Федеральное государственное автономное образовательное учреждение   
высшего образования «Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»  
НИИМ Нижегородского университета

**УТВЕРЖДАЮ**

Зав. кафедрой ИАНИ

\_\_\_\_\_\_\_\_\_\_\_\_ М.Х. Прилуцкий

\_\_ \_\_\_\_\_\_\_\_ 2022 г.

МОДЕЛИРОВАНИЕ ПОТОКА ЖИДКОСТИ С ИСПОЛЬЗОВАНИЕМ ГЛУБОКОГО ОБУЧЕНИЯ, ОСНОВАННОГО НА ФИЗИЧЕСКИХ МОДЕЛЯХ

**Руководство программиста**

Листов 16

2022

# **Содержание**

[**Содержание** 2](#_Toc105354593)

[**Введение** 3](#_Toc105354594)

[**Модуль RDA.py** 5](#_Toc105354595)

[**Модуль adapter.py** 9](#_Toc105354596)

[**Модуль saver.py** 10](#_Toc105354597)

[**Модуль data\_generator.py** 11](#_Toc105354598)

[**Модуль DfpNet.py** 12](#_Toc105354599)

[**Модуль dataset.py** 14](#_Toc105354600)

[**Модуль utils.py** 15](#_Toc105354601)

[**Модули runTrain.py и runTest.py** 16](#_Toc105354602)

# **Введение**

На Рисунке 1 приведена общая схема модулей. В верхней части (1) – представлена программная часть, а также модуль генерации данных, в нижней части (2) – представлена часть нейронной сети (обучение, тренировка).

main.py – исполняемый модуль, использующий прочие модули, а также включающий в себя вывод в консоль.

RDA.py – модуль алгоритма случайного леса.

adapter.py – адаптер для передачи входных данных в алгоритм случайного леса.

saver.py – модуль сохранения, загрузки данных и интерпретации данных в виде изображений.

DfpNet.py – модуль архитектуры нейронной сети.

data\_generator.py – модуль генерации данных (для тренировки нейронной сети и тестирования ПО).

dataset.py – модуль набора данных и их распределения.

runTrain.py – модуль тренировки нейронной сети.

runTest.py – модуль тестирования нейронной сети.

utils.py – модуль дополнительных функций (вычисление ошибки, вывод результатов тестирования).

