**СОДЕРЖАНИЕ**

[Введение 3](#_Toc136338267)

[1 Создание класса 4](#_Toc136338268)

[1.1 Инициализация класса 4](#_Toc136338269)

[1.2 Методы класса 4](#_Toc136338270)

[2 Инструмент Node-Red 2](#_Toc136338271)

[2.1 Описание Node-Red 2](#_Toc136338272)

[3 Примеры использования системы 4](#_Toc136338273)

[3.1 Классификация Cifar-10 4](#_Toc136338274)

[3.1.1 Описание задачи 4](#_Toc136338275)

[3.1.2 LeNet 5](#_Toc136338276)

[3.1.3 ResNet 6](#_Toc136338277)

[3.1.4 Создание системы на Node-Red 7](#_Toc136338278)

[3.1.5 Результат 7](#_Toc136338279)

[3.2 Семантическая сегментация Cityscapes 8](#_Toc136338280)

[3.2.1 Описание задачи 8](#_Toc136338281)

[3.2.2 Fully Convolutional Network (FCN) 9](#_Toc136338282)

[3.2.3 UNet 11](#_Toc136338283)

[3.2.4 Создание системы на Node-Red 13](#_Toc136338284)

[3.2.5 Результат 13](#_Toc136338285)

[3.3 Классификация красных автомобилей в дорожном потоке. 15](#_Toc136338286)

[3.3.1 Описание задачи 15](#_Toc136338287)

[3.3.2 Создание системы на Node-Red 16](#_Toc136338288)

[3.3.3 Результат 16](#_Toc136338289)

[Заключение 17](#_Toc136338290)

[Список использованных источников 18](#_Toc136338291)

[Приложение А Код программы 19](#_Toc136338292)

Введение

Задача: разработать систему, позволяющую осуществлять последовательный и параллельный запуск нейросетей с целью оптимизации ресурсов.

Цель: разработать систему, которая реализует поставленную задачу.

Для достижения цели были выполнены следующие действия:

1. Изучение информации о создании нейросетей.
2. Изучение инструмента потокового программирования Node-Red.
3. Создание класса на языке Python, позволяющего создавать программы-нейросети путём применения описанных в нём функций.
4. Применение разработанного класса к инструменту Node-Red с целью создания системы, реализующей поставленную задачу.
5. Решение задачи классификации изображения на наборе данных Cifar-10 с применением архитектур LeNet, ResNet и разработанной системы.
6. Решение задачи сегментации на наборе данных Cityscapes с применением архитектур UNet, FCN и разработанной системы.
7. Решение задачи сегментации изображения с последовательным применением архитектур UNet, ResNet и разработанной системы с целью оптимизации ресурсов при решении задачи.
8. Создание класса
   1. Инициализация класса

Для инициализации класса используется словарь, содержащий пути ко всем функциям, которые будут вызываться в программе, а также значения всех параметров, которые будут переданы в них.

Пример ключа:

params = {

"dataloader\_path": "C:\\files\\class\_dataset\\segmentation\_dataset",

"model\_path": "C:\\files\\models\\FCN",

"train\_path": "C:\\files\\models\\UNet",

"predict\_path": "C:\\files\\models\\UNet",

"save\_path": "C:\\files\\statistics\\FCN.xlsx",

"data\_dir": "C:\\files\\dataset\\cityscapes\_data",

"pretrained\_model": None,

"output\_dir": "C:\\files\\Images",

"model\_name": "FCN",

"image\_count": 2,

"num\_items": 10000,

"batch\_size": 16,

"valid\_size": 0.2,

"save\_model": None,

"num\_classes": 5,

"learning\_rate": 0.01,

"num\_epochs": 0

}

* 1. Методы класса

Методы извлекают из словаря, содержащегося в поле класса, путь к python-файлу. Данный файл содержит реализацию функции, название которой совпадает с названием метода, который мы вызываем. Импортируем функцию и вызываем её, передав в неё словарь и требуемые для неё параметры.

Содержащиеся методы: model, dataloader, train, predict, test\_accuracy, train\_stat.

Пример метода:

def dataloader(self):

"""

Данная функция импортирует функции для создания выборок и преобразования

их в даталоадеры (загрузчики выборок в нейросети)

"""

# Извлекаем значения параметров и импортируем функцию

loader\_path = self.params['dataloader\_path']

sys.path.append(loader\_path)

module = \_\_import\_\_("dataloader")

dataloader = getattr(module, "dataloader")

loaders = dataloader(self.params)

print("\_\_\_\_\_\_\_\_\_\_\_Dataloaders creation was successful\_\_\_\_\_\_\_")

return loaders

1. Инструмент Node-Red
   1. Описание Node-Red

Node-Red – это инструмент потокового программирования, основанный на JavaScript. Данный инструмент работает по следующему принципу:

1. Каждый узел является модулем с определенной функциональностью, которая не зависит от других узлов.
2. Узлы размещаются на рабочей области интерфейса с помощью перетаскивания из палитры.
3. Узлы соединяются между собой каналами, по которым происходит передача данных между ними.
4. Для запуска проекта необходимо его задеплоить и активировать inject-узел.

Таким образом, можно создать GUI для программы, где каждый узел представляет собой нейросеть или обработчик результатов ее работы, а по каналам передаются данные, полученные в результате работы узлов.

Для запуска кода, написанного на Python, необходимо использовать узел pythonshell, который не входит в стандартную палитру узлов Node-RED.

Узел pythonshell работает следующим образом: информация, пришедшая по каналам в узел, передается Python как аргументы, а вывод Python кода возвращается узлом в виде payload.

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

Рисунок — Демонстрация Node-Red

1. Примеры использования системы
   1. Классификация Cifar-10
      1. Описание задачи

Для демонстрации работы с разработанной системой решим задачу классификации изображения на наборе данных Cifar-10 и определим, какая из архитектур: LeNet или ResNet больше подходит для этого. Cifar-10 содержит 60000 цветных изображений размера 32 на 32 пикселя разделенных на 10 классов: «самолет», «автомобиль», «птица», «кошка», «олень», «собака», «лягушка», «лошадь», «корабль», «грузовик».

Изображение выглядит как снимок экрана, коллаж

Автоматически созданное описание

Рисунок ­­ — Примеры изображений

Реализуем решение поставленной задачи на языке Python с помощью фреймворка Pytorch, используя архитектуры LeNet и ResNet. Затем адаптируем код под Nde-Red и произведём запуск полученной системы, получив в результате Excel-файл, содержащий параметры наших нейросетей, а также достигнутые метрики. На основе собранной информации сравним точность выбранных архитектур в рамках нашей задачи.

* + 1. LeNet

LeNet – это одна из первых сверточных нейросетей, разработанная в 1998 году для распознавания рукописных цифр и символов.

Используемая реализация LeNet имеет следующую структуру:

1. Сверточный слой: принимает изображение в RGB и применяет свертку с 32 фильтрами размером 5 x 5 пикселей.
2. Батч-нормализация: нормализует выход сверточного слоя по батчу, то есть группе изображений.
3. Сверточный слой: применяет свертку с 64 фильтрами размером 5 x 5 пикселей на предыдущем выходе.
4. Батч-нормализация: нормализует выход сверточного слоя по батчу, то есть группе изображений.
5. Сверточный слой: применяет свертку с 128 фильтрами размером 3 x 3 пикселя на предыдущем выходе.

