**Классы:**

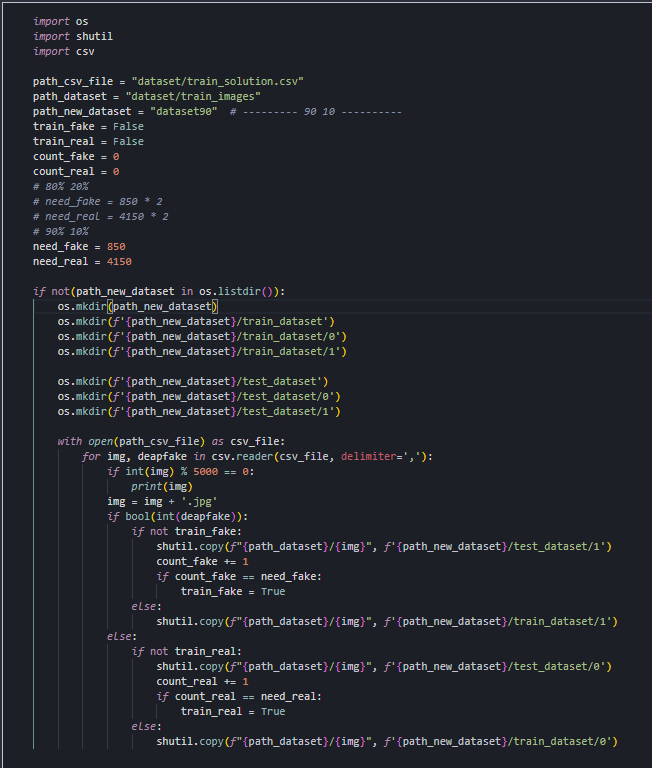
* Реальные лица
* deepfake лица

**Структура:**

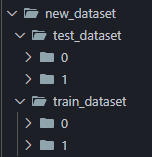
* train\_images - 50\_000 images
* test\_images - 10\_000 images
* train\_solution - метки классов для обуч выборки(0, 1)

**Особенности датасета:**

* Дисбаланс классов
* Наличие шума

1. Преобразование датасета

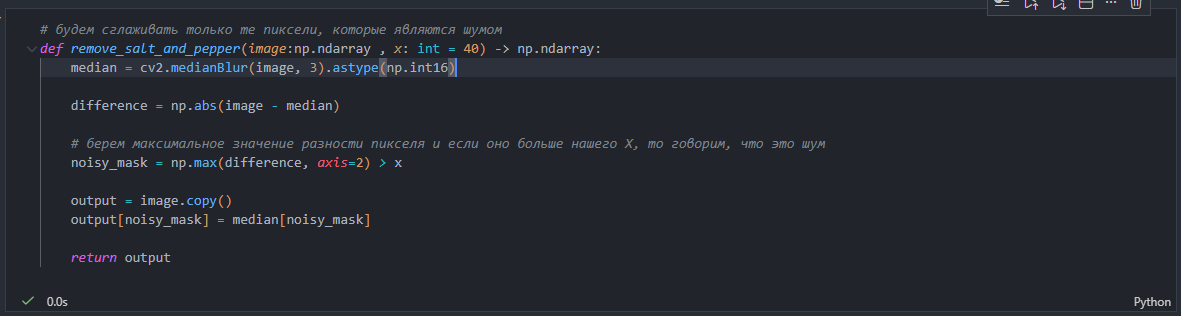
Так мы создаём датасет с такой структурой:



Распределили классы по папкам, чтобы прочитать датасет как ImageFolder.

2.Шум убирал [@THE\_ALlVE](https://t.me/THE_ALlVE), пробовал использовать модели и брал среднее от соседей у битых пикселей.

Я([@THE\_ALlVE](https://t.me/THE_ALlVE)) проверял разные модели на их способность убирать salt and pepper шум, но и UNet, и ResNet убирали шум так, что появлялось размытие лица. Для нашей задачи это критично, тк именно с помощью мелких текстур лица модель должна определять, сгенерированное это лицо или нет. В итоге я просто попробовал самую примитивную структуру, заменять критичные пиксели(либо слишком темных, либо слишком светлых) на средние соседних с ним. Это дало лучший результат, поэтому все последующие преобразования грязного датасета в чистый будут использовать именно этот метод (функция remove\_salt\_and\_pepper)



3. Балансировка классов:

**WeightedRandomSampler** – создаём баланс в батчах, подбирает батчи где кол-во изображений классов равно.



**Аугментация и сохранение изображений меньшего класса до равенства**

*import* random

*def* balance\_deepfake\_dataset(real\_path: str, deepfake\_path: str) -> None:

    real\_count = *len*([

        f *for* f *in* os.listdir(real\_path)])

    files = os.listdir(deepfake\_path)

*# оригиналы*

    originals = [

        f *for* f *in* files

*if* not (f.startswith("flip\_") or

                f.startswith("affine\_") or

                f.startswith("light\_") or

                f.startswith("all\_"))

        ]

*print*(*f*"Оригинальных fake: {*len*(originals)}")

*print*(*f*"Количество real: {real\_count}")

*# аугментации*

    flip\_transform = transforms.RandomHorizontalFlip(p=1.0)

    affine\_transform = transforms.RandomAffine(

        degrees=5,

        translate=(0.1, 0.1),

        scale=(0.9, 1.1),

    )

    light\_transform = transforms.ColorJitter(

        brightness=0.25,

        contrast=0.25,

        saturation=0.25,

    )

    all\_transform = transforms.Compose([flip\_transform, affine\_transform, light\_transform])

*# создаём по одной копии каждой аугментации*

*for* img\_file *in* originals:

        img\_path = os.path.join(deepfake\_path, img\_file)

        img = Image.open(img\_path).convert("RGB")

