Variational Ground State Search of the Transverse-Field Ising Model using Matrix Product States and Gradient Descent

Pablo Miguel Llaguno Salazar
Computational Methods in Many-Body Physics
(Dated: July 28, 2025)

The study of quantum many-body systems is often hindered by the exponential scaling of the Hilbert space. Tensor networks, particularly Matrix Product States (MPS), provide an efficient variational ansatz for the ground states of one-dimensional gapped Hamiltonians. In this work, we explore a modern, gradient-based approach to finding the ground state of the 1D Transverse-Field Ising Model (TFIM). We implement an MPS representation within the PyTorch framework and use its automatic differentiation capabilities to perform a direct gradient descent on the energy expectation value. To benchmark this method, we compare its performance, in terms of convergence speed and final accuracy, against two well-established algorithms: the Density Matrix Renormalization Group (DMRG) and imaginary time evolution with the Time-Evolving Block Decimation (TEBD). Our results confirm that while gradient descent successfully finds the ground state, the highly specialized nature of DMRG makes it a vastly more efficient and accurate tool for this class of problems.

I. INTRODUCTION

A central challenge in computational condensed matter physics is the simulation of quantum many-body systems. The dimension of the Hilbert space grows exponentially with the number of particles, a difficulty often referred to as the "curse of dimensionality." This makes exact diagonalization of the system's Hamiltonian feasible only for very small system sizes.

Fortunately, the ground states of many physically relevant Hamiltonians, particularly those with local interactions and an energy gap, do not explore the entirety of this vast Hilbert space. They typically exhibit low entanglement, satisfying an "area law" for entanglement entropy [1]. For one-dimensional systems, this property allows for an efficient representation of such states using Matrix Product States (MPS) [2]. The task of finding the ground state can then be reformulated as a variational optimization problem within the manifold of MPS with a fixed bond dimension χ. The Density Matrix Renormalization Group (DMRG) [3] and the Time-Evolving Block Decimation (TEBD) algorithm [4] are state-of-the-art methods that solve this problem with remarkable efficiency.

In this report, we investigate an alternative approach that leverages tools from the machine learning community. We treat the MPS as a variational neural network and the energy as a cost function. By implementing the MPS and the Hamiltonian for the Transverse-Field Ising Model (TFIM) in PyTorch, we can use automatic differentiation to compute the gradient of the energy with respect to all parameters in the MPS tensors. We then perform a direct gradient descent to search for the ground state. The primary goal of this work is to implement this method and quantitatively compare its performance against our implementations of DMRG and TEBD.

II. THEORY

A. Matrix Product States

An MPS represents a quantum state $|\Psi\rangle$ by decomposing its high-rank coefficient tensor into a product of smaller, site-specific tensors. For a chain with open boundary conditions (OBC), the state is written as:

$$|\Psi\rangle = \sum_{j_1,\ldots,j_L} (M^{[1]j_1} M^{[2]j_2} \cdots M^{[L]j_L}) |j_1, \ldots, j_L\rangle \tag{1}$$

where each $M^{[i]j_i}$ is a $\chi_i \times \chi_{i+1}$ matrix for a given physical index j_i. For periodic boundary conditions (PBC), a trace is taken over the virtual indices.

B. The Transverse-Field Ising Model (TFIM)

The 1D TFIM is a canonical model of a quantum phase transition. Its Hamiltonian is given by:

$$H = -J \sum_{i=0}^{L-2} \sigma_i^x \sigma_{i+1}^x - g \sum_{i=0}^{L-1} \sigma_i^z \tag{2}$$

where σ_i^α are the Pauli matrices acting on site i, L is the system size, J is the nearest-neighbor coupling strength, and g is the transverse magnetic field. For efficient computation, this Hamiltonian can be expressed as a Matrix Product Operator (MPO), where the tensor for each site in the bulk is given by:

$$W^{[i]} = \begin{pmatrix} \mathbb{I} & \sigma^x & -g\sigma^z \\ 0 & 0 & -J\sigma^x \\ 0 & 0 & \mathbb{I} \end{pmatrix} \tag{3}$$

C. Ground State Search Algorithms

1. *Variational Gradient Descent*

Based on the variational principle, the energy expectation value $E(\Psi) = \frac{\langle\Psi|H|\Psi\rangle}{\langle\Psi|\Psi\rangle}$ is minimized by the ground state. We can therefore treat E as a loss function and the MPS tensors as its parameters. Gradient descent provides a general method for this optimization. Starting from a random MPS, we iteratively update its tensors by taking a small step in the direction of the negative energy gradient:

$$\text{params}_{\text{new}} = \text{params}_{\text{old}} - \eta \nabla E(\Psi) \tag{4}$$

where η is the learning rate. The gradient ∇E can be computed analytically by contracting the energy ”sandwich” with the identity operator at the position of the tensor being differentiated. However, modern frameworks like PyTorch compute this gradient automatically.

2. *DMRG and TEBD*

DMRG is a variational algorithm that iteratively optimizes the MPS tensors by sweeping through the chain and solving a local eigenvalue problem for two sites at a time. This locally optimal update ensures that the energy is monotonically decreased at every step. TEBD finds the ground state by simulating evolution in imaginary time, $\tau = it$, where the operator $e^{-\tau H}$ projects out excited states.