Будем обучать нейросеть, используя следующие параметры:

1. batch\_size: 64;
2. optimizer: SGD;
3. criterion: CrossEntropyLoss;
4. device: cpu;
5. valid\_size: 0.2;
6. num\_classes: 10;
7. beta1: 0.8;
8. beta2: 0.98;
9. learning\_rate: 0.01;
10. momentum: 0.9;
11. num\_epochs: 60;
    * 1. ResNet

ResNet – это архитектура сверточных нейронных сетей, предложенная Хе и другими в 2015 году для решения задачи классификации изображений1 ResNet стоит за Residual Network, то есть сеть с остаточными соединениями. Остаточные соединения – это специальные виды пропусков, которые позволяют модели пропускать один или несколько слоев. Это помогает решить проблему затухающего/взрывающегося градиента, которая возникает при обучении очень глубоких нейронных сетей. Благодаря этому подходу ResNet может обучаться на сотнях или даже тысячах слоев и все еще достигать высокой производительности.

Используемая реализация ResNet имеет следующую структуру:

1. Входной слой: принимает изображение в RGB и применяет свертку, батч-нормализацию и ReLU.
2. Сверточный слой: применяет свертку, батч-нормализацию, ReLU и макс-пулинг.
3. Остаточный блок: состоит из двух подблоков, каждый из которых имеет свертку, батч-нормализацию и ReLU. Выходы подблоков складываются с входами блока или их проекциями и применяется ReLU. Это формирует остаточные соединения.
4. Сверточный слой: применяет свертку, батч-нормализацию, ReLU и макс-пулинг.
5. Сверточный слой: применяет свертку, батч-нормализацию, ReLU и макс-пулинг.
6. Остаточный блок: аналогичен предыдущему.
7. Классификатор: применяет макс-пулинг, выравнивание, линейный слой и softmax.

Будем обучать нейросеть, используя следующие параметры:

1. batch\_size: 64;
2. optimizer: SGD;
3. criterion: CrossEntropyLoss;
4. device: cpu;
5. valid\_size: 0.2;
6. num\_classes: 10;
7. beta1: 0.8;
8. beta2: 0.98;
9. learning\_rate: 0.01;
10. momentum: 0.9;
11. num\_epochs: 60;
    * 1. Создание системы на Node-Red

Изображение выглядит как текст, программное обеспечение, линия, снимок экрана

Автоматически созданное описание

Рисунок — Классификация в Node-Red

* + 1. Результат

В результате запуска данной системы подсчитанные метрики и параметры нейросетей были записаны в Excel-файл. Обучим нейросети на 30 и 60 эпохах.

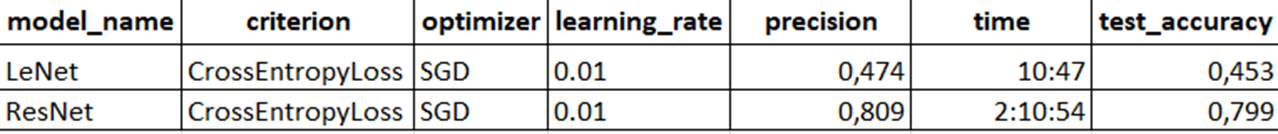


Рисунок — Таблица метрик LeNet и ResNet (30 эпох)

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Рисунок — Таблица метрик LeNet и ResNet (60 эпох)

Изучив результаты, можем сделать вывод о том, что, несмотря на ощутимо большее время обучения, архитектура ResNet показывает высокую точность в сравнении с LeNet.

* 1. Семантическая сегментация Cityscapes
     1. Описание задачи

Для демонстрации работы с разработанной системой решим задачу семантической сегментацией изображения на наборе данных Cityscapes и определим, какая из архитектур: FCN или UNet больше подходит для этого. Cityscapes – это набор данных, состоящий из 20000 изображений разнообразных городских уличных сцен в 50 разных городах в разное время года, а также маркировку к ним.

Изображение выглядит как снимок экрана, Компьютерная игра, Программное обеспечение для видеоигр, мультфильм

Автоматически созданное описание

Рисунок — Пример аннотации к изображениям.

Реализуем решение поставленной задачи на языке Python с помощью фреймворка Pytorch, используя архитектуры FCN и UNet. Затем адаптируем код под Node-Red и произведём запуск полученной системы, получив в результате Excel-файл, содержащий параметры наших нейросетей, а также достигнутые метрики. Так же выведем на экран предсказанные маски для текстового набора данных. На основе собранной информации сравним точность выбранных архитектур в рамках нашей задачи.

* + 1. Fully Convolutional Network (FCN)

Fully Convolutional Network – это архитектура, используемая в основном для семантической сегментации изображений. Она состоит только из локально связанных слоев, таких как свертка, пулинг и апсемплинг. Отсутствие полносвязных слоев означает меньше параметров (что делает сеть быстрее в обучении). Также это означает, что FCN может работать с изображениями разных размеров, так как все связи локальные. Сеть состоит из пути уменьшения размерности, используемого для извлечения и интерпретации контекста, и пути увеличения размерности, который позволяет локализовать объекты.

Используемая реализация архитектуры FCN имеет следующую структуру:

Encoder:

1. Сверточный блок с двумя сверточными слоями, батч-нормализацией и ReLU активацией для каждого слоя.
2. Сверточный блок с двумя сверточными слоями, батч-нормализацией и ReLU активацией для каждого слоя.
3. Сверточный блок с двумя сверточными слоями, батч-нормализацией и ReLU активацией для каждого слоя.
4. Слой макс-пулинга для уменьшения размерности.

Decoder:

1. Обратный сверточный блок с двумя обратными сверточными слоями, батч-нормализацией и ReLU активацией для каждого слоя.
2. Обратный сверточный блок с двумя обратными сверточными слоями, батч-нормализацией и ReLU активацией для каждого слоя.
3. Обратный сверточный блок с двумя обратными сверточными слоями, батч-нормализацией и ReLU активацией для каждого слоя.

Будем обучать нейросеть, используя следующие параметры:

1. optimizer: Adam;
2. criterion: CrossEntropyLoss;
3. device: cpu;
4. image\_count: 3;
5. num\_items: 10000;
6. batch\_size: 16;
7. valid\_size: 0.2;
8. num\_classes: 5;
9. learning\_rate: 0.01;
10. num\_epochs: 20;
    * 1. UNet

Unet – это архитектура для семантической сегментации изображений, которая усовершенствована на основе FCN. Она состоит из сужающего пути (encoder) для захвата контекста и симметричного расширяющегося пути (decoder) для точной локализации. Она также использует пропускные соединения (skip connections) для объединения карт признаков из сужающего и расширяющегося путей.

Используемая реализация архитектуры UNet имеет следующую структуру:

Encoder:

1. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
2. Слой макс-пулинга для уменьшения размерности.
3. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
4. Слой макс-пулинга для уменьшения размерности.
5. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
6. Слой макс-пулинга для уменьшения размерности.
7. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
8. Слой макс-пулинга для уменьшения размерности.
9. Middle:
10. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.

