ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт компьютерных наук и кибербезопасности

Отчет о прохождении учебной (научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)) практики

Киричек Мария Романовна

(Ф.И.О. обучающегося)

1 курс, 5130203/40001

(номер курса обучения и учебной группы)

02.03.03 Математическое обеспечение и администрирование информационных систем

(направление подготовки (код и наименование))

Место прохождения практики: ФГАОУ ВО «СПбПУ», ИКНиК, ВШТИИ,

(указывается наименование профильной организации или наименование структурного подразделения

г. Санкт-Петербург, ул. Обручевых, д. 1, лит. В

ФГАОУ ВО «СПбПУ», фактический адрес)

Сроки практики: с 20.06.2025 по 17.07.2025

Руководитель практической подготовки от ФГАОУ ВО «СПбПУ»: Пак Вадим Геннадьевич, к.ф.-м.н., доцент ВШТИИ

(*Ф.И.О., уч. степень, должность)*

Консультант практической подготовки от ФГАОУ ВО «СПбПУ»: нет

(*Ф.И.О., уч. степень, должность)*

Руководитель практической подготовки от профильной организации: нет

Оценка:

Руководитель практической подготовки
от ФГАОУ ВО «СПбПУ»: /Пак В.Г./

Обучающийся: /Киричек М. Р./

Дата: 17.07.25

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методы градиентного спуска лежат в основе обучения искуственных нейронных сетей (ИНС, далее - нейросетей), позволяя минимизировать функцию потерь и находить оптимальные параметры моделей. Их адаптивные варианты (SGD, Adam, AdaBoost и др.) критически важны для повышения эффективности и точности обучения в условиях больших данных.

Актуальность исследования — градиентные методы (Adam, SGD, AdaBoost и др.) — основа современного машинного обучения. Их оптимизация критически важна для ускорения обучения ИНС и повышения точности моделей, особенно при работе с большими данными. Исследование этих методов позволяет создавать более эффективные алгоритмы для задач ИИ.

Объект исследования — методы градиентного спуска (SGD, Adam) и градиентный бустинг (AdaBoost).

Предмет исследования — математические и алгоритмические особенности, эффективность.

Цель исследования — провести сравнительный анализ эффективности методов (SGD, Adam, AdaBoost и др.) при обучении ИНС. В соответствии с целью исследования, определяются следующие **задачи работы**:

1. Изучить литературу по методам градиентного спуска.
2. Проанализировать математические основы Adam, SGD и AdaBoost.
3. Разработать критерии сравнения эффективности.
4. Реализовать методы с использованием PyTorch.
5. Провести эксперимент и анализ результатов.

1 ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Изучение литературы по истории, современным достижениям и актуальным задачам методов оптимизации градиентного спуска

Методы градиентного спуска являются основой обучения нейросетей [7]. Их развитие прошло три ключевых этапа. Изначально метод был теоретически обоснован Огюстеном Коши в XIX веке, но его применение ограничивалось малыми задачами из-за необходимости полного расчёта градиента[6]. С середины XX века началась стохастическая революция: Роббинс предложил SGD (1951), Поляк ввёл momentum, Нестеров разработал ускоренную сходимость. В 1995 году Фройнд предложил AdaBoost — адаптивный бустинг, повлиявший на подходы к оптимизации.

С ростом глубокого обучения появились адаптивные методы, в первую очередь Adam (2014), который автоматически регулирует шаг обучения для каждого параметра. Поздние модификации (AdamW, AMSGrad) устранили проблемы со сходимостью и стабильностью.[1]

Современные алгоритмы существенно ускоряют обучение. Adam позволяет на 15–30% быстрее достигать нужной точности. Для распределённых систем, таких как федеративное обучение, применяются методы вроде FedAvg, обеспечивающие обучение без передачи персональных данных. В условиях ограниченных вычислительных ресурсов (например, на устройствах IoT) используются методы сжатия градиентов, такие как Deep Gradient Compression, сокращающие объём передаваемой информации до 0.1%. Автоматизация подбора гиперпараметров с помощью инструментов вроде Optuna позволяет значительно ускорить настройку моделей.

Актуальные задачи включают повышение устойчивости оптимизаторов на сложных функциях потерь, эффективную работу в распределённых и энергоограниченных системах, создание «лёгких» версий алгоритмов (например, TinyAdam). Ведутся исследования по объединению сильных сторон различных подходов (гибридные методы), интеграции с квантовыми вычислениями и адаптации алгоритмов под новые типы аппаратуры, включая нейроморфные чипы.

Таким образом, современная оптимизация стала более гибкой, автоматизированной и масштабируемой, а дальнейшее развитие связано с адаптацией к новым вычислительным условиям и аппаратным платформам.

1.2 Выводы

В первой главе описано, как методы градиентного спуска стали адаптивными, быстрыми и удобными в применении. Они успешно работают в распределённых и ограниченных по ресурсам системах. Дальнейшее развитие связано с автоматизацией, гибридными подходами и поддержкой новых вычислительных платформ.

2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

2.1 Изучение математических основ методов градиентного спуска

Данная глава посвящена анализу математических основ ключевых алгоритмов оптимизации. В разделе исследуются фундаментальные методы: классический градиентный спуск, современные оптимизаторы (SGD, Adam) и градиентный бустинг. Раздел 2.1.4 содержит сравнительный анализ их преимуществ и ограничений.

2.1.1 Градиентный спуск

Градиентный спуск — итерационный метод оптимизации функций, широко используемый в машинном обучении [5]. Основная идея заключается в движении в направлении *антиградиента* функции потерь для нахождения её локального минимума.

2.1.1.1 Математические основы

Пусть $f : \mathbb{R}^n \to \mathbb{R}$ — дифференцируемая функция, которую необходимо минимизировать. Градиент функции в точке $\mathbf{x}$ определяется как:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^T, \tag{2.1}$$

где

f — функция, которую минимизируем,

$\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ — вектор параметров,

$\frac{\partial f}{\partial x_i}$ — частная производная по x_i.

Направление наискорейшего убывания функции в точке $\mathbf{x}$ задаётся антиградиентом $-\nabla f(\mathbf{x})$. Обновление параметров на шаге t выполняется по правилу:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \nabla f(\mathbf{x}^{(t)}), \tag{2.2}$$

где

$\mathbf{x}^{(t)}$ — вектор параметров на шаге t,

$\eta > 0$ — скорость обучения (learning rate).

2.1.1.2 Алгоритм градиентного спуска

1. Инициализировать начальный вектор параметров $\mathbf{x}^{(0)}$;
2. Повторять до сходимости:
    2.1. Вычислить градиент $\nabla f(\mathbf{x}^{(t)})$;
    2.2. Обновить параметры: $\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} - \eta \nabla f(\mathbf{x}^{(t)})$;
    2.3. Проверить критерий остановки (малое изменение функции или ограничение итераций).

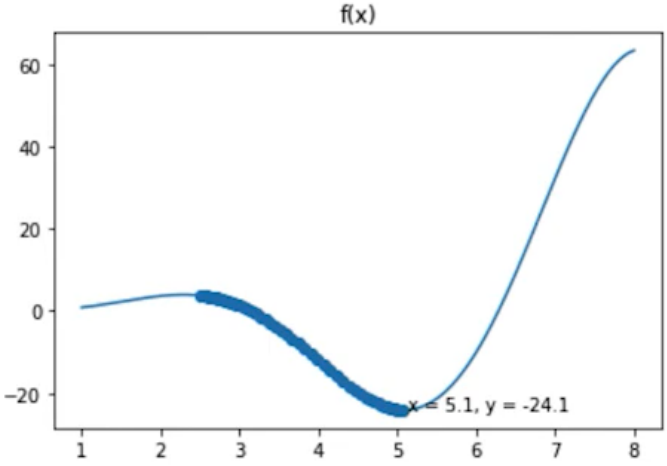

Рис.2.1. Визуализация работы градиентного спуска.

2.1.1.3 Пример оптимизации

Рассмотрим функцию $f(x) = x^2$. Градиент: $\nabla f(x) = 2x$. Процесс продолжается до приближения к минимуму $x = 0$. Обновление параметра:

$$
\begin{aligned}
x^{(0)} &= 3, \quad \eta = 0.1; \\
x^{(1)} &= 3 - 0.1 \cdot 6 = 2.4; \\
x^{(2)} &= 2.4 - 0.1 \cdot 4.8 = 1.92; \\
x^{(3)} &= 1.92 - 0.1 \cdot 3.84 = 1.536.
\end{aligned}
\tag{2.3}
$$

где

$x^{(t)}$ — значение параметра на шаге t,

η — скорость обучения,

числовые значения — вычисления по формуле обновления.

2.1.1.4 Критические параметры

1. **Скорость обучения** η:
    1.1. Слишком большое значение: расходимость;
    1.2. Слишком малое: медленная сходимость.
2. **Критерии остановки**:
    2.1. Норма градиента $\|\nabla f\| < \varepsilon$;
    2.2. Максимальное число итераций;
    2.3. Стагнация значения функции.

Примечание: Для невыпуклых функций метод может сходиться к локальным минимумам.

2.1.2 Методы оптимизации градиентного спуска (SGD, Adam)

В машинном обучении широко применяются итеративные методы оптимизации первого порядка, среди которых наиболее популярны:

2.1.2.1 Стохастический градиентный спуск (SGD)

Классический метод градиентного спуска минимизирует функцию потерь $J(\theta)$ по параметрам θ по следующему правилу:

$$
\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t), \tag{2.4}
$$

где

θ_t — параметры модели на шаге t,

η — скорость обучения,

$\nabla_\theta J(\theta_t)$ — градиент функции потерь.

В стохастическом варианте (SGD) градиент вычисляется не по всем данным, а по мини-батчу размера m:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta J(\theta_t; x_i, y_i), \tag{2.5}$$

где

(x_i, y_i) — элементы мини-батча,

остальные обозначения как выше.

2.1.2.2 Метод Adam

Adam (Adaptive Moment Estimation) сочетает идеи RMSProp и Momentum. Метод вычисляет адаптивные оценки первых и вторых моментов градиентов:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{2.6}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{2.7}$$

где

m_t — экспоненциальное скользящее среднее градиента (первый момент),

v_t — экспоненциальное скользящее среднее квадратов градиентов (второй момент),

$g_t = \nabla_\theta J(\theta_t)$ — градиент на шаге t,

$\beta_1, \beta_2 \in [0,1)$ — гиперпараметры (по умолчанию $\beta_1 = 0.9$, $\beta_2 = 0.999$).

После коррекции смещения:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \tag{2.8}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \tag{2.9}$$

где

$\hat{m}_t$, $\hat{v}_t$ — скорректированные оценки моментов.

Обновление параметров:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t, \tag{2.10}$$

где

ε — малая константа для численной стабильности.

2.1.3 Градиентный бустинг

Градиентный бустинг (Gradient Boosting) — метод машинного обучения, строящий композицию базовых алгоритмов через последовательную оптимизацию дифференцируемой функции потерь. Рассмотрим математические основы метода.

Постановка задачи

Пусть задана обучающая выборка $\mathscr{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, где $\mathbf{x}_i \in \mathscr{X} \subseteq \mathbb{R}^d$, $y_i \in \mathscr{Y}$. Требуется построить модель $F(\mathbf{x}) : \mathscr{X} \to \mathscr{Y}$, минимизирующую эмпирический риск:

$$\mathscr{R}(F) = \sum_{i=1}^{n} L(y_i, F(\mathbf{x}_i)) \to \min_{F}, \tag{2.11}$$

где

$L : \mathscr{Y} \times \mathbb{R} \to \mathbb{R}^+$ — дифференцируемая функция потерь (например, квадратичная $L(y,\hat{y}) = (y - \hat{y})^2$ или логистическая).

Аддитивное построение модели

$$F_M(\mathbf{x}) = F_0(\mathbf{x}) + \sum_{m=1}^{M} \beta_m h_m(\mathbf{x}), \tag{2.12}$$

где

$F_0(\mathbf{x})$ — начальное приближение,

$h_m(\mathbf{x}) \in \mathscr{H}$ — базовые алгоритмы (слабые предсказатели),

β_m — весовые коэффициенты.

Алгоритм градиентного бустинга

1)

$$F_0(\mathbf{x}) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma), \tag{2.13}$$

где

γ — константное начальное приближение модели.

2)

$$r_{im} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F=F_{m-1}}, \quad i = 1, \ldots, n, \tag{2.14}$$

где

r_{im} — остатки (отклики) на шаге m для объекта i,

L — функция потерь,

F_{m-1} — модель на предыдущем шаге.

3)

$$h_m = \arg\min_{h \in \mathscr{H}} \sum_{i=1}^{n} (r_{im} - h(\mathbf{x}_i))^2, \tag{2.15}$$

где

h_m — базовый алгоритм (слабый предсказатель) на шаге m,

$\mathscr{H}$ — пространство базовых моделей.

4)

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(\mathbf{x}_i) + \beta h_m(\mathbf{x}_i)\right), \tag{2.16}$$

где

β_m — коэффициент обновления для базового алгоритма h_m.

5)

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu \cdot \beta_m h_m(\mathbf{x}), \tag{2.17}$$

где

$\nu \in (0,1]$ — темп обучения (shrinkage).

Пример: квадратичная функция потерь

$$r_{im} = y_i - F_{m-1}(\mathbf{x}_i), \tag{2.18}$$

где

y_i — истинное значение отклика,

$F_{m-1}(\mathbf{x}_i)$ — прогноз модели на шаге $m - 1$.

