Machine Learning of Many Body Localization

Philipp Krüger
(Dated: July 28, 2020)

The goal of this study was to find the quantum phase transition at intermediate local disorder strengths on a Heisenberg chain. Exact diagonalization was used to find the reduced density matrices for a different number of consecutive spins for the lowest energy eigenstate of the Heisenberg model with an additional random field in z-direction at low and high disorder strengths. The resulting dataset representing extended and localized phases was used to train a neural network. Afterwards, the trained network was applied on intermediate disorder strengths to deduct the critical disorder strength for a phase transition. This phase transition was predicted for all system sizes to be around $W_c = 2.5J$ for the system sizes $L \in \{8, 9, 10, 11, 12\}$ and block sizes $n \in [1, 6]$. Low block and system sizes suffered from low accuracy and high losses in the machine learning model, whereas for $n > 3$ block sizes the W_c value showed smaller deviations from a previously published theoretical value $W_c \approx 3.6$ calculated with entanglement entropy on systems up to $L = 22$. This deviation can be attributed to the effect of the smaller system sizes and the effect of open boundary conditions.

I. INTRODUCTION

The physical model and the concept of exact diagonalization is presented first. As we use reduced density matrices as features for the neural network, we explain briefly their computation and meaning.

A. Physical model

1. Hamiltonian of the Heisenberg model and physical expectation

The Hamiltonian of the Heisenberg model is shown in equation 1. In the course of further analysis, we choose $J = 1$ and sample h from a uniform distribution such that $h_i \in [-W, W]$.

$$H = \underbrace{J \sum_i \vec{S}_i \cdot \vec{S}_{i+1}}_{\text{Exchange Energy}} - \underbrace{\sum_i h_i S_i^z}_{\text{Random Field}} \tag{1}$$

The expectation for the ground state is dependent on the ratio of the coupling and the local random field.

For $\frac{W}{J} \ll 1$, we expect a delocalized, extended phase, since the exchange energy dominates over the small external field. Therefore, the system can relax to thermal equilibrium serving as its own heat bath in the limit of large system size $L \to \infty$. Here, the reduced density operator of a finite subsystem converges to the equilibrium thermal distribution for $L \to \infty$.[1]

For $\frac{W}{J} \gg 1$, we can expect a localized phase, since the h_i factors dominate over the exchange energy. The resulting states are expected to be product states of spins "up" or "down", as the external field points in z-direction. Also, an infinite system cannot equilibrate itself. The local configurations are set by the initial conditions at all times and are adiabatically connected to the trivial state.[1]

2. Characterization of ergodic and localized regimes by different metrics

There are a few different ways of how to distinguish the ergodic and the localized regime by accessible metrics. One can study spectral analysis, where the ergodic states are distributed like a Gaussian Orthogonal Ensemble (GOE), and the localized states follow a Poisson distribution.[2]

Another interesting metric is the entaglement entropy, which indicates the information spread between different system parts.[3] In the ergodic phase, the reduced density matrix ρ_A of a ground state is expected to be thermal. This leads to a volume law scaling for the entanglement entropy.[4] On the other hand, localized eigenstates show area-law scaling.[5]

In this study, we solve the Hamiltonian via exact diagonalization to predict the phase change by training a neural network with low and high disorder strengths, assuming that the resulting reduced density matrices represent an ergodic and localized phase respectively. To access the ground states, previous approaches made use of the shift-invert method [6]. The same paper suggests a critical disorder strength of $h_c = 3.62$ for the same system by using the evolution of the entanglement entropy over different system sizes as an argument.

B. Exact diagonalization

Exact diagonalization (ED) is a numerical technique we can use to solve the time independent Schrödinger Equation $H |\psi\rangle = E |\psi\rangle$ for the eigenvalues E and eigenvectors $|\psi\rangle$. This only works if the Hamiltonian H represents a discrete and finite system. Most quantum many-particle problems lead to a sparse matrix representation of the Hamiltonian, where only a very small fraction of the matrix elements is non-zero.[7] An efficient method to find ground states is the Lanczos algorithm.[8] At first, the algorithm was numerically unstable. This issue was overcome in 1970 by Ojalvo and Newman.[9] In this study, we rely on the Lanczos algorithm for the eigensolver.

C. Reduced Density Matrix

The usefulness of reduced density matrices has already been shown by White in 1992 with ground states of Heisenberg chains.[10] In our case we use areal density matrices as features for the neural network to predict the critical disorder strength of a phase change from delocalized to localized. The reduced density matrix is defined in equation 3. Physically, the reduced density matrix ρ_A, provides correct measurement statistics for subsystem A.

$$\rho_{AB} = |\psi_A\rangle \langle\psi_A| \otimes |\psi_B\rangle \langle\psi_B| \tag{2}$$

$$\rho_A = \mathrm{Tr}_B(\rho_{AB}) = |\psi_A\rangle \langle\psi_A| \,\mathrm{Tr}\left(|\psi_B\rangle \langle\psi_B|\right) \tag{3}$$

The reduced density matrix was also used by Zhang in 2019 to learn the localization transition in disordered quantum Ising spin chains. Here, the motivation was to reduce the dimension and filter out redundant information. However, it proved to be inferior in comparison to the full density matrix in the analysis.[11] However, due to RAM limitations, we will rely on reduced density matrices.

D. Artificial Neural Networks

In 1958, Rosenblatt published his concept of the probabilistic model for information storage and organization in the brain, which greatly inspired others to use those models for computation.[12] Over the course of the last decades, they have evolved into a tool that can be used for a variety of applications including computer vision, speech recognition, medical diagnosis, playing games or even artistic painting.[13]

The reduced density matrices are essentially complex 2D arrays with length $2^n \times 2^n$. As we want to classify for an arbitrary W whether we have a localized or delocalized phase, it is convenient to use a machine learning classifier. The density matrices can then be thought of as a complex and real image that can be fed into it analogously to classical image classification. A prominent tool to use in spatially or temporarly related data is the Convolutional Neural Network, which will be applied here for two reasons: Firstly, they reduce the total number of weights, which is convenient for a low sample size [14]. Secondly, the thermalization of the density matrix may be learnable by a convolution kernel because we are expecting the density matrices to show different local distributions.

II. COMPUTATIONAL METHODS

The strategy for implementation was as follows:

1. Generate Hamiltonian from random disorder strength and system size. Then calculate lowest eigenstate near Energy $E = 0$.
2. Generate density matrix from the eigenstate and the respective reduced density matrices for defined block sizes n.
3. Set up machine learning model per n, L that takes density matrices of different W as an input and predicts whether the state represents an extended or a localized phase.
4. Make predictions for different system sizes L and block sizes n and plot the predictions over W. Then extract W_c from the data by using a fit function.

Critical decisions and specifications for each steps are listed below. Afterwards, a brief motivation for the parameter range and resolution is given.

A. Eigenvalue solver and shift invert method

For the eigenvalue solution, we use SciPy's method `eigsh`. However, the computation of eigenvalues near zero is computationally very costly and sometimes did not even converge during the standard maximum of iterations.

Most ARPACK functionalities are included in `eigsh`. To compute the eigenvalues, we are relying on the so called shift-invert method, which is a mode that allows a quick determination of non-external eigenvalues. This mode involves transforming the eigenvalue problem to an equivalent problem with different eigenvalues.

