Deep Learning in Finance

GANs for time series generation

Damien Challet
damien.challet@centralesupelec.fr

5th November 2024

Timeseries generation

1. Bootstrap
2. Parametric model
3. **GAN:** generative adversarial network

1. Bootstrap

- Sample set

$$\text{sample}\,\{r_i\}_{i=1,\cdots,N}$$

- Bootstrap: random resampling with replacement

$$\tilde{r}_i^{(b)} = r_{b_i}$$
$$b_i \sim \{1, \cdots, N\}$$

- Obvious fact:

$$P(\tilde{r}) \overset{\text{law}}{=} P(r)$$

- Less obvious: given estimator F, std from data = std from bootstrapped data

$$E(\text{std}[F(\{\tilde{r}\})]) = E[\text{std}[F(\{r\})])$$

Naive bootstrap

- random resampling with replacement from the sample set

$$\begin{aligned} \tilde{r}_i^{(b)} &= r_{b_i} \\ b_i &\sim \{1, \cdots, N\} \\ b_i &\quad \text{i.i.d.} \end{aligned}$$

- Problem:
  - r_t not i.i.d.
  - $|r_t|$ not i.i.d.

Block bootstraps

- Block length L $\rightarrow$ chunks of length N/L
- Bootstrap the chunks
- Rule of thumb:

$$L \geq \text{memory length of process } X$$

- Rule of thumb:

$$\text{memory length of process} \sim \int_1^\infty C_X(\tau)d\tau$$

- Volatility:

$$\int_1^\infty C_{|r|}(\tau)d\tau = \infty$$

Parametric model

- Stochastic volatility σ_t modulates fluctuations

$$r_t = \sigma_t \eta_t \equiv \epsilon_t$$
$$\eta_t \sim \mathcal{N}(0, 1)$$
$$\epsilon_t \sim \mathcal{N}(0, \sigma_t)$$

- Volatility contains all the complexity of the dynamics

Parametric model: properties

- ACF(r_t) $= 0$
- ACF($|r_t|$)=ACF(σ_t)
- ...

- Stochastic volatility: self-referential + external noise component
  - GARCH
  - Heston
  - ZGARCH [link]
  - Quadratic Hawkes processes, ZHawkes [link]
  - rough (fBm $H \simeq 0.15$) [link]
- Factor models

- **DL?**

Generative Adversarial Networks

Electronic Frontier Foundation report [source]

Generative Adversarial Networks

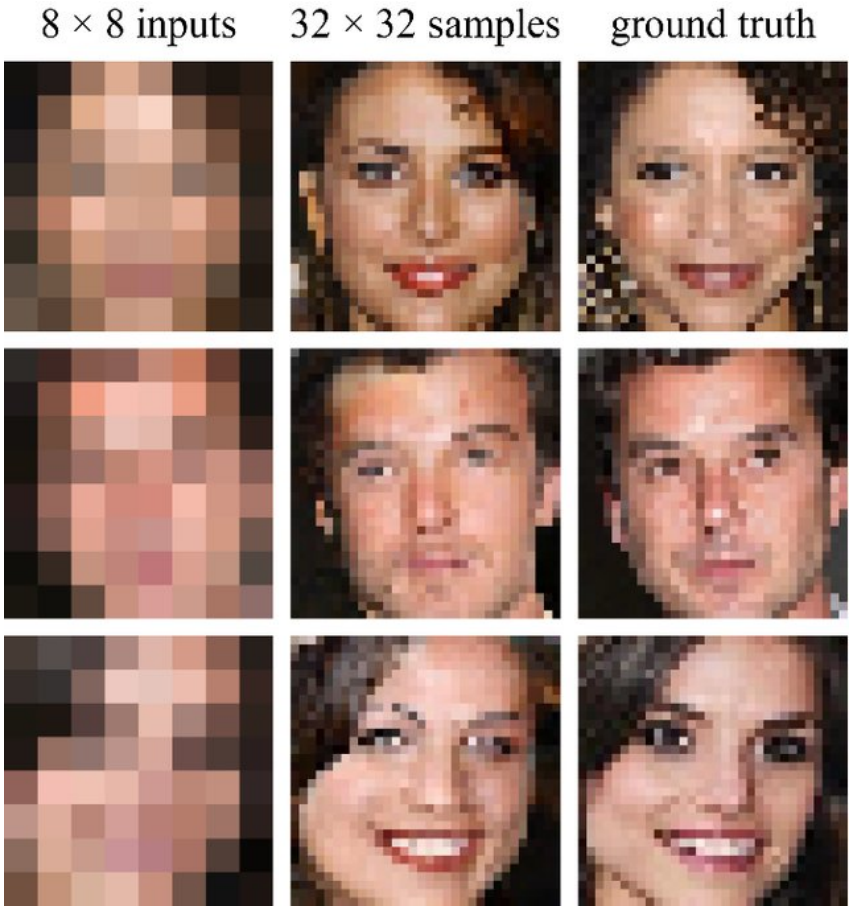
18 Impressive Applications of Generative Adversarial Networks (GANs) [gallery]

- Faces
- Colors
- Missing parts of images
- Images from hand drawings
- Superresolution
- ...

Curated github list [gans-awesome-applications]

Superresolution

[source]



GAN in finance

- Anonymize data
  - generate look-alike data, but do not publish original data.
- Data augmentation
  - financial timeseries are not very many and diverse
  - classification problems: *train on fake, trade on real*
- Challenge:

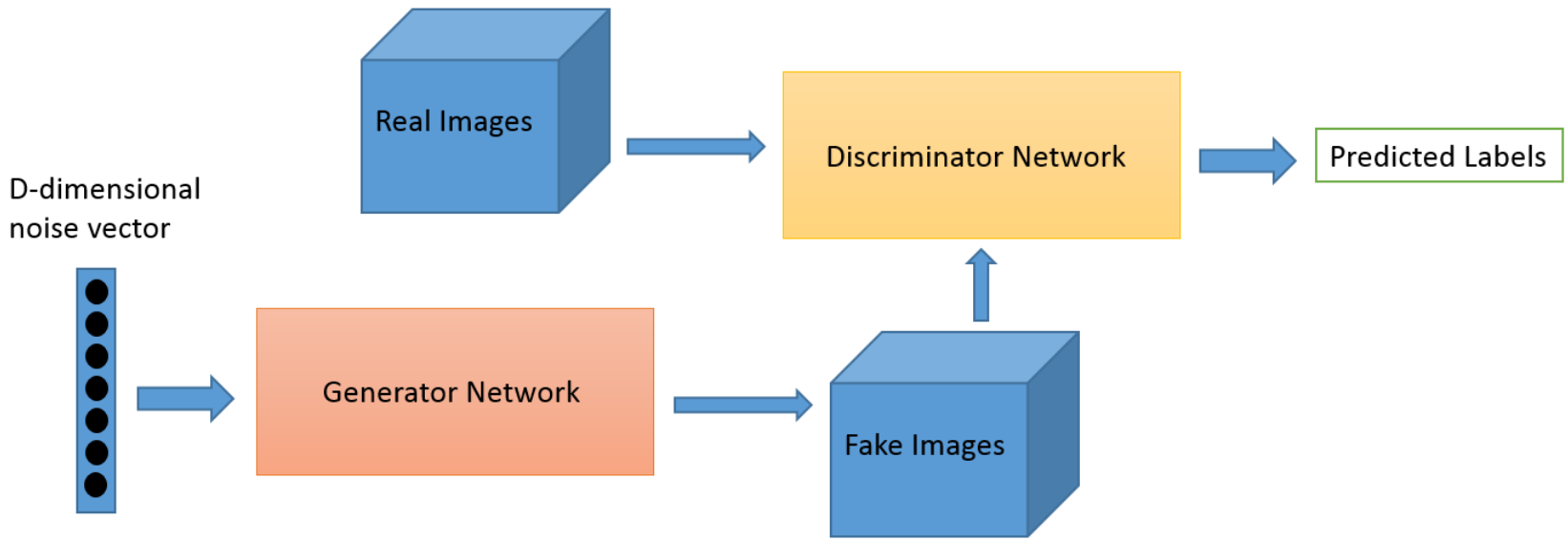
  i.i.d. Gaussian noise $\rightarrow$ heavy-tailed noise with memory

Financial time series generation with DL

- Review: [review]
- Pardo et al. VIX [MSt thesis] [paper]
- Takahashi et al. : MLP (Dense) / CNN [paper]
- Wiese et al.: TCNs [link]
- Buhler et al. signatures [link]
- Ni et al.: Conditional Sig-Wasserstein GANs [preprint]
- Paul et al.: PSA-GAN [preprint]
- Rusnak et al.: attention too [MSt thesis]
- FinGAN [preprint]
- tailGAN [preprint]
- evGAN [preprint]
- COT-GAN [preprint]
- CEGEN [preprint]
- Schrödinger bridge [preprint]

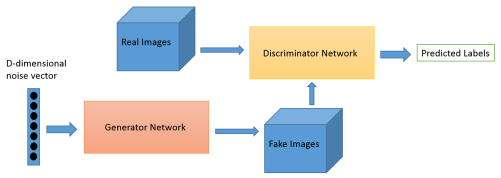
Generic GAN workflow

Goodfellow (2017) [link] [image source]



Generic GAN workflow

Goodfellow (2017) [link] [image source]



- Discriminator learns to differentiate real and fake sample
- Generator G learns to fool discriminator D
- Alternate learning
- Aim: discriminator $P(\text{real}) = P(\text{fake}) = 1/2$ (or 1?)
  - G must be good enough to fool D
  - D must be good enough to force G to learn
  - D must not be too good: convergence? learning rate?