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{n} \left[(y_i - F_{m-1}(\mathbf{x}_i)) - \beta h_m(\mathbf{x}_i)\right]^2, \tag{2.19}$$

где

минимизация по скалярному параметру β.

$$\beta_m = \frac{\sum h_m(\mathbf{x}_i) r_{im}}{\sum h_m^2(\mathbf{x}_i)}. \tag{2.20}$$

Оптимизация через спуск в пространстве функций

$$F_m = F_{m-1} - \nu \nabla_F \mathscr{R}(F_{m-1}), \tag{2.21}$$

где

$\nabla_F \mathscr{R}$ — градиент эмпирического риска по модели F,

ν — темп обучения.

$$\nabla_F \mathscr{R} = \left[\frac{\partial \mathscr{R}}{\partial F(\mathbf{x}_1)}, \ldots, \frac{\partial \mathscr{R}}{\partial F(\mathbf{x}_n)} \right]^T. \tag{2.22}$$

где

вектор градиентов по предсказаниям модели для всех объектов выборки.

2.1.4 Преимущества и ограничения методов

2.1.4.1 Градиентный спуск и его вариации

1. **Преимущества:**
    1. Универсальность: применим к широкому классу функций, где существует градиент $f : \mathbb{R}^d \to \mathbb{R}$, ∇f существует;
    2. Простота реализации: базовый алгоритм требует лишь вычисления градиента;
    3. Масштабируемость: стохастические варианты (SGD) работают с большими данными и позволяют итеративную обработку обучающей выборки.
2. **Ограничения:**
    1. Чувствительность к выбору гиперпараметров, особенно скорости обучения η;
    2. Возможность сходимости к локальным минимумам, особенно при оптимизации невыпуклых функций;
    3. Замедленная сходимость или колебания на плато и в "оврагах"функции, что требует использования адаптивных методов, таких как Adam или RMSProp.

2.1.4.2 Методы оптимизации (SGD, Adam)

1. **Преимущества:**

1. Adam обеспечивает адаптивное масштабирование градиентов:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2;$$

2. Устойчивость к разреженным и шумным градиентам;
3. Быстрая сходимость на невыпуклых функциях, особенно в задачах глубокого обучения.

2. **Ограничения:**
   1. Наличие дополнительных гиперпараметров β_1, β_2, ε, чувствительных к выбору;
   2. Требование дополнительной памяти: Adam использует $O(2d)$ памяти против $O(d)$ у SGD;
   3. Возможность расходимости при плохой настройке параметров, особенно на нестабильных ландшафтах функции потерь.

2.1.4.3 Градиентный бустинг

1. **Преимущества:**
   1. Высокая точность моделей, особенно при увеличении количества итераций M, что снижает смещение:

$$\text{Bias} \downarrow \text{ при } M \uparrow;$$

   2. Гибкость: возможность выбора различных функций потерь, например:

$$L(y, F) \in \{\text{MSE, LogLoss, Huber, ...}\};$$

   3. Регуляризация через параметр shrinkage ν, что повышает обобщающую способность модели.

2. **Ограничения:**
   1. Высокая вычислительная сложность $O(M \cdot T(n,d))$, где $T(n,d)$ — сложность построения одного базового алгоритма;
   2. Риск переобучения при избыточном числе итераций M;
   3. Чувствительность к шуму в обучающих данных;
   4. Снижение интерпретируемости по сравнению с линейными моделями.

2.2 Пример кода Adam, AdaBoost, SGD на языке Python

2.2.1 SGD (стохастический градиентный спуск)

SGD — это итеративный метод оптимизации, в котором на каждом шаге градиент функции потерь вычисляется не по всей выборке, а по одному (или небольшому мини-батчу) случайно выбранному примеру. В каждом обновлении параметры сдвигаются вдоль антиградиента одного случайного объекта.

```python
def sgd (grad_func, theta_init, data, alpha=0.01, epochs=1000):
    theta = theta_init
    for _ in range(epochs):
        x_i, y_i = random.choice(data)
        g = grad_func(theta, x_i, y_i)
        theta = theta - alpha * g
    return theta
```

Рис.2.2. код алгоритма SGD на языке Python

2.2.2 Adam (адаптивный метод оптимизации)

Adam — это стохастический алгоритм оптимизации, который адаптивно подбирает темп обучения для каждого параметра по оценкам первого (m) и второго (v) моментов градиента. На каждом шаге он обновляет скользящие средние градиента и квадрата градиента, а затем корректирует их смещение и обновляет параметры.

```python
def adam(grad_func, theta_init, alpha=0.001, beta1=0.9, beta2=0.999, eps=1e-8, epochs=1000):
        theta = theta_init
        m = 0
        v = 0
        for t in range(1, epochs + 1):
            g = grad_func(theta)
            m = beta1 * m + (1 - beta1) * g
            v = beta2 * v + (1 - beta2) * (g ** 2)
            m_hat = m / (1 - beta1 ** t)
            v_hat = v / (1 - beta2 ** t)
            theta = theta - alpha * m_hat / (v_hat ** 0.5 + eps)
        return theta
```

Рис.2.3. код алгоритма Adam на языке Python

2.2.3 AdaBoost (адаптивный бустинг)

AdaBoost — это метод бустинга, который последовательно строит ансамбль слабых классификаторов, фокусируясь на ошибочно классифицированных объектах предыдущими моделями (в рамках adaboost.py - П1.1). Сначала всем примерам присваиваются одинаковые веса, затем на каждом шаге обучается новый слабый классификатор на выборке с этими весами, рассчитывается его взвешенная ошибка, вычисляется коэффициент α и пересчитываются веса примеров.

2.3 Используемые инструменты и библиотеки для построения моделей ИНС

В практической части для реализации и тестирования моделей использовались следующие библиотеки: `PyTorch`, `TorchText`, `NumPy`, `pandas`, `matplotlib`, `scikit-learn`. Средой выполнения выступала платформа Google Colab, обеспечивающая доступ к GPU и необходимым вычислительным ресурсам. Данный выбор объясняется удобством работы с нейросетевыми моделями и наличием готовых средств визуализации.

2.4 Данные для обучения моделей

2.4.1 Синтетические данные

2.4.1.1 Функция Растригина

Характеризуется наличием большого количества локальных минимумов, что делает ее сложной для поиска глобального оптимума.

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i) \right], \tag{2.23}$$

где:

$\mathbf{x} = (x_1, x_2, \ldots, x_n)$ — вектор входных значений;

A — параметр функции, обычно $A = 10$;

n — размерность пространства.

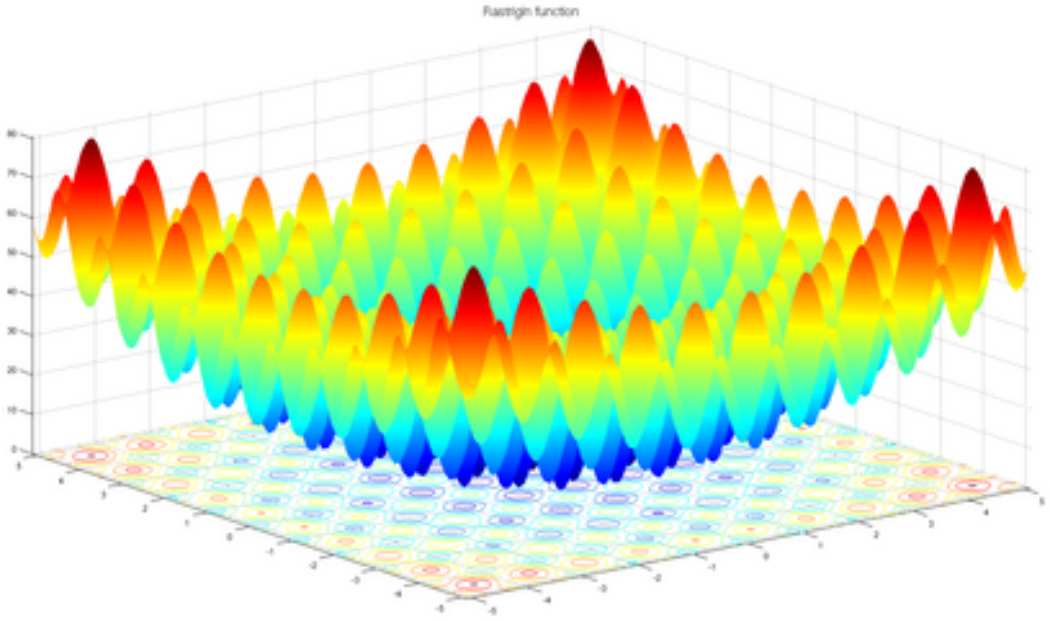

Рис.2.4. График функции Растригина

2.4.1.2 Функция Экли

Представляет собой невыпуклую задачу, которая также имеет множество локальных минимумов, создавая риск "застревания"для алгоритмов.

$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(cx_i)\right) + a + \exp(1), \qquad (2.24)$$

где:

$\mathbf{x} = (x_1, x_2, \ldots, x_n)$ — вектор входных значений;

a — параметр функции, обычно $a = 20$;

b — параметр функции, обычно $b = 0.2$;

c — параметр функции, обычно $c = 2\pi$;

n — размерность пространства.

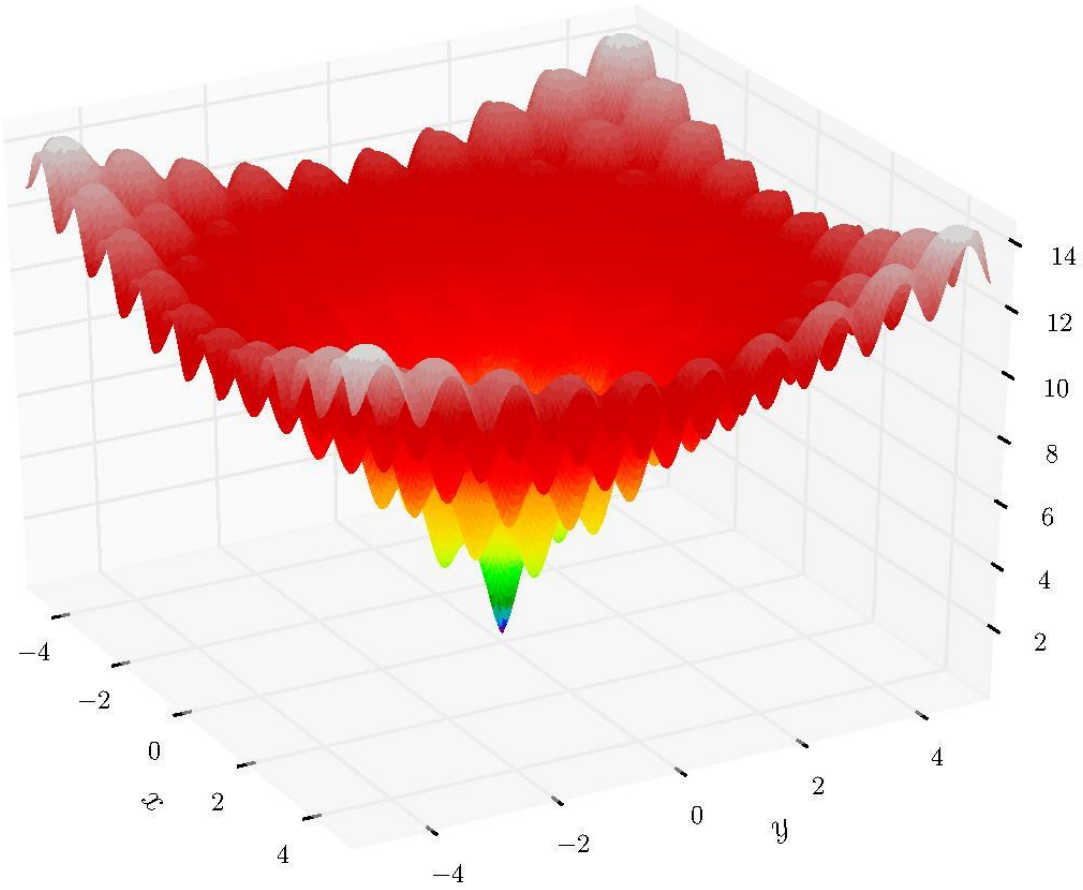

Рис.2.5. График функции Экли

2.4.1.3 Функция Розенброка

Имеет один глобальный минимум, который находится в узкой, параболической "долине". Алгоритмам легко найти саму долину, но крайне сложно сойтись к глобальному минимуму.

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right], \tag{2.25}$$

где:

$\mathbf{x} = (x_1, x_2, \ldots, x_n)$ — вектор входных значений;

n — размерность пространства.

