Application of the Wasserstein Generative Adverserial Network (WGAN) on timeseries data

Ilias Aarab

April 4, 2022

Contents

Chapter 1

Application of the Wasserstein Generative Adverserial Network (WGAN) on timeseries data

This notebook applies the revised WGAN-GP model on timeseries data, with the ultimate goal of generating timeseries from the financial domain. Full implementation can be found here

1.1 Import libraries and configure Jupyter Notebook environment

In [1]:
```
# Import lib and config notebook with `DataAnalyst package`
from DataAnalyst.utils.libraries import * #load all packages into JN
from DataAnalyst import IPythonConfig
IPythonConfig.config_jedi()
IPythonConfig.set_cwd()
IPythonConfig.set_autoreload()
IPythonConfig.config_libs()

# Tensorflow
import tensorflow as tf
import tensorflow.keras as tfk

# Sklearn
from sklearn.model_selection import train_test_split

import warnings
```

```
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Working directory: C:Science_mvp

In [2]:
```
import matplotlib.pyplot as plt
from matplotlib_inline.backend_inline import set_matplotlib_formats
import seaborn as sns
sns.set_context(context='notebook', font_scale=1.2)
sns.set(rc={"figure.dpi": 100, 'savefig.dpi': 500})
set_matplotlib_formats('retina')
sns.set_style("ticks", {'axes.grid': True, 'grid.linestyle': ":",
               "grid.alpha": 1, "grid.color": "#343434", "lines.linewidth": 2.5, "xtick.bottom" : True
})

np.set_printoptions(suppress=True)
pd.set_option('display.float_format', lambda x: '%.3f' % x)
```

In [3]:
```
# Import in-house modules
# -----------------------
from src.model.gan_base import GANBase
from src.utils.nnvis import GANvis
from src.utils.toydata_generator import dataGenerator
from src.utils.parser import Parser
```

Chapter 2

Focus on our `Generator`

Let's see if our `Generator` implementation works as expected.

We first generate our *real* dataset from a multivariate Gaussian distribution with the following configuration:

$$P_r \sim \mathcal{N}(\mu, \Omega)$$

with

$$\mu = [2, 2], \Omega = \begin{bmatrix} 1 & 0.7 \\ 0.7 & 1 \end{bmatrix}$$

```
In [55]: # Create "real" dataset
         n, v= 1000, 2
         distribution= "multivariate Gaussian"
         mu, std= [2, 2], [[1, 0.7], [0.7, 1]]
         #mu, std= 10, 1
         Pr = dataGenerator.generate_data(nsamples=n, ndim=v, distribution=distribution, loc= mu, scale= std)
         display(Markdown("A sample output of the dataset:"))
         Pr.head()
```

A sample output of the dataset:

`Out[55]:`

	feature_1	feature_2
0	1.773	1.630
1	2.231	2.847
2	1.323	2.891
3	1.862	2.811
4	1.974	3.652

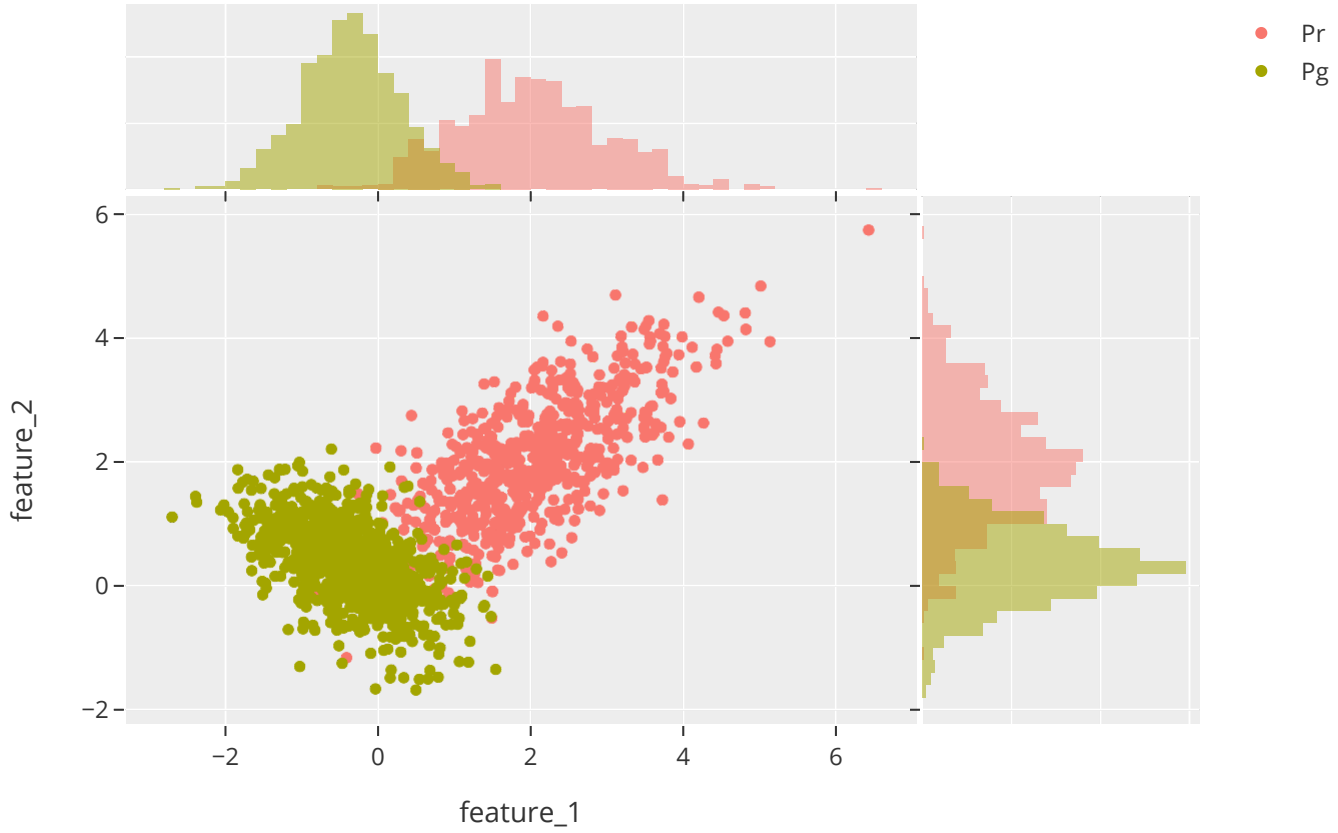
2.1 Drawing from our untrained `Generator`

Let's now initialize our `GAN` model and draw samples from the untrained `Generator` and compare the samples with the ones coming from the real dataset.

We see that the original dataset has an elliptical distribution as expected. In contrast the `Generator` simulates random noise that is tightly centered around 0. This is due to the fact that we have initiated the latent space of the `Generator` to be derived from a standard multivariate Gaussian distribution.

