

Machine Learning of Many Body Localization

Philipp Krüger
(Dated: July 23, 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 for all system sizes predicted to be around $W_c = 1.5J$ for the system sizes $L \in \{9, 10, 11, 12\}$ and block sizes $n \in [1, 7]$. Low block sizes suffered from a low accuracy in the machine learning model, whereas for higher block sizes the W_c values approached $W_c = J$.

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

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 $h_i \in [-W, W]$.

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

2. Expectations for the ground state

<https://journals.aps.org/prb/abstract/10.1103/PhysRevB.91.081103>

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 an 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 $L \rightarrow \infty$. Here, the reduced density operator of a finite subsystem converges to the equilibrium thermal distribution for $L \rightarrow \infty$. [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]

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. [2] An efficient method to find ground states is the Lanczos algorithm. [3] At first, the algorithm was numerically unstable. This issue was overcome in 1970 by Ojalvo and Newman. [4] In this study, we rely on the Lanczos algorithm for the eigen-solver.

C. Reduced Density Matrix

The usefulness of reduced density matrices has already been shown by White in 1992 with ground states of Heisenberg chains [5]. 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.

$$\rho_{AB} = |\psi_A\rangle \langle \psi_A| \otimes |\psi_B\rangle \langle \psi_B| \quad (2)$$

$$\rho_A = \text{Tr}_B(\rho_{AB}) = |\psi_A\rangle \langle \psi_A| \text{Tr}(|\psi_B\rangle \langle \psi_B|) \quad (3)$$

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. [6] However, due to RAM limitations, we will rely on reduced density matrices.

D. Artificial Neural Networks

Rosenblatt published in 1958 his concept of the probabilistic model for information storage and organization in the brain, which greatly inspired others to use those models for computation.[7] Over the course of years, they have evolved to a tool that can be used for a variety of applications including computer vision, speech recognition, medical diagnosis, playing games or even artistic painting.[8]

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.

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

For the eigenvalue solution, we use SciPy's method `eigsh` through QuTiP's method `groundstate`[9, 10]. In comparison, a naive parameter choice for `eigsh` for $N = 10$ lattice sites needed 70 s to calculate the ground state, whereas `groundstate` only took 0.7 s, by choosing an optimized parameter set for `eigsh`. Of course, `eigsh` supplies the user with k eigenvalues instead of only one, but this feature was not found to be critical for the further analysis. Therefore, `groundstate` is used throughout the program, to avoid making a non optimal parameter choice.

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. Luckily, 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.[10]

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} as $n/2$ 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 serve then 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. [11]

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 [12, 13], which cannot be generalized, as they heavily depend on feature size and target dimension. As a rule of thumb the approximation was used that each weight should be influenced by at least seven samples. Using this we get from 500 samples roughly 70 weights.

The optimizer Adam was chosen, because it is computationally efficient, has little memory requirements. [14]

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.[15] The definition for a two class cross-

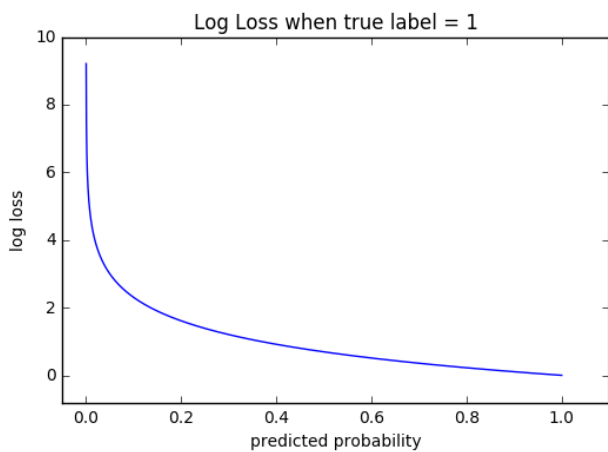


Figure 2: Cross-Entropy Loss

entropy loss can be found in equation 4, 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.

$$L(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (4)$$

D. Extraction of critical disorder strength W_c

To fit for the critical disorder strength W_c , two functions were compared. The logistic Fermi-Dirac like function:

$$L: \mathbb{R} \rightarrow [0, 1] \quad (5)$$

$$W_{pred} \mapsto \frac{1}{\exp(-\alpha(W_{pred} - W_c)) + 1} \quad (6)$$

and the heaviside function:

$$H: \mathbb{R} \rightarrow \{0, 1\} \quad (7)$$

$$W_{pred} \mapsto \begin{cases} 0 & W_{pred} < W_c \\ 1 & W_{pred} \geq W_c \end{cases} \quad (8)$$

The fully delocalized phase is defined as 0 and fully localized as 1. Whereas the heaviside function has an abrupt step and only maps to the extrema, the logistic function serves as a smoother option for a transition, depending on the parameter α . The motivation came also from an optimizers view: Differentiable functions are easier to fit for the computer.[16] Therefore, the logistic function was used to extract W_c with the empiric decision of $\alpha = 50$.

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 one minute.
2. Block size n : 500 samples, $L = 9$, $n = 8$ required 4 GB of storage for the training set, exceeding the machines performance during model fitting. Therefore, $n = 7$ was found to be sufficient for all system sizes.
3. Sample size: 500 samples can be generated for $L = 12$, $n_{max} = 7$ in approximately 9 hours. 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 five 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 were solely needed to compute the ground states of the $L = 12$ system.

Parameter	Range or Set
System size:	$L \in \{9, 10, 11, 12\}$
Block size:	$n \in \{1, 2, 3, 4, 5, 6, 7\}$
Repetitions:	$r = 500$

Table I: Parameter choice for training set generation

In order to give some visual intuition, Figure 3 shows realizations for different block sizes and phases.

