Н.В. Безвиконный
А.А. Гуськов
С.Д. Лавров

2024

Программная эмуляция оптомагнитной нейронной сети для анализа рукописного текста

Методическое пособие

МИР НАУКИ
ИЗДАТЕЛЬСТВО

УДК 004.8
ББК 32.813
Б 39

Рецензент: Шестакова Анастасия Павловна, к.ф.-м.н., доцент кафедры оптико-электронных приборов и систем, РТУ МИРЭА.

Безвиконный, Никита Владиславович
Гуськов, Андрей Александрович
Лавров, Сергей Дмитриевич

Б 39 Программная эмуляция оптомагнитной нейронной сети для анализа рукописного текста. Методическое пособие – М.: Мир науки, 2024. – Режим доступа: https://izd-mn.com/PDF/51MNNPU24.pdf – Загл. с экрана.

ISBN 978-5-907891-36-4
DOI: 10.15862/51MNNPU24

Методическое пособие предназначено для студентов старших курсов технических вузов, изучающих современные подходы к моделированию и разработке нейронных сетей. Оно используется в рамках курса, посвященного программным методам эмуляции физических систем, и обеспечивает теоретическую и практическую базу для освоения принципов работы оптических сетей. В первой части пособия излагаются основы оптомагнитных нейронных сетей, включая физические принципы управления и адаптации магнитных состояний с помощью лазерных импульсов. Во второй части представлено пошаговое руководство по разработке программного кода на Python с использованием PyTorch для обработки изображений и распознавания рукописных символов. Пособие способствует формированию компетенций по интеграции физических моделей в программную реализацию, анализу данных и применению нейронных сетей в задачах обработки информации.

Методическое пособие издается в авторской редакции.

Авторский коллектив: Безвиконный Никита Владиславович, Гуськов Андрей Александрович, Лавров Сергей Дмитриевич.

Данное пособие было создано в ходе выполнения проекта №075-15-2022-1131, финансируемого Министерством науки и образования РФ.

ISBN 978-5-907891-36-4

Оглавление

Глава 1. Магнитооптическая схема искусственной нейронной сети

1. Введение

Современные вычислительные системы, использующие искусственные нейронные сети (ИНС), сталкиваются с быстрым ростом энергопотребления, особенно при решении сложных научных задач, где ИНС предлагают значительные преимущества перед традиционными алгоритмами. Это особенно актуально в таких областях, как физика конденсированных сред и элементарных частиц, где объемы передаваемых данных могут достигать 1 Пбит/с. Решение таких задач требует огромных энергетических затрат, что ограничивает использование стандартного вычислительного оборудования. В этой связи нейроморфные вычислительные системы, особенно импульсные нейронные сети, представляют собой перспективное направление, обеспечивая низкое энергопотребление при высокой скорости обработки данных. Эти системы уже находят применение в оптимизации и исследовании квантовых состояний.

Одним из эффективных подходов к снижению энергозатрат является использование аналоговых вычислений в памяти (AIMC), которые уменьшают потребность в передаче данных между слоями сети и повышают энергоэффективность. Тем не менее, реальные возможности AIMC для ИНС в физических приложениях остаются недостаточно исследованными, и в настоящее время физики часто не учитывают энергозатраты вычислительных систем как важный критерий их эффективности.

В последние годы значительный прогресс в управлении магнетизмом с помощью ультракоротких лазерных импульсов открыл новые возможности для более быстрого и энергоэффективного переключения магнитных состояний, что превосходит традиционные методы с использованием внешних магнитных полей. Особенно важным открытием стало наблюдение многошаговой динамики переключения в материалах, таких как Co/Pt, что позволило разработать опто-магнитные синаптические элементы с возможностью постоянной адаптации. Это открывает перспективы для создания нейронных сетей с опто-магнитными синапсами, которые могут обучаться с использованием глобальной обратной связи, значительно минимизируя потребность в внешнем хранении данных.

2. Принцип функционирования магнитооптической нейронной сети

Магнитооптическая нейронная сеть — это система, использующая взаимодействие света и магнитных свойств материала для выполнения вычислений, аналогичных процессам в биологических нейронных сетях. Основная идея заключается в том, чтобы управлять магнитными состояниями материала с помощью лазерного излучения, при этом эти магнитные состояния выступают в роли синаптических весов сети.

Общий процесс работы такой сети можно разбить на 6 отдельных пунктов:

1. Подготовка входного сигнала и его кодирование.

Входные данные, такие как изображения, символы или бинарные последовательности, преобразуются в оптический сигнал. Это достигается путем кодирования входной информации в сами параметры электромагнитного излучения— интенсивности, поляризации или пространственное распределение. Например, наличие или отсутствие света может соответствовать бинарным значениям 1 или 0. Цель этого этапа — представить данные в форме, которую сеть может принять и обработать оптическим методом.

2. Управление магнитными состояниями и обучение сети.

Магнитные состояния используемых в подобных системах материалов, например тонких пленок кобальта и платины, служат синаптическими весами. Эти состояния можно изменять с помощью ультракоротких лазерных импульсов определенной поляризации, которые увеличивают или уменьшают магнитный момент в выбранных областях. Процесс обучения сети заключается в инициализации магнитных состояний и последующей их корректировке в соответствии с алгоритмом обучения. В данном случае наиболее часто используются алгоритмы, схожие с обучением перцептрона. В ходе такого итеративного обучения после каждого введенного обучающего примера синаптические веса адаптируются для уменьшения ошибки классификации.

3. Хранение синаптических весов.

Одним из ключевых преимуществ магнитных состояний является их способность сохранять свое состояние без постоянного энергопитания, что фактически обеспечивает энергонезависимое хранение информации. Стабильность магнитных доменов позволяет сети сохранять свои настройки между сеансами работы, снижая энергопотребление и обеспечивая долговременное хранение синаптических весов.

4. Обработка входного сигнала.

После обучения сеть способна обрабатывать новые входные данные. Оптический сигнал, представляющий новые данные, направляется на материал с зафиксированными в ходе проведенного обучения магнитными состояниями. Взаимодействие света с магнитными доменами вызывает изменения в параметрах светового сигнала из-за магнитооптических эффектов, таких как эффект Керра или Фарадея. Эти изменения соответствуют применению синаптических весов к входному сигналу, позволяя сети выполнять необходимые вычислительные операции.

5. Суммирование сигналов и активация нейронов.

Измененные оптические сигналы от каждого синапса собираются и суммируются, имитируя процесс суммирования в биологическом нейроне. Важно понимать, что суммирование сигналов осуществляется путем объединения интенсивностей оптических сигналов от каждого синапса на детекторе (например, CCD-камере). В эксперименте оптические сигналы от различных синапсов собираются и их интенсивности суммируются на уровне детектирования, что эквивалентно суммированию взвешенных входов в нейроне. Полученный суммарный сигнал сравнивается с определенным пороговым значением с помощью активационной функции. Если суммарный сигнал превышает порог, нейрон активируется (выдает сигнал "1"); если нет, он остается неактивным ("0"). Это позволяет сети принимать решения и классифицировать входные данные на основе обработанной информации.

6. Вывод результата.

Изменения в оптическом сигнале фиксируются с помощью детекторов, таких как фотодиоды или CCD-камеры. Полученные данные преобразуются в цифровую форму, и программное обеспечение интерпретирует результаты, предоставляя их в удобном для пользователя формате — числовые значения, графики или визуальные представления. Этот этап завершает процесс обработки, позволяя пользователю получить конечный результат работы сети.

Таким образом, магнитооптическая нейронная сеть работает путем преобразования входных данных в оптический сигнал, управления магнитными состояниями материала для хранения и адаптации синаптических весов, обработки входного сигнала с учетом этих весов, суммирования и активации нейронов, а затем вывода результата. Такая сеть сочетает в себе преимущества оптической передачи данных и магнитного хранения информации, обеспечивая высокую скорость, энергоэффективность и возможность долговременного хранения настроек сети.

На текущий момент представлено несколько вариаций физической реализации подобной магнито-оптической системы. Одна из первых работ, в которой был продемоснтрирована концепция такой магнитооптической сети была представлена в статье: "An opto-magnetic neural network capable of learning and classification of digitized 3×3 characters exploiting local storage in the magnetic material" (Applied Physics Letters 120, 022403 (2022); doi: 10.1063/5.0073280). Для удобства сначала опишем особенности реализации ее компоновки и состава.

3. Экспериментальная реализация магнитооптической нейронной сети

Принципиальная функциональная схема экспериментальной установки, предназначенной для демонстрации функционирования магнитооптической нейронной сети, представлена на рисунке 1. В ней используются лазерные импульсы, генерируемые регенеративным усилителем титан-сапфировой (Ti:Sapphire) лазерной системы с длиной волны 800 нм, частотой повторения 1 кГц и длительностью импульса 4 пс. Такие системы обладают высокой энергией в импульсе, необходимой для переключения магнитных материалов, поэтому оптимально использовать источники на основе титан-сапфировых фемтосекундных лазеров.

Лазерный пучок разделяется на два луча с помощью несимметричного делителя луча, например, соотношением 80:20. Таким образом, формируются более мощный луч накачки и слабый луч зондирования.

Луч зондирования диаметром в несколько миллиметров после прохождения через образец и объектив с небольшим увеличением, расположенный между двумя почти перекрестными призмами Глана — поляризатором и анализатором, регистрируется монохромной CCD-камерой с заданной экспозицией. Такая конфигурация позволяет фиксировать изменения магнитных состояний в образце посредством магнитооптического эффекта.

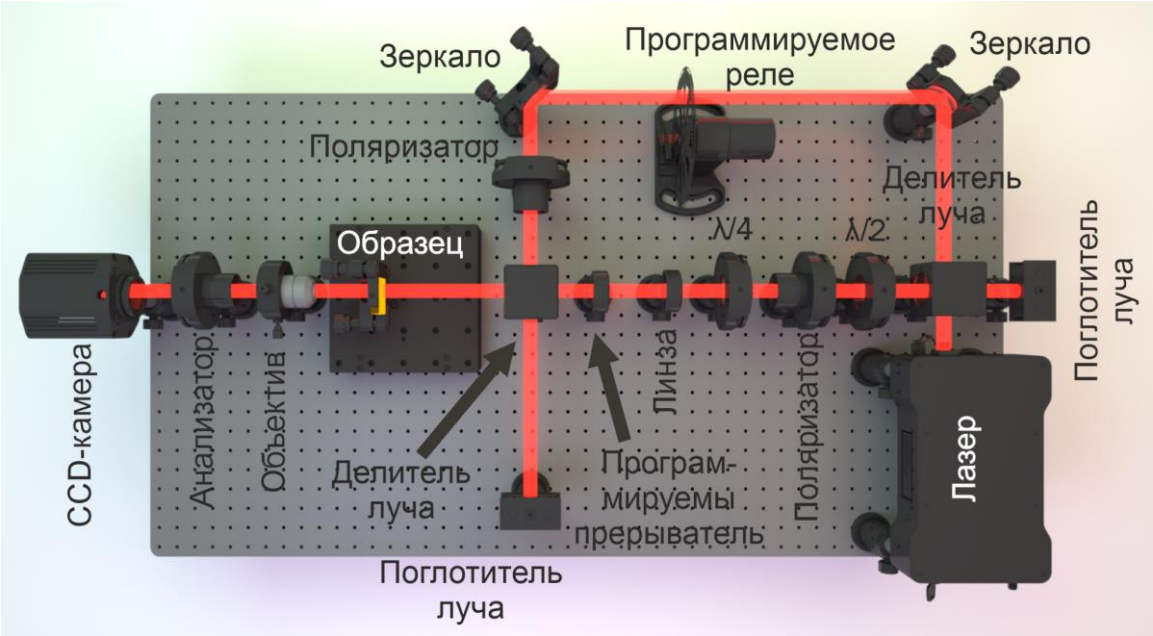

Рисунок 1. Принципиальная схема экспериментальной установки для создания опто-магнитной нейронной сети

Для рассматриваемого эксперимента используются достаточно сложные многослойные магнитные образцы. Например в указанной выше работе использовался образец представляющие собой стекло/Ta/Pt/Co/Pt/MgO/Ta. Подобные пленки обладают сильной перпендикулярной магнитной анизотропией и коэрцитивной силой порядка нескольких десятков мТл.

Программируемое реле на линейной стадии направляет линейно поляризованный луч зондирования на образец через светоделительный кубик или перекрывает его во время воздействия луча накачки. Количество импульсов луча накачки, падающих на образец, точно контролируется с помощью программируемого прерывателя луча (шаттер). Луч накачки адресует девять различных областей на образце, соответствующих синапсам нейронной сети. Доступ к этим областям осуществляется путем перемещения образца с помощью программируемой движущейся платформы в пути луча накачки.

Две полуволновые пластины (λ/2), установленные на пути лучей накачки и зондирования, контролируют энергию лучей накачки и зондирования (λ/2 пластина в плече зондирования не нарисована на рисунке). Вращающаяся четвертьволновая пластина (λ/4) в процессе обучения позволяет переключать поляризационные состояния луча накачки между правой и левой круговой поляризацией, что используется для увеличения или уменьшения намагниченности в образце. Фокусировка луча накачки на образец достигается с помощью линзы.