In particular, the method is based on the observation that one can find for the generalized eigenvalue problem

$$Ax = \lambda M x \tag{4}$$

that

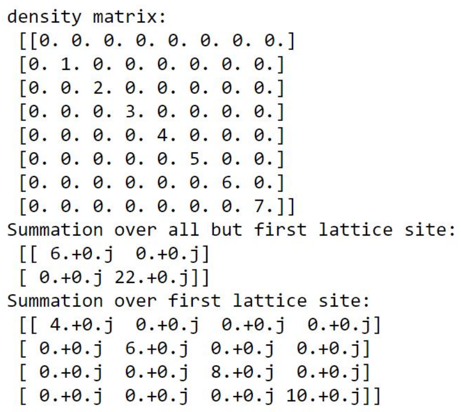
$$(A - \sigma M)^{-1} M x = v x, \quad v = \frac{1}{\lambda - \sigma}. \tag{5}$$

As we want to find the ground state, our choice for σ is zero. The transformed eigenvalues will then satisfy $v = 1/\lambda$, so our small eigenvalues become large eigenvalues for which the Lanczos algorithm converges faster.[15] This method is also used as part of a efficient phase characterization method by Luitz in 2015.[6]

B. Computation of the reduced density matrix

To get the reduced density matrix of system A, one has to "trace out" all states outside of A. The library QuTiP supplies a method `ptrace`, which does exactly that. It is important to note that the method takes those indices as an argument which should be kept.[16]

A demonstration of the functionality can be found in Figure 1.



```
density matrix:
 [[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 0. 0. 0. 0. 0.]
 [0. 0. 0. 3. 0. 0. 0. 0.]
 [0. 0. 0. 0. 4. 0. 0. 0.]
 [0. 0. 0. 0. 0. 5. 0. 0.]
 [0. 0. 0. 0. 0. 0. 6. 0.]
 [0. 0. 0. 0. 0. 0. 0. 7.]]
Summation over all but first lattice site:
 [[ 6.+0.j  0.+0.j]
 [ 0.+0.j 22.+0.j]]
Summation over first lattice site:
 [[ 4.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  6.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  8.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j 10.+0.j]]
```

Figure 1: Proof of concept for partial trace calculation similar to QuTiP-Guide/ptrace.

The algorithm of selecting the position vector of n consecutive sites was implemented as follows:

1. Find the center spin rounded to next lowest integer.
2. Determine left chain length n_{left} to be $n/2$, which is rounded to the next lowest integer.
3. Determine right chain length n_{right} as $n - n_{\text{left}}$.
4. Select spins from left chain end to right chain end around center spin.

This results in a behavior that picks left indices more favorably, but succeeds if equally spaced ends exist. Let the spins be numbered as $\{1, 2, 3, 4, 5\}$ for $N = 5$, then $n = 3$ results in $\{2, 3, 4\}$, whereas $n = 2$ results in $\{2, 3\}$.

These lattice sites then serve as an input to the partial trace function, such that the density matrix represents the measurement statistics of the center system.

C. Machine learning models and error metrics

The decision for the machine learning framework `keras` was motivated by its flexibility and simplicity.[17]

When setting up the machine learning model, one can already specify the first and last layer: The first (input) layer has to match the sample size of the incoming data, this can be already computed in advance. The length *len* for block size n is $2 \cdot (2^n \times 2^n)$. The factor 2 comes from a preprocessing step, where the complex values are mapped to a second real picture, since the fitting procedure usually does not expect complex numbers. The last layer is a one node sigmoid, as the target output is the one-dimensional classification in $[0, 1]$.

For small sample sizes, there exist various approaches to choose the right amount of layers and regularization methods, which cannot be generalized, as they heavily depend on feature size and target dimension.[18, 19]

To balance off the trade between overfitting and loss, the starting point for the model was one hidden layer with 64 nodes. Since the reduced density matrices are similar to image classification and the inspection of the training set indicated that the density matrices had different slopes, a convolutional layer was employed for block sizes of $n < 3$, as a $8x8$ picture seemed too small for kernel operations. To compensate the lacking layer, a dense layer of 32 nodes was chosen.

The optimizer Adam was chosen, because it is computationally efficient and has little memory requirements.[20]

For a two label classification problem, it is useful to use cross-entropy as a loss metric, as the penalty increases exponentially the further one deviates from the correct prediction.[21] The definition for a two class cross-entropy loss can be found in equation 6, where $y \in \{0, 1\}$ is the true class and $\hat{y} \in [0, 1]$ the predicted probability. This loss is also plotted in Figure 2.

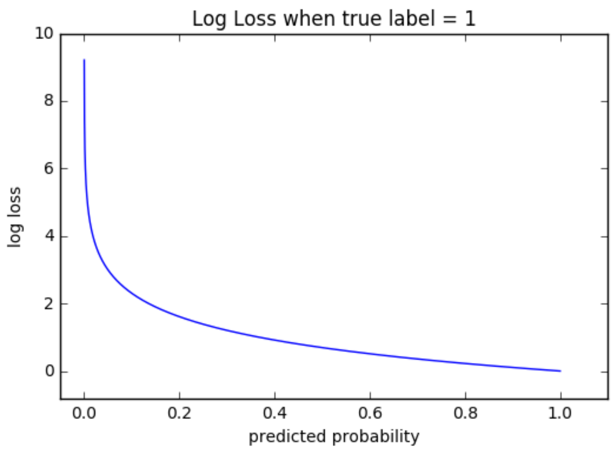


Figure 2: Example for cross-entropy loss depending on the predicted probability of $\hat{y} = 1$.

$$L(\hat{y}, y) = -\left(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right) \tag{6}$$

D. Extraction of critical disorder strength W_c

To fit for the critical disorder strength W_c, 180 sample predictions were averaged and plotted over the corresponding disorder strength, for which the test sample was initialized. The first approach was to fit for W_c by using parameters of a fitted linear function or logistic function. As this approach proved to be unstable and not prone to outliers, a far simpler method was employed, which just extracted the nearest guess to $\hat{y} = 0.5$, resembling equal prediction probabilities for the localized and ergodic phase.

```
nearest = np.argmin(np.abs(y - 0.5))
```

E. Limitations for parameter range and resolution

1. System size L: Limited by computing time of eigenvalue solver. For the system size $L = 12$, one calculation lasted approximately two minutes.

2. Block size n: We go up to $n = 6$, which is half of the system size of the biggest system.

3. Sample size: 2400 samples can be generated for $L = 12$, $n_{max} = 6$ in approximately 15 hours on the provided machine. Assuming that the whole program should be reproducable in a reasonable time frame, this was found to be a sufficient sample size per system and block size.

4. Disorder strength W for the testing set: Since each point of a test set comes with a Hamiltonian with randomly drawn $h_i \in [-W, W]$, a decent amount of variance can be expected for the phase prediction. As we want to extract the phase change in general, and are not interested in the particular phase predictions of one specific Hamiltonian we choose to regularize the prediction by averaging over 180 predicted samples.

III. RESULTS

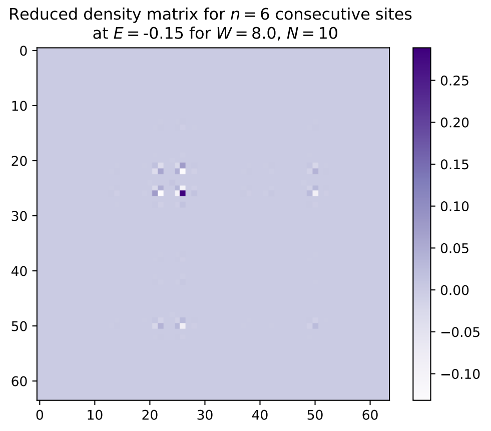
A. Generation of reduced density matrix training set

The parameter range for the computation of the reduced density matrices can be found in Table I. The total computation time was 16.5 h, where 12.5 h where solely needed to compute the ground states of the $L = 12$ system.

