УДК 004.89

Программная реализация алгоритма генеративных состязательных сетей для задач синтеза изображений

М.О. Лукащук[*1], С.А. Зори[*2]

[*1] магистрант, Донецкий национальный технический университет,
mikhail.lukashchuk@mail.ru

[*2] д.т.н., доцент, Донецкий национальный технический университет,
ik.ivt.rec@mail.ru, OrcID: 0000-0003-4018-234X, SPIN-код: 3565-6330

Аннотация

В статье рассмотрена программная реализация алгоритма генеративных состязательных сетей для задач синтеза изображений. Представлены этапы подготовки данных, проектирования архитектуры генератора и дискриминатора, а также методика обучения модели с использованием комбинированных функций потерь, рассмотрена оценка качества работы генератора. В ходе обучения модели были достигнуты значимые результаты, подтверждённые количественными метриками качества PSNR и SSIM. Перспективами дальнейшего развития являются оптимизация процесса обучения для повышения устойчивости модели, внедрение модификаций архитектуры GAN, а также расширение функциональности системы за счёт поддержки более сложных и разнообразных типов входных данных.

Введение

Современные достижения в области искусственного интеллекта и машинного обучения открывают новые возможности для синтеза и генерации данных [1, 2]. Одним из наиболее революционных направлений в этой области стали генеративные состязательные сети (GAN), которые позволили добиться значительного прогресса в создании фотореалистичных изображений, синтетических видео, текстов, музыки и других видов данных [3].

Принцип работы GAN основан на взаимодействии двух нейронных сетей — генератора и дискриминатора, которые обучаются в процессе состязательной игры. Генератор стремится создавать данные, максимально похожие на реальные, в то время как дискриминатор пытается отличить синтетические данные от настоящих. Благодаря такому подходу обе модели постепенно совершенствуют свои способности, что позволяет добиваться впечатляющих результатов в генеративных задачах.

Реализация алгоритма требует тщательной настройки архитектуры нейросетей, выбора оптимальных методов обучения, обеспечения стабильности сходимости, а также эффективной обработки обучающих данных. Ошибки на любом из этапов могут привести к коллапсу модели или получению низкокачественных результатов.

Целью данной работы является разработка программной реализации базового алгоритма GAN с использованием современных инструментов глубокого обучения и программных моделей алгоритмов GAN.

Теоретические основы алгоритма GAN

Идея алгоритма генеративных состязательных сетей заключается в построении двух нейронных сетей — генератора и дискриминатора, которые обучаются в процессе конкуренции друг с другом. В классическом варианте GAN: Генератор принимает на вход случайный вектор, сгенерированный, например, из нормального распределения, и преобразует его в синтетический пример данных (например, изображение). Дискриминатор принимает на вход либо реальный пример из обучающего набора данных, либо пример, созданный генератором, и должен определить, является ли он подлинным или сгенерированным.

Процесс обучения устроен таким образом, что Генератор стремится "обмануть" дискриминатор, создавая всё более реалистичные данные, а Дискриминатор старается точнее различать реальные и искусственно сгенерированные данные.

Их взаимодействие можно представить как двухстороннюю игру с нулевой суммой, где улучшение одной модели приводит к усложнению задачи для другой.

Проектирование программной реализации алгоритма GAN, выбор инструментария

Программная реализация алгоритма GAN в рамках данного проекта ориентирована на обработку видеофайлов с последующей генерацией новых кадров, имитирующих заданные характеристики. При проектировании

решения особое внимание уделялось модульности кода, обеспечению воспроизводимости экспериментов и удобству последующего масштабирования.

Для реализации программной модели GAN выбран язык программирования Python. В качестве основной платформы для построения и обучения нейронных сетей используется библиотека TensorFlow 2.x [4]. Выбор TensorFlow обусловлен его высокой производительностью, наличием встроенной поддержки работы на GPU и TPU, а также широкими возможностями для масштабирования моделей от локальных экспериментов до промышленного развертывания. Кроме того, TensorFlow обеспечивает стабильную поддержку современных алгоритмов оптимизации и интеграцию с популярными инструментами мониторинга обучения, такими как TensorBoard.

В рамках работы с TensorFlow используется высокоуровневый API Keras [5], который позволяет быстро проектировать и тренировать сложные архитектуры нейронных сетей благодаря удобной модульной структуре и гибкой системе компоновки слоёв. Keras предлагает понятный и лаконичный синтаксис, что значительно ускоряет процесс разработки, минимизирует вероятность ошибок и упрощает модификацию модели в процессе экспериментов.

