Speech Synthesis, Reinforcement Learning

Milan Straka

May 13, 2019
Our goal is to model speech, using a auto-regressive model

\[ p(x) = \prod_{t} p(x_t | x_{t-1}, \ldots, x_1). \]

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

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

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 $\mu$-law transformation and quantize the samples into 256 values using

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

Gated Activation

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

$$z = \tanh(W_f \ast x) \cdot \sigma(W_g \ast x).$$
Figure 4: Overview of the residual block and the entire architecture.

Global Conditioning

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

$$ z = \tanh(W_f \ast x + V_f h) \cdot \sigma(W_g \ast x + V_g h). $$

Local Conditioning

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

$$ z = \tanh(W_f \ast x + V_f y) \cdot \sigma(W_g \ast x + V_g y). $$
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 $1 \times 1$ 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.

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

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

The logistic distribution is a distribution with a $\sigma$ as cumulative density function (where the mean and steepness is parametrized by $\mu$ 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 $\infty$).
Auto-regressive (sequential) inference is extremely slow in WaveNet. Instead, we use a following trick. We will model $p(x_t)$ as $p(x_t | z_{<t})$ for a random $z$ drawn from a logistic distribution. Then, we compute

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

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

$$x_t^i = x_{t-1}^i \cdot s^i (x_{<t}^{i-1}) + \mu^i (x_{<t}^{i-1}).$$
Parallel WaveNet

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

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

Figure 1 of paper "Natural TTS Synthesis by...", https://arxiv.org/abs/1712.05884.
<table>
<thead>
<tr>
<th>System</th>
<th>MOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parametric</td>
<td>3.492 ± 0.096</td>
</tr>
<tr>
<td>Tacotron (Griffin-Lim)</td>
<td>4.001 ± 0.087</td>
</tr>
<tr>
<td>Concatenative</td>
<td>4.166 ± 0.091</td>
</tr>
<tr>
<td>WaveNet (Linguistic)</td>
<td>4.341 ± 0.051</td>
</tr>
<tr>
<td>Ground truth</td>
<td>4.582 ± 0.053</td>
</tr>
<tr>
<td>Tacotron 2 (this paper)</td>
<td>4.526 ± 0.066</td>
</tr>
</tbody>
</table>

Table 1 of paper "Natural TTS Synthesis by…", https://arxiv.org/abs/1712.05884.
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
  - 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
Multi-armed Bandits

Figure 2.1 of "Reinforcement Learning: An Introduction, Second Edition".
Multi-armed Bandits

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 $q_\ast(a)$ be the real value of an action $a$:

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

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 $q_\ast(a)$. A natural way to estimate $Q_t(a)$ is

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

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

$$A_t(a) \overset{\text{def}}{=} \arg \max_a Q_t(a).$$
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 $\epsilon$-greedy method follows the greedy action with probability $1 - \epsilon$, and chooses a uniformly random action with probability $\epsilon$. 
**ε-greedy Method**

Figure 2.2 of "Reinforcement Learning: An Introduction, Second Edition".
**ε-greedy Method**

**Incremental Implementation**

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

\[
Q_{n+1} = \frac{1}{n} \sum_{i=1}^{n} R_i
\]

\[
= \frac{1}{n} (R_n + \frac{n-1}{n-1} \sum_{i=1}^{n-1} R_i)
\]

\[
= \frac{1}{n} (R_n + (n-1)Q_n)
\]

\[
= \frac{1}{n} (R_n + nQ_n - Q_n)
\]

\[
= Q_n + \frac{1}{n} (R_n - Q_n)
\]
**A simple bandit algorithm**

Initialize, for $a = 1$ to $k$:

$$Q(a) \leftarrow 0$$
$$N(a) \leftarrow 0$$

Loop forever:

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

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

Algorithm 2.4 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,
- \(p(S_{t+1} = s' | S_t = s, A_t = a)\) 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 \(G_t \overset{\text{def}}{=} \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}\). The goal is to optimize \(\mathbb{E}[G_0]\).
To formulate $n$-armed bandits problem as MDP, we do not need states. Therefore, we could formulate it as:

- one-element set of states, $S = \{S\}$;
- an action for every arm, $A = \{a_1, a_2, \ldots, a_n\}$;
- 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 $\text{next}(s)$, the MDP dynamics function for contextualized bandits problem is given by

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

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

An action-value function for a policy $\pi$ is defined analogously as

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

Evidently,

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

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

Optimal state-value function is defined as

\[ v_*(s) \overset{\text{def}}{=} \max_\pi v_\pi(s), \]

analogously

\[ q_*(s, a) \overset{\text{def}}{=} \max_\pi q_\pi(s, a). \]

