AI TOOLS FOR ACTUARIES
Chapter 11: Deep Generative Models

Ronald Richman, Mario V. Wüthrich

2025-11-30

Overview

Overview

- This chapter focuses on *deep generative models* (DGMs).
- DGMs do not only provide point predictors, but they construct full predictive distributions.
- Specifically, they may serve at sampling a new portfolio $\boldsymbol{X} \sim \mathbb{P}$ or a joint distribution of claims and covariates $(Y, \boldsymbol{X}) \sim \mathbb{P}$.
- We discuss the following methods in this chapter:
  - *latent factor approaches*, including
    - variational auto-encoders (VAEs),
    - generative adversarial networks (GANs),
    - denoising diffusion models;
  - *implicit probability distribution approaches*, including
    - transformer decoders used, e.g., in generative pre-trained transformers (GPT).

Introduction

- Most of the above chapters follow the classical paradigm of *supervised learning*.
- Supervised learning provides *point predictors* in a regression framework.
- This chapter discusses methods that produce full *predictive distributions*, i.e., that go beyond point predictors.
- The goal of this chapter is to learn the underlying probability distribution $\boldsymbol{X} \sim \mathbb{P}$ of the instances itself, or a joint distribution of responses and covariates $(Y, \boldsymbol{X}) \sim \mathbb{P}$.
- This is not done by just fitting a (simplistic) parametric distribution, but rather by using a network architecture to learn the parameters of a predictive distribution in a complex setting.

Possible applications

- *Generate new samples*: Draw new instances $\boldsymbol{X} \sim \mathbb{P}$ that resemble the data the model was trained on. This is useful for *data augmentation*, *scenario simulation*, *synthetic data creation* and *fairness analysis*.
- *Estimate probabilities*: Evaluate the likelihood $p(\boldsymbol{X})$ of a given data point $\boldsymbol{X}$, which is useful for *anomaly detection*, *scenario analysis* and *risk management*.
- *Infer latent representations*: Learn a lower-dimensional, *compressed representation* $\boldsymbol{Z}$ of the data $\boldsymbol{X}$, similar to the auto-encoders of the last chapter, but allowing for re-simulations.
- *Perform conditional data generation*: Generate new samples from a conditional distribution, e.g., $\boldsymbol{X}|_Y \sim \mathbb{P}(\cdot|Y)$ allowing for *targeted data synthesis* and *fairness analysis*.

A first summary

- The distinction between DGMs and unsupervised learning is that one does not only try to understand the structure in the data $\boldsymbol{X}$, but one wants to generate new data of the same structure.
- Most of the approaches in this field operate on a huge complex body of high-dimensional data, and they build the foundation for modern large language models (LLMs) such as ChatGPT, see next chapter.
- The literature in this field typically uses the notation $p(\boldsymbol{X})$ for the density and the likelihood of $\boldsymbol{X} \sim \mathbb{P}$, we adopt this convention in this chapter (being slightly inconsistent with previous chapters).
- Summary: The goal of DGMs is to infer (or approximate) the true probability distribution $\boldsymbol{X} \sim \mathbb{P}$ and its density $p(\boldsymbol{X})$ from a finite sample $\mathcal{L} = (\boldsymbol{X}_i)_{i=1}^n$ of observations.

Main deep generative modeling approaches

- We see two main approaches for DGMs in the literature:
  1. *latent factor approaches*, and
  2. *implicit probability distribution approaches*.
- A latent factor approach considers a parametric model

$$p_\vartheta(\boldsymbol{X}) = \int p_\vartheta(\boldsymbol{X}|\boldsymbol{z})\,\pi(\boldsymbol{z})\,\mathrm{d}\boldsymbol{z},$$

  where $\boldsymbol{Z} \sim \pi(\boldsymbol{z})$ is the latent variable capturing unobserved factors that create variability in the data samples.
- The implicit probability distribution approach directly learns a conditional distribution over samples $\boldsymbol{X} = (X_1, \ldots, X_T) = X_{1:T}$

$$p_\vartheta(X_{1:T}) = \prod_{t=1}^{T} p_\vartheta\left(X_t|X_{1:t-1}\right).$$

Popular latent factor approaches

- *Variational auto-encoder* (VAE): An *encoder network* approximates the posterior distribution $q_\vartheta(\boldsymbol{Z}|\boldsymbol{X})$ – mapping $\boldsymbol{X}$ to $\boldsymbol{Z}$ – and a *decoder network* approximates the conditional likelihood of the data $p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})$ – mapping $\boldsymbol{Z}$ to $\boldsymbol{X}$. This lifts the auto-encoder of the previous chapter to a variational version, by inputting noise $\pi(\boldsymbol{Z})$ to the latent factor.
- *Generative adversarial network* (GAN): By sampling directly from an assumed latent factor distribution $\boldsymbol{Z} \sim \pi(\boldsymbol{Z})$ a DGM is designed by an adversarial game involving a generator (G) and a discriminator (D). The discriminator tries to distinguish real data from the fake data sampled by the generator.
- *Denoising diffusion models*: By learning how to remove noise from noisy samples, a noisy sample is turned into a structured picture by denoising the underlying structure. This latter approach is more suitable for images removing noise in the image.

Implicit probability distribution approach

- The implicit probability distribution approach is different from the latent factor approach as it does *not* rely on explicit latent factors $\boldsymbol{Z}$.
- Instead, a neural network is used to directly learn the conditional distribution after the auto-regressive factorization

$$p_\vartheta(X_{1:T}) = \prod_{t=1}^{T} p_\vartheta\left(X_t | X_{1:t-1}\right).$$

- This can be interpreted as integrating out the latent factors.
- The architectures used are usually *decoder transformer* based:
  - *generative pre-trained transformer* (GPT),
  - *bi-directional and auto-regressive transformer* (BART),
  - *text-to-text transfer transformer* (T5).

Variational auto-encoders

Overview

- VAEs were introduced by Kingma and Welling (2013), Kingma and Welling (2019).
- VAEs consist of an *encoder network* and a *decoder network*, that parameterize two underlying distributions.
- Since the resulting log-likelihood is not available in closed form, one derives a lower bound – the ELBO – which is maximized for model learning instead of the true model log-likelihood.
- The VAE uses the same structure for model fitting as the statistical method of the expectation-maximization (EM) algorithm.
- This is the easiest and most robust method discussed in this chapter.

VAE architecture

- *Encoder* (inference/recognition model): The encoder network takes an input data point $\boldsymbol{X}$ and outputs the parameters of a latent distribution, typically a multivariate Gaussian distribution,

$$q_\vartheta(\boldsymbol{Z}|\boldsymbol{X}) \overset{(\mathrm{d})}{=} \mathcal{N}(\boldsymbol{\mu}_\vartheta(\boldsymbol{X}), \Sigma_\vartheta(\boldsymbol{X})).$$

  That is, $\boldsymbol{\mu}_\vartheta(\boldsymbol{X})$ and $\Sigma_\vartheta(\boldsymbol{X})$ are given by the encoder's output.

- *Decoder* (generative model): The decoder network takes a latent sample $\boldsymbol{Z}$ and outputs the parameters of a distribution over the data space, e.g.,

$$p_\vartheta(\boldsymbol{X}|\boldsymbol{Z}) \overset{(\mathrm{d})}{=} \mathcal{N}(\boldsymbol{m}_\vartheta(\boldsymbol{Z}), S_\vartheta(\boldsymbol{Z})),$$

  for absolutely continuous data $\boldsymbol{X}$.

- If $\boldsymbol{X}$ is binary, discrete or mixed, the decoder network estimates the parameters of a Bernoulli, a discrete or a mixed distribution.
- The decoder is the *generative component*, once trained, it can take random samples $\boldsymbol{Z} \sim \pi(\boldsymbol{Z})$ from the latent space and produce synthetic data points by generating new data $\boldsymbol{X}' \sim p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})$ that resembles the original data.
- Typically, the prior $\pi(\boldsymbol{Z})$ is chosen as a standard multivariate Gaussian distribution with independent components.
- Gaussian distributions lead to significant simplifications in model fitting because the Kullback–Leibler (KL) divergence used below is available in closed form.

Variational objective: the evidence lower bound

- The goal is to train the two components – encoder and decoder.
- Important: There are only observations for the data $\boldsymbol{X}$, but not for the latent factors $\boldsymbol{Z}$.
- A first fitting attempt is to the marginalize for the observations $\boldsymbol{X}$

$$p_\vartheta(\boldsymbol{X}) = \int p_\vartheta\left(\boldsymbol{X}|\boldsymbol{z}\right) \pi(\boldsymbol{z}) \, \mathrm{d}\boldsymbol{z}.$$

- Directly optimizing this log-likelihood $\log p_\vartheta(\boldsymbol{X})$ is generally unfeasible because in most cases the marginal likelihood is not available in closed form, and numerical integration is too costly.
- This is rather similar to the expectation-maximization (EM) algorithm used in latent factor and Bayesian models.

