Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Сравнительный анализ методов оптимизации при обучении нейронных сетей

Студент ИУ9-71Б (Группа) _____ (Подпись, дата) **Т. Р. Ионов** (И.О.Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) **Ю. Т. Каганов** (И.О.Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О.Фамилия)

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Самым популярным инструментов искусственного интеллекта считаются нейронные сети. С каждым днем появляются новые архитектуры искусственных нейронных сетей, которые решают все больше прикладных задач, начиная от оценки кредитоспособности и заканчивая генерацией реалистичных изображений и аудиозаписей. Задачи машинного обучения можно разделить на три класса:

1. обучение без учителя, в котором неизвестен правильный ответ системы на входные данные;
2. обучение с учителем, в котором известны правильные ответы поставленной задачи;
3. обучение с подкреплением, в котором модель получает стимулы от окружающей среды.

Самым распространённым является второй тип - обучение с учителем, поскольку он позволяет моделировать системы, выдающие прогнозируемый результат на основе размеченных данных. Соотвественно, для получения требуемого результата, необходимо оптимизировать нейронную сеть.

Общая постановка задачи оптимизации порождает большое разнообразие методов, каждый из которых эффективен в некоторой области. Метод оптимизации характеризуется целевой функцией (функцией потерь) и допустимая область. Существует классификация в соотвествии с задачей оптимизации:

- Локальные методы: сходятся к некоторому локальному экстремуму целевой функции;
- Глобальные методы: выявляют тенденции глобального поведения целевой функции.

Задача обучения нейронной сети относится к поиску локального минимум целевой функции.

По критерию размеренности существует разделение на одномерную и многомерную оптимизацию. К последней относится оптимизация

многочисленных параметров модели. Также, существует классификация по методам поиска:

1. Детерминированные;
2. Стохастические (случайные);
3. Комбинированные;

Детерминированные методы не подходят для работы с большими данными, поэтому в данной задачи уместнее всего использовать стохастические методы, одним из которых является стохастические градиентный спуск.

Градиентный спуск - численный метод нахождения локального минимум с помощью движения вдоль градиента с некоторым шагом. Обучение нейронной сети - процесс минимизации функции потерь за счет оптимизации обучаемых параметров. Основное преимущественно данного метода - итеративное использование данных: на каждой итерации выбирается пакет данных и совершается спуск. Оптимизатором называется метод оптимизации параметров нейронных сетей, в данном случае - метод минимизации функции потерь. Проведем анализ различных модификаций оптимизаторов на основе градиентного спуска при обучении искусственных нейронных сетей.

1. Обзор методов оптимизации в задаче обучения нейронной сети

Главной идеей всех современных методов оптимизации в задаче обучения нейронной сети - движение вдоль градиента с некоторым шагом. Данный метод применим для решения задач в многомерных пространствах, что позволяет оптимизировать многослойные нейронные сети, содержащие сотни тысяч параметров. Для достижения удовлетворяющей обобщающей способности нейронной сети необходимо большое количество данных. Вычисление функции ошибки на всем тренировочном подмножестве вычислительно требовательно, поэтому ошибка считается не на всей выборке, а итерационно по небольшим подвыборкам - пакетам.

Введем следующие основные обозначения:

t - индекс итерации обучения;

θ - обучаемые параметры модели;

f - функция потерь;

γ - скорость обучения;

∇ - оператор взятия частных производных;

ε - маленькое ненулевое число порядка 10^{-8} степени.

1.1 Стохастический градиентный спуск

Стохастический градиентный спуск (англ. Stochastic gradient descent, SGD) [1] - самая простая модификация градиентного спуска, лежащая в основе всех последующих оптимизаторов. В отличие от стандартного градиентного спуска, шаг оптимизации происходит по пакетам.

Ниже приводится алгоритм цикла обучения с использованием метода градиентного спуска (1.1):

$$\begin{aligned} &\text{for } t = 1 \text{ to } \ldots \text{ do:} \\ &\quad g_t \leftarrow \nabla_\theta f_t\left(\theta_{t-1}\right) \\ &\quad \theta_t \leftarrow \theta_{t-1} - \gamma g_t \end{aligned}$$

Преимуществами данного метода является вычислительная легкость и минимальное потребление памяти. Главным недостатком является постоянное значение скорости обучения, требующего большей внимательности к выбору ее

выбору. При больших значениях, алгоритм имеет возможность не сойтись, а при маленьких - оптимизация займет длительное время. Вместе с тем, существует возможность выхода из локального или глобального минимума, поскольку шаг оптимизатора напрямую связан с текущим пакетом и антиградиент будет плохо обобщен для всего набора данных.

1.2 Импульс

Импульс (англ. Momentum) [2] является модификацией стохастического градиентого спуска и уменьшает дисперсию измнения весов θ, используя физический закон движения для преодоления локальных минимумов.

Ниже приводится алгоритм цикла обучения с использованием импульса (1.2):

$$\begin{aligned} &\text{for } t = 1 \text{ to } \ldots \text{ do:} \\ &\quad v_t \leftarrow \mu v_{t-1} + \nabla_\theta f_t(\theta_{t-1}) \\ &\quad \theta_t \leftarrow \theta_{t-1} - \gamma v_t \end{aligned}$$

где $\mu \in [0, 1]$ - коэффициент влияния импульса.

Преимуществом данного алгоритма является более быстрая скорость сходимости по сравнению со стохастическим градиентным спуском за счет использования импульса с предыдущего шага. В зависимости от величины импульса, оптимизатор способен преодолевать плато. Геометрическая интерпретация сравнения алгоритмов изображена на рисунке 1.1.

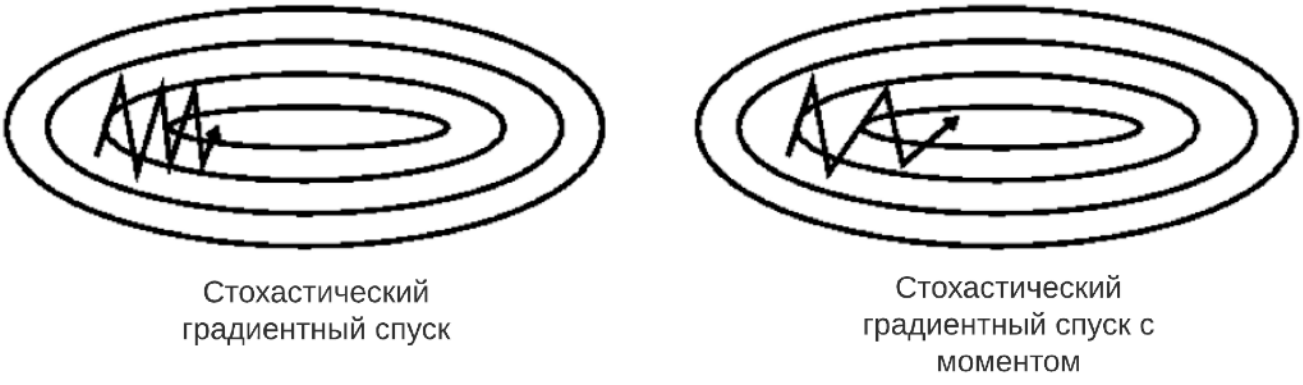

Рисунок 1.1 - сравнение классического метода и импульса.

