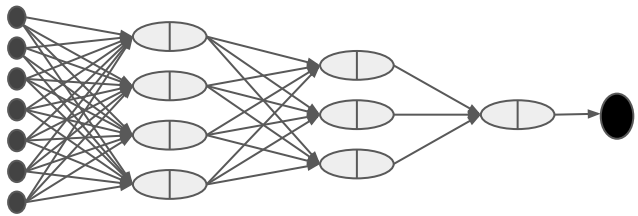
Recurrent neural network (RNN)

So far we have been processing input to the network without any notion of sequentiality (although with CNNs for image tasks 2D data is spatial structure is taken into account):



No concept of ordering for inputs - same applies to intermediate outputs.

Sequential data is, however, abundant:

- Text data is by nature sequential (interpreted in word or character level)
- Voice data is sequential
- Video can be interpreted as sequence of frames
- Processes in nature, such as temperatures etc exhibit sequentiality
- Human behaviour can often be interpreted as sequences
- We will now concentrate on text data →

Word embeddings

One-hot encoding has some very undesirable properties:

- Dimension = size of dictionary (could be 10000 - 60000)
- Each word has the same distance to all other words - distance information is quite useless

How about mapping each word to a vector of dimension < size of dictionary?

Could there be a mapping that is more compact and also captures something about the relationships between words?

Word embeddings - imaginary example

	female	juicy	large	edible	yellowish
king	0.01	0.05	0.6	0.04	0.6
queen	0.95	0.04	0.55	0.03	0.64
uncle	0.03	0.01	0.7	0.02	0.62
aunt	0.98	0.03	0.6	0.04	0.66
apple	0.1	0.6	0.1	0.95	0.2
orange	0.1	0.8	0.07	0.99	0.8

Word embeddings - properties

king - queen = [-0.94 0.01 0.05 0.01 -0.04]

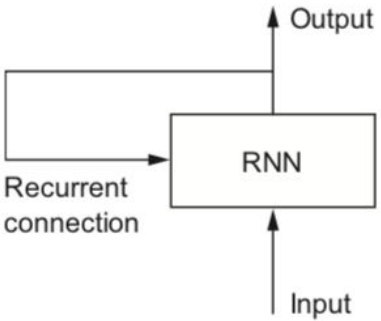
uncle - aunt = [-0.95 -0.02 0.1 -0.02 -0.04]

So king - queen ≈ uncle - aunt

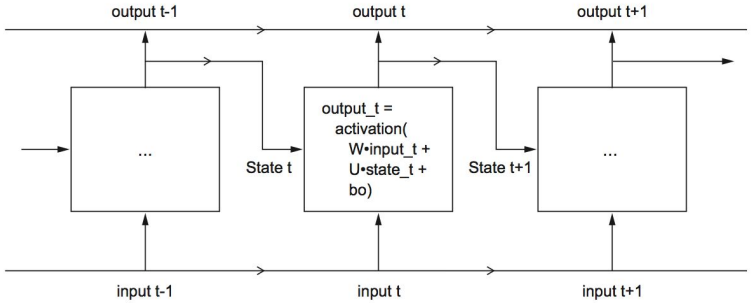
“King is to queen like uncle is to aunt”

(Try out your favorite samples for near words, analogies etc with http://epsilon-it.utu.fi/wv_demo/)

Deep network layer type for sequential data - RNN

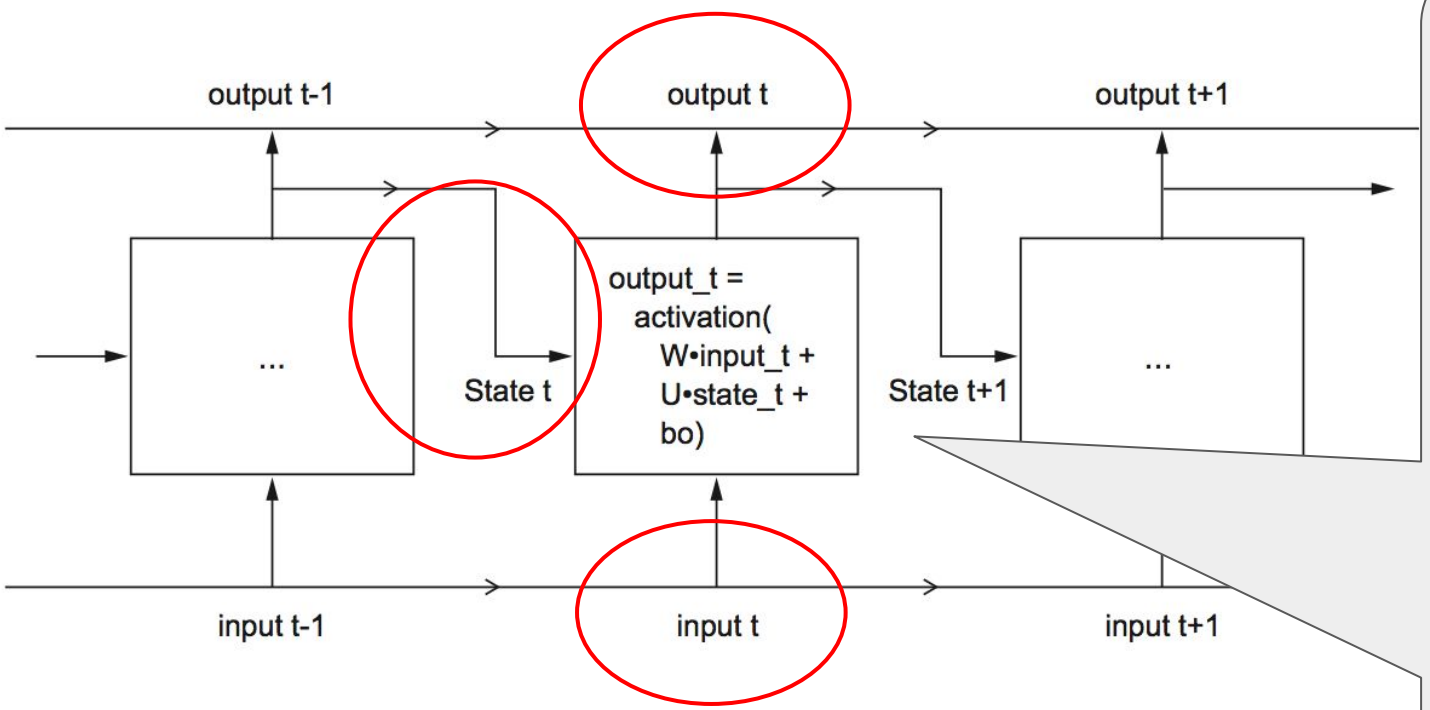


RNN processes its input by iterating over input sequence elements one by one and maintains a state (a memory) that is determined by the data it has processed so far.



RNN **unrolled** over timesteps: computation proceeds from left to right (from the beginning of the sequence to the end). When sequences are long, this means that the network is quite deep.

Recurrent neural network (RNN)



When computing output for position `t`, input `t` and output from previous position are combined. Multiplier matrices `W` and `U` as well as bias `bo` are for the unrolled node, they are **shared across the layer (= over all timesteps)** when unrolling. All signals flowing in the network are vectors.

