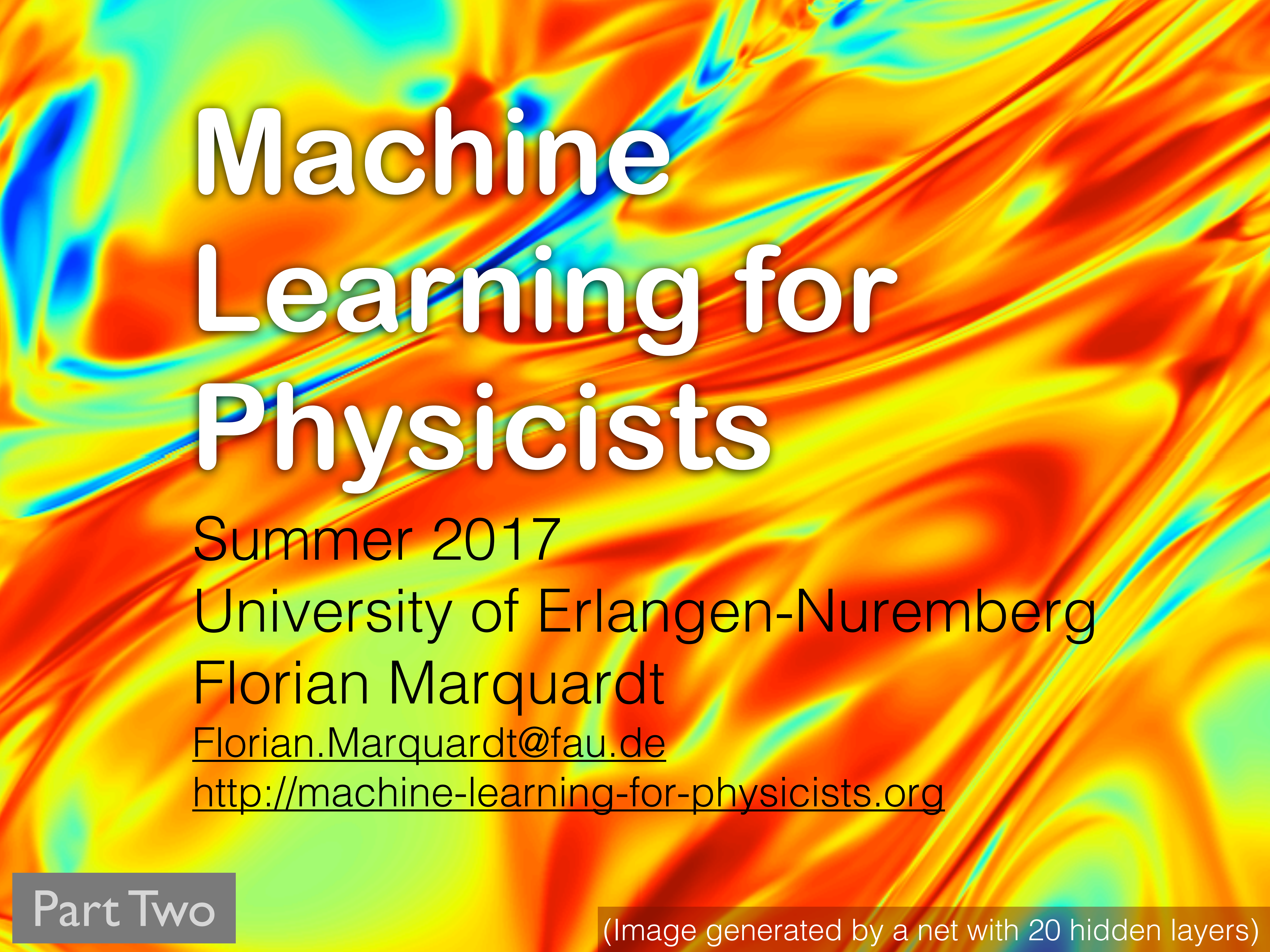
Machine Learning for Physicists

Summer 2017

University of Erlangen-Nuremberg
Florian Marquardt
Florian.Marquardt@fau.de

http://machine-learning-for-physicists.org

Part Two



(Image generated by a net with 20 hidden layers)

Optimized gradient descent algorithms

How to speed up stochastic gradient descent?

- accelerate ("momentum") towards minimum
- Automatically choose learning rate
- ...even different rates for different weights

Summary: a few gradient update methods

see overview by S. Ruder https://arxiv.org/abs/1609.04747

adagrad	divide by RMS of all previous gradients
RMSprop	divide by RMS of last few previous gradients
adadelta	same, but multiply also by RMS of last few parameter updates
adam	divide running average of last few gradients by RMS of last few gradients (* with corrections during earliest steps)

adam often works best

"adagrad"

$$g_j = \frac{\partial C}{\partial \theta_j}$$

Standard:

$$\dot{\theta} = -\eta g$$

or in discrete time steps

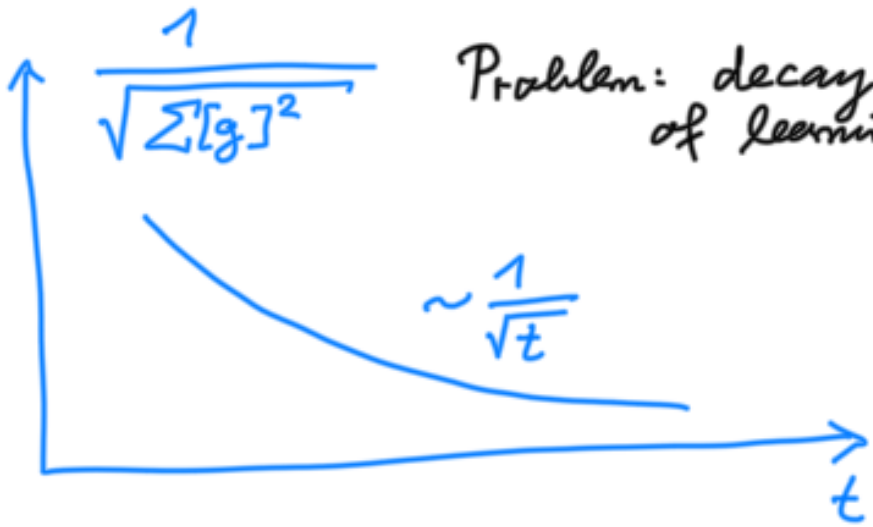
$$\theta^{(t+1)} = \theta^{(t)} - \eta g^{(t)}$$

Idea: rescale according to estimate of typical size of g

$$\Delta\theta_j = -\eta \frac{g_j^{(t)}}{\sqrt{\sum_{t' \leq t} [g_j^{(t')}]^2 + \underbrace{\varepsilon}_{\text{avoid division by zero}}}}$$

"root mean square (RMS)" of all previous gradients

"RMSprop"



Problem: decay of learning rate

<u>Solution</u>: Only take RMS of "last few gradients"

$$V^{(t)} = \underbrace{\gamma}_{\text{decay term (e.g. } \gamma \sim 0.9)} V^{(t-1)} + (1-\gamma)[g^{(t)}]^2$$

$$\Rightarrow \quad V^{(t)} = (1-\gamma) \sum_{t' \leq t} \underbrace{\gamma^{t-t'}}_{\text{exponential decay of earlier contributions}} \cdot [g^{(t')}]^2$$

$\Rightarrow$ Roughly: contributions from $\sim \frac{1}{1-\gamma}$ last terms

e.g. for time-independent $g^{(t)} = g$:

$$V^{(t)} = (1-\gamma) \underbrace{\left(\sum_{t' \leq t} \gamma^{t-t'} \right)}_{\frac{1}{1-\gamma}} g^2 = g^2 \ \checkmark$$

"RMSprop"

$$\Delta \theta_j^{(t)} = -\eta \frac{g_j^t}{\sqrt{V_j^{(t)} + \varepsilon}}$$

“adadelta”

$$\tilde{V}^{(t)} = \gamma \tilde{V}^{(t-1)} + (1-\gamma)\left[\Delta\theta^{(t)}\right]^2$$

$$\Delta\theta^{(t)} = -\underbrace{\frac{\tilde{V}^{(t-1)}}{V^{(t)}}\, g^{(t)}}_{\text{\underline{no} learning rate } \eta}$$

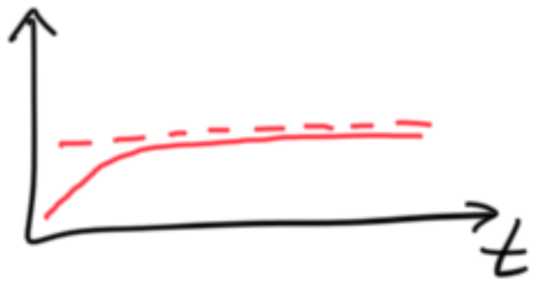
"has right dimensions" $\frac{\Delta\theta}{g} \cdot g = \Delta\theta$

"adam"

$$m^{(t)} = \beta m^{(t-1)} + (1-\beta) g^{(t)}$$

$V^{(t)}$ like before

Little problem: $m^{(t)} \approx 0$, $V^{(t)} \approx 0$ in first steps



$\Rightarrow$ correct via

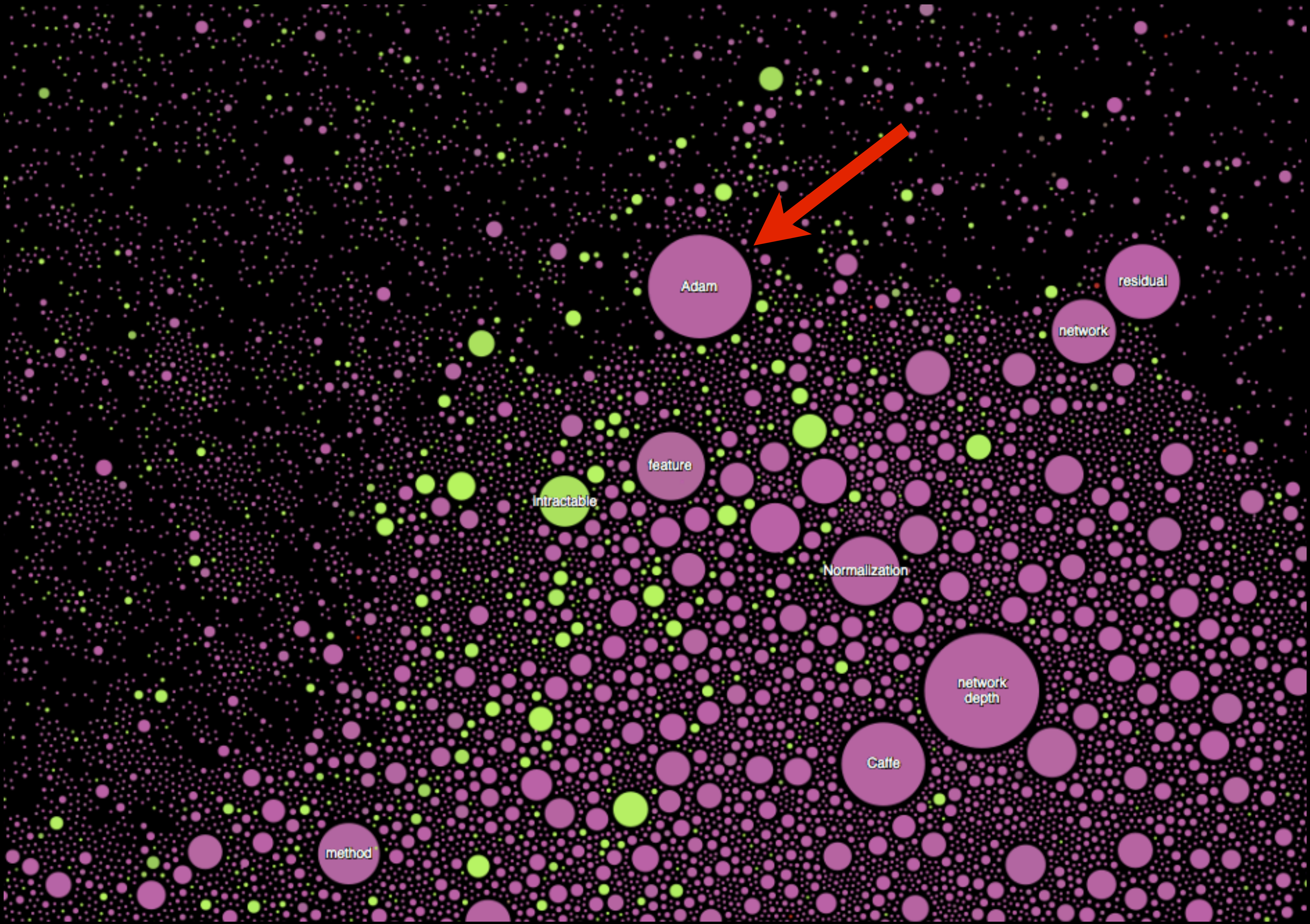
$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta^t}$$

$$\hat{V}^{(t)} = \frac{V^{(t)}}{1-\gamma^t}$$

$\Rightarrow$ set

$$\Delta\theta^{(t)} = -\eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{V}^{(t)} + \varepsilon}}$$

$\beta \approx 0.9$
$\gamma \approx 0.999$
$\varepsilon \approx 10^{-8}$

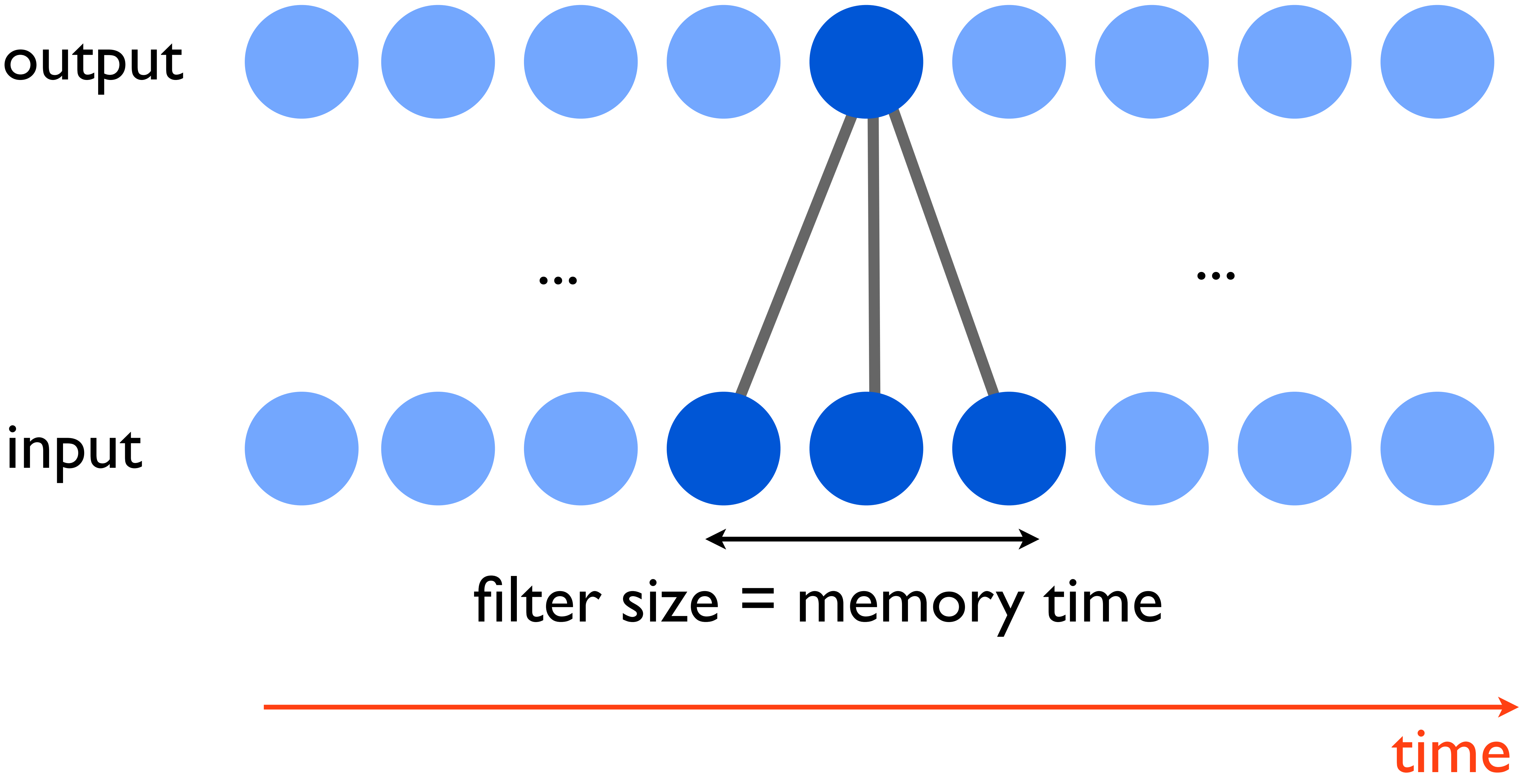

Adam
residual
network
feature
Intractable
Normalization
network depth
Caffe
method

Recurrent neural networks

Networks "with memory"

Useful for analyzing time-evolution (time-series of data), for analyzing and translating sentences, for control/feedback (e.g. robotics or action games), and many other things

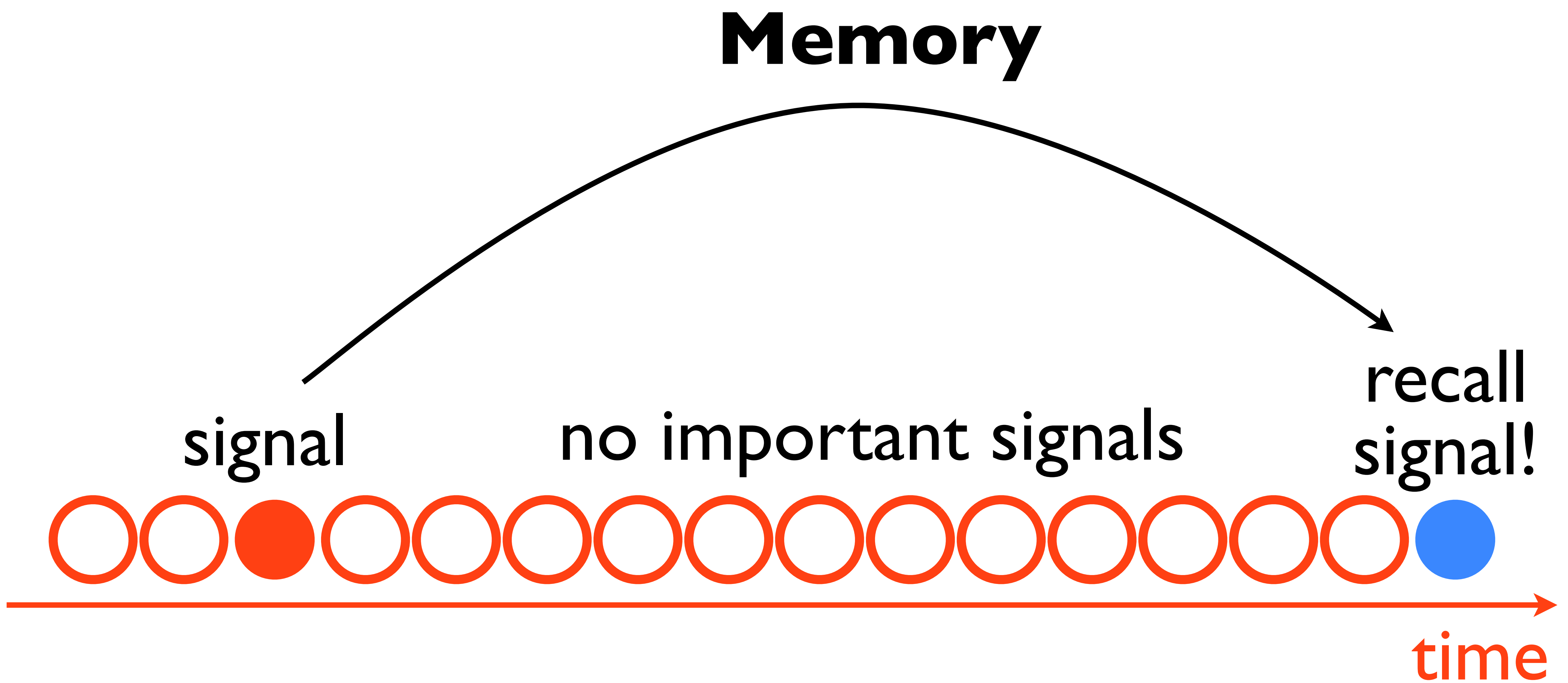
Could use a convolutional network!



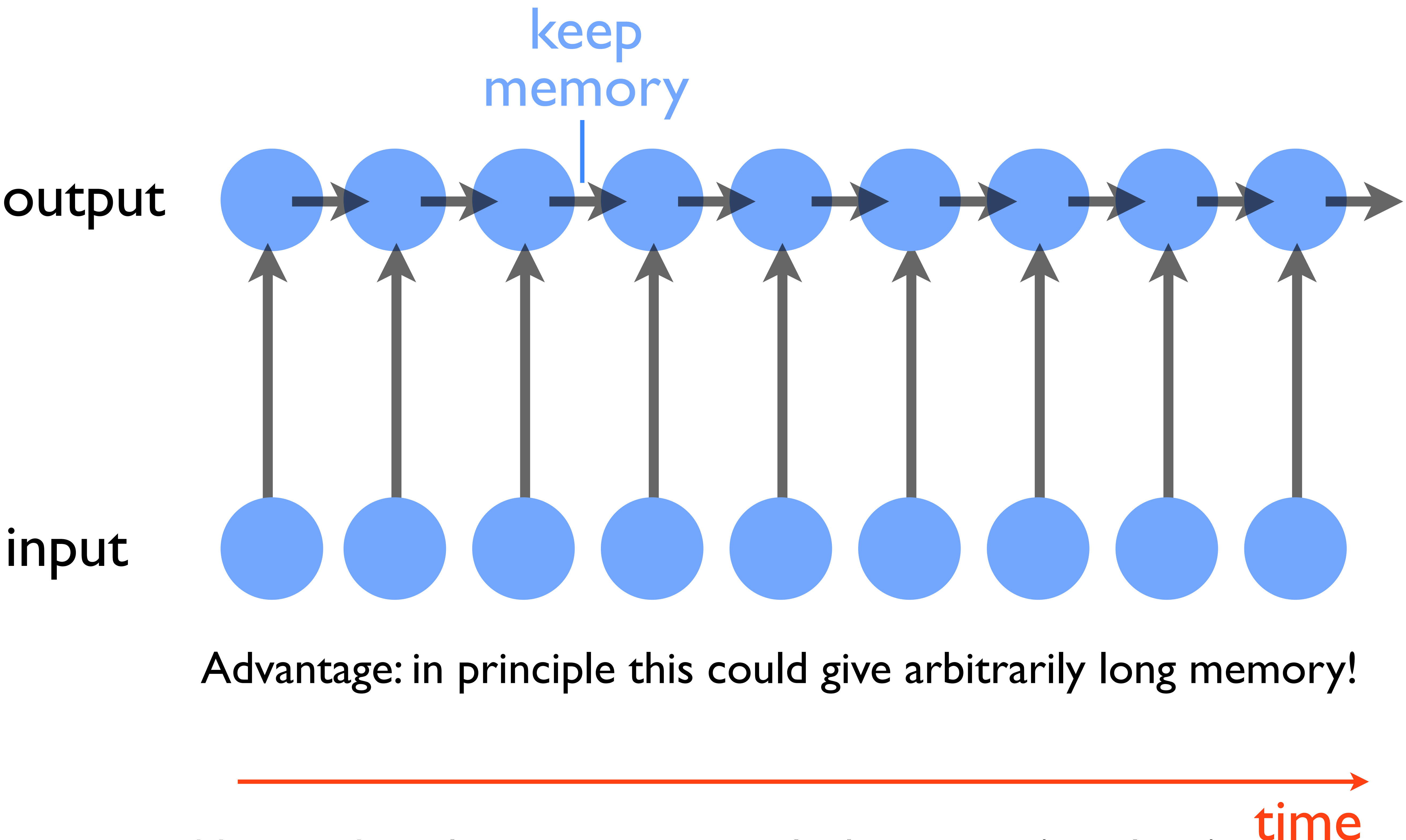
Long memories with convolutional nets are challenging:

- would need large filter sizes
- even then, would need to know required memory time beforehand
- can expand memory time efficiently by multi-layer network with subsampling (pooling), but this is still problematic for precise long-term memory

But: may be OK for some physics applications! (problems local in time, with short memory)


Memory
signal
no important signals
recall
signal!
time

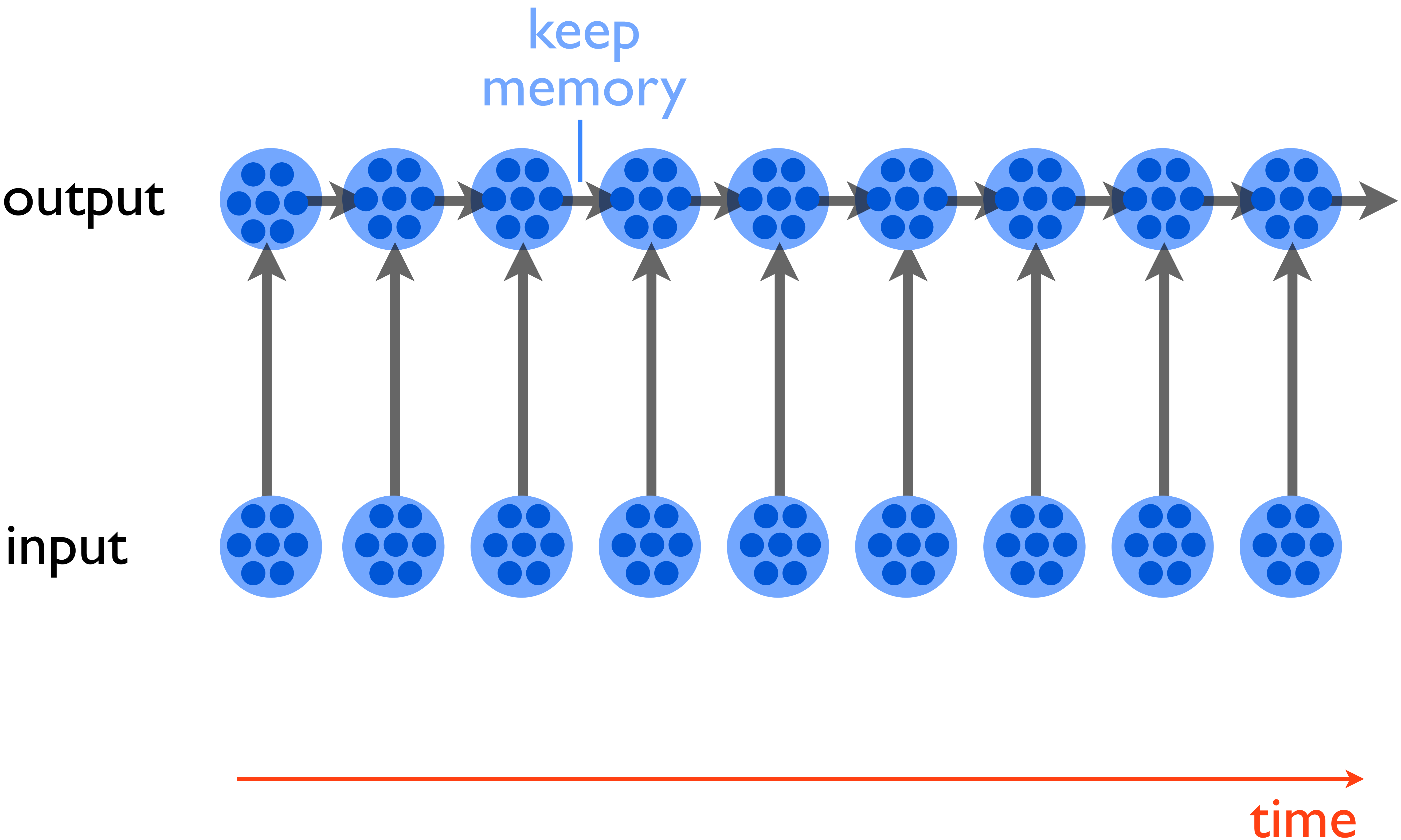
Solution: Recurrent Neural Networks (RNN)



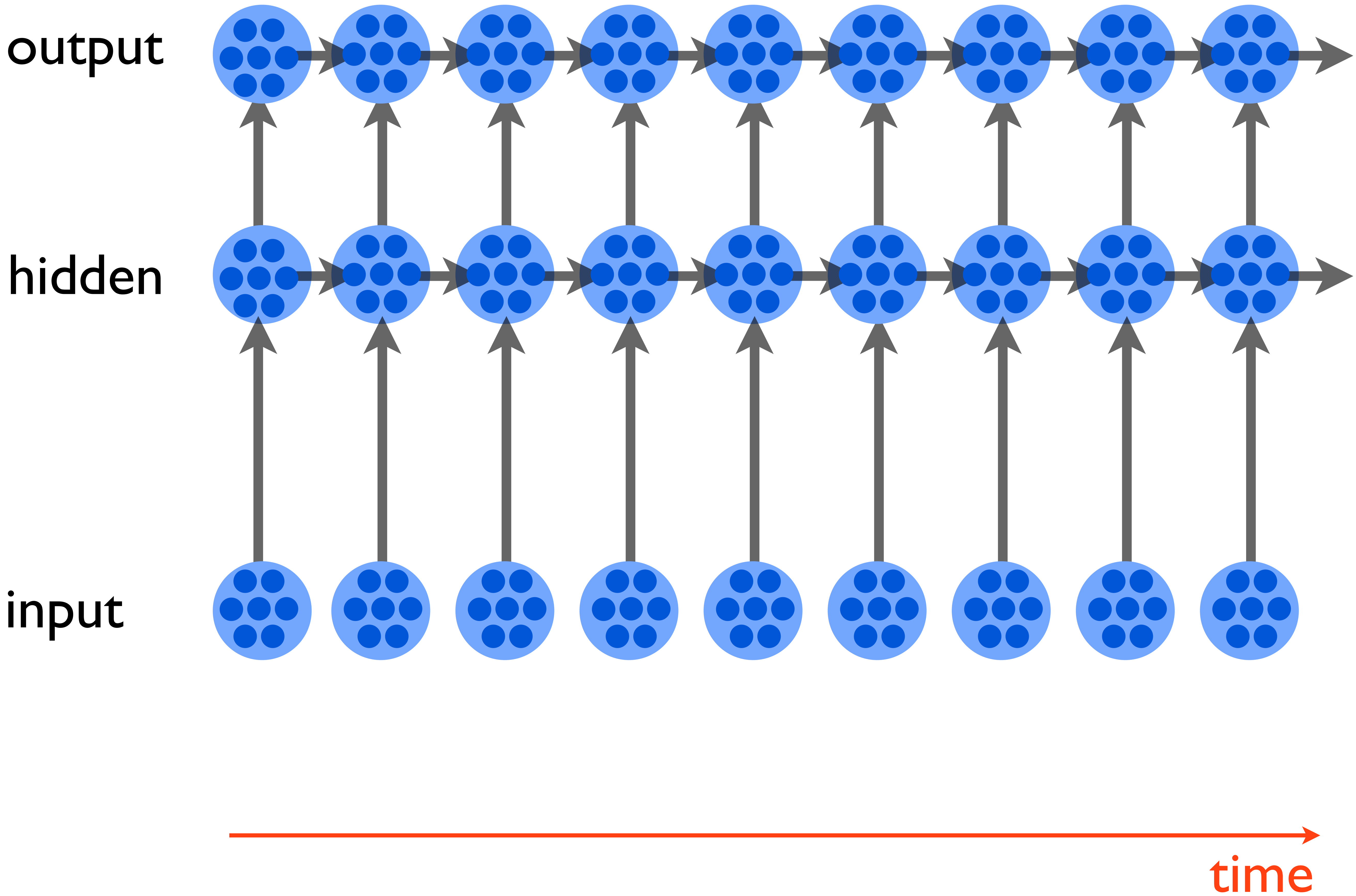
Advantage: in principle this could give arbitrarily long memory!

