Sample Efficient Actor-Critic with Experience Replay

作者姓名

2024 年 8 月 16 日

1 论文的创新点

这篇论文的主要创新点包括以下几个方面：

1.1 经验回放与Actor-Critic方法的结合

该论文提出了ACER（Actor-Critic with Experience Replay）算法，这是一个结合了经验回放技术的Actor-Critic强化学习方法。传统的Actor-Critic方法在样本效率上表现不佳，而通过引入经验回放，ACER显著提高了样本效率，使得算法在离线学习场景中表现更为优越。

1.2 截断重要性采样与偏差校正

为了在经验回放过程中控制方差并提高稳定性，论文引入了截断重要性采样（Truncated Importance Sampling）与偏差校正（Bias Correction）技术。这种方法通过限制重要性权重的值，减少了方差的累积，同时引入了偏差校正项以保持估计的无偏性。

1.3 多步回报的Retrace算法

论文采用了Retrace算法来估计状态-动作值函数 Q。相比传统的单步回报方法，Retrace能够通过多步回报估计显著降低策略梯度估计的偏差，并加速Critic部分的学习。这一技术极大地增强了ACER在策略梯度估计中的表现。

1.4 信赖域策略优化

为了确保策略更新的稳定性，论文引入了一种新的信赖域策略优化方法。与传统的TRPO（Trust Region Policy Optimization）方法不同，ACER通过一个平均策略网络来控制策略更新的步长，从而在保持算法效率的同时，确保策略更新的安全性和稳定性。

1.5 随机对抗网络结构

论文还提出了一种新的网络结构——随机对抗网络（Stochastic Dueling Networks, SDNs），该网络结构能够在估计 V^π 和 Q^π 时保持一致性。通过这种网络结构，ACER在连续动作空间中也能够高效学习。

1.6 总结

这些创新点结合起来，使得ACER算法在离散和连续动作空间中都表现出色，特别是在样本效率和计算效率方面，显著优于传统的A3C算法以及其他强化学习方法。这些技术的整合不仅提高了算法的性能，还扩展了Actor-Critic方法的适用范围。

2 背景和问题设置

在强化学习（Reinforcement Learning, RL）中，智能体通过与环境的交互，在离散的时间步长内逐步学习一个策略，以最大化其获得的累积回报。具体来说，智能体在每个时间步 t 观察到一个 n_x 维的状态向量 $x_t \in X \subseteq \mathbb{R}^{n_x}$，并根据策略 $\pi(a|x_t)$ 选择一个动作 a_t，从环境中获得一个奖励信号 $r_t \in \mathbb{R}$。

2.1 目标

智能体的目标是最大化期望的折扣回报 R_t，其定义为：

$$R_t = \sum_{i \geq 0} \gamma^i r_{t+i}$$

其中，折扣因子 $\gamma \in [0, 1)$ 用于平衡即时奖励和未来奖励的重要性。

2.2 状态-动作值函数和状态值函数

对于一个跟随策略 π 的智能体，定义状态-动作值函数 $Q^\pi(x_t, a_t)$ 和状态值函数 $V^\pi(x_t)$ 如下：

$$Q^\pi(x_t, a_t) = \mathbb{E}_{x_{t+1:\infty}, a_{t+1:\infty}} [R_t \mid x_t, a_t]$$

$$V^\pi(x_t) = \mathbb{E}_{a_t} [Q^\pi(x_t, a_t) \mid x_t]$$

其中，期望是相对于策略 π 生成的状态序列 $x_{t+1:\infty}$ 和动作序列 $a_{t+1:\infty}$ 而言的。状态值函数 $V^\pi(x_t)$ 通过对所有可能的动作 a_t 的 $Q^\pi(x_t, a_t)$ 取期望来计算。

2.3 优势函数

为了提供每个动作的相对价值，还需要定义优势函数 $A^\pi(x_t, a_t)$：

$$A^\pi(x_t, a_t) = Q^\pi(x_t, a_t) - V^\pi(x_t)$$

优势函数 $A^\pi(x_t, a_t)$ 表示在状态 x_t 下采取动作 a_t 相对于平均值的优势。值得注意的是，对所有动作的期望值 $\mathbb{E}_{a_t} [A^\pi(x_t, a_t)] = 0$。

2.4 策略梯度更新

可微策略 $\pi_\theta(a_t|x_t)$ 的参数 θ 可以使用折扣近似的策略梯度来更新，策略梯度定义如下：

$$g = \mathbb{E}_{x_{0:\infty}, a_{0:\infty}} \left[\sum_{t \geq 0} A^\pi(x_t, a_t) \nabla_\theta \log \pi_\theta(a_t|x_t) \right]$$

根据 Schulman 等人的研究，可以用状态-动作值函数 $Q^\pi(x_t, a_t)$、折扣回报 R_t 或时间差分残差来替换上述表达式中的 $A^\pi(x_t, a_t)$。

3 重要性加权策略梯度估计的详细解释

在离线策略学习中，我们希望使用从行为策略 μ 生成的数据来估计目标策略 π 的梯度。重要性加权策略梯度估计的核心公式如下：

$$\hat{g}_{\text{imp}} = \left(\prod_{t=0}^{k} \rho_t \right) \left(\sum_{t=0}^{k} \gamma^t r_{t+1} \right) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

其中：

- $\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$ 是重要性权重，它衡量在时间步 t 下目标策略 π 和行为策略 μ 在选择动作 a_t 上的差异。
- $\pi(a_t \mid s_t)$ 是目标策略在状态 s_t 下选择动作 a_t 的概率。
- $\mu(a_t \mid s_t)$ 是行为策略在状态 s_t 下选择动作 a_t 的概率。
- γ 是折扣因子，用于平衡当前奖励与未来奖励的相对重要性。
- r_{t+1} 是在时间步 $t+1$ 时获得的即时奖励。
- $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ 是策略的梯度，表示策略对参数 θ 的敏感性，即如何调整策略参数 θ 会影响选择动作 a_t 的概率。

3.1 公式解释

1. **重要性权重** ρ_t:
   - 重要性权重 ρ_t 的作用是调整行为策略生成的数据，使其反映目标策略的效果。
   - 当行为策略和目标策略不同时，直接使用行为策略生成的数据来估计目标策略的期望值会有偏差。重要性权重通过重新加权这些样本，消除这种偏差。
2. **梯度估计** $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$:
   - 这是策略梯度方法的核心部分，它表示在当前策略参数 θ 下，选择动作 a_t 的对数概率的梯度。
   - 这个梯度用于更新策略参数，以使策略能够更好地选择那些能够获得高回报的动作。
3. **回报估计** $\sum_{t=0}^{k} \gamma^t r_{t+1}$:
   - 这部分表示从时间步 t 开始的累积回报，考虑了未来 k 个时间步的回报。
   - 折扣因子 γ 用于控制未来回报对当前回报的影响，γ 越大，未来回报的影响越大。
4. **整体结构**:
   - 整个公式表示为从时间步 0 到 k 的梯度和加权累积回报的乘积，再乘以所有时间步的权重乘积。

- 这个估计是无偏的，但由于权重的乘积可能导致非常大的方差，特别是在长时间序列上，这个公式在实际应用中可能会不稳定。

4 从公式3到公式4的推导与分析

在本节中，我们详细解释了如何从公式3推导出公式4，并讨论了引入边际值函数和限制分布的重要性。这些概念在处理离线策略学习中的高方差问题时起到了关键作用。

4.1 公式 3：重要性加权的策略梯度估计

首先，我们来看公式3，这是基于重要性加权的方法来估计策略梯度：

$$\hat{g}_{\text{imp}} = \left(\prod_{t=0}^{k} \rho_t\right) \left(\sum_{t=0}^{k} \gamma^t r_{t+1}\right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