SimpleRNN parameter count

of output features: 20

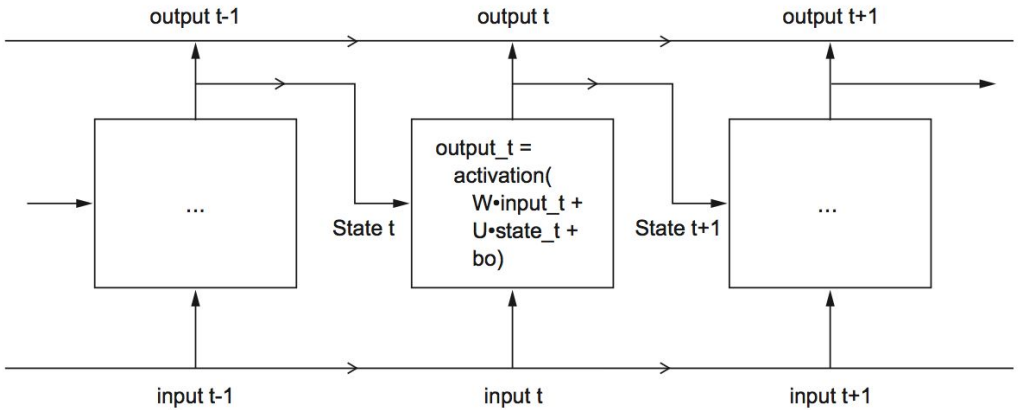
# of input features: 5

Matrix `W` dimensions: 20 x 5

Matrix `U` dimensions: 20 x 20

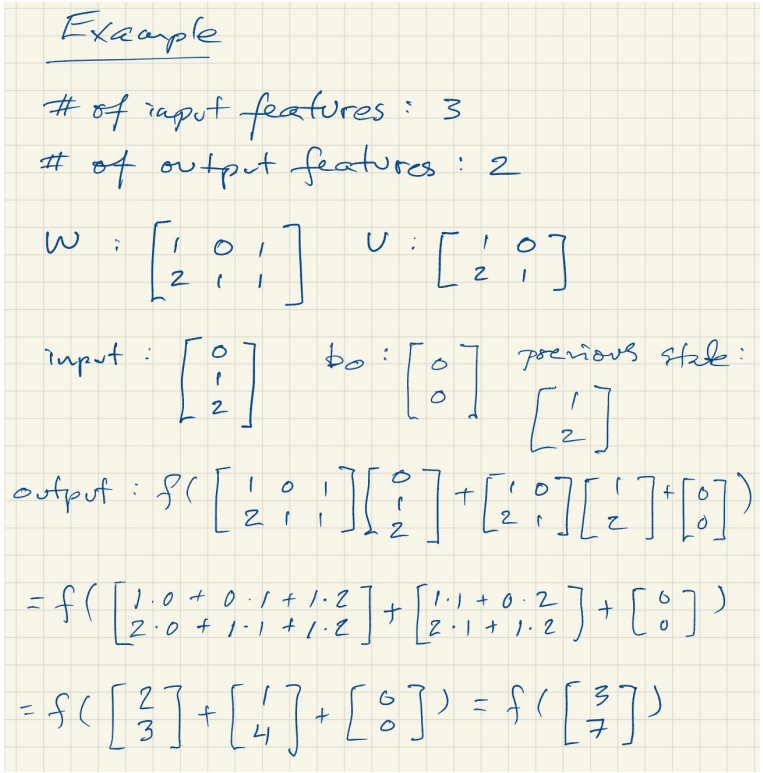
Bias `bo` dimensions: 1 x 20

Total: 100 + 400 + 20



Again, remember the parameters are shared over timesteps in unrolled diagram = `W`, `U`, `bo` are identical for all time steps.

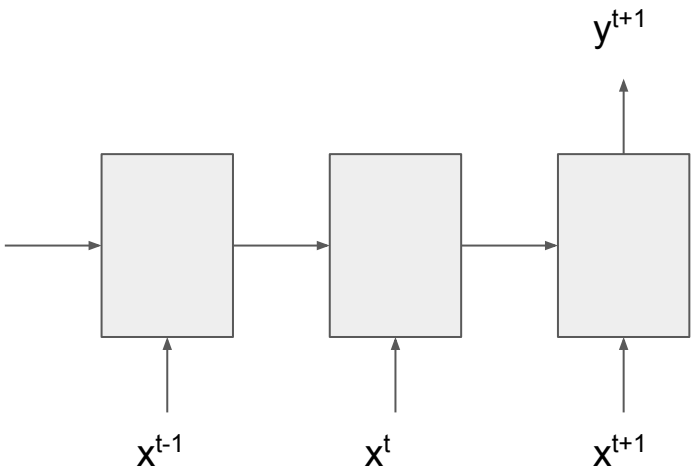
SimpleRNN example calculation



Here we are assuming W, U and bo have some values (learned during previous steps), and previous state (state t in diagram) has a value.

The next output is computed based on those values, an activation function (in the example f()) is applied to that which gives the next output (and state for next step).

RNN architectures: many-to-one

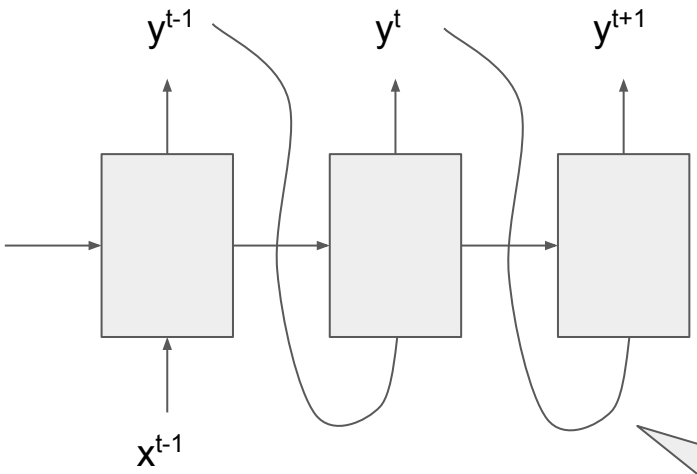


Output at the end of sequence only (`return_sequences=False` in Keras). The output depends on the whole input sequence.

Application examples for many-to-one

- Sentiment analysis: from a product review text predict the number of stars, or positive/negative sentiment of the review
- Recognize activity from video: running/jumping/etc

RNN architectures: one-to-many



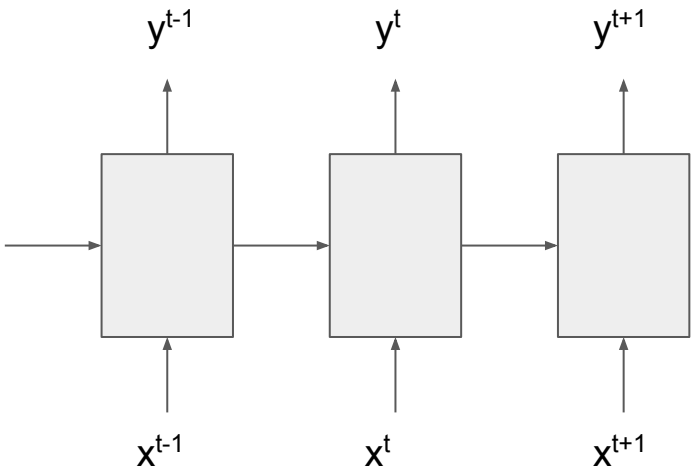
Output a value from each element. Internally assign output from previous step to input of next step.

Starting from an element (or from empty), generate a sequence:

- Generate words or music (sometimes start from a few elements, especially when generating text).

Curved lines represent connection from output of preceding node to input of next node

RNN architectures: many-to-many

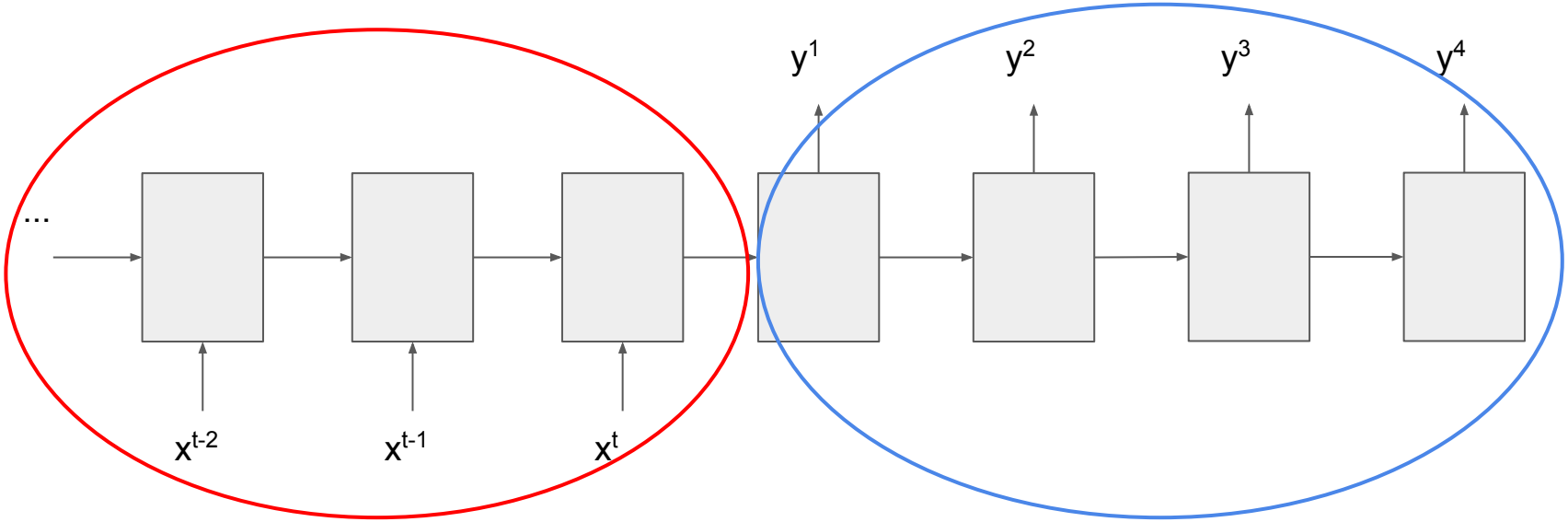


Some typical examples for many-to-many architectures include:

- Speech-to-text
- (Machine translation)
- (Image captioning - based on an image, produce text to caption it)

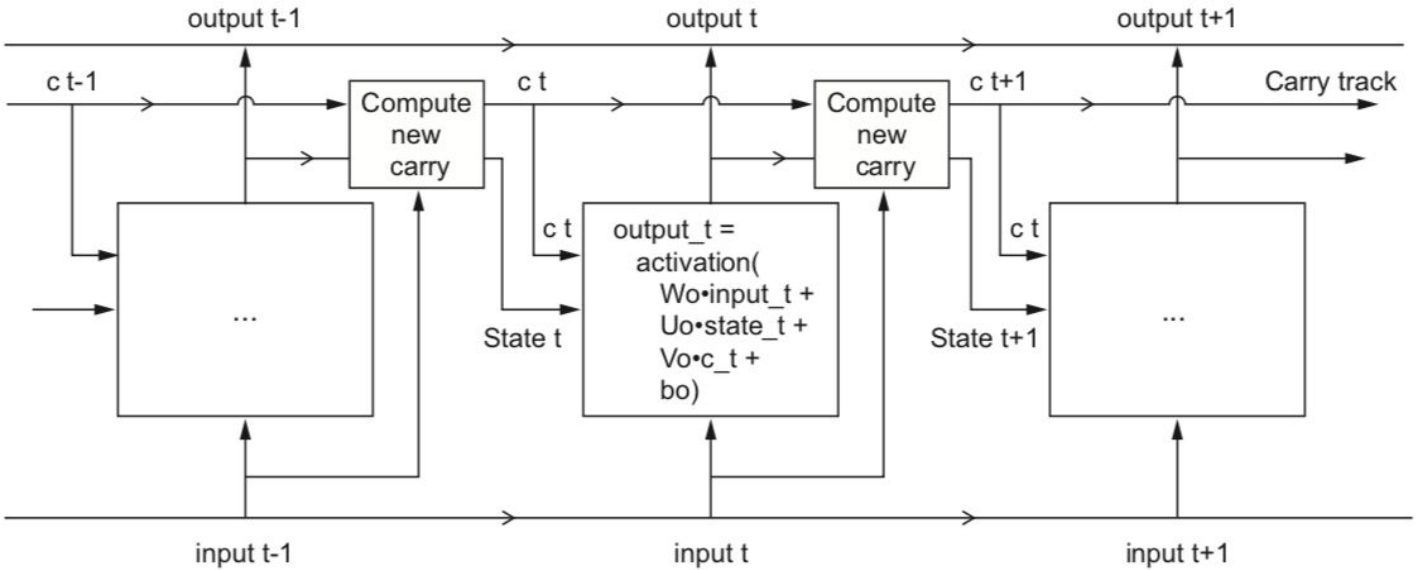
For many of these problems input sequence length ≠ output sequence length. How to deal with that? Padding, restricting sequence length, and see next.

RNN architectures: many-to-many (encoder-decoder)



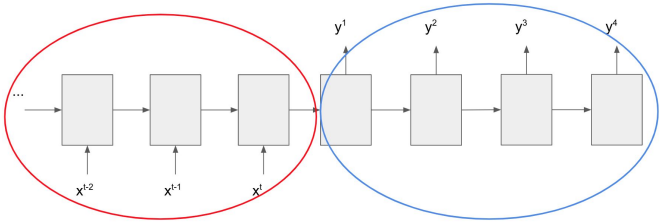
- Input sequence length ≠ output sequence length
- Learn **representation** of input and generate output sequence from that
- Two parts: encoder (reads input, encodes it into internal representation) and decoder (reads internal representation to produce output)

LSTM (Long Short-Term Memory)

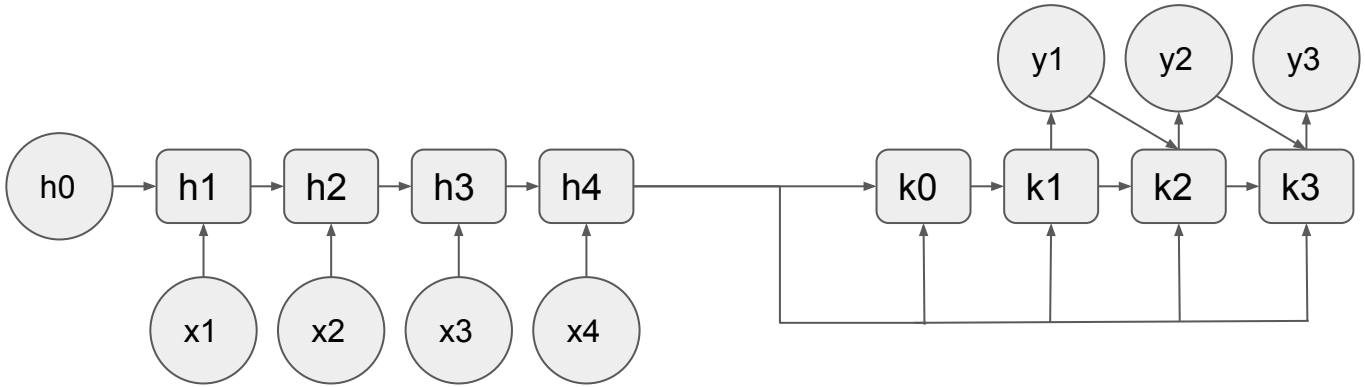


Chollet, page 204, for a more complete explanation see http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Language translation



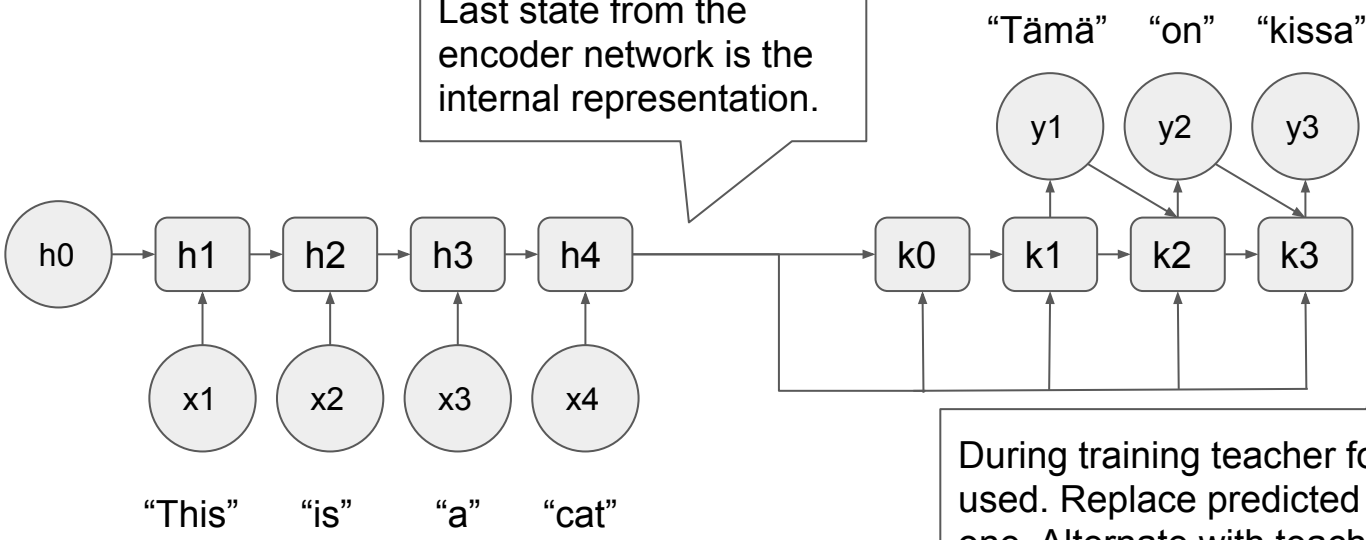
Translation could be implemented with an encoder-decoder network - encoder creates an internal representation of the source sentence, and decoder turns the representation into sentence in target language.



Translation with encoder-decoder

Predict probability distribution of words from each step. But: target is generate the most probable sentence, not just most probable words. *Beam search* algorithm is one possibility.

Last state from the encoder network is the internal representation.

