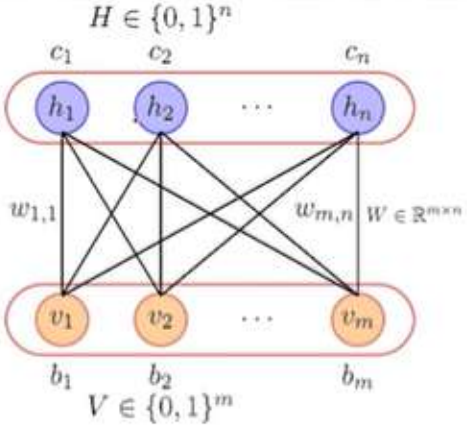
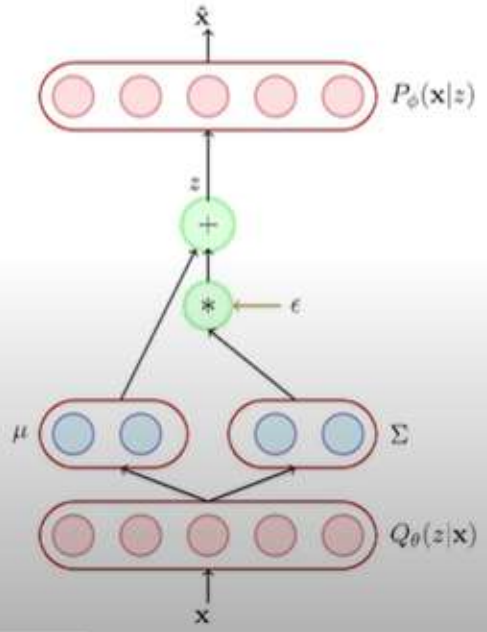
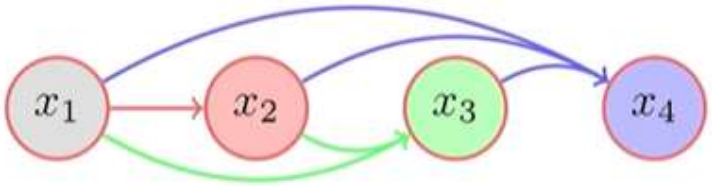
Module 21.1 Neural Autoregressive Density Estimator (NADE)

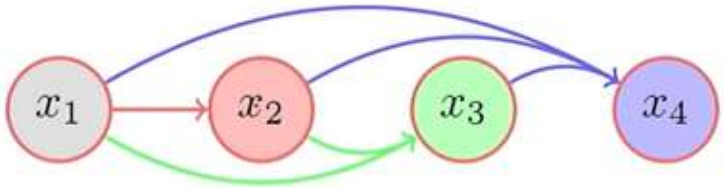




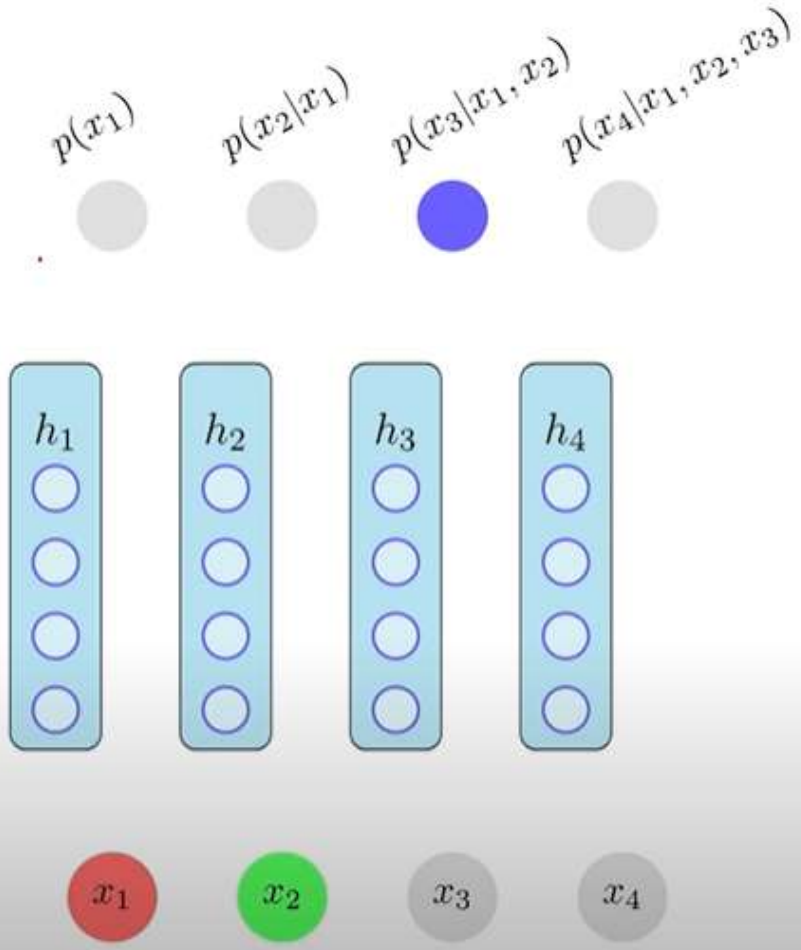
- So far we have seen a few latent variable generation models such as RBMs and VAEs
- Latent variable models make certain independence assumptions which reduces the number of factors and in turn the number of parameters in the model
- For example, in RBMs we assumed that the visible variables were independent given the hidden variables which allowed us to do Block Gibbs Sampling
- Similarly in VAEs we assumed $P(\mathbf{x}|z) = \mathcal{N}(0, I)$ which effectively means that given the latent variables, the $\mathbf{x}$'s are independent of each other (Since $\Sigma = \mathrm{I}$)