Для обработки графических данных выбрана библиотека OpenCV [6]. OpenCV предоставляет мощные инструменты для захвата видеопотоков, извлечения и обработки отдельных кадров, изменения размеров изображений, а также их предварительной нормализации. Выбор данной библиотеки обусловлен её высокой производительностью, поддержкой широкого спектра форматов мультимедиа и кроссплатформенностью.

Оценка качества сгенерированных изображений проводилась с использованием метрик PSNR (Peak Signal-to-Noise Ratio) и SSIM (Structural Similarity Index), реализованных в библиотеке scikit-image [7]. Scikit-image представляет собой надёжный и проверенный инструмент для обработки и анализа изображений в Python, предоставляя широкий набор метрик для объективной оценки качества визуальных данных. Применение PSNR и SSIM позволяет количественно оценить степень приближения сгенерированных изображений к реальным, что является главным для оценки эффективности работы генеративной модели.

Архитектура программной системы

Проект состоит из следующих основных модулей:

Извлечение кадров из видео: с помощью OpenCV осуществляется разбивка видеофайла на отдельные изображения.

Предобработка данных: кадры приводятся к фиксированному размеру (256×256 пикселей) и нормализуются в диапазоне [0,1] для оптимальной работы нейросетей.

Построение генератора: последовательная модель с несколькими слоями Conv2DTranspose [8], предназначенная для поэтапного увеличения размерности изображения.

Построение дискриминатора: сверточная нейронная сеть, выполняющая задачу бинарной классификации реальных и сгенерированных изображений.

Определение функций потерь: для дискриминатора используется бинарная кросс-энтропия, для генератора – комбинация контентной потери и состязательной потери.

Цикл обучения: чередующееся обновление генератора и дискриминатора с использованием оптимизаторов Adam [9].

Оценка результатов: вычисление средних значений PSNR и SSIM для оценки качества синтезированных кадров.

Модульная структура кода

Программный код структурирован в логические блоки, обеспечивающие удобство тестирования и масштабирования:

– функция для извлечения кадров;
– функция предобработки кадров;
– построение архитектуры генератора;
– построение архитектуры дискриминатора;
– функции вычисления потерь;
– функция одной итерации обучения;
– функция для оценки качества генерации.

Таким образом, архитектура системы позволяет эффективно реализовать полный цикл работы GAN – от подготовки данных до оценки качества сгенерированных кадров.

Описание алгоритма и реализация

Реализация алгоритма генеративных состязательных сетей требует строгого следования установленной архитектуре взаимодействия между генератором и дискриминатором, правильной организации процесса обучения, а также тщательной предобработки данных. В данной реализации программной системы GAN весь процесс был разделён на несколько последовательных этапов, каждый из которых реализует конкретную функцию.

Извлечение кадров из видео

На первом этапе реализации осуществляется подготовка обучающих данных путём извлечения кадров из видеопотока. Работа с видео предполагает его покадровое считывание

с последующим сохранением каждого отдельного кадра. Это позволяет сформировать статичный набор изображений, которые в дальнейшем будут использоваться для обучения генеративной модели. Выбор этого подхода обусловлен необходимостью работать с реальными данными, а также потребностью в большом объёме разнообразных кадров для достижения высокой генеративной способности сети. Каждый кадр сохраняется в виде отдельного изображения с уникальным именем, обеспечивая корректную индексацию данных. На этом этапе закладывается основа для всех последующих стадий работы алгоритма, поскольку качество исходных данных напрямую влияет на эффективность обучения. Основная функция для этого этапа представлена на рисунке 1.

```
import cv2
import os

def extract_frames(video_path, output_folder):
    # Создание папки для вывода
    os.makedirs(output_folder, exist_ok=True)

    # Открытие файла с видео
    cap = cv2.VideoCapture(video_path)

    frame_count = 0
    while True:
        # Чтение фрейма из видео
        ret, frame = cap.read()

        if not ret:
            break

        # Сохранение фрейма как картинки
        output_path = os.path.join(output_folder, f"frame_{frame_count:04d}.jpg")
        cv2.imwrite(output_path, frame)

        frame_count += 1

    cap.release()
```

