Airfoil GAN

Кодировка и синтез профиля крыла для оптимизации аэродинамических характеристик

Кирилл Ленский

Moscow Institute of Physics and Technology

28 октября 2023 г.

Содержание

Постановка задачи

- Найти способ генерации новых профилей крыла
- Научиться предсказывать подъемную силу, сопротивление, момент тангажа и критическую скорость
- Научиться генерировать профили с заданными характеристиками

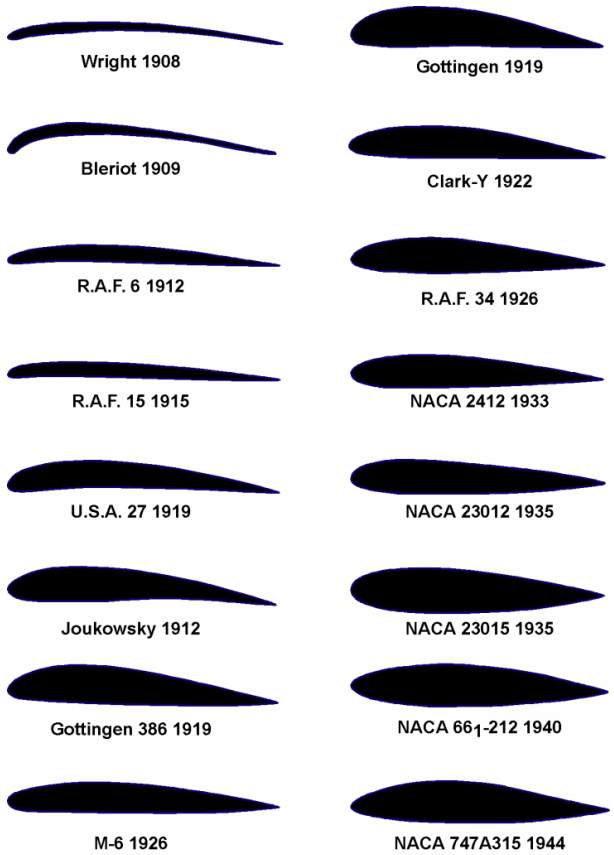


Рис.: Примеры профиля крыла

Представление данных

Как описать профиль крыла компьютеру?

Представим задачу: векторизовать профиль. Неформально говоря, мы хотим параметризовать форму крыла векторным пространством, с помощью которого можно будет генерировать новые профили.

Представление данных

Как описать профиль крыла компьютеру?

Представим задачу: векторизовать профиль. Неформально говоря, мы хотим параметризовать форму крыла векторным пространством, с помощью которого можно будет генерировать новые профили.

- Полиномиальная и сплайновая кодировка
- Free-Form Deformation (FFD)
- ???

Представление данных

Полиномы, кривые Безье, сплайны

- Можно задавать форму профиля линейной комбинацией фиксированных многочленов
- Может появиться проблема оверфиттинга – многочлены старшего порядка будут зашумлять кривую

Представление данных

Полиномы, кривые Безье, сплайны

- Можно задавать форму профиля линейной комбинацией фиксированных многочленов
- Может появиться проблема оверфиттинга – многочлены старшего порядка будут зашумлять кривую
- Можно использовать кривые Безье или сплайны

Но как быть, если размерность данных > 1? Что, если мы хотим параметризовать не просто сечение, а целое крыло? В таком случае полиномиальная параметризация потребует слишком много опорных точек, а сама форма будет "волнистой".

Представление данных

Free-Form Deformation (FFD)

- FFD – описание преобразования объекта через преобразование содержащего его выпуклого множества (шара или куба).
- Преобразование можно связать со свойствами крыла аналитически.