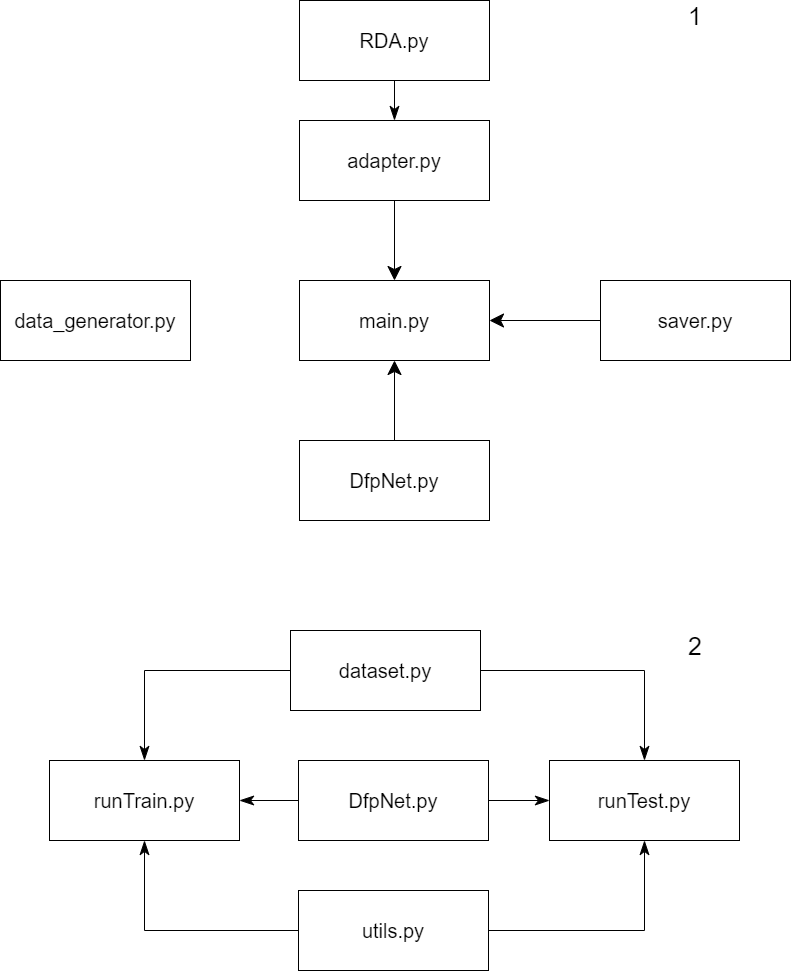


Рисунок 1. Взаимодействие модулей

**Модуль RDA.py**

На Рисунке 2 приведена UML-диаграмма для модуля RDA.py.

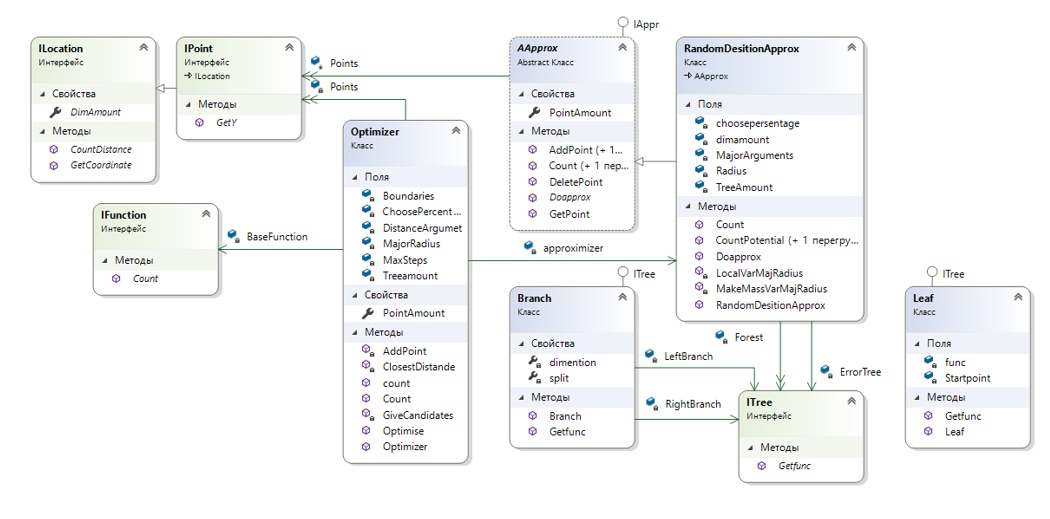


Рисунок 2. RDA.py

Класс **Location**:

Поля:

* dimension – количество измерений.
* coors (list) – список координат.

Методы:

* get\_dim – вернуть количество измерений.
* get\_coor(index) – вернуть координату в индексе index.

Конструкторы:

* по-умолчанию – 0 измерений.
* (coors: list) – столько же измерений, сколько элементов в списке coors, список координат повторяет список coors.

Класс **Point:Location**:

Поля:

* func – значение функции в координатах.

Методы:

* get\_func – вернуть значение func.

Конструкторы:

* по-умолчанию – родительские поля конструируются как в родительском классе, значение func равно 0.
* (coors: list, func) – родительские поля конструируются как в родительском классе, значение поля func равно func.

Класс **ITree**:

Абстрактный класс с единственным методом:

* get\_func(location: Location).

Класс **Leaf**:**ITree**:

Поля:

* start\_p: Point – точка.

Методы:

* get\_func – вернуть значение func в start\_p.

Конструкторы:

* (p: Point) – значение поля starp\_p равно p.

