Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.М04-мм

Оптимизационные алгоритмы для нейросетевых гидроаэромеханических расчётов

Егоров Павел Алексеевич

Отчёт по преддипломной практике
в форме «Производственное задание»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н. Гориховский В. И.

Санкт-Петербург
2024

Оглавление

Введение

На сегодняшний день машинное обучение и нейронные сети стали неотъемлемой частью современной науки. Они позволяют анализировать огромные объемы данных, выявлять скрытые закономерности в них и строить достаточно точные прогнозы. Это важно, поскольку данные, часто слишком сложные для восприятия человеком, могут быть эффективно обработаны и проанализированы компьютерами с использованием искусственного интеллекта.

Применение методов машинного обучения и нейронных сетей позволяет решать разнообразные сложные задачи, включая задачи гидроаэромеханики. В контексте этой дисциплины, машинное обучение позволяет автоматизировать процессы анализа данных, прогнозирование результатов и создание высокоэффективных моделей, что в свою очередь улучшает понимание процессов и явлений, которые ранее могли быть сложными для восприятия или анализа. К подобным задачам можно отнести моделирование высокоскоростных течений, например, при обтекании сферы пятикомпонентной воздушной смесью [34], или подбор оптимальной аэромеханической формы тела [17]. Также одной из перспективных задач является расчет поуровневых коэффициентов скорости колебательных энергообменов. В работе [33] было предложено решение расчета с помощью регрессионных аппроксимаций, однако, моделирование можно провести с использованием нейросетевых технологий.

Важным аспектом машинного обучения является применение оптимизационных алгоритмов. Это необходимо при решении различных типов задач с использованием нейронных сетей, поскольку оптимизаторы позволяют настраивать параметры модели таким образом, чтобы минимизировать функцию потерь. Это позволяет улучшить производительность моделей и повысить точность их прогнозирования. Существуют различные подходы для решения задач оптимизации. В нейронных сетях, как правило, используют алгоритмы, основанные на вычислении градиента или гессиана оптимизируемой функции. К алгоритмам

первого порядка, которые используют градиент для вычисления локального минимума, можно отнести стохастический градиентный спуск (SGD) [20], Adam [15] и т.д. Методы второго порядка для оптимизации используют матрицу Гессе, также к ним относятся квазиньютоновские алгоритмы, основанные на приближенных выражениях для матрицы Гессе, например, BFGS [9].

Одним из распространенных подходов для разработки интерактивных веб-приложений, предназначенных для анализа данных и применения методов машинного обучения, является использование фреймворка Streamlit [28]. С помощью данного инструмента на кафедре гидроаэромеханики было разработано приложение machine-learning-ui в рамках исследовательского проекта, финансируемого Санкт-Петербургским государственным университетом[1], для решения задач неравновесной газовой динамики с использованием методов машинного обучения. Целью настоящей работы является реализация и интеграция в приложение machine-learning-ui оптимизационных алгоритмов AdaMod [4], MADGRAD [6], RAdam [24], Apollo [19], AdaHessian [1], LARS [30], LAMB [16], их сравнение и решение задачи расчета поуровневых коэффициентов скорости колебательных энергообменов с помощью нейросетевой модели с различными оптимизаторами.

[1]M1_2021 - 3: Машинное обучение в задачах неравновесной аэромеханики: 2023 г. этап 3

1. Постановка задачи

Целью данной работы является реализация и интеграция в приложение machine-learning-ui оптимизационных алгоритмов. Для выполнения данной цели были поставлены следующие задачи:

1. выполнить обзор алгоритмов и существующих решений;
2. реализовать и интегрировать в machine-learning-ui оптимизационные алгоритмы;
3. провести сравнение эффективности и точности реализованных алгоритмов;
4. выполнить расчет поуровневых коэффициентов скорости колебательных энергообменов с применением нейросетевого подхода.

2. Обзор

2.1. Оптимизация в ML

Одной из ключевых задач в области нейронных сетей является оптимизация. Она заключается в поиске оптимальных параметров модели, которые минимизируют функцию потерь и улучшают качество предсказаний. Формально, задачу оптимизации в нейронных сетях можно сформулировать следующим образом:

Определение 1. *Пусть дана нейронная сеть с параметрами* θ *и функцией потерь* $L(\theta)$*. Требуется найти такие значения параметров* $\hat{\theta}$*, которые минимизируют функцию потерь:* $\hat{\theta} = min(L(\theta))$*.*

При решении задачи оптимизации мы можем разделить все функции на два условных класса: выпуклые и невыпуклые.

Определение 2. *Функция* $f : [a, b] \to \mathbb{R}$ *называется: выпуклой вниз на* $[a, b]$*, если для любых* $x_1, x_2 \in [a, b]$ *и* $t \in (0, 1)$ *выполянется неравенство* $f(tx_1 + (1 - t)x_2) \leqslant tf(x_1) + (1 - t)f(x_2)$*. [31]*

Важным свойством выпуклых функций является тот факт, что локальный и глобальный минимумы совпадают. Это обстоятельство помогает избежать неприятных ситуаций, которые могут встретиться в невыпуклом случае.

Теорема 1. *Пусть* A *– алгоритм оптимизации, использующий локальную информацию (все производные в точке). Тогда существует такая невыпуклая функция* $f[0, 1]^d \to [0, 1]$*, что для нахождения глобального минимума на квадрате* $[0, 1]^d$ *с точностью* $\frac{1}{m}$ *требуется совершить хотя бы* m^d *шагов. [32]*

Функции потерь при обучении нейронных сетей обычно не являются выпуклыми. Однако это не означает, что любой алгоритм оптимизации будет неэффективен. Поиск глобального минимума невыпуклой функции является сложной задачей, но часто достаточно нахождения

локального минимума, который является стационарной точкой, где производная равна нулю. Теоретические результаты в случае невыпуклых задач обычно связаны с поиском таких точек, и алгоритмы направлены на их обнаружение.

Большинство алгоритмов оптимизации, разработанных для выпуклых случаев, успешно применяются и для невыпуклых задач. В выпуклом случае поиск стационарной точки и минимума являются одной и той же задачей, поэтому методы, эффективные для поиска минимума в выпуклом случае, также хорошо работают для поиска стационарных точек в невыпуклом случае.

На практике необязательно находить глобальный минимум для невыпуклых функций, поскольку это достаточно нетривиальная задача. Вот несколько объяснений данному факту:

- В окрестности локального минимума функция становится выпуклой. Там функция сможет быстро сойтись.
- Некоторые невыпуклые функции достаточно похожи на выпуклые. Например, функция Леви (рис. 1) [32].
- Градиентные методы часто сходятся к локальным минимумам [12].

Одним из классических методов оптимизации является градиентный спуск. Градиент — это вектор, указывающий направление наискорейшего роста функции. Идея метода заключается в использовании антиградиента для движения в направлении локального убывания. Математически задача формулируется следующим образом:

$$x_{i+1} = x_i - \alpha \nabla f(x_i)$$

где f(x) — оптимизируемая функция, α — размер шага или же скорость градиентного спуска.

Обратное распространение ошибки (backpropagation) является ключевым методом вычисления градиента в нейронных сетях. Он основан

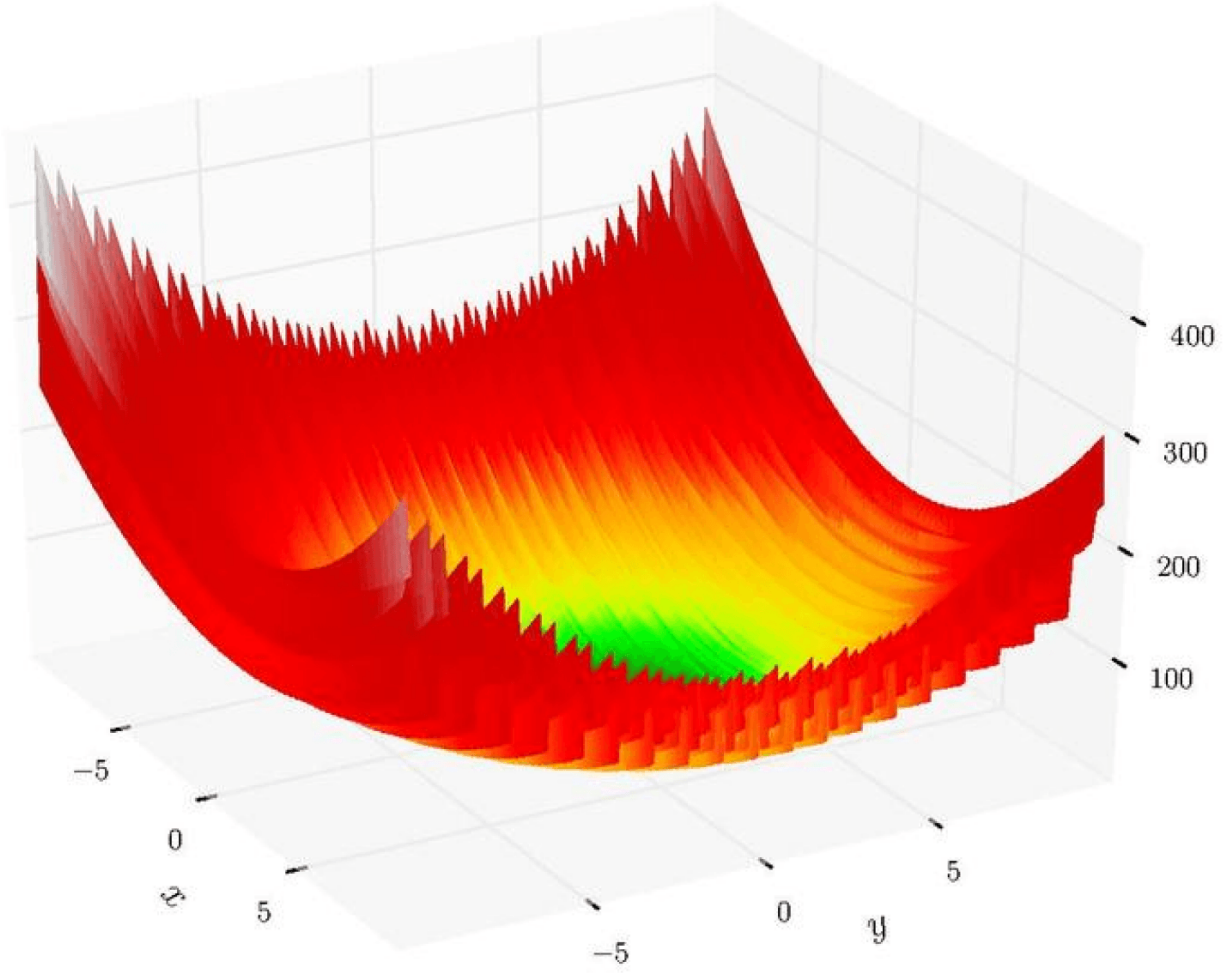

Рис. 1: Функция Леви

на цепном правиле дифференцирования и позволяет эффективно вычислять градиент функции потерь по параметрам каждого слоя сети.

В процессе обратного распространения ошибки градиент функции потерь по выходу сети передается назад через слои сети, вычисляя градиенты по параметрам каждого слоя. Это позволяет использовать градиент для обновления параметров сети в процессе оптимизации.

При обычном решении задачи градиентного спуска для вычисления градиента на одном шаге нужно подать на вход сети весь набор данных, вычислить ошибку для каждого объекта и рассчитать коррекцию коэффициентов сети, при этом выборка может занимать десятки, а то и сотни гигабайт. Такой способ обучения будет очень медленным и вы-

числительно затратным, поэтому на практике рассчитывают не весь градиент, а его аппроксимацию — градиент, вычисленный на случайном подмножестве тестовых данных. Данный метод называется стохастическим градиентным спуском [20].

Существует множество обобщений стохастического градиентного спуска. Одной из ранних доработок алгоритма является добавление импульса [26]. Основная идея SGD с импульсом заключается в том, что на каждой итерации обновления параметров учитывается не только текущий градиент функции потерь, но также предыдущее изменение параметров (импульс):

$$x_{i+1} = x_i - \alpha \nabla f(x_i) + \beta(x_i - x_{i-1})$$

где β — коэффициент импульса.

Понятие «импульс» берет свое начало из физики — импульс помогает двигаться в правильном направлении и ускоряет сходимость к минимуму функции потерь. Коэффициент затухания β действует как сила трения, которая постепенно уменьшает импульс, предотвращая перескакивание модели через минимум.