4. Принцип работы экспериментальной установки

Основой функционирования магнитооптической нейронной сети является взаимодействие света с магнитными состояниями материала посредством магнитооптического эффекта. Система использует разделение лазерного пучка на два луча: луч накачки для управления синаптическими весами и луч зондирования для их детектирования.

Луч зондирования проходит через области образца с различной намагниченностью, функционирующие как синапсы. Интенсивность прошедшего света зависит от намагниченности этих областей. Поляризатор и анализатор настроены почти перпендикулярно друг другу, что позволяет детектировать изменения поляризации света, вызванные магнитооптическим эффектом, и, следовательно, изменения магнитных состояний в образце. Таким образом, синаптические веса могут быть изменены управлением намагниченностью посредством оптомагнитных эффектов.

Для управления синаптическими весами используются ультракороткие лазерные импульсы луча накачки с определённой круговой поляризацией. В указанных выше тонких плёнках возможно обратимо изменять намагниченность под воздействием серий импульсов разной поляризации, что позволяет точно регулировать синаптические веса.

Например, на поверхности магнитного образца можно выделить матрицу пикселей фиксированного размера, например 3×3 или больше. Такая матрица может служить набором синапсов нейронной сети, где каждый пиксель соответствует отдельному синапсу. Управляя намагниченностью каждого пикселя с помощью лазерных импульсов определённой поляризации, можно устанавливать необходимые синаптические веса для выполнения вычислительных операций.

В процессе работы сети входные данные кодируются в виде оптических сигналов, которые направляются на соответствующие пиксели (синапсы) матрицы. Проходя через эти области, оптические сигналы модифицируются в соответствии с магнитными состояниями пикселей благодаря магнитооптическому эффекту. Изменения в поляризации или интенсивности света после прохождения через синапсы соответствуют применению синаптических весов к входным сигналам.

Модифицированные оптические сигналы затем суммируются, формируя общий выходной сигнал. Суммирование происходит за счёт объединения оптических сигналов от всех синапсов, что соответствует сумме взвешенных входов в нейроне. Этот суммарный сигнал детектируется с помощью фотодетектора или CCD-камеры.

Полученный выходной сигнал сравнивается с пороговым значением для определения активации нейрона. Если суммарная интенсивность превышает порог, нейрон считается активированным (выход "1"); если нет, он остаётся неактивным (выход "0"). Это позволяет сети выполнять необходимые вычислительные операции, такие как распознавание образов или классификация данных.

На рисунке 2 схематически показан процесс обучения магнитооптической сети. Изображены ячейки оптомагнитного синапса, которые изначально находятся в полностью поляризованном состоянии. Под воздействием последовательных импульсов света происходит сегментарная переполяризация этих ячеек, которая в итоге принимает форму круга. Это изменение магнитного состояния ячеек влияет на интенсивность света, проходящего через них. Таким образом, отражается принцип работы оптомагнитных синапсов, где изменения в магнитных доменах приводят к изменению оптических свойств материала, что показано на примере изменений оптического контраста при обучении нейронной сети в статье. Важно отметить, что использование различных параметров поляризации накачивающего излучения (левоциркулярная или правоциркулярная поляризация) позволяет как двигаться сверху вниз по изображению, так и наоборот — снизу вверх. Из таких синаптических ячеек может быть собрана матрица, которая, в свою очередь, будет использоваться для обработки изображений.

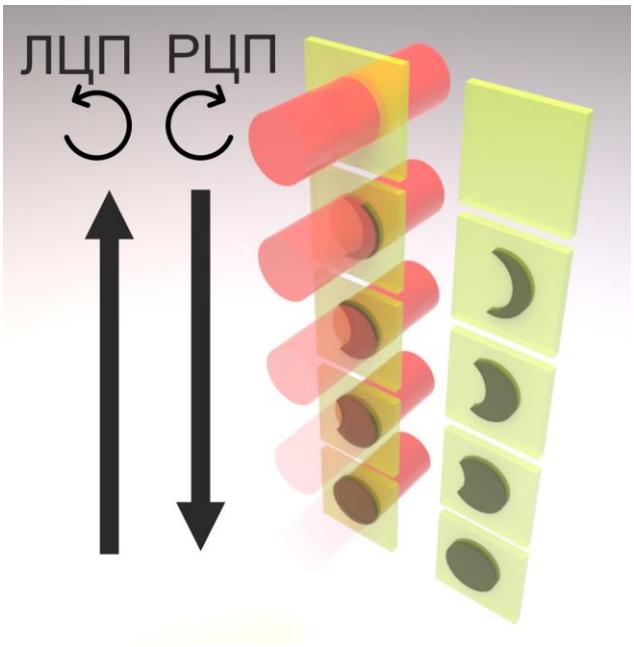

Рисунок 2 – Схематическое изображение процесса обучения магнитооптической сети. Инициализация ячеек синапса в полностью поляризованном состоянии и их сегментарная переполяризация под воздействием световых импульсов, что влияет на магнитное состояние и интенсивность проходящего света. Показаны изменения оптического контраста при тренировке нейронной сети. Справа представлено изображение после проведения обучения.

В этой главе мы подробно рассмотрели принцип функционирования магнитооптической нейронной сети и ее экспериментальную реализацию. Магнитооптические нейронные сети объединяют преимущества оптической передачи данных и энергонезависимого магнитного хранения информации, обеспечивая высокую скорость обработки и энергоэффективность. Экспериментальные исследования показали возможность управления синаптическими весами посредством ультракоротких лазерных импульсов и продемонстрировали потенциал таких систем в задачах распознавания образов и классификации данных.

Понимание принципов работы и экспериментальной реализации магнитооптической нейронной сети является важным шагом к ее практическому применению. В следующей главе мы перейдем к программной реализации модели такой сети на языке Python. Мы рассмотрим алгоритмы обучения и распознавания, используемые в магнитооптических сетях, и продемонстрируем, как они могут быть реализованы и протестированы в программной среде. Это позволит оценить эффективность и возможности магнитооптических нейронных сетей при решении различных задач и сравнить их с традиционными подходами.

Глава 2. Реализация искусственной нейронной сети

1. Введение

Распознавание рукописного текста является одной из ключевых задач в области компьютерного зрения и обработки естественного языка. Оно находит широкое применение в различных сферах, таких как автоматизация ввода данных, цифровизация архивов, системы распознавания почерка и многое другое. Приведенный далее код представляет собой полноценное решение для распознавания рукописных слов на основе сверточной рекуррентной нейронной сети (CRNN), реализованной с использованием библиотеки PyTorch.

Целью данного проекта является разработка и обучение модели, способной эффективно распознавать рукописные слова из изображений. Для достижения этой цели можно выделить несколько ключевых задач:

Сбор и Предобработка Данных: Обработка изображений рукописных слов, включая загрузку, изменение размера, преобразование в градации серого и нормализацию. Также требуется построение словарей для преобразования символов в числовые индексы и обратно.

Аугментация Данных: Применение различных техник аугментации данных для увеличения разнообразия обучающих примеров и повышения устойчивости модели к различным искажениям изображений.

Разделение Данных: Разделение набора данных на тренировочную и валидационную выборки для объективной оценки производительности модели и предотвращения переобучения.

Создание Пользовательского Датасета и Загрузчиков Данных: Реализация класса Dataset для удобного доступа к данным и создание DataLoader для эффективной загрузки данных в процессе обучения и валидации.

Определение Архитектуры Модели (CRNN): Построение сверточной рекуррентной нейронной сети, которая сочетает в себе возможности сверток для извлечения признаков из изображений и рекуррентных слоев для обработки последовательностей символов.

Настройка Процесса Обучения: Определение функции потерь (CTC Loss), выбор оптимизатора (Adam), настройка планировщика скорости обучения и внедрение механизма ранней остановки для оптимизации процесса обучения.

Обучение Модели: Реализация цикла обучения, включающего тренировочную и валидационную фазы, с регулярным мониторингом показателей потерь и ошибок.

Оценка и Визуализация Результатов: Вычисление и визуализация метрик производительности модели, таких как функция потерь и показатель ошибки символов (CER), а также сохранение информации о неправильно распознанных словах для дальнейшего анализа.

Структура Кода

Код структурирован следующим образом:

1. **Импорт Библиотек:** На первом этапе происходит импорт необходимых библиотек, включая стандартные библиотеки Python, а также специализированные библиотеки для машинного обучения и обработки изображений, такие как PyTorch, torchvision, scikit-learn и другие.

2. **Парсинг Файла Слов (words.txt):** Реализована функция parse_words_file, которая обрабатывает файл words.txt, извлекая идентификаторы слов, их метки и полную строку информации. Это позволяет подготовить данные для дальнейшей обработки и обучения модели.

3. **Построение Пути К Изображениям:** Функция get_image_path отвечает за конструкцию полного пути к изображению слова на основе базового пути и идентификатора слова.

4. **Предобработка Изображений и Построение Словаря Символов:** Загрузка изображений, их изменение размера и преобразование в числовые массивы. Также формируется словарь символов, который используется для преобразования символов в числовые индексы и обратно.

5. **Разделение Данных на Тренировочную и Валидационную Выборки:** используется функция train_test_split для разделения данных, обеспечивая при этом случайность и репрезентативность выборок.

6. **Создание Пользовательского Датасета (**HandwritingDataset**):** Класс HandwritingDataset наследуется от torch.utils.data.Dataset и предоставляет методы для доступа к изображениям и соответствующим меткам, а также применяет необходимые трансформации к изображениям.

7. **Определение Архитектуры Модели (**CRNN**):** Класс CRNN реализует сверточную рекуррентную нейронную сеть, включающую сверточные слои для извлечения признаков и рекуррентные слои (LSTM) для обработки

последовательностей символов. Модель обучается с использованием функции потерь CTC (Connectionist Temporal Classification), что позволяет эффективно справляться с задачами распознавания последовательностей без необходимости предварительного выравнивания.

8. **Настройка Процесса Обучения:** Определение функции потерь (nn.CTCLoss), оптимизатора (optim.Adam), планировщика скорости обучения (ReduceLROnPlateau) и параметров для механизма ранней остановки. Также устанавливается параметр для обрезки градиентов, что помогает избежать проблем с исчезающими или взрывающимися градиентами.

9. **Цикл Обучения:** Реализован основной цикл обучения, который включает тренировочную фазу, валидационную фазу, декодирование предсказаний, вычисление метрик ошибок (WER и CER), обновление весов модели и сохранение лучших весов на основе показателей валидации. Также реализован механизм ранней остановки, который прекращает обучение, если модель не демонстрирует улучшений в течение заданного числа эпох.

10. **Визуализация Результатов Обучения:** С использованием библиотеки Matplotlib строятся графики, отображающие динамику функции потерь на тренировочной и валидационной выборках, а также изменение показателя CER на валидационной выборке в зависимости от числа эпох обучения. Это позволяет визуально оценить процесс обучения и выявить возможные проблемы, такие как переобучение или недообучение.

11. **Итоговая Оценка Модели:** после завершения обучения проводится финальная оценка модели на валидационной выборке, включающая декодирование предсказаний, вычисление средних значений показателей ошибок и сохранение информации о неправильно распознанных словах для дальнейшего анализа.

Ключевые Технологии и Подходы

Сверточные Нейронные Сети (CNN): используются для извлечения пространственных признаков из изображений рукописного текста. Сверточные слои последовательно уменьшают размерность изображений, сохраняя при этом важные признаки.

Рекуррентные Нейронные Сети (RNN) с LSTM: применяются для обработки последовательностей символов, полученных из извлеченных признаков. LSTM-слои эффективно справляются с задачами распознавания последовательностей, сохраняя информацию о предыдущих символах.

CTC Loss: Функция потерь CTC позволяет обучать модели распознавания последовательностей без необходимости предварительного выравнивания входных и целевых последовательностей, что существенно упрощает процесс обучения.

Аугментация Данных: Применение различных техник аугментации, таких как случайные повороты, аффинные преобразования, изменения яркости и контрастности, а также случайное стирание, помогает повысить обобщающую способность модели и ее устойчивость к различным искажениям в данных.

Мониторинг и Регуляризация: Использование планировщиков скорости обучения, обрезки градиентов и механизма ранней остановки способствует стабильности и эффективности процесса обучения, предотвращая переобучение и обеспечивая оптимальное сходимость модели.

Метрики Оценки Качества: Character Error Rate (CER) позволяет объективно оценивать качество распознавания модели, предоставляя информацию о количестве ошибок на уровне символов.

Данный проект демонстрирует полный цикл разработки модели распознавания рукописного текста, начиная от предобработки данных и заканчивая обучением, оценкой и визуализацией результатов. Использование современных методов машинного обучения и глубоких нейронных сетей обеспечивает высокую точность и эффективность модели. Кроме того, внедрение механизмов аугментации данных, регуляризации и мониторинга метрик позволяет создавать устойчивые и обобщающие модели, способные эффективно справляться с реальными задачами распознавания рукописного текста.

Этот подход может служить основой для дальнейших исследований и разработок в области распознавания рукописных символов и текстов, а также быть адаптированным для решения смежных задач в области компьютерного зрения и обработки естественного языка.