Generator from representation

- From vector in feature space $\eta \in \mathbb{R}^{N_F}$

$$G(\eta): \text{ object}$$

- Example

$$G(\eta) \in \mathbb{R}^{T}: \text{ timeseries}$$

- Backpropagation $\rightarrow$ weights so that $\{G(\eta)\}\sim\{\text{real data}\}$

Discriminator: classification problem

- Given input x_i, discriminator

$$D(x_i) \in [0,1] = P(x_i \text{ is true})$$

- Notations:
  - $p_i = 1$ if x_i is true, $p_i = 0$ if x_i is fake
  - $\hat{p}_i = D(x_i)$

- Loss: increasing function of $p_i f(\hat{p}_i)$ and $(1 - p_i) f(1 - \hat{p}_i)$

Discriminator: classification problem

- Loss: increasing function of $p_i f(\hat{p}_i)$ and $(1-p_i) f(1-\hat{p}_i)$
- Cross-entropy

$$p_i \log(\hat{p}_i) + (1-p_i) \log(1-\hat{p}_i)$$

- batch of M samples

$$\mathcal{L}_D = \sum_{i=1}^{M} p_i \log(\hat{p}_i) + (1-p_i) \log(1-\hat{p}_i)$$

- learn classification $\equiv$ minimize $\mathcal{L}_{\mathcal{D}}$. Perfect learning if $\mathcal{L}_{\mathcal{D}} = 0$
- Note: works well for balanced datasets.
  For unbalanced ones, see [focal loss]

Cross-entropy for the generator

- Aim: maximize the probability that the discriminator predicts true given fake data
- Generator: from random vector $\eta_i \in \mathbb{R}^{N_F}$, $G(\eta_i) = x_i \in \mathbb{R}^T$
- $G(\eta_i) = x_i$ is fed to discriminator $\rightarrow D(x_i)$
- Cost function of M samples

$$\begin{aligned}\mathcal{L}_G &= \sum_{i=1}^{M} \log\left(1 - \hat{p}_i(x_i)\right) \\ &= \sum_{i=1}^{M} 1 \times \log\left(1 - \hat{p}_i[G(\eta_i)]\right)\end{aligned}$$

- $\mathcal{L}_D$ is differentiable $\rightarrow$ backpropagation

Generic GAN workflow

Goodfellow (2017) [link]: define

1. feature space $\eta \in R^{N_F}$
2. generator (NN): $\eta \rightarrow$ fake object
3. discriminator (NN): fake object $\rightarrow$ classification
4. discriminator:
   - 4.1 generate M samples from generator
   - 4.2 take M samples from real data
   - 4.3 ask discriminator to classify fake and real data
   - 4.4 compute loss of discriminator $\rightarrow$ modify weights of discriminator
5. generator
   - 5.1 generate M samples from generator
   - 5.2 compute loss of generator $\rightarrow$ modify weights of generator
6. Repeat 4 and 5.

GANs for financial timeseries

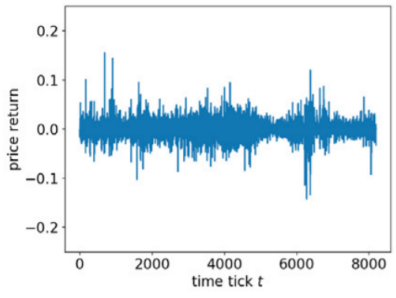
Literature:

- $D-$dimensional noise vector $\rightarrow$ generator $\rightarrow$ T timesteps
- timeseries$\rightarrow$ discriminator
  - TRUE / FAKE
- Stylized facts: $P(r_{GAN}), C_{r_{GAN}}, C_{|r_{GAN}|}$?

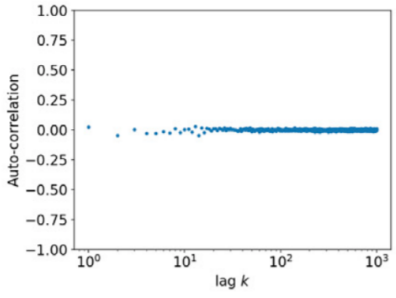
Example: GANs with Dense (MLP) networks

Takayashi *et al.* (2019) [paper]