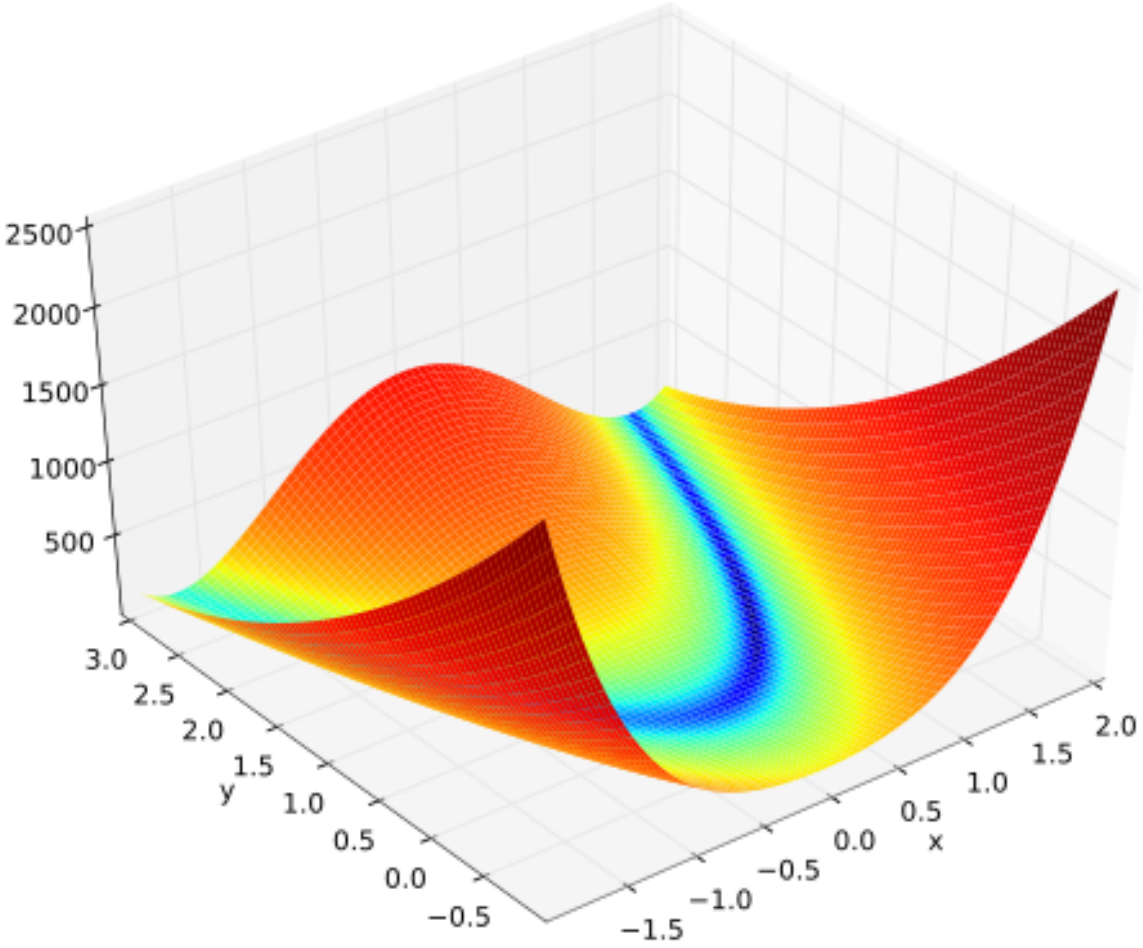

Рис.2.6. График функции Розенброка

2.4.2 Датасет `Language Identification Dataset`

2.4.2.1 Основные характеристики:

1. **Объём данных:**
    1. 21 859 уникальных текстовых примеров (столбец `Element method14`);
    2. 22 уникальных языка.
2. **Языковое разнообразие:** Включает 22 языка, что покрывает широкий спектр мультиязычных сценариев. Пример метки: `"Estonian"`.
3. **Формат данных:**
    1. Тексты хранятся в столбце `Element method14`;
    2. Соответствующие метки языков представлены в отдельном столбце.
4. **Источник:** Датасет загружен с платформы Kaggle через `kagglehub`, что обеспечивает его стандартизированное хранение и доступность.

2.5 Выводы

Во второй главе подробно рассмотрены алгоритмы стохастического градиентного спуска (SGD), адаптивной оптимизации (Adam) и бустинга (AdaBoost). Для каждого из них приведены математические формулы, описывающие процесс обновления весов, а также выделены ключевые параметры, влияющие на эффективность обучения. Также приведены данные для дальнейшего обучения моделей.

- **SGD** — очень простой и широко используемый метод оптимизации. Он требует минимальных вычислительных ресурсов для реализации, так как на каждом шаге обновляет параметры по одному примеру. Благодаря своей простоте отлично подходит для базовых задач оптимизации и легко реализуется в коде.
- **Adam** сочетает в себе техники моментов и адаптивного масштаба шага. Его алгоритм более громоздкий (две скользящие усреднённые статистики), но благодаря этому он сходится быстрее и устойчивее. Для реализации нужны дополнительные массивы (m, v) и формулы коррекции смещения, однако в современных библиотеках это встроено и требует нескольких строчек кода.
- **AdaBoost** — метод бустинга, базирующийся на создании ансамбля слабых классификаторов. Его реализация наиболее трудоёмка: надо циклически обучать модели с учётом весов примеров и корректировать эти веса для фокусировки на ошибках предыдущих классификаторов. Как правило, AdaBoost используют через готовые функции (например, `AdaBoostClassifier`), иначе придётся самостоятельно реализовывать слабую модель и процедуру взвешивания, что существенно сложнее.

3 ПРАКТИЧЕСКАЯ ЧАСТЬ

3.1 Постановка эксперимента на реальных данных

3.1.1 Обучение моделей

3.1.1.1 SGD

Линейная модель (в рамках language_indent_SGD.py - П1.2) была реализована с использованием стохастического градиентного спуска (SGD), встроенного в библиотеку torch.optim.

Модель показала следующие результаты:

```
[Epoch 1] Loss: 4967.0681, Accuracy: 0.5038
[Epoch 2] Loss: 3715.7890, Accuracy: 0.7795
[Epoch 3] Loss: 3040.6394, Accuracy: 0.8428
[Epoch 4] Loss: 2628.5060, Accuracy: 0.8698
[Epoch 5] Loss: 2346.7280, Accuracy: 0.8879
[Epoch 6] Loss: 2135.6013, Accuracy: 0.8926
[Epoch 7] Loss: 1968.6788, Accuracy: 0.9024
[Epoch 8] Loss: 1831.6089, Accuracy: 0.9052
[Epoch 9] Loss: 1714.7190, Accuracy: 0.9084
[Epoch 10] Loss: 1613.1574, Accuracy: 0.9106
Finished Training
```

Рис.3.1. Метрики loss и accuracy на тренировочных данных

```
Test Accuracy: 0.9173
```

Рис.3.2. Метрика accuracy на тестовых данных

Модель обучилась значительно быстро, визуализация обучения включает построение графиков функции потерь по эпохам и итерациям, с отображением направлений шагов градиентного спуска:

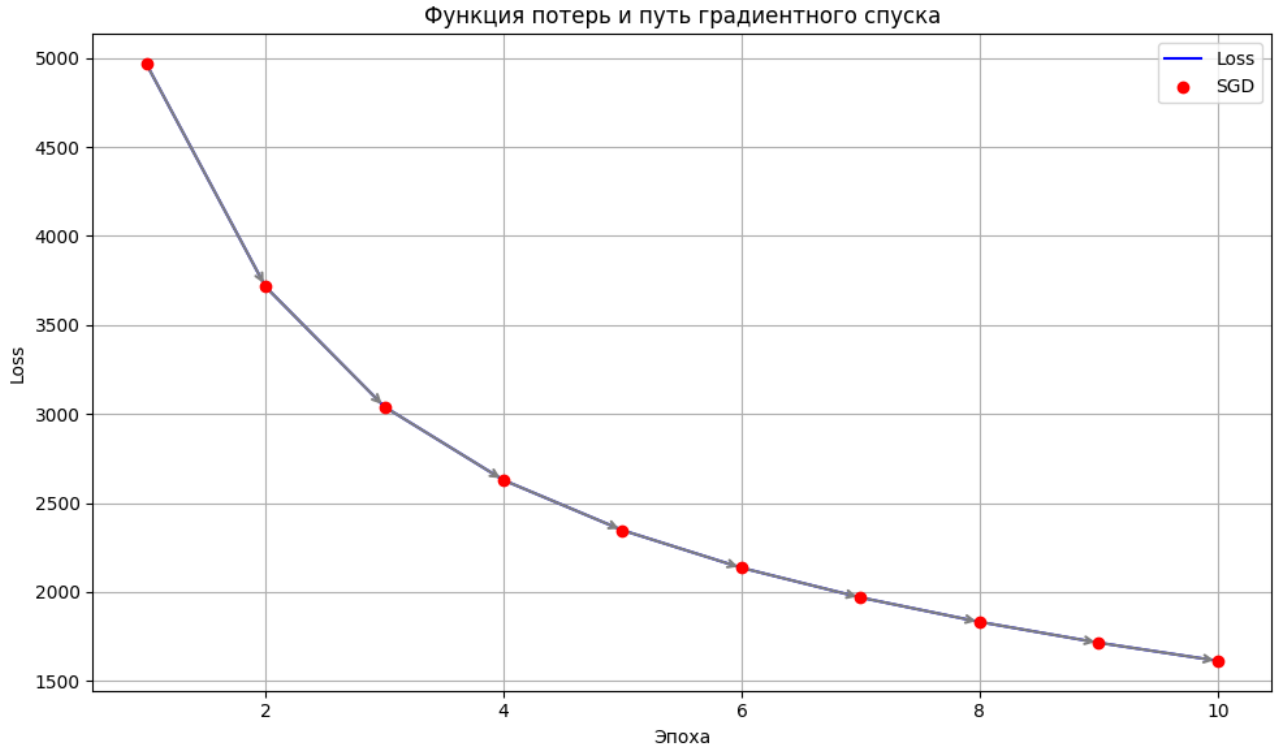

Рис.3.3. Функция потерь и путь градиентного спуска по эпохам

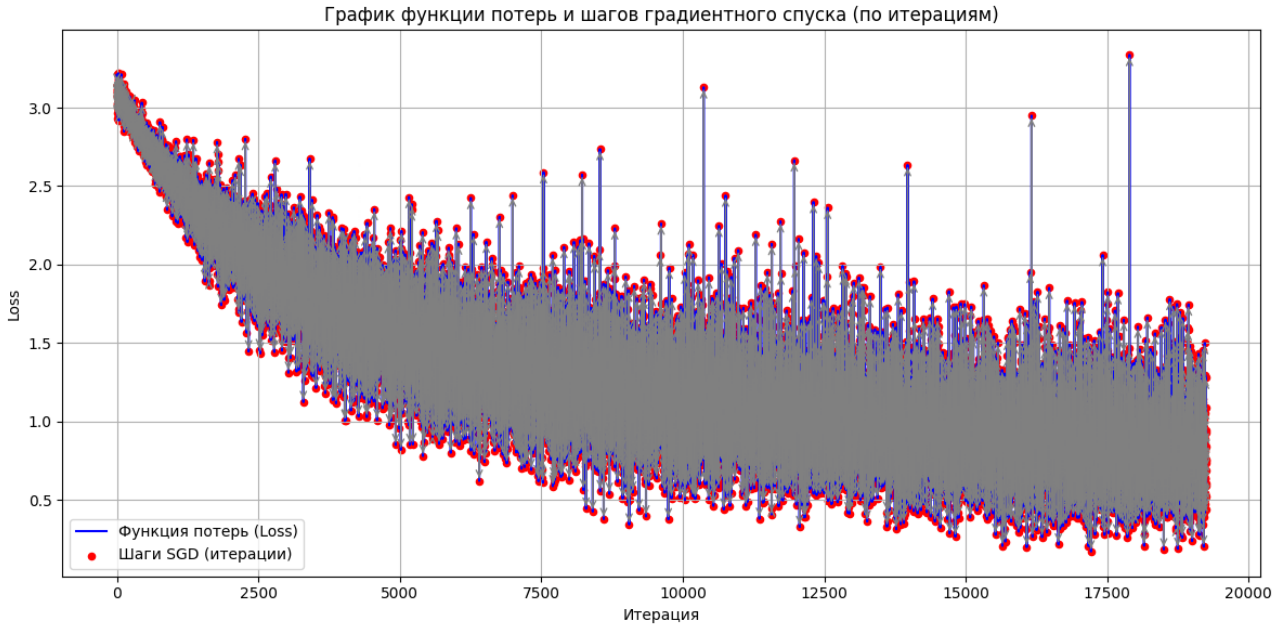

Рис.3.4. Функция потерь и путь градиентного спуска по итерациям

3.1.1.2 Adam

Линейная модель (в рамках language_indent_Adam.py - П1.3) была реализована с использованием адаптивного метода оптимизации Adam, встроенного в библиотеку torch.optim.

Модель показала следующие результаты:

```
[Epoch 1] Loss: 2383.7023, Accuracy: 0.7852
[Epoch 2] Loss: 765.2097, Accuracy: 0.9455
[Epoch 3] Loss: 468.0436, Accuracy: 0.9579
[Epoch 4] Loss: 365.5217, Accuracy: 0.9637
[Epoch 5] Loss: 317.0232, Accuracy: 0.9657
[Epoch 6] Loss: 289.4503, Accuracy: 0.9679
[Epoch 7] Loss: 272.2493, Accuracy: 0.9693
[Epoch 8] Loss: 259.3417, Accuracy: 0.9697
[Epoch 9] Loss: 249.9034, Accuracy: 0.9706
[Epoch 10] Loss: 242.6293, Accuracy: 0.9716
Finished Training
```

Рис.3.5. Метрики loss и accuracy на тренировочных данных

```
Test Accuracy: 0.9692
```

Рис.3.6. Метрика accuracy на тестовых данных

У этой модели показатели метрики accuracy выше, чем у модели с SGD.

