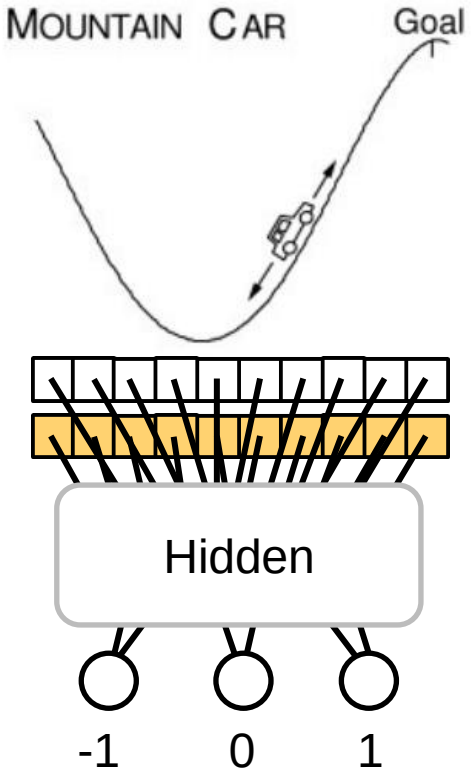
On-Line Evolutionary Computation for Reinforcement Learning in Stochastic Domains
(Whiteson and Stone 2006)

Elvir Sabic

TECHNISCHE UNIVERSITÄT DARMSTADT



Overview

TECHNISCHE UNIVERSITÄT DARMSTADT

- Introduction
- Recap: NeuroEvolution of Augmenting Topologies (NEAT)
- NEAT in Stochastic Domains
- Policy Selection Methods
- Experimental setups
  - Mountain Car
  - Server Job Scheduling
- Results
- Conclusion

Introduction

TECHNISCHE UNIVERSITÄT DARMSTADT

- Evolutionary computation is one of the most promising approaches to reinforcement learning
  - NEAT (Stanley and Miikkulainen 2002) is one approach
  - Intended for **off-line** learning scenarios

- Problem
  - Evolutionary computation is usually used in deterministic domains but **stochastical domains** are more practical

  ➔ Thus, **on-line** learning would be preferable

Introduction

TECHNISCHE UNIVERSITÄT DARMSTADT

Can we improve evolutionary computation in stochastic domains?

- Temporal difference (TD) methods are used in **on-line** learning scenarios
  - They have action selection mechanisms which consider a **balance between** ...
    - **exploration** (search for better Policies)
    - **exploitation** (accrue maximal reward)

Idea:
Integrate the TD selection mechanisms in evolutionary computation (Here: in NEAT)!

Recap: NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen 2002)

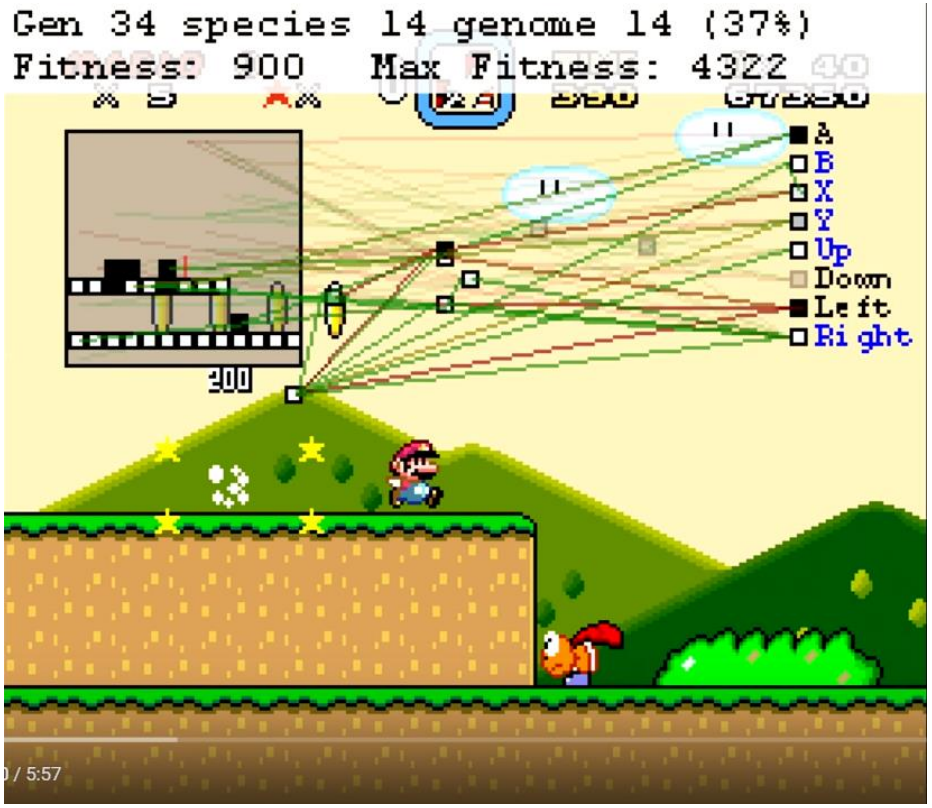
TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT …

- … uses evolutionary algorithms (e.g. Genetic algorithm) to train artificial neural networks
- … allows to optimize and complexify solutions simultaneously
- … minimizes the dimensionality of the weight space
- … focuses on evolving topologies along with weights rather than optimizing weights directly (e.g. Backpropagation)

Thus, NEAT is a **policy search method** which uses genetic algorithm where **each individual** in the population is a **candidate policy**

Example:
MarI/O – NEAT Implementation in „Super Mario World“



https://www.youtube.com/watch?v=qv6UVOQ0F44

NEAT Algorithm

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT(S, A, p, m_n, m_l, g, e)

```
1:  // S: set of all states, A: set of all actions, p: population size, m_n: node mutation rate
2:  // m_l: link mutation rate, g: number of generations, e: episodes per generation
3:
4:  P[] ← INIT-POPULATION(S, A, p)            // create new population P with random networks
5:  for i ← 1 to g do
6:      for j ← 1 to e do
7:          N, s, s' ← P[j % p], null, INIT-STATE(S)                    // select next network
8:          repeat
9:              Q[] ← EVAL-NET(N, s')              // evaluate selected network on current state
10:             a' ← argmax_i Q[i]                    // select action with highest activation
11:             s, a ← s', a'
12:             r, s' ← TAKE-ACTION(a')            // take action and transition to new state
13:             N.fitness ← N.fitness + r             // update total reward accrued by N
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1            // update total number of episodes for N
16:     P'[] ← new array of size p                  // new array will store next generation
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])          // make a new network based on fit parents in P
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j]) // add node to new network
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])          // add link to new network
21:     P[] ← P'[]
```

NEAT Algorithm – Parameters

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT$(S, A, p, m_n, m_l, g, e)$

```
1:
2:
3:
4:  P[] ← INIT-POPULATION(S, A, p)
5:  for i ← 1 to g do
6:      for j ← 1 to e do
7:          N, s, s' ← P[j % p], null, INIT-STATE(S)
8:          repeat
9:              Q[] ← EVAL-NET(N, s')
10:             a' ← argmax_i Q[i]
11:             s, a ← s', a'
12:             r, s' ← TAKE-ACTION(a')
13:             N.fitness ← N.fitness + r
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1
16:     P'[] ← new array of size p
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j])
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])
21:     P[] ← P'[]
```

S = Set of all state variables

A = Set of all action variables

p = Population size

m_n = Probability to mutate a node

m_l = Probability to mutate a link

g = Number of generation

e = Number of episodes per generation

NEAT Algorithm – Population initialization

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT$(S, A, p, m_n, m_l, g, e)$

```
1:
2:
3:
4:  P[] ← INIT-POPULATION(S, A, p)
5:  for i ← 1 to g do
6:      for j ← 1 to e do
7:          N, s, s' ← P[j % p], null, INIT-STATE(S)
8:          repeat
9:              Q[] ← EVAL-NET(N, s')
10:             a' ← argmax_i Q[i]
11:             s, a ← s', a'
12:             r, s' ← TAKE-ACTION(a')
13:             N.fitness ← N.fitness + r
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1
16:     P'[] ← new array of size p
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j])
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])
21:     P[] ← P'[]
```

- Initializing population by uniformly sampling *p* individuals
- Each individual $N \in P$
  - … is a neural network with …
    - … states as inputs
    - … actions as outputs
  - … **initially** has only **one link at all** between one state and one action

NEAT Algorithm – Generation routine

TECHNISCHE UNIVERSITÄT DARMSTADT

```
NEAT(S, A, p, m_n, m_l, g, e)
 1:
 2:
 3:
 4: P[] ← INIT-POPULATION(S, A, p)
 5: for i ← 1 to g do
 6:     for j ← 1 to e do
 7:         N, s, s' ← P[j % p], null, INIT-STATE(S)
 8:         repeat
 9:             Q[] ← EVAL-NET(N, s')
10:             a' ← argmax_i Q[i]
11:             s, a ← s', a'
12:             r, s' ← TAKE-ACTION(a')
13:             N.fitness ← N.fitness + r
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1
16:     P'[] ← new array of size p
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j])
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])
21:     P[] ← P'[]
```

1

2

For each generation there is …

1. … an **Evaluation Phase**

2. … a **Breed Phase**

NEAT Algorithm – Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT$(S, A, p, m_n, m_l, g, e)$

```
 1:
 2:
 3:
 4: P[] ← INIT-POPULATION(S, A, p)
 5: for i ← 1 to g do
 6:     for j ← 1 to e do
 7:         N, s, s' ← P[j % p], null, INIT-STATE(S)
 8:         repeat
 9:             Q[] ← EVAL-NET(N, s')
10:             a' ← argmax_i Q[i]
11:             s, a ← s', a'
12:             r, s' ← TAKE-ACTION(a')
13:             N.fitness ← N.fitness + r
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1
16:     P'[] ← new array of size p
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j])
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])
21:     P[] ← P'[]
```