```
In [5]: # Init base class of GAN
        gan= GANBase(Pr)
        gan.init_discriminator(output_activation=tfk.activations.sigmoid)
        gan.init_latent_space("Gaussian")
        gan.init_generator(output_activation= tfk.activations.linear)
        # Sample real and generated data
        Pr_sampled= gan.generate_data('Pr', n= 1000, as_df= True)
        Pg_sampled= gan.generate_data('Pg', n= 1000, as_df= True)
        # Plot samples
        GANvis.plot_scatter(data= (Pr_sampled, Pg_sampled)).show(config= GANvis.config)
```

```
2022-04-04 01:26:26 | DEBUG | src.model.gan_base:__init__ | line 34 | Initialize GANBase class
```


Distributions of Pr and Pg
Pr
Pg
feature_2
feature_1
6
4
2
0
−2
−2
0
2
4
6

Chapter 3

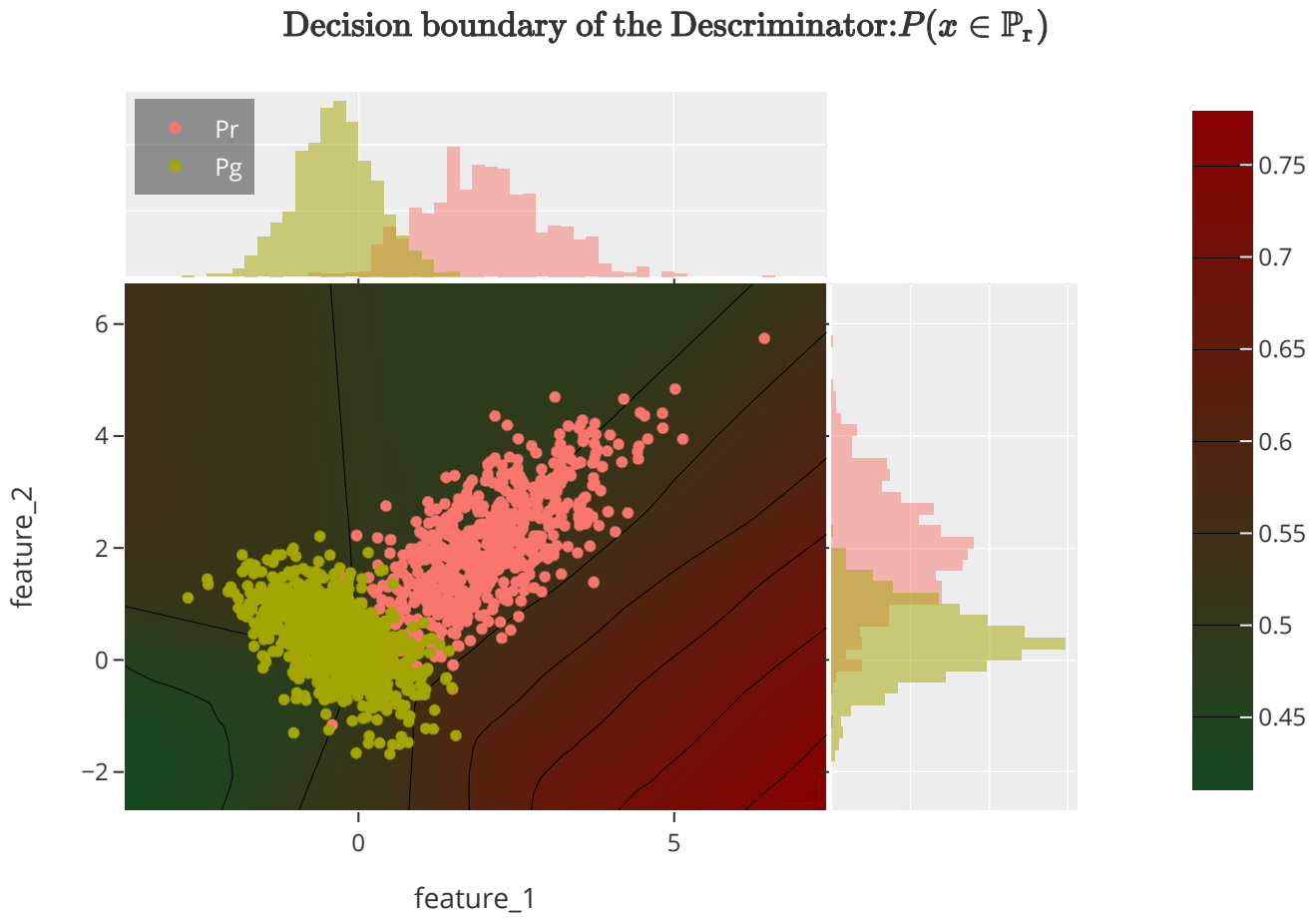
Zoom into our `Discriminator`

Now that we know that we can generate data with our `Generator` (albeit still untrained), let us a take a look at our `Discriminator` and see what its current decision boundary is in an untrained state.

We first use our `GAN` framework to generate data from both P_r and P_g, next we use our `Parser` to parse the data to a typical machine learning tuple (X, y) with y a binary variable equal to 1 if the data is coming from the real distribution P_r and else 0.

The parsed data is then feeded into the `Discriminator` and we use our `GANvis` to visualize it's current decision boundary.

```
In [7]: # Parse data into an (X, y) tuple
        X, y= Parser.to_ml_tuple(data= (Pr_sampled, Pg_sampled))
        # Compute and plot decision boundary of the Discriminator
        fig = GANvis.plot_decision_boundary(data= (Pr_sampled, Pg_sampled) , mdl= gan.D, col_names= ["feature 1", "feature 2"])
        fig.show(config=GANvis.config)
```



In it's current state, the decision boundary of the `Discriminator` is all over the place indicating that the `Discriminator` is unable to distinguish between the *real* and *generated* data.

Let's train the `Discriminator` for a couple of epochs (without training the `Generator`), to see whether improvements can quickly be seen.

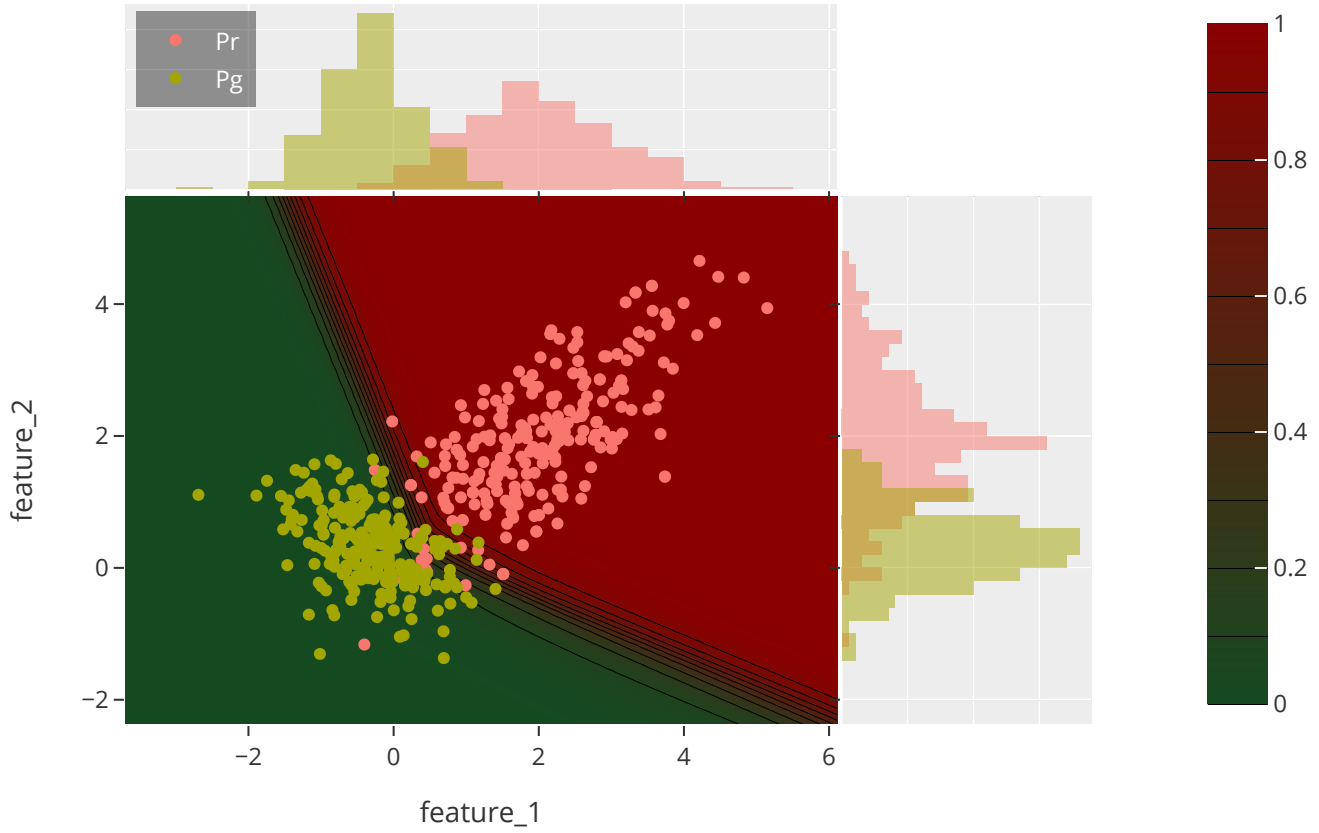
We use the `SKlearn`'s `train_test_split` function to split the machine learning tuple into a training and test set, and feed the training set to the `Discriminator` for 10 epochs:

In [8]:
```
# Train the Discriminator now for a couple of epochs
# Create train and test split
xtrain, xtest, ytrain, ytest= train_test_split(X, y, test_size= 0.25, random_state=0)
# Train Discriminator
gan.D.compile(loss='BinaryCrossentropy', metrics= ['Accuracy'])
gan.D.fit(x= xtrain, y= ytrain, epochs= 10)
```

