Обучение на кропах

Идея обучения архитектуры трансформер на кропах людей заключается в том, чтобы подтвердить гипотезу о том, что модель способна работать и обучаться понимать поведение объекта на заданной последовательности кадров.

Цель - обучить классификатор на архитектуре трансформер, в который подается последовательный набор кропов. На выходе модель должна выдавать результат от 0 до 1, где 1 - человек, а 0 - фон.

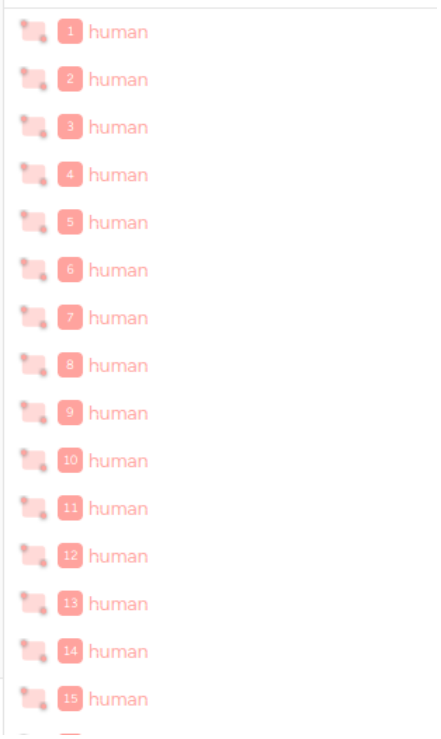
Подготовка датасета

Разметка видео

Обучающий и валидационные датасеты будут собираться из кадров новых видео, у которых заранее были убраны верные края:

- window_vid_1
- window_vid_2
- window_vid_4

Разметка происходит в программе label-studio.
Для создания аннотации создадим проект, в который добавим наши видео.
(**Важно** заранее сконвертировать видео в формат .webm, так как браузер в котором, запускается label-studio не поддерживает .mp4.) Сам процесс разметки достаточно прост, но хочется подчеркнуть 1 момент, что все боксы представлены в виде объектов, которые их олицетворяют на кадрах:



Иными словами, каждый бокс проиндексирован.

Вся аннотация может экспортироваться в разные форматы, по типу json, csv, txt. Мной был выбрал формат json, так как помимо разметки, предоставляется мета информация о наборе данных. Подробнее об этом формате:

Label-studio json format

Экспортируемый файл хранить аннотацию по всему проекту в label-studio. То есть, если в проекте разметки 3 видео, то в файле будет хранится список, с 3мя словарями, которые хранят разметку + мету для соответствующего видео.

Файл имеет следующую структуру:

```
[
        {
                "data": 'path/to/vid'
```

```
        "annotations": [
                {
                        "id": str
                        "result": [  # <--- objects
                                {
                                        "id": str
                                        "value": {
                                                "labels": ["human"],
                                                "sequence": [
                                                        {'frame': 5389,
                                                        'enabled': True,
                                                        'rotation': 0,
                                                        'x': 24.944196428571335,
                                                        'y': 75.5952380952378,
                                                        'width': 4.575892857142935,
                                                        'height': 9.226190476190709,
                                                        'time': 215.56},
                                                        .
                                                        .
                                                        .
                                                ]
                                        }
                                }
                                .
                                .
                                .
                        ]

                }
                ...
        ]
    }
]
```

Здесь приведены параметры, которые нас интересуют.
Вся информация о аннотации хранится в ключе **annotations**. В нем есть **results**, который является списком объектов. Каждый объект хранит такие параметры: номер кадра, поворот, координаты, время в видео.

Количество аннотаций будет зависеть от количества видео в проекте разметки.

Аннотация объекта

Аннотация объекта ограничивающей рамкой представляется в виде последовательности **изменения** этой рамки. То есть json хранит не по кадровую аннотацию а только координаты и размер в том фрейме, где они изменились. Поэтому необходимо проинтерполировать координаты ограничивающей рамки объекта относительно каждого кадра, в котором присутствует этот объект.

Координаты рамки выглядят следующим образом:

- x - значение верхнего левого угла рамки по ширине
- y - значение верхнего левого угла рамки по высоте
- width - ширина рамки
- height - высота рамки

Все значения представлены в % соотношении. То есть, чтобы получить реальные координаты нужно домножить их на размер изображения и поделить на 100

```
x * img_width / 100
y * img_height / 100
width * img_width / 100
height * img_height / 100
```

Конвертация и преобразование

Конвертация аннотации в ieos формат

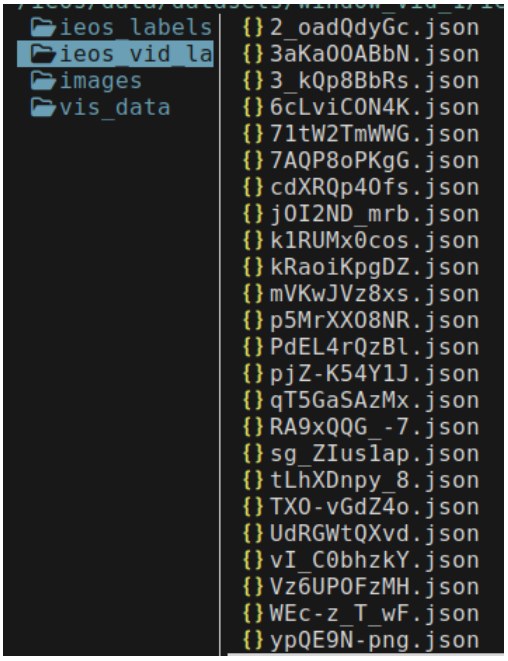
Как было написано выше, аннотация объекта хранит координаты только в том фрейме, в котором было ручное перемещение бокса. Например мы имеем информацию о двух кадрах:

```
{
        'frame': 5389,
        'enabled': True,
        'rotation': 0,
        'x': 24.944196428571335,
        'y': 75.5952380952378,
        'width': 4.575892857142935,
        'height': 9.226190476190709,
        'time': 215.56
        },
{
        'x': 25.83705357142846,
        'y': 75.29761904761875,
        'width': 4.575892857142935,
        'height': 9.226190476190709,
        'rotation': 0,
        'frame': 5399,
        'enabled': True,
        'time': 215.96
        },
```