其中：

- $\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$ 是重要性权重，用于调整行为策略 μ 和目标策略 π 之间的差异。
- γ 是折扣因子，r_{t+1} 是时间步 $t+1$ 的即时奖励。
- $\nabla_\theta \log \pi_\theta(a_t|s_t)$ 是策略梯度，表示策略对参数 θ 的敏感性。

公式3通过重要性权重调整了行为策略生成的数据，使其更好地反映目标策略。然而，这个公式面临的主要问题是高方差。原因在于多个时间步的权重 ρ_t 的乘积可能会引入数值不稳定性，尤其是在长时间序列上。

4.2 从公式3到公式4的推导

为了减少高方差问题并获得更稳定的策略梯度估计，我们引入了边际值函数 $Q^*(s,a)$ 和限制分布 $\beta(s)$ 这两个关键概念。

4.2.1 边际值函数的引入

边际值函数 $Q^*(s,a)$ 是在一个限制分布 $\beta(s)$ 下计算的长期期望回报。它表示在状态 s 下选择动作 a 后，在长期稳定的状态分布下遵循目标策略 π 的期望累积回报：

$$Q^*(s,a) = \mathbb{E}_\beta \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_t = s, a_t = a\right]$$

通过使用边际值函数 $Q^*(s,a)$ 代替传统的值函数 $Q^\pi(s,a)$，我们避免了由于初始状态和策略不稳定性引起的高方差问题。

4.2.2 限制分布的应用

限制分布 $\beta(s)$ 是在行为策略 μ 下，经过长时间的交互后，状态 s 达到的平稳分布。使用限制分布 $\beta(s)$ 进行期望计算，意味着我们关注的是系统在长期运行后的稳态行为，而不是某个特定初始条件下的短期行为。

4.3 公式 4：改进的策略梯度估计

在引入边际值函数和限制分布后，我们可以将公式3中的重要性加权策略梯度估计转换为公式4：

$$g_{\text{marg}} = \mathbb{E}_{x_t, a_t \sim \mu} \left[\rho_t \nabla_\theta \log \pi_\theta(a_t|s_t) Q^*(s_t, a_t)\right]$$

公式4的引入简化了重要性权重的处理，使得梯度估计更具鲁棒性，特别是在离线策略学习中，它能够有效减少由初始状态和短期策略波动引起的估计不稳定性。

5 公式5与公式6的详细解析

在本节中，我们结合参数 θ_v，详细解释了公式5和公式6的含义及其相互关系，特别是在使用 Retrace 算法进行状态-动作值函数估计和优化策略梯度的过程中。

5.1 公式5的详细解释

公式5描述了如何使用 Retrace 算法递归地估计状态-动作值函数 $Q^\pi(s,a)$，它表示在状态 s_t 下执行动作 a_t 后，在遵循策略 π 时，未来所有可能的回报的折现和。公式5的表达式如下：

$$Q^{\text{ret}}(x_t, a_t) = r_t + \gamma \bar{\rho}_{t+1} \left[Q^{\text{ret}}(x_{t+1}, a_{t+1}) - Q(x_{t+1}, a_{t+1})\right] + \gamma V(x_{t+1})$$

其中，$Q^{\text{ret}}(x_t, a_t)$ 是通过 Retrace 估计的状态-动作值函数，r_t 是即时奖励，γ 是折扣因子，$\bar{\rho}_{t+1}$ 是截断的重要性权重，$Q(x_{t+1}, a_{t+1})$ 是当前策略参数 θ_v 下的值函数估计，$V(x_{t+1})$ 是状态 x_{t+1} 的值函数。

公式5通过递归的方式结合即时奖励、未来状态的值函数，以及截断的重要性权重来修正当前的值函数估计，从而稳定地逼近目标策略的值函数。

5.2 公式6的详细解释

公式6利用公式5中的估计值 $Q^{\text{ret}}(x_t, a_t)$，来优化当前策略参数 θ_v 下的值函数 $Q_{\theta_v}(x_t, a_t)$。公式6的表达式为：

$$(Q^{\text{ret}}(x_t, a_t) - Q_{\theta_v}(x_t, a_t))\nabla_{\theta_v} Q_{\theta_v}(x_t, a_t)$$

公式6的核心思想是通过最小化当前估计值 $Q_{\theta_v}(x_t, a_t)$ 与目标值 $Q^{\text{ret}}(x_t, a_t)$ 之间的差异来优化策略参数 θ_v。这个差异被称为“TD误差”（Temporal Difference Error），优化目标是使得这个误差尽可能小，从而通过梯度下降法更新 θ_v 来提高策略的性能。

5.3 公式5到公式6的推导与关系

公式5通过多步回报递归估计状态-动作值函数，而公式6则利用这一估计值作为目标，通过最小化与当前网络输出之间的误差，来优化策略参数 θ_v。这两个公式结合了递归回报估计和梯度优化方法，使得强化学习中的策略优化更加高效和稳健。

6 公式7、8、9的推导与解释

本节详细解释了公式7、8、9及其推导过程，探讨了在离线策略学习中使用重要性采样进行策略梯度估计时，如何通过截断重要性权重并引入偏差校正来控制高方差问题，并利用神经网络进行值函数的估计。这些公式逐步引入了截断、偏差校正和神经网络近似，以构建一个稳健的策略梯度估计框架。

6.1 公式7的推导与解释

公式7的推导主要涉及重要性权重的截断和偏差校正的引入。公式7最初表达为：

$$g_{\text{marg}} = \mathbb{E}_{x_t \sim \mu} \left[\rho_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) Q^\pi(x_t, a_t) \right]$$

这里，$\rho_t = \frac{\pi_\theta(a_t|x_t)}{\mu(a_t|x_t)}$ 是重要性权重，用于调整行为策略生成的数据以匹配目标策略。

6.1.1 引入截断和偏差校正

由于原始重要性权重 ρ_t 可能导致高方差问题，因此引入截断重要性权重 $\bar{\rho}_t$：

$$\bar{\rho}_t = \min(c, \rho_t)$$

为了确保截断后的梯度估计是无偏的，我们引入了偏差校正项：

$$\frac{\rho_t(a) - c}{\rho_t(a)} \mathbb{I}_{\rho_t(a) > c}$$

这个校正项确保了当 $\rho_t(a) > c$ 时，策略梯度的估计仍然是无偏的。

6.1.2 公式7的最终表达

结合上述的截断和偏差校正，公式7被推导为：

$$g_{\text{marg}} = \mathbb{E}_{x_t \sim \mu} \left[\bar{\rho}_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) Q^\pi(x_t, a_t) \right] + \mathbb{E}_{x_t \sim \mu} \left[\mathbb{E}_{a \sim \pi_\theta} \left[\frac{\rho_t(a) - c}{\rho_t(a)} \mathbb{I}_{\rho_t(a) > c} \nabla_\theta \log \pi_\theta(a \mid x_t) Q^\pi(x_t, a_t) \right] \right]$$

这两个期望值分别表示：

- **主要的梯度估计**：基于截断后的重要性权重 $\bar{\rho}_t$。
- **偏差校正项**：对超出截断阈值的部分进行校正。

6.2 公式8的推导与解释

公式8在公式7的基础上进一步扩展，结合了神经网络近似 $Q_\theta(x_t, a_t)$ 来估计值函数。公式8的表达为：

$$g_{\text{marg}} = \mathbb{E}_{x_t \sim \mu} \left[\bar{\rho}_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) Q^{\text{ret}}(x_t, a_t) \right] + \mathbb{E}_{x_t \sim \mu} \left[\mathbb{E}_{a \sim \pi_\theta} \left[\frac{\rho_t(a) - c}{\rho_t(a)} \mathbb{I}_{\rho_t(a) > c} \nabla_\theta \log \pi_\theta(a \mid x_t) Q_\theta(x_t, a_t) \right] \right]$$

