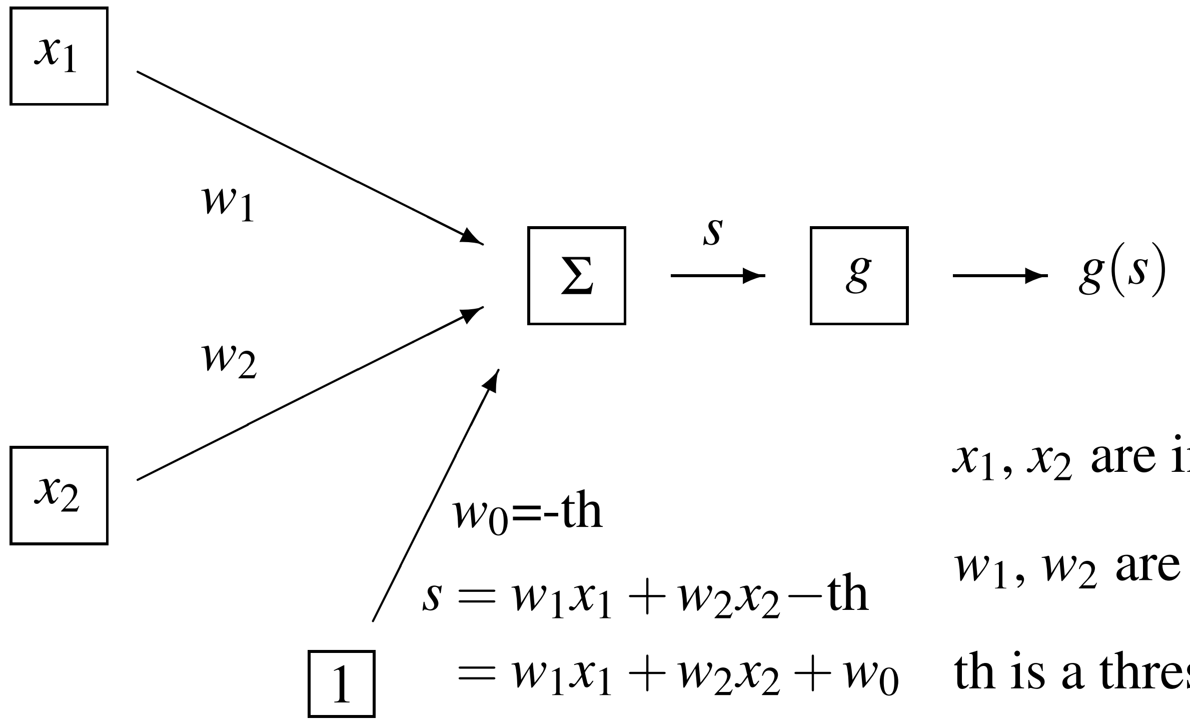
COMP9444
Neural Networks and Deep Learning

10b. Summary

McCulloch & Pitts Model of a Single Neuron



$$\begin{aligned} s &= w_1x_1 + w_2x_2 - \text{th} \\ &= w_1x_1 + w_2x_2 + w_0 \end{aligned}$$

x_1, x_2 are inputs

w_1, w_2 are synaptic weights

th is a threshold

w_0 is a **bias** weight

g is transfer function

Perceptron Learning Rule

Adjust the weights as each input is presented.

recall: $s = w_1 x_1 + w_2 x_2 + w_0$

if $g(s) = 0$ but should be 1,

$$\begin{aligned} w_k &\leftarrow w_k + \eta\, x_k \\ w_0 &\leftarrow w_0 + \eta \\ \text{so} \quad s &\leftarrow s + \eta\,(1 + \sum_k x_k^2) \end{aligned}$$

if $g(s) = 1$ but should be 0,

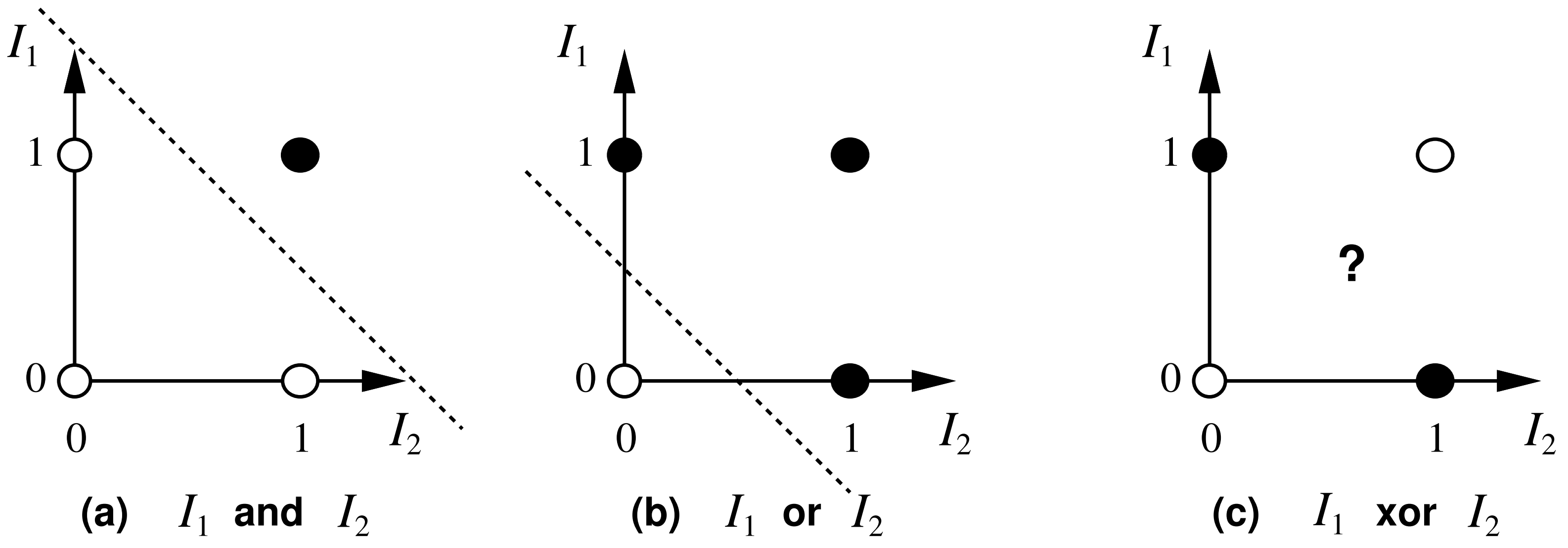
$$\begin{aligned} w_k &\leftarrow w_k - \eta\, x_k \\ w_0 &\leftarrow w_0 - \eta \\ \text{so} \quad s &\leftarrow s - \eta\,(1 + \sum_k x_k^2) \end{aligned}$$

otherwise, weights are unchanged. ($\eta > 0$ is called the **learning rate**)

Theorem: This will eventually learn to classify the data correctly, as long as they are **linearly separable**.

Limitations of Perceptrons

Problem: many useful functions are not linearly separable (e.g. XOR)

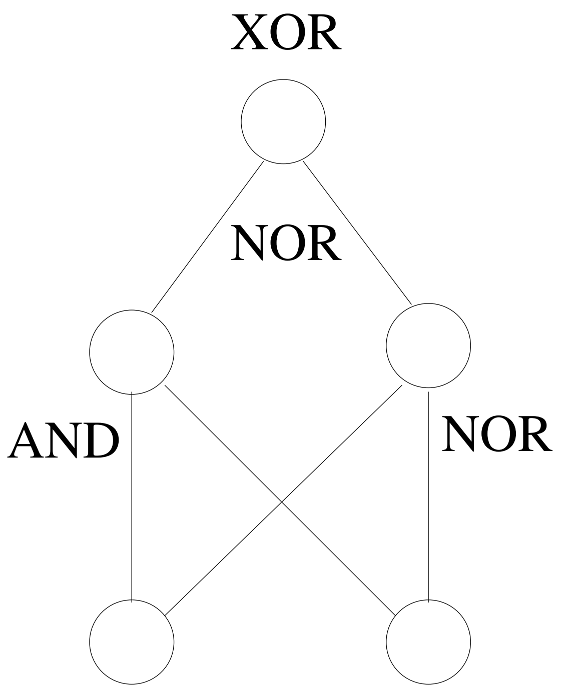


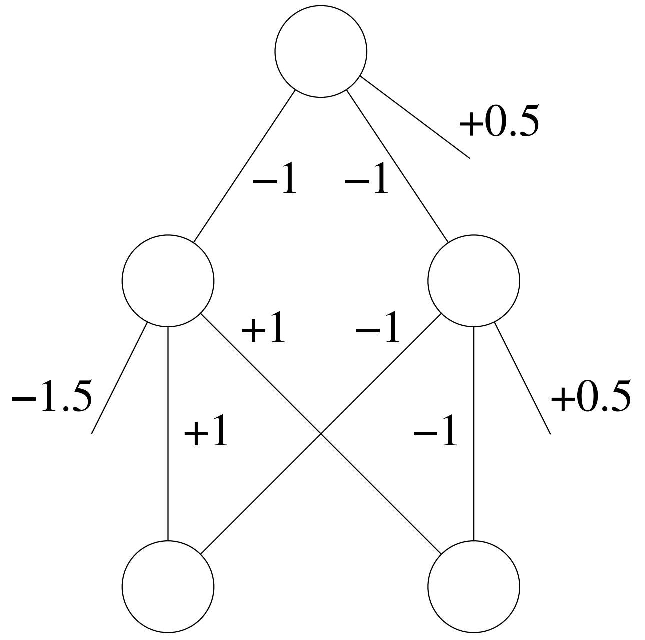
Possible solution:

x_1 XOR x_2 can be written as: (x_1 AND x_2) NOR (x_1 NOR x_2)

Recall that AND, OR and NOR can be implemented by perceptrons.

Multi-Layer Neural Networks





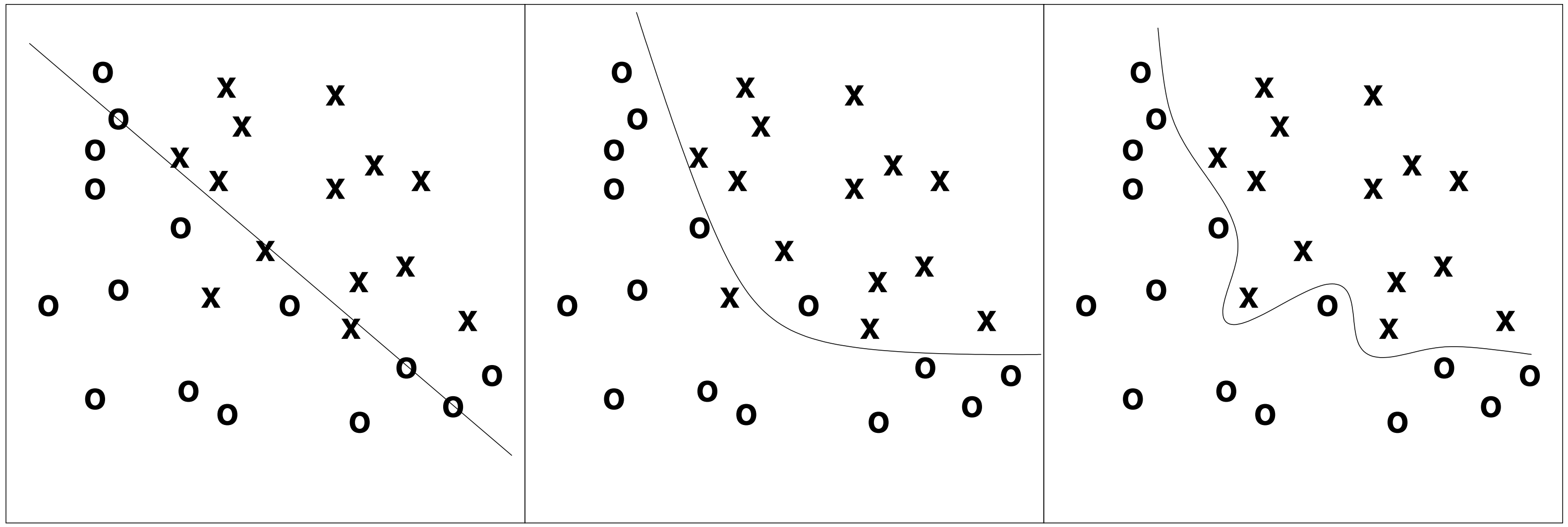
Problem: How can we train it to learn a new function? (credit assignment)

Types of Learning

- Supervised Learning
  - agent is presented with examples of inputs and their target outputs
- Reinforcement Learning
  - agent is not presented with target outputs, but is given a reward signal, which it aims to maximize
- Unsupervised Learning
  - agent is only presented with the inputs themselves, and aims to find structure in these inputs

Ockham's Razor

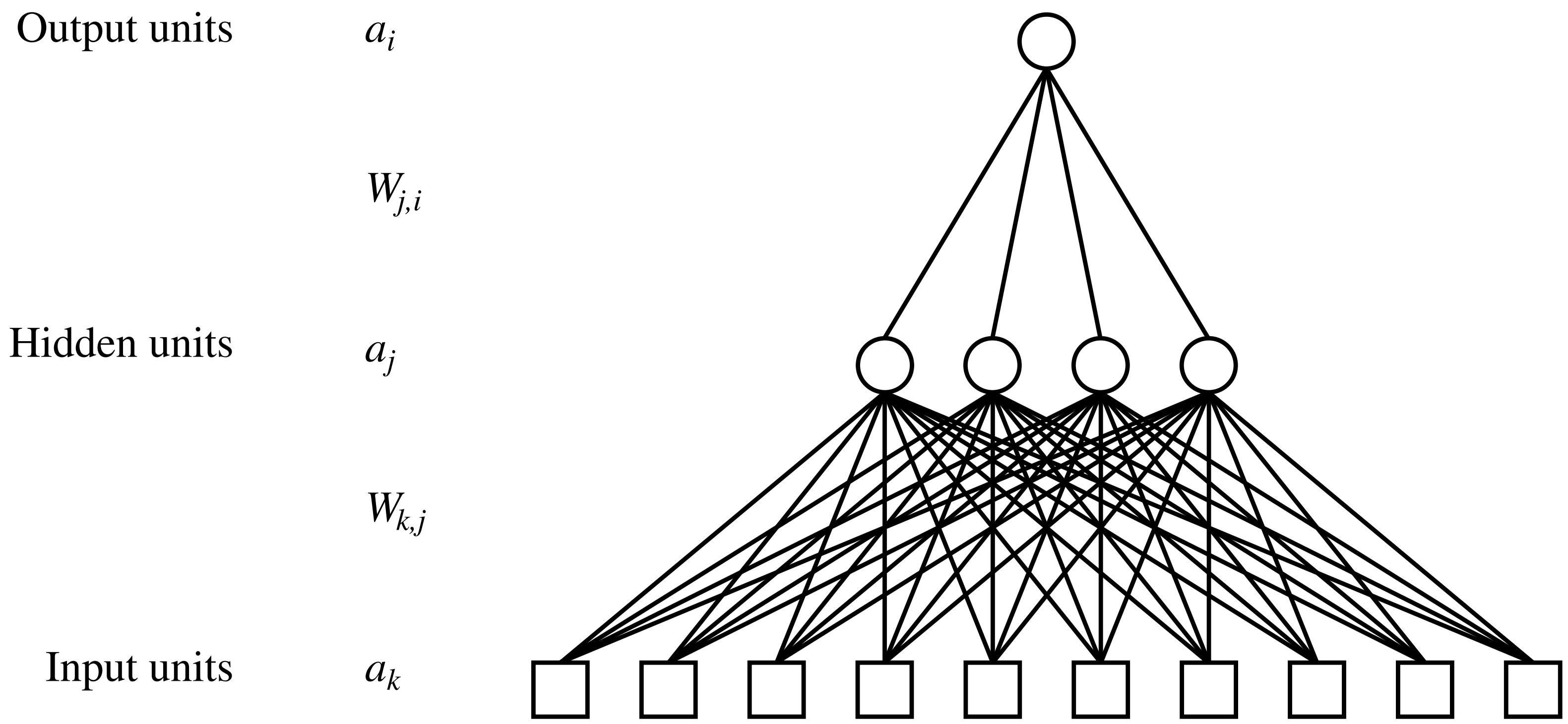
"The most likely hypothesis is the simplest one consistent with the data."



