} \sum_s \frac{|\Psi_\theta(s)|^2\,\hat{O}^\theta_{\text{loc}}(s)}{|\Psi_\theta(s)|^2}
\end{aligned} \tag{2.25}$$

In the derivation, the *local estimator* (Equation 2.26) is introduced. Because the computational base is sparse, only very few elements of the sum over $s'$ in Equation 2.26 are non-zero [20]. A local estimator can therefore be evaluated in constant time.

$$\hat{O}^\theta_{\text{loc}}(s) = \sum_{s'}\langle s|\hat{O}|s'\rangle\frac{\langle s'|\Psi_\theta\rangle}{\langle s|\Psi_\theta\rangle} \tag{2.26}$$

The final result in Equation 2.25 has the form of a *probability distribution* over $s$. So the calculation of a quantum mechanical expectation value can be re-formulated as the calculation of the statistical expectation value of $s$ of the local estimator's values.

On its own, this is not helpful, because the size of the vector space of $|s\rangle$ is still exponentially large in regards to the problem size. But this shortcoming can be resolved with the introduction of *Monte Carlo* sampling. The *probability density* can be estimated, by looking at only a subset of possible $|s\rangle$.
By randomly sampling a large, but manageable number $N_{\mathrm{MC}}$ of states $s_{(n)}$ with the *Metropolis algorithm* [8], the true value can be approximated (Equation 2.27) [20].

$$\frac{\langle\Psi_\theta|\,\hat{\mathrm{O}}\,|\Psi_\theta\rangle}{\langle\Psi_\theta|\Psi_\theta\rangle} \approx \frac{1}{N_{\mathrm{MC}}}\sum_{n=1}^{N_{\mathrm{MC}}}\hat{\mathrm{O}}^{\theta}_{\mathrm{loc}}(s_{(n)}) \tag{2.27}$$

Therefore the energy of a system can be estimated with Equation 2.28. All the terms can be efficiently computed, using methods from *subsection 2.1.3 Solution through Exact Diagonalization*.

$$E(\theta) = \frac{\langle\Psi_\theta|\,\mathcal{H}\,|\Psi_\theta\rangle}{\langle\Psi_\theta|\Psi_\theta\rangle} \approx \frac{1}{N_{\mathrm{MC}}}\sum_{n=1}^{N_{\mathrm{MC}}}\mathcal{H}^{\theta}_{\mathrm{loc}}(s_{(n)}) \overset{2.26}{=} \frac{1}{N_{\mathrm{MC}}}\sum_{n=1}^{N_{\mathrm{MC}}}\left(\sum_{s'}\left\langle s_{(n)}\right|\mathcal{H}\left|s'\right\rangle\frac{\langle s'|\Psi_\theta\rangle}{\left\langle s_{(n)}\right|\left.\Psi_\theta\right\rangle}\right) \tag{2.28}$$

As it is known, that the ground state energy $E_0$ is the smallest energy, the ground state can be estimated in the following way:

1. Start with a randomly initialized NQS
2. Estimate its energy with Equation 2.28 and MC-sampled states
3. Calculate the neural network error, assuming $E(\theta)$ should be smaller (formula in [20])
4. Propagate the error back into the network, updating it to improve its performance
5. Repeat at 2. with the updated NQS

Because the method iteratively updates the network by performing tiny variations, it is called *variational Monte Carlo* (VMC) method.

### 2.1.5 Imaginary Time Evolution

Many different iterative methods are being employed to calculate the desired physical quantities. In this section, a mathematical idea will be discussed, that has several applications. The method is employed in the *diffusion Monte Carlo* method (DMC) [8], a second application of Monte Carlo sampling in comparison to VMC.
This section is about more advanced quantum mechanical principles and requires prior experiences. It is of supplemental nature and can therefore be skipped without sacrificing on the possibility to understand subsequent parts of the thesis.
The method is described in [21].
Equation 2.6 can be generalized to be no longer time independent. Equation 2.29 is called the *time-dependent Schrödinger equation*. $\hbar$ is the *reduced Planck constant*, $t$ is the time.

$$\mathcal{H}\,|\Psi(t)\rangle = i\hbar\frac{\partial}{\partial t}\,|\Psi(t)\rangle \tag{2.29}$$

In this case, the wavefunction also needs to depend on the time as a variable. For a time-independent hamiltonian, the time dependent wavefunction can be obtained via Equation 2.30 [13] with the energy eigenvalues $E_n$ ($E_0$ being the ground state energy), the energy eigenbase states $|n\rangle$ and the wavefunction at time $t = 0$, $|\Psi(0)\rangle$.

$$|\Psi(t)\rangle = e^{-i\mathcal{H}t/\hbar} |\Psi(0)\rangle \tag{2.30}$$

That Equation 2.30 is a solution to Equation 2.29 can be verified like in Equation 2.31 (notice that the hamiltonian is time-independent).

$$\begin{aligned} i\hbar\frac{\partial}{\partial t} |\Psi(t)\rangle &\overset{2.30}{=} i\hbar\frac{\partial}{\partial t} e^{-i\mathcal{H}t/\hbar} |\Psi(0)\rangle \\ &= \frac{-i \cdot i\mathcal{H}\hbar}{\hbar} e^{-i\mathcal{H}t/\hbar} |\Psi(0)\rangle \\ &\overset{2.30}{=} \mathcal{H} |\Psi(t)\rangle \end{aligned} \tag{2.31}$$

Equation 2.30 can be rewritten in terms of the energy eigenbase.

$$\begin{aligned} |\Psi(t)\rangle &= e^{-i\mathcal{H}t/\hbar} |\Psi(0)\rangle \\ &\overset{2.2,2.3}{=} \sum_n e^{-i\mathcal{H}t/\hbar} \langle n|\Psi(0)\rangle |n\rangle \\ &= \sum_n e^{-iE_n t/\hbar} \langle n|\Psi(0)\rangle |n\rangle \end{aligned} \tag{2.32}$$

In Equation 2.32, an operator gets applied from inside an exponent. This can be explained with the possibility to express the $e$-function in terms of its *Taylor series* [13]. This is important to understand the transition from $\mathcal{H}$ to $E_n$ in the exponent. For the transformation, the function is written as its Taylor series, the operator gets applied and then the Taylor series is reversed into the function. Remember that $\langle n|\Psi(0)\rangle$ is simply a complex number.
The method is called *imaginary* time evolution, because of the step in Equation 2.33.

$$\tau = i \cdot t \tag{2.33}$$

An "imaginary" time $\tau$ can be obtained by equation Equation 2.33. Switching variables in Equation 2.32 leads to the representation shown in Equation 2.34.

$$|\Psi(\tau)\rangle = \sum_n e^{-E_n\tau/\hbar} \langle n|\Psi(0)\rangle |n\rangle \tag{2.34}$$

$E_0 > 0$ can be assumed without loss of generality. It is known by definition, that $E_0 \leq E_j$, $j \neq 0$. Equation 2.34 contains only terms, that decay exponentially with $\tau$. Because of that, for large values of $\tau$ all terms decay to 0, but the $n = 0$ term decays the slowest. Equation 2.35 is therefore won.

$$\lim_{\tau\to\infty} \frac{|\Psi(\tau)\rangle}{e^{-E_0\tau/\hbar}} \overset{2.34}{=} \langle 0|\Psi(0)\rangle\,|0\rangle \tag{2.35}$$

This means, that if the starting wavefunction has an overlap with the ground state ($\langle 0|\Psi(0)\rangle \neq 0$), if the wavefunction is evolved in imaginary time, all contributions except the ground state contribution will be eliminated.
This can be used to find the ground state [8]. A random starting function is chosen and evolved in complex time with the aid of Monte Carlo sampling. The ground state can then be extracted. If there is no overlap between the starting function and the ground state, the method converges to the base state with the smallest energy that also has overlap.

### 2.1.6 Explored Lattice Patterns

In Figure 2.1 two examples of lattice-shapes were already introduced. In the research leading to this thesis, a selection of lattices was chosen. The implementation supports a 1D lattice (the default **linear** lattice/chain (Figure 7.6)) and five different 2D lattices (**square** (Figure 7.1), **hexagonal** (Figure 7.2), **trigonal_square** (Figure 7.4), **trigonal_hexagonal** (Figure 7.3) and **trigonal_diamond** (Figure 7.5)).
[18]: `/structures/visualize.py` allows for an interactive visualization of the lattice structure. The same code was used to generate the referenced visualization printed in the Appendix. The construction of the hexagonal lattice types, makes use of the hexagonal coordinate systems defined in [22].
The numbers, that can be seen in the visualization, denote the location-index of the memory, that is responsible for carrying the spin state at that lattice site in the later representation. As will become obvious in *subsection 2.2.4 Employment of Graphs for Problem Transfer*, the structure of a lattice cannot always be represented directly analogous in memory. Therefore an alternative way of addressing lattice sites and their surroundings needs to be employed. In this case, the framework generates the set of *nearest neighbor* (marked green in the visualization) and *next-nearest neighbor* (marked yellow in the visualization) indices for a given lattice site (marked red in the visualization). These are used as structural inputs for later discussed *graph* architectures.
A quite important feature for the physical aspect of the calculation is the *periodicity* of a lattice. As real world crystals have many magnitudes more lattice sites than can be currently simulated, *boundary effects* play a significant role in simulated lattices. This often is an unwanted manner, if the goal is to calculate the behavior of the system far away from the edges. In that region, lattices should have the property of *translational symmetry*. To aid with this, a *periodic* repetition of the lattice can be toggled. It is responsible for defining neighbors for lattice sites close to the edges of the lattice. The Lattice Visualization in the appendix shows this, by providing the information whether the pictured lattice is periodic or not.
The way in that the lattices are tiled should become clear from the visualization. It is noteworthy, that the **square**, **trigonal_square** and **trigonal_diamond** lattices have repetition along two axes, while the **hexagonal** and **trigonal_hexagonal** have three. The assumption is that the **trigonal_hexagonal** and **trigonal_square**/**trigonal_diamond** lattices show the same behavior

in non-periodic mode (as they all three represent a trigonal lattice), while differences manifest in periodic inspection.
Last, there is a setting to *randomly swap* lattice positions. The effect of this can be seen in Figure 2.2.

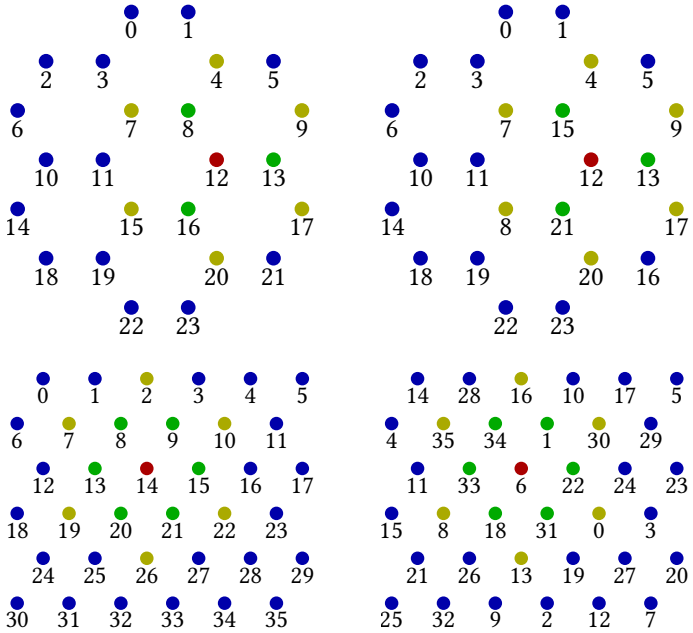


**Figure 2.2**: Visualization of the property *random swaps* of the lattice structure helper. In the first set of figures, two swaps were performed on a hexagonal lattice of size 2 (8 with 15 and 16 with 21). The neighbor-index calculation takes this into effect, therefore the highlighted lattice sites do not change. The second set of figures shows the setting -1 for a trigonal_square lattice of size 3. This basically makes the generator perform such a high number of pair swaps, that the index-position correlation can be assumed to be random.

The goal of this setting is to decouple the in-memory representation and the structure of the simulated lattice. Effects of this will be explored in *subsection 4.2.2 Resiliency to the Choice of Lattice Encoding*.
General data about the lattices is provided in Table 2.1.

| Lattice name | #nn | #nnn | $L(1)$ | $L(2)$ | $L(3)$ | $L(4)$ | $L(5)$ | $L(6)$ | $L(7)$ |
|---|---|---|---|---|---|---|---|---|---|
| linear | 2 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| square | 4 | 4 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
| trigonal_square | 6 | 6 | 4 | 16 | 36 | 64 | 100 | 144 | 196 |
| trigonal_diamond | 6 | 6 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
| trigonal_hexagonal | 6 | 6 | 7 | 19 | 37 | 61 | 91 | 127 | 169 |
| hexagonal | 3 | 6 | 6 | 24 | 54 | 96 | 150 | 216 | 294 |

**Table 2.1**: General information about the explored lattices. **#nn** describes the number of nearest neighbors, **#nnn** the number of next nearest neighbors. The numbers describe the maximum possible neighbor count. If a lattice is non-periodic, some sites can have less neighbors. $L(i)$ describes the number of lattice sites for a lattice of **size**=$i$

.

## 2.2 From the Perspective of Computer Science

The first sections gave a brief overview of the problem that needs to be computed, as well as a basic understanding of the performed calculations. The subsection 2.1.4 presented the NQS and promised, that its job would be performed by a *neural network*. This theory section will give a brief introduction into the basics of neural network training.

### 2.2.1 The Image Classification Task

It has been more than 30 years, since it was proven, that already simple neural network architectures can in fact be *universal approximators* [23]. That means that if a problem can be formulated as a mathematical function, this function can be approximated by a neural network. This has huge consequences, because if input and output are encoded digitally in a predefined format, *all* mappings between these can be formulated in form of a mathematical function. Be it only the trivial one, that maps each input encoding to the desired output encoding. Because of that, seemingly impossibly complex problems can be tackled nowadays.
Part of the experimentation in the thesis will focus on the application of a select breed of neural networks on the *image classification task*. This is not a novel problem, but one that has been tried and solved by many teams of researchers. The gist is to build a neural network that can label the things pictured on some kind of image. In our case an even simpler version of the problem will be considered. The only job of the network is to select one of $N$ predefined labels, that describes the picture best.
A generalization of the task pipeline can be viewed in Figure 2.3.
The aforementioned trivial mapping of inputs to outputs is obviously not feasable in reality, as for a $224 \times 224$ pixel 8-bit grayscale image the function would need to consist of $256^{224 \cdot 224} \approx 7 \times 10^{120\,835}$ cases, most of which do not even make sense in terms of the predefined categories or even define a realistically possible image.
In order to solve the problem, the mapping function is *approximated* with a neural network.
To train the network, a pre-labeled set of training images is passed through it, the results are compared to the labels and the error is used to iteratively update the network with use of the *backpropagation algorithm* [25], therefore improving the performance of the network in the next step.
A big challenge in training these kinds of networks is getting a sufficient amount of labeled training data. Today a lot of publicly available, high quality datasets exist for download. In this case, the *CIFAR-10* [26] dataset was used for the first steps in the methods. A subset of *100 classes* from the *ImageNet* [27] dataset was used in the experiments that were used to evaluate the general functionality of the explored network architectures. The subset of chosen classes can be viewed in [28]: `/managing_imagenet_data/used_synsets.txt`.
In the task the *accuracy* describes the percentage of validation images whose label was guessed correctly by the network. It is not allowed to use the validation images during training. That way, the network only would need to memorize a tiny amount of images to perform good on the test. This is undesirable behavior, as the general idea is to teach the network to be able

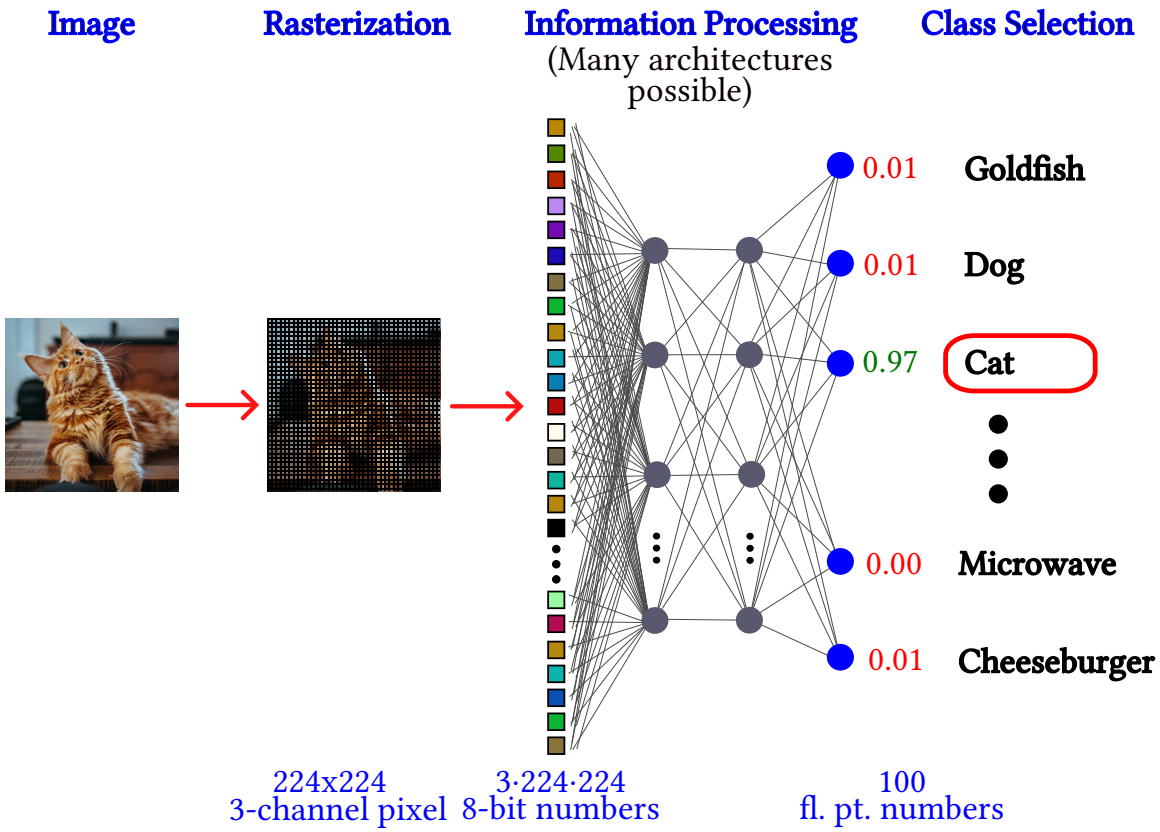


**Figure 2.3:** Generalized schema of the image classification pipeline used in the experiments section of this thesis. The image gets rasterized, resized to 224×224 pixel and 8 bit color depth per each of the three color channels. Then the image is normalized, reshaped into vector form and processed from there. The MLP in this image is only a placeholder. Many of the later discussed neural network architectures treat the reshaping, casting and processing differently. The output for all networks needs to be in form of 100 floating point numbers. The one closest to 1 gets selected as the networks choice. The class name the index represents can then be looked up. Cat image from [24].

to classify previously unseen images in the right way. The *training accuracy* is the analogous value, but calculated for the training images - that *are* used during training (Because of that, this accuracy is often way higher). The *top-3-accuracy* is the percentage of validation images, for that the correct label was among the top three highest valued choices of the network. The same is true analogously for the *top-5-accuracy*.

### 2.2.2 Neural Network Training

When initializing neural networks, they most often come only with an empty structure, that allows for the possibility to acquire a computing machine if all the internal parameters and weights are set accordingly. For very small computational networks, it is possible to set the weights by hand. But already for low tens or hundreds of numbers this becomes an unfeasible task.

Because of that, neural networks are *trained* to get them into a working state. In the training phase the structure of the network is (normally) left static, only the parameter values are

adjusted. The algorithm that accomplishes an update of the weights, that nudges the network *towards* the desired solution, is called *backpropagation* [25]. It is based on the chain rule of differentiation and uses the severity of changes a parameter inflicted on a (intermediate) result, to update the parameter in a way that steers said result towards the desired one.
As mentioned, for one run the structure is typically unchanged. The general shape of the network and many turning knobs on the update process can be controlled to allow for a more efficient representation and weight update. The controlling variables are called *hyperparameters*. The most important hyperparameters in the following experiments will be the *optimizer* (*stochastic gradient descent* and *adamw* [29] are compared), the *loss function* (*cross entropy loss* is used), and the *batch size*. Additionally, the *learning rate*, *weight decay*, *dampening* and *momentum* can be set to modify the behavior of the optimizer.
Different optimizers change the updating behavior of the network. The loss function defines a measure on how far or close the computed solution is to the known solution. The learning rate defines the step size that gets taken towards the target in one iteration. Choosing a value too small results in slow convergence, choosing a value too big can result in overshooting and subsequently also lowering the speed of convergence. Dampening and momentum are generally used to combat either of these effects. Weight decay generally helps against *overfitting*. If a model's generalization capacity is exhausted or the training examples are too monotone, this can lead to memorization of a subset of training examples. This increases the performance on training data, but hurts the performance on unseen data. As for our applications, overfitting is unimportant or even desired (see section 3.2), this parameter left unchanged.
The article *A ConvNet for the 2020s* [30] makes the point, that by heavily investing in choosing the appropriate hyperparameters, state of the art performance can be extracted even from "standard" model architectures. As the focus in this thesis is to compare the performance of different *architectures* with each other, *no efforts* are made to tune hyperparameters to achieve the absolute best performance. In contrary, hyperparameters were left *deliberately* static, to allow for a more fair comparison where possible.
One more important parameter is the batch size. By processing more than one training example at a time, a more stable and efficient update process can be achieved. However the batch size is limited by the available memory. As most of the calculations are performed on a GPU (*graphics* processing unit), the video memory is important. The calculations were either performed on a *NVIDIA GeForce GTX 1070* with 8 GB of video memory or a *NVIDIA TITAN Xp* with 12 GB. Both cards are mid to low range in performance and video memory, in comparison to the most up to date cards. Therefore often the batch size and number of iterations had to be set lower than may be desirable for an optimal result. Though it speaks in favor of the applied methods, that good results could be achieved without top-tier hardware.
Very important for generating training data for an image recognition task are the *augmentations*. They are a big focus of the research performed in [30] and [31]. As transformer models generally require really large datasets to train, it is desirable to generate more data from the data that is already collected. This can be achieved by augmenting the training images in a way that does not change the content according to human recognition, but produces a different input

for the network. Flipping, rotating, cropping or adding noise to an image are all possible augmentations. In the experiments only random horizontal flips are employed, to still be able to overfit all models. This is necessary to be able to judge the models *inductive bias* (section 3.2). To paint a more complete picture, there are other strategies, than just forcefully training a model directly from scratch. Some of those will be be presented in the following section.

