МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

РЕФЕРАТ

Градиентные методы обучения нейронных сетей первого порядка

студента 5 курса 531 группы
направления 10.05.01—Компьютерная безопасность
факультета КНиИТ
Сенокосова Владислава Владимировича

Проверил
доцент _____ И. И. Слеповичев
подпись, дата

Саратов 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На сегодняшний день представить жизнь без использования цифровых технологий кажется невозможным. Ежедневно появляются новые открытия и технологии, способствующие активному развитию программного обеспечения, которое внедряется в различные аспекты повседневной жизни. Практически вся жизнедеятельность человека перенесена в цифровой мир, что значительно улучшило скорость и качество выполнения множества поставленных перед человеком задач.

Одним из важнейших достижений технологического прогресса стал искусственный интеллект, активно применяемый в различных информационных структурах. Основой функционирования искусственного интеллекта является базовое понимание и использование нейронных сетей. Для обучения таких сетей с целью анализа данных и получения корректных результатов используются различные методы, среди которых выделяются градиентные методы обучения.

Градиентные методы первого порядка являются ключевым инструментом настройки параметров нейронных сетей, обеспечивая оптимизацию функции ошибки. Современные подходы основаны на итеративных процессах, где на каждом шаге вычисляется градиент, определяющий направление изменения параметров. Эти методы находят применение как в классических задачах машинного обучения, так и в современных глубоких нейронных сетях.

1. Формальное описание градиента и его сущность

В повседневной жизни часто возникают задачи, требующие оценки некоторых характеристик, которые могут в течение времени принимать различные значения. Интересно также знать характер их поведения, то есть возрастают они или убывают, поэтому для их определения используется мощный аппарат математического анализа – градиент.

Градиентом называется характеристика, указывающая направление наибольшего роста некоторой величины, значение которой меняется от одной точки пространства к другой. В математике градиентом функции $f(x_1, x_2, \dots, x_n)$, определенной на R^n, представляет собой вектор, компоненты которого равны частным производным функции f по её аргументам и обозначается ∇f или $grad(f)$. Поэтому можно записать градиент для f как [13]:

$$\nabla f(x_1, x_2, \dots, x_n) = (\frac{df}{dx_1}, \frac{df}{dx_2}, \dots, \frac{df}{dx_n})$$

где $\frac{df}{dx_i}$, частная производная функции f по переменной x_i, $i = 1, .., n$.

Геометрически градиент указывает направление наиболее быстрого возрастания функции, а его норма (длина) соответствует скорости изменения функции в этом направлении. В геометрических задачах происходит разложение градиента через базисные вектора, с целью его дальнейшего применения, поэтому определим градиент с учетом базисных векторов векторного пространства.[5] Пусть ∇f – градиент функции $f(x_1, x_2, \dots, x_n)$, а $(e_1, e_2, .., e_n)$ – ортонормированный базис в R^n. Тогда градиент можно разложить по этому базису следующим образом:

$$\nabla f = \sum_{i=1}^{n} \frac{df}{dx_i} e_i,$$

где $\frac{df}{dx_i}$, частная производная f по координате x_i, e_i – соответствующий базисный вектор.

Смысл градиента любой скалярной функции f состоит в том, что его произведение с бесконечно малым вектором перемещения dx с компонентами $dx_1, dx_2, \dots, dx_n$ дает полный дифференциал этой функции [14]:

$$df = \frac{df}{dx_1}dx_1 + \frac{df}{dx_2}dx_2 + \cdots + \frac{df}{dx_n}dx_n = grad(f) \cdot dx$$

2. Градиентные методы обучения нейронных сетей первого порядка

Для того, чтобы модель нейронной системы решала некоторую поставленную задачу ее нужно правильно обучить. Под обучением нейронной сети понимается настройка её параметров (весов и смещений) таким образом, чтобы минимизировать функцию потерь. Этот процесс позволяет сети корректно преобразовывать входные данные в желаемый выход и эффективно решать поставленную задачу. Минимизация функции потерь достигается путем поиска противоположного вектора вектору градиента, который и указывает направление уменьшения функции [1].

На сегодняшний день существует большое количество градиентных методов первого порядка для оптимизации функции потерь, каждый из которых имеет свои отличительные особенности. На момент 2020 года их распределение по использованию имело следующий вид (См. Рис.1):

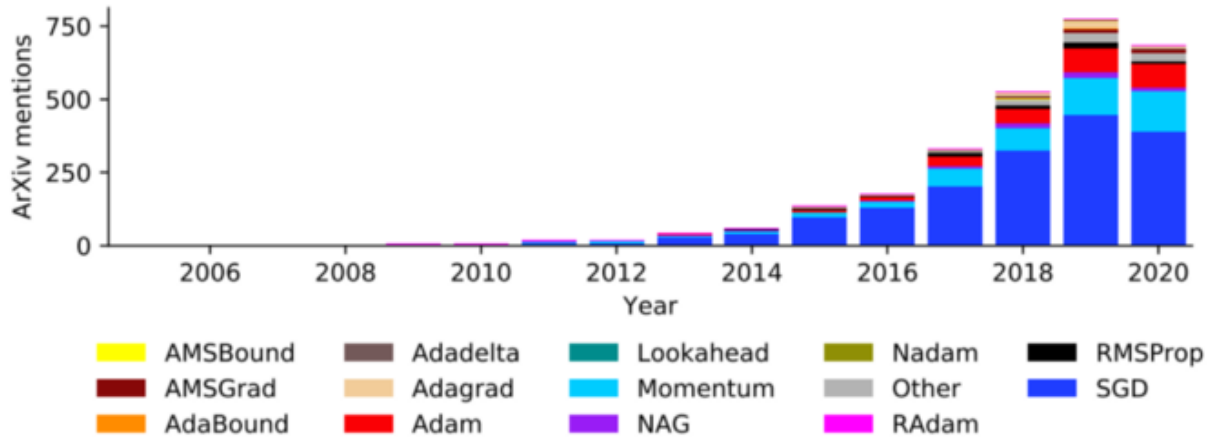

Рисунок 1 – Рейтинг оптимизаторов за 2006 – 2020 года [10]

Можно заметить, что достаточно долгое время лидируют SGD, Momentum и Adam. Они и другие методы будут рассмотрены далее в работе. Для дальнейшего исследования обозначим основные элементы нейронной сети:

X – некоторая входная выборка (тестируемый набор данных);

C – множество классов (результат в результате работы нейронной сети);

W – множество весов нейронной сети;

h – сама нейронная сеть, решающая некоторую задачу (зависящая от X и W).

Под функциями потерь для нейронных сетей обычно используется среднеквадратическая ошибка или средняя кросс-энтропия [6].

Среднеквадратическая ошибка:

$$E = \frac{1}{2}\sum_{j}\left(y_j - c_j\right)^2$$

где y_j – выход сети номер j, c_j – правильный ответ для выхода j (который заранее известен исследователю).

Когда в качестве результата работы нейронной сети используются вероятности, то предпочтительней использовать среднюю кросс-энтропию:

$$E = \frac{1}{k}\sum_{i=1}^{k}(-\log(p_i))$$

где k - количество примеров, p_i- выданная нейронной сетью вероятность принадлежности примера X_i к своему рассматриваемому классу ответов C_i.

Тогда, имея функцию E, мы теперь можем формально поставить задачу обучения нейронной сети $h(X, W)$ следующим образом: обучение нейронной сети есть минимизация функции потери в пространстве весов.

$$\min_{W} E(h(X, W), C)$$

Для того, чтобы минимизировать значение функции необходимо градиент функции $\nabla E(W)$ в точке W направить в противоположную сторону и изменить параметры W на это смещение (полученное из $\nabla E(W)$). Этот метод называется методом градиентного спуска.

Большинство градиентных методов основывается на следующей формуле:

$$\Delta W = \eta \nabla E$$

где η – параметр, который называется «скоростью обучения», он определяет величину шага процесса оптимизации. Определить какой именно необходим параметр достаточно сложно, поэтому их подбор — это отдельно решаемая задача. Для упрощенного подбора используют значения $0 < \eta < 1$, однако сходимость к минимуму может быть не такой быстрой при случайно выбранном значении η [8].

2.1. Метод обратного распространения ошибки

Для решения задачи оптимизации методом градиентного спуска, описанном в предыдущем разделе, нам необходимо найти способ вычисления градиента функции потери ∇E, который представляет собой вектор частных производных [6].

$$\nabla E(W) = \left[\frac{dE}{dw_1}, \ldots, \frac{dE}{dw_k}\right]$$

где k – общее количество весов в сети.

