Введение в курс DL

Кафедра технологий проектирования сложных технических систем

Вводное слово

- Нейронные сети получают все большее распространение на практике
- Повсеместные тенденции масштабирования (размер модели, размер набора данных, размер списка соавторов)
- Разработка на основе ipynb больше не является устойчивой
- Каждая модель - это гораздо больше, чем просто архитектура, потери и даже данные
- Инженерные знания пригодятся даже для исследований SOTA
- Для практических применений производительность и ремонтопригодность являются ключевыми факторами

Цель курса

- Практические детали обучения моделей и их теории в целом
- Небольшие изменения в коде могут значительно ускорить ваше обучение/логический вывод
- Развертывание обученных сетей как самостоятельно, так и в составе более крупной системы
- Упрощенное обслуживание за счет обращения с моделями ML как с любым другим кодом (тестирование, управление версиями и т.д.)

План курса

Содержание курса

1. Signal processing – Pytorch
    1. Фреймворки и pytorch
    2. Рекуррентные модели
2. Natural language processing
    1. Вероятностные модели, постановка задач
    2. Seq2Seq
    3. Attention, Transformer
3. Computer Vision
    1. Свертки и CNN, задача классификации
    2. Self-supervision, metric learning
    3. Сегментация, encoder-decoder
    4. Распознавание образов
    5. Трекинг объектов
4. Reinforcement learning
    1. Value policy
    2. Q-learning
5. Бонус (если успеем)
    1. Сжатие моделей, оптимизация и дистилляция моделей
    2. Docker, MLOps

Административные вопросы

1. Форма занятий

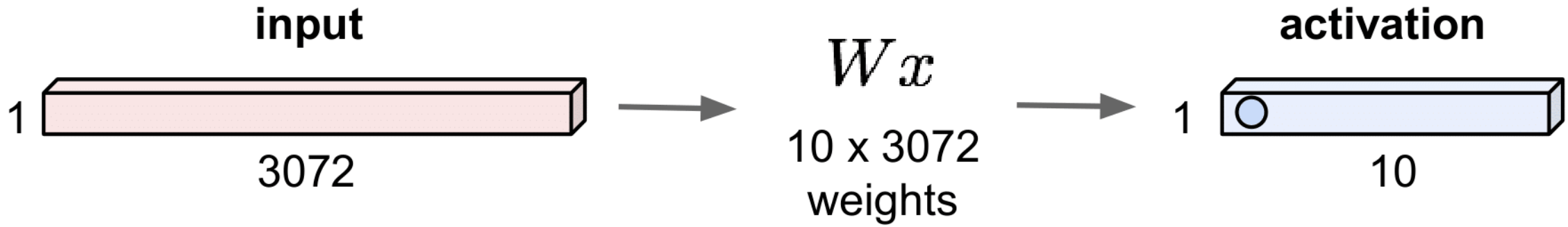
Записи занятий: будут
Форма отчетности: экзамен
Количество: 14 лекций и семинаров
Время занятий: суббота 10:35 – 13:30
Страница с оценками: будет опубликована в группе
Материалы: https://github.com/ml-dafe/ml_mipt_dafe

