ex08_solution

April 8, 2025

Computational Quantum Physics - PHYS 463

Lecturer: Prof. G. Carleo

Assistants: alessandro.sinibaldi@epfl.ch, linda.mauron@epfl.ch, lorenzo.fioroni@epfl.ch

1 Solutions 8 - Variational Monte Carlo for Transverse Field Ising Model with Neural Network Ansatzes

[1]:
```
# Run this if you are on Windows and you don't want to use GPU
import os

os.environ["JAX_PLATFORM_NAME"] = "cpu"
```

[2]:
```
from typing import Any, Tuple

import flax.linen as nn
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
import netket as nk
import numpy as np
```

1.1 Analytical solution

[3]:
```
# Analytical solution for the ground state energy of TFIM in 1D with PBC
# (same as past week)
# Gamma (transverse field) is in units of the interaction strength J
def ising1d_energy(L, Gamma):
    def Epsilon(k, h):
        eps = 1 + h**2 + 2 * h * np.cos(k)
        return 2 * np.sqrt(eps)

    i = np.arange(L)
    k = np.pi * (2 * i + 1) / L
    energy = Epsilon(k, Gamma).sum()
    return -0.5 * energy
```

1.1.1 Setup of objects in Netket

[4]:
```
# Number of spins
N = 20

# Number of samples at each optimization step
# If there are too few samples, the statistical fluctuation may prevent us from␣
 ↪finding an accurate solution
# After finding a solution, we may increase Ns and check if the solution changes
Ns = 1024

# How many Markov chains to run in parallel
# On GPU it can be larger
n_chains = 16

# How many samples to discard for thermalization before each optimization step
# Pay attention to the R hat (Gelman--Rubin statistic) in the VMC output
# If R hat is too high (usually we use 1.05 as a threshold), we need to␣
 ↪increase n_discard
n_discard = 128

# Graph: a chain of length N with periodic boundary conditions
g = nk.graph.Chain(N, pbc=True)
# If multi-dimensional graph
# g = nk.graph.Hypercube(length=N, n_dim=1, pbc=True)

# Hilbert space: spin-1/2, the number of spins is the same as the number of␣
 ↪nodes in the graph
hi = nk.hilbert.Spin(s=1/2, N=g.n_nodes)

# Hamiltonian: TFIM with h = 1 (transverse field)
ha = nk.operator.Ising(hilbert=hi, graph=g, h=1)
# Sampler (single-spin flips)
sa = nk.sampler.MetropolisLocal(hi, n_chains=n_chains)
# Optimizer
op = nk.optimizer.Sgd(learning_rate=0.01)
```

[5]:
```
# Compute the energy with the analytical solution
E0 = ising1d_energy(N, Gamma=1)
E0
```

[5]:
```
np.float64(-25.49098968636475)
```

[6]:
```
# Exact diagonalization
E0 = nk.exact.lanczos_ed(ha)
E0
```

[6]:
```
array([-25.49098969])
```

1.2 Problem 8.0 A first test of VMC optimization with RBM ansatz

[7]:
```python
# Before we define the MLP, if you want to try out the optimization procedure,
# we build an RBM (restricted Boltzmann machine) that is already defined in NetKet
ma_rbm = nk.models.RBM(param_dtype=jnp.float64)
# Variational state object
vs = nk.vqs.MCState(sa, ma_rbm, n_samples=Ns, n_discard_per_chain=n_discard)
# Preconditioner (accelerates the optimization)
sr = nk.optimizer.SR(diag_shift=0.01)

# VMC optimization driver
vmc = nk.VMC(ha, op, variational_state=vs, preconditioner=sr)
```

[8]:
```python
vmc.run(n_iter=300)
```

```
 0%|
 …
```

[8]:
```
()
```

[9]:
```python
rbm_energy = vs.expect(ha)
error = abs((rbm_energy.mean - E0) / E0)
print("Optimized energy and relative error: ", rbm_energy, error)
```

```
Optimized energy and relative error:  -25.4781 ± 0.0063 [ ²=0.0400, R=1.0060]
[0.00050439]
```

