Платформа 2.0: Новое для разработчиков. PlatformAPI. PlatLib. ATP

Вишняков Борис Ваисович, к.ф.м.-н., начальник подразделения «Искусственный интеллект и анализ данных» ФАУ «ГосНИИАС»

PLAT-2023

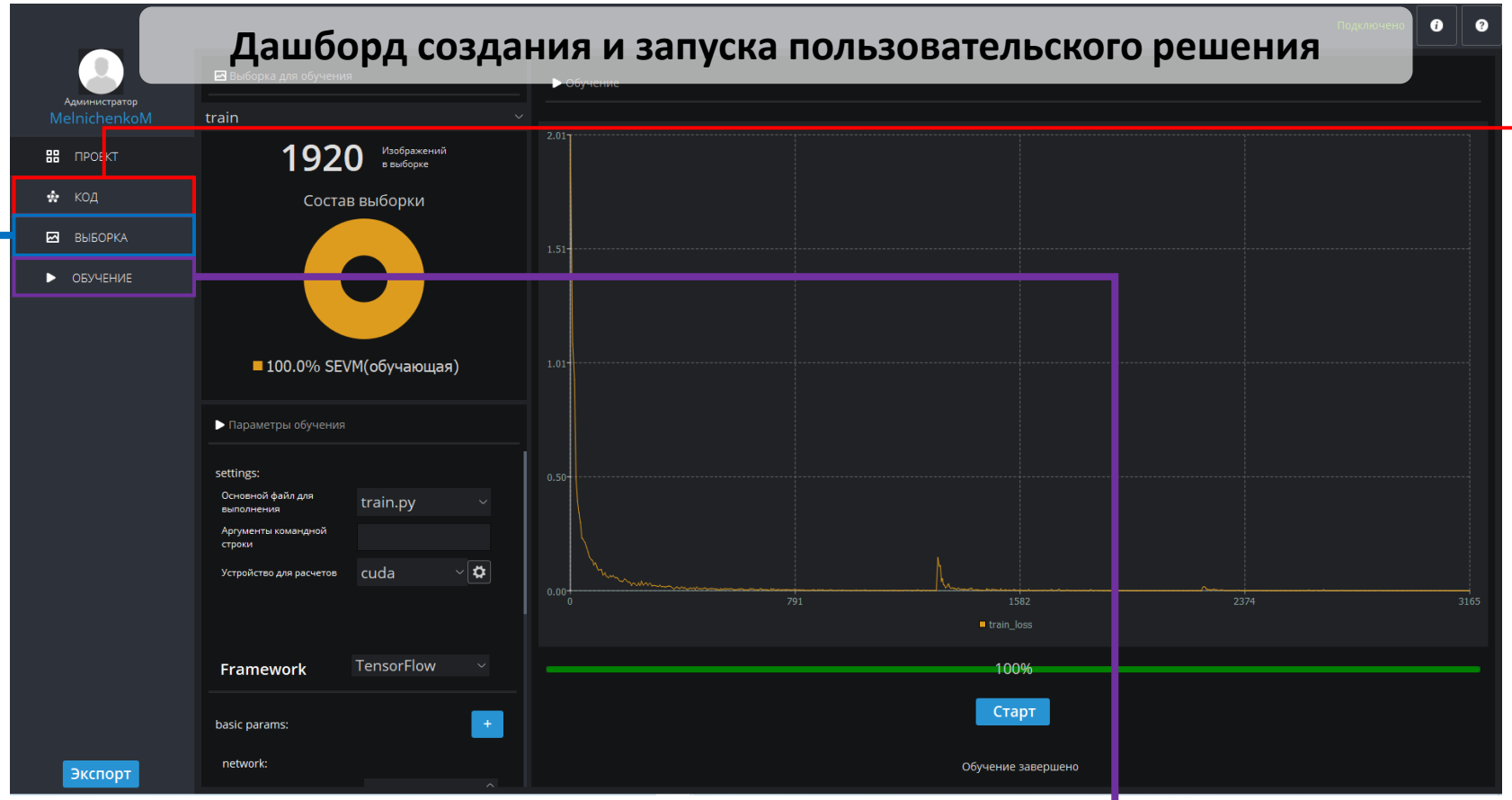
Семинар-конференция «PLAT-2023»

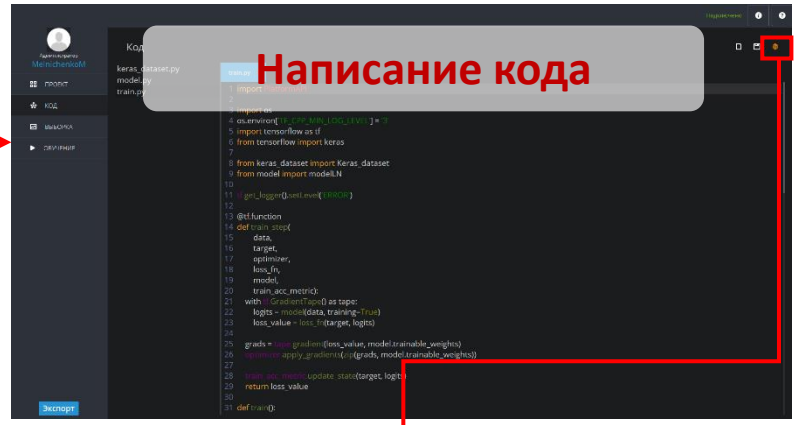
Москва, ГосНИИАС, 15.02.2023

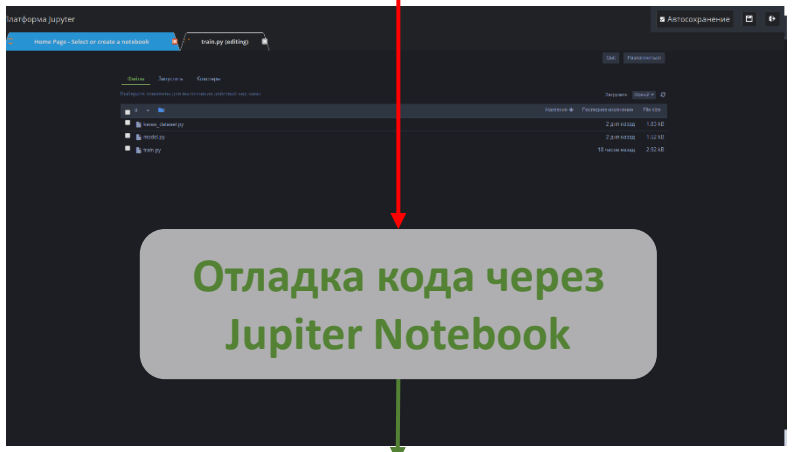
Средства встраивания и интеграции сторонних решений, появившиеся в версии 2.0 «Автомат»

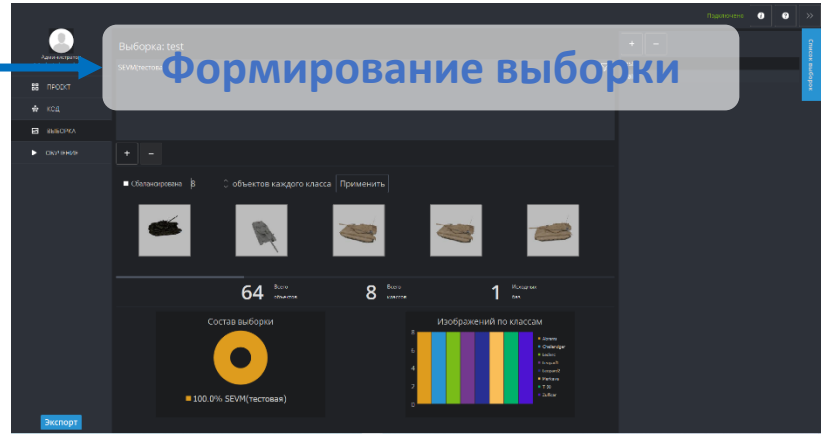
Пользовательские решения.	PlatformAPI

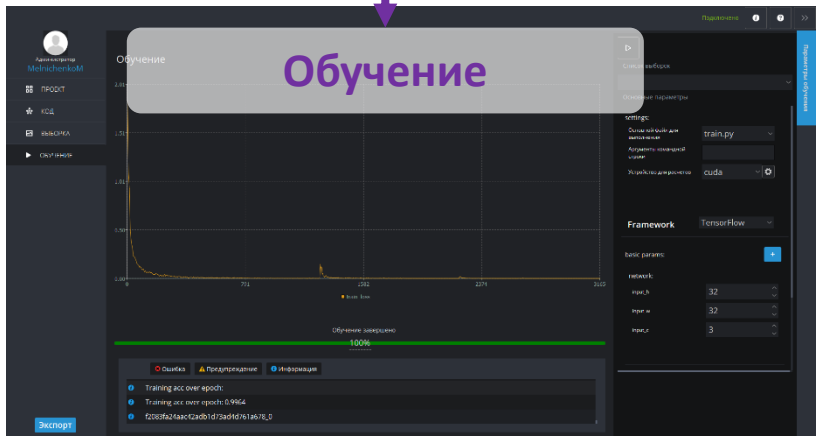
Работа с кодом. Создание пользовательского решения

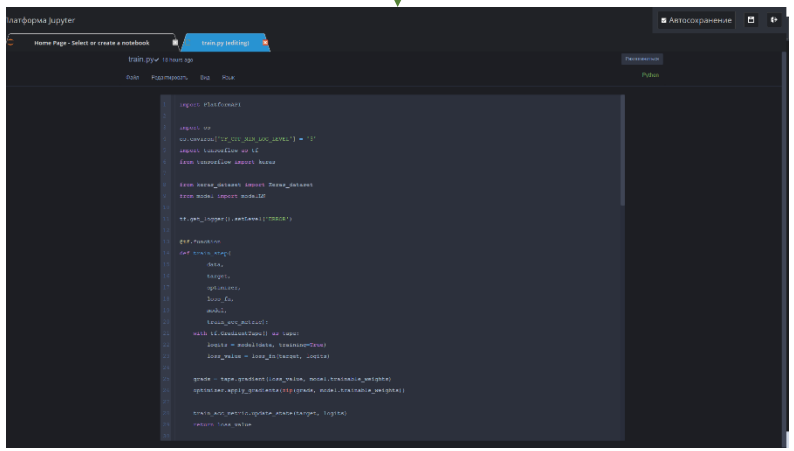




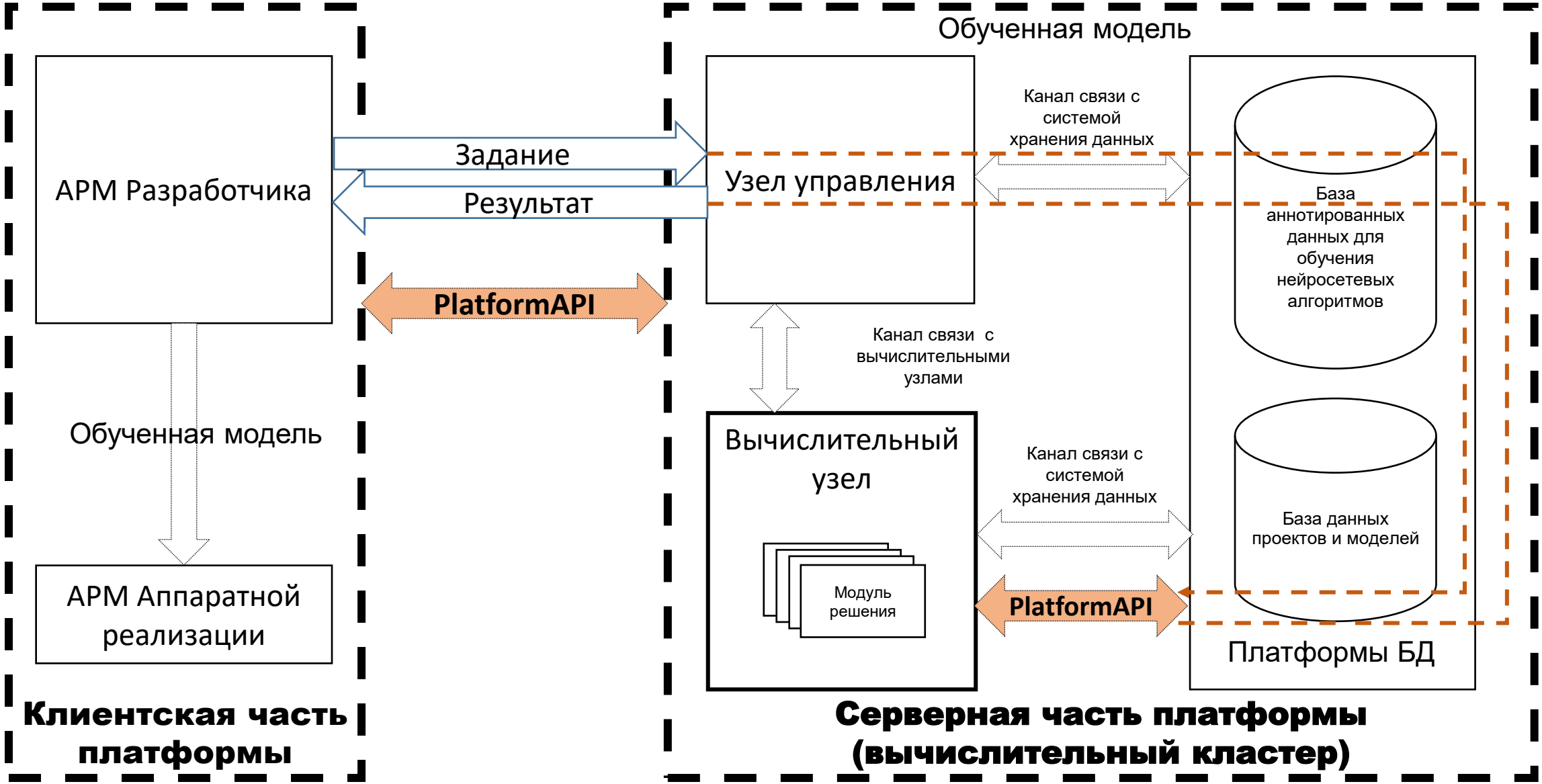








Системные и алгоритмические библиотеки. PlatformAPI



Системные и алгоритмические библиотеки. PlatformAPI

Модули решений взаимодействуют с БД Платформы-ГНС через REST API

Для формирования, передачи и загрузки данных применяется PlatformAPI

В PlatformAPI реализованы следующие базовые классы:

