CS 383 – Machine Learning

Deep Learning

Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

Objectives

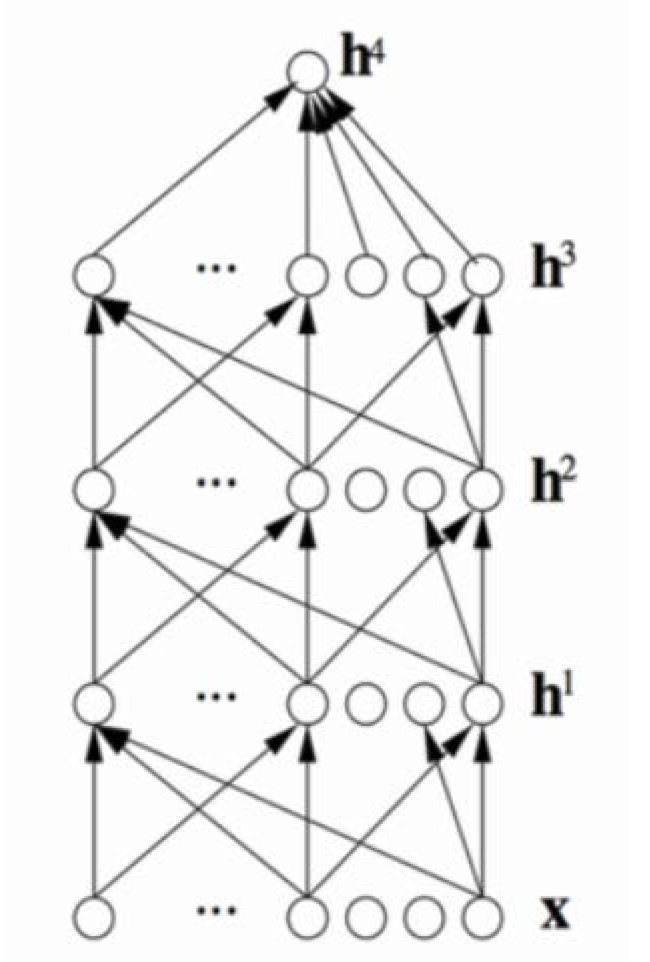
- Deep Learning

Deep Learning

- Most current machine learning works well because of human-designed representations and input features
- Machine learning becomes just optimizing weights to best make a final prediction
- **Deep learning** algorithms attempt to learn multiple levels of representation of increasing complexity/abstraction

Deep Architecture

- Same idea as regular ANNs but with additional hidden layers.
- Output layer – Here predicting a supervised target
- Hidden layers – These learn more abstract representations as you head up
- Input layer – Raw sensory inputs (roughly)

