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

Градиентные методы обучения нейронных сетей первого порядка

РЕФЕРАТ

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 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 течении 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:

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

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 $dx_1, dx_2, \ldots, dx_n$ дает полный дифференциал этой функции:

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

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 на уменьшение функции.

Обозначим основные элементы нейронной сети для дальнейшего исследования.

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.

Среднеквадратическая ошибка:

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

где y_j – выход сети номер j, c_j – это 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 — это 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 и 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.

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

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

Базовая версия метода градиентного спуска имеет значительный недостаток, а именно застревание в локальных минимумах функции. Для повышения эффективности и устранения недостатков чистого градиентного спуска применяется стратегия mini-batch. Вместо обновления параметров по всему набору данных, вычисления градиентов проводятся на небольших подмножествах (batch) данных. Это снижает вычислительные затраты и уменьшает шум в оценке градиента, обеспечивая баланс между точностью и скоростью. [2]

Также отличительной особенностью модификации является использования метода моментов, который добавляет инерцию к изменению параметров, что позволяет избежать "застревания" в локальных минимумах.

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:

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

где η – коэффициент скорости обучения

∇E – градиент функции потери

μ – коэффициент момента

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

ρ – коэффициент регуляризации

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.

2.3. Метод quickProp

От базового метода он отличается тем, что параметр момента μ и коэффициент скорости обучения η задаются индивидуально для каждого параметра. Изменение параметров описывается следующим соотношением.[1]

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

ρ – коэффициент регуляризации

Параметр скорости обучения вычисляется по следующим соотношениям:

$$\eta = \begin{cases} \eta_0, & \text{при } (\Delta W = 0) \vee (-\Delta W \cdot S > 0) \\ 0, & \text{иначе} \end{cases}$$

где $\eta_0 \in (0.01, 0.6)$ – константа, $S = \nabla E + \rho W$

Параметр момента высчитывается следующим образом:

$$\mu = \begin{cases} \mu_{max}, & \text{при } (\beta > \mu max) \vee (\gamma < 0) \\ \beta, & \text{иначе} \end{cases}$$

Где $\mu_{max} = 1.75$ – константа, $S = \nabla E + \rho W,\ \beta = \frac{S(t)}{S(t-1) - S(t)}, \gamma = S(-\Delta W)\beta$

2.4. Метод rProp

В случае rProp моменты и регуляризация не используются, применяется простая стратегия full-batch. 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

Изменение параметров выглядит следующим образом:

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

При этом:

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

2.5. Метод сопряженных градиентов

Особенность этого метода - специальный выбор направление 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):

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

Способ 2 (формула Полака-Рибьера):

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

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

Значительным недостатком данного метода является накопление погрешности вычислений. Этого можно избегать 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]

2.6. Метод NAG (Nesterov's Accelerated Gradient)

NAG это еще одна модификация стандартного алгоритма градиентного спуска, здесь градиент вычисляется относительно сдвинутых на значение момента весов.

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

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.

Алгоритм был предложен Нестеровым в 1983 году и активно применяется (наряду с другими оптимизаторами) при обучении алгоритмов, в частности, нейронных сетей.[8]

2.7. Метод AdaGrad (Adaptive Gradient)

Во всех рассмотренных выше алгоритмах вычисления производились без учета предыдущих значений градиента, поэтому чтобы учитывать историю значений можно воспользоваться следующей формулой [4]:

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

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

где η – коэффициент скорости обучения,

∇E – градиент функции потери,

μ – коэффициент момента,

$\mathit{\Delta} W_{t-1}$ – изменение весов на предыдущей итерации

ρ – коэффициент регуляризации,

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

2.8. Метод AdaDelta

Метод AdaDelta учитывает историю значений градиента и историю изменения весов следующим образом.

$$S_t = \alpha \cdot S_{t-1} + (1-\alpha) \cdot \nabla E_t^2 \ ; \ S_0 = 0$$

$$D_t = \beta \cdot D_{t-1} + (1-\beta) \cdot \mathit{\Delta} W_{t-1}^2 \ ; \ D0 = 0$$

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

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

где η – коэффициент скорости обучения,

∇E – градиент функции потери,

μ – коэффициент момента,

$\mathit{\Delta} W_{t-1}$ – изменение весов на предыдущей итерации

ρ – коэффициент регуляризации,

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

$\alpha = \beta = 0.9$

2.9. Метод Adam

Метод Adam преобразует градиент следующим образом:

$$S_t = \alpha \cdot S_{t-1} + (1-\alpha) \cdot \nabla E_t^2\ ;\ S_0 = 0$$

$$D_t = \beta \cdot D_{t-1} + (1-\beta) \cdot \nabla E_t\ ;\ D_0 = 0$$

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

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

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.

$\alpha = 0.9999, \beta = 0.9$

3. Тестирование градиентных методов обучения нейронных сетей

Специалисты в области изучения нейронных сетей провели достаточное количество экспериментов, что позволило наглядно посмотреть на поведение методов градиентного обучения и выделить особенности каждого из них.[7]

В качестве тестового набора специалистами был рассмотрен набор из различных точек использовать набор точек. Каждый набор точек перед запуском процедуры обучения случайным образом делиться на три части: учебный, контрольный и тестовый. Ввиду схожести поведения некоторых методов, была рассмотрена лишь часть. [9]

Набор точек представлен на рисунках 1, 2.

