17 - Probabilistic Models

Ludwig Krippahl

Probabilistic Models

Summary

- Graphical Models
- Gibbs sampling
- Restricted Boltzmann Machine

Graphical Models

Discriminative vs Generative

- Discriminative models learn conditional probabilities

$$p(y \mid x)$$

- If we want to generate examples we need to sample from the joint probability distribution

$$p(y, x)$$

- With Variational Autoencoders and Generative Adversarial Networks we mapped known distributions to the target distributions
- With probabilistic models we try to capture explicitly these joint distributions.

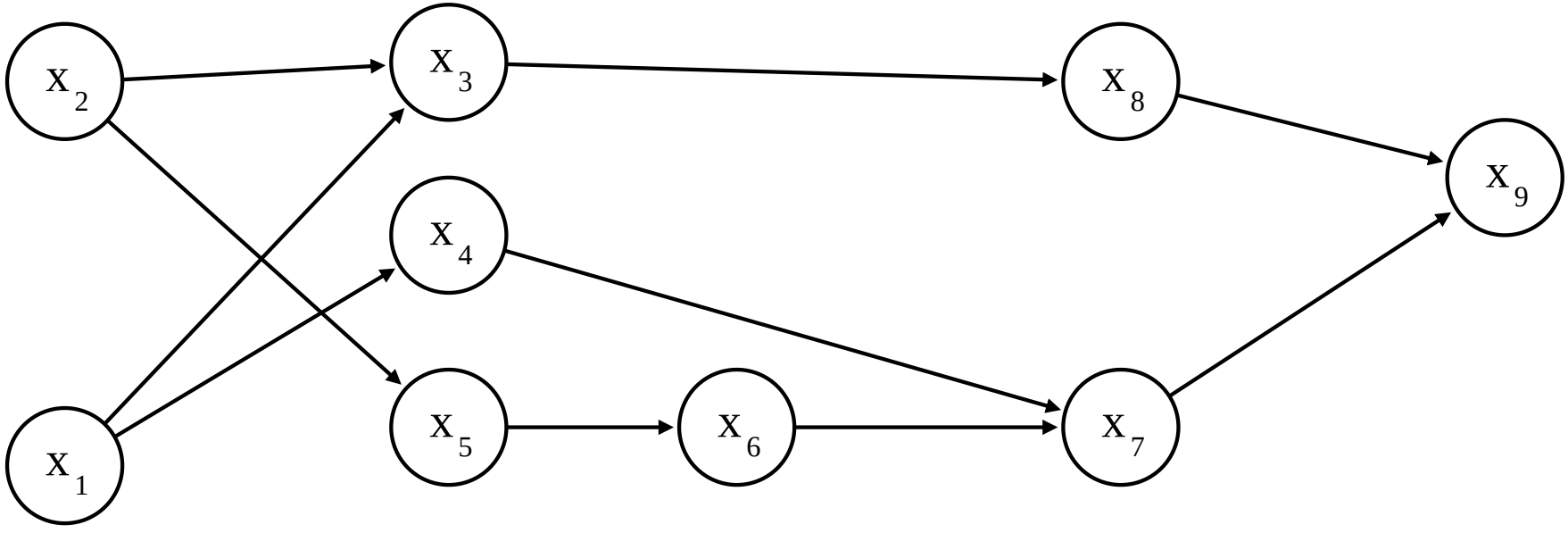
Graphical Models

- Graphical models are a practical way of representing relations between variables
- For realistic problems we cannot store or compute all combinations
  - E.g. For n variables with k values each we need k^n combinations
- Graphical models allow us to specify which variables are related and how

Bayesian Networks

- Bayesian networks are directed acyclical graphs representing conditional dependency

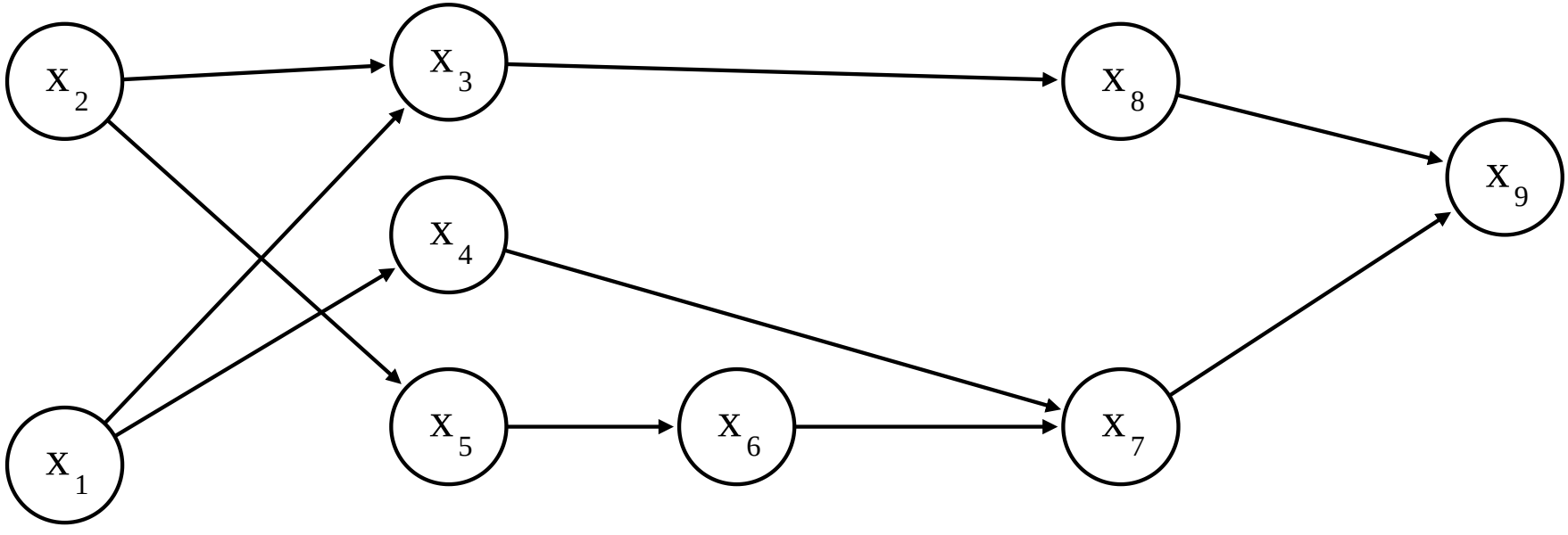
- E.g in this network, $p(x_3)$ depends on x_1 and x_2