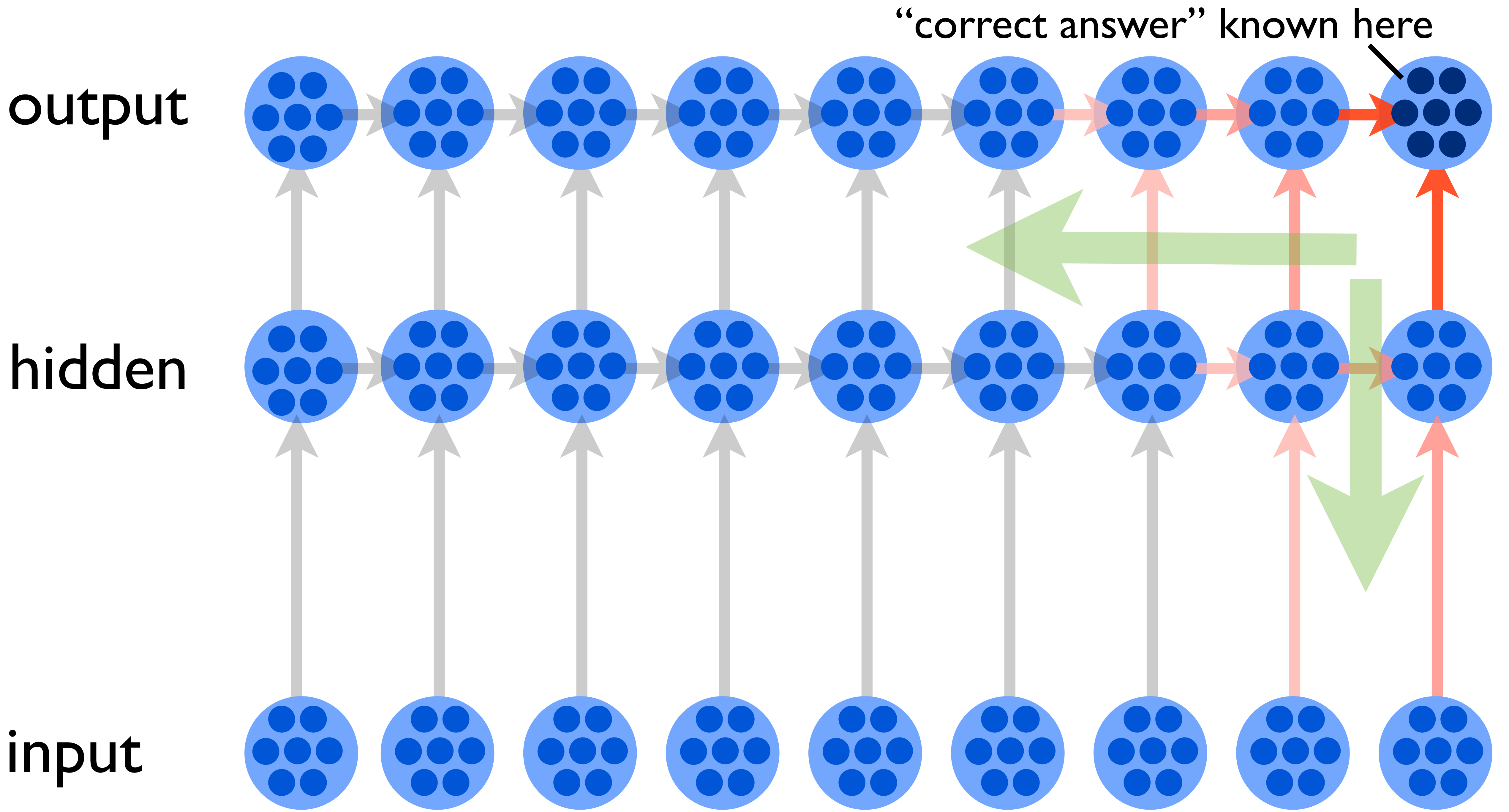
Note: each circle may represent multiple neurons (i.e. a layer)
Each arrow then represents all possible connections between those neurons

Solution: Recurrent Neural Networks (RNN)



Note: the weights are not time-dependent, i.e. need to store only one set of weights (similar to convolutional net)


output
hidden
input
time



"Backpropagation through time"

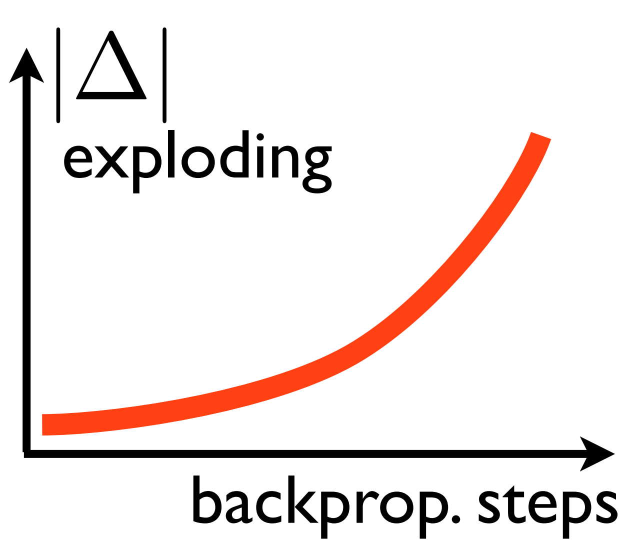
Long memories with recurrent networks are challenging, due to a feature of backpropagation:

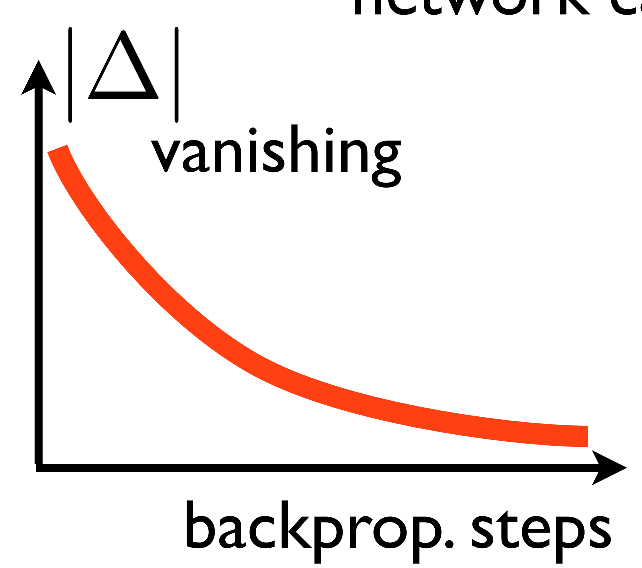
"Exploding gradients" / "Vanishing gradients"

Backpropagation through many layers (in a deep network) or through many time-steps (in a recurrent network):

Something like $\Delta_{t-1} = M_t \Delta_t$ (for the recurrent network case)

Depending on typical eigenvalues of the matrices M:





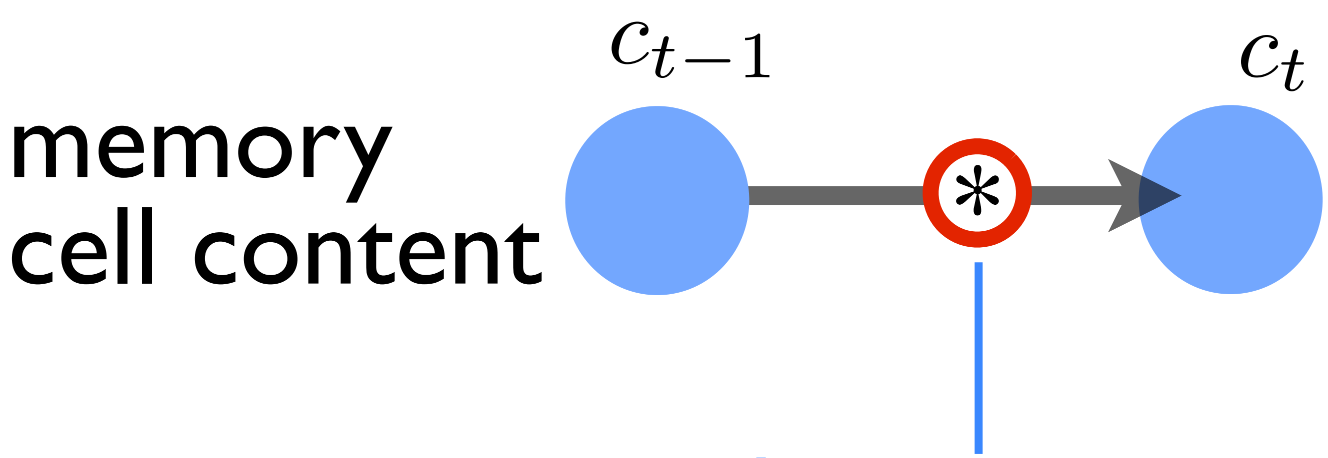
Long short-term memory (LSTM)

Why this name? "Long-term memory" would be the weights that are adapted during training and then stored forever. "Short-term memory" is the input-dependent memory we are talking about here. "Long short-term memory" tries to have long memory times in a robust way, for this short-term memory.

Sepp Hochreiter and Jürgen Schmidhuber, 1997

Main idea: determine read/write/delete operations of a memory cell via the network (through other neurons)
Most of the time, a memory neuron just sits there and is not used/changed!

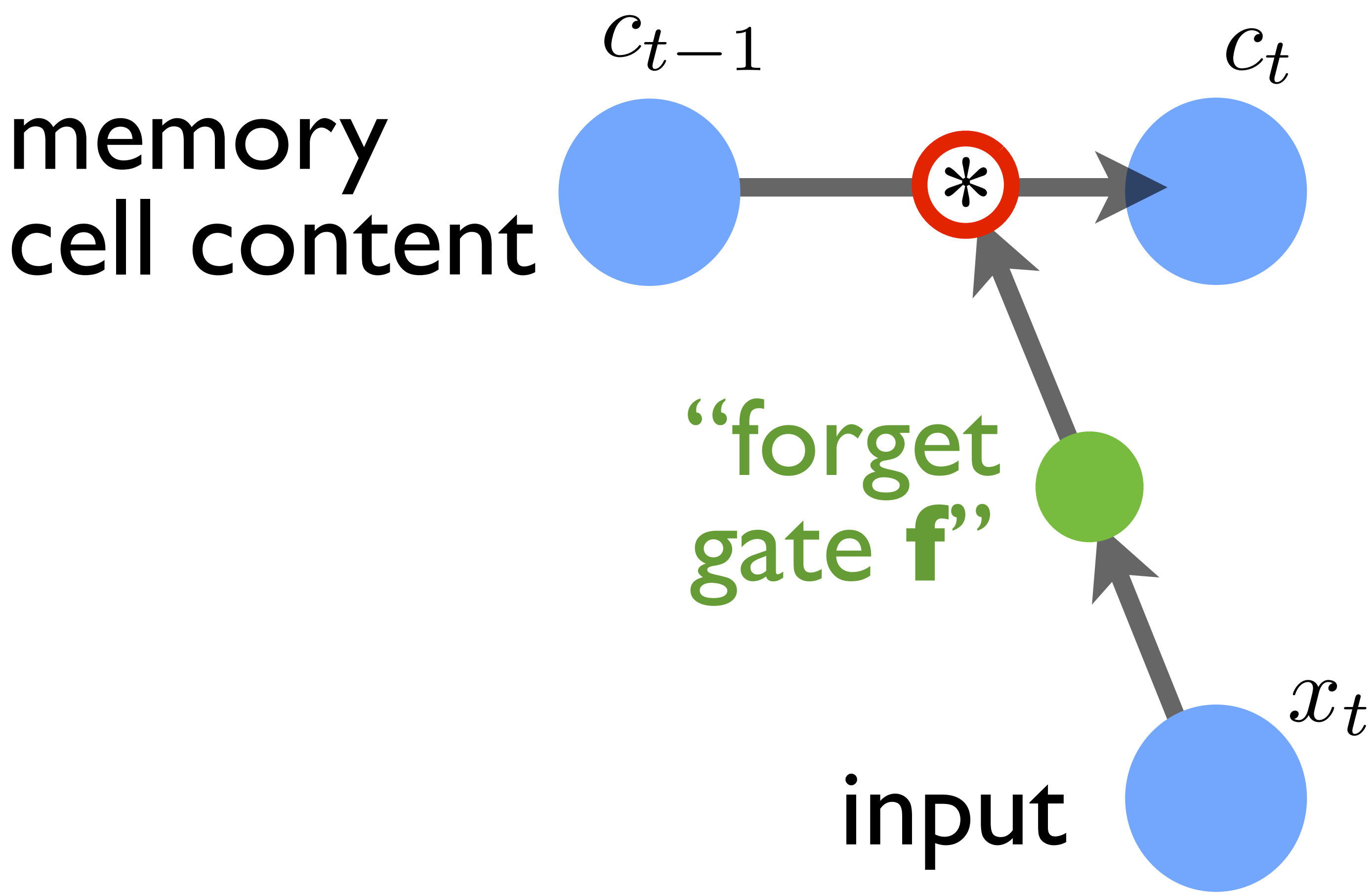
LSTM: Forget gate (delete)



keep: $c_t = 1 * c_{t-1}$

delete: $c_t = 0 * c_{t-1}$

LSTM: Forget gate (delete)



Calculate "forget gate":

$$f = \sigma(W^{(f)} x_t + b^{(f)})$$

sigmoid

(usually x,b,f are vectors, W the weight matrix)

Obtain new memory content:

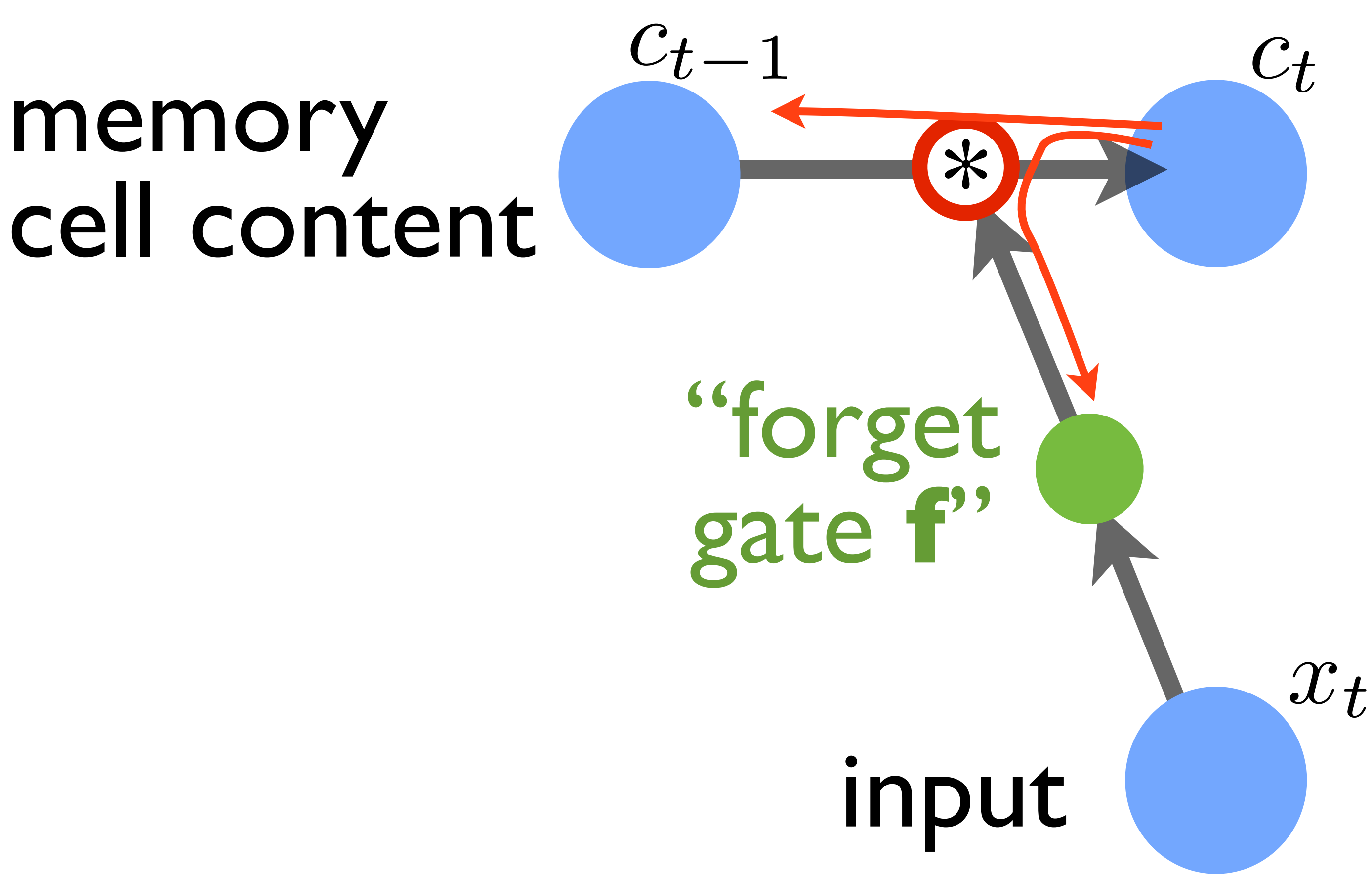
$$c_t = f * c_{t-1}$$

elementwise product

NEW: for the first time, we are **multiplying** neuron values!

LSTM: Forget gate (delete)

Backpropagation



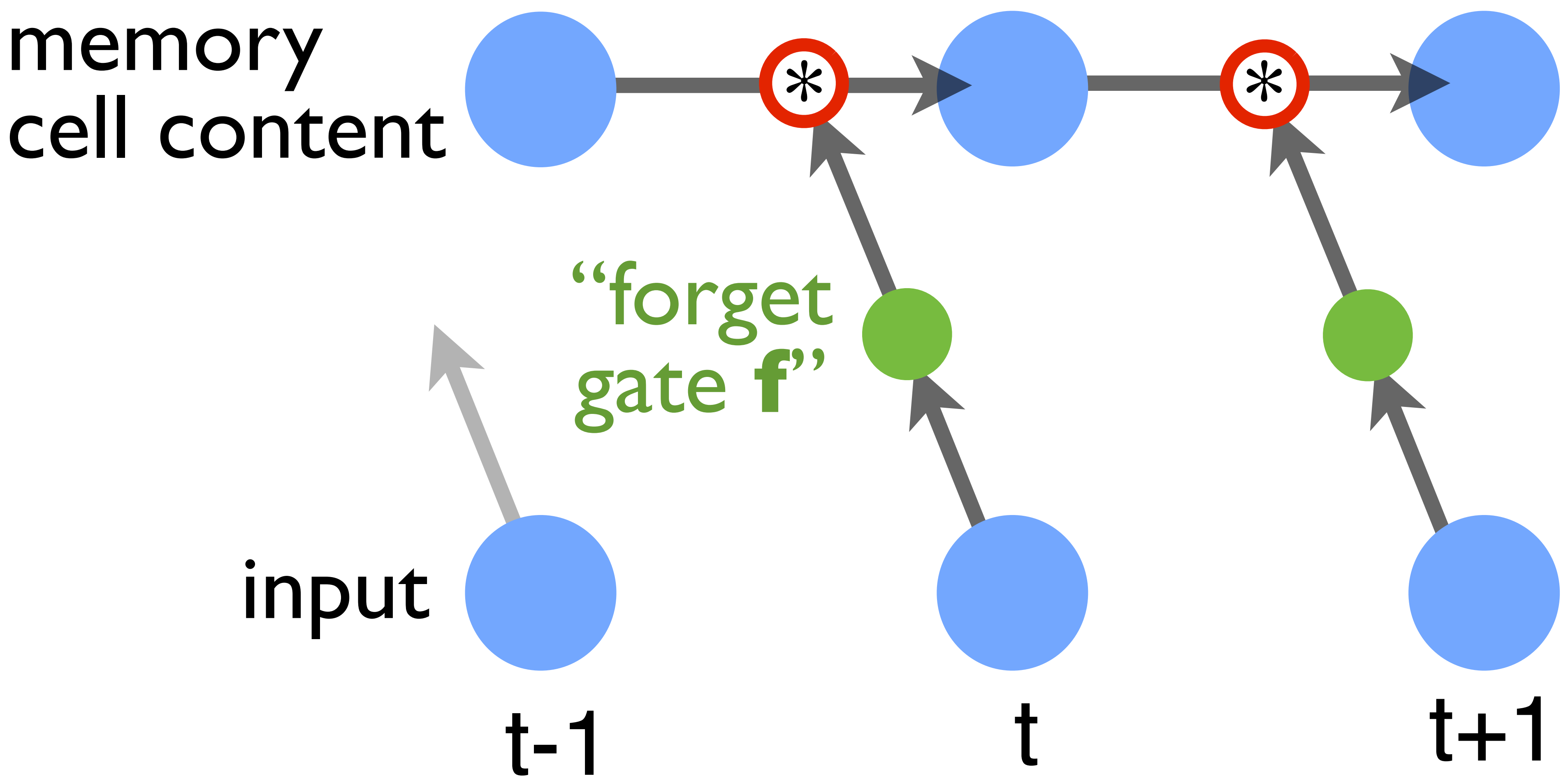
The multiplication * splits the error backpropagation into two branches

product rule:

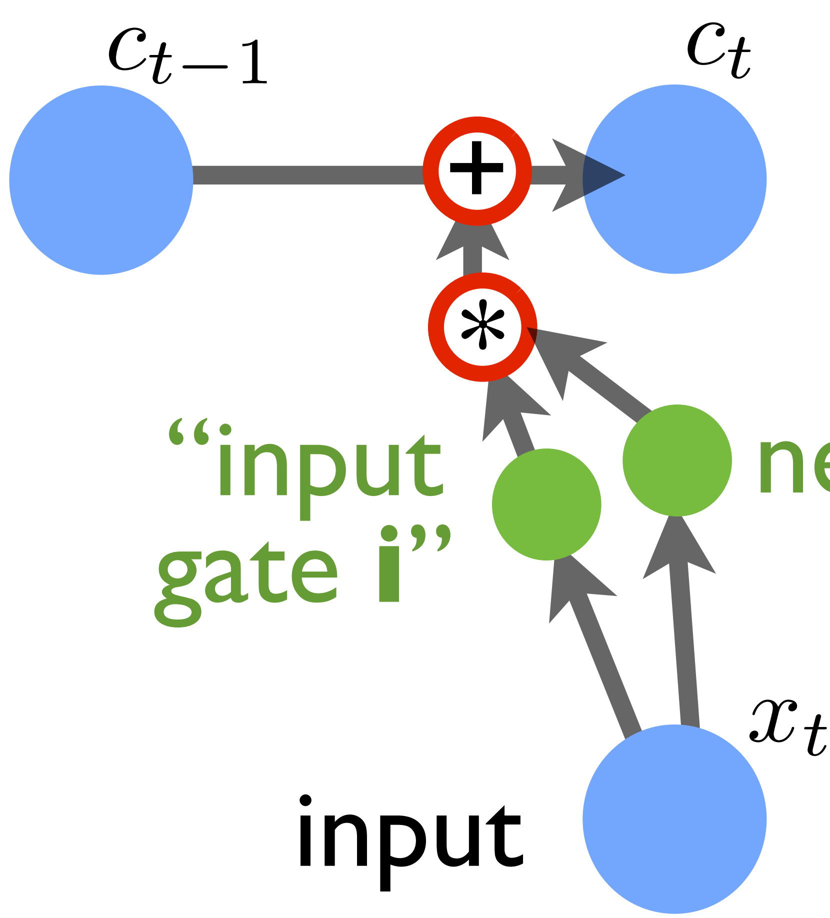
$$\frac{\partial f_j c_{t-1,j}}{\partial w_*} = \frac{\partial f_j}{\partial w_*} c_{t-1,j} + f_j \frac{\partial c_{t-1,j}}{\partial w_*}$$

(Note: if time is not specified, we are referring to t)

LSTM: Forget gate (delete)



LSTM: Write new memory value



$$i = \sigma(W^{(i)} x_t + b^{(i)})$$

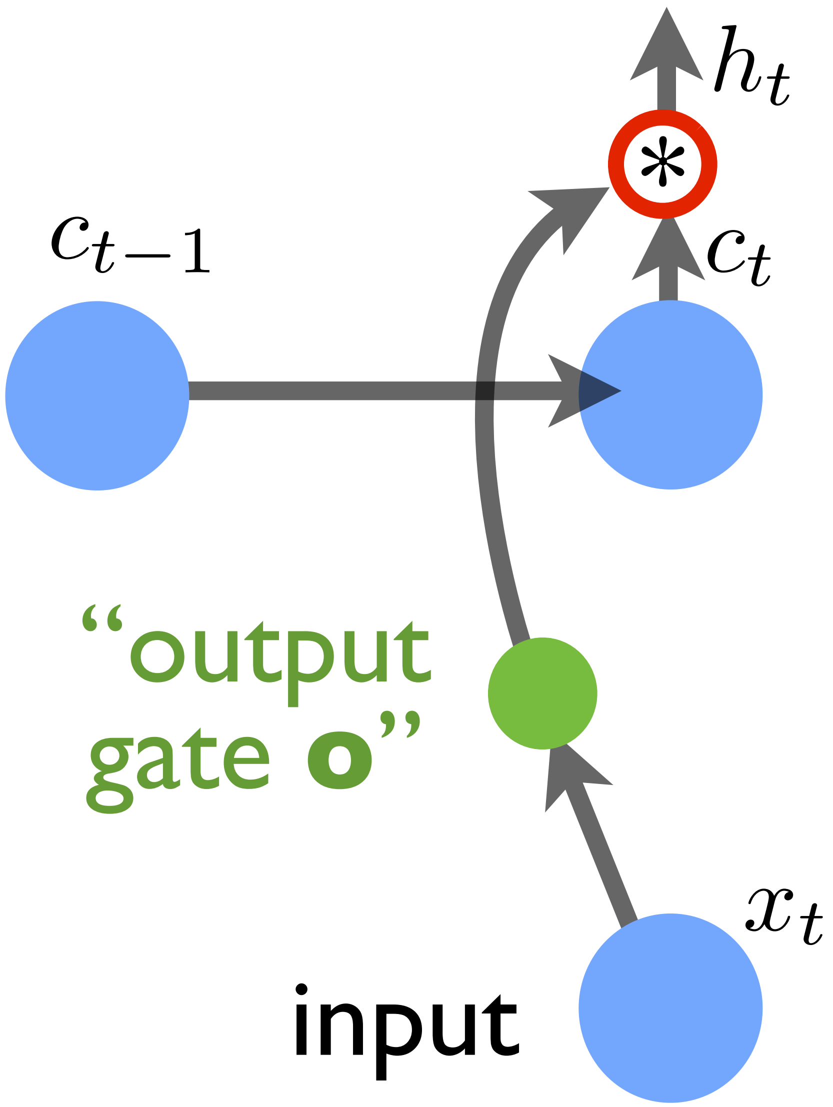
$$\tilde{c}_t = \tanh(W^{(c)} x_t + b^{(c)})$$

both delete and write together:

$$c_t = f * c_{t-1} + i * \tilde{c}_t$$

forget new value

LSTM: Read (output) memory value



$$o = \sigma(W^{(o)} x_t + b^{(o)})$$

$$h_t = o * \tanh(c_t)$$

LSTM: exploit previous memory output ‘h’

make f,i,o etc. at time t depend on output ‘h’ calculated in previous time step!

(otherwise: ‘h’ could only be used in higher layers, but not to control memory access in present layer)

$$f = \sigma(W^{(f)} x_t + U^{(f)} h_{t-1} + b^{(f)})$$

...and likewise for every other quantity!

Thus, result of readout can actually influence subsequent operations (e.g.: readout of some selected other memory cell!)

Sometimes, o is even made to depend on c_t

LSTM: backpropagation through time is OK

As long as memory content is not read or written, the backpropagation gradient is trivial:

$$c_t = c_{t-1} = c_{t-2} = \dots$$

$$\frac{\partial c_t}{\partial w_*} = \frac{\partial c_{t-1}}{\partial w_*} = \frac{\partial c_{t-2}}{\partial w_*} = \dots$$

(deviation vector multiplied by 1)

During those 'silent' time-intervals: No explosion or vanishing gradient!

