1. Введение в глубокое обучение с подкреплением

1.1 Основные понятия: агент, среда, награда, состояние, действие

Глубокое обучение с подкреплением (Deep Reinforcement Learning, DRL) представляет собой область машинного обучения, которая объединяет методы обучения с подкреплением (Reinforcement Learning, RL) с возможностями глубоких нейронных сетей. Эта комбинация позволяет решать сложные задачи принятия решений в условиях неопределенности, где агент должен взаимодействовать с окружающей средой для достижения поставленных целей.

Ключевые компоненты обучения с подкреплением

Обучение с подкреплением основано на взаимодействии между несколькими ключевыми компонентами:

Агент (Agent) — это сущность, которая принимает решения и выполняет действия. Агент может быть представлен алгоритмом, программой или физическим устройством, способным воспринимать информацию из окружающей среды и воздействовать на неё. Цель агента — максимизировать накопленное вознаграждение путем выбора оптимальных действий в каждом состоянии.

Среда (Environment) — это внешний мир, с которым взаимодействует агент. Среда может быть реальной (например, физический робот в лаборатории) или виртуальной (компьютерная игра, симуляция). Среда реагирует на действия агента, изменяя своё состояние и предоставляя агенту новую информацию и вознаграждение.

Состояние (State) — это представление текущей ситуации в среде. Состояние содержит всю информацию, необходимую агенту для принятия решения о следующем действии. В зависимости от задачи, состояние может быть:

- Полностью наблюдаемым (агент имеет доступ ко всей информации о среде)
- Частично наблюдаемым (агент видит только часть информации)

В случае частично наблюдаемых сред вместо состояния часто используется термин **наблюдение (Observation)**, которое представляет доступную агенту информацию о текущем состоянии среды.

Действие (Action) — это операция, которую агент может выполнить, чтобы воздействовать на среду. Набор возможных действий может быть:

- Дискретным (конечное число вариантов, например, "вверх", "вниз", "влево", "вправо")
- Непрерывным (бесконечное число вариантов в определенном диапазоне, например, угол поворота руля от -180° до $+180^\circ$)

Награда (Reward) — это числовой сигнал, который среда предоставляет агенту после выполнения действия. Награда является ключевым механизмом обратной связи, который указывает агенту, насколько хорошим было выбранное действие. Цель агента — максимизировать суммарную награду, полученную с течением времени.

Политика (Policy) — это стратегия, которой следует агент при выборе действий. Формально политика представляет собой отображение из пространства состояний в пространство действий, то есть определяет, какое действие следует выбрать в каждом состоянии. Политика может быть:

- Детерминированной (в каждом состоянии выбирается одно конкретное действие)
- Стохастической (действие выбирается согласно некоторому распределению вероятностей)

Цикл взаимодействия в обучении с подкреплением

Процесс обучения с подкреплением можно представить как последовательность шагов:

1. Агент получает информацию о текущем состоянии среды s_1
2. На основе этого состояния агент выбирает действие a_1 согласно своей политике
3. Агент выполняет выбранное действие, воздействуя на среду
4. Среда переходит в новое состояние s_2
5. Среда предоставляет агенту награду r_1, которая оценивает качество выполненного действия
6. Агент получает информацию о новом состоянии s_2 и награде r_1
7. Процесс повторяется с шага 2, но уже для нового состояния s_2

Этот цикл продолжается до тех пор, пока не будет достигнуто терминальное состояние (если оно существует в данной задаче) или не будет исчерпано заданное количество шагов.

Эпизоды и траектории

В обучении с подкреплением часто используются понятия эпизода и траектории:

Эпизод (Episode) — это последовательность состояний, действий и наград от начального состояния до терминального. Например, в игре шахматы эпизодом будет одна партия от начальной расстановки фигур до мата или пата.

Траектория (Trajectory) — это последовательность состояний, действий и наград, полученных в результате взаимодействия агента со средой. Траектория может быть как эпизодической (имеющей конечную длину), так и непрерывной (бесконечной).

Типы задач обучения с подкреплением

В зависимости от структуры задачи, обучение с подкреплением можно разделить на несколько типов:

Эпизодические задачи — задачи, имеющие четко определенное начало и конец (терминальное состояние). Примеры: игры (шахматы, го), навигация робота к цели.

Непрерывные задачи — задачи без естественного завершения, где взаимодействие агента со средой продолжается неограниченно долго. Примеры: управление температурой в здании, торговля на финансовых рынках.

Задачи с дискретным временем — время представлено дискретными шагами, на каждом из которых агент выбирает действие.

Задачи с непрерывным временем — время течет непрерывно, и агент должен принимать решения в реальном времени.

Особенности глубокого обучения с подкреплением

Глубокое обучение с подкреплением отличается от классического обучения с подкреплением использованием глубоких нейронных сетей для аппроксимации ключевых функций:

1. **Аппроксимация функции ценности** — нейронная сеть используется для оценки ожидаемой будущей награды в каждом состоянии.
2. **Аппроксимация политики** — нейронная сеть напрямую параметризует политику агента, определяя вероятности выбора различных действий.
3. **Аппроксимация модели среды** — нейронная сеть может использоваться для предсказания следующего состояния и награды на основе текущего состояния и действия.

Использование глубоких нейронных сетей позволяет эффективно работать с высокоразмерными пространствами состояний и действий, что делает возможным применение методов обучения с подкреплением к сложным практическим задачам, таким как компьютерное зрение, обработка естественного языка, робототехника и многие другие.

1.2 Формализация задачи обучения с подкреплением (уравнение Беллмана)

Математическая формализация обучения с подкреплением

Для строгого математического описания задачи обучения с подкреплением используется формализм марковских процессов принятия решений (Markov Decision Process, MDP). MDP представляет собой математическую структуру, которая идеально подходит для моделирования процесса принятия решений в условиях, когда результаты частично случайны и частично зависят от действий лица, принимающего решения.

Марковский процесс принятия решений (MDP)

Формально, MDP определяется как кортеж (S, A, P, R, γ), где:

- S — конечное множество состояний
- A — конечное множество действий
- $P : S \times A \times S \to [0, 1]$ — функция вероятности перехода, где $P(s'|s, a)$ определяет вероятность перехода в состояние s' из состояния s при выполнении действия a
- $R : S \times A \times S \to \mathbb{R}$ — функция вознаграждения, где $R(s, a, s')$ определяет немедленное вознаграждение, полученное при переходе из состояния s в состояние s' в результате действия a
- $\gamma \in [0, 1]$ — коэффициент дисконтирования, который определяет относительную важность будущих вознаграждений по сравнению с немедленными

Марковское свойство

Ключевым свойством MDP является марковское свойство, которое утверждает, что будущее состояние зависит только от текущего состояния и действия, но не от предыдущей истории:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1}|s_t, a_t)$$

Это свойство значительно упрощает анализ и решение задач обучения с подкреплением, поскольку позволяет агенту принимать решения только на основе текущего состояния, не требуя хранения всей истории взаимодействия со средой.

Функции ценности и уравнение Беллмана

В обучении с подкреплением ключевую роль играют функции ценности, которые оценивают, насколько хорошо находиться в определенном состоянии или выполнять определенное действие в состоянии, следуя заданной политике.

Функция ценности состояния (State-Value Function)

Функция ценности состояния $V^\pi(s)$ определяет ожидаемую суммарную дисконтированную награду, которую агент может получить, начиная с состояния s и

следуя политике π:

$$V^{\pi}(s) = E_{\pi}[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s]$$

где:

- $E_{\pi}[\cdot]$ — математическое ожидание при следовании политике π
- γ — коэффициент дисконтирования
- R_{t+1} — награда, полученная на шаге $t+1$
- S_0 — начальное состояние

Функция ценности действия (Action-Value Function)

Функция ценности действия $Q^{\pi}(s, a)$ определяет ожидаемую суммарную дисконтированную награду, которую агент может получить, начиная с состояния s, выполняя действие a, а затем следуя политике π:

$$Q^{\pi}(s, a) = E_{\pi}[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a]$$

Уравнение Беллмана для функции ценности состояния

Уравнение Беллмана для $V^{\pi}(s)$ выражает рекурсивную связь между ценностью текущего состояния и ценностями возможных следующих состояний:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma V^{\pi}(s')]$$

Это уравнение можно интерпретировать следующим образом: ценность состояния s при следовании политике π равна ожидаемой немедленной награде плюс дисконтированная ожидаемая ценность следующего состояния.

Уравнение Беллмана для функции ценности действия

Аналогично, уравнение Беллмана для $Q^{\pi}(s, a)$ имеет вид:

$$Q^{\pi}(s, a) = \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s', a')]$$

Оптимальные функции ценности

Оптимальная функция ценности состояния $V^*(s)$ определяет максимальную ожидаемую суммарную дисконтированную награду, которую можно получить, начиная с состояния s и следуя оптимальной политике:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

Аналогично, оптимальная функция ценности действия $Q^*(s, a)$ определяется как:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a)$$

Уравнение Беллмана оптимальности

Уравнение Беллмана оптимальности для $V^*(s)$ имеет вид:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma V^*(s')]$$

Для $Q^*(s, a)$:

$$Q^*(s, a) = \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \gamma \max_{a' \in A} Q^*(s', a')]$$

Эти уравнения являются основой для многих алгоритмов обучения с подкреплением, таких как Value Iteration, Policy Iteration и Q-learning.

Оптимальная политика

Оптимальная политика π^* — это политика, которая максимизирует ожидаемую суммарную дисконтированную награду для всех состояний:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a)$$

Важное свойство оптимальной политики заключается в том, что она всегда детерминирована, то есть для каждого состояния существует единственное оптимальное действие (хотя может быть несколько действий с одинаковой оптимальной ценностью).

Практическое значение уравнения Беллмана

Уравнение Беллмана имеет фундаментальное значение для обучения с подкреплением по нескольким причинам:

1. **Теоретическая основа**: Оно обеспечивает математическую основу для понимания и анализа задач обучения с подкреплением.
2. **Алгоритмическая основа**: Многие алгоритмы обучения с подкреплением основаны на решении или аппроксимации уравнения Беллмана.
3. **Итеративное решение**: Уравнение Беллмана можно решать итеративно, что приводит к таким алгоритмам, как Value Iteration и Policy Iteration.
4. **Связь с динамическим программированием**: Уравнение Беллмана устанавливает связь между обучением с подкреплением и динамическим программированием, позволяя применять методы динамического программирования к задачам обучения с подкреплением.

Ограничения и расширения

Хотя формализм MDP и уравнение Беллмана предоставляют мощный инструмент для анализа и решения задач обучения с подкреплением, они имеют некоторые ограничения:

1. **Размерность пространства состояний**: Для больших или непрерывных пространств состояний точное решение уравнения Беллмана становится вычислительно неосуществимым.
2. **Неизвестная модель среды**: В большинстве практических задач функции P и R неизвестны, что требует использования методов обучения без модели (model-free).
3. **Частично наблюдаемые среды**: В некоторых задачах агент не имеет полного доступа к состоянию среды, что приводит к необходимости использования частично наблюдаемых марковских процессов принятия решений (POMDP).

Для преодоления этих ограничений в современном обучении с подкреплением используются различные подходы, включая аппроксимацию функций с помощью нейронных сетей, методы обучения без модели и методы, основанные на градиентной политике.

1.3 Краткий обзор алгоритмов RL (Q-learning, DQN, Policy Gradient)

В этом разделе мы рассмотрим основные алгоритмы обучения с подкреплением, которые составляют фундамент современного глубокого обучения с подкреплением. Понимание этих алгоритмов критически важно для дальнейшего изучения более сложных методов и их практического применения.

Q-learning: основы табличного метода

Q-learning — это один из фундаментальных алгоритмов обучения с подкреплением, относящийся к классу методов временной разности (temporal difference, TD). Он был предложен Кристофером Уоткинсом в 1989 году и является методом обучения без модели (model-free), то есть не требует знания вероятностей переходов и функции вознаграждения среды.
Основная идея Q-learning заключается в итеративном обновлении оценок Q-функции на основе полученного опыта взаимодействия со средой. Q-функция $Q(s, a)$ представляет ожидаемую суммарную дисконтированную награду при выполнении действия a в состоянии s и следовании оптимальной политике в дальнейшем.

Алгоритм Q-learning

1. Инициализация Q-таблицы $Q(s, a)$ произвольными значениями (обычно нулями)
2. Для каждого эпизода:
   a. Инициализация начального состояния s
   b. Для каждого шага эпизода:

i. Выбор действия a в состоянии s с использованием ε-жадной политики (или другой политики исследования)
ii. Выполнение действия a, наблюдение награды r и следующего состояния s'
iii. Обновление Q-значения по формуле:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$$

iv. $s \leftarrow s'$
v. Если s является терминальным состоянием, завершить эпизод

где:

- α — скорость обучения (learning rate), обычно в диапазоне $(0, 1]$
- γ — коэффициент дисконтирования, обычно в диапазоне $[0, 1)$
- $\max_{a'} Q(s', a')$ — максимальное Q-значение для следующего состояния s' по всем возможным действиям a'

Сходимость Q-learning

Теоретически доказано, что при определенных условиях (все пары состояние-действие посещаются бесконечное число раз, скорость обучения уменьшается со временем по определенному закону) Q-learning сходится к оптимальной Q-функции $Q^*(s,a)$ независимо от используемой политики исследования.

Ограничения табличного Q-learning

Несмотря на теоретическую элегантность, табличный Q-learning имеет серьезные ограничения:

- Не масштабируется на большие пространства состояний и действий (проблема "проклятия размерности")
- Требует хранения Q-значений для всех пар состояние-действие в таблице
- Не обобщает знания между похожими состояниями

Эти ограничения привели к разработке методов аппроксимации Q-функции, в частности, с использованием глубоких нейронных сетей, что привело к появлению алгоритма Deep Q-Network (DQN).

Deep Q-Network (DQN): глубокое Q-обучение

Deep Q-Network (DQN) — это алгоритм, предложенный исследователями из DeepMind в 2013-2015 годах, который успешно объединил глубокие нейронные сети с Q-learning. DQN стал прорывом в области обучения с подкреплением, продемонстрировав способность обучаться играть в различные игры Atari на уровне человека, используя только пиксели экрана в качестве входных данных.

Основные компоненты DQN

1. **Аппроксимация Q-функции нейронной сетью**: Вместо хранения Q-значений в таблице, DQN использует глубокую нейронную сеть для аппроксимации Q-функции: $Q(s, a; \theta)$, где θ — параметры сети.
2. **Experience Replay (воспроизведение опыта)**: DQN использует буфер воспроизведения опыта, который хранит переходы (s, a, r, s') из предыдущих взаимодействий со средой. Во время обучения, мини-пакеты переходов случайно выбираются из этого буфера, что делает процесс обучения более стабильным и эффективным.
3. **Target Network (целевая сеть)**: DQN использует отдельную "целевую" нейронную сеть с параметрами θ^-, которая является копией основной сети, но обновляется реже. Это помогает стабилизировать обучение, предотвращая "погоню за движущейся целью".

Алгоритм DQN

1. Инициализация основной сети $Q(s, a; \theta)$ с случайными весами θ
2. Инициализация целевой сети $Q(s, a; \theta^-)$ с весами $\theta^- = \theta$
3. Инициализация буфера воспроизведения опыта D
4. Для каждого эпизода:
   a. Инициализация начального состояния s
   b. Для каждого шага эпизода:
   i. Выбор действия a в состоянии s с использованием ε-жадной политики на основе $Q(s, a; \theta)$
   ii. Выполнение действия a, наблюдение награды r и следующего состояния s'
   iii. Сохранение перехода (s, a, r, s') в буфере D
   iv. Выборка случайного мини-пакета переходов (s_j, a_j, r_j, s'_j) из D
   v. Для каждого перехода в мини-пакете:
   - Если s'_j является терминальным состоянием, то $y_j = r_j$
   - Иначе $y_j = r_j + \gamma \cdot \max_{a'} Q(s'_j, a'; \theta^-)$
   vi. Обновление параметров θ путем минимизации функции потерь:

$$L(\theta) = \frac{1}{N} \cdot \sum_j (y_j - Q(s_j, a_j; \theta))^2$$

   vii. Каждые C шагов обновление целевой сети: $\theta^- \leftarrow \theta$
   viii. $s \leftarrow s'$
   ix. Если s является терминальным состоянием, завершить эпизод

Улучшения DQN

С момента появления оригинального DQN было предложено множество улучшений:

1. **Double DQN**: Решает проблему переоценки Q-значений, используя основную сеть для выбора действия и целевую сеть для оценки его значения.
2. **Dueling DQN**: Разделяет Q-функцию на функцию ценности состояния $V(s)$ и функцию преимущества действия $A(s, a)$, что позволяет более эффективно оценивать ценность состояний.
3. **Prioritized Experience Replay**: Выбирает переходы из буфера не равномерно случайно, а с вероятностью, пропорциональной их TD-ошибке, что ускоряет обучение.
4. **Noisy Networks**: Добавляет параметрический шум к весам сети, что способствует более эффективному исследованию.
5. **Distributional DQN**: Моделирует распределение возможных наград, а не только их ожидаемое значение.
6. **Rainbow**: Объединяет все вышеперечисленные улучшения в один алгоритм, достигая еще лучших результатов.

Policy Gradient: методы градиента политики

В отличие от методов, основанных на ценности (value-based), таких как Q-learning и DQN, методы градиента политики (Policy Gradient) напрямую оптимизируют параметры политики без промежуточного вычисления функции ценности. Эти методы особенно полезны для задач с непрерывными пространствами действий и стохастическими политиками.

Основная идея Policy Gradient

Основная идея методов градиента политики заключается в параметризации политики $\pi(a|s;\theta)$ с помощью параметров θ (например, весов нейронной сети) и обновлении этих параметров в направлении, которое увеличивает ожидаемую суммарную награду.

Теорема о градиенте политики

Ключевым результатом, лежащим в основе методов градиента политики, является теорема о градиенте политики, которая утверждает, что градиент ожидаемой суммарной награды по параметрам политики может быть выражен как:

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta \log \pi(a|s;\theta) \cdot Q^\pi(s,a)]$$

где:

- $J(\theta)$ — ожидаемая суммарная награда при следовании политике $\pi(a|s;\theta)$
- $\nabla_\theta \log \pi(a|s;\theta)$ — градиент логарифма вероятности выбора действия a в состоянии s
- $Q^\pi(s,a)$ — функция ценности действия для политики π

REINFORCE: базовый алгоритм градиента политики

REINFORCE — это простейший алгоритм градиента политики, который использует метод Монте-Карло для оценки $Q^\pi(s, a)$:

1. Инициализация параметров политики θ случайными значениями
2. Для каждого эпизода:
   a. Генерация траектории $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_T)$ следуя политике $\pi(a|s;\theta)$
   b. Для каждого шага t в траектории:
   i. Вычисление суммарной дисконтированной награды от шага t:

$$G_t = \sum_{k=t}^{T} \gamma^{(k-t)} \cdot r_{k+1}$$

   ii. Обновление параметров политики:

$$\theta \leftarrow \theta + \alpha \cdot \nabla_\theta \log \pi(a_t|s_t;\theta) \cdot G_t$$

Проблемы REINFORCE и их решения

REINFORCE имеет несколько проблем:

- Высокая дисперсия оценок градиента
- Низкая эффективность использования данных (каждая траектория используется только один раз)
- Медленная сходимость

Для решения этих проблем были предложены различные улучшения:

1. **Baseline**: Вычитание базового уровня $b(s)$ из оценки награды снижает дисперсию, не внося смещения в ожидаемый градиент:

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta \log \pi(a|s;\theta) \cdot (Q^\pi(s, a) - b(s))]$$

   Часто в качестве базового уровня используется функция ценности состояния $V^\pi(s)$.
2. **Actor-Critic**: Объединяет методы градиента политики (actor) с методами, основанными на ценности (critic). Critic оценивает функцию ценности, которая затем используется для обновления параметров политики actor'a.
3. **Advantage Actor-Critic (A2C/A3C)**: Использует функцию преимущества $A(s, a) = Q(s, a) - V(s)$ вместо $Q(s, a)$, что снижает дисперсию. A3C (Asynchronous Advantage Actor-Critic) дополнительно использует асинхронное обучение на нескольких параллельных средах.
4. **Trust Region Policy Optimization (TRPO)**: Ограничивает размер обновления политики, чтобы предотвратить слишком большие изменения, которые могут привести к коллапсу производительности.
5. **Proximal Policy Optimization (PPO)**: Упрощает TRPO, используя обрезанную функцию потерь, которая ограничивает изменение политики без необходимости вычисления сложных вторых производных.

Современные алгоритмы на основе градиента политики

На основе методов градиента политики были разработаны многие современные алгоритмы глубокого обучения с подкреплением:

1. **Soft Actor-Critic (SAC)**: Добавляет энтропийную регуляризацию к целевой функции, поощряя исследование и предотвращая преждевременную сходимость к субоптимальным политикам.
2. **Twin Delayed Deep Deterministic Policy Gradient (TD3)**: Улучшает алгоритм DDPG, используя двойные Q-сети для уменьшения переоценки и задержку обновления политики для стабилизации обучения.
3. **Distributed Distributional Deep Deterministic Policy Gradient (D4PG)**: Расширяет DDPG, моделируя распределение наград и используя распределенное обучение.

Сравнение подходов и выбор алгоритма

При выборе алгоритма обучения с подкреплением для конкретной задачи следует учитывать несколько факторов:

Пространство действий

- **Дискретное пространство действий**: Q-learning, DQN и их варианты обычно хорошо работают.
- **Непрерывное пространство действий**: Методы градиента политики (DDPG, TD3, SAC) более подходящие.

Стохастичность среды

- **Детерминированная среда**: Методы, основанные на ценности, могут быть более эффективными.
- **Стохастическая среда**: Методы градиента политики часто более устойчивы к шуму.

Размер пространства состояний

- **Небольшое пространство состояний**: Табличные методы (Q-learning) могут быть достаточными.
- **Большое или непрерывное пространство состояний**: Методы с функциональной аппроксимацией (DQN, Policy Gradient) необходимы.

