Machine Learning of Many Body Localization

Philipp Krüger

(Dated: July 21, 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 $h_i \in [-W, W]$.

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

2. *Expectations for the ground state*

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 $L \to \infty$. 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]

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 of 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 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 [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.

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

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

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. [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.

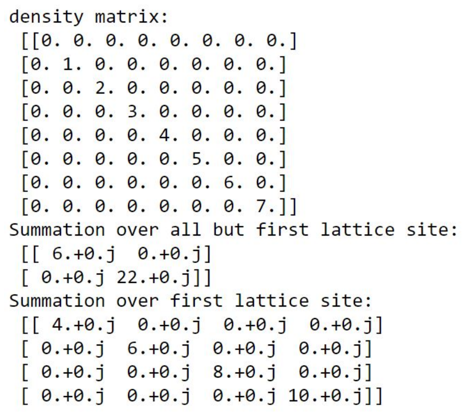
A. Eigenvalue solver

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-

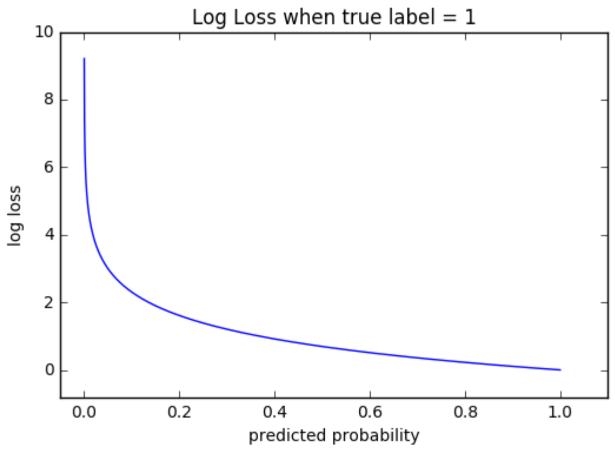


Figure 2: Cross-Entropy Loss

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.

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

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:

$$\mathrm{L}\colon\ \mathbb{R} \to [0, 1] \tag{5}$$

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

and the heaviside function:

$$\mathrm{H}\colon\ \mathbb{R} \to \{0, 1\} \tag{7}$$

$$W_{pred} \mapsto \begin{cases} 0: & W_{pred} < W_c \\ 1: & W_{pred} \geq W_c \end{cases} \tag{8}$$

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 *alpha*. 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

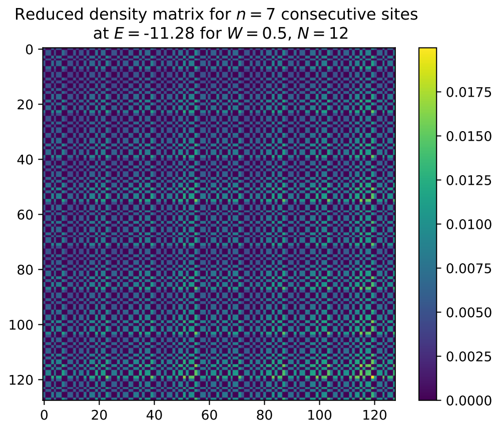
A. Generation of reduced density matrix training set

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 where solely needed to compute the ground states of the $L = 12$ system.

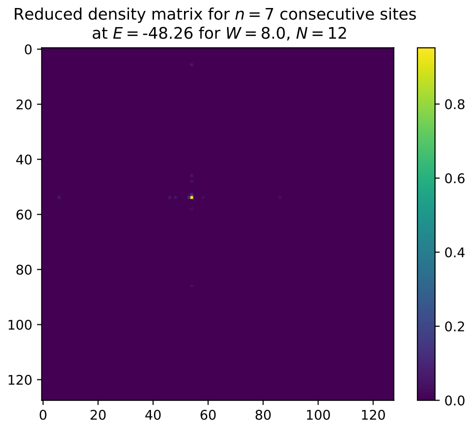
Parameter	Range or Set
System size:	$L \in \{9, 10, 11, 12\}$
Block size:	$n \in \{1, 2, 3, 4, 5, 6, 7\}$
Repetitions:	$r = 500$