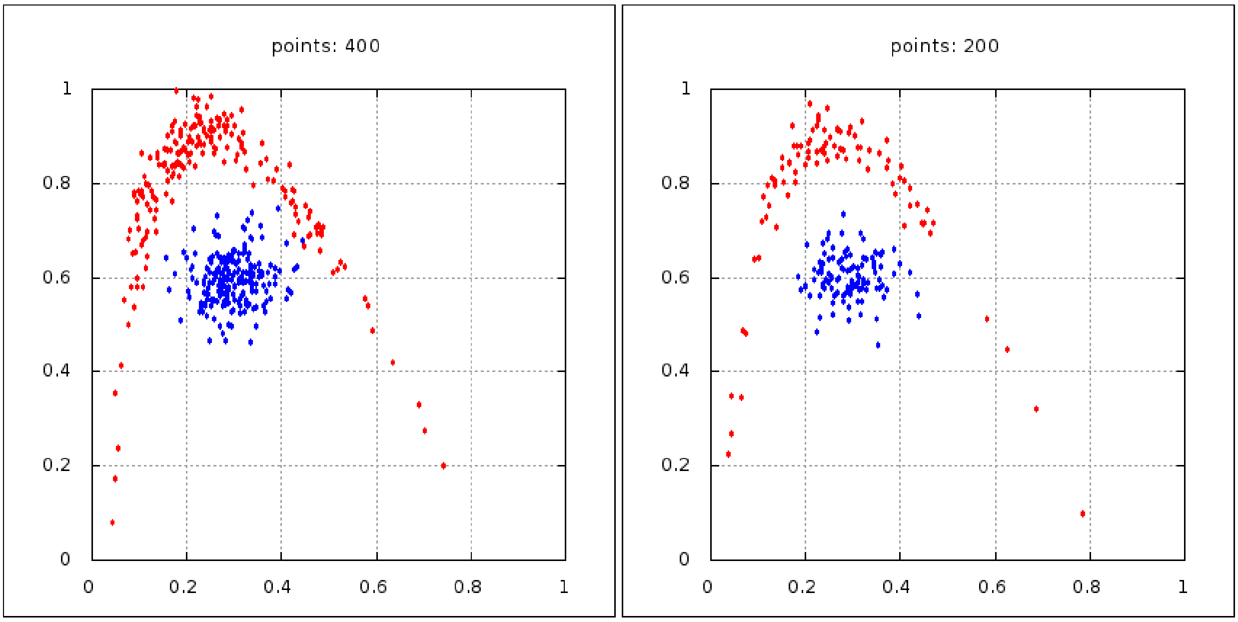


Рисунок 1 – Учебный набор данных (слева), контрольный набор данных (справа)

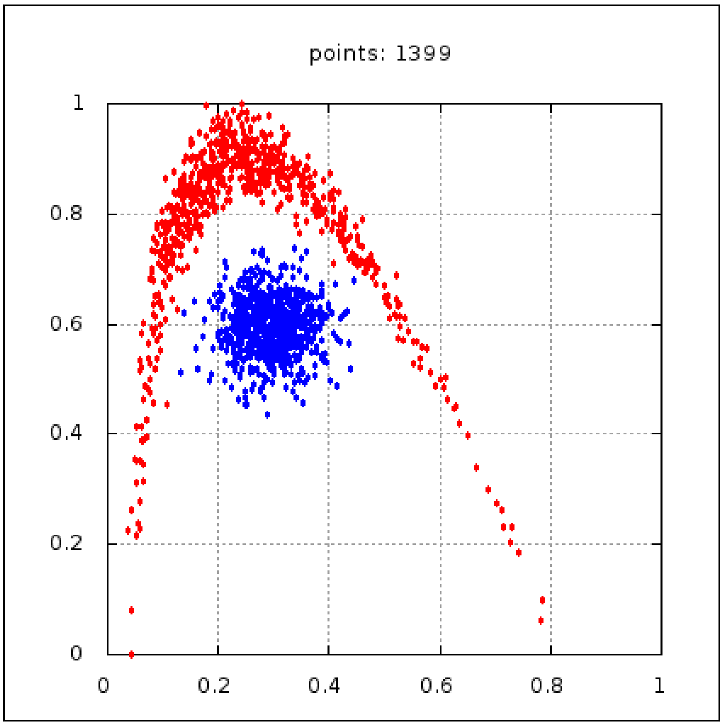


Рисунок 2 – Тестовый набор

В качестве конфигурации будет использоваться нейронная сеть с двумя обрабатывающими слоями: входной слой 2 нейрона, первый обрабатывающий слой 10 нейронов с активацией $ReLU$ ($max(x, 0)$), второй слой - 2 нейрона (по числу классов) с активацией $softmax$ (вероятностный выход). В результате были получены следующие результаты.