- Variational inference aims at maximizing a lower bound instead, the so-called *evidence lower bound* (ELBO).
- If this lower bound is large, also the log-likelihood will be large.
- Jensen's inequality implies – this is the basic step in the ELBO –

$$
\begin{aligned}
\log p_\vartheta(\boldsymbol{X}) &= \log \int p_\vartheta(\boldsymbol{X}|\boldsymbol{z})\,\pi(\boldsymbol{z})\,\mathrm{d}\boldsymbol{z} \\
&= \log \int \frac{p_\vartheta(\boldsymbol{X}|\boldsymbol{z})\,\pi(\boldsymbol{z})}{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}\, q_\vartheta(\boldsymbol{z}|\boldsymbol{X})\,\mathrm{d}\boldsymbol{z} \\
&= \log \mathbb{E}_{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}\left[\frac{p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})\,\pi(\boldsymbol{Z})}{q_\vartheta(\boldsymbol{Z}|\boldsymbol{X})}\right] \\
&\geq \mathbb{E}_{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}\left[\log\left(\frac{p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})\,\pi(\boldsymbol{Z})}{q_\vartheta(\boldsymbol{Z}|\boldsymbol{X})}\right)\right] \\
&= \mathbb{E}_{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}\Big[\log p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})\Big] - D_{\mathrm{KL}}\Big(q_\vartheta(\cdot|\boldsymbol{X})\Big\|\pi\Big) \\
&=: \mathcal{E}\left(\vartheta;\boldsymbol{X}\right),
\end{aligned}
$$

where $D_{\mathrm{KL}}(q_\vartheta(\cdot|\boldsymbol{X})\|\pi)$ is the KL divergence from π to $q_\vartheta(\cdot|\boldsymbol{X})$.

- The term $\mathcal{E}(\vartheta; \boldsymbol{X})$ is the ELBO that we try to maximize.
- The ELBO consists of the following two terms:

1. *Reconstruction term*:
   $\mathbb{E}_{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}[\log p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})]$ encourages the decoder to reconstruct the original $\boldsymbol{X}$ from the latent code $\boldsymbol{Z}$, that is, the latent factor $\boldsymbol{Z}$ should likely attain values that maximize the conditional log-likelihood $\log p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})$ of the given observation $\boldsymbol{X}$.
2. *Regularization term*:
   $-D_{\mathrm{KL}}(q_\vartheta(\cdot|\boldsymbol{X})\|\pi)$ aligns the encoder's approximate posterior with the prior $\pi(\boldsymbol{Z})$. Thus, the posterior $q_\vartheta(\cdot|\boldsymbol{X})$ is regularized towards that prior π.

- Combining these two items yields a balance between a faithful reconstruction of $\boldsymbol{X}$ and a latent space constraint to follow the prior assumptions (allowing one to simulate new samples from that prior).

- Given an i.i.d. learning sample $\mathcal{L} = (\boldsymbol{X}_i)_{i=1}^n$, the training target aims at maximizing the ELBO on that learning sample

$$\widehat{\vartheta}^{\mathrm{ELBO}} \in \arg\max_{\vartheta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{E}\left(\vartheta; \boldsymbol{X}_i\right).$$

- In general, this does not yield the MLE of ϑ, but hopefully it still provides a sufficiently good parameter estimate for ϑ.
- The ELBO is still intractable, also simulation is not straightforward, because the unknown parameter ϑ enters the probability densities.
- The *reparameterization trick* solves this problem in the Gaussian case.

Reparameterization trick

- The ELBO cannot easily be computed and the direct gradient of $\mathbb{E}_{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}[\cdot]$ involves the unknown parameter ϑ in the expectation operator.
- In the Gaussian case, the reparameterization trick of Kingma and Welling (2013) solves this problem.
- Assume a Gaussian encoder. The reparameterization trick rewrites the Gaussian latent factor $\boldsymbol{Z}$ as follows

$$\boldsymbol{Z} \overset{(\mathrm{d})}{=} \boldsymbol{\mu}_\vartheta(\boldsymbol{X}) + \Sigma_\vartheta^{1/2}(\boldsymbol{X})\,\boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}).$$

- The crucial observation is that $\boldsymbol{\varepsilon}$ is independent of ϑ, and we can now back-propagate through the encoder network outputs $\boldsymbol{\mu}_\vartheta(\boldsymbol{X})$ and $\Sigma_\vartheta(\boldsymbol{X})$ without any further issues.

- Using the reparameterization trick yields for given $\boldsymbol{X}$

$$\mathbb{E}_{q_\vartheta(\boldsymbol{z}|\boldsymbol{X})}\left[\log p_\vartheta(\boldsymbol{X}|\boldsymbol{Z})\right] = \mathbb{E}_\varepsilon\left[\log p_\vartheta\left(\boldsymbol{X}\,\middle|\,\boldsymbol{\mu}_\vartheta(\boldsymbol{X}) + \Sigma_\vartheta^{1/2}(\boldsymbol{X})\,\varepsilon\right)\right];$$

  the parameter ϑ is no longer part of the expectation operator $\mathbb{E}_\varepsilon[\cdot]$.

- The estimation process then employs a Monte Carlo version by
  1. first, sample $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ in the forward pass, and
  2. second, during the backward pass, take the gradients w.r.t. the mean and variance parameters of the encoder network.

- A single Monte Carlo sample often suffices to approximate the ELBO

$$\begin{aligned}\mathcal{E}(\vartheta; \boldsymbol{X}) &\approx \widetilde{\mathcal{E}}(\vartheta; \boldsymbol{X}, \varepsilon)\\ &:= \log p_\vartheta\left(\boldsymbol{X}\,\middle|\,\boldsymbol{\mu}_\vartheta(\boldsymbol{X}) + \Sigma_\vartheta^{1/2}(\boldsymbol{X})\,\varepsilon\right) - D_{\mathrm{KL}}\left(q_\vartheta(\cdot|\boldsymbol{X}) \,\|\, \pi\right).\end{aligned}$$

  This is an empirical approximation of the ELBO.

- By performing this Monte Carlo sampling during the fitting procedure, VAEs learn both the inference (encoder) and the generative (decoder) networks, respectively, by maximizing this Monte Carlo ELBO.
- Thus, solve

$$\widehat{\vartheta} \in \arg\max_{\vartheta} \frac{1}{n}\sum_{i=1}^{n}\frac{1}{J}\sum_{j=1}^{J}\Bigg[\log p_{\vartheta}\left(\boldsymbol{X}_i \,\middle|\, \boldsymbol{\mu}_{\vartheta}(\boldsymbol{X}_i) + \Sigma_{\vartheta}^{1/2}(\boldsymbol{X}_i)\,\boldsymbol{\varepsilon}_{i,j}\right) - D_{\mathrm{KL}}\left(q_{\vartheta}(\cdot|\boldsymbol{X}_i)\,\|\,\pi\right)\Bigg],$$

  for i.i.d. standard Gaussian samples $\boldsymbol{\varepsilon}_{i,j}$.
- Often, a single Monte Carlo sample $J = 1$ is sufficient.
- Once trained, we can generate new data by sampling $\boldsymbol{Z} \sim \pi(\boldsymbol{z})$ and then drawing $\boldsymbol{X}' \sim p_{\widehat{\vartheta}}(\boldsymbol{x}|\boldsymbol{Z})$ from the decoder network of the estimated DGM based on the estimated parameter $\widehat{\vartheta}$.

Discussion

- VAEs illustrate the latent factor approach described earlier: the hidden variables $\boldsymbol{Z}$ capture underlying structure, and the encoder-decoder networks map between $\boldsymbol{X}$ and $\boldsymbol{Z}$.
- This ensures that VAEs can both (1) reconstruct existing data, and (2) sample novel data points, all while maintaining a tractable training objective, the ELBO.
- In practice, many variants of VAEs exist, e.g., β-VAEs or conditional VAEs, each modifying the objective or architecture to emphasize different aspects, such as disentangled latent representations or conditional generation.
- Overall, VAEs remain one of the most popular DGMs due to their relative conceptual simplicity, stable training procedure, and the ability to produce both probabilistic encodings and realistic sample generations.

Kullback–Leiber divergence

- A last critical point is an efficient evaluation of the KL divergence in the ELBO.
- For Gaussian distributions $q \overset{(\mathrm{d})}{=} \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and $\pi \overset{(\mathrm{d})}{=} \mathcal{N}(\tau, T)$, the KL divergence from π to q is

$$\begin{aligned} D_{\mathrm{KL}}(q\|\pi) &= \frac{1}{2}\Bigg[\log\left(\frac{|T|}{|\Sigma|}\right) - r \\ &\qquad + \operatorname{trace}\left(T^{-1}\Sigma\right) + (\tau-\boldsymbol{\mu})^{\top} T^{-1}(\tau-\boldsymbol{\mu})\Bigg], \end{aligned}$$

  where r is the dimension of the latent variables.
- Thus, in the Gaussian case there is an efficient KL divergence computation.