1.3 Problem 8.1 MLP ansatz

1.3.1 (a) Definition of MLP ansatz

[10]:
```python
# Feedforward neural network (FFNN) is just another name for multilayer perceptron (MLP)
class FFNN(nn.Module):
    # We define attributes at the module level, optionally with a default
    # This allows you to easily change some hyperparameter without redefining the whole Flax module
    #
    # Example
    # width: int = 5
    #
    # If we are unsure about the type, we can use Any, e.g.
    # dtype: Any = complex

    # Here features contain the widths of the layers in the network
```

```
    features: Tuple[int, ...]

    @nn.compact
    def __call__(self, x):
        # Implement the feedforward neural network
        for i, nf in enumerate(self.features):
            # Define a dense layer and apply it to the input x
            x = nn.Dense(nf)(x)

            # Don't add the activation function if we are at the last layer
            if i == len(self.features) - 1:
                break

            # Add a ReLU activation function when we are not at the last layer
            x = nn.relu(x)

        # Sum over the features to get a scalar wave function value,
        # and only the batch dimension remains
        # Be sure to use functions from jnp (jax.numpy), not np (the original␣
↪numpy)
        x = jnp.sum(x, axis=-1)
        return x
```

1.3.2 (b) VMC optimization with MLP ansatz

[11]:
```
n_layers = 2
n_features = 5

# Initialize the MLP ansatz
ma = FFNN((n_features,) * n_layers)

# Then just replace the RBM above with the MLP you defined
# Variational state object
vs = nk.vqs.MCState(sa, ma, n_samples=Ns, n_discard_per_chain=n_discard)
# VMC optimization driver
vmc = nk.VMC(ha, op, variational_state=vs, preconditioner=sr)
```

[12]:
```
vmc.run(n_iter=300)
```

```
  0%|                                                                        ␣
↪                                          …
```

[12]: ()

[13]:
```
ffnn_energy = vs.expect(ha)
error = abs((ffnn_energy.mean - E0) / E0)
print("Optimized energy and relative error: ", ffnn_energy, error)
```

```
Optimized energy and relative error:  -24.069 ± 0.050 [ ²=2.261, R=1.0136]
[0.05577236]
```

1.3.3 (c) VMC with different network hyperparameters

[14]:
```
# An auxillary function to help us do many tests with different hyperparameters
# Also we compare the results with and without SR
def compute_energy_ffnn(n_layers, n_features, use_sr):
    # In general, numbers of features at different layers can be different
    # Here we use the same n_features at all layers to reduce the number of␣
 ↪hyperparameters
    ma = FFNN((n_features,) * n_layers)

    vs = nk.vqs.MCState(sa, ma, n_samples=Ns, n_discard_per_chain=n_discard)
    vmc = nk.VMC(ha, op, variational_state=vs, preconditioner=sr if use_sr else␣
 ↪None)

    # Increase n_iter to get more accurate results
    vmc.run(n_iter=300)
    E = vs.expect(ha)
    return E

# Relative error
def err_rel(x, y):
    return abs((x - y) / y)
```

[15]:
```
# Results without SR
res = []
for n_layers in [1, 2, 3]:
    for n_features in [5, 10, 15, 20]:
        print(n_layers, n_features)
        E = compute_energy_ffnn(n_layers, n_features, use_sr=False)
        res.append((n_layers, n_features, E))
```