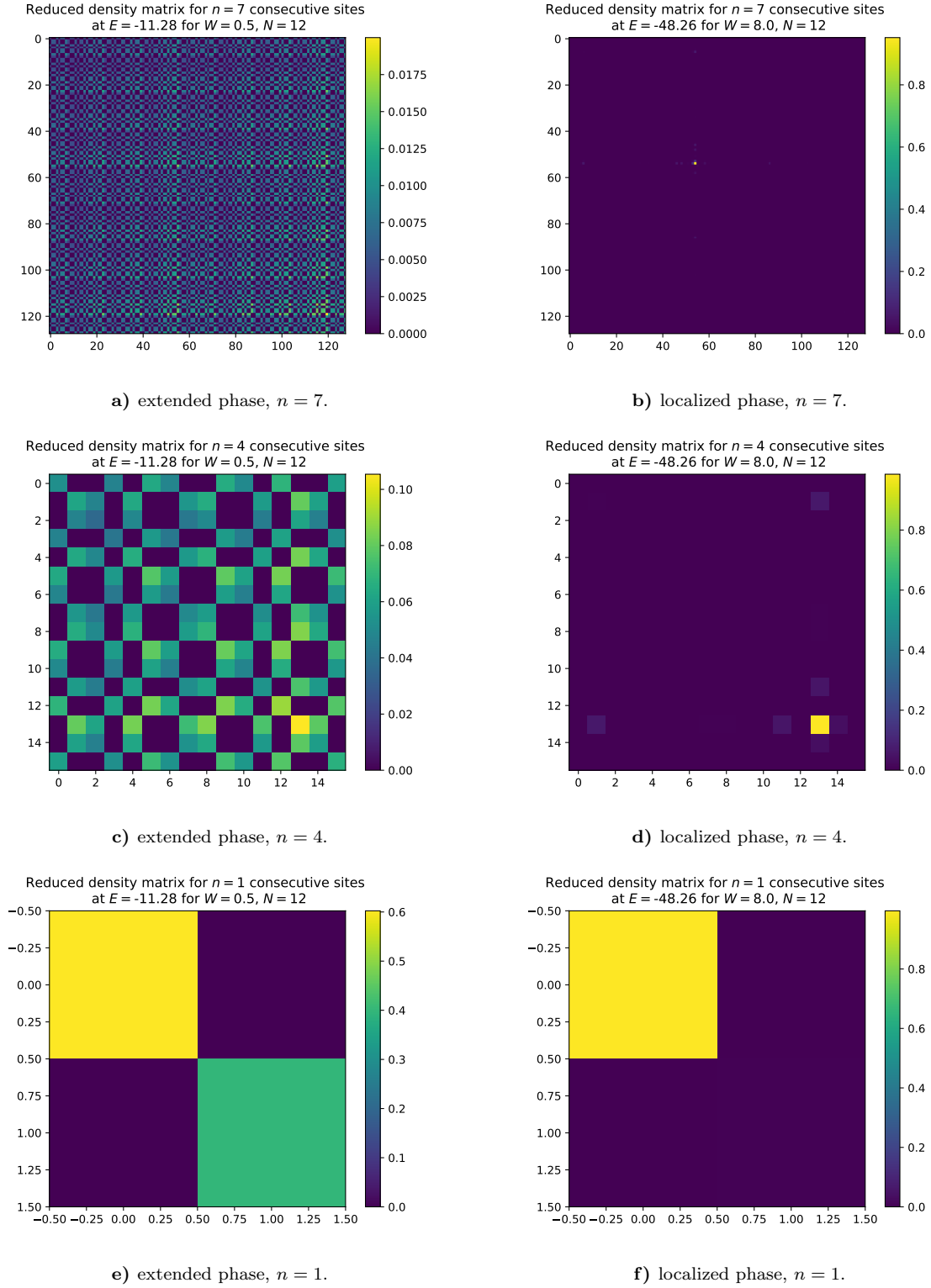


Figure 3: Real part of the density matrix of an ergodic and localized phases for block sizes $n = \{4, 5\}$ and system size $L = 12$.

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 checkerboard pat-

tern structure, which reflects that some configurations are more preferred than others. These energetically expensive states are related to neighboring unaligned spins. Another observation is that the density matrix reductions of the full ground state conserved these properties for $n > 2$, when comparing $n = 7$ to $n = \{6, 5, 4, 3, 2\}$. The similarity between the two phases gets smaller the smaller the block size n . For $n = 1$, one could argue that the density matrices are very similar, as they only deviate for half the matrix elements.

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:

```
1 model = models.Sequential()
2 model.add(layers.Flatten(input_shape=(np.shape(
    self.X_train)[1], np.shape(self.X_train)[1],
```

```
2)))
3 model.add(layers.Dense(64, activation='relu',
    bias_regularizer='l2'))
4 model.add(layers.Dense(64, activation='relu',
    bias_regularizer='l2'))
5 model.add(layers.Dense(1, activation='sigmoid'))
6 model.compile(optimizer='adam', loss='
    binary_crossentropy', metrics=['accuracy'])
```

Two strategies are employed to prevent over-fitting:

1. 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.
2. A bias regularizer was introduced to move the output functions closer to the origin. Even though some further regularization might still be possible, a kernel regularizer did not prove to be useful and impacted the resulting scores heavily.

The model training was executed by using a batch size of 70 and 200 epochs, where the batch size was limited by the CPU performance and no significant loss or accuracy improvements were noted after 120 epochs.

An example of the accuracy and loss dependency on the number of epochs for system size $L = 10$, and block sizes $n = \{1, 7\}$ is presented below in Figure .

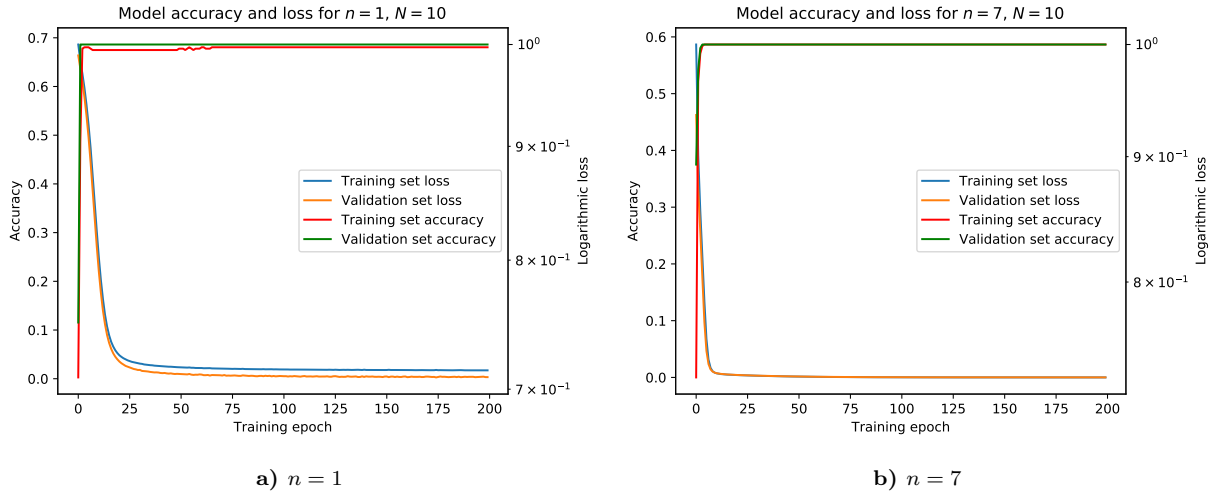


Figure 4: Loss and accuracy on training and validation set for system size $L = 10$.