Координаты в промежуточных кадрах нам не известны, но мы можем получить их линейно проинтерполировав координаты. После я могу преобразовать данные о кадрах в классический вариант ieos_format, который хранит данные о каждом кадре и также аннотацию объектов, где разметка хранится уже об отдельном объекте в виде последовательности (ieos_vid_labels).

ieos_vid_labels

Аннотация каждого объекта хранится в файле json.



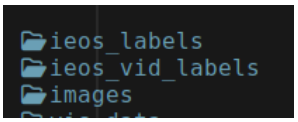
Каждый json имеет следующую структуру:

```
{
        id:str,
        label:str,
        frames:list,
        coords:list
}
```

- id - айдишник объекта. Берется из аннотации label-studio
- label - класс объекта
- frames - список номеров кадров, в которых есть этот объект
- coords - список координат объекта в кадре в формате [top_x, top_y, bot_x, bot_y]. Длина списка coords равна списку frames.

Структура датасета

После конвертации датасет выглядит следующим образом:



где images - это директория, которая хранит в себе кадры изображения полученные после раскадровки.

Таким образом конвертация аннотации и создания датасета была сделана для всех 3ех видео, с помощью написанного скрипта, который в дальнейшем который делает этот процесс автоматическим.

Создание обучающих и валидационных данных

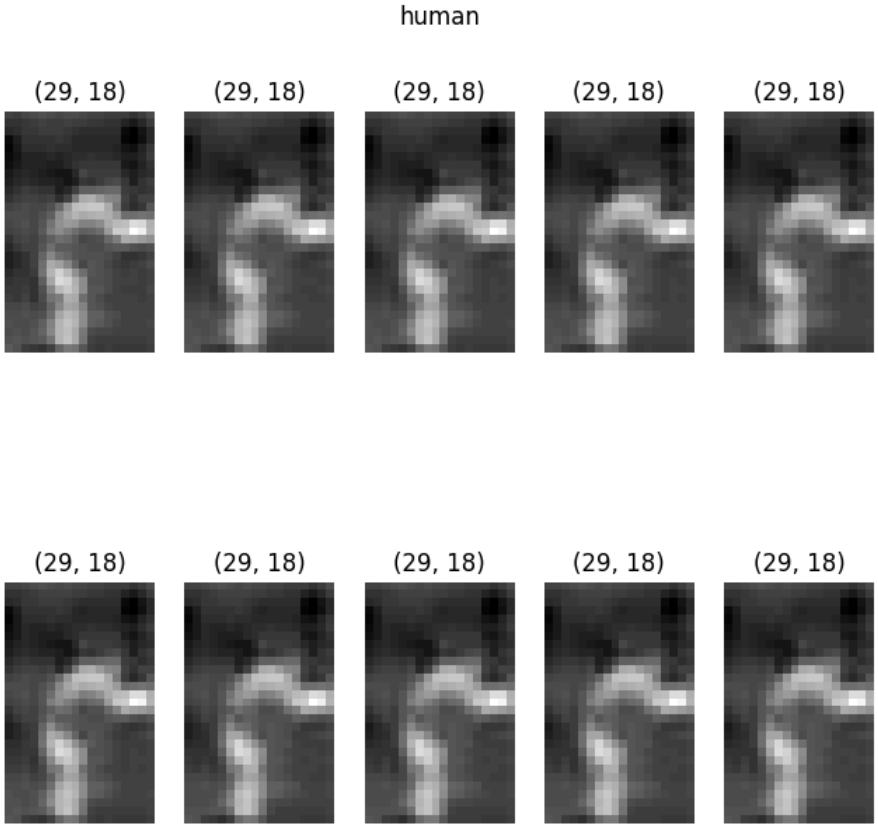
Несмотря на то, что мы подготовили изображения и аннотацию, мы не можем обучать на таких данных модель. Во-первых мы хотим подавать, сразу несколько кадров (а именно 10). Во-вторых мы подаем кропы людей/фона, а не целые изображения. В третьих, Текущая цель это классификация, которая предсказывает класс кропа.
Поэтому нужно сделать кропы объектов из кадров, составить последовательность и только после этого уже обучать модель

Генерация сэмплов людей

Генерация сэмплов людей достаточно проста:

1. берем аннотацию объекта, в которой указаны номера кадром и боксы объекта
2. кропаем по координатам, тем самым получаем последовательность кропов
3. разбиваем последовательность на равные части (чанки) равным 10 (интерпретируем это как поток 10 кадров).