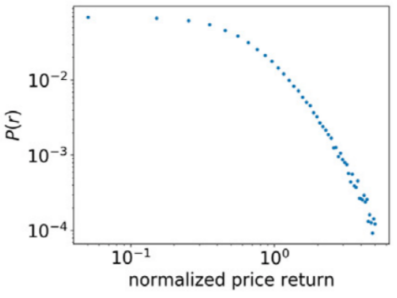
- $N_F = 100,\ T = 8192$



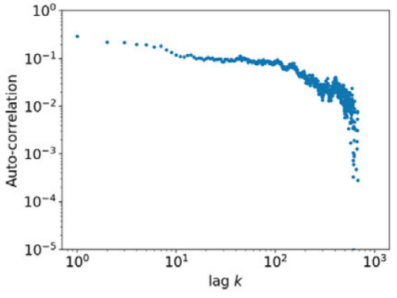
(a) time-series



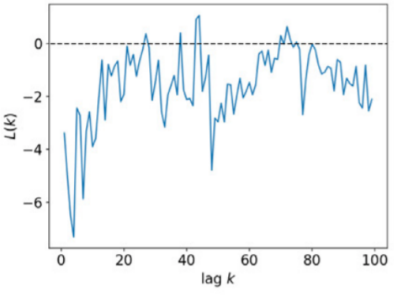
(b) linear unpredictability



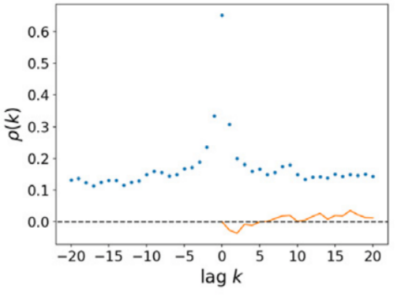
(c) heavy-tailed distribution



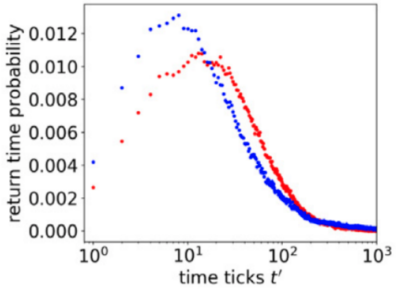
(d) volatility clustering



(e) leverage effect



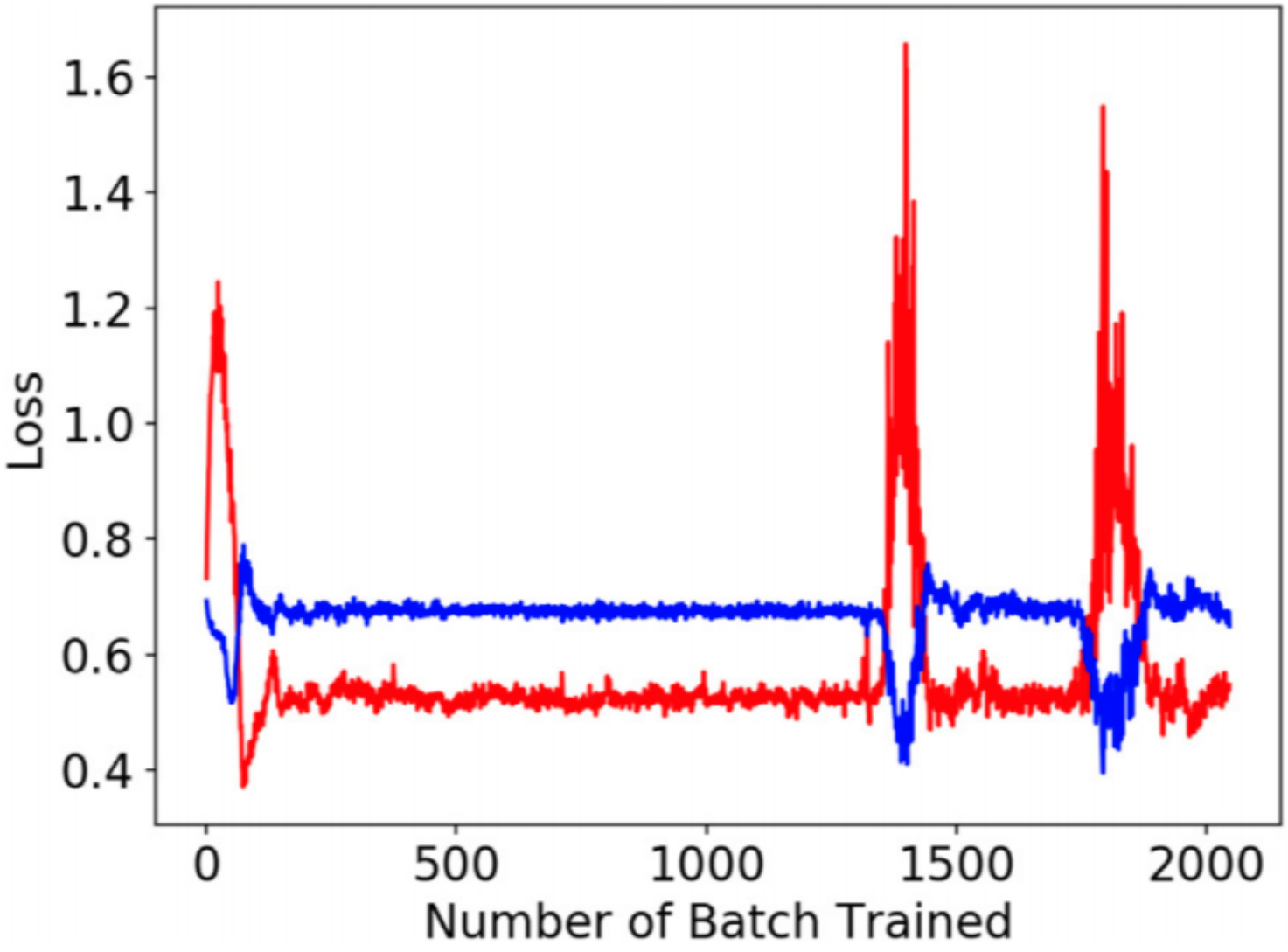
(f) coarse-fine volatility correlation



(g) gain/loss asymmetry

Generic problem with GAN: convergence

Takayashi *et al.* (2019) [paper]



The devil is in the details

discriminator D. See Algorithm 1 for the detail. Adam [53] was chosen as the stochastic gradient algorithm to update the parameters of the generator θ_G and the discriminator θ_D. The learning rate is set to $\alpha_G = 2 \cdot 10^{-4}$ and the parameter $\beta_{1,G} = 0.5$ for the generator and the learning rate $\alpha_D = 1 \cdot 10^{-5}$ the parameter $\beta_{1,D} = 0.1$ for the discriminator. The rest of the hyper-parameters of Adam are set to the originally proposed values. Batch normalization [54] is applied to the MLP and CNNs layers of the generator. The effect of batch normalization is investigated with the MLP generator. A technique called noisy labeling [55] is also used in the implementation, with which noise are added to the labels for D. Uniform distribution from 0.9 to 1.1 and from 0.1 to 0.3 are used for the labels of dataset and generated time-series, respectively. The batch size is set to $m = 24$. The batch of the real data of 8192 steps is randomly fetched from the dataset. The most of the hyper-parameters are taken from implementations of GANs for image generation and are unchanged from the default values. Learning rates of Adam optimizers are tuned by hand to maintain the robustness of the training process. Excessively large values of learning rates lead the model to fail to be well trained.

Note:

- learning rate of discriminator $<$ learning rate of generator
- β = local memory of gradients. Memory length $\sim 1/\beta$
- Not my definition of robust

Adam optimizer

- Adam optimizer (Kingma and Ba 2014 [link]): moving average of gradient g_t

- tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, amsgrad=False, name="Adam", **kwargs)
  - β_1 : exponential moving average rate of g_t

$$m_{t+1} = m_t(1-\beta_1) + \beta_1 g_{t+1}$$

  - β_2: exponential moving average of g_t^2

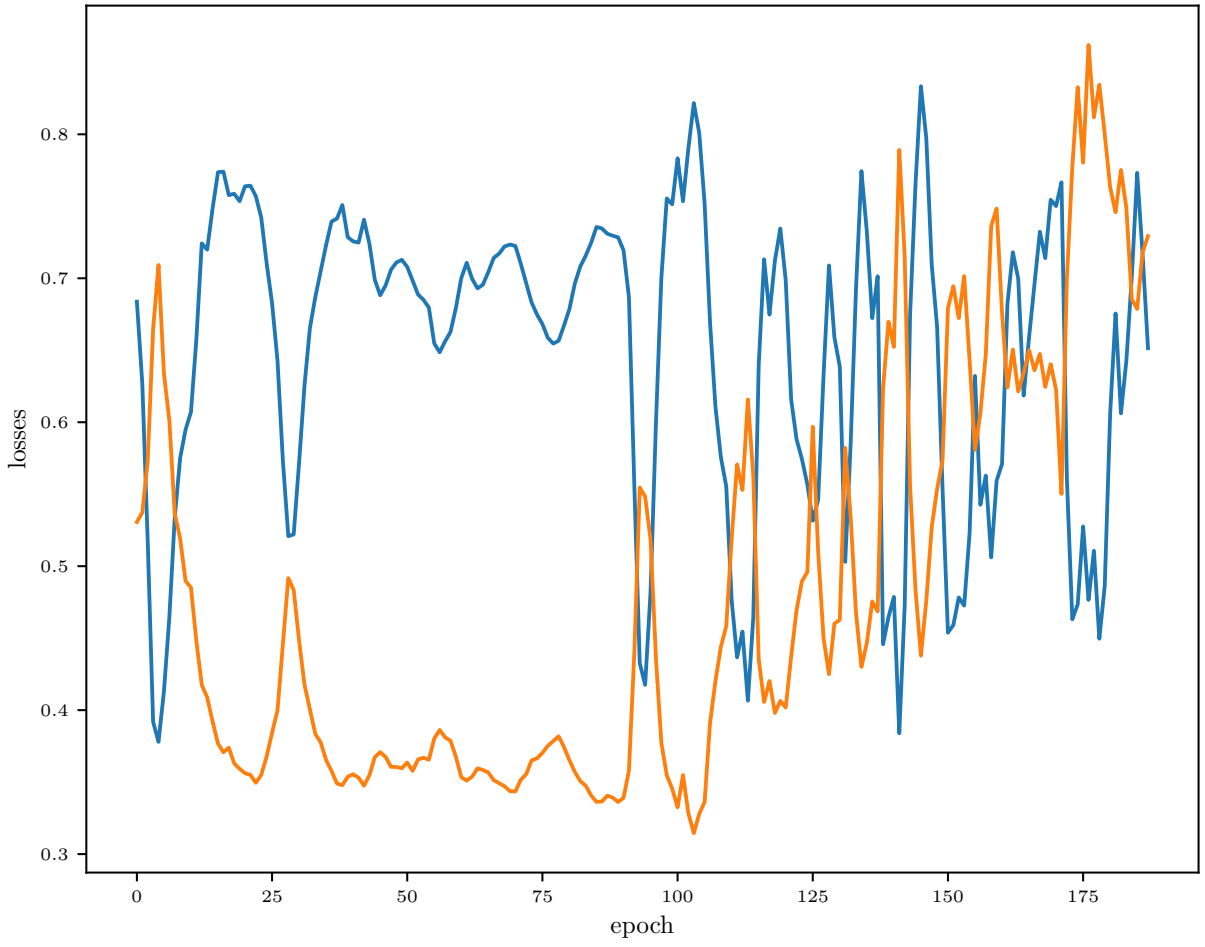
$$v_{t+1} = v_t(1-\beta_2) + \beta_2 g_{t+1}^2$$

  - update (for large times): roughly

$$\theta_{t+1} = \theta_t - \alpha \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon}$$

Convergence problems: unstable $(\frac{1}{2}, \frac{1}{2})$

MLP `use_bias=False` for both G and D.