III. IMPLEMENTATION DETAILS

Our implementation is built entirely in PyTorch. The ‘MPS‘ and ‘TFIModel‘ classes store their data as ‘torch.Tensor‘ objects.

A. Initial State

A crucial aspect of variational optimization is the choice of the initial state. For DMRG and TEBD, starting with a simple product state (e.g., all spins up) is sufficient, as these algorithms can dynamically increase the bond dimension and explore the Hilbert space effectively.

Gradient descent, however, does not change the bond dimension. Therefore, the initial MPS must have a bond dimension χ_{max} large enough to support the entanglement of the ground state. Furthermore, if all tensors are identical, the gradient for each tensor will be the same, which can hinder effective optimization. To break this symmetry and ensure a rich starting point, we initialize the MPS with random tensors drawn from a Gaussian distribution.

B. Learning Rate

The learning rate η is a critical hyperparameter. If it is too large, the optimization can become unstable and oscillate or diverge. If it is too small, convergence will be prohibitively slow. Through experimentation, we found that a learning rate of $\eta = 0.01$ with the Adam optimizer provided a good balance of stability and speed for the system under study.

IV. RESULTS AND DISCUSSION

We performed simulations for a TFIM chain of size $L = 14$ with parameters $J = 1.0$ and $g = 1.5$. For a fair comparison, both DMRG and Gradient Descent were initialized with the same random MPS of bond dimension $\chi = 30$. TEBD was initialized from a simple product state with all spins pointing upwards.

A. Energy Convergence

Figure 1 shows the energy as a function of optimization steps. All methods successfully converge to the known exact ground state energy, validating our implementations. However, their convergence behaviors differ dramatically.

DMRG exhibits a spectacular rate of convergence, reaching the ground state energy within just two sweeps. TEBD shows a smooth and rapid decay. Gradient Descent also converges, but requires significantly more iterations.

From a physical analogy, DMRG acts like an expert rock climber, meticulously finding the best possible local hold (optimizing two sites at a time) to guarantee

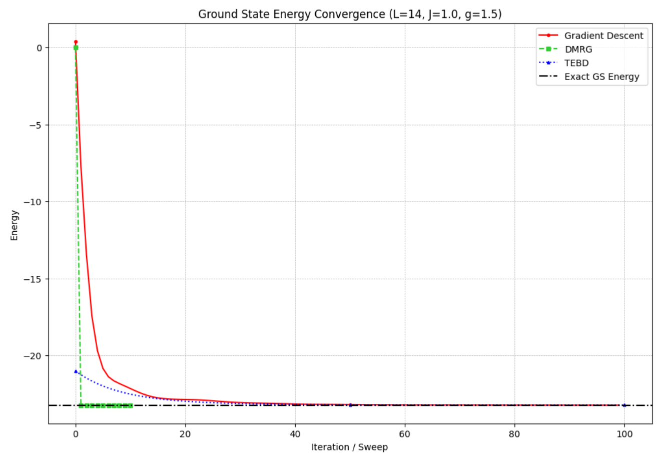

FIG. 1. Ground state energy convergence for the three algorithms. DMRG converges in only 2 sweeps. TEBD and Gradient Descent require hundreds of iterations to reach the same energy.

progress. In contrast, gradient descent is like a hiker with a blindfold who can only feel the steepest downward slope at their feet. While this hiker will eventually get to the bottom of the valley, their path is far less direct and efficient.

B. Relative Error

The difference in performance is even more stark when viewing the relative error on a logarithmic scale, as shown in Figure 2.

The plot clearly shows that DMRG is in a class of its own. After only two sweeps, the relative error plummets to the order of 10^{-15}, which is the limit of double-precision floating-point arithmetic. This indicates that DMRG has found a nearly exact representation of the ground state within the given bond dimension.

Both TEBD and Gradient Descent converge much more slowly. While they reach a respectable accuracy (relative error of $\sim 10^{-4}$), they are orders of magnitude less precise than DMRG after a comparable number of iterations.

V. CONCLUSION

In this work, we successfully implemented a variational ground state search for the Transverse-Field Ising Model using a direct gradient descent on the Matrix Product State tensors. By leveraging the PyTorch library, we were able to use automatic differentiation to compute the energy gradients, providing a powerful and flexible optimization framework.

Our comparison with the established DMRG and TEBD algorithms yielded a clear and expected result: for one-dimensional gapped systems, **DMRG is overwhelmingly superior in both convergence speed and final accuracy**. Its physically motivated, locally optimal update strategy is far more effective than the simple, global update of gradient descent.

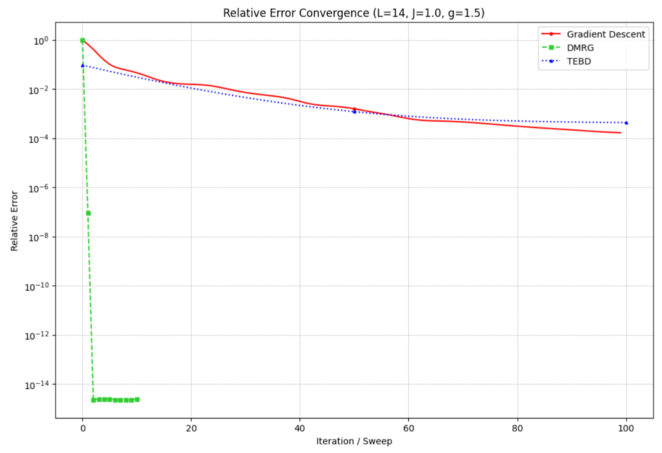

FIG. 2. Relative error with respect to the exact ground state energy. Note the logarithmic scale on the y-axis. DMRG reaches machine precision ($\sim 10^{-15}$) almost immediately.

