Restricted Boltzmann machine

Artificial neural networks

Christoffer Arvidsson

Results

Parameters used to estimate the model distribution is shown in Table 1. Notably, the lower learning rate helped training the model for small number of hidden neurons.

Table 1: The parameters used in the model.

Parameter	Value
k	100
number of minibatches (trials)	3000
n_{visible}	3
n_{hidden}	{1,2,3,4,5,6,7,8}
learning rate η	0.001
minibatch size	20
generation realisations	2000
generation samples/realization	1000
averaging runs	3

The KL-divergence over number of hidden neurons is shown in Figure 1. We can see that three neurons seems to be the critical number of hidden neurons. These results do align with the theoretical upper bound, although the KL-divergence is slightly higher than this upper bound. This could be due to the CD-k algorithm converging to a suboptimal solution.

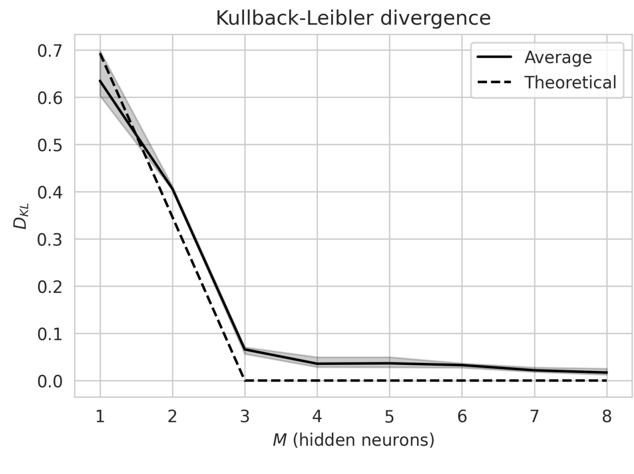


Figure 1: The KL-divergence over the number of hidden neurons in the Boltzmann machine for the XOR problem. The shaded region is the confidence interval. The dashed line is the theoretical upper bound.

Notebook

RBM implementation

```
import numpy as np

class RBM:
    def __init__(self, n_visible, n_hidden):
        self.weights = np.random.normal(0, 1, size=(n_hidden, n_visible))
        self.t_vis = np.zeros((n_visible,), dtype=float)
        self.t_hid = np.zeros((n_hidden,), dtype=float)

    def generate(self, x, num_samples):
        v_state = x.copy()
        h_state = np.zeros_like(self.t_hid, dtype=int)

        # update hidden neurons
        b_h0 = np.dot(v_state, self.weights.T) - self.t_hid
        p_b = (1+np.exp(-2*b_h0))**-1
        r = np.random.uniform(size=h_state.shape)
        h_state[r<p_b] = 1
        h_state[r>=p_b] = -1

        patterns = np.zeros((num_samples, x.shape[0]), dtype=int)

        for i in range(num_samples):
            # update visible neurons
            b_v = np.dot(h_state, self.weights) - self.t_vis
            p_b = (1+np.exp(-2*b_v))**-1
            r = np.random.uniform(size=v_state.shape)
            v_state[r<p_b] = 1
            v_state[r>=p_b] = -1

            # update hidden neurons
            b_h = np.dot(v_state, self.weights.T) - self.t_hid
            p_b = (1+np.exp(-2*b_h))**-1
            r = np.random.uniform(size=h_state.shape)
            h_state[r<p_b] = 1
            h_state[r>=p_b] = -1

            patterns[i] = v_state

        return patterns

    def run_cd_k(self, batch, k=100, learning_rate=0.1):
        dW = np.zeros_like(self.weights, dtype=float)
        dT_vis = np.zeros_like(self.t_vis, dtype=float)
        dT_hid = np.zeros_like(self.t_hid, dtype=float)
        h_state = np.zeros_like(self.t_hid, dtype=int)

        for x in batch:
            v_state = x.copy()

            # update hidden neurons
            b_h = np.dot(v_state, self.weights.T) - self.t_hid
            b_h0 = b_h.copy()
            p_b = (1+np.exp(-2*b_h))**-1
            r = np.random.uniform(size=h_state.shape)
            h_state[r<p_b] = 1
            h_state[r>=p_b] = -1

            for t in range(k):
                # update visible neurons
                b_v = np.dot(h_state, self.weights) - self.t_vis
                p_b = (1+np.exp(-2*b_v))**-1
                r = np.random.uniform(size=v_state.shape)
                v_state[r<p_b] = 1
                v_state[r>=p_b] = -1

                # update hidden neurons
                b_h = np.dot(v_state, self.weights.T) - self.t_hid
                p_b = (1+np.exp(-2*b_h))**-1
                r = np.random.uniform(size=h_state.shape)
                h_state[r<p_b] = 1
                h_state[r>=p_b] = -1

            dW += learning_rate*(np.outer(np.tanh(b_h0), x) - np.outer(np.tanh(b_h), v_state))
            dT_vis -= learning_rate*(x - v_state)
            dT_hid -= learning_rate*(np.tanh(b_h0) - np.tanh(b_h))

        self.weights += dW
        self.t_vis += dT_vis
```

```python
        self.t_hid += dT_hid
```

