ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ
РОССИЙСКОЙ ФЕДЕРАЦИИ»
(ФИНАНСОВЫЙ УНИВЕРСИТЕТ)**

Департамент анализа данных
и машинного обучения

Дисциплина: «Технологии анализа данных и машинного обучения»
Направление подготовки: «Прикладная математика и информатика»
Профиль: «Анализ данных и принятие решений в экономике и финансах»
Факультет информационных технологий и анализа больших данных
Форма обучения очная
Учебный 2022/2023 год, 6 семестр

Курсовая работа на тему:
«Анализ параметров выходных нейронов глубоких нейронных сетей»

Выполнил:

студент группы ПМ20-4

Козлов М. В.

Научный руководитель:

доцент, к.т.н., Куликов А. А.

Москва 2023

Оглавление

Введение

С развитием глубокого обучения и нейронных сетей стало ясно, что выбор и настройка параметров играют решающую роль в эффективности и производительности моделей. В контексте сложности современных задач и объемов данных, оптимальный подбор параметров нейронной сети становится ключевым шагом для достижения высокой точности и обобщения на новые данные.

Курсовая работа посвящена исследованию методов и стратегий подбора параметров нейронных сетей, включая архитектурные решения, функции активации, скорость обучения, методы оптимизации и другие ключевые аспекты. Анализ и эксперименты в этой области помогут понять влияние каждого параметра на процесс обучения и качество модели, а также разработать практические рекомендации для эффективного подбора параметров в реальных приложениях.

Тема исследования “Анализ параметров выходных нейронов глубоких нейронных сетей” будет раскрыта с точки зрения оптимизации выходных нейронов (их максимизации) с целью получения лучшей точности нейронной сети.

Экскурс в историю глубоких нейронный сетей

Искусственный интеллект (ИИ, англ. *AI*) – это область компьютерных наук, стремящаяся создать алгоритмы и системы, способные анализировать данные и выполнять задачи, требующие человеческого интеллекта. Одной из ключевых технологий в этой области стали глубокие нейронные сети, которые стали настоящим прорывом и позволили достичь впечатляющих результатов в областях, требующих анализа и обработки больших объемов данных.

Перед появлением глубоких нейронных сетей стояли различные попытки создать модели, способные анализировать данные. Начиная с 1940-х годов, были предложены различные архитектуры, но они сталкивались с ограничениями вычислительной мощности и недостаточными объемами данных. В 1960-х появилась идея персептрона – простой модели, имитирующей нейроны в мозге, но ограниченной в своей способности обучения сложным функциям.
Прорыв произошел в 1980-х годах, когда ученые разработали алгоритм обратного распространения ошибки (англ. *backpropagation*), позволяющий эффективно обучать нейронные сети с несколькими слоями. Это открыло путь к созданию глубоких нейронных сетей, состоящих из множества слоев, каждый из которых выполняет определенные операции над данными.

Глубокие нейронные сети (ГНС, англ. *Deep Learning Network, DLN*) представляют собой многократно сложенные архитектуры, вдохновленные структурой и функционированием человеческого мозга. Вместо явного программирования, ГНС способны извлекать закономерности непосредственно из данных, что делает их универсальным инструментом для решения разнообразных задач: от обработки изображений до обнаружения мошенничества.
Одной из важнейших сфер, где ГНС стали настоящим прорывом, является компьютерное зрение. Способность нейронных сетей распознавать объекты на изображениях сравнима с человеческими возможностями. Это привело к развитию автоматической машинной классификации, а также к применению в медицинских исследованиях для диагностики заболеваний по медицинским изображениям.
ГНС также нашли применение в обработке естественного языка. Они способны анализировать тексты, выделять смысловые аспекты и даже создавать свои оригинальные тексты. Это привело к созданию чат-ботов, переводчиков, анализаторов настроения и многих других инструментов для работы с текстовой информацией.

Глубокие нейронные сети – это важный шаг в развитии искусственного интеллекта. Они обогатили нашу способность анализировать данные, решать сложные задачи и автоматизировать процессы, ранее требовавшие участия человека. С появлением более мощных вычислительных ресурсов и расширением объемов данных, глубокие нейронные сети будут продолжать эволюционировать, открывая новые горизонты в области искусственного интеллекта и технологий.

Обзор архитектуры простейшей глубокой нейронной сети

Нейронные сети – это компьютерные модели, которые пытаются имитировать работу человеческого мозга, позволяя компьютерам обучаться и принимать решения на основе данных. Одной из фундаментальных архитектур в мире нейронных сетей является персептрон, который лежит в основе более сложных глубоких нейронных сетей. Персептрон может быть использован для решения задач классификации, регрессии и даже аппроксимации сложных функций. Он был первым шагом в развитии нейронных сетей и служит основой для более сложных архитектур, таких как многослойные персептроны и сверточные нейронные сети.
Архитектура персептрона представляет собой ключевую структуру в мире нейронных сетей. Понимание её компонентов и принципов работы является важным шагом в изучении более сложных моделей глубокого обучения. Персептрон – это не только исторический момент в развитии ИИ, но и основа, на которой строятся многие современные инновации в области машинного обучения и искусственного интеллекта.

Персептрон состоит из нескольких ключевых компонентов, каждый из которых играет важную роль в его функционировании. Рассмотрим каждый из них.

Входные данные

На начальном этапе данные поступают в нейронную сеть в виде вектора. Для простоты представим, что это могут быть пиксели черно-белого изображения, каждый из которых представлен целым числом в диапазоне $[0; 255]$, отображающим интенсивность цвета. Для удобства дальнейшего вычисления мы нормируем их в диапазон от $[0; 1]$. Эти входные данные формируют входной вектор $x = [x_1, x_2, \ldots, x_n]$, где n – количество входных признаков, в нашем случае – пикселей изображения.

Веса

Каждому входному признаку x_i соответствует вес w_i. Веса представляют собой числовые значения, определяющие важность каждого признака для выхода нейрона. Веса объединяются в вектор весов $w = [w_1, w_2, \ldots, w_n]$, который будет использоваться для вычисления суммы взвешенных входных данных.

Выбор начальных значений параметров для слоев, составляющих модель, является ключевым принципом. Установка всех параметров в ноль может серьезно затруднить процесс обучения, так как ни один параметр не будет активирован изначально. Присвоение параметрам значений в диапазоне $[-1; 1]$ также не всегда является оптимальным подходом. В некоторых случаях правильная инициализация сети может существенно повлиять на её способность к обучению, даже влияя на достижение

наивысшей производительности или на успешную сходимость. Даже если задача не требует настолько строгих условий, грамотный выбор метода инициализации начальных параметров оказывает значительное влияние на способность модели к обучению, так как это устанавливает параметры модели с учетом целевой функции.