This study serves as an excellent demonstration of both the power of the MPS ansatz and the efficiency of specialized tensor network algorithms. While gradient-based methods are highly flexible and powerful tools, for the specific problem of finding 1D ground states, purpose-built algorithms like DMRG remain the undisputed state of the art.

[1] J. Eisert, M. Cramer, and M. B. Plenio, "Area laws for the entanglement entropy," Rev. Mod. Phys. 82, 277 (2010).
[2] U. Schollwöck, "The density-matrix renormalization group in the age of matrix product states," Annals of Physics 326, 96 (2011).
[3] S. R. White, "Density matrix formulation for quantum renormalization groups," Phys. Rev. Lett. 69, 2863 (1992).
[4] G. Vidal, "Efficient Simulation of One-Dimensional Quantum Many-Body Systems," Phys. Rev. Lett. 93, 040502 (2004).

Appendix A: Core Gradient Descent Code

The core logic of the gradient descent algorithm is contained in the main optimization loop. The following Python snippet, using PyTorch, demonstrates the implementation.

```python
# %%
import torch
from src.a_mps_torch import MPS, init_random_mps, init_spinup_MPS
from src.b_model_torch import TFIModel
from src.dmrg import DMRGEngine
from src.tfi_exact import finite_gs_energy
import src.tebd_torch as tebd
import src.gd as gd

# %%
L = 14
J = 1.0
g = 1.5

theoretical_energy = finite_gs_energy(L, J, g)
print(f"Theoretical ground state energy: {theoretical_energy:.6f}")

psi = init_random_mps(L, chi_max=30)
model = TFIModel(L, J=J, g=g)

# %%
# DMRG
def run_dmrg(chi_max, num_sweeps):
    """Performs DMRG and returns the energy history."""
    print("\n--- Running DMRG ---")
    psi_dmrg = psi.copy()
    eng = DMRGEngine(psi_dmrg, model, chi_max=chi_max)

    initial_energy = model.energy(psi_dmrg)/ psi_dmrg.norm_squared()
    energies = [initial_energy.item()]
    # energies = []

    for sweep in range(num_sweeps):
        eng.sweep()
        # After a sweep, the MPS norm is ~1, so we can use the unnormalized energy
        energy = model.energy(eng.psi)
        energies.append(energy)
        err = abs((energy - theoretical_energy) / theoretical_energy)
        print(f"Sweep {sweep + 1:2d} | Energy = {energy:.12f} | rel. error {err:.4e}")

    return energies

energies_dmrg = run_dmrg(chi_max=30, num_sweeps=10)
energies_dmrg

# %%
# TEBD
def run_tebd(L, chi_max, num_steps=100, dt=0.01):
    """Performs imaginary time TEBD and returns the energy history."""
    print("\n--- Running TEBD ---")

    # TEBD is often started from a simple product state
    psi_tebd = init_spinup_MPS(L)
    U_bonds = tebd.calc_U_bonds(model, dt)

    initial_energy = model.energy(psi_tebd)
    energies = [initial_energy.item()]

    for step in range(num_steps):
        tebd.run_TEBD(psi_tebd, U_bonds, N_steps=1, chi_max=chi_max, eps=1.e-14)
        energy = model.energy(psi_tebd)
        energies.append(energy)
        if step % 20 == 0 or step == num_steps - 1:
            print(f"Step {step:5d} | Time = {dt*(step+1):.2f} | Energy = {energy:.12f}")

    return energies

# tebd.example_TEBD_gs_finite(L, J, g)
energies_tebd = run_tebd(L, chi_max=30)

# %%
def run_gradient_descent(num_steps, learning_rate):
    """Performs gradient descent and returns the energy history."""
    print("\n--- Running Gradient Descent ---")
    psi_gd = psi.copy()
    for B in psi_gd.Bs:
        B.requires_grad = True
    for S in psi_gd.Ss:
        S.requires_grad = True

    # Re-enable gradients on the copied tensors for this optimization run
    for tensor in psi_gd.get_tensors_as_list():
        tensor.requires_grad = True

    optimizer = torch.optim.Adam(psi_gd.get_tensors_as_list(), lr=learning_rate)
    energies = []

    for step in range(num_steps):
        optimizer.zero_grad() # zero gradient from previous step
        loss =  model.energy_mpo(psi_gd) / psi_gd.norm_squared()

        loss.backward() # backward propagation to use Pytorch's autograd

    # update all MPS tensors
        optimizer.step()

    # store and print energy for monitoring
        current_energy = loss.item()
        energies.append(current_energy)
        if step % 10 == 0 or step == num_steps - 1:
            print(f"Step {step:5d} | Energy = {current_energy:.12f}")

    return energies

energies_gd = run_gradient_descent(num_steps=100, learning_rate=0.1)

# %%
# ==============================================================================
# 3. Plotting the Comparison
# ==============================================================================

import matplotlib.pyplot as plt
import numpy as np

print("\n--- Generating Plots ---")

# Plot 1: Energy Convergence
plt.figure(figsize=(12, 8))
plt.plot(energies_gd, label='Gradient Descent', color='red', marker='o', markersize=3, linestyle='-
    ', markevery=50)
# DMRG sweeps are much more powerful, so we plot vs sweeps (not individual updates)
plt.plot(np.arange(len(energies_dmrg)), energies_dmrg, label='DMRG', color='limegreen', marker='s',
     markersize=5, linestyle='--')
plt.plot(energies_tebd, label='TEBD', color='blue', marker='^', markersize=3, linestyle=':',
    markevery=50)

plt.axhline(y=theoretical_energy, color='black', linestyle='-.', label=f'Exact GS Energy')

plt.xlabel("Iteration / Sweep")
plt.ylabel("Energy")
plt.title(f"Ground State Energy Convergence (L={L}, J={J}, g={g})")
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()

# Plot 2: Relative Error Convergence (Log Scale)
rel_error_gd = [abs((E - theoretical_energy) / theoretical_energy) for E in energies_gd]
rel_error_dmrg = [abs((E - theoretical_energy) / theoretical_energy) for E in energies_dmrg]
rel_error_tebd = [abs((E - theoretical_energy) / theoretical_energy) for E in energies_tebd]

plt.figure(figsize=(12, 8))
plt.semilogy(rel_error_gd, label='Gradient Descent', color='red', marker='o', markersize=3,
