Санкт-Петербургский государственный университет

Кафедра информатики

Группа 23.Б16-мм

Реализация и внедрение архитектуры Gated Recurrent Unit в пакет DEGANN

Мурадян Денис Степанович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
доцент кафедры ситемного программирования, к.ф.-м.н., Гориховский В. И.

Консультант:
Преподаватель СПбГУ, Алимов П. Г.

Санкт-Петербург
2024

Оглавление

Введение

Дифференциальные уравнения являются важным инструментом для описания процессов во многих областях науки и техники, включая физику, химию, биологию и экономику. Однако с усложнением и увеличением размерности задачи традиционные численные методы решения дифференциальных уравнений начинают сталкиваться с проблемами, связанными с длительным временем вычислений и недостаточной точностью. Это стимулирует поиск новых подходов к решению таких задач, одним из которых является пакет нейросетевой аппроксимации дифференциальных уравнений DEGANN [9].

Однако на текущий момент в пакете реализована только одна архитектура нейронной сети — полносвязная сеть, что ограничивает возможности системы в аппроксимации сложных уравнений. Это создаёт необходимость в расширении пакета за счет добавления новых архитектур нейронных сетей, которые позволят улучшить точность и эффективность работы системы.

Рекуррентные нейронные сети (*Recurrent Neural Networks*, RNN[7]) представляют собой перспективный подход для задач аппроксимации дифференциальных уравнений, поскольку, в отличие от полносвязных сетей, они используют механизм запоминания состояний предыдущих слоёв. Это позволяет учитывать накопленную информацию из предыдущих шагов, что делает их более эффективными для моделирования сложных зависимостей и улучшает согласованность выходных значений. Однако RNN имеют известную проблему затухания градиента, из-за чего обучение становится затруднительным на длинных последовательностях.

Архитектура GRU (*Gated Recurrent Unit* — «Рекуррентный блок с управляемыми элементами»[4]) является усовершенствованной версией RNN и частично решает эту проблему благодаря использованию специальных элементов управления потоком данных — “элемента обнавления” (update gate) и “элемента сброса” (reset gate). Эти элементы регулируют, какую часть информации из предыдущих состояний сле-

дует сохранить, а какую — игнорировать, что позволяет сети избегать накопления нерелевантной информации. За счёт этого GRU демонстрирует высокую стабильность в обучении, сохраняя способность учитывать долгосрочные зависимости. Такое сочетание компактной структуры (меньшее количество параметров по сравнению с LSTM[5]) и способности эффективно моделировать зависимости делает GRU особенно подходящей для задач, связанных с аппроксимацией решений сложных дифференциальных уравнений.

Целью данной работы является реализация архитектуры GRU в пакете DEGANN[9] и проведение экспериментов, демонстрирующих её в задачах аппроксимации решений дифференциальных уравнений. Это позволит не только расширить функционал пакета, но и повысить его практическую применимость для сложных задач в научных и инженерных приложениях.

1. Постановка задачи

Целью работы является расширение функциональности пакета DEGANN[9] — путём внедрения рекуррентных нейронных сетей с архитектурой GRU (Gated Recurrent Unit)[4].

Для её выполнения были поставлены следующие задачи:

Общая архитектура решения

1. Реализация модуля с топологией рекуррентной сети со слоями tensorflow для архитектуры GRU и внедрение его в пакет DEGANN (раздел 3.1);

2. Создание модели с архитектурой GRU. Создание и интеграция callback-функции для расширения функциональности обучения и его трекинга (раздел 3.2);

Валидация решения

1. Создание датасета для корректной передачи данных в модель. Реализация сложно-аппроксимируемой функции для тестирования (раздел 4.1);

2. Подбор метрики, настройка гиперпараметров (раздел 4.2);

А так же обучить модель, продемонстрировать результаты в качестве сравнения значений лосс-функций и графиков аппроксимации.