Хотя всегда можно начать с случайных начальных значений, предпочтительно выбирать специальные методы инициализации. Ниже приведены наиболее распространенные из них:

Метод инициализации Ксавьера (метод Глорота)

Центральной концепцией данного метода является упрощение передачи сигнала через слой как в процессе прямого распространения, так и в обратном распространении ошибки, особенно для линейной функции активации. Этот подход также успешно применяется к сигмоидной функции, поскольку участок её графика, где не происходит насыщение, также обладает линейными свойствами.

При вычислении параметров этот метод опирается на вероятностное распределение (равномерное или нормальное) с дисперсией, равной:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

Где n_{in} – количество входных нейронов, а n_{out} – количество выходных нейронов.

Метод инициализации Ге

Вариация метода Ксавьера, больше подходящая функции активации ReLU, компенсирующая тот факт, что эта функция возвращает нуль для половины области определения. А именно, в этом случае:

$$Var(W) = \frac{2}{n_{in}}$$

Где n_{in} – количество входных нейронов.

Смещение

Смещение (англ. *bias*) – это дополнительный параметр в каждом нейроне, который добавляется к взвешенной сумме входных данных перед применением активационной функции, обозначается литерой b. Смещение позволяет нейрону более гибко настраиваться на данные и повышает его способность выявления сложных паттернов. Смещение добавляется к взвешенной сумме входных данных, и результат передается в активационную функцию.

Смещение играет ключевую роль в обучении нейронных сетей. Без смещения все нейроны в слое имели бы одинаковый весовой вклад в сумматоре, что ограничило бы способность нейронной сети к аппроксимации разнообразных функций.

Сумматор

Сумматор выполняет линейную комбинацию входных данных и их весов, а также прибавляет смещение к итоговому значению, что в итоге дает взвешенную сумму. Выход сумматора передается далее в активационную функцию, которая определит, будет ли нейрон активирован (выдаст выход) или нет.

$$z = \sum_{i=1}^{n} (w_i \cdot x_i) + b$$

Где w_i – вес, присвоенный i-му входному признаку, x_i – i-й входной признак, b – параметр сдвига, n – количество входных нейронов (признаков), z – выход сумматора (взвешенная сумма входных данных).

Функция активации

Активационная функция определяет, будет ли нейрон активирован и какой будет его выход. Простейшей активационной функцией является пороговая функция, которая возвращает 1, если входная сумма превышает определенный порог, и 0 в противном случае.
Более сложные активационные функции, такие как сигмоида или ReLU, позволяют модели улавливать нелинейные зависимости в данных.
Рассмотрим наиболее популярные функции активации.

Ступенчатая функция

Пороговая функция активации, известная также как ступенчатая функция (англ. *binary step function*), представляет собой активационную функцию, которая активирует нейрон, когда сумма входных значений превышает или падает ниже определенного порогового значения. Такой подход хорошо подходит для задач бинарной классификации. Однако этот тип функции не справляется с ситуациями, когда требуется использование более чем двух нейронов для классификации и имеется больше чем два класса.

$$f(z) = 1,\ z \geq 0.5$$
$$f(z) = 0,\ z < 0.5$$

Линейная функция

Линейная функция (англ. *linear function*) представляет собой прямую линию, то есть

$$f(z) = c \cdot \mathrm{z}$$

Это означает, что результат этой функции пропорционален её входному аргументу. В отличие от предыдущей функции, она дает выходные значения в диапазоне, а не ограничивается только бинарными 0 и 1, что решает проблему классификации с множеством классов.

Тем не менее, у линейной функции активации есть две основные недостатки:

1) Она несовместима с методом обратного распространения ошибки. В этом методе используется градиентный спуск, который требует вычисления производной, но для линейной функции активации производная постоянна и не зависит от входных значений. Это означает, что при обновлении весов невозможно определить, улучшается ли эмпирический риск на данном шаге.

2) Рассмотрим многослойную нейронную сеть с этой функцией активации. Поскольку каждый слой даёт линейный выход, все слои в совокупности образуют линейную комбинацию, что приводит к получению на выходе линейной функции. Таким образом, финальная активация на последнем слое зависит лишь от входных значений первого слоя. В результате любое количество слоев можно заменить одним, что делает многослойную архитектуру бессмысленной.

Основное различие линейной функции активации от остальных заключается в том, что её область значений не ограничена: $(-\infty;\ +\infty)$. Поэтому линейную функцию следует использовать, когда требуется выходное значение нейрона в непрерывном диапазоне $\in R$, а не в ограниченном интервале.

Сигмоидная функция

Сигмоидная функция (англ. *sigmoid function*), которую также называет логистической (англ. *logistic function*), является гладкой монотонно возрастающей нелинейной функцией:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Поскольку эта функция является нелинейной, она подходит для использования в глубоких нейронных сетях с множеством слоев, и её можно обучать методом обратного распространения ошибки. Сигмоида ограничивается двумя горизонтальными асимптотами, $y = 1$ и $y = 0$, что обеспечивает нормализацию выходных значений каждого нейрона.

Кроме того, для сигмоидной функции характерен плавный градиент, который предотвращает "скачки" при вычислении выходных значений. Еще одним преимуществом этой функции является то, что при значениях $x > 2$ и $x <- 2, y$ "прижимается" к одной из асимптот, что способствует чётким предсказаниям классов.

Не смотря на множество сильных сторон сигмоидной функции, у нее есть существенный недостаток. Производная этой функции крайне мала практически во всех точках, кроме небольшого диапазона. Это затрудняет процесс обновления весов с использованием градиентного спуска. Эта проблема становится более серьёзной в случае глубоких моделей. Эта проблема называется проблемой затухающего градиента.

Что касается использования сигмоидной функции, ее преимущество перед другими функциями заключается в способности нормализовать выходные значения. Иногда это особенно важно. Например, когда конечное значение слоя должно представлять вероятность случайной величины. Кроме того, данную функцию удобно применять в задачах классификации, благодаря свойству "прижимания" к асимптотам.

Функция гиперболического тангенса

Функция гиперболического тангенса (англ. *hyperbolic tangent*) имеет вид:

$$tanh(z) = \frac{2}{1+e^{-2z}} - 1$$

Эта функция является скорректированной сигмоидной функцией

$$tanh(z) = 2 \cdot sigma(2z) - 1$$

То есть она сохраняет те же преимущества и недостатки, но уже для диапазона значений $(-1; 1)$.