2. Набор данных для обучения

Для начала необходимо подготовить набор изображений, содержащих рукописный текст, и индексировать его. Важно отметить, что выполнить это вручную практически невозможно. Для успешного обучения нейронной сети требуется словарь, состоящий из нескольких сотен тысяч изображений букв. При этом эти изображения не должны быть написаны одним автором, поскольку в противном случае нейронная сеть будет эффективно распознавать текст только одного почерка, а для анализа текста, написанного другими людьми, результат

может быть существенно хуже. В связи с этим рекомендуется использовать уже существующие открытые базы данных, такие как IAM Handwriting Database.

IAM Handwriting Database — это открытая база данных, которая является стандартом де-факто для разработки и тестирования алгоритмов распознавания рукописного текста. Она была создана для использования в научных исследованиях в области машинного обучения, компьютерного зрения и обработки естественного языка.

Эта база данных содержит изображения рукописного текста, аннотированные на уровне слов, строк и полных предложений, что делает её универсальным инструментом для разработки моделей как для распознавания текста, так и для сегментации.

Основные характеристики IAM Handwriting Database:

1. **Разнообразие почерков**: База данных включает рукописные записи от большого числа авторов (свыше 650), что обеспечивает высокую вариативность почерков и делает её реалистичной для использования в реальных задачах.
2. **Тип данных**: содержит текст на английском языке, представленный как строки и отдельные слова, а также бинарные изображения (черно-белые) высокого качества.
3. **Аннотации**: Данные снабжены текстовыми метаданными, включая разметку слов, строк и координат области интереса (bounding boxes).
4. **Разделение данных**: IAM Handwriting Database организована таким образом, чтобы можно было легко разделить данные на обучающую, валидационную и тестовую выборки, что обеспечивает удобство для обучения и оценки моделей.

Эта база данных широко используется для задач, таких как: Распознавание текста (Text Recognition), оптическое распознавание символов (OCR), сегментация текста (Text Segmentation), классификация почерков (Writer Identification).

Благодаря высокому качеству и стандартам разметки IAM Handwriting Database позволяет исследователям и разработчикам эффективно разрабатывать и тестировать модели для распознавания рукописного текста. Использование этой базы данных в обучении нейронных сетей предоставляет возможность сравнивать результаты с другими работами, так как она является общепринятым эталоном в данной области.

Для получения доступа к IAM Handwriting Database необходимо пройти регистрацию и соблюсти условия её использования, которые включают ограничение на коммерческое применение без согласования с авторами.

3. Описание кода на Python для реализации модели нейронной сети и ее обучения

Код был написан и протестирован в среде разработке PyCharm. Использовавшая версия языка Python - 3.12.

Этот код начинается с раздела импорта, в котором подключаются различные библиотеки и модули, необходимые для выполнения задач обработки изображений, построения и обучения нейронной сети, а также для анализа и визуализации результатов.

Первой библиотекой является os, стандартный модуль Python, который предоставляет функции для взаимодействия с операционной системой. Он используется для работы с файловой системой, например, для построения путей к файлам изображений и проверки их существования.

numpy импортируется как np и служит для работы с многомерными массивами и матрицами, а также для выполнения математических операций над этими структурами данных. В данном коде numpy используется для преобразования изображений в массивы чисел, что упрощает их дальнейшую обработку и подачу в модель.

Библиотека PIL (Python Imaging Library), а именно модули Image и UnidentifiedImageError, используются для загрузки и обработки изображений. Image позволяет открывать, изменять размер и конвертировать изображения в различные форматы, а UnidentifiedImageError обрабатывает исключения, возникающие при попытке открыть некорректные или поврежденные файлы изображений.

Модуль train_test_split из библиотеки sklearn.model_selection применяется для разделения данных на обучающую и валидационную выборки. Это необходимо для оценки производительности модели на данных, которые не использовались во время обучения, что помогает избежать переобучения.

Далее, импортируются основные компоненты библиотеки torch, включая torch.nn для определения нейронных сетей, torch.optim для оптимизаторов, Dataset и DataLoader из torch.utils.data для создания пользовательских наборов данных и их загрузки в пакетах. Эти инструменты являются фундаментальными для построения, обучения и управления процессом обучения модели на основе глубокого обучения.

torchvision.transforms предоставляет инструменты для аугментации и преобразования изображений, такие как повороты, сдвиги, изменение яркости и

контрастности. Аугментация данных улучшает обобщающую способность модели, делая её более устойчивой к вариациям в данных.

Модуль Counter из библиотеки collections используется для подсчета частоты появления элементов, что в данном случае применяется для анализа распределения классов в наборе данных.

matplotlib.pyplot импортируется как plt и используется для визуализации данных и результатов обучения модели. С его помощью можно строить графики потерь, ошибок и других метрик, что помогает в анализе эффективности модели.

jiwer предоставляет функции для вычисления показателей качества распознавания, таких как Word Error Rate (WER) и Character Error Rate (CER). Эти метрики позволяют количественно оценить точность модели в задачах распознавания текста.

Наконец, модуль warnings используется для контроля предупреждений, возникающих во время выполнения кода. В данном случае, предупреждения подавляются с помощью команды warnings.filterwarnings("ignore"), что позволяет избежать избыточного вывода сообщений и сосредоточиться на основных результатах.

Таким образом, раздел импорта тщательно подбирает необходимые инструменты и библиотеки, обеспечивая все необходимые функции для загрузки и обработки данных, построения и обучения модели, а также для анализа и визуализации результатов.

```
import os
import numpy as np
from PIL import Image, UnidentifiedImageError
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from collections import Counter
import matplotlib.pyplot as plt
from jiwer import wer, cer
import warnings

warnings.filterwarnings("ignore")
```

После выполнения импорта всех необходимых модулей осуществляется ввод служебных функций. Первые из них предназначены для подготовки

словарей из изображений, на которых и будет проводится обучение и апробация создаваемой нейронной модели.

3.1. Чтение исходных данных и их предварительная подготовка

Функция parse_words_file играет ключевую роль в подготовке данных для модели распознавания рукописного текста. Она отвечает за чтение и разбор файла words.txt, извлекая из него необходимые сведения о словах, которые будут использоваться в обучении и валидации модели.

```
    def get_image_path(base_path, word_id):
        parts = word_id.split('-')
        subdir1 = parts[0]
        subdir2 = '-'.join(parts[:2])
        image_name = f"{word_id}.png"
        image_path = os.path.join(base_path, subdir1,
subdir2, image_name)
        return image_path
```

Функция принимает два аргумента: words_file_path и include_err, где words_file_path — это путь к файлу words.txt, a include_err — булевый параметр, определяющий, следует ли включать в выборку слова, помеченные как ошибки ('err'). По умолчанию include_err установлен в False, что означает исключение таких слов из выборки.

Внутри функции инициализируется пустой список samples, который будет хранить кортежи с информацией о каждом слове. Затем происходит открытие файла по указанному пути с использованием менеджера контекста with, что обеспечивает автоматическое закрытие файла после завершения работы с ним.

Функция последовательно перебирает каждую строку файла. Если строка начинается с символа '#', она считается комментарием и пропускается, так как не содержит полезной информации о словах. Затем строка очищается от пробельных символов по краям с помощью метода strip(). Если после очистки строка оказывается пустой, она также пропускается, чтобы избежать обработки лишних данных.

Далее строка разбивается на части по пробелам с помощью метода split(' '), результатом чего является список parts. Функция проверяет, содержит ли этот список как минимум 9 элементов. Такое условие предполагает, что строка имеет необходимую структуру и содержит все требуемые поля для извлечения информации о слове.

Если условие о длине списка выполняется, из него извлекаются три ключевых элемента:

1. word_id — идентификатор слова, который находится в первом элементе списка parts[0].

2. word_label — метка слова, расположенная в последнем элементе списка parts[-1].

3. status — статус слова, находящийся во втором элементе списка parts[1].

Далее функция проверяет значение переменной status. Если статус равен 'err' и параметр include_err установлен в False, это означает, что слово помечено как ошибка и не должно включаться в выборку. В таком случае строка пропускается, и слово не добавляется в список samples. Если же include_err установлен в True, то даже слова со статусом 'err' будут включены в выборку.

Если слово проходит все проверки, оно добавляется в список samples в виде кортежа, содержащего word_id, word_label и всю исходную строку line. Это обеспечивает сохранение полной информации о слове, что может быть полезно для последующего анализа или отладки.

После завершения перебора всех строк файла функция возвращает список samples, содержащий все отобранные слова, соответствующие заданным критериям. Этот список затем используется для дальнейшей загрузки изображений слов и подготовки данных для обучения модели.

Таким образом, функция parse_words_file эффективно обрабатывает файл words.txt, фильтруя и структурируя данные в удобный для последующей обработки формат. Она обеспечивает гибкость за счет возможности включать или исключать слова с ошибками, что позволяет адаптировать подготовку данных под конкретные задачи и требования модели.

Функция get_image_path предназначена для построения полного пути к файлу изображения слова на основе базового пути и уникального идентификатора слова. Эта функция играет важную роль в процессе загрузки и обработки изображений, обеспечивая правильное нахождение и доступ к необходимым файлам в структуре директорий проекта.

Функция принимает два аргумента: base_path и word_id. Параметр base_path представляет собой строку, указывающую на базовую директорию, где хранятся изображения слов. Это может быть, например, папка words, содержащая все изображения, организованные по определенной структуре

поддиректорий. Второй параметр, word_id, также является строкой и служит уникальным идентификатором для каждого слова. Этот идентификатор обычно содержит информацию, необходимую для определения местоположения конкретного изображения внутри базовой директории.

Внутри функции происходит несколько шагов для формирования полного пути к изображению. Сначала строка word_id разбивается на части с помощью метода split('-'), который разделяет строку по дефисам. Результатом является список parts, содержащий отдельные сегменты идентификатора слова. Например, если word_id равен "abc-123-def", то после разделения получим список ['abc', '123', 'def'].

Далее из этого списка извлекаются первые два элемента для формирования поддиректорий. Переменная subdir1 получает значение первого элемента списка parts[0], что в нашем примере будет 'abc'. Переменная subdir2 формируется объединением первых двух элементов списка с помощью метода '-'.join(parts[:2]), что даст строку 'abc-123'. Такая организация поддиректорий предполагает, что изображения слов распределены по директориям на основе первых частей их идентификаторов, что помогает эффективно структурировать и быстро находить необходимые файлы.

После формирования поддиректорий создается имя файла изображения. Переменная image_name получает значение строки, составленной из word_id с добавлением расширения .png, то есть f"{word_id}.png". В нашем примере это будет 'abc-123-def.png'. Это предполагает, что все изображения слов имеют формат PNG и названы в соответствии с их уникальными идентификаторами.

Следующим шагом используется функция os.path.join для объединения базового пути, поддиректорий и имени файла в единый полный путь к изображению. Переменная image_path содержит результат этой операции, что в нашем примере будет выглядеть как os.path.join('words', 'abc', 'abc-123', 'abc-123-def.png'). Функция os.path.join автоматически учитывает особенности операционной системы при формировании пути, обеспечивая корректность независимо от платформы (Windows, macOS, Linux).

Наконец, функция возвращает сформированный полный путь к файлу изображения через оператор return image_path. Этот путь затем используется в других частях кода для загрузки и обработки соответствующего изображения слова.

Таким образом, функция get_image_path обеспечивает систематическое и надежное построение путей к файлам изображений на основе уникальных идентификаторов слов и заданной структуры директорий. Это упрощает доступ к данным, делает код более читаемым и поддерживаемым, а также способствует эффективной организации файловой системы проекта.

```
    def get_image_path(base_path, word_id):
        parts = word_id.split('-')
        subdir1 = parts[0]
        subdir2 = '-'.join(parts[:2])
        image_name = f"{word_id}.png"
        image_path = os.path.join(base_path, subdir1,
subdir2, image_name)
        return image_path
```

Следующий блок кода отвечает за настройку путей к данным, определение размеров изображений, а также за настройку преобразований (аугментаций) данных для обучения и валидации модели. Давайте подробно рассмотрим каждую часть этого блока.

Сначала задаются пути к основным файлам и директориям, используемым в проекте. Переменная words_file_path устанавливается равной строке 'words.txt', что указывает на файл, содержащий информацию о словах, которые будут использоваться для обучения и валидации модели. Комментарий # Update with your actual path if different напоминает разработчику о том, что при необходимости этот путь можно изменить на актуальный, соответствующий структуре его проекта. Аналогично, переменная base_image_path устанавливается как 'words', что предполагает, что изображения слов хранятся в директории с названием words. Опять же, комментарий подчеркивает возможность изменения этого пути в зависимости от конкретной организации файловой системы проекта.

Далее, с помощью функции parse_words_file производится парсинг (разбор) файла words.txt. Вызов samples = parse_words_file(words_file_path, include_err=False) означает, что функция будет обрабатывать указанный файл, исключая слова, помеченные как ошибки ('err'), поскольку параметр include_err установлен в False. Это позволяет сформировать выборку данных, содержащую только корректные примеры для обучения модели, что важно для повышения качества обучения и предотвращения переобучения на ошибочных данных.