Для случая нейронной сети градиент можно записать следующим образом:

$$\frac{dE}{dw_{ij}} = \frac{dE}{dy_j}\frac{dy_j}{ds_j}\frac{ds_j}{dw_{ij}}$$

где E – функция потери, w_{ij} – вес связи нейронов i и j, y_j – выход нейрона номер j, s_j – состояние нейрона j.

Опишем каждую из компонент:

1. $\frac{ds_j}{dw_{ij}}$ – выход i-того нейрона предыдущего (по отношению к нейрону j) слоя, эта часть определена в явном виде;

2. $\frac{dy_j}{ds_j}$ – значение производной активационной функции по ее аргументу для нейрона j, эту часть достаточно просто можно вычислить;

3. $\frac{dE}{dy_j}$ – ошибка нейрона номер j, здесь возникают некоторые затруднения. Значения ошибки определены явно только для нейронов выходного слоя. А по формуле нужно получить для скрытых слоев. В этом случае и применяется метод обратного распространения ошибки.[8]

Суть его заключается в последовательном вычислении ошибок скрытых слоёв с помощью значений ошибки выходного слоя, т.е. значения ошибки распространяются по сети в обратном направлении от выхода к входу. Вычисления производятся по следующим формулам:

Для выходного слоя: $\delta_i = \frac{dE}{dy_i}$.

Для скрытого слоя: $\delta_i = \frac{dy_i}{ds_i} \sum_j \delta_i w_{ij}$.

где $\frac{dy}{ds}$ – значение производной функции активации

Представим алгоритм расчета градиента функции потерь методом обратного распространения ошибки:

1. прямой проход: вычислить состояния нейронов s всех слоёв и выход сети;

2. вычисляем значения $\delta = \frac{dE}{dy}$ для выходного слоя;

3. обратный проход: последовательно от конца к началу для всех скрытых слоёв вычисляем $\delta_i = \frac{dy_i}{ds_i}\sum_j \delta_i w_{ij}$;

4. для каждого слоя вычисляем значение градиента $\nabla E = \frac{dE}{dW} = y \cdot \delta^T$ где y – вектор-вход слоя, δ – вектор ошибки на выходе слоя.

2.2. Модифицированный метод градиентного спуска

При применении градиентных методов обучения нейронных сетей первого порядка используется следующие 3 подхода обработки примеров обучающих выборок [4]:

1. **SGD (Stochastic Gradient Descent)** — это метод оптимизации, корректирующий веса нейронной сети с помощью вычисленного градиента функции ошибки на одном случайном обучающем примере. При таком подходе обучения шаги обучения происходят чаще, а также основным преимуществом является отсутствие необходимости хранить множество всех градиентов обучающих примеров [4].
2. **BatchGD (Batch Gradient Descent) –** метод оптимизации, при котором градиент функции ошибки высчитывается как сумма всех градиентов для всего множества обучающих примеров. Часто при описании базового градиентного спуска понимается именно данный тип. Данный метод плох в случаях, когда функция ошибки не является гладкой и имеет множество локальных минимумов, что приводит к застреванию в локальных минимумах. При большом наборе обучающих данных значительно возрастает потребление памяти [16].
3. **Mini-batch –** это метод оптимизации, совмещающий в себе подходы BatchGD и SGD, в этом случае изменение параметров весов происходит за счет вычисления градиентов от функции ошибки

некоторого небольшого подмножества примеров обучающей выборки. Это снижает вычислительные затраты и уменьшает шум в оценке градиента, обеспечивая баланс между точностью и скоростью.

Графическое представление каждого из методов представлено на рисунке 2.

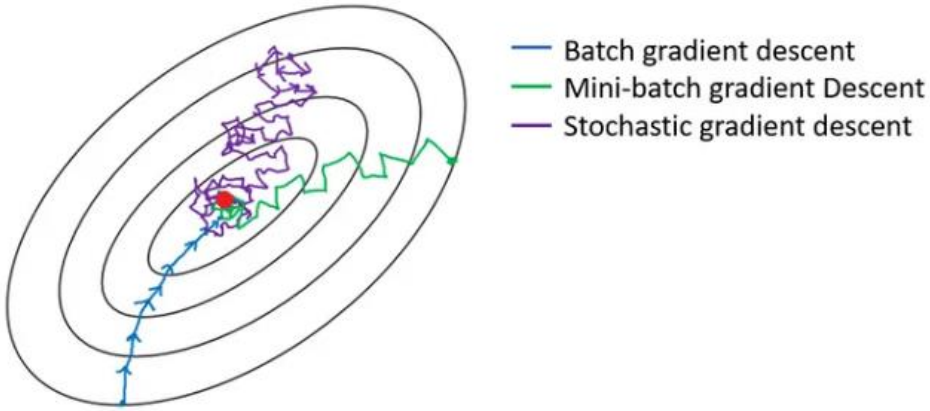

Рисунок 2 – Поведение методов SGD, BatchGD и Mini-batch при сходимости к минимуму [16]

Таким образом, в некоторых задачах с целью оптимизации градиентного метода обучения могут рассматриваться подходы по обработке входных данных, которые могут оказать положительное влияние на скорость обучения нейронной сети.

Также отличительной особенностью модификации стандартного градиентного спуска является использование метода моментов, который добавляет инерцию к изменению параметров, что позволяет избежать «застревания» в локальных минимумах. На каждой новой итерации используется среднее значение прошлых градиентов и добавляет в алгоритм оптимизации эффект импульса, что позволяет скорректировать направление шага относительно исторически доминирующего направления и сгладить траекторию обновления, особенно в задачах с шумными градиентами или узкими долинами в пространстве параметров.

Ниже на рисунке 3 приведен пример сглаживания данных при помощи скользящего среднего. Синяя линия — это одномерное стохастическое случайное блуждание. Гладкие жёлтая и зеленая линии — это усреднение с различным параметром β – коэффициент момента.

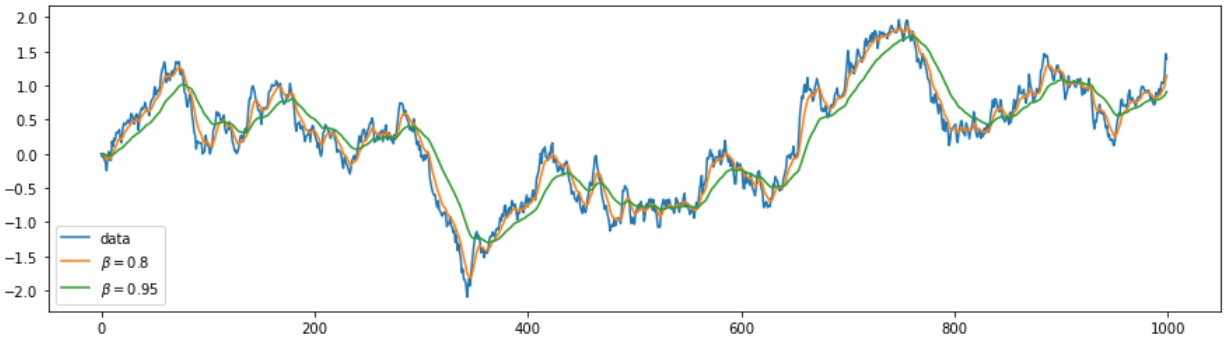

Рисунок 3 – Сглаживание шумных данных с различными значениями коэффициента момента β [14]

Преимуществом является хранение только скользящего среднего значения.

Обновление весов происходит по формуле:

$$\Delta W_t = \eta \cdot \nabla E + \mu \cdot \Delta W_{t-1}$$

где η – коэффициент скорости обучения;

∇E – градиент функции потери;

μ – коэффициент момента;

ΔW_{t-1} – изменение весов на предыдущей итерации.

Однако можно еще улучшить полученную формулу путем наложения ограничений на чрезмерный рост значений весов это помогает бороться с переобучением – ситуацией в машинном обучении, когда модель слишком хорошо подстраивается под обучающие данные, но при этом теряет способность обобщать свои знания на новые данные [6]:

$$\Delta W_t = \eta \cdot (\nabla E + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}$$

где η – коэффициент скорости обучения;

∇E – градиент функции потери;

μ – коэффициент момента;

ΔW_{t-1} – изменение весов на предыдущей итерации;

ρ – коэффициент регуляризации, который предотвращает слишком большие значения весов;

W_{t-1} – значения весов на предыдущей итерации.

При этом коэффициент скорость обучения η будем изменять в зависимости от изменения ошибки:

$$\eta_t = \begin{cases} \alpha \cdot \eta_{t-1} \text{ при } \Delta E > 0 \\ \beta \cdot \eta_{t-1} \text{ иначе} \end{cases}$$

где $\eta_0 = 0.01$ – начальное значение скорости обучения;

$\Delta E = E_t - \gamma \cdot E_{t-1}$ – изменение ошибки;

$\alpha = 0.99,\ \beta = 1.01,\ \gamma = 1.01$ – константы, которые могут изменяться в зависимости от задачи.