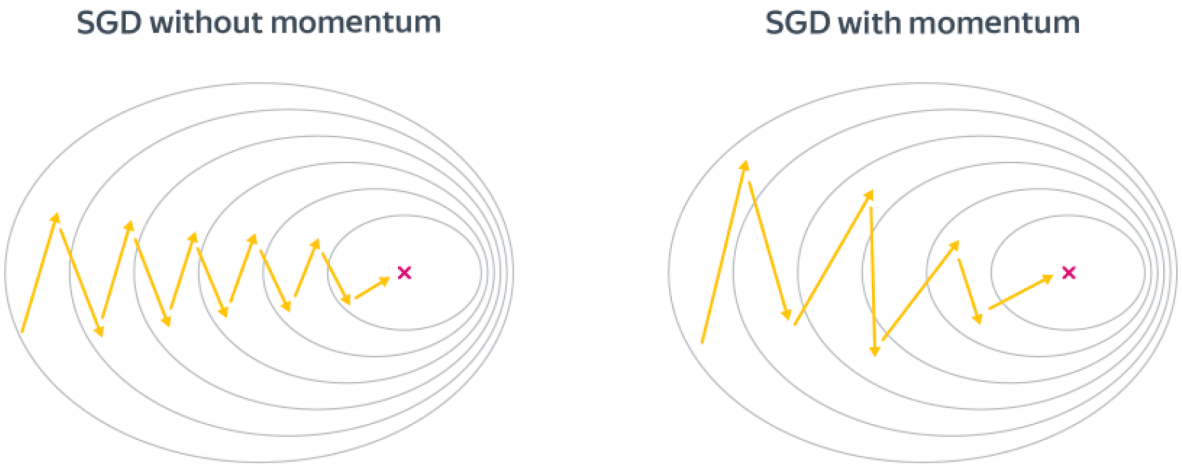

Рис. 2: SGD с импульсом и без импульса [32]

Важным шагом в развитии градиентных методов был алгоритм AdaGrad (Adaptive Gradient algorithm) [7], опубликованный в 2011 году.

Алгоритм адаптирует скорость обучения для каждого параметра модели на основе истории градиентов. Основная идея AdaGrad заключается в том, чтобы уменьшать скорость обучения для параметров, которые имеют большие обновления, и увеличивать скорость обучения для параметров с меньшими обновлениями. Обновление параметров осуществляется по следующей формуле:

$$x_{i+1} = x_i - \frac{\alpha}{\sqrt{G_{i+1}} + \epsilon} \nabla f(x_i)$$

где α — начальная скорость обучения, G_i — сумма квадратов градиентов $\nabla f(x_i)$ на итерации i, ϵ — параметр, который добавляется для обеспечения числовой устойчивости.

Изначально AdaGrad разрабатывался для оптимизации выпуклых функций, но при этом он успешно применяется и в невыпуклых случаях. Основным недостатком данного метода является ситуация, когда размер шага уменьшается слишком быстро. Для решения этой проблемы был разработан другой алгоритм.

RMSProp (Root Mean Square Propagation) [14] — это оптимизационный алгоритм, который используется для обновления весов в глубоком обучении. Он является модификацией стохастического градиентного спуска (SGD) и направлен на устранение проблемы слишком больших или маленьких обновлений весов. Основная идея алгоритма RMSProp заключается в том, что он адаптирует скорость обучения для каждого параметра. Это достигается путем вычисления скользящего среднего квадратов градиентов для каждого параметра. Затем скорость обучения делится на корень из этого среднего, что позволяет уменьшить вклад больших градиентов и увеличить вклад маленьких градиентов.

$$\nu_{i+1} = \gamma v_i + (1 - \gamma) \nabla (f(x_i))^2$$
$$x_{i+1} = x_i - \frac{\alpha}{\sqrt{\nu_{i+1}} + \epsilon}$$

где α — скорость обучения, ν_i — скользящее среднее квадратов градиентов, γ — коэффициент затухания, ϵ — параметр для обеспечения устойчивости.

Алгоритм RMSProp позволяет эффективно обучать модели глубокого обучения, ускоряя сходимость и улучшая качество обучения за счет адаптивного изменения размера шага.

Adam (Adaptive Moment Estimation) [15] — это оптимизационный алгоритм, который является улучшением алгоритма RMSProp. Adam сохраняет экспоненциально затухающие средние градиентов и квадратов градиентов для каждого параметра. На каждой итерации он вычисляет оценки смещенных моментов первого и второго порядков, которые корректируются для смещения на начальных этапах обучения. Формулы обновления весов в алгоритме Adam выглядят следующим образом:

$$
\begin{aligned}
m_{i+1} &= \beta_1 m_i + (1-\beta_1)\nabla f(x_i) \\
v_{i+1} &= \beta_2 v_i + (1-\beta_2)\nabla (f(x_i))^2 \\
\hat{m_{i+1}} &= \frac{m_{i+1}}{1-\beta_1^{i+1}} \\
\hat{v_{i+1}} &= \frac{v_{i+1}}{1-\beta_2^{i+1}} \\
x_{i+1} &= x_i - \alpha \frac{\hat{m_{i+1}}}{\sqrt{\hat{v_{i+1}}} + \epsilon}
\end{aligned}
$$

где α — скорость обучения, β_1, β_2 — коэффициенты сглаживания для первого и второго моментов, ϵ — константа для численной стабильности.

Adam является одним из наиболее популярных алгоритмов оптимизации для обучения нейронных сетей благодаря своей эффективности и способности адаптироваться к различным типам данных и задачам.

Помимо градиентных методов, для оптимизации в машинном обучении используют также методы второго порядка. Они используют информацию о кривизне целевой функции для вычисления направления спуска с высокой точностью, приводя к более быстрой сходимости и лучшей производительности.

В методе Ньютона итеративный процесс имеет вид:

$$x_{i+1} = x_i - (H(x_i))^{-1} \nabla f(x_i)$$

где $H(x_i)$ — гессиан функции $f(x)$.

Ньютоновские методы требуют больше вычислительных ресурсов, чем методы первого порядка, из-за необходимости вычисления и инвертирования гессиана. На практике это зачастую не представляется возможным, поэтому прибегают к использованию квазиньютоновских методов. Основная идея квазиньютоновских методов заключается в том, что вместо вычисления точного гессиана функции на каждой итерации, они аппроксимируют его с помощью некоторой матрицы, которая обновляется на каждом шаге в соответствии с изменениями градиента.

Одним из наиболее известных квазиньютоновских методов является метод BFGS (Broyden-Fletcher-Goldfarb-Shanno) [9]. Он основан на идее аппроксимации гессиана функции с помощью матрицы, которая обновляется на каждом шаге итерации. Еще одним популярным методом является метод L-BFGS(Limited-memory BFGS) [22] — вариация метода BFGS с ограниченной памятью, которая используется в случаях, когда хранение полной матрицы Гессе затруднительно из-за большого размера задачи. L-BFGS хранит только небольшое количество векторов, чтобы приближенно вычислять обратное действие гессиана на вектор. Это позволяет использовать метод L-BFGS для оптимизации задач с большим числом переменных или ограниченной памятью.

2.2. Используемые технологии

Для взаимодействия с моделями машинного обучения и решения практических задач у приложения должен быть удобный пользовательский интерфейс. Для этого при проектировании решений используют фреймворки и библиотеки. В нашем случае, приложение написано с использованием Streamlit [28] — фреймворка Python с открытым исходным кодом, используемого для создания интерактивных веб-приложений для анализа данных и машинного обучения. Данный ин-

струмент позволяет быстро и легко создавать сложные динамичные приложения для визуализации данных, исследовательского анализа и развертывания моделей машинного обучения.

Streamlit основан на концепции интерактивных страниц, которые представляют собой отдельные компоненты приложения с собственной логикой и пользовательским интерфейсом. Страницы определяются с помощью функций Python, которые используют объекты Streamlit для добавления интерактивных элементов, таких как текстовые поля, кнопки, виджеты и диаграммы. Также Streamlit сравнительно прост относительно других веб-фреймворков, поскольку для визуального оформления не требует использования HTML [13] и CSS [5].

В качестве инструмента для создания и обучения нейросетевых моделей в machine-learning-ui используется TensorFlow [10] — библиотека для глубокого обучения с открытым исходным кодом, разработанная компанией Google, которая предоставляет высокоуровневый API для построения различных типов моделей нейронных сетей.

TensorFlow основан на концепции графов вычислений. Граф вычислений представляет собой направленный ациклический граф (DAG), узлы которого представляют собой операции, а ребра — данные, передаваемые между операциями. Граф разбивается на более мелкие подграфы, которые могут выполняться на различных устройствах, например, ЦП или ГП. Библиотека оперирует данными в виде тензоров — многомерных массивов данных, которые используются для представления входных данных, промежуточных значений и выходов моделей.

TensorFlow предоставляет широкий спектр возможностей для разработки и обучения моделей машинного обучения. Среди основных преимуществ TensorFlow можно выделить поддержку различных типов нейронных сетей, автоматическое дифференцирование для оптимизации параметров моделей, интеграцию с различными устройствами и языками программирования, а также возможность распределенного обучения моделей на кластерах серверов.

В силу своей масштабируемости и большого количества доступных инструментов TensorFlow дает возможность создавать собственные ка-

стомизированные решения. Таким образом, реализованные оптимизаторы должны быть подклассами базового класса Optimizer библиотеки TensorFlow. Их интеграция в приложение должна учитывать специфику фреймворка Streamlit.

2.3. Существующие реализации

Набор существующих оптимизационных алгоритмов в библиотеке TensorFlow достаточно сильно ограничен: нет реализаций оптимизаторов второго порядка, а практически все публикации доступных алгоритмов первого порядка датируются до 2018 года включительно.

Некоторые из нужных алгоритмов (LAMB [16], RAdam [24]) были реализованы в TensorFlow Addons [11] — открытой библиотеке дополнений, которую развивали пользователи TensorFlow. Однако, поддержка проекта была прекращена, чтобы предотвратить дублирование с другими модулями. В связи с тем, что TensorFlow Addons работает на более старых версиях TensorFlow, реализации доступных в данной библиотеке оптимизаторов не подходят для внедрения их в приложение, несмотря на их качественное исполнение со стороны специалистов в области машинного обучения.

Существует репозиторий pytorch-optimizer [23] с большим набором оптимизаторов, совместимых с модулем optim фреймворка PyTorch[8]. В данный момент PyTorch является наиболее популярным решением в задачах, связанных с нейросетями и машинным обучением (рис. 2). Поэтому есть большое сообщество, которое развивает данный репозиторий. Но использовать эти алгоритмы не представляется возможным по той причине, что реализуемое приложение использует в своей основе TensorFlow.

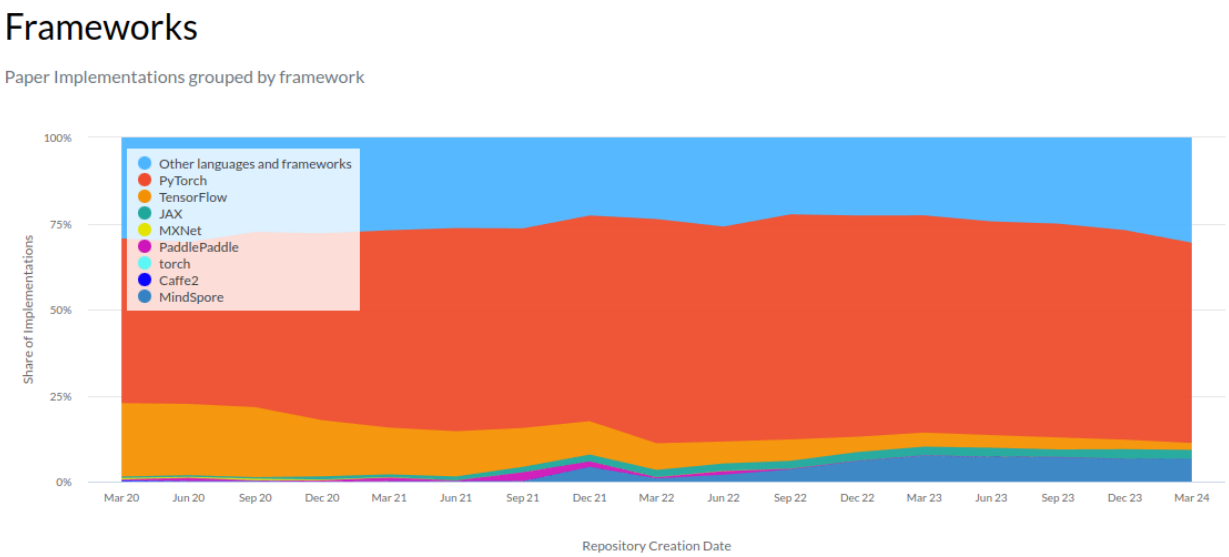

Рис. 3: Тренды наиболее популярных технологий в ML [25]

2.4. Регрессионная задача для расчёта поуровневых коэффициентов скорости колебательных энергообменов

Зачастую в газах могут происходить физико-химические процессы, нарушающие термодинамическое равновесие системы. В этом случае классических уравнений не хватает для описания происходящих процессов, поэтому для решения включают дополнительные кинетические уравнения.