    linestyle='-', markevery=50)
plt.semilogy(np.arange(len(rel_error_dmrg)), rel_error_dmrg, label='DMRG', color='limegreen',
    marker='s', markersize=5, linestyle='--')
plt.semilogy(rel_error_tebd, label='TEBD', color='blue', marker='^', markersize=3, linestyle=':',
    markevery=50)
```

```python

plt.xlabel("Iteration / Sweep")
plt.ylabel("Relative Error")
plt.title(f"Relative Error Convergence (L={L}, J={J}, g={g})")
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
# plt.ylim(bottom=1e-7) # Set a lower limit for the y-axis to see convergence floor
plt.show()
```

```python
"""Toy code implementing a matrix product state, ported to PyTorch
This version is designed for gradient-based optimization
"""

import torch
from typing import List
from scipy.linalg import svd


class MPS:
    """Class for a matrix product state.

    We index sites with `i` from 0 to L-1; bond `i` is left of site `i`.
    THe MPS is stored in a right-canonical form using the Vidal notation (B, S)

    Attributes
    ----------
    Bs : list of torch.Tensor
        The 'B' tensors in right-canonical form, one for each physical site.
        Each `B[i]` has legs (virtual left, physical, virtual right), in short ``vL i vR``
    Ss : list of torch.Tensor
        The Schmidt values (singular values) at each bond. `Ss[i]` is for the bond
        to the left of site `i`.
    L : int
        Number of sites.
    """

    def __init__(self, Bs: List[torch.Tensor], Ss: List[torch.Tensor]):
        self.Bs = Bs
        self.Ss = Ss
        self.L = len(Bs)

    def copy(self):
        new_Bs = [B.clone().detach() for B in self.Bs]
        new_Ss = [S.clone().detach() for S in self.Ss]
        return MPS(new_Bs, new_Ss)

    def get_tensors_as_list(self) -> List[torch.Tensor]:
        """Returns a flat list of all tensors in the MPS that have gradients."""
        tensors = []
        for B in self.Bs:
            if B.requires_grad:
                tensors.append(B)
        for S in self.Ss:
            if S.requires_grad:
                tensors.append(S)
        return tensors

    def get_theta1(self, i: int) -> torch.Tensor:
        """
        Calculate the effective single-site wave function on site `i`
        in mixed canonical form.

        The returned array has legs (vL, i, vR).
        """
        # vL [vL'], [vL] i vR -> vL i vR
        return torch.tensordot(torch.diag(self.Ss[i]), self.Bs[i], dims=([1], [0]))

    def get_theta2(self, i: int) -> torch.Tensor:
        """
        Calculate the effective two-site wave function on sites i, j=(i+1)
        in mixed canonical form.

        The returned array has legs (vL, i, j, vR).
        """
        j = i + 1
        # vL i [vR], [vL] j vR -> vL i j vR
        theta1 = self.get_theta1(i)
        return torch.tensordot(theta1, self.Bs[j], dims=([2], [0]))

    def get_chi(self):
        """Return bond dimensions."""
        return [self.Bs[i].shape[2] for i in range(self.L - 1)]

    def site_expectation_value(self, op: torch.Tensor) -> List[torch.Tensor]:
        """Calculate expectation values of a local operator at each site."""
        result = []
        for i in range(self.L):
            theta = self.get_theta1(i)  # vL i vR
            # op acts on physical leg 'i'
            op_theta = torch.tensordot(op, theta, dims=([1], [1]))  # i [i*], vL [i] vR -> i vL vR
            # contract with conjugate to get expectation value
            # [vL*] [i*] [vR*], [i] [vL] [vR]
            exp_val = torch.tensordot(theta.conj(), op_theta, dims=([0, 1, 2], [1, 0, 2]))
            result.append(exp_val.real)
        return result

    def bond_expectation_value(self, op_list: List[torch.Tensor]) -> List[torch.Tensor]:
        """Calculate expectation values of two-site operators on each bond."""
        result = []
        for i in range(self.L - 1):
            theta = self.get_theta2(i)  # vL i j vR
            # op acts on physical legs 'i' and 'j'
            op_theta = torch.tensordot(op_list[i], theta, dims=([2, 3], [1, 2])) # i j [i*] [j*], 
    vL [i] [j] vR -> i j vL vR
            # contract with conjugate
            # [vL*] [i*] [j*] [vR*], [i] [j] [vL] [vR]
            exp_val = torch.tensordot(theta.conj(), op_theta, dims=([0, 1, 2, 3], [2, 0, 1, 3]))
            result.append(exp_val.real)
        return result

    def norm_squared(self) -> torch.Tensor:
        """
        Calculates the squared norm <psi|psi> of the MPS.
        This is done by contracting the network from left to right.
        """
        # Start with a 1x1 identity matrix, representing the left boundary
        contr = torch.eye(1, dtype=self.Bs[0].dtype)
        for i in range(self.L):
            B = self.Bs[i] # vL, i, vR
            # Contract the current accumulated tensor with the MPS tensor B
            # contr: [vL*], vL
            # B: vL, i, vR
            temp = torch.tensordot(contr, B, dims=([1], [0])) # [vL*] [vL], i, vR -> [vL*] i vR
            # Contract with the conjugate of B to form the transfer matrix for the site
            # temp: [vL*] i vR
            # B.conj(): vL*, i*, vR*
            contr = torch.tensordot(temp, B.conj(), dims=([0, 1], [0, 1])) # [vL*] [i] vR, [vL*] [i
    *] vR* -> vR vR*
        # The final result should be a 1x1 matrix, its single element is the norm squared
        return contr.squeeze()

    def expectation_value_mpo(self, mpo: List[torch.Tensor]) -> torch.Tensor:
        """
        Compute the expectation value <psi|H_mpo|psi>.
        This is done by contracting the "sandwich" network from left to right.
        """
        # Start with the left boundary vector for the MPO contraction
        contr = torch.zeros(1, mpo[0].shape[0], 1, dtype=self.Bs[0].dtype)
        contr[0, 0, 0] = 1.0 # Corresponds to the identity operator at the start of the MPO

