

Расспознавание изображений рака на МРТ снимках головного мозга

Искомая задача состоит в следующем: пусть есть единичное изображение МРТ мозга, программа должна сказать есть ли на этом фото рак. Более формально на каждое фото программа должна выдать вероятность присутствия рака. Это задача относится к классу задач, называемых задачи бинарной классификации.

В машинном обучении для решения задач бинарной классификации используются различные методы, такие например как деревья решений или логистическая регрессия. Так как наши входные данные это изображения это сильно сужает возможность произвольного выбора метода. То есть, лучше всего будет использовать методы наиболее хорошо работающие с изображениями, такие например как метод Виолы - Джонса, Метод направленных градиентов, свёрточные нейронные сети.

Для начала нужно получить данные загрузить их и сделать предобработку.

Предварительно загрузили датасет с сайта kaggle и поместили его в папку Data нашего проекта.

<https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection> (<https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection>)

```
Ввод [1]: import pandas as pd
import os
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import cv2
import imutils
import torch
from torch.utils.data import TensorDataset, DataLoader
from torchsummary import summary
import torchvision.models as models
import torchvision
import PIL
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

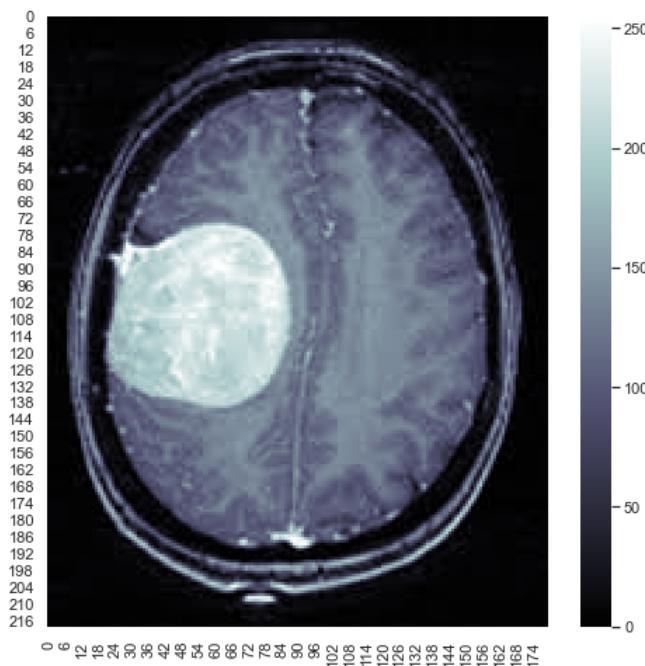
sns.set(rc={'figure.figsize':(8, 8)})
```

Для начала можно загрузить единичное изображение и посмотреть как это выглядит.

```
Ввод [2]: A = cv2.imread('Data/yes/Y1.jpg')
```

```
Ввод [3]: sns.heatmap(A[:, :, 0], cmap='bone')
```

```
Out[3]: <AxesSubplot:>
```



Так как данные мы загрузили локально на диск и их количество их небольшое мы можем смело их сразу все загрузить в оперативную память для работы с ними

```
Ввод [4]: df = []
labels = []
```

Следующие циклы загружают все фотографии из папки. Помимо этого некоторые фотографии имеют три канала цветов, несмотря на то что они монохромные. В таких фотографиях мы оставляем только один произвольный канал, какой именно - не важно так как фотографии монохромные и все три канала одинаковы.

```
Ввод [5]: for path in os.listdir("Data/yes"):
    temp = cv2.imread('Data/yes/'+path)
    if len(temp.shape) == 3:
        temp = temp[:, :, 0]
    df.append(temp)
    labels.append(1)

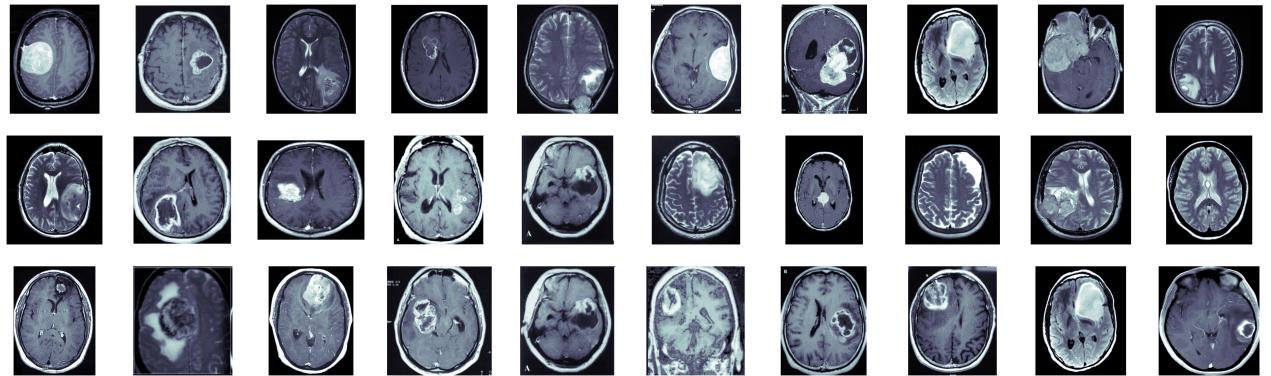
for path in os.listdir("Data/no"):
    temp = cv2.imread('Data/no/'+path)

    if len(temp.shape) == 3:
        temp = temp[:, :, 0]
    df.append(temp)
    labels.append(0)
```

