KDnuggets Cartoon

"The machine learning algorithm wants to know if we'd like a dozen wireless mice to feed the Python book we just bought."

Deep Learning 1

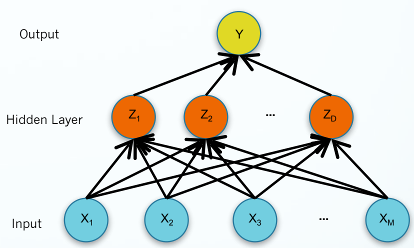
Mark Dredze
Some slides by Matt Gormley

Machine Learning
CS 601.475

Outline

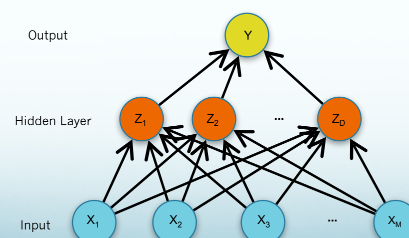
Recap of Neural Nets

- Motivation:
  - Learn features automatically
  - Capture non-linearities of data
- Two layers of binary logistic regression define a two-layer neural network
- Neural networks are universal approximators
  - Very powerful class of hypotheses



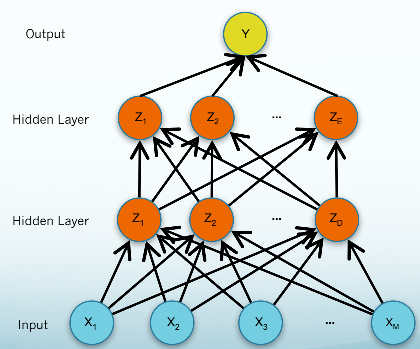
Deeper Networks

Last lecture:



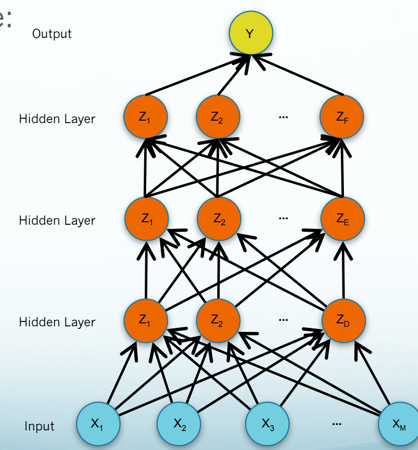
Deeper Networks

This lecture:



Deeper Networks

This lecture: Making the neural networks deeper



Motivation: Why go Deep?

- 2-layer Neural Nets are already universal function approximators!
- A neural network with 1 hidden layer is a universal function approximator
  - For any continuous function g(x) there exists a 1-hidden layer neural network $h_\theta(x)$ with sigmoid activation functions such that
    $$|h_\theta(x) - g(x)| < \epsilon \forall x$$
  - Cybenko (1989)

Motivation: Why go Deep?

- Before 2006: deep networks are harder to train so let's stick with shallow networks
- After 2006: deep networks are easier to train for many problems
- Why are they easier? You need to know the right set of tricks!

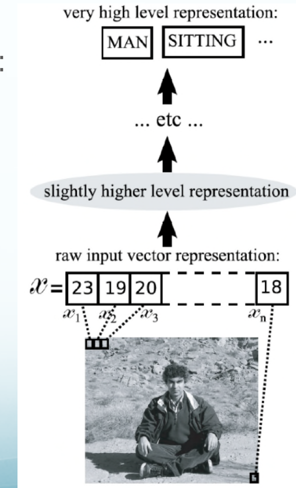
Motivation: Why go Deep?

- Why is it easier to train?
  - Deep architectures can be representationally efficient
- Deep representations allow for a hierarchy, non-local generalizations
  - Possible with shallow networks, but fewer computational units for the same function in deep network
- Deep Nets: Multiple levels of latent variables allow combinatorial sharing of statistical strength

Slide adapted from Honglak Lee (NIPS 2010)

The Promise of Deep Architectures

- Transform input image into higher levels of representation:
  - edges, local shapes, object parts, etc.
- We don't know the "right" levels of abstraction
- So let the model figure it out!