Sports car example, revisited

- In the notebooks on unsupervised learning we were studying an example using car insurance features to label sports cars.
- We revisit this data, and we aim for simulating similar data.

```
library(tensorflow)
library(keras)         # Keras version 2
library(MASS)          # For generating multivariate normal data
library(ggplot2)
library(corrplot)

set.seed(1234)
tensorflow::set_random_seed(1234)
```

- We load the data and standardize the columns of the design matrix.

```
load(file="../../Data/SportsCars.rda")
dat    <- SportsCars
dat$x1 <- log(dat$weight/dat$max_power)
dat$x2 <- log(dat$max_power/dat$cubic_capacity)
dat$x3 <- log(dat$max_torque)
dat$x4 <- log(dat$max_engine_speed)
dat$x5 <- log(dat$cubic_capacity)
dat1   <- dat[, c("x1","x2","x3","x4","x5")]

# standardization of columns of design matrix
X01 <- dat1-colMeans(dat1)[col(dat1)]
X   <- as.matrix(X01/sqrt(colMeans(X01^2))[col(X01)])

n_samples <- nrow(X)
dim(X)
```

```
[1] 475   5
```

- We start be the encoder network $\boldsymbol{X} \to \boldsymbol{Z}$.

```
build_encoder <- function(qq, seed) {
  set.seed(seed)
  set_random_seed(seed)

  # input in X
  encoder_inputs <- layer_input(shape = c(qq[1]))

  # deep encoder FNN
  h <- encoder_inputs %>% layer_dense(qq[2], activation = "silu") %>%
                          layer_dense(qq[2], activation = "silu")

  # latent space parameters
  z_mean    <- layer_dense(h, qq[3])
  z_log_var <- layer_dense(h, qq[3])  # on log-scale to ensure positivity

  --- to be continued ...
```

```
  ... continued ---

  # sampling function
  sampling <- function(args) {
    z_mean    <- args[[1]]
    z_log_var <- args[[2]]
    epsilon   <- k_random_normal(shape = c(k_shape(z_log_var)[[1]]),
                                 mean = 0., stddev = 1.)
    z_mean + k_exp(z_log_var/2) * epsilon
  }

  # sample latent factor
  z <- layer_lambda(list(z_mean, z_log_var), sampling)

  # encoder model
  encoder <- keras_model(encoder_inputs, list(z_mean, z_log_var, z))
  return(encoder)
}
```

- Decoder network $\boldsymbol{Z} \rightarrow \boldsymbol{X}'$.

```
build_decoder <- function(qq, seed) {
  set.seed(seed); set_random_seed(seed)
  # input Z
  latent_input <- layer_input(shape = c(qq[3]))
  # deep decoder FNN
  h <- latent_input %>% layer_dense(qq[2], activation = "silu") %>%
                        layer_dense(qq[2], activation = "silu")
  # output layer
  x_decoded_mean <- layer_dense(h, qq[1], activation = "linear")
  decoder <- keras_model(latent_input, x_decoded_mean)
  return(decoder)
}
```

- This decoder network only estimates the mean $\boldsymbol{m}_\vartheta(\boldsymbol{Z})$ but not the covariance matrix $S_\vartheta(\boldsymbol{Z})$ of the Gaussian decoder model.
- The selected synthetic instance $\boldsymbol{X}'$ will be the most likely one, i.e., the mode of the Gaussian decoder.

- Building the VAE.

```
build_vae <- function(qq, seed) {
  encoder <- build_encoder(qq, seed)
  decoder <- build_decoder(qq, seed)
  # input layer
  x <- layer_input(shape = c(qq[1]))
  # encoder
  encoder_outputs <- encoder(x)
  z_mean          <- encoder_outputs[[1]]
  z_log_var       <- encoder_outputs[[2]]
  z               <- encoder_outputs[[3]]
  # decode
  x_decoded_mean  <- decoder(z)
  keras_model(x, list(x_decoded_mean = x_decoded_mean,
                      z_mean         = z_mean,
                      z_log_var      = z_log_var))
  }
```

- Defining the ELBO loss function for a Gaussian KL divergence.

```
tau <- 1  # Prior variance parameter

# Use separate loss functions for each output

# Square loss for decoder (we only model the mode but not the whole distribution)
reconstruction_loss <- function(y_true, y_pred) {
  k_mean(k_sum(k_square(y_true - y_pred), axis = 1))
}

kl_loss_mean <- function(y_true, y_pred) {
  kl1 <- k_square(y_pred) / (tau^2)/2
  k_mean(k_sum(kl1, axis=1))
}

kl_loss_var <- function(y_true, y_pred) {
  # variance of KL divergence from N(0, tau^2) and N(0, exp(z_log_var))
  kl1 <- (k_log(tau^2) - y_pred - 1 + k_exp(y_pred) / (tau^2))/2
  k_mean(k_sum(kl1, axis=1))
}
```

- Defining the VAE architecture

```
(qq <- c(ncol(X), 32, 10))   # input, hidden, latent space dimension
```

```
[1]  5 32 10
```

```
VAE <- build_vae(qq=qq, seed=1)

#VAE: compile model with separate losses for each output
VAE %>% compile(
  optimizer    = optimizer_adam(learning_rate = 0.001),
  loss         = list(x_decoded_mean = reconstruction_loss,
                      z_mean         = kl_loss_mean,
                      z_log_var      = kl_loss_var),
  loss_weights = list(x_decoded_mean = 1, z_mean = 1, z_log_var = 1)
)
```

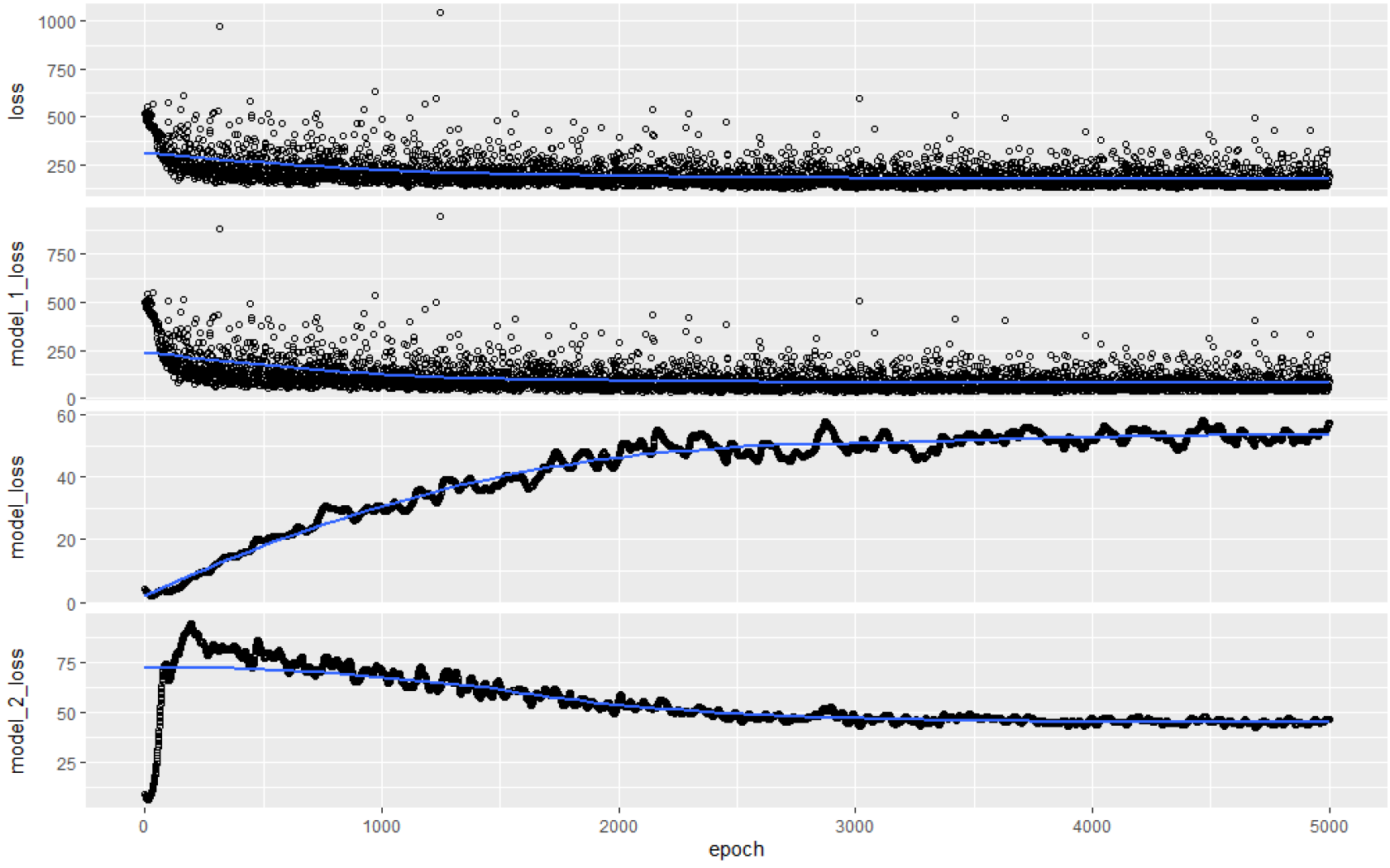
```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 5)]               0
 model (Functional)          [(None, 10),              1908
                              (None, 10),
                              (None, 10)]
 model_1 (Functional)        (None, 5)                 1573
=================================================================
Total params: 3481 (13.60 KB)
Trainable params: 3481 (13.60 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- A smaller latent space dimension provides data compression.
- Our goal is good reconstruction, therefore we choose a large latent space dimension.

- Training the VAE.