Рисунок 1 – Извлечение кадров из видео

Предобработка данных

После извлечения кадров требуется этап их предобработки. Предобработка включает изменение размеров всех кадров до единого стандартизированного формата 256 на 256 пикселей. Это позволяет стандартизировать входные данные и избежать ошибок, связанных с различием размеров изображений. Кроме того, проводится нормализация значений пикселей: значения, изначально находящиеся в диапазоне от 0 до 255, приводятся к диапазону от 0 до 1. Такая нормализация необходима для стабилизации процесса обучения и предотвращения возникновения слишком больших градиентов. Предобработанные изображения сохраняются в отдельную директорию, что обеспечивает возможность многократного и быстрого доступа к ним без необходимости повторной обработки. Этот этап существенно увеличивает скорость обучения моделей и повышает качество итогового синтеза. Основная функция для этого этапа представлена на рисунке 2.

```
def preprocess_frames(input_folder, output_folder, target_size=(256, 256)):
    os.makedirs(output_folder, exist_ok=True)

    for filename in os.listdir(input_folder):
        # Читает фрейм
        frame = cv2.imread(os.path.join(input_folder, filename))

        # Преобразует в нужное разрешение
        frame = cv2.resize(frame, target_size)

        # Нормализует значение пикселей
        frame = frame.astype('float32') / 255.0

        # Сохраняет фрейм
        output_path = os.path.join(output_folder, filename)
        cv2.imwrite(output_path, frame)
```

Рисунок 2 – Предобработка данных

Построение генератора

Следующим шагом является построение архитектуры генератора, который является центральным элементом всей генеративной модели. Генератор предназначен для преобразования случайного шума, полученного из нормального распределения, в реалистичное изображение. Архитектура сети строится таким образом, чтобы постепенно увеличивать пространственное разрешение выходного изображения. На начальном этапе входной вектор шума преобразуется в плотное представление с помощью полносвязного слоя. Далее через последовательность слоёв транспонированной свёртки осуществляется поэтапное увеличение размера изображения. На каждом промежуточном уровне используются функции активации LeakyReLU [10], что позволяет избежать эффекта "затухания" градиентов и способствует более стабильному обучению. Завершающий слой применяет функцию tanh (гиперболический тангенс), обеспечивая нормализацию выходных данных в диапазоне от -1 до 1, что соответствует масштабу нормализованных реальных изображений. Конструкция генератора спроектирована таким образом, чтобы создавать максимально фотореалистичные изображения, которые могли бы "обмануть" дискриминатор. Основная функция для этого этапа представлена на рисунке 3.

```
from tensorflow.keras import layers, models

def build_generator():
    model = models.Sequential()

    # Добавляет несколько слоев
    model.add(layers.Dense(64 * 64 * 256, input_dim=noise_dim))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Reshape((64, 64, 256)))

    model.add(layers.Conv2DTranspose(128, kernel_size=5, strides=2, padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2)

    model.add(layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))

    # Выходной слой
    model.add(layers.Conv2DTranspose(3, kernel_size=5, strides=2, padding='same', activation='tanh'))

    return model
```

Рисунок 3 – Построение генератора

Построение дискриминатора

Параллельно с построением генератора проектируется дискриминатор – вторая ключевая составляющая GAN, выполняющая роль "критика" для искусственных данных. Дискриминатор представляет собой сверточную нейронную сеть, которая принимает на вход изображение и должна определить, является ли оно реальным или сгенерированным. Архитектура дискриминатора включает последовательные слои свёртки с последующим применением функций активации LeakyReLU, что способствует улучшению обработки признаков и ускоряет обучение. После нескольких уровней свёртки данные, а затем проходят через полносвязный выходной слой с активацией сигмоиды, который формирует вероятность принадлежности изображения к классу реальных. Основной задачей дискриминатора является обучение максимально точному распознаванию различий между реальными и синтетическими изображениями, что стимулирует генератор к созданию всё более качественных результатов. Процесс построения дискриминатора должен обеспечивать баланс между его способностью различать изображения и сохранением возможностей для генератора получать полезный обучающий сигнал. Основная функция для этого этапа представлена на рисунке 4.

```
def build_discriminator():
    model = models.Sequential()

    model.add(layers.Conv2D(64, kernel_size=5, strides=2, padding='same', input_shape=(256, 256, 3)))
    model.add(layers.LeakyReLU(alpha=0.2))

    model.add(layers.Conv2D(128, kernel_size=5, strides=2, padding='same'))
    model.add(layers.LeakyReLU(alpha=0.2))

    model.add(layers.Flatten())

    model.add(layers.Dense(1, activation='sigmoid'))

    return model
```