How to improve convergence

Convergence problems $\equiv$ game problems $\equiv$ design problems

- Adapt G and D architecture
  e.g. MLP + `use_bias=False` for both G and D
- Noisy labelling (Salimans *et al.* 2016 [paper]):
  randomize a fraction of correct / false labels
- Better loss: compare full distributions of $\hat{p}_i$
  WassersteinGAN [link]
- Relativistic losses (next week)

GAN: problems specific to finance

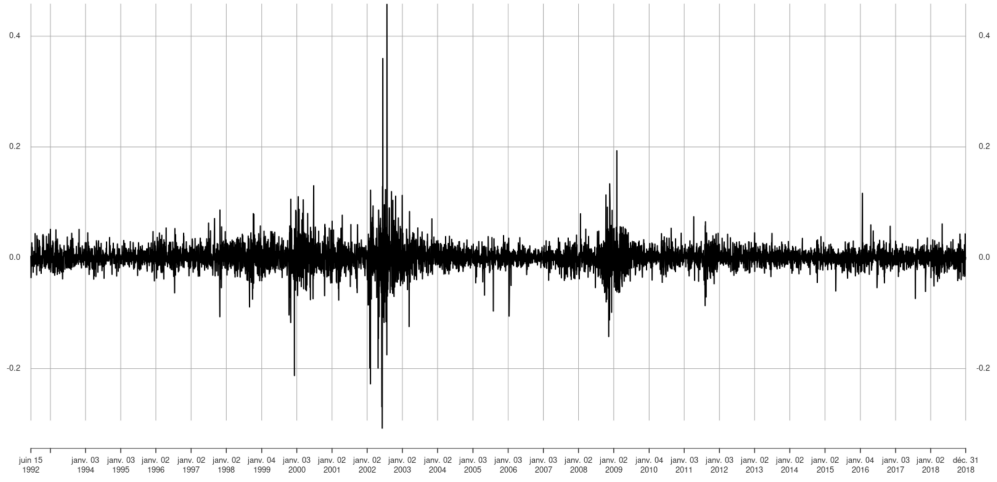
Challenge

How to generate realistic time series ?

Realistic $\equiv$ stylized facts, realistic dynamics

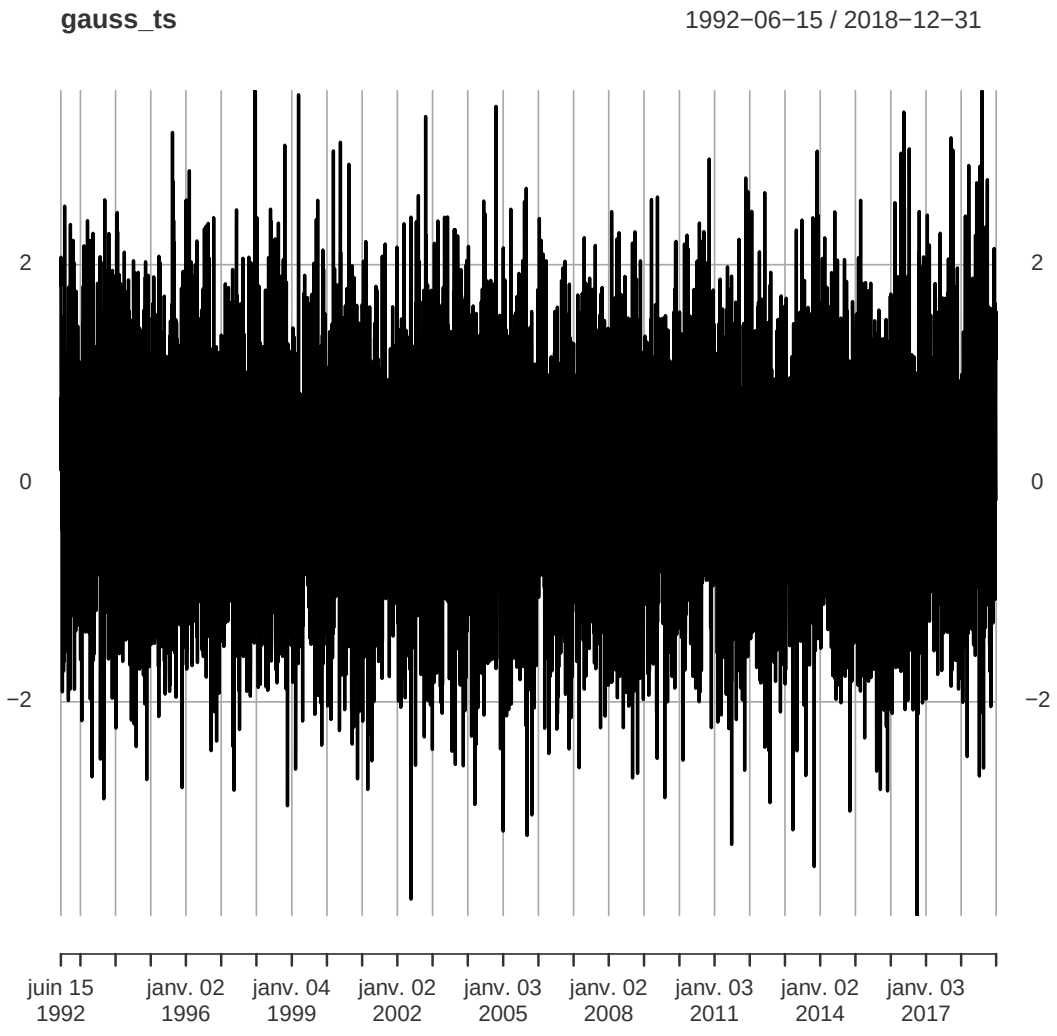
Basic stylized facts

- Heavy-tailed price returns
- Clustered volatility



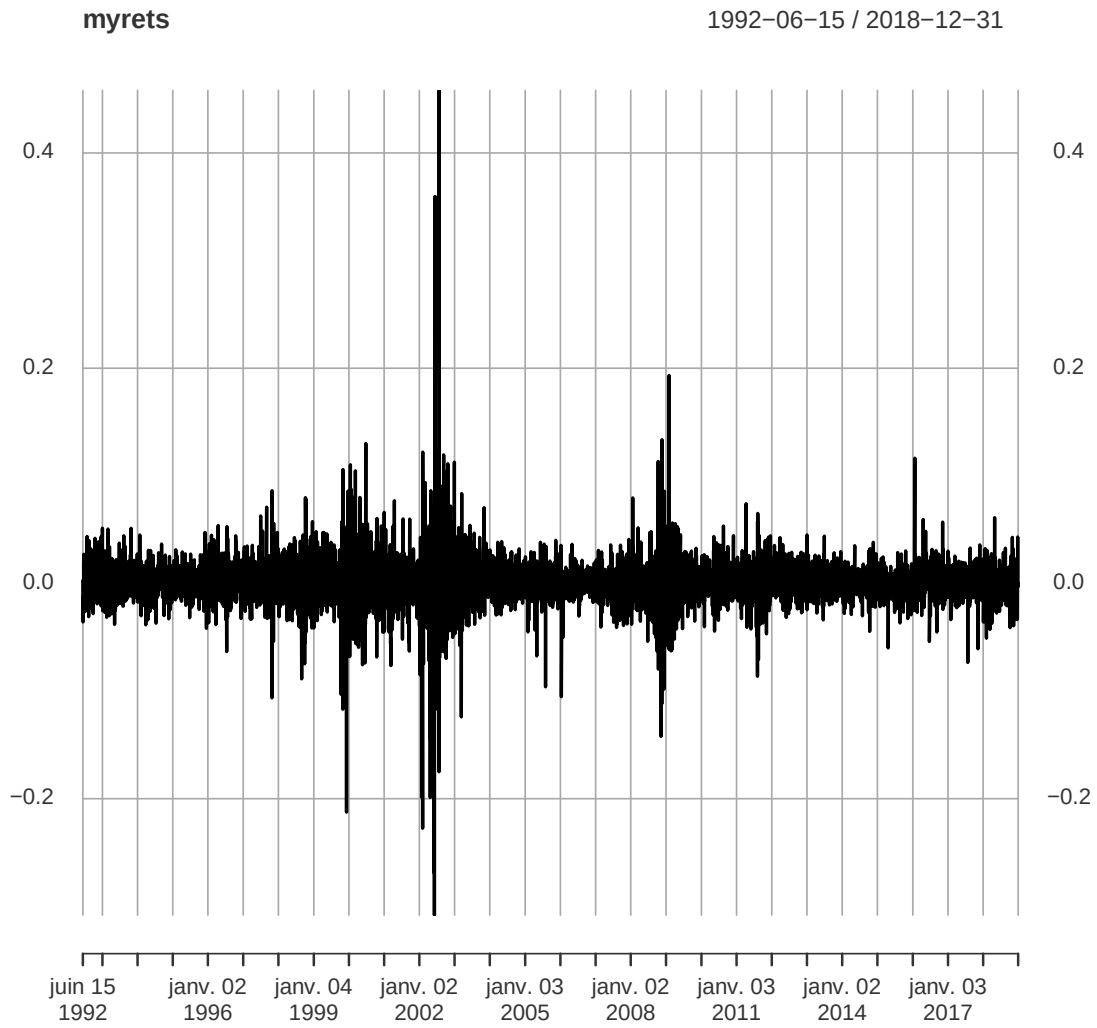
Visual inspection

Gaussian returns (simulated)