```
# For training, provide appropriate targets for each output
fit <- VAE %>% fit(
      list(X),
      list(x_decoded_mean = X,
           z_mean         = matrix(0, nrow(X), qq[3]),  # not needed in loss
           z_log_var      = matrix(0, nrow(X), qq[3])), # not needed in loss
      batch_size = n_samples,
      epochs = 5000
)

# Plot the fitting results
plot(fit)
```

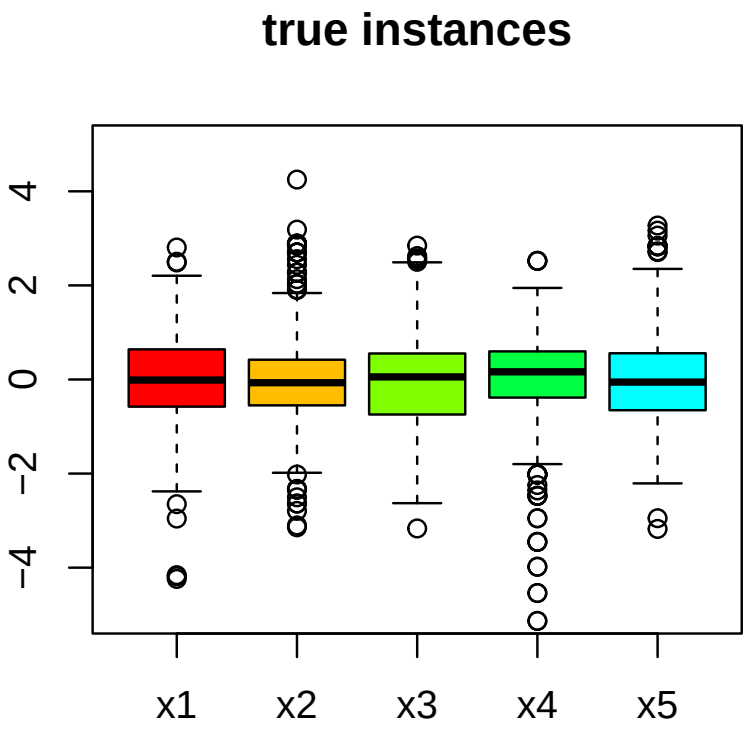
- Note: We do not track over-fitting, and we do not use drop-out. In bigger problems these should be done.

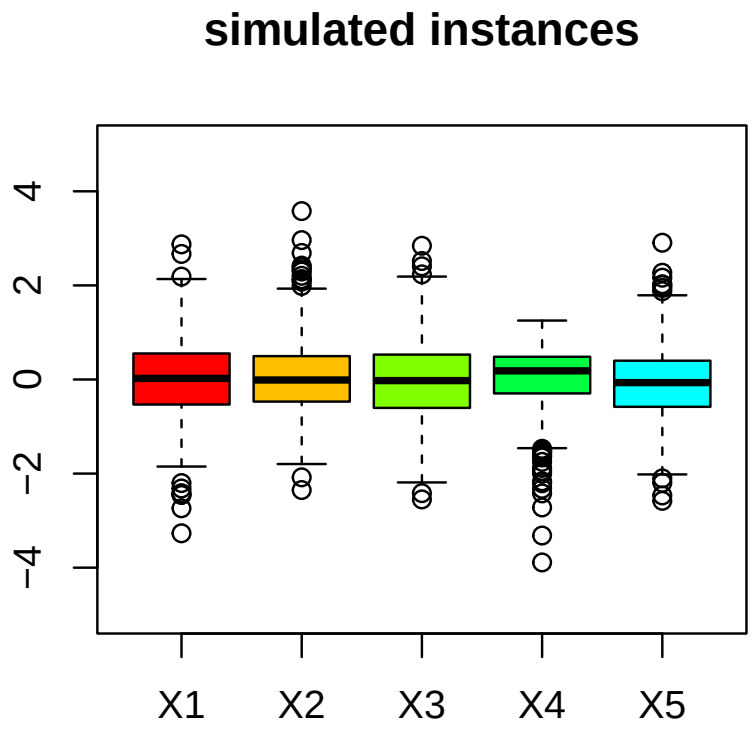


- We see a trade-off between the reconstruction error and the KL divergence with the Gaussian prior π – this is precisely how the ELBO balances.

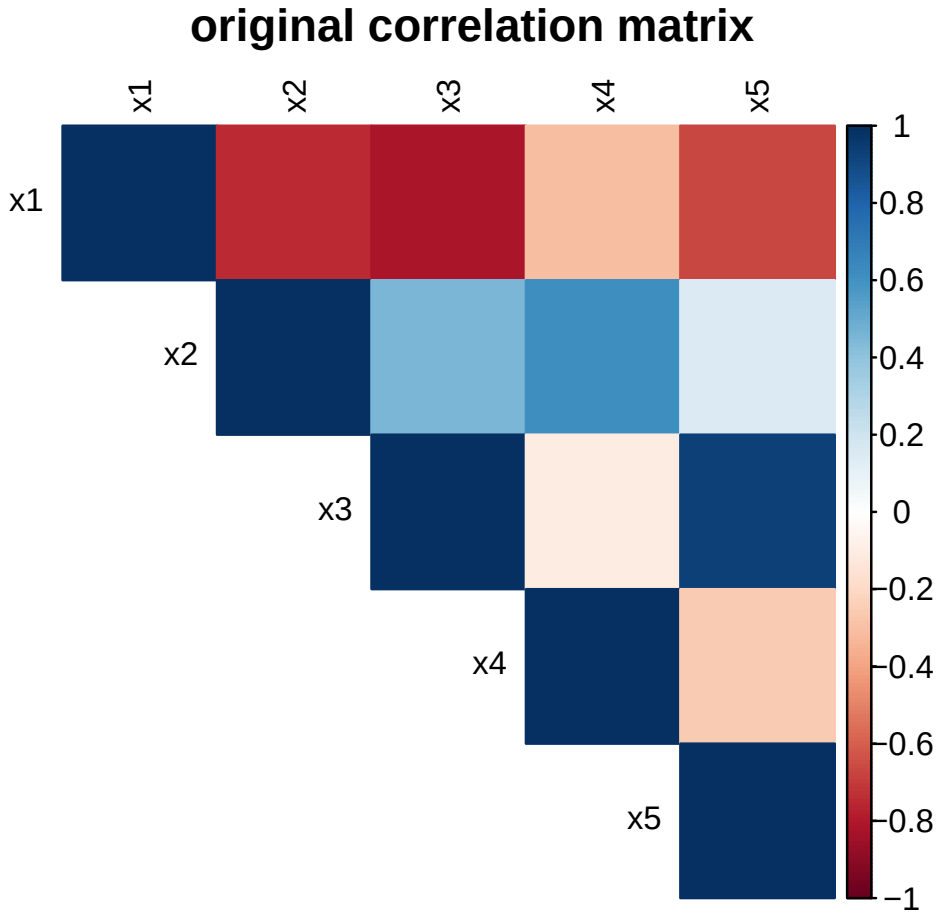
- Generate new samples $\boldsymbol{X}'$.
- Attention: We did not estimate the output covariance matrix $S_\vartheta(\boldsymbol{Z})$, and we can thus only give the most likely outcome $\boldsymbol{X}'$ in the mode of the estimated Gaussian. I.e., the volatility in the resulting sample will be too small.

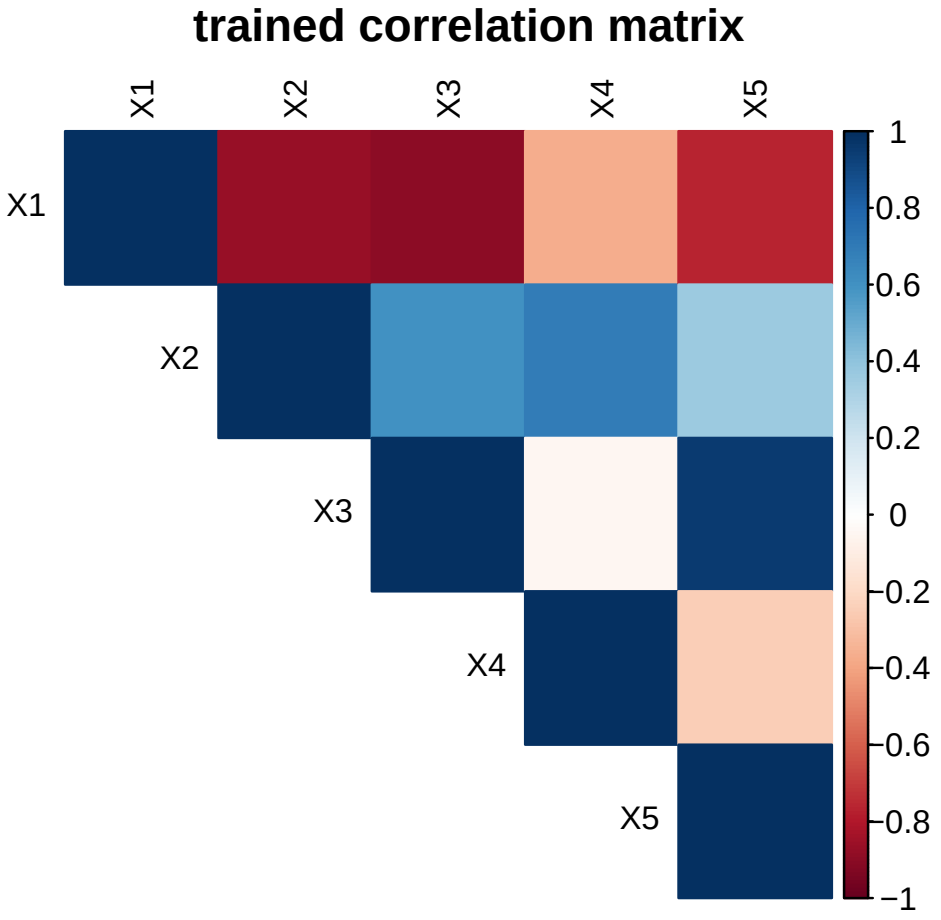
```
# we initialize the encoder to the learned weights - brute force
w0      <- get_weights(VAE)
Decoder <- build_decoder(qq=qq, seed=1)
w1      <- get_weights(Decoder)
for (jj in 1:6){ w1[[jj]] <- array(w0[[jj+8]], dim=dim(w1[[jj]])) }
set_weights(Decoder, w1)

