

Machine Learning for Quantum Many-Body Physics

Efficient Representation of Vertex Functions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Dipl.-Ing. Sebastian Hepp

Matrikelnummer 01015083

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.in rer.nat. Sabine Andergassen

Mitwirkung: Dr. Daniel Springer

Wien, 29. Jänner 2026

Sebastian Hepp

Sabine Andergassen



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Machine Learning for Quantum Many-Body Physics

Efficient Representation of Vertex Functions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Dipl.-Ing. Sebastian Hepp

Registration Number 01015083

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.in rer.nat. Sabine Andergassen

Assistance: Dr. Daniel Springer

Vienna, January 29, 2026

Sebastian Hepp

Sabine Andergassen

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Sebastian Hepp

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 29. Jänner 2026

Sebastian Hepp

Acknowledgements

I would like to thank my supervisors Prof. Andergassen and Dr. Springer for their support and insightful discussions, which allowed me to successfully participate in research on this complex domain-specific topic. Their prompt responses and constructive feedback in our meetings constantly moved the work forward.

My gratitude goes to my mother, who encouraged my curiosity and supported my educational endeavors from the very beginning.

I am also thankful to my girlfriend, who provided both distractions when needed and encouragement when it felt tiring to continue.

Kurzfassung

In dieser Arbeit werden Techniken des Maschinenlernens in der Vielkörperphysik angewendet. Der Fokus liegt auf der Vertexfunktion, einem hochdimensionalen Objekt, das die Wechselwirkungen von Teilchen abbildet und eine Herausforderung für Interpretation und Berechnungen darstellt.

Inspiziert von Methodiken in der Objekterkennung entwickeln wir ein Konzept, bei dem ein Autoencoder zum Erlernen einer reduzierten Repräsentation der Vertexfunktion eingesetzt wird. Um quantenphysikalische Besonderheiten erfassen zu können, wird eine Samplingstrategie für Vertices entwickelt. Im Kern dieser Arbeit wird untersucht, ob der dimensional reduzierte Vertexraum ein physikalisches Verständnis der Vertexfunktion erfasst, erlaubt Vertices nicht gelernter Phasen zu extrapolieren und ermöglicht Eigenschaften eines Vertex mit höherer Konfidenz abzuleiten. Der reduzierte Vertexraum wird weitergehend analysiert, indem ein Klassifizierer gelernt wird, um die Phase aus der reduzierten Repräsentation abzuleiten. Abschließend wird ein Autoencoder trainiert, um den Nachbarn eines Eingabevertex vorherzusagen.

Die Experimente zeigen, dass der Autoencoder beim Rekonstruieren reduzierter Vertices sehr niedrige Fehler erreicht, selbst wenn ungelernete Phasen extrapoliert werden. Das demonstriert die Fähigkeit des Modells die physikalischen Grundsätze der Vertexfunktion zu erfassen. Zudem wird lediglich ein sehr geringer Teil der Daten eines Vertex benötigt, um einen effektiven Vertexraum zu erlernen. Durch die Verwendung reduzierter Vertices konnte die Genauigkeit der Klassifikation nicht verbessert werden, die Ergebnisse bestätigen allerdings, dass diese wesentliche Eigenschaften der Vertexfunktion erfassen.

Für das Vorhersagen eines benachbarten Vertex konnten bisher keine zufriedenstellenden Ergebnisse erzielt werden. Möglichkeiten für eine Verbesserung könnten sein komplexere Modellarchitekturen wie das U-Net einzusetzen oder einen für Rekonstruktion trainierten Autoencoder durch anschließendes Fine-Tuning anzupassen.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

In this work, machine learning techniques are applied to many-body physics, focusing on the vertex function, a high-dimensional object encoding particle interactions, posing a challenge for interpretation and computation.

Inspired by successful approaches in object detection, we develop a framework using an autoencoder neural network to learn a lower-dimensional representation of the vertex function. To capture quantum-physical peculiarities, a subsampling strategy for vertices is developed. In the core of this work we investigate if the dimensionally reduced vertex space captures physically meaningful features and a physical understanding of the vertex function, allows to extrapolate to vertices of unseen phases and enables to infer vertex features more confidently. The reduced vertex space is analyzed further, by training a classifier to infer the phase from a reduced vertex representation. Finally, the autoencoder is trained to predict the neighbor of an input vertex.

The experiments show that the autoencoder achieves very low errors, when reconstructing vertices from the reduced space, even when extrapolating to unseen phases, demonstrating the models ability to capture the defining features and physical fundamentals of the vertex function. Furthermore, only a very small amount of the data in a vertex is required to learn a well performing vertex space. While the reduced vertex space does not improve classification accuracy compared to using original vertices, the results confirm that essential vertex features captured.

When predicting a neighboring vertex, no satisfying results could be achieved by directly training an autoencoder. Suggestions for improvement include using a more advanced model architecture like U-Net or fine-tuning an autoencoder for reduction and reconstruction towards the new task.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	ix
Abstract	xi
Contents	xiii
1 Introduction	1
2 Physical Modeling of Electron Interaction	5
2.1 State of the Art	7
2.2 Data	8
3 Machine Learning Framework	13
4 Latent Space Representation of Vertices	17
4.1 Training data and sampling strategy	18
4.2 Autoencoder	20
5 Results	25
5.1 Autoencoder training convergence	26
5.2 Reconstructing vertices from latent space representations	27
5.3 Analysis of latent space	37
5.4 Predicting new vertices	48
6 Conclusion and Outlook	53
6.1 Vertex latent space model	53
6.2 Latent space analysis	54
6.3 Further research	55
Bibliography	59
	xiii



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Introduction

In recent years, the utilization of machine learning (ML) techniques for theoretical and computational physics has gained significant momentum. Machine learning enables to identify patterns, reduce dimensionality, and approximate complex functions directly from data, generated by simulations or experiments. This has the advantage that any underlying mathematical model does not need to be solved, which – especially in quantum physics – may become very complex.

One such area where we have to deal with complex mathematical descriptions, is many-body physics. The objective of this subfield of quantum physics is to understand the collective behavior of interacting particles. The ability to model many-body systems eventually helps to determine properties of materials and enables to understand which parameters influence those properties and how. This understanding allows to design materials with certain properties, such as superconductivity [Tos11].

One particular approach studies the movement of electrons between positions on a lattice. The paradigmatic Hubbard model describes these movements including quantum-mechanical correlation. The resulting phase diagram exhibits an anti-ferromagnetic (AFM), a superconducting (SC), and a ferromagnetic (FM) phase. The Hubbard model can be approximately solved by the functional renormalization group method [DSMacT⁺22]. The resulting effective interaction is encoded in the so-called vertex function Γ . This function covers the complete interaction dynamics and, therefore, fully describes the system. In this work, we consider a vertex function defined for a

two-dimensional lattice and a fixed frequency-vector of length 24, which results in a tensor of size 24^6 .

Due to the high dimensionality of the vertex function, it is difficult to interpret. However, the underlying flow equations are highly correlated and have been shown to be reducible to fewer effective equations using machine learning [Sum22]. This indicates the presence of a non-trivial structure in the vertex function, and possibly in the functions of the Hubbard model. Understanding and being able to efficiently use the vertex function in the computation of physical quantities, would enable to feasibly solve questions about their structure, infer knowledge about their properties and determine the conditions for designing materials with specific properties.

In this work, we aim to employ a machine learning framework to learn the underlying physics directly from the data of a vertex function. For this we use an autoencoder which encodes the input data to a smaller dimensional latent space. This approach is an established technique in object detection. It eliminates redundant and non-essential information, thereby reducing the object to its fundamental physical features.

We investigate first how well such a latent space can learn the underlying vertex function from data by reconstructing encoded vertices not seen during training, and second how well its knowledge can be transferred to vertices from yet unseen phases, which enables identifying vertices for materials in specific phases not yet studied.

We then analyze the latent space in the context of one exemplary usecase, where we train a classifier that infers the phase of a vertex from its latent space representation. Thereby, we first evaluate whether the latent space has learned the physically meaningful features of the vertex, and second whether the latent space can be interpreted with more confidence than the original vertex.

Finally, we investigate whether it is possible to train an autoencoder to infer the next vertex on the respective parameter scale from a given vertex. This possibility is interesting for designing materials, as it would enable to extrapolate from known systems and obtain the vertex for yet unknown systems.

The thesis is structured as follows: In the next chapter 2, we explain the physics fundamentals of the vertex functions and present the dataset. In chapter 3 the machine learning framework PhysML, a product of this thesis, is introduced, which is essential for the efficient and well-documented model training and evaluation. Chapter 4 describes the methodology for the training and evaluation of the autoencoder to learn the latent

space representation of vertices for different scenarios, which help us investigate different aspects of the vertex function. The results are discussed and interpreted in chapter 5, where we also present the methodology for the phase classification usecase and its results. We conclude with an investigation of the potential to predict the next vertex on the respective parameter range. In the final chapter 6, all findings and their implications on the vertex function representations are summarized, and summarize potential methodological improvements and further investigations.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Physical Modeling of Electron Interaction

The objective of many-body physics is to understand, describe, and predict the collective behavior of systems composed of a large number of interacting particles. The ability to model many-body systems eventually helps to determine properties of materials and enables to understand which parameters influence those properties and how. This understanding allows to design materials with certain properties, such as superconductivity [Tos11].

One particular approach of many-body physics uses lattice models, where electrons move between lattice sites with a certain hopping amplitude t and interaction potential U in case of double occupancy of an atomic site.

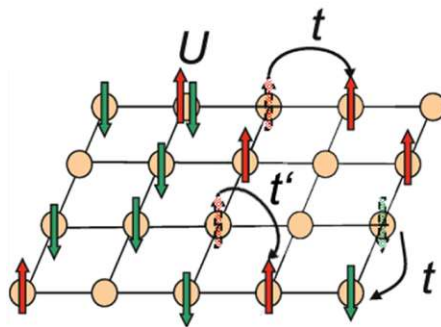


Figure 2.1: Movement of electrons on the lattice in x-/y-direction and diagonal [YIM18] (edited).

The prototypical Hubbard model [Hub63] offers a condensed description of interactions in strongly correlated systems, including those between electrons when they occupy the same site on a lattice. This results in a nuanced phase diagram, which helps characterizing the system. We use the model written in second quantized notation defined by the Hamiltonian

$$H = \sum_{\mathbf{k},\sigma} \xi_{\mathbf{k}} c_{\mathbf{k}\sigma}^\dagger c_{\mathbf{k}\sigma} - \mu \sum_{i,\sigma} n_{i\sigma} + U \sum_i \hat{n}_{i\uparrow} \hat{n}_{i\downarrow}, \quad (2.1)$$

where the first term describes the kinetic energy of electrons hopping between neighboring lattice sites with a momentum vector \mathbf{k} and spin σ . The expression $\xi_{\mathbf{k}} = -2t [\cos(k_x) + \cos(k_y)] - 4t' \cos(k_x) \cos(k_y)$ describes how the electrons move between lattice sites, where t is the electron hopping amplitude between nearest neighbors and t' is the amplitude between next-nearest neighbors (see Fig. 2.1). $c_{k\sigma}^\dagger$ and $c_{k\sigma}$ are the creation and annihilation operators for an electron with momentum k and spin σ . The second term in the Hamiltonian is the sum of all electrons on the lattice multiplied by the chemical potential μ , where $n_{i\sigma}$ yields the number of electrons on site i . The third term describes the Coulomb repulsion of two electrons occupying the same site i , where $U > 0$ is the intensity of repulsive interaction that correlates electron motion by disfavoring configurations with two electrons per site [ZMK⁺24]. For $U/t \ll 1$ the kinetic energy dominates: the material is likely metallic and possesses high conductive properties. For $U/t \gg 1$ the Coulomb interaction dominates: the material is an insulator, most electrons are localized by strong Coulomb repulsion [EFG⁺05].

When approximately solving the lattice problem in equation 2.1, we can use the so-called self-energy Σ and vertex function Γ to encode the interaction of a single and a pair of electrons with the system. These objects contain the full momentum- and frequency-resolved interaction dynamics and, therefore, are sufficient to fully describe the system. However, due to their high-dimensional structure, they are difficult to interpret. For the present work, we will be interested in the two-particle vertex

$$\Gamma(k_1, k_2, k_3), \quad (2.2)$$

where $k_i = (\mathbf{k}_i, \nu_i)$ denotes a vector of momentum \mathbf{k}_i and frequency ν_i [ZMK⁺24].

From the calculated vertices the eigenvalues as well as different physical observables can be computed, which enable to infer the phase diagram of a material, as shown in Fig. 2.2. In this work, the model undergoes phase transitions upon modification of the parameters μ and t' . With a gradual increment of μ and t' an antiferromagnetic (AFM)

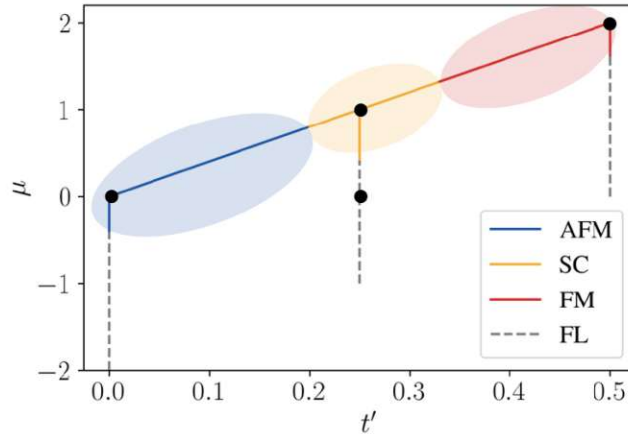


Figure 2.2: Phase diagram for the Hubbard model [ZMK⁺24]

phase changes first to superconducting (SC) or a non-ordered Fermi liquid (FL) and then to ferromagnetic (FM).

For a two-dimensional lattice – and therefore a two-dimensional momentum vector – and a fixed frequency, the resulting vertex function can be represented as a six-dimensional tensor. It is calculated using the functional renormalization group, which involves millions of coupled non-linear equations. These magnitudes require large computational and memory resources, which increase with increasing system interaction and decreasing temperature [ZMK⁺24]. This complexity makes an exhaustive search for new materials infeasible. In this work, machine learning is applied to find smaller-dimensional representations, which can be used much more computationally efficient.

2.1 State of the Art

First steps in the research of machine learning approaches applied to vertex functions were focusing on compression of vertices based on the wavelet transform [MDT⁺24][Zab24], principal component analysis (PCA) [ZMK⁺24] and autoencoder neural networks [Zab24][ZMK⁺24].

The wavelet transform decomposes a signal into frequencies similar to the Fourier transform. However, in addition it resolves temporal information of the signal, thus representing complex and dynamic data in a few significant coefficients [Zab24][MDT⁺24]. The wavelet transform is then recursively decomposed in high and low frequencies. For compression, in each recursion step, thresholding is applied to the high frequency values,

setting all values outside a chosen quantile to 0.

In contrast, PCA identifies a new set of variables, called principal components, by decomposing the covariance matrix to get its eigenvectors. The first few selected eigenvectors ordered by the eigenvalues represent the principal components onto which the data space is projected. Thus PCA maximizes the variance of the data and transforms it into a new coordinate system. In the work of Zang et al. this achieved very low dimensional representation from originally 10^6 down to 10-20 components, while accurately capturing the physics [ZMK⁺24].

The autoencoder is a type of neural network that reduces the input data dimensionally into a smaller representation, a so-called latent space. The data is then upscaled again to the original dimensions. The network is optimized by comparing input and final output using an appropriate loss function. Zang et al. used 3d-convolutional layers for their autoencoder. The full vertices as three-dimensional tensors are used as inputs, after down-sampling the size from 576^3 to 144^3 to reduce computational cost.

PCA was found to be superior to the autoencoder in achieving the lowest reconstruction error for a fixed dimensionality, supposedly because convolution operates locally and thus cannot capture the global patterns of non-local (quantum) interactions [ZMK⁺24].

2.2 Data

The dataset used in this project is the same as in Zang et al. [ZMK⁺24] and consists of 51 vertices. Each data sample corresponds to a vertex function Γ for a specific (μ, t') parameter pair according to Fig. 2.2.