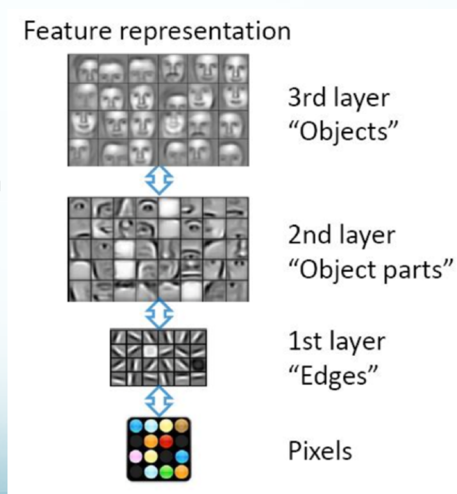


Example from Bengio (2009)

Different Levels of Abstraction

Face Recognition:

- Deep Network can build up increasingly higher levels of abstraction
- Lines, parts, regions
- Sharing abstraction: learn "object parts" once and all higher layers can use it



Example from Honglak Lee (NIPS 2010)

NN Packages

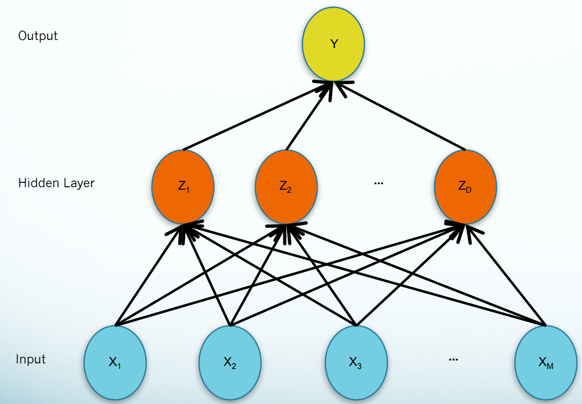
- Why are neural network packages so popular and successful?
  - PyTorch
  - TensorFlow
  - Caffe
  - mxnet
  - CNTK

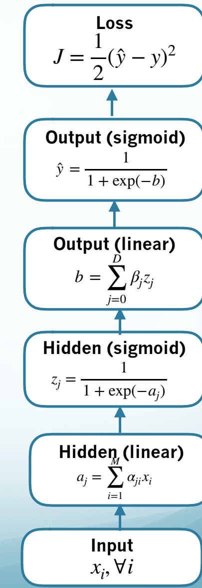
Modularity

- The core building blocks can be combined to create new models
  - Library provides core building blocks
  - User combines them into models
- Don't we need to write out the gradients for our models?

Network Structure

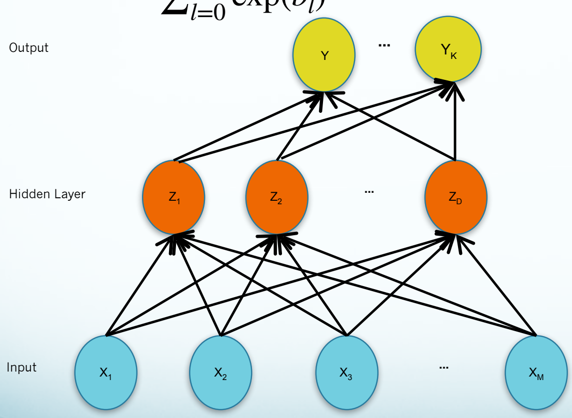
$$y(\mathbf{x}) = h^{(2)}\left(\sum_{j=1}^{M} \beta_j^{(2)} h^{(1)}\left(\sum_{i=1}^{D} \alpha_{ji}^{(1)} x_i\right)\right)$$