The overall losses for $n > 1$ were always found to be ≈ 0.01 with accuracies of 1.0, whereas the models at $n = 1$ showed larger losses of ≈ 0.1 and mostly acceptable accuracies near 1.

Figure 4 illustrates that not only the scores for small block sizes were lower, but also the convergence rates. In conclusion, the scores show that the learning of the phases was prone to severe over-fitting and resulted in

acceptable scores for the next step of prediction, where $n = 1$ can be expected to have a worse performance during the prediction. This behavior was expected, when we noticed the similarity for $n = 1$ block size samples in the training set.

C. Analysis of critical disorder strength

1. Dependency on block size

First, the testing set was generated. Following the parameter discussion in section II E, we generate five sam-

ples for each $W \in [0, 4]$, with step $\Delta W = 0.05$, resulting in 400 samples per system and block size. Afterwards, the predictions were fitted with a logistic function to obtain W_c as described in section II D. Five predicted phases are averaged at each point and plotted to a heat map. The fitted W_c is plotted along in Figure 5.

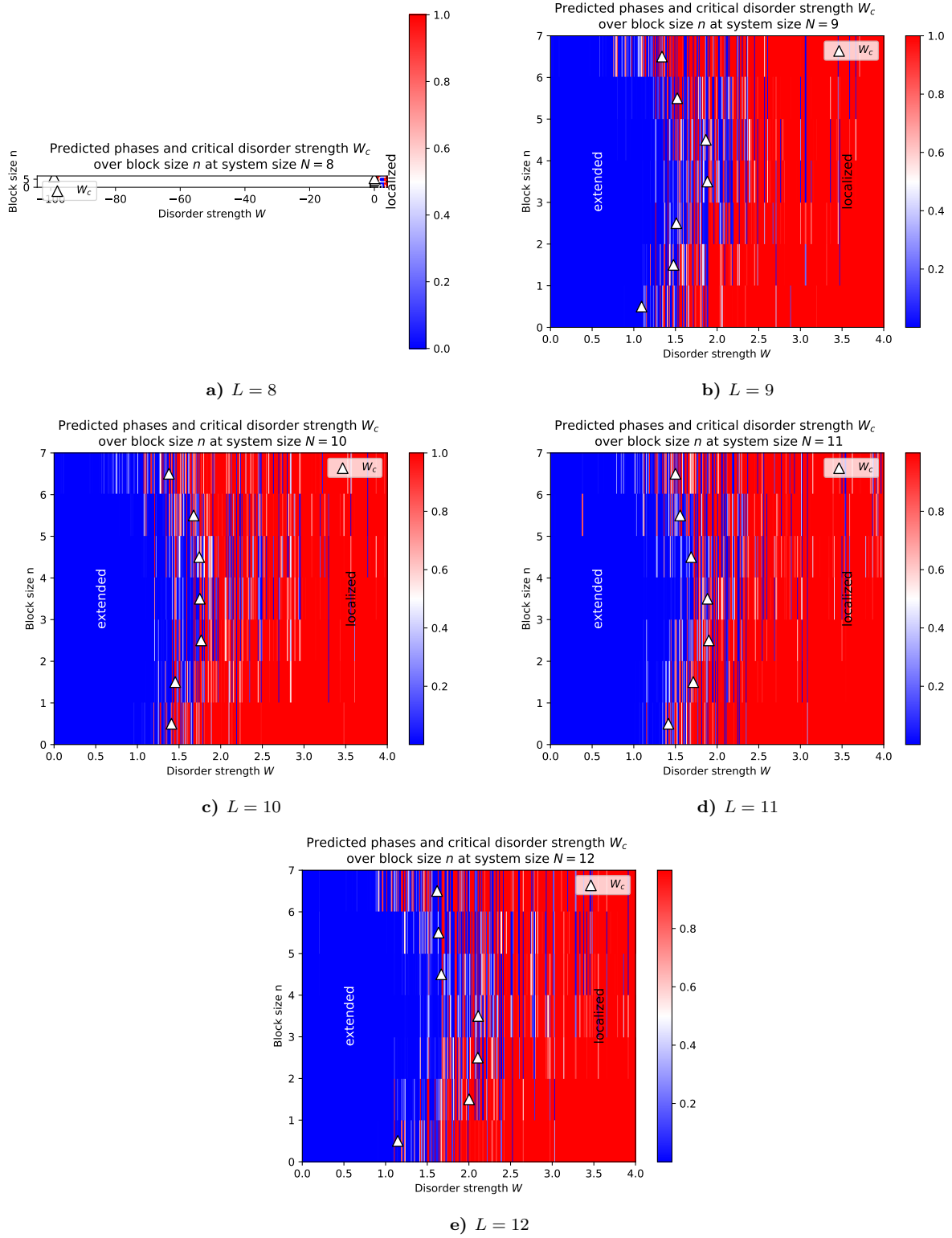


Figure 5: Dependency of the phase transition on block size n for different system sizes.

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

used for prediction. The low W_c values for $n = \{1, 2\}$ might just as well be attributed to the poor loss and ac-

curacy values shown in section III B. An explanation for this decay might be that a bigger block size can more accurately reflect the level of disorder forced on the system. For smaller block sizes, for some spins the information is lost whether the configuration was the result of interacting lattice sites or the random disorder strength.

