Pytorch

????????

Типы обучения:

- С учителем: есть данные и метки.
- Без учителя: есть данные и ожидаемое количество меток. Проводится классификация, затем говорится: это то, это= это.
- Трансферное обучение: используем некую уже обученную модель и дообучаем ее.
- Обучение с подкреплением: среда, агент и награда за корректный ответ.

Тензоры

```
scalar = torch.tensor(7)     # создание тензора
scalar.ndim                  # размерность 0 это один на один
scalar.item()                # элементы тензора
tns = torch.rand(3, 4)       # слчайный тензор
zero_tns = torch.zeros(3, 4) # тензор из нолей
one_tns = torch.ones(3, 4)   # тензор из единиц
one_tns.dtype                # тип данных в тензоре float32,
one_to_thirtyone = torch.arange(start=1, end=43, step=15) # tensor([ 1, 16, 31])
three_zeros_like = torch.zeros_like(input=one_to_thirtyone) # тензор похожий на шаблонный но
все нули
one_tns.T                    # транспонирование тензора
```

dtype определяет тип данных. По умолчанию float32

```
one_to_ten = torch.tensor([3.0, 6.0, 9.0], dtype=torch.float16)
print(one_to_ten.dtype)
float_16_tensor = float_32_tensor.type(torch.float16) # преобразование типов данных
```

device определяет устройство. По умолчанию cpu. Может быть cuda. Если тензоры на разных устройствах - будет ошибка.

```
tensor_on_gpu = tensor.to("cuda:0")
tensor_on_cpu = tensor_on_gpu.cpu()
```

Тензор на gpu нельзя преобразовать в numpy

requires_grad=False - расчет градиентов.

3 частые ошибки при работе с тензорами:

- неправильный тип данных
- неправильная форма
- тензоры при операции на разных устройствах

Операции с тензорами

Скалярные сложение, разность, умножение, деление тензора на число - как в python.

Векторное умножение torch.matmul(tenz, tenz)

Дополнительные методы

```
torch.min # само значение
torch.max

torch.argmin # индекс минимального значения
torch.argmax

torch.mean # усреднение значений
torch.sum

# reshape - преобразование размера входного тензора в нужный размер
# размер (кол-во элементов, т е произведение размерностей) нового и старого тензоров должны совпадать.
x_reshaped = x.reshape(3, 3)
# view - возвращает  вид входного тензора в указанном размере, но сохраняет память оригинального тензора
# т е это просто ссылка, а reshape создает копию
x_reviewed = x.view(1,9)
# stack - склеивает тензоры по горизонтали или вертикали
# dim=0 - по горизонтали
# dim=1 - по вертикали
x_new = torch.stack([x,x,x], dim=0)
# squeeze - удаляет все единичные измерения из тензора

# unsqueeze - добавляет одно единичное измерение к тензору

# permute - изменение порядка на входе функции последовательность новой
x_new = x_cur.permute(2,1,0)
```

Нумерация элементов тензора

```
x = torch.arange(1., 10.)
x_v = x.view(1, 3, 3)
print(x_v[0,2,2]) # третий элемент третьей строки первого блока
print(x_v[:,:,2]) # tensor([[3., 6., 9.]])
```

При конвертации в pytorch обязательно помнить о возможном несоответствии типов данных

Воспроизводимость: получить такой же случайный тензор. Нужно вызывать перед каждой генерацией случайных чисел.

torch.manual_seed(SOME_NUMBER)

Функции потерь

Разница в каком-то виде между требуемыми и полученными данными. Их довольно много.