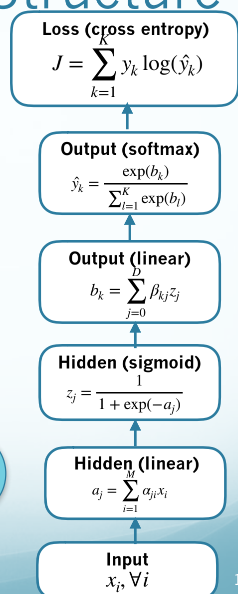




Multi-Class Network Structure

Softmax $$\hat{y}_k = \frac{\exp(b_k)}{\sum_{l=0}^{K} \exp(b_l)}$$





Common ML Training

- Given training data $\{x_i, y_i\}_{i=1}^{N}$
- Select
  - Prediction function (network structure)
  - Loss function
- Train model to minimize loss function
  - Stochastic gradient descent

$$\theta^{t+1} = \theta^t - \eta_t \nabla \ell(f_\theta(x_i), y_i)$$

Compute Gradients

- We need a way to:
  - Compute gradients of arbitrary network structure
  - Make the gradient computation efficient

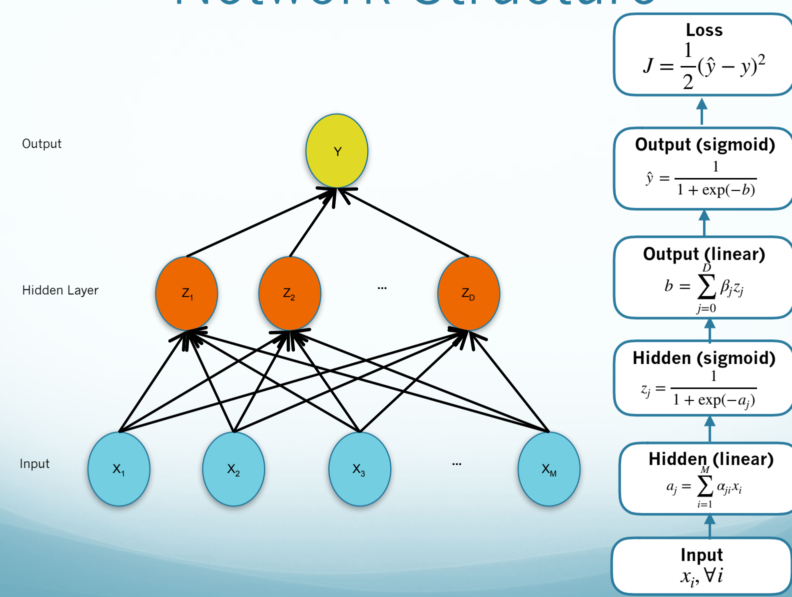
Automatic Differentiation

- Write the objective function as a combination of smaller building blocks
  - Algorithm will **automatically** compute the derivative for use in learning
- Wikipedia description: a set of techniques to numerically evaluate the derivative of a function... [which] exploits the fact that every computer program... executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically

Backpropogation

- Backprop of errors in MLPs is a special case of automatic differentiation
- AD allows us to compute gradient of arbitrary functions as long as we can specify how the function breaks down into parts
  - Computation graph: a DAG where each node is a variable in the function
  - Where would we get such a graph?

Network Structure



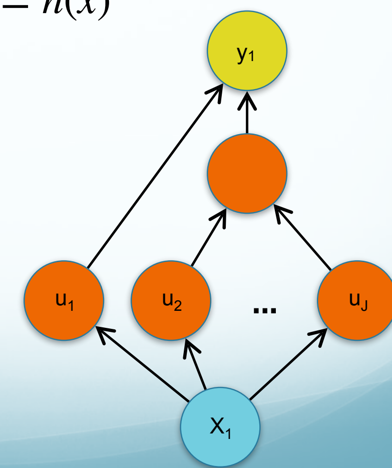
Forward Propogation

- Forward computation
  - Write out the network structure as a directed acyclic graph of computations
    - "Computation graph"
  - Visit each node in topological order
    - For each variable u_i with inputs $v_1 \dots v_N$
    - Compute $u_i = g(v_1 \dots v_N)$
    - Store the result at the node
  - You now have network output, as well as all internal nodes