функция для отрисовки большого количества изображений:

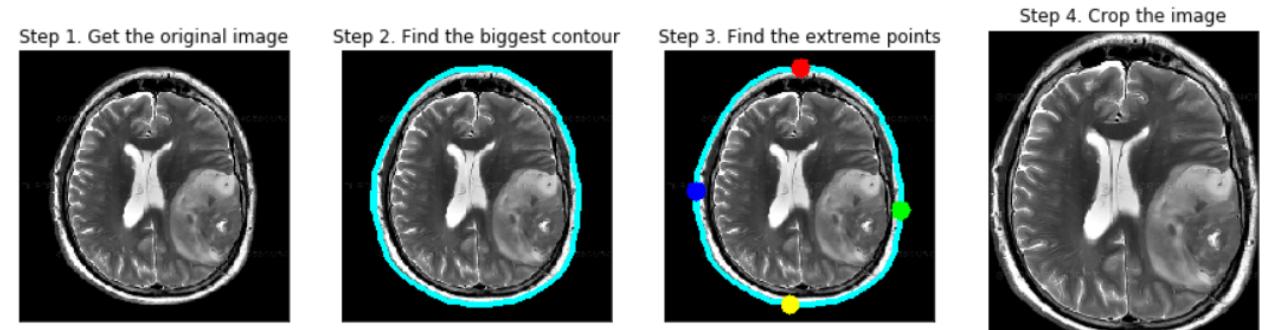
```
Ввод [6]: def show_image(df, n, k):
    plt.figure(figsize=(n*10, k*10))
    for i in range(n*k):
        plt.subplot(k, n, i+1)
        plt.axis("off")
        if (len(df[i].shape) == 3):
            plt.imshow(df[i][:, :, :], cmap="bone")
        else:
            plt.imshow(df[i], cmap="bone")
    plt.show()
```

```
Ввод [7]: show_image(df, 10, 3)
```



Тут сразу видно две проблемы, которые могут помешать алгоритму выдать хороший результат. Во первых фотографии все разного размера, во второй некоторые изображения имеют большую чёрную часть по краям. Небходимо обрезать все фотографии до краёв изображения мозга. Делать это будем используя библиотеки OpenCV и imutils. Подробнее про алгоритм можно прочитать в следующей статье:

