ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(СПБГУ)

Образовательная программа магистратуры
«Прикладные математика и физика»

Отчёт по учебной практике

Применение искусственных нейронных сетей к решению спектральных задач квантовой механики

Выполнил студент
1 курса магистратуры
(группа 22.М21-фз)
Козлов Александр

Санкт-Петербург
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Технологии машинного обучения все глубже проникают в жизнь человека, находя свое применение в широком спектре областей человеческой деятельности — от детектирования редких смертельных заболеваний до прогнозирования индексов климатических мод Земли. Неотъемлемой составляющей каждой технологии машинного обучения является используемая модель машинного обучения — некий «черный ящик», производящий трансформацию входных данных. В последние десятилетия наиболее широкое распространение получила такая модель машинного обучения, как искусственная нейронная сеть (НС). Главным преимуществом данной модели машинного обучения является большое количество свободных параметров, правильно подобрав которые, можно со сколь угодно высокой точностью (которая зависит от количества свободных параметров) аппроксимировать любую функциональную зависимость [1]. Стоит заметить, что из-за большого количества параметров, процесс их подбора долгое время оставался крайне тяжелой вычислительной задачей, успешно начать решать которую позволило развитие технологий многомерной оптимизации.

Ввиду того, что НС успешно решают многомерные задачи (обработка натурального языка, генерация изображений и т.д.), естественным представляется использовать НС при решении многомерных задач математической физики. В таком приложении зачастую используются полносвязные НС прямого распространения, которые получили название physics-informed neural networks (нейронные сети, знающие физику, НСЗФ) [2], хотя их отличие от остальных типов НС заключается лишь в форме минимизируемого функционала (в него добавлены имеющие физическую интерпретацию члены). Форма минимизируемого функционала (функционала потерь или целевого функционала) зависит от специфики решаемой физической задачи, так, если ставится задача получить приближенное решение некоторого уравнения и НС используется в качестве пробной функции, то в функционал потерь разумно включить норму невязки НС по уравнению, граничное условие, если таковое имеется, а также все прочие соотношения, которым должно удовлетворять решение.

В 90-ые годы прошлого века было предложено использовать НС для решения задач на собственные значения и собственные функции — в работе [3] предлагается решать стационарное уравнение Шредингера с помощью НС с одним скрытым слоем, включая в функционал потерь квадратичную норму невязки пробной функции по уравнению. Такой подход позволил находить основное состояние, а также несколько возбужденных состояний стационарных квантово-механических систем. Подход, применяемый в [3], подразумевает, что каждому состоянию отвечает отдельная НС, а состояния находятся последовательно, начиная с основного (при отыскании возбужденных состояний в функционал потерь включаются члены, отвечающие за ортогональность искомого состояния к уже найденным состояниям).

В работе [4] предложено использовать одну НС для отыскания сразу нескольких состояний, включая в функционал потерь члены, отвечающие за взаимную ортогональность рассматриваемых состояний, такой подход мог бы позволить значительно ускорить

решение спектральных задач высокой размерности. Кроме того, в данной работе уделяется внимание необходимости правильно задавать распределение точек в координатном пространстве, что крайне важно при решении задач в пространствах высокой размерности, и предлагается использовать в качестве плотности распределения точке среднюю по отыскиваемым состояниям плотность вероятности. Однако, работа [4] полна спорных моментов, главным из которых является используемый функционал потерь — в работе [5] приводится строгое доказательство того, что используемый в [4] функционал потерь имеет минимумы на функциях, не являющихся решениями задачи. Кроме того, в работе [5] произведена модернизация функционала потерь (в него добавлен член, отвечающий за невязку пробной функции по уравнению), чтобы тот достигал минимумы лишь на решении задачи, и с использованием модернизированного функционала потерь успешно решены спектральные квантово-механические задачи в размерностях 1 и 2. Целью настоящей работы является распространение успехов работы [5] на большие размерности.

1 ТЕСТОВАЯ СПЕКТРАЛЬНАЯ ЗАДАЧА

1.1 ФОРМУЛИРОВКА

В качестве тестового примера, на котором в дальнейшем будет отрабатываться рассматриваемая методика решения спектральных задач квантовой механики, будет выступать задача о квантово-механическом гармоническом осцилляторе. Такая задача легко формулируется в любых размерностях, аналитически решается и обладает хорошо разнесенным спектром, почему и была выбрана. Математически такая задача ставится следующим образом:

$$\hat{H}\psi(\mathbf{r}) = E\psi(\mathbf{r}),\ \hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + \frac{k}{2}\sum_{d=1}^{D}|r_d|^2, \tag{1.1}$$

где $\hat{H}$ — гамильтониан задачи, $\psi(\mathbf{r})$ — волновая функция задачи (волновая функция, которая удовлетворяет уравнению, называется собственной функцией), E — значение энергии (значение энергии, которое отвечает собственной функции, называется собственным значением), D — размерность координатного пространства. Требуется найти собственные функции и собственные значения. Далее будет рассматриваться безразмерное уравнение, полученное заменой $\hbar^2/m = 1$ и $k = 1$.

1.2 ИЗВЕСТНОЕ РЕШЕНИЕ

Задача о квантово-механическом осцилляторе широко известна и её аналитическое решение приведено во многих классических курсах квантовой механики, например, в [6]. Точный спектр такой задачи задается соотношением

$$E_{n_1,\ldots,n_D} = \frac{D}{2} + \sum_{d=1}^{D} n_d, \tag{1.2}$$

где $n_d = 0, 1, 2, \ldots$ для $d = \overline{1,\, D}$. Собственные функции задачи в одномерном случае имеют вид:

$$\psi_m(x) = \frac{1}{\sqrt{\sqrt{\pi}\,2^m\,m!}}\,H_m(x)\,\exp\bigl(-x^2/2\bigr), \tag{1.3}$$

где m — номер состояния, а $H_m(x)$ — полином Эрмита [7]. В пространствах большей размерности собственные функции собираются из собственных функций одномерной задачи с точностью до нормировочного фактора следующим образом:

$$\psi_{n_1,\ldots,n_D}(\mathbf{r}) = \prod_{d=1}^{D}\psi_{n_d}(r_d), \tag{1.4}$$

где $n_d = 0, 1, 2, \ldots$ для $d = \overline{1,\, D}$, а $\psi_{n_d}(r_d)$ — собственная функция одномерной задачи.