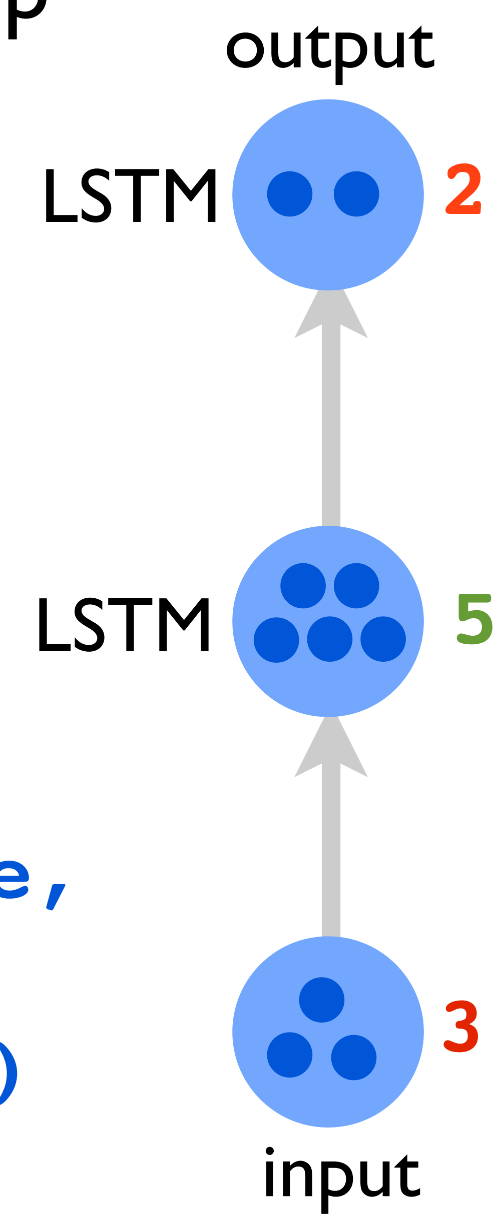
Adding an LSTM layer with 10 memory cells:

Each of those cells has the full structure, with **f**,**i**,**o** gates and the memory content **c**, and the output **h**.

```
rnn.add(LSTM(10, return_sequences=True))
```

whether to return the full
time sequence of outputs, or only
the output at the final time

Two LSTM layers (input > LSTM > LSTM=output), taking an input of 3 neuron values for each time step and producing a time sequence with 2 neuron values for each time step

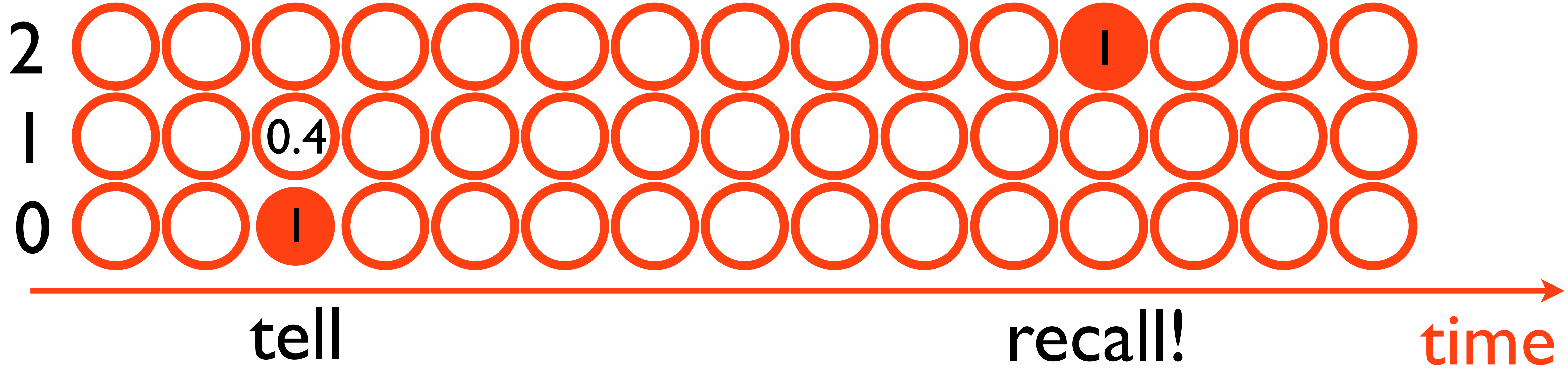



```
def init_memory_net():
    global rnn, batchsize, timesteps
     rnn = Sequential()
    # note: batch_input_shape is
(batchsize,timesteps,data_dim)
    rnn.add(LSTM(5, batch_input_shape=(None,
   timesteps, 3), return_sequences=True))
    rnn.add(LSTM(2, return_sequences=True))
    rnn.compile(loss='mean_squared_error',
   optimizer='adam', metrics=['accuracy'])
```

Example: A network for recall

(see code on website)

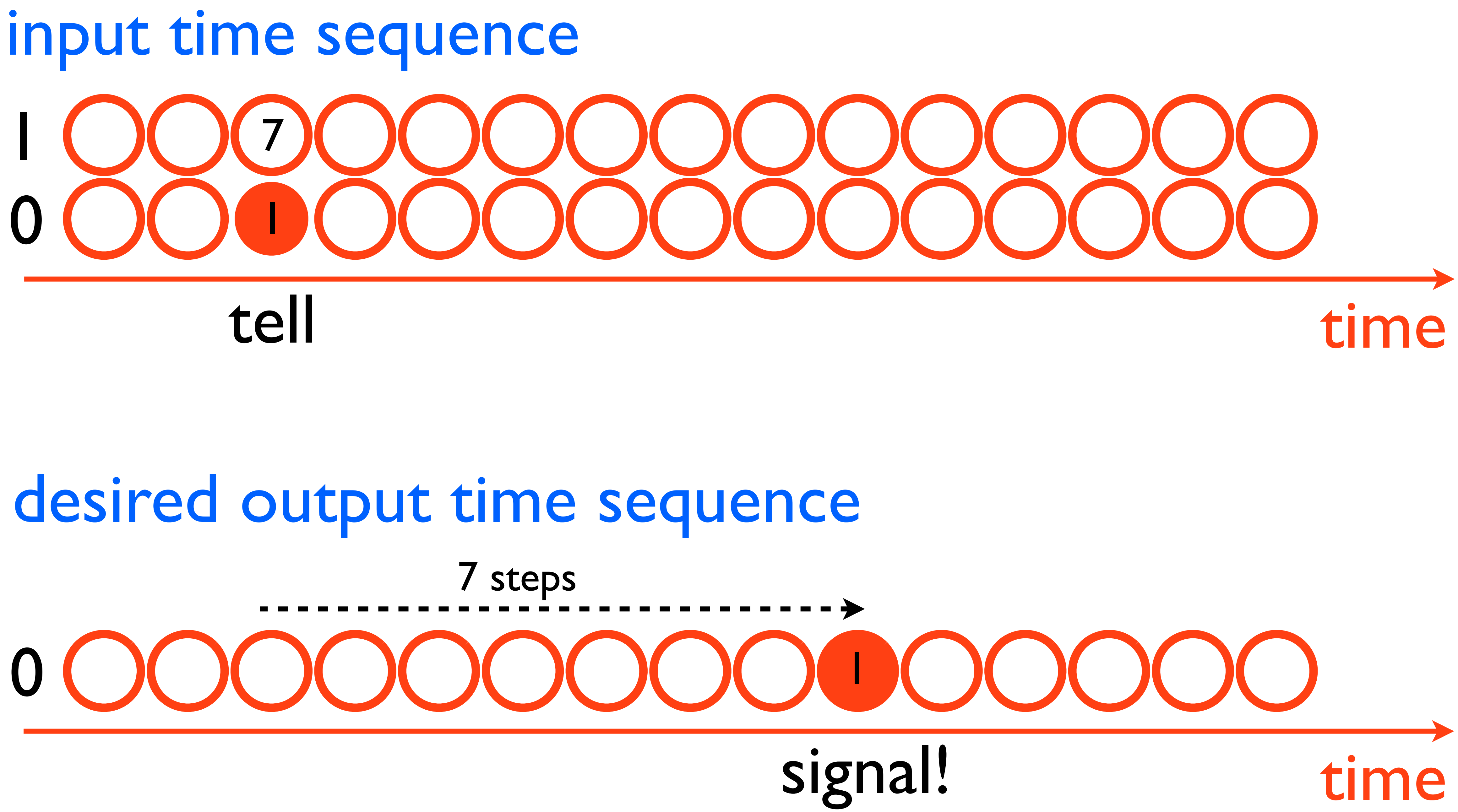
input time sequence



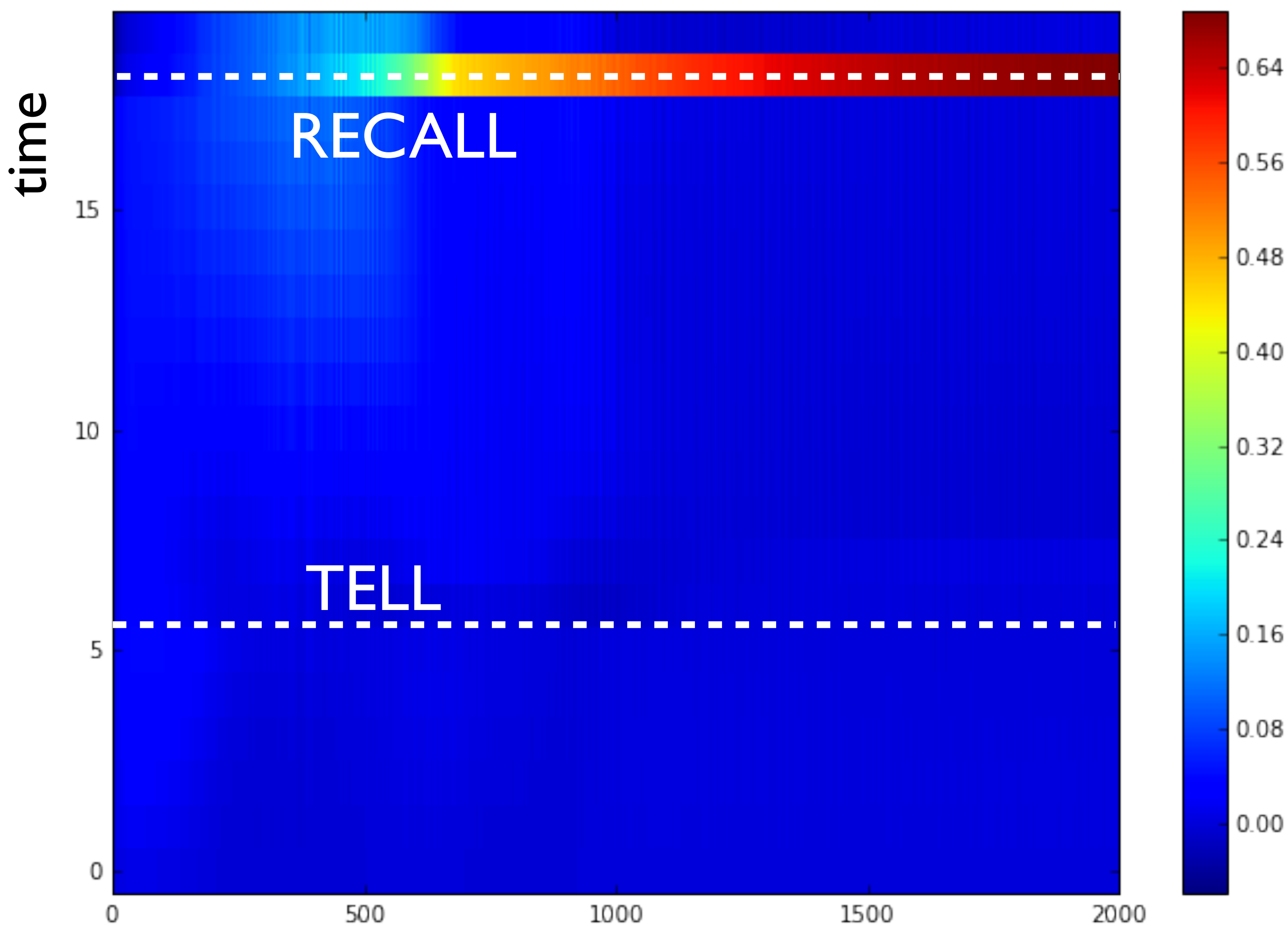
desired output time sequence

Example: A network that counts down

(see code on website)

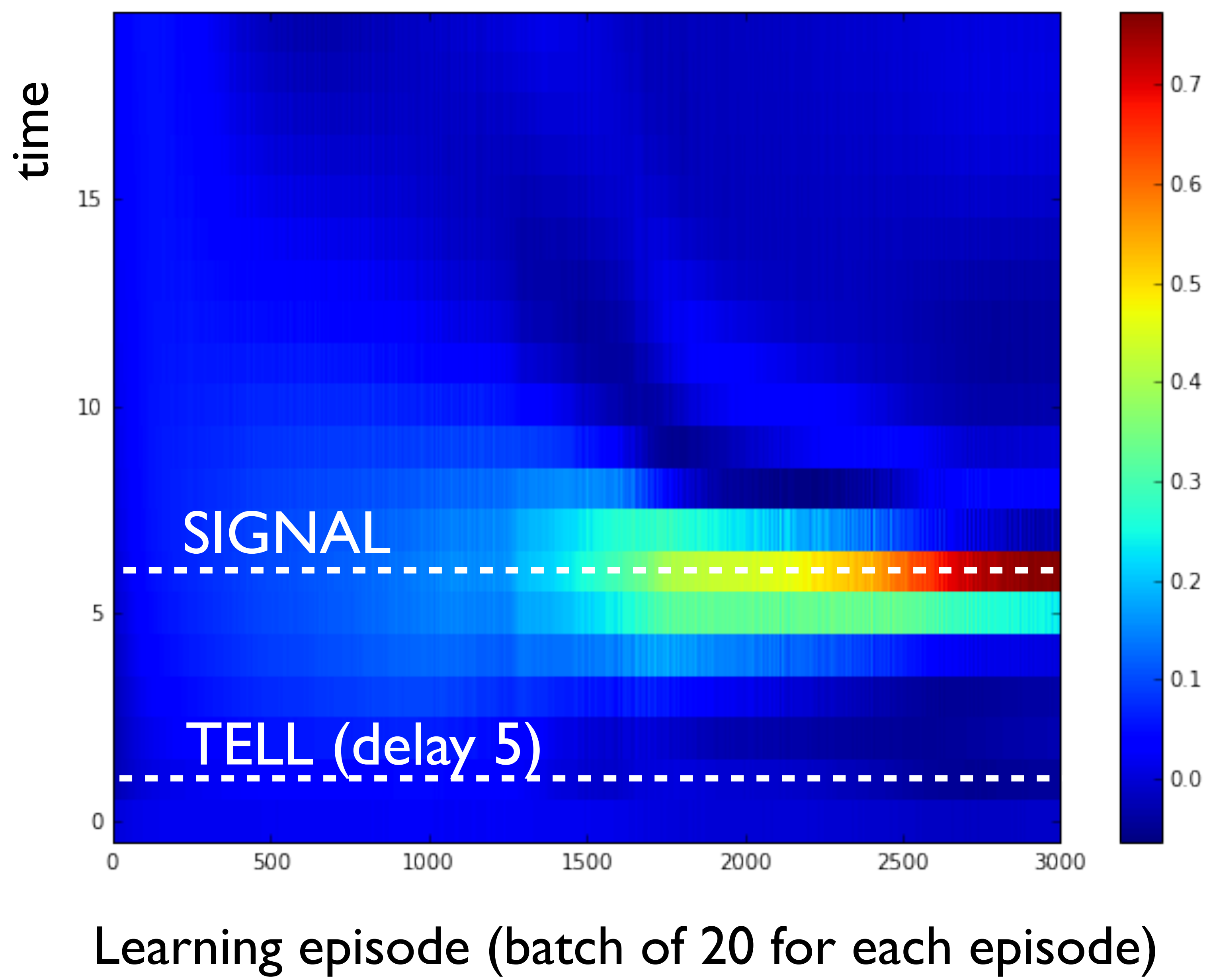


Output of the recall network, evolving during training (for a fixed input sequence)



Learning episode (batch of 20 for each episode)

Output of the countdown network, evolving during training (for a fixed input sequence)



Character generation

input sequence

T H E _ T H E O R Y _ O F _ G E

(characters in one-hot encoding)

time

desired output: predict next character

H E _ T H E O R Y _ O F _ G E N

network will output probability for **each** possible character, at each time step

ABCDEFGHIJKLMNOPQRSTUVWXYZ_

(example for second time-step)

Character generation

Example by Andrej Karpathy

training on MBs of text

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on
aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

```
we counter. He stutn co des. His stanted out one ofler that concossions and was
to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn
```

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

Homework

Train a network that is eventually able to carry out sums or differences:

```
input     3+5=??
output    ....08

input     7-5=??
output    ....02
```

How do you encode the input/output sequences? What happens when the result has two digits? etc.

Word vectors

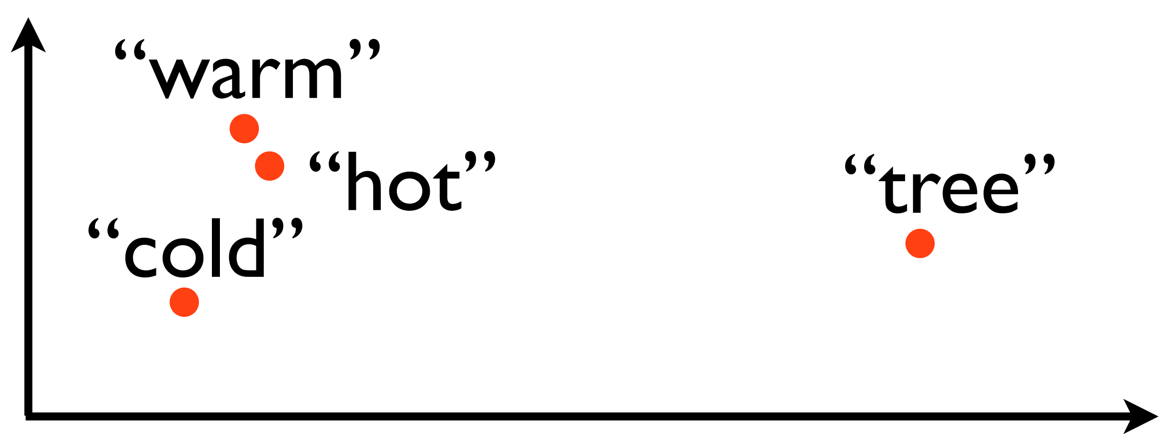
simple one-hot encoding of words needs large vectors (and they do not carry any special meaning):

"warm"	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	...
	tree	bird	apple	sun	desk	hot	cold	warm	that	full	nice	one	two	three	was	

dimension: number of words in dictionary

word2vec – reduction to vectors in much lower dimension, where similar words lie closer together:

"warm"	0.3	0	0.1	0	2	1.2	0
"hot"	0.2	0	0.2	0	2.5	1.3	0
"cold"	0.1	0	0	2	0.4	0.2	0.3



Word vectors: recurrent net for training

Mikolov, Yih, Zweig 2013

$$\mathrm{w}(t) \longrightarrow \mathrm{s}(t) \longrightarrow \mathrm{y}(t)$$

input word at time t (one-hot; dimension D= size of dictionary)

dim. N

predicted next word (probabilities for each word in vocab.; dimension D)

$$\mathbf{s}(t) = f\left(\overset{\text{NxD}}{\mathbf{U}}\mathbf{w}(t) + \overset{\text{NxN}}{\mathbf{W}}\mathbf{s}(t{-}1)\right)$$

$$\mathbf{y}(t) = g\left(\overset{\text{DxN}}{\mathbf{V}}\mathbf{s}(t)\right),$$

where U matrix (NxD) contains word vectors!

$$f(z) = \frac{1}{1+e^{-z}}, \quad g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}.$$

sigmoid SOFTMAX

Word vectors: how to train them

Predicting the probability of any word in the dictionary, given the context words (most recent word): very expensive!

Alternative:

Noise-contrastive estimation: provide a few noisy (wrong) examples, and train the model to predict that they are fake (but that the true one is correct)!

Word vectors: how to train them

Two approaches:

"continuous bag of words" Context words $\longrightarrow$ word

"skip-gram" word $\longrightarrow$ context words

Example dataset:

the quick brown fox jumped over the lazy dog

word	context words (here: just surrounding words)
quick	**the, brown**
over	**jumped, the**
lazy	**the, dog**
...	**...**

Word vectors: how to train them

Model tries to predict:

$P_\theta(w, h)$

prob. that w is the correct word,
given the context word h

parameters of the model, i.e. weights, biases,
and entries of embedding vectors

$$P_\theta(w, h) = \sigma(W_{jk} e_k(h) + b_j)$$

j: index for word w in dictionary
k: index in embedding vector [Einstein sum]
e(h): embedding vector for word h
W,b: weights, biases

At each time-step: go down the gradient of

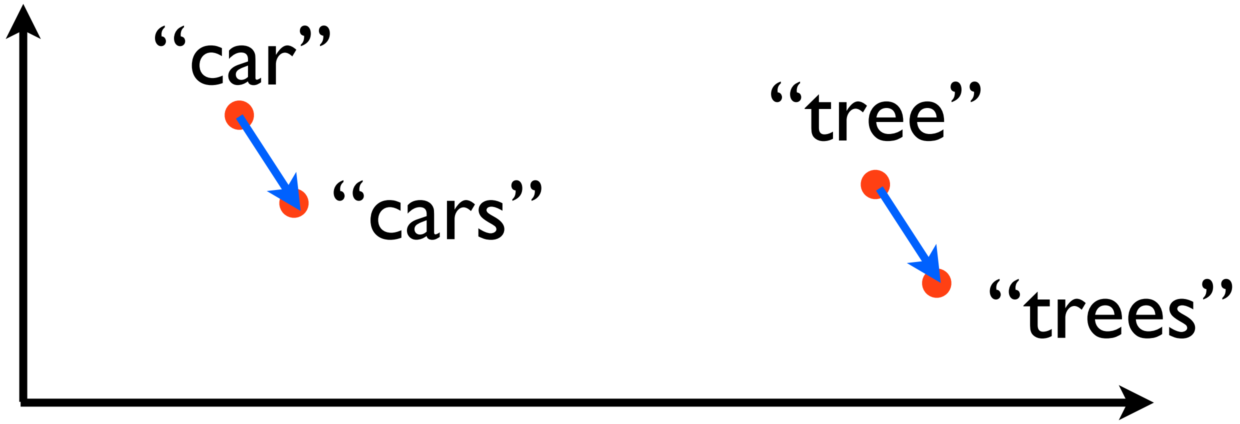
$$C^{(t)} = \ln P_\theta(w_t, h) + \sum_{\tilde{w}} \ln(1 - P_\theta(\tilde{w}, h))$$

noisy examples

Word vectors encode meaning

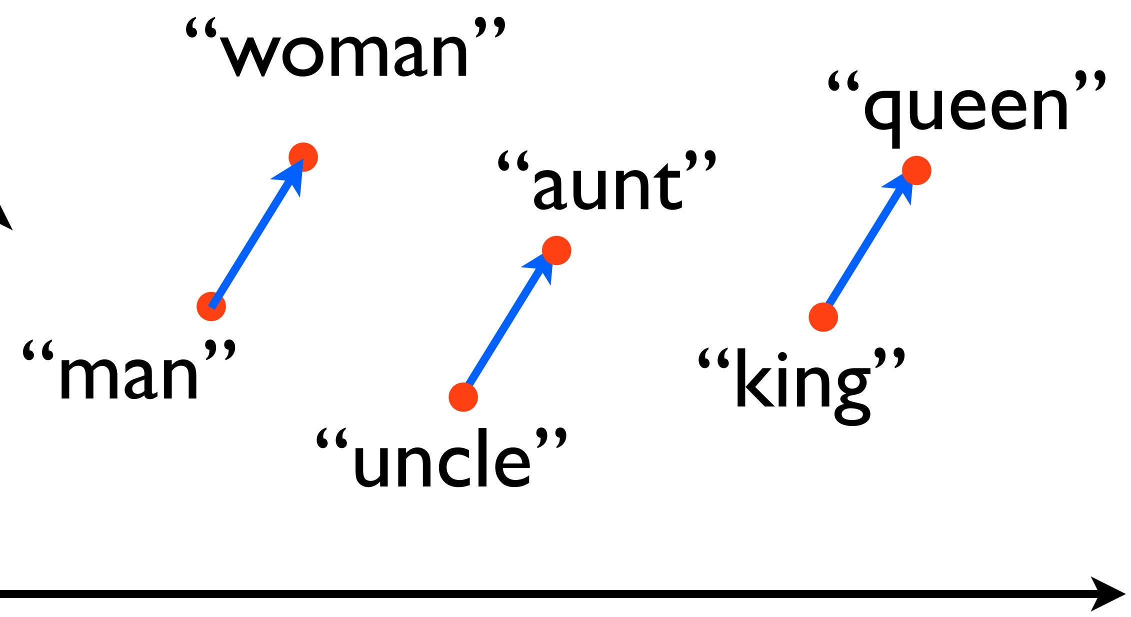
Mikolov, Yih, Zweig 2013

car-cars ~ tree-trees
(subtracting the word vectors on each side yields approx. identical vectors)



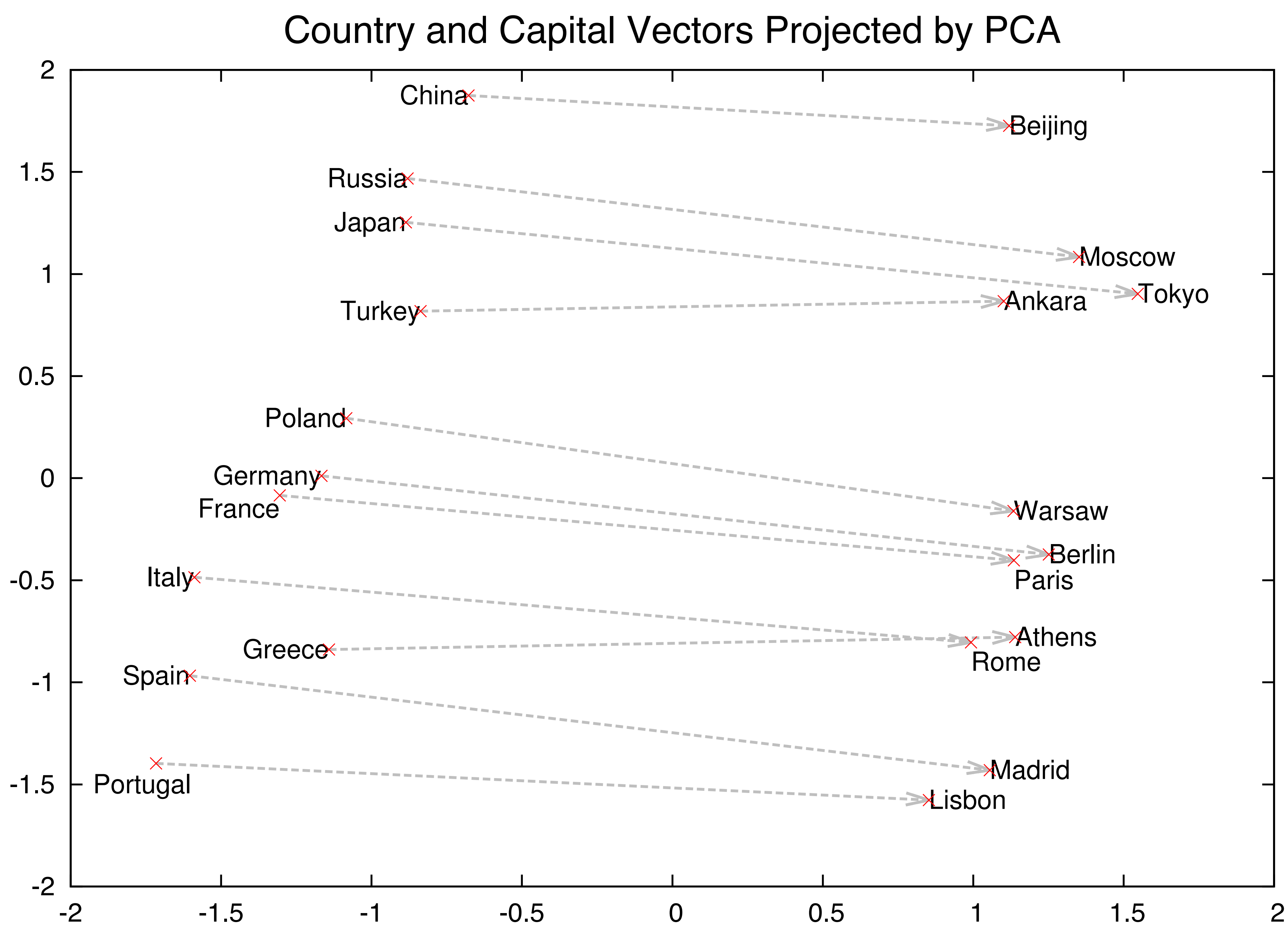
Word vectors encode meaning

Mikolov, Yih, Zweig 2013



Word vectors encode meaning

Mikolov et al. 2013 ”Distributed Representations of Words and Phrases and their Compositionality”



Word vectors in keras

Layer for mapping word indices (integer numbers representing position in a dictionary) to word vectors (of length EMBEDDING_DIM), for input sequences of some given length

```
embedding_layer = Embedding(len(word_index) + 1,
        EMBEDDING_DIM,
        input_length=MAX_SEQUENCE_LENGTH)
```

Helper routines for converting actual text into a sequence of word indices. See especially:

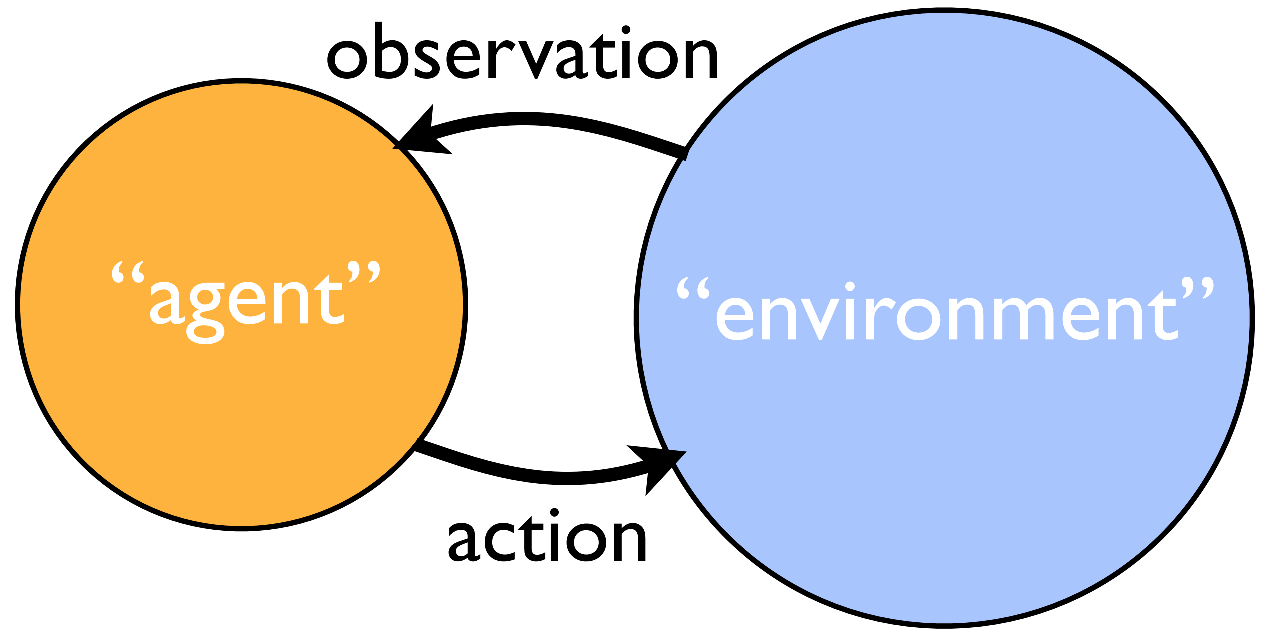
function/class	Keras documentation
`Tokenizer`	Text Preprocessing
`pad_sequences`	Sequence Preprocessing
`(and others)`	

Search for "GloVe word embeddings": 800 MB database pre-trained on a 2014 dump of the English Wikipedia, encoding 400k words in 100-dimensional vectors

Reinforcement Learning

Image: Wikipedia

Reinforcement learning



fully observed vs. partially observed “state” of the environment

Self-driving cars, robotics:
Observe immediate environment & move
Games:
Observe board & place stone
Observe video screen & move player

Challenge: the “correct” action is not known!
Therefore: no supervised learning!

