|  |
| --- |
|  |
| 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 3](#_Toc197250833)

[1 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ ПО МЕТОДУ «ТРАНСФОРМЕР» С НАБОРОМ ДАННЫХ 4](#_Toc197250834)

[**1.1** **Описание алгоритма** 4](#_Toc197250835)

[**1.2** **Программная реализация обучения нейронной сети по методу «Трансформер»** 5](#_Toc197250836)

[2 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ МЕТОДОМ «ГЕНЕРАТИВНАЯ СЕТЬ (DCGAN)» 6](#_Toc197250837)

[**2.1** **Описание алгоритма** 6](#_Toc197250838)

[**2.2** **Программная реализация обучения нейронной сети методом «Генеративная сеть (DCGAN)»** 7](#_Toc197250839)

[3 ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ИСПОЛЬЗОВАНИЕМ МЕТОДА ГРАФОВЫХ НЕЙРОННЫХ СЕТЕЙ (GCN) 10](#_Toc197250840)

[**3.1** **Описание алгоритма** 10](#_Toc197250841)

[**3.2** **Программная реализация обучения нейронной сети с использованием алгоритма обратного распространения ошибки** 10](#_Toc197250842)

[ЗАКЛЮЧЕНИЕ 12](#_Toc197250843)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 13](#_Toc197250844)

[ПРИЛОЖЕНИЯ 14](#_Toc197250845)

**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.

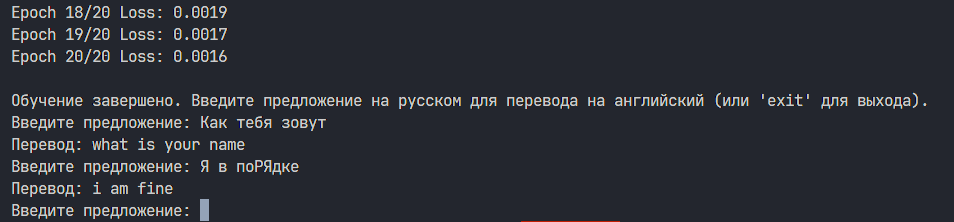
* Класс TransformerModel объединяет работу энкодера и декодера, а также включает финальный линейный слой, преобразующий декодированные представления в распределение вероятностей по словарю целевого языка. При обучении используется механизм teacher forcing, а при тестировании – жадное декодирование.

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:

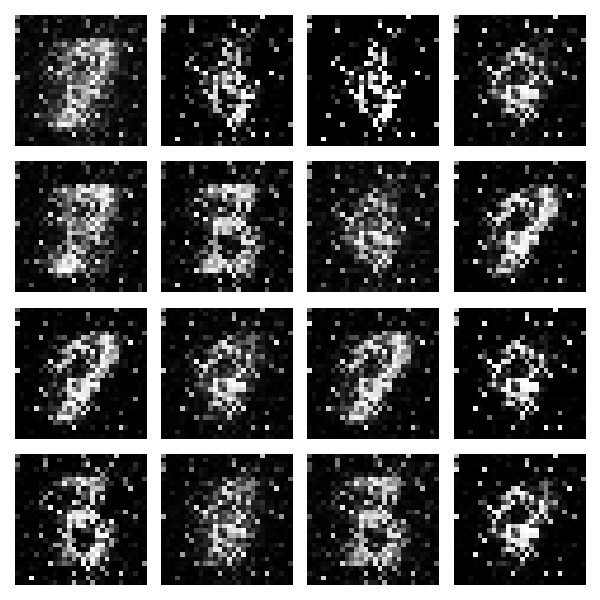
Загружается датасет изображений (например, MNIST или логотипы футбольных команд) с использованием необходимых преобразований (изменение размера, нормализация и т.д.).

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

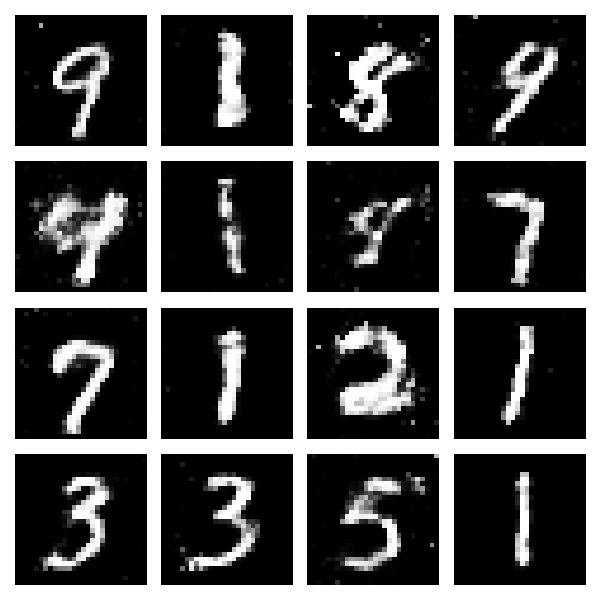
Настраиваются оптимизаторы и функция потерь (BCEWithLogitsLoss).

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:

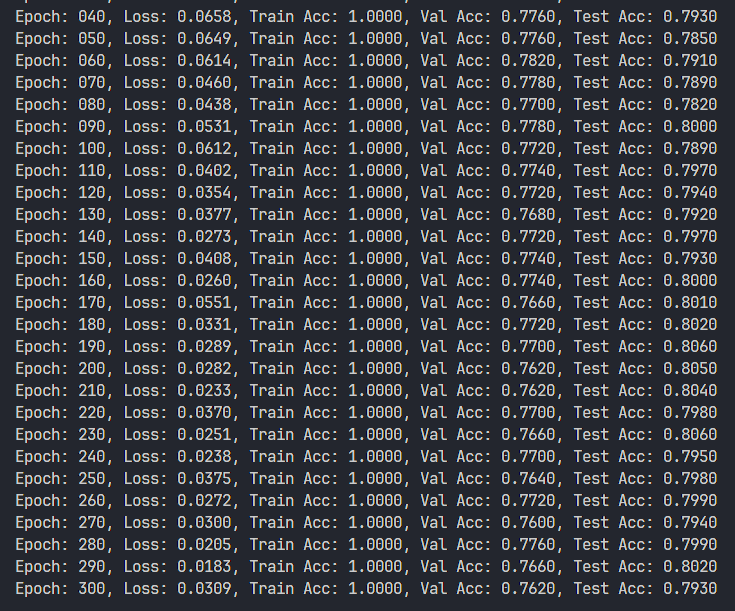
* Датасет CORA используется для представления графа, где узлы содержат признаки и метки классов.

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)**

**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 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}') |