# we generate Gaussian samples and run them through the decoder
set.seed(1234)
Z  <- mvrnorm(n = n_samples, mu = rep(0, qq[3]), Sigma = diag(qq[3]))
X1 <- Decoder %>% predict(list(Z), batch_size=10^6, verbose=0)
```





- The two samples seem to have a similar behavior, e.g., w.r.t. outliers.
- As mentioned above, the volatility in the simulated sample is too small because it only reflects the best estimates $\boldsymbol{m}_\vartheta(\boldsymbol{Z})$ – mode of the Gaussian.





- The correlation picture of the VAE looks convincing.
- The correlations of the VAE are slightly stronger; they are reduced by replacing the best estimates $\boldsymbol{m}_\vartheta(\boldsymbol{Z})$ by simulated samples $\boldsymbol{X}'$.

Generative adversarial networks

Overview

- GANs were introduced by Goodfellow *et al.* (2014).
- GANs are based on an adversarial game between a *generator* (G) and a *discriminator* (D).
- The generator learns to generate (fake) samples that resemble the true data.
- The discriminator aims at distinguishing fake from real data.
- The adversarial game has an objective function promoting that both, the generator and the discriminator, improve their skills, so that eventually, the generator samples data that is (almost) indistinguishable from the true one.

The GAN components

- *Generator* (G): A generator network G takes as input a random noise vector $\boldsymbol{Z} \sim \pi(\boldsymbol{Z})$, commonly a standard multivariate Gaussian distribution, and outputs a synthetic instance

$$\boldsymbol{X}' = G(\boldsymbol{Z}; \vartheta_1).$$

  The generator's objective is to produce synthetic instances that are indistinguishable from the real data.

- *Discriminator* (D): A discriminator network D receives at random either a real instance $\boldsymbol{X}$ (from the true dataset) or a synthetic instance $\boldsymbol{X}'$ (from the generator), and it outputs a Bernoulli probability for the received instance $\boldsymbol{X}''$ being a real instance

$$p := D(\boldsymbol{X}''; \vartheta_2) \in [0, 1].$$

  The discriminator aims to correctly classify real and synthetic data.

- Thus, the generator G is designed to take a random noise vector $\boldsymbol{Z}$ and transform it into a generated output $\boldsymbol{X}'$ by processing the noise through deep neural network layers.
- This process allows the generator to learn how to create realistic samples from random inputs.
- On the other hand, the discriminator D receives a sample $\boldsymbol{X}''$ as input and then processes it through a deep FNN to output a probability p that indicates whether the sample is real ($p \approx 1$) or synthetic ($p \approx 0$).
- We use $\vartheta = (\vartheta_1, \vartheta_2)$ generically to denote the model parameter, though in implementations one typically maintains separate sets of parameters for G and D.
- These two networks are trained simultaneously in a *minimax game*.

The GAN objective function

- GAN training considers a zero-sum game between the generator and the discriminator given by the minimax objective

$$\min_G \max_D V(D,G) = \min_G \max_D \Big(\mathbb{E}_{\boldsymbol{X}\sim p_{\text{data}}(\boldsymbol{x})} \Big[\log D(\boldsymbol{X}) \Big] + \mathbb{E}_{\boldsymbol{Z}\sim \pi(\boldsymbol{z})} \Big[\log \left(1 - D(G(\boldsymbol{Z}))\right) \Big] \Big).$$

- $\boldsymbol{X} \sim p_{\text{data}}(\boldsymbol{x})$ denotes the true (unknown) data distribution.
- $\boldsymbol{Z} \sim \pi(\boldsymbol{z})$ is the distribution of the pure noise vector.
- $V(D,G)$ is the value function of this minimax game.

- The discriminator D aims at maximizing the value $V(D, G)$, trying to distinguish real and fake instances. This is just the Bernoulli log-likelihood of labels $Y \sim \mathrm{Bernoulli}(p)$, given by

$$Y \log p + (1 - Y) \log(1 - p).$$

- The generator G aims at minimizing the value $V(D, G)$, trying to "fool'' the discriminator D so that its generated instances are classified as real.
- Optimization is performed via alternating gradient-based updates.
- The discriminator's weights are frozen during the training of the generator G, meaning to say that while the GAN is being trained to improve the generator, only the generator's parameters are updated. The purpose of freezing the discriminator in this step is to ensure that updates are made only to the generator network G, allowing it to learn how to produce samples that can optimally fool the discriminator.

Training GANs

- Training GANs is known to be difficult and can suffer some issues:
- *Mode collapse*: The generator learns to produce only a limited variety of samples, basically just sticking to a data cluster in the real data.
- *Vanishing gradients*: Gradient descent gets stuck in flat areas of the value function.
- Despite these challenges, with proper techniques – e.g., careful network design, hyper-parameter tuning, and objective variants like Wasserstein GAN – GANs can generate highly detailed and convincing samples.
- The training can be improved by slightly randomizing the labels, and, clearly, the data needs to be shuffled so that real and generated instances spread equally across the batches in gradient descent.

Sports car example, revisited

- We revisit the Sports car data from above and generate similar data.
- We set up a GAN model class; the code for this is taken and adapted from Chollet and Falbel (2022).
- This GAN model class alternates gradient descents updates between the discriminator D and the generator G.

```
GAN <- new_model_class("GAN",
  initialize = function(discriminator, generator, latent_dim, input_dim) {
    super()$`__init__`()
    self$discriminator <- discriminator
    self$generator     <- generator
    self$latent_dim    <- latent_dim
    self$input_dim     <- input_dim
  },
--- to be continued ...
```

- We define separate optimizations for discriminator and generator.

```
... continued ---

  compile = function(d_optimizer, g_optimizer, loss_fn) {
    super()$compile()
    self$d_optimizer   <- d_optimizer
    self$g_optimizer   <- g_optimizer
    self$loss_fn       <- loss_fn
    self$d_loss_metric <- tf$keras$metrics$Mean(name = "d_loss")
    self$g_loss_metric <- tf$keras$metrics$Mean(name = "g_loss")
  },

  metrics = mark_active(function() {
    list(self$d_loss_metric, self$g_loss_metric)
  }),

--- to be continued ...
```

- We define the training step for the discriminator D.

```
... continued ---

  train_step = function(real_samples) {
    # Sample random Gaussian factors in the latent space
    batch_size            <- tf$shape(real_samples)[1]
    random_latent_factors <- tf$random$normal(shape = reticulate::tuple(batch_size,
↪   self$latent_dim))

    # Decode random Gaussian factors to generated (fake) samples
    generated_samples <- self$generator(random_latent_factors)

    # Combine them with real samples
    combined_samples  <- tf$concat(list(generated_samples, real_samples), axis = 0L)

    # Generate labels discriminating generated and real samples (we set label zero for
    ↪  generated and label one for real)
    labels <- tf$concat(list(tf$zeros(reticulate::tuple(batch_size, 1L)),
↪   tf$ones(reticulate::tuple(batch_size, 1L))), axis = 0L)

--- to be continued ...
```

```
... continued ---

    # Add random noise to the labels - important trick (Chollet--Falbel, 2022)
    labels <- labels + 0.05 * tf$random$uniform(tf$shape(labels))

    # Shuffle the rows of real and generated samples
    X2   <- tf$concat(list(combined_samples, labels), axis = 1L)
    X2   <- tf$random$shuffle(X2)
    shuffled_samples <- X2[,c(1L:self$input_dim)]
    labels           <- X2[, (self$input_dim+1L)]

    # Train the discriminator: gradient descent step
    with(tf$GradientTape() %as% tape, {
      predictions <- self$discriminator(shuffled_samples)
      d_loss      <- self$loss_fn(labels, predictions)
    })
    grads <- tape$gradient(d_loss, self$discriminator$trainable_weights)
    self$d_optimizer$apply_gradients(zip_lists(grads,
↪   self$discriminator$trainable_weights))

--- to be continued ...
```

- We define the training step for the generator G.

```
... continued ---

    # Sample random Gaussian factors in the latent space
    random_latent_vectors <- tf$random$normal(shape = reticulate::tuple(batch_size,
↪   self$latent_dim))

    # Attach labels that say "all real samples" (label one is for real)
    misleading_labels <- tf$ones(reticulate::tuple(batch_size, 1L))

    # Train the generator (only update the weights of the generator)
    with(tf$GradientTape() %as% tape, {
      predictions <- self$discriminator(self$generator(random_latent_vectors))
      g_loss      <- self$loss_fn(misleading_labels, predictions)
    })
    grads <- tape$gradient(g_loss, self$generator$trainable_weights)
    self$g_optimizer$apply_gradients(zip_lists(grads, self$generator$trainable_weights))

--- to be continued ...
```

- Finally, collect the discriminator's and generator's loss metrics.