Визуализация обучения включает построение графиков функции потерь по эпохам и итерациям, с отображением направлений шагов градиентного спуска:

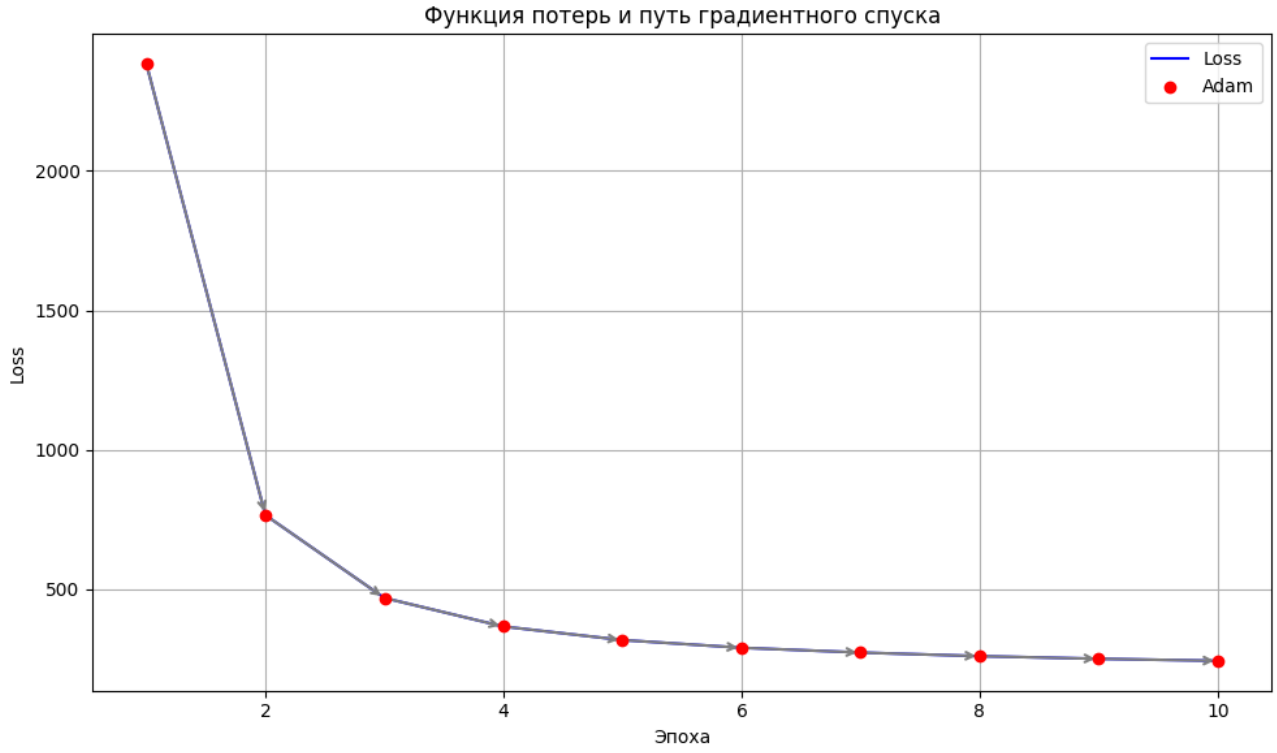

Рис.3.7. Функция потерь и путь градиентного спуска по эпохам

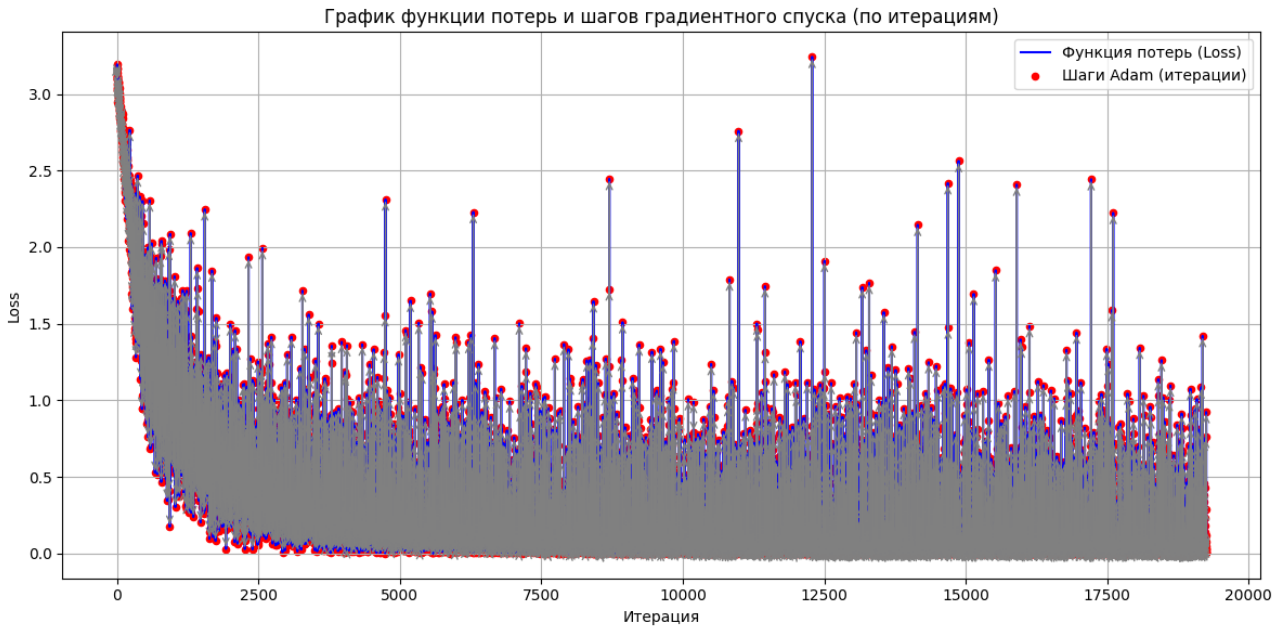

Рис.3.8. Функция потерь и путь градиентного спуска по итерациям

3.1.1.3 AdaBoost

Модель ИНС (в рамках language_indent_adaboost.py — П1.4) была реализована на базе простой нейронной сети с одним скрытым слоем, использующей слой эмбеддинга (nn.EmbeddingBag) для представления входных символов в виде плотных векторов. В качестве функции активации применялась ReLU, а выходной слой соответствовал количеству языковых классов.

Обучение модели происходило в составе ансамбля по методу AdaBoost, где каждая базовая нейросеть обучалась на взвешенной выборке, сформированной с учетом ошибок предыдущих моделей. Оптимизация параметров сети выполнялась с использованием адаптивного градиентного оптимизатора Adam (torch.optim.Adam), а в качестве функции потерь применялась взвешенная CrossEntropyLoss, учитывающая важность каждого обучающего примера.

Итоговая модель агрегировала предсказания всех нейросетей ансамбля с помощью взвешенного голосования по их уверенностям (softmax-прогнозам), масштабированным коэффициентами доверия (альфами AdaBoost).

Модель показала следующие результаты:

```
Estimator 1/10 — Loss: 0.1616 Train Acc: 0.9674 Test Acc: 0.9655
Estimator 2/10 — Loss: 0.3881 Train Acc: 0.9647 Test Acc: 0.9650
Estimator 3/10 — Loss: 0.5296 Train Acc: 0.9599 Test Acc: 0.9655
Estimator 4/10 — Loss: 0.5942 Train Acc: 0.9601 Test Acc: 0.9656
Estimator 5/10 — Loss: 0.6097 Train Acc: 0.9601 Test Acc: 0.9665
Estimator 6/10 — Loss: 0.6204 Train Acc: 0.9626 Test Acc: 0.9665
Estimator 7/10 — Loss: 0.6267 Train Acc: 0.9607 Test Acc: 0.9665
Estimator 8/10 — Loss: 0.6375 Train Acc: 0.9619 Test Acc: 0.9665
Estimator 9/10 — Loss: 0.6378 Train Acc: 0.9606 Test Acc: 0.9665
Estimator 10/10 — Loss: 0.6388 Train Acc: 0.9612 Test Acc: 0.9665
```

Рис.3.9. Метрики loss и accuracy на тренировочных и тестовых данных

```
Итоговая точность AdaBoost: 0.9665
```

Рис.3.10. Метрика accuracy на тестовых данных

Визуализация обучения включает построение графиков:

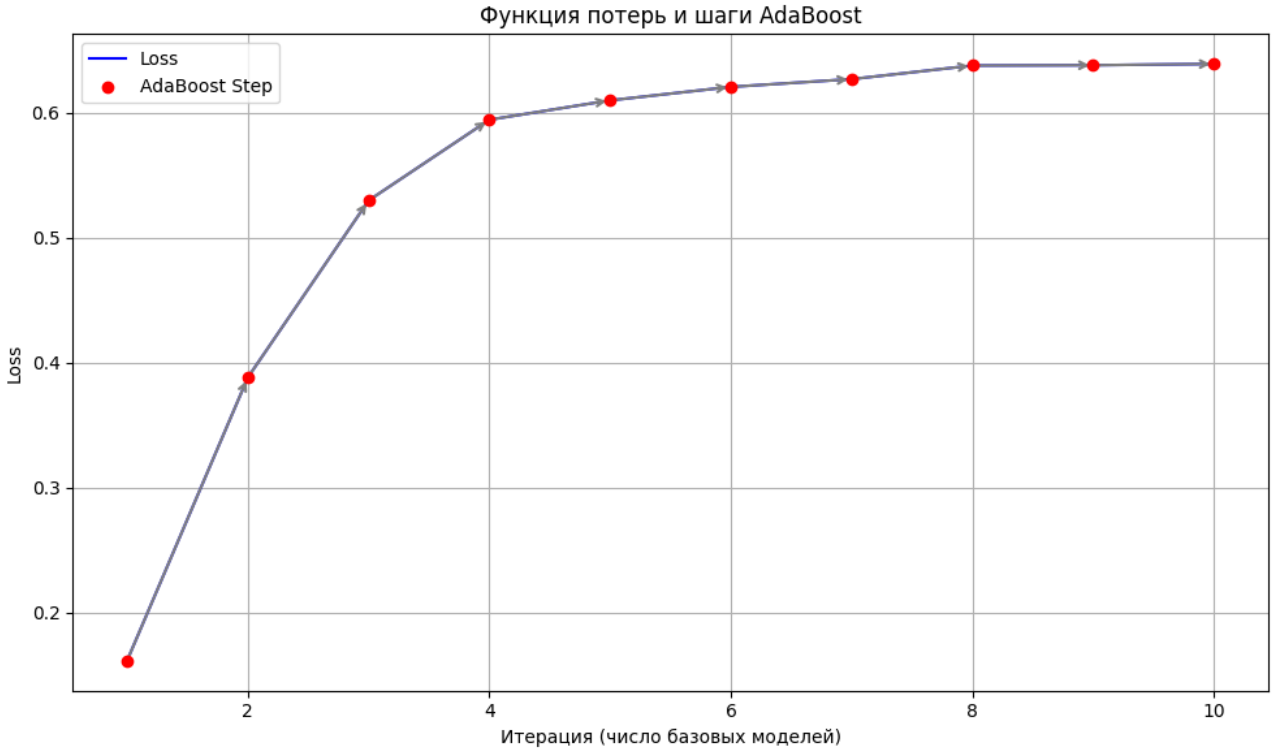

Рис.3.11. Функция потерь и шаги AdaBoost

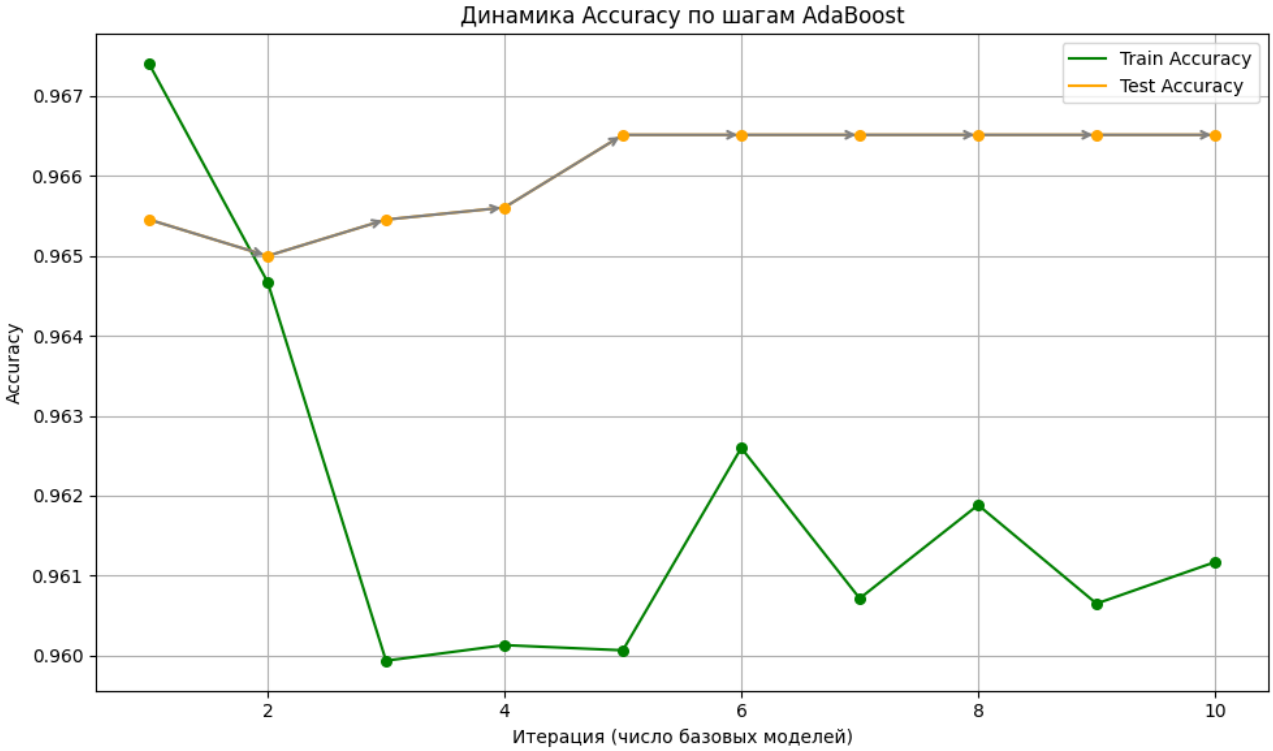

Рис.3.12. Функция потерь и шаги AdaBoost

3.2 Постановка эксперимента на синтетических данных

3.2.1 функция Растригина

Нелинейная функция Растригина (в рамках sint1.py — П1.5) была исследована с использованием трёх различных методов оптимизации: стохастического градиентного спуска (SGD), адаптивного метода Adam и ансамблевого подхода AdaBoost[3].

Градиентные методы реализованы через torch.optim с многократными перезапусками из случайных точек для выхода из локальных минимумов.

AdaBoost использован через sklearn.ensemble.AdaBoostRegressor, обученный на синтетических данных, с предсказанием минимума по сетке параметров.