- Dataset – класс базы данных в формате Платформы
- TrainMetrics – класс для записи метрик обучения в формате Платформы
- TestMetrics – класс для записи метрик тестирования в формате Платформы
- MongoAPI – класс, реализующий API доступа к MongoDB
- RedisAPI – класс, реализующий API доступа к Redis
- StorageAPI – класс, реализующий API доступ к хранилищу объектов

В PlatformAPI реализованы следующие основные функции (методы):

- Получение входных параметров

- Получение БД

- Вывод прогресса выполнения

- Вывод информационных сообщений

- Сохранение графиков обучения

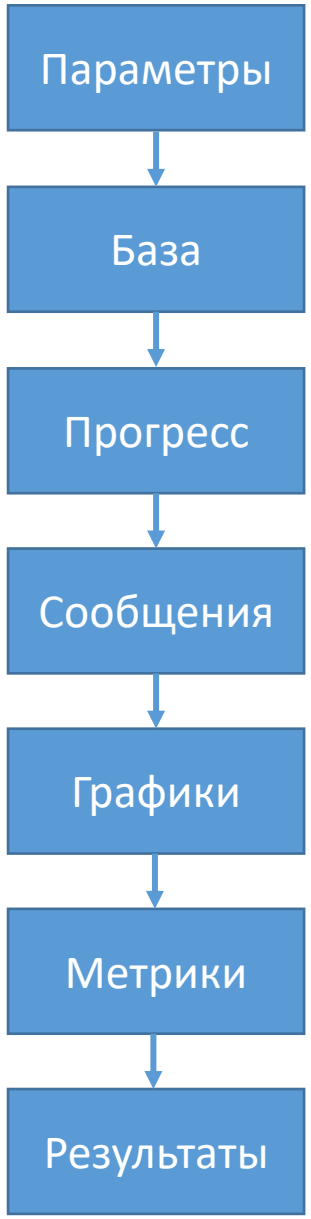
- Сохранение весовых коэффициентов модели

- Сохранение результатов тестирования

- Сохранение метрик тестирования

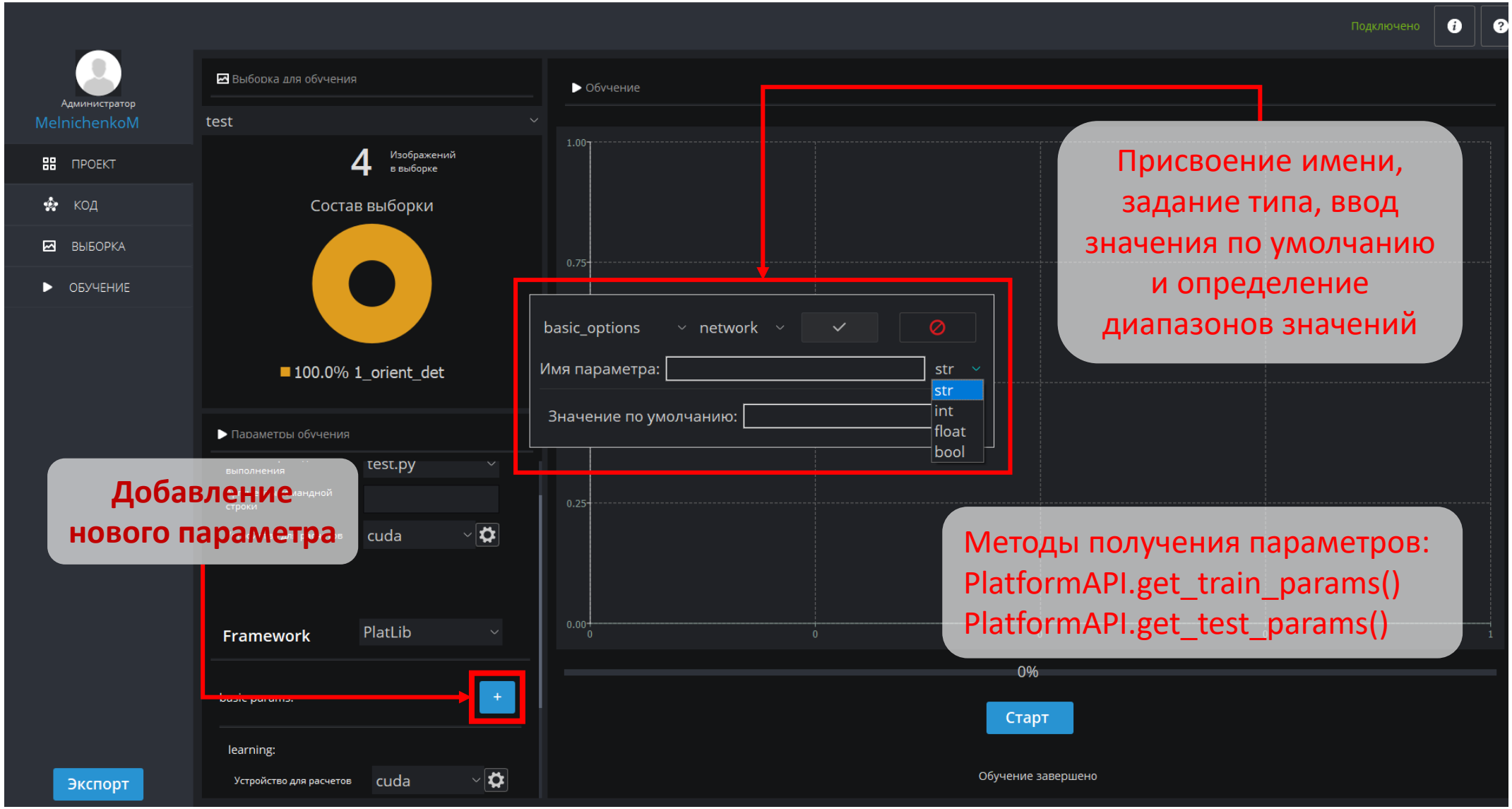
Этапы погружения пользовательского решения в Платформу (методы PlatformAPI)

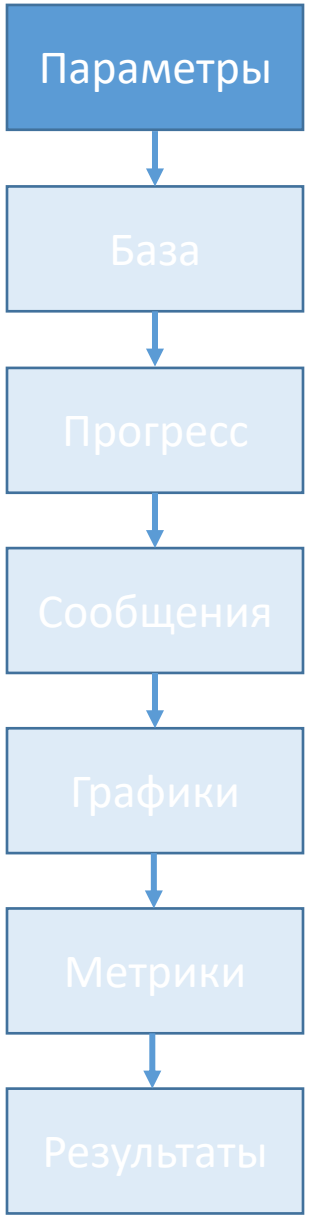
1. Задание входных параметров
2. Обращение к базе данных для работы с выборками
3. Изменение прогресса выполнения эксперимента (обучения / тестирования)
4. Вывод сообщений
5. Построение графиков обучения
6. Формирование метрик тестирования
7. Сохранение весовых коэффициентов модели
8. Сохранение результатов тестирования
9. Перенос в «Пользовательские решения»



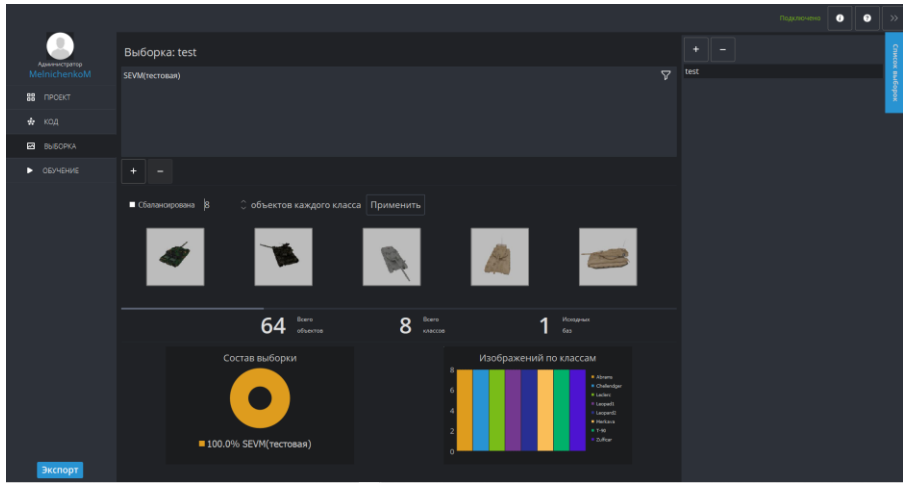
Шаг 1. Задание входных параметров

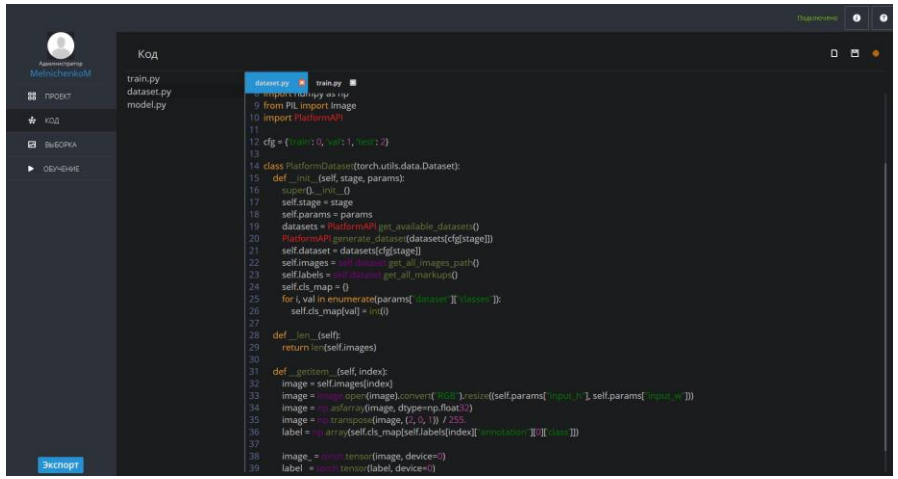
Пользователю необходимо сформировать набор параметров, которые будут влиять на ход выполнения программы. Параметры задаются через интерфейс.