Запуск на GPU

Тренды библиотек ИИ

https://www.learnpytorch.io

https://colab.research.google.com/

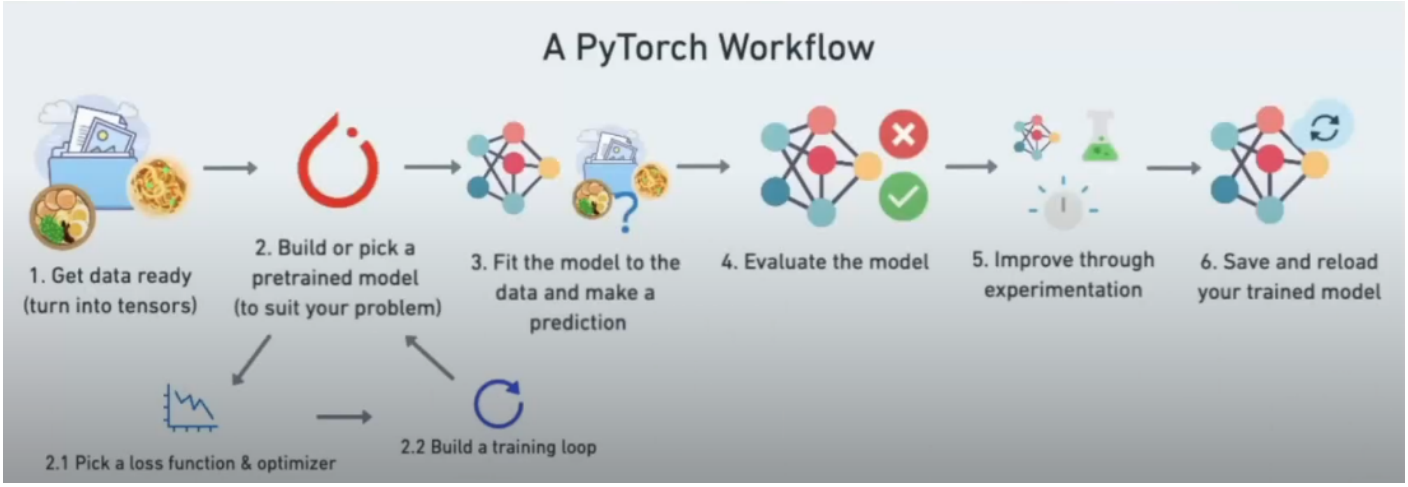
Библиотека машинного обучения. Можно запускать предобученные модели. Например для компьютерного зрения:

```
from torchvision import models


print(dir(models))
```

Названия в верхнем регистре - классы, реализующие популярные архитектуры. В нижнем - экземпляры сетей с определенным количеством слоев, нейронов, возможно с весами, т е объекты.

??????? ???????? ?????? ? pytorch



- Класс nn содержит все необходимое для создания нейросети
- nn.Parameter - какие параметры может наша модель пробовать обучить, обычно слой PyTorch будет настраивать их
- nn.Module - базовый класс нейросетей, все модели потомки этого класса. Должен содержать переопределенную процедуру forward
- torch.optim - оптимизатор при создании модели
- torch.inference_mode() - режим вывода при прогнозировании. Убирает очень много дополнительных данных

Приближение модели к нужным параметрам происходит с помощью фукции потерь (Loss function) Она показывает, насколько сильно полученные данные отличаются от требуемых.

Оптимизатор учитывает потери модели и корректирует параметры модели.

```
import torch
from torch import nn
import matplotlib.pyplot as plt

'''
Задача: сделаем данные на основе линейной регрессии (линейного уравнения).
Обучим модель и попытаемся предсказать значения.
'''

def plot_data(train_data=[], train_labels=[],
```

```
                test_data=[], test_labels=[],
                prediction=None):
    plt.figure(figsize=(10, 7))
    plt.scatter(train_data, train_labels, c='b', s=4)
    plt.scatter(test_data, test_labels, c='g', s=4)
    if prediction is not None:
        plt.scatter(test_data, prediction, c='r', s=4)

    plt.show()

class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(1,
                                                requires_grad=True,
                                                dtype=torch.float))
        self.bias = nn.Parameter(torch.randn (1,
                                              requires_grad=True,
                                              dtype=torch.float))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.weights * x + self.bias


weight = 0.7
bias = 0.3

# стартовый набор данных
start = 0
end = 1
step = 0.02

X = torch.arange(start, end, step).unsqueeze(dim=1)
Y = weight * X + bias

# Разделим данные на обучающие 60%-80%, тестовые 20%
train_size = int(0.8 * len(X))
X_train, Y_train = X[:train_size], Y[:train_size]
X_test, Y_test = X[train_size:], Y[train_size:]
```

```
torch.manual_seed(42)
model_0 = LinearRegressionModel()
print(list(model_0.parameters()))

# делаем предсказание о качестве метки
with torch.inference_mode():
    y_preds = model_0(X_test)

# Рисуем тестовые данные
plot_data(X_train, Y_train, X_test, Y_test, y_preds)
```

Сохранение модели.

torch.save() / torch.load() Стандартная сериализация и сохранение любого объекта, в частности - torch. Можно через state_dict, Сохраняет текущее состояние модели. По ощущениям проще save/load. Хотя наверное при гигабайтных моделях объект будет весить 1:10.

```
from pathlib import Path

model_0 = LinearRegressionModel()
...

MODEL_PATH = Path("models")
MODEL_PATH.mkdir(parent=True, exist_ok=True)
MODEL_NAME = "testname.pt"
MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

torch.save(obj=model_0.state_dict(), f=MODEL_SAVE_PATH)
# новый объект, загруженный из файла
loaded_model = LinearRegressionModel()
loaded_model.load_state_dict(torch.load(f=MODEL_SAVE_PATH))
```

???? PyTorch

Например есть класс.

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.weights * x + self.bias
```

nn.Linear

Слой линейной регрессии, in_features - кол-во входящих переменных, out_features - кол-во исходящих значений. В функции Y = F(X) это размерность входного и получаемого тензоров.