Visual inspection

Real-life price returns



Heavy-tailed price returns

- Large price returns
- Largest price returns are many times larger than typical price return
- Reminder: $P(x)$ is *heavy-tailed* $\iff$

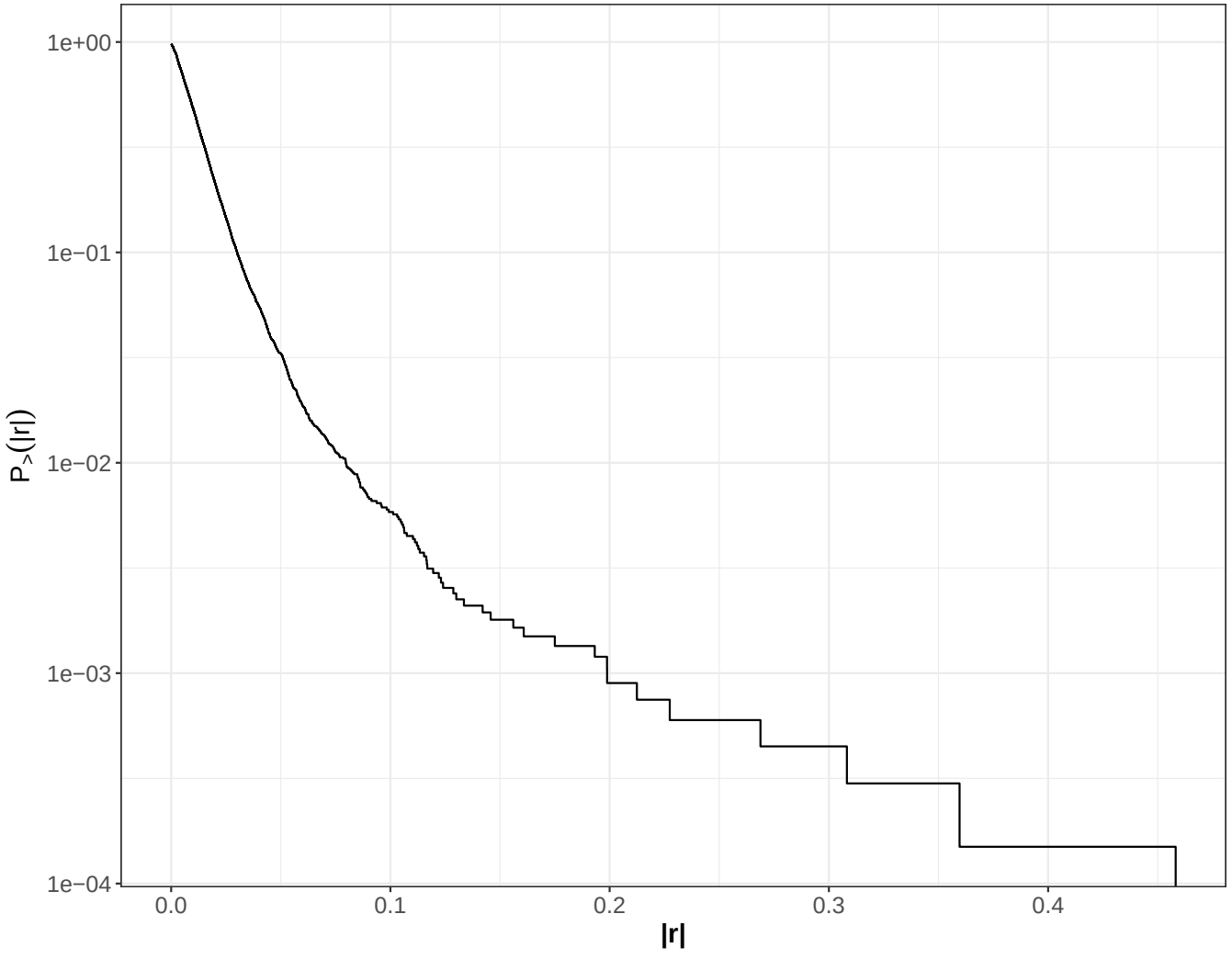
$$\lim_{x \to \infty} e^{\lambda x} P(x) = \infty \qquad \forall \lambda > 0$$

decays much more slowly than any exponential

- $P(x) \propto e^{-\kappa x} \rightarrow P_{>}(r) \propto e^{-\kappa x} \rightarrow \log P_{>}(r) = cst - \kappa x$

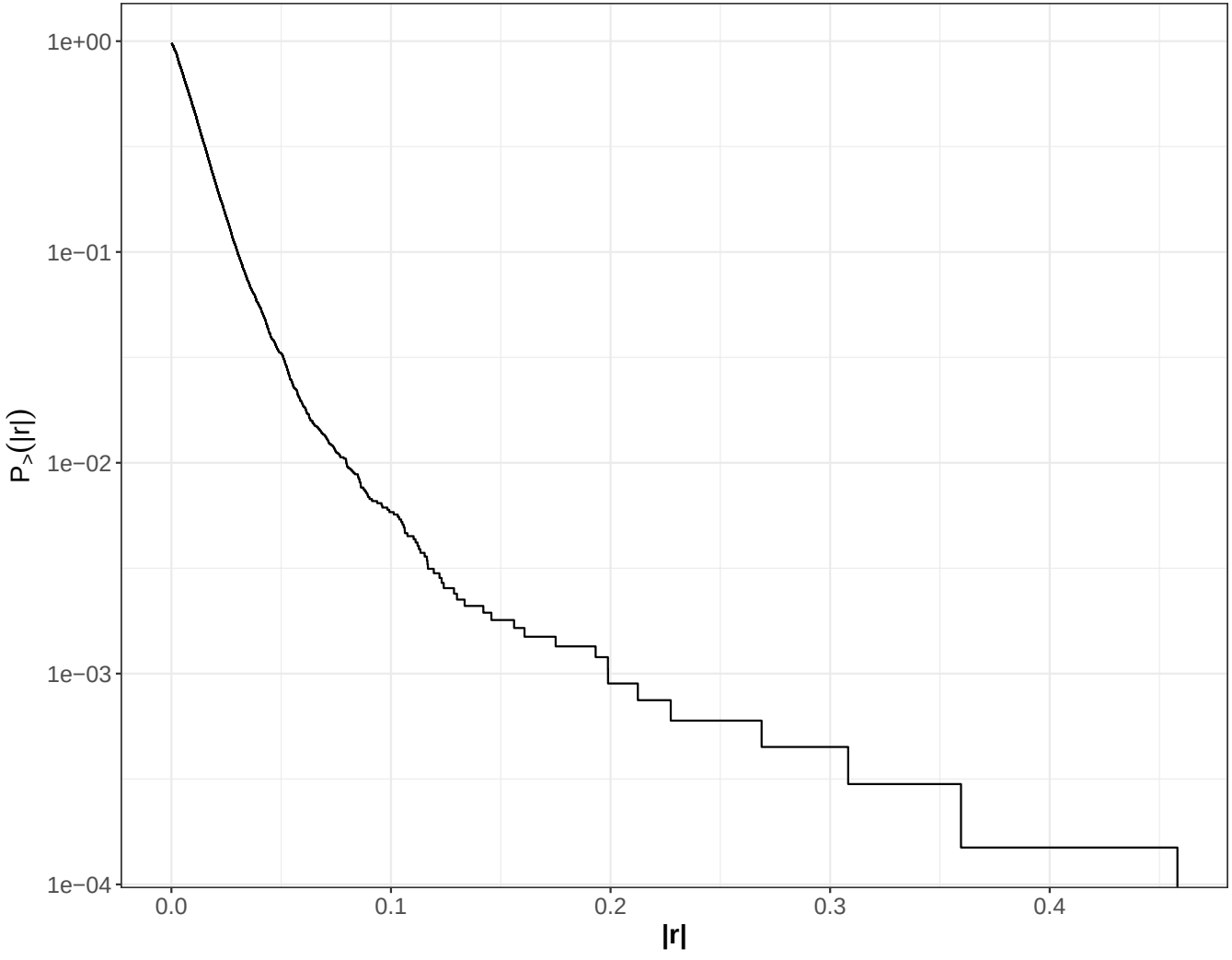
Visual inspection of $P_>(|r|)$

Reciprocal empirical cumulative distribution function (recdf)