- An individual $N \in P$ is selected for evaluation for an entire run
- The fitness of N is the accumulated sum of rewards

NEAT Algorithm – Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT$(S, A, p, m_n, m_l, g, e)$

```
1:
2:
3:
4:  P[] ← INIT-POPULATION(S, A, p)
5:  for i ← 1 to g do
6:      for j ← 1 to e do
7:          N, s, s' ← P[j % p], null, INIT-STATE(S)
8:          repeat
9:              Q[] ← EVAL-NET(N, s')
10:             a' ← argmax_i Q[i]
11:             s, a ← s', a'
12:             r, s' ← TAKE-ACTION(a')
13:             N.fitness ← N.fitness + r
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1
16:     P'[] ← new array of size p
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j])
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])
21:     P[] ← P'[]
```

- Each individual $N \in P$ will be evaluated for $\frac{e}{p}$ episodes (at most)
- In **deterministic domains**, we can use NEAT with $e = p$
- In **stochastic domains**, we need to evaluate each neural network many times
  - Thus, e may be much bigger than p

NEAT in stochastic domains

TECHNISCHE UNIVERSITÄT DARMSTADT

- In **on-line** learning, evaluating each individual equally for $\frac{e}{p}$ episodes is not optimal because it is purely **exploratory**
- Therefore, we need mechanisms to also allow **exploitation** of individiuals within a generation
- Again: In TD learning there exist action selectors which **balance exploration and exploitation**

➔ How can we utilize these selection methods?

NEAT Algorithm – Policy selection

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT$(S, A, p, m_n, m_l, g, e)$

```
1:
2:
3:
4:  P[] ← INIT-POPULATION(S, A, p)
5:  for i ← 1 to g do
6:      for j ← 1 to e do
7:          N, s, s' ← P[j % p], null, INIT-STATE(S)
8:          repeat
9:              Q[] ← EVAL-NET(N, s')
10:
11:             policySelector()
12:             r, s' ← TAKE-ACTION(a')
13:             N.fitness ← N.fitness + r
14:         until TERMINAL-STATE?(s)
15:         N.episodes ← N.episodes + 1
16:     P'[] ← new array of size p
17:     for j ← 1 to p do
18:         P'[j] ← BREED-NET(P[])
19:         with-probability m_n: ADD-NODE-MUTATION(P'[j])
20:         with-probability m_l: ADD-LINK-MUTATION(P'[j])
21:     P[] ← P'[]
```

- We can‘t use **TD action selectors** directly, because in NEAT there are **no value functions**
- But we can transform them into **policy selectors**

Policy selection methods

TECHNISCHE UNIVERSITÄT DARMSTADT

Here, **three** TD selection methods will be transformed

1. ϵ-Greedy
2. Softmax
3. Interval Estimation (Upper bound estimate)

In the following slides:

- $f(p)$ is the fitness of the policy p **averaged** over **all episodes** for which it has been **previously evaluated**

1. ϵ-Greedy Evolution

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithm 1 ϵ-GREEDY SELECTION(P, ϵ)

1: *// P: population, ϵ: NEAT's exploration rate*
2:
3: **with-prob**(ϵ) return RANDOM(P)
4: **else** return $\text{argmax}_{p \in P} f(p)$

Basic idea:

- With probability $1 - \epsilon$,
  select the best (fittest) policy of P
- Otherwise,
  a random policy is selected

2. Softmax Evolution

Algorithm 2 SOFTMAX SELECTION(P, τ)

```
1: // P: population, τ: softmax temperature
2:
3: if ∃ p ∈ P | e(p) = 0 then
4:     return p
5: else
6:     total ← Σ_{p∈P} e^{f(p)/τ}
7:     for all p ∈ P do
8:         with-prob(e^{f(p)/τ} / total) return p
9:         else total ← total − e^{f(p)/τ}
```

- Every policy $p \in P$ will be run atleast once
- Each policy has a chance to be chosen with a certain probability (higher fitness ➔ higher chance)
- The higher the temperature $\tau > 0$, the more equiprobable the policies are

3. Interval Estimation Evolution

Algorithm 3 INTERVAL ESTIMATION(P, α)

1: *// P: population, α: uncertainty in confidence interval*
2:
3: **if** $\exists\, p \in P \mid e(p) = 0$ **then**
4: return p
5: **else**
6: return $\text{argmax}_{p \in P}[f(p) + z(\frac{100-\alpha}{200})\frac{\sigma(p)}{\sqrt{e(p)}}]$