<https://pyimagesearch.com/2016/04/11/finding-extreme-points-in-contours-with-opencv/> (<https://pyimagesearch.com/2016/04/11/finding-extreme-points-in-contours-with-opencv/>)



```
Ввод [8]: def crop_imgs(set_name, add_pixels_value=0):
    """
        Finds the extreme points on the image and crops the rectangular out of them
    """
    set_new = []
    for img in set_name:
        #gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        #gray = cv2.GaussianBlur(gray, (5, 5), 0)
        gray = cv2.GaussianBlur(img, (5, 5), 0)

        # threshold the image, then perform a series of erosions +
        # dilations to remove any small regions of noise
        thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)[1]
        thresh = cv2.erode(thresh, None, iterations=2)
        thresh = cv2.dilate(thresh, None, iterations=2)

        # find contours in thresholded image, then grab the largest one
        cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        c = max(cnts, key=cv2.contourArea)

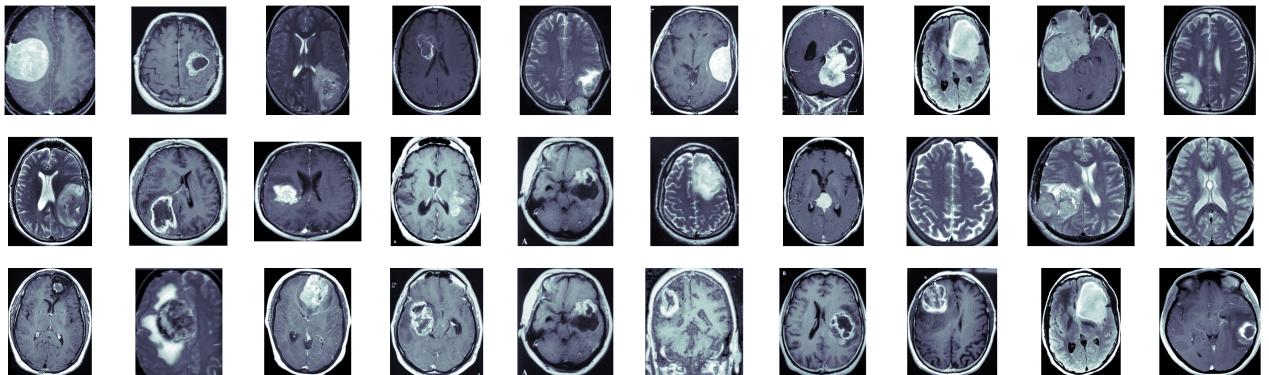
        # find the extreme points
        extLeft = tuple(c[c[:, :, 0].argmin()][0])
        extRight = tuple(c[c[:, :, 0].argmax()][0])
        extTop = tuple(c[c[:, :, 1].argmin()][0])
        extBot = tuple(c[c[:, :, 1].argmax()][0])

        ADD_PIXELS = add_pixels_value
        new_img = img[extTop[1]-ADD_PIXELS:extBot[1]+ADD_PIXELS, extLeft[0]-ADD_PIXELS:extRight[0]+ADD_PIXELS].copy()
        set_new.append(new_img)

    return set_new
```

```
Ввод [9]: df = crop_imgs(df)
```

```
Ввод [10]: show_image(df, 10, 3)
```



Видно что стало лучше, изображения больше похожи друг на друга.

Далее необходимо прибегнуть к особой технике, а именно Data Augmentation. Это способ увеличить размер тренирующей выборки при помощи небольшого видоизменения исходных данных. Необходимо это из за того, что наш датасет очень маленький. Это кстати типичная проблема если решается задача из области медицины.

Да изображений такими изменениями может быть зеркалирование, поворот, сдвиг, обрезка, изменение цвета, аффинное преобразование и т.д.

Мы будем использовать библиотеку PyTorch для нейросетей, она же предоставляет функционал для Data Augmentation.

Более подробно об этом можно прочитать в следующей статье:

[\(https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py\).](https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py)

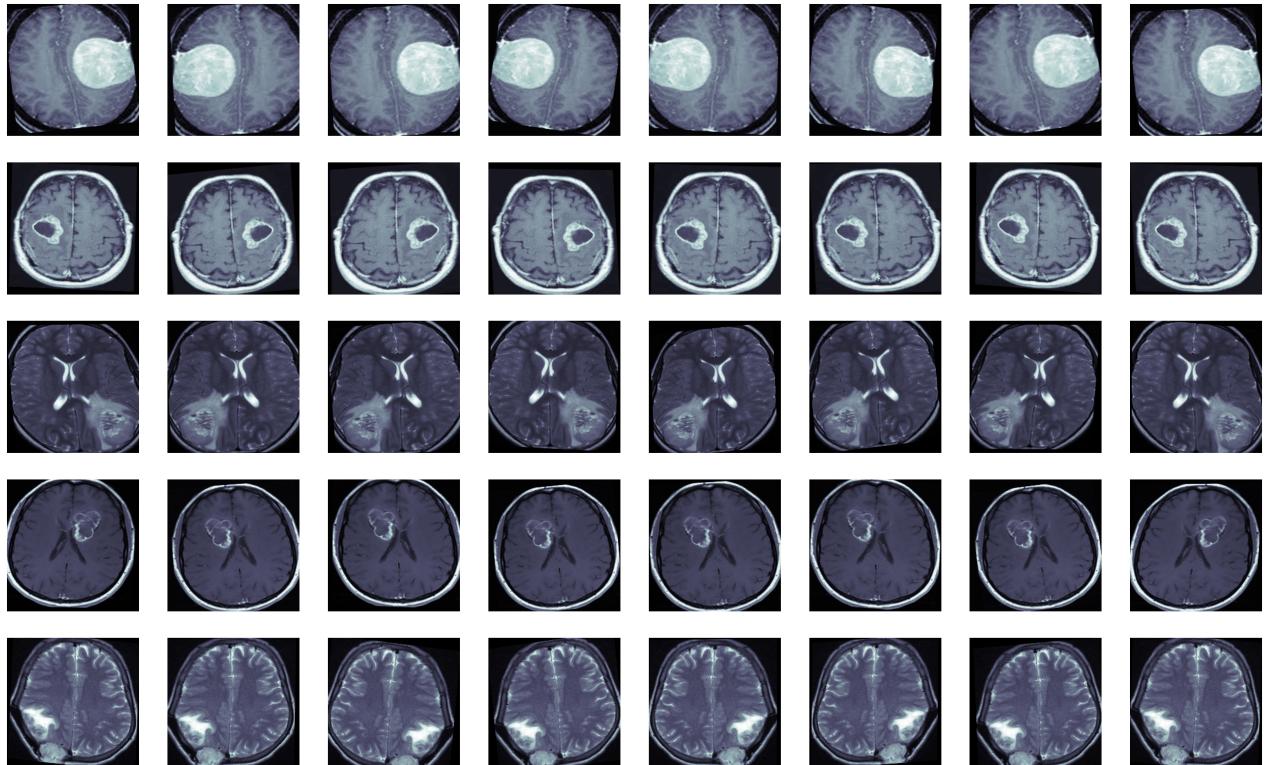
```
Ввод [11]: transforms = torchvision.transforms.Compose([
    # Это композиция преобразований, изображение проходит через каждое по порядку
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Resize((230,230), interpolation=torchvision.transforms.InterpolationMode.BICUBIC),
    torchvision.transforms.CenterCrop((224, 224)), # Обрезка изображения
    torchvision.transforms.RandomHorizontalFlip(0.5), # зеркалирование изображения по горизонтали с вероятностью 0.5
    # Особенно актуально для симметричного мозга
    torchvision.transforms.RandomAffine(7, (0.03,0.03), (0.93, 1), (-3,3,-3,3),
        interpolation=torchvision.transforms.InterpolationMode.BILINEAR),
    # Случайное Афинное преобразование:
    # Повернуть в случайную сторону не более чем на 7 градусов
    # сместить по вертикали и горизонтали не более чем на 3 %
    # Уменьшить на не более чем в 0.93 раз
    # Сдвинуть изображение не более чем на 3 по каждой из осей
    torchvision.transforms.Normalize([0],[1])
])
```