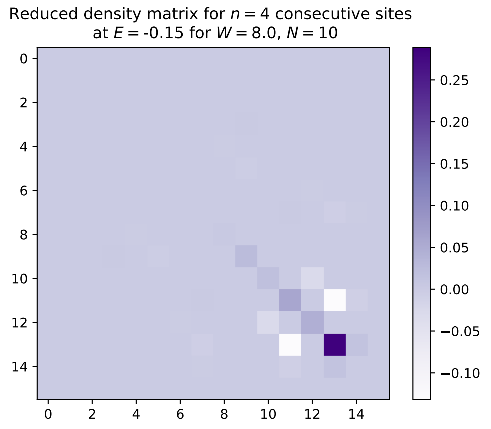
Parameter	Range or Set
System size:	$L \in \{8, 9, 10, 11, 12\}$
Block size:	$n \in \{1, 2, 3, 4, 5, 6\}$
Repetitions:	$r = 500$

Table I: Parameter choice for training set generation

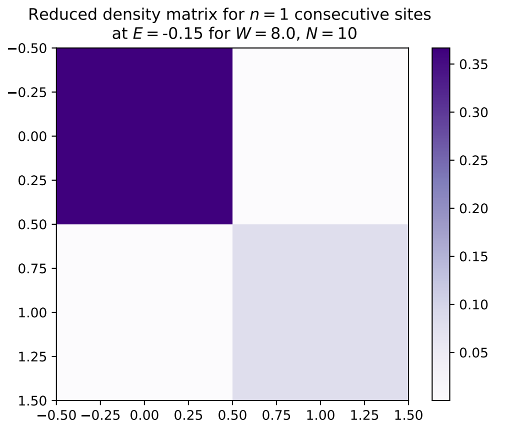
In order to give some visual intuition, Figures 3 and 4 show realizations for different block sizes and phases.



a) Visualization of the ground state for a large block size $n = 6$ in the localized phase.



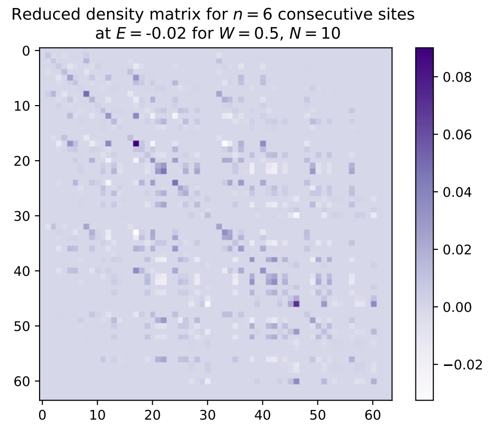
b) Visualization of the ground state for an intermediate block size $n = 4$ in the localized phase.



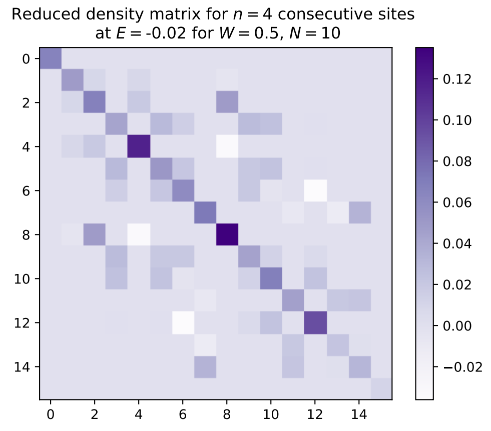
c) Visualization of the ground state for the minimal block size $n = 1$ in the localized phase.

Figure 3: Ground states for different block sizes n and phases.

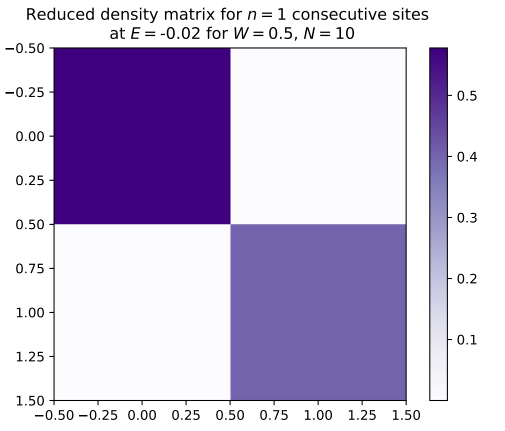
The visual inspection indicates that the density matrix of the localized phase has a sharp maximum at the preferred state that is forced by the random disorder strength. The extended phase shows a sparse but certainly more even distribution, which reflects that some configurations are still more preferred than others in-



a) Visualization of the ground state for a large block size $n = 6$ in the ergodic phase.



b) Visualization of the ground state for an intermediate block size $n = 4$ in the ergodic phase.



c) Visualization of the ground state for the minimal block size $n = 1$ in the ergodic phase.

Figure 4: Ground states for different block sizes n and phases.

duced by the coupling term in the Hamiltonian. The biggest maxima are likely still influenced by the random disorder strength. Another observation is that the density matrix reductions of the full ground state conserved these properties for $n > 2$, when comparing $n = 6$ to $n = [2, 5]$. The similarity between the two phases gets smaller the smaller the block size n gets. For $n = 1$, one could argue that the density matrices are very similar, as they only deviate for half of the matrix elements. In conclusion, the ergodic reduced density matrix shows far more thermalization than the localized one.

B. Model training

Before we can predict the phase of a newly generated test set, we have to train the neural network with our available training data. For each system and block size a separate model was trained, as a different system size might influence the physical behavior due to open boundary conditions.

The neural networks are generated as a sequential keras model with the following configuration, as discussed in section II C:

```
model = models.Sequential()
if self.n > 3:
  filters = self.n*self.n
  model.add(layers.Conv2D(filters, (3, 3),
    activation='relu', input_shape=(np.shape(
    self.X_train[0])[0], np.shape(self.X_train
    [0])[1], 2)))
  model.add(layers.MaxPooling2D((4, 4)))
  model.add(layers.Dropout(rate=0.1))
  model.add(layers.Flatten())
else:
  model.add(layers.Flatten(input_shape=(np.shape
    (self.X_train)[1], np.shape(self.X_train)
    [1], 2)))
  model.add(layers.Dropout(rate=0.1))
  model.add(layers.Dense(32, activation='relu'))
    ,

model.add(layers.Dropout(rate=0.1))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='
    binary_crossentropy', metrics=['accuracy'])
```

To prevent over-fitting, 30 % of the training set was used for validation. To avoid a biased split, we relied on `sklearn`'s method `train_test_split` that samples randomly from the training set. Two dropout layers granted a better regularization, such that the training set was not overfitted.

The model training was executed by using a batch size of 70 and 100 epochs, where the batch size was limited by the CPU performance and no significant loss or accuracy improvements where noted after > 100 epochs.

An example of the accuracy and loss dependency on the number of epochs for system size $L = 8$, and block sizes $n = \{1, 6\}$ is presented below in Figure 5.

Figure 5 illustrates that the model is still not completely prone to overfitting. This can be accounted to the

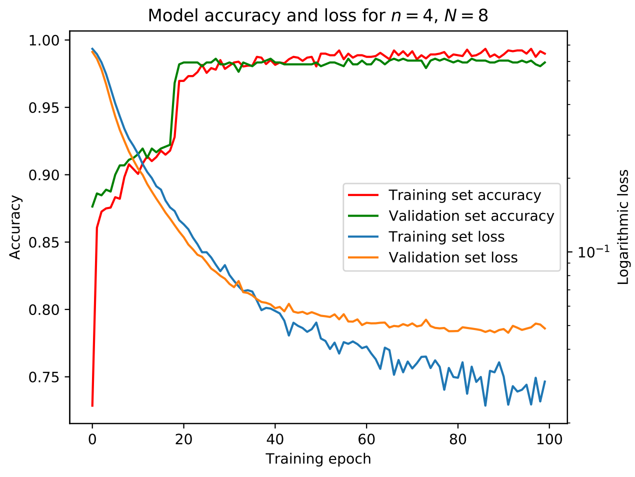


Figure 5: Accuracy and loss evolution over training epochs.

