Neural Networks as Quantum States

NNs in Quantum Many-Body Problems

Deep Learning con Applicazioni

Stefano Lonardoni

18 july, 2025

UNIVERSITÀ
DEGLI STUDI
DI MILANO

Introduction

Neural Networks as Quantum States

- Quantum-many-body problems feature an exponential Hilbert-space growth
- Traditional variational ansätze reduce complexity sactificing correlation and entanglement
- Neural-network quantum states can offer expressivity with a compact set of parameters

Neural Networks can represent ground states for a many-body quantum system[1]

[1] Giuseppe Carleo, Matthias Troyer, Solving the quantum many-body problem with artificial neural networks. *Science* 355, 602-606 (2017)

Project Outline

Find the Ground State

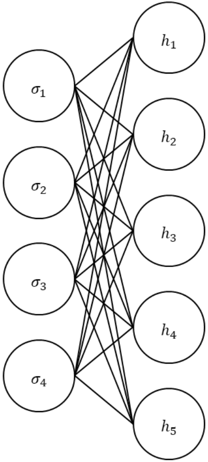
- Restricted Boltzmann Machine as a neural network
- Sample configurations from RBM
- Train RBM to find the ground state

UNIVERSITÀ DEGLI STUDI DI MILANO

Restricted Boltzmann Machine

Generative Model

- Learns $p(\vec{\sigma})$ of the input data $\vec{\sigma}$
- Two layers: N visible and M hidden
- Parameters $\mathcal{W}$: weights W_{ij} and biases b_i, c_j



Restricted Boltzmann Machine

Probability Distribution

Internal Energy:

$$E(\vec{\sigma}, \vec{h}) = -\sum_i b_i \sigma_i - \sum_j c_j h_j - \sum_{i,j} W_{ij} \sigma_i h_j$$

Probability Distribution:

$$p(\vec{\sigma}, \vec{h}) = \frac{1}{Z} e^{-E(\vec{\sigma}, \vec{h})}$$

where Z is the partition function:

$$Z = \sum_{\vec{\sigma}, \vec{h}} e^{-E(\vec{\sigma}, \vec{h})}$$

Restricted Boltzmann Machine

Application to Quantum States

Considering a spin configuration $\vec{\sigma}$ and a set of parameters $\mathcal{W}$:

$$\psi\left(\vec{\sigma}, \mathcal{W}\right) = e^{\sum_i b_i \sigma_i + \sum_j c_j h_j + \sum_{i,j} W_{ij} \sigma_i h_j}$$

With no intra-layer connections:

$$\psi\left(\vec{\sigma}, \mathcal{W}\right) = e^{\sum_i b_i \sigma_i} \times \prod_{j=1}^{M} 2cosh\left[c_j + \sum_i W_{ij} \sigma_i\right]$$

From the RBM we can sample configurations $\vec{\sigma}$.

Training

Reinforcement Learning

Update parameters $\mathcal{W}$ towards energy minimum.

$$E_0 \leq \frac{\langle \psi_{\mathcal{W}} | \hat{H} | \psi_{\mathcal{W}} \rangle}{\langle \psi_{\mathcal{W}} | \psi_{\mathcal{W}} \rangle}$$

The parameters $\mathcal{W}$ are optimized using a gradient descent method:

- Variational Monte Carlo (VMC)[2]
- Stochastic Reconfiguration (SR)[3]

---

[2] Moritz Reh, Markus Schmitt, Martin Gärttner, Optimizing design choices for neural quantum states, *Phys. Rev. B 107*, 195115 (2023)

[3] Becca F, Sorella S. *Quantum Monte Carlo Approaches for Correlated Systems*. Cambridge University Press; 2017

Training

Variational Monte Carlo

We can compute the gradients:

$$\begin{aligned}\nabla_{\mathcal{W}}\left(E\right) &= \nabla_{\mathcal{W}}\langle E_{\text{loc}}\rangle \\ &= \langle E_{\text{loc}}\nabla_{\mathcal{W}} log(p_{\psi})\rangle \\ &= 2\Re\left[\langle E_{\text{loc}}\nabla_{\mathcal{W}} log(\psi)\rangle - \langle E_{\text{loc}}\rangle\langle\nabla_{\mathcal{W}} log(\psi)\rangle\right]\end{aligned}$$