2. Обзор

В данном разделе представлен обзор рекуррентных нейронных сетей (RNN) и её усовершенствованной архитектуры GRU (Gated Recurrent Unit). Рассмотрены ключевые принципы работы GRU, включая механизм управления потоками информации, а также её особенности и преимущества в задачах обработки последовательных данных.

2.1. Архитектура и принцип работы GRU

Рекуррентные нейронные сети (RNN, Recurrent Neural Networks)[7] были впервые предложены Дэвидом Румельхартом, Джеффри Хинтоном и Рональдом Уильямсом в 1986 году. Они разработали метод, позволяющий учитывать временные зависимости в последовательных данных с помощью скрытых состояний, которые сохраняют информацию о предыдущих шагах.

Одной из главных проблем стандартных RNN является **затухание градиентов**. Это явление связано с использованием цепного правила при обратном распространении ошибки: производные функции активации (например, сигмоида или гиперболический тангенс) уменьшаются на каждом шаге. При длинных последовательностях значения градиентов становятся настолько малыми, что веса перестают обновляться. Это затрудняет обучение зависимостей на больших временных промежутках. В некоторых случаях возможны и **взрывные градиенты**, когда значения градиентов увеличиваются экспоненциально, что делает обучение нестабильным.

Для частичного решения этой проблемы в 2014 году Кён Чо и его коллеги предложили архитектуру GRU (Gated Recurrent Unit). В её основе лежит использование двух гейтов: **обновляющего гейта** и **сбрасывающего гейта**, которые позволяют гибко управлять сохранением и сбросом информации.

1. **Обновляющий гейт** (z_t) отвечает за то, какую часть информа-

ции из предыдущего скрытого состояния следует сохранить:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z),$$

где W_z и b_z — параметры обновляющего гейта, h_{t-1} — скрытое состояние на предыдущем шаге, x_t — входные данные, а σ — сигмоида.

2. **Сбрасывающий гейт** (r_t) определяет, какую часть информации из предыдущего состояния следует игнорировать:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r).$$

3. **Новое скрытое состояние** (h_t) вычисляется следующим образом:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t,$$

4. где **промежуточное состояние** ($\tilde{h}_t$) рассчитывается по формуле:

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h).$$

Такая структура позволяет GRU эффективно решать проблему затухания градиентов, сохраняя долгосрочные зависимости в данных. В отличие от архитектуры LSTM (Long Short-Term Memory), GRU имеет более простую структуру, поскольку использует меньше параметров и не включает отдельной ячейки памяти. Это делает GRU менее вычислительно затратной и удобной для использования в задачах, требующих обработки последовательных данных.

2.2. Рекуррентные слои в TensorFlow

Для реализации архитектуры GRU в рамках работе использованы встроенные слои рекуррентных нейронных сетей из библиотеки TensorFlow, такие как **tf.keras.layers.GRU**[8]. Эти слои предоставляют реализацию гейтов и скрытых состояний, включая механизмы обновления и сброса, что позволяет сосредоточиться на настройке модели и

анализе её поведения, а так же настройке гиперпараметров и экспериментах.

2.3. Выводы

Обзор существующих решений показывает, что архитектура GRU является эффективным и вычислительно выгодным методом для обработки последовательных данных, решая основные проблемы классических RNN, такие, как затухание градиентов. Использование TensorFlow и его встроенных слоёв предоставляет гибкие инструменты для реализации и настройки моделей, что ускоряет процесс разработки и экспериментов. Эти технологии обеспечивают надёжную основу для интеграции GRU в проект DEGANN и проведения дальнейших исследований её применимости в задачах аппроксимации решений дифференциальных уравнений.

3. Общая архитектура решения

3.1. Реализация модуля с топологией RNN GRU

Основная идея данного модуля — предоставить возможность создания GRU-сети в рамках существующей архитектуры `DEGANN`. Для этого был разработан класс `TensorflowGRUNet`, который наследуется от `tf.keras.Model`[1] и определяет необходимую логику построения слоёв GRU, их инициализацию, а также способ компоновки выходных данных. Важным моментом является то, что реализация идёт в связке с существующей инфраструктурой `DEGANN`, где выбор конкретной топологии сети (в данном случае GRU) осуществляется через параметр `net_type`, передаваемый при создании экземпляра класса `IModel`.