Reward will be rare (or decided only at end)

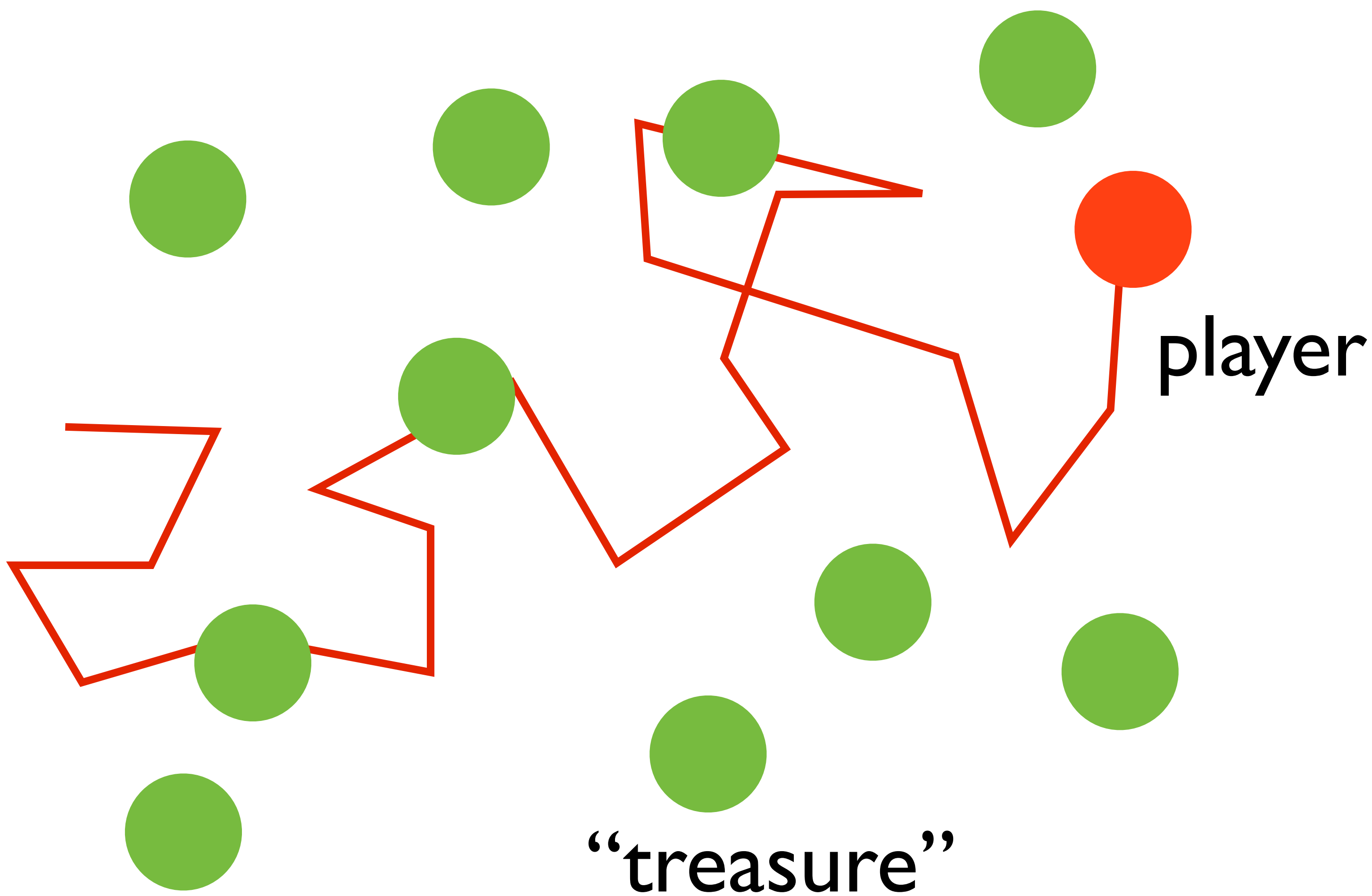
Reinforcement learning

Use **reinforcement learning**:

Training a network to produce actions based on rare rewards (instead of being told the 'correct' action!)

Challenge: We could use the final reward to define a cost function, but we cannot know how the environment reacts to a proposed change of the actions that were taken!

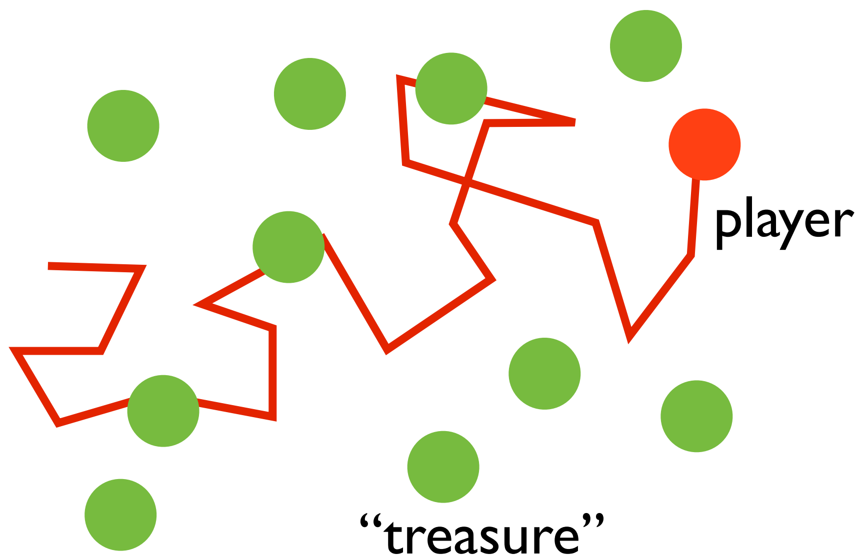
(unless we have a model of the environment)



“State”=full map
“Action”=move
Reward e.g. based on how many “treasures” were collected

Policy Gradient

=REINFORCE (Williams 1992): The simplest model-free general reinforcement learning technique



Basic idea: Use probabilistic action choice. If the reward at the end turns out to be high, make **all** the actions in this sequence **more likely** (otherwise do the opposite)

This will also sometimes reinforce 'bad' actions, but since they occur more likely in trajectories with low reward, the net effect will still be to suppress them!

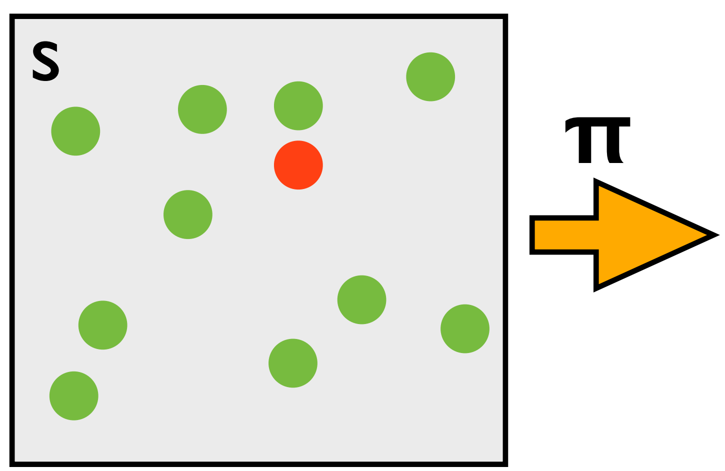
Policy Gradient

Probabilistic policy:

Probability to take action a, given the current state s

$$\pi_\theta(a|s)$$

parameters of the network



a	π
down	0.1
up	0.6
left	0.2
right	0.1

Environment: makes (possibly stochastic) transition to a new state s', and possibly gives a reward r

Transition function $\quad P(s'|s,a)$

Policy Gradient

Probability for having a certain trajectory of actions and states:

product over time steps

$$P_\theta(\tau) = \Pi_t P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

trajectory: $\tau = (\mathbf{a}, \mathbf{s})$
$\mathbf{a} = a_0, a_1, a_2, \ldots$
$\mathbf{s} = s_1, s_2, \ldots$ (state 0 is fixed)

Expected overall reward: sum over all trajectories

$$\bar{R} = E[R] = \sum_\tau P_\theta(\tau) R(\tau)$$

reward for this sequence (sum over individual rewards r for all times)

sum over all actions at all times
and over all states at all times >0

$$\sum_\tau \cdots = \sum_{a_0, a_1, a_2, \ldots, s_1, s_2, \ldots} \cdots$$

Try to maximize expected reward by changing parameters of policy:

$$\frac{\partial \bar{R}}{\partial \theta} = ?$$

Policy Gradient

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t \sum_\tau R(\tau) \underbrace{\frac{\partial \pi_\theta(a_t|s_t)}{\partial \theta} \frac{1}{\pi_\theta(a_t|s_t)}}_{\frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}} \Pi_{t'} P(s_{t'+1}|s_{t'}, a_{t'}) \pi_\theta(a_{t'}|s_{t'})$$

Main formula of policy gradient method:

$$\boxed{\frac{\partial \bar{R}}{\partial \theta} = \sum_t E[R \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]}$$

Stochastic gradient descent:

$$\Delta\theta = \eta \frac{\partial \bar{R}}{\partial \theta}$$

where $E[\ldots]$ is approximated via the value for one trajectory (or a batch)

Policy Gradient

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t E[R \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]$$

Increase the probability of all action choices in the given sequence, depending on size of reward R.
Even if R>0 always, due to normalization of probabilities this will tend to suppress the action choices in sequences with lower-than-average rewards.

Abbreviation:

$$G_k = \frac{\partial \ln P_\theta(\tau)}{\partial \theta_k} = \sum_t \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta_k}$$

$$\frac{\partial \bar{R}}{\partial \theta_k} = E[R G_k]$$

Policy Gradient: reward baseline

Challenge: fluctuations of estimate for reward gradient can be huge. Things improve if one subtracts a constant baseline from the reward.

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t E[(R - b)\frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]$$
$$= E[(R-b)G]$$

This is the same as before. Proof:

$$E[G_k] = \sum_\tau P_\theta(\tau)\frac{\partial \ln P_\theta(\tau)}{\partial \theta_k} = \frac{\partial}{\partial \theta_k}\sum_\tau P_\theta(\tau) = 0$$

However, the variance of the fluctuating random variable (R-b)G is different, and can be smaller (depending on the value of b)!

Optimal baseline

$$X_k = (R - b_k)G_k$$

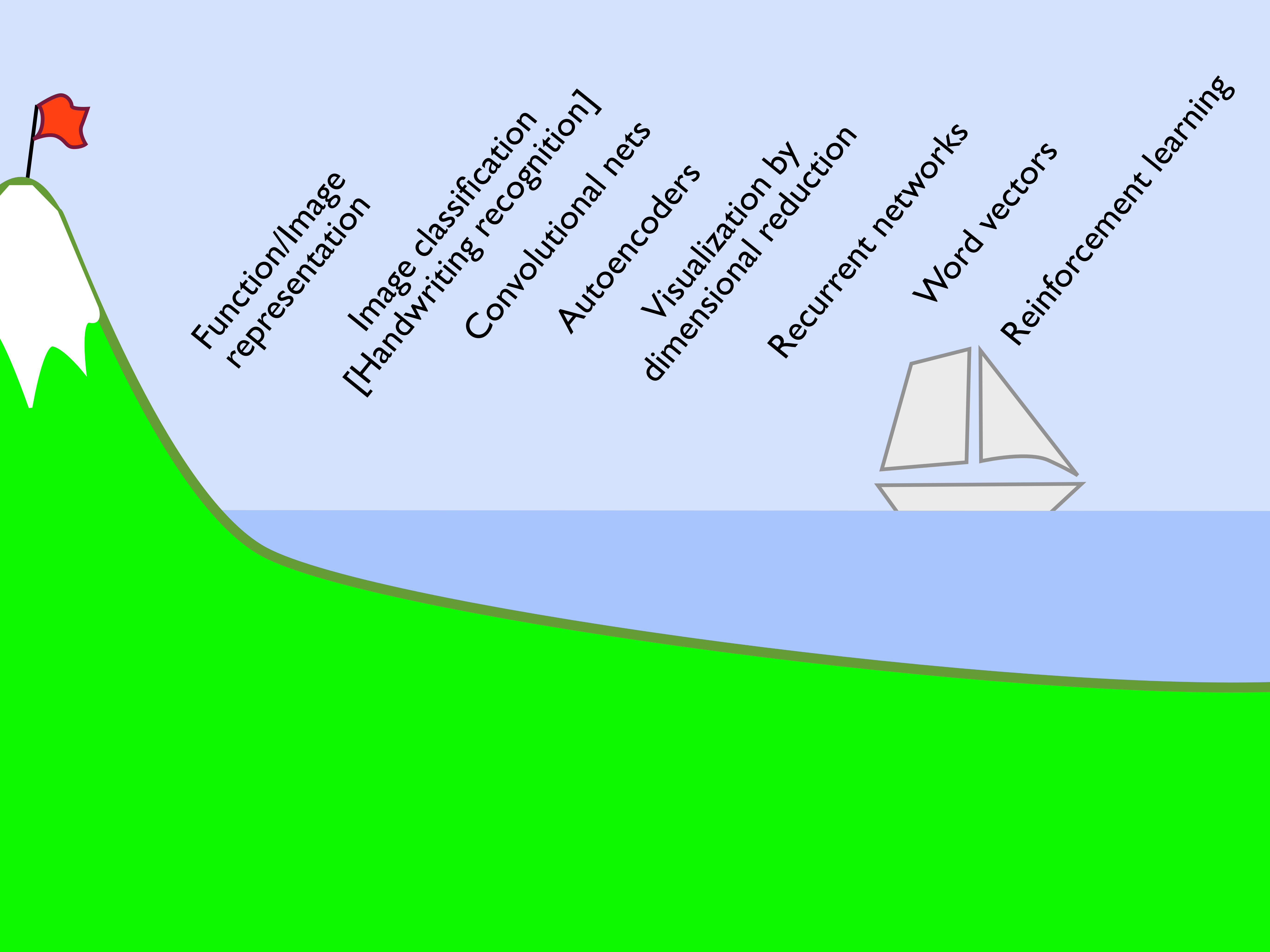
$$\mathrm{Var}[X_k] = E[X_k^2] - E[X_k]^2 = \min$$

$$\frac{\partial \mathrm{Var}[X_k]}{\partial b_k} = 0$$

$$b_k = \frac{E[G_k^2 R]}{E[G_k^2]}$$

$$G_k = \frac{\partial \ln P_\theta(\tau)}{\partial \theta_k}$$

$$\Delta\theta_k = -\eta E[G_k(R - b_k)]$$


Function/Image representation
Image classification [Handwriting recognition]
Convolutional nets
Autoencoders
Visualization by dimensional reduction
Recurrent networks
Word vectors
Reinforcement learning

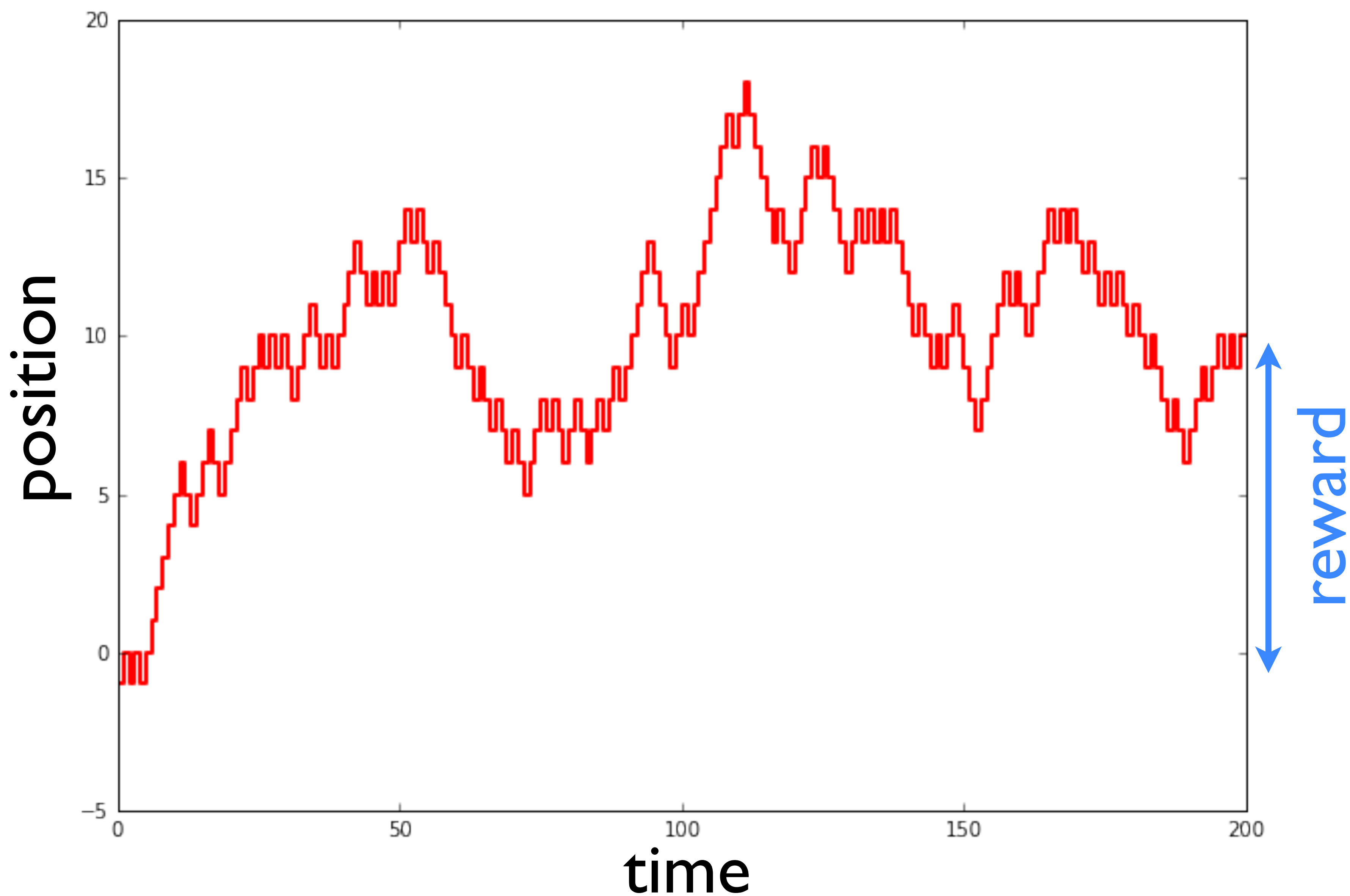
For more in-depth treatment, see David Silver's course on reinforcement learning (University College London):

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

The simplest RL example ever

A random walk, where the probability to go "up" is determined by the policy, and where the reward is given by the final position (ideal strategy: always go up!)

(Note: this policy does not even depend on the current state)



The simplest RL example ever

A random walk, where the probability to go "up" is determined by the policy, and where the reward is given by the final position (ideal strategy: always go up!)

(Note: this policy does not even depend on the current state)

policy $\pi_\theta(\mathrm{up}) = \frac{1}{1+e^{-\theta}}$ **reward** $R = x(T)$

RL update $\Delta\theta = \eta \sum_t \left\langle R \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle$

$a_t = \mathrm{up\,or\,down}$

$$\frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} = \underbrace{\pm}_{\text{+ for up, - for down}} e^{-\theta}\pi_\theta(a_t) = \pm(1-\pi_\theta(a_t)) = \begin{cases} 1-\pi_\theta(\mathrm{up}) & \text{for up} \\ -\pi_\theta(\mathrm{up}) & \text{for down} \end{cases}$$

$$\sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} = \overset{\text{number of 'up-steps'}}{N_{\mathrm{up}}} - N\pi_\theta(\mathrm{up})$$

N=number of time steps

The simplest RL example ever

reward $$R = x(T) = N_{\rm up} - N_{\rm down} = 2N_{\rm up} - N$$

RL update $$\Delta\theta = \eta \sum_t \left\langle R \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle$$

$$a_t = \text{up or down}$$

$$\left\langle R \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle = 2 \left\langle (N_{\rm up} - \frac{N}{2})(N_{\rm up} - \bar{N}_{\rm up}) \right\rangle$$

(general analytical expression for average update, rare)

Initially, when $\pi_\theta(\text{up}) = \frac{1}{2}$:

$$\Delta\theta = 2\eta \left\langle (N_{\rm up} - \frac{N}{2})^2 \right\rangle = 2\eta \mathrm{Var}(N_{\rm up}) = \eta \frac{N}{2} > 0$$

(binomial distribution!)

The simplest RL example ever

In general:

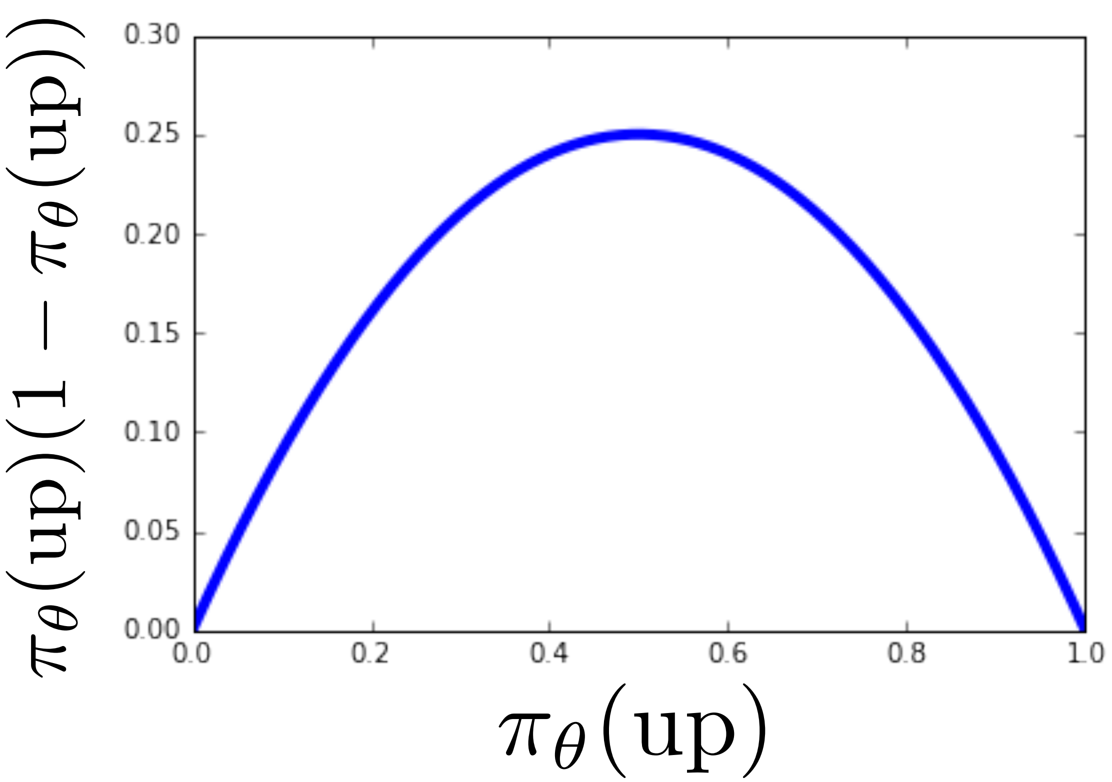
$$\left\langle R \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle = 2\left\langle (N_{\mathrm{up}} - \frac{N}{2})(N_{\mathrm{up}} - \bar{N}_{\mathrm{up}}) \right\rangle$$

$$= 2\left\langle \left((N_{\mathrm{up}} - \bar{N}_{\mathrm{up}}) + (\bar{N}_{\mathrm{up}} - \frac{N}{2}) \right) (N_{\mathrm{up}} - \bar{N}_{\mathrm{up}}) \right\rangle$$

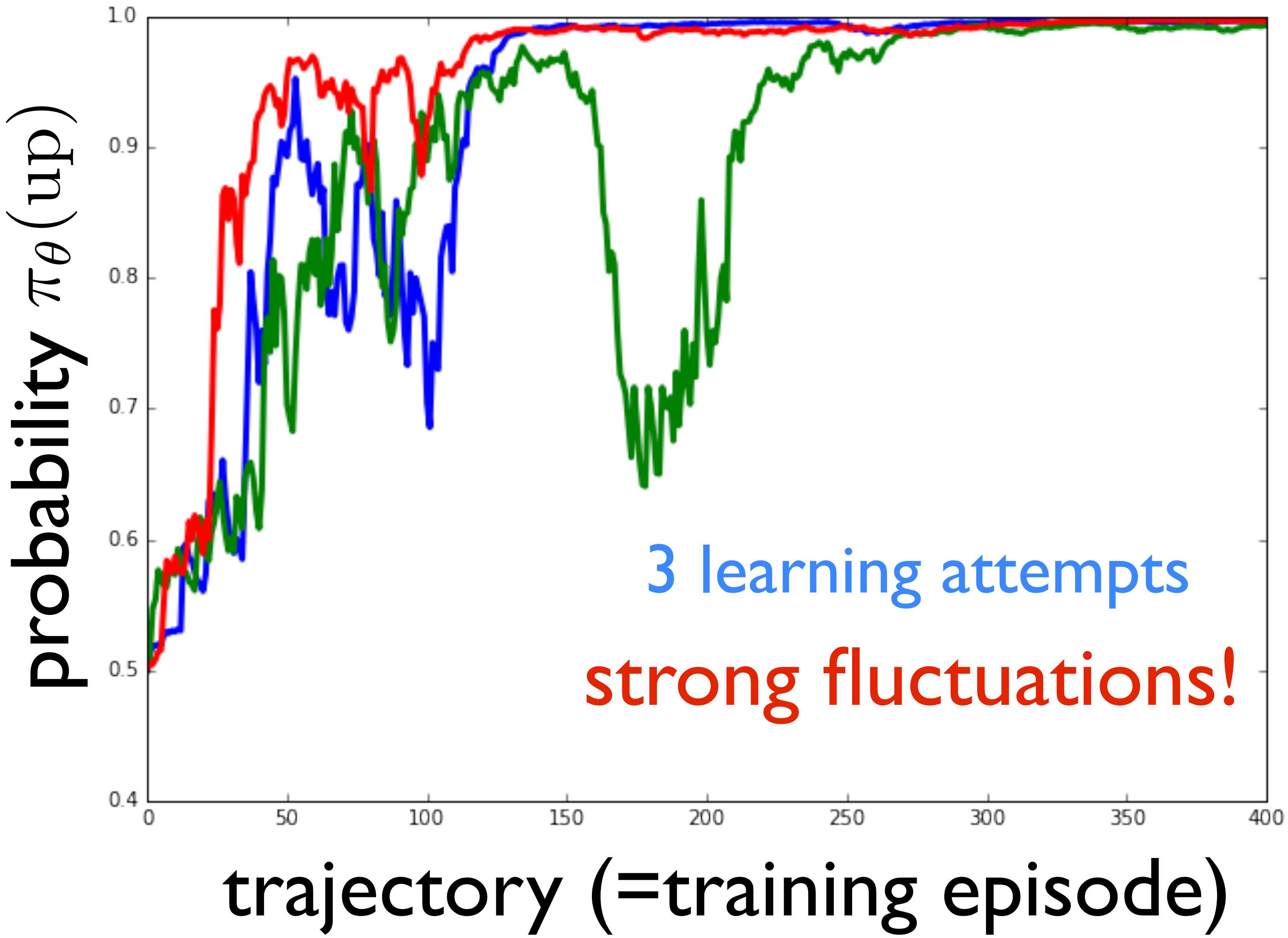
$$= 2\mathrm{Var} N_{\mathrm{up}} + 2(\bar{N}_{\mathrm{up}} - \frac{N}{2}) \left\langle N_{\mathrm{up}} - \bar{N}_{\mathrm{up}} \right\rangle$$

$$= 2\mathrm{Var} N_{\mathrm{up}} = 2N\pi_\theta(\mathrm{up})(1-\pi_\theta(\mathrm{up}))$$