Decoder:

1. Обратный сверточный слой для увеличения размерности.
2. Конкатенация с обрезанной картой признаков из encoder.
3. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
4. Обратный сверточный слой для увеличения размерности.
5. Конкатенация с обрезанной картой признаков из encoder.
6. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
7. Обратный сверточный слой для увеличения размерности.
8. Конкатенация с обрезанной картой признаков из encoder.
9. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.
10. Обратный сверточный слой для увеличения размерности.
11. Конкатенация с обрезанной картой признаков из encoder.
12. Сверточный блок с двумя сверточными слоями, ReLU активацией и батч-нормализацией для каждого слоя.

Output:

1. Сверточный слой для получения карты признаков с num\_classes каналами.
2. Будем обучать нейросеть, используя следующие параметры:
3. optimizer: Adam;
4. criterion: CrossEntropyLoss;
5. device: cpu;
6. image\_count: 3;
7. num\_items: 10000
8. batch\_size: 16;
9. valid\_size: 0.2;
10. num\_classes: 5;
11. learning\_rate: 0.01;
12. num\_epochs: 20;
    * 1. Создание системы на Node-Red

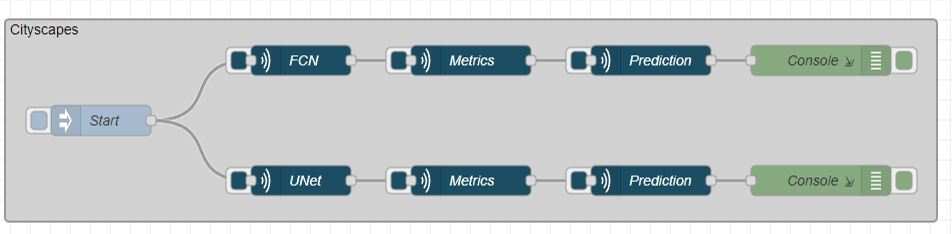


Рисунок — Сегментация в Node-Red

* + 1. Результат

В результате запуска данной системы подсчитанные метрики и параметры нейросетей были записаны в Excel-файл. Обучим нейросети на 20 эпохах.

Изображение выглядит как снимок экрана, текст

Автоматически созданное описание

Рисунок — Сегментация на тестовых данных с помощью UNet

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рисунок — Сегментация на тестовых данных с помощью FCN

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок — Таблица метрик LeNet и ResNet (60 эпох)

Метрика IoU оценивает качество сегментации объектов путем сравнения исходной сегментации и предсказанной сегментации. Она вычисляется путем нахождения отношения площади пересечения этих двух сегментаций к площади их объединения. Чем ближе это отношение к единице, тем более точно предсказана сегментация объектов на изображении.

Сравнив нейросети в действии и опираясь на значения метрики Iou можем сделать вывод, что UNet лучше подходит для решения нашей задачи.

* 1. Классификация красных автомобилей в дорожном потоке.
     1. Описание задачи

Продемонстрируем возможность оптимизации ресурсов при решении задач с помощью применения разработанного класса и инструмента Node-Red. Нами ранее была разработана нейросеть с архитектурой UNet для сегментации изображений дорожного потока и уличных сцен из датасета Cityscapes. Данная нейросеть была обучена выделять 10 различных классов. Предположим, перед нами была поставлена задача сегментировать только те изображения, на которых отсутствуют люди. Осуществлять решение поставленной задачи путем изменения UNet предполагало бы пересмотр количества классов и обучение нейросети с нуля с учетом новых параметров. Кроме того, если на изображении всё же присутствуют люди, то мы зря пропускали это изображение через нашу довольно глубокую нейросеть. Эти проблемы можно оптимизировать, воспользовавшись функционалом нашего класса и Node-Red. Для этого разделим нашу задачу на две. Для начала будем определять, есть ли на изображении люди и лишь в том случае, если данного класса объектов на фотографии нет, будем осуществлять сегментацию. Будем последовательно применять две нейросети. В качестве первой будет выступать описанная ранее ResNet. Обучим её определять наличие людей в кадре, используя выборки из всё того же набора Cityscapes. Затем для каждого изображения из тестового набора данных будем осуществлять предсказание и в соответствии с ним помещать каждое изображение в соответствующий каталог. Затем, к изображениям, хранящимся в каталоге с фотографиями без людей в кадре, применим нашу нейросеть UNet. Таким образом, обучение ResNet на новых данных займёт значительно меньше времени, чем заняло бы обучение UNet. А также мы экономим время и мощности, пропуская изображения преимущественно через ResNet.

Структуру архитектур, а также набор параметров оставим без изменений. Изменится только размер входных изображений и количество классов для ResNet.

* + 1. Создание системы на Node-Red

Изображение выглядит как снимок экрана, текст

Автоматически созданное описание

Рисунок — Классификация и сегментация в Node-Red

* + 1. Результат

В результате обучения ResNet на выборке из датасета Cityscapes в течение 40 эпох получили следующие данные:

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

Рисунок — Результат обучения ResNet

Таким образом, полученная система с вероятностью 80% определяет присутствие или отсутствие людей на изображении. Затем, помещает изображения в директории в соответствии с предсказанными метками. Пути к директориям подаются в ранее обученную нейросеть UNet в качестве выборки для создания тестового загрузчика данных. Для данной выборки осуществляется предсказание маски.

Заключение

В ходе работы были выполнены следующие задачи:

1. Изучена информация о создании нейросетей.
2. Освоен инструмент потокового программирования Node-Red.
3. Разработан класс на языке Python, позволяющиц создавать программы-нейросети с помощью встроенных функций.
4. Разработанный класс был интегрирован в инструмент Node-Red для создания системы.
5. Было продемонстрировано применение системы к задаче классификации изображения на наборе данных Cifar-10 с использованием архитектур LeNet, ResNet и разработанной системы.
6. Было продемонстрировано применение системы к задаче сегментации на наборе данных Cityscapes с использованием архитектур UNet, FCN и разработанной системы.
7. Было продемонстрировано применение системы к задаче сегментации изображения с последовательным применением архитектур UNet, ResNet и разработанной системы для оптимизации ресурсов при решении задачи.

Таким образом, созданная система позволяет комбинировать нейросети и оценивать их работу в потоковом режиме. Система также демонстрирует возможность оптимизации ресурсов при решении задач путём их дробления на более простые или уже решенные.

Систему можно улучшить, если:

1. Упростить взаимодействие с системой
2. Автоматически формировать пути к используемым функциям
3. Автоматизировать подбор оптимальных параметров для нейросетей.
4. Расширить функционал класса, путём добавления новых функций, например: вывода графиков, объединения изображения в видео с целью их сегментации.

Список использованных источников

1. Архитектуры нейросетей // Habr URL: <https://habr.com/ru/companies/nix/articles/430524/> (Дата обращения 27.10.2022)
2. Лекция 7. Задача сегментации. // Teach-in URL: <https://teach-in.ru/lecture/2019-11-11-Konushin> (Дата обращения 23.04.2023)
3. Курс «Нейронные сети и компьютерное зрение» // Stepik URL: https://stepik.org/course/50352/syllabus (Дата обращения 02.02.2023)

Приложение А

Код программы

Реализация класса Neuron\_node:

import sys

import torch

import pandas as pd

class neuron\_node(object):

def \_\_init\_\_(self, params):

"""

Из файла, хранящегося по пути params\_path считываем значения параметров в словарь params.