3.1.1. Основные параметры и их назначение

Класс `TensorflowGRUNet` включает несколько важных параметров, которые определяют архитектуру сети:

- **input_size:** размер входных данных. Задаёт количество входных признаков в данных.
- **block_size:** список, определяющий количество нейронов в каждом слое GRU. Все слои должны иметь одинаковое количество нейронов.
- **output_size:** количество выходных нейронов. Обычно для задач регрессии используется один выходной нейрон.
- **activation_func и recurrent_activation:** функции активации для основного состояния и рекуррентного состояния соответственно. По умолчанию используются `tanh` и `sigmoid`, так как они являются стандартными для GRU.
- **dropout_rate:** уровень dropout, добавляемый после каждого слоя GRU для предотвращения переобучения.

- **weight и biases:** инициализаторы весов и смещений. Используются случайные значения в заданном диапазоне для начальной настройки параметров модели.

- **return_sequences:** флаг, определяющий, будет ли сеть возвращать последовательности — это важно для многослойных рекуррентных сетей.

Данные параметры позволяют гибко настраивать модель под задачу, сохраняя при этом целостность и удобство использования в экосистеме `DEGANN`.

3.1.2. Интеграция в `DEGANN`:

Для внедрения новой архитектуры в `DEGANN` мы используем существующий интерфейс `IModel`. Создание модели осуществляется примерно следующим образом: при инициализации `IModel` мы передаем параметр `net_type="GRUNet"`, что приводит к созданию внутри экземпляра класса сети с топологией `TensorflowGRUNet`.

Создание модели с топологией GRU происходит следующим образом:

Листинг 1: Создание модели с использованием топологии GRUNet

```
count_size = 5  # количествослоёв
gru_units = 30  # количестванейроноввслое
shape = [gru_units] * count_size
GRU_IModel = IModel(
    input_size=1,
    output_size=1,
    net_type="GRUNet",  # топология GRUNet
    block_size=shape)  # передаёмразмерыслоёв
```

В данном случае, внутри класса `IModel` вызывается метод `_create_functions["GRUNet"]`, возвращающий экземпляр

`TensorflowGRUNet`, унаследованный от `tf.keras.Model`. Таким образом, мы сохраняем единообразный интерфейс для работы с моделями различных типов (DenseNet, GRUNet и т.д.) внутри `DEGANN`.

3.1.3. Как работает класс TensorflowGRUNet

При инициализации класс `TensorflowGRUNet` создаёт несколько GRU-слоёв, каждый из которых имеет одинаковое количество нейронов, заданное параметром `block_size`. На этапе инициализации происходит проверка, чтобы все слои сети имели одинаковый размер, поскольку это важное условие для корректной работы рекуррентных нейронных сетей. Если размер слоёв отличается, выбрасывается исключение.

После каждого GRU-слоя добавляется слой Dropout для предотвращения переобучения. Эти слои помогают повысить устойчивость модели, добавляя случайные обнуления нейронов во время тренировки.

3.1.4. Реализация функции call

Функция `call` является важной частью класса `TensorflowGRUNet`, поскольку она выполняет прямой проход через сеть. В этой функции данные проходят через все GRU-слои, а затем через выходной полносвязный слой, который выполняет линейную активацию для задач регрессии.

3.1.5. Особенности реализации

Ключевой особенностью реализации является соблюдение совместимости с другими компонентами библиотеки DEGANN, что позволяет использовать стандартные методы обучения, тестирования и экспорта моделей. Например, параметры модели могут быть экспортированы в виде словаря, что упрощает сохранение и интеграцию в другие модули. Метод `to`$_d ict, , , ,$.