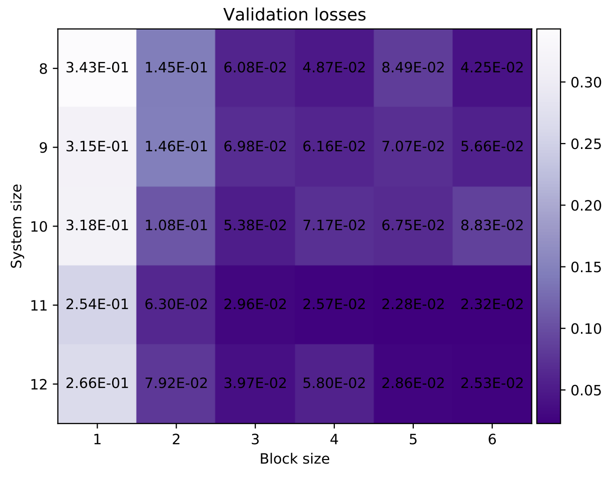


Figure 7: Overview over the resulting losses on the validation set.

big model size in comparison to the sample size. Still, the loss stagnated at an acceptable optimum after 100 training epochs for all block and sytem sizes. The validation loss was the lowest for large system sizes and intermediate block sizes.

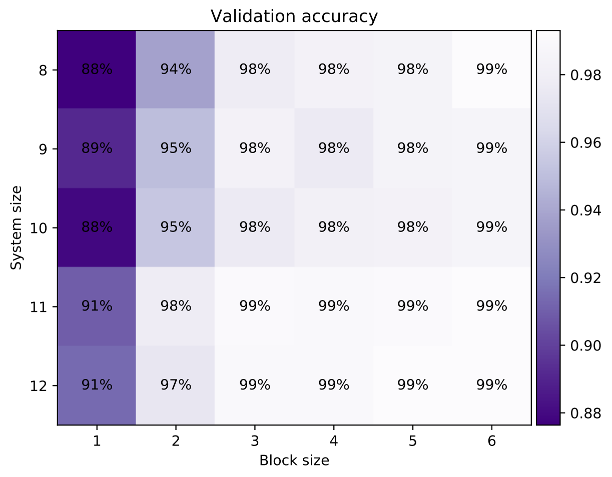


Figure 6: Overview over the resulting accuracies on the validation set.

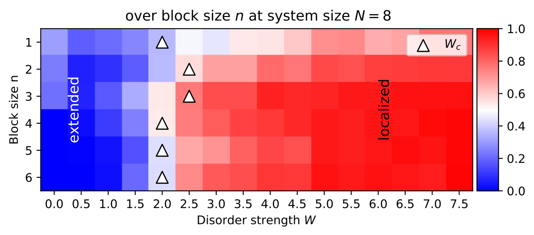
The validation losses in Figure 7 tell a similar story. Here, intermediate block sizes of $n = \{3, 4, 5, 6\}$ show the lowest validation loss. In the intermediate regime, the density matrices have a reasonable trade off between level of detail and number of weights in the neural network, since the first layer must match the input dimensions.

In conclusion, large system sizes and intermediate block sizes showed the best results. In addition, one can safely disregard predictions by models for block size $n = 1$.

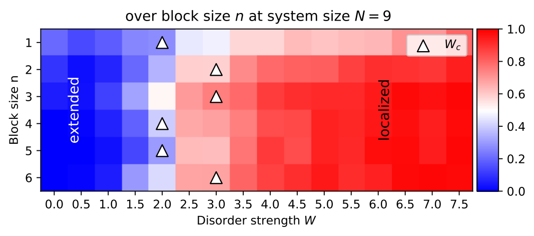
C. Analysis of critical disorder strength

1. Dependency on block size

At first, the testing set was generated. Following the parameter discussion in section II E, we generated 180 samples for each $W \in [0, 4]$, with step $\Delta W = 0.5$, resulting in 1440 samples per system and block size. Afterwards, W_c was obtained as described in section II D. 180 predicted phases are averaged at each point and plotted onto a heat map. W_c is plotted in addition to that in Figure 8.

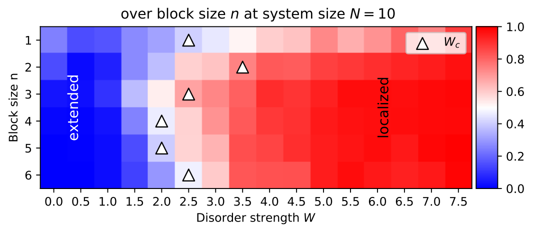


a) $L = 8$

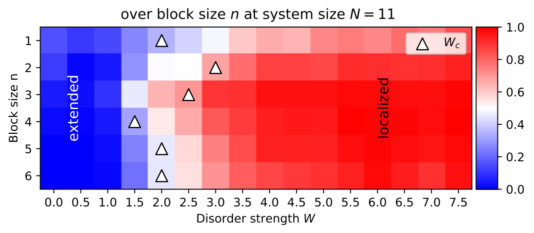


b) $L = 9$

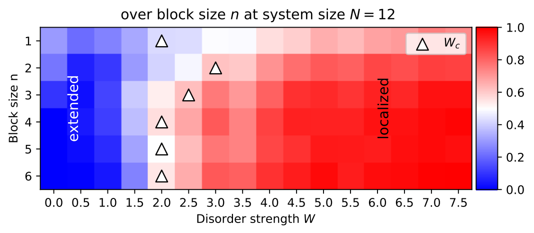
In conclusion, the predicted critical disorder strength W_c decayed, when models with larger block sizes n were



c) $L = 10$



d) $L = 11$



e) $L = 12$

Figure 8: Dependency of the phase transition on block size n for different system sizes.

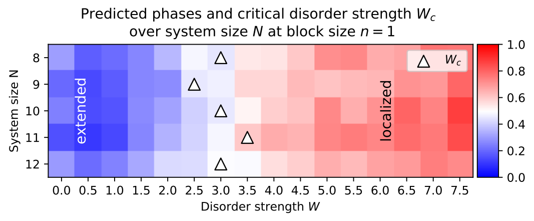
used for prediction. On average, the extracted critical disorder strength was around $W_c = 2.5J$.

2. *Dependency on system size*

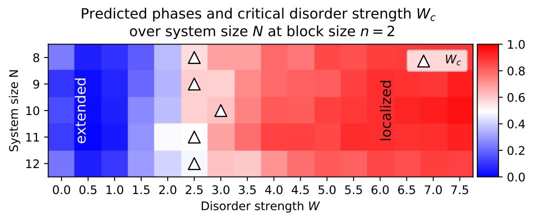
Now, we rearrange the data, such that we can see the dependency on the system size by ordering each prediction set first by block size and plot them over system size.

In comparison to the previous Figure, one can now see the dependency on the system size. If we look at block size $n = 1$, we can see again that the predicitons are not as well defined as for the other block sizes. This weakness can be tracked back to the low validation accuracy of $n = 1$ models, which originated from the reduced density matrices being very similar for the ergodic and localized phase.

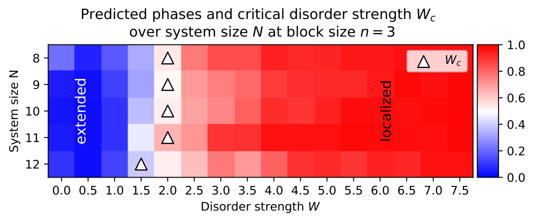
The plots are showing a near to constant dependency on the system size.