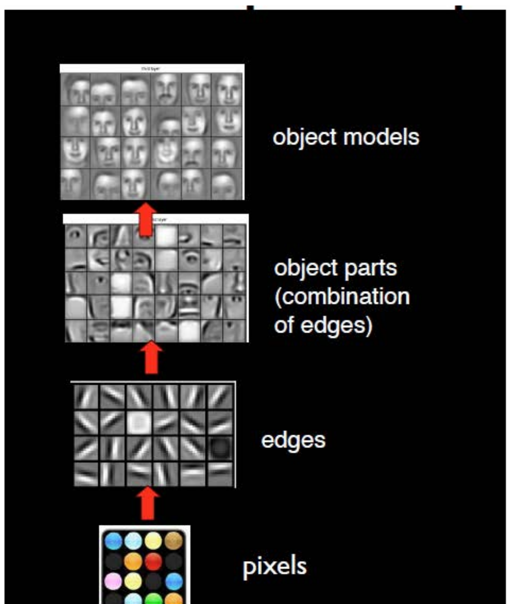


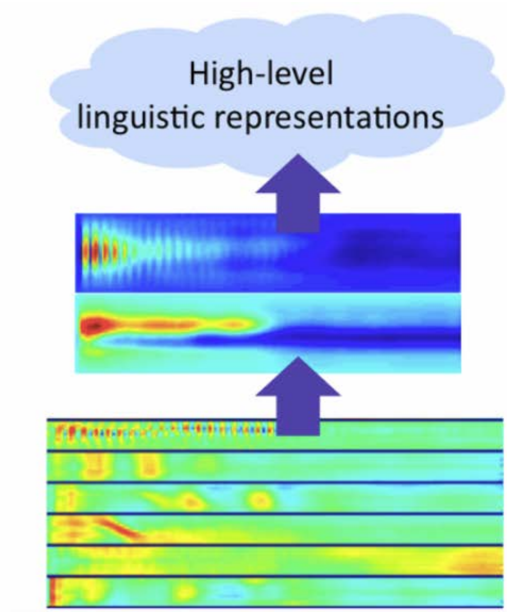
Why Deep Learning?

- Handcrafting features is time-consuming
- The features are often both over-specific and incomplete
- The works has to be done again for each task/domain
- We must move beyond handcrafted features and simple ML
  - Humans develop representations for learning and reasoning
  - Our computers should do the same

Deep Learning Overview

- Multiple layers work to build an improved feature space
  - First layer learns 1st order features (e.g. edges, etc..)
  - 2nd layer learns high order features (combinations of first layer features, combination of edges, etc..)
  - Then final layer features are fed into supervised layer(s)





Training Deep Networks

- So how do we train a deep network?
- One idea is just to generalize the forward-backwards propagation algorithm
- Let's assume there are M layers such that layer $m = 1$ is the input layer and layer $\mathrm{m} = M$ is the output layer and that the matrix $W^{m,m+1}$ is the set of weights going from layer m to layer $m + 1$.
- Also assuming batch processing, let's denote the values coming in and out of each node at layer m as in^m and out^m, respectively.

Training Deep Networks

- So how do we train a deep network?
- One idea is just to generalize the forward-backwards propagation algorithm
  - Compute the loss at output layer $m = M$
    $$\delta^M = Y - out^M$$
  - Update the weights from the second-to-last layer to the output layer:
    $$W^{M-1,M} = W^{M-1,M} + \frac{\eta}{N}(out^{M-1})^T \delta^M$$
  - For each layer $m = M-1, \ldots, 1$
    - Compute the loss (note the element-wise .* operator):
      $$\delta^m = \delta^{m+1} W^{m,m+1} .* \, out^m .* (1 - out^m)$$
    - Update the weights:
      $$W^{m-1,m} = W^{m-1,m} + \frac{\eta}{N}(out^{m-1})^T \delta^m$$

Training Deep Networks

- So how do we train a deep network?
- One idea is just to generalize the forward-backwards propagation algorithm
- Problems:
  - Late layers (ones year the output) learn quicky/well and therefore early layers (near the input stage) have little error and therefor become relatively useless.
  - Slow
- We want a way to allow early layers to contribute so that
  - We will hopefully converge quicker
  - We learn “intermediate” concepts

Greedy Layer-wise Training

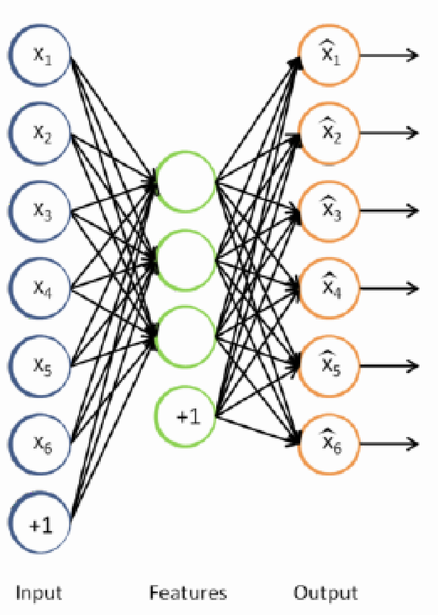
- Most deep learning approaches use a greedy method:
  1. Train first layer using just your input data
  2. "Freeze" the first layer parameters and start training the second layer using the output of the first layer
  3. Repeat 1-2 for as many hidden layers as desired
  4. Use the output of the final layer as inputs to the output layers, using labeled data to train this final set of weights
  5. (Optional) Unfreeze all weights and fine tune the full network by training with a supervised approach, given the pre-processed weight settings (aka do complete forwards-backwards propagation)