Существует несколько подходов для расчёта коэффициентов скорости колебательных энергообменов: теория Шварца-Славского-Герцфельда (SSH) [27], модель нагруженного гармонического осциллятора (FHO) [29], а также модель гармонического осциллятора с учетом свободных вращений (FHO-FR) [3, 2]. Первые две модели не учитывают всей полноты при описании системы, в связи с чем применение данных подходов может давать некорректные результаты. Модель FHO-FR учитывает трёхмерный характер столкновения частиц и их свободное вращение [33]. Однако, для использования данной модели требуются достаточно большие вычислительные мощности, что не всегда представляется возможным.

В работе [33] было предложено использовать нелинейную регрессию

для аппроксимации модели нагруженного гармонического осциллятора с учетом свободных вращений. По формулам FHO-FR были рассчитаны три набора данных для случаев взаимодействия молекулы O_2 с молекулами партнерами по столкновению (O_2, O, Ar). Для обучения была использована следующая модель нелинейной регрессии:

$$k(T) = A(T^{-\frac{1}{3}})e^{B(T^{-\frac{1}{3}})} + C(T^{-\frac{1}{3}})$$

где A, B и C — некоторые полиномы от температуры газа степени l_A, l_B и l_C соответственно [33]. Применение данного подхода позволило значительно сократить время расчета каждого коэффициента, при этом была достигнута высокая точность предсказаний.

3. Оптимизационные алгоритмы

В ходе работы были реализованы различные методы оптимизации нейронных сетей с использованием Python3 при помощи библиотеки глубокого обучения TensorFlow. Далее все алгоритмы были интегрированы в приложение machine-learning-ui для решения задач гидроаэромеханики, разработанное на основе фреймворка Streamlit. Таким образом, при сборке нейронной сети можно выбрать один из реализованных оптимизаторов для нахождения оптимального набора параметров модели.

3.1. AdaMod

AdaMod (Adaptive and Momental Bound Method) [4] — это адаптивный оптимизационный алгоритм, который является усовершенствованным оптимизатором Adam [15]. Основная идея алгоритма состоит в том, чтобы ограничить неожиданно высокие скорости обучения, особенно на начальных этапах, адаптивными и моментальными верхними границами.

Так же как и в алгоритме Adam, сначала вычисляется градиент, потом первый и второй моменты градиентов как экспоненциальные скользящие средние и их смещение с использованием параметров β_1 и β_2. После этого вычисляется адаптивная скорость обучения и сглаженное значение скорости с помощью параметра β_3, из которых выбирается минимум. Далее происходит обновление параметров модели.

В основе алгоритма лежит адаптивная скорость обучения, которая учитывает как текущий градиент, так и предыдущие обновления параметров. Это позволяет более эффективно настраивать скорость обучения для каждого параметра модели и улучшает сходимость алгоритма.

Ниже представлен псевдокод алгоритма AdaMod.

Algorithm 1 AdaMod

Input: initial parameter θ_0, step sizes $\{\alpha_t\}_{t=1}^{T}$, moment decay $\{\beta_1, \beta_2, \beta_3\}$, regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $m_0 = 0$, $v_0 = 0$, $s_0 = 0$
for $t = 1$ to T **do**
    $g_t = \nabla f_t(\theta_{t-1})$
    $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\hat{m}_t = m_t / (1 - \beta_1^t)$
    $\hat{v}_t = v_t / (1 - \beta_2^t)$
    $\eta_t = \alpha_t / (\sqrt{\hat{v}_t} + \epsilon)$
    $s_t = \beta_3 s_{t-1} + (1 - \beta_3) \eta_t$
    $\hat{\eta}_t = \min(\eta_t, s_t)$
    $\theta_t = \theta_{t-1} - \hat{\eta}_t \hat{m}_t$
end for

3.2. MADGRAD

MADGRAD (Momentumized, Adaptive, Dual Averaged Gradient Method) [6] так же относится к семейству адаптационных алгоритмов оптимизации первого порядка. Разработан компанией Facebook. По словам авторов оригинальной статьи, алгоритм показывает лучшую сходимость по сравнению с классическими алгоритмами SGD и Adam в различных задачах.

На первом шаге вычисляется градиент. Затем масштабируется скорость обучения с учетом текущего шага для численной стабильности. Далее рассчитываются первый и второй моменты градиентов с использованием обновленного шага обучения. Далее происходит двойное усреднение градиентов и обновление весов с добавлением импульса.

Одним из главных преимуществ алгоритма MADGRAD является тот факт, что не требуется особого подбора гиперпараметров как в других оптимизаторах.

Псевдокод алгоритма MADGRAD представлен ниже.

Algorithm 2 MADGRAD

Input: initial parameter θ_0, step sizes $\{\alpha_t\}_{t=1}^{T}$, momentum sequence $\{c_t\}_{t=1}^{T}$, regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $m_0 = 0$, $v_0 = 0$
for $t = 0$ to T **do**
    $g_t = \nabla f_t(\theta_t)$
    $\lambda_t = \alpha_t \sqrt{t+1}$
    $m_{t+1} = m_t + \lambda_t g_t$
    $v_{t+1} = v_t + \lambda_t (g_t * g_t)$
    $z_{k+1} = \theta_0 - \frac{1}{(\sqrt[3]{v_{t+1}} + \epsilon)} * m_{t+1}$
    $\theta_{t+1} = (1 - c_{t+1})\theta_t + c_{t+1} z_{t+1}$
end for

3.3. RAdam

RAdam (Rectified Adam) [24] является еще одним адаптивным алгоритмом оптимизации и пытается решить проблему плохой сходимости оптимизатора Adam из-за высокой дисперсии адаптивной скорости обучения модели.

Так же, как и многие другие алгоритмы первого порядка, RAdam использует первый и второй моменты градиентов. В качестве нововведения появляется приближение экспоненциального скользящего среднего — простое скользящее среднее (simple moving average). Если отклонение приемлемо, то есть длина простого скользящего среднего меньше четырех, то дисперсия адаптивного темпа обучения становится неразрешимой, и адаптивный темп обучения не используется. В противном случае вычисляется член исправления дисперсии(r_t) и обновляются параметры с помощью адаптивной скорости обучения.

Псевдокод алгоритма RAdam представлен ниже.

Algorithm 3 RAdam

Input: initial parameter θ_0, step sizes $\{\alpha_t\}_{t=1}^{T}$, moment decay $\{\beta_1, \beta_2\}$, regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $m_0 = 0$, $v_0 = 0$
$\rho_\infty = 2/(1-\beta_2) - 1$
for $t = 1$ to T **do**
    $g_t = \nabla f_t(\theta_{t-1})$
    $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
    $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
    $\hat{m}_t = m_t / (1-\beta_1^t)$
    $\rho_t = \rho_\infty - 2t\beta_2^t / (1-\beta_2^t)$
    if $\rho_t \geq 4$ **then**
        $l_t = \sqrt{(1-\beta_2^t)/v_t}$
        $r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}$
        $\theta_t = \theta_{t-1} - \alpha_t r_t \hat{m}_t l_t$
    else
        $\theta_t = \theta_{t-1} - \alpha_t \hat{m}_t$
    end if
end for

3.4. Apollo

Apollo (Adaptive Parameter-wise Diagonal Quasi-Newton Method) [19] представляет собой метод оптимизации для невыпуклых стохастических задач, который динамически учитывает форму функции потерь, используя диагональную аппроксимацию гессиана. Этот метод обновляет и хранит диагональную матрицу Гессе эффективно, аналогично адаптивным методам первого порядка, но с линейной сложностью по времени и памяти. Для борьбы с невыпуклостью, гессиан заменяется его абсолютным значением, гарантированно положительно определенным. Результаты экспериментов на задачах компьютерного зрения и обработки естественных языков показывают, что Apollo превосходит другие стохастические методы оптимизации, включая SGD и различные варианты Adam, как по скорости сходимости, так и по обобщающей способности.

Метод является квазиньютоновским, то есть матрица Гессе не вычисляется из-за больших затрат, поэтому берется ее приближение - диа-

гональная аппроксимация. Для снижения дисперсии используется экспоненциальное скользящее среднее с коэффициентом затухания β. Для того, чтобы избежать невыпуклость выбирается максимум между приближением гессиана и коэффициентом выпуклости, равным по умолчанию 0,01.

Ниже представлен псевдокод алгоритма Apollo.

Algorithm 4 Apollo

Input: initial parameter θ_0, step sizes $\{\alpha_t\}_{t=1}^{T}$, moment decay β, regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $m_0 = 0$, $v_0 = 0$, $B_0 = 0$
for $t = 0$ to T **do**
    $g_{t+1} = \nabla f_t(\theta_t)$
    $m_{t+1} = \frac{\beta(1-\beta^t)}{1-\beta^{t+1}} m_t + \frac{1-\beta}{1-\beta^{t+1}} g_{t+1}$
    $a = \frac{d_t^T (m_{t+1} - m_t) + d_t^T B_t d_t}{(||d_t||_4 + \epsilon)^4}$
    $B_{t+1} = B_t - a \cdot Diag(d_t^2)$
    $D_{t+1} = rectify(B_{t+1}, 0.01)$
    $d_{t+1} = D_{t+1}^{-1} m_{t+1}$
    $\theta_{t+1} = \theta_t - \alpha_{t+1} d_{t+1}$
end for

3.5. AdaHessian

AdaHessian (Adaptive Second Order Optimizer) [1] — это адаптивный алгоритм оптимизации второго порядка, так же как и предыдущий алгоритм, динамически учитывающий кривизну функции потерь. В данном методе применяются новые подходы оптимизации, позволяющие избежать всех тех проблем, возникающих при использовании классических методов второго порядка. Во-первых, для аппроксимации матрицы Гессе ее диагональю используется метод Хатчинсона. Его суть заключается в матричном умножении гессиана на случайный вектор из элементов (-1, 1) и последующем поэлементом умнжении результата данного произведения на тот же случайный вектор. Во-вторых, метод использует среднеквадратичное экспоненциальное скользящее среднее для сглаживания изменений диагонали гессиана на разных итерациях.

В-третьих, для уменьшения дисперсии диагональных элементов гессиана используется блочное диагональное усреднение.

AdaHessian, помимо скорости обучения, использует два параметра: β_1 для вычисления скользящего среднего градиентов и β_2 для вычисления скользящего среднего аппроксимации матрицы Гессе.

Ниже приведен псевдокод алгоритма AdaHessian.

Algorithm 5 AdaHessian

Input: initial parameter θ_0, step sizes $\{\alpha_t\}_{t=1}^{T}$, moment decay $\{\beta_1, \beta_2\}$, regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $m_0 = 0$, $v_0 = 0$
for $t = 1$ to T **do**
    $g_t = \nabla f_t(\theta_{t-1})$
    $D_t = Diag(H)$
    $m_t = \frac{(1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} g_i}{1-\beta_1^t}$
    $v_t = \sqrt{\frac{(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} D_i D_i}{1-\beta_2^t}}$
    $\theta_t = \theta_{t-1} - \alpha \frac{m_t}{v_t+\epsilon}$
end for

3.6. LARS

LARS (Layer-wise Adaptive Rate Scaling) [30] — это алгоритм оптимизации, основанный на послойном адаптивном масштабировании скорости обучения. Он опирается на идею о том, что глубокие слои модели имеют различные скорости обучения, поскольку они могут обладать различной чувствительностью к изменению входных данных. LARS является адаптивным методом масштабирования скорости обучения, который позволяет эффективно справляться с проблемой расходимости, часто возникающей при использовании больших размеров батчей.

