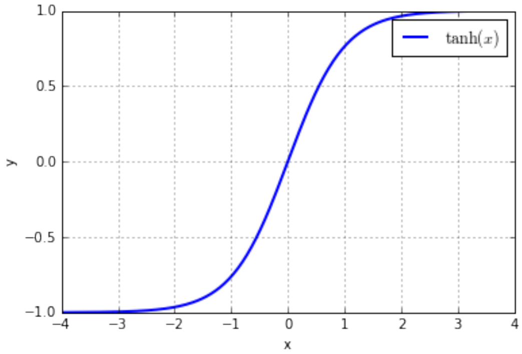
Are there alternatives to Sigmoid Hidden Units?

Hidden Unit Transfer Functions
Initialising Deep Networks

Steve Renals

Machine Learning Practical — MLP Lecture 6
28 October 2015

tanh

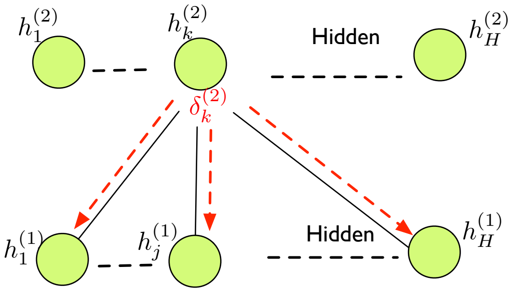


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad ; \quad \text{sigmoid}(x) = \frac{1 + \tanh(x/2)}{2}$$

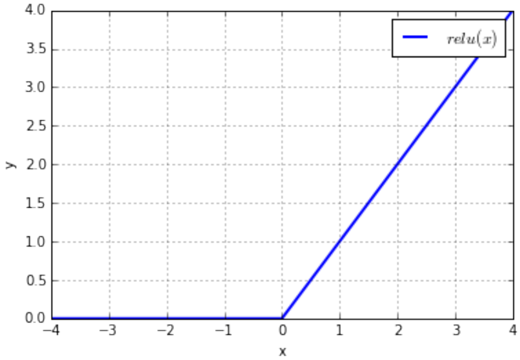
Derivative: $$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

tanh hidden units

- tanh has same shape as sigmoid but has output range ± 1
- Results about approximation capability of sigmoid networks also apply to tanh networks
- Possible reason to prefer tanh over sigmoid: allowing units to be positive or negative allows gradient for weights into a hidden unit to have a different sign



Rectified Linear Unit – ReLU



$$\mathrm{relu}(x) = \max(0, x)$$

Derivative: $$\frac{d}{dx}\mathrm{relu}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

ReLU hidden units

- Similar approximation results to tanh and sigmoid hidden units
- Empirical results for speech and vision show consistent improvements using relu over sigmoid or tanh
- Unlike tanh or sigmoid there is no positive saturation – saturation results in very small derivatives (and hence slower learning)
- Negative input to relu results in zero gradient (and hence no learning)
- Relu is computationally efficient: $\max(0, x)$
- Relu units can "die" (i.e. respond with 0 to everything)
- Relu units can be very sensitive to the learning rate

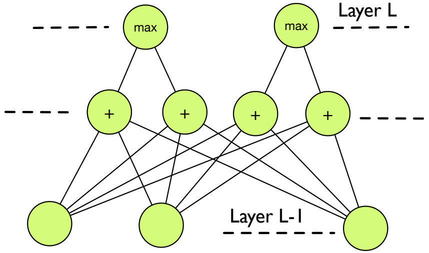
Maxout units

- Unit that takes the max of two linear functions $z_i = \mathbf{w}^i \mathbf{h}^{\mathbf{L-1}}$:

$$h = \max(z_1, z_2)$$

(if $\mathbf{w}^2 = 0$ then we have Relu)

- Has the benefits of Relu (piecewise linear, no saturation), without the drawback of dying units
- Twice the number of parameters



Generalising maxout

- Units can take the max over G linear functions z_i:

$$h = \max_{i=0}^{G}(z_i)$$

- Maxout can be generalised to other functions, e.g. p-norm

$$h = ||\mathbf{z}||_p = \left(\sum_{i=0}^{G} |z_i|^p \right)^{1/p}$$

  Typically $p = 2$

- p can be learned by gradient descent.
  (Exercise: What is the gradient $\partial E / \partial p$ for a p-norm unit?)

How should we initialise deep networks?