Недостатком алгоритма является гиперпараметр μ, который необходимо подбирать эмпирически: при близких к нулю значениях алгоритм теряет преимущества импульса, при близких к единице появляется вероятность пропуска минимумов. Избегание застревания в неглубоких минимумах может ухудшить сходимость, вследствие появления чрезмерной инерции.

1.3 Импульс Нестерова

Импульс Нестерова (англ. Nesterov momentum) [3] - модификация импульса. Он обладает большей скоростью сходимости по сравнению со стандартным методом импульса. Основная идея алгоритма заключается в том, что вместо вычисления градиента на текущей позиции, вычисляется градиент на аппроксимированной новой позиции, то есть заново считается функция потерь f при обновленных параметрах θ. Геометрическая интерпретация Нестерова импульса изображена на рисунке 1.2.

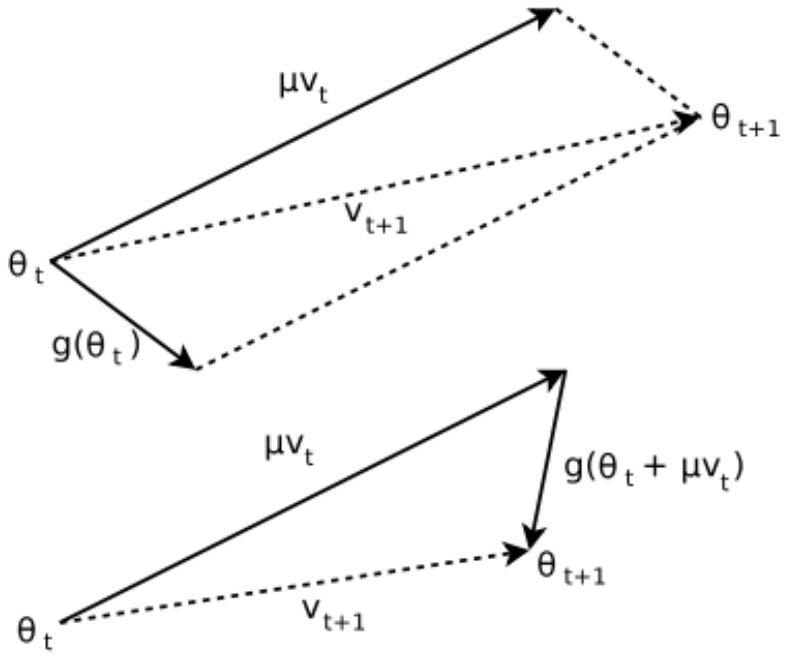

Рисунок 1.2 - сравнение классического импульса и Нестерова импульса.

Это объясняется тем, что если у оптимизатора накоплен некоторый импульс, то в текущем состоянии уже известно, куда в будущем будет направлен шаг, что позволяет вычислить градиент в новой точке, лежащей на шаг ближе по направлению момента.

Ниже приводится алгоритм цикла обучения с использованием импульса Нестерова (3):

$$\begin{aligned}&\text{for } t = 1 \text{ to } \ldots \text{ do:}\\&\quad v_t \leftarrow \mu v_{t-1} + \nabla_\theta f_t\left(\theta_{t-1} - \mu v_{t-1}\right)\\&\quad \theta_t \leftarrow \theta_{t-1} - \gamma v_t\end{aligned}$$

где $\mu \in [0, 1]$ - коэффициент влияния Нестерова импульса.

Преимуществом данного метода является использование момента к новой точке, что позволяет быстрее находить минимум. Главным недостатком алгоритма является дополнительные вычисления градиента и затраты по памяти на копию модели. Таким образом, на каждой итерации производится два прохода данных через нейронную сеть и два вычисления ошибки. Проблема классического импульса с возможным накоплением чрезмерно высокой инерции остается.

1.4 Адаптивный градиент

Все предыдущие методы имели один общий недостаток - постоянный шаг скорости обучения γ для всего набора параметров θ. Адаптивный градиент (англ. Adaptive gradient, AdaGrad) [4] - первый представитель адаптивных алгоритмов градиентного спуска, автоматически регулирующих скорость обучения.

Ниже приводится алгоритм цикла обучения с использованием адаптивного градиента (4):

$$\begin{aligned}&\text{for } t = 1 \text{ to } \ldots \text{ do:}\\&\quad g_t \leftarrow \nabla_\theta f_t\left(\theta_{t-1}\right)\\&\quad N_t \leftarrow N_{t-1} + g_t^2\\&\quad \theta_t \leftarrow \theta_{t-1} - \gamma \frac{g_t}{\sqrt{N_t} + \varepsilon}\end{aligned}$$

где N - переменная, аккумулирующая квадраты градиентов ($N_0 = 0$).

В данном методе ε используется для численной стабильности, поскольку корень в знаменателе может быть равен нулю.

Стоит отметить, что векторные операции в данном методе поэлементные. Из этого вытекает главная особенность алгоритма - скорость обучения изменяется адаптивно для каждого параметра: параметры, которые изменялись сильно на прошлой итерации, будут замедлены и наоборот. Также, параметр γ не требует тонкой настройки - скорость обучения корректируется в процессе.

Из недостатков стоит выделить вычислительные затраты при математических операциях, затратах памяти на переменную N и неограниченность N_t, что может привести к остановке обновления параметров.

1.5 Среднее квадратичное распространение

Аккумулирующая переменная N в предыдущем методе растёт неограниченного и монотонно, что приводит к монотонному уменьшению шага обучения, влекущее в конечном счете к затуханию градиента. Чтобы исправить эту проблему, был разработан метод среднего квадратического распространения (англ. Root mean squared propogation, RMSprop) [5]. RMSprop использует скользящее среднее, уменьшая наколенный градиент в прошлом.

Ниже приводится алгоритм цикла обучения с использованием среднего квадратичного распространения (5):

$$\text{for } t = 1 \text{ to } \ldots \text{ do:}$$
$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$$
$$N_t \leftarrow \alpha N_{t-1} + (1-\alpha) g_t^2$$
$$\theta_t \leftarrow \theta_{t-1} - \gamma \frac{g_t}{\sqrt{N_t} + \varepsilon}$$

где N - переменная, аккумулирующая квадраты градиентов ($N_0 = 0$);

$\alpha \in (0, 1)$- коэффициент влияния прошлых градиентов.

Скользящее среднее позволяет исправить недостаток адаптивного градиента и производить преждевременную остановку обучения. Преимущества прошлого алгоритма сохраняются.

Вычислительные недостатки предыдущего алгоритма остаются, к ним добавляется еще один незначительный - дополнительный гиперпараметр, который подбирается эмпирически.

1.6 Адаптивный импульс

Адаптивный импульс (англ. Adaptive momentum, Adam) [6] является самым распространённым оптимизатором, совмещающим два похода: накопление импульса и адаптивное изменение скорости обучения. Идея Adam - это комбинация RMSprop и Momentum. В отличие от импульса, идет накопление не изменения параметров θ, а непосредственно градиентов, и в отличие от среднего квадратичного распространения используются два скользящих средних по производной первого и второго порядка.