Таким образом, при существенном росте ошибки E шаг изменения параметров η, уменьшается, в противном случае – шаг η увеличивается. Это дополнение может увеличить скорость сходимости алгоритма при обучении глубоких нейронных сетей.

2.3. Метод NAG (Nesterov's Accelerated Gradient)

Алгоритм был предложен Нестеровым в 1983 году (NAG) — это модификация алгоритма градиентного спуска с использованием моментов, здесь градиент вычисляется относительно сдвинутых на значение момента весов. Основная идея алгоритма заключается в том, чтобы учитывать будущее направление движения при вычислении текущего обновления, что позволяет добиться более точного приближения к минимуму [6].

$$\Delta W_t = \eta \cdot (\nabla E(W_{t-1} + \mu \cdot \Delta W_{t-1}) + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}$$

где η – коэффициент скорости обучения;

∇E – градиент функции потери;

μ – коэффициент момента;

ΔW_{t-1} – изменение весов на предыдущей итерации;

ρ – коэффициент регуляризации;

W_{t-1} – значения весов на предыдущей итерации;

$W_{t-1} + \mu \cdot \Delta W_{t-1}$ – предсказанная точка для вычисления градиента (шаг предсказания).

Такой подход с предсказанием позволяет избежать лишнего движения в противоположную сторону, если градиент резко изменяется. Таким образом, алгоритм заранее учитывает будущее влияние инерции.

2.4. Метод quickProp

Несмотря на то, что метод обратного распространения ошибки показывает хорошие результаты при обучении нейронной сети, он имеет значительный недостаток в виде плохой масштабируемости в больших сетях, что приводит к долгому обучению модели [5]. По этим причинам в 1988 году было предложено альтернативное локальное и более быстрое правило обновления, в котором функция общих потерь E аппроксимируется квадратичной полиномиальной функцией (используя разложение Тейлора) для каждого веса независимо (предполагая, что каждое обновление имеет ограниченное влияние на соседей). Иногда данный метод могут относить к градиентным методам 2-го порядка. Результирующее правило обновления веса (которое является методом псевдовторого порядка) выглядит следующим образом [6]:

$$\Delta W_{ij}^{t} = \frac{\Delta W_{ij}^{t-1} \cdot \nabla E^{t}}{\nabla E^{t-1} - \nabla E^{t}}$$

$$W_{ij}^{t} = W_{ij}^{t} + \alpha \Delta W_{ij}^{t}$$

где E – функция ошибки;

$\Delta W_t -$ изменение весов на текущей итерации;

$\Delta W_{ij}^{t-1} -$ изменение весов на предыдущей итерации между i нейроном и j;

∇E^t - градиент функции ошибки на текущей итерации;

∇E^{t-1} - градиент функции ошибки на предыдущей итерации.

К недостаткам данного метода можно отнести скачкообразное поведение при малых значениях градиентов, а также сложность подбора начальных параметров.

2.5. Метод rProp

Метод rProp также направлен на ускорение обучения нейронной сети и его особенностью является использование только знаков градиентов для вычисления обновлений. Он хорошо работает во многих ситуациях, поскольку динамически адаптирует размер шага для каждого веса независимо. Ключевую роль играет параметр скорости обучения η, он рассчитывается для каждого веса индивидуально.[6]

$$\eta(t) = \begin{cases} min(\eta_{max}, a \cdot \eta(t-1)), & S > 0 \\ max(\eta_{min}, b \cdot \eta(t-1)), & S < 0 \\ \eta(t-1), & S = 0 \end{cases}$$

где $S = \nabla E(t-1) \cdot \nabla E(t)$ – произведения значений градиента на этом и предыдущем шаге, $\eta_{max} = 50$, $\eta_{min} = 10^{-6}$, $a = 1.2$, $b = 0.5$ – константы.

Значение a – масштабирует размер шага в зависимости от того, следует ли увеличить или уменьшить скорость. Затем размер шага обрезается с помощью min и max чтобы избежать слишком большого или слишком маленького значения.

Изменение параметров выглядит следующим образом [10]:

$$\Delta W_t = \eta \cdot (sign(\nabla E) + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}$$

При этом:

$$sign(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

Основным преимуществом данного алгоритма является разный размер шага для каждого веса. Если один вес уже очень близок к оптимальному значению, а второй вес все еще требует значительного изменения, это не проблема для RProp [3].

2.6. Метод сопряженных градиентов

Особенность этого метода - специальный выбор направления изменения параметров. Оно выбирается таким образом, чтобы было ортогональным к предыдущим направлениям. Полное изменение весов выглядит следующим образом.[8]

$$\Delta W = \eta \cdot (p + \rho \cdot W) + \mu \cdot \Delta W$$

где η – коэффициент скорости обучения;

p – направление изменения параметров;

μ – коэффициент момента;

ΔW – изменение весов на предыдущей итерации;

ρ – коэффициент регуляризации;

W – значения весов на предыдущей итерации;

При этом коэффициент скорости обучения η, выбирается на каждой итерации, путём решения задачи оптимизации.

$$\min_{\eta} E(\Delta W(\eta))$$

Направление изменения параметров выбирается следующим образом.

$$p = \nabla E + \beta \cdot p$$

Начальное направление выбирается как $p_0 = \nabla E$.

Основным и отличительным признаком данного алгоритма является вычисление коэффициента сопряжения β. Его могут вычислять двумя различными способами:

Способ 1 (формула Флетчера-Ривса) [6]:

$$\beta = \frac{g_t^T \cdot g_t}{g_{t-1}^T \cdot g_{t-1}}$$

Способ 2 (формула Полака-Рибьера) [6]:

$$\beta = \frac{g_t^T \cdot (g_t - g_{t-1})}{g_{t-1}^T \cdot g_{t-1}}$$

В каждом из способов $g = \nabla E$ – градиент функции потери на этой и предыдущей итерациях.

Геометрически поведение метода сопряженных градиентов значительно отличается от обычного градиентного спуска. (См. Рис. 4)

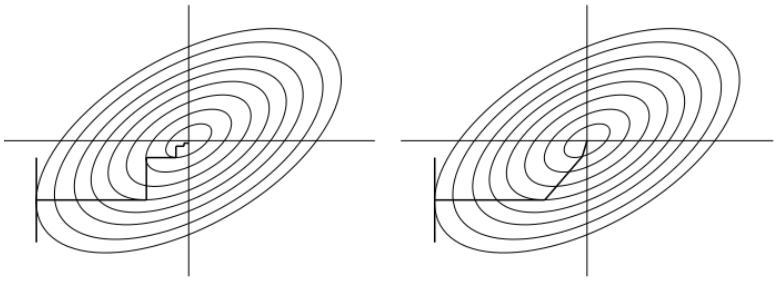

Рисунок 4 – Сходимость методов градиентного спуска (слева) и сопряженных градиентов (справа)[1]

[1] Составлено автором самостоятельно

Значительным недостатком данного метода является накопление погрешности вычислений. Этого можно избежать путем обнуления β и p после прохождения каждого из n циклов, сам же параметр n как правило подбирается в зависимости от рассматриваемой нейронной сети. [3]

2.7. Метод AdaGrad (Adaptive Gradient)

AdaGrad (Adaptive Gradient) – алгоритм, который адаптивно изменяет скорость обучения η для каждого параметра в зависимости от суммарной величины градиентов вычисленных на предыдущих шагах. Он гарантирует, что параметры, получающие мало обновлений, получат более высокие скорости обучения, что ускоряет сходимость. Этот метод подходит для задач, где градиенты имеют разные масштабы, например, в задачах разреженного представления данных [16].

Основная формула выглядит следующим образом [6]:

$$g_t = \frac{\nabla \mathrm{E}_t}{\sqrt{\sum_{i=1}^{t} \nabla \mathrm{E}_i^2}}$$

$$\Delta W_t = \eta \cdot (g_t + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}$$

где η – коэффициент скорости обучения;

∇E – градиент функции потери;

μ – коэффициент момента;

ΔW_{t-1} – изменение весов на предыдущей итерации;

ρ – коэффициент регуляризации;

W_{t-1} – значения весов на предыдущей итерации.

К минусам данного алгоритма можно отнести рост суммы квадратов со временем, что может привести к слишком низкому уровню скорости обучения, а также отсутствию контроля над скоростью обучения отдельных параметров, поскольку он глобально накапливает квадратичные градиенты для всех параметров [11]. Это может быть проблемой в случаях, когда необходимы разные скорости обучения. В качестве решения данных проблем выступают методы, рассмотренные ниже.