Пример сэмпла человека:

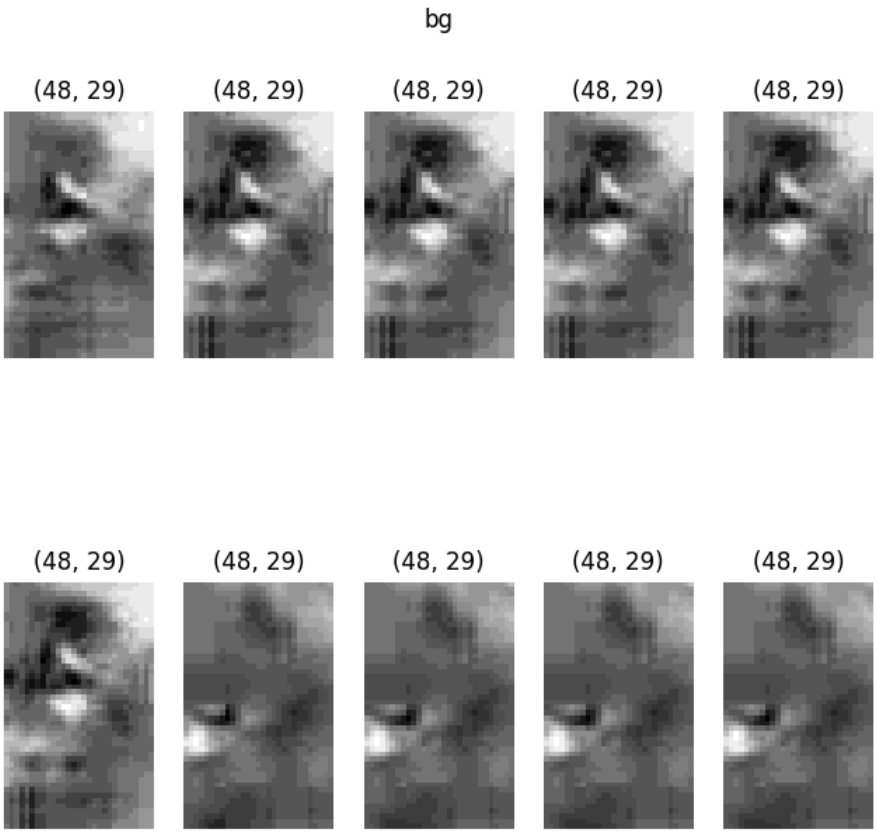


На изображении показано 10 изображений размером (29, 18).

Генерация сэмплов фона

Так как размеченного фона у нас нет, то я решил его просто сгенерировать из уже имеющихся кадров. Каким образом? У меня есть по кадровая аннотация видео. Из случайного кадра я выбираю случайные координаты фона (генерируются размеры вертикального прямоугольника размеры которого варьируются от 15 до 50 px) и кропаю на 10 изображений.

Разумеется тут может быть попадание на человека. Именно поэтому мы работаем с по кадровой разметкой и проверяем координаты фона с координатами людей в кадре.
На рисунке ниже представлен пример сэмпла фона.



Сохранение сэмплов

После генерации сэмпла мы получаем список из 10 изображений:
`[crop1, crop2, crop3, ... crop10]` . Эти сэмплы вместе с меткой сохраняются в соответствующую директорию класса в формате pickle. То есть `(sample, label) -> vid_1_0.pkl`.

Таким образом из 3ех датасетов, было сгенерировано 4153 сэмпла людей и 4153 сэмпла фона.

Чистка сэмплов

Так же были удалены плохие сэмплы, которые включали в себя плохую аннотацию:

- точки
- линии
- сэмплы имеющие очень маленькую ширину/длину относительно противоположной стороны

Сплит на обучающую и тестовую выборки

После генерации сэмплов осталось разбить данные на обучающие и тестовые. Я не стал создавать новые директории и копировать туда сэмплы. Вместо этого, я сделал сплит путей к сэмплам и сохранил эти пути в файлы json
Пример train.json:

```
[
"data/human_data/window_vid_4_842.pickle",
"data/human_data/window_vid_2_47.pickle",
"data/human_data/window_vid_2_1771.pickle",
"data/human_data/window_vid_2_1478.pickle",
"data/human_data/window_vid_2_905.pickle",
"data/human_data/window_vid_4_33.pickle",
"data/human_data/window_vid_4_1146.pickle",
"data/human_data/window_vid_2_588.pickle",
"data/human_data/window_vid_4_2135.pickle",
```

```
"data/human_data/window_vid_4_456.pickle",
"data/human_data/window_vid_4_1832.pickle",
"data/human_data/window_vid_4_345.pickle",
"data/human_data/window_vid_4_97.pickle",
]
```

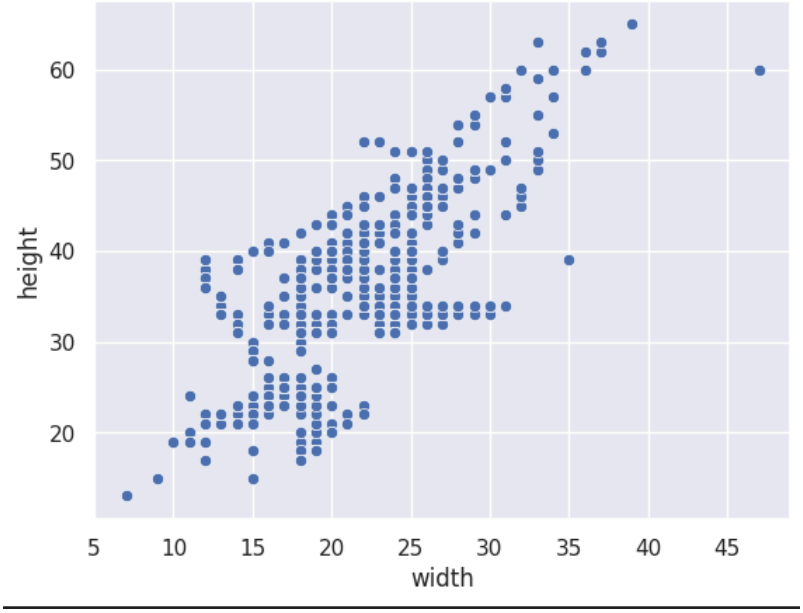
Таким образом я поделил train-val в соотношении 1/3.