Алгоритм имеет несколько отличительных особенностей. Во-первых, LARS использует отдельную скорость обучения для каждого слоя, а не для каждого веса, что позволяет эффективнее управлять процессом обучения и предотвращать расходимость. Во-вторых, LARS учитывает норму весов при масштабировании скорости обучения. Это помогает

контролировать скорость обучения в зависимости от степени обновления весов и их норм.

В качестве базового оптимизатора, так же как и в оригинальной статье, используется стохастический градиентный спуск (SGD).

Псевдокод алгоритма LARS представлен ниже.

Algorithm 6 LARS

Input: initial parameter θ_0, base LR γ_0, momentum c, wight decay β, LARS coefficient η regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $v_0 = 0$
for $t = 0$ to T **do**
    $g_t = \nabla f_t(\theta_t)$
    $\gamma_t = \gamma_0(1 - t/T)^2$
    $\lambda = \frac{||\theta_t||}{||g_t||+\beta||\theta||}$
    $v_{t+1} = v_t + \gamma_t\lambda(g_t + \beta\theta_t)$
    $\theta_{t+1} = \theta_t - v_{t+1}$
end for

3.7. LAMB

LAMB (Layer-wise Adaptive Large Batch Optimization Technique) [16] так же как и предыдущий алгоритм, основывается на послойном адаптивном обучении нейросетевых моделей.

Отличием от оптимизатора LARS является использование второго момента градиентов, как в алгоритме Adam. На основании первого и второго моментов рассчитывается норма каждого отдельного слоя. Это помогает достигнуть лучшей сходимости алгоритма при наименьших временных затратах.

Алгоритм, так же как и LARS, в качестве базового алгоритма использует стохастический градиентный спуск (SGD).

Ниже приведен псевдокод алгоритма LAMB.

Algorithm 7 LAMB

Input: initial parameter θ_0, step sizes $\{\alpha_t\}_{t=1}^{T}$, moment decay $\{\beta_1, \beta_2\}$, wight decay β, LARS coefficient η regularization constant ϵ, stochastic objective function $f(\theta_0)$

Initialize $m_0 = 0$, $v_0 = 0$
for $t = 1$ to T **do**
    $g_t = \nabla f_t(\theta_t)$
    $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $m_t = m_t / (1 - \beta_1^t)$
    $v_t = v_t / (1 - \beta_2^t)$
    $r_t = \frac{m_t}{\sqrt{v_t} + \epsilon}$
    $\theta_{t+1} = \theta_t - \alpha_t \frac{||\theta_t||}{||r_t||} r_t$
end for

3.8. Особенности реализации

В качестве основного инструмента для реализации оптимизационных алгоритмов была использована библиотека TensorFlow. Таким образом, все оптимизаторы должны наследоваться от базового класса `tensorflow.keras.optimizers.Optimizer`. При создании алгоритмов необходимо переопределить несколько методов базового класса. В конструкторе `__init__()` определяются гиперпараметры модели, такие как скорость обучения, коэффициенты затухания и т.д. Инициализация переменных оптимизатора, например, импульса или экспоненциального скользящего среднего градиентов осуществляется в методе `build()`, принимающем на вход список переменных модели `var_list`. Основная логика алгоритма реализуется в методе `update_step()`, который в качестве аргументов принимает переменную и градиент. Результат работы данного метода обновляет веса нейронной сети. Для сохранения конфигурации оптимизатора и последующего запуска реализуется метод `get_config()`, возвращающий словарь со значениями гиперпараметров оптимизационного алгоритма.

В оптимизаторе AdaHessian, помимо переменных и градиентов, которые автоматически вычисляются в TensorFlow, необходимо хранить информацию о аппроксимации матрицы Гессе, которая рассчитыва-

ются при дифференцировании градиентов функции потерь. Поэтому при реализации данного алгоритма были переопределены и другие методы класса Optimizer, включая такие, как `compute_gradients()`, `apply_gradients()`, `minimize()` и т.д.

Основное приложение написано с использованием фреймворка Streamlit. В связи с этим для интеграции оптимизационных алгоритмов были реализованы виджеты с учетом логики Streamlit. Классы виджетов содержат в себе описания, диапазоны значений и строковые форматы гиперпараметров оптимизаторов.

4. Апробация алгоритмов

4.1. Условия эксперимента

Для тестирования работы оптимизационных алгоритмов с помощью библиотек NumPy и Pandas был сгенерирован набор данных, равномерно распределенных на отрезке $[0, 1)$, размерности 10000×50 с добавлением шума, имеющего стандартное нормальное распределение. В качестве тестовой модели была использована нейронная сеть с тремя входными слоями, каждый из которых соединен с отдельным полносвязным слоем. Далее расположен слой конкатенации, связывающий три предыдущих слоя, а за ним еще один полносвязный слой. Он, в свою очередь, разделяется еще на два полносвязных слоя, которые являются выходными слоями модели (рис. 4).

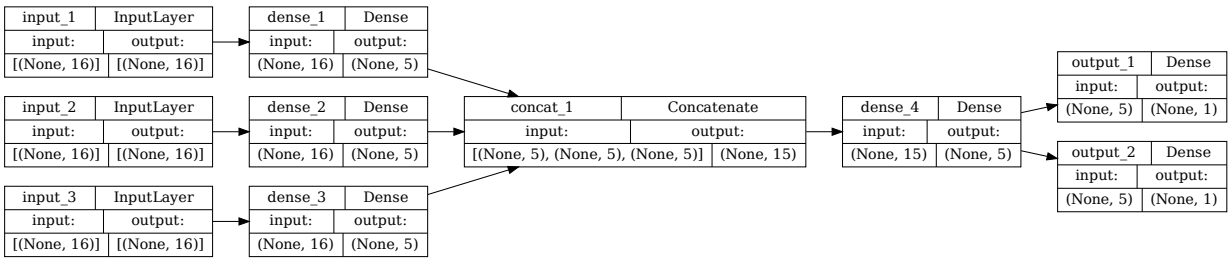

Рис. 4: Модель тестовой нейронной сети

Эксперименты проводилось на рабочей станции MSI Modern 14 B4MW со следующими характеристиками:

- CPU: AMD Ryzen 5 4500U
- GRAPHICS: AMD Radeon Graphics
- MEMORY: 8 GB
- OS: Ubuntu 22.04.1

Помимо обучения с использованием реализованных оптимизационных методов, в ходе эксперимента модель была обучена с оптимизато-

Оптимизатор	Гиперпараметры
AdaMod	$lr = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\beta_3 = 0.995$
RAdam	$lr = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
MADGRAD	$lr = 0.01$, `momentum` $= 0.9$
Apollo	$lr = 0.01$, $\beta = 0.9$, `weight_decay` $= 0.001$
AdaHessian	$lr = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, `weight_decay` $= 0.001$
LARS	$lr = 0.01$, $\beta = 0.9$
LAMB	$lr = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
Adam	$lr = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

Таблица 1: Оптимизаторы и их гиперпараметры

ром Adam, доступным в библиотеке TensorFlow. В таблице 1 представлены гиперпараметры алгоритмов, использованные в процессе эксперимента.

4.2. Используемые метрики

В качестве функции потерь использовалась среднеквадратичная ошибка (Mean Squared Error, MSE) [21]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

где y_i - фактическое значение, $\hat{y}_i$ - предсказанное значение. Она имеет свойство штрафовать большие ошибки сильнее, нежели другие метрики за счет того, что результат ошибки возводится в квадрат

Также для оценки качества обученных моделей была использована метрика — логарифм гиперболического косинуса ошибки предсказания [18]:

$$Logcosh = \frac{1}{n}\sum_{i=1}^{n}\log(ch(y_i - \hat{y}_i))$$

Данная метрика менее чувствительна к выбросам, в отличие от среднеквадратичной и средней абсолютной ошибок, а исходные данные были зашумлены.

4.3. Результаты обучения

На рисунке 5 представлены функции потерь модели с различными оптимизаторами, а на рисунках 6 и 7 логарифмы гиперболического косинуса ошибки.

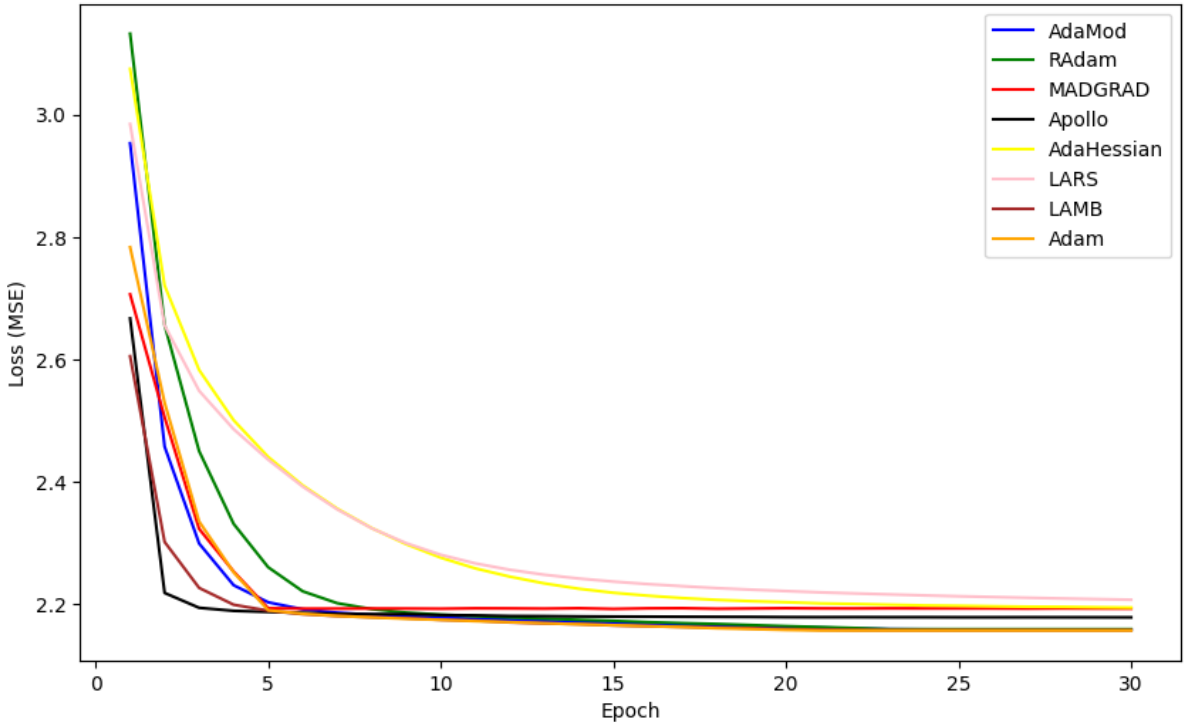

Рис. 5: Функции потерь модели на 30 эпохах

Из графиков видно, что быстрее всего сошлись алгоритмы Apollo, LAMB и AdaMod. А медленнее всего LARS, AdaHessian и RAdam, хотя при этом, у данных оптимизаторов самые гладкие графики, что может говорить о хорошем обучении модели без резких скачков. На графиках логарифма гиперболического косинуса также видно, что Adam сходится быстрее, чем AdaMod, а AdaHessian быстрее, чем LARS.