2. *Dependency on system size*

something something

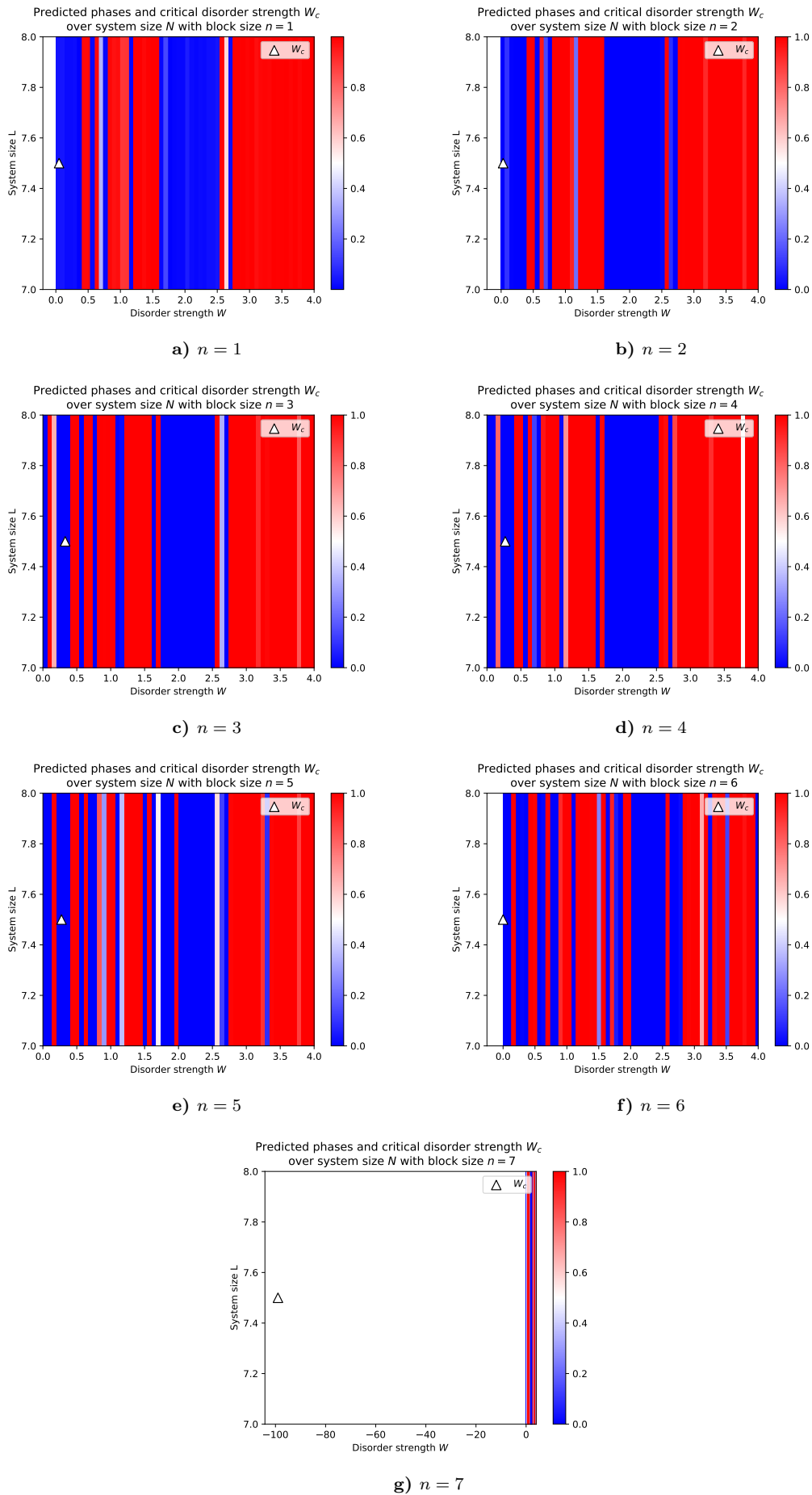


Figure 6: Dependency of the phase transition over system sizes L for different block sizes n .

The plots are indicating that a bigger system size requires a larger disorder strength to perform the phase transition.

IV. CONCLUSION

W_c depends on n , L (yes/no).

W_c prediction coincides with the expectation (yes/no)

W_c is dependent on these and that effects = scaling

analysis? (yes/no)

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

-
- [1] A. Pal and D. A. Huse, Many-body localization phase transition, *Phys. Rev. B* **82**, 174411 (2010).
 - [2] 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.
 - [3] 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).
 - [4] I. U. Ojalvo and M. Newman, Vibration modes of large structures by an automatic matrix-reduction method, *AIAA Journal* **8**, 1234 (1970).
 - [5] S. R. White, Density matrix formulation for quantum renormalization groups, *Physical Review Letters* **69**, 2863 (1992).
 - [6] 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).
 - [7] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain., *Psychological Review* **65**, 386 (1958).
 - [8] L. A. Gatys, A. S. Ecker, and M. Bethge, A neural algorithm of artistic style, *CoRR* **abs/1508.06576** (2015), arXiv:1508.06576.
 - [9] 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).
 - [10] 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).
 - [11] F. Chollet *et al.*, Keras, <https://keras.io> (2015).
 - [12] 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.
 - [13] S. Feng, H. Zhou, and H. Dong, Using deep neural network with small dataset to predict material defects, *Materials & Design* **162**, 300 (2019).
 - [14] D. Kingma and J. Ba, Adam: A method for stochastic optimization, *International Conference on Learning Representations* (2014).
 - [15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) <http://www.deeplearningbook.org>.
 - [16] N. Kyurkchiev and S. Markov, *Sigmoid Functions Some Approximation and Modelling Aspects: Some Moduli in Programming Environment MATHEMATICA* (2015).

Appendix A: Code listing

The code consists essentially of five different files, which are callable through a main function, but can also be run separately. Every file serves a number of different purposes as listed below.

1. **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.
2. **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.
3. **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.
4. **generate_test_set.py**: A set of reduced density matrices for ground states in the intermediate regime is generated.
5. **load_model_get_wc.py**: The models for each system and block size make phase predictions to the respective test sets, extract W_c and plot everything together as a heat map.

1. Training set generation

```

1 from ed import *
2 import time
3 import pickle
4 # import qutip
5 import matplotlib.pyplot as plt
6 # from tqdm import trange, tqdm
7 from scipy.sparse.linalg import eigsh, ArpackNoConvergence
8 from operator import itemgetter
9
10
11 def generate_training_set(Ns, Ws, n_max, repetitions):
12     start_time = time.time()
13     for N in Ns:
14         training_set_generator = TrainingSetGenerator(N, Ws, n_max, repetitions)
15         print("Training Set N=" + str(N) + " completed after %s seconds." % (time.time() -
16 start_time))
17         for n in range(1, n_max + 1):
18             # save_groundstate_figures(N, training_set_generator.training_set[n], n)
19             save_pickle("lanczos/training_sets/N" + str(N) + "n" + str(n) + "_Trainset",
20 training_set_generator.training_set[n])
21         print("--- Training set generation lasted %s seconds ---" % (time.time() - start_time))
22         pass
23
24 def save_groundstate_figures(N, training_set, n): # reduced_rho, W, self.N, n, E, rep
25     """
26     Plots a heatmap to the lowest groundstate of a specified system and block size.
27
28     :param N: system size
29     :param training_set: tra
30     :param n:
31     :return:
32     """
33     ergodic = [item for item in training_set if item[1] == 0.5 and item[-1] == 0] # len:
34     repetitions
35     localized = [item for item in training_set if item[1] == 8.0 and item[-1] == 0] # len:
36     repetitions
37
38     ergodic = sorted(ergodic, key=itemgetter(4))[0] # sort by lowest E
39     localized = sorted(localized, key=itemgetter(4))[0] # sort by lowest E