Класс **Branch**:**ITree**:

Поля:

* left\_branch: ITree – левая ветвь дерева (либо Branch, либо Leaf).
* right\_branch: ITree – правая ветвь дерева (либо Branch, либо Leaf).
* global\_max\_between – максимальное расстояние по координатам между заданным списком точек.
* dimension – индекс координаты, по которой определено максимальное расстояние.
* split – значение, по которому распределяются точки по правой и левой ветвям.

Методы:

- get\_func(location: Location) – вычисление значения функции в неизвестной точке, путем прохода через дерево.

Конструкторы:

* (p\_arr: list) – p\_arr является списком точек, которые распределяются по правым и левым ветвям. Среди этого списка вычисляется максимальное расстояние по координатам и заносится в global\_max\_between, индекс координаты с найденным максимальным расстоянием заносится в dimension. split вычисляется как global\_max\_between/2.

Класс **Aprox**:

Поля:

* points: list – список точек, с помощью которых будет происходить аппроксимация.

Методы:

* add\_point(coors: list, func) – сконструировать и добавить объект Point в список points.
* get\_point(index) – вернуть точку под индексом index из списка points.
* delete\_point(index) – удалить точку под индексом index из списка points.
* count(loc: Location) – виртуальный метод.
* count(coors: list) – конструирует объект Location и вызывает count с ним.

Конструкторы:

* По-умолчанию – points – пустой список.

Класс **RDA**:**Aprox**:

Поля:

* forest: list – лес деревьев, через которые будет расчитываться функция в неизвестной точке.
* dim\_am – количество измерений.
* tree\_am – количество деревьев.
* choose\_per – процент деревьев, принимаемых в рассмотрение.
* error\_tree – дерево ошибки.

Методы:

* do\_approx – задание леса forest и дерева ошибки error\_tree
* count(loc: Location) – расчет функции в точке loc, путем расчета среднего значения полученного из каждого дерева в forest с отклонением ошибки error\_tree.

Конструкторы:

* (dim\_am, tree\_am=50, choose\_per=0.8) – points, forest – пустые списки, dim\_am равно dim\_am, tree\_am равно tree\_am, choose\_per равно choose\_per, дерево ошибки пустое.

**Модуль adapter.py**

Функции:

**ret\_nan\_coors(np\_arr)** => **points:list –** вернуть список положений неизвестных точек в массиве NumPy.

**ret\_coors(np\_arr)** => **points:list** – вернуть список положений и известных значений массива NumPy.

**count\_p(arr)** – сконструировать RDA с помощью известных точек массива NumPy arr, рассчитать неизвестные точки и заменить их в массиве arr.

**Модуль saver.py**

Функции:

**save\_np\_img(data, name:str)** – сохранить массив NumPy data в виде изображения с именем name.

**scale\_data(data) => new\_data** – вернуть увеличенную размерность данных массива NumPy data в виде new\_data.

**save\_np\_scaled\_img(data, name:str)** – сохранить увеличенный массив NumPy data в виде изображения с именем name.

**plot\_grid\_val(grid, name:str)** – сохранить сетку PhiFlow data в виде изображения с именем name.

**save\_tensor\_f(data, name:str)** – сохранить тензор PyTorch data в виде файла с именем name.

**read\_tensor\_f(name:str)** => **Tensor** – загрузить файл с именем name и вернуть тензор PyTorch.

**save\_np\_f(data, name:str) –** сохранить массив NumPy в виде файла с именем name.

**read\_np\_f(name:str) => array –** загрузить файл с именем name и вернуть NumPy массив.

**Модуль data\_generator.py**

Функции:

**step(velocity, smoke, pressure, dt=1.0, buoyancy\_factor=1.0, INFLOW=None)** => **velocity, smoke, pressure** – запустить шаг симуляции ламинарного течения жидкости по уравнению Навье-Стокса. velocity, smoke, pressure – сетки скорости, концентрации, давления жидкости соответственно. dt – шаг времени для симуляции, buoyancy\_factor – коэффициент плавучести, INFLOW – сетка источника течения жидкости.

**multi\_step(t, bf, INFLOW=None)** => **velocity, smoke, pressure** – запустить симуляцию в течении времени t, bf – коэффициент плавучести, INFLOW – сетка источника течения жидкости.