Bayesian Networks

- The joint probability distribution can be computed from ancestors to descendants:

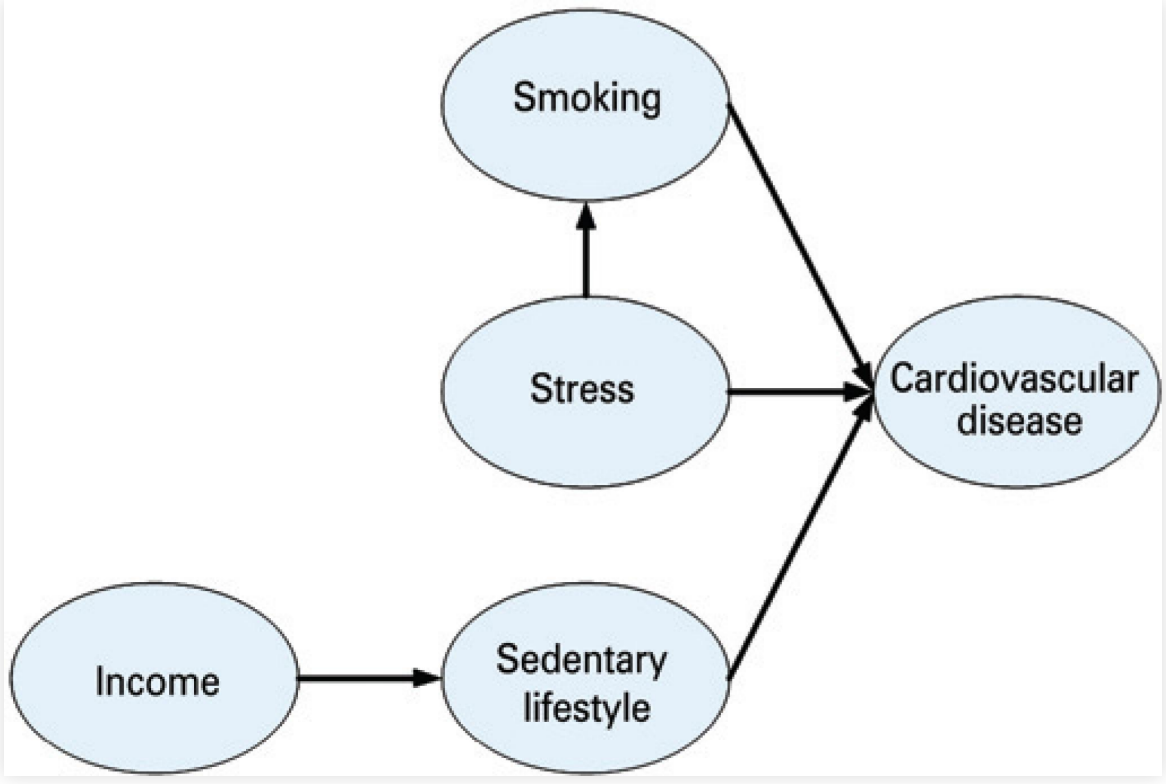
$$p(x_1, \ldots, x_9) = p(x_1)p(x_2)p(x_3 \mid x_1, x_2)p(x_4 \mid x_1)p(x_5 \mid x_2)$$
$$p(x_6 \mid x_5)p(x_7 \mid x_6, x_4)p(x_8 \mid x_3)p(x_9 \mid x_7, x_8)$$