inadequate good compromise over-fitting

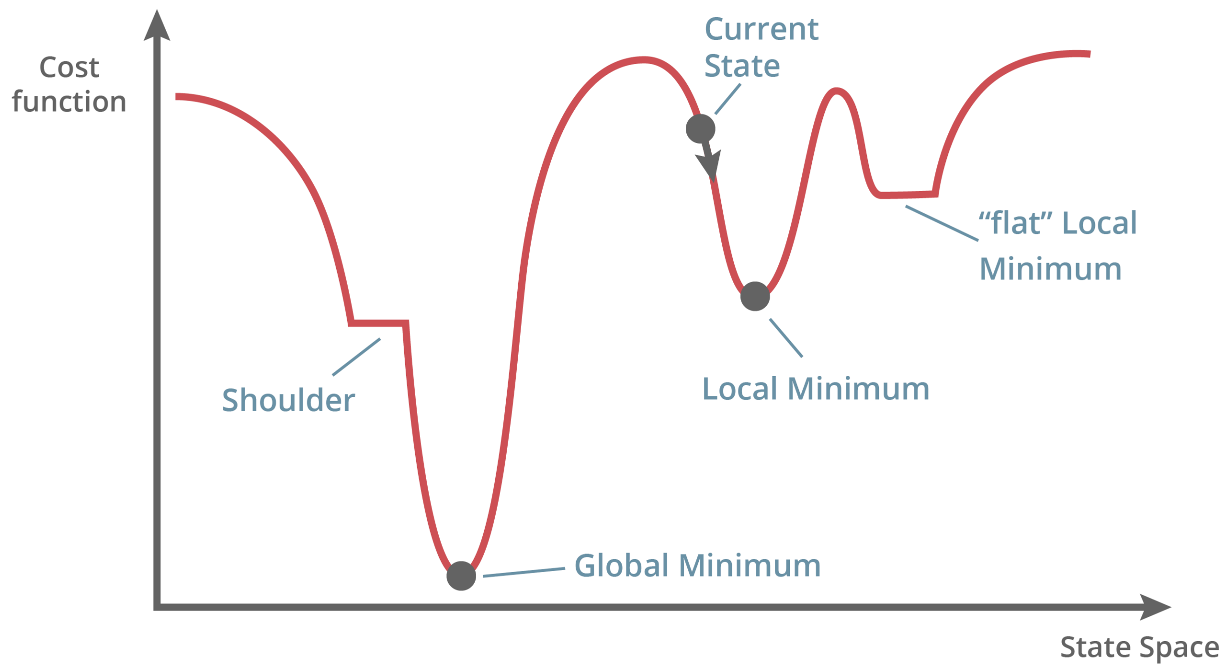
Since there can be noise in the measurements, in practice need to make a tradeoff between simplicity of the hypothesis and how well it fits the data.

Two-Layer Neural Network



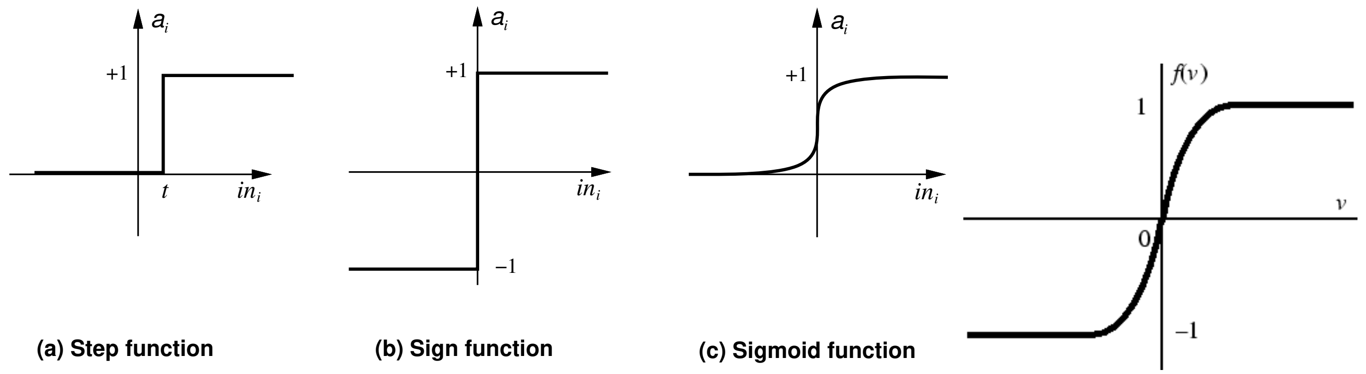
Normally, the numbers of input and output units are fixed, but we can choose the number of hidden units.

Local Search in Weight Space



Problem: because of the step function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and "shoulders", with occasional discontinuous jumps.

Key Idea



Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

$$g(s) = \frac{1}{1+e^{-s}}$$

or hyperbolic tangent

$$g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2\Big(\frac{1}{1+e^{-2s}}\Big) - 1$$

Gradient Descent (4.3)

Recall that the **error function** E is (half) the sum over all input patterns of the square of the difference between actual output and desired output

$$E = \frac{1}{2}\sum(z-t)^2$$

The aim is to find a set of weights for which E is very low.

If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

$$w \leftarrow w - \eta\,\frac{\partial E}{\partial w}$$

Parameter η is called the learning rate.

Variations on Backprop

- Cross Entropy
  - problem: least squares error function unsuitable for classification, where target = 0 or 1
  - mathematical theory: maximum likelihood
  - solution: replace with cross entropy error function
- Weight Decay
  - problem: weights “blow up”, and inhibit further learning
  - mathematical theory: Bayes’ rule
  - solution: add weight decay term to error function
- Momentum
  - problem: weights oscillate in a “rain gutter”
  - solution: weighted average of gradient over time

Cross Entropy

For classification tasks, target t is either 0 or 1, so better to use

$$E = -t\log(z) - (1-t)\log(1-z)$$

This can be justified mathematically, and works well in practice – especially when negative examples vastly outweigh positive ones.
It also makes the backprop computations simpler

$$\begin{aligned}
\frac{\partial E}{\partial z} &= \frac{z-t}{z(1-z)} \\
\text{if} \qquad z &= \frac{1}{1+e^{-s}}, \\
\frac{\partial E}{\partial s} &= \frac{\partial E}{\partial z}\frac{\partial z}{\partial s} = z-t
\end{aligned}$$

Bayes' Rule (3.11)

The formula for conditional probability can be manipulated to find a relationship when the two variables are swapped:

$$P(a \wedge b) = P(a\,|\,b)P(b) = P(b\,|\,a)P(a)$$

$$\rightarrow \text{Bayes' rule } P(a\,|\,b) = \frac{P(b\,|\,a)P(a)}{P(b)}$$

This is often useful for assessing the probability of an underlying cause after an effect has been observed:

$$P(\text{Cause}|\text{Effect}) = \frac{P(\text{Effect}|\text{Cause})P(\text{Cause})}{P(\text{Effect})}$$

Bayesian Inference

H is a class of hypotheses

$P(D\,|\,h)$ = probability of data D being generated under hypothesis $h \in H$.

$P(h\,|\,D)$ = probability that h is correct, given that data D were observed.

Bayes' Theorem:

$$\begin{aligned} P(h\,|\,D)P(D) &= P(D\,|\,h)P(h) \\ P(h\,|\,D) &= \frac{P(D\,|\,h)P(h)}{P(D)} \end{aligned}$$

$P(h)$ is called the prior.

Weight Decay (5.2.2)

Sometimes we add a penalty term to the loss function which encourages the neural network weights w_j to remain small:

$$E = \frac{1}{2}\sum_i (z_i - t_i)^2 + \frac{\lambda}{2}\sum_j w_j^2$$

This can prevent the weights from "saturating" to very high values.

It is sometimes referred to as "elastic weights" because the weights experience a force as if there were a spring pulling them back towards the origin according to Hooke's Law.

The scaling factor λ needs to be determined from experience, or empirically.

Momentum (8.3)

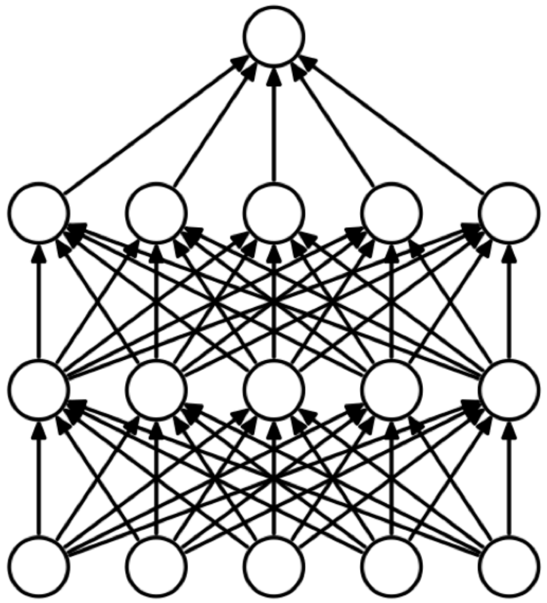
If landscape is shaped like a “rain gutter”, weights will tend to oscillate without much improvement.

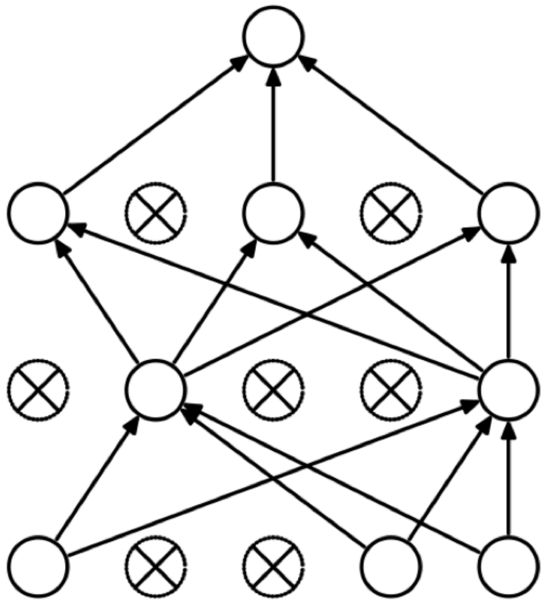
Solution: add a momentum factor

$$
\begin{aligned}
\delta w &\leftarrow \alpha\,\delta w - \eta\,\frac{\partial E}{\partial w}\\
w &\leftarrow w + \delta w
\end{aligned}
$$

Hopefully, this will dampen sideways oscillations but amplify downhill motion by $\frac{1}{1-\alpha}$.

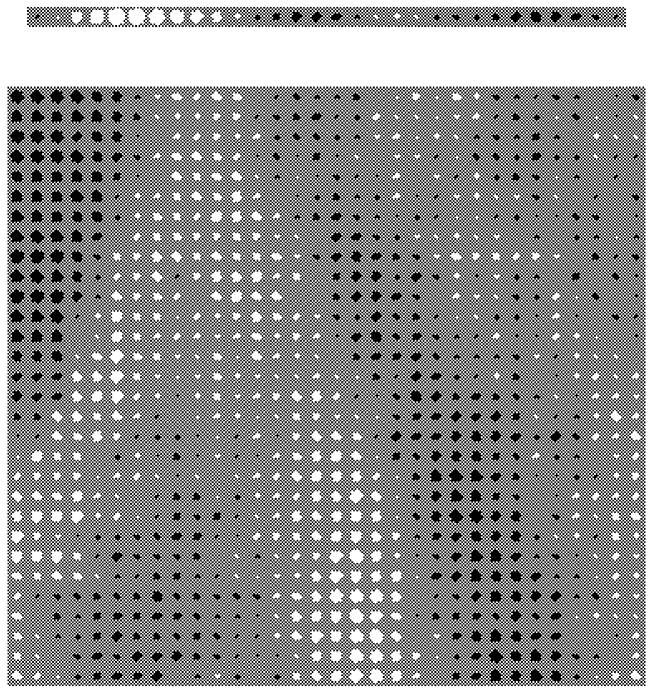
Dropout (7.12)

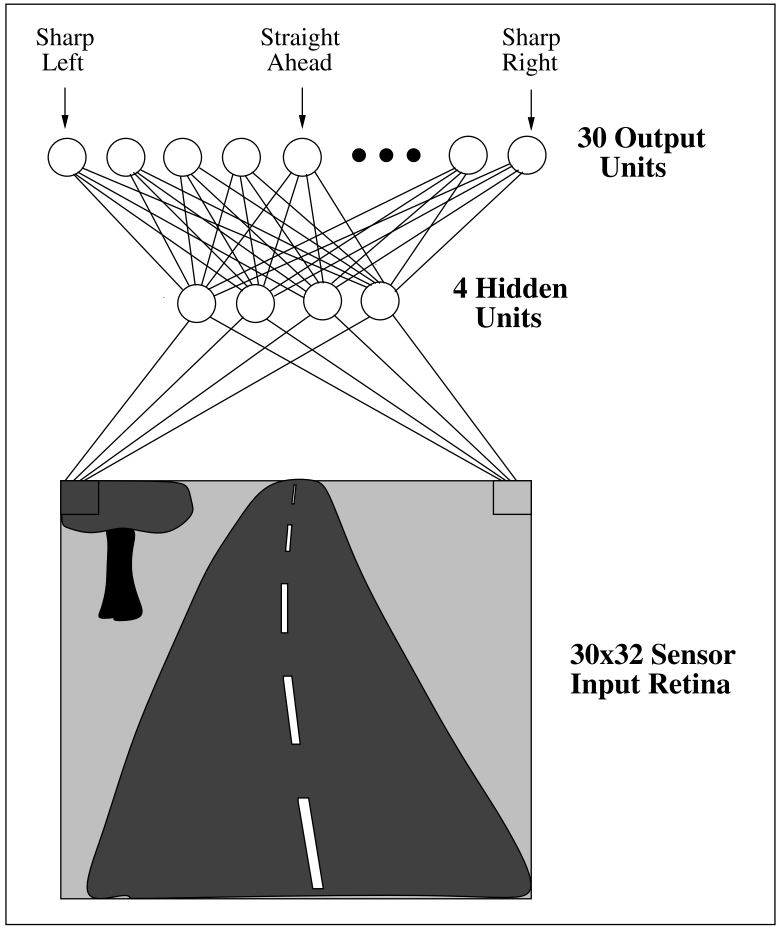

(a) Standard Neural Net


(b) After applying dropout.

Nodes are randomly chosen to not be used, with some fixed probability (usually, one half).

Hinton Diagrams

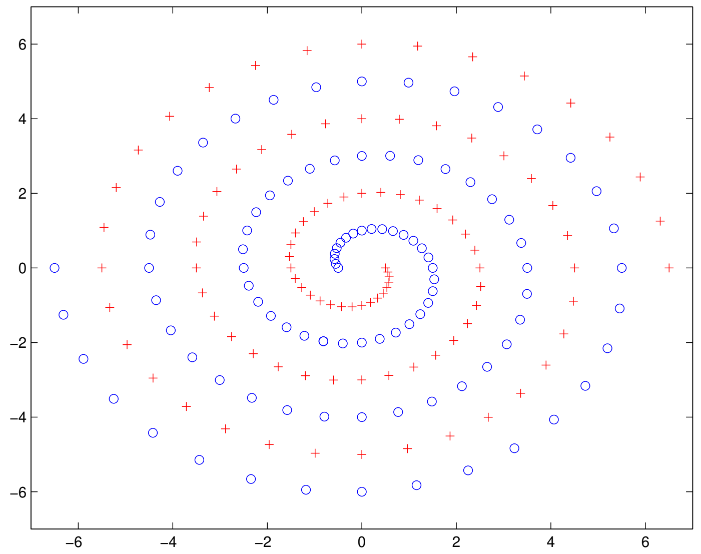


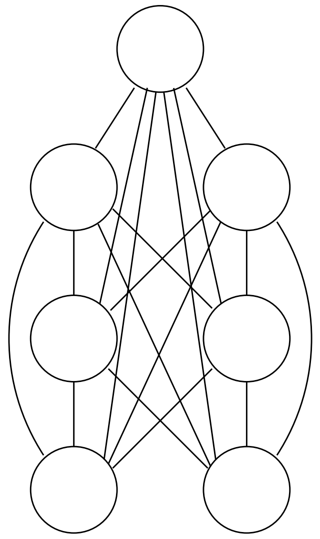


- used to visualize higher dimensions
- white = positive, black = negative

Limitations of Two-Layer Neural Networks

Some functions cannot be learned with a 2-layer sigmoidal network.





For example, this Twin Spirals problem cannot be learned with a 2-layer network, but it can be learned using a 3-layer network if we include shortcut connections between non-consecutive layers.

