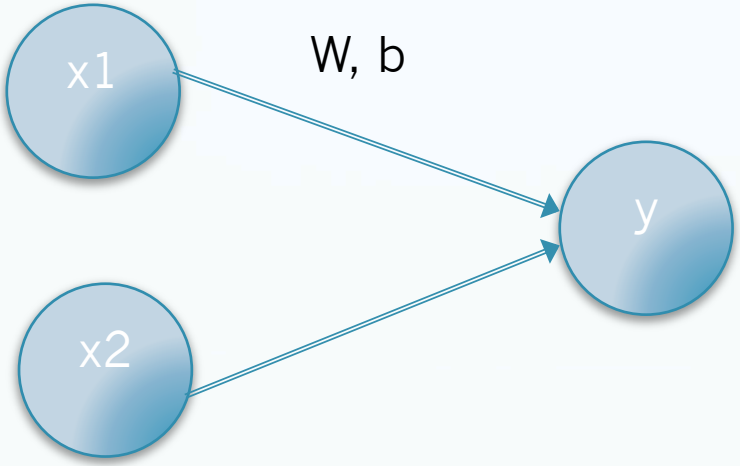
Learning Deep Architectures for AI

- Yoshua Bengio

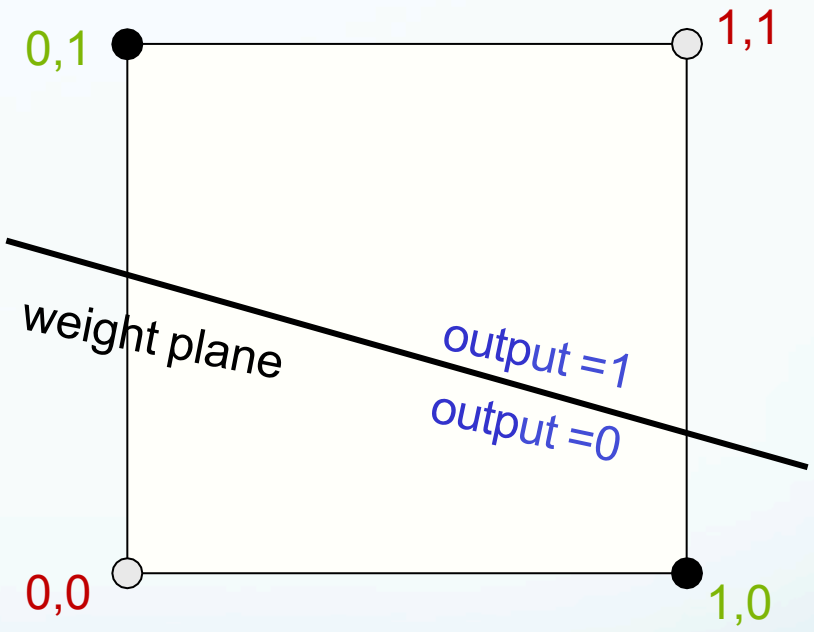
Part II

- Vijay Chakilam

Limitations of Perceptron

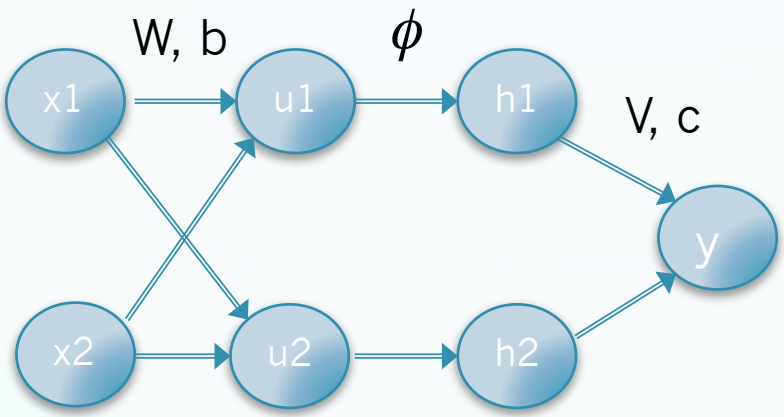


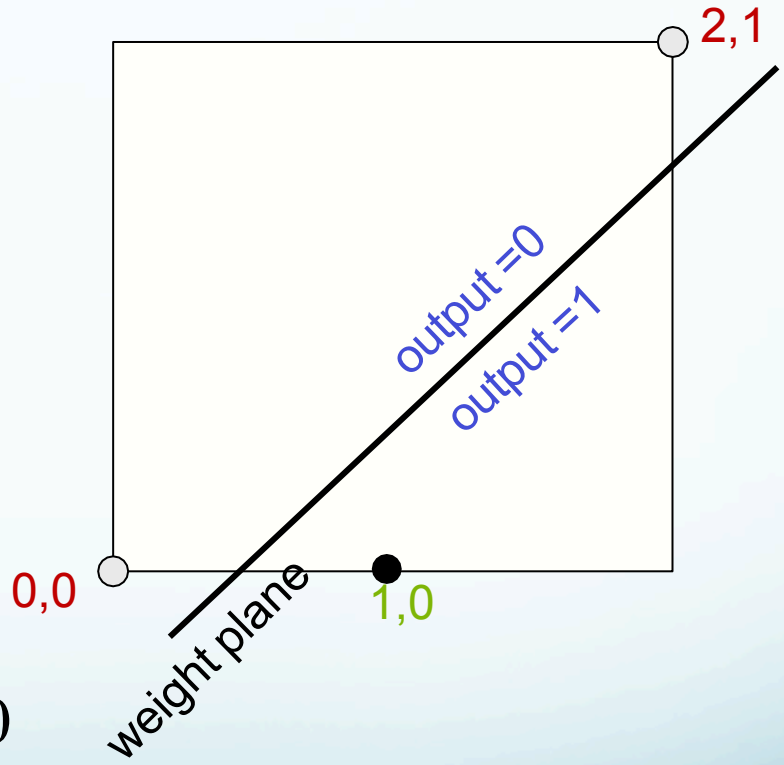
- There is no value for W and b such that the model results in right target for every example



A graphical view of the XOR problem

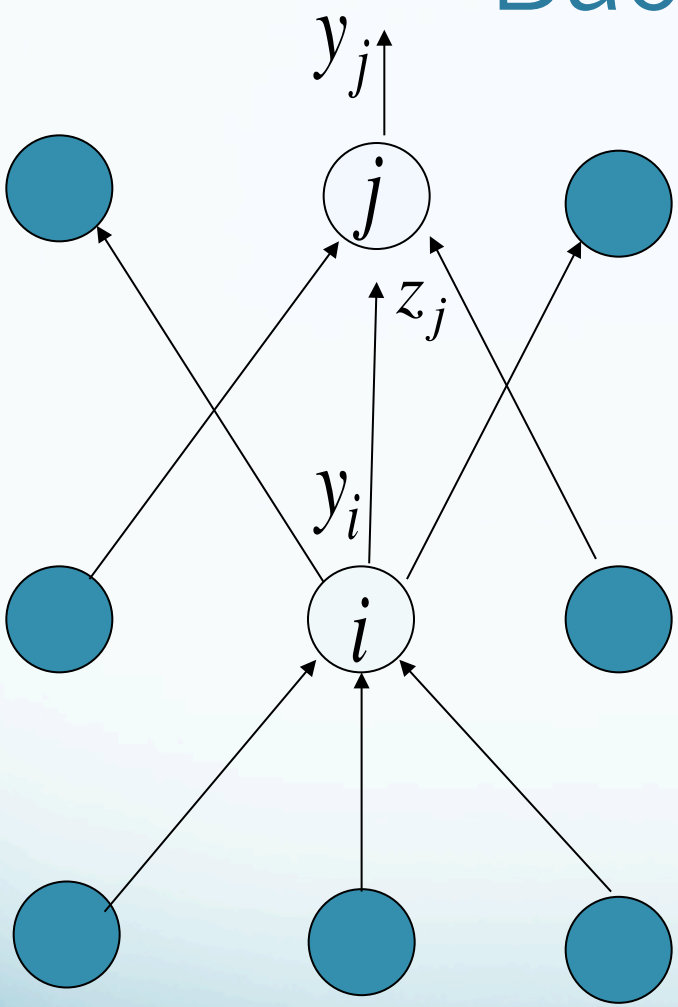
Learning representations





$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \; b = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \; V = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \text{ and } c = 0$$

Backpropagation



$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$

$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j} \frac{\partial E}{\partial y_j} = y_j (1 - y_j) \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

Deep vs Shallow

- Pascanu et al. (2013) compared deep rectifier networks with their shallow counterparts.
  - For a deep model with n_0 inputs and k hidden layers of width n, the maximal number of response regions per parameter behaves as

$$\Omega\left(\left\lfloor \frac{n}{n_0} \right\rfloor^{k-1} \frac{n^{n_0-2}}{k}\right)$$

  - For a shallow model with n_0 inputs and nk hidden units, the maximal number of response regions per parameter behaves as

$$O\left(k^{n_0-1} n^{n_0-1}\right)$$

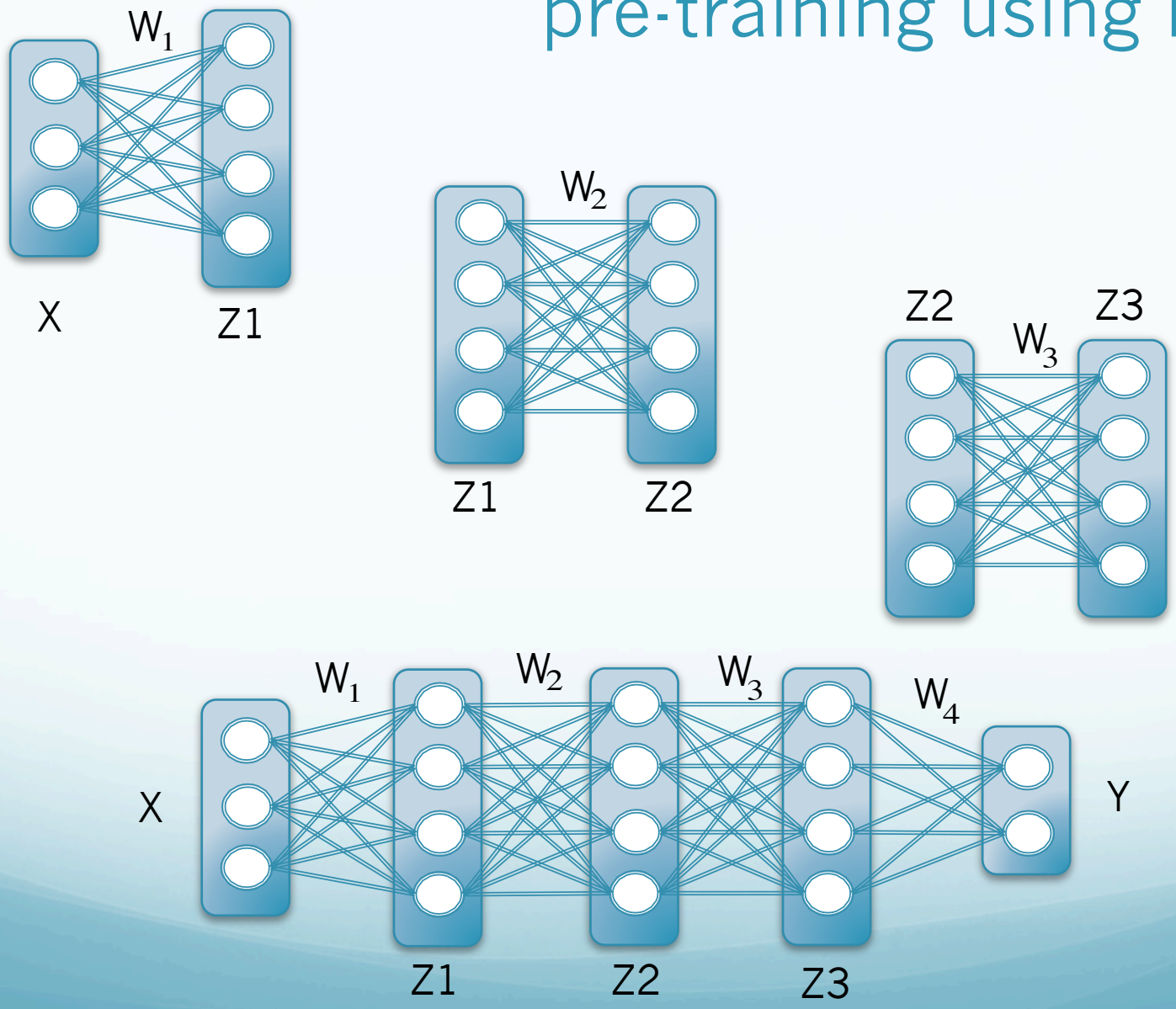
Deep vs Shallow

- Pascanu et al. (2013) showed that the deep model can generate exponentially more regions per parameter in terms of the number of hidden layers, and at least order polynomially more regions per parameter in terms of layer width n.
- Montufar et al. (2014) came up with significantly improved lower bound on the maximal linear regions.

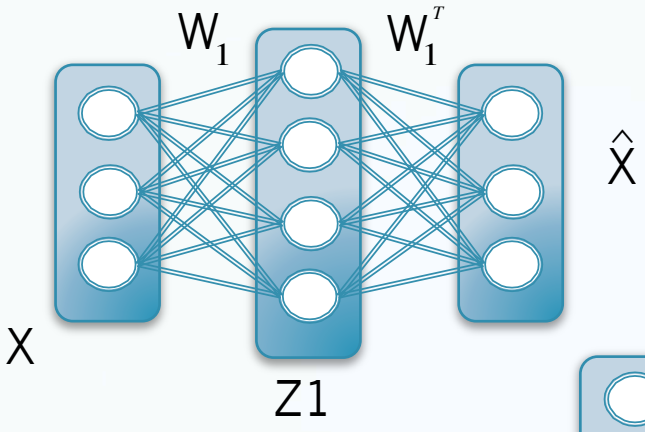
Unsupervised Pre-training

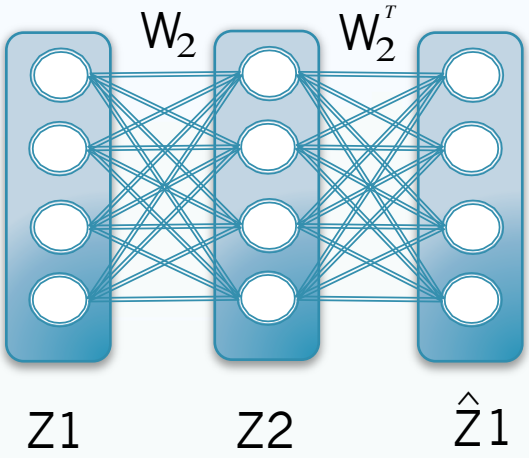
- Scarcity of labeled examples and availability of many unlabeled examples.
- Unknown future tasks.
- Once a good high-level representation is learned, other learning tasks could be much easier.
  - For example, kernel machines can be very powerful if using an appropriate kernel, i.e. an appropriate feature space.

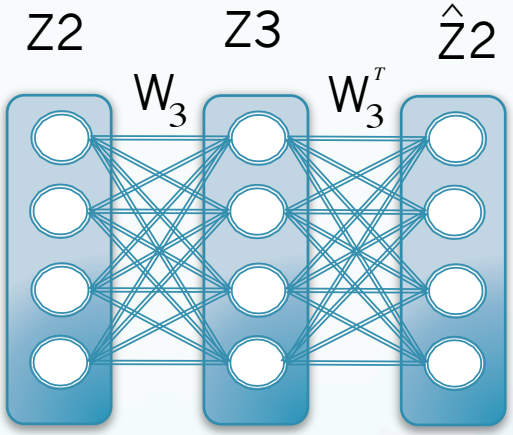
DNN with unsupervised pre-training using RBMs

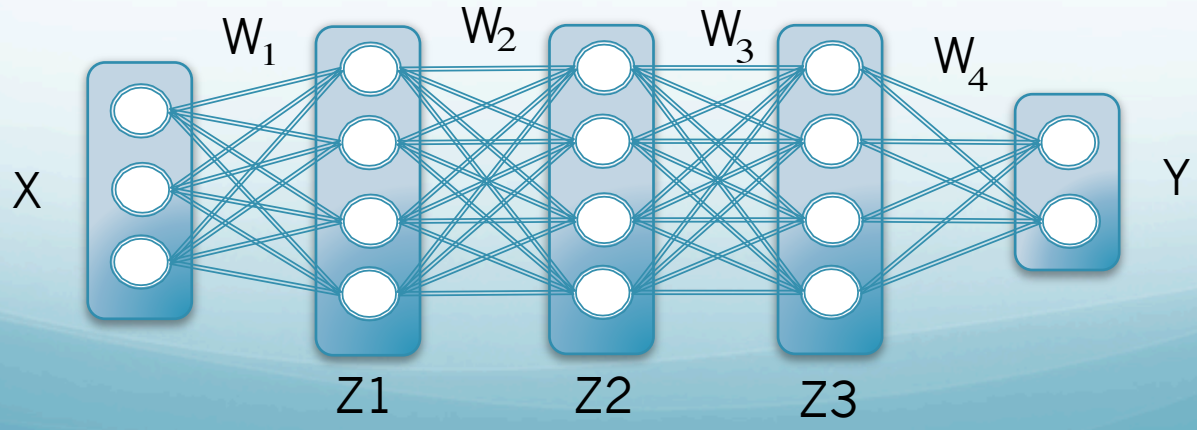


DNN with unsupervised pre-training using Autoencoders









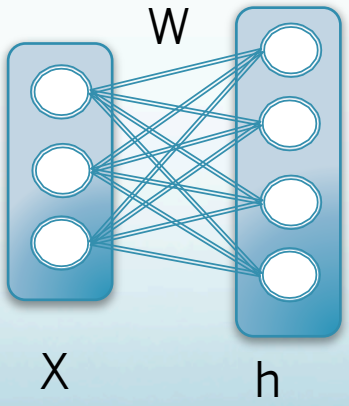
Restricted Boltzmann Machines

Restricted Boltzmann Machine

- Energy function: $E(x,h) = -h^T W x - c^T x - b^T h$

$$= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j$$

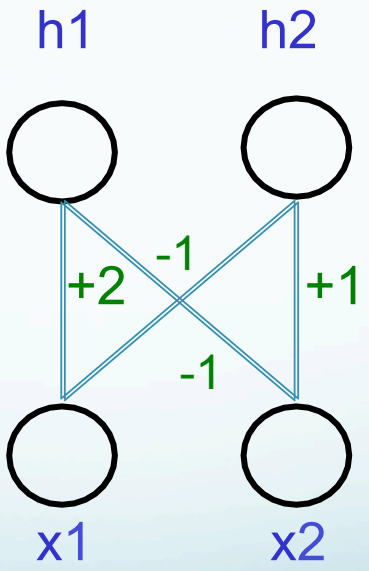
- Distribution: $p(x,h) = \exp(-E(x,h)) / Z$



An example of how weights define a distribution

x	h	$-E$	e^{-E}	$P(\mathbf{x}, \mathbf{h})$	$P(\mathbf{x})$
1 1	1 1	2	7.39	.186	
1 1	1 0	2	7.39	.186	
1 1	0 1	1	2.72	.069	0.466
1 1	0 0	0	1	.025	
1 0	1 1	1	2.72	.069	
1 0	1 0	2	7.39	.186	
1 0	0 1	0	1	.025	0.305
1 0	0 0	0	1	.025	
0 1	1 1	0	1	.025	
0 1	1 0	0	1	.025	
0 1	0 1	1	2.72	.069	0.144
0 1	0 0	0	1	.025	
0 0	1 1	-1	0.37	.009	
0 0	1 0	0	1	.025	0.084
0 0	0 1	0	1	.025	
0 0	0 0	0	1	.025	

39.70



Log likelihood

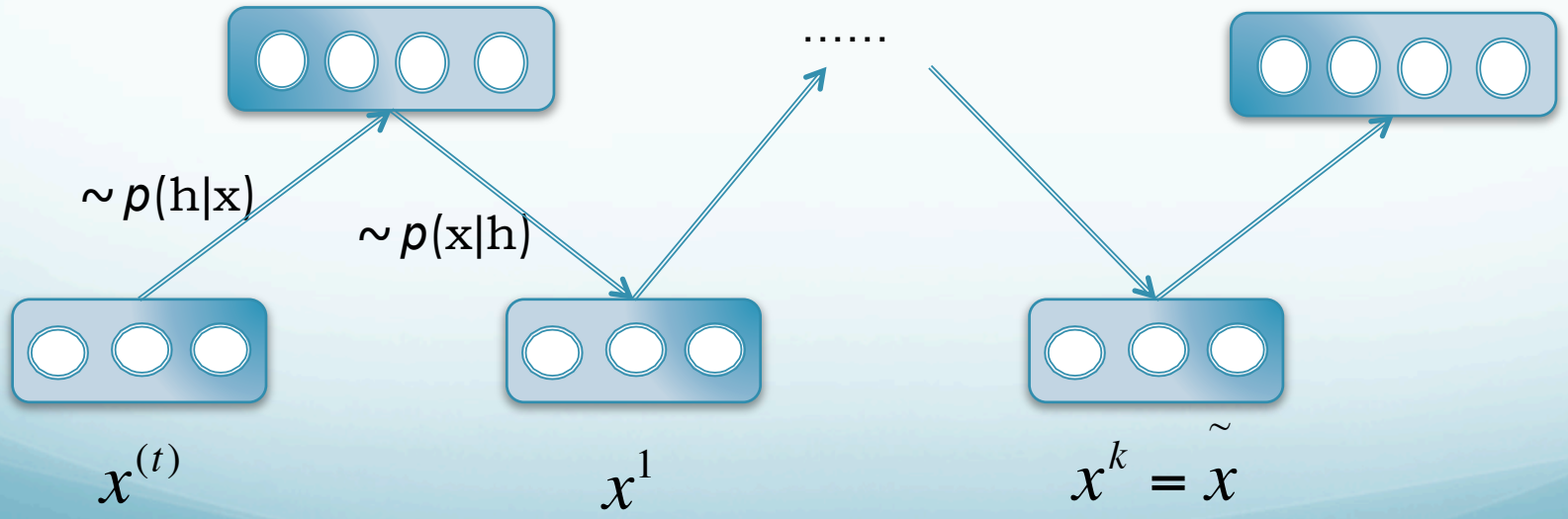
- To train an RBM, we'd like to minimize the average negative log-likelihood

$$\frac{1}{T}\sum_t l(f(x^{(t)})) = \frac{1}{T}\sum_t -\log p(x^{(t)})$$

$$\frac{\partial -\log p(x^{(t)})}{\partial \theta} = E_h\left[\frac{\partial E(x^{(t)},h)}{\partial \theta} \mid x^{(t)}\right] - E_{x,h}\left[\frac{\partial E(x,h)}{\partial \theta}\right]$$

Contrastive Divergence

- Idea:
  - Replace the expectation by a point estimate at x˜
  - Obtain the point x˜ by Gibbs sampling
  - Start sampling chain at x(t)



Conditional probabilities

$$p(h \mid x) = \prod_j p(h_j \mid x)$$

$$p(h_j = 1 \mid x) = \frac{1}{1 + \exp(-(b_j + W_{j.} x))}$$

$$= sigm(b_j + W_{j.} x)$$

$$p(x \mid h) = \prod_k p(x_k \mid h)$$

$$p(x_k = 1 \mid h) = \frac{1}{1 + \exp(-(c_k + h^T W_{.k}))}$$

$$= sigm(c_k + h^T W_{.k})$$

Conditional probabilities

$$
\begin{aligned}
p(\mathbf{h}|\mathbf{x}) &= p(\mathbf{x},\mathbf{h}) / \sum_{\mathbf{h}'} p(\mathbf{x},\mathbf{h}') \\
&= \frac{\exp(\mathbf{h}^\top \mathbf{W}\mathbf{x} + \cancel{\mathbf{c}^\top \mathbf{x}} + \mathbf{b}^\top \mathbf{h}) / \cancel{Z}}{\sum_{\mathbf{h}' \in \{0,1\}^H} \exp(\mathbf{h}'^\top \mathbf{W}\mathbf{x} + \cancel{\mathbf{c}^\top \mathbf{x}} + \mathbf{b}^\top \mathbf{h}') / \cancel{Z}} \\
&= \frac{\exp(\sum_j h_j \mathbf{W}_{j.}\mathbf{x} + b_j h_j)}{\sum_{h'_1 \in \{0,1\}} \cdots \sum_{h'_H \in \{0,1\}} \exp(\sum_j h'_j \mathbf{W}_{j.}\mathbf{x} + b_j h'_j)} \\
&= \frac{\prod_j \exp(h_j \mathbf{W}_{j.}\mathbf{x} + b_j h_j)}{\sum_{h'_1 \in \{0,1\}} \cdots \sum_{h'_H \in \{0,1\}} \prod_j \exp(h'_j \mathbf{W}_{j.}\mathbf{x} + b_j h'_j)} \\
&= \frac{\prod_j \exp(h_j \mathbf{W}_{j.}\mathbf{x} + b_j h_j)}{\left(\sum_{h'_1 \in \{0,1\}} \exp(h'_1 \mathbf{W}_{1.}\mathbf{x} + b_1 h'_1)\right) \cdots \left(\sum_{h'_H \in \{0,1\}} \exp(h'_H \mathbf{W}_{H.}\mathbf{x} + b_H h'_H)\right)} \\
&= \frac{\prod_j \exp(h_j \mathbf{W}_{j.}\mathbf{x} + b_j h_j)}{\prod_j \left(\sum_{h'_j \in \{0,1\}} \exp(h'_j \mathbf{W}_{j.}\mathbf{x} + b_j h'_j)\right)} \\
&= \frac{\prod_j \exp(h_j \mathbf{W}_{j.}\mathbf{x} + b_j h_j)}{\prod_j \left(1 + \exp(b_j + \mathbf{W}_{j.}\mathbf{x})\right)} \\
&= \prod_j \frac{\exp(h_j \mathbf{W}_{j.}\mathbf{x} + b_j h_j)}{1 + \exp(b_j + \mathbf{W}_{j.}\mathbf{x})} \\
&= \prod_j p(h_j|\mathbf{x})
\end{aligned}
$$

Contrastive Divergence – Parameter Updates

- Derivation of $\frac{\partial E(x,h)}{\partial \theta}$ for $\theta = W_{jk}$

$$\frac{\partial E(x,h)}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}}\left(-\sum_{jk} W_{jk} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j\right)$$

$$= -\frac{\partial}{\partial W_{jk}} \sum_{jk} W_{jk} h_j x_k$$

$$= -h_j x_k$$

$$\nabla_W E(x,h) = -hx^T$$

Contrastive Divergence – Parameter Updates

- Derivation of $\mathrm{E}_h\left[\frac{\partial E(x,h)}{\partial \theta} \mid x\right]$ for $\theta = W_{jk}$

$$\mathrm{E}_h\left[\frac{\partial E(x,h)}{\partial W_{jk}} \mid x\right] = \mathrm{E}_h\left[-h_j x_k \mid x\right] = \sum_{h_j \in \{0,1\}} -h_j x_k p(h_j \mid x)$$

$$= -x_k p(h_j = 1 \mid x)$$

$$\mathrm{E}_h\left[\nabla_W E(x,h) \mid x\right] = -h(x)x^T$$

Contrastive Divergence – Parameter Updates

- Given $x^{(t)}$ and $\tilde{x}$ the learning rule for $\theta = W_{jk}$ becomes

$$
\begin{aligned}
W &\Leftarrow W - \alpha\left(\nabla_W - \log p(x^{(t)})\right) \\
&\Leftarrow W - \alpha\left(\mathrm{E}_h\left[\nabla_W E(x^{(t)}, h) \mid x^{(t)}\right] - \mathrm{E}_{x,h}\left[\nabla_W E(x, h)\right]\right) \\
&\Leftarrow W - \alpha\left(\mathrm{E}_h\left[\nabla_W E(x^{(t)}, h) \mid x^{(t)}\right] - \mathrm{E}_h\left[\nabla_W E(\tilde{x}, h) \mid \tilde{x}\right]\right) \\
&\Leftarrow W + \alpha\left(h(x^{(t)})x^{(t)T} - h(\tilde{x})\tilde{x}^T\right)
\end{aligned}
$$

Contrastive Divergence - Algorithm

- For each training example $x^{(t)}$
  - Generate a negative sample $\tilde{x}$ using k steps of Gibbs sampling, starting at $x^{(t)}$
  - Update parameters

$$W \Leftarrow W + \alpha\left(h(x^{(t)})x^{(t)T} - h(\tilde{x})\tilde{x}^{T}\right)$$

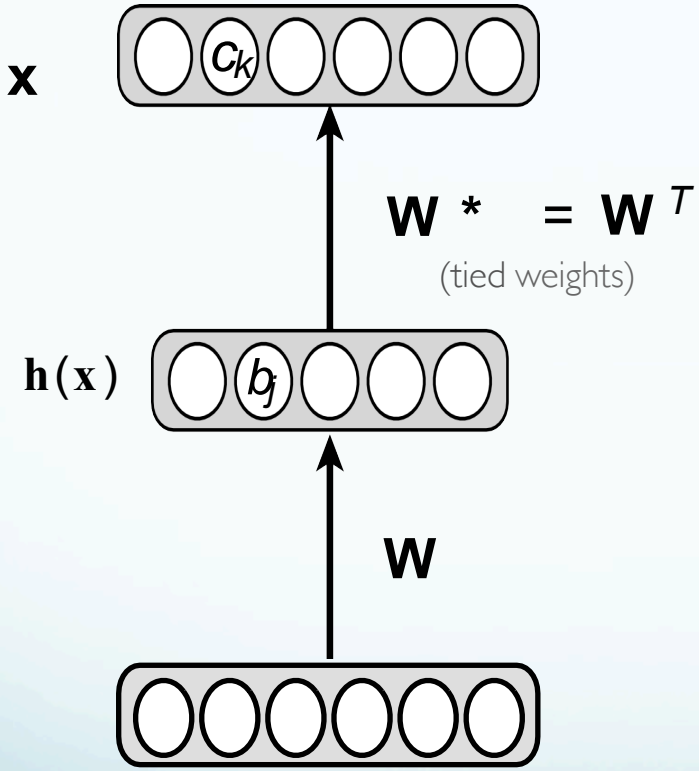
$$b \Leftarrow b + \alpha\left(h(x^{(t)}) - h(\tilde{x})\right)$$

$$c \Leftarrow c + \alpha\left(x^{(t)} - \tilde{x}\right)$$

- Repeat until stopping criterion

Autoencoders

Autoencoder



Decoder

$$\mathbf{x} = o(\mathbf{a}(\mathbf{x}))$$

$$= \text{sigm}(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x}))$$

for binary inputs

Encoder

$$\mathbf{h}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x}))$$

$$= \text{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x})$$

Autoencoder: Loss Function

- Loss function:
- Use $l(f(x)) = -\log p(x \mid \mu)$ as the loss function
- Example: We get the squared error loss

$$l(f(x)) = \frac{1}{2}\sum_k (\hat{x}_k - x_k)^2$$

  - By choosing a Gaussian distribution with mean μ and identity covariance

$$p(x \mid \mu) = \frac{1}{(2\pi)^{D/2}} \exp\left(-\frac{1}{2}\sum_k (x_k - \mu_k)^2\right)$$

    and choosing $\mu = c + W^* h(x)$

- For binary inputs:

$$l(f(x)) = -\sum_k (x_k \log(\hat{x}_k) + (1 - x_k)\log(1 - \hat{x}_k))$$

Autoencoder: Gradient

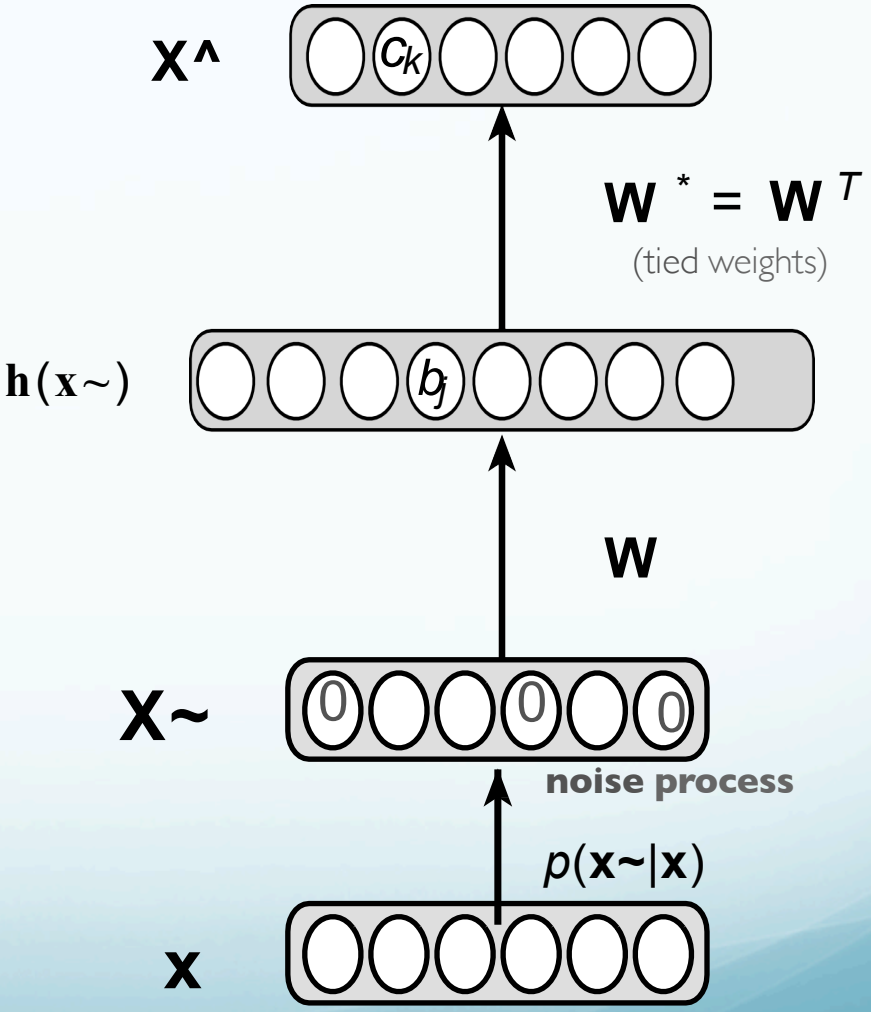
- For both binary and real valued inputs, the gradient has a very simple form

$$\nabla_{\hat{a}(x^{(t)})} l(f(x^{(t)})) = \hat{x}^{(t)} - x^{(t)}$$

- Parameter gradients are obtained by backpropagating the gradient like in a regular network

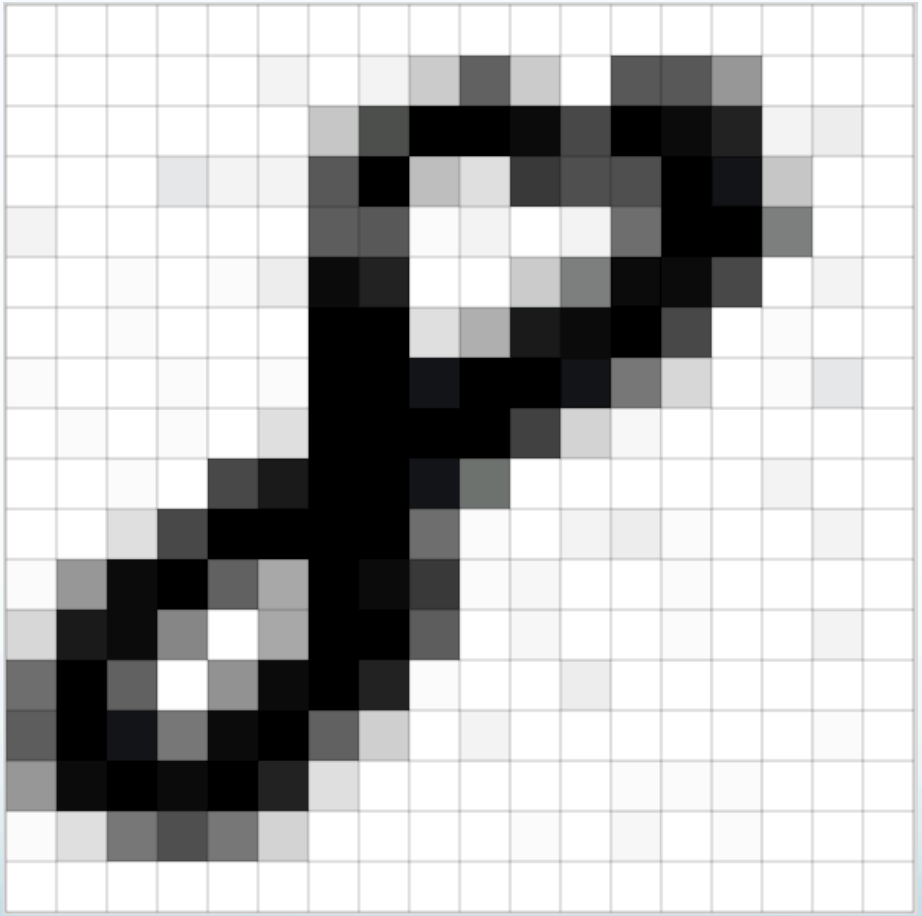
Denoising Autoencoder

- Representation should be robust to introduction of noise
  - Random assignment of subset of inputs to 0, with probability v
  - Gaussian additive noise
- Reconstruction computed from the corrupted input
- Loss function compares reconstruction with original input



Dataset

- MNIST ("Modified National Institute of Standards and Technology") is the de facto “hello world” dataset of computer vision.
- Contains gray-scale images of hand-drawn digits, from zero through nine.
- Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

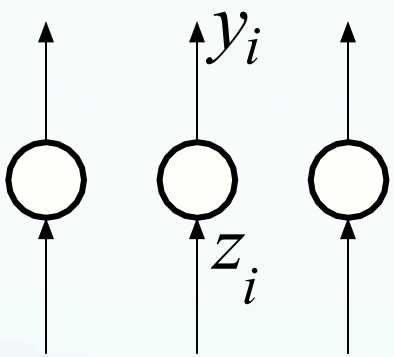


0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	12	0	11	39	137	37	0	152	147	84	0	0	0
0	0	1	0	0	0	41	160	250	255	235	162	255	238	206	11	13	0
0	0	0	16	9	9	150	251	45	21	184	159	154	255	233	40	0	0
10	0	0	0	0	0	145	146	3	10	0	11	124	253	255	107	0	0
0	0	3	0	4	15	236	216	0	0	38	109	247	240	169	0	11	0
1	0	2	0	0	0	253	253	23	62	224	241	255	164	0	5	0	0
6	0	0	4	0	3	252	250	228	255	255	234	112	28	0	2	17	0
0	2	1	4	0	21	255	253	251	255	172	31	8	0	1	0	0	0
0	0	4	0	163	225	251	255	229	120	0	0	0	0	0	11	0	0
0	0	21	162	255	255	254	255	126	6	0	10	14	6	0	0	9	0
3	79	242	255	141	66	255	245	189	7	8	0	0	5	0	0	0	0
26	221	237	98	0	67	251	255	144	0	8	0	0	7	0	0	11	0
125	255	141	0	87	244	255	208	3	0	0	13	0	1	0	1	0	0
145	248	228	116	235	255	141	34	0	11	0	1	0	0	0	1	3	0
85	237	253	246	255	210	21	1	0	1	0	0	6	2	4	0	0	0
6	23	112	157	114	32	0	0	0	0	2	0	8	0	7	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Softmax

The output units in a softmax group use a non-local non-linearity:

$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$



softmax group

$$\frac{\partial y_i}{\partial z_i} = y_i (1 - y_i)$$

Theano

- Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.
- Tight integration with NumPy
- Transparent use of a GPU.
  - Perform data-intensive computations much faster than on a CPU.
- Efficient symbolic differentiation.
  - Theano does your derivatives for functions with one or many inputs.
- Speed and stability optimizations.

Theano

- Installation and more information:
  - http://deeplearning.net/software/theano/
- Latest reference:
  - Theano: A Python framework for fast computation of mathematical expressions

Theano.Tensor

- Tensor
  - A Theano module for symbolic NumPy types and operations

```
In [1]: import theano.tensor as T
```

  - There are many types of symbolic expressions. Basic Tensor functionality includes creation of tensor variables such as scalar, vector, matrix, etc.

```
In [2]: c = T.scalar('c')
```

```
In [3]: v = T.vector('v')
```

```
In [4]: A = T.matrix('A')
```

- Create algebraic expressions on symbolic variables:

```
[In [6]: w = A.dot(v)
```

- Theano Functions

```
[In [7]: import theano

 In [8]: matrix_times_vector = theano.function(inputs=[A, v], outputs=w)
```

```
[In [10]: import numpy as np

[In [11]: A_val = np.array([[1,2], [3,4]])

[In [12]: v_val = np.array([5,6])

[In [13]: w_val = matrix_times_vector(A_val, v_val)

[In [14]: print w_val
 [ 17.  39.]
```

- Create a shared variable so we can do gradient descent

```
In [15]: x = theano.shared(20.0, 'x')
```

- Define a cost function that has a minimum value

```
In [16]: cost = x*x + x + 1
```

- Compute the gradient

```
In [17]: x_update = x - 0.3*T.grad(cost, x)
```

- Get data and initialize network

```
# step 1: get the data and define all the usual variables
X, Y = get_normalized_data()

max_iter = 20
print_period = 10

lr = 0.00004
reg = 0.01

Xtrain = X[:-1000,]
Ytrain = Y[:-1000]
Xtest  = X[-1000:,]
Ytest  = Y[-1000:]
Ytrain_ind = y2indicator(Ytrain)
Ytest_ind = y2indicator(Ytest)

N, D = Xtrain.shape
batch_sz = 500
n_batches = N / batch_sz

M = 300
K = 10
W1_init = np.random.randn(D, M) / 28
b1_init = np.zeros(M)
W2_init = np.random.randn(M, K) / np.sqrt(M)
b2_init = np.zeros(K)
```

- Define Theano variables and expressions

```
# step 2: define theano variables and expressions
thX = T.matrix('X')
thT = T.matrix('T')
W1 = theano.shared(W1_init, 'W1')
b1 = theano.shared(b1_init, 'b1')
W2 = theano.shared(W2_init, 'W2')
b2 = theano.shared(b2_init, 'b2')

# we can use the built-in theano functions to do relu and softmax
thZ = T.nnet.relu( thX.dot(W1) + b1 )
thY = T.nnet.softmax( thZ.dot(W2) + b2 )

# define the cost function and prediction
cost = -(thT * T.log(thY)).sum() + reg*((W1*W1).sum() + (b1*b1).sum() + (W2*W2).sum() + (b2*b2).sum())
prediction = T.argmax(thY, axis=1)
```

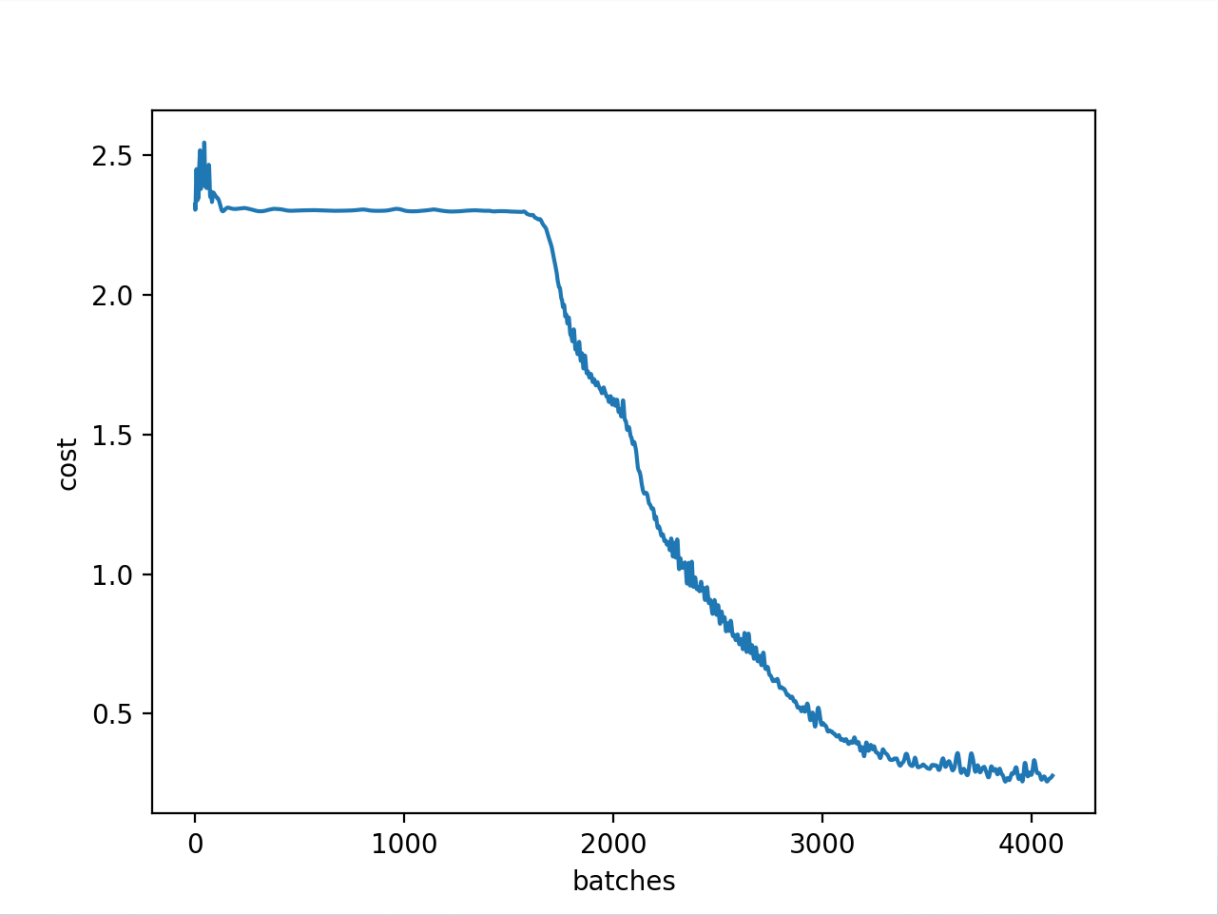
- Define training expressions and functions

```
# step 3: training expressions and functions
update_W1 = W1 - lr*T.grad(cost, W1)
update_b1 = b1 - lr*T.grad(cost, b1)
update_W2 = W2 - lr*T.grad(cost, W2)
update_b2 = b2 - lr*T.grad(cost, b2)

train = theano.function(
    inputs=[thX, thT],
    updates=[(W1, update_W1), (b1, update_b1), (W2, update_W2), (b2, update_b2)],
)
```

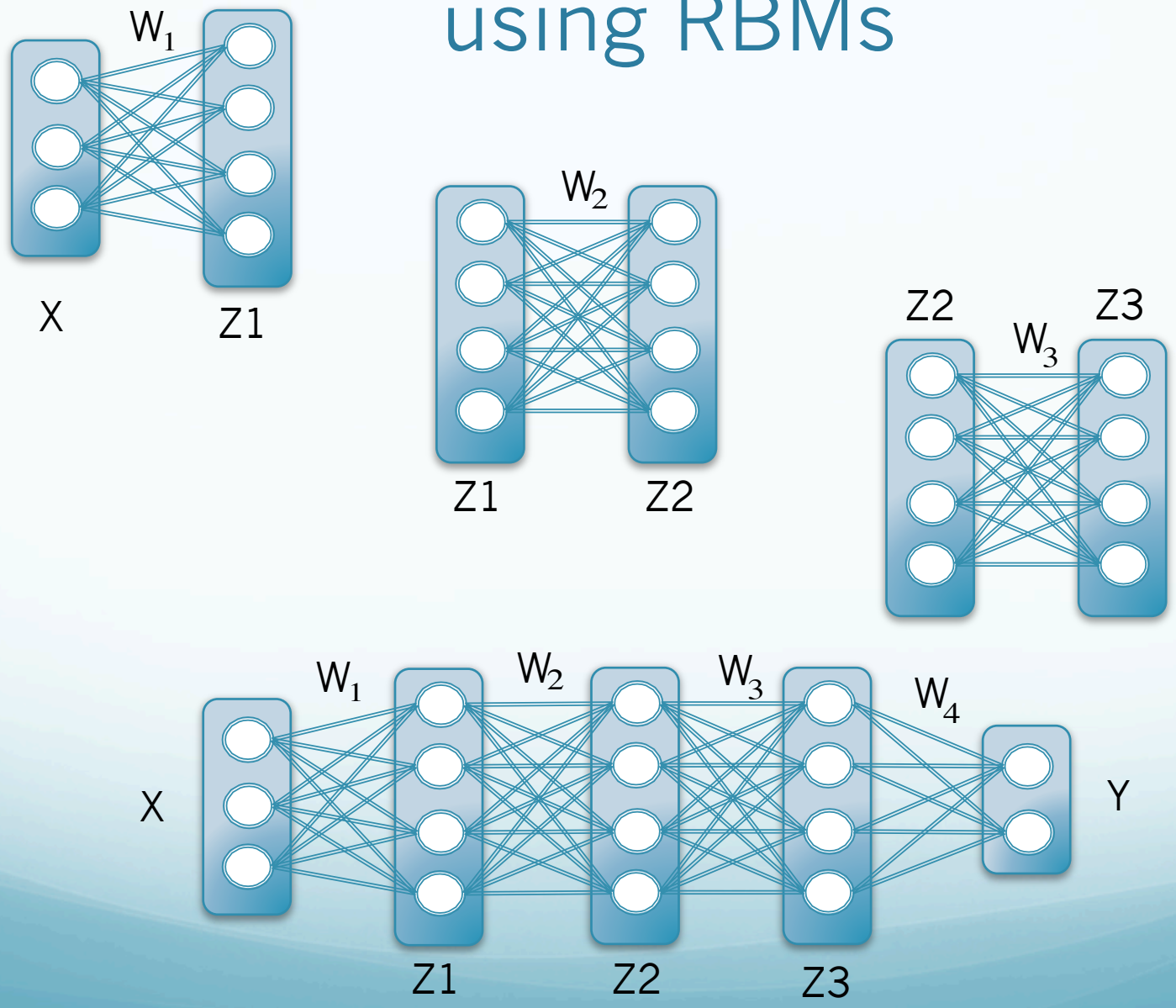
- Loop through the data in batches

```
costs = []
for i in xrange(max_iter):
    for j in xrange(n_batches):
        Xbatch = Xtrain[j*batch_sz:(j*batch_sz + batch_sz),]
        Ybatch = Ytrain_ind[j*batch_sz:(j*batch_sz + batch_sz),]

        train(Xbatch, Ybatch)
        if j % print_period == 0:
            cost_val, prediction_val = get_prediction(Xtest, Ytest_ind)
            err = error_rate(prediction_val, Ytest)
            print "Cost / err at iteration i=%d, j=%d: %.3f / %.3f" % (i, j, cost_val, err)
            costs.append(cost_val)

plt.plot(costs)
plt.show()
```

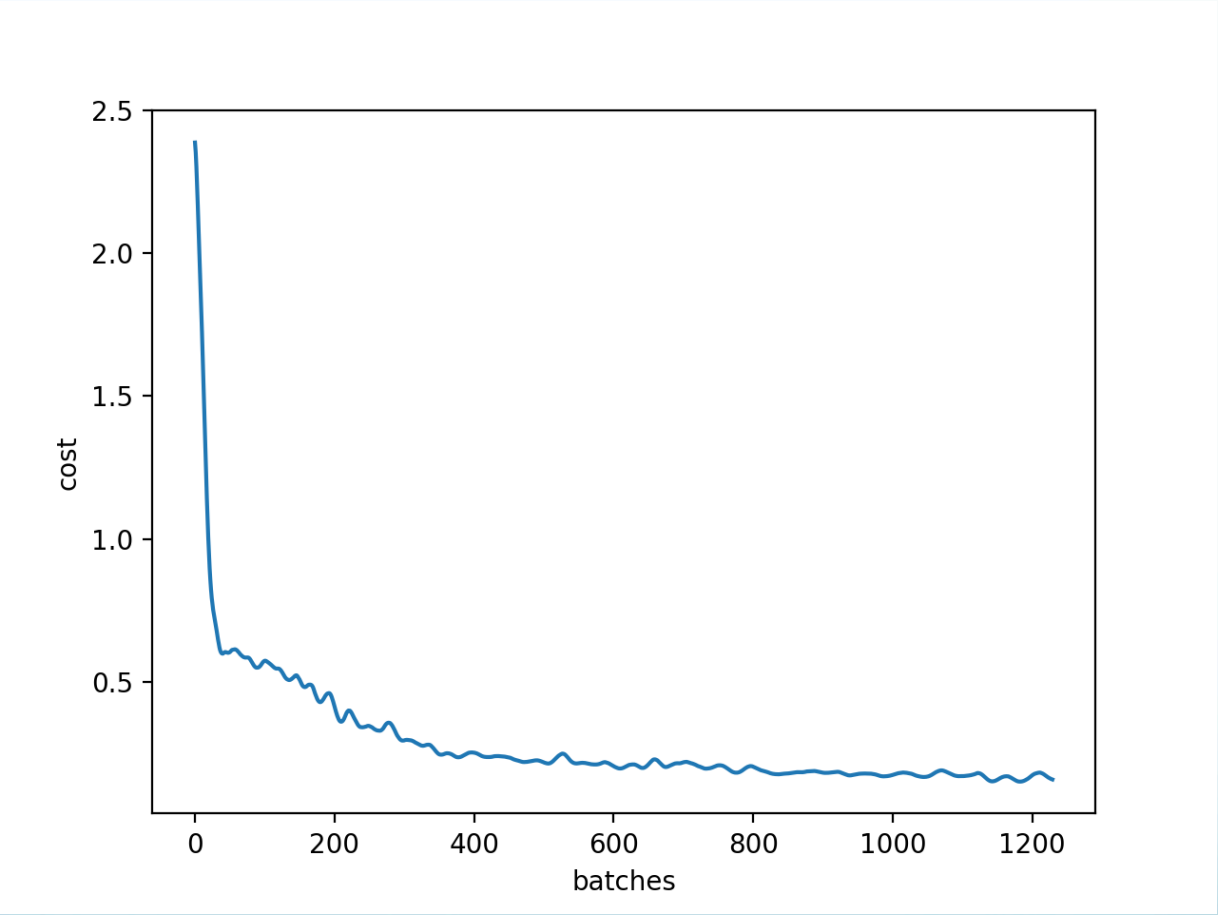
DNN without unsupervised pre-training



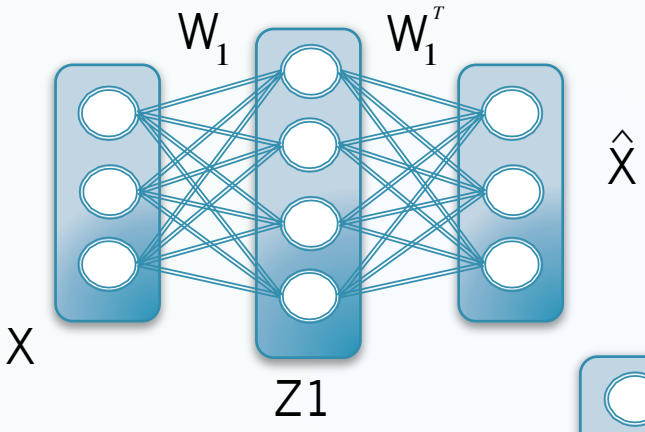
DNN with unsupervised pre-training using RBMs

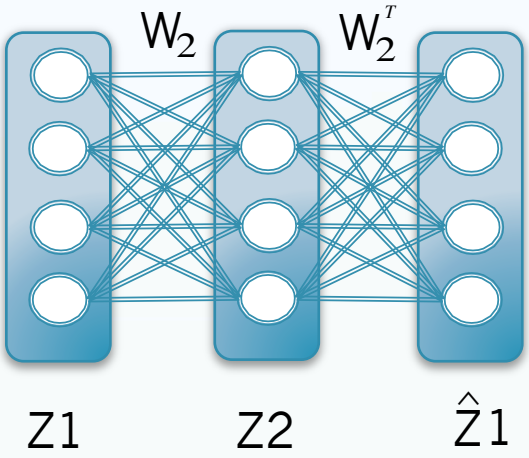


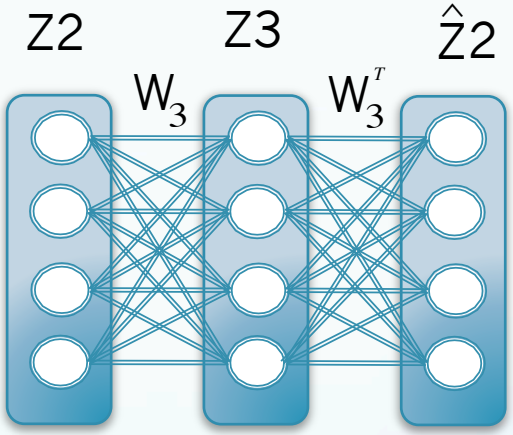
DNN unsupervised pre-training with RBMs

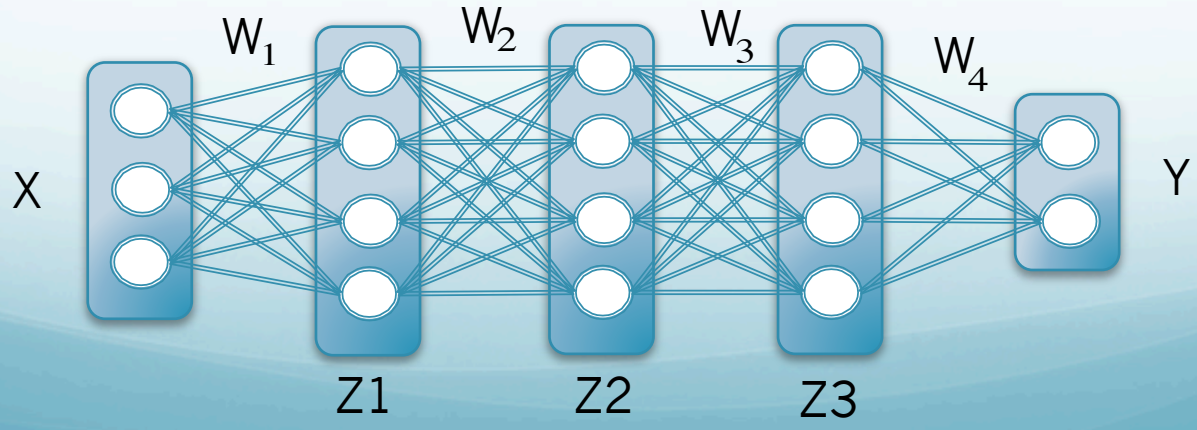


DNN with unsupervised pre-training using Autoencoders

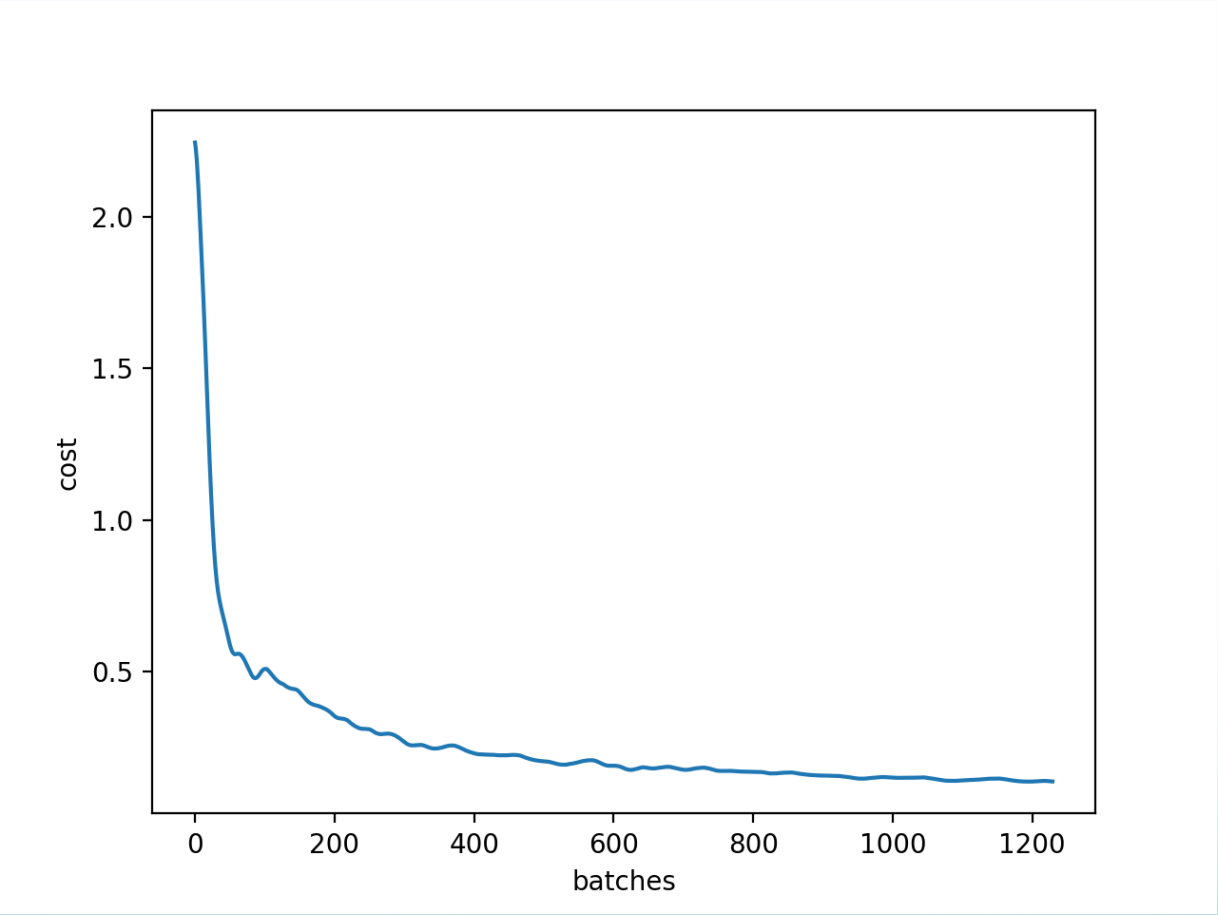








DNN unsupervised pre-training with AutoEncoders



Unsupervised Pre-training

- Layer-wise unsupervised learning:
  - The author believes that greedy layer-wise unsupervised pre-training overcomes the challenges of deep learning by introducing a useful prior to the supervised fine-tuning training procedure.
  - Gradients of a criterion defined at the output layer could become less useful as they are propagated backwards to lower layers.
  - The author states it is reasonable to believe unsupervised learning criterion defined at the level of a single layer could be used to move its parameters in a favorable direction.

Unsupervised Pre-training

- The author claims unsupervised pre-training acts as a regularizer by establishing an initialization point of the fine-tuning procedure inside a region of parameter space in which the parameters are henceforth restricted.
- Another motivation is that greedy layer-wise unsupervised pre-training could be a way to naturally decompose the problem into sub-problems associated with different levels of abstraction and extract salient information about the input distribution and capture in a distributed representation.

References

- Yoshua Bengio, *Learning deep architecures for AI*: http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf
- Hugo Larochelle, Neural networks: http://info.usherbrooke.ca/hlarochelle/neural_networks/content.html
- Geoff Hinton, *Neural networks for machine learning*: https://www.coursera.org/learn/neural-networks
- Goodfellow et al., *Deep Learning*: http://www.deeplearningbook.org/
- Montufar et al., *On the number of linear regions of Deep Networks*: https://arxiv.org/pdf/1402.1869.pdf

References

- Pascanu et al., *On the number of response regions of deep feed forward neural networks with piece-wise linear* activations: https://arxiv.org/pdf/1312.6098.pdf
- Perceptron weight update graphics: https://www.cs.utah.edu/~piyush/teaching/8-9-print.pdf
- Proof of Theorem (Block, 1962, and Novikoff, 1962). http://cs229.stanford.edu/notes/cs229-notes6.pdf
- Python and Theano code reference: https://deeplearningcourses.com/

References

- Erhan et al., Why does unsupervised pre-training help deep learning: http://www.jmlr.org/papers/volume11/erhan10a/erhan10a.pdf
- Theano installation and more information:

  http://deeplearning.net/software/theano/
- Theano latest reference:

  Theano: A Python framework for fast computation of mathematical expressions