Требования к исследованию

- **Высокая потребность в исследовании**: Методы с явным механизмом исследования (ε-жадная политика, энтропийная регуляризация) предпочтительны.

- **Низкая потребность в исследовании**: Методы, ориентированные на эксплуатацию, могут быть более эффективными.

Вычислительные ресурсы

- **Ограниченные ресурсы**: Более простые алгоритмы (Q-learning, REINFORCE) могут быть предпочтительны.
- **Достаточные ресурсы**: Более сложные алгоритмы (Rainbow, SAC, PPO) могут дать лучшие результаты.

2. Необходимый математический аппарат

2.1 Математический анализ

2.1.1 Пределы, производные, частные производные, градиенты

Математический анализ является фундаментальным инструментом для понимания и разработки алгоритмов глубокого обучения с подкреплением. В этом разделе мы рассмотрим ключевые концепции математического анализа, которые широко используются в теории и практике обучения с подкреплением.

Пределы функций

Определение предела функции

Предел функции $f(x)$ при x, стремящемся к точке a, обозначается как $\lim_{x \to a} f(x) = L$ и означает, что значения функции $f(x)$ могут быть сделаны сколь угодно близкими к числу L при достаточном приближении x к точке a.

Формально, для любого $\varepsilon > 0$ существует $\delta > 0$ такое, что если $0 < |x - a| < \delta$, то $|f(x) - L| < \varepsilon$.

Свойства пределов

1. **Единственность предела**: Если предел существует, то он единственен.
2. **Арифметические операции над пределами**:
   - $\lim_{x \to a} [f(x) + g(x)] = \lim_{x \to a} f(x) + \lim_{x \to a} g(x)$
   - $\lim_{x \to a} [f(x) - g(x)] = \lim_{x \to a} f(x) - \lim_{x \to a} g(x)$
   - $\lim_{x \to a} [f(x) \cdot g(x)] = \lim_{x \to a} f(x) \cdot \lim_{x \to a} g(x)$
   - $\lim_{x \to a} [f(x) / g(x)] = \lim_{x \to a} f(x) / \lim_{x \to a} g(x)$, если $\lim_{x \to a} g(x) \neq 0$
3. **Предел композиции функций**: Если $\lim_{x \to a} g(x) = b$ и функция f непрерывна в точке b, то $\lim_{x \to a} f(g(x)) = f(\lim_{x \to a} g(x)) = f(b)$.
   Примеры вычисления пределов

Пример 1: Найти $\lim_{x \to 2} \frac{x^2-4}{x-2}$

Решение:
При прямой подстановке $x = 2$ получаем неопределенность вида $\frac{0}{0}$. Преобразуем выражение:
$\frac{x^2-4}{x-2} = \frac{x^2-2^2}{x-2} = \frac{(x-2)(x+2)}{x-2} = x + 2$

Теперь можно вычислить предел:
$\lim_{x \to 2} \frac{x^2-4}{x-2} = \lim_{x \to 2}(x + 2) = 2 + 2 = 4$

Пример 2: Найти $\lim_{x \to 0} \frac{1-\cos(x)}{x^2}$

Решение:
При $x = 0$ имеем неопределенность вида $\frac{0}{0}$. Используем разложение косинуса в ряд Тейлора:
$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \dots$

Подставляем:
$\lim_{x \to 0} \frac{1-\cos(x)}{x^2} = \lim_{x \to 0} \frac{1-(1-\frac{x^2}{2}+\frac{x^4}{24}-...)}{x^2} = \lim_{x \to 0} \frac{\frac{x^2}{2}-\frac{x^4}{24}+\dots}{x^2} = \lim_{x \to 0}(\frac{1}{2} - \frac{x^2}{24} + \dots) = \frac{1}{2}$

Производные функций

Определение производной

Производная функции $f(x)$ в точке $x = a$, обозначаемая как $f'(a)$ или $\frac{df}{dx}\Big|_{x=a}$, определяется как:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

Производная характеризует скорость изменения функции в данной точке и геометрически представляет собой угловой коэффициент касательной к графику функции в этой точке.

Правила дифференцирования

1. **Производная константы**: Если $f(x) = C$, то $f'(x) = 0$
2. **Производная степенной функции**: Если $f(x) = x^n$, то $f'(x) = n \cdot x^{n-1}$
3. **Линейность производной**:
   - $(\alpha f(x) + \beta g(x))' = \alpha f'(x) + \beta g'(x)$
4. **Правило произведения**:
   - $(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$
5. **Правило частного**:
   - $\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{[g(x)]^2}$
6. **Правило цепи (сложной функции)**:
   - Если $y = f(g(x))$, то $\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$

Производные элементарных функций

1. $(\sin x)' = \cos x$
2. $(\cos x)' = -\sin x$
3. $(e^x)' = e^x$
4. $(\ln x)' = \frac{1}{x}$
5. $(a^x)' = a^x \cdot \ln a$
6. $(\log_a x)' = \frac{1}{x \cdot \ln a}$

Примеры вычисления производных

Пример 1: Найти производную функции $f(x) = x^3 - 2x^2 + 3x - 5$

Решение:
$f'(x) = 3x^2 - 4x + 3$

Пример 2: Найти производную функции $f(x) = \sin(x^2)$

Решение:
Используем правило цепи. Пусть $g(x) = x^2$, тогда $f(x) = \sin(g(x))$.
$f'(x) = \cos(g(x)) \cdot g'(x) = \cos(x^2) \cdot 2x = 2x \cdot \cos(x^2)$

Пример 3: Найти производную функции $f(x) = \frac{e^x}{1+x^2}$

Решение:
Используем правило частного:
$f'(x) = \frac{(e^x)' \cdot (1+x^2) - e^x \cdot (1+x^2)'}{(1+x^2)^2}$
$= \frac{e^x \cdot (1+x^2) - e^x \cdot 2x}{(1+x^2)^2}$
$= \frac{e^x \cdot (1+x^2-2x)}{(1+x^2)^2}$
$= \frac{e^x \cdot (1-2x+x^2)}{(1+x^2)^2}$
$= \frac{e^x \cdot (1-x)^2}{(1+x^2)^2}$

Частные производные

Определение частной производной

Для функции нескольких переменных $f(x_1, x_2, \ldots, x_n)$ частная производная по переменной x_i, обозначаемая как $\frac{\partial f}{\partial x_i}$, определяется как производная функции f по переменной x_i при фиксированных значениях всех остальных переменных.

Формально:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

Правила вычисления частных производных

Правила вычисления частных производных аналогичны правилам для обычных производных, но применяются к одной переменной при фиксированных значениях остальных переменных.

Примеры вычисления частных производных

Пример 1: Найти частные производные функции $f(x, y) = x^2 + 3xy - 2y^2$

Решение:
$\frac{\partial f}{\partial x} = 2x + 3y$
$\frac{\partial f}{\partial y} = 3x - 4y$

Пример 2: Найти частные производные функции $f(x, y, z) = e^{(x+y)} \cdot \sin(yz)$

Решение:

$$\frac{\partial f}{\partial x} = e^{x+y} \cdot \sin(yz)$$

$$\frac{\partial f}{\partial y} = e^{x+y} \cdot \sin(yz) + e^{x+y} \cdot \cos(yz) \cdot z$$

$$\frac{\partial f}{\partial z} = e^{x+y} \cdot \cos(yz) \cdot y$$

Градиент функции

Определение градиента

Градиент функции $f(x_1, x_2, \ldots, x_n)$, обозначаемый как ∇f или $\operatorname{grad} f$, представляет собой вектор, компонентами которого являются частные производные функции по соответствующим переменным:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

Свойства градиента

1. **Направление наибыстрейшего возрастания**: Градиент функции в точке указывает направление наибыстрейшего возрастания функции.
2. **Перпендикулярность к линиям уровня**: Градиент функции в точке перпендикулярен к линии уровня функции, проходящей через эту точку.
3. **Линейность**:

$$\nabla(\alpha f + \beta g) = \alpha \nabla f + \beta \nabla g$$

4. **Правило произведения**:

$$\nabla(f \cdot g) = f \cdot \nabla g + g \cdot \nabla f$$

5. **Правило цепи**: Если $F(x) = f(g(x))$, то

$$\nabla F(x) = f'(g(x)) \cdot \nabla g(x)$$

Примеры вычисления градиента

Пример 1: Найти градиент функции $f(x, y) = x^2 + y^2$

Решение:

$$\frac{\partial f}{\partial x} = 2x$$

$$\frac{\partial f}{\partial y} = 2y$$

$$\nabla f = (2x, 2y)$$

Пример 2: Найти градиент функции $f(x, y, z) = xy + yz + zx$

Решение:

$$\frac{\partial f}{\partial x} = y + z$$

$$\frac{\partial f}{\partial y} = x + z$$

$$\frac{\partial f}{\partial z} = y + x$$

$$\nabla f = (y + z, x + z, y + x)$$

Применение производных и градиентов в глубоком обучении с подкреплением

Градиентный спуск

Градиентный спуск — это итеративный алгоритм оптимизации, который используется для минимизации функции потерь в задачах машинного обучения. Основная идея заключается в обновлении параметров модели в направлении, противоположном градиенту функции потерь:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla J(\theta_t)$$

где:

- θ_t — параметры модели на итерации t
- α — скорость обучения (learning rate)
- $\nabla J(\theta_t)$ — градиент функции потерь J по параметрам θ на итерации t

Обратное распространение ошибки (Backpropagation)

Обратное распространение ошибки — это алгоритм, используемый для эффективного вычисления градиентов в нейронных сетях. Он основан на правиле цепи для вычисления частных производных функции потерь по параметрам сети.

Градиент политики

В методах градиента политики градиент ожидаемой суммарной награды по параметрам политики используется для обновления параметров:

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

где:

- $J(\theta)$ — ожидаемая суммарная награда
- $\pi(a|s;\theta)$ — параметризованная политика
- $Q^\pi(s,a)$ — функция ценности действия

2.1.2 Примеры и задачи на поиск экстремумов функций

Необходимые и достаточные условия экстремума

Необходимое условие экстремума функции одной переменной

Если функция $f(x)$ имеет экстремум в точке $x = a$, то $f'(a) = 0$ или $f'(a)$ не существует.

Достаточное условие экстремума функции одной переменной

1. Если $f'(a) = 0$ и $f''(a) > 0$, то в точке $x = a$ функция имеет локальный минимум.
2. Если $f'(a) = 0$ и $f''(a) < 0$, то в точке $x = a$ функция имеет локальный максимум.
3. Если $f'(a) = 0$ и $f''(a) = 0$, то требуется дополнительное исследование.

Необходимое условие экстремума функции нескольких переменных

Если функция $f(x_1, x_2, \ldots, x_n)$ имеет экстремум в точке $(a_1, a_2, \ldots, a_n)$, то все частные производные в этой точке равны нулю или не существуют:

$$\left.\frac{\partial f}{\partial x_i}\right|_{(a_1,a_2,\ldots,a_n)} = 0 \quad \text{для всех } i = 1, 2, \ldots, n$$

Достаточное условие экстремума функции двух переменных

Пусть функция $f(x, y)$ имеет непрерывные частные производные второго порядка в окрестности точки (a, b), и пусть $\left.\frac{\partial f}{\partial x}\right|_{(a,b)} = \left.\frac{\partial f}{\partial y}\right|_{(a,b)} = 0$. Обозначим:

$$A = \left.\frac{\partial^2 f}{\partial x^2}\right|_{(a,b)}$$

$$B = \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{(a,b)}$$

$$C = \left.\frac{\partial^2 f}{\partial y^2}\right|_{(a,b)}$$

$$D = AC - B^2$$

Тогда:

1. Если $D > 0$ и $A > 0$, то в точке (a, b) функция имеет локальный минимум.
2. Если $D > 0$ и $A < 0$, то в точке (a, b) функция имеет локальный максимум.
3. Если $D < 0$, то в точке (a, b) функция имеет седловую точку.
4. Если $D = 0$, то требуется дополнительное исследование.

Примеры решения задач на поиск экстремумов

Задача 1: Найти экстремумы функции $f(x) = x^3 - 3x^2 + 3x - 1$

Решение:

1. Находим производную: $f'(x) = 3x^2 - 6x + 3 = 3(x^2 - 2x + 1) = 3(x - 1)^2$
2. Приравниваем к нулю: $3(x - 1)^2 = 0$
3. Решаем уравнение: $(x - 1)^2 = 0 \implies x = 1$
4. Находим вторую производную: $f''(x) = 6x - 6 = 6(x - 1)$
5. Проверяем знак второй производной в точке $x = 1$: $f''(1) = 6(1 - 1) = 0$

Поскольку вторая производная равна нулю, необходимо дополнительное исследование. Заметим, что $f'(x) = 3(x - 1)^2 \geq 0$ для всех x, и $f'(x) = 0$ только при $x = 1$. Это означает, что функция возрастает при $x < 1$ и при $x > 1$, а в точке $x = 1$ имеет горизонтальную касательную. Следовательно, в точке $x = 1$ функция имеет точку перегиба, а не экстремум.

Задача 2: Найти экстремумы функции $f(x, y) = x^2 + y^2 - 2x - 4y + 5$

Решение:

1. Находим частные производные:

$$\frac{\partial f}{\partial x} = 2x - 2$$

$$\frac{\partial f}{\partial y} = 2y - 4$$

2. Приравниваем к нулю:

$$2x - 2 = 0 \implies x = 1$$
$$2y - 4 = 0 \implies y = 2$$

3. Находим вторые частные производные:

$$\frac{\partial^2 f}{\partial x^2} = 2$$

$$\frac{\partial^2 f}{\partial x \partial y} = 0$$

$$\frac{\partial^2 f}{\partial y^2} = 2$$

4. Вычисляем определитель:

$$D = \left(\frac{\partial^2 f}{\partial x^2}\right) \cdot \left(\frac{\partial^2 f}{\partial y^2}\right) - \left(\frac{\partial^2 f}{\partial x \partial y}\right)^2 = 2 \cdot 2 - 0^2 = 4 > 0$$

5. Поскольку $D > 0$ и $\frac{\partial^2 f}{\partial x^2} > 0$, в точке $(1, 2)$ функция имеет локальный минимум.
6. Вычисляем значение функции в точке минимума:

$$f(1, 2) = 1^2 + 2^2 - 2 \cdot 1 - 4 \cdot 2 + 5 = 1 + 4 - 2 - 8 + 5 = 0$$

Задача 3: Найти экстремумы функции $f(x, y) = xy - x^3 - y^3$

Решение:

1. Находим частные производные:

$$\frac{\partial f}{\partial x} = y - 3x^2$$

$$\frac{\partial f}{\partial y} = x - 3y^2$$

2. Приравниваем к нулю:

$$y - 3x^2 = 0 \implies y = 3x^2$$

$$x - 3y^2 = 0 \implies x = 3y^2$$

3. Подставляем первое уравнение во второе:

$$x = 3(3x^2)^2 = 3 \cdot 9x^4 = 27x^4$$

$$x - 27x^4 = 0$$

$$x(1 - 27x^3) = 0$$

   Отсюда $x = 0$ или $x = (1/27)^{1/3} = 1/3$
4. Находим соответствующие значения y:
   Если $x = 0$, то $y = 3x^2 = 0$
   Если $x = 1/3$, то $y = 3(1/3)^2 = 3 \cdot 1/9 = 1/3$
5. Получаем две критические точки: $(0, 0)$ и $(1/3, 1/3)$
6. Находим вторые частные производные:

$$\frac{\partial^2 f}{\partial x^2} = -6x$$

$$\frac{\partial^2 f}{\partial x \partial y} = 1$$

$$\frac{\partial^2 f}{\partial y^2} = -6y$$

7. Проверяем точку $(0, 0)$:

$$A = \left.\frac{\partial^2 f}{\partial x^2}\right|_{(0,0)} = -6 \cdot 0 = 0$$

$$B = \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{(0,0)} = 1$$

$$C = \left.\frac{\partial^2 f}{\partial y^2}\right|_{(0,0)} = -6 \cdot 0 = 0$$

$$D = AC - B^2 = 0 \cdot 0 - 1^2 = -1 < 0$$

   Поскольку $D < 0$, в точке $(0, 0)$ функция имеет седловую точку.

8. Проверяем точку $(1/3, 1/3)$:

$$A = \left.\frac{\partial^2 f}{\partial x^2}\right|_{(1/3,1/3)} = -6 \cdot (1/3) = -2$$

$$B = \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{(1/3,1/3)} = 1$$

$$C = \left.\frac{\partial^2 f}{\partial y^2}\right|_{(1/3,1/3)} = -6 \cdot (1/3) = -2$$

$$D = AC - B^2 = (-2) \cdot (-2) - 1^2 = 4 - 1 = 3 > 0$$

   Поскольку $D > 0$ и $A < 0$, в точке $(1/3, 1/3)$ функция имеет локальный максимум.

9. Вычисляем значение функции в точке максимума:

$$f(1/3, 1/3) = (1/3) \cdot (1/3) - (1/3)^3 - (1/3)^3 = 1/9 - 1/27 - 1/27 = 1/9 - 2/27 = 3/27 - 2/27$$

Применение экстремумов функций в глубоком обучении с подкреплением

Оптимизация функции потерь

В глубоком обучении с подкреплением оптимизация функции потерь является ключевой задачей. Градиентный спуск и его варианты используются для нахождения минимума функции потерь, что соответствует оптимальным параметрам модели.

Поиск оптимальной политики

В методах градиента политики целью является максимизация ожидаемой суммарной награды $J(\theta)$, что эквивалентно поиску максимума функции $J(\theta)$ по параметрам θ.

Исследование ландшафта функции потерь

Анализ экстремумов функции потерь помогает понять структуру пространства параметров и выбрать подходящие методы оптимизации. Наличие множества локальных минимумов, седловых точек и плато может существенно влиять на процесс обучения.

2.2 Линейная алгебра

2.2.1 Векторы и матрицы, операции над ними

Линейная алгебра является одним из фундаментальных математических инструментов в области глубокого обучения с подкреплением. Она предоставляет мощный аппарат для работы с многомерными данными, преобразования пространств и решения систем уравнений. В этом разделе мы рассмотрим основные понятия линейной алгебры и их применение в контексте глубокого обучения с подкреплением.

Векторы и векторные пространства

Определение вектора

Вектор — это упорядоченный набор чисел. В контексте линейной алгебры мы обычно рассматриваем векторы как элементы векторного пространства. Вектор размерности n можно представить как:

$$x = (x_1, x_2, \ldots, x_n)^T \quad \text{или} \quad x = [x_1, x_2, \ldots, x_n]^T$$

где x_i — компоненты вектора, а символ ^{T} обозначает операцию транспонирования.

Векторное пространство

Векторное пространство V над полем F (обычно полем действительных чисел $\mathbb{R}$) — это множество, элементы которого (векторы) удовлетворяют аксиомам сложения и умножения на скаляр. Основные аксиомы включают:

1. Замкнутость относительно сложения: $\forall x, y \in V : x + y \in V$
2. Коммутативность сложения: $\forall x, y \in V : x + y = y + x$
3. Ассоциативность сложения: $\forall x, y, z \in V : (x + y) + z = x + (y + z)$
4. Существование нулевого вектора: $\exists 0 \in V : \forall x \in V : x + 0 = x$
5. Существование противоположного вектора: $\forall x \in V : \exists (-x) \in V : x + (-x) = 0$
6. Замкнутость относительно умножения на скаляр: $\forall \alpha \in F, \forall x \in V : \alpha x \in V$
7. Дистрибутивность скалярного умножения относительно сложения векторов: $\forall \alpha \in F, \forall x, y \in V : \alpha(x + y) = \alpha x + \alpha y$

8. Дистрибутивность скалярного умножения относительно сложения скаляров: $\forall \alpha, \beta \in F, \forall x \in V : (\alpha + \beta)x = \alpha x + \beta x$
9. Ассоциативность скалярного умножения: $\forall \alpha, \beta \in F, \forall x \in V : \alpha(\beta x) = (\alpha\beta)x$
10. Единичный скаляр: $\forall x \in V : 1x = x$

Операции над векторами

1. **Сложение векторов**: Если $x = (x_1, x_2, \ldots, x_n)^T$ и $y = (y_1, y_2, \ldots, y_n)^T$, то

$$x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)^T$$

2. **Умножение вектора на скаляр**: Если $x = (x_1, x_2, \ldots, x_n)^T$ и α — скаляр, то

$$\alpha x = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n)^T$$

3. **Скалярное произведение векторов**: Если $x = (x_1, x_2, \ldots, x_n)^T$ и $y = (y_1, y_2, \ldots, y_n)^T$, то скалярное произведение

$$x \cdot y = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

4. **Норма вектора**: Норма вектора x, обозначаемая как $\|x\|$, представляет собой меру "длины" вектора. Наиболее распространенной является евклидова норма (L_2 -норма):

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} = \sqrt{x \cdot x}$$

   Другие распространенные нормы включают:
   - L_1-норма:

$$\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n|$$

   - L_∞-норма:

$$\|x\|_\infty = \max(|x_1|, |x_2|, \ldots, |x_n|)$$

5. **Расстояние между векторами**: Расстояние между векторами x и y определяется как норма их разности:

$$d(x, y) = \|x - y\|$$

6. **Угол между векторами**: Угол θ между ненулевыми векторами x и y определяется через их скалярное произведение:

$$\cos(\theta) = \frac{x \cdot y}{\|x\|_2 \cdot \|y\|_2}$$

7. **Ортогональность векторов**: Векторы x и y называются ортогональными (перпендикулярными), если их скалярное произведение равно нулю: $x \cdot y = 0$

Примеры операций над векторами

Пример 1: Даны векторы $x = (2, -1, 3)^T$ и $y = (4, 0, -2)^T$. Найти $x + y$, $2x$, $x \cdot y$ и $\|x\|_2$.