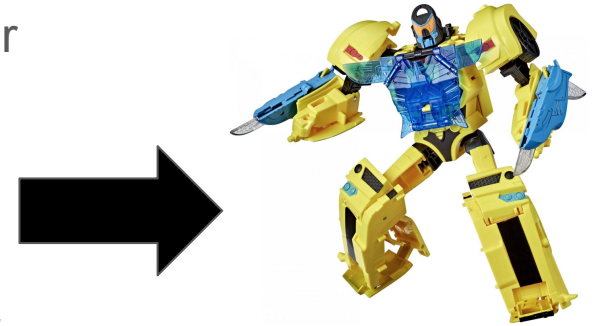


During training teacher forcing is often used. Replace predicted y(n) with correct one. Alternate with teacher forcing on and off during training.

From sequence-to-sequence model to transformer architecture

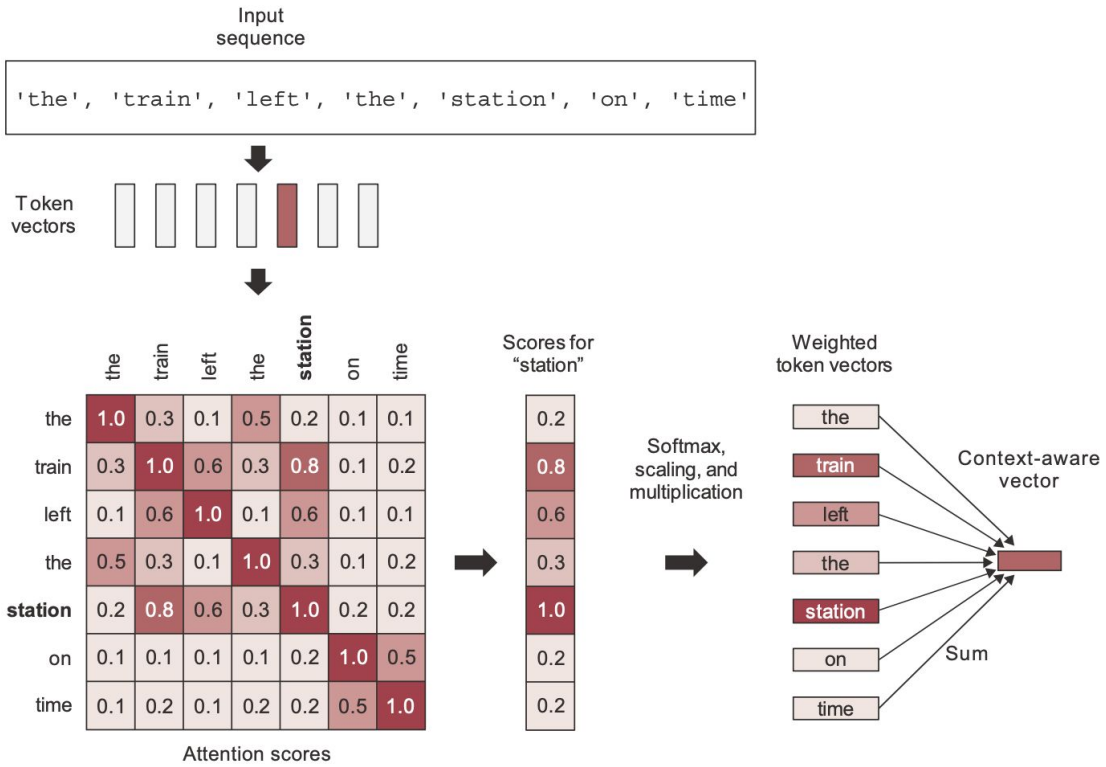
Sequence-to-sequence translation has shortcomings:

- Maybe very complicated sources (sentencies / paragraphs / documents) can't be held in encoder state vector (to create an effective intermediate representation)
- The length of context in a RNN (even if it is an LSTM) is limited - when processing a large document the context learned in the beginning of the sequence doesn't necessarily progress to the end
- Only a single embedding is learned for each word → a word has only one "meaning"



Transformer architecture

Context-aware attention example



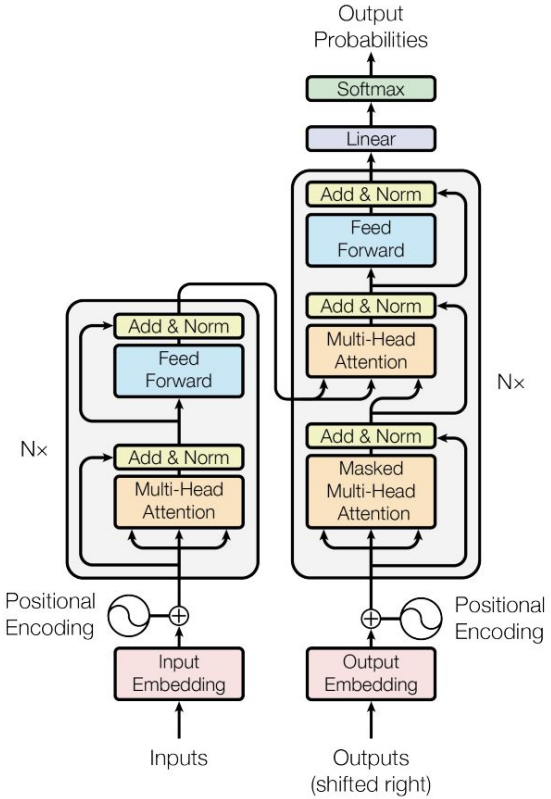
Example from Deep Learning With Python 2nd Edition by Chollet

Find context-aware vector for "station": attention score computed by taking the inner product, or dot product, of each pair of vectors representing words in the sentence is calculated.

For "station" attention score of "train" is highest (ignoring "station") → it gets highest weight in the context-aware vector .

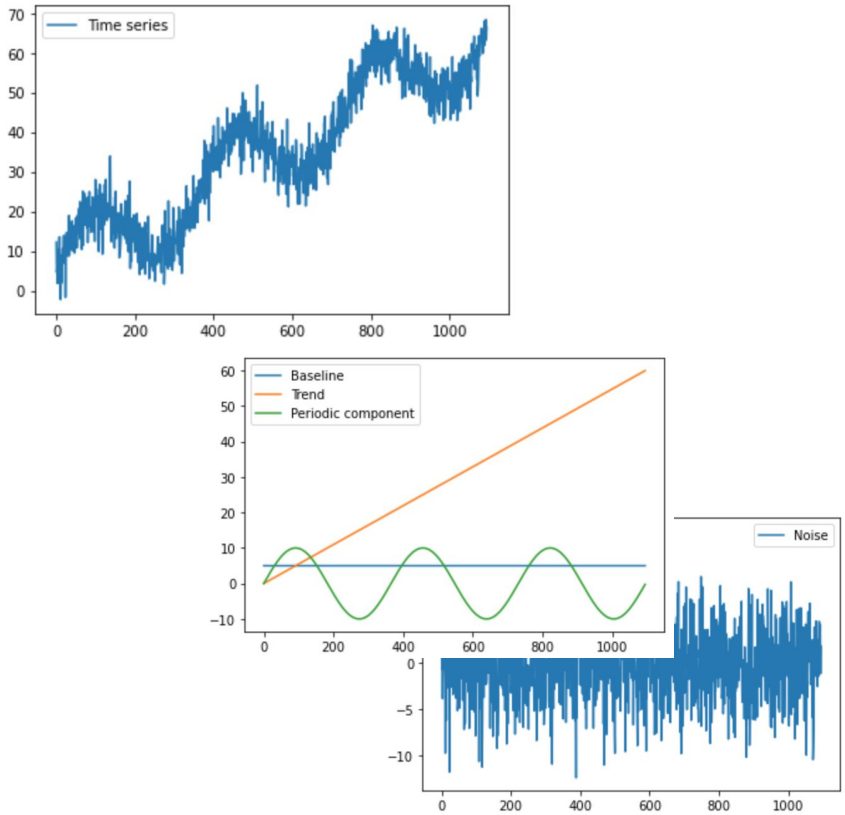
The resulting context-aware vector is the one that is used as the word vector in this context ie. it replaces the original static embedding.