```
Epoch 1/10
47/47 [==============================] - 3s 3ms/step - loss: 0.3111 - Accuracy: 0.9067
Epoch 2/10
47/47 [==============================] - 0s 5ms/step - loss: 0.1265 - Accuracy: 0.9573
Epoch 3/10
47/47 [==============================] - 0s 4ms/step - loss: 0.0963 - Accuracy: 0.9667
Epoch 4/10
47/47 [==============================] - 0s 6ms/step - loss: 0.0924 - Accuracy: 0.9673
Epoch 5/10
47/47 [==============================] - 0s 5ms/step - loss: 0.0915 - Accuracy: 0.9687
Epoch 6/10
47/47 [==============================] - 0s 4ms/step - loss: 0.0924 - Accuracy: 0.9693
Epoch 7/10
47/47 [==============================] - 0s 4ms/step - loss: 0.0924 - Accuracy: 0.9660
Epoch 8/10
47/47 [==============================] - 0s 5ms/step - loss: 0.0918 - Accuracy: 0.9700
Epoch 9/10
47/47 [==============================] - 0s 4ms/step - loss: 0.0908 - Accuracy: 0.9680
Epoch 10/10
47/47 [==============================] - 0s 4ms/step - loss: 0.0900 - Accuracy: 0.9687
```

Out[8]:
```
<keras.callbacks.History at 0x23fabec4160>
```

In [9]:
```
fig = GANvis.plot_decision_boundary(data= (xtest, ytest) , mdl= gan.D, col_names= ["feature1", "feature2"])
fig.show(config=GANvis.config)
```



It's clear that the `Discriminator` was able to learn the optimal decision boundary to distinguish between *real* and *generated* data. With both subcomponents working as expected, let's take a look at the composite model now.

Chapter 4

Zoom into our WGAN-GP

The `WGANGP` is a subclass of our `GANBase` that incorporates

(i) the *wasserstein* loss function for continous differentiation
(ii) an added *Langrage* term to ensure *Lipschitz* continuity

For more theoretical information, refere to the previous notebooks.

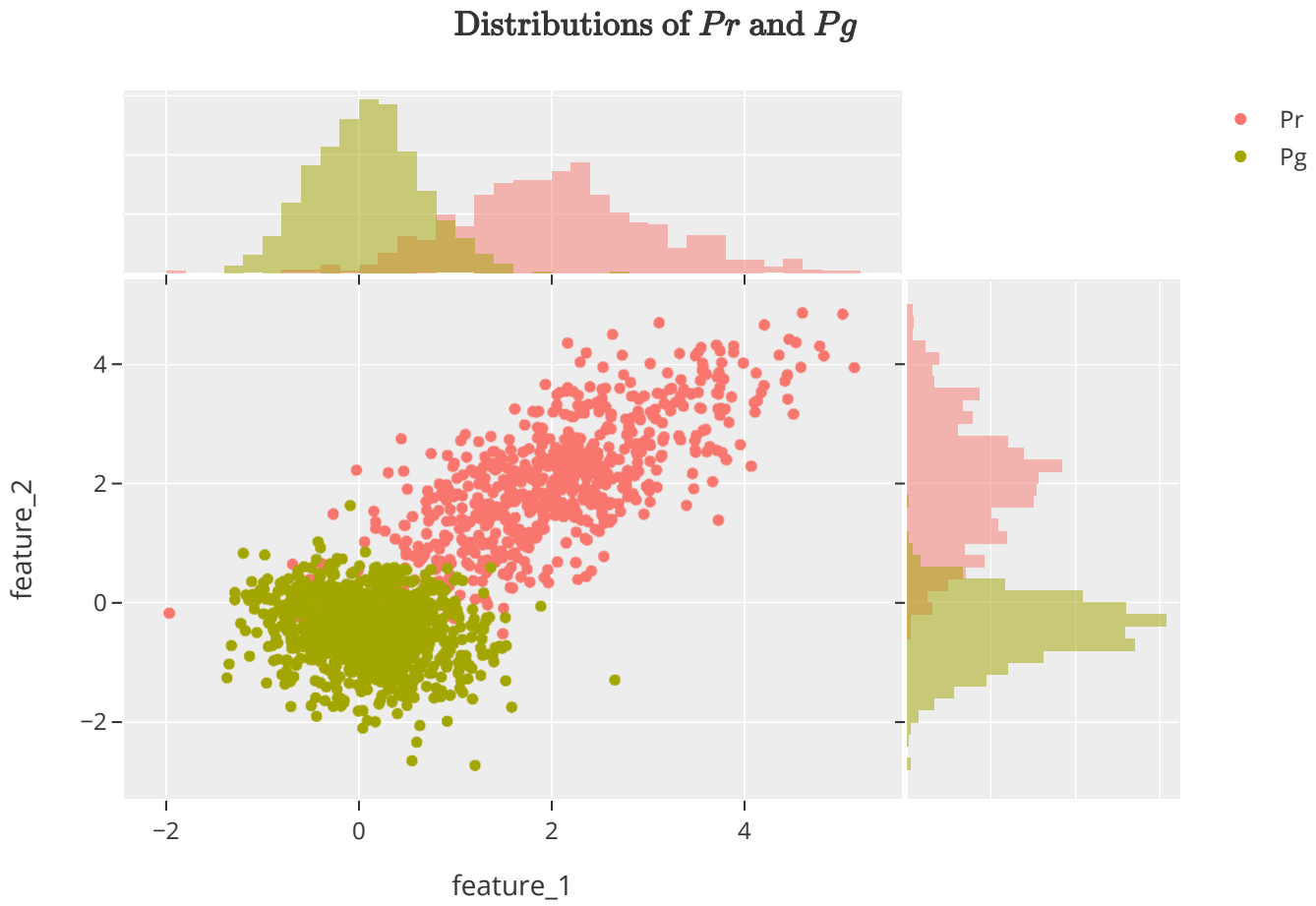
```
In [17]: from src.model.wgan_gp import WGANGP
         from src.model.gan_base import GANBase
```

```
In [18]: wgan = WGANGP(Pr= Pr, dim_latent_space= 128)
         wgan.init_discriminator()
         wgan.init_generator(output_activation= tfk.activations.linear)
         wgan.init_latent_space("Gaussian")
```

```
2022-04-04 01:46:35 | DEBUG | src.model.wgan_gp:__init__ | line 29 | Initialize tfk.Model
2022-04-04 01:46:35 | DEBUG | src.model.gan_base:__init__ | line 34 | Initialize GANBase class
2022-04-04 01:46:35 | DEBUG | src.model.wgan_gp:__init__ | line 34 | Initialize WGANGP
```

Let's again generate data from the same correlated Gaussian distribution that we have used above.