Bayesian Networks

- Bayesian Networks are good when we know how the variables are related

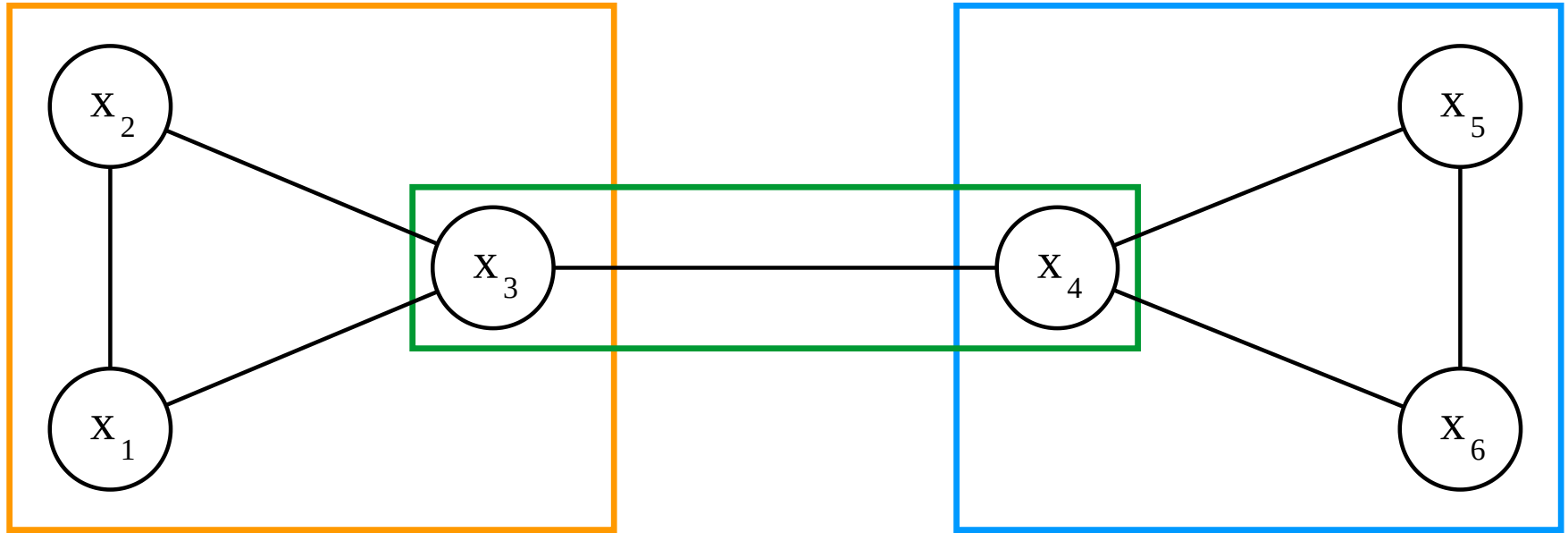
- E.g. causal relations



Sato and Sato, Probabilistic graphic models applied to identification of diseases

Markov Random Fields

- Markov Random Fields (MRF) and undirected graphs
  - Edges represent interactions between variables, but no specific direction
- Joint probability distribution is a function of cliques in the graph
  - We can consider just the maximal cliques



Markov Random Fields

- Markov Random Fields (MRF) and undirected graphs
  - Edges represent interactions between variables, but no specific direction
- Joint probability distribution is a function of cliques in the graph
  - We can consider just the maximal cliques
- If for each clique we use a nonnegative measure $\phi(C)$ of the tendency for each possible state, the unnormalized probability distribution is

$$\tilde{p}(x) = \prod_C \phi(C)$$

- Which we can normalize dividing by the partition function Z

$$p(x) = \frac{1}{Z} \prod_C \phi(C) \qquad Z = \int \tilde{p}(x) dx$$

Energy based models

- Taking inspiration from thermodynamics, we can use an "energy" function and the Boltzmann distribution:

$$\tilde{p}(x) = e^{-E(x)}$$

- Using the energy as an exponent guarantees a nonnegative function and also makes it easy to decompose contributions from different cliques

$$e^a e^b = e^{a+b}$$

- This is especially convenient when taking the logarithm of the probabilities, $log\, \tilde{p}(x)$

- Just add the energies

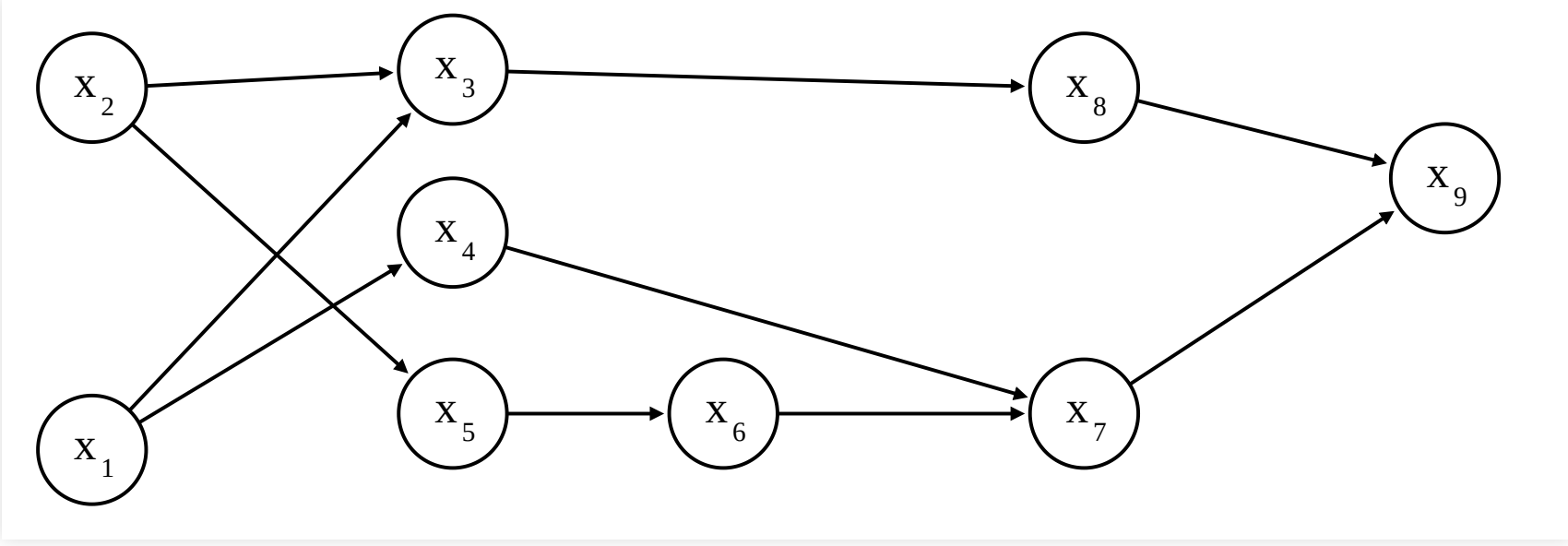
- A Markov Random Field with these energy functions is called a Boltzmann Machine

Sampling

Bayesian Networks

- Since BN are directed, we can use ancestral sampling.