6.2.1 引入神经网络近似

在公式8中，$Q^\pi(x_t, a_t)$ 被 $Q^{\text{ret}}(x_t, a_t)$ 替代，后者是通过 Retrace 算法计算得到的多步回报的累积值。此外，使用神经网络近似 $Q_\theta(x_t, a_t)$ 来替代部分回报估计，使得模型能够更好地捕捉复杂的值函数特性。

6.2.2 公式8的最终表达

公式8因此扩展了公式7，通过神经网络近似进一步增强了策略梯度的估计精度，同时保持了偏差校正的无偏性。

6.3 公式9的推导与解释

公式9在公式8的基础上进一步引入了ACER（Actor-Critic with Experience Replay）算法的具体框架。公式9的表达为：

$$\hat{g}_t^{\text{ACER}} = \bar{\rho}_t \nabla_\theta \log \pi_\theta(a_t \mid x_t) \left[Q^{\text{ret}}(x_t, a_t) - V_\theta(x_t) \right] + \mathbb{E}_{a \sim \pi_\theta} \left[\frac{\rho_t(a) - c}{\rho_t(a)} \mathbb{I}_{\rho_t(a) > c} \nabla_\theta \log \pi_\theta(a \mid x_t) \left[Q_\theta(x_t, a_t) - V_\theta(x_t) \right] \right]$$

6.3.1 引入基线以减少方差

在公式9中，状态值函数 $V_\theta(x_t)$ 被引入作为基线（Baseline），用于减少策略梯度估计中的方差。基线技术通过减去状态值函数，能够显著降低策略梯度估计的方差，提高学习的稳定性。

6.3.2 公式9的最终表达

公式9结合了截断、偏差校正、神经网络近似以及基线技术，构成了ACER算法中的策略梯度估计框架。它有效地控制了策略优化中的方差和偏差问题。

6.4 公式7、8、9的推导关系

- **从公式7到公式8**：公式7通过截断和偏差校正，提供了一个基本的策略梯度估计方法。公式8在此基础上，引入神经网络来近似值函数，使得模型具有更强的表达能力，并能够处理更复杂的环境。
- **从公式8到公式9**：公式9在公式8的基础上，将这些技术应用于ACER算法中，进一步优化策略梯度估计。特别是引入了基线 $V_\theta(x_t)$ 技术，以减少方差，增强学习的稳定性和效率。

公式7、8、9通过逐步引入截断、偏差校正、神经网络近似和基线技术，构建了一个稳健的策略梯度估计框架。这些公式逐步递进，最终形成了适用于复杂强化学习环境的ACER算法的策略优化方法。

7 由公式7、8、9推导到公式10、11、12的过程与解释

在本节中，我们将详细解释如何通过引入信赖域（Trust Region）的概念，从公式7、8、9推导到公式10、11、12。信赖域的引入旨在限制策略更新的幅度，以确保策略优化的稳定性。

7.1 从公式7、8、9到公式10的推导

公式7、8、9定义了ACER算法中的策略梯度估计，其中重要性采样、截断和偏差校正，以及神经网络近似被用于估计策略梯度。公式7给出了基础的策略梯度估计，而公式8引入了神经网络近似，并用Retrace算法估计状态-动作值函数。公式9进一步结合基线（Baseline）技术减少方差，将这些思想应用于ACER算法中。

为了更好地控制策略更新的幅度，公式10引入了信赖域的思想。公式10中的策略梯度表达式为：

$$\hat{g}_t^{\text{ACER}} = \bar{\rho}_t \nabla_\phi \log f(a_t \mid \phi(x_t)) \left[Q^{\text{ret}}(x_t, a_t) - V_\theta(x_t) \right] + \mathbb{E}_{a \sim \pi_\phi} \left[\frac{\rho_t(a) - c}{\rho_t(a)} \mathbb{I}_{\rho_t(a) > c} \nabla_\phi \log f(a \mid \phi(x_t)) \left[Q_\theta(x_t, a_t) - V_\theta(x_t) \right] \right]$$

在公式10中，梯度相对于策略参数 ϕ 进行计算，而不是直接相对于策略参数 θ。这意味着在更新策略时，考虑了信赖域的约束。信赖域优化通过限制更新幅度，确保策略变化不会太大，以避免学习过程中策略的剧烈波动。

7.2 从公式10到公式11的推导

公式11定义了优化问题，用于在考虑KL散度（Kullback-Leibler Divergence）约束的条件下，最小化策略梯度的更新幅度：

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \| \hat{g}_t^{\text{ACER}} - z \|_2^2 \\ \text{subject to} \quad & \nabla_\phi D_{\text{KL}} \left[f(\cdot \mid \phi_{\theta_a}(x_t)) \| f(\cdot \mid \phi(x_t)) \right]^T z \leq \delta \end{aligned}$$

在这个优化问题中，引入了KL散度约束来控制新旧策略之间的差异。KL散度测量了两个分布之间的相似度，因此，通过限制KL散度，可以控制策略更新的幅度，使得新策略不会偏离旧策略太多。

7.3 从公式11到公式12的推导

公式12给出了优化问题的解，它在满足约束条件的情况下，提供了最优的策略更新步长：

$$z^* = \hat{g}_t^{\text{ACER}} - \max \left(0, \frac{\nabla_\phi D_{\text{KL}} \left[f(\cdot \mid \phi_{\theta_a}(x_t)) \| f(\cdot \mid \phi(x_t)) \right]^T \hat{g}_t^{\text{ACER}} - \delta}{\| \nabla_\phi D_{\text{KL}} \left[f(\cdot \mid \phi_{\theta_a}(x_t)) \| f(\cdot \mid \phi(x_t)) \right] \|_2^2} \right) \times \nabla_\phi D_{\text{KL}} \left[f(\cdot \mid \phi_{\theta_a}(x_t)) \| f(\cdot \mid \phi(x_t)) \right]$$

公式12是通过拉格朗日乘子法和KKT（Karush-Kuhn-Tucker）条件推导出来的。当策略的更新不满足KL散度约束时，通过调整更新方向的大小，使得策略更新保持在信赖域内。公式12通过最大化约束条件的余量来决定更新步长。如果更新幅度超出允许范围，则通过缩放策略梯度，确保更新步长在可接受的范围内。

7.4 总结

公式10、11、12通过引入信赖域优化思想，将策略更新限制在一个安全范围内，以确保策略优化的稳定性。

这些公式从基础的策略梯度估计（公式7、8、9）开始，通过引入策略参数的变化控制、KL散度约束和二次优化问题，逐步推导出最终的策略更新方案（公式12）。这些推导确保了策略更新过程中的稳定性和高效性，同时兼顾了策略更新的灵活性。

A 优势函数、状态-动作值函数和状态值函数的区别

在强化学习（Reinforcement Learning, RL）中，优势函数、状态-动作值函数（Q 函数）和状态值函数（V 函数）是关键的概念，它们帮助我们理解和计算智能体在环境中的表现。以下是对这些函数的定义和区别的详细解释。

