|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА - Российский технологический университет»**  **РТУ МИРЭА** |

Институт Информационных Технологий

Кафедра Вычислительной Техники (ВТ)

**ОТЧЁТ ПО ПРАКТИЧЕСКИМ РАБОТАМ**

по дисциплине

«Проектирование и обучение нейронных сетей»

|  |  |
| --- | --- |
| Выполнил студент группы:  ИКБО-15-22 | Оганнисян Г.А. |
| Принял старший преподаватель кафедры ВТ | Семенов Р.Э. |
| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_2025 г. |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_2025 г. |

Москва 2025 г.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc193367360)

[1 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ ПО МЕТОДУ «ТРАНСФОРМЕР» С НАБОРОМ ДАННЫХ 5](#_Toc193367361)

[1.1 Описание алгоритма 5](#_Toc193367362)

[1.2 Программная реализация обучения нейронной сети по методу «Трансформер» 6](#_Toc193367363)

[2 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ МЕТОДОМ «ГЕНЕРАТИВНАЯ СЕТЬ (DCGAN)» 8](#_Toc193367364)

[2.1 Описание алгоритма 8](#_Toc193367365)

[2.2 Программная реализация обучения нейронной сети методом «Генеративная сеть (DCGAN)» 8](#_Toc193367366)

[3 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ МЕТОДА ГРАФОВЫХ НЕЙРОННЫХ СЕТЕЙ (GCN) 11](#_Toc193367367)

[3.1 Описание алгоритма 11](#_Toc193367368)

[3.2 Программная реализация обучения нейронной сети с использованием алгоритма обратного распространения ошибки 11](#_Toc193367369)

[4 ПРИМЕНИМОСТЬ СВЕРТОЧНЫХ НЕЙРОСЕТЕЙ И ТРАНСФОРМЕРОВ К ЗАДАЧАМ КЛАССИФИКАЦИИ ИЗОБРАЖЕНИЙ 13](#_Toc193367370)

[4.1 Введение и цель работы 13](#_Toc193367371)

[4.2 Методика реализации 13](#_Toc193367372)

[4.2.1 Реализация сверточной нейросети 13](#_Toc193367373)

[4.2.2 Реализация трансформерной модели 14](#_Toc193367374)

[4.3 Эксперементальная часть 15](#_Toc193367375)

[4.3.1 Датасет 15](#_Toc193367376)

[4.3.2 Параметры моделей 15](#_Toc193367377)

[4.3.3 Результаты обучения 16](#_Toc193367378)

[4.4 Сравнительный анализ 19](#_Toc193367379)

[4.5 Примеры результатов предсказания 19](#_Toc193367380)

[4.6 Вывод и заключение 22](#_Toc193367381)

[ЗАКЛЮЧЕНИЕ 24](#_Toc193367382)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 25](#_Toc193367383)

[ПРИЛОЖЕНИЯ 26](#_Toc193367384)

**ВВЕДЕНИЕ**

В последние годы нейронные сети стали основой многих современных решений в области искусственного интеллекта, позволяя эффективно решать задачи обработки изображений, генерации контента, анализа графовых структур и многое другое. Развитие аппаратных ресурсов и появление новых архитектур нейронных сетей привели к появлению специализированных методов обучения, ориентированных на различные типы данных и прикладные задачи. В рамках данного отчёта рассматриваются четыре подхода к обучению нейронных сетей: метод «Трансформер», который использует механизмы внимания для глобального анализа входных данных; метод «Генеративная сеть (DCGAN)», применяемый для генерации новых образцов данных с помощью состязательной игры генератора и дискриминатора; метод «Графовые нейронные сети (GCN)», позволяющий эффективно работать с данными в виде графов; а также сравнительный анализ сверточных нейросетей и трансформеров в задаче классификации изображений.

Изучение различных алгоритмов обучения позволяет выбрать наиболее подходящий метод для конкретной задачи, учитывая особенности данных и требования к модели. Кроме того, понимание принципов работы различных методов обучения способствует разработке более эффективных и устойчивых нейронных сетей, способных справляться с разнообразными вызовами в реальных приложениях.

Таким образом, исследование и разработка методов обучения нейронных сетей продолжают оставаться важным направлением в области искусственного интеллекта, открывая новые возможности для создания интеллектуальных систем, способных решать сложные задачи и адаптироваться к изменяющимся условиям.

1. **ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ ПО МЕТОДУ «ТРАНСФОРМЕР» С НАБОРОМ ДАННЫХ**
   1. **Описание алгоритма**

Алгоритм машинного перевода, основанный на трансформерной архитектуре, эффективно сопоставляет представления исходного и целевого языков для генерации корректного перевода. Основная идея заключается в следующем:

Представление входных данных. Исходное предложение на русском языке и целевое предложение (английский перевод) сначала преобразуются в эмбеддинги с использованием специального слоя, после чего к ним добавляются позиционные кодировки. Это обеспечивает учет порядка токенов в предложении.

Обработка с помощью энкодера. Исходное предложение обрабатывается через несколько слоев энкодера, каждый из которых включает механизм многошагового внимания (MultiHeadAttention), слой нормализации и позиционно-зависимую полносвязную нейронную сеть. Благодаря механизму self-attention модель выделяет важные зависимости в предложении.

Генерация представления для перевода. Целевое предложение формируется с помощью декодера, построенного на базе self-attention. При этом применяется cross-attention, позволяющий учитывать закодированное представление исходного текста для выбора наиболее релевантных токенов на английском языке.

Формирование перевода. После объединения представлений декодера и информации об исходном предложении на каждом шаге генерируется логит, определяющий вероятность появления того или иного токена. На этапе инференса используется жадный алгоритм (greedy decoding), который последовательно выбирает токены с наибольшей вероятностью, что обеспечивает формирование связного перевода..

В итоге алгоритм позволяет с высокой точностью преобразовать текст с одного языка на другой, используя мощь механизмов внимания для учета семантических и синтаксических особенностей исходного текста.

* 1. **Программная реализация обучения нейронной сети по методу «Трансформер»**

Выполним программную реализацию модели для машинного перевода на языке Python (Приложения А – Д). Программная реализация организована модульно с использованием библиотеки PyTorch, что обеспечивает гибкость и удобство отладки. Основные компоненты реализации включают:

1. Базовые блоки трансформера.

* Позиционное кодирование. Реализовано синусоидальное позиционное кодирование, позволяющее модели учитывать относительный порядок токенов.
* MultiHeadAttention – механизм многошагового внимания, где входное представление разбивается на несколько «голов», каждая из которых обрабатывается независимо, а затем результаты объединяются.

1. Энкодер и декодер.

* EncoderLayer и TransformerEncoder – состоят из блоков self-attention, полносвязных сетей, слоев нормализации и dropout. Эти компоненты кодируют исходное предложение, выделяя его внутренние зависимости.
* DecoderLayer и TransformerDecoder – аналогичны энкодеру, но дополнительно используют cross-attention для интеграции информации из исходного предложения при формировании перевода.

1. Интеграция и декодирование.

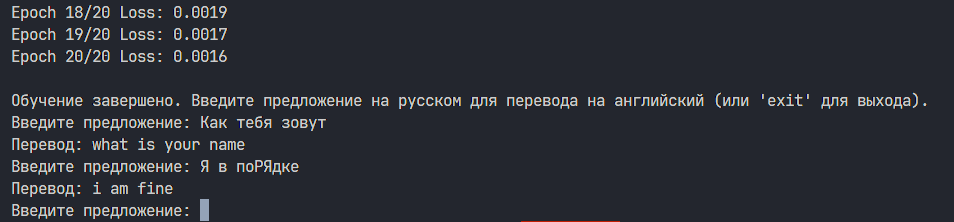
* Класс TransformerModel объединяет работу энкодера и декодера, а также включает финальный линейный слой, преобразующий декодированные представления в распределение вероятностей по словарю целевого языка. При обучении используется механизм teacher forcing, а при тестировании – жадное декодирование.

1. Инференс и обучение.

* Функции подготовки данных, формирования батчей с применением паддинга и маскирования, а также процедуры обучения и тестового перевода реализуют полный цикл работы модели. При обучении вычисляется кросс-энтропия для прогнозируемых токенов, а оптимизация проводится с использованием Adam-оптимизатора.

1. Работа с датасетами.

* В качестве обучающих данных используются пары предложений на русском и английском языках. Каждый пример включает исходное предложение, его перевод и соответствующую токенизацию, что позволяет обеспечить корректное сопоставление и обучение модели. Результат работы модели продемонстрирован на Рисунке 1.1



**Рисунок 1.1 – Пример перевода предложения с русского на английский, выполненного обученной моделью**

1. **ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ МЕТОДОМ «ГЕНЕРАТИВНАЯ СЕТЬ (DCGAN)»**
   1. **Описание алгоритма**

Проект реализует генеративно-состязательную сеть (GAN), состоящую из двух основных компонентов: генератора и дискриминатора.