The vertices are calculated using the functional renormalization group (fRG) method [MSH⁺12], a computational tool in theoretical physics to study strongly interacting systems. In essence, with fRG a system is solved by a set of coupled non-linear differential flow equations [HS01]. The solution are N_k^{3d} complex numbers, where N_k is the discretized momentum space and d is the spatial dimension [ZMK⁺24], which is two in our case of studying a lattice. Thus, one obtains the momentum space, which was discretized to $N_k = 24$, along two spatial dimensions for each of the three momenta, resulting in a six-dimensional tensor with axis length 24 containing more than 191 million values.

Fig. 2.3 shows a visualization of a vertex for each phase (AFM, SC, FM). For each vertex a slice through each momentum (k_1, k_2, k_3) – with the other two momenta fixed at $x, y = 18$ – is shown. The axes represent the x- and y-coordinates for the momentum.

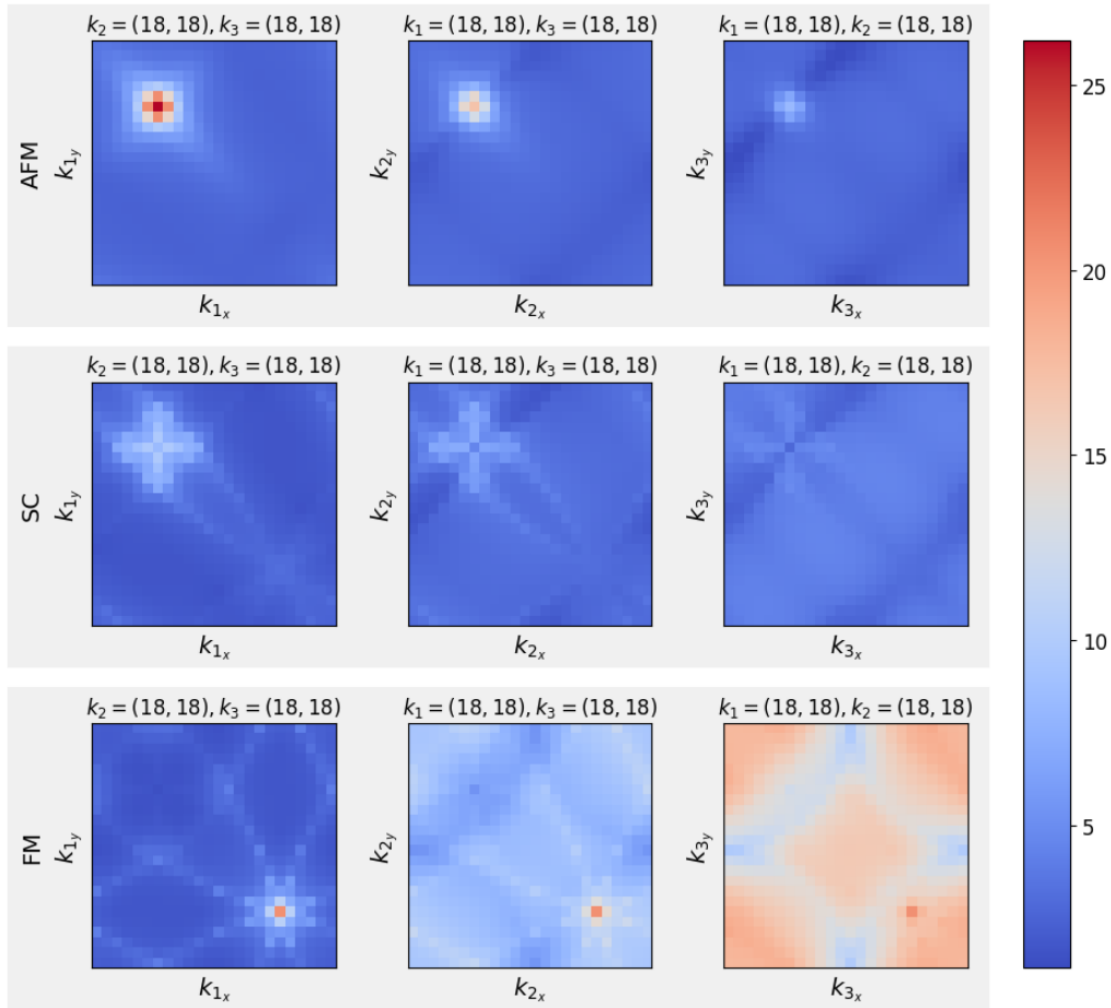


Figure 2.3: Visualization of sections through 3 vertices of different phases and 3 different momenta k sliced at index 18 (of 24) along each axis.

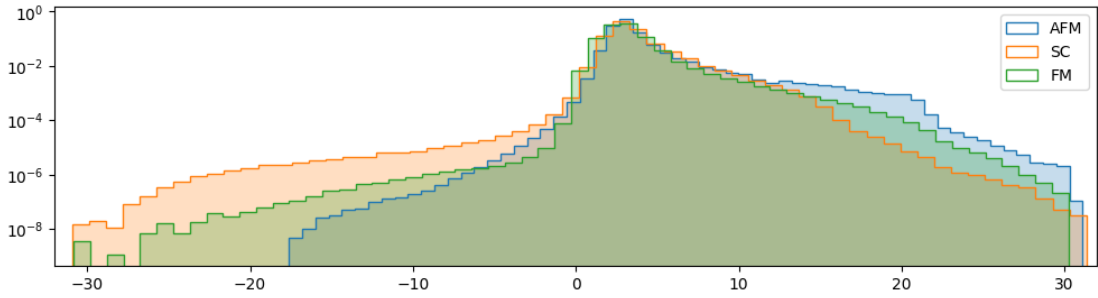


Figure 2.4: Histogram (density) for absolute values across all vertices for each phase. The y-axis is shown in log-scale to make the full range of values visible.

To some extent we can distinguish phases thanks to certain visual features, but they transition smoothly between phases.

To further examine how phases differ, some statistical analysis was performed. Fig. 2.4 shows the histogram of absolute values across all vertices for each phase. The graphic shows similar distributions. The values per phase have a similar mean between ca. 3.2 and 3.5. The only remarkable difference is that the value range for the AFM-phase starts around -17 while the others start around -31. From descriptive statistics alone, the phases cannot be well distinguished.

The matrix in Fig. 2.5 shows the correlation between each pair of vertices. The correlation between two vertices \mathbf{V}_i and \mathbf{V}_j is calculated as

$$\text{corr}_{i,j} = \mathbf{V}_{i\text{norm}} \cdot \mathbf{V}_{j\text{norm}} \quad (2.3)$$

with

$$\mathbf{V}_{\text{norm}} = \frac{\mathbf{V}}{\sqrt{\sum_{x \in \mathbf{V}} x^2}}, \quad (2.4)$$

where \mathbf{V} is a 6-dimensional tensor flattened to a vector.

The correlation is between 0.59 and 1.0. Although, approximately, the correlation is lower, the farther apart vertices are with respect to t' , it is still considerable for all pairs of vertices. We can also fuzzily see an increase in correlation in sections of the t' -range corresponding to the same phase. Lastly, we can see a less clear correlation between vertices of the AFM- and SC-phase (with t' roughly from 0.0 to 0.2 and from 0.2 to 0.33, respectively) as well, whereas vertices of the FM-phase (t' between ca. 0.33 and 0.5) are much more distinguishable.

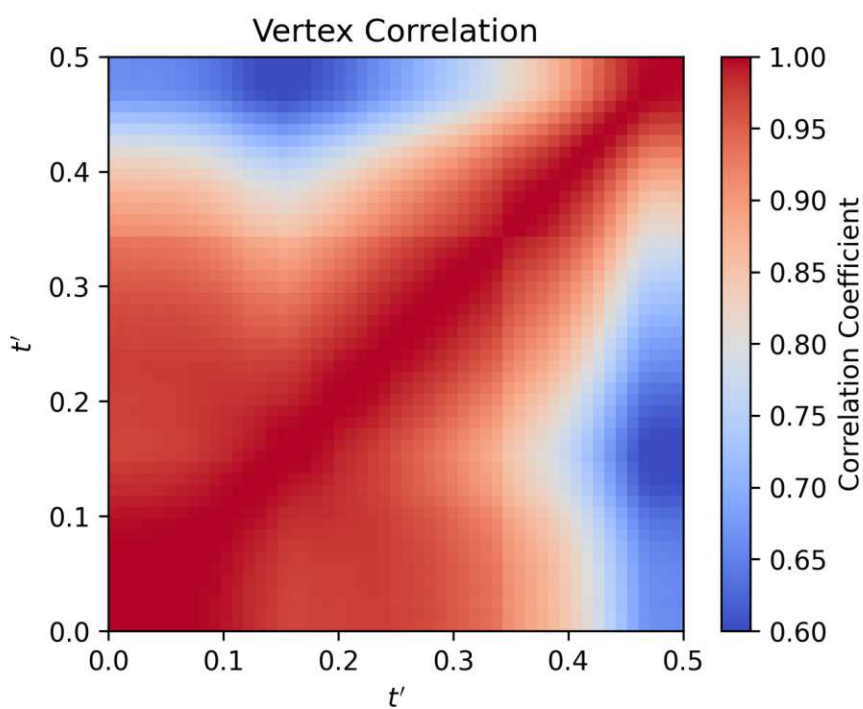


Figure 2.5: Correlation between vertices with t' between 0 and 0.5 for real and encoded vertices using an autoencoder with a latent space dimension of 32.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Machine Learning Framework

To efficiently implement and run different machine learning models and algorithms, the machine learning framework PhysML¹ was implemented.

The core objective is to standardize data pipelines, dynamically construct a modular machine learning architecture and to parallelize training on large GPU clusters. The framework is written in Python 3.12 and primarily uses PyTorch and PyTorch Lightning to perform deep learning. The framework is based on a set of base classes with pre-implemented functions, that can be overwritten to adapt to new projects.

The main components of PhysML are five modules. The *config* module implements Python *dataclasses*, which contain all settings of the project. These include all training and model parameters and callbacks for e.g. logging or early stopping. Furthermore, the config-class contains qualified names of modules of the training architecture, such as for data loading or training execution, and training functions like optimizers and activations, which are imported dynamically. The config-class is serializable and can be saved to or loaded from a JSON-file. This approach enables to fully configure a model training through a human- and machine-readable file, without the need to intervene in the code and furthermore simplifies executing series of trainings, e.g. for hyperparameter optimization or analyzing different scenarios.

In the *load_data* module PyTorch Dataset classes are implemented, which load the data from files and prepare it for model training or validation. The *models* module

¹<https://github.com/DanielSpringer/PhysML>

3. MACHINE LEARNING FRAMEWORK

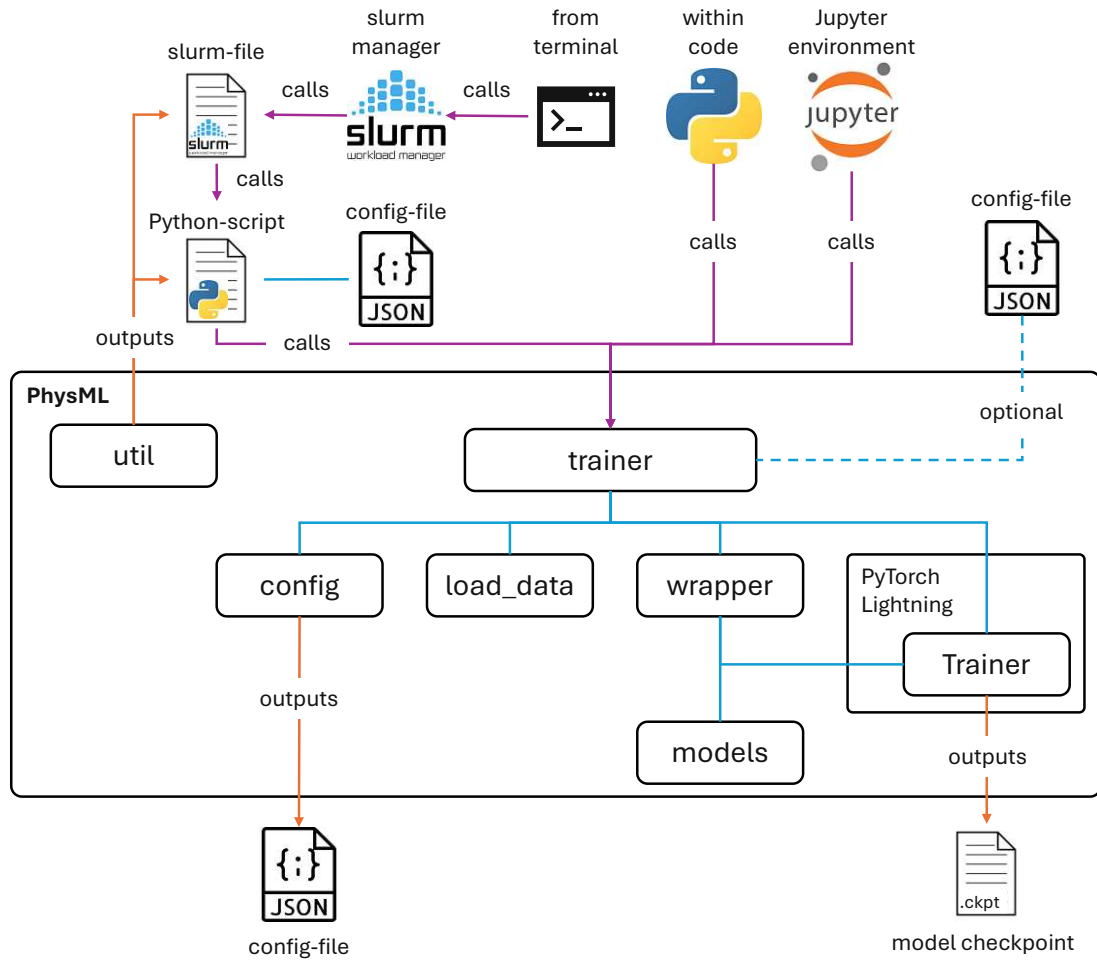


Figure 3.1: Architecture and integration of PhysML framework.

contains neural network architectures in form of PyTorch modules. The *wrapper* module contains classes based on PyTorch Lightning modules which host model and optimizer and execute the model training and validation steps. Finally, the *trainer* module dynamically creates, initializes and combines the other components and executes jobs using a PyTorch Lightning Trainer. After the training the configuration is stored to disk. The progress can be logged and additional callbacks allow to store training steps to disk.

In each class the processes are broken down into sufficiently many methods. Therefore, for new projects or variants, a child class can be derived from an existing class, and the methods that need to be adapted can be overwritten. PhysML also uses extensive type hinting, including generic types for base classes, enabling developers a clean perspective on the requirements and return values of a respective function or code element. To

further reduce the implementation effort, a function exists which creates a python script and a *Slurm*-file. The *Slurm*-file contains settings for running the job and executes the python script, which runs the machine learning task, on a *Slurm*-managed cluster. The function takes a configuration JSON-file or keyword arguments to configure both the slurm-job and the ML-task. With this framework any deep learning task can be run from command-line using a configuration-JSON but also from a Jupyter-notebook. All configurations are saved to the task-specific output folder as a JSON-file. By default intermediate validation results are logged as well for convergence analysis.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Latent Space Representation of Vertices

The frequency and momentum dependent vertex function contains the full information on interaction processes depending on momentum and frequency. However, due to its high dimensionality, it is difficult to interpret.

Therefore, we aim to employ a machine learning framework to learn the underlying physics directly from data. A common tool to facilitate interpreting the information encoded in high-dimensional objects is neural networks that create low dimensional representations in a so-called latent space. In this latent space, the information contained in the vertex should not just be compressed but reduced to essential abstract rules (in the form of patterns). The hypothesis is that this low-dimensional object can then be used in other models to infer any desired information more easily and with higher certainty, since all potentially superfluous information has been eliminated.

In this chapter, first, the sampling strategy for the training data is discussed, which is fundamental for the model to learn the underlying concepts in the data with quantum-physical peculiarities. Then the autoencoder and its model architecture is introduced as our model of choice. Furthermore, we explain the training process, which, based on different training scenarios and hyperparameter settings, focuses on different aspects of the latent space. We also validate the model, to assure that it is trained as intended, and finally describe the evaluation process.