Ниже приводится алгоритм цикла обучения с использованием адаптивного импульса (5):

$$\text{for } t = 1 \text{ to } \ldots \text{ do:}$$

$$g_t \leftarrow \nabla_\theta f_t\left(\theta_{t-1}\right)$$

$$M_t \leftarrow \beta_1 M_{t-1} + \left(1 - \beta_1\right) g_t$$

$$N_t \leftarrow \beta_2 N_{t-1} + \left(1 - \beta_2\right) g_t^2$$

$$\hat{m}_t \leftarrow \frac{M_t}{1 - \beta_1^t}$$

$$\hat{n}_t \leftarrow \frac{N_t}{1 - \beta_2^t}$$

$$\theta_t \leftarrow \theta_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{n}_t} + \varepsilon}$$

Где M - переменная, аккумулирующая градиенты ($M_0 = 0$);

$\beta_1 \in \left[0, 1\right]$ - гиперпараметр при градиенте первой степени;

N - переменная, аккумулирующая квадраты градиентов ($N_0 = 0$);

$\beta_2 \in \left[0, 1\right]$ - гиперпараметр при градиенте второй степени;

Данный оптимизатор совмещает преимущества импульсных и адаптивных методов. Скорость обучения не требует тонкой настройки. Adam решает проблему медленного накопления за счет $1 - \beta^t$ в знаменателе, свойственную RMSprop из-за инициализации нулями.

Из недостатков алгоритма стоит выделить лишь вычислительные затраты. Помимо этого, необходимо получать информацию о текущей эпохе, в то время

как остальные алгоритмы независимы от этого параметра. Таким образом, количество эпох и разбиение на пакеты выступает в роли дополнительного неявного гиперпараметра, определяемого исключительно эмпирическим способом.

Существует множество других стохастических градиентных методов оптимизации, построенных на уже известных идеях, однако имеет смысл сравнить основных представителей из каждого класса.

2. Разработка методов оптимизации и выбор тестируемых моделей

В качестве языка программирования для реализации методов оптимизации, моделей и цикла обучения был выбран Python [7], как самый популярный для задачи машинного обучения. В качестве среды разработки был выбран интерактивный блокнот Jupyter [8], позволяющий хранить вычислений нескольких ячеек кода и визуализировать результаты с помощью веб-страницы.

Воспользуемся библиотекой с открытым исходным кодом NumPy [9] для быстрых операций с многомерными массивами и Matplotlib [10] для визуализации результатов обучения.

2.1 Разработка модели и функции потерь

В первую очередь, необходимо определить формат модели и целевые функции. В качестве модели была выбрана полносвязная нейронная, поскольку с помощью нее можно наглядно сравнить методы оптимизации. Схема искусственного нейрона изображена на рисунке 2.1.

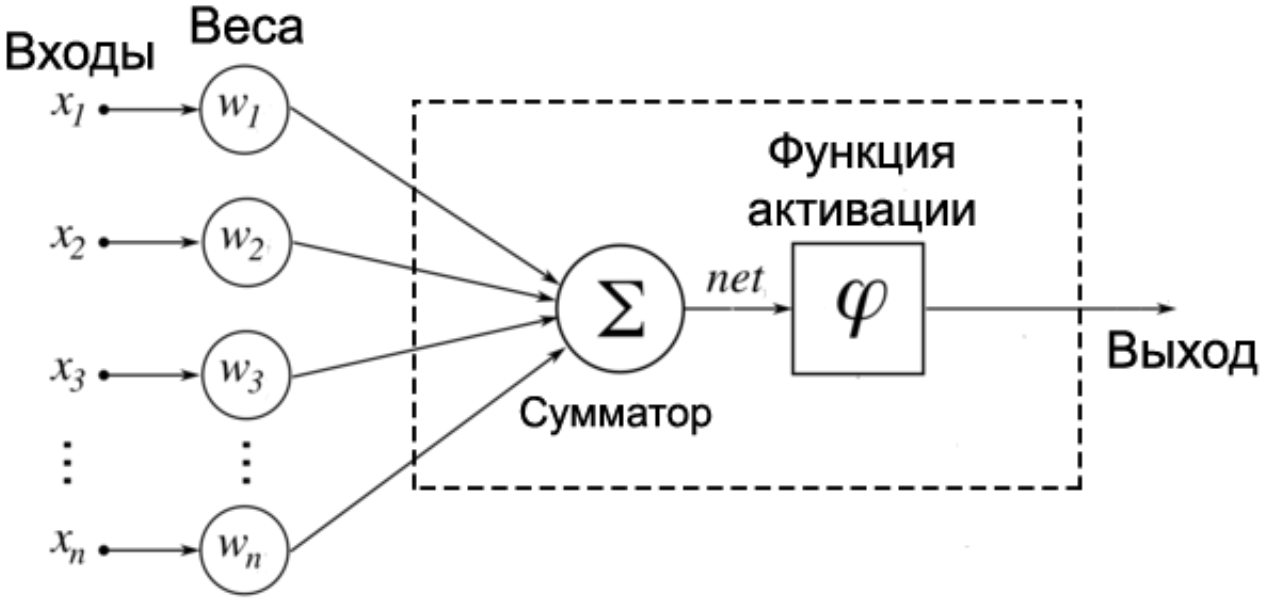

Рисунок 2.1 - схема искусственного нейрона

Таким образом, необходимо разработать слой, хранящий веса и производящий прямой и обратный проход, функцию активации и функцию потерь. В памяти компьютера веса удобно хранить в виде многомерной матрицы чисел, тогда сумматором выступает скалярное произведение. Реализация полносвязаного слоя нейронной сети представлена листингом 2.1.

Листинг 2.1 - класс полносвязного слоя нейронной сети.

```
class Linear():
    def __init__(self, n_in, n_out):
        self.w = np.random.randn(n_in,n_out)*np.sqrt(2/n_in)
        self.b = np.zeros(n_out)

    def forward(self, x):
        self.old_x = x
        return np.dot(x, self.w) + self.b

   def backward(self, grad):
        self.grad_b = grad.mean(axis=0)
        bs = grad.shape[0]
        self.grad_w = (self.old_x.T @ grad) / bs
        return np.dot(grad, self.w.T)
```

Пары атрибутов w и b, grad_w и grad_b образуют обучаемые параметры модели и их градиенты соответственно. Таким образом, прямой и обратный проход осуществляются итеративно по слоям, выход одного слоя является входом другого в обоих случаях.

В качестве функции активации используется сигмоида, вычисляемая по формуле (2.1):

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
$$\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right)$$

Графики сигмоиды и ее производной изображены на рисунке 2.2.

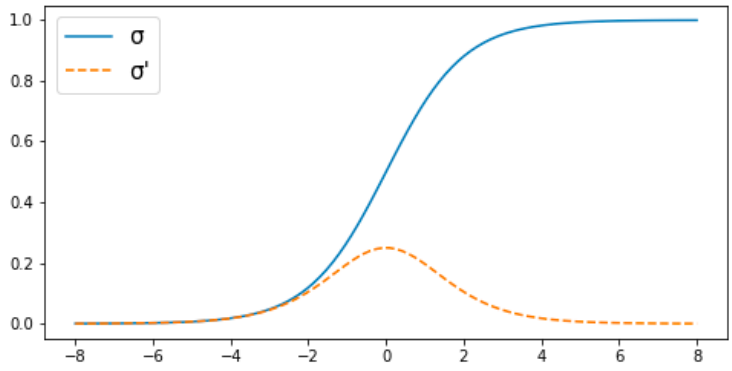

Рисунок 2.2 - функция активации сигмоида и ее производная

Для реализации прямого прохода нейронной сети и обратного распространения ошибки необходимо сохранять общую логику для

взаимодействия данных, то есть реализовать функцию активации как слой. Реализация функции активации представлена листингом 2.2.