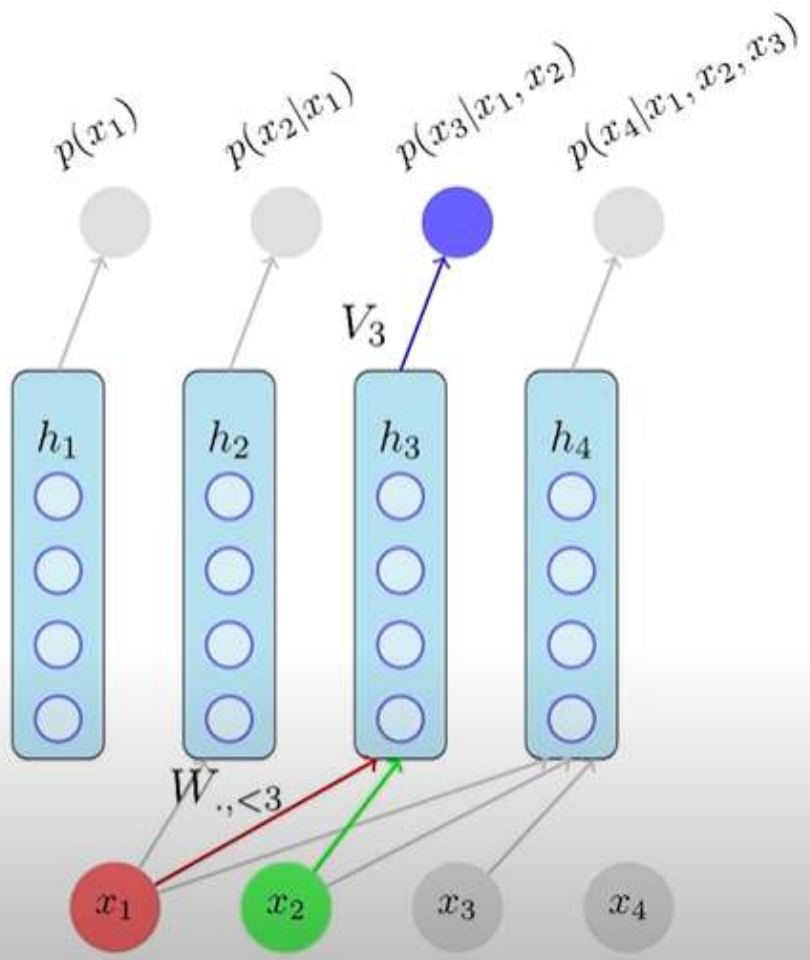
- We will now look at Autoregressive (AR) Models which do not contain any latent variables
- The aim of course is to learn a joint distribution over $\mathbf{x}$
- As usual, for ease of illustration we will assume $\mathbf{x} \in \{0,1\}^n$
- AR models do not make any independence assumption but use the default factorization of $p(\mathbf{x})$ given by the chain rule $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$
- The above factorization contains n factors and some of these factors contain many parameters ($O(2^n)$ in total)



- Obviously, it is infeasible to learn such an exponential number of parameters
- AR models work around this by using a neural network to parameterize these factors and then learn the parameters of this neural network
- What does this mean? Let us see!



- At the output layer we want to predict n conditional probability distributions (each corresponding to one of the factors in our joint distribution)
- At the input layer we are given the n input variables
- Now the catch is that the n^{th} output should only be connected to the previous n-1 inputs
- In particular, when we are computing $p(x_3|x_2, x_1)$ the only inputs that we should consider are x_1, x_2 because these are the only variables *given* to us while computing the conditional