Training: Chain Rule

- Given $y = g(u)$ $\quad u = h(x)$
- Chain rule:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j}\frac{du_j}{dx_k}, \forall i, k$$



Slide from Matt Gormley

Backward Propogation

- Backward computation
  - Initialize all partial derivatives

$$\frac{\partial y}{\partial u_j} = 0 \qquad \frac{\partial y}{\partial y} = 1$$

  - Visit each node in reverse topological order
  - For variable $u_i = g(v_1 \dots v_N)$
    - We already know $\frac{\partial y}{\partial u_i}$
    - Increment $\frac{\partial y}{\partial v_j}$ by $\frac{\partial y}{\partial u_i}\frac{\partial u_i}{\partial v_j}$
    - We will choose g to make this easy
- Return: $\frac{\partial y}{\partial u_j} \forall u_j$

Slide from Matt Gormley

Training | **Backpropagation**

Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward

$J = cos(u)$

$u = u_1 + u_2$

$u_1 = sin(t)$

$u_2 = 3t$

$t = x^2$

Slide from Matt Gormley

Training | Backpropagation

Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward	Backward
$J = cos(u)$	$\frac{dJ}{du} \mathrel{+}= -sin(u)$
$u = u_1 + u_2$	$\frac{dJ}{du_1} \mathrel{+}= \frac{dJ}{du}\frac{du}{du_1}, \quad \frac{du}{du_1} = 1 \qquad \frac{dJ}{du_2} \mathrel{+}= \frac{dJ}{du}\frac{du}{du_2}, \quad \frac{du}{du_2} = 1$
$u_1 = sin(t)$	$\frac{dJ}{dt} \mathrel{+}= \frac{dJ}{du_1}\frac{du_1}{dt}, \quad \frac{du_1}{dt} = \cos(t)$
$u_2 = 3t$	$\frac{dJ}{dt} \mathrel{+}= \frac{dJ}{du_2}\frac{du_2}{dt}, \quad \frac{du_2}{dt} = 3$
$t = x^2$	$\frac{dJ}{dx} \mathrel{+}= \frac{dJ}{dt}\frac{dt}{dx}, \quad \frac{dt}{dx} = 2x$

Slide from Matt Gormley

Common ML Training w/ Backprop

- Given training data $\{x_i, y_i\}_{i=1}^N$
- Select
  - Prediction function (network structure)
  - Loss function
- Train model to minimize loss function
  - Compute all partial derivatives using backprop

$$\theta^{t+1} = \theta^t - \eta_t \nabla \ell(f_\theta(x_i), y_i)$$

  - Stochastic gradient descent

Common ML Training w/ Backprop

- Given training data $\{x_i, y_i\}_{i=1}^N$
- Select
  - Prediction function (network structure)
  - Loss function
- Train model to minimize loss function
  - Compute all partial deriva