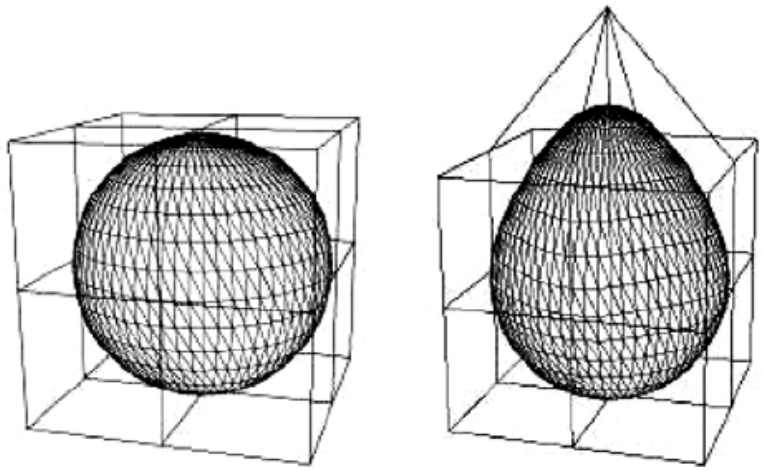


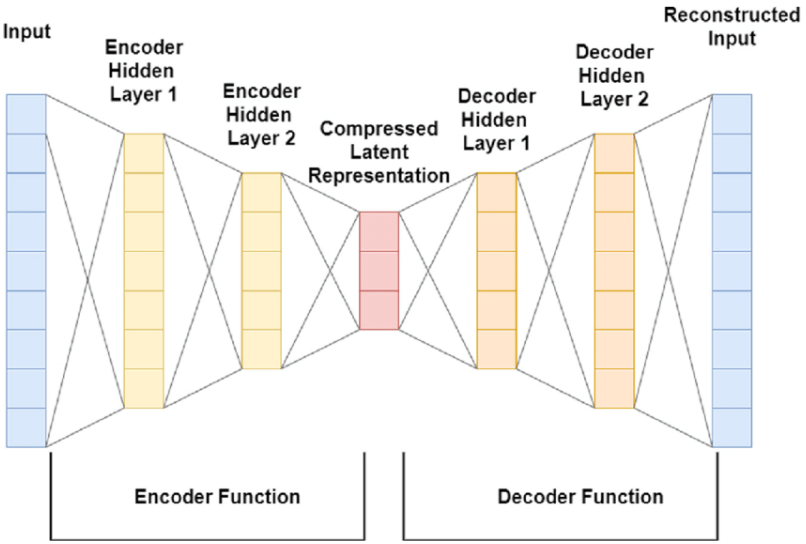
Рис.: FFD

Классические способы понижения размерности могут давать очень хорошие результаты, но обычно преобразование в них задается "вручную".

Представление данных

Автокодировщики

А вот с задачей понижения размерности без участия человека хорошо справляются автокодировщики:



- Автокодировщик представляет из себя нейросеть, напоминающую песочные часы
- В первой половине данные "сжимаются", во второй – "разжимаются".
- Во время обучения кодировщик оптимизирует какую-либо функцию потерь между исходными данными и данными, прошедшими через кодировщик

Вариационный автокодировщик

Variational **A**uto **E**ncoder (**VAE**) – модификация обычного автокодировщика, которая не просто переводит данные в пространство меньшей размерности, но делает это случайным образом, стремясь к тому, чтобы закодированные объекты были распределены нормально в **латентном пространстве**. Пусть

$$z \sim \mathrm{Enc}\,(x) = q(z|x)$$

$\sim \mathrm{Dec}\,(z) = p(\tilde{x}|z)$ где мы предполагаем, что $q(z|x)$ – условное распределение латентного вектора z, который выдал нам кодировщик, относительно σ-алгебры вектора признаков x, а $p(\tilde{x}|z)$ – распределение результата работы декодировщика относительно латентного вектора z. Будем считать, что $p(z) \sim \mathcal{N}(0, \mathbf{I})$.

Вариационный автокодировщик

Случайность в **VAE** достигается тем, что кодировщик генерирует 2 детерминированных вектора

- $\mu(x)$ – вектор средних значений
- $\sigma(x)$ – вектор стандартных отклонений

А после конструирует из них случайный гауссовский вектор:

$$\text{Enc}\,(x) = \mathcal{N}(\mu, \text{diag}\,(\sigma_1, \ldots, \sigma_n))$$

Вариационный автокодировщик

Функция потерь $\mathcal{L}_{\text{VAE}}$ вычисляется по следующей формуле

$$\begin{aligned}\mathcal{L}_{\text{VAE}} &= [\mathcal{L}_{\text{recon}}] + [\mathcal{L}_{\text{prior}}] = \\ &= \left[\|x - \tilde{x}\|_2^2\right] + \left[D_{\text{KL}}\left(q(z|x)\|p(z)\right)\right]\end{aligned}$$

, где D_{KL} – дивергенция Кульбака-Лейблера

Дивергенция Кульбака-Лейблера

$$D_{\text{KL}}(q\|p) = \int_{\mathcal{X}} q(x) \ln \frac{q(x)}{p(x)}\, d\mu(x)$$

Главная проблема *VAE* – он не достаточно хорош в генерации примеров, которых не было в тренировочной выборке.

Генеративно-Состязательные нейросети

Generative **A**dversarial **N**etwork (**GAN**)– система, состоящая из дискриминатора $\mathcal{D}$ и генератора $\mathcal{G}$ – двух нейросетей. Генератор получает на вход случайный шум $\tilde{p}$, который преобразует в сгенерированный объект $\tilde{x}$. После этого $\mathcal{D}$ получает последовательно на вход 2 объекта: x – настоящий объект из выборки, и $\tilde{x}$ – объект, сгенерированный $\mathcal{G}$.

В качестве ошибки, в силу причин Байесовского характера, обычно берется кросс-энтропия:

$$\mathcal{L}_{\text{GAN}} = \log \mathcal{D}(x) + \log(1 - \mathcal{D}(\tilde{x}))$$

VAE vs. GAN

VAE:

- Позволяет понижать размерность
- Плохо генерирует новые объекты

GAN:

- Не позволяет понижать размерность
- Хорошо генерирует новые объекты

VAEGAN совмещает плюсы **VAE** и **GAN**

Архитектура **VAEGAN**

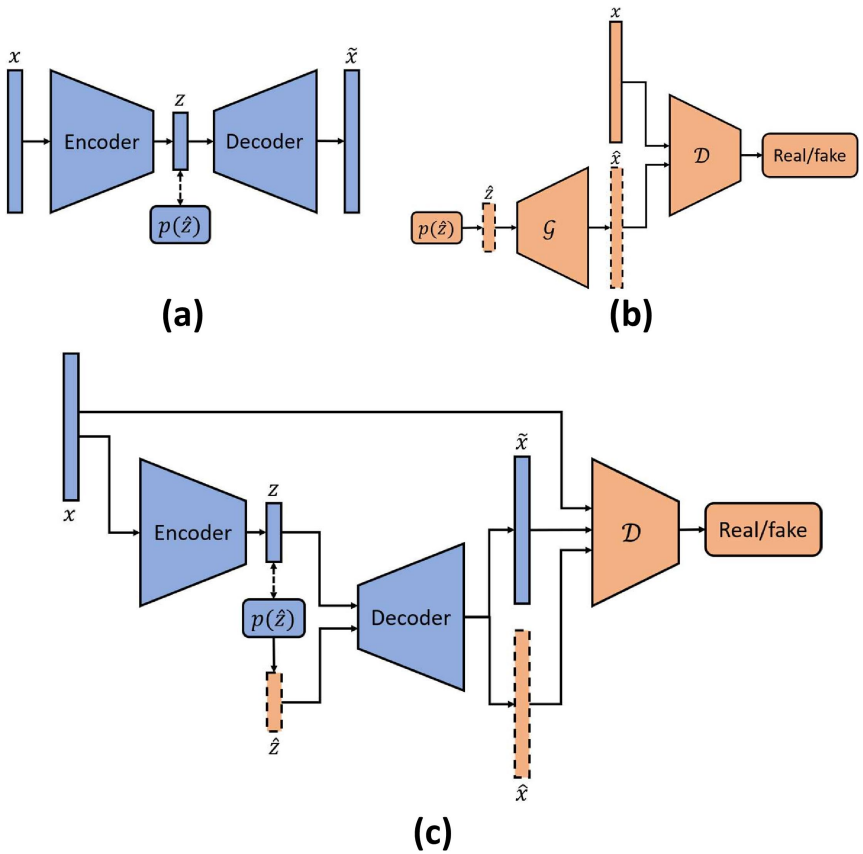


Рис.: VAE, GAN и VAEGAN

- x – реальный объект из выборки
- $\tilde{x} = \mathrm{Dec}\left(\mathrm{Enc}\left(x\right)\right)$ – объект, пропущенный сквозь автокодировщик
- $\hat{x} = \mathrm{Dec}\left(z\right)$ – объект, сгенерированный **VAE** из нормального шума

Архитектура VAEGAN

Функция потерь **VAEGAN** представляет из себя взвешенную сумму нескольких функций:

$\mathcal{L}_{\text{GAN}}$ – лосс-функция, относящаяся к **GAN**. Теперь дискриминатору надо классифицировать $\hat{x}$ и $\tilde{x}$ как неправильные, а x как правильный объект, поэтому в кросс-энтропии три слагаемых:

$$\mathcal{L}_{\text{GAN}} = \log(\mathcal{D}(x)) + \log(1 - \mathcal{D}(\text{Dec}\,(z)) + \log(1 - \mathcal{D}(\text{Dec}\,(\text{Enc}\,(x)))$$

Архитектура VAEGAN

Функция потерь **VAEGAN** представляет из себя взвешенную сумму нескольких функций:

$\mathcal{L}_{layer}$ – дополнительная функция потерь для **VAE**, которая сравнивает, насколько отличается результат на скрытом слое $\mathcal{D}$ для шума и закодированного объекта:

$$\mathcal{L} = \|\mathcal{D}_l(x) - \mathcal{D}_l(\mathrm{Dec}\,(z))\|_1$$

В нашем случае в качестве слоя выбран предпоследний скрытый слой.

Архитектура **VAEGAN**

Функция потерь **VAEGAN** представляет из себя взвешенную сумму нескольких функций:

Функции $\mathcal{L}_{\text{recon}}$ и $\mathcal{L}_{\text{prior}}$, которые уже встречались нам в **VAE**

$$\mathcal{L}_{\text{recon}} = \|\tilde{x} - x\|_2^2$$

$$\mathcal{L}_{\text{prior}} = D_{\text{KL}}(q(z|x)\|p(z))$$

Причем дивергенция вычисляется явно через $\mu(x)$ и $\sigma(x)$ как

$$D_{\text{KL}} = \frac{1}{2}\left[\left(\sum_{i=1}^{n}\mu_i^2 + \sum_{i=1}^{n}\sigma_i^2\right) - \sum_{i=1}^{n}(\log(\sigma_i^2) + 1)\right]$$

Архитектура VAEGAN

Функция потерь **VAEGAN** представляет из себя взвешенную сумму нескольких функций:

$$\mathcal{L} = \lambda_0 \mathcal{L}_{\text{preior}} + \lambda_1 \mathcal{L}_{\text{recon}} + \lambda_2 \mathcal{L}_{\text{layer}} + \lambda_3 \mathcal{L}_{\text{GAN}}$$

где гиперпараметры λ_i модели надо подбирать отдельно.

Препроцессинг

Прежде, чем мы начнем имплементировать[1] **VAEGAN**, необходимо привести данные к регулярному виду. Авторы использовали сглаживание кубическими сплайнами

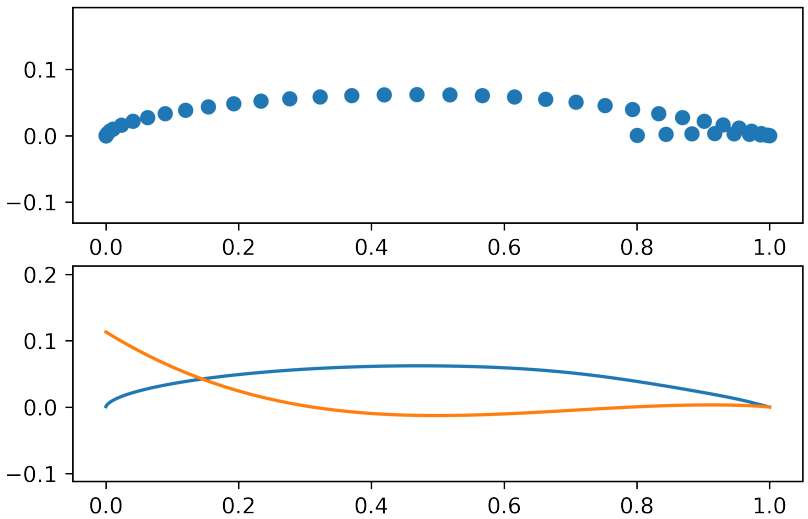
Алгоритм приближения

- Разбить профиль крыла на верхнюю и нижнюю кривые
- Приблизить каждую из них кубическими сплайнами
- Выбрать $N = 100$ опорных координат x, $x_k = 1 - \cos\left(\frac{\pi k}{2N}\right)$
- Вернуть вектор из 200 чисел – конкатенацию $y_{\text{up}}(x_i)$ и $y_{\text{down}}(x_i)$
- Нормализовать получившиеся векторы

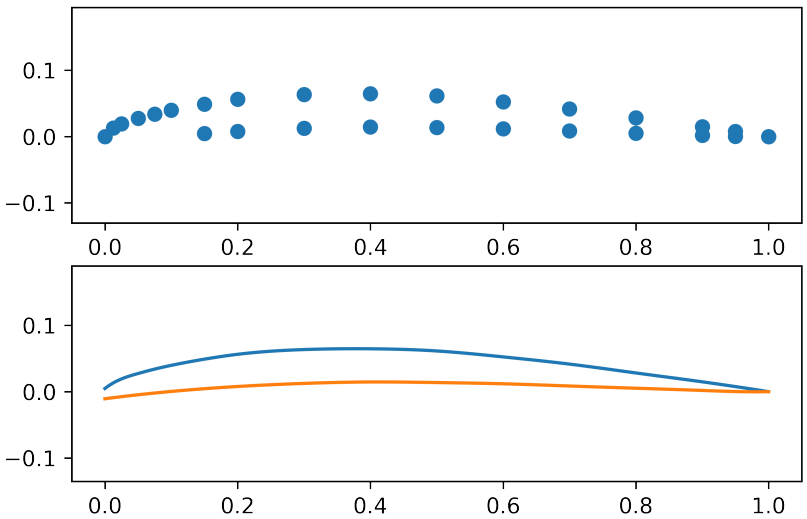
[1] ссылка на имплементацию

Препроцессинг

При приближении может возникнуть множество артефактов, которые необходимо корректно обработать:



(a) Артефакт приближения



(b) Удачное приближение

Имплементация нейросети

В качестве основного фреймворка я использовал PyTorch. JupyterNotebook со всей необходимой реализацией можно найти по этой ссылке.