Обычно, гиперболический тангенс ($tanh$) предпочтителен по сравнению со сигмоидной в ситуациях, когда не требуется нормализация. Это объясняется тем, что область определения этой функции активации центрирована вокруг нуля, что убирает ограничение на градиент для движения в определенном направлении.

Кроме того, производная гиперболического тангенса значительно больше вблизи нуля, обеспечивая более сильный градиент для градиентного спуска и, следовательно, более быструю сходимость.

ReLU

ReLU (англ. *Rectified Linear Unit*) – это наиболее часто используемая функция активации при глубоком обучении. Данная функция возвращает 0, если принимает отрицательный аргумент, в случае же положительного аргумента, функция возвращает само число. То есть она может быть записана как:

$$f(z) = max(0, z)$$

Сначала может показаться, что функция активации ReLU является линейной и сталкивается с теми же проблемами, что и линейная функция. Однако это

неправильное представление, и ReLU может успешно использоваться в глубоких нейронных сетях.

Функция ReLU обладает несколькими преимуществами перед сигмоидой и гиперболическим тангенсом:

1) Ее производная вычисляется очень быстро и просто. Для отрицательных значений производная равна 0, а для положительных – 1.

2) ReLU способствует разреженной активации. В сетях с большим числом нейронов использование сигмоидной функции или гиперболического тангенса как функции активации приводит к активации почти всех нейронов, что может замедлить процесс обучения. В случае с ReLU количество активированных нейронов сокращается из-за особенностей функции, что улучшает обучение сети.

Однако у функции активации ReLU есть один недостаток, известный как проблема «умирающего» ReLU. Поскольку некоторая часть производной функции равна нулю, градиент также будет нулевым. Это означает, что веса не обновляются во время обучения, и нейронная сеть прекращает обучение.

Функцию активации ReLU следует использовать, если не требуется конкретное ограничение выходных значений нейронов, так как она не имеет ограниченной области определения. Если после обучения модели результаты неудовлетворительны, стоит рассмотреть другие функции активации, которые могут дать более лучшие результаты.

Взглянем на визуализацию функций активации (см. Приложение №1).

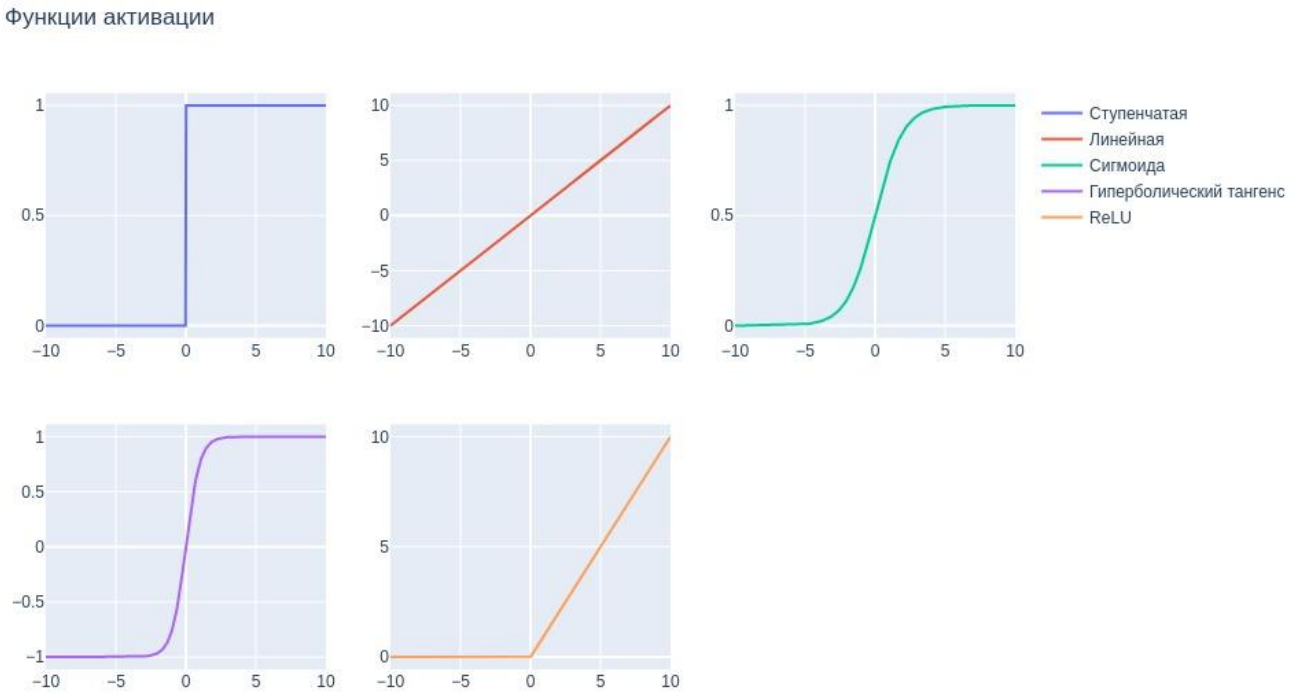

Рис. 1 — Сравнение функций активации

Функция ошибки

Функция ошибки (или функция потерь) – это математическая мера, которая оценивает разницу между прогнозами модели и фактическими значениями (или целевыми значениями). Она играет ключевую роль в процессе обучения нейронных сетей, так как помогает определить, насколько хорошо модель выполняет задачу и как нужно корректировать веса нейронов для улучшения ее производительности.

Процесс обучения нейронной сети заключается в нахождении значений весов, при которых функция ошибки минимизируется. Это достигается с использованием методов оптимизации, таких как градиентный спуск.
Вот несколько распространенных функций ошибки и их применение:

Среднеквадратичная ошибка (англ. *Mean Squared Error, MSE*)

Эта функция используется часто в задачах регрессии. Она измеряет среднее квадратов разницы между прогнозами и фактическими значениями. Формула для MSE:

$$MSE = \frac{1}{n}\Sigma(target - output)^2$$

Где n – количество примеров в обучающем наборе, $target$ – целевые значения, $output$ – выходы модели.

Категориальная кросс-энтропия (англ. *Categorical Cross-Entropy, CE*)

Эта функция используется в задачах классификации с несколькими классами. Она измеряет разницу между прогнозами и верными классами, учитывая вероятности. Формула для категориальной кросс-энтропии:

$$CE = -\ \Sigma(target \cdot log(output))$$

Где $target$ – вероятности верных классов, $output$ – прогнозы модели.