Генератор принимает на вход случайный латентный вектор (шум) и последовательно проходит через серию линейных и (в данном случае) транспонированных слоёв, с последующим применением BatchNorm и ReLU, генерируя изображение фиксированного размера с пикселями, нормированными в диапазоне [-1, 1].

Дискриминатор – сверточная (или полностью связная, в зависимости от архитектуры) нейронная сеть, которая принимает изображение (реальное или сгенерированное) и, применяя серию слоёв с LeakyReLU и BatchNorm, оценивает вероятность того, что изображение является подлинным.

Во время обучения обе модели взаимодействуют в состязательном режиме: дискриминатор обучается различать реальные и сгенерированные изображения, а генератор – создавать изображения, способные обмануть дискриминатор. Обе сети оптимизируются с использованием функции потерь, основанной на бинарной кросс-энтропии, а также алгоритма Adam.

* 1. **Программная реализация обучения нейронной сети методом «Генеративная сеть (DCGAN)»**

Выполним программную реализацию генеративной сети на языке Python (Приложение Д). Реализация дискриминатора.

Модель определена с использованием нескольких линейных слоёв, каждый из которых преобразует входное изображение, а на выходном слое применяется функция, позволяющая получить логит вероятности подлинности изображения.

Применяемые операции: свёртки, нормализация (BatchNorm), нелинейность (LeakyReLU).

Реализация генератора.

Генератор принимает случайный латентный вектор и с помощью последовательных слоёв, включающих линейные преобразования с ReLU и BatchNorm, поэтапно увеличивает размерность данных до требуемой размерности изображения. На последнем слое применяется Tanh для нормировки значений пикселей.

Основной цикл обучения GAN.

В тренировочном скрипте:

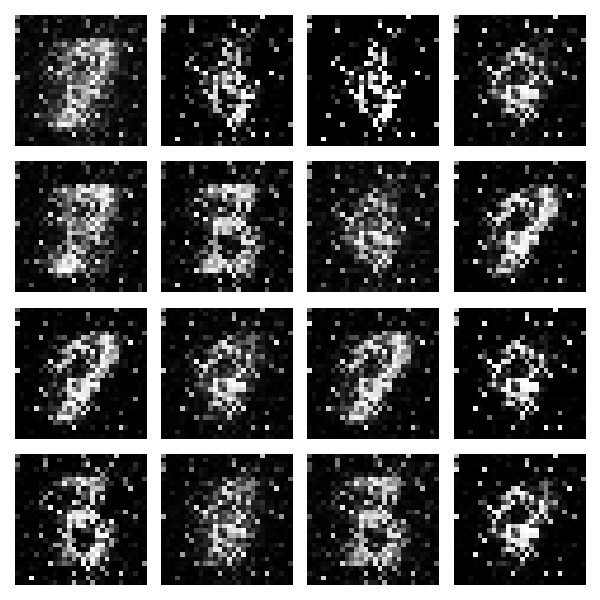
Загружается датасет изображений (например, MNIST или логотипы футбольных команд) с использованием необходимых преобразований (изменение размера, нормализация и т.д.).

Инициализируются модели генератора и дискриминатора, а также их веса с помощью специальной функции инициализации.

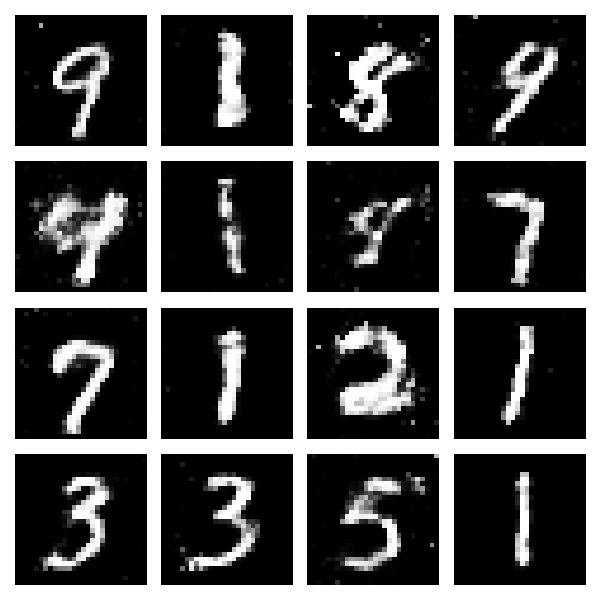
Настраиваются оптимизаторы и функция потерь (BCEWithLogitsLoss).

Обучение проводится в цикле, где для каждого батча сначала обновляется дискриминатор (с отдельными шагами для реальных и сгенерированных изображений), затем – генератор, стремящийся обмануть дискриминатор.

Периодически генерируются и сохраняются примеры изображений, полученные генератором, что позволяет визуально оценить динамику качества обучения. Результат работы модели продемонстрирован на Рисунках 2.1 и 2.2



**Рисунок 2.1 – Примеры сгенерированных изображений на начальных этапах обучения нейронной сети методом «Генеративная сеть»**



**Рисунок 2.2 – Примеры сгенерированных изображений по окончании обучения нейронной сети методом «Генеративная сеть»**

1. **ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ МЕТОДА ГРАФОВЫХ НЕЙРОННЫХ СЕТЕЙ (GCN)** 
   1. **Описание алгоритма**

Проект направлен на анализ ассоциаций между лекарствами с использованием графовых нейронных сетей. Основные этапы алгоритма:

Генерация синтетического графа: На основе реального датасета, например, CORA, создается граф, где узлы представляют лекарства, а ребра – их взаимосвязи (например, совпадение фармакологических профилей или совместное назначение). Каждому лекарству назначается вектор признаков, а также метка, указывающая на наличие атипичного паттерна ассоциаций.

Классификация с помощью GCN: На основе созданного графа обучается графовая сверточная сеть (GCN), состоящая из двух слоев. Первый слой преобразует входные признаки в скрытое представление с применением функции ReLU и dropout для регуляризации, а второй слой генерирует логарифмы вероятностей для каждого класса, что позволяет классифицировать лекарства по паттерну ассоциаций.

Интерпретация результатов: После обучения вычисляется точность классификации, а также производится анализ степени узлов (числа ассоциаций). Статистический анализ (среднее значение и стандартное отклонение) используется для формирования объяснения, позволяющего интерпретировать результаты работы модели.

* 1. **Программная реализация обучения нейронной сети с использованием алгоритма обратного распространения ошибки**

Выполним программную реализацию модели с использованием библиотеки Torch Geometric на языке Python (Приложение Е).

1. Генерация датасета:

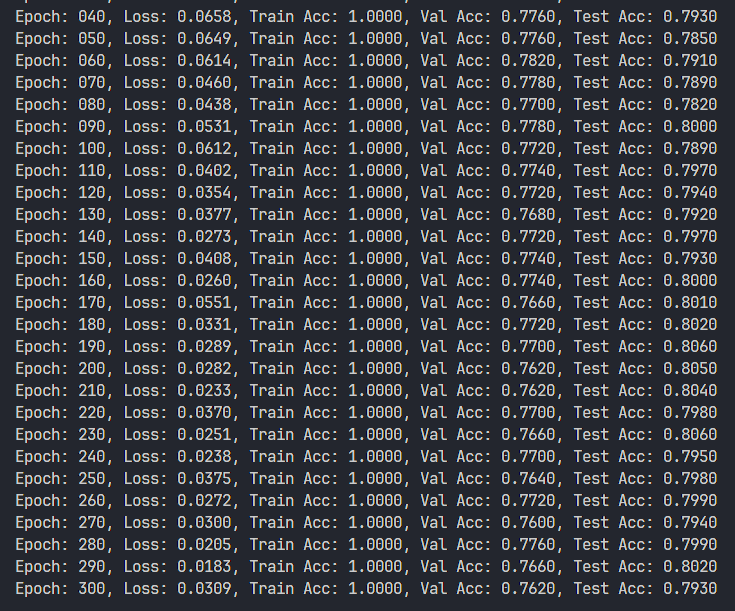
* Датасет CORA используется для представления графа, где узлы содержат признаки и метки классов.

1. Модель GCN:

* Класс GCN реализует графовую сверточную сеть с двумя слоями GCNConv. На первом слое применяется функция ReLU, после чего используется dropout для предотвращения переобучения. Второй слой выводит логарифмы вероятностей по классам, что позволяет проводить классификацию узлов.