Единственным полем у класса является словарь, содержащий параметры

"""

self.params = params

def model(self, \*args):

"""

Функция для создания экземпляра модели нейросети. Импортирует класс нейросети, путь к которому

указан в параметрах, затем создаёт экземпляр этой модели.

Извлекаем из словаря параметры для запуска обучения модели.

Подключаем модуль с моделью нейросети.

Переменная model\_module содержит в себе модуль, хранящий в себе класс нейросети

Переменная model содержит в себе класс нейросети. То есть с помощью model уже можно как раз создавать

экземпляр класса

"""

model\_path = self.params['model\_path']

sys.path.append(model\_path)

module = \_\_import\_\_("model")

model = getattr(module, "model")

# Создаём экземпляр модели

model = model(self.params, args)

pretrained\_model = self.params['pretrained\_model']

if pretrained\_model:

model.load\_state\_dict(torch.load(pretrained\_model))

print("\_\_\_\_\_\_\_\_\_\_\_Model creation was successful\_\_\_\_\_\_\_")

return model

def dataloader(self):

"""

Данная функция импортирует функции для создания выборок и преобразования

их в даталоадеры (загрузчики выборок в нейросети)

"""

# Извлекаем значения параметров

loader\_path = self.params['dataloader\_path']

sys.path.append(loader\_path)

module = \_\_import\_\_("dataloader")

dataloader = getattr(module, "dataloader")

loaders = dataloader(self.params)

print("\_\_\_\_\_\_\_\_\_\_\_Dataloaders creation was successful\_\_\_\_\_\_\_")

return loaders

def train(self, \*args):

"""

Импортирует функцию train, путь к которой указан в параметрах,

затем запускает эту функцию и обучает модель.

"""

# Импортируем класс с методом train

train\_path = self.params.get("train\_path")

sys.path.append(train\_path)

train\_module = \_\_import\_\_("train")

train = getattr(train\_module, "train")

train\_model = train(self.params, \*args)

# Запуск обучения нейросети

print('----------- Finished Training -----------')

return train\_model

# Функция предсказания обученной модели на тестовых данных test\_loader для экземпляра модели model

def predict(self, \*args):

# Импортируем функцию predict, хранящуюся по пути predict\_path

predict\_path = self.params['predict\_path']

sys.path.append(predict\_path)

predict\_module = \_\_import\_\_("predict")

predict = getattr(predict\_module, "predict")

predict(self.params, \*args)

print('----------- Finished predicting -----------')

# Функция подсчёта метрик на тестовых данных test\_loader для экземпляра модели model

def test\_accuracy(self, \*args):

# Импортируем функцию test\_accuracy, хранящуюся по пути test\_accuracy\_path

test\_accuracy\_path = self.params['test\_accuracy\_path']

sys.path.append(test\_accuracy\_path)

test\_accuracy\_module = \_\_import\_\_("test\_accuracy")

test\_accuracy = getattr(test\_accuracy\_module, "test\_accuracy")

# Вызываем метод train и возвращаем показатель точности

return test\_accuracy(self.params, \*args)

def train\_stat(self, \*args):

train\_loss\_list, accuracy\_list, valid\_loss\_list, time\_list, save\_path = args

train\_stat = pd.DataFrame()

train\_stat['train\_loss'] = pd.Series(train\_loss\_list)

train\_stat['valid\_loss\_list'] = pd.Series(valid\_loss\_list)

train\_stat['accuracy\_list'] = pd.Series(accuracy\_list)

train\_stat['time\_list'] = pd.Series(time\_list)

train\_stat.reset\_index()

train\_stat.to\_excel(save\_path, index=True)

print('----------- Finished creating file -----------')

"""

Функция для объединения информации о метриках и параметрах модели, на которых эти метрики

были достигнуты. Информация выводится в виде таблицы

"""

def model\_report(model, metrics, model\_params, indexes):

from collections import defaultdict

layer\_names = [] # список для хранения названий слоев

layer\_counts = [] # список для хранения количества слоев каждого типа

layer\_dict = defaultdict(int)

for layer in model.modules():

layer\_dict[type(layer).\_\_name\_\_] += 1

for layer\_type, count in layer\_dict.items():

layer\_names.append(layer\_type)

layer\_counts.append(count)

for metr in layer\_names:

metrics.append(metr)

for params in layer\_counts:

model\_params.append(params)

df = pd.DataFrame([model\_params], columns=metrics, index=indexes)

return df

Код архитектуры LeNet:

import torch.nn as nn

import torch.nn.functional as F

class model(nn.Module):

def \_\_init\_\_(self, params, \*args):

super(model, self).\_\_init\_\_()

num\_classes = int(params["num\_classes"])

# Сверточные слои

self.conv1 = nn.Conv2d(3, 32, 5)

self.bn1 = nn.BatchNorm2d(32)

self.conv2 = nn.Conv2d(32, 64, 5)

self.bn2 = nn.BatchNorm2d(64)

self.conv3 = nn.Conv2d(64, 128, 3)

# Полносвязные слои

self.fc1 = nn.Linear(128 \* 3 \* 3, 512)

self.dp1 = nn.Dropout(p=0.25)

self.fc2 = nn.Linear(512, 256)

self.dp2 = nn.Dropout(p=0.25)

self.fc3 = nn.Linear(256, num\_classes)

def forward(self, x):

x = F.relu(self.bn1(self.conv1(x)))

x = F.max\_pool2d(x, 2)

x = F.relu(self.bn2(self.conv2(x)))

x = F.max\_pool2d(x, 2)

x = F.relu(self.conv3(x))

x = x.view(x.size()[0], -1)

x = F.relu(self.fc1(x))

x = self.dp1(x)

x = F.relu(self.fc2(x))

x = self.dp2(x)

x = self.fc3(x)

return x

Код архитектуры ResNet:

from torch import nn

class model(nn.Module):

def \_\_init\_\_(self, params, \*args):

num\_classes = int(params["num\_classes"])

super(model, self).\_\_init\_\_()

self.conv1 = nn.Sequential(

nn.Conv2d(in\_channels=3, out\_channels=64, kernel\_size=3, padding=1),

nn.BatchNorm2d(64),

nn.ReLU(inplace=True)

)

self.conv2 = nn.Sequential(

nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=3, padding=1),

nn.BatchNorm2d(128),

nn.ReLU(inplace=True),

nn.MaxPool2d(2)

)

self.res1 = nn.Sequential(nn.Sequential(

nn.Conv2d(in\_channels=128, out\_channels=128, kernel\_size=3, padding=1),

nn.BatchNorm2d(128),

nn.ReLU(inplace=True)

), nn.Sequential(

nn.Conv2d(in\_channels=128, out\_channels=128, kernel\_size=3, padding=1),

nn.BatchNorm2d(128),

nn.ReLU(inplace=True))

)

self.conv3 = nn.Sequential(

nn.Conv2d(in\_channels=128, out\_channels=256, kernel\_size=3, padding=1),

nn.BatchNorm2d(256),

nn.ReLU(inplace=True),

nn.MaxPool2d(2)

)

self.conv4 = nn.Sequential(

nn.Conv2d(in\_channels=256, out\_channels=512, kernel\_size=3, padding=1),

nn.BatchNorm2d(512),

nn.ReLU(inplace=True),

nn.MaxPool2d(2)

)

self.res2 = nn.Sequential(nn.Sequential(

nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1),

nn.BatchNorm2d(512),

nn.ReLU(inplace=True)