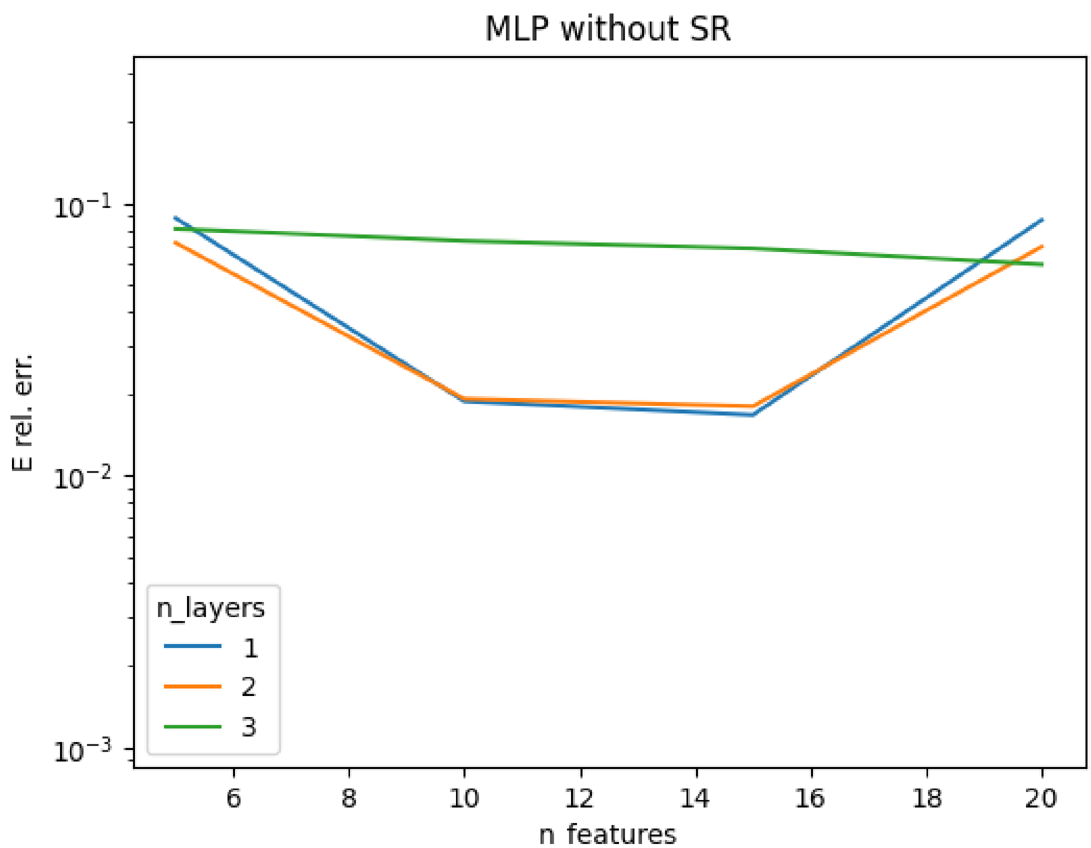
```
1 5
  0%|                                                                          ␣
 ↪                                  …
1 10
  0%|                                                                          ␣
 ↪                                  …
1 15
  0%|                                                                          ␣
 ↪                                  …
1 20
```

```
  0%|                                                                                ␣
  ↪                                  …
2 5
  0%|                                                                                ␣
  ↪                                  …
2 10
  0%|                                                                                ␣
  ↪                                  …
2 15
  0%|                                                                                ␣
  ↪                                  …
2 20
  0%|                                                                                ␣
  ↪                                  …
3 5
  0%|                                                                                ␣
  ↪                                  …
3 10
  0%|                                                                                ␣
  ↪                                  …
3 15
  0%|                                                                                ␣
  ↪                                  …
3 20
  0%|                                                                                ␣
  ↪                                  …
```

[16]:
```python
xs = [5, 10, 15, 20]
a = [err_rel(x[-1].mean, E0) for x in res[:4]]
b = [err_rel(x[-1].mean, E0) for x in res[4:8]]
c = [err_rel(x[-1].mean, E0) for x in res[8:]]

plt.plot(xs, a, label=1)
plt.plot(xs, b, label=2)
plt.plot(xs, c, label=3)

plt.title("MLP without SR")
plt.xlabel("n_features")
plt.ylabel("E rel. err.")
```

```
plt.ylim(0.85e-3, 3.5e-1)
plt.yscale("log")
plt.legend(title="n_layers")
```

[16]: <matplotlib.legend.Legend at 0x3cebbb610>



[17]:
```
# Results with SR
res_sr = []
for n_layers in [1, 2, 3]:
    for n_features in [5, 10, 15, 20]:
        print(n_layers, n_features)
        E = compute_energy_ffnn(n_layers, n_features, use_sr=True)
        res_sr.append((n_layers, n_features, E))
```