And consequently update the parameters:

$$\Delta\mathcal{W} = -\eta\nabla_{\mathcal{W}}\left(E\right)$$

Training

Stochastic Reconfiguration

Parameters evolution in the variational space:

$$\Delta\mathcal{W} = -\eta \mathbf{S}^{-1}\vec{F}$$

where $\mathbf{S}^{-1}$ is the pseudo-inverse of the covariance matrix:

$$\mathbf{S}_{ij} = \langle O_i^* O_j \rangle - \langle O_i^* \rangle \langle O_j \rangle$$

and $\vec{F}$ is the force vector:

$$F_i = \langle O_i^* E_{\text{loc}} \rangle - \langle O_i^* \rangle \langle E_{\text{loc}} \rangle$$

where O_i are the local operators defined as:

$$O_i = \frac{\partial log\left(\psi\left(\vec{\sigma}, \mathcal{W}\right)\right)}{\partial W_{ij}}$$

Code Implementation

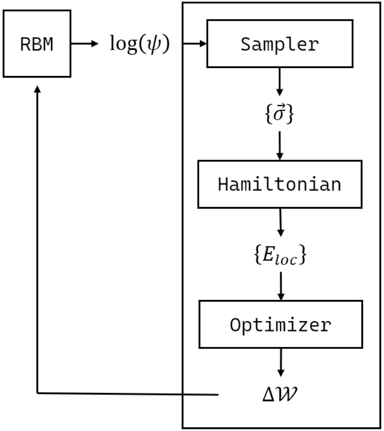
- RBM
- Sampler
- Hamiltonian
- Optimizers

UNIVERSITÀ
DEGLI STUDI
DI MILANO

Code Implementation

Code Structure

```
nnqs/
- nnqs.py             -> RBM
- hamiltonian.py      -> Ising1D
- optimizer.py        -> MRT2, GibbsSampler
- sampler.py          -> VMC, SR
```



Code Implementation
RBM

Custom `tf.module` implementing the RBM.

Weights and biases are initialized randomly as complex values:

```
self.W = tf.Variable(
    tf.random.normal([self.N_visible, self.N_hidden], dtype=tf.complex64)
)
self.b = tf.Variable(tf.random.normal([self.N_visible], dtype=tf.complex64))
self.c = tf.Variable(tf.random.normal([self.N_hidden], dtype=tf.complex64))
```

$$\psi\left(\vec{\sigma}, \mathcal{W}\right) = e^{\sum_i b_i \sigma_i} \times \prod_{j=1}^{M} 2cosh\left[c_j + \sum_i W_{ij}\sigma_i\right]$$

Code Implementation

RBM - Logarithm of the Wave Function

$$\vec{\sigma} = (\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_N) \longrightarrow \boxed{NN} \longrightarrow \log(\psi)$$

$$log\left(\psi\right) = \sum_i b_i \sigma_i + \sum_{j=1}^{M} log\left[2cosh\left(c_j + \sum_i W_{ij}\sigma_i\right)\right]$$

Code Implementation

RBM - Logarithm of the Wave Function

```
sum_visible = tf.reduce_sum(
    a * spins, axis=1
)
w_h = b + tf.matmul(spins, W)
sum_hidden = tf.reduce_sum(
    tf.math.log(2.0 * (
        tf.math.cosh(b + tf.matmul(spins, W))
    )),axis=1
)
return sum_visible + sum_hidden
```

$$\sum_i b_i \sigma_i$$

$$\sum_{j=1}^{M} log \left[2cosh \left(c_j + \sum_i W_{ij} \sigma_i \right) \right]$$

Code Implementation

Sampler

Samples configurations from the RBM.

- **Metropolis-Hastings** MCMC using *TensorFlow Probability*
- **Gibbs sampling** Double-step method to sample visible and hidden variables[4]

Provides batches of configurations. Example:

```
>>> sam.sample(rbm)
<tf.Tensor: shape=(5, 4), dtype=float32, numpy=
array([[0., 0., 0., 0.],
       [1., 1., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 1.]], dtype=float32)>
```

[4] Francesco D'Angelo, Lucas Böttcher, Learning the Ising model with generative neural networks, *Phys. Rev. Research 2*, 023266 (2020)

Code Implementation

Hamiltonian - 1D Ising

Simple 1D Ising Hamiltonian:

$$H = -J \sum_i \sigma_i \sigma_{i+1} - h \sum_i \sigma_i$$

Provides `local_energy(samples)` which computes E_{loc} for each configuration $\vec{\sigma}$.