2.8. Метод AdaDelta

Метод AdaDelta представляет собой адаптивный вариант алгоритма градиентного спуска, предназначенный для устранения ограничений AdaGrad, связанных с бесконечным уменьшением скорости обучения.

Обновление значений весов происходит по следующим формулам [6]:

$$S_0 = 0$$

$$S_t = \alpha \cdot S_{t-1} + (1 - \alpha) \cdot \nabla E_t^2$$

$$D_0 = 0$$

$$D_t = \beta \cdot D_{t-1} + (1 - \beta) \cdot \Delta W_{t-1}^2$$

$$g_t = \frac{\sqrt{D_t}}{\sqrt{S_t}} \cdot \nabla E_t$$

$$\Delta W_t = \eta \cdot (g_t + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}$$

где η – коэффициент скорости обучения;

g_t – масштабированный градиент для адаптивного обновления весов;

∇E – градиент функции потери;

μ – коэффициент момента;

ΔW_{t-1} – изменение весов на предыдущей итерации;

ρ – коэффициент регуляризации;

W_{t-1} – значения весов на предыдущей итерации;

$\alpha = \beta = 0.9$

Adadelta обновляет размер шага, используя скользящее среднее квадратов градиентов, в отличие от AdaGrad, который корректирует размер шага на основе суммы квадратов градиентов. Это позволяет Adadelta иметь более стабильную и гибкую скорость обучения, что в некоторых обстоятельствах может привести к большей производительности и более быстрой сходимости [9].

2.9. Метод RMSprop

RMSprop (Root Mean Square Propagation) — это модификация для метода AdaGrad, который регулирует скорость обучения для каждого параметра, основываясь на среднем значении квадратов недавних градиентов. Он часто используется в задачах, где скорость обучения требуется стабилизировать [10].

Сглаживание квадратов градиентов:

$$S_t = \alpha \cdot S_{t-1} + (1 - \alpha) \cdot (\nabla E_t)^2$$

S_t - накопленная средняя величина квадрата градиента;

α - коэффициент сглаживания, например, α=0.9;

∇E_t - текущий градиент для параметров.

Обновление весов:

$$\Delta Wt = -\frac{\eta}{\sqrt{S_t + \epsilon}} \cdot \nabla Et$$

$$W_t = W_{t-1} + \Delta W_t$$

η - скорость обучения;

$\sqrt{S_t + \epsilon}$ - нормализация градиента для предотвращения больших шагов;

∇E_t - текущий градиент.

RMSprop особенно хорошо работает для задач глубокого обучения, включая рекуррентные нейронные сети (RNN), где градиенты могут быстро затухать или взрываться [5].

2.10. Метод Adam

Adam — один из самых эффективных алгоритмов оптимизации в обучении нейронных сетей. Он сочетает в себе идеи RMSProp и оптимизатора импульса. Вместо того, чтобы адаптировать скорость обучения параметров на основе среднего первого момента (среднего значения), как в RMSProp, Adam также использует среднее значение вторых моментов градиентов [7]. В частности, алгоритм вычисляет экспоненциальное скользящее среднее градиента и квадратичный градиент, а параметры α и β управляют скоростью затухания этих скользящих средних [6].

$$S_0 = 0$$

$$S_t = \alpha \cdot S_{t-1} + (1 - \alpha) \cdot \nabla E_t^2$$

$$D_0 = 0$$

$$D_t = \beta \cdot D_{t-1} + (1 - \beta) \cdot \nabla E_t$$

$$\Delta W_t = \eta \cdot (g_t + \rho \cdot W_{t-1}) + \mu \cdot \Delta W_{t-1}$$

$$g_t = \frac{D_t}{1 - \beta} \cdot \sqrt{\frac{1 - \alpha}{S_t}}$$

где η – коэффициент скорости обучения;

∇E – градиент функции потери;

μ – коэффициент момента;

ΔW_{t-1} – изменение весов на предыдущей итерации;

ρ – коэффициент регуляризации;

W_{t-1} – значения весов на предыдущей итерации.

$\alpha = 0.9999, \beta = 0.9$

Adam лучше других алгоритмов адаптивной скорости обучения из-за более быстрой сходимости и надежности в решении проблем. Сравнение траектории сходимости наиболее популярных методов представлено на рисунках 5, 6.

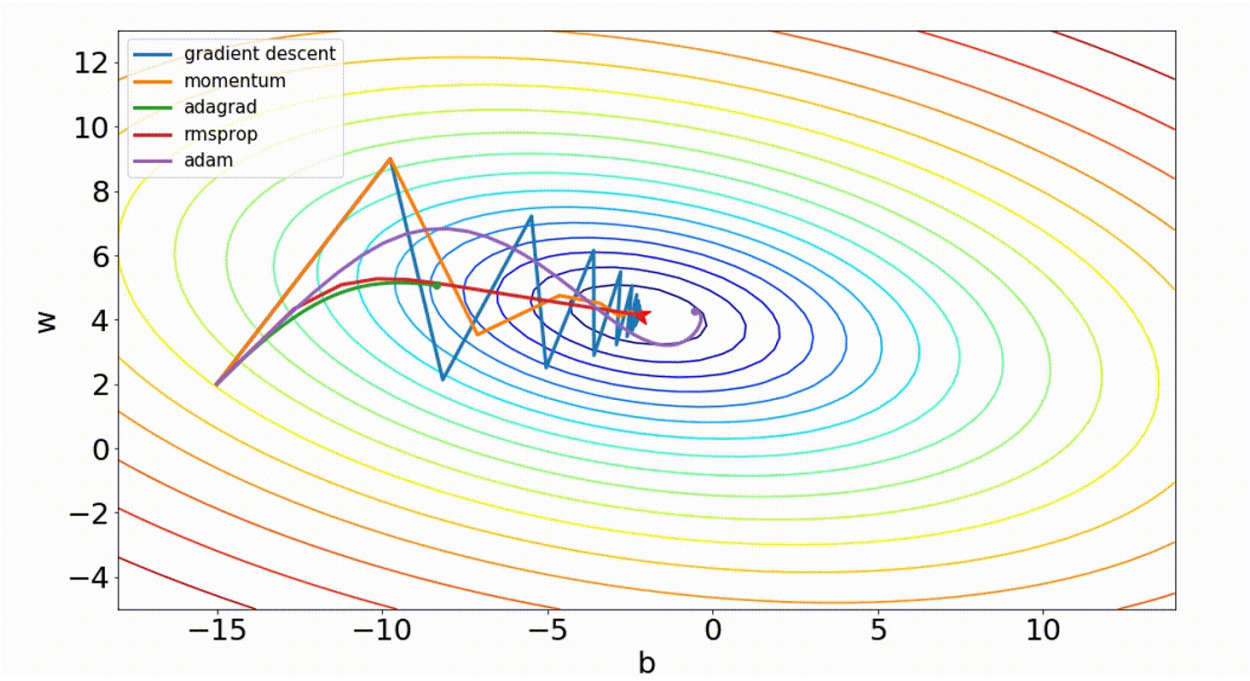

Рисунок 5 – Сравнение сходимости Adam и других методов к минимуму [10]