2 АРХИТЕКТУРА ПРОБНОЙ ФУНКЦИИ

Пробная функция (ПФ) $\mathcal{F}$ определяется как отображение из пространства $\mathbb{R}^{B\times D}$ в пространство $\mathbb{R}^{B\times M}$, где B — размер партии точек в координатном пространстве, D — размерность координатного пространства и M — число состояний, которые мы хотим найти. Таким образом, ПФ $\mathcal{F}$ преобразует B точек в D-мерном пространстве в B значений M волновых функций в этих точках.

ПФ конструируется как произведение нейронной сети (НС) $\mathcal{N}$ и амплитудной функции (АФ) $\mathcal{A}$. Использование НС обусловлено тем, что она является универсальным аппроксиматором, что позволит ПФ аппроксимировать любую функциональную зависимость, в том числе и решение спектральной задачи. АФ же используется из тех соображений, что ПФ должна удовлетворять граничным условиям и требуемым асимптотикам. Рассмотрим каждую из этих двух сущностей в отдельности.

2.1 АМПЛИТУДНАЯ ФУНКЦИЯ

АФ должна удовлетворять граничным условиям и известным асимптотикам решения. В дальнейшем будет рассматриваться тестовая задача (1.1), её основное состояние достаточно просто находится и волновая функция, отвечающая ему, широко известна — это гауссиан $\mathcal{G}$, который с точностью до нормировочного множителя имеет вид:

$$\mathcal{G}(\mathbf{r}) = \exp\left(-\frac{1}{2}\sum_{d=1}^{D} r_d^2\right). \tag{2.1}$$

Гауссиан $\mathcal{G}$ обеспечивает удовлетворение граничных условий на бесконечности и требуемых асимптотик (что лекго видеть из аналитической формы решений (1.3)), поэтому он и был выбран в качестве АФ $\mathcal{F} = \mathcal{G}$.

2.2 НЕЙРОННАЯ СЕТЬ

В качестве НС использовалась полносвязная НС прямого распространения с 3-мя внутренними слоями, число нейронов на которых обозначается через N_h и может изменяться пользователем. Входной слой пребразует входные данные X из пространства $\mathbb{R}^{B\times D}$ в пространство $\mathbb{R}^{B\times N_h}$ следующим образом (используется подход, применяемый в фреймворке PyTorch; см. `https://pytorch.org/`):

$$Y^{(1)} = f(XW_{(1)}^{\mathrm{T}} + B_{(1)}), \tag{2.2}$$

где f — функция активации (во всей НС используется одна и та же функция активации), $W_{(1)} \in \mathbb{R}^{N_h\times D}$ — матрица весов входного слоя, $B_{(1)} \in \mathbb{R}^{N_h}$ — вектор сдвигов входного слоя. Первый и второй внутренние слои преобразуют данные аналогичным образом:

$$Y^{(i+1)} = f(Y^{(i)}W_{(i+1)}^{\mathrm{T}} + B_{(i+1)}),\ i = 1, 2 \tag{2.3}$$

где $W_{(i+1)} \in \mathbb{R}^{N_h \times N_h}$ — матрица весов i-ого внутреннего слоя, $B_{(i+1)} \in \mathbb{R}^{N_h}$ — вектор сдвигов i-ого внутреннего слоя. Последний внутренний слой преобразует данные к выходу следующим образом:

$$Y = Y^{(4)} = f(Y^{(3)} W_{(4)}^{\mathrm{T}} + B_{(4)}), \tag{2.4}$$

где $W_{(4)} \in \mathbb{R}^{M \times N_h}$ — матрица весов последнего внутреннего слоя, $B_{(4)} \in \mathbb{R}^{M}$ — вектор сдвигов последнего внутреннего слоя.

Все матрицы весов W и вектора сдвигов B являются обучаемыми параметрами НС и подбираются в ходе ее обучения. Общее число свободных параметров у НС с такой архитектурой составляет

$$(D \times N_h + N_h) + 2 \cdot (N_h \times N_h + N_h) + (M \times N_h + M).$$

Таким образом, число обучаемых параметров линейно по размерности координатного пространства D, линейно по числу отыскиваемых состояний M и квадратично по числу нейронов в каждом из внутренних слоев N_h.

3 ФУНКЦИОНАЛ ПОТЕРЬ

В данном разделе рассматривается состав функционала потерь и объясняется смысл каждой из составляющих функционал потерь частей. Функционал потерь строится, как и в работе [5], из четырех частей:

1. Члена $R = \sum_{m=1}^{M} \left\| \hat{H}\psi_m - E_m\psi_m \right\|^2 / \|\psi_m\|^2$, отвечающего за невязку по уравнению;
2. Члена $A = \sum_{m=1}^{M} (\|\psi_m\|^2 - 1)^2$, отвечающего за нормировку пробных функций;
3. Члена $B = \sum_{m_1=2}^{M} \sum_{m_2=1}^{m_1} \langle \psi_{m_1} | \psi_{m_2} \rangle / \|\psi_{m_1}\|^2 / \|\psi_{m_2}\|^2$, отвечающего за ортогонализацию пробных функций;
4. Члена $E = \sum_{m=1}^{M} E_m$, отвечающего за минимизацию рассматриваемых собственных значений.