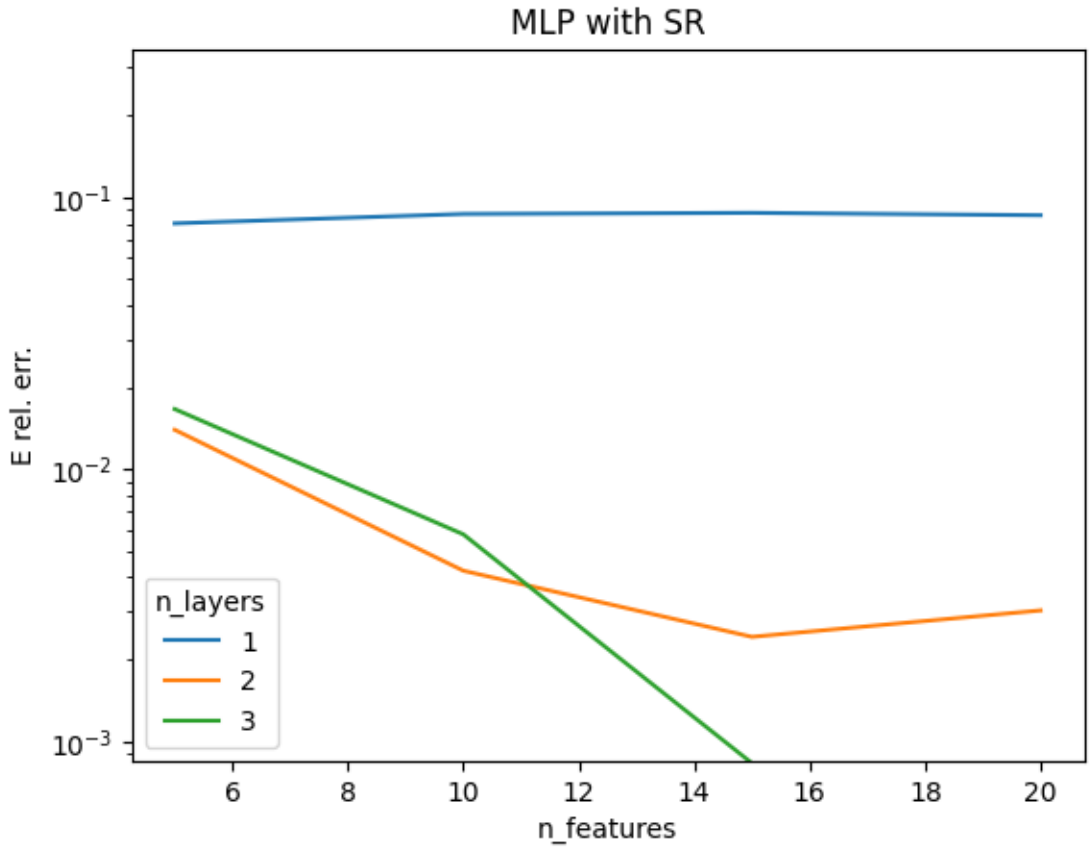
```
1 5
  0%|
↪                                         …
1 10
```

```
  0%|                                                                                ␣
  ↪                                  …
1 15
  0%|                                                                                ␣
  ↪                                  …
1 20
  0%|                                                                                ␣
  ↪                                  …
2 5
  0%|                                                                                ␣
  ↪                                  …
2 10
  0%|                                                                                ␣
  ↪                                  …
2 15
  0%|                                                                                ␣
  ↪                                  …
2 20
  0%|                                                                                ␣
  ↪                                  …
3 5
  0%|                                                                                ␣
  ↪                                  …
3 10
  0%|                                                                                ␣
  ↪                                  …
3 15
  0%|                                                                                ␣
  ↪                                  …
3 20
  0%|                                                                                ␣
  ↪                                  …
```

[18]:
```
a = [err_rel(x[-1].mean, E0) for x in res_sr[:4]]
b = [err_rel(x[-1].mean, E0) for x in res_sr[4:8]]
c = [err_rel(x[-1].mean, E0) for x in res_sr[8:]]
```

```
plt.plot(xs, a, label=1)
plt.plot(xs, b, label=2)
plt.plot(xs, c, label=3)

plt.title("MLP with SR")
plt.xlabel("n_features")
plt.ylabel("E rel. err.")
plt.ylim(0.85e-3, 3.5e-1)
plt.yscale("log")
plt.legend(title="n_layers")
```