Метод градиентного спуска:

Сеть достигла порога ошибки за 814 эпох, результаты представлены на рисунках 3, 4:

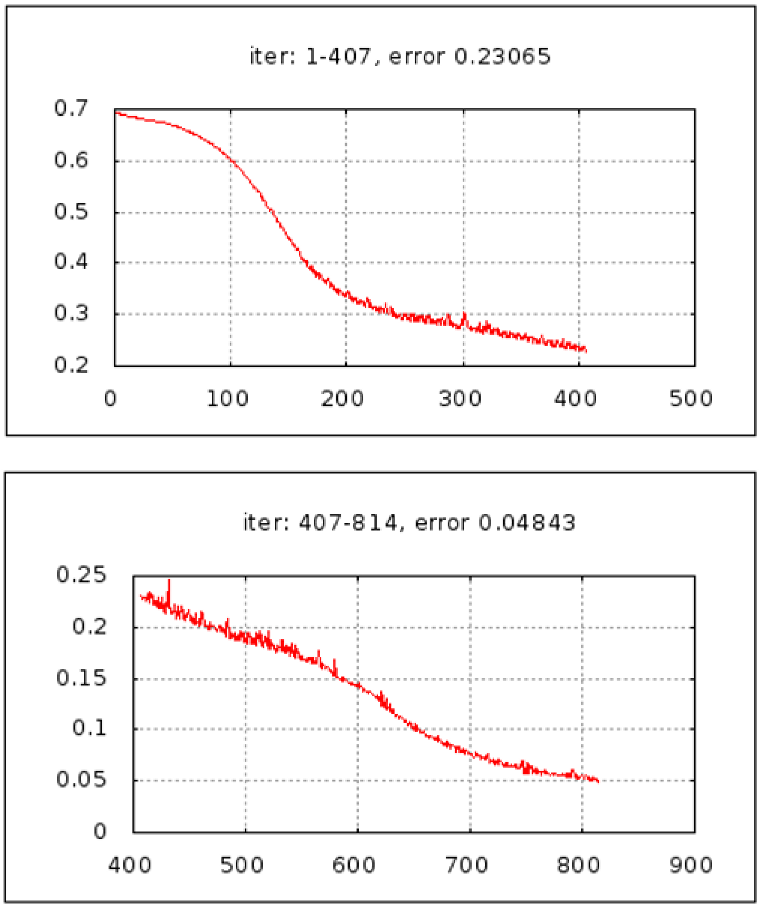


Рисунок 3 – Изменение ошибки и характер поведения градиентного метода обучения от количества итераций

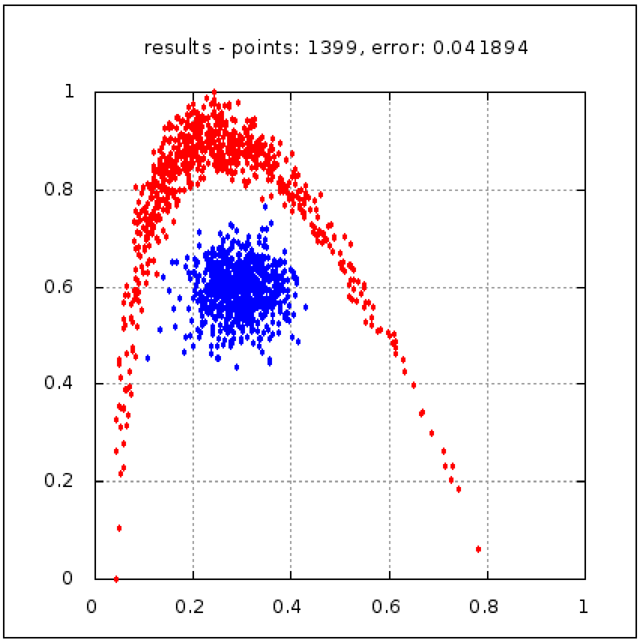
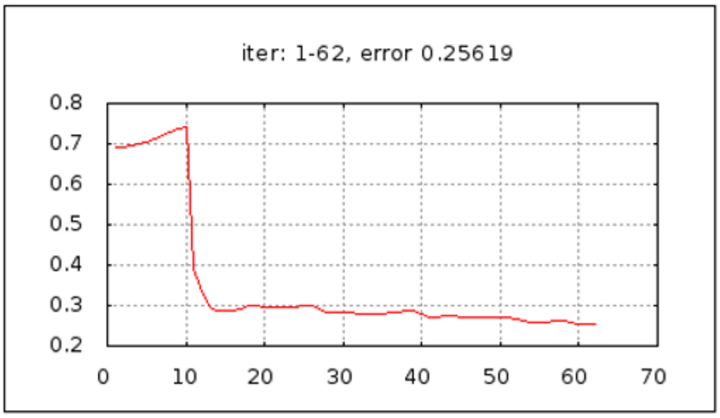


Рисунок 4 – Результат на выходе нейронной сети и общая ошибка

Метод сопряженных градиентов:

Сеть достигла порога ошибки за 124 эпохи, результаты представлены на рисунках 5, 6:



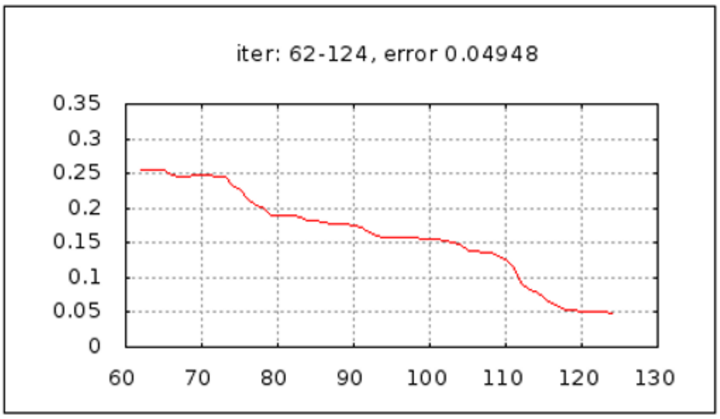


Рисунок 5 – Изменение ошибки и характер поведения метода сопряженных градиентов от количества итераций

