Artificial Neural Networks

Lecture 7: Recurrent Nets

Tudor Berariu

tudor.berariu@gmail.com

Faculty of Automatic Control and Computers

University Politehnica of Bucharest

Lecture : 2nd of December, 2015

Last Updated: 2nd of December, 2015

Course Progress

1. Introduction to Artificial Neural Networks
2. Linear Discriminants; The Perceptron Algorithm
3. Feedforward Neural Networks; Backpropagation
4. Optimization Algorithms
5. Convolutional Neural Networks
6. Radial Basis Function Networks
7. Reinforcement Learning; **Recurrent Neural Networks**

Today's Outline

Today's Outline

Today's Outline

Types of machine learning problems

Definition

Problems in which training data comprising of input-target pairs is available are called supervised learning problems.

Definition

Problems in which training data consists of input vectors without any target labels are called unsupervised learning problems.

Definition

Problems in which an agent learns actions to take in order to maximize a [long-term] reward are known as reinforcement learning problems.

Reinforcement Learning

- an agent acts in an environment
- occasionally he gets some reward (positive or negative)

Reinforcement Learning

- an agent acts in an environment
- occasionally he gets some reward (positive or negative)
- the goal: learn what action to take in each state in order to maximize the rewards

$$\text{maximize} \quad R \tag{1}$$

Reinforcement Learning

- an agent acts in an environment
- occasionally he gets some reward (positive or negative)
- the goal: learn what action to take in each state in order to maximize the long-term rewards

$$\text{maximize} \quad \sum_{t=0}^{\infty} \quad R(t) \tag{1}$$

Reinforcement Learning

- an agent acts in an environment
- occasionally he gets some reward (positive or negative)
- the goal: learn what action to take in each state in order to maximize the long-term discounted rewards

$$\text{maximize} \quad \sum_{t=0}^{\infty} \gamma^t \cdot R(t) \tag{1}$$

γ - discount factor ($0 \leq \gamma \leq 1$)

Reinforcement Learning

- an agent acts in an environment
- occasionally he gets some reward (positive or negative)
- the goal: learn what action to take in each state in order to maximize the long-term discounted rewards

$$\text{maximize} \quad \sum_{t=0}^{\infty} \gamma^t \cdot R(t) \tag{1}$$

γ - discount factor ($0 \leq \gamma \leq 1$)

- A function that returns the action to perform in a given state is called a policy:

$$\pi : \mathcal{S} \longrightarrow \mathcal{A} \tag{2}$$

Utilities and optimal policies

- The expected utility of [being in] a state:

$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{t} R(s_t)\right] \tag{3}$$

  where the distribution over $s_1, s_2, \ldots, s_t, \ldots$ is determined by s_0, π and by the environmnet.

- The learning objective: **the optimal policy**

$$\pi^* = \underset{\pi}{\operatorname{argmax}} U^{\pi}(s_0) \tag{4}$$

Reinforcement Learning Problems

- How much do we know about the environment?
  - deterministic vs. stochastic
  - observable vs. partially observable
  - known vs. unknown transition model

Today's Outline

MDPs

Definition

A sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a **Markov decision process** and consists of:

- a set of states $\mathcal{S}$ (with an initial state s_0)
- a set of actions for each state $A : \mathcal{S} \rightarrow \mathcal{A}$
- a transition model $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$
- a reward function $R : \mathcal{S} \rightarrow \mathbb{R}$

[RN09]

Example of an MDP

Consider a robot that navigates on a 2×3 grid map.

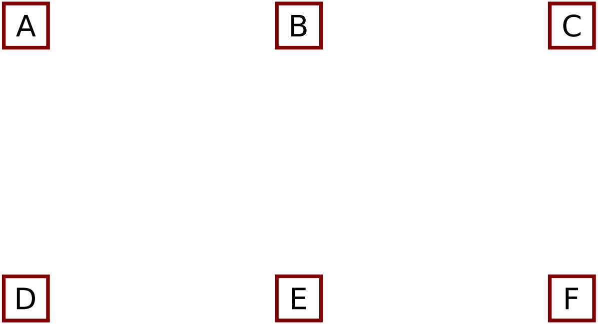
Rewards:

0	0	0
0	-10	10

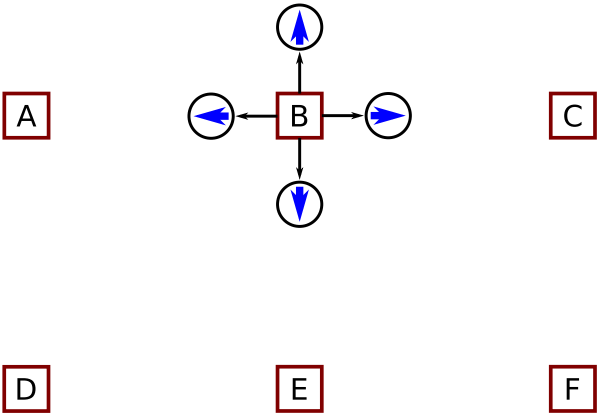
Type of states:

START	0	0
0	FINAL	FINAL

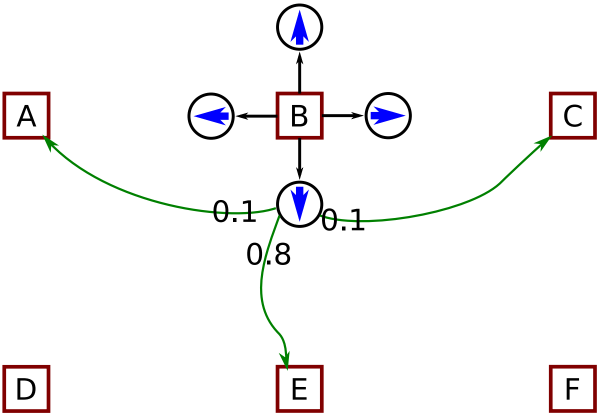
MDP Example



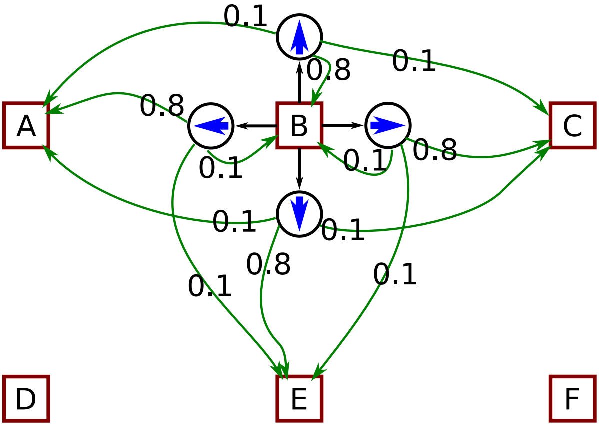
MDP Example



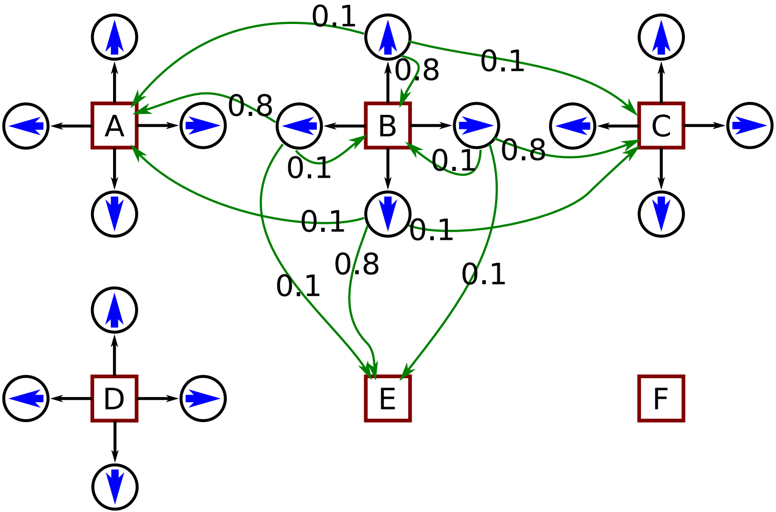
MDP Example



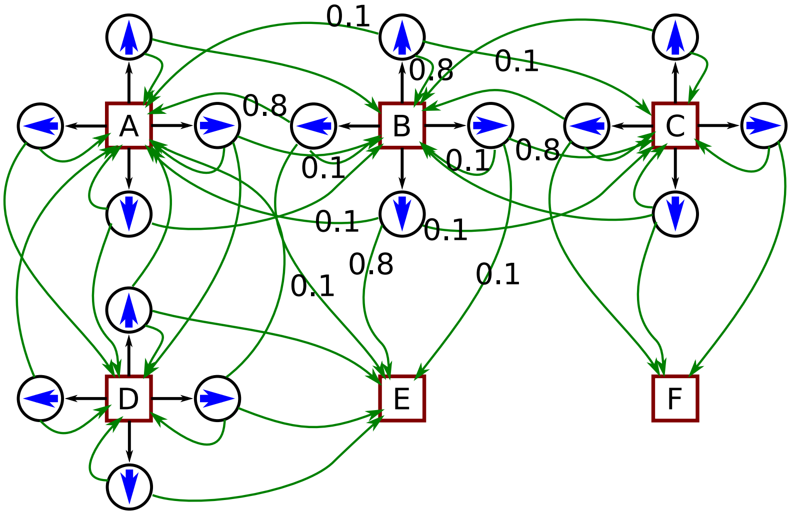
MDP Example



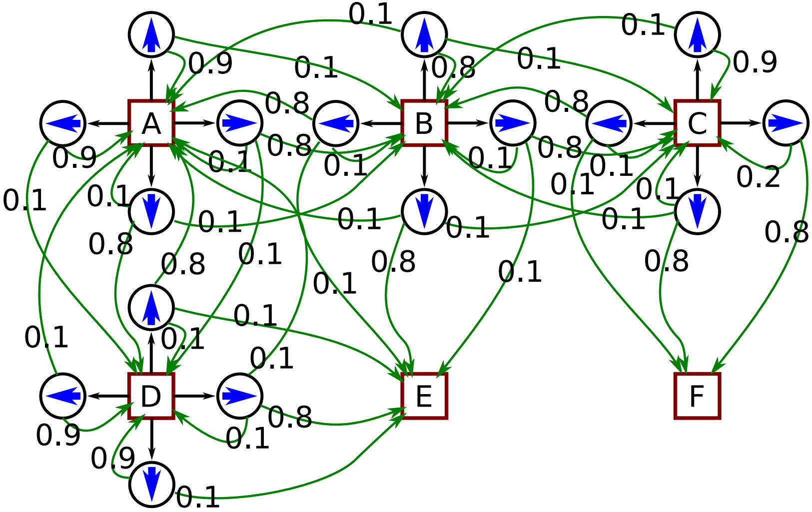
MDP Example



MDP Example



MDP Example


A
B
C
D
E
F
0.1
0.1
0.9
0.1
0.8
0.1
0.9
0.8
0.8
0.9
0.1
0.8
0.1
0.1
0.8
0.1
0.1
0.1
0.1
0.1
0.1
0.2
0.1
0.1
0.1
0.1
0.8
0.8
0.8
0.1
0.8
0.1
0.8
0.1
0.1
0.1
0.1
0.1
0.1
0.8
0.9
0.1
0.9
0.1

Optimal policy

$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{t} \cdot R(s_t)\right] \tag{5}$$

Optimal policy:

$$\begin{aligned} \pi_s^* &= \underset{\pi}{\operatorname{argmax}} U^{\pi}(s) && (6) \\ &= \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} \sum_{s_{next}} P(s_{next} | s, a) \cdot U(s_{next}) && (7) \end{aligned}$$

Bellman Equations

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s_{next} \in \mathcal{S}} P(s_{next} | s, a) \cdot U(s_{next}) \tag{8}$$

- $|\mathcal{S}|$ equations with $|\mathcal{S}|$ unknown variables $U(s)$, but... not easy to solve because of the max function
- an iterative method might be applied

Value Iteration

Algorithm 1 Value Iteration

procedure $\textsc{ValueIteration}(\text{MDP } \langle \mathcal{S}, A, P, R \rangle,\ \gamma)$
    for $s \in \mathcal{S}$ **do**
        $U(s) \leftarrow 0$
    repeat
        $U_{old} \leftarrow U$
        $\delta \leftarrow 0$
        for $s \in \mathcal{S}$ **do**
            $U(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s_{next}} P(s_{next} | s, a) U_{old}(s_{next})$
            $\delta \leftarrow \max(\delta, |U(s) - U_{old}(s)|)$
    until $\delta < \epsilon$
    return U

Value Iteration in Lua

```lua
local U = torch.zeros(2, 3)
repeat
   local Uold = U:clone()
   for row = 1, height do
      for col = 1, width do
         Umax = torch.cmul(Uold, P[row][col][1]):sum()
         for action = 2,4 do
            Ua = torch.cmul(Uold, P[row][col][action]):sum()
            Umax = math.max(Umax, Ua)
         end -- for action
         U[row][col] = R[row][col] + discount * Umax
      end -- col
   end -- row
local delta = (Uold - U):abs():max()
until delta < 0.001
```

The Optimal Policy - Implementation

```
local policy = torch.Tensor(2, 3)
for row = 1, height do
   for col = 1, width do
      Umax = torch.cmul(U, P[row][col][1]):sum()
      best_action = 1
      for action = 2, 4 do -- there are 4 actions: N, E, S, W
         Ua = torch.cmul(U, P[row][col][action]):sum()
         if Ua > Umax then
             Umax = Ua
             best_action = action
         end -- if
      end -- for action
      policy[row][col] = best_action
   end -- for col
end -- for row
```

Full code here:

`github.com/tudor-berariu/machine_learning_torch_hacks/blob/master/miscellaneous/value_iteration.lua`

Convergence

Theorem

The **value iteration** *algorithm eventually converges to the unique solutions of the Bellman equations if* $\gamma < 1$.

See [RN09, Section 17.2.3] for a complete proof using contractions.

Utility-based agents for real problems?

- an utility-based agent **must** have a model of the environment
- for Partially Observable MDPs, see [RN09, Section 17.4]

- what if P is not available?