[18]: <matplotlib.legend.Legend at 0x3d9b2e810>



We can conclude: * The results with SR are much better, because a naive gradient-based optimizer may be trapped in local minima, and SR changes the parameter space into a more natural one and reduces local minima. * More features usually lead to better results, but if the neural network is too large, it may become harder to optimize in a limited time, and the resulting energy may be higher. * Two layers perform better than one layer, and they are enough for a simple physical system like TFIM, as three layers did not improve the results much.

1.4 Problem 8.2 CNN ansatz

Look at https://github.com/8bitmp3/JAX-Flax-Tutorial-Image-Classification-with-Linen to have better examples of CNN using MNIST data for example.

1.4.1 (a) Definition of CNN ansatz

[19]:
```
class CNN(nn.Module):
    channels: Tuple[int, ...]
    kernel_sizes: Tuple[int, ...]

    @nn.compact
    def __call__(self, x):
        # Before feeding the input x into the convolutions,
        # we add a channel dimension as the last dimension
        x = jnp.expand_dims(x, axis=-1)

        for i, (nc, ks) in enumerate(zip(self.channels, self.kernel_sizes)):
            # Use circular padding because the physical system has PBC
            # If OBC, we may use zero padding
            x = nn.Conv(features=nc, kernel_size=(ks,), padding="CIRCULAR")(x)

            # Don't add the activation function if we are at the last layer
            if i == len(self.channels) - 1:
                break

            # Add a ReLU activation function when we are not at the last layer
            x = nn.relu(x)

        # Reshape to combine the spatial and the channel dimensions
        x = x.reshape((x.shape[0], -1))

        # Sum over the spatial and the channel dimensions to get a scalar wave␣
↪function value,
        # and only the batch dimension remains
        x = jnp.sum(x, axis=-1)
        return x
```

1.4.2 (b) VMC optimization

[20]:
```
# Initialize the CNN ansatz
ma = CNN((2,) * 2, (5,) * 2)

# Variational state object
vs = nk.vqs.MCState(sa, ma, n_samples=Ns, n_discard_per_chain=n_discard)
# VMC optimization driver
vmc = nk.VMC(ha, op, variational_state=vs, preconditioner=sr)
```

[21]:
```
vmc.run(n_iter=300)
```

```
  0%|                                                                         ␣
↪                                    …
```

[21]: ()

[22]:
```
cnn_energy = vs.expect(ha)
error = abs((cnn_energy.mean - E0) / E0)
print("Optimized energy and relative error: ", cnn_energy, error)
```

```
Optimized energy and relative error:  -25.405 ± 0.015 [ ²=0.193, R=1.0184]
[0.00337501]
```