После этого определяются размеры изображений, с которыми будет работать модель. Переменные image_width и image_height устанавливаются равными 160 и 40 соответственно. Эти значения определяют ширину и высоту изображений в пикселях. Выбор размеров изображений влияет на входные данные модели: слишком большие изображения могут увеличить время обучения и потребление памяти, тогда как слишком маленькие могут привести к

потере важных деталей. Таким образом, выбор оптимальных размеров является балансом между точностью и эффективностью обучения.

3.2. Аугментация исходных данных

Следующей частью кода является настройка преобразований данных для обучения. Переменная train_transforms определяется с помощью transforms.Compose, что позволяет объединить несколько преобразований в одну последовательность. Первым шагом в этой последовательности является преобразование изображения из массива NumPy в объект PIL с помощью transforms.ToPILImage(). Это необходимо, поскольку многие последующие преобразования в библиотеке torchvision.transforms работают с объектами PIL.

Далее применяется случайный поворот изображения на угол до 10 градусов с помощью transforms.RandomRotation(degrees=10, fill=(0,)). Поворот помогает модели стать более устойчивой к различным ориентациям входных данных. Параметр fill=(0,) указывает, что области, появившиеся за пределами исходного изображения после поворота, будут заполнены черным цветом (значение 0 для градаций серого).

Следующим шагом используется transforms.RandomAffine, который выполняет аффинные преобразования, такие как сдвиги и сдвиги с наклоном. Параметры degrees=0 означают, что поворота не будет, но будут применяться сдвиги по осям translate=(0.1, 0.1) и наклоны shear=10. Эти преобразования добавляют разнообразие к данным, позволяя модели лучше обобщать информацию и справляться с различными вариациями входных изображений.

Преобразование transforms.ColorJitter(brightness=0.2, contrast=0.2) изменяет яркость и контрастность изображения на случайные значения в пределах заданных диапазонов. Это помогает модели быть устойчивой к изменениям освещения и контрастности, что часто встречается в реальных условиях.

После этого изображение снова преобразуется в тензор с помощью transforms.ToTensor(). Этот шаг необходим для преобразования данных в формат, подходящий для подачи в нейронную сеть, где каждый пиксель представляется как значение в диапазоне от 0 до 1.

Далее применяется transforms.RandomErasing(p=0.1, scale=(0.02, 0.33), ratio=(0.3, 3.3)), которое случайным образом стирает прямоугольные области изображения. Параметр p=0.1 указывает вероятность применения этого преобразования к каждому изображению. Параметры scale и ratio определяют диапазоны размеров и соотношений сторон стираемых областей. Это помогает модели быть устойчивой к частичным затуханиям или загрязнениям на изображении.

Заключительным шагом в цепочке преобразований является нормализация изображения с помощью transforms.Normalize((0.5,), (0.5,)). Нормализация приводит значения пикселей к заданному диапазону, что способствует ускорению и стабилизации процесса обучения модели.

Для валидационной выборки определяется переменная val_transforms, которая также использует transforms.Compose, но включает только базовые преобразования без случайных аугментаций. Сначала изображение преобразуется в объект PIL с помощью transforms.ToPILImage(), затем преобразуется в тензор с помощью transforms.ToTensor(), и, наконец, нормализуется с помощью transforms.Normalize((0.5,), (0.5,)). Отсутствие случайных преобразований при валидации позволяет объективно оценить производительность модели на неизмененных данных, обеспечивая более точные метрики качества.

Таким образом, данный блок кода настраивает пути к данным, определяет размеры изображений и устанавливает последовательности преобразований для подготовки данных к обучению и валидации модели. Аугментации данных для обучения помогают модели стать более устойчивой к разнообразным вариациям входных данных, улучшая её обобщающую способность, тогда как упрощённые преобразования для валидации обеспечивают объективную оценку её производительности.

```
    words_file_path = 'words.txt'
    base_image_path = 'words'

    samples = parse_words_file(words_file_path,
include_err=False)

    image_width = 160
    image_height = 40

    train_transforms = transforms.Compose([
       transforms.ToPILImage(),
       transforms.RandomRotation(degrees=10, fill=(0,)),
       transforms.RandomAffine(degrees=0, translate=(0.1,
0.1), shear=10),
       transforms.ColorJitter(brightness=0.2,
contrast=0.2),
       transforms.ToTensor(),
```

```
        transforms.RandomErasing(p=0.1,        scale=(0.02,
0.33), ratio=(0.3, 3.3)),  # Corrected placement
        transforms.Normalize((0.5,), (0.5,))
    ])

    val_transforms = transforms.Compose([
        transforms.ToPILImage(),
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])
```

3.3. Словарь символов и сплит данных

Следующая часть кода отвечает за построение словаря символов, загрузку и предварительную обработку изображений, кодирование меток, проверку распределения классов и разделение данных на обучающую и валидационную выборки.

```
    chars = set()


    images = []
    labels = []
    word_ids = []
    full_lines = []


    for word_id, word_label, full_line in samples:
        image_path    =    get_image_path(base_image_path,
word_id)
        if not os.path.exists(image_path):
            print(f"Image not found: {image_path}")
            continue
        try:

            img = Image.open(image_path).convert('L')
            img  =  img.resize((image_width,  image_height),
Image.LANCZOS)
            img = np.array(img)
```

```
            images.append(img)
            labels.append(word_label)
            word_ids.append(word_id)
            full_lines.append(full_line)
            chars.update(list(word_label))
        except (IOError, SyntaxError,
UnidentifiedImageError) as e:
            print(f"Skipping corrupted image:
{image_path}. Error: {e}")
            continue


    X = images
    y = labels


    chars = sorted(list(chars))
    char_to_num = {c: i + 1 for i, c in enumerate(chars)}
    num_to_char = {i + 1: c for i, c in enumerate(chars)}
    num_to_char[0] = ''
    encoded_labels = []
    label_lengths = []
    for label in y:
        label_seq = [char_to_num[c] for c in label]
        encoded_labels.append(label_seq)
        label_lengths.append(len(label_seq))

    for idx, lbl in enumerate(encoded_labels):
        if 0 in lbl:
            print(f"Label contains blank index at sample
{idx}")
    label_counter = Counter([char for label in y for char
in label])
    print("Class Distribution:", label_counter)
    X_train, X_val, y_train_enc, y_val_enc, y_train_str,
y_val_str, train_label_lengths, val_label_lengths,
word_train_ids, word_val_ids, word_train_lines,
word_val_lines = train_test_split(
```

```
        X, encoded_labels, y, label_lengths, word_ids,
full_lines, test_size=0.2, random_state=42, shuffle=True)
```

Рассмотрим каждую из этих операций более подробно.

Сначала создаётся пустое множество chars, которое будет использоваться для построения словаря уникальных символов, присутствующих в метках слов. Использование множества гарантирует, что каждый символ будет представлен только один раз, что важно для формирования корректного словаря.

Далее инициализируются четыре пустых списка: images, labels, word_ids и full_lines. Эти списки предназначены для хранения загруженных изображений, соответствующих им меток, идентификаторов слов и полных строк из файла words.txt соответственно. Такая структура данных позволяет удобно организовать информацию для последующей обработки и подачи в модель.

Следующий шаг включает цикл for, который перебирает каждый элемент из списка samples, полученного ранее при разборе файла words.txt. Для каждого слова извлекаются его идентификатор word_id, метка word_label и полная строка full_line. С помощью функции get_image_path строится полный путь к изображению слова на основе базового пути base_image_path и уникального идентификатора word_id.

Проверяется существование файла изображения по сформированному пути с помощью функции os.path.exists(image_path). Если изображение не найдено, выводится сообщение об отсутствии файла, и цикл переходит к следующему слову, пропуская текущий. Это предотвращает попытки загрузки несуществующих или перемещённых файлов, что могло бы вызвать ошибки во время выполнения.

Если файл изображения существует, происходит попытка его открытия и обработки внутри блока try-except. Изображение загружается с помощью Image.open(image_path) и преобразуется в оттенки серого с помощью метода .convert('L'), что упрощает дальнейшую обработку и снижает вычислительные затраты. Затем изображение изменяется по размеру до заданных размеров image_width и image_height с использованием метода .resize, применяя фильтр Image.LANCZOS для высококачественного масштабирования.

Преобразованное изображение конвертируется в массив NumPy с помощью np.array(img), что делает его удобным для хранения и последующей обработки в нейронной сети. Обратите внимание, что аугментации данных не применяются на этом этапе, поскольку они выполняются динамически во время загрузки данных через DataLoader с использованием заданных трансформаций.

Загруженное изображение добавляется в список images, а соответствующая ему метка word_label — в список labels. Также сохраняются

идентификатор слова word_id и полная строка full_line в соответствующих списках word_ids и full_lines. Это обеспечивает сохранение всей необходимой информации для каждого слова, что может быть полезно для анализа результатов и отладки модели.

Кроме того, множество chars обновляется с помощью метода chars.update(list(word_label)), добавляя все уникальные символы из текущей метки слова. Это необходимо для построения полного словаря символов, который будет использоваться для кодирования меток и декодирования предсказаний модели.

В случае возникновения исключений, таких как IOError, SyntaxError или UnidentifiedImageError, выводится сообщение о пропуске повреждённого изображения вместе с описанием ошибки. Это помогает идентифицировать и исключить некорректные файлы из выборки, предотвращая возможные сбои во время обучения.

После завершения цикла загрузки и предварительной обработки изображений, списки images и labels присваиваются переменным X и y соответственно для удобства дальнейшей работы.

Далее строятся отображения символов в числовые значения и обратно. Множество chars преобразуется в отсортированный список, чтобы обеспечить последовательность индексов для каждого символа. Создаются два словаря: char_to_num, который сопоставляет каждому символу уникальный числовой идентификатор, начиная с 1, и num_to_char, который выполняет обратное сопоставление, добавляя также символ с индексом 0, предназначенный для использования в качестве пустого символа в функции потерь CTC (Connectionist Temporal Classification).

После этого начинается процесс кодирования меток. Для каждой метки слова из списка y создаётся последовательность числовых идентификаторов с помощью словаря char_to_num. Эти закодированные последовательности добавляются в список encoded_labels, а длины этих последовательностей сохраняются в списке label_lengths. Такой подход необходим для подготовки данных к использованию в функции потерь CTC, которая требует информации о длинах меток.

Проводится проверка на наличие нулевых индексов в закодированных метках. Если хотя бы одна метка содержит индекс 0, выводится предупреждающее сообщение с указанием номера соответствующего образца. Это важно, поскольку индекс 0 зарезервирован для пустого символа в CTC и не должен присутствовать в исходных метках.

Далее осуществляется анализ распределения классов с помощью объекта Counter, который подсчитывает частоту появления каждого символа в метках

слов. Результат выводится на экран, предоставляя информацию о том, как часто встречаются различные символы в наборе данных. Это может быть полезно для оценки сбалансированности данных и принятия решений о необходимости дополнительных мер для обработки несбалансированных классов.

Наконец, происходит разделение данных на обучающую и валидационную выборки с помощью функции train_test_split из библиотеки sklearn.model_selection. В данном случае используется параметр test_size=0.2, что означает, что 20% данных будут отведены на валидацию, а оставшиеся 80% — на обучение. Параметр random_state=42 обеспечивает воспроизводимость разделения, а shuffle=True гарантирует случайное перемешивание данных перед разделением. При этом отсутствует стратификация (stratify), что позволяет избежать ошибок, связанных с несоответствием распределения классов в обучающей и валидационной выборках.

В результате выполнения этой части кода формируются все необходимые компоненты для дальнейшего обучения модели: загруженные и предварительно обработанные изображения, закодированные метки, словарь символов, а также разделённые на обучающую и валидационную выборки данные. Этот этап является фундаментальным для обеспечения корректного и эффективного обучения нейронной сети, поскольку качественная подготовка данных напрямую влияет на способность модели к обучению и её общую производительность.