Шаг 2. Обращение к базе данных для работы с выборками





Необходимо получить список доступных выборок:

datasets = PlatformAPI.get_available_datasets()

Подготовить выборку для дальнейшей работы «train»: i = 0, «val»: i = 1, «test»: i = 2

PlatformAPI.generate_dataset(datasets[i])

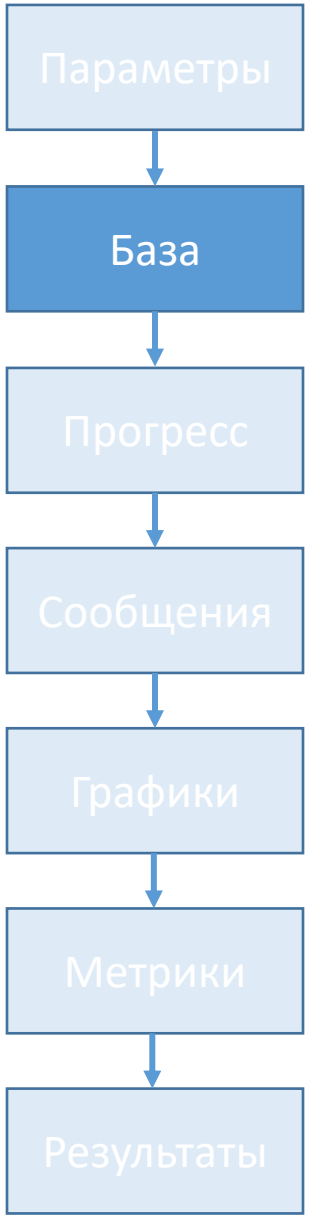
Получить элементы выборки:

```
train_db = datasets[i]

images_path = train_db.get_all_images_path()

add_images_path = train_db.get_all_add_images_path()

markups_path = train_db.get_all_markups ()
```

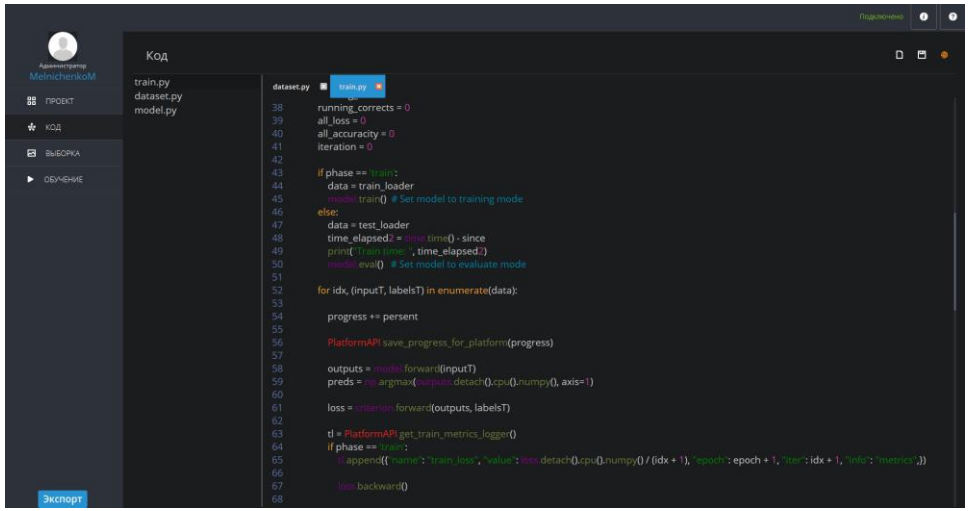
images_path – список путей для основных элементов базы (изображения/файлы/таблицы)

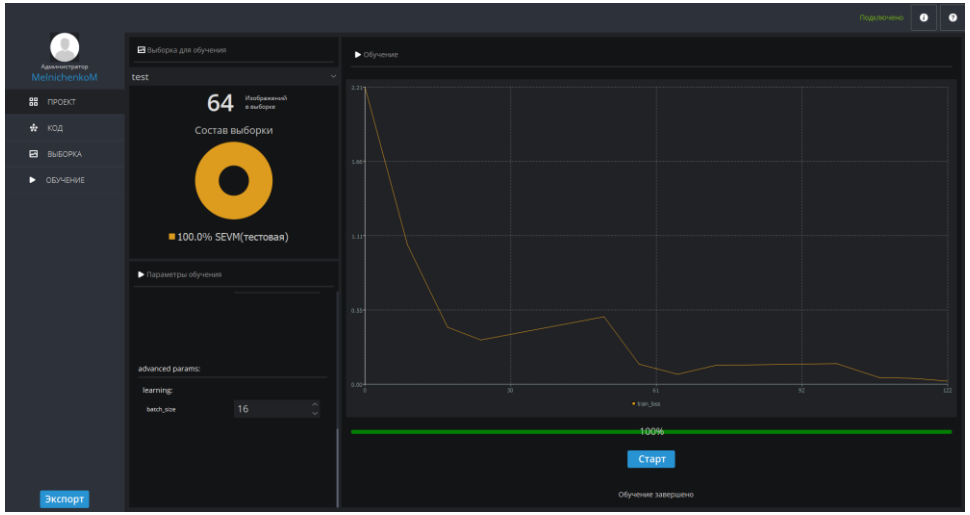
add_images_path – список путей до дополнительный элементов базы (*опционально*)

markups_path – список путей до json фалов, содержащих разметку (*опционально*)



Шаг 3. Индикация хода выполнения задания (обучения /тестирования)

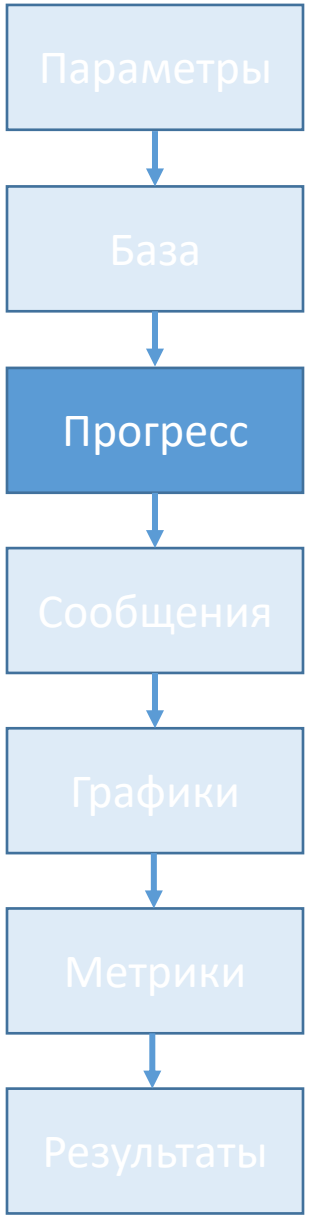




- При запуске программы необходимо прогрессу присвоить стартовое нулевое значение

  PlatformAPI.save_progress_for_platform(float)

- В ходе выполнения программы передавать значение переменной прогресса выполнения от 0 до 100
- **Обновление статуса выполнения задания отображается на прогресс-баре в АРМ Разработчика**

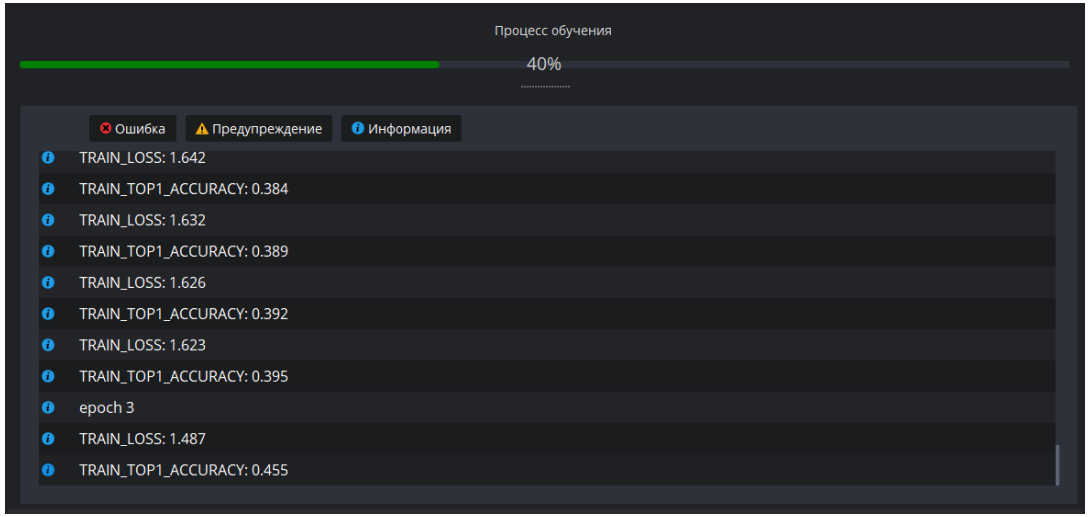


Шаг 4. Вывод сообщений в АРМ Разработчика

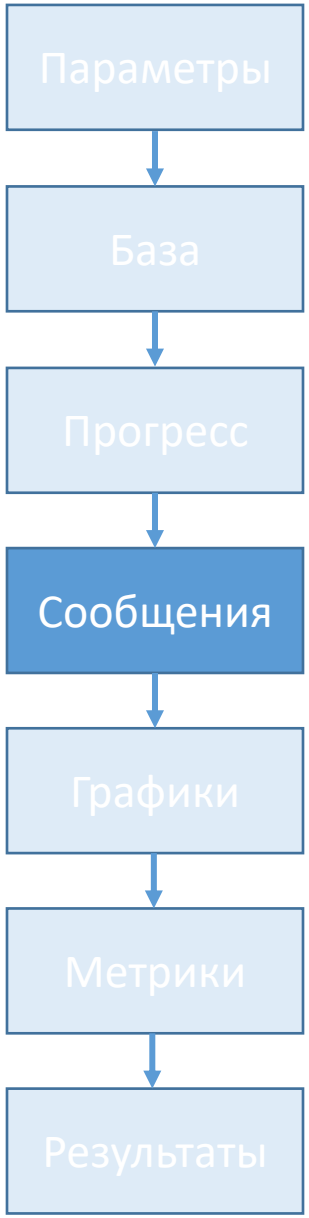
- **В ходе выполнения работы доступен вывод трех типов сообщений: информационных, вывод предупреждения и сообщение об ошибке**
- Необходимо инициализировать объект, передающий информационные сообщения

logger = PlatformAPI.get_logger()

- Обратиться к одному из трех методов и передать строку с сообщение, которая будет доступна в соответствующей вкладке информационного поля




```
mes = ('TRAIN ||Epoch: %d || iter: %d' %
(epoch + 1, i + 1)
logger.info(mes) # информация
logger.error(mes) # ошибка
logger.warning(mes) # предупреждение
```