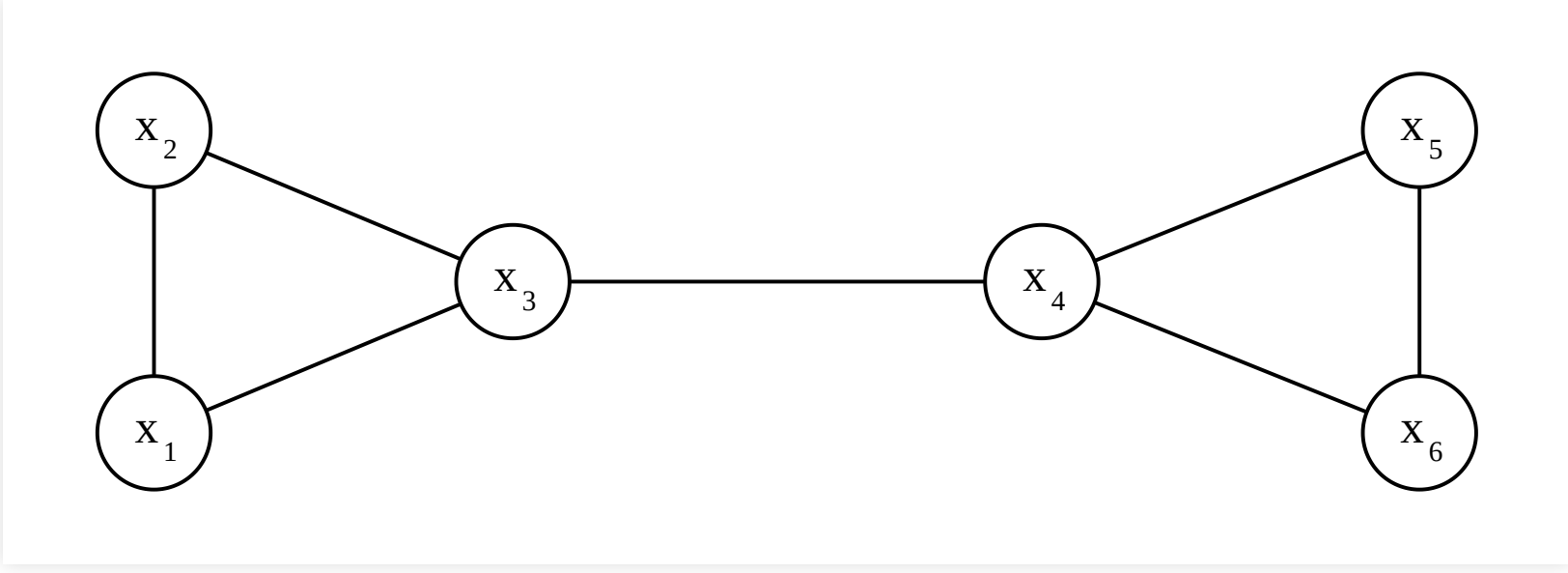
- Sort variables from ancestors to descendants
- Sample in this order
- For every variable that is conditionally dependent on some ancestor, use the values of the ancestors to determine the distribution from which to sample



Markov Random Fields

- For undirected models we can use Gibbs sampling

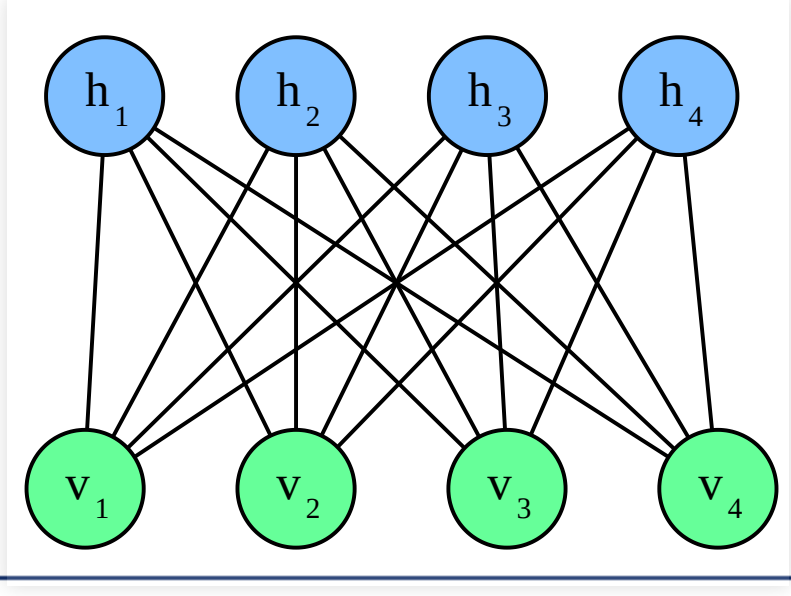
- Start with random values
- Sample from variable (or block) conditioned on neighbours
- Any variable is independent of all others given its neighbours
- Repeat until convergence