3.4. Итоговая подготовка данных для загрузки в модель

Данный блок кода отвечает за создание пользовательского набора данных, определение функции объединения наборов данных и настройку загрузчиков данных для обучения и валидации модели. Эти компоненты играют критическую роль в эффективной подготовке и подаче данных в нейронную сеть, обеспечивая гибкость и оптимизацию процесса обучения.

```
    class HandwritingDataset(Dataset):
        def __init__(self, images, labels, label_lengths,
word_ids, full_lines, transform=None):
            self.images = images
            self.labels = labels
            self.label_lengths = label_lengths
            self.word_ids = word_ids
            self.full_lines = full_lines
            self.transform = transform
```

```
        def __len__(self):
            return len(self.images)

        def __getitem__(self, idx):
            image = self.images[idx]  # [H, W]
            label = self.labels[idx]
            label_length = self.label_lengths[idx]
            word_id = self.word_ids[idx]
            full_line = self.full_lines[idx]

            if self.transform:
                image = self.transform(image)
            else:
                image = torch.tensor(image,
dtype=torch.float32).unsqueeze(0)  # [1, H, W]

            return image, label, label_length, word_id,
full_line


    def collate_fn(batch):
        images, labels, label_lengths, word_ids, full_lines
= zip(*batch)

        images = torch.stack(images)

        labels_concat = []
        for lbl in labels:
            labels_concat.extend(lbl)
        labels_tensor = torch.tensor(labels_concat,
dtype=torch.long)

        label_lengths_tensor = torch.tensor(label_lengths,
dtype=torch.long)

        input_lengths = torch.full(size=(len(images),),
fill_value=34, dtype=torch.long)
```

```
        return images, labels_tensor, input_lengths,
label_lengths_tensor, word_ids, full_lines


    batch_size = 64

    train_dataset = HandwritingDataset(X_train,
y_train_enc, train_label_lengths, word_train_ids,
word_train_lines,

transform=train_transforms)
    val_dataset = HandwritingDataset(X_val, y_val_enc,
val_label_lengths, word_val_ids, word_val_lines,

transform=val_transforms)

    train_loader = DataLoader(train_dataset,
batch_size=batch_size, shuffle=True,
collate_fn=collate_fn, num_workers=4,
                              pin_memory=True)
    val_loader = DataLoader(val_dataset,
batch_size=batch_size, shuffle=False,
collate_fn=collate_fn, num_workers=4,
                            pin_memory=True)
```

Начинается всё с определения класса HandwritingDataset, который наследуется от базового класса Dataset из библиотеки torch.utils.data. Этот пользовательский класс предназначен для организации данных, специфичных для задачи распознавания рукописного текста. Внутри класса определяется метод __init__, который инициализирует основные атрибуты экземпляра. Он принимает списки изображений, закодированных меток, длин меток, идентификаторов слов и полных строк из файла words.txt, а также необязательный параметр transform, отвечающий за преобразования данных. Эти атрибуты сохраняются как экземплярные переменные, что позволяет последующим методам класса иметь к ним доступ.

Метод __len__ возвращает количество образцов в наборе данных, основываясь на длине списка изображений. Это важно для правильного функционирования загрузчиков данных, которые используют эту информацию для определения числа наборов данных и итераций во время обучения.

Метод __getitem__ отвечает за получение конкретного образца данных по индексу idx. Внутри этого метода извлекаются изображение, метка, длина метки, идентификатор слова и полная строка, соответствующие заданному индексу. Если для набора данных задано преобразование (transform), оно применяется к изображению, обеспечивая тем самым возможность динамической аугментации данных непосредственно во время загрузки. В противном случае изображение преобразуется в тензор с типом данных float32 и добавляется дополнительная размерность с помощью метода unsqueeze(0), что соответствует ожидаемому формату [1, H, W] для последующей подачи в модель. В конце метод возвращает кортеж, содержащий изображение, метку, длину метки, идентификатор слова и полную строку, что позволяет сохранить всю необходимую информацию для обучения и анализа модели.

Следующим важным элементом является определение функции collate_fn, которая служит пользовательской функцией объединения групп данных. Эта функция необходима для обработки пакетов данных с переменной длиной меток, что типично для задач распознавания последовательностей. Внутри функции происходит распаковка элементов на отдельные компоненты: изображения, метки, длины меток, идентификаторы слов и полные строки.

Изображения объединяются в единый тензор с помощью функции torch.stack, что позволяет сформировать набор данных размером [batch, 1, H, W]. Метки, которые представлены списками числовых последовательностей, конкатенируются в одну длинную последовательность с помощью цикла for, после чего преобразуются в тензор типа long. Длины меток также преобразуются в тензоры, что необходимо для корректного вычисления функции потерь CTC.

Кроме того, определяется тензор input_lengths, который фиксирует длину последовательностей на выходе CNN, предполагая, что ширина после прохождения через сверточные слои составляет 34 пикселя для изображений размером 160x40 пикселей. Это значение используется в функции потерь для согласования длин входных и выходных последовательностей. В конце функция возвращает кортеж, содержащий набор изображений, объединённые метки, длины входных последовательностей, длины меток, идентификаторы слов и полные строки, что обеспечивает полную информацию для дальнейшего обучения модели.

Далее устанавливается размер набора данных, равный 64, что определяет количество образцов, обрабатываемых одновременно в одном шаге обучения. Такой размер обеспечивает баланс между эффективностью использования вычислительных ресурсов и стабильностью градиентного спуска.

Создаются экземпляры классов HandwritingDataset для обучающей и валидационной выборок с помощью переменных X_train, y_train_enc,

train_label_lengths, word_train_ids, word_train_lines для обучения и соответствующих переменных для валидации. Для обучающей выборки применяется набор преобразований train_transforms, включающий аугментации, направленные на увеличение разнообразия данных и улучшение обобщающей способности модели. Валидационная выборка использует набор преобразований val_transforms, который включает только базовые преобразования без аугментаций, что позволяет объективно оценивать производительность модели на неизменённых данных.

После создания экземпляров наборов данных, инициализируются загрузчики данных train_loader и val_loader с помощью класса DataLoader из библиотеки torch.utils.data. Для обучающего загрузчика данных устанавливается параметр shuffle=True, что обеспечивает случайное перемешивание данных перед каждой эпохой обучения, предотвращая возможное переобучение на фиксированном порядке данных. Параметр collate_fn устанавливается на ранее определённую функцию collate_fn, что позволяет корректно объединять наборы данных с переменной длиной меток. Дополнительно устанавливаются параметры num_workers=4 и pin_memory=True, которые оптимизируют использование многопоточности и памяти, ускоряя процесс загрузки данных на устройства с поддержкой CUDA.

Для валидационного загрузчика данных параметры аналогичны, за исключением shuffle=False, что позволяет сохранить порядок данных для объективной оценки модели. Таким образом, загрузчики данных обеспечивают эффективную и гибкую подачу данных в модель, учитывая особенности задачи распознавания рукописного текста и требования к обработке последовательностей переменной длины.

В целом, эта часть кода тщательно организует процесс подготовки данных, обеспечивая корректную загрузку, предварительную обработку и эффективное объединение наборов для обучения и валидации модели. Пользовательский класс HandwritingDataset и функция collate_fn предоставляют необходимую гибкость и контроль над процессом подачи данных, что является важным аспектом при работе с задачами распознавания последовательностей и глубокого обучения.

3.5. Создание модели

Далее определяется класс CRNN, который реализует сверточно-рекуррентную нейронную сеть (Convolutional Recurrent Neural Network) для задачи распознавания рукописного текста. Этот класс наследуется от базового

класса nn.Module из библиотеки PyTorch, что позволяет использовать все возможности фреймворка для построения и обучения нейронных сетей.

```
    class CRNN(nn.Module):
        def __init__(self, imgH, nc, nclass, nh):
            super(CRNN, self).__init__()
            self.cnn = nn.Sequential(
                nn.Conv2d(nc, 64, kernel_size=3, stride=1,
padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(2, 2),

                nn.Conv2d(64,      128,      kernel_size=3,
stride=1, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(2, 2),

                nn.Conv2d(128,     256,      kernel_size=3,
stride=1, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.MaxPool2d((2,  2),  (2,  1),  padding=(0,
1)),

                nn.Conv2d(256,     256,      kernel_size=3,
stride=1, padding=1),  # [batch, 256, H/8, W/4 +1]
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.MaxPool2d((2,  2),  (2,  1),  padding=(0,
1)),
            )

            self.rnn_linear = nn.Linear(256 * (imgH // 16),
nh)
            self.rnn_relu = nn.ReLU(inplace=True)
            self.dropout = nn.Dropout(p=0.5)
        self.lstm1 = nn.LSTM(nh, nh, bidirectional=True)
```

```
        self.lstm2 = nn.LSTM(nh * 2, nh, bidirectional=True)
        self.fc = nn.Linear(nh * 2, nclass)

    def forward(self, x):
        conv = self.cnn(x)
        b, c, h, w = conv.size()
        conv = conv.permute(3, 0, 1, 2)
        conv = conv.contiguous().view(w, b, -1)
        conv = self.rnn_linear(conv)
        conv = self.rnn_relu(conv)
        conv = self.dropout(conv)

        output, _ = self.lstm1(conv)
        output, _ = self.lstm2(output)


        output = self.fc(output)
        return output.log_softmax(2)
```

Внутри класса начинается с метода __init__, который инициализирует все слои и компоненты сети. Первый компонент — это сверточная часть сети, определяемая с помощью последовательности слоев nn.Sequential. Сеть начинается с двухмерного сверточного слоя nn.Conv2d, который принимает на вход nc каналов (в данном случае 1, так как изображения в градациях серого) и преобразует их в 64 канала с помощью ядра размером 3x3, шага 1 и паддинга 1. Следом за сверточным слоем идёт слой пакетной нормализации nn.BatchNorm2d(64), который стабилизирует распределение активаций, ускоряя и улучшая процесс обучения. Затем применяется функция активации ReLU (nn.ReLU(inplace=True)), которая вводит нелинейность в модель, позволяя ей обучаться более сложным функциям.

После этого следует слой максимального объединения nn.MaxPool2d(2, 2), который уменьшает пространственные размеры признаков вдвое, сокращая вычислительные затраты и позволяя модели фокусироваться на более крупных структурах в изображении. Этот паттерн повторяется во втором сверточном блоке, где количество каналов увеличивается до 128, сохраняя ядро и параметры шага, что позволяет модели захватывать более сложные и высокоуровневые признаки.

Третий и четвёртый сверточные блоки расширяют количество каналов до 256 и снова используют пакетную нормализацию, ReLU и максимальное объединение. Однако, здесь используются более сложные параметры для слоя объединения: (2, 2) для размера окна и (2, 1) для шага с паддингом (0, 1). Это приводит к изменению пространственных размеров по осям высоты и ширины неравномерно, что важно для сохранения информации о последовательности символов в изображении при уменьшении его размера.

После сверточной части сети переходим к рекуррентным слоям. Сначала используется линейный слой self.rnn_linear, который преобразует выходные данные сверточной сети из размерности 256 * (imgH // 16) (где imgH — высота изображения) в размерность nh, равную 1024. Этот слой служит для подготовки данных к подаче в рекуррентные слои. Далее следует функция активации ReLU и слой регуляризации nn.Dropout(p=0.5), который помогает предотвратить переобучение, случайным образом обнуляя некоторые активации во время обучения.

Рекуррентная часть сети состоит из двух слоёв LSTM (self.lstm1 и self.lstm2), каждый из которых двунаправленный (bidirectional=True). Первый слой LSTM принимает на вход данные размерности nh и выводит данные размерности nh * 2, поскольку он двунаправленный. Второй слой LSTM принимает данные размерности nh * 2 и снова выводит данные размерности nh * 2, что позволяет модели захватывать как предшествующие, так и последующие контексты в последовательности символов.

После прохождения через рекуррентные слои данные подаются на полностью связанный слой self.fc, который преобразует выходные данные из размерности nh * 2 в размерность nclass, соответствующую количеству классов (символов) плюс один для пустого символа, используемого в функции потерь CTC (Connectionist Temporal Classification). Финальным шагом является применение функции логарифмического софтмакса log_softmax(2), которая вычисляется по классовой оси, преобразуя выходные данные в логарифмы вероятностей для каждого класса.

Метод forward определяет прямой проход данных через сеть. Сначала входное изображение x, имеющее форму [batch, 1, H, W], проходит через сверточную часть сети self.cnn, преобразуясь в тензор размерности [batch, 256, H/16, W'], где W' зависит от исходной ширины изображения и параметров слоев объединения. Далее полученный тензор переставляется с помощью метода permute в размерность [W', batch, 256, H/16], что подготавливает данные к подаче в рекуррентные слои, обрабатывающие последовательности.

Следующим шагом тензор преобразуется методом contiguous().view(w, b, -1) в форму [W', batch, 256 * H/16], что соответствует формату, необходимому для

линейного слоя и последующих рекуррентных слоёв. После этого данные проходят через линейный слой self.rnn_linear, функцию активации ReLU и слой Dropout, после чего подаются на первый LSTM слой self.lstm1. Выходные данные из первого LSTM слоя поступают на второй LSTM слой self.lstm2, после чего проходят через полностью связанный слой self.fc, преобразуя их в логарифмы вероятностей по каждому классу.

Таким образом, класс CRNN сочетает в себе возможности сверточных слоёв для извлечения пространственных признаков из изображений и рекуррентных слоёв для обработки последовательной информации, что особенно полезно в задачах распознавания текста, где порядок символов имеет значение. Архитектура сети позволяет эффективно обрабатывать изображения рукописного текста, извлекая из них релевантные признаки и последовательно распознавая символы, что делает её мощным инструментом для задач оптического распознавания символов (OCR).

3.6. Инициализация параметров модели

Теперь необходимо произвести инициализацию параметров модели, настройку устройства для вычислений (CPU или GPU), создание экземпляра модели CRNN, а также инициализацию весов нейронных сетей с использованием метода инициализации Xavier. Рассмотрим каждую из этих операций подробно.

Сначала определяется устройство для вычислений с помощью строки

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Этот код проверяет, доступен ли графический процессор (GPU) с поддержкой CUDA. Если CUDA доступна, устройство устанавливается на 'cuda', что позволяет использовать GPU для ускорения вычислений. В противном случае устройство устанавливается на 'cpu'. Такая настройка обеспечивает гибкость и оптимизацию процесса обучения, позволяя автоматически использовать наиболее производительное доступное оборудование.