```

```python
        for i in range(self.L):
            B_ket = self.Bs[i]       # vL, i, vR
            B_bra = B_ket.conj()   # vL*, i*, vR*
            W = mpo[i]             # wL, wR, i, i*

            # Contract with the ket
            # contr: vL_bra, wL, vL_ket
            # B_ket: vL_ket, i, vR_ket
            temp = torch.tensordot(contr, B_ket, dims=([2], [0])) # vL_bra, wL, [vL_ket], i, vR_ket -> vL_bra, wL, i, vR_ket
            # Contract with the MPO tensor W
            # temp: vL_bra, wL, i, vR_ket
            # W: wL, wR, i, i*
            temp = torch.tensordot(temp, W, dims=([1, 2], [0, 2])) # vL_bra, [wL], [i], vR_ket, wR, i* -> vL_bra, vR_ket, wR, i*
            # Contract with the bra
            # temp: vL_bra, vR_ket, wR, i*
            # B_bra: vL_bra, i*, vR_bra
            contr = torch.tensordot(temp, B_bra, dims=([0, 3], [0, 1])) # [vL_bra], vR_ket, wR, [i*], [vL_bra], [i*], vR_bra -> vR_ket, wR, vR_bra
            # Transpose to get the correct order for the next iteration: vR_bra, wR, vR_ket
            contr = contr.permute(2, 1, 0)

        # The final result is the element corresponding to the end of the MPO
        return contr[0, -1, 0].real

    def entanglement_entropy(self) -> List[float]:
        """Return the (von-Neumann) entanglement entropy for each bond."""
        result = []
        for i in range(1, self.L):
            S = self.Ss[i].clone()
            S = S[S > 1e-30]  # Avoid log(0)
            S2 = S * S
            # The norm of Schmidt values should be 1.
            assert abs(torch.linalg.norm(S) - 1.) < 1.e-10
            entropy = -torch.sum(S2 * torch.log(S2))
            result.append(entropy.item())
        return result


def init_spinup_MPS(L: int) -> MPS:
    """Return a product state with all spins up as a PyTorch MPS."""
    B = torch.zeros((1, 2, 1), dtype=torch.float64)
    B[0, 0, 0] = 1.
    S = torch.ones(1, dtype=torch.float64)
    Bs = [B.clone() for _ in range(L)]
    Ss = [S.clone() for _ in range(L + 1)] # L+1 bonds for L sites
    return MPS(Bs, Ss)

def init_random_mps(L: int, chi_max: int, d: int = 2) -> MPS:
    """Initializes a random MPS with requires_grad=True for optimization."""
    Bs = []
    Ss = []

    # Left-most bond dimension is 1
    chi_left = 1

    # First S matrix (bond 0)
    s0 = torch.ones(1, dtype=torch.float64, requires_grad=True)
    Ss.append(s0)

    for i in range(L):
        chi_right = min(chi_max, d ** (i + 1), d ** (L - i - 1))
        if i == L - 1:
            chi_right = 1

        # Random B tensor
        B = torch.randn((chi_left, d, chi_right), dtype=torch.float64, requires_grad=True)
        Bs.append(B)

        # Random S matrix
        s = torch.rand(chi_right, dtype=torch.float64, requires_grad=True)
        with torch.no_grad():
            s = s / torch.linalg.norm(s)
        Ss.append(s)

        chi_left = chi_right

    return MPS(Bs, Ss)

def split_truncate_theta(theta: torch.Tensor, chi_max: int, eps: float):
  """
  Split and truncate a two-site wave function in mixed canonical form.

  Split a two-site wave function as follows::
    vL --(theta)-- vR     =>    vL --(A)--diag(S)--(B)-- vR
        |   |                       |             |
        i   j                       i             j

  Afterwards, truncate in the new leg (labeled ``vC``).

  Parameters
  ----------
  theta : torch.Tensor
    Two-site wave function in mixed canonical form, with legs ``vL, i, j, vR``.
  chi_max : int
    Maximum number of singular values to keep
  eps : float
    Discard any singular values smaller than that.

  Returns
  -------
  A : torch.Tensor
    Left-canonical matrix on site i, with legs ``vL, i, vC``
  S : torch.Tensor
    Singular/Schmidt values.
  B : torch.Tensor
    Right-canonical matrix on site j, with legs ``vC, j, vR``
  """
  chivL, dL, dR, chivR = theta.shape
  theta = theta.reshape(chivL * dL, dR * chivR)

  # SVD
  U, S, Vh = torch.linalg.svd(theta, full_matrices=False)

  # Truncate
  chivC = min(chi_max, torch.sum(S > eps).item())
  assert chivC >= 1
