ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Буянтуев Александр Алексеевич

# Построение линейных избыточных кодов при помощи обучения с подкреплением

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., профессор Кузькин В. А.

Рецензент:
ООО «МПГ АйТи Солюшнз», инженер ключевых проектов,
Пусев Р. С.

Консультант:
ООО «Техкомпания Хуавей», инженер ключевых проектов,
Онегин Е. Е.

Санкт-Петербург
2024

# Оглавление

# Аннотация

Данная работа посвящена разработке программной системы для решения ряда задач дискретной оптимизации порождающей матрицы двоичного линейного блокового кода и оценки его эффективности. Входными данными для программы оптимизации являются размерность порождающей матрицы, тип и параметры канала, функции оценки порождающей матрицы. Выходом является порождающая матрица заданной размерности, которая минимизирует заданный функционал.

Для решения задачи оптимизации было использовано обучение с подкреплением (reinforcement learning, RL). В качестве критерия эффективности кодов рассматривалась оценка вероятности успешной передачи сообщения (success rate, SR) через заданный канал связи с потерями. Разработанная программная система RL-FEC позволяет моделировать различные условия потерь данных, рассчитывать оценку эффективности кода и получать двоичные линейные блоковые коды с высоким показателем SR.

В рамках ряда проведенных экспериментов с различными параметрами и каналами передачи данных с потерями были получены двоичные линейные блоковые коды, которые сравнимы по показателю SR с лучшими представителями других методов кодирования.

*Ключевые слова:* упреждающая коррекция ошибок (FEC), обучение с подкреплением (RL), линейный блоковый код, порождающая матрица, канал передачи с потерями.

# Abstract

This work is devoted to the development of a software system for solving several problems of discrete optimization of the generator matrix of a binary linear block code and evaluating its efficiency. The input data for the optimization program are the dimension of the generator matrix, the type and parameters of the channel, and the evaluation functions of the generator matrix. The output is a generator matrix of a given dimension that minimizes a given function.

Reinforcement learning (RL) was used to solve the optimization problem. The evaluation of the probability of successful transmission of a message (success rate, SR) through a given communication channel was considered as a criterion for the efficiency of codes. The developed RL-FEC software system allows you to simulate various data loss conditions, calculate code efficiency, and obtain binary linear block codes with a high SR.

As part of several experiments conducted with various parameters and data transmission channels, binary linear block codes were obtained, that are comparable in terms of SR with the best representatives of other coding methods.



*Key concepts:* Forward Error Correction (FEC), reinforcement learning (RL), linear block code, generator matrix, erasure channel.

# Введение

## Актуальность работы

Линейные блоковые коды используются для защиты данных в процессе передачи по каналу, в котором возникают потери. Общая идея заключается в том, чтобы к сообщению длины $n$ добавить $m$ дополнительных символов, которые помогли бы восстановить исходные данные в случае потерь. Известно, что если в процессе передачи было потеряно более $m$ символов, то восстановить изначальное сообщение невозможно. Если же было потеряно не более чем $m$ символов, то существует несколько линейных блоковых кодов, позволяющих решить данную задачу.

К примеру, коды с максимальным достижимым расстоянием (Maximum Distance Separable, MDS) позволяют восстановить сообщение, если получены любые $n$ символов закодированного сообщения. Пожалуй, самыми известными кодами, обладающими данным свойством, являются коды Рида-Соломона [16]. Несмотря на оптимальность с точки зрения количества восстановимых паттернов потерь в закодированном сообщении, на практике реализация процессов кодирования и декодирования связаны с большим количеством операций умножения в конечных полях порядка $2^q$. В силу того, что подавляющее большинство цифровых устройств не имеют поддержки упомянутых операций на уровне инструкций процессора, данные вычисления могут существенно влиять на быстродействие и энергоэффективность практических реализаций алгоритмов кодирования, что в частности, существенно ограничивает их применение для защиты данных в сетевых протоколах транспортного уровня на мобильных устройствах (например, видеоконференцсвязь).

В связи с этим, многие исследователи обратили свое внимание на вопрос построения линейных блоковых кодов над полем $\mathbb{F}_2$ по следующим причинам:

1. реализации алгоритмов кодирования и декодирования не требуют операций умножения в конечных полях,

2. возможно добиться низкой вычислительной сложности декодирования за счет использования субоптимальных декодеров,

3. данные алгоритмы хорошо масштабируются до задач, в которых каждый символ закодированного сообщения представляет собой вектор из $\mathbb{F}_2$ (пакет данных).

Примерами являются различные фонтанные коды, Low Density Parity-Check (LDPC) коды (коды с малой плотностью проверок на чётность) [17], FlexFEC [15], и др. Несмотря на то, что двоичные линейные коды не являются кодами с максимальным достижимым расстоянием, для многих из них показано, что они асимптотически стремятся к оптимуму с ростом длины кода.

На практике часто возникает задача поиска наилучшей по некоторой метрике $f$ порождающей матрицы линейного кода заданной размерности, когда имеется фиксированный декодер, а модель потерь и ее параметры известны приблизительно (в том числе в виде эмпирической оценки вероятности паттернов ошибок, возникающих в действительности). Такая постановка задачи выходит за рамки классических задач теории помехоустойчивого кодирования, и может быть сформулирована как следующая задача дискретной оптимизации:

$$G^* = \arg\min_G f(G),\ G \in \mathbb{F}_2^{n\times(n+m)}.$$

В последнее время появились работы [5, 6], в которых демонстрируется успешное применение обучения с подкреплением для задач дискретной оптимизации. Преимущество данного подхода состоит в том, что он позволяет оставлять архитектуру алгоритма неизменной, при этом настраивать входные параметры под каждую конкретную задачу, что существенно экономит время на поиск оптимальных порождающих матриц для исследователя.

## Цель и задачи

Цель данной дипломной работы заключается в создании программного обеспечения RL-FEC для построения при помощи обучения с подкреплением линейных блоковых кодов над полем $\mathbb{F}_2$, наиболее оптимальных по заданной метрике $f$.

Для достижения этой цели были поставлены следующие задачи:

1. Разработать пайплайн обучения с подкреплением для построения линейных блоковых кодов над полем $\mathbb{F}_2$:
   - Разработать среду обучения для построения и модификации линейных кодов.
   - Разработать агента для взаимодействия со средой.
   - Разработать функцию награды для обучения агента.
2. Провести тестирование RL-FEC на основе показателя SR:
   - Разработать алгоритм для подсчета SR линейного кода в рамках заданной модели с потерями.
   - Провести эксперименты на разных моделях потерь.
   - Сравнить полученные линейные коды с другими методами.

## Достигнутые результаты

В рамках данной работы было разработано программное обеспечение RL-FEC для построения линейных блоковых кодов в поле $\mathbb{F}_2$. В качестве алгоритма обучения с подкреплением был использован DoubleDQN. RL-FEC позволяет симулировать две модели потерь данных: Binary Erasure Channel (BEC) и простая модель Гильберта. Для оценки эффективности полученных при помощи RL-FEC линейных кодов используется метрика SR. Для сравнения эффективности на основе SR были выбраны коды Рида-Соломона, LDPC-коды и алгоритм FlexFEC. В рамках ряда проведенных экспериментов с различными параметрами и каналами передачи данных с потерями были получены двоичные линейные блоковые коды, которые сравнимы по показателю SR с лучшими представителями других методов кодирования.

## Структура работы

В **первой** главе рассматривается литература по моделям потерь данных, алгоритма избыточного кодирования информации, а также по алгоритмам обучения с подкреплением. Во **второй** главе рассматривается общий принцип работы программного обеспечения RL-FEC. В последующих главах данной дипломной работы рассматриваются внутренние механизмы, которые были использованы для эффективной работы RL-FEC. В **третьей** главе рассматриваются различные способы представления линейных блоковых кодов. В **четвертой** главе рассматривается архитектура обучения с подкреплением, которая использовалась в данной работе для построения и модификации линейных блоковых кодов. В **пятой** главе рассматривается алгоритм, который был использован для оценки эффективности линейных кодов на основе метрики SR. В **шестой** главе приводится сравнение полученных кодов с другими кодами на основе метрики SR.

# 1. Обзор литературы

## 1.1. Модели потерь данных

### 1.1.1. Binary Erasure Channel (BEC)

Канал двоичного стирания (Binary Erasure Channel, BEC) является моделью канала связи, который часто используется в теории кодирования. BEC позволяет моделировать процесс стирания в момент передачи. Передатчик отправляет один бит информации по каналу, а приемник либо получает бит корректно, либо с некоторой вероятностью $p$ бит теряется. Формально этот процесс можно изобразить при помощи следующей схемы:

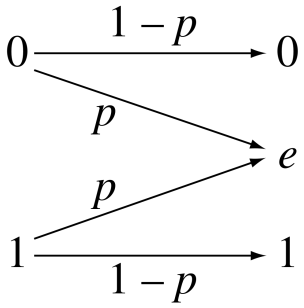



Рис. 1: Принцип работы канала BEC.

Данный канал можно также рассматривать и для пакетов данных, когда в процессе передачи с какой-то вероятностью $p$ теряется весь пакет целиком.

### 1.1.2. Модель Гильберта

Данная модель представляет собой цепь Маркова с двумя состояниями (*хорошее* и *плохое*) и переходами между ними. Существует несколько различных вариаций данной модели, которые позволяют моделировать разные условия передачи по сети. Более подробно свойства каждой модели были рассмотрены в статье [10].

#### Простая модель Гильберта

У данной модели есть 2 независимых параметра — $p$ и $q$. Когда канал передачи находится в *хорошем* состоянии, считается, что пакет данных был доставлен, в *плохом* состоянии — потерян. Модель позволяет изучить каналы, где возможна ситуация последовательной потери нескольких пакетов.

BEC является частным случаем простой модели Гильберта, где $q = 1 - p$.

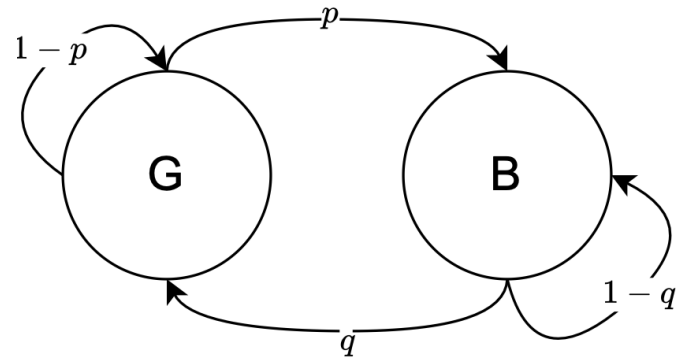



Рис. 2: Принцип работы простой модели Гильберта.

**Модель Гильберта**

В этой модели добавляется еще один параметр $h$ — вероятность того, что пакет данных был доставлен, когда канал передачи находился в *плохом* состоянии. Данный параметр позволяет моделировать плотность потерь ($\mu = 1 - h$) в канале.

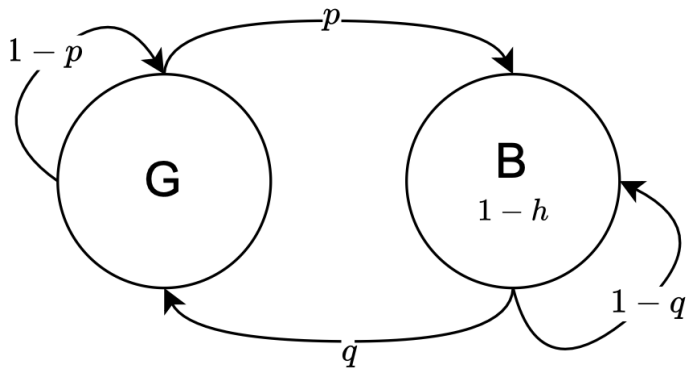



Рис. 3: Принцип работы модели Гильберта.

**Модель Гильберта-Эллиота**

Эта модель является обобщенным вариантом модели Гильберта. Добавляется параметр $k$ — вероятность того, что пакет данных был доставлен, когда канал передачи находился в *хорошем* состоянии.

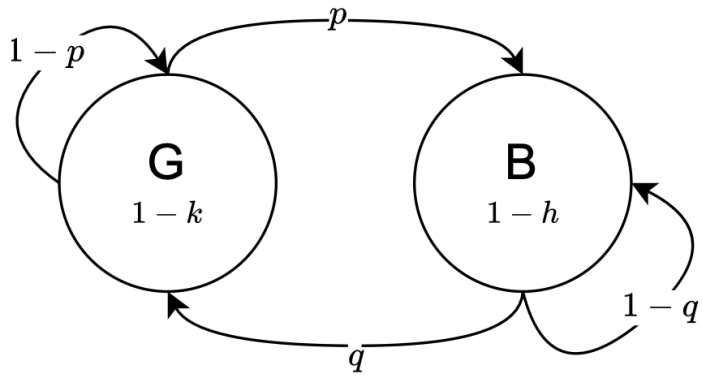



Рис. 4: Принцип работы модели Гильберта-Эллиота.

## 1.2. Линейные коды

В рамках данной главы рассматриваются методы, с которыми в дальнейшем будет проводиться сравнение линейных блоковых кодов, найденных при помощи RL-FEC. Для сравнения были выбраны следующие методы:

1. Коды Рида-Соломона [16], так как они являются представителями класса MDS кодов. Такие коды позволяют гарантированно восстановить сообщение длины $n$, если было передано $n + m$ символов и в процессе передачи было потеряно не более $m$ символов.

2. Алгоритмы, использующие операции в поле $\mathbb{F}_2$, такие как:

   - LDPC-коды [17], так как они обладают высокой эффективностью в каналах с потерями.
   - FlexFEC [15], так как процесс кодирования можно адаптировать под характеристики канала передачи.

Подробное описание того, почему были выбраны такие методы для сравнения, представлено в *главе* 6.

### 1.2.1. Коды Рида-Соломона

Коды Рида-Соломона [16] используют алгебраические принципы для обеспечения коррекции ошибок. В их основе лежат операции над элементами полей Галуа. Процесс кодирования информации с использованием кодов Рида-Соломона устроен следующим образом:

1. Исходное сообщение представляется в виде многочлена $m(x)$, при этом каждый символ сообщения интерпретируется как коэффициент многочлена. Для сообщения из $k$ символов многочлен будет иметь вид:

$$m(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_1x + m_0,$$

где $m_0, m_1, \ldots, m_{k-1}$ — символы сообщения, представленные в виде элементов конечного поля.

2. Определяется генерирующий многочлен $g(x)$ степени $n-k$:

$$g(x) = (x - \alpha^0)(x - \alpha^1)\ldots(x - \alpha^{n-k-1}),$$

где $\alpha$ — примитивный элемент поля Галуа.

3. Закодированное сообщение $c(x)$ получают путем перемножения многочленов:

$$c(x) = m(x) \times g(x).$$

4. Коэффициенты $c_{n-1}, c_{n-2}, \ldots, c_0$ многочлена $c(x)$ составляют итоговое кодовое слово, готовое к передаче по каналу связи.

Так как коды относятся к классу MDS, вероятность доставки сообщения по каналу с потерями для них будет выше в сравнении с другими алгоритмами. Однако при работе с байтами для построения кодов Рида-Соломона приходится использовать операции в поле $\mathbb{F}_{256}$, что существенно влияет на вычислительную сложность кодирования и декодирования.

### 1.2.2. Поле $\mathbb{F}_2$

Поле $\mathbb{F}_2$ задано множеством $\{0, 1\}$, на котором определены две операции:

- «исключающее или» (XOR, $\oplus$): $0 \oplus 0 = 0,\ 0 \oplus 1 = 1,\ 1 \oplus 0 = 1, 1 \oplus 1 = 0$,
- «и» (AND, $\cdot$): $0 \cdot 0 = 0,\ 0 \cdot 1 = 0,\ 1 \cdot 0 = 0, 1 \cdot 1 = 1$.

### 1.2.3. LDPC-коды

LDPC (Low-Density Parity-Check) коды [17] — это линейные блоковые коды над полем $\mathbb{F}_2$. Основная идея LDPC-кодов заключается в использовании разреженных матриц проверок на четность, что позволяет эффективно декодировать сообщения с помощью итеративных алгоритмов. В общем случае считаетс, что количество единиц в матрице растет линейно с увеличением количества столбцов проверочной матрицы.

**Регулярные** LDPC-коды обладают следующими свойствами:

- все строки матрицы содержат $i < n$ единиц,
- все столбцы матрицы содержат $j < (n - k)$ единиц.

### 1.2.4. FlexFEC

Алгоритм избыточного кодирования FlexFEC [15] является частью фреймворка WebRTC, который используется для передачи информации в режиме реального времени. Данный алгоритм позволяет эффективно кодировать различные виды файлов, такие как текст, аудио, видео и даже исполняемые файлы. FlexFEC тажке как и LDPC-коды использует только операции в $\mathbb{F}_2$ для создания избыточных данных. Кодирование при помощи FlexFEC устроено следующим образом:

1. Формируется окно размера $D \times L$, куда складываются пакеты передаваемого сообщения.

2. Когда окно полностью заполнено, применяем операцию XOR, используя один из вариантов кодирования:

   - 1-D кодирование по рядам — над подряд идущими в строке окна $L$ пакетами и получаем один дополнительный пакет;
   - 1-D кодирование по столбцам — над подряд идущими в столбце окна $D$ пакетами и получаем один дополнительный пакет;
   - 2-D кодирование — по рядам и столбцам одновременно;

Еще одним преимуществом данного подхода является его *гибкость* — можно настраивать размер окна в зависимости от характеристик канала передачи. Это позволяет использовать FlexFEC для построения различных вариантов кодирования информации при заданных $k$ и $n$. Можно также кодировать исходное сообщение, используя различные конфигурации окна, а затем передавать все полученные избыточные пакеты по одному каналу связи, тем самым увеличивая эффективность передачи.

## 1.3. Алгоритмы обучения с подкреплением

Обучение с подкреплением (Reinforcement Learning, RL) — это область машинного обучения, в которой агент учится принимать решения, взаимодействуя с окружающей средой для достижения цели. За совершаемые действия агент получает от среды определенную награду. Задача агента — научиться максимизировать суммарную награду, полученную во время сессии взаимодействия со средой.

Более формально процесс обучения с подкреплением можно описать так: агент находится в состоянии среды $s$ и совершает действие $a$. От среды в ответ он получает новое состояние $s'$, награду за действие $r$, а также флаг $d$, который показывает, является ли новое состояние конечным (критерий окончания сессии). На основе наблюдений $(s, a, s', r, d)$ строится процесс обучения агента.

В данной главе рассматриваются ключевые алгоритмы обучения с подкреплением, а также их применение для решения различных задач.

### 1.3.1. DQN

DQN — это один из первых алгоритмов обучения с подкреплением, который использовал глубокие нейронные сети. Этот подход был разработан компанией DeepMind [12] и показал выдающиеся результаты на нескольких играх Atari, при этом дополнительная настройка архитектуры алгоритма не проводилась.

В основе данного алгоритма лежит Q-функция $Q(s, a)$, которая оценивает качество выполнения действия $a$ в состоянии $s$. Также она представляет ожидаемое суммарное вознаграждение, которое агент получит, начиная с состояния $s$ и выбрав действие $a$, а затем следуя оптимальной политики. Цель алгоритма DQN — аппроксимировать Q-функцию с помощью нейронной сети. В процессе аппроксимации мы хотим приблизить Q-функцию $Q(s, a)$ к оптимальной Q-функции $Q^*(s, a)$, удовлетворяющей уравнению Беллмана:

$$Q^*(s, a) = \mathbb{E}[R(s, a) + \gamma \max_{a'} Q^*(s', a')],$$

где:

$R(s, a)$ – вознаграждение после выполнения действия $a$ в состоянии $s$,
$s'$ – новое состояние,
$a'$ – любое возможное действие в новом состоянии,
$\gamma$ – коэффициент дисконтирования, который уменьшает значение будущих вознаграждений.

Для этого в в алгоритме вводится уравнение обновления для аппроксимации с помощью минимизации потерь:

$$L(\theta) = \mathbb{E}[(R(s, a) + \gamma \max_{a'} Q_{\text{target}}(s', a'; \theta^-) - Q(s, a; \theta))^2],$$

где:

$\theta$ – веса основной сети $Q$,
$\theta^-$ – веса целевой сети $Q_{\text{target}}$.

#### Experience replay

Для повышения эффективности и стабильности обучения DQN используют технику *experience replay*. В памяти сохраняются все переходы агента в виде $(s, a, R, s', d)$. Затем из этой памяти случайный образом выбирается небольшой набор данных, которые используют для обучения. Такое решение помогает избавиться от корреляции между последовательными шагами обучения.

### 1.3.2. DoubleDQN

Данный алгоритм является улучшением стандартного DQN. Модификация алгоритма была предложена для решения проблемы переоценки значений Q-функции [21]. Для этого используется сразу две нейронных сети: одна — для выбора действия, а другая — для оценки ценности действия. Такое изменение позволило увеличить стабильность обучения. В алгоритме DoubleDQN уравнение обновления устроено следующий образом:

$$L(\theta) = \mathbb{E}[(R(s,a) + \gamma Q_{\text{target}}(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - Q(s,a;\theta))^2],$$

где:

$\theta$ — веса основной сети $Q$,
$\theta^-$ — веса целевой сети $Q_{\text{target}}$.

### 1.3.3. SAC

Алгоритм Soft Actor-Critic (SAC) представляет собой метод обучения с подкреплением, основанный на максимизации суммарного вознаграждения и энтропии политики [18]. Это делает его особенно эффективным при решение задач в сложных и многомерных средах. В основе алгоритма лежит два основных компонента: *actor* — агент, который выбирает действия, и *critic* — агент, который оценивает действия.

В SAC используется стохастическая политика $\pi_\phi(a|s)$. Энтропия этой политики используется для поощрения исследования и помогает избежать локальных оптимумов. Также в алгоритме определяются функции $V_\phi(s)$ и $Q_\theta(s,a)$:

$$V_\psi(s) = \mathbb{E}_{a\sim\pi_\phi}[Q_\theta(s,a) - \log \pi_\phi(a|s)],$$

$$Q_\theta(s,a) = \mathbb{E}_{s'\sim\mathcal{P}, a'\sim\pi_\phi}[R(s,a) + \gamma V_{\psi'}(s')].$$

Оптимизация параметров $\phi, \psi, \theta$ происходит путем минимизации соответствующих функций потерь:

$$J_\pi(\phi) = \mathbb{E}_{s\sim\mathcal{D}, a\sim\pi_\phi}[\log \pi_\phi(a|s) - Q_\theta(s,a)],$$

$$J_Q(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}[(Q_\theta(s,a) - r - \gamma V_{\psi'}(s'))^2],$$

$$J_V(\psi) = \mathbb{E}_{s\sim\mathcal{D}}[(V_\psi(s) - \mathbb{E}_{a\sim\pi_\phi}[Q_\theta(s,a) - \log \pi_\phi(a|s)])^2].$$

Данный алгоритм показал повышенную эффективность с точки зрения скорости обучения и финальной производительности на сложных средах обучения (Ant-v1, Humanoid-v1) в сравнение с другими алгоритмами обучения с подкреплением, таких как DDPG [3] и PPO [13].

### 1.3.4. Сценарии применения обучения с подкреплением

Алгоритмы обучения с подкреплением получили свою популярность благодаря их способности превосходить человека на играх для одного игрока — Atari, и двух игроков — AlphaGo. Помимо этого данные алгоритмы используются и для решения прикладных задач в областях энергетики, финансов, здравоохранения и робототехники [9].

Обучение с подкреплением также используют для поиска новых алгоритмов. Так, авторы статьи [6] при помощи обучения с подкреплением открыли более эффективные алгоритмы перемножения матриц.

Существуют также исследования, изучающие эффективность данного подхода для построения графов под определенные условия [5], что показывает возможность эффективности обучения с подкреплением для построения линейных блоковых кодов, так как их можно представлять при помощи двудольных графов (о чем будет рассказано в *главе* 3).

# 2. Принцип работы RL-FEC

Программное обеспечение RL-FEC работает следующим образом:

1. Происходит инициализация RL-FEC следующими параметрами:
   - модель потерь данных $M$,
   - параметры линейных блоковых кодов — $n$ и $m$,
   - функция для оценки эффективности линейных блоковых кодов на основе порождающей матрицы $G$ — $f(G; M)$, которая будет оптимизироваться в процессе обучения,
   - среда $E$ для построения и оптимизации линейных блоковых кодов на основе функции $f$.
2. Запускается обучения агента $A$. В процессе обучения агент взаимодействует со средой $E$. Агент использует обучение с подкреплением для того, чтобы найти линейный блоковый код над полем $\mathbb{F}_2$ с оптимальной порождающей матрицей $G^*$ для заданных $f$ и $M$:

   $$f(G^*; M) = \max_G f(G; M).$$

   На основе значения функции $f$ агент принимает решение о том, как модифицировать линейный блоковый код.
3. Найденные агентом линейные коды сохраняются в процессе обучения для дальнейшего сравнения с другими методами.

Такая конфигурация программного обеспечения позволяет получить линейные коды практически для любых условий, потому что:

1. в качестве модели $M$ можно использовать эмпирические модели, которые сложно поддаются строгому математическому описанию,
2. при изменении среды $E$ можно изменить структуру линейных кодов или наложить на них дополнительные ограничения,
3. функция $f$ может задавать любую метрику для оценки кодов (вероятность успешной передачи, сложность кодирования/декодирования и др.)

Это делает RL-FEC достаточно универсальным решением, которое легко можно адаптировать под требования конкретной области применения линейных кодов.

Текущая реализация RL-FEC позволяет моделировать следующие параметры:

1. модель потерь $M$:

- Binary Erasure Channel (BEC),
- простая модель Гильберта.

2. параметры линейных кодов — можно использовать любые $n$ и $m$.
3. в качестве функции $f$ сейчас можно использовать метрику SR.

# 3. Способы представления линейных блоковых кодов

В данной главе рассматривается три способа представления линейных блоковых кодов. Каждое из этих представлений можно получить из другого. У каждого из представлений есть свои преимущества, которые можно использовать для обучения агента. Помимо этого, в *разделе* 3.4 показана целесообразность использования обучения с подкреплением для построения линейных кодов.

В реализации RL-FEC было использовано *компактное представление порождающей матрицы*, так как оно позволяет значительно ускорить процесс обучения агента. Изучение других представлений и их преимуществ мы оставили для дальнейших исследований.

## 3.1. Порождающая матрица

Порождающая матрица линейного кода — это матрица линейного оператора, который отображает исходное сообщение в закодированное. Другими словами этот процесс можно описать так:

Все операции совершаются над полем Галуа $\mathbb{F}_q$. Пусть у нас есть исходное сообщение $m = (m_1, \ldots, m_n) \in \mathbb{F}_q^n$, состоящее из $n$ символов. Мы хотим получить закодированное сообщение $s = (s_1, \ldots, s_{n+m}) \in \mathbb{F}_q^{n+m}$, состоящее из $n + m$ символов. Для этого мы используем порождающую матрицу $G$ размера $n \times (n + m)$, преобразование выглядит следующим образом:

$$s = mG.$$

В качестве примера приведем порождающую матрицу кода Хэмминга над полем $\mathbb{F}_2^7$:

$$G_{\text{Ham}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Удобно рассматривать порождающие матрицы в стандартной форме:

$$G_{n\times(n+m)} = [I_n|P],$$

где $I_n$ — единичная матрица размера $n$, а $P$ — матрица размера $n \times m$.

В рамках нашего исследования мы рассматривали линейные коды над полем $\mathbb{F}_2$ с операциями сложения и умножения по модулю 2. Такие коды не требуют вычислительно сложных операций при кодировании и декодировании, но с ними также можно добиться высокого уровня эффективности передачи информации.

Преимущество представления линейных блоковых кодов при помощи порождающих матриц заключается в том, что их можно использовать вместе со сверточными нейронными сетями.

## 3.2. Двудольный граф

Линейные коды также можно представлять при помощи двудольных графов. Такой граф легко получить из порождающей матрицы $G$. Рассмотрим порождающую матрицу $G$ в стандартной форме. Двудольный граф $B(n, m)$ построим следующим образом:

1. Пусть $n$ — размер левой доли, $m$ — размер правой доли.
2. Рассмотрим столбцы матрицы $P$. Каждый столбец задает вершину правой доли.
3. Теперь построим ребра двудольного графа. Рассмотрим $p_i$ — $i$-й столбец матрицы $P$. Если $p_{ij} = 1$, то соединим ребром $j$-ю вершину левой доли с $i$-й вершиной правой доли.

Рассмотрим пример преобразования. Мы передаем сообщение из $n = 2$ символов и используем для кодирования $m = 3$ избыточных символа. Порождающая матрица выглядит следующим образом:

$$G_{2\times 5} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Из данной матрицы получаем двудольный граф $B(2, 3)$ (см. *рис.* 5).

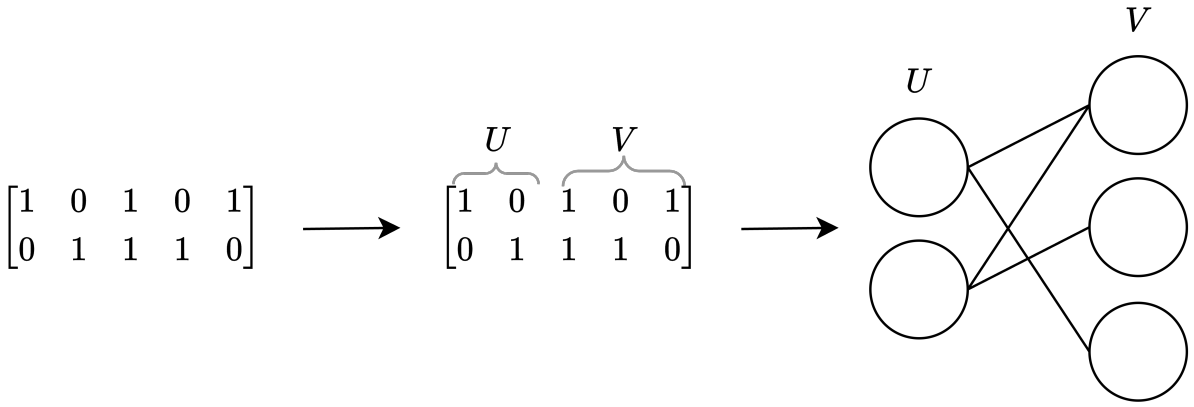



Рис. 5: Преобразование порождающей матрицы в двудольный граф.

Преимущество данного представления заключается в том, что можно использовать эмбеддинги для графов (такой подход использовали в статье [5]).

## 3.3. Компактное представление порождающей матрицы

Из порождающей матрицы линейного блокового кода можно получить компактную форму, которая при этом будет хранить всю необходимую нам информацию. Сделать это можно следующим образом:

1. Рассмотрим порождающую матрицу $G$.
2. Каждый столбец матрицы будем рассматривать, как двоичное число, где биты числа расположены в обратном порядке (первый элемент столбца — младший бит числа).
3. Преобразуем каждый столбец матрицы в десятичное число и получим компактную форму.

Для удобства обозначим данное представление — $CF(n, m)$. Рассмотрим пример преобразования. Используем матрицу $G_{2\times5}$ из предыдущей секции:

$$\underbrace{\begin{bmatrix}1 & 0 & 1 & 0 & 1\\ 0 & 1 & 1 & 1 & 0\end{bmatrix}}_{G_{2\times5}} \rightarrow \underbrace{(1, 2, 3, 2, 1)}_{CF(2,3)}.$$

У данного вида представления есть преимущество — для хранения линейных кодов, у которых $n + m \leq 64$ можно использовать тип данных int64. Это позволяет эффективно хранить такие коды в памяти компьютера, а также быстро совершать операции над столбцами порождающей матрицы. Данное преимущество будет использовано в алгоритме восстановления исходного сообщения (см. *главу* 5).

## 3.4. Количество различных линейных блоковых кодов

Мы можем вывести формулу для подсчета количества различных линейных кодов, что позволит оценить нам сложность исследуемой области.

Давайте рассмотрим двудольный граф $B(n, m)$. Нас интересует принцип построения правой доли данного графа. Его можно описать следующим образом:

1. Начинаем с $m = 0$ — пустая правая доля.
2. Добавляем вершину в правую долю. У нас есть $2^n - 1$ различных способов соединить ее с вершинами левой доли (нам не интересен случай, когда мы не проводим ребра).
3. Таким же образом строим все $m$ вершин правой доли.
4. Также учитываем всевозможные перестановки вершин правой доли (их существует $m!$).

В итоге мы получаем следующую формулу для количества различных линейных блоковых кодов:

$$m! \cdot \binom{2^n - 1}{m}.$$

Данное число достаточно быстро растет, например при $n = 7$ и $m = 5$ будет примерно $2.6 \cdot 10^8$, а при $n = 9$ и $m = 6$ уже примерно $2.4 \cdot 10^{13}$. При этом стоит отметить, что классов эквивалентности двудольных графов тоже много (данное утверждение было доказано в статье [1]).

Именно поэтому возникла идея применения обучения с подкреплением для построения линейных блоковых кодов, так как это в теории позволит быстрее изучить исследуемое пространство и найти эффективные коды.

# 4. Архитектура обучения с подкреплением

В данной главе рассматривается внутренняя архитектура обучения с подкреплением, на котором основывается принцип работы RL-FEC. В *разделе* 4.1 описывается среда, которая была разработана для построения и модификации линейных кодов. В *разделе* 4.2 описывается реализация функции $f(G; M)$, которая позволяет оптимизировать оценку SR. В *разделе* 4.3 описывается сессия взаимодействия со средой, архитектура агента и его гиперпараметры.

## 4.1. Среда

В обучении с подкреплением среда должна описывать Марковский процесс принятия решений (МППР). Давайте опишем этот процесс для нашей задачи следующим образом:

Пусть $n$ — количество символов передаваемого сообщения, $m$ — количество избыточных символов. Мы хотим построить линейный блоковый код для передачи информации, для этого будем использовать *представление* $CF(n, m)$. Начнем с состояния $CF(n, 0)$ — это когда нет избыточной информации. Будем последовательно добавлять по одному избыточному символу. Таким образом, МППР выглядит так:

$$\begin{gathered} CF(n,0) \xrightarrow{a_1} CF(n,1) \xrightarrow{a_2} \ldots \xrightarrow{a_m} CF(n,m), \\ (a_1, a_2, \ldots, a_m) \in A - \text{набор совершенных действий.} \end{gathered} \tag{1}$$

Иными словами это можно описать так — мы моделируем процесс последовательного построения правой доли двудольного графа $B(n, m)$, описанный в *главе* 3.4. Начинаем с пустой правой доли, в левой доли $n$ вершин. Действие — добавить вершину в правую долю и сразу же соединить ее ребрами с каким-то подмножеством вершин левой доли. Заканчиваем, когда в правую долю добавили $m$ вершин.

### 4.1.1. Состояние среды для обучения агента

Для обучения агента нам нужно состояние фиксированного размера, которое мы будем передавать на вход нейронной сети. С помощью этого состояния мы хотим сохранить информацию о структуре двудольного графа. На основе описанного выше алгоритма можно сделать следущие выводы:

1. Размер левой доли графа не меняется, он задается при инициализации среды.
2. Для моделирования процесса построения линейного блокового кода нам важно знать текущее количество вершин в правой доли, а также ребра графа.

В рамках данной работы было принято решение использовать следующее состояние среды для обучения агента $S$:

- $|S| = 2^n$, где $n$ — количество вершин в левой доле.
- $\forall s_i \in S \ : \ s_i \in \{0, 1\}$.

Такое состояние $S$ описывает уникальные действия, которые уже использовал агент во время обучения. Начальное состояние среды $S_0$ будем инициализировать следующим образом:

1. Выбираем количество уже выполненных действий $k \in [0; m-1]$.
2. Случайным образом расставляем $k$ единиц в состоянии $S$. Остальные элементы состояния оставляем 0.

Такое состояние задает двудольный граф, где в левой доле $n$ вершин, в правой доле случайное количество вершин от 0 до $m-1$, каждая из них каким-то образом соединена с вершинами правой доли.

Далее агент совершает действие — добавляет вершину в правую долю и сразу же соединяет ее ребрами с какими-то вершинами левой доли. Данный принцип был описан в *разделе* 3.2.

В состоянии $S$ такое действие можно отобразить так: поставим 1 на позицию, соответствующую компактному представлению столбца порождающей матрицы, которое было описано в *разделе* 3.3. Считаем, что индексация позиций начинается с 0.
При исследовании данного состояния было сделано несколько выводов:

1. Количество единиц в векторе $S$ — количество вершин в правой доле двудольного графа.
2. Позиции единиц в векторе $S$ показывают совершенные агентом действия. Однако $S$ описывает только набор выбранных агентом действий, но не показывает порядок, в котором они были выбраны.
3. Такое состояние автоматически накладывает ограничения на игру агента — если агент выполнит действие, которое уже совершал до этого — состояние $S$ не изменится. С точки зрения решаемой задачи такое ограничение разумно — не нужно несколько раз защищать один и тот же набор изначальных символов.

Рассмотрим на примере, как выглядит процесс изменения состояния $S$. Пусть агент совершил следующую цепочку действий, представленную на рис. 6.

С точки зрения построения порождающей матрицы этот процесс будет выглядеть следующим образом:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

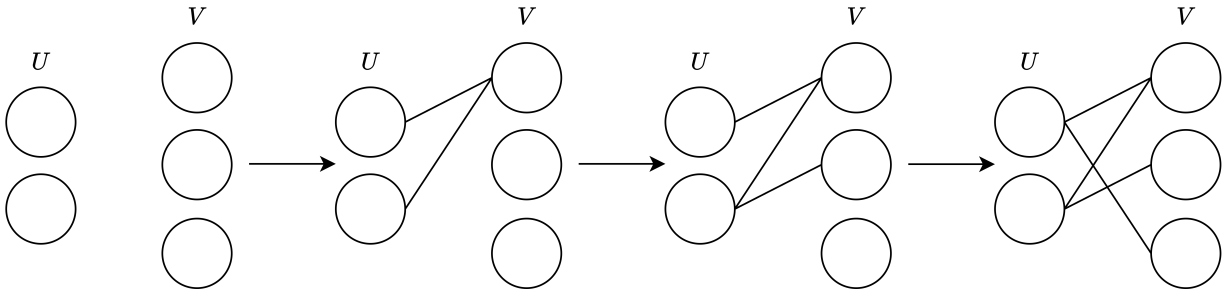



Рис. 6: Цепочка действий, совершенных агентом.

Компактное представление $CF(n, m)$ изменяется следующим образом:

$$\underbrace{(1,2,0,0,0)}_{CF(n,0)} \to \underbrace{(1,2,3,0,0)}_{CF(n,1)} \to \underbrace{(1,2,3,2,0)}_{CF(n,2)} \to \underbrace{(1,2,3,2,1)}_{CF(n,3)}.$$

Теперь посмотрим на процесс изменения состояния $S$ (для удобства обозначим операцию преобразования столбца порождающей матрицы, с помощью которой мы получаем $CF(n, m)$, как $cf(g_i)$):

$$\underbrace{(0,0,0,0)}_{S_0} \xrightarrow{g_1=\begin{bmatrix}1\\1\end{bmatrix},\ cf(g_1)=3} \underbrace{(0,0,0,1)}_{S_1} \xrightarrow{g_2=\begin{bmatrix}0\\1\end{bmatrix},\ cf(g_1)=2} \underbrace{(0,0,1,1)}_{S_2} \xrightarrow{g_3=\begin{bmatrix}1\\0\end{bmatrix},\ cf(g_1)=1} \underbrace{(0,1,1,1)}_{S_3}.$$

Переход, описанный в *пункте* 3 выглядит так:

$$\underbrace{(0,1,0,0)}_{S} \xrightarrow{g=\begin{bmatrix}1\\0\end{bmatrix},\ cf(g)=1} \underbrace{(0,1,0,0)}_{S'}.$$

### 4.1.2. Состояние среды для тестирования линейных блоковых кодов

При тестировании линейных кодов нам может быть важен порядок, в котором агент строил порождающую матрицу кода и совершал действие. Если мы рассматриваем порождающую матрицу $G = [I_n|P]$ на модели BEC, то от перестановки столбцов подматрицы $P$ эффективность кода не изменяется. Однако если мы рассматриваем вариации модели Гильберта-Эллиота, то перестановка столбцов будет влиять на эффективность кода, что было выявлено в процессе экспериментов.

Так как состояние $S$ *не хранит информацию* о порядке совершенных действий, нам нужно будет хранить в среде дополнительное состояние. Можно заметить, что *компактное представление* $CF(n, m)$ хранит в себе информацию о порядке совершенных действий, при этом в точности описывает порождающую матрицу линейного кода, что нам как раз и нужно для тестирования.

**Полное состояние среды**

Определим полное состояние среды $\mathcal{S}$ как пару $(S, CF(n,m))$.
Начальное состояние $\mathcal{S}_0 = (S_0, CF(n,0))$.

## 4.2. Функция награды

От функции награды напрямую зависит, как будет проходить обучение агента. Основная задача агента — максимизировать значение данной функции. При проектировании функции были поставлены следующие задачи:

1. Функция награды должна показывать оценку SR — приближенную вероятность успешной передачи сообщения через канал связи с потерями. Для этого нужно было придумать критерий, позволяющий оценить данную метрику путем проверки $CF(n,m)$.

2. В процессе обучения агент может совершать повторяющиеся действия, но с точки зрения решаемой задачи это не нужно делать. Агент должен обучиться этому на основе функции награды.

В данной главе будут рассмотрены две функции:

- суммарная функция награды $R(\tau)$, которую получает агент по результатам одной сессии взаимодействия со средой,
- функция награды $r(s, a, s')$, которая используется для обучения агента.

### 4.2.1. Суммарная функция награды $R(\tau)$

Было принято решение использовать следующую формулу для суммарной функции награды:

$$R(\tau) = SR[CF(n,m)] + P(\tau) \tag{2}$$

где:

| | |
|---|---|
| $\tau$ | – траектория агента (последовательность состояний и действий), |
| $R(\tau)$ | – финальная награда за траекторию $\tau$, |
| $SR[CF(n,m)]$ | – подсчитанный SR на основе $CF(n,m)$, |
| $P(\tau)$ | – штраф за повторяющиеся действия в траектории агента. |

В данном разделе рассказывается, как рассчитывается каждая из частей данной формулы.

### 4.2.2. Подсчет SR на основе $CF(n, m)$

При сравнении линейных кодов мы считаем вероятность успешной передачи сообщения через канал связи с потерями. К сожалению, чтобы в точности посчитать такую вероятность, требуется проверить все возможные комбинации потерь среди $n + m$ символов, то есть $2^{n+m}$ экспериментов по передаче сообщения. Для каждого линейного кода подсчет такой вероятности занял бы слишком много времени и памяти, поэтому для ее оценки используется метрика $SR[CF(n, m)]$, которая определяется следующим образом:

$$SR[CF(n, m)] = \frac{K}{N}, \tag{3}$$

где:

$N$ – общее количество экспериментов, на которых тестировался линейный блоковый код (для тестирования используется компактное представление $CF(n, m)$),

$K$ – количество экспериментов, при которых удалось успешно восстановить все символы исходного сообщения.

Алгоритм подсчета данной метрики организован следующим образом:

1. Выбирается модель потерь, для которой будем считать метрику.
2. При помощи модели потерь генерируется набор из $N$ экспериментов. Эксперимент — набор длины $n + m$, состоящий из 0 и 1.
3. Эксперимент применяется к $CF(n, m)$, в результате чего теряется часть передаваемой информации — некоторые элементы зануляются.
4. К полученной информации применяется алгоритм восстановления исходного сообщения. Если все символы исходного сообщения удалось восстановить, то эксперимент считается успешным.
5. Пункты 3-4 применяются ко всем сгенерированным экспериментам, в итоге получаем $K$ — количество успешных экспериментов.

Алгоритмы восстановления исходного сообщения подробно описаны в *главе 5*.

### 4.2.3. Штраф за повторяющиеся действия

Из *уравнения 3* видно, что $SR[CF(n, m)] \in [0; 1]$. При обучении агента мы хотим добиться того, чтобы в процессе изменения *состояния* $S$ как можно реже возникала ситуация, описанная в *пункте 3*.

Для этого введем штраф: $-1$ за каждое действие, уже выбранное ранее в этой сессии. Тогда итоговый штраф:

$$P(\tau) = -1 \cdot k, \tag{4}$$

где:

$k$ – количество раз, когда агент совершил повторяющееся действие в траектории $\tau$.

В результате этого, если агент хотя бы один раз в своей стратегии выберет действие, которое уже использовал до этого, то значение полученной за сессию *функции награды* $R(\tau)$ будет $\leq 0$.

Проверить то, что агент совершил выбранное ранее действие очень просто — для этого можно использовать состояние $S$ и *полученные для него выводы*:

1. Пусть агент совершил действие $a$ и перешел из состояния $S$ в состояние $S'$.
2. Сравним состояния $S$ и $S'$. Эти состояния будут одинаковыми, если агент в результате действия $a$ выбрал позицию для единицы, которую уже выбирал до этого в текущей сессии.

Использование таких штрафов оказалось очень эффективно при обучении агента. Более подробно их влияние можно увидеть на графиках, представленных в *разделе* 6.3.

### 4.2.4. Функция награды $r(s, a, s')$

При обучении агент получает награду за каждый шаг, поэтому было принято решение использовать следующую функцию награды:

$$r(s,a,s') = \begin{cases} 0, & \text{количество различных выполненных действий меньше } m \text{ и } s \neq s', \\ -1, & \text{количество различных выполненных действий меньше } m \text{ и } s = s', \\ SR[CF(n,m)], & \text{количество различных выполненных действий равно } m. \end{cases}$$

Суммарная функция награды $R(\tau)$ связана с $r(s, a, s')$ следующим уравнением:

$$R(\tau) = \sum_{t=0}^{T} r(s_t, a_t, s_{t+1}).$$

В процессе обучения агента задача состоит в том, чтобы максимизировать значение $R(\tau)$.

## 4.3. Агент

В рамках исследования тестировались различные алгоритмы обучения с подкреплением, такие как DQN, DoubleDQN, SAC, PPO и A2C. Был выбран DoubleDQN, так

как в процессе обучения он показал стабильные результаты и высокую скорость обучения в связи с хорошо подобранными гиперпараметрами. Более детальное изучение алгоритмов и подбор оптимальных гиперпараметров для них мы оставили для дальнейшего исследования. В этой главе описываются детали реализации архитектуры агента, который был использован в данной работе.

### 4.3.1. Сессия взаимодействия со средой

Одна сессия взаимодействия по построению $CF(n, m)$ устроена следующим образом:

1. Сессия начинается с изначального *состояния* $\mathcal{S}_0$.
2. Агент выполняет действие согласно своей стратегии.
3. Если агент совершил повторяющееся действие — увеличивается *штраф* $P(\tau)$ (4).
4. Когда агент совершил $m$ различных действий — тестируется эффективность полученного линейного кода и агент получает *награду* $SR[CF(n, m)]$ (3). Такое состояние $\mathcal{S}_t$ считается терминальным и на этом сессия заканчивается (здесь $t$ — номер текущего шага сессии).
5. Итоговая награда за сессию считается по *формуле* 2.

Важно отметить, что агент хранит информацию о всех промежуточных состояниях, которые были получены во время сессии, и использует ее в процессе обучения.

### 4.3.2. Архитектура нейронной сети

В DoubleDQN используется две нейронные сети - основная и целевая. В рамках исследования было принято решение выбрать для них одинаковую архитектуру. Каждая нейронная сеть состоит из следующих слоев:

- Входной слой размера $2^n$. На вход нейронной сети подается состояние $S$, используемое для обучения.
- Далее идет 2 внутренних полносвязных слоя по 64 нейрона.
- Выходной слой размера $2^n$.

### 4.3.3. Гиперпараметры агента

Все гиперпараметры агента можно настраивать на старте обучения. Здесь описаны значения, с которыми агент инициализируется.

**Exploration rate (ER).** Данный коэффициент отвечает за вероятность выбора случайного действия. Начальное значение — 1, минимальное значение — 0.01. В конце каждого эпизода ER домножается на уменьшающий коэффициент — decay. От данного коэффициента зависит скорость обучения агента. Значение по умолчанию — 0.9995.

**Коэффициент обновления весов целевой нейронной сети.** Каждые несколько эпизодов веса целевой сети обновляются при помощи весов основной сети в соответствии со следующим уравнением:

$$\theta' = \mu \cdot \theta + (1 - \mu) \cdot \theta',$$

где:

$\theta'$ – веса целевой сети,
$\theta$ – веса основной сети,
$\mu$ – коэффициент обновления.

Значение по умолчанию $\mu = 10^{-3}$.

**Коэффициент дисконтирования.** Этот коэффициент позволяет ускорить сходимость алгоритма, так как приоритизируются краткосрочные вознаграждения (в процессе обучения мы считаем суммарную дисконтированную награду $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$). Значение по умолчанию $\gamma = 0.99$.

# 5. Алгоритм для тестирования передачи сообщения

В данной главе рассматривается общая формулировка алгоритма, а также конкретная реализация, которая была использована в рамках данного исследования.
Мы рассматриваем процесс передачи информации информации по каналу с потерями. Формально его можно описать так: у нас есть исходное сообщение, которое мы хотим передать:

$$m = (m_1, m_2, \ldots, m_n).$$

Используя порождающую матрицу линейного кода $G$ мы формируем закодированное сообщение

$$c = (c_1, c_2, \ldots, c_{n+m}).$$

и передаем его по каналу с заданной моделью потерь. Каждое $c_i$ можно построить с помощью уравнения:

$$c_i = a_{1i}m_1 \oplus a_{2i}m_2 \oplus \cdots \oplus a_{ni}m_n,$$

где $a_i = (a_{1i}, \ldots, a_{ni})$ — это $i$-й столбец порождающей матрицы $G$. В процессе передачи часть информации теряется и мы получаем $c' = (c_{i_1}, c_{i_2}, \ldots, c_{i_k}),\ 1 \leq i_1 < i_2 < \cdots < i_k \leq n+m,\ k \leq n+m$.

Задача алгоритма состоит в следующем: нужно проверить, можно ли на основе $c'$ получить все исходное сообщение $m$. При этом про каждый $c'_i$ мы знаем, суммой каких компонент $m$ он является. Поэтому на самом деле мы можем составить систему уравнений вида $a_{i1}m_1 \oplus a_{i2}m_2 \oplus \cdots \oplus a_{in}m_n = c'_i$ и решить ее относительно $m_i$. Однако для расчета эффективности способа кодирования нам важно только проверить, что решение системы существует — тогда мы считаем процесс восстановления успешным, само решение находить не нужно.

Рассмотрим данный алгоритм на примере. Пусть у нас есть сообщение $m$ и порождающая матрица $G$ следующего вида:

$$m = (6, 3, 5), \qquad G = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Тогда закодированная информация $c$ получается следующим образом:

$$\begin{cases} 1 \cdot m_1 \oplus 0 \cdot m_2 \oplus 0 \cdot m_3 = c_1, \\ 0 \cdot m_1 \oplus 1 \cdot m_2 \oplus 0 \cdot m_3 = c_2, \\ 0 \cdot m_1 \oplus 0 \cdot m_2 \oplus 1 \cdot m_3 = c_3, \\ 1 \cdot m_1 \oplus 1 \cdot m_2 \oplus 1 \cdot m_3 = c_4, \\ 0 \cdot m_1 \oplus 1 \cdot m_2 \oplus 1 \cdot m_3 = c_5, \\ 1 \cdot m_1 \oplus 0 \cdot m_2 \oplus 1 \cdot m_3 = c_6 \end{cases} \Rightarrow \begin{cases} m_1 = c_1, \\ m_2 = c_2, \\ m_3 = c_3, \\ m_1 \oplus m_2 \oplus m_3 = c_4, \\ m_2 \oplus m_3 = c_5, \\ m_1 \oplus m_3 = c_6 \end{cases} \Rightarrow \quad c = (6, 3, 5, 0, 6, 3).$$

Пусть в процессе передачи по каналу мы потеряли $c_1, c_2, c_3$ и мы получили только $c' = (0, 6, 3)$. Отсюда мы можем перейти к следующей системе уравнений:

$$\begin{cases} m_1 \oplus m_2 \oplus m_3 = 0, \\ m_2 \oplus m_3 = 6, \\ m_1 \oplus m_3 = 3. \end{cases} \tag{5}$$

Решив эту систему, мы сможем восстановить $m$:

$$\begin{cases} m_1 \oplus m_2 \oplus m_3 = 0, \quad (1) \\ m_2 \oplus m_3 = 6, \quad (2) \\ m_1 \oplus m_3 = 3 \quad (3) \end{cases} \xRightarrow{(1)\oplus(2)} \begin{cases} m_1 = 6, \quad (1) \\ m_2 \oplus m_3 = 6, \quad (2) \\ m_1 \oplus m_3 = 3 \quad (3) \end{cases} \xRightarrow{(1)\oplus(3)}$$

$$\begin{cases} m_1 = 6, \quad (1) \\ m_2 \oplus m_3 = 6, \quad (2) \\ m_3 = 5 \quad (3) \end{cases} \xRightarrow{(2)\oplus(3)} \begin{cases} m_1 = 6, \\ m_2 = 3, \\ m_3 = 5. \end{cases}$$

Таким образом, эксперимент считается успешным, если нам удалось восстановить все переданные символы $m_i$ изначального сообщения $m$.

В этой секции мы рассмотрели принцип работы алгоритма тестирования передачи сообщения и разобрали пример с успешным восстановлением исходного сообщения. Далее рассмотрим конкретную реализацию, которая использовалась в рамках программного обеспечения RL-FEC.

## 5.1. Реализация алгоритма при помощи подсчета ранга матрицы

Данная идея заключается в том, что мы составляем матрицу полученной системы уравнений и считаем ее ранг в $\mathbb{F}_2$. Если ранг матрицы будет равен $n$ — количеству символов в исходном сообщении, то мы сможем восстановить все исходное сообще-

ние. Это так, потому что если ранг матрицы равен $n$, то значит она приводится к единичной матрице $E$, из которой однозначно можно будет получить все $m_i$.

В качестве примера рассмотрим уже использованную ранее *систему уравнений 5*. Для нее матрица системы будет выглядеть так:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

Ранг данной матрицы в $\mathbb{F}_2$ равен $\text{rank}_{\mathbb{F}_2}(A) = 3$, и как и было показано, восстановить из этой системы все исходное сообщение возможно.

Для конкретного эксперимента можно быстро получить матрицу системы из $CF(n, m)$:

- Получим порождающую матрицу $G$ из компактного представления $CF(n, m)$.
- Рассмотрим $G^T$. Все строки этой матрицы как раз задают полную систему уравнений для $c_i$.
- Эксперимент задает нам определенную подматрицу $A$ — во время передачи мы теряем какие-то строки $G^T$.
- Для подматрицы $A$ считаем ранг в $\mathbb{F}_2$.
- Если ранг матрицы равен $n$, то сможем восстановить исходное сообщение.

У данного алгоритма есть существенный недостаток — фиксированный размер матриц, зависящий от $n$ и $m$. Поэтому при обучении агента для одного случая приходится хранить в памяти компьютера большое количество данных, что делает практически невозможным запуск нескольких экспериментов одновременно.

Также в рамках тестирования данной реализации алгоритма было выявлено, что он позволяет эффективно проверять $\approx 10^5$ экспериментов на одном $CF(n, m)$. Однако в рамках нашего исследования нам нужно проверять $\approx 10^8$ экспериментов, чтобы добиться желаемой точности оценки SR, поэтому была реализована другая версия алгоритма.

## 5.2. Реализация алгоритма при помощи покрывающих множеств

Данный алгоритм основывается на теоретическом утверждении, полученном в статье [7]. Идея состоит в следующем:

1. Рассматриваем линейный блоковый код $C$, который преобразует сообщения длины $n$ в закодированные сообщения длины $n + m$.

2. Введем следующие множества:

   - Множество позиций в закодированном сообщении — $N = \{1, 2, \ldots, n+m\}$.
   - Множество позиций, символы на которых были потеряны в процессе передачи — $I = \{i_1, i_2, \ldots, i_t\}$, где $t$ — количество потерянных символов.

3. Для каждого закодированного сообщения $c = (c_1, \ldots, c_{n+m}),\ c_i \in \mathbb{F}_2$ введем множество $\mathrm{Supp}(c) = \{i | c_i = 1\}$.

4. Будем называть $I$ **покрывающим множеством** тогда и только тогда, когда $\mathrm{Supp}(c) \subseteq I$ хотя бы для какого-то закодированного сообщения $c$, полученного при помощи линейного кода $C$.

5. Следующие утверждения *эквивалентны*:

   - $I$ — покрывающее множество,
   - Ранг подматрицы $A$, которую мы использовали в *пункте* 5.1, меньше $n$.

Реализация алгоритма основывается на использовании первого утверждения. Алгоритм работает следующим образом:

1. Рассматриваем линейный блоковый код, используем представление $CF(n, m)$.
2. Перебираем всевозможные исходные сообщения $m = (m_1, m_2, \ldots, m_n) \in \mathbb{F}_2$.
3. Реализуем операцию кодирования: $m \times CF(n, m) = mG$ (операция выполняется в $\mathbb{F}_2$).
4. При помощи модели потерь генерируем маску потерь *Err*, где 1 означает, что символ был потерян в процессе передачи.
5. Отсюда получаем следующую проверку — если $\exists m \ : \ \mathrm{Supp}(m \times CF(n, m)) \subseteq \mathrm{Supp}(Err)$, тогда $\mathrm{Supp}(Err)$ — покрывающее множество и восстановить сообщение невозможно.

Данную проверку реализуем для различных масок потерь *Err*. Преимущество данного алгоритма состоит в том, что в рамках нашего исследования $n+m \leq 64$, что позволяет использовать тип данных $int64$ для хранения $\mathrm{Supp}(m \times CF(n, m))$ и $\mathrm{Supp}(Err)$. Такая реализация также позволяет использовать алгоритм для нескольких экспериментов одновременно — загрузим в память компьютера данные для случая $n + m = 64$, а внутренние механизмы работы с типами данных позволят использовать эти данные $\forall n, m \ : \ n + m \leq 64$.

В рамках программного обеспечения RL-FEC алгоритм был реализован как отдельный микросервис, который отдает запросы нескольким экспериментам одновременно. Это позволило добиться необходимой точности, сохранить высокую скорость

обработки запросов, а также значительно уменьшить ресурсы, необходимые для работы алгоритма.

# 6. Эксперименты

В данной главе рассматриваются эксперименты, которые были поставлены в рамках тестирования программного обеспечения RL-FEC. Были проведены симуляции двух моделей потерь: Binary Erasure Channel (BEC) и простая модель Гильберта. Для каждой из моделей был проведен ряд экспериментов и получены линейные блоковые коды над полем $\mathbb{F}_2$. Далее был проведен сравнительный анализ с тремя подходами избыточного кодирования: коды Рида-Соломона (RS), LDPC-коды и алгоритм FlexFEC. Ключевым критерием для сравнения является метрика SR, но помимо этого также приводится сравнение с точки зрения количества операций, необходимых для кодирования и декодирования.

## 6.1. BEC

Для данной модели потерь параметры экспериментов были выбраны следующим образом:

1. Фиксируем вероятность потери $p_{\text{loss}}$. В рамках исследования мы рассматривали $p_{\text{loss}} \in \{0.1,\ 0.2,\ 0.3,\ 0.4,\ 0.5\}$.

2. Выбираем исходную длину сообщения $n$. Были рассмотрены $n \in [2; 8]$ в зависимости от вероятности $p_{\text{loss}}$.

3. Подбираем $m$ — количество избыточных символов. Для этого считаем вероятность восстановления сообщения исходного сообщения. Это будет возможно, если в процессе передачи мы потеряли $\leq m$ символов из $n+m$ переданных. Такая вероятность считается по следующей формуле:

$$P_m = \sum_{t=1}^{m} \binom{n+m}{t} \cdot p_{\text{loss}}^t \cdot (1 - p_{\text{loss}})^{n+m-t}.$$

4. Выбираем минимальное $m$, при котором $P_m \geq 0.999$.

Важно отметить, что в рамках исследования мы рассматриваем линейный блоковые коды без повторяющихся столбцов в порождающей матрице — это означает, что количество различных действий должно быть $\geq m$. В *главе* 4 мы выяснили, что количество различных действий — $2^n - 1$, поэтому получаем ограничение на параметры для экспериментов — $2^n - 1 \geq m$.

В итоге были получены результаты, представленные на *рисунке* 7. Из графиков видно, что полученные линейные блоковые коды с точки зрения эффективности передачи превосходят FlexFEC и приближены к кодам Рида-Соломона, которые считаются самыми эффективными.

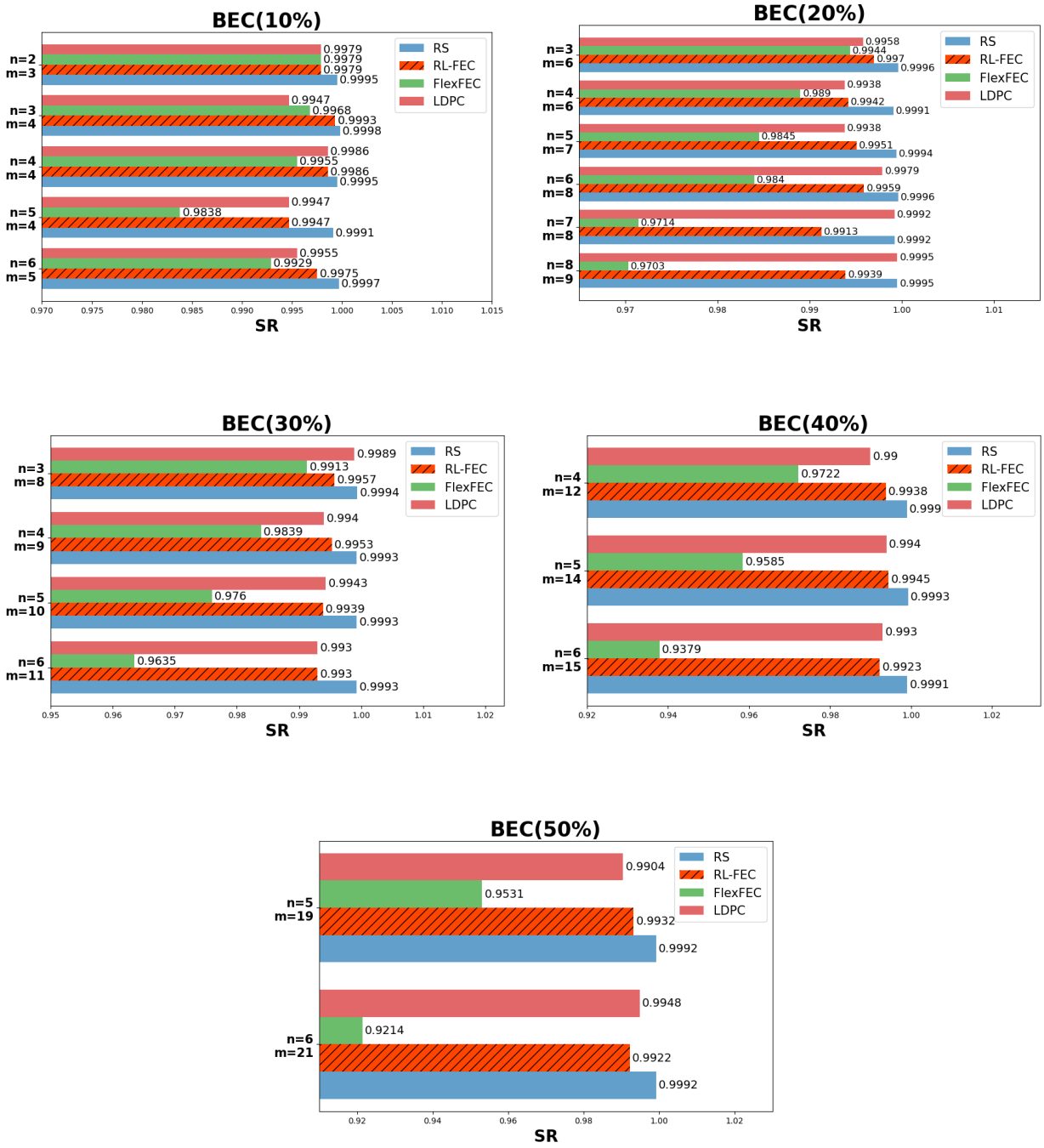



Рис. 7: Сравнение FlexFEC, кодов Рида-Соломона (RS) и RL-FEC на модели BEC.

## 6.2. Простая модель Гильберта

Параметры для модели были получены на основе параметров для модели BEC. Для данной модели можно получить стационарные вероятности:

$$\pi_G = \frac{q}{p+q}, \quad \pi_B = \frac{p}{p+q}.$$

Также можно вычислить матожидание продолжительности *burst* (количество подряд потерянных символов):

$$\mathbb{E}(|burst|) = \frac{1}{q}.$$

На основе этих данных параметры были получены следующим образом:

1. Сохраним $n$ и $m$, полученные на модели BEC для вероятности $p_{\text{loss}}$.
2. Зафиксируем $\pi_B = p_{\text{loss}}$.
3. Смоделируем $\mathbb{E}(|burst|) \approx \frac{n+m}{3}$.
4. При помощи $\pi_B$ и $\mathbb{E}(|burst|)$ выразим параметры $p$ и $q$ для данной модели.

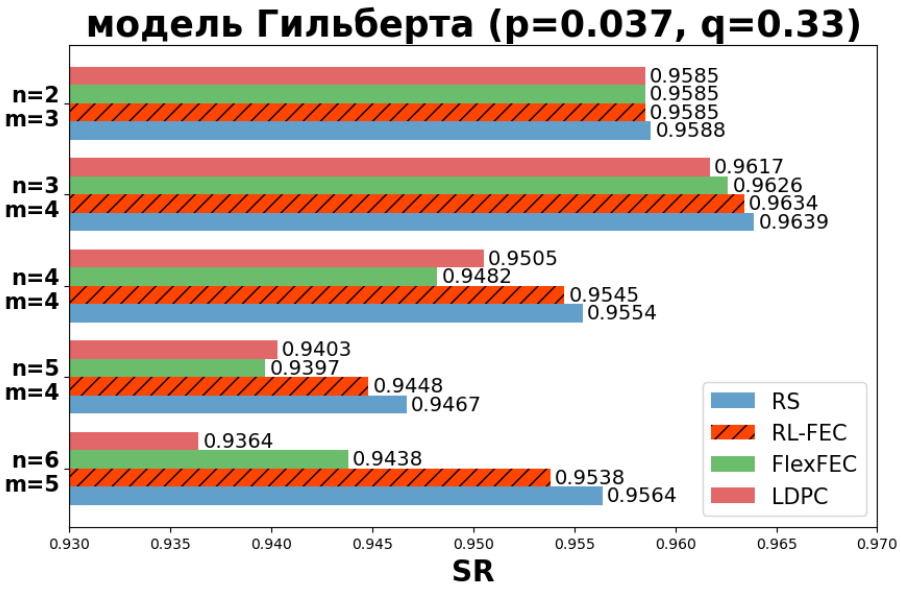
В итоге были получены следующие модели потерь:

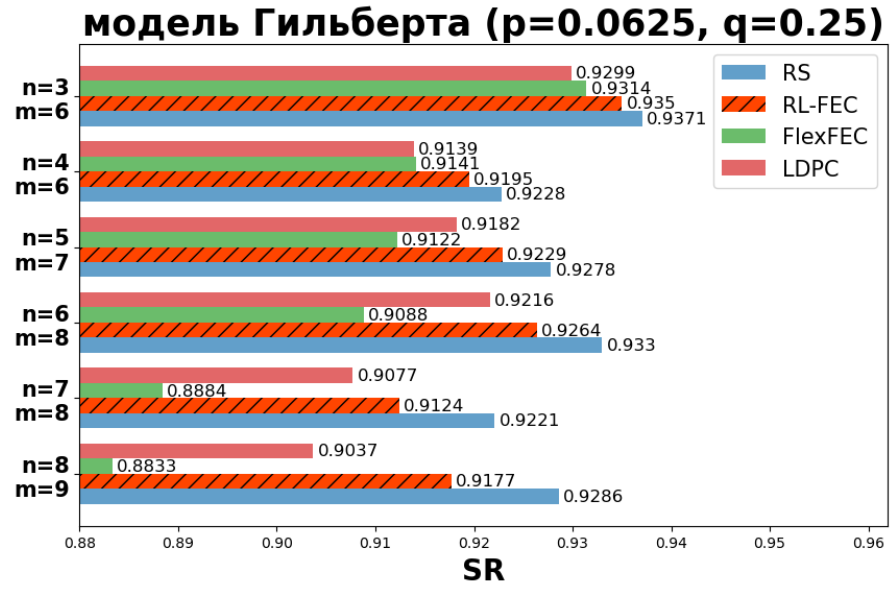
$$\text{BEC}(p_{\text{loss}} = 0.1),\ \mathbb{E}(|burst|) = 3 \Rightarrow \text{Gilbert}(p = 0.037, q = 0.33);$$

$$\text{BEC}(p_{\text{loss}} = 0.2),\ \mathbb{E}(|burst|) = 4 \Rightarrow \text{Gilbert}(p = 0.0625, q = 0.25);$$

$$\text{BEC}(p_{\text{loss}} = 0.3),\ \mathbb{E}(|burst|) = 5 \Rightarrow \text{Gilbert}(p = 0.085, q = 0.2);$$

Для данных моделей были получены результаты, представленные на *рисунке* 8.



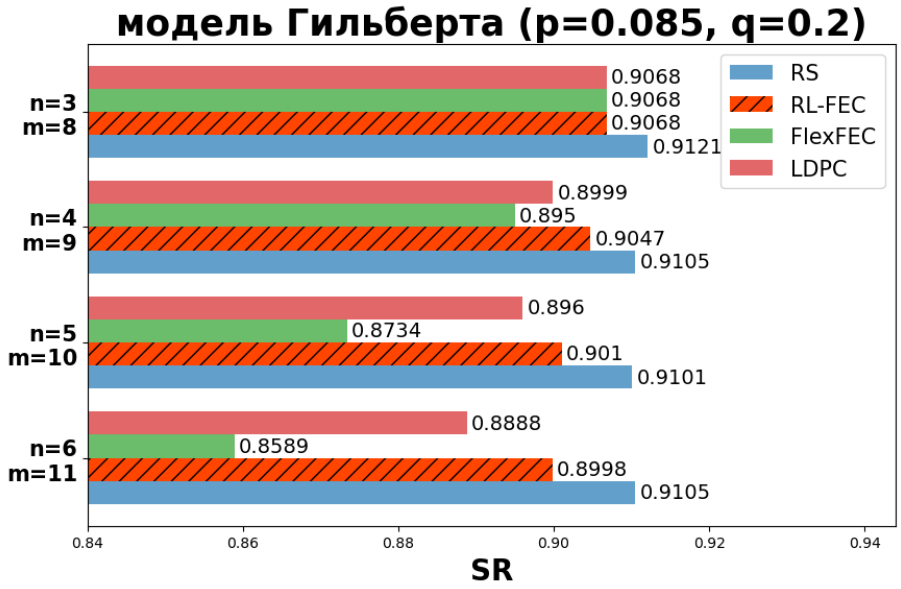




Рис. 8: Сравнение FlexFEC, кодов Рида-Соломона (RS) и RL-FEC на простой модели Гильберта.

## 6.3. Исследование процесса обучения

При использовании обучения с подкреплением нам было важно понять эффективность алгоритма по двум параметрам:

1. скорость обучения агента,
2. постоянство в процессе обучения — как часто сходится алгоритм и сколько итераций для этого требуется.

Для исследования данных параметров в процессе обучения были собраны следующие данные для каждой проведенной сессии агента:

- средняя награда за сессию — позволяет увидеть, что агент максимизирует награду,
- средняя продолжительность сессии — позволяет увидеть, что агент перестает совершать повторяющиеся действия в процессе обучения,
- среднее значение функции потерь — позволяет следить за процессом обучения нейронной сети.

На основе этих данных были построены графики. Для доказательства эффективности алгоритма один и тот же эксперимент был проведен несколько раз и построены доверительные интервалы.

В качестве примера рассмотрим следующий эксперимент:

- модель потерь — BEC с вероятностью потери $p = 0.4$,
- исходное сообщение состоит из $n = 5$ символов, для кодирования используется $m = 14$ избыточных символов,
- в качестве агента был выбран алгоритм DoubleDQN.

Для каждой величины были построены доверительные интервалы с уровнем доверия $\beta = 0.95$. В результате были получены графики, представленные на *рисунке* 9.

Из графиков можно видеть, что разброс значений достаточно небольшой, что показывает постоянство в процессе обучения. Также из графика средней награды можно сделать вывод о скорости обучения агента — потребовалось $\approx 4000$ сессий, чтобы награда стала $\geq 0$ (одна сессия длится $\approx 1$ сек.).



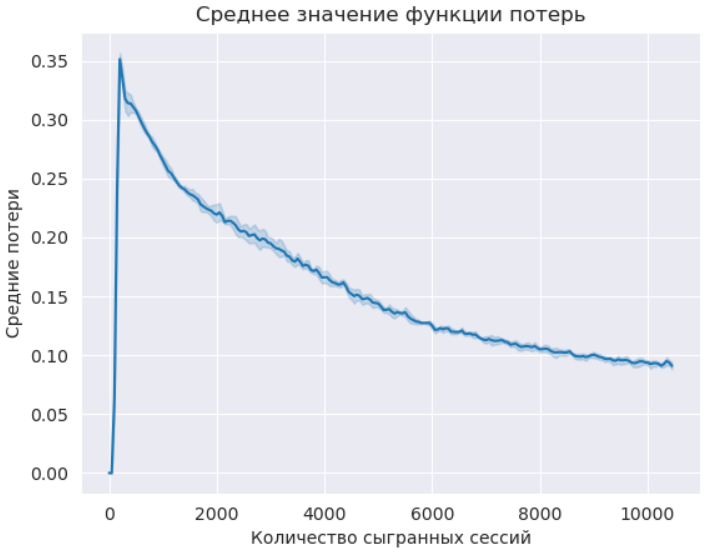



Рис. 9: Обучение агента на $n = 5$, $m = 14$, модель BEC($p = 0.4$).

# Заключение

Главным результатом работы является создание программного обеспечения RL-FEC для построения линейных блоковых кодов при помощи обучения с подкреплением. RL-FEC позволяет настраивать и моделировать множество параметров, включая различные модели потерь данных, параметры линейных кодов, а также метрики, под которые оптимизируются полученные коды.

Программное обеспечение RL-FEC было протестировано на двух моделях потерь: Binary Erasure Channel (BEC) и простая модель Гильберта. Оптимизация кодов проводилась на основе метрики SR. После этого был проведен сравнительный анализ полученных при помощи RL-FEC линейных кодов с кодами Рида-Соломона, LDPC-кодами и алгоритмом FlexFEC. Найденные линейные блоковые коды сравнимы по показателю SR с лучшими представителями других методов, что показывает возможности RL-FEC по оптимизации заданного функционала.

Стоит отметить, что не удалось получить самые оптимальные линейные блоковые коды во всех проведенных экспериментах. Это показывает то, что программное обеспечение RL-FEC требует улучшений и аппробации новых подходов, на что и нацелены дальнейшие исследования.

## Дальнейшие планы

Данная работа представляет одну из первых попыток применения обучения с подкреплением для построения линейных блоковых кодов. В рамках исследования было выдвинуто несколько направлений для дальнейшего изучения.

В первую очередь планируется исследовать другие варианты представления линейных кодов и использовать их для обучения агента. Выделяется два направления исследования:

1. Использование сверточных слоев нейронной сети для получения информации о линейном блоковом коде из его представления в виде порождающей матрицы.

2. Использованние предобученных эмбеддингов для графов (structure2vec [4], BiGI [2]) для получения информации из двудольного графа $B(n, m)$.

Также планируется продолжить исследование других алгоритмов обучения с подкреплением, в частности SAC и PPO. Одним из частых случаев, который мы планируем использовать, является следующая конфигурация среды:

- большое количество пакетов в процессе передачи — $n + m > 30$,

- высокая вероятность потери пакета для модели BEC — $p_{\text{loss}} > 0.5$.

При использовании алгоритма DoubleDQN и состояния среды $\mathcal{S}$ было замечено, что агенту не хватает информации для исследования среды и нахождения самых эффективных (с точки зрения оценки SR) линейных блоковых кодов. Даже путем рандомного заполнения порождающей матрицы были найдены линейные коды, SR для которых оказалась выше, чем для кодов, найденных при помощи RL-FEC. Использование других алгоритмов и состояний среды, в теории, поможет решить данную проблему.

Помимо этого планируется добавить в RL-FEC поддержку следующих модель потерь:

1. модель Гильберта-Эллиота,
2. цепь Маркова из четырех состояний, представленная в статье [10].

Дальнейшие исследования в данных направлениях помогут повысить эффективность программного обеспечения RL-FEC и сделать его более универсальным.

# Список литературы


[1] Atmaca Abdullah and Oruç A. Yavuz. Counting Unlabeled Bipartite Graphs Using Polya's Theorem // Bulletin of the Belgian Mathematical Society - Simon Stevin. — 2018. — Vol. 25, no. 5. — P. 699 – 715.

[2] Cao Jiangxia, Lin Xixun, Guo Shu, Liu Luchen, Liu Tingwen, and Wang Bin. Bipartite Graph Embedding via Mutual Information Maximization. — 2020. — arXiv: 2012.05442.

[3] Lillicrap Timothy P., Hunt Jonathan J., Pritzel Alexander, Heess Nicolas, Erez Tom, Tassa Yuval, Silver David, and Wierstra Daan. Continuous control with deep reinforcement learning. — 2019. — arXiv: 1509.02971.

[4] Dai Hanjun, Dai Bo, and Song Le. Discriminative Embeddings of Latent Variable Models for Structured Data. — 2020. — arXiv: 1603.05629.

[5] Darvariu Victor-Alexandru, Hailes Stephen, and Musolesi Mirco. Goal-directed graph construction using reinforcement learning // Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences. — 2021. — oct. — Vol. 477, no. 2254. — P. 20210168.

[6] Fawzi Alhussein, Balog Matej, Huang Aja, Hubert Thomas, Romera-Paredes Bernardino, Barekatain Mohammadamin, Novikov Alexander, Ruiz Francisco, Schrittwieser Julian, Swirszcz Grzegorz, Silver David, Hassabis Demis, and Kohli Pushmeet. Discovering faster matrix multiplication algorithms with reinforcement learning // Nature. — 2022. — 10. — Vol. 610. — P. 47–53.

[7] Dumer Ilya and Farrell Patrick. Erasure Correction Performance of Linear Block Codes. // Lecture Notes in Computer Science. — 1994. — 06. — Vol. 781. — P. 316–326.

[8] Frnda Jaroslav, Vozňák Miroslav, and Sevcik Lukas. Impact of packet loss and delay variation on the quality of real-time video streaming // Telecommunication Systems. — 2015. — 04. — Vol. 62.

[9] Li Yuxi. Reinforcement Learning Applications. — 2019. — arXiv: 1908.06973.

[10] Ludovici F and Ordine A. Definition of a general and intuitive loss model for packet networks and its implementation in the Linux kernel Netem. — 2012. — 01. — Access mode: `http://netgroup.uniroma2.it/Stefano_Salsano/papers/TR-loss-netem.pdf`.

[11] Nielsen Jimmy J., Leyva-Mayorga Israel, and Popovski Petar. Reliability and Error Burst Length Analysis of Wireless Multi-Connectivity // 2019 16th International Symposium on Wireless Communication Systems (ISWCS). — 2019. — Aug. — P. 107–111.

[12] Mnih Volodymyr, Kavukcuoglu Koray, Silver David, Graves Alex, Antonoglou Ioannis, Wierstra Daan, and Riedmiller Martin. Playing Atari with Deep Reinforcement Learning. — 2013. — arXiv: 1312.5602.

[13] Schulman John, Wolski Filip, Dhariwal Prafulla, Radford Alec, and Klimov Oleg. Proximal Policy Optimization Algorithms. — 2017. — arXiv: 1707.06347.

[14] Langley Adam, Riddoch Alistair, Wilk Alyssa, Vicente Antonio, Krasic Charles, Zhang Dan, Yang Fan, Kouranov Fedor, Swett Ian, Iyengar Janardhan, Bailey Jeff, Dorfman Jeremy, Roskind Jim, Kulik Joanna, Westin Patrik, Tenneti Raman, Shade Robbie, Hamilton Ryan, Vasiliev Victor, Chang Wan-Teh, and Shi Zhongyi. The QUIC Transport Protocol: Design and Internet-Scale Deployment // Proceedings of the Conference of the ACM Special Interest Group on Data Communication. — New York, NY, USA : Association for Computing Machinery. — 2017. — SIGCOMM '17. — P. 183–196. — Access mode: https://doi.org/10.1145/3098822.3098842.

[15] Zanaty Mo, Singh Varun, Begen Ali C., and Mandyam Giridhar. RTP Payload Format for Flexible Forward Error Correction (FEC). — RFC 8627. — 2019. — July. — Access mode: https://www.rfc-editor.org/info/rfc8627.

[16] Reed I. S. and Solomon G. Polynomial Codes Over Certain Finite Fields // Journal of the Society for Industrial and Applied Mathematics. — 1960. — Vol. 8, no. 2. — P. 300–304. — https://doi.org/10.1137/0108018.

[17] Shokrollahi Amin. LDPC Codes: an Introduction. — 2004. — 01. — P. 85–110. — ISBN: 978-3-0348-9602-3.

[18] Haarnoja Tuomas, Zhou Aurick, Abbeel Pieter, and Levine Sergey. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. — 2018. — arXiv: 1801.01290.

[19] Sutton R.S. and Barto A.G. Reinforcement Learning: An Introduction // IEEE Transactions on Neural Networks. — 1998. — Vol. 9, no. 5. — P. 1054–1054.

[20] Garrido Pablo, Sanchez Isabel, Ferlin Simone, Aguero Ramon, and Alay Ozgu. rQUIC: Integrating FEC with QUIC for Robust Wireless Communications // 2019 IEEE Global Communications Conference (GLOBECOM). — 2019. — P. 1–7.

[21] van Hasselt Hado, Guez Arthur, and Silver David. Deep Reinforcement Learning with Double Q-learning. — 2015. — arXiv: 1509.06461.