Table I: Parameter choice for training set generation

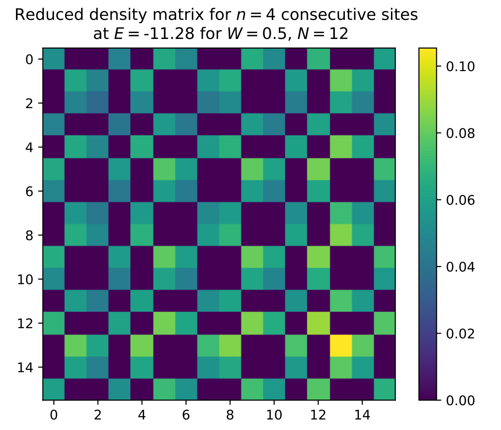
In order to give some visual intuition, Figure 3 shows realizations for different block sizes and phases.



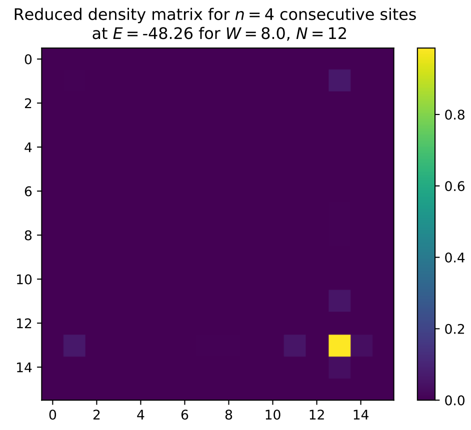
a) extended phase, $n = 7$.



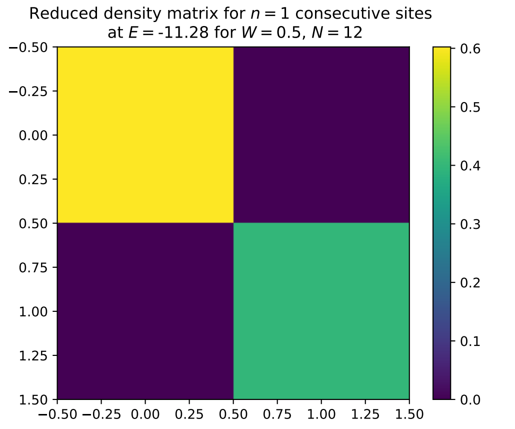
b) localized phase, $n = 7$.



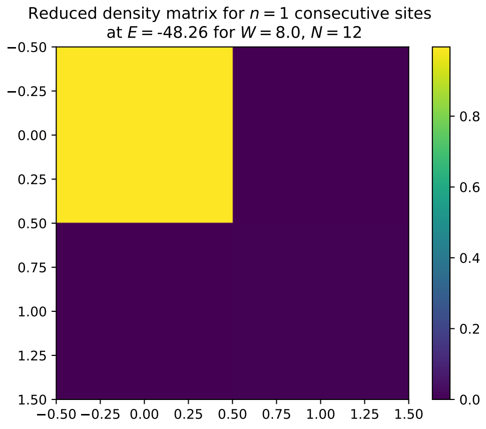
c) extended phase, $n = 4$.



d) localized phase, $n = 4$.



e) extended phase, $n = 1$.



f) localized phase, $n = 1$.

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 pattern 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:

```
model = models.Sequential()
model.add(layers.Flatten(input_shape=(np.shape(
    self.X_train)[1], np.shape(self.X_train)[1],
    2)))
model.add(layers.Dense(64, activation='relu',
    bias_regularizer='l2'))
model.add(layers.Dense(64, activation='relu',
    bias_regularizer='l2'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='
    binary_crossentropy', metrics=['accuracy'])
```

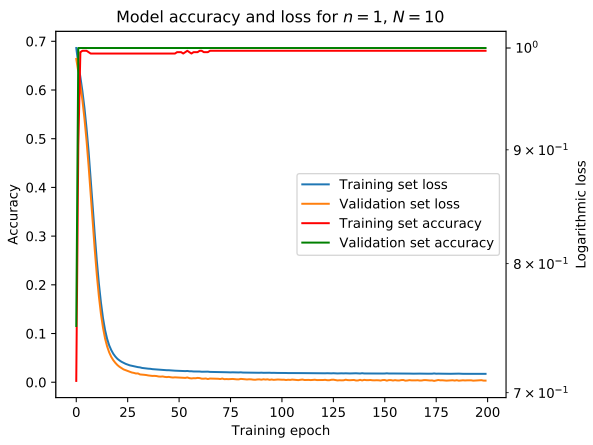
Two strategies are employed to prevent over-fitting:

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.