Today's Outline

1. Reinforcement Learning
   - Introduction
   - RL in Known Environments
   - RL in Unknown Environments
   - Deep Reinforcement Learning
2. Analysing sequences
3. Recurrent Neural Networks
4. Long Short-Term Memory

RL in Unknown Environments

- What if P (the transition model) is not available?
  1. estimate it: increment $N_{actions}[s_1, a]$ and $N_{outcomes}[s_1, a, s_2]$ each time the agent applies action a in state s_1 and the outcome state is s_2

$$P(s_2|s, a) \leftarrow \frac{N_{outcomes}[s_1, a, s_2]}{N_{actions}[s_1, a]}$$

  2. learn (state,action) utilities Q instead of state-utilities U using temporal-difference methods

$$U(s) = \max_{a \in A(s)} Q(s, a)$$

Temporal Difference Learning

- slightly adjust the utility estimates towards the ideal equilibrium given by:
$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A(s')} Q(s',a') \tag{9}$$
- update equation for TD Q-learning:
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R(s) + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a) \right) \tag{10}$$
every time the agent reaches s' as an outcome of applying a in s
- Q-Learning is *model-free*

Q-Learning

Algorithm 2 Q-Learning

procedure $\textsc{Q-Learning}(\mathcal{S}, s_0, A, R, \gamma, \alpha, \epsilon)$
    for each episode **do**
        $s \leftarrow s_0$
        while s is not final **do**
            $a \leftarrow \epsilon\texttt{-Greedy}(Q, s, A, \epsilon)$
            $s' \leftarrow \texttt{ApplyAction}(s, a)$
            $r' \leftarrow R(s')$
            $Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha \left(r' + \gamma \max_{a' \in A(s')} Q(s', a') \right)$
            $s \leftarrow s'$
    return Q

Exploration vs. exploitation

- explore with probability ϵ, and exploit best known action with probability $1 - \epsilon$

Algorithm 3 ϵ-greedy

```
procedure ε-Greedy(Q, s, A, ε)
    if rand() < ε then
        return choice(A(s))
    else
        return argmax_{a∈A(s)} Q(s, a)
```

Offline learning vs Online learning

- SARSA algorithm [RN94] [SS96]:

$$\begin{aligned} Q_{t+1}(s_t, a_t) \quad \longleftarrow \quad & Q_t(s_t, a_t) + \\ & \alpha_t \left(R(s_{t+1}) + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \right) \end{aligned}$$

About online learning

[...] *perhaps it would be better to learn the policy that is optimal, given that you will explore* ϵ *of the time. It would be like a person who, when walking, always takes a random step every 100 paces or so. Such a person would avoid walking along the top of a cliff, even when that is the optimal policy for a person who doesn't explore randomly.* [BI99]

Today's Outline

Why not deep reinforcement learning?

- Unsupervised learning : deep belief networks
- Supervised learning : convolutional networks, and others...

Why not deep reinforcement learning?

- Unsupervised learning : deep belief networks
- Supervised learning : convolutional networks, and others...

- Why not **deep** reinforcement learning?

Function approximation

- for real problems the state space is huge (Go: 10^{170}) or continuous
- Q-Learning (and all other algorithms) provide no generalization
  - it uses a lookup table for the Q values
  - there is a Q-value for each possible state-action pair
  - Q-learning needs hand-crafted features or a reasonable state space

Function approximation

- for real problems the state space is huge (Go: 10^{170}) or continuous
- Q-Learning (and all other algorithms) provide no generalization
  - it uses a lookup table for the Q values
  - there is a Q-value for each possible state-action pair
  - Q-learning needs hand-crafted features or a reasonable state space
- **function approximation**:

$$\hat{Q}(s, a, \mathbf{w}) \approx Q(s, a) \tag{11}$$

  - linear combinations of features
  - neural networks [Rie05]
  - decision trees
  - nearest neighbour

Function approximation

- for real problems the state space is huge (Go: 10^{170}) or continuous
- Q-Learning (and all other algorithms) provide no generalization
  - it uses a lookup table for the Q values
  - there is a Q-value for each possible state-action pair
  - Q-learning needs hand-crafted features or a reasonable state space
- **function approximation**:

$$\hat{Q}(s, a, \mathbf{w}) \approx Q(s, a) \tag{11}$$

  - linear combinations of features - differentiable
  - neural networks [Rie05] - differentiable
  - decision trees
  - nearest neighbour

Using function approximation

- method:
  - use gradient descent to minimize a loss function

$$E(\mathbf{w}) = \mathbb{E}_\pi \left[\left(Q_\pi(s,a) - \hat{Q}(s,a,\mathbf{w}) \right)^2 \right] \tag{12}$$

- problems:
  - learning algorithms for NN assume i.i.d. data
  - consecutive states in rl scenarios are strongly correlated
  - the policy might change and suddenly change the state distribution
  - output values and target values are correlated
  - the learning algorithm does not converge [TVR97]

Playing Atari with Deep Reinforcement Learning

This section is based on this article from February 2015:

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al., *Human-level control through deep reinforcement learning*, Nature **518** (2015), no. 7540, 529–533

DQN

- the network learns to output $Q(s, a)$ from matrix of pixels s
- s is composed of the last 4 frames
- output is action-utility $Q(s, a)$ for 18 commands
- reward is change in score

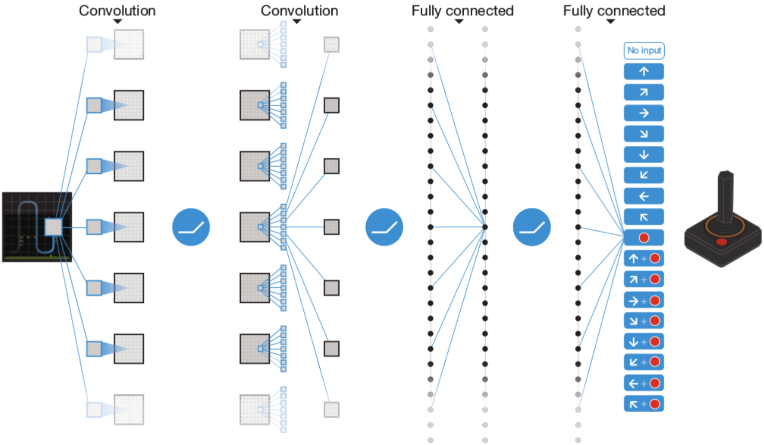
Why does it work?