Рисунок 4 – Построение дискриминатора

Определение функций потерь

Для успешного обучения генератора и дискриминатора необходима корректная формализация функций потерь (рис. 5). Для дискриминатора в качестве функции потерь используется бинарная кросс-энтропия между предсказанными метками и истинными метками (реальное изображение или поддельное). Это позволяет эффективно наказывать модель за ошибочные классификации и усиливать её способность различать данные. Для генератора формулируется комбинированная функция потерь, состоящая из двух частей. Первая часть — это среднеквадратичная ошибка (MSE) между сгенерированным изображением и желаемым целевым изображением, что позволяет контролировать соответствие содержания. Вторая часть — состязательная потеря, представляющая собой бинарную кросс-энтропию, рассчитываемую на выходе дискриминатора, при которой генератор стремится "обмануть" дискриминатор. Функция представлена на рисунке 6.

```
def discriminator_loss(real_predictions, fake_predictions):
    real_loss = tf.reduce_mean(tf.keras.losses.binary_crossentropy(tf.ones_like(real_predictions), real_predictions))
    fake_loss = tf.reduce_mean(tf.keras.losses.binary_crossentropy(tf.zeros_like(fake_predictions), fake_predictions))
    total_loss = real_loss + fake_loss
    return total_loss
```

Рисунок 5 – Расчет потерь для дискриминатора

```
import tensorflow as tf

# Среднеквадратическое отклонение
def content_loss(desired_frame, generated_frame):
    return tf.reduce_mean(tf.square(desired_frame - generated_frame))

# Бинарная кросс-энтропия
def adversarial_loss(fake_predictions):
    return tf.reduce_mean(tf.keras.losses.binary_crossentropy(tf.ones_like(fake_predictions), fake_predictions))
```

Рисунок 6 – Расчет потерь для генератора

Процесс обучения модели

Обучение генеративных состязательных сетей строится на попеременном обновлении параметров генератора и дискриминатора с целью достижения равновесия между их возможностями. На каждой итерации обучения первым шагом осуществляется обновление дискриминатора. В рамках этого этапа дискриминатор получает на вход реальный кадр из обучающей выборки и сгенерированный кадр, созданный генератором. Для реальных данных дискриминатор должен выдать высокую вероятность подлинности, а для искусственных – низкую. На основании бинарной кросс-энтропии рассчитывается функция потерь дискриминатора, после чего с помощью механизма автоматического дифференцирования вычисляются градиенты, и обновляются веса модели с применением оптимизатора Adam. Вторым шагом в рамках одной итерации является обновление генератора. Генератор создает новый набор изображений на основе случайных векторов шума, передаёт их дискриминатору и получает обратную связь в виде функции потерь (рис. 7). Потери генератора рассчитываются как сумма контентной составляющей (различие между ожидаемым и сгенерированным изображением) и состязательной составляющей (стремление "обмануть" дискриминатор). После расчета потерь также осуществляется вычисление градиентов и обновление параметров генератора с использованием оптимизатора. Такой двухступенчатый процесс позволяет поддерживать динамическое соревнование между двумя сетями, в ходе которого каждая из них стремится улучшить собственную производительность.

Структура тренировочного цикла

Тренировочный цикл организован в виде серии эпох, каждая из которых включает в себя последовательную обработку всех доступных обучающих данных пакетами фиксированного размера. В рамках каждой эпохи для каждого батча проводится полная процедура обучения: обновление дискриминатора и обновление генератора. После завершения обработки всех батчей накапливается статистика потерь, что позволяет контролировать динамику обучения. Для повышения стабильности процесса через определенные интервалы времени осуществляется сохранение весов моделей и промежуточных результатов генерации. Это обеспечивает возможность восстановления обучения в случае прерывания процесса, а также позволяет отслеживать прогресс на различных стадиях тренировки.