2. Оценка курса

Устный ответ на экзамене: 50%
Домашняя работа (допуск к экзамену): 50%

Нейронные сети – краткий гайд

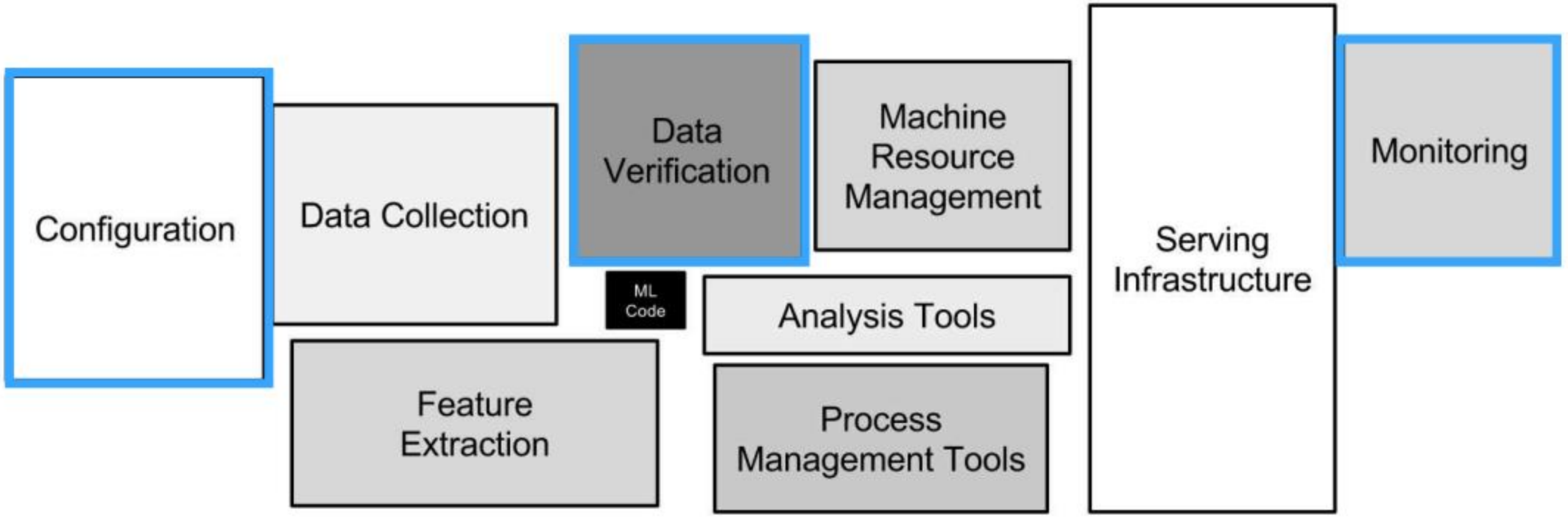
32x32x3 image -> stretch to 3072 x 1



Source: Stanford CS231n Lecture 5 2017 by Fei-Fei Li & Justin Johnson & Serena Yeung

Фреймворки и инструменты

- Фреймворки для обучения – Pytorch, PyTorch Lightning, Catalyst, Tensorflow, Keras, H2O
- Логирование экспериментов – Tensorboard, Neptune, Wandb, ClearML, DVC, Hydra
- Подготовка данных – CVAT, Label Studio
- Вычислительные ресурсы – Kaggle, Google Colab, Yandex Lens (гранты)



“Hidden Technical Debt in Machine Learning Systems”. Sculley et al., NeurIPS 2015

Pytorch

```
import torch

a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
```

$$Q = 3a^3 - b^2$$

$$\frac{\partial Q}{\partial a} = 9a^2$$

```
# check if collected gradients are correct
print(9*a**2 == a.grad)
print(-2*b == b.grad)
```

Out:

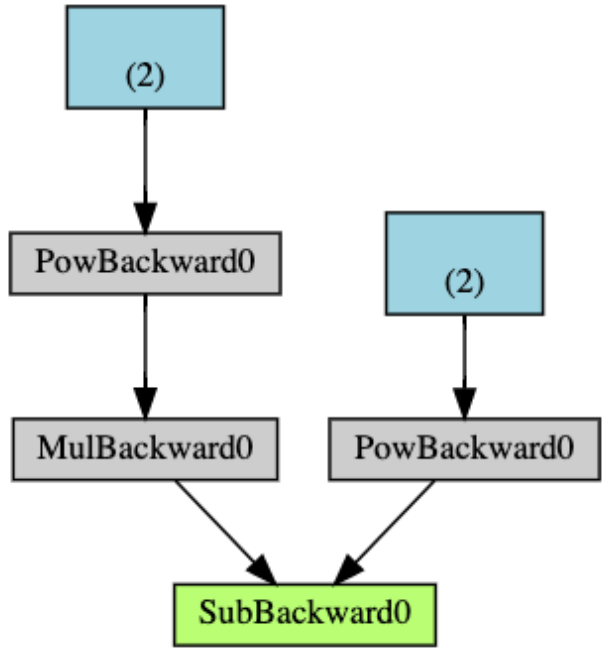
```
tensor([True, True])
tensor([True, True])
```

```
x = torch.rand(5, 5)
y = torch.rand(5, 5)
z = torch.rand((5, 5), requires_grad=True)

a = x + y
print(f"Does `a` require gradients? : {a.requires_grad}")
b = x + z
print(f"Does `b` require gradients?: {b.requires_grad}")
```