*# flip*

        flip\_name = "flip\_" + img\_file

*if* flip\_name not in files:

            flip\_transform(img).save(os.path.join(deepfake\_path, flip\_name))

*# affine*

        affine\_name = "affine\_" + img\_file

*if* affine\_name not in files:

            affine\_transform(img).save(os.path.join(deepfake\_path, affine\_name))

*# light*

        light\_name = "light\_" + img\_file

*if* light\_name not in files:

            light\_transform(img).save(os.path.join(deepfake\_path, light\_name))

*# all*

        all\_name = "all\_" + img\_file

*if* all\_name not in files:

            all\_transform(img).save(os.path.join(deepfake\_path, all\_name))

*def* list\_images():

*return* [f *for* f *in* os.listdir(deepfake\_path)]

*def* random\_transform(img\_path):

        augmentations = [["flip\_", flip\_transform],

                         ["affine", affine\_transform],

                         ["light\_", light\_transform],

                         ["all\_", all\_transform]]

        prefix, augmentation = random.choice(augmentations)

        img = Image.open(img\_path).convert('RGB')

*return* (prefix, augmentation(img))

*# считаем после аугментаций*

    current\_deepfake = *len*(list\_images())

    need\_add = current\_deepfake - real\_count

*print*(*f*"\nПосле аугментаций deepfake = {current\_deepfake} (real = {real\_count})")

*# если у нас кол-во deepfake > real*

*if* need\_add > 0:

        for\_delete = [

            f *for* f *in* list\_images()

*if* f.startswith(("flip\_", "affine\_", "light\_", "all\_"))]

*# уменьшаем diff, если аугментаций меньше*

        need\_add = *min*(need\_add, *len*(for\_delete))

        to\_delete = random.sample(for\_delete, need\_add)

*for* f *in* to\_delete:

            os.remove(os.path.join(deepfake\_path, f))

*print*(*f*"Удалено файлов: {*len*(to\_delete)}")

*elif* need\_add < 0:

        image\_list\_for\_aug = random.choices(list\_images(), k=*abs*(need\_add))

*for* img\_file *in* image\_list\_for\_aug:

            img\_path = os.path.join(deepfake\_path, img\_file)

            prefix, augmented\_img = random\_transform(img\_path)

            augmented\_name = prefix + img\_file

            augmented\_img.save(os.path.join(deepfake\_path, augmented\_name))

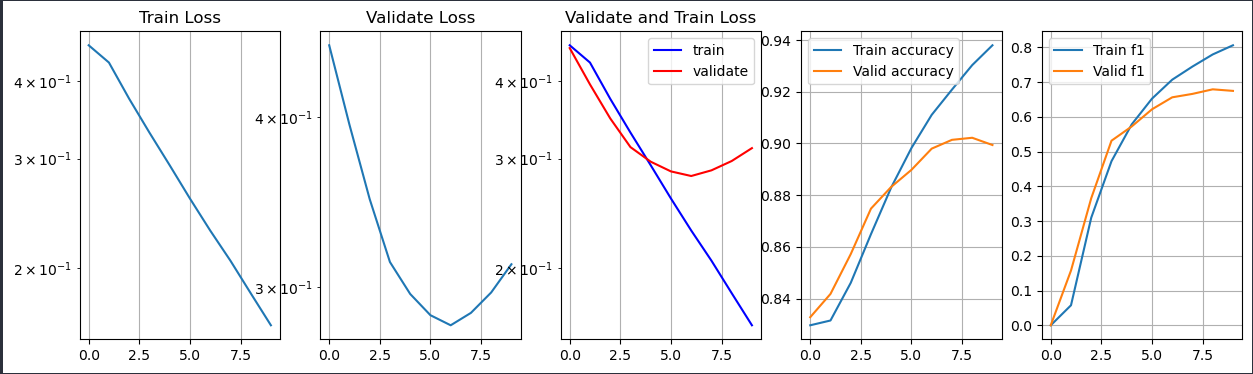
*print*(*f*"\nИтоговое количество deepfake = {*len*(list\_images())}")

**Модели:**

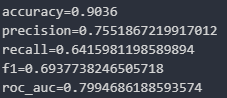
Я написал: **простая CNN, Resnet18, Resnet50**

Мой напарник написал **Unet, resnet (для денойзинга), Inception v1, Inceptionv3**

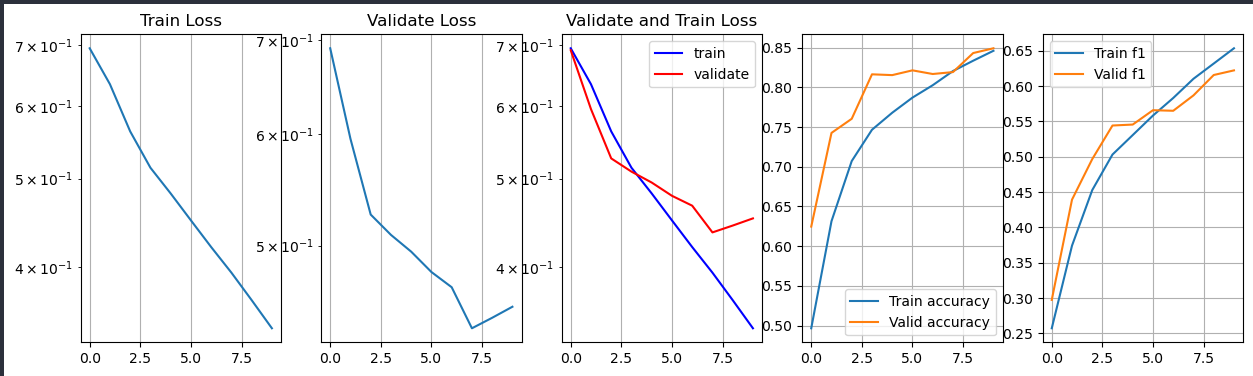
Обучение моделей:

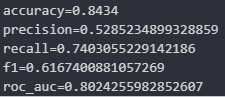
**Cnn**: обучаем 10 эпох на изображениях с шумом, смотрим как модель будет справляться, метрика f1 для 0 класса.