Визуализация включала трёхмерную поверхность функции и траектории оптимизаторов для наглядного сравнения эффективности:

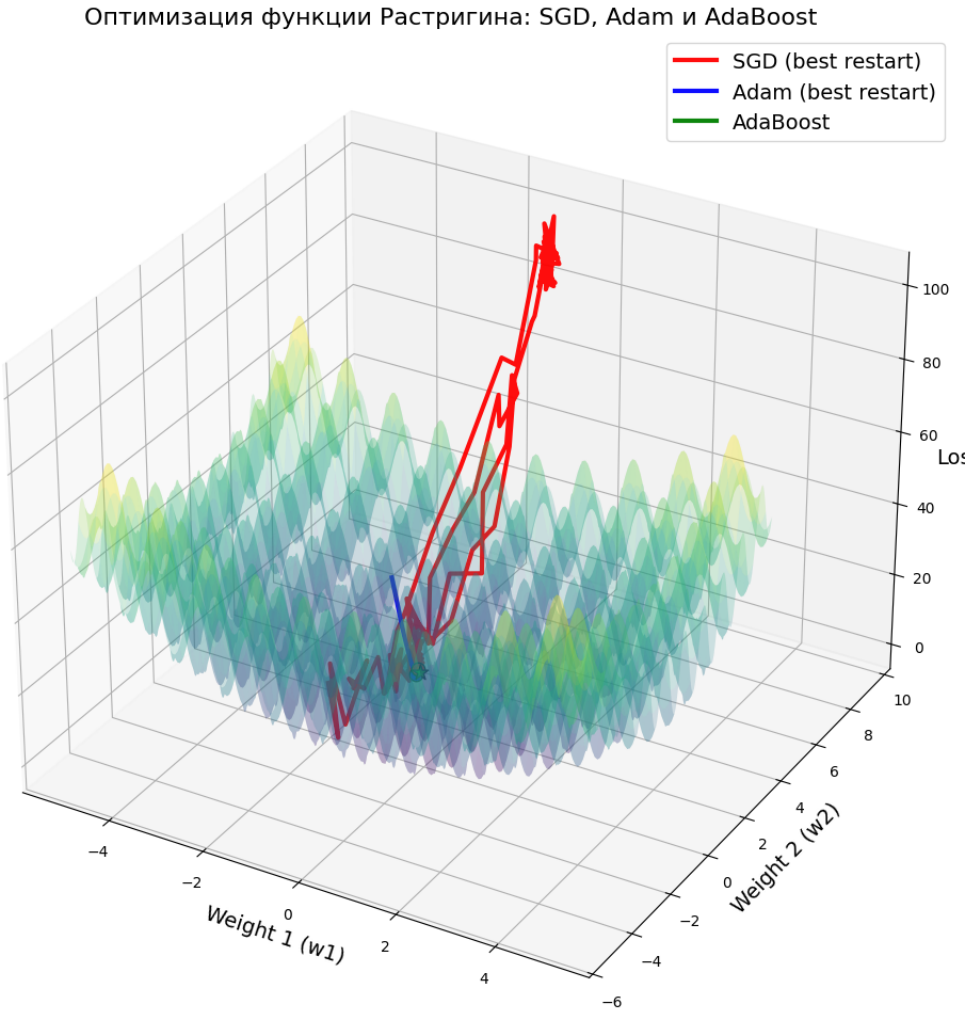

Рис.3.13. Функция Растригина

Насколько близко оптимизаторы подошли к глобальному минимуму:

```
SGD final:
  w1 = -0.9139, w2 = 1.9024, loss = 7.3906
Adam final:
  w1 = -0.9952, w2 = 1.9898, loss = 4.9748
AdaBoost final:
  w1 = -0.0700, w2 = -0.0200, loss = 19.3667
```

Рис.3.14. Результат работы оптимизаторов

3.2.2 Функция Экли

Нелинейная функция Экли (в рамках sint2.py — П1.6) была исследована с использованием трёх различных методов оптимизации: стохастического градиентного спуска (SGD), адаптивного метода Adam и ансамблевого подхода AdaBoost[2].

Градиентные методы реализованы через `torch.optim` с многократными перезапусками из случайных точек для выхода из локальных минимумов.

AdaBoost использован через sklearn.ensemble.AdaBoostRegressor, обученный на синтетических данных, с предсказанием минимума по сетке параметров.

Визуализация включала трёхмерную поверхность функции и траектории оптимизаторов для наглядного сравнения эффективности:

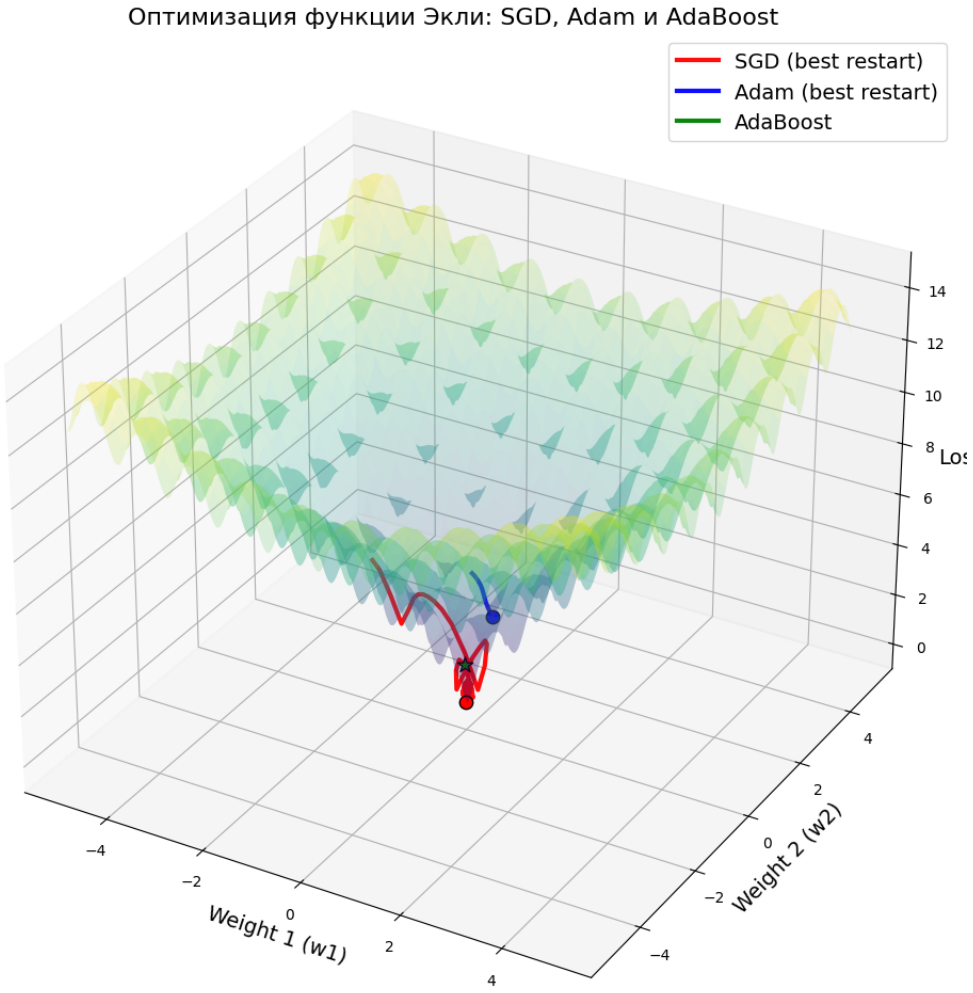

Рис.3.15. Функция Экли

Насколько близко оптимизаторы подошли к глобальному минимуму:

```
SGD final:
  w1 = -0.0116, w2 = 0.0150, loss = 0.0748
Adam final:
  w1 = 0.0001, w2 = 0.9522, loss = 2.5799
AdaBoost final:
  w1 = 0.0000, w2 = -0.0500, loss = 1.6645
```

Рис.3.16. Результат работы оптимизаторов

3.2.3 Функция Розенброка

Нелинейная функция Розенброка (в рамках sint3.py — П1.7) была исследована с использованием трёх различных методов оптимизации: стохастического градиентного спуска (SGD), адаптивного метода Adam и ансамблевого подхода AdaBoost[4].

Градиентные методы реализованы через `torch.optim` с многократными перезапусками из случайных точек для выхода из локальных минимумов.

AdaBoost использован через sklearn.ensemble.AdaBoostRegressor, обученный на синтетических данных, с предсказанием минимума по сетке параметров.

Визуализация включала трёхмерную поверхность функции и траектории оптимизаторов для наглядного сравнения эффективности:

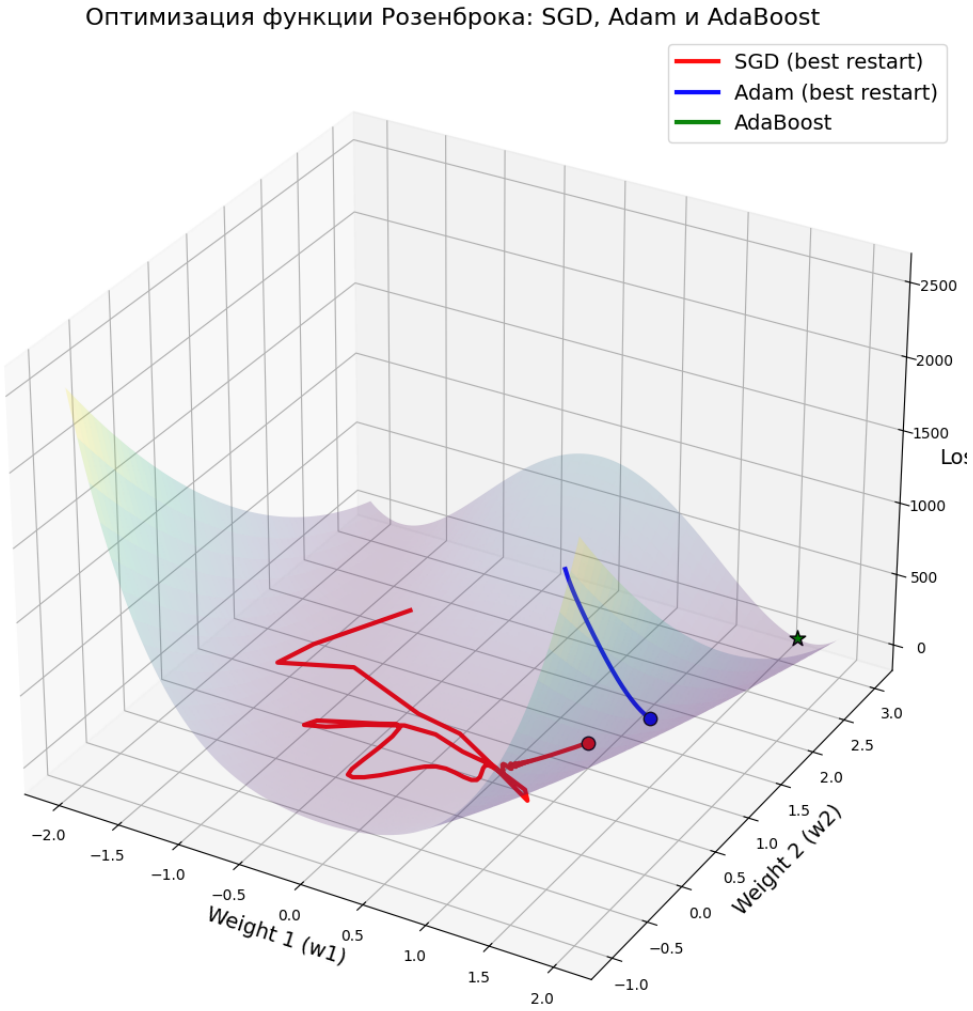

Рис.3.17. Функция Розенброка

Насколько близко оптимизаторы подошли к глобальному минимуму:

```
SGD final:
  w1 = 0.9999, w2 = 0.9999, loss = 0.0000
Adam final:
  w1 = 1.2259, w2 = 1.5036, loss = 0.0511
AdaBoost final:
  w1 = 1.7000, w2 = 2.9000, loss = 69.4799
```

Рис.3.18. Результат работы оптимизаторов

3.3 Сравнительный анализ методов

Таблица 3.1

Результаты тестирования моделей

Использованный метод	Значение accuracy на тестовых данных
SGD	0.9173
Adam	0.9692
AdaBoost	0.9665

Анализ:

1) Оптимизатор Adam показывает наилучший результат по точности: 0.9692, немного обгоняя AdaBoost (0.9665).
2) Метод SGD значительно уступает по качеству классификации — 0.9173.

Вывод: Для данной задачи классификации адаптивные методы (Adam, AdaBoost) явно превосходят по качеству простой стохастический градиентный спуск (SGD).

Таблица 3.2

Сравнение оптимизаторов на различных функциях

Функция	Оптимизатор	Параметры (w1, w2)	Значение функции (loss)
Растригина	SGD	-0.9139, 1.9024	7.3906
Растригина	Adam	-0.9952, 1.9898	4.9748
Растригина	AdaBoost	-0.0700, -0.0200	19.3667
Экли	SGD	-0.0116, 0.0150	0.0748
Экли	Adam	0.0001, 0.9522	2.5799
Экли	AdaBoost	0.0000, -0.0500	1.6645
Розенброка	SGD	0.9999, 0.9999	0.0000
Розенброка	Adam	1.2259, 1.5036	0.0511
Розенброка	AdaBoost	1.7000, 2.9000	69.4799

1. Функция Растригина (сложная, с множеством локальных минимумов):
    1. Adam — лучшее качество (наименьший loss: 4.9748).
    2. SGD — хуже (7.3906), но лучше, чем AdaBoost (19.3667).

Вывод: Adam наиболее стабилен на этой функции.

2. Функция Экли (Ackley):
    1. SGD дал наилучший результат: 0.0748.
    2. AdaBoost — 1.6645.
    3. Adam — 2.5799.

    Вывод: Здесь SGD оказался эффективнее.