Эти части суммируются с соответствующими им весовыми коэффициентами и формируют итоговый функционал потерь:

$$\mathcal{L} = w_R R + w_A A + w_B B + w_E E. \tag{3.1}$$

3.1 ВЫЧИСЛЕНИЕ МАТРИЧНЫХ ЭЛЕМЕНТОВ И СКАЛЯРНЫХ ПРОИЗВЕДЕНИЙ

Для расчета функционала потерь (3.1) необходимо уметь вычислять матричные элементы и скалярное произведение. Если множество точек мощностью B в координатном пространстве, на котором оценивается значение функционала потерь, подчиняется некому распределению с функцией плотности вероятности W, то согласно методу выборки по важности матричный элемент некоторого оператора $\hat{O}$ можно вычислить следующим образом:

$$\langle \psi_{m_1} | \hat{O} | \psi_{m_2} \rangle = \frac{1}{B} \sum_{b=1}^{B} \frac{\psi_{m_1}(\mathbf{r}_b) \cdot (\hat{O}\psi_{m_2})|_{\mathbf{r}_b}}{W(\mathbf{r}_b)}, \tag{3.2}$$

где $\{\mathbf{r}_b\}_{b=1}^{B}$ — точки в координатном пространстве, подчиняющиеся распределению с плотностью вероятности W. Аналогично вычисляются скалярные произведения и квадрат нормы функций в координатном пространстве.

4 ОБУЧЕНИЕ ПРОБНОЙ ФУНКЦИИ

В данном разделе будет приведена информация о процессе обучения пробной функции, будут рассмотрены такие темы, как семплирование точек в координатном пространстве и используемый метод многомерной оптимизации, кроме того, будет приведен краткий алгоритм обучения пробной функции.

4.1 СЕМПЛИРОВАНИЕ ТОЧЕК В КООРДИНАТНОМ ПРОСТРАНСТВЕ

В настоящей работе, в отличии от работы [5], используются наборы точек в координатном пространстве, распределенные неравномерно. В качестве плотности вероятности распределения точек в координатном пространстве был взят усредненный по состояниям квадрат волновой функции, то есть функция

$$W(\mathbf{r}) = \frac{1}{M} \sum_{m=1}^{M} |\psi_m(\mathbf{r})|^2. \tag{4.1}$$

Данная идея предлагается в работе [4].

Каждую итерацию обучения в начале итерации происходит трансформация набора точек в координатном пространстве таким образом, чтобы набор точек подчинялся распределению с плотностью вероятности, которая пропорциональна актуальной функции W. Для этого используется алгоритм Метрополиса [8], при этом набор точек в координатном пространстве на первой итерации обучения генерируется таким образом, что точки равномерно распределены в некоторой области. В Листинге 1 представлена реализация одного шага алгоритма Метрополиса на языке Python с использованием фреймворка PyTorch.

Листинг 1: Функция, осуществляющая один шаг алгоритма Метрополиса.

```
def MetropolisAlghorithmStep(x, f):
    """One step of the Metropolis algorithm, it return the updated sample x"""
    global parameters
    x_prime = x + parameters["epsilon"] * (2 * torch.rand_like(x).to(device) - 1)
    alpha = f(x_prime).to(device) / f(x).to(device)
    doesXMove = torch.rand((x.shape[0])).to(device) <= alpha
    x_prime = (
        torch.mul(doesXMove.int(), x_prime.t()).t()
        + torch.mul((1 - doesXMove.int()), x.t()).t()
    )
    return x_prime
```

4.2 МЕТОД МНОГОМЕРНОЙ ОПТИМИЗАЦИИ

В качестве метода многомерной оптимизации был избран метод AdamW [9], который выгодно отличается от метода Adam [10], использовавшегося в работе [5], тем, что подходит к проблеме «вшитой в алгоритм L2-регуляризации» (в обоих алгоритмах есть возможность включить «вшитую L2-регуляризацию», но производятся они по–разному, в этом и отличаются методы оптимизации) более правильно и позволяет лучше избегать переобучения нейросетевых моделей.

4.3 АЛГОРИТМ ОБУЧЕНИЯ ПРОБНОЙ ФУНКЦИИ

Пускай инициализирована пробная функция $\mathcal{F}$, чтобы обучить её в течение `STEPS` итерация требуется выполнить `STEPS` раз следующую последовательность действий:

1. Если итерация первая, то происходит генерация набора точек в координатном пространстве, при этом точки генерируются равномерно распределенные в кубе со сторой `std` (это гиперпараметр модели); если же итерация не первая, то к набору точек в координатном пространстве несколько раз применяет шаг алгоритма Метрополиса (с функцией плотности распределения W из (4.1));

2. Рассчитывается функционал потерь (3.1);

3. Вызывается оптимизатор (AdamW), изменяющий обучаемые параметры пробной функции из принципа минимизации функционала потерь.

5 РЕЗУЛЬТАТЫ

В данном разделе будут приведены результаты обучения пробных функций в различных размерностях. С помощью пробной функции решается тестовая задача (1.1) в соответствующей размерности. Исходный код пробной функции и процедуры ее обучения приведен в Приложении в Листинге 2.

5.1 3-МЕРНЫЙ СЛУЧАЙ

Для решения 3-мерной задачи была выбрана пробная функция с числом нейронов на каждом внутреннем слое $N_h = 100$. Отыскивалось $M = 5$ нижних по энергиям состояний. Весовые факторы в функционале потерь (3.1) были выбраны следующие:

$$w_R = 1,\ w_A = 1,\ w_B = 40,\ w_E = 1, \tag{5.1}$$

что соответствует выбору весовых множителей в функционале потерь в работе [5]. Полный список значений гиперпараметров пробной функции, которая использовалась при решении задачи в 3-мерном случае, приведен в Таблице 1. Обучение модели происходило на ноутбуке с графическим ускорителем NVIDIA GeForce RTX 3050 Laptop GPU, обладающим 3901MB памяти.

Имя	Значение
Функция активации	sin
Амплитудная функция	$\exp\left(-\sum_{d=1}^{d} x^2/2\right)$
Число внутренних слоев	3
N_h	100
D	3
M	5
`std`	3
B	2^{13}
`learningRate`	10^{-3}
`weightDecay`	10^{-3}
w_R	1
w_A	1
w_B	40
w_E	1
ε	1

Таблица 1: Гиперпараметры пробной функции, которая использовалась при решении задачи о квантово-механическом гармоническом осцилляторе в 3-мерном случае.

