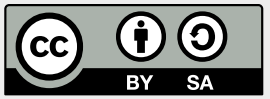
Speech Synthesis, Reinforcement Learning

Milan Straka

May 13, 2019

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics



Our goal is to model speech, using a auto-regressive model

$$p(\boldsymbol{x}) = \prod_t p(x_t | x_{t-1}, \ldots, x_1).$$

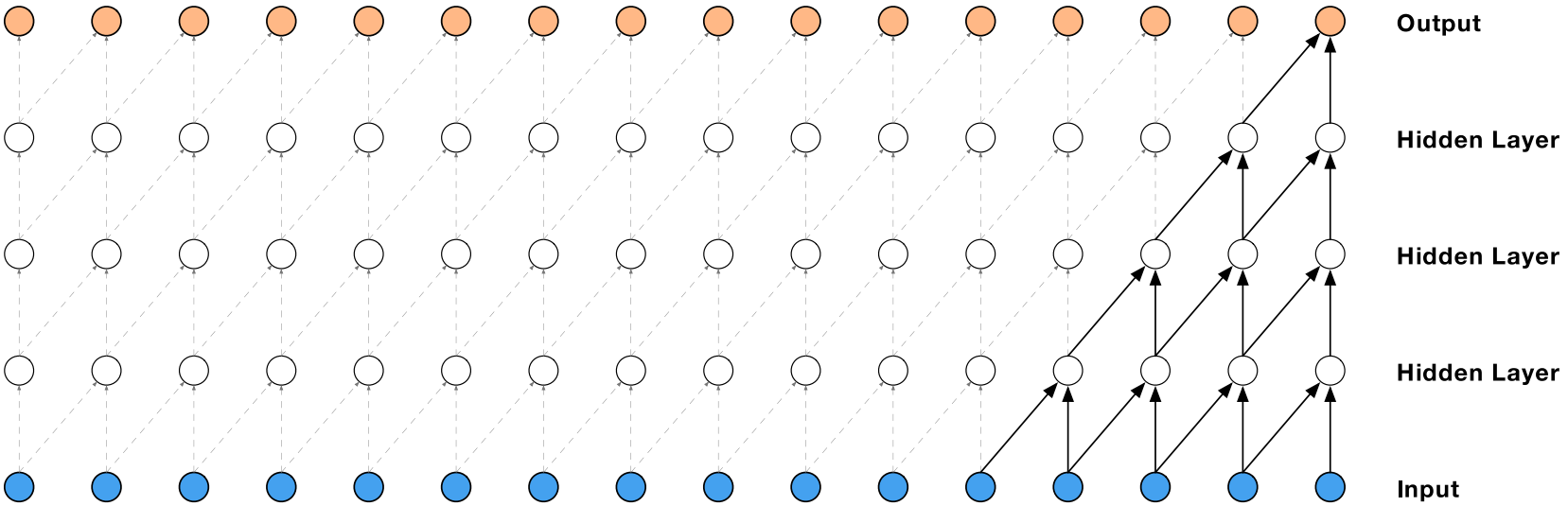


Figure 2: Visualization of a stack of causal convolutional layers.

Figure 2 of paper "WaveNet: A Generative Model for Raw Audio", https://arxiv.org/abs/1609.03499.

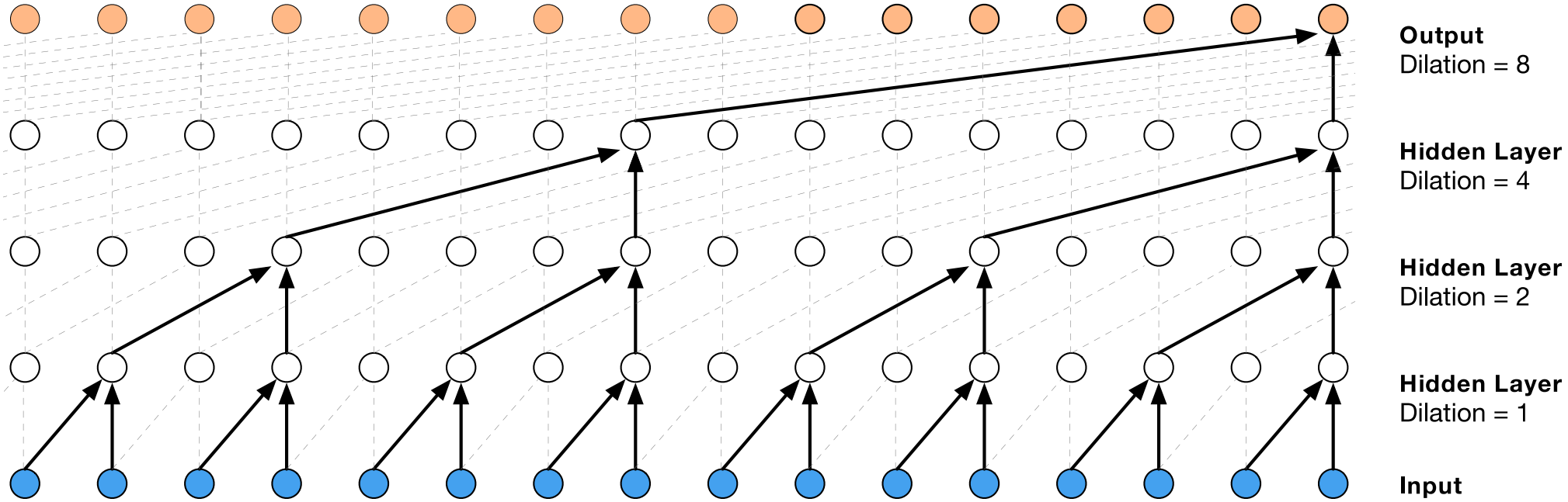


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

Figure 3 of paper "WaveNet: A Generative Model for Raw Audio", https://arxiv.org/abs/1609.03499.

Output Distribution

The raw audio is usually stored in 16-bit samples. However, classification into $65\,536$ classes would not be tractable, and instead WaveNet adopts μ-law transformation and quantize the samples into 256 values using

$$\operatorname{sign}(x)\frac{\ln(1+255|x|)}{\ln(1+255)}.$$

Gated Activation

To allow greater flexibility, the outputs of the dilated convolutions are passed through the gated activation units

$$\boldsymbol{z} = \tanh(\boldsymbol{W}_f * \boldsymbol{x}) \cdot \sigma(\boldsymbol{W}_g * \boldsymbol{x}).$$

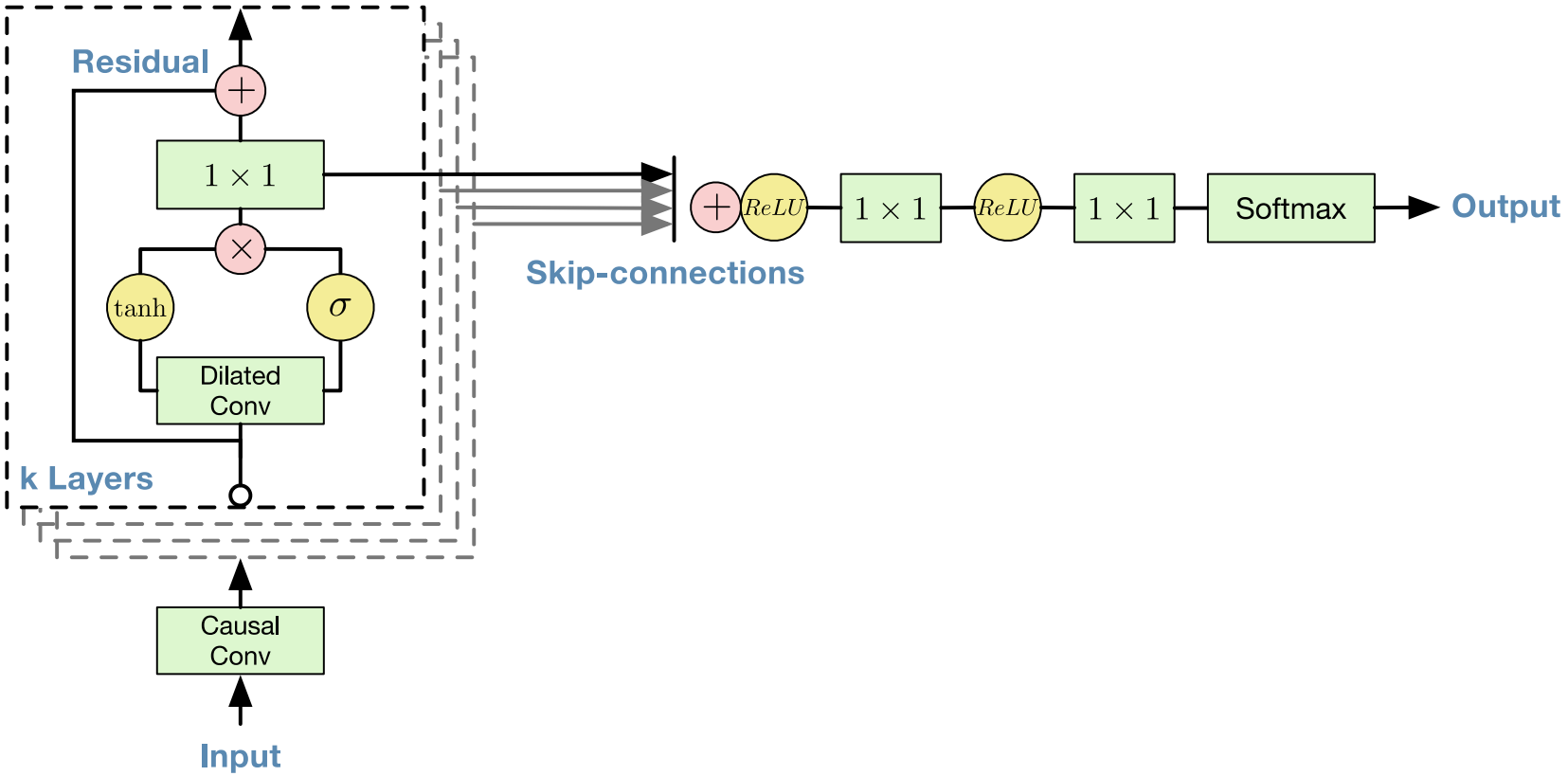


Figure 4: Overview of the residual block and the entire architecture.

Figure 4 of paper "WaveNet: A Generative Model for Raw Audio", https://arxiv.org/abs/1609.03499.

Global Conditioning

Global conditioning is performed by a single latent representation $\boldsymbol{h}$, changing the gated activation function to

$$\boldsymbol{z} = \tanh(\boldsymbol{W}_f * \boldsymbol{x} + \boldsymbol{V}_f \boldsymbol{h}) \cdot \sigma(\boldsymbol{W}_g * \boldsymbol{x} + \boldsymbol{V}_g \boldsymbol{h}).$$

Local Conditioning

For local conditioning, we are given a timeseries h_t, possibly with a lower sampling frequency. We first use transposed convolutions $\boldsymbol{y} = f(\boldsymbol{h})$ to match resolution and then compute analogously to global conditioning

$$\boldsymbol{z} = \tanh(\boldsymbol{W}_f * \boldsymbol{x} + \boldsymbol{V}_f * \boldsymbol{y}) \cdot \sigma(\boldsymbol{W}_g * \boldsymbol{x} + \boldsymbol{V}_g * \boldsymbol{y}).$$

WaveNet

The original paper did not mention hyperparameters, but later it was revealed that:

- 30 layers were used
  - grouped into 3 dilation stacks with 10 layers each
  - in a dilation stack, dilation rate increases by a factor of 2, starting with rate 1 and reaching maximum dilation of 512
- filter size of a dilated convolution is 3
- residual connection has dimension 512
- gating layer uses 256+256 hidden units
- the 1×1 output convolution produces 256 filters
- trained for $1\,000\,000$ steps using Adam with a fixed learning rate of 0.0002

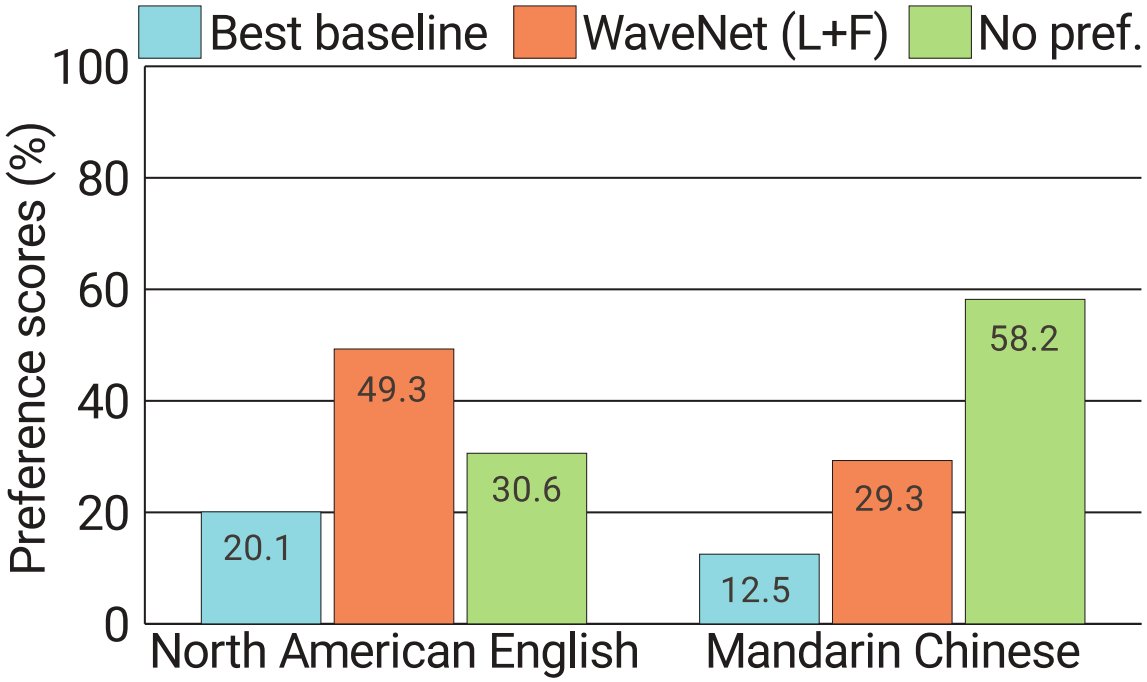


Figure 5: Subjective preference scores (%) of speech samples between (top) two baselines, (middle) two WaveNets, and (bottom) the best baseline and WaveNet. Note that LSTM and Concat correspond to LSTM-RNN-based statistical parametric and HMM-driven unit selection concatenative baseline synthesizers, and WaveNet (L) and WaveNet (L+F) correspond to the WaveNet conditioned on linguistic features only and that conditioned on both linguistic features and $\log F_0$ values.

Figure 5 of paper "WaveNet: A Generative Model for Raw Audio", https://arxiv.org/abs/1609.03499.

Parallel WaveNet

The output distribution was changed from 256 μ-law values to a Mixture of Logistic (suggested for another paper, but reused in other architectures since):

$$\nu \sim \sum_i \pi_i \operatorname{logistic}(\mu_i, s_i).$$

The logistic distribution is a distribution with a σ as cumulative density function (where the mean and steepness is parametrized by μ and s). Therefore, we can write

$$\nu \sim \sum_i \pi_i \left[\sigma((x + 0.5 - \mu_i)/s_i) - \sigma((x - 0.5 - \mu_i)/s_i)\right].$$

(where we replace -0.5 and 0.5 in the edge cases by $-\infty$ and ∞).

Parallel WaveNet

Auto-regressive (sequential) inference is extremely slow in WaveNet.

Instead, we use a following trick. We will model $p(x_t)$ as $p(x_t|\boldsymbol{z}_{\leq t})$ for a *random* $\boldsymbol{z}$ drawn from a logistic distribution. Then, we compute

$$x_t = z_t \cdot \boldsymbol{s}(\boldsymbol{z}_{<t}) + \boldsymbol{\mu}(\boldsymbol{z}_{<t}).$$

Usually, one iteration of the algorithm does not produce good enough results – 4 iterations were used by the authors. In further iterations,

$$x_t^i = x_t^{i-1} \cdot \boldsymbol{s}^i(\boldsymbol{x}_{<t}^{i-1}) + \boldsymbol{\mu}^i(\boldsymbol{x}_{<t}^{i-1}).$$

Parallel WaveNet

The network is trained using a *probability density distillation* using a teacher WaveNet, using KL-divergence as loss.

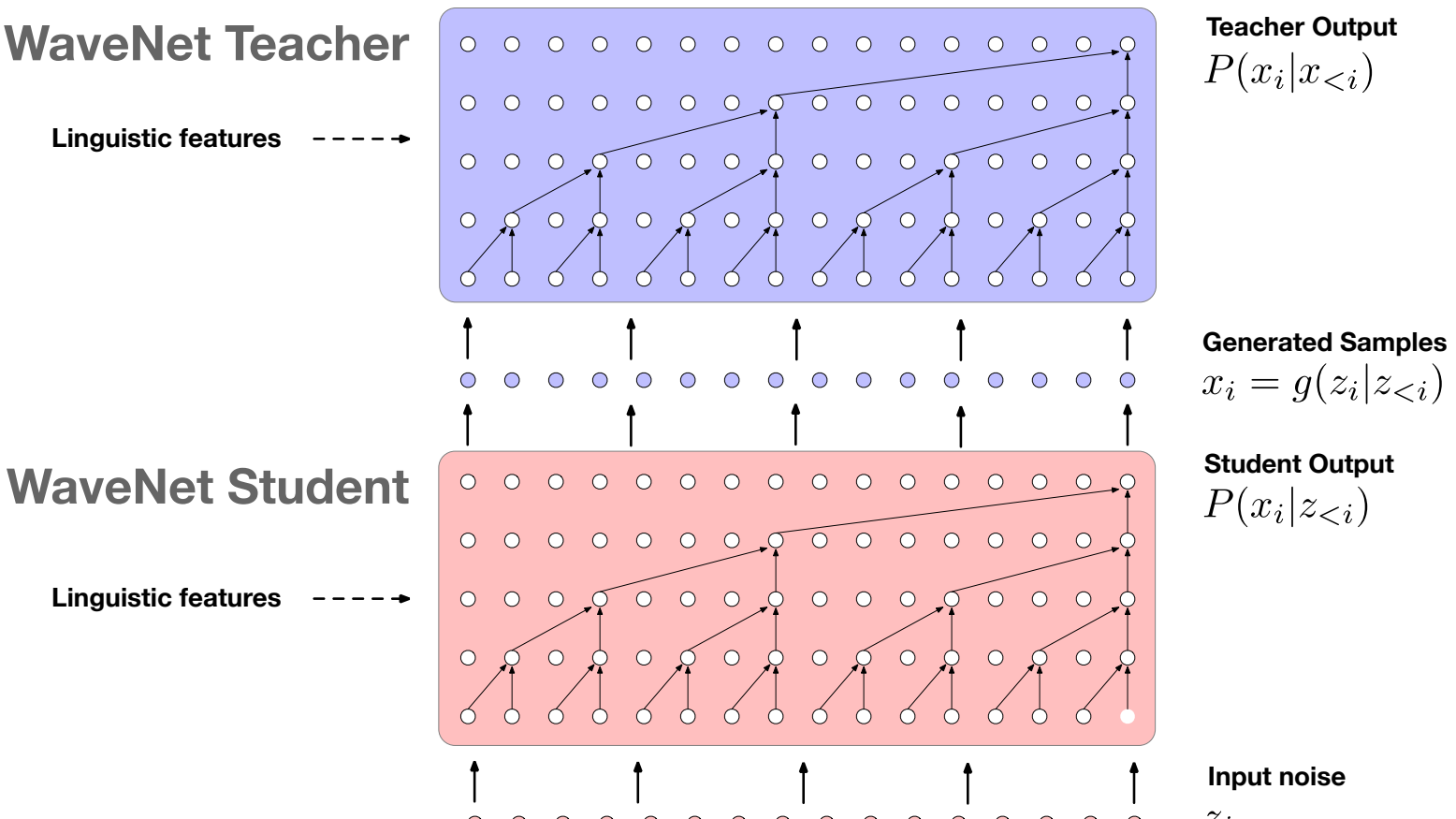


Figure 2 of paper "Parallel WaveNet: Fast High-Fidelity Speech Synthesis", https://arxiv.org/abs/1711.10433.

Parallel WaveNet

Method	Subjective 5-scale MOS
16kHz, 8-bit μ-law, 25h data:	
LSTM-RNN parametric [27]	3.67 ± 0.098
HMM-driven concatenative [27]	3.86 ± 0.137
WaveNet [27]	4.21 ± 0.081
24kHz, 16-bit linear PCM, 65h data:	
HMM-driven concatenative	4.19 ± 0.097
Autoregressive WaveNet	4.41 ± 0.069
Distilled WaveNet	4.41 ± 0.078

Table 1 of paper "Parallel WaveNet: Fast High-Fidelity Speech Synthesis", https://arxiv.org/abs/1711.10433.

Tacotron

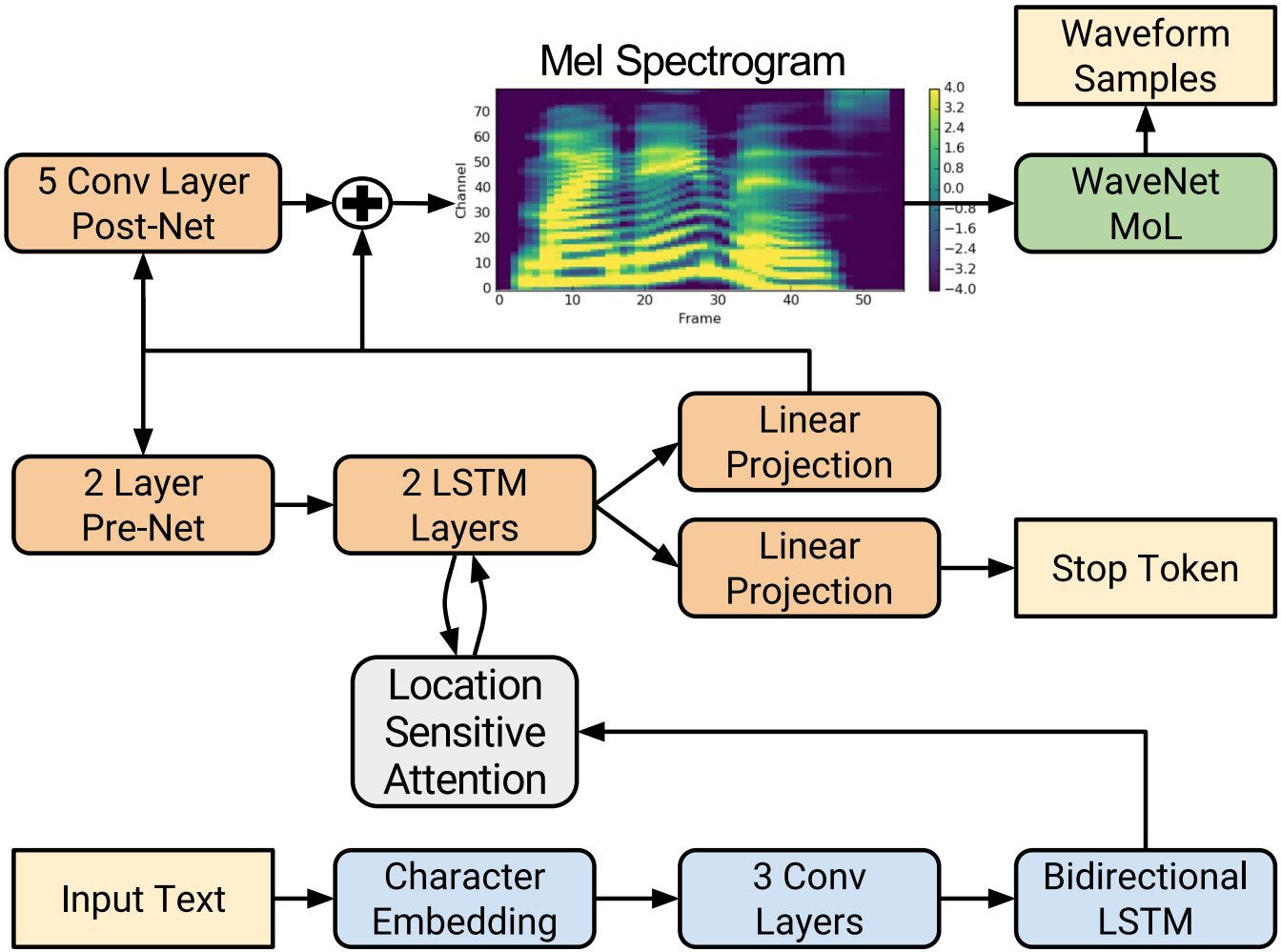


Figure 1 of paper "Natural TTS Synthesis by...", https://arxiv.org/abs/1712.05884.

System	MOS
Parametric	3.492 ± 0.096
Tacotron (Griffin-Lim)	4.001 ± 0.087
Concatenative	4.166 ± 0.091
WaveNet (Linguistic)	4.341 ± 0.051
Ground truth	4.582 ± 0.053
Tacotron 2 (this paper)	$\mathbf{4.526 \pm 0.066}$

Table 1 of paper "Natural TTS Synthesis by...", https://arxiv.org/abs/1712.05884.

Tacotron

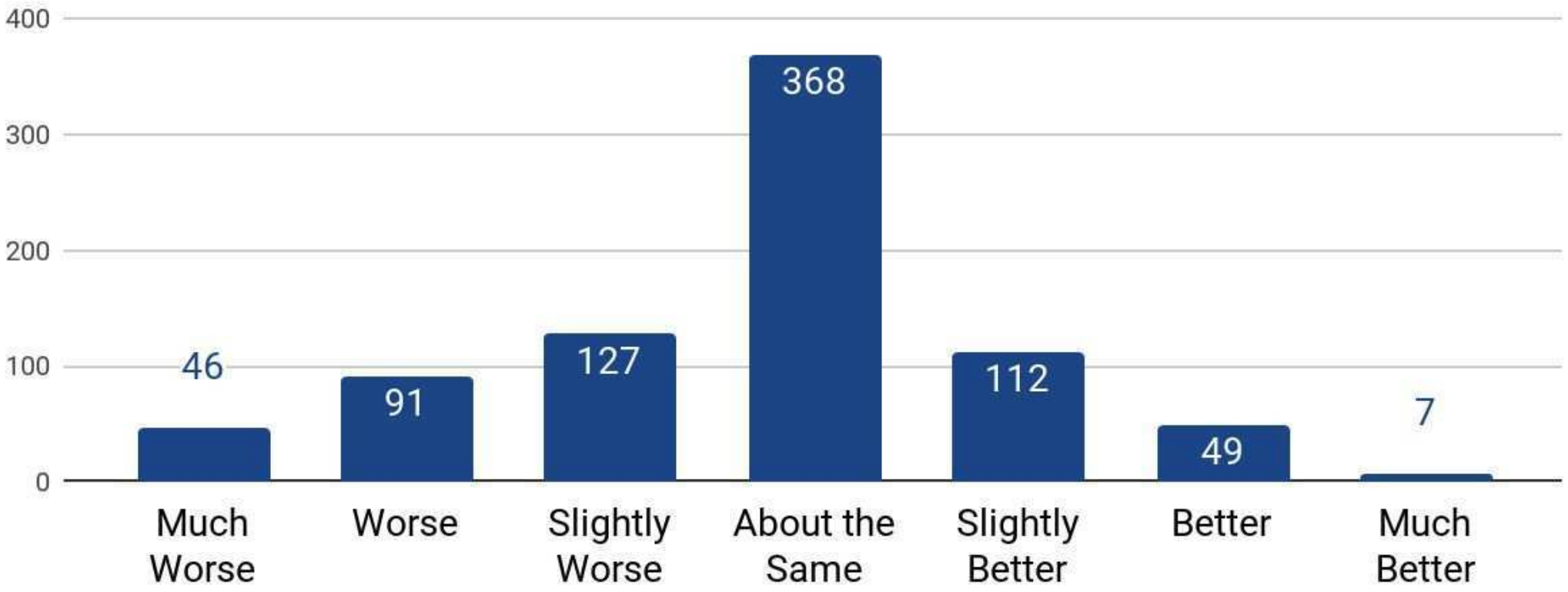


Figure 2 of paper "Natural TTS Synthesis by...", https://arxiv.org/abs/1712.05884.

Reinforcement Learning

History of Reinforcement Learning

Develop goal-seeking agent trained using reward signal.

- *Optimal control* in 1950s – Richard Bellman
- Trial and error learning – since 1850s
  - Law and effect – Edward Thorndike, 1911
  - Shannon, Minsky, Clark&Farley, ... – 1950s and 1960s
  - Tsetlin, Holland, Klopf – 1970s
  - Sutton, Barto – since 1980s
- Arthur Samuel – first implementation of temporal difference methods for playing checkers

Notable Successes of Reinforcement Learning

- IBM Watson in Jeopardy – 2011
- Human-level video game playing (DQN) – 2013 (2015 Nature), Mnih. et al, Deepmind
  - 29 games out of 49 comparable or better to professional game players
  - 8 days on GPU
  - human-normalized mean: 121.9%, median: 47.5% on 57 games
- A3C – 2016, Mnih. et al
  - 4 days on 16-threaded CPU
  - human-normalized mean: 623.0%, median: 112.6% on 57 games
- Rainbow – 2017
  - human-normalized median: 153%
- Impala – Feb 2018
  - one network and set of parameters to rule them all
  - human-normalized mean: 176.9%, median: 59.7% on 57 games

Notable Successes of Reinforcement Learning

- AlphaGo
  - Mar 2016 – beat 9-dan professional player Lee Sedol
- AlphaGo Master – Dec 2016
  - beat 60 professionals
  - beat Ke Jie in May 2017
- AlphaGo Zero – 2017
  - trained only using self-play
  - surpassed all previous version after 40 days of training
- AlphaZero – Dec 2017
  - self-play only
  - defeated AlphaGo Zero after 34 hours of training (21 million games)
  - impressive chess and shogi performance after 9h and 12h, respectively

Notable Successes of Reinforcement Learning

- Dota2 – Aug 2017
  - won 1v1 matches against a professional player
- MERLIN – Mar 2018
  - unsupervised representation of states using external memory
  - beat human in unknown maze navigation
- FTW – Jul 2018
  - beat professional players in two-player-team Capture the flag FPS
  - trained solely by self-play on 450k games
    - each 5 minutes, 4500 agent steps (15 per second)
- OpenAI Five – Aug 2018
  - won 5v5 best-of-three match against professional team
  - 256 GPUs, 128k CPUs
    - 180 years of experience per day
- AlphaStar – Jan 2019
  - played 11 games against StarCraft II professionals, reaching 10 wins and 1 loss

Notable Successes of Reinforcement Learning

- Neural Architecture Search – 2017
  - automatically designing CNN image recognition networks surpassing state-of-the-art performance
  - AutoML: automatically discovering
    - architectures (CNN, RNN, overall topology)
    - activation functions
    - optimizers
    - ...
- System for automatic control of data-center cooling – 2017

Multi-armed Bandits

http://www.infoslotmachine.com/img/one-armed-bandit.jpg

Multi-armed Bandits

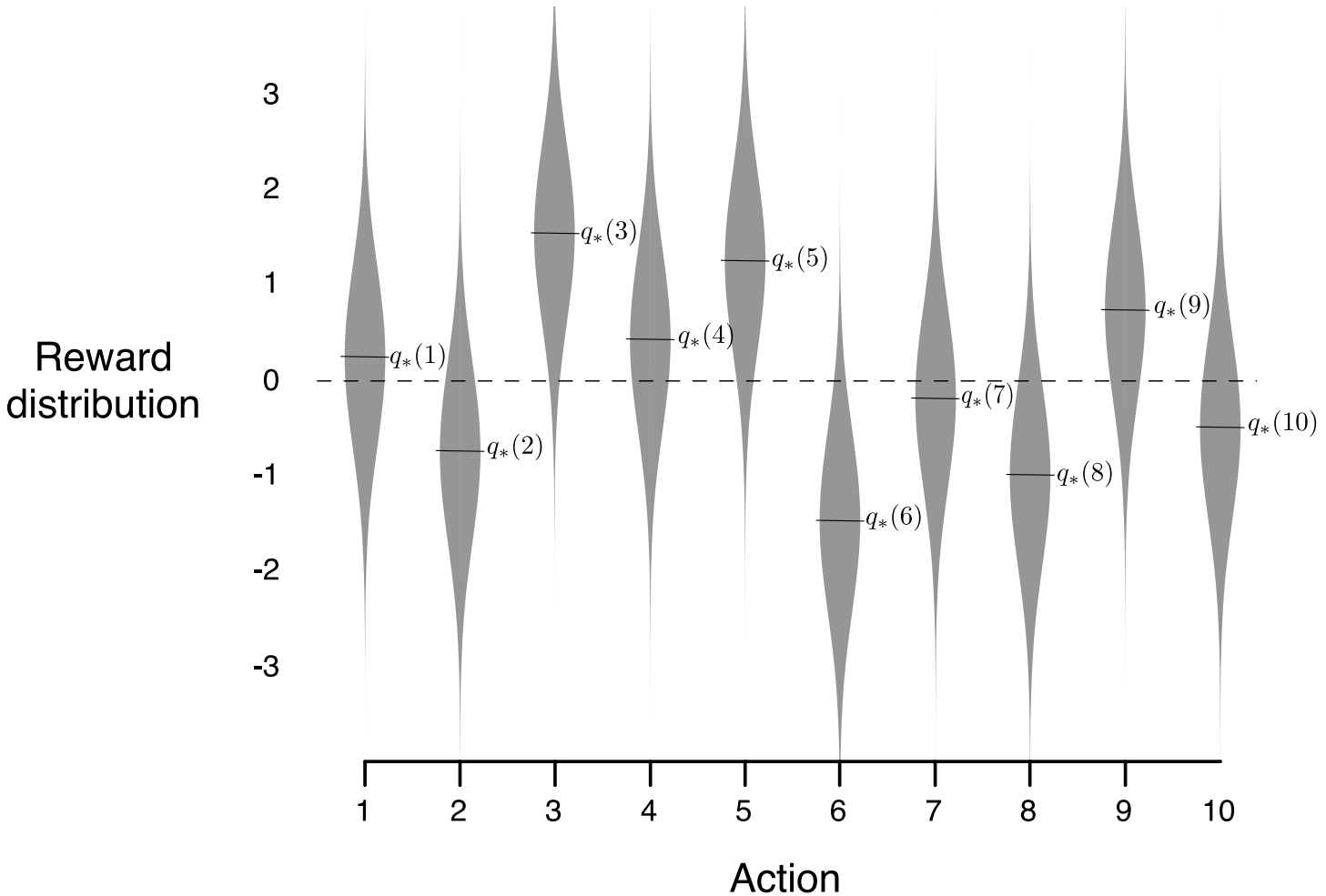


Figure 2.1 of "Reinforcement Learning: An Introduction, Second Edition".

We start by selecting action A_1, which is the index of the arm to use, and we get a reward of R_1. We then repeat the process by selecting actions A_2, A_3, ...

Let $q_*(a)$ be the real *value* of an action a:

$$q_*(a) = \mathbb{E}[R_t | A_t = a].$$

Denoting $Q_t(a)$ our estimated value of action a at time t (before taking trial t), we would like $Q_t(a)$ to converge to $q_*(a)$. A natural way to estimate $Q_t(a)$ is

$$Q_t(a) \stackrel{\text{def}}{=} \frac{\text{sum of rewards when action } a \text{ is taken}}{\text{number of times action } a \text{ was taken}}.$$

Following the definition of $Q_t(a)$, we could choose a *greedy action* A_t as

$$A_t(a) \stackrel{\text{def}}{=} \arg\max_a Q_t(a).$$

Exploitation versus Exploration

Choosing a greedy action is *exploitation* of current estimates. We however also need to *explore* the space of actions to improve our estimates.

An *ε-greedy* method follows the greedy action with probability $1 - \varepsilon$, and chooses a uniformly random action with probability ε.

ε-greedy Method

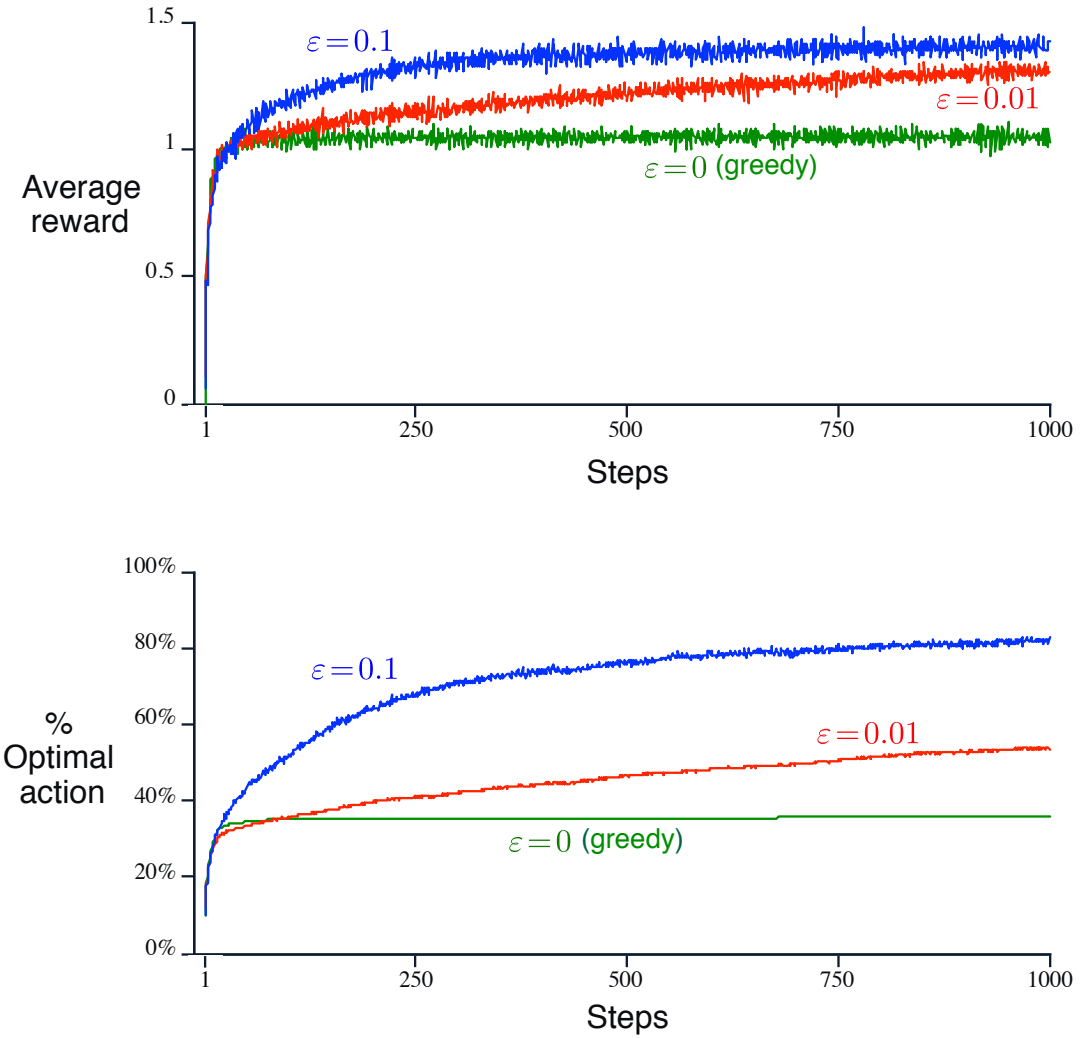


Figure 2.2 of "Reinforcement Learning: An Introduction, Second Edition".

Incremental Implementation

Let Q_{n+1} be an estimate using n rewards $R_1, \ldots, R_n$.

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n} \sum_{i=1}^{n} R_i \\
&= \frac{1}{n} (R_n + \frac{n-1}{n-1} \sum_{i=1}^{n-1} R_i) \\
&= \frac{1}{n} (R_n + (n-1) Q_n) \\
&= \frac{1}{n} (R_n + n Q_n - Q_n) \\
&= Q_n + \frac{1}{n} \Big(R_n - Q_n \Big)
\end{aligned}
$$

ε-greedy Method Algorithm

A simple bandit algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Algorithm 2.4 of "Reinforcement Learning: An Introduction, Second Edition".

Markov Decision Process

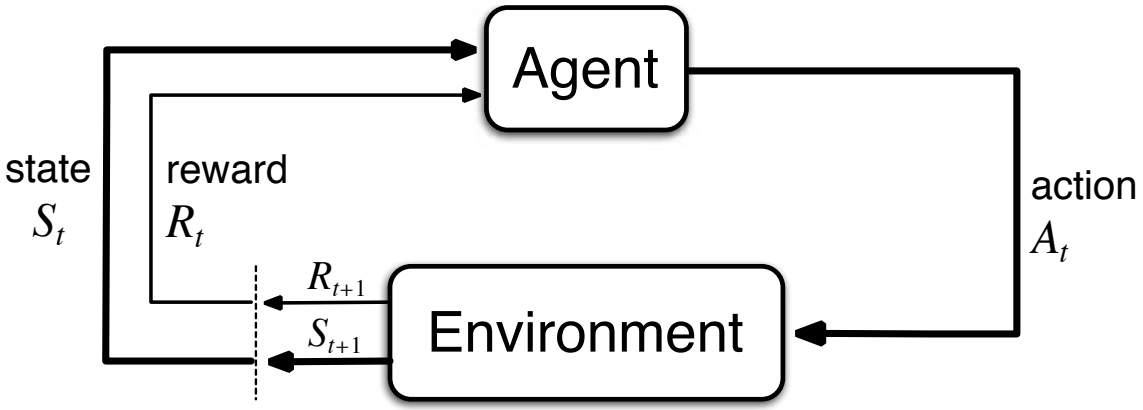


Figure 3.1 of "Reinforcement Learning: An Introduction, Second Edition".

A *Markov decision process* (MDP) is a quadruple $(\mathcal{S}, \mathcal{A}, p, \gamma)$, where:

- $\mathcal{S}$ is a set of states,
- $\mathcal{A}$ is a set of actions,
- $p(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$ is a probability that action $a \in \mathcal{A}$ will lead from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$, producing a *reward* $r \in \mathbb{R}$,
- $\gamma \in [0, 1]$ is a *discount factor* (we will always use $\gamma = 1$).

Let a *return* G_t be $G_t \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$. The goal is to optimize $\mathbb{E}[G_0]$.

Multi-armed Bandits as MDP

To formulate n-armed bandits problem as MDP, we do not need states. Therefore, we could formulate it as:

- one-element set of states, $\mathcal{S} = \{S\}$;
- an action for every arm, $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$;
- assuming every arm produces rewards with a distribution of $\mathcal{N}(\mu_i, \sigma_i^2)$, the MDP dynamics function p is defined as

$$p(S, r|S, a_i) = \mathcal{N}(r|\mu_i, \sigma_i^2).$$

One possibility to introduce states in multi-armed bandits problem is to have separate reward distribution for every state. Such generalization is usually called *Contextualized Bandits* problem. Assuming that state transitions are independent on rewards and given by a distribution $next(s)$, the MDP dynamics function for contextualized bandits problem is given by

$$p(s', r|s, a_i) = \mathcal{N}(r|\mu_{i,s}, \sigma_{i,s}^2) \cdot next(s'|s).$$

(State-)Value and Action-Value Functions

A *policy* π computes a distribution of actions in a given state, i.e., $\pi(a|s)$ corresponds to a probability of performing an action a in state s.

To evaluate a quality of a policy, we define *value function* $v_\pi(s)$, or *state-value function*, as

$$v_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}_\pi\left[G_t | S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right].$$

An *action-value function* for a policy π is defined analogously as

$$q_\pi(s,a) \stackrel{\text{def}}{=} \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right].$$

Evidently,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[q_\pi(s,a)], \\ q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]. \end{aligned}$$

Optimal Value Functions

Optimal state-value function is defined as

$$v_*(s) \stackrel{\text{def}}{=} \max_\pi v_\pi(s),$$

analogously

$$q_*(s,a) \stackrel{\text{def}}{=} \max_\pi q_\pi(s,a).$$

Any policy π_* with $v_{\pi_*} = v_*$ is called an *optimal policy*. Such policy can be defined as
$\pi_*(s) \stackrel{\text{def}}{=} \arg\max_a q_*(s,a) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$.

Existence

Under some mild assumptions, there always exists a unique optimal state-value function, unique optimal action-value function, and (not necessarily unique) optimal policy. The mild assumptions are that either termination is guaranteed from all reachable states, or $\gamma < 1$.

Monte Carlo Methods

We now present the first algorithm for computing optimal policies without assuming a knowledge of the environment dynamics.

However, we still assume there are finitely many states $\mathcal{S}$ and we will store estimates for each of them.

Monte Carlo methods are based on estimating returns from complete episodes. Furthermore, if the model (of the environment) is not known, we need to estimate returns for the action-value function q instead of v.

To guarantee convergence, we need to visit each state infinitely many times. One of the simplest way to achieve that is to assume *exploring starts*, where we randomly select the first state and first action, each pair with nonzero probability.

Furthermore, if a state-action pair appears multiple times in one episode, the sampled returns are not independent. The literature distinguishes two cases:

- *first visit*: only the first occurence of a state-action pair in an episode is considered
- *every visit*: all occurences of a state-action pair are considered.

Even though first-visit is easier to analyze, it can be proven that for both approaches, policy evaluation converges. Contrary to the Reinforcement Learning: An Introduction book, which presents first-visit algorithms, we use every-visit.

Monte Carlo with Exploring Starts

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Modification (no first-visit) of algorithm 5.3 of "Reinforcement Learning: An Introduction, Second Edition".

Monte Carlo and ε-soft Policies

A policy is called ε-soft, if

$$\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}.$$

For ε-soft policy, Monte Carlo policy evaluation also converges, without the need of exploring starts.

We call a policy ε-greedy, if one action has maximum probability of $1 - \varepsilon + \frac{\varepsilon}{|A(s)|}$.

The policy improvement theorem can be proved also for the class of ε-soft policies, and using ε-greedy policy in policy improvement step, policy iteration has the same convergence properties. (We can embed the ε-soft behaviour "inside" the environment and prove equivalence.)

On-policy every-visit Monte Carlo for ε-soft Policies

Algorithm parameter: small $\varepsilon > 0$

Initialize $Q(s, a) \in \mathbb{R}$ arbitrarily (usually to 0), for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $C(s, a) \in \mathbb{Z}$ to 0, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Repeat forever (for each episode):

- Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, by generating actions as follows:
  - With probability ε, generate a random uniform action
  - Otherwise, set $A_t \overset{\text{def}}{=} \arg\max_a Q(S_t, a)$
- $G \leftarrow 0$
- For each $t = T-1, T-2, \ldots, 0$:
  - $G \leftarrow \gamma G + R_{T+1}$
  - $C(S_t, A_t) \leftarrow C(S_t, A_t) + 1$
  - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{C(S_t, A_t)}(G - Q(S_t, A_t))$

Policy Gradient Methods

Instead of predicting expected returns, we could train the method to directly predict the policy

$$\pi(a|s;\boldsymbol{\theta}).$$

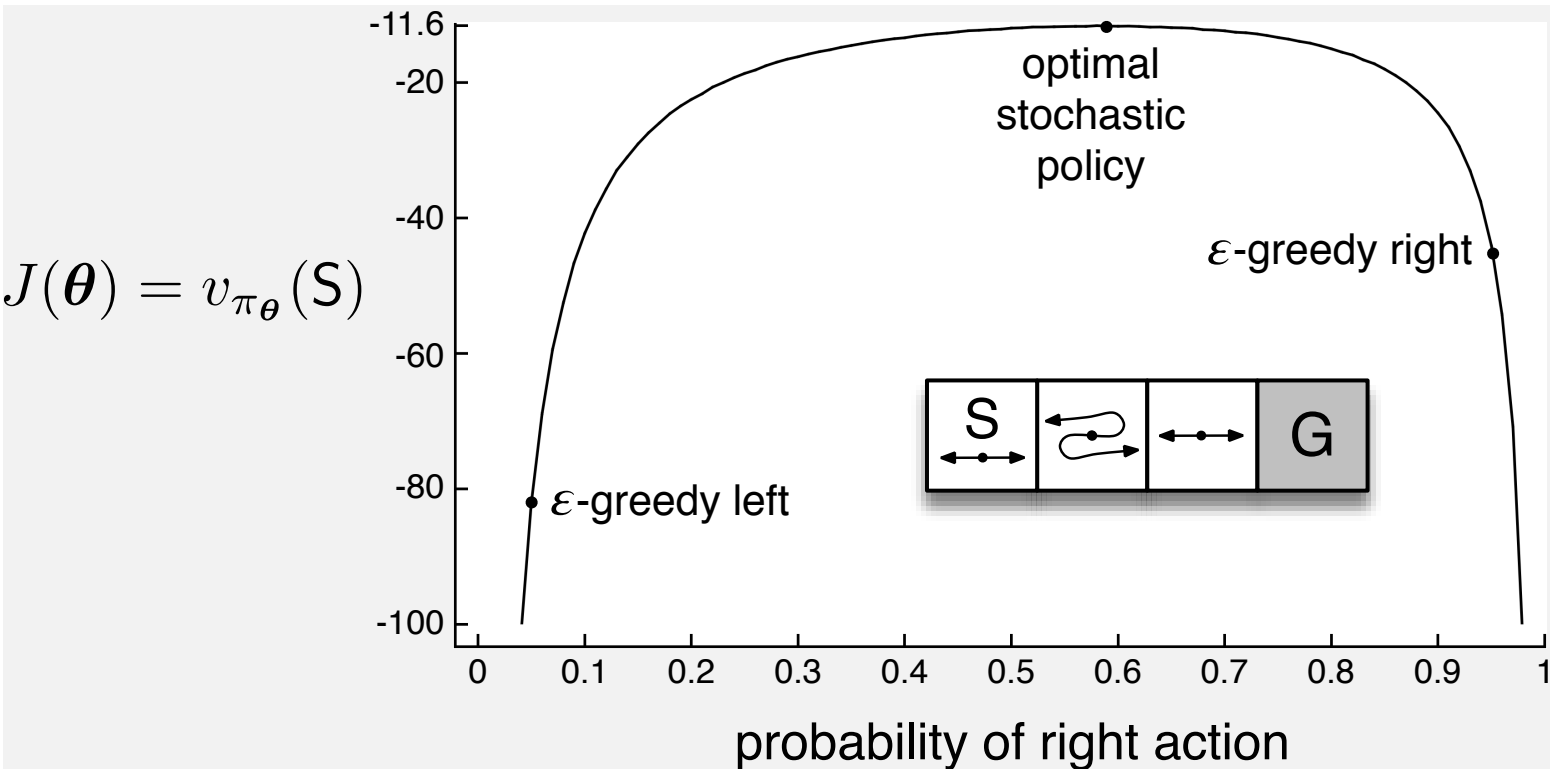
Obtaining the full distribution over all actions would also allow us to sample the actions according to the distribution π instead of just ε-greedy sampling.

However, to train the network, we maximize the expected return $v_\pi(s)$ and to that account we need to compute its *gradient* $\nabla_{\boldsymbol{\theta}} v_\pi(s)$.

Policy Gradient Methods

In addition to discarding ε-greedy action selection, policy gradient methods allow producing policies which are by nature stochastic, as in card games with imperfect information, while the action-value methods have no natural way of finding stochastic policies (distributional RL might be of some use though).

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\mathrm{S})$$



Example 13.1 of "Reinforcement Learning: An Introduction, Second Edition".

Policy Gradient Theorem

Let $\pi(a|s;\boldsymbol{\theta})$ be a parametrized policy. We denote the initial state distribution as $h(s)$ and the on-policy distribution under π as $\mu(s)$. Let also $J(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \mathbb{E}_{h,\pi} v_\pi(s)$.

Then

$$\nabla_{\boldsymbol{\theta}} v_\pi(s) \propto \sum_{s' \in \mathcal{S}} P(s \to \ldots \to s'|\pi) \sum_{a \in \mathcal{A}} q_\pi(s', a) \nabla_{\boldsymbol{\theta}} \pi(a|s';\boldsymbol{\theta})$$

and

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}),$$

where $P(s \to \ldots \to s'|\pi)$ is probability of transitioning from state s to s' using 0, 1, ... steps.

Proof of Policy Gradient Theorem

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla\Big[\sum_a \pi(a|s;\boldsymbol{\theta}) q_\pi(s,a)\Big] \\
&= \sum_a \Big[\nabla\pi(a|s;\boldsymbol{\theta}) q_\pi(s,a) + \pi(a|s;\boldsymbol{\theta})\nabla q_\pi(s,a)\Big] \\
&= \sum_a \Big[\nabla\pi(a|s;\boldsymbol{\theta}) q_\pi(s,a) + \pi(a|s;\boldsymbol{\theta})\nabla\big(\sum_{s'} p(s'|s,a)(r + v_\pi(s'))\big)\Big] \\
&= \sum_a \Big[\nabla\pi(a|s;\boldsymbol{\theta}) q_\pi(s,a) + \pi(a|s;\boldsymbol{\theta})\big(\sum_{s'} p(s'|s,a)\nabla v_\pi(s')\big)\Big]
\end{aligned}$$

We now expand $v_\pi(s')$.

$$\begin{aligned}
&= \sum_a \Big[\nabla\pi(a|s;\boldsymbol{\theta}) q_\pi(s,a) + \pi(a|s;\boldsymbol{\theta})\Big(\sum_{s'} p(s'|s,a)\Big(\\
&\qquad \sum_{a'} \Big[\nabla\pi(a'|s';\boldsymbol{\theta}) q_\pi(s',a') + \pi(a'|s';\boldsymbol{\theta})\Big(\sum_{s''} p(s''|s',a')\nabla v_\pi(s'')\Big)\Big)\Big]
\end{aligned}$$

Continuing to expand all $v_\pi(s'')$, *we obtain the following:*

$$\nabla v_\pi(s) = \sum_{s'\in\mathcal{S}} P(s \to \ldots \to s'|\pi) \sum_{a\in\mathcal{A}} q_\pi(s',a)\nabla_{\boldsymbol{\theta}}\pi(a|s';\boldsymbol{\theta}).$$

Proof of Policy Gradient Theorem

Recall that the initial state distribution is $h(s)$ and the on-policy distribution under π is $\mu(s)$. If we let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode, we have

$$\eta(s) = h(s) + \sum_{s'} \eta(s') \sum_{a} \pi(a|s') p(s|s', a).$$

The on-policy distribution is then the normalization of $\eta(s)$:

$$\mu(s) \stackrel{\text{def}}{=} \frac{\eta(s)}{\sum_{s'} \eta(s')}.$$

The last part of the policy gradient theorem follows from the fact that $\mu(s)$ is

$$\mu(s) = \mathbb{E}_{s_0 \sim h(s)} P(s_0 \to \ldots \to s|\pi).$$

REINFORCE Algorithm

The REINFORCE algorithm (Williams, 1992) uses directly the policy gradient theorem, maximizing $J(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \mathbb{E}_{h,\pi} v_\pi(s)$. The loss is defined as

$$\begin{aligned} -\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}) \\ &= \mathbb{E}_{s \sim \mu} \sum_{a \in \mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}). \end{aligned}$$

However, the sum over all actions is problematic. Instead, we rewrite it to an expectation which we can estimate by sampling:

$$-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \mathbb{E}_{s \sim \mu} \mathbb{E}_{a \sim \pi} q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \ln \pi(a|s;\boldsymbol{\theta}),$$

where we used the fact that

$$\nabla_{\boldsymbol{\theta}} \ln \pi(a|s;\boldsymbol{\theta}) = \frac{1}{\pi(a|s;\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}).$$

REINFORCE Algorithm

REINFORCE therefore minimizes the loss

$$-\mathbb{E}_{s\sim\mu}\mathbb{E}_{a\sim\pi} q_\pi(s,a)\nabla_{\boldsymbol{\theta}} \ln \pi(a|s;\boldsymbol{\theta}),$$

estimating the $q_\pi(s,a)$ by a single sample.

Note that the loss is just a weighted variant of negative log likelihood (NLL), where the sampled actions play a role of gold labels and are weighted according to their return.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \quad \alpha\, G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Modification of Algorithm 13.3 of "Reinforcement Learning: An Introduction, Second Edition".

REINFORCE with Baseline

The returns can be arbitrary – better-than-average and worse-than-average returns cannot be recognized from the absolute value of the return.

Hopefully, we can generalize the policy gradient theorem using a baseline $b(s)$ to

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} \big(q_\pi(s,a) - b(s)\big) \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}).$$

The baseline $b(s)$ can be a function or even a random variable, as long as it does not depend on a, because

$$\sum_a b(s) \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}) = b(s) \sum_a \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}) = b(s) \nabla 1 = 0.$$

REINFORCE with Baseline

A good choice for $b(s)$ is $v_\pi(s)$, which can be shown to minimize variance of the estimator. Such baseline reminds centering of returns, given that

$$v_\pi(s) = \mathbb{E}_{a\sim\pi} q_\pi(s,a).$$

Then, better-than-average returns are positive and worse-than-average returns are negative. The resulting $q_\pi(s,a) - v_\pi(s)$ function is also called an *advantage function*

$$a_\pi(s,a) \stackrel{\text{def}}{=} q_\pi(s,a) - v_\pi(s).$$

Of course, the $v_\pi(s)$ baseline can be only approximated. If neural networks are used to estimate $\pi(a|s;\boldsymbol{\theta})$, then some part of the network is usually shared between the policy and value function estimation, which is trained using mean square error of the predicted and observed return.

REINFORCE with Baseline

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \qquad (G_t)$$

$$\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \quad \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$$

Modification of Algorithm 13.4 of "Reinforcement Learning: An Introduction, Second Edition".

REINFORCE with Baseline

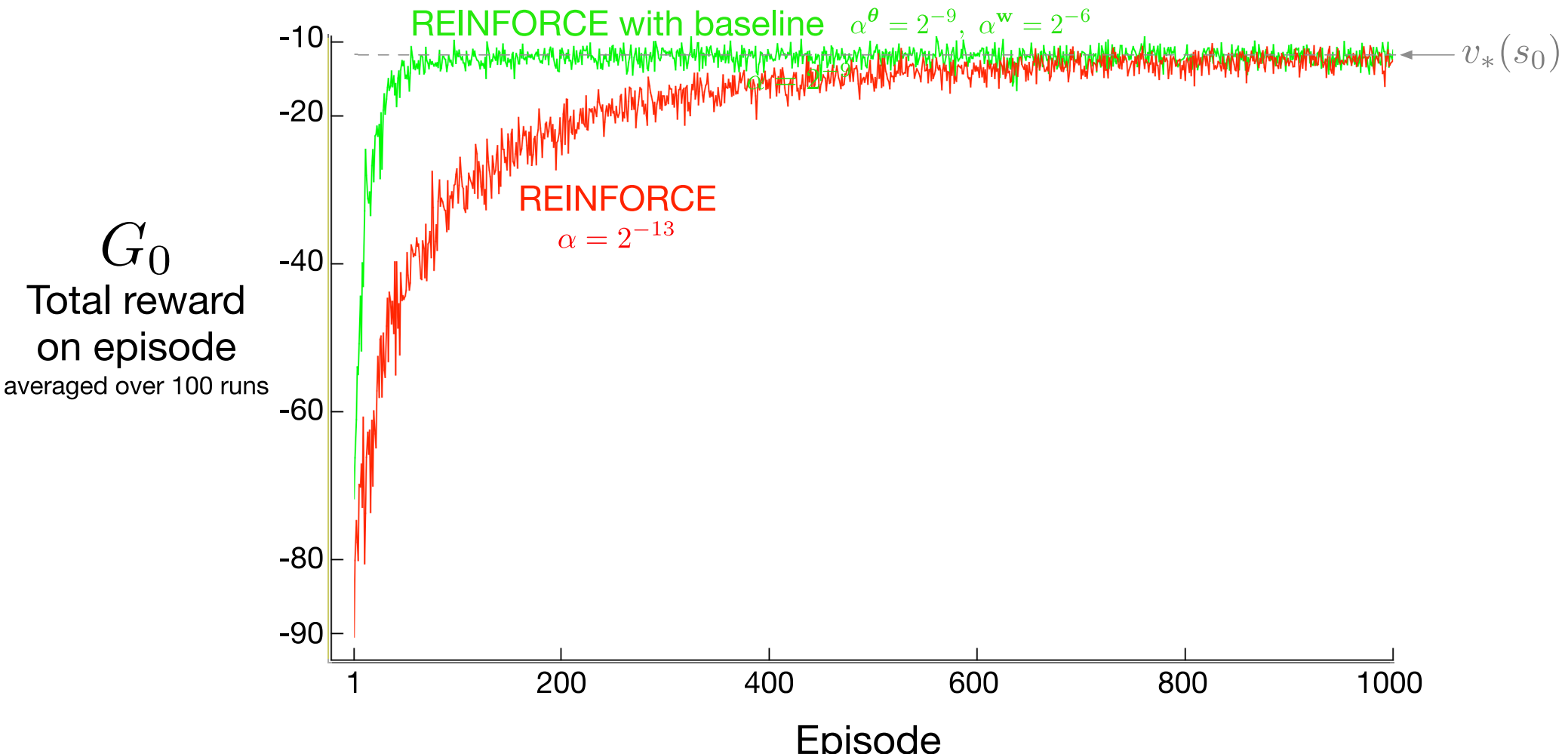


Figure 13.2 of "Reinforcement Learning: An Introduction, Second Edition".