Any policy \( \pi_* \) with \( v_{\pi_*} = v_* \) is called an optimal policy. Such policy can be defined as

\[ \pi_*(s) \overset{\text{def}}{=} \arg\max_a q_*(s, a) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]. \]

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 $S$ 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 $v$. 
Monte Carlo Methods

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 occurrence of a state-action pair in an episode is considered
- *every visit*: all occurrences 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:
\[
\begin{align*}
\pi(s) & \in \mathcal{A}(s) \text{ (arbitrarily), for all } s \in S \\
Q(s, a) & \in \mathbb{R} \text{ (arbitrarily), for all } s \in S, a \in \mathcal{A}(s) \\
Returns(s, a) & \leftarrow \text{empty list, for all } s \in S, a \in \mathcal{A}(s)
\end{align*}
\]

Loop forever (for each episode):
Choose \( S_0 \in S, A_0 \in \mathcal{A}(S_0) \) randomly such that all pairs have probability \( > 0 \)
Generate an episode from \( S_0, A_0, \), following \( \pi: S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T \)
\( G \leftarrow 0 \)
Loop for each step of episode, \( t = T-1, T-2, \ldots, 0 \):
\[
\begin{align*}
G & \leftarrow \gamma G + R_{t+1} \\
\text{Append } G & \text{ to } Returns(S_t, A_t) \\
Q(S_t, A_t) & \leftarrow \text{average}(Returns(S_t, A_t)) \\
\pi(S_t) & \leftarrow \arg \max_a Q(S_t, a)
\end{align*}
\]

Modification (no first-visit) of algorithm 5.3 of "Reinforcement Learning: An Introduction, Second Edition".
Monte Carlo and $\varepsilon$-soft Policies

A policy is called $\varepsilon$-soft, if

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

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

We call a policy $\varepsilon$-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 $\varepsilon$-soft policies, and using $\varepsilon$-greedy policy in policy improvement step, policy iteration has the same convergence properties. (We can embed the $\varepsilon$-soft behaviour “inside” the environment and prove equivalence.)
Monte Carlo for $\varepsilon$-soft Policies

On-policy every-visit Monte Carlo for $\varepsilon$-soft Policies

Algorithm parameter: small $\varepsilon > 0$

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

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

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 $\varepsilon$, generate a random uniform action
  - Otherwise, set $A_t \overset{\text{def}}{=} \arg \max_a Q(S_t, a)$

- $G \leftarrow 0$

- For each $t = T - 1, T - 2, \ldots, 0$:
  - $G \leftarrow \gamma G + R_{T+1}$
  - $C(S_t, A_t) \leftarrow C(S_t, A_t) + 1$
  - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{C(S_t, A_t)}(G - Q(S_t, A_t))$
Instead of predicting expected returns, we could train the method to directly predict the policy

$$\pi(a|s; \theta).$$

Obtaining the full distribution over all actions would also allow us to sample the actions according to the distribution $\pi$ instead of just $\varepsilon$-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 $\nabla_\theta v_\pi(s)$. 
In addition to discarding $\varepsilon$-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).

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

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

Then

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

and

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

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

\[ \nabla v_\pi(s) = \nabla \left[ \sum_a \pi(a|s; \theta)q_\pi(s, a) \right] \]

\[ = \sum_a \left[ \nabla \pi(a|s; \theta)q_\pi(s, a) + \pi(a|s; \theta)\nabla q_\pi(s, a) \right] \]

\[ = \sum_a \left[ \nabla \pi(a|s; \theta)q_\pi(s, a) + \pi(a|s; \theta)\nabla \left( \sum_{s'} p(s'|s, a)(r + v_\pi(s')) \right) \right] \]

\[ = \sum_a \left[ \nabla \pi(a|s; \theta)q_\pi(s, a) + \pi(a|s; \theta) \left( \sum_{s'} p(s'|s, a) \nabla v_\pi(s') \right) \right] \]

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

\[ = \sum_a \left[ \nabla \pi(a|s; \theta)q_\pi(s, a) + \pi(a|s; \theta) \left( \sum_{s'} p(s'|s, a) \left( \sum_{a'} \left[ \nabla \pi(a'|s'; \theta)q_\pi(s', a') + \pi(a'|s'; \theta) \left( \sum_{s''} p(s''|s', a') \nabla v_\pi(s'') \right) \right] \right) \right) \right] \]

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

\[ \nabla v_\pi(s) = \sum_{s' \in S} P(s \rightarrow \ldots \rightarrow s'|\pi) \sum_{a \in A} q_\pi(s', a) \nabla_\theta \pi(a|s'; \theta). \]
Proof of Policy Gradient Theorem

Recall that the initial state distribution is $h(s)$ and the on-policy distribution under $\pi$ 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

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

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

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

$$= \mathbb{E}_{s \sim \mu} \sum_{a \in A} q_\pi(s, a) \nabla_\theta \pi(a|s; \theta).$$