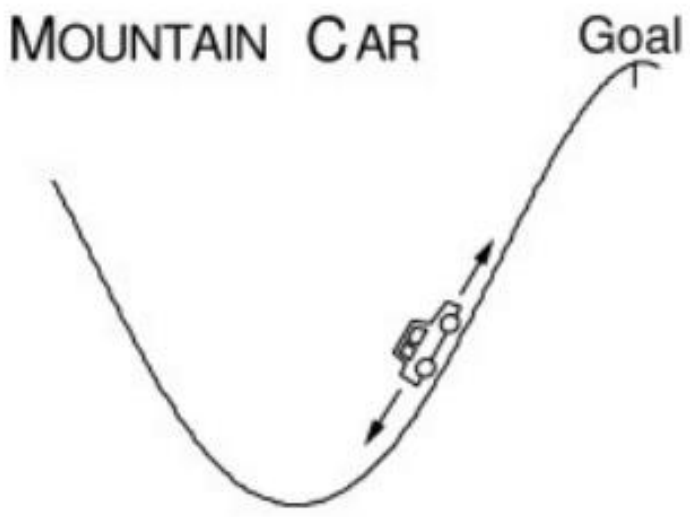
- As in Softmax,
  every policy $p \in P$ will be run atleast once
- $z(\cdot)$ is the
  standard normal distribution
- Estimation results into
  a $(100 - \alpha)\%$ confidence interval

For policy p

- $\sigma(p)$ is the standard deviation
  according to $f(p)$
- $e(p)$ is the number of episodes

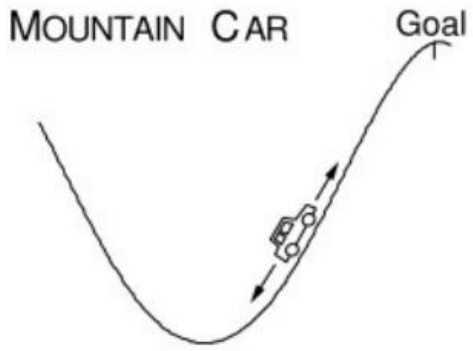
In a nutshell:
The policy with its highest upper bound will be selected

First experimental setup: Mountain Car



- The task for the agent is to drive a car **to the top** of a steep mountain
  - **Problem:** To overcome gravity, needs to get enough inertia to ascend to the goal
  - Thus, the agent needs to learn when to drive **fordwards** or **backwards**

First experimental setup: Mountain Car

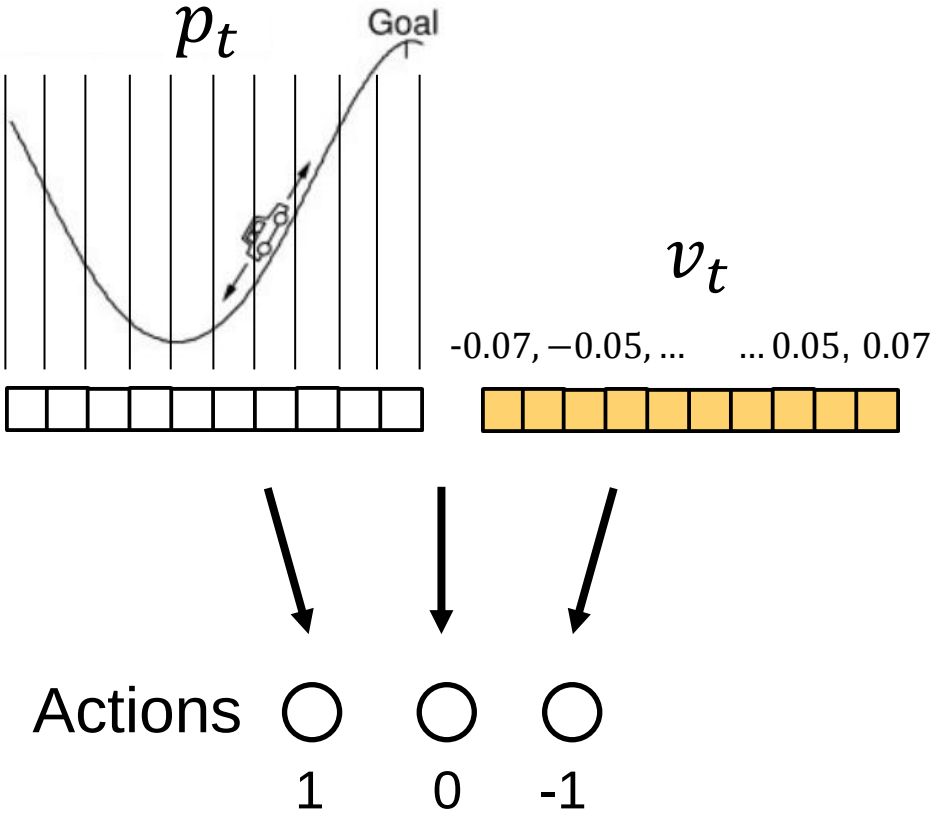


- Agent‘s state at timestep t:
  - Position p_t
  - Velocity v_t
- Reward for each timestep: -1
  - Until goal state is reached (➔ end of episode)
- Actions 1, 0 and -1 correspond to throttle settings