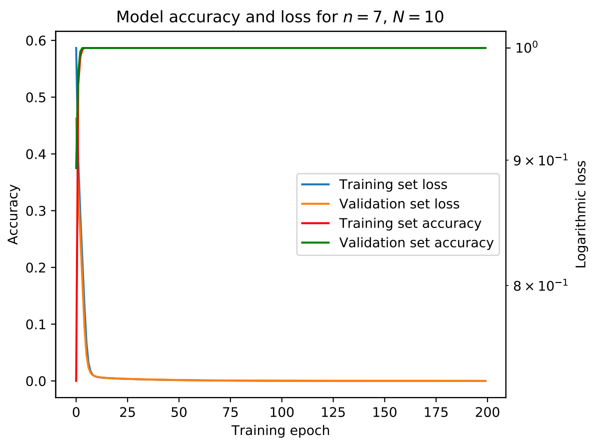
2. A bias regularizer was introduced to move the output functions closer to the origin. Even though some further regularization might still be possible, a kernel regularizer did not prove to be useful and impacted the resulting scores heavily.

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 where 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 .



a) $n = 1$



b) $n = 7$

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

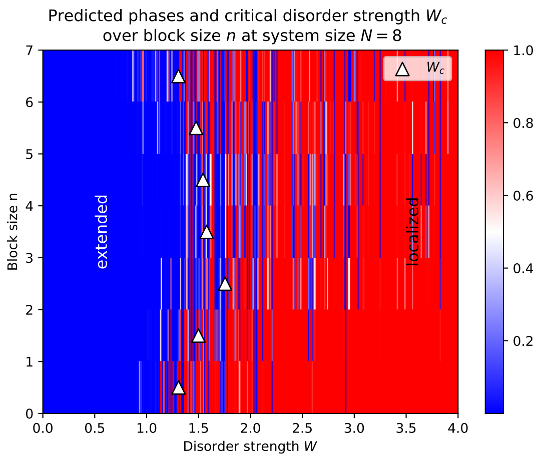
The overall losses for $n > 1$ where always found to be ≈ 0.01 with accuracies of 1.0, whereas the models at $n = 1$ showed larger losses of ≈ 0.1 and mostly acceptable accuracies near 1.

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 notized the similarity for $n = 1$ block size samples in the training set.

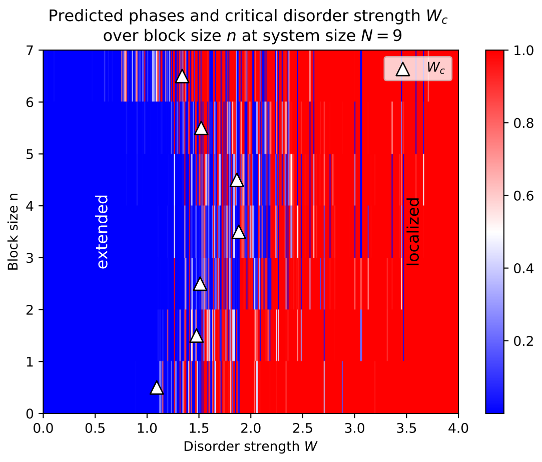
C. Analysis of critical disorder strength

1. Dependency on block size

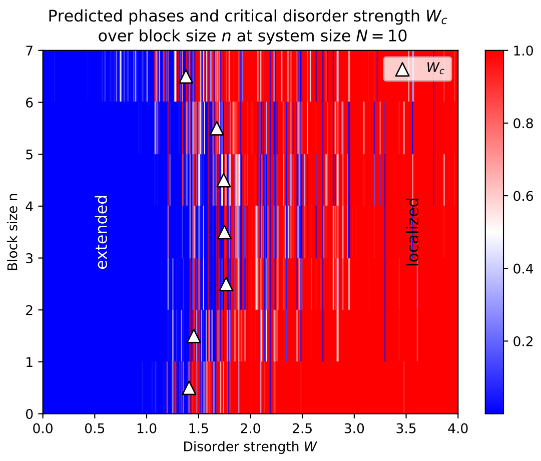
First, the testing set was generated. Following the parameter discussion in section II E, we generate five samples for each $W \in [0, 4]$, with step $\Delta W = 0.05$, resulting in 400 samples per system and block size. Afterwards, the predictions were fitted with a logistic function to obtain W_c as 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.