**scatter\_noise(data) => new\_data –** нанестишум на данные data и вернуть новые данные new\_data.

**save\_noise\_img(data, name:str)** – сохранить зашумленные данные data в виде изображения с именем name.

**grid\_to\_val(velocity, smoke, pressure) => 4x torch.tensor –** преобразовать сетки в тензоры PyTorch.

**grid\_to\_np(velocity, smoke, pressure) => 4x np.array** – преобразовать сетки в массивы NumPy.

**generate\_data(plot=False, specific\_name=’’) => velocity, smoke, pressure –** случайносгенерировать данные, с возможностью указания специального имени specific\_name, plot указывает строить графики или нет. Случайно задаваемые параметры время, коэффициент плавучести, скорость, положение и размер потока жидкости.

**generate\_simple\_data(plot=False, specific\_name=’’) => velocity, smoke, pressure** – случайносгенерировать упрощенные данные, с возможностью указания специального имени specific\_name, plot указывает строить графики или нет. Случайно задаваемые параметры время, положение и размер потока жидкости.

**save\_simple\_data(plot=False, specific\_name=’’) –** сгенерировать и сохранить упрощенные данные, с возможностью указания специального имени specific\_name, plot указывает строить графики или нет.

**generate\_noised\_data(plot=False) => 4x np.array** – сгенерировать зашумленные данные, plot указывает строить графики или нет.

**save\_noised\_data(plot=False)** – сгенерировать и сохранить зашумленные данные, plot указывает строить графики или нет.

**Модуль DfpNet.py**

Функции:

**weights\_init(m)** – случайное задание весов нейронной сети.

**blockUNet(in\_c, out\_c, name, transposed=False, bn=False, relu=True, size=4, str\_=2, pad=1, dropout=0., sf=2)** => **torch.nn.Sequential()** – создает и возвращает объект класса torch.nn.Sequential(), являющимся блоком нейронной сети. in\_c – количество входных каналов, out\_c – количество выходных каналов, transposed – использовать развертку или свертку, bn – использовать ли нормализацию, relu – использовать функцию активации ReLu или LeakyReLu, size – размер сверточного ядра, str\_ – шаг свертки, pad – отступ при свертке, dropout – возможность зануления канала, sf – множитель пространственного размера для развертки.

Класс **UNet\_**:**torch.nn.Module**:

Поля:

* Сверточные слои (layer1, layer2, layer2b, layer3).
* Разверточные слои (dlayer1, dlayer2, dlayer2b, dlayer3).

Методы:

* **forward(x)** => **dout1:np.array** – прогон входа x с возвращением результата dout1.

Конструкторы:

* **(channelExponent=6, dropout=0.)** – конструирует нейронную сеть согласно архитектуре на рисунке 3, channelExponent определяет количество каналов в каждом слое, dropout – возможность зануления канала.