- Car movement control
  - $p_{t+1} = bound_p(p_t + v_{t+1})$
  - $v_{t+1} = bound_v(v_t + 0.001a_t - 0.0025\cos(3p_t))$
    - $bound_p(\cdot)$ enforces $-1.2 \leq p_{t+1} \leq 0.5$
    - $bound_v(\cdot)$ enforces $-0.07 \leq v_{t+1} \leq 0.07$

- In each episode, the **start position is chosen randomly**
- Episode terminates automatically after 2500 timesteps

First experimental setup: Mountain Car



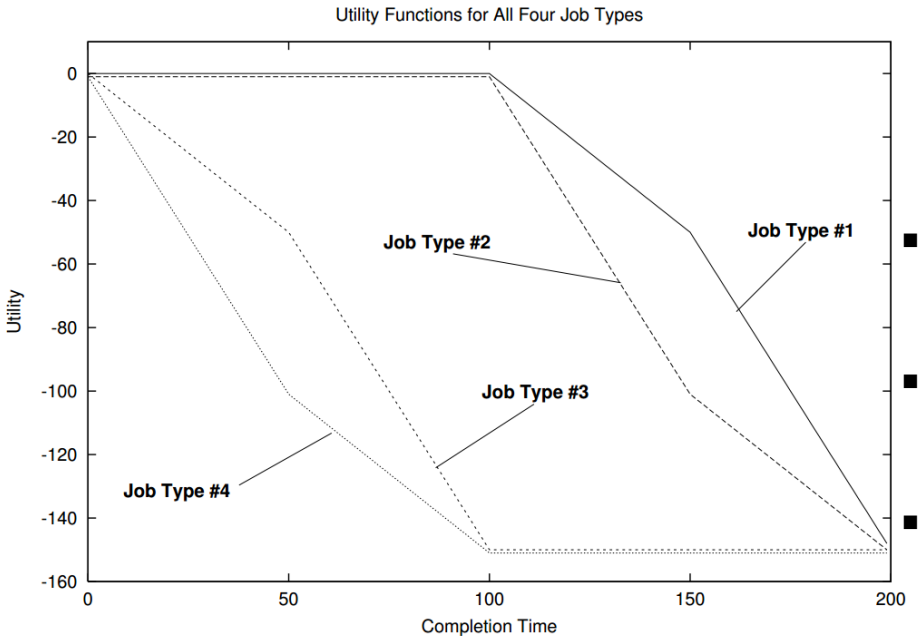
- To represent the current state for the neural network, **each state feature** is divided into **ten regions**
- Thus, **one input for each region**
  - Input is set to 1, if the agents current state is in that region
- Because each p_t and v_t fall into exact one region, only **two inputs are activated simultaniously**

Second experimental setup: Server Job Scheduling

TECHNISCHE UNIVERSITÄT DARMSTADT

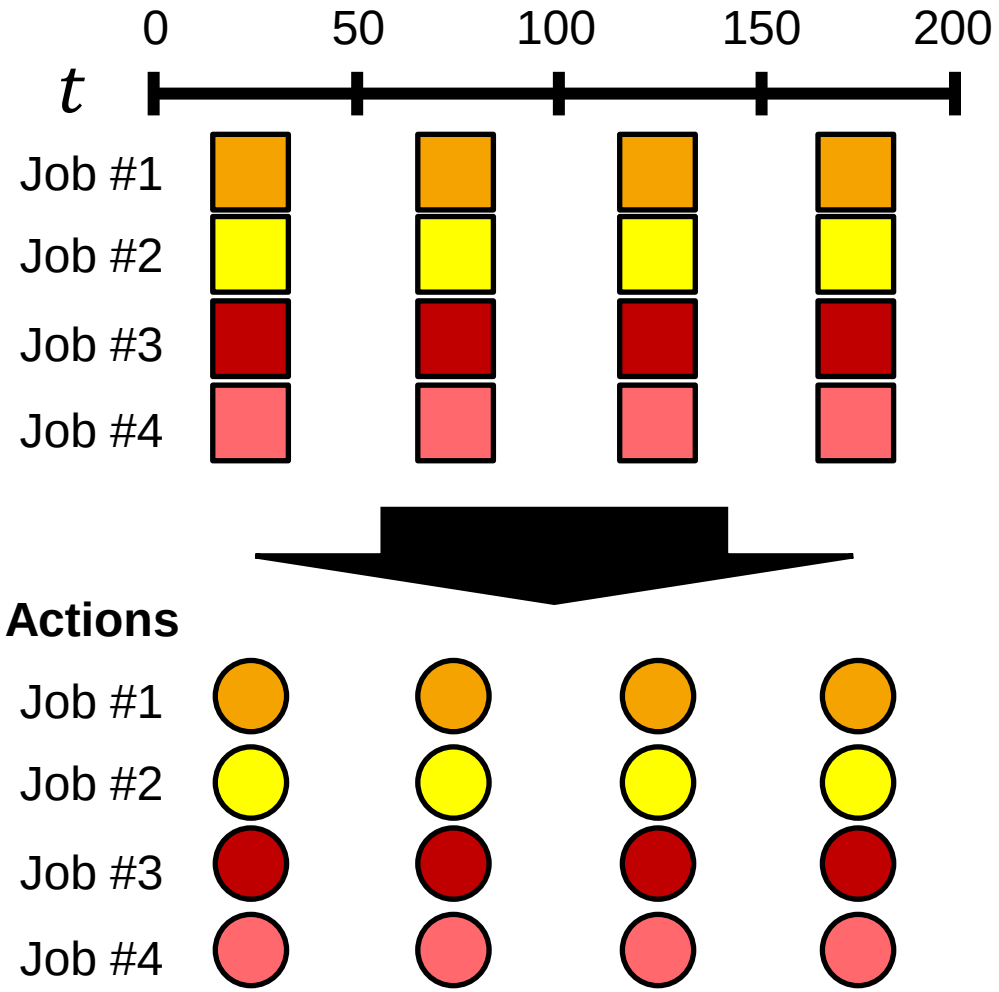
- In server job scheduling, we have with …
  - … a server (the role of the agent)
  - … a list of jobs (job queue)
- The task for the agent is to determine in what order to process the jobs in the queue
  - Thus, its goal is to **maximize the aggregate utility** of all jobs the agent processes ➔ Minimize waiting time for jobs
  - **Problems:**
    - Utility functions are usually non-linear
    - Multiple types of jobs
    - By time, random new jobs are added (with a random type)