Dataset

```python
import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
from model import RBM
from multiprocessing import Pool
import os

sns.set_style("whitegrid")
plt.rcParams['figure.dpi'] = 300
plt.rcParams['savefig.dpi'] = 300

dataset = np.array([
    [-1,-1,-1],
    [1,-1,1],
    [-1,1,1],
    [1,1,-1],
])
```

Simulation

Define the simulation function

```python
def run_simulation(dataset, k, weight_updates, n_visible, n_hidden,
                   learning_rate, batch_size,
                   n_samples, n_realizations):

    machine = RBM(n_visible, n_hidden)

    # Training
    mus = np.random.choice(dataset.shape[0], size=(weight_updates, batch_size), replace=True)
    counts = np.zeros(dataset.shape[0])
    for i in range(weight_updates):
        mu = mus[i]
        machine.run_cd_k(dataset[mu], k=k, learning_rate=learning_rate)

    # Generation
    random_patterns = np.random.choice([-1, 1], size=(n_realizations, n_visible),
                                       replace=True)
    patterns = np.zeros((n_realizations, n_samples, n_visible),
                        dtype=int)
    for r in range(n_realizations):
        patterns[r] = machine.generate(random_patterns[r], n_samples)

    patterns[patterns == -1] = 0
    n_patterns = n_realizations*n_samples
    patterns = patterns.reshape((n_patterns,-1))
    hashes = patterns.dot(1 << np.arange(patterns.shape[-1]-1, -1, -1))
    unique, counts = np.unique(hashes, return_counts=True)

    distribution = np.zeros(2**n_visible)
    distribution[unique] = counts

    p_model = distribution/(n_patterns)

    return machine, p_model
```

Parameters used.

```python
k = 100              # monte carlo iterations
weight_updates = 3000
n_visible = 3        # N
n_hidden = np.arange(1,8+1)
learning_rate = 0.001
batch_size = 20
n_realizations = 2000
n_samples = 1000
num_processes = 12 # Number of threads you have access to on your cpu.
averaging_runs = 3
```

Let's do some multiprocessing to speed this up. This runs the simulation for each value of `n_visible`, `averaging_runs` number of times.

```python
def _pool_func(n_hidden):
   print(f'pid: {os.getpid()}\tRunning for n_hidden: {n_hidden}\n')
   return run_simulation(dataset, k, weight_updates, n_visible, n_hidden,
                         learning_rate, batch_size,
                         n_samples, n_realizations)

requests = []
p_models = np.zeros((averaging_runs, len(n_hidden), 2**n_visible), dtype=float)
with Pool(processes=num_processes) as pool:
    for r in range(averaging_runs):
        requests.append([pool.apply_async(_pool_func, (n,)) for n in n_hidden])

    for r in range(averaging_runs):
        results = [req.get() for req in requests[r]]
        # Effectively transposes our list of tuples into a tuple of lists
        _, p_model = map(list, zip(*results))
        p_models[r] = np.array(p_model)

print('Done with simulations.')
```