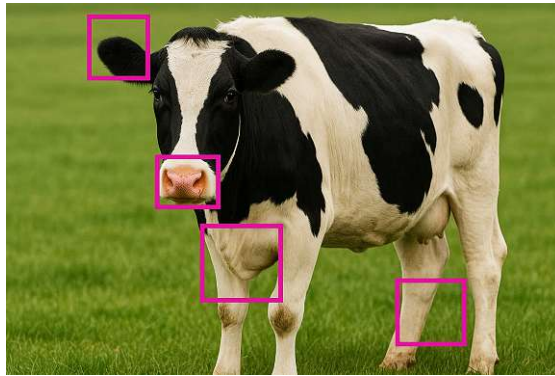


Figure 4.1: Object detection training: Decomposing an object into sub-elements for improved learning.

4.1 Training data and sampling strategy

To ensure the latent space captures the actual relevant physical-mathematical concepts, we borrow techniques successfully applied in visual computing, particularly object detection. Here, to achieve good performance the model should not identify objects solely based on their structure – such as shapes, patterns, or colors – or contextual cues (the surrounding environment). The model would rely on spurious correlations and therefore would have difficulties as soon as an object is shown in a slightly different angle or unfamiliar surroundings as it has been trained on. An object detection model, for example, that learned a cow as having a specific shape, is brown and stands on grassland may not recognize the object if it is standing on a beach or is black and white spotted.

Therefore, the model should also learn features of the object and their interplay, which creates a conceptual understanding of the object. This is accomplished by decomposing a training sample into subsamples, usually local sub-regions. Each subsample loses its global embedding, and thus, the model focuses on sub-elements of the whole – like shown in Fig. 4.1 where the concept of a cow is based on characteristic parts of its body.

We apply this paradigm to our vertex objects. Since we can thus draw many different sub-samples from one vertex object, we can use more diverse validation data, i.e., slight variations of training samples for validation, so the model learns to generalize to similar objects.

Furthermore, we designed the subsampling strategy in such a way, that it enables the model to learn concepts, features and topologies of the object instead of its global representation. In object detection it is sufficient to subdivide into locally constrained

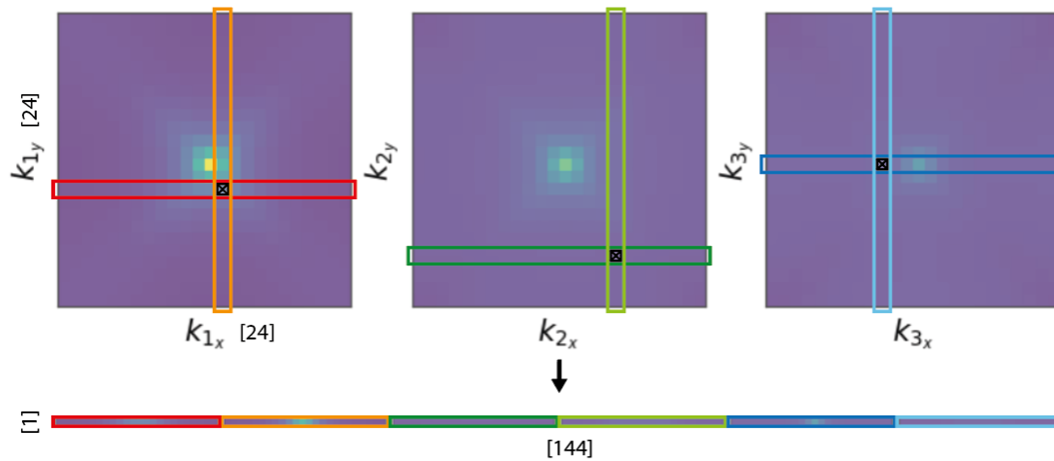


Figure 4.2: Sub-sampling method: Global information is captured by selecting a random point in the vertex (here visualized by three slices through the six-dimensional vertex), selecting a row vector along each of the six axes through this point and concatenating these row vectors.

sub-elements of the object. The vertex function, however, represents quantum-physical particle interactions which are not locally constrained. A subsample, therefore, should capture as much global information as possible. Naturally, this conflicts with the concept of subsampling, so a good balance between small subsamples and capturing global information is required. Furthermore, training and inference should be efficient, which means subsampling has to be performed in a structured way.

The subsampling method used in this work constructs a sample from each axis of the six-dimensional vertex object.

For each subsample a random coordinate within the vertex object is selected. Along each axis of the six-dimensional vertex a full row vector of size 24 is extracted through this point. These six vectors are concatenated to a single vector of length 144 (see Fig. 4.2). This sample captures information from different areas of the vertex - from center to edges, along every dimension of the object - and has a small enough total length for efficient parallelization. Furthermore, it has a defined structure, so that no additional positional information is required.

4.2 Autoencoder

An autoencoder is a type of artificial neural network that reduces the input data into a smaller-dimensional representation, the so-called latent space. This space represents not only a compression of the data, but also distills the most important characteristic features of the data.

The neural network consists of an encoder- and decoder-part. The encoder receives the input sample and reduces its size by transforming it through one or more hidden layers into a latent space representation. For optimization, that is to find a latent space representation that maps the input data optimally, the output is sent to the decoder-part. This upscales the compressed data through one or more hidden layers to the original size of the input sample. The network can then be optimized by measuring the reconstruction error between original input sample and output of the decoder.

Neural networks in general are prone to overfitting, which is prevented by the use of a dedicated validation dataset and evaluation with unseen test data, that covers the full range of parameters and also tests generalizability to parameter ranges unknown to the trained model.

Furthermore, hyperparameter optimization is complex and expensive for neural network training. Therefore, in this work the model architecture configuration itself is not optimized. However, two hyperparameters of special interest in this context have been considered in detail: One is the number of subsamples drawn from each vertex for model training, which tells us, how much of a vertex the model needs to see to be able to reconstruct it. The other parameter is the latent space dimension, which indicates the compression rate with respect to the reconstruction quality.

For the autoencoder in particular it is also necessary to examine whether the latent space has learned the right kind of features. In this case, the latent space needs to be aware of the physical-mathematical features of the vertex function in contrast to structural-topological features of the vertex object in the form of the six-dimensional numerical tensor. We assure this by using an appropriate sampling strategy, as described in the previous section, and a suitable loss function, which is discussed in the following section.

4.2.1 Model architecture

The autoencoder used in this work has six to eight linear layers depending on the latent space dimension, see Fig. 4.3. We compared different latent space dimensions between 32

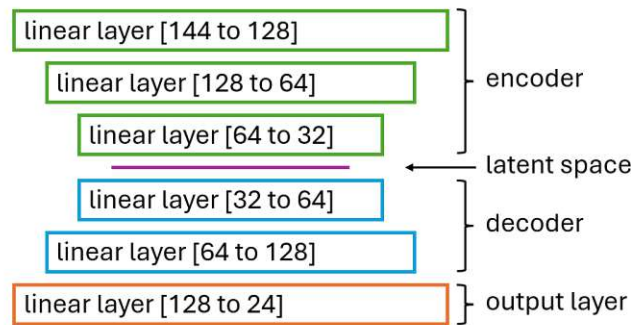


Figure 4.3: Autoencoder with latent space dimension of 32: The encoder compresses the input sample over 3 layers to a smaller representation (latent space). The decoder then reconstructs this to the original representation. The output layer makes sure to only return one of the six axis from the input sample, which is necessary when reconstructing the full vertex.

and 8. With a dimension below 24, an additional encoder- and decoder layer is used for smoother reduction.

The encoder stepwise reduces the input vector over several layers to a length denoted as latent space dimension. The output of the last encoder layer is the latent space. Here, the decoder attaches, which basically reverses the process of the encoder. The decoder is similar in structure to the encoder, but has one layer less (the layer with the largest size). Instead, it is followed by an output layer of size 24, which corresponds to a specific axis of the vertex object. The other five axis-vectors, as part of the input sample, add contextual and positional information. By returning only a single axis, we are able to use a more efficient reconstruction algorithm (see chapter 4.2.3).

Activation and optimizer

The activation of the neural network is the ReLU (rectified linear unit) function, which has a good balance between effectiveness and computational efficiency. The optimizer algorithm is AdamW (Adaptive Moment Estimation with decoupled weight decay regularization [LH19]). This optimizer tries to avoid local minima by using momentum, adapts learning rates per parameter, decouples weight decay from learning rate updating which improves generalization performance, and converges fast by accelerating progress along flat directions.

scenario	train-test-split	loss	trained phases
$S_{[AFM+SC+FM]}$	no splitting	MSE	all
$S_{[AFM+FM]}$	80/20	MSE	w/o SC
$S_{[SC+FM]}$	80/20	MSE	w/o AFM
$S_{[AFM+SC]}$	80/20	MSE	w/o FM
$S_{[SC]}$	80/20	MSE	SC only
$S_{[AFM]}$	80/20	MSE	AFM only
$S_{[FM]}$	80/20	MSE	FM only

Table 4.1: Different model training scenarios.

Loss functions

Every model is trained using the mean squared error (MSE) loss function, measured between the part of the input vector corresponding to the prediction axis (one of the six axes of which the sample is composed of) and the output vector with the function

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (x_i - y_i)^2. \quad (4.1)$$

4.2.2 Training

As shown in table 4.1 the models are trained for different scenarios, using different dataset splits, to evaluate different aspects of the latent space learning.

With scenario $S_{[AFM+SC+FM]}$, trained on all phases, we get a benchmark for the maximum reconstruction quality possible for the autoencoder. For the other scenarios only vertices from certain phases are used in training. This enables us to evaluate the model by reconstructing a vertex from an entirely unknown phase. Here we use 80% of the vertices from the selected phases for training, the rest of the vertices for model testing. How well concepts learned from one phase can be transferred to another unseen phase indicates whether the model indeed has captured an understanding of the underlying physical concepts of the vertex function.

For each model training the optimizer uses a learning rate of 10^{-4} and a weight decay of 10^{-5} . As described before, the models output a vector of only a single axis of the

latent space dimension	layer in-/output dimensions
8	144 → 128 → 64 → 32 → 8 → 32 → 64 → 128 → 24
16	144 → 128 → 64 → 32 → 16 → 32 → 64 → 128 → 24
20	144 → 128 → 64 → 32 → 20 → 32 → 64 → 128 → 24
24	144 → 128 → 64 → 24 → 64 → 128 → 24
32	144 → 128 → 64 → 32 → 64 → 128 → 24

Table 4.2: Neural network layer sizes depending on the latent space dimension

vertex object. For every model the third axis was chosen to be returned. Models are trained using a batch size of 8192 for 25,000 epochs. The time for training depends on the number of vertices and subsamples per vertex used. Using all data training takes around 41 hours. Training was executed on two NVIDIA A100 GPUs.

From each training vertex a certain number of subsamples were drawn. The number of samples per vertex is a hyperparameter. We compared the results for values between 2,000 and 64,000.

Also different latent space sizes between 8 and 32 are compared. The neural network layer sizes depending on the latent dimension are shown in table 4.2.

20% of the training data is used for validation, while the remaining 80% are used for training. Validation is performed after each training step, which is the processing of one batch of input samples. After each validation step a model checkpoint is saved to disk if the validation result is the best achieved so far. The checkpoint can be used to continue training or perform inference.

Model validation

The output layer of the autoencoder should return the reconstructed values for a single axis, which corresponds to a specific subsection of the input vector. Therefore, it has to be ruled out that the model just directly extracts this section from the input and returns it. If this were the case, the model would not learn a proper latent space representation of the vertex-sample. To check this, an algorithm is applied, where every entry in the input vector – except the section that is to be predicted – is replaced by a random value. If the model simply extracted the output, the other values in the input vector would not influence the result. It showed, however, that the output differs vastly from

the corresponding input vector section for all input samples and therefore the model is computing the output vector from all input values, as it is intended.

4.2.3 Evaluation

In order to decompose a vertex object into sufficient subsamples to reconstruct the complete vertex, every coordinate in five of the six dimensions is selected. The sixth dimension is the reconstruction axis the model returns as output. For each coordinate a random point is taken along the remaining axis. Through these selected points the subsamples, as described in section 4.1, are created. To illustrate, in three-dimensional space we would iterate through all points in the xy-plane of a cube and select a random value along the z-axis for each point. Model prediction is then performed on each of those samples.

The output of the model is the row vector for the fixed axis. The output vectors are arranged to form a reconstruction of the entire input vertex. In the three-dimensional space, we would predict the z-axis vectors for each point in the xy-plane. Stacking them together on the xy-plane would then recreate the entire cube.

We evaluate the model using two distinct test sets. First, approximately 20% of the vertices from the phases selected for the respective model training scenario were split off to evaluate the reconstruction performance. Second, the vertices from phases not present in the training data were used to evaluate the generalization performance of the model – the ability to reconstruct vertices from unseen phases.

The quality of the reconstruction is measured using root mean squared error (RMSE) between original vertex and reconstruction. The RMSE given as

$$\mathcal{L}_{\text{RMSE}} = \sqrt{\frac{1}{n} \sum_{i=1}^n (V_i - \hat{V}_i)^2} \quad (4.2)$$

measures the deviation of the reconstructed vertex tensor \hat{V} to the original V . The final error score is the mean of the RMSEs of all test vertices.

CHAPTER 5

Results

In this chapter the findings from the autoencoder are discussed. Besides comparing all training scenarios, for every analysis we also look at different latent space sizes, numbers of subsamples per vertex and training data subsets for a selected scenario.

First we look at the training convergence, to confirm training was sufficient and successful. We then discuss the reconstruction results for unseen vertices. We reconstructed vertices from phases seen during training to assess the ability to retrieve the full vertex information from the latent space. Subsequently, we reconstructed vertices from unseen phases to assess the ability to generalize, which hints to an understanding of the underlying physics through the latent space. We then compare our results to the first approach using neural networks by Zang et al. [ZMK⁺24]. We also look at the correlation matrix of the latent space, to see if the latent space successfully reinforced physically meaningful features.

This is followed by a section where we analyse the latent space in detail. For this we utilize the latent space representation of vertices in a classifier that determines the vertex phase. We identify suitable classifier algorithms and present the training and evaluation process. Based on the classification results, we discuss how well the latent space captures the features of vertices. We further analyse how vertex features are represented in the latent space by training an alternative encoder model using contrastive learning. Finally, we investigate the potential to predict a vertex for the next higher t' -parameter given any other vertex as input.

run	wall time [hours]	min. loss	epoch of min. loss	epoch of min. loss (10% threshold)	max. epochs w/o decrease
$S_{[AFM+SC+FM]}$	41	0.000072	22,574	7,957	22,574
$S_{[AFM+FM]}$	24	0.000379	24,019	2,832	17,170
$S_{[SC+FM]}$	20	0.000072	19,716	6,325	18,949
$S_{[AFM+SC]}$	29	0.000078	23,323	4,295	16,690
$S_{[SC]}$	8	0.000077	24,280	1,467	23,421
$S_{[AFM]}$	13	0.000337	24,221	1,000	22,469
$S_{[FM]}$	11	0.000091	7,115	1,164	7,115

Table 5.1: Statistics for the training of each scenario (with latent space dimension of 32 and subsamples per vertex of 24,000).

5.1 Autoencoder training convergence

For most models the loss decreases quickly, but new minima can still be found after many – even thousands of – training epochs, making it difficult to identify complete model convergence. Nevertheless, no significant overtraining can be seen either. Experiments were first done with early stopping with a patience of up to 100 epochs but due to the very slow convergence this stops training far too early.

Fig. 5.1 shows the progress of the validation loss for different scenarios, latent space dimensions, and subsamples per vertex. In general, it can be said that models converge faster the more training data they use (in respect to both number of vertices and number of subsamples per vertex) or the larger the latent space.

Table 5.1 shows some statistics for different training scenarios: The training time for all epochs, the total number of epochs, and the lowest loss achieved. Furthermore, it indicates the first epoch, where the loss is within a 10%-margin of the minimum loss. Finally, the table shows the longest training segment (in number of epochs) without any decrease in the loss, which gives an idea about the early stopping patience required for full training.