$$\theta^{t+1} = \theta^t - \eta$$

  - Stochastic gradient desce

PyTorch

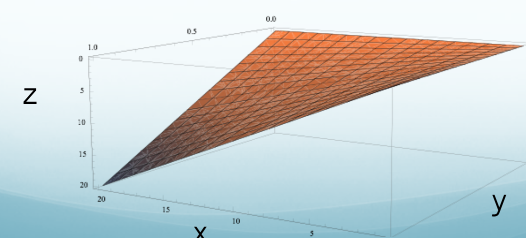
```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

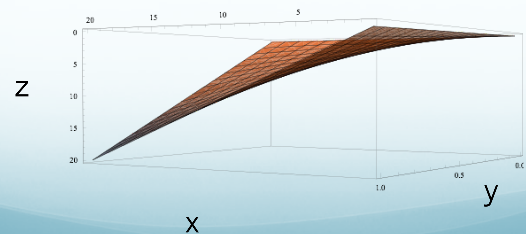
The problem: *Nonconvexity*

- Where does the nonconvexity come from?
- Even a simple quadratic z = xy objective is nonconvex:
- Neural networks: Composition of convex functions is not convex
  - Universal approximators: convex functions can't (well) approximate non-convex functions!



The problem: *Nonconvexity*

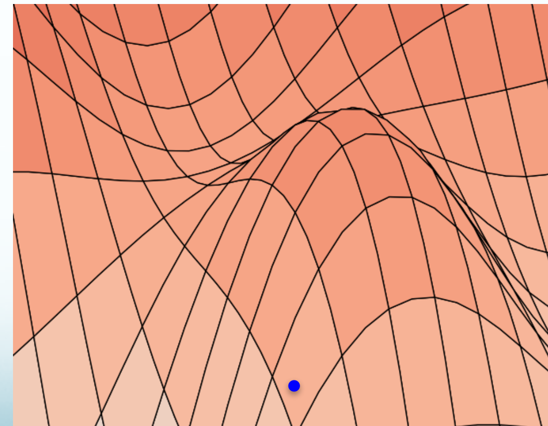
- Where does the nonconvexity come from?
- Even a simple quadratic $z = xy$ objective is nonconvex:
- Neural networks: Composition of convex functions is not convex
  - Universal approximators: convex functions can't (well) approximate non-convex functions!



The problem: *Nonconvexity*

Stochastic Gradient Descent...

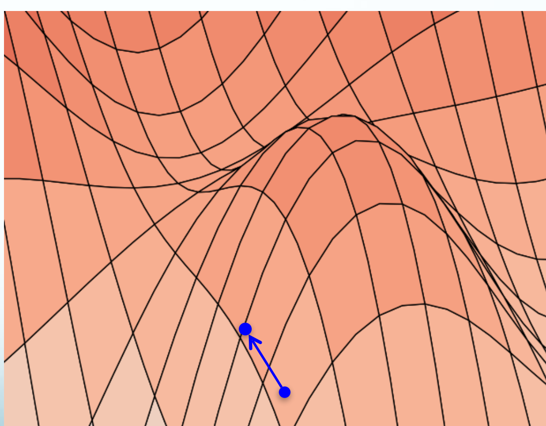
...climbs to the top of the nearest hill...



The problem: *Nonconvexity*

Stochastic Gradient Descent...

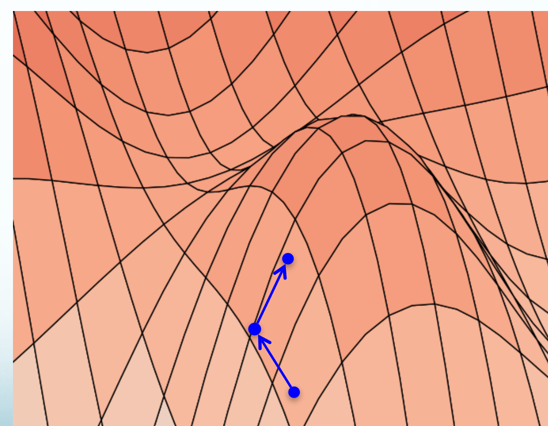
...climbs to the top of the nearest hill...



The problem: *Nonconvexity*

Stochastic Gradient Descent...

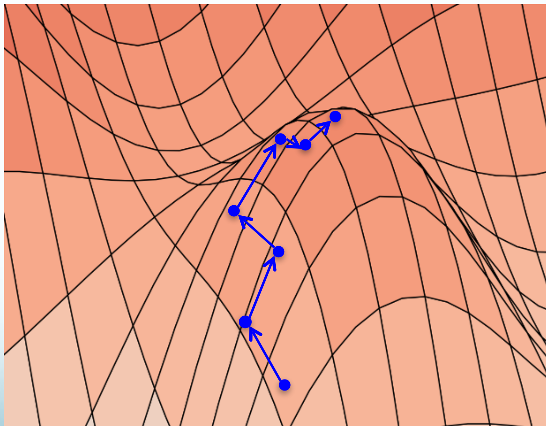
...climbs to the top of the nearest hill...



The problem: *Nonconvexity*

Stochastic Gradient Descent…

…climbs to the top of the nearest hill…

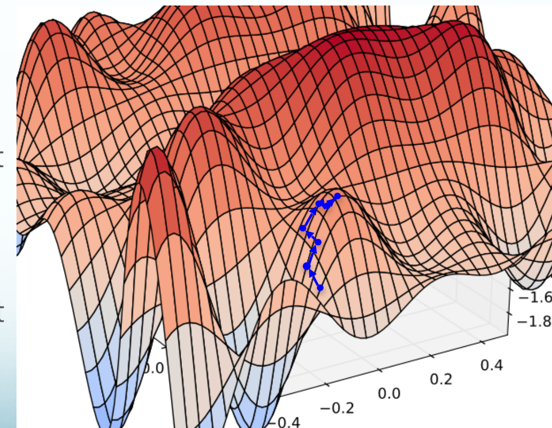


The problem: *Nonconvexity*

Stochastic Gradient Descent…