```
In [19]: # Sample real and generated data
         Pr_sampled= wgan.generate_data('Pr', n= 1000, as_df= True)
         Pg_sampled= wgan.generate_data('Pg', n= 1000, as_df= True)
         # Plot samples
         GANvis.plot_scatter(data= (Pr_sampled, Pg_sampled)).show(config= GANvis.config)
```



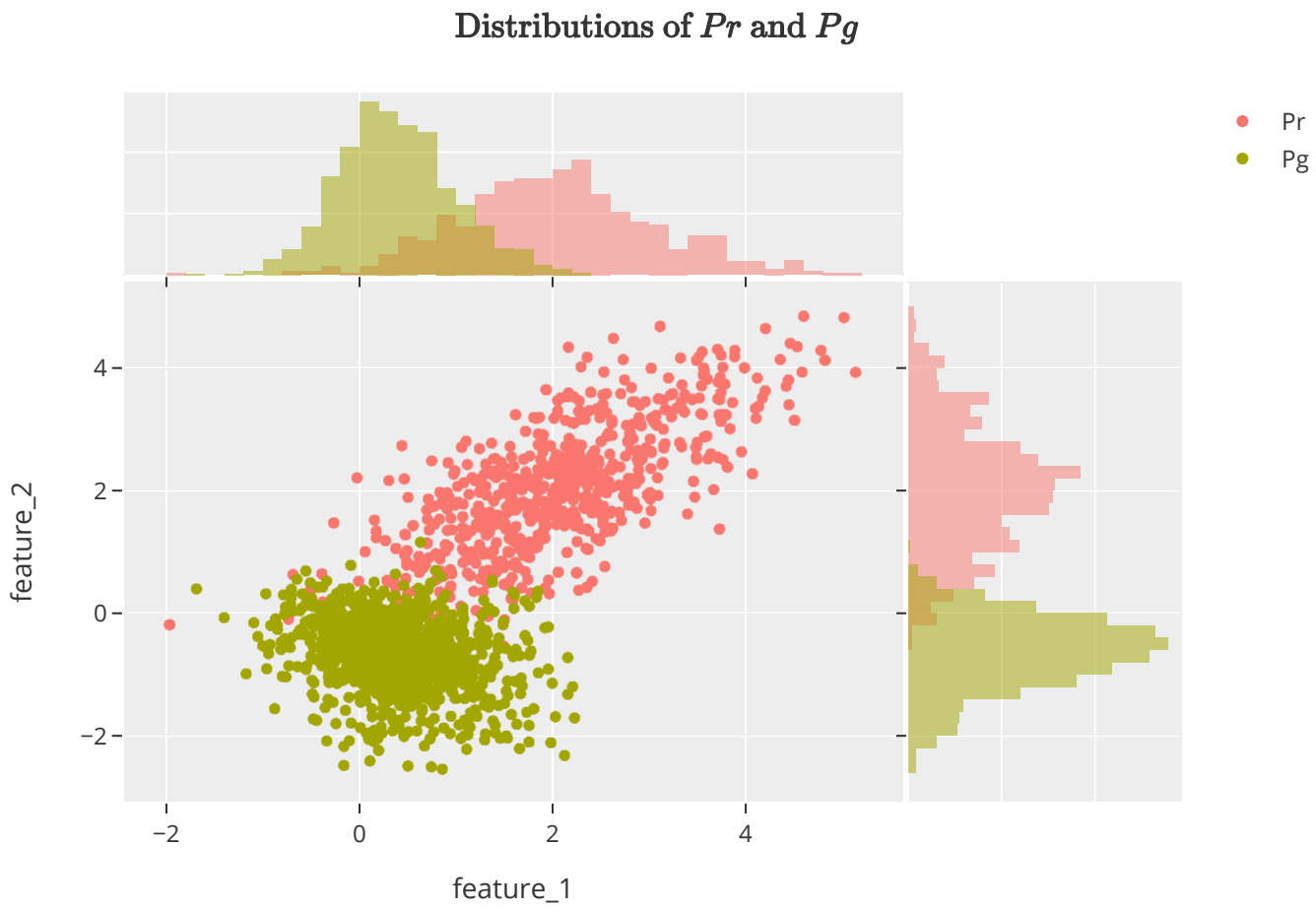
Now we train `WGANGP` as a whole, where the `Generator` and `Discriminator` try to solve their min-max optimization problem. We start with 10 epochs:

```
In [20]: # Define optimizers
         D_opti = tfk.optimizers.Adam(
```

```
            learning_rate=0.0002, beta_1=0.5, beta_2=0.9
        )

        G_opti = tfk.optimizers.Adam(
            learning_rate=0.0002, beta_1=0.5, beta_2=0.9
        )


        # Define losses
        def D_loss(Pr, Pg):
            #return - (tf.reduce_mean(Pg) - tf.reduce_mean(Pr))
            return tf.reduce_mean(Pg) - tf.reduce_mean(Pr)
        def G_loss(Pg):
            return -tf.reduce_mean(Pg)

        wgan.compile(D_opti, G_opti, D_loss, G_loss)

        history = wgan.fit(Pr.to_numpy().astype("float32"), batch_size= Pr.shape[0], epochs= 10)
```

```
Epoch 1/10
1/1 [==============================] - 8s 8s/step - d_loss: 5.7788 - g_loss: -0.0536
Epoch 2/10
1/1 [==============================] - 0s 48ms/step - d_loss: 5.5212 - g_loss: -0.0693
Epoch 3/10
1/1 [==============================] - 0s 48ms/step - d_loss: 5.1299 - g_loss: -0.0851
Epoch 4/10
1/1 [==============================] - 0s 45ms/step - d_loss: 4.8734 - g_loss: -0.1019
Epoch 5/10
1/1 [==============================] - 0s 38ms/step - d_loss: 4.4799 - g_loss: -0.1200
Epoch 6/10
1/1 [==============================] - 0s 38ms/step - d_loss: 4.2346 - g_loss: -0.1403
Epoch 7/10
1/1 [==============================] - 0s 34ms/step - d_loss: 3.9030 - g_loss: -0.1625
Epoch 8/10
1/1 [==============================] - 0s 39ms/step - d_loss: 3.5806 - g_loss: -0.1866
Epoch 9/10
1/1 [==============================] - 0s 52ms/step - d_loss: 3.3382 - g_loss: -0.2134
Epoch 10/10
1/1 [==============================] - 0s 44ms/step - d_loss: 3.0719 - g_loss: -0.2432
```

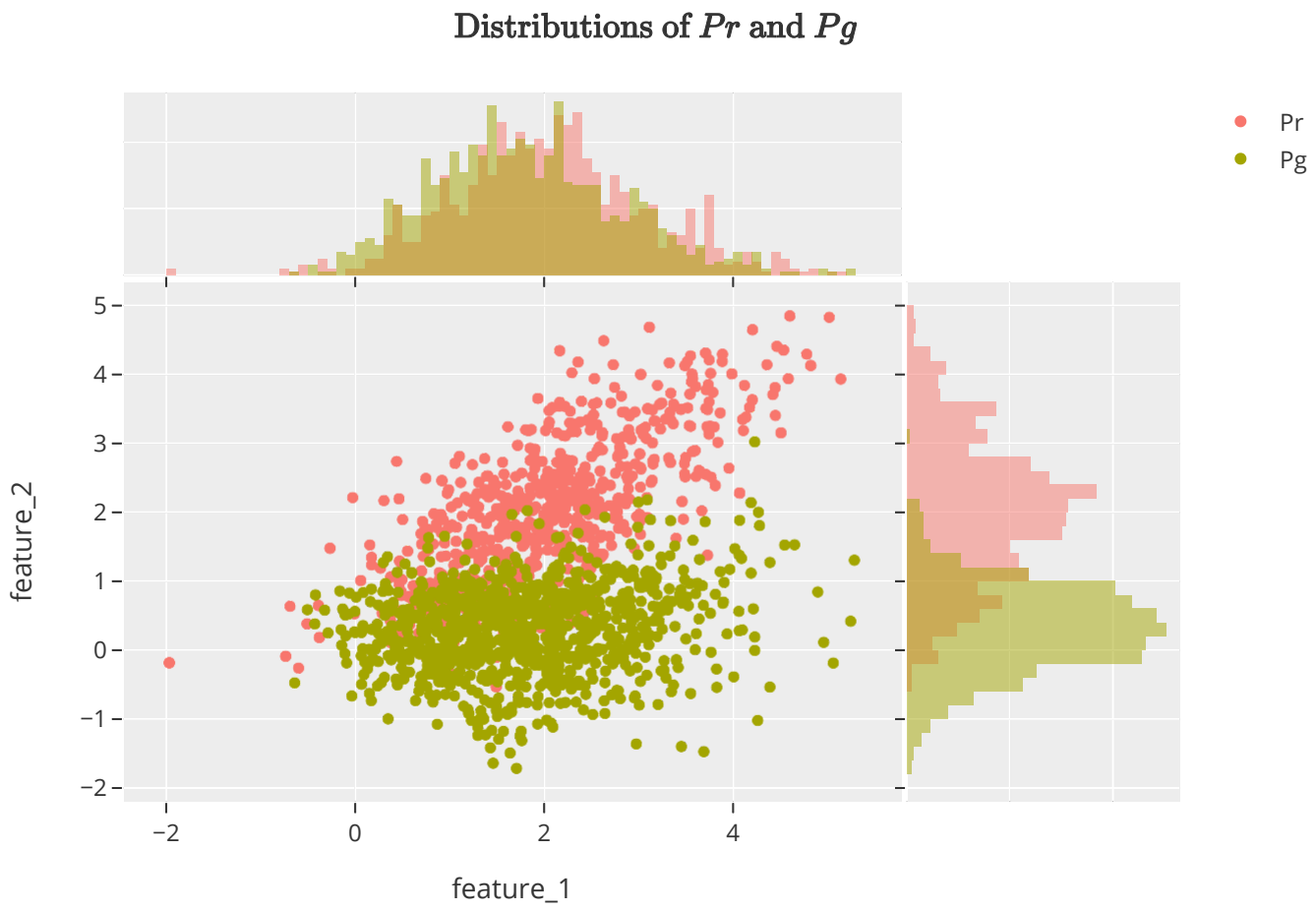
```
In [21]: Pg_new= wgan.generate_data("Pg", n= len(Pr_sampled), as_df= True, to_numpy= True)
         GANvis.plot_scatter(data= (Pr_sampled, Pg_new)).show(config= GANvis.config)
```



The results are quite poor, but we do see the *generated* distribution trying to take on a different shape. Let's continue the training until 100 epochs is reached:

```
In [ ]: history = wgan.fit(Pr.to_numpy().astype("float32"), batch_size= Pr.shape[0], epochs= 90)
```

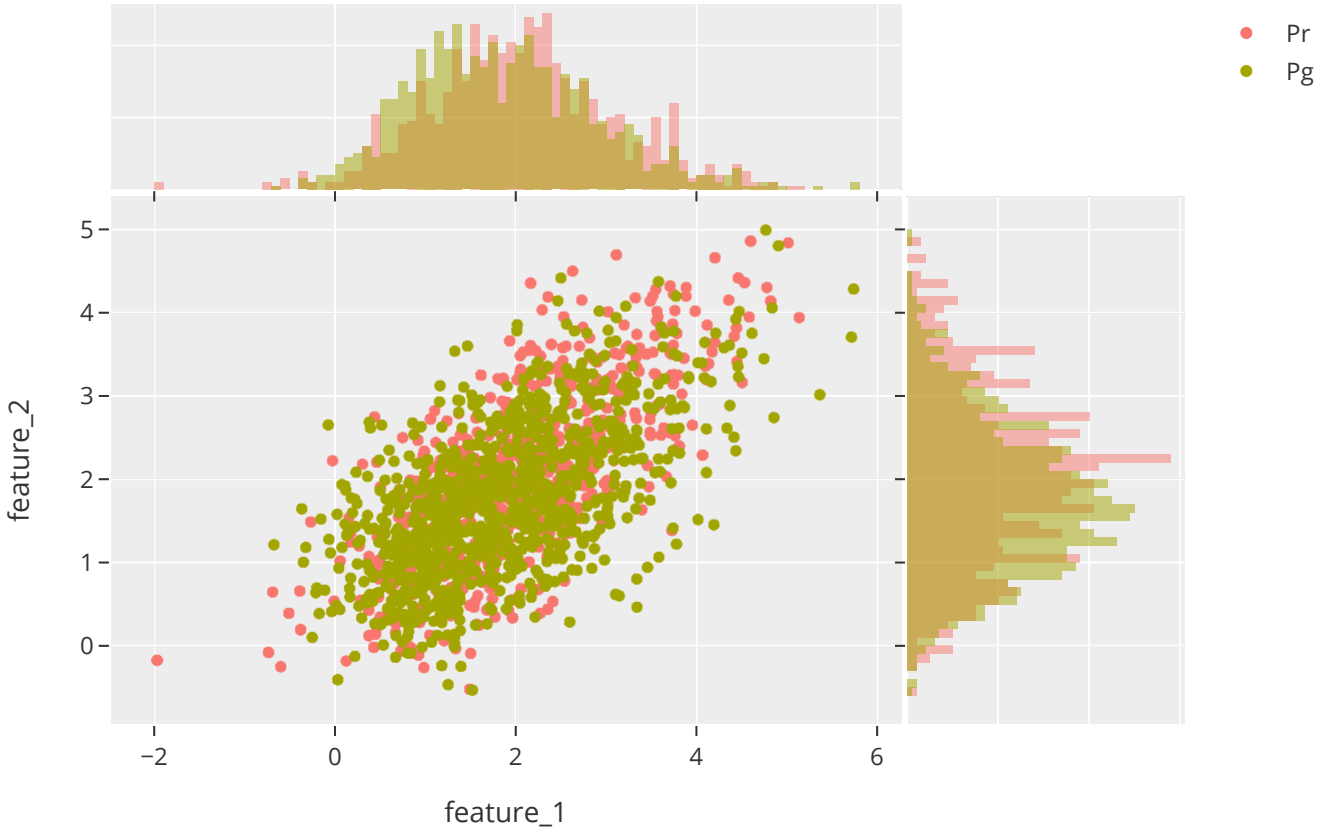
```
In [23]: Pg_new= wgan.generate_data("Pg", n= len(Pr_sampled), as_df= True, to_numpy= True)
         GANvis.plot_scatter(data= (Pr_sampled, Pg_new)).show(config= GANvis.config)
```



We see a lot of improvement, with the *generated* distribution spreading out over the domain space of the *real* distribution, however the correlation pattern is still missing. Let's continue the training for a couple of epochs more.

```
In [ ]: history = wgan.fit(Pr.to_numpy().astype("float32"), batch_size= Pr.shape[0], epochs= 200)
```