Vanishing / Exploding Gradients

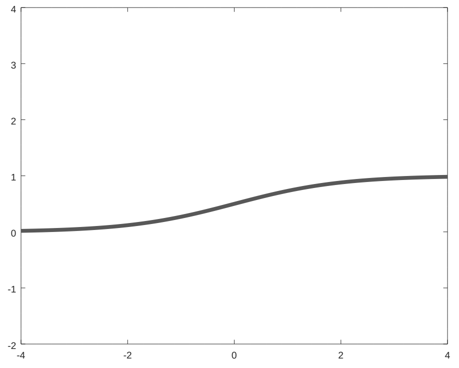
Training by backpropagation in networks with many layers is difficult.

When the weights are small, the differentials become smaller and smaller as we backpropagate through the layers, and end up having no effect.

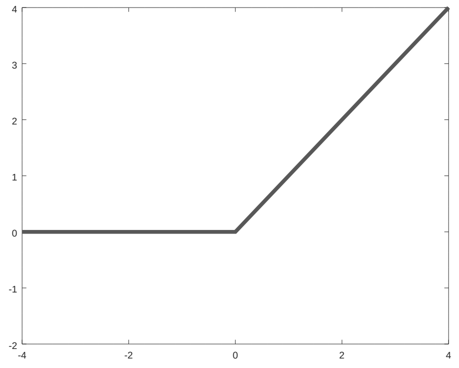
When the weights are large, the activations in the higher layers will saturate to extreme values. As a result, the gradients at those layers will become very small, and will not be propagated to the earlier layers.

When the weights have intermediate values, the differentials will sometimes get multiplied many times is places where the transfer function is steep, causing them to blow up to large values.

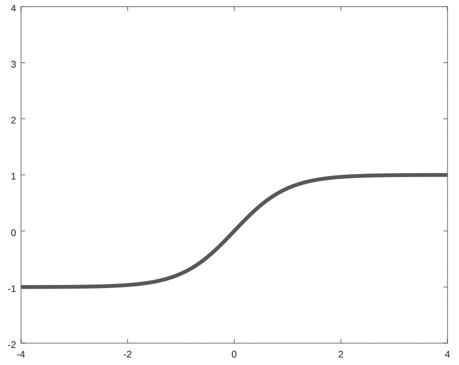
Activation Functions (6.3)



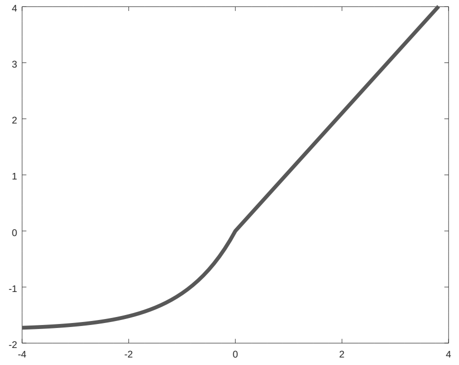
Sigmoid



Rectified Linear Unit (ReLU)

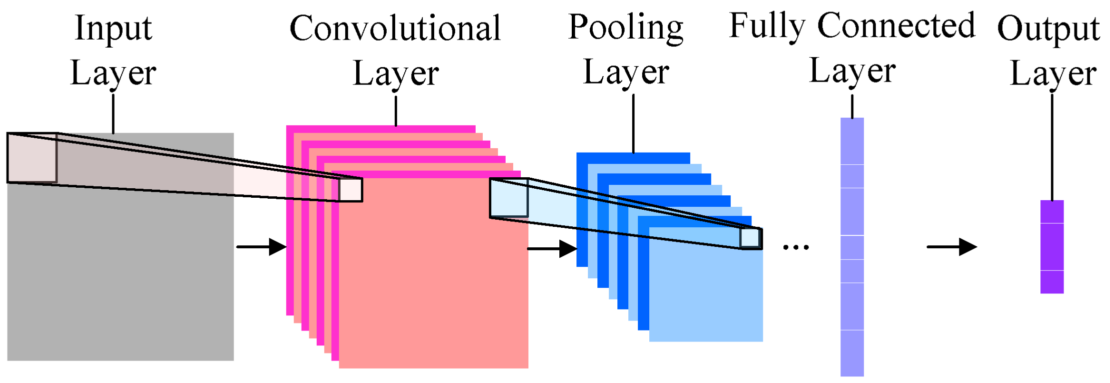


Hyperbolic Tangent



Scaled Exponential Linear Unit (SELU)

Convolutional Network Components



- **convolution layers**: extract shift-invariant features from the previous layer
- **subsampling or pooling layers**: combine the activations of multiple units from the previous layer into one unit
- **fully connected layers**: collect spatially diffuse information
- **output layer**: choose between classes

Softmax (6.2.2)

Consider a classification task with N classes, and assume z_j is the output of the unit corresponding to class j.

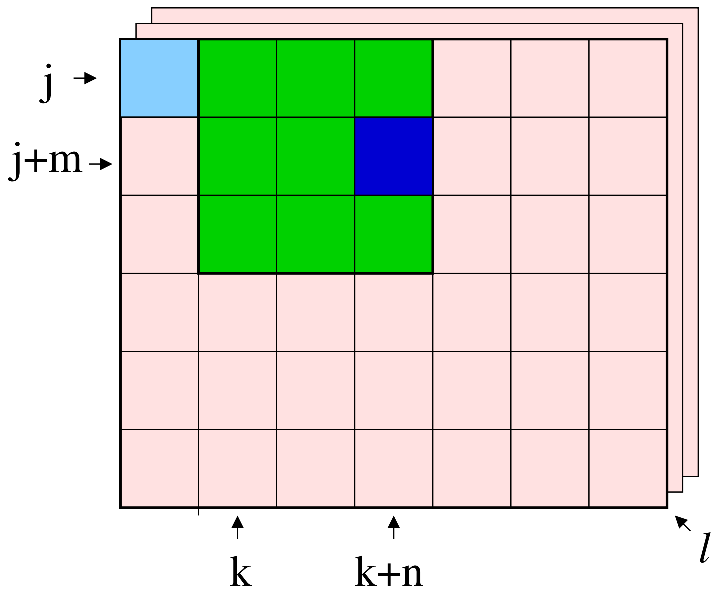
We assume the network's estimate of the probability of each class j is proportional to $\exp(z_j)$. Because the probabilites must add up to 1, we need to normalize by dividing by their sum:

$$\mathrm{Prob}(i) = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$$

$$\log \mathrm{Prob}(i) = z_i - \log \sum_{j=1}^{N} \exp(z_j)$$

If the correct class is i, we can treat $-\log \mathrm{Prob}(i)$ as our cost function. The first term pushes up the correct class i, while the second term mainly pushes down the incorrect class j with the highest activation (if $j \neq i$).

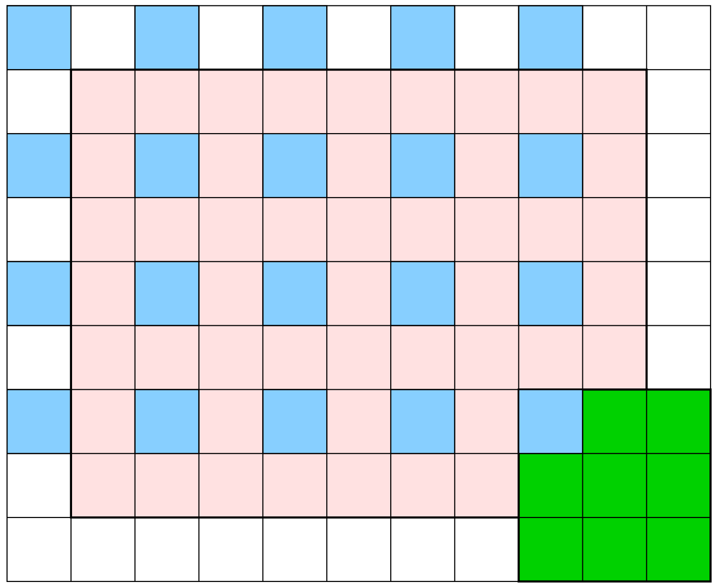
Convolutional Neural Networks



$$Z^i_{j,k} = g\Big(b^i + \sum_l \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} K^i_{l,m,n} V^l_{j+m,k+n}\Big)$$

The same weights are applied to the next $M \times N$ block of inputs, to compute the next hidden unit in the convolution layer ("weight sharing").

Stride with Zero Padding



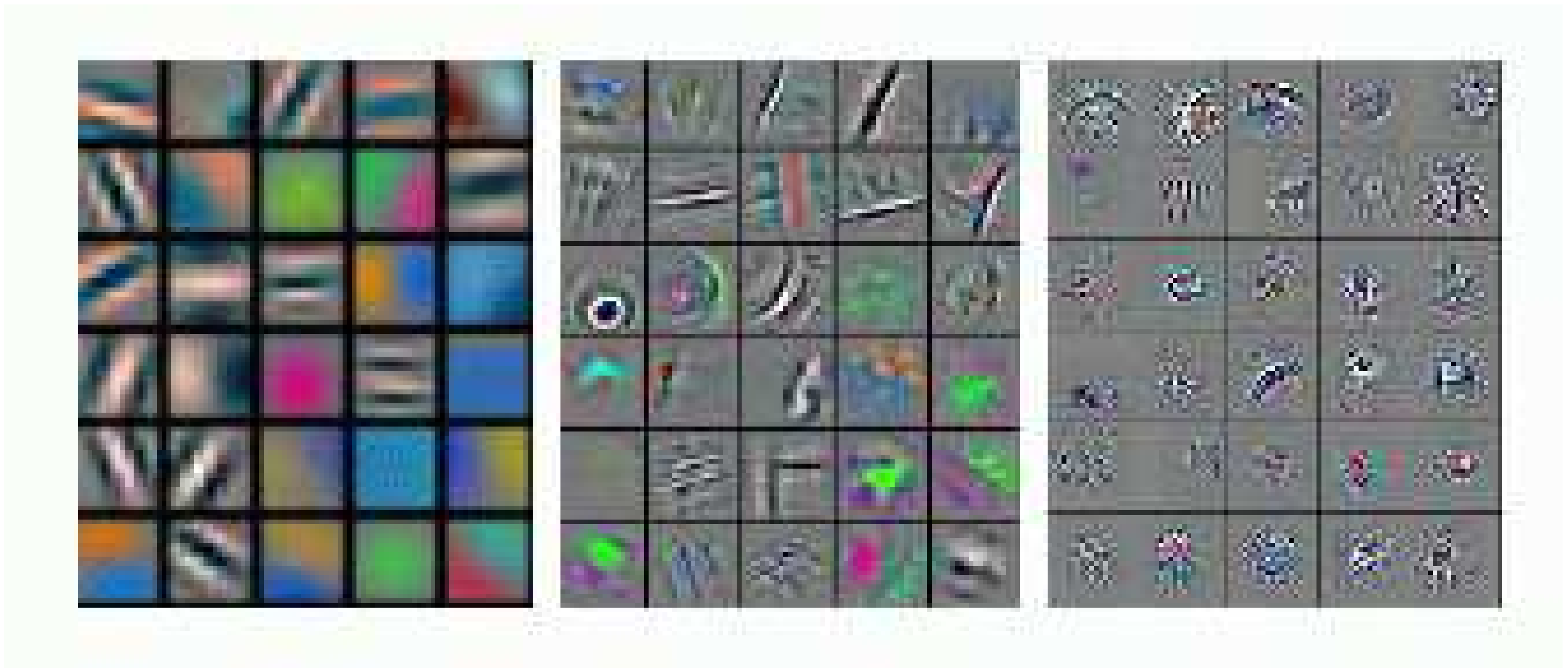
When combined with zero padding of width P,

j takes on the values $0, s, 2s, \ldots, (J+2P-M)$

k takes on the values $0, s, 2s, \ldots, (K+2P-N)$

The next layer is $(1+(J+2P-M)/s)$ by $(1+(K+2P-N)/s)$

Convolutional Filters



First Layer Second Layer Third Layer

Weight Initialization

In order to have healthy forward and backward propagation, each term in the product must be approximately equal to 1. Any deviation from this could cause the activations to either vanish or saturate, and the differentials to either decay or explode exponentially.

$$\text{Var}[z] \simeq \left(\prod_{i=1}^{D} G_0\, n_i^{\text{in}} \text{Var}[w^{(i)}]\right) \text{Var}[x]$$

$$\text{Var}[\frac{\partial}{\partial x}] \simeq \left(\prod_{i=1}^{D} G_1\, n_i^{\text{out}} \text{Var}[w^{(i)}]\right) \text{Var}[\frac{\partial}{\partial z}]$$

We therefore choose the initial weights $\{w_{jk}^{(i)}\}$ in each layer (i) such that

$$G_1 n_i^{\text{out}} \text{Var}[w^{(i)}] = 1$$

Batch Normalization

We can normalize the activations $x_k^{(i)}$ of node k in layer (i) relative to the mean and variance of those activations, calculated over a mini-batch of training items:

$$\hat{x}_k^{(i)} = \frac{x_k^{(i)} - \mathrm{Mean}[x_k^{(i)}]}{\sqrt{\mathrm{Var}[x_k^{(i)}]}}$$

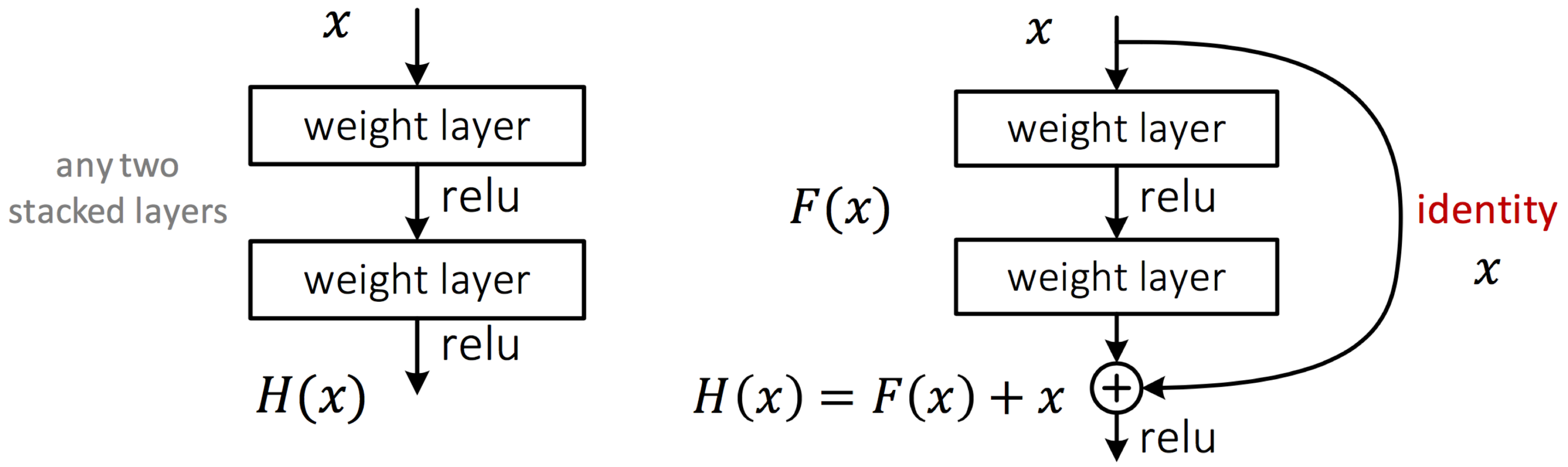
These activations can then be shifted and re-scaled to

$$y_k^{(i)} = \beta_k^{(i)} + \gamma_k^{(i)} \hat{x}_k^{(i)}$$

$\beta_k^{(i)}, \gamma_k^{(i)}$ are additional parameters, for each node, which are trained by backpropagation along with the other parameters (weights) in the network.

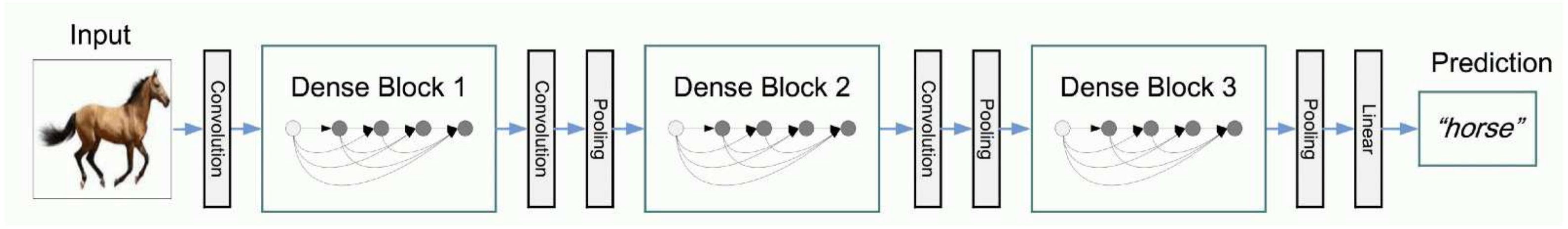
After training is complete, $\mathrm{Mean}[x_k^{(i)}]$ and $\mathrm{Var}[x_k^{(i)}]$ are either pre-computed on the entire training set, or updated using running averages.