- **experience replay** removes correlations from the observed sequence
  - store experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset D_t
  - update using the gradient for:

$$\mathbb{E}_{(s,a,r,s')\tilde{U}(D)} \left[\left(r + \gamma Q(s', a', \mathbf{W}^{-}) - Q(s, a, \mathbf{W}) \right)^2 \right]$$

- Q values are **only periodically updated**, removing the correlation between target values and outputs

The Deep Neural Network



Training details

- inputs: 4 consecutive screens rescaled it to 84x84
- trained with RMSProp[1] for mini-batches of 32
- ϵ-greedy with ϵ from 1.0 to 0.1 in the first 1000000 games, fixed afterwards
- training set: 50 million frames (39 days of experiences)

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

rmsprop

- rprop (using just the gradient sign) had good results, but did not work with mini-batches
- rmsprop tries to offer both advantages
- divide the gradient by a running average of its recent magnitude
- rmsprop (Tijmen Tieleman) computes a moving average of the squared gradient for each weight:

$$ms_i^{(\tau)} = 0.9 \cdot ms_i^{(\tau-1)} + 0.1\left(\frac{\partial E}{\partial w_i^{(\tau)}}\right)^2 \tag{13}$$

rmsprop

- rprop (using just the gradient sign) had good results, but did not work with mini-batches
- rmsprop tries to offer both advantages
- divide the gradient by a running average of its recent magnitude
- rmsprop (Tijmen Tieleman) computes a moving average of the squared gradient for each weight:

$$ms_i^{(\tau)} = 0.9 \cdot ms_i^{(\tau-1)} + 0.1 \left(\frac{\partial E}{\partial w_i^{(\tau)}} \right)^2 \tag{13}$$

$$\Delta w_i^{(\tau)} = -\eta \cdot \frac{1}{\sqrt{ms_i^{(\tau)}}} \cdot \frac{\partial E}{\partial w_i^{(\tau)}} \tag{14}$$

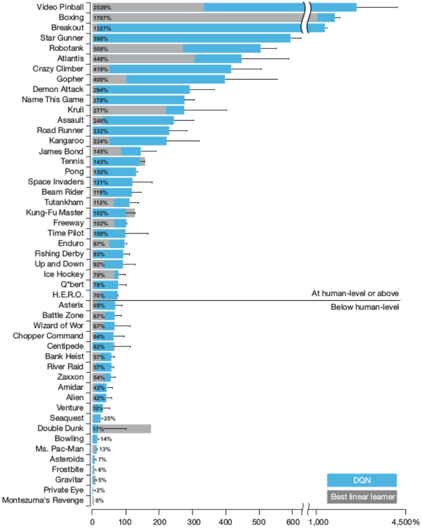
Learning algorithm

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

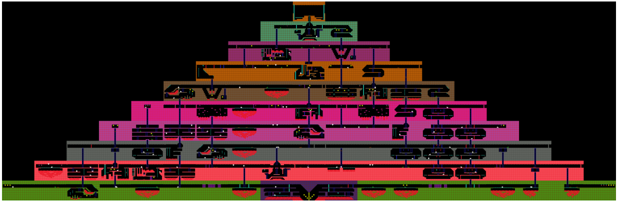
Results on Atari 2600 games

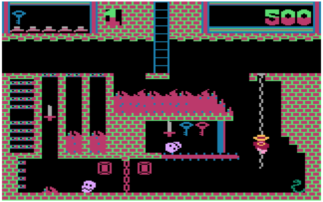
- DQN better than all previous algorithms on most of the games
- DQN comparable with a professional game tester using the same algorithm, network architecture and meta-parameters

Results on Atari



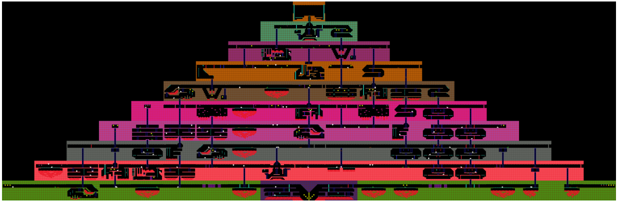
No good results on some of the games

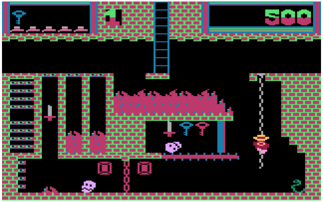




screens from Montezuma's Revenge

No good results on some of the games





screens from Montezuma's Revenge

you need to remember the past in order to predict the future

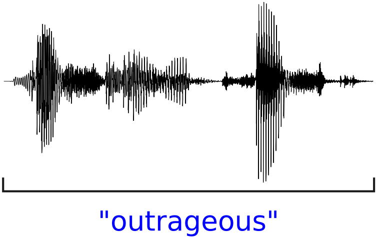
Today's Outline

Problems involving sequences

- sequence labelling
- next value prediction
- system control loops

Different classification problems

- **Sequence classification**
  - each input sequence is assigned to a single label
  - examples: recognizing a single spoken word



Different classification problems

- Sequence classification
  - each input sequence is assigned to a single label
  - examples: recognizing a single spoken word
- **Segment classification**
  - each segment is assigned to a label
  - segments are known in advance
  - examples: part-of-speech tagging

heat water in a large vessel

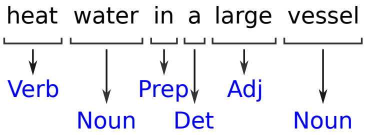
Different classification problems

- Sequence classification
  - each input sequence is assigned to a single label
  - examples: recognizing a single spoken word
- **Segment classification**
  - each segment is assigned to a label
  - segments are known in advance
  - examples: part-of-speech tagging

heat water in a large vessel

Different classification problems

- Sequence classification
  - each input sequence is assigned to a single label
  - examples: recognizing a single spoken word
- **Segment classification**
  - each segment is assigned to a label
  - segments are known in advance
  - examples: part-of-speech tagging

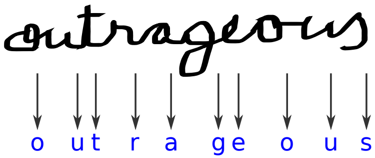


Different classification problems

- Sequence classification
  - each input sequence is assigned to a single label
  - examples: recognizing a single spoken word
- Segment classification
  - each segment is assigned to a label
  - segments are known in advance
  - examples: part-of-speech tagging
- **Temporal classification**
  - segments are not knwon *a priori*
  - examples: handwritten text, activity recognition

Different classification problems

- Sequence classification
  - each input sequence is assigned to a single label
  - examples: recognizing a single spoken word
- Segment classification
  - each segment is assigned to a label
  - segments are known in advance
  - examples: part-of-speech tagging
- **Temporal classification**
  - segments are not knwon *a priori*
  - examples: handwritten text, activity recognition



Context

- Models used to analyze sequences:
  - Linear Dynamical Systems
  - Hidden Markov Models
  - **Recurrent Neural Networks**
- These models use information about the past
- *internal memory*

Today's Outline

Recurrent Neural Networks

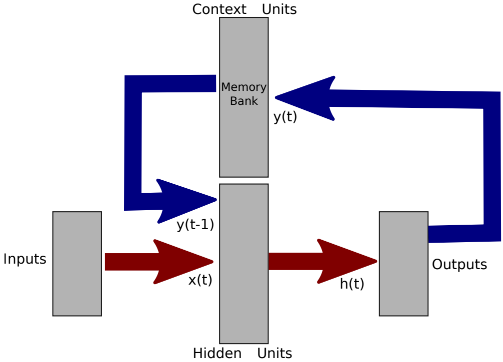
Definition

Recurrent Neural Networks (RNNs) are artificial neural networks with **cyclical connections**. In a RNN the current output of some unit might influence a future state the same unit.

- Feedforward neural networks map examples from the input space to vectors in the output space.
- RNNs map history of all previous inputs to vectors in the output space.