1. Обучение и интерпретация:

* Основной скрипт включает функции train и test для обучения модели с использованием оптимизатора Adam и функции потерь Negative Log-Likelihood. Данные делятся на тренировочную, валидационную и тестовую выборки, а итоговые метрики точности выводятся для анализа работы модели. Результат работы программы продемонстрирован на Рисунке 3.1.



**Рисунок 3.1 – Результат работы программы обучения нейронной сети с использованием метода графовых нейронных сетей (GCN)**

1. **СРАВНИТЕЛЬНЫЙ АНАЛИЗ** **ЭФФЕКТИВНОСТИ СВЕРТОЧНЫХ СЕТЕЙ И ТРАНСФОРМЕРОВ В ЗАДАЧАХ КОМПЬЮТЕРНОГО ЗРЕНИЯ**
   1. **Введение и цель работы**

В последние годы задачи компьютерного зрения занимают центральное место в области искусственного интеллекта, находя применение в таких сферах, как автономное вождение, медицинская диагностика, системы видеонаблюдения и др. С развитием глубокого обучения на смену классическим сверточным нейросетям (CNN) приходят трансформерные модели, адаптированные для обработки изображений (Vision Transformers, ViT).

Цель работы – провести сравнительный анализ эффективности сверточных сетей и трансформерных моделей в условиях ограниченного объёма данных и вычислительных ресурсов. В эксперименте использован датасет CIFAR-10, состоящий из 50000 изображений для обучения и 10000 изображений для тестирования.

* 1. **Методика реализации**
     1. **Реализация сверточной нейросети**

Для решения задачи классификации изображений была реализована простая сверточная нейросеть, состоящая из трёх сверточных блоков. Каждый блок включает свёрточный слой с ядром 3×3, функцию активации ReLU и операцию max-pooling, что приводит к уменьшению пространственных размеров изображения. После сверточных слоёв используется полносвязная часть, которая преобразует извлечённые признаки в логиты классов. Обучение модели проводится с использованием оптимизатора Adam и функции кросс-энтропии. Подробный программный код, реализующий данный подход, находится в приложении (Приложение Ж).

* + 1. **Реализация трансформерной модели**

В качестве альтернативного подхода реализована модель Vision Transformer (ViT). В данной модели изображение разбивается на небольшие патчи (в эксперименте используется размер патча 4×4, что обеспечивает получение равномерного набора патчей для изображения 32×32). Каждому патчу сопоставляется embedding посредством линейного слоя, после чего к эмбеддингам добавляются позиционные коды и классификационный токен. Полученная последовательность обрабатывается несколькими слоями TransformerEncoder, а на выходе применяется классификационная голова – полносвязный слой, выдающий логиты классов. Настройка гиперпараметров (embed\_dim=128, num\_layers=4, num\_heads=4 и dropout) позволяет достичь баланса между качеством классификации и вычислительной эффективностью. Полный цикл обучения трансформерной модели приведён в приложении (Приложение З).

* 1. **Эксперементальная часть**
     1. **Датасет**

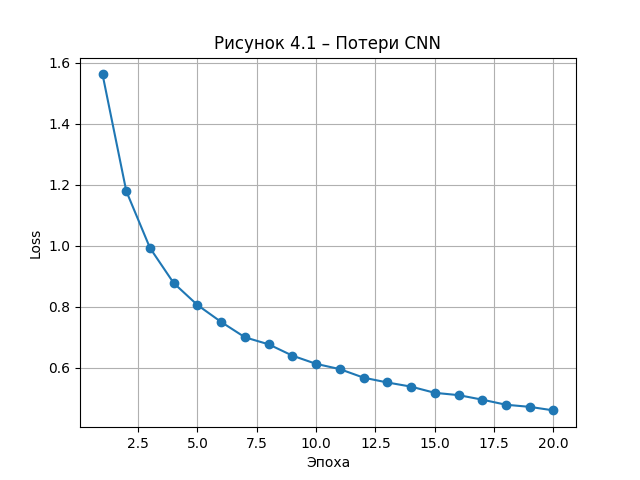
В эксперименте используется датасет CIFAR-10, состоящий из 10 классов изображений. Данные предварительно аугментируются (RandomCrop, RandomHorizontalFlip) и нормализуются, что позволяет улучшить обобщающую способность моделей и обеспечить корректное сравнение.

* + 1. **Параметры моделей**

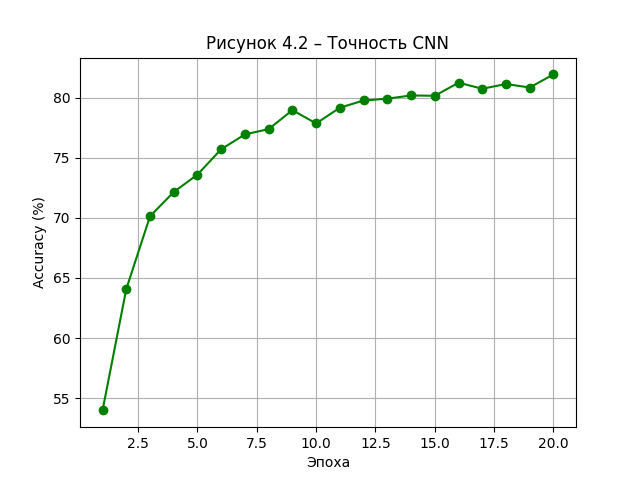
Для сверточной модели использована компактная архитектура, обеспечивающая быструю сходимость и стабильную классификацию. Для трансформерной модели применён подход patch embedding с последующей обработкой последовательности эмбеддингов с помощью стандартного TransformerEncoder. Проводились эксперименты по оптимизации гиперпараметров, и итоговые настройки (например, learning rate=1e-3, batch\_size=128, число эпох=20) были выбраны для обеспечения сопоставимых условий обучения.

* + 1. **Результаты обучения**

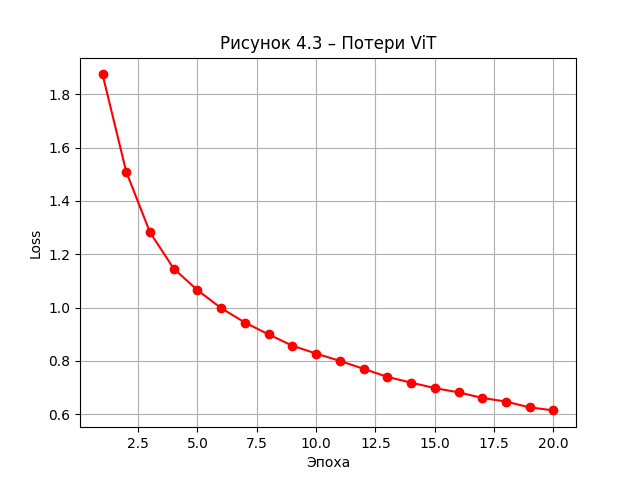
В ходе эксперимента для обеих моделей отслеживались динамика значений функции потерь (loss) и точность (accuracy) на тестовой выборке. Полный код анализа предоставлен в Приложении И. Для визуализации результатов были построены следующие графики:



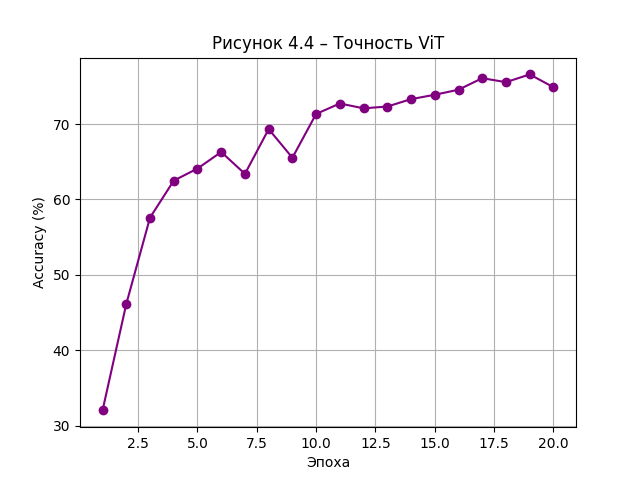
**Рисунок 4.1 – График потерь сверточной нейросети**



**Рисунок 4.2 – График точности сверточной нейросети**



**Рисунок 4.3 – График потерь трансферной модели**



**Рисунок 4.4 – График точности трансферной модели**

Сравнительный анализ показал, что сверточная модель сходится быстрее и демонстрирует высокую точность за счёт эффективного извлечения локальных признаков, в то время как трансформерная модель, способная моделировать глобальные взаимосвязи, требует более длительного обучения и тщательной настройки гиперпараметров.