Решение:

- $x + y = (2 + 4, -1 + 0, 3 + (-2))^T = (6, -1, 1)^T$
- $2x = (2 \cdot 2, 2 \cdot (-1), 2 \cdot 3)^T = (4, -2, 6)^T$
- $x \cdot y = 2 \cdot 4 + (-1) \cdot 0 + 3 \cdot (-2) = 8 + 0 - 6 = 2$
- $\|x\|_2 = \sqrt{2^2 + (-1)^2 + 3^2} = \sqrt{4 + 1 + 9} = \sqrt{14} \approx 3.74$

Пример 2: Проверить, являются ли векторы $x = (1, 2, 3)^T$ и $y = (3, -3, 1)^T$ ортогональными.

Решение:

$$x \cdot y = 1 \cdot 3 + 2 \cdot (-3) + 3 \cdot 1 = 3 - 6 + 3 = 0$$

Поскольку скалярное произведение равно нулю, векторы x и y ортогональны.

Матрицы и матричные операции

Определение матрицы

Матрица — это прямоугольная таблица чисел, организованная в строки и столбцы. Матрица размера $m \times n$ (m строк и n столбцов) обозначается как:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

где a_{ij} — элемент матрицы A, находящийся в i-й строке и j-м столбце.

Специальные типы матриц

1. **Квадратная матрица**: Матрица, у которой число строк равно числу столбцов ($m = n$).
2. **Диагональная матрица**: Квадратная матрица, у которой все элементы вне главной диагонали равны нулю ($a_{ij} = 0$ при $i \neq j$).
3. **Единичная матрица**: Диагональная матрица, у которой все элементы на главной диагонали равны единице ($a_{ii} = 1$ для всех i). Обозначается как I или E.
4. **Нулевая матрица**: Матрица, все элементы которой равны нулю. Обозначается как 0.
5. **Симметричная матрица**: Квадратная матрица, которая равна своей транспонированной ($A = A^T$, или $a_{ij} = a_{ji}$ для всех i, j).
6. **Кососимметричная матрица**: Квадратная матрица, транспонированная которой равна ей самой с противоположным знаком ($A = -A^T$, или $a_{ij} = -a_{ji}$ для всех i, j).

7. **Верхнетреугольная матрица**: Квадратная матрица, у которой все элементы ниже главной диагонали равны нулю ($a_{ij} = 0$ при $i > j$).
8. **Нижнетреугольная матрица**: Квадратная матрица, у которой все элементы выше главной диагонали равны нулю ($a_{ij} = 0$ при $i < j$).

Операции над матрицами

1. **Сложение матриц**: Если A и B — матрицы одинакового размера $m \times n$, то их сумма $C = A + B$ — это матрица размера $m \times n$ с элементами $c_{ij} = a_{ij} + b_{ij}$.
2. **Умножение матрицы на скаляр**: Если A — матрица размера $m \times n$ и α — скаляр, то αA — это матрица размера $m \times n$ с элементами $(\alpha A)_{ij} = \alpha \cdot a_{ij}$.
3. **Умножение матриц**: Если A — матрица размера $m \times n$ и B — матрица размера $n \times p$, то их произведение $C = A \cdot B$ — это матрица размера $m \times p$ с элементами:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \cdots + a_{in} \cdot b_{nj}$$

4. **Транспонирование матрицы**: Транспонированная матрица A^T получается из матрицы A заменой строк на столбцы (и наоборот). Если A — матрица размера $m \times n$, то A^T — матрица размера $n \times m$ с элементами $(A^T)_{ij} = a_{ji}$.
5. **След матрицы**: След квадратной матрицы A размера $n \times n$, обозначаемый как $\mathrm{tr}(A)$, — это сумма элементов на главной диагонали:

$$\mathrm{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}$$

6. **Определитель матрицы**: Определитель квадратной матрицы A размера $n \times n$, обозначаемый как $\det(A)$ или $|A|$, — это скаляр, вычисляемый по специальным правилам. Для матрицы 2×2:

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc$$

   Для матриц большего размера определитель вычисляется через разложение по строке или столбцу.
7. **Обратная матрица**: Обратная матрица A^{-1} к невырожденной квадратной матрице A — это такая матрица, что $A \cdot A^{-1} = A^{-1} \cdot A = I$. Обратная матрица существует тогда и только тогда, когда $\det(A) \neq 0$.
8. **Ранг матрицы**: Ранг матрицы — это максимальное число линейно независимых строк (или столбцов) матрицы.

Примеры матричных операций

Пример 1: Даны матрицы $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ и $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$. Найти $A + B$, $2A$ и $A \cdot B$.

Решение:

- $$A + B = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$
- $$2A = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$
- $$A \cdot B = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Пример 2: Найти определитель и обратную матрицу для $A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$.

Решение:

- $$\det(A) = 2 \cdot 3 - 1 \cdot 1 = 6 - 1 = 5$$
- $$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} = \frac{1}{5} \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 3/5 & -1/5 \\ -1/5 & 2/5 \end{bmatrix}$$

Проверка:

$$A \cdot A^{-1} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 3/5 & -1/5 \\ -1/5 & 2/5 \end{bmatrix} = \begin{bmatrix} 2 \cdot (3/5) + 1 \cdot (-1/5) & 2 \cdot (-1/5) + 1 \cdot (2/5) \\ 1 \cdot (3/5) + 3 \cdot (-1/5) & 1 \cdot (-1/5) + 3 \cdot (2/5) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

Линейные преобразования и матричное представление

Линейные преобразования

Линейное преобразование (или линейный оператор) $T : V \to W$ из векторного пространства V в векторное пространство W — это отображение, удовлетворяющее следующим свойствам:

1. Аддитивность: $T(x + y) = T(x) + T(y)$ для всех $x, y \in V$
2. Однородность: $T(\alpha x) = \alpha T(x)$ для всех $x \in V$ и всех скаляров α

Матричное представление линейных преобразований

Любое линейное преобразование $T : \mathbb{R}^n \to \mathbb{R}^m$ может быть представлено в виде умножения на матрицу A размера $m \times n$:

$$T(x) = Ax$$

где x — вектор из $\mathbb{R}^n$, а Ax — результат умножения матрицы A на вектор x.

Примеры линейных преобразований

Пример 1: Рассмотрим линейное преобразование $T : \mathbb{R}^2 \to \mathbb{R}^2$, заданное формулами:

$$T(x, y) = (2x + y, x - y)$$

Найти матрицу этого преобразования и применить его к вектору $v = (3, 1)^T$.

Решение:
Обозначим стандартный базис в $\mathbb{R}^2$ как $e_1 = (1, 0)^T$ и $e_2 = (0, 1)^T$.

$$T(e_1) = T(1, 0) = (2 \cdot 1 + 0, 1 - 0) = (2, 1)^T$$

$$T(e_2) = T(0, 1) = (2 \cdot 0 + 1, 0 - 1) = (1, -1)^T$$

Матрица преобразования A имеет столбцами образы базисных векторов:

$$A = [T(e_1)\ T(e_2)] = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}$$

Применение преобразования к вектору v:

$$T(v) = A \cdot v = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 + 1 \cdot 1 \\ 1 \cdot 3 + (-1) \cdot 1 \end{bmatrix} = \begin{bmatrix} 6 + 1 \\ 3 - 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 2 \end{bmatrix}$$

2.2.2 Собственные значения и собственные векторы, спектральное разложение

Собственные значения и собственные векторы

Определение собственных значений и собственных векторов

Пусть A — квадратная матрица размера $n \times n$. Ненулевой вектор v называется собственным вектором матрицы A, соответствующим собственному значению λ, если выполняется равенство:

$$Av = \lambda v$$

Другими словами, при умножении матрицы A на собственный вектор v результат отличается от v только скалярным множителем λ.

Характеристическое уравнение

Для нахождения собственных значений матрицы A необходимо решить характеристическое уравнение:

$$\det(A - \lambda I) = 0$$

где I — единичная матрица того же размера, что и A.
Корни этого уравнения являются собственными значениями матрицы A.

Нахождение собственных векторов

После нахождения собственного значения λ соответствующий ему собственный вектор v можно найти, решив систему линейных уравнений:

$$(A - \lambda I)v = 0$$

Свойства собственных значений и собственных векторов

1. Если v — собственный вектор матрицы A, соответствующий собственному значению λ, то αv (где $\alpha \neq 0$) также является собственным вектором, соответствующим тому же собственному значению.
2. Собственные векторы, соответствующие различным собственным значениям, линейно независимы.
3. Сумма собственных значений матрицы равна её следу:

$$\sum_i \lambda_i = \mathrm{tr}(A)$$

4. Произведение собственных значений матрицы равно её определителю:

$$\prod_i \lambda_i = \det(A)$$

5. Собственные значения симметричной матрицы всегда действительны.
6. Собственные векторы симметричной матрицы, соответствующие различным собственным значениям, ортогональны.

Примеры нахождения собственных значений и собственных векторов

Пример 1: Найти собственные значения и собственные векторы матрицы $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$.

Решение:

1. Составляем характеристическое уравнение:

$$\det(A - \lambda I) = \det\left(\begin{bmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{bmatrix}\right) = (2-\lambda)(2-\lambda) - 1 \cdot 1 = (2-\lambda)^2 - 1 = 4 - 4\lambda + \lambda^2 - 1$$

2. Решаем квадратное уравнение:

$$\lambda^2 - 4\lambda + 3 = 0$$

$$D = 16 - 12 = 4$$

$$\lambda_1 = (4 + 2)/2 = 3$$

$$\lambda_2 = (4 - 2)/2 = 1$$

3. Находим собственный вектор для $\lambda_1 = 3$:

$$(A - 3I)v = 0$$

$$\begin{bmatrix} 2-3 & 1 \\ 1 & 2-3 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$-v_1 + v_2 = 0$$

$$v_1 = v_2$$

Таким образом, собственный вектор для $\lambda_1 = 3$ имеет вид $v_1 = \alpha \cdot [1; 1]$, где $\alpha \neq 0$. Можно выбрать, например, $v_1 = [1; 1]$.

4. Находим собственный вектор для $\lambda_2 = 1$:

$$(A - 1I)v = 0$$

$$\begin{bmatrix} 2-1 & 1 \\ 1 & 2-1 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$v_1 + v_2 = 0$$

$$v_1 = -v_2$$

Таким образом, собственный вектор для $\lambda_2 = 1$ имеет вид $v_2 = \beta \cdot [1; -1]$, где $\beta \neq 0$. Можно выбрать, например, $v_2 = [1; -1]$.

Спектральное разложение матрицы

Диагонализация матрицы

Если квадратная матрица A размера $n \times n$ имеет n линейно независимых собственных векторов, то она может быть диагонализирована, то есть представлена в виде:

$$A = PDP^{-1}$$

где:

- P — матрица, столбцами которой являются собственные векторы матрицы A
- D — диагональная матрица, на главной диагонали которой стоят собственные значения матрицы A
- P^{-1} — обратная матрица к P

Спектральное разложение симметричных матриц

Для симметричной матрицы A ($A = A^T$) спектральное разложение имеет особенно простой вид:

$$A = Q\Lambda Q^T$$

где:

- Q — ортогональная матрица ($QQ^T = Q^TQ = I$), столбцами которой являются ортонормированные собственные векторы матрицы A

- Λ — диагональная матрица, на главной диагонали которой стоят собственные значения матрицы A

Применение линейной алгебры в глубоком обучении с подкреплением

Представление состояний и действий

В глубоком обучении с подкреплением состояния и действия часто представляются в виде векторов. Например, в задаче управления роботом состояние может быть представлено вектором, содержащим координаты и скорости различных частей робота.

Нейронные сети как линейные преобразования

Каждый слой нейронной сети можно рассматривать как линейное преобразование (умножение на матрицу весов), за которым следует нелинейная активационная функция. Например, для полносвязного слоя с входным вектором x, матрицей весов W и вектором смещений b, выход слоя до применения активационной функции равен $Wx + b$.

Матрицы ковариации в стохастических политиках

В алгоритмах, использующих стохастические политики с непрерывными действиями (например, в некоторых вариантах Policy Gradient), распределение действий часто моделируется как многомерное нормальное распределение с ковариационной матрицей, которая определяет форму и ориентацию эллипсоида вероятностей в пространстве действий.

2.3 Теория вероятностей и статистика

2.3.1 Случайные величины, математическое ожидание, дисперсия

Теория вероятностей и статистика играют фундаментальную роль в глубоком обучении с подкреплением, поскольку многие аспекты взаимодействия агента со средой имеют стохастическую природу. В этом разделе мы рассмотрим основные понятия теории вероятностей и их применение в контексте обучения с подкреплением.

Вероятностное пространство и случайные события

Вероятностное пространство

Вероятностное пространство — это тройка (Ω, F, P), где:

- Ω — пространство элементарных исходов (множество всех возможных исходов эксперимента)
- F — σ-алгебра подмножеств Ω (множество событий)
- P — вероятностная мера, сопоставляющая каждому событию $A \in F$ число $P(A) \in [0, 1]$, удовлетворяющая аксиомам вероятности

Аксиомы вероятности (аксиомы Колмогорова)

1. Неотрицательность: $P(A) \geq 0$ для любого события $A \in F$
2. Нормировка: $P(\Omega) = 1$
3. Счетная аддитивность: Если $A_1, A_2, \ldots$ — попарно несовместные события ($A_i \cap A_j = \emptyset$ при $i \neq j$), то

$$P(A_1 \cup A_2 \cup \ldots) = P(A_1) + P(A_2) + \ldots$$

Условная вероятность и независимость событий

Условная вероятность события A при условии события B (при $P(B) > 0$) определяется как:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

События A и B называются независимыми, если:

$$P(A \cap B) = P(A) \cdot P(B)$$

или, эквивалентно, если $P(A|B) = P(A)$ (при $P(B) > 0$).

Случайные величины

Определение случайной величины

Случайная величина — это функция $X : \Omega \to \mathbb{R}$, сопоставляющая каждому элементарному исходу $\omega \in \Omega$ некоторое действительное число $X(\omega)$.

Формально, случайная величина X должна быть измеримой функцией, то есть для любого борелевского множества $B \subset \mathbb{R}$ прообраз $X^{-1}(B) = \{\omega \in \Omega : X(\omega) \in B\}$ должен быть событием (принадлежать F).

Дискретные и непрерывные случайные величины

Дискретная случайная величина принимает значения из конечного или счетного множества. Её распределение задается функцией вероятности:

$$p(x) = P(X = x)$$

где $p(x) \geq 0$ для всех x и $\sum_x p(x) = 1$.

Непрерывная случайная величина может принимать любое значение из некоторого интервала. Её распределение задается функцией плотности вероятности $f(x)$, такой что:

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

где $f(x) \geq 0$ для всех x и $\int_{-\infty}^{\infty} f(x)dx = 1$.

Функция распределения

Функция распределения случайной величины X определяется как:

$$F(x) = P(X \leq x)$$

Функция распределения обладает следующими свойствами:

1. $F(x)$ не убывает: если $x_1 < x_2$, то $F(x_1) \leq F(x_2)$
2. $\lim_{x \to -\infty} F(x) = 0$ и $\lim_{x \to \infty} F(x) = 1$
3. $F(x)$ непрерывна справа: $\lim_{h \to 0^+} F(x+h) = F(x)$

Для дискретной случайной величины:

$$F(x) = \sum_{t \leq x} p(t)$$

Для непрерывной случайной величины:

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

Числовые характеристики случайных величин

Математическое ожидание

Математическое ожидание (среднее значение) случайной величины X, обозначаемое как $\mathbb{E}[X]$ или μ_X, определяется как:

Для дискретной случайной величины:

$$\mathbb{E}[X] = \sum_x x \cdot p(x)$$

Для непрерывной случайной величины:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot f(x)dx$$

Свойства математического ожидания

1. Линейность: $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$ для любых констант a, b и случайных величин X, Y

2. Если X и Y независимы, то $\mathbb{E}[XY] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$
3. Если $X \geq 0$ почти наверное, то $\mathbb{E}[X] \geq 0$
4. Если c — константа, то $\mathbb{E}[c] = c$

Дисперсия и стандартное отклонение

Дисперсия случайной величины X, обозначаемая как $\mathrm{Var}(X)$ или σ_X^2, определяется как:

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

Стандартное отклонение σ_X определяется как квадратный корень из дисперсии:

$$\sigma_X = \sqrt{\mathrm{Var}(X)}$$

Свойства дисперсии

1. $\mathrm{Var}(X) \geq 0$
2. $\mathrm{Var}(aX + b) = a^2 \cdot \mathrm{Var}(X)$ для любых констант a, b
3. Если X и Y независимы, то $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$

Ковариация и корреляция

Ковариация случайных величин X и Y, обозначаемая как $\mathrm{Cov}(X, Y)$, определяется как:

$$\mathrm{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$$

Коэффициент корреляции Пирсона ρ определяется как:

$$\rho = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y}$$

где σ_X и σ_Y — стандартные отклонения X и Y соответственно.

Коэффициент корреляции принимает значения в диапазоне $[-1, 1]$:

- $\rho = 1$ соответствует полной положительной линейной зависимости
- $\rho = -1$ соответствует полной отрицательной линейной зависимости
- $\rho = 0$ указывает на отсутствие линейной зависимости

Моменты случайных величин

Момент k-го порядка случайной величины X определяется как:

$$\mu_k = \mathbb{E}[X^k]$$

Центральный момент k-го порядка определяется как:

$$\mu'_k = \mathbb{E}[(X - \mathbb{E}[X])^k]$$

Дисперсия — это центральный момент второго порядка: $\mathrm{Var}(X) = \mu'_2$.

Примеры вычисления характеристик случайных величин

Пример 1: Пусть X — дискретная случайная величина с распределением:
$P(X = 1) = 0.2$, $P(X = 2) = 0.5$, $P(X = 3) = 0.3$

Найти математическое ожидание и дисперсию X.

Решение:

$$\mathbb{E}[X] = 1 \cdot 0.2 + 2 \cdot 0.5 + 3 \cdot 0.3 = 0.2 + 1.0 + 0.9 = 2.1$$

$$\mathbb{E}[X^2] = 1^2 \cdot 0.2 + 2^2 \cdot 0.5 + 3^2 \cdot 0.3 = 0.2 + 2.0 + 2.7 = 4.9$$

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 = 4.9 - 2.1^2 = 4.9 - 4.41 = 0.49$$

Стандартное отклонение: $\sigma_X = \sqrt{0.49} = 0.7$

Пример 2: Пусть X — непрерывная случайная величина с плотностью вероятности:
$f(x) = 2x$ для $0 \leq x \leq 1$, и $f(x) = 0$ в противном случае.

Найти математическое ожидание и дисперсию X.

Решение:

$$\mathbb{E}[X] = \int_0^1 x \cdot 2x dx = 2 \cdot \int_0^1 x^2 dx = 2 \cdot \left[\frac{x^3}{3}\right]_0^1 = 2 \cdot \frac{1}{3} = \frac{2}{3} \approx 0.667$$

$$\mathbb{E}[X^2] = \int_0^1 x^2 \cdot 2x dx = 2 \cdot \int_0^1 x^3 dx = 2 \cdot \left[\frac{x^4}{4}\right]_0^1 = 2 \cdot \frac{1}{4} = \frac{1}{2} = 0.5$$

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 = 0.5 - \left(\frac{2}{3}\right)^2 = 0.5 - \frac{4}{9} = 0.5 - 0.444 = 0.056$$

Стандартное отклонение: $\sigma_X = \sqrt{0.056} \approx 0.236$

Многомерные случайные величины

Совместное распределение

Для двух случайных величин X и Y совместное распределение задается:

Для дискретных случайных величин — совместной функцией вероятности:

$$p(x, y) = P(X = x, Y = y)$$

Для непрерывных случайных величин — совместной плотностью вероятности $f(x, y)$, такой что:

$$P(a \leq X \leq b, c \leq Y \leq d) = \int_a^b \int_c^d f(x, y) dy dx$$

Условное распределение

Условное распределение случайной величины X при условии $Y = y$ задается:

Для дискретных случайных величин:

$$p(x|y) = P(X = x|Y = y) = \frac{p(x, y)}{p_Y(y)}, \quad \text{если } p_Y(y) > 0$$

Для непрерывных случайных величин:

$$f(x|y) = \frac{f(x, y)}{f_Y(y)}, \quad \text{если } f_Y(y) > 0$$

где $p_Y(y)$ и $f_Y(y)$ — маргинальные распределения Y.

Независимость случайных величин

Случайные величины X и Y называются независимыми, если:

Для дискретных случайных величин:

$$p(x, y) = p_X(x) \cdot p_Y(y) \quad \text{для всех } x, y$$

Для непрерывных случайных величин:

$$f(x, y) = f_X(x) \cdot f_Y(y) \quad \text{для всех } x, y$$

где $p_X(x), p_Y(y)$ и $f_X(x), f_Y(y)$ — маргинальные распределения X и Y соответственно.

Условное математическое ожидание

Условное математическое ожидание $\mathbb{E}[X|Y = y]$ определяется как:

Для дискретных случайных величин:

$$\mathbb{E}[X|Y = y] = \sum_x x \cdot p(x|y)$$

Для непрерывных случайных величин:

$$\mathbb{E}[X|Y = y] = \int_{-\infty}^{\infty} x \cdot f(x|y) dx$$

Условное математическое ожидание $\mathbb{E}[X|Y]$ — это случайная величина, являющаяся функцией от Y.

Свойства условного математического ожидания

1. $\mathbb{E}[\mathbb{E}[X|Y]] = \mathbb{E}[X]$ (закон полного математического ожидания)
2. Если X и Y независимы, то $\mathbb{E}[X|Y] = \mathbb{E}[X]$
3. $\mathbb{E}[g(Y)X|Y] = g(Y)\mathbb{E}[X|Y]$ для любой функции g

Применение в глубоком обучении с подкреплением

Стохастические политики

В обучении с подкреплением политика $\pi(a|s)$ часто является стохастической, то есть задает распределение вероятностей действий a для каждого состояния s. Математическое ожидание используется для вычисления ожидаемой награды:

$$\mathbb{E}[R|s, \pi] = \sum_a \pi(a|s) \cdot Q(s, a)$$

для дискретных действий или

$$\mathbb{E}[R|s, \pi] = \int_a \pi(a|s) \cdot Q(s, a) da$$

для непрерывных действий.