  # Keep the largest `chivC` singular values
  piv = torch.argsort(S, descending=True)[:chivC]
  U, S, Vh = U[:, piv], S[piv], Vh[piv, :]

  # Renormalize
  S = S / torch.linalg.norm(S)

  # Reshape back to tensors
  A = U.reshape(chivL, dL, chivC)
  B = Vh.reshape(chivC, dR, chivR)

  return A, S, B
```

```
"""
Toy code implementing the transverse-field ising model, ported to PyTorch.
"""

import torch
from typing import List
from src.a_mps_torch import MPS

class TFIModel:
```

```python
"""
Class generating the Hamiltonian of the transverse-field Ising model using PyTorch.

The Hamiltonian reads
H = - J \\sum_{i} \\sigma^x_i \\sigma^x_{i+1} - g \\sum_{i} \\sigma^z_i

Parameters
----------
L : int
  Number of sites.
J, g : float
  Coupling parameters of the above defined Hamiltonian.
device : str
  The device to store the tensors on, e.g., 'cpu' or 'cuda'.
"""

def __init__(self, L: int, J: float, g: float, device: str = 'cpu'):
  self.L, self.d = L, 2
  self.J, self.g = J, g
  self.device = device

  # Define Pauli matrices as torch tensors
  self.sigmax = torch.tensor([[0., 1.], [1., 0.]], dtype=torch.float64, device=self.device)
  self.sigmay = torch.tensor([[0., -1j], [1j, 0.]], dtype=torch.complex128, device=self.device)
  self.sigmaz = torch.tensor([[1., 0.], [0., -1.]], dtype=torch.float64, device=self.device)
  self.id = torch.eye(2, dtype=torch.float64, device=self.device)

  self.H_bonds = self._init_H_bonds()
  self.H_mpo = self._init_H_mpo()

def _init_H_bonds(self) -> List[torch.Tensor]:
  """Initialize `H_bonds` hamiltonian. Called by __init__()."""
  sx, sz, id = self.sigmax, self.sigmaz, self.id
  d = self.d
  H_list = []

  for i in range(self.L - 1):
    gL = gR = 0.5 * self.g
    # For finite systems, the boundary terms are different
    if i == 0:
      gL = self.g
    if i == self.L - 2: # Corrected boundary condition for the last bond
      gR = self.g

    # Construct the two-site Hamiltonian term for the bond
    H_bond = -self.J * torch.kron(sx, sx) - gL * torch.kron(sz, id) - gR * torch.kron(id, sz)

    # H_bond has legs (i out, j out, i in, j in)
    H_list.append(H_bond.reshape(d, d, d, d))

  return H_list


def _init_H_mpo(self) -> List[torch.Tensor]:
  """Initialize `H_mpo` representation. Used for DMRG."""
  w_list = []
  for i in range(self.L):
    # MPO tensor for the bulk
    w = torch.zeros((3, 3, self.d, self.d), dtype=torch.float64, device=self.device)
    w[0, 0] = self.id
    w[2, 2] = self.id
    w[0, 1] = self.sigmax
    w[0, 2] = -self.g * self.sigmaz
    w[1, 2] = -self.J * self.sigmax
    #  W = np.array([[id, sx, -g*sz], [zeros, zeros, -J*sx], [zeros, zeros, id]])
    w_list.append(w)

  return w_list

def energy(self, psi: MPS) -> torch.Tensor:
  """Evaluate energy E = <psi|H|psi> using the bond representation."""
  assert psi.L == self.L
  bond_energies = psi.bond_expectation_value(self.H_bonds)
  return torch.sum(torch.stack(bond_energies))

def energy_mpo(self, psi: MPS) -> torch.Tensor:
    """Evaluate energy E = <psi|H_mpo|psi> using the MPO representation."""
    assert psi.L == self.L
    return psi.expectation_value_mpo(self.H_mpo)
```

```python
"""
Toy code implementing the density-matrix renormalization group (DMRG).
This version is a faithful port of the original NumPy/SciPy implementation,
adapted to work with a PyTorch MPS object for comparison purposes.
"""

import torch
import numpy as np
import scipy.sparse
import scipy.sparse.linalg._eigen.arpack as arp
from src.a_mps_torch import MPS, split_truncate_theta
from src.b_model_torch import TFIModel
from typing import List

class HEffective(scipy.sparse.linalg.LinearOperator):
    """
    Class for the effective Hamiltonian, directly adapted from the original NumPy version.
    It defines the matrix-vector product for use with scipy's iterative eigensolvers.
    """
    def __init__(self, LP: np.ndarray, RP: np.ndarray, W1: np.ndarray, W2: np.ndarray):
        self.LP = LP  # vL wL vL*
        self.RP = RP  # vR* wR vR
        self.W1 = W1  # wL wC i i*
        self.W2 = W2  # wC wR j j*
        chi_left, _, _ = LP.shape
        chi_right, _, _ = RP.shape