Dataloader

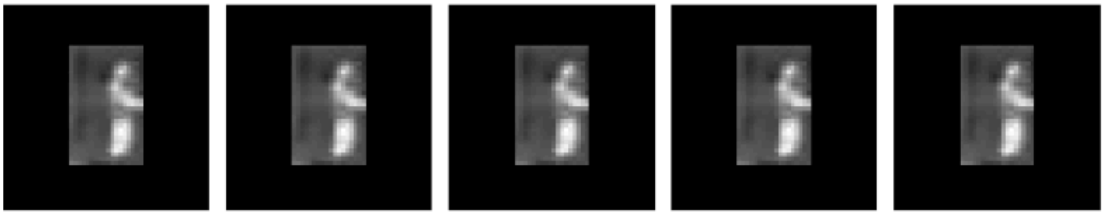
Теперь, когда данные для обучения готовы, осталось написать загрузчик, который будет загружать данные, делать паддинг кропов конвертировать изображения в тензор.

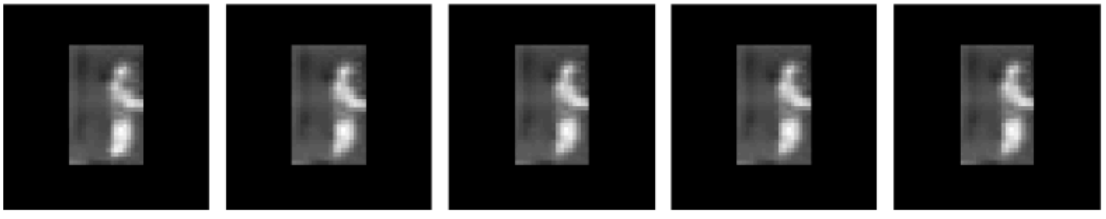
Паддинг

В целях упрощения подачи данных в модель мы отказались делить изображения на патчи, так как изображения и так малы и можем представить их как один патч. Проблема такого подхода в том, что размеры сэмплов между друг другом разные, поэтому было принято решение нормировать их к одному размеру (50x50). Размеры были выбраны из статистики размеров сэмплов:



Таким образом, если высота сэмпла была больше 50, то просто ресайзим высоту до 50. Если же кроп меньше (50x50), то добавляем нули, чтобы размер был (50,50).
Пример такого результата показан ниже:





Процесс работы Dataloader

Dataloader хранит список путей к сэмплам (то есть при инициализации он получает json) и, когда он получает индекс пути, он загружает кортеж сэмпла и метки класса. Далее, делаем кроп сэмпла и, как мы помним, сэмпл - это список 10 изображений в формате ndarray. Модель такое не поймет и поэтому нам надо преобразовать его в torch.Tenzor. На выходе мы получаем сэмпл размерностью (10, 50, 50) и скаляр метки класса.

Итог

Как итог были размечены видео, из которых были сгенерированы обучающие и тестовые сэмплы. Написан набор инструментов для работы с данными, такие как конвертация разметки в удобный доля работы форматы (что понадобится в будущем) и генерация сэмплов и вспомогательные скрипты для исследования данных, такие ка сбор статистики и визуализаций.

Создание модели

Модель состоит из 3 компонентов:

- энкодер - входная часть, которая состоит из 2D свертки и решейпа
- трансформер - основная часть модели
- классификатор - выходная часть, отвечающая за классификацию

Энкодер

Основная задача энкодера - это преобразовать наш сэмпл из тензора, который представляет собой последовательность кропов, в векторные представления.

```
class PatchEncoderConv2D(nn.Module):

        def __init__(self, num_patches, emb_dim, P, C):
                super().__init__()
                self.emb_dim = emb_dim
                self.num_patches = num_patches
                self.conv_layer = nn.Conv2d(C, emb_dim, kernel_size=P, stride=P)
                self.pos_emb = nn.Parameter(torch.randn(num_patches, emb_dim))

        def forward(self, img:Tensor) -> Tensor:
                batch = img.shape[0]
                patches = self.conv_layer(img)
                patches = patches.reshape(batch, self.emb_dim, self.num_patches).swapaxes(1,2)
                return patches + self.pos_emb
```

На вход подается тензор размерностью (B, 10, 50, 50), где B - размер батча, 10 - количество изображений, 50 размеры высоты и ширины.
Свертка выполняет здесь сразу несколько задач:

1. создание фичей
2. разбивка этих фичей на векторы
   После свертки мы делаем решейп, чтобы получить тензор размером `(B, K*K, out_dim)`, где K - размер окна свертки.

Предположим, у нас входной тензор имеет размер (1, 10, 50, 50), подаем его в енкодер с `out_dim = 256, K=5` и на выходе мы получаем тензор размером
(1, 100, 256).