3. Функция Розенброка (узкая долина, сложна для оптимизации):
    1. SGD почти попал в идеальный минимум.
    2. Adam — также близок к оптимуму: 0.0511.
    3. AdaBoost — провал: 69.4799.

    Вывод: SGD оказался лучшим, AdaBoost — неэффективен.

3.4 Выводы

Таблица 3.3

Сравнительный анализ оптимизаторов

Метод	Преимущества	Недостатки
SGD	Отлично работает на функции Экли; попал в глобальный минимум функции Розенброка	Худший результат классификации; уступает на функции Растригина
Adam	Лучшая точность классификации; лучший результат на функции Растригина; близок к минимуму функции Розенброка	Уступает SGD на функции Экли; не достиг глобального минимума на Розенброке
AdaBoost	Высокая точность классификации (на уровне Adam)	Худшие значения функции потерь на всех задачах оптимизации; особенно неэффективен на функции Розенброка

ЗАКЛЮЧЕНИЕ

В ходе выполнения научно-исследовательской работы была достигнута поставленная цель — проведён сравнительный анализ эффективности методов градиентного спуска (SGD, Adam) и градиентного бустинга (AdaBoost) при обучении искусственных нейронных сетей. Результаты работы подтверждают успешное решение всех поставленных задач:

1. Проведено детальное изучение литературы по методам градиентного спуска, включая их историю, современные достижения и актуальные задачи. Анализ показал, что современные методы оптимизации стали более адаптивными, автоматизированными и масштабируемыми, что позволяет эффективно решать задачи машинного обучения даже в условиях ограниченных ресурсов.
2. Исследованы математические основы методов SGD, Adam и AdaBoost. Для каждого алгоритма были рассмотрены ключевые формулы, преимущества и ограничения, что позволило глубже понять их работу и области применения.
3. Разработаны критерии сравнения эффективности методов, включая точность классификации и скорость сходимости на различных типах данных. Это позволило объективно оценить результаты экспериментов.
4. Практически реализованы методы с использованием библиотеки `PyTorch`. Код для каждого алгоритма был адаптирован под конкретные задачи, что обеспечило воспроизводимость экспериментов.
5. Проведены эксперименты на реальных и синтетических данных, включая датасет *Language Identification Dataset* и тестовые функции (Растригина, Экли, Розенброка). Результаты показали, что:
    - **Adam** демонстрирует наилучшую точность (0.9692) на реальных данных и стабильность на сложных функциях (например, Растригина).
    - **SGD** эффективен на функциях с узкими “долинами” (например, Розенброка) и простых задачах, но уступает в точности на реальных данных.
    - **AdaBoost** показывает высокую точность классификации (0.9665), но менее эффективен для оптимизации сложных функций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Прикладная практика оптимизации и немного истории. — URL: https://habr.com/ru/articles/250855/ (дата обращения: 16.07.2025).

2. *Wikipedia contributors*. Ackley function. — 2024. — (Accessed: 2025-07-17). https://en.wikipedia.org/wiki/Ackley_function.

3. *Wikipedia contributors*. Rastrigin function. — 2024. — (Accessed: 2025-07-17). https://en.wikipedia.org/wiki/Rastrigin_function.

4. *Wikipedia contributors*. Rosenbrock function. — 2024. — (Accessed: 2025-07-17). https://en.wikipedia.org/wiki/Rosenbrock_function.

5. *Википедия*. Градиентный спуск — Википедия. — 2023. — (Версия статьи от 17.07.2025). https://ru.wikipedia.org/wiki/РУСЃРѳРұРчРхР„СЂР„СЌРє_СѓР»СЂСѓРә.

6. Gradient Descent / Wikipedia. — URL: https://en.wikipedia.org/wiki/Gradient_descent (visited on 16.07.2025).

7. Recent Advances in Optimization Methods for Machine Learning // Mathematics. — 2025. — Vol. 13, no. 13. — P. 2210. — URL: https://doi.org/10.3390/math13132210.

Приложение 1

Листинг программного кода

Листинг П1.1

adaboost.py

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

def adaboost(X, y, T=50):
    N = len(y)
    D = np.ones(N) / N
    classifiers = []
    alphas = []

    for t in range(T):
        clf = DecisionTreeClassifier(max_depth=1)
        clf.fit(X, y, sample_weight=D)
        pred = clf.predict(X)
        error = np.sum(D * (pred != y))

        if error >= 0.5:
            break

    alpha = 0.5 * np.log((1 - error) / (error + 1e-10))
    D = D * np.exp(-alpha * y * pred)
    D = D / np.sum(D)

    classifiers.append(clf)
    alphas.append(alpha)

    def final_model(x):
        result = sum(alpha * clf.predict(x.reshape(1, -1))[0]
           for alpha, clf in zip(alphas, classifiers))
        return np.sign(result)

    return final_model
```

Листинг П1.2

language_indent_SGD.py

```
import torch
```

```
device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")

# Загрузка датасета
import kagglehub
path = kagglehub.dataset_download("zarajamshaid/language-
   identification-datasst")
print("Path to dataset files:", path)

# Чтение CSV
import pandas as pd
data = pd.read_csv(path + "/dataset.csv", sep=',')
print("Пропущенные значения:", data.columns[data.isna().any()
   ].tolist())

# Кодируем языки
data["language"] = data["language"].astype('category').cat.
   codes
language_count = data["language"].nunique()

# 1. collections.Counter
from collections import Counter

def yield_tokens(data_train):
    for text in data_train:
        yield list(text)

all_tokens = [token for tokens in yield_tokens(data["Text"])
   for token in tokens]
counter = Counter(all_tokens)
specials = ["<unk>"]
vocab = {token: idx for idx, token in enumerate(specials +
   list(counter.keys()))}
default_index = vocab["<unk>"]

# Разделение данных
from sklearn.model_selection import train_test_split

y = data["language"]
X = data["Text"]
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.3, random_state=42)

# Класс Dataset
import torch
```

```
from torch.utils.data import Dataset

class TextDataset(Dataset):
    def __init__(self, data_text, labels):
        self.data = data_text.reset_index(drop=True)
        self.labels = labels.reset_index(drop=True)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        token_indices = tokens_to_indices(list(self.data[idx])
            )
        text_tensor = torch.tensor(token_indices, dtype=torch.
            int64)
        label_tensor = torch.tensor(self.labels[idx], dtype=
            torch.int64)
        return text_tensor, label_tensor

trainset = TextDataset(X_train, y_train)
testset = TextDataset(X_test, y_test)

# Collate-функция
def collate_batch(batch):
    label_list, text_list, offsets = [], [], [0]
    for text, label in batch:
        label_list.append(label)
        text_list.append(text)
        offsets.append(text.size(0))
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text_list = torch.cat(text_list)
    label_list = torch.tensor(label_list, dtype=torch.int64)
    return label_list, text_list, offsets

# DataLoader
from torch.utils.data import DataLoader

trainloader = DataLoader(trainset, batch_size=8, shuffle=True,
    collate_fn=collate_batch)
testloader = DataLoader(testset, batch_size=8, shuffle=False,
   collate_fn=collate_batch)

# Модель
import torch.nn as nn
```

```
vocab_size = len(vocab)
num_class = language_count

class TextClassificationModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, 64,
           sparse=True)
        self.fc = nn.Linear(64, num_class)

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)

net = TextClassificationModel()

def tokens_to_indices(tokens):
    return [vocab.get(token, default_index) for token in
       tokens]

# Обучение модели
import torch.optim as optim
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

losses = []
all_losses = []  # (iteration, epoch, loss)
net.train()

iteration = 0

for epoch in range(10):  # или 50
    running_loss = 0.0
    running_corrects = 0
    total = 0

    for labels, inputs, offsets in trainloader:
        optimizer.zero_grad()
        outputs = net(inputs, offsets)
        loss = criterion(outputs, labels)
        loss.backward()
```

```
        optimizer.step()

        # фиксируем loss на каждой итерации
        all_losses.append((iteration, epoch, loss.item()))
        iteration += 1

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item()
        running_corrects += (preds == labels).sum().item()
        total += labels.size(0)

    print(f"[Epoch {epoch + 1}] Loss: {running_loss:.4f},
       Accuracy: {running_corrects / total:.4f}")
    losses.append(running_loss)

print("Finished Training")

# Тестирование
net.eval()
running_correct = 0
num_of_tests = 0

with torch.no_grad():
    for labels, inputs, offsets in testloader:
        outputs = net(inputs, offsets)
        _, predicted = torch.max(outputs, 1)
        running_correct += (predicted == labels).sum().item()
        num_of_tests += labels.size(0)

print(f"Test Accuracy: {running_correct / num_of_tests:.4f}")

# Первый график

import matplotlib.pyplot as plt
import numpy as np

epochs = np.arange(1, len(losses) + 1)
loss_values = np.array(losses)

plt.figure(figsize=(10, 6))

# Основная линия это функция потерь
plt.plot(epochs, loss_values, color='blue', linestyle='-',
   label='Loss')
```

```
# Точки градиентного спуска
plt.scatter(epochs, loss_values, color='red', zorder=5, label
    ='SGD')

# Стрелки (градиентный спуск)
for i in range(1, len(epochs)):
    plt.annotate('',
                 xy=(epochs[i], loss_values[i]),
                 xytext=(epochs[i-1], loss_values[i-1]),
                 arrowprops=dict(arrowstyle='->', color='gray
                     ', lw=1.5))

# Настройки графика
plt.title("Функция потерь и путь градиентного спуска")
plt.xlabel("Эпоха")
plt.ylabel("Loss")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Второй график

import matplotlib.pyplot as plt
import numpy as np

# Извлекаем значения
iterations, epochs_list, loss_values = zip(*all_losses)
iterations = np.array(iterations)
loss_values = np.array(loss_values)

plt.figure(figsize=(12, 6))

# Линия лосса
plt.plot(iterations, loss_values, color='blue', linestyle='-',
    label='Функция потерь (Loss)')

# Точки шагов SGD
plt.scatter(iterations, loss_values, color='red', s=20, label
    ='Шаги SGD (итерации)')

# Стрелки градиентного спуска (опционально)
for i in range(1, len(iterations)):
    plt.annotate('',
                 xy=(iterations[i], loss_values[i]),
```

```
                    xytext=(iterations[i - 1], loss_values[i -
                        1]),
                    arrowprops=dict(arrowstyle='->', color='gray
                        ', lw=1),
                    annotation_clip=False)

# Оформление
plt.title("График функции потерь и шагов градиентного спуска (
    по итерациям)")
plt.xlabel("Итерация")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Листинг П1.3

language_indent_Adam.py

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")

# Загрузка датасета
import kagglehub
path = kagglehub.dataset_download("zarajamshaid/language-
    identification-datasst")
print("Path to dataset files:", path)

# Чтение CSV
import pandas as pd
data = pd.read_csv(path + "/dataset.csv", sep=',')
print("Пропущенные значения:", data.columns[data.isna().any()
    ].tolist())

# Кодируем языки
data["language"] = data["language"].astype('category').cat.
    codes
language_count = data["language"].nunique()

# 1. collections.Counter
from collections import Counter

def yield_tokens(data_train):
```

```
    for text in data_train:
        yield list(text)

all_tokens = [token for tokens in yield_tokens(data["Text"])
   for token in tokens]
counter = Counter(all_tokens)
specials = ["<unk>"]
vocab = {token: idx for idx, token in enumerate(specials +
   list(counter.keys()))}
default_index = vocab["<unk>"]

# Разделение данных
from sklearn.model_selection import train_test_split

y = data["language"]
X = data["Text"]
X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.3, random_state=42)

# Класс Dataset
import torch
from torch.utils.data import Dataset

class TextDataset(Dataset):
    def __init__(self, data_text, labels):
        self.data = data_text.reset_index(drop=True)
        self.labels = labels.reset_index(drop=True)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        token_indices = tokens_to_indices(list(self.data[idx])
           )
        text_tensor = torch.tensor(token_indices, dtype=torch.
           int64)
        label_tensor = torch.tensor(self.labels[idx], dtype=
           torch.int64)
        return text_tensor, label_tensor

trainset = TextDataset(X_train, y_train)
testset = TextDataset(X_test, y_test)

# Collate-функция
def collate_batch(batch):
```

```
    label_list, text_list, offsets = [], [], [0]
    for text, label in batch:
        label_list.append(label)
        text_list.append(text)
        offsets.append(text.size(0))
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text_list = torch.cat(text_list)
    label_list = torch.tensor(label_list, dtype=torch.int64)
    return label_list, text_list, offsets

# DataLoader
from torch.utils.data import DataLoader

trainloader = DataLoader(trainset, batch_size=8, shuffle=True,
    collate_fn=collate_batch)
testloader = DataLoader(testset, batch_size=8, shuffle=False,
   collate_fn=collate_batch)

# Модель
import torch.nn as nn

vocab_size = len(vocab)
num_class = language_count

class TextClassificationModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, 64)  # бе
           з sparse=True
        self.fc = nn.Linear(64, num_class)

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)

net = TextClassificationModel()

def tokens_to_indices(tokens):
    return [vocab.get(token, default_index) for token in
       tokens]

# Обучение модели
import torch.optim as optim
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
import numpy as np

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)  # Обычно д
   ля Adam берут меньший lr

losses = []
all_losses = []  # (iteration, epoch, loss)
net.train()

iteration = 0

for epoch in range(10):  # или 50
    running_loss = 0.0
    running_corrects = 0
    total = 0

    for labels, inputs, offsets in trainloader:
        optimizer.zero_grad()
        outputs = net(inputs, offsets)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # фиксируем loss на каждой итерации
        all_losses.append((iteration, epoch, loss.item()))
        iteration += 1

        _, preds = torch.max(outputs, 1)
        running_loss += loss.item()
        running_corrects += (preds == labels).sum().item()
        total += labels.size(0)

    print(f"[Epoch {epoch + 1}] Loss: {running_loss:.4f},
       Accuracy: {running_corrects / total:.4f}")
    losses.append(running_loss)

print("Finished Training")