Residual Networks



Idea: Take any two consecutive stacked layers in a deep network and add a "skip" connection which bipasses these layers and is added to their output.

Dense Networks



Recently, good results have been achieved using networks with densely connected blocks, within which each layer is connected by shortcut connections to all the preceding layers.

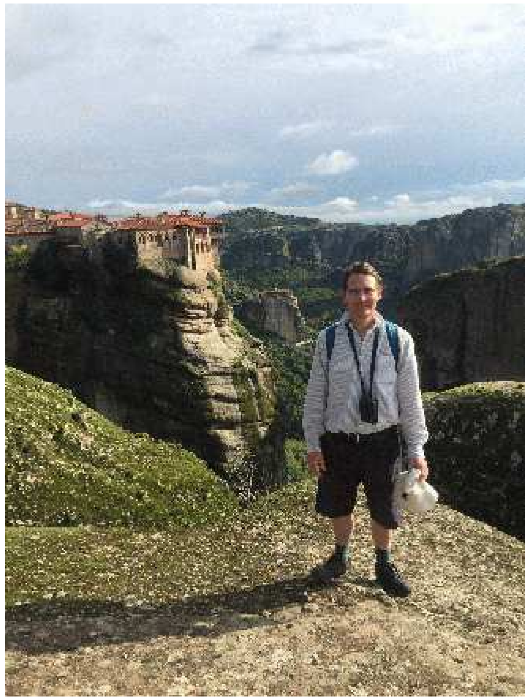
Neural Texture Synthesis

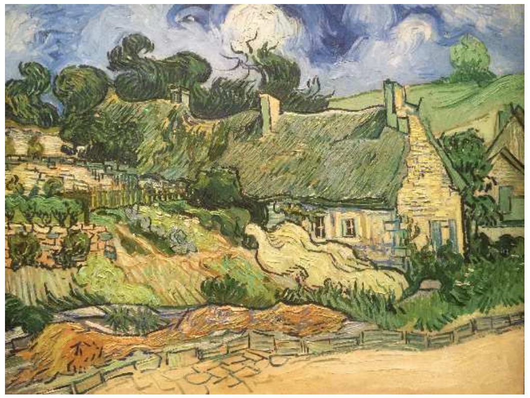
1. pretrain CNN on ImageNet (VGG-19)
2. pass input texture through CNN; compute feature map F^l_{ik} for i^{th} filter at spatial location k in layer (depth) l
3. compute the Gram matrix for each pair of features

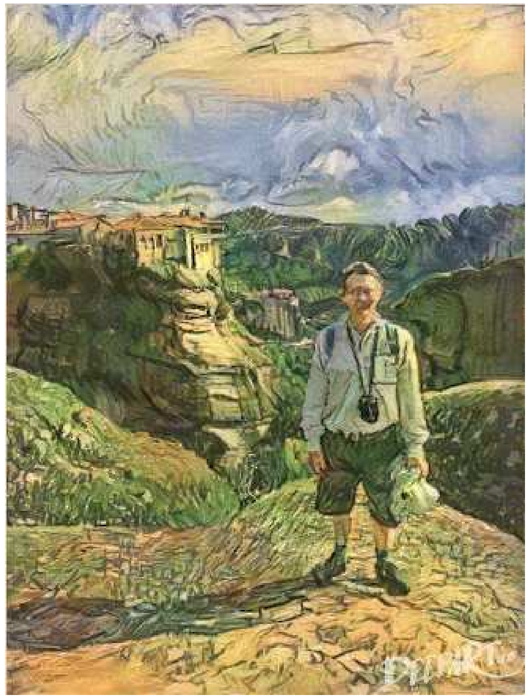
$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}$$

4. feed (initially random) image into CNN
5. compute L2 distance between Gram matrices of original and new image
6. backprop to get gradient on image pixels
7. update image and go to step 5.

Neural Style Transfer







content + style → new image

Neural Style Transfer

For Neural Style Transfer, we minimize a cost function which is

$$\begin{aligned} E_{\text{total}} &= \alpha\ E_{\text{content}} \quad + \quad \beta\, E_{\text{style}} \\ &= \frac{\alpha}{2}\sum_{i,k} ||F^l_{ik}(x) - F^l_{ik}(x_c)||^2 + \frac{\beta}{4}\sum_{l=0}^{L}\frac{w_l}{N_l^2 M_l^2}\sum_{i,j}(G^l_{ij} - A^l_{ij})^2 \end{aligned}$$

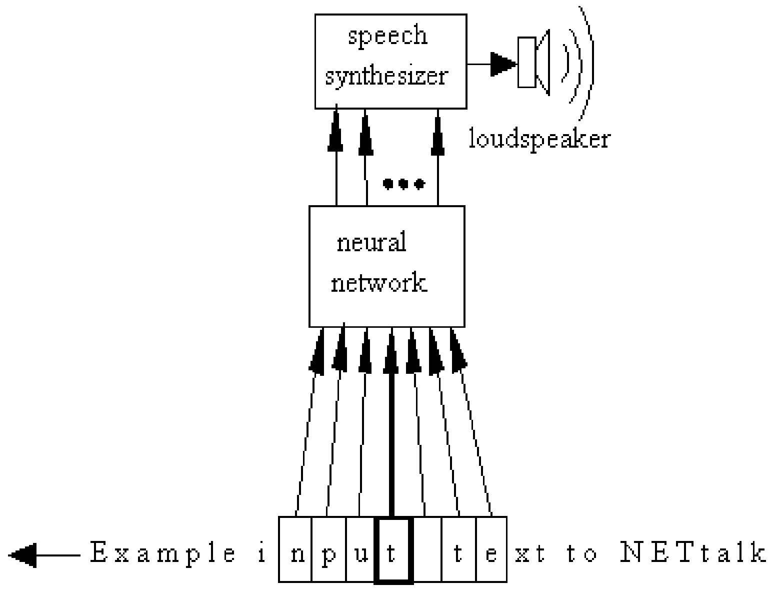
where

x_c, x	$=$	content image, synthetic image
F^l_{ik}	$=$	i^{th} filter at position k in layer l
$N_l,\ M_l$	$=$	number of filters, and size of feature maps, in layer l
w_l	$=$	weighting factor for layer l
$G^l_{ij},\ A^l_{ij}$	$=$	Gram matrices for style image, and synthetic image

Recurrent Networks

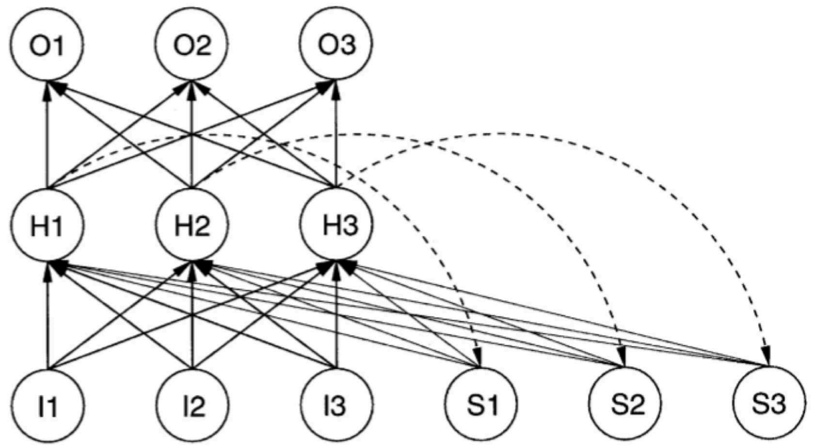
- Processing Temporal Sequences
- Sliding Window
- Recurrent Network Architectures
- Hidden Unit Dynamics
- Long Short Term Memory

Sliding Window



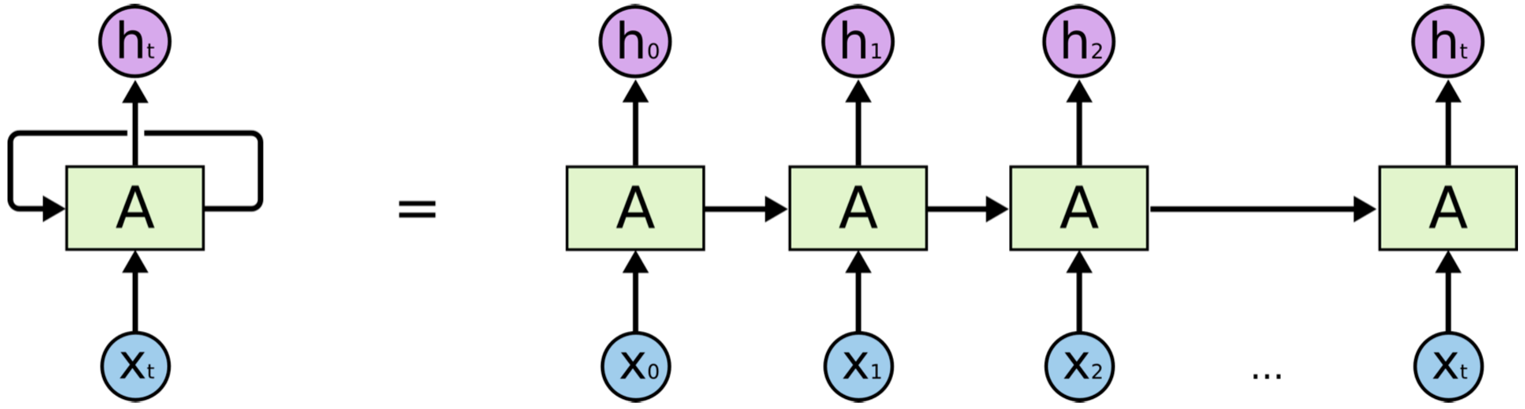
The simplest way to feed temporal input to a neural network is the "sliding window" approach, first used in the NetTalk system (Sejnowski & Rosenberg, 1987).

Simple Recurrent Network (Elman, 1990)



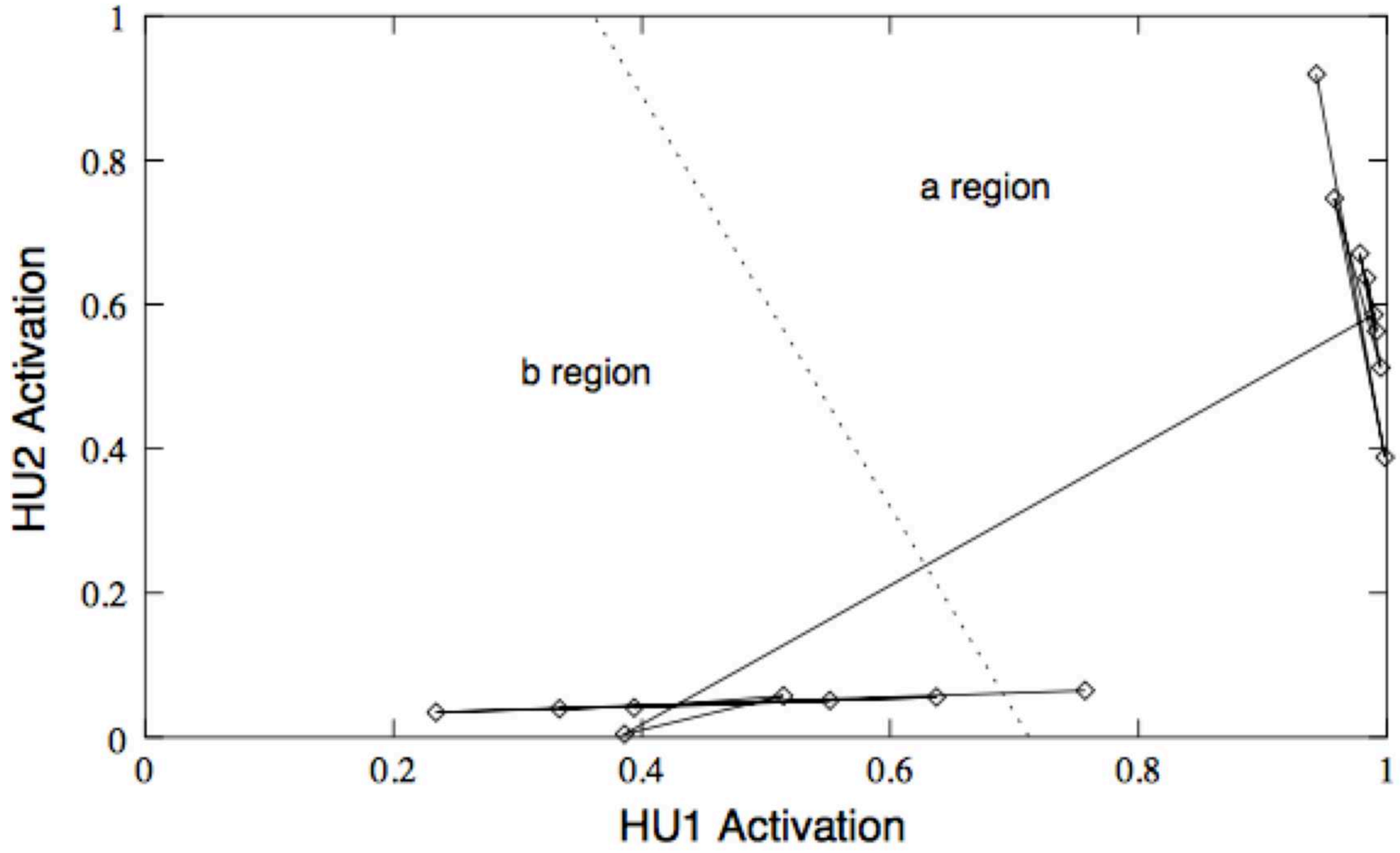
- at each time step, hidden layer activations are copied to "context" layer
- hidden layer receives connections from input and context layers
- the inputs are fed one at a time to the network, it uses the context layer to "remember" whatever information is required for it to produce the correct output

Back Propagation Through Time

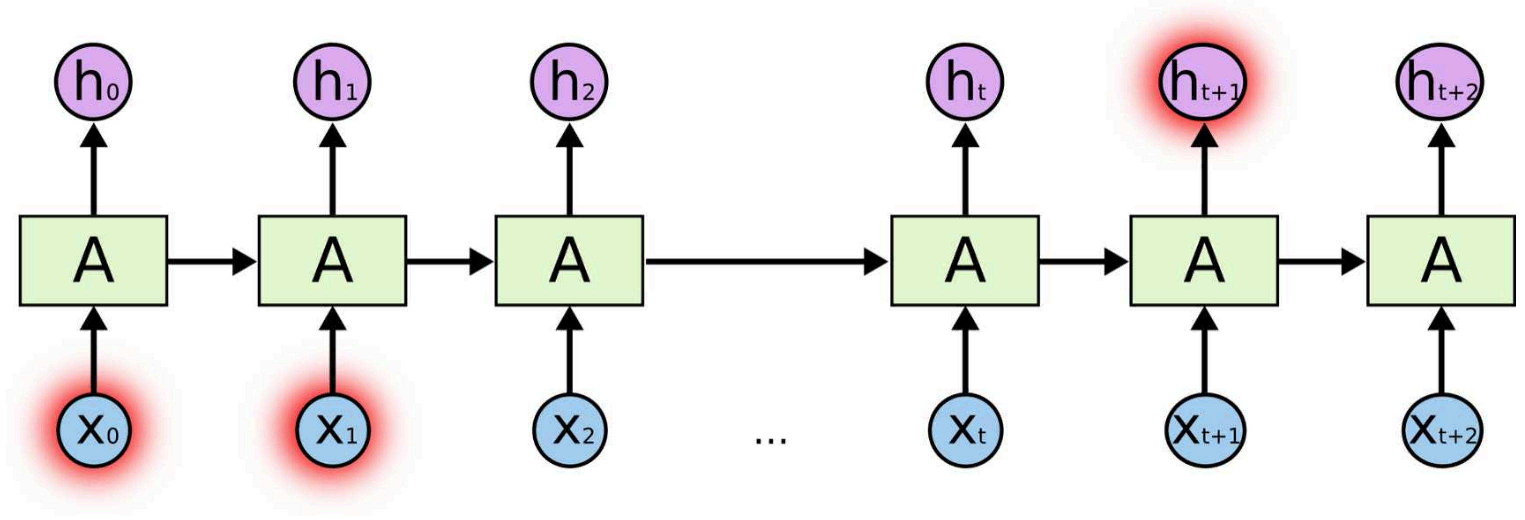


- we can "unroll" a recurrent architecture into an equivalent feedforward architecture, with shared weights
- applying backpropagation to the unrolled architecture is reffered to as "backpropagation through time"
- we can backpropagate just one timestep, or a fixed number of timesteps, or all the way back to beginning of the sequence