Метрики модели: (Метрики для 0 класса)



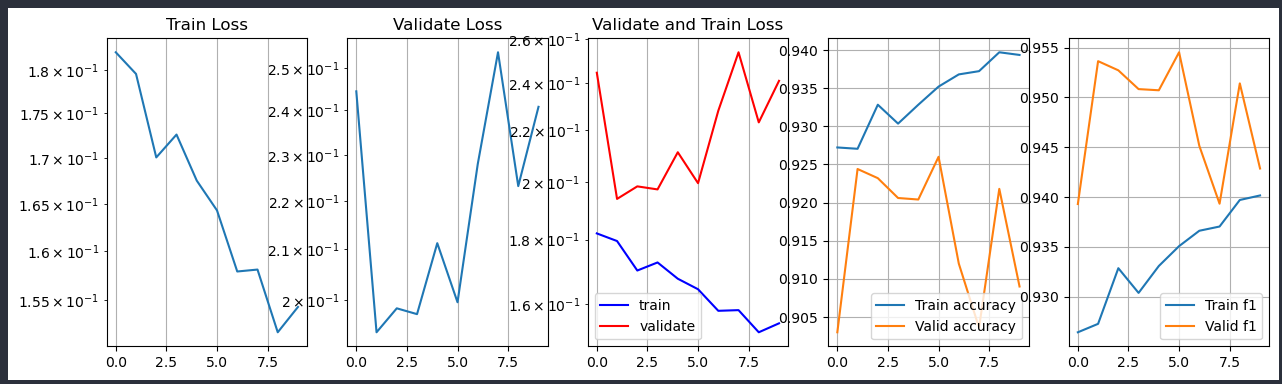
Cnn 10 эпох с измененными весами классов в функции потерь. Метрики для 0 класса. Изображения с шумом

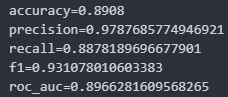


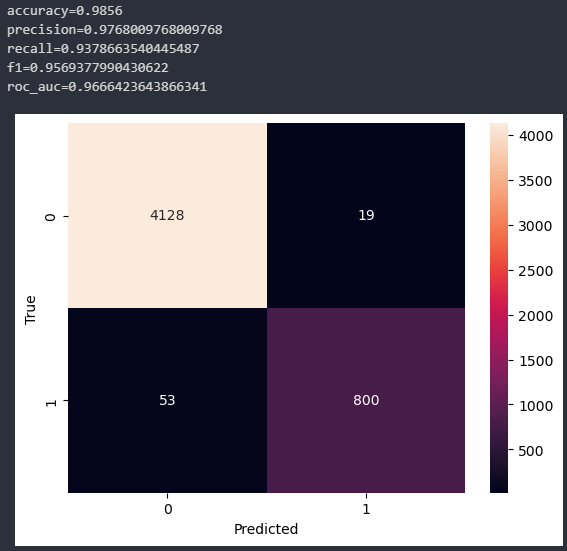
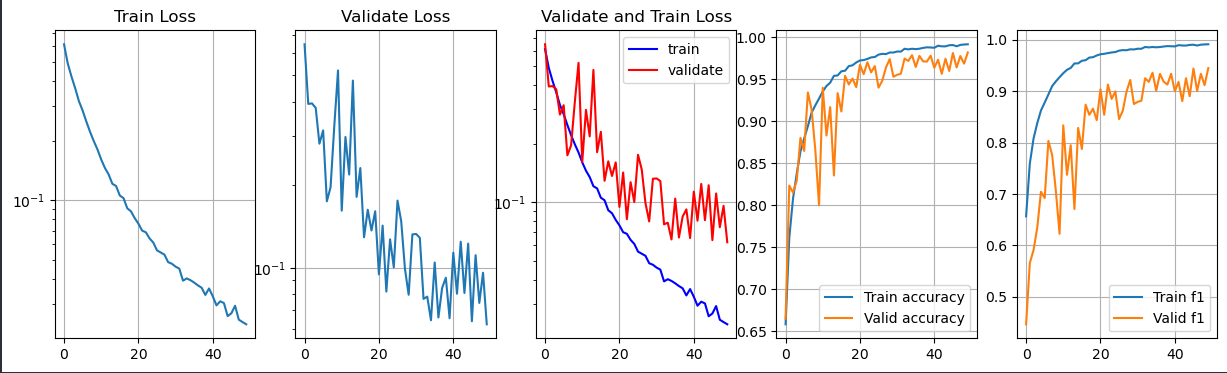


Модель уже не успевает сойтись за 10 эпох,

Cnn с аугментацией и WRS 30 эпох. На графике послдние 10 эпох, обучал за раз по 10 эпох метрики для 0 класса. Изображения с шумом