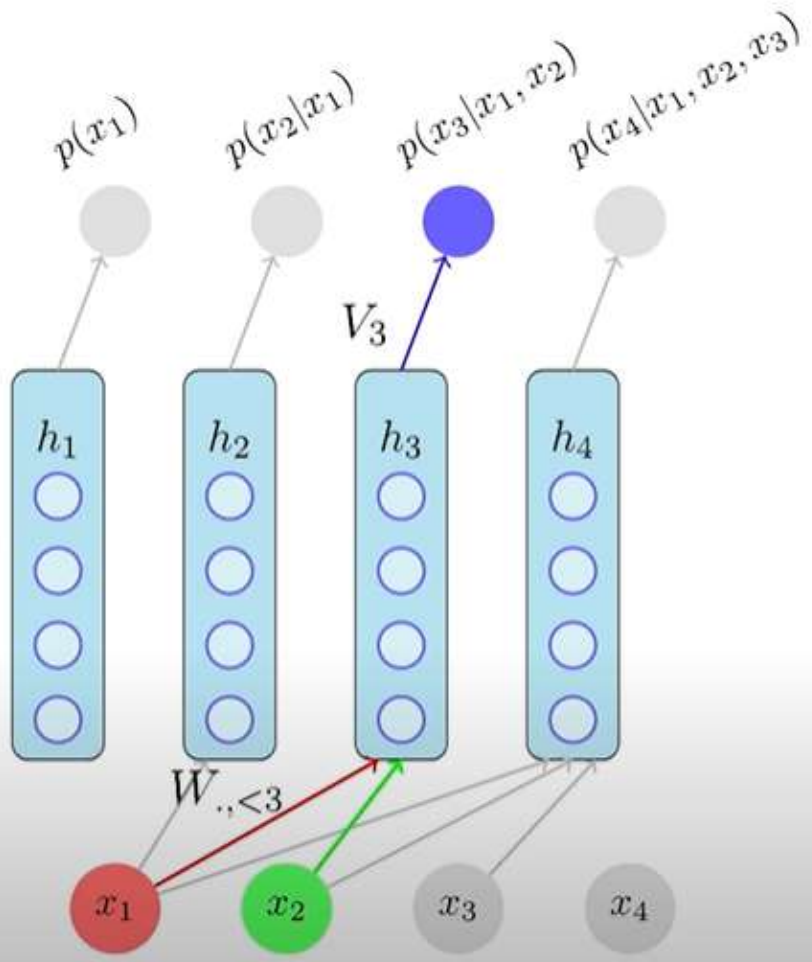
- The Neural Autoregressive Density Estimator (NADE) proposes a simple solution for this
- First for every output unit, we compute a hidden representation using only the relevant input units
- For example, for the k^{th} output unit, the hidden representation will be computed using:

$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$

where $h_k \in R^d, W \in R^{d \times n}, W_{.,<k}$ are the first k columns of W

- We now compute the output $p(x_k|\mathbf{x}_1^{k-1})$ as:

$$y_k = p(x_k|\mathbf{x}_1^{k-1}) = \sigma(V_k h_k + c_k)$$

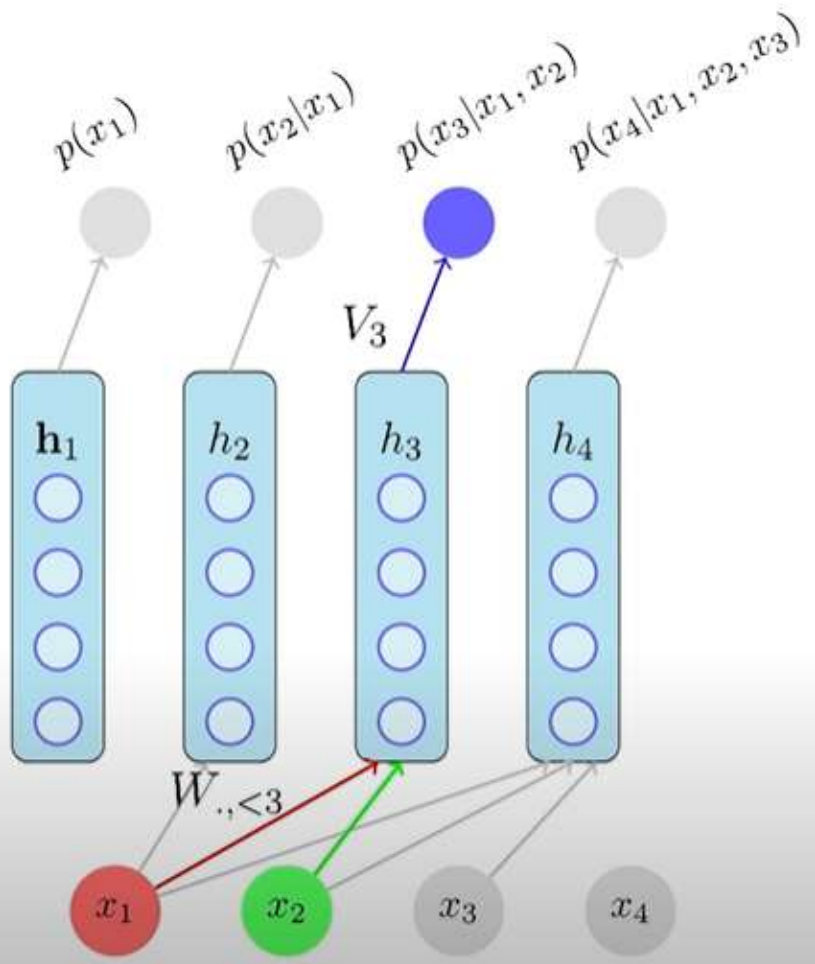


- Let us look at the equations carefully

$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$

$$y_k = p(x_k|\mathbf{x}_1^{k-1}) = \sigma(V_k h_k + c_k)$$

- How many parameters does this model have ?
- Note that $W \in R^{d \times n}$ and $b \in R^{d \times 1}$ are shared parameters and the same W, b are used for computing h_k for all the n factors (of course only the relevant columns of W are used for each k)
- In addition, we have $V_k \in R^{d \times 1}$ and $c_k \in R$ for each of the n factors resulting in a total of $n * (d + 1)$ parameters

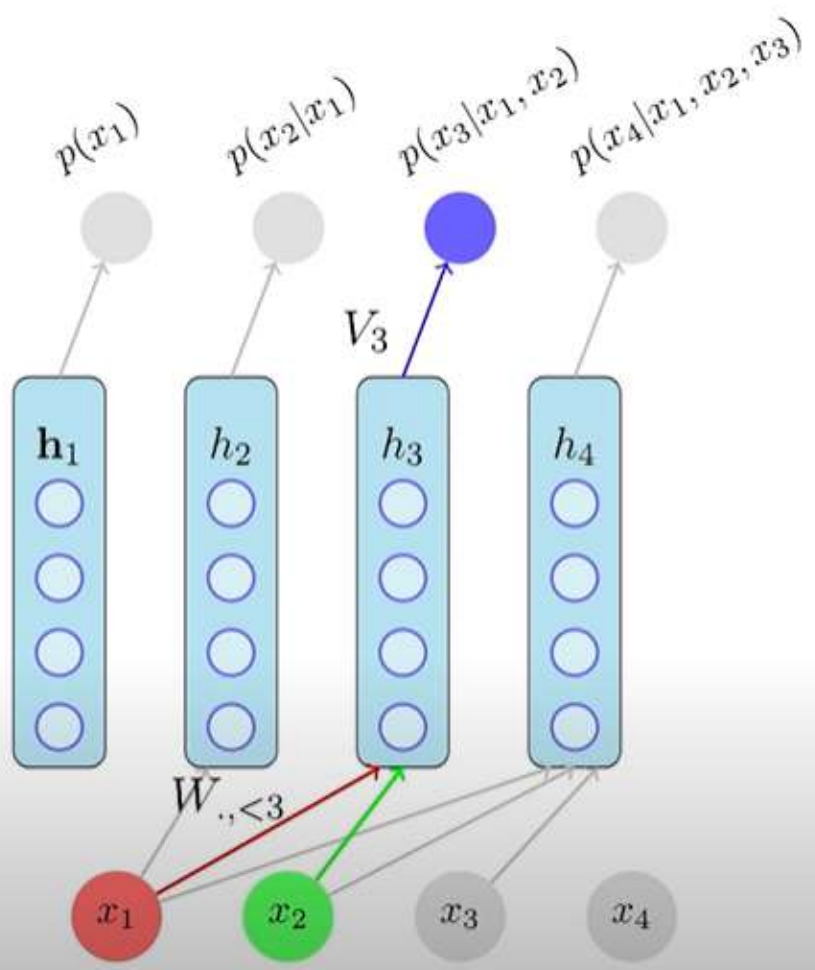


- There is also an additional parameter $h_1 \in R^d$ (similar to the initial state in LSTMs, RNNs)
- The total number of parameters in the model is thus $(n+1)d + 2n(d) + d = (3n+2)d$ which is linear in n
- In other words, the model does not have an exponential number of parameters which is typically the case for the default factorization