# Тестирование
net.eval()
running_correct = 0
num_of_tests = 0

with torch.no_grad():
```

```
    for labels, inputs, offsets in testloader:
        outputs = net(inputs, offsets)
        _, predicted = torch.max(outputs, 1)
        running_correct += (predicted == labels).sum().item()
        num_of_tests += labels.size(0)

print(f"Test Accuracy: {running_correct / num_of_tests:.4f}")

# Первый график

import matplotlib.pyplot as plt
import numpy as np

epochs = np.arange(1, len(losses) + 1)
loss_values = np.array(losses)

plt.figure(figsize=(10, 6))

# Основная линия функция потерь
plt.plot(epochs, loss_values, color='blue', linestyle='-',
   label='Loss')

# Точки градиентного спуска
plt.scatter(epochs, loss_values, color='red', zorder=5, label
   ='Adam')

# Стрелки (градиентный спуск)
for i in range(1, len(epochs)):
    plt.annotate('',
                 xy=(epochs[i], loss_values[i]),
                 xytext=(epochs[i-1], loss_values[i-1]),
                 arrowprops=dict(arrowstyle='->', color='gray
                    ', lw=1.5))

# Настройки графика
plt.title("Функция потерь и путь градиентного спуска")
plt.xlabel("Эпоха")
plt.ylabel("Loss")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Второй график
```

```
import matplotlib.pyplot as plt
import numpy as np

# Извлекаем значения
iterations, epochs_list, loss_values = zip(*all_losses)
iterations = np.array(iterations)
loss_values = np.array(loss_values)

plt.figure(figsize=(12, 6))

# Линия лосса
plt.plot(iterations, loss_values, color='blue', linestyle='-',
    label='Функция потерь (Loss)')

# Точки шагов SGD
plt.scatter(iterations, loss_values, color='red', s=20, label
   ='Шаги Adam (итерации)')

# Стрелки градиентного спуска (опционально)
for i in range(1, len(iterations)):
    plt.annotate('',
                 xy=(iterations[i], loss_values[i]),
                 xytext=(iterations[i - 1], loss_values[i -
                    1]),
                 arrowprops=dict(arrowstyle='->', color='gray
                    ', lw=1),
                 annotation_clip=False)

# Оформление
plt.title("График функции потерь и шагов градиентного спуска (
   по итерациям)")
plt.xlabel("Итерация")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Листинг П1.4

language_indent_adaboost.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
```

```
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from collections import Counter
from sklearn.model_selection import train_test_split
import pandas as pd
import kagglehub

device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")

# --- Загрузка датасета ---
path = kagglehub.dataset_download("zarajamshaid/language-
   identification-datasst")
print("Path to dataset files:", path)

data = pd.read_csv(path + "/dataset.csv", sep=',')
print("Пропущенные значения:", data.columns[data.isna().any()
   ].tolist())

# Кодируем языки
data["language"] = data["language"].astype('category').cat.
   codes
language_count = data["language"].nunique()

# Создание словаря и индексов
def yield_tokens(data_train):
    for text in data_train:
        yield list(text)

all_tokens = [token for tokens in yield_tokens(data["Text"])
   for token in tokens]
counter = Counter(all_tokens)
specials = ["<unk>"]
vocab = {token: idx for idx, token in enumerate(specials +
   list(counter.keys()))}
default_index = vocab["<unk>"]

def tokens_to_indices(tokens):
    return [vocab.get(token, default_index) for token in
       tokens]

# Разделение данных
y = data["language"]
X = data["Text"]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)

# Класс Dataset
class TextDataset(Dataset):
    def __init__(self, data_text, labels):
        self.data = data_text.reset_index(drop=True)
        self.labels = labels.reset_index(drop=True)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        token_indices = tokens_to_indices(list(self.data[idx])
            )
        text_tensor = torch.tensor(token_indices, dtype=torch.
            int64)
        label_tensor = torch.tensor(self.labels[idx], dtype=
            torch.int64)
        return text_tensor, label_tensor

trainset = TextDataset(X_train, y_train)
testset = TextDataset(X_test, y_test)

# Collate-функция
def collate_batch(batch):
    label_list, text_list, offsets = [], [], [0]
    for text, label in batch:
        label_list.append(label)
        text_list.append(text)
        offsets.append(text.size(0))
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text_list = torch.cat(text_list)
    label_list = torch.tensor(label_list, dtype=torch.int64)
    return label_list, text_list, offsets

# DataLoader (shuffle=False для train, важный момент для весов
    AdaBoost)
trainloader = DataLoader(trainset, batch_size=8, shuffle=False
    , collate_fn=collate_batch)
testloader = DataLoader(testset, batch_size=8, shuffle=False,
    collate_fn=collate_batch)

# Модель
vocab_size = len(vocab)
```

```
num_class = language_count

class TextClassificationModel(nn.Module):
    def __init__(self, vocab_size, num_class):
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, 64,
           sparse=False)  # sparse=False здесь
        self.fc = nn.Linear(64, num_class)

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)

class AdaBoostNN:
    def __init__(self, n_estimators, vocab_size, num_class,
       device):
        self.n_estimators = n_estimators
        self.models = []
        self.alphas = []
        self.vocab_size = vocab_size
        self.num_class = num_class
        self.device = device

    def fit(self, trainloader, testloader):
        n_samples = len(trainloader.dataset)
        sample_weights = torch.ones(n_samples) / n_samples

        criterion = nn.CrossEntropyLoss(reduction='none')
        num_epochs_per_estimator = 5

        train_losses, train_accuracies, test_accuracies = [],
           [], []

        for m in range(self.n_estimators):
            net = TextClassificationModel(self.vocab_size,
               self.num_class).to(self.device)
            optimizer = optim.Adam(net.parameters(), lr=0.001)

            for epoch in range(num_epochs_per_estimator):
                net.train()
                idx_sample = 0
                running_loss = 0
                running_correct = 0
                total_samples = 0
```

```
            for labels, inputs, offsets in trainloader:
                optimizer.zero_grad()
                labels = labels.to(self.device)
                inputs = inputs.to(self.device)
                offsets = offsets.to(self.device)

                outputs = net(inputs, offsets)
                losses = criterion(outputs, labels)

                batch_weights = sample_weights[idx_sample:
                    idx_sample + labels.size(0)].to(self.
                    device)
                batch_weights = batch_weights /
                    batch_weights.sum()  # нормализуем внут
                    ри батча

                loss = (losses * batch_weights).sum()
                loss.backward()
                optimizer.step()

                running_loss += loss.item() * labels.size
                    (0)
                _, preds = torch.max(outputs, 1)
                running_correct += (preds == labels).sum()
                    .item()
                total_samples += labels.size(0)
                idx_sample += labels.size(0)

        train_loss = running_loss / total_samples
        train_acc = running_correct / total_samples

        # Подсчёт ошибок для обновления весов
        net.eval()
        incorrect_mask = torch.zeros(n_samples, dtype=
            torch.bool)
        with torch.no_grad():
            idx_sample = 0
            for labels, inputs, offsets in trainloader:
                labels = labels.to(self.device)
                inputs = inputs.to(self.device)
                offsets = offsets.to(self.device)

                outputs = net(inputs, offsets)
                preds = torch.argmax(outputs, dim=1)
```

```
                    batch_size = labels.size(0)
                    incorrect_mask[idx_sample:idx_sample +
                        batch_size] = (preds.cpu() != labels.
                        cpu())
                    idx_sample += batch_size

            err_m = torch.sum(sample_weights * incorrect_mask.
                float()) / torch.sum(sample_weights)
            alpha_m = 0.5 * torch.log((1 - err_m) / (err_m + 1
                e-10))

            sample_weights = sample_weights * torch.exp(
                alpha_m * incorrect_mask.float())
            sample_weights = sample_weights / torch.sum(
                sample_weights)

            self.models.append(net)
            self.alphas.append(alpha_m.item())

            test_acc = self.evaluate(testloader)
            train_losses.append(train_loss)
            train_accuracies.append(train_acc)
            test_accuracies.append(test_acc)

            print(f"Estimator {m + 1}/{self.n_estimators} Loss
                : {train_loss:.4f} Train Acc: {train_acc:.4f}
                Test Acc: {test_acc:.4f}")

        return train_losses, train_accuracies, test_accuracies

    def evaluate(self, dataloader):
        correct = 0
        total = 0
        for labels, inputs, offsets in dataloader:
            labels = labels.to(self.device)
            inputs = inputs.to(self.device)
            offsets = offsets.to(self.device)
            preds = self.predict_single_batch(inputs, offsets)
            correct += (preds == labels).sum().item()
            total += labels.size(0)
        return correct / total

    def predict_single_batch(self, inputs, offsets):
        total_preds = None
        with torch.no_grad():
```

```
            for alpha, model in zip(self.alphas, self.models):
                outputs = model(inputs, offsets)
                probs = torch.softmax(outputs, dim=1)
                weighted_probs = probs * alpha
                if total_preds is None:
                    total_preds = weighted_probs
                else:
                    total_preds += weighted_probs
        return torch.argmax(total_preds, dim=1)

    def predict(self, dataloader):
        all_preds = []
        with torch.no_grad():
            for labels, inputs, offsets in dataloader:
                inputs = inputs.to(self.device)
                offsets = offsets.to(self.device)
                preds = self.predict_single_batch(inputs,
                   offsets)
                all_preds.append(preds.cpu())
        return torch.cat(all_preds)


 # --- Запуск обучения AdaBoost с нейросетью ---

ada = AdaBoostNN(n_estimators=10, vocab_size=vocab_size,
   num_class=num_class, device=device)

train_losses, train_accs, test_accs = ada.fit(trainloader,
   testloader)

# Итоговая точность на тесте
preds = ada.predict(testloader)
labels = torch.cat([labels for labels, _, _ in testloader])
accuracy = (preds == labels).float().mean().item()
print(f"Итоговая точность AdaBoost: {accuracy:.4f}")

import matplotlib.pyplot as plt
import numpy as np

epochs = np.arange(1, len(train_losses) + 1)
loss_values = np.array(train_losses)

plt.figure(figsize=(10, 6))

# Основная линия функция потерь
```

```
plt.plot(epochs, loss_values, color='blue', linestyle='-',
    label='Loss')

# Точки (итерации AdaBoost)
plt.scatter(epochs, loss_values, color='red', zorder=5, label
    ='AdaBoost Step')

# Стрелки движение loss
for i in range(1, len(epochs)):
    plt.annotate('',
                 xy=(epochs[i], loss_values[i]),
                 xytext=(epochs[i - 1], loss_values[i - 1]),
                 arrowprops=dict(arrowstyle='->', color='gray
                     ', lw=1.5))

# Оформление графика
plt.title("Функция потерь и шаги AdaBoost")
plt.xlabel("Итерация (число базовых моделей)")
plt.ylabel("Loss")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()


epochs = np.arange(1, len(train_accs) + 1)
train_vals = np.array(train_accs)
test_vals = np.array(test_accs)

plt.figure(figsize=(10, 6))

# Линии точности
plt.plot(epochs, train_vals, color='green', linestyle='-',
    label='Train Accuracy')
plt.plot(epochs, test_vals, color='orange', linestyle='-',
    label='Test Accuracy')

# Точки
plt.scatter(epochs, train_vals, color='green', s=30, zorder=5)
plt.scatter(epochs, test_vals, color='orange', s=30, zorder=5)

# Стрелки на тестовой метрике
for i in range(1, len(epochs)):
    plt.annotate('',
                 xy=(epochs[i], test_vals[i]),
```

```
                xytext=(epochs[i - 1], test_vals[i - 1]),
                arrowprops=dict(arrowstyle='->', color='gray
                    ', lw=1.5))

# Оформление графика
plt.title("Динамика Accuracy по шагам AdaBoost")
plt.xlabel("Итерация (число базовых моделей)")
plt.ylabel("Accuracy")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Листинг П1.5

sint1.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import torch
from torch.optim import Adam, SGD
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor

# Функция Растригина
def rastrigin(X, Y, A=10):
    return A * 2 + (X**2 - A * np.cos(2 * np.pi * X)) + (Y**2
        - A * np.cos(2 * np.pi * Y))

# Генерация ландшафта
w1_range = np.linspace(-5.12, 5.12, 200)
w2_range = np.linspace(-5.12, 5.12, 200)
W1, W2 = np.meshgrid(w1_range, w2_range)
loss = rastrigin(W1, W2)

# Модель
class OptimizationModel(torch.nn.Module):
    def __init__(self, init_w1, init_w2):
        super().__init__()
        self.w1 = torch.nn.Parameter(torch.tensor([init_w1]))
        self.w2 = torch.nn.Parameter(torch.tensor([init_w2]))

    def forward(self):
        A = 10
```

```
        return A * 2 + (self.w1**2 - A * torch.cos(2 * np.pi *
            self.w1)) + \
                (self.w2**2 - A * torch.cos(2 * np.pi * self.w2
                    ))

# Градиентные методы с рестартами
def optimize_with_restarts(optimizer_type='sgd', restarts=5,
    lr=0.01, n_steps=100):
    best_result = None
    for _ in range(restarts):
        init_w1 = np.random.uniform(-5.12, 5.12)
        init_w2 = np.random.uniform(-5.12, 5.12)
        model = OptimizationModel(init_w1, init_w2)

        if optimizer_type == 'sgd':
            optimizer = SGD(model.parameters(), lr=lr,
                momentum=0.9)
        elif optimizer_type == 'adam':
            optimizer = Adam(model.parameters(), lr=lr)

        history = {'w1': [], 'w2': [], 'loss': []}
        for _ in range(n_steps):
            loss_val = model()
            optimizer.zero_grad()
            loss_val.backward()
            optimizer.step()
            history['w1'].append(model.w1.item())
            history['w2'].append(model.w2.item())
            history['loss'].append(loss_val.item())

        if best_result is None or history['loss'][-1] <
            best_result['loss'][-1]:
            best_result = history
    return best_result