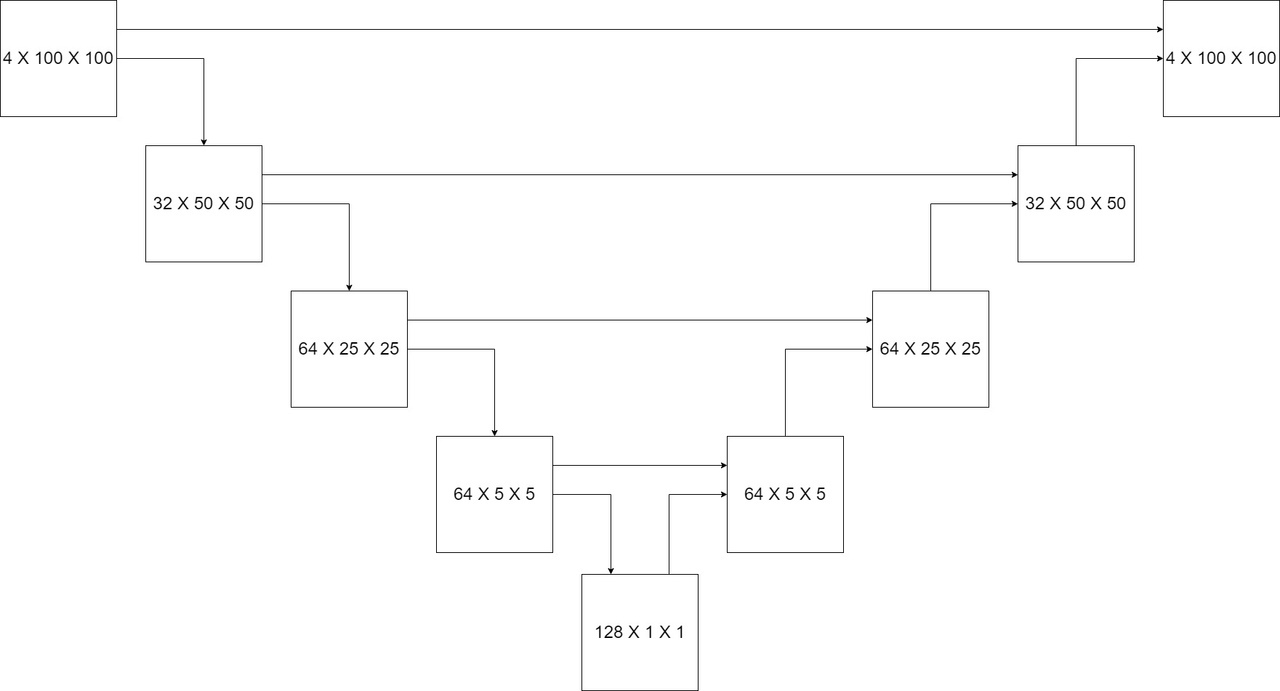


Рисунок 3. Архитектура нейронной сети

**Модуль dataset.py**

Функции:

**LoaderNormalizer(data, isTest=False, ratio=0.8) => data** – считывание данных и занесение их в экземпляр класса TurbDataset – data. isTest – означает берутся ли данные для тестирования, ratio означает долю разделения тренировочных и тестовых данных.

Класс **TurbDataset**:**torch.Dataset**

Поля:

* **mode** – режим набора данных – тренировка или тестирование.
* **dataDir** – путь к главной папке с данными.
* **dataDirTest** – путь к папке с тренировочными данными.
* **totalLength** – количество тренировочных данных.
* **inputs** – входные данные для тренировки.
* **targets** – таргетные данные для тренировки.
* **valiLength** – количество данных для тестирования.
* **valiInputs** – входные данные для тестирования.
* **valiTargets** – таргетные данные для тестирования.

Методы:

* **\_\_len\_\_ => totalLength**
* **\_\_getitem\_\_(idx) => inputs[idx], targets[idx]**

Конструкторы:

* **(mode=TRAIN, dataDir=”../data\_1/train/”, dataDirTest = ”../data\_1/test/”, ratio=0.8)** – параметры приравниваются к соответствующим полям, в конструкторе вызывается функция LoaderNormalizer.

**Модуль utils.py**

Функции:

**log(file, line, doPrint=True)** – записать строчку line в файл file, doPrint указывает печатать ли строчку в консоль или нет.

**resetLog(file) –** переоткрыть файл.

**computLR(i, epochs, minLR, maxLR) –** вычислить скорость обучения с затуханием.

**imageOut(filename, \_outputs, \_targets, saveTargets=False, saveMontage=True) –** сохранить 4 изображения с именем filename, \_outputs – выходные данные, \_targets – таргетные данные, saveTargets – сохранять ли таргентные данные, saveMontage – указывает сохранить ли разницу между выходными и таргетными данными.

**saveAsImage(filename, field\_param) –** сохранить одно изображение с именем filename данных field\_param.

**makeDirs(directoryList) –** создать папки directoryList при их отсутствии.

**Модули runTrain.py и runTest.py**

В модулях runTrain.py и runTest.py функции и классы отсутствует, используется стандартная процедура тренировки и тестирования нейронной сети с использованием средств PyTorch