Посмотрим какие преобразования получаются:

```
Ввод [12]: n = 8
k = 5

plt.figure(figsize=(n*10, k*10))
for i in range(n * k):
    plt.subplot(k, n, i+1)
    plt.axis("off")
    plt.imshow(transforms(df[i//n]).numpy().reshape((224, 224)), cmap="bone")

plt.show()
```



Так как изображения всё равно очень похожи друг на друга, необходимо не допустить того, что бы разные варианты одного изображения оказались одновременно и в тренировочной и тестовой выборке. Поэтому для начала разделим датасет на тренировочную и тестовую выборку.

```
Ввод [13]: type(df[0])
```

```
Out[13]: numpy.ndarray
```

```
Ввод [14]: train_dsX, test_dsX, train_dsy, test_dsy = train_test_split(df, labels, stratify=labels, test_size=0.20)
```

Теперь мы можем увеличить наш датасет, количество копий исходных изображений я взял 5. Это немного, так сделано потому что получаемые изображения всё равно сильно похожи друг на друга.

```
Ввод [15]: k = 10 #Сколько копий одного изображения будем делать
train_X = []
train_y = []

for i in range(len(train_dsX)):
    for j in range(k):
        temp = transforms(train_dsX[i]).numpy()[0,:,:,:]
        temp = np.stack((temp, temp, temp))
        train_X.append(temp)
        train_y.append(train_dsy[i])
train_X = np.array(train_X)
train_y = np.array(train_y, dtype='int')

#k = 1
test_X = []
test_y = []

for i in range(len(test_dsX)):
    for j in range(k):
        temp = transforms(test_dsX[i]).numpy()[0,:,:,:]
        temp = np.stack((temp, temp, temp))
        test_X.append(temp)
        test_y.append(test_dsy[i])
test_X = np.array(test_X)
test_y = np.array(test_y, dtype='int')
```

Теперь необходимо провести нормализацию для того что бы алгоритмы лучше работали. Так как torchvision.transforms.Normalize заставить работать не удалось, будем использовать sklearn и его StandardScaler