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$$

$$\left(W_{.,<k}\, x_{<k} + b\right) \in \mathbb{R}^d$$

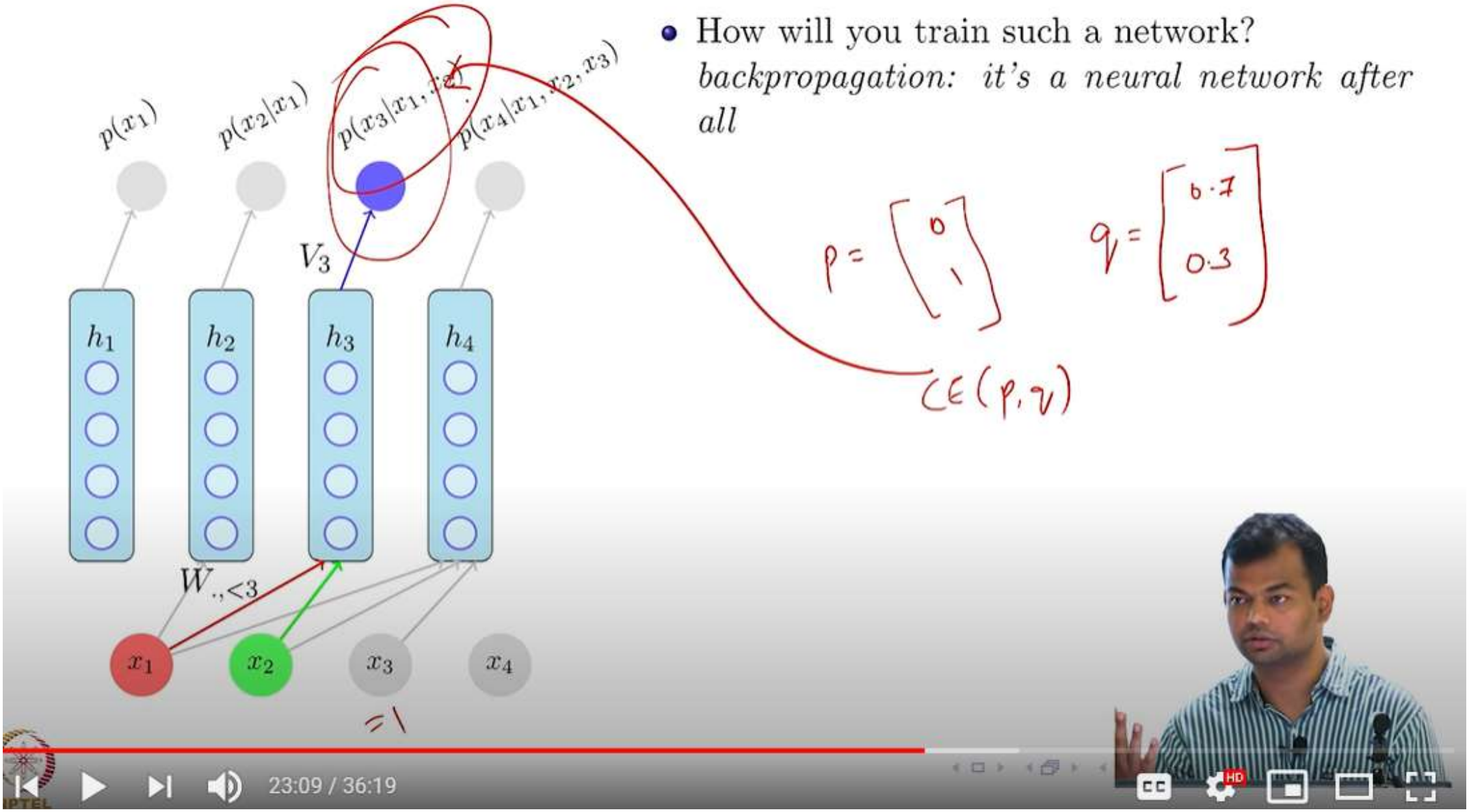
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \\ 0 \end{bmatrix}$$