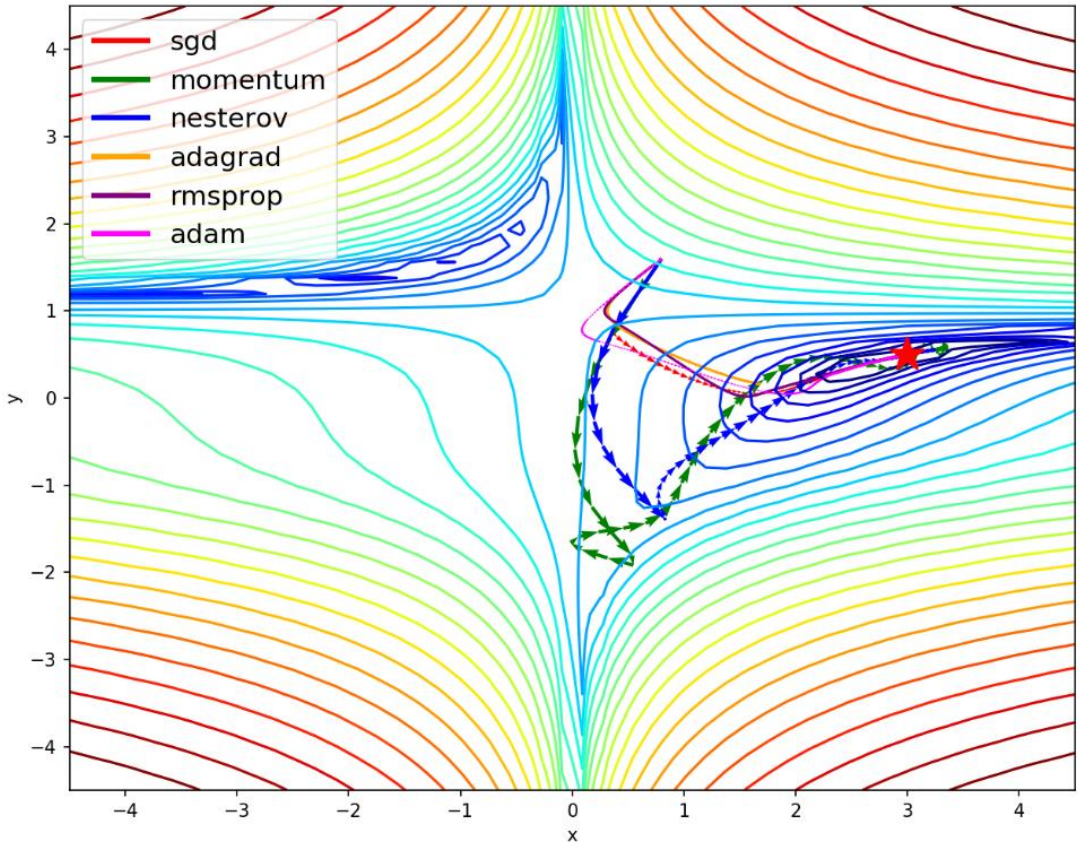

Рисунок 6 (Продолжение) – Сравнение сходимости Adam и других методов к минимуму [10]

Визуально характер поведения метода Adam отличается от AdaGrad, хотя и является ее модифицированной версией. Также нельзя не заметить значительного прироста скорости схождения у Adam. Таким образом, данный метод действительно демонстрирует хорошие результаты при сходимости [2].

3. Практическая часть

В качестве практической части была реализована программа на языке Python, которая применяет каждый из рассмотренных градиентных методов к исходной нейронной сети и строит график изменении ошибки. История ошибок для каждого метода записывается в отдельный файл, что позволяет проанализировать их в дальнейшем. В качестве функции активации использовалась сигмоида.

Входные данные представлены в виде сети (См. Рис. 7) и обучающей выборкой (См. Рис. 8).

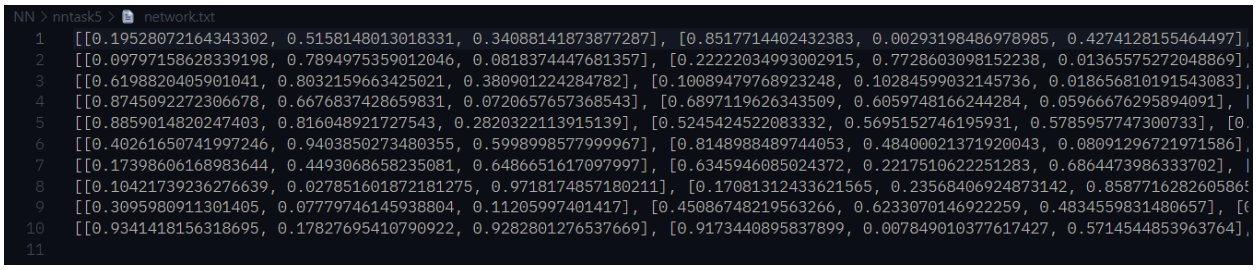

Рисунок 7 – Сеть на входе[2]

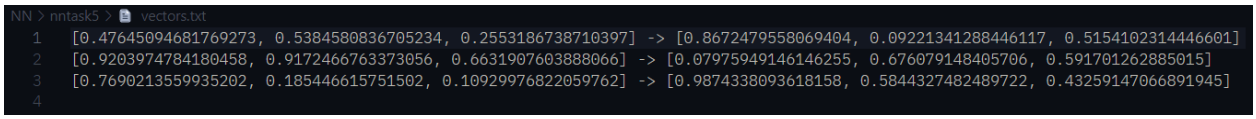

Рисунок 8 – Обучающая выборка[3]

Запустим программу с количеством итераций 1000 и коэффициентом 0.2. (См.Рис 9)

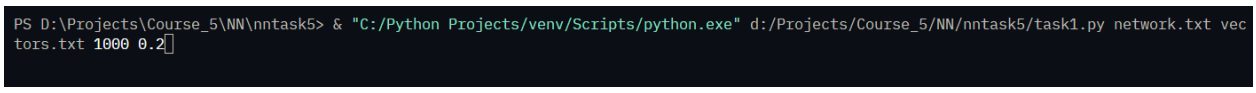

Рисунок 9 – Запуск программы[4]

[2] Составлено автором самостоятельно
[3] Составлено автором самостоятельно
[4] Составлено автором самостоятельно

В результате получили следующий график. (См. Рис. 10)

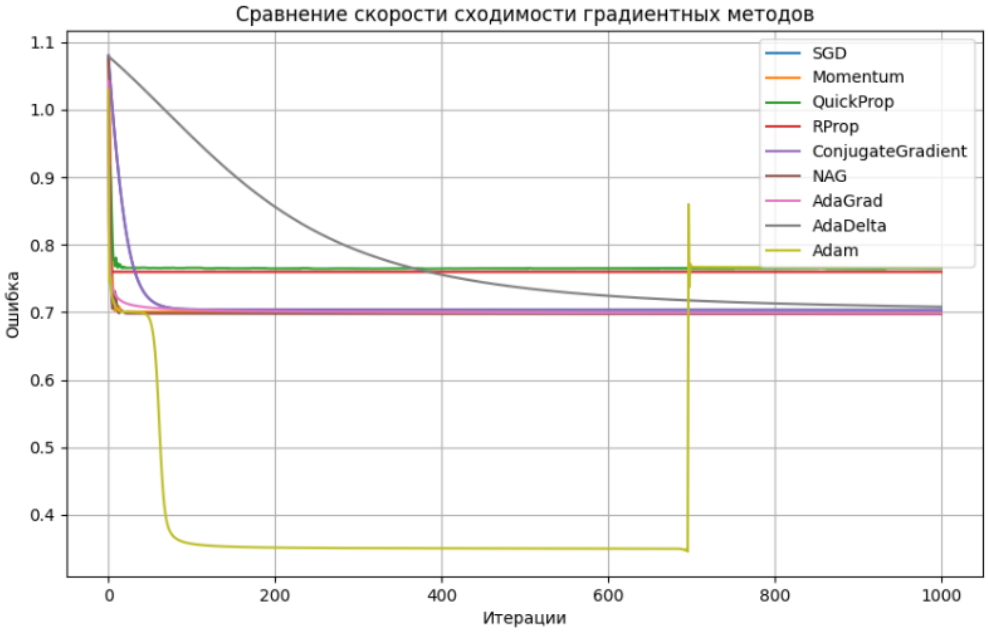

Рисунок 10 – Результат тестирования в графическом представлении[5]

При увеличении можно увидеть характер поведения более детально. (См. Рис. 11)

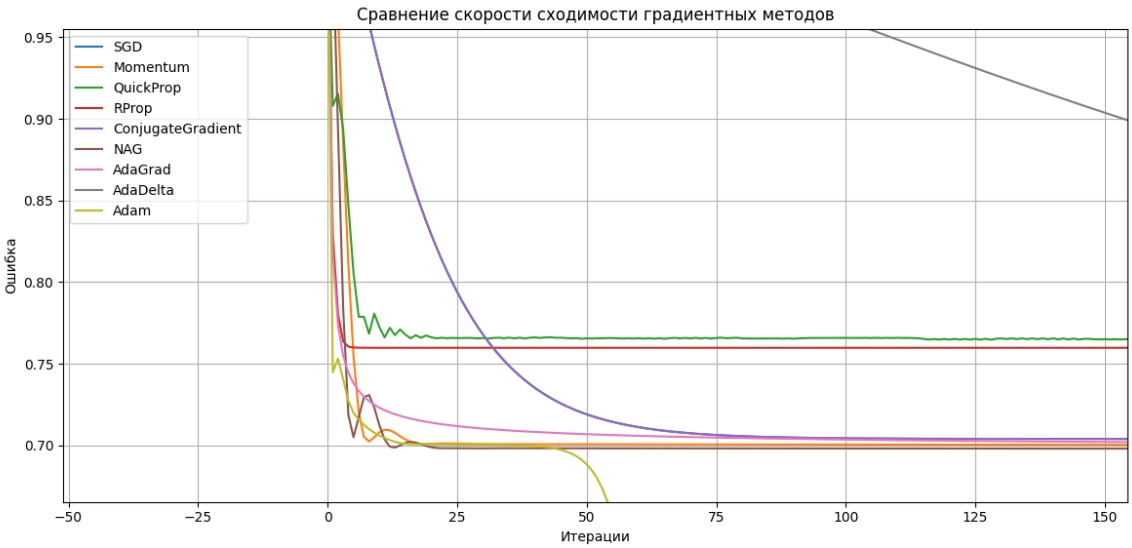

Рисунок 11 – Постепенное приближение методов первого порядка к минимуму[6]

Поведение методов в целом схожи между собой, однако наиболее интересный характер поведения дают Adam и AdaDelta. AdaDelta сходится

[5] Составлено автором самостоятельно
[6] Составлено автором самостоятельно

более устойчиво, а Adam на ранних и более поздних стадиях нестабильна, однако все равно показывают в результате приближенные значения к другим методам. Это может быть связанно с настройкой параметров и адаптацией параметра η. Также нельзя не отметить, что некоторые методы сходятся к другим значениям. Разница возникает из-за их стратегий обновления параметров, а также из-за влияния особенностей целевой функции (несколько минимумов) и структуры данных.