Данная пробная функция обучалась в течение `STEPS` = 15000 итераций, что заняло 50 минут и 45 секунд. Зависимость оценочных значений собственных значений задачи от номера итерации обучения приведена на верхней панели Рис. 1. Видно, что за 15000 итераций обучения пробная функция приближается к решению задачи. Оценочные значения собственных значений на последней итерации обучения, а также их относительные

ошибки приведены в Таблице 2. Видно, что нижние 4 состояния в результате обучения обеспечивают относительную ошибку по собственным значениям примерно в 0.5%, в то время как 4-ое состояние обеспечивает относительную ошибку выше — примерно в 3%.

На нижней панели Рис. 1 приведена зависимость значений членов функционала потерь от номера итерации обучения, что позволяет заключить, что на первых этапах обучения (до 4000-ой итерации) оптимизировались лишь члены, отвечающие за невязку по уравнению (R-term) и ортогонализацию (B-term); после этого уменьшение члена, отвечающего за ортогонализацию прекращается, он начинает возрастать, в то время как член, отвечающий за выполнение нормировки (A-term), начинает уменьшаться; на 10000-ой итерации оба эти члена стабилизируются и более сильно не меняются, а уменьшение члена, отвечающего за невязку по уравнению (R-term), продолжается до самого завершения обучения.

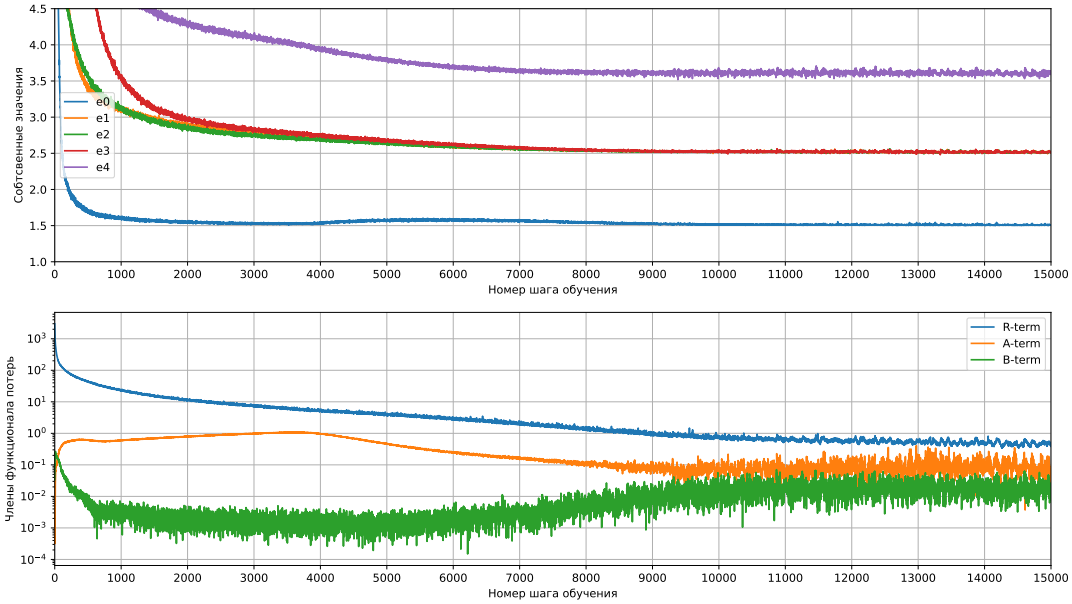

Рис. 1: (Верхняя панель): Зависимость оценочных значений собственных значений задачи (1.1) в 3-мерном случае от номера итерации обучения; (нижняя панель): зависимость значений членов функционала потерь от номера итерации обучения.

m	E_m^{approx}	$\Delta E_m / E_m$
0	1.5078	0.0052
1	2.5137	0.0055
2	2.5160	0.0064
3	2.5113	0.0045
4	3.6113	0.0318

Таблица 2: Оценочные значения собственных значений на последней итерации обучения и их относительные ошибки, полученные в результате обучения пробной функции с гиперпараметрами, указанными в Таблице 1, в течение `STEPS` = 15000 итераций.

5.2 4-МЕРНЫЙ СЛУЧАЙ

Для решения 4-мерной задачи была выбрана пробная функция с числом нейронов на каждом внутреннем слое $N_h = 100$. Отыскивалось $M = 5$ нижних по энергиям состояний. Весовые факторы в функционале потерь (3.1) были выбраны следующие:

$$w_R = 1,\ w_A = 1,\ w_B = 40,\ w_E = 1, \tag{5.2}$$

что соответствует выбору весовых множителей в функционале потерь в работе [5]. Полный список значений гиперпараметров пробной функции, которая использовалась при решении задачи в 4-мерном случае, приведен в Таблице 3. Обучение модели происходило на сервисе Kaggle (см. `https://www.kaggle.com/`) с графическим ускорителем Tesla P100-PCIE-16GB, обладающим 15.9GB памяти.

Имя	Значение
Функция активации	sin
Амплитудная функция	$\exp\left(-\sum_{d=1}^{d} x^2/2\right)$
Число внутренних слоев	4
N_h	100
D	3
M	5
`std`	3
B	2^{14}
`learningRate`	10^{-3}
`weightDecay`	10^{-3}
w_R	1
w_A	1
w_B	40
w_E	1
ε	1

Таблица 3: Гиперпараметры пробной функции, которая использовалась при решении задачи о квантово-механическом гармоническом осцилляторе в 3-мерном случае.

Данная пробная функция обучалась в течение `STEPS` = 15000 итераций, что заняло 1 час 23 минуты и 11 секунд. Зависимость оценочных значений собственных значений задачи от номера итерации обучения приведена на верхней панели Рис. 2. Видно, что за 15000 итераций обучения пробная функция приближается к решению задачи. Оценочные значения собственных значений на последней итерации обучения, а также их относительные ошибки приведены в Таблице 4. Видно, что энергия основного состояния определилась с относительной погрешностью в 0.4%, а энергии первого возбужденного состояния определились с точностью примерно в 1%.