Логистическая (бинарная) кросс-энтропия (англ. *Binary Cross-Entropy, BCE*)

Эта функция используется в бинарной классификации (когда есть только два класса). Она аналогична категориальной кросс-энтропии, но учитывает только два класса. Формула для логистической кросс-энтропии:

$$BCE = -\ \Sigma(target \cdot log(output) + (1 - target) \cdot log(1 - output))$$

Где $target$ – целевые бинарные значения, $output$ – прогнозы модели.

Выбор функции ошибки зависит от типа задачи (регрессия, бинарная или многоклассовая классификация) и характера выходных данных. Цель обучения – минимизировать значение функции ошибки путем коррекции весов нейронов с помощью методов оптимизации, таких как градиентный спуск.

Обратное распространение ошибки

Обратное распространение ошибки (англ. *backpropagation*) – это ключевой алгоритм, используемый для обучения нейронных сетей, позволяющий корректировать веса нейронов на основе расчета градиентов функции ошибки по этим весам. Этот алгоритм позволяет нейронным сетям находить оптимальные веса для минимизации ошибки прогнозов.
Процесс обратного распространения ошибки включает несколько этапов:

Прямое распространение (англ. *forward propagation*)
На этом этапе входные данные проходят через нейронную сеть от входного слоя к выходному. Каждый нейрон вычисляет свою взвешенную сумму входов, применяет активационную функцию и передает выход далее по сети.

Вычисление ошибки
После прямого распространения сеть делает прогнозы, и эти прогнозы сравниваются с фактическими значениями (целевыми значениями). Разница между прогнозами и целевыми значениями измеряется с помощью функции ошибки.

Обратное распространение ошибки

На этом этапе градиент функции ошибки вычисляется по всем весам сети с помощью правила цепной дифференциации. Градиент указывает направление и величину изменения ошибки относительно изменения весов. Этот этап начинается с вычисления градиента на выходном слое и последовательно распространяется назад к входному слою.

Обновление весов
Полученные градиенты используются для корректировки весов нейронов. Веса обновляются в направлении, противоположном градиенту, с учетом скорости обучения (англ. *learning rate*), которая регулирует величину изменения весов на каждом шаге.

Повторение
Процесс прямого и обратного распространения ошибки повторяется на множестве тренировочных примеров (эпохах), пока значение функции ошибки не достигнет минимума или пока не будут выполнены другие критерии останова.

Обратное распространение ошибки позволяет нейронным сетям адаптироваться к данным и находить оптимальные веса для достижения минимизации ошибки. Однако важно учитывать, что этот алгоритм может столкнуться с проблемой затухания или взрыва градиентов в глубоких сетях. Поэтому часто используются методы оптимизации, такие как различные вариации стохастического градиентного спуска (англ. *SGD*) и адаптивные скорости обучения, чтобы облегчить обучение.

Глубокая нейронная сеть для датасета MNIST

MNIST – это один из самых известных и широко используемых наборов данных в области машинного обучения и компьютерного зрения. MNIST представляет собой коллекцию изображений рукописных цифр, созданных на основе написанных людьми цифр от 0 до 9.
Этот датасет стал своеобразной "пробной площадкой" для многих исследований и алгоритмов, так как позволяет проверить и сравнить эффективность различных методов классификации и распознавания образов. Каждое изображение состоит из 28x28 пикселей, что в сумме даёт 784 пикселя на изображение.
Для исследования подбора параметров нейронной сети обратимся к выборке изображений, составляющих класс рукописных цифр. В нём 10 классов, отображающих каждую цифру от 0 до 9 включительно.