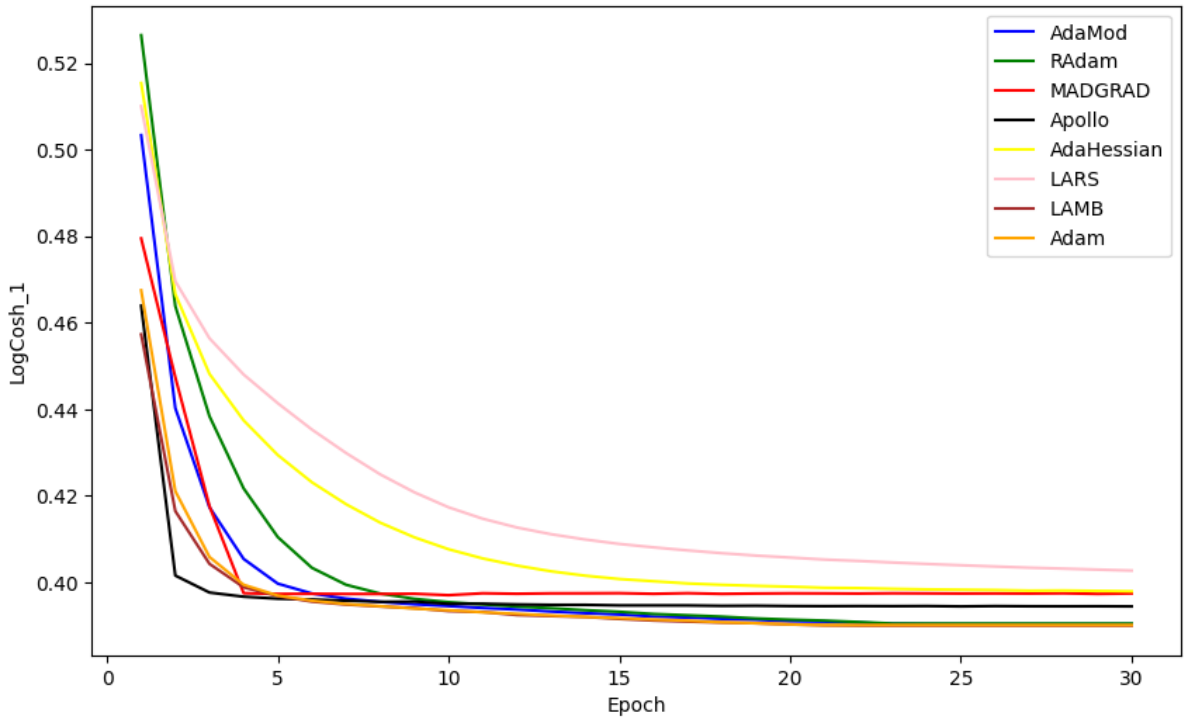

Рис. 6: Логарифм гиперболического косинуса первого выходного слоя

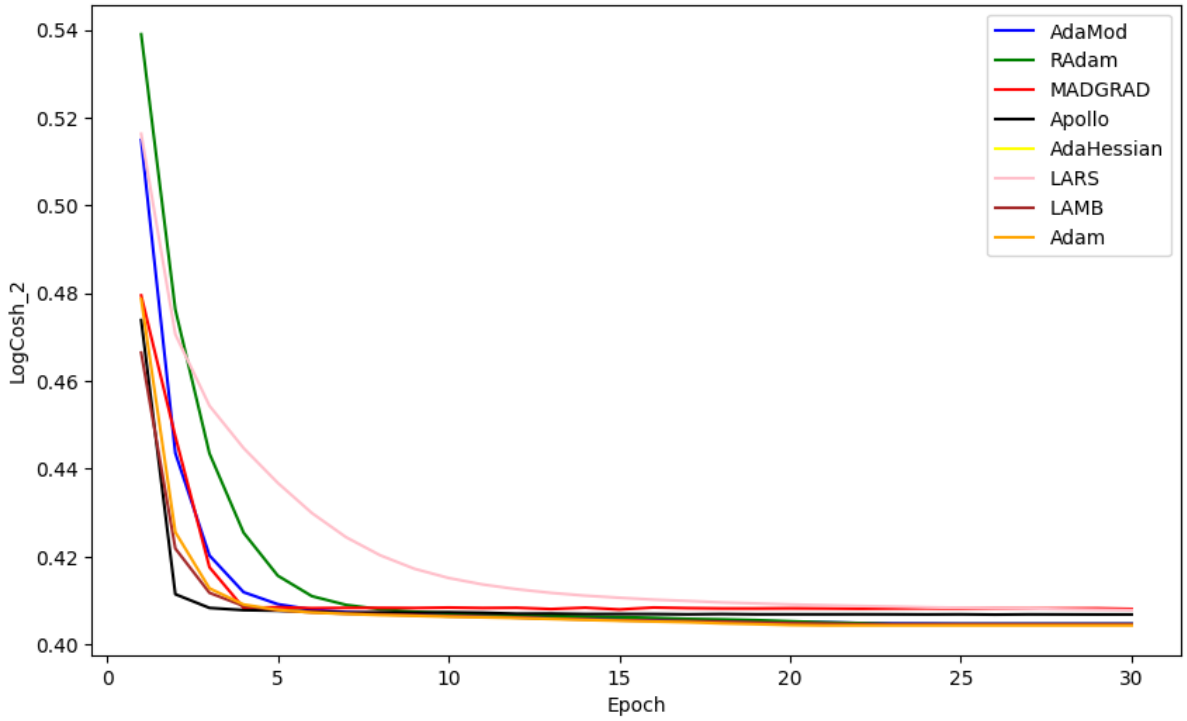

Рис. 7: Логарифм гиперболического косинуса второго выходного слоя

В таблице 2 представлены значения времени обучения модели с различными оптимизаторами.

Алгоритм	Время обучения, с	Число эпох
AdaMod	$15,8 \pm 1,5$	22
MADGRAD	$19,6 \pm 1,3$	24
RAdam	$17,9 \pm 0,4$	23
Apollo	$12,6 \pm 1,4$	17
AdaHessian	$24,1 \pm 0,6$	30
LARS	$21,5 \pm 0,4$	30
LAMB	$15,8 \pm 1,5$	21
Adam	$21,9 \pm 1,1$	21

Таблица 2: Время обучения алгоритмов

Быстрее всех обучилась модель с оптимизатором Apollo — приблизительно 12,5 секунд. Примерно одинаковое время обучения показали алгоритмы первого порядка AdaMod, MADGRAD и LAMB — около 16 секунд. RAdam превзошел по скорости Adam — он оказался одним из самых долгообучаемых среди алгоритмов первого порядка, за исключением оптимизатора LARS со средней скоростью обучения 21,5 секунда. Медленнее всех обучилась модель с оптимизатором AdaHessian — более 24 секунд.

5. Расчет поуровневых коэффициентов скорости колебательных энергообменов

Для решения данной задачи были предоставлены наборы данных VT-обменов между молекулой O_2 и частицами партнерами по столкновению (O_2, O, Ar), рассчитанные с помощью модели нагруженного гармонического осциллятора с учетом свободных вращений. Размерности наборов данных составили 525 элементов для всех пар частиц. Элементы наборов данных состоят из номеров уровней перехода, температур и вычисленных коэффициентов. В работе [33] регрессионной моделью аппроксимировали коэффициенты отдельно для каждого уровня только по значениям температуры, но в этом случае размерность набора данных составляет всего 16 элементов. Данный подход нецелесообразно использовать при обучении нейронной сети. Поэтому нейросетевые модели были обучены по всему набору данных. Предсказание коэффициентов осуществляется по двум переменным — уровню перехода и температуре.

Для обучения была использована шестислойная модель нейронной сети (рис. 8), подобранная эмпирическим путем. В первом и последнем полносвязных слоях использовалась тождественная функция активации для обработки выходного значения. В центральном полносвязном слое была применена сигмоидная функция активации, поскольку все данные больше нуля, а также для добавления нелинейности в модель. Для увеличения производительности и борьбы с переобучением между полносвязными слоями были добавлены слои пакетной нормализации (Batch Normalization Layer).

Поскольку абсолютные метрики могут негативно сказываться на процессе обучения из-за отсутствия возможности дифференцирования во всех точках, особенно для методов второго порядка, в качестве функции потерь была выбрана среднеквадратичная ошибка (Mean Squared

Error, MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

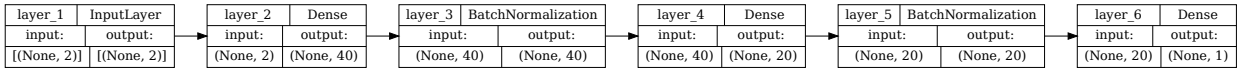

Рис. 8: Архитектура нейронной сети

5.1. Необработанные данные

В таблице 3 приведены значения ошибок и времени обучения модели для всех наборов необработанных данных.

Алгоритм	O_2-O		O_2-Ar		O_2-O_2	
	MSE	Время, с	MSE	Время, с	MSE	Время, с
AdaMod	$1,41\times10^{-3}$	6,90	$1,41\times10^{-3}$	6,46	$2,77\times10^{-3}$	6,64
MADGRAD	$8,64\times10^{-3}$	8,02	$7,70\times10^{-3}$	6,48	$1,01\times10^{-2}$	7,75
RAdam	$1,34\times10^{-3}$	7,57	$9,70\times10^{-4}$	6,85	$1,49\times10^{-3}$	6,76
Apollo	$1,83\times10^{-2}$	3,88	$2,79\times10^{-2}$	3,82	$2,66\times10^{-2}$	3,74
AdaHessian	$5,71\times10^{-4}$	8,79	$2,06\times10^{-3}$	11,73	$2,37\times10^{-3}$	10,93
LARS	$3,05\times10^{-5}$	12,01	$7,12\times10^{-5}$	8,05	$3,57\times10^{-5}$	7,91
LAMB	$1,75\times10^{-4}$	7,19	$1,06\times10^{-4}$	6,69	$1,30\times10^{-5}$	8,04
Adam	$8,63\times10^{-5}$	7,94	$8,02\times10^{-5}$	11,86	$3,87\times10^{-4}$	9,02

Таблица 3: Результаты обучения на «сырых» данных

На рисунках 9, 10, 11 представлены результаты валидации обученных моделей с различными оптимизаторами для O_2-O, O_2-Ar, O_2-O_2 взаимодействий соответственно.