В процессе анализа полученных графических данных оказалось сложно оценить правильность сходимости градиентных методов во входной нейронной сети, поэтому было принято решение протестировать методы для более простых функций, например $4x^2 + 12x + 1$. Построим график и увидим, где у нее находится минимум. (См. Рис. 12)

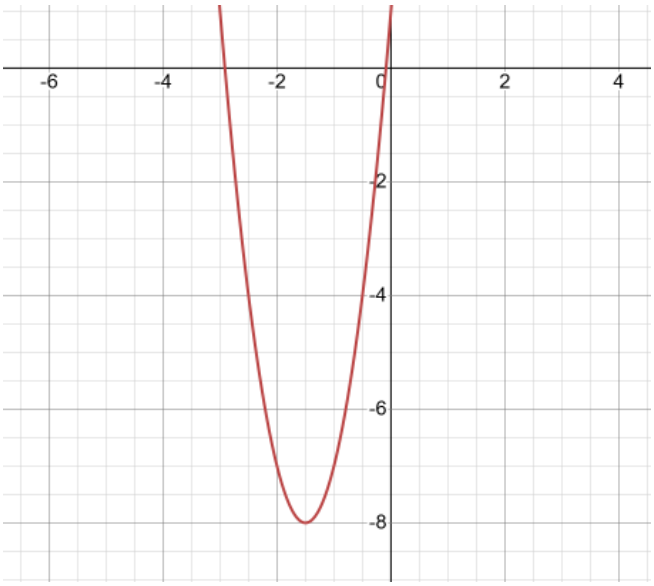

Рисунок 12 – График функции $4x^2 + 12x + 1$ с точкой в минимуме -8 [7]

В результате запуска программы получили следующий график (См. Рис. 13):

[7] Составлено автором самостоятельно

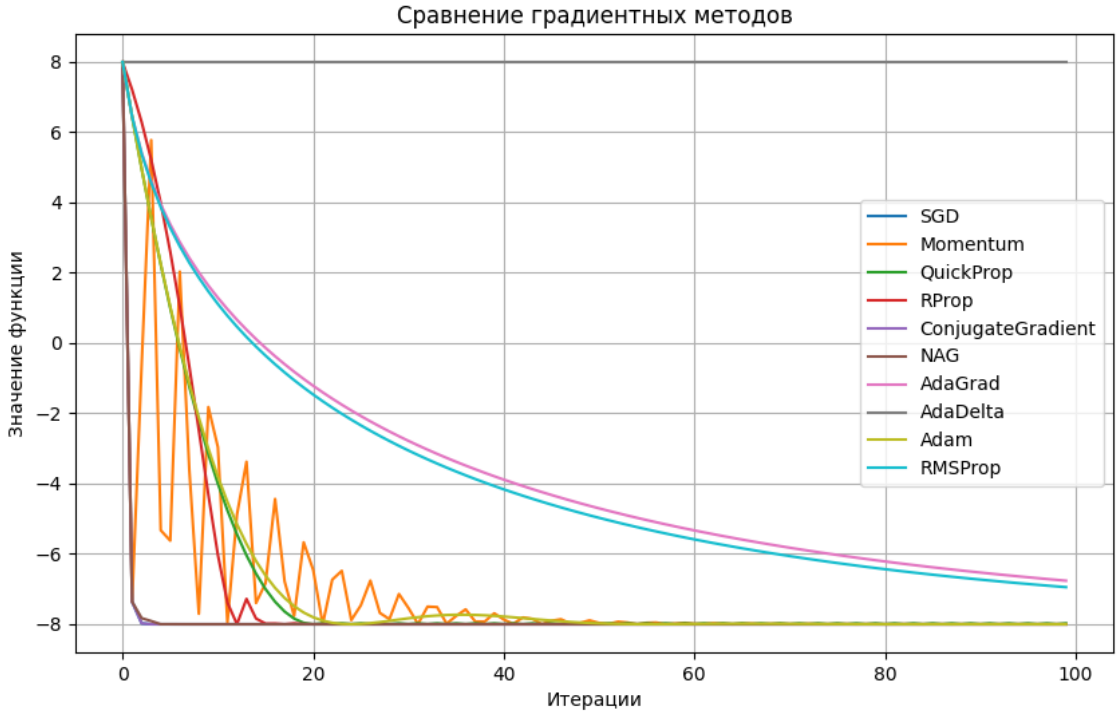

Рисунок 13 – Сходимость градиентных методов к минимуму функции за 100 итераций[8]

Рассмотрим функции при увеличении (См. Рис. 14)

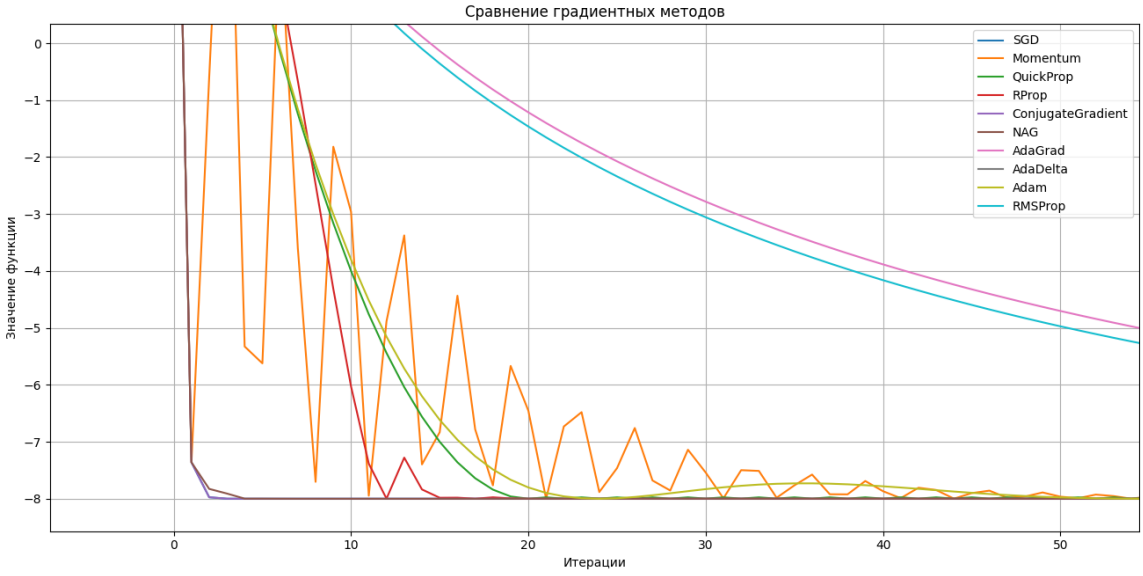

Рисунок 14 – Сходимость градиентных методов к минимуму функции за 100 (увеличенное изображение)[9]

[8] Составлено автором самостоятельно
[9] Составлено автором самостоятельно

Можно заметить, что методы сошлись достаточно быстро, однако метод AdaDelta даже не приблизился к ответу. Сойтись данному методу к минимуму удастся только, через 80000 итераций, что очень много по сравнению с другими. (См. Рис. 15)

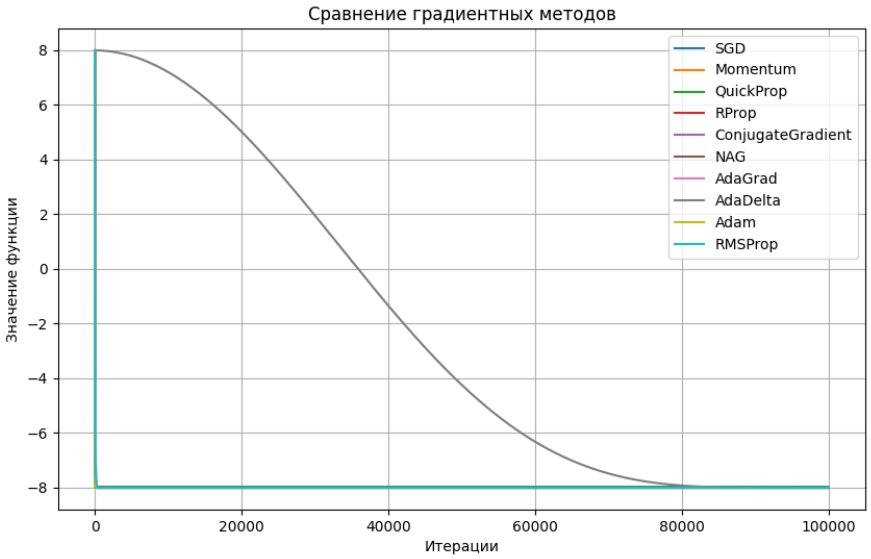

Рисунок 15 – Схождение к минимуму функции методом AdaDelta через 80000 итераций[10]

Ввиду того, что реализованные алгоритмы в некоторых местах были упрощены, считаю, что программная реализация отработала корректно, и поставленная задача выполнена.