Second experimental setup: Server Job Scheduling



- Four different job types are given
- Each job …
  - … can be completed in one timestep
  - … ages from timestep 0 to 200
  - ... has a utility function
- After job completion, an immediate reward is determined by its utility function
- #1 and #2 are less urgent than #3 and #4
- In each episode
  - 100 Jobs are preloaded
  - After each timestep t, a random job with random type is added
  - The episode ends after 200 timesteps

Second experimental setup: Server Job Scheduling



- For the neural Network, we again divide the range of job ages into four sections for each job type
  ➔ **16 state features**
- The Job selection is also divided into four sections for each job type
  ➔ **16 distinct actions**
- Therefore, a job is selected from one of 16 subsets

Results – Parameters

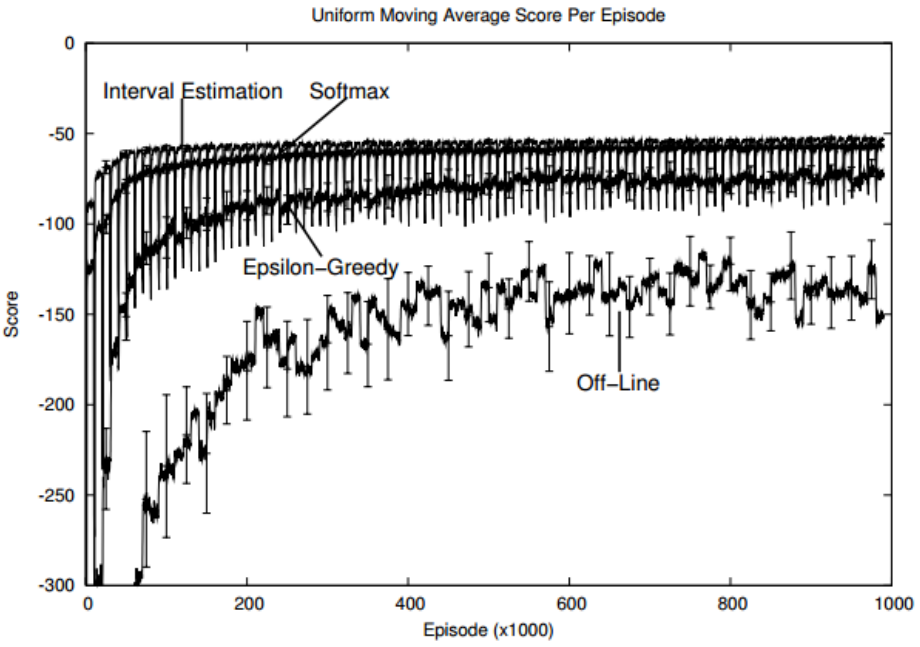
$p = 100$, size of Population

$e = 10000$ episodes per generation

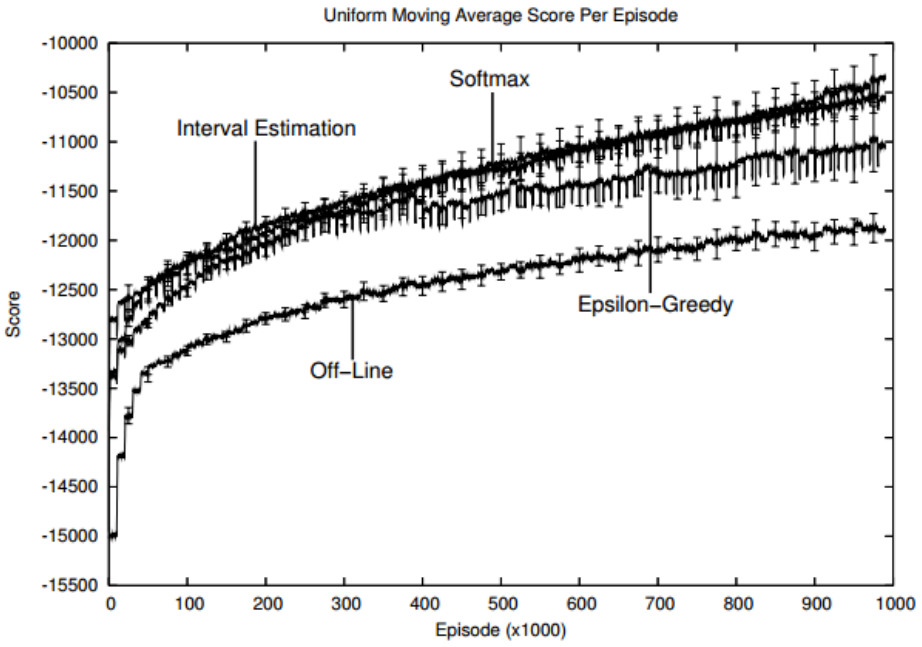
Four versions of NEAT were applied

- Original (**off-line**) NEAT as a baseline
  - Each policy in P was evaluated $\frac{e}{p} = 100$ times
- (**on-line**) NEAT with …
  - … ϵ-Greedy Evolution ($\epsilon = 0{,}25$)
  - … Softmax Evolution ($\tau = 50$ in Mountain Car, $\tau = 500$ in Server Job Scheduling)
  - … Interval Estimation Evolution ($\alpha = 20$, resulting in 80% confidence intervals)
- The parameters for the policy selection methods were found by informal experiments