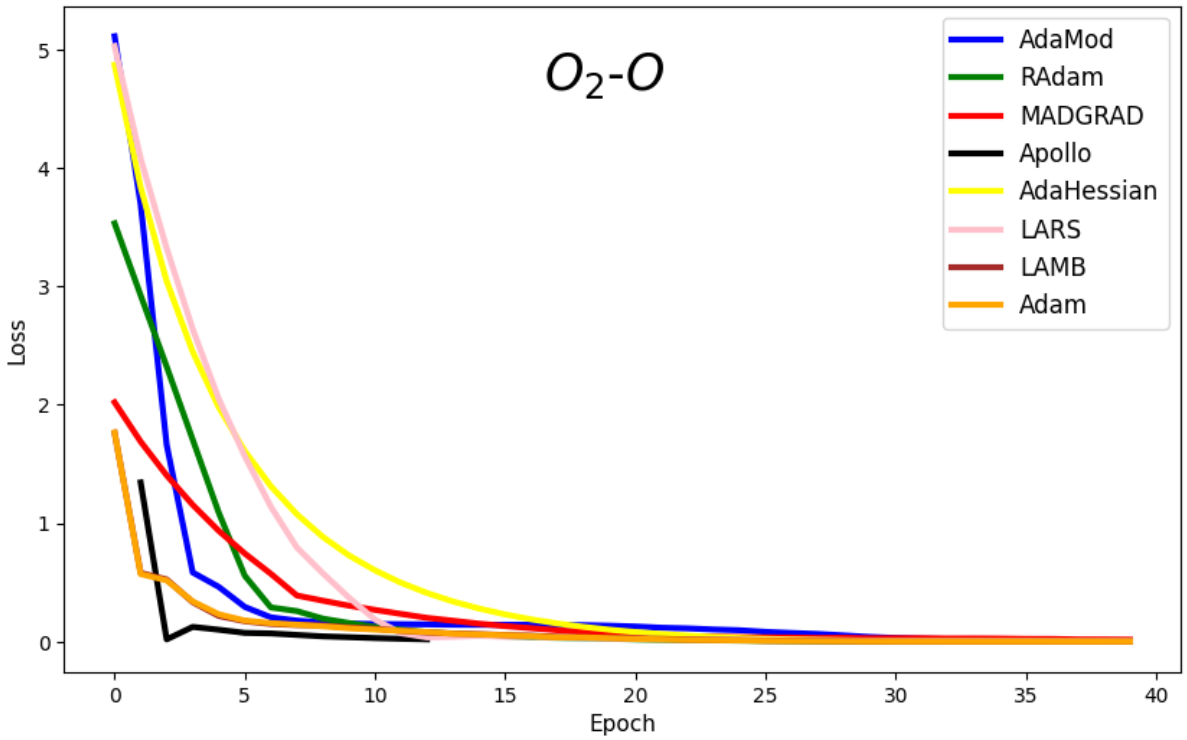

Рис. 9: Результаты валидации моделей, обученных на O_2-O наборе данных

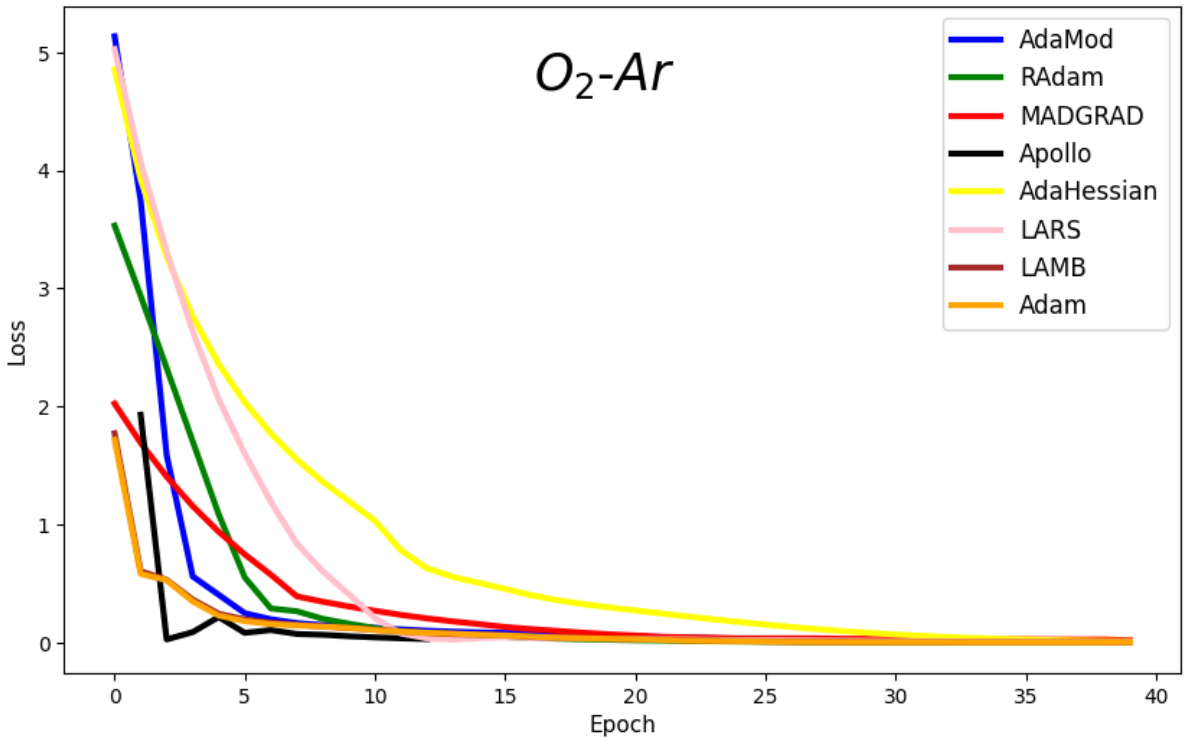

Рис. 10: Результаты валидации моделей, обученных на O_2-Ar наборе данных

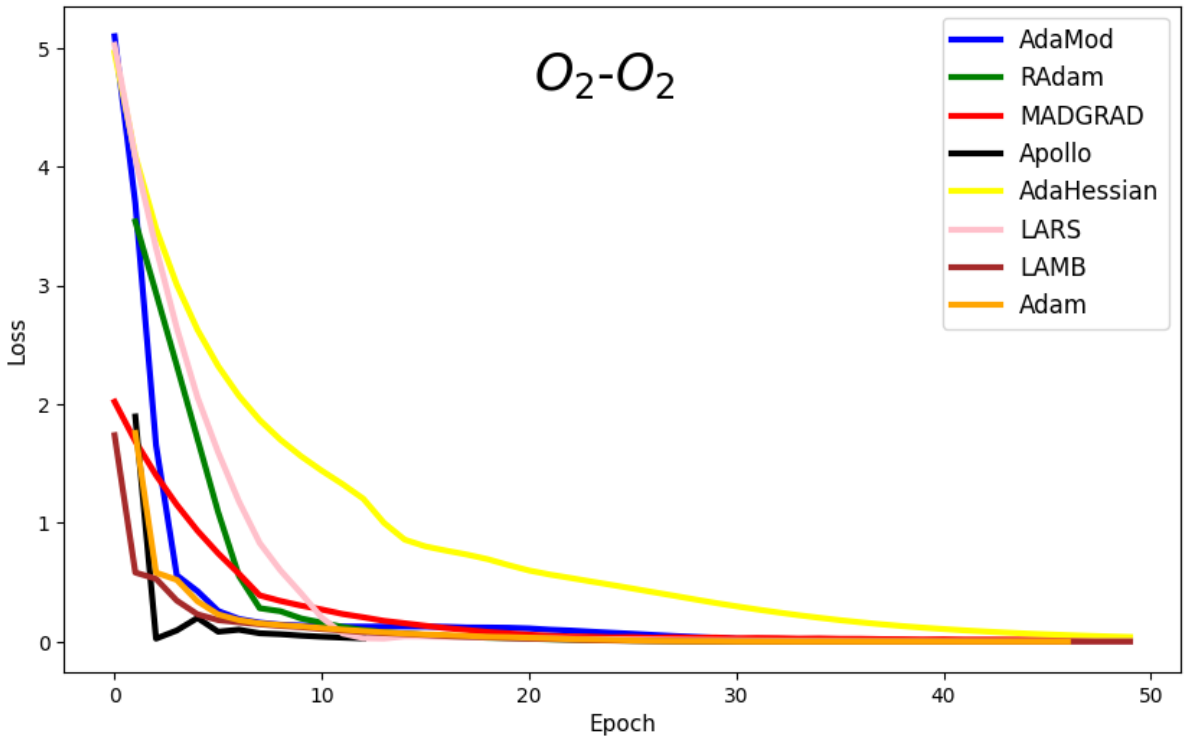

Рис. 11: Результаты валидаций моделей, обученных на O_2-O_2 наборе данных

Практически все алгоритмы показали высокую точность. Для набора данных O_2-O минимальная ошибка была достигнута в алгоритме LARS — $3,05 \times 10^{-5}$, того же порядка ошибка в алгоритме Adam — $8,63 \times 10^{-5}$. Наибольшее значение MSE показал Apollo — $1,83 \times 10^{-2}$. В оптимизаторах LAMB и AdaHessian ошибки оказались равными $1,75 \times 10^{-4}$ и $5,71 \times 10^{-4}$. Остальные методы достигли значений MSE третьего порядка малости при валидации.

Для O_2-Ar набора данных LARS и Adam так же показали лучшие результаты обучения — $7,12 \times 10^{-5}$ и $8,02 \times 10^{-5}$. Ошибка Apollo оказалась наибольшей — $2,79 \times 10^{-2}$, а результаты алгоритмов LAMB и RAdam получились равны $1,06 \times 10^{-4}$ и $9,70 \times 10^{-4}$. Другие алгоритмы сошлись к ошибке третьего порядка малости.

При решении задачи для O_2-O_2 набора данных лучшее значение ошибки при валидации модели показал оптимизатор LAMB — $1,30 \times 10^{-5}$. LARS так же достиг MSE пятого порядка — $3,57 \times 10^{-5}$, а Adam четвертого порядка — $3,87 \times 10^{-4}$. Результат Apollo снова оказался наи-

худшим — $2,66 \times 10^{-2}$, ошибку того же порядка удалось получить при обучении с оптимизатором MADGRAD — $1,01 \times 10^{-2}$. Результаты MSE остальных алгоритмов при валидации имеют третий порядок малости.

Несмотря на то что Apollo показал наименьшую точность, модель с этим алгоритмом для всех наборов данных сошлась быстрее остальных методов — около 3.8 секунд. AdaHessian для O_2-Ar и O_2-O_2 датасетов оказался самым медленным — время сходимости превысило 10 секунд. А для O_2-O набора данных дольше всех обучался LARS — 12 секунд.

Наиболее удачными алгоритмами с точки зрения скорости и качества оказались оптимизаторы LARS, LAMB и Adam.

5.2. Обработанные данные

Поскольку значения коэффициентов малы, для улучшения качества предсказаний они были предварительно отмасштабированы — умножены на 10×10^{18} и преобразованы до диапазона [0, 1] с помощью MinMaxScaler:

$$\hat{x_i} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

где x_i - значение i-го элемента признака, x_{min} и x_{max} - минимальное и максимальное значения признака.

В таблице 4 представлены значения ошибок и времени обучения для всех наборов данных после масштабирования.

Алгоритм	O_2-O		O_2-Ar		O_2-O_2	
	MSE	Время, с	MSE	Время, с	MSE	Время, с
AdaMod	0,004	6,23	0,117	4,31	0,015	4,44
MADGRAD	0,223	4,76	0,256	4,68	0,168	4,57
RAdam	0,072	6,65	0,069	5,04	0,026	4,94
Apollo	0,261	2,48	0,300	2,85	0,174	2,46
AdaHessian	0,129	5,92	0,087	5,86	0,114	5,85
LARS	0,007	4,43	0,025	4,38	0,029	4,43
LAMB	0,027	4,47	0,063	4,78	0,036	4,67
Adam	0,100	4,23	0,338	3,11	0,352	3,42

Таблица 4: Результаты обучения на преобразованных данных

На рисунках 12, 13, 14 представлены результаты валидации моделей на преобразованных данных.

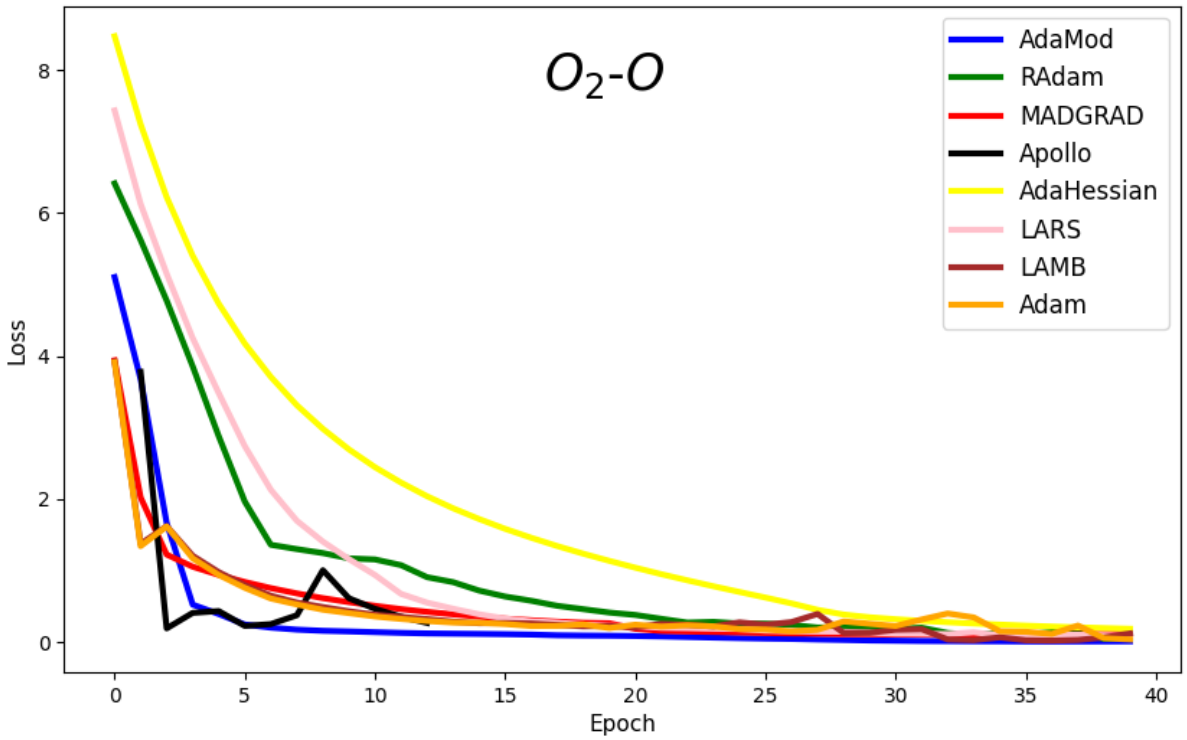

Рис. 12: Результаты валидации моделей, обученных на O_2-O наборе данных после масштабирования