3.2. Создание модели и реализация callback-функций

3.2.1. Создание модели GRU

Для создания модели с использованием топологии GRU использовалась модульная архитектура DEGANN, наследуемую от базового класса IModel, не внося изменений в его структуру. Такой подход позволил адаптировать сеть GRU к задачам аппроксимации дифференциальных уравнений и интегрировать её в существующий функционал проекта.

3.2.2. callbacks

В процессе разработки и обучения модели важно не только корректно настроить её архитектуру, но и обеспечить инструментами, позволяющими наглядно оценивать ход обучения, качество модели и её производительность. Это включает как отслеживание ключевых метрик, таких как функция потерь, так и визуализацию предсказаний модели на различных этапах.

Основные цели анализа обучения и валидации включают:

- **Оценка функции потерь**: Анализ функции потерь на тренировочных и валидационных данных позволяет выявить динамику обучения, а также возможные проблемы, такие как переобучение или недостаточное обучение. (с помощью метода **history** из tf)
- **Раннее прекращение обучения**: Обеспечивает остановку обучения, если модель перестала улучшаться, предотвращая ненужные вычисления и переобучение.
- **Сохранение лучшей модели**: Позволяет фиксировать параметры модели с минимальной валидационной ошибкой, чтобы использовать её для дальнейших экспериментов.

- **Визуализация**: Сравнение предсказаний модели с истинной функцией и визуализация потерь предоставляет интуитивное понимание, насколько хорошо модель аппроксимирует данные.

- **Измерение времени обучения**: Позволяет оценить, насколько эффективно используются вычислительные ресурсы при обучении модели. (встроенный метод MeasureTrainTime в DEGANN)

Для достижения этих целей были реализованы и интегрированы несколько callback-функций, обеспечивающих автоматическое выполнение перечисленных выше задач. Эти callback-функции разработаны как часть архитектуры проекта DEGANN и могут быть легко добавлены при обучении любой модели.

3.2.3. Пример использования callback-функций

Для анализа и визуализации обучения были разработаны и внедрены следующие callback-классы:

- EarlyStopping: Реализует раннее прекращение обучения при отсутствии улучшений в течение заданного количества эпох.

- SaveBestModel: Сохраняет параметры модели с минимальной валидационной ошибкой.

- VisualizationTS: Визуализирует процесс обучения и предсказания модели, сравнивая их с истинной функцией.

Листинг 2: Пример получения данных от callback-функций

```
Epoch 1: finished in: 6.68 seconds | Training Loss: 0.4940,
   Validation Loss: 0.4203
Epoch 2: finished in: 0.97 seconds | Training Loss: 0.4340,
   Validation Loss: 0.4031
...
```

```
Epoch 49: finished in: 0.40 seconds | Training Loss: 0.0597,
   Validation Loss: 0.0272
Epoch 50: finished in: 0.48 seconds | Training Loss: 0.0617,
   Validation Loss: 0.0411
Total training time: 55.3523 seconds
Average time per epoch: 1.107045 seconds
Loss before training = 0.606187
Loss after training = 0.059869
different between Losses =0.546318
validation loss before training = 0.610131
validation loss after training = 0.041137
Difference in validation loss = 0.568993
```

3.2.4. Описание callback-функций

EarlyStopping: Реализует механизм ранней остановки обучения, если валидационная ошибка не уменьшается в течение заданного числа эпох **patience**. Это помогает избежать переобучения и экономит ресурсы.

SaveBestModel: Сохраняет лучшую модель, найденную в процессе обучения, фиксируя параметры сети с минимальной валидационной ошибкой.

VisualizationTS: Генерирует графики, отображающие:

- Динамику функции потерь на тренировочных и валидационных данных.
- Сравнение истинной функции, предсказаний модели и тренировочных данных.

Данные callback-функции позволяют автоматизировать рутинные задачи анализа и визуализации обучения, делая процесс более удобным и наглядным.