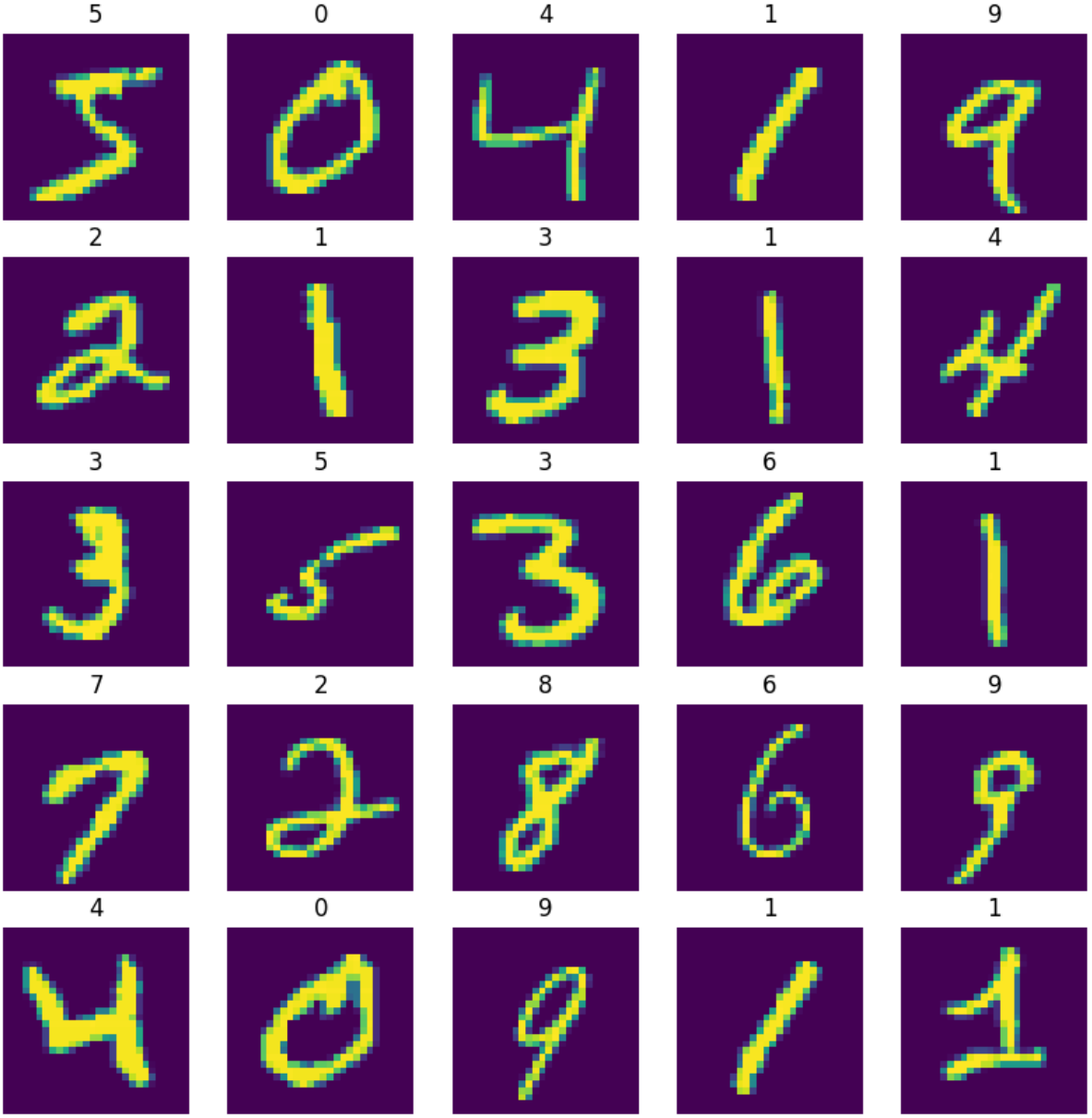

Рис. 2 — Пример изображений из датасета MNIST

Для набора данных MNIST, который включает в себя рукописные цифры, подойдут различные архитектуры нейронных сетей. Ниже перечислены некоторые архитектуры, которые могут быть эффективными для обработки данных MNIST:

1. Нейронная сеть с прямой связью (*FNN*)
FNN (англ. *Feedforward Neural Network*) – это класс нейронных сетей, также известных как нейронные сети прямого распространения. Это наиболее базовый тип нейронных сетей, в которых информация передается только в одном направлении, от входных узлов к выходным, без обратных связей или циклов. Сети FNN состоят из трех типов слоев: входного слоя, скрытых слоев и выходного слоя.
Хотя сети FNN могут быть простыми и могут иметь ограниченные возможности для работы с данными, они образуют основу для более сложных архитектур нейронных сетей, таких как сверточные нейронные сети и рекуррентные нейронные сети, которые способны эффективно решать более сложные задачи.

2. Сверточные нейронные сети (*CNN*)
CNN (англ. *Convolutional Neural Network*) показали отличные результаты в задачах распознавания изображений, включая символы. Сверточные слои позволяют модели автоматически извлекать важные признаки из изображений, что делает их подходящими для распознавания разнообразных символов разных стилей написания.

В данном исследовании обратимся к самой наглядной и базовой архитектуре – FNN.

Архитектура FNN

FNN – Feedforward Neural Network, сеть прямого распространения. По своей сути представляет персептрон, структура и принцип работы которого были рассмотрены в предыдущем разделе.

Реализация представлена с использованием библиотеки PyTorch (см. Приложение №2), разработка и запуск кода производились в среде Google Colab.

Определимся с гиперпараметрами (в дальнейшем будем называть параметрами) сети, которые можно менять:

1) количество слоев
2) размер слоев (кроме первого и последнего)
3) функция активации
4) оптимизатор
5) изначальная инициализация весов
6) дропауты
7) размер батча
8) значение скорости обучение (англ. *learning rate*)

Полный список не исчерпывается этими параметрами и во многом зависит от выбранной архитектуры и исследуемой задачи.
Теперь, с целью сделать исследование приемлемого размера, отберем только некоторые самые базовые параметры и начнём их варьировать.
Выберем следующие:

1) размер слоев (в данном случае – единственного скрытого слоя)
2) функция активации
3) оптимизатор
4) размер батча
5) значение скорости обучения

Обсудим вкратце каждый из них.

1. Размер скрытого слоя
   Размер скрытого слоя определяет количество нейронов в слое, которые обрабатывают и преобразуют входные данные. Больший размер скрытого слоя может увеличить способность сети к обучению сложных паттернов, но может также привести к переобучению.
2. Функция активации
   Функция активации применяется к выходу каждого нейрона в нейронной сети. Она вводит нелинейность и позволяет сети обучаться и моделировать сложные отношения в данных. Различные функции активации, такие как ReLU и Sigmoid, могут иметь различные свойства и применяются в зависимости от задачи.
3. Оптимизатор

Оптимизатор отвечает за обновление весов и смещений нейронной сети в процессе обучения. Он использует алгоритмы оптимизации, такие как Adam и SGD, для минимизации функции потерь и улучшения производительности сети. Различные оптимизаторы могут иметь различные свойства и применяются в зависимости от задачи.

4. Размер батча
   Размер батча определяет количество образцов, обрабатываемых в каждой итерации во время обучения. Больший размер батча может ускорить обучение, но требует больше памяти. Размер батча может быть подобран в зависимости от размера обучающей выборки и доступной памяти.
5. Скорость обучения
   Скорость обучения определяет шаг, с которым оптимизатор обновляет веса и смещения нейронной сети. Она контролирует скорость обучения и сходимость к оптимальному решению. Различные значения скорости обучения могут привести к различным результатам, и ее выбор зависит от задачи.

Диапазоны варьирования:

1. Размер слоя: [64, 256, 1028]
2. Функция активации: [ReLU, Sigmoid]
3. Оптимизатор: [SGD, Adam]
4. Размер батча: [32, 64, 128]
5. Скорость обучения: [0.05, 0.10, 0.20]

Общее количество конфигураций $3 \times 2 \times 2 \times 3 \times 3 = 108$.
Разделим датасет MNIST на две выборки. В train_dataset будет 60 тысяч изображений, ее будем использовать для обучения нашей сети. В test_dataset будет 10 тысяч изображений, с помощью которых после обучения мы получим точность предсказания. Обучение будет длиться 5 эпох. Поскольку ресурсы ограничены, сеть несложная, а конфигураций в конечном счёте получится 108, то решено взять небольшое количество эпох. Но даже при таком количестве сеть обучается более чем хорошо.

Запустим цикл обучения. Время обучения 108 конфигураций составило 2 часа 15 минут. Визуализируем распределение точности предсказаний с помощью библиотеки matplotlib.