Далее задаются основные параметры модели:

```
imgH = image_height
nc = 1
nclass = len(chars) + 1
nh = 1024
```

Переменная imgH устанавливается равной значению image_height, которое определяет высоту входных изображений в пикселях. Параметр nc задаёт количество входных каналов изображений, равное 1, поскольку изображения преобразованы в градации серого, и, следовательно, имеют только один канал.

Параметр nclass определяется как количество уникальных символов в наборе данных (len(chars)) плюс один дополнительный класс, предназначенный для пустого символа, используемого в функции потерь CTC (Connectionist Temporal Classification). Это необходимо для корректного расчёта вероятностей классов и обработки последовательностей символов различной длины.

Параметр nh устанавливается равным 1024, что определяет количество скрытых единиц в рекуррентных слоях модели. Увеличение этого значения улучшает ёмкость модели, позволяя ей захватывать более сложные зависимости в данных, но при этом может привести к увеличению времени обучения и потребления памяти.

После определения параметров создаётся экземпляр модели CRNN и перемещается на выбранное устройство с помощью метода .to(device):

```
model = CRNN(imgH, nc, nclass, nh).to(device)
```

Конструктор класса CRNN принимает четыре аргумента: высоту изображения imgH, количество входных каналов nc, количество классов nclass и размер скрытого слоя nh. Созданный экземпляр модели затем переносится на устройство, определённое ранее (GPU или CPU), что позволяет эффективно использовать вычислительные ресурсы для обучения и инференса.

Далее определяется функция init_weights, предназначенная для инициализации весов сверточных и линейных слоёв модели:

```
def init_weights(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)
```

Функция init_weights принимает один аргумент m, который представляет собой слой нейронной сети. Внутри функции осуществляется проверка типа слоя с помощью isinstance. Если слой является экземпляром класса nn.Conv2d

(сверточный слой) или nn.Linear (полносвязный слой), то его веса инициализируются с использованием метода xavier_uniform_ из модуля torch.nn.init. Этот метод устанавливает веса слоя равномерно распределёнными значениями в соответствии с инициализацией Xavier (также известной как Glorot), которая способствует стабильности градиентов во время обучения, особенно в глубоких сетях.

Кроме инициализации весов, проверяется наличие смещений (bias) в слое. Если смещения присутствуют (m.bias is not None), они инициализируются константой 0 с помощью метода nn.init.constant_. Это стандартная практика, направленная на устранение начальных смещений, что может способствовать более быстрому и стабильному обучению модели.

После определения функции инициализации весов, она применяется ко всей модели с помощью метода apply:

```
model.apply(init_weights)
```

Метод apply рекурсивно применяет переданную функцию init_weights ко всем подслоям модели model. Это гарантирует, что все сверточные и линейные слои модели будут корректно инициализированы перед началом процесса обучения. Правильная инициализация весов является критически важным шагом, который может значительно повлиять на скорость сходимости и общую эффективность обучения модели.

Таким образом, данный блок кода обеспечивает настройку устройства для вычислений, определение основных параметров модели, создание и подготовку экземпляра модели CRNN, а также инициализацию её весов с использованием подходящей схемы инициализации. Эти шаги являются фундаментальными для успешного обучения нейронной сети, обеспечивая её правильную структуру и подготовку к обучающему процессу.

В данном блоке кода производится настройка параметров обучения модели распознавания рукописного текста.

```
    criterion = nn.CTCLoss(blank=0, zero_infinity=True)
    optimizer = optim.Adam(model.parameters(), lr=0.0001,
weight_decay=1e-4)
    scheduler                                          =
optim.lr_scheduler.ReduceLROnPlateau(optimizer,
mode='min', factor=0.1, patience=3, verbose=True)
```

```
max_grad_norm = 5
num_epochs = 200
patience = 15
best_val_loss = float('inf')
trigger_times = 0
train_losses = []
val_losses = []
val_cers = []
```

Сначала определяется функция потерь и оптимизатор. Используется критерий CTCLoss, который особенно подходит для задач распознавания последовательностей, таких как распознавание текста. Параметр blank установлен в значение 0, что обозначает символ пустоты, используемый в алгоритме CTC (Connectionist Temporal Classification). Параметр zero_infinity=True позволяет игнорировать случаи, когда значение потерь стремится к бесконечности, что может происходить при невозможности выровнять входную и целевую последовательности.

Оптимизатором выбран метод Adam, который является одним из наиболее эффективных алгоритмов для обучения нейронных сетей благодаря адаптивному изменению скорости обучения. Начальная скорость обучения (lr) установлена на уровне 0.0001, что обеспечивает постепенное обновление весов модели. Дополнительно применяется параметр weight_decay со значением 1e-4 для регуляризации, предотвращая переобучение путем добавления штрафа за большие веса.

Далее, используется планировщик скорости обучения (scheduler) ReduceLROnPlateau. Этот планировщик уменьшает скорость обучения на порядок (фактор factor=0.1), если наблюдается отсутствие улучшений в значении функции потерь на валидационной выборке в течение заданного количества эпох (patience=3). Параметр verbose=True включает вывод информации о снижении скорости обучения, что помогает отслеживать процесс обучения модели.

Для предотвращения чрезмерного роста градиентов применяется обрезка градиентов (gradient clipping). Параметр max_grad_norm установлен на значение 5, что ограничивает норму градиентов и помогает стабилизировать процесс обучения, особенно в глубоких нейронных сетях с рекуррентными слоями.

Инициализируются параметры для ранней остановки (Early Stopping), что позволяет прекратить обучение модели, если улучшение качества на валидационной выборке прекращается. Общее количество эпох (num_epochs) увеличено до 200, предоставляя модели больше возможностей для обучения и

достижения лучшей сходимости. Параметр patience установлен на значение 15, что означает, что обучение будет продолжаться до тех пор, пока значение функции потерь на валидации не улучшится в течение 15 последовательных эпох. Переменная best_val_loss инициализируется бесконечностью и будет обновляться по мере улучшения функции потерь на валидационной выборке. Счетчик trigger_times отслеживает количество эпох, прошедших без улучшения, и используется для принятия решения о прекращении обучения при достижении заданного порога терпимости.

Наконец, создаются три пустые переменные train_losses, val_losses и val_cers, которые будут использоваться для хранения значений функции потерь на тренировочной и валидационной выборках, а также показателя ошибки символов (CER) для последующей визуализации и анализа динамики обучения модели.

Этот блок кода обеспечивает гибкую и эффективную настройку процесса обучения, позволяя модели адаптироваться к данным, предотвращать переобучение и стабилизировать обучение через регуляризацию и планирование скорости обучения.

3.7. Функция decode_predictions и Цикл Обучения

В данном блоке кода реализуется функция decode_predictions, отвечающая за декодирование выходных данных модели, а также основной цикл обучения модели, включающий как тренировочную, так и валидационную фазы.

Функция decode_predictions выполняет жадное декодирование (greedy decoding) выходных данных модели для получения предсказанных последовательностей символов. Она принимает на вход тензор outputs, содержащий логарифмические вероятности для каждого класса символов, имеющий форму [W', batch, nclass], где W' — ширина выходного признакового пространства, batch — размер пакета, а nclass — количество классов (включая символ пустоты).

```
def decode_predictions(outputs):
    decoded_sequences = []
    outputs = outputs.transpose(0, 1)
    for output in outputs:
        probs = output.detach().cpu()
        max_indices = probs.argmax(dim=1)
        max_indices = max_indices.numpy()
        decoded = []
        prev_idx = -1
```

```
            for idx in max_indices:
                if idx != prev_idx and idx != 0:
                    decoded.append(idx)
                prev_idx = idx
            decoded_sequences.append(decoded)
        return decoded_sequences
```

Внутри функции происходит транспонирование тензора outputs с помощью outputs.transpose(0, 1), что изменяет его форму на [batch, W', nclass]. Это упрощает дальнейшую обработку предсказаний для каждого образца в пакете. Затем для каждого выходного тензора output определяется индекс класса с максимальной вероятностью по каждому временно́му шагу с помощью probs.argmax(dim=1). Полученные индексы преобразуются в массивы с помощью max_indices.numpy(). Далее выполняется удаление повторяющихся последовательных дубликатов и символов пустоты (индекс 0), что реализуется в цикле:

```
decoded = []
prev_idx = -1
for idx in max_indices:
    if idx != prev_idx and idx != 0:
        decoded.append(idx)
    prev_idx = idx
```

Результатом является список decoded_sequences, содержащий декодированные последовательности для каждого образца в пакете.

Основной цикл обучения модели начинается с итерации по количеству заданных эпох (num_epochs). Каждая эпоха включает тренировочную и валидационную фазы.

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (images, labels, input_lengths,
label_lengths, word_ids_batch, full_lines_batch) in
enumerate(
            train_loader):
        images = images.to(device)
        labels = labels.to(device)
```

```
            input_lengths = input_lengths.to(device)
            label_lengths = label_lengths.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            log_probs = outputs
            loss     =     criterion(log_probs,     labels,
input_lengths, label_lengths)
            loss.backward()

torch.nn.utils.clip_grad_norm_(model.parameters(),
max_grad_norm)

            optimizer.step()

            running_loss += loss.item()
            if (batch_idx + 1) % 100 == 0:
                print(
                    f"Epoch  [{epoch  +  1}/{num_epochs}],
Step   [{batch_idx   +   1}/{len(train_loader)}],   Loss:
{loss.item():.4f}")

        avg_train_loss = running_loss / len(train_loader)
        train_losses.append(avg_train_loss)
        print(f"Epoch [{epoch + 1}/{num_epochs}], Average
Training Loss: {avg_train_loss:.4f}")
```

Тренировочная Фаза:

1. **Режим обучения:** Модель переводится в режим обучения с помощью model.train(), что включает использование слоев, таких как Dropout и BatchNorm, в режиме обучения.

2. **Итерация по наборам данных:** Для каждого набора из train_loader изображения и соответствующие метки перемещаются на устройство вычислений (CPU или GPU) с помощью .to(device).

3. **Обнуление градиентов:** Перед обратным распространением ошибки градиенты оптимизатора обнуляются методом optimizer.zero_grad().

4. **Прямой проход:** Модель обрабатывает входные изображения, генерируя предсказания outputs = model(images), имеющие форму [W', batch, nclass].

5. **Вычисление функции потерь:** Используется критерий CTCLoss для вычисления значения функции потерь:

```
loss = criterion(log_probs, labels, input_lengths, label_lengths)
```

6. **Обратное распространение ошибки:** Выполняется обратное распространение ошибки с помощью loss.backward().

7. **Обрезка градиентов:** Применяется обрезка градиентов для предотвращения их чрезмерного роста:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
```

8. **Обновление весов:** Оптимизатор обновляет веса модели методом optimizer.step().

9. **Накопление потерь:** Значение функции потерь добавляется к переменной running_loss.

10. **Мониторинг обучения:** Каждые 100 шагов выводится текущее значение функции потерь для отслеживания процесса обучения:

```
if (batch_idx + 1) % 100 == 0:
    print(
        f"Epoch [{epoch + 1}/{num_epochs}], Step [{batch_idx + 1}/{len(train_loader)}], Loss: {loss.item():.4f}")
```

11. **Средняя тренировочная потеря:** После завершения всех наборов данных в эпохе рассчитывается среднее значение тренировочной функции потерь и выводится:

```
avg_train_loss = running_loss / len(train_loader)
```

```
    train_losses.append(avg_train_loss)
    print(f"Epoch [{epoch + 1}/{num_epochs}], Average
Training Loss: {avg_train_loss:.4f}")
```

Валидационная фаза:

После тренировочной фазы начинается валидационная фаза, которая оценивает производительность модели на валидационной выборке.

```
    model.eval()
    with torch.no_grad():
       val_loss = 0.0
       total_cer = 0.0
       total_samples = 0
       misrecognized_words = []
       for images, labels, input_lengths, label_lengths,
word_ids_val, full_lines_val in val_loader:
           images = images.to(device)
           labels = labels.to(device)
           input_lengths = input_lengths.to(device)
           label_lengths = label_lengths.to(device)

           outputs = model(images)
           log_probs = outputs
           loss = criterion(log_probs, labels,
input_lengths, label_lengths)
           val_loss += loss.item(
           decoded_sequences =
decode_predictions(log_probs)

           for i in range(len(decoded_sequences)):
               pred_label = ''.join([num_to_char.get(idx,
'') for idx in decoded_sequences[i]])
               true_label = y_val_str[i]
               sample_wer = wer(true_label, pred_label)
               sample_cer = cer(true_label, pred_label)
               total_cer += sample_cer
               total_samples += 1
```

```
            if sample_wer > 0:
                misrecognized_words.append({
                    'word_id': word_val_ids[i],
                    'true_label': true_label,
                    'predicted_label': pred_label,
                    'full_line': word_val_lines[i],
                    'wer': sample_wer,
                    'cer': sample_cer
                })
```

1. **Режим оценки:** Модель переводится в режим оценки с помощью model.eval(), что отключает такие слои, как Dropout и BatchNorm, в режиме обучения.