A.1 状态值函数 $V^\pi(S)$

定义: 状态值函数 $V^\pi(s)$ 表示在状态 s 下，智能体遵循策略 π 时，未来获得的期望累积回报。即：

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]$$

其中，γ 是折扣因子，用于平衡当前和未来奖励的相对重要性，r_t 是时间步 t 的奖励。

用途: $V^\pi(s)$ 评估的是在给定状态下，智能体总体的“好坏”程度。它只与当前状态和策略相关，而不直接考虑特定的动作。

A.2 状态-动作值函数 $Q^\pi(S, a)$

定义: 状态-动作值函数 $Q^\pi(s, a)$ 表示在状态 s 下执行动作 a，然后在未来遵循策略 π 时，期望获得的累积回报。即：

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right]$$

用途: $Q^\pi(s, a)$ 用于评估在特定状态下采取特定动作的好坏程度。它不仅考虑了当前状态，还直接与智能体选择的动作相关。

A.3 优势函数 $A^\pi(S, a)$

定义: 优势函数 $A^\pi(s, a)$ 表示在状态 s 下采取动作 a 相对于在该状态下遵循策略 π 的平均行为的相对优势。即：

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

其中，$Q^\pi(s, a)$ 是在状态 s 下采取动作 a 的期望回报，$V^\pi(s)$ 是在状态 s 下跟随策略 π 的期望回报。

用途: 优势函数 $A^\pi(s, a)$ 用于衡量一个动作在当前状态下的相对价值。如果 $A^\pi(s, a) > 0$，意味着这个动作比策略中通常选择的动作更优；反之，如果 $A^\pi(s, a) < 0$，则意味着这个动作比策略中的平均选择要差。

A.4 三者的区别和联系

联系: - $Q^\pi(s, a)$ 和 $V^\pi(s)$ 都是期望回报的度量，它们分别关注的是状态-动作对和状态。 - 优势函数 $A^\pi(s, a)$ 则是通过 $Q^\pi(s, a)$ 和 $V^\pi(s)$ 的差值来衡量动作的相对价值。

区别: - $V^\pi(s)$ 仅仅关注状态，不关心在该状态下采取的具体动作。 - $Q^\pi(s, a)$ 关注的是在特定状态下选择特定动作的长期回报。 - $A^\pi(s, a)$ 强调的是在特定状态下，某个动作相对于该状态下策略的平均表现的优劣程度。

通过这些函数，强化学习算法可以更好地引导智能体做出在长远来看最优的决策。

B 策略梯度在强化学习中的作用

策略梯度（Policy Gradient）在强化学习中扮演着至关重要的角色，尤其是在策略优化和策略搜索方法中。它提供了一种直接优化策略的途径，使得智能体能够在连续动作空间或复杂的策略框架下进行有效学习。以下是策略梯度在强化学习中的主要作用和意义。

B.1 直接优化策略

强化学习中通常有两种主要的学习范式：值函数方法和策略梯度方法。

- **值函数方法**：例如 Q-learning，通过估计每个状态或状态-动作对的值函数，间接推导出最优策略。这种方法通常适用于离散动作空间。

- **策略梯度方法**：直接通过优化策略的参数化表示来获得最优策略，适用于复杂和连续的动作空间。策略梯度方法优化的目标是最大化智能体在环境中长期的期望回报。

B.2 在连续动作空间中的应用

在一些复杂的环境中，动作空间是连续的，无法简单地通过枚举所有可能的动作来寻找最优动作。策略梯度方法可以直接对参数化的策略进行优化，而不需要明确地定义值函数，这使得它非常适用于连续动作空间。

例如，假设策略由参数 θ 控制，策略梯度方法的目标是通过调整 θ 来最大化以下目标函数：

$$J(\theta) = \mathbb{E}_{\pi}[R]$$

这里，R 是累积回报，策略梯度通过计算 $\nabla_\theta J(\theta)$ 来优化策略参数 θ。

B.3 处理高维和复杂策略

在高维或复杂策略的场景中，传统的值函数方法往往很难有效地学习。策略梯度方法允许对复杂策略进行直接优化，而不需要先估计值函数。通过策略的参数化表示（例如，神经网络），策略梯度方法可以处理更为复杂的策略结构。

B.4 软策略和探索-利用平衡

策略梯度方法还支持软策略（Soft Policy），即策略不是直接选择最优动作，而是以某种概率分布来选择动作。这种方式不仅允许策略进行探索，还能更好地平衡探索和利用之间的关系。例如，软策略会在动作选择上进行随机化，从而避免陷入局部最优解。

B.5 常见的策略梯度方法

策略梯度方法有多种变体和扩展，它们在不同场景中得到了广泛应用：

- **REINFORCE**：最经典的策略梯度算法，通过直接使用蒙特卡洛方法计算梯度并更新策略参数。
- **Actor-Critic**：结合了值函数（Critic）和策略（Actor）的方法，其中 Critic 用来估计值函数，Actor 用来直接优化策略参数。
- **TRPO 和 PPO**：是对策略梯度方法的改进，它们通过引入信赖域优化（Trust Region Optimization）来限制每次更新的步长，从而避免策略更新过大导致不稳定性。

B.6 优化目标函数

策略梯度方法的核心是通过计算目标函数 $J(\theta)$ 的梯度来优化策略参数。通常这个梯度是通过以下公式计算的：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s) Q^\pi(s,a)]$$

这里的 $Q^\pi(s,a)$ 是状态-动作值函数，表示在状态 s 下采取动作 a 后的期望回报。

B.7 提高样本效率

在策略梯度方法中，通过使用经验回放（Experience Replay）和重要性采样（Importance Sampling）等技术，可以有效提高样本效率。这些方法允许智能体利用过去的经验来改进当前的策略，从而减少对新样本的依赖，加速学习过程。

B.8 总结

策略梯度在强化学习中提供了一种强大且灵活的策略优化方法，尤其适用于复杂、连续的动作空间。通过直接优化策略参数，策略梯度方法能够在多样化的环境中实现有效的策略学习。它的核心作用包括直接优化策略、处理复杂和高维策略结构、支持软策略以及提高样本效率等。这使得策略梯度方法成为许多先进强化学习算法的基础，例如 Actor-Critic 方法、TRPO、PPO 等。

C 策略梯度更新的详细解释

策略梯度更新是强化学习中一种常见的优化方法，用于通过调整策略参数来最大化智能体的期望累积回报。以下是对策略梯度更新的详细解释，从基础概念开始，逐步深入。

C.1 强化学习中的基本概念

在强化学习中，智能体（Agent）通过与环境（Environment）的交互，学习一个策略（Policy）来决定在每个状态下采取的动作。主要的目标是最大化累积回报。强化学习的基本要素包括：

- **状态（State,** s）：描述环境当前的情况。状态通常表示为一个向量 s_t。

- **动作（Action,** a）：智能体在某个状态 s_t 下可以选择的行为。动作 a_t 决定了智能体在当前状态下的反应。
- **奖励（Reward,** r）：在某个状态 s_t 下执行动作 a_t 后，智能体从环境中获得的反馈，表示当前动作的好坏。
- **策略（Policy,** π）：决定智能体在每个状态下选择动作的规则。策略可以是确定性的，也可以是随机的（即给定状态 s，动作的选择是一个概率分布）。

C.2 累积回报（Cumulative Reward）

智能体的目标是最大化从初始状态到未来无限时间步内获得的累积回报。通常使用折扣因子 $\gamma \in [0,1)$ 来计算累积回报：

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

其中，γ 是折扣因子，用于平衡当前奖励和未来奖励的重要性。通常，γ 越接近于 1，智能体越重视未来的奖励。

C.3 策略梯度方法的基本思想

策略梯度方法的核心思想是通过直接优化策略函数 π_θ 来最大化期望累积回报。策略函数 $\pi_\theta(a|s)$ 通常是一个参数化的函数，表示在状态 s 下选择动作 a 的概率。策略的参数用 θ 表示。

为了优化策略，我们定义一个目标函数 $J(\theta)$，通常为累积回报的期望：

$$J(\theta) = \mathbb{E}_{\pi_\theta}[R]$$

策略梯度方法通过计算这个目标函数关于策略参数 θ 的梯度，并使用梯度上升法（或下降法）来更新参数，从而使策略逐渐变得更优。

C.4 策略梯度定理

根据策略梯度定理，目标函数 $J(\theta)$ 的梯度可以表示为：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)\right]$$