Out:

```
Does `a` require gradients? : False
Does `b` require gradients?: True
```



Computational Graph – directed acyclic graph (DAG)

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

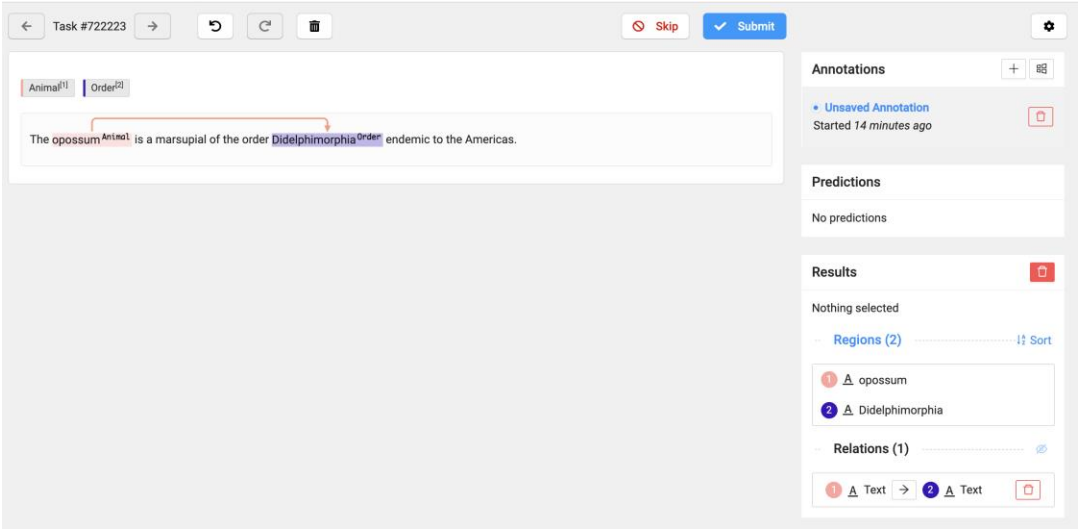
Подготовка данных

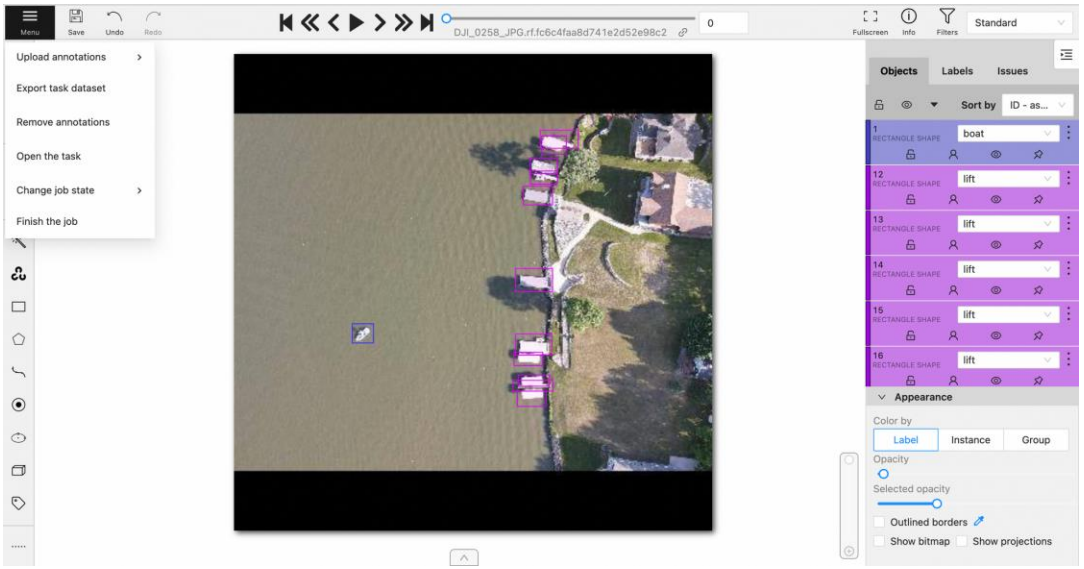
Поиск датасетов:

1. Google datasets
2. Kaggle datasets
3. Papers …

Ручная подготовка данных:

1. Разметка текста – Label Studio
2. Разметка изображений - CVAT

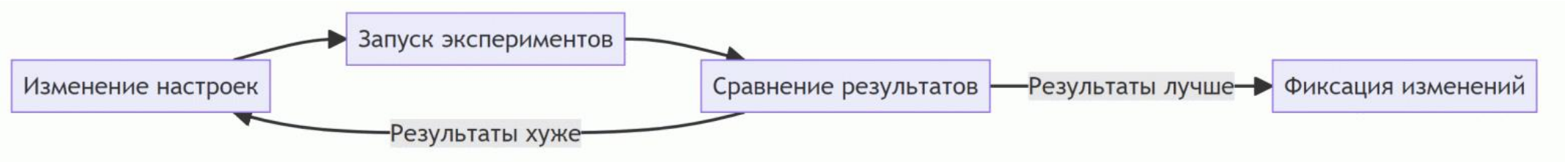




Логирование экспериментов

Инструменты – Tensorboard, Neptune, Wandb, ClearML, DVC, Hydra:

- Сохранить артефакты (конфигурации, графики, веса модели)
- Визуализация процесса обучения и валидации
- Логирование результатов и попыток
- Версионирование данных и исходного кода
- Зафиксировать окружение



Что логировать

Очевидно, что нам нужна таблица с идентификаторами выполнения и конечными показателями

- Графики с показателями за шаг/за секунду (конвергенция и эффективность)
- Хэш фиксации Git для воспроизводимости
- Визуализация входных/выходных данных модели
- Стандартный вывод/stderr вашего обучающего скрипта (потратьте время на хорошие журналы)
- В некоторых случаях полная информация об окружающей среде (Docker)

https://neptune.ai

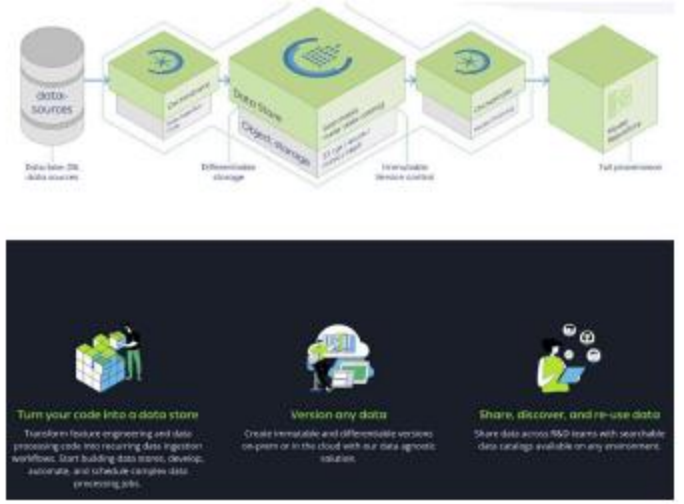
https://wandb.ai

Версионирование данных

- Код - не единственный компонент в вашей системе
- Данные - важнейшая зависимость, особенно в сложных конвейерах
- Отслеживание изменений
- Привязка каждого эксперимента к его данным повышает воспроизводимость

Версионирование данных. Решения

- Несколько существующих проектов позволяют интегрировать управление версиями артефактов в конвейеры
- Поддержка внешнего хранилища, сопоставление с коммитами, сравнение показателей
- Возможность повторного запуска определенных частей конвейера при изменении данных / конфигурации