* 1. **Сравнительный анализ**

Анализ результатов эксперимента выявил следующие ключевые моменты:

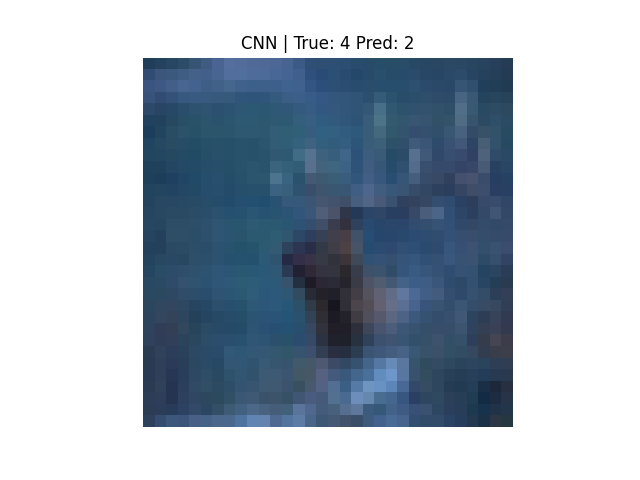
* Извлечение признаков: CNN благодаря локальным фильтрам эффективно различает мелкие детали, что особенно важно для тонкой классификации схожих объектов (например, различение собак и кошек). Трансформер, используя механизм self-attention, способен учитывать глобальные взаимосвязи между патчами, однако его эффективность зависит от оптимальной настройки параметров патча и размерности embedding.
* Скорость обучения и вычислительная сложность: Сверточная нейросеть обучается быстрее и требует меньше вычислительных ресурсов, тогда как трансформерная модель при не оптимальных настройках может существенно замедлять процесс обучения.
* Качество классификации: В эксперименте CNN продемонстрировала высокую точность, тогда как трансформерная модель, несмотря на потенциал для извлечения глобальных зависимостей, иногда сталкивалась с проблемами тонкого различения классов.
  1. **Примеры результатов предсказания**

Для наглядной демонстрации работы моделей были выбраны отдельные тестовые изображения, не входящие в обучающую выборку. Каждый пример проходит через процедуру инференса, при которой изображение предварительно обрабатывается (изменение размера и нормализация), а затем подаётся в модель для получения предсказания.

На рисунках ниже представлены примеры корректных и ошибочных предсказаний для обеих моделей:

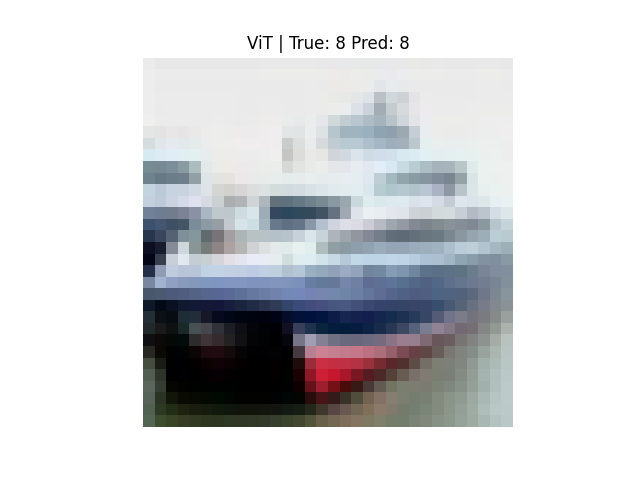


**Рисунок 4.5 – Пример правильного предсказания сверточной нейросети**

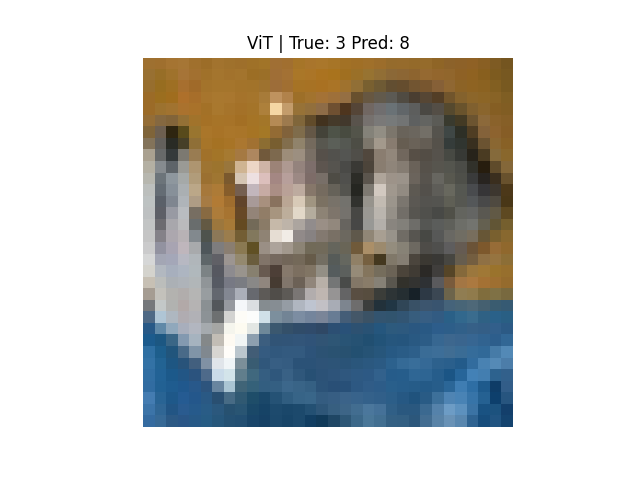


**Рисунок 4.6 – Пример ошибочного предсказания сверточной нейросети**

На рисунке 4.6, изображен олень, а сверточная нейросесть ошибочно предсказала класс указав, что изображена птица.



**Рисунок 4.7 – Пример правильного предсказания трансформерной модели**



**Рисунок 4.8 – Пример ошибочного предсказания трансформерной модели**

На рисунке 4.8, изображена кошка, а модель ошибочно предсказала класс указав, что изображен корабль.

* 1. **Вывод и заключение**

В ходе экспериментов было выявлено, что сверточные нейросети обладают преимуществами в скорости обучения и эффективном извлечении локальных признаков, что делает их надёжным решением для задач классификации изображений при ограниченных вычислительных ресурсах. Трансформерные модели, несмотря на более высокую вычислительную сложность, демонстрируют потенциал для моделирования глобальных взаимосвязей, что может быть полезно при обработке сложных изображений. Однако для достижения конкурентоспособных результатов требуется более тщательная настройка гиперпараметров и, возможно, увеличение объёма обучающих данных.

Таким образом, выбор архитектуры зависит от специфики задачи, доступных ресурсов и требуемой точности. Дальнейшие исследования могут быть направлены на интеграцию преимуществ обоих подходов, а также на оптимизацию трансформерных моделей для задач компьютерного зрения.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения работы были рассмотрены и проанализированы различные подходы к обучению нейронных сетей, применяемых для решения широкого спектра задач в области компьютерного зрения, обработки последовательностей и анализа структурированных данных. Исследование охватывало применение трансформерных моделей для машинного перевода, генеративных сетей (DCGAN) для синтеза изображений, графовых нейронных сетей (GCN) для анализа взаимосвязей, а также сверточных нейросетей (CNN) для классификации изображений.

Полученные результаты демонстрируют, что каждый из рассмотренных методов имеет свои явные преимущества и ограничения. Классические сверточные сети эффективно извлекают локальные признаки и обеспечивают быструю сходимость, что делает их оптимальным выбором для задач с ограниченными вычислительными ресурсами. Трансформерные модели, благодаря механизму self-attention, способны моделировать глобальные взаимосвязи, что особенно полезно при работе с большими объёмами данных и сложными структурами, однако их обучение требует более тщательной настройки гиперпараметров и значительных вычислительных затрат. Генеративные сети демонстрируют потенциал для создания реалистичных изображений и дополнения датасетов, а графовые нейронные сети позволяют эффективно анализировать взаимосвязи между объектами в структурированных данных.

Таким образом, выбор метода обучения нейронных сетей должен основываться на особенностях решаемой задачи, доступных ресурсах и требуемой точности. В дальнейшем перспективным направлением исследований может стать интеграция различных архитектур для создания гибридных моделей, способных объединить сильные стороны каждого подхода и обеспечить более высокую производительность в реальных приложениях.

**СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ**

1. Широков, И.Б. Анализ технологий глубокого обучения с подкреплением для систем машинного зрения / И.Б. Широков, С.В. Колесова, В.А. Кучеренко, М.Ю. Серебряков / Известия ТулГУ 2022.
2. Сущеня, Р.В. Нейронные сети и их классификация. Основные виды нейронных сетей / Р.В. Сущеня, А.Э. Кокаев / Международный научный журнал «Вестник науки» 2023.
3. Паршин, А.И. Случайное мультимодальное глубокое обучение в задаче распознавания изображений / А.И. Паршин, М.Н. Аралов, В.Ф. Барабанов, Н.И. Гребенникова / Вестник Воронежского государственного университета 2021.

**ПРИЛОЖЕНИЯ**

Приложение А – Токенизация и построение словаря.

Приложение Б – Датасет и подготовка батча.

Приложение В – Базовые компоненты трансформера (позиционное кодирование, внимание, FFN).

Приложение Г – Слои энкодера и декодера.

Приложение Д – Полный код DCGAN.

Приложение Е – Полный код GCN.

Приложение Ж – Реализация свёрточной нейросети (CNN).

Приложение З – Реализация трансформерной модели.

Приложение И – Полный код для сравнительного анализа эффективности сверточных сетей и трансформеров в задачах компьютерного зрения.

**Приложение А**

Токенизация и построение словаря.