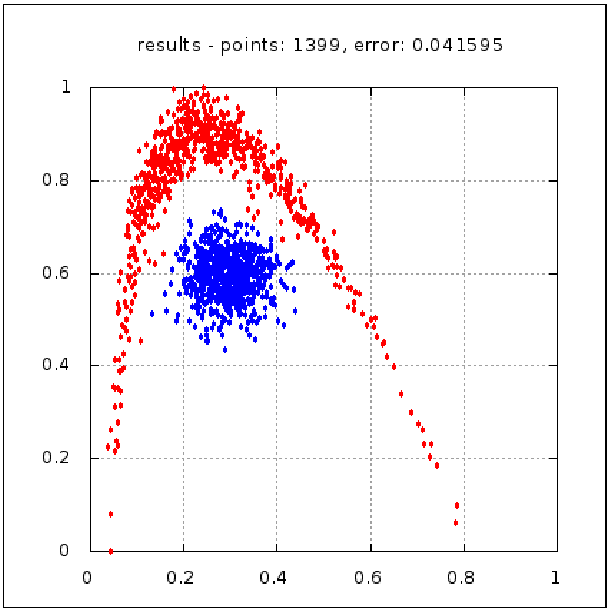


Рисунок 6 – Результат на выходе нейронной сети и общая ошибка

Метод quickProp:

Сеть достигла порога ошибки за 1074 эпохи, результаты представлены на рисунках 7, 8:

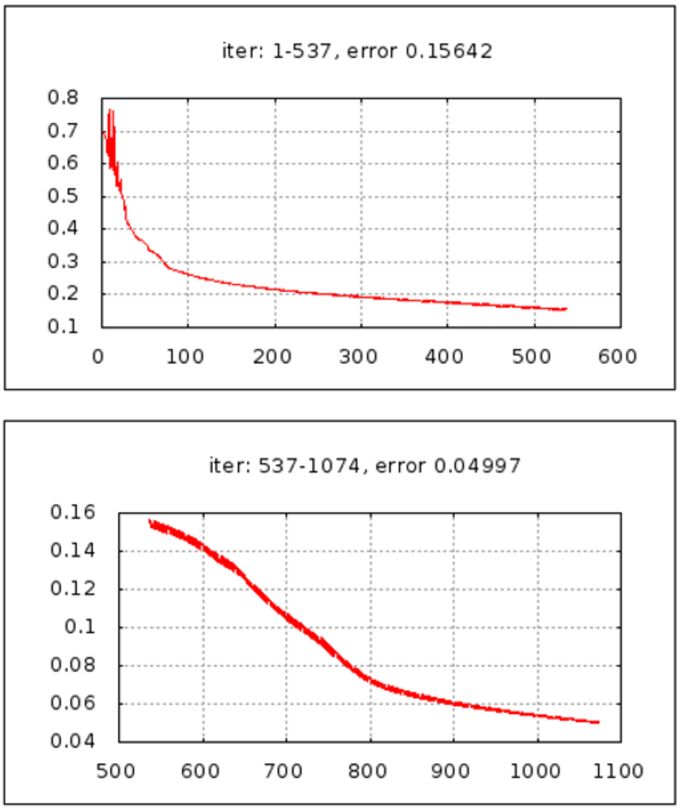


Рисунок 7 – Изменение ошибки и характер поведения метода quickProp от количества итераций

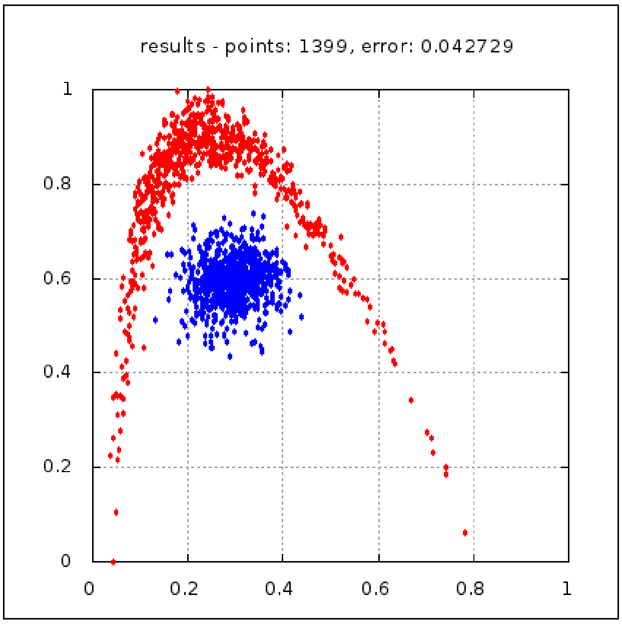


Рисунок 8 – Результат на выходе нейронной сети и общая ошибка

Метод rProp:

Сеть достигла порога ошибки за 55 эпох, результаты представлены на рисунках 9, 10:

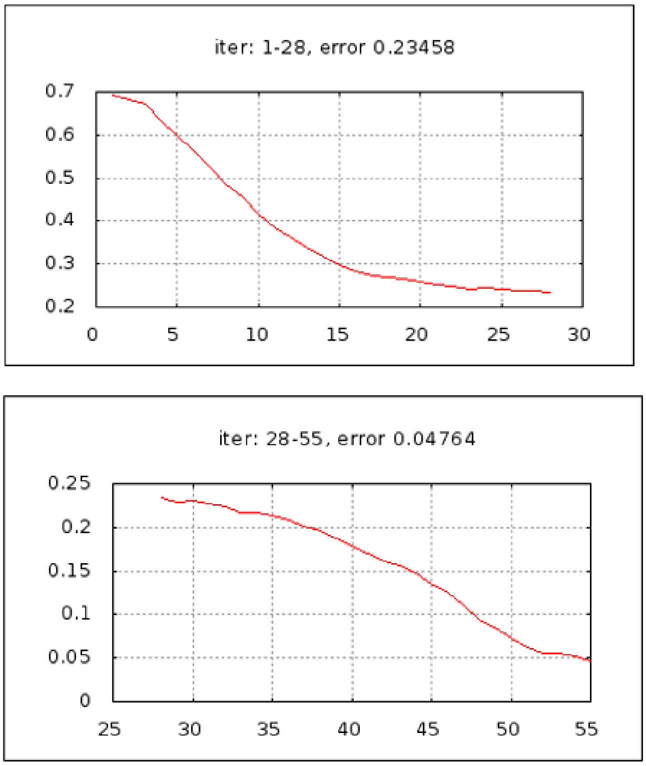


Рисунок 9 – Изменение ошибки и характер поведения метода rProp от количества итераций

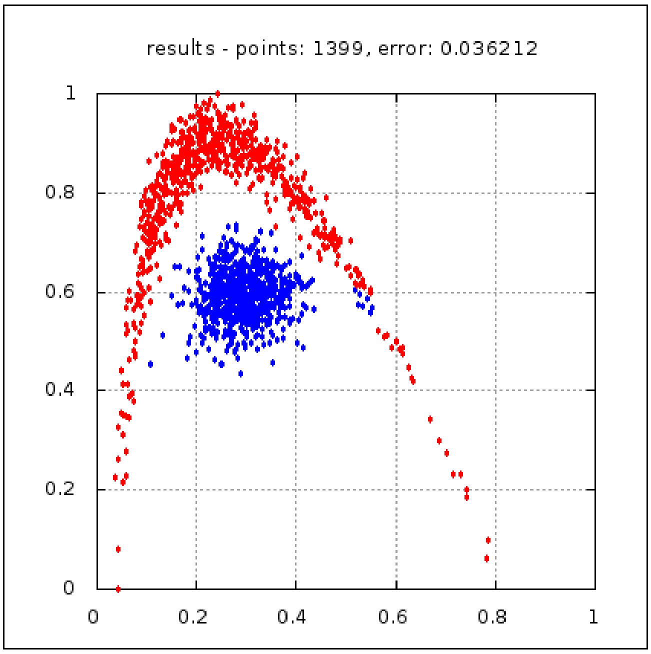


Рисунок 10 – Результат на выходе нейронной сети и общая ошибка

Из представленных данных можно сделать вывод о том, как ведет себя каждый из рассмотренных градиентных методов обучения, а также общую ошибку каждого из них при проведении тестов. Стоит отметить, что параметры при тестировании подбирались произвольно, и вероятно результаты могут быть улучшены при более тонкой настройке параметров. [10]

4. 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. 11) и обучающей выборкой (См. Рис. 12).

```
NN > nntask5 > network.txt
[[0.19528072164343302, 0.5158148013018331, 0.34088141873877287], [0.8517714402432383, 0.00293198486978985, 0.4274128155464497],
[[0.09797158628339198, 0.7894975359012046, 0.08183744447681357], [0.22222034993002915, 0.7728603098152238, 0.01365575272048869],
[[0.6198820405901041, 0.8032159663425021, 0.380901224284782], [0.10089479768923248, 0.10284599032145736, 0.018656810191543083],
[[0.8745092272306678, 0.6676837428659831, 0.0720657657368543], [0.6897119626343509, 0.6059748166244284, 0.05966676295894091], |
[[0.8859014820247403, 0.816048921727543, 0.2820322113915139], [0.5245424522083332, 0.5695152746195931, 0.5785957747300733], [0.
[[0.40261650741997246, 0.9403850273480355, 0.5998998577999967], [0.8148988489744053, 0.48400021371920043, 0.08091296721971586],
[[0.17398606168983644, 0.4493068658235081, 0.6486651617097997], [0.6345946085024372, 0.2217510622251283, 0.6864473986333702], |
[[0.10421739236276639, 0.027851601872181275, 0.9718174857180211], [0.17081312433621565, 0.23568406924873142, 0.858771628260586
[[0.3095980911301405, 0.07779746145938804, 0.11205997401417], [0.45086748219563266, 0.6233070146922259, 0.4834559831480657], [(
[[0.9341418156318695, 0.17827695410790922, 0.9282801276537669], [0.9173440895837899, 0.007849010377617427, 0.5714544853963764],
```

Рисунок 11 – Сеть на входе

```
NN > nntask5 > vectors.txt
[0.47645094681769273, 0.5384580836705234, 0.25531867387103971] -> [0.8672479558069404, 0.09221341288446117, 0.5154102314446601]
[0.9203974784180458, 0.9172466763373056, 0.6631907603888066] -> [0.07975949146146255, 0.676079148405706, 0.591701262885015]
[0.7690213559935202, 0.185446615751502, 0.10929976822059762] -> [0.9874338093618158, 0.5844327482489722, 0.43259147066891945]
```