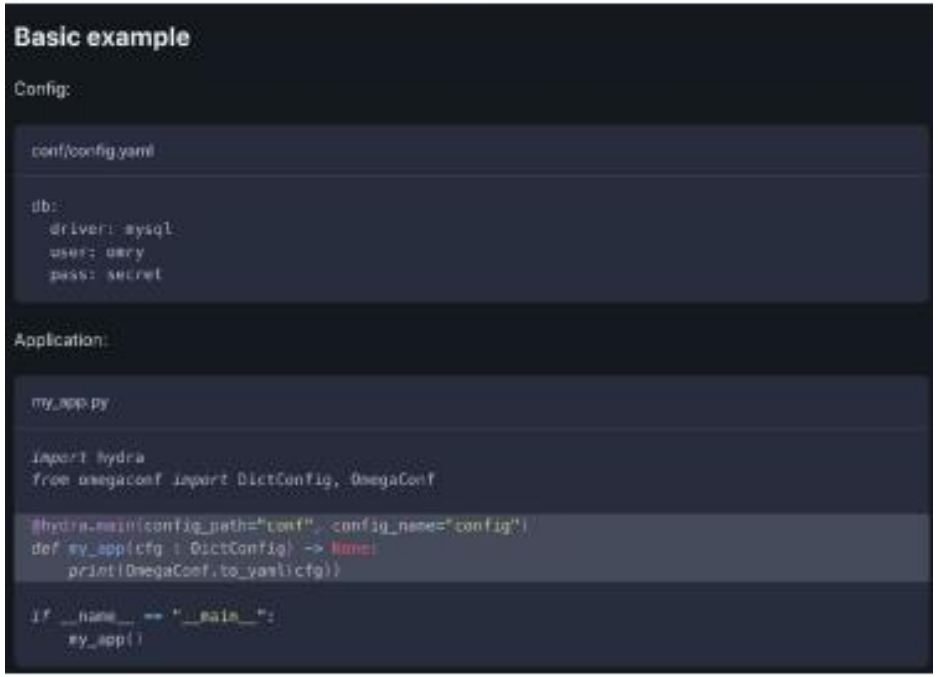
https://dvc.org/ https://www.pachyderm.com/ https://clear.ml/

Конфигурации экспериментов

- По мере роста вашего проекта количество “движущихся частей” увеличивается
  - Инфраструктура: конечные точки API, URL-адреса данных и т.д.
  - Гиперпараметры и компоненты модели
  - Изменение их вручную во всем репозитории не является устойчивым
- Решения на основе argparse/click сложны в написании и правильной версии
- Жесткое кодирование значений в выделенных файлах Python недостаточно гибко

Конфигурации экспериментов. Решение

- Одно из самых популярных решений для обработки конфигурации используется конфигурации YAML
  - Значения из командной строки
  - Простая проверка типов с помощью структурированных конфигураций
  - Сгруппированные конфигурации обеспечивают легкое переключение