Visual inspection with $P_>(|r|)$

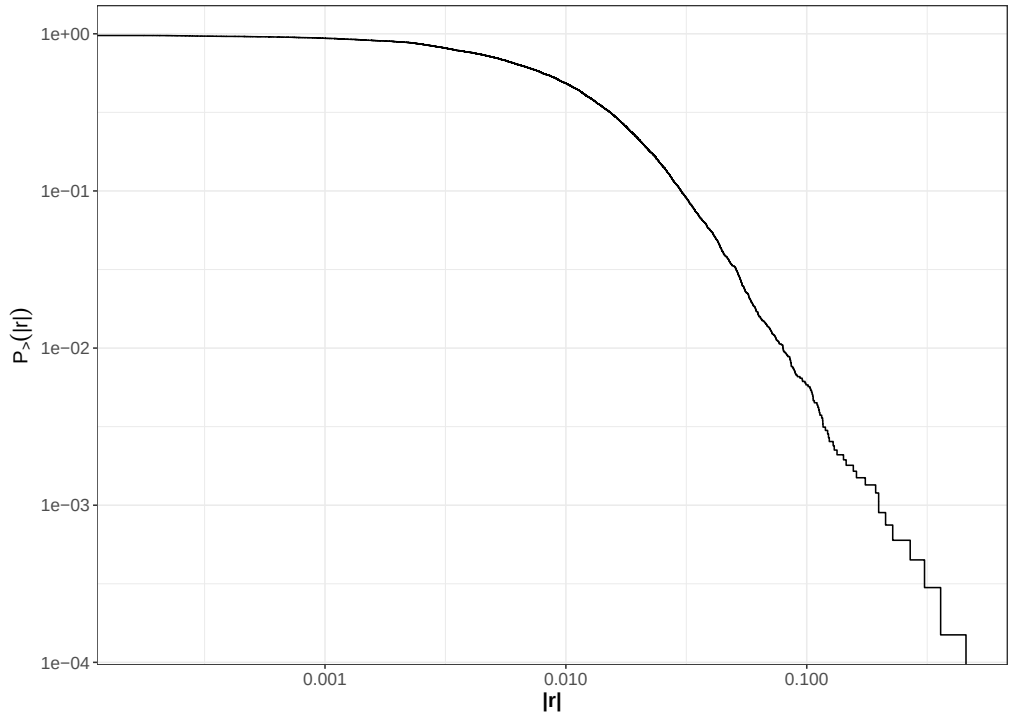
heavy-tailed $P(x) \iff \log P_>(x)$ is convex.



Price returns: log-log plots

If $P(|r|) \propto |r|^{-\alpha-1}$, $P_>(|r|) \propto |r|^{-\alpha}$

$$\log P_>(|r|) = cst - \alpha \log |r|$$



Heavy-tailed distribution: theorems

[link] and [link]

1. CNNs cannot transform Gaussian noise into heavy-tailed noise

2. MLPs cannot transform Gaussian noise into heavy-tailed noise

Solutions:

1. Use another type of noise

2. Use another NN architecture

3. Transform data

How (not?) to generate heavy tails?

- Wiese et al. (2019): our NNs cannot generate heavy tails
- Method: inverse Lambert-W transform LW^{-1} $(t \to \log t)$
  - Instead of generating $\tilde{r}_t$ as close as possible to r_t,
  - define g_t, a Gaussianized version of r_t: $g_t = LW^{-1}(r_t)$
  - learn to generate $\tilde{g}_t$
  - $\tilde{r}_t = LW(\tilde{g}_t)$
- R @ CRAN: [LambertW]
  Python: [scipy.special.lambertw]

Clustered volatility

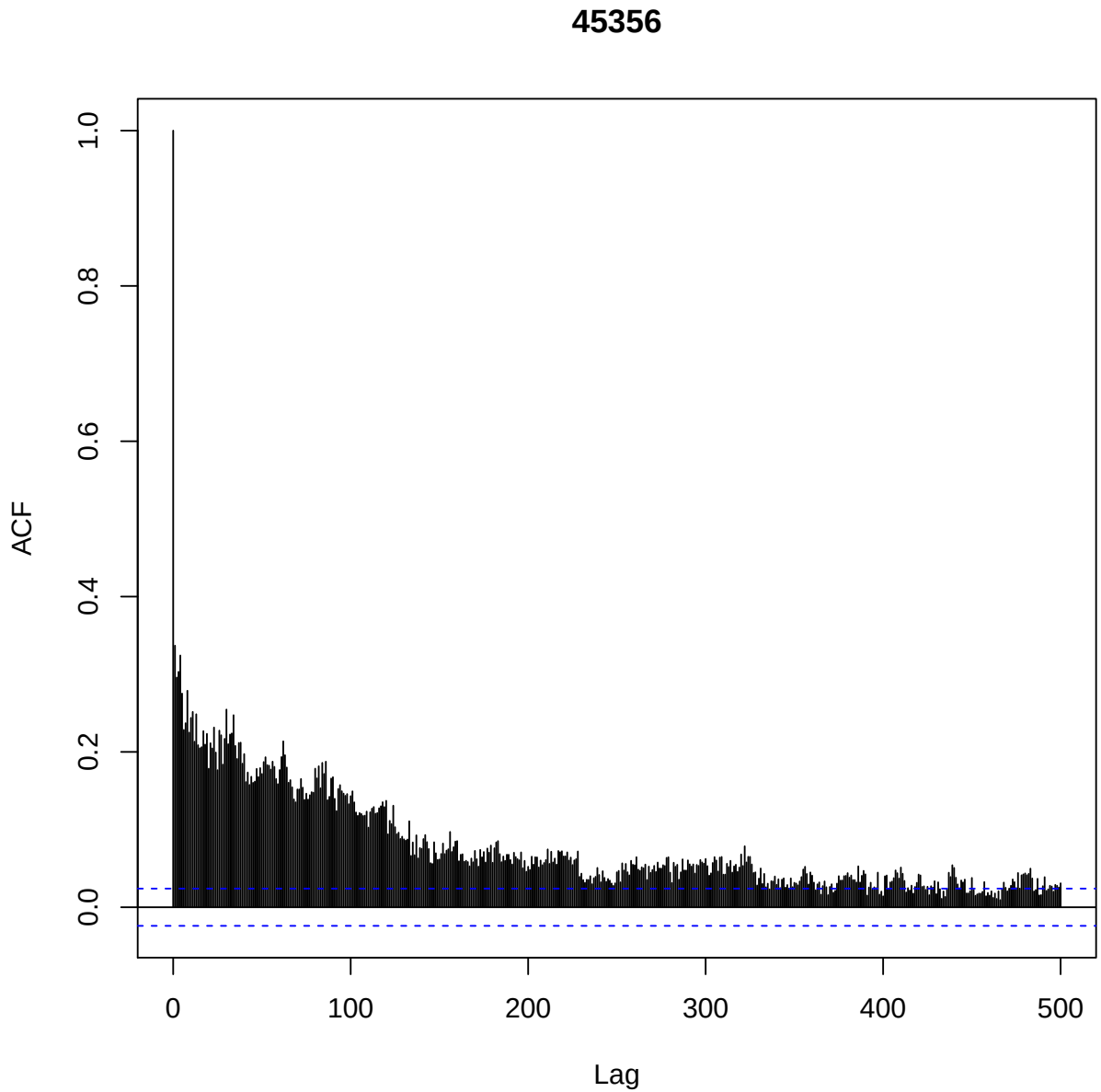
- Clusters of volatility: active and calm periods
- Volatility has a memory
- If

$$C_{|r|}(\tau) \propto (|r_t| - \overline{|r|})(|r_{t+\tau}| - \overline{|r|}) \propto \tau^{-\beta}$$

and if $\beta < 1$

$$\int_1^\infty C_{|r|}(\tau) d\tau \propto \int_1^\infty \tau^{-\beta} d\tau = \infty : \quad \text{'long memory'}$$