2. **Отключение градиентов:** Используется контекстный менеджер with torch.no_grad(), который отключает вычисление градиентов для экономии памяти и ускорения вычислений.

3. **Итерация по валидационным данным:** Для каждого набора данных из val_loader изображения и метки перемещаются на устройство вычислений.

4. **Прямой проход:** Модель генерирует предсказания outputs = model(images).

5. **Вычисление функции потерь:** Вычисляется функция потерь на валидационной выборке:

```
    loss = criterion(log_probs, labels, input_lengths, label_lengths)
    loss.backward()
```

6. **Декодирование предсказаний:** Предсказания декодируются с помощью функции decode_predictions:

```
    decoded_sequences = decode_predictions(log_probs)
```

7. **Вычисление показателей ошибок:** Для каждого предсказанного образца рассчитываются показатели Word Error Rate (WER) и Character Error Rate (CER):

```
    pred_label = ''.join([num_to_char.get(idx, '') for idx
in decoded_sequences[i]])
    true_label = y_val_str[i]
    sample_wer = wer(true_label, pred_label)
    sample_cer = cer(true_label, pred_label)   # Compute
CER
    total_cer += sample_cer
    total_samples += 1
```

8. **Сохранение неверно распознанных слов:** Если WER превышает ноль, информация о неверно распознанном слове сохраняется в список misrecognized_words для последующего анализа:

```
    if sample_wer > 0:
       misrecognized_words.append({
           'word_id': word_val_ids[i],
           'true_label': true_label,
           'predicted_label': pred_label,
           'full_line': word_val_lines[i],
           'wer': sample_wer,
           'cer': sample_cer
       })
```

После обработки всех наборов данных валидационной выборки рассчитываются средние значения валидационной функции потерь и CER, которые добавляются в соответствующие списки и выводятся для мониторинга эффективности обучения:

```
    avg_val_loss = val_loss / len(val_loader)
    avg_cer = total_cer / total_samples
    val_losses.append(avg_val_loss)
    val_cers.append(avg_cer)  # Store CER
    print(f"Epoch [{epoch + 1}/{num_epochs}], Validation
Loss: {avg_val_loss:.4f}, Validation CER: {avg_cer:.4f}")
```

Обновление планировщика скорости обучения и проверка ранней остановки:

После валидационной фазы планировщик скорости обучения обновляется на основе среднего значения валидационной потери. Если текущее значение валидационной потери лучше предыдущего наилучшего (best_val_loss), модель сохраняется на диск, и счетчик trigger_times сбрасывается. В противном случае счетчик увеличивается, и при достижении заданного порога терпимости (patience) обучение прерывается досрочно (Early Stopping), что предотвращает переобучение модели.

```
    scheduler.step(avg_val_loss)

    if avg_val_loss < best_val_loss:
       best_val_loss = avg_val_loss
       torch.save(model.state_dict(),
'best_crnn_model.pth')
       print("Model saved!")
       trigger_times = 0
    else:
       trigger_times += 1
       print(f"Trigger times: {trigger_times}")
       if trigger_times >= patience:
           print("Early stopping!")
           break
```

Сохранение информации о неверно распознанных словах:

В конце каждой эпохи информация о неверно распознанных словах сохраняется в файл с именем, содержащим номер текущей эпохи. Это позволяет проводить детальный анализ ошибок модели и вносить необходимые коррективы в процесс обучения или архитектуру модели.

```
    with     open(f'misrecognized_words_epoch_{epoch     +
1}.txt', 'w') as f:
       for word in misrecognized_words:
           f.write(f"Word ID: {word['word_id']}\n")
           f.write(f"Full Line: {word['full_line']}\n")
           f.write(f"True Label: {word['true_label']}\n")
           f.write(f"Predicted                        Label:
{word['predicted_label']}\n")
```

```
            f.write(f"Word Error Rate: {word['wer']}\n")
            f.write(f"Character Error Rate:
{word['cer']}\n")
            f.write('---\n')
```

Итоговое описание цикла обучения:

1. **Начало эпохи:** Каждая эпоха начинается с перехода модели в режим обучения и инициализации переменной running_loss.

2. **Тренировочная итерация:** Для каждого набора выполняются прямой и обратный проходы, вычисляется функция потерь, применяется обрезка градиентов, и обновляются веса модели. Накопленные потери выводятся каждые 100 шагов.

3. **Валидация:** После тренировочной фазы модель переводится в режим оценки, и выполняется вычисление функции потерь на валидационной выборке. Предсказания декодируются, и рассчитываются показатели WER и CER. Неверно распознанные слова сохраняются для анализа.

4. **Обновление метрик и планировщика:** Средние значения валидационной потери и CER сохраняются и выводятся. Планировщик скорости обучения обновляется на основе валидационной потери.

5. **Ранняя остановка:** Если валидационная потеря не улучшается в течение заданного числа эпох (patience), обучение прерывается досрочно для предотвращения переобучения.

6. **Сохранение ошибок:** Информация о неверно распознанных словах сохраняется в файл для дальнейшего анализа.

Этот блок кода обеспечивает эффективное обучение модели распознавания рукописного текста, обеспечивая мониторинг процесса обучения, адаптацию параметров обучения и сохранение лучших весов модели для последующего использования.

3.8. Визуализация результатов обучения

В этом блоке кода осуществляется визуализация результатов обучения модели с помощью библиотеки Matplotlib. Визуализируются изменения функции потерь как на тренировочной, так и на валидационной выборках, а также изменение показателя ошибки символов (CER) на валидационной выборке в зависимости от числа эпох обучения.

Первым шагом создается фигура с заданными размерами, обеспечивающими достаточное пространство для размещения графиков:

```
plt.figure(figsize=(18, 6))
```

Затем создается первый подграфик, предназначенный для отображения значений функции потерь на тренировочной и валидационной выборках. Используется метод plt.subplot(1, 2, 1), который делит фигуру на 1 строку и 2 столбца, выбирая первый подграфик для построения:

```
plt.plot(range(1, len(train_losses) + 1), train_losses, label='Training Loss', color='blue')
plt.plot(range(1, len(val_losses) + 1), val_losses, label='Validation Loss', color='orange')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.legend()
```

В этом вложенном графике:

range(1, len(train_losses) + 1) генерирует последовательность номеров эпох от 1 до количества записанных значений функции потерь.

train_losses и val_losses представляют собой списки, содержащие значения функции потерь на каждой эпохе для тренировочной и валидационной выборок соответственно.

label задает название для каждой кривой, которое будет отображено в легенде.

color определяет цвет линии графика, что помогает различать тренировочную и валидационную потери.

Метки осей xlabel и ylabel указывают, что по оси X отложены номера эпох, а по оси Y — значения функции потерь.

Заголовок графика plt.title описывает содержание графика.

Метод plt.legend() добавляет легенду, позволяя легко идентифицировать, какая кривая соответствует тренировочной, а какая — валидационной выборке.

Следующим шагом создается второй подграфик для отображения динамики показателя Character Error Rate (CER) на валидационной выборке:

```
    plt.subplot(1, 2, 2)
    plt.plot(range(1, len(val_cers) + 1), val_cers,
label='Validation CER', color='green')
    plt.xlabel('Epochs')
    plt.ylabel('Character Error Rate (CER)')
    plt.title('Validation CER Over Epochs')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

В этом подграфике:

range(1, len(val_cers) + 1) генерирует последовательность номеров эпох от 1 до количества записанных значений CER. val_cers представляет собой список, содержащий значения CER на каждой эпохе для валидационной выборки.

label задает название для кривой CER, которое будет отображено в легенде.

color определяет цвет линии графика, отличающийся от цвета графика функции потерь. Метки осей и заголовок аналогично описывают содержимое графика. Метод plt.legend() добавляет легенду для идентификации кривой CER.

После построения обоих подграфиков вызывается метод plt.tight_layout(), который автоматически настраивает параметры подграфиков для предотвращения наложения элементов и обеспечения оптимального размещения:

```
    plt.tight_layout()
```

Наконец, графики отображаются на экране с помощью команды plt.show(), позволяя визуально оценить процесс обучения модели:

```
    plt.show()
```

Таким образом, данный блок кода предоставляет наглядное представление о динамике обучения модели, позволяя отслеживать снижение функции потерь на тренировочной и валидационной выборках, а также улучшение показателя CER на валидационной выборке. Это способствует более глубокому пониманию процесса обучения, выявлению возможных проблем, таких как переобучение

или недообучение, и принятию обоснованных решений по дальнейшему улучшению модели.

3.9. Оценка модели на валидационной выборке

В этом блоке кода осуществляется оценка производительности обученной модели на валидационной выборке. Процесс включает перевод модели в режим оценки, отключение вычисления градиентов для повышения эффективности, декодирование предсказаний модели, вычисление показателей ошибок и сохранение информации о неправильно распознанных словах для последующего анализа.

Перевод модели в режим оценки и отключение градиентов

```
model.eval()
with torch.no_grad():
```

Первоначально модель переводится в режим оценки с помощью метода model.eval(). Это переключает модель в режим, при котором такие слои, как Dropout и BatchNorm, функционируют в режиме вывода, что обеспечивает детерминированное поведение модели. Использование контекстного менеджера torch.no_grad() отключает вычисление градиентов, что снижает потребление памяти и ускоряет выполнение операций, так как градиенты не нужны во время оценки.

Инициализация переменных для накопления метрик

```
total_wer = 0.0
total_cer = 0.0
total_samples = 0
misrecognized_words = []
```

Здесь инициализируются переменные для накопления суммарных значений **Word Error Rate (WER)** и **Character Error Rate (CER)**, а также счетчик общего числа обработанных образцов. Список misrecognized_words предназначен для хранения информации о неправильно распознанных словах, что позволяет проводить детальный анализ ошибок модели.

Итерация по валидационной выборке

```
    for images, labels, input_lengths, label_lengths,
word_ids_val, full_lines_val in val_loader:
        images = images.to(device)
```

```
        outputs = model(images)
        log_probs = outputs
```

Цикл for проходит по всем наборам валидационной выборки, предоставляемой val_loader. Каждое изображение перемещается на устройство вычислений (CPU или GPU) с помощью метода images.to(device). Затем модель обрабатывает изображения, генерируя предсказания outputs, которые представляют собой логарифмические вероятности для каждого класса символов. Эти предсказания имеют форму [W', batch, nclass], где W' — ширина выходного признакового пространства, batch — размер набора, а nclass — количество классов (включая символ пустоты).

Декодирование предсказаний с помощью “жадного” декодирования

```
decoded_sequences = decode_predictions(log_probs)
```

Функция decode_predictions выполняет жадное декодирование выходных данных модели, преобразуя логарифмические вероятности в последовательности индексов классов. Результатом является список декодированных последовательностей для каждого образца в наборе данных. Это позволяет преобразовать числовые предсказания модели обратно в текстовые метки.

Обработка каждого предсказанного образца

```
        for i in range(len(decoded_sequences)):
            pred_label = ''.join([num_to_char.get(idx, '') for
idx in decoded_sequences[i]])
            true_label = y_val_str[i]
            sample_wer = wer(true_label, pred_label)
            sample_cer = cer(true_label, pred_label)
            total_wer += sample_wer
            total_cer += sample_cer
            total_samples += 1

            if sample_wer > 0:
                misrecognized_words.append({
                    'word_id': word_val_ids[i],
                    'true_label': true_label,
                    'predicted_label': pred_label,
```

```
                'full_line': word_val_lines[i],
                'wer': sample_wer,
                'cer': sample_cer
            })

        print(f"True label: {true_label}")
        print(f"Predicted: {pred_label}")
        print(f"Word Error Rate: {sample_wer}")
        print(f"Character Error Rate: {sample_cer}")
        print('---')
    break
```

Для каждого декодированного предсказания выполняются следующие шаги:

Предсказанные индексы классов преобразуются обратно в символы с помощью словаря num_to_char, после чего формируется строка предсказанного слова pred_label. Истинная метка извлекается из списка y_val_str по соответствующему индексу. Затем с помощью функций wer и cer вычисляются показатели **Word Error Rate (WER)** и **Character Error Rate (CER)** для текущего предсказания. Эти значения добавляются к суммарным показателям ошибок total_wer и total_cer, а также увеличивается счетчик обработанных образцов total_samples.

Если **WER** для текущего предсказания больше нуля, информация о неправильно распознанном слове добавляется в список misrecognized_words. Это включает идентификатор слова, полную строку из файла words.txt, истинную и предсказанную метки, а также вычисленные значения **WER** и **CER**. Далее выводится информация о текущем предсказании, включая истинную метку, предсказанную метку и соответствующие показатели ошибок. Цикл прерывается после обработки первого набора данных с помощью оператора break, что позволяет ограничить объем выводимой информации для проверки работы кода без необходимости обработки всей валидационной выборки.

Вычисление средних значений показателей ошибок

```
average_wer = total_wer / total_samples
average_cer = total_cer / total_samples
print(f"Average Word Error Rate on Validation Set: {average_wer:.4f}")
print(f"Average Character Error Rate on Validation Set: {average_cer:.4f}")
```

После завершения итерации по валидационной выборке рассчитываются средние значения **WER** и **CER**. Эти показатели предоставляют обобщенную оценку качества модели на валидационной выборке, показывая, насколько часто слова и символы распознаются неправильно в среднем по всем обработанным образцам.

Сохранение информации о неправильно распознанных словах