其中：

- $\nabla_\theta \log \pi_\theta(a_t|s_t)$ 表示策略函数的对数关于参数 θ 的梯度，表示策略在当前状态下选择某一动作的敏感性。
- $Q^\pi(s_t, a_t)$ 表示状态-动作值函数，衡量在状态 s_t 下执行动作 a_t 的长期回报。

该公式意味着：我们通过更新策略参数 θ 来增加那些在给定状态下产生高回报的动作的概率，同时减少那些产生低回报的动作的概率。

C.5 策略梯度的更新规则

策略参数 θ 的更新遵循梯度上升法的基本规则：

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

其中，α 是学习率（Learning Rate），控制每次更新的步幅大小。

C.6 为什么策略梯度有效？

策略梯度方法之所以有效，是因为它直接基于回报来优化策略。相比于只依赖状态值函数 $V(s)$ 的方法，策略梯度方法考虑了策略对动作选择的实际影响，并调整策略以直接提高期望回报。

C.7 实际应用中的策略梯度

在实际应用中，策略梯度可能会结合许多优化技巧，如：

- **优势函数（Advantage Function,** $A^\pi(s,a)$）：用来替代 $Q^\pi(s_t, a_t)$，减少方差。
- **近端策略优化（Proximal Policy Optimization, PPO）**：一种常用的策略梯度优化算法，通过引入信任区域来限制每次更新的步幅，确保策略更新的稳定性。

D 目标策略与行为策略的区别与联系

在强化学习中，目标策略（Target Policy, π）和行为策略（Behavior Policy, μ）是两个关键概念，尤其是在离线策略学习（Off-policy learning）中，这两个策略起着至关重要的作用。

D.1 目标策略（Target Policy, π）

定义：目标策略是强化学习中我们希望学习和优化的策略。它决定了智能体在每个状态下应该采取的动作，以最大化长期回报。

作用：目标策略是我们希望优化的对象，我们希望通过强化学习找到一个最优的目标策略，使得智能体能够在环境中表现得最好，即获得最大化的累积回报。

符号：在许多强化学习算法中，目标策略通常用 π 表示。例如，$\pi(a \mid s)$ 表示在状态 s 下选择动作 a 的概率。

D.2 行为策略（Behavior Policy, μ）

定义：行为策略是智能体在与环境交互时实际使用的策略。它决定了在每个状态下智能体实际选择的动作。

作用：行为策略通常是用于收集训练数据的策略。在离线策略学习中，智能体可能使用一个行为策略与环境交互，生成一组状态-动作-奖励的样本数据。

符号：行为策略通常用 μ 表示。例如，$\mu(a \mid s)$ 表示在状态 s 下选择动作 a 的概率。

D.3 目标策略和行为策略的关系

在强化学习中，目标策略和行为策略的区别和联系主要体现在以下几个方面：

- **策略优化**：我们希望通过强化学习算法来优化目标策略 π，使其在环境中表现更好。然而，直接从目标策略采样数据可能会非常昂贵或难以实现。
- **数据采集**：因此，通常使用行为策略 μ 来收集数据，这样智能体可以在行为策略下生成大量的样本，然后使用这些样本来估计和优化目标策略。
- **离线策略学习**：在离线策略学习中，我们从行为策略 μ 中生成的数据来估计目标策略 π 的值函数。这时，重要性采样方法通过重新加权行为策略的数据，使其能够反映目标策略的预期回报。

D.4 举例说明

目标策略：例如，假设一个智能体的目标是学习一套在游戏中获胜的策略，这个策略就是目标策略。目标策略可能通过多次试验和学习最终收敛到一个可以使得智能体在游戏中表现最优的策略。

行为策略：在学习过程中，智能体可能使用一种探索性的策略（例如 epsilon-greedy 策略）与游戏环境交互，试图了解哪些动作能够获得高分。这种探索性的策略就是行为策略。

D.5 重要性采样的作用

在实际应用中，由于行为策略 μ 和目标策略 π 之间可能存在差异，直接使用行为策略的数据来估计目标策略可能会引入偏差。为了解决这个问题，我们使用重要性采样来调整行为策略生成的数据，使得我们可以从行为策略的数据中估计出目标策略的表现。

重要性采样的关键在于计算重要性权重 $\rho_t = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$，用于修正由于行为策略和目标策略不同而引入的偏差。

D.6 总结

目标策略 π：我们希望优化和学习的策略，旨在最大化智能体的长期回报。

行为策略 μ：智能体在与环境交互时实际使用的策略，主要用于收集数据。

关系：行为策略生成的数据用于估计和优化目标策略，通过重要性采样可以减小由于策略差异带来的偏差。

理解目标策略和行为策略之间的关系对于掌握强化学习中的离线策略学习方法（如 Q-learning、DQN、ACER 等）至关重要。

E 信赖域的解释

信赖域（Trust Region）是优化算法中的一种概念，用来确保在优化过程中，每次迭代的更新步长不会偏离当前解的“信赖范围”太远，从而保证算法的收敛性和稳定性。

E.1 基本概念

在许多优化问题中，直接进行大幅度的参数更新可能导致不稳定的行为，尤其是在非线性或高维度的优化问题中。这是因为目标函数的局部曲率在不同区域可能

变化很大，如果步长太大，可能会越过目标函数的低谷，甚至导致发散。

信赖域方法提出了一种策略，即在每次迭代中，先假设目标函数在当前解附近的一定范围内可以被有效近似，然后只在这个区域（即信赖域）内进行优化。

E.2 信赖域的定义

信赖域优化的基本思想是：在每次迭代中，通过在当前点 x_k 附近构造一个简化的目标函数模型 $m_k(p)$，并仅在一个信赖域 Δ_k 内找到优化方向 p_k，满足以下优化问题：

$$\min_{p \in \Delta_k} m_k(p)$$

其中，p 是搜索方向或步长，Δ_k 是信赖域的半径，通常以欧几里得距离（例如 $\|p\|_2$）定义。

信赖域的关键是通过控制 Δ_k 的大小来决定步长。如果模型 $m_k(p)$ 能够很好地近似实际的目标函数，信赖域的半径 Δ_k 可以增加；否则，需要缩小信赖域，减少步长以确保优化的稳定性。

E.3 信赖域在优化中的应用

信赖域方法在优化算法中的应用主要包括以下几个方面：

E.3.1 Quadratic Model（二次模型）

在信赖域优化中，常常使用目标函数的二阶泰勒展开作为简化模型：

$$m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$$

这里，$\nabla f(x_k)$ 是目标函数在 x_k 处的梯度，而 B_k 通常是 $\nabla^2 f(x_k)$ 的近似，即 Hessian 矩阵。信赖域优化的任务是找到在信赖域 Δ_k 内，使 $m_k(p)$ 最小化的步长 p_k。

E.3.2 信赖域半径的调整

在每次迭代后，根据当前模型 $m_k(p)$ 对实际目标函数 $f(x)$ 的近似情况来调整信赖域的半径。常见的策略包括：

- **扩大信赖域**: 如果模型近似效果很好，意味着可以信任这个模型，增大信赖域半径 Δ_k。
- **缩小信赖域**: 如果模型近似效果较差，则需要减小信赖域半径，减少步长，保证优化的稳定性。

E.3.3 应用在TRPO中的信赖域

在强化学习中的TRPO（Trust Region Policy Optimization）算法中，信赖域的概念被用于限制策略更新的幅度。具体来说，TRPO限制了新旧策略之间的KL散度（Kullback-Leibler Divergence），从而确保策略的变化在一个可以信任的范围内。这一策略大大提高了策略优化的稳定性和收敛性。

E.4 信赖域与梯度下降的比较

信赖域优化与传统的梯度下降法不同：

- **梯度下降**: 每次迭代直接沿梯度方向更新参数，步长是固定的或根据一定规则调整，但不考虑目标函数的局部几何性质。
- **信赖域方法**: 每次迭代不仅考虑更新方向，还考虑当前点附近目标函数的局部曲率，通过构造一个信赖域范围内的简化模型进行优化，从而更好地控制优化过程中的步长。