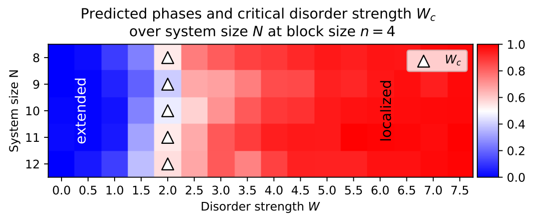
a) $n = 1$



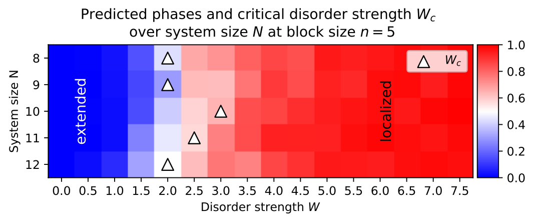
b) $n = 2$



c) $n = 3$



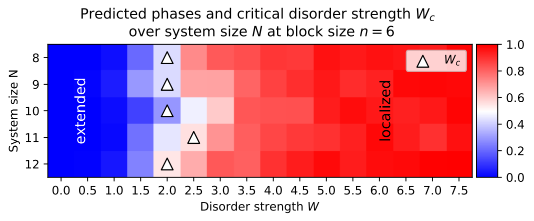
d) $n = 4$



e) $n = 5$

IV. CONCLUSION AND OUTLOOK

In summary, the training set generation could be solved efficiently by employing the shift invert method in combination with the lanczos algorithm. The visual inspection of the density matrices showed that the ergodic phase is more thermalized than the localized phase as expected. Of course, the neural network still requires some tuning. Even though high accuracies were demonstrated, the validation loss was almost an order of magnitude higher than the training loss. An exception can be made for the block size $n = 1$, which was found to be



f) $n = 6$

Figure 9: Dependency of the phase transition over system sizes L for different block sizes n.

very hard to classify, as the thermalization did not have big enough of an impact to the input data. However, the predicitons for intermediate disorder strength showed that a region for the phase transition could successfully be found for all block and system sizes. The classification accuracy outside of the phase transition was almost 100%.

Finally, the extracted critical disorder was hard to classify for a block size of $n = 1$, and did not vary much for block sizes $n = \{4, 5, 6\}$. In the system size analysis, block size $n = 4$ had the least deviations. In the analysis no clear trend was found with respect to finite size scaling. This would probably change if more effort on training set generation could be made. The here presented results can completely be generated within 24h on an average computer, which sets a clear constraint.

Still, a phase change could be consistently observed around $W_c = 2.5$, deviating slightly from the result $W_c \approx 3.6$ in Luitz et al. 2015 for larger system up to $L = 22$.[6] It is expected that smaller systems with open boundary conditions show a smaller disorder strength, as fluctuations are not suppressed as much.

[1] A. Pal and D. A. Huse, Many-body localization phase transition, Phys. Rev. B **82**, 174411 (2010).

[2] C. Laumann, A. Pal, and A. Scardicchio, Many-body mobility edge in a mean-field quantum spin glass, Physical Review Letters **113**, 10.1103/physrevlett.113.200405 (2014).

[3] R. Nandkishore and D. A. Huse, Many-body localization and thermalization in quantum statistical mechanics, Annual Review of Condensed Matter Physics **6**, 15–38 (2015).

[4] E. Altman and R. Vosk, Universal dynamics and renormalization in many-body-localized systems, Annual Review of Condensed Matter Physics **6**, 383–409 (2015).

[5] X. Yu, D. Pekker, and B. K. Clark, Bulk geometry of the many body localized phase from wilson-wegner flow (2019), arXiv:1909.11097 [cond-mat.str-el].

[6] D. J. Luitz, N. Laflorencie, and F. Alet, Many-body localization edge in the random-field heisenberg chain, Physical Review B **91**, 10.1103/physrevb.91.081103 (2015).

[7] A. Weiße and H. Fehske, Exact diagonalization techniques, in *Computational Many-Particle Physics*, edited by H. Fehske, R. Schneider, and A. Weiße (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008) pp. 529–544.

[8] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, Journal of Research of the National Bureau of Standards **45**, 255 (1950).

[9] I. U. Ojalvo and M. Newman, Vibration modes of large structures by an automatic matrix-reductionmethod, AIAA Journal **8**, 1234 (1970).

[10] S. R. White, Density matrix formulation for quantum renormalization groups, Physical Review Letters **69**, 2863 (1992).

[11] W. Zhang, L. Wang, and Z. Wang, Interpretable machine learning study of the many-body localization transition in disordered quantum ising spin chains, Physical Review B **99**, 10.1103/physrevb.99.054208 (2019).

[12] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain., Psychological Review **65**, 386 (1958).

[13] L. A. Gatys, A. S. Ecker, and M. Bethge, A neural algorithm of artistic style, CoRR **abs/1508.06576** (2015), arXiv:1508.06576.

[14] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Networks **61**, 85 (2015).

[15] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, and et al., Scipy 1.0: fundamental algorithms for scientific computing in python, Nature Methods **17**, 261–272 (2020).

[16] J. Johansson, P. Nation, and F. Nori, Qutip: An open-source python framework for the dynamics of open quantum systems, Computer Physics Communications **183**, 1760–1772 (2012).

[17] F. Chollet *et al.*, Keras, `https://keras.io` (2015).

[18] M. Olson, A. Wyner, and R. Berk, Modern neural networks generalize on small data sets, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 3619–3628.

[19] S. Feng, H. Zhou, and H. Dong, Using deep neural network with small dataset to predict material defects, Materials & Design **162**, 300 (2019).

[20] D. Kingma and J. Ba, Adam: A method for stochastic optimization, International Conference on Learning Representations (2014).

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

[22] N. Kyurkchiev and S. Markov, *Sigmoid Functions Some Approximation and Modelling Aspects: Some Moduli in Programming Environment MATHEMATICA* (2015).

Appendix A: Code listing

The process can be broken down into five different steps, which can be all be executed through a main function, but can also be run separately: Generation of the training set, model training, generation if the testing set, generating predictions. Every file serves a number of different purposes as listed below. In general, the time for each step is estimated and logged to the console, such that the user can easily narrow down the parameter range for his own system.

To replicate the results, you can access the whole code and all plots at `https://gitlab.lrz.de/phkrueger/final-project-ph2264.git`. The tutors of PH2264 have been given guest access. Interested readers will be given access by contacting me via `ph.krueger@tum.de`.

1. **main.py**: Executes the whole pipeline.
2. **generate_training_set.py**: Here, the training set is generated and some example plots of ground states are saved to the results folder. The training sets are saved in the `training_sets` folder, where they are numbered with their system and block size.
3. **ed.py**: The training set is generated by using the functions from the tutorial. A new function was added that generates the Hamiltonian using the random local disorder strength.
4. **dataset_preparation.py**: While this file contains many important functions to preprocess and label the training and testing sets and load and save functions, it also has a method that plots the visualizations of a few ground states.
5. **model_save_train.py**: First, models are generated that automatically match the input data of different block sizes n, afterwards, they are trained with a certain amount of epochs and batch sizes. The history of the validation and accuracy is plotted individually into the results folder.
6. **generate_test_set.py**: A set of reduced density matrices for ground states in the intermediate regime is generated.
7. **generate_predictions.py**: The test sets are fed into the trained machine learning models. The predicted phases are averaged for every W, n, N combination and saved into a prediction dataset.
8. **plot_wc_dependency.py**: The predictions are loaded, we extract W_c and plot everything together as a heat map over system and block sizes.