Кроме того, предусмотрена система вывода промежуточных метрик потерь, что позволяет оперативно выявлять возможные проблемы, такие как коллапс генератора или чрезмерное усиление дискриминатора. Программная модель цикла представлена на рисунке 8.

```
@tf.function
def train_step(real_frames, desired_frames):
    # Тренируем дискриминатор
    with tf.GradientTape() as disc_tape:
        generated_frames = generator(tf.random.normal([batch_size, noise_dim]))
        real_predictions = discriminator(real_frames)
        fake_predictions = discriminator(generated_frames)
        disc_loss = discriminator_loss(real_predictions, fake_predictions)

    disc_gradients = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
    disc_optimizer.apply_gradients(zip(disc_gradients, discriminator.trainable_variables))

    # Тренируем генератор
    with tf.GradientTape() as gen_tape:
        generated_frames = generator(tf.random.normal([batch_size, noise_dim]))
        fake_predictions = discriminator(generated_frames)
        content_loss = content_loss(desired_frames, generated_frames)
        adv_loss = adversarial_loss(fake_predictions)
        gen_loss = content_loss_weight * content_loss + adv_loss_weight * adv_loss

    gen_gradients = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gen_optimizer.apply_gradients(zip(gen_gradients, generator.trainable_variables))

    return gen_loss, disc_loss
```

Рисунок 7 – Обучение модели

```
for epoch in range(num_epochs):
    for batch_idx in range(len(data) // batch_size):
        # Случайно выбирает батч видео фрейма
        real_frames_batch = sample_real_frames(batch_size)
        desired_frames_batch = sample_desired_frames(batch_size)

        # Одна итерация тренировки
        gen_loss, disc_loss = train_step(real_frames_batch, desired_frames_batch)

    # Лог для мониторинга
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Generator Loss: {gen_loss}, Discriminator Loss: {disc_loss}")
```

Рисунок 8 – Тренировочный цикл

Оценка качества генерации

Для количественной оценки качества работы генератора используются два основных показателя: PSNR и SSIM. Метрика PSNR измеряет качество восстановления изображения, определяя, насколько сгенерированное изображение близко к эталонному по уровню искажений. Чем выше значение PSNR, тем лучше качество восстановления. Метрика SSIM оценивает структурное сходство между двумя изображениями, принимая во внимание освещенность, контрастность и структуру. Высокие значения SSIM указывают на высокую визуальную идентичность между целевым и сгенерированным изображением.

В процессе оценки на каждом этапе генератор принимает на вход заранее подготовленные контрольные изображения и формирует их сгенерированные версии. Далее для каждой пары изображений рассчитываются значения PSNR и SSIM, после чего выводятся средние значения по всему тестовому набору. Это позволяет объективно оценить прогресс обучения и сравнивать различные версии моделей между собой. Оценка качества генерации представлена на рисунке 9.

```
def evaluate_generator(generator, desired_postures_folder):
    psnr_scores = []
    ssim_scores = []

    for filename in os.listdir(desired_postures_folder):
        # Читает нужный кадр
        desired_frame = cv2.imread(os.path.join(desired_postures_folder, filename))
        desired_frame = cv2.cvtColor(desired_frame, cv2.COLOR_BGR2RGB)

        # Изменение размера и нормализация кадра в соответствии с размером и диапазоном входных данных генератора
        desired_frame = cv2.resize(desired_frame, (256, 256))
        desired_frame = desired_frame.astype('float32') / 255.0

        # Генерирует кадр
        generated_frame = generator(np.random.randn(1, noise_dim)).numpy().squeeze()

        # Вычисляет PSNR и SSIM
        psnr_score = psnr(desired_frame, generated_frame)
        ssim_score = ssim(desired_frame, generated_frame, multichannel=True)

        psnr_scores.append(psnr_score)
        ssim_scores.append(ssim_score)

        # Сохраняет фрейм
        cv2.imwrite(f'generated_postures/{filename}', generated_frame * 255)

    # средние показатели PSNR и SSIM
    mean_psnr = np.mean(psnr_scores)
    mean_ssim = np.mean(ssim_scores)

    return mean_psnr, mean_ssim
```

Рисунок 9 – Оценка качества генерации

Результаты реализации

В ходе реализации проекта была успешно построена и обучена нейросетевая модель, способная синтезировать изображения на основе случайных входных данных. Для количественной оценки качества генерации были применены две объективные метрики: PSNR (Peak Signal-to-Noise Ratio) и SSIM (Structural Similarity Index). Полученные результаты составили:

– PSNR: 18.57;

– SSIM: 0.5620;

Значение PSNR указывает на умеренный уровень визуального шума в сгенерированных изображениях по сравнению с оригиналом, что является типичным результатом для базовой версии GAN без дополнительных методов стабилизации. Показатель SSIM отражает структурное сходство между сгенерированными и реальными изображениями. Значение 0.5620 демонстрирует наличие определённого соответствия между формами, текстурами и пространственными взаимосвязями в изображениях, однако оставляет пространство для дальнейшей оптимизации архитектуры модели и алгоритма обучения. Результатом работа нейросетевой модели является сгенерированное видео на основе обучающих кадров. На рисунке 10 изображен кадр из исходного видео, а на рисунке 11 кадр, полученный на выходе генератора после обучения модели.