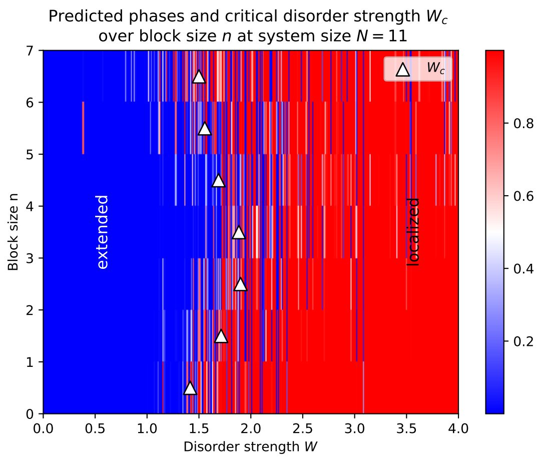
a) $L = 8$



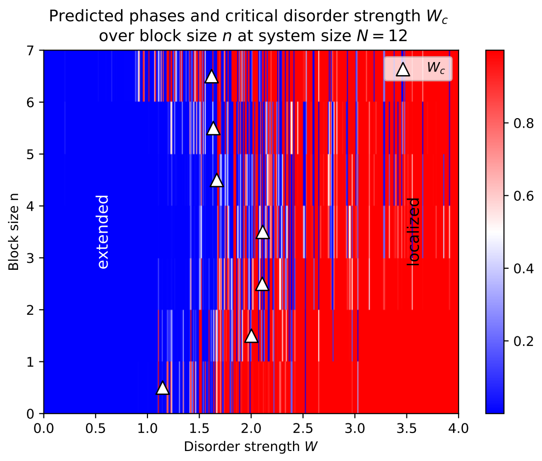
b) $L = 9$



c) $L = 10$



d) $L = 11$



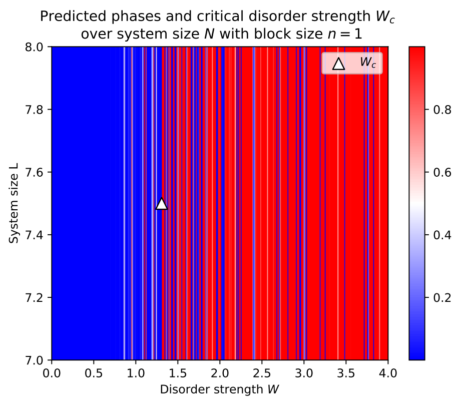
e) $L = 12$

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

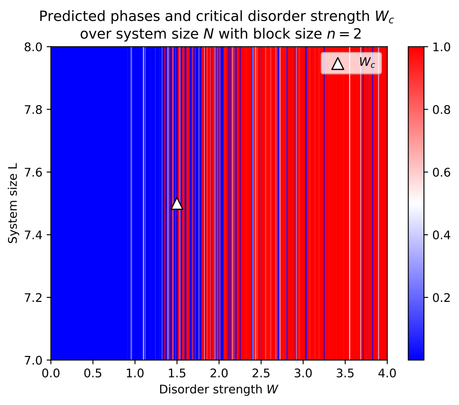
In conclusion, the predicted critical disorder strength W_c decayed, when models with larger block sizes n were used for prediction. The low W_c values for $n = \{1, 2\}$ might just as well be attributed to the poor loss and accuracy values shown in section III B. An explanation for this decay might 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. Dependeny on system size