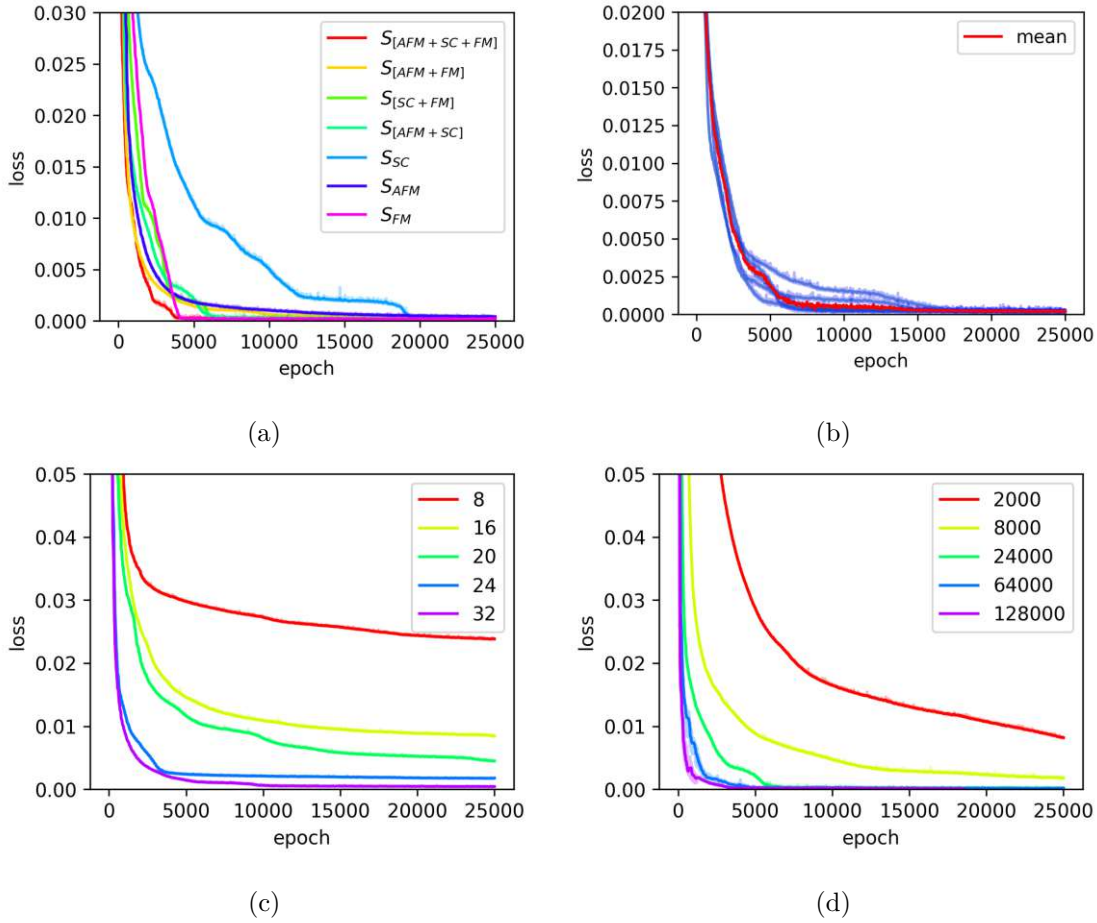


Figure 5.1: Progress of validation loss for each scenario (a), for scenario $S_{[AFM+FM]}$ with different latent space dimensions (c), for scenario $S_{[AFM+SC]}$ with training data subsets (b), and different numbers of subsamples per vertex (d).

5.2 Reconstructing vertices from latent space representations

General reconstruction performance

In the first part of the model performance analysis, we evaluate the reconstruction performance through the RMSE. The data used to run inference on consists of vertices unseen during training but from phases known to the model. This avoids any bias from leaking training data or confusion from unknown phases. In Fig. 5.2a we see the RMSEs as boxplots for all scenarios, where $S_{[AFM+SC+FM]}$ is our benchmark.

The orange line in the boxplot indicates the mean of the RMSEs for all vertices. The extent of the rectangle covers the inter-quartile range, that is the range where 50% of all RMSEs are located; 25% of the values are each higher and lower. The lines attached to both ends of the rectangle, the whiskers, extend the inter-quartile range by 1.5; they usually cover only a few percent of the data. The dots further out are considered outliers.

Looking at scenarios trained only on some phases, the errors are significantly different depending on the phases included in training. We look at some scenarios in detail later in this section.

As expected, increasing the latent space dimension (5.2b) or the number of subsamples per vertex (5.2d) improves the reconstruction quality. We also see that the margin of the reconstruction errors (one result for each vertex) becomes smaller resulting in a more stable reconstruction.

In Fig. 5.2c we see the RMSEs for scenario $S_{[AFM+SC]}$ using different data subsets used for training. The error margins differ a lot, while the mean error differs roughly between 0.008 and 0.01. No data subsets leaves out larger numbers of subsequent vertices, which would explain a deteriorated performance in some t' subdomains. Fig 5.3 shows that all subsets behave similarly over the whole t' domain. Therefore, we assume that the difference does not come from the vertices selected for training but from the choice of subsamples drawn from a vertex. With 24,000 subsamples per vertex we only sample around 1.9% of the data contained in a vertex. Since characteristic features are clustered in subregions, leaving large subregions of very low variation, it is easily possible that the drawn subsamples miss any subregions containing important information.

In general, it is impressive that we can reconstruct a vertex with such a small reconstruction error of 0.008 (that is only around 0.2% of the mean of the vertex values) with a model that is trained on only ca. 1.9% of the data in a vertex (24,000 subsamples with 144 values each drawn from a vertex with 24^6 entries). This is one of the proofs that a vertex function can be strongly reduced: the overwhelming amount of vertex information can be represented by a small subset of the data.

Generalization to unseen phases

In the second part, we evaluate the models ability to generalize to unseen phases. We compare models trained on vertices from two phases and a single phase only. Fig. 5.4 shows the mean RMSEs for reconstructing vertices from phases the model has not seen during training, which each scenario being trained on a different set of phases (see 4.1).

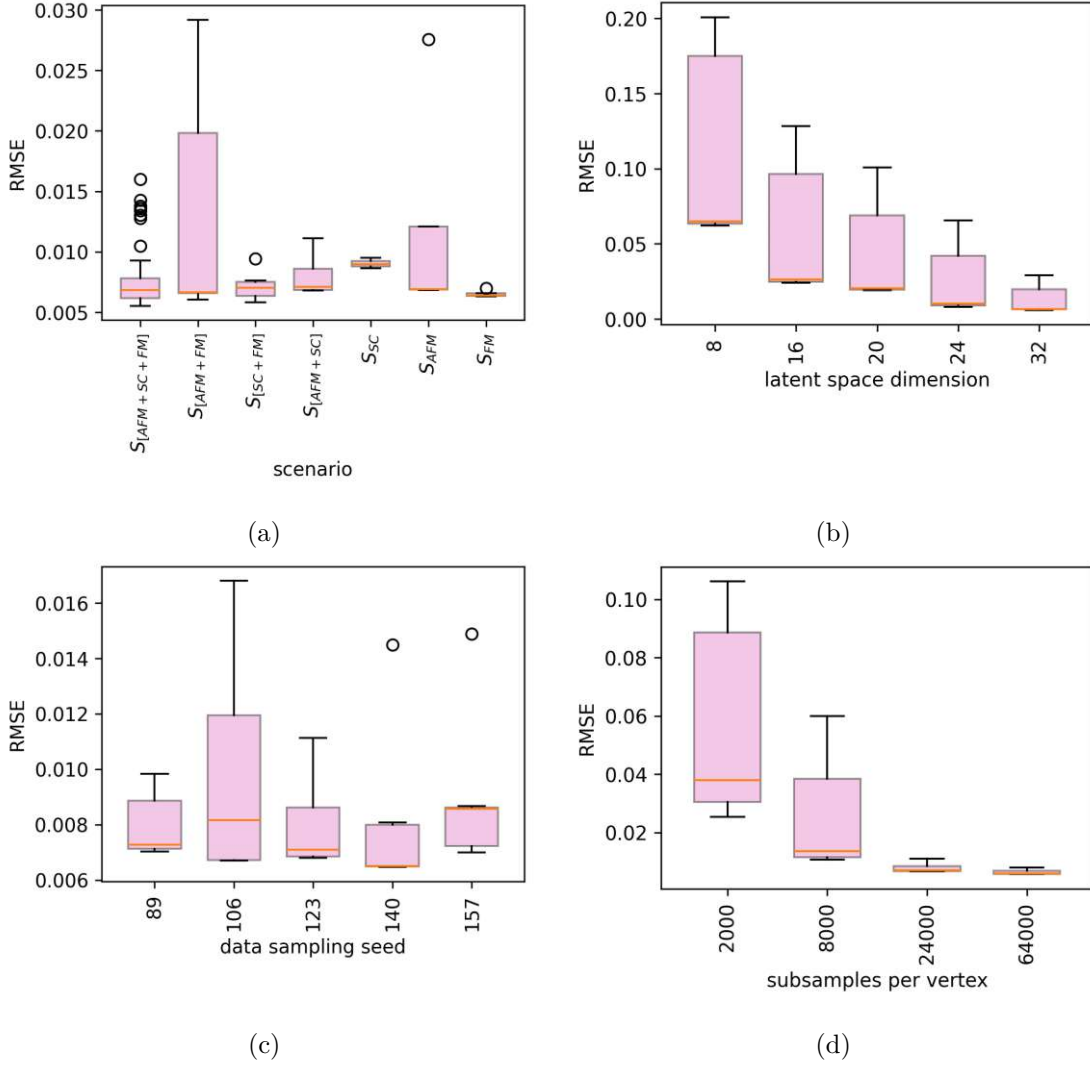


Figure 5.2: Reconstruction errors (RMSE) for vertices from trained phases for each scenario (latent space size 32 and 24000 subsamples per vertex) (a), for scenario $S_{[AFM+FM]}$ with different latent space dimensions (b) and for scenario $S_{[AFM+SC]}$ with different training data subsets (c), and numbers of subsamples per vertex (d).

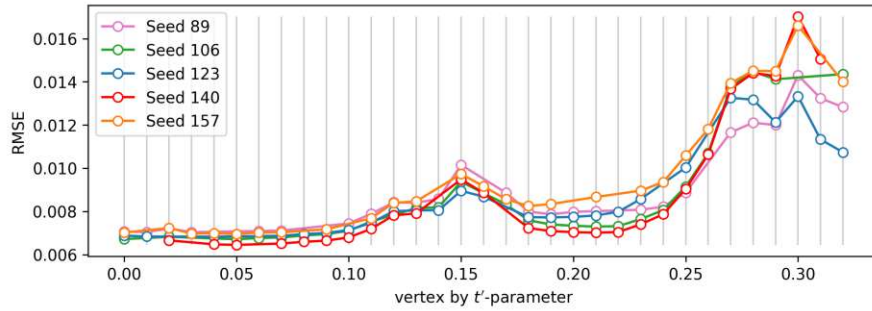


Figure 5.3: RMSE per vertex for different training data subsets for scenario $S_{[AFM+SC]}$.

We see that the performance depends much more on which phases the model has seen, than on the number of phases used for training. In general, however, all graphs show a very good generalization ability to new phases.

Reconstruction performance per vertex

In the third part we look at some scenarios in detail. Each panel in Fig. 5.5 shows the reconstruction error for each vertex (identified by the value of the parameter t'), with blue dots representing vertices used for training and pink ones for unseen vertices. The horizontal lines show the overall mean over all vertices, and the means over vertices seen during training and unseen vertices. This helps us distinguish how the model performs for unseen vertices versus known vertices. It shows that the decisive factor is not whether a vertex has been used for training, but whether a phase was present during training. The model is able to reconstruct vertices very well, as long as it was trained on vertices from the respective phase. We note that there is no significant difference between seen and unseen vertices within each phase.

The benchmark scenario $S_{[AFM+SC+FM]}$, trained on all data, shows that the SC-phase (in the middle) is the most difficult to model. Some reasons can be drawn from the raw data: The SC-phase contains the least vertices, but this difference is only minor. We have seen in the correlation analysis (see Fig. 2.5) that the SC and FM phases have the strongest differentiation. The same behavior can be seen in the error rates. It seems the model can easily adapt to the FM-phase, but has the biggest difficulties at the SC-FM transition.

It is also noticeable that the error rates for test vertices (not used in training) from training phases are about the same as for neighboring training vertices. So the models can reconstruct a similar vertex roughly equally well, no matter if the model has seen it

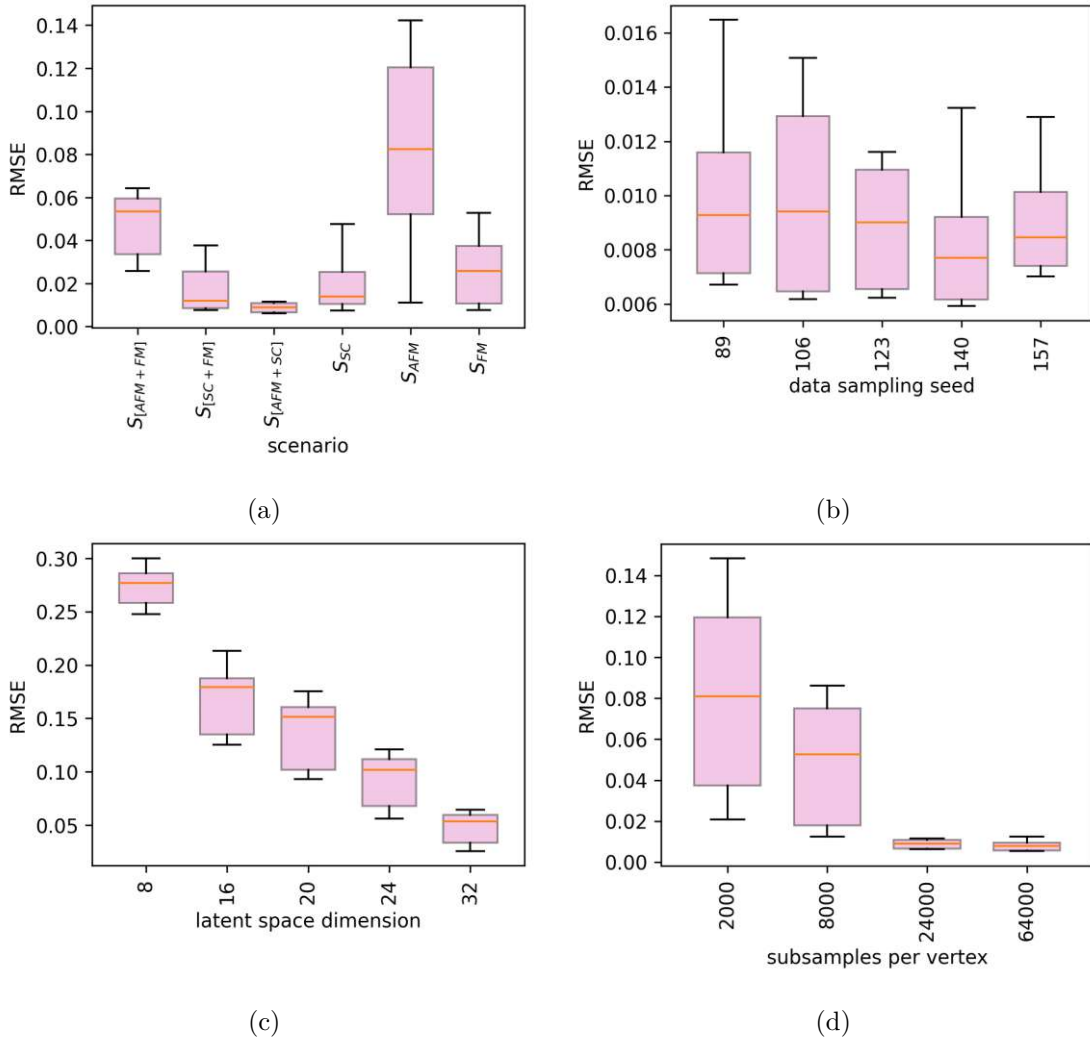


Figure 5.4: Reconstruction errors (RMSE) for vertices from unseen phases for each scenario (latent space size 32 and 24000 subsamples per vertex) (a), for scenario $S_{[AFM+FM]}$ with different latent space dimensions (c) and for scenario $S_{[AFM+SC]}$ with different training data subsets (b) and numbers of subsamples per vertex (d).

during training or not, as long as it is from the same phase. This indicates an extremely well generalization within known phases and shows how similar the vertex data is within the same phase.

Training on combined SC- and FM-phase achieves a good generalization ability over all phases (scenario $S_{[AFM+FM]}$). Also with the SC-phase alone (scenario $S_{[SC]}$) a low mean error is achieved, maybe because this phase is in the middle and – since transitions are smooth – contains information from the two adjacent phases. By contrast, training only on the AFM-phase (scenario $S_{[AFM]}$) results in the worst performance. The AFM-phase is physically more similar to the SC- than to the FM-phase. This is also seen in the correlation matrix. Therefore, a possible reason may be that on one hand the AFM-phase contains little information about the FM-phase and on the other hand it is too similar to the SC phase, so that the model rather reconstructs an AFM-vertex instead of a SC-vertex.