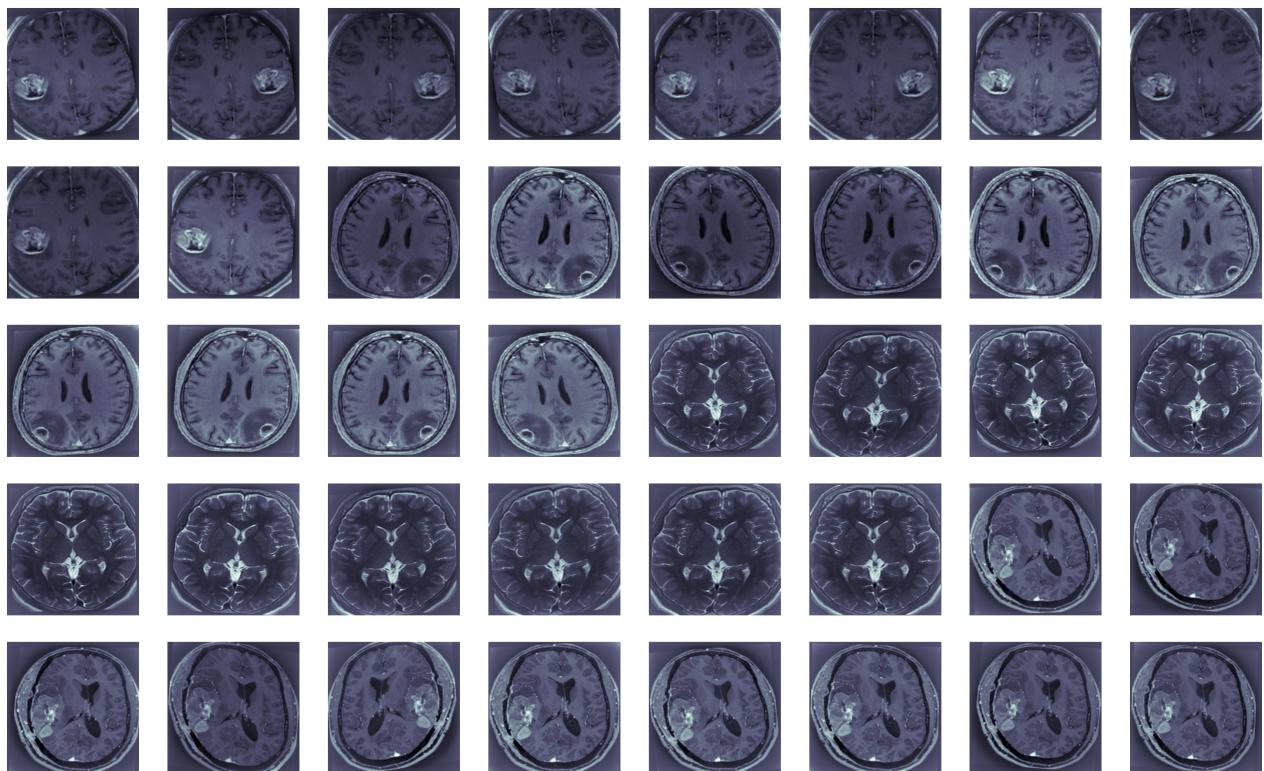
```
Ввод [16]: scalar = StandardScaler()
scalar.fit(np.vstack((train_X, test_X)).reshape(-1, 3*224*224))
train_X = scalar.transform(train_X.reshape(-1, 3*224*224))
test_X = scalar.transform(test_X.reshape(-1, 3*224*224))

train_X = train_X.reshape(-1, 3, 224, 224)
test_X = test_X.reshape(-1, 3, 224, 224)
```

```
Ввод [17]: print(train_X.shape)
print(train_y.shape)
print(test_X.shape)
print(test_y.shape)
```

(2020, 3, 224, 224)
(2020,)
(510, 3, 224, 224)
(510,)

```
Ввод [18]: show_image(train_X, 8, 5)
```



Теперь перейдём к основной части, необходимо создать алгоритм распознающий целевые изображения. Мы будем использовать

свёрточную нейронную сеть, так как начиная с того момента как в 2012 году AlexNet победил на конкурсе ImageNet, нейронные сети теперь это самые эффективные модели компьютерного зрения.

Так же будем использовать технику под названием Transfer Learning. Это техника которая позволяет существенно сократить затраты и время создания модели. Суть в том что мы возьмём уже обученную ранее сеть для другой задачи, далее немного видоизменим её и дообучим.

Мы возьмём сеть ResNet18 которая создавалась для конкурса ImageNet, заменим у неё последний слой, остальные слои заморозим чтобы они не менялись и обучим только последний слой.

```
Ввод [19]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

```
Out[19]: device(type='cuda')
```

Для загрузки сети используем torchvision

```
Ввод [20]: model = models.resnet18(weights=torchvision.models.ResNet18_Weights.IMGNET1K_V1)
```

Теперь можем посмотреть структуру сети

Ввод [21]: model

```

Out[21]: ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)

```

А так же можем посмотреть количество обучаемых параметров