```

```

39 fig, ax1 = plt.subplots()
40 pos = ax1.imshow(np.real(ergodic[0]), cmap='bwr')
41 fig.colorbar(pos, ax=ax1)
42 plt.title("Reduced density matrix for $n=$" + str(n) + " consecutive sites \n at $E=$"
43           + str(round(ergodic[4], 2)) + " for $W=$" + str(ergodic[1]) + ", $N = $" + str(N))
44 plt.savefig(
45     "results/groundstates/N" + str(N) + ".n" + str(n) + "_trainingset_groundstate_Wmax" + str(
46     ergodic[1]) + ".pdf")
47 plt.close()
48
49 fig, ax1 = plt.subplots()
50 pos = ax1.imshow(np.real(localized[0]), cmap='bwr')
51 fig.colorbar(pos, ax=ax1)
52 plt.title("Reduced density matrix for $n=$" + str(localized[3]) + " consecutive sites \n at $E=
53           + str(round(localized[4], 2)) + " for $W=$" + str(localized[1]) + ", $N = $" + str(N)
54           )
55 plt.savefig(
56     "results/groundstates/N" + str(N) + ".n" + str(localized[3]) + "
57     _trainingset_groundstate_Wmax" + str(
58     localized[1]) + ".pdf")
59 plt.close()
60 pass
61
62 def save_pickle(filename, data):
63     with open(filename, 'wb') as f:
64         pickle.dump(data, f)
65
66 class TrainingSetGenerator:
67
68     def __init__(self, N, Ws, n_max, repetitions):
69         self.N = int(N)
70         self.n_max = n_max
71         self.repetitions = repetitions
72         self.Ws = Ws
73         self.training_set = self.generate_training_set_m_lanczos_list() # self.
74         generate_training_set_list()
75
76     def generate_training_set_m_lanczos_list(self):
77         """
78         Returns training set with shape samples x [density matrix, W, lattice sites, block size,
79         ground state energy]
80         :return: training set
81         """
82         training_set = {consecutive_spins: [] for consecutive_spins in range(1, self.n_max + 1)}
83         for rep in range(self.repetitions):
84             for W in self.Ws:
85                 h = np.random.uniform(-W, W, size=self.N)
86                 H = gen_hamiltonian_lists(self.N, h, J=1)
87                 try:
88                     Es, vs = eigsh(H, k=6, sigma=0, which='SM', tol=0.01)
89                     # sigma=0 for shift invert mode Eigval near to zero
90                     # following the advice of https://docs.scipy.org/doc/scipy/reference/tutorial/
91                     arpack.html
92                 except ArpackNoConvergence as err:
93                     Es = err.eigenvalues
94                     vs = err.eigenvectors
95                     for i in range(len(Es)):
96                         rho = np.outer(vs[:,i], vs[:,i])
97                         for n in range(1, self.n_max + 1):
98                             reduced_rho = self.get_partial_trace_first(rho, n) # fixme
99                             # if Es[i]:
100                             training_set[n].append([reduced_rho, W, self.N, n, Es[i], rep])
101                             # else:
102                             #     print("Diagonalization did not converge.")
103         print(np.shape(training_set[n]), np.shape(training_set[n][0]))
104         return training_set

```

```

102 # def get_partial_trace_mid(self, rho, n):
103 #     """
104 #     calculates partial trace of middle n sites
105 #     :param rho: full density matrix
106 #     :param n: block size
107 #     :return: reduced density matrix
108 #     """
109 #     kept_sites = self.get_keep_indices(n)
110 #     qutip_dm = qutip.Qobj(rho, dims=[[2] * self.N] * 2)
111 #     reduced_dm_via_qutip = qutip_dm.ptrace(kept_sites).full()
112 #     return reduced_dm_via_qutip
113
114 def get_partial_trace_first(self, rho, n):
115     """
116     calculates partial trace of first n sites
117     :param rho: full density matrix
118     :param n: block size
119     :return: reduced density matrix
120     """
121     rho_ = rho.reshape((2 ** n, 2 ** (self.N - n), 2 ** n, 2 ** (self.N - n)))
122     return np.einsum('jiki->jk', rho_)
123
124 def diff(self, first, second):
125     second = set(second)
126     return [item for item in first if item not in second]
127
128 def get_keep_indices(self, n):
129     """
130     Determines the middle indices for lattice sites numbered from 0 to N-1. Picks left indices
131     more favourably.
132     :return: List of complement of n consecutive indices
133     """
134     left_center = n // 2
135     right_center = n - left_center
136     middle = self.N // 2
137     sites = np.arange(self.N)
138     return sites[middle - left_center:middle + right_center].tolist()
139
140 if __name__ == "__main__":
141     Ns = [12]
142     n_max = 6
143     Ws = [0.5, 8.0] # 0.5 => ergodic/delocalized phase, 8.0 localized phase
144     repetitions = 100
145     generate_training_set(Ns, Ws, n_max, repetitions)
146
147     # N=09, n=7, rep=10 7s=> rep=500: 6 min
148     # N=10, n=7, rep=10 31s => rep=500: 25 min
149     # N=11, n=7, rep=10 182s=> rep=500: 2,5 h
150     # N=12, n=7, rep=10 00s=> rep=500
151
152     # N=8, n=7, rep=100, first, 26s
153     # N=8, n=7, rep=100, mid, 32s

```

2. Model Training

```

1 from sklearn.model_selection import train_test_split
2 import pickle
3 from tensorflow.keras import layers, models
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import tensorflow.keras.backend as k
7 import time
8
9
10 def load_pickle(filename, to_numeric=1):
11     with open(filename, 'rb') as f:
12         data = pickle.load(f)
13     return data
14