Исследование и эксплуатация

Баланс между исследованием и эксплуатацией в обучении с подкреплением часто моделируется с использованием стохастических политик, где дисперсия распределения действий контролирует степень исследования.

Оценка градиента политики

В методах градиента политики используется оценка градиента ожидаемой суммарной награды:

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

где математическое ожидание берется по траекториям, генерируемым политикой π.

2.3.2 Вероятностные распределения (нормальное, Бернулли, категориальное)

В этом разделе мы рассмотрим основные вероятностные распределения, которые широко используются в глубоком обучении с подкреплением.

Дискретные распределения

Распределение Бернулли

Распределение Бернулли описывает случайный эксперимент с двумя возможными исходами, обычно обозначаемыми как "успех" (1) и "неудача" (0).

Если $X \sim \text{Bernoulli}(p)$, то:

- $P(X = 1) = p$
- $P(X = 0) = 1 - p$

Характеристики:

- $\mathbb{E}[X] = p$
- $\mathrm{Var}(X) = p(1-p)$

Пример применения: В ϵ-жадной политике выбор между исследованием и эксплуатацией может моделироваться как случайная величина с распределением Бернулли, где $p = \epsilon$ — вероятность выбора случайного действия (исследование).

Биномиальное распределение

Биномиальное распределение описывает количество успехов в n независимых испытаниях Бернулли с вероятностью успеха p.

Если $X \sim \mathrm{Binomial}(n, p)$, то:

- $$P(X = k) = C(n, k) \cdot p^k \cdot (1-p)^{n-k} \quad \text{для } k = 0, 1, \ldots, n$$
  где $C(n, k) = \frac{n!}{k! \cdot (n-k)!}$ — биномиальный коэффициент.

Характеристики:

- $\mathbb{E}[X] = np$
- $\mathrm{Var}(X) = np(1-p)$

Пример применения: При моделировании среды с несколькими агентами, биномиальное распределение может использоваться для описания количества агентов, выбирающих определенное действие.

Категориальное распределение

Категориальное распределение является обобщением распределения Бернулли на случай более чем двух возможных исходов.

Если $X \sim \mathrm{Categorical}(p_1, p_2, \ldots, p_k)$, то:

- $P(X = i) = p_i$ для $i = 1, 2, \ldots, k$
  где $p_i \geq 0$ для всех i и $\sum_i p_i = 1$.

Пример применения: В задачах с дискретным пространством действий стохастическая политика часто моделируется как категориальное распределение, где p_i — вероятность выбора действия i.

Распределение Пуассона

Распределение Пуассона описывает количество событий, происходящих в фиксированном интервале времени или пространства, при условии, что события происходят с постоянной средней скоростью и независимо друг от друга.

Если $X \sim \mathrm{Poisson}(\lambda)$, то:

- $$P(X = k) = \frac{e^{-\lambda} \cdot \lambda^k}{k!} \quad \text{для } k = 0, 1, 2, \ldots$$

  где $\lambda > 0$ — параметр интенсивности.

Характеристики:

- $\mathbb{E}[X] = \lambda$
- $\mathrm{Var}(X) = \lambda$

Пример применения: В моделировании сред с редкими событиями, такими как появление специальных объектов или возникновение определенных ситуаций, распределение Пуассона может использоваться для генерации этих событий.

Непрерывные распределения

Равномерное распределение

Равномерное распределение на интервале $[a, b]$ имеет постоянную плотность вероятности на этом интервале.

Если $X \sim \mathrm{Uniform}(a, b)$, то:

- $$f(x) = \frac{1}{b - a} \quad \text{для } a \leq x \leq b$$
- $f(x) = 0$ в противном случае

Характеристики:

- $\mathbb{E}[X] = \frac{a+b}{2}$
- $\mathrm{Var}(X) = \frac{(b-a)^2}{12}$

Пример применения: При исследовании непрерывного пространства действий агент может выбирать случайные действия из равномерного распределения.

Нормальное (гауссово) распределение

Нормальное распределение — одно из наиболее важных распределений в теории вероятностей и статистике.

Если $X \sim \mathcal{N}(\mu, \sigma^2)$, то плотность вероятности:

- $$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad \text{для } -\infty < x < \infty$$

  где μ — математическое ожидание, σ^2 — дисперсия.

Характеристики:

- $\mathbb{E}[X] = \mu$
- $\mathrm{Var}(X) = \sigma^2$

Пример применения: В алгоритмах, использующих стохастические политики с непрерывными действиями (например, в некоторых вариантах Policy Gradient), распределение действий часто моделируется как нормальное распределение.

Многомерное нормальное распределение

Многомерное нормальное распределение является обобщением одномерного нормального распределения на случай нескольких переменных.

Если $X \sim \mathcal{N}(\mu, \Sigma)$, где X — n-мерный вектор, μ — n-мерный вектор средних значений, Σ — положительно определенная ковариационная матрица размера $n \times n$, то плотность вероятности:

- $$f(x) = \frac{1}{(2\pi)^{n/2} \cdot \det(\Sigma)^{1/2}} \cdot e^{-\frac{1}{2} \cdot (x-\mu)^T \cdot \Sigma^{-1} \cdot (x-\mu)}$$

Характеристики:

- $\mathbb{E}[X] = \mu$
- $\mathrm{Cov}(X) = \Sigma$

Пример применения: В алгоритмах, использующих стохастические политики с многомерными непрерывными действиями, распределение действий часто моделируется как многомерное нормальное распределение.

2.3.3 Методы статистической оценки (MLE, MAP)

Оценка максимального правдоподобия (MLE)

Оценка максимального правдоподобия (Maximum Likelihood Estimation, MLE) — это метод оценки параметров статистической модели. Идея заключается в выборе таких значений параметров, которые максимизируют вероятность (правдоподобие) наблюдаемых данных.

Пусть $X_1, X_2, \ldots, X_n$ — независимые и одинаково распределенные случайные величины с плотностью вероятности $f(x|\theta)$, зависящей от параметра θ. Функция правдоподобия определяется как:

$$L(\theta) = \prod_{i=1}^{n} f(X_i|\theta)$$

Оценка максимального правдоподобия $\hat{\theta}$ находится как:

$$\hat{\theta} = \arg\max_{\theta} L(\theta)$$

Часто удобнее максимизировать логарифм функции правдоподобия:

$$\log L(\theta) = \sum_{i=1}^{n} \log f(X_i|\theta)$$

Пример: Оценка параметров нормального распределения

Пусть $X_1, X_2, \ldots, X_n \sim \mathcal{N}(\mu, \sigma^2)$. Найти оценки максимального правдоподобия для μ и σ^2.

Решение:
Логарифмическая функция правдоподобия:

$$\log L(\mu, \sigma^2) = \sum_{i=1}^{n} \log f(X_i|\mu, \sigma^2) = \sum_{i=1}^{n} \log \left[\frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(X_i-\mu)^2}{2\sigma^2}} \right]$$

$$= -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(X_i - \mu)^2$$

Дифференцируя по μ и приравнивая к нулю:

$$\frac{\partial(\log L)}{\partial \mu} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(X_i - \mu) = 0$$

Отсюда

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Дифференцируя по σ^2 и приравнивая к нулю:

$$\frac{\partial(\log L)}{\partial(\sigma^2)} = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^{n}(X_i - \mu)^2 = 0$$

Отсюда

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \hat{\mu})^2$$

Таким образом, оценки максимального правдоподобия для параметров нормального распределения — это выборочное среднее и выборочная дисперсия.

Оценка максимума апостериорной вероятности (MAP)

Оценка максимума апостериорной вероятности (Maximum A Posteriori, MAP) — это метод оценки параметров, учитывающий априорную информацию о параметрах. В отличие от MLE, который максимизирует правдоподобие данных, MAP максимизирует апостериорную вероятность параметров.

По формуле Байеса:

$$P(\theta|X) \propto P(X|\theta) \cdot P(\theta)$$

где $P(\theta|X)$ — апостериорная вероятность, $P(X|\theta)$ — правдоподобие, $P(\theta)$ — априорная вероятность.

Оценка MAP находится как:

$$\hat{\theta} = \arg\max_{\theta} P(\theta|X) = \arg\max_{\theta} [P(X|\theta) \cdot P(\theta)]$$

Часто удобнее максимизировать логарифм:

$$\log P(\theta|X) = \log P(X|\theta) + \log P(\theta) + \text{const}$$

Пример: Оценка параметра распределения Бернулли с бета-априорным распределением

Пусть $X_1, X_2, \ldots, X_n \sim \text{Bernoulli}(p)$, и априорное распределение для p — бета-распределение с параметрами α и β: $p \sim \text{Beta}(\alpha, \beta)$.

Найти оценку MAP для p.

Решение:
Правдоподобие:

$$P(X|p) = \prod_{i=1}^{n} p^{X_i} \cdot (1-p)^{1-X_i} = p^k \cdot (1-p)^{n-k}$$

где $k = \sum_{i=1}^{n} X_i$ — число успехов.

Априорная плотность:

$$P(p) \propto p^{\alpha-1} \cdot (1-p)^{\beta-1}$$

Апостериорная плотность:

$$P(p|X) \propto P(X|p) \cdot P(p) \propto p^k \cdot (1-p)^{n-k} \cdot p^{\alpha-1} \cdot (1-p)^{\beta-1} = p^{k+\alpha-1} \cdot (1-p)^{n-k+\beta-1}$$

Логарифм апостериорной плотности:

$$\log P(p|X) = (k+\alpha-1)\log p + (n-k+\beta-1)\log(1-p) + \text{const}$$

Дифференцируя по p и приравнивая к нулю:

$$\frac{\partial(\log P(p|X))}{\partial p} = \frac{k+\alpha-1}{p} - \frac{n-k+\beta-1}{1-p} = 0$$

Отсюда:

$$(k+\alpha-1)(1-p) = (n-k+\beta-1)p$$

$$k+\alpha-1-p(k+\alpha-1) = p(n-k+\beta-1)$$

$$k + \alpha - 1 = p(k + \alpha - 1) + p(n - k + \beta - 1) = p(k + \alpha - 1 + n - k + \beta - 1) = p(n + \alpha + \beta - 2)$$

Таким образом,

$$\hat{p} = \frac{k + \alpha - 1}{n + \alpha + \beta - 2}$$

Для бета-априорного распределения с параметрами $\alpha = \beta = 1$ (равномерное априорное распределение) оценка MAP совпадает с оценкой MLE: $\hat{p} = k/n$.

Регуляризация как априорное распределение

В машинном обучении регуляризация часто интерпретируется как введение априорного распределения на параметры модели:

- L1 регуляризация (лассо) соответствует лапласовскому априорному распределению
- L2 регуляризация (гребневая регрессия) соответствует нормальному априорному распределению с нулевым средним

Применение в глубоком обучении с подкреплением

Оценка функции ценности

В методах временных различий (TD) оценка функции ценности может рассматриваться как задача статистической оценки, где целью является минимизация ошибки между предсказанной и фактической ценностью.

Оценка параметров политики

В методах градиента политики параметры политики оптимизируются для максимизации ожидаемой суммарной награды, что можно интерпретировать как задачу максимизации правдоподобия.

Байесовское обучение с подкреплением

В байесовском обучении с подкреплением неопределенность относительно параметров модели среды или функции ценности моделируется с использованием априорных и апостериорных распределений.

2.3.4 Задачи с решениями по теории вероятностей

Задача 1: Вероятность выбора оптимального действия

Агент использует ϵ-жадную политику с $\epsilon = 0.1$ для выбора действий в среде с 4 возможными действиями. В каждом состоянии одно из действий является оптимальным. Какова вероятность того, что агент выберет оптимальное действие?

Решение:
При использовании ϵ-жадной политики агент с вероятностью $1 - \epsilon$ выбирает действие с наибольшей оценкой Q (оптимальное действие), а с вероятностью ϵ выбирает случайное действие из всех возможных.

Вероятность выбора оптимального действия:

$$P(\text{оптимальное}) = (1 - \epsilon) + \epsilon \cdot \frac{1}{4} = 0.9 + 0.1 \cdot 0.25 = 0.9 + 0.025 = 0.925 \text{ или } 92.5\%$$

Задача 2: Ожидаемая награда при стохастической политике

Агент находится в состоянии s и использует стохастическую политику $\pi(a|s)$ для выбора действий. Вероятности выбора действий и соответствующие Q-значения приведены в таблице:

Действие	**$\pi(a**	**s)$**
a_1	0.4	5
a_2	0.3	8
a_3	0.2	3
a_4	0.1	10

Найти ожидаемую награду при следовании этой политике в состоянии s.

Решение:
Ожидаемая награда вычисляется как математическое ожидание Q-значений по распределению действий:

$$\mathbb{E}[Q(s,a)] = \sum_a \pi(a|s) \cdot Q(s,a) = 0.4 \cdot 5 + 0.3 \cdot 8 + 0.2 \cdot 3 + 0.1 \cdot 10 = 2 + 2.4 + 0.6 + 1 = 6$$

Задача 3: Оценка параметров нормального распределения

Агент собрал следующие данные о наградах, полученных при выполнении определенного действия в определенном состоянии: 3, 5, 2, 4, 6, 3, 5, 4. Предполагая, что награды распределены нормально, найти оценки максимального правдоподобия для параметров μ и σ^2.

Решение:
Оценка максимального правдоподобия для математического ожидания:

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i = \frac{3 + 5 + 2 + 4 + 6 + 3 + 5 + 4}{8} = \frac{32}{8} = 4$$

Оценка максимального правдоподобия для дисперсии:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 = \frac{(3-4)^2 + (5-4)^2 + (2-4)^2 + (4-4)^2 + (6-4)^2 + (3-4)^2 + (5-4)^2 +}{8}$$

$$= \frac{1+1+4+0+4+1+1+0}{8} = \frac{12}{8} = 1.5$$

Задача 4: Условная вероятность в марковском процессе

В марковском процессе вероятности переходов между состояниями s_1, s_2 и s_3 заданы матрицей переходов:

$$P = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{bmatrix}$$

где $P(i, j)$ — вероятность перехода из состояния s_i в состояние s_j.

Если агент начинает в состоянии s_1, какова вероятность того, что через два шага он окажется в состоянии s_3?

Решение:
Вероятность оказаться в состоянии s_3 через два шага, начиная с s_1, можно вычислить как:

$$P(s_3 \text{ после 2 шагов} | s_1 \text{ сейчас}) = \sum_i P(s_1 \to s_i) \cdot P(s_i \to s_3)$$

где $P(s_1 \to s_i)$ — вероятность перехода из s_1 в s_i за один шаг, а $P(s_i \to s_3)$ — вероятность перехода из s_i в s_3 за один шаг.

$$P(s_3 \text{ после 2 шагов} | s_1 \text{ сейчас}) = P(s_1 \to s_1) \cdot P(s_1 \to s_3) + P(s_1 \to s_2) \cdot P(s_2 \to s_3) + P(s_1 \to s_3) \cdot$$

$$= 0.7 \cdot 0.1 + 0.2 \cdot 0.3 + 0.1 \cdot 0.5$$

$$= 0.07 + 0.06 + 0.05$$

$$= 0.18 \text{ или } 18\%$$

Альтернативно, можно использовать матричное умножение. Вероятность перехода за два шага задается матрицей P^2:

$$P^2 = P \cdot P = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{bmatrix}$$

Вычисляя произведение матриц, получаем P^2. Вероятность перехода из s_1 в s_3 за два шага — это элемент $P^2(1, 3)$.

Задача 5: Оценка MAP с априорным распределением

Агент использует монету для принятия решений. После 10 подбрасываний монета выпала орлом 7 раз. Найти оценку максимального правдоподобия (MLE) и оценку

максимума апостериорной вероятности (MAP) для вероятности выпадения орла, если априорное распределение — бета-распределение с параметрами $\alpha = 2$ и $\beta = 2$.

Решение:
Оценка максимального правдоподобия:

$$\hat{p}_{\text{MLE}} = \frac{\text{число успехов}}{\text{число испытаний}} = \frac{7}{10} = 0.7$$

Оценка максимума апостериорной вероятности с бета-априорным распределением:

$$\hat{p}_{\text{MAP}} = \frac{\text{число успехов} + \alpha - 1}{\text{число испытаний} + \alpha + \beta - 2} = \frac{7 + 2 - 1}{10 + 2 + 2 - 2} = \frac{8}{12} = \frac{2}{3} \approx 0.667$$

2.4 Дифференциальные уравнения

2.4.1 Основные типы дифференциальных уравнений и методы их решения

Дифференциальные уравнения являются мощным математическим инструментом для моделирования динамических систем, в которых скорость изменения величины зависит от её текущего значения. В контексте глубокого обучения с подкреплением дифференциальные уравнения используются для моделирования динамики среды, описания процессов обучения и оптимизации, а также для анализа сходимости алгоритмов. В этом разделе мы рассмотрим основные типы дифференциальных уравнений и методы их решения.

Обыкновенные дифференциальные уравнения (ОДУ)

Определение и классификация ОДУ

Обыкновенное дифференциальное уравнение — это уравнение, связывающее неизвестную функцию одной переменной, её производные и саму независимую переменную.

Общий вид ОДУ n-го порядка:

$$F(x, y, y', y'', \ldots, y^{(n)}) = 0$$

где $y = y(x)$ — неизвестная функция, $y', y'', \ldots, y^{(n)}$ — её производные по x.

ОДУ классифицируются по следующим признакам:

1. **По порядку**: Порядок ОДУ — это порядок наивысшей производной, входящей в уравнение.
   - ОДУ первого порядка: $F(x, y, y') = 0$
   - ОДУ второго порядка: $F(x, y, y', y'') = 0$
   - и т.д.

2. **По линейности**:
   - Линейное ОДУ:

$$a_n(x)y^{(n)} + a_{n-1}(x)y^{(n-1)} + \cdots + a_1(x)y' + a_0(x)y = f(x)$$

   - Нелинейное ОДУ: все остальные
3. **По однородности** (для линейных ОДУ):
   - Однородное:

$$a_n(x)y^{(n)} + a_{n-1}(x)y^{(n-1)} + \cdots + a_1(x)y' + a_0(x)y = 0$$

   - Неоднородное:

$$a_n(x)y^{(n)} + a_{n-1}(x)y^{(n-1)} + \cdots + a_1(x)y' + a_0(x)y = f(x), \quad \text{где } f(x) \neq 0$$

ОДУ первого порядка

Уравнения с разделяющимися переменными

Уравнение вида:

$$g(y)y' = f(x) \quad \text{или} \quad g(y)\frac{dy}{dx} = f(x)$$

можно решить разделением переменных:

$$g(y)dy = f(x)dx$$

Интегрируя обе части:

$$\int g(y)dy = \int f(x)dx + C$$

где C — произвольная постоянная.

Пример: Решить уравнение $y' = ky$, где k — константа.

Решение:

1. Разделяем переменные: $\frac{dy}{y} = k \cdot dx$
2. Интегрируем обе части: $\int \frac{dy}{y} = \int k \cdot dx$
3. Получаем: $\ln|y| = kx + C$, где C — произвольная постоянная
4. Потенцируем: $y = \pm e^{kx+C} = \pm e^C \cdot e^{kx} = C_1 \cdot e^{kx}$, где $C_1 = \pm e^C$ — новая произвольная постоянная
5. Так как C_1 может быть любым числом, включая отрицательные, можно записать общее решение как $y = C_1 \cdot e^{kx}$

Линейные ОДУ первого порядка

Линейное ОДУ первого порядка имеет вид:

$$y' + p(x)y = q(x)$$

Для его решения используется метод интегрирующего множителя:

1. Находим интегрирующий множитель:
$$\mu(x) = e^{\int p(x)dx}$$
2. Умножаем обе части уравнения на $\mu(x)$:
$$\mu(x)y' + \mu(x)p(x)y = \mu(x)q(x)$$
3. Левая часть представляет собой производную произведения:
$$(\mu(x)y)' = \mu(x)q(x)$$
4. Интегрируем:
$$\mu(x)y = \int \mu(x)q(x)dx + C$$
5. Выражаем y:
$$y = \frac{1}{\mu(x)}\left(\int \mu(x)q(x)dx + C\right)$$

Пример: Решить уравнение $y' + 2xy = x$.

Решение:

1. Идентифицируем $p(x) = 2x$ и $q(x) = x$
2. Находим интегрирующий множитель: $\mu(x) = e^{\int 2xdx} = e^{x^2}$
3. Умножаем уравнение на $\mu(x)$: $e^{x^2}y' + 2xe^{x^2}y = xe^{x^2}$
4. Левая часть: $(e^{x^2}y)' = xe^{x^2}$
5. Интегрируем: $e^{x^2}y = \int xe^{x^2}dx + C = \frac{1}{2}e^{x^2} + C$
6. Выражаем y: $y = \frac{1}{2} + Ce^{-x^2}$

ОДУ второго порядка

Линейные однородные ОДУ второго порядка с постоянными коэффициентами

Уравнение вида:

$$ay'' + by' + cy = 0$$

где a, b, c — константы, $a \neq 0$.

Для решения составляем характеристическое уравнение:

$$ar^2 + br + c = 0$$

Корни характеристического уравнения определяют вид общего решения:

1. Если корни различны и действительны ($r_1 \neq r_2$): $y = C_1 e^{r_1 x} + C_2 e^{r_2 x}$
2. Если корни совпадают ($r_1 = r_2 = r$): $y = (C_1 + C_2 x) e^{rx}$
3. Если корни комплексно-сопряженные ($r_{1,2} = \alpha \pm i\beta$): $y = e^{\alpha x}(C_1 \cos(\beta x) + C_2 \sin(\beta x))$

Пример: Решить уравнение $y'' - 5y' + 6y = 0$.

Решение:

1. Составляем характеристическое уравнение: $r^2 - 5r + 6 = 0$
2. Находим корни: $r_1 = 2, r_2 = 3$
3. Общее решение: $y = C_1 e^{2x} + C_2 e^{3x}$

Линейные неоднородные ОДУ второго порядка

Уравнение вида:

$$ay'' + by' + cy = f(x)$$

Общее решение состоит из суммы общего решения соответствующего однородного уравнения y_h и частного решения неоднородного уравнения y_p:

$$y = y_h + y_p$$

Для нахождения y_p используются методы:

- Метод неопределенных коэффициентов (для $f(x)$ специального вида)
- Метод вариации произвольных постоянных (метод Лагранжа)

Пример: Решить уравнение $y'' - 4y = x^2$.