```
Ввод [22]: summary(model.cuda(), (3, 224, 224))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 128, 28, 28]	73,728
BatchNorm2d-20	[-1, 128, 28, 28]	256
ReLU-21	[-1, 128, 28, 28]	0
Conv2d-22	[-1, 128, 28, 28]	147,456
BatchNorm2d-23	[-1, 128, 28, 28]	256
Conv2d-24	[-1, 128, 28, 28]	8,192
BatchNorm2d-25	[-1, 128, 28, 28]	256
ReLU-26	[-1, 128, 28, 28]	0
BasicBlock-27	[-1, 128, 28, 28]	0
Conv2d-28	[-1, 128, 28, 28]	147,456
BatchNorm2d-29	[-1, 128, 28, 28]	256
ReLU-30	[-1, 128, 28, 28]	0
Conv2d-31	[-1, 128, 28, 28]	147,456
BatchNorm2d-32	[-1, 128, 28, 28]	256
ReLU-33	[-1, 128, 28, 28]	0
BasicBlock-34	[-1, 128, 28, 28]	0
Conv2d-35	[-1, 256, 14, 14]	294,912
BatchNorm2d-36	[-1, 256, 14, 14]	512
ReLU-37	[-1, 256, 14, 14]	0
Conv2d-38	[-1, 256, 14, 14]	589,824
BatchNorm2d-39	[-1, 256, 14, 14]	512
Conv2d-40	[-1, 256, 14, 14]	32,768
BatchNorm2d-41	[-1, 256, 14, 14]	512
ReLU-42	[-1, 256, 14, 14]	0
BasicBlock-43	[-1, 256, 14, 14]	0
Conv2d-44	[-1, 256, 14, 14]	589,824
BatchNorm2d-45	[-1, 256, 14, 14]	512
ReLU-46	[-1, 256, 14, 14]	0
Conv2d-47	[-1, 256, 14, 14]	589,824
BatchNorm2d-48	[-1, 256, 14, 14]	512
ReLU-49	[-1, 256, 14, 14]	0
BasicBlock-50	[-1, 256, 14, 14]	0
Conv2d-51	[-1, 512, 7, 7]	1,179,648
BatchNorm2d-52	[-1, 512, 7, 7]	1,024
ReLU-53	[-1, 512, 7, 7]	0
Conv2d-54	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-55	[-1, 512, 7, 7]	1,024
Conv2d-56	[-1, 512, 7, 7]	131,072
BatchNorm2d-57	[-1, 512, 7, 7]	1,024
ReLU-58	[-1, 512, 7, 7]	0
BasicBlock-59	[-1, 512, 7, 7]	0
Conv2d-60	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-61	[-1, 512, 7, 7]	1,024
ReLU-62	[-1, 512, 7, 7]	0
Conv2d-63	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-64	[-1, 512, 7, 7]	1,024
ReLU-65	[-1, 512, 7, 7]	0
BasicBlock-66	[-1, 512, 7, 7]	0
AdaptiveAvgPool2d-67	[-1, 512, 1, 1]	0
Linear-68	[-1, 1000]	513,000
<hr/>		
Total params: 11,689,512		
Trainable params: 11,689,512		
Non-trainable params: 0		
<hr/>		
Input size (MB): 0.57		
Forward/backward pass size (MB): 62.79		
Params size (MB): 44.59		
Estimated Total Size (MB): 107.96		
<hr/>		

Видно что модель имеет более 11 миллионов обучаемых параметров, если мы будем их все обучать модель неизбежно переобучится, кроме того обучаться это будет очень долго. Поэтому для начала заморозим обучение всех существующих слоёв:

```
Ввод [23]: for param in model.parameters():
    param.requires_grad = False
```

Теперь заменим последний слой, на два слоя, так что бы выходных нейронов было всего 2

```
Ввод [24]: model.fc = torch.nn.Sequential(
    torch.nn.Dropout(p=0.5),
    torch.nn.Linear(512, 128),
    torch.nn.ReLU(inplace=True),
    torch.nn.Dropout(p=0.5),
    torch.nn.Linear(128, 2))
```

```
Ввод [25]: model = model.to(device)
```