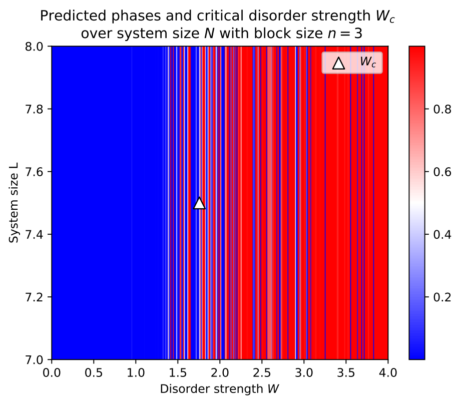
something something



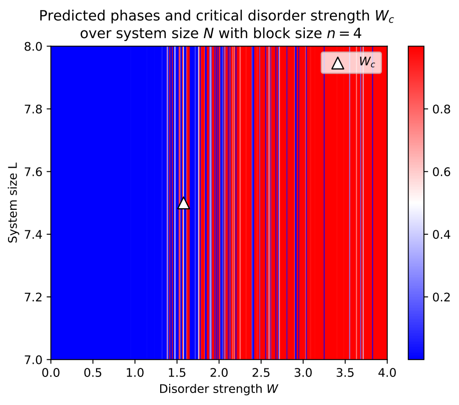
a) $n = 1$



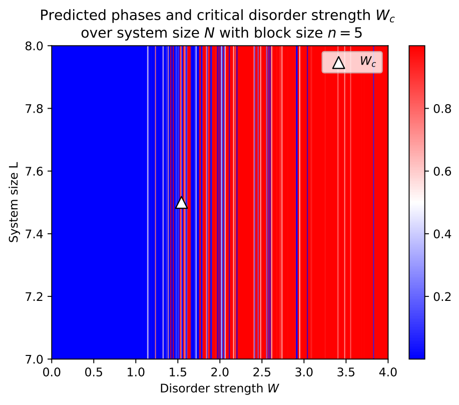
b) $n = 2$



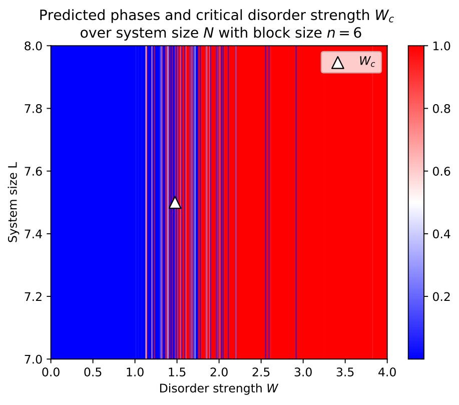
c) $n = 3$



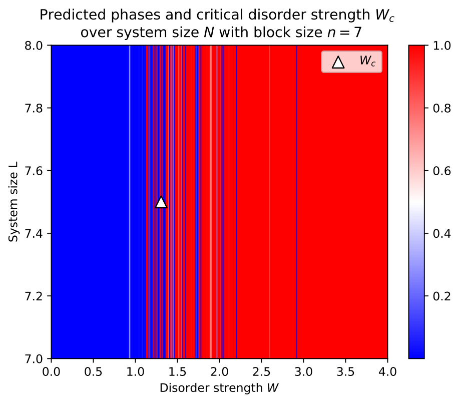
d) $n = 4$



e) $n = 5$



f) $n = 6$



g) $n = 7$

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-reductionmethod, 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

```python
from ed import *
import time
import pickle
import qutip
import matplotlib.pyplot as plt
from tqdm import trange, tqdm


def generate_training_set(Ns, Ws, n_max, repetitions):
    start_time = time.time()
    for N in tqdm(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_groundstate_figures(N, training_set_generator.training_set[n], n)
            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


def save_groundstate_figures(N, training_set, n):  # reduced_rho, W, self.N, n, E
    ergodic = [item for item in training_set if item[1] == 0.5 and item[-1] == 0][0]  # len: repetitions
    localized = [item for item in training_set if item[1] == 8 and item[-1] == 0][0]  # len: repetitions

    fig, ax1 = plt.subplots()
    pos = ax1.imshow(np.real(ergodic[0]), cmap='bwr')
    fig.colorbar(pos, ax=ax1)
    plt.title("Reduced density matrix for $n=$" + str(n) + " consecutive sites \n at $E=$"
              + str(round(ergodic[4], 2)) + " for $W=$" + str(ergodic[1]) + ", $N = $" + str(N))
    plt.savefig(
        "results/groundstates/N" + str(N) + "n" + str(n) + "_trainingset_groundstate_Wmax" + str(ergodic[1]) + ".pdf")
    plt.close()

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


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


class TrainingSetGenerator:

    def __init__(self, N, Ws, n_max, repetitions):
        self.N = int(N)  # Lattice sites
        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 W in self.Ws:
            for rep in range(self.repetitions):
                h = np.random.uniform(-W, W, size=self.N)
                H = gen_hamiltonian_lists(self.N, h, J=1)
                E, v = qutip.Qobj(H).groundstate()  # fixme might not be sparse, make sparse=True
    !!!, Eigsh SA? LM?
                rho = np.outer(v, v)
                for n in range(1, self.n_max + 1):
                    reduced_rho = self.get_partial_trace_first(rho, n) # fixme mid
                    training_set[n].append([reduced_rho, W, self.N, n, E, rep])
        return training_set

    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 diff(self, first, second):
        second = set(second)
        return [item for item in first if item not in second]

    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]
    n_max = 7
    Ws = [0.5, 8.0]  # 0.5 => ergodic/delocalized phase, 8.0 localized phase
    repetitions = 100
    generate_training_set(Ns, Ws, n_max, repetitions)

    # N=09, n=7, rep=10 7s=> rep=500: 6 min
    # N=10, n=7, rep=10 31s => rep=500: 25 min
    # N=11, n=7, rep=10 182s=> rep=500: 2,5 h
    # N=12, n=7, rep=10 00s=> rep=500

    # N=8, n=7, rep=100, first, 26s
    # N=8, n=7, rep=100, mid, 32s
```

2. Model Training