Рисунок 12 – Обучающая выборка

Запустим программу с количеством итераций 1000 и коэффициентом 0.2. (См.Рис 13)

```
PS D:\Projects\Course_5\NN\nntask5> & "C:/Python Projects/venv/Scripts/python.exe" d:/Projects/Course_5/NN/nntask5/task1.py network.txt vec
tors.txt 1000 0.2
```

Рисунок 13 – Запуск программы

В результате получили следующий график. (См. Рис. 14)

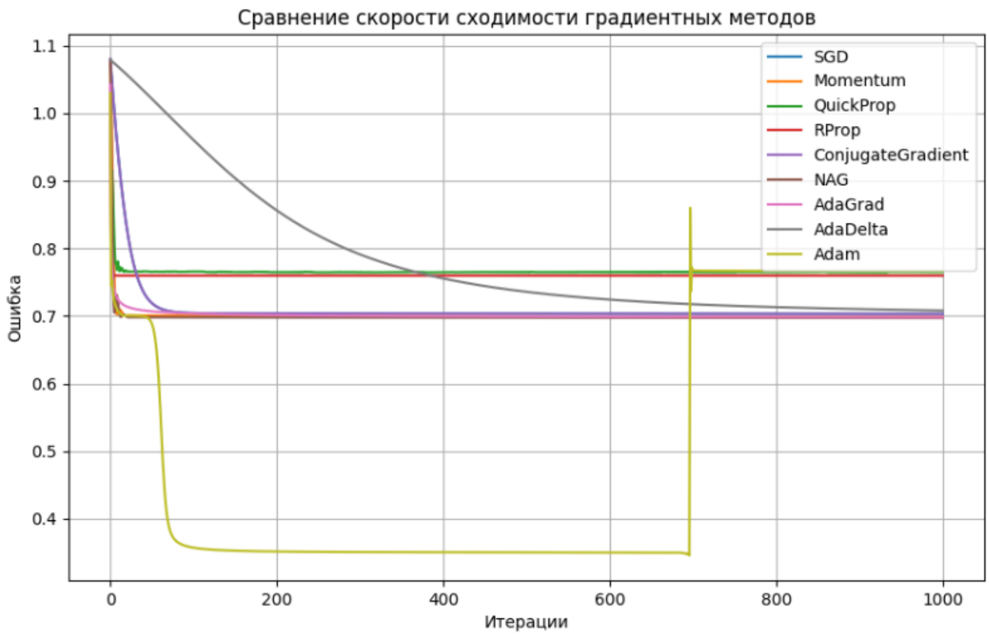


Рисунок 14 – Результат тестирования в графическом представлении

При увеличении можно увидеть характер поведения более детально. (См. Рис. 15)

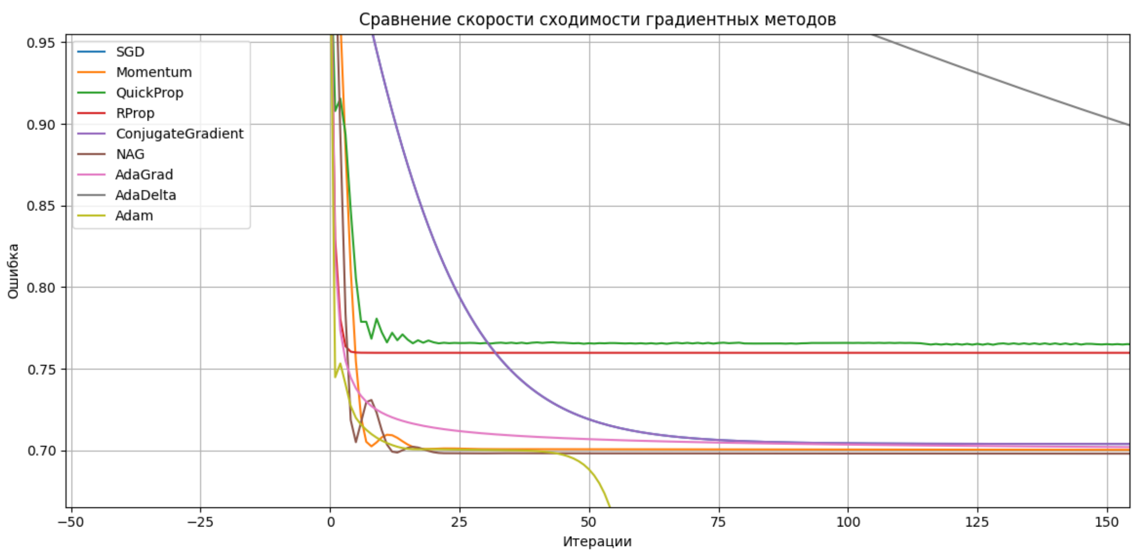


Рисунок 15 – Постепенное приближение методов первого порядка к минимуму

Поведение методов в целом схожи между собой, однако наиболее интересный характер поведения дают Adam и AdaDelta. AdaDelta сходится 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. 16)