Transformer architecture



- Key parts of the architecture are the multi-head attention modules
- RNN (GRU/LSTM) is not used at all - it is replaced with dense networks only (Feed Forward).
- Residual connections (Add & Norm) are used for the same reason as in resnet architecture - rectify vanishing gradients problem.
- But but but! In dense networks there is no ordering. Add to context-aware embedding vector positional encoding.
- In decoder part multi-head attention is masked - attention can only depend on previous positions, not ones after the current position, in training

Time series analysis

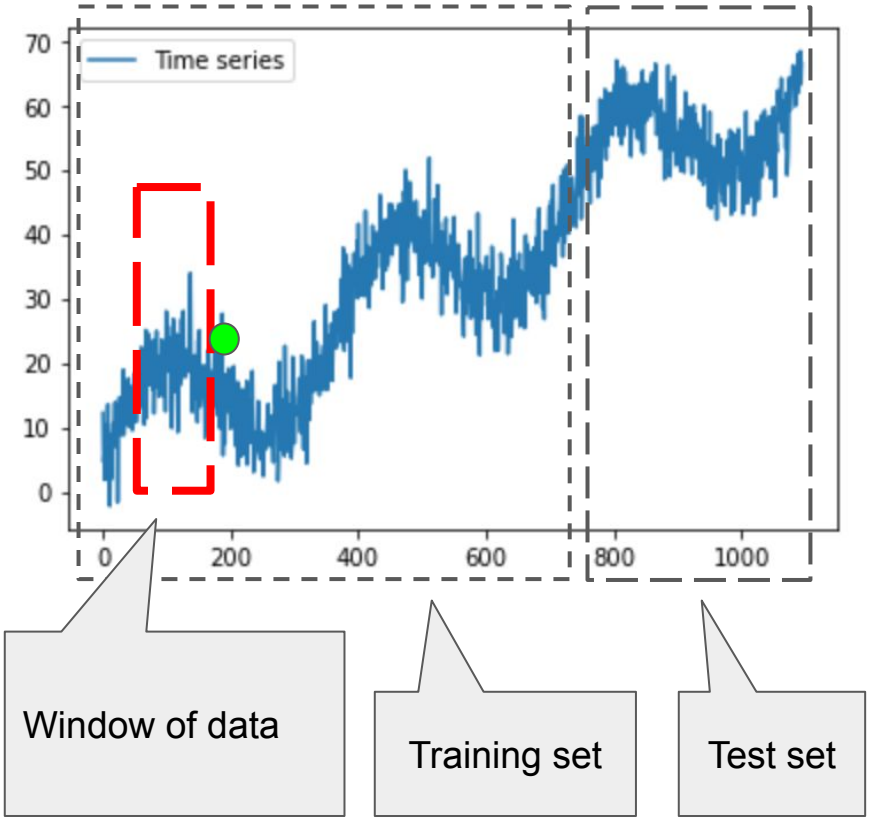


ANNs can be applied to making predictions based on existing time series data. By its nature time series data is strongly correlated - next value typically depends on previous values over a time slice.

Time series can often be divided into

- Baseline
- Trend component
- Periodic/seasonal component
- Noise

Training and test sets and windows



Using training samples, learn to predict next value (green dot) based on values in red box.

Sample for the model:

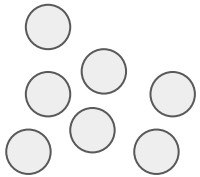
([data in window], value after the window)

1D convolution

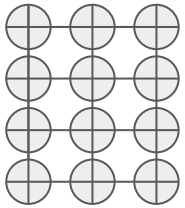
Like 2D convolution but in one dimension only:

1D convolution could be used for processing voice signal, but it can be useful with any sequential data.

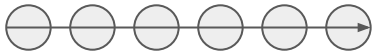
Different type of data → different ANNs



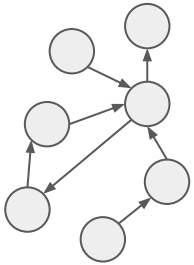
- Dense net
- Unstructured data



- Convnet
- 2D data

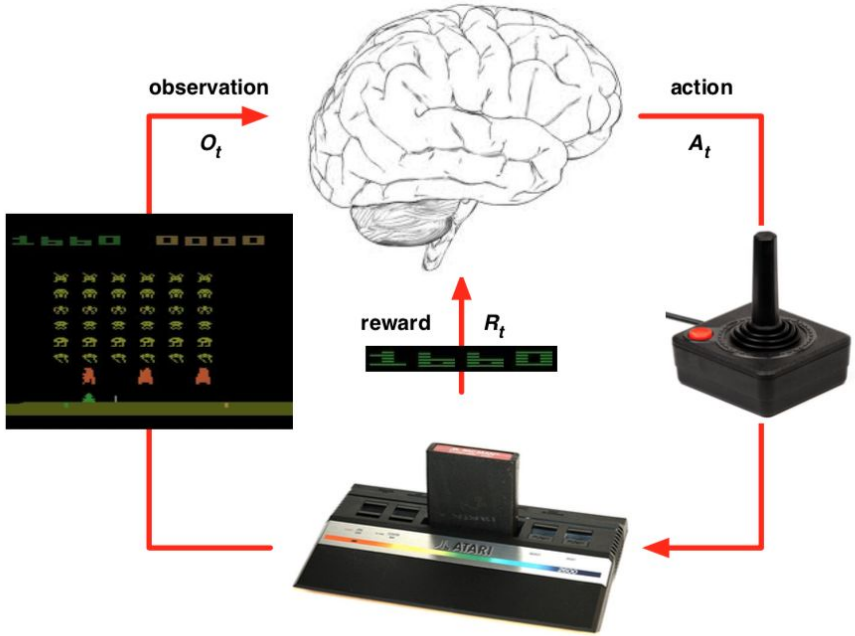


- RNN
- 1D sequential data



- Graph neural nets
- Graph-structured data
- Semi-supervised learning - some nodes are labeled, some not → predict labels for unlabeled, based on graph structure

Atari Example: Reinforcement Learning



- Rules of the game are unknown
- Learn directly from interactive game-play
- Pick actions on joystick, see pixels and scores

From Reinforcement Learning course slides by David Silver (https://www.davidsilver.uk/teaching/)

Modeling the environment - Markov state

We say that the state model is **Markovian** iff (if and only if) the probability function for entering the next state S_{t+1} from state S_t satisfies

$$p(S_{t+1} \mid S_t) = p(S_{t+1} \mid S_1, S_2, \ldots, S_t)$$

In other words, **probability of entering the next state depends only on current state**, not on states before the current state.

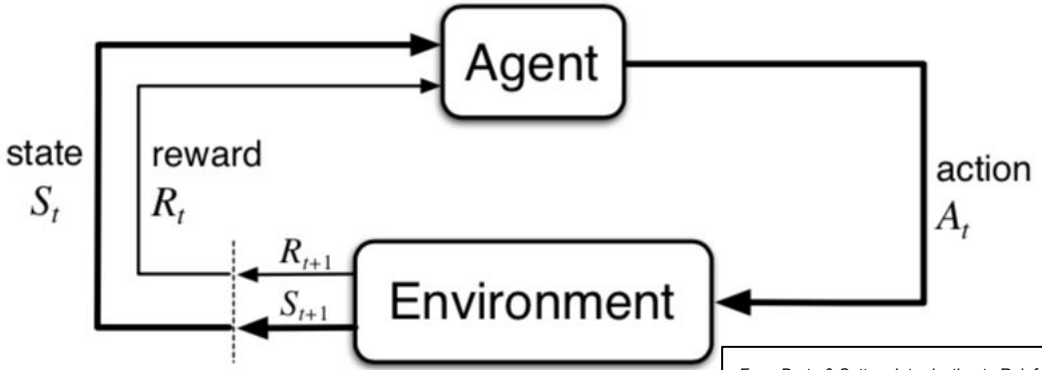
So, the current state captures everything there is to know - we need not care about the history.

The environment state in reinforcement learning is often/usually assumed to be Markovian.