```
... continued ---

    # Update metrics
    self$d_loss_metric$update_state(d_loss)
    self$g_loss_metric$update_state(g_loss)
    list(
      "d_loss" = self$d_loss_metric$result(),
      "g_loss" = self$g_loss_metric$result()
    )
  }
)
```

- This builds the model class `GAN` to train in an alternating way the discriminator and the generator.
- It involves latent factor sampling, generating new fakes samples in each loop of the training algorithm.

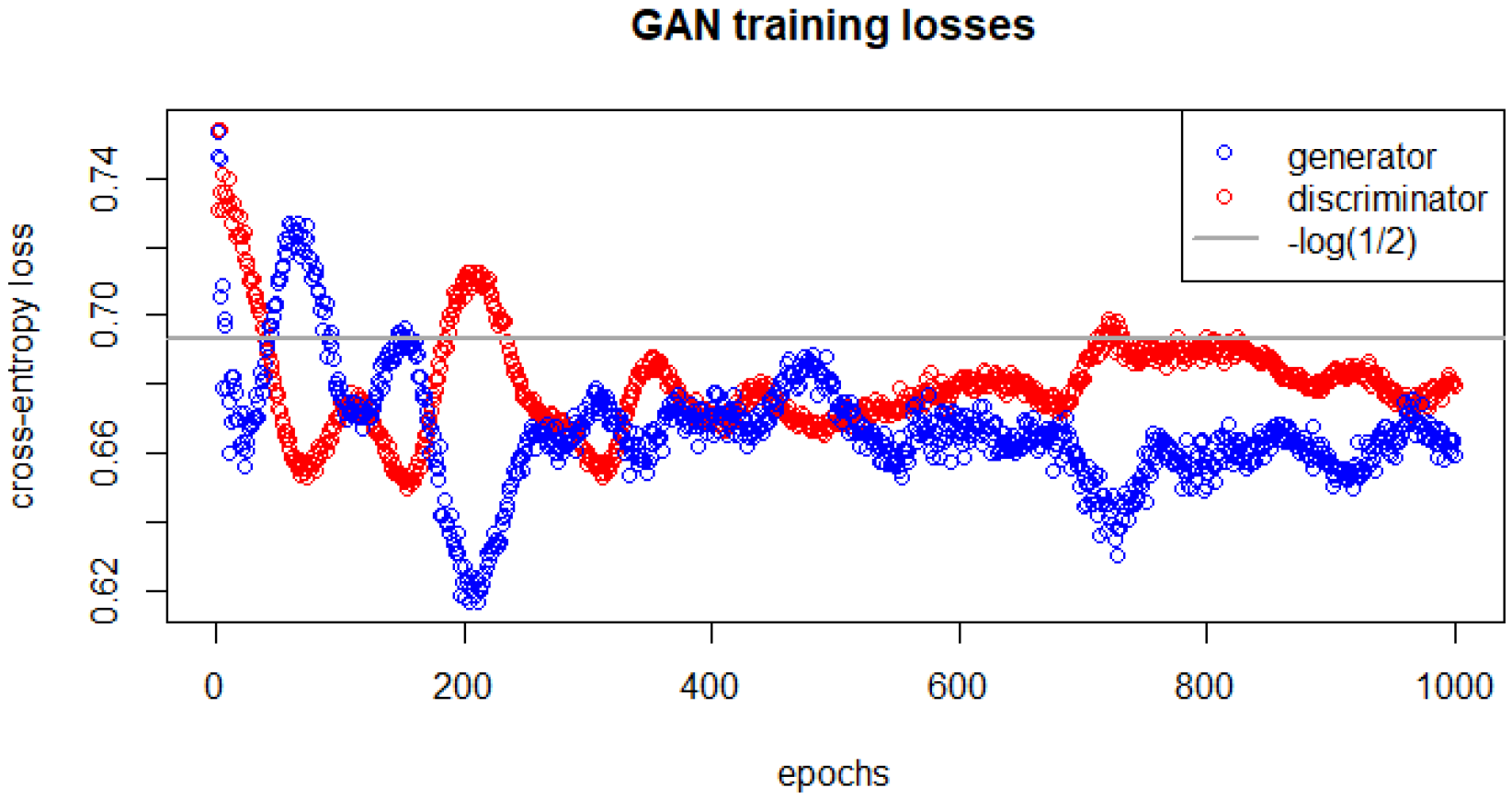
- Define the discriminator D and the generator G.

```
latent_dim <- 10L    # dimension of the latent space
tf$keras$backend$clear_session()
set.seed(1234)
tensorflow::set_random_seed(1234)

discriminator <- keras_model_sequential(name = "discriminator", input_shape =
↪  shape(ncol(X))) %>%
                layer_dense(20, activation = "tanh") %>%
                layer_dense(10, activation = "tanh") %>%
                layer_dense(1, activation = "sigmoid")
#summary(discriminator)

generator     <- keras_model_sequential(name = "generator", input_shape =
↪  shape(latent_dim)) %>%
                layer_dense(15, activation = "tanh") %>%
                layer_dense(10, activation = "tanh") %>%
                layer_dense(ncol(X), activation = "linear")
#summary(generator)
```

```
Model: "discriminator"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 20)                120
 dense_1 (Dense)             (None, 10)                210
 dense (Dense)               (None, 1)                 11
=================================================================
Total params: 341 (1.33 KB)
Trainable params: 341 (1.33 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

Model: "generator"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 15)                165
 dense_4 (Dense)             (None, 10)                160
 dense_3 (Dense)             (None, 5)                 55
=================================================================
Total params: 380 (1.48 KB)
Trainable params: 380 (1.48 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- Train the GAN architecture.

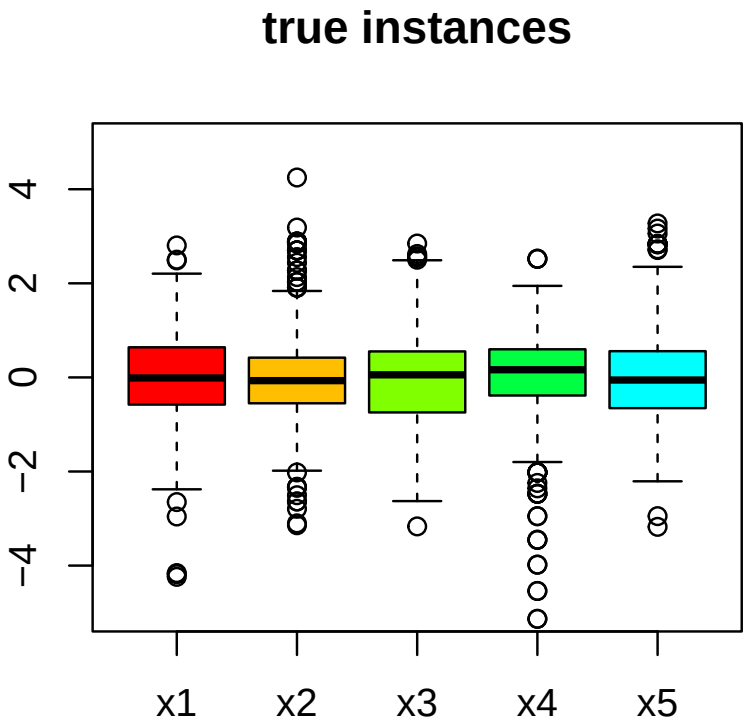
```
gan    <- GAN(discriminator = discriminator,
              generator     = generator,
              latent_dim    = latent_dim,
              input_dim     = ncol(X))

gan %>% compile(d_optimizer = optimizer_adam(learning_rate = 1e-4),
                g_optimizer = optimizer_adam(learning_rate = 1e-4),
                loss_fn     = loss_binary_crossentropy())

epochs <- 1000

#fit <- gan %>% fit(X, epochs = epochs)
```