```
from sklearn.model_selection import train_test_split
import pickle
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
import tensorflow.keras.backend as k
import time


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


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))
    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.")

def mean_pred(y_true, y_pred):
    return k.mean(y_pred)


class ModelTrainer:

    def __init__(self, x, y, N, n_max):
        self.N = N
        self.n_max = n_max
        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(self):
        model = models.Sequential()
        model.add(layers.Flatten())
        model.add(layers.Dense(64, activation='relu'))
        model.add(layers.Dense(128, activation='relu'))
        model.add(layers.Dense(64, activation='relu'))
        model.add(layers.Dense(32, activation='relu'))
        model.add(layers.Dense(1, activation='sigmoid'))
        model.compile(optimizer='rmsprop', loss='mae', metrics=['accuracy'])#loss used to be mae
    loss # metrics: 'mean_absolute_error', 'mean_squared_error',
        return model

    def generate_model_sparse(self):
        model = models.Sequential()
        # if self.N != 12:
        # model.add(layers.Conv2D(32, (6, 6), activation='relu', input_shape=(np.shape(self.X_train
    )[1], np.shape(self.X_train)[1], 2)))
        # model.add(layers.MaxPooling2D((4, 4)))
        model.add(layers.Flatten(input_shape=(np.shape(self.X_train)[1], np.shape(self.X_train)[1],
     2)))
        model.add(layers.Dense(64, activation='relu', bias_regularizer='l2')), # #
        model.add(layers.Dense(64, activation='relu', bias_regularizer='l2')) # fixme use kernel
    regularizer!! l1 loss as squared error is dangerous below 1
        model.add(layers.Dense(1, activation='sigmoid')) # fixme 0 1, 1 0, drop out layer
        model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])#loss used
     to be mae loss # metrics: 'mean_absolute_error', 'mean_squared_error',
        return model

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

    def fit_model(self, batch_size, epochs):
        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)
                        )
        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['loss'], label='Training set loss')
        ln2 = ax1.plot(history.history['val_loss'], label='Validation set loss')

        #  "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')
```

```python
        ax2.tick_params(axis='y')
        ln3 = ax2.plot(history.history['acc'], 'r', label='Training set accuracy')
        ln4 = ax2.plot(history.history['val_acc'], 'g', label='Validation set accuracy')

        # 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 range(1, n_max+1):
            X, y = preprocess_training_data("lanczos/training_sets/N"+str(N)+"n"+str(n)+"_Trainset"
    )
            model_trainer = ModelTrainer(X, y, N, n_max)
            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


if __name__ == "__main__":
    Ns = [9, 10]
    n_max = 7
    train_save_model(Ns, n_max,
                     batch_size=70,
                     epochs=40)
```