# AdaBoost
def optimize_adaboost(n_estimators=100):
    X = np.random.uniform(-5.12, 5.12, (5000, 2))
    y = rastrigin(X[:, 0], X[:, 1])

    model = AdaBoostRegressor(
        DecisionTreeRegressor(max_depth=5),
        n_estimators=n_estimators,
        learning_rate=0.3
    )
```

```
    model.fit(X, y)

    grid = np.mgrid[-5.12:5.12:0.05, -5.12:5.12:0.05].reshape
       (2, -1).T
    preds = model.predict(grid)
    best_idx = np.argmin(preds)
    best_w1, best_w2 = grid[best_idx]

    return {'w1': [best_w1], 'w2': [best_w2], 'loss': [preds[
       best_idx]]}

# Оптимизация
hist_sgd = optimize_with_restarts('sgd', restarts=10, lr=0.01,
    n_steps=150)
hist_adam = optimize_with_restarts('adam', restarts=10, lr
   =0.01, n_steps=150)
hist_adaboost = optimize_adaboost()

# Визуализация
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

# Возвращаем нормальную поверхность
ax.plot_surface(W1, W2, loss, cmap='viridis', alpha=0.2,
   antialiased=True)

# Функция отрисовки траектории
def plot_trajectory(ax, hist, color, label, final_marker='o',
   marker_size=80):
    ax.plot(hist['w1'], hist['w2'], hist['loss'],
            color=color, linewidth=3.0, alpha=0.95, label=
               label)

    ax.scatter(hist['w1'][-1], hist['w2'][-1], hist['loss
       '][-1],
               c=color, marker=final_marker, s=marker_size,
                  edgecolor='black')

# Траектории
plot_trajectory(ax, hist_sgd, 'red', 'SGD (best restart)', 'o
   ', 80)
plot_trajectory(ax, hist_adam, 'blue', 'Adam (best restart)',
   'o', 80)
plot_trajectory(ax, hist_adaboost, 'green', 'AdaBoost', '*',
   120)
```

```
# Подписи
ax.set_xlabel('Weight 1 (w1)', fontsize=14)
ax.set_ylabel('Weight 2 (w2)', fontsize=14)
ax.set_zlabel('Loss', fontsize=14)
ax.legend(fontsize=14)
plt.title('Оптимизация функции Растригина: SGD, Adam и
   AdaBoost', fontsize=16)
plt.tight_layout()
plt.show()

# === Проверка, куда попали оптимизаторы ===
print("SGD final:")
print(f"  w1 = {hist_sgd['w1'][-1]:.4f}, w2 = {hist_sgd['w2
   '][-1]:.4f}, loss = {hist_sgd['loss'][-1]:.4f}")

print("Adam final:")
print(f"  w1 = {hist_adam['w1'][-1]:.4f}, w2 = {hist_adam['w2
   '][-1]:.4f}, loss = {hist_adam['loss'][-1]:.4f}")

print("AdaBoost final:")
print(f"  w1 = {hist_adaboost['w1'][0]:.4f}, w2 = {
   hist_adaboost['w2'][0]:.4f}, loss = {hist_adaboost['loss
   '][0]:.4f}")
```

Листинг П1.6

sint2.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import torch
from torch.optim import Adam, SGD
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor

# Функция Экли (Ackley function)
def ackley(X, Y, a=20, b=0.2, c=2 * np.pi):
    term1 = -a * np.exp(-b * np.sqrt(0.5 * (X**2 + Y**2)))
    term2 = -np.exp(0.5 * (np.cos(c * X) + np.cos(c * Y)))
    return term1 + term2 + a + np.exp(1)

# Генерация ландшафта
w1_range = np.linspace(-5, 5, 200)
w2_range = np.linspace(-5, 5, 200)
```

```
W1, W2 = np.meshgrid(w1_range, w2_range)
loss = ackley(W1, W2)

# Модель
class OptimizationModel(torch.nn.Module):
    def __init__(self, init_w1, init_w2):
        super().__init__()
        self.w1 = torch.nn.Parameter(torch.tensor([init_w1],
           dtype=torch.float32))
        self.w2 = torch.nn.Parameter(torch.tensor([init_w2],
           dtype=torch.float32))

    def forward(self):
        a = 20
        b = 0.2
        c = 2 * np.pi
        term1 = -a * torch.exp(-b * torch.sqrt(0.5 * (self.w1
           **2 + self.w2**2)))
        term2 = -torch.exp(0.5 * (torch.cos(c * self.w1) +
           torch.cos(c * self.w2)))
        return term1 + term2 + a + np.exp(1)

# Градиентные методы с рестартами
def optimize_with_restarts(optimizer_type='sgd', restarts=5,
   lr=0.01, n_steps=100):
    best_result = None
    for _ in range(restarts):
        init_w1 = np.random.uniform(-5, 5)
        init_w2 = np.random.uniform(-5, 5)
        model = OptimizationModel(init_w1, init_w2)

        if optimizer_type == 'sgd':
            optimizer = SGD(model.parameters(), lr=lr,
               momentum=0.9)
        elif optimizer_type == 'adam':
            optimizer = Adam(model.parameters(), lr=lr)

        history = {'w1': [], 'w2': [], 'loss': []}
        for _ in range(n_steps):
            loss_val = model()
            optimizer.zero_grad()
            loss_val.backward()
            optimizer.step()
            history['w1'].append(model.w1.item())
            history['w2'].append(model.w2.item())
```

```
            history['loss'].append(loss_val.item())

        if best_result is None or history['loss'][-1] <
           best_result['loss'][-1]:
            best_result = history
    return best_result

# AdaBoost
def optimize_adaboost(n_estimators=100):
    X = np.random.uniform(-5, 5, (5000, 2))
    y = ackley(X[:, 0], X[:, 1])

    model = AdaBoostRegressor(
        DecisionTreeRegressor(max_depth=5),
        n_estimators=n_estimators,
        learning_rate=0.3
    )
    model.fit(X, y)

    grid = np.mgrid[-5:5:0.05, -5:5:0.05].reshape(2, -1).T
    preds = model.predict(grid)
    best_idx = np.argmin(preds)
    best_w1, best_w2 = grid[best_idx]

    return {'w1': [best_w1], 'w2': [best_w2], 'loss': [preds[
       best_idx]]}

# Оптимизация
hist_sgd = optimize_with_restarts('sgd', restarts=10, lr=0.01,
    n_steps=150)
hist_adam = optimize_with_restarts('adam', restarts=10, lr
   =0.01, n_steps=150)
hist_adaboost = optimize_adaboost()

# Визуализация
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

# Поверхность функции
ax.plot_surface(W1, W2, loss, cmap='viridis', alpha=0.2,
   antialiased=True)

# Функция отрисовки траектории
def plot_trajectory(ax, hist, color, label, final_marker='o',
   marker_size=80):
```

```
    ax.plot(hist['w1'], hist['w2'], hist['loss'],
            color=color, linewidth=3.0, alpha=0.95, label=
               label)

    ax.scatter(hist['w1'][-1], hist['w2'][-1], hist['loss
       '][-1],
               c=color, marker=final_marker, s=marker_size,
                  edgecolor='black')

# Траектории
plot_trajectory(ax, hist_sgd, 'red', 'SGD (best restart)', 'o
   ', 80)
plot_trajectory(ax, hist_adam, 'blue', 'Adam (best restart)',
   'o', 80)
plot_trajectory(ax, hist_adaboost, 'green', 'AdaBoost', '*',
   120)

# Подписи
ax.set_xlabel('Weight 1 (w1)', fontsize=14)
ax.set_ylabel('Weight 2 (w2)', fontsize=14)
ax.set_zlabel('Loss', fontsize=14)
ax.legend(fontsize=14)
plt.title('Оптимизация функции Экли: SGD, Adam и AdaBoost',
   fontsize=16)
plt.tight_layout()
plt.show()

# === Проверка, куда попали оптимизаторы ===
print("SGD final:")
print(f"  w1 = {hist_sgd['w1'][-1]:.4f}, w2 = {hist_sgd['w2
   '][-1]:.4f}, loss = {hist_sgd['loss'][-1]:.4f}")

print("Adam final:")
print(f"  w1 = {hist_adam['w1'][-1]:.4f}, w2 = {hist_adam['w2
   '][-1]:.4f}, loss = {hist_adam['loss'][-1]:.4f}")

print("AdaBoost final:")
print(f"  w1 = {hist_adaboost['w1'][0]:.4f}, w2 = {
   hist_adaboost['w2'][0]:.4f}, loss = {hist_adaboost['loss
   '][0]:.4f}")
```

Листинг П1.7

sint3.py

```
import numpy as np
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import torch
from torch.optim import Adam, SGD
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor

# Функция Розенброка
def rosenbrock(X, Y, a=1, b=100):
    return (a - X)**2 + b * (Y - X**2)**2

# Генерация ландшафта
w1_range = np.linspace(-2, 2, 200)
w2_range = np.linspace(-1, 3, 200)
W1, W2 = np.meshgrid(w1_range, w2_range)
loss = rosenbrock(W1, W2)

# Модель
class OptimizationModel(torch.nn.Module):
    def __init__(self, init_w1, init_w2):
        super().__init__()
        self.w1 = torch.nn.Parameter(torch.tensor([init_w1],
           dtype=torch.float32))
        self.w2 = torch.nn.Parameter(torch.tensor([init_w2],
           dtype=torch.float32))

    def forward(self):
        a = 1
        b = 100
        return (a - self.w1)**2 + b * (self.w2 - self.w1**2)
           **2

# Градиентные методы с рестартами
def optimize_with_restarts(optimizer_type='sgd', restarts=5,
   lr=0.001, n_steps=2000):
    best_result = None
    for _ in range(restarts):
        init_w1 = np.random.uniform(-2, 2)
        init_w2 = np.random.uniform(-1, 3)
        model = OptimizationModel(init_w1, init_w2)

        if optimizer_type == 'sgd':
            optimizer = SGD(model.parameters(), lr=lr,
               momentum=0.9)
        elif optimizer_type == 'adam':
```

```
            optimizer = Adam(model.parameters(), lr=lr)

        history = {'w1': [], 'w2': [], 'loss': []}
        for _ in range(n_steps):
            loss_val = model()
            optimizer.zero_grad()
            loss_val.backward()
            optimizer.step()
            history['w1'].append(model.w1.item())
            history['w2'].append(model.w2.item())
            history['loss'].append(loss_val.item())

        if best_result is None or history['loss'][-1] <
           best_result['loss'][-1]:
            best_result = history
    return best_result

# AdaBoost
def optimize_adaboost(n_estimators=100):
    X = np.random.uniform(low=[-2, -1], high=[2, 3], size
       =(5000, 2))
    y = rosenbrock(X[:, 0], X[:, 1])

    model = AdaBoostRegressor(
        DecisionTreeRegressor(max_depth=5),
        n_estimators=n_estimators,
        learning_rate=0.3
    )
    model.fit(X, y)

    grid_x, grid_y = np.mgrid[-2:2:0.05, -1:3:0.05]
    grid = np.vstack([grid_x.ravel(), grid_y.ravel()]).T
    preds = model.predict(grid)
    best_idx = np.argmin(preds)
    best_w1, best_w2 = grid[best_idx]

    return {'w1': [best_w1], 'w2': [best_w2], 'loss': [preds[
       best_idx]]}

# Оптимизация
hist_sgd = optimize_with_restarts('sgd', restarts=10, lr
   =0.001, n_steps=2000)
hist_adam = optimize_with_restarts('adam', restarts=10, lr
   =0.001, n_steps=2000)
hist_adaboost = optimize_adaboost()
```

```
# Визуализация
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

# Поверхность функции
ax.plot_surface(W1, W2, loss, cmap='viridis', alpha=0.2,
    antialiased=True)

# Функция отрисовки траектории
def plot_trajectory(ax, hist, color, label, final_marker='o',
    marker_size=80):
    ax.plot(hist['w1'], hist['w2'], hist['loss'],
            color=color, linewidth=3.0, alpha=0.95, label=
               label)

    ax.scatter(hist['w1'][-1], hist['w2'][-1], hist['loss
       '][-1],
               c=color, marker=final_marker, s=marker_size,
                  edgecolor='black')

# Траектории
plot_trajectory(ax, hist_sgd, 'red', 'SGD (best restart)', 'o
    ', 80)
plot_trajectory(ax, hist_adam, 'blue', 'Adam (best restart)',
    'o', 80)
plot_trajectory(ax, hist_adaboost, 'green', 'AdaBoost', '*',
    120)

# Подписи
ax.set_xlabel('Weight 1 (w1)', fontsize=14)
ax.set_ylabel('Weight 2 (w2)', fontsize=14)
ax.set_zlabel('Loss', fontsize=14)
ax.legend(fontsize=14)
plt.title('Оптимизация функции Розенброка: SGD, Adam и
    AdaBoost', fontsize=16)
plt.tight_layout()
plt.show()

# === Проверка, куда попали оптимизаторы ===
print("SGD final:")
print(f"  w1 = {hist_sgd['w1'][-1]:.4f}, w2 = {hist_sgd['w2
    '][-1]:.4f}, loss = {hist_sgd['loss'][-1]:.4f}")

print("Adam final:")
```

```
print(f"  w1 = {hist_adam['w1'][-1]:.4f}, w2 = {hist_adam['w2
    '][-1]:.4f}, loss = {hist_adam['loss'][-1]:.4f}")

print("AdaBoost final:")
print(f"  w1 = {hist_adaboost['w1'][0]:.4f}, w2 = {
    hist_adaboost['w2'][0]:.4f}, loss = {hist_adaboost['loss
    '][0]:.4f}")
```