…climbs to the top of the nearest hill…

…which might not lead to the top of the mountain



Why Does SGD Work?

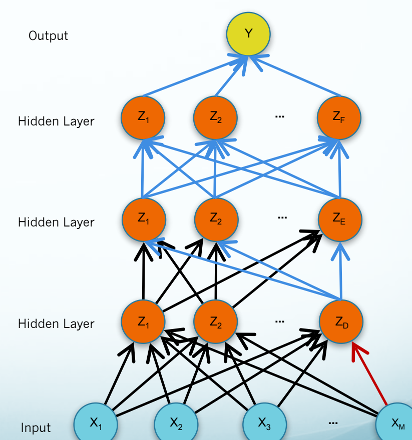
- Even non-convex SGD will converge*
  - *Converge = arbitrarily small gradients
- The “best” solution isn’t the only “good” solution
  - Big data: with enough data you can find reasonable solutions
- Learning rates
  - We have lots of adaptive algorithms
- Random restarts/extensive hyper-parameter optimization
- Regularization
  - Will discuss next time

Problem: *Vanishing Gradients*

The gradient for an edge at the base of the network depends on the gradients of many edges above it

The chain rule multiplies many of these gradients together

We’ll discuss next time



Deep Network Training

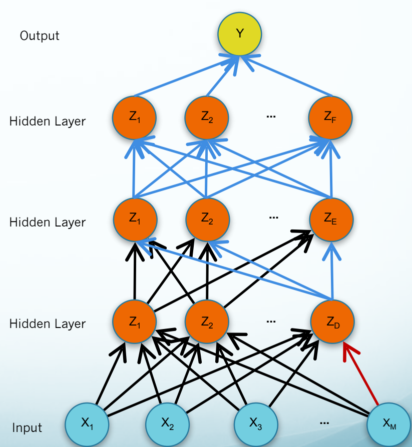
- Deep networks are successful because
  - We got smarter about the training algorithms
    - Though not *that* much smarter
  - Faster computers
    - We can train for much longer, bigger networks, hyper-parameter optimization
  - More data
    - These networks are *very* data hungry
    - Simple baselines work better in small data settings

Data

- We have lots and lots of data!
- Labeled data
  - Still not much labeled data for many tasks
- How can we make use of unlabeled data for neural network training?

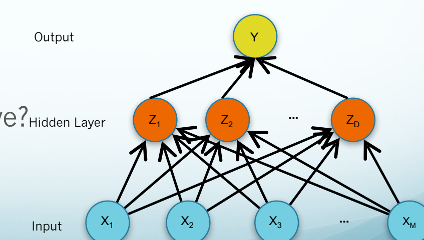
Unsupervised Training Insight

- The hidden layers of the network are learning representations of the data