), nn.Sequential(

nn.Conv2d(in\_channels=512, out\_channels=512, kernel\_size=3, padding=1),

nn.BatchNorm2d(512),

nn.ReLU(inplace=True))

)

self.classifier = nn.Sequential(

nn.MaxPool2d(4),

nn.Flatten(),

nn.Linear(512, num\_classes)

)

def forward(self, x):

x = self.conv1(x)

x = self.conv2(x)

x = self.res1(x) + x

x = self.conv3(x)

x = self.conv4(x)

x = self.res2(x) + x

x = self.classifier(x)

return x

Код функции Dataloader:

import torch

import numpy as np

import torchvision.transforms as transforms

import torchvision.datasets as datasets

from torch.utils.data.sampler import SubsetRandomSampler

def dataloader(params):

size = int(params['batch\_size'])

valid\_size = float(params['valid\_size'])

download\_dataset = bool(params['download\_dataset'])

transform = transforms.Compose([transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),

transforms.RandomCrop(32, padding=4, padding\_mode='reflect'),

transforms.RandomHorizontalFlip()])

trainset = datasets.CIFAR10(root='./data', train=True,

download=download\_dataset, transform=transform)

validationset = datasets.CIFAR10(

root='./data', train=True, download=download\_dataset, transform=transform)

testset = datasets.CIFAR10(root='./data', train=False,

download=download\_dataset, transform=transform)

num\_train = len(trainset)

indices = list(range(num\_train))

split = int(np.floor(valid\_size \* num\_train))

train\_idx, valid\_idx = indices[split:], indices[:split]

TRAIN\_SAMPLER = SubsetRandomSampler(train\_idx)

VALID\_SAMPLER = SubsetRandomSampler(valid\_idx)

NUM\_WORKERS = 2

trainloader = torch.utils.data.DataLoader(

trainset, batch\_size=size,

sampler=TRAIN\_SAMPLER, num\_workers=NUM\_WORKERS

)

validloader = torch.utils.data.DataLoader(

validationset, batch\_size=size,

sampler=VALID\_SAMPLER, num\_workers=NUM\_WORKERS

)

testloader = torch.utils.data.DataLoader(

testset, batch\_size=size, num\_workers=NUM\_WORKERS

)

return trainloader, validloader, testloader, len(TRAIN\_SAMPLER), len(VALID\_SAMPLER)

Код функции Train:

import numpy as np

import os

import torch

import time

from tqdm import tqdm

from torch import nn, optim

def train(params, \*args):

net, valid\_loader, train\_loader, train\_len, valid\_len = args

num\_epochs = int(params['num\_epochs'])

model\_name = params['model\_name']

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

net.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(

net.parameters(),

lr=float(params['learning\_rate']),

betas=(float(params['beta1']), float(params['beta2'])),

amsgrad=bool(params['amsgrad']),

)

train\_loss\_list = []

accuracy\_list = []

valid\_loss\_list = []

time\_list = []

valid\_loss\_min = np.Inf

print('\n----------- Start Training -----------')

print(f'Model: {model\_name} \tEpochs: {num\_epochs}')

startTime = time.time()

for epoch in tqdm(range(num\_epochs)):

train\_loss = 0.0

net.train()

for inputs, labels in train\_loader:

inputs, labels = inputs.to(device), labels.to(device)

optimizer.zero\_grad()

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

train\_loss += loss.item() \* inputs.size(0)

correct = 0

valid\_loss = 0.0

net.eval()

if valid\_len != 0:

for inputs, labels in valid\_loader:

with torch.no\_grad():

inputs, labels = inputs.to(device), labels.to(device)

outputs = net(inputs)

loss = criterion(outputs, labels)

valid\_loss += loss.item() \* inputs.size(0)

predicted = torch.max(outputs.data, 1)[1]

correct += (predicted == labels).sum().item()

train\_loss = train\_loss / train\_len

valid\_loss = valid\_loss / valid\_len

accuracy = correct / valid\_len

endTime = time.time()

time\_list.append(endTime - startTime)

train\_loss\_list.append(train\_loss)

accuracy\_list.append(accuracy)

valid\_loss\_list.append(valid\_loss)

print(

f'Epoch: {epoch + 1} \tTraining Loss: {train\_loss:.6f} \t'

f'Validation Loss: {valid\_loss:.6f} \t'

f'Validation Accuracy: {accuracy:.6f}'

)

save\_dir = os.getcwd() + '\\save\_models\\'

file\_name = model\_name + str(num\_epochs) + '.pt'

if not os.path.exists(save\_dir):

os.makedirs(save\_dir)

torch.save(net.state\_dict(), save\_dir + file\_name)

valid\_loss\_min = valid\_loss

return train\_loss\_list, accuracy\_list, valid\_loss\_list, time\_list

Код функции Predict:

import torch

from PIL import Image

import numpy as np

def predict(params, \*args):

net, test\_loader = args

save\_path = params['save\_path']

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

net.to(device)

print('\n----------- Start Predicting -----------')

net.eval()

idx = 1

for inputs, labels in test\_loader:

with torch.no\_grad():

inputs, labels = inputs.to(device), labels.to(device)

outputs = net(inputs)

predicted = torch.max(outputs.data, 1)[1]

print(predicted)

for i, (inp, label) in enumerate(zip(inputs, labels)):

image = inp.cpu().numpy() # добавлено преобразование tensor -> ndarray

image = image.transpose((1, 2, 0)) # Если порядок каналов RGB, эта строка не нужна

image = Image.fromarray((image \* 255).astype(np.uint8))

image.save(save\_path + f'\\image\_{idx}.png')

Код функции Test accuracy:

import torch

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from tqdm import tqdm

def test\_accuracy(params, \*args):

net, testloader = args

num\_epochs = int(params['num\_epochs'])

model\_name = params['model\_name']

correct = 0

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

with torch.no\_grad():

net.eval()

list\_predicted = list()

list\_labels = list()

print('\n----------- Start Testing Accuracy -----------')

print(f'Model: {model\_name} \tEpochs: {num\_epochs}')

for images, labels in tqdm(testloader):

list\_labels.append(labels)

images, labels = images.to(device), labels.to(device)

outputs = net(images)

predict = torch.max(outputs.data, 1)[1]

list\_predicted.append(predict)

correct += (predict == labels).sum().item()

# объединяем списки меток и предсказаний

labels\_flat = torch.cat(list\_labels).tolist()

predicted\_flat = torch.cat(list\_predicted).tolist()

# вычисляем точность, полноту и F1-меру для всего датасета

precision = precision\_score(labels\_flat, predicted\_flat, average='macro')

recall = recall\_score(labels\_flat, predicted\_flat, average='macro')

f1 = f1\_score(labels\_flat, predicted\_flat, average='macro')

# Возвращаем соотношение правильных предсказаний

accuracy = correct / len(testloader.dataset)

print('----------- Finished Testing Accuracy -----------')

return accuracy, precision, recall, f1

Код для запуска LeNet на Python:

from neuron\_node import neuron\_node

params = {

"dataloader\_path": "C:\\files\\class\_dataset\\cifar\_dataset",

"model\_path": "C:\\files\\models\\LeNet2",

"train\_path": "C:\\files\\models\\ResNet",

"test\_accuracy\_path": "C:\\files\\models\\ResNet",

"save\_path": "C:\\files\\statistics\\LeNet2.xlsx",

"pretrained\_model": "C:\\files\\pretrained\_models\\LeNet70.pt",

"model\_name": "LeNet2",

"batch\_size": 64,

"valid\_size": 0.2,

"download\_dataset": False,

"amsgrad": True,

"save\_model": None,

"num\_classes": 10,

"beta1": 0.8,

"beta2": 0.98,

"learning\_rate": 0.01,

"momentum": 0.9,

"num\_epochs": 10

}

if \_\_name\_\_ == "\_\_main\_\_":