(general analytical expression for average update, fully simplified, extremely rare)



The simplest RL example ever



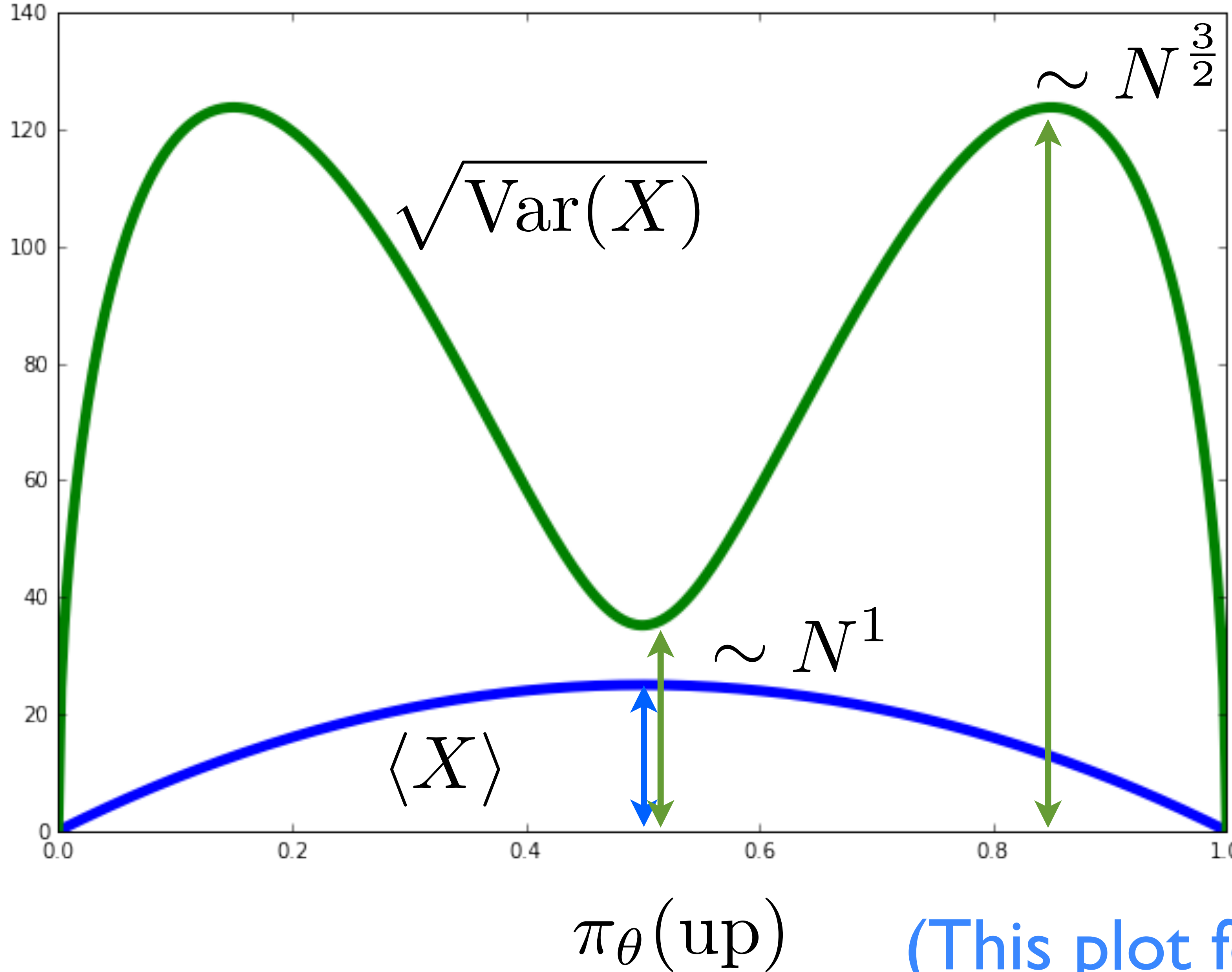
(This plot for N=100 time steps in a trajectory; eta=0.001)

Spread of the update step

$$Y = N_{\mathrm{up}} - \bar{N}_{\mathrm{up}} \qquad c = \bar{N}_{\mathrm{up}} - N/2 \qquad X = (Y+c)Y$$

(Note: to get Var X, we need central moments of binomial distribution up to 4th moment)

X=update (except prefactor of 2)



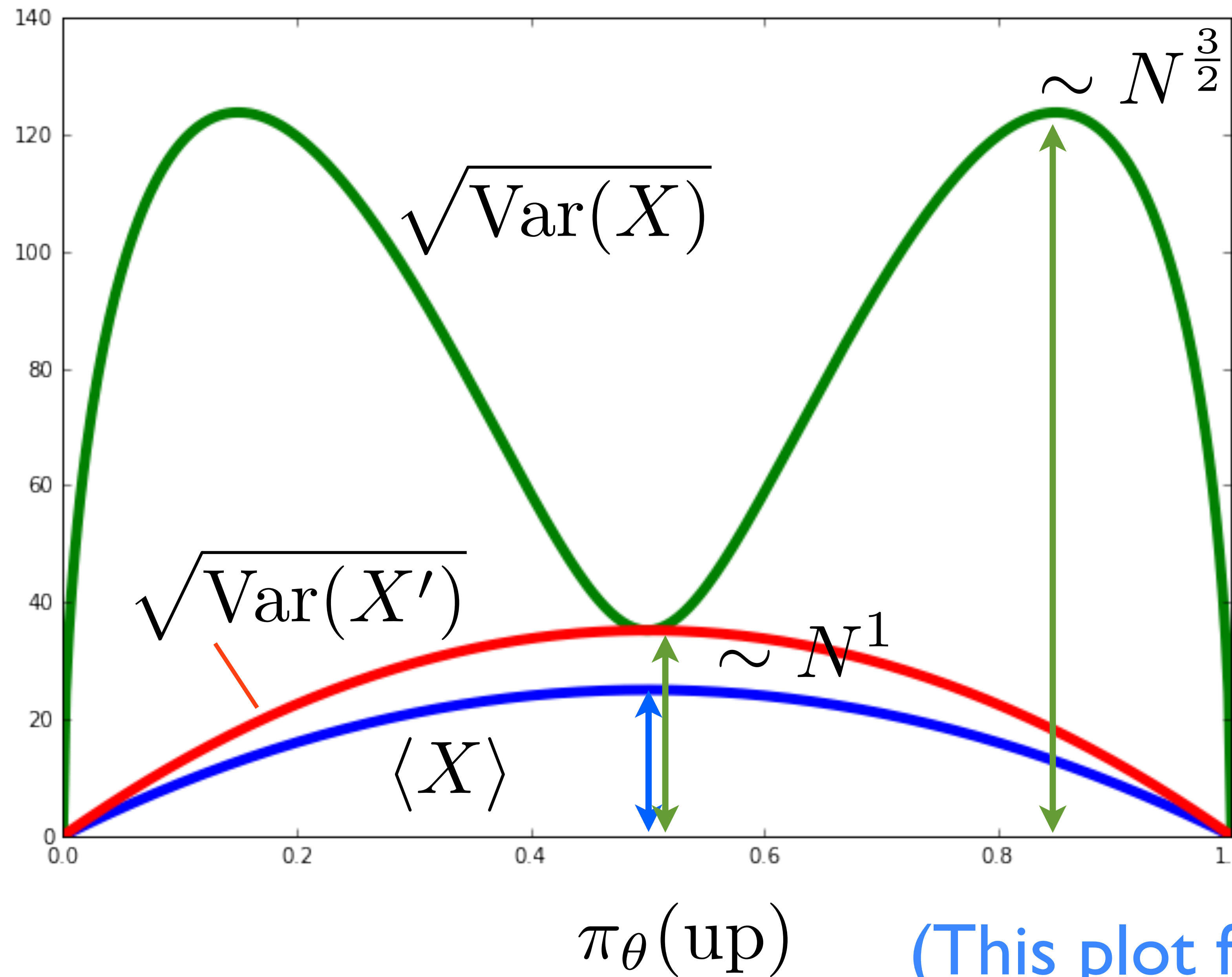
(This plot for N=100)

Optimal baseline suppresses spread!

$$Y = N_{\rm up} - \bar{N}_{\rm up} \qquad c = \bar{N}_{\rm up} - N/2 \qquad X = (Y+c)Y$$

with optimal baseline:

$$X' = (Y + c - b)Y \qquad b = \frac{\langle Y^2(Y+c)\rangle}{\langle Y^2\rangle}$$



(This plot for N=100)

Note: Many update steps reduce relative spread

M = number of update steps

$$\Delta X = \sum_{j=1}^{M} X_j$$

$$\langle \Delta X \rangle = M \langle X \rangle$$

$$\sqrt{\mathrm{Var}\Delta X} = \sqrt{M}\sqrt{\mathrm{Var}X}$$

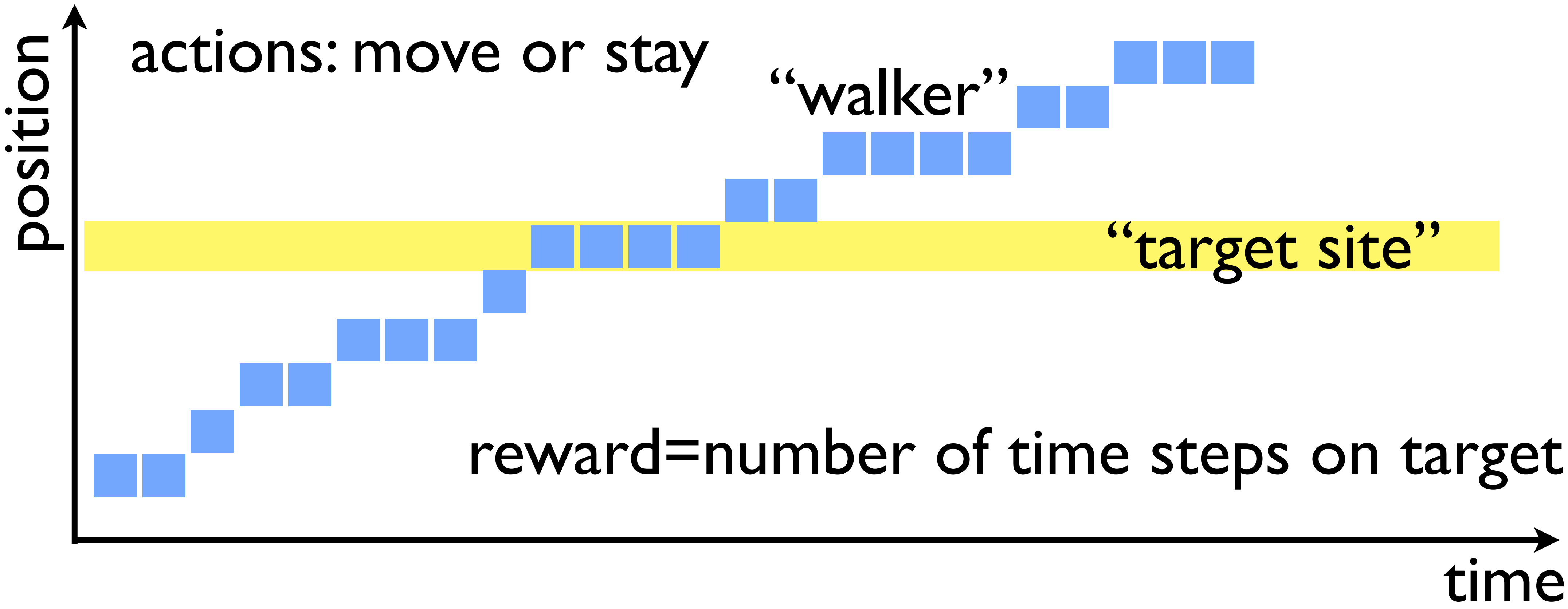
relative spread $$\frac{\sqrt{\mathrm{Var}\Delta X}}{\langle \Delta X \rangle} \sim \frac{1}{\sqrt{M}}$$

Homework

Implement the RL update including the optimal baseline and run some stochastic learning attempts. Can you observe the improvement over the no-baseline results shown here?

Note: You do not need to simulate the individual random walk trajectories, just exploit the binomial distribution.

The second-simplest RL example

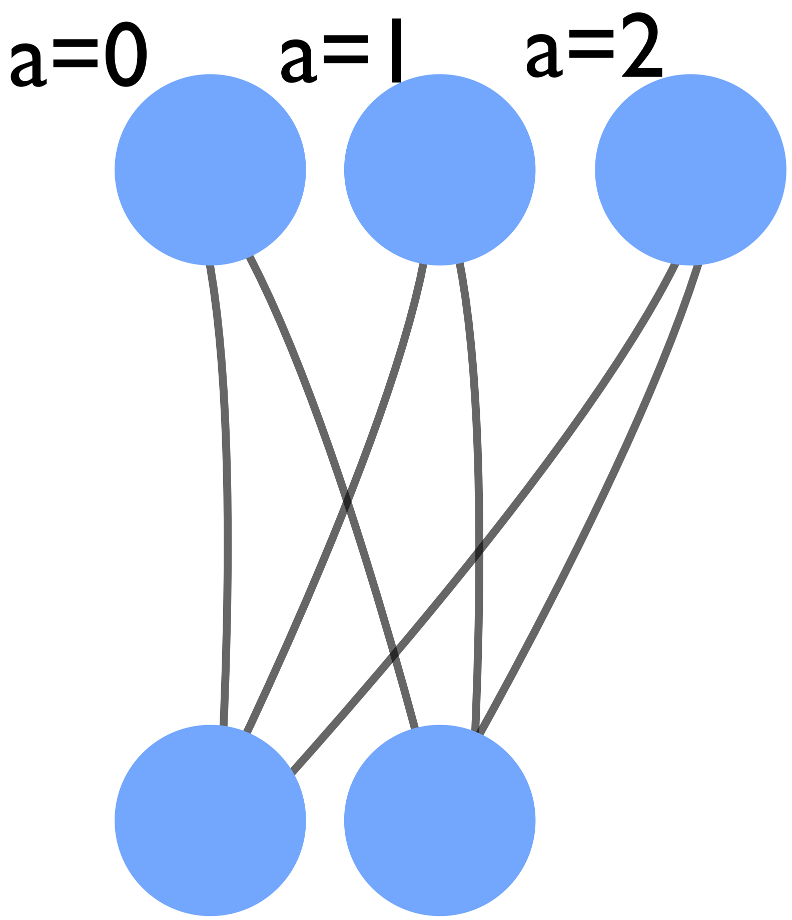


See code on website: "SimpleRL_WalkerTarget"

RL in keras: categorical cross-entropy trick

output = action probabilities (softmax)

$\pi_\theta(a|s)$



input = state

categorical cross-entropy

$$C = -\sum_a P(a) \ln \pi_\theta(a|s)$$

distr. from net

desired distribution

Set

$$P(a) = R$$

for a=action that was taken

$$P(a) = 0$$

for all other actions a

$$\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$$

implements policy gradient

“alpha-Go”

Among the major board games, “Go” was not yet played on a superhuman level by any program (very large state space on a 19x19 board!)

alpha-Go beat the world’s best player in 2017

"alpha-Go"

First: try to learn from human expert players

sampled state-action pairs (s, a), using stochastic gradient ascent to maximize the likelihood of the human move a selected in state s

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial\sigma}$$

We trained a 13-layer policy network, which we call the SL policy network, from 30 million positions from the KGS Go Server. The net-

Silver et al., "Mastering the game of Go with deep neural networks and tree search" (Google Deepmind team), Nature, January 2016

"alpha-Go"

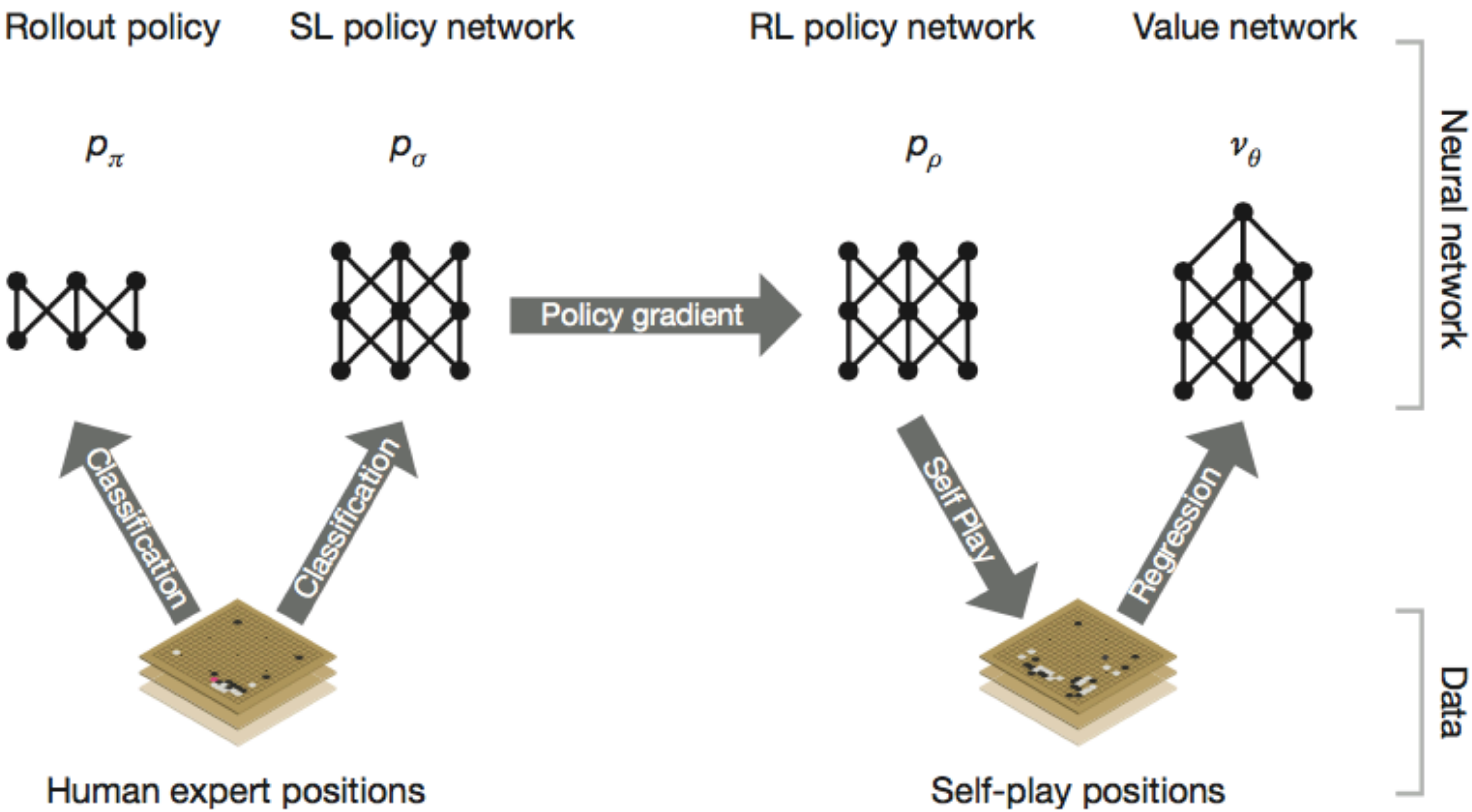
Second: use policy gradient RL on games played against previous versions of the program

to the current policy. We use a reward function $r(s)$ that is zero for all non-terminal time steps $t < T$. The outcome $z_t = \pm r(s_T)$ is the terminal reward at the end of the game from the perspective of the current player at time step t: $+1$ for winning and -1 for losing. Weights are then updated at each time step t by stochastic gradient ascent in the direction that maximizes expected outcome[25]

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

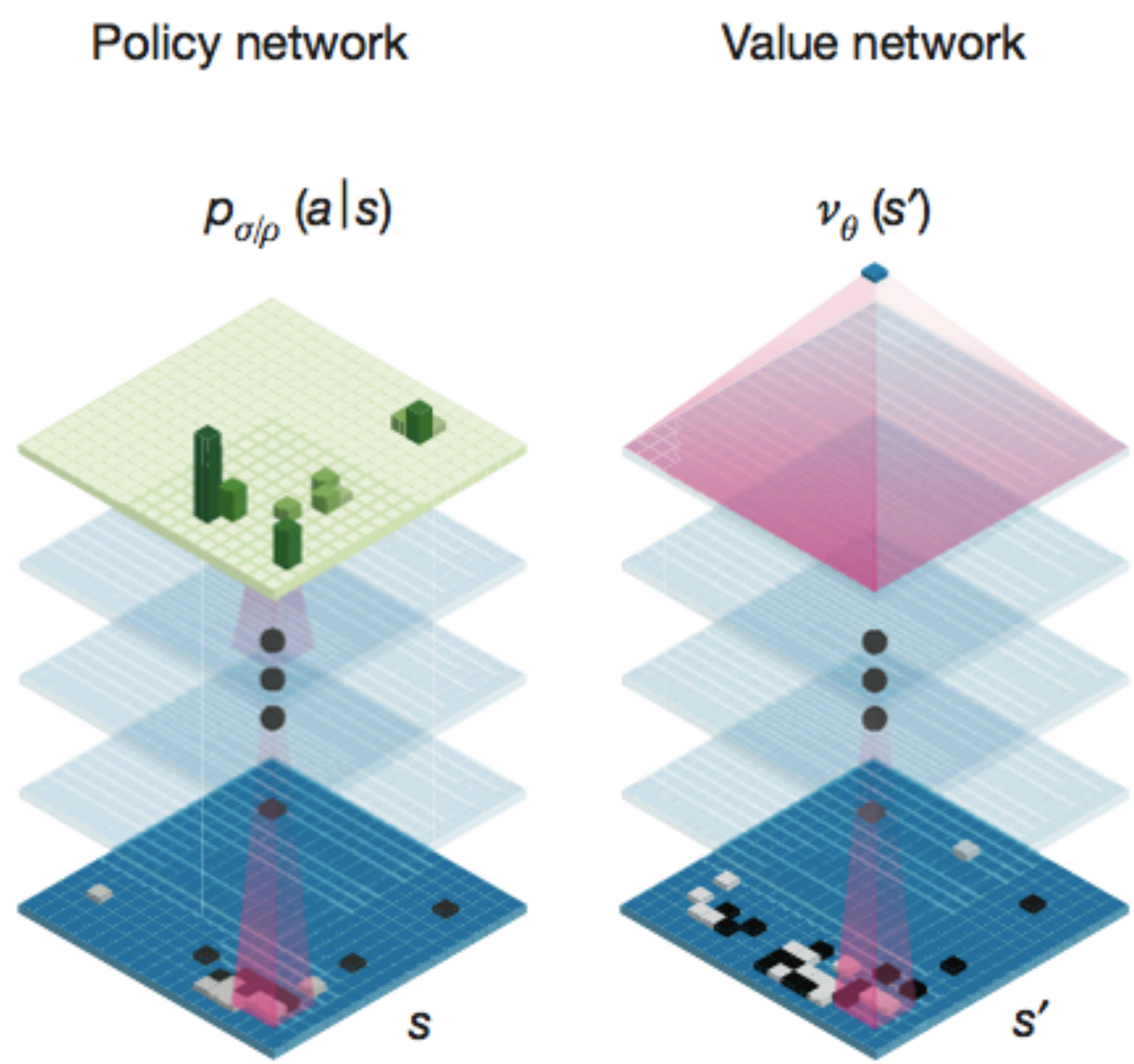
Silver et al., "Mastering the game of Go with deep neural networks and tree search" (Google Deepmind team), Nature, January 2016

“alpha-Go”



Silver et al.,“Mastering the game of Go with deep neural networks and tree search” (Google Deepmind team), Nature, January 2016

"alpha-Go"

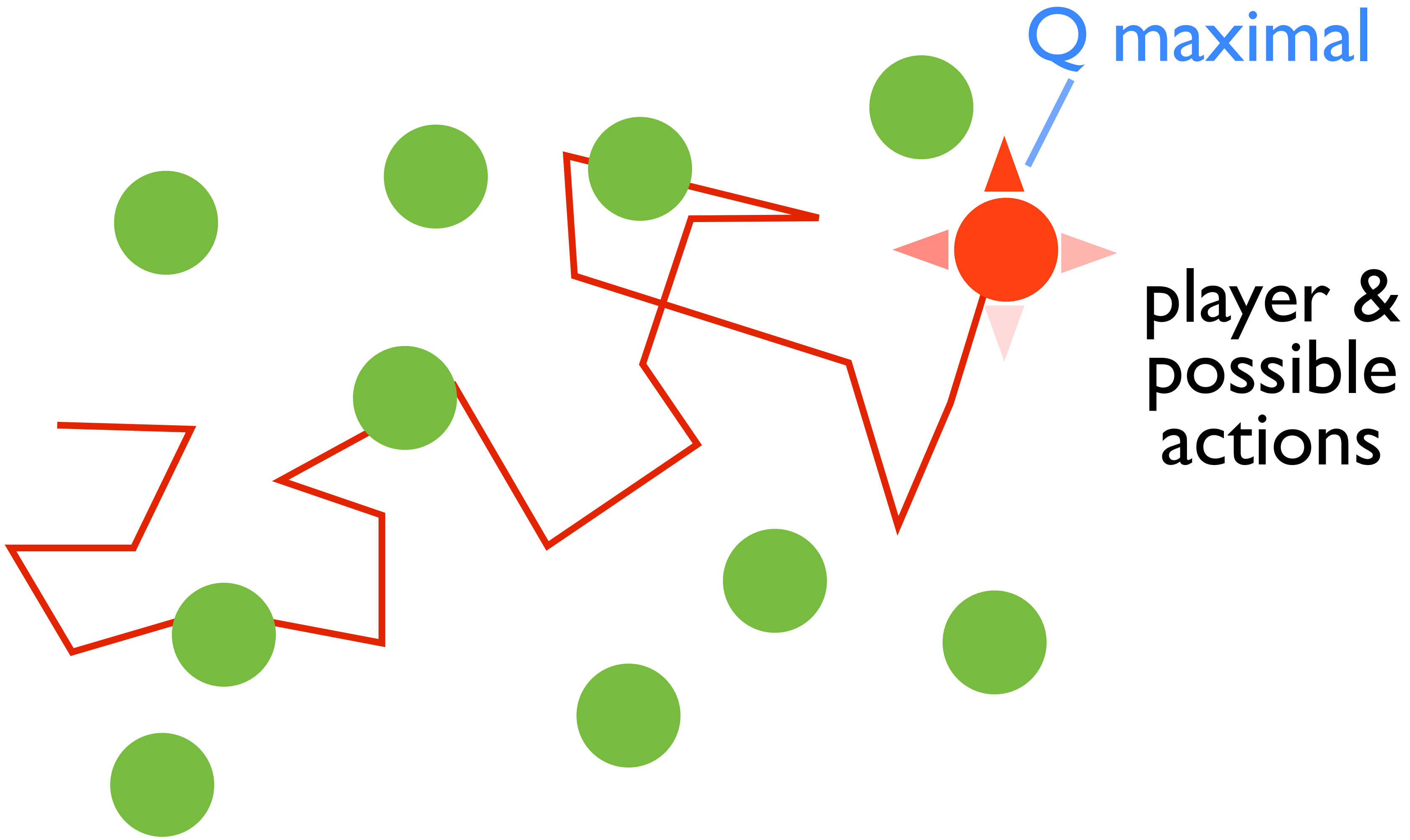


Silver et al., "Mastering the game of Go with deep neural networks and tree search" (Google Deepmind team), Nature, January 2016

Q-learning

An alternative to the policy gradient approach

Introduce a quality function Q that predicts the future reward for a given state s and a given action a. **Deterministic policy**: just select the action with the largest Q!


Q maximal
player &
possible
actions

Q-learning

Introduce a quality function Q that predicts the future reward for a given state s and a given action a. **Deterministic policy**: just select the action with the largest Q!

$$Q(s_t, a_t) = E[R_t | s_t, a_t]$$

(assuming future steps to follow the policy!)

"Discounted" future reward: $$R_t = \sum_{t'=t}^{T} r_{t'} \gamma^{t'-t}$$

Reward at time step t: r_t

depends on state and action at time t

Discount factor: $0 < \gamma \leq 1$

learning somewhat easier for smaller factor (short memory times)

Note: The 'value' of a state is $V(s) = \max_a Q(s, a)$

How do we obtain Q?