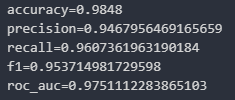


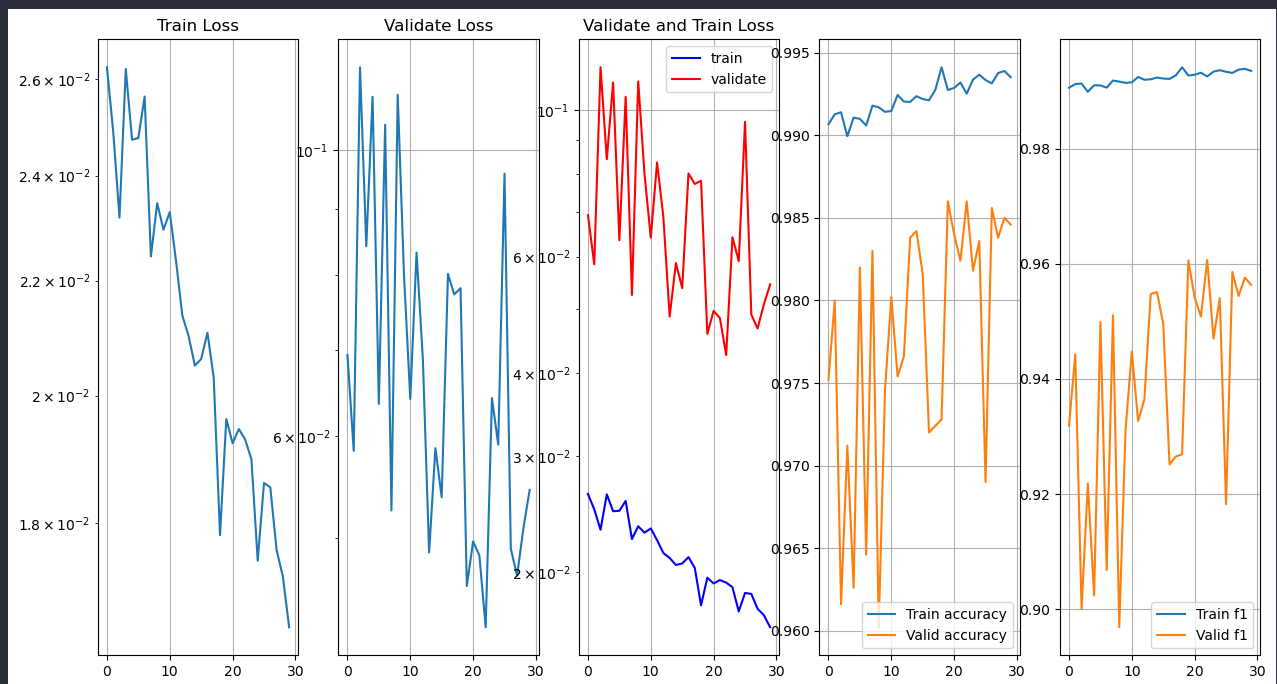
**ResNet18**, Аугментация, WRS, 50 эпох, изображения с шумом. Метрики уже нужного класса. 

Модель получилась хорошей, хорошо предсказывает оба класса.

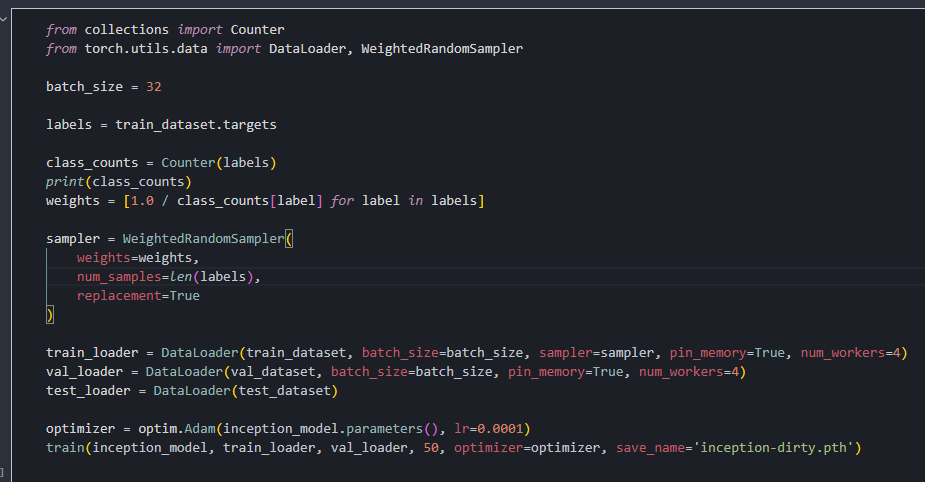
Далее пробуем усложнить модель. **ResNet50** обучаем 100 эпох на чистых изображениях + WRS.

На графике снизу последние 30 эпох. Модель получилась, чуть хуже чем resnet18.