Атеншн

Подробно о работе аттеншена, здесь расписано не будет, лишь его техническая часть:

```
class SelfAttention(nn.Module):
        def __init__(self, emb_dim=256, key_dim=64, dropout=0.0) -> None:
                super().__init__()
                self.emb_dim = emb_dim
                self.key_dim = key_dim

                self.query = nn.Linear(emb_dim, key_dim)
```

```
                self.key = nn.Linear(emb_dim, key_dim)
                self.value = nn.Linear(emb_dim, key_dim)

        def forward(self, x):
                q = self.query(x)
                k = self.key(x)
                v = self.value(x)

                dot_prod = torch.matmul(q, torch.transpose(k, -2, -1))
                scaled_dot_prod = dot_prod / np.sqrt(self.key_dim)
                attention_weights = F.softmax(scaled_dot_prod, dim=1)
                weighted_values = torch.matmul(attention_weights, v)
                return weighted_values
```

Входной тензор проходит через линейные слои, которые дают нам **query**, **key**, **value**. Далее идет вычисление "внимания" после которого получаем взвешенные значения.

Блок трансформера

Представляет собой последовательность нормализации, атеншена и mlp слоя.

```
class TransformerBlock(nn.Module):
        def __init__(self, emb_dim=256, num_heads=8, hidden_dim=1024, dropout_prob=0.1) -> None:
                super().__init__()
                self.MSA = MultiHeadAttention(emb_dim, num_heads)
                self.MLP = MLP(emb_dim, hidden_dim)

                self.layer_norm1 = nn.LayerNorm(emb_dim)
                self.layer_norm2 = nn.LayerNorm(emb_dim)

                self.dropout1 = nn.Dropout(p=dropout_prob)
                self.dropout2 = nn.Dropout(p=dropout_prob)
                self.dropout3 = nn.Dropout(p=dropout_prob)

        def forward(self, x):
                out_drop_1 = self.dropout1(x)
                out_norm_1 = self.layer_norm1(out_drop_1)
                out_msa = self.MSA(out_norm_1)
                out_drop_2 = self.dropout2(out_msa)
                add = x + out_drop_2
                out_norm_2 = self.layer_norm2(add)
                out_mlp = self.MLP(out_norm_2)
                out_drop_3 = self.dropout3(out_mlp)
                return add + out_drop_3
```

где mlp - это многослойный перцептрон:

```
class MLP(nn.Module):
        def __init__(self, emb_dim=256, hidden_dim=1024) -> None:
                super().__init__()

                self.mlp = nn.Sequential(
                        nn.Linear(emb_dim, hidden_dim),
                        nn.GELU(),
                        nn.Linear(hidden_dim, emb_dim)
                        )

        def forward(self, x):
                x = self.mlp(x)
                return x
```

Входной тензор попадает в слой нормализации, после чего переходит в атеншн. Далее попадает в слой нормализации и в mlp. Полученный результат складывается со входными данными и идет на выход.

Трансформер

Как было сказано ранее, трансформер состоит из энкодера, трансформер блоков и классификатора. После блоков трансформера, я использую пуллинг по среднему, после чего линейными слоями делаю классификацию:

```python
class VisTransformer(nn.Module):
        def __init__(self, num_blocks:int, encoder_type:str=None, num_cls:int=1) -> None:
                super().__init__()
                self.num_blocks = num_blocks
                self.num_cls = num_cls
                if encoder_type == 'pad':
                        self.encoder = nn.Sequential(
                                PatchEncoderPad((50,50)),
                                PatchEncoderConv2D(100, 256, 5, 10)
                                )
                else:
                        self.encoder = PatchEncoderConv2D(100, 256, 5, 10)
                self.transformer_blocks = nn.Sequential(
                        *[TransformerBlock() for _ in range(num_blocks)]
                        )

                self.glob_avg = nn.AdaptiveAvgPool2d((1,1))
                self.head = nn.Sequential(
                        nn.Linear(1, 256)
                        nn.GELU(),
                        nn.Linear(256, num_cls),
                        nn.Sigmoid()
                        )

        def forward(self, x):
                encoded = self.encoder(x)
                blocks_out = self.transformer_blocks(encoded)
                avg = self.glob_avg(blocks_out).squeeze(1)
                out = self.head(avg)
                return out.squeeze()
```

Подавая сэмпл размером (1, 10, 50, 50) в модель, сперва попадает в энкодер, который, в качестве результата выдает тензор размером (1, 100, 256) в блоки трансформера. Пройдя все блоки, выходная карта признаков будет размером (1, 100, 256). После пуллинга по среднему, получаем тензор размером (1, 1), который подается в классификатор, который выдает значение размером (1,).