E.5 优缺点分析

E.5.1 优点

- **稳定性**: 信赖域方法通过控制每次迭代的步长，避免了大步长带来的震荡或发散问题。
- **适应性**: 信赖域的大小可以动态调整，适应目标函数局部特性的变化，从而提高了收敛速度。

E.5.2 缺点

- **计算复杂度**: 信赖域优化涉及到二次规划问题的求解，计算复杂度相对较高，特别是在高维度问题中。
- **参数选择**: 需要选择合适的信赖域半径调整策略，选择不当可能影响算法性能。

E.6 总结

信赖域是优化算法中用于控制每次迭代步长的一个重要概念，通过限制更新范围，确保优化的稳定性和效率。在强化学习和其他复杂的非线性优化问题中，信赖域方法提供了一种有效的手段来平衡模型近似与实际优化效果之间的关系。

F ACER代码分析与实现

本节将详细分析 'acer.py' 文件的实现，并结合论文中的关键创新点进行解释。

F.1 ACER算法概述

ACER（Actor-Critic with Experience Replay）算法旨在通过结合离线学习和经验回放，提高样本效率。'acer.py' 文件的核心是如何在 Actor-Critic 框架中有效地引入经验回放、重要性采样、截断和信赖域优化等技术。

F.2 策略网络与价值网络

'Model' 类定义了策略网络（用于动作选择）和价值网络（用于估计值函数）。这些网络是 Actor-Critic 架构的核心部分，策略（Actor）负责决策，而价值函数（Critic）则估计这些决策的价值。

- **策略网络**：用于确定在给定状态下采取何种动作。
- **价值网络**：用于评估当前策略的长期收益。

F.3 经验回放机制

代码中使用了 'buffer' 对象存储过去的经验（状态、动作、奖励、下一个状态），并通过经验回放机制（Experience Replay）从这些存储的经验中学习。此机制通过多次利用相同的经验来提高样本效率，这是 ACER 算法的关键之一。

- **代码中的实现**：'call' 方法中决定是从 'buffer' 中采样（离线学习）还是直接从环境中获取新的数据（在线学习）。

F.4 截断重要性采样与偏差校正

为了修正离线学习中的偏差，代码实现了截断重要性采样（Truncated Importance Sampling）和偏差校正（Bias Correction）。这些技术用于控制离线数据带来的高方差问题，同时确保估计的无偏性。

- **重要性采样**：用于调整不同策略下采样数据的权重，解决离线数据与当前策略不一致的问题。
- **截断与校正**：通过截断过大的重要性权重来减少方差，偏差校正项则保证了估计的无偏性。

代码中通过 `q_retrace` 和 `train` 方法实现了这一机制，在更新策略时应用了截断因子 `c`。

F.5 信赖域策略优化

信赖域优化是一种限制策略更新幅度的方法，防止策略更新过快导致的不稳定。代码中，通过 KL 散度控制新旧策略之间的变化，确保更新在一个“信赖”的范围内，从而保证策略的稳定性。

- **信赖域的实现**：在 'train' 方法中，通过计算 KL 散度并调整梯度更新来实现信赖域优化。

F.6 Retrace算法与稳定的价值更新

`q_retrace` 函数实现了 Retrace 算法，用于计算价值函数的目标值。相比传统的单步回报方法，Retrace 算法通过多步回报估计，显著降低了策略梯度估计的偏差，并加速了价值网络的学习。

- **Retrace算法**：提供了一个更稳定的价值函数估计，通过多个未来步骤的考量，调整重要性权重，改善了价值估计的准确性。
- **Retrace算法**：提供了一个更稳定的价值函数估计，通过多个未来步骤的考量，调整重要性权重，改善了价值估计的准确性。

F.7 代码与论文创新点的结合

通过上述分析，可以看出 'acer.py' 代码充分实现了论文中的各个创新点：

- **经验回放与离线学习的结合**：显著提高了样本利用效率。

- **截断重要性采样与偏差校正**：有效控制了学习过程中的高方差问题。
- **信赖域优化**：确保了策略更新的稳定性和安全性。
- **Retrace算法**：进一步增强了价值函数估计的稳定性，减少了策略更新的偏差。

F.8 代码文件

下面是实现上述分析内容的 ‘acer.py‘ 文件：

```python
import time
import functools
import numpy as np
import tensorflow as tf
from baselines import logger

from baselines.common import set_global_seeds
from baselines.common.policies import build_policy
from baselines.common.tf_util import get_session, save_variables, load_variables
from baselines.common.vec_env.vec_frame_stack import VecFrameStack

from baselines.a2c.utils import batch_to_seq, seq_to_batch
from baselines.a2c.utils import cat_entropy_softmax
from baselines.a2c.utils import Scheduler, find_trainable_variables
from baselines.a2c.utils import EpisodeStats
from baselines.a2c.utils import get_by_index, check_shape, avg_norm, gradient_add, q_explained_variance
from baselines.acer.buffer import Buffer
from baselines.acer.runner import Runner

# remove last step
def strip(var, nenvs, nsteps, flat = False):
    vars = batch_to_seq(var, nenvs, nsteps + 1, flat)
    return seq_to_batch(vars[:-1], flat)

def q_retrace(R, D, q_i, v, rho_i, nenvs, nsteps, gamma):
    """
    Calculates q_retrace targets

    :param R: Rewards
    :param D: Dones
    :param q_i: Q values for actions taken
    :param v: V values
    :param rho_i: Importance weight for each action
    :return: Q_retrace values
    """
    rho_bar = batch_to_seq(tf.minimum(1.0, rho_i), nenvs, nsteps, True)  # list of len steps, shape [nenvs]
    rs = batch_to_seq(R, nenvs, nsteps, True)  # list of len steps, shape [nenvs]
    ds = batch_to_seq(D, nenvs, nsteps, True)  # list of len steps, shape [nenvs]
    q_is = batch_to_seq(q_i, nenvs, nsteps, True)
    vs = batch_to_seq(v, nenvs, nsteps + 1, True)
    v_final = vs[-1]
    qret = v_final
    qrets = []
    for i in range(nsteps - 1, -1, -1):
        check_shape([qret, ds[i], rs[i], rho_bar[i], q_is[i], vs[i]], [[nenvs]] * 6)
        qret = rs[i] + gamma * qret * (1.0 - ds[i])
        qrets.append(qret)
        qret = (rho_bar[i] * (qret - q_is[i])) + vs[i]
    qrets = qrets[::-1]
    qret = seq_to_batch(qrets, flat=True)
    return qret

# For ACER with PPO clipping instead of trust region
# def clip(ratio, eps_clip):
#     # assume 0 <= eps_clip <= 1
#     return tf.minimum(1 + eps_clip, tf.maximum(1 - eps_clip, ratio))

class Model(object):
    def __init__(self, policy, ob_space, ac_space, nenvs, nsteps, ent_coef, q_coef, gamma, max_grad_norm, lr,
                 rprop_alpha, rprop_epsilon, total_timesteps, lrschedule,
                 c, trust_region, alpha, delta):

        sess = get_session()
        nact = ac_space.n
        nbatch = nenvs * nsteps

        A = tf.placeholder(tf.int32, [nbatch]) # actions
        D = tf.placeholder(tf.float32, [nbatch]) # dones
        R = tf.placeholder(tf.float32, [nbatch]) # rewards, not returns
        MU = tf.placeholder(tf.float32, [nbatch, nact]) # mu's
        LR = tf.placeholder(tf.float32, [])
        eps = 1e-6

        step_ob_placeholder = tf.placeholder(dtype=ob_space.dtype, shape=(nenvs,) + ob_space.shape)
        train_ob_placeholder = tf.placeholder(dtype=ob_space.dtype, shape=(nenvs*(nsteps+1),) + ob_space.shape)
        with tf.variable_scope('acer_model', reuse=tf.AUTO_REUSE):

            step_model = policy(nbatch=nenvs, nsteps=1, observ_placeholder=step_ob_placeholder, sess=sess)
            train_model = policy(nbatch=nbatch, nsteps=nsteps, observ_placeholder=train_ob_placeholder, sess=sess)


        params = find_trainable_variables("acer_model")
        print("Params {}".format(len(params)))
        for var in params:
            print(var)

        # create polyak averaged model
        ema = tf.train.ExponentialMovingAverage(alpha)
        ema_apply_op = ema.apply(params)