```
In [27]: Pg_new= wgan.generate_data("Pg", n= len(Pr_sampled), as_df= True, to_numpy= True)
         GANvis.plot_scatter(data= (Pr_sampled, Pg_new)).show(config= GANvis.config)
```



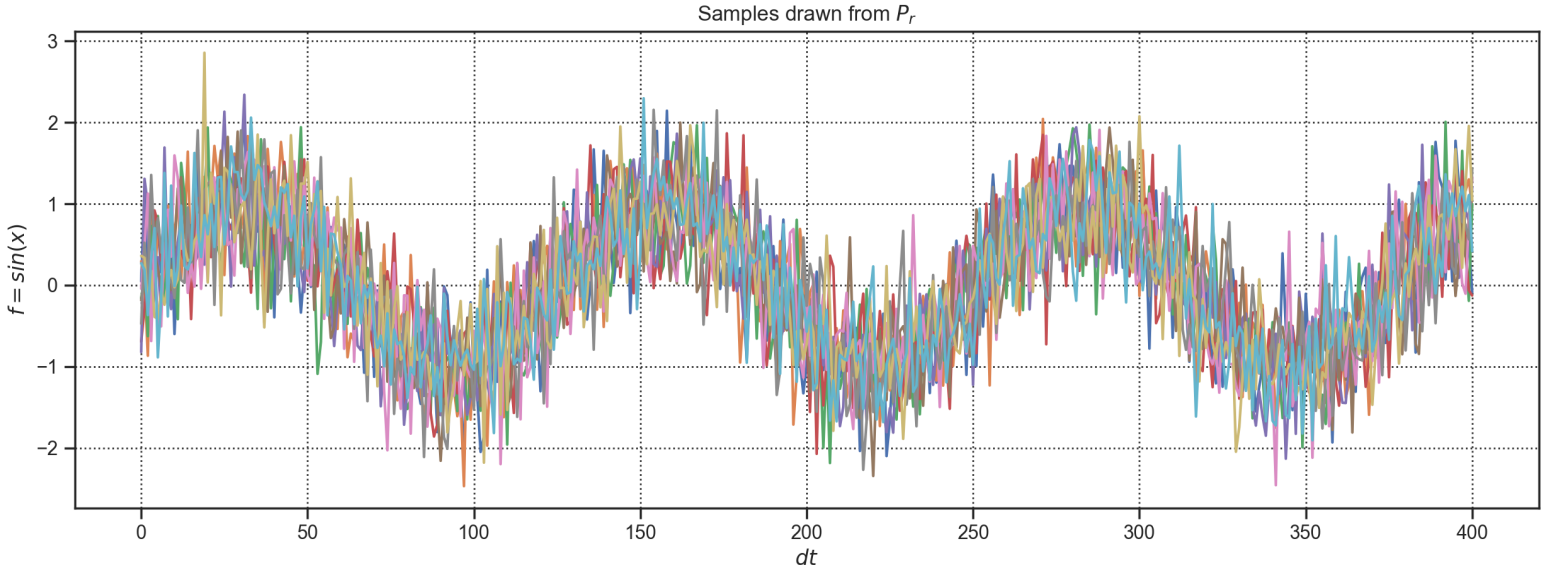
We see a big improvement now, with the `Generator` having learned both the domain space and the correlation pattern of the *real* distribution. Let's move on to timeseries data now.

Chapter 5

Generation of sinus waves

Let's try to generate simple sinus waves. The first step is to simulate a *real* dataset containing noisy sinus waves. The noise is drawn from a standard normal:

```
In [61]: # Step 1: timeseries grid
         n = 1000
         T = 20
         resolution = 20


         v = resolution*T+1
         x = np.linspace([0]*n, [T]*n, v).T

         y = np.sin(x) + np.random.normal(scale= .5, size=(n, v))

         df = pd.DataFrame(y)
         fig, ax= plt.subplots(1, 1, figsize=(15,5))
         ax.plot(df.head(10).T)
         ax.set_title(r"Samples drawn from $P_r$")
         ax.set_xlabel(r"$dt$")
         ax.set_ylabel(r"$f=sin(x)$");
```



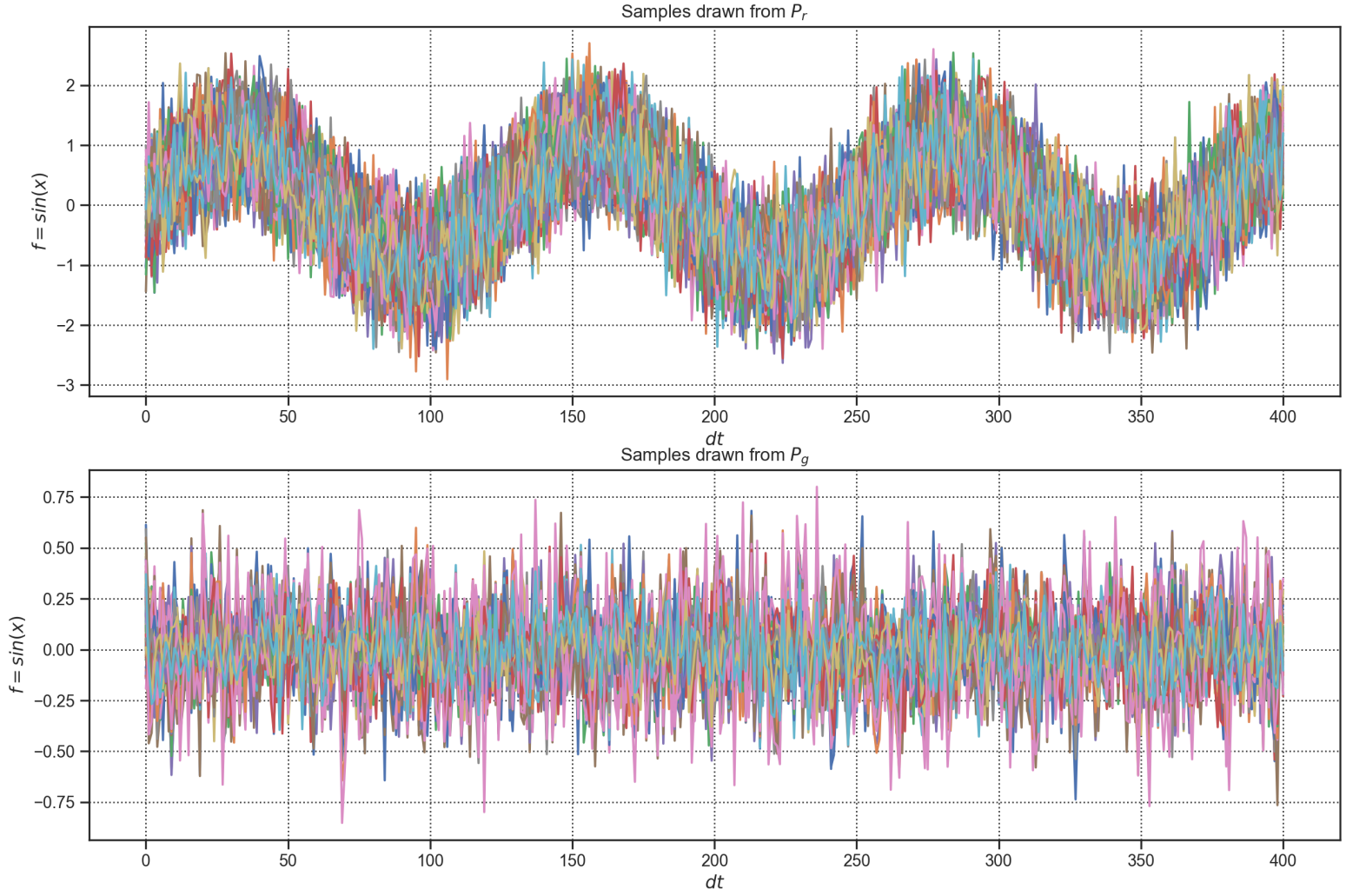
Now we set up our `WGANGP` and generate data from the *real* and *generated* distributions and compare the results:

```
In [16]: wgan = WGANGP(Pr= df, dim_latent_space= 128)
         wgan.init_discriminator()
         wgan.init_generator(output_activation= tfk.activations.linear)
         wgan.init_latent_space("Gaussian")
```

```
2022-04-03 18:49:21 | DEBUG | src.model.wgan_gp:__init__ | line 29 | Initialize tfk.Model
2022-04-03 18:49:21 | DEBUG | src.model.gan_base:__init__ | line 34 | Initialize GANBase class
2022-04-03 18:49:21 | DEBUG | src.model.wgan_gp:__init__ | line 34 | Initialize WGANGP
```

```
In [17]: # Sample real and generated data
         Pr_sampled= wgan.generate_data('Pr', n= 100, as_df= True)
         Pg_sampled= wgan.generate_data('Pg', n= 100, as_df= True)

         fig, ax= plt.subplots(2, 1, figsize=(15,10))
         ax[0].plot(Pr_sampled.T)
```