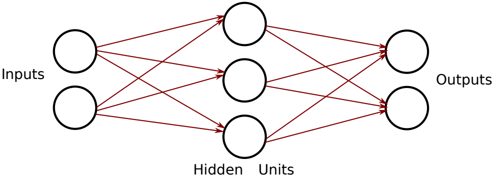
Varieties of RNN Architectures

- Jordan Networks [Jor86]



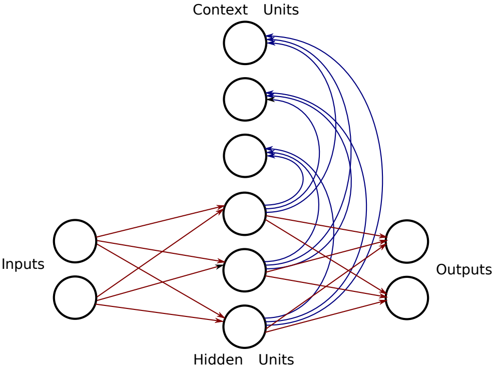
Varieties of RNN Architectures

- Jordan Networks [Jor86]
- Elman Networks [Elm90]



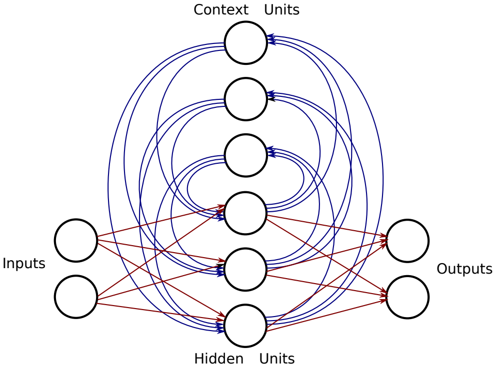
Varieties of RNN Architectures

- Jordan Networks [Jor86]
- Elman Networks [Elm90]



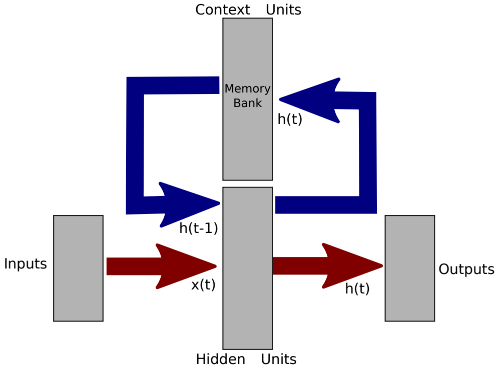
Varieties of RNN Architectures

- Jordan Networks [Jor86]
- Elman Networks [Elm90]



Varieties of RNN Architectures

- Jordan Networks [Jor86]
- Elman Networks [Elm90]



Varieties of RNN Architectures

- Jordan Networks [Jor86]
- Elman Networks [Elm90]
- Time Delay Neural Networks
- Echo State Networks [JH04]
- LSTM [HS97]
- Hopfield Networks

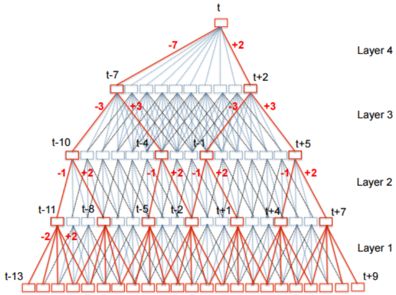
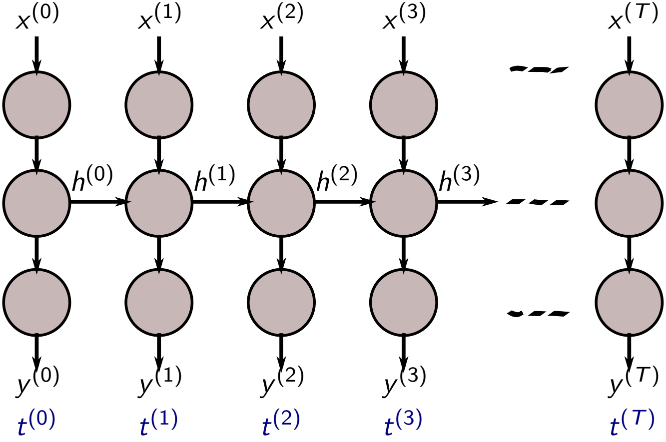


Figure 1: Computation in TDNN with sub-sampling (red) and without sub-sampling (blue+red)

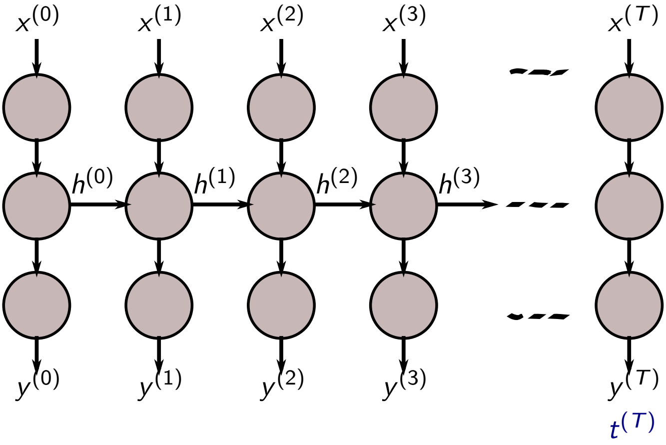
Today's Outline

Error function - Continuous output



$$E = \sum_{\tau=0}^{T} |y^{\tau} - t^{\tau}|^2$$

Error function - Sequence labelling



$$E = \sum_k t_k^T \log y_k^T$$

Learning Algorithms

- remember from FNN: *batch* learning vs. *stochastic* learning
- likewise:
  1. epochwise training
  2. continuous training - suitable for *on-line* learning

Learning Algorithms

- remember from FNN: *batch* learning vs. *stochastic* learning
- likewise:
  1. epochwise training
  2. continuous training - suitable for *on-line* learning

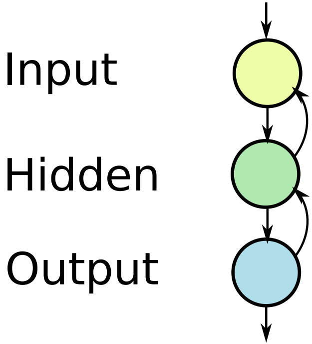
- computing the derivatives of the loss function with respect to the weights:
  - Real Time Reccurent Learning [RF87]
  - Backpropagation Through Time [WZ95]

Today's Outline

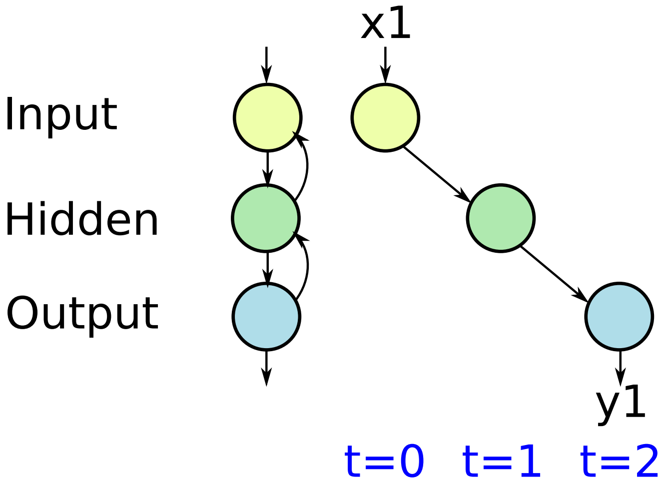
Backpropagation through time

- it is an extension of the classic *backpropagation algorithm*
- idea: *unfold* the RNN into a FNN whose size grows at every time-step