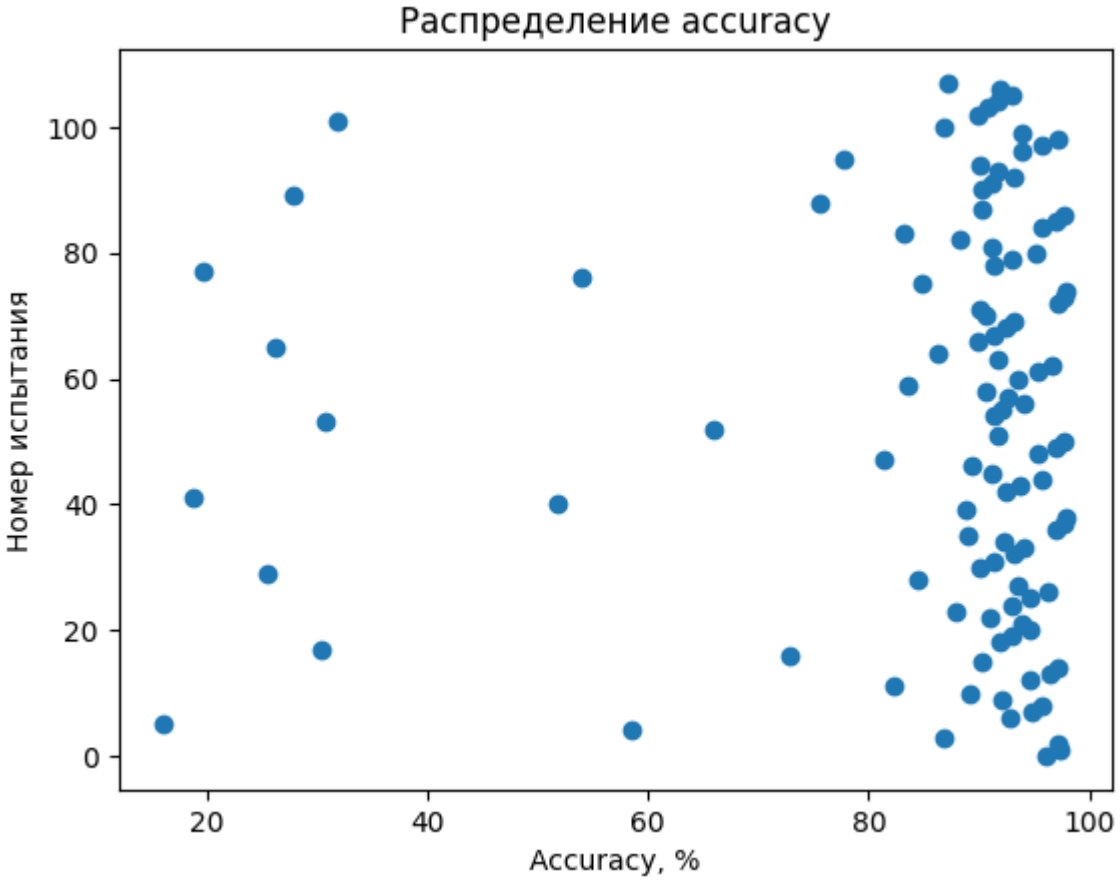

Рис. 3 — Распределение точности предсказания моделей

Как можно заметить, только 16 конфигураций дали результат менее 80% точности.

Отрисуем график гистограммы для тех же данных.

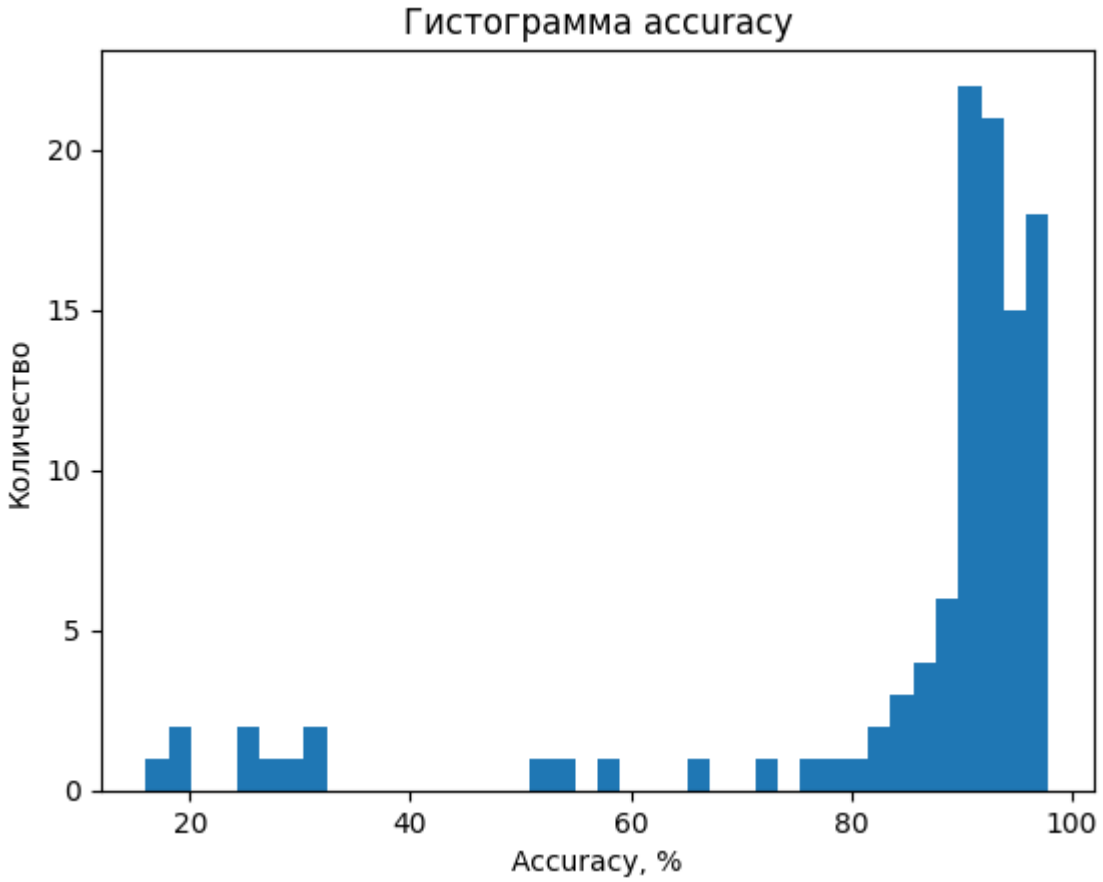

Рис. 4 — Гистограмма точности предсказания моделей

Как можно заметить, здесь еще более наглядно видно, что более 80% конфигураций дали результат более 80% точности предсказаний.

Перейдём к самому главному, сделаем таблицу с отображением точности предсказания для каждой из конфигураций и попытаемся проанализировать данные.