        _, _, d, _ = W1.shape
        self.theta_shape = (chi_left, d, d, chi_right)
        self.shape = (chi_left * d * d * chi_right, chi_left * d * d * chi_right)
        self.dtype = W1.dtype

    def _matvec(self, theta_flat: np.ndarray) -> np.ndarray:
        """Calculates the matrix-vector product H_eff * theta using NumPy."""
        theta = theta_flat.reshape(self.theta_shape)
        x = np.tensordot(self.LP, theta, axes=([2], [0]))
        x = np.tensordot(x, self.W1, axes=([1, 2], [0, 3]))
        x = np.tensordot(x, self.W2, axes=([3, 1], [0, 3]))
        x = np.tensordot(x, self.RP, axes=([1, 3], [2, 1]))
        return x.reshape(-1)

class DMRGEngine:
    """
    DMRG algorithm adapted to operate on a PyTorch MPS object.
    The internal calculations are performed in NumPy for stability.
    """
    def __init__(self, psi: MPS, model: TFIModel, chi_max: int = 100):
        self.psi = psi
        self.model = model
        self.chi_max = chi_max
        self.LPs = [None] * self.psi.L
        self.RPs = [None] * self.psi.L
        self._initialize_environments()

    def _initialize_environments(self):
        """Pre-calculates the right environments before the first sweep."""
        H_mpo_np = [w.cpu().numpy() for w in self.model.H_mpo]
        D = H_mpo_np[-1].shape[1]
        chi = self.psi.Bs[-1].shape[2]
        RP = np.zeros((chi, D, chi), dtype=np.float64)
        RP[0, D - 1, 0] = 1.0
        self.RPs[-1] = RP
        for i in range(self.psi.L - 1, 0, -1):
            self._update_RP(i)

    def sweep(self):
        """Performs a single DMRG sweep (left-to-right and right-to-left)."""
        for i in range(self.psi.L - 1):
            self._update_bond(i)
        for i in range(self.psi.L - 2, -1, -1):
            self._update_bond(i)

    def _update_bond(self, i: int):
        j = i + 1

        if self.LPs[i] is None:
            D = self.model.H_mpo[0].cpu().numpy().shape[0]
            chi = self.psi.Bs[i].detach().cpu().numpy().shape[0]
            LP = np.zeros((chi, D, chi), dtype=np.float64)
            LP[0, 0, 0] = 1.0
            self.LPs[i] = LP

        Heff = HEffective(self.LPs[i], self.RPs[j], self.model.H_mpo[i].cpu().numpy(), self.model.
    H_mpo[j].cpu().numpy())

        theta0_torch = self.psi.get_theta2(i)
        theta0_np = theta0_torch.detach().cpu().numpy().reshape(-1)

        e, v = arp.eigsh(Heff, k=1, which='SA', v0=theta0_np.astype(Heff.dtype),
    return_eigenvectors=True)

        theta_np = v[:, 0].reshape(Heff.theta_shape)

        theta_torch = torch.from_numpy(theta_np)
        Ai, Sj, Bj = split_truncate_theta(theta_torch, self.chi_max, eps=1.e-14)

        # Update MPS tensors using the logic from the original TEBD code
        # This ensures the MPS remains in a manageable (though not strictly canonical) form
        Ss_i_inv_np = np.diag(1. / self.psi.Ss[i].detach().cpu().numpy())
        Gi_np = np.tensordot(Ss_i_inv_np, Ai.cpu().numpy(), axes=([1], [0]))

        self.psi.Bs[i] = torch.from_numpy(np.tensordot(Gi_np, np.diag(Sj.cpu().numpy()), axes=([2],
     [0])))
        self.psi.Ss[j] = Sj
        self.psi.Bs[j] = Bj

        self._update_LP(i)
        self._update_RP(j)

    def _update_RP(self, i: int):
        """Calculate RP right of site `i-1` from RP right of site `i`."""
        j = i - 1
        B = self.psi.Bs[i].detach().cpu().numpy()
        W = self.model.H_mpo[i].cpu().numpy()
        RP_i = self.RPs[i]

        temp = np.tensordot(B, RP_i, axes=([2], [0]))
        temp = np.tensordot(temp, W, axes=([1, 2], [3, 1]))
        RP_j = np.tensordot(temp, B.conj(), axes=([1, 3], [2, 1]))
        self.RPs[j] = RP_j

    def _update_LP(self, i: int):
        """Calculate LP left of site `i+1` from LP left of site `i`."""
        j = i + 1

        # This function requires a left-canonical 'A' tensor.
        # We calculate it on the fly from the 'B' and 'S' tensors.
        B_i_np = self.psi.Bs[i].detach().cpu().numpy()
        Ss_i_np = self.psi.Ss[i].detach().cpu().numpy()
        Ss_j_np = self.psi.Ss[j].detach().cpu().numpy()