1. Pipeline execution

```python
from generate_training_set import generate_training_set
from model_save_train import train_save_model, plot_model_losses
from generate_test_set import generate_test_set
from generate_predictions import generate_predictions
from plot_wc_dependency import WcPlotter

import numpy as np

if __name__ == "__main__":
    # training set
    Ns = [8, 9, 10, 11, 12]
    n_max = 6
    Ws = [0.5, 8.0]  # 0.5: ergodic/delocalized phase, 8.0: localized phase
    repetitions = 200
    generate_training_set(Ns, Ws, n_max, repetitions)
    # model train
    n_max = 6
    train_save_model(Ns, n_max,
                     batch_size=70,
                     epochs=100)
    plot_model_losses(Ns, n_max)
    # test set
    Ws = np.arange(0., 8.0, 0.5)
    repetitions = 30
    n_max = 6
    generate_test_set(Ns, Ws, n_max, repetitions)
    # predict
    Ws = np.arange(0., 8.0, 0.5)
    ns = np.arange(1, 6 + 1, 1)
    generate_predictions(Ns, ns, Ws)
    # get wc
    Ws = np.arange(0., 8.0, 0.5)
    ns = np.arange(1, 6 + 1, 1)
    wc_plotter = WcPlotter(Ns, ns, Ws)
    wc_plotter.plot_all()
```

2. Training set generation

```python
from ed import *
import time
from dataset_preparation import save_pickle
import qutip
from tqdm import trange, tqdm
from scipy.sparse.linalg import eigsh


def generate_training_set(Ns, Ws, n_max, repetitions):
    start_time = time.time()
    for N in Ns:
        training_set_generator = TrainingSetGenerator(N, Ws, n_max, repetitions)
        print("Training Set N=" + str(N) + " completed after %s seconds." % (time.time() -
    start_time))
        for n in range(1, n_max+1):
            save_pickle("lanczos/training_sets/N" + str(N) + "n" + str(n) + "_Trainset",
                        training_set_generator.training_set[n])
    print("--- Training set generation lasted %s seconds ---" % (time.time() - start_time))
    pass


class TrainingSetGenerator:

    def __init__(self, N, Ws, n_max, repetitions):
        self.N = int(N)
        self.n_max = n_max
        self.repetitions = repetitions
        self.Ws = Ws
        self.training_set = self.generate_training_set_m_lanczos_list()  # self.
    generate_training_set_list()

    def generate_training_set_m_lanczos_list(self):
        """
        Returns training set with shape samples x [density matrix, W, lattice sites, block size,
    ground state energy]
        :return: training set
        """
        training_set = {consecutive_spins: [] for consecutive_spins in range(1, self.n_max + 1)}
        for rep in trange(self.repetitions):
            for W in self.Ws:
                Es, vs = self.get_ground_states(W)
                for i in range(len(Es)):
                    rho = np.outer(vs[:,i],vs[:,i])
                    for n in range(1, self.n_max + 1):
                        reduced_rho = self.get_partial_trace_mid(rho, n)
                        training_set[n].append([reduced_rho, W, self.N, n, Es[i], rep])
        return training_set

    def get_ground_states(self, W):
        hs = np.random.uniform(-W, W, size=self.N)
        # print(hs)
        H = gen_hamiltonian_lists(self.N, hs, J=-1)  # J defined as in original task
        try:
            Es, vs = eigsh(H, k=6, sigma=0, which='LM', tol=0.01)  # SM 1.4s, sigma=0, LM 5.2/s
            # sigma=0, 'LM' for shift invert mode Eigval near to zero
            # following the advice of https://docs.scipy.org/doc/scipy/reference/tutorial/arpack.
    html
        except:
            Es, vs = self.get_ground_states(W)
        return Es, vs

    def get_partial_trace_mid(self, rho, n):
        """
        calculates partial trace of middle n sites
        :param rho: full density matrix
        :param n: block size
        :return: reduced density matrix
        """
        kept_sites = self.get_keep_indices(n)
        qutip_dm = qutip.Qobj(rho, dims=[[2] * self.N] * 2)
        reduced_dm_via_qutip = qutip_dm.ptrace(kept_sites).full()
        return reduced_dm_via_qutip

    def get_partial_trace_first(self, rho, n):
        """
        calculates partial trace of first n sites
        :param rho: full density matrix
        :param n: block size
        :return: reduced density matrix
        """
        rho_ = rho.reshape((2 ** n, 2 ** (self.N - n), 2 ** n, 2 ** (self.N - n)))
        return np.einsum('jiki->jk', rho_)

    def get_keep_indices(self, n):
        """
        Determines the middle indices for lattice sites numbered from 0 to N-1. Picks left indices
    more favourably.
        :return: List of complement of n consecutive indices
        """
        left_center = n // 2
        right_center = n - left_center
        middle = self.N // 2
        sites = np.arange(self.N)
        return sites[middle - left_center:middle + right_center].tolist()


if __name__ == "__main__":
    Ns = [8] # up to date: 9, 10, 11
    n_max = 6
    Ws = [0.5, 8.0]  # 0.5 => ergodic/delocalized phase, 8.0 localized phase
    repetitions = 100
    generate_training_set(Ns, Ws, n_max, repetitions)
```

3. Exact diagonalization

```
import numpy as np
from scipy import sparse

Id = sparse.csr_matrix(np.eye(2))
Sx = (1/2)*sparse.csr_matrix([[0., 1.], [1., 0.]])
Sy = (1/2)*sparse.csr_matrix([[0., -1.j], [1.j, 0.]])
Sz = (1/2)*sparse.csr_matrix([[1., 0.], [0., -1.]])
Splus = sparse.csr_matrix([[0., 1.], [0., 0.]])
Sminus = sparse.csr_matrix([[0., 0.], [1., 0.]])


def singlesite_to_full(op, i, L):
    op_list = [Id]*L  # = [Id, Id, Id ...] with L entries
    op_list[i] = op
    full = op_list[0]
    for op_i in op_list[1:]:
        full = sparse.kron(full, op_i, format="csr")
    return full


```

```python
def gen_sx_list(L):
    return [singlesite_to_full(Sx, i, L) for i in range(L)]

def gen_sy_list(L):
    return [singlesite_to_full(Sy, i, L) for i in range(L)]

def gen_sz_list(L):
    return [singlesite_to_full(Sz, i, L) for i in range(L)]


def gen_hamiltonian_periodic(sx_list, sz_list, g, J=1.):
    """ assumes periodic boundery conditions """
    L = len(sx_list)
    H = sparse.csr_matrix((2**L, 2**L))
    for j in range(L):
        H = H - J *( sx_list[j] * sx_list[(j+1)%L])
        H = H - g * sz_list[j]
    return H


def gen_hamiltonian_lists(L, hs, J):
    sx_list = gen_sx_list(L)
    sy_list = gen_sy_list(L)
    sz_list = gen_sz_list(L)
    H = sparse.csr_matrix((2 ** L, 2 ** L))
    H = H + J*(sx_list[0] * sx_list[1] + sy_list[0] * sy_list[1] + sz_list[0] * sz_list[1]) - hs
    [0]*sz_list[0]
    for i in range(1, L-1):
        H += + J*(sx_list[i] * sx_list[i+1] + sy_list[i] * sy_list[i+1] + sz_list[i] * sz_list[i
    +1]) - hs[i]*sz_list[i]
    return H
```