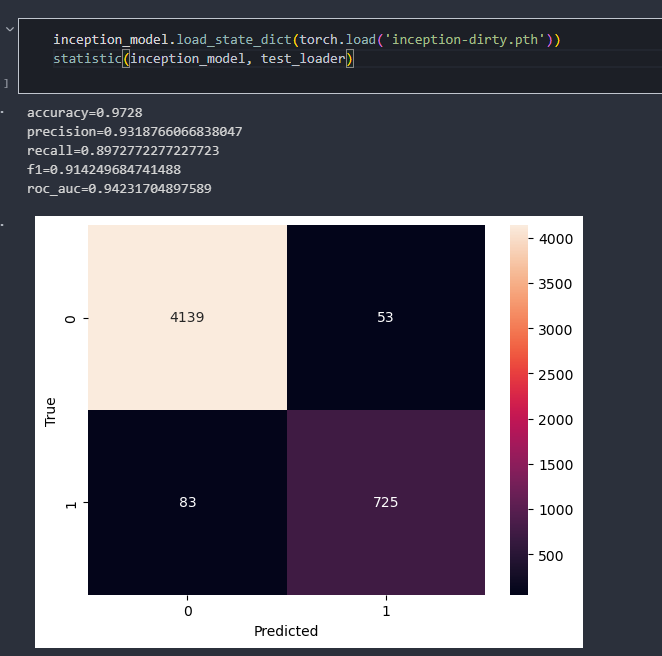




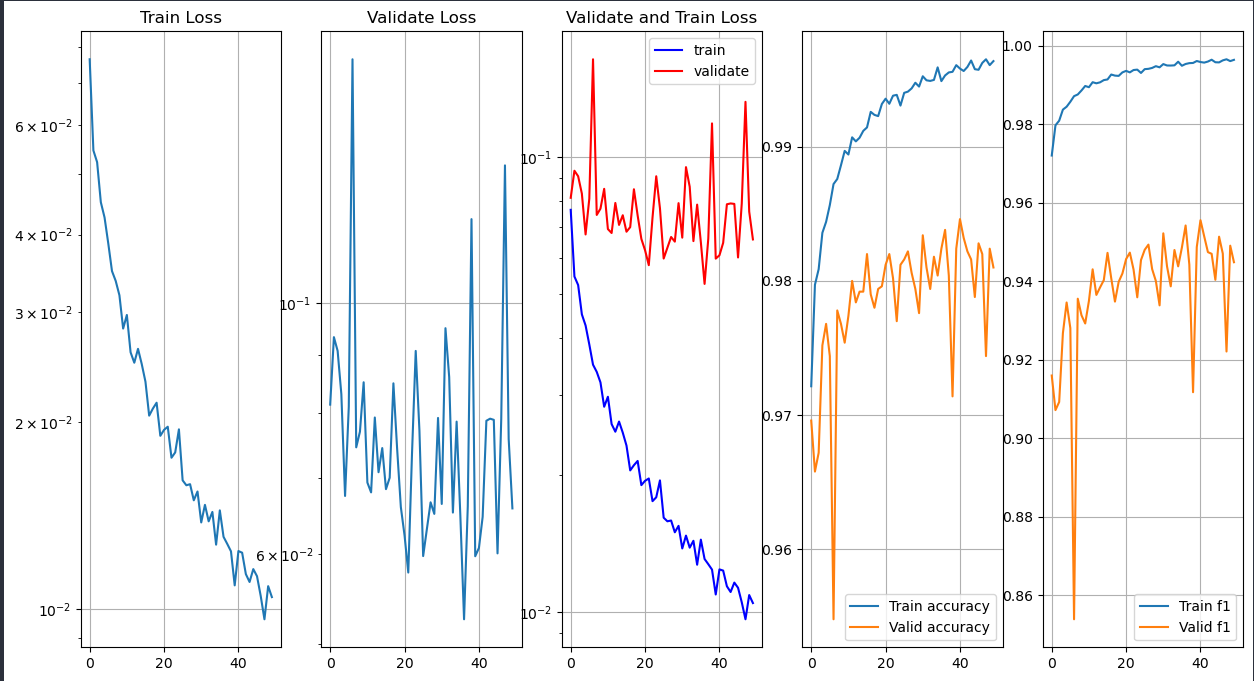
Мы заметили что когда мы берем модель обученную на чистых изображениях, и смотрим её результат на битых фото результат получается сильно хуже чем результат на чистых изображениях. Модель обученная на шумных картинках даёт почти одинаковый результат на шумных\чистых изображениях. Обучая модель на шумных фото она запоминает одни признаки, обучая на чистых другие. Пробуем обучить модель сначала на шумных изображениях, затем на чистых.

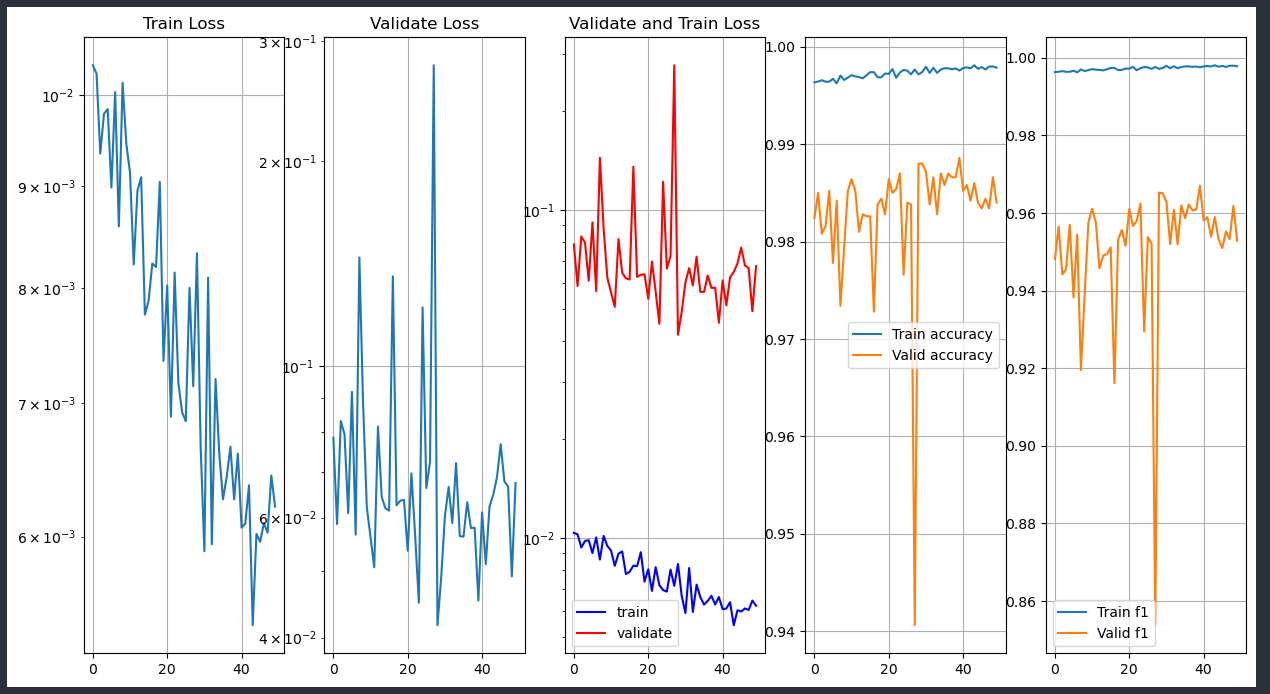
Пробуем **Inception v1** с сбалансированным датасетом с помощью аугментации 