```

```python
        G = np.tensordot(B_i_np, np.diag(1./Ss_j_np), axes=([2],[0]))
        A = np.tensordot(np.diag(Ss_i_np), G, axes=([1],[0]))
        Ac = A.conj()

        W = self.model.H_mpo[i].cpu().numpy()
        LP_i = self.LPs[i]

        temp = np.tensordot(LP_i, A, axes=([2], [0]))
        temp = np.tensordot(W, temp, axes=([0, 3], [1, 2]))
        LP_j = np.tensordot(Ac, temp, axes=([0, 1], [2, 1]))
        self.LPs[j] = LP_j
```

```python
"""
Toy code implementing the time evolving block decimation (TEBD), ported to PyTorch.
"""

import torch
from src.a_mps_torch import MPS, split_truncate_theta
from src.b_model_torch import TFIModel
from typing import List

def calc_U_bonds(model: TFIModel, dt: float) -> List[torch.Tensor]:
    """
    Given a model, calculate U_bonds[i] = expm(-dt*model.H_bonds[i]).

    Each local operator has legs (i out, (i+1) out, i in, (i+1) in).
    Note that no imaginary 'i' is included, thus real `dt` means imaginary time evolution!
    """
    H_bonds = model.H_bonds
    d = H_bonds[0].shape[0]
    U_bonds = []
    for H in H_bonds:
        # Reshape H to a matrix to exponentiate
        H_matrix = H.reshape(d * d, d * d)
        # Use torch.matrix_exp for the matrix exponential
        U = torch.matrix_exp(-dt * H_matrix)
        U_bonds.append(U.reshape(d, d, d, d))
    return U_bonds


def run_TEBD(psi: MPS, U_bonds: List[torch.Tensor], N_steps: int, chi_max: int, eps: float):
    """
    Evolve the state `psi` for `N_steps` time steps with (first order) TEBD.
    The state psi is modified in place.
    """
    Nbonds = psi.L - 1
    assert len(U_bonds) == Nbonds
    for n in range(N_steps):
        # Apply gates to even bonds, then odd bonds
        for k in [0, 1]:  # 0 for even, 1 for odd
            for i_bond in range(k, Nbonds, 2):
                update_bond(psi, i_bond, U_bonds[i_bond], chi_max, eps)


def update_bond(psi: MPS, i: int, U_bond: torch.Tensor, chi_max: int, eps: float):
    """Apply `U_bond` acting on i,j=(i+1) to `psi`."""
    j = i + 1

    # 1. Construct the two-site wavefunction theta
    theta = psi.get_theta2(i)  # Legs: vL, i, j, vR

    # 2. Apply the two-site gate U_bond
    # U_bond legs: i, j, i*, j*
    # theta legs: vL, i, j, vR
    # Contraction: U_bond[i,j,i*,j*] * theta[vL,i*,j*,vR]
    Utheta = torch.tensordot(U_bond, theta, dims=([2, 3], [1, 2]))  # Legs: i, j, vL, vR

    # Transpose to bring virtual legs to the outside
    Utheta = Utheta.permute(2, 0, 1, 3)  # Legs: vL, i, j, vR

    # 3. Split and truncate using SVD
    Ai, Sj, Bj = split_truncate_theta(Utheta, chi_max, eps)

    # 4. Put the new tensors back into the MPS
    # This step updates the MPS tensors while maintaining the canonical form structure.
    # We need to absorb the inverse of the old Schmidt values on the left
    # and multiply by the new ones on the right.
    Gi = torch.tensordot(torch.diag(1. / psi.Ss[i]), Ai, dims=([1], [0]))
    psi.Bs[i] = torch.tensordot(Gi, torch.diag(Sj), dims=([2], [0]))
    psi.Ss[j] = Sj
    psi.Bs[j] = Bj


def example_TEBD_gs_finite(L: int, J: float, g: float):
    """Example of finding the ground state using imaginary time evolution with TEBD."""
    print("finite TEBD (imaginary time evolution) with PyTorch")
    print(f"L={L}, J={J:.1f}, g={g:.2f}")

    import src.a_mps_torch

    model = TFIModel(L, J=J, g=g)
    psi = src.a_mps_torch.init_spinup_MPS(L)

    for dt in [0.1, 0.01, 0.001, 1.e-4, 1.e-5]:
        U_bonds = calc_U_bonds(model, dt)
        run_TEBD(psi, U_bonds, N_steps=100, chi_max=30, eps=1.e-12)
        E = model.energy(psi)
        print(f"dt = {dt:.5f}: E = {E.item():.13f}")

    print("Final bond dimensions: ", psi.get_chi())

    # For small systems, compare to exact diagonalization (requires porting tfi_exact or using
    original)
    # E_exact = ...
    # print(f"Exact diagonalization: E = {E_exact:.13f}")
    # print(f"Relative error: {abs((E.item() - E_exact) / E_exact)}")

    return E, psi, model


if __name__ == "__main__":
    example_TEBD_gs_finite(L=14, J=1., g=1.5)
```