Oscillating Solution for $a^n b^n$

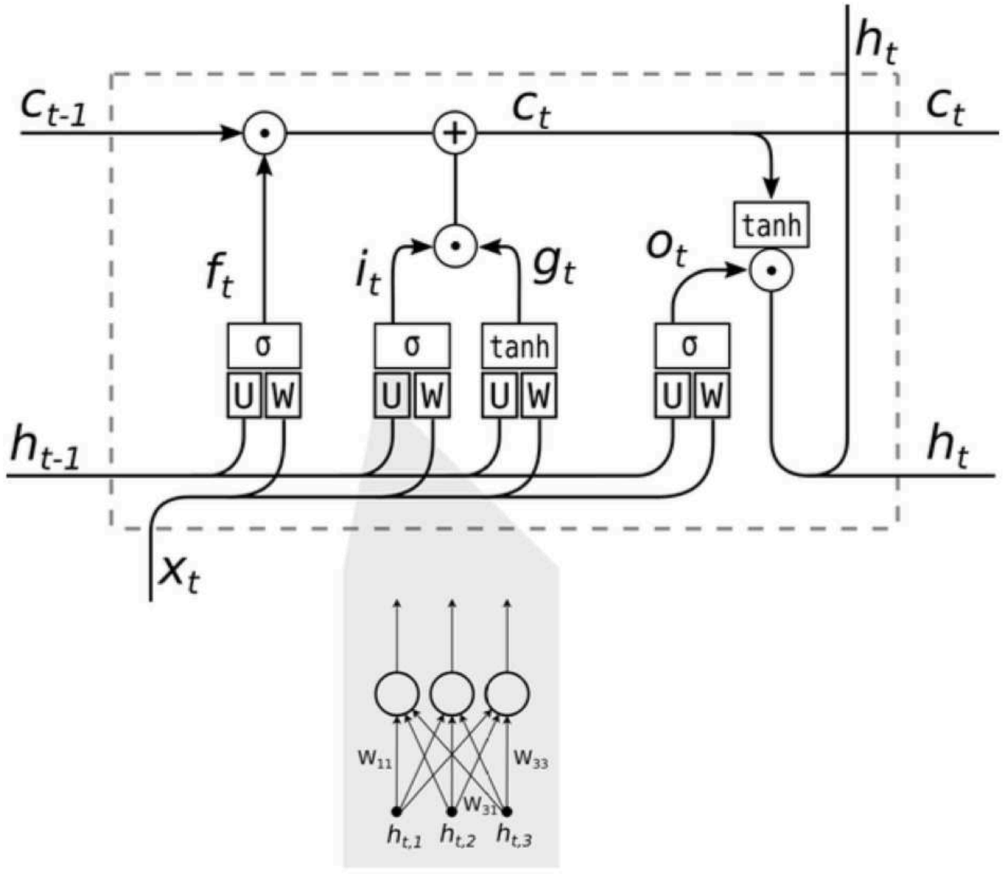


Long Range Dependencies



- Simple Recurrent Networks (SRNs) can learn medium-range dependencies but have difficulty learning long range dependencies
- Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) can learn long range dependencies better than SRN

Long Short Term Memory



Gates:

$$\mathbf{f}_t = \sigma\left(W_f\mathbf{x}_t + U_f\mathbf{h}_{t-1} + \mathbf{b}_f\right)$$
$$\mathbf{i}_t = \sigma\left(W_i\mathbf{x}_t + U_i\mathbf{h}_{t-1} + \mathbf{b}_i\right)$$
$$\mathbf{g}_t = \tanh\left(W_g\mathbf{x}_t + U_g\mathbf{h}_{t-1} + \mathbf{b}_g\right)$$
$$\mathbf{o}_t = \sigma\left(W_o\mathbf{x}_t + U_o\mathbf{h}_{t-1} + \mathbf{b}_o\right)$$

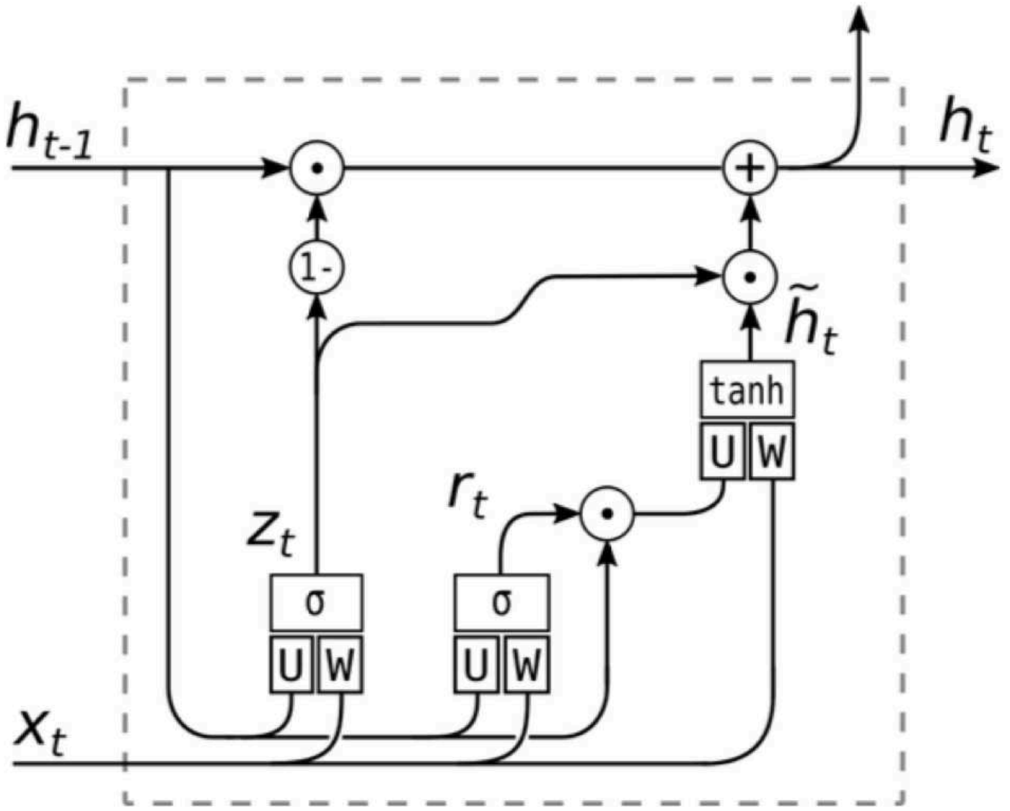
State:

$$\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t + \mathbf{i}_t \odot \mathbf{g}_t$$

Output:

$$\mathbf{h}_t = \tanh \mathbf{c}_t \odot \mathbf{o}_t$$

Gated Recurrent Unit



Gates:

$\mathbf{z}_t = \sigma(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1} + \mathbf{b}_z)$

$\mathbf{r}_t = \sigma(W_r\mathbf{x}_t + U_r\mathbf{h}_{t-1} + \mathbf{b}_r)$

Candidate Activation:

$\tilde{\mathbf{h}}_t =$

$\tanh(W\mathbf{x}_t + U(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h)$

Output:

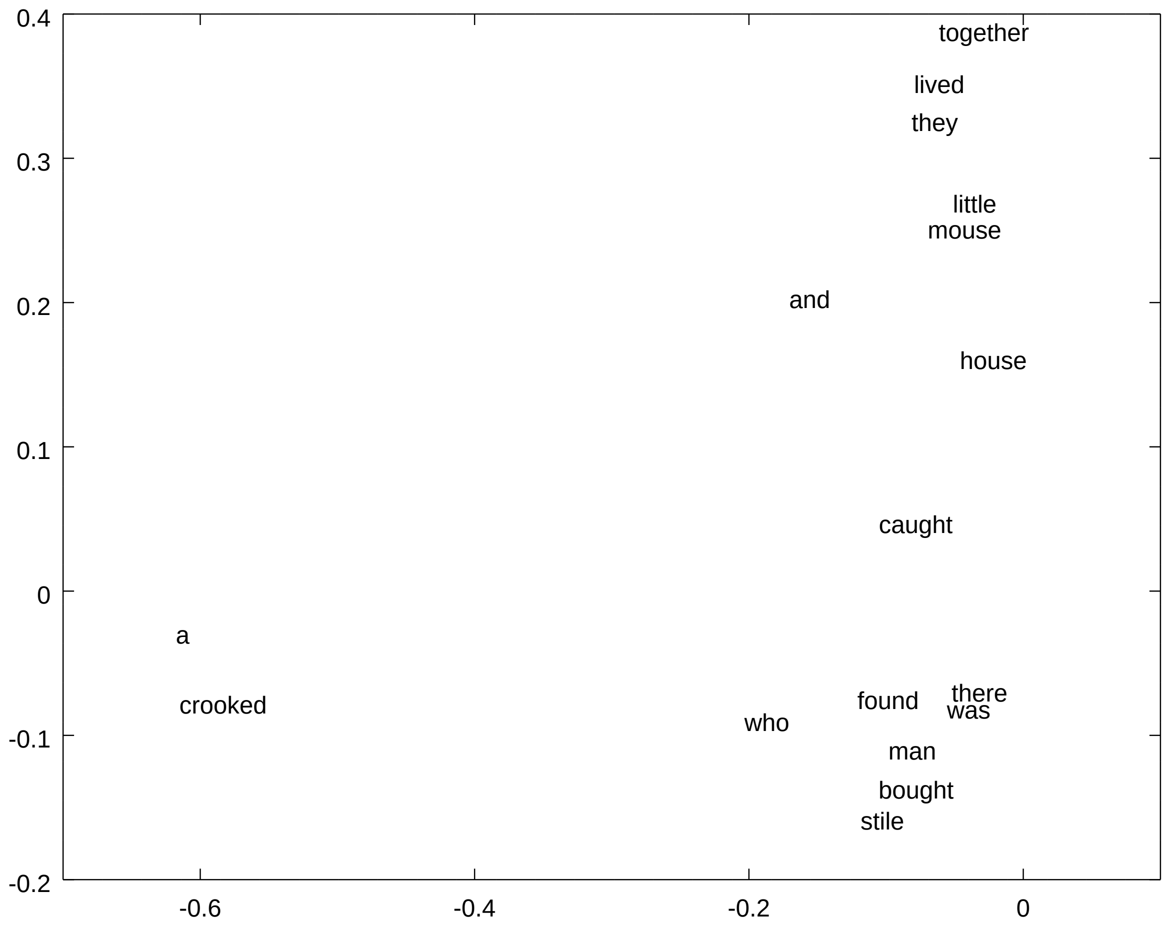
$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$

GRU is similar to LSTM but has only two gates instead of three.

Co-occurrence Matrix (2-word window)

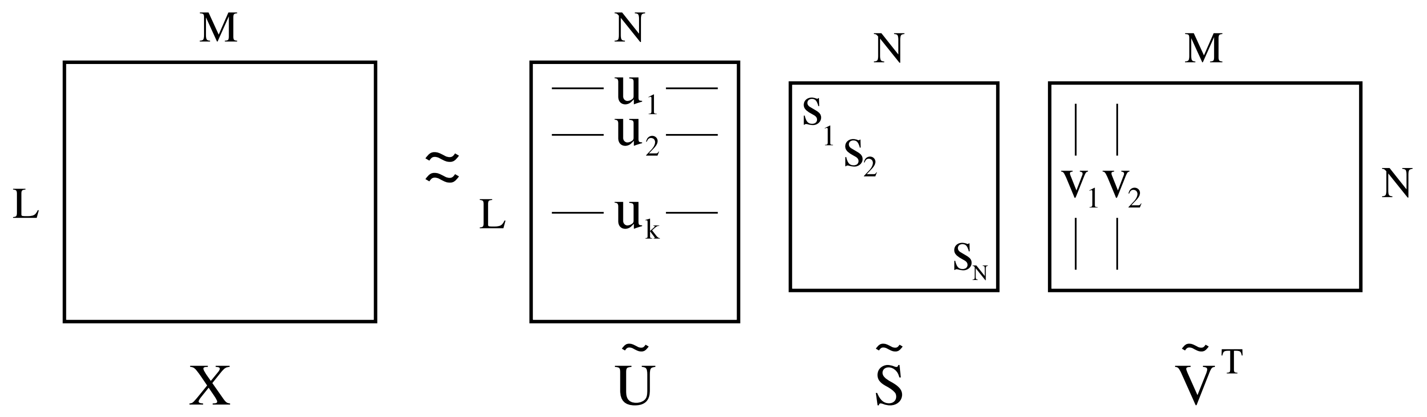
word	a	all	and	bought	cat	caught	crooked	found	he	house	in	little	lived	man	mile	mouse	sixpence	stile	there	they	together	upon	walked	was	who
a				1		1	6	1			1	1										1	1	1	
all													1							1					
and								1							1	1				1					
bought	1								1																
cat							1																		1
caught	1																								1
crooked	6				1					1		1		1	1	1	1	1							
found	1		1																						
he				1														1							
house							1																		
in	1																				1				
little	1						1																		
lived		1																			1				
man							1																		1
mile			1				1																		
mouse			1				1																		
sixpence							1															1			
stile							1		1																
there																								1	
they		1	1																						
together											1		1												
upon	1																1								
walked	1																								1
was	1																		1						
who					1	1								1									1		

Word Embeddings



Singular Value Decomposition

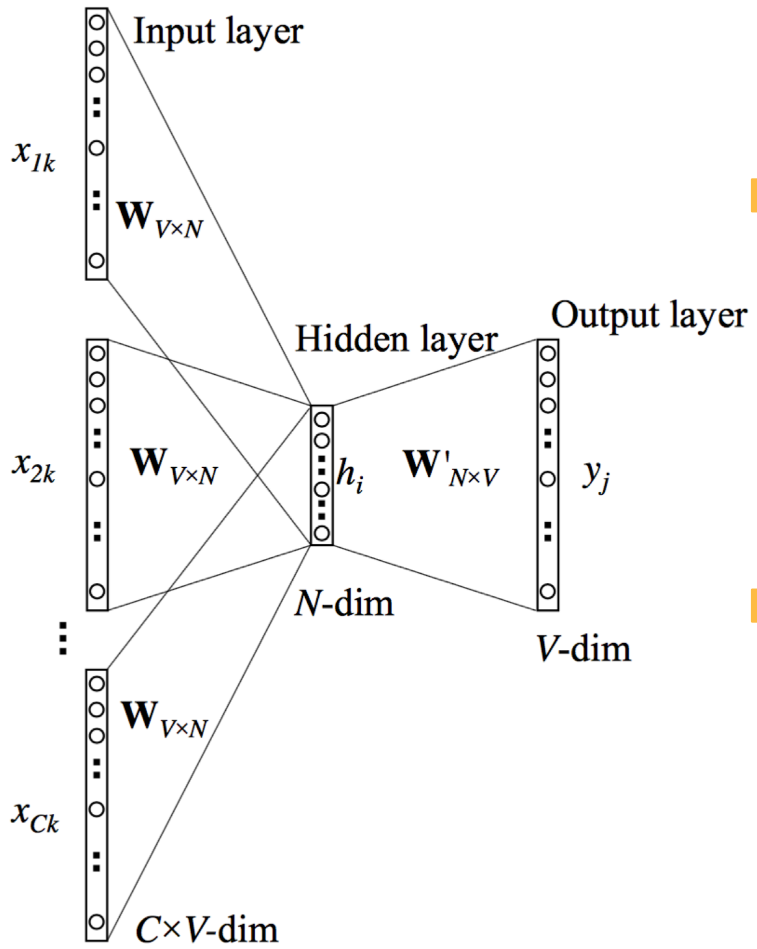
Co-occurrence matrix X can be decomposed as $X = USV^T$ where U, V are unitary (all columns have unit length) and S is diagonal.



Columns 1 to n of row k of U then provide an n-dimensional vector representing the kth word in the vocabulary.