Рисунок 10 – Кадр из исходного видео

Рисунок 11 – Кадр из сгенерированного видео

Заключение

В рамках исследования реализован алгоритм генеративных состязательных сетей (GAN) для задач синтеза видео. В статье подробно рассмотрены все этапы построения системы: от подготовки данных и проектирования архитектуры генератора и дискриминатора до определения функций потерь и организации процесса обучения.

Программная реализация выполнена с использованием современных инструментов глубокого обучения, что обеспечило высокую гибкость разработки и стабильность модели на этапах тренировки и применения. В ходе обучения модели достигнуты значимые результаты, подтверждённые количественными метриками качества — PSNR и SSIM.

Перспективами дальнейшего развития проекта являются оптимизация процесса

обучения для повышения устойчивости модели, внедрение модификаций архитектуры GAN, а также расширение функциональности системы за счёт поддержки более сложных и разнообразных типов входных данных.

Литература

1. Мулявин, Д. Е. Расширение возможностей систем генерации изображений путем использования нейронных сетей / Д. Е. Мулявин, Р. В. Мальчева, А. А. Койбаш // Информатика и кибернетика. - Донецк: ДонНТУ, 2024. - № 3 (37). - С. 13-18.

2. Goodfellow, Ian, et al. Generative adversarial nets // Advances in neural information processing systems, 2014.

3. Сметана, В. В. Развитие машинного обучения: от теории к приложениям / В. В. Сметана // Национальная ассоциация ученых. – 2024. – Т. 1. - № 101. – С. 133.

4. TensorFlow. Официальная документация [Электронный ресурс]. – Режим доступа: https://www.tensorflow.org/ – Загл. с экрана.

5. Keras: Deep Learning for Humans [Электронный ресурс]. – Режим доступа: https://keras.io/ – Загл. с экрана.

6. OpenCV: Open Source Computer Vision Library [Электронный ресурс]. – Режим доступа: https://opencv.org/ – Загл. с экрана.

7. scikit-image: Image processing in Python [Электронный ресурс]. – Режим доступа: https://scikit-image.org/ – Загл. с экрана.

8. ConvTranspose2d — PyTorch 2.6 documentation [Электронный ресурс]. – Режим доступа: https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html – Загл. с экрана.

9. Adam (Adaptive Moment Estimation) [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/adam-optimizer-in-tensorflow/ – Загл. с экрана.

10. LeakyReLU — PyTorch 2.6 documentation [Электронный ресурс]. – Режим доступа: https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html – Загл. с экрана.

Лукащук М.О., Зори С.А. Программная реализация алгоритма генеративных состязательных сетей. *В статье рассмотрена программная реализация алгоритма генеративных состязательных сетей для задач синтеза изображений. Представлены этапы подготовки данных, проектирования архитектуры генератора и дискриминатора, а также методика обучения модели с использованием комбинированных функций потерь, рассмотрена оценка качества работы генератора. В ходе обучения модели были достигнуты значимые результаты, подтверждённые количественными метриками качества PSNR и SSIM. Перспективами дальнейшего развития являются оптимизация процесса обучения для повышения устойчивости модели, внедрение модификаций архитектуры GAN, а также расширение функциональности системы за счёт поддержки более сложных и разнообразных типов входных данных.*

Ключевые слова: *синтез видео, GAN, TensorFlow, генератор, дискриминатор, PSNR, SSIM.*

Lukashchuk M.O., Zori S.A. Software Implementation of the Generative Adversarial Network Algorithm. *The article discusses the software implementation of the Generative Adversarial Network (GAN) algorithm for image synthesis tasks. It presents the stages of data preprocessing, the design of the generator and discriminator architectures, and the training methodology using combined loss function, the quality of the generator's output is evaluateds. During the training of the model, significant results were achieved, confirmed by the quantitative quality metrics PSNR and SSIM. The prospects for further development include optimizing the learning process to increase the stability of the model, introducing modifications to the GAN architecture, as well as expanding the functionality of the system by supporting more complex and diverse types of input data.*

Keywords: *video synthesis, GAN, TensorFlow, generator, discriminator, PSNR, SSIM.*

Статья поступила в редакцию 17.03.2025
Рекомендована к публикации профессором Мальчевой Р. В.