*Листинг А – Токенизация и построение словаря*

|  |
| --- |
| import torch  # Специальные токены и их индексы  PAD\_TOKEN = "<pad>"  SOS\_TOKEN = "<sos>"  EOS\_TOKEN = "<eos>"  UNK\_TOKEN = "<unk>"  PAD\_IDX = 0  SOS\_IDX = 1  EOS\_IDX = 2  UNK\_IDX = 3  # Функция для построения словаря из предложений  def build\_vocab(sentences, specials=[PAD\_TOKEN, SOS\_TOKEN, EOS\_TOKEN, UNK\_TOKEN]):  vocab = {token: idx for idx, token in enumerate(specials)}  for sentence in sentences:  for token in sentence.strip().split():  if token not in vocab:  vocab[token] = len(vocab)  return vocab  # Инвертирование словаря (индекс -> токен)  def invert\_vocab(vocab):  return {idx: token for token, idx in vocab.items()}  # Токенизация с добавлением специальных токенов  def tokenize(sentence, vocab):  tokens = sentence.strip().split()  tokens = [SOS\_TOKEN] + tokens + [EOS\_TOKEN]  return [vocab.get(token, UNK\_IDX) for token in tokens] |

**Приложение Б**

Датасет и подготовка батча.

*Листинг Б – Датасет и подготовка батча*

|  |
| --- |
| import torch  from torch.utils.data import Dataset, DataLoader  import torch.nn as nn  # Датасет для машинного перевода: русские предложения -> английские переводы  class TranslationDataset(Dataset):  def \_\_init\_\_(self, src\_sentences, tgt\_sentences, src\_vocab, tgt\_vocab):  self.src\_sentences = src\_sentences  self.tgt\_sentences = tgt\_sentences  self.src\_vocab = src\_vocab  self.tgt\_vocab = tgt\_vocab  def \_\_len\_\_(self):  return len(self.src\_sentences)  def \_\_getitem\_\_(self, idx):  src\_indices = tokenize(self.src\_sentences[idx], self.src\_vocab)  tgt\_indices = tokenize(self.tgt\_sentences[idx], self.tgt\_vocab)  return torch.tensor(src\_indices, dtype=torch.long), torch.tensor(tgt\_indices, dtype=torch.long)  # Функция формирования батча с паддингом (batch\_first=True)  def collate\_fn(batch):  src\_batch, tgt\_batch = zip(\*batch)  src\_batch = nn.utils.rnn.pad\_sequence(src\_batch, batch\_first=True, padding\_value=PAD\_IDX)  tgt\_batch = nn.utils.rnn.pad\_sequence(tgt\_batch, batch\_first=True, padding\_value=PAD\_IDX)  return src\_batch, tgt\_batch |

**Приложение В**

Базовые компоненты трансформера.

*Листинг В – Базовые компоненты трансформера*

|  |
| --- |
| import math  import torch  import torch.nn as nn  # Позиционное кодирование (синусоидальное)  class PositionalEncoding(nn.Module):  def \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):  super(PositionalEncoding, self).\_\_init\_\_()  self.dropout = nn.Dropout(p=dropout)  pe = torch.zeros(max\_len, d\_model) # (max\_len, d\_model)  position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1) # (max\_len, 1)  div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* (-math.log(10000.0) / d\_model))  pe[:, 0::2] = torch.sin(position \* div\_term) # четные индексы  pe[:, 1::2] = torch.cos(position \* div\_term) # нечетные  pe = pe.unsqueeze(0) # (1, max\_len, d\_model)  self.register\_buffer('pe', pe)  def forward(self, x):  # x: (batch\_size, seq\_len, d\_model)  x = x + self.pe[:, :x.size(1), :]  return self.dropout(x)  # Масштабированное скалярное произведение (scaled dot-product attention)  def scaled\_dot\_product\_attention(Q, K, V, mask=None):  d\_k = Q.size(-1)  scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d\_k)  if mask is not None:  scores = scores.masked\_fill(mask == 0, -1e9)  attn = torch.softmax(scores, dim=-1)  output = torch.matmul(attn, V)  return output, attn  # Многопоточное внимание (Multi-Head Attention)  class MultiHeadAttention(nn.Module):  def \_\_init\_\_(self, d\_model, num\_heads, dropout=0.1):  super(MultiHeadAttention, self).\_\_init\_\_()  assert d\_model % num\_heads == 0, "d\_model должно делиться на число голов"  self.num\_heads = num\_heads  self.d\_k = d\_model // num\_heads  self.d\_model = d\_model  self.linear\_q = nn.Linear(d\_model, d\_model)  self.linear\_k = nn.Linear(d\_model, d\_model)  self.linear\_v = nn.Linear(d\_model, d\_model)  self.linear\_out = nn.Linear(d\_model, d\_model)  self.dropout = nn.Dropout(dropout)  def forward(self, query, key, value, mask=None):  # query, key, value: (batch\_size, seq\_len, d\_model)  batch\_size = query.size(0)  Q = self.linear\_q(query)  K = self.linear\_k(key)  V = self.linear\_v(value) |

*Продолжение листинга В*

|  |
| --- |
| # Разбиваем на головы: (batch\_size, num\_heads, seq\_len, d\_k)  Q = Q.view(batch\_size, -1, self.num\_heads, self.d\_k).transpose(1, 2)  K = K.view(batch\_size, -1, self.num\_heads, self.d\_k).transpose(1, 2)  V = V.view(batch\_size, -1, self.num\_heads, self.d\_k).transpose(1, 2)  if mask is not None:  if mask.dim() == 2:  mask = mask.unsqueeze(0)  mask = mask.unsqueeze(1)  attn\_output, \_ = scaled\_dot\_product\_attention(Q, K, V, mask)  # Конкатенация голов: (batch\_size, seq\_len, d\_model)  attn\_output = attn\_output.transpose(1, 2).contiguous().view(batch\_size, -1, self.d\_model)  output = self.linear\_out(attn\_output)  return output  # Полносвязная сеть (Feed Forward)  class PositionwiseFeedForward(nn.Module):  def \_\_init\_\_(self, d\_model, d\_ff, dropout=0.1):  super(PositionwiseFeedForward, self).\_\_init\_\_()  self.linear1 = nn.Linear(d\_model, d\_ff)  self.relu = nn.ReLU()  self.dropout = nn.Dropout(dropout)  self.linear2 = nn.Linear(d\_ff, d\_model)  def forward(self, x):  return self.linear2(self.dropout(self.relu(self.linear1(x)))) |

**Приложение Г**

Слои энкодера и декодера.

*Листинг Г – Слои энкодера и декодерай*

|  |
| --- |
| import torch.nn as nn  # Слой энкодера Transformer  class TransformerEncoderLayer(nn.Module):  def \_\_init\_\_(self, d\_model, num\_heads, d\_ff, dropout=0.1):  super(TransformerEncoderLayer, self).\_\_init\_\_()  self.self\_attn = MultiHeadAttention(d\_model, num\_heads, dropout)  self.ff = PositionwiseFeedForward(d\_model, d\_ff, dropout)  self.norm1 = nn.LayerNorm(d\_model)  self.norm2 = nn.LayerNorm(d\_model)  self.dropout = nn.Dropout(dropout)  def forward(self, x, mask=None):  attn\_output = self.self\_attn(x, x, x, mask)  x = self.norm1(x + self.dropout(attn\_output))  ff\_output = self.ff(x)  x = self.norm2(x + self.dropout(ff\_output))  return x  # Слой декодера Transformer  class TransformerDecoderLayer(nn.Module):  def \_\_init\_\_(self, d\_model, num\_heads, d\_ff, dropout=0.1):  super(TransformerDecoderLayer, self).\_\_init\_\_()  self.self\_attn = MultiHeadAttention(d\_model, num\_heads, dropout)  self.enc\_dec\_attn = MultiHeadAttention(d\_model, num\_heads, dropout)  self.ff = PositionwiseFeedForward(d\_model, d\_ff, dropout)  self.norm1 = nn.LayerNorm(d\_model)  self.norm2 = nn.LayerNorm(d\_model)  self.norm3 = nn.LayerNorm(d\_model)  self.dropout = nn.Dropout(dropout)  def forward(self, x, encoder\_output, tgt\_mask=None, memory\_mask=None):  self\_attn\_output = self.self\_attn(x, x, x, tgt\_mask)  x = self.norm1(x + self.dropout(self\_attn\_output))  enc\_dec\_attn\_output = self.enc\_dec\_attn(x, encoder\_output, encoder\_output, memory\_mask)  x = self.norm2(x + self.dropout(enc\_dec\_attn\_output))  ff\_output = self.ff(x)  x = self.norm3(x + self.dropout(ff\_output))  return x  # Стек слоёв энкодера  class TransformerEncoder(nn.Module):  def \_\_init\_\_(self, num\_layers, d\_model, num\_heads, d\_ff, dropout=0.1):  super(TransformerEncoder, self).\_\_init\_\_()  self.layers = nn.ModuleList([  TransformerEncoderLayer(d\_model, num\_heads, d\_ff, dropout)  for \_ in range(num\_layers)  ])  def forward(self, x, mask=None):  for layer in self.layers:  x = layer(x, mask)  return x  # Стек слоёв декодера |

*Продолжение листинга Г*

|  |
| --- |
| class TransformerDecoder(nn.Module):  def \_\_init\_\_(self, num\_layers, d\_model, num\_heads, d\_ff, dropout=0.1):  super(TransformerDecoder, self).\_\_init\_\_()  self.layers = nn.ModuleList([  TransformerDecoderLayer(d\_model, num\_heads, d\_ff, dropout)  for \_ in range(num\_layers)  ])  def forward(self, x, encoder\_output, tgt\_mask=None, memory\_mask=None):  for layer in self.layers:  x = layer(x, encoder\_output, tgt\_mask, memory\_mask)  return x |

**Приложение Д**

Полный код DCGAN.