Посмотрим на количество обучаемых параметров:

```
Ввод [26]: summary(model, (3, 224, 224))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 128, 28, 28]	73,728
BatchNorm2d-20	[-1, 128, 28, 28]	256
ReLU-21	[-1, 128, 28, 28]	0
Conv2d-22	[-1, 128, 28, 28]	147,456
BatchNorm2d-23	[-1, 128, 28, 28]	256
Conv2d-24	[-1, 128, 28, 28]	8,192
BatchNorm2d-25	[-1, 128, 28, 28]	256
ReLU-26	[-1, 128, 28, 28]	0
BasicBlock-27	[-1, 128, 28, 28]	0
Conv2d-28	[-1, 128, 28, 28]	147,456
BatchNorm2d-29	[-1, 128, 28, 28]	256
ReLU-30	[-1, 128, 28, 28]	0
Conv2d-31	[-1, 128, 28, 28]	147,456
BatchNorm2d-32	[-1, 128, 28, 28]	256
ReLU-33	[-1, 128, 28, 28]	0
BasicBlock-34	[-1, 128, 28, 28]	0
Conv2d-35	[-1, 256, 14, 14]	294,912
BatchNorm2d-36	[-1, 256, 14, 14]	512
ReLU-37	[-1, 256, 14, 14]	0
Conv2d-38	[-1, 256, 14, 14]	589,824
BatchNorm2d-39	[-1, 256, 14, 14]	512
Conv2d-40	[-1, 256, 14, 14]	32,768
BatchNorm2d-41	[-1, 256, 14, 14]	512
ReLU-42	[-1, 256, 14, 14]	0
BasicBlock-43	[-1, 256, 14, 14]	0
Conv2d-44	[-1, 256, 14, 14]	589,824
BatchNorm2d-45	[-1, 256, 14, 14]	512
ReLU-46	[-1, 256, 14, 14]	0
Conv2d-47	[-1, 256, 14, 14]	589,824
BatchNorm2d-48	[-1, 256, 14, 14]	512
ReLU-49	[-1, 256, 14, 14]	0
BasicBlock-50	[-1, 256, 14, 14]	0
Conv2d-51	[-1, 512, 7, 7]	1,179,648
BatchNorm2d-52	[-1, 512, 7, 7]	1,024
ReLU-53	[-1, 512, 7, 7]	0
Conv2d-54	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-55	[-1, 512, 7, 7]	1,024
Conv2d-56	[-1, 512, 7, 7]	131,072
BatchNorm2d-57	[-1, 512, 7, 7]	1,024
ReLU-58	[-1, 512, 7, 7]	0
BasicBlock-59	[-1, 512, 7, 7]	0
Conv2d-60	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-61	[-1, 512, 7, 7]	1,024
ReLU-62	[-1, 512, 7, 7]	0
Conv2d-63	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-64	[-1, 512, 7, 7]	1,024
ReLU-65	[-1, 512, 7, 7]	0
BasicBlock-66	[-1, 512, 7, 7]	0
AdaptiveAvgPool2d-67	[-1, 512, 1, 1]	0
Dropout-68	[-1, 512]	0
Linear-69	[-1, 128]	65,664
ReLU-70	[-1, 128]	0
Dropout-71	[-1, 128]	0
Linear-72	[-1, 2]	258

Total params: 11,242,434

Trainable params: 65,922

Non-trainable params: 11,176,512

Input size (MB): 0.57

Forward/backward pass size (MB): 62.79

Params size (MB): 42.89

Estimated Total Size (MB): 106.25

Теперь параметров значительно меньше.

Теперь зададим параметры обучения, а именно перекрёстную энтропию как функция потерь. И Adam алгоритм как оптимизатор.

```
Ввод [27]: criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.00005)
```

Для работы в pytorch есть удобное решение "под ключ" для подачи данных во время обучения, это особый класс DataLoader очень удобный в использовании.

```
Ввод [28]: train_dataset = TensorDataset(torch.Tensor(train_X), torch.Tensor(train_y))
test_dataset = TensorDataset(torch.Tensor(test_X), torch.Tensor(test_y))

bt = 2
train_dataloader = DataLoader(train_dataset, batch_size = bt, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size = bt, shuffle=True)

loaders = {"train": train_dataloader, "test": test_dataloader}
```

```
Ввод [29]: accuracy = {"train": [], "test": []}
```

Далее следует основной цикл обучения, этот блок кода можно запускать несколько раз для полного дообучения модели

```
Ввод [30]: max_epochs = 30
for epoch in range(max_epochs):
    for k, dataloader in loaders.items():
        epoch_correct = 0
        epoch_all = 0
        for x_batch, y_batch in dataloader:
            x_batch = x_batch.to(device)
            y_batch = y_batch.to(device)
            if k == "train":
                model.train() # <----- переводим модель в режим train
                optimizer.zero_grad() # <----- обнуляем градиенты модели
                #print(x_batch)
                outp = model(x_batch)
            else:
                model.eval() # <----- переводим модель в режим eval
                with torch.no_grad(): # <----- НЕ считаем градиенты
                    outp = model(x_batch) # <----- получаем "логиты" из модели
            preds = outp.argmax(1)
            correct = (preds == y_batch).type(torch.long).sum()
            all = bt
            epoch_correct += correct.item()
            epoch_all += all
            if k == "train":
                y_batch=y_batch.to(torch.int64)
                loss = criterion(outp, y_batch)
                loss.backward() # <----- считаем градиенты
                optimizer.step() # <----- делаем шаг градиентного спуска