3. Test set generation

```python
from generate_training_set import TrainingSetGenerator, save_pickle
from model_save_train import *
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 = [9, 10]
    Ws = np.arange(0., 4.0, 0.05)
    repetitions = 5
    n_max = 7
    generate_test_set(Ns, Ws, n_max, repetitions)
```

4. Prediction and evaluation of W_c

```python
from model_save_train import *
from scipy.optimize import curve_fit


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


def get_wc(N, n, Ws, repetitions):
    """
    Calculates Wc

    :param N: system size for Model and Testset
    :param n: block size for Model and Testset
    :param Ws: chosen interval for fitting
    :param repetitions: Number of datapoints per W_pred
    :return: Wc
    """
    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 = np.reshape(state_prediction, (int(len(state_prediction)/repetitions),
    repetitions))
    state_prediction = np.mean(state_prediction, axis=1)

    popt, pcov = curve_fit(logistic, Ws, np.reshape(state_prediction, (len(state_prediction))))  #
    state_prediction.astype(np.float))
    # plot_wc_fit(N,popt,state_prediction)
    return popt[0]


class HeatMapPlotter:

    def __init__(self, Ns, Ws, n_max, repetitions):
        self.Ns = Ns
        self.Ws = Ws
        self.n_max = n_max
        self.repetitions = repetitions


    def predict_w_n(self):
        W_preds = {system_size : [] for system_size in self.Ns}
        for N in self.Ns:
            for n in range(1, self.n_max + 1):
                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')
                W_preds[N].append(model.predict(X))
        return W_preds

    def fit_wc_n(self):
        W_c_fit = {system_size : [] for system_size in self.Ns}
        for N in self.Ns:
            for n in range(1, self.n_max + 1):
                W_c_fit[N].append((get_wc(N, n, self.Ws, self.repetitions), n))
        return W_c_fit

    def predict_w_N(self):
        W_preds = {block_size : [] for block_size in range(1, self.n_max+1)}
        for n in range(1, self.n_max + 1):
            for N in self.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')
                W_preds[n].append(model.predict(X))
        return W_preds

    def fit_wc_N(self):
        W_c_fit = {block_size : [] for block_size in range(1, self.n_max+1)}
        for n in range(1, self.n_max + 1):
            for N in self.Ns:
                W_c_fit[n].append((get_wc(N, n, self.Ws, self.repetitions),N))
        return W_c_fit

    def plot_wc_heatmap_n(self):
        """
        Plots Heatmap with blocksize and W_pred

        W_pred: W x n array
        W_c_fit: W_c(n) x 1 array
        """
        self.W_preds = self.predict_w_n()
        self.W_c_fit = self.fit_wc_n()

        for N in self.Ns:
            W_pred = np.asarray(self.W_preds[N])
            W_pred = np.reshape(W_pred, (np.shape(W_pred)[0],np.shape(W_pred)[1]))
            W_c_fit = np.array(self.W_c_fit[N])


            # W_c_fit = np.reshape(W_c_fit, (np.shape(W_c_fit)[0], np.shape(W_c_fit)[1]))
            fig, ax = plt.subplots()
            plt.title("Predicted phases and critical disorder strength $W_c$ \n over block size $n$
     at system size $N=$" + str(N))
            plt.text(0.5, 3.5, 'extended', {'color': 'w', 'fontsize': 12},
                     horizontalalignment='left',
                     verticalalignment='center',
                     rotation=90,
                     )
            plt.text(3.5, 3.5, 'localized', {'color': 'k', 'fontsize': 12},
                     horizontalalignment='left',
                     verticalalignment='center',
                     rotation=90,
                     )
            pos = ax.imshow(W_pred, extent=(0, 4, 0, 7), aspect=0.5, cmap='bwr')
            fig.colorbar(pos, ax=ax)
            ax.scatter(W_c_fit[:,0], W_c_fit[:,1]-0.5, s=100, c="w", marker='^', label='$W_c$',