In Fig. 5.6 we see the reconstruction errors for scenario $S_{[AFM+FM]}$ trained with different latent space dimensions. The shape and the peaks of the curves are very similar, but the margin of errors varies considerably. The latent space dimension influences the reconstruction error for every vertex in a similar extend.

In Fig. 5.7 we compare results using different numbers of subsamples per vertex for training with scenario $S_{[AFM+SC]}$. Again, the shape is roughly the same. Interestingly, the SC phase seems to be the most difficult for this scenario-model, even though SC is part of the training.

Finally, using different dataset subsets for training creates very similar distributions of reconstruction errors among different vertices. This shows that the distribution of important regions in the vertex space is similar for every vertex.

Distribution of reconstruction errors in a vertex

Finally, we look at the reconstructions of some vertices. In Fig. 2.4 we have seen that the values in a vertex are very imbalanced. The majority of the values are small, but the distinguishing features of the vertices are determined by large values. If the autoencoder returned mostly small values, the mean error might still be rather small since the large values do not have a large enough weight in the vertex, but the characteristic features of the vertex would be missing. So we need to confirm that we have acceptable errors in all value subranges. In Fig. 5.8 we report the values for two selected vertices each from three different scenarios (for each scenario the vertices with lowest and highest reconstruction

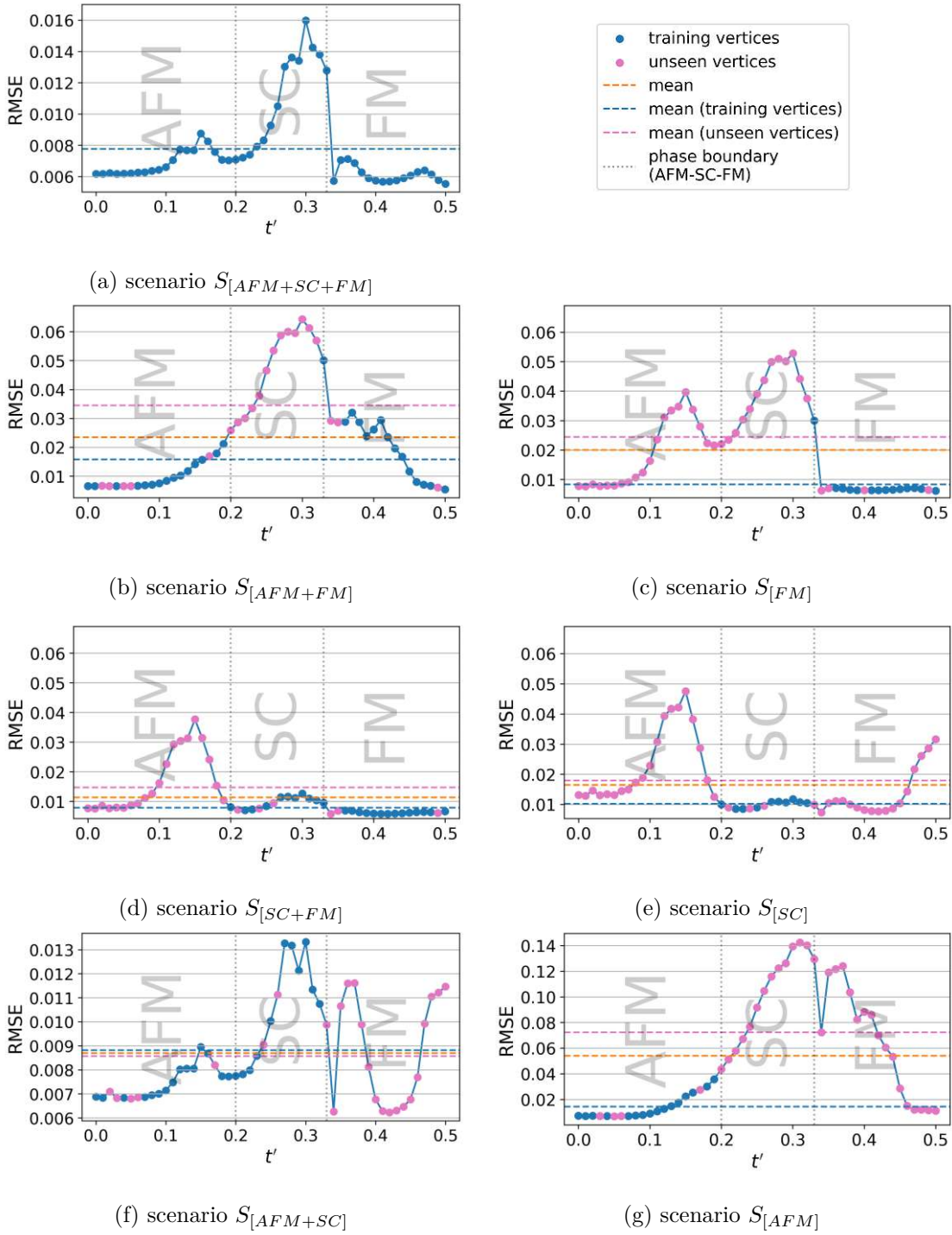


Figure 5.5: Reconstruction errors (RMSE) for every vertex (identified by the value of t') for different scenario models (each with latent space dimension 32 and 24,000 subsamples per vertex).

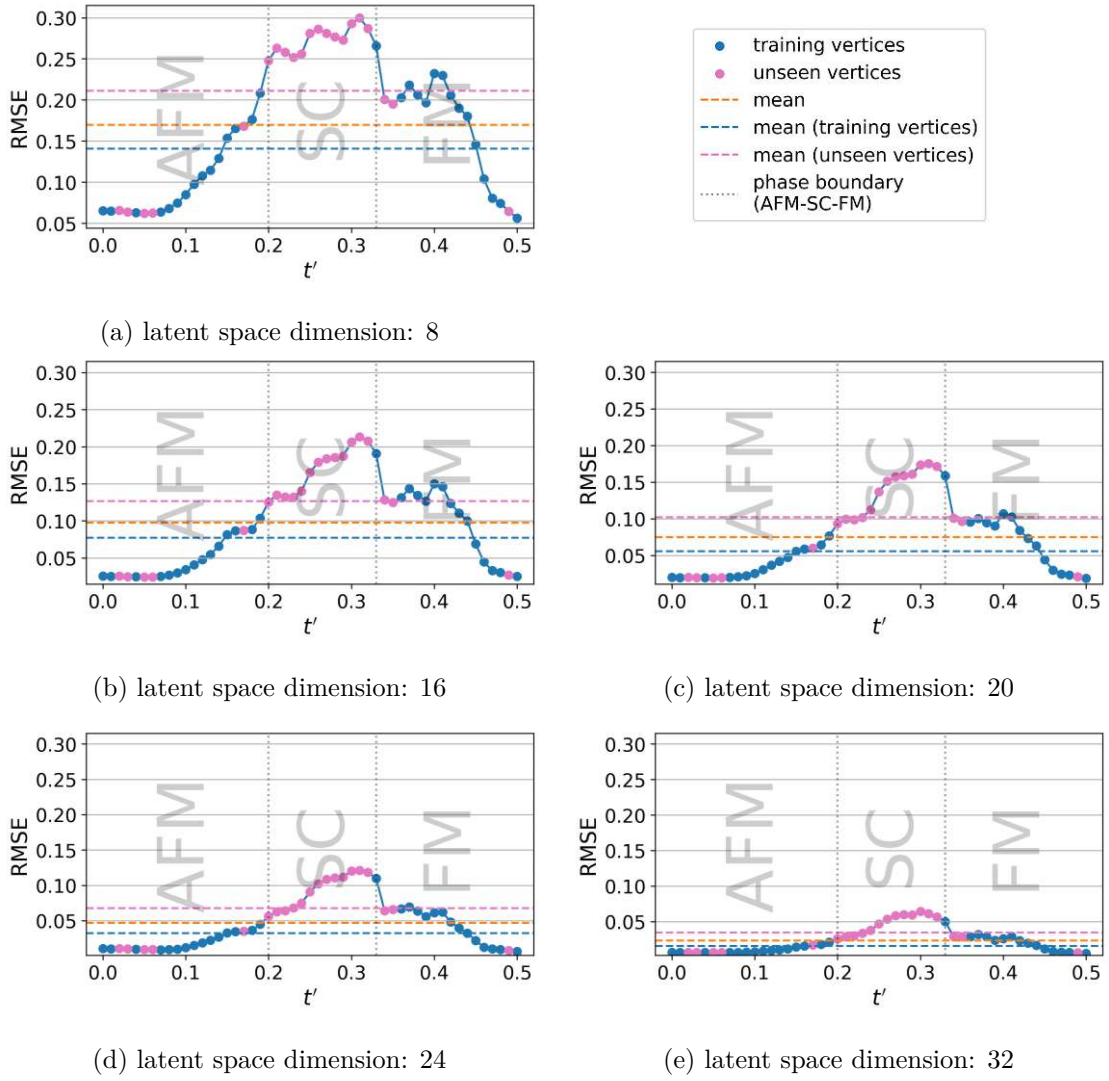


Figure 5.6: Reconstruction errors (RMSE) for every vertex (identified by the value of t') for scenario $S_{[AFM+FM]}$ for different latent space dimensions (each with 24,000 subsamples per vertex). Note the different y -axis scales.

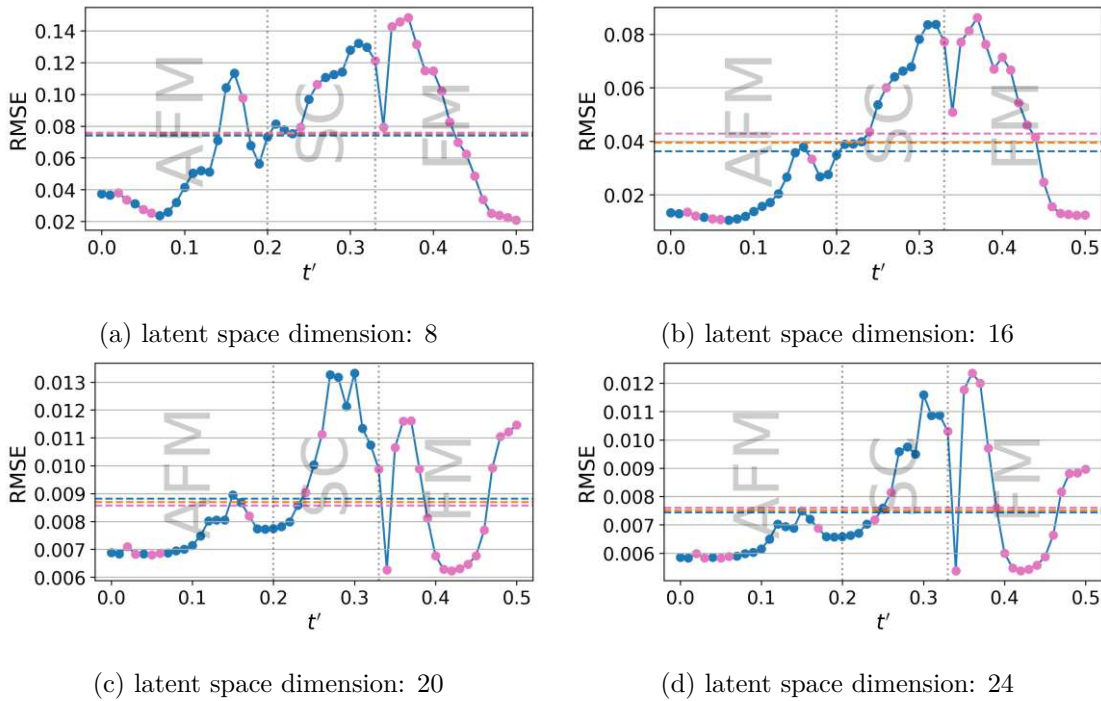


Figure 5.7: Reconstruction errors (RMSE) for every vertex (identified by the value of t') for scenario $S_{[AFM+SC]}$ for different numbers of subsamples per vertex (each with a latent space dimension of 32).

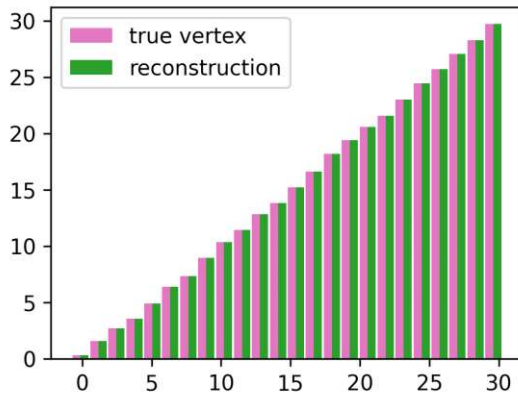
error are shown) and their reconstructions. As expected, the smaller the values, the more confidently they are reconstructed. The reconstruction error is mostly distributed among the extreme values on both ends, and especially among the negative values, which occur considerably less often in the vertices than large positive values.

5.2.1 Comparison to convolutional neural network approach

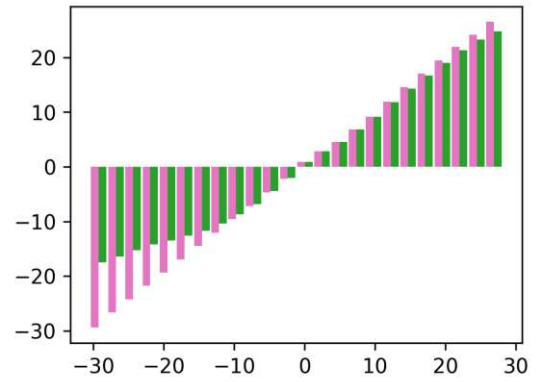
Zang et al. [ZMK⁺24] implemented a classical autoencoder using convolutional layers instead of linear layers. They use the same vertex data but reshaped from six-dimensions to a three-dimensional tensor with axis length 576 and then downsampled to a size of 144^3 . Each input sample is an entire vertex tensor. The downsampling is necessary to assure feasible computation.

The convolutional layer applies an aggregation algorithm on a three-dimensional subregion of the input and returns it with a smaller axis length. This algorithm is applied on every subregion. The layer then returns an aggregated version of the input with a smaller axis

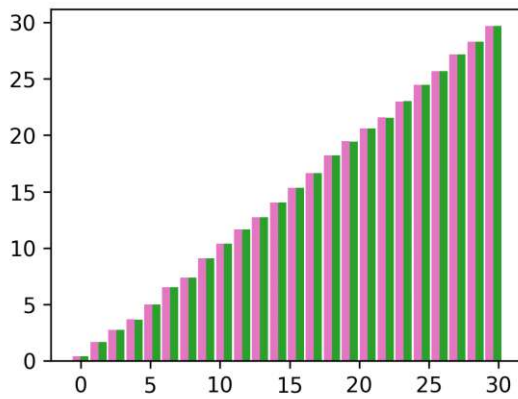
5. RESULTS



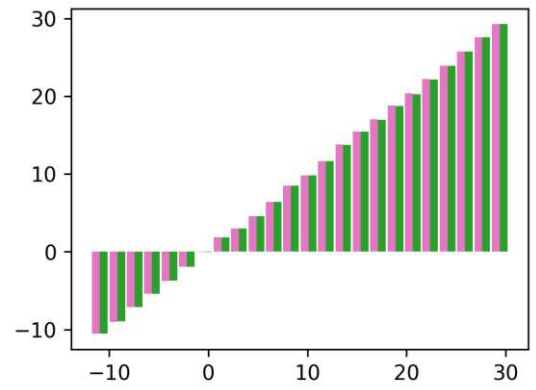
(a) S_{AFM+FM} , $t' = 0.03$, $RMSE = 0.0066$



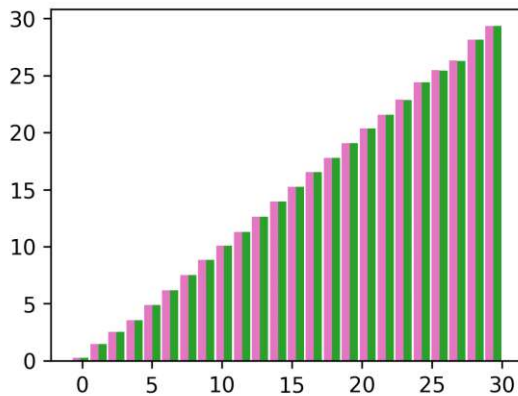
(b) S_{AFM+FM} , $t' = 0.30$, $RMSE = 0.0643$



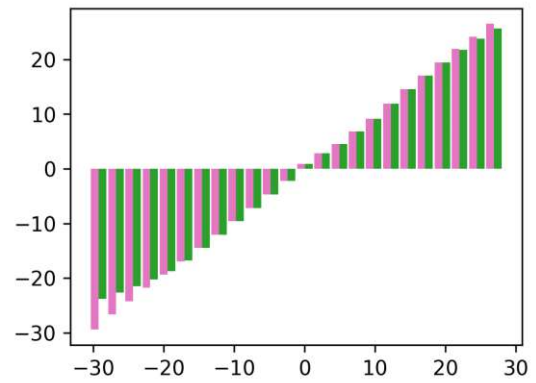
(c) S_{SC+FM} , $t' = 0.01$, $RMSE = 0.0076$



(d) S_{SC+FM} , $t' = 0.15$, $RMSE = 0.0377$



(e) S_{AFM+SC} , $t' = 0.42$, $RMSE = 0.0062$



(f) S_{AFM+SC} , $t' = 0.30$, $RMSE = 0.0133$

Figure 5.8: Means of true and reconstructed values for each subrange of the input vertices value domain for selected vertices.

length. Layers are stacked until the desired output dimension is achieved.