```
with open('misrecognized_words.txt', 'w') as f:
    for word in misrecognized_words:
        f.write(f"Word ID: {word['word_id']}\n")
        f.write(f"Full Line: {word['full_line']}\n")
        f.write(f"True Label: {word['true_label']}\n")
        f.write(f"Predicted Label: {word['predicted_label']}\n")
        f.write(f"Word Error Rate: {word['wer']}\n")
        f.write(f"Character Error Rate: {word['cer']}\n")
        f.write('---\n')
```

Список misrecognized_words, содержащий информацию о всех неправильно распознанных словах, сохраняется в файл misrecognized_words.txt. Каждое неправильно распознанное слово записывается в файл в формате, содержащем идентификатор слова, полную строку из файла words.txt, истинную и предсказанную метки, а также соответствующие значения **WER** и **CER**. Это позволяет провести детальный анализ ошибок модели, выявить закономерности и определить направления для дальнейшего улучшения качества распознавания.

В этом блоке кода осуществляется комплексная оценка производительности модели на валидационной выборке. Модель переводится в режим оценки, что отключает специфичные для обучения слои и оптимизирует процесс вычислений. В цикле итерации по валидационным данным производится декодирование предсказаний модели, вычисление показателей ошибок для каждого образца, накопление суммарных значений этих показателей и сохранение информации о неправильно распознанных словах. После обработки всех данных рассчитываются средние значения **WER** и **CER**, которые выводятся для оценки общего качества модели. Дополнительно информация о неправильно распознанных словах сохраняется в файл, что позволяет проводить детальный анализ и выявлять слабые места модели для дальнейшего улучшения.

3.10. Пример выходных данных

Ниже продемонстрированы выходные данные по итогу обучения модели на наборе данных.

В консоль выводятся следующие данные

Average Word Error Rate on Validation Set: 0.2500
Average Character Error Rate on Validation Set: 0.1044

В данном выводе главной меткой качества обучения модели является Average CER. В данном случае средняя ошибка в распознавания символа составляет 10%, что является хорошим результатом без углубленной подстройки.

Также по итогам обучения код предоставляет график для отслеживания динамики ошибки между эпохами обучения. График отображен ниже на рисунке 3.

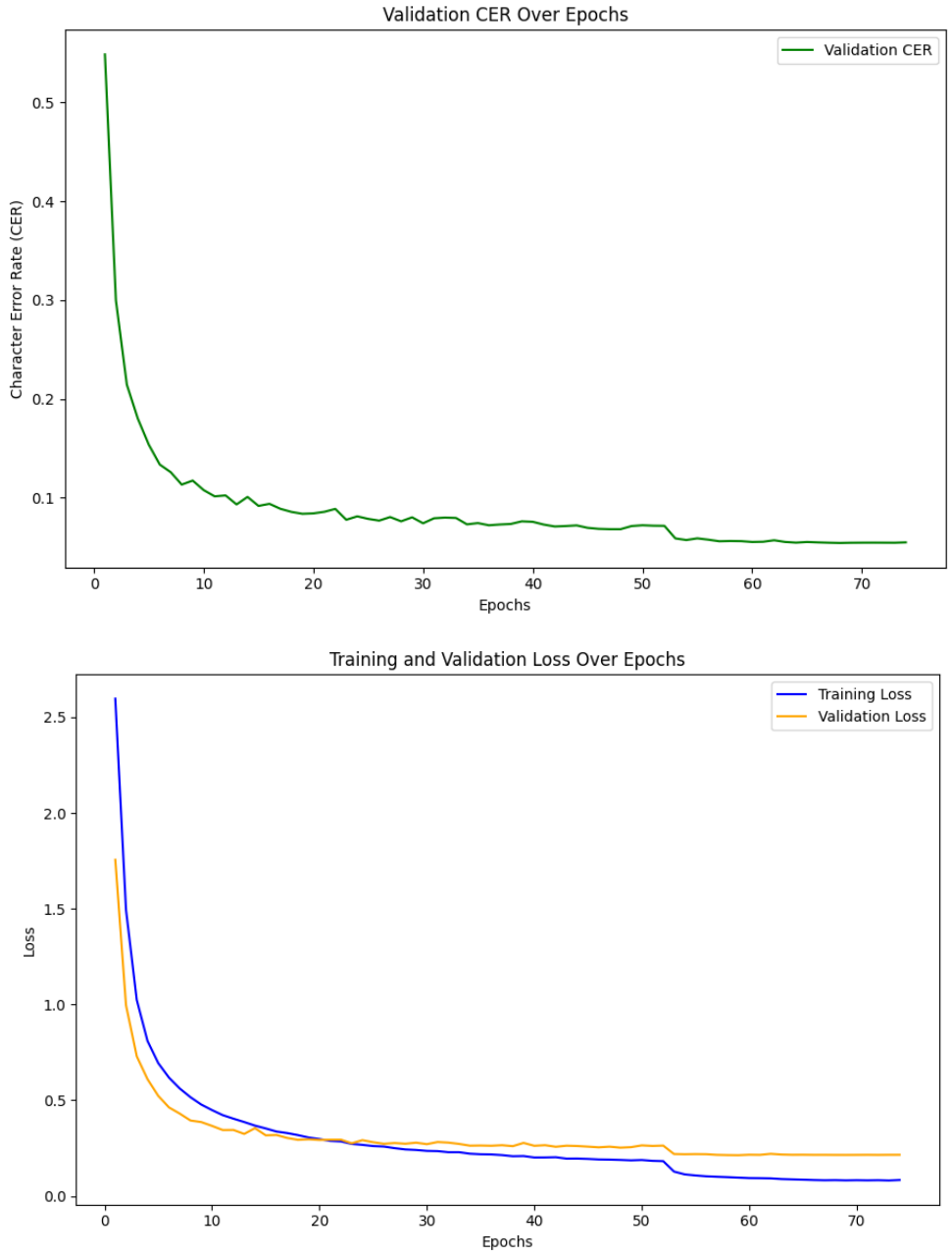

Рисунок 3. Отображение динамики обучения модели между эпохами.

На рисунке график слева отображает кривые для отслеживания того, сколько ошибок совершает модель в каждую эпоху обучения пословно. Слово с ошибкой - 0, слово без ошибки - 1. Далее считается средняя ошибка за эпоху. Синий график отображает изменение потерь для обучающего набора. Оранжевый отображает ошибку для валидационного набора. График справа отображает среднюю ошибку по символам. Неправильно определенный символ - 0, верно определенный символ - 1. Далее высчитывается среднее по набору в эпохе.

4. Дальнейшие рекомендации

В процессе разработки и обучения нейронной сети для распознавания рукописного текста были достигнуты хорошие результаты, однако для улучшения производительности модели и повышения её адаптивности в различных условиях можно реализовать несколько дополнительных шагов. В данном разделе приведены рекомендации по улучшению архитектуры модели, предобработке данных, настройке гиперпараметров и внедрению более сложных методов декодирования и регуляризации. Эти подходы помогут повысить точность распознавания, а также улучшить общее качество работы сети при обработке реальных данных. Реализация предложенных рекомендаций создаст возможность для более эффективного применения модели в различных прикладных задачах, таких как обработка рукописных документов, автоматическое распознавание текста и другие.

Перспективы дальнейшего улучшения модели

Представленный код позволяет успешно обучить и оценить нейронную сеть для распознавания рукописного текста, но существуют дополнительные шаги и настройки, которые могут быть реализованы для дальнейшего повышения качества модели:

1. **Оптимизация архитектуры модели**:

Увеличение количества фильтров или слоев в CNN для повышения способности модели извлекать более сложные признаки.

Добавление дополнительных рекуррентных слоев (например, GRU вместо LSTM) для улучшения обработки последовательностей.

Эксперименты с другими активационными функциями (например, LeakyReLU или Swish) для улучшения обучения.

2. **Улучшение предобработки данных**:

Использование дополнительных методов аугментации изображений, таких как случайное размытие, добавление шума или искажение перспективы.

Применение методов адаптивной нормализации контраста (CLAHE), чтобы улучшить видимость текста в изображениях.

3. **Использование внешних данных**:

Подключение дополнительных датасетов с рукописным текстом для обучения модели на большем объеме данных.

Применение подходов предварительного обучения (pretraining) на схожих задачах, чтобы использовать уже обученные слои для извлечения признаков.

4. **Настройка гиперпараметров**:

Проведение поиска по сетке (grid search) или байесовского оптимизации для подбора гиперпараметров, таких как скорость обучения, размер скрытого слоя (nh), или параметры аугментации.

Эксперименты с различными оптимизаторами (например, SGD, RMSprop) и scheduler'ами для адаптации скорости обучения.

5. **Регуляризация**:

Использование продвинутых методов регуляризации, таких как DropBlock или CutMix, чтобы уменьшить переобучение.

Увеличение коэффициента регуляризации L2 (weight decay) для уменьшения избыточной сложности модели.

6. **Продвинутые методы декодирования**:

Реализация beam search декодирования вместо greedy decoding, что может значительно улучшить качество предсказаний на длинных последовательностях.

Добавление языковой модели (например, на основе N-грамм или LSTM) для исправления ошибок в распознанных последовательностях.

7. **Анализ ошибок**:

Более глубокий анализ неправильно распознанных слов для выявления систематических ошибок, например, путаницы похожих символов или трудностей с длинными словами.

Улучшение обучения модели на сложных случаях, используя метод hard example mining.

8. **Тонкая настройка CTC Loss**:

Эксперименты с альтернативными функциями потерь, такими как Connectionist Temporal Classification (CTC) с изменением весов или использованием дополнительных функций потерь для балансировки символов.

9. **Улучшение визуализации и интерпретации**:

Применение Grad-CAM или других методов визуализации для понимания того, какие области изображения оказывают наибольшее влияние на предсказания. Реализация одного или нескольких из предложенных методов может существенно улучшить качество распознавания текста и расширить возможности модели для применения в реальных задачах.

Вопросы для самопроверки

Какие основные физические принципы лежат в основе магнитооптической нейронной сети?

В чем заключается основное преимущество использования опто-магнитных синаптических элементов в нейронных сетях?

Опишите роль лазерных импульсов в работе магнитооптической нейронной сети.

Какие преимущества дает использование энергии света для управления магнитными состояниями?

Почему AIMC считается перспективным для использования в нейронных сетях?

Как осуществляется кодирование входных данных в магнитооптической нейронной сети?

Какие материалы используются для создания синаптических элементов в магнитооптической сети?

Какие функции выполняет зондирующий лазерный луч в экспериментальной установке?

Как осуществляется хранение синаптических весов в магнитооптической нейронной сети?

В чем заключается принцип работы функции decode_predictions, используемой в программной реализации?

Какие подходы к предобработке данных рекомендованы для распознавания рукописного текста?

В чем преимущество аугментации данных при обучении модели?

Какую роль играет LSTM в структуре нейронной сети CRNN?

Какие метрики используются для оценки качества модели распознавания текста?

В чем заключается уникальность магнитооптической реализации по сравнению с традиционными нейронными сетями?

С базовой рабочей версией программы можно ознакомиться в репозитории GitHub по ссылке - https://github.com/uselessshark/Handwriting_project.

Список рекомендованной литературы

1. Изучаем Python // М. Лутц. ; [пер. с англ. Ю.Н. Артеменко, ред. С.Н. Тригуб] — 5-е изд. — Москва: Диалектика, 2020. — 824 с. : ил. ISBN 978-5-907144-52-1 (В пер.).
2. Chakravarty A. et al. Training and pattern recognition by an opto-magnetic neural network //Applied Physics Letters. – 2022. – Т. 120. – №. 2.
3. Искусственный интеллект: современный подход // Стюарт Рассел, Питер Норвиг ; [пер. с англ. и ред. К. А. Птицына]. — 2-е изд. — Москва [и др.] : Вильямс, 2015. — 1407 с. : ил.,табл. : 24 см; ISBN 978-5-8459-1968-7 (В пер.).

Безвиконный Никита Владиславович

Гуськов Андрей Александрович

Лавров Сергей Дмитриевич

Программная эмуляция оптомагнитной нейронной сети для анализа рукописного текста

Методическое пособие издано в авторской редакции

Сетевое издание

Ответственный за выпуск – Алимова Н.К.

Учебное издание

Системные требования:

операционная система Windows XP или новее, macOS 10.12 или новее, Linux.

Программное обеспечение для чтения файлов PDF.

Объем данных 1 Мб

Принято к публикации «11» декабря 2024 года

Режим доступа: https://izd-mn.com/PDF/51MNNPU24.pdf свободный. – Загл. с экрана. – Яз. рус., англ.

ООО «Издательство «Мир науки»

«Publishing company «World of science», LLC

Адрес:

Юридический адрес — 127055, г. Москва, пер. Порядковый, д. 21, офис 401.

Почтовый адрес — 127055, г. Москва, пер. Порядковый, д. 21, офис 401.

https://izd-mn.com/