https://hydra.cc/docs/intro/

Тестирование

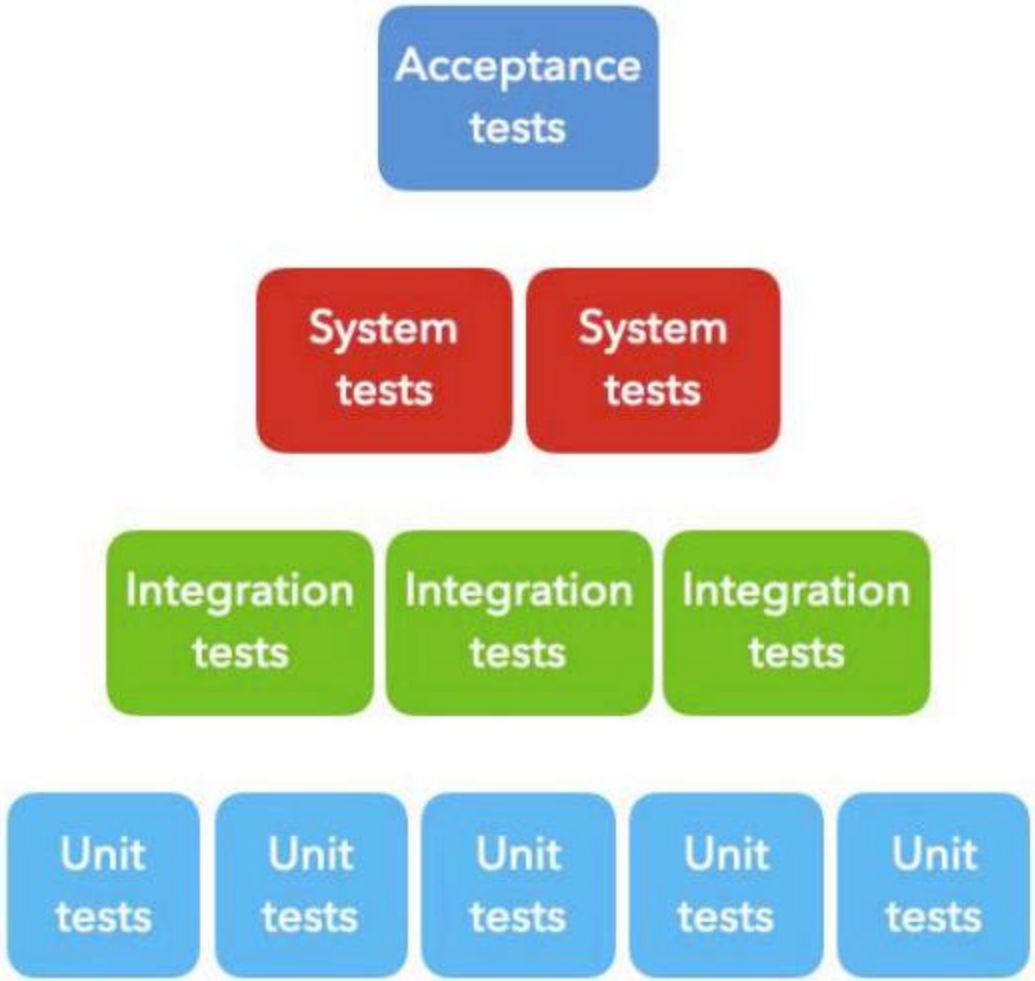
В общем, тестирование относится к проверке предполагаемых свойств кода:

- Не только корректности, но и производительности, обработки входных данных и т.д.

Почему мы должны тестировать наш код?

- Это помогает избежать ошибок (как сейчас, так и при рефакторинге)
- Но это не предотвращает их! Относитесь к тестам как к классификаторам, применяемым к вашему коду
- Это улучшает общее качество кода за счет разделения
- Самодокументированный код бесплатно

Виды тестирования



tests on the **requirements** for the application

tests on the **design** of a system by validating inputs with outputs

tests on the **integration** of individual components

tests on individual **components** that have *single responsibilities*

https://madewithml.com/courses/mlops/testing/

Виды тестирования

Существует множество видов и типологий, например:
1. **Unit tests** проверяют корректность работы отдельного компонента
2. **Integration tests** гарантируют совместную работу модулей
3. **End-to-end tests** проверяют корректность всего приложения
4. **Stress/load/performance tests** производительности проверяют скорость кода под нагрузкой

Мы сосредоточимся на 1 и 2: они самые простые в написании и охватывают большинство случаев

```python
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

Как тестировать

Встроенный Python: unittest
Довольно простой, готовый к использованию
Минусы: имеет свой собственный синтаксис, не такой гибкий

Лучше: pytest
Гибкий, работает с операторами assert, имеет множество интеграций с помощью плагинов


```
# content of test_sample.py
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 5
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                        [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
========================= short test summary info ==========================
FAILED test_sample.py::test_answer - assert 4 == 5
============================== 1 failed in 0.12s ===============================
```

Test-driven development in ML context

- Мы можем начать с бизнес-требований
- **Сделайте тесты естественной частью вашего рабочего процесса!** Это означает получение удобной настройки как локально, так и в CI
- Используйте TDD также и для вашего ML-кода
- Протестируйте ожидаемые изменения в поведении вашего режима

test-driven development, TDD

https://habr.com/ru/companies/ruvds/articles/450316/

О тестировании

- Как мы генерируем тестовые примеры?
  - Использование наших собственных входных данных не является исчерпывающим
  - По сути, мы проверяем только то, что код работает для заданных входных данных
  - Кроме того, наши требования становятся неясными

- Тестирование на основе свойств направлено на решение этой проблемы
  - Вместо того, чтобы указывать точные входные данные, мы сообщаем, какими они должны быть
  - Фреймворк тестирует код на многих входных данных и пытается упростить случаи сбоев