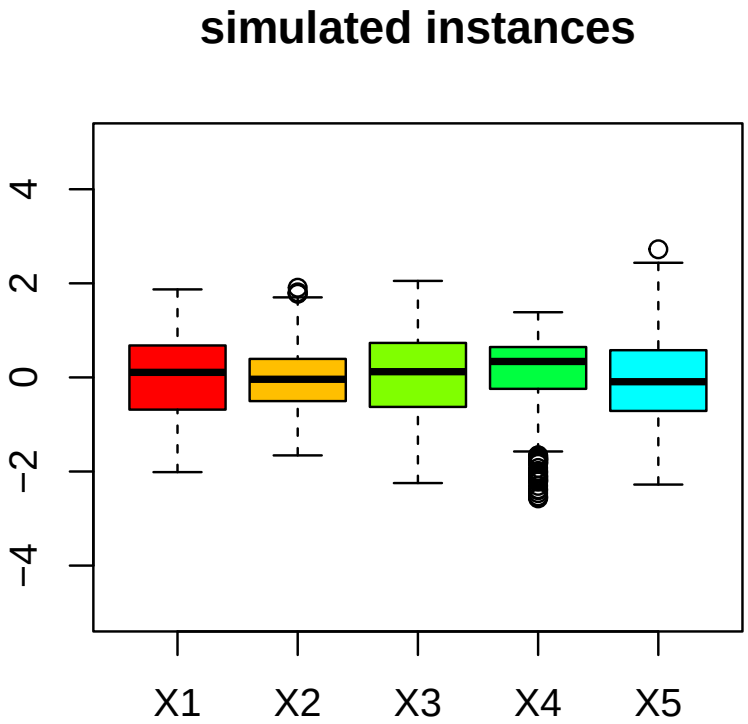


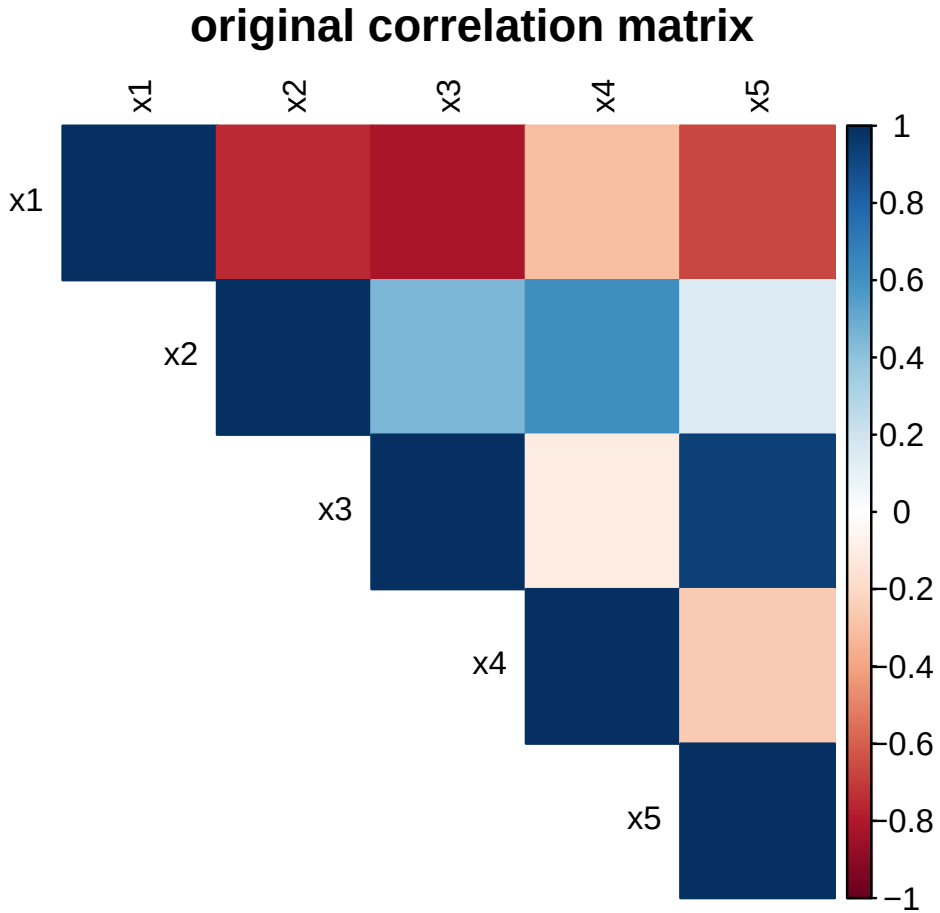
- The discriminator's loss is close to $-\log(1/2)$ (gray horizontal line), i.e., the discriminator is equally good as random guessing ($p = 1/2$) in detecting fake and real instances, respectively.

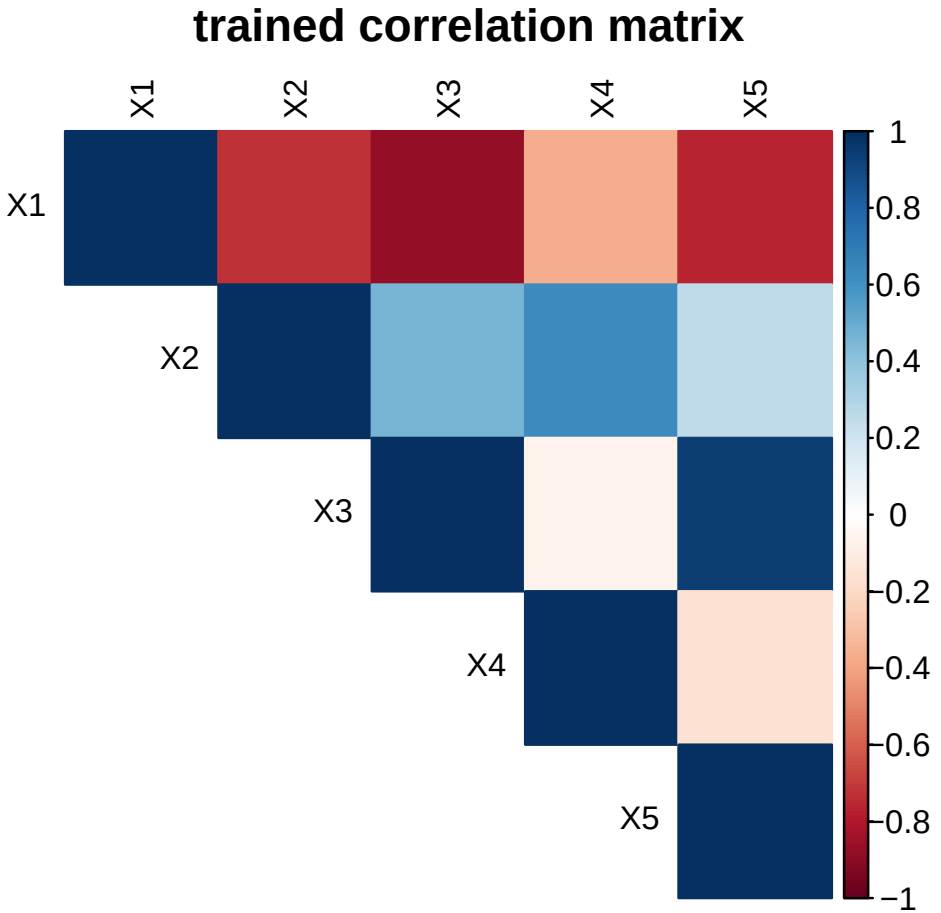
- Generate random instances $\boldsymbol{X}'$ based on the generator.

```
Z   <- mvrnorm(n = n_samples, mu = rep(0, latent_dim), Sigma = diag(latent_dim))
X1  <- gan$generator(Z)
```









- Overall, the generated GAN samples seem slightly less convincing than in the VAE case, though both methods are able replicate the correlation matrix fairly well.
- We refrain giving further detailed analysis of the generated samples.

Denoising diffusion models

Overview

- Denoising diffusion models were introduced by Song and Ermon (2019).
- They have gained significant attention as a state-of-the-art approach for image and audio generation; Ho, Jain and Abbeel (2020).
- Denoising diffusion models do not rely on a latent factor directly, but they learn how to denoise data, to recover the structure in noisy (corrupted) samples.
- This is done by setting up a Markov process removing the noise layer-by-layer.
- We present this method in this section because it can be seen as a multi-step VAE after integrating out the latent factor $\boldsymbol{Z}$.

Forward noising process

- A typical forward noising process is defined as a discrete-time Markov process of fixed length T; Sohl-Dickstein *et al.* (2015) and Ho, Jain and Abbeel (2020).
- Starting with a real data sample $\boldsymbol{X}_0$, produce a sequence of increasingly noisy samples

$$\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_T,$$

with each step $\boldsymbol{X}_t$ being more noisy

$$q(\boldsymbol{X}_t | \boldsymbol{X}_{t-1}) \overset{(\mathrm{d})}{=} \mathcal{N}\left(\sqrt{1-\beta_t}\, \boldsymbol{X}_{t-1},\, \beta_t\, \mathbf{I}\right),$$

with a variance schedule $(\beta_t)_{t=1}^T \in (0,1)$.

- The final sample is nearly indistinguishable from pure Gaussian noise; supposed the variance schedule is sufficiently large.

Training objective - reverse denoising

- The training aims to learn the reverse distribution $p_\vartheta(\boldsymbol{X}_{t-1}|\boldsymbol{X}_t)$.
- One way to view this is through a variational perspective, which yields an ELBO on the negative log-likelihood of $\boldsymbol{X}_0$.
- A simplified, yet empirically effective, noise-estimation objective aims to determine the noise that was added in a forward step:
  - Let $\boldsymbol{X}_0$ be a real sample and $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ be drawn independently.
  - Define the noised version of $\boldsymbol{X}_0$ at step t via

$$\boldsymbol{X}_t \;=\; \sqrt{\alpha_t}\,\boldsymbol{X}_0 \;+\; \sqrt{1-\alpha_t}\,\varepsilon,$$

    where $\alpha_t = \prod_{s=1}^{t}(1-\beta_s)$.
  - A neural network $\boldsymbol{\epsilon}_\vartheta$ is trained to estimate ε from the input $(\boldsymbol{X}_t, t)$ by minimizing

$$L_{\text{simple}}(\vartheta) \;=\; \mathbb{E}_{\boldsymbol{X}_0,\,\varepsilon,\,t}\Big[\big\|\varepsilon \;-\; \boldsymbol{\epsilon}_\vartheta(\boldsymbol{X}_t,\; t)\big\|_2^2\Big].$$

- By minimizing the loss $L_{\text{simple}}(\vartheta)$, the model learns to denoise $\boldsymbol{X}_t$ at each time step t, effectively approximating the score function (the gradient of the log-density w.r.t. the data) and providing a route to reverse the forward chain of noising the inputs.
- Once trained, sampling proceeds by starting with $\boldsymbol{X}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and recursively applying the denoising simulation

$$\boldsymbol{X}_{t-1} \sim p_\vartheta(\boldsymbol{X}_{t-1}|\boldsymbol{X}_t), \quad t = T, \ldots, 1,$$

  to obtain a final sample $\boldsymbol{X}_0$ that resembles the data distribution.
- In applications, the neural network estimates either the noise ε or the clean sample $\boldsymbol{X}_0$ from $(\boldsymbol{X}_t, t)$.