```
ax[0].set_title(r"Samples drawn from $P_r$")
ax[0].set_xlabel(r"$dt$")
ax[0].set_ylabel(r"$f=sin(x)$")
ax[1].plot(Pg_sampled.T);
ax[1].set_title(r"Samples drawn from $P_g$")
ax[1].set_xlabel(r"$dt$")
ax[1].set_ylabel(r"$f=sin(x)$");
```

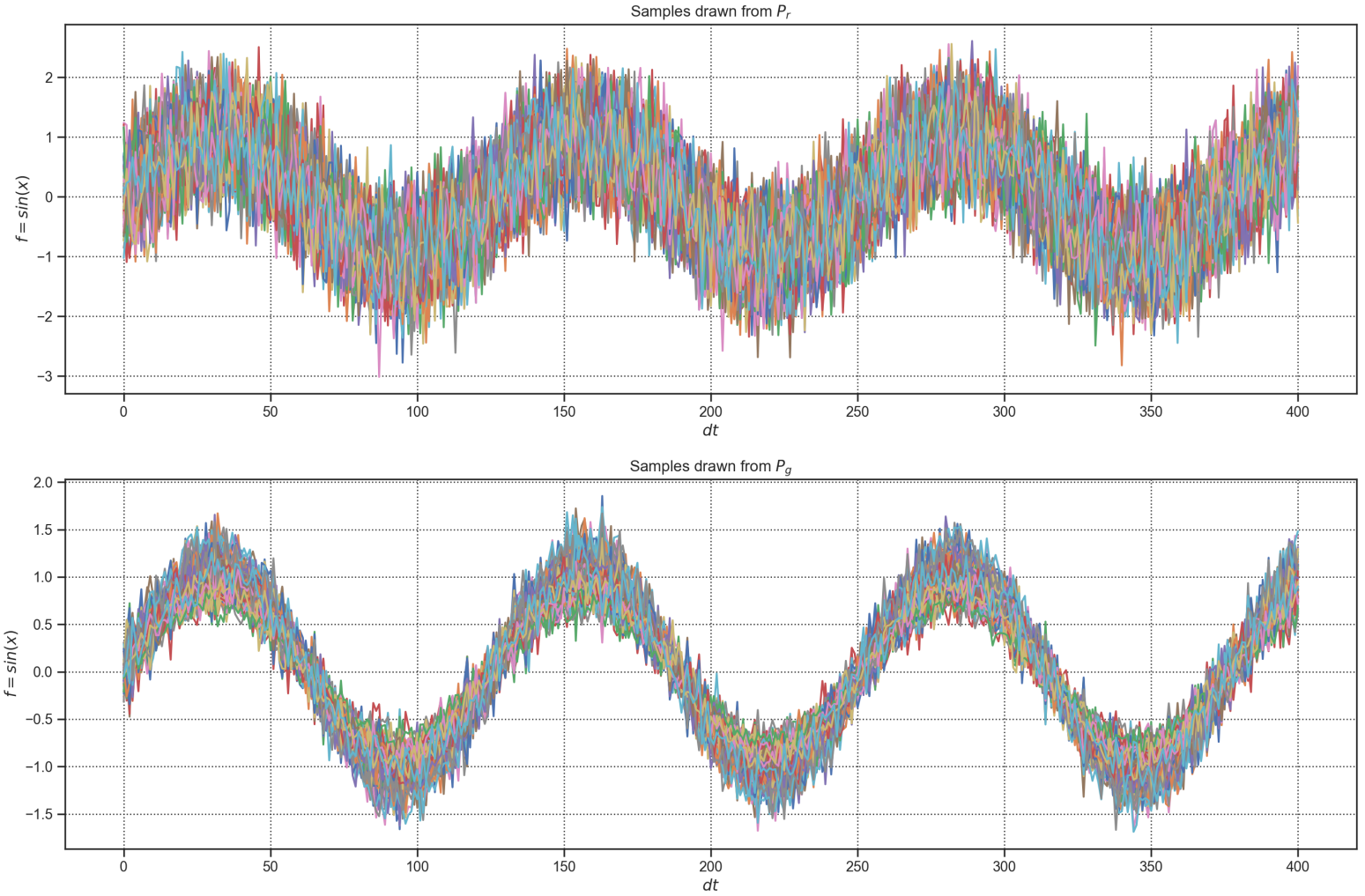


`WGANGP` *generated* timeseries look like pure noise for now. Let's train the neural networks and see if we see improvements:

```
In [ ]: # Define optimizers
        D_opti = tfk.optimizers.Adam(
```

```
        learning_rate=0.0002, beta_1=0.5, beta_2=0.9
    )

    G_opti = tfk.optimizers.Adam(
        learning_rate=0.0002, beta_1=0.5, beta_2=0.9
    )


    # Define losses
    def D_loss(Pr, Pg):
        #return - (tf.reduce_mean(Pg) - tf.reduce_mean(Pr))
        return tf.reduce_mean(Pg) - tf.reduce_mean(Pr)
    def G_loss(Pg):
        return -tf.reduce_mean(Pg)

    wgan.compile(D_opti, G_opti, D_loss, G_loss)

    history = wgan.fit(df.to_numpy().astype("float32"), batch_size= df.shape[0], epochs= 1000)
```

In [19]:
```
 # Sample real and generated data
 Pr_sampled= wgan.generate_data('Pr', n= 100, as_df= True)
 Pg_sampled= wgan.generate_data('Pg', n= 100, as_df= True)

 fig, ax= plt.subplots(2, 1, figsize=(15,10))
 ax[0].plot(Pr_sampled.T)
 ax[0].set_title(r"Samples drawn from $P_r$")
 ax[0].set_xlabel(r"$dt$")
 ax[0].set_ylabel(r"$f=sin(x)$")
 ax[1].plot(Pg_sampled.T);
 ax[1].set_title(r"Samples drawn from $P_g$")
 ax[1].set_xlabel(r"$dt$")
 ax[1].set_ylabel(r"$f=sin(x)$");
 plt.tight_layout()
```



Voila! A big improvement, seems like the `Generator` was able to learn the underlying distribution of the *real* dataset pretty efficiently. Note how the *generated* sinus waves are less noisy compared to original ones, indicating the `WGANGP` is able to infer the true underlying distribution of the *real* dataset, while filtering away noise.

Chapter 6

Generation of autoregressive process

Let's now try generate something that is more in line with typical financial timeseries. To do so we generate a dataset based on the following AR(2) autoregressive proccess:

$$y_{t+1} = y_t w_0 + y_{t-1} w_1 + \epsilon$$

with

$$w_0 = .95, w_1 = -.9$$

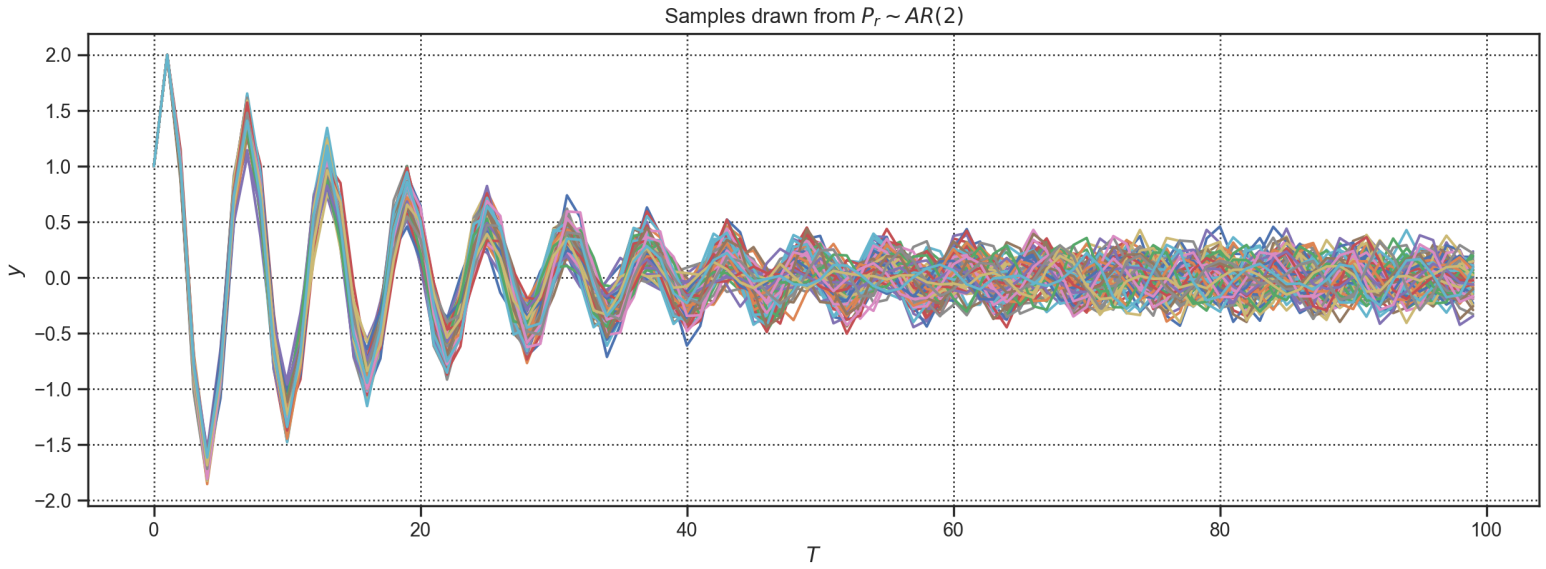
in order to induce *mean-reversion* into the process. For the initial points we simply choose,

$$x_0 = 1, x_1 = 2$$