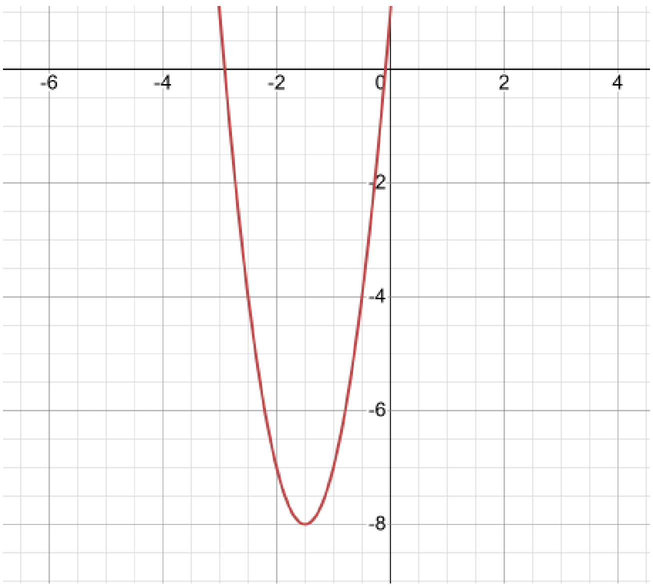


Рисунок 16 – График функции $4x^2 + 12x + 1$ с точкой в минимуме -8

В результате запуска программы получили следующий график (См. Рис. 17):

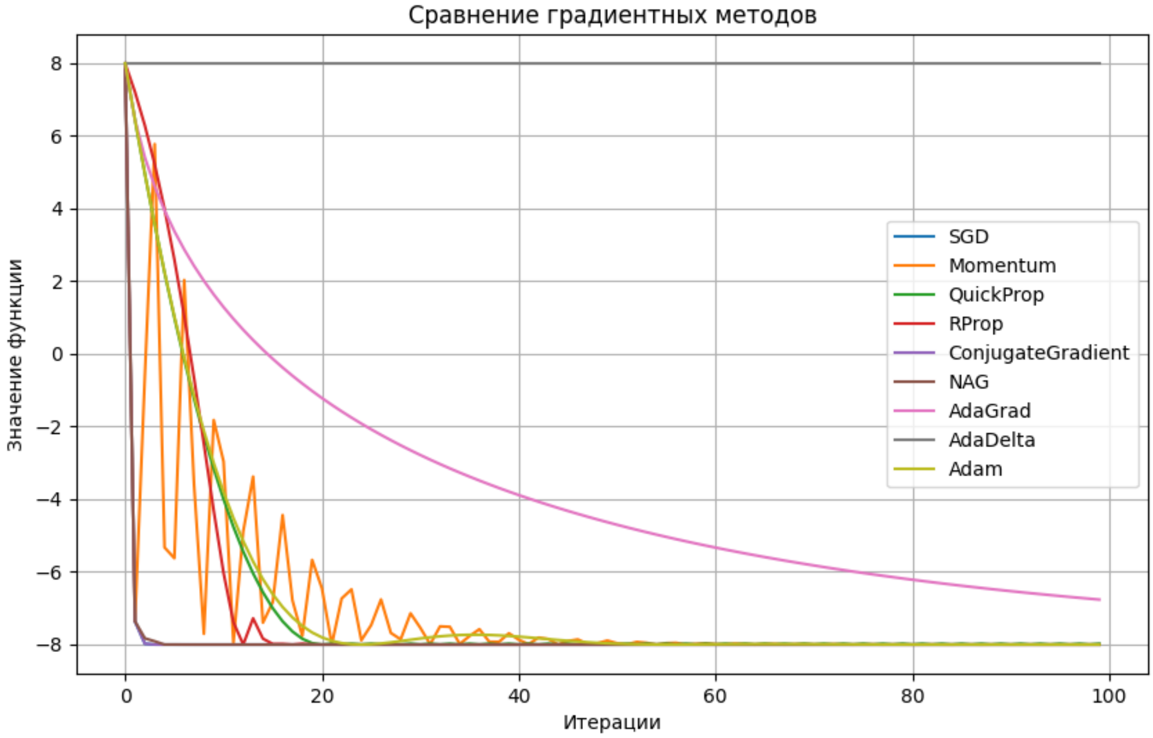


Рисунок 17 – Сходимость градиентных методов к минимуму функции за 100 итераций

Рассмотрим функции при увеличении (См. Рис. 18)

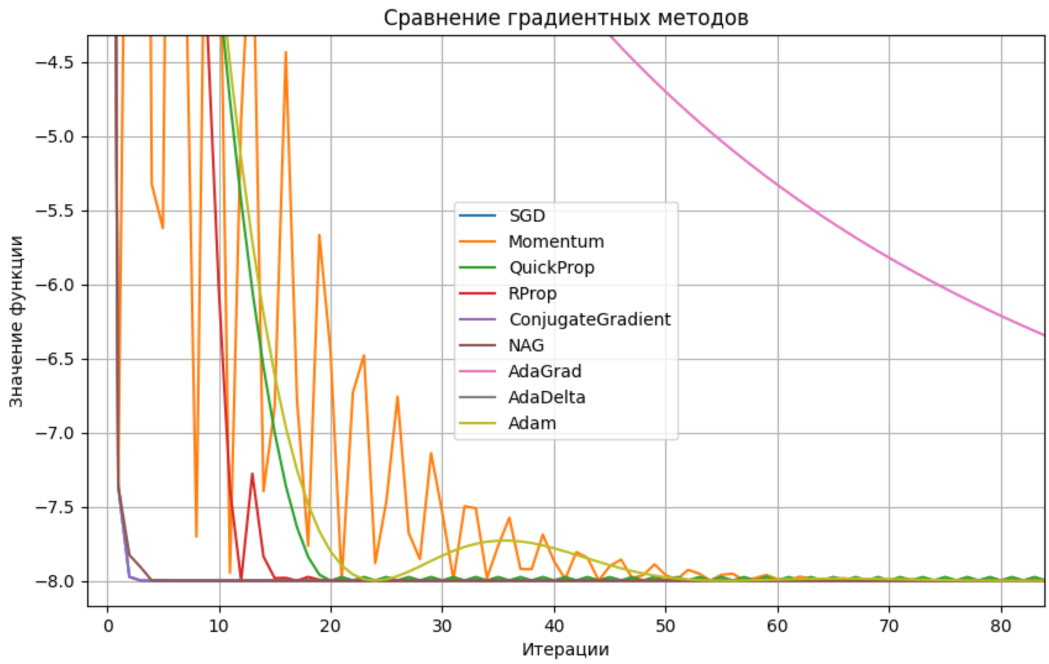
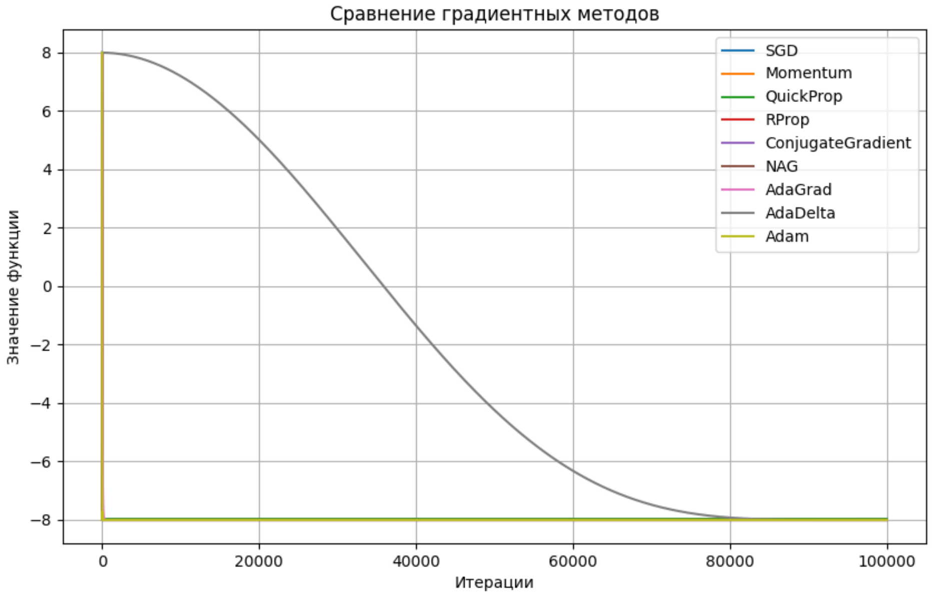


Рисунок 18 – Сходимость градиентных методов к минимуму функции за 100 (увеличенное изображение)

Можно заметить, что методы сошлись достаточно быстро, однако метод AdaDelta даже не приблизилась к ответу. Сойтись данному методу удасться только, через 100000 итераций, что очень много по сравнению с другими методами. (См. Рис. 19)



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

5. Преимущества и недостатки градиентных методов

Исходя из изученных теоретических материалов и практического наблюдения можно сформулировать основные положительные и отрицательные стороны градиентных методов 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. RProp прост в реализации, но менее стабилен в задачах с шумными данными.

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. *Крючин О.В. О*бучение искусственной нейронной сети при помощи распараллеленных градиентных алгоритмов rprop, quickprop и метода наискорейшего спуска // Актуальные инновационные исследования: наука и практика. - 2010. № 2. С. 8.
2. *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.
3. Обзор градиентных методов в задачах математической оптимизации [Электронный ресурс]: - URL: https://habr.com/ru/articles/413853/ (дата обращения 6.05.2024) – Рус.яз.
4. 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.
5. 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.
6. 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.
7. ML: Градиентный метод [Электронный ресурс]: - URL:https://qudata.com/ml/ru/ML_Grad_Method.html (дата обращения 7.12.2024) - Рус.яз.
8. 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.

9. Цыдыпова С.Ю. Гиперпараметры градиентных методов обучения нейронных сетей/ А.С.Цыбиков // В сборнике: ГЕОМЕТРИЯ МНОГООБРАЗИЙ И ЕЕ ПРИЛОЖЕНИЯ. материалы Шестой научной конференции с международным участием. Улан-Удэ. - 2020. С. 216-222.
10. Каширина И.Л. Исследование и сравнительный анализ методов оптимизации, используемых при обучении нейронных сетей/ М.В. Демченко// Вестник Воронежского государственного университета. Серия: Системный анализ и информационные технологии. - 2018. - № 4. С. 123-132.

ПРИЛОЖЕНИЕ А

Тестирование сходимости различных градиентных методов

```
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 на заданной функции

```
import matplotlib.pyplot as plt

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":
            x -= learning_rate * 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 == "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 x**2

    def gradient_function(x):
        return 2 * x

    x_init = 0.5 # Начальная точка
     methods = ["SGD", "Momentum", "QuickProp", "RProp", "ConjugateGradient", "NAG", "AdaGrad", "AdaDelta", "Adam"]
    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)
```