Q-learning: Update rule

Bellmann equation: (from optimal control theory)

$$Q(s_t, a_t) = E[r_t + \gamma \mathrm{max}_a Q(s_{t+1}, a) | s_t, a_t]$$

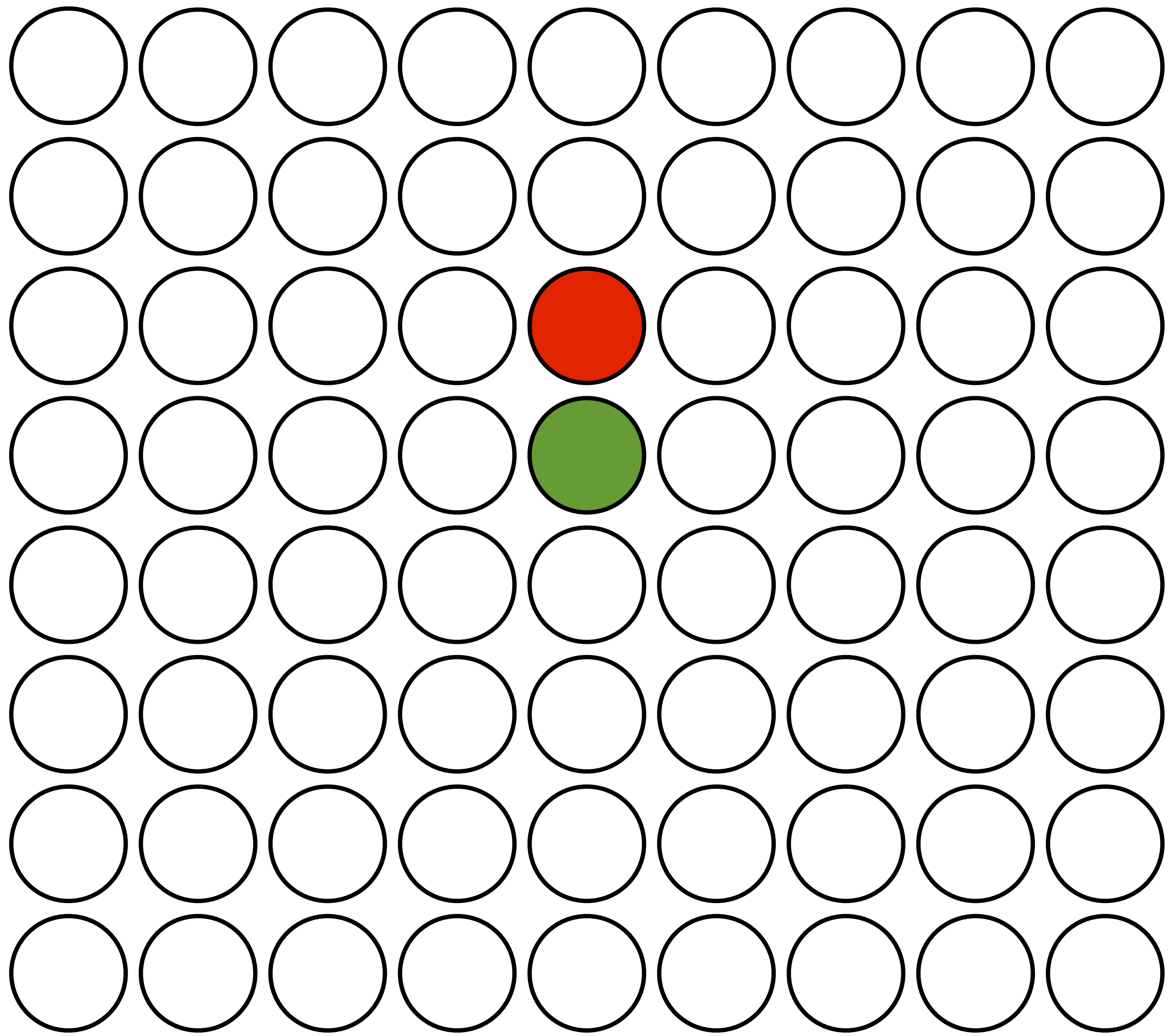
In practice, we do not know the Q function yet, so we cannot directly use the Bellmann equation. However, the following update rule has the correct Q function as a fixed point:

$$Q^{\mathrm{new}}(s_t, a_t) = Q^{\mathrm{old}}(s_t, a_t) + \alpha(r_t + \gamma \mathrm{max}_a Q^{\mathrm{old}}(s_{t+1}, a) - Q^{\mathrm{old}}(s_t, a_t))$$

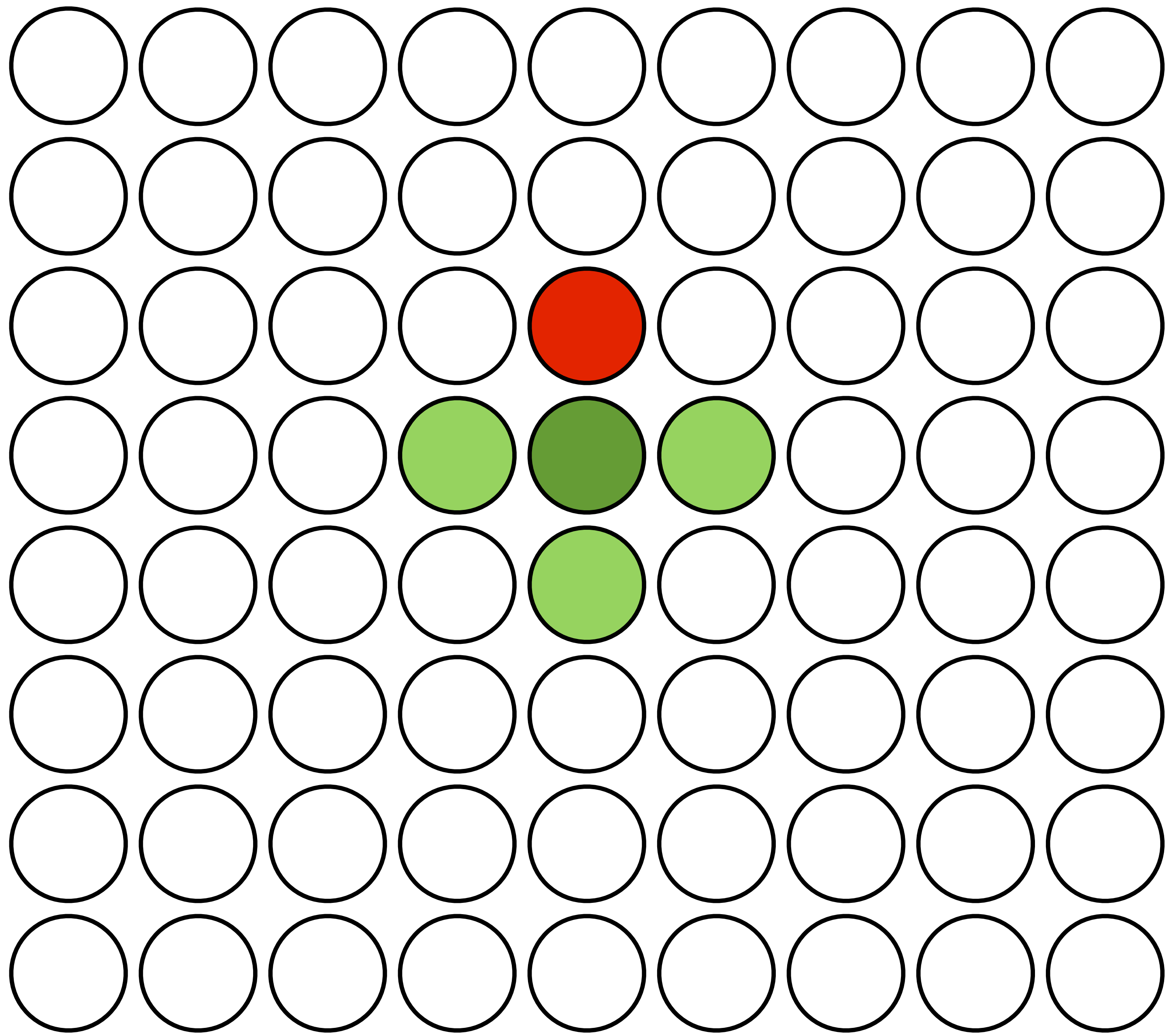
small (<1) update factor

will be zero, once we have converged to the correct Q

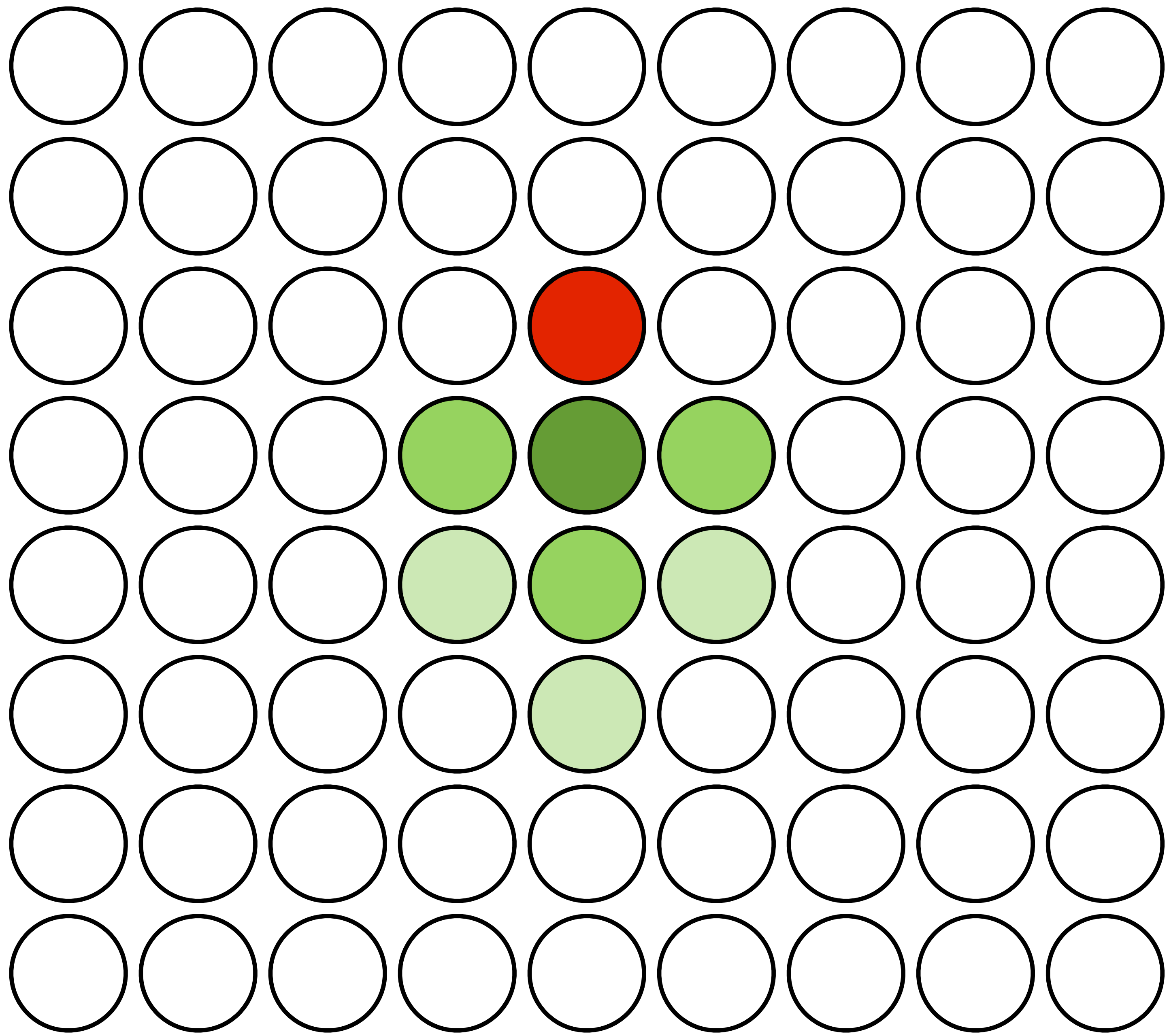
If we use a neural network to calculate Q, it will be trained to yield the "new" value in each step.



Q(a=up,s)



Q(a=up,s)



Q(a=up,s)

Q-learning: Exploration

Initially, Q is arbitrary. It will be bad to follow this Q all the time. Therefore, introduce probability ϵ of random action ("exploration")!

Follow Q: "**exploitation**"

Do something random (new): "**exploration**"

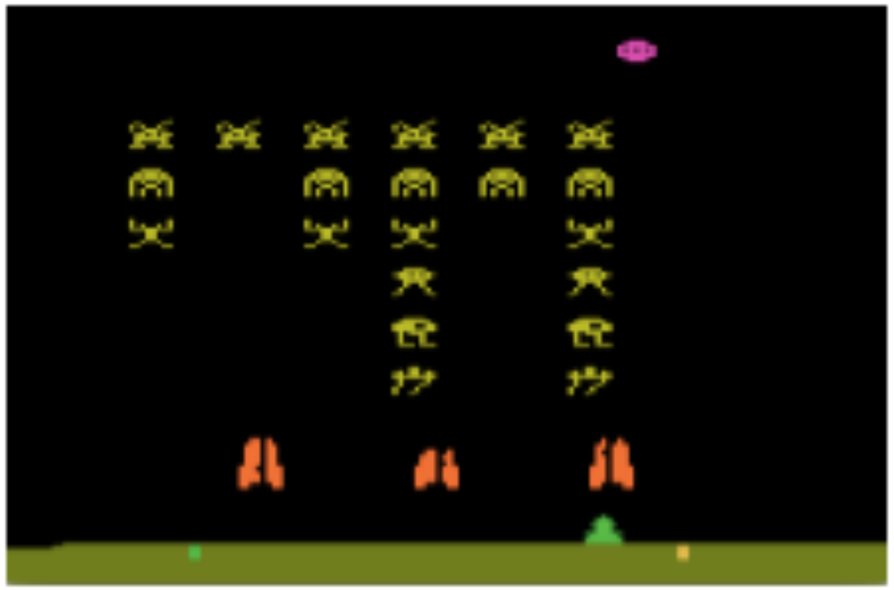
"ϵ-greedy"

Reduce this randomness later!

Example: Learning to play Atari Video Games

"Human-level control through deep reinforcement learning", Mnih et al., Nature, February 2015



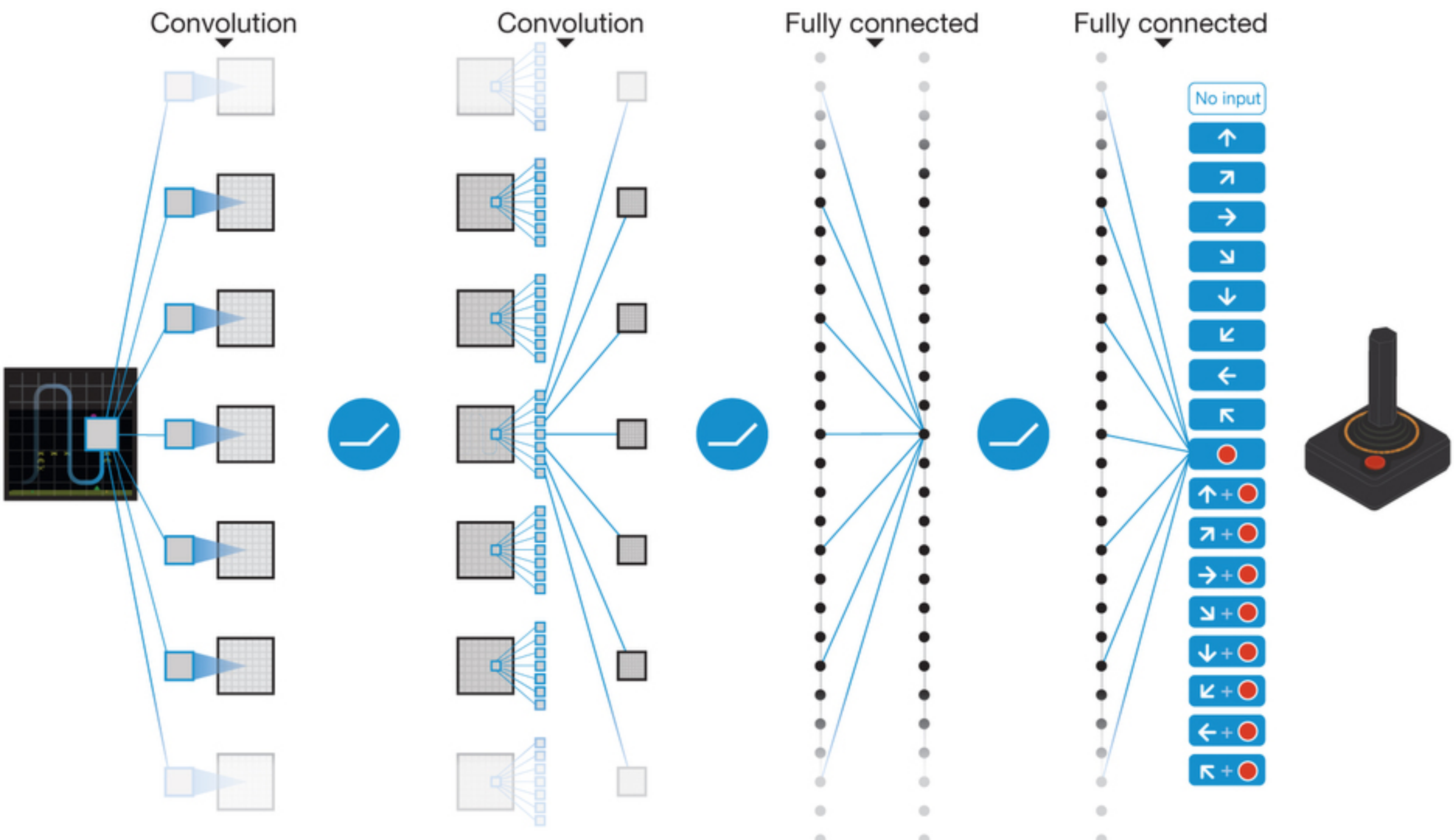



last four 84x84 pixel images as input [=state]

motion as output [=action]

Example: Learning to play Atari Video Games

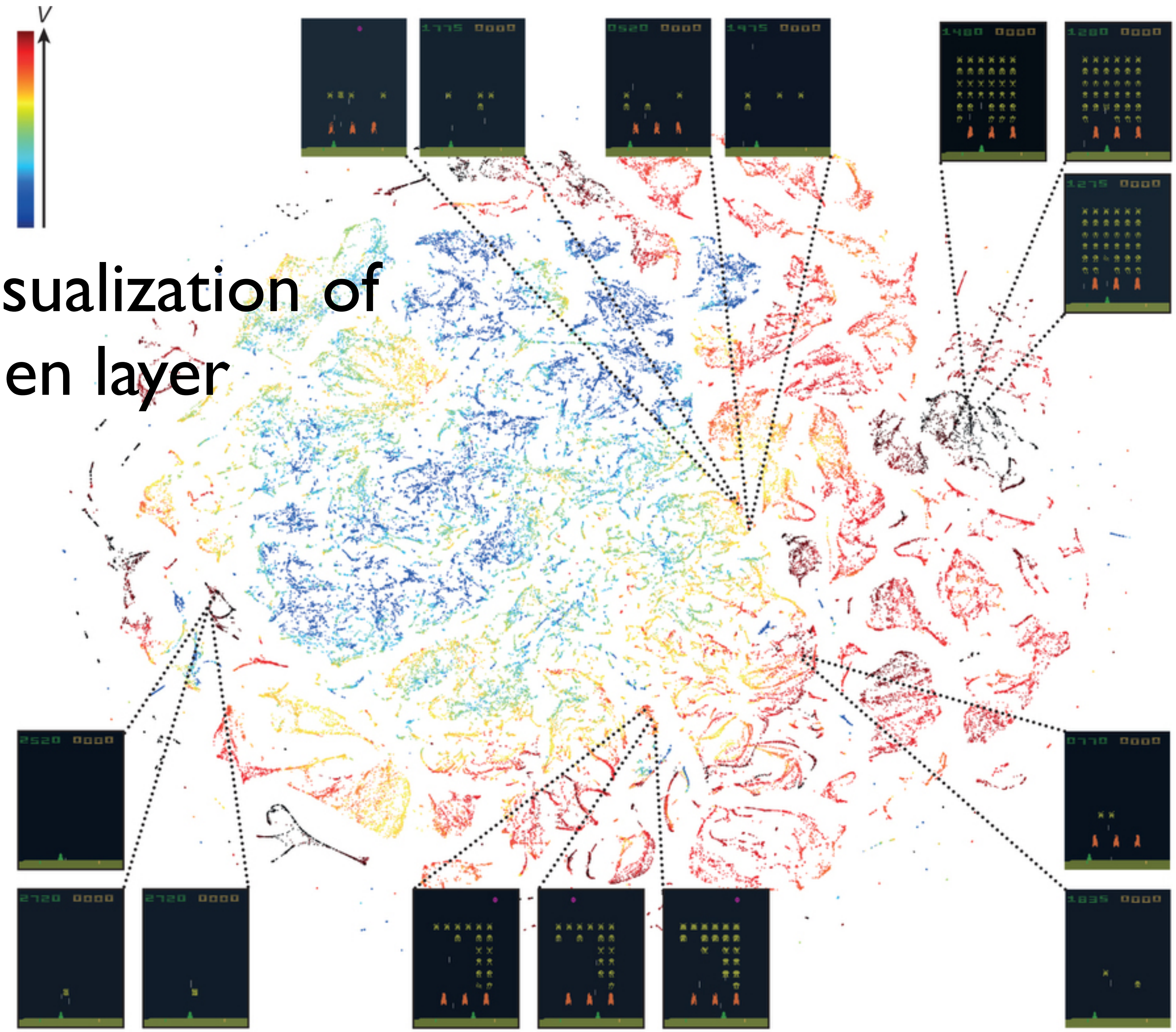
"Human-level control through deep reinforcement learning", Mnih et al., Nature, February 2015

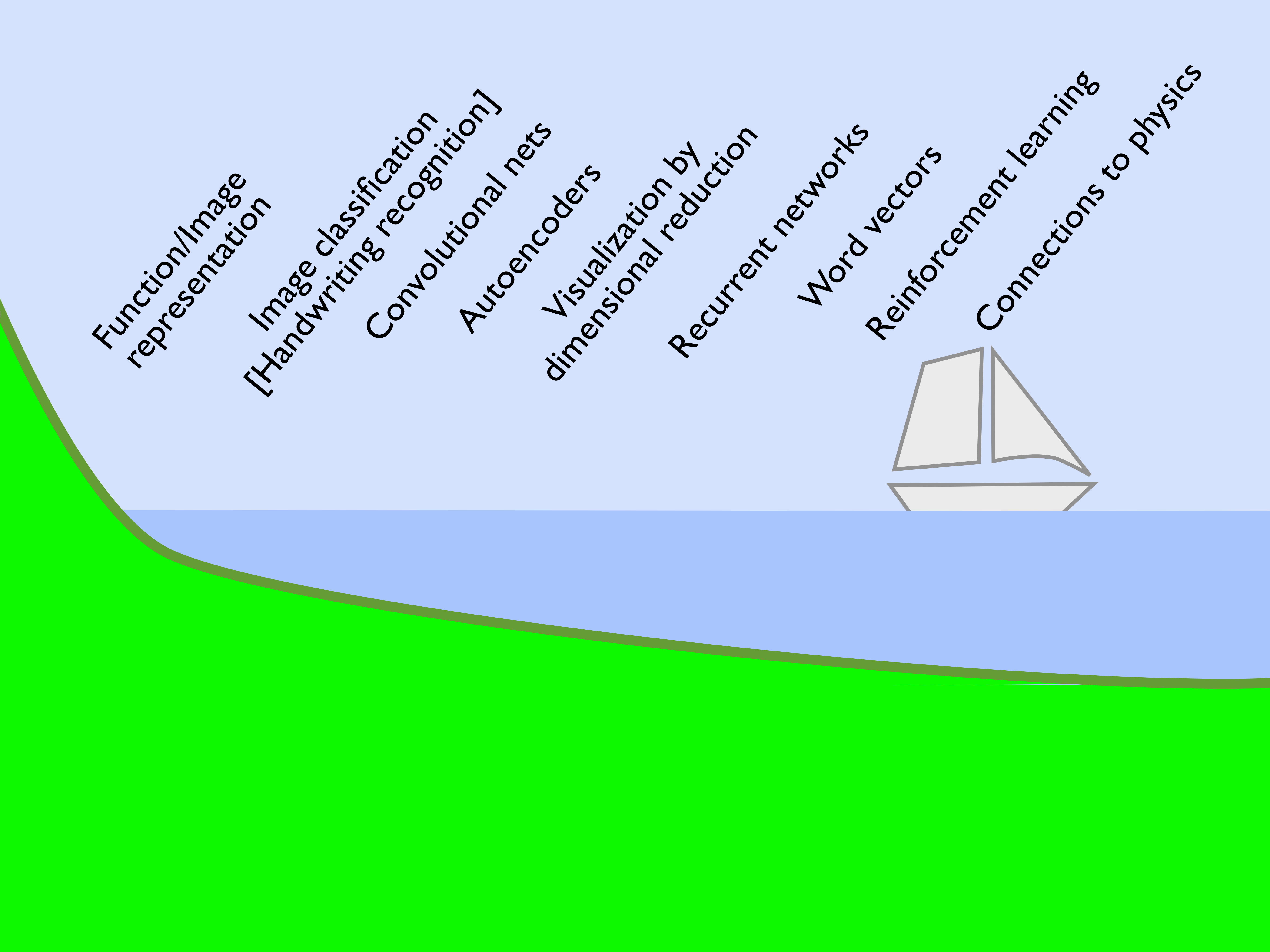


Example: Learning to play Atari Video Games

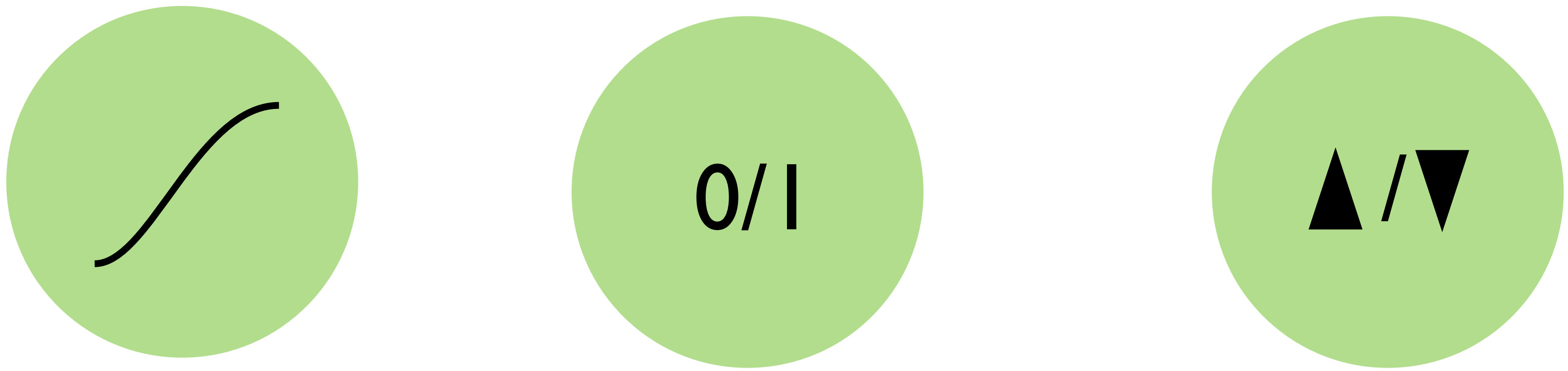
"Human-level control through deep reinforcement learning", Mnih et al., Nature, February 2015

t-SNE visualization of last hidden layer




Function/Image representation
Image classification [Handwriting recognition]
Convolutional nets
Autoencoders
Visualization by dimensional reduction
Recurrent networks
Word vectors
Reinforcement learning
Connections to physics

Neural networks and spin models



Artifical neuron

Bit

Spin

Neural networks with stochastic transitions, and with some energy functional similar to spin models in physics; e.g. as described by Hopfield and others starting from the 80s

Modeling probability distributions

Goal: Use a neural network to generate previously unseen examples, according to the probability distribution of training samples

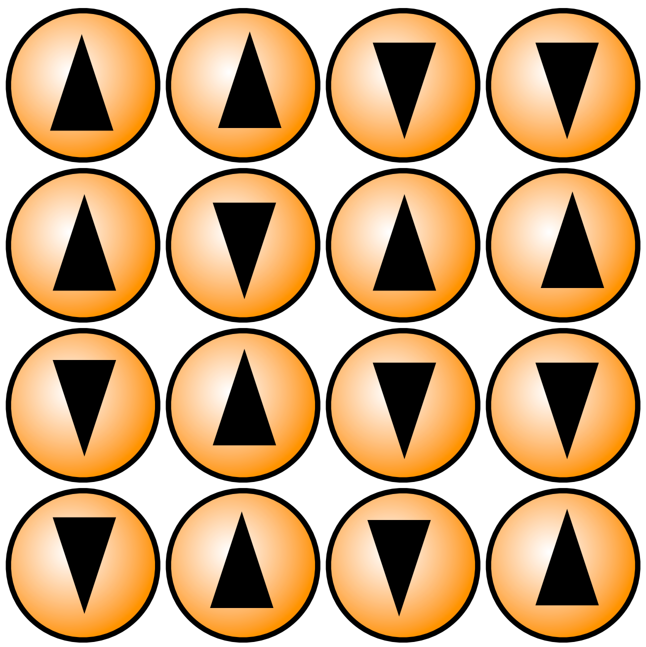
One example already mentioned in these lectures: generating new random (but kind-of reasonable) text after seeing lots of it

Example: Generate new images after looking at many, generate handwritten text

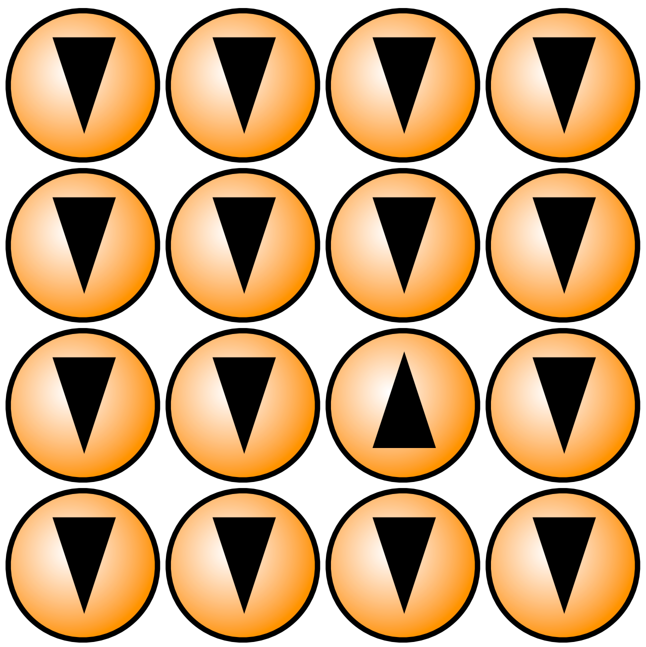
The solution will exploit the connection between neural networks and the statistical physics of spin models!