Restricted Boltzmann Machine

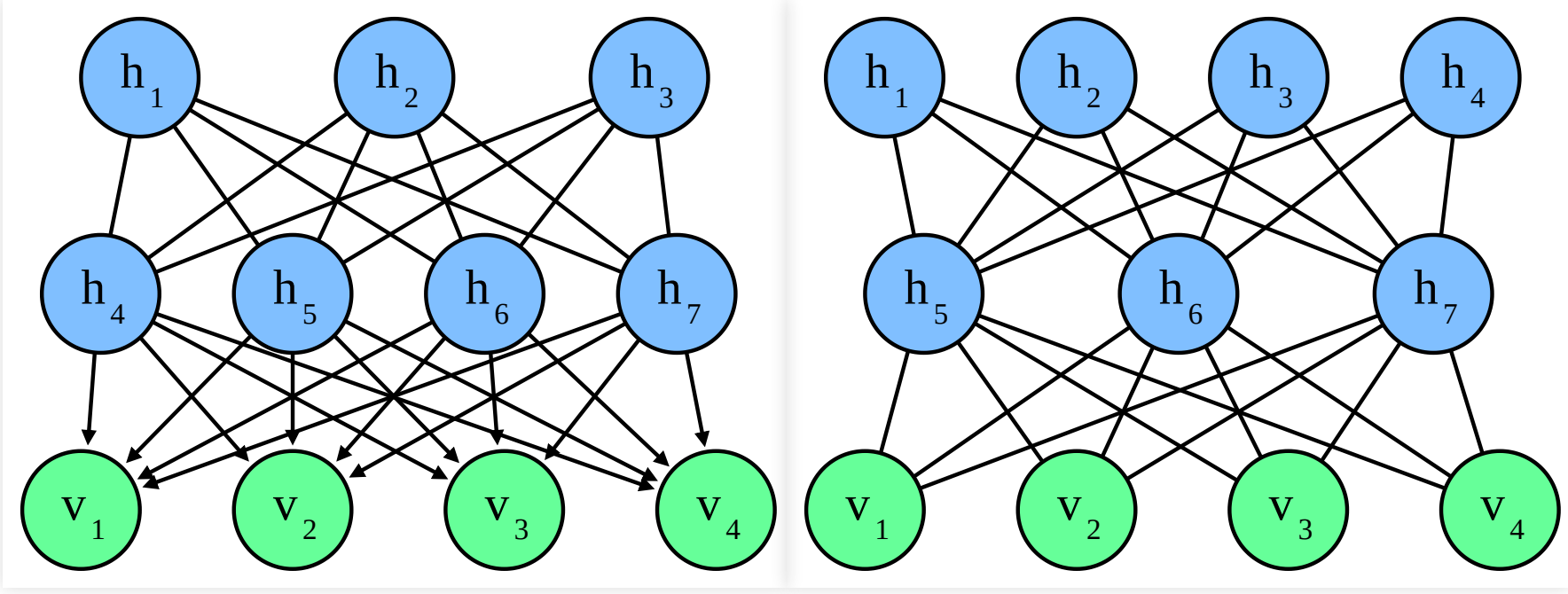
Restricted Boltzmann Machine

- A Restricted Boltzmann Machine (RBM) is a Boltzmann Machine
- A Markov Random Field using energy and the Boltzmann distribution for $\tilde{p}(x)$
- In which the graph is a bipartite graph
- Can be partitioned into two sets of nodes such that edges are only between sets
- with one set is is composed of hidden variables and the other of observable variables



Restricted Boltzmann Machine

- Restricted Boltzmann Machines are not deep models
- But they serve as the basis for deep models
- Deep Belief Networks, Deep Boltzmann Machines



Restricted Boltzmann Machine

- The probability of a given state is:

$$P(\mathrm{v} = v, \mathrm{h} = h) = \frac{1}{Z} e^{-E(v,h)} \qquad Z = \sum_v \sum_h e^{-E(v,h)}$$

- And the energy function is given by:

$$E(v,h) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_i \sum_j v_i w_{i,j} h_j$$

$$E(v,h) = -b^\top v - c^\top h - v^\top W h$$

- Where b and c are the biases for the visible and hidden variables, and W the weights matrix relating the two.

Restricted Boltzmann Machine

- Since edges only connect hidden to visible units, we can separate into two blocks

$$P(h \mid v)\$ \qquad \$P(v \mid h)$$

- For $P(h \mid v)$, considering v constant:

$$\begin{aligned} P(h \mid v) = \frac{P(h,v)}{P(v)} &= \frac{1}{P(v)} \frac{1}{Z} e^{b^\top v + c^\top h + v^\top W h} \\ &= \frac{1}{Z'} e^{c^\top h + v^\top W h} \\ &= \frac{1}{Z'} \prod_{j=1}^{n_h} e^{c_j h_j + v^\top W_{:,j} h_j} \end{aligned}$$

Restricted Boltzmann Machine

- Considering a simple RBM with binary variables:

$$P(h_j = 1 \mid v) = \frac{\tilde{P}(h_j = 1 \mid v)}{\tilde{P}(h_j = 0 \mid v) + \tilde{P}(h_j = 1 \mid v)}$$
$$= \frac{e^{c_j + v^\top W_{:,j}}}{e^0 + e^{c_j + v^\top W_{:,j}}}$$
$$= \sigma(c_j + v^\top W_{:,j})$$

- Thus:

$$P(h \mid v) = \prod_{j=1}^{n_h} \sigma\big((2h - 1) \odot \big(c + W^\top v\big)\big)_j$$

- And similarly for v

$$P(v \mid h) = \prod_{i=1}^{n_v} \sigma((2v - 1) \odot (b + Wh))_i$$

Training a RBM

- The probability distribution in a Markov Random Field:

$$p(\mathrm{x};\theta) = \frac{\tilde{p}(\mathrm{x};\theta)}{Z(\theta)}$$

- If we maximize the logarithm of the likelihood:

$$\nabla_\theta \log p(\mathrm{x};\theta) = \nabla_\theta \log \tilde{p}(\mathrm{x};\theta) - \nabla_\theta \log Z(\theta)$$

- Positive phase:

- Maximize the unnormalized probability the model assigns to the observations (training data)

- Negative phase:

- Minimize the probability of other states ("hallucinations")
- Remember: Z is the sum of the unnormalized probabilities of all states

Training a RBM

Contrastive divergence algorithm

- From sample v, compute probabilities of h and obtain a sample h
- Positive gradient of W: outer product of v and the sampled h

- Intuitively, how to change W to maximize probability of data

- Sample model state from the probability distribution $p(v, h)$

- Use Gibbs sampling generating samples v' and h' until convergence.
- Few iterations; we start from observed v and model should approximate it

- Negative gradient of W: outer product of sampled v' and h'.