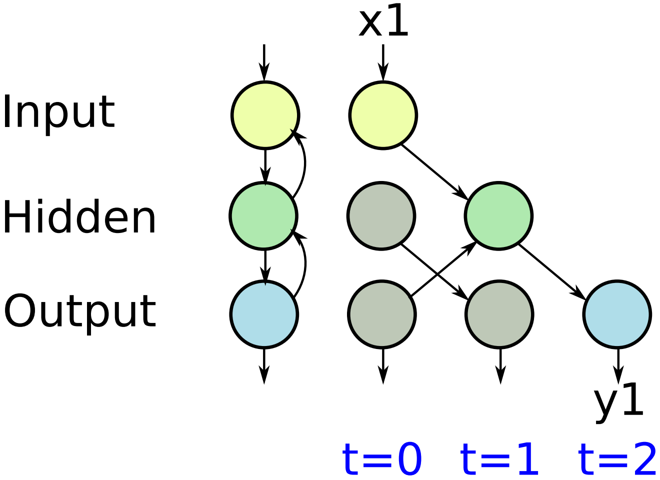
Unfolding RNNs



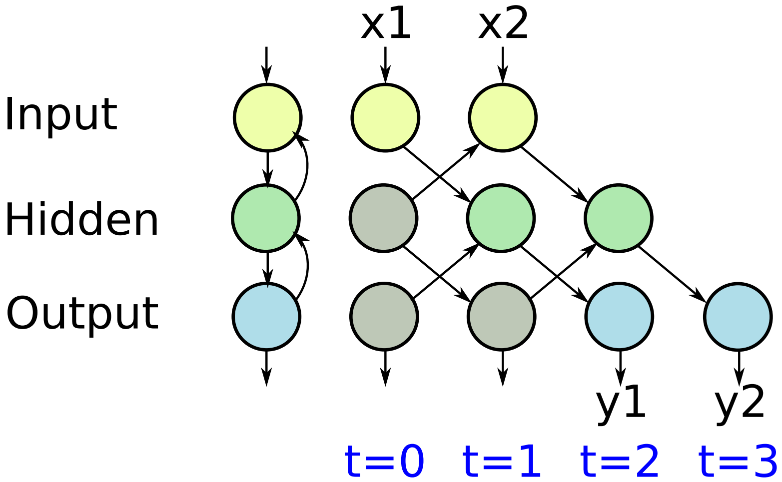
Unfolding RNNs



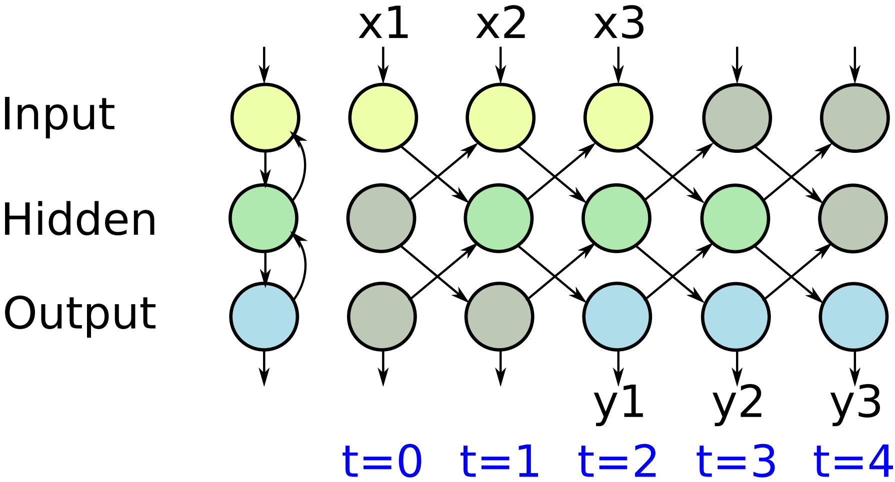
Unfolding RNNs



Unfolding RNNs



Unfolding RNNs



The Error Function

- We consider a network trained to minimize the following error:

$$E = \frac{1}{2} \sum_{t=0}^{T} \sum_{k} \left(y_k^{(t)} - t_k^{(t)} \right)^2 \tag{15}$$

Partial derivatives

- Remember from classic backpropagation:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} \tag{16}$$

Partial derivatives

- Remember from classic backpropagation:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} \tag{16}$$

- Remember that:

$$a_j = \sum_i w_{ji} \cdot z_i$$

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

Partial derivatives

- Remember from classic backpropagation:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} \tag{16}$$

- Remember that:

$$a_j = \sum_i w_{ji} \cdot z_i$$

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

- Notation (δ - errors)

$$\delta_j = \frac{\partial E_n}{\partial a_j}$$

Partial derivatives

- Remember from classic backpropagation:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} = \delta_j \cdot z_i \tag{16}$$

- Remember that:

$$a_j = \sum_i w_{ji} \cdot z_i$$

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

- Notation (δ - errors)

$$\delta_j = \frac{\partial E_n}{\partial a_j}$$

Epoch-wise backpropagation

$$\delta_j^{(t)} = \frac{\partial E}{\partial a_j^{(t)}} \tag{17}$$

Epoch-wise backpropagation

$$\delta_j^{(t)} = \frac{\partial E}{\partial a_j^{(t)}} \tag{17}$$

$$t = T \quad : \quad \delta_j^{(t)} = f'\left(a_j^{(T)}\right)\left(z_j^{(T)} - t_h^{(T)}\right) \tag{18}$$

$$t < T \quad : \quad \delta_j^{(t)} = f'\left(a_j^{(t)}\right)\left[\left(z_j^{(t)} - t_h^{(t)}\right) + \sum_k w_{jk}\delta_k^{(t+1)}\right] \tag{19}$$

Epoch-wise backpropagation

$$\delta_j^{(t)} = \frac{\partial E}{\partial a_j^{(t)}} \tag{17}$$

$$t = T \quad : \quad \delta_j^{(t)} = f'\left(a_j^{(T)}\right)\left(z_j^{(T)} - t_h^{(T)}\right) \tag{18}$$

$$t < T \quad : \quad \delta_j^{(t)} = f'\left(a_j^{(t)}\right)\left[\left(z_j^{(t)} - t_h^{(t)}\right) + \sum_k w_{jk}\delta_k^{(t+1)}\right] \tag{19}$$

Adjusting the weights (gradient descent):

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \sum_t \delta_j^{(t)} z_i^{(t-1)} \tag{20}$$

Continuous Learning

- learn *on-line*
- at each moment of time compute the error for a truncated history (look back only h steps)

$$\delta_j^\tau = \frac{\partial E}{\partial a_j^\tau} \qquad \forall t - h < \tau \leq t \tag{21}$$

Continuous Learning

- learn *on-line*
- at each moment of time compute the error for a truncated history (look back only h steps)

$$\delta_j^\tau = \frac{\partial E}{\partial a_j^\tau} \qquad \forall t - h < \tau \leq t \tag{21}$$

$$\tau = t \quad : \quad \delta_j^\tau = \frac{\partial E}{\partial a_j^\tau} \tag{22}$$

$$t - h < \tau < t \quad : \quad \delta_j^\tau = f'\left(a_j^{(\tau)}\right)\left(\sum_k w_{jk}^{(\tau)} \delta_k^{(\tau+1)}\right) \tag{23}$$

You need to keep the weights from the last h steps.