However, the authors did not observe satisfying results. The reason is likely due to convolutions being applied on locally constrained subregions. Therefore, the model can only learn local features. However, the vertex function, representing quantum-physical particle interactions, is highly non-local. Based on this finding, we developed our subsampling strategy which is able to capture global features.

5.2.2 Correlation

Figure 5.9 shows the correlation between each pair of vertices for the real vertices on the left and for vertices encoded with an autoencoder (with latent space dimension of 32) on the right. Both show the same qualitative pattern and high correlations. However, the minimum correlation for the encoded vertices is significantly higher (0.86 compared to 0.59 for the real vertices) which indicates that an autoencoder manages to discard less important information and concentrates the correlation, which exists naturally between the vertices. Both plots also fuzzily show an increased correlation between vertices of the same phase, however this distinction could not be amplified by the autoencoder.

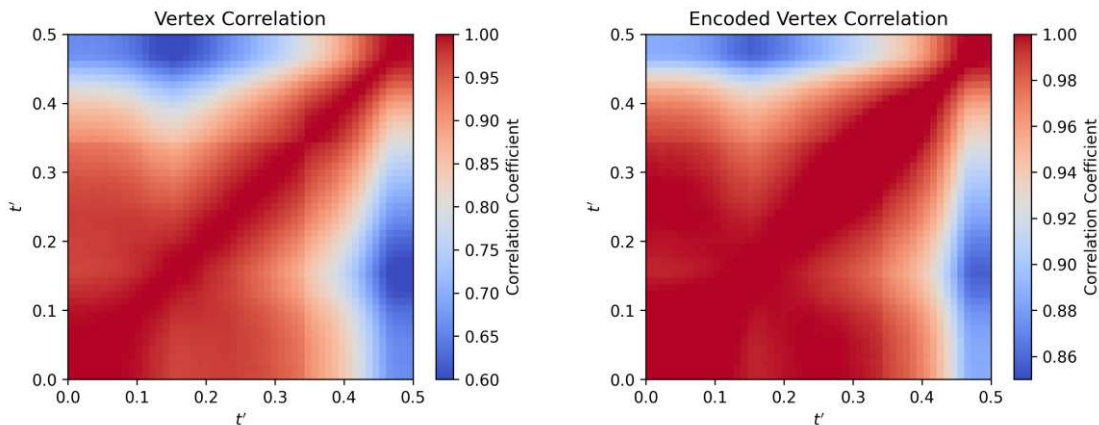


Figure 5.9: Correlation between vertices with t' between 0 and 0.5 for real and encoded vertices using an autoencoder with a latent space dimension of 32. Note the different start values of the color-scales.

5.3 Analysis of latent space

The latent space representations of a vertex obtained by the model training contains the essential reduced quantum-physical information of the vertex function, which we hope

can help to interpret it and, for example, let us infer properties more confidently. Even if the latent space cannot be directly interpreted, we can evaluate whether the model has learned physically meaningful features of the vertex function by applying the latent space representation in usecases.

In this section we analyse the latent space using one such usecase: We classify the phase of a vertex and investigate if a latent space representation improves the classification accuracy. For this we train classifier models and apply them on original vertex objects and their latent space representations. An improved accuracy would then indicate that the latent space can be interpreted with more confidence than the original vertex function.

The phase of a vertex is identified by the t' -parameter as shown in Fig. 2.2. For our data the phase is one of AFM, SC or FM. We evaluate the encoder models for every training scenario to investigate different aspects of the latent space and the vertex function. We also compare different classifier model architectures.

Initially, from the many available model types for classification, suitable ones are identified and introduced in detail. Then the training and evaluation process is described. Finally we present and interpret the results.

5.3.1 Phase classifier model architecture

Several of the most effective model techniques for classification tasks have been considered. The desired features of the model are good classification performance, fast training, easy hyperparameter tuning and interpretability.

The techniques are compared on scenario $S_{[AFM+FM]}$. The random forest classifier was selected for all other scenarios, as it is fast, achieves very good classification accuracy, and is interpretable.

Random forest

A random forest (RF) is a collection of decision trees. A decision tree iteratively splits the set of training data into subsets, until a stopping criteria is fulfilled. At each iteration a data attribute is chosen. The value that splits the dataset is found by maximizing the purity of the resulting subsets using a measurement function. Stopping criteria may be that a given number of splits (which is the depth of the decision tree) or subset purity is reached. Each decision tree of the RF is trained on a different random subset of the data. The final result is obtained by aggregating the results from all decision trees, e.g.

by majority vote. By relying on many decision trees, the random forest mitigates the problem that a single decision tree often overfits on the data. The stopping criteria, process of attribute selection, measurement function, acceptable impurity of the subsets, and number of decision trees are hyperparameters. Decision trees allow to obtain the feature importance, which gives insight into which data attributes are most determining for the resulting decision. Furthermore, random forest are rather fast in training and can be parallelized easily.

Our RF trains 100 decision trees. The quality of a split is determined using Gini impurity, which measures how often an item in a node would be labeled incorrectly, if it were labeled randomly according to the distribution of the labels in the node. For each node in the tree, it is defined as

$$I_G = \sum_i p_i(1 - p_i), \quad (5.1)$$

summing the impurity for each label i using each label's probability p_i in the node. Nodes are split until they are pure.

Gaussian process

The Gaussian process (GP) technique interprets the data as random variables defined by multivariate normal distributions. The data is modeled by fitting a joint distribution through those normal distributions. This model is able to model complex data and enables to obtain probabilities and confidence intervals. Our Gaussian process model uses a radial basis function (RBF) kernel.

For computation GP creates a matrix, where the size depends on the number of features (here the length of the subsample vectors). In our case the matrix becomes extremely big, which makes computation impossible with regular hardware. GP was therefore excluded.

Support vector machines

Support vector machines (SVM) classify data by fitting a hyperplane through the data space such that the purity of the classes and the margin separating the data is maximized. SVMs usually achieve a good classification performance. The training, however, is very slow. Since all data is needed for hyperplane fitting at the same time, the full dataset needs to be held in the memory. The choice of the kernel function, responsible for calculating the hyperplane, and the strength of regularization are hyperparameters. We use an SVM with an RBF kernel.

Neural network

A simple neural network (NN) classifier is also used. It consists of three linear layers with incrementally smaller size and returns the probability for each class. The model uses an AdamW optimizer, cross entropy loss, a learning rate of 0.001, a weight decay of 10^{-5} , and a batch size of 4096. It is trained on GPU for up to 1000 epochs with early stopping and a patience of 10 epochs. Neural networks are memory efficient and easily parallelizable on GPU but still rather slow, since they perform a high amount of calculations and need a lot of data to achieve good results.

Random forest regression

We also performed classification using regression models. This is possible because a phase corresponds just to a specific subrange of t' . Therefore, we regress on t' and then classify the t' -value obtained.

Since we apply the random forest classifier, we also apply a regressor version. Here, the results of the single decision trees are averaged for the final return value. The configuration is the same as for the classifier except for using the squared-error function to measure the split quality.

Polynomial regression

Polynomial regression is a classic regression technique that uses polynomial terms in the regression function to model non-linear relationships. There are two different implementation techniques: We can either explicitly compute the polynomial combinations of the features or we compute a spline basis functions for each feature. The explicit polynomial approach can result in a very large matrix, depending on the degree of the polynomial. This not only requires a lot of memory but may also hamper fitting the regressor. The spline approach is computationally much more feasible, since the non-linear relations are modeled by parametrized functions.

In our experiments the explicit polynomial approach failed, since it resulted in a polynomial-matrix too large to hold in memory and to fit. The spline approach was executed successfully with a degree of three. Two versions were trained with four and ten knots for the spline functions. When using more knots, also this approach failed due to its large memory demand.

		predicted phase	
		positive	negative
actual phase	positive	true positive (TP)	false negative (FN)
	negative	false positive (FP)	true negative (TN)

Table 5.2: Structure of a confusion matrix for one class.

5.3.2 Training

For each encoder model a phase classifier model is trained. Additionally, a classifier is trained on the original real-space vertices.

To train the classifiers, 24,000 random subsamples are drawn from each vertex and encoded to their latent space representation by the encoder model. The output is used to train the classifier. For scenario $S_{[AFM+SC]}$ results have been compared for classifiers trained on numbers of subsamples per vertex between 2,000 and 64,000.

5.3.3 Evaluation

From each vertex 4,800 random subsamples are drawn and classified by the trained model. To compare predicted and true labels confusion matrix and f1-score are computed.

Table 5.2 shows a confusion matrix for one class. The x -axis denotes the correct class. The columns show the number of items having this class and the number of remaining items. The y -axis denotes the class predicted by the classifier. The rows show the number of predictions for this class and the number of other predictions. For a perfect result, the confusion matrix would show ones on the diagonal and zeros otherwise. The confusion matrices in section 5.3.4 show the number of subsamples row-wise normalized per actual class for each combination of actual and predicted class.

The f1-score rates the classification accuracy from 0 to 1 and is defined as

$$f_1 = 2 \cdot \frac{PRE \cdot REC}{PRE + REC} \quad (5.2)$$

with

$$PRE = \frac{TP}{TP + FP} \quad (5.3)$$

and

$$REC = \frac{TP}{TP + FN}. \quad (5.4)$$

It combines the precision and recall score and therefore equally considers false positives and false negatives. In our multiclass case TP , FP , and FN are counted for every class and then respectively summed.

For regression the RMSE is calculated between the t' -values of the vertices from which the subsamples are taken and the predicted t' -values. Furthermore, the phase is derived from the predicted t' -value such that we obtain a phase classification, same as for the classifier-models.

The classifier trained on the original real-space vertices serves as a benchmark, such that we can determine if phase inference is improved by latent space encoding. We use the scenario $S_{[AFM+SC+FM]}$ encoder models to evaluate the achievable accuracy of phase classification on a latent space trained with all vertices. With the other encoder models we evaluate how well the underlying physics concepts are captured by the latent space, so that it can generalize those to unseen vertices. Furthermore, we study the impact of the number of subsamples and the latent space dimension of the encoder models on the classifier accuracy. We also compare encoders using simple and contrastive loss, to see if we can enhance the expressivity of the latent space towards specific applications.

5.3.4 Results and interpretation

We compared the results for each suitable classifier type on scenario $S_{[AFM+FM]}$ in Fig. 5.10a. Random forest achieves by far the best scores, the regressor a little worse than the classifier. This is expected: With regression it becomes more difficult to separate phases close to the phase boundaries. The neural network and SVM are in the middle, but have a lot of potential for hyperparameter optimization. The spline-based polynomial regressors achieved very low scores. However, performance increases steeply with more knots used for the spline-functions. So in theory, a better performance may be possible, however, is not possible with the employed algorithm due to computational infeasibility.

Figure 5.11 shows the true t' of the input vertex versus the predicted t' for our three regressor models (spline-base polynomial regressor with four and ten knots for the spline-function and random forest regressor). The dashed red lines show a theoretical perfect prediction, where every t' is predicted on point. The strong lines show the mean prediction for every true t' (which is in discrete steps of 0.01). The soft-colored area shows the full range of predictions. We see that the polynomial regressions failed completely. Only for the highest t' -values does the model show some alignment to the true values, while for the

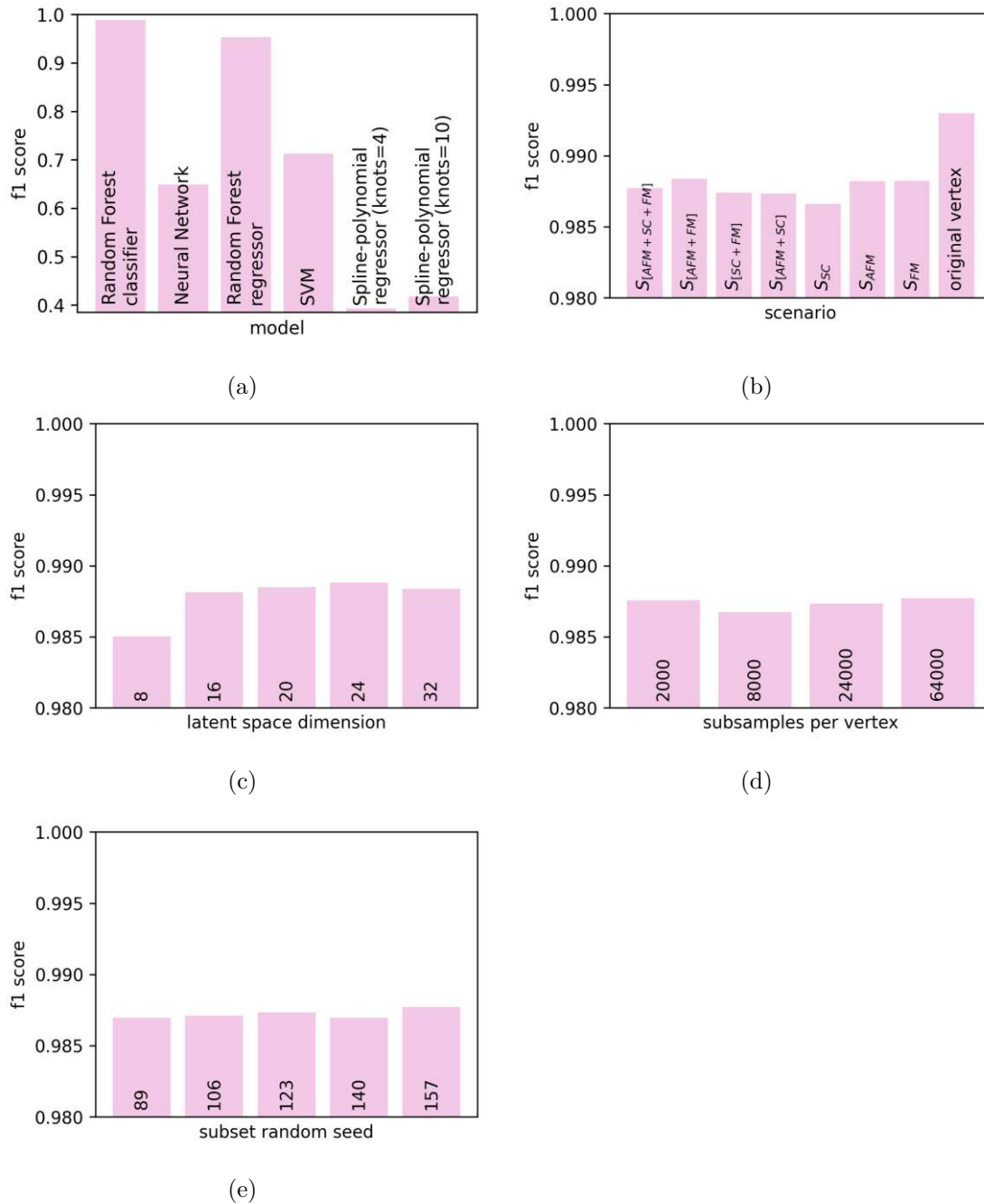


Figure 5.10: F1-score for classifying the phase of vertices. The first panel shows the results for latent spaces produced by the encoder model for scenario $S_{[AFM+SC]}$ of different classifier types (a). All other panels show results for the Random forest classifier: (b) shows results for all scenarios, (c) for scenario $S_{[AFM+FM]}$ for different latent space dimensions, and (d) numbers of subsamples per vertex; (e) shows results for scenario $S_{[AFM+SC]}$ using different training data subsets.

rest the prediction is simply near the average t' of 0.25. Furthermore, some predictions are much higher than the maximum true t' of 0.5.