- Only the last layer of the network directly depends on knowing the label y
- Can we learn better representations on unlabeled data, and then prediction parameters on labeled data?

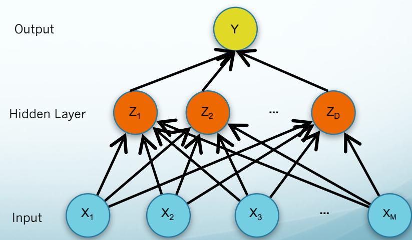


Unsupervised pre-training

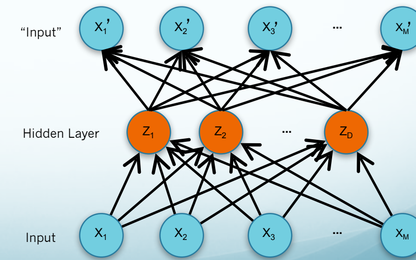
- Pre-training
  - Training before you actually (supervised) train
- Pre-train early layers of the network
  - What should it predict?
  - What else do we observe?
  - **The input!**



Unsupervised pre-training



Unsupervised pre-training



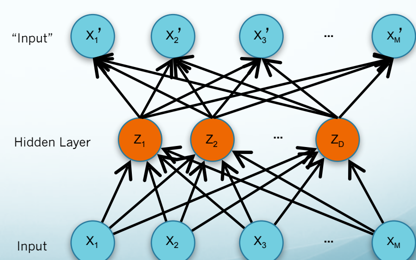
Auto-Encoders

Key idea: Encourage z to give small reconstruction error:

- x' is the *reconstruction* of x
- Loss = || x – DECODER(ENCODER(x)) ||^2
- Train with the same backpropagation algorithm for 2-layer Neural Networks with x_m as both input and output.

DECODER: x' = h(W'z)

ENCODER: z = h(Wx)

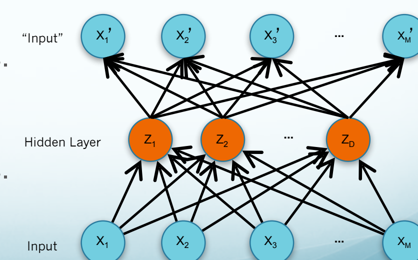


Slide adapted from Raman Arora

Unsupervised pre-training

Unsupervised pre-training

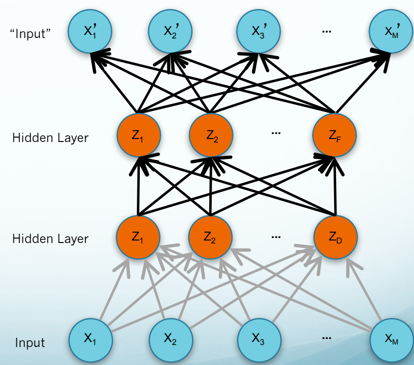
- Work bottom-up
  - Train hidden layer 1. Then fix its parameters.
  - Train hidden layer 2. Then fix its parameters.
  - …
  - Train hidden layer n. Then fix its parameters.



Unsupervised pre-training

Unsupervised pre-training

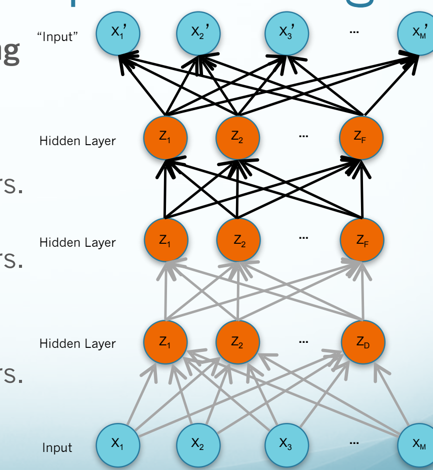
- Work bottom-up
  - Train hidden layer 1.
    Then fix its parameters.
  - Train hidden layer 2.
    Then fix its parameters.
  - …
  - Train hidden layer n.
    Then fix its parameters.