Clustered volatility: empirical check



Clustered volatility: empirical check

- If

$$y = Cx^{-\alpha}$$

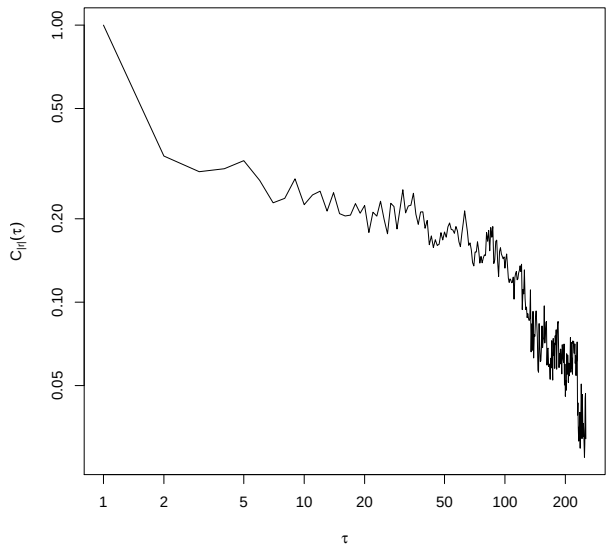
then

$$\log y = \log C - \alpha \log x$$

linear function

- Is $\log C_{|r|}(\tau)$ a linear function of $\log \tau$?

Visual check



GANs with TCNs

Wiese et al. (2019) [preprint]

- $T = 100$

- TCN: temporal convolutional networks $\equiv$ Wavenets (Oord et al. 2016 [preprint])
  stacked CNN with dilation ($\rightarrow O(\log N)$ coefficients)

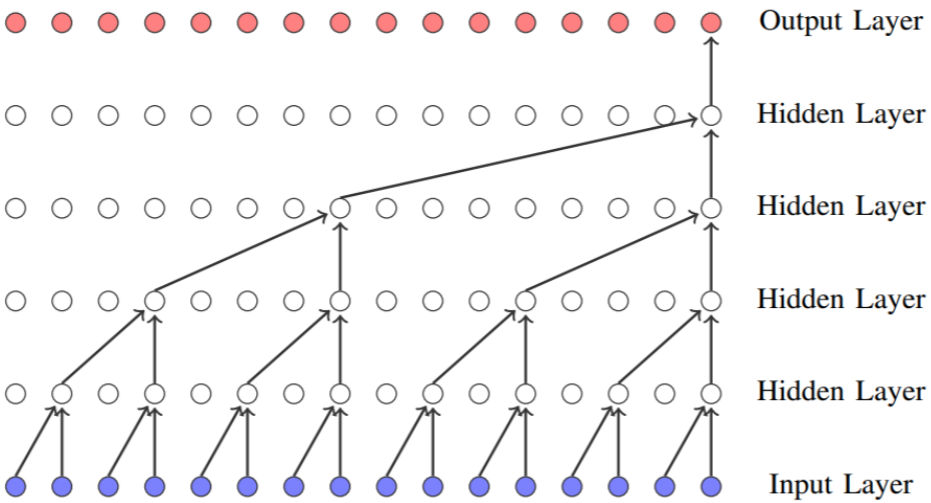


Figure 6: Vanilla TCN with 4 hidden layers, kernel size $K = 2$ and dilation factor $D = 2$ (cf. van den Oord et al. (2016)).

[keras-tcn]

What to generate?

Wiese *et al.*

- Everything: $NN \rightarrow r_t$
- Volatility and trend

$$\tilde{r}_t = \sigma_t \epsilon_t + \mu_t$$

- TCNs compute rolling non-linear averages of $\eta_t \in \mathbb{R}^{N_F = T \times D}$

More stylized facts

- Absence of price return autocorrelation
- Volume
- Volatility
- Leverage effect
- Intraday effects
  - correlation
  - volatility
  - time between trades
  - price jumps
  - flash crashes
  - ...

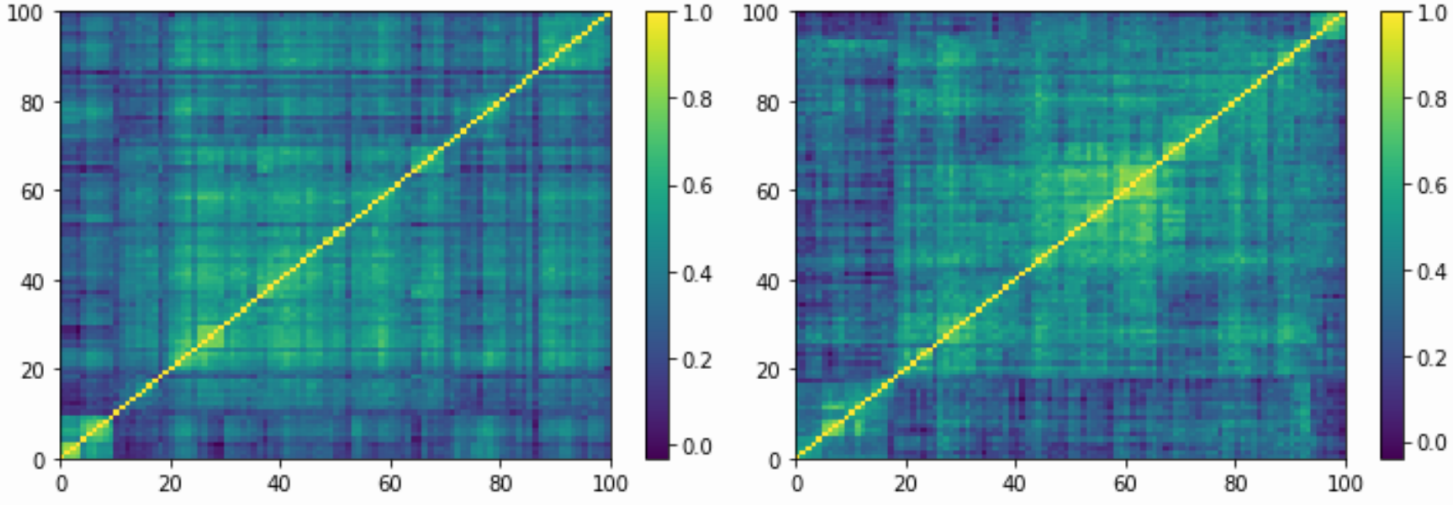
GAN for images: a sound methodology

1. Train a GAN with CNNs in an agnostic way
2. Convergence problems, solutions: happy
3. Look at images

It works because CNNs learn spatial features

Example: generating correlation matrices as images

Marti (2019) [preprint][link]



(Left) Empirical correlation matrix estimated on stocks returns; (Right) GAN-generated correlation matrix.

Nice looking, but

- generated matrices are not positive definite
- eigenvalue distributions not realistic

Conditional GANs

- feature vector encodes state (world, economy, wished object, ...)
- cGAN [preprint]
  - feature vector = past returns + random noise
  - simple MLP
  - stopping criterion: MSE
- Fin-GAN [preprint]
  - feature vector = past returns + random noise
  - generator loss: add PnL measures
- varying noise $\rightarrow$ scenarios

GAN for financial timeseries: a weird methodology

1. Train a GAN in an agnostic way
2. Convergence: happy
3. Look at timeseries.
4. Noise $\rightarrow$ test a set of stylized facts

Unconsistent aims and methodology

Example: heavy tails treatment in GAN literature

[review 2021]