```

```

15
16 def preprocess_training_data(path): # reduced_rho, W, self.N, n, E
17     data = load_pickle(path)
18     X = data
19     X = [item[0] for item in X]
20     X = np.reshape(X, (np.shape(X)[0], np.shape(X)[1], np.shape(X)[2], 1))
21     X = np.asarray(np.concatenate((np.real(X), np.imag(X)), axis=3))
22     y = data
23     y = np.reshape(np.asarray([map_target(item[1]) for item in data]), (np.shape(y)[0], 1))
24     return X, y
25
26 def map_target(item):
27     if item == 0.5:
28         return 0 # ergodic/delocalized phase
29     elif item == 8.0:
30         return 1 # localized phase
31     else:
32         print("Invalid training data.")
33
34 def mean_pred(y_true, y_pred):
35     return k.mean(y_pred)
36
37
38 class ModelTrainer:
39
40     def __init__(self, x, y, N, n_max):
41         self.N = N
42         self.n_max = n_max
43         self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(x, y, test_size
=0.3, random_state=42)
44         self.model = self.generate_model_sparse()
45
46     def generate_model(self):
47         model = models.Sequential()
48         model.add(layers.Flatten())
49         model.add(layers.Dense(64, activation='relu'))
50         model.add(layers.Dense(128, activation='relu'))
51         model.add(layers.Dense(64, activation='relu'))
52         model.add(layers.Dense(32, activation='relu'))
53         model.add(layers.Dense(1, activation='sigmoid'))
54         model.compile(optimizer='rmsprop', loss='mae', metrics=['accuracy'])#loss used to be mae
loss # metrics: 'mean_absolute_error', 'mean_squared_error',
55         return model
56
57     def generate_model_sparse(self):
58         model = models.Sequential()
59         # if self.N != 12:
60         # model.add(layers.Conv2D(32, (6, 6), activation='relu', input_shape=(np.shape(self.X_train
)[1], np.shape(self.X_train)[1], 2)))
61         # model.add(layers.MaxPooling2D((4, 4)))
62         model.add(layers.Flatten(input_shape=(np.shape(self.X_train)[1], np.shape(self.X_train)[1],
2)))
63         model.add(layers.Dense(64, activation='relu', bias_regularizer='l2')), # fixme maybe try l2
kernel reg.?
64         model.add(layers.Dense(64, activation='relu', bias_regularizer='l2')) # fixme use kernel
regularizer!! l1 loss as squared error is dangerous below 1
65         model.add(layers.Dense(1, activation='sigmoid')) # fixme drop out layer
66         model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])#loss used
to be mae loss # metrics: 'mean_absolute_error', 'mean_squared_error',
67         return model
68
69     def score(self):
70         score = self.model.evaluate(self.X_test, self.y_test, verbose=0)
71         print("test loss: %.3E, test acc: %.3E" % (score[0], score[1]))
72         pass
73
74     def fit_model(self, batch_size, epochs):
75         history = self.model.fit(self.X_train, self.y_train,
76                                 batch_size=batch_size,
77                                 epochs=epochs,

```

```

78         verbose=0,#2
79         validation_data=(self.X_test, self.y_test)
80     )
81     return history
82
83 def save_model(self, filepath):
84     self.model.save(filepath)
85
86 def training_history(self, history, n, N):
87
88     fig, ax1 = plt.subplots()
89     plt.title('Model accuracy and loss for $n=$'+str(n)+'', $N=$'+str(N))
90     plt.xlabel('Training epoch')
91
92     # "Loss"
93     ax1.set_ylabel('Accuracy') # we already handled the x-label with ax1
94     ax1.tick_params(axis='y')
95
96     ln1 = ax1.plot(history.history['loss'], label='Training set loss')
97     ln2 = ax1.plot(history.history['val_loss'], label='Validation set loss')
98
99     # "Accuracy"
100    ax2 = ax1.twinx() # instantiate a second axes that shares the same x-axis
101    ax2.set_ylabel('Logarithmic loss') # we already handled the x-label with ax1
102    ax2.set_yscale('log')
103    ax2.tick_params(axis='y')
104    ln3 = ax2.plot(history.history['acc'], 'r', label='Training set accuracy')
105    ln4 = ax2.plot(history.history['val_acc'], 'g', label='Validation set accuracy')
106
107    # Joined Legend
108    lns = ln1 + ln2 + ln3 + ln4
109    labs = [l.get_label() for l in lns]
110    ax1.legend(lns, labs, loc="center right")
111
112    plt.tight_layout()
113    plt.savefig("results/accuracy_loss_epochs/N"+str(self.N)+"n"+str(n)+"_accuracy_loss_epochs.
pdf")
114    print("Scores for N=" + str(N) + ", n=" + str(n))
115    plt.close()
116    self.score()
117    pass
118
119 def train_save_model(Ns, n_max, batch_size, epochs):
120     start_time = time.time()
121     for N in Ns:
122         start_model_time = time.time()
123         for n in range(1, n_max+1):
124             X, y = preprocess_training_data("lanczos/training_sets/N"+str(N)+"n"+str(n)+"_Trainset"
)
125             model_trainer = ModelTrainer(X, y, N, n_max)
126             history = model_trainer.fit_model(batch_size=batch_size,
epochs=epochs)
127             model_trainer.training_history(history, n, N)
128             model_trainer.save_model("lanczos/models/N"+str(N)+"n"+str(n)+"_Model")
129             print("--- Model trainings for N=" + str(N) + " lasted %s seconds ---" % (
time.time() - start_model_time))
130         print("--- Model training lasted %s seconds ---" % (time.time() - start_time))
131     pass
132
133
134
135
136 if __name__ == "__main__":
137     Ns = [9, 10]
138     n_max = 7
139     train_save_model(Ns, n_max,
batch_size=70,
epochs=40)

```

3. Test set generation

```

1 from generate_training_set import TrainingSetGenerator, save_pickle

```

```

2 from model_save_train import *
3 import time
4
5 def generate_test_set(Ns, Ws, n_max, repetitions):
6     start_time = time.time()
7     for N in Ns:
8         training_set_generator = TrainingSetGenerator(N, Ws, n_max, repetitions)
9         print("Testing Set N=" + str(N) + " completed after %s seconds." % (time.time() -
10 start_time))
11         for n in range(1, n_max+1):
12             save_pickle("lanczos/test_sets/N"+str(N)+"n"+str(n)+"_Testset", training_set_generator.
13 training_set[n])
14         print("--- Testing set generation lasted %s seconds ---" % (time.time() - start_time))
15         pass
16
17 if __name__ == "__main__":
18     Ns = [9, 10, 11]
19     Ws = np.arange(0., 8.0, 0.5)
20     repetitions = 20
21     n_max = 7
22     generate_test_set(Ns, Ws, n_max, repetitions)