# Инициализируем наш класс block с помощью параметров из текстового файла

LeNet = neuron\_node(params)

# Создаем экземпляр модели

model = LeNet.model()

# Получаем загрузчики данных

train\_loader, valid\_loader, test\_loader, train\_len, valid\_len = LeNet.dataloader()

# Обучаем нейросеть

train\_loss\_list, accuracy\_list, time\_list = LeNet.train(model, valid\_loader, train\_loader, train\_len, valid\_len)

# Тестируем обученную нейросеть на тестовых данных

LeNet.train\_stat(train\_loss\_list, accuracy\_list, time\_list, params['save\_path'])

Код для запуска ResNet на Python:

from neuron\_node import neuron\_node

params = {

"dataloader\_path": "C:\\files\\class\_dataset\\cifar\_dataset",

"model\_path": "C:\\files\\models\\ResNet",

"train\_path": "C:\\files\\models\\ResNet",

"test\_accuracy\_path": "C:\\files\\models\\ResNet",

"save\_path": "C:\\files\\statistics\\ResNet.xlsx",

"pretrained\_model": "C:\\neuron-node\\save\_models\\ResNet50.pt",

"model\_name": "ResNet",

"batch\_size": 100,

"valid\_size": 0.2,

"download\_dataset": False,

"amsgrad": True,

"save\_model": None,

"num\_classes": 10,

"beta1": 0.8,

"beta2": 0.98,

"learning\_rate": 0.01,

"momentum": 0.9,

"num\_epochs": 10

}

if \_\_name\_\_ == "\_\_main\_\_":

# Инициализируем наш класс block с помощью параметров из текстового файла

ResNet = neuron\_node(params)

# Создаем экземпляр модели

model = ResNet.model()

# Получаем загрузчики данных

train\_loader, valid\_loader, test\_loader, train\_len, valid\_len = ResNet.dataloader()

# Обучаем нейросеть

train\_loss\_list, accuracy\_list, valid\_loss\_list, time\_list = ResNet.train(model, valid\_loader, train\_loader, train\_len, valid\_len)

# Тестируем обученную нейросеть на тестовых данных

ResNet.train\_stat(train\_loss\_list, accuracy\_list, valid\_loss\_list, time\_list, params['save\_path'])

Код архитектуры FCN:

import torch.nn as nn

class model(nn.Module):

def \_\_init\_\_(self, params, \*args):

super(model, self).\_\_init\_\_()

input\_channels = 3

num\_classes = int(params['num\_classes'])

# Encoder

self.conv0 = self.conv\_block(input\_channels, 64)

self.conv1 = self.conv\_block(64, 128)

self.conv2 = self.conv\_block(128, 256)

self.pool = nn.MaxPool2d(kernel\_size=2, stride=2)

# Decoder

self.deconv0 = self.deconv\_block(256, 128)

self.deconv1 = self.deconv\_block(128, 64)

self.deconv2 = self.deconv\_block(64, num\_classes)

def conv\_block(self, in\_channels, out\_channels):

return nn.Sequential(

nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True),

nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True)

)

def deconv\_block(self, in\_channels, out\_channels):

return nn.Sequential(

nn.ConvTranspose2d(in\_channels, out\_channels, kernel\_size=3, stride=2, padding=1, output\_padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(out\_channels, out\_channels, kernel\_size=3, padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True)

)

def forward(self, x):

# Encoder

x = self.conv0(x)

x = self.pool(x)

x = self.conv1(x)

x = self.pool(x)

x = self.conv2(x)

x = self.pool(x)

# Decoder

x = self.deconv0(x)

x = self.deconv1(x)

x = self.deconv2(x)

return x

Код архитектуры UNet:

import torch.nn as nn

class model(nn.Module):

def \_\_init\_\_(self, params, \*args):

super(model, self).\_\_init\_\_()

input\_channels = 3

num\_classes = int(params['num\_classes'])

# Encoder

self.conv0 = self.conv\_block(input\_channels, 64)

self.conv1 = self.conv\_block(64, 128)

self.conv2 = self.conv\_block(128, 256)

self.pool = nn.MaxPool2d(kernel\_size=2, stride=2)

# Decoder

self.deconv0 = self.deconv\_block(256, 128)

self.deconv1 = self.deconv\_block(128, 64)

self.deconv2 = self.deconv\_block(64, num\_classes)

def conv\_block(self, in\_channels, out\_channels):

return nn.Sequential(

nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True),

nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True)

)

def deconv\_block(self, in\_channels, out\_channels):

return nn.Sequential(

nn.ConvTranspose2d(in\_channels, out\_channels, kernel\_size=3, stride=2, padding=1, output\_padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(out\_channels, out\_channels, kernel\_size=3, padding=1),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True)

)

def forward(self, x):

# Encoder

x = self.conv0(x)

x = self.pool(x)

x = self.conv1(x)

x = self.pool(x)

x = self.conv2(x)

x = self.pool(x)

# Decoder

x = self.deconv0(x)

x = self.deconv1(x)

x = self.deconv2(x)

return x

Код функции Dataloader:

import os

import torch

import numpy as np

from PIL import Image

from sklearn.cluster import KMeans

from torch.utils.data import Dataset, DataLoader

from torchvision import transforms

from tqdm import tqdm

from torch.utils.data.sampler import SubsetRandomSampler

class CityscapeDataset(Dataset):

def \_\_init\_\_(self, image\_dir, label\_model):

self.image\_dir = image\_dir

self.image\_fns = os.listdir(image\_dir)

self.label\_model = label\_model

def \_\_len\_\_(self):

return len(self.image\_fns)

def \_\_getitem\_\_(self, index):

image\_fn = self.image\_fns[index]

image\_fp = os.path.join(self.image\_dir, image\_fn)

image = Image.open(image\_fp).convert('RGB')

image = np.array(image)

cityscape, label = self.split\_image(image)

label\_class = self.label\_model.predict(label.reshape(-1, 3)).reshape(256, 256)

cityscape = self.transform(cityscape)

label\_class = torch.Tensor(label\_class).long()

return cityscape, label\_class

def split\_image(self, image):

image = np.array(image)

cityscape, label = image[:, :256, :], image[:, 256:, :]

return cityscape, label

def transform(self, image):

transform\_ops = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))

])

return transform\_ops(image)

def split\_image(image):

image = np.array(image)

cityscape, label = image[:, :256, :], image[:, 256:, :]

return cityscape, label

"""

Обрабатываем изображения и маски, преобразовывая их в даталоадеры

"""

def dataloader(params):

# Импортируем параметры

size = int(params['batch\_size'])

num\_items = int(params['num\_items'])

num\_classes = int(params['num\_classes'])

valid\_size = float(params['valid\_size'])

data\_dir = params['data\_dir']

# Получаем пути для выборок

train\_dir = os.path.join(data\_dir, "train")

val\_dir = os.path.join(data\_dir, "val")

train\_fns = os.listdir(train\_dir)

val\_fns = os.listdir(val\_dir)

"""Первый образец изображения используется для извлечения меток классов пикселей, которые будут использоваться

для всех остальных изображений в наборе данных. Код разбивает изображение на две части: городскую сцену и метки (

классы пикселей). Затем метки конвертируются в многомерное пространство и используется метод K-means для их

кластеризации и получения количества классов (num\_classes)."""

sample\_image\_fp = os.path.join(train\_dir, train\_fns[0])

sample\_image = Image.open(sample\_image\_fp).convert("RGB")

sample\_image = np.array(sample\_image)

cityscape, label = split\_image(sample\_image)

cityscape, label = Image.fromarray(cityscape), Image.fromarray(label)

color\_set = set()

# Обрабатываем изображения с целью осуществления кластеризации, чтобы получить информацию о том, какие классы

# объектов могут быть на изображении и разделяем визуализацию классов различными цветами

for train\_fn in tqdm(train\_fns[:10]):

train\_fp = os.path.join(train\_dir, train\_fn)

image = np.array(Image.open(train\_fp))

cityscape, label = split\_image(sample\_image)

label = label.reshape(-1, 3)

local\_color\_set = set([tuple(c) for c in list(label)])

color\_set.update(local\_color\_set)

color\_array = np.array(list(color\_set))

# Получаем набор классов и изменяем маски в соответствии с ними

color\_array = np.random.choice(range(256), 3 \* num\_items).reshape(-1, 3)

label\_model = KMeans(n\_clusters=num\_classes)

label\_model.fit(color\_array)

label\_model.predict(color\_array[:5, :])

cityscape, label = split\_image(sample\_image)

# Используем обученную модель для прогнозирования меток классов пикселей для каждого пикселя в `label`,

# основываясь на массиве цветов.

label\_class = label\_model.predict(label.reshape(-1, 3)).reshape(256, 256)

trainset = CityscapeDataset(train\_dir, label\_model)

num\_train = len(trainset)

indices = list(range(num\_train))

split = int(np.floor(valid\_size \* num\_train))

train\_idx, valid\_idx = indices[split:], indices[:split]

#

TRAIN\_SAMPLER = SubsetRandomSampler(train\_idx)

VALID\_SAMPLER = SubsetRandomSampler(valid\_idx)

#

NUM\_WORKERS = 2

trainloader = torch.utils.data.DataLoader(

trainset, batch\_size=size,

sampler=TRAIN\_SAMPLER, num\_workers=NUM\_WORKERS

)

validloader = torch.utils.data.DataLoader(

trainset, batch\_size=size,

sampler=VALID\_SAMPLER, num\_workers=NUM\_WORKERS

)

testloader = torch.utils.data.DataLoader(

trainset, batch\_size=size, num\_workers=NUM\_WORKERS

)

return trainloader, validloader, testloader

Код функции Train:

import time

import torch

import torch.nn as nn

import torch.optim as optim

from tqdm import tqdm

def train(params, \*args):

model, valid\_loader, train\_loader = args

model\_name = params['model\_name']

num\_epochs = int(params['num\_epochs'])

learning\_rate = float(params['learning\_rate'])

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

pretrained\_model = params['pretrained\_model']

if pretrained\_model:

model.load\_state\_dict(torch.load(pretrained\_model))

step\_losses = []

train\_loss\_list = []

time\_list = []

startTime = time.time()

print('\n----------- Start Training -----------')

print(f'Model: {model\_name} \tEpochs: {num\_epochs}')

startTime = time.time()

for epoch in tqdm(range(num\_epochs)):

epoch\_loss = 0

for X, Y in tqdm(train\_loader, total=len(train\_loader), leave=False):

X, Y = X.to(device), Y.to(device)

optimizer.zero\_grad()

Y\_pred = model(X)

loss = criterion(Y\_pred, Y)

loss.backward()

optimizer.step()

epoch\_loss += loss.item()

step\_losses.append(loss.item())

endTime = time.time()

time\_list.append(endTime-startTime)

train\_loss\_list.append(epoch\_loss / len(train\_loader))

file\_name = model\_name + str(epoch) + ".pth"

torch.save(model.state\_dict(), file\_name)

# Сохраняем нашу модель после каждой эпохи

with open("output.txt", "w") as file:

for line in train\_loss\_list:

file.write(str(line))

return train\_loss\_list, time\_list

Код функции Predict:

import numpy as np

import torch

import matplotlib.pyplot as plt

import os

from torchvision import transforms

def iou(pred, target):

intersection = (pred \* target).sum()

union = pred.sum() + target.sum() - intersection

return intersection / union

def predict(params, \*args):

model, test\_loader = args

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model = model.to(device)

image\_count = int(params['image\_count'])

X, Y = next(iter(test\_loader))

X, Y = X.to(device), Y.to(device)

Y\_pred = model(X)

Y\_pred = torch.argmax(Y\_pred, dim=1)

inverse\_transform = transforms.Compose([

transforms.Normalize((-0.485 / 0.229, -0.456 / 0.224, -0.406 / 0.225), (1 / 0.229, 1 / 0.224, 1 / 0.225))

])

fig, axes = plt.subplots()

fig.set\_size\_inches(256 / 100, 256 / 100)

measurer = lambda pred, target: iou((pred > 0).float(), (target > 0).float())

avg\_iou = 0.0

for i in range(image\_count):

landscape = inverse\_transform(X[i]).permute(1, 2, 0).cpu().detach().numpy()

label\_class = Y[i].cpu().detach().numpy()

label\_class\_predicted = Y\_pred[i].cpu().detach().numpy()

iou\_score = measurer(Y\_pred[i].cpu(), Y[i].cpu())

avg\_iou += iou\_score

color = 8

# Создаем маску, соответствующую не `color` пикселям

mask = np.ones\_like(label\_class\_predicted)

mask[label\_class\_predicted == color] = 0

# Заменяем все пиксели, не равные `color`, на белый цвет (255)

landscape[mask == 1] = [255, 255, 255]

# Добавляем дополнительное измерение к `color\_mask`

color\_mask\_3d = np.expand\_dims(mask, axis=2)

# Находим все пиксели, в которых `color\_mask` равно 1, умножая `landscape`

# на `color\_mask\_3d` (получаем результат, где только те пиксели, где `color\_mask`

# равно 1, сохраняют исходный цвет)

color\_preserved = landscape \* color\_mask\_3d

# Получаем бинарную маску из `color\_mask` и выполняем сложение с `color\_preserved`

color\_mask\_3d = color\_mask\_3d > 0

landscape = np.where(color\_mask\_3d, color\_preserved, landscape)

axes[i, 0].imshow(landscape)

axes[i, 0].set\_title("Landscape")

axes[i, 1].imshow(label\_class)

axes[i, 1].set\_title("Label Class")

axes[i, 2].imshow(label\_class\_predicted)

axes[i, 2].set\_title(f"Label Class - Predicted\nImage {i}: IoU Score {iou\_score}")

print(f"Image {i}: IoU Score {iou\_score}")

if not os.path.exists(params['output\_dir']):

os.makedirs(params['output\_dir'])

plt.axis('off')

plt.imshow(landscape)

fig.savefig(os.path.join(params['output\_dir'], f"image\_{i}.png"), dpi=100)

plt.show()

avg\_iou /= image\_count

print(f"Average IoU Score for {image\_count} images: {avg\_iou}")