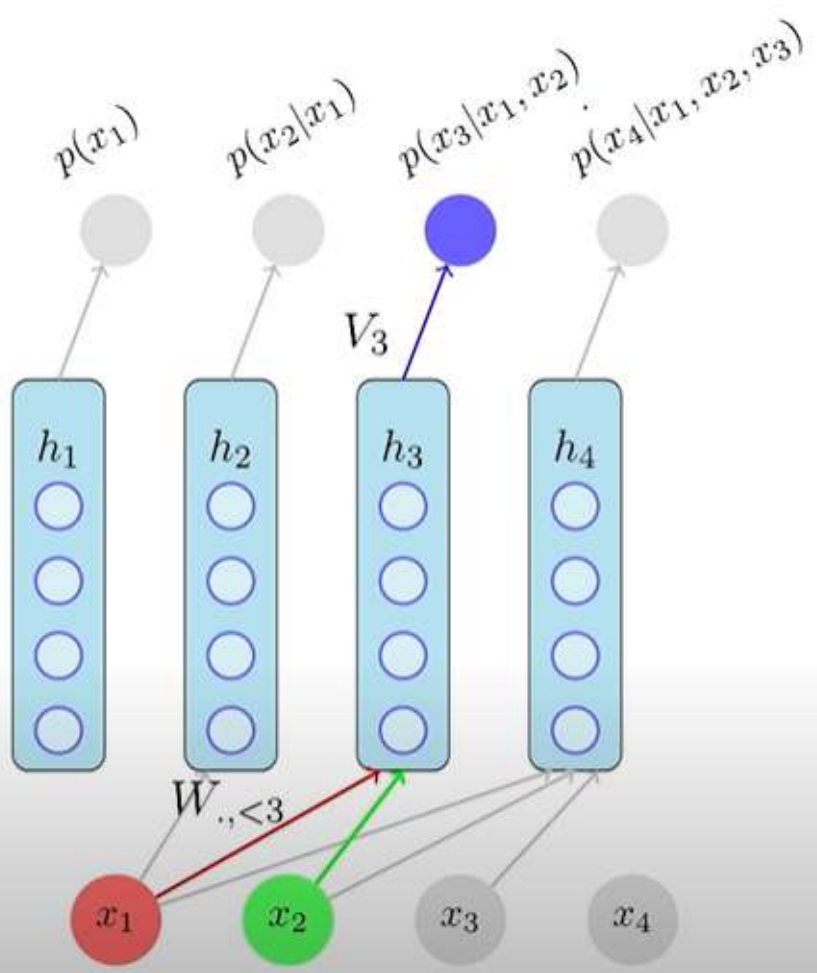


- There is also an additional parameter $h_1 \in R^d$ (similar to the initial state in LSTMs, RNNs)
- The total number of parameters in the model is thus $(n+1)d + 2n(d) + d = (3n+2)d$ which is linear in n
- In other words, the model does not have an exponential number of parameters which is typically the case for the default factorization

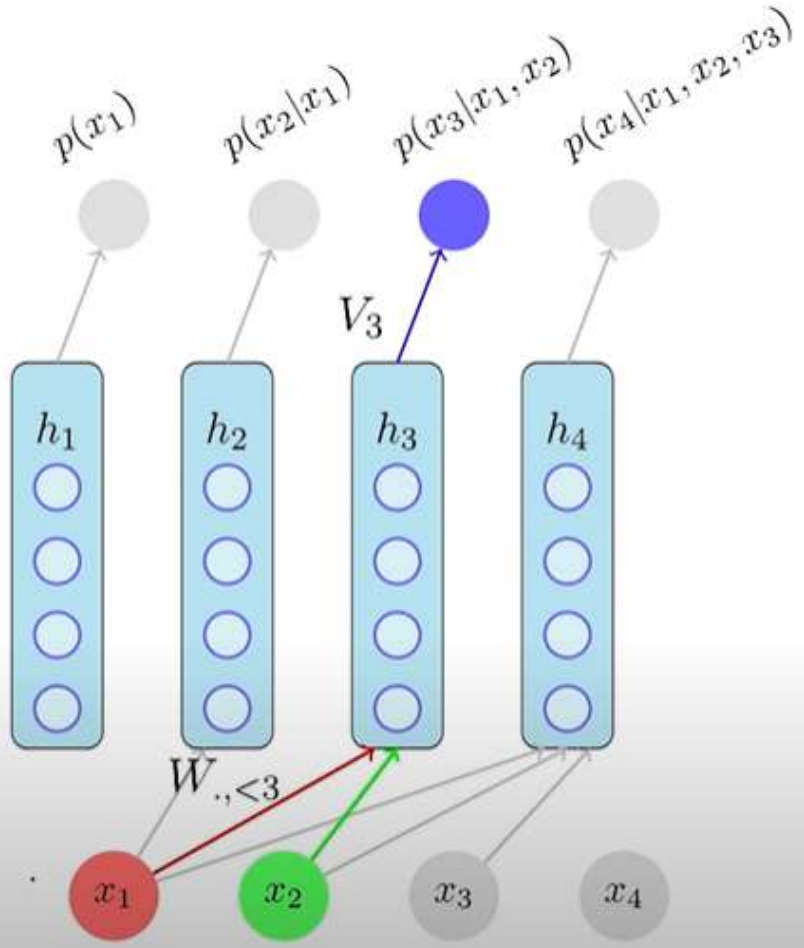
$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$$

- Why? Because we are sharing the parameters across the factors
- The same W, b contribute to all the factors