Решение:

1. Решаем соответствующее однородное уравнение $y'' - 4y = 0$:
   - Характеристическое уравнение: $r^2 - 4 = 0$
   - Корни: $r_1 = 2, r_2 = -2$
   - Общее решение однородного уравнения: $y_h = C_1 e^{2x} + C_2 e^{-2x}$
2. Ищем частное решение неоднородного уравнения методом неопределенных коэффициентов.
   Так как $f(x) = x^2$, предполагаем y_p в виде: $y_p = Ax^2 + Bx + C$

   Находим: $y_p' = 2Ax + B$, $y_p'' = 2A$

   Подставляем в исходное уравнение: $2A - 4(Ax^2 + Bx + C) = x^2$

   Приравниваем коэффициенты при одинаковых степенях x:

   - при x^2: $-4A = 1$, откуда $A = -1/4$

- при x: $-4B = 0$, откуда $B = 0$
- при x^0: $2A - 4C = 0$, откуда $C = 2A/4 = -1/8$

Таким образом, $y_p = -x^2/4 - 1/8$

3. Общее решение: $y = y_h + y_p = C_1 e^{2x} + C_2 e^{-2x} - x^2/4 - 1/8$

Системы обыкновенных дифференциальных уравнений

Определение и классификация систем ОДУ

Система ОДУ — это набор дифференциальных уравнений, связывающих несколько неизвестных функций и их производные.

Общий вид системы n ОДУ первого порядка:

$$y_1' = f_1(x, y_1, y_2, \ldots, y_n)$$
$$y_2' = f_2(x, y_1, y_2, \ldots, y_n)$$
$$\ldots$$
$$y_n' = f_n(x, y_1, y_2, \ldots, y_n)$$

Любое ОДУ n-го порядка можно преобразовать в систему n ОДУ первого порядка путем введения новых переменных.

Линейные системы ОДУ с постоянными коэффициентами

Система линейных ОДУ с постоянными коэффициентами имеет вид:

$$y_1' = a_{11}y_1 + a_{12}y_2 + \cdots + a_{1n}y_n + g_1(x)$$
$$y_2' = a_{21}y_1 + a_{22}y_2 + \cdots + a_{2n}y_n + g_2(x)$$
$$\ldots$$
$$y_n' = a_{n1}y_1 + a_{n2}y_2 + \cdots + a_{nn}y_n + g_n(x)$$

В матричной форме:

$$Y' = AY + G(x)$$

где $Y = [y_1, y_2, \ldots, y_n]^T$, A — матрица коэффициентов, $G(x) = [g_1(x), g_2(x), \ldots, g_n(x)]^T$.

Если $G(x) = 0$, система называется однородной.

Методы решения систем ОДУ

Метод исключения

Для системы двух уравнений:

$$y_1' = a_{11}y_1 + a_{12}y_2$$

$$y_2' = a_{21}y_1 + a_{22}y_2$$

Дифференцируем первое уравнение:

$$y_1'' = a_{11}y_1' + a_{12}y_2'$$

Подставляем выражение для y_2' из второго уравнения:

$$y_1'' = a_{11}y_1' + a_{12}(a_{21}y_1 + a_{22}y_2)$$

Выражаем y_2 из первого уравнения:

$$y_2 = \frac{y_1' - a_{11}y_1}{a_{12}}$$

Подставляем:

$$y_1'' = a_{11}y_1' + a_{12}a_{21}y_1 + a_{12}a_{22}\left(\frac{y_1' - a_{11}y_1}{a_{12}}\right)$$

$$y_1'' = a_{11}y_1' + a_{12}a_{21}y_1 + a_{22}y_1' - a_{22}a_{11}y_1$$

$$y_1'' = (a_{11} + a_{22})y_1' + (a_{12}a_{21} - a_{22}a_{11})y_1$$

Получаем ОДУ второго порядка для y_1, которое можно решить стандартными методами.

Метод собственных значений

Для однородной системы $Y' = AY$ общее решение имеет вид:

$$Y = c_1e^{\lambda_1 x}V_1 + c_2e^{\lambda_2 x}V_2 + \cdots + c_ne^{\lambda_n x}V_n$$

где λ_i — собственные значения матрицы A, V_i — соответствующие собственные векторы, c_i — произвольные константы.

Пример: Решить систему

$$y_1' = 3y_1 + 2y_2$$

$$y_2' = 2y_1 + 3y_2$$

Решение:

1. Записываем систему в матричной форме: $Y' = AY$, где $A = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$
2. Находим собственные значения матрицы A:

$$\det(A - \lambda I) = \det\left(\begin{bmatrix} 3-\lambda & 2 \\ 2 & 3-\lambda \end{bmatrix}\right) = (3-\lambda)^2 - 4 = \lambda^2 - 6\lambda + 5 = 0$$

$$\lambda_1 = 5, \lambda_2 = 1$$

3. Находим собственные векторы:
   Для $\lambda_1 = 5$: $(A - 5I)V_1 = 0$

$$\begin{bmatrix} -2 & 2 \\ 2 & -2 \end{bmatrix} V_1 = 0$$

Получаем $V_1 = [1; 1]$

Для $\lambda_2 = 1$: $(A - 1I)V_2 = 0$

$$\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} V_2 = 0$$

Получаем $V_2 = [1; -1]$

4. Общее решение:

$$Y = c_1 e^{5x} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + c_2 e^{x} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

или $y_1 = c_1 e^{5x} + c_2 e^{x}$, $y_2 = c_1 e^{5x} - c_2 e^{x}$

Дифференциальные уравнения в частных производных (ДУЧП)

Определение и классификация ДУЧП

Дифференциальное уравнение в частных производных — это уравнение, содержащее неизвестную функцию нескольких переменных и её частные производные.

Общий вид ДУЧП второго порядка для функции $u(x, y)$:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu = G$$

где A, B, C, D, E, F, G — функции от x и y.

ДУЧП классифицируются по типу:

- Эллиптические: $B^2 - 4AC < 0$ (например, уравнение Лапласа)
- Параболические: $B^2 - 4AC = 0$ (например, уравнение теплопроводности)
- Гиперболические: $B^2 - 4AC > 0$ (например, волновое уравнение)

Основные типы ДУЧП

Уравнение Лапласа

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Описывает стационарные процессы, такие как стационарное распределение температуры или электростатический потенциал.

Уравнение теплопроводности

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

Описывает распространение тепла в среде.

Волновое уравнение

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

Описывает распространение волн.

Методы решения ДУЧП

Метод разделения переменных

Для решения линейных ДУЧП с однородными граничными условиями часто используется метод разделения переменных (метод Фурье).

Предполагаем решение в виде произведения функций каждой переменной:

$$u(x, y, t) = X(x)Y(y)T(t)$$

Подставляем в уравнение и разделяем переменные, получая обыкновенные дифференциальные уравнения для каждой функции.

Метод конечных разностей

Для численного решения ДУЧП часто используется метод конечных разностей, в котором производные заменяются их конечно-разностными аппроксимациями.

Численные методы решения дифференциальных уравнений

Методы Эйлера

Явный метод Эйлера

Для ОДУ $y' = f(x, y)$ с начальным условием $y(x_0) = y_0$:

$$y_{n+1} = y_n + h \cdot f(x_n, y_n)$$

где h — шаг интегрирования, $x_{n+1} = x_n + h$.

Неявный метод Эйлера

$$y_{n+1} = y_n + h \cdot f(x_{n+1}, y_{n+1})$$

Требует решения нелинейного уравнения на каждом шаге.

Методы Рунге-Кутты

Метод Рунге-Кутты четвертого порядка (RK4)

Для ОДУ $y' = f(x, y)$:

$$k_1 = f(x_n, y_n)$$

$$k_2 = f(x_n + h/2, y_n + h \cdot k_1/2)$$

$$k_3 = f(x_n + h/2, y_n + h \cdot k_2/2)$$

$$k_4 = f(x_n + h, y_n + h \cdot k_3)$$

$$y_{n+1} = y_n + \frac{h}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4)$$

Многошаговые методы

Метод Адамса-Башфорта

Использует значения функции f в нескольких предыдущих точках для вычисления следующего значения y.

Например, двухшаговый метод Адамса-Башфорта:

$$y_{n+1} = y_n + \frac{h}{2} \cdot (3f(x_n, y_n) - f(x_{n-1}, y_{n-1}))$$

Применение дифференциальных уравнений в глубоком обучении с подкреплением

Моделирование динамики среды

Дифференциальные уравнения используются для моделирования физических систем, с которыми взаимодействует агент. Например:

- Уравнения движения в задачах управления роботами
- Уравнения динамики в задачах управления транспортными средствами
- Уравнения, описывающие экономические или биологические процессы

Анализ сходимости алгоритмов

Процесс обучения в алгоритмах глубокого обучения с подкреплением можно представить как динамическую систему, описываемую дифференциальными уравнениями. Анализ этих уравнений позволяет исследовать сходимость алгоритмов и их устойчивость.

Непрерывные версии алгоритмов обучения с подкреплением

Некоторые алгоритмы обучения с подкреплением имеют непрерывные версии, описываемые дифференциальными уравнениями:

- Непрерывный временной TD-learning
- Непрерывные версии алгоритмов градиента политики

2.4.2 Задачи на динамические системы для моделирования поведения агентов

В этом разделе мы рассмотрим примеры задач, связанных с динамическими системами, которые могут использоваться для моделирования поведения агентов в задачах обучения с подкреплением.

Задача 1: Маятник с управлением

Рассмотрим задачу управления маятником, где агент может прикладывать момент силы для стабилизации маятника в верхнем положении.

Динамика маятника описывается уравнением:

$$\theta'' + b \cdot \theta' + c \cdot \sin(\theta) = u$$

где θ — угол отклонения от вертикали, b — коэффициент трения, c — отношение g/l (g — ускорение свободного падения, l — длина маятника), u — управляющее воздействие (момент силы).

Требуется найти управление $u(t)$, которое переводит маятник из нижнего положения ($\theta = \pi$) в верхнее ($\theta = 0$) и стабилизирует его там.

Решение

Преобразуем уравнение второго порядка в систему уравнений первого порядка, введя переменные $x_1 = \theta$ и $x_2 = \theta'$:

$$x_1' = x_2$$

$$x_2' = -b \cdot x_2 - c \cdot \sin(x_1) + u$$

Для стабилизации маятника в верхнем положении можно использовать линейное управление вида:

$$u = k_1 \cdot x_1 + k_2 \cdot x_2$$

где k_1 и k_2 — коэффициенты усиления.

Подставляя это управление в систему:

$$x_1' = x_2$$

$$x_2' = -b \cdot x_2 - c \cdot \sin(x_1) + k_1 \cdot x_1 + k_2 \cdot x_2 = (k_2 - b) \cdot x_2 + k_1 \cdot x_1 - c \cdot \sin(x_1)$$

Для малых отклонений от верхнего положения можно линеаризовать систему, используя приближение $\sin(x_1) \approx x_1$:

$$x_1' = x_2$$

$$x_2' = (k_2 - b) \cdot x_2 + (k_1 - c) \cdot x_1$$

Для устойчивости линеаризованной системы необходимо, чтобы все собственные значения матрицы системы имели отрицательные действительные части. Матрица системы:

$$A = \begin{bmatrix} 0 & 1 \\ k_1 - c & k_2 - b \end{bmatrix}$$

Характеристическое уравнение:

$$\det(A - \lambda I) = \lambda^2 - (k_2 - b)\lambda - (k_1 - c) = 0$$

Для устойчивости необходимо, чтобы $k_2 < b$ и $k_1 < c$.

Однако для перевода маятника из нижнего положения в верхнее линейного управления недостаточно. Здесь требуется нелинейное управление или управление с переключением режимов.

Задача 2: Модель популяционной динамики с управлением

Рассмотрим модель Лотки-Вольтерры (хищник-жертва) с управлением:

$$x' = \alpha x - \beta xy + u_1$$

$$y' = -\gamma y + \delta xy + u_2$$

где x — численность жертв, y — численность хищников, $\alpha, \beta, \gamma, \delta$ — положительные константы, u_1 и u_2 — управляющие воздействия (например, регулирование численности популяций).

Требуется найти управление, которое стабилизирует систему в заданной точке равновесия (x^*, y^*).

Решение

Без управления ($u_1 = u_2 = 0$) система имеет точку равновесия $(x^*, y^*) = (\gamma/\delta, \alpha/\beta)$.

Для стабилизации системы в произвольной точке $(\bar{x}, \bar{y})$ можно использовать управление вида:

$$u_1 = -\alpha x + \beta xy + v_1$$

$$u_2 = \gamma y - \delta xy + v_2$$

где v_1 и v_2 — новые управляющие воздействия.

Подставляя это управление в исходную систему:

$$x' = v_1$$

$$y' = v_2$$

Теперь можно использовать линейное управление для стабилизации:

$$v_1 = -k_1(x - \bar{x})$$

$$v_2 = -k_2(y - \bar{y})$$

где k_1 и k_2 — положительные коэффициенты.

Окончательно:

$$x' = -k_1(x - \bar{x})$$

$$y' = -k_2(y - \bar{y})$$

Эта система имеет единственную точку равновесия $(\bar{x}, \bar{y})$, которая глобально асимптотически устойчива.

Задача 3: Оптимальное управление линейной системой

Рассмотрим линейную систему:

$$x' = Ax + Bu$$

где x — вектор состояния, u — вектор управления, A и B — матрицы соответствующих размерностей.

Требуется найти управление $u(t)$, минимизирующее функционал:

$$J = \int_0^T (x^T Q x + u^T R u) dt + x^T(T) S x(T)$$

где Q, R, S — положительно определенные матрицы, T — конечное время.

Решение

Эта задача известна как линейно-квадратичный регулятор (LQR). Оптимальное управление имеет вид:

$$u(t) = -R^{-1} B^T P(t) x(t)$$

где $P(t)$ — решение матричного дифференциального уравнения Риккати:

$$-P'(t) = A^T P(t) + P(t) A - P(t) B R^{-1} B^T P(t) + Q$$

$$P(T) = S$$

Для бесконечного горизонта ($T \to \infty$) и стационарной системы P становится постоянной матрицей, удовлетворяющей алгебраическому уравнению Риккати:

$$A^T P + P A - P B R^{-1} B^T P + Q = 0$$

Задача 4: Управление нелинейной системой с использованием линеаризации

Рассмотрим нелинейную систему:

$$x' = f(x, u)$$

где x — вектор состояния, u — вектор управления, f — нелинейная функция.

Требуется найти управление, стабилизирующее систему в точке равновесия x^*.

Решение

Линеаризуем систему в окрестности точки равновесия (x^*, u^*), где $f(x^*, u^*) = 0$:

$$\delta x' = A\delta x + B\delta u$$

где $\delta x = x - x^*$, $\delta u = u - u^*$, $A = \frac{\partial f}{\partial x}\Big|_{(x^*, u^*)}$, $B = \frac{\partial f}{\partial u}\Big|_{(x^*, u^*)}$.

Для стабилизации линеаризованной системы можно использовать линейное управление:

$$\delta u = -K\delta x$$

где K — матрица коэффициентов усиления, выбранная так, чтобы матрица $(A - BK)$ была устойчивой (все собственные значения имели отрицательные действительные части).

Окончательно:

$$u = u^* - K(x - x^*)$$

Это управление обеспечивает локальную асимптотическую устойчивость нелинейной системы в окрестности точки равновесия x^*.

Задача 5: Оптимальное время перехода

Рассмотрим систему:

$$x' = u$$

где $|u| \leq 1$ — ограниченное управление.

Требуется перевести систему из начального состояния $x(0) = x_0$ в конечное состояние $x(T) = 0$ за минимальное время T.

Решение

Эта задача известна как задача о быстродействии. Согласно принципу максимума Понтрягина, оптимальное управление имеет вид:

$$u(t) = -\text{sign}(p(t))$$

где $p(t)$ — сопряженная переменная, удовлетворяющая уравнению:

$$p'(t) = 0$$

Следовательно, $p(t) = \text{const} = p_0$.

Если $p_0 \neq 0$, то $u(t) = -\text{sign}(p_0) = \text{const} = \pm 1$.

Интегрируя уравнение $x' = u$ с постоянным управлением $u = \pm 1$:

$$x(t) = x_0 + u \cdot t$$

Для достижения $x(T) = 0$:

$$0 = x_0 + u \cdot T$$

$$T = -x_0/u$$

Минимальное время достигается при $u = -\text{sign}(x_0)$, то есть:

$$u(t) = -\text{sign}(x_0)$$

$$T = |x_0|$$

Таким образом, оптимальное управление — это постоянное управление максимальной интенсивности, направленное в сторону, противоположную начальному отклонению.

2.5 Выпуклая оптимизация

2.5.1 Формулировка задачи оптимизации, выпуклость функции, условия оптимальности

Оптимизация играет центральную роль в глубоком обучении с подкреплением, поскольку обучение агента сводится к поиску оптимальных параметров политики или функции ценности. В этом разделе мы рассмотрим основные понятия выпуклой оптимизации и их применение в контексте глубокого обучения с подкреплением.

Общая формулировка задачи оптимизации

Задача оптимизации в общем виде формулируется следующим образом:

$$\min_{x \in \mathcal{D}} f(x)$$

или

$$\max_{x \in \mathcal{D}} f(x)$$

где:

- x — вектор оптимизируемых переменных
- $\mathcal{D}$ — допустимое множество (область определения задачи)
- $f(x)$ — целевая функция (функция стоимости или функция полезности)

Задача минимизации может быть преобразована в задачу максимизации (и наоборот) путем замены целевой функции на противоположную: $\min f(x) = \max(-f(x))$.

Задача условной оптимизации

Задача условной оптимизации включает ограничения на допустимое множество:

$$\min_x f(x)$$

при условиях:

$$g_i(x) \leq 0, \quad i = 1, 2, \ldots, m$$

$$h_j(x) = 0, \quad j = 1, 2, \ldots, p$$

где $g_i(x)$ — функции ограничений-неравенств, $h_j(x)$ — функции ограничений-равенств.

Выпуклые множества и выпуклые функции

Выпуклое множество

Множество $\mathcal{C}$ называется выпуклым, если для любых двух точек $x, y \in \mathcal{C}$ и любого $\lambda \in [0, 1]$ точка $\lambda x + (1 - \lambda) y$ также принадлежит $\mathcal{C}$.

Другими словами, если взять любые две точки из выпуклого множества, то весь отрезок, соединяющий эти точки, также принадлежит этому множеству.

Примеры выпуклых множеств:

- Шар: $\{x \in \mathbb{R}^n : \|x - x_0\| \leq r\}$
- Полупространство: $\{x \in \mathbb{R}^n : a^T x \leq b\}$
- Многогранник: $\{x \in \mathbb{R}^n : Ax \leq b\}$
- Положительный ортант: $\{x \in \mathbb{R}^n : x_i \geq 0, i = 1, 2, \ldots, n\}$

Выпуклая функция

Функция $f : \mathcal{D} \to \mathbb{R}$ называется выпуклой на выпуклом множестве $\mathcal{D}$, если для любых $x, y \in \mathcal{D}$ и любого $\lambda \in [0, 1]$ выполняется неравенство:

$$f(\lambda x + (1 - \lambda) y) \leq \lambda f(x) + (1 - \lambda) f(y)$$

Если неравенство строгое для всех $x \neq y$ и $\lambda \in (0, 1)$, то функция называется строго выпуклой.

Функция f называется вогнутой, если $-f$ выпуклая.

Геометрически, выпуклость функции означает, что график функции лежит не выше любой касательной к нему.

Свойства выпуклых функций

1. Если f и g — выпуклые функции, то $f + g$ также выпуклая функция.
2. Если f — выпуклая функция и $\alpha \geq 0$, то αf также выпуклая функция.
3. Если $f_1, f_2, \ldots, f_m$ — выпуклые функции, то $\max\{f_1, f_2, \ldots, f_m\}$ также выпуклая функция.

4. Если f — выпуклая функция и A — матрица, то $g(x) = f(Ax + b)$ также выпуклая функция.

Примеры выпуклых функций

1. Линейная функция: $f(x) = a^T x + b$
2. Квадратичная функция с положительно полуопределенной матрицей: $f(x) = x^T P x + q^T x + r$, где $P \succeq 0$
3. Норма: $f(x) = \|x\|$
4. Экспонента: $f(x) = e^{ax}$, где $a \in \mathbb{R}$
5. Отрицательный логарифм: $f(x) = -\log(x)$ на $\mathcal{D} = (0, \infty)$
6. Энтропия: $f(p) = \sum_i p_i \log(p_i)$ на множестве вероятностных распределений

Условия оптимальности

Необходимые условия первого порядка

Для дифференцируемой функции $f : \mathbb{R}^n \to \mathbb{R}$ необходимым условием локального минимума в точке x^* является равенство нулю градиента:

$$\nabla f(x^*) = 0$$

Это условие называется условием первого порядка.

Достаточные условия второго порядка

Для дважды дифференцируемой функции $f : \mathbb{R}^n \to \mathbb{R}$ достаточным условием локального минимума в точке x^*, удовлетворяющей условию первого порядка, является положительная определенность матрицы Гессе:

$$\nabla^2 f(x^*) \succ 0$$

Если матрица Гессе положительно полуопределена ($\nabla^2 f(x^*) \succeq 0$), то x^* — точка локального минимума или седловая точка.

Условия оптимальности для выпуклых функций

Для выпуклой функции f на выпуклом множестве $\mathcal{D}$ точка x^* является глобальным минимумом тогда и только тогда, когда:

$$\nabla f(x^*)^T (y - x^*) \geq 0 \quad \forall y \in \mathcal{D}$$

Если $\mathcal{D} = \mathbb{R}^n$, то условие упрощается до $\nabla f(x^*) = 0$.

Для строго выпуклой функции глобальный минимум, если он существует, единственен.