Листинг 2.2 - класс функции активации.

```
class Sigmoid():
    def forward(self, x):
        self.old_y = np.exp(x) / (1 + np.exp(x))
        return self.old_y

    def backward(self, grad):
        return self.old_y * (1 - self.old_y) * grad
```

Построенная модель будет предназначена для классификации поэтому воспользуемся стандартной для данной задачи функцией потерь - среднеквадратичной ошибкой (англ. man squared error, MSE), вычисляемой по формуле (2.2):

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

где n - общее число примеров;

y - вектор наблюдаемых значений;

$\hat{y}$ - вектор предсказаний модели.

Также, MSE имеет достаточно простую производную, позволяющую без сильных вычислительных затратах находить градиент. Реализация MSE представлена листингом 2.3.

Листинг 2.3 - класс функции потери.

```
class MSE():
    def forward(self, x, y):
        self.old_x = x, self.old_y = y
        return (1/2 * np.square(x-y)).mean()

    def backward(self):
        return self.old_x - self.old_y
```

Реализуемый класс сохраняет логику послойного прохождения данных.

2.2 Разработка методов оптимизации и цикла обучения

Оптимизатор - ключевой элемент в обучении нейросетевых моделей. Общая постановка задачи для оптимизатора - минимизировать функцию потерь при заданных параметрах θ. Для этого, необходимо иметь доступ к обучаемым

параметрам модели, их градиентам и реализовывать шаг оптимизации. На листинге 2.4 представлен класс стохастического градиентного спуска.

Листинг 2.4 - класс стохастического градиентного спуска.

```
class SGD():
    def __init__(self, lr=0.01):
        self.lr = lr

    def init_params(self, model):
        self.layers = [
l for l in model.layers if type(l)==Linear
]

    def step(self):
        for layer in self.layers:
            layer.w -= self.lr * layer.grad_w
            layer.b -= self.lr * layer.grad_b
```

Атрибут layers хранит обучаемые параметры модели, метод init_params вынесен за конструктор по причине того, что классы слоев и оптимизатора инициализируются независимо и взаимодействуют внутри класса модели, отвечающей за цикл обучения. Класс модели изображен на листинге 2.5.

Листинг 2.5 - класс модели

```
class Model():
    def __init__(self, layers, cost, optimizer,
num_classes=10):
        self.layers = layers
        self.cost = cost
        self.optimizer = optimizer
        self.optimizer.init_params(self)
        self.num_classes = num_classes

    def one_hot(self, label):
        return np.eye(self.num_classes)[label]

    def forward(self,x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def loss(self,x,y):
        return self.cost.forward(self.forward(x), y)

    def backward(self):
        grad = self.cost.backward()
        for i in range(len(self.layers)-1, -1, -1):
            grad = self.layers[i].backward(grad)
```

Продолжение листинга 2.5.

```
    def train(self, train_loader, epochs):
        # итерация по эпохам
        for epoch in range(epochs):
            # итерация по пакетам
            for x, label in train_loader:
                # преобразование лейбла в вектор
                y = self.one_hot(label)
                # прямой проход и рассчет потерь
                self.loss(x, y)
                # обратный проход, накопление градиентов
                self.backward()
                # шаг оптимизатора
                self.optimazier.step()
```

Конструктор модели принимает на вход массив из чередующихся полносвязных слоев и слоев активации, экземпляр класса функции потерь (в нашем случае, MSE) и экземпляр класса оптимизатора. Метод one_hot преобразовывает метку класса в вектор с помощью унитарного кода.

Последующие оптимизаторы во многом аналогично реализованы, однако импульс Нестерова отличается тем, что вычисляет градиент второй раз. алгоритм шага импульса Нестерова изображен на листинге

Листинг 2.6 - реализация шага Нестерова импульса.

```
    def step(self):
        # создаем копию модели
        model_ahead = deepcopy(self.model)
        ahead_layers = [l  for l in model_ahead.layers if type(l)
== Linear]
        # совершает шаг, получая модель из "будущего"
        for i, layer in enumerate(ahead_layers):
            layer.w -= self.m * self.velocity_w[i]
            layer.b -= self.m * self.velocity_b[i]
        # считаем новые градиенты
        model_ahead.loss(self.model.cur_x, self.model.cur_y)
        model_ahead.backward()

       ahead_layers = [l  for l in model_ahead.layers if type(l)
== Linear]
        # к старым слоям применяем новые градиенты
        for i, (ahead_layer, layer) in
enumerate(zip(ahead_layers, self.layers)):
            self.velocity_w[i] = self.m * self.velocity_w[i] +
self.lr * ahead_layer.grad_w
```

Продолжение листинга 2.6.

```
            self.velocity_b[i] = self.m * self.velocity_b[i] +
self.lr * ahead_layer.grad_b

            layer.w -= self.velocity_w[i]
            layer.b -= self.velocity_b[i]
```

Поля velocity_w и velocity_b хранят имульс обучаемых параметров слоев. В цикл обучения добавляется сохранение текущего пакета для повторного прохода.

3. Подготовка набора данных для обучения нейронной сети

Для работы с данными воспользуемся фреймворков машинного обучения для языка Python с открытым исходным кодом Pytorch[11], данный фреймворк позволит загрузить и обработать набор данных в пару строк кода. В качестве наборов данных используются стандартные MNIST [12] (Modified National Institute of Standards and Technology) и Fashion MNIST.

3.2 Обзор данных для обучения

База данных MNIST представлена в виде образцов рукописного написания цифр и является стандартом, предложенным Национальным институтом стандартов и технологий США, с целью калибровки и сопоставления методов распознавания изображений с помощью машинного обучения, в первую очередь на основе нейронных сетей. База данных MNIST содержит 60000 изображений для обучения и 10000 изображений для тестирования. Пример данных изображен на рисунке 3.1.

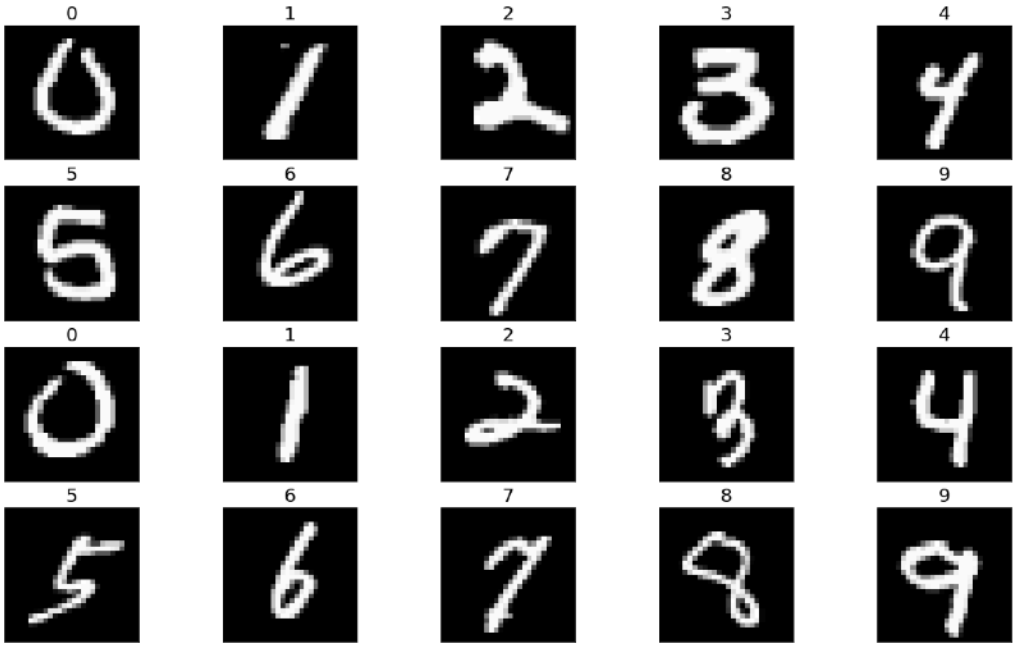

Рисунок 3.1 - пример данных набора MNIST

Изображения представлены в черно-белом формате размером 28x28 пикселей, принимающих значения от 0 до 255.