4. Dataset Preparation

```python
import matplotlib.pyplot as plt
from operator import itemgetter
import numpy as np
import pickle


def load_pickle(filename, to_numeric=1):
    with open(filename, 'rb') as f:
        data = pickle.load(f)
    return data


def save_pickle(filename, data):
    with open(filename, 'wb') as f:
        pickle.dump(data, f)


def preprocess_training_data(path):  # reduced_rho, W, self.N, n, E
    data = load_pickle(path)
    X = data
    X = [item[0] for item in X]
    X = np.reshape(X, (np.shape(X)[0], np.shape(X)[1], np.shape(X)[2], 1))
    X = np.asarray(np.concatenate((np.real(X), np.imag(X)), axis=3))
    y = data
    y = np.reshape(np.asarray([map_target(item[1]) for item in data]), (np.shape(y)[0], 1))
    print("Number of samples:", len(X))
    return X, y


def map_target(item):
    if item == 0.5:
        return 0  # ergodic/delocalized phase
    elif item == 8.0:
        return 1  # localized phase
    else:
        print("Invalid training data.")
```

```python


def save_ground_state_figures(Ns, n_max):
    for N in Ns:
        for n in range(1, n_max+1):
            data_list = load_pickle("lanczos/training_sets/N" + str(N) + "n" + str(n) + "_Trainset"
    )
            try:
                save_groundstate_figure(get_ergodic(data_list))
                save_groundstate_figure(get_localized(data_list))
            except:
                print("Ground State Figure for N=" + str(N) + ",n=" + str(n) + " could not be
    generated")
    pass


def get_ergodic(training_set):
    ergodic = [item for item in training_set if item[1] == 0.5]  # len: repetitions
    ergodic = sorted(ergodic, key=itemgetter(4))[0]  # sort by lowest E
    return ergodic


def get_localized(training_set):
    localized = [item for item in training_set if item[1] == 8.0]  # len: repetitions
    localized = sorted(localized, key=itemgetter(4))[0]  # sort by lowest E
    return localized


def save_groundstate_figure(sample):  # reduced_rho, W, self.N, n, E, rep
    """
    Plots a heatmap to the lowest groundstate of a specified system and block size.

    :param N: system size
    :param training_set: tra
    :param n: block size
    :return:
    """
    fig, ax1 = plt.subplots()
    pos = ax1.imshow(np.real(sample[0]), cmap='Purples')
    fig.colorbar(pos, ax=ax1)
    plt.title("Reduced density matrix for $n=$" + str(sample[3]) + " consecutive sites \n at $E=$"
            + str(round(sample[4], 2)) + " for $W=$" + str(sample[1]) + ", $N = $" + str(sample
    [2]))
    plt.savefig(
        "results/groundstates/N" + str(sample[2]) + "n" + str(sample[3]) + "
    _trainingset_groundstate_Wmax" + str(
            sample[1]) + ".pdf")
    plt.close()
    pass


if __name__ == "__main__":
    Ns = [8, 9, 10, 11, 12]
    n_max = 6
    save_ground_state_figures(Ns, n_max)
```

5. Model Training

```python
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, models, callbacks
import numpy as np
import matplotlib.pyplot as plt
import time
from dataset_preparation import preprocess_training_data
from tqdm import trange
from numpy import genfromtxt
from mpl_toolkits.axes_grid1 import make_axes_locatable


class ModelTrainer:

    def __init__(self, x, y, N, n):
        self.N = N
        self.n = n
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(x, y, test_size
    =0.3, random_state=42)
        self.model = self.generate_model_sparse()

    def generate_model_sparse(self):
        model = models.Sequential()
        if self.n > 3:
            filters = self.n*self.n
            model.add(layers.Conv2D(filters, (3, 3), activation='relu', input_shape=(np.shape(self.
    X_train[0])[0], np.shape(self.X_train[0])[1], 2)))
            model.add(layers.MaxPooling2D((4, 4)))
            model.add(layers.Dropout(rate=0.1))
            model.add(layers.Flatten())
        else:
            model.add(layers.Flatten(input_shape=(np.shape(self.X_train)[1], np.shape(self.X_train)
    [1], 2)))
            model.add(layers.Dropout(rate=0.1))
            model.add(layers.Dense(32, activation='relu')),

        model.add(layers.Dropout(rate=0.1))
        model.add(layers.Dense(32, activation='relu'))
        model.add(layers.Dense(1, activation='sigmoid'))
        model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
        return model

    def score(self):
        score = self.model.evaluate(self.X_test, self.y_test, verbose=0)
        print("test loss:"+"{:.3E}".format(score[0])+", test acc:" + "{:.0%}".format(score[1]))
        pass

    def fit_model(self, batch_size, epochs):
        csv_logger = callbacks.CSVLogger("lanczos/models/N"+str(self.N)+"n"+str(self.n)+"
    _model_loss.csv",
                                         separator=",",
                                         append=False)
        history = self.model.fit(self.X_train, self.y_train,
                            batch_size=batch_size,
                            epochs=epochs,
                            verbose=0,#2
                            validation_data=(self.X_test, self.y_test),
                            callbacks=[csv_logger]
                            )
        return history

    def save_model(self, filepath):
        self.model.save(filepath)

    def training_history(self, history, n, N):

        fig, ax1 = plt.subplots()
        plt.title('Model accuracy and loss for $n=$'+str(n)+', $N=$'+str(N))
        plt.xlabel('Training epoch')

        # "Loss"
        ax1.set_ylabel('Accuracy')  # we already handled the x-label with ax1
        ax1.tick_params(axis='y')
        ln1 = ax1.plot(history.history['acc'], 'r', label='Training set accuracy')
        ln2 = ax1.plot(history.history['val_acc'], 'g', label='Validation set accuracy')


        #  "Accuracy"
        ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis
        ax2.set_ylabel('Logarithmic loss')  # we already handled the x-label with ax1
        ax2.set_yscale('log')
        ax2.tick_params(axis='y')
        ln3 = ax2.plot(history.history['loss'], label='Training set loss')
        ln4 = ax2.plot(history.history['val_loss'], label='Validation set loss')


        # Joined Legend
        lns = ln1 + ln2 + ln3 + ln4
        labs = [l.get_label() for l in lns]
        ax1.legend(lns, labs, loc="center right")

        plt.tight_layout()
        plt.savefig("results/accuracy_loss_epochs/N"+str(self.N)+"n"+str(n)+"_accuracy_loss_epochs.pdf")
        print("Scores for N=" + str(N) + ", n=" + str(n))
        plt.close()
        self.score()
        pass

def train_save_model(Ns, n_max, batch_size, epochs):
    start_time = time.time()
    for N in Ns:
        start_model_time = time.time()
        for n in trange(1, n_max+1):
            X, y = preprocess_training_data(str("lanczos/training_sets/N"+str(N)+"n"+str(n)+"_Trainset"))
            model_trainer = ModelTrainer(X, y, N, n)
            history = model_trainer.fit_model(batch_size=batch_size,
                                              epochs=epochs)
            model_trainer.training_history(history, n, N)
            model_trainer.save_model("lanczos/models/N"+str(N)+"n"+str(n)+"_Model")
        print("--- Model trainings for N=" + str(N) + " lasted %s seconds ---" % (
                        time.time() - start_model_time))
    print("--- Model training lasted %s seconds ---" % (time.time() - start_time))
    pass

def get_metric(metric, Ns, n_max):
    """
    :param metric: 0:epoch, 1:acc, 2:loss, 3:val_acc, 4:val_loss
    :return: metric values per system and block size
    """
    values = np.zeros((len(Ns), n_max))
    for N in range(0, len(Ns)):
        for n in range(0, n_max):
            path = "lanczos/models/N" + str(min(Ns) + N) + "n" + str(n + 1) + "_model_loss.csv"
            my_data = genfromtxt(path, delimiter=',')
            values[N, n] = float(my_data[-1, metric])  # val loss 4
    return values

def plot_model_losses(Ns, n_max):
    titles = ["Epochs", "Training accuracy", "Training losses", "Validation accuracy", "Validation losses"]
    for train_val in [1, 2, 3, 4]:
        losses = get_metric(train_val, Ns, n_max)
        ns = np.arange(1, n_max+1, 1)
        fig, ax = plt.subplots()
        im = ax.imshow(losses, cmap='Purples_r') #fixme plot again
        ax.set_xticks(np.arange(len(ns)))
        ax.set_yticks(np.arange(len(Ns)))
        ax.set_xticklabels(ns)
        ax.set_yticklabels(Ns)
        for i in range(len(Ns)):
            for j in range(len(ns)):
                if train_val == 1 or train_val == 3:
                    text = ax.text(j, i, "{:.0%}".format(losses[i, j]),
                                   ha="center", va="center", color="k")
                else:
                    text = ax.text(j, i, "{0:.2E}".format(losses[i, j]),
                               ha="center", va="center", color="k")

        ax.set_title(titles[train_val])
        plt.xlabel("Block size")
        plt.ylabel("System size")
        # colorbar matches figure height
        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size="5%", pad=0.05)
        plt.colorbar(im, cax=cax)
        fig.tight_layout()
        plt.savefig("results/accuracy_loss_epochs/all_"+titles[train_val].lower().replace(' ', '_')
    +".pdf")
    pass


if __name__ == "__main__":
    Ns = [8, 9, 10, 11, 12]
    n_max = 6
    train_save_model(Ns, n_max,
                     batch_size=70,
                     epochs=100)
    plot_model_losses(Ns, n_max)
```