```
In [62]: from typing import List
         def AR2(n: int, T: int, x0: List[float], w: List[float]) -> np.ndarray:

             x0, x1= x0
             w0, w1= w

             df = []
             for _ in range(n):
                 y = [x0, x1]
                 for _ in range(T-2):
                     y.append(y[-1]*w0 + y[-2]*w1 + np.random.normal(scale= .05))
                 df.append(y)
```

```
        return np.array(df)
```

In [68]:
```
n = 1000
T = 100
w = [.95, -.9]
x0 = [1, 2]

df = AR2(n, T, x0, w)
df = pd.DataFrame(df)
fig, ax= plt.subplots(1, 1, figsize=(15,5))
ax.plot(df.head(100).T)
ax.set_title(r"Samples drawn from $P_r \sim AR(2)$")
ax.set_xlabel(r"$T$")
ax.set_ylabel(r"$y$");
```

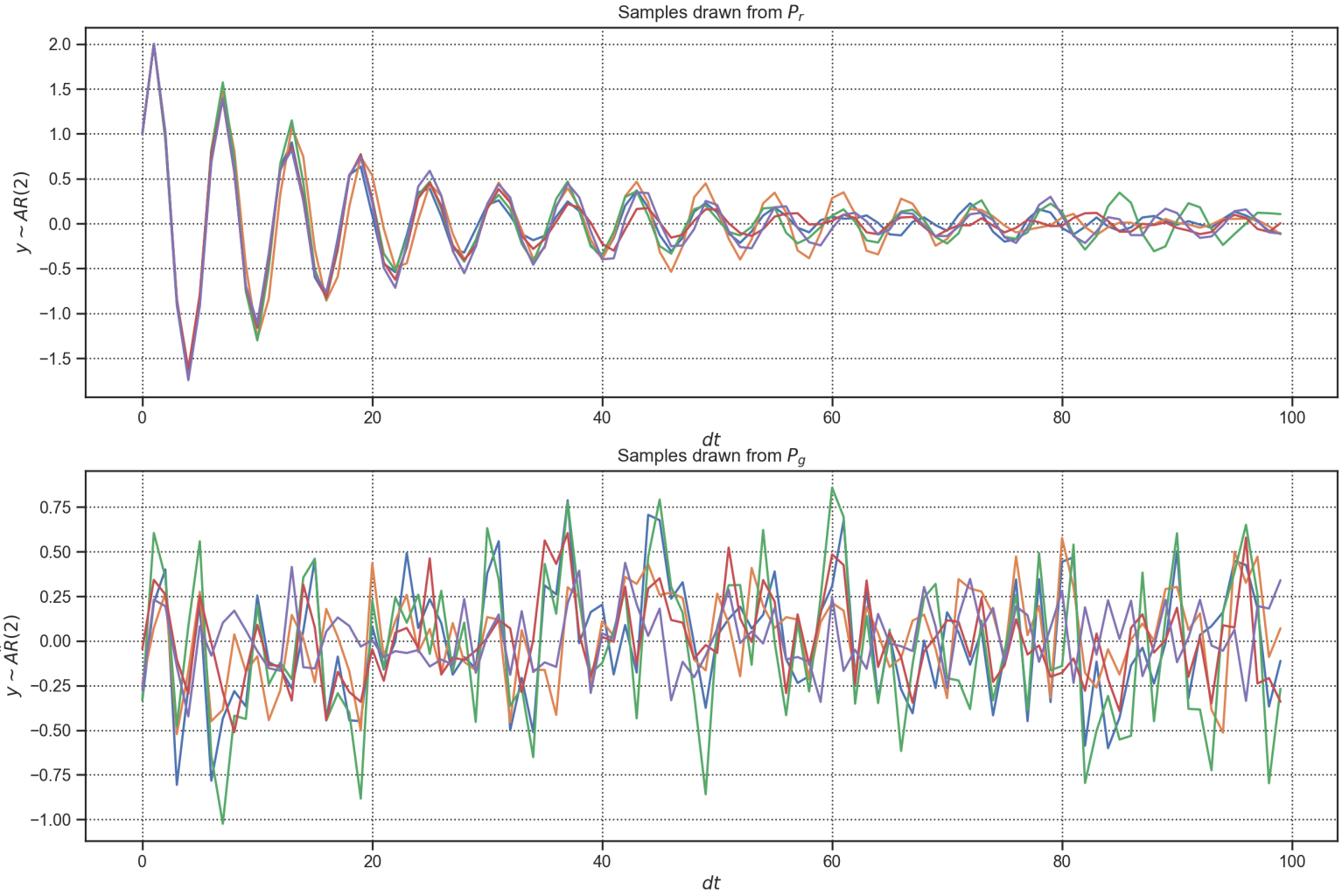


Again, let's setup `WGANGP` and generate from both the *real* and *generated* distribution:

In [71]:
```
wgan = WGANGP(Pr= df, dim_latent_space= 128)
wgan.init_discriminator()
```

```
        wgan.init_generator(output_activation= tfk.activations.linear)
        wgan.init_latent_space("Gaussian")
        # Sample real and generated data
        Pr_sampled= wgan.generate_data('Pr', n= 5, as_df= True)
        Pg_sampled= wgan.generate_data('Pg', n= 5, as_df= True)

        fig, ax= plt.subplots(2, 1, figsize=(15,10))
        ax[0].plot(Pr_sampled.T)
        ax[0].set_title(r"Samples drawn from $P_r$")
        ax[0].set_xlabel(r"$dt$")
        ax[0].set_ylabel(r"$y \sim AR(2)$")
        ax[1].plot(Pg_sampled.T);
        ax[1].set_title(r"Samples drawn from $P_g$")
        ax[1].set_xlabel(r"$dt$")
        ax[1].set_ylabel(r"$y \sim \hat{AR}(2)$");
```

```
2022-04-04 12:06:27 | DEBUG | src.model.wgan_gp:__init__ | line 29 | Initialize tfk.Model
2022-04-04 12:06:27 | DEBUG | src.model.gan_base:__init__ | line 34 | Initialize GANBase class
2022-04-04 12:06:27 | DEBUG | src.model.wgan_gp:__init__ | line 34 | Initialize WGANGP
```