### 2.2.3 Neural Network Pre-Training

For a human it is often complicated to jump into an unfamiliar task and learn everything necessary to solve it at once. The same is true for neural networks. Different strategies have been devised to combat this difficulty. None of them is applied in the experiments section of this thesis. However it may be fruitful to look into their application in further research.
One method is to *distill* knowledge from one network into an other. For that strategy, first a large and expensive network could be trained. Then a smaller network is trained with the task to replicate the behavior of the large network. This method among other things is discussed as a complementary measure in [7]. The idea is, that by forcing the information into less weights, a higher level of abstraction and problem extrapolation can be achieved, and pure memorization of the solutions is less likely.
An "online" approach to this method are *teacher-pupil* methods. By training multiple networks in parallel and motivating them to outperform each other, a better performance overall can be achieved. This is an important point in [31].
Finally, very large performance gains can be won from the employment of *pre-training*. This method resembles the human learning strategy to first acquire general knowledge about a problem by segmenting it into basic blocks and practicing on them. Then learning to refine the general knowledge into one that solves a specific task.
Pre-training in *language processing* is a key aspect of the research in [32]. A neural network could for example be trained to reconstruct a sentence, of which one word has been masked out. Generating labeled training data for such cases is basically trivial (taking text from the internet, masking words and remembering what was masked as the solution). Therefore the neural network can learn a lot about word proximity and basic sentence structures.
Because it is a highly labor intensive process to acquire texts with verified and suited translations alongside them, there will always be a shortage of such data. A network with pre-trained understanding of sentence structure and linguistic mannerisms can extract knowledge from this limited dataset more efficiently and quickly. The process of adapting a pre-trained network to a desired task is called *fine-tuning*.
The datasets in pre-training are generally larger and easier to generate. For example for images, letting a network *cluster* a vast amount of images into a predefined number of categories will force it to extract information about similarities and regularities among the pictures, while not requiring the image to be labeled. This approach therefore is called an *unsupervised* pre-training [31]. The clustering may not group the images into the same categories as a human wants the network to group them, but it is basically certain that important features must be taken into account somehow. A smaller, labeled training set can then be used to fine-tune the

networks prediction to reflect the desired output mapping. Pre-training is integral to the use of transformers for image recognition [33].
However as this thesis focuses on the performance of different types of architectures and not on the extraction of maximum efficiency from one network, no pre-training is used in the following experiments.

### 2.2.4 Employment of Graphs for Problem Transfer

When using data for training neural networks, a consistent *encoding* has to be used. In Figure 2.3 an image is transformed into a neural network input by reshaping its pixels into one column of numbers. This also works for images with multiple color channels. A highly useful property of images is, that they can directly be encoded into *tensors* depending on their format (e.g. a $224 \times 224$ matrix for a grayscale image or a $3 \times 224 \times 224$ tensor for an image with three color channels).
By leaving this structure in the way it already is stored in, properties like the pixel-proximity relation is conserved (while reordering into one column obviously does not preserve this information in the same way). Network elements like *convolutions* (subsection 3.1.2) can take advantage of this preserved information. The utilization of such kind of information is called an *implicit bias* and discussed in detail in section 3.2.
The canonical representation of images in memory is also the reason, why GPUs (*graphics* processing units) are highly optimized to run calculation on tensors. Because of their hardware design, operations like matrix multiplication or tensor addition can be carried out highly parallelized and efficiently. The hardware was originally designed to render graphics to the computer screen. It is however no longer used exclusively for that. If a computational problem can be reduced to solving a matrix calculation, it can easily be carried out by a GPU.
But not all data is generally representable in tensor form. A common data structure - in that many nodes exist which have relations defined between them - is called a *graph*. Examples for graph data are social media friendship relations or street data inside a navigational system. As graphs are quite versatile however, a lot of things can be represented with them.
The novel concept is, to represent the lattice structures from subsection 2.1.6 as a graph and then apply neural network architectures, that are specifically designed to deal with that kind of data.
A big advantage is, that as graphs are more versatile than the strict tensor representation, the tensor data of an image can *also* be represented as a graph. A square picture can for example be represented as a graph, if every pixel is treated as its own node. By connecting the neighboring pixels with edges, the problem can be transferred into a graph representation, without losing the information about the proximity of pixels in relation to each other.
Established operations like *pooling* (subsection 3.1.1) or *convolutions* (subsection 3.1.2) can be defined to work on graph data. That will be done in subsection 3.1.5. The advantage of the possibility to represent images in both encodings is, that the functionality of the graph operators can be verified and compared against the established implementations for tensors. That way,

one can be confident, that the calculations will also work for data that is *not* representable in a tensor form.
As a graph with one node per pixel of an image would be quite large - even for low resolution images - larger chunks of the image, called a *patch* (e.g. $16 \times 16$ pixel) can be treated as one unit. With that method, localized interactions can be computed inside a patch, while mid to far range interactions can be computed between nodes/patches, while at the same time drastically cutting down the number of nodes in the graph.
Patching has already been used extensively in *vision transformer* architectures (subsection 3.1.3) [31, 33]. The graph-versions of these transformers are however relatively uncommon.

# 3 Machine Learning Architectures

## 3.1 Used Architectures

In order to be able to compare different machine learning architectures, it needs to be clear what the architectures are all about. For this reason, a brief introduction to neural networks is given. As there is quite a lot of literature on the basic forms of networks, the focus of this section will be explaining the more complex mechanisms. Basic architectures however, will only be briefly named.

### 3.1.1 Perceptron and Pooling Architectures

In the most basic sense, all neural network operators need to be able to take in a set of numbers and perform calculations on them, then provide an output. The shape can change from input to output, depending on the operator. Some operators, like the *perceptron* (or its conceptual successors, the *fcl* and *MLP*) have trainable weights that get updated like described in subsection 2.2.2. Others, like the *pooling operator* transform the data according to static rules, without trainable parameters. A neural network can't be trained, if it doesn't contain operators with adjustable weights. However operators like pooling are very efficient in supporting trainable operators (as will be shown later).

**The perceptron** is the most basic building block of all neural network architectures. A schematic view is given in Figure 3.1.

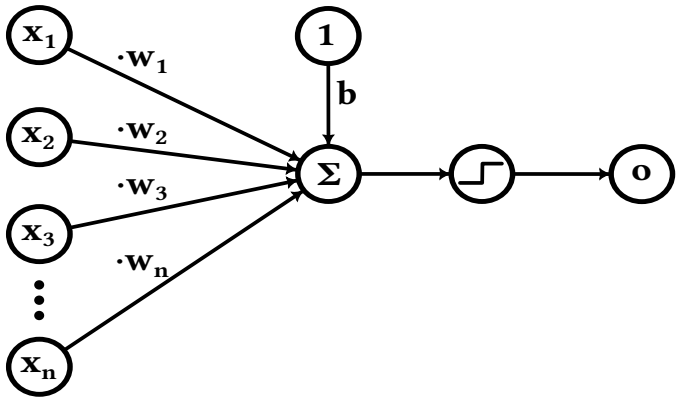


**Figure 3.1**: Schematic representation of a perceptron. Inputs are denoted with $x_i$, weights with $w_i$, the bias value with $b$ and the output with $o$. The inputs get scaled with the weights, summed and the bias is added. Then the value is passed through a not specified non-linear function (e.g. ReLU or the sigmoid function).

The output gets calculated like $o = \text{ReLU}\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right)$. Calculating multiple perceptrons with different weights in parallel for the same input values $x$ is called a fully connected layer (fcl). The formula for the elements in a fcl is show in Equation 3.1,

$$o_j = \sum_{i=1}^{n} x_i \cdot w_{i,j} + b_j \tag{3.1}$$

which is equivalent to a matrix-vector-multiplication. Therefore the fcl can be evaluated quickly. Figure 2.3 shows multiple fcls in series, which is called a multi layer perceptron (MLP).

**The pooling operator** is basically a function that selects a new value based on values in a specific range around the target position. The rules can range from taking the *minimal* value, to the *maximum* value or the *average*. In this thesis, all pooling operations are average pooling.

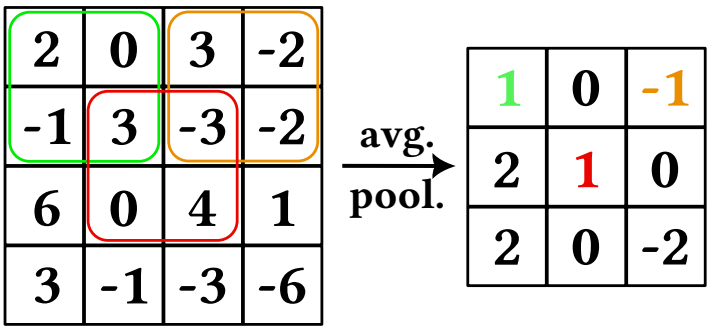


**Figure 3.2:** Example for a 2×2-average-pooling operation. The colors indicate what numbers get averaged. Not all blocks are colored, but the calculation is performed analogously.

Figure 3.2 shows an example of a 2×2-average-pooling operation. Where the dimension of input and output need to stay the same, the border of the input is padded with zero values (no padding in this example).

### 3.1.2 Convolutional Architectures

The convolution operation is a combination of concepts from the fcl and pooling. Like in pooling, values get aggregated from inside a specific region of influence. This region is called the *kernel*. They do however not get treated uniformly, but scaled with (learnable) weights before being summed up. The weights are stored as entries in the kernel. The operation is equivalent to the discrete mathematical 2D-convolution, however due to convention the kernel is flipped before the calculation in most mathematical definitions [34] and is not flipped in most machine learning frameworks. This has to be taken into account for non-symmetric kernels.
The same as for pooling, the sides are padded with zeros if the input size is supposed to match to output size. An example convolution (un-padded) is calculated in Equation 3.2, including the sample calculation for one of the values.

$$\begin{pmatrix} 2 & -3 & 1 & -4 \\ 1 & 2 & 1 & -2 \\ -3 & 2 & -3 & 2 \\ -1 & -1 & 1 & 4 \end{pmatrix} \odot \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 9 & y & 5 \\ -5 & 2 & -1 \\ -8 & 4 & 1 \end{pmatrix} \tag{3.2}$$
$$y = (-3) \cdot 1 + 1 \cdot (-1) + 2 \cdot 2 + 1 \cdot 1 = 1$$

**Full convolutions** are the kind of convolutions, for that the kernel's *channel* dimension $c$ matches the channel dimension of the input values. Images - that are one of the primary targets for convolutions - typically have not only a width and a height, but also a channel dimension (e.g. color channels, like rgb). Because of that, images are most of the time stored as 3D-tensors $(c \times h \times w)$. A 3D-convolution with a $(c \times o \times k_h \times k_w)$ kernel, produces an output of shape $(o \times h \times w)$ (with appropriate padding, which is always assumed from this point onward). The dimension $o$ can be seen as the number of kernels, that get independently applied and later stacked. The leftmost case in Figure 3.3 shows such a convolution with $h = w = 8, k_h = k_w = 3, c = 3$ and $o = 1$.

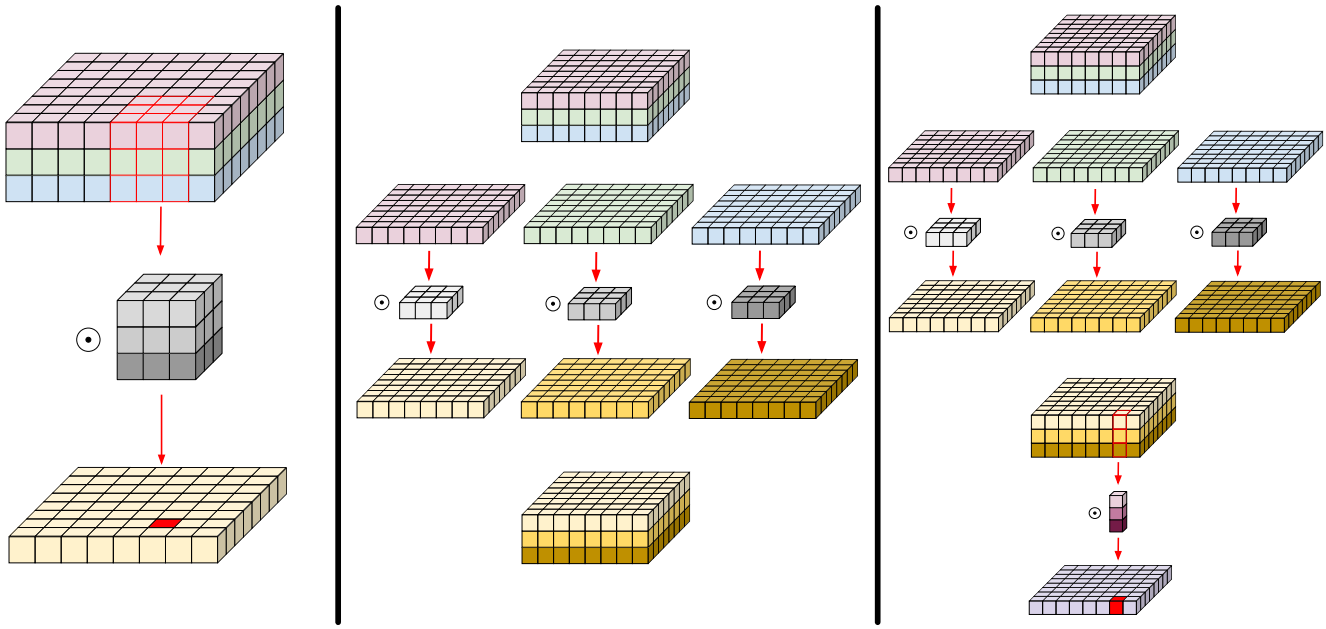

**Figure 3.3**: This figure shows the differences between a *full convolution* (left), a *depthwise convolution* (center) and a *depthwise separable convolution* (right). The full convolution has a 3D-kernel. The depthwise convolution splits it into multiple 2D-kernels. The depthwise separable convolution works basically in the same way as the depthwise, but at the end a 1D-kernel (1×1-convolution) is used to combine the channels. Original images from [35], but modified and almost completely redrawn.

**Depthwise convolutions** are similar to full convolutions at a first glance (such a convolution is pictured in the center of Figure 3.3). They employ an independent $(1 \times 1 \times k_h \times k_w)$ kernel for each channel and stack their results. The output channel count $o$ can only be a whole number multiple of the input channel count (one in the figure, but it is easy to use for example

two filters per channel). Other groupings of channels are possible [36]. By itself, this model computes no interaction across input channels. This seems like a disadvantage, but in reality is exact what will be needed for the construction of a *strict convolution based metaformer* (see subsection 3.1.4).

**Depthwise separable convolutions** are the logical continuation to depthwise convolutions and were made popular in image processing neural networks with the introduction of the *MobileNets* [7]. They get rid of two limitations of the depthwise convolutions: the number of output channels can be chosen freely and the information exchange is not limited to the channels in isolation. This is done by following up the $(1 \times 1 \times k_h \times k_w)$ convolutions with a $(c \times o \times 1 \times 1)$ convolution, that brings the number of output channels $o$ up to any desired number ($o = 1$ on the right side of Figure 3.3). The advantage is the reduction of the kernel size. This makes for easier computation and smaller model storage footprints. The weight count is compared in Table 3.1.

| Convolution Type | Formula #weights | # for $o = 6, c = 3,$ $k_h = k_w = 3, s = 3$ | # for $o = 6, c = 6,$ $k_h = k_w = 4, s = 4$ |
|---|---|---|---|
| full | $o \cdot (c \cdot k_h \cdot k_w)$ | 162 | 576 |
| depthwise | $(o//c) \cdot c \cdot (1 \cdot k_h \cdot k_w)$ | 54 | 96 |
| depthw. separable | $c \cdot (1 \cdot k_h \cdot k_w) + o \cdot (c \cdot 1 \cdot 1)$ | 45 | 132 |
| full symm. | $o \cdot (c \cdot s)$ | 54 | 144 |
| depthwise symm. | $(o//c) \cdot c \cdot (1 \cdot s)$ | 18 | 24 |
| depthw. sep. symm. | $c \cdot (1 \cdot s) + o \cdot (c \cdot 1 \cdot 1)$ | 27 | 60 |

**Table 3.1**: Comparison of the number of weights needed for the different convolution types. $o$ is the number of output channels, $c$ the number of input channels, $k_h$ and $k_w$ denote the kernel size and $s$ is the number of different tracked distance-weights in a symmetric case. Note that the depthwise symmetric case is always the most efficient in terms of the number of parameters. This is also the case, that gets employed in the metaformer, because there the mixing across channels is actually not needed (depthwise separable and full convolutions provide that mixing). For $o >> c$, the depthwise separable convolution will always require less weights than the depthwise.

**Symmetric convolutions** are a further extension to the convolution concept. They employ *shared weights* in the kernel. That means not every kernel entry needs to be stored separately. An example for a typical $3 \times 3$ symmetric kernel is shown in Equation 3.3.

$$\begin{pmatrix} w_2 & w_1 & w_2 \\ w_1 & w_0 & w_1 \\ w_2 & w_1 & w_2 \end{pmatrix} \tag{3.3}$$

This $3 \times 3$ kernel only needs three weights to be stored. While implementing the forward pass for such kernels is trivial, it is not quite obvious how to achieve an efficient calculation of the error gradient needed for the backpropagation. Implementations of symmetric cases exists [37], but not in the same format as Equation 3.3. The error values from convolutions also get computed through a convolution operation [38]. The machine learning frameworks probably perform convolution backpropagation way more efficient, than it was possible to implement the mathematics to propagate the correct error term into the shared weights by hand.

As a solution, the convolution kernel can be assembled from multiple kernels that can be scaled individually and added after the convolution is performed for each of them. This computes the correct convolution and can be backpropagated by the frameworks by default. The calculation is shown in Equation 3.4.

$$\begin{aligned} \overleftrightarrow{X} \odot \begin{pmatrix} w_2 & w_1 & w_2 \\ w_1 & w_0 & w_1 \\ w_2 & w_1 & w_2 \end{pmatrix} = \\ w_0 \cdot \overleftrightarrow{X} \odot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} + w_1 \cdot \overleftrightarrow{X} \odot \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} + w_2 \cdot \overleftrightarrow{X} \odot \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{aligned} \tag{3.4}$$

The implementation of the symmetric (depthwise) convolution for images can be found at [28]: `/models/helpers/SymmConv2d.py`.

### 3.1.3 Attention and the Transformer

In contrast to the decade old neural network architectures like MLPs or convolutions, the *transformer* architecture was only just introduced in 2017 with the works of *Attention Is All You Need* [5]. There, the *scaled dot product attention* was established as a highly performant operator for use in *natural language processing* (primarily - but not limited to - *translation*). In combination with other neural network components, the attention module was combined into the transformer. The original transformer was designed for language tasks and therefore had several design elements that were implemented specifically to compute in that domain.
In the followup paper *An Image is Worth 16x16 Words* [33] the *vision transformer* was introduced and has since been improved upon numerous times [39]. As the transformer used in the experiments of this thesis is only a special case for the later introduced *metaformer*, the transformer architecture will not be explained separately. This section will instead only introduce the attention calculation and related concepts.

**Attention** describes an algorithm, that allows a neural network to *attend* to every presented bit of information at once, each scaled by learnable factors that depend on the currently processed input. Specifically, the *scaled dot product attention* mechanism will be discussed.
The algorithm for *one attention head* is shown in Figure 3.4. During the experiments, the transformers will use *multi head attention*. The algorithm is exactly the same, but the calculation is repeated for multiple "heads" that all calculate the results independently with their own set of weights. Because of that, different heads can attend to the information with different strategies.
The algorithm starts with step 1, presented in Equation 3.5. The input $x$ is used to calculate the queries $Q$, the keys $K$ and the values $V$, by matrix multiplying with the respective learnable weights $W^q, W^k$ and $W^v$.
Step 2 (Equation 3.6) is calculating the matrix $QK^{\mathrm{T}}$, by evaluation the "outer product" between $Q$ and $K$ (outer product in regards to the number of patches $n$). As for big problem sets, $n$ can

get quite large ($e$ is a fixed hyperparameter), this step is quite likely to be comparably expensive, as it has a complexity of $\mathcal{O}(n^2)$. Especially for images, as $n$ doesn't describe the side length of a picture, but $n \propto h \cdot w$. $QK^T$ is a very important matrix, because it is a measure for the "overlap" of every key with every query. This is the core concept of the attention mechanism. The elements in $QK^T$ are dot products of queries $q \in Q$ and keys $k \in K$. If $q$ and $k$ are *parallel* - or in other words if the query and key align - the resulting output is going to be large. If they are *orthogonal* - speaking this is not the key the query is asking for - the value is going to be close to zero.
The steps 3 and 4 (Equation 3.7) are *scaling* the values and taking the *softmax*. This is to get *interaction probabilities* out of the previously calculated quantities.
The last step 5 is listed in Equation 3.8. The answer to the attention process $A$ is calculated as a weighted (with the scaled interaction matrix $M'$) sum of the values $V$ for each of the $n$ patches separately. This means for each of the $n$ entries of $x$, a *query* is produced that is checked against all the other $n$ *keys*. The degree of similarity determining the proportion, the corresponding *value* is going constitute to the answer.