	32											
	ReLU						Sigmoid					
	SGD			Adam			SGD			Adam		
	0.05	0.10	0.20	0.05	0.10	0.20	0.05	0.10	0.20	0.05	0.10	0.20
64	96.00	97.28	97.18	86.84	58.54	16.03	92.87	94.83	95.81	92.14	89.25	82.22
256	96.97	97.76	97.81	88.76	51.73	18.77	92.43	93.73	95.79	91.15	89.36	81.31
1024	97.12	97.71	97.92	84.82	54.00	19.66	91.39	92.94	95.20	91.10	88.19	83.15

Рис. 5 — Результаты для batch_size=32

	64											
	ReLU						Sigmoid					
	SGD			Adam			SGD			Adam		
	0.05	0.10	0.20	0.05	0.10	0.20	0.05	0.10	0.20	0.05	0.10	0.20
64	94.59	96.48	97.15	90.21	72.88	30.28	91.82	93.03	94.66	93.82	91.03	87.99
256	95.35	96.93	97.69	91.66	65.93	30.66	91.31	92.07	94.14	92.67	90.71	83.62
1024	95.75	97.07	97.66	90.32	75.62	27.73	90.24	91.24	93.24	91.73	90.01	77.79

Рис. 6 — Результаты для batch_size=64

	128											
	ReLU						Sigmoid					
	SGD			Adam			SGD			Adam		
	0.05	0.10	0.20	0.05	0.10	0.20	0.05	0.10	0.20	0.05	0.10	0.20
64	92.95	94.65	96.19	93.55	84.54	25.52	90.05	91.43	93.15	94.17	92.21	89.01
256	93.61	95.43	96.68	91.81	86.26	26.24	89.92	91.42	92.47	93.20	90.55	90.15
1024	93.97	95.66	97.12	93.99	86.74	31.73	89.85	90.77	91.73	92.97	91.95	87.14

Рис. 7 — Результаты для batch_size=128

Верхняя линия обозначает размер батча, столбце слева обозначает количество нейронов скрытого слоя. В самой таблице даны значения точности предсказания в процентах с точностью до двух знаков после запятой.

Как нетрудно заметить, комбинация [ReLU; Adam; lr=0.20] дает наихудшие результаты вне зависимости от размера батча или количества нейронов скрытого слоя.

При использовании Sigmoid средняя и максимальная точность падает по мере увеличения размера батча.
Если используется Sigmoid, то выгоднее всего снижать размер батча, использовать SGD и увеличивать lr.

Использование оптимизатора Adam, как показывает таблица, всегда даёт более низкую точность предсказания, нежели оптимизатор SGD.

Повышение числа нейронов при использование Sigmoid влечет снижение качества. При использовании ReLU – наоборот.

При использовании SGD, повышение lr влечет за собой повышение качества. При использовании Adam – наоборот.

При использовании Adam, повышение размера батча влечет за собой повышение точности. При использовании SGD – наоборот.

Комбинация ReLU и Adam является худшей комбинацией, что наглядно показывает нам обилие оранжевого и красного окраса в таблице.

Лучший же средний и максимальный результат получается при [SGD; lr=0.20; ReLU]. Самый же максимальный даёт сеть на 1024 нейрона в скрытом слое, батч размером 32, SGD, ReLU, lr=0.20. Точность предсказания в данном случае равна 97.92%, что является крайне хорошим результатом, особенно беря во внимание малое количество эпох обучения (5).

Получим минимальные и максимальные значения:

	min	*avg*	*max*
64	16.03	85.29	97.28
256	18.77	84.89	97.81
1024	19.66	84.87	97.92

Рис. 8 — Анализ показателей для различных значений количества нейронов в скрытом слое

Заключение

Грамотный подбор параметров нейронной сети и умелое использование их в контексте задачи – это ключевые компоненты для достижения выдающихся результатов в мире искусственного интеллекта и обработки данных.
Именно благодаря качественному подбору параметров представленная нейронная сеть смогла достичь точности предсказания вплоть до 98%.
Если бы мы подбирали параметры наугад, то с вероятность 20% попали бы в худшие конфигурации модели, дающие менее 80% точности.

Важность подбора параметров нейронной сети нельзя недооценивать. Качественно настроенные параметры определяют эффективность обучения и способность модели решать задачи с высокой точностью. От выбора архитектуры сети до определения функций активации, скорости обучения и методов оптимизации – каждый параметр влияет на общую производительность модели. Эмпирический подход и систематический анализ результатов позволяют найти оптимальные настройки для конкретной задачи и набора данных.

Нейронные сети играют важнейшую роль в современной области машинного обучения и искусственного интеллекта. Они преуспели в решении широкого спектра задач, от обработки изображений и текстов до анализа данных и управления роботами. Благодаря своей способности обучаться из данных, нейронные сети способны выявлять сложные паттерны и зависимости в информации, которые не всегда легко заметить человеку. Их гибкость и адаптивность позволяют использовать их в самых разнообразных областях, внося инновации и улучшения в различные аспекты нашей повседневной жизни.

Список использованных источников

Литература

1. Искусственные нейронные сети : учебник / Е. Ю. Бутырский, Н. А. Жукова, В. Б. Мельников [и др.] ; под ред. В. В. Цехановского. — Москва : КноРус, 2023. — 350 с. — ISBN 978-5-406-10678-5. — URL: https://book.ru/book/947113 (дата обращения: 26.08.2023). — Текст : электронный.

Интернет-ресурсы

1. https://neerc.ifmo.ru/wiki/index.php?title=Практики_реализации_нейронных_сетей
2. https://neerc.ifmo.ru/wiki/index.php?title=Инициализация_параметров_глубокой_сети
3. https://neerc.ifmo.ru/wiki/index.php?title=Настройка_глубокой_сети
4. https://habr.com/ru/articles/725182/
5. https://habr.com/ru/articles/724418/

Приложение

Приложение №1. Код “Функции активации.ipynb”

```
!pip install -U numpy plotly kaleido
```

```
import os

import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
path_to_images = "img"

if not os.path.exists(path_to_images):
    os.mkdir(path_to_images)
```

```
def step_function(x):
    return np.where(x >= 0, 1, 0)


def linear_activation(x):
    return x


def sigmoid_function(x):
    return 1 / (1 + np.exp(-x))


def tanh_function(x):
    return np.tanh(x)


def relu_function(x):
    return np.maximum(0, x)
```

```
x_values = np.linspace(-10, 10, 500)

y_step = step_function(x_values)
y_linear = linear_activation(x_values)
y_sigmoid = sigmoid_function(x_values)
y_tanh = tanh_function(x_values)
y_relu = relu_function(x_values)


fig = make_subplots(rows=2, cols=3)
```

```
fig.add_trace(
    go.Scatter(x=x_values, y=y_step, name="Ступенчатая"),
    row=1, col=1
)

fig.add_trace(
    go.Scatter(x=x_values, y=y_linear, name="Линейная"),
    row=1, col=2
)

fig.add_trace(
    go.Scatter(x=x_values, y=y_sigmoid, name="Сигмоида"),
    row=1, col=3
)

fig.add_trace(
    go.Scatter(x=x_values, y=y_tanh, name="Гиперболический тангенс"),
    row=2, col=1
)

fig.add_trace(
    go.Scatter(x=x_values, y=y_relu, name="ReLU"),
    row=2, col=2
)


fig.update_layout(height=600, width=1000, title_text="Функции активации")
fig.show()

fig.write_image(f"{path_to_images}/Функция активации.jpeg")
```