Markov decision process

A tuple <*S, A, p,* γ> is a Markov decision process where

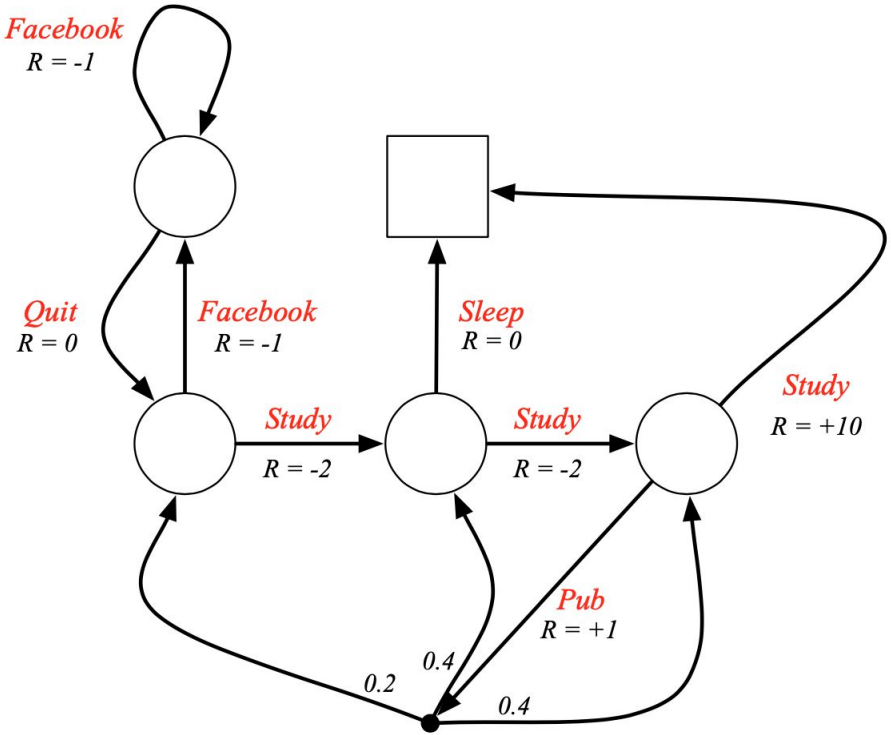
- *S* is the set of **states**, $S = \{ S_t , t = 0, 1, 2, ... \}$
- *A* is the set of **actions** $A = \{ A_t, t = 0, 1, 2, \dots \}$
- *p* is the **transition function** $p(s', r \mid s, a) \stackrel{def}{=} Pr \{ s' = S_t , r = R_t \mid s = S_{t-1} , a = A_t \}$ where R_t is the return (a real number) at time step t
- γ is the discount factor for computing **return**: $G_t \stackrel{def}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$



From Barto & Sutton: Introduction to Reinforcement Learning, 2nd Edition

In reinforcement learning the environment is usually assumed (either explicitly or implicitly) to operate like a Markov decision process.

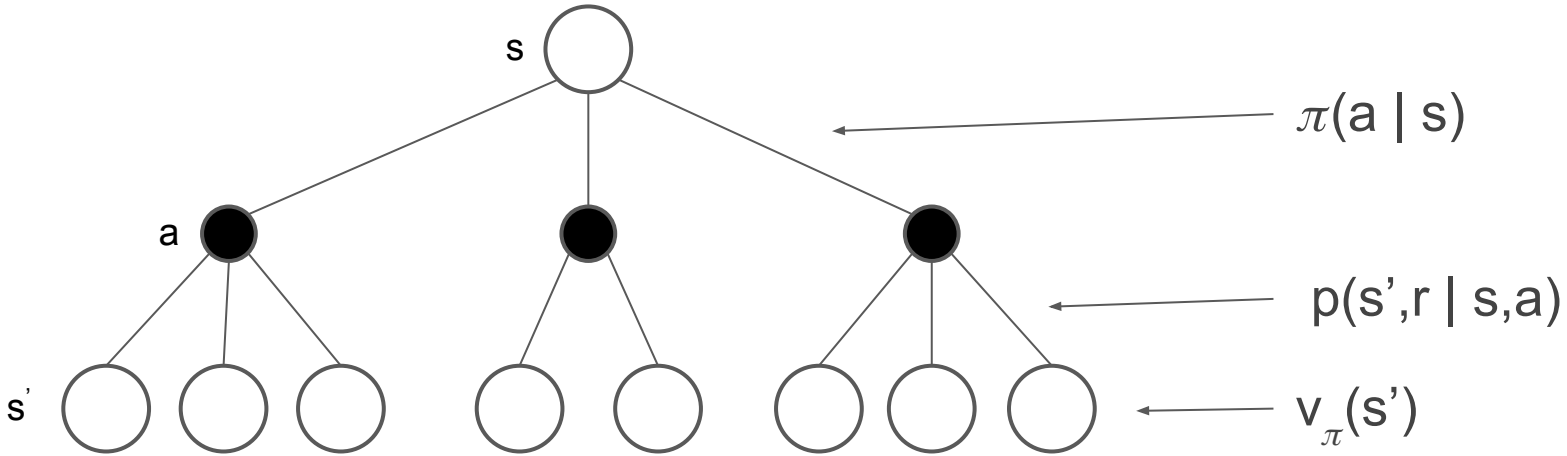
MDP example: Student life



(Here reward depends only on the action taken, not on the outcome of the action. This simplifies specification and speeds up learning; in the end it is a modeling choice.)

From Reinforcement Learning course slides by David Silver (https://www.davidsilver.uk/teaching/)

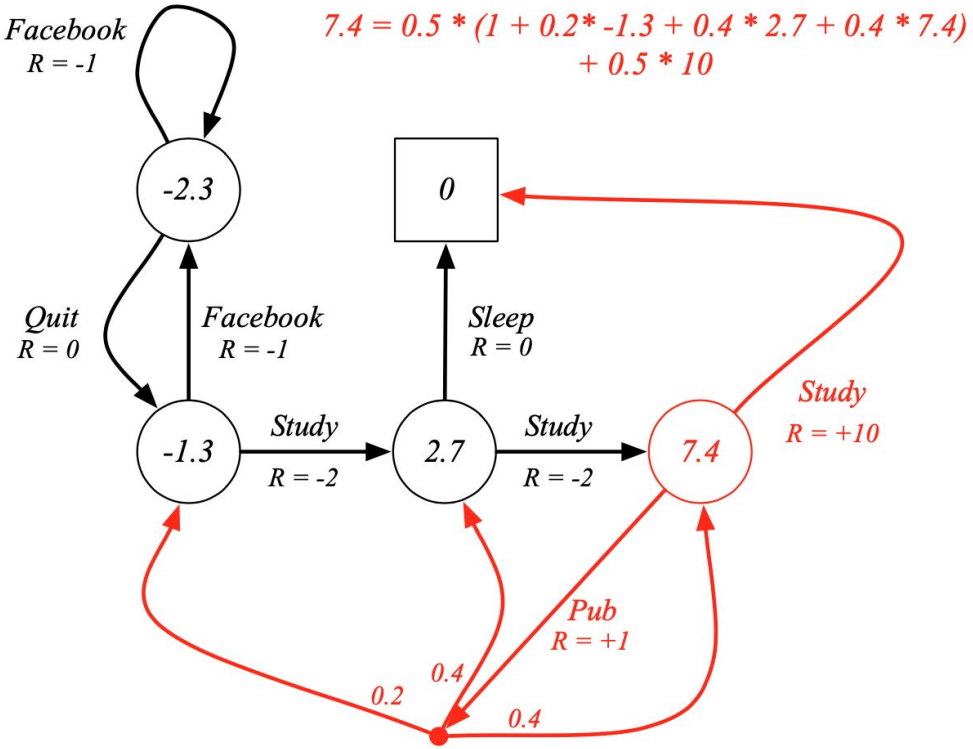
Backup diagram for state-value function



$v_\pi(s) = \sum_a [\pi(a \mid s) \sum_{s',r} p(s',r \mid s,a)(r + \gamma v_\pi(s'))]$

(Bellman equation for state-value function)

Bellman expectation equation example



Policy evaluation

Target: compute the state-value function for a given policy.

Idea: let's turn the Bellman equation for state-values into iteration:

Compute new state values (for all states) using state-values from previous iteration in the right-hand side of Bellman expectation equation. Start iteration from an initial function, for example v(s) = 0 for all s:

$$\mathbf{v_{k+1}(s)} = \sum_{a \in A} \pi(a \mid s) \left[r(a, s) + \gamma \sum_{s' \in S} (p(a, s, s') \mathbf{v_k(s')}) \right]$$

Here v is a function from state-vector to state-vector.

Policy evaluation

0.25 * (-1 + (-2)) +
0.25 * (- 1 + (-2)) +
0.25 * (- 1 + (-2)) +
0.25 * (- 1 + (-1.75)) =
-2.9375

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

Limited precision:
1.7 in (1,2) is
actually 1.75

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

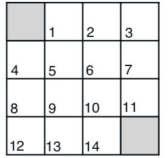
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Upper left corner and lower right corner are the end states (no actions taken there).

Actions are movements to N,E,S,W - if not possible to move, will stay in place.

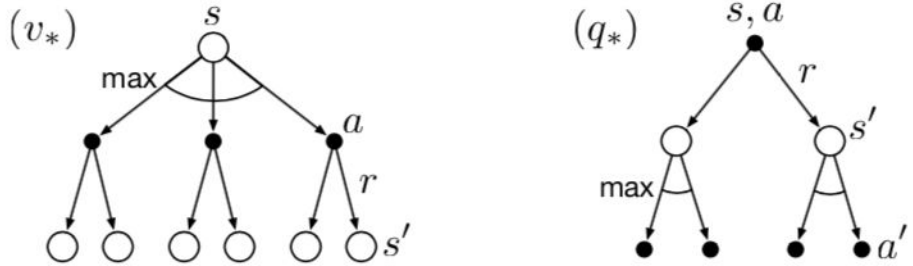
Reward is -1 for all actions, and discount factor = 1. All actions are deterministic.