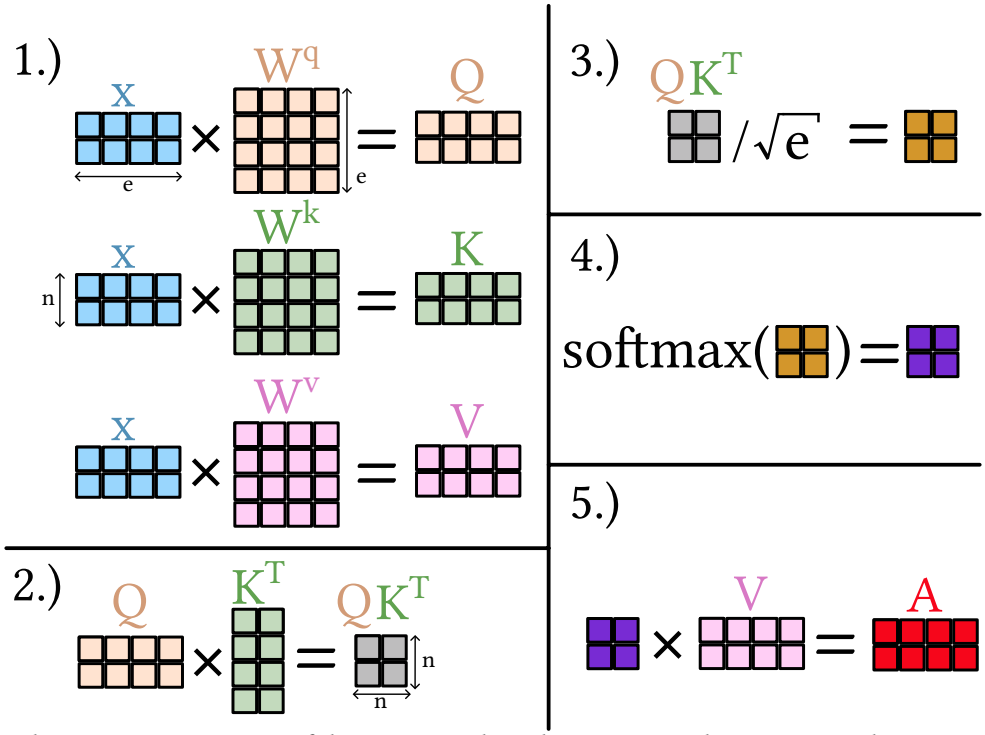


**Figure 3.4:** Schematic representation of the attention algorithm to picture the interacting dimensions. Only one attention head is pictured. In this example, the number of patches $n = 2$ and the embed dimension $e = 4$. Figure inspired by [40], but modified.

$$Q = x \cdot W^q \qquad K = x \cdot W^k \qquad V = x \cdot W^v \tag{3.5}$$

$$M = Q \cdot K^T \tag{3.6}$$

$$M' = \text{softmax}\left(\frac{M}{\sqrt{e}}\right) \tag{3.7}$$

$$A = M' \cdot V \tag{3.8}$$

The implementation of the algorithm for multiple heads in parallel is appended in section 7.2.

**Positional encoding** is an additional module, that can be used to boost the performance of a transformer. The method works by (additively) enriching the embedded tokens with "metadata", that gives the transformer vital information about the original location of the tokens.
During the attention computation, the model can take in information from any location at the same time, whilst there are no discernible differences between two patches of the same content but from different origins. As it is clear, that the same word can have a different meaning, depending if it is located at the start or the end of a sentence, this is an obvious drawback of the attention module. For convolutions the direction and distance (and for pooling at least the distance) that a bit of information has traveled can be inferred, depending on the module's depth and size of the kernel. As this is impossible in the attention calculation, the encoding is used to provide this helpful information to the model.
The positional encoding is an unique vector that represents a position in the input sentence/image. It can either be learned as part of the trainable parameters (implementation reference: [41]) or be predefined (fore example a *sinusoidal encoding* - implementation reference: [42]). The encoding either gets appended or added on top of the embedded token's value.

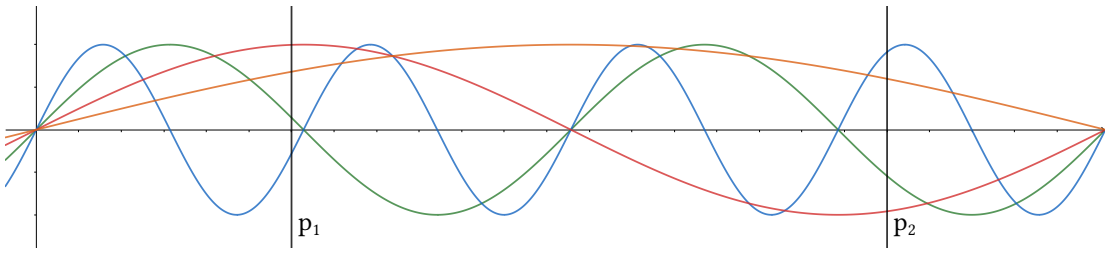


**Figure 3.5:** Visualization of the sinusoidal positional encoding for an one-dimensional structure. The curves are sinusoidal waves with different frequencies. The encodings can be sampled at arbitrary positions, and always give out a different encoding vector (if the frequencies are chosen accordingly). The distance of the sampled positions can be extracted from the encoded vectors.

In Figure 3.5, a sinusoidal positional encoding is visualized. The corresponding encoding values are presented in Equation 3.9.

$$e(\mathrm{p}_1) = \begin{pmatrix} -0.279 \\ 0.141 \\ 0.997 \\ 0.682 \end{pmatrix} \qquad e(\mathrm{p}_2) = \begin{pmatrix} 0.913 \\ -0.544 \\ -0.959 \\ 0.598 \end{pmatrix} \tag{3.9}$$

### 3.1.4 The Metaformer

The research performed in the paper *MetaFormer is Actually What You Need for Vision* [6] has been very influential in advertising the potential of the *transformer architecture*. Previously, the model's performance was attributed to the attention module, that was introduced in the

previous subsection 3.1.3. By building alternative models - that shared the structure of the transformer in everything but the attention module - the researchers were able to show that the credit mainly belongs to the structural architecture.
A new class of models, dubbed *metaformer* by the authors, was introduced. The architecture's good performance can be motivated through its inductive biases, which will be addressed in subsection 3.2.2. The following section will introduce the naming scheme that is going to be used in the later parts of this thesis.
The general structure of the metaformer can be seen in Figure 3.6.

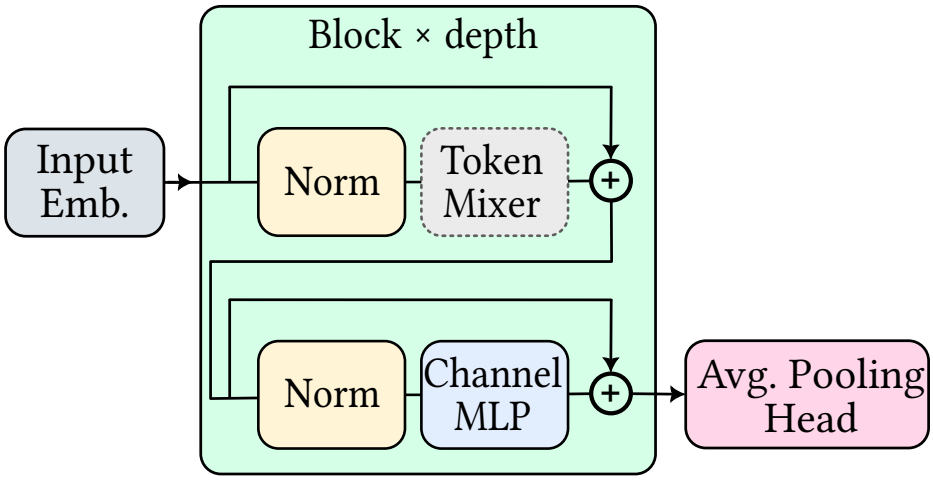


**Figure 3.6**: Schematic representation of the *metaformer* concept architecture, adapted to represent the models used in the thesis. The model starts with an input embedding block, that converts the input data into the shape required for the metaformer ($n \times e$) [33]. This is also the place where possible position encodings are added. After that, the information is passed through *depth* blocks sequentially. A block consists of *norm*, *token mixer*, *residual connection* (+), *norm*, *channel mlp*, *residual connection* (+). The model's internal state is brought to the desired output dimension with an *average pooling* operation to get from $n \times e$ to $1 \times e$ and a mlp to get to the desired output dimension $1 \times d_{\text{out}}$.

It is important to understand, that the *token mixer* is not a predefined element, but a location where multiple different blocks can be inserted. This can be seen in Figure 3.7 for a lineup of some special token mixers.
The job of the token mixer is to exchange information between the $n$ tokens. While the job of processing the information inside a token's embedding (of size $e$) is performed by the *channel mlp*. The *norm* block is typically a *layer norm*, that is introduced to keep the magnitude of the weights inside a reasonable range.
The name of the metaformer results from the name of the used token mixer. An overview is given in Table 3.2
The table contains an overview of all the metaformer variants that are examined in this thesis. However other combinations are possible. *GDCF*, *SCF-NN* and *SCF-NNN* are not implemented in the accompanying code [18, 28].
How the graph-equivalent modules *graph (depthwise) convolution*, *graph pooling* and *graph masked attention* can be constructed, is described in subsection 3.1.5.

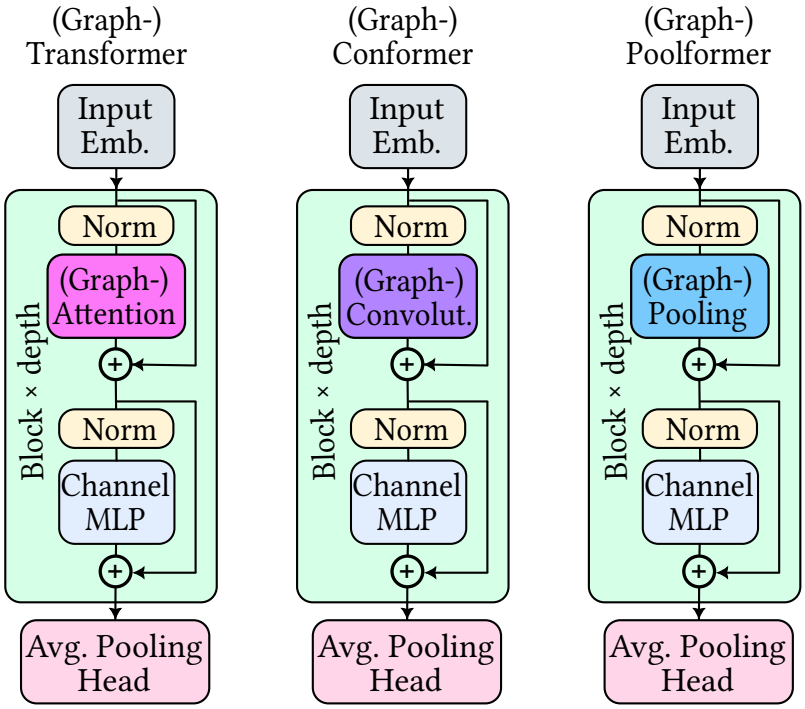


**Figure 3.7**: Different possible instances of the metaformer architecture presented in Figure 3.6. All structural elements are equivalent to the respective ones in that figure. Only for the placeholder *token mixer*, an *attention*-, *convolution*- and *pooling*-module were inserted, to respectively generate a *transformer*, *conformer* and *poolformer*. As different types of convolutions can be defined, the shown models are still technically *classes* of models, but the figure should be enough to convey the concept of token mixer swapping. It is important to understand, that the different models' *only difference* is the token mixer they are using. The graph versions of the operations are discussed in subsection 3.1.5.

| Token mixer | Name | Short | Strict |
|---|---|---|---|
| sdp attention | (Vision-)Transformer | TF | yes |
| attention, masked to nn | Graph (Vision-)Transformer NN | GTF-NN | yes |
| attention, masked to nnn | Graph (Vision-)Transformer NNN | GTF-NNN | yes |
| pooling | Poolformer | PF | yes |
| graph pooling, nn | Poolformer | GPF-NN | yes |
| graph pooling, nnn | Poolformer | GPF-NNN | yes |
| full convolution | Conformer | CF | no |
| symm. convolution, nn | symm. Conformer NN | SCF-NN | no |
| symm. convolution, nnn | symm. Conformer NNN | SCF-NNN | no |
| depthw. convolution | dConformer | DCF | yes |
| symm. depthw. convolution, nn | symm. dConformer NN | SDCF-NN | yes |
| symm. depthw. convolution, nnn | symm. dConformer NNN | SDCF-NNN | yes |
| depthw. graph convolution | graph dConformer | GDCF | yes |
| symm. depthw. graph conv., nn | symm. graph dConformer NN | SGDCF-NN | yes |
| symm. depthw. graph conv., nnn | symm. graph dConformer NNN | SGDCF-NNN | yes |

**Table 3.2**: Overview of the most important metaformer networks. If a network mixes along the *embed dimension* during the token mixing, it is not deemed strict. For images or other data stored in rectangular tensor form, the graph- and non-graph-version of poolformer and conformer are equivalent. Only the graph variants use the localization bias of data that can not be canonically stored in tensor form.

An exchange of the token mixer module can have different benefits. Some can use specially tailored inductive biases to aid the model with its calculation. Others vary in their storage or computational requirement. Even more specialized architectures can be attained, if hyperparameters like the embed dimension or the choice of token mixer is varied across the depth of the metaformer. Starting with a poolformer of high embed dimension for the first blocks for feature extraction and then reducing the embed dimension to employ a high performance attention module for the final evaluation can result in higher performance than both of the models on their own [6]. Hybrid approaches are not discussed in this paper's experiments, but could be a target for future research.
A further modification is shown in Figure 3.6 for the last element. An *average pooling* operator is used in conjunction with a *MLP* to reshape the output to dimension necessary for the task. In the original vision transformers, a *classifier token* was used [31], as this was motivated by the language processing origins of the transformer. Experiments (during the writing of this thesis) showed, the vision transformer performing better on our use cases, if the average pooling head was used, instead of the classifier token. Therefore the average pooling head is included as a fixed component of the metaformer architecture in this thesis.

### 3.1.5 Graph Architectures

As motivated in subsection 2.2.4, there are benefits to encoding a problem in a graph structure. In subsection 3.2.3 the advantages and possible disadvantages of graph architectures are going to be discussed. This section however will focus on the method that is going to be used to implement compatibility for graph encodings and the equivalent operators to *convolutions*, *pooling* and *attention* for graph networks.
The fundamental mathematical definition of how operations on relationships of graph nodes can be expressed, is defined in [11]. However the mathematical summation notation is used. As GPUs are optimized for the calculation of tensor operations, a way to express these operations is chosen, that utilizes matrix operations. That way the calculations can be performed highly parallelized on GPU hardware.

**The adjacency matrix** is the central element to the methods and is therefore going to be introduced first. The graph pictured in Figure 3.8 is described by the adjacency matrix $A$ in Equation 3.10.

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{3.10}$$

In the adjacency matrix, the element $A_{ij}$ is set to 1, if the graph has an edge between the nodes with index $i$ and $j$. All $A_{ii}$ here are set to 0, because generally nodes do not need to be "connected" to themselves.

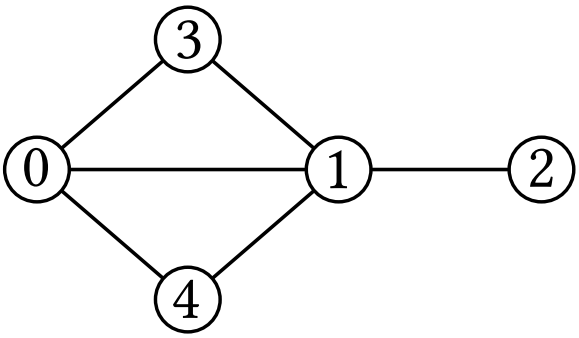


**Figure 3.8**: Graphic representation of a graph with five nodes and six edges.

As the graphs inside the neural networks however describe the flow of information, it is quite sensible to assume that the information from one node in one step should somehow be taken into account when generating the information for the next step. Because of that, it is useful to define multiple adjacency matrices, each with a different purpose.
In Equation 3.11, the adjacency matrices for *self-reference*, *nearest-neighbor-reference* and *next-nearest-neighbor-reference* are shown, referring to the graph in Figure 3.8. By chance almost every node happens to be a nnn for every node. Only 1 and 2 do not have this property. For graphs of bigger size, the matrices are generally quite likely to be highly sparse.

$$A_{\text{self}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad A_{\text{nn}} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \quad A_{\text{nnn}} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{3.11}$$

To simulate the information exchange at different rates, the matrices can be weighted with different factors. Equation 3.12 shows an adjacency matrix that allows for information to be taken in from the nn, while valuing the current value of the node proportionally higher.

$$A_{\text{weighted}} = 0.9 \cdot A_{\text{self}} + 0.2 \cdot A_{\text{nn}} + 0 \cdot A_{\text{nnn}} = \begin{pmatrix} 0.9 & 0.2 & 0 & 0.2 & 0.2 \\ 0.2 & 0.9 & 0.2 & 0.2 & 0.2 \\ 0 & 0.2 & 0.9 & 0 & 0 \\ 0.2 & 0.2 & 0 & 0.9 & 0 \\ 0.2 & 0.2 & 0 & 0 & 0.9 \end{pmatrix} \tag{3.12}$$

Note that as the number of neighbors is not fixed and the accumulated values get added together, the result generally will vary in magnitude depending on the amount of connected nodes in the relations.
The *averaging adjacency matrix* $\tilde{A}$ gets rid of this behavior. To generate $\tilde{A}$, the matrices $D$ and subsequently $D^{-\frac{1}{2}}$ (Equation 3.13) are needed. $D$ is the matrix, that counts to number of edges on the corresponding nodes (still for the graph in Figure 3.8). Raising a square matrix to a negative fractional power can not only be defined, but even efficiently computed for purely diagonal matrices. The calculation is performed element-wise. Because the matrix has no entries aside

from the main diagonal, the expected properties for this inverse are satisfied for the normal matrix multiplication.

$$D = \begin{pmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix} \qquad D^{-\frac{1}{2}} = \begin{pmatrix} \frac{1}{\sqrt{4}} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{5}} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{3}} \end{pmatrix} \tag{3.13}$$

When studying Figure 3.8, one might notice the edge count is equivalent to the number of edges +1 in Equation 3.13. This can be explained by the fact that the self-connection this time is counted ($A = 1 \cdot A_{\text{self}} + 1 \cdot A_{\text{nn}} + 0 \cdot A_{\text{nnn}}$). By adapting the relations represented by the adjacency matrices or the weighting factors of Equation 3.12, the matrix can be tuned to suit a wide range of intended calculations. It is however important, to make sure that every node has at least one neighbor counted in $D$, or a division by zero will occur during the calculation of $D^{-\frac{1}{2}}$, crashing the neural network.
$\tilde{A}$ can then be calculated according to Equation 3.14.

$$\tilde{A} = D^{-\frac{1}{2}} \cdot A \cdot D^{-\frac{1}{2}} \overset{(Example)}{=} \begin{pmatrix} \frac{1}{4} & \frac{1}{\sqrt{5}\cdot\sqrt{4}} & 0 & \frac{1}{\sqrt{3}\cdot\sqrt{4}} & \frac{1}{\sqrt{3}\cdot\sqrt{4}} \\ \frac{1}{\sqrt{4}\cdot\sqrt{5}} & \frac{1}{5} & \frac{1}{\sqrt{2}\cdot\sqrt{5}} & \frac{1}{\sqrt{3}\cdot\sqrt{5}} & \frac{1}{\sqrt{3}\cdot\sqrt{5}} \\ 0 & \frac{1}{\sqrt{5}\cdot\sqrt{2}} & \frac{1}{2} & 0 & 0 \\ \frac{1}{\sqrt{4}\cdot\sqrt{3}} & \frac{1}{\sqrt{5}\cdot\sqrt{3}} & 0 & \frac{1}{3} & 0 \\ \frac{1}{\sqrt{4}\cdot\sqrt{3}} & \frac{1}{\sqrt{5}\cdot\sqrt{3}} & 0 & 0 & \frac{1}{3} \end{pmatrix} \tag{3.14}$$

It was previously stated, that this thesis is supposed to experiment on an unification of the *metaformer* [6] and *graph networks*. The question therefore is, how the concept of the adjacency matrix is supposed to be applied to the metaformer framework.

**Graph convolutions** can be implemented with the knowledge of how to generate the adjacency matrix. Imagine an intermediate state of a *conformer* to be in the form $b \times h \times w \times e$, where $b$ is the batch dimension, $h$ is the height of the image in patches, $w$ the corresponding width and $e$ the embed dimension (corresponding to the channel dimension $c$ of a depthwise convolution). For a *depthwise convolution* in a conformer, $e$ 2D-convolutions (one for each channel) would be applied along the axis $h$ and $w$. It is now a simple step to transition to the graph version of that problem. For the *graph conformer*, the intermediate state has the shape $b \times (h \cdot w) \times e = b \times n \times e$, where $h \cdot w = n$ is the number of nodes. It is also the side length of the adjacency matrix $A$ for the corresponding graph. Comparing Equation 3.4 and Equation 3.12 it should become clear that for an image, the depthwise symmetric convolution can be represented by the information in an adjacency matrix that is assembled from $A_{\text{self}}$, $A_{\text{nn}}$ and $A_{\text{nnn}}$ if the correct weights $w_i$ are chosen. This is visualized in Figure 3.9.
The intermediate state for one element in the batch has the shape $n \times e$. That means the shape is $n \times 1$ if only one channel is calculated. Conveniently, the matrix multiplication of $(n \times n) \cdot (n \times 1)$

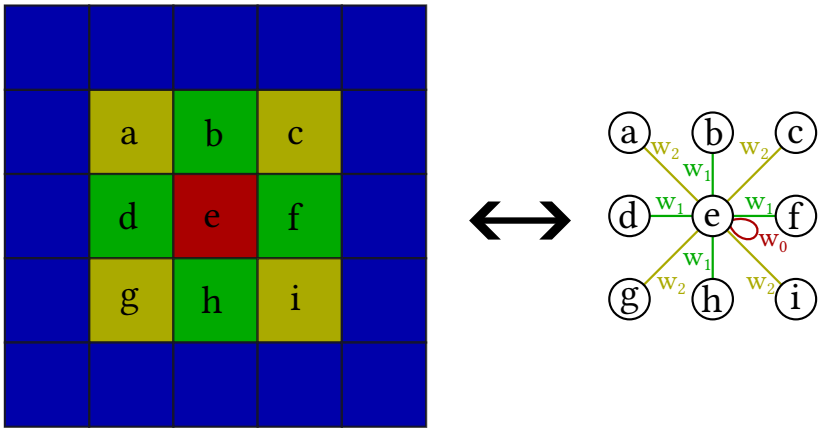


**Figure 3.9**: Schematic representation that shows the parallels between a depthwise convolution (only one channel shown) of a $3 \times 3$ Kernel over a patched image and the corresponding graph. Not involved nodes as well as edges are omitted from the graph on the right side. Both representations show only a subset of the whole situation, as only the calculation for the convolution of node $e$ is shown. The calculations for the other nodes are performed analogously, but shifted to another center patch. When comparing to Equation 3.12, $w_0 = 0.9$, $w_1 = 0.2$ and $w_2 = 0$, but with the proper adjacency matrices. The color scheme for neighbor visualization is the same as in the Lattice Visualization.

is not only possible, but it also gives the correct output dimension of $(n \times 1)$ and the exact same calculation is performed as in a regular symmetric convolution.
When trying to perform the depthwise convolution for an input dimension of $e$, $3 \cdot e$ weights are used ($w_0$,$w_1$ and $w_2$ for each channel, while there are $e$ channels). This can be evaluated by assembling the adjacency matrix that represents the symmetric convolution from each of the $e$ channels from the matrices $A_{\text{self}}$, $A_{\text{nn}}$ and $A_{\text{nnn}}$ in conjunction with the corresponding three weights. It is however more efficient to perform the matrix multiplication for each of the three different adjacency matrices only once and spread the result $e$ times. Then scaling the results with the weights and finally adding. The results are the same, because of the distributivity property of matrix multiplication. This is shown in Equation 3.15, assuming the intermediate state at step $t$ is named $v^{(t)}$.

$$\begin{aligned} v_{b=i,e=j}^{(t+1)} &= \left[ \left( w_{0,j} \cdot A_{\text{self}} + w_{1,j} \cdot A_{\text{nn}} + w_{2,j} \cdot A_{\text{nnn}} \right) \cdot v_{b=i}^{(t)} \right]_{e=j} \\ &= \left[ w_{0,j} \cdot \left( A_{\text{self}} \cdot v_{b=i}^{(t)} \right) + w_{1,j} \cdot \left( A_{\text{nn}} \cdot v_{b=i}^{(t)} \right) + w_{2,j} \cdot \left( A_{\text{nnn}} \cdot v_{b=i}^{(t)} \right) \right]_{e=j} \end{aligned} \tag{3.15}$$

The jax style implementation for this calculation can be seen in the appendix at 7.3.