4. Валидация решения

4.1. Датасет и сложно-аппроксимируемая функция

4.1.1. Генерация датасета

Для тестирования работы архитектуры GRU-сети был подготовлен специальный датасет, который генерировался с помощью встроенных инструментов библиотеки DEGANN. В качестве сложно-аппроксимируемой функции была выбрана функция hardsin, которая была добавлена в модуль functions.py:

Эта функция обладает сложной математической структурой, что делает её трудно аппроксимируемой стандартными архитектурами нейронных сетей. Её аналитическое выражение имеет следующий вид:

$$f(x) = \sin\left(\ln(x^{\sin(10x)})\right)$$

4.1.2. График функции

На рисунке ниже представлен график функции hardsin, демонстрирующий её сложное поведение:

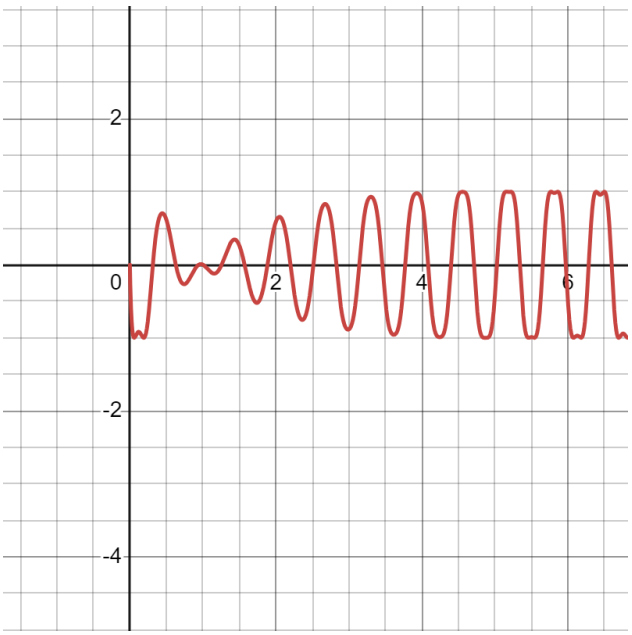

Рис. 1: График функции $f(x) = \sin\left(\ln(x^{\sin(10x)})\right)$

4.1.3. Добавление шума

Для повышения устойчивости модели и её способности к обобщению в процессе обучения на тренировочные данные добавлялся шум, который следовал нормальному распределению. Это позволяет модели лучше адаптироваться к реальным данным, которые часто содержат случайные колебания и погрешности. Нормальное распределение (также называемое гауссовским распределением) является одним из наиболее часто используемых видов распределений в статистике и машинном обучении.

Формула нормального распределения: Нормальное распределение определяется следующей плотностью вероятности:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

где:

- μ - математическое ожидание (среднее значение),
- σ - стандартное отклонение (мера разброса данных),
- x - случайная величина.

Параметры μ и σ в нашем случае были рассчитаны на основе целевой переменной тренировочного датасета train_data_y, чтобы шум соответствовал масштабу данных.

Применение шума в датасете: Добавление шума выполнялось с использованием библиотеки NumPy, где функция np.random.normal() генерирует массив случайных значений, следующих нормальному распределению. В нашем случае среднее значение шума $\mu = 0$, а стандартное отклонение рассчитывается как $0.1 \cdot \sigma$, где σ --- стандартное отклонение значений целевой переменной. Формула добавления шума выглядит следующим образом:

$$\texttt{train\_data\_y\_noisy} = \texttt{train\_data\_y} + \xi, \quad \xi \sim \mathcal{N}(0, 0.1 \cdot \sigma)$$

4.1.4. Загрузка и обработка данных

Для создания тренировочного и валидационного наборов данных использовались встроенные методы генерации датасетов в DEGANN, которые позволяют сохранять данные в формате .csv для последующего использования.

4.1.5. Временные последовательности

Рекуррентные нейронные сети, такие как GRU, предназначены для обработки временных последовательностей. Их ключевая особенность заключается в возможности сохранять информацию о предыдущих состояниях для анализа текущих данных. Это делает их идеальными для задач, где данные зависят от контекста, например, аппроксимация временных рядов.