```

4. Prediction and evaluation of W_c

```

1 from model_save_train import *
2 from scipy.optimize import curve_fit
3
4
5 def preprocess_test_data(path):
6     """
7     :param path: Path to pickled test_set
8     :return: X: reduced density matrices, W: Disorder strength that was used for generating the
9     sample
10     """
11     print("Accessing ", path)
12     data = load_pickle(path)
13     X = [item[0] for item in data]
14     # print("Input shape (Ws, Imagedim1, Imagedim2): ", np.shape(X))
15     X = np.reshape(X, (np.shape(X)[0], np.shape(X)[1], np.shape(X)[2], 1))
16     X = np.asarray(np.concatenate((np.real(X), np.imag(X)), axis=3))
17     W = np.reshape(np.asarray([item[1] for item in data]), (np.shape(data)[0], 1))
18     return X, W
19
20 def logistic(x, a):
21     return 1 / (1 + np.exp(-50 * (x - a)))
22
23
24 def heaviside(x, a):
25     return 0.5 * np.sign(x - a) + 0.5
26
27
28 def load_model(path):
29     return models.load_model(path)
30
31
32 def get_wc(N, n, Ws, repetitions):
33     """
34     Calculates  $W_c$ 
35
36     :param N: system size for Model and Testset
37     :param n: block size for Model and Testset
38     :param Ws: chosen interval for fitting
39     :param repetitions: Number of datapoints per  $W_{pred}$ 
40     :return:  $W_c$ 
41     """
42     model = load_model('lanczos/models/N' + str(N) + 'n' + str(n) + '_Model')
43     X, W = preprocess_test_data('lanczos/test_sets/N' + str(N) + 'n' + str(n) + '_Testset')
44     print(W)

```



```

45 state_prediction = model.predict(X)
46
47 state_prediction_w = [list(x) for x in zip(state_prediction, W)]
48 state_prediction_w_averaged = []
49 for W in Ws:
50     item = [np.mean([item[0] for item in state_prediction_w if item[1] == W]), W]
51     if not np.isnan(item[0]):
52         state_prediction_w_averaged.append(item)
53 print(state_prediction_w_averaged)
54 state_prediction_w_averaged = np.asarray(state_prediction_w_averaged)
55 print(state_prediction_w_averaged)
56 popt, pcov = curve_fit(logistic, state_prediction_w_averaged[:,0], state_prediction_w_averaged
57                       [:,1]) # state_prediction.astype(np.float))
58 # plot_wc_fit(N,popt,state_prediction)
59 return popt[0], state_prediction_w_averaged[:,0], state_prediction_w_averaged[:,1] # Wc, Ws,
60 predictions
61
62 class HeatMapPlotter:
63
64     def __init__(self, Ns, Ws, n_max, repetitions):
65         self.Ns = Ns
66         self.Ws = Ws
67         self.n_max = n_max
68         self.repetitions = repetitions
69
70     def predict_w_n(self):
71         W_preds = {system_size: [] for system_size in self.Ns}
72         for N in self.Ns:
73             for n in range(1, self.n_max + 1):
74                 model = load_model('lanczos/models/N' + str(N) + 'n' + str(n) + '_Model')
75                 X, W = preprocess_test_data('lanczos/test_sets/N' + str(N) + 'n' + str(n) + '
76                 _Testset')
77                 W_preds[N].append(model.predict(X))
78             return W_preds
79
80     def fit_wc_n(self):
81         W_c_fit = {system_size: [] for system_size in self.Ns}
82         for N in self.Ns:
83             for n in range(1, self.n_max + 1):
84                 W_c_fit[N].append((get_wc(N, n, self.Ws, self.repetitions), n))
85             return W_c_fit
86
87     def predict_w_N(self):
88         W_preds = {block_size: [] for block_size in range(1, self.n_max + 1)}
89         for n in range(1, self.n_max + 1):
90             for N in self.Ns:
91                 model = load_model('lanczos/models/N' + str(N) + 'n' + str(n) + '_Model')
92                 X, W = preprocess_test_data('lanczos/test_sets/N' + str(N) + 'n' + str(n) + '
93                 _Testset')
94                 W_preds[n].append(model.predict(X))
95             return W_preds
96
97     def fit_wc_N(self):
98         W_c_fit = {block_size: [] for block_size in range(1, self.n_max + 1)}
99         for n in range(1, self.n_max + 1):
100             for N in self.Ns:
101                 W_c_fit[n].append((get_wc(N, n, self.Ws, self.repetitions), N))
102             return W_c_fit
103
104     def plot_wc_heatmap_n(self):
105         """
106         Plots Heatmap with blocksize and W_pred
107
108         W_pred: W x n array
109         W_c_fit: W_c(n) x 1 array
110         """
111         self.W_preds = self.predict_w_n()
112         self.W_c_fit = self.fit_wc_n()
113         Wc, Ws, predictions =

```

```

111
112
113     for N in self.Ns:
114         W_pred = np.asarray(self.W_preds[N])
115         W_pred = np.reshape(W_pred, (np.shape(W_pred)[0], np.shape(W_pred)[1]))
116         W_c_fit = np.array(self.W_c_fit[N])
117
118         # W_c_fit = np.reshape(W_c_fit, (np.shape(W_c_fit)[0], np.shape(W_c_fit)[1]))
119         fig, ax = plt.subplots()
120         plt.title(
121             "Predicted phases and critical disorder strength  $W_c$  \n over block size  $n$  at
system size  $N$ =" + str(
122                 N))
123         plt.text(0.5, 3.5, 'extended', {'color': 'w', 'fontsize': 12},
124                 horizontalalignment='left',
125                 verticalalignment='center',
126                 rotation=90,
127                 )
128         plt.text(3.5, 3.5, 'localized', {'color': 'k', 'fontsize': 12},
129                 horizontalalignment='left',
130                 verticalalignment='center',
131                 rotation=90,
132                 )
133         pos = ax.imshow(W_pred, extent=(0, 4, 0, 7), aspect=0.5, cmap='bwr')
134         fig.colorbar(pos, ax=ax)
135         ax.scatter(W_c_fit[:, 0], W_c_fit[:, 1] - 0.5, s=100, c="w", marker='^', label='$W_c$',
edgecolors="k")
136         plt.ylabel("Block size n")
137         plt.xlabel("Disorder strength  $W$ ")
138         ax.legend()
139         plt.tight_layout()
140         plt.savefig('results/Wc/N' + str(N) + '_Wc_n_dependency.pdf')
141         plt.close()
142     pass
143
144 def plot_wc_heatmap_N(self):
145     """
146     Plots Heatmap with blocksize and W_pred
147
148     W_pred: W x n array
149     W_c_fit: W_c(n) x 1 array
150     """
151     self.W_preds = self.predict_w_N()
152     self.W_c_fit = self.fit_wc_N()
153
154     for n in range(1, self.n_max + 1):
155         W_pred = np.asarray(self.W_preds[n])
156         W_pred = np.reshape(W_pred, (np.shape(W_pred)[0], np.shape(W_pred)[1]))
157         W_c_fit = np.array(self.W_c_fit[n])
158
159         # W_c_fit = np.reshape(W_c_fit, (np.shape(W_c_fit)[0], np.shape(W_c_fit)[1]))
160         fig, ax = plt.subplots()
161         plt.title(
162             "Predicted phases and critical disorder strength  $W_c$  \n over system size  $N$  with
block size  $n$ =" + str(
163                 n))
164         plt.text(0.5, 10, 'extended', {'color': 'w', 'fontsize': 12},
165                 horizontalalignment='left',
166                 verticalalignment='center',
167                 rotation=90,
168                 )
169         plt.text(3.5, 10, 'localized', {'color': 'k', 'fontsize': 12},
170                 horizontalalignment='left',
171                 verticalalignment='center',
172                 rotation=90,
173                 )
174         pos = ax.imshow(W_pred, extent=(0, 4, self.Ns[0] - 1, self.Ns[-1]), aspect='auto', cmap
='bwr')
175         # Shift ticks to be at 0.5, 1.5, etc
176         # ax.yaxis.set(ticks=np.arange(0.5, len(self.Ns)), ticklabels=map(str, input(self.Ns)))