1.4.3 (c) VMC with different network hyperparameters

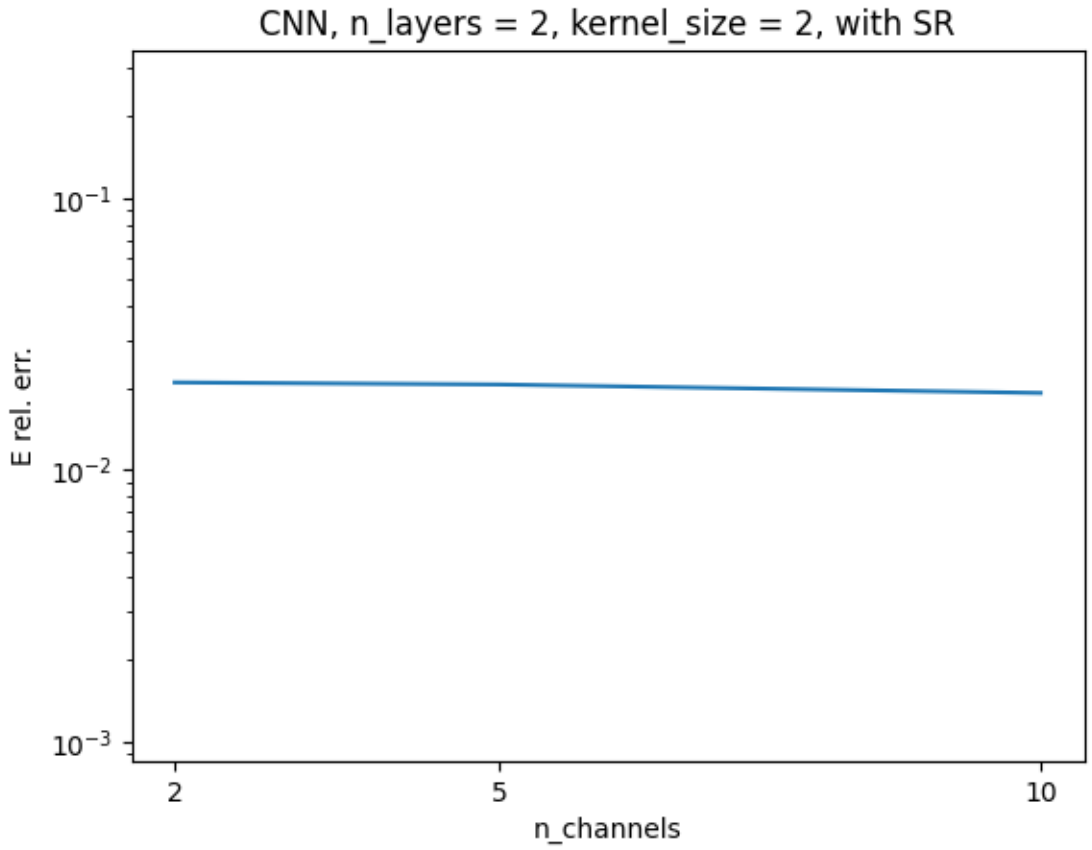
[23]:
```
def compute_energy_cnn(n_layers, n_channels, kernel_size):
    # Here we use the same n_channels and kernel_size at all layers to reduce␣
 ↪the number of hyperparameters
    ma = CNN((n_channels,) * n_layers, (kernel_size,) * n_layers)

    vs = nk.vqs.MCState(sa, ma, n_samples=Ns, n_discard_per_chain=n_discard)
    vmc = nk.VMC(ha, op, variational_state=vs, preconditioner=sr)

    # Increase n_iter to get more accurate results
    vmc.run(n_iter=100, out=None)
    E = vs.expect(ha)
    return E
```

First we keep `n_layers = 2`, `kernel_size = 2`, and try to change `n_channels`.

[24]:
```
res_cnn_nc = []
for n_channels in [2, 5, 10]:
    print(n_channels)
    E = compute_energy_cnn(n_layers=2, n_channels=n_channels, kernel_size=2)
    res_cnn_nc.append((n_channels, E))
```

```
2

  0%|                                                                         ␣
↪                                    …

5

  0%|                                                                         ␣
↪                                    …

10

  0%|                                                                         ␣
↪                                    …
```

[25]:
```
xs = [2, 5, 10]
ys = [err_rel(x[-1].mean, E0) for x in res_cnn_nc]

plt.plot(xs, ys)

plt.title("CNN, n_layers = 2, kernel_size = 2, with SR")
plt.xlabel("n_channels")
plt.ylabel("E rel. err.")
plt.xticks(xs)
plt.ylim(0.85e-3, 3.5e-1)
plt.yscale("log")
```



For a simple physical system like TFIM, two channels are enough, and increasing the number of channels will not improve the result significantly.