Получаем такой результат, обученная на шумных изображениях

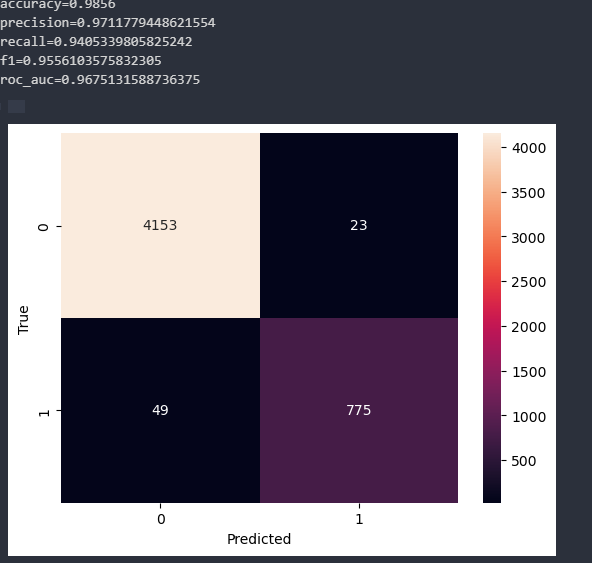


Теперь дадим этой же модели чистые изображения, чтобы она выучила новые признаки:



Пробуем обучить ещё 50 эпох

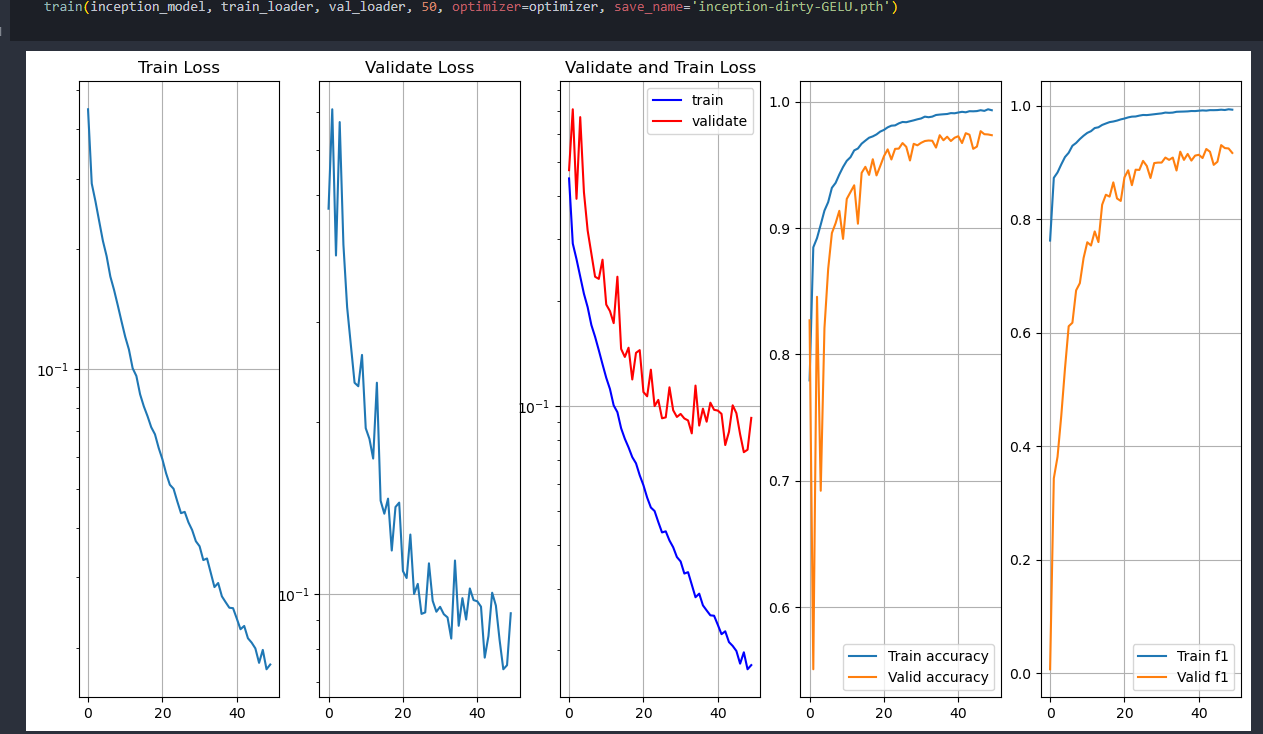
Вот результаты лучшей модели по accuracy



Довольно хороший результат.

**Пробуем улучшить его разбив датасет 90% трейн 10% валидация, и изменяем функцию активацию с ReLU -> GELU. Обучаем по такой-же технологии.**

50 на грязных



Добавим ещё эпох

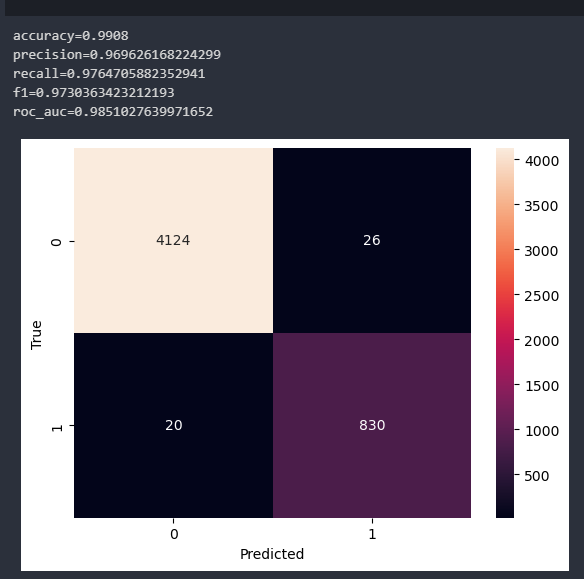
50 на чистых 

Почти дошли до f1 = 0.97 пробуем дообучить модель



Превысили 0.97 f1 ёё результат на валидационной выборке: (тк делили 90% 10% тест данных нет, тестом будет kaggle)