Boltzmann-Gibbs distribution

Probabilities of states of a physical system,
in thermal equilibrium?



energy high: less likely



energy low: more likely

$$P(s) = \frac{1}{Z} e^{-\frac{E(s)}{k_B T}}$$

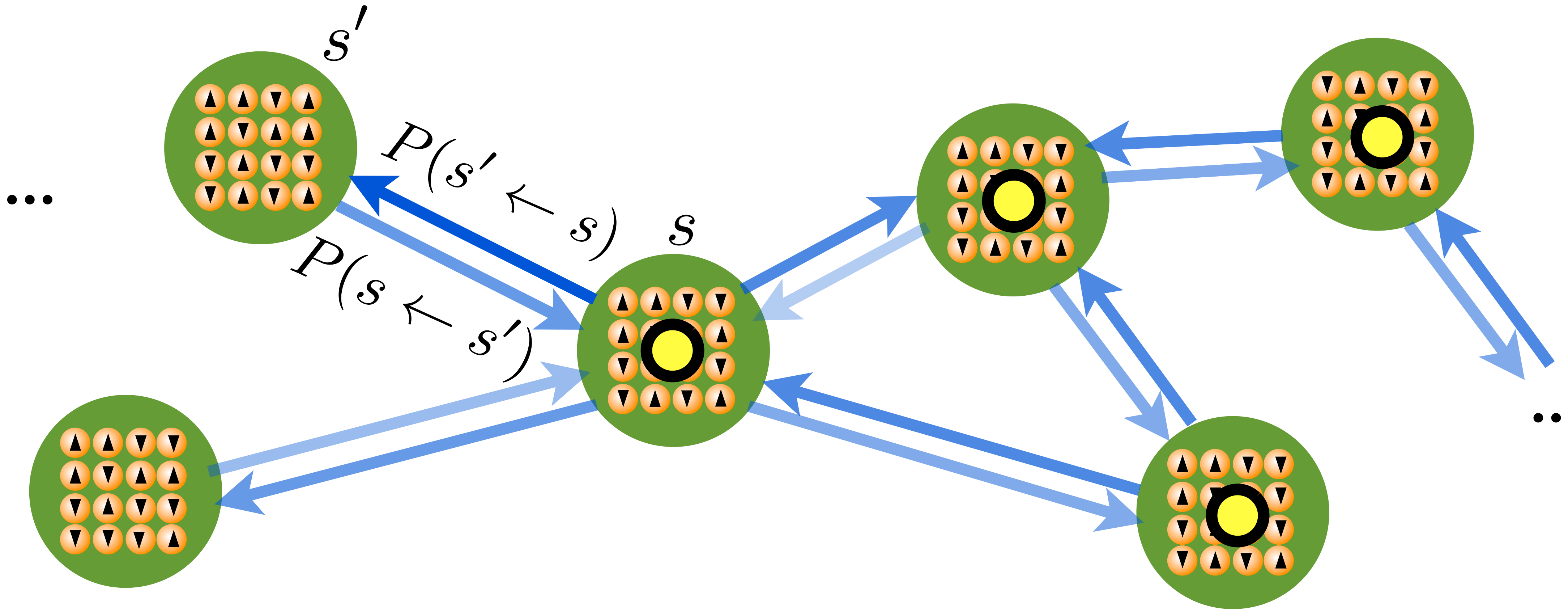
probability for state s,
in thermal equilibrium

$$Z = \sum_{s'} e^{-\frac{E(s')}{k_B T}}$$

Z for normalization:
“partition function”

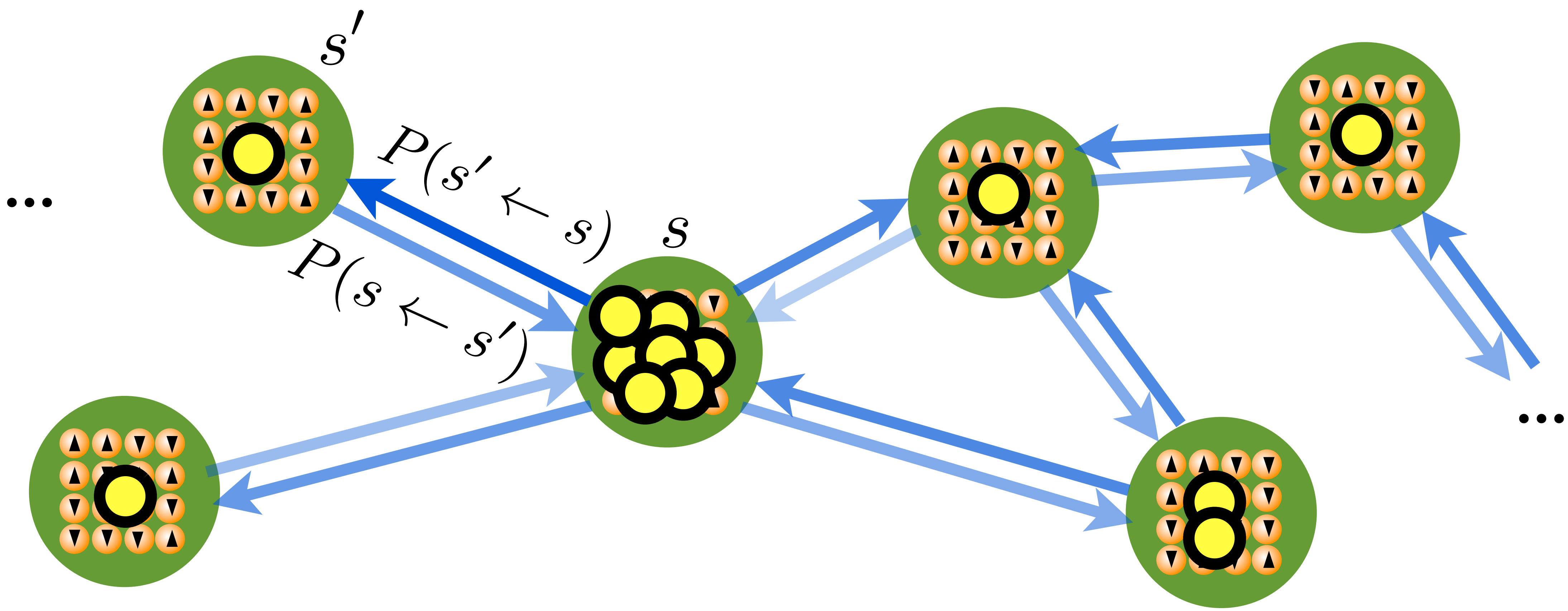
Problem: for a many-body system, exponentially many states (for example 2^N spin states). Cannot go through all of them!

Monte Carlo approach



Place system in some state, make stochastic transitions to other states (with prescribed transition probabilities)

Monte Carlo approach



Time evolution of ensemble?

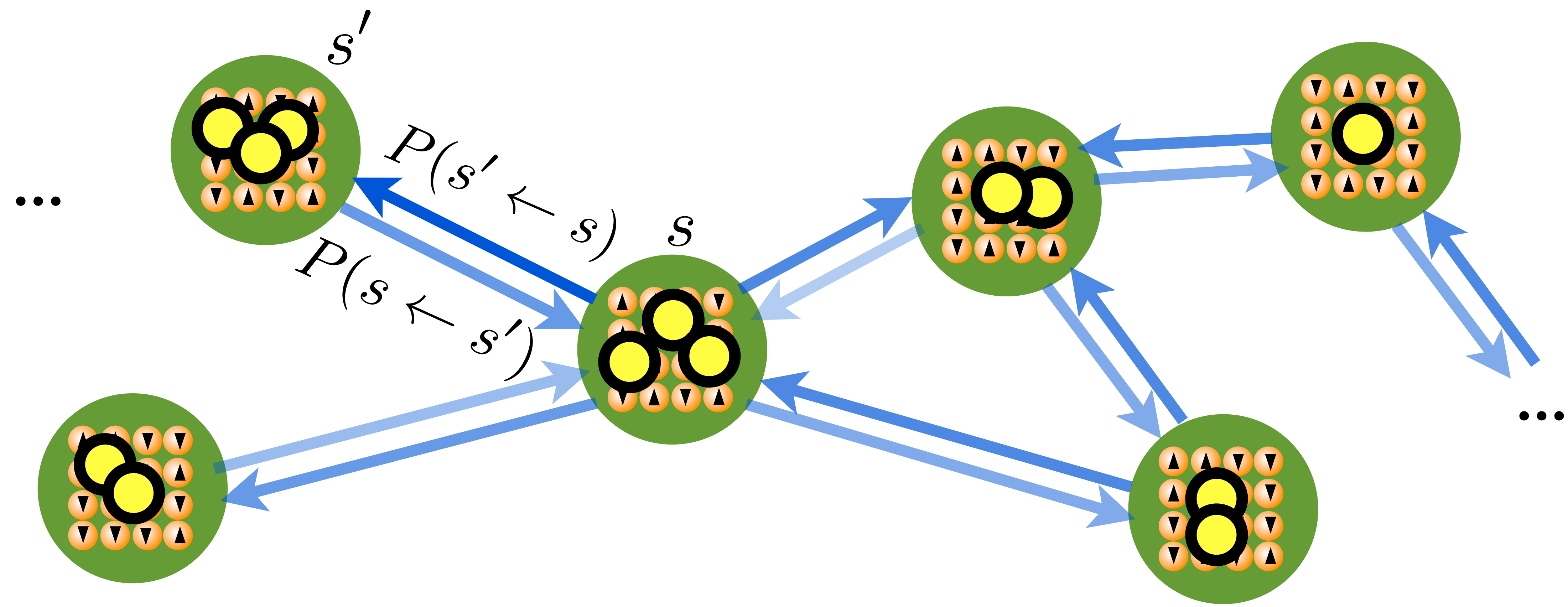
"IN" "OUT"

$$\Delta P(s) = \sum_{s'} P(s \leftarrow s')P(s') - P(s' \leftarrow s)P(s)$$

change in one time-step

P(s) = probability to find the system in state s (or: fraction of ensemble in this state)

Monte Carlo approach



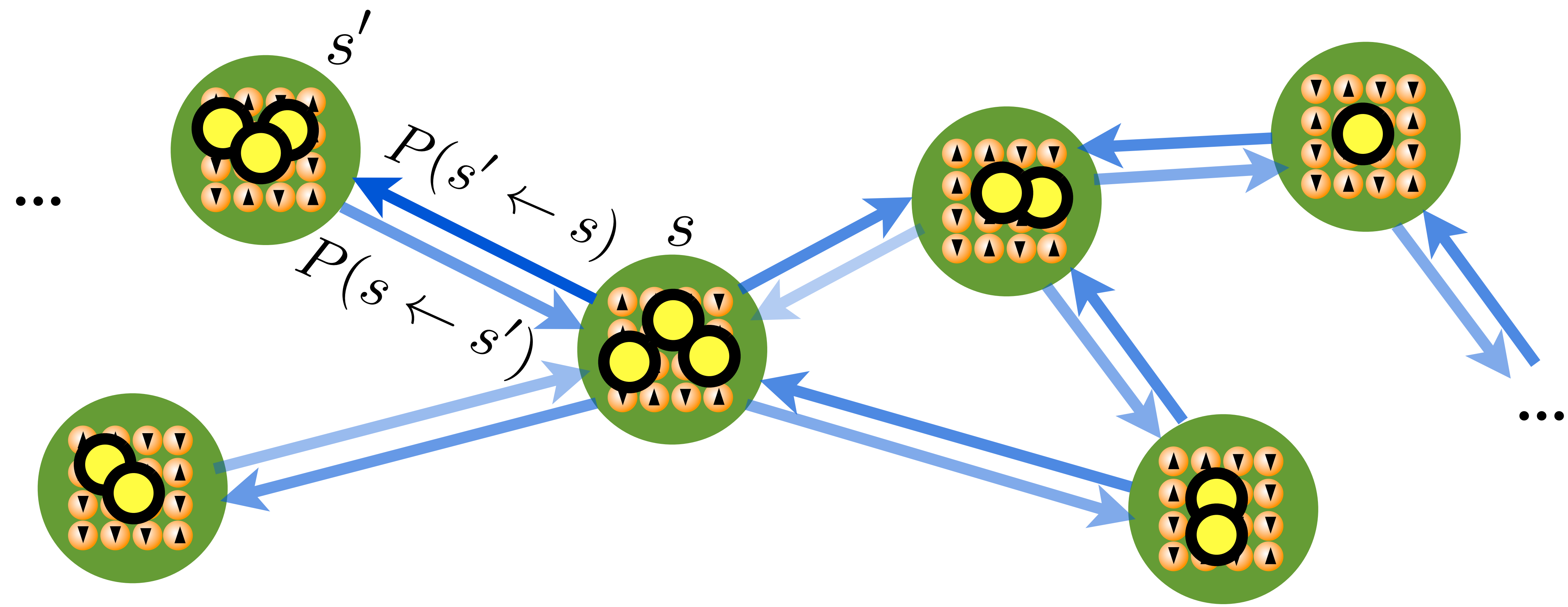
At long times: stable steady state distribution

If we have "detailed balance", i.e. if there exists a distribution P(s), such that for any pair of states:

$$\frac{P(s \leftarrow s')}{P(s' \leftarrow s)} = \frac{P(s)}{P(s')}$$

then P(s) is the long-time distribution!

Monte Carlo approach

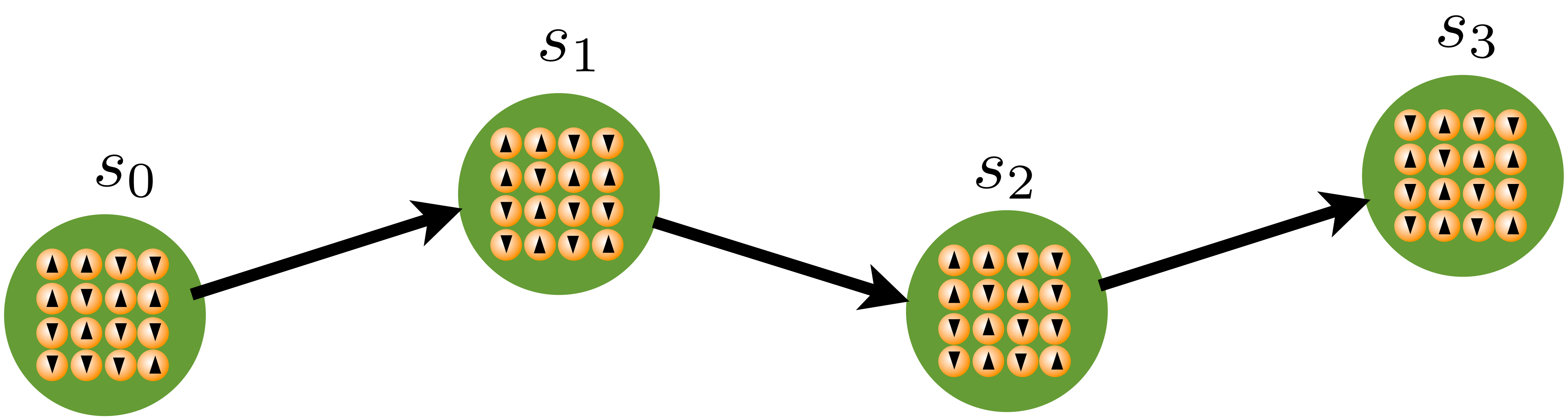


Monte Carlo for thermal equilibrium: choose transition probabilities such that P(s) will be the Boltzmann distribution!

$$\frac{P(s \leftarrow s')}{P(s' \leftarrow s)} = e^{\frac{E(s')-E(s)}{k_B T}}$$

example Metropolis algorithm: pick random spin, calculate energy change for spin flip. Do the flip if it lowers the energy. If the energy increases, only flip with probability $\exp(-\Delta E/k_B T)$

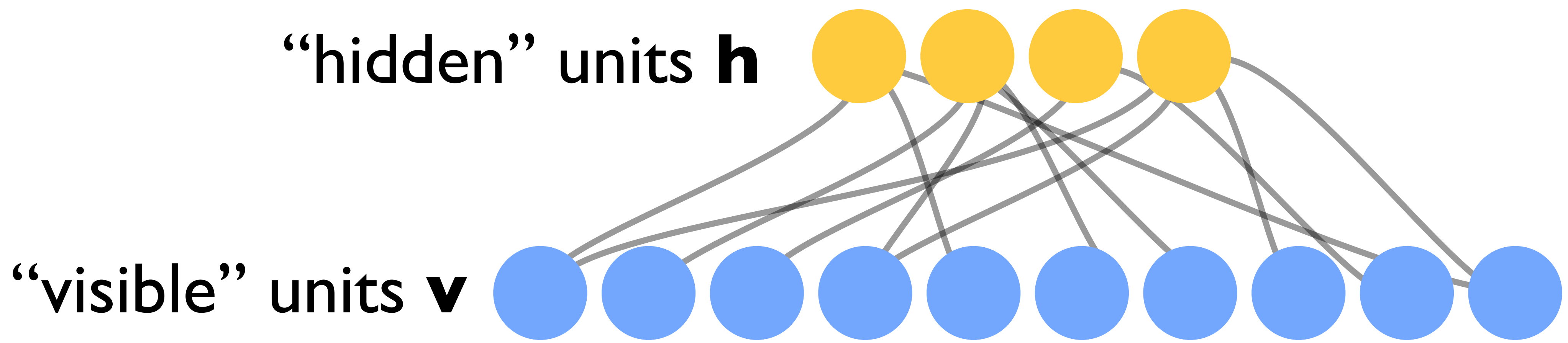
Markov chain



The sequence of visited states forms a so-called "Markov chain"

Markov = transitions without memory

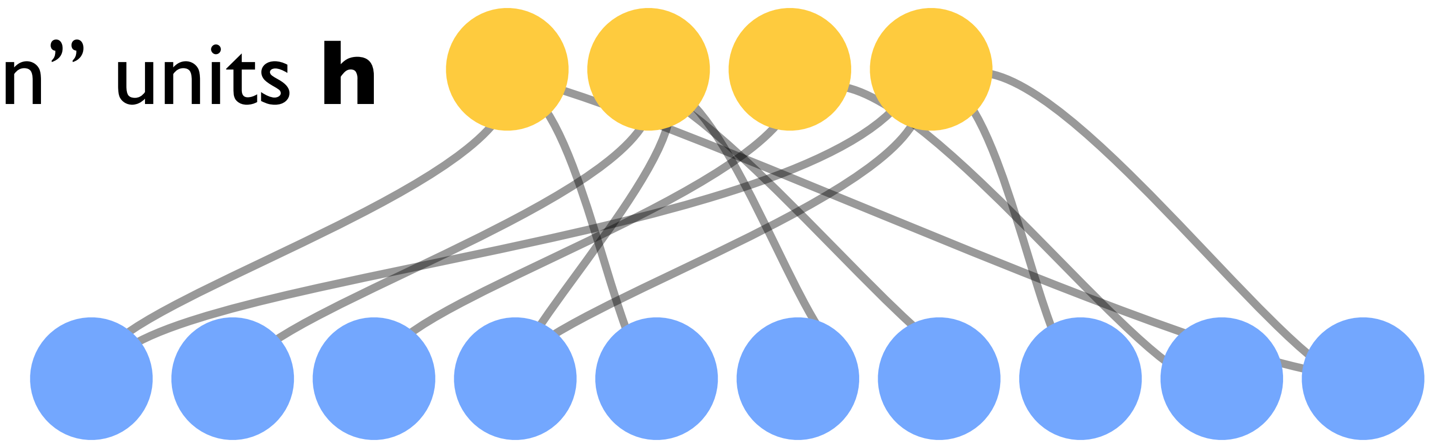
Restricted Boltzmann Machine



Each “unit” is like a spin (or a bit) that can be 0 or 1

Restricted Boltzmann Machine

“hidden” units **h**



“visible” units **v**

Each “unit” is like a spin (or a bit) that can be 0 or 1

Define “energy” (we will set $k_B T=1$)

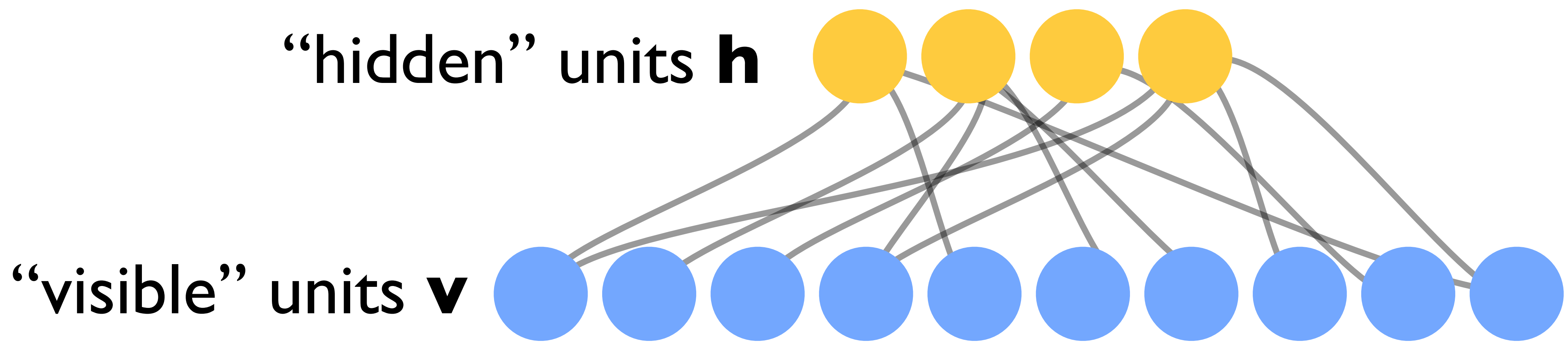
$$E(\mathbf{v},\mathbf{h}) = -\sum_{i\in\text{visible}} a_i v_i - \sum_{j\in\text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

“restricted”: no coupling v-v or h-h

w: couplings (weights)

see G. Hinton’s guide: http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf

Restricted Boltzmann Machine



Each “unit” is like a spin (or a bit) that can be 0 or 1

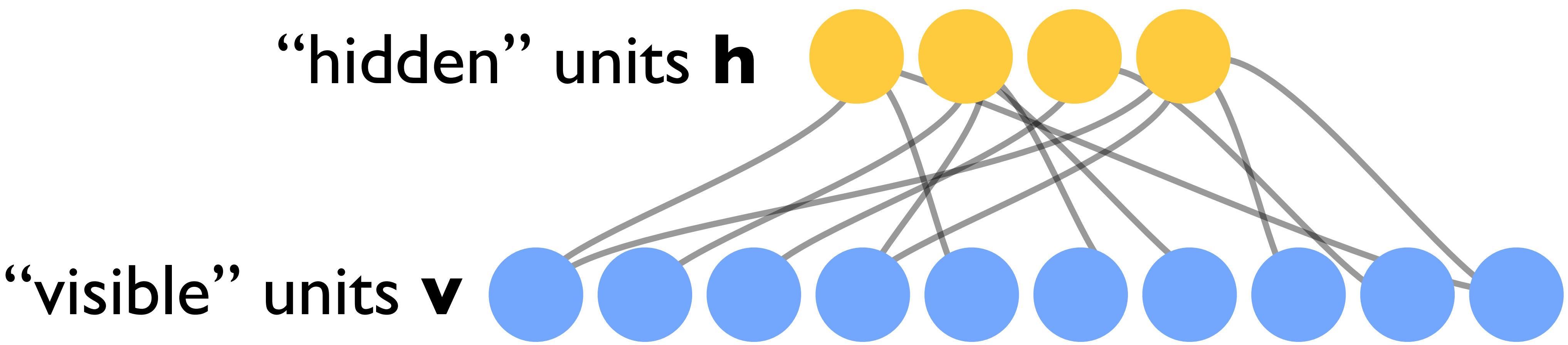
$$P(v,h) = \frac{e^{-E(v,h)}}{Z} \qquad Z = \sum_{v,h} e^{-E(v,h)}$$

$$P(v) = \sum_h P(v,h)$$

Goal: adapt weights (and biases), such that the probability distribution of a set of training examples is approximately reproduced by P(v)

$P(v) \approx P_0(v)$ —— from training samples

Restricted Boltzmann Machine



Each "unit" is like a spin (or a bit) that can be 0 or 1

Interpretation: the 'hidden units' represent categories of data (e.g. "dog+white+big")

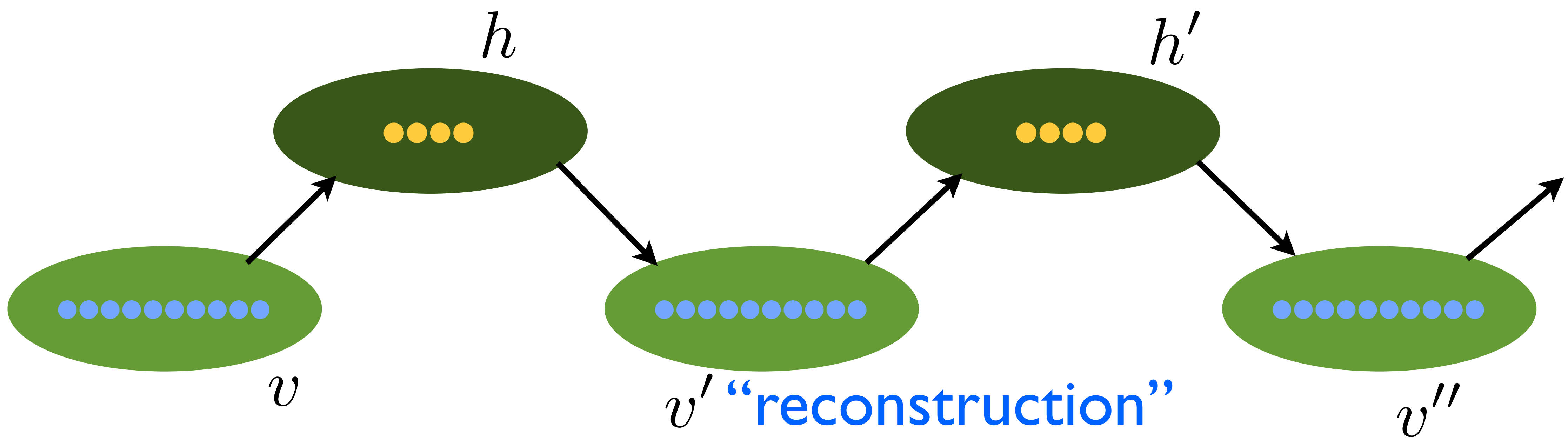
Building a Markov chain

Instead of the full state s=(v,h): Consider alternating transitions between v and h states

Set:

$$P(h \leftarrow v) = P(h|v) = \frac{P(v,h)}{P(v)}$$

$$P(v \leftarrow h) = P(v|h) = \frac{P(v,h)}{P(h)}$$



These transition probabilities fulfill detailed balance!

$$\frac{P(h \leftarrow v)}{P(v \leftarrow h)} = \frac{P(h)}{P(v)}$$

Thus: P(v) [and P(h)] are the steady-state distributions!

Building a Markov chain

$$ZP(v) = \sum_h e^{-E(v,h)} = \sum_h e^{\sum_i a_i v_i + \sum_j b_j h_j + \sum_{i,j} v_i h_j w_{ij}}$$

$$= e^{\sum_i a_i v_i} \Pi_j (1 + e^{z_j})$$

with: $$z_j = b_j + \sum_i v_i w_{ij}$$

where we used: $e^{\sum_j X_j} = \Pi_j e^{X_j}$ $\quad \sum_h \cdots = \sum_{h_0=0,1} \sum_{h_1=0,1} \sum_{h_2=0,1} \cdots$

Therefore:

$$P(h|v) = \frac{e^{-E(v,h)}}{ZP(v)} = \Pi_j \frac{e^{z_j h_j}}{1 + e^{z_j}}$$

Product of probabilities! All the h_j are independently distributed, with probabilities:

$$P(h_j = 1|v) = \frac{e^{z_j}}{1 + e^{z_j}} = \sigma(z_j)$$

sigmoid

$$P(h_j = 0|v) = 1 - \sigma(z_j)$$

Building a Markov chain

Given some visible-units state vector v, calculate the probabilities

$$P(h_j = 1|v) = \frac{e^{z_j}}{1+e^{z_j}} = \sigma(z_j)$$

sigmoid

Then assign 1 or 0, according to these probabilities, to obtain the new hidden state vector h

Similarly, go from h to a new v', using:

$$P(v'_i = 1|h) = \sigma(z'_i)$$

$$z'_i = a_i + \sum_j w_{ij} h_j$$

Updating the weights

Goal: adapt weights (and biases), such that the probability distribution of a set of training examples is approximately reproduced by P(v)

$$P(v) \approx P_0(v)$$ — from training samples

Minimize the categorical cross-entropy

$$C = -\sum_v P_0(v) \ln P(v)$$

But now (unlike earlier examples), there are exponentially many values for v, so we cannot simply have a network output P(v) for all v.
Still, let us take the derivative of C with respect to the weights w!