Continuous Learning

- learn *on-line*
- at each moment of time compute the error for a truncated history (look back only h steps)

$$\delta_j^\tau = \frac{\partial E}{\partial a_j^\tau} \qquad \forall t - h < \tau \leq t \tag{21}$$

$$\tau = t \quad : \quad \delta_j^\tau = \frac{\partial E}{\partial a_j^\tau} \tag{22}$$

$$t - h < \tau < t \quad : \quad \delta_j^\tau = f'\left(a_j^{(\tau)}\right)\left(\sum_k w_{jk}^{(\tau)} \delta_k^{(\tau+1)}\right) \tag{23}$$

You need to keep the weights from the last h steps.

$$\Delta w_{ji}^{(t)} = -\eta \sum_{\tau=t-h+1}^{t} \delta_j^{(\tau)} z_i \tag{24}$$

Tricks for BPTT

- BPTT is not guaranteed to converge to a local minimum
- teacher forcing
  - use target value instead of previous output if output units' states are being fed back to the network
- truncated BPTT
  - use only the last h steps

Today's Outline

Real Time Reccurent Learning (I)

- Error of output units (generalization):

$$e_k^{(t)} = \begin{cases} t_k^{(t)} - z_k^{(t)} & \text{if } t_k^{(t)} \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

- The error at some point in time t:

$$E^{(t)} = \sum_k e_k^{(t)}$$

- The error of some sequence:

$$E_{t_1 \to t_2} = \sum_{\tau=t_1}^{t_2} E^{(\tau)}$$

Real Time Reccurent Learning (II)

- The error of some sequence:

$$\nabla_{\mathbf{w}} E_{t_1 \to t+1} = \nabla_{\mathbf{w}} E_{t_1 \to t} + \nabla_{\mathbf{w}} E^{(t+1)}$$

- For a particular time step t:

$$\frac{\partial E^{(t)}}{\partial w_{ij}} = \sum_k \frac{\partial E^{(t)}}{\partial z_k^{(t)}} \frac{\partial z_k^{(t)}}{\partial w_{ij}} = \sum_k e_k^{(t)} \frac{\partial z_k^{(t)}}{\partial w_{ij}}$$

- Notation:

$$p_{k,(i,j)}^{(t)} = \frac{\partial z_k^{(t)}}{\partial w_{ij}}$$

The recursive computation of *ps*

$$\frac{\partial z_k^{(t+1)}}{\partial w_{ij}} = f'\left(a_k^{(t+1)}\right)\left[\sum_l w_{kl}\frac{\partial z_l^{(t)}}{\partial w_{ij}} + \delta_{ik} z_j^{(t)}\right] \tag{25}$$

that translates to:

$$p_{k,(i,j)}^{(t+1)} = f'\left(a_k^{(t+1)}\right)\left[\sum_l w_{kl} p_{l,(i,j)}^{(t)} + \delta_{ik} z_j^{(t)}\right] \tag{26}$$

RTRL

- previous formula (starting from $p_{k,(ij)}^{(0)} = 0$) permits computation of all $p^{(t)}$s at time t
- that provides the gradient:

$$\frac{\partial E^{(t)}}{\partial w_{ij}} = \sum_k e_k^{(t)} \frac{\partial z_k^{(t)}}{\partial w_{ij}}$$

- the drawback: large time complexity per time-step

Comparison between BPTT and RTRL

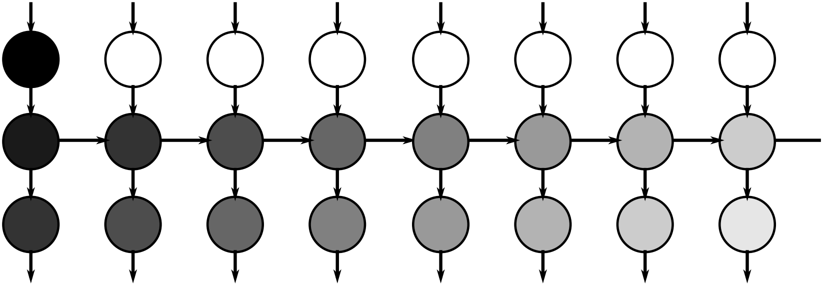
- both involve the propagation of derivatives
  - ... in the backward direction (BPTT)
  - ... in the forward direction (RTRL)
- **BPTT** requires **less computation** than RTRL does
- **RTRL** requires **less memory** thar BPTT does
- both converge very slow

Today's Outline

Vanishing / Exploding Gradient

- there is no squashing function to limit erros in the backpropagation phase
- the influence of an input either vanishes or blows up as it cycles though the network's recurrent connections
- BPTT or RTRL do not work in practice for more than 10 steps

Input Influence in a RNN



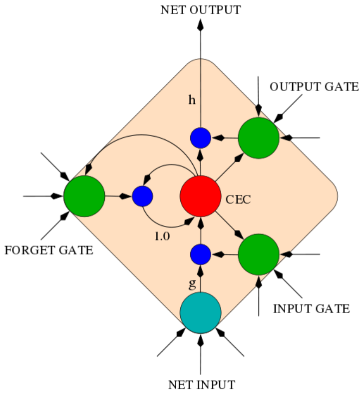
Solutions for longer sequences

- Hessian Free optimization
- LSTM
- Echo-State Networks

Today's Outline

CEC

- a LSTM block contains memory cells with 3 **multiplicative gates**:
  1. input (write) gate
  2. output (read) gate
  3. forget gate



Training LSTMs

- LSTMs are trained with classic gradient descent
- BPTT can be used to compute the gradients
- the error flow through the cells is constant

LSTM blocks

- We consider a network with
  1. I input units
  2. H hidden units
  3. C cells in a memory block
  4. K outputs
- ι refers to the input gates
- ϕ refers to the forget gate
- ω refers to the output gate
- s_c^t is the state of cell c at time t
- G - notation for the total number of inputs (as in Graves)

LSTM network

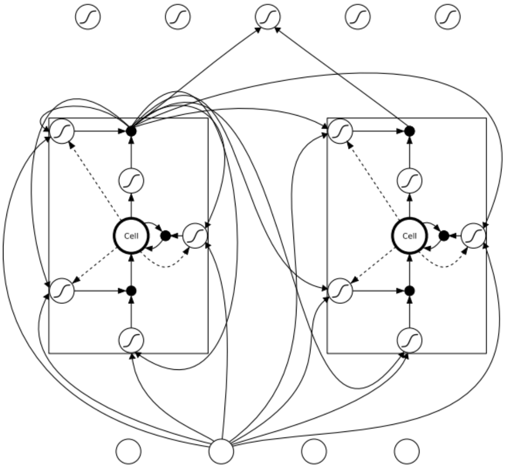


Figure 4.3: **An LSTM network.** The network consists of four input units, a hidden layer of two single-cell LSTM memory blocks and five output units. Not all connections are shown. Note that each block has four inputs but only one output.