Условия Каруша-Куна-Таккера (KKT)

Для задачи условной оптимизации:

$$\min_x f(x)$$

при условиях:

$$g_i(x) \leq 0, \quad i = 1, 2, \ldots, m$$

$$h_j(x) = 0, \quad j = 1, 2, \ldots, p$$

необходимыми условиями оптимальности являются условия KKT:

1. Стационарность:

$$\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i \nabla g_i(x^*) + \sum_{j=1}^{p} \mu_j \nabla h_j(x^*) = 0$$

2. Допустимость: $g_i(x^*) \leq 0$ для всех i и $h_j(x^*) = 0$ для всех j
3. Дополняющая нежесткость: $\lambda_i g_i(x^*) = 0$ для всех i
4. Неотрицательность множителей Лагранжа: $\lambda_i \geq 0$ для всех i

где λ_i и μ_j — множители Лагранжа.

Если задача выпуклая (т.е. f и g_i — выпуклые функции, а h_j — аффинные функции), то условия KKT являются также достаточными для глобального минимума.

Двойственность в оптимизации

Функция Лагранжа

Для задачи условной оптимизации функция Лагранжа определяется как:

$$L(x, \lambda, \mu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \mu_j h_j(x)$$

где $\lambda_i \geq 0$ — множители Лагранжа для ограничений-неравенств, μ_j — множители Лагранжа для ограничений-равенств.

Двойственная функция

Двойственная функция определяется как:

$$g(\lambda, \mu) = \inf_x L(x, \lambda, \mu)$$

Двойственная функция всегда вогнута, даже если исходная задача не выпукла.

Двойственная задача

Двойственная задача формулируется как:

$$\max_{\lambda \geq 0, \mu} g(\lambda, \mu)$$

Слабая двойственность

Для любого допустимого решения x исходной (прямой) задачи и любых $\lambda \geq 0$, μ выполняется неравенство:

$$g(\lambda, \mu) \leq f(x)$$

Это означает, что значение двойственной функции всегда дает нижнюю оценку для оптимального значения прямой задачи.

Сильная двойственность

Если выполняется условие сильной двойственности, то оптимальные значения прямой и двойственной задач совпадают:

$$\min_x f(x) = \max_{\lambda \geq 0, \mu} g(\lambda, \mu)$$

Условие сильной двойственности выполняется, например, если прямая задача выпукла и удовлетворяет условию Слейтера (существует строго допустимая точка).

2.5.2 Методы решения оптимизационных задач: градиентный спуск, стохастический градиентный спуск

В этом разделе мы рассмотрим основные методы решения оптимизационных задач, которые широко используются в глубоком обучении с подкреплением.

Градиентный спуск

Градиентный спуск — это итеративный метод оптимизации первого порядка для нахождения локального минимума дифференцируемой функции.

Алгоритм градиентного спуска

1. Выбрать начальную точку x_0
2. Для $t = 0, 1, 2, \ldots$ до сходимости:
   - Вычислить градиент $\nabla f(x_t)$
   - Обновить

     $$x_{t+1} = x_t - \alpha_t \nabla f(x_t)$$

     , где $\alpha_t > 0$ — размер шага (learning rate)

Выбор размера шага

Размер шага α_t может быть:

- Постоянным: $\alpha_t = \alpha$
- Убывающим: $\alpha_t = \alpha_0 / (1 + \beta t)$ или $\alpha_t = \alpha_0 / \sqrt{t}$
- Адаптивным: выбирается на каждой итерации с помощью линейного поиска

Сходимость градиентного спуска

Для выпуклой функции f с липшицевым градиентом (константа Липшица L) и постоянным размером шага $\alpha \leq 1/L$ градиентный спуск сходится со скоростью $O(1/t)$:

$$f(x_t) - f(x^*) \leq \frac{\|x_0 - x^*\|^2}{2\alpha t}$$

где x^* — точка глобального минимума.

Для сильно выпуклой функции (с константой сильной выпуклости $\mu > 0$) и постоянным размером шага $\alpha \leq 2/(\mu + L)$ градиентный спуск сходится линейно:

$$f(x_t) - f(x^*) \leq (1 - \alpha\mu)^t \cdot (f(x_0) - f(x^*))$$

Стохастический градиентный спуск (SGD)

Стохастический градиентный спуск — это вариант градиентного спуска, который использует оценку градиента вместо точного значения. Он особенно эффективен для задач с большими объемами данных.

Алгоритм стохастического градиентного спуска

1. Выбрать начальную точку x_0
2. Для $t = 0, 1, 2, \ldots$ до сходимости:
   - Выбрать случайный индекс i_t или мини-батч B_t
   - Вычислить стохастический градиент $g_t = \nabla f_{i_t}(x_t)$ или

     $$g_t = \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f_i(x_t)$$

   - Обновить

     $$x_{t+1} = x_t - \alpha_t g_t$$

где f_i — компонента целевой функции, соответствующая i-му примеру данных, а $f = \frac{1}{n} \sum_{i=1}^{n} f_i$.

Сходимость стохастического градиентного спуска

Для выпуклой функции f с ограниченной дисперсией стохастического градиента и убывающим размером шага $\alpha_t = \alpha_0/\sqrt{t}$ стохастический градиентный спуск сходится со скоростью $O(1/\sqrt{t})$:

$$\mathbb{E}[f(\bar{x}_t) - f(x^*)] \leq \frac{C}{\sqrt{t}}$$

где $\bar{x}_t = \frac{1}{t} \sum_{i=0}^{t-1} x_i$ — усреднение всех итераций, а C — константа, зависящая от начальных условий и свойств функции.

Модификации градиентного спуска

Градиентный спуск с моментом

Градиентный спуск с моментом учитывает предыдущие направления обновления, что помогает ускорить сходимость и преодолеть локальные минимумы:

1. Инициализировать $v_0 = 0$
2. Для каждой итерации:
   - Вычислить градиент $g_t = \nabla f(x_t)$
   - Обновить момент: $v_t = \beta v_{t-1} + g_t$
   - Обновить параметры: $x_{t+1} = x_t - \alpha v_t$

где $\beta \in [0, 1)$ — коэффициент момента.

RMSProp

RMSProp адаптирует размер шага для каждого параметра на основе истории градиентов:

1. Инициализировать $v_0 = 0$
2. Для каждой итерации:
   - Вычислить градиент $g_t = \nabla f(x_t)$
   - Обновить накопленный квадрат градиента: $v_t = \beta v_{t-1} + (1 - \beta) g_t^2$
   - Обновить параметры:

$$x_{t+1} = x_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} g_t$$

где $\beta \in [0, 1)$ — коэффициент затухания, $\epsilon > 0$ — малая константа для численной стабильности.

Adam

Adam (Adaptive Moment Estimation) объединяет идеи момента и RMSProp:

1. Инициализировать $m_0 = 0$, $v_0 = 0$, $t = 0$
2. Для каждой итерации до сходимости:
   - $t = t + 1$
   - Вычислить градиент $g_t = \nabla f(x_t)$
   - Обновить оценку первого момента: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
   - Обновить оценку второго момента: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
   - Скорректировать оценки моментов:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Обновить параметры:

$$x_{t+1} = x_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

где $\beta_1, \beta_2 \in [0, 1)$ — коэффициенты затухания для оценок моментов.

Adam обычно работает хорошо в практических задачах и является одним из наиболее популярных оптимизаторов в глубоком обучении.

Методы второго порядка

Метод Ньютона

Метод Ньютона использует информацию о второй производной (матрице Гессе) для более эффективного поиска минимума:

$$x_{t+1} = x_t - [\nabla^2 f(x_t)]^{-1} \nabla f(x_t)$$

Метод Ньютона обладает квадратичной скоростью сходимости вблизи минимума, но требует вычисления и обращения матрицы Гессе, что может быть вычислительно затратно для задач большой размерности.

Квазиньютоновские методы

Квазиньютоновские методы аппроксимируют матрицу Гессе или ее обратную, избегая явного вычисления и обращения:

$$x_{t+1} = x_t - \alpha_t B_t^{-1} \nabla f(x_t)$$

где B_t — аппроксимация матрицы Гессе.

Наиболее известный квазиньютоновский метод — BFGS (Broyden-Fletcher-Goldfarb-Shanno), который обновляет аппроксимацию обратной матрицы Гессе на каждой итерации.

L-BFGS

L-BFGS (Limited-memory BFGS) — это версия BFGS с ограниченной памятью, которая хранит только несколько последних векторов обновления вместо полной матрицы. Это делает метод применимым для задач большой размерности.

Методы условной оптимизации

Метод проекции градиента

Для задачи минимизации $f(x)$ на выпуклом множестве $\mathcal{C}$ метод проекции градиента выполняет итерации:

$$x_{t+1} = P_{\mathcal{C}}(x_t - \alpha_t \nabla f(x_t))$$

где $P_{\mathcal{C}}(y)$ — проекция точки y на множество $\mathcal{C}$:

$$P_{\mathcal{C}}(y) = \arg\min_{x \in \mathcal{C}} \|x - y\|_2$$

Метод множителей Лагранжа

Для задачи с ограничениями-равенствами:

$$\min_x f(x) \quad \text{при условии} \quad h(x) = 0$$

метод множителей Лагранжа решает систему уравнений:

$$\nabla f(x) + \lambda \nabla h(x) = 0$$

$$h(x) = 0$$

где λ — множитель Лагранжа.

Метод штрафных функций

Метод штрафных функций заменяет задачу условной оптимизации последовательностью задач безусловной оптимизации:

$$\min_x f(x) + \mu P(x)$$

где $P(x)$ — штрафная функция, которая штрафует за нарушение ограничений, а $\mu > 0$ — параметр штрафа.

Для ограничений-равенств $h(x) = 0$ часто используется квадратичная штрафная функция $P(x) = \|h(x)\|_2^2$.

Применение методов оптимизации в глубоком обучении с подкреплением

Оптимизация функции ценности

В методах, основанных на ценности (value-based methods), параметры функции ценности V_θ или Q_θ оптимизируются путем минимизации среднеквадратичной ошибки:

$$\min_\theta \mathbb{E}[(y - V_\theta(s))^2]$$

или

$$\min_\theta \mathbb{E}[(y - Q_\theta(s, a))^2]$$

где y — целевое значение, вычисленное с использованием уравнения Беллмана.

Для оптимизации обычно используется стохастический градиентный спуск или его модификации (Adam, RMSProp).

Оптимизация политики

В методах градиента политики (policy gradient methods) параметры политики π_θ оптимизируются путем максимизации ожидаемой суммарной награды:

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

где τ — траектория, $R(\tau)$ — суммарная награда за траекторию.

Градиент целевой функции:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot R(\tau)]$$

Для уменьшения дисперсии оценки градиента часто используется базовая функция (baseline):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot (R(\tau) - b(s))]$$

где $b(s)$ — базовая функция, обычно аппроксимация функции ценности $V(s)$.

Оптимизация в алгоритме TRPO

Trust Region Policy Optimization (TRPO) решает задачу оптимизации политики с ограничением на величину изменения политики:

$$\max_\theta \mathbb{E}_{s \sim \rho_{\pi_{\theta_{\text{old}}}}, a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A^{\pi_{\theta_{\text{old}}}}(s,a) \right]$$

при условии:

$$\mathbb{E}_{s \sim \rho_{\pi_{\theta_{\text{old}}}}}[D_{KL}(\pi_{\theta_{\text{old}}}(\cdot|s) || \pi_\theta(\cdot|s))] \leq \delta$$

где $A^{\pi_{\theta_{\text{old}}}}(s,a)$ — функция преимущества, D_{KL} — дивергенция Кульбака-Лейблера, δ — параметр, контролирующий размер области доверия.

TRPO использует метод сопряженных градиентов для аппроксимации решения этой задачи.

Оптимизация в алгоритме PPO

Proximal Policy Optimization (PPO) упрощает TRPO, заменяя ограничение на штрафную функцию или обрезание целевой функции:

$$\max_\theta \mathbb{E}_{s \sim \rho_{\pi_{\theta_{\text{old}}}}, a \sim \pi_{\theta_{\text{old}}}} [\min(r_t(\theta) A^{\pi_{\theta_{\text{old}}}}(s,a), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A^{\pi_{\theta_{\text{old}}}}(s,a))]$$

где $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$, ϵ — параметр обрезания.

PPO обычно оптимизируется с помощью стохастического градиентного спуска или Adam.

2.5.3 Примеры задач и их решения

В этом разделе мы рассмотрим примеры задач оптимизации и их решения, которые иллюстрируют применение методов выпуклой оптимизации.

Задача 1: Линейная регрессия

Рассмотрим задачу линейной регрессии:

$$\min_{\theta} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \theta^T x_i)^2$$

где θ — вектор параметров, (x_i, y_i) — пары признак-ответ.

Решение

Целевая функция:

$$f(\theta) = \frac{1}{2n} \|y - X\theta\|_2^2$$

где X — матрица признаков, y — вектор ответов.

Градиент:

$$\nabla f(\theta) = \frac{1}{n} X^T (X\theta - y)$$

Приравнивая градиент к нулю:

$$X^T X\theta = X^T y$$

Аналитическое решение (нормальные уравнения):

$$\theta^* = (X^T X)^{-1} X^T y$$

при условии, что матрица $X^T X$ невырожденная.

Если матрица $X^T X$ вырожденная или плохо обусловленная, можно использовать регуляризацию или градиентный спуск.

Задача 2: Регуляризованная линейная регрессия (Ridge)

Рассмотрим задачу Ridge-регрессии:

$$\min_{\theta} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \theta^T x_i)^2 + \frac{\lambda}{2} \|\theta\|_2^2$$

где $\lambda > 0$ — параметр регуляризации.

Решение

Целевая функция:

$$f(\theta) = \frac{1}{2n}\|y - X\theta\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2$$

Градиент:

$$\nabla f(\theta) = \frac{1}{n}X^T X\theta - \frac{1}{n}X^T y + \lambda\theta = \left(\frac{1}{n}X^T X + \lambda I\right)\theta - \frac{1}{n}X^T y$$

Приравнивая градиент к нулю:

$$\left(X^T X + n\lambda I\right)\theta = X^T y$$

Аналитическое решение:

$$\theta^* = \left(X^T X + n\lambda I\right)^{-1} X^T y$$

Матрица $(X^T X + n\lambda I)$ всегда невырожденная при $\lambda > 0$, что гарантирует единственность решения.

Задача 3: Логистическая регрессия

Рассмотрим задачу бинарной классификации с использованием логистической регрессии:

$$\min_\theta -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\sigma(\theta^T x_i)) + (1 - y_i)\log(1 - \sigma(\theta^T x_i))]$$

где $\sigma(z) = \frac{1}{1+e^{-z}}$ — сигмоидная функция, $y_i \in \{0, 1\}$ — метки классов.

Решение

Целевая функция (отрицательное логарифмическое правдоподобие):

$$f(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\sigma(\theta^T x_i)) + (1 - y_i)\log(1 - \sigma(\theta^T x_i))]$$

Градиент:

$$\nabla f(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i - \sigma(\theta^T x_i)]x_i = \frac{1}{n}X^T(\sigma(X\theta) - y)$$

где $\sigma(X\theta)$ — вектор предсказанных вероятностей.

Для логистической регрессии нет аналитического решения, поэтому используется градиентный спуск или его модификации:

$$\theta_{t+1} = \theta_t - \alpha\nabla f(\theta_t) = \theta_t - \alpha\frac{1}{n}X^T(\sigma(X\theta_t) - y)$$

Задача 4: Метод опорных векторов (SVM)

Рассмотрим задачу линейного SVM с мягким зазором:

$$\min_{\theta,b,\xi} \frac{1}{2}\|\theta\|_2^2 + C\sum_{i=1}^{n}\xi_i$$

при условиях:

$$y_i(\theta^T x_i + b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots, n$$

$$\xi_i \geq 0, \quad i = 1, 2, \ldots, n$$

где $C > 0$ — параметр регуляризации, ξ_i — переменные невязки.

Решение

Эту задачу можно решить с помощью двойственной формулировки:

$$\max_{\alpha} \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j$$

при условиях:

$$0 \leq \alpha_i \leq C, \quad i = 1, 2, \ldots, n$$

$$\sum_{i=1}^{n}\alpha_i y_i = 0$$

где α_i — двойственные переменные.

После решения двойственной задачи оптимальные значения исходных переменных:

$$\theta^* = \sum_{i=1}^{n}\alpha_i y_i x_i$$

$$b^* = y_j - \theta^{*T} x_j \quad \text{для любого } j \text{ такого, что } 0 < \alpha_j < C$$

Задача 5: Выпуклая оптимизация в глубоком обучении с подкреплением

Рассмотрим задачу оптимизации политики в алгоритме REINFORCE:

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

где τ — траектория, $R(\tau)$ — суммарная награда за траекторию.

Решение

Градиент целевой функции:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot R(\tau)]$$

Можно оценить этот градиент по выборке траекторий:

$$\nabla_\theta J(\theta) \approx \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \cdot R(\tau^i)$$

где m — количество траекторий, T — длина траектории.

Обновление параметров с помощью градиентного восхождения:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k)$$

Для уменьшения дисперсии оценки градиента можно использовать базовую функцию:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \cdot (R_t^i - b(s_t^i))$$

где R_t^i — суммарная награда с момента t, $b(s_t^i)$ — базовая функция, обычно аппроксимация функции ценности $V(s_t^i)$.

3. Применение математики в глубоком обучении с подкреплением

3.1 Примеры использования математических концепций в нейронных сетях

В этом разделе мы рассмотрим, как математические концепции, изученные в предыдущих разделах, применяются при построении и обучении нейронных сетей в контексте глубокого обучения с подкреплением.

3.1.1 Линейная алгебра в нейронных сетях

Матричное представление слоев нейронной сети

Основной строительный блок нейронных сетей — полносвязный (линейный) слой, который можно представить как линейное преобразование:

$$y = Wx + b$$

где:

- $x \in \mathbb{R}^n$ — входной вектор
- $W \in \mathbb{R}^{m \times n}$ — матрица весов
- $b \in \mathbb{R}^m$ — вектор смещений
- $y \in \mathbb{R}^m$ — выходной вектор

Для батча входных данных $X \in \mathbb{R}^{b \times n}$, где b — размер батча, преобразование записывается как:

$$Y = XW^T + b$$

где $Y \in \mathbb{R}^{b \times m}$.

Эффективность нейронных сетей во многом обусловлена возможностью выполнять эти операции как матричные умножения, которые хорошо оптимизированы в современных библиотеках и аппаратном обеспечении.

Свертки как линейные операции

Свёрточные слои, часто используемые в обработке изображений и других структурированных данных, также можно представить как линейные операции. Свертка фильтра K с входными данными X может быть записана как:

$$(X * K)(i, j) = \sum_m \sum_n X(i+m, j+n) \cdot K(m, n)$$

Эту операцию можно реализовать как матричное умножение, преобразовав входные данные и фильтры соответствующим образом (операция im2col).

Собственные значения и векторы в анализе нейронных сетей

Собственные значения и собственные векторы матрицы весов играют важную роль в анализе динамики обучения нейронных сетей:

1. **Инициализация весов**: Методы инициализации, такие как инициализация Ксавье или Хе, учитывают спектральные свойства матриц весов для поддержания дисперсии активаций между слоями.
2. **Анализ сходимости**: Скорость сходимости градиентного спуска зависит от спектра гессиана функции потерь. Отношение наибольшего собственного значения к наименьшему (число обусловленности) определяет, насколько "сложной" является оптимизация.
3. **Спектральная нормализация**: Техника регуляризации, которая ограничивает спектральную норму (наибольшее сингулярное значение) матрицы весов, что помогает стабилизировать обучение, особенно в генеративно-состязательных сетях (GAN).

3.1.2 Математический анализ в нейронных сетях

Градиенты и обратное распространение ошибки

Обучение нейронных сетей основано на методе градиентного спуска, который требует вычисления градиентов функции потерь по параметрам сети. Алгоритм обратного распространения ошибки (backpropagation) эффективно вычисляет эти градиенты, используя правило цепи из математического анализа.

Для функции потерь L и параметров сети θ, градиент $\nabla_\theta L$ вычисляется путем последовательного применения правила цепи:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial \theta}$$

где y — выход сети.

Для многослойной сети с параметрами θ_l в слое l:

$$\frac{\partial L}{\partial \theta_l} = \frac{\partial L}{\partial y_l} \cdot \frac{\partial y_l}{\partial \theta_l}$$

где y_l — выход слоя l.

Градиент по выходу слоя l вычисляется рекурсивно:

$$\frac{\partial L}{\partial y_l} = \frac{\partial L}{\partial y_{l+1}} \cdot \frac{\partial y_{l+1}}{\partial y_l}$$

Это позволяет эффективно вычислять градиенты для всех параметров сети, начиная с выходного слоя и двигаясь к входному.

Функции активации и их производные

Нелинейные функции активации необходимы для моделирования сложных зависимостей в данных. Производные этих функций играют ключевую роль в обратном распространении ошибки.

Некоторые распространенные функции активации и их производные:

1. **Сигмоида**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

   - Производная:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

2. **Гиперболический тангенс**:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

   - Производная:

$$\tanh'(x) = 1 - \tanh^2(x)$$

3. **ReLU (Rectified Linear Unit)**:

$$\mathrm{ReLU}(x) = \max(0, x)$$

   - Производная:

$$\mathrm{ReLU}'(x) = \begin{cases} 1, & \text{если } x > 0 \\ 0, & \text{если } x \leq 0 \end{cases}$$

4. **Leaky ReLU**:

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{если } x > 0 \\ \alpha x, & \text{если } x \leq 0 \end{cases}$$

   - Производная:

$$\text{LeakyReLU}'(x) = \begin{cases} 1, & \text{если } x > 0 \\ \alpha, & \text{если } x \leq 0 \end{cases}$$

5. **Softmax**:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

   - Производная:

$$\frac{\partial \text{softmax}(x_i)}{\partial x_j} = \begin{cases} \text{softmax}(x_i) \cdot (1 - \text{softmax}(x_i)), & \text{если } i = j \\ -\text{softmax}(x_i) \cdot \text{softmax}(x_j), & \text{если } i \neq j \end{cases}$$

Проблема исчезающего и взрывного градиента

При обучении глубоких нейронных сетей может возникать проблема исчезающего или взрывного градиента, когда градиенты становятся очень маленькими или очень большими при распространении через многие слои.