*Листинг Д – Полный код DCGAN*

|  |
| --- |
| import os  import torch  import torch.nn as nn  import torch.optim as optim  import torchvision.datasets as datasets  import torchvision.transforms as transforms  import matplotlib  matplotlib.use('Agg') # используем неинтерактивный backend  import matplotlib.pyplot as plt  import time  if torch.cuda.is\_available():  torch.backends.cudnn.benchmark = True  # Гиперпараметры  batch\_size = 128  lr = 0.0002  epochs = 1000  noise\_dim = 100 # размер случайного шума  save\_every = 5 # сохранять изображения каждые N эпох  # Путь для сохранения изображений  save\_dir = "./semestr-6/Neyronki/GAN/generated\_images"  os.makedirs(save\_dir, exist\_ok=True)  # Трансформации и загрузка MNIST  transform = transforms.Compose([  transforms.Resize(28),  transforms.ToTensor(),  transforms.Normalize((0.5,), (0.5,))  ])  dataset = datasets.MNIST(root='./semestr-6/Neyronki/GAN/data', train=True, download=True, transform=transform)  dataloader = torch.utils.data.DataLoader(  dataset, batch\_size=batch\_size, shuffle=True,  num\_workers=4, pin\_memory=torch.cuda.is\_available()  )  # Генератор  class Generator(nn.Module):  def \_\_init\_\_(self, noise\_dim):  super(Generator, self).\_\_init\_\_()  self.model = nn.Sequential(  nn.Linear(noise\_dim, 128),  nn.ReLU(inplace=True),  nn.Linear(128, 256),  nn.ReLU(inplace=True),  nn.Linear(256, 28 \* 28),  nn.Tanh()  )  def forward(self, z):  img = self.model(z)  return img.view(-1, 1, 28, 28) |

*Продолжение листинга Д*

|  |
| --- |
| # Дискриминатор (без Sigmoid, т.к. используем BCEWithLogitsLoss)  class Discriminator(nn.Module):  def \_\_init\_\_(self):  super(Discriminator, self).\_\_init\_\_()  self.model = nn.Sequential(  nn.Linear(28 \* 28, 256),  nn.LeakyReLU(0.2, inplace=True),  nn.Linear(256, 128),  nn.LeakyReLU(0.2, inplace=True),  nn.Linear(128, 1)  )  def forward(self, img):  x = img.view(-1, 28 \* 28)  return self.model(x)  # Функция для сохранения сгенерированных изображений в указанную папку  def show\_generated\_images(epoch, generator, device, fixed\_noise):  generator.eval()  with torch.no\_grad():  gen\_imgs = generator(fixed\_noise.to(device)).cpu().numpy()  generator.train()  fig, axes = plt.subplots(4, 4, figsize=(6, 6))  for i, ax in enumerate(axes.flat):  img = gen\_imgs[i].squeeze()  img = (img + 1) / 2.0 # денормализация из [-1, 1] в [0, 1]  ax.imshow(img, cmap='gray')  ax.axis("off")  plt.tight\_layout()    save\_path = os.path.join(save\_dir, f"generated\_epoch\_{epoch:03d}.png")  plt.savefig(save\_path)  plt.close(fig)  def train():  device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  generator = Generator(noise\_dim).to(device)  discriminator = Discriminator().to(device)  optimizer\_G = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))  optimizer\_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))    criterion = nn.BCEWithLogitsLoss()  # Если используется GPU, применяем AMP для ускорения (новый синтаксис)  scaler\_G = torch.amp.GradScaler(device\_type='cuda') if device.type == 'cuda' else None  scaler\_D = torch.amp.GradScaler(device\_type='cuda') if device.type == 'cuda' else None  fixed\_noise = torch.randn(16, noise\_dim)  for epoch in range(1, epochs + 1):  for batch\_idx, (imgs, \_) in enumerate(dataloader):  imgs = imgs.to(device)  current\_batch = imgs.size(0)  valid = torch.ones(current\_batch, 1, device=device)  fake = torch.zeros(current\_batch, 1, device=device)  # Обучение генератора |

*Окончание листинга Д*

|  |
| --- |
| optimizer\_G.zero\_grad()  z = torch.randn(current\_batch, noise\_dim, device=device)  if scaler\_G:  with torch.amp.autocast(device\_type='cuda'):  gen\_imgs = generator(z)  g\_loss = criterion(discriminator(gen\_imgs), valid)  scaler\_G.scale(g\_loss).backward()  scaler\_G.step(optimizer\_G)  scaler\_G.update()  else:  gen\_imgs = generator(z)  g\_loss = criterion(discriminator(gen\_imgs), valid)  g\_loss.backward()  optimizer\_G.step()  # Обучение дискриминатора  optimizer\_D.zero\_grad()  if scaler\_D:  with torch.amp.autocast(device\_type='cuda'):  real\_loss = criterion(discriminator(imgs), valid)  fake\_loss = criterion(discriminator(gen\_imgs.detach()), fake)  d\_loss = (real\_loss + fake\_loss) / 2  scaler\_D.scale(d\_loss).backward()  scaler\_D.step(optimizer\_D)  scaler\_D.update()  else:  real\_loss = criterion(discriminator(imgs), valid)  fake\_loss = criterion(discriminator(gen\_imgs.detach()), fake)  d\_loss = (real\_loss + fake\_loss) / 2  d\_loss.backward()  optimizer\_D.step()  if batch\_idx % 100 == 0:  print(f"Epoch [{epoch}/{epochs}] Batch [{batch\_idx}/{len(dataloader)}] "  f"D\_loss: {d\_loss.item():.4f} G\_loss: {g\_loss.item():.4f}")  # Сохраняем изображения только каждые save\_every эпох  if epoch % save\_every == 0:  show\_generated\_images(epoch, generator, device, fixed\_noise)  if \_\_name\_\_ == '\_\_main\_\_':  train() |

**Приложение Е**

Полный код GCN

*Листинг Е – Полный код GCN*

|  |
| --- |
| import torch  import torch.nn.functional as F  from torch\_geometric.datasets import Planetoid  from torch\_geometric.nn import GCNConv  # Загружаем датасет CORA  dataset = Planetoid(root='C:/Users/Grigo/Documents/GitGrisha/Practic\_MIREA/semestr-6/Neyronki/GNN/Cora', name='Cora')  data = dataset[0] # В датасете CORA весь граф хранится в одном объекте  # Определяем модель GCN  class GCN(torch.nn.Module):  def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  super(GCN, self).\_\_init\_\_()  # Первый графовый сверточный слой: преобразует входные признаки в скрытое представление  self.conv1 = GCNConv(input\_dim, hidden\_dim)  # Второй графовый сверточный слой: выводит вероятности классов  self.conv2 = GCNConv(hidden\_dim, output\_dim)  def forward(self, data):  x, edge\_index = data.x, data.edge\_index  # Применяем первый слой и ReLU-активацию  x = self.conv1(x, edge\_index)  x = F.relu(x)  # Применяем dropout для регуляризации  x = F.dropout(x, training=self.training)  # Применяем второй слой  x = self.conv2(x, edge\_index)  # Вычисляем логарифм вероятностей по классам  return F.log\_softmax(x, dim=1)  # Инициализируем модель, оптимизатор и задаём параметры обучения  model = GCN(dataset.num\_node\_features, hidden\_dim=16, output\_dim=dataset.num\_classes)  optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight\_decay=5e-4)  # Функция обучения  def train():  model.train()  optimizer.zero\_grad()  out = model(data)  # Вычисляем потерю (loss) по узлам, принадлежащим тренировочной выборке  loss = F.nll\_loss(out[data.train\_mask], data.y[data.train\_mask])  loss.backward()  optimizer.step()  return loss.item()  # Функция тестирования: вычисляем точность на тренировочной, валидационной и тестовой выборках  def test():  model.eval()  logits = model(data)  accs = []  for mask in [data.train\_mask, data.val\_mask, data.test\_mask]: |

*Продолжение листинга Е*

|  |
| --- |
| pred = logits[mask].max(1)[1] # предсказание: индекс максимального значения  acc = pred.eq(data.y[mask]).sum().item() / mask.sum().item()  accs.append(acc)  return accs  # Обучаем модель в течение 300 эпох  for epoch in range(1, 301):  loss = train()  train\_acc, val\_acc, test\_acc = test()  if epoch % 10 == 0:  print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, '  f'Train Acc: {train\_acc:.4f}, Val Acc: {val\_acc:.4f}, Test Acc: {test\_acc:.4f}') |

**Приложение Ж**

Реализация свёрточной нейросети (CNN).