**На kaggle получили Public score: 0.97784**



**Я(**[@THE\_ALlVE](https://t.me/THE_ALlVE)**),** по мимо обучения моделей для денойзинга(**denoise\_UNet\_model, denoise\_UNet\_n2v\_model, denoise\_ResNet\_model, ResNet\_n2v**) обучил **UNet** заточенный под классификацию, **ResNet\_18**, **InceptionV1**, **InceptionV3**.

Сначала рассмотрим denoise модели:

**denoise\_UNet\_model**



**denoise\_UNet\_n2v\_model**



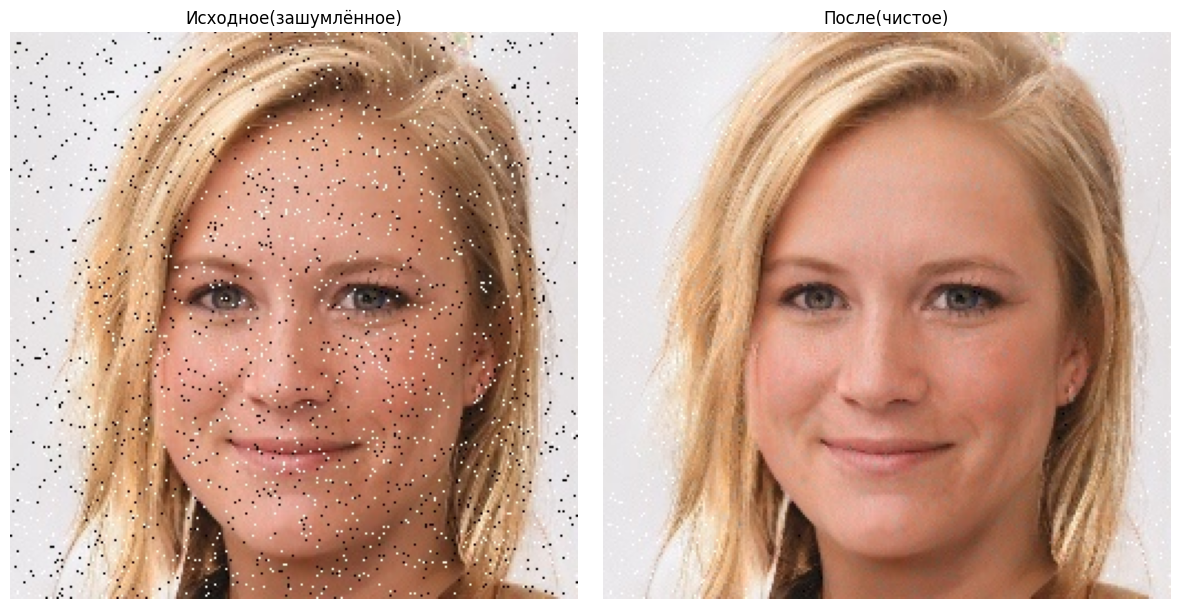
**denoise\_ResNet\_model**



**ResNet\_n2v**

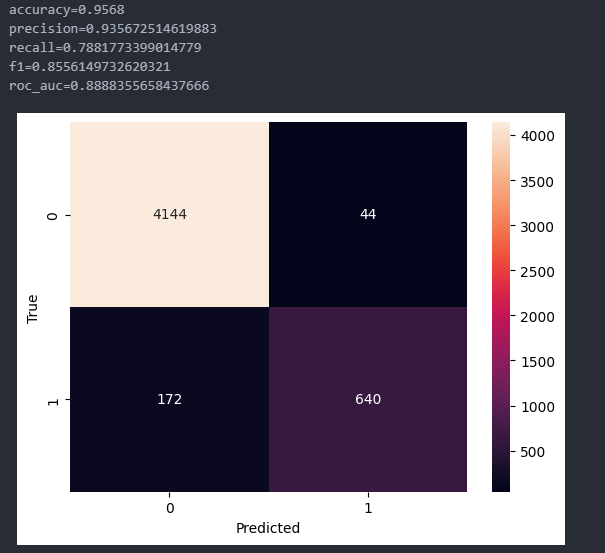


**Ну и простая архитектура замены критичных пикселей на среднее вокруг этого пикселя**

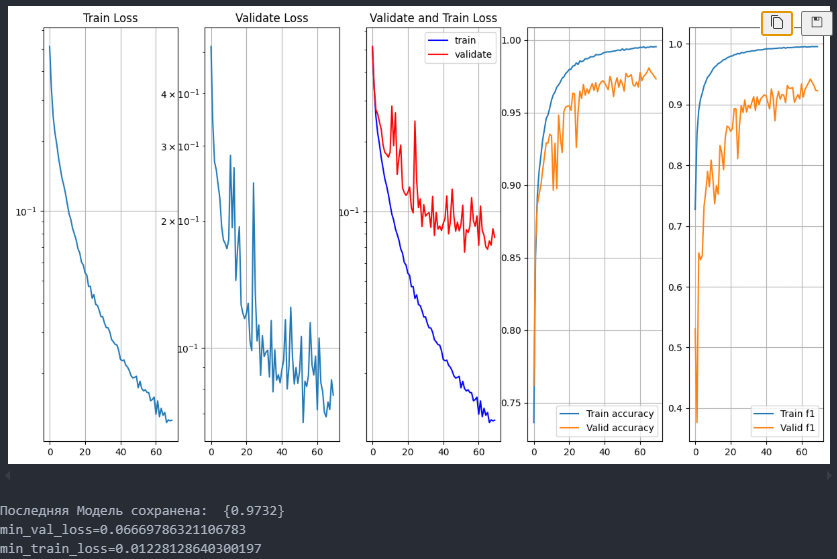


Теперь модели классификации

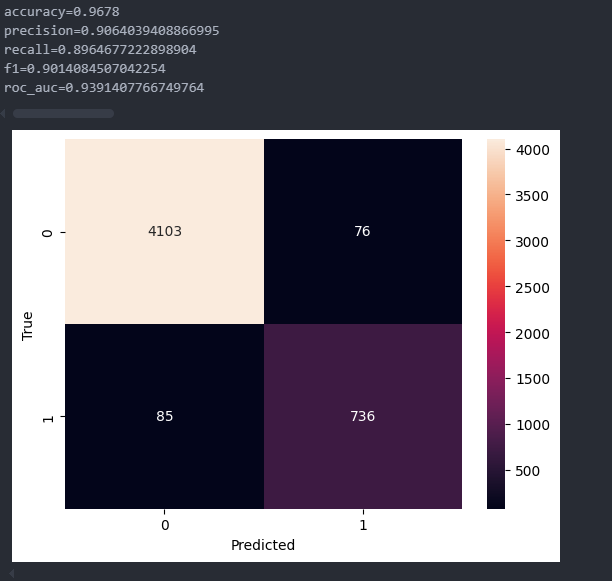
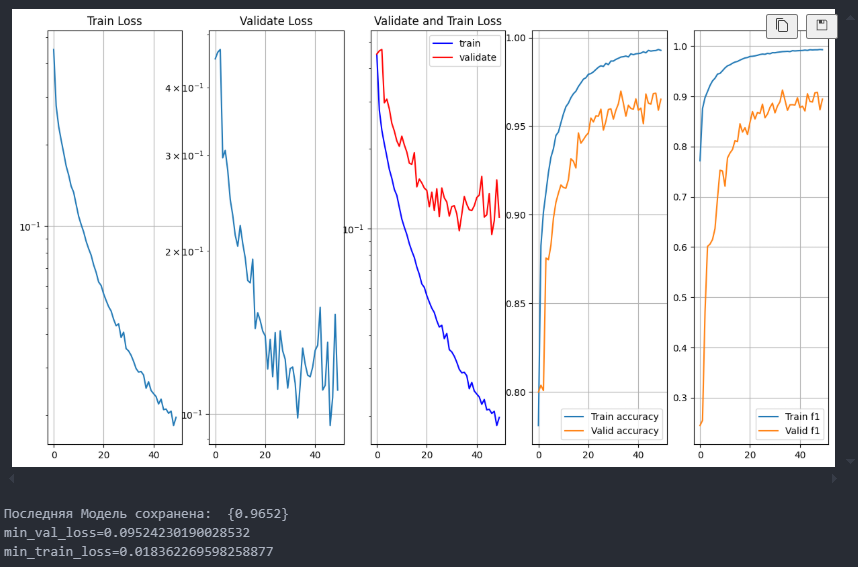