Results – Uniform moving average score per episode



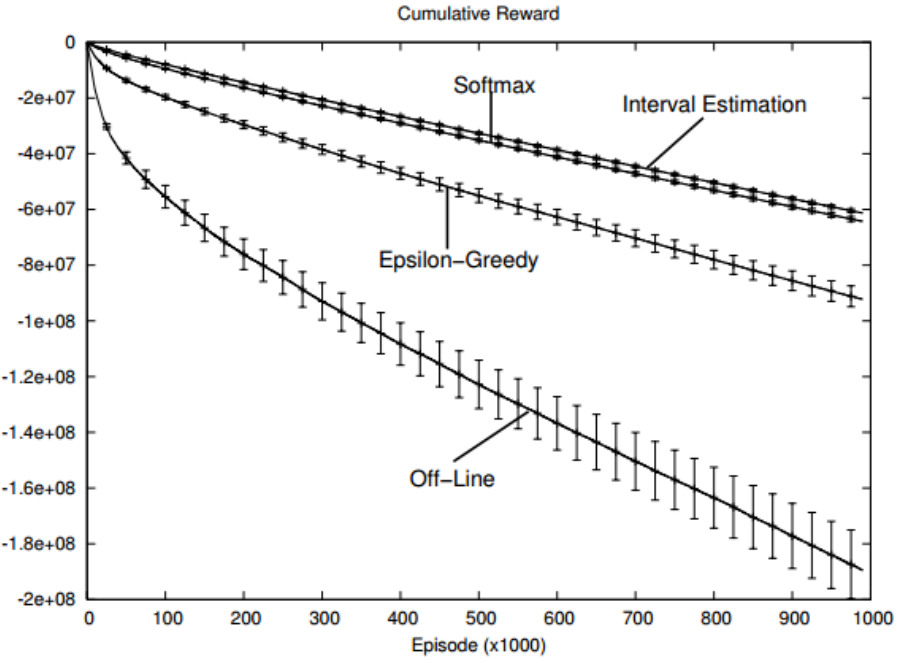
(a) Mountain Car



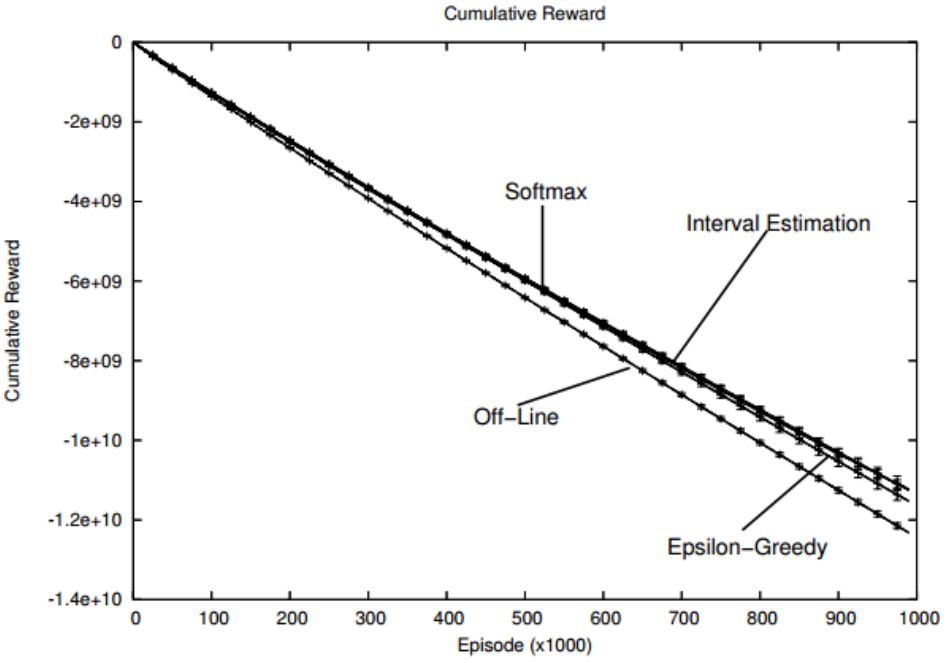
(b) Server Job Scheduling

- **Uniform moving average** (used in **on-line metrics**) over last 1000 episodes
- In both domains, rewards are negative
  ➔ Agents strive to get average reward as close to zero as possible
- Vertical bars on the graph indicate 95% confidence intervals

Results – Cumulative Reward



(a) Mountain Car



(b) Server Job Scheduling

- In Mountain Car domain, one can see that the on-line NEAT versions converge faster than off-line NEAT
- In Server Job Scheduling, Off-Line NEAT receives much less reward than On-Line NEAT by time

Results – Final remarks

TECHNISCHE UNIVERSITÄT DARMSTADT

- The experiments verify that selection mechanisms from TD methods can improve on-line performance
- **Softmax Evolution** and **Interval Estimation Evolution** also outperform ϵ-Greedy
  - But Softmax-parameter τ may require problem specific knowledge
    ➔ Harder to tune
- **Interval Estimation Evolution** may be the best choice
  - Easy to tune uncertainty parameter α

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

- TD **action selection methods** were transformed into **policy selection methods**
- These methods were then applied for policies in NEAT

  ➔ Expectation: These methods could also improve other policy search techniques (e.g. policy gradient methods (Sutton, McAllester et al. 1999))

- But there is also a limitation for **policy search techniques**
  - They do not exploit the specific structure of the reinforcement learning problem
  - **Follow up**

    ➔ **Evolutionary function approximation** (Whiteson 2010)

Evolutionary Function Approximation

TECHNISCHE UNIVERSITÄT DARMSTADT

NEAT+Q($S, A, c, p, m_n, m_l, g, e, \alpha, \gamma, \lambda, \varepsilon_{td}$)