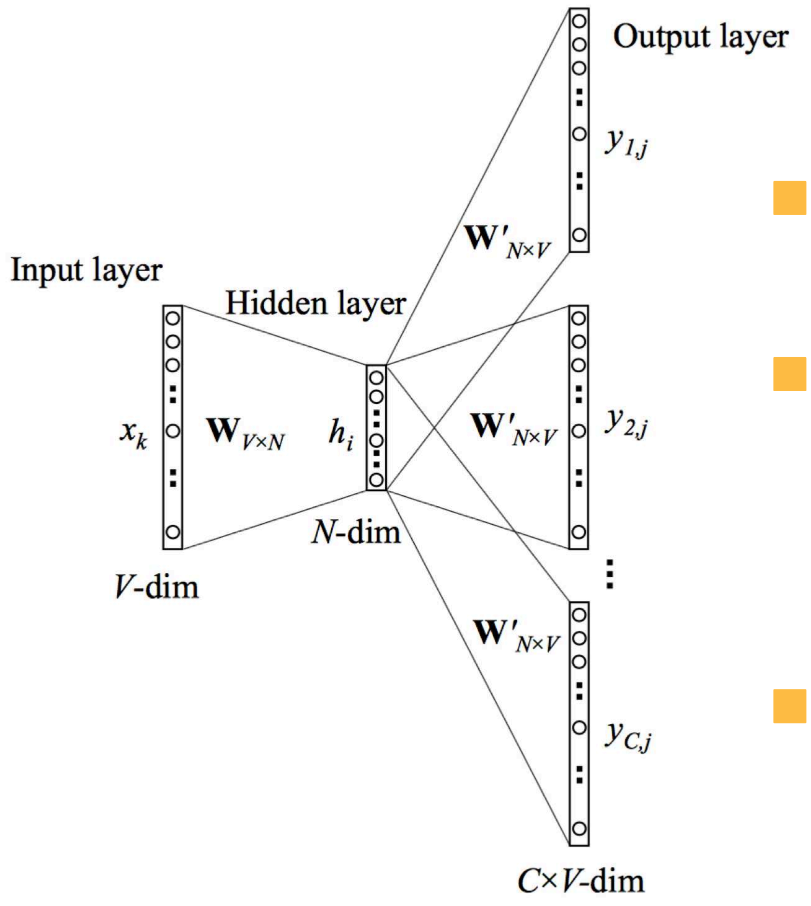
SVD is computationally expensive, proportional to $L \times M^2$ if $L \geq M$. Can we do something similar with less computation, and incrementally?

Continuous Bag Of Words



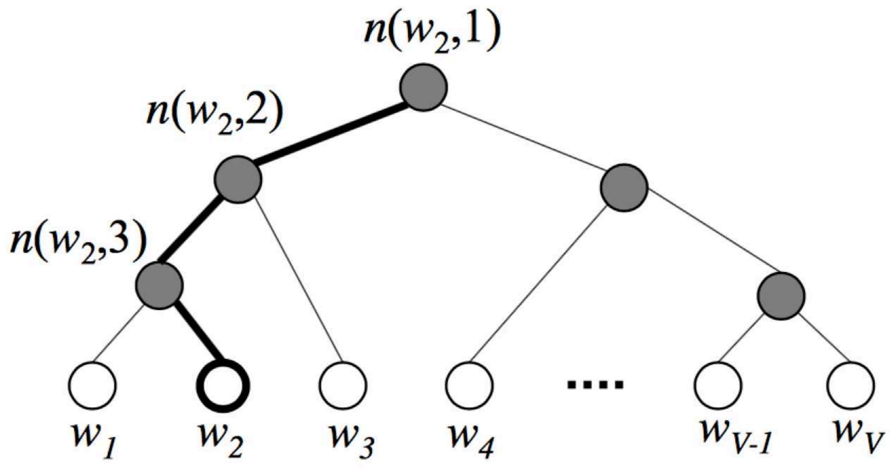
- If several context words are each used independently to predict the center word, the hidden activation becomes a sum (or average) over all the context words
- Note the difference between this and NetTalk – in word2vec (CBOW) all context words share the same input-to-hidden weights

word2vec Skip-Gram Model



- try to predict the context words, given the center word
- this skip-gram model is similar to CBOW, except that in this case a single input word is used to predict multiple context words
- all context words share the same hidden-to-output weights

Hierarchical Softmax



$$[\![n' = \text{child}(n)]\!] = \begin{cases} +1, & \text{if } n' \text{ is left child of node } n, \\ -1, & \text{otherwise.} \end{cases}$$

$$\sigma(u) = 1/(1 - \exp(-u))$$

$$\text{prob}(w = w_t) = \prod_{j=1}^{L(w)-1} \sigma([\![n(w, j+1) = \text{child}(n(w, j))]\!] \mathbf{v}'_{n(w,j)}{}^{\mathrm{T}} \mathbf{h})$$

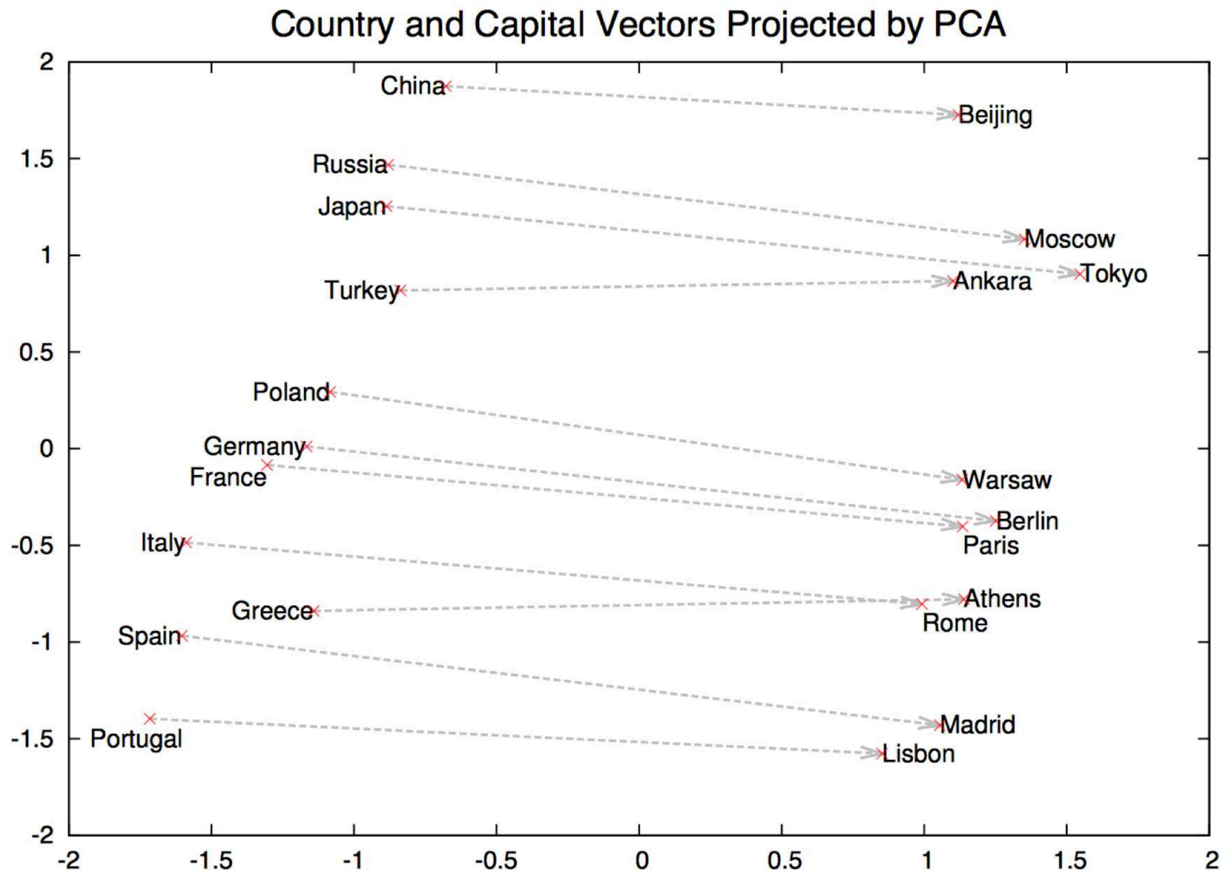
Negative Sampling

The idea of negative sampling is that we train the network to increase its estimation of the target word j^* and reduce its estimate not of all the words in the vocabulary but just a subset of them $\mathcal{W}_{\text{neg}}$, drawn from an appropriate distribution.

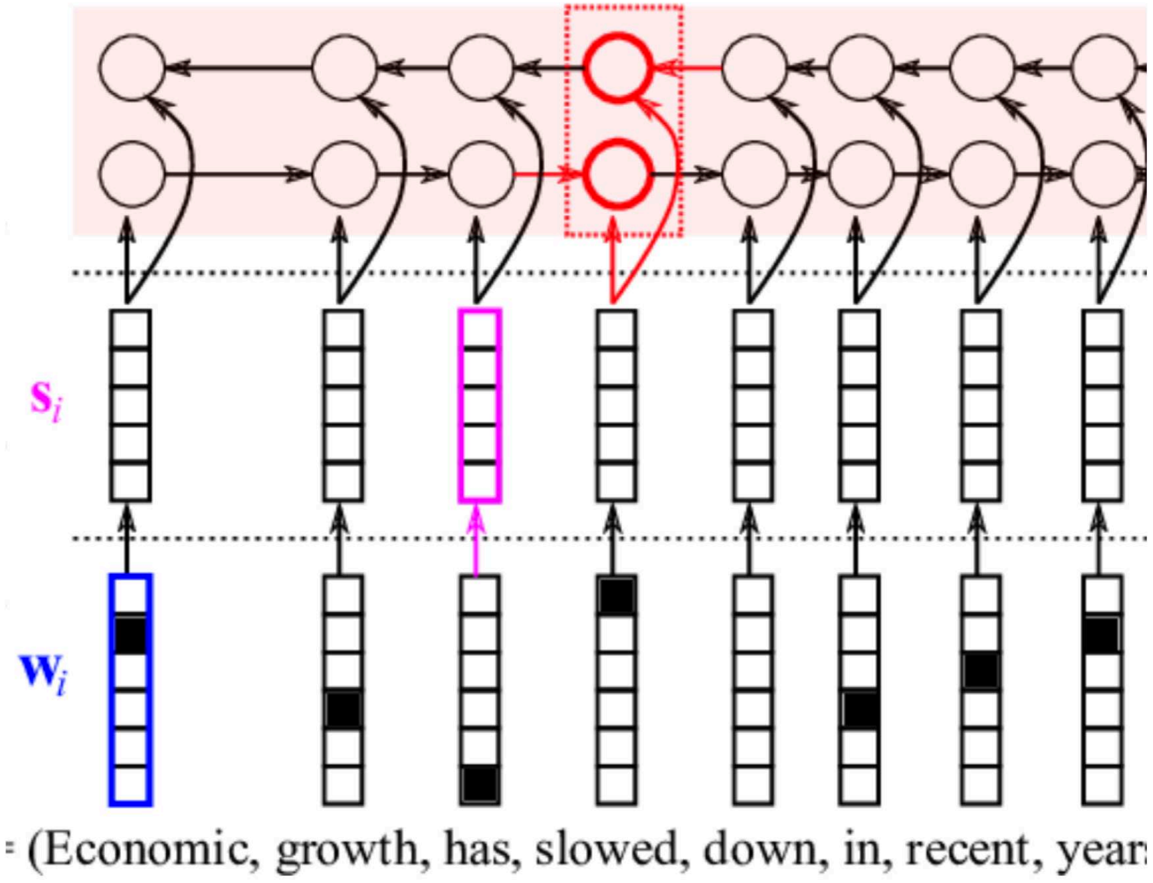
$$E = -\log \sigma(\mathbf{v}'_{j^*}{}^{\mathrm{T}}\mathbf{h}) - \sum_{j \in \mathcal{W}_{\text{neg}}} \log \sigma(-\mathbf{v}'_j{}^{\mathrm{T}}\mathbf{h})$$

This is a simplified version of Noise Constrastive Estimation (NCE).
It is not guaranteed to produce a well-defined probability distribution, but in practice it does produce high-quality word embeddings.

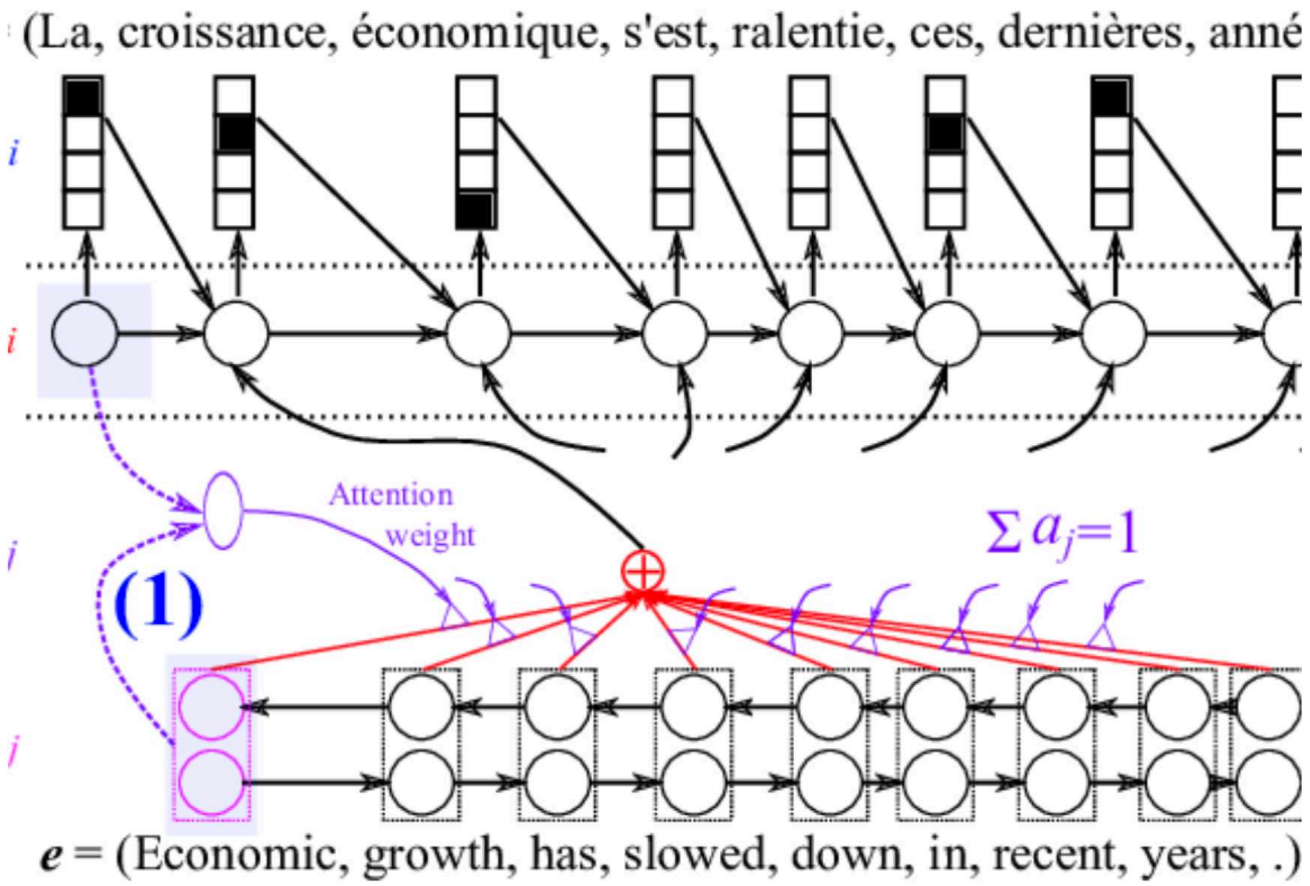
Word Vector Arithmetic



Bidirectional Recurrent Encoder



Attention Mechanism


(La, croissance, économique, s'est, ralentie, ces, dernières, anné
Attention weight
$\Sigma a_j = 1$
(1)
e = (Economic, growth, has, slowed, down, in, recent, years, .)

Reinforcement Learning Framework

- An agent interacts with its environment.
- There is a set $\mathcal{S}$ of *states* and a set $\mathcal{A}$ of *actions*.
- At each time step t, the agent is in some state s_t.
  It must choose an action a_t, whereupon it goes into state
  $s_{t+1} = \delta(s_t, a_t)$ and receives reward $r_t = \mathcal{R}(s_t, a_t)$
- Agent has a *policy* $\pi : \mathcal{S} \to \mathcal{A}$. We aim to find an optimal policy π^* which maximizes the cumulative reward.
- In general, δ, $\mathcal{R}$ and π can be multi-valued, with a random element, in which case we write them as probability distributions

$$\delta(s_{t+1} = s \,|\, s_t, a_t) \quad \mathcal{R}(r_t = r \,|\, s_t, a_t) \quad \pi(a_t = a \,|\, s_t)$$

Models of optimality

Is a fast nickel worth a slow dime?