Code Implementation

Optimizers

Leverage `tf.GradientTape` to compute gradients of $\mathit{log}\,(\psi)$

```
with tf.GradientTape() as tape:
    log_psi = self.wave_function.log_psi(samples)
grad_log_psi = tape.jacobian(log_psi, self.wave_function.trainable_variables)
```

VMC and SR methods obtain parameters update in different ways.

Update the parameters of the wave function in the same way:

```
for grad_val, var in grads_vars:
    var.assign_sub(self.learning_rate * grad_val)
```

Code Implementation

Optimizers - VMC

$$\nabla_{\mathcal{W}} \langle E \rangle = 2\Re \left[\langle E_{\text{loc}} \nabla_{\mathcal{W}} log(\psi) \rangle - \langle E_{\text{loc}} \rangle \langle \nabla_{\mathcal{W}} log(\psi) \rangle \right]$$

```python
mean_energy = tf.reduce_mean(local_energies)
for grad_i in grad_log_psi:
    loc_grad = e_loc * grad_i
    grad_energy = tf.reduce_mean(loc_grad, axis=0)
    grad_log_psi_mean = tf.reduce_mean(grad_i, axis=0)

    vmc_grad = 2.0 * tf.math.real(
        grad_energy - mean_energy * grad_log_psi_mean
    )
    gradients.append(vmc_grad)
```

Code Implementation

Optimizers - Stochastic Reconfiguration

The SR approach computes the variational increments as follows:

```
# compute covariance matrix / QGT
O_mean = tf.reduce_mean(O, axis=0)
O_centered = O - O_mean
O_O = tf.matmul(
    O_centered, O_centered, transpose_a=True
)
S = O_O / norm
```

$$S_{ij} = \langle O_i^* O_j \rangle - \langle O_i^* \rangle \langle O_j \rangle$$

Code Implementation

Optimizers - Stochastic Reconfiguration

$$F = \langle E_{\text{loc}} \nabla_{\mathcal{W}} log(\psi) \rangle - \langle E_{\text{loc}} \rangle \langle \nabla_{\mathcal{W}} log(\psi) \rangle$$

```
# compute force vector
mean_energy = tf.reduce_mean(local_energies)
F_vec = (
    tf.reduce_mean(local_energies * O, axis=0) - mean_energy * O_mean
)
```

Code Implementation

Optimizers - Stochastic Reconfiguration

$$\Delta \mathcal{W} = S^{-1} F$$

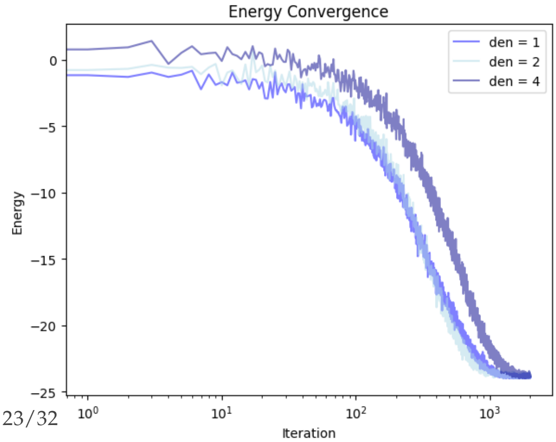
```
# solve for delta
P = tf.shape(S)[0]
S_reg = S + self.epsilon * tf.eye(P, dtype=S.dtype)
# dW = S^{-1} F
gradients = tf.linalg.solve(S_reg, tf.expand_dims(F_vec, 1))
```