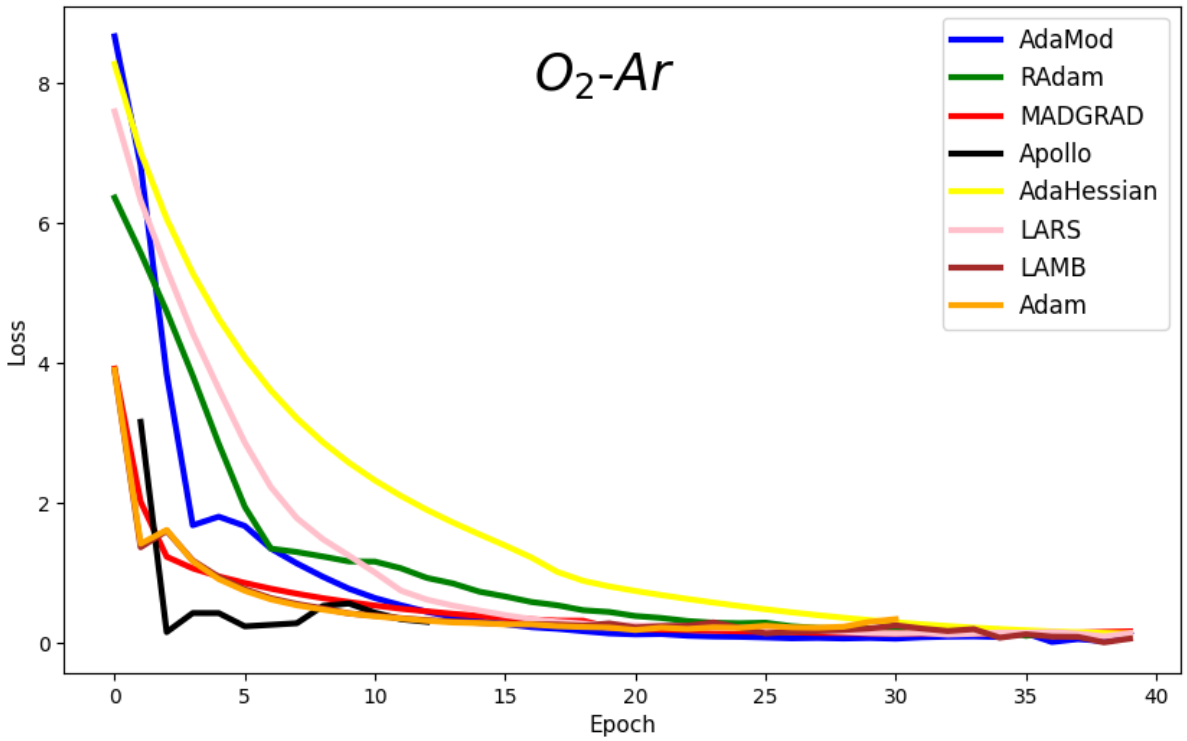

Рис. 13: Результаты валидации моделей, обученных на O_2-Ar наборе данных после масштабирования

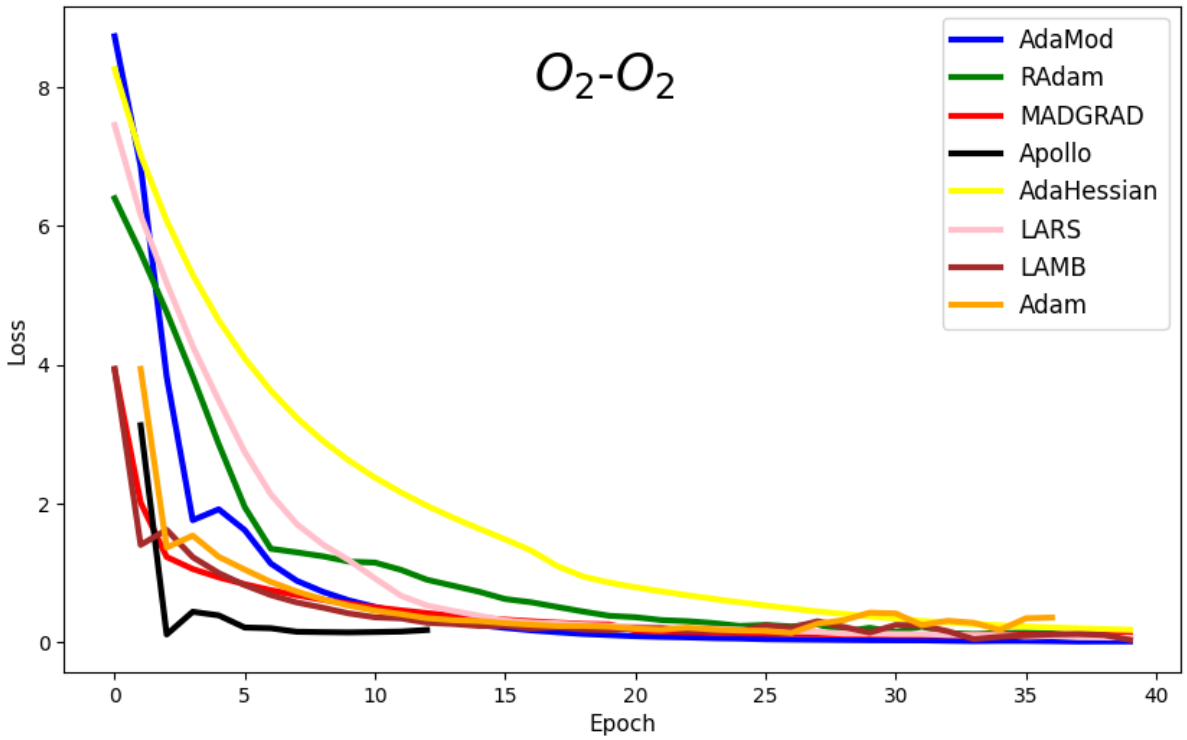

Рис. 14: Результаты валидаций моделей, обученных на O_2-O_2 наборе данных после масштабирования

Лучшие результаты валидации для набора данных O_2-O взаимодействий показали алгоритмы AdaMod и LARS — 0,004 и 0,007. На порядок больше значение ошибок у оптимизаторов LAMB и RAdam — 0,027 и 0,072. Самая большие ошибки были получены при использовании методов Apollo и MADGRAD — 0,261 и 0,223. Значения MSE для AdaHessin и Adam составили 0,129 и 0,1.

Для O_2-Ar набора данных лучшее значение ошибки показал алгоритм LARS – 0,025. Ошибок того же порядка удалось достичь при использовании оптимизаторов LAMB, RAdam и AdaHessian — 0,063, 0,069 и 0,087. Худшего результата достиг Adam со значением ошибки 0,338. Значения MSE моделей с алгоритмами Apollo, MADGRAD и AdaMod оказались равными 0,3, 0,256 и 0,117.

При решении задачи O_2-O_2 взаимодействий самой точной оказалась модель с оптимизатором AdaMod — 0,015. Близкие результаты показали алгоритмы RAdam, Lars и LAMB — 0,026, 0,029 и 0,036. Самая большая ошибка была получена при использовании оптимизатора Adam —

0,352. Значения MSE алгоритмов AdaHessian, MADGRAD и Apollo оказались достаточно равными 0,114, 0,168 и 0,174.

Самым быстрым снова оказался Apollo — менее 3 секунд. Для O_2-O датасета наибольшее время показали AdaMod и RAdam – 6,23 и 6,65 секунд. В случае O_2-Ar и O_2-O_2 взаимодействий дольше всех обучалась модель с оптимизатором AdaHessian — 5,86 и 5,85 секунд.

На рисунках 15, 16, 17 представлены предсказания модели с оптимизатором LARS для различных уровней переходов трех наборов данных.