[10] Составлено автором самостоятельно

4. Преимущества и недостатки градиентных методов первого порядка

Исходя из изученных теоретических материалов и практического наблюдения можно сформулировать основные положительные и отрицательные стороны градиентных методов первого порядка изучения нейронных сетей.

К преимуществам можно отнести:

1. Относительная простота реализации;
2. Адаптация под конкретные задачи, а также возможность применения их к любым дифференцируемым функциям;
3. Масштабируемость на большое количество данных;
4. При правильных параметрах могут достигать оптимального решения.

К недостаткам можно отнести:

1. Сложность выбора и чувствительность параметров, которые могут привести к значительному снижению скорости;
2. Возможны ситуации при которых происходит застревание в локальных минимумах или седловых точках (местах, в которых градиент мал или равен нулю, но это не является минимумом);
3. Для более качественного результата может потребоваться большое количество итераций;
4. Проблема переобучения – если модель слишком мощная (много параметров), градиентные методы могут идеально подстроить веса под обучающую выборку, но потерять обобщающую способность (неспособность обрабатывать другие примеры);
5. Отсутствие гарантии нахождения глобального минимума – Большинство задач обучения нейронных сетей связаны с

минимизацией нелинейных и негладких функций. Градиентные методы не гарантируют нахождение глобального минимума, особенно для функций с множеством локальных экстремумов.

ЗАКЛЮЧЕНИЕ

В результате проведённого исследования градиентных методов обучения нейронных сетей первого порядка были рассмотрены их теоретические основы, ключевые алгоритмы и практическое применение. Градиентные методы, являясь основным инструментом оптимизации, продемонстрировали свою эффективность в настройке параметров нейронных сетей, обеспечивая минимизацию функции потерь.

Практическая часть исследования подтвердила различия в поведении и эффективности различных методов, таких как AdaDelta, Adam, RProp, QuickProp и других. Выявлено, что выбор подходящего метода зависит от структуры данных, архитектуры нейронной сети и требований к сходимости. Сравнение различных подходов позволило сделать выводы о ключевых преимуществах и недостатках методов. В частности, Adam сочетает в себе скорость и адаптивность, однако требует более сложной настройки гиперпараметров.

Таким образом, выбор метода должен учитывать особенности задачи, размер данных и требования к производительности. Градиентные методы остаются фундаментальным инструментом для обучения нейронных сетей, и их дальнейшее совершенствование является перспективным направлением исследований в области машинного обучения.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Градиентный спуск: всё, что нужно знать [Электронный ресурс]: - URL: https://neurohive.io/ru/osnovy-data-science/gradient-descent/ (дата обращения 15.12.2024) - Рус.яз.
2. Каширина, И.Л. Исследование и сравнительный анализ методов оптимизации, используемых при обучении нейронных сетей/ И.Л. Каширина, М.В. Демченко// Вестник Воронежского государственного университета. Серия: Системный анализ и информационные технологии. - 2018. - № 4. - С. 123-132.
3. Крючин, О.В. Обучение искусственной нейронной сети при помощи распараллеленных градиентных алгоритмов rprop, quickprop и метода наискорейшего спуска // Актуальные инновационные исследования: наука и практика. - 2010. - № 2. - С. 8.
4. Математика за оптимизаторами нейронных сетей [Электронный ресурс]: - URL: https://habr.com/ru/companies/otus/articles/561080/ (дата обращения 17.12.2024) - Рус.яз.
5. Методы оптимизации нейронных сетей [Электронный ресурс]: - URL: https://habr.com/ru/articles/318970/ (дата обращения 9.12.2024) – Рус.яз.
6. О методах обучения многослойных нейронных сетей прямого распространения [Электронный ресурс]: - URL: http://mechanoid.su/neural-net-backprop2.html (дата обращения 9.12.2024) - Рус.яз.
7. Обзор градиентных методов в задачах математической оптимизации [Электронный ресурс]: - URL: https://habr.com/ru/articles/413853/ (дата обращения 6.05.2024) – Рус.яз.
8. Оптимизаторы градиентных алгоритмов: RMSProp, AdaDelta, Adam, Nadam [Электронный ресурс]: - URL:https://proproprogs.ru/ml/ml-optimizatory-gradientnyh-algoritmov-rmsprop-adadelta-adam-nadam (дата обращения 12.05.2024) – Рус.яз.

9. Перков, А.С. Исследование градиентных методов обучения многослойных нейронных сетей в задачах классификации/ Т.Р.Жангиров // Наука настоящего и будущего. - 2018. - Т. 1. - С. 200-203.
10. Реализуем и сравниваем оптимизаторы моделей в глубоком обучении [Электронный ресурс]: - URL: https://habr.com/ru/companies/skillfactory/articles/525214/ (дата обращения 18.05.2024) – Рус.яз.
11. Цыдыпова, С.Ю. Гиперпараметры градиентных методов обучения нейронных сетей/ А.С.Цыбиков // Геометрия многообразий и ее приложения - Улан-Удэ. - 2020. - С. 216-222.
12. A Simple Guide to Gradient Descent Algorithm [Электронный ресурс]: - URL:https://medium.com/@datasciencewizards/a-simple-guide-to-gradient-descent-algorithm-60cbb66a0df9 (дата обращения 2.12.2024) - Англ.яз.
13. Gradient Descent Algorithm in Machine Learning [Электронный ресурс]: - URL:https://www.geeksforgeeks.org/gradient-descent-algorithm-and-its-variants/ (дата обращения 6.05.2024) - Англ.яз.
14. ML: Градиентный метод [Электронный ресурс]: - URL:https://qudata.com/ml/ru/ML_Grad_Method.html (дата обращения 7.12.2024) - Рус.яз.
15. RProp [Электронный ресурс]: - URL: https://florian.github.io/rprop/ (дата обращения 13.12.2024) - Англ.яз.
16. Understanding the AdaGrad Optimization Algorithm: An Adaptive Learning Rate Approach [Электронный ресурс]: - URL: https://medium.com/@brijesh_soni/understanding-the-adagrad-optimization-algorithm-an-adaptive-learning-rate-approach-9dfaae2077bb (дата обращения 9.12.2024) – Англ.яз.

ПРИЛОЖЕНИЕ А

Тестирование сходимости различных градиентных методов на нейронной сети

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

# Функция активации (сигмоида) и её производная
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Загрузка нейронной сети из файла
def load_network(file_path):
    with open(file_path, "r") as file:
        lines = file.readlines()
        layers = [eval(line.strip()) for line in lines]
    return [np.array(layer) for layer in layers]

# Загрузка обучающей выборки
def load_training_data(file_path):
    with open(file_path, "r") as file:
        lines = file.readlines()
        data = []
        for line in lines:
            x, y = line.strip().split("->")
            x = eval(x.strip())
            y = eval(y.strip())
            data.append((np.array(x), np.array(y)))
    return data

# Метод обратного распространения ошибки с различными градиентными
методами
def train_network(network, training_data, iterations, learning_rate,
method, output_file="training_history.txt"):
    history = []
```

```
    with open(output_file, "w") as file:
        momentum = [np.zeros_like(layer) for layer in network]
        rprop_grad_prev = [np.zeros_like(layer) for layer in network]
        rprop_step = [np.full_like(layer, 0.1) for layer in network]
        adagrad_cache = [np.zeros_like(layer) for layer in network]
        adadelta_cache = [np.zeros_like(layer) for layer in network]
        adadelta_momentum = [np.zeros_like(layer) for layer in
network]
        adam_m = [np.zeros_like(layer) for layer in network]
        adam_v = [np.zeros_like(layer) for layer in network]
        beta1, beta2, epsilon = 0.9, 0.999, 1e-8

        for iteration in range(1, iterations + 1):
            total_error = 0

            for x, y in training_data:
                # Прямое распространение
                activations = [x]
                for layer in network:
                    activations.append(sigmoid(np.dot(activations[-1],
layer)))

                # Ошибка выходного слоя
                error = y - activations[-1]
                total_error += np.sum(error ** 2)

                # Обратное распространение
                deltas = [error * sigmoid_derivative(activations[-1])]
                for i in range(len(network) - 1, 0, -1):
                    delta = deltas[-1].dot(network[i].T) *
sigmoid_derivative(activations[i])
                    deltas.append(delta)
                deltas.reverse()

                # Обновление весов
                for i in range(len(network)):
                    gradient = activations[i][:, np.newaxis] @
deltas[i][np.newaxis, :]
```

```
                    if method == "SGD":
                        network[i] += learning_rate * gradient
                    elif method == "Momentum":
                        momentum[i] = 0.9 * momentum[i] +
learning_rate * gradient
                        network[i] += momentum[i]
                    elif method == "QuickProp":
                        network[i] += gradient / (np.abs(gradient) +
epsilon) * learning_rate
                    elif method == "RProp":
                        sign_change = np.sign(rprop_grad_prev[i] *
gradient)
                        rprop_step[i][sign_change > 0] *= 1.2
                        rprop_step[i][sign_change < 0] *= 0.5
                        rprop_grad_prev[i] = gradient
                        network[i] += rprop_step[i] *
np.sign(gradient)
                    elif method == "ConjugateGradient":
                        # Упрощённый placeholder, полноценный метод
требует больше логики
                        network[i] += learning_rate * gradient
                    elif method == "NAG":
                        lookahead = network[i] + 0.9 * momentum[i]
                        momentum[i] = 0.9 * momentum[i] +
learning_rate * gradient
                        network[i] = lookahead + momentum[i]
                    elif method == "AdaGrad":
                        adagrad_cache[i] += gradient ** 2
                        network[i] += learning_rate * gradient /
(np.sqrt(adagrad_cache[i]) + epsilon)
                    elif method == "AdaDelta":
                        adadelta_cache[i] = 0.9 * adadelta_cache[i] +
0.1 * gradient ** 2
                        update = gradient *
np.sqrt(adadelta_momentum[i] + epsilon) / np.sqrt(adadelta_cache[i] +
epsilon)
                        adadelta_momentum[i] = 0.9 *
adadelta_momentum[i] + 0.1 * update ** 2
```

```
                        network[i] += update
                    elif method == "Adam":
                        adam_m[i] = beta1 * adam_m[i] + (1 - beta1) *
gradient
                        adam_v[i] = beta2 * adam_v[i] + (1 - beta2) *
(gradient ** 2)
                        m_hat = adam_m[i] / (1 - beta1 ** iteration)
                        v_hat = adam_v[i] / (1 - beta2 ** iteration)
                        network[i] += learning_rate * m_hat /
(np.sqrt(v_hat) + epsilon)

            # Сохранение ошибки для текущей итерации
            history.append(total_error)
            file.write(f"Iteration {iteration}: Error =
{total_error}\n")

    return history

# Построение графика
def plot_training_history(histories, methods):
    plt.figure(figsize=(10, 6))
    for history, method in zip(histories, methods):
        plt.plot(history, label=method)
    plt.xlabel("Итерации")
    plt.ylabel("Ошибка")
    plt.title("Сравнение скорости сходимости градиентных методов")
    plt.legend()
    plt.grid()
    plt.savefig("training_comparison.png")
    plt.show()

# Основная программа
if __name__ == "__main__":
    import sys

    if len(sys.argv) < 5:
        print("Использование: python nntask5.py <network_file>
<training_file> <iterations> <learning_rate>")
        sys.exit(1)
```

```
    network_file = sys.argv[1]
    training_file = sys.argv[2]
    iterations = int(sys.argv[3])
    learning_rate = float(sys.argv[4])

    network = load_network(network_file)
    training_data = load_training_data(training_file)

    methods = [
        "SGD", "Momentum", "QuickProp", "RProp", "ConjugateGradient",
        "NAG", "AdaGrad", "AdaDelta", "Adam"
    ]
    histories = []

    for method in methods:
        # Копируем сеть для каждого метода
        network_copy = [np.copy(layer) for layer in network]

        try:
            output_file = f"training_history_{method}.txt"
            history = train_network(network_copy, training_data,
iterations, learning_rate, method, output_file=output_file)
            histories.append(history)
        except Exception as err:
            with open("error.txt", "w", encoding="UTF-8") as f:
                f.write(f"Ошибка для метода {method}: {str(err)}\n")
            print(f"Ошибка для метода {method}! Подробности сохранены
в 'error.txt'.")

    # Построение графика
    plot_training_history(histories, methods)
```

ПРИЛОЖЕНИЕ В

Сходимость градиентных методов к минимуму заданной функции

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

def optimize(method, x_init, gradient_fn, target_fn, learning_rate=0.1,
iterations=1000):
    x = x_init
    history = []

    momentum = 0
    grad_squared = 0
    rprop_grad_prev = 0
    rprop_step = 0.1
    v = 0
    m = 0
    rho = 0.9
    beta1, beta2 = 0.9, 0.999
    epsilon = 1e-8

    for t in range(1, iterations + 1):
        grad = gradient_fn(x)
        history.append(target_fn(x))

        if method == "SGD":
            x -= learning_rate * grad

        elif method == "Momentum":
            momentum = rho * momentum - learning_rate * grad
            x += momentum

        elif method == "QuickProp":
            if grad != 0:
                x -= grad / (abs(grad) + epsilon) * learning_rate

        elif method == "RProp":
            sign_change = (rprop_grad_prev * grad) > 0
```

```
            if sign_change:
                rprop_step *= 1.2
            else:
                rprop_step *= 0.5
            rprop_grad_prev = grad
            x -= rprop_step * (1 if grad > 0 else -1)

        elif method == "ConjugateGradient":
            beta = np.dot(grad, grad) / np.dot(prev_grad, prev_grad)
            p = grad + beta * p
            x -= learning_rate * (p + rho * x) + 0.01 * x
            prev_grad = grad

        elif method == "NAG":
            lookahead = x + rho * momentum
            momentum = rho * momentum - learning_rate *
gradient_fn(lookahead)
            x += momentum

        elif method == "AdaGrad":
            grad_squared += grad**2
            adjusted_lr = learning_rate / ((grad_squared**0.5) +
epsilon)
            x -= adjusted_lr * grad

        elif method == "AdaDelta":
            grad_squared = rho * grad_squared + (1 - rho) * grad**2
            delta_x = -(v**0.5 + epsilon) / ((grad_squared**0.5) +
epsilon) * grad
            v = rho * v + (1 - rho) * delta_x**2
            x += delta_x

        elif method == "RMSProp":
            grad_squared += grad**2
            grad_squared_avg = rho * grad_squared + (1 - rho) * grad**2
            adjusted_lr = learning_rate / (np.sqrt(grad_squared_avg) +
epsilon)
            x -= adjusted_lr * grad
```

```
        elif method == "Adam":
            m = beta1 * m + (1 - beta1) * grad
            v = beta2 * v + (1 - beta2) * grad**2
            m_hat = m / (1 - beta1**t)
            v_hat = v / (1 - beta2**t)
            x -= learning_rate * m_hat / ((v_hat**0.5) + epsilon)

        else:
            raise ValueError(f"Unknown method: {method}")

    return history

def plot_results(histories, methods):
    plt.figure(figsize=(10, 6))
    for history, method in zip(histories, methods):
        plt.plot(history, label=method)
    plt.xlabel("Итерации")
    plt.ylabel("Значение функции")
    plt.title("Сравнение градиентных методов")
    plt.legend()
    plt.grid()
    plt.show()

if __name__ == "__main__":

    def target_function(x):
        return 4 * x**2 + 12 * x + 1

    def gradient_function(x):
        return 8 * x + 12

    x_init = 0.5 # Начальная точка
    methods = ["SGD", "Momentum", "QuickProp", "RProp",
"ConjugateGradient", "NAG", "AdaGrad", "AdaDelta", "Adam", "RMSProp"]
    learning_rate = 0.1
    iterations = 100

    # Сравнение методов
    histories = []
```

```
    for method in methods:
        try:
            history = optimize(method, x_init, gradient_function,
target_function, learning_rate, iterations)
            histories.append(history)
        except Exception as e:
            print(f"Ошибка в методе {method}: {e}")

    # Построение графиков
    plot_results(histories, methods)
```