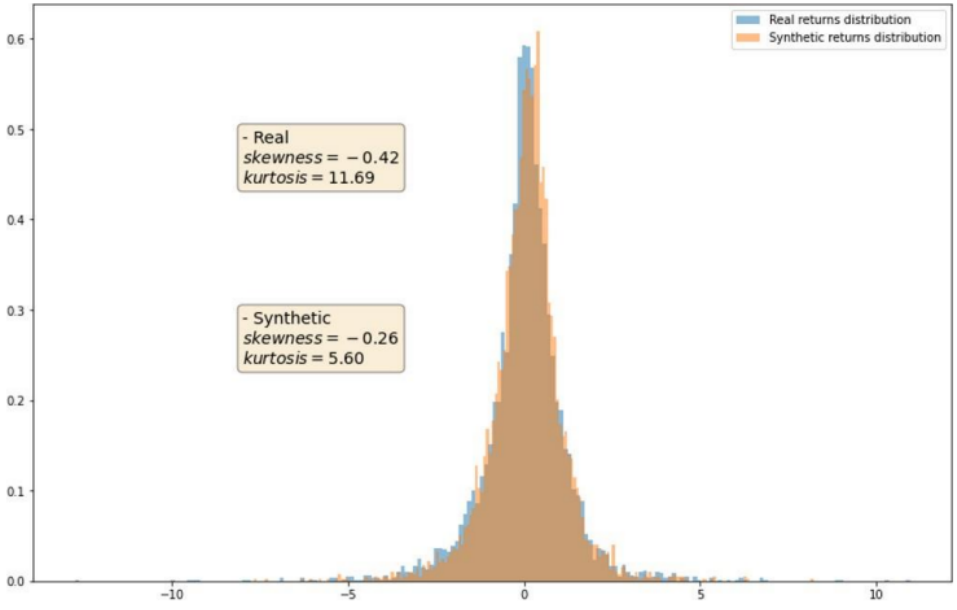
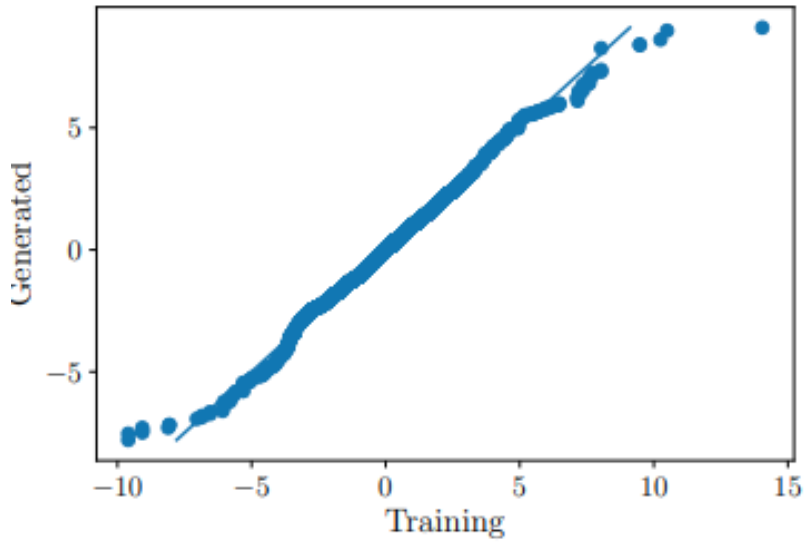


Figure 13: PDF - SAGAN

[Lemzi et al. 2021]



(b) Student's t-distribution with $\nu = 4$

Ask politely what you want

Include what you want in loss

- TailGAN (Cont *et al.* 2022 [link]): VaR, ES
  scenarios with consistent tail measures

Use variable transforms

- EV-GAN (Allouch *et al.* 2022 [link])
  generate heavy tails consistent with data

What does the discriminator learn?

Without help, the discriminator probably learns

- scale, average

Possibly

- Hurst exponent
- $\sum_\tau C_{|r|}(\tau)$

Not

- quantiles, distribution tail exponent (see EV-GAN)

Ask what you want: pre-processing

Pre-process data (remember: MLP cannot compute quantiles)

- Hurst exponent of $|r_t|$
- $\sum_{\tau=1} |C_r(\tau)|$: absence of linear autocorrelation
- $\sum_{\tau>0} C_{|r|}(\tau)$: long memory
- Non-Gaussian test (e.g. N-test [preprint])
- Heavy-tail test: quantiles ratio
- ...

Pre-processing: maths and stats

- math: $\rightarrow$ tf.math
  e.g. `np.sum()` $\rightarrow$ `tf.math.reduce_sum()`

- stats: import tensorflow_probability as tfp

  ```
  tfp.stats.auto_correlation()
  ```

How to help the discriminator: Lambda layers

- Python: lambda function
- Keras: Lambda layer: apply a function to the input of a layer
- Here: price returns $\rightarrow$ discriminator $\rightarrow$ statistics $\{a, b, c, \cdots\}$
- Discriminator: first layer is a Lambda layer that calls the function computing the statistics
- Technical hint: convert list [a,b,c] to tensor

```
return tf.transpose(tf.convert_to_tensor([a,b,c]))
```

Keras: functional models

- Several inputs, several sub-networks
- Merge inputs, outputs

```
input_merged = Concatenate()([input_conds, input_eta])

model=Dense(dim_conds,use_bias=use_bias,activation="PReLU")(input_merged)
```

- Shortcuts: merge input and model output (inception)

A practical course on GANs with Keras

Inspired by [link]. See provided notebooks.

- Define a generator object as MLP with N_F-dimensional input, T- dimensional output, no optimiser, do not compile
- Define a discriminator object as an MLP with $T-$dimensional input, 1-dimensional output, sigmoid activation

- For $T \simeq O(10^2)$, the following works well, for INIT_LR = 3e-4

```
opt_discrinator = Adam(lr=INIT_LR/2, beta_1=0.5)#, decay=INIT_LR / NUM_EPOCHS)
discriminator.compile(loss='binary_crossentropy', optimizer=opt_discrinator)
discriminator.summary()
```

- Then one defines a GAN object (dim_noise= N_F)

```
discriminator.trainable = False

ganInput = Input(shape=(dim_noise,))
ganOutput = discriminator(generator(ganInput))
gan = Model(ganInput, ganOutput)
# compile the GAN
opt_gan = Adam(lr=INIT_LR/4, beta_1=0.5)#, decay=INIT_LR / NUM_EPOCHS)
gan.compile(loss="binary_crossentropy", optimizer=opt_gan)
```

GAN in practice, continued

By default, the weights of the discriminator are fixed

```
discriminator.trainable = False

ganInput = Input(shape=(dim_noise,))
ganOutput = discriminator(generator(ganInput))
gan = Model(ganInput, ganOutput)
# compile the GAN
opt_gan = Adam(lr=INIT_LR/4, beta_1=0.5)#, decay=INIT_LR / NUM_EPOCHS)
gan.compile(loss="binary_crossentropy", optimizer=opt_gan)
```

so that

- Calling `discriminator.train_on_batch(X, y)` only trains the discriminator weights.

- Calling gan.train_on_batch(...) only trains the generator weights

A batch in the life of a GAN: discriminator

- Choose M subsamples of size T of r_t
  $\rightarrow$ matrix $X_{real} \in \mathbb{R}^{M \times T}$
- Generate M samples of the feature noise $\eta \in \mathbb{R}^D$ from a Gaussian(?) distribution.
- Use the generator to predict M vectors $\tilde{r}_t$
  $\rightarrow$ matrix $X_{GAN} \in \mathbb{R}^{M \times T}$
- Concatenate X_{real} and $X_{GAN} \rightarrow X_{realGAN}$
- Define the label vector $y_{realGAN} = (1, \cdots, 1, 0, \cdots 0) \in \mathbb{R}^{2M}$
- Use the shuffle function to shuffle the order of the lines of X and y (at the same time)
- train_on_batch the discriminator

An epoch in the life of a GAN: generator

- Generate M' samples of the feature noise $\eta \in \mathbb{R}^D$
- Use the generator to predict the M' $\tilde{r}_t$
  $\rightarrow$ matrix $X_{FAKE} \in \mathbb{R}^{M' \times T}$
- Pretend that all of them are true $y_{FAKE} = (1, \cdots, 1) \in \mathbb{R}^{M'}$
- train_on_batch the generator (`gan` object here)

Technical note on model re-use: weights

```
#define model
model.compile(...)
model.fit(...)
model.saveweigths('model_weights.h5')
```

Later, elsewhere

```
#define model
model.compile(...)
model.loadweigths('model_weights.h5')   #instead of .fi
```

Technical note on model re-use: architecture

- Each model can be described as a tree
- Model architecture saved as json [link]

Save

```
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
model.save_weigths('model_weights.h5')
```

Load

```
with open('model.json', 'r') as json_file:
    model_descr = json_file.read()
model = model_from_json(model_descr)
model.load_weights("model_weights.h5")
model.compile(loss,optimizer)
```

Technical note on model re-use: full model

[doc]

Model = architecture + optimizer + losses + weights

Save

```
model.save('mymodel.keras')
```

Load

```
model=keras.models.load_model("mymodel.keras")
```