Код для запуска FCN на Python:

from neuron\_node import neuron\_node

params = {

"dataloader\_path": "C:\\files\\class\_dataset\\segmentation\_dataset",

"model\_path": "C:\\files\\models\\FCN",

"train\_path": "C:\\files\\models\\UNet",

"predict\_path": "C:\\files\\models\\UNet",

"save\_path": "C:\\files\\statistics\\FCN.xlsx",

"data\_dir": "C:\\files\\dataset\\cityscapes\_data",

"pretrained\_model": "C:\\files\\pretrained\_models\\FCN9.pth",

"output\_dir": "C:\\files\\Images",

"model\_name": "FCN",

"image\_count": 100,

"num\_items": 10000,

"batch\_size": 100,

"valid\_size": 0.2,

"save\_model": None,

"num\_classes": 10,

"learning\_rate": 0.01,

"num\_epochs": 0

}

if \_\_name\_\_ == '\_\_main\_\_':

# Инициализируем наш класс block с помощью параметров из текстового файла

UNet = neuron\_node(params)

# Создаем экземпляр модели

model = UNet.model()

# Получаем загрузчики данных

train\_loader, valid\_loader, test\_loader = UNet.dataloader()

# Обучаем нейросеть

train\_loss\_list, time\_list = UNet.train(model, valid\_loader, train\_loader)

# Тестируем обученную нейросеть на тестовых данных

train\_accuracy\_list = []

valid\_accuracy\_list = []

UNet.train\_stat(train\_loss\_list, train\_accuracy\_list, valid\_accuracy\_list, time\_list, params['save\_path'])

UNet.predict(model, test\_loader)

Код для запуска UNet на Python:

from neuron\_node import neuron\_node

params = {

"dataloader\_path": "C:\\files\\class\_dataset\\segmentation\_dataset",

"model\_path": "C:\\files\\models\\UNet",

"train\_path": "C:\\files\\models\\UNet",

"predict\_path": "C:\\files\\models\\UNet",

"save\_path": "C:\\files\\statistics\\UNet.xlsx",

"data\_dir": "C:\\files\\dataset\\cityscapes\_data",

"pretrained\_model": "C:\\neuron-node\\UNet4.pth",

"output\_dir": "C:\\files\\Images",

"model\_name": "UNet",

"image\_count": 2,

"num\_items": 3000,

"batch\_size": 16,

"valid\_size": 0.2,

"save\_model": None,

"num\_classes": 30,

"learning\_rate": 0.01,

"num\_epochs": 0

}

if \_\_name\_\_ == '\_\_main\_\_':

# Инициализируем наш класс block с помощью параметров из текстового файла

UNet = neuron\_node(params)

# Создаем экземпляр модели

model = UNet.model()

# Получаем загрузчики данных

train\_loader, valid\_loader, test\_loader = UNet.dataloader()

# Обучаем нейросеть

train\_loss\_list, time\_list = UNet.train(model, valid\_loader, train\_loader)

# Тестируем обученную нейросеть на тестовых данных

train\_accuracy\_list = []

UNet.train\_stat(train\_loss\_list, train\_accuracy\_list, time\_list, params['save\_path'])

UNet.predict(model, test\_loader)

Код функции Predict:

import torch

import numpy as np

from PIL import Image

def predict(params, \*args):

net, test\_loader = args

save\_path1 = params['save\_path1']

save\_path2 = params['save\_path2']

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

net.to(device)

print('\n----------- Start Predicting -----------')

net.eval()

idx = 1

for inputs, labels in test\_loader:

with torch.no\_grad():

inputs, labels = inputs.to(device), labels.to(device)

outputs = net(inputs)

predicted = torch.max(outputs.data, 1)[1]

print(predicted)

for i, (inp, label) in enumerate(zip(inputs, labels)):

image = inp.cpu().numpy() # добавлено преобразование tensor -> ndarray

image = image.transpose((1, 2, 0)) # Если порядок каналов RGB, эта строка не нужна

image = Image.fromarray((image \* 255).astype(np.uint8))

if predicted[i] == 1:

image.save(save\_path1 + f'\\image\_{idx}.png')

else:

image.save(save\_path2 + f'\\image\_{idx}.png')

idx += 1

Код для запуска ResNet на Python:

from neuron\_node import neuron\_node

Res\_params = {

"dataloader\_path": "C:\\files\\class\_dataset\\transport\_dataset",

"predict\_path": "C:\\files\\models\\ResNet\\",

"model\_path": "C:\\files\\models\\Cars",

"train\_path": "C:\\files\\models\\ResNet",

"test\_accuracy\_path": "C:\\files\\models\\ResNet",

"data\_dir": "C:\\files\\dataset\\traffic",

"save\_path": "C:\\files\\statistics\\ResNet.xlsx",

"pretrained\_model": None,

"transforms": True,

"save\_path1": "C:\\files\\output\\without\_humans",

"save\_path2": "C:\\files\\output\\humans",

"model\_name": "ResNet",

"num\_classes": 2,

"batch\_size": 64,

"valid\_size": 0.2,

"download\_dataset": False,

"amsgrad": True,

"save\_model": None,

"beta1": 0.8,

"beta2": 0.98,

"learning\_rate": 0.01,

"momentum": 0.9,

"num\_epochs": 0

}

U\_params = {

"dataloader\_path": "C:\\files\\class\_dataset\\output",

"model\_path": "C:\\files\\models\\UNet",

"train\_path": "C:\\files\\models\\UNet",

"predict\_path": "C:\\files\\models\\UNet",

"save\_path": "C:\\files\\statistics\\UNet.xlsx",

"data\_dir": "C:\\files\\dataset\\cityscapes\_data",

"pretrained\_model": "C:\\files\\pretrained\_models\\UNet20.pth",

"output\_dir": "C:\\files\\Images",

"model\_name": "UNet",

"image\_count": 64,

"num\_items": 10000,

"batch\_size": 64,

"valid\_size": 0.2,

"save\_model": None,

"num\_classes": 10,

"learning\_rate": 0.01,

"num\_epochs": 0

}

if \_\_name\_\_ == "\_\_main\_\_":

# Инициализируем наш класс block с помощью параметров из текстового файла

ResNet = neuron\_node(Res\_params)

# Создаем экземпляр модели

model = ResNet.model()

# Получаем загрузчики данных

train\_loader, valid\_loader, test\_loader, train\_len, valid\_len = ResNet.dataloader()

# Обучаем нейросеть

train\_loss\_list, accuracy\_list, valid\_loss\_list, time\_list = ResNet.train(model, valid\_loader, train\_loader,

train\_len, valid\_len)

ResNet.predict(model, test\_loader)

# Инициализируем наш класс block с помощью параметров из текстового файла

UNet = neuron\_node(U\_params)

# Создаем экземпляр модели

model = UNet.model()

# Получаем загрузчики данных

train\_loader, valid\_loader, test\_loader = UNet.dataloader()

# Обучаем нейросеть

train\_loss\_list, time\_list = UNet.train(model, valid\_loader, train\_loader)

# Тестируем обученную нейросеть на тестовых данных

UNet.predict(model, test\_loader)