- Intuitively, how to change W to minimize "halluciunations"

- Update W: positive minus negative gradient (times learning rate)
- Updates biases: $v - v'$ and $h - h'$.

- Intuitively, adjust so states are closer to what they should be

Implementing a RBM

Implementing a RBM

Based on a demo by Gulli et al. "Deep Learning with TensorFlow 2 and Keras"

- Train RBM on MNIST (unsupervised learning)

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

(train_data, _), (test_data, _) = tf.keras.datasets.mnist.load_data()
train_data = train_data/np.float32(255)
train_data = np.reshape(train_data, (train_data.shape[0], 784))
test_data = test_data/np.float32(255)
test_data = np.reshape(test_data, (test_data.shape[0], 784))

learning_rate = 1.0
visible_size = train_data.shape[1]
hidden_size = 200
batchsize = 100
```

- 784 visible variables (28x28)
- 200 hidden variables

Implementing a RBM

- Computing probabilities and sampling

```
def prob_h_given_v(visible, w, hb):
    return tf.nn.sigmoid(tf.matmul(visible, w) + hb)

def prob_v_given_h(hidden, w, vb):
    return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)

def sample_prob(probs):
    return tf.nn.relu(tf.sign(probs - tf.random.uniform(tf.shape(probs))))
```

- `relu` is convenient for turning -1 to 0 keeping the values 1

Implementing a RBM

- Gibbs sampling

```
def gibbs_sampling(batch,k=1):
    hidden_prob = prob_h_given_v(batch, weights,h_bias)
    hidden_states_0 = sample_prob(hidden_prob)
    original_hs = hidden_states_0
    for _ in range(k):
        visible_prob = prob_v_given_h(hidden_states_0, weights, v_bias)
        visible_states = sample_prob(visible_prob)
        hidden_prob = prob_h_given_v(visible_states, weights,h_bias)
        hidden_states = sample_prob(hidden_prob)
        hidden_states_0 = hidden_states
    return original_hs,hidden_states,visible_states
```

Implementing a RBM

- Training loop

```
def train(X, epochs=10):
    global weights, h_bias, v_bias
    loss = []
    for epoch in range(epochs):
        for start, end in zip(range(0, len(X), batchsize),
                              range(batchsize,len(X), batchsize)):
            batch = X[start:end]
            ohs,hs,vs = gibbs_sampling(batch,k=2)
            positive_grad = tf.matmul(tf.transpose(batch), ohs)
            negative_grad = tf.matmul(tf.transpose(vs), hs)
            b_size = tf.dtypes.cast(tf.shape(batch)[0],tf.float32)
            dw = (positive_grad - negative_grad) / b_size
            weights += learning_rate * dw
            v_bias += learning_rate * tf.reduce_mean(batch - vs, 0)
            h_bias += learning_rate * tf.reduce_mean(ohs - hs, 0)
        err = tf.reduce_mean(tf.square(batch - vs))
        print (f'Epoch: {epoch}, reconstruction error: {err}')
        loss.append(err)
    return loss
```

Implementing a RBM

- Train:

```
weights = tf.zeros([visible_size, hidden_size], np.float32)
h_bias = tf.zeros([hidden_size], np.float32)
v_bias = tf.zeros([visible_size], np.float32)
err = train(train_data,50)
```

- After training, reconstruct image, smoother (take probability as an expected value)

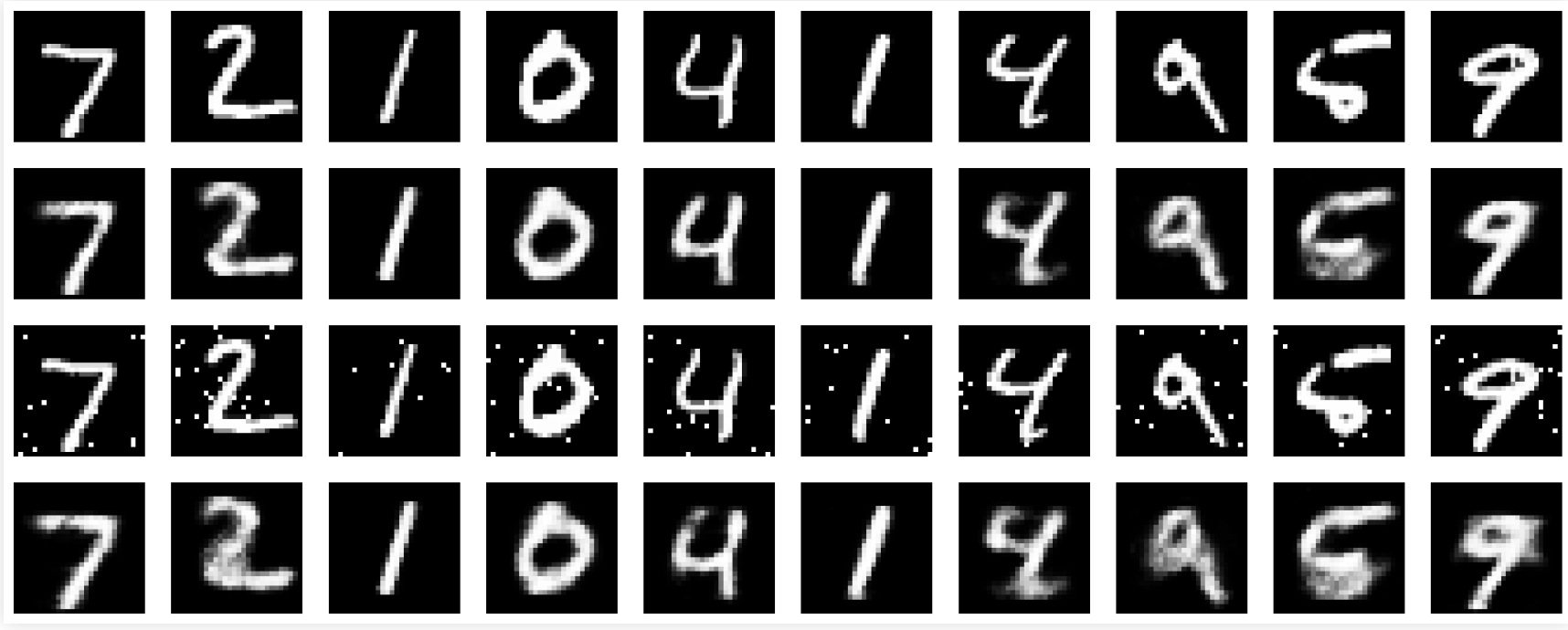
```
def rbm_reconstruct(X):
    h = tf.nn.sigmoid(tf.matmul(X, weights) + h_bias)
    reconstruct = tf.nn.sigmoid(tf.matmul(h, tf.transpose(weights)) + v_bias)
    return reconstruct
```

Implementing a RBM

- Reconstructing and denoising the test set