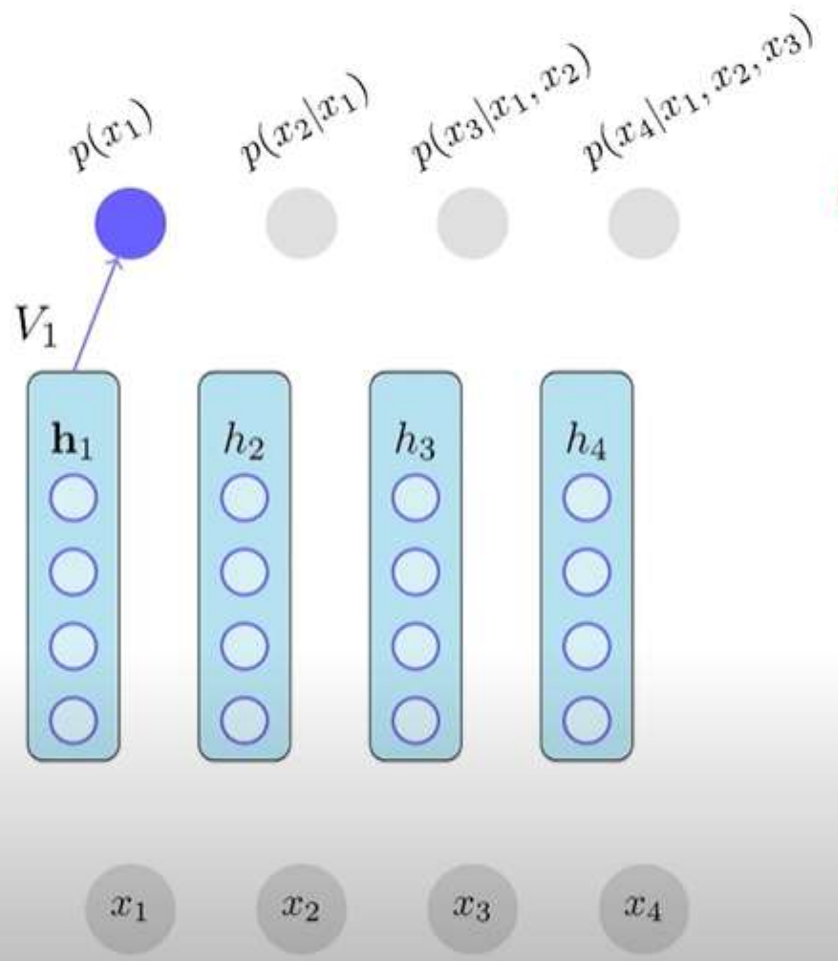

How will you train such a network?
backpropagation: it's a neural network after all
$p(x_1)$
$p(x_2|x_1)$
$p(x_3|x_1, x_2)$
$p(x_4|x_1, x_2, x_3)$
V_3
h_1
h_2
h_3
h_4
$W_{.,<3}$
x_1
x_2
x_3
x_4
$p = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
$q = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$
$CE(p, q)$
23:09 / 36:19



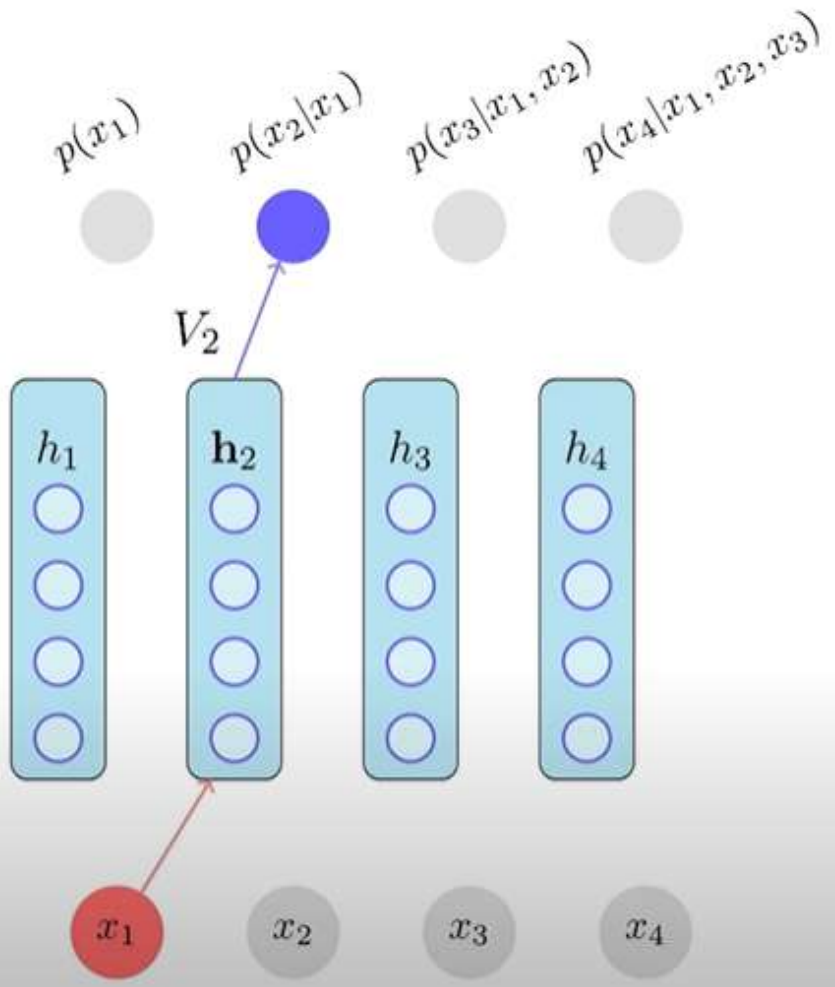
- How will you train such a network?
  backpropagation: it's a neural network after all
- What is the loss function that you will choose?
- For every output node we know the true probability distribution
- For example, for a given training instance, if $\mathbf{x}_3 = 1$ then the true probability distribution is given by $p(\mathbf{x}_3 = 1|x_2, x_1) = 1, p(\mathbf{x}_3 = 0|x_2, x_1) = 0$ or $p = [0, 1]$
- If the predicted distribution is $q = [0.7, 0.3]$ then we can just take the cross entropy between p and q as the loss function
- The total loss will be the sum of this cross entropy loss for all the n output nodes



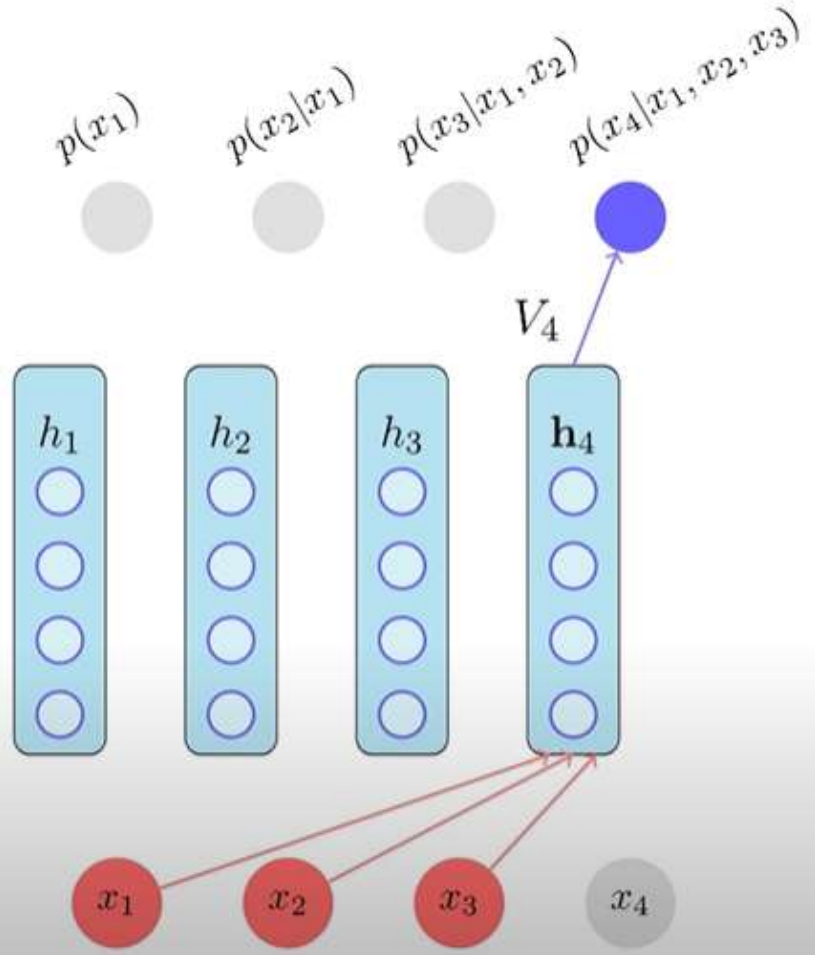
- Now let's ask a couple of questions about the model (assume training is done)
- Can the model be used for abstraction? i.e. if we give it a test instance $\mathbf{x}$, can the model give us a hidden abstract representation for $\mathbf{x}$
- Well, you will get a sequence of hidden representations $h_1, h_2, ..., h_n$ but these are not really the kind of abstract representations that we are interested in
- For example, h_n only captures the information required to reconstruct x_n given x_1 to x_{n-1} (compare this with an autoencoder wherein the hidden representation can reconstruct all of $x_1, x_2, ..., x_n$)
- These are not latent variable models and are by design not meant for abstraction



- Can we use the model for generation? How ?
- Well, we first compute $p(\mathbf{x}_1 = 1)$ as $y_1 = \sigma(V_1 h_1 + c_1)$ = 0.4



- We will now use the sampled value of x_1 and compute h_2 as
  $h_2 = \sigma(W_{.,<2}\mathbf{x}_{<2} + b)$



- We will now use the sampled value of x_1 and compute h_2 as
  $h_2 = \sigma(W_{.,<2}\mathbf{x}_{<2} + b)$
- Using h_2 we will compute $P(\mathbf{x}_2 = 1|\mathbf{x}_1 = x_1)$ as $y_2 = \sigma(V_2 h_2 + c_2)$
- We will then sample a value for x_2 from the distribution $Bernoulli(y_2)$
- We will then continue this process till x_n generating the value of one random variable at a time
- If $\mathbf{x}$ is an image then this is equivalent to generating the image one pixel at a time (very slow)

- Of course, the model requires a lot of computations because for generating each pixel we need to compute

$$h_k = \sigma(W_{.,<k}\mathbf{x}_{<k} + b)$$
$$y_k = p(x_k|\mathbf{x}_1^{k-1}) = \sigma(V_k h_k + c_k)$$

- However notice that

$$W_{.,<k+1}\mathbf{x}_{<k+1} + b = W_{.,<k}\mathbf{x}_{<k} + b + W_{k+1}$$

- Thus we can reuse some of the computations done for pixel k while predicting the pixel $k+1$ (this can be done even at training time)

Things to remember about NADE

- Uses the explicit representation of the joint distribution $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|\mathbf{x}_{<k})$
- Each node in the output layer corresponds to one factor in this explicit representation
- Reduces the number of parameters by sharing weights in the neural network
- Not designed for abstraction
- Generation is slow because the model generates one pixel (or one random variable) at a time
- Possible to speed up the computation by reusing some previous computations