    edgecolors="k")
            plt.ylabel("Block size n")
            plt.xlabel("Disorder strength $W$")
            ax.legend()
            plt.tight_layout()
            plt.savefig('results/Wc/N'+str(N)+'_Wc_n_dependency.pdf')
            plt.close()
        pass

    def plot_wc_heatmap_N(self):
        """
        Plots Heatmap with blocksize and W_pred

        W_pred: W x n array
        W_c_fit: W_c(n) x 1 array
        """
        self.W_preds = self.predict_w_N()
        self.W_c_fit = self.fit_wc_N()

        for n in range(1, self.n_max+1):
            W_pred = np.asarray(self.W_preds[n])
            W_pred = np.reshape(W_pred, (np.shape(W_pred)[0],np.shape(W_pred)[1]))
            W_c_fit = np.array(self.W_c_fit[n])


            # W_c_fit = np.reshape(W_c_fit, (np.shape(W_c_fit)[0], np.shape(W_c_fit)[1]))
            fig, ax = plt.subplots()
            plt.title("Predicted phases and critical disorder strength $W_c$ \n over system size 
    $N$ with block size $n=$" + str(n))
            plt.text(0.5, 10, 'extended', {'color': 'w', 'fontsize': 12},
                     horizontalalignment='left',
                     verticalalignment='center',
                     rotation=90,
                     )
            plt.text(3.5, 10, 'localized', {'color': 'k', 'fontsize': 12},
                     horizontalalignment='left',
                     verticalalignment='center',
                     rotation=90,
                     )
            pos = ax.imshow(W_pred, extent=(0, 4, self.Ns[0]-1, self.Ns[-1]), aspect='auto', cmap='
    bwr')
            # Shift ticks to be at 0.5, 1.5, etc
            # ax.yaxis.set(ticks=np.arange(0.5, len(self.Ns)), ticklabels=map(str, input(self.Ns)))

            fig.colorbar(pos, ax=ax)
            # print(W_c_fit)
            ax.scatter(W_c_fit[:,0], W_c_fit[:,1]-0.5, s=100, c="w", marker='^', label='$W_c$',
    edgecolors="k")
            plt.ylabel("System size L")
            plt.xlabel("Disorder strength $W$")
            ax.legend()
            plt.savefig('results/Wc/n'+str(n)+'_Wc_N_dependency.pdf')
            plt.close()
        pass

    def plot_wc_fit(self, N, popt, state_prediction):
        fig, ax1 = plt.subplots()
        ax1 = plt.scatter(self.Ws, state_prediction)
        ax1 = plt.plot(self.Ws, logistic(self.Ws, *popt), 'k')

        plt.title('Phase prediction $N = $' + str(N) + ", $W_c = $" + "{0:.3g}".format(popt[0]))
        plt.ylabel('Probability of localized phase')
        plt.xlabel('$W_{max}$')
        plt.legend(['Logistic fit', 'Predicted phase'], loc='upper left')
        plt.savefig('results/N' + str(N) + '_predict_wc.pdf')
        pass


if __name__ == "__main__":
    Ns = [9, 10, 11, 12]
    Ws = np.arange(0., 4.0, 0.05)
    n_max = 7
    repetitions = 5
    heat_map_plotter = HeatMapPlotter(Ns, Ws, n_max, repetitions)
    heat_map_plotter.plot_wc_heatmap_n()
    print("done")
    heat_map_plotter.plot_wc_heatmap_N()
    print("done")
```

5. Exact diagonalization

```python
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


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


def gen_hamiltonian_lists(L, h, J=1):
    sx_list = gen_sx_list(L)
    sy_list = gen_sy_list(L)
    sz_list = gen_sz_list(L)

    H = J*(sx_list[0] * sx_list[1] + sy_list[0] * sy_list[1] + sz_list[0] * sz_list[1]) - h[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])
    - h[i]*sz_list[i]
    return H

# fixme delete

# def gen_hamiltonian_random_h(L, W, J=1.):
#     """ assumes open boundary conditions """
#     sx_list = gen_sx_list(L)
#     sz_list = gen_sz_list(L)
#     H = sparse.csr_matrix((2**L, 2**L))
#     for j in range(L-1):
#         H = H - J *( sx_list[j] * sx_list[(j+1)%L])
#         H = H - np.random.uniform(-W, W) * sz_list[j]
#     H = H - np.random.uniform(-W, W) * sz_list[-1]
#     return H
```