1 Method & implementations

1.1 Ising model

As presented in the theory section (**??**), we will implement the Ising model, and evaluate this to get estimates of the local energy, that we are looking to minimize. The following shows how our simple 1D Ising model can be implemented in Python, with periodic boundary conditions

```
spin_configuration = ...
spin_configuration = np.append(spin_configuration, spin_configuration[0])
J = ...
mu = ...
energy = 0
for idx in range(len(spin_configuration) - 1):
    energy -= J * spin_configuration[idx] * spin_configuration[idx + 1] + mu * spin_configuration[idx]
```

and for the long-range interaction case

```
spin_configuration = ...
J = ...
mu = ...
alpha = ...
nv = ...
energy = 0
def Jij(i, j, alpha, nv):
    dist = np.abs(i - j) ** alpha
    if dist > nv:
        return 0
    return J / dist

for idx in range(len(spin_configuration) - 1):
    energy -= mu * spin_configuration[idx]
    for jdx in range(idx+1, len(spin_configuration))
        energy -= Jij(idx, jdx, alpha, nv) * spin_configuration[idx] * spin_configuration[jdx]
```

where the choice of J, μ, α and n_v are parameters that we can tune to our liking.

1.2 Monte-Carlo sampler

One of the most important implementations is the Monte-Carlo sampler, this will allow us to both evaluate the local energy that we are hoping to minimize, as well as finding the gradients needed to achieve this minimization, by optimizing the network parameters. As an example, the following code snippet shows a simple MCMC sampling implementation

```
import numpy as np
n_steps = ...
curent_configuration = ...
grads = []
local_energies = []
for i in range(n_steps):
    old_probability = RBM.marginal_probability(curent_configuration) ** 2
    new_configuration = propose_new_configuration(curent_configuration)
    new_probability = RBM.marginal_probability(new_configuration) ** 2
    acceptance_ratio = min(1, (new_probability / old_probability) ** 2)
    if np.random.rand() < acceptance_ratio:
        curent_configuration = new_configuration
    grads.append(RBM.local_energy_gradients(curent_configuration))
    local_energies.append(RBM.local_energy(curent_configuration))
grads = np.mean(grads, axis=0)
local_energies = np.mean(local_energies)
```

This is the general structure we wish to implement for our project, following the steps introduced in the theory sections. We will use this to evaluate the local energy and the gradients of the local energy, which we will use to optimize the network parameters.

1.3 Restricted Boltzmann machine

As presented in section (**??**), we will set up our RBM neural network using a class architecture in Python. This is an efficient way for us to easily store and manipulate weights and biases of a given network instance. Initializing the weights and biases, as well as the two layers, are easily done using the Numpy library[**harris2020array**], and the following code snippet illustrates how this is done.

```
    import numpy as np
    n_visible = ...
    n_hidden = ...
    self.W = np.random.randn(n_visible, n_hidden) * 0.1
```

```
    self.a = np.random.randn(n_visible) * 0.1
    self.b = np.random.randn(n_hidden) * 0.1

    visible_layer = np.random.choice([-1, 1], size=n_visible, p=[0.5, 0.5])
    hidden_layer = np.random.choice([-1, 1], size=n_hidden, p=[0.5, 0.5])
```

With this, our network is initialized with random weights and biases, and the two layers are initialized with random values sampled from the standard normal distribution (numpy.randon.randn). The properties of the RBM (probabilities etc.) are calculated easily using the formulas from section (**??**).

1.4 Optimization

Finding the gradients of the local energy is crucial for the optimization of the network parameters. Extracting the gradients from the monte-carlo sampling as shown in previous sections, we can use these to update the network parameters following the steepest descent algorithm outlined in section (**??**). To following will outline the general structure of the optimization algorithm

1: Initialize the network parameters θ
2: Initialize the learning rate η
3: Initialize the number of epochs n_{epochs}
4: **while** not converged and not $> n_{\text{epochs}}$ **do**
5: Run the Monte-Carlo sampler to obtain estimates of the gradients of the local energy
6: Update the network parameters $\theta \leftarrow \theta - \eta \nabla_\theta E_{\text{local}}$
7: Calculate the local energy E_{local}, and check for convergence
8: **end while**