На нижней панели Рис. 2 приведена зависимость значений членов функционала потерь от номера итерации обучения, что позволяет заключить, что на первых этапах обучения (примерно до 2000-ой итерации) оптимизировались лишь члены, отвечающие за

невязку по уравнению (R-term) и ортогонализацию (B-term); после этого начинает уменьшаться и член, отвечающий за нормировку пробной функции (A-term); на примерно 5700-ой итерации член, отвечающий на нормировку, прекращает уменьшаться, возрастая до примерно 7500-ой итерации, примерно в это же время член, отвечающий за ортогонализацию, прекращает уменьшаться и начинает возрастать, что он делает до самого конца обучения; член, отвечающий за нормировку, примерно с 7500-ой итерации начинает уменьшаться, что делает до самого окончания обучения; член, отвечающий за невязку по уравнению, все время обучения уменьшается.

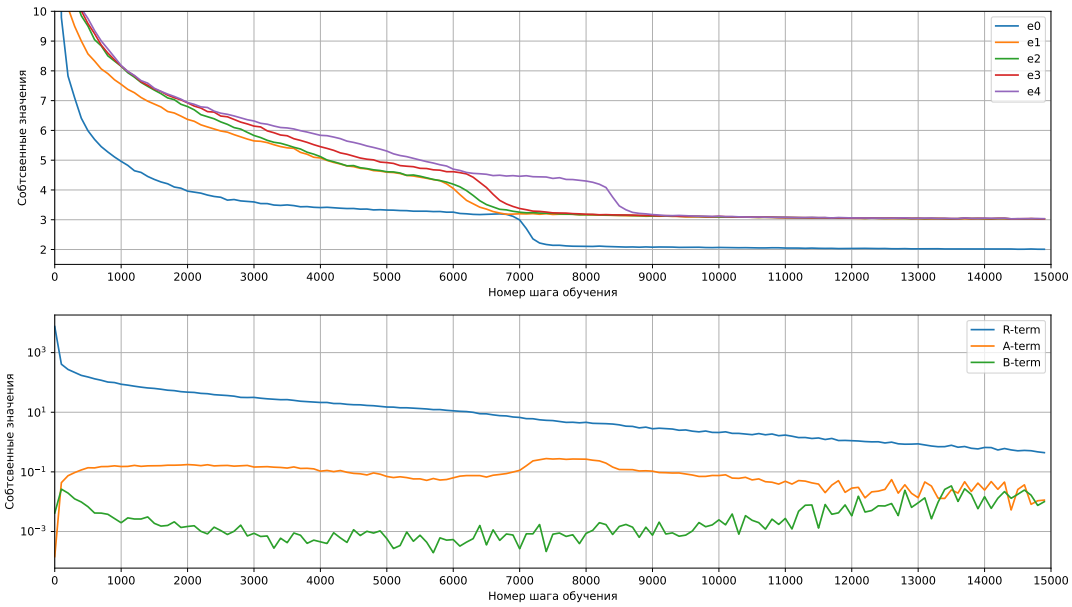

Рис. 2: (Верхняя панель): Зависимость оценочных значений собственных значений задачи (1.1) в 4-мерном случае от номера итерации обучения; (нижняя панель): зависимость значений членов функционала потерь от номера итерации обучения.

m	E_m^{approx}	$\Delta E_m / E_m$
0	2.0081	0.0040
1	3.0259	0.0086
2	3.0266	0.0089
3	3.0267	0.0089
4	3.0350	0.0117

Таблица 4: Оценочные значения собственных значений на последней итерации обучения и их относительные ошибки, полученные в результате обучения пробной функции с гиперпараметрами, указанными в Таблице 3, в течение `STEPS` = 15000 итераций.

ЗАКЛЮЧЕНИЕ

В данной работе были развиты идеи работы [5], улучшен подход к решению спектральных задач квантовой механики в пространствах высокой размерности и успешно решены 3-мерная и 4-мерная задачи. А именно (не считая смены архитектуры нейронной сети) было добавлено динамическое изменение распределения точек в координатном пространстве согласно усредненному по состояниям квадрату модуля волновой функции (для чего использовался алгоритм Метрополиса). Внесенная модификация позволила обучаться модель преимущественно в той области координатного пространства, где выше плотность вероятности и волновые функции сильнее всего отличны от нуля (то есть в тех областях, которые наиболее важны для усешной аппроксимации решения). Такой метод семплирования особенно важен в пространствах высокой размерности, так как с ростом размерности координатного пространства объём области, где волновые функции значительно отличны от нуля, экспоненциально падает, что значительно осложняет использование методов, завязанных на равномерном распределении точек в координатном пространстве.

В будущем предполагается исследование больших размерностей, однако эти требует и больших вычислительных мощностей (требуется большой объем памяти на графическом ускорителе). В настоящей работе рассматривались примеры с лишь 5 состояниями, в дальнейшем было бы интересно решать задачи с большим числом состояний. Кроме того, было бы интересно отработать методику на различных спектральных задачах.

СПИСОК ЛИТЕРАТУРЫ

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[2] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[3] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural network methods in quantum mechanics. *Computer Physics Communications*, 104(1):1–14, 1997.

[4] Hong Li, Qilong Zhai, and Jeff Z. Y. Chen. Neural-network-based multistate solver for a static schrödinger equation. *Phys. Rev. A*, 103:032405, Mar 2021.

[5] Цыренов Эрдэм Цыденжапович. Выпускная квалификационная работа. Численное решение стационарного уравнения Шредингера с использованием искусственных нейронных сетей. *СПбГУ*, 2022.

[6] Лифшиц Е.М. Ландау Л.Д. *Том 3. Квантовая механика. Нерелятивистская теория.* Теоретическая физика в десяти томах. М.: Наука. Гл. ред. физ-мат. лит., 4-е изд., испр., 1986.