Для корректной работы RNN данные из исходного датасета были преобразованы в последовательности фиксированной длины с помощью функции create sequences:

Листинг 3: Создание временных последовательностей

```
1 def create_sequences(data_x, data_y, time_steps):
2     x, y = [], []
3     for i in range(len(data_x) - time_steps):
4         x.append(data_x[i:i + time_steps])
5         y.append(data_y[i + time_steps])
6     return np.array(x), np.array(y)
7
8 time_steps = 10
9 train_data_x, train_data_y = create_sequences(train_data_x,
    train_data_y, time_steps)
```

```
10 val_data_x, val_data_y = create_sequences(val_data_x, val_data_y,
    time_steps)
```

В данном примере time_steps задаёт количество шагов в последовательности. Это означает, что каждый входной элемент сети состоит из time_steps предыдущих значений, а целевая переменная соответствует следующему значению функции.

Почему RNN работают с временными последовательностями? Основное преимущество RNN заключается в их способности передавать информацию через скрытые состояния (*hidden states*). В отличие от обычных нейронных сетей, которые обрабатывают каждый вход независимо, RNN сохраняют информацию из предыдущих шагов и используют её для обработки текущего ввода. Это реализуется через рекуррентную связь, которая обновляет скрытое состояние на каждом временном шаге.

4.2. Метрики и настройка гиперпараметров

Для эффективного обучения моделей на основе архитектуры GRU важен тщательный выбор функции потерь, оптимизатора и гиперпараметров. Этот процесс позволяет достичь баланса между качеством аппрокисмации и вычислительными затратами.

4.2.1. Функция потерь

В качестве функции потерь использовалась среднеквадратичная ошибка (*Mean Squared Error*, MSE), определяемая формулой:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

где y_i - истинное значение, $\hat{y}_i$ - предсказание модели, n - количество примеров.

Среднеквадратичная ошибка была выбрана по следующим причинам:

- MSE минимизирует крупные отклонения между истинными и предсказанными значениями, что особенно полезно при аппроксимации сложных функций.
- Эта метрика стандартна для задач регрессии и обеспечивает устойчивую сходимость моделей [2].

Таким образом, MSE является подходящим выбором для задач, связанных с аппроксимацией решений дифференциальных уравнений.

4.2.2. Гиперпараметры модели

Для данной архитектуры были выбраны параметры, исходя из результатов экспериментов и тестов, при которых наблюдался хороший результат при доступном времени обучения и количестве эпох. Проведённые эксперименты позволили определить следующие параметры:

- **Количество слоёв**: 5;
- **Количество нейронов в каждом слое**: 30.

Выбор таких параметров обусловлен следующими факторами:

- Увеличение числа слоёв и нейронов значительно повышает сложность модели и требуют большего времени обучения. При этом избыточное увеличение параметров не всегда приводит к заметному улучшению качества, а ограниченные ресурсы накладывают свои ограничения.
- Уменьшение количества слоёв или нейронов приводит к недообучению модели, что негативно сказывается на качестве аппроксимации функций.
- В выбранной конфигурации модель способна эффективно решать задачи аппроксимации, оставаясь достаточно компактной.

4.2.3. Оптимизатор

Для оптимизации был выбран *Adam*, поскольку он эффективно регулирует скорость обучения для каждого параметра и демонстрирует высокую стабильность в задачах регрессии [3]. *Adam* сочетает преимущества методов *Momentum* и *RMSprop*, что делает его особенно полезным при обучении рекуррентных сетей, таких как GRU.

Преимущества выбора оптимизатора *Adam*:

- Автоматическая адаптация шага обучения для быстрого достижения сходимости.
- Хорошие результаты при обучении моделей, работающих с временными последовательностями [6].
- Стабильность градиентного спуска, что особенно важно для задач с длительными зависимостями.

