Understanding GAN

Gaspard Ulysse Fragnière

August 2022

1 Discriminator

- When given a sample, the discriminator produce a preditcion, i.e

$$\text{prediction} = \text{discrimantor}(\text{sample}) \tag{1}$$

- A prediction is a "matrix" of size 1×1, hence a scalar. If its value is bigger than 0, it means the the discriminator considered the given sample as legit

2 Generator

- When the generator is trained, the function "predict" can be used. It needs to be provided a random vector of dimension "latent_dim". This vector act as a seed to produce a sample,

$$\text{sample} = \text{generator}(\text{seed}) \tag{2}$$

- test

3 GAN

- The training of the GAN happens when “fit” is called
- When fit is called, the function "train_step" is called iteravely.
- the function "train_step" does the following:
  - the generator generate eigenvectors using the random seed
  - the fake eigenvectors are normalized
  - Combine fake (i.e. generated) eigenvectors and real eigenvectors, resulting in an array of dimesion: eigenvector_size $\times$ 2batch_size.
  - we then produce the groundtruth labels for these eigen vectors, i.e. a vector of dimension batch_size full of ones, followed by a vector of dimension batch_size full of zeros.
  - train the discrimantors, i.e. make the discriminator guess what eigenvectors are real, compute d_loss.(guess, groundtruth) and use it to update weights (**TODO: read code again here**)

- In order to train the generator this time, we produce another random seed and a vector full of zero of dimension batch_size to be the new groundtruth
- We then provide this second seed to the generator, in order to produce batch_size fake eigenvectors.
- We feed those fake eigenvectors and save its prediction.
- We compute the g_loss (new groundtruth, prediction of the discriminator).
- We use g_loss to update the weights of the generator.
- Finally we monitor the losses (**Understand what that means**)

4 The data

We need to understand what is the shape of the data:

4.1 Data used for the training:

- **CSM**: The cross spectral matrix is computed with

$$\mathbf{C} = \mathbf{p}\mathbf{p}^H \tag{3}$$

  Where $\mathbf{p} \in \mathbb{C}^M$ is the complex sound pressure vector in the array of M microphones. The CSM can approximated by:

$$\hat{\mathbf{C}} = \frac{1}{B}\sum_{b=1}^{B} \mathbf{p}\mathbf{p}^H \tag{4}$$

  With B snapshots. The size of the CSM is then $M \times M$. Its eigendecomposition contains M eigenvalues and associated eigenvectors.

- **CSMTRIU**: a compressed version of the CSM. The compression is done using the fact that the CSM is hermitian (i.e. a complex squared matrix that is equal to its complex conjugated transpose, i.e. $a_{ij} = \bar{a}_{ji}$)

- **Eigendecomposition** The size of the CSM is then $M \times M$. Its eigendecomposition contains M eigenvalues and associated eigenvectors, and can be written as

$$\hat{\mathbf{C}} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^H \tag{5}$$

  where $\mathbf{V} = [\mathbf{v}_1^T, \ldots, \mathbf{v}_M^T]$, $\mathbf{v}_i$ being the ith eigenvector and where $\mathbf{\Lambda}$ is a diagonal matrix, where λ_{ii} is the ith eigenvalue. Currently the eigendecompistion is performed automatically by the parser function in the load_data.py file.

Currently the data provided for the training are the eigenvectors $[\mathbf{v}_1^T, \ldots, \mathbf{v}_M^T]$. It needs to be extended to generate also the eigenvalues $[\lambda_{11}, \ldots, \lambda_{MM}]$.

More specifically, the data provided for training comes in batches of size batch_size. Hence the data provided for training has **dimension**: batch_size $\times M \times M \times 2$ (real, imaginary).

4.2 Generator

- **in**: random vector of dimension latent_dim. **Dimension**: batch_size × latent_dim.
- **out**: eigenvectors matrix of dimension $M \times M$ **Dimension**: batch_size $\times M \times M \times 2$

4.3 Discriminator

- **in**: eigenvectors matrix of dimension $M \times M$. **Dimension**: batch_size $\times M \times M \times 2$
- **out**: $0, 1$, "boolean". **Dimension**: batch_size

4.4 How to improve a GAN

from https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b
Common pitfalls:

- Non-convergence
- Mode Collapse

4.5 Wasserstein GAN

-> see article: https://machinelearningmastery.com/how-to-implement-wasserstein-loss-for-generative-adversarial-networks/

In a WGAN, the discriminator is replaced by a critic. Unlike a discriminator that say if a sample is fake or not, the role of a critic is to quantity the level of fakeness (or realness) of a receive sample. More formally, a critic produce an output in $\{0, 1\}$ and a critic in $[0, 1]$. This simple change is supposed to have a great impact on the convergence of a model.

Implementation of a WGAN requires a few changes from regular implementation, i.e.

- Use a linear activation function in the output layer of the critic model (instead of sigmoid).
- Use Wasserstein loss to train the critic and generator models that promote larger difference between scores for real and generated images.
- Constrain critic model weights to a limited range after each mini batch update (e.g. [-0.01,0.01]).
- Update the critic model more times than the generator each iteration (e.g. 5).
- Use the RMSProp version of gradient descent with small learning rate and no momentum (e.g. 0.00005). (quote: "... we report that WGAN training becomes unstable at times when one uses a momentum based optimizer such as Adam [...] We therefore switched to RMSProp ...")

4.5.1 Wasserstein Loss:

How to implement a Wasserstein loss:

- **Goal:** increase the gap between the scores for real and generated images
- **Critic Loss**: difference between average critic score on real images and average critic score on fake images
- **Generator Loss** the negation of the average critic score on fake images
-