**Graph pooling** is basically implemented analogously to the graph convolution. However it utilizes the averaging matrix $\tilde{A}$ (Equation 3.14) instead, in order to simulate an *average pooling* operation. As pooling does not require different or trainable weights along the axis $e$, the matrix can be pre-computed and cached. The jax style implementation is printed in the appendix at 7.4.

**Graph attention** is the metaformer token mixer requiring the least modifications. In [39] the transformer attention is limited to non-uniform, local blocks, by only calculating parts of the outer multiplication in the scaled dot product attention algorithm (Equation 3.6).
Here a more complex relationship of the patches is pictured. The dimensionality of the outer product matrix $M$ (from Equation 3.6) and the adjacency matrix for the encoded graph match. This is no coincidence, as the entry $M_{ij}$ corresponds to the attention from patch $i$ onto patch $j$, while $A_{ij}$ is a measure for how "connected" the patches are according to the graph structure. *Graph attention* is therefore calculated as the element-wise multiplication of $M$ and $A$. Because of the element-wise multiplication, it is not sensible to use $\tilde{A}$, which is motivated by matrix multiplication. Before passing the masked values through the softmax function, all zero values need to get set to $-\infty$. By doing that, it is ensured their contribution to the softmax evaluates to 0 and the corresponding connection is therefore masked out completely. The element wise multiplication and replacement with $-\infty$ happens on line 162 of the attention algorithm implementation in jax style (appendix, 7.2).

**Graph patching** can be achieved in multiple different ways. In order to get the input configuration into a state that is processable by transformers, the input shape needs to have the dimension $n \times e$, with the number of lattice sites $n$ and the embed dimension $e$. To achieve this, a state encoding of shape $n \times x$ is generated from the randomly sampled spin state $s_{\text{input}} \in \{0,1\}^n$. $x$ depends on the chosen *embed_mode* strategy (*duplicate_spread*, *duplicate_nn* or *duplicate_nnn*), as well as the lattice shape. An example for the different strategies is displayed in Figure 3.10.

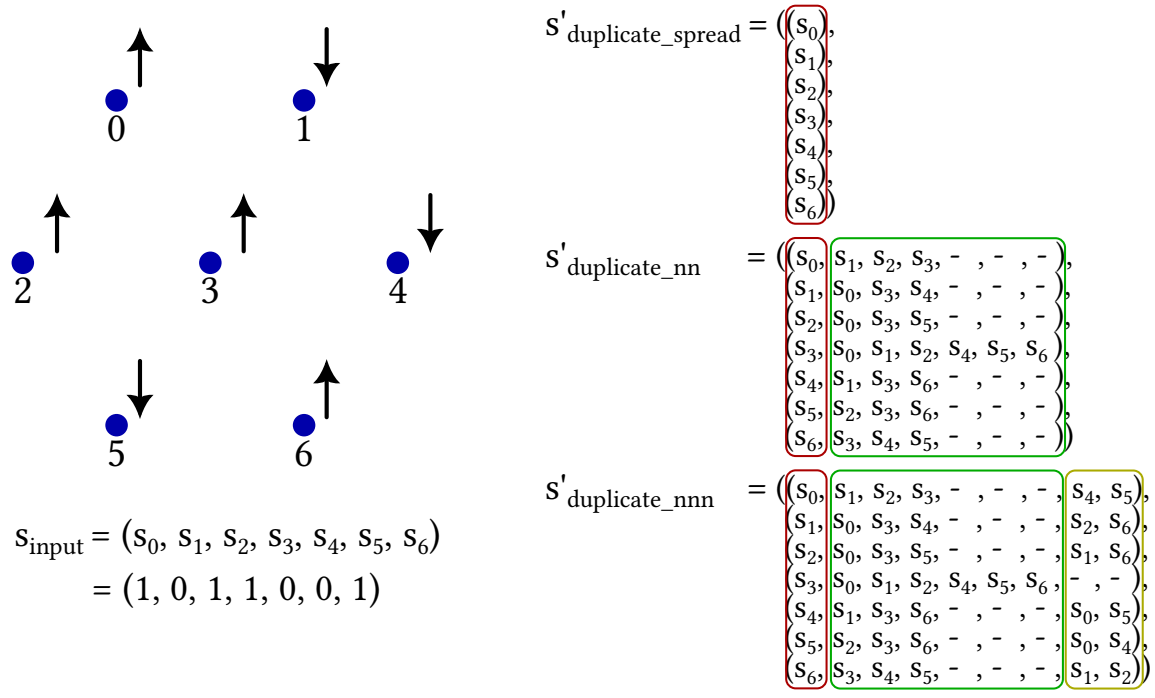


**Figure 3.10**: Assembling of the transformed inputs $s'$ of shape $n \times x$. $x$ depends on the embed_mode strategy. The intermediate step's shape further depends on the shape of the chosen lattice. In the example, a trigonal_hexagonal lattice of size 1 is chosen. The color scheme for neighbor visualization is the same as in the Lattice Visualization.

To convert the intermediate state $s'$ into the final form, the values are newly assigned ($1 \rightarrow 1, \quad 0 \rightarrow -1, \quad (-) \rightarrow 0$). To be easier for the network to differentiate, (0,1) is mapped to (-1,1) and previously unassigned vales are set to 0.
To reshape the element of size $n \times x$ to the input $n \times e$, a fcl is used. This strategy is analogous to the patching algorithm of most vision transformers.
The mapping pictured in Figure 3.10 can be performed quite efficiently by using a helper matrix and applying one *einsum* operation. The details can be referenced in the code [18].

## 3.2 Usage of Inductive Biases

An *inductive bias* is a necessary assumption or rule that allows an algorithm to make an inductive leap and generalize knowledge to be able to evaluate previously unseen data [43]. The bias can result from the used algorithm or from the underlying architecture. Having a bias is not optional, because without a bias an algorithm can never "learn", only memorize previously seen examples. Generally biases can be seen as constraints, that make a procedure *prioritize one solution over an other* [11].
It is obvious, that some biases are better suited for learning the solution to a task than others. The bias "every picture is a cat", allows an algorithm to judge any picture. However it will probably decide wrong most of the time.
Different machine learning architectures are motivated by different inductive biases. Because of that, they may perform differently on the same task. While (as previously established) it is in theory possible to learn every function with a large enough neural network, in practice some network designs outperform others by quite a margin.

### 3.2.1 Conventional Architectures

Every building block inside a sophisticated neural network contributes to the architecture's inductive bias as a whole, starting from the most basic ones.

**Fully connected Layers** have a relatively weak bias, because all inputs relate to all outputs independently [11]. The motivation is to construct a block with relatively high flexibility. It is barely restricted on the sort of data it can process and can therefore easily be used alongside other computational elements. In conjunction with a non-linear element (like ReLU), the architecture resembles the structure of the human brain. This motivates the assumption, that the element can perform useful computations. In the field of NQS, the *restricted Boltzmann machine* is often employed. The RBM basically consists of fully connected layers and can be motivated to be very well suited to model the statistical interactions of many body quantum mechanics [19].

**Convolutions** work with smaller kernels that get shifted over a larger input space. This implies the two biases of *locality* and *translation* [11]. That means, things close together relate

more than things far apart. Something that in most cases is very true for images (if a dog's ears are detected, but far apart from each other, this most likely isn't a dog. If they are detected next to each other, this is a good indication for a dog). Also it doesn't matter, where things are located on the image (a dog in the left corner is a dog in the same way as one on the right).

**Symmetric convolutions** have the same biases as their generic counterparts. They however have the additional bias of *only regarding the distance, not the direction*, which implies a kind of *rotational* symmetry. This helpful, if that data shows dependencies on the proximity of elements in relation to each other. The Ising lattices, reviewed in this thesis, have this property. Whether two spins interact or not is only dictated by how close they are, not by the direction one is from the other.

**Depthwise (separable) convolutions** were introduced by research on how to efficiently reduce network size [7]. Their bias is that interaction between values of one channel (for example one color channel in the case of rgb-images) can be computed independent and then combined in a second step. This makes sense, as there is already quite a lot of information inside one color channel. And as extracting the information per channel before comparing across channels saves a lot of computational resources, this is a logical step to take. Also it should in most cases be possible to recognize e.g. a dogs shape only from one color channel. Therefore checking for a shape in all channels separately and then comparing seems easier than forcefully training to assemble shapes across channels.

**Recurrent layers** are used a lot in translation tasks and video analysis, because of their *time invariance* [11]. These would likely be very powerful for investigating the development of a quantum system over time. As this thesis focuses on stationary problems, recurrency is not used.

**Residual connections** are used for a value in a calculation to be able to bypass a block completely. The metaformer architecture in Figure 3.6 includes residual connections as an important part. They are motivated by the bias, that it is simpler to zero out an element in a machine learning architecture, than to learn the *identity operation*. So if the network recognizes a block does more harm than good in terms of getting the correct solution, it can easily be bypassed by zeroing the block's inputs and using the residual connection. At the same time it is unlikely for the element to hurt, as basically each building block can be trained to learn the identity operation anyway. But if a block is not needed it only generates training errors, no additional value [44]. As this element is very likely to be useful in deep networks like metaformers, all the metaformer based architectures in the performed experiments use residual connections.

**Pooling operators** have many similarities to convolutions. They however have the added bias, that the relation does not even depend on the position of a value inside the kernels influence. It solely depends on the relative values. If this true for the data and the answer really only depends

on for example the *average* of the values, this module greatly increases computational speed and decreases the required memory. This on the other hand allows for evaluating larger/deeper networks which may be desirable.

### 3.2.2 Metaformer Architectures

(Vision-) transformers have shown to have a vast potential of computing capability. This can for one be attributed to the *attention* module. However as the research in [6] has shown, the overall structure is also important.

**The attention module** was originally motivated by natural language processing tasks [5]. The inductive bias was reduced as much as possible, to allow every computation to attend to information everywhere in a sentence with the necessary weights computed on the fly. Because the module has a very small inductive bias built in, the transformer is an expensive architecture to train. As it requires the calculation of all interactions, a lot of computational expense is needed to acquire the results of one pass. Also because of the low inherent bias, a lot of data is needed to shape the calculation into something useful. The advantage on the other hand is, that if one succeeds in training a transformer, the interactions deemed important by the training process are probably ever so slightly more effective as the ones forced by strict inductive biases like convolutional kernels. That visualized attention heatmaps behave very predicable and similar to convolution kernels is visualized in [31] and [33].

**The general metaformer structure** was deemed more important than the attention module for the success of transformer as general computing backbones by [6]. Separating the computation of local interactions inside tokens/patches from the *mixing* of information between tokens/patches, shows strong similarities to the separation applied to convolutions to turn them into depthwise convolutions [7]. Here the applied bias is, that it seems to be very effective to have only a small amount of tokens/patches interact with each other (this allows for powerful, but expensive operations - like attention - to be employed). While at the same time expanding the network to allow for dense calculations inside larger fully connected layers. Without sacrificing on speed, because interactions are only possible inside one patch. Lastly, the network is again shrunken behind the fcl to allow for the next inter-token computation and overall deeper networks. Residual connections are also employed to aid with the training of deep transformers.

**Poolformer and conformer** [6] combine the bias of *locality*, *translation* and *positional independence* with the biases of the *metaformer structure*. The idea is to combine both of the biases to create networks that may be slightly worse in performance than a pure transformer, but have a way smaller footprint because of the harsh bias restrictions. It could even be possible, that these stronger restrictions force the model to find a more fundamental and therefore better representation in some applications.

### 3.2.3 Graph Architectures

Extending the base metaformer architecture to a graph architecture has three advantages. Though a hidden problem will be discussed at the end of this section.

**Allowing for the application of established operators** like convolutions and pooling to be used on data that does not support encoding in tensor like structures. This makes it possible to apply a vast pool of experience and knowledge about convolutional/pooling architectures to graph data, without having to put much thought into inventing new architectures.

**Improving performance** of the architecture by introducing new biases. The *shifted window vision transformer* (swin transformer) [39] was able to improve the quality of vision transformers by introducing *non-uniform* blocks that get assembled in a *hierarchical* manner (combining smaller entities to form larger ones and repeating this to form increasingly larger structures). The swin transformer was not implemented as a graph transformer, because of the already tensor-like structure of images. However the good performance of the irregularly merged patches might suggest, that using the powers of the graph networks to *force* irregular interactions may be more efficient, than merely hard-encoding a specific lattice structure into the architecture. Primarily the patch encoder for the lattice input would be a good candidate for a target of further research in that domain.

**Becoming encoding independent** , as the location in memory and the location in the data structure do not have to be correlated. When representing an image with a tensor, the locality information is stored *implicitly* by the location of the values in relation to each other in memory. For graph structures, this information is stored *explicitly* in the graph's edges. Because of that, the data encoding can be changed and the performance of the graph network is not affected, while changing the positions where data is encoded to breaks a convolutional filter. The subsection 4.2.2 examines this.

A challenge that may present itself, is to what degree a dataset *requires* being encoded as a graph.
It is already proven, that for example image classification networks exist, that do not use the inductive biases of locality or translation [33]. It might be harder for a neural network to encode a trigonal lattice as a linear array than as a purposefully built graph network with the lattice structure built in. But as it will become evident in the experiments, it is definitely possible.
By taking a look at larger lattices, it is obvious they all have a huge degree of periodicity - which is good as that is what defines a lattice. But at the same time this raises a question: Surely, it is convenient to represent everything as a graph, because they can model most arbitrary relationships (in fact not all relationships [11]). Though they also come with a significant performance overhead when implemented in the current hardware and the methods described in subsection 3.1.5.

This poses the question, whether the lattice data experimented on is even *irregular enough* to gain a performance boost from the employment of graph networks. In follow up research it would therefore definitely be desirable to measure the impact of graph architectures across multiple magnitudes of "irregularity" in their structure.

# 4 Experiments and Verification

## 4.1 Metaformer in Image Classification

The first set of experiments will be on the image classification task. The goal is to compare the architectural advantages and drawbacks between multiple different kinds of metaformers.
Training was be performed on a subset of 100 classes from the *ImageNet* dataset [27] ($\approx$ 126 000 images). The accompanying code to replicate these experiments can be found on GitHub [28].
The neural networks, as well as the training and evaluation code were written in python. The *PyTorch* [45] machine learning framework was used as a measure to efficiently define neural networks and lever the computational capabilities of parallelization via GPUs.
The main metaformer model was originally based on the vision transformer found in an implementation of *DINO* [41] and modified strongly. The code base provides a DINO implementation with as little modifications as possible to compare to own models. Also the code for comparing against a pre-implemented *poolformer* [46] is provided.
The *einops* package [47] is used in multiple locations. It provides tensor operations, configurable with the *Einstein notation* and simplifies the notation and subsequently readability. Lastly, the implementation of the *positional encoding* is provided by [42].

### 4.1.1 Training Settings

The comparison of different architectures in a fair manner is not trivial to do. Because it is possible that the performance of an architecture is highly dependent on the used hyperparameters, for a fair comparison it would be necessary to optimize the hyperparameters for every architecture and only compare the top scoring results. As performing training runs for this thesis takes durations ranging from 30 min to 12 h, it is not feasible to run the powerset of hyperparameter combinations and choose only the best. Even with the resources provided by Augsburg University to calculate a higher number of runs than were ever possible on only my local machines, investing months into these calculations is neither possible, nor sensible.
Therefore an informed pre-selection of likely interesting runs had to be chosen beforehand. Helpful in this manner were the previous experiences by other researchers. The metaformer's hyperparameters *image size*, *number of heads*, *patch size*, *embed dimension*, *depth* and *mlp ratio* have been chosen as the same values, the *DINO tiny* model was implemented in [41]. As the transformer in the DINO paper [31] was a structural template for this metaformer, selecting the smallest model from their lineup made it possible to perform as many comparisons as possible with limited resources.
In order to understand what is measured, a full overview of the recorded parameters is presented in Figure 4.1.

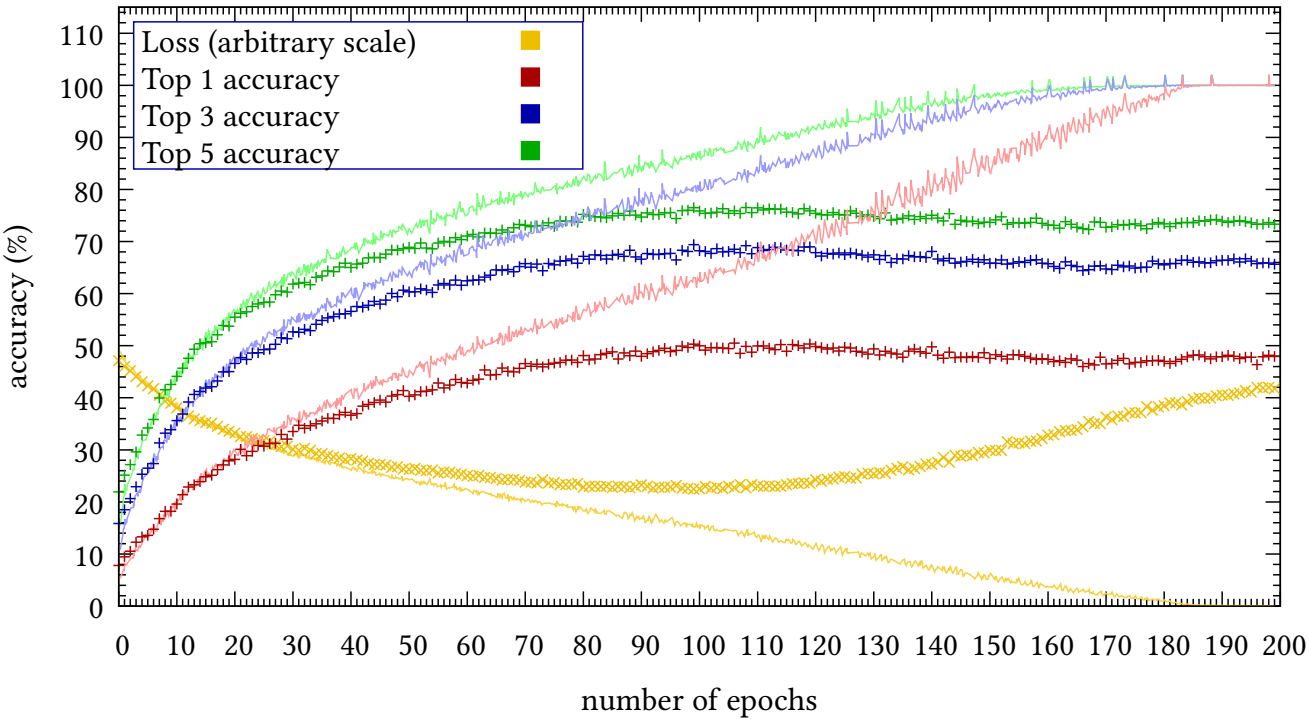


**Figure 4.1**: Graphic depiction of all recorded metrics for the image classification task. The data is from a *CF* metaformer and has been selected because it clearly shows the expected behavior of convergence. The *accuracy* (subsection 2.2.1) and *loss* (subsection 2.2.2) are depicted. The results from *testing* the model on the validation set are depicted as individual data points and colored dark. The data collected during the *training* calculations are colored in the corresponding color, but shaded more brightly. The training data is plotted as a smoothed line, to allow for a more pleasant viewing experience. The loss data is scaled arbitrarily to fit into the graph, as the absolute values are not relevant.

All the runs share a few key similarities. Because of that, the full data for one run is only shown once. Later figures will leave out most of the metrics, because they are almost completely redundant.

The model training always follows the same default structure: The accuracy starts low and the loss starts at a high value. This is expected as in the beginning the model only guesses randomly. Therefore a top-1-accuracy should start at around 1 % for a classification task of 100 classes. It can be lower or higher depending on random chance, but in general it holds for very early values. As the results improve very rapidly in the first few epochs, this can not be seen in the charts for most runs. Also it should be noted, that the first recorded *test* data point is calculated *after* the first epoch. The model therefore is already advanced and no longer random. This explains, why the curves start higher than expected.