```

```
    def custom_getter(getter, *args, **kwargs):
        v = ema.average(getter(*args, **kwargs))
        print(v.name)
        return v

    with tf.variable_scope("acer_model", custom_getter=custom_getter, reuse=True):
        polyak_model = policy(nbatch=nbatch, nsteps=nsteps, observ_placeholder=train_ob_placeholder, sess=sess)

    # Notation: (var) = batch variable, (var)s = seqeuence variable, (var)_i = variable index by action at step i

    # action probability distributions according to train_model, polyak_model and step_model
    # poilcy.pi is probability distribution parameters; to obtain distribution that sums to 1 need to take softmax
    train_model_p = tf.nn.softmax(train_model.pi)
    polyak_model_p = tf.nn.softmax(polyak_model.pi)
    step_model_p = tf.nn.softmax(step_model.pi)
    v = tf.reduce_sum(train_model_p * train_model.q, axis = -1) # shape is [nenvs * (nsteps + 1)]

    # strip off last step
    f, f_pol, q = map(lambda var: strip(var, nenvs, nsteps), [train_model_p, polyak_model_p, train_model.q])
    # Get pi and q values for actions taken
    f_i = get_by_index(f, A)
    q_i = get_by_index(q, A)

    # Compute ratios for importance truncation
    rho = f / (MU + eps)
    rho_i = get_by_index(rho, A)

    # Calculate Q_retrace targets
    qret = q_retrace(R, D, q_i, v, rho_i, nenvs, nsteps, gamma)

    # Calculate losses
    # Entropy
    # entropy = tf.reduce_mean(strip(train_model.pd.entropy(), nenvs, nsteps))
    entropy = tf.reduce_mean(cat_entropy_softmax(f))

    # Policy Graident loss, with truncated importance sampling & bias correction
    v = strip(v, nenvs, nsteps, True)
    check_shape([qret, v, rho_i, f_i], [[nenvs * nsteps]] * 4)
    check_shape([rho, f, q], [[nenvs * nsteps, nact]] * 2)

    # Truncated importance sampling
    adv = qret - v
    logf = tf.log(f_i + eps)
    gain_f = logf * tf.stop_gradient(adv * tf.minimum(c, rho_i))  # [nenvs * nsteps]
    loss_f = -tf.reduce_mean(gain_f)

    # Bias correction for the truncation
    adv_bc = (q - tf.reshape(v, [nenvs * nsteps, 1])) # [nenvs * nsteps, nact]
    logf_bc = tf.log(f + eps) # / (f_old + eps)
    check_shape([adv_bc, logf_bc], [[nenvs * nsteps, nact]]*2)
    gain_bc = tf.reduce_sum(logf_bc * tf.stop_gradient(adv_bc * tf.nn.relu(1.0 - (c / (rho + eps))) * f), axis = 1) #IMP: This is sum, as expectation wrt f
    loss_bc= -tf.reduce_mean(gain_bc)

    loss_policy = loss_f + loss_bc

    # Value/Q function loss, and explained variance
    check_shape([qret, q_i], [[nenvs * nsteps]]*2)
    ev = q_explained_variance(tf.reshape(q_i, [nenvs, nsteps]), tf.reshape(qret, [nenvs, nsteps]))
    loss_q = tf.reduce_mean(tf.square(tf.stop_gradient(qret) - q_i)*0.5)

    # Net loss
    check_shape([loss_policy, loss_q, entropy], [[]] * 3)
    loss = loss_policy + q_coef * loss_q - ent_coef * entropy

    if trust_region:
        g = tf.gradients(- (loss_policy - ent_coef * entropy) * nsteps * nenvs, f) #[nenvs * nsteps, nact]
        # k = tf.gradients(KL(f_pol || f), f)
        k = - f_pol / (f + eps) #[nenvs * nsteps, nact] # Directly computed gradient of KL divergence wrt f
        k_dot_g = tf.reduce_sum(k * g, axis=-1)
        adj = tf.maximum(0.0, (tf.reduce_sum(k * g, axis=-1) - delta) / (tf.reduce_sum(tf.square(k), axis=-1) + eps)) #[nenvs * nsteps]

        # Calculate stats (before doing adjustment) for logging.
        avg_norm_k = avg_norm(k)
        avg_norm_g = avg_norm(g)
        avg_norm_k_dot_g = tf.reduce_mean(tf.abs(k_dot_g))
        avg_norm_adj = tf.reduce_mean(tf.abs(adj))

        g = g - tf.reshape(adj, [nenvs * nsteps, 1]) * k
        grads_f = -g/(nenvs*nsteps) # These are turst region adjusted gradients wrt f ie statistics of policy pi
        grads_policy = tf.gradients(f, params, grads_f)
        grads_q = tf.gradients(loss_q * q_coef, params)
        grads = [gradient_add(g1, g2, param) for (g1, g2, param) in zip(grads_policy, grads_q, params)]

        avg_norm_grads_f = avg_norm(grads_f) * (nsteps * nenvs)
        norm_grads_q = tf.global_norm(grads_q)
```

```
        norm_grads_policy = tf.global_norm(grads_policy)
    else:
        grads = tf.gradients(loss, params)

    if max_grad_norm is not None:
        grads, norm_grads = tf.clip_by_global_norm(grads, max_grad_norm)
    grads = list(zip(grads, params))
    trainer = tf.train.RMSPropOptimizer(learning_rate=LR, decay=rprop_alpha, epsilon=rprop_epsilon)
    _opt_op = trainer.apply_gradients(grads)

    # so when you call _train, you first do the gradient step, then you apply ema
    with tf.control_dependencies([_opt_op]):
        _train = tf.group(ema_apply_op)

    lr = Scheduler(v=lr, nvalues=total_timesteps, schedule=lrschedule)

    # Ops/Summaries to run, and their names for logging
    run_ops = [_train, loss, loss_q, entropy, loss_policy, loss_f, loss_bc, ev, norm_grads]
    names_ops = ['loss', 'loss_q', 'entropy', 'loss_policy', 'loss_f', 'loss_bc', 'explained_variance',
                 'norm_grads']
    if trust_region:
        run_ops = run_ops + [norm_grads_q, norm_grads_policy, avg_norm_grads_f, avg_norm_k, avg_norm_g, avg_norm_k_dot_g,
                             avg_norm_adj]
        names_ops = names_ops + ['norm_grads_q', 'norm_grads_policy', 'avg_norm_grads_f', 'avg_norm_k', 'avg_norm_g',
                                 'avg_norm_k_dot_g', 'avg_norm_adj']

    def train(obs, actions, rewards, dones, mus, states, masks, steps):
        cur_lr = lr.value_steps(steps)
        td_map = {train_model.X: obs, polyak_model.X: obs, A: actions, R: rewards, D: dones, MU: mus, LR: cur_lr}
        if states is not None:
            td_map[train_model.S] = states
            td_map[train_model.M] = masks
            td_map[polyak_model.S] = states
            td_map[polyak_model.M] = masks

        return names_ops, sess.run(run_ops, td_map)[1:]  # strip off _train

    def _step(observation, **kwargs):
        return step_model._evaluate([step_model.action, step_model_p, step_model.state], observation, **kwargs)


    self.train = train
    self.save = functools.partial(save_variables, sess=sess)
    self.load = functools.partial(load_variables, sess=sess)
    self.train_model = train_model
    self.step_model = step_model
    self._step = _step
    self.step = self.step_model.step

    self.initial_state = step_model.initial_state
    tf.global_variables_initializer().run(session=sess)


class Acer():
    def __init__(self, runner, model, buffer, log_interval):
        self.runner = runner
        self.model = model
        self.buffer = buffer
        self.log_interval = log_interval
        self.tstart = None
        self.episode_stats = EpisodeStats(runner.nsteps, runner.nenv)
        self.steps = None

    def call(self, on_policy):
        runner, model, buffer, steps = self.runner, self.model, self.buffer, self.steps
        if on_policy:
            enc_obs, obs, actions, rewards, mus, dones, masks = runner.run()
            self.episode_stats.feed(rewards, dones)
            if buffer is not None:
                buffer.put(enc_obs, actions, rewards, mus, dones, masks)
        else:
            # get obs, actions, rewards, mus, dones from buffer.
            obs, actions, rewards, mus, dones, masks = buffer.get()


        # reshape stuff correctly