```
sample = test_data[:10,:]
noisy = np.copy(sample)
noisy[np.random.rand(*sample.shape)>0.98] = 1
out = rbm_reconstruct(noisy)
orig_out = rbm_reconstruct(sample)
row, col = 4, 10
fig, axs = plt.subplots(row, col, sharex=True, sharey=True,figsize=(20,8))
for c in range(col):
    axs[0,c].imshow(tf.reshape(sample[c],[28, 28]), cmap='Greys_r')
    axs[1,c].imshow(tf.reshape(orig_out[c],[28, 28]), cmap='Greys_r')
    axs[2,c].imshow(tf.reshape(noisy[c],[28, 28]), cmap='Greys_r')
    axs[3,c].imshow(tf.reshape(out[c],[28, 28]), cmap='Greys_r')
for ax in axs.flatten():
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

Implementing a RBM



Implementing a RBM

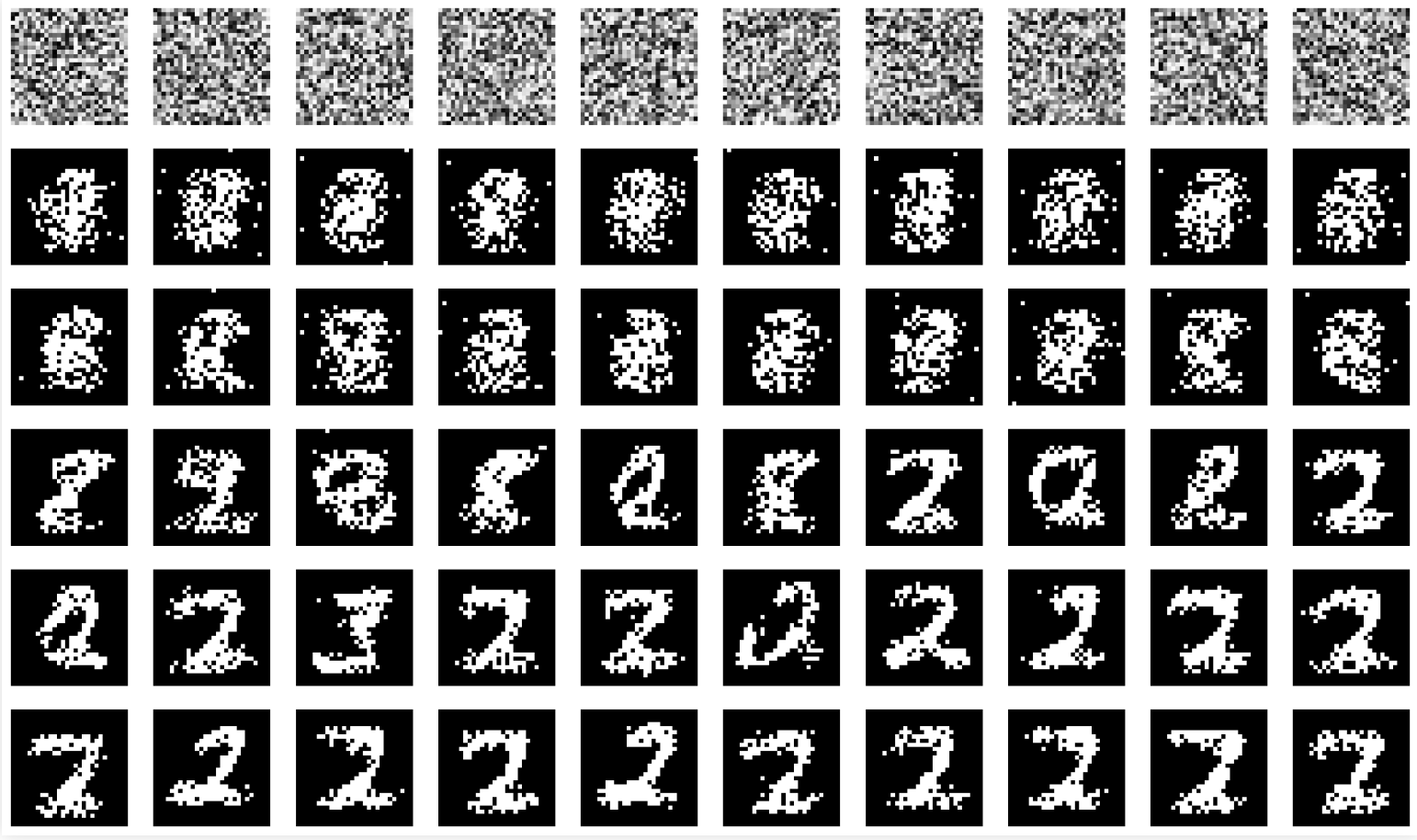
- Gibbs sampling to generate data

- Note: there are better methods, but this one is simple to illustrate
- Breuleux et al., 2011, Quickly Generating Representative Samples from an RBM-Derived Process