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

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

where we used the fact that

$$\nabla_\theta \ln \pi(a|s; \theta) = \frac{1}{\pi(a|s; \theta)} \nabla_\theta \pi(a|s; \theta).$$
REINFORCE therefore minimizes the loss

\[-E_{s \sim \mu} E_{a \sim \pi} q_\pi(s, a) \nabla \ln \pi(a|s; \theta),\]

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 \(\pi_*\)

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

Loop forever (for each episode):

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

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

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

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

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

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

Of course, the $v_\pi(s)$ baseline can be only approximated. If neural networks are used to estimate $\pi(a|s; \theta)$, 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 $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s, w)$
Algorithm parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 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 $\pi(\cdot|\cdot, \theta)$
  Loop for each step of the episode $t = 0, 1, \ldots, T - 1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ \hfill $(G_t)$
    $\delta \leftarrow G - \hat{v}(S_t, w)$
    $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S_t, w)$
    $\theta \leftarrow \theta + \alpha^\theta \delta \nabla \ln \pi(A_t|S_t, \theta)$


NPFL114, Lecture 11
**REINFORCE with Baseline**

![Graph showing REINFORCE with and without baseline](image)

- **Total reward on episode** averaged over 100 runs

$G_0$

$\alpha = 2^{-13}$

$\alpha^\theta = 2^{-9}$, $\alpha^w = 2^{-6}$

$\nu_*(s_0)$

---

Figure 13.2 of "Reinforcement Learning: An Introduction, Second Edition".
• Using REINFORCE with baseline, we can design neural network architectures.
• We fix the overall architecture and design only Normal and Reduction cells.

• Every block is designed by a RNN controller generating individual operations.

Figure 3. Controller model architecture for recursively constructing one block of a convolutional cell. Each block requires selecting 5 discrete parameters, each of which corresponds to the output of a softmax layer. Example constructed block shown on right. A convolutional cell contains $B$ blocks, hence the controller contains $5B$ softmax layers for predicting the architecture of a convolutional cell. In our experiments, the number of blocks $B$ is 5.

Neural Architecture Search (NASNet) – 2017


<table>
<thead>
<tr>
<th>Model</th>
<th>image size</th>
<th># parameters</th>
<th>Mult-Adds</th>
<th>Top 1 Acc. (%)</th>
<th>Top 5 Acc. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception V2 [29]</td>
<td>224×224</td>
<td>11.2 M</td>
<td>1.94 B</td>
<td>74.8</td>
<td>92.2</td>
</tr>
<tr>
<td>NASNet-A (5 @ 1538)</td>
<td>299×299</td>
<td>10.9 M</td>
<td>2.35 B</td>
<td>78.6</td>
<td>94.2</td>
</tr>
<tr>
<td>Inception V3 [59]</td>
<td>299×299</td>
<td>23.8 M</td>
<td>5.72 B</td>
<td>78.0</td>
<td>93.9</td>
</tr>
<tr>
<td>Xception [9]</td>
<td>299×299</td>
<td>22.8 M</td>
<td>8.38 B</td>
<td>79.0</td>
<td>94.5</td>
</tr>
<tr>
<td>Inception ResNet V2 [57]</td>
<td>299×299</td>
<td>55.8 M</td>
<td>13.2 B</td>
<td>80.4</td>
<td>95.3</td>
</tr>
<tr>
<td>NASNet-A (7 @ 1920)</td>
<td>299×299</td>
<td>22.6 M</td>
<td>4.93 B</td>
<td>80.8</td>
<td>95.3</td>
</tr>
<tr>
<td>ResNeXt-101 (64 x 4d) [67]</td>
<td>320×320</td>
<td>83.6 M</td>
<td>31.5 B</td>
<td>80.9</td>
<td>95.6</td>
</tr>
<tr>
<td>PolyNet [68]</td>
<td>331×331</td>
<td>92 M</td>
<td>34.7 B</td>
<td>81.3</td>
<td>95.8</td>
</tr>
<tr>
<td>DPN-131 [8]</td>
<td>320×320</td>
<td>79.5 M</td>
<td>32.0 B</td>
<td>81.5</td>
<td>95.8</td>
</tr>
<tr>
<td>SENet [25]</td>
<td>320×320</td>
<td>145.8 M</td>
<td>42.3 B</td>
<td>82.7</td>
<td>96.2</td>
</tr>
<tr>
<td>NASNet-A (6 @ 4032)</td>
<td>331×331</td>
<td>88.9 M</td>
<td>23.8 B</td>
<td>82.7</td>
<td>96.2</td>
</tr>
</tbody>
</table>
Figure 5 of paper "Learning Transferable Architectures for Scalable Image Recognition", https://arxiv.org/abs/1707.07012.