```

```

177         fig.colorbar(pos, ax=ax)
178         # print(W_c_fit)
179         ax.scatter(W_c_fit[:, 0], W_c_fit[:, 1] - 0.5, s=100, c="w", marker='^', label='$W_c$',
180         edgcolors="k")
181         plt.ylabel("System size L")
182         plt.xlabel("Disorder strength $W$")
183         ax.legend()
184         plt.savefig('results/Wc/n' + str(n) + '_Wc_N_dependency.pdf')
185         plt.close()
186     pass
187
188     def plot_wc_fit(self, N, popt, state_prediction):
189         fig, ax1 = plt.subplots()
190         ax1 = plt.scatter(self.Ws, state_prediction)
191         ax1 = plt.plot(self.Ws, logistic(self.Ws, *popt), 'k')
192
193         plt.title('Phase prediction $N = $' + str(N) + ", $W_c = $" + "{0:.3g}".format(popt[0]))
194         plt.ylabel('Probability of localized phase')
195         plt.xlabel('$W_{max}$')
196         plt.legend(['Logistic fit', 'Predicted phase'], loc='upper left')
197         plt.savefig('results/N' + str(N) + '_predict_wc.pdf')
198     pass
199
200
201 if __name__ == "__main__":
202     Ns = [8]
203     Ws = np.arange(0., 4.0, 0.5)
204     n_max = 7
205     repetitions = 2 * 6
206     heat_map_plotter = HeatMapPlotter(Ns, Ws, n_max, repetitions)
207     heat_map_plotter.plot_wc_heatmap_n()
208     print("done")
209     heat_map_plotter.plot_wc_heatmap_N()
210     print("done")

```

5. Exact diagonalization

```

1 import numpy as np
2 from scipy import sparse
3
4 Id = sparse.csr_matrix(np.eye(2))
5 Sx = (1/2)*sparse.csr_matrix([[0., 1.], [1., 0.]])
6 Sy = (1/2)*sparse.csr_matrix([[0., -1.j], [1.j, 0.]])
7 Sz = (1/2)*sparse.csr_matrix([[1., 0.], [0., -1.]])
8 Splus = sparse.csr_matrix([[0., 1.], [0., 0.]])
9 Sminus = sparse.csr_matrix([[0., 0.], [1., 0.]])
10
11
12 def singlesite_to_full(op, i, L):
13     op_list = [Id]*L # = [Id, Id, Id ...] with L entries
14     op_list[i] = op
15     full = op_list[0]
16     for op_i in op_list[1:]:
17         full = sparse.kron(full, op_i, format="csr")
18     return full
19
20
21 def gen_sx_list(L):
22     return [singlesite_to_full(Sx, i, L) for i in range(L)]
23
24 def gen_sy_list(L):
25     return [singlesite_to_full(Sy, i, L) for i in range(L)]
26
27 def gen_sz_list(L):
28     return [singlesite_to_full(Sz, i, L) for i in range(L)]
29
30
31 def gen_hamiltonian_periodic(sx_list, sz_list, g, J=1.):
32     """ assumes periodic boundary conditions """

```

```

33 L = len(sx_list)
34 H = sparse.csr_matrix((2**L, 2**L))
35 for j in range(L):
36     H = H - J * ( sx_list[j] * sx_list[(j+1)%L])
37     H = H - g * sz_list[j]
38 return H
39
40
41 def gen_hamiltonian(sx_list, sz_list, g, J=1.):
42     """ assumes open boundary conditions """
43     L = len(sx_list)
44     H = sparse.csr_matrix((2**L, 2**L))
45     for j in range(L-1):
46         H = H - J * ( sx_list[j] * sx_list[(j+1)%L])
47         H = H - g * sz_list[j]
48     H = H - g * sz_list[-1]
49     return H
50
51
52 def gen_hamiltonian_lists(L, h, J=1):
53     sx_list = gen_sx_list(L)
54     sy_list = gen_sy_list(L)
55     sz_list = gen_sz_list(L)
56     H = sparse.csc_matrix((2 ** L, 2 ** L))
57     H = H + J*(sx_list[0] * sx_list[1] + sy_list[0] * sy_list[1] + sz_list[0] * sz_list[1]) - h[0]*
58     sz_list[0]
59     for i in range(1, L-1):
60         H += J*(sx_list[i] * sx_list[i+1] + sy_list[i] * sy_list[i+1] + sz_list[i] * sz_list[i+1])
61         - h[i]*sz_list[i]
62     return H
63
64 # fixme delete
65
66 # def gen_hamiltonian_random_h(L, W, J=1.):
67 #     """ assumes open boundary conditions """
68 #     sx_list = gen_sx_list(L)
69 #     sz_list = gen_sz_list(L)
70 #     H = sparse.csr_matrix((2**L, 2**L))
71 #     for j in range(L-1):
72 #         H = H - J * ( sx_list[j] * sx_list[(j+1)%L])
73 #         H = H - np.random.uniform(-W, W) * sz_list[j]
74 #     H = H - np.random.uniform(-W, W) * sz_list[-1]
75 #     return H

```