            if k == "train":
                print(f"Epoch: {epoch+1}")
                print(f"Loader: {k}. Accuracy: {epoch_correct/epoch_all}")
                accuracy[k].append(epoch_correct/epoch_all)
plt.plot(range(len(accuracy["train"])), accuracy["train"])
plt.plot(range(len(accuracy["train"])), accuracy["test"])
```

Epoch: 1
 Loader: train. Accuracy: 0.5900990099009901
 Loader: test. Accuracy: 0.7019607843137254
 Epoch: 2
 Loader: train. Accuracy: 0.6188118811881188
 Loader: test. Accuracy: 0.6372549019607843
 Epoch: 3
 Loader: train. Accuracy: 0.6460396039603961
 Loader: test. Accuracy: 0.6980392156862745
 Epoch: 4
 Loader: train. Accuracy: 0.6767326732673268
 Loader: test. Accuracy: 0.8019607843137255
 Epoch: 5
 Loader: train. Accuracy: 0.6688118811881189
 Loader: test. Accuracy: 0.6764705882352942
 Epoch: 6
 Loader: train. Accuracy: 0.6737623762376238
 Loader: test. Accuracy: 0.7980392156862746
 Epoch: 7
 Loader: train. Accuracy: 0.6643564356435644
 Loader: test. Accuracy: 0.7686274509803922
 Epoch: 8
 Loader: train. Accuracy: 0.6767326732673268
 Loader: test. Accuracy: 0.7294117647058823
 Epoch: 9
 Loader: train. Accuracy: 0.6787128712871288
 Loader: test. Accuracy: 0.7058823529411765
 Epoch: 10
 Loader: train. Accuracy: 0.691089108910891
 Loader: test. Accuracy: 0.7431372549019608
 Epoch: 11
 Loader: train. Accuracy: 0.7128712871287128
 Loader: test. Accuracy: 0.7725490196078432
 Epoch: 12
 Loader: train. Accuracy: 0.7029702970297029
 Loader: test. Accuracy: 0.7686274509803922
 Epoch: 13
 Loader: train. Accuracy: 0.6915841584158415
 Loader: test. Accuracy: 0.7235294117647059
 Epoch: 14
 Loader: train. Accuracy: 0.6886138613861386
 Loader: test. Accuracy: 0.7901960784313725
 Epoch: 15
 Loader: train. Accuracy: 0.6851485148514852
 Loader: test. Accuracy: 0.7764705882352941
 Epoch: 16
 Loader: train. Accuracy: 0.7049504950495049
 Loader: test. Accuracy: 0.7647058823529411
 Epoch: 17
 Loader: train. Accuracy: 0.7183168316831683
 Loader: test. Accuracy: 0.8
 Epoch: 18

```
Loader: train. Accuracy: 0.6955445544554455
Loader: test. Accuracy: 0.8156862745098039
Epoch: 19
Loader: train. Accuracy: 0.7044554455445544
Loader: test. Accuracy: 0.7901960784313725
Epoch: 20
Loader: train. Accuracy: 0.7123762376237623
Loader: test. Accuracy: 0.7607843137254902
Epoch: 21
Loader: train. Accuracy: 0.7074257425742574
Loader: test. Accuracy: 0.803921568627451
Epoch: 22
Loader: train. Accuracy: 0.7153465346534653
Loader: test. Accuracy: 0.796078431372549
Epoch: 23
Loader: train. Accuracy: 0.7133663366336633
Loader: test. Accuracy: 0.792156862745098
Epoch: 24
Loader: train. Accuracy: 0.7103960396039604
Loader: test. Accuracy: 0.7941176470588235
Epoch: 25
Loader: train. Accuracy: 0.7103960396039604
Loader: test. Accuracy: 0.7490196078431373
Epoch: 26
Loader: train. Accuracy: 0.7198019801980198
Loader: test. Accuracy: 0.8156862745098039
Epoch: 27
Loader: train. Accuracy: 0.7163366336633663
Loader: test. Accuracy: 0.803921568627451
Epoch: 28
Loader: train. Accuracy: 0.7128712871287128
Loader: test. Accuracy: 0.8235294117647058
Epoch: 29
Loader: train. Accuracy: 0.7113861386138614
Loader: test. Accuracy: 0.7980392156862746
Epoch: 30
Loader: train. Accuracy: 0.696039603960396
Loader: test. Accuracy: 0.7549019607843137
```

Out[30]: [`<matplotlib.lines.Line2D at 0x24bee4f1300>`]



Выше график после циклов 30 обучения. Полученный результат около 80% точности на тестовой выборке.

Ввод []: