Я реализовал pix2pix на датасетах facades и cityscapes. Ссылки на датасеты взяты с гитхаба из статьи.

Для генератора использовалась модель Unet, дискриминатор рассматривал патчи 70x70, детали реализации старался сделать такими же, как в исходной статье, а именно Appendix 6.1 и гитхаб, ссылка на который есть в нем же.

В целом, Unet на вход получает разметку, последовательно уменьшает изображение с помощью конволюций, используя LeakyReLU и BatchNorm2d (я решил использовать InstanceNorm2d, причина ниже), далее последовательно увеличивает изображение с помощью аналогичных конволюций, также используя ReLU и BatchNorm2d.
Дополнительно в Unet есть residual connections между парами слоев в энкодере (уменьшает изображение) и декодере (увеличивает изображение).
Дискриминатор на вход получает разметку и изображение для тестирования, объединяет каналы (3+3=6), последовательно уменьшает изображение с помощью конволюций таким образом, что в итоге получается матрица 30x30, в которой каждый элемент соответствует окну 70x70 в исходном изображении. Таким трюком мы предполагаем, что за один проход мы сразу рассмотрим несколько окон размера 70x70 в изображении.
Размер батча был выбран 1 по совету из статьи. В реализации на pytorch я увидел использование InstanceNorm, попробовав его (вместе с другими изменениями) я увидел заметное улучшение в качестве, поэтому оставил его вместо BatchNorm. Также если после конволюции использовалась нормализация, то bias из конволюции убирался за ненадобностью.
На вход подается картинка 256x256, в тренировке она увеличивается до 286x286, после чего выбирается случайное окно 256x256, а также случайно горизонтально переворачивается. При предсказании слои dropout не выключаются.
Тренировка происходила следующим образом: берется батч, сначала на нем обучается дискриминатор, после чего обучается генератор (то есть генератор и дискриминатор обучаются не параллельно).
Дискриминатор решает задачу бинарной классификации патча 70x70 как настоящего (1) и сгенерированного (0), генератор пытается его обмануть, а также уменьшить L1 метрику до настоящего изображения.
BCE - BinaryCrossEntropy, G - Generator, D - Discriminator
Generator_loss = BCE(D(source, G(source)), 1) + 100 * L1(target, G(source)),
Discriminator_loss = BCE(D(source, target), 1) + BCE(D(source, G(source)), 0)

Помимо упомянутых функций потерь (будем называть GAN), для сравнения модель тренировалась без дискриминатора, в таком случае Generator_loss состоял только из L1 расстояния (будем называть L1).

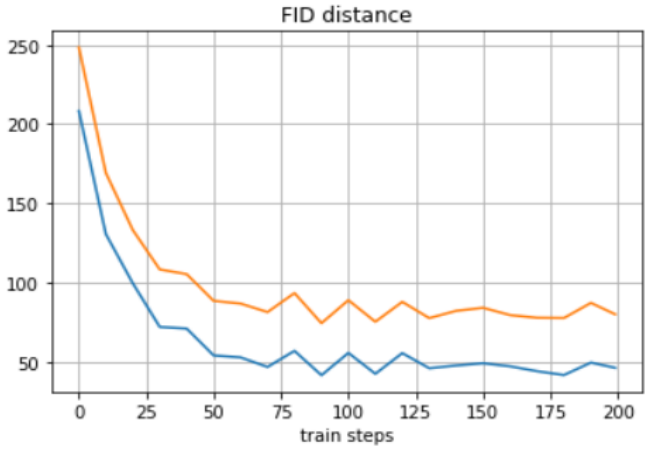
Оптимизатор брался из статьи(как и все остальное):
Adam(lr=0.0002, beta1=0.5, beta2=0.999)

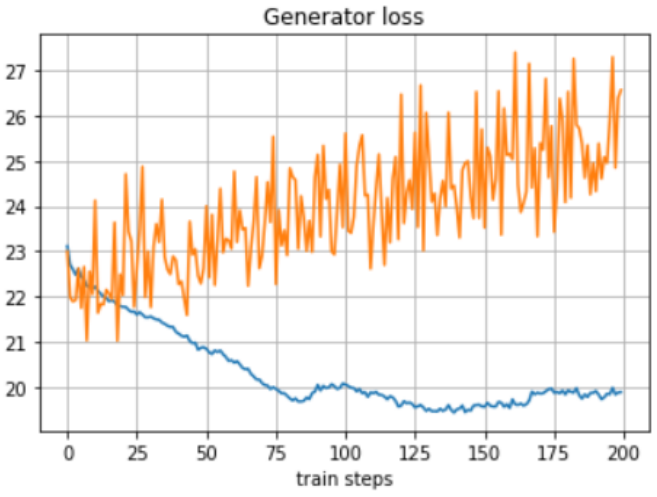
Датасет facades разбит на train, val, test с размерами 400, 100, 106 картинок.
Датасет cityscapes не имел тестовой выборки, валидационная выборка была разбита.
Итоговые размеры: 2975, 300, 200 картинок.

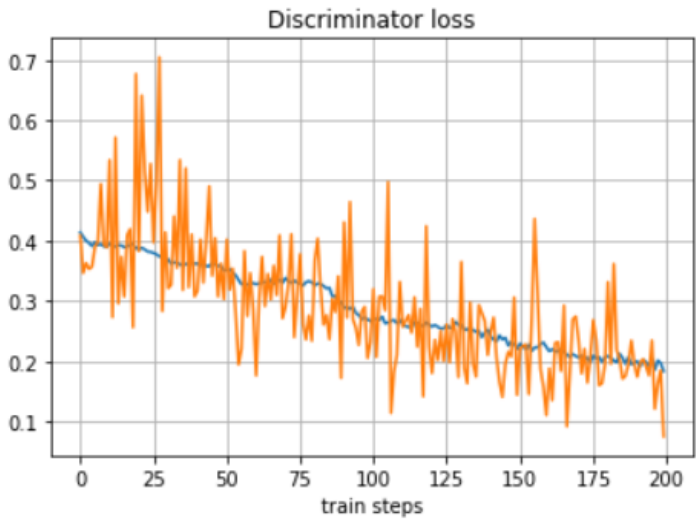
Датасеты обучались 200 эпох.

В качестве количественной метрики используется FID, который считается каждые 10 эпох.
Сначала рассмотрим cityscapes.
Сравним финальный FID: GAN vs L1
train: 46 vs 148
val: 80 vs 188
test: 105 vs 203
время: 370m vs 256m

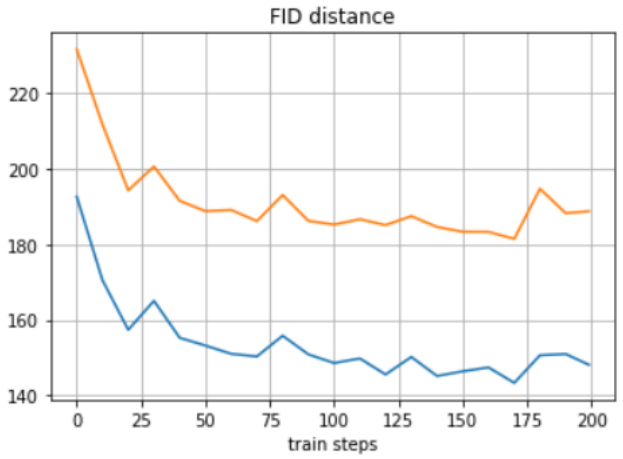
GAN



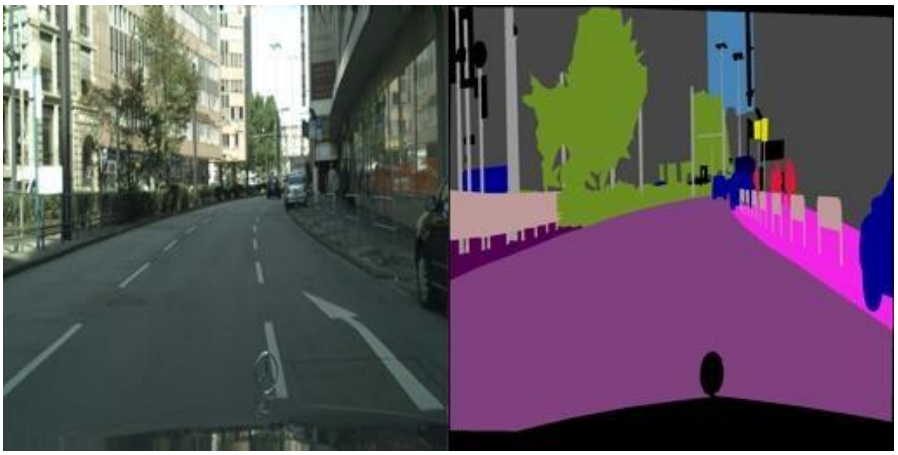




L1



Посмотрим на изменение результатов предсказаний в зависимости от эпохи для одного примера из валидации



Эпоха: 1 100 200

GAN:

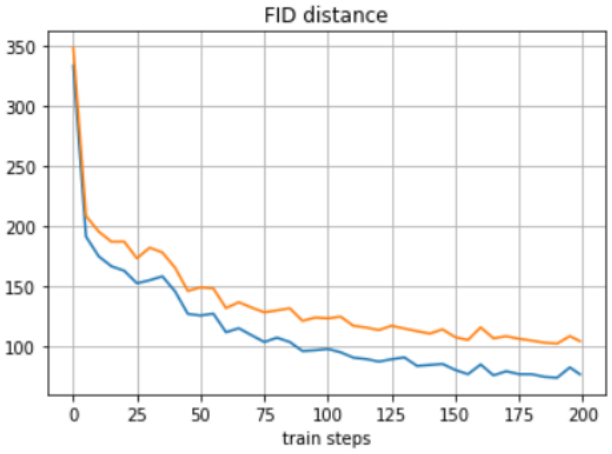
val:

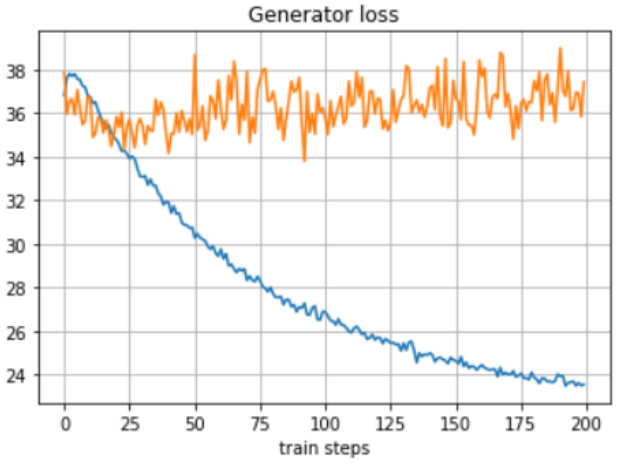
В конце приведем несколько примеров из тестовой выборки

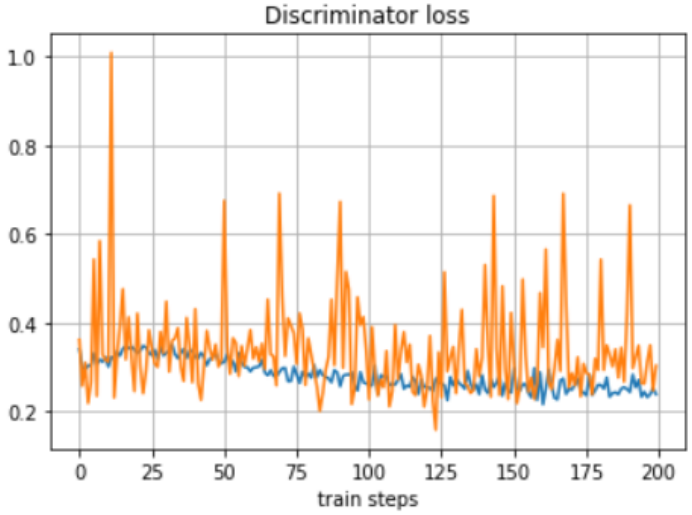
разметка ответ GAN L1

Теперь рассмотрим facades.
Сравним финальный FID: GAN vs L1
train: 76 vs 151
val: 103 vs 171
test: 115 vs 187
время: 70m vs 45m
Сравнивая график FID для GAN с остальными тремя FID графиками можно предположить, что увеличенный размер датасета увеличил время сходимости. Для L1 графика это не так выражено.

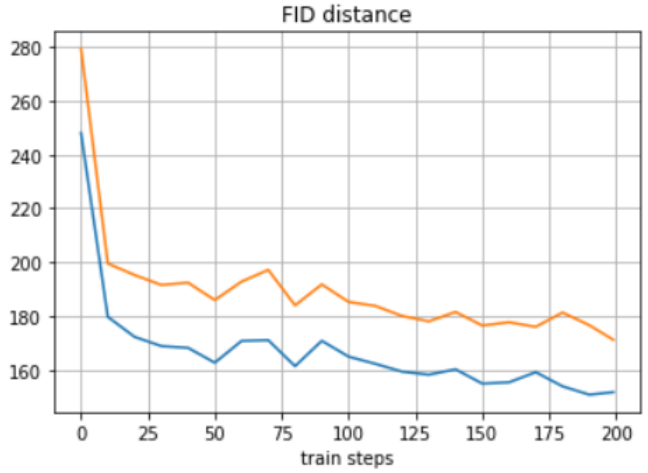
GAN

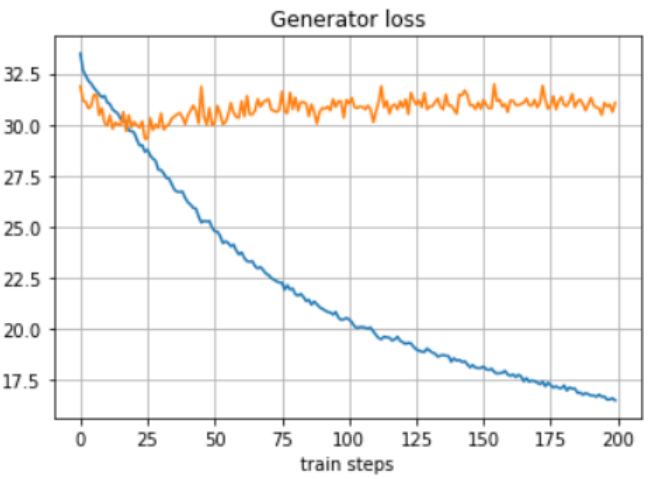




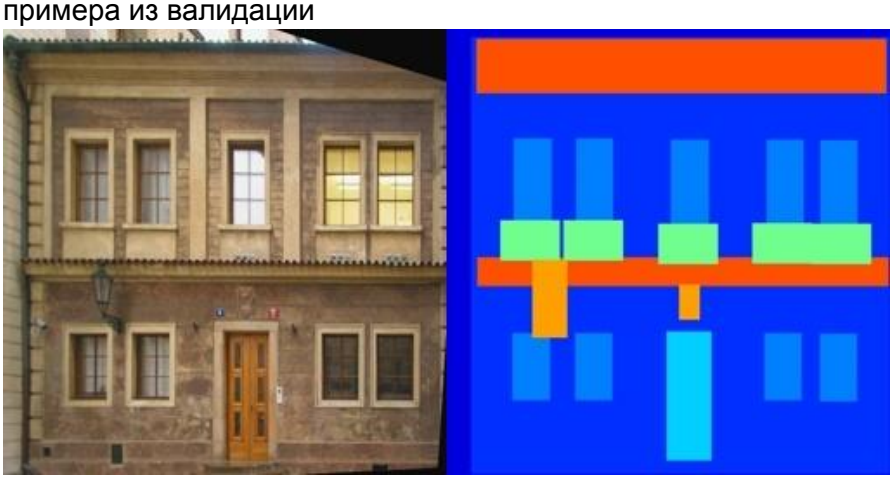


L1





Посмотрим на изменение результатов предсказаний в зависимости от эпохи для одного примера из валидации



Эпоха: 1 100 200

GAN:

val:

В конце также приведем несколько примеров из тестовой выборки

разметка ответ GAN L1