Policy is random movement N,E,S,W with equal probabilities.



$R_t = -1$
on all transitions

Bellman optimality equations



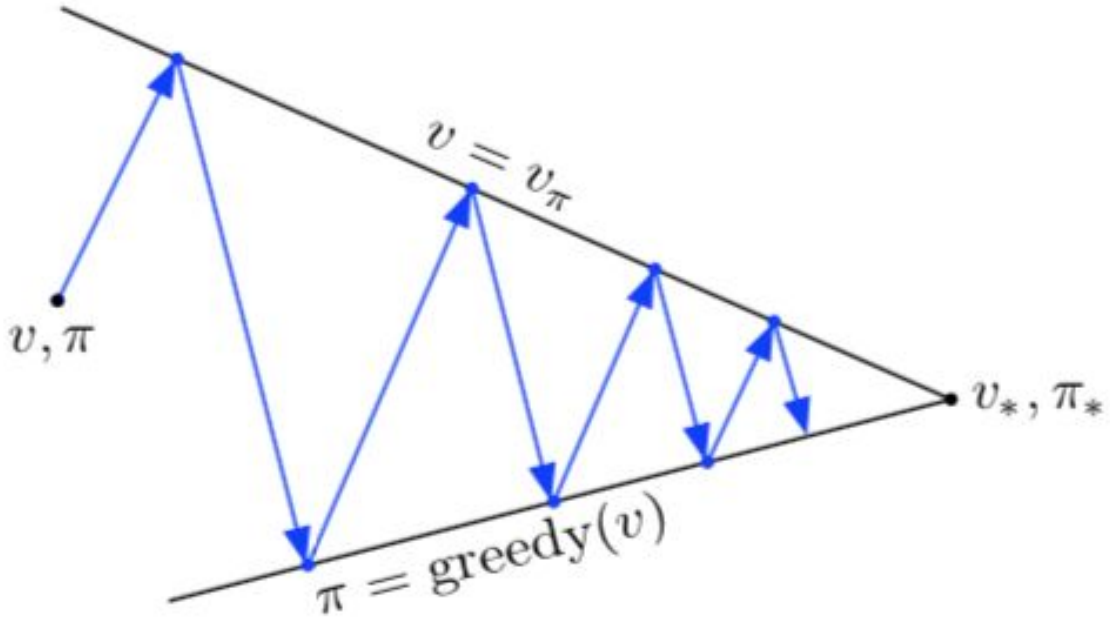
$v_*(s)$ = $\max_{a \text{ in } A(s)} q_*(s,a)$

$q_*(s,a)$ = $r + \gamma \sum_{s' \in S}(p(a, s, s') \max_{a' \text{ in } A(s')} q_*(s',a'))$

Bellman optimality equations are non-linear - there are no general (analytical) solutions.

Iterative methods must be used, we'll take a look at **policy iteration**. (There are more efficient methods like Q learning and Sarsa).

Policy iteration



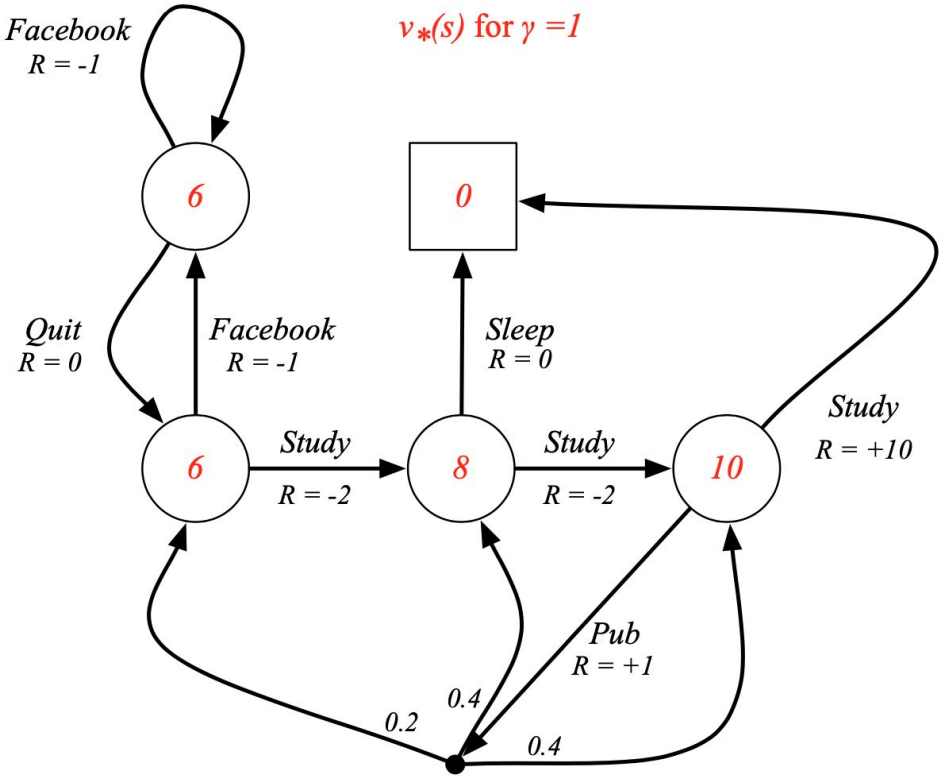
Idea of policy iteration:

Starting point: policy π (can be for example random uniform)

Repeat:

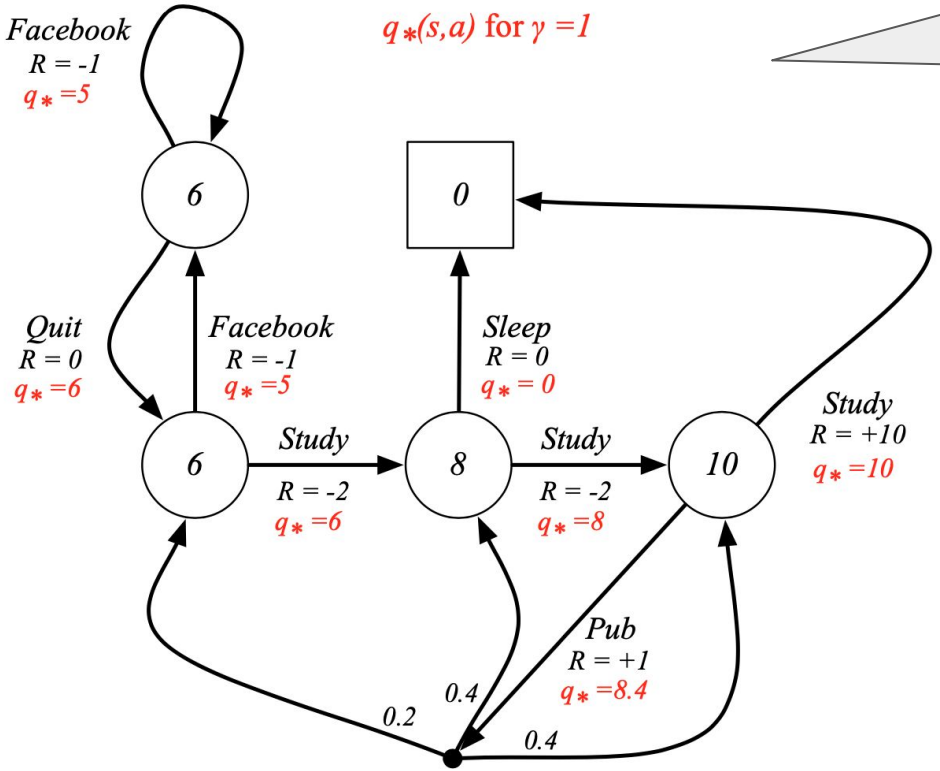
- **Evaluate** policy: compute the v corresponding to policy π
- Define new policy π: act **greedily** on v, ie. at each state s select the action with highest expected return

Optimal state-value function example

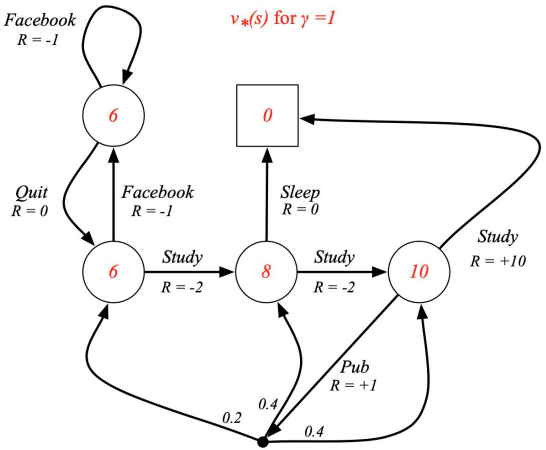


State-value function when acting according to optimal policy.