Однако, MNIST является довольно простым набором данных и может быть проблематично сравнивать алгоритмы с одинаково высокой точностью. В

качестве дополнительного набора данных выберем Fashion MNIST, представляющий из себя изображения элементов одежды. Примеры данных Fashion MNIST изображен на рисунке 3.3.

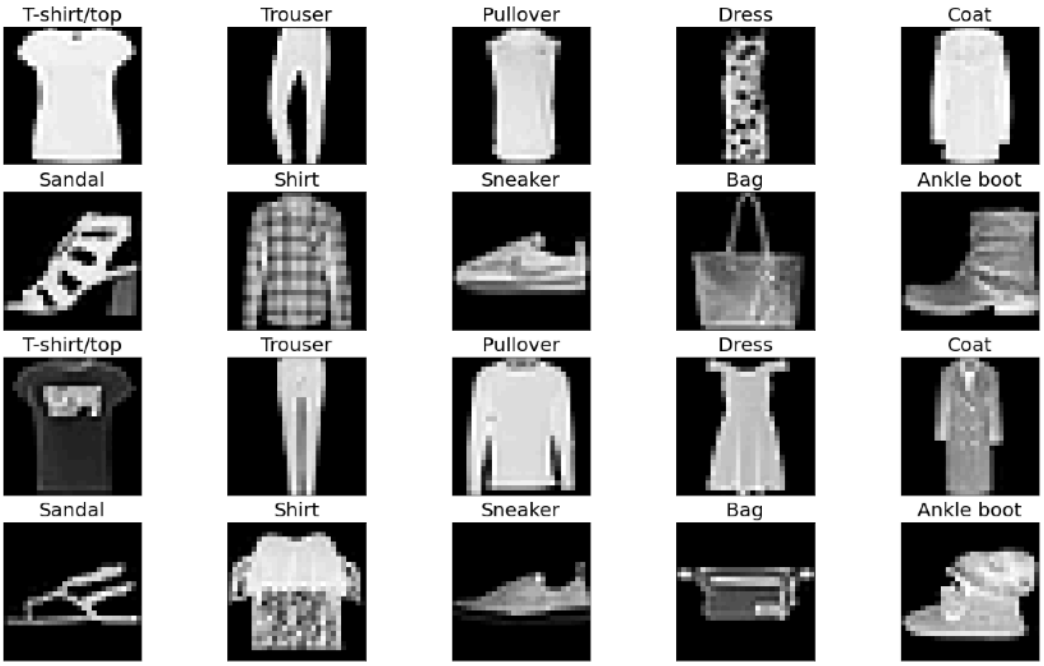

Рисунок 3.2 - пример данных набора Fashion MNIST

Домен элементов одежды несколько сложнее устроен и имеет больше разнообразия. Рассмотрим распределения классов в наборах данных для подбора оптимальной метрики качества предсказания. Распределения классов изображены на рисунке 3.3.

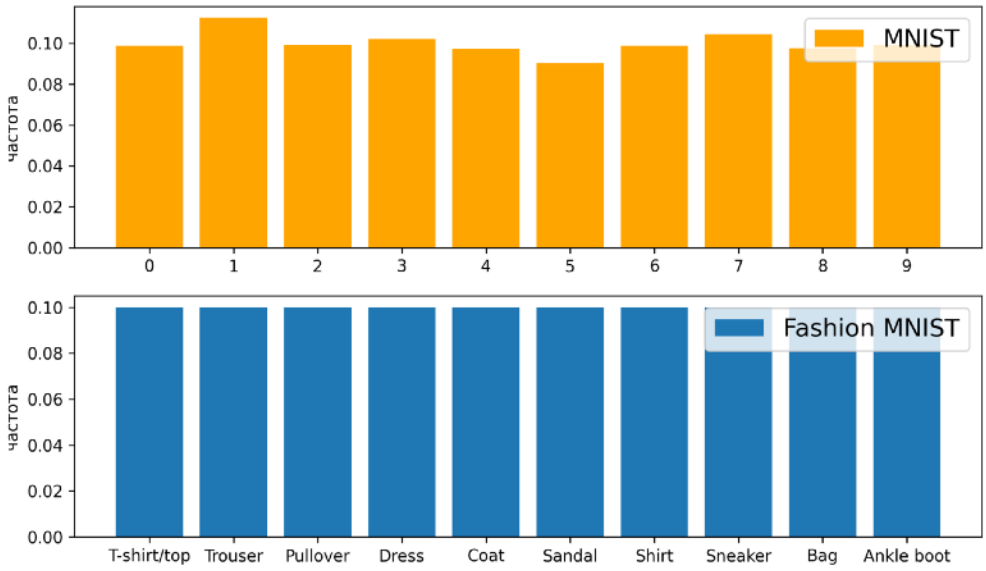

Рисунок 3.3 - распределения классов в наборах данных

Таким образом, в корректирование набора данных нет необходимости, модель не станет переобучаться под конкретный класс, поэтому корректно использовать метрику качества - долю верных предсказаний (англ. accuracy), рассчитываемую по формуле (3.1):

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n} \sum_{1}^{n} (y_i = \hat{y}_i)$$

где n - общее число примеров;

y - вектор наблюдаемых значений;

$\hat{y}$ - вектор предсказаний модели.

3.2 Обработка данных для обучения

Так как полносвязная нейронная сеть работает с векторным представлением данных, необходимо представить изображение в виде одномерного вектора, то есть понизить размерность с двух до единицы. Для этого воспользуемся преобразованием двумерного массива к одномерному с помощью формулы (3.2):

$$b_k = a_{i,j}, i = \overline{1 \ldots n}, j = \overline{1 \ldots m}, k = i * m + j$$

где b - одномерное представление изображения;

a - двумерное представление изображения;

n - количество строк в массиве a;

m - количество столбцов в массиве a;

Помимо выпрямления, необходимо нормализовать и центрировать вектор, поскольку поведение нейросетевой модели нестабильно при числах, по модулю больших единицы. Для этого воспользуемся приведением к интервалу (-0.5, 0.5) с помощью формулы (3.3):

$$x'_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} - 0.5$$

где x' - нормализованный вектор;

x - исходный вектор;

x_{min} - минимальное значение вектора x;

x_{max} - максимальное значение вектора x;

Реализация функций загрузки и обработки данных представлены на листинге 3.1:

Листинг 3.1 - реализация загрузки и обработки данных

```
import torchvision
from torchvision import datasets

class Normalize:
    def __call__(self, img):
        img = np.array(img)
        return (img - img.min())/(img.max() - img.min())-0.5

class Flatten:
    def __call__(self, img):
        # 1x28x28 -> 28x28
        img = img.squeeze()
        n, m = img.shape
        b = np.zeros(shape=(n*m))
        for i in range(n):
            for j in range(m):
                b[i*m + j] = img[i][j]
        return b

transforms = torchvision.transforms.Compose([
    Normalize(),
    Flatten(),
])

    train_dataset = datasets.MNIST(root='data', train=True,
download=True, transform=transforms)
    train_loader = DataLoader(train_dataset, batch_size=64,
shuffle=True)
```

Переменная transforms содержит в себе преобразования, исполняемые в момент вызова итератора train_loader по train_dataset с размером пакета равным аргументу batch_size.