Математически это можно объяснить с помощью правила цепи. Для глубокой сети с L слоями градиент функции потерь по параметрам первого слоя:

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial y_L} \cdot \frac{\partial y_L}{\partial y_{L-1}} \cdot \ldots \cdot \frac{\partial y_2}{\partial y_1} \cdot \frac{\partial y_1}{\partial \theta_1}$$

Если производные $\frac{\partial y_{l+1}}{\partial y_l}$ имеют нормы меньше 1, их произведение будет стремиться к 0 при увеличении L (исчезающий градиент). Если нормы больше 1, произведение может стать очень большим (взрывной градиент).

Методы решения этой проблемы включают:

- Использование функций активации с хорошими свойствами производных (например, ReLU)
- Нормализацию входов слоев (batch normalization, layer normalization)
- Остаточные соединения (residual connections)
- Инициализацию весов с учетом спектральных свойств

3.1.3 Теория вероятностей и статистика в нейронных сетях

Вероятностная интерпретация выходов нейронной сети

Выходы нейронной сети часто интерпретируются как вероятности или параметры вероятностных распределений:

1. **Классификация**: Функция softmax преобразует выходы сети в вероятности принадлежности к классам:

$$P(y = k|x) = \text{softmax}(z_k) = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

   где z_k — выход сети для класса k.
2. **Регрессия**: Выходы сети могут интерпретироваться как параметры нормального распределения:

$$P(y|x) = \mathcal{N}(y|\mu(x), \sigma^2(x))$$

   где $\mu(x)$ и $\sigma^2(x)$ — выходы сети, представляющие среднее и дисперсию.
3. **Политика в RL**: В задачах с дискретными действиями выходы сети представляют вероятности выбора действий:

$$\pi(a|s) = P(a|s) = \text{softmax}(z_a)$$

   где z_a — выход сети для действия a.

Функции потерь и их статистическая интерпретация

Функции потерь в нейронных сетях часто имеют статистическую интерпретацию:

1. **Среднеквадратичная ошибка (MSE)**: Соответствует максимизации правдоподобия при предположении о нормальном распределении шума:

$$L_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

2. **Кросс-энтропия**: Соответствует максимизации правдоподобия для категориального распределения:

$$L_{\text{CE}} = -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik})$$

   где y_{ik} — индикатор принадлежности примера i к классу k, $\hat{y}_{ik}$ — предсказанная вероятность.
3. **Функция потерь в RL**: В методах градиента политики функция потерь основана на ожидаемой суммарной награде:

$$L_{\text{PG}} = -\mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$$

   где τ — траектория, $R(\tau)$ — суммарная награда.

Регуляризация и байесовская интерпретация

Методы регуляризации в нейронных сетях имеют байесовскую интерпретацию:

1. **L2-регуляризация (weight decay)**: Соответствует априорному нормальному распределению параметров:

   $$P(\theta) \propto \exp\left(-\frac{\lambda}{2}\|\theta\|_2^2\right)$$

   Функция потерь с L2-регуляризацией:

   $$L_{\text{reg}} = L + \frac{\lambda}{2}\|\theta\|_2^2$$

2. **L1-регуляризация**: Соответствует априорному распределению Лапласа:

   $$P(\theta) \propto \exp(-\lambda\|\theta\|_1)$$

   Функция потерь с L1-регуляризацией:

   $$L_{\text{reg}} = L + \lambda\|\theta\|_1$$

3. **Dropout**: Может рассматриваться как приближенный байесовский вывод, где выходы сети представляют апостериорное распределение.

3.1.4 Дифференциальные уравнения в нейронных сетях

Рекуррентные нейронные сети как дискретные динамические системы

Рекуррентные нейронные сети (RNN) можно рассматривать как дискретные динамические системы:

$$h_t = f(h_{t-1}, x_t; \theta)$$

где h_t — скрытое состояние в момент времени t, x_t — вход, θ — параметры.

Для простой RNN:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

Это дискретный аналог дифференциального уравнения:

$$\frac{dh}{dt} = f(h, x; \theta)$$

Нейронные ОДУ (Neural ODEs)

Нейронные ОДУ (Neural Ordinary Differential Equations) — это модели, которые явно параметризуют производную скрытого состояния:

$$\frac{dh(t)}{dt} = f(h(t), t; \theta)$$

Выход модели получается путем интегрирования этого уравнения от начального состояния $h(t_0)$ до конечного времени t_1:

$$h(t_1) = h(t_0) + \int_{t_0}^{t_1} f(h(t), t; \theta) dt$$

Это интегрирование может быть выполнено численно с помощью методов решения ОДУ, таких как метод Рунге-Кутты.

Градиенты для обучения нейронных ОДУ могут быть вычислены с помощью сопряженного метода, который требует решения другого ОДУ в обратном направлении по времени.

Устойчивость и долговременная память

Анализ устойчивости RNN и LSTM можно провести с помощью теории динамических систем. Для простой RNN с линейной активацией:

$$h_t = W_{hh} h_{t-1} + W_{xh} x_t$$

Устойчивость определяется собственными значениями матрицы W_{hh}. Если все собственные значения по модулю меньше 1, система устойчива, но может страдать от проблемы исчезающего градиента. Если есть собственные значения по модулю больше 1, система неустойчива и может страдать от проблемы взрывного градиента.

LSTM и GRU решают эту проблему, используя механизмы ворот, которые позволяют сети выборочно сохранять или забывать информацию, что делает их более устойчивыми к проблемам долговременных зависимостей.

3.1.5 Выпуклая оптимизация в нейронных сетях

Оптимизация функции потерь

Обучение нейронных сетей сводится к минимизации функции потерь:

$$\min_{\theta} L(\theta) = \frac{1}{n} \sum_{i=1}^{n} l(f(x_i; \theta), y_i) + R(\theta)$$

где l — функция потерь для отдельного примера, R — регуляризатор.

Для глубоких нейронных сетей эта задача невыпукла из-за нелинейных функций активации. Однако методы выпуклой оптимизации, такие как градиентный спуск и его модификации, успешно применяются на практике.

Адаптивные методы оптимизации

Адаптивные методы оптимизации, такие как Adam, RMSProp и AdaGrad, адаптируют размер шага для каждого параметра на основе истории градиентов:

1. **Adam**:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

2. **RMSProp**:

$$v_t = \beta v_{t-1} + (1 - \beta) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha g_t}{\sqrt{v_t} + \epsilon}$$

Эти методы помогают преодолеть проблемы с выбором размера шага и ускоряют сходимость в невыпуклых задачах.

Регуляризация и ограничения

Регуляризация помогает предотвратить переобучение и улучшить обобщающую способность модели:

1. **L2-регуляризация**:

$$R(\theta) = \frac{\lambda}{2} \|\theta\|_2^2$$

2. **L1-регуляризация**:

$$R(\theta) = \lambda \|\theta\|_1$$

3. **Dropout**: Случайное отключение нейронов во время обучения с вероятностью p.
4. **Batch Normalization**: Нормализация активаций внутри мини-батча:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma \hat{x} + \beta$$

где μ_B и σ_B^2 — среднее и дисперсия мини-батча, γ и β — обучаемые параметры.

3.2 Пошаговая реализация алгоритмов RL с использованием PyTorch

В этом разделе мы рассмотрим пошаговую реализацию основных алгоритмов глубокого обучения с подкреплением с использованием библиотеки PyTorch.

3.2.1 Реализация Deep Q-Network (DQN)

DQN — это алгоритм, который использует глубокую нейронную сеть для аппроксимации Q-функции в методе Q-обучения.

Шаг 1: Определение архитектуры нейронной сети для DQN

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import gym
import random
from collections import deque

class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Шаг 2: Реализация буфера воспроизведения опыта (Experience Replay Buffer)

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*batch)

        return (
            torch.FloatTensor(np.array(state)),
            torch.LongTensor(np.array(action)),
            torch.FloatTensor(np.array(reward)),
            torch.FloatTensor(np.array(next_state)),
            torch.FloatTensor(np.array(done))
```

```
        )

    def __len__(self):
        return len(self.buffer)
```

Шаг 3: Реализация агента DQN

```
class DQNAgent:
    def __init__(self, state_dim, action_dim, gamma=0.99, epsilon=1.0, epsilon_min=0.01, epsilon_decay=0.995, lr=0.001):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma  # Коэффициент дисконтирования $\gamma$
        self.epsilon = epsilon  # Параметр исследования $\epsilon$
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.lr = lr

        self.policy_net = DQN(state_dim, action_dim)
        self.target_net = DQN(state_dim, action_dim)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)
        self.memory = ReplayBuffer(10000)
        self.batch_size = 64
        self.update_target_every = 100
        self.steps = 0

    def select_action(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_dim)

        state = torch.FloatTensor(np.array([state]))
        with torch.no_grad():
            q_values = self.policy_net(state)
        return q_values.max(1)[1].item()

    def train(self):
        if len(self.memory) < self.batch_size:
            return

        # Выборка из буфера опыта
        state, action, reward, next_state, done = self.memory.sample(self.batch_size)

        # Вычисление Q-значений для текущих состояний
        # $Q(s, a; \theta)$
        q_values = self.policy_net(state).gather(1, action.unsqueeze(1))
```

```
41
42          # Вычисление максимальных Q-значений для следующих состояний
43          # $\max_{a'} Q(s', a'; \theta^-)$
44          with torch.no_grad():
45              next_q_values = self.target_net(next_state).max(1)[0]
46
47          # Вычисление целевых Q-значений
48          # $y = r + (1 - \text{done}) \cdot \gamma \cdot \max_{a'} Q(s',
    a'; \theta^-)$
49          target_q_values = reward + (1 - done) * self.gamma * next_q_values
50
51          # Вычисление функции потерь (MSE)
52          # $L = \text{MSE}(Q(s, a; \theta), y)$
53          loss = F.mse_loss(q_values.squeeze(), target_q_values)
54
55          # Обновление весов
56          self.optimizer.zero_grad()
57          loss.backward()
58          self.optimizer.step()
59
60          # Обновление epsilon
61          self.epsilon = max(self.epsilon_min, self.epsilon *
    self.epsilon_decay)
62
63          # Обновление целевой сети
64          self.steps += 1
65          if self.steps % self.update_target_every == 0:
66              self.target_net.load_state_dict(self.policy_net.state_dict())
```

Шаг 4: Обучение агента DQN

```
1   # Создание среды
2   env = gym.make('CartPole-v1')
3   state_dim = env.observation_space.shape[0]
4   action_dim = env.action_space.n
5
6   # Создание агента
7   agent = DQNAgent(state_dim, action_dim)
8
9   # Параметры обучения
10  num_episodes = 500
11  max_steps = 500
12
13  # Обучение
14  rewards = []
15
16  for episode in range(num_episodes):
17      state = env.reset()
18      episode_reward = 0
```

```
    for step in range(max_steps):
        # Выбор действия
        action = agent.select_action(state)

        # Выполнение действия
        next_state, reward, done, _ = env.step(action)

        # Сохранение опыта
        agent.memory.push(state, action, reward, next_state, done)

        # Обучение агента
        agent.train()

        state = next_state
        episode_reward += reward

        if done:
            break

    rewards.append(episode_reward)

    # Вывод прогресса
    if episode % 10 == 0:
        avg_reward = np.mean(rewards[-10:])
        print(f"Episode: {episode}, Average Reward (last 10): {avg_reward:.2f}, Epsilon: {agent.epsilon:.2f}")

# Сохранение модели
torch.save(agent.policy_net.state_dict(), 'dqn_cartpole.pth')
```

Шаг 5: Тестирование обученного агента

```
# Загрузка модели
agent.policy_net.load_state_dict(torch.load('dqn_cartpole.pth'))
agent.epsilon = 0.0  # Отключение исследования

# Тестирование
test_episodes = 10
test_rewards = []

for episode in range(test_episodes):
    state = env.reset()
    episode_reward = 0
    done = False

    while not done:
        env.render()  # Визуализация среды
        action = agent.select_action(state)
```

```
            next_state, reward, done, _ = env.step(action)

            state = next_state
            episode_reward += reward

        test_rewards.append(episode_reward)
        print(f"Test Episode: {episode}, Reward: {episode_reward}")

    print(f"Average Test Reward: {np.mean(test_rewards):.2f}")
    env.close()
```

3.2.2 Реализация REINFORCE (Policy Gradient)

REINFORCE — это алгоритм градиента политики, который напрямую оптимизирует параметры политики для максимизации ожидаемой суммарной награды.

Шаг 1: Определение архитектуры нейронной сети для политики

```
class PolicyNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.softmax(self.fc2(x), dim=1) # Выход - вероятности $\pi(a|s;\theta)$
        return x
```

Шаг 2: Реализация агента REINFORCE

```
class REINFORCEAgent:
    def __init__(self, state_dim, action_dim, gamma=0.99, lr=0.001):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma # Коэффициент дисконтирования $\gamma$
        self.lr = lr

        self.policy = PolicyNetwork(state_dim, action_dim)
        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)

        self.saved_log_probs = [] # Хранение $\log \pi(a_t|s_t;\theta)$
        self.rewards = [] # Хранение наград $r_t$

    def select_action(self, state):
        state = torch.FloatTensor(np.array([state]))
```

```
        probs = self.policy(state)

        # Выбор действия из распределения
        m = torch.distributions.Categorical(probs)
        action = m.sample()

        self.saved_log_probs.append(m.log_prob(action)) # Сохраняем логарифм вероятности выбранного действия

        return action.item()

    def finish_episode(self):
        # Вычисление дисконтированных наград (returns) $R_t = \sum_{k=t}^T \gamma^{k-t} r_k$
        R = 0
        returns = []

        for r in self.rewards[::-1]:
            R = r + self.gamma * R
            returns.insert(0, R)

        returns = torch.FloatTensor(returns)

        # Нормализация наград для уменьшения дисперсии
        returns = (returns - returns.mean()) / (returns.std() + 1e-9)

        # Вычисление функции потерь $L = -\sum_t \log \pi(a_t|s_t;\theta) R_t$
        policy_loss = []
        for log_prob, R in zip(self.saved_log_probs, returns):
            policy_loss.append(-log_prob * R)

        policy_loss = torch.cat(policy_loss).sum()

        # Обновление весов
        self.optimizer.zero_grad()
        policy_loss.backward() # Вычисление градиента $\nabla_\theta L$
        self.optimizer.step() # Обновление $\theta \leftarrow \theta - \alpha \nabla_\theta L$ (Adam использует адаптивный шаг)

        # Очистка истории
        self.saved_log_probs = []
        self.rewards = []
```

Шаг 3: Обучение агента REINFORCE

```
# Создание среды
env = gym.make('CartPole-v1')
state_dim = env.observation_space.shape[0]
```

```python
action_dim = env.action_space.n

# Создание агента
agent = REINFORCEAgent(state_dim, action_dim)

# Параметры обучения
num_episodes = 1000
max_steps = 1000

# Обучение
rewards = []

for episode in range(num_episodes):
    state = env.reset()
    episode_reward = 0

    for step in range(max_steps):
        # Выбор действия
        action = agent.select_action(state)

        # Выполнение действия
        next_state, reward, done, _ = env.step(action)

        # Сохранение награды
        agent.rewards.append(reward)

        state = next_state
        episode_reward += reward

        if done:
            break

    # Обучение агента в конце эпизода
    agent.finish_episode()

    rewards.append(episode_reward)

    # Вывод прогресса
    if episode % 10 == 0:
        avg_reward = np.mean(rewards[-10:])
        print(f"Episode: {episode}, Average Reward (last 10): {avg_reward:.2f}")

# Сохранение модели
torch.save(agent.policy.state_dict(), 'reinforce_cartpole.pth')
```

3.2.3 Реализация Actor-Critic

Actor-Critic — это алгоритм, который объединяет идеи методов, основанных на ценности, и методов градиента политики. Он использует две нейронные сети: актера (политика) и критика (функция ценности).

Шаг 1: Определение архитектуры нейронной сети для Actor-Critic

```
class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ActorCritic, self).__init__()

        # Общие слои
        self.fc1 = nn.Linear(state_dim, 128)

        # Слои актера (политика)
        self.actor_fc = nn.Linear(128, action_dim)

        # Слои критика (функция ценности)
        self.critic_fc = nn.Linear(128, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))

        # Выход актера (вероятности действий $\pi(a|s;\theta_{actor})$)
        action_probs = F.softmax(self.actor_fc(x), dim=1)

        # Выход критика (оценка состояния $V(s;\theta_{critic})$)
        state_values = self.critic_fc(x)

        return action_probs, state_values
```

Шаг 2: Реализация агента Actor-Critic

```
class ActorCriticAgent:
    def __init__(self, state_dim, action_dim, gamma=0.99, lr=0.001):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma # Коэффициент дисконтирования $\gamma$
        self.lr = lr

        self.model = ActorCritic(state_dim, action_dim)
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr)

        self.saved_actions = [] # Хранение ($\log \pi(a_t|s_t)$, $V(s_t)$)
        self.rewards = [] # Хранение наград $r_t$

    def select_action(self, state):
        state = torch.FloatTensor(np.array([state]))
```

```
        probs, value = self.model(state)

        # Выбор действия из распределения
        m = torch.distributions.Categorical(probs)
        action = m.sample()

        self.saved_actions.append((m.log_prob(action), value)) # Сохраняем лог. вероятности и оценку состояния

        return action.item()

    def finish_episode(self):
        R = 0
        saved_actions = self.saved_actions
        policy_losses = [] # Потери актера
        value_losses = [] # Потери критика
        returns = []

        # Вычисление дисконтированных наград (returns) $R_t = \sum_{k=t}^T \gamma^{k-t} r_k$
        for r in self.rewards[::-1]:
            R = r + self.gamma * R
            returns.insert(0, R)

        returns = torch.FloatTensor(returns)
        # Нормализация returns не обязательна, но может помочь
        # returns = (returns - returns.mean()) / (returns.std() + 1e-9)

        # Вычисление функций потерь
        for (log_prob, value), R in zip(saved_actions, returns):
            advantage = R - value.item() # Преимущество $A_t = R_t - V(s_t)$

            # Функция потерь для политики (актера) $L_{actor} = -\log \pi(a_t|s_t) A_t$
            policy_losses.append(-log_prob * advantage)

            # Функция потерь для функции ценности (критика) $L_{critic} = (R_t - V(s_t))^2$ (MSE)
            value_losses.append(F.mse_loss(value, torch.FloatTensor([R])))

        # Суммарная функция потерь $L = \sum_t (L_{actor}^{(t)} + L_{critic}^{(t)})$
        loss = torch.stack(policy_losses).sum() + torch.stack(value_losses).sum()

        # Обновление весов
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

```
59
60          # Очистка истории
61          self.saved_actions = []
62          self.rewards = []
```

Шаг 3: Обучение агента Actor-Critic

```
1   # Создание среды
2   env = gym.make('CartPole-v1')
3   state_dim = env.observation_space.shape[0]
4   action_dim = env.action_space.n
5
6   # Создание агента
7   agent = ActorCriticAgent(state_dim, action_dim)
8
9   # Параметры обучения
10  num_episodes = 500
11
12  # Обучение
13  for episode in range(num_episodes):
14      state = env.reset()
15      done = False
16      total_reward = 0
17
18      while not done:
19          action = agent.select_action(state)
20          next_state, reward, done, _ = env.step(action)
21
22          agent.rewards.append(reward)
23          state = next_state
24          total_reward += reward
25
26      agent.finish_episode()
27
28      print(f"Episode: {episode}, Total Reward: {total_reward}")
```

3.2.4 Реализация Proximal Policy Optimization (PPO)

PPO — это алгоритм, который улучшает стабильность обучения, ограничивая изменение политики на каждой итерации.