A shared trend for all datasets is, that the top-3-accuracy lies above the top-1-accuracy and the top-5-accuracy above both. This is necessary, because if a model classifies the correct answer among the best three highest rated answers, it automatically has also classified it among the top five. Because of that only the performance for the top class predictions will be shown, as the other metrics provide not much additional insight into the comparison between architectures.

It can also be observed, that the *test* accuracy rises at first, then plateaus and finally falls off. The *training* accuracy however constantly rises and can even hit 100 %. This can be explained by overfitting (subsection 2.2.2). As explained, the amount of training data is limited. Therefore the model is presented with the same image many times. As it is allowed to adjust its weights based on the training data, it is possible for it to "remember" all solutions. As can be clearly seen by the descending test performance after a specific point, doing so is not helpful for classifying unseen images. Overfitting can be counteracted by setting so called *dropout* connections/weights. By randomly setting weights to zero, the model is always restricted to a subset of its weights and therefore has to develop a more robust internal representation. This reduces/delays the overfitting behavior.
The goal however is to detect differences in the architectural design. Thus it is important to be able to differentiate whether a poolformer is inherently more robust to overfitting than a conformer or the other way around. Because of that the dropout always is set to zero.
Subsequently, all models were trained to the point of overfitting. If statements about the accuracy of a model are made, the highest value of the *test accuracy* is presented. For Figure 4.1 this is at epoch 106. It can be noted, that this is also the point, at which the train loss starts to rise again. As explained previously, the loss is a measure of model's performance. The smaller the loss, the better the representation. If the test loss is rising and the train loss is still falling, this is a clear indication for the model being in the process of overfitting. This can be summarized to the rule, that all models were trained to the point, where either the test accuracy was noticeably dropping or the test loss was noticeably rising. If accuracies are plotted, that have no clearly visible drop-off at the end, this is because the overfitting could be recognized in trend of the loss. Calculations then often were aborted to save resources.
Figure 4.2 shows a subset of the experiments performed to choose an appropriate *optimizer* and learning rate. It can be easily seen, that the choice of optimization algorithm influences the speed of convergence, as well as the maximum performance of the networks. The *adamw* [29] optimizer performed best in both of these metrics. The established *stochastic gradient descent* update algorithm scored a between 10 to 15 % worse test accuracy, as well as a two to four times slower rate of convergence in relation to the number of epochs.
Increasing the momentum value comes with no additional costs or drawbacks, however it leads to overall better performing models, that converge faster [48]. Because of that 0.9 was chosen as the momentum value for the sgd in all subsequent runs.
In literature, multiple comparisons between the different optimization mechanisms have been performed [49]. Though the adamw optimizer performed better, the update procedure is also more expensive, in particular requiring more memory. Because of that, the batch size could not be set to 128 (like for the sgd optimizer), but had to be reduced to 64, as the available GPU memory was quite limited at the start of the experiments. At the same time, the sgd algorithm produced more consistent trends with less abrupt changes. In order to again choose reliable comparability above maximum performance, only the sgd optimizer is used in subsequent image recognition experiments.
All of the optimizers use a constant *learning rate* of 0.001.

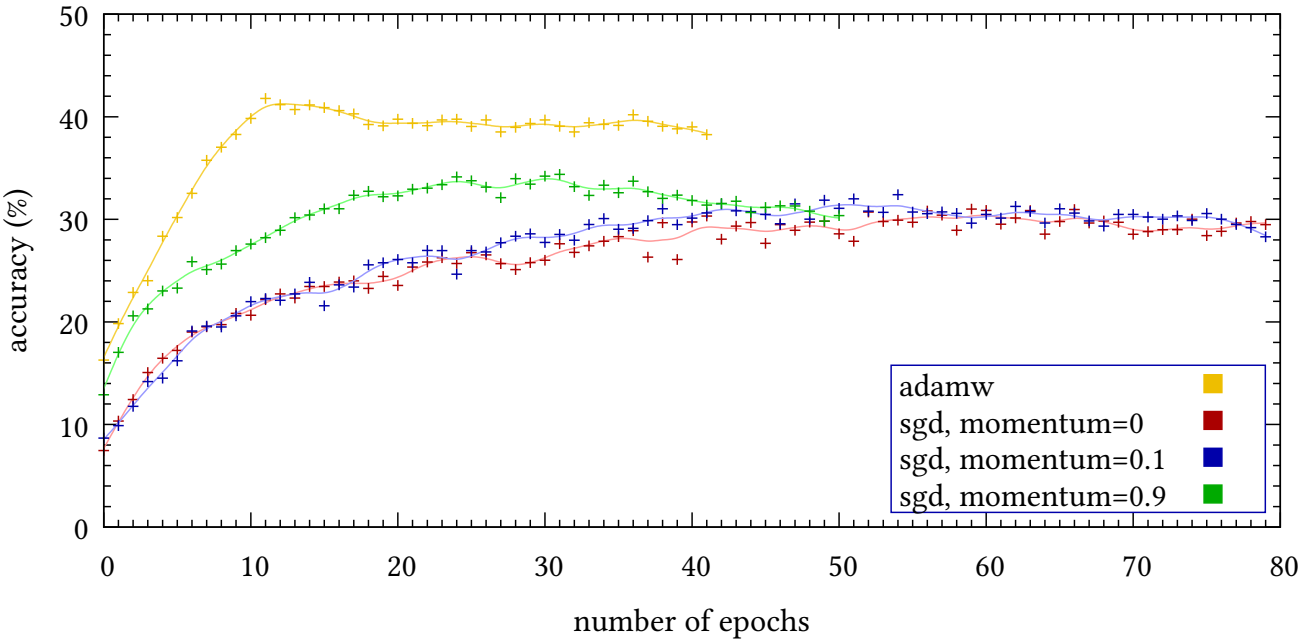


**Figure 4.2**: Comparison of the top-1-accuracy for training a modified DINO tiny transformer. Different optimizer algorithms are used for the backpropagation. The training accuracy is pictured, the lighter colored lines are interpolations of the data with square splines. This is supposed to make the differentiation of the red and blue curve easier.

### 4.1.2 Importance of Positional Encoding

As described in subsection 3.1.3, the transformer architecture's performance can be improved by providing *positional encoding* (pe) information. While with convolutions and pooling identical patches at different locations can be distinguished, in the attention mechanism they can not. Adding pe resolves this issue.
The question arises, whether the *masked* attention, that uses the graph information to confine the transformer to only its local neighborhood, also requires pe or not. Additionally if efficient training requires pe, it needs to be discussed what type works best.
The simulations are visualized in Figure 4.3. It can be seen, that the different architectures react differently to the types of pe.
The data clearly shows benefits in employing pe. All models show an increase in accuracy of about 3 to 4 %. This is less than the difference between some of the other token mixing elements, but still a significant margin in comparison to the absolute performance. As adding pe is rather cheap, the employment of this strategy can definitely be useful: Both types of pe are only added at the start of the calculation. Therefore the cost is not dependent on the depth or type of token-mixer. Calculating sinusoidal pe can theoretically be done once and the result can be cached, rendering it basically instantaneous to employ and of no negative computational impact. Furthermore it is resolution independent, as the sinus curves can later be sampled with a smaller interval in order to increase network resolution retrospectively. Learned sinusoidal encodings on the other hand require a backwards propagation of error values into the encoding weights.

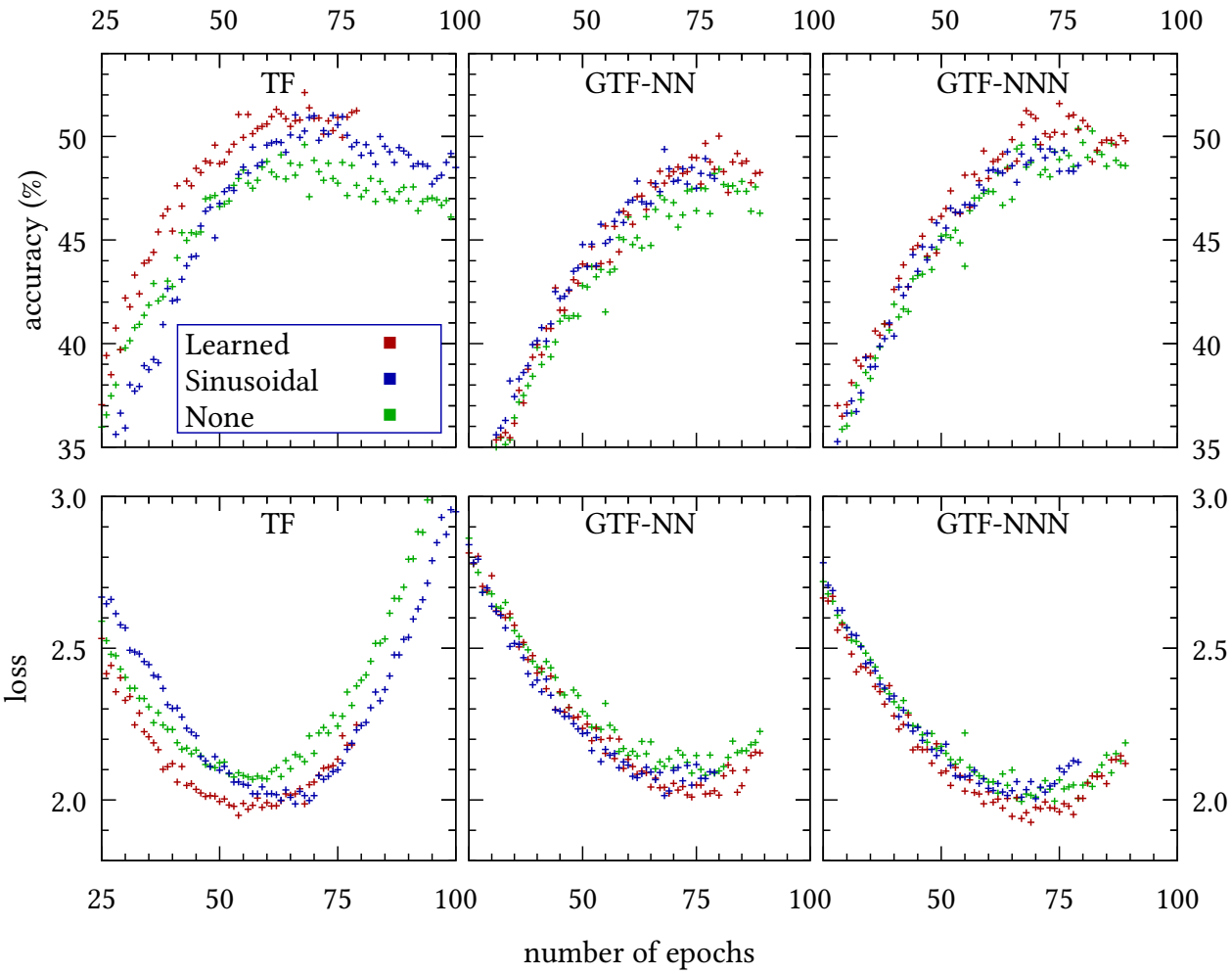


**Figure 4.3**: Visualization of the training attention and loss metrics for three different models. *TF*, *GTF-NN* and *GTF-NNN*. Only part of the full training is shown. Each model was trained once with *sinusoidal* pe, *learned* pe and *without* any additional pe. Apart from that all hyperparameters are kept static.

This requires additional computation and storage space, as well as providing no obvious way to update the encoding resolution after the fact.
The pure transformer definitely performs best with a learned pe. Not only is the overall accuracy higher ($\approx$ 1.5 %), but also is the loss-minimum reached about 12 epochs earlier - a huge potential reduction in necessary computation.
The GTF-NN model shows less benefit from applying pe. This makes sense, as by design the number of attention targets is intentionally very limited. Sinusoidal pe and learned pe perform very similar, with sinusoidal pe converging ever so slightly faster than learned. It should be noted though, that the sample size is too limited to draw definite conclusions from this tiny difference.
For the GTF-NNN the learned pe outperforms the sinusoidal one, taking about 5 epochs longer to converge, but accomplishing an about 2 % higher accuracy.

To summarize, pe can definitely improve the performance of an attention based neural network. The correlation can hold true not only for normal attention as already know [5, 33], but also for the attention mechanism with graph-limited influence, as the calculations demonstrate.
A challenge in more complex graph structures is undoubtably the definition of suitable pes, as the canonical sinusoidal encoding is not extendable to irregular graphs. The learned pe would probably constitute the easiest and best solution, but comes with a computational overhead both in terms of the cost of one epoch, as well as the speed of convergence, i.e. the needed number of epochs until the calculation converges.
In the ground state search task, no positional encoding is applied. Its application could be a task for future research, especially on highly irregular graphs.

### 4.1.3 Comparison of Different Token Mixers

In subsection 3.1.5 it was established, that for tensor-like data, *convolutions* and *graph-convolutions* are exactly the same, only calculated differently.
This thesis was validated experimentally and the results are visualized in Figure 4.4.

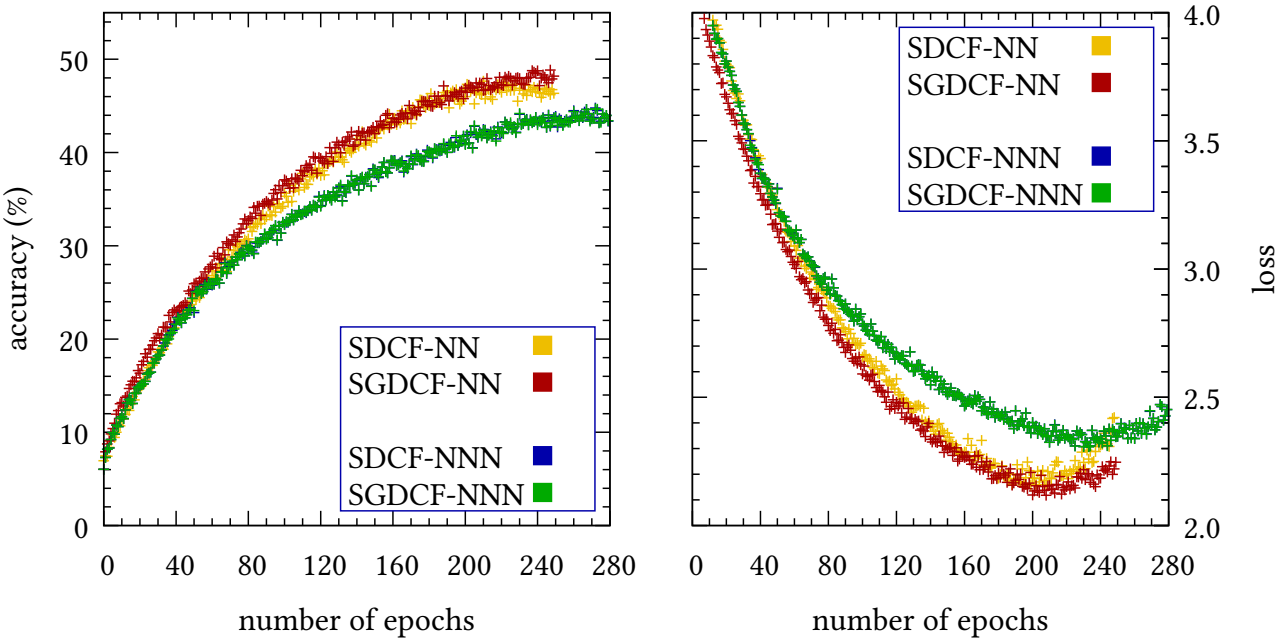


**Figure 4.4**: Comparison of the behavior of the traditional convolution based conformers (yellow and blue) to the graph based ones for image training data. The Blue and the green curve are completely identical (but calculated by optimizing different models), therefore blue is not visible. The reason, why the red and yellow curve are not also identical is due to different uses of the random number generator.

The data shows, that the implementations of SDCF-NNN and SGDCF-NNN behave exactly identical, even though they compute the path interactions with different operations, as they are mathematically identical.
One can expect, that SDCF-NN and SGDCF-NN behave likewise, but slight differences in the data of these two can be observed. This is because of the different ways, the pseudo random number generator of PyTorch is used:

The implementation of the graph-convolution (`GraphMaskConvolution` in [28]: `/models/metaformer.py`) and the conventional-convolution (`SymmDepthSepConv2d` in [28]: `/models/helpers/SymmConv2d.py`) query the random number generator a different number of times, because the graph implementation always allocates $(3 \cdot \text{channels})$ weights and the conventional implementation only allocates $(2 \cdot \text{channels})$ weights for nn interactions and $(3 \cdot \text{channels})$ for nnn. This causes the random number generator to drift apart and produces a propagating difference in the calculations *only* for nn and not for nnn.
While this is a fixable flaw of the implementation, it clearly shows that the graph and non-graph implementations are behaving similar, even if the starting conditions are non-equivalent. The comparison among the whole set of metaformers will emphasize how similar these two curves really are.
This proves, that convolutions can directly be translated into a graph context and motivates their function on non-tensor-like data structures.

The architectures are compared against each other in the last section discussing the first experiment. Table 4.1 shows the one to one comparison of all implemented metaformer variations on the same task. The hyperparameters (optimizer, batch size, embed dimension, depth, etc.) were *exactly the same for all runs*. The complete course of the validation metrics for selected runs is printed in Figure 4.5.
As it is impossible to discuss all the collected data in depth, only the most interesting observations will be emphasized. Several trends can be noticed across the whole metaformer lineup.

**The attention based transformers** for the start score the highest overall accuracy. The graph-masked attention falls only slightly behind the full transformer in the case of the nn-limited attention and even pulls ahead in some of the positional encodings in the case of the nnn-limited interaction. All the models are basically identical in terms of required parameters, only the learned pe requires a noticeable different amount of weights as already discussed in subsection 4.1.2. The weights inside the graph masks are basically negligible.
Contrary large discrepancies occur in the overhead in computational time. The graph variants require both more epochs to achieve maximum accuracy, as well as the epochs itself taking longer to be calculated. Both is sensible, as the masking requires extra calculations and the zeroing of interactions creates a *choke point* for the backpropagating information (this will become even more pronounced in the later architectures). Still it is clear, that masking attention does not hurt the performance as much as one could expect (the defining characteristic of the attention module being the *global* interaction range, that is here taken away).
As the nn-graph-attention takes both longer to train and achieves less accuracy than the nnn-graph-attention, using the latter only brings advantages for the image classification task. That the nnn variant even outperforms the full transformer in terms of raw accuracy can probably be attributed to the relatively small model size and un-optimized training procedure. As a perfectly trained full transformer should be able to replicate a graph transformer, this shows that while

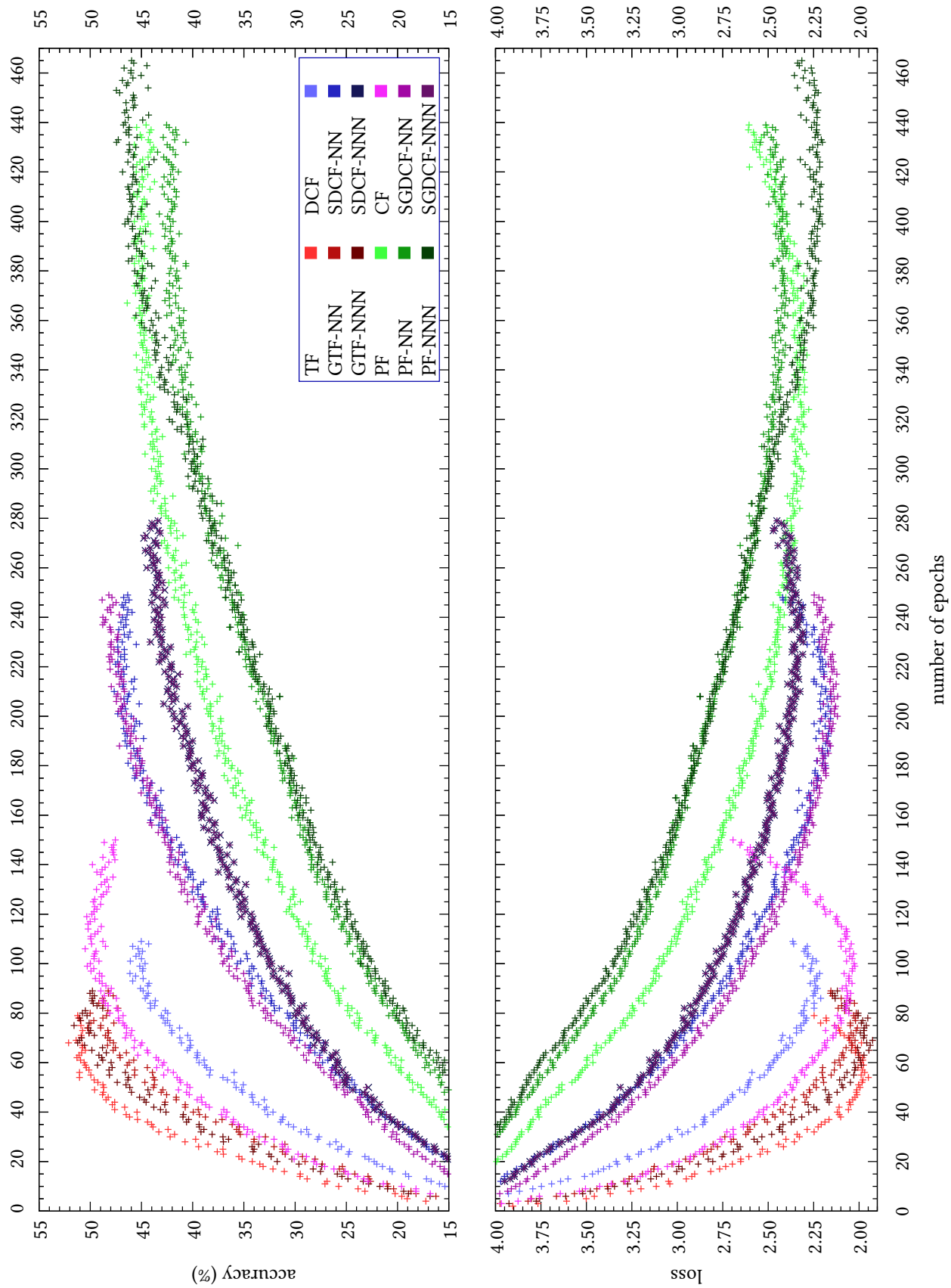


**Figure 4.5**: Graphic depiction of accuracy and loss of the metaformer architectures on the image classification task. For the transformer architectures, only the *learned* pe is drawn, as the other encodings are already printed in Figure 4.3. This data is from the same runs as the data in Table 4.1.

| name | $\text{acc}_{\text{max}}$ | $\text{ep}_{\text{max}}$ | #params | $t$ / ep | $t_{\text{max}}$ | $\frac{t}{\text{param}}$ | $\frac{t_{\text{max}}}{\text{acc}}$ | $\frac{\text{param}}{\text{acc}}$ |
|---|---|---|---|---|---|---|---|---|
| TF (learned) | 51.30 % | 62 | 5 543 332 | 391.01 s | 6.73 h | 1.00 | 1.00 | 1.00 |
| TF (sinus) | 51.03 % | 66 | 5 505 700 | 391.32 s | 7.17 h | 1.01 | 1.07 | 1.00 |
| TF (none) | 49.11 % | 63 | 5 505 700 | 391.22 s | 6.85 h | 1.01 | 1.06 | 1.04 |
| GTF-NN (learned) | 50.01 % | 80 | 5 543 368 | 435.25 s | 9.67 h | 1.11 | 1.47 | 1.03 |
| GTF-NN (sinus) | 49.38 % | 68 | 5 505 736 | 435.53 s | 8.23 h | 1.12 | 1.27 | 1.03 |
| GTF-NN (none) | 48.42 % | 80 | 5 505 736 | 435.73 s | 9.68 h | 1.12 | 1.52 | 1.05 |
| GTF-NNN (learned) | 51.58 % | 75 | 5 543 368 | 435.27 s | 9.07 h | 1.11 | 1.34 | 0.99 |
| GTF-NNN (sinus) | 49.87 % | 70 | 5 505 736 | 435.47 s | 8.47 h | 1.12 | 1.29 | 1.02 |
| GTF-NNN (none) | 50.38 % | 79 | 5 505 736 | 434.84 s | 9.54 h | 1.12 | 1.44 | 1.01 |
| PF | 46.39 % | 367 | 3 727 012 | 217.14 s | 22.14 h | 0.83 | 3.64 | 0.74 |
| GPF-NN | 43.26 % | 403 | 3 727 012 | 213.39 s | 23.89 h | 0.81 | 4.21 | 0.80 |
| GPF-NNN | 47.43 % | 432 | 3 727 012 | 213.40 s | 25.61 h | 0.81 | 4.11 | 0.73 |
| DCF | 46.13 % | 96 | 3 747 748 | 317.41 s | 8.46 h | 1.20 | 1.40 | 0.75 |
| SDCF-NN | 47.66 % | 210 | 3 731 620 | 319.98 s | 18.67 h | 1.22 | 2.98 | 0.72 |
| SDCF-NNN | 44.49 % | 260 | 3 733 924 | 319.72 s | 23.09 h | 1.21 | 3.95 | 0.78 |
| SGDCF-NN | 48.78 % | 237 | 3 731 620 | 250.64 s | 16.50 h | 0.95 | 2.58 | 0.71 |
| SGDCF-NNN | 44.39 % | 260 | 3 733 924 | 280.66 s | 20.27 h | 1.07 | 3.48 | 0.78 |
| CF | 50.50 % | 106 | 7 710 628 | 386.45 s | 11.38 h | 0.71 | 1.72 | 1.41 |

**Table 4.1**: Tabular representation of the performance data for different metaformers in the image classification task. The columns from left to right are: maximum top-1-accuracy, epoch this max accuracy was reached, the number of trainable parameters, the time per epoch, the time until the maximum accuracy was reached, the calculation time per parameter factor, the time until the maximum accuracy was reached per maximum accuracy factor and the number of parameters per maximum accuracy factor. The three factors are scaled so that *TF (learned)* corresponds to 1.00. For everything but $\text{acc}_{\text{max}}$ lower is better. The maximum accuracy and the corresponding epoch have been chosen manually to select values that take both the accuracy and the loss curves into account. Because of that, sometimes bigger accuracies are reached later than the ones listed in the table. But they were excluded as lucky outliers, if the loss was already signaling overfitting.

in theory the latter is inferior, in practice for small models or imperfect training procedures it can definitely be beneficial to force the inductive bias of *locality* upon the model.
A so far unmentioned detail could furthermore vastly improve the graph-transformers. This will be discussed in subsection 4.1.4.

**The convolution based conformers** continue all the described trends. They require less trainable weights (around 33 % less) than the transformers. Their maximum accuracy is slightly worse than the transformers', and they require longer to reach peak accuracy. Though as soon as they are fully trained, they can both be evaluated quicker and have an overall smaller footprint due to the smaller weight count. This shows, that for our small size of network, shaping a model around a hard coded inductive bias takes more effort, but can pay of in terms of efficiency.
Interestingly, when comparing the symmetric and non-symmetric variants, the non-symmetric one converges *significantly* faster in training (by a factor of 2.1 to 2.7). This can again be attributed to the strict choke point of only two weights for SDCF-NN in comparison to the nine inside the DCF kernel.
It is however very interesting to see, that the conformer architecture is the only kind of metaformer where the nn-convolution performed better than the nnn variant and the full vari-

ant, by a very significant margin of 4.2 %. This shows that it is clearly worthwhile to experiment with different interaction schemes, when employing graph networks.
As previously discussed, the traditional convolution and graph convolution variations are equivalent on tensor data. The models clearly show this, especially when comparing the course of the curves in Figure 4.5. Unexpected on the other hand is the significantly worse performance of the traditional variant in terms of computational speed (13 to 27 % longer time per epoch). This may seem counter intuitive, but can be explained when looking at the implementation. As all the models use the same underlying framework, the internal representation is always stored as $b \times (n = w \cdot h) \times e$, with the batch dimension $b$, the number of patches $n$, the width and height in patches $w$ and $h$ and the embed dimension $e$. When a traditional convolution should get applied, this needs to be reshaped to correspond to $b \times e \times w \times h$ and reversed after the convolution operation. This step creates an unnecessary overhead, that could be left out, if only traditional convolutions were needed.

**The pooling based poolformers** sit at the opposite end of the overall trends. They require the least amount of parameters and - because of the very loose coupling of the pooling operation - the by far longest time to train. A fully trained poolformer though provides both the best parameters per accuracy efficiency, as well as the lowest evaluation time. PF and GPF-NNN perform practically identical, which was expected on tensor data.
The overall accuracy is definitely lower than that of a transformer, but because of the significant efficiency advantage the poolformer's performance is still definitely competitive.

**The full conformer** is listed in the lineup, but it - as already discussed - can not be considered a *strict* metaformer. The performance is relatively high and the speed of convergence is average. Though the extremely large parameter count (more than 106 % *more* than a depthwise conformer) drives the point why depthwise separable convolutions are so desirable in the metaformer framework and beyond.

### 4.1.4 Efficiency of Graph Attention

As a closing argument to the first set of experiments, the efficiency of graph masked attention will be discussed.
While the experimental results in subsection 4.1.3 (especially Table 4.1) showed a performance lead for the attention based transformers, as stated this operation is still comparably expensive. The *GTF* architecture had by far the longest training time per epoch of the examined models. This is because of the inherent cost of the *attention* operation that scales $\propto \mathcal{O}(n^2)$ with the number of nodes (in case of images, the number of patches) $n$, onto which the costs for masking the not required attention connections are added (also $\propto \mathcal{O}(n^2)$).
This doesn't have to be the case though, it merely is a result of the chosen computational algorithm:

1. calculate *all* interactions from every node with every node

2. mask and scale only the interactions belonging to
    - the node itself
    - the nearest neighbors
    - the next nearest neighbors
3. add all scaled interactions
4. replace all zeros with $-\infty$ to make them vanish in the softmax

Looking at this again, it is clear a great number of computations is wasted. Every entry of the outer product that is no self, nn- or nnn-interaction *will* get overwritten with $-\infty$. Also all entries that contain $-\infty$ get passed through softmax, *even though* is is clear beforehand that they will not contribute anything.
Each node has (depending on the chosen interaction paradigm) only a constant number $c$ of nodes it is going to interact with. For large Graphs, this $c << n$. With that in mind, the calculation really should have the complexity $\mathcal{O}(c \cdot n) \rightarrow \mathcal{O}(n)$, because for each of the $n$ nodes, only $c$ interactions should have to be calculated and passed through the softmax.
As the code in the implementation makes use of GPUs to accelerate the computation, it is still faster to calculate the unused connections and "throw" them away shortly after. Because this harnesses the GPUs, highly optimized implementation of operations like matrix multiplication. But through implementing the linear time calculation efficiently and compiling it to the GPU instructionset, a massive speedup could be achieved. This would bring in the benefits of the attention operation, but get rid of the $\mathcal{O}(n^2)$ complexity problem. While not having the full attention at ones disposal, the graph masked attention still is absolutely competitive for some use cases (Table 4.1).
An even greater speedup could be achieved by implementing the described algorithm directly in hardware. While the current demand for graph attention calculation may not yet be high enough to justify this, future developments could easily make investing in this technology worthwhile.

## 4.2 Metaformer in Ground State Search

For the second set of experiments, a quantum mechanical *ground state search* will be implemented and performed. The goal is to employ multiple metaformer architectures as a NQS and perform VMC (subsection 2.1.4) to solve for the ground state. The speed of convergence and other metrics will be compared.
The code for replicating these experiments was also written in python and can likewise be found on GitHub [18]. The *jax* framework is used to supply the necessary *autograd* and *XLA* functionality [50]. It is extended with the *flax* [51] module for improved definitions of machine learning tasks and models. Finally, the *jVMC* [52] package supplies the base functionality for handling quantum mechanical wavefunctions and performing optimizations with VMC.

### 4.2.1 Comparison to Established Architectures

As with the last experiment, the biggest challenge will be discovering the appropriate hyperparameter combinations. Depending on the shape or size of the lattice and the ratio or sign of the Ising parameters, different architectures can either be more or less successful. Some wavefunctions might be more difficult to represent and therefore require more or less weights depending on these factors. As with the image classification task, it will never be possible to compare all combinations from the powerset of hyperparameters with each other, nor is it realistic to optimize each metric and compare only the top scoring results. It is important to note, that while some architectures may perform good on a specific lattice, on a different problem they might be easily surpassed. Therefore the correct NQS needs to be discovered for every problem, as no "silver bullet" exists.
For all of the subsequent measurements, the hyperparameter combinations and shared constants were chosen in advance, to minimize the opportunity for cherry picking. In the image classification experiment for example the metaformer hyperparameters were chosen to be static, which resulted in a different number of trainable parameters for each network. For the comparison of metaformer models on the ground state search task, the number of weights was chosen to be as similar as adjusting the hyperparameters allowed. The performance of different NQS models is visualized in Figure 4.6.
From the number of trainable parameters listed in the image caption it should instantly become clear, that the networks applied as NQS are tiny in comparison to the ones used in image classification (Table 4.1, also consider these are based on *DINO-tiny*, the smallest vision transformer in the DINO lineup [41]). This is generally not problematic, because the metaformer architecture is highly adjustable and can be tuned to such small sizes. One could only argue whether a metaformer with depth one still belongs to the metaformer class.
The two most important metrics in the ground state search are the energy $E$ (in the experiments always the energy per lattice site is stated, as to make the graphs independent from the lattice size) and the variance of the energy $\mathrm{Var}(E)$ (also scaled to the number of lattice sites $L$). The energy is important, because it shows an easy to understand macroscopic parameter, that even corresponds to the eigenvalue computable with the exact diagonalization (subsection 2.1.3). The variance on the other hand is not as tangible, but still a very important metric. It can only be zero if the wavefunction is an exact eigenstate of the energy operator. Furthermore eigenstates that are not the ground state are exponentially less likely the occur. So the distance of the calculated variance to zero is a direct measure of how good the ground state wavefunction is represented. It even is an indication, whether the wavefunction is really parametrized properly, or the calculated energy just happens to be the ground state energy by chance.
Looking at Figure 4.6, the behavior of different metaformers in a specific Ising problem can be observed. The metaformers are compared against a *CNN* and a *RBM* that are bundled in the *jVMC* package [52]. The RBM is in this case not able to encode the wavefunction, therefore the calculated energy is different from the rest. Also the high variance (notice the log scale) indicates the RBM is not suited for representing this wavefunction. To reiterate, this doesn't mean RBMs are bad, as in literature they have been employed successfully numerous times [19].

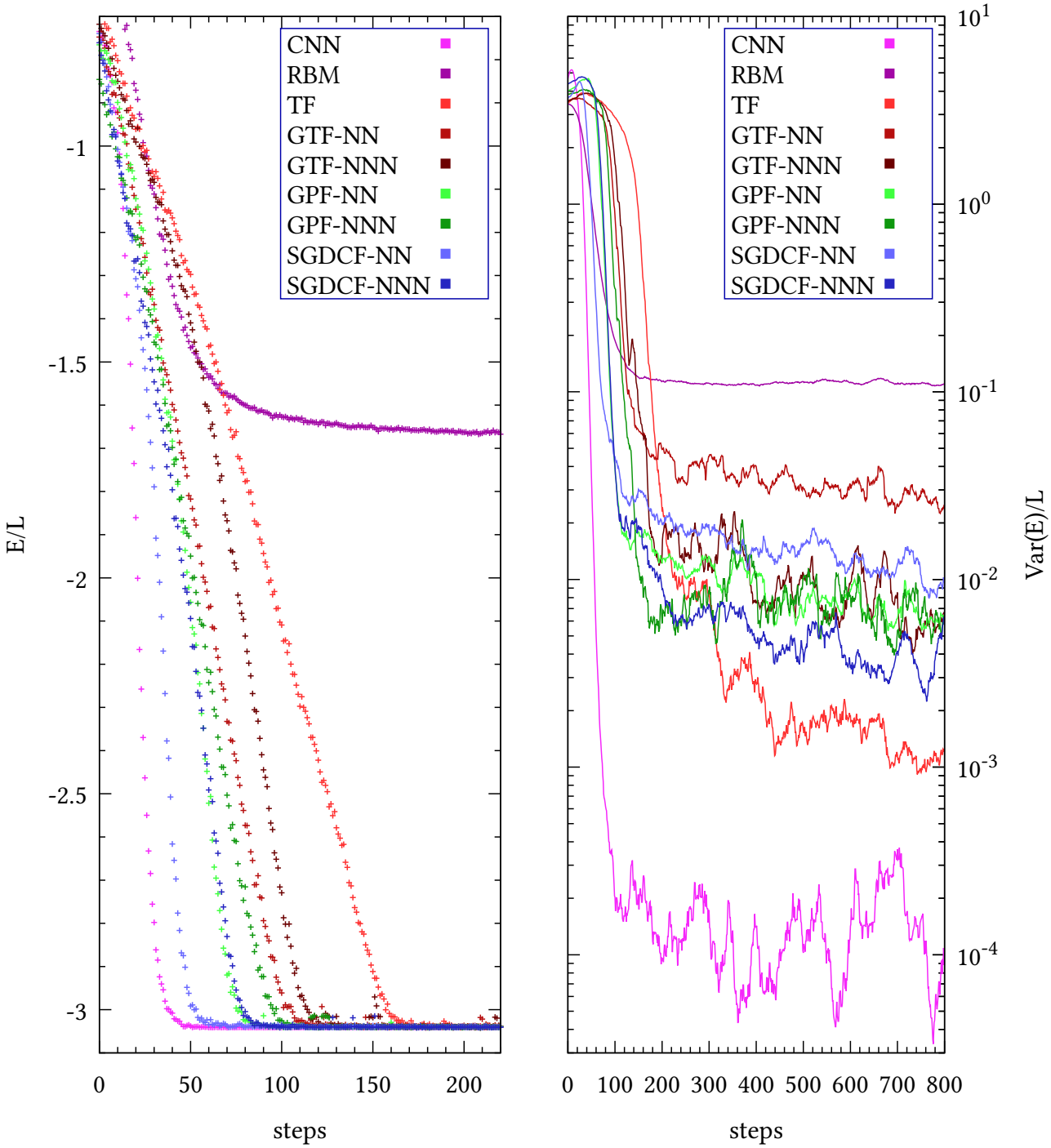


**Figure 4.6**: A comparison of different established and novel neural network architectures used as NQS in a ground state search. The calculations were performed for a 2D-trigonal_hexagonal lattice of size 3 (37 lattice sites). The lattice is periodic and the encoding is set to be random. The models were configured to be as similar as possible in terms of the number of trainable weights. The *CNN* was set to have 16 channels ($\hat{=}$ 592 weights), The *RBM* was set have 8 hidden layers ($\hat{=}$ 592 weights). All metaformers were set to a depth of 2, and a mlp-ratio of 2. The poolformers have an embed-dimension of 7 ($\hat{=}$ 560 weights for *GPF-NN* and 602 for *GPF-NNN*). The conformers also have their embed dimension set to 7 ($\hat{=}$ 602 weights for *SGDCF-NN* and 644 for *SGDCF-NNN*). Finally the transformers are set to have an embed-dimension of 5 ($\hat{=}$ 530 weights for *TF*, 566 weights for *GTF-NN* and 596 weights for *GTF-NN*). Important to notice are the different scales of the x-axes. The energy per site is only shown until step 200, because after that nothing of interest happens, while the variance of the energy is shown until step 800. The variance data is interpolated with a moving average in the logarithmic scale of width 35 steps.

The CNN converged the fastest. It also required the least amount of time for the performance of one step. Therefore it not only produced its results in less steps, but also the fastest. The metaformers generally were slower to compute, but not by a significant margin. Most of the calculations for one step were in the neighborhood of 2.5 to 4 times the step time of the CNN.
The variance graph shows, that the state representation of the CNN converges to a better representation than any of the metaformers. The variance data is highly smoothed to make it possible to display, still the CNN's variance fluctuates over about one order of magnitude. Looking at the data, it seems that the metaformers converge with less fluctuations at a more stable rate, still the CNN's variance is overall lower.
It can also be noted, that by step 800 all of the metaformers' variances are still on a clear downwards trend, so training them longer would probably result in a further increasing precision. Because of the rather big fluctuations of the CNN's variance it is harder to pinpoint the trend reliably.
Only by carefully selecting the desired metrics, a suited network can be chosen. While in the experiment, the TF network converged by far the slowest, it also showed the best precision (except for the CNN) after 800 steps. So if fast convergence should be achieved, a different network must be used, than if maximum precision is required.
To finally summarize some other observations: The transformer architectures show a clear trend. The more of the attention is masked, the less the network's performance. So GTF-NN has the smallest precision, TF the largest. On the flip side, because of the more strict inductive bias, the GTF-NN converges the fastest, beating GTF-NNN and TF.
The graph-conformers and poolformers fall into the middle field with similar behavior in regards to the number of interactions.

### 4.2.2 Resiliency to the Choice of Lattice Encoding

In subsection 3.2.3 it is discussed, that a pure graph network is independent to the encoding of the problem in e.g. memory.
This is demonstrated in an experiment pictured in Figure 4.7. There a CNN and a SGDCF-NNN are studied on the same Ising problem. In this case the absolute performance is irrelevant, only the relative performance difference resulting from a change to the encoding (random swaps, like described in subsection 2.1.6) will be discussed.
The investigated lattice is a linear 1D-chain, therefore the canonical representation of the lattice indicees maps them to a linear array in memory. This aligns perfectly with the shape of a 1D convolution, like it is used in the CNN. Because of that, the CNN in this case achieves a very high degree of precision (upper green graphs in Figure 4.7).
Using this relation requires the model to have architectural similarities with the underlying lattice. This not only requires manual engineering, but may even be very impractical for highly irregular lattices.
Randomizing the lattice indicees shows the CNN's dependency on the canonical structure (upper red graphs in Figure 4.7). The variance only converges to a value one magnitude further away

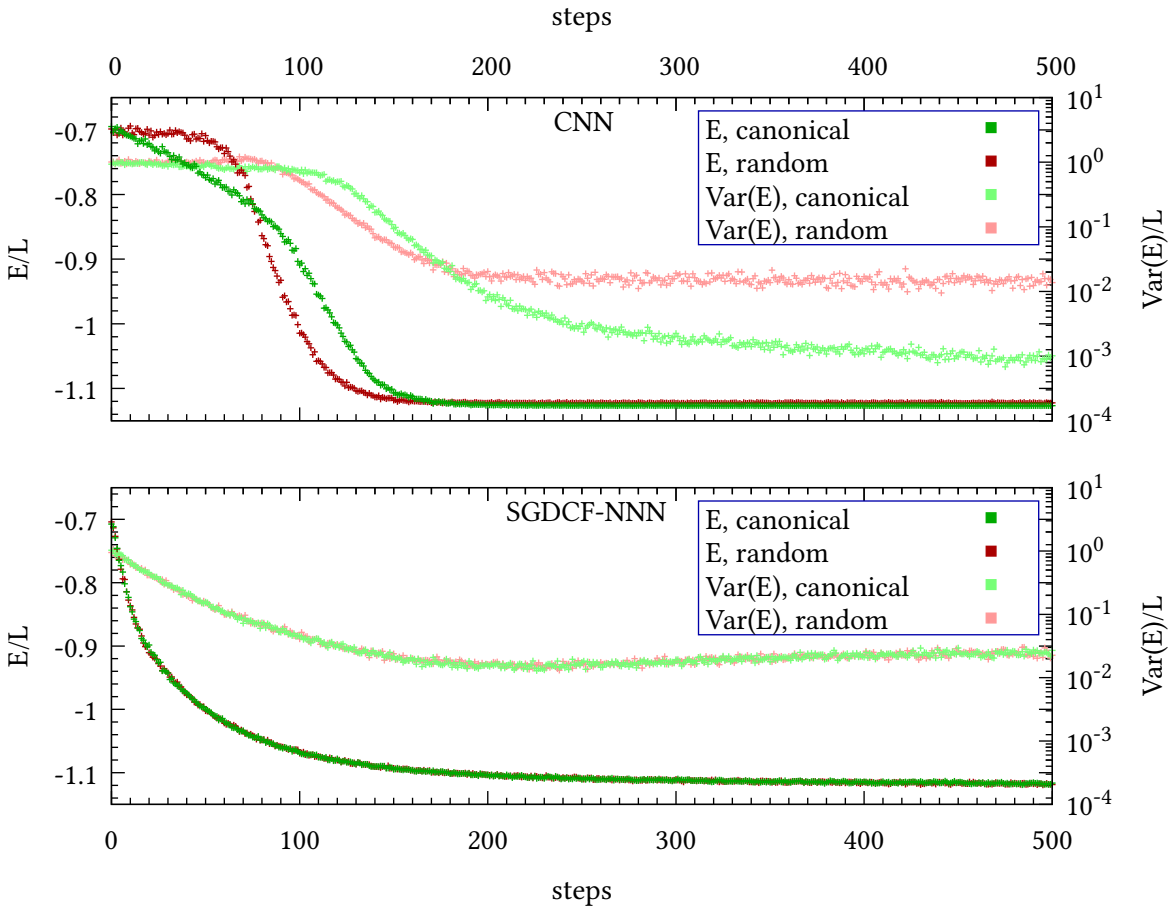


**Figure 4.7**: Visualization of the reaction of a CNN and a SGDCF-NNN network to a change in lattice encoding. The calculations are performed for a transverse field Ising model with $J = -1$, $h = -0.7$ on a 1D-linear lattice chain. The SGDCF-NNN has a depth of 1, an embed dimension of 16 and a mlp-ratio of 2, making both models have approximately the same number of parameters (CNN: 1280, SGDCF-NNN: 1312). The canonical numbering for the lattice sites is printed in green, while the random numbering and therefore unstructured representation in memory is pictured red. The variance is pictured in the same graph, colored in the respective light shade.

from zero. Instantly indicating the number of weights does no longer suffice to fully parametrize the wavefunction in that ansatz.
In comparison, the SGDCF-NNN - which is a completely graph-based model - receives absolutely no drawback from the encoding change. The course of the energy and variance are not affected. This shows the potential of graph networks to be employed as computational “black boxes” that do not need to be engineered to efficiently represent the desired lattice structure, as the adjacency matrix takes care of this automatically. Furthermore if a good performing network is found, it is more likely to be translatable to related problems.

### 4.2.3 Optimizing the Ansatz

A functional parameterization of a problem is called an *ansatz*. Not every ansatz is able to represent every wavefunction and some are way more efficient in the amount of necessary trainable parameters than others.

Even if the metaformer architecture is already chosen as an ansatz, there are still multiple different possibilities to output the final value for the wavefunction.
It is common knowledge that the quantum mechanical wavefunction is a function that maps into the complex domain. Though some problems require a complex valued wavefunction (this for example is very common for *time-dynamic* problems, but here only *static* wavefunctions are calculated), it can also be sufficient to have a completely real valued one. If the real part of the wavefunction is sufficient, it is obviously not helpful to dedicate resources to the imaginary part.
Figure 4.8 shows the four possible choices of ansatz implemented in the accompanying code. Of the four network architectures, three of them are entirely real-valued (sr, tr and spc). The forth one (sc) is a *complex valued network* [53]. The sc network ansatz will not be shown in later lineups. Because of its complex weights, every number requires the double the amount of memory (equal precision storage of real and imaginary part). Also the multiplication of complex numbers requires more operations to perform. Both of these facts make the training and evaluation of the network slower. Not only require more weights most of the time a bigger number of backpropagation steps, but also the time per step is stretched. Overall making the sc ansatz too expensive to compute in this application.
The ansatz with the real output and the two versions with the assembled complex output are measured for their rate of convergence in Figure 4.9. It can be noted, that the selected metaformer *hyperparameters* (subsection 4.2.4) influence the most and least efficient ansatz. No one ansatz can be pointed out as the best suited one, not even for a fixed lattice problem.
This once again underlines the flexibility that comes with the presented metaformer framework, but stresses the requirement for intensive testing to be able to choose the most appropriate architecture.

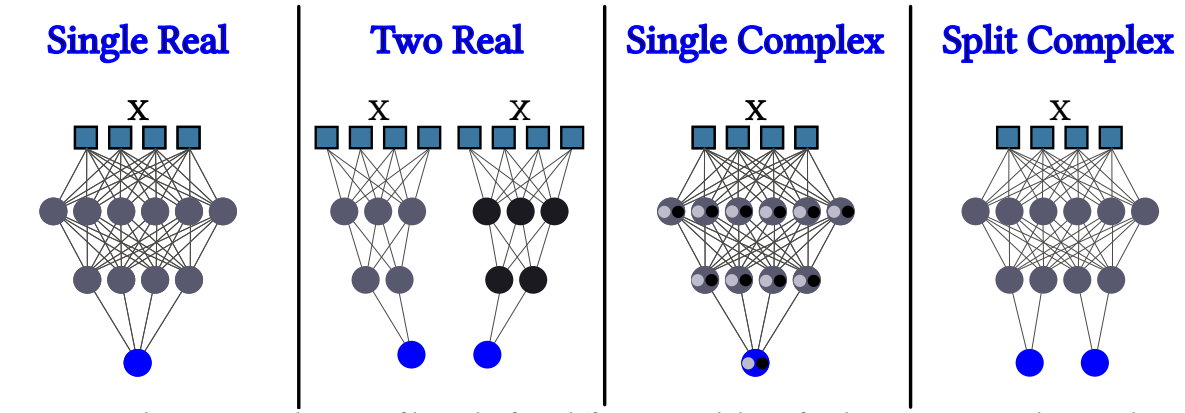


**Figure 4.8**: Schematic visualization of how the four different possibilities for the ansatz are implemented. *Single Real* gives a pure real wavefunction output. The architecture is a single network with only real parameters. *Two Real* gives a complex wavefunction output (two real numbers, one for the phase and one for the amplitude). The architecture consists of two half-size networks with only real parameters. They do not interact. *Single Complex* gives a complex wavefunction output (directly one complex type number). The architecture is a single network with every parameter being a complex number. *Split Complex* gives a complex wavefunction output (two real numbers, one for the phase and one for the amplitude). The architecture is a single network with only real parameters, but at the last stage it is split to give two outputs instead of one. The difference to the two real ansatz is that the two "halves" of the network can interact.

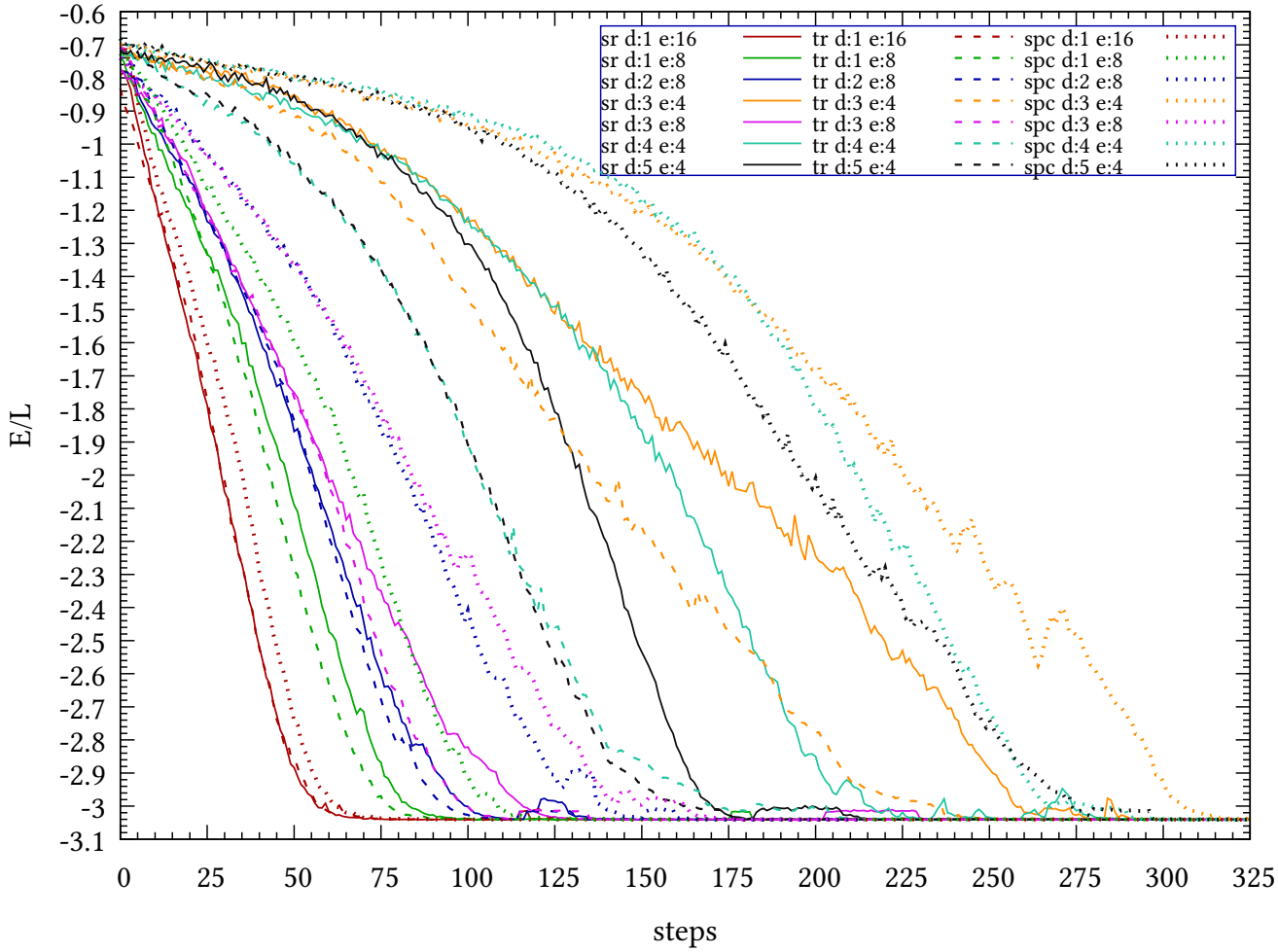


**Figure 4.9**: Comparison of hyperparameter combinations. The ansatz is encoded in the dash type, the color specifies the metaformer hyperparameters. *sr d:1 e:8* meaning *single real*, *depth 1* and *embed dimension 8*. All datasets have been calculated for a randomly encoded *trigonal_square* lattice with 64 lattice sites. Every model is of the type *GPF-NNN* with a mlp-ratio of 4. The Ising parameters are $J = -1$ and $h = -0.7$.

### 4.2.4 Choice of Hyperparameters

Like mentioned, Figure 4.9 compares not only different choices for the network ansatz, but also different hyperparameter combinations. In general, the behavior of a GPF-NNN on a trigonal_square lattice is observed.
Many different correlations could be extracted when discussing the graph. But the obtained relations could not be guaranteed to generalize to different lattice shapes or different Ising parameters. Because of that, a detailed analysis of the figure would not be very helpful.
Therefore only two clear trends will be mentioned:
First, it can be observed that all combinations of *ansatz*, *depth* and *embed dimension* converge to the same energy (they also have their variance converging in the same order of magnitude, but that data is not pictured in the thesis). This underlines the robustness of the metaformer architecture. As all combinations seemingly are a valid parameterization of the wavefunction, it is possible to select the one with the most appropriate performance behavior for the task.
Second, the shallow networks (depth of one, in Figure 4.9 red and green) seem to perform best in this task. In image classification tasks the general rule was established, that while deeper networks are generally harder to train, they perform usually better if successfully trained. An only one block deep metaformer on the other hand can probably be described as the opposite of a *deep network*. While the good performance of shallow networks can have multiple reasons, a likely one is the problem's overall difficulty. In this thesis it was hinted multiple times, that the presented lattices are not *irregular* enough to justify really deep networks. More "complicated" lattice structures or Ising problems might be required to take full advantage of the metaformer's strengths. The last section will pick up this thought.

### 4.2.5 Differences across the Phase Diagram

The same quantum mechanical system can have different states that are described by wavefunctions of different complexity. An example might be the behavior of a system as it approaches a specific temperature. This could be the *boiling point* of a fluid, at which the liquid turns into steam and starts behaving completely differently. It could also be the approach to *absolute zero*, that is commonly known to be responsible to induce unique behavior in quantum objects. Depending on the nature of the system, the approach to such a point could make the wavefunction simpler or more difficult to parameterize.
Temperature is not the only parameter that could induce such a transition, which is generally known as a *phase transition*. In our case the *strength of the transverse electromagnetic field* can also cause a phase transition.
The location of the transition is dictated by the ratio of the Ising parameters $J$ and $h$, $\lambda = \frac{h}{J}$, as well as the shape and dimensionality of the lattice. The experiments in the preceding sections all used a $\lambda = \frac{h=-0.7}{J=-1} = 0.7$. In the book *Quantum Ising Phases and Transitions in Transverse Ising Models* [10], the transverse field Ising phase transition is said to occur around $\lambda = 2$ to 4 for 2D latices of square nature.

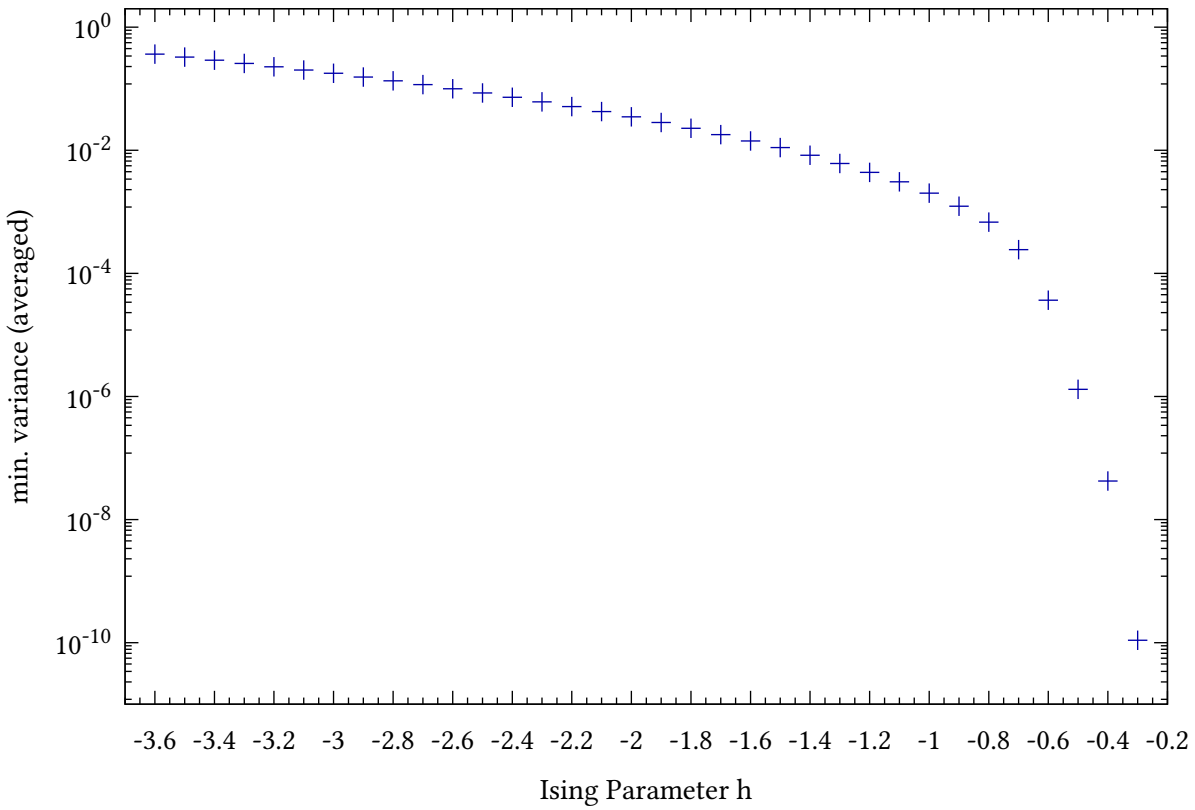


**Figure 4.10**: A ground state search was performed with a CNN as a NQS for different values of the Ising parameter $h$. The recorded values are an average of the energy variance in the logarithmic scale for about 500 steps after the CNN had converged. The lattice is of the trigonal_square type, with periodic boundary conditions and a size of 4 (64 lattice sites). The Ising $J$ parameter as always set to -1.

Experiments with a CNN on a periodic 2D-trigonal_square lattice around the region of $J = -1$, $h = -0.3$ to $-3.6$ show significant differences for the minimal achievable variance (Figure 4.10). For some $\lambda$, the smallest variance the CNN achieved was larger by several orders of magnitude compared to the experiments in the past sections. Bringing the $h$ parameter closer to zero, made the minimum variance drop quickly ($\approx 0.02$ for $h = -1.8$ down to falling to $< 10 \times 10^{-3}$ for $h \geq -0.8$). From $h = -2.5$ to $-3.6$ the minimum variance rises approximately linear on the logarithmic scale. This suggests representing the wavefunction gets exponentially harder for the CNN as $\lambda$ grows.

The reason for operating close to the phase transition region is exactly this increasing complexity of the wavefunction. Previous sections suggested, that the metaformer architecture might be too "complex" to represent the basic wavefunctions around $\lambda = 0.7$. This might be the reason, why the extremely simple CNN is outperforming the sophisticated metaformer in these examples.

Figure 4.11 validates this theory. For three different values of $\lambda$, the CNN and a SGDCF-NNN metaformer are compared. The conformer was set up to have a slightly smaller footprint than the CNN (CNN: 1024 parameters, SGDCF-NNN: 972 parameters). For the smallest value of $\lambda = 1.8$, the two networks behave the same way as in subsection 4.2.1.

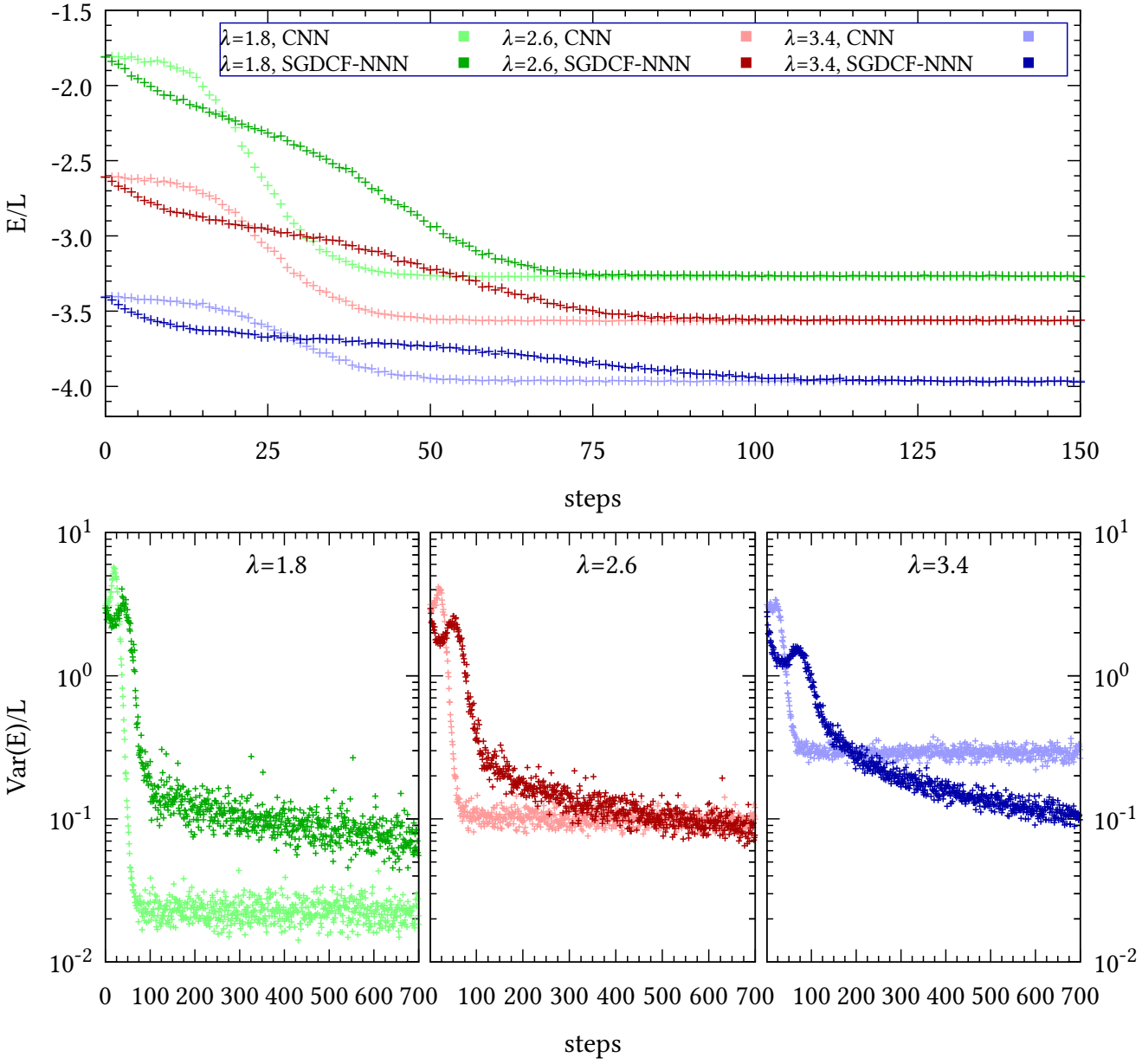


**Figure 4.11**: Detailed depiction of a selection of runs from the experiment shown in Figure 4.10. The CNN and lattice configuration are the same in both cases. Additionally the same problem was solved with a SGDCF-NNN as the NQS. The depthwise graph conformer was configured with a depth of 2, an embed dimension of 9 and a mlp-ratio of 2. The Ising $h$ parameter and therefore the different values of $\lambda$ are encoded in the curve's color. The CNN is depicted in the light shade, the SGDCF-NNN in the dark.

But by raising the value of $\lambda$ to 2.6, the conformer may finally make use of its more specialized design. The red colored curves show the two models successfully rivaling each other in terms of the minimum variance. The conformer even being slightly ahead. This can be attributed to the conformer keeping its level of performance constant, even though the CNN performs noticeably worse on this wavefunction.
For even larger values of $\lambda$ - supposedly close to the *critical point* of the phase transition - the CNN's accuracy collapses. For $\lambda = 3.4$ (blue curves), the minimum variance of the CNN is approximately one order of magnitude worse than for $\lambda = 1.8$. As the conformer is still performing without a large dropoff in accuracy, it is at this stage able to surpass the CNN and parametrize the wavefunction in a better way than its adversary.
Finally it is important to point out, that in all cases the conformer variance was still dropping at the end of the calculation, while the CNN's variance had clearly already converged. And even though the conformer had a smaller number of trainable parameters, it was able to outperform the established CNN for large enough $\lambda$ due to its strong consistency across the phase domain.
None of the architectures is perfect, as both showed unstable calculations in the regions of $\lambda < 0$. As e.g. all of the tested models quickly crashed due to *NaN* errors for $\lambda = -3.6$, it is safe to say that not all problems can be solved by the presented ansatz. Though the experiments show that metaformers provide a great deal of flexibility and resiliency against several difficulties posed by the computational problems.
Further exploration of the metaformer concept inside the phase transition region promises to provide an excellent opportunity for further research in this domain.

# 5 Conclusion

This thesis serves as an introduction for pure computer scientists to a comparably approachable problem in computational quantum many body physics - *ground state search* with the *transverse field Ising model*. An outlook into more advanced computational methods like *VMC* and *DMC* was given. This had the intent of motivating aspiring computer scientists with experiences in neural network architectures to take interest in physically motivated calculation problems as opposed to the established task like *natural language processing* or *image classification*.
The general concept of the specialized neural networks of the *metaformer architecture* were explained. Furthermore these networks were extended to be able to efficiently work on problems that can only be encoded in a *graph* representation and not in the established cubic tensor format. The *inductive biases* of different neural network building blocks were presented as a motivation for why the current machine learning calculations operate the way they do.
Solving the *image classification* problem with small networks proved that the envisioned *graph masked attention* was in fact working as expected and possesses the capability to outperform the full transformer by a whole complexity class if implemented properly. At the same time, the equivalent operations to *convolutions* and *pooling* were introduced and tested for graph representations.
A *ground state search* implementation was modified to support a range of different *2-dimensional lattices*. The search was first performed with an established *CNN* architecture and validated with the *exact diagonalization* for a problem with a small number of lattice sites.
Finally the graph metaformers were used as the *NQS* for the *VMC* ground state search. There it could be shown, that the graph architectures can easily be used as a sufficient parameterization for solving the computational search. And while the graph metaformers were slower in solving the problem compared to the established CNN, the experiments showed them offering a very high degree of customizability due to their structure, good precision and built in resiliency against changes to the graph encoding.
Concluding experiments demonstrated the metaformer's general performance benefits in representing highly complicated wavefunctions, closer to the *phase transition* of quantum mechanical systems. And while not all variations of the metaformer will always be applicable to discussed computational tasks, already the thesis' small scale experiments have shown the concept to be successful.
Thus, the field of *graph metaformers* and their application in *quantum mechanical computational problems* offers many possibilities for further research, as the vast number of applications sadly surpasses the scope of only one bachelor thesis.

# 6 Bibliography


[1] *Historical cost of computer memory and storage*, Our World in Data, https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage (visited on 09/09/2022).

[2] Jacquelyn Bulao, *How Much Data Is Created Every Day in 2022?*, Techjury, (May 9, 2022) https://techjury.net/blog/how-much-data-is-created-every-day/ (visited on 09/09/2022).

[3] *Artificial Intelligence Market Size Report, 2022-2030*, https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-ai-market (visited on 09/09/2022).

[4] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, *Summarizing CPU and GPU Design Trends with Product Data*, July 13, 2020.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need", 10.48550/ARXIV.1706.03762 (2017).

[6] W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng, and S. Yan, "MetaFormer Is Actually What You Need for Vision", 10.48550/ARXIV.2111.11418 (2021).

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", 10.48550/ARXIV.1704.04861 (2017).

[8] W. M. C. Foulkes, L. Mitas, R. J. Needs, and G. Rajagopal, "Quantum Monte Carlo simulations of solids", *Reviews of Modern Physics* **73**, 33–83 (2001).

[9] G. Carleo and M. Troyer, "Solving the quantum many-body problem with artificial neural networks", *Science* **355**, 602–606 (2017).

[10] S. Suzuki, J.-i. Inoue, and B. K. Chakrabarti, *Quantum Ising Phases and Transitions in Transverse Ising Models* (Springer Berlin, Heidelberg, 2013).

[11] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks", 10.48550/ARXIV.1806.01261 (2018).

[12] W. Demtröder, *Experimentalphysik 1*, Springer-Lehrbuch (Springer, Berlin, Heidelberg, 2015).

[13] Franz Schwabl, *Quantenmechanik (QM I)*, Springer-Lehrbuch (Springer, Berlin, Heidelberg, 2007).

[14] Ernst Ising, *Beitrag zur Theorie des Ferromagnetismus | SpringerLink*, (Feb. 1925) https://link.springer.com/article/10.1007/BF02980577 (visited on 09/09/2022).

[15] A. Läuchli, *Neuartige Ordnungsphänomene in frustrierten Quantenmagneten*, https://www.mpg.de/357126/forschungsSchwerpunkt?c=147242 (visited on 09/10/2022).

[16] *scipy.sparse.linalg.eigsh — SciPy v1.9.1 Manual*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigsh.html#scipy.sparse.linalg.eigsh (visited on 09/11/2022).

[17] *CODATA Value: Avogadro constant*, https://physics.nist.gov/cgi-bin/cuu/Value?na (visited on 09/11/2022).

[18] J. Kell, *Physis-Part implementation and Code Reference*, (2022) https://github.com/jonas-kell/bachelor-thesis-code.

[19] R. G. Melko, G. Carleo, J. Carrasquilla, and J. I. Cirac, "Restricted Boltzmann machines in quantum physics", 10.1038/s41567-019-0545-1 (2019).

[20] M. Schmitt and M. Reh, "jVMC: Versatile and performant variational Monte Carlo leveraging automated differentiation and GPU acceleration", 10.48550/ARXIV.2108.03409 (2021).

[21] A. Goldberg and J. L. Schwartz, "Integration of the Schrödinger equation in imaginary time", *Journal of Computational Physics* **1**, 433–447 (1967).

[22] A. Patel, *Hexagonal Grids*, (2013) https://www.redblobgames.com/grids/hexagons/ (visited on 09/06/2022).

[23] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators", *Neural Networks* **2**, 359–366 (1989).

[24] A. Kipp, *Cat Photo*, https://unsplash.com/photos/75715CVEJhI?utm_source=unsplash&utm_medium=referral&utm_content=creditShareLink (visited on 09/13/2022).

[25] T. M. Mitchell, *Machine Learning*, McGraw-Hill International Editions (McGraw-Hill Education, 1997).

[26] A. Krizhevsky, V. Nair, and G. Hinton, *The CIFAR-10 dataset*, https://www.cs.toronto.edu/~kriz/cifar.html (visited on 09/06/2022).

[27] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", *International Journal of Computer Vision (IJCV)* **115**, 211–252 (2015).

[28] J. Kell, *Computer-Science-Part implementation and Code Reference*, (2022) https://github.com/jonas-kell/bachelor-thesis-experiments.

[29] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization", 10.48550/ARXIV.1711.05101 (2017).

[30] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s", 10.48550/ARXIV.2201.03545 (2022).

[31] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, "Emerging Properties in Self-Supervised Vision Transformers", 10.48550/ARXIV.2104.14294 (2021).

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 10.48550/ARXIV.1810.04805 (2018).

[33] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", 10.48550/ARXIV.2010.11929 (2020).

[34] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*, Prentice-Hall signal processing series (Prentice Hall, 1984).

[35] E. Bendersky, "Depthwise separable convolutions for machine learning", (2018).

[36] H. Gao, Z. Wang, and S. Ji, "ChannelNets: Compact and Efficient Convolutional Neural Networks via Channel-Wise Convolutions", 10.48550/ARXIV.1809.01330 (2018).

[37] M. Treder, *CNN-Symmetry*, (2019) https://github.com/treder/CNN-Symmetry (visited on 09/06/2022).

[38] P. Solai, "Convolutions and Backpropagations", (2018).

[39] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", 10.48550/ARXIV.2103.14030 (2021).

[40] S. Raschka, *RNNs and Transformers for Sequence-to-Sequence Modeling*, (2021) https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L19_seq2seq_rnn-transformers__slides.pdf (visited on 09/20/2022).

[41] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, *Emerging Properties in Self-Supervised Vision Transformers*, (2021) https://github.com/facebookresearch/dino (visited on 09/06/2022).

[42] P. Tatkowski, *1D, 2D, and 3D Sinusoidal Postional Encoding (Pytorch and Tensorflow)*, (2021) https://github.com/tatp22/multidim-positional-encoding (visited on 09/06/2022).

[43] T. Mitchell, "The Need for Biases in Learning Generalizations", (1980).

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition", 10.48550/ARXIV.1512.03385 (2015).

[45] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019), pp. 8024–8035.

[46] S. A. Lab, *PoolFormer: MetaFormer Is Actually What You Need for Vision (CVPR 2022 Oral)*, (2021) https://github.com/sail-sg/poolformer (visited on 09/06/2022).

[47] A. Rogozhnikov, *Einops: Clear and Reliable Tensor Manipulations with Einstein-like Notation*, (2022) https://openreview.net/forum?id=oapKSVM2bcj.

[48] N. Qian, "On the momentum term in gradient descent learning algorithms", *Neural Networks* **12**, 145–151 (1999).

[49] I. Loshchilov and F. Hutter, "Fixing Weight Decay Regularization in Adam", (2018).

[50] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, *JAX: composable transformations of Python+NumPy programs*, version 0.3.13, (2018) http://github.com/google/jax.

[51] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee, *Flax: A neural network library and ecosystem for JAX*, version 0.6.0, (2020) http://github.com/google/flax.

[52] M. Schmitt, *jVMC*, version 1.1.2, https://github.com/markusschmitt/vmc_jax.

[53] C. Trabelsi, O. Bilaniuk, Y. Zhang, D. Serdyuk, S. Subramanian, J. F. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. J. Pal, "Deep Complex Networks", 10.48550/ARXIV.1705.09792 (2017).

[54] J. Kell, *Investigation of transformer architectures for geometrical graph structures and their application to two-dimensional spin systems*, (2022) https://github.com/jonas-kell/bachelor-thesis-documents.

# 7 Appendix

## 7.1 Lattice Visualization

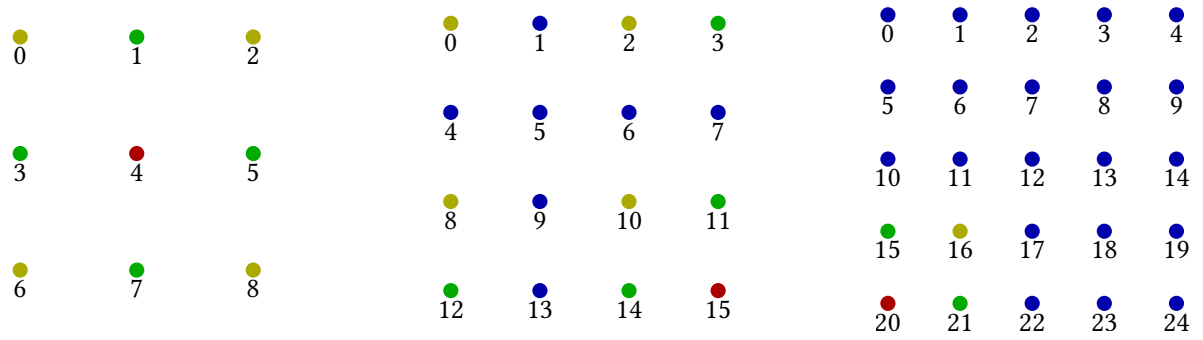


**Figure 7.1**: A visualization of the **2D-square** lattice structure, measured in this thesis. The lattices from left to right can be described by the parameters
1: *size=2, non-periodic* 2: *size=3, periodic* 3: *size=4, non-periodic*

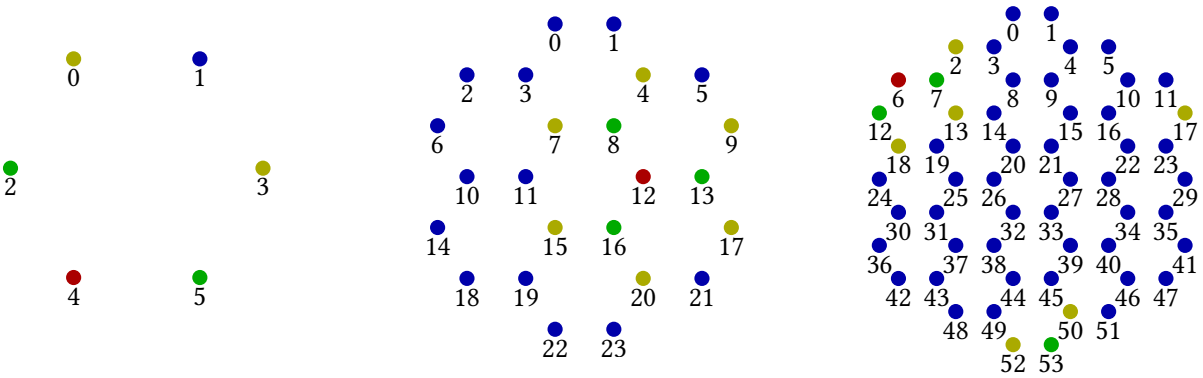


**Figure 7.2**: A visualization of the **2D-hexagonal** lattice structure, measured in this thesis. The lattices from left to right can be described by the parameters
1: *size=1, non-periodic* 2: *size=2, non-periodic* 3: *size=3, periodic*

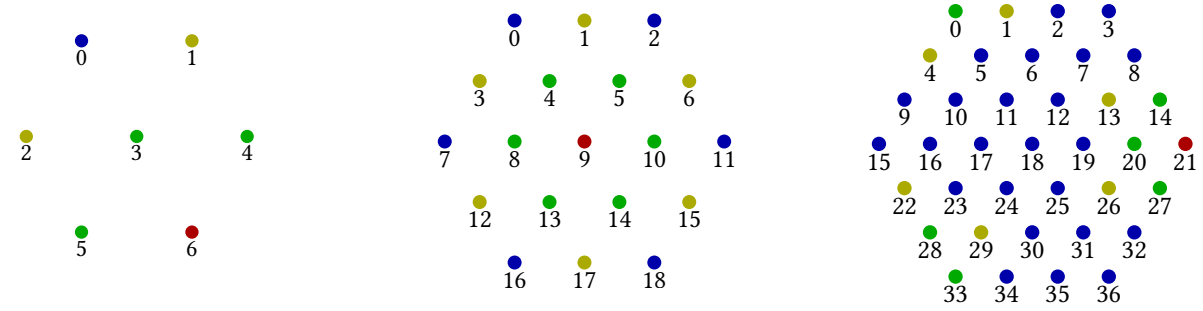


**Figure 7.3**: A visualization of the **2D-trigonal_hexagonal** lattice structure, measured in this thesis. The lattices from left to right can be described by the parameters
1: *size=1, non-periodic* 2: *size=2, non-periodic* 3: *size=3, periodic*

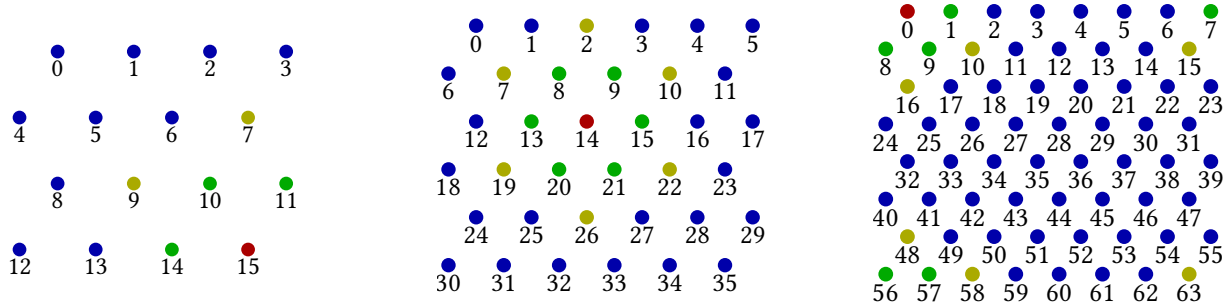


**Figure 7.4**: A visualization of the **2D-trigonal_square** lattice structure, measured in this thesis. The lattices from left to right can be described by the parameters
1: *size=2, non-periodic* 2: *size=3, non-periodic* 3: *size=4, periodic*

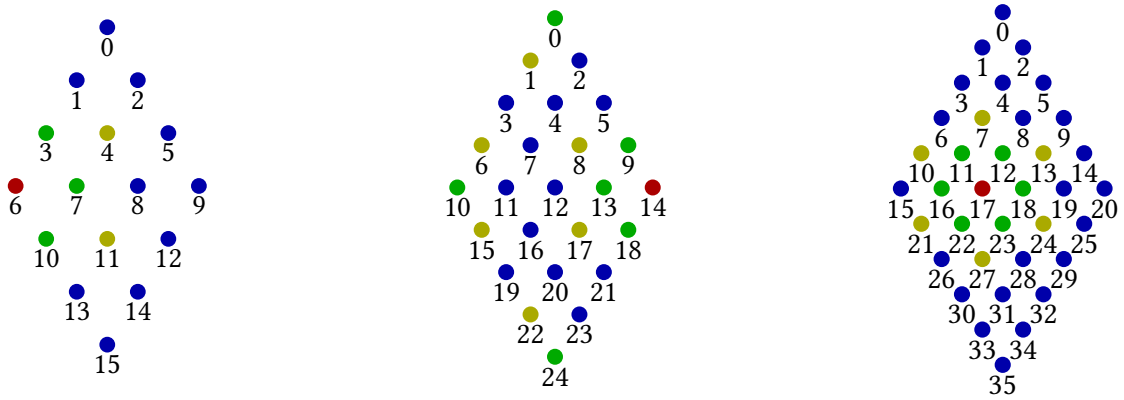


**Figure 7.5**: A visualization of the **2D-trigonal_diamond** lattice structure, measured in this thesis. The lattices from left to right can be described by the parameters
1: *size=3, non-periodic* 2: *size=4, periodic* 3: *size=5, non-periodic*

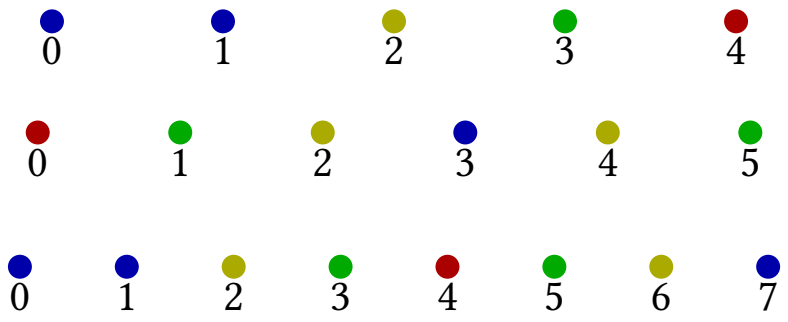


**Figure 7.6**: A visualization of the **1D-linear** lattice structure, measured in this thesis. The lattices from top to bottom can be described by the parameters
1: *size=5, non-periodic* 2: *size=6, periodic* 3: *size=8, non-periodic*

## 7.2 Scaled Dot Product Attention (jax)

[18]: /models/metaformer.py

```
149     def __call__(self, x):
150         N, C = x.shape
151 
152         qkv = (
153             self.qkv(x)
154             .reshape(N, 3, self.num_heads, C // self.num_heads)
155             .transpose(1, 2, 0, 3)
156         )
157 
158         q, k, v = qkv[0], qkv[1], qkv[2]
159 
160         attn = (q @ k.transpose(0, 2, 1)) * self.scale
161 
162         attn = self.graph_projection(attn)  # modifies to graph
            ↪ attention
163 
164         attn = nn.softmax(attn, axis=2)
165 
166         x = (attn @ v).transpose(0, 2, 1).reshape(N, C)
167 
168         x = self.proj(x)
169 
170         return x
```

## 7.3 Graph Conformer Module (jax)

[18]: /models/metaformer.py

```
242 class GraphMaskConvolution(nn.Module):
243     """Graph-Mask implementation that uses adding neighbor
        ↪ interaction in order to simulate convolutions"""
244 
245     lattice_parameters: LatticeParameters
246     embed_dim: int
247     graph_layer: Literal["symm_nn", "symm_nnn"] = "symm_nn"
248     complex_values: bool = False
249 
```

```
    def setup(self):
        self.factors = self.param(
            "factors",
            complex_init if self.complex_values else
             ↪  nn.initializers.normal(),
            (3, self.embed_dim),
        )

    def __call__(self, x):
        res = jnp.einsum(
            "d,nd->nd",
            self.factors[0],
            jnp.matmul(
                self.lattice_parameters["adjacency_matrices"]
                 ↪  ["add_self_matrix"],
                 ↪  x
            ),
        )
        if self.graph_layer in ["symm_nn", "symm_nnn"]:
            res += jnp.einsum(
                "d,nd->nd",
                self.factors[1],
                jnp.matmul(
                    self.lattice_parameters["adjacency_matrices"]
                     ↪  ["add_nn_matrix"],
                     ↪  x
                ),
            )
        if self.graph_layer == "symm_nnn":
            res += jnp.einsum(
                "d,nd->nd",
                self.factors[2],
                jnp.matmul(
                    self.lattice_parameters["adjacency_matrices"]
                     ↪  ["add_nnn_matrix"],
                     ↪  x
                ),
            )

        return x
```

## 7.4 Graph Poolformer Module (jax)

[18]: /models/metaformer.py

```python
class GraphMaskPooling(nn.Module):
    """Graph-Mask implementation that uses averaging neighbor interaction in order to simulate pooling"""

    lattice_parameters: LatticeParameters
    graph_layer: Literal["symm_nn", "symm_nnn"] = "symm_nn"

    # interaction scaling here has been chosen to be uniform. (1, 1, 1). This could be explored further
    def setup(self):
        self.pooling_interaction_matrix = (
            1 * self.lattice_parameters["adjacency_matrices"]["avg_self_matrix"]
        )

        if self.graph_layer in ["symm_nn", "symm_nnn"]:
            self.pooling_interaction_matrix += (
                1 * self.lattice_parameters["adjacency_matrices"]["avg_nn_matrix"]
            )

        if self.graph_layer == "symm_nnn":
            self.pooling_interaction_matrix += (
                1 * self.lattice_parameters["adjacency_matrices"]["avg_nnn_matrix"]
            )

        self.pooling_interaction_matrix = stop_gradient(self.pooling_interaction_matrix)

    def __call__(self, x):
        x = jnp.matmul(self.pooling_interaction_matrix, x)

        return x
```