On the other hand, there is the random forest regressor in panel c). In addition, the more detailed plot shows for each true t' the distribution of predictions as boxplots. The black dotted lines indicate the phase borders between the AFM, SC, and FM phase. The mean prediction for each t' (indicated by the orange horizontal line in each boxplot) is quite close to the red diagonal line. Since in the end we use the predicted t' to infer the phase, we are mainly interested if the predictions are located in the diagonal segments between the phase boundary lines. Looking at the boxes of the boxplots, we see that for most t' the majority of the predictions are in the correct segments and thus would be classified as the correct phase. Furthermore, we see that the model on average is especially good in predicting the SC phase, while in the FM-segment the offsets from the diagonal are largest. To evaluate the stability and confidence of the predictions, we can look at the full extent of the boxplots and see that in the AFM-segment predictions are most trustworthy, while for the other phases there are considerable amounts of misclassifications. The random forest classifier was used for all other evaluations, since it achieves good scores, and furthermore is fast to train and can be interpreted.

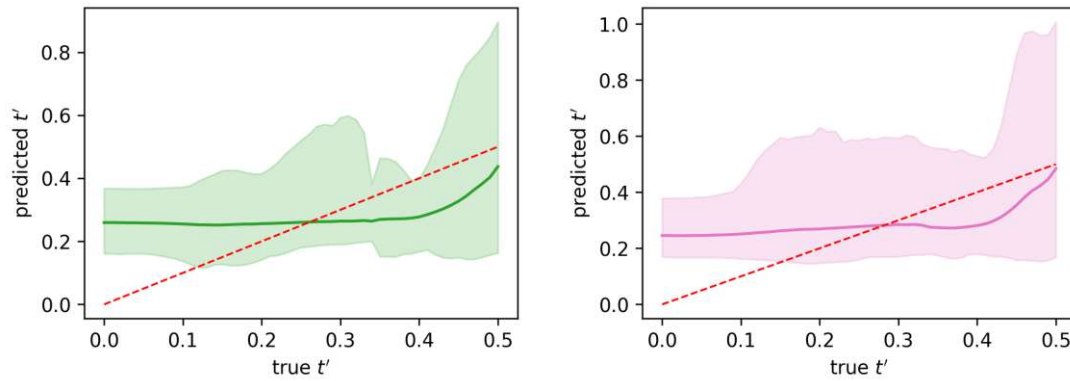
In Fig. 5.10 we see that for all scenarios a high f1-score was achieved around 0.9875, with only minor differences between the scenarios. However, phase classification directly on the original and not encoded vertices achieved even higher results.

Comparing the results for encoder models trained with different data subsets or numbers of subsamples per vertex achieve minimal differences, explainable by the natural variations of results from the indeterminate neural network models. Among encoder models with different latent space dimensions, only the smallest size of 8 returns a considerably lower score.

The fact that phase classification does not achieve higher scores when working on latent space representations, shows that in this usecase, the latent space is not easier to interpret than the original space. However, since the difference is only about 1%, we can say that the properties of the vertices are at least preserved in the latent space.

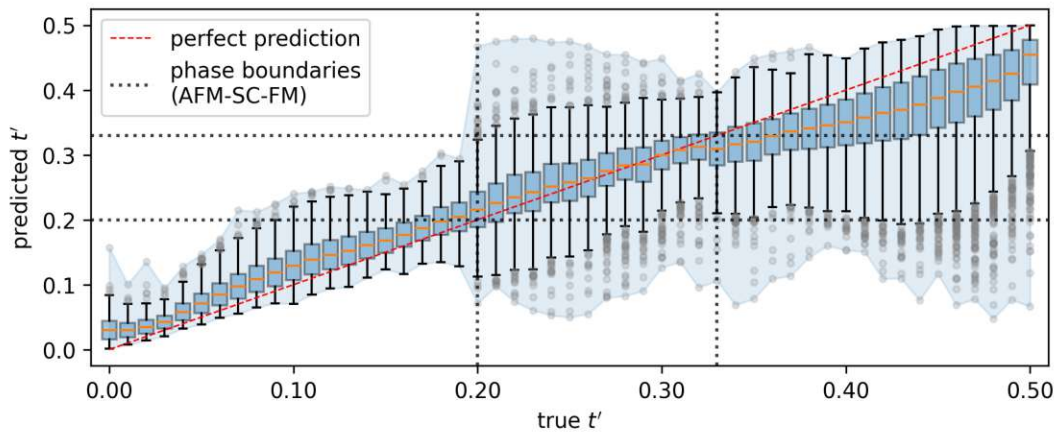
5.3.5 Contrastive learning

In physics, phases correspond to clearly different states, inducing very different properties. The fact that the encoder model is able to extrapolate to unseen phases so well suggests



(a) Polynomial regression based on spline with 4 knots

(b) Polynomial regression based on spline with 10 knots



(c) Random forest regression

Figure 5.11: RMSE per vertex (by t' -parameter) for different regressor models.

that the model learns a continuous representation of the vertex features. This leads to the question, what happens, if the encoder model is forced to differentiate phases.

For this we train the autoencoder using a contrastive learning strategy. Contrastive learning focuses the model training to distinguish between samples of different classes. Here, it ensures that samples of the same phase are mapped closely together in the latent space and far away otherwise.

This is realized by combining the encoders MSE loss with the InfoNCE loss [vdOLV18]. InfoNCE compares the output to a different sample from the same phase (which should be mapped close in the latent space) as well as to one sample from every other phase (which should be mapped far away). The InfoNCE loss is added to the MSE loss to

5. RESULTS

run	wall time [hours]	min. loss	epoch of min. loss	epoch of min. loss (10% threshold)	max. epochs w/o decrease
$S_{[AFM+SC+FM]}$	30	0.017430	49,891	1,035	39,344
$S_{[AFM+FM]}$	15	0.002904	20,281	1,331	12,850
$S_{[SC+FM]}$	14	0.009826	48,288	3,151	37,298
$S_{[AFM+SC]}$	16	0.014092	49,349	1,971	41,767

Table 5.3: Statistics for the training of each scenario (with latent space dimension of 32 and subsamples per vertex of 24,000).

create a latent space optimized for both full vertex reconstruction and phase distinction.

When using contrastive loss, the batch size is divided by four (to 2048), since here each input sample is a combination of four subsamples (the main subsample, a sample from the same phase as a positive example and a sample from each of the other two phases as negative examples). These models were trained for 50,000 epochs since they converge a lot slower. Contrastive training takes around 30 hours using all data.

Training convergence

Figure 5.12 shows the progress of the validation loss for different scenarios.

Using the contrastive loss approach, models converge slower to an extent that for some models even after twice the number of epochs, the loss is still visibly decreasing.

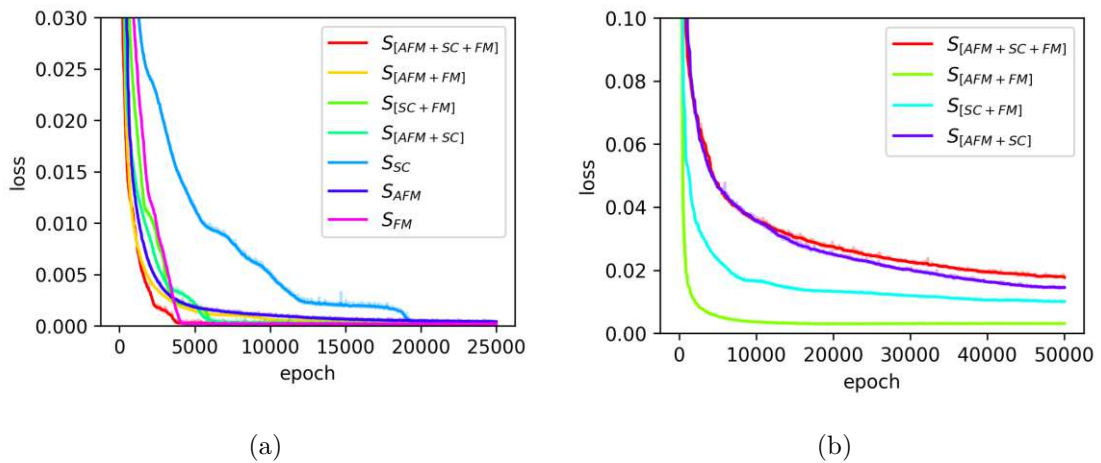


Figure 5.12: Progress of validation loss for each scenario using MSE loss only (a) and contrastive learning loss (b).

Results with contrastive learning

With contrastive learning, the reconstruction errors are higher for all scenarios (see Fig. 5.13b). This is not unexpected: The model is forced to cluster the latent space by phase, which corrupts the continuity in the latent space so that average reconstruction over the full range of vertices is less accurate.

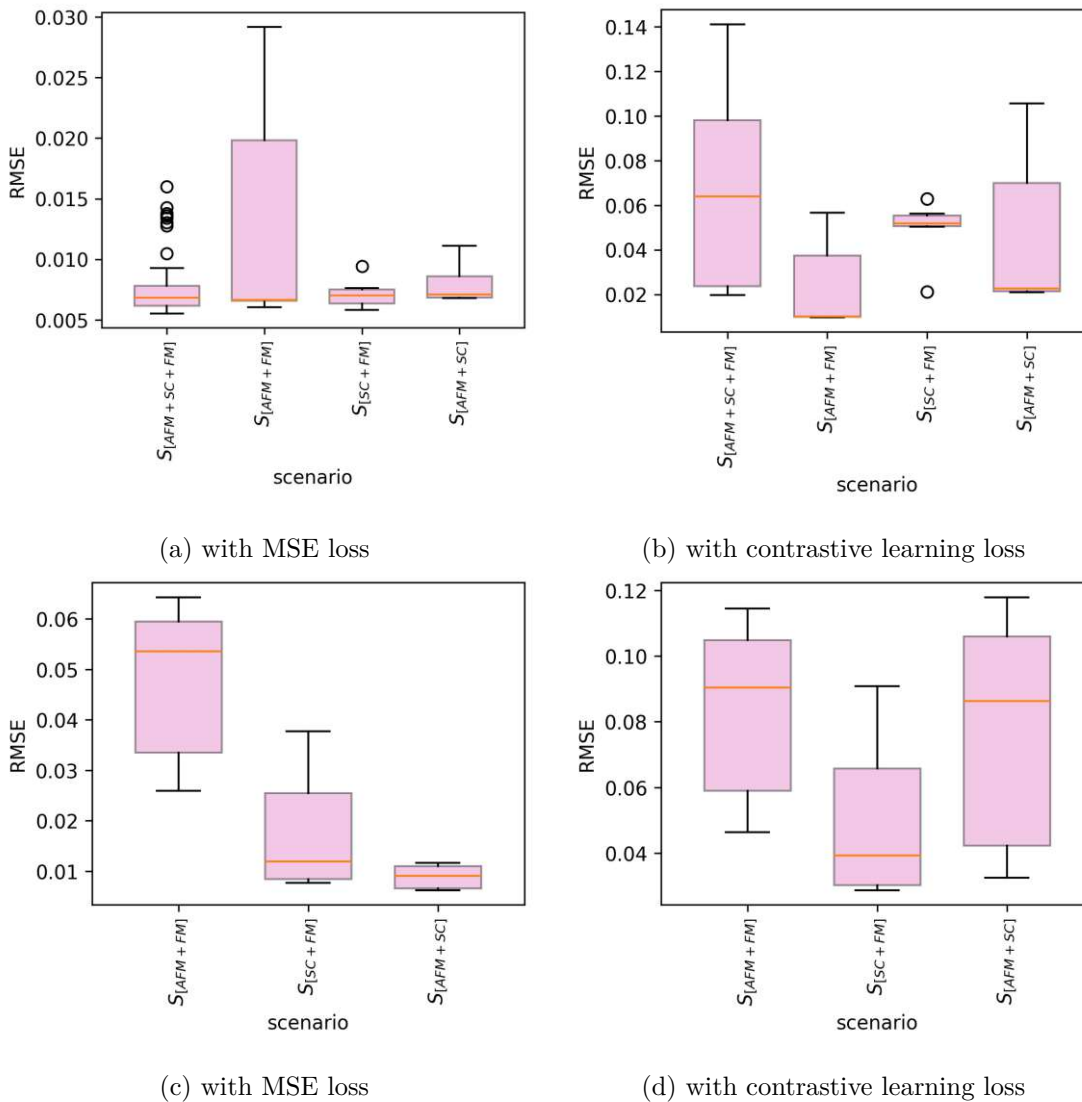


Figure 5.13: Reconstruction errors (RMSE) for each scenario using MSE loss only (a) and contrastive learning loss (b) for vertices from trained phases, and for vertices from phases not seen during training (c) resp. (d).

In Fig. 5.14 we see that the latent space is easier to interpret with respect to phase classification, if it is clustered by phase. All classifier models score higher at above 0.99, where previous models score around 0.9875. The difference is not great, but consistent along all models (where the difference in scores among models of the same loss type is very small).

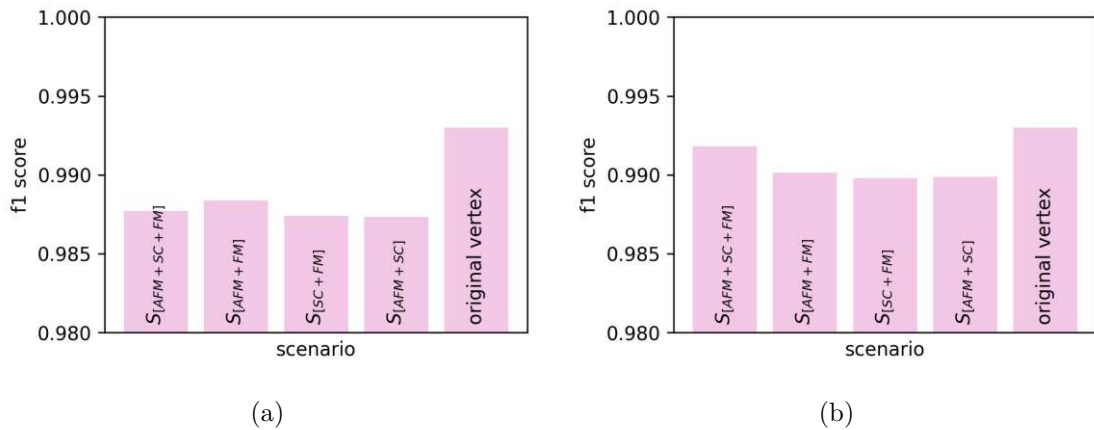


Figure 5.14: F1-score for classifying the phase of vertices from latent spaces produced by the encoder models of all scenarios using MSE loss only (a) and contrastive learning loss (b).

5.4 Predicting new vertices

It is possible to reconstruct vertices the model has not seen before. A next step would be to construct a vertex with a different t' from the input vertex. We train the model in such a way that, instead of reconstructing the input vertex, it returns the next vertex corresponding to the subsequent value of t' augmented by 0.01. In Fig. 5.15 we see immediately that all RMSE are much higher (up to 100x).

We can look at the predicted values more closely in Fig. 5.16. The plot shows for each subrange of the value domain of the input vertex the true average and the average of the predicted values. The model has difficulty predicting extreme values (highs and lows). For $t' = 0.06$, having the lowest RMSE, predictions for the highest values are especially far off. For vertices with the highest RMSEs, we can see that most predictions are around the same value, suggesting that the model just returns an average value instead of reproducing patterns.