Finite horizon reward	$\sum_{i=0}^{h-1} r_{t+i}$
Infinite discounted reward	$\sum_{i=0}^{\infty} \gamma^i r_{t+i}, \qquad 0 \leq \gamma < 1$
Average reward	$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h-1} r_{t+i}$

- Finite horizon reward is simple computationally
- Infinite discounted reward is easier for proving theorems
- Average reward is hard to deal with, because can't sensibly choose between small reward soon and large reward very far in the future.

RL Approaches

- Value Function Learning
  - TD-Learning
  - Q-Learning
- Policy Learning
  - Hill Climbing
  - Policy Gradients
  - Evolutionary Strategy
- Actor-Critic
  - combination of Value and Policy learning

Exploration / Exploitation Tradeoff

Most of the time we should choose what we think is the best action.

However, in order to ensure convergence to the optimal strategy, we must occasionally choose something different from our preferred action, e.g.

- choose a random action 5% of the time, or
- use Softmax (Boltzmann distribution) to choose the next action:

$$P(a) = \frac{e^{\mathcal{R}(a))/T}}{\sum_{b\in\mathcal{A}} e^{\mathcal{R}(b))/T}}$$

Temporal Difference Learning

Let's first assume that $\mathcal{R}$ and δ are deterministic. Then the (true) value $V^*(s)$ of the current state s should be equal to the immediate reward plus the discounted value of the next state

$$V^*(s) = \mathcal{R}(s,a) + \gamma V^*(\delta(s,a))$$

We can turn this into an update rule for the estimated value, i.e.

$$V(s_t) \leftarrow r_t + \gamma V(s_{t+1})$$

If $\mathcal{R}$ and δ are stochastic (multi-valued), it is not safe to simply replace $V(s)$ with the expression on the right hand side. Instead, we move its value fractionally in this direction, proportional to a learning rate η

$$V(s_t) \leftarrow V(s_t) + \eta\,[r_t + \gamma V(s_{t+1}) - V(s_t)]$$

Q-Learning

For a deterministic environment, π^*, Q^* and V^* are related by

$$\pi^*(s) = \text{argmax}_a Q^*(s,a)$$
$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma V^*(\delta(s,a))$$
$$V^*(s) = \max_b Q^*(s,b)$$

So

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \max_b Q^*(\delta(s,a),b)$$

This allows us to iteratively approximate Q by

$$Q(s_t,a_t) \leftarrow r_t + \gamma \max_b Q(s_{t+1},b)$$

If the environment is stochastic, we instead write

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta\,[r_t + \gamma \max_b Q(s_{t+1},b) - Q(s_t,a_t)]$$

Policy Gradients

If $r_{\text{total}} = +1$ for a win and -1 for a loss, we can simply multiply the log probability by r_{total}. Differentials can be calculated using the gradient

$$\nabla_\theta\, r_{\text{total}} \sum_{t=1}^{m} \log \pi_\theta(a_t|s_t) = r_{\text{total}} \sum_{t=1}^{m} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

The gradient of the log probability can be calculated nicely using Softmax.

If r_{total} takes some other range of values, we can replace it with $(r_{\text{total}} - b)$ where b is a fixed value, called the baseline.

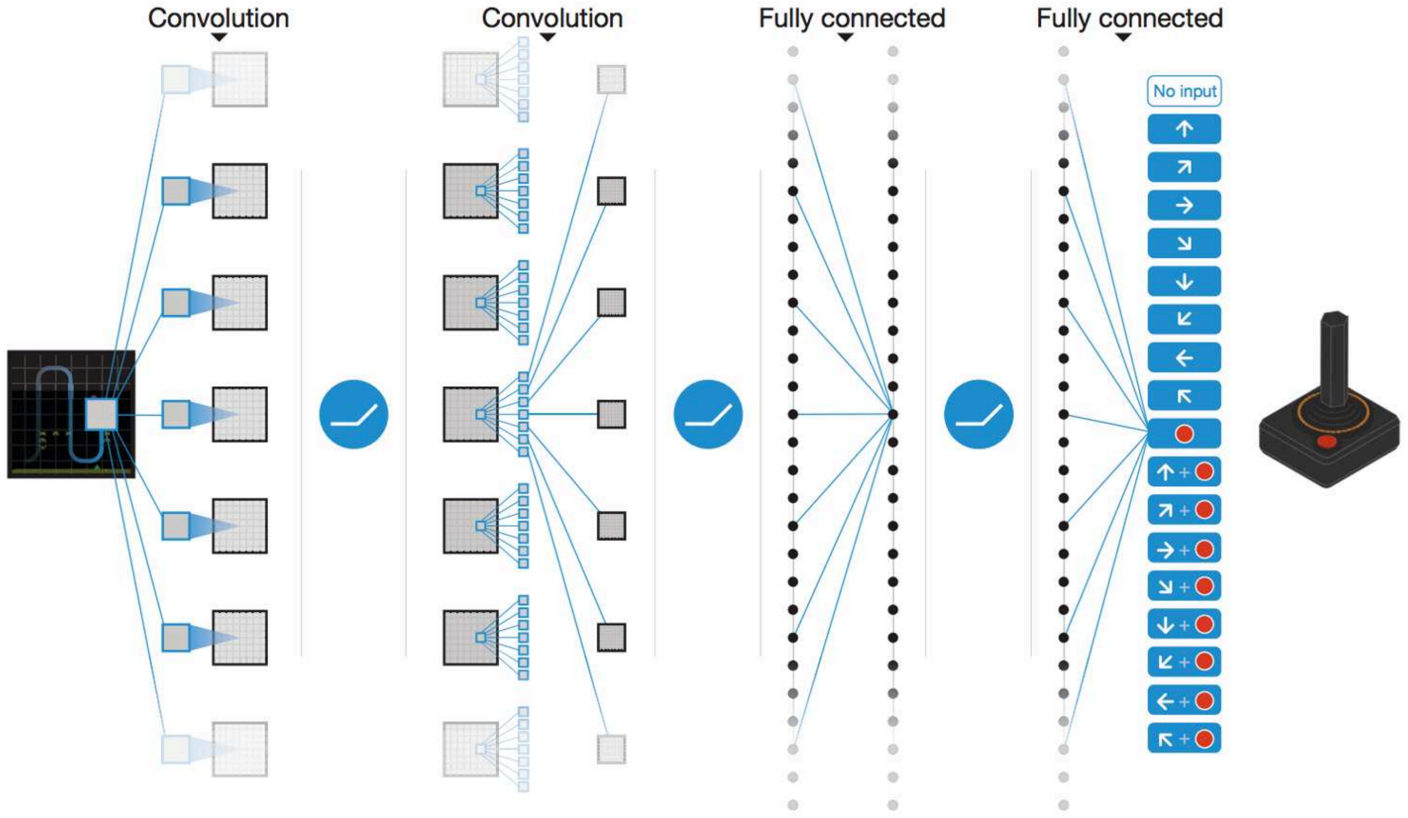
REINFORCE Algorithm

We then get the following REINFORCE algorithm:

```
for each trial
    run trial and collect states $s_t$, actions $a_t$, and reward $r_{\text{total}}$
    for $t = 1$ to length(trial)
        $\theta \leftarrow \theta + \eta(r_{\text{total}} - b)\nabla_\theta \log \pi_\theta(a_t|s_t)$
    end
end
```

This algorithm has successfully been applied, for example, to learn to play the game of Pong from raw image pixels.

Deep Q-Network



Deep Q-Learning with Experience Replay

- choose actions using current Q function (ε-greedy)
- build a database of experiences (s_t, a_t, r_t, s_{t+1})
- sample asynchronously from database and apply update, to minimize

$$[r_t + \gamma \max_b Q_w(s_{t+1}, b) - Q_w(s_t, a_t)]^2$$

- removes temporal correlations by sampling from variety of game situations in random order
- makes it easier to parallelize the algorithm on multiple GPUs

Double Q-Learning

- if the same weights w are used to select actions and evaluate actions, this can lead to a kind of confirmation bias
- could maintain two sets of weights w and $\overline{w}$, with one used for selection and the other for evaluation (then swap their roles)
- in the context of Deep Q-Learning, a simpler approach is to use the current "online" version of w for selection, and an older "target" version $\overline{w}$ for evaluation; we therefore minimize

$$[r_t + \gamma Q_{\overline{w}}(s_{t+1}, \text{argmax}_b\, Q_w(s_{t+1}, b)) - Q_w(s_t, a_t)]^2$$

- a new version of $\overline{w}$ is periodically calculated from the distributed values of w, and this $\overline{w}$ is broadcast to all processors.

Advantage Actor Critic

Recall that in the REINFORCE algorithm, a baseline b could be subtracted from r_{total}

$$\theta \leftarrow \theta + \eta(r_{\text{total}} - b)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

In the actor-critic framework, r_{total} is replaced by $Q(s_t, a_t)$

$$\theta \leftarrow \theta + \eta_\theta Q(s_t, a_t)\nabla_\theta \log \pi_\theta(a_t \,|\, s_t)$$

We can also subtract a baseline from $Q(s_t, a_t)$. This baseline must be independent of the action a_t, but it could be dependent on the state s_t. A good choice of baseline is the value function $V_u(s)$, in which case the Q function is replaced by the advantage function

$$A_w(s,a) = Q(s,a) - V_u(s)$$

Asynchronous Advantage Actor Critic

- use policy network to choose actions
- learn a parameterized Value function $V_u(s)$ by TD-Learning
- estimate Q-value by n-step sample

$$Q(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_u(s_{t+n})$$

- update policy by

$$\theta \leftarrow \theta + \eta_\theta [Q(s_t, a_t) - V_u(s_t)] \nabla_\theta \log \pi_\theta(a_t \,|\, s_t)$$

- update Value function my minimizing

$$[Q(s_t, a_t) - V_u(s_t)]^2$$

Hopfield Network

$$E(x) = -(\frac{1}{2}\sum_{i,j} x_i w_{ij} x_j + \sum_i b_i x_i)$$

Start with an initial state x and then repeatedly try to "flip" neuron activations one at a time, in order to reach a lower-energy state. If we choose to modify neuron x_i, its new value should be

$$x_i \leftarrow \begin{cases} +1, & \text{if } \sum_j w_{ij} x_j + b_i > 0, \\ x_i\,, & \text{if } \sum_j w_{ij} x_j + b_i = 0, \\ -1, & \text{if } \sum_j w_{ij} x_j + b_i < 0. \end{cases}$$

This ensures that the energy $E(x)$ will never increase. It will eventually reach a local minimum.

Boltzmann Machine (20.1)

The Boltzmann Machine uses exactly the same energy function as the Hopfield network:

$$E(x) = -\left(\sum_{i<j} x_i w_{ij} x_j + \sum_i b_i x_i\right)$$

The Boltzmann Machine is very similar to the Hopfield Network, except that

- components (neurons) x_i take on the values $0, 1$ instead of $-1, +1$
- used to generate new states rather than retrieving stored states
- update is not deterministic but stochastic, using the sigmoid

Boltzmann Machine

The Boltzmann Machine operates similarly to a Hopfield Network, except that there is some randomness in the neuron updates.

In both cases, we repeatedly choose one neuron x_i and decide whether or not to "flip" the value of x_i, thus changing from state x into x'.

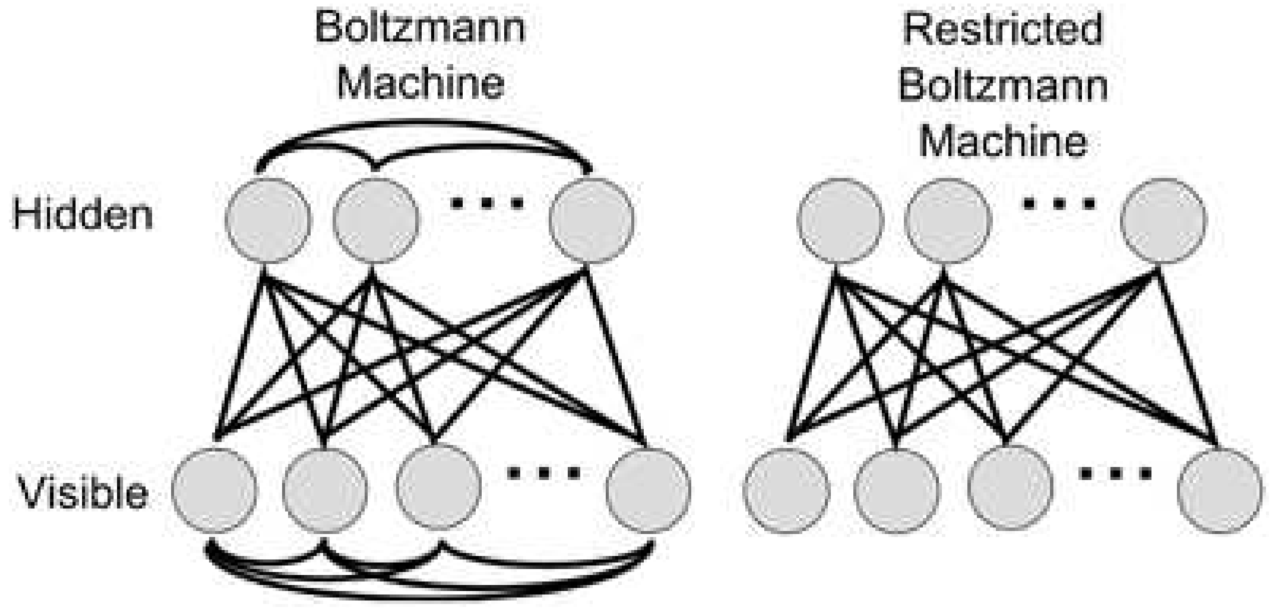
For the Hopfield Network, we do not change from x to x' unless $\Delta E \leq 0$, i.e. we never move to a higher energy state. For the Boltzmann machine, we instead choose $x_i = 1$ with probability

$$p = \frac{1}{1 + e^{-\Delta E/T}}$$

In other words, there is some probability of moving to a higher energy state (or remaining in a higher energy state when a lower one is available).

Restricted Boltzmann Machine (16.7)

If we allow visible-to-visible and hidden-to-hidden connections, the network takes too long to train. So we normally restrict the model by allowing only visible-to-hidden connections.



This is known as a Restricted Boltzmann Machine.

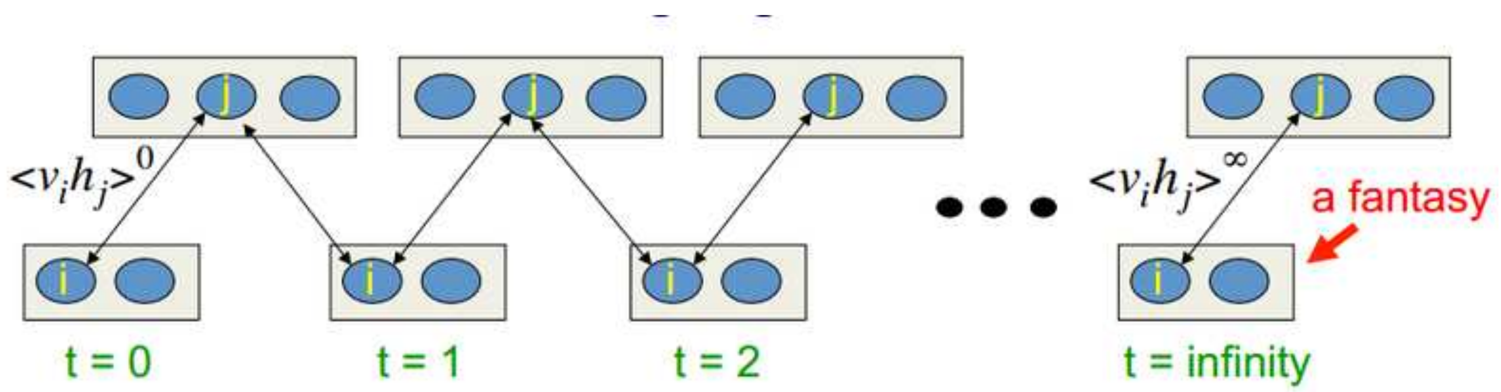
Restricted Boltzmann Machine

- inputs are binary vectors
- two-layer bi-directional neural network
  - visible layer v
  - hidden layer h
- no vis-to-vis or hidden-to-hidden connections
- all visible units connected to all hidden units

$$E(v,h) = -(\sum_i b_i v_i + \sum_j c_j h_j + \sum_{i,j} v_i w_{ij} h_j)$$

- trained to maximize the expected log probability of the data

Alternating Gibbs Sampling

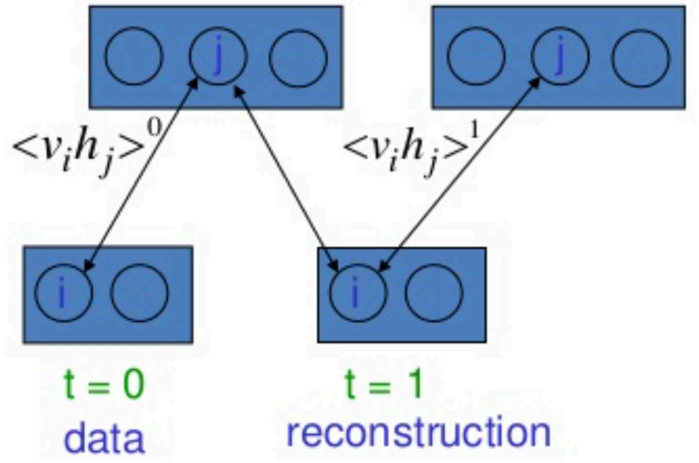


With the Restricted Boltzmann Machine, we can sample from the Boltzmann distribution as follows:

choose v_0 randomly
then sample h_0 from $p(h \mid v_0)$
then sample v_1 from $p(v \mid h_0)$
then sample h_1 from $p(h \mid v_1)$
etc.