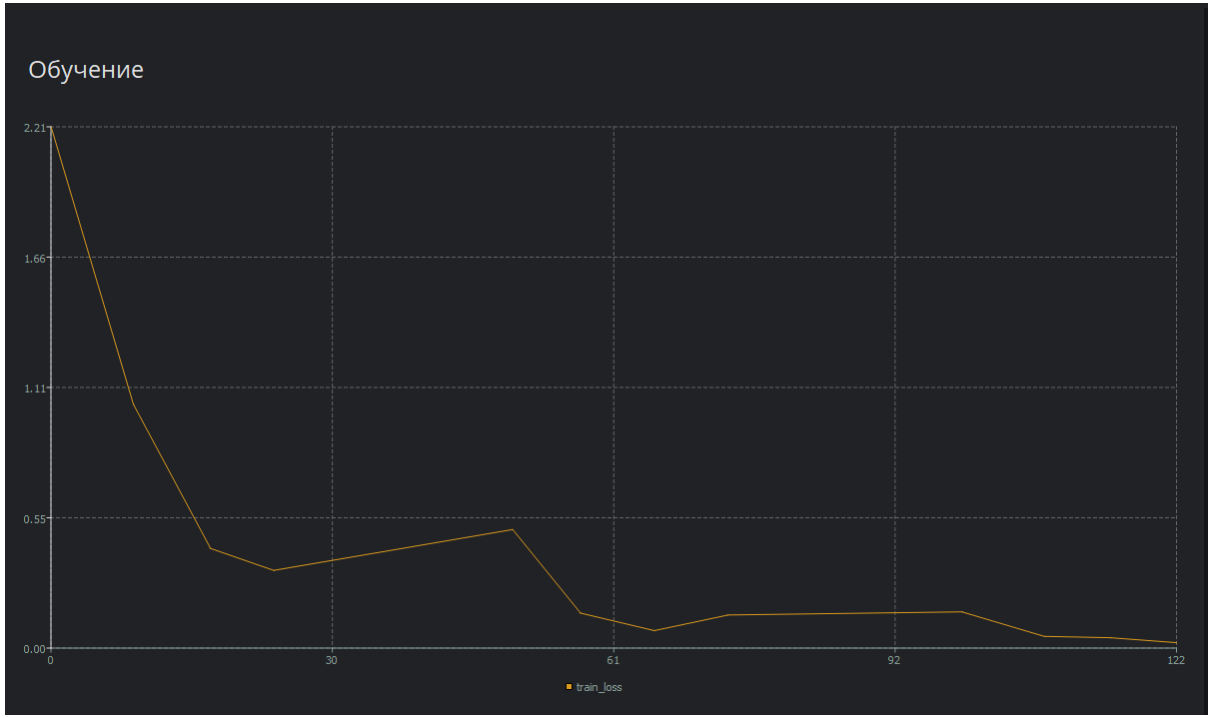


Шаг 5. Построение графиков обучения

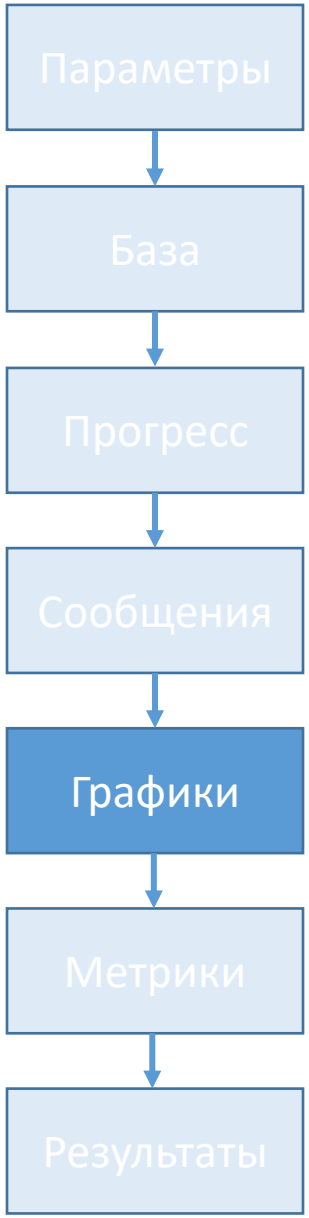
- Необходимо инициализировать объект, передающий информацию о графиках обучения

```
train_logger = PlatformAPI.get_train_graph_logger()
```

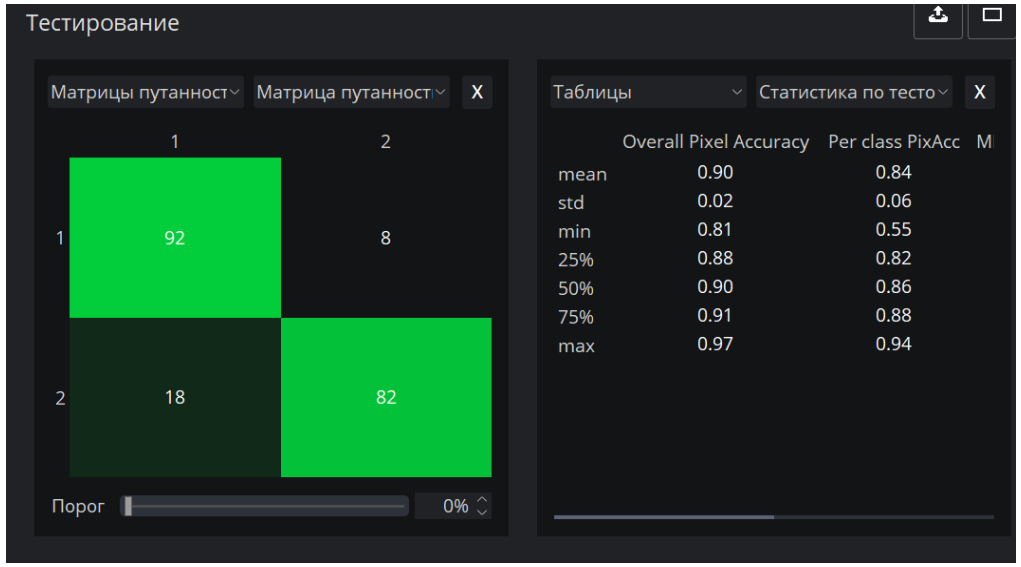
- Для записи точки на графике используется метод log_point



log_point(name, value, epoch, iter, info)
name – имя графика;
value – значение точки;
epoch – номер эпохи;
iter – номер итерации;
info – дополнительная информация для точки на графике.



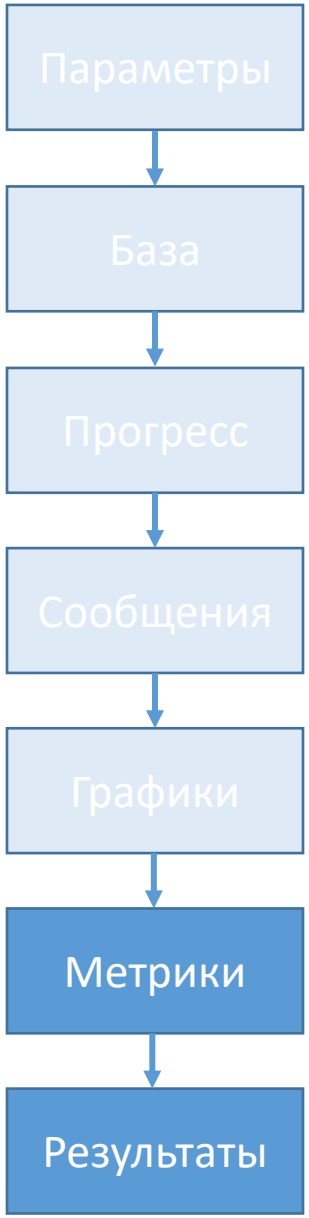
Шаг 6. Формирование метрик тестирования



- Необходимо инициализировать объект, передающий информацию о метриках тестирования

```
test_logger = PlatformAPI.get_test_metrics_logger()
```

- Для сохранения полученных значений метрик в виде таблицы используется класс Table, в виде матрицы - Matrix, матрицы путанице - ConfusionMatrix, гистограммы - Histogram.



Шаг 7. Сохранение весовых коэффициентов модели

Сохранить веса можно в нескольких форматах:

- PyTorch

```
saver = TorchPklSaver._save(model,fpath,input_example)
```

- ONNX

```
saver = T2OnnxSaver._save(model,fpath,input_example)
```

- их комбинация

```
saver = CombSaver._save(model,fpath,input_example)
```

Шаг 8. Сохранение результатов тестирования

Результатом обработки элемента базы является JSON-файл.

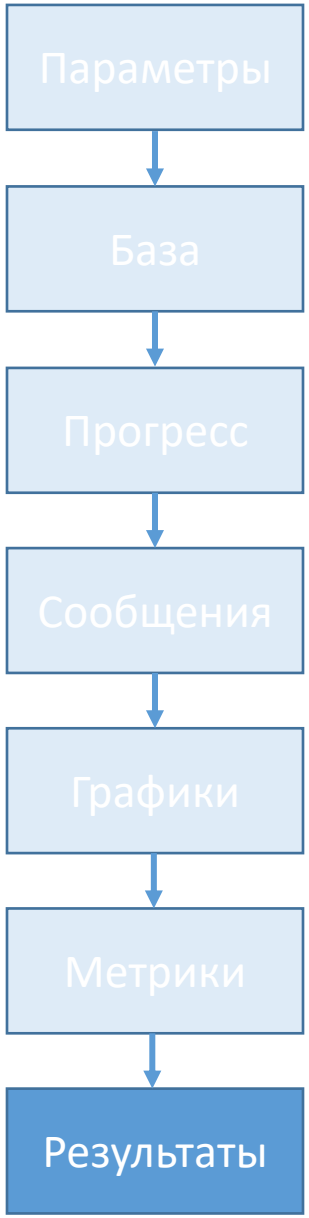
Его заполнение зависит от типа задачи и типа разметки.

```
anno = {
    "version": 1,
    "image": {
        "representation": "",
        "value": ""
    },
    "annotation": []
}
detections_on_image = detections[idx]
for box in detections_on_image:
    score, class_idx, _, restored_coords = box
    class_name = test_set.r_classmap[class_idx]
    conf_score = round(score.item(), 3)
    x1 = int(restored_coords[0])
    y1 = int(restored_coords[1])
    x2 = int(restored_coords[2])
    y2 = int(restored_coords[3])
    anno['annotation'].append(
        {
            "description": "ObjectDetection",
            "class": class_name,
            "confidence": conf_score,
            "shape": {
                "type": "rectangle",
                "exterior": [[x1, y1], [x2, y2]],
                "interior": []
            }
        }
    )
```

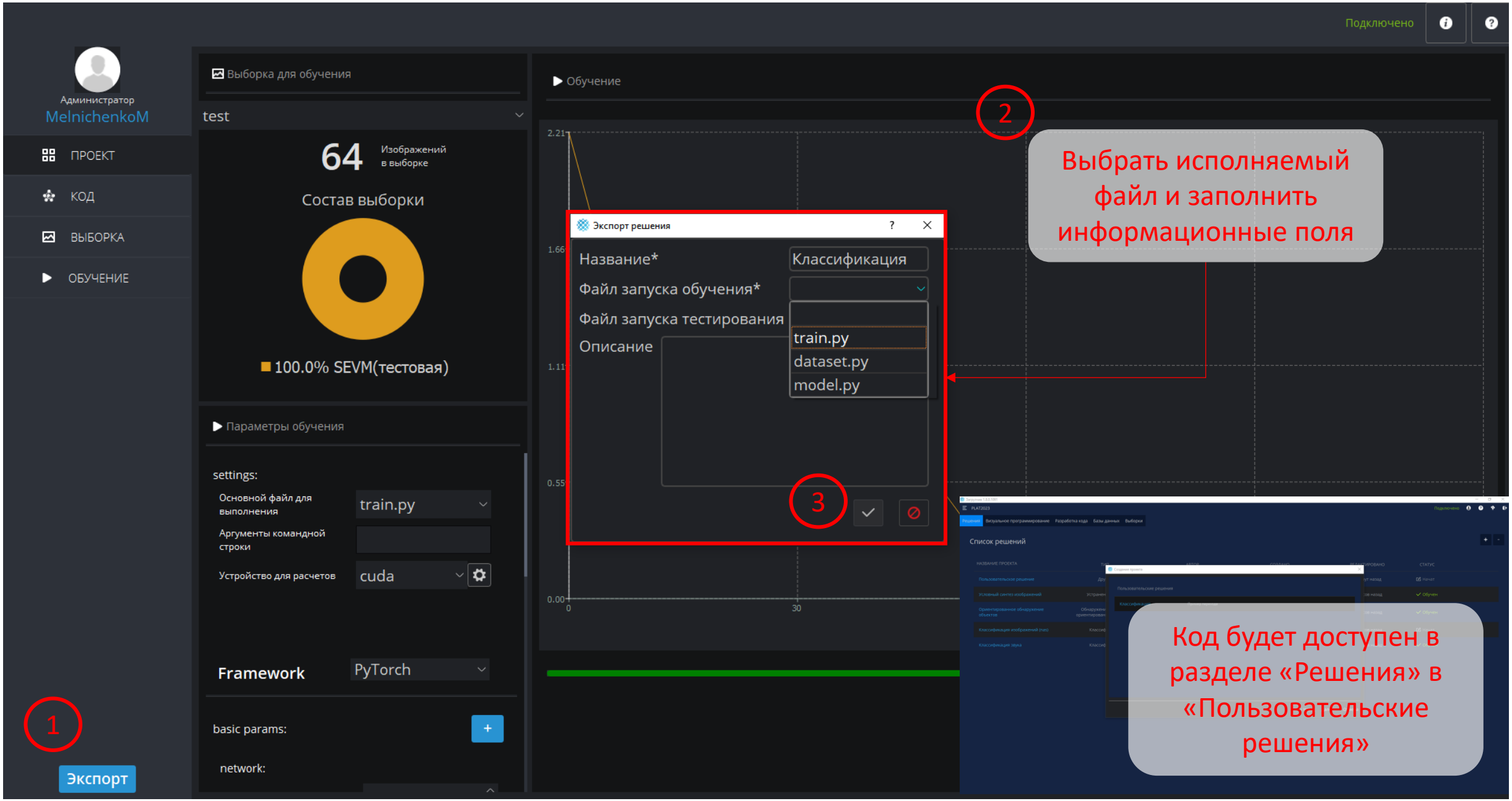
Пользователю необходимо сформировать JSON и передать его с применением метода:

PlatformAPI.get_markup_dict_for_images(json, description)

и указанием типа решаемой задачи (description)



Шаг 9. Экспорт готового пользовательского решения



Отечественная библиотека машинного обучения.

Изменения в версии 2.0

Platlib. Унификация с PyTorch

Фреймворки и платформы глубокого обучения

Классические фреймворки для обучения нейронных сетей: **PyTorch, TensorFlow, Keras, PlatLib** и др.