Упомяну только особо специфические детали реализации:

- клиппинг – значения параметров в кросс-энтропии пришлось ограничить, чтобы избежать NaN при попытке взять логарифм от нуля.

```
L_gan = torch.log(torch.clamp(y,           min=1e-2)) + \
        torch.log(torch.clamp(1-y_tilde,   min=1e-2)) + \
        torch.log(torch.clamp(1-y_hat,     min=1e-2))
```

Имплементация нейросети

- Нормализация σ – чтобы матрица ковариаций была положительно определена, пришлось возвести σ в квадрат

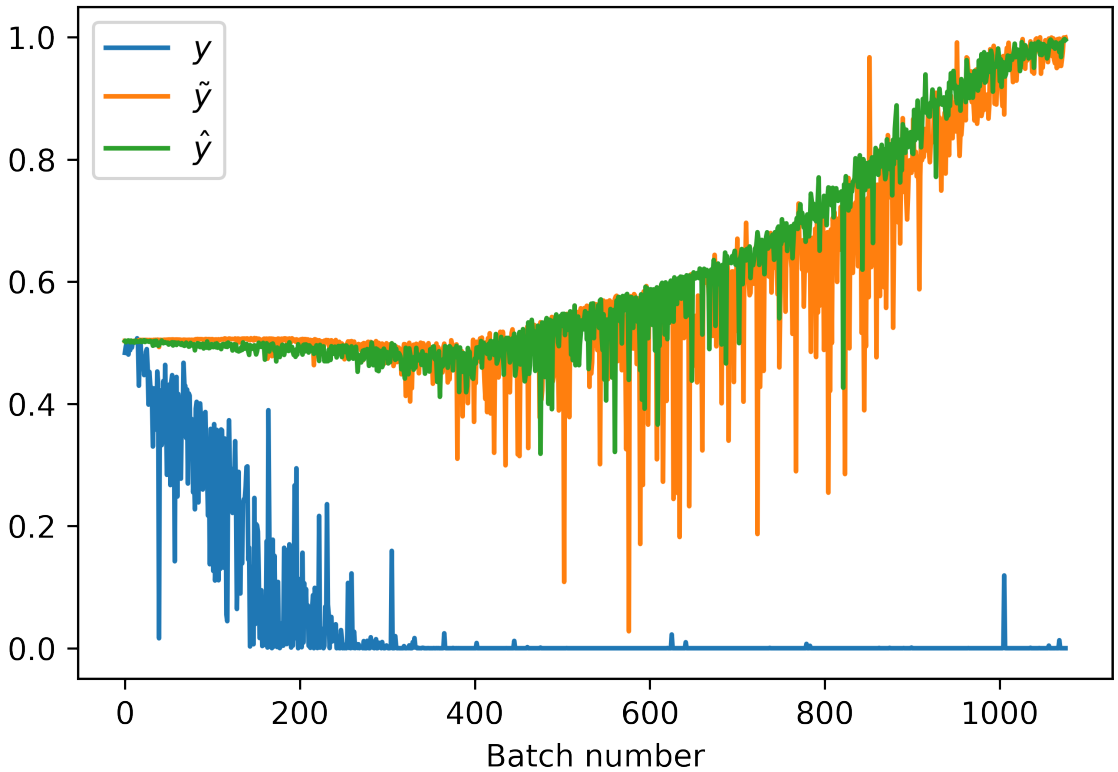
```
mu = x[:l_dim]
sigma = x[l_dim:]**2

# Kullback-Leibler Divergence
self.kl = (sigma**2 + mu**2 - torch.log(sigma) - 1/2).sum()
```

Конечно, можно было взять $|\sigma|$, но особой разницы нет – x^2 довольно здорово приближается перцептроном вблизи 0.

Обучение нейросети

В качестве оптимизатора я выбрал `Adam`. Модель обучалась на протяжении 10^3 эпох, причем каждые 100 эпох создавался бэкап.



Заключение

Спасибо за внимание:)