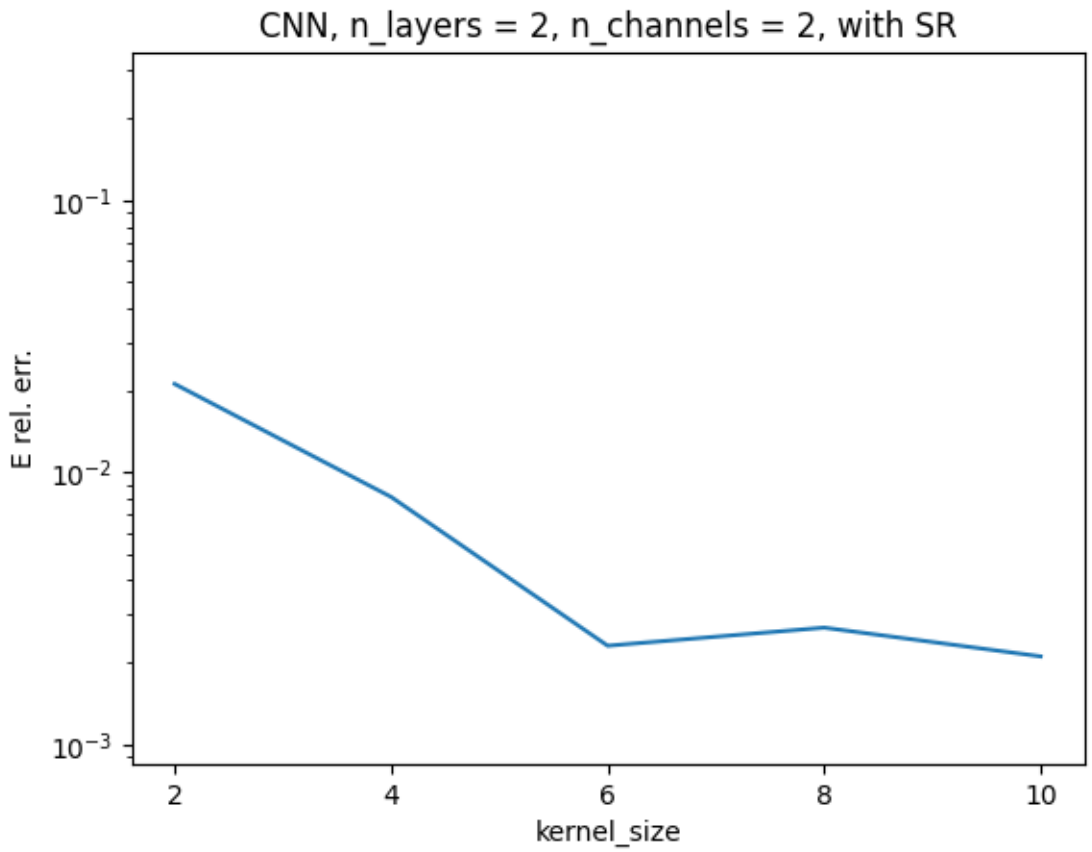
Then we keep `n_channels = 2`, and try to change `kernel_size`.

[26]:
```
res_cnn_ks = []
for kernel_size in [2, 4, 6, 8, 10]:
    print(kernel_size)
```

```
    E = compute_energy_cnn(n_layers=2, n_channels=2, kernel_size=kernel_size)
    res_cnn_ks.append((kernel_size, E))
```

```
2
  0%|                                                                                ␣
 ↪                                 …
4
  0%|                                                                                ␣
 ↪                                 …
6
  0%|                                                                                ␣
 ↪                                 …
8
  0%|                                                                                ␣
 ↪                                 …
10
  0%|                                                                                ␣
 ↪                                 …
```

[27]:
```
xs = [2, 4, 6, 8, 10]
ys = [err_rel(x[-1].mean, E0) for x in res_cnn_ks]

plt.plot(xs, ys)

plt.title("CNN, n_layers = 2, n_channels = 2, with SR")
plt.xlabel("kernel_size")
plt.ylabel("E rel. err.")
plt.xticks(xs)
plt.ylim(0.85e-3, 3.5e-1)
plt.yscale("log")
```



A larger kernel size usually leads to a better energy.

Notice that we have defined the transverse field `h = 1`, which is the critical point of 1D TFIM. At the critical point, the correlations between spins decay algebraically (in power law) with the distance, so each spin is somewhat strongly correlated with distant spins, and we need a large kernel to capture this correlation.

If `h` is away from the critical point, then the correlations decay exponentially, so each spin is only strongly correlated with nearby spins, and we may get good results with smaller kernels.

Compared to MLP, CNN can reach a lower variational energy with fewer variational parameters, because the convolutions respect the translational symmetry and the locality of the physical system.