Самостоя-тельная работа с данными, разметка доступными средствами → Создание архитектуры сети и программирование pipeline на Python, включая аугментации данных → Обучение на серверах CPU/GPU → Готовая модель с весами: ONNX (общий формат), pt/pth (PyTorch), pb/tflite (TensorFlow) → Материал для написания статей либо самостоятельная интеграция с помощью инструментариев производителя

Платформы для обучения нейронных сетей: **Datarobot, Roboflow, Supervisely, Платформа-ГНС**

Работа с данными – разметка, включая интеллектуальные (Платформа, Supervisely) или простейшие (Datarobot, Roboflow) помощники

Зарубежные платформы: → Выбор готового типового решения или написание pipeline на Python (Datarobot, Roboflow, Supervisely) → Обучение на серверах CPU/GPU средствами классических фреймворков PyTorch/TensorFlow (Datarobot, Roboflow, Supervisely) → Готовая модель с весами в собственных форматах для использования в собственной экосистеме → Готовое облачное решение или локальное решение (Datarobot, Roboflow, Supervisely)

Платформа-ГНС: → Выбор готового решения / **визуальное программирование** (элементы есть у Roboflow) / написание pipeline на Python → Обучение на серверах CPU/GPU + **отечественных вычислителях**. → Готовая модель с весами: **ONNX**, bin/json (Платформа) → Готовое встраиваемое решение **с поддержкой отечественных вычислителей** (Платформа)

Системные и алгоритмические библиотеки. Фреймворк PlatLib

Слой интерфейса разработчика PlatAPI (Python)

Слой обвязки функций C++ на Python (pybind)

Верхний алгоритмический слой (C++)
Содержит также PlatAPI в реализации на C++

Распределенное обучение
MPI | NCCL, RCCL

Нижний алгоритмический слой (C++)
Включает высокопроизводительные вычисления под разные аппаратные платформы

NVIDIA (CUDA, CUDNN)

AMD (HIP, MIOPEN)

Elbrus

x86-64 / Elbrus

Особенности PlatLib

- Поддержка параллельности выполнения операций в виде распределенного обучения через MPI, в перспективе NCCL, RCCL
- Поддержка высокоскоростных вычислений на вычислителях производства NVIDIA, AMD, Elbrus, Elbrus+Neuromatrix
- Поддержка работы на процессорах x86-64, Elbrus
- Создание динамического графа при осуществлении операций с тензорами
- Поддержка формата сохранения и загрузки моделей в формате Платформы

MPI

NVIDIA

GPU AMD

elbrus

intel

CPU AMD

elbrus

A

Работа с PlatLib. **Унификация программного интерфейса с PyTorch**

Разработка нейронных сетей с PlatLib:

- Унификация программного интерфейса с PyTorch
- Не требуется дополнительное изучение при переходе с PyTorch
- Поддержка работы в Платформе

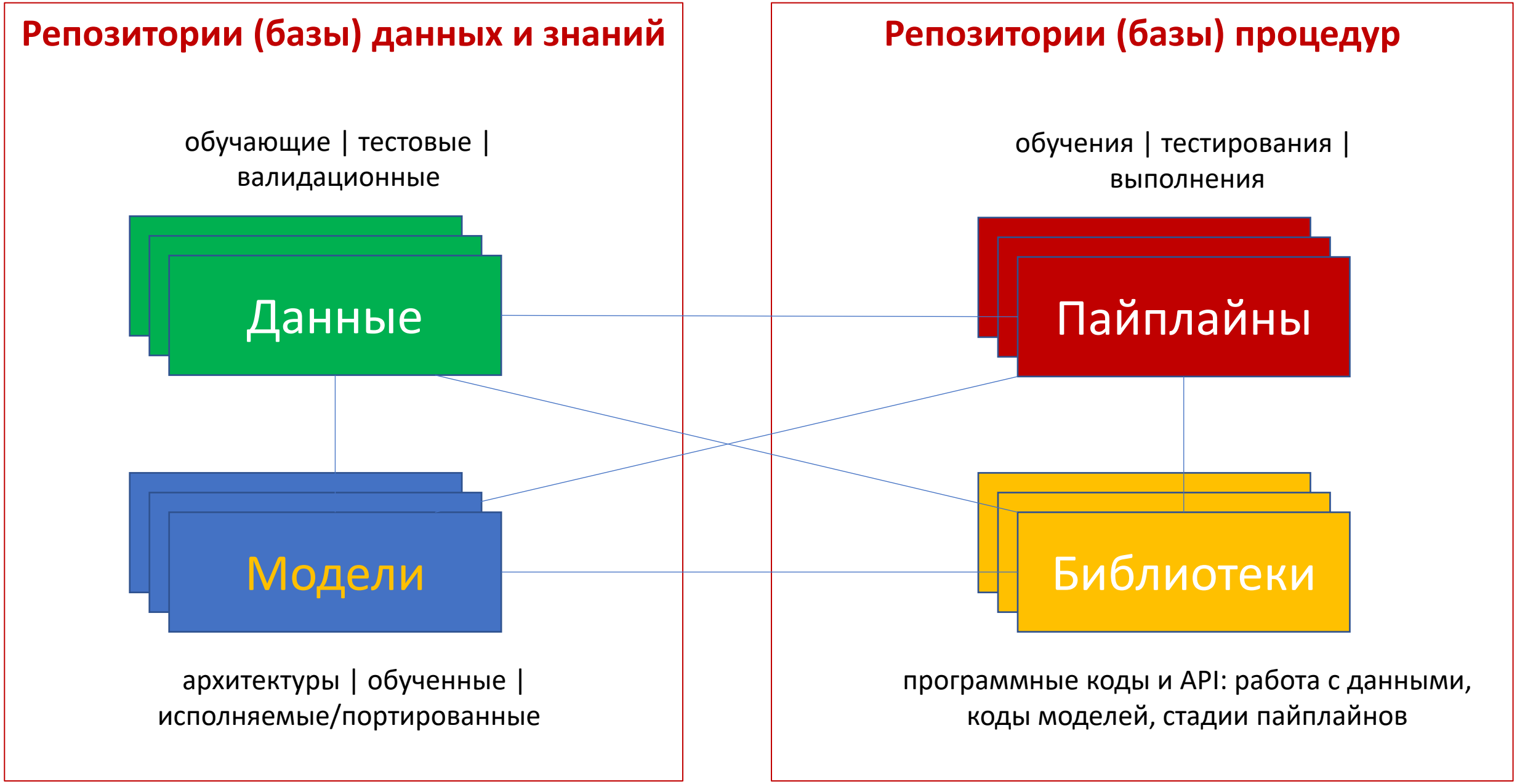
```
class ResNet(plat.nn.Module):
    def __init__(self, in_channels, num_cls, block, num_blocks):
        super(ResNet, self).__init__()
        self.in_planes = 64
        self.num_cls = num_cls
        self.conv1 = plat.nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = plat.nn.BatchNorm2d(64)
        self.relu1 = plat.nn.ReLU()
        self.maxpool = plat.nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.classifer = plat.nn.Linear(512 * block.expansion, num_cls)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return plat.nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu1(self.bn1(self.conv1(x)))
        out = self.maxpool(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
```

Библиотека классов для создания решений

Работа с пайплайнами

Абстрактное типовое решение

Обобщенная архитектура репозиториев экосистемы машинного обучения



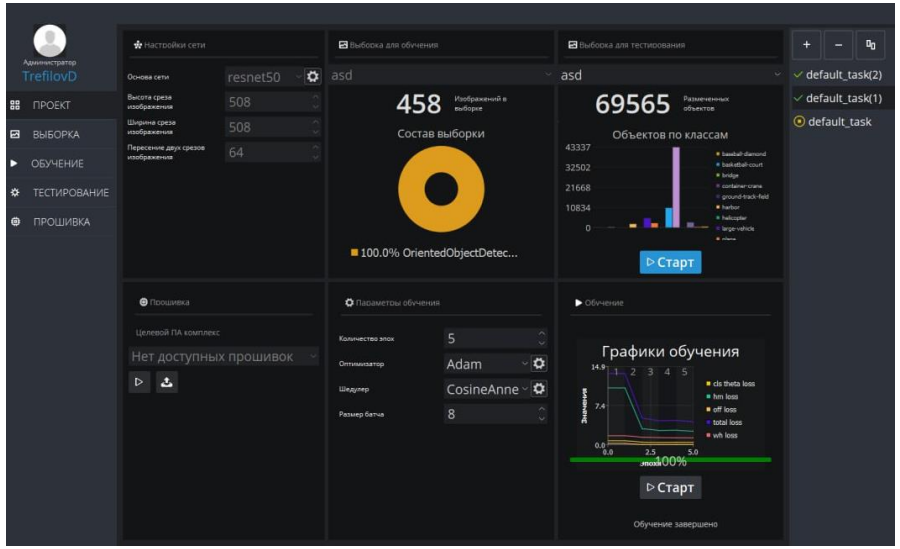
Платформа-ГНС как интегрированная среда / конструктор программных приложений

	Системный уровень	Прикладной уровень
	Репозитории (БД)	Прикладное ПО
Инструменты	Пользователи Ресурсы Проекты Пакеты	**Прикладные пакеты** (типовые решения, решения) — GUI Решений, Решения **Пользовательские проекты** (темы, проекты, задачи, запуски) — GUI Решений, Решения **Программные комплексы** (обучение, тестирование, применение) — GUI комплекса, Решения
Решения ИИ	Данные Модели Пайплайны Библиотеки	

Идея экосистемы PLAT: на системном фундаменте Платформы можно построить целый город

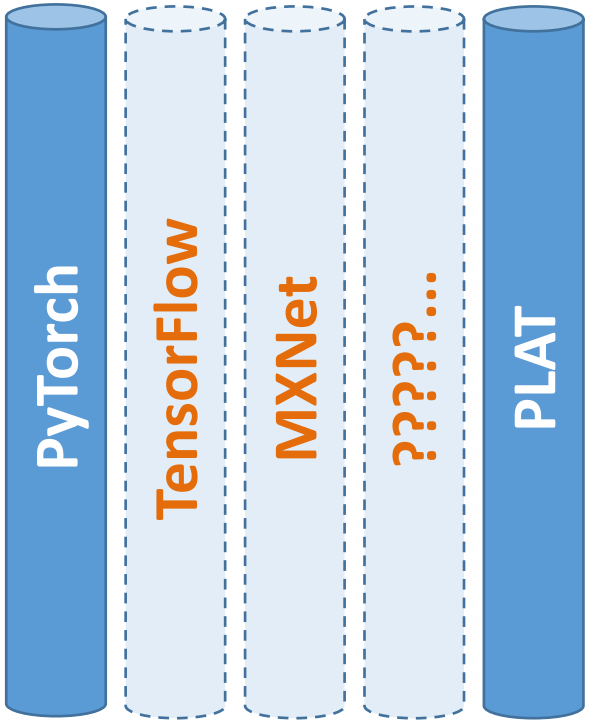
Интегрированная программная платформа – частный случай **программного комплекса**, который включает как **прикладные пакеты**, так и **пользовательские проекты**

Уровень работы с пайплайнами. **Абстрактное типовое решение**



Ограничения типовых решений
Платформы («фастфуд»):

- Реализованы только простые пайплайны обучения сетей (from scratch)
- Отсутствует возможность конвейерного запуска нескольких сетей в одном решении
- Сложная реализация дообучения (неявно через checkpoint) и отсутствие возможности дообучения на новые классы в старом проекте
- **Необходим удобный инструмент формирования сложных пайплайнов**



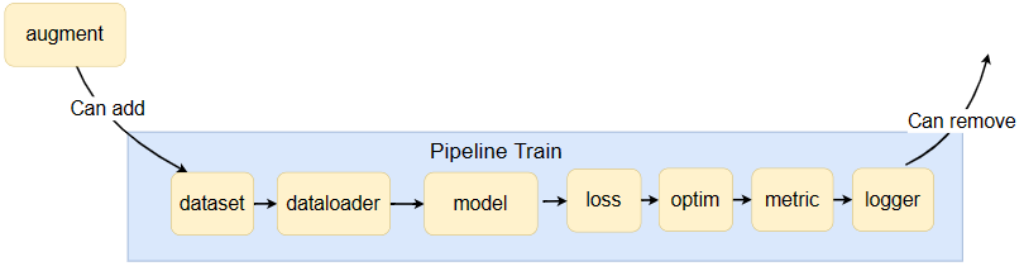
Проблема перехода между фреймворками:

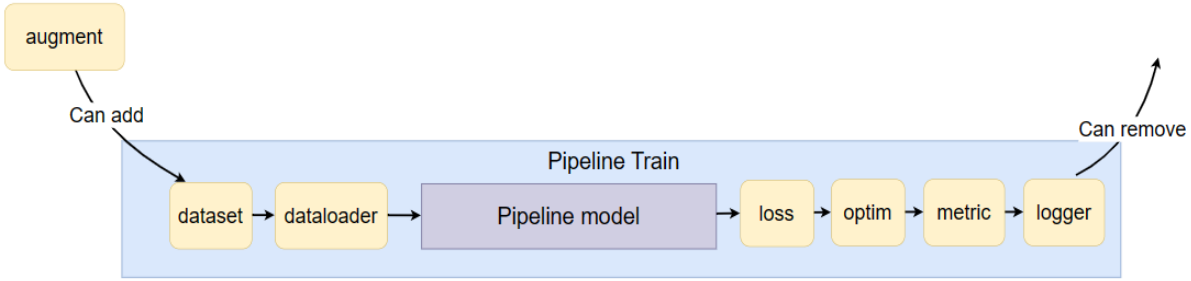
- PlatLib и PyTorch унифицированы по интерфейсам (возможен автоматический «перевод» решений)
- Другие фреймворки имеют различные интерфейсы
- **Необходим более «абстрактный» уровень описания решения «над кодом»**

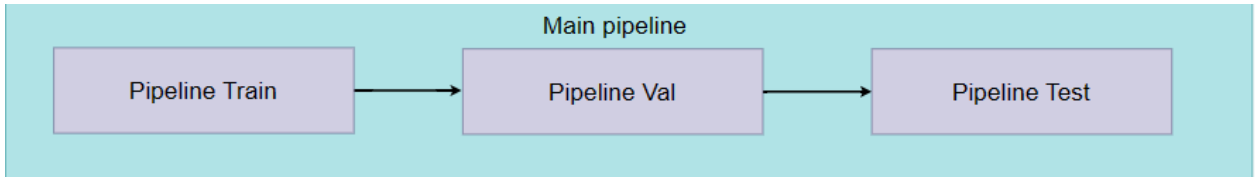
Мы предлагаем решать эти проблемы с помощью инструмента **«Абстрактное типовое решение»**

Преимущества абстрактного типового решения

- **Унифицированный способ описания пайплайнов машинного обучения с помощью конфигурационного файла формата YAML**
- Легко модифицировать пайплайн: добавлять, удалять и изменять элементы внутри пайплайна
- Сложные схемы можно представить иерархическими блоками (подграфы, стадии и т.д.)
- Отдельные элементы пайплайнов и сами пайплайны можно использовать многократно, объединять и т.д.

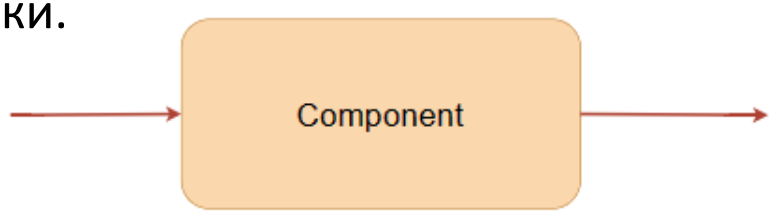




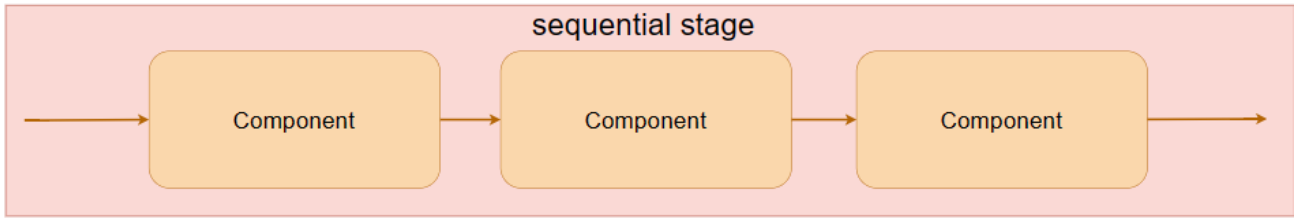


Основные сущности и логика работы

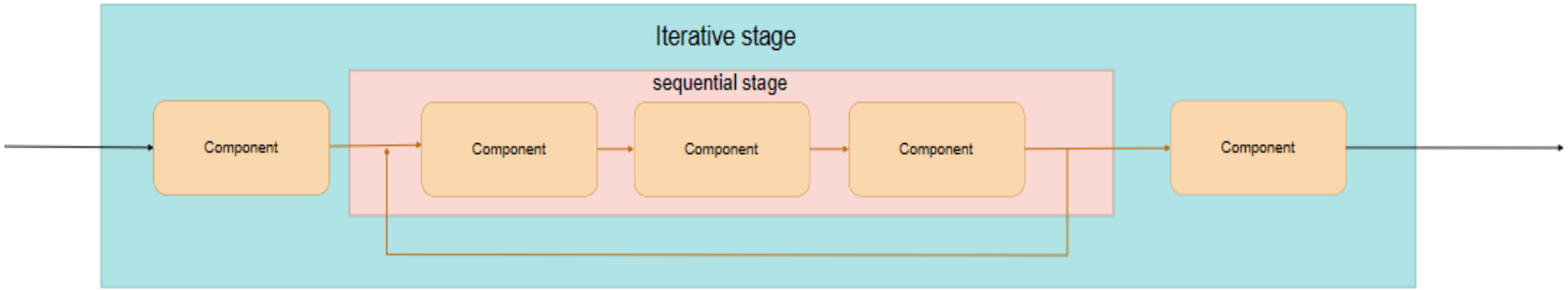
- **Компонента** - черный ящик некоторого процесса. Она позволяет использовать класс/функцию (код или пайплайн) как звено цепочки.



- **Последовательная стадия** - белый ящик некоторого процесса. Она позволяет структурировать компоненты пайплайна, объединять их в сборные компоненты.



- **Итеративная стадия** - прозрачный цикл процесса. Она позволяет использовать компоненты в цикле.



Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Пример конфигурационного файла (формат YAML)

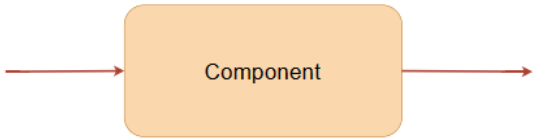
Компонента dataset

Компонента model

Компонента crit

Компонента optimizer

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```



- Компонента – код или пайплайн
- Компоненты имеют входы и выходы
- Через входы и выходы передаются данные
- Компоненты могут иметь состояние

Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Шаг 2. Последовательная стадия
Имеется дополнительный атрибут «process», описывающий порядок вызова компонент и их методов.

Пример конфигурационного файла (формат YAML)

Компонента dataset

Компонента model

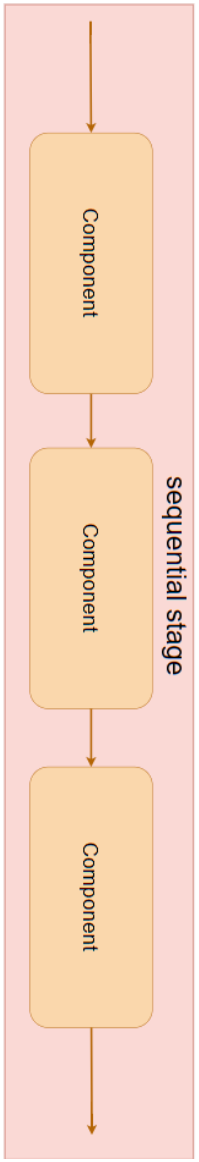
Компонента crit

Компонента optimizer

process

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```

Можно задать правила или различные варианты выполнения (какие компоненты при каких условиях выполняются)



Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Шаг 2. Последовательная стадия
Имеется дополнительный атрибут «process», описывающий порядок вызова компонент и их методов.

Шаг 3. Итеративная стадия.
Имеется атрибут «iterator», который позволяет добавить в процесс итерируемую компоненту. Атрибут «process» становится составным.

Пример конфигурационного файла (формат YAML)

Компонента dataset

Iterator

Компонента model

Компонента crit

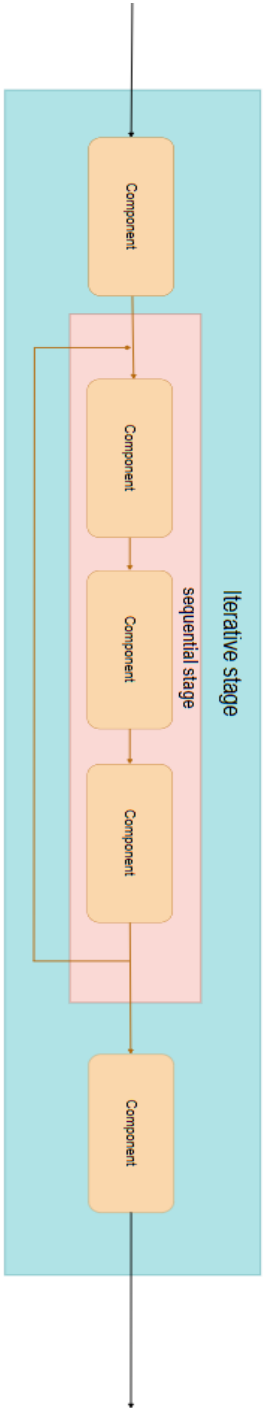
Компонента optimizer

process

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```

Итеративная стадия Train_stage

Итерации могут быть иерархическими (циклы в цикле)



Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Шаг 2. Последовательная стадия
Имеется дополнительный атрибут «process», описывающий порядок вызова компонент и их методов.

Шаг 3. Итеративная стадия.
Имеется атрибут «iterator», который позволяет добавить в процесс итерируемую компоненту. Атрибут «process» становится составным.

Пример конфигурационного файла (формат YAML)

Компонента dataset

Iterator

Компонента model

Компонента crit

Компонента optimizer

process

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```

Итеративная стадия Train_stage

Специальные символы (введены нами):

- "**#**" – символ-ссылки на **компоненты**, описанные в других файлах конфигураций, либо виджеты и т. п.
- "**@**" – символ-ссылки на **компоненту** в данном конфигурационном файле
- "**$**" – символ-ссылки на **данные** в контексте данной стадии, ссылка на один из выходов ранее использованных компонент
- Имя, начинающееся с **числового** символа, означает позиционный аргумент
- "**?**" - символ вызова справки

Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Шаг 2. Последовательная стадия
Имеется дополнительный атрибут «process», описывающий порядок вызова компонент и их методов.

Шаг 3. Итеративная стадия.
Имеется атрибут «iterator», который позволяет добавить в процесс итерируемую компоненту. Атрибут «process» становится составным.

Пример конфигурационного файла (формат YAML)

Компонента dataset

Iterator

Компонента model

Компонента crit

Компонента optimizer

process

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```

Итеративная стадия Train_stage

Специальные символы (введены нами):

- "**#**" – символ-ссылки на **компоненты**, описанные в других файлах конфигураций, либо виджеты и т. п.
- "**@**" – символ-ссылки на **компоненту** в данном конфигурационном файле
- "**$**" – символ-ссылки на **данные** в контексте данной стадии, ссылка на один из выходов ранее использованных компонент
- Имя, начинающееся с **числового** символа, означает позиционный аргумент
- "**?**" - символ вызова справки

Компонента – код или пайплайн

Два вида ссылок на компоненты различаются как раз для того, чтобы пайплайны можно было хранить отдельно от кодов (и других ранее созданных пайцплайнов).

Это позволяет пайплайны многократно иерархически применять к разным процедурам

Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Шаг 2. Последовательная стадия
Имеется дополнительный атрибут «process», описывающий порядок вызова компонент и их методов.

Шаг 3. Итеративная стадия.
Имеется атрибут «iterator», который позволяет добавить в процесс итерируемую компоненту. Атрибут «process» становится составным.

Пример конфигурационного файла (формат YAML)

Компонента dataset

Iterator

Компонента model

Компонента crit

Компонента optimizer

process

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```

Итеративная стадия Train_stage

Специальные символы (введены нами):

- "**#**" – символ-ссылки на **компоненты**, описанные в других файлах конфигураций, либо виджеты и т. п.
- "**@**" – символ-ссылки на **компоненту** в данном конфигурационном файле
- "**$**" – символ-ссылки на **данные** в контексте данной стадии, ссылка на один из выходов ранее использованных компонент
- Имя, начинающееся с **числового** символа, означает позиционный аргумент
- "**?**" - символ вызова справки

У любой компоненты по умолчанию есть вход(ы) и выход(ы), через которые передаются данные.

Если нужно получить какие-то данные изнутри компоненты (в обход инкапсуляции входов и выходов), то используется такая ссылка.