Fig. 5.17 shows some visualizations of sections through different vertices. It can be seen

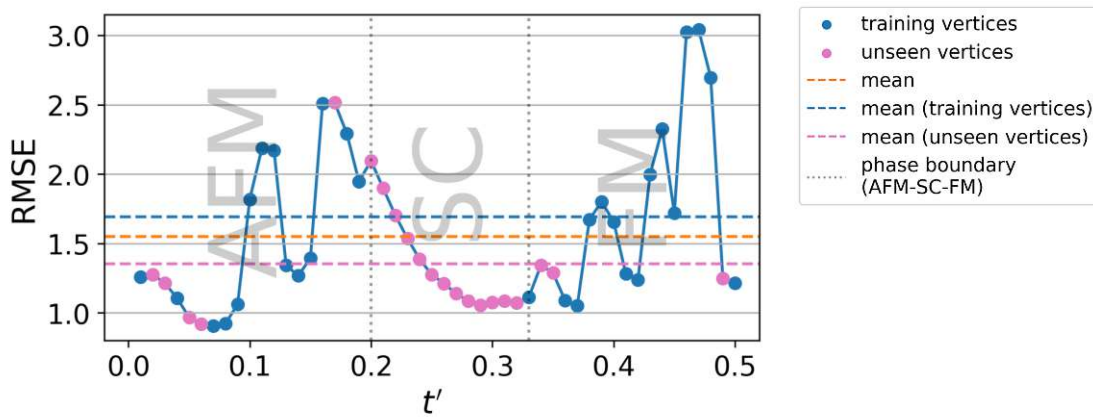
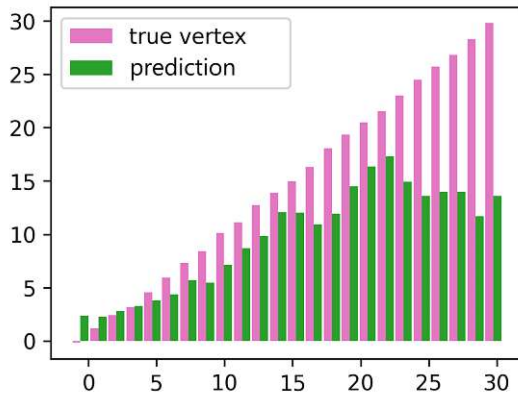
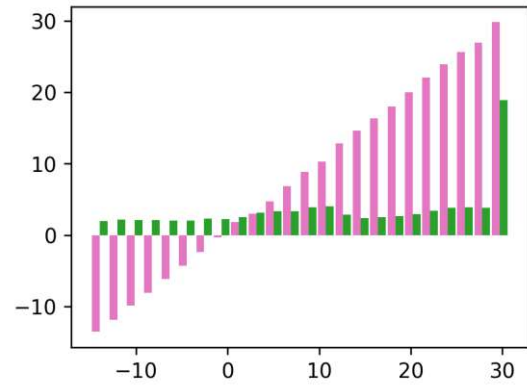


Figure 5.15: Reconstruction errors (RMSE) for every predicted vertex (identified by the value of t') when given a vertex for the preceding t' as input.

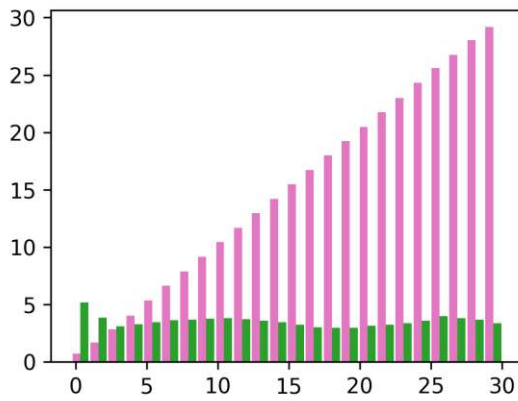
that the predictions with higher RMSEs fail to capture any real patterns.



(a) $t' = 0.06$, $RMSE = 0.92$



(b) $t' = 0.17$, $RMSE = 2.52$



(c) $t' = 0.47$, $RMSE = 3.04$

Figure 5.16: Means of true and predicted values for each subrange of the input vertices value domain for selected vertices.

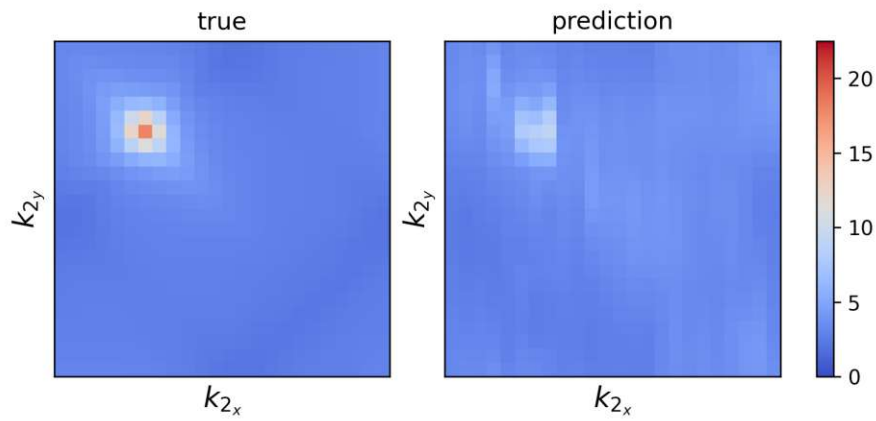
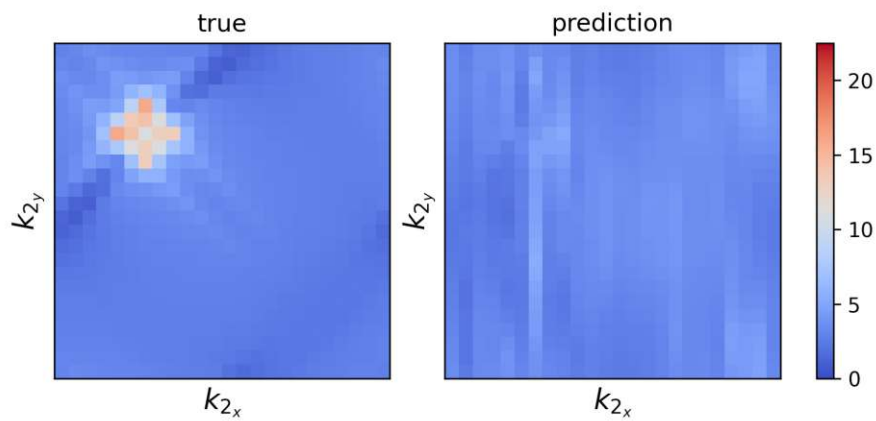
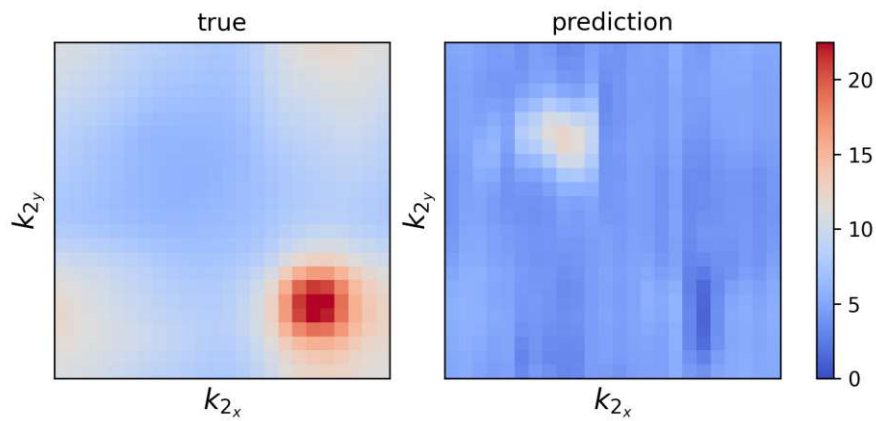
(a) $t' = 0.06$, $RMSE = 0.92$ (b) $t' = 0.17$, $RMSE = 2.52$ (c) $t' = 0.47$, $RMSE = 3.04$

Figure 5.17: Visualizations of slices through different true vertices and the prediction at the subsequent value of t' from the input vertex. Each slice is through $(k_{1_x} = 18, k_{1_y} = 18, k_{3_x} = 18, k_{3_y} = 18)$.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Conclusion and Outlook

In the first part of this work we trained an encoder model which creates a smaller dimensional representation of vertices in latent space. We showed that this space has learned the underlying physical concepts of the vertex function well enough to reconstruct vertices, not seen by the model during training, and even vertices from phases yet unseen by the model.

Subsequently, we created a classifier to determine the phase from the latent space representation of the vertex, to assess its physical content. In this context, we further investigated different properties of this latent space representation by clustering it and applying contrastive learning on the encoder model.

Finally, we tried to predict the vertex for the subsequent t' given any vertex as input, however, did not achieve a satisfactory accuracy.

We first summarize the results for the encoder model followed by the phase classification. Afterwards, we discuss potential methodological improvements and possible further investigations.

6.1 Vertex latent space model

With the development of the vertex autoencoder, we foremost addressed two questions: how we can compress a vertex into a smaller dimensional representation and whether we can generalize knowledge learned from some vertices to other vertices.

With the approach developed here, we did not directly achieve a considerable reduction of the objects size. Using the smallest latent space dimension of 8, that we evaluated, results in a reduction to 33% of the original size. Using a latent space dimension of 32, which achieves considerably lower reconstruction errors, results in an encoded size of 133% of the original size – so an increase.

However, we discovered that only a small random subset of a full vertex is sufficient to train a latent space. We evaluated models using between 0.15 and 4.8% of the data in a vertex. With only 4.8% of randomly selected vertex subsamples we already achieve extremely low reconstruction errors of ca. 0.007 RMSE. Furthermore, these models can infer vertex features from a single vertex subsample. This proves that most of the information within a vertex can be represented by only a small random subset and, therefore, is repetitive and redundant.

We trained models on vertices from only some phases and then reconstructed vertices from other phases. We found that SC phase vertices are slightly more difficult to infer from other phases. However, for all phases, we demonstrated that the models are able to transfer knowledge to yet unseen phases with very low errors.

In physics, phases are quite different states, with specific distinct properties. However, the evaluation of the encoder models suggests that the models identify and model a continuous representation of the vertices features, which enables to extrapolate to unseen phases.

6.2 Latent space analysis

We analyzed the latent space by using it in a classifier that determines the vertex phase. The classifier can infer the phase from a single subsample with a very high success-rate. However, classifying the latent space achieved lower scores than directly classifying subsamples from the original vertices. For this usecase the latent space representation did not yield a better interpretability than the original vertex. However, the fact that very high scores are achieved in either way shows that the latent space captures the vertex features sufficiently well.

We then trained the encoder model employing contrastive learning, so that the latent space learns to better differentiate between phases. As expected, this improves phase classification but increases the reconstruction error. This supports the assumption that

the latent space contains continuous features. When clustered by phase, the continuity and generalization ability of the latent space is diminished.

Since we can assume a continuous vertex space with respect to the t' -parameter, we tried to train a model that takes a vertex as input and finds a corresponding vertex one step further on the t' scale. The model then returns a vertex with $t'_{out} = t'_{in} + 0.01$. Unfortunately, the predicted vertex deviates greatly from the true vertex at the respective t' . The model is not able to predict values in the vertex near both ends of the value-domain, often missing subregions in the vertex of high variation.

6.3 Further research

An issue within the vertex tensors is that the vast majority of values are very small, whereas high values are rare and clustered in a few small areas in the vertex space. To account for this, the subsampling strategy can be further adapted to a stratified sampling, such that for each vertex an approximately uniform distribution of values is sampled. This avoids bias towards small values. Furthermore, since we only sample up to 5% of a vertex, there is currently the danger that no subsample covers any areas of high values in the vertex, thus carrying little information about the interactions within the vertex.

There is currently also an imbalance with respect to phases. Each phase covers a different number of vertices such that in any training dataset, the superconducting phase is especially underrepresented (20 vertices for AFM, 13 for SC, 18 for FM). We can reduce this imbalance by drawing an equal number of subsamples per phase. It does not remove imbalance completely though, since more different vertices add more diversity than drawing more subsamples per vertex. This is already applied for the contrastive learning approach, because it requires an equal number of samples per vertex, since it matches each sample with a sample from the other phases.

It may also be interesting to investigate more elaborate encoder model architectures. The field of object detection inspired us to employ an autoencoder. However, the autoencoder has long been replaced by more complex architectures like the U-net or transformers. While these models are computationally more demanding, they greatly improve over the autoencoder. The U-net may easily replace an autoencoder since their implementation is very similar, only that the U-net has so-called skip connections. These skip connections pass data from encoder layers to decoder layers. This helps to retain fine-detailed features which otherwise may get lost during encoding. For this, one could implement the U-net

using linear layers and aggregate the data from the previous layer and the skip-connection by averaging.

Although predicting the subsequent vertex on the t' domain does not achieve satisfactory results, we expect it to be theoretically possible. Here also, the U-net or even more advanced architectures like the Transformer may achieve better results. Another possibility is finetuning a reconstruction model: First, a model is trained for vertex reconstruction, then training is continued using the subsequent vertex as a target for each input vertex.

The present analysis can be further generalized to other more complex physical models beyond the Hubbard model. The latent space representation can be applied beyond the extraction of physical information. Currently, the large size of a vertex represents the actual bottleneck in computational quantum many-body theory. Our findings can be used as a starting point for developing efficient computational algorithms, which would imply a major step forward.

Overview of Generative AI Tools Used

- *Microsoft 365 Copilot* using *OpenAI GPT-5* for generating latex code (formatting figures and tables).



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [DSMacT⁺22] Domenico Di Sante, Matija Medvidović, Alessandro Toschi, Giorgio Sangiovanni, Cesare Franchini, Anirvan M. Sengupta, and Andrew J. Millis. Deep learning the functional renormalization group. *Phys. Rev. Lett.*, 129:136402, Sep 2022.
- [EFG⁺05] Fabian H. L. Essler, Holger Frahm, Frank Göhmann, Andreas Klümper, and Vladimir E. Korepin. *The Hubbard Hamiltonian and its symmetries*, page 20–49. Cambridge University Press, 2005.
- [HS01] Carsten Honerkamp and Manfred Salmhofer. Temperature-flow renormalization group and the competition between superconductivity and ferromagnetism. *Phys. Rev. B*, 64:184516, Oct 2001.
- [Hub63] J. Hubbard. Electron correlations in narrow energy bands. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 276:238 – 257, 1963.
- [LH19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [MDT⁺24] Emin Moghadas, Nikolaus Dräger, Alessandro Toschi, Jiawei Zang, Matija Medvidović, Dominik Kiese, Andrew J. Millis, Anirvan M. Sengupta, Sabine Andergassen, and Domenico Di Sante. Compressing the two-particle green’s function using wavelets: Theory and application to the hubbard atom. *The European Physical Journal Plus*, 139(8), 2024.
- [MSH⁺12] Walter Metzner, Manfred Salmhofer, Carsten Honerkamp, Volker Meden, and Kurt Schönhammer. Functional renormalization group approach to

correlated fermion systems. *Reviews of Modern Physics*, 84(1):299–352, March 2012.

- [Sum22] Thomas Sumner. Artificial Intelligence Reduces a 100,000-Equation Quantum Physics Problem to Only Four Equations. <https://www.simonfoundation.org/2022/09/26/artificial-intelligence-reduces-a-100000-equation-quantum-physics-problem-to-only-four-equations/>, accessed: 18/10/2024, 2022.
- [Tos11] Alessandro Toschi. *Strong Electronic Correlation in Dynamical Mean Field Theory and beyond*. Professorial dissertation, Technische Universität Wien, 2011.
- [vdOLV18] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018.
- [YIM18] Susumu Yamada, Toshiyuki Imamura, and Masahiko Machida. High Performance LOBPCG Method for Solving Multiple Eigenvalues of Hubbard Model: Efficiency of Communication Avoiding Neumann Expansion Preconditioner - Scientific Figure on ResearchGate. https://www.researchgate.net/figure/A-schematic-figure-of-the-2-dimensional-Hubbard-model-where-t-is-the-hopping-parameter_fig2_323863889, accessed: 18/10/2024, 2018.
- [Zab24] Daniel Zabielski. Wavelet and machine learning compressions in quantum many-body physics: Compact vertex function representations. Bachelor’s thesis, Technische Universität Wien, 2024.
- [ZMK⁺24] Jiawei Zang, Matija Medvidović, Dominik Kiese, Domenico Di Sante, Anirvan M. Sengupta, and Andrew J. Millis. Machine learning-based compression of quantum many body physics: Pca and autoencoder representation of the vertex function. *Machine Learning: Science and Technology*, 5(4), 2024.