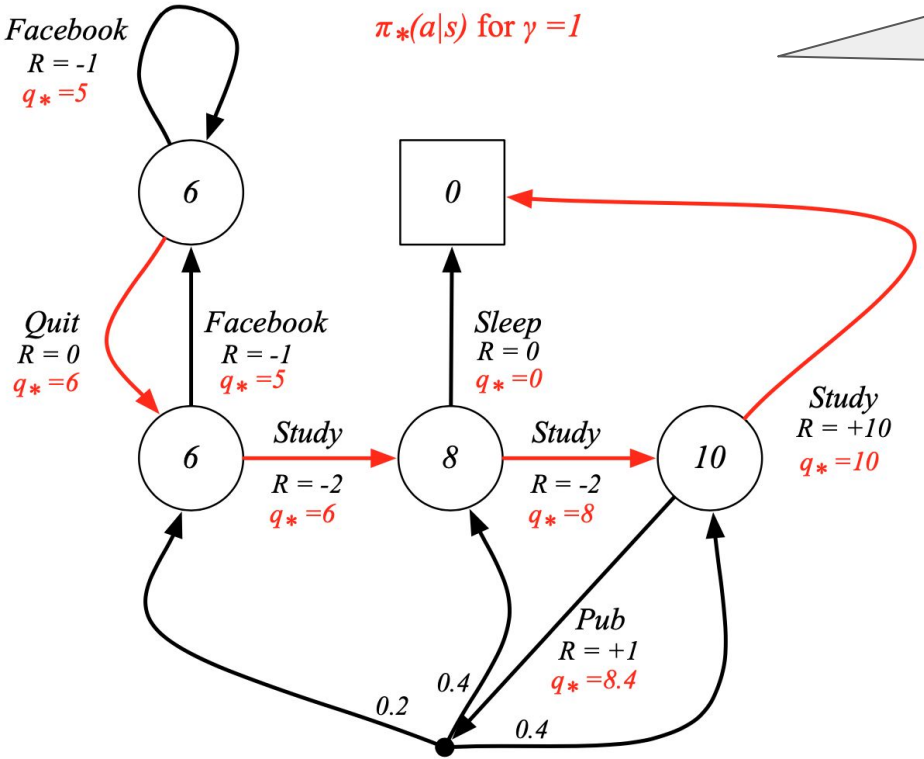
Optimal action-value function



At each state compute over all actions sum of reward and the optimal state-value function.



Optimal policy



At each state select the action with the highest action-value function.

Monte Carlo action value estimation

Reminder: action value function q(s,a) is the value of taking action a plus the expected value of being in state s (under some policy).

To estimate q(s,a) instead of v(s) the immediate reward + future value estimate needs to be computed for each (state, action) pair. If there are n states, and m possible actions in each state, this means that n * m values need to computed.

The computation for a single (s,a) pair is very similar to that for a single state s.

However, we now have even more values to be estimated than in value function estimation - how to make sure we find all of them?

ϵ -greedy exploration

Assume *m* actions are possible at state *s*, then $a^* = argmax_{a \text{ in } Actions(s)}\ q(s, a)$ is the best action, the one that would be selected when acting greedily.

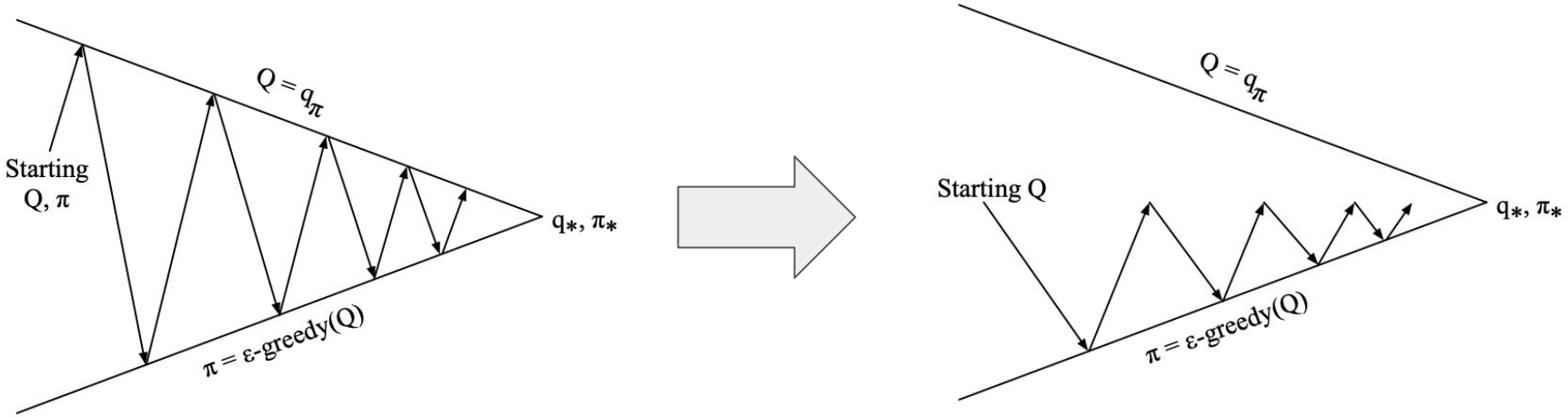
In ϵ -greedy exploration **a random action is selected with probability ϵ and the greedy action a^* with probability 1 - ϵ**. The policy will look like:

$\pi(a \mid s) = \epsilon / m + 1 - \epsilon$, if $a = a^*$

$\pi(a \mid s) = \epsilon / m$, if $a \neq a^*$

where m is the number of actions available in state s. (Note: often the policy itself is not explicitly needed.)

Monte Carlo policy iteration



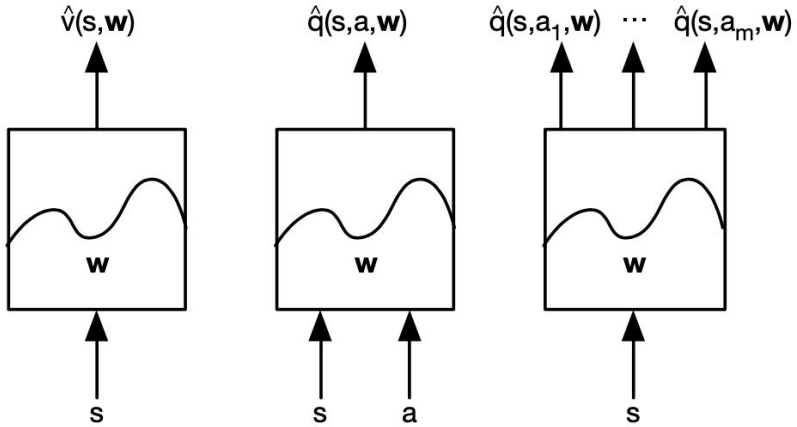
No need to fully evaluate policy before improving it - improve after each episodic sample.

Function approximators

To overcome the state space size problem, v and q functions can be approximated with

- Linear, or other shallow models
- In more complicated cases by neural networks

The idea is to train a (parametric) model that represents the value function:



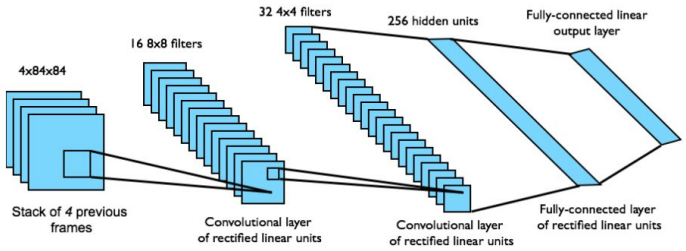
Value function approximation - batch

In iterative methods the samples collected are discarded after they are used.

Store experience in replay memory. Sample a mini-batch from experience and use stochastic gradient descent to adjust the function approximator parameters.

This approach is used in DQN (Atari games system):

- State s is 4 frames of raw pixels (the game screen)
- A convnet is trained to predict q function, ie. value for each action (up,down,shoot, etc)



Summary

Agent and environment: interaction with actions, rewards, and observations (states).

MDP for describing the dynamics, state-value function v and action-value function q for the MDP with given policy.

Learning optimal policy when MDP is given.

Learning optimal policy in model-free way with Monte Carlo.

Using deep nets to approximate value functions.