[7] Eric W. Weisstein. Hermite polynomial. From MathWorld—A Wolfram Web Resource. `https://mathworld.wolfram.com/HermitePolynomial.html`.

[8] Bernd A. Berg. Introduction to markov chain monte carlo simulations and their statistical analysis, 2004.

[9] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

ПРИЛОЖЕНИЕ

Листинг 2: Исходный код класса пробной функции и процедуры обучения пробной функции, написанный на языке Python с использованием фреймворка PyTorch.

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and device detection
# Date time
import datetime as dt

# System
import sys

# Time
import time

# NumPy
import numpy as np

# Pandas
import pandas as pd

# PyTorch framework
import torch
from torch import nn

# Device
if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Using {device}: {torch.cuda.get_device_name()}")
else:
    device = torch.device("cpu")
    print("No GPU found, using cpu")


# # Parameters
# Dictionary of model parameters
parameters = {}


def fillParameters(
    AF: str,  # Activation function
    AMPLITUDE: str,  # Amplitude function
    Nh: int,  # Number of nodes in each hidden layer
    D: int,  # Dimension of the coordinate space
    M: int,  # Number of states we want to find
    std: float,  # Standard deviation of sample distribution
    B: int,  # Batch size
    learningRate: float,  # Starting value of the learning rate (we use ADAM
    # optimizer)
    weightDecay: float,  # Weight decay (we use ADAM optimizer)
    wR: float,  # Residual term weight
    wA: float,  # Normalisation term weight
    wB: float,  # Orthogonalisation term weight
    wE: float,  # Energy term weight
    epsilon: float,  # Metropolis algorithm parameter
):
    """Fills dictionary of model parameters"""
```

```
    global parameters
    parameters["AF"] = AF
    parameters["AMPLITUDE"] = AMPLITUDE
    parameters["Nh"] = Nh
    parameters["D"] = D
    parameters["M"] = M
    parameters["std"] = std
    parameters["B"] = B
    parameters["learningRate"] = learningRate
    parameters["weightDecay"] = weightDecay
    parameters["wR"] = wR
    parameters["wA"] = wA
    parameters["wB"] = wB
    parameters["wE"] = wE
    s = "staticWeights"
    for key in parameters.keys():
        if key != "configuration":
            s = s + f"{key}{parameters[key]}"
    parameters["configuration"] = s  # Model configuration


# # Activation functions


class Sin(nn.Module):
    """Custom sin activation function class"""

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return torch.sin(x)

    def diff(self, x: torch.Tensor) -> torch.Tensor:
        return torch.cos(x)

    def diff2(self, x: torch.Tensor) -> torch.Tensor:
        return -torch.sin(x)


class Tanh(nn.Module):
    """Custom tanh activation function class"""

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return torch.tanh(x)

    def diff(self, x: torch.Tensor) -> torch.Tensor:
        return 1 / torch.cosh(x) ** 2

    def diff2(self, x: torch.Tensor) -> torch.Tensor:
        return -2 * torch.sinh(x) / torch.cosh(x) ** 3


# # Amplitude functions


class Gaussian(nn.Module):
    """Custom Gaussian amplitude function class"""

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong␣shape␣0␣of␣the␣tensor␣x"
        assert x.shape[1] == parameters["D"], "Wrong␣shape␣1␣of␣the␣tensor␣x"
        result = torch.ones((x.shape[0], parameters["M"])).to(device)
```

```
        result = torch.mul(result.t(), torch.exp(-torch.sum(x**2, axis=1) / 2)).t()
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result

    def gradient(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["D"], parameters["M"])).to(device)
        for d in range(parameters["D"]):
            # first term
            a = torch.ones((x.shape[0], parameters["M"])).to(device)
            a = torch.mul(
                a.t(),
                torch.exp(-torch.sum(x**2, axis=1) / 2) * (-2 * x[:, d] / 2),
            ).t()
            # keeping result
            result[:, d, :] = a
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["D"], "Wrong shape 1 of the result"
        assert result.shape[2] == parameters["M"], "Wrong shape 2 of the result"
        return result

    def laplacian(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["M"])).to(device)
        result = torch.mul(
            result.t(),
            torch.exp(-torch.sum(x**2, axis=1) / 2)
            * (
                torch.sum((-2 * x / 2) ** 2, axis=1)
                - 2 * torch.sum(torch.ones_like(x) / 2, axis=1)
            ),
        ).t()
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result


class Exponent4(nn.Module):
    """Custom Exponent amplitude function class"""

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["M"])).to(device)
        result = torch.mul(
            result.t(),
            torch.exp(-torch.sum(x**4, axis=1) / parameters["std"] ** 4),
        ).t()
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result

    def gradient(self, x: torch.Tensor) -> torch.Tensor:
```

```
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["D"], parameters["M"])).to(device)
        for d in range(parameters["D"]):
            # first term
            a = torch.ones((x.shape[0], parameters["M"])).to(device)
            a = torch.mul(
                a.t(),
                torch.exp(-torch.sum(x**4, axis=1) / parameters["std"] ** 4)
                * (-4 * x[:, d] ** 3 / parameters["std"] ** 4),
            ).t()
            # keeping result
            result[:, d, :] = a
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["D"], "Wrong shape 1 of the result"
        assert result.shape[2] == parameters["M"], "Wrong shape 2 of the result"
        return result

    def laplacian(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["M"])).to(device)
        result = torch.mul(
            result.t(),
            torch.exp(-torch.sum(x**4, axis=1) / parameters["std"] ** 4)
            * (
                torch.sum((-4 * x**3 / parameters["std"] ** 4) ** 2, axis=1)
                - 12 * torch.sum(x**2 / parameters["std"] ** 4, axis=1)
            ),
        ).t()
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result