Таким образом, выбор MSE в качестве функции потерь, использование оптимизатора *Adam* и разумная конфигурация гиперпараметров обеспечивают эффективное обучение модели. Данная настройка параметров позволяет достичь приемлемого качества аппроксимации функций в условиях ограниченных вычислительных ресурсов.

5. Эксперимент

Цель экмперемента - обучить и сравнить модели с полносвязной и рекуррентной архитектурами с одинаковым количеством слоев и нейронов в качестве аппроксимации встроенных в DEGANN функций, а так же на реализованной сложно-аппроксимируемой функции. Сравнить время обучения, показания лосс функций, а так же качество аппроксимации.

5.1. Условия эксперимента

- Количество слоев: 5
- Количество нейронов в каждом слое: 30
- Количество эпох (несколько тестов):
  - 100
  - 200
  - 500
  - 1000

5.2. Исследовательские вопросы

RQ1 : Лучше ли реккурентная сеть справляется с аппроксимацией уравнений, встроенных в DEGANN?

RQ2 : Насколько существенна разница в аппроксимации реккурентной сети и полносвязной?

5.3. Итоги

На данном этапе работы, внедренная архитектура находится на этапе тестирования. Эксперемент еще не до конца завершен и его содержание еще не до конца написано.

Заключение

В рамках данной работы выполнены следующие задачи:

- Разработан и реализован модуль с топологией рекуррентной сети для архитектуры GRU.
- Создана модель с архитектурой GRU и интегрированы callback-функции для оптимизации процесса обучения, включая раннее прекращение, сохранение лучшей модели и визуализацию результатов.
- Подготовлен специальные датасеты для тестирования, а так же добавлена сложно-аппроксимируемая функция hardsin, реализовано добавление шума в датасет для повышения устойчивости модели.
- Проведён подбор метрики (MSE) и настройка гиперпараметров, обеспечивающая хорошие результаты и баланс между качеством аппроксимации и вычислительными затратами.

Работа на данном этапе продемонстрировала успешную интеграцию архитектуры GRU в пакет DEGANN.

Планы на продолжение работы

Для дальнейшего расширения функциональности системы планируется:

- Разработать собственную (кастомную) реализацию слоёв GRU для расширения функциональности настройки сети.
- Завершить тестирование архитектуры на различных наборах данных и провести более глубокий анализ результатов, представив их в итоговом эксперименте.

Код и результаты

Код, написанный в ходе работы, свободно доступен здесь: https://github.com/Denigmma/degann/tree/RNN$_G RU$

- Модуль с топологией находится здесь: topology
- Модель находится здесь: model

Список литературы

[1] Chen Haiping. Документация к tensorflow // TFnet. — 2019. — https://tensorflownet.readthedocs.io/en/latest/.

[2] Goodfellow Ian, Bengio Yoshua, Courville Aaron. Deep Learning // MIT. — 2016. — http://www.deeplearningbook.org.

[3] Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. — USA : O'Reilly Media, 2019. — ISBN: 000.

[4] Jeeva Cathrine. Gated Recurrent Unit // scaler. — 4 May 2023. — https://www.scaler.com/topics/deep-learning/gru-network/#topic-challenge-container-578866.

[5] LSTM Networks // GF.org. — 05 Jun, 2023. — https://www.geeksforgeeks.org/understanding-of-lstm-networks/.

[6] Makinde Ahmad. Optimizing Time Series Forecasting: A Comparative Study of Adam // ar5iv. — 5 Nov 2024. — https://ar5iv.org/html/2410.01843.

[7] Stryker Cole. Recurrent Neural Network // IBM. — 4 October, 2024. — https://www.ibm.com/topics/recurrent-neural-networks.

[8] tensorflow.org. Документация к втроенным слоям GRU в tf. — URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU.

[9] П. Г. Алимпов. Пакет нейросетевой аппроксимации дифференциальных уравнений. — URL: https://github.com/Krekep/degann.