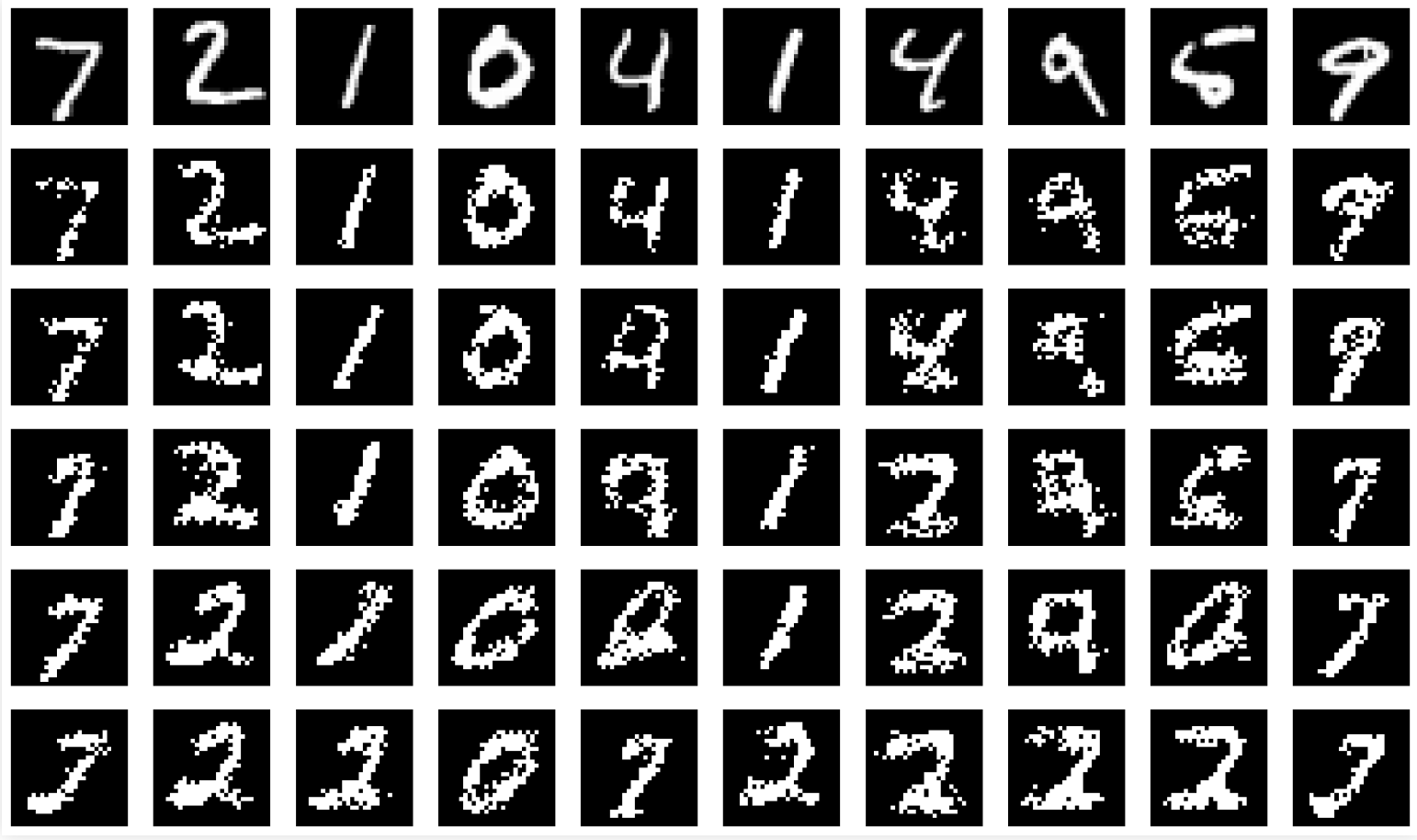
```
sample = np.random.rand(10,28*28).astype(np.float32)
k = 1
row, col = 6, 10
outs = []
for _ in range(1,row):
    _,_,o = gibbs_sampling(sample,k)
    k *= 10
    outs.append(o)

fig, axs = plt.subplots(row, col, sharex=True, sharey=True,figsize=(20,2*row))
for c in range(col):
    axs[0,c].imshow(tf.reshape(sample[c],[28, 28]), cmap='Greys_r')
    for ix in range(1,row):
        axs[ix,c].imshow(tf.reshape(outs[ix-1][c],[28, 28]), cmap='Greys_r')
for ax in axs.flatten():
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

Implementing a RBM



Implementing a RBM



- Same, but starting from test images

Summary

Probabilistic Models

Summary

- Graphical Models
- Restricted Boltzmann Machine
- Sampling
- Contrastive Divergence

Further reading:

- Goodfellow et.al, Deep learning, Chapter 16 and Sections 18.1, 18.2, 20.1-4
- Optional: Breleux, Quickly Generating Representative Samples from an RBM-Derived Process