Initialising deep networks (Pretraining)

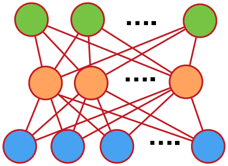
- Why is training deep networks hard?
  - Vanishing (or exploding) gradients – gradients for layers closer to the input layer are computed multiplicatively using backprop
  - If sigmoid/tanh hidden units near the output saturate then back-propagated gradients will be very small
  - Good discussion in chapter 5 of *Neural Networks and Deep Learning*
- Solve by *stacked* pretraining
  - Train the first hidden layer
  - Add a new hidden layer, and train only the parameters relating to the new hidden layer. Repeat.
  - The use the pretrained weights to initialise the network – emphfine-tune the complete network using gradient descent
- Approaches to pre-training
  - Supervised: Layer-by-layer cross-entropy training
  - Unsupervised: Autoencoders
  - Unsupervised: Restricted Boltzmann machines (not covered in this course)

Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

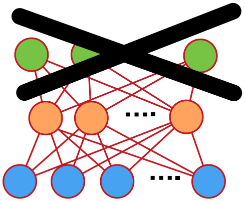
Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough



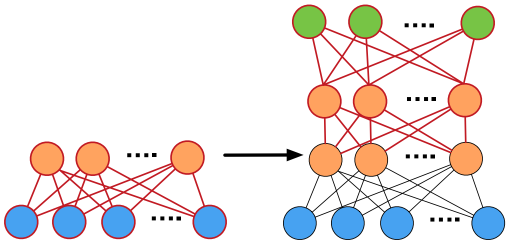
Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough



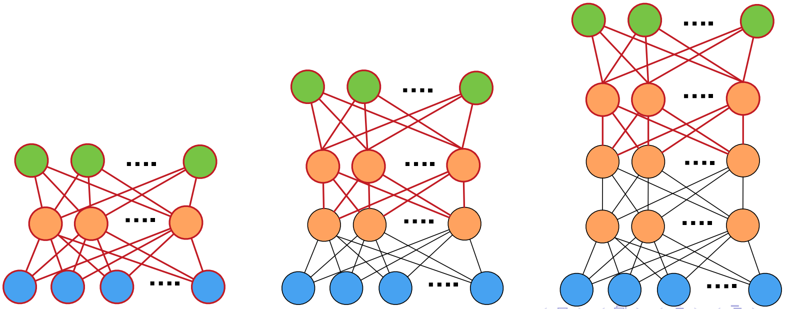
Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough



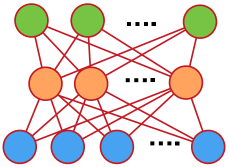
Greedy Layer-by-layer cross-entropy training

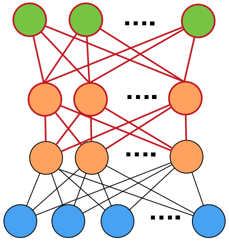
1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

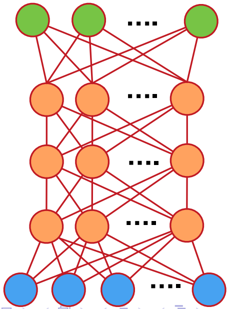


Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

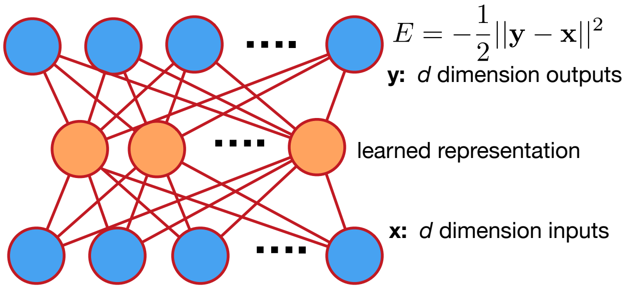






Autoencoders

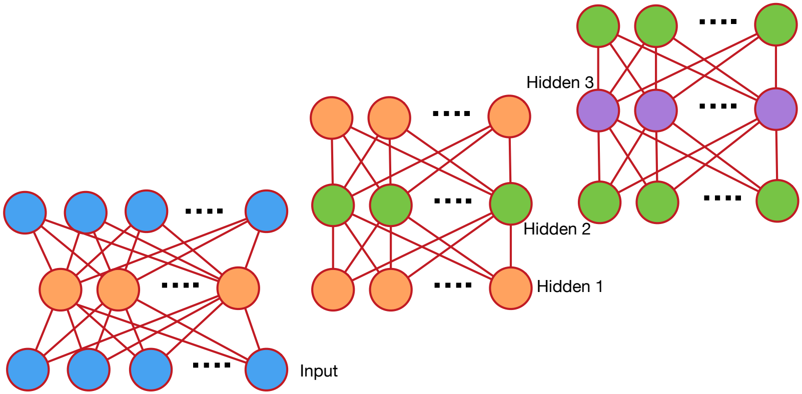
- An autoencoder is a neural network trained to map its input into a distributed representation from which the input can be reconstructed
- Example: single hidden layer network, with an output the same dimension as the input, trained to reproduce the input using squared error cost function


$E = -\frac{1}{2}||\mathbf{y} - \mathbf{x}||^2$
y: *d* dimension outputs
learned representation
x: *d* dimension inputs