6. Test set generation

```
from generate_training_set import TrainingSetGenerator, save_pickle
import numpy as np
import time

def generate_test_set(Ns, Ws, n_max, repetitions):
    start_time = time.time()
    for N in Ns:
        training_set_generator = TrainingSetGenerator(N, Ws, n_max, repetitions)
        print("Testing Set N=" + str(N) + " completed after %s seconds." % (time.time() -
    start_time))
        for n in range(1, n_max+1):
            save_pickle("lanczos/test_sets/N"+str(N)+"n"+str(n)+"_Testset", training_set_generator.
    training_set[n])
    print("--- Testing set generation lasted %s seconds ---" % (time.time() - start_time))
    pass


if __name__ == "__main__":
    Ns = [8, 9, 10, 11, 12]
    Ws = np.arange(0., 8.0, 0.5)
    repetitions = 10
    n_max = 6
    generate_test_set(Ns, Ws, n_max, repetitions)
```

7. Prediction

```
from dataset_preparation import load_pickle, save_pickle
import numpy as np
from model_save_train import models


def preprocess_test_data(path):
    """
    :param path: Path to pickled test_set
    :return: X: reduced density matrices, W: Disorder strength that was used for generating the
    sample
    """
    print("Accessing ", path)
    data = load_pickle(path)
    X = [item[0] for item in data]
    # print("Input shape (Ws, Imagedim1, Imagedim2): ", np.shape(X))
    X = np.reshape(X, (np.shape(X)[0], np.shape(X)[1], np.shape(X)[2], 1))
    X = np.asarray(np.concatenate((np.real(X), np.imag(X)), axis=3))
    W = np.reshape(np.asarray([item[1] for item in data]), (np.shape(data)[0], 1))
    return X, W


def load_model(path):
    return models.load_model(path)
```

```python


def generate_predictions(Ns, ns, Ws):
    """
    saves predictions into lanczos/avg_prediction_sets

    :param Ns: system sizes for Model and Testset
    :param ns: block sizes for Model and Testset
    :param Ws: chosen interval for fitting
    """
    for N in Ns:
        for n in ns:
            model = load_model('lanczos/models/N' + str(N) + 'n' + str(n) + '_Model')
            X, W = preprocess_test_data('lanczos/test_sets/N' + str(N) + 'n' + str(n) + '_Testset')
            state_prediction = model.predict(X)
            state_prediction_w = [list(x) for x in zip(state_prediction, W)]
            prediction_set = []
            for W in Ws:
                average = np.mean([item[0] for item in state_prediction_w if item[1] == W])
                prediction_set.append([average, W])
            print(prediction_set)
            save_pickle("lanczos/avg_prediction_sets/N" + str(N) + "n" + str(n) + "_prediction_set"
    ,
                        prediction_set)
    pass


if __name__ == "__main__":
    Ns = [8, 9, 10, 11]
    Ws = np.arange(0., 8.0, 0.5)
    ns = np.arange(1, 6+1, 1)
    print(ns)
    generate_predictions(Ns, ns, Ws)
```

8. Evaluation of W_c

```python
from dataset_preparation import load_pickle
from model_save_train import *
import matplotlib.pyplot as plt
import matplotlib.colors as col
from scipy.optimize import curve_fit


class WcPlotter:

    def __init__(self, Ns, ns, Ws):
        self.Ws = Ws
        self.Ns = Ns
        self.ns = ns
        self.predictions = self.get_prediction()

    def plot_all(self):
        for N in self.Ns:
            print(N)
            array = self.predictions[N - min(self.Ns), :]
            wcs = self.get_wc(array)
            title = str("Predicted phases and critical disorder strength $W_c$ "
                        + "\n over block size $n$ at system size $N=$" + str(N))
            self.plot_heat_map(wcs, array, "Disorder strength $W$", "Block size n", np.asarray(self
    .ns), title)
            plt.savefig('results/Wc/N' + str(N) + '_Wc_n_dependency.pdf')
            plt.close()
        for n in self.ns:
            print(n)
            array = self.predictions[:, n - min(self.ns)]
            wcs = self.get_wc(array)
            title = str("Predicted phases and critical disorder strength $W_c$ "
                        + "\n over system size $N$ at block size $n=$" + str(n))
            self.plot_heat_map(wcs, array, "Disorder strength $W$", "System size N", np.asarray(
    self.Ns), title)
```

```python
            plt.savefig('results/Wc/n' + str(n) + '_Wc_N_dependency.pdf')
            plt.close()
        pass

    def get_wc(self, array):
        """
        Returns Ws of given array of predicted phases over system or block sizes over Ws
        """
        wcs = []
        for N_n in range(np.shape(array)[0]):
            # wcs.append(curve_fit(logistic, array[:,N_n-1], array[:,N_n-1])[0])
            nearest = np.argmin(np.abs(array[N_n - 1, :] - 0.5))#.argmin()
            print("selection:",np.abs(array[N_n - 1, :] - 0.5),"selected the element:", nearest)
            print("plotted:",array[N_n - 1, :])
            wcs.append(nearest)
        return np.asarray(wcs)

    def get_prediction(self):
        """
        Returns all predictions as N x n array of Ws
        """
        all_predictions = np.zeros((len(self.Ns), len(self.ns), len(self.Ws)))
        for N in self.Ns:
            for n in self.ns:
                element = np.array(
                    load_pickle("lanczos/avg_prediction_sets/N" + str(N) + "n" + str(n) + "_prediction_set"))
                for i in range(len(self.Ws)):
                    all_predictions[N - min(self.Ns) - 1, n - 1, i] = float(element[:, 0][i])
        return all_predictions

    def plot_heat_map(self, wcs, array, xlabel, ylabel, yticks, title):
        fig, ax = plt.subplots()
        plt.title(title)
        plt.text(0.5, len(yticks) / 2 - 0.5, 'extended', {'color': 'w', 'fontsize': 12},
                 horizontalalignment='left',
                 verticalalignment='center',
                 rotation=90,
                 )
        plt.text(3 * len(self.Ws) / 4, len(yticks) / 2 - 0.5, 'localized', {'color': 'k', 'fontsize': 12},
                 horizontalalignment='left',
                 verticalalignment='center',
                 rotation=90,
                 )
        norm = col.Normalize(vmin=0, vmax=1)
        pos = ax.imshow(array, cmap='bwr', vmin=0.0, vmax=1.0, norm=norm)  # aspect=1, #Purples
        ax.scatter(wcs, yticks - min(yticks), s=100, c="w", marker='^', label='$W_c$', edgecolors="k") #wcs
        plt.ylabel(ylabel)
        plt.xlabel(xlabel)
        # colorbar
        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size="5%", pad=0.05)
        plt.colorbar(pos, cax=cax)
        # ticks
        ax.set_xticks(np.arange(len(self.Ws)))
        ax.set_yticks(np.arange(len(yticks)))
        ax.set_xticklabels(self.Ws)
        ax.set_yticklabels(yticks)
        ax.legend()
        plt.tight_layout()
        pass

    def find_intersection(self, array):
        p = np.polyfit(self.Ws, array, 1)
        return (0.5 - p[0])/p[1]


def logistic(x, a):
    return 1 / (1 + np.exp(-50 * (x - a)))


def heaviside(x, a):
    return 0.5 * np.sign(x - a) + 0.5


def linear(x, a):
    return a * x


def load_model(path):
    return models.load_model(path)


if __name__ == "__main__":
    Ns = [8, 9, 10, 11, 12]
    Ws = np.arange(0., 8.0, 0.5)
    ns = np.arange(1, 6 + 1, 1)
    wc_plotter = WcPlotter(Ns, ns, Ws)
    wc_plotter.plot_all()
```