**UNet для классификации** не дал хороших результатов, это ожидаемо, тк его цель это далеко не классификация



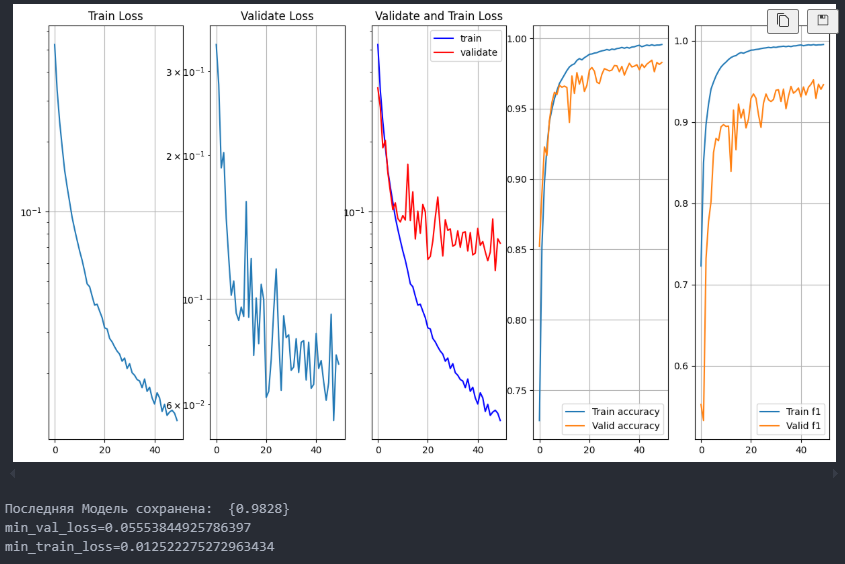
**ResNet\_18** обучил только на грязных данных, результат тоже не супер.



**Inception V1** сначала на грязных



Потом на чистых, **в моменте лучший f1-score**





После этих результатов, мы поняли, что все дальнейшие идеи лучше реализовывать на ней, тк результаты очень хорошие за относительно короткое время обучения. Это довольно ожидаемо, тк идея использования нескольких сверток параллельно, а потом брать лучшую помогает модели найти самые важные признаки не используя слишком трудоемких процессов, как, например, VGG.

**Inception V3, в моменте дал самый лучший f1-score**, но он не был очень далек от f1-score Inception V1, поэтому мы использовали Inception V1(его было быстрее обучать).

На чистых