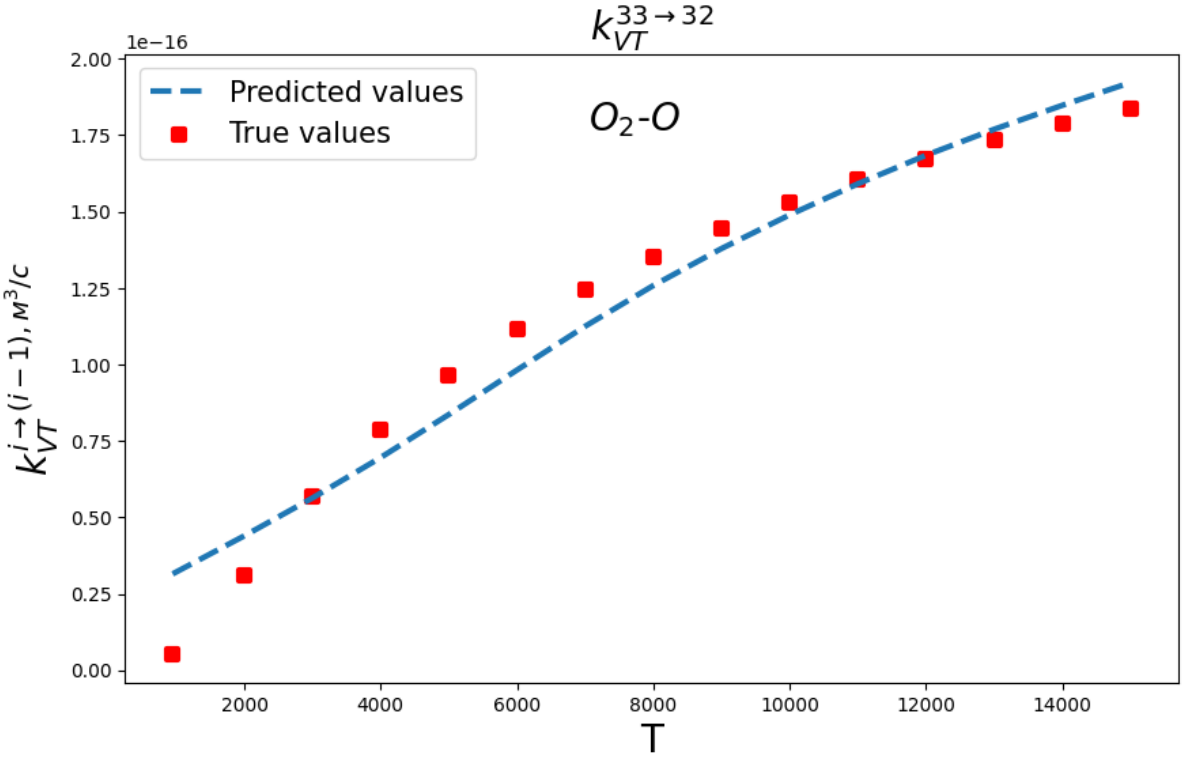

Рис. 15: $k_{VT}^{33\to32}$ O_2-O взаимодействий

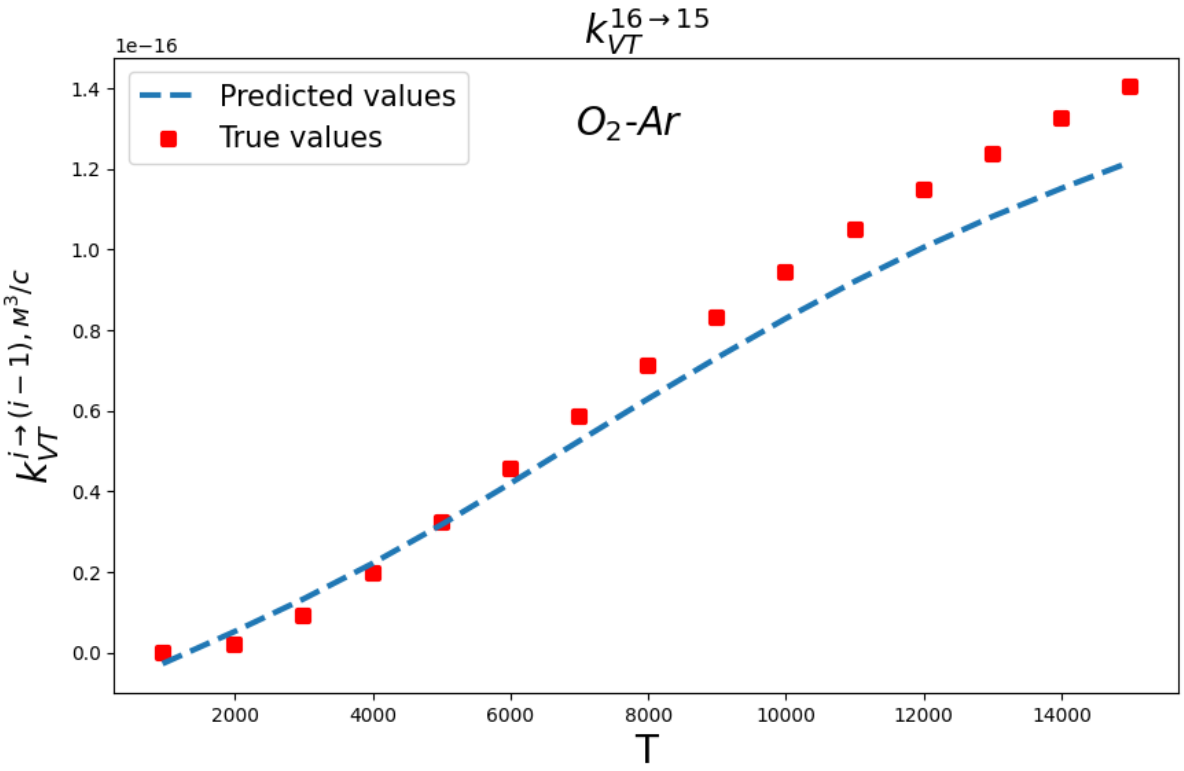

Рис. 16: $k_{VT}^{16\to15}$ O_2-Ar взаимодействий

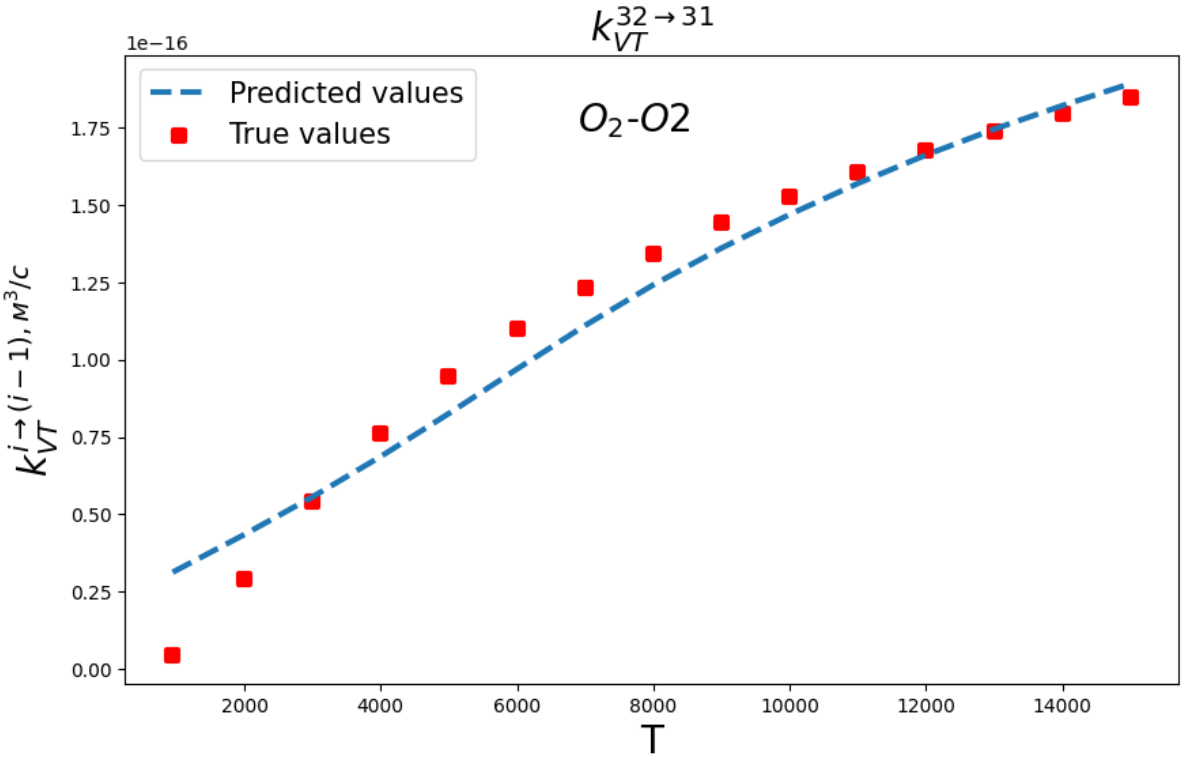

Рис. 17: $k_{VT}^{32\to31}$ O_2-O взаимодействий

Наиболее вероятной причиной ошибок в предсказаниях является тот факт, что значения коэффициентов для разных уровней отличаются на несколько порядков: 10^{-22} для первого и 10^{-16} для тридцать пятого. Даже с учетом масштабирования разница остается существенной. При этом из рисунков видно, что модель ошибается незначительно, предсказанные значения близки к истинным.

Таким образом, масштабирование данных позволили сократить среднее время обучения более чем в полтора раза для всех наборов данных и улучшить качество предсказаний.

Заключение

В ходе работы были получены следующие результаты:

1. На языке Python с использованием библиотеки TensorFlow были реализованы и интегрированы в приложение machine-learning-ui оптимизационные алгоритмы: AdaMod, Radam, MADGRAD, Apollo, AdaHessian, LARS, LAMB.

2. Была проведена апробация алгоритмов. Модель нейросети со всеми оптимизаторами сошлась в течение 30 эпох.

3. Была решена задача расчета поуровневых коэффициентов скорости колебательных энергообменов для пар частиц (O_2-O, O_2-Ar, O_2-O_2) с применением нейросетевого подхода. Была подобрана оптимальная топология модели для текущей задачи. Обучение модели с различными оптимизационными алгоритмами заняло менее 6 секунд для наборов с учетом масштабирования. Значение ошибки при валидации оптимизаторов составило от 0,352 до 0,004. Предсказанные значения оказались близки к истинным.

С кодом реализованных оптимизационных алгоритмов можно ознакомиться по ссылке в репозитории проекта: `https://github.com/daffeu/machine-learning ui/tree/optimizers/src/mlui/CustomOptimizers`.

Список литературы

[1] Yao Zhewei, Gholami Amir, Shen Sheng et al. ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning. — 2021. — 2006.00719.

[2] Adamovich Igor V. Three-Dimensional Analytic Model of Vibrational Energy Transfer in Molecule-Molecule Collisions // AIAA Journal. — 2001. — Vol. 39, no. 10. — P. 1916–1925. — https://doi.org/10.2514/2.1181.

[3] Adamovich Igor V., Rich J. William. Three-dimensional nonperturbative analytic model of vibrational energy transfer in atom–molecule collisions // The Journal of Chemical Physics. — 1998. — 11. — Vol. 109, no. 18. — P. 7711–7724. — https://pubs.aip.org/aip/jcp/article-pdf/109/18/7711/19114845/7711_1_online.pdf.

[4] Ding Jianbang, Ren Xuancheng, Luo Ruixuan, Sun Xu. An Adaptive and Momental Bound Method for Stochastic Learning. — 2019. — 1910.12249.

[5] CSS. — URL: https://www.w3.org/Style/CSS/Overview.en.html.

[6] Defazio Aaron, Jelassi Samy. Adaptivity without Compromise: A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization. — 2021. — 2101.11075.

[7] Duchi John C., Hazan Elad, Singer Yoram. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization // J. Mach. Learn. Res. — 2011. — Vol. 12. — P. 2121–2159. — URL: https://api.semanticscholar.org/CorpusID:538820.

[8] Facebook. PyTorch. — URL: https://pytorch.org/.

[9] Fletcher, Roger. Practical Methods of Optimization (2nd ed.). — New York, USA : John Wiley & Sons, 1987. — ISBN: 978-0-471-91547-8.

[10] Google Brain. TensorFlow. — URL: https://www.tensorflow.org.

[11] Google Brain. TensorFlow Addons. — URL: https://www.tensorflow.org/addons/overview.

[12] Gradient Descent Only Converges to Minimizers / Jason D. Lee, Max Simchowitz, Michael I. Jordan, Benjamin Recht // 29th Annual Conference on Learning Theory / Ed. by Vitaly Feldman, Alexander Rakhlin, Ohad Shamir. — Vol. 49 of Proceedings of Machine Learning Research. — Columbia University, New York, New York, USA : PMLR, 2016. — 23–26 Jun. — P. 1246–1257. — URL: https://proceedings.mlr.press/v49/lee16.html.

[13] HTML. — URL: https://www.w3.org/html/.

[14] Hinton. Root Mean Square Propagation. — URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[15] Kingma Diederik P., Ba Jimmy. Adam: A Method for Stochastic Optimization. — 2017. — 1412.6980.

[16] You Yang, Li Jing, Reddi Sashank et al. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. — 2020. — 1904.00962.

[17] Li Jichao, Du Xiaosong, Martins Joaquim R.R.A. Machine learning in aerodynamic shape optimization // Progress in Aerospace Sciences. — 2022. — Vol. 134. — P. 100849. — URL: https://www.sciencedirect.com/science/article/pii/S0376042122000410.

[18] Log-Cosh loss. — URL: https://www.tensorflow.org/api_docs/python/tf/keras/losses/logcosh.

[19] Ma Xuezhe. Apollo: An Adaptive Parameter-wise Diagonal Quasi-Newton Method for Nonconvex Stochastic Optimization. — 2021. — 2009.13586.

[20] Matt Taddy. Business Data Science: Combining Machine Learning and Economics to Optimize, Automate, and Accelerate Business Decisions. — New York, USA : McGraw Hill, 2019. — ISBN: 978-1-260-45277-8.

[21] Mean Squared Error (MSE). — URL: www.probabilitycourse.com.

[22] Nocedal Jorge, Wright Stephen J. Numerical Optimization. — New York, USA : Springer, 2006. — ISBN: 978-0387-30303-1.

[23] Novik Nikolay. pytorch-optimizer. — URL: https://github.com/jettify/pytorch-optimizer.

[24] Liu Liyuan, Jiang Haoming, He Pengcheng et al. On the Variance of the Adaptive Learning Rate and Beyond. — 2021. — 1908.03265.

[25] PapersWithCode. — URL: https://paperswithcode.com/trends.

[26] Rumelhart David E., Hinton Geoffrey E., Williams Ronald J. Learning representations by back-propagating errors // Nature. — 1986. — Vol. 323. — Volume 323, Issue 6088.

[27] Schwartz R. N., Slawsky Z. I., Herzfeld K. F. Calculation of Vibrational Relaxation Times in Gases // The Journal of Chemical Physics. — 1952. — 10. — Vol. 20, no. 10. — P. 1591–1599. — https://pubs.aip.org/aip/jcp/article-pdf/20/10/1591/18801568/1591_1_online.pdf.

[28] Snowflake Inc. Streamlit. — URL: https://streamlit.io/.

[29] Vibrational Energy Transfer Rates Using a Forced Harmonic Oscillator Model / Igor V. Adamovich, Sergey O. Macheret, J. William Rich, Charles E. Treanor // Journal of Thermophysics and Heat Transfer. — 1998. — Vol. 12, no. 1. — P. 57–65. — https://doi.org/10.2514/2.6302.

[30] You Yang, Gitman Igor, Ginsburg Boris. Large Batch Training of Convolutional Networks. — 2017. — 1708.03888.

[31] Виноградов О. Л. Математический анализ: учебник. — Санкт-Петербург, Россия : БХВ-Петербург, 2017. — ISBN: 978-5-9775-3815-2.

[32] Даниил Тяпкин. Учебник по машинному обучению. 14.1. Оптимизация в ML. — URL: https://education.yandex.ru/handbook/ml/article/optimizaciya-v-ml.

[33] Исаков Андрей Алексеевич, Гориховский Вячеслав Игоревич, Мельник Максим Юрьевич. Модели регрессии для расчёта поуровневых коэффициентов скорости колебательных энергообменов // ВЕСТНИК САНКТ-ПЕТЕРБУРГСКОГО УНИВЕРСИТЕТА. МАТЕМАТИКА. МЕХАНИКА. АСТРОНОМИЯ. — 2024.

[34] Истомин Владимир Андреевич, Павлов Семён Анатольевич. Suitability of different machine learning methods for high-speed flow modeling issues // Cybernetics and Physics. — 2023. — . — Vol. 12, no. 4. — P. 264–274.