Now plot the KL-divergence

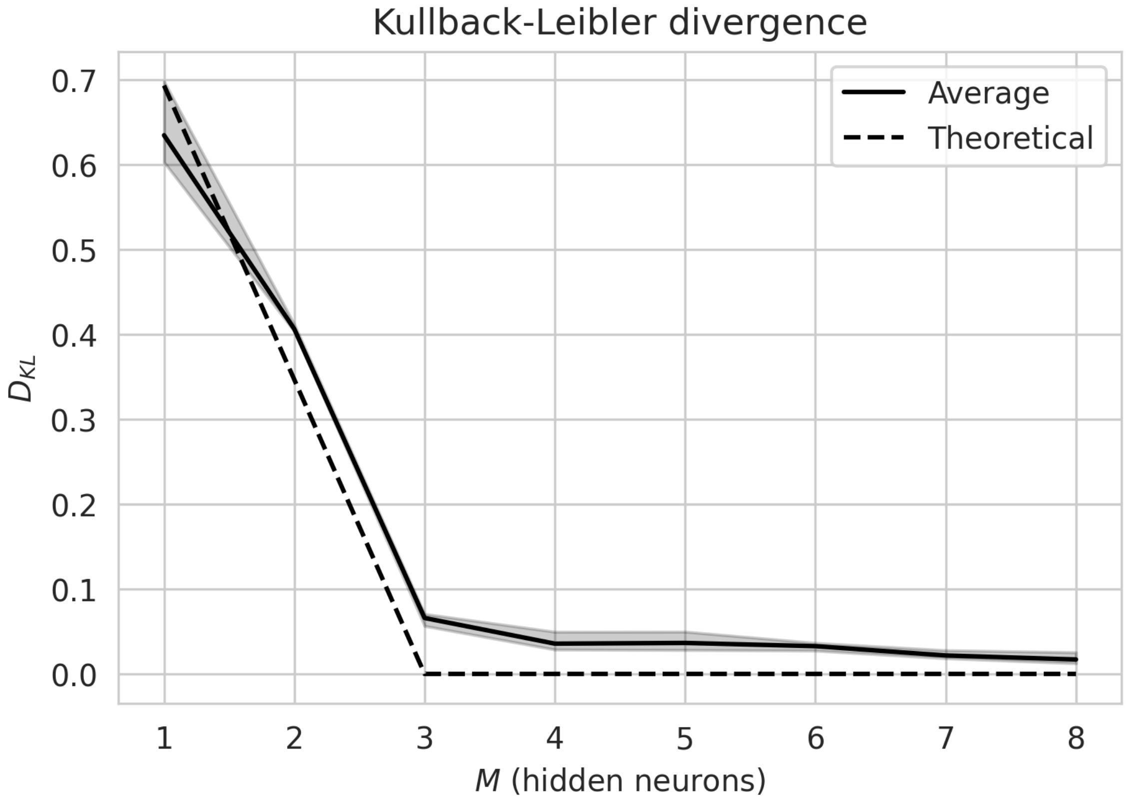
```python
xor = [0,3,5,6] # Represents indices for the xor patterns among random ones
p_data = np.zeros((2**n_visible))
p_data[xor] = 1/4

def kl_divergence_bound(n, m):
    return np.log(2) * (n - np.floor(np.log2(m + 1)) - (m+1)/(2**(np.floor(np.log2(m+1)))))

samples = []
for r in range(averaging_runs):
    for i, p_model in enumerate(p_models[r]):
        kl_divergence = np.sum(p_data[xor] * np.log(p_data[xor] / p_model[xor]))
        samples.append((r, n_hidden[i], kl_divergence))

nonzero_m = n_hidden[n_hidden < 2**(n_visible - 1) -1]
theoretical = np.zeros(len(n_hidden))
theoretical[n_hidden < 2**(n_visible - 1) -1] = kl_divergence_bound(n_visible, nonzero_m)

data = pd.DataFrame(samples, columns=['run', 'hidden_neurons', 'kl_divergence'])
g = sns.lineplot(x='hidden_neurons', y='kl_divergence', data=data, color='black', label='Average')
sns.lineplot(x=n_hidden, y=theoretical, ax=g, color='black', linestyle='--', label='Theoretical')
g.set_xlabel('$M$ (hidden neurons)')
g.set_ylabel(r'$D_{KL}$')
g.set_title('Kullback-Leibler divergence')
plt.show()
```



Kullback-Leibler divergence
Average
Theoretical
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
D_{KL}
1
2
3
4
5
6
7
8
M (hidden neurons)