        obs = obs.reshape(runner.batch_ob_shape)
        actions = actions.reshape([runner.nbatch])
        rewards = rewards.reshape([runner.nbatch])
        mus = mus.reshape([runner.nbatch, runner.nact])
        dones = dones.reshape([runner.nbatch])
        masks = masks.reshape([runner.batch_ob_shape[0]])

        names_ops, values_ops = model.train(obs, actions, rewards, dones, mus, model.initial_state, masks, steps)

        if on_policy and (int(steps/runner.nbatch) % self.log_interval == 0):
            logger.record_tabular("total_timesteps", steps)
            logger.record_tabular("fps", int(steps/(time.time() - self.tstart)))
```

```
            # IMP: In EpisodicLife env, during training, we get done=True at each loss of life, not just at the terminal state.
            # Thus, this is mean until end of life, not end of episode.
            # For true episode rewards, see the monitor files in the log folder.
            logger.record_tabular("mean_episode_length", self.episode_stats.mean_length())
            logger.record_tabular("mean_episode_reward", self.episode_stats.mean_reward())
            for name, val in zip(names_ops, values_ops):
                logger.record_tabular(name, float(val))
            logger.dump_tabular()


def learn(network, env, seed=None, nsteps=20, total_timesteps=int(80e6), q_coef=0.5, ent_coef=0.01,
          max_grad_norm=10, lr=7e-4, lrschedule='linear', rprop_epsilon=1e-5, rprop_alpha=0.99, gamma=0.99,
          log_interval=100, buffer_size=50000, replay_ratio=4, replay_start=10000, c=10.0,
          trust_region=True, alpha=0.99, delta=1, load_path=None, **network_kwargs):

    '''
    Main entrypoint for ACER (Actor-Critic with Experience Replay) algorithm (https://arxiv.org/pdf/1611.01224.pdf)
    Train an agent with given network architecture on a given environment using ACER.

    Parameters:
    ----------

    network:            policy network architecture. Either string (mlp, lstm, lnlstm, cnn_lstm, cnn, cnn_small, conv_only - see baselines.common/models.py for full list)
                        specifying the standard network architecture, or a function that takes tensorflow tensor as input and returns
                        tuple (output_tensor, extra_feed) where output tensor is the last network layer output, extra_feed is None for feed-forward
                        neural nets, and extra_feed is a dictionary describing how to feed state into the network for recurrent neural nets.
                        See baselines.common/policies.py/lstm for more details on using recurrent nets in policies

    env:                environment. Needs to be vectorized for parallel environment simulation.
                        The environments produced by gym.make can be wrapped using baselines.common.vec_env.DummyVecEnv class.

    nsteps:             int, number of steps of the vectorized environment per update (i.e. batch size is nsteps * nenv where
                        nenv is number of environment copies simulated in parallel) (default: 20)

    nstack:             int, size of the frame stack, i.e. number of the frames passed to the step model. Frames are stacked along channel dimension
                        (last image dimension) (default: 4)

    total_timesteps:    int, number of timesteps (i.e. number of actions taken in the environment) (default: 80M)

    q_coef:             float, value function loss coefficient in the optimization objective (analog of vf_coef for other actor-critic methods)

    ent_coef:           float, policy entropy coefficient in the optimization objective (default: 0.01)

    max_grad_norm:      float, gradient norm clipping coefficient. If set to None, no clipping. (default: 10),

    lr:                 float, learning rate for RMSProp (current implementation has RMSProp hardcoded in) (default: 7e-4)

    lrschedule:         schedule of learning rate. Can be 'linear', 'constant', or a function [0..1] -> [0..1] that takes fraction of the training progress as input and
                        returns fraction of the learning rate (specified as lr) as output

    rprop_epsilon:      float, RMSProp epsilon (stabilizes square root computation in denominator of RMSProp update) (default: 1e-5)

    rprop_alpha:        float, RMSProp decay parameter (default: 0.99)

    gamma:              float, reward discounting factor (default: 0.99)

    log_interval:       int, number of updates between logging events (default: 100)

    buffer_size:        int, size of the replay buffer (default: 50k)

    replay_ratio:       int, now many (on average) batches of data to sample from the replay buffer take after batch from the environment (default: 4)

    replay_start:       int, the sampling from the replay buffer does not start until replay buffer has at least that many samples (default: 10k)

    c:                  float, importance weight clipping factor (default: 10)
```

```
    trust_region        bool, whether or not algorithms estimates the gradient KL divergence between the old and updated policy and uses it to determine step size  (default: True)

    delta:              float, max KL divergence between the old policy and updated policy (default: 1)

    alpha:              float, momentum factor in the Polyak (exponential moving average) averaging of the model parameters (default: 0.99)

    load_path:          str, path to load the model from (default: None)

    **network_kwargs:               keyword arguments to the policy / network builder. See baselines.common/policies.py/build_policy and arguments to a particular type of network
                                    For instance, 'mlp' network architecture has arguments num_hidden and num_layers.

    '''

    print("Running Acer Simple")
    print(locals())
    set_global_seeds(seed)
    if not isinstance(env, VecFrameStack):
        env = VecFrameStack(env, 1)

    policy = build_policy(env, network, estimate_q=True, **network_kwargs)
    nenvs = env.num_envs
    ob_space = env.observation_space
    ac_space = env.action_space

    nstack = env.nstack
    model = Model(policy=policy, ob_space=ob_space, ac_space=ac_space, nenvs=nenvs, nsteps=nsteps,
                  ent_coef=ent_coef, q_coef=q_coef, gamma=gamma,
                  max_grad_norm=max_grad_norm, lr=lr, rprop_alpha=rprop_alpha, rprop_epsilon=rprop_epsilon,
                  total_timesteps=total_timesteps, lrschedule=lrschedule, c=c,
                  trust_region=trust_region, alpha=alpha, delta=delta)

    if load_path is not None:
        model.load(load_path)

    runner = Runner(env=env, model=model, nsteps=nsteps)
    if replay_ratio > 0:
        buffer = Buffer(env=env, nsteps=nsteps, size=buffer_size)
    else:
        buffer = None
    nbatch = nenvs*nsteps
    acer = Acer(runner, model, buffer, log_interval)
    acer.tstart = time.time()

    for acer.steps in range(0, total_timesteps, nbatch): # nbatch samples, 1 on_policy call and multiple off-policy calls
        acer.call(on_policy=True)
        if replay_ratio > 0 and buffer.has_atleast(replay_start):
            n = np.random.poisson(replay_ratio)
            for _ in range(n):
                acer.call(on_policy=False)  # no simulation steps in this

    return model
```