*Листинг Ж – Реализация свёрточной нейросети (CNN)*

|  |
| --- |
| class CNNClassifier(nn.Module):  def \_\_init\_\_(self, num\_classes=10):  super(CNNClassifier, self).\_\_init\_\_()  self.features = nn.Sequential(  nn.Conv2d(3, 32, kernel\_size=3, padding=1),  nn.ReLU(),  nn.MaxPool2d(2),  nn.Conv2d(32, 64, kernel\_size=3, padding=1),  nn.ReLU(),  nn.MaxPool2d(2),  nn.Conv2d(64, 128, kernel\_size=3, padding=1),  nn.ReLU(),  nn.MaxPool2d(2)  )  self.classifier = nn.Sequential(  nn.Flatten(),  nn.Linear(128 \* 4 \* 4, 256),  nn.ReLU(),  nn.Linear(256, num\_classes)  )  def forward(self, x):  x = self.features(x)  x = self.classifier(x)  return x |

**Приложение З**

Реализация трансформерной модели.

*Листинг З – Реализация трансформерной модели*

|  |
| --- |
| class VisionTransformer(nn.Module):  def \_\_init\_\_(self, image\_size=32, patch\_size=4, in\_channels=3, num\_classes=10,  embed\_dim=128, num\_layers=4, num\_heads=4, dropout=0.1):  super(VisionTransformer, self).\_\_init\_\_()  assert image\_size % patch\_size == 0, "Размер изображения должен делиться на размер патча"  self.num\_patches = (image\_size // patch\_size) \*\* 2  self.patch\_dim = in\_channels \* patch\_size \* patch\_size  self.patch\_embed = nn.Linear(self.patch\_dim, embed\_dim)  self.cls\_token = nn.Parameter(torch.zeros(1, 1, embed\_dim))  self.pos\_embed = nn.Parameter(torch.zeros(1, self.num\_patches + 1, embed\_dim))  self.dropout = nn.Dropout(dropout)  encoder\_layer = nn.TransformerEncoderLayer(d\_model=embed\_dim, nhead=num\_heads, dropout=dropout)  self.transformer\_encoder = nn.TransformerEncoder(encoder\_layer, num\_layers=num\_layers)  self.mlp\_head = nn.Sequential(  nn.LayerNorm(embed\_dim),  nn.Linear(embed\_dim, num\_classes)  )    def forward(self, x):  B, C, H, W = x.shape  patch\_size = int(np.sqrt(self.patch\_dim / C))  x = x.unfold(2, patch\_size, patch\_size).unfold(3, patch\_size, patch\_size)  x = x.contiguous().view(B, C, -1, patch\_size, patch\_size)  x = x.permute(0, 2, 1, 3, 4).contiguous().view(B, self.num\_patches, -1)  x = self.patch\_embed(x)  cls\_tokens = self.cls\_token.expand(B, -1, -1)  x = torch.cat((cls\_tokens, x), dim=1)  x = x + self.pos\_embed  x = self.dropout(x)  x = x.transpose(0, 1)  x = self.transformer\_encoder(x)  x = x.transpose(0, 1)  x = x[:, 0]  x = self.mlp\_head(x)  return x |

**Приложение И**

Полный код для сравнительного анализа эффективности сверточных сетей и трансформеров в задачах компьютерного зрения.

*Листинг И – Полный код для сравнительного анализа эффективности сверточных сетей и трансформеров в задачах компьютерного зрения*

|  |
| --- |
| import os  import torch  import torch.nn as nn  import torch.optim as optim  import torchvision  import torchvision.transforms as transforms  import matplotlib.pyplot as plt  import numpy as np  # Создаём папку для логов, если её нет  logs\_dir = "C:/Users/Grigo/Documents/Work/Practic\_MIREA/semestr-6/Neyronki/cursach/logs"  os.makedirs(logs\_dir, exist\_ok=True)  # Проверка наличия GPU  device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  # Параметры обучения  num\_epochs = 20  batch\_size = 128  learning\_rate = 1e-3  # Подготовка датасета CIFAR-10 с аугментацией  transform\_train = transforms.Compose([  transforms.RandomCrop(32, padding=4),  transforms.RandomHorizontalFlip(),  transforms.ToTensor(),  transforms.Normalize((0.4914, 0.4822, 0.4465),  (0.2023, 0.1994, 0.2010)),  ])  transform\_test = transforms.Compose([  transforms.ToTensor(),  transforms.Normalize((0.4914, 0.4822, 0.4465),  (0.2023, 0.1994, 0.2010)),  ])  train\_dataset = torchvision.datasets.CIFAR10(  root='C:/Users/Grigo/Documents/Work/Practic\_MIREA/semestr-6/Neyronki/cursach',  train=True, download=True, transform=transform\_train)  test\_dataset = torchvision.datasets.CIFAR10(  root='C:/Users/Grigo/Documents/Work/Practic\_MIREA/semestr-6/Neyronki/cursach',  train=False, download=True, transform=transform\_test)  train\_loader = torch.utils.data.DataLoader(  train\_dataset, batch\_size=batch\_size, shuffle=True, num\_workers=2)  test\_loader = torch.utils.data.DataLoader(  test\_dataset, batch\_size=batch\_size, shuffle=False, num\_workers=2)  # 4.2.1. Реализация сверточной нейросети (CNN)  class CNNClassifier(nn.Module): |

*Продолжение листинга И*

|  |
| --- |
| def \_\_init\_\_(self, num\_classes=10):  super(CNNClassifier, self).\_\_init\_\_()  self.features = nn.Sequential(  nn.Conv2d(3, 32, kernel\_size=3, padding=1), # [B,32,32,32]  nn.ReLU(),  nn.MaxPool2d(2), # [B,32,16,16]  nn.Conv2d(32, 64, kernel\_size=3, padding=1), # [B,64,16,16]  nn.ReLU(),  nn.MaxPool2d(2), # [B,64,8,8]  nn.Conv2d(64, 128, kernel\_size=3, padding=1), # [B,128,8,8]  nn.ReLU(),  nn.MaxPool2d(2) # [B,128,4,4]  )  self.classifier = nn.Sequential(  nn.Flatten(),  nn.Linear(128 \* 4 \* 4, 256),  nn.ReLU(),  nn.Linear(256, num\_classes)  )  def forward(self, x):  x = self.features(x)  x = self.classifier(x)  return x  # 4.2.2. Реализация трансформерной модели (Vision Transformer)  class VisionTransformer(nn.Module):  def \_\_init\_\_(self, image\_size=32, patch\_size=4, in\_channels=3, num\_classes=10,  embed\_dim=128, num\_layers=4, num\_heads=4, dropout=0.1):  super(VisionTransformer, self).\_\_init\_\_()  assert image\_size % patch\_size == 0, "Размер изображения должен делиться на размер патча"  self.num\_patches = (image\_size // patch\_size) \*\* 2  self.patch\_dim = in\_channels \* patch\_size \* patch\_size  # Patch embedding  self.patch\_embed = nn.Linear(self.patch\_dim, embed\_dim)  # Классификационный токен  self.cls\_token = nn.Parameter(torch.zeros(1, 1, embed\_dim))  # Позиционные эмбеддинги  self.pos\_embed = nn.Parameter(torch.zeros(1, self.num\_patches + 1, embed\_dim))  self.dropout = nn.Dropout(dropout)  # Transformer Encoder  encoder\_layer = nn.TransformerEncoderLayer(d\_model=embed\_dim, nhead=num\_heads, dropout=dropout)  self.transformer\_encoder = nn.TransformerEncoder(encoder\_layer, num\_layers=num\_layers)  # Классификационная голова  self.mlp\_head = nn.Sequential(  nn.LayerNorm(embed\_dim),  nn.Linear(embed\_dim, num\_classes)  )    def forward(self, x):  B, C, H, W = x.shape  patch\_size = int(np.sqrt(self.patch\_dim / C))  # Разбивка изображения на патчи  x = x.unfold(2, patch\_size, patch\_size).unfold(3, patch\_size, patch\_size) |