Пример описания

Шаг 1. Компоненты
Основой конфигурационного файла являются компоненты, необходимые для запуска стадии (dataset, model, crit, optimizer…)

Шаг 2. Последовательная стадия
Имеется дополнительный атрибут «process», описывающий порядок вызова компонент и их методов.

Шаг 3. Итеративная стадия.
Имеется атрибут «iterator», который позволяет добавить в процесс итерируемую компоненту. Атрибут «process» становится составным.

Пример конфигурационного файла (формат YAML)

Компонента dataset

Iterator

Компонента model

Компонента crit

Компонента optimizer

process

```yaml
Train_stage:
    dataset:
        module: "torchvision.datasets"
        name: CIFAR10
        root: "root"
        download: True
    iterator:
        name: DataLoader
        module: torch.utils.data.dataloader
        num_workers: 0
        batch_size: 8
        dataset: "@dataset"
        outputs: ["images","labels"]
    model:
        name: resnet18
        module: torchvision.models
        inputs: ["images"]
        outputs: ["model_out"]
    crit:
        name: CrossEntropyLoss
        module: torch.nn
        inputs: ["model_out", "labels"]
        outputs: ["loss"]
    optimizer:
        name: AdamW
        module: torch.optim
        "1": "@model.parameters()"
        lr: 0.001
    process:
        set_iter:
            - "@model.train"
            - "@model.zero_grad"
        process_iter:
            - "@model"
            - "@crit"
            - "$loss.backward"
            - "@optimizer.step"
            - "@optimizer.zero_grad"
```

Итеративная стадия Train_stage

Специальные символы (введены нами):

- "**#**" – символ-ссылки на компоненты, описанные в других файлах конфигураций, либо виджеты и т. п.
- "**@**" – символ-ссылки на компоненту в данном конфигурационном файле
- "**$**" – символ артефакта процесса выполнения в контексте данной стадии, ссылка на один из выходов ранее использованных компонент
- Имя, начинающееся с **числового** символа, означает позиционный аргумент
- "**?**" - символ вызова справки

Справка по запросу выводит необходимые для работы компоненты аргументы в виде таблицы

Object (*args and **kwargs):

empty	default	default value
self	lr	0.001
params	betas	(0.9, 0.999)
	eps	1e-08
	weight_decay	0
	amsgrad	False
	foreach	None
	maximize	False
	capturable	False
	differentiable	False
	fused	False

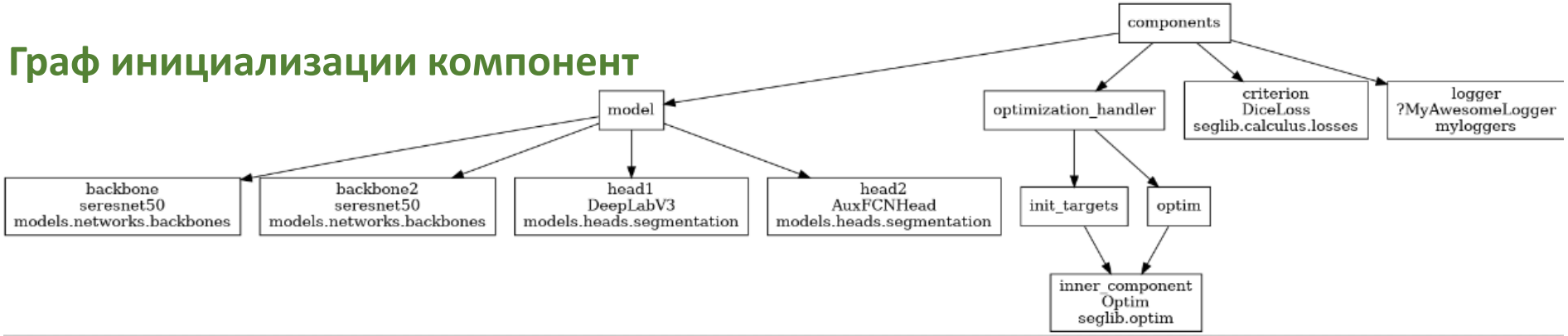
Валидация, визуализация и запуск пайплайнов

До начала запуска все компоненты проходят валидацию:

1. Проверяется возможность **инициализации компонент и связи между компонентами**, определяется порядок их создания

При правильной инициализации всех компонент и последовательности действий внутри процесса можно реализовать их визуализацию в виде графа. На нем показаны связи между компонентами по входных и выходных данным и последовательность выполнения действий внутри процесса.

Граф инициализации компонент



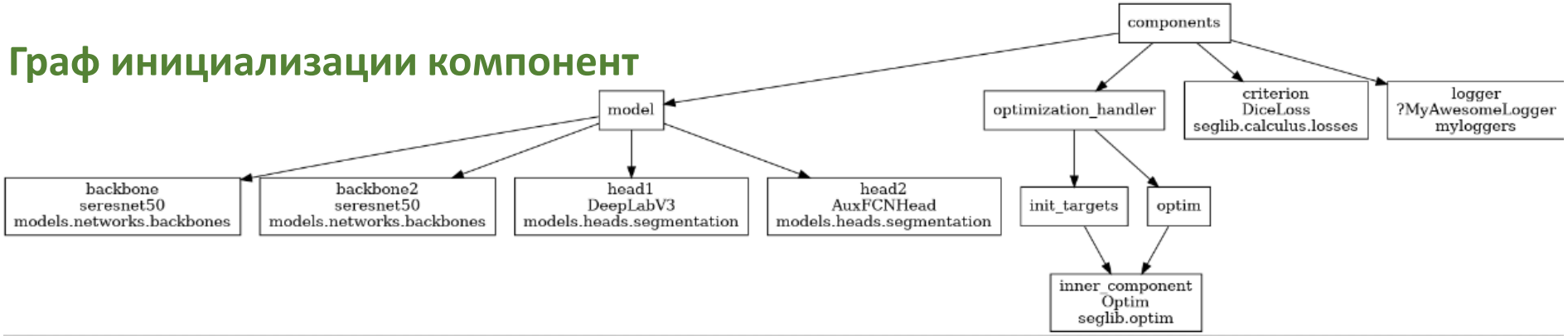
Валидация, визуализация и запуск пайплайнов

До начала запуска все компоненты проходят валидацию:

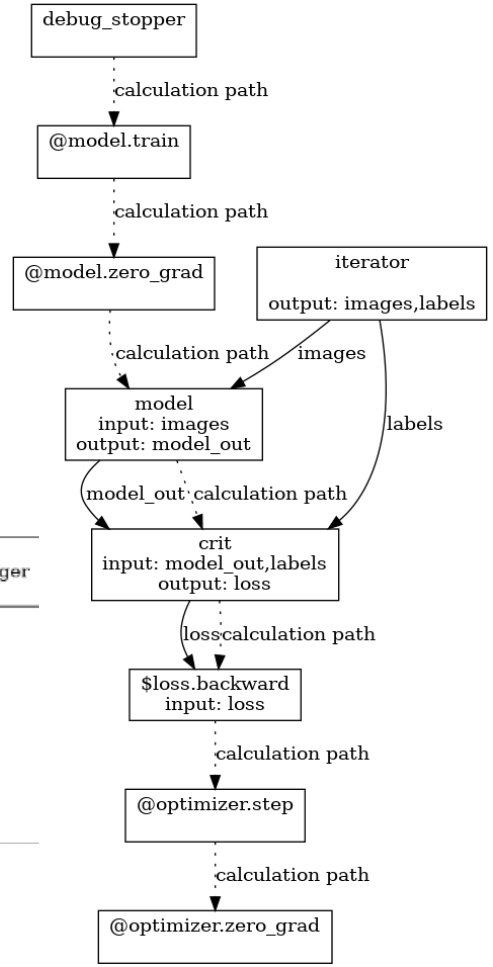
1. Проверяется возможность **инициализации компонент и связи между компонентами**, определяется порядок их создания
2. Далее происходит проверка **графа выполнения процесса (пайплайна)**, а именно проверяется возможность вызова всех компонент и их методов, проверяется порядок вызова по входным и выходным данным.

При правильной инициализации всех компонент и последовательности действий внутри процесса можно реализовать их визуализацию в виде графа. На нем показаны связи между компонентами по входных и выходных данным и последовательность выполнения действий внутри процесса.

Граф инициализации компонент



Граф выполнения процесса (пайплайн)



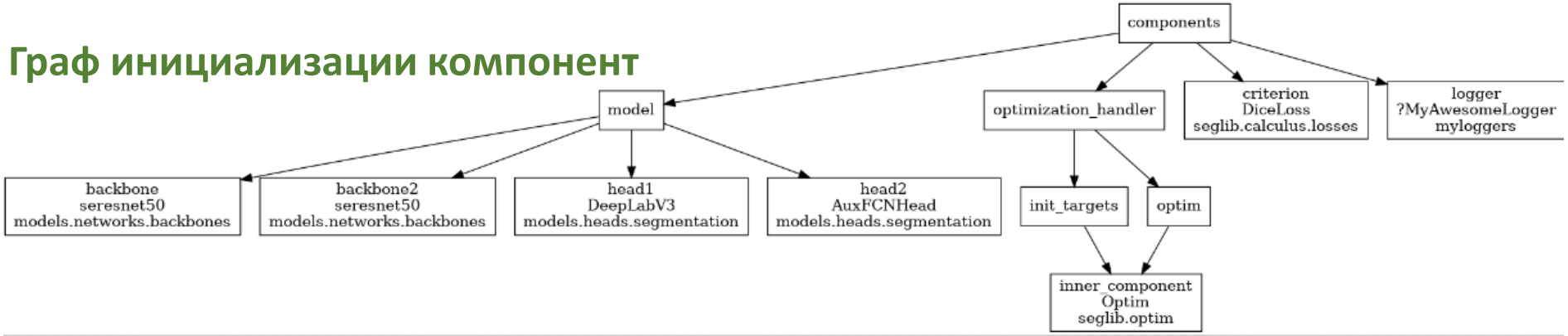
Валидация, визуализация и запуск пайплайнов

До начала запуска все компоненты проходят валидацию:

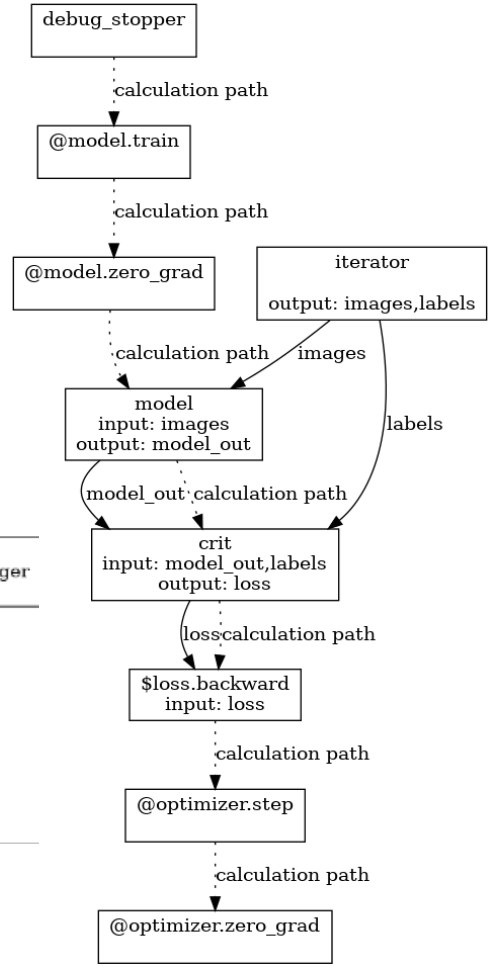
1. Проверяется возможность **инициализации компонент и связи между компонентами**, определяется порядок их создания
2. Далее происходит проверка **графа выполнения процесса (пайплайна)**, а именно проверяется возможность вызова всех компонент и их методов, проверяется порядок вызова по входным и выходным данным.

При правильной инициализации всех компонент и последовательности действий внутри процесса можно реализовать их визуализацию в виде графа. На нем показаны связи между компонентами по входных и выходных данным и последовательность выполнения действий внутри процесса.