Forward Equations

- Input Gates:

$$
\begin{aligned}
a_\iota^t &= \sum_{i=0}^{I} w_{\iota i} x_i^t + \sum_{h=0}^{H} w_{\iota h} z_h^{t-1} + \sum_{c=0}^{C} w_{\iota c} s_c^{t-1} && (27) \\
z_\iota^t &= f(a_\iota^t) && (28)
\end{aligned}
$$

- Forget Gates:

$$
\begin{aligned}
a_\phi^t &= \sum_{i=0}^{I} w_{\phi i} x_i^t + \sum_{h=0}^{H} w_{\phi h} z_h^{t-1} + \sum_{c=0}^{C} w_{\phi c} s_c^{t-1} && (29) \\
z_\phi^t &= f(a_\phi^t) && (30)
\end{aligned}
$$

f is usually the logistic function (0 for *closed*, 1 for *open*)

Forward Equations

- Cells Gates:

$$
\begin{aligned}
a_c^t &= \sum_{i=0}^{I} w_{ci} x_i^t + \sum_{h=0}^{H} w_{ch} z_h^{t-1} && (31)\\
s_c^t &= z_\phi^t s_c^{t-1} + z_\iota^t g(a_c^t) && (32)
\end{aligned}
$$

- Output Gates:

$$
\begin{aligned}
a_\omega^t &= \sum_{i=0}^{I} w_{\omega i} x_i^t + \sum_{h=0}^{H} w_{\omega h} z_h^{t-1} + \sum_{c=0}^{C} w_{\omega c} s_c^{t-1} && (33)\\
z_\omega^t &= f(a_\omega^t) && (34)
\end{aligned}
$$

g is usually the tanh function

Forward Equations

- Cell Outputs:

$$z_c^t = z_\omega^t h(s_c^t) \tag{35}$$

h is a usually the identity function (or logistic, or tanh)

Backward Pass

$$\delta_j^t \stackrel{def}{=} \frac{\partial E}{\partial a_j^t} \tag{36}$$

$$\epsilon_c^t \stackrel{def}{=} \frac{\partial E}{\partial z_c^t} \tag{37}$$

$$\epsilon_s^t = \frac{\partial E}{\partial s_c^t} \tag{38}$$

- Cell Outputs:

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck}\delta_k^t + \sum_{g=1}^{G} w_{cg}\delta_g^{t+1} \tag{39}$$

- Output Gates:

$$\delta_\omega^t = f'(a_\omega^t)\sum_{c=1}^{C} h(s_c^t)\delta_c^t \tag{40}$$

Backward Pass

- States

$$\epsilon_s^t = z_\omega^t h'(s_c^t)\epsilon_c^t + z_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t \quad (41)$$

- Cells:

$$\delta_c^t = z_\iota^t g'(a_c^t)\epsilon_s^t \quad (42)$$

Backward Pass

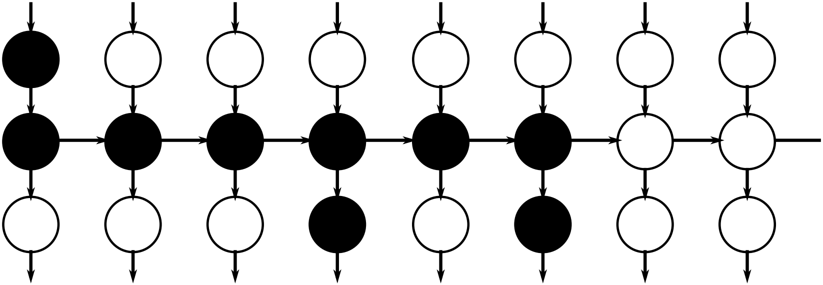
- Forget Gates:

$$\delta_{\phi}^{t} = f'(a_{\phi}^{t}) \sum_{c=1}^{C} s_{c}^{t-1})\epsilon_{s}^{t} \tag{43}$$

- Input Gates:

$$\delta_{\iota}^{t} = f'(a_{\iota}^{t}) \sum_{c=1}^{C} z_{c}^{t}\epsilon_{s}^{t} \tag{44}$$

Input Influence in LSTM



Read further about LSTM

Alex Graves et al., *Supervised sequence labelling with recurrent neural networks*, vol. 385, Springer, 2012

Today's Outline

Summary

- Reinforcement Learning techniques learn policies for long-term reward maximization without apriori knowledge.
- Combining deep learning techniques with reinforcement learning might lead to systems capable of learning from high sensory inputs as in [MKS$^+$15].
- Recurrent Neural Networks can, in principle, learn any program, but they are hard to train (main problem: the vanishing gradient).
- LSTM networks solve the problems that previous recurrent neural networks had (training for long sequences without vanishing gradient).

Today's Outline

References I

- L.C. Baird III, *Reinforcement learning through gradient descent*, Ph.D. thesis, Citeseer, 1999.
- Jeffrey L Elman, *Finding structure in time*, Cognitive science **14** (1990), no. 2, 179–211.
- Alex Graves et al., *Supervised sequence labelling with recurrent neural networks*, vol. 385, Springer, 2012.
- Sepp Hochreiter and Jürgen Schmidhuber, *Long short-term memory*, Neural computation **9** (1997), no. 8, 1735–1780.
- Herbert Jaeger and Harald Haas, *Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication*, Science **304** (2004), no. 5667, 78–80.

References II

- Michael I Jordan, *Attractor dynamics and parallellism in a connectionist sequential machine*.

- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al., *Human-level control through deep reinforcement learning*, Nature **518** (2015), no. 7540, 529–533.

- AJ Robinson and Frank Fallside, *The utility driven dynamic error propagation network*, University of Cambridge Department of Engineering, 1987.

- Martin Riedmiller, *Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method*, Machine Learning: ECML 2005, Springer, 2005, pp. 317–328.

References III

- G.A. Rummery and M. Niranjan, *On-line q-learning using connectionist systems*, University of Cambridge, Department of Engineering, 1994.
- Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach*, 3rd ed., Prentice Hall Press, Upper Saddle River, NJ, USA, 2009.
- S.P. Singh and R.S. Sutton, *Reinforcement learning with replacing eligibility traces*, Recent Advances in Reinforcement Learning (1996), 123–158.
- John N Tsitsiklis and Benjamin Van Roy, *An analysis of temporal-difference learning with function approximation*, Automatic Control, IEEE Transactions on **42** (1997), no. 5, 674–690.

References IV

Ronald J Williams and David Zipser, *Gradient-based learning algorithms for recurrent networks and their computational complexity*, Back-propagation: Theory, architectures and applications (1995), 433–486.