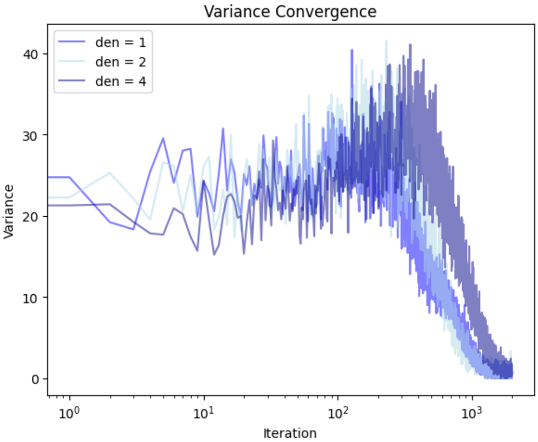
Results

UNIVERSITÀ
DEGLI STUDI
DI MILANO

Results
1D Ising Model - 16 spins

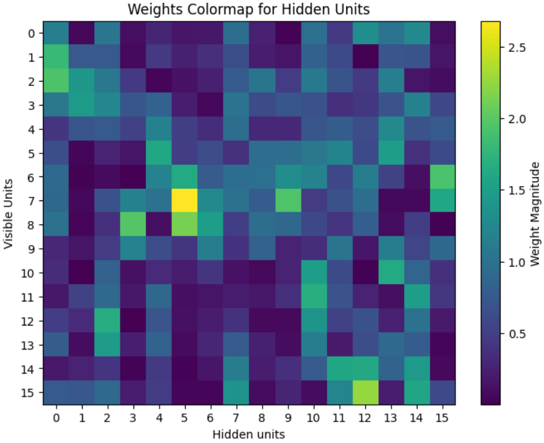
Using $J = 1.0$ and $h = 0.5$





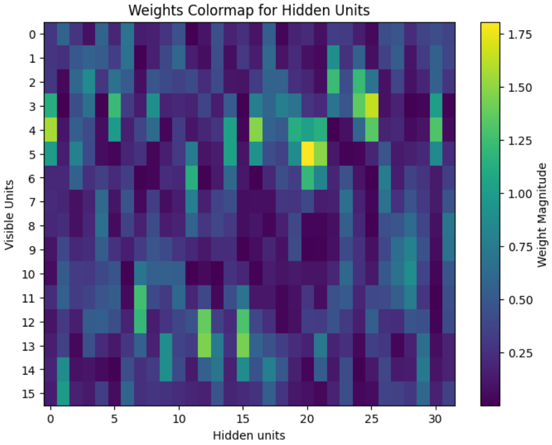
Results

Weights for $den = 1$



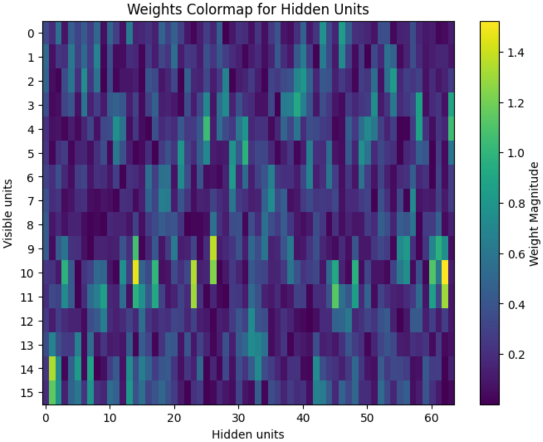
Results

Weights for $den = 2$



Results

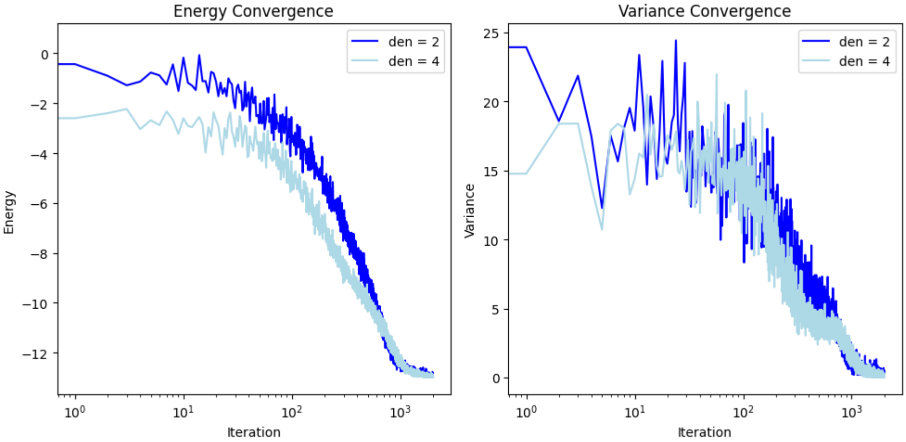
Weights for $den = 4$



Results

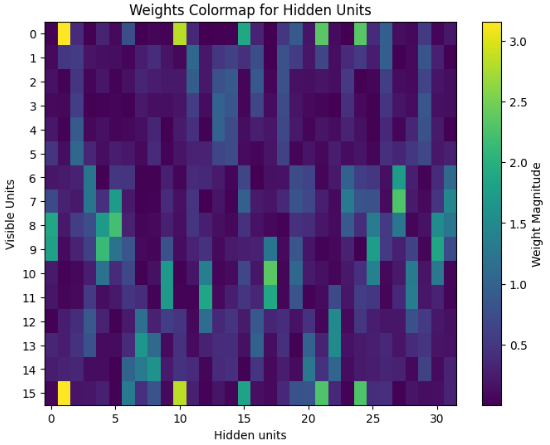
1D Ising Model - 16 spins

Using $J = -1.0$ and $h = 0.5$



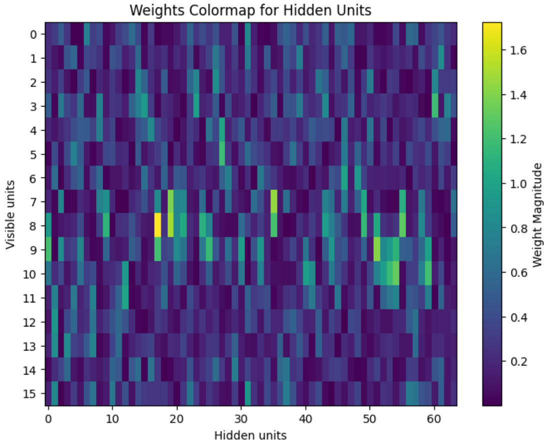
Results

Weights for $den = 2$



Results

Weights for $den = 4$



Further Work

Optimization and Extensions

Extensions

- Extend to 2D systems
- Other neural networks (FFNN, CNN)
- Unitary (and non) dynamics

Optimization

- Efficient SR update (single parameter updates)
- Change tensor framework (JAX, ...)

Other Projects

NetKet: The Machine-Learning toolbox for Quantum Physics

References

- Giuseppe Carleo, Matthias Troyer, Solving the quantum many-body problem with artificial neural networks. *Science* 355, 602-606 (2017).
- Moritz Reh, Markus Schmitt, Martin Gärttner, Optimizing design choices for neural quantum states, *Phys. Rev. B 107*, 195115 (2023).
- Becca F, Sorella S. *Quantum Monte Carlo Approaches for Correlated Systems*. Cambridge University Press; 2017.
- Francesco D'Angelo, Lucas Böttcher, Learning the Ising model with generative neural networks, *Phys. Rev. Research 2*, 023266 (2020).

Thanks for your attention!

Appendix

Appendix
Probabilities for the RBM

Thanks to no intra-layer connections, we can factorize the joint probability:

$$p(\vec{\sigma} \mid \vec{h}) = \prod_i p(\sigma_i \mid \vec{h}) \qquad\qquad p(\vec{h} \mid \vec{\sigma}) = \prod_j p(h_j \mid \vec{\sigma})$$

where:

$$p(\sigma_i | h) = \sigma\left(b_i + \sum_j W_{ij} h_j\right)$$

$$p(h_j | \sigma) = \sigma\left(c_j + \sum_i W_{ij} \sigma_i\right)$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic or sigmoid function.

Appendix

Local operators for the RBM

Explicit form of the partial derivatives of $log\left(\psi\right)$

Appendix

Metropolis-Hastings and MCMC sampling

Propose a single bit flip, and accept it with the Metropolis criterion:

$$\alpha = \min\left(1, \frac{p(\sigma')}{p(\sigma)}\right)$$

where σ' is the proposed state and σ is the current state.

Uses the `tfp.mcmc` module for efficient sampling.

- Requires target $log(p)$
- Requires new state proposal

Appendix

Gibbs sampling

We can sample the visible and hidden variables in a double-step Gibbs sampling:

```
def sample(self, wave_function):
    v = tf.cast(self.current_state, tf.float32)
    for _ in range(self.k):
        v_complex = tf.cast(v, wave_function.W.dtype)
        v_W = tf.matmul(v_complex, wave_function.W)
        p_h = tf.sigmoid(tf.math.real(wave_function.b + v_W))
        h = tf.cast(tf.random.uniform(tf.shape(p_h), dtype=tf.float32) < p_h, tf.
    float32)
        h_complex = tf.cast(h, wave_function.W.dtype)
        h_Wt = tf.matmul(h_complex, tf.transpose(wave_function.W))
        p_v = tf.sigmoid(tf.math.real(wave_function.a + h_Wt))
        v = tf.cast(tf.random.uniform(tf.shape(p_v), dtype=tf.float32) < p_v, tf.
    float32)
    self.current_state.assign(tf.cast(v, tf.int32))
return v
```