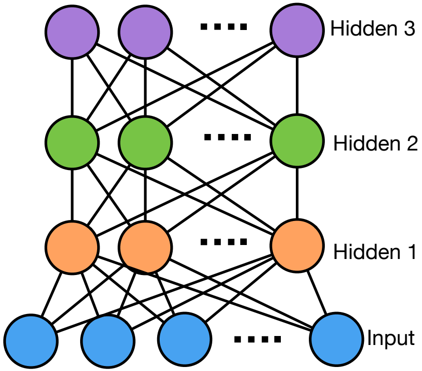
Stacked autoencoders

- Can the hidden layer just copy the input (if it has an equal or higher dimension)?
  - In practice experiments show that nonlinear autoencoders trained with stochastic gradient descent result in useful hidden representations
  - Early stopping acts as a regulariser
- **Stacked autoencoders** – train a sequence of autoencoders, layer-by-layer
  - First train a single hidden layer autoencoder
  - Then use the learned hidden layer as the input to a new autoencoder

Stacked Autoencoders

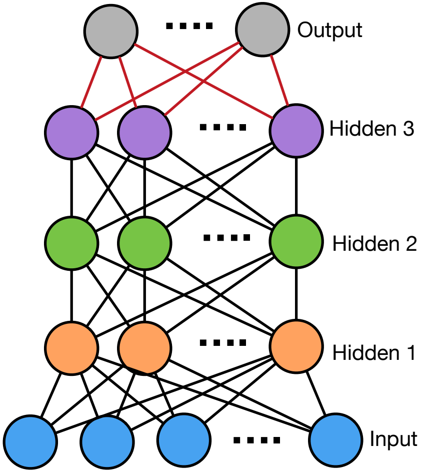


Pretraining using Stacked autoencoder



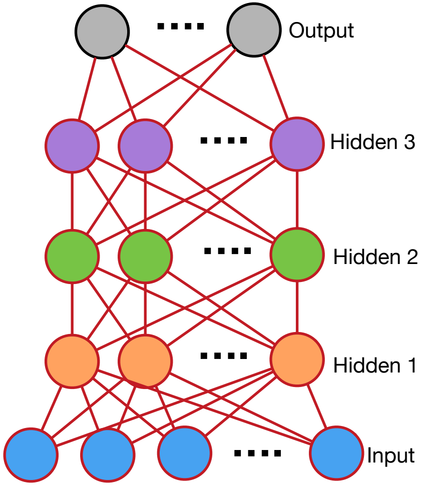
Initialise hidden layers

Pretraining using Stacked autoencoder



Train output layer

Pretraining using Stacked autoencoder

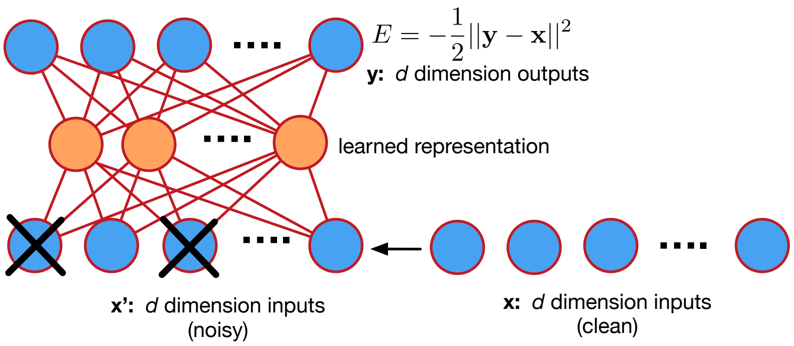


Fine tune whole network

Denoising Autoencoders

- Basic idea: Map from a corrupted version of the input to a clean version (at the output)
- Forces the learned representation to be stable and robust to noise and variations in the input
- To perform the denoising task well requires a representation which models the important structure in the input
- The aim is to learn a representation that is robust to noise, not to perform the denoising mapping as well as possible
- Noise in the input:
    - Random **Gaussian** noise added to each input vector
    - **Masking** – randomly setting some components of the input vector to 0
    - **Salt & Pepper** – randomly setting some components of the input vector to 0 and others to 1
- Stacked denoising autoencoders – noise is only applied to the input vectors, not to the learned representations

Denoising Autoencoder



$E = -\frac{1}{2}||\mathbf{y} - \mathbf{x}||^2$
y: *d* dimension outputs
learned representation
x': *d* dimension inputs (noisy)
x: *d* dimension inputs (clean)

Summary

- Hidden unit transfer functions: tanh, ReLU, Maxout
- Layer-by-layer Pretraining and Autoencoders
  - For many tasks (e.g. MNIST) pre-training seems to be necessary / useful for training deep networks
  - For some tasks with very large sets of training data (e.g. speech recognition) pre-training may not be necessary
  - (Can also pre-train using stacked restricted Boltzmann machines)
- Reading: Michael Nielsen, chapter 5 of *Neural Networks and Deep Learning*
  http://neuralnetworksanddeeplearning.com/chap5.html
  Pascal Vincent et al, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", JMLR, 11:3371–3408, 2010.
  http://www.jmlr.org/papers/volume11/vincent10a/vincent10a.pdf