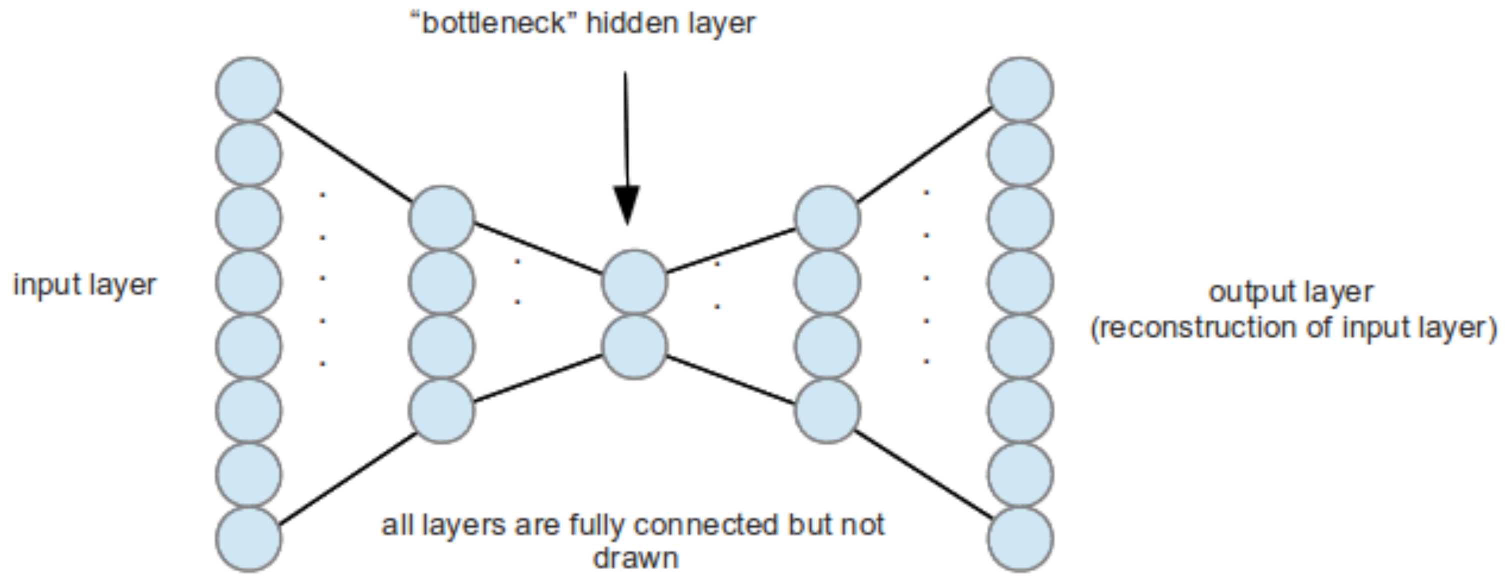
Quick Contrastive Divergence

It was noticed in the early 2000's that the process can be sped up by taking just one additional sample instead of running for many iterations.



- v_0, h_0 are used as positive sample, and v_1, h_1 as negative sample
- this can be compared to the Negative Sampling that was used with word2vec – it is not guaranteed to approximate the true gradient, but it works well in practice

Autoencoder Networks


"bottleneck" hidden layer
input layer
output layer
(reconstruction of input layer)
all layers are fully connected but not drawn

- output is trained to reproduce the input as closely as possible
- activations normally pass through a bottleneck, so the network is forced to compress the data in some way
- like the RBM, Autoencoders can be used to automatically extract abstract features from the input

Regularized Autoencoders (14.2)

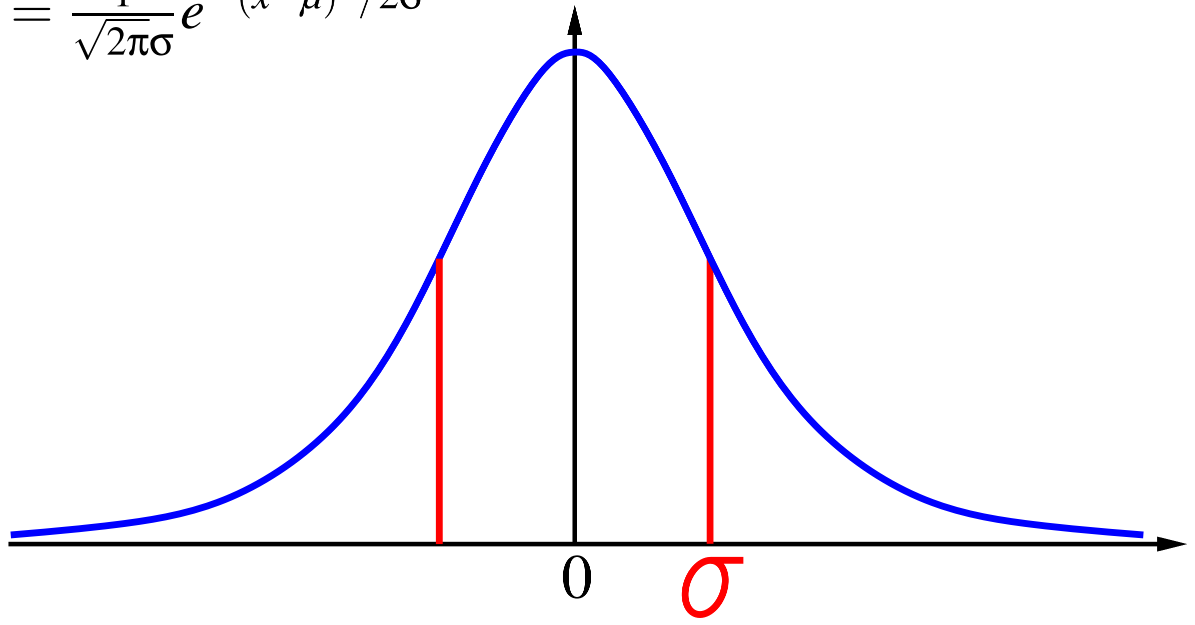
- sparse autoencoders
- autoencoders with dropout at hidden layer(s)
- contractive autoencoders
- denoising autoencoders

Generative Models

- Sometimes, as well as reproducing the training items $\{x^{(i)}\}$, we also want to be able to use the decoder to generate new items which are of a similar "style" to the training items.
- In other words, we want to be able to choose latent variables z from a standard Normal distribution $p(z)$, feed these values of z to the decoder, and have it produce a new item x which is somehow similar to the training items.
- Generative models can be:
  - explicit (Variational Autoencoders)
  - implicit (Generative Adversarial Networks)

Gaussian Distribution (3.9.3)

$P_{\mu,\sigma}(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$



$\mu =$ mean

$\sigma =$ standard deviation

Multivariate Gaussian: $P_{\mu,\sigma}(x) = \prod_i P_{\mu_i,\sigma_i}(x_i)$

Entropy and KL-Divergence

- The entropy of a distribution $q()$ is $H(q) = \int_\theta q(\theta)(-\log q(\theta))d\theta$
- In Information Theory, H(q) is the amount of information (bits) required to transmit a random sample from distribution $q()$
- For a Gaussian distribution, $H(q) = \sum_i \log \sigma_i$
- KL-Divergence $\mathrm{D_{KL}}(q \,\|\, p) = \int_\theta q(\theta)(\log q(\theta) - \log p(\theta))d\theta$
- $\mathrm{D_{KL}}(q \,\|\, p)$ is the number of extra bits we need to trasmit if we designed a code for $p()$ but the samples are drawn from $q()$ instead.
- If $p(z)$ is Standard Normal distribution, minimizing $D_{\mathrm{KL}}\big(q_\phi(z) \| p(z)\big)$ encourages $q_\phi()$ to center on zero and spread out to approximate $p()$.

Variational Autoencoder (20.10.3)

Instead of producing a single z for each $x^{(i)}$, the encoder (with parameters ϕ) can be made to produce a mean $\mu_{z|x^{(i)}}$ and standard deviation $\sigma_{z|x^{(i)}}$

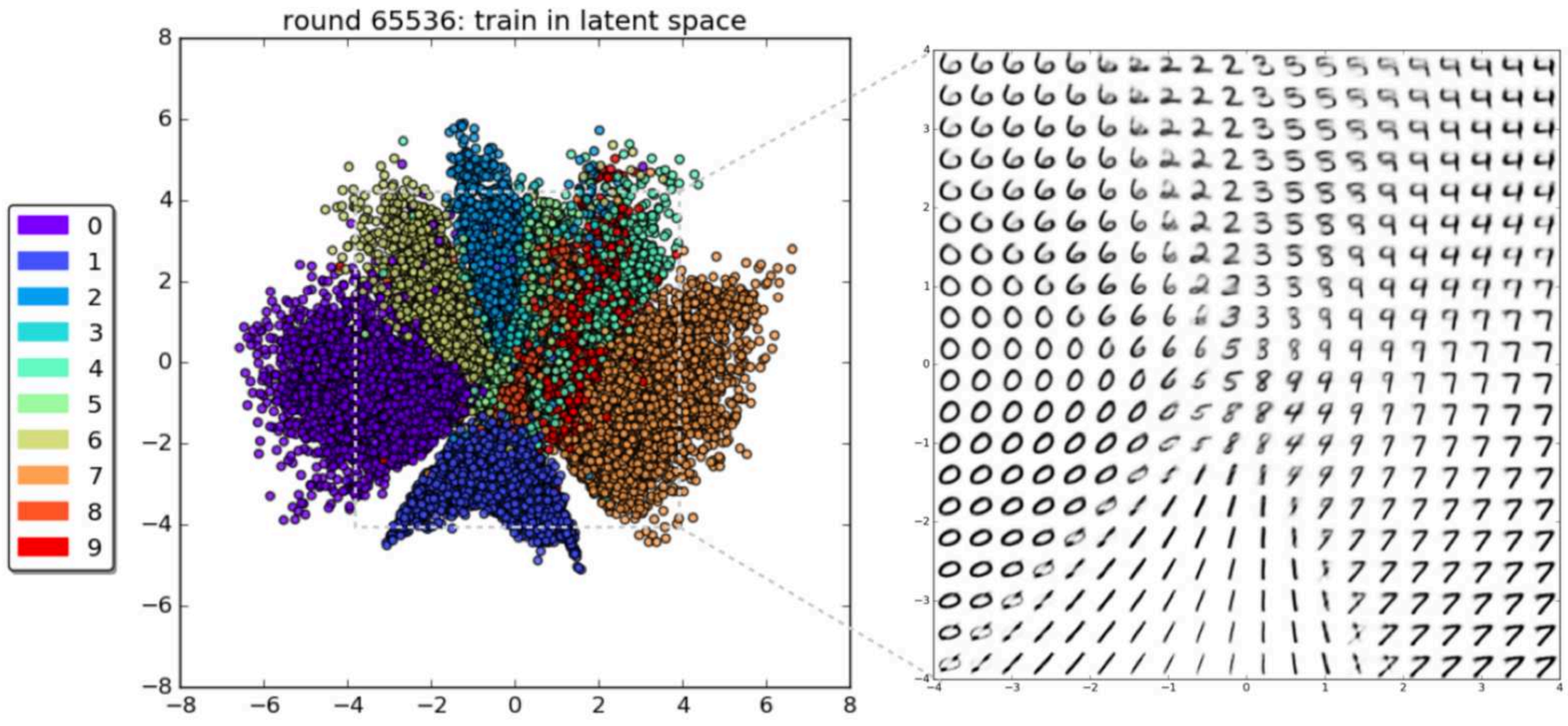
This defines a conditional (Gaussian) probability distribution $q_\phi(z|x^{(i)})$

We then train the system to maximize

$$\mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)}|z)\right] \;-\; D_{\text{KL}}\left(q_\phi(z|x^{(i)})\|p(z)\right)$$

- the first term enforces that any sample z drawn from the conditional distribution $q_\phi(z|x^{(i)})$ should, when fed to the decoder, produce somthing approximating $x^{(i)}$
- the second term encourages $q_\phi(z|x^{(i)})$ to approximate $p(z)$
- in practice, the distributions $q_\phi(z|x^{(i)})$ for various $x^{(i)}$ will occupy complementary regions within the overall distribution $p(z)$

Variational Autoencoder Digits



Generative Adversarial Networks

Generator (Artist) G_θ and Discriminator (Critic) D_ψ are both Deep Convolutional Neural Networks.

Generator $G_\theta : z \mapsto x$, with parameters θ, generates an image x from latent variables z (sampled from a Normal distribution).

Discriminator $D_\psi : x \mapsto D_\psi(x) \in (0,1)$, with parameters ψ, takes an image x and estimates the probability of the image being real.

Generator and Discriminator play a 2-player zero-sum game to compute:

$$\min_\theta \max_\psi \Big(\mathbf{E}_{x \sim p_\text{data}} \big[\log D_\psi(x) \big] + \mathbf{E}_{z \sim p_\text{model}} \big[\log \big(1 - D_\psi(G_\theta(z)) \big) \big] \Big)$$

Discriminator tries to maximize the bracketed expression,
Generator tries to minimize it.

Generative Adversarial Networks

Alternate between:

Gradient ascent on Discriminator:

$$\max_{\psi} \Big(\mathbf{E}_{x \sim p_{\text{data}}} \big[\log D_{\psi}(x) \big] + \mathbf{E}_{z \sim p_{\text{model}}} \big[\log \big(1 - D_{\psi}(G_{\theta}(z)) \big) \big] \Big)$$

Gradient descent on Generator, using:

$$\cancel{\min_{\theta} \mathbf{E}_{z \sim p_{\text{model}}} \big[\log \big(1 - D_{\psi}(G_{\theta}(z)) \big) \big]}$$

This formula puts too much emphasis on images that are correctly classified. Better to do gradient ascent on Generator, using:

$$\max_{\theta} \mathbf{E}_{z \sim p_{\text{model}}} \big[\log \big(D_{\psi}(G_{\theta}(z)) \big) \big]$$

This puts more emphasis on the images that are wrongly classified.

Generative Adversarial Networks

```
repeat:
    for k steps do
        sample minibatch of $m$ latent samples $\{z^{(1)},\ldots,z^{(m)}\}$ from $p(z)$
        sample minibatch of $m$ training items $\{x^{(1)},\ldots,x^{(m)}\}$
        update Discriminator by gradient ascent on $\psi$:
            $$\nabla_\psi \frac{1}{m}\sum_{i=1}^{m}\left[\log D_\psi(x^{(i)}) + \log\left(1 - D_\psi(G_\theta(z^{(i)}))\right)\right]$$
    end for
    sample minibatch of $m$ latent samples $\{z^{(1)},\ldots,z^{(m)}\}$ from $p(z)$
    update Generator by gradient ascent on $\theta$:
        $$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m}\log\left(D_\psi(G_\theta(z^{(i)}))\right)$$
end repeat
```

GAN Generated Images