```
1: // S: set of all states, A: set of all actions, c: output scale, p: population size
2: // m_n: node mutation rate, m_l: link mutation rate, g: number of generations
3: // e: number of episodes per generation, α: learning rate, γ: discount factor
4: // λ: eligibility decay rate, ε_td: exploration rate
5:
6: P[] ← INIT-POPULATION(S, A, p)          // create new population P with random networks
7: for i ← 1 to g do
8:    for j ← 1 to e do
9:       N, s, s' ← P[j % p], null, INIT-STATE(S)                    // select next network
10:      repeat
11:         Q[] ← c× EVAL-NET(N, s')          // compute value estimates for current state
12:
13:         with-prob(ε_td) a' ← RANDOM(A)                  // select random exploratory action
14:         else a' ← argmax_k Q[k]                                 // or select greedy action
15:         if s ≠ null then
16:            BACKPROP(N, s, a, (r + γmax_k Q[k])/c, α, γ, λ)                  // adjust weights
17:
18:         s, a ← s', a'
19:         r, s' ← TAKE-ACTION(a')                  // take action and transition to new state
20:         N.fitness ← N.fitness + r                  // update total reward accrued by N
21:      until TERMINAL-STATE?(s)
22:      N.episodes ← N.episodes + 1                  // update total number of episodes for N
23:   P'[] ← new array of size p                  // new array will store next generation
24:   for j ← 1 to p do
25:      P'[j] ← BREED-NET(P[])                  // make a new network based on fit parents in P
26:      with-probability m_n: ADD-NODE-MUTATION(P'[j]) // add node to new network
27:      with-probability m_l: ADD-LINK-MUTATION(P'[j])                  // add link to new network
28:   P[] ← P'[]
```

Basically NEAT, but with a reinterpretation of the output values

Thank you for your attention!