# # Neural network (NN)


class NN(nn.Module):
    """Class for a neural network"""

    def __init__(self):
        super().__init__()
        global parameters
        if parameters["AF"] == "sin":
            self.AF = Sin()
        elif parameters["AF"] == "tanh":
            self.AF = Tanh()
        else:
            print("Wrong AF")
            sys.exit(1)
        # NUMBER OF HIDDEN LAYERS IS ASSUMED TO BE 3
        self.stack = nn.Sequential(
            nn.Linear(parameters["D"], parameters["Nh"]),
            self.AF,
            nn.Linear(parameters["Nh"], parameters["Nh"]),
            self.AF,
            nn.Linear(parameters["Nh"], parameters["Nh"]),
```

```
            self.AF,
            nn.Linear(parameters["Nh"], parameters["M"]),
            self.AF,
        )
        for i in range(len(self.stack)):
            if "weight" in dir(self.stack[i]):
                torch.nn.init.normal_(self.stack[i].weight, 0, np.sqrt(0.1))
        global device
        # chooseDevice()
        self.stack.to(device)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = self.stack(x).to(device)
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result

    def gradient(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["D"], parameters["M"])).to(device)
        for d in range(parameters["D"]):
            ydiff = self.AF.diff(self.stack[0](x)) * self.stack[0].weight[:, d]
            y = self.stack[0 * 2 + 1](self.stack[0 * 2](x))
            # NUMBER OF HIDDEN LAYERS IS ASSUMED TO BE 3
            for i in range(1, 3 + 1):
                ydiff = self.AF.diff(self.stack[i * 2](y)) * torch.matmul(
                    ydiff, self.stack[i * 2].weight.t()
                )
                y = self.stack[i * 2 + 1](self.stack[i * 2](y))
            # keeping result
            result[:, d, :] = ydiff
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["D"], "Wrong shape 1 of the result"
        assert result.shape[2] == parameters["M"], "Wrong shape 2 of the result"
        return result

    def laplacian(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        preresult = torch.ones((x.shape[0], parameters["D"], parameters["M"])).to(
            device
        )
        for d in range(parameters["D"]):
            ydiff = self.AF.diff(self.stack[0](x)) * self.stack[0].weight[:, d]
            ydiff2 = self.AF.diff2(self.stack[0](x)) * self.stack[0].weight[:, d] ** 2
            y = self.stack[0 * 2 + 1](self.stack[0 * 2](x))
            # NUMBER OF HIDDEN LAYERS IS ASSUMED TO BE 3
            for i in range(1, 3 + 1):
                # second derivative
                ydiff2 = self.AF.diff2(self.stack[i * 2](y)) * torch.matmul(
                    ydiff, self.stack[i * 2].weight.t()
                ) ** 2 + self.AF.diff(self.stack[i * 2](y)) * torch.matmul(
                    ydiff2, self.stack[i * 2].weight.t()
```

```
                )
                # first derivative
                ydiff = self.AF.diff(self.stack[i * 2](y)) * torch.matmul(
                    ydiff, self.stack[i * 2].weight.t()
                )
                # output of the current layer
                y = self.stack[i * 2 + 1](self.stack[i * 2](y))
            # Keeping preresult
            preresult[:, d, :] = ydiff2
        result = torch.sum(preresult, axis=1)
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result


# # Trial function


class TrialFunction(nn.Module):
    """Trial function class"""

    def __init__(self):
        super().__init__()
        self.nn = NN()
        global parameters
        if parameters["AMPLITUDE"] == "gaussian":
            self.amplitude = Gaussian()
        elif parameters["AMPLITUDE"] == "exponent4":
            self.amplitude = Exponent4()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = self.nn(x).to(device) * self.amplitude(x).to(device)
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["M"], "Wrong shape 1 of the result"
        return result

    def gradient(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
        result = torch.ones((x.shape[0], parameters["D"], parameters["M"])).to(device)
        a = self.amplitude(x)
        ag = self.amplitude.gradient(x)
        nn = self.nn(x)
        nng = self.nn.gradient(x)
        for d in range(parameters["D"]):
            result[:, d, :] = nng[:, d, :] * a + nn * ag[:, d, :]
        assert result.shape[0] == x.shape[0], "Wrong shape 0 of the result"
        assert result.shape[1] == parameters["D"], "Wrong shape 1 of the result"
        assert result.shape[2] == parameters["M"], "Wrong shape 2 of the result"
        return result

    def laplacian(self, x: torch.Tensor) -> torch.Tensor:
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong shape 0 of the tensor x"
        assert x.shape[1] == parameters["D"], "Wrong shape 1 of the tensor x"
```

```
        result = (
            self.nn.laplacian(x) * self.amplitude(x)
            + self.nn(x) * self.amplitude.laplacian(x)
            + 2 * torch.sum(self.nn.gradient(x) * self.amplitude.gradient(x), axis=1)
        )
        assert result.shape[0] == x.shape[0], "Wrong␣shape␣0␣of␣the␣result"
        assert result.shape[1] == parameters["M"], "Wrong␣shape␣1␣of␣the␣result"
        return result

    def weight(self, x: torch.Tensor) -> torch.Tensor:
        """Weight function"""
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong␣shape␣0␣of␣the␣tensor␣x"
        assert x.shape[1] == parameters["D"], "Wrong␣shape␣1␣of␣the␣tensor␣x"
        result = torch.mean(self.forward(x) ** 2, axis=1)
        # result = (
        #     1
        #     / (parameters["std"] * float(np.sqrt(2 * np.pi))) ** parameters["D"]
        #     * torch.exp(-0.5 * torch.sum(x * x, axis=1) / parameters["std"] ** 2)
        # )
        assert result.shape[0] == x.shape[0], "Wrong␣shape␣0␣of␣the␣result"
        return result

    def l2Norm(self, x: torch.Tensor) -> torch.Tensor:
        """Computes L2-norm of the trial function"""
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong␣shape␣0␣of␣the␣tensor␣x"
        assert x.shape[1] == parameters["D"], "Wrong␣shape␣1␣of␣the␣tensor␣x"
        result = torch.zeros(parameters["M"]).to(device)
        f = self.forward(x)
        w = self.weight(x)
        for m in range(parameters["M"]):
            result[m] = torch.mean(abs(f[:, m]) ** 2 / w)
        assert result.shape[0] == parameters["M"], "Wrong␣shape␣0␣of␣the␣result"
        return result