Discussion

- Diffusion models present a compelling alternative to VAEs and GANs for high-fidelity data generation, particularly for large-scale image and audio data.
- In contrast to the single-step generation of VAEs and GANs (where latent noise is transformed into a final sample in one forward pass), diffusion models gradually refine pure noise into structured data.
- Although this multi-step sampling can be slower, the gradual nature often leads to stable training dynamics and high-quality images.
- Moreover, diffusion-based approaches can be unified through a differential equation perspective.
- When it comes to actuarial problems, applications of denoising diffusion models are scarce, and we refrain giving an example but we refer to the Keras Tutorial (2022) on generative deep learning.

Decoder transformer models

Overview

- *Decoder transformers* extend the transformer framework to the task of auto-regressive data generation.
- The *encoder-only* transformers from above process an entire input sequence to extract a meaningful representation.
- *Decoder-only* architectures focus on predicting each subsequent element of an output sequence recursively.
- This makes them naturally suited to deep generative modeling, particularly of text, but also of other sequential modalities.
- Decoder-based transformer architecture include *generative pre-trained transformers* (GPT), see Radford *et al.* (2018) and Brown *et al.* (2020), *bi-directional and auto-regressive transformers* (BART), see Lewis and Liu (2020), and *text-to-text transfer transformers* (T5), see Raffel *et al.* (2020).

Architecture overview

- A decoder transformer follows sequential data

$$X_{1:T} = (X_1, X_2, \ldots, X_T),$$

by defining a joint distribution over all tokens via the factorization

$$p_\vartheta(X_{1:T}) = \prod_{t=1}^{T} p_\vartheta\left(X_t | X_{1:t-1}\right).$$

- At each time step t, the decoder uses self-attention over the previously generated tokens $X_{1:t-1}$, together with positional embeddings, to form a representation to predict the next token X_t.
- A *causal mask* is applied in the self-attention mechanism to ensure time-causality.
- The positional embeddings in decoding transformers are not learned, but are a static (previsible) function.

Self-attention with causal masking

- Recall the scaled dot-product attention for query Q, key K and value V

$$H = \text{softmax}\left(QK^\top/\sqrt{q}\right)V,$$

where q is the embedding dimension of the query $\boldsymbol{q}_u$, key $\boldsymbol{k}_u$ and value vectors $\boldsymbol{v}_u$, for $1 \leq u \leq t-1$, of the tensors $Q, K, V \in \mathbb{R}^{(t-1)\times q}$.

- To preserve time-causality, each token's query $\boldsymbol{q}_u$ is restricted to only attend to keys from the previous positions $(\boldsymbol{k}_s)_{s=1}^u$.
- This is implemented via a causal mask in the softmax step

$$\text{mask}_{s,u} = \begin{cases} 0 & \text{for } s \leq u, \\ -\infty & \text{for } s > u, \end{cases}$$

where the mask is added to $\boldsymbol{q}_u^\top \boldsymbol{k}_s/\sqrt{q}$.

- This results in an attention score of 0, whenever $s > u$.

(Softmax) outputs and probability calibration

- The final step of a decoder transformer is the output function.
- If this architecture is used in a language context, the hidden representation of the decoder transformer is projected onto the vocabulary space $\mathcal{W}$, yielding $\text{logits}(w; X_{1;t-1})$ for each token $w \in \mathcal{W}$.
- Applying the softmax function gives the distribution on the next token

$$p_\vartheta\left(X_t = w | X_{1:t-1}\right) = \frac{\exp(\text{logits}(w; X_{1:t-1})/\tau)}{\sum_{u \in \mathcal{W}} \exp(\text{logits}(u; X_{1:t-1})/\tau)},$$

  where τ is referred to as the *temperature*.
- Conceptually, decoder-only transformers align with the implicit probability distribution approach discussed above.

- The next token is fully parametrized by the model, conditioned solely on the past $X_{1:t-1}$.
- In practice, this allows for straightforward sampling – token by token – and yields a model that scales well as the dataset grows.
- We must note that large neural networks are frequently miscalibrated, i.e., their predicted probabilities do not align well with the true outcome frequencies.
- Temperature scaling can alleviate the problem by adjusting the sharpness of the distribution:
  - increasing $\tau > 1$ flattens the distribution to reflect higher uncertainty,
  - decreasing $\tau < 1$ sharpens the distribution to form confident predictions.
- This temperature scaling is rather similar to the choice of a Bayesian prior distribution (strong vs. vague priors).
- This post-hoc calibration affects the sampling diversity, and it ensures that probability estimates align better with actual uncertainty.

Training

- Decoder transformers are trained with *teacher forcing*: In each step of the recursive problem to predict the next token X_t, we insert the ground-truth tokens $X_{1:t-1}$ as inputs, and not their predicted counterparts $\widehat{X}_{1:t-1}$ from the previous steps.
- This ensures capturing the sequential dependencies of the real data.
- This training scheme results in a very efficient use of sequenced data, accelerating training by providing a correct context.
- Specifically, we maximize the log-likelihood

$$\ell_{X_{1:T}}(\vartheta) \;=\; \sum_{t=1}^{T} \log p_{\vartheta}\left(X_t | X_{1:t-1}\right),$$

  on a dataset of sequences.

Inference

- Once trained, decoding proceeds token-by-token.
- We begin with a first token X_1, compute $p_\vartheta(X_2|X_1)$, sample or select the most probable token, append it to the partial sequence, and continue until we reach a predefined end-of-sequence token or a desired length.
- This iterative procedure naturally yields samples that reflect the learned distribution over sequences.

Applications

- *Language modeling and text generation*: GPT models achieve state-of-the-art results on various NLP benchmarks, facilitating tasks such as summarization, extraction, translation, and open-domain dialogue, among others.
- *Conditional text generation*: Using additional conditioning signals (e.g., prompts, context paragraphs), decoder transformers can produce targeted text in specific domains. BART and T5 exemplify architectures that excel at summarization and question-answering.
- *Code generation*: Extensions of the decoder transformers successfully generate programming code and assist software development.
- Scaling up decoder transformers to trillions of parameters – *large language models* – yields models capable of coherent long-range generation and *in-context learning* (ICL), i.e., generalization to new tasks if the right context is provided (without additional training).

Copyright

- © The Authors
- This notebook and these slides are part of the project "AI Tools for Actuaries". The lecture notes can be downloaded from:

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5162304

References I

Brown, T. *et al.* (2020) 'Language models are few-shot learners', *Advances in Neural Information Processing Systems*, 33. Available at: https://arxiv.org/abs/2005.14165.

Chollet, F. and Falbel, D. (2022) 'DCGAN to generate face images'. Available at: https://tensorflow.rstudio.com/examples/dcgan_overriding_train_step.

Goodfellow, I. *et al.* (2014) 'Generative adversarial nets', *Advances in Neural Information Processing Systems*, 27. Available at: https://arxiv.org/abs/1406.2661.

References II

Ho, J., Jain, A. and Abbeel, P. (2020) 'Denoising diffusion probabilistic models', *Advances in Neural Information Processing Systems*, 33. Available at: https://arxiv.org/abs/2006.11239.

Keras Tutorial (2022) 'Keras examples: Generative deep learning'. Available at: https://keras.io/examples/generative/.

Kingma, D.P. and Welling, M. (2013) 'Auto-encoding variational Bayes', *arXiv preprint arXiv:1312.6114* [Preprint]. Available at: https://arxiv.org/abs/1312.6114.

Kingma, D.P. and Welling, M. (2019) 'An introduction to variational autoencoders', *Foundations and Trends in Machine Learning*, 12(4), pp. 307–392. Available at: https://arxiv.org/abs/1906.02691.

References III

Lewis, M. and Liu, G., Y. (2020) 'BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension', *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 7871. Available at: https://arxiv.org/abs/1910.13461.

Radford, A. *et al.* (2018) 'Improving language understanding by generative pre-training', *OpenAI Technical Report* [Preprint]. Available at: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

References IV

Raffel, C. *et al.* (2020) 'Exploring the limits of transfer learning with a unified text-to-text transformer', *Journal of Machine Learning Research*, 21(140). Available at: https://jmlr.org/papers/volume21/20-074/20-074.pdf.

Sohl-Dickstein, J. *et al.* (2015) 'Deep unsupervised learning using nonequilibrium thermodynamics', *Proceedings of the 32nd International Conference on Machine Learning*, 32. Available at: https://arxiv.org/abs/1503.03585.

Song, Y. and Ermon, S. (2019) 'Generative modeling by estimating gradients of the data distribution', *Advances in Neural Information Processing Systems*, 32. Available at: https://arxiv.org/abs/1907.05600.