Types of Deep Networks

- There are several ways to train the intermediate layers
  - Restricted Botlzmann Machines (RBMs)
    - Computes the energy of the current configuration in order to define probability distributions for the current configuration.
  - Auto-encoders
    - Attempts to find a hidden layer that can reproduce the input
- We’ll just look at auto-encoders

Auto-Encoders

- We want to find a ANN that can reproduce the input
- A trivial solution would be to have the hidden layer be the same size as the input and then just have all the weights be one or zero
- To avoid this, we typically corrupt the input (add some noise) and try to learn the uncorrupted output

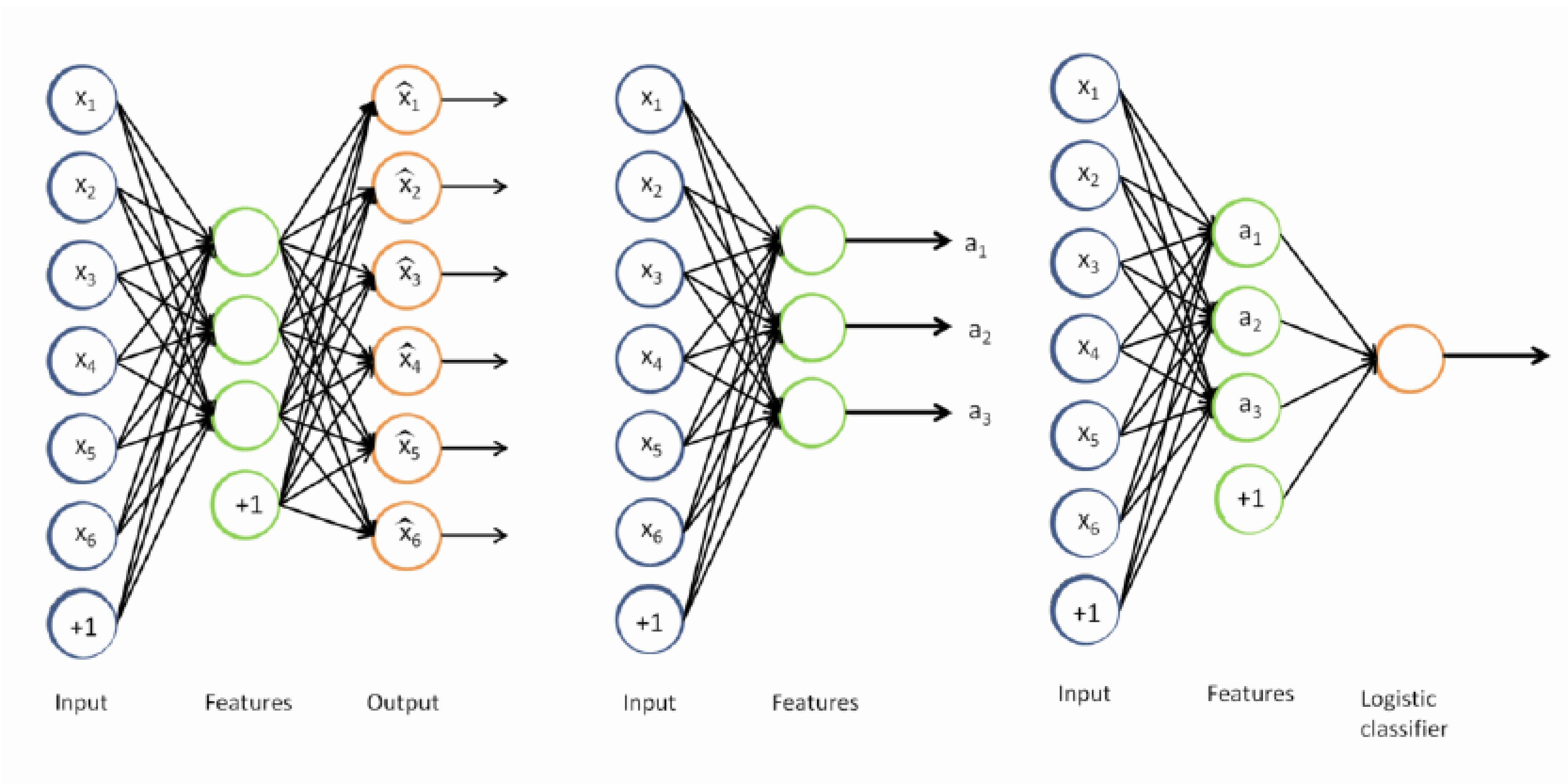


One Auto-Encoder

- So the basic process to get a hidden layer from one auto-encoder is:

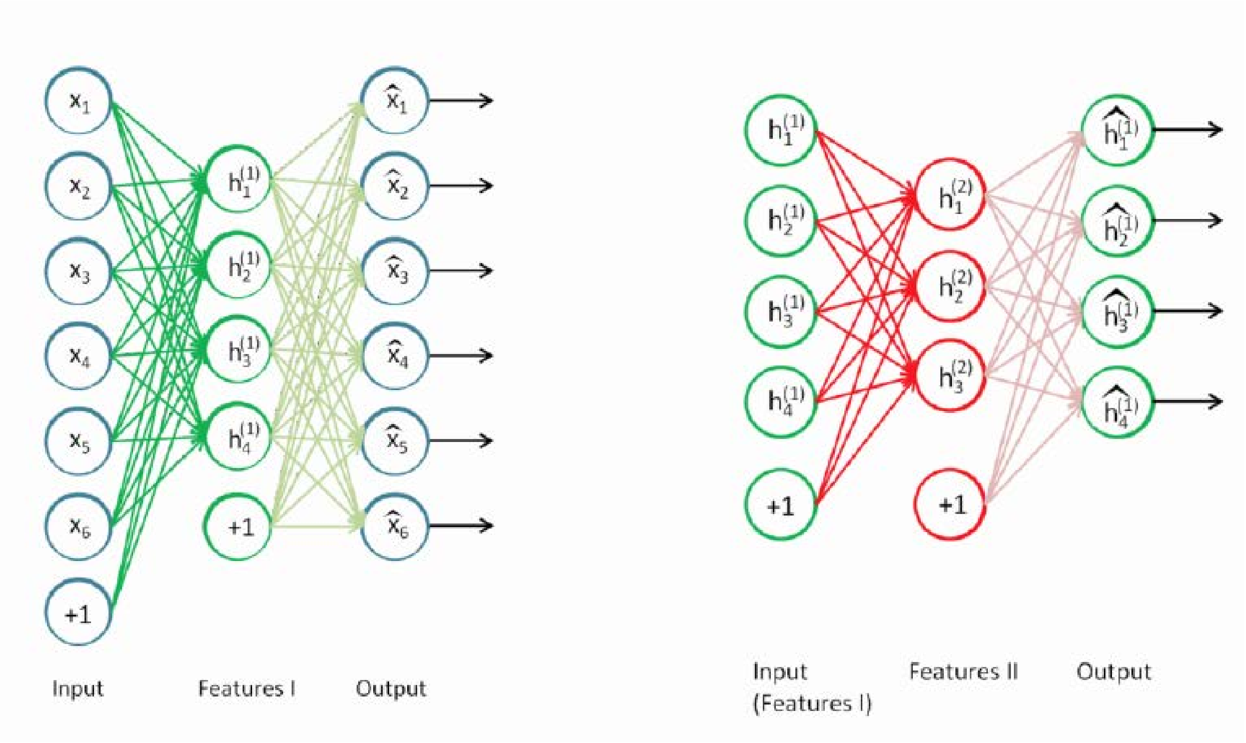
1. Take the input, add some noise to it, and add a bias node
2. Choose the hidden layer size to be less than the input size
3. The output layer should be the same size as the input (minus the bias node)
4. Train this auto-encoder using the uncorrupted data as the desired output values.
5. After training, remove the output layer (and its weights). Now you have your hidden layer to act as the input to the next layer!

One Auto-Encoder



Stacked Auto-encoders

- Stack sparse auto-encoders on top of each other, drop decode layer each time



Stacked auto-encoders

- Do supervised training on last layer
- Then do supervised training on whole network to fine tune the weights