Шаг 1: Определение архитектур актера и критика для PPO

```
1   class ActorCriticPPO(nn.Module):
2       def __init__(self, input_dim, n_actions):
3           super(ActorCriticPPO, self).__init__()
4
5           # Слои актера (политика)
```

```
        self.actor = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, n_actions),
            nn.Softmax(dim=-1) # Выход - вероятности $\pi(a|s;\theta_{actor})$
        )

        # Слои критика (функция ценности)
        self.critic = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 1) # Выход - оценка состояния $V(s;\theta_{critic})$
        )

    def forward(self):
        raise NotImplementedError

    def act(self, state):
        probs = self.actor(state)
        dist = torch.distributions.Categorical(probs)
        action = dist.sample()
        return action, dist.log_prob(action) # Возвращает действие и лог. его вероятности

    def evaluate(self, state, action):
        probs = self.actor(state)
        dist = torch.distributions.Categorical(probs)

        action_log_probs = dist.log_prob(action) # Лог. вероятности заданного действия
        dist_entropy = dist.entropy() # Энтропия распределения (для регуляризации)

        state_value = self.critic(state) # Оценка состояния

        return action_log_probs, state_value, dist_entropy
```

Шаг 2: Реализация буфера для PPO

```
class PPOMemory:
    def __init__(self, batch_size):
        self.states = []
        self.actions = []
```

```
        self.probs = [] # Лог. вероятности действий $\log \pi_{old}(a|s)$
        self.vals = [] # Оценки состояний $V_{old}(s)$
        self.rewards = []
        self.dones = []
        self.batch_size = batch_size

    def store(self, state, action, probs, vals, reward, done):
        self.states.append(state)
        self.actions.append(action)
        self.probs.append(probs)
        self.vals.append(vals)
        self.rewards.append(reward)
        self.dones.append(done)

    def clear(self):
        self.states = []
        self.actions = []
        self.probs = []
        self.vals = []
        self.rewards = []
        self.dones = []

    def generate_batches(self):
        n_states = len(self.states)
        batch_start = np.arange(0, n_states, self.batch_size)
        indices = np.arange(n_states, dtype=np.int64)
        np.random.shuffle(indices)
        batches = [indices[i:i+self.batch_size] for i in batch_start]

        return np.array(self.states), np.array(self.actions),
np.array(self.probs), \
                np.array(self.vals), np.array(self.rewards),
np.array(self.dones), batches
```

Шаг 3: Реализация агента PPO

```
class PPOAgent:
    def __init__(self, state_dim, action_dim, gamma=0.99, alpha=0.0003,
gae_lambda=0.95,
                 policy_clip=0.2, batch_size=64, n_epochs=10):
        self.gamma = gamma # $\gamma$
        self.policy_clip = policy_clip # $\epsilon$ для обрезания
        self.n_epochs = n_epochs # Количество эпох обучения на собранных
данных
        self.gae_lambda = gae_lambda # Параметр $\lambda$ для GAE

        self.actor_critic = ActorCriticPPO(state_dim, action_dim)
        self.optimizer = optim.Adam(self.actor_critic.parameters(),
lr=alpha)
```

```
        self.memory = PPOMemory(batch_size)

    def select_action(self, observation):
        state = torch.FloatTensor(np.array([observation]))
        action, prob = self.actor_critic.act(state)
        value = self.actor_critic.critic(state)

        return action.item(), prob.item(), value.item()

    def store(self, state, action, probs, vals, reward, done):
        self.memory.store(state, action, probs, vals, reward, done)

    def learn(self):
        for _ in range(self.n_epochs):
            state_arr, action_arr, old_prob_arr, vals_arr, reward_arr, dones_arr, batches = self.memory.generate_batches()

            values = vals_arr
            advantage = np.zeros(len(reward_arr), dtype=np.float32)

            # Вычисление преимущества с использованием GAE (Generalized Advantage Estimation)
            # $A_t^{GAE} = \sum_{k=0}^{T-t-1} (\gamma \lambda)^k \delta_{t+k}$, где $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
            for t in range(len(reward_arr)-1):
                discount = 1
                a_t = 0
                for k in range(t, len(reward_arr)-1):
                    delta = reward_arr[k] + self.gamma * values[k+1] * (1-dones_arr[k]) - values[k]
                    a_t += discount * delta
                    discount *= self.gamma * self.gae_lambda
                advantage[t] = a_t

            advantage = torch.FloatTensor(advantage)
            # Нормализация преимущества
            advantage = (advantage - advantage.mean()) / (advantage.std() + 1e-9)
            values = torch.FloatTensor(values)

            for batch in batches:
                states = torch.FloatTensor(state_arr[batch])
                old_probs = torch.FloatTensor(old_prob_arr[batch]) # $\log \pi_{old}(a|s)$
                actions = torch.LongTensor(action_arr[batch])

                # Вычисление новых вероятностей и значений
                new_probs, critic_value, entropy = self.actor_critic.evaluate(states, actions) # $\log \pi_{new}(a|s)$, $V_{new}(s)$, $H(\pi_{new})$
```

```
                critic_value = critic_value.squeeze()

                # Отношение новых и старых вероятностей $r_t(\theta) = \frac{\pi_{new}(a|s)}{\pi_{old}(a|s)} = \exp(\log \pi_{new} - \log \pi_{old})$
                prob_ratio = (new_probs - old_probs).exp()

                # Вычисление суррогатной функции потерь PPO-Clip
                # $L^{CLIP}(\theta) = \mathbb{E}_t [\min(r_t(\theta) A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t)]$
                weighted_probs = advantage[batch] * prob_ratio
                weighted_clipped_probs = torch.clamp(prob_ratio, 1-self.policy_clip, 1+self.policy_clip) * advantage[batch]

                # Функция потерь для актера (минимизируем отрицательное значение)
                actor_loss = -torch.min(weighted_probs, weighted_clipped_probs).mean()

                # Функция потерь для критика (MSE) $L^{VF}(\theta) = (V_{new}(s_t) - R_t)^2$
                returns = advantage[batch] + values[batch] # $R_t = A_t + V_{old}(s_t)$
                critic_loss = F.mse_loss(critic_value, returns)

                # Общая функция потерь $L = L^{CLIP} + c_1 L^{VF} - c_2 H(\pi_{new})$
                total_loss = actor_loss + 0.5 * critic_loss - 0.01 * entropy.mean()

                # Обновление весов
                self.optimizer.zero_grad()
                total_loss.backward()
                self.optimizer.step()

        self.memory.clear()
```

Шаг 4: Обучение агента PPO

```
# Создание среды
env = gym.make('CartPole-v1')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n

# Создание агента
agent = PPOAgent(state_dim, action_dim)

# Параметры обучения
num_episodes = 300
```

```
max_steps = 500
learn_iters = 0
avg_score = 0
n_steps = 0
update_timestep = 20 # Частота обновления

# Обучение
for episode in range(num_episodes):
    observation = env.reset()
    done = False
    score = 0

    for step in range(max_steps):
        action, prob, val = agent.select_action(observation)
        next_observation, reward, done, _ = env.step(action)
        n_steps += 1
        score += reward

        agent.store(observation, action, prob, val, reward, done)

        # Обновление, если накоплено достаточно шагов
        if n_steps % update_timestep == 0:
            agent.learn()
            learn_iters += 1

        observation = next_observation

        if done:
            break

    avg_score = score if episode == 0 else avg_score * 0.95 + score * 0.05

    print(f'Episode: {episode}, Score: {score}, Avg Score:
{avg_score:.2f}, Learning Steps: {learn_iters}')

# Сохранение модели
torch.save(agent.actor_critic.state_dict(), 'ppo_cartpole.pth')
```

3.2.5 Реализация Deep Deterministic Policy Gradient (DDPG)

DDPG — это алгоритм, который объединяет идеи DQN и градиента политики для работы с непрерывными пространствами действий.

Шаг 1: Определение архитектур актера и критика для DDPG

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()

```

```
        self.fc1 = nn.Linear(state_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, action_dim)

        self.max_action = max_action # Максимальное значение действия

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # Выход актера - детерминированное действие $\mu(s;\theta^\mu)$
        # Ограничиваем выход с помощью tanh и масштабируем
        x = torch.tanh(self.fc3(x)) * self.max_action
        return x

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        # Вход критика - состояние и действие
        self.fc1 = nn.Linear(state_dim + action_dim, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, 1) # Выход - Q-значение $Q(s, a; \theta^Q)$

    def forward(self, x, u): # x - state, u - action
        x = torch.cat([x, u], 1) # Конкатенация состояния и действия
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Шаг 2: Реализация шума Орнштейна-Уленбека для исследования

```
class OUNoise:
    # Процесс Орнштейна-Уленбека для генерации коррелированного шума
    # $dX_t = \theta (\mu - X_t) dt + \sigma dW_t$
    def __init__(self, action_dimension, mu=0, theta=0.15, sigma=0.2):
        self.action_dimension = action_dimension
        self.mu = mu
        self.theta = theta
        self.sigma = sigma
        self.state = np.ones(self.action_dimension) * self.mu
        self.reset()

    def reset(self):
        self.state = np.ones(self.action_dimension) * self.mu

    def sample(self):
```

```
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma *
np.random.randn(len(x))
        self.state = x + dx
        return self.state
```

Шаг 3: Реализация агента DDPG

```
class DDPGAgent:
    def __init__(self, state_dim, action_dim, max_action, gamma=0.99,
tau=0.001, lr_actor=0.0001, lr_critic=0.001):
        self.gamma = gamma # $\gamma$
        self.tau = tau # $\tau$ для мягкого обновления целевых сетей

        # Актер и его целевая сеть
        self.actor = Actor(state_dim, action_dim, max_action)
        self.actor_target = Actor(state_dim, action_dim, max_action)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = optim.Adam(self.actor.parameters(),
lr=lr_actor)

        # Критик и его целевая сеть
        self.critic = Critic(state_dim, action_dim)
        self.critic_target = Critic(state_dim, action_dim)
        self.critic_target.load_state_dict(self.critic.state_dict())
        self.critic_optimizer = optim.Adam(self.critic.parameters(),
lr=lr_critic)

        self.memory = ReplayBuffer(1000000) # Буфер воспроизведения опыта
        self.batch_size = 64

        self.noise = OUNoise(action_dim) # Шум для исследования

    def select_action(self, state, add_noise=True):
        state = torch.FloatTensor(state.reshape(1, -1))
        # Выбираем действие детерминированно с помощью актера
        action = self.actor(state).cpu().data.numpy().flatten()

        if add_noise:
            # Добавляем шум для исследования
            action += self.noise.sample()

        # Ограничиваем действие в допустимом диапазоне
        return np.clip(action, -self.actor.max_action,
self.actor.max_action)

    def train(self):
        if len(self.memory) < self.batch_size:
            return
```

```
38
39          # Выборка из буфера опыта
40          state, action, reward, next_state, done =
    self.memory.sample(self.batch_size)
41          state = torch.FloatTensor(state)
42          action = torch.FloatTensor(action)
43          reward = torch.FloatTensor(reward).unsqueeze(1)
44          next_state = torch.FloatTensor(next_state)
45          done = torch.FloatTensor(done).unsqueeze(1)
46
47
48          # --- Обновление критика ---
49          with torch.no_grad():
50              # Выбираем следующее действие с помощью целевого актера
51              next_action = self.actor_target(next_state)
52              # Вычисляем целевое Q-значение с помощью целевого критика
53              # $y = r + \gamma (1 - \text{done}) Q'(s', \mu'(s';
    \theta^{\mu'}); \theta^{Q'})$
54              target_Q = self.critic_target(next_state, next_action)
55              target_Q = reward + (1 - done) * self.gamma * target_Q
56
57          # Текущее Q-значение $Q(s, a; \theta^Q)$
58          current_Q = self.critic(state, action)
59
60          # Функция потерь критика (MSE) $L(\theta^Q) = \mathbb{E}[(y - Q(s,
    a; \theta^Q))^2]$
61          critic_loss = F.mse_loss(current_Q, target_Q)
62
63          # Оптимизация критика
64          self.critic_optimizer.zero_grad()
65          critic_loss.backward()
66          self.critic_optimizer.step()
67
68          # --- Обновление актера ---
69          # Функция потерь актера (минимизируем отрицательное Q-значение)
70          # $L(\theta^\mu) = -\mathbb{E}[Q(s, \mu(s; \theta^\mu);
    \theta^Q)]$
71          actor_loss = -self.critic(state, self.actor(state)).mean()
72
73          # Оптимизация актера по градиенту политики
74          # $\nabla_{\theta^\mu} J \approx \mathbb{E}[\nabla_a Q(s, a;
    \theta^Q)|_{a=\mu(s)} \nabla_{\theta^\mu} \mu(s; \theta^\mu)]$
75          self.actor_optimizer.zero_grad()
76          actor_loss.backward()
77          self.actor_optimizer.step()
78
79          # --- Мягкое обновление целевых сетей ---
80          # $\theta' \leftarrow \tau \theta + (1 - \tau) \theta'$
81          for param, target_param in zip(self.critic.parameters(),
    self.critic_target.parameters()):
```

```
            target_param.data.copy_(self.tau * param.data + (1 - self.tau)
    * target_param.data)

        for param, target_param in zip(self.actor.parameters(),
    self.actor_target.parameters()):
            target_param.data.copy_(self.tau * param.data + (1 - self.tau)
    * target_param.data)
```

3.3 Практические задачи обучения агентов

В этом разделе мы рассмотрим практические задачи обучения агентов с использованием алгоритмов глубокого обучения с подкреплением.

3.3.1 Задача балансировки перевернутого маятника (CartPole)

Постановка задачи

Задача CartPole — это классическая задача управления, в которой агент должен балансировать шест, прикрепленный к тележке, которая может двигаться влево или вправо. Цель — удерживать шест в вертикальном положении как можно дольше.

- **Состояние**: 4-мерный вектор $s \in \mathbb{R}^4$, содержащий положение тележки, скорость тележки, угол шеста и угловую скорость шеста.
- **Действия**: 2 дискретных действия $a \in \{0, 1\}$ (движение влево или вправо).
- **Награда**: $r = +1$ за каждый временной шаг, пока шест не упадет.
- **Окончание эпизода**: Эпизод заканчивается, когда угол шеста превышает 15 градусов от вертикали или тележка выходит за пределы трека.

Выбор алгоритма и математическое обоснование

Для этой задачи мы выберем алгоритм DQN, так как:

1. Пространство действий дискретное и небольшое (2 действия).
2. Пространство состояний непрерывное, но низкоразмерное (4 измерения).
3. Задача требует изучения долгосрочных зависимостей между действиями и наградами.

Математически, DQN аппроксимирует Q-функцию:

$$Q(s, a) \approx Q(s, a; \theta)$$

где θ — параметры нейронной сети.

Обновление параметров происходит путем минимизации функции потерь:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

где θ^- — параметры целевой сети, $\mathcal{D}$ — буфер воспроизведения опыта.

Демонстрация процесса обучения

```python
# Создание среды
env = gym.make('CartPole-v1')
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n

# Создание агента
agent = DQNAgent(state_dim, action_dim)

# Параметры обучения
num_episodes = 500
max_steps = 500

# Обучение
rewards = []

for episode in range(num_episodes):
    state = env.reset()
    episode_reward = 0

    for step in range(max_steps):
        # Выбор действия
        action = agent.select_action(state)

        # Выполнение действия
        next_state, reward, done, _ = env.step(action)

        # Сохранение опыта
        agent.memory.push(state, action, reward, next_state, done)

        # Обучение агента
        agent.train()

        state = next_state
        episode_reward += reward

        if done:
            break

    rewards.append(episode_reward)

    # Вывод прогресса
    if episode % 10 == 0:
```

```
            avg_reward = np.mean(rewards[-10:])
            print(f"Episode: {episode}, Average Reward (last 10):
    {avg_reward:.2f}, Epsilon: {agent.epsilon:.2f}")

    # Визуализация результатов обучения
    import matplotlib.pyplot as plt

    plt.figure(figsize=(10, 5))
    plt.plot(rewards)
    plt.title('Reward per Episode')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.savefig('cartpole_dqn_rewards.png')
    plt.show()
```

3.3.2 Задача игры в Pong

Постановка задачи

Pong — это классическая видеоигра, в которой два игрока управляют ракетками, отбивая мяч. Цель — не пропустить мяч за свою ракетку.

- **Состояние**: Изображение игрового поля (обычно предварительно обработанное, например, 80×80 пикселей).
- **Действия**: 3 дискретных действия $a \in \{0, 1, 2\}$ (вверх, вниз, остаться на месте).
- **Награда**: $r = +1$, когда противник пропускает мяч, $r = -1$, когда агент пропускает мяч, $r = 0$ в остальных случаях.
- **Окончание эпизода**: Эпизод продолжается до достижения определенного счета (например, 21 очко).

Выбор алгоритма и математическое обоснование

Для этой задачи мы выберем алгоритм Policy Gradient (REINFORCE), так как:

1. Задача имеет разреженные и отложенные награды.
2. Пространство состояний высокоразмерное (изображения).
3. Политика может быть напрямую параметризована нейронной сетью.

Математически, REINFORCE максимизирует ожидаемую суммарную награду:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

Градиент этой функции:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_t \right]$$

где $R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ — дисконтированная суммарная награда с момента t.

Демонстрация процесса обучения

```
# Предварительная обработка изображения
def preprocess(image):
    # Преобразование в оттенки серого
    image = image[35:195]  # Обрезка
    image = image[::2, ::2, 0]  # Уменьшение размерности
    image[image == 144] = 0  # Удаление фона
    image[image == 109] = 0  # Удаление фона
    image[image != 0] = 1  # Бинаризация
    return image.astype(np.float32).reshape(1, -1) # Возвращает вектор 1x6400

# Создание среды
env = gym.make('Pong-v0')
state_dim = 80 * 80  # Размер предобработанного изображения
action_dim = 3  # Вверх, вниз, остаться на месте (в реализации REINFORCE выше используется 2 действия, нужно адаптировать)

# Создание агента (нужно адаптировать REINFORCEAgent для 3 действий)
# agent = REINFORCEAgent(state_dim, action_dim) # Пример

# Параметры обучения
num_episodes = 1000

# Обучение (примерный цикл, требует адаптации агента)
# for episode in range(num_episodes):
#     observation = env.reset()
#     prev_x = None
#     episode_reward = 0
#
#     while True:
#         # Предварительная обработка изображения
#         cur_x = preprocess(observation)
#
#         # Вычисление разницы между текущим и предыдущим кадрами
#         x = cur_x - prev_x if prev_x is not None else np.zeros((1, state_dim))
#         prev_x = cur_x
#
#         # Выбор действия
#         action_idx = agent.select_action(x.flatten()) # Передаем плоский вектор
#
#         # Преобразование индекса действия в действие среды
```

```
#         # В Pong обычно действия 2 (UP) и 3 (DOWN). 0 (NOOP) может быть полезно.
#         # Если action_dim=3, то action_idx может быть 0, 1, 2.
#         # Нужно сопоставить их с действиями среды. Например:
#         if action_idx == 0: action = 2 # UP
#         elif action_idx == 1: action = 3 # DOWN
#         else: action = 0 # NOOP (если action_dim=3)
#         # Или если action_dim=2 (только UP/DOWN):
#         # action = action_idx + 2
#
#         # Выполнение действия
#         observation, reward, done, _ = env.step(action)
#
#         # Сохранение награды
#         agent.rewards.append(reward)
#         episode_reward += reward
#
#         if done:
#             # Обучение агента в конце эпизода
#             agent.finish_episode()
#             break
#
#     print(f"Episode: {episode}, Reward: {episode_reward}")
```

(Примечание: Полный код для Pong с REINFORCE требует более тщательной настройки и адаптации, включая обработку действий и возможно использование сверточных сетей вместо полносвязных для работы с изображениями)

3.3.3 Задача управления роботом-манипулятором

Постановка задачи

Задача управления роботом-манипулятором заключается в перемещении конечного эффектора (захвата) робота в целевую позицию. Пример среды - `Reacher-v2` из OpenAI Gym.

- **Состояние**: Вектор $s \in \mathbb{R}^n$, включающий углы суставов робота, угловые скорости, положение конечного эффектора, вектор от эффектора до цели.
- **Действия**: Непрерывный вектор $a \in \mathbb{R}^m$, представляющий моменты сил, прикладываемые к суставам. Значения обычно ограничены, например $a \in [-1, 1]^m$.
- **Награда**: Обычно зависит от расстояния до цели и затраченных усилий. Например, $r = -\text{distance}(\text{effector}, \text{target}) - \lambda \|a\|^2$.
- **Окончание эпизода**: Эпизод заканчивается, когда конечный эффектор достигает цели (в пределах допуска) или истекает максимальное время.

Выбор алгоритма и математическое обоснование

Для этой задачи мы выберем алгоритм DDPG, так как:

1. Пространство действий непрерывное.
2. Задача требует точного управления.
3. Награда зависит от точности позиционирования.

Математически, DDPG объединяет идеи DQN и градиента политики. Он использует детерминированную политику $\mu(s;\theta^\mu)$ и Q-функцию $Q(s,a;\theta^Q)$.

Обновление критика происходит путем минимизации функции потерь:

$$L(\theta^Q) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(y - Q(s,a;\theta^Q)\right)^2\right]$$

где целевое значение $y = r + \gamma Q'(s', \mu'(s';\theta^{\mu'});\theta^{Q'})$ вычисляется с использованием целевых сетей (обозначены штрихом).

Обновление актера происходит путем максимизации ожидаемой Q-ценности, используя детерминированный градиент политики:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q(s,a;\theta^Q)|_{a=\mu(s;\theta^\mu)} \nabla_{\theta^\mu}\mu(s;\theta^\mu)\right]$$

Демонстрация процесса обучения

```python
# Создание среды (Пример: Pendulum-v1, т.к. Reacher-v2 может требовать MuJoCo)
# env = gym.make('Reacher-v2')
env = gym.make('Pendulum-v1') # Простая среда с непрерывными действиями
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

# Создание агента
agent = DDPGAgent(state_dim, action_dim, max_action)

# Параметры обучения
num_episodes = 200 # DDPG может требовать меньше эпизодов, но больше шагов
max_steps = 200 # Для Pendulum
rewards_history = []

# Обучение
for episode in range(num_episodes):
    state = env.reset()
    agent.noise.reset()
    episode_reward = 0

    for step in range(max_steps):
        action = agent.select_action(state)
        next_state, reward, done, _ = env.step(action * max_action) # Масштабируем действие к диапазону среды
```

```
25
26          # Сохраняем опыт (state, action (в диапазоне [-1, 1]), reward,
    next_state, done)
27          agent.memory.push(state, action, reward, next_state, float(done))
28          agent.train()
29
30          state = next_state
31          episode_reward += reward
32
33          # В Pendulum нет 'done', эпизод всегда длится max_steps
34          # if done:
35          #     break
36
37      rewards_history.append(episode_reward)
38
39      if episode % 10 == 0:
40          avg_reward = np.mean(rewards_history[-10:])
41          print(f"Episode: {episode}, Average Reward (last 10):
    {avg_reward:.2f}")
42
43  # Визуализация результатов
44  import matplotlib.pyplot as plt
45
46  plt.figure(figsize=(10, 5))
47  plt.plot(rewards_history)
48  plt.title('Reward per Episode (Pendulum DDPG)')
49  plt.xlabel('Episode')
50  plt.ylabel('Reward')
51  plt.savefig('pendulum_ddpg_rewards.png')
52  plt.show()
53
54  # Демонстрация обученного агента
55  def show_video(agent, env, num_episodes=5):
56      max_action = float(env.action_space.high[0])
57      for episode in range(num_episodes):
58          state = env.reset()
59          done = False
60          episode_reward = 0
61          step = 0
62          max_steps_eval = 200 # Ограничение для Pendulum
63
64          while not done and step < max_steps_eval:
65              env.render()
66              action = agent.select_action(state, add_noise=False)
67              next_state, reward, done, _ = env.step(action * max_action)
68              episode_reward += reward
69              state = next_state
70              step += 1
71
72          print(f"Episode {episode}, Reward: {episode_reward}")
```

```
73
74        env.close()
75
76    # show_video(agent, env) # Раскомментировать для просмотра
```