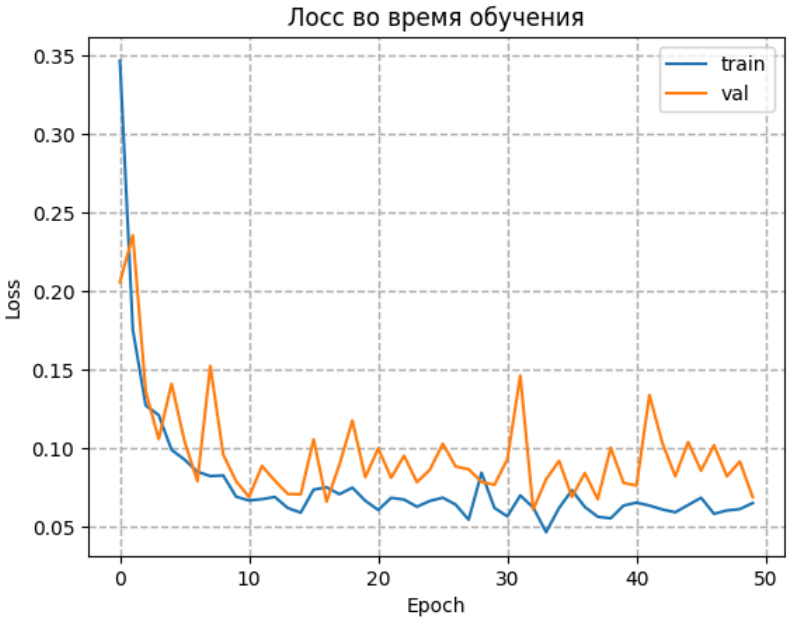
Обучение

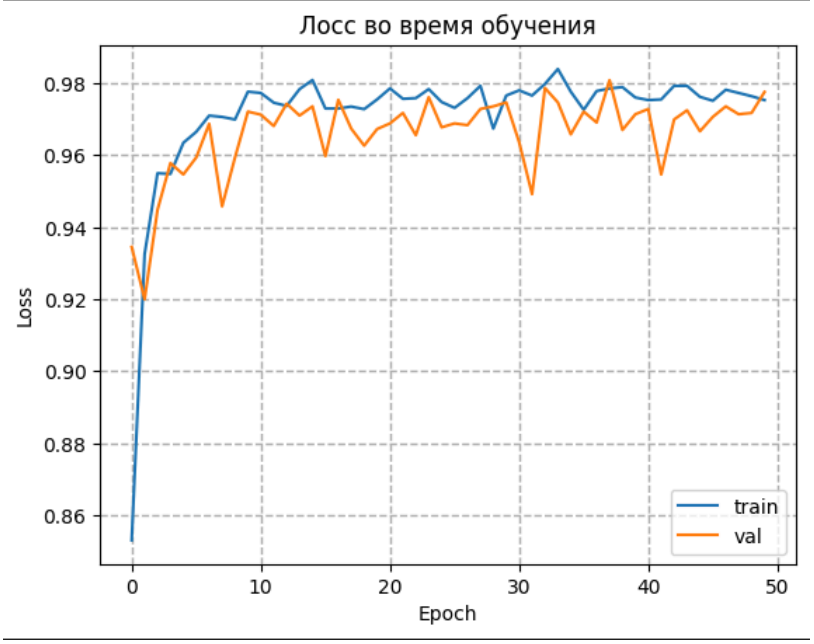
После того, как были собраны данные, подготовлена модель. Можно приступать к обучению.

В качестве эксперимента было выбрано использовать 1 блок трансформера, чтобы оценить насколько хорошо справится наименьшая модель.

Модель обучалась 50 эпох. В качестве оптимизатора был выбран Adam с шагом обучения 0.001. В качестве лосс функции была выбрана бинарная кроссэнтропия.

Ниже представлены графики обучения:

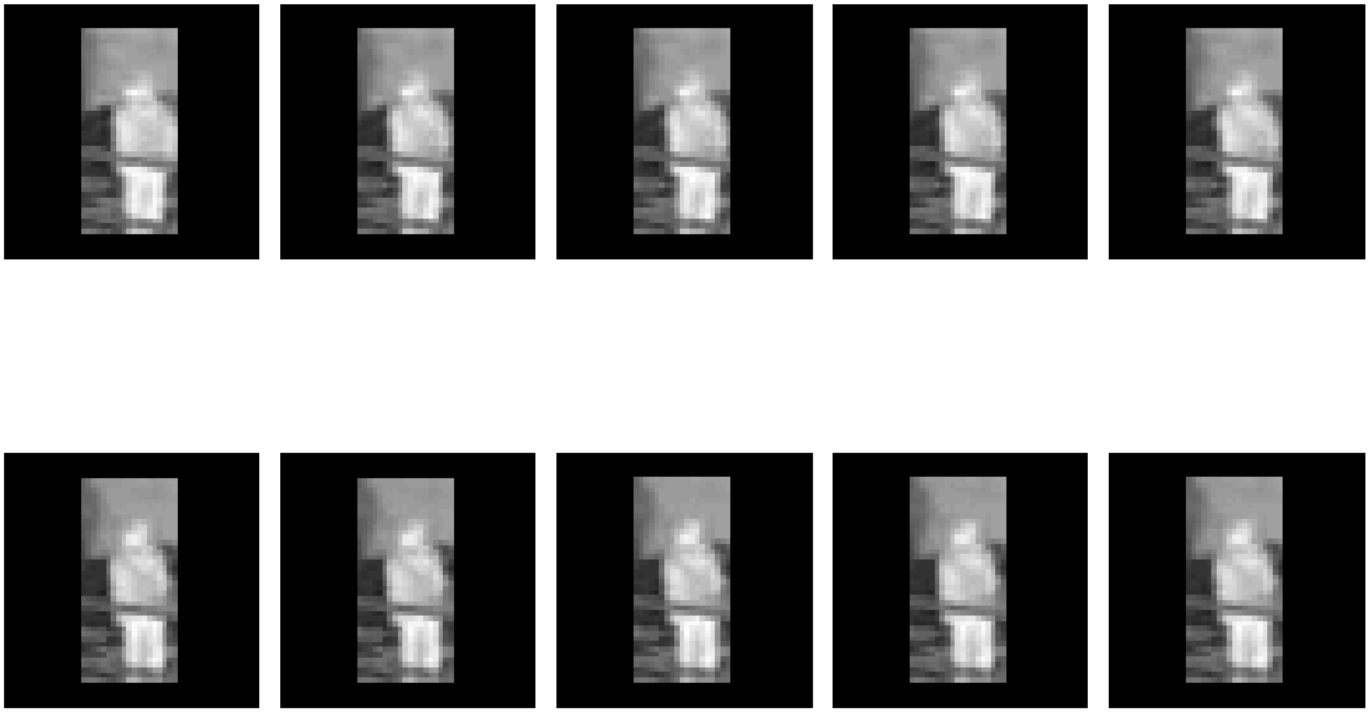




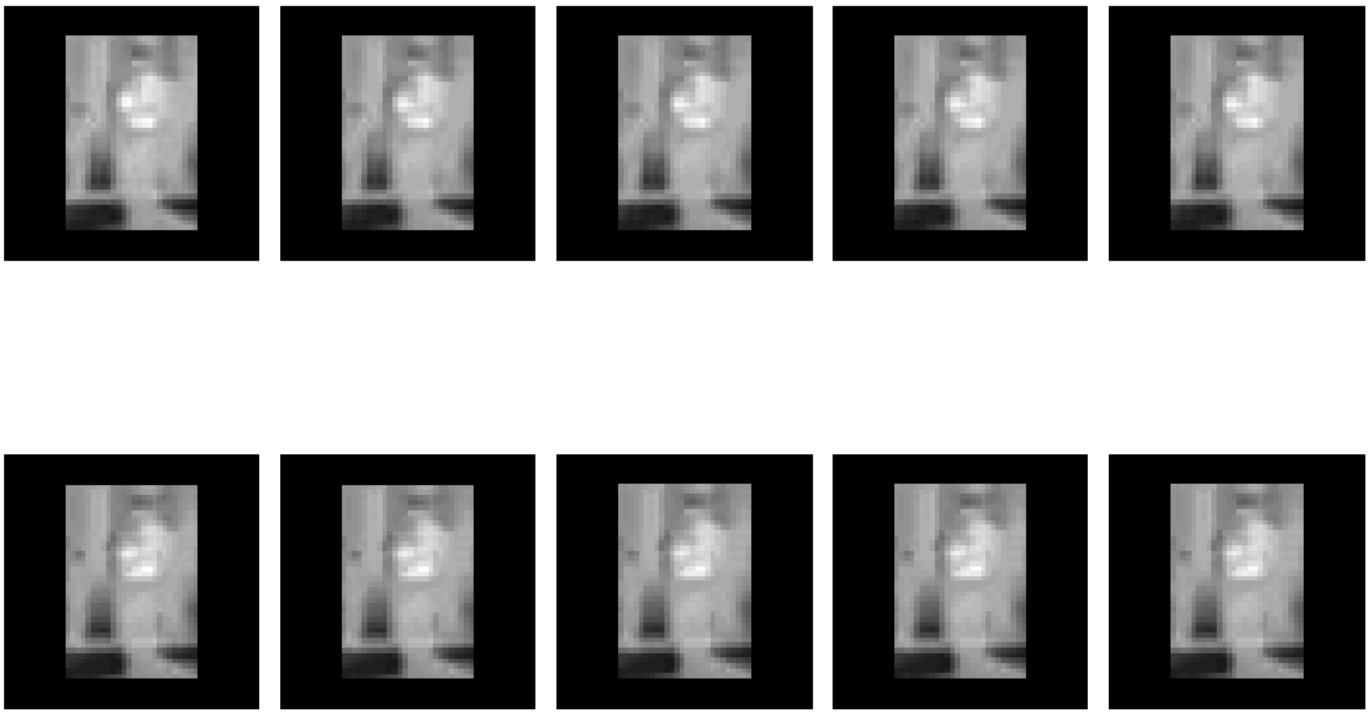
Как видно из результатов обучения модель из одного блока трансформера способна достигать до 98% точности без переобучения, что дает нам утверждать: "Трансформер способен работать с последовательностью кадров и выявлять оттуда людей различных небольших размеров".

Примеры предсказаний:

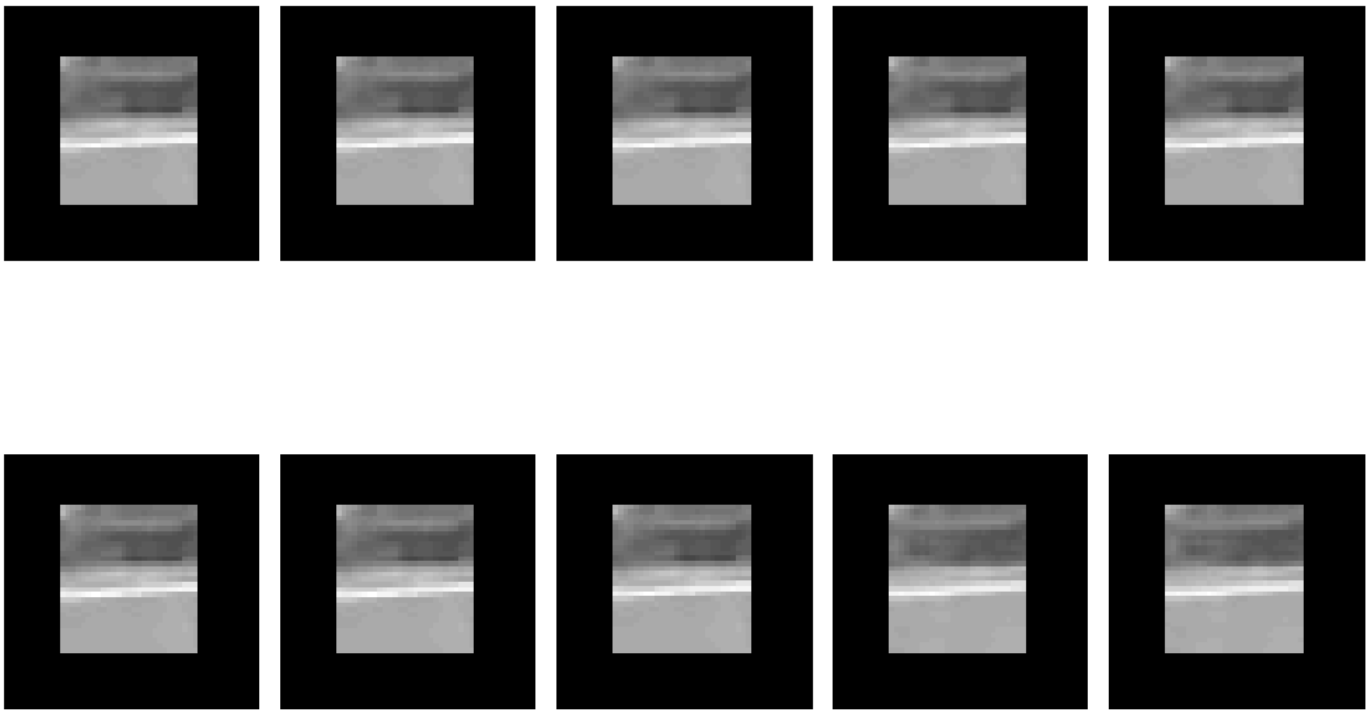
human | pred: 1.00
sample size: (40, 19)



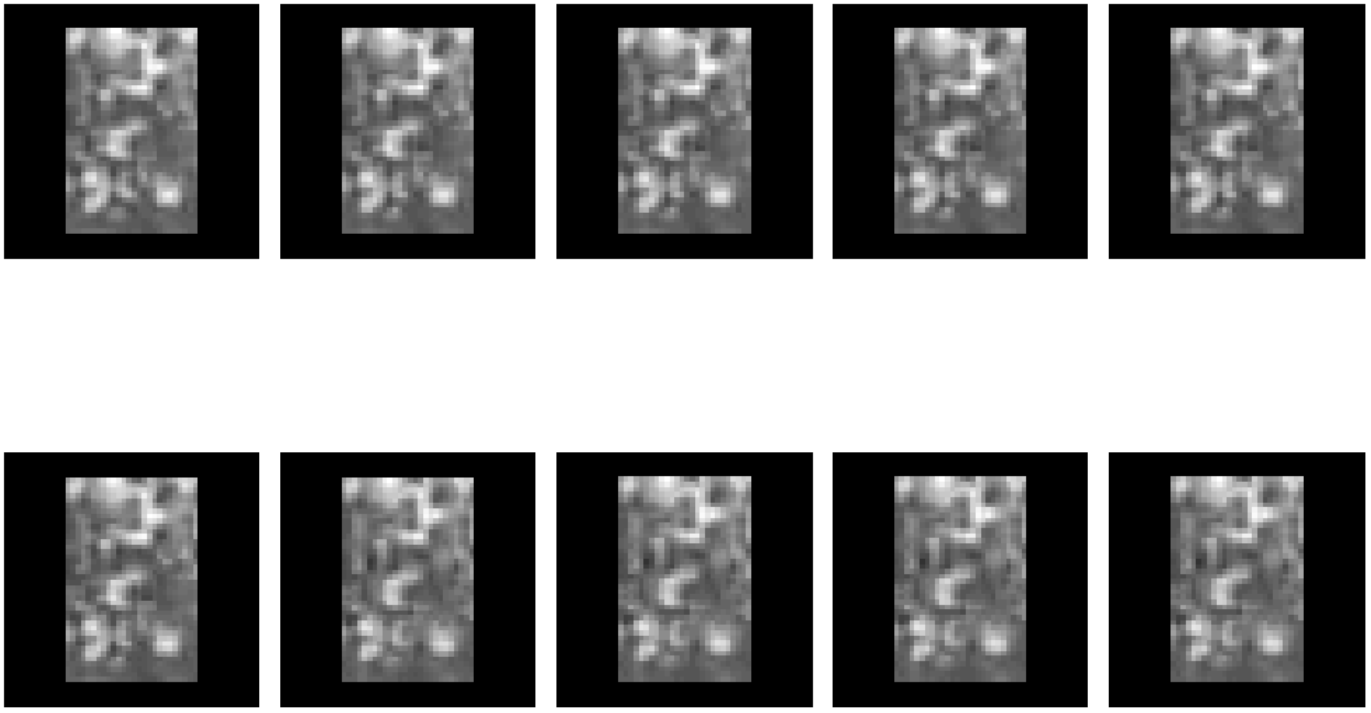
human | pred: 1.00
sample size: (38, 26)



bg | pred: 0.00
sample size: (29, 27)



bg | pred: 0.47
sample size: (40, 26)



Что дальше?

- Модернизировать модель для детекции
- Провести эксперимент по детекции людей на последовательностях полного кадра