Unsupervised pre-training

Unsupervised pre-training

- Work bottom-up
  - Train hidden layer 1.
    Then fix its parameters.
  - Train hidden layer 2.
    Then fix its parameters.
  - …
  - Train hidden layer n.
    Then fix its parameters.

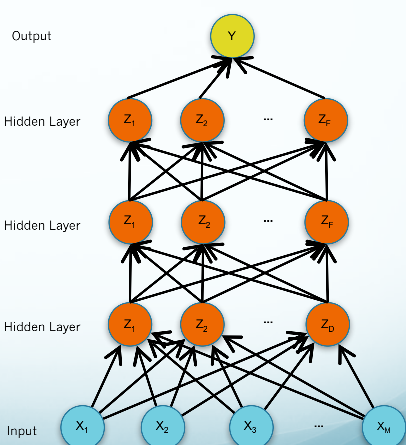


Unsupervised pre-training

Unsupervised pre-training

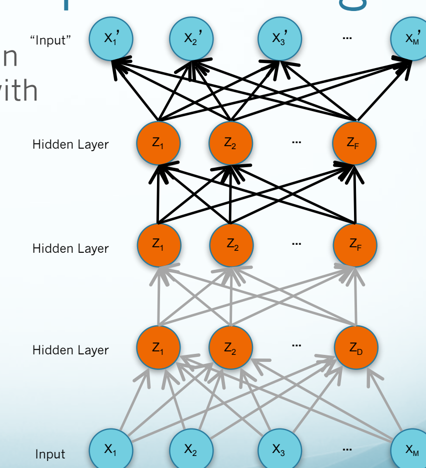
- Work bottom-up
  - Train hidden layer 1.
    Then fix its parameters.
  - Train hidden layer 2.
    Then fix its parameters.
  - …
  - Train hidden layer n.
    Then fix its parameters.

Supervised fine-tuning
Backprop and update all parameters



Unsupervised pre-training

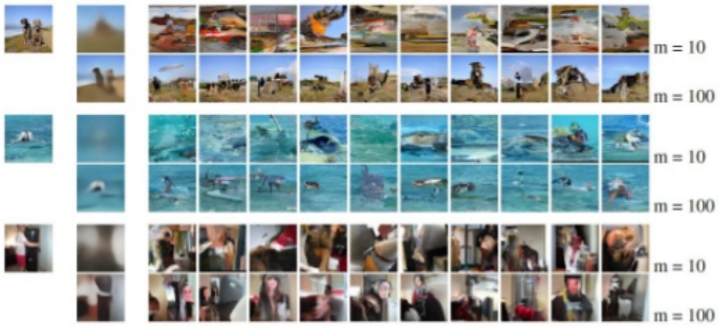
- In practice, we can train all the layers at once with an auto-encoder



Experimental Results (PixelCNN Auto-Encoder)

- Data: 32x32 ImageNet patches

(m: dimensional bottleneck)



(Left to right: original image, reconstruction by auto-encoder, conditional samples from PixelCNN auto-encoder)

https://www.slideshare.net/suga93/conditional-image-generation-with-pixelcnn-decoders

Denoising Auto-encoders

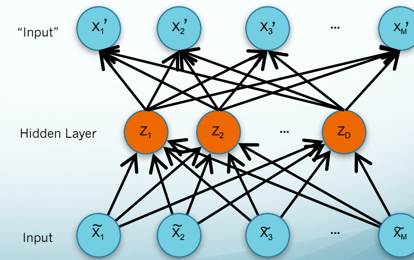
Key idea: Encourage z to give small reconstruction error

- x' is the *reconstruction* of $\tilde{x}$ = x+noise
- Loss = || x – DECODER(ENCODER(x + noise)) ||^2
- Train with the same backpropagation algorithm

DECODER: x' = h(W'z)

ENCODER: z = h(W$\tilde{x}$)

where $\tilde{x}$ = x+noise



Slide adapted from Raman Arora