    def spectrum(self, x: torch.Tensor) -> torch.Tensor:
        """Finds current spectrum"""
        global parameters
        global device
        assert x.shape[0] > 0, "Wrong␣shape␣0␣of␣the␣tensor␣x"
        assert x.shape[1] == parameters["D"], "Wrong␣shape␣1␣of␣the␣tensor␣x"
        result = torch.zeros(parameters["M"]).to(device)
        f = self.forward(x)
        lap = self.laplacian(x)
        w = self.weight(x)
        v = 0.5 * torch.sum(x * x, axis=1)
        for m in range(parameters["M"]):
            result[m] = torch.mean(f[:, m] * (-0.5 * lap[:, m] + v * f[:, m]) / w)
        result = result / self.l2Norm(x)
        assert result.shape[0] == parameters["M"], "Wrong␣shape␣0␣of␣the␣result"
        return result

    def save(self):
        """Saves model weights and biases"""
        torch.save(
            self.state_dict(),
            "../models/" + parameters["configuration"] + ".pt",
        )

```

```
    def load(self):
        """Loads model weights and biases"""
        self.load_state_dict("../models/" + parameters["configuration"] + ".pt")


# # Metropolis algorithm


def MetropolisAlghorithmStep(x, f):
    """One step of the Metropolis algorithm, it return the updated sample x"""
    global parameters
    x_prime = x + parameters["epsilon"] * (2 * torch.rand_like(x).to(device) - 1)
    alpha = f(x_prime).to(device) / f(x).to(device)
    doesXMove = torch.rand((x.shape[0])).to(device) <= alpha
    x_prime = (
        torch.mul(doesXMove.int(), x_prime.t()).t()
        + torch.mul((1 - doesXMove.int()), x.t()).t()
    )
    return x_prime


# # Training


def training(
    trialFunction: TrialFunction,  # Trial function we will train
    steps: int,  # Number of steps
):
    """Trains trial function"""
    global parameters
    global device
    print("Model configuration:")
    print(parameters["configuration"])
    print("")
    print("Training has been started...")
    print("")
    print(
        "Step, Time [s], Residual-term, Normalization-term,"
        + " Orthogonalization-term, Sorted energies"
    )
    time0 = time.time()
    time_history = np.full((steps), np.nan)
    loss_history = np.full((steps), np.nan)
    r_history = np.full((steps), np.nan)
    a_history = np.full((steps), np.nan)
    b_history = np.full((steps), np.nan)
    spectrum_history = np.full((parameters["M"], steps), np.nan)
    optimizer = torch.optim.AdamW(
        params=list(trialFunction.parameters()),
        lr=parameters["learningRate"],
        weight_decay=parameters["weightDecay"],
    )
    lossFunction = nn.MSELoss()
    for step in range(steps):
        # Sample
        if step == 0:
            x = parameters["std"] * (
                torch.rand((parameters["B"], parameters["D"])) - 0.5
            ).to(device)
        else:
            for i in range(10):
                x = MetropolisAlghorithmStep(x, trialFunction.weight)
```

```
        # Spectrum (Energies)
        e = trialFunction.spectrum(x)
        # Forward (trial function)
        f = trialFunction(x)
        # Laplacian
        lap = trialFunction.laplacian(x)
        # Weight function
        w = trialFunction.weight(x)
        # Squared norm
        sqrNorm = trialFunction.l2Norm(x)
        # R-term
        r = sum(
            [
                torch.mean(
                    (
                        -0.5 * lap[:, m]
                        + 0.5 * f[:, m] * torch.sum(x**2, axis=1)
                        - f[:, m] * e[m]
                    )
                    ** 2
                    / w
                )
                / sqrNorm[m]
                for m in range(parameters["M"])
            ]
        )
        # A-term
        a = torch.sum((sqrNorm - torch.tensor(1).to(device)) ** 2)
        # B-term
        b = torch.zeros(parameters["M"], parameters["M"]).to(device)
        for m1 in range(1, parameters["M"]):
            for m2 in range(0, m1):
                maxs = (torch.max(f[:, m1]) * torch.max(f[:, m2])).detach() + 0.001
                torch.Tensor.detach(maxs)
                b[m1, m2] = (
                    torch.square(torch.mean(f[:, m1] * f[:, m2] / w))
                    / sqrNorm[m1]
                    / sqrNorm[m2]
                )
        # The total loss
        loss = lossFunction(
            parameters["wR"] * r
            + parameters["wA"] * a
            + parameters["wB"] * torch.sum(b)
            + parameters["wE"] * torch.sum(e),
            torch.tensor(0.0).to(device),
        )
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Save a state of the model
        time_ = time.time() - time0
        time_history[step] = time_
        if np.isnan(loss.item()):
            print("LOSS␣IS␣NAN")
            return
        loss_history[step] = loss.item()
        r_history[step] = r.cpu().detach().numpy()
        a_history[step] = a.cpu().detach().numpy()
        b_history[step] = torch.sum(b).cpu().detach().numpy()
        spectrum_history[:, step] = e.cpu().detach().numpy()
```

```
        # Release some memory
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
        # Print the state of the model
        if step % 100 == 0:
            print(
                f"{step}, {time_:.2f}, {r_history[step]:.4e},"
                + f" {a_history[step]:.4e}, {b_history[step]:.4e},"
                + f" {np.sort(spectrum_history[:, step])}"
            )
    # Save history
    d = {
        "Time[s]": time_history,
        "R-term": r_history,
        "A-term": a_history,
        "B-term": b_history,
    }
    for i in range(parameters["M"]):
        d[f"e{i}"] = spectrum_history[i, :]
    history_df = pd.DataFrame(data=d)
    now = dt.datetime.now()
    dt_string = now.strftime("%d.%m.%Y_%H:%M:%S")
    history_df.to_csv("../data/" + dt_string + parameters["configuration"] + ".csv")
    return history_df
```