Граф инициализации компонент



Граф выполнения процесса (пайплайн)



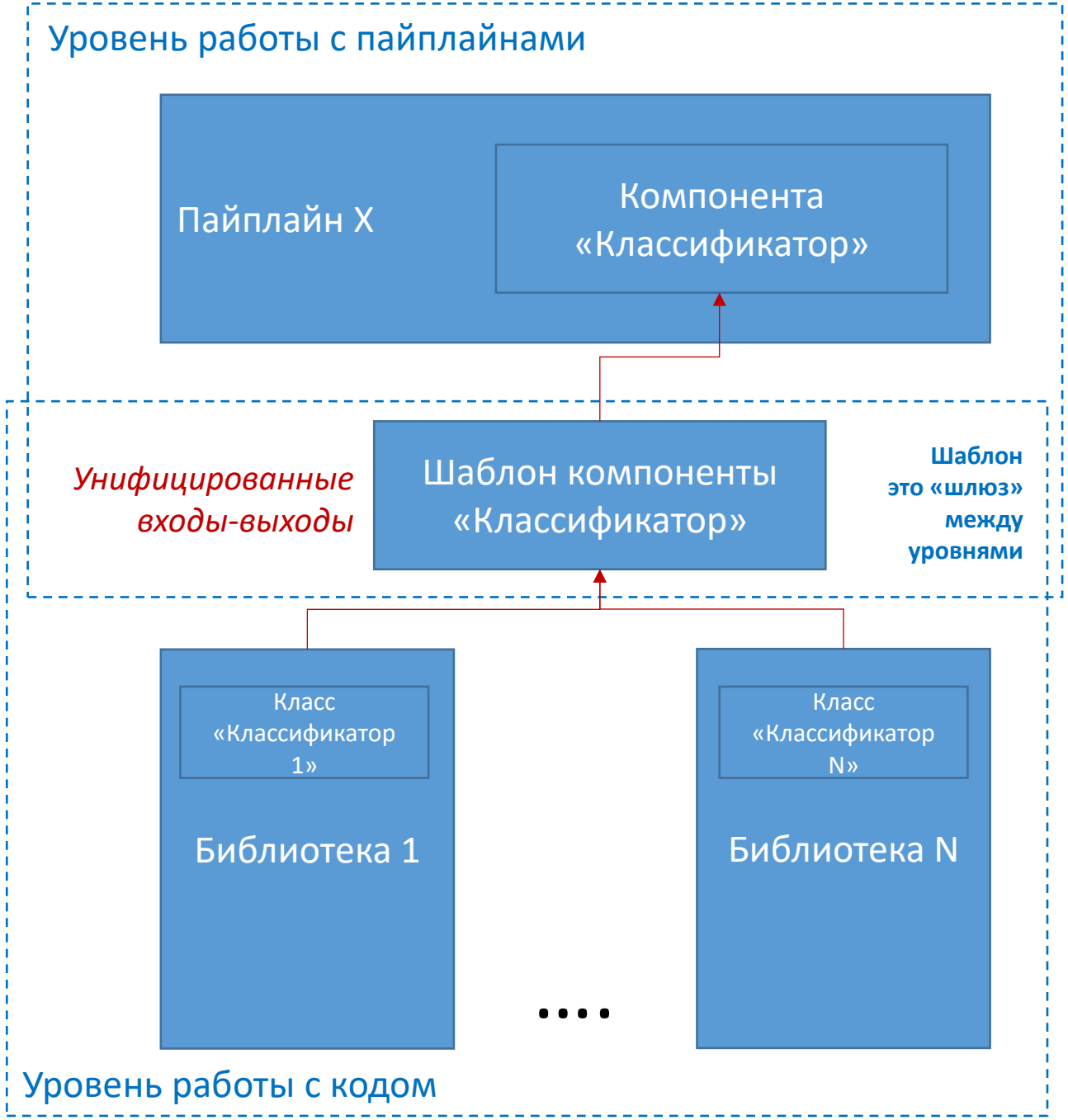
После валидации производится запуск:

3. Инициализация компонент и пайплайнов
4. Запуск пайплана

ML & DNN. Шаблоны репозитория core

Текущий список шаблонов:

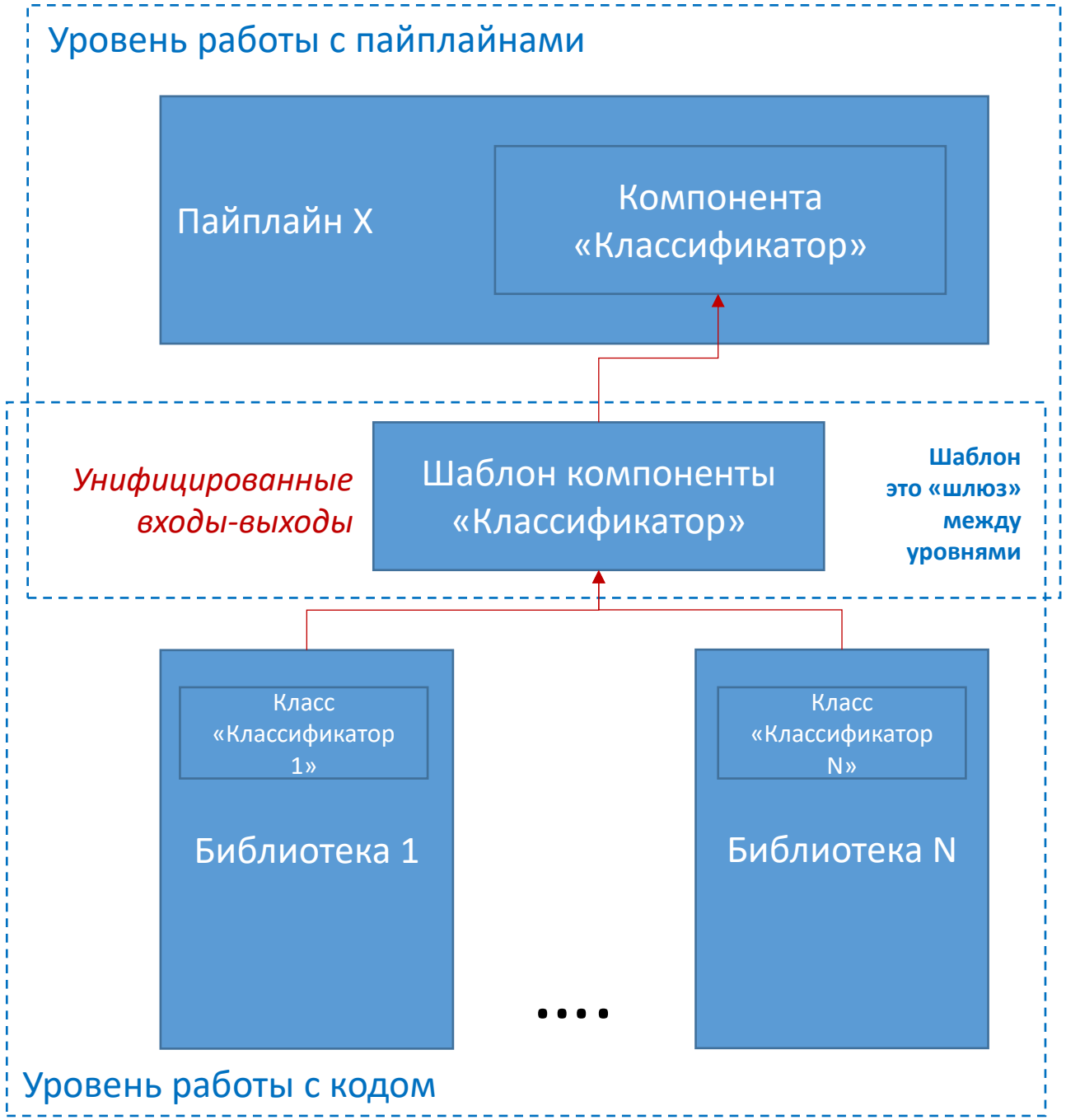
- **Утилиты**
  - Базы данных (Dataset)
  - Алгоритм оптимизации
  - Планировщик скорости обучения (Scheduler)
  - Функция ошибки
  - Аугментация данных
  - Метрики
  - Вывод информации (Logger)
- **Модели**
  - Блоки модели
  - Backbone (Encoder)
  - Head (Decoder)
  - Атомарная сеть



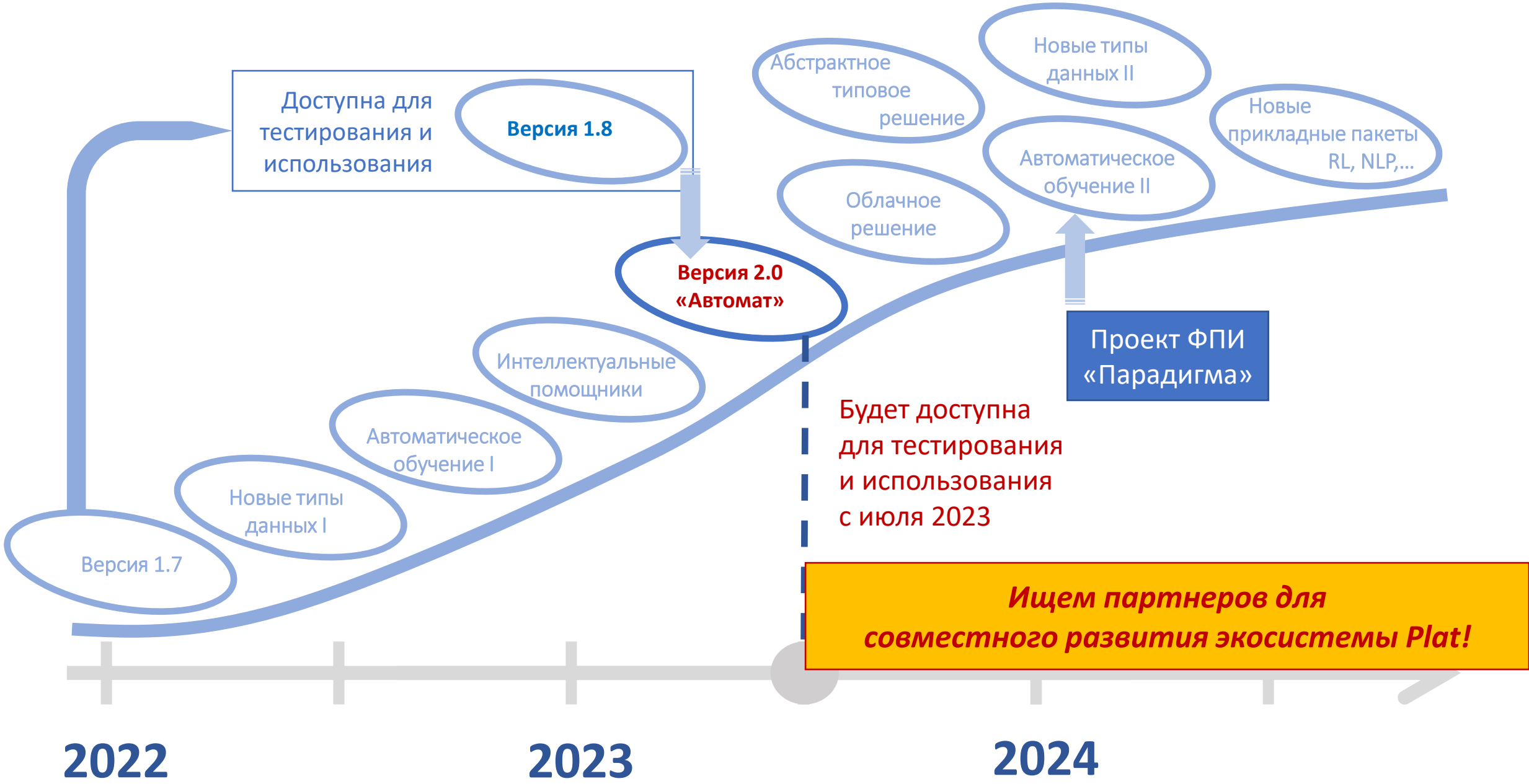
ML & DNN. Шаблоны репозитория core

Шаблоны позволяют:

- **Упростить и унифицировать написание кода отдельных компонент**, например, датасета, логгера, модели, алгоритма оптимизации и пр.
- **Унифицировать взаимодействие между компонентами**, использовать при необходимости контекст-менеджеры, зависимости от общего контекста выполнения, например, в случае распределенного обучения, использования смешанной точности (mixed precision), отключение расчета графа вычислений при валидации/тестировании
- **Унифицировать использование аугментаций, препроцессинга, постпроцессинга, прокси-задач…**
- **Унифицировать способ загрузки и визуализации данных**, если данные представимы в одном из существующих видов визуализации, полученных в ходе выполнения абстрактного типового решения. Например, развернуть метрику в таблицу по всем записям в датасете, либо по всем классам



Платформа-ГНС и PLAT: Перспективы развития (дорожная карта)



Платформа 2.0: Новое для разработчиков. PlatformAPI. PlatLib. ATP

Вишняков Борис Ваисович, начальник подразделения
«Искусственный интеллект и анализ данных» ФАУ «ГосНИИАС»

Спасибо за внимание!

PLAT-2023

Семинар-конференция «PLAT-2023»

Контакт по Платформе Plat:

support.plat@gosniias.ru

Москва, ГосНИИАС, 15.02.2023