Приложение №2. Код “Модель FNN.ipynb”

```
import time
import json
import random
from string import digits

import numpy as np
import torch
import torchvision
from torchvision import transforms, datasets
from torchvision.transforms import Normalize, ToTensor, Lambda
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
%matplotlib inline
```

```
!mkdir dataset
!ls
```

```
labels = digits
labels
```

```
train_dataset = datasets.MNIST(
    root='./dataset',
    train=True,
    download=True,
    transform=transforms.Compose([ToTensor(), torch.squeeze, Lambda(lambda x:
torch.reshape(x, (-1,)))]),
)

test_dataset = datasets.MNIST(
    root='./dataset',
    train=False,
    download=True,
    transform=transforms.Compose([ToTensor(), torch.squeeze, Lambda(lambda x:
torch.reshape(x, (-1,)))]),
)
```

```
train_len, test_len = len(train_dataset), len(test_dataset)

print(f"Тренировочная выборка: {train_len}")
print(f"Тестовая выборка: {test_len}")
print(f"Суммарное количество изображений: {train_len + test_len}")
```

```
print(f"train_dataset[0] = ({type(train_dataset[0][0])},
{type(train_dataset[0][1])})")
print(f"train_dataset[0] = ({train_dataset[0][0].size()},
{train_dataset[0][1]})")
```

```
row, col = 5, 5
fig, ax = plt.subplots(nrows=row, ncols=col, figsize=(10, 10))
for r in range(row):
    for c in range(col):
        img, index = train_dataset[r*row + c]
        ax[r, c].imshow(img.reshape((28, 28)))
        ax[r, c].set_title(f"{index}")
        ax[r, c].set_axis_off()

plt.show()
```

```
def get_score(
    batch_size,
    num_epoch,
    input_layer,
    hidden_layer,
    output_layer,
    learning_rate,
    loss_fn,
    optimize_fn,
    activation_fn,
    gpu
    ):
    train_dataloader = DataLoader(dataset=train_dataset,
                                  batch_size=batch_size,
                                  shuffle=True)
    test_dataloader = DataLoader(dataset=test_dataset,
                                 batch_size=batch_size,
                                 shuffle=False)

    model = nn.Sequential(
          nn.Linear(input_layer, hidden_layer),
          activation_fn(),
          nn.Linear(hidden_layer, output_layer),
        )

    if gpu and torch.cuda.is_available():
        model = model.cuda()
        loss_fn = loss_fn.cuda()

    optimizer = optimize_fn(model.parameters(), lr=learning_rate)

    model.train()
    for epoch in range(num_epoch):
```

```
        for idx, (X, y) in enumerate(train_dataloader):
            if gpu and torch.cuda.is_available():
                X, y = X.cuda(), y.cuda()

            optimizer.zero_grad()

            pred = model(X)
            loss = loss_fn(pred, y)
            loss.backward()
            optimizer.step()

    all_count = len(test_dataset)
    true_count = 0

    with torch.no_grad():
        model.eval()
        for idx, (X, y) in enumerate(test_dataloader):
            if gpu and torch.cuda.is_available():
                X, y = X.cuda(), y.cuda()
            pred = model(X)
            true_count += (y == torch.argmax(pred, dim=1)).sum()

    percentage = round((true_count / all_count * 100).item(), 3)
    print(f"Процент правильных ответов: {percentage}%")
    return percentage
```

```
hidden_layers = [64, 256, 1024]
batch_sizes = [32, 64, 128]
activation_fns = [nn.ReLU, nn.Sigmoid]
optimizing_fns = [torch.optim.SGD, torch.optim.Adam]
learning_rates = [0.05, 0.1, 0.2]

loss_fn = nn.CrossEntropyLoss()
input_layer = 28*28
output_layer = len(labels)
num_epoch = 5
gpu = False
```

```
start_time = time.perf_counter()


d = {i: [] for i in hidden_layers}
scores = {}

for hidden_layer in hidden_layers:
    for batch_size in batch_sizes:
        for activation_fn in activation_fns:
            for optimize_fn in optimizing_fns:
                for learning_rate in learning_rates:
```

```
                    params = {
                        "hidden_layer": hidden_layer,
                        "batch_size": batch_size,
                        "loss_fn": loss_fn,
                        "learning_rate": learning_rate,
                        "optimize_fn": optimize_fn,
                        "activation_fn": activation_fn,
                        "input_layer": input_layer,
                        "output_layer": output_layer,
                        "num_epoch": num_epoch,
                        "gpu": gpu
                    }

                    scores[hidden_layer] = scores.get(hidden_layer, {})
                    next_step = scores[hidden_layer]

                    next_step[batch_size] = next_step.get(batch_size, {})
                    next_step = next_step[batch_size]

                    next_step[activation_fn.__name__] =
next_step.get(activation_fn.__name__, {})
                    next_step = next_step[activation_fn.__name__]

                    next_step[optimize_fn.__name__] =
next_step.get(optimize_fn.__name__, {})
                    next_step = next_step[optimize_fn.__name__]

                    prediction_score = get_score(**params)
                    next_step[learning_rate] = next_step.get(learning_rate,
prediction_score)

                    d[hidden_layer].append(prediction_score)


minutes_to_go = (time.perf_counter() - start_time) // 60
seconds_to_go = (time.perf_counter() - start_time) % 60
print(f"{minutes_to_go} минут, {seconds_to_go} секунд")
```

```
all_scores = np.array(list(d.values())).flatten()
print(all_scores, all_scores.size)
```

```
plt.scatter(y=np.arange(108), x=all_scores)
plt.title("Распределение accuracy")
plt.xlabel("Accuracy, %")
plt.ylabel("Номер испытания")
plt.show()
```

```
plt.hist(all_scores, bins=40)
```

```
plt.title("Гистограмма accuracy")
plt.xlabel("Accuracy, %")
plt.ylabel("Количество")
plt.show()
```