Перевести в векторную форму необходимо и метки класса. Произвести это можно с помощью унитарного кодирования, преобразовав категориальный признак в вид двоичного вектора: 1 на i-ой позиции означает принадлежность к i-ому классу, 0 - к непринадлежности. На листинге 3.2 представлена реализация преобразования класса в векторное представление.

Листинг 3.2 - реализация вектроризации меток класса

```
class OneHotEncoder():
    def __init__(self, num_classes=10):
        self.num_classes = num_classes
        self.classes = list(range(num_classes))
        self.mapping = {}
        for i in range(self.num_classes):
            self.mapping[self.classes[i]] = i

    def __call__(self, label):
        vec = np.zeros(self.num_classes, dtype=int)
        vec[self.mapping[label]] = 1
        return vec

target_transoforms = torchvision.transforms.Compose([
    OneHotEncoder(NUM_CLASSES),
])
```

Размер пакета обозначим равным 64, стандартным для подобных задач с учетом вычислительных способностей компьютера.

4. Тестирование методов оптимизации

Для репрезентативного тестирования необходимо провести серию экспериментов с различными гиперпараметрами, которыми могут быть, например, количество слоев нейронной сети, количество параметров и скорость обучения. Оцениваться будет качество модели по метрике accuracy, по значению функции потерь MSE, по скорости сходимости и потреблению памяти.

4.1 Постановка экспериментов

Проверять несколько архитектур критически важно, поскольку от этого зависит количество обучаемых параметров модели.

Рассматриваемые архитектуры изображены на таблицах 4.1-4.3:

Таблица 4.1 - архитектура первой модели

Слой, номер	Тип	Количество нейронов
1	Полносвязный (вход)	784x100
2	Активация	0
3	Полносвязный (выход)	100x10

Таблица 4.2 - архитектура второй модели

Слой, номер	Тип	Количество нейронов
1	Полносвязный (вход)	784x100
2	Активация	0
3	Полносвязный	100x10
4	Активация	0
5	Полносвязный (выход)	100x10

Таблица 4.3 - архитектура третьей модели

Слой, номер	Тип	Количество нейронов
1	Полносвязный (вход)	784x256
2	Активация	0
3	Полносвязный	256x128
4	Активация	0
5	Полносвязный	128x64

Продолжение таблицы 4.3.

6	Активация	0
7	Полносвязный	64x10

Рассмотренные три архитектуры имеют вместимость (количество параметров) равную 79510, 89610 и 242762 соотвественно. Расчет параметров производится с помощью функцию, представленной на листинге 4.1

Листинг 4.1 - функция расчёта параметров модели

```
def calc_params(model):
    # учитываем только полносвязные слои
    layers = [l for l in model.layers if type(l) == Linear]
    params = 0
    # обход слоев
    for layer in layers:
        # получаем кол-во входных и выходных нейронов
        in_p, out_p = layer.w.shape
        # перемножаем in_p и out_p, получая размер матрицы w
        # out_p - размер порога b
        params += in_p * out_p + out_p
    return params
```

4.2 Анализ экспериментов

Результаты тестирования первой модели в процессе обучения при скорости обучения равной 0.001 на наборе данных MNIST изображены на рисунке 4.1 и представлены на таблице 4.3:

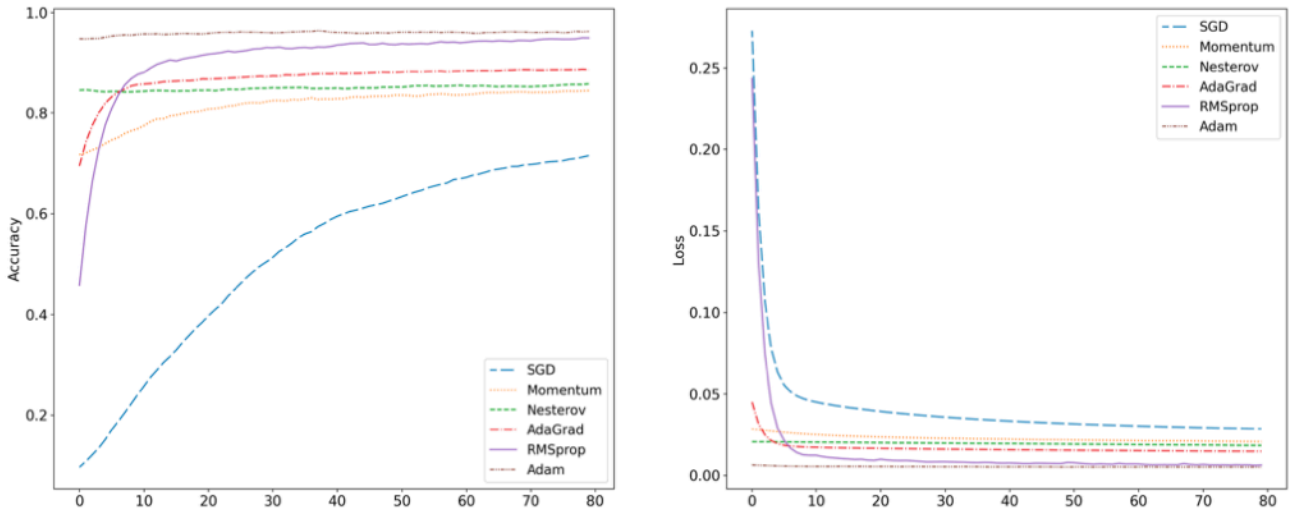

Рисунок 4.1 - Результаты обучения первой модели при скорости 0.001

Таблица 4.3 - Результаты обучения первой модели при скорости 0.001 на MNIST

Оптимизатор	среднее время на шаг, мс	максимальная точность	минимальная потеря	потребление памяти, байты
SGD	0.088	0.722656	0.028505	28
Momentum	0.156	0.846875	0.020697	636528
Nesterov	3.940	0.861328	0.018413	1273056
AdaGrad	0.241	0.889648	0.014808	636528
RMSprop	0.292	0.954883	0.005910	636528
Adam	0.820	0.967578	0.004994	1273056

По результатам эксперимента, лучшие результаты в данной конфигурации показывает Adam. Ожидаемо, максимальная точность алгоритмов совпадает с порядком их открытия и освещения в данной курсовой.

Со значительным отрывом по скорости проигрывает Nesterov, поскольку совершать два шага вместо одного дорогая вычислительная операция. SGD ожидаемо является самым быстрым алгоритмом. Адаптивные алгоритмы AdaGrad и RMSprop затрачивают примерно одинаковое время на шаг, но Adam в два раза медленнее последнего из-за дополнительных вычислений экспоненты в степени текущей эпохи.

RMSprop достаточно близок по точности к Adam, при этом требует в два раза меньше памяти. Алгоритм SGD не требует хранения дополнительных массивов, поэтому хранится лишь одна переменная - скорость обучения. Алгоритмы Momentum, AdaGrad и RMSprop хранят один массив импульсов каждого параметра, Nesterov хранит копию модели помимо импульсов, Adam хранит два массива импульсов разных порядков. Потребление памяти во всех экспериментах будет расти линейно с коэффициентом равным единице по отношению к росту потребления памяти модели.

Скорость обучения - важный гиперпараметр, который может сильно влиять не только на скорость сходимости, но и на качество модели и даже факт сходимости в целом. При выборе больших значений, обычно, оптимизатор либо показывает посредственные результаты, либо расходится; при выборе

маленьких - необходимое количество шагов до сходимости устремляется к бесконечности. Результаты точности обучения первой модели в зависимости от скорости обучения представлены на таблице 4.4.

Таблица 4.4 - Результаты обучения первой модели с различными скоростями обучения, метрика accuracy

Оптимизатор	γ=0.001	γ=0.01	γ=0.1	γ=1
SGD	0.722656	0.963672	0.938086	-
Momentum	0.846875	0.967578	0.825195	-
Nesterov	0.861328	0.967578	0.775586	-
AdaGrad	0.889648	0.970117	-	-
RMSprop	0.954883	0.920508	-	-
Adam	0.967578	0.943164	-	-

Пропуски в таблице означают то, что оптимизатор не сошелся и стал предсказывать случайный класс. Классический SGD и инерционные методы Momentum и Nesterov показывают себя лучше, чем адаптивные методы RMSprop и Adam при больших γ, но хуже AdaGrad. Это объясняется тем, что адаптивные методы имеют собственные механизмы увеличения или уменьшения шага обучения, однако AdaGrad способствует ее монотонному уменьшению, что позволяет стабилизировать обучения и позволяет выбирать γ с запасом.

Необходимо убедится, что оптимизаторы способны к масштабированию и оптимизации более сложных архитектур. На рисунке 4.2 изображены результаты обучения второй модели при γ равным 0.001. Количество параметров в ней 89610, что в 1.3 раза больше по сравнению с первой моделью. Также, чередующиеся применение слоев активации, в частности сигмоидальных функций, приводит к затуханию градиента для последующих слоев во время обратного распространения ошибки. Адаптивные методы стараются подстроится под эту особенность. На рисунке 4.2 изображены результаты обучения второй модели при скорости обучения равной 0.001.

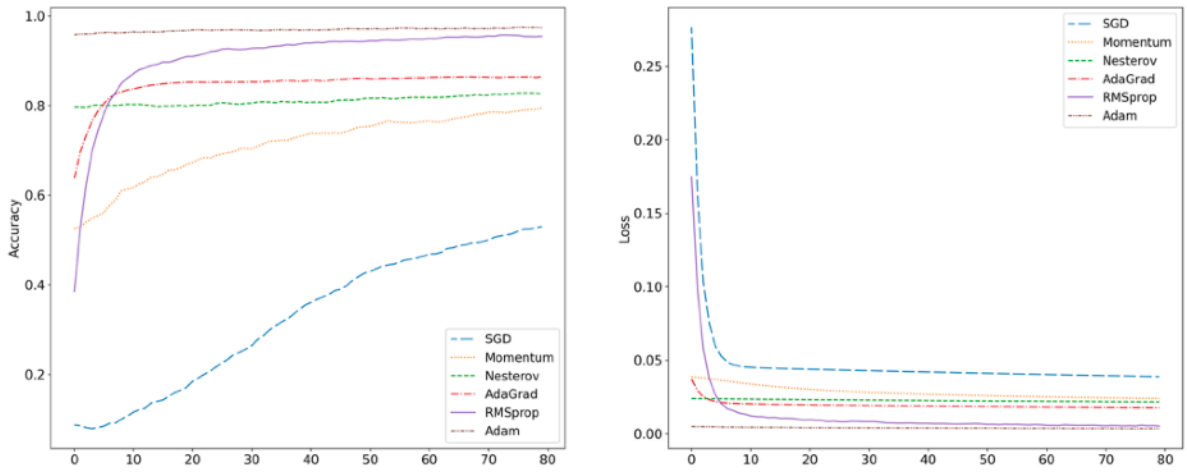

Рисунок 4.2 - Результаты обучения второй модели при скорости 0.001

Относительные положения оптимизаторов сохраняются, однако лишь RMSprop и Adam сохранили высокое качество распознавания и скорость сходимости.

На рисунке 4.3 изображены результаты обучения третьей модели при γ равным 0.001. Количество параметров в ней 242762, что в 3 раза больше по сравнению с первой моделью и в 3 раза больше скрытых слоев.

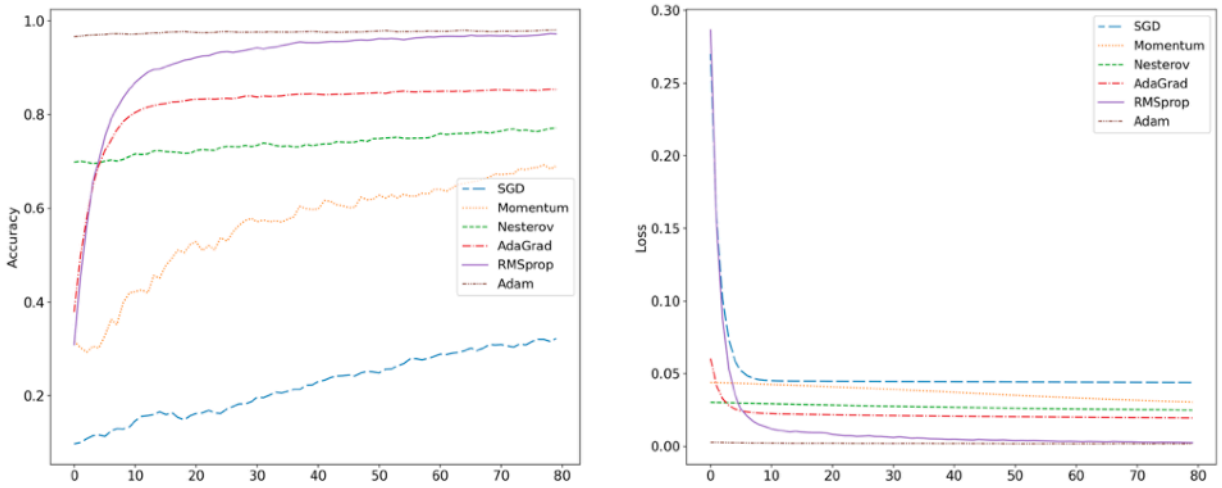

Рисунок 4.3 - Результаты обучения третьей модели при скорости 0.001

Тенденция, намеченная при обучение первой и второй модели сохраняется. RMSprop и Adam по прежнему показывают достойные результаты, а вот классический SGD и инерционные методы явно отстают как по функции потерь, так и по метрике accuracy даже в сравнение с легковесными моделями. Такие образом, неадаптивные методы оказались чувствительны к вместимости

модели. На таблице 4.5 представлены лучшие показатели метрики accuracy в зависимости от модели по всем оптимизаторам среди всех значений скорости обучения.

Таблица 4.5 - лучшие показатели метрики accuracy в зависимости от архитектуры и оптимизатора.

Оптимизатор	первая модель	вторая модель	третья модель
SGD	0.963672	0.978906	0.963672
Momentum	0.967578	0.977930	0.979883
Nesterov	0.967578	0.978906	0.980859
AdaGrad	0.970117	0.980859	**0.982812**
RMSprop	0.954883	0.964258	0.976953
Adam	0.967578	0.976953	0.980859

Таким образом, при переборе различных скоростей обучения, AdaGrad выдает лучший показать метрики accuracy - 98.3%. Адаптивные методы в среднем показывают результаты лучше по сравнению с классическими и инерционными.

Важная деталь, что увеличение вместимости модели улучшает результат, поэтому чаще всего именно архитектура является ключевым гиперпараметром при решение задачи с помощью искусственных нейронных сетей.

Осталось проверить работоспособность на более сложном домене. Зафиксируем скорость обучения равной 0.01 и архитектуру третьей модели. Результаты обучения представлены на таблице 4.6 и рисунке 4.4.

Таблица 4.6 - Результаты обучения третьей модели при скорости 0.01 на Fashion MNIST

Оптимизатор	среднее время на шаг, мс	максимальная точность	минимальная потеря
SGD	0.000189	0.119141	0.045077
Momentum	0.000352	0.118164	0.044989

Продолжение таблицы 4.6.

Nesterov	0.008778	0.118164	0.044984
AdaGrad	0.000560	0.503125	0.032790
RMSprop	0.000692	0.808984	0.013851
Adam	0.002131	0.864258	0.010202

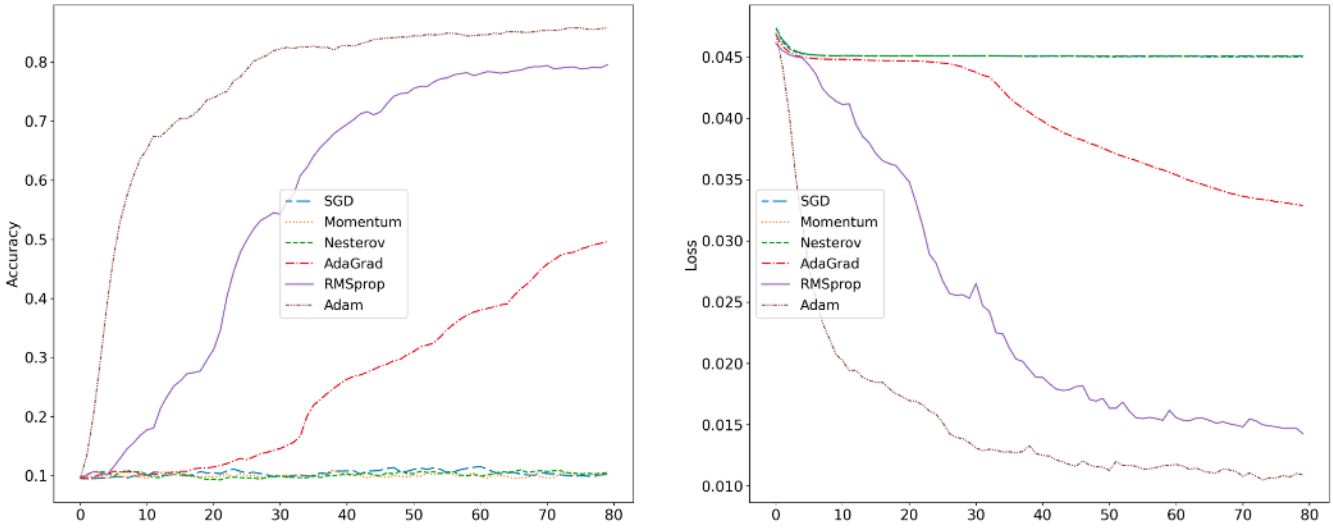

Рисунок 4.4 - Результаты обучения третьей модели при скорости 0.01 на Fashion MNIST

На более сложном домене разница между оптимизаторами становится кардинально больше. Более того, ни один из неадаптивных методом не сошёлся застрял на раннем плато, показав качество предсказания чуть лучшее, чем случайное предсказание. Между тем, Adam показывает достойные результаты как по показателю функции потерь, так и по метрике accuracy.

ЗАКЛЮЧЕНИЕ

В результате выполнения данной курсовой работы были рассмотрены основы оптимизации и, в частности, методы оптимизации, применяемые в задаче обучения нейронной сети. Также были изучены основы работы и обучения искусственных нейронных сетей.

На основе изученной информации, были разработаны стохастические методы оптимизации, основанные на градиентном спуске. Также были разработаны классы нейросетевых слоев, слоев активации, целевые функции и цикл обучения, поддерживающие взаимодействия с разработанными методами оптимизации.

Были проведены множественные эксперименты с различными гиперпараметрами и реализованы метрики качества полученных моделей. С значительным отрывом, лучше всего себя показывают адаптивные методы оптимизации.

С увеличением популярности нейронный сетей, все чаще встает вопрос об ускорении их обучения. В дальнейшей, стоит продумать использование графических ускорителей или квантовых компьютеров для более эффективных матричных вычислений, поскольку весь процесс обучения модели упирается в прямой и обратных проход данных, представляющих собой последовательное перемножения матриц больших размерностей. Несмотря на использование процессора, слабо оптимизированного под эту задачу, удалось достичь хороших результатов как по времени работы, так и по качеству классификации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. SGD Generalizes Better Than GD (And Regularization Doesn't Help) - URL: https://arxiv.org/abs/2102.01117 (дата обращения: 20.12.2022)
2. Accelerating SGD with momentum for over-parameterized learning - URL: https://arxiv.org/abs/1810.13395 (дата обращения: 20.12.2022)
3. Nesterov's Accelerated Gradient and Momentum as approximations to Regularised Update Descent - URL: https://arxiv.org/abs/1607.01981 (дата обращения: 20.12.2022)
4. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization - URL https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf (дата обращения: 20.12.2022)
5. RMSProp and equilibrated adaptive learning rates for non-convex optimization - URL: https://dblp.uni-trier.de/db/journals/corr/corr1502.html#DauphinVCB15 (дата обращения: 20.12.2022)
6. Adam: A Method for Stochastic Optimization - URL: https://arxiv.org/abs/1412.6980 (дата обращения: 20.12.2022)
7. Documentation Python. - URL: https://docs.python.org/3/ (дата обращения: 20.12.2022)
8. Documentation Jupyter. - URL: https://docs.jupyter.org/en/latest/ (дата обращения: 20.12.2022)
9. Documentation NumPy. - URL: https://numpy.org/doc/stable/ (дата обращения: 20.12.2022)
10. Documentation Matplotlib. - URL: https://matplotlib.org/stable/index.html (дата обращения: 20.12.2022)
11. Documentation PyTorch. - URL: https://pytorch.org/docs/stable/index.html (дата обращения: 20.12.2022)
12. Documentation MNIST. - URL: http://yann.lecun.com/exdb/mnist/ (дата обращения: 20.12.2022)