Updating the weights

$$C = -\sum_v P_0(v) \ln P(v)$$

$$\frac{\partial}{\partial w_{ij}} \ln P(v) = \frac{\frac{\partial}{\partial w_{ij}} \sum_h P(v,h)}{\sum_h P(v,h)}$$

$$\boxed{Z = \sum_{v',h'} e^{-E(v',h')}}$$

$$= \frac{\frac{\partial}{\partial w_{ij}} \sum_h e^{-E(v,h)}}{\sum_h e^{-E(v,h)}} - \frac{\frac{\partial}{\partial w_{ij}} \frac{1}{Z}}{\frac{1}{Z}}$$

$$= \frac{\sum_h v_i h_j e^{-E(v,h)}}{\sum_h e^{-E(v,h)}} - \frac{\sum_{v',h'} v'_i h'_j e^{-E(v',h')}}{Z}$$

overall:

$$\sum_v P_0(v) \frac{\partial}{\partial w_{ij}} \ln P(v) = \sum_{v,h} v_i h_j P(h|v) P_0(v) - \sum_{v',h'} v'_i h'_j P(v',h')$$

Updating the weights

$$\sum_v P_0(v) \frac{\partial}{\partial w_{ij}} \ln P(v) = \sum_{v,h} v_i h_j P(h|v) P_0(v) - \sum_{v',h'} v'_i h'_j P(h'|v') P(v')$$

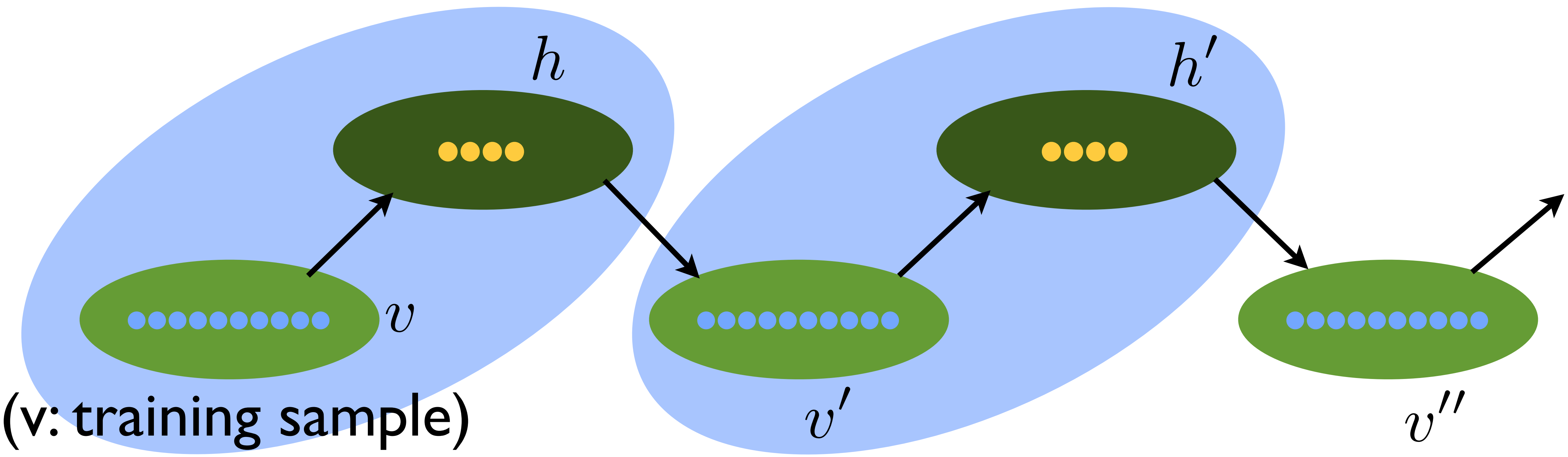
easy: draw one training sample v, then do one Markov chain step from v to h; average over all samples v

hard: need to average over the correct distribution P(v) belonging to the Boltzmann machine!

Updating the weights

$$\sum_v P_0(v) \frac{\partial}{\partial w_{ij}} \ln P(v) = \sum_{v,h} v_i h_j P(h|v) P_0(v) - \sum_{v',h'} v_i' h_j' P(h'|v') P(v')$$

Could obtain P(v) by running the Markov chain for really long times! Very expensive!



(v: training sample)

Rough approximation, used in practice: Just take v',h' from the second pair of the chain! [For better approx.: can take a pair further down the chain]

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle - \langle v_i' h_j' \rangle)$$

(averaged over a batch of training samples v starting the chain)

Updating the weights

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle - \langle v_i' h_j' \rangle)$$

(averaged over a batch of training samples v starting the chain)

"Contrastive Divergence" (CD) algorithm by G. Hinton

Note: At least we can claim that $P_0(v) = P(v)$ would be a fixed point of this update rule, since then the two averages on the right-hand-side yield identical results. Of course, usually the restricted Boltzmann machine will not be able to reach this point, since it cannot represent arbitrary P(v).

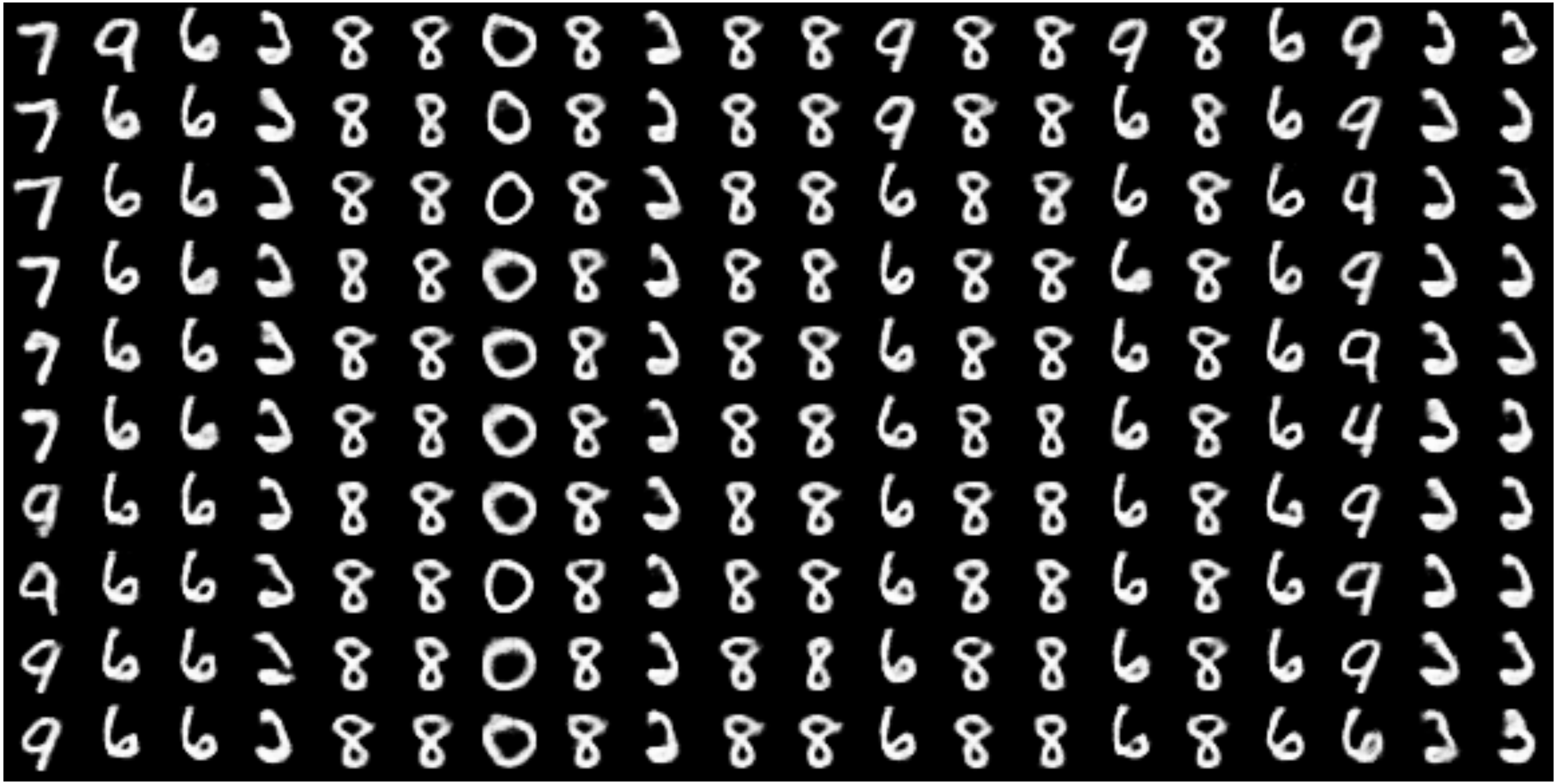
$$\Delta a_i = \eta(\langle v_i \rangle - \langle v_i' \rangle)$$

$$\Delta b_j = \eta(\langle h_j \rangle - \langle h_j' \rangle)$$

Restricted Boltzmann Machine for MNIST

example from http://deeplearning.net/tutorial/rbm.html

Markov chain steps (1000 steps between each row!)

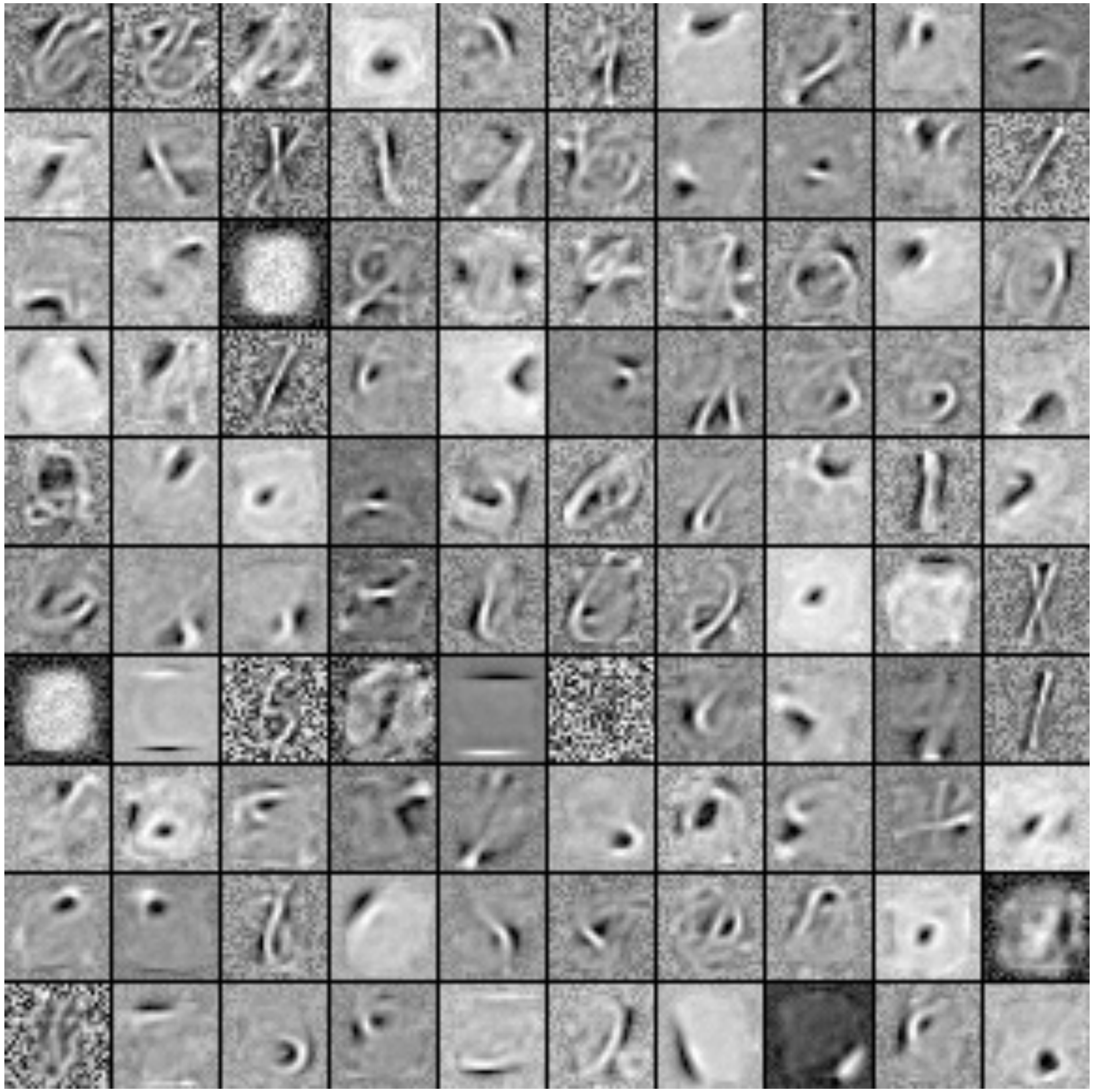


Each column: a different, independent Markov chain

Restricted Boltzmann Machine for MNIST

example from http://deeplearning.net/tutorial/rbm.html

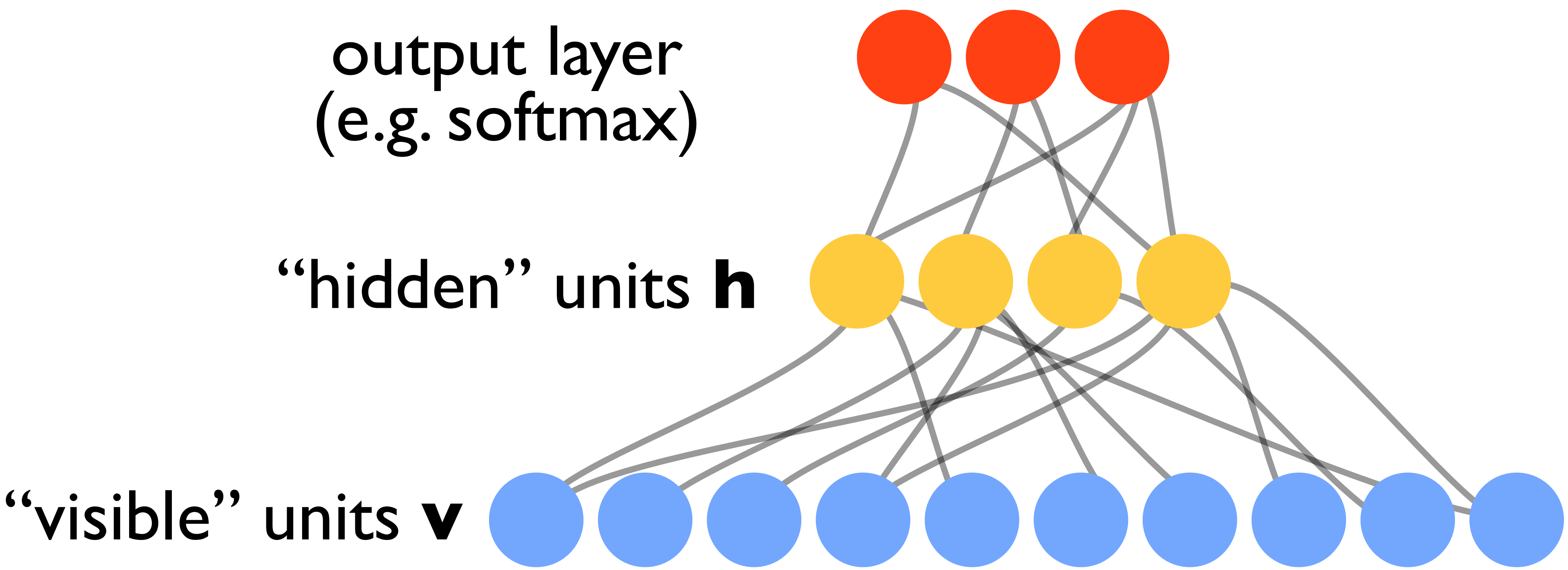
The learned weights for the 100 hidden units



RBM as a starting point

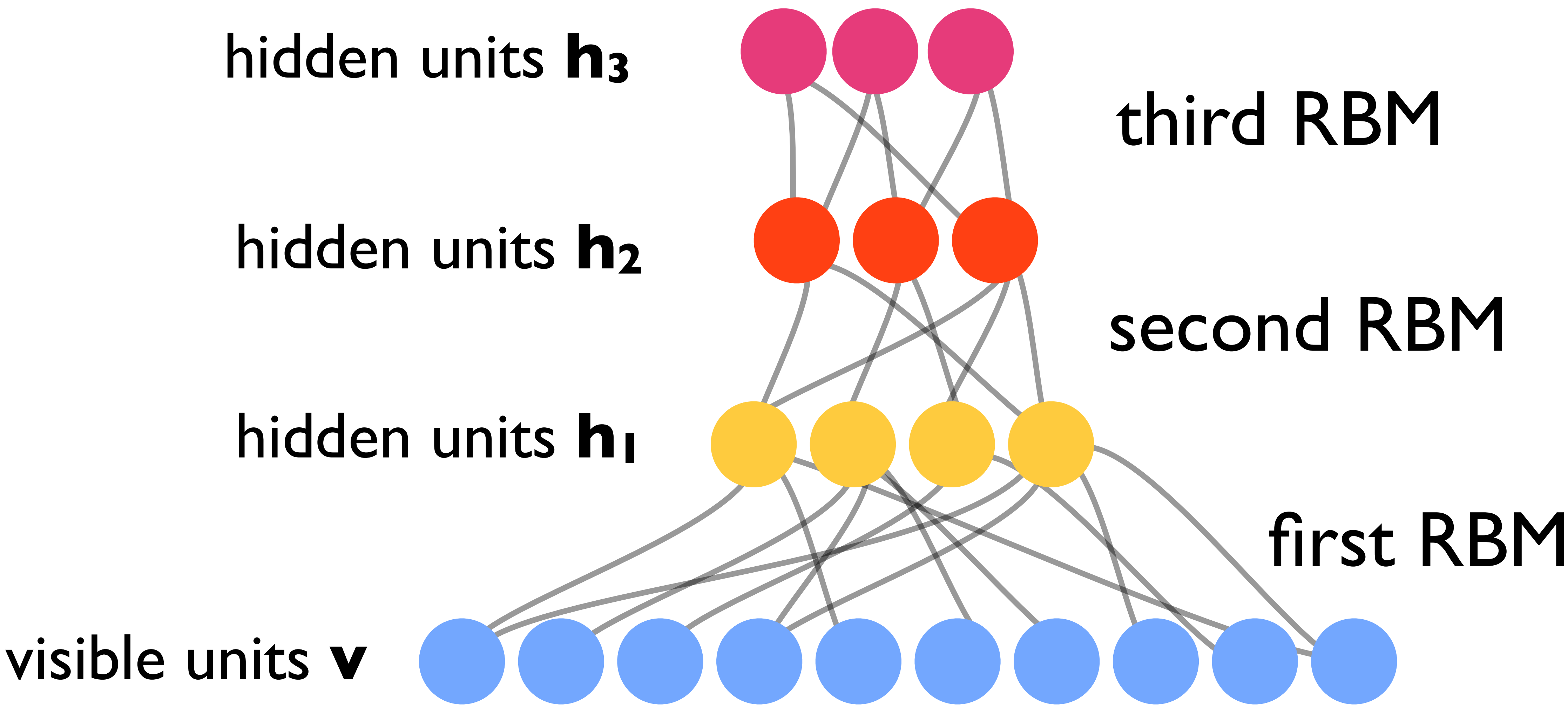
First train RBM, then connect hidden layer to some output layer for supervised learning of classification

Idea: RBM provides unsupervised learning of important features in the training set (pre-training)



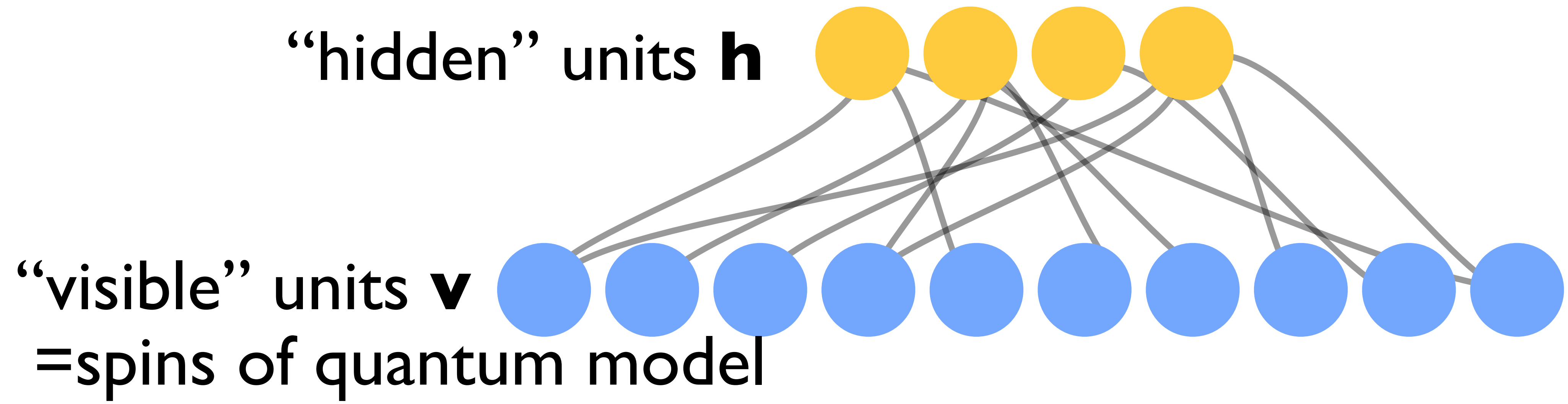
Deep belief networks

Stack RBMs: First train a simple RBM, then use its hidden units as input to another RBM, and so on



Afterwards, fine-tune weights, e.g. by supervised learning

Application to Quantum Physics



Try to solve a quantum many-body problem (quantum spin model) using the following **variational ansatz** for the wave function amplitudes:

$$\Psi(\mathcal{S}) = \sum_h e^{\sum_j a_j \sigma_j^z + \sum_i b_i h_i + \sum_{ij} h_i \sigma_j^z W_{ij}}$$

Carleo & Troyer, Science 2017

$$\mathcal{S} = (\sigma_1^z, \sigma_2^z, \ldots, \sigma_N^z)$$ one basis state in the many-body Hilbert space

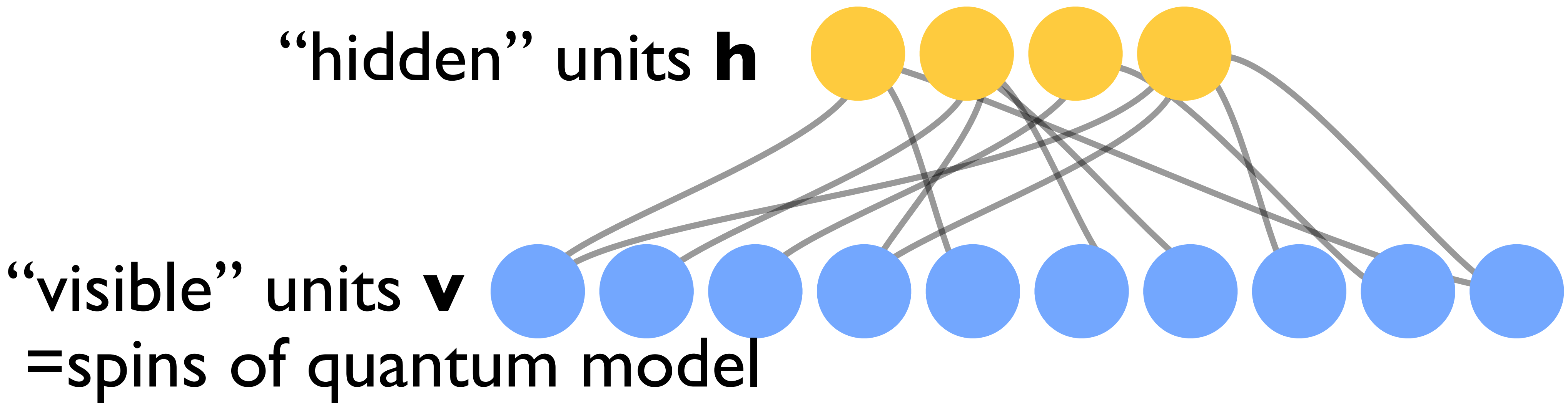
$$\sigma_j^z = \pm 1$$

(in general, a,b,W may be complex)

$$h_i = \pm 1$$

This is exactly (proportional to) the RBM representation for P(v) [with v=S]!

Application to Quantum Physics



Minimize the energy $$\frac{\left\langle \Psi \middle| \hat{H} \middle| \Psi \right\rangle}{\langle \Psi | \Psi \rangle}$$

by adapting the weights W and biases a and b!

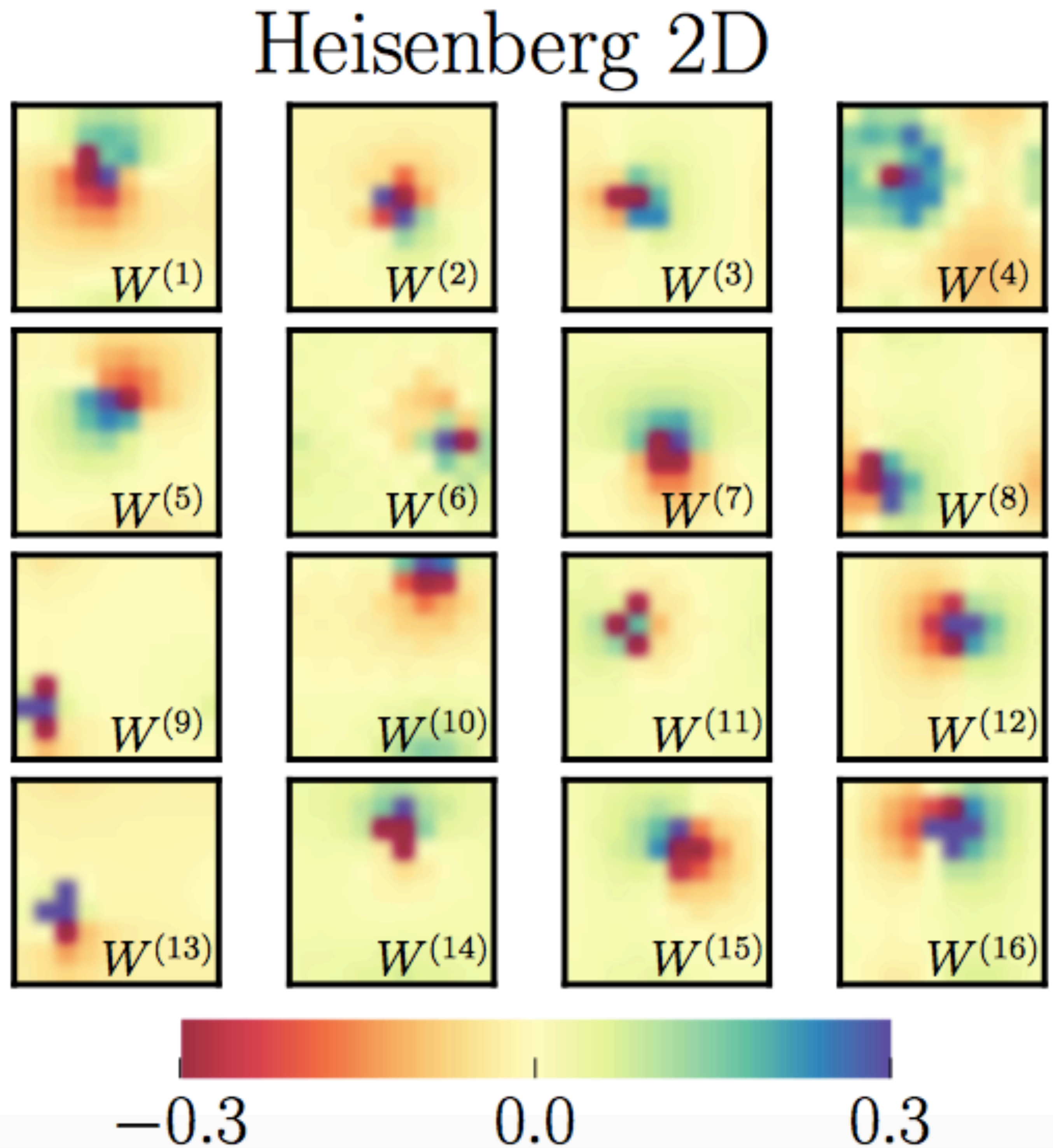
[requires additional Monte Carlo simulation, to obtain a stochastic sampling of the gradient with respect to these parameters]

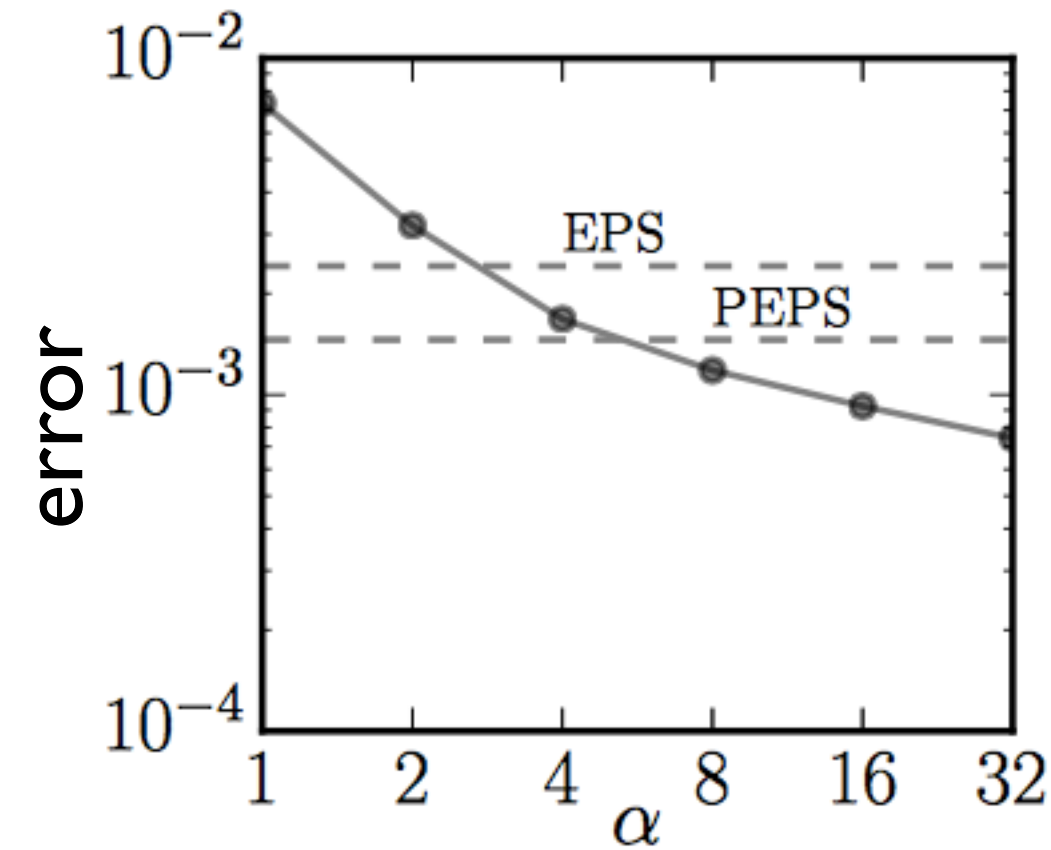
For example: sample probabilities by using Metropolis algorithm, with transition probabilities

$$P(\mathcal{S}' \leftarrow \mathcal{S}) = \min(1, \left| \frac{\Psi(\mathcal{S}')}{\Psi(\mathcal{S})} \right|^2)$$

Application to Quantum Physics

Carleo & Troyer, Science 2017





number of ‘filters’

Exploit translational invariance (like in convolutional nets); weights are “filters” (convolutional kernels)

Find updates on

http://machine-learning-for-physicists.org