*Продолжение листинга И*

|  |  |
| --- | --- |
| x = x.contiguous().view(B, C, -1, patch\_size, patch\_size) # [B, C, num\_patches, patch\_size, patch\_size]  x = x.permute(0, 2, 1, 3, 4).contiguous().view(B, self.num\_patches, -1) # [B, num\_patches, patch\_dim]  # Применяем линейное преобразование к каждому патчу  x = self.patch\_embed(x) # [B, num\_patches, embed\_dim]  # Добавляем классификационный токен  cls\_tokens = self.cls\_token.expand(B, -1, -1) # [B, 1, embed\_dim]  x = torch.cat((cls\_tokens, x), dim=1) # [B, num\_patches+1, embed\_dim]  # Добавляем позиционные эмбеддинги и применяем dropout  x = x + self.pos\_embed  x = self.dropout(x)  # Трансформер ожидает вход с размерностью [seq\_len, B, embed\_dim]  x = x.transpose(0, 1)  x = self.transformer\_encoder(x)  x = x.transpose(0, 1) # [B, num\_patches+1, embed\_dim]  # Используем эмбеддинг классификационного токена для классификации  x = x[:, 0]  x = self.mlp\_head(x)  return x  # Функция обучения модели с сбором статистики (loss и accuracy за эпоху)  def train\_model(model, optimizer, criterion, train\_loader, test\_loader, num\_epochs):  model.train()  losses = []  acc\_list = []  for epoch in range(num\_epochs):  running\_loss = 0.0  for images, labels in train\_loader:  images, labels = images.to(device), labels.to(device)  optimizer.zero\_grad()  outputs = model(images)  loss = criterion(outputs, labels)  loss.backward()  optimizer.step()  running\_loss += loss.item() \* images.size(0)  epoch\_loss = running\_loss / len(train\_loader.dataset)  losses.append(epoch\_loss)  # Вычисляем точность на тестовой выборке после каждой эпохи  acc = evaluate\_model(model, test\_loader)  acc\_list.append(acc)  print(f"Epoch [{epoch+1}/{num\_epochs}] Loss: {epoch\_loss:.4f} Test Acc: {acc:.2f}%")  return losses, acc\_list  # Функция тестирования модели  def evaluate\_model(model, test\_loader):  model.eval()  correct = 0  total = 0  with torch.no\_grad():  for images, labels in test\_loader:  images, labels = images.to(device), labels.to(device)  outputs = model(images)  \_, predicted = torch.max(outputs.data, 1)  total += labels.size(0)  correct += (predicted == labels).sum().item() |  |

*Продолжение листинга И*

|  |
| --- |
| acc = 100 \* correct / total  return acc  # Функция для получения примеров предсказаний (правильных и ошибочных)  def get\_prediction\_examples(model, test\_loader, num\_examples=1):  model.eval()  correct\_examples = []  incorrect\_examples = []  with torch.no\_grad():  for images, labels in test\_loader:  images, labels = images.to(device), labels.to(device)  outputs = model(images)  \_, predicted = torch.max(outputs.data, 1)  for i in range(images.size(0)):  if predicted[i] == labels[i] and len(correct\_examples) < num\_examples:  correct\_examples.append((images[i].cpu(), labels[i].cpu(), predicted[i].cpu()))  elif predicted[i] != labels[i] and len(incorrect\_examples) < num\_examples:  incorrect\_examples.append((images[i].cpu(), labels[i].cpu(), predicted[i].cpu()))  if len(correct\_examples) >= num\_examples and len(incorrect\_examples) >= num\_examples:  break  if len(correct\_examples) >= num\_examples and len(incorrect\_examples) >= num\_examples:  break  return correct\_examples, incorrect\_examples  # Функция для денормализации изображений CIFAR-10  def unnormalize(img, mean=(0.4914, 0.4822, 0.4465), std=(0.2023, 0.1994, 0.2010)):  for t, m, s in zip(img, mean, std):  t.mul\_(s).add\_(m)  return img  # Функция для сохранения примера предсказания с подписью  def save\_prediction\_example(example, filename, model\_name):  # example: (image, true\_label, predicted\_label)  img, true\_label, pred\_label = example  img = unnormalize(img)  npimg = img.numpy().transpose(1, 2, 0)  plt.figure()  plt.imshow(np.clip(npimg, 0, 1))  plt.title(f"{model\_name} | True: {true\_label.item()} Pred: {pred\_label.item()}")  plt.axis("off")  plt.savefig(os.path.join(logs\_dir, filename))  plt.close()  if \_\_name\_\_ == '\_\_main\_\_':  # Инициализация моделей, оптимизаторов и функции потерь  cnn\_model = CNNClassifier(num\_classes=10).to(device)  vit\_model = VisionTransformer(image\_size=32, patch\_size=4, in\_channels=3, num\_classes=10,  embed\_dim=128, num\_layers=4, num\_heads=4, dropout=0.1).to(device)    criterion = nn.CrossEntropyLoss()  optimizer\_cnn = optim.Adam(cnn\_model.parameters(), lr=learning\_rate)  optimizer\_vit = optim.Adam(vit\_model.parameters(), lr=learning\_rate)  print("Обучение сверточной нейросети...")  cnn\_losses, cnn\_acc\_list = train\_model(cnn\_model, optimizer\_cnn, criterion, train\_loader, test\_loader, num\_epochs)  final\_cnn\_acc = evaluate\_model(cnn\_model, test\_loader)  print(f"Точность CNN: {final\_cnn\_acc:.2f}%")  print("\nОбучение Vision Transformer...")  vit\_losses, vit\_acc\_list = train\_model(vit\_model, optimizer\_vit, criterion, train\_loader, test\_loader, num\_epochs)  final\_vit\_acc = evaluate\_model(vit\_model, test\_loader)  print(f"Точность ViT: {final\_vit\_acc:.2f}%")  # Сохранение графиков обучения  plt.figure()  plt.plot(range(1, num\_epochs+1), cnn\_losses, marker='o')  plt.xlabel("Эпоха")  plt.ylabel("Loss")  plt.title("Рисунок 4.1 – Потери CNN")  plt.grid(True)  plt.savefig(os.path.join(logs\_dir, "CNN\_loss.png"))  plt.close()  plt.figure()  plt.plot(range(1, num\_epochs+1), cnn\_acc\_list, marker='o', color='green')  plt.xlabel("Эпоха")  plt.ylabel("Accuracy (%)")  plt.title("Рисунок 4.2 – Точность CNN")  plt.grid(True)  plt.savefig(os.path.join(logs\_dir, "CNN\_accuracy.png"))  plt.close()  plt.figure()  plt.plot(range(1, num\_epochs+1), vit\_losses, marker='o', color='red')  plt.xlabel("Эпоха")  plt.ylabel("Loss")  plt.title("Рисунок 4.3 – Потери ViT")  plt.grid(True)  plt.savefig(os.path.join(logs\_dir, "ViT\_loss.png"))  plt.close()  plt.figure()  plt.plot(range(1, num\_epochs+1), vit\_acc\_list, marker='o', color='purple')  plt.xlabel("Эпоха")  plt.ylabel("Accuracy (%)")  plt.title("Рисунок 4.4 – Точность ViT")  plt.grid(True)  plt.savefig(os.path.join(logs\_dir, "ViT\_accuracy.png"))  plt.close()  # Получаем примеры предсказаний для CNN  cnn\_correct, cnn\_incorrect = get\_prediction\_examples(cnn\_model, test\_loader, num\_examples=1)  if cnn\_correct:  save\_prediction\_example(cnn\_correct[0], "CNN\_correct.png", "CNN")  if cnn\_incorrect:  save\_prediction\_example(cnn\_incorrect[0], "CNN\_incorrect.png", "CNN")    # Получаем примеры предсказаний для ViT  vit\_correct, vit\_incorrect = get\_prediction\_examples(vit\_model, test\_loader, num\_examples=1)  if vit\_correct:  save\_prediction\_example(vit\_correct[0], "ViT\_correct.png", "ViT")  if vit\_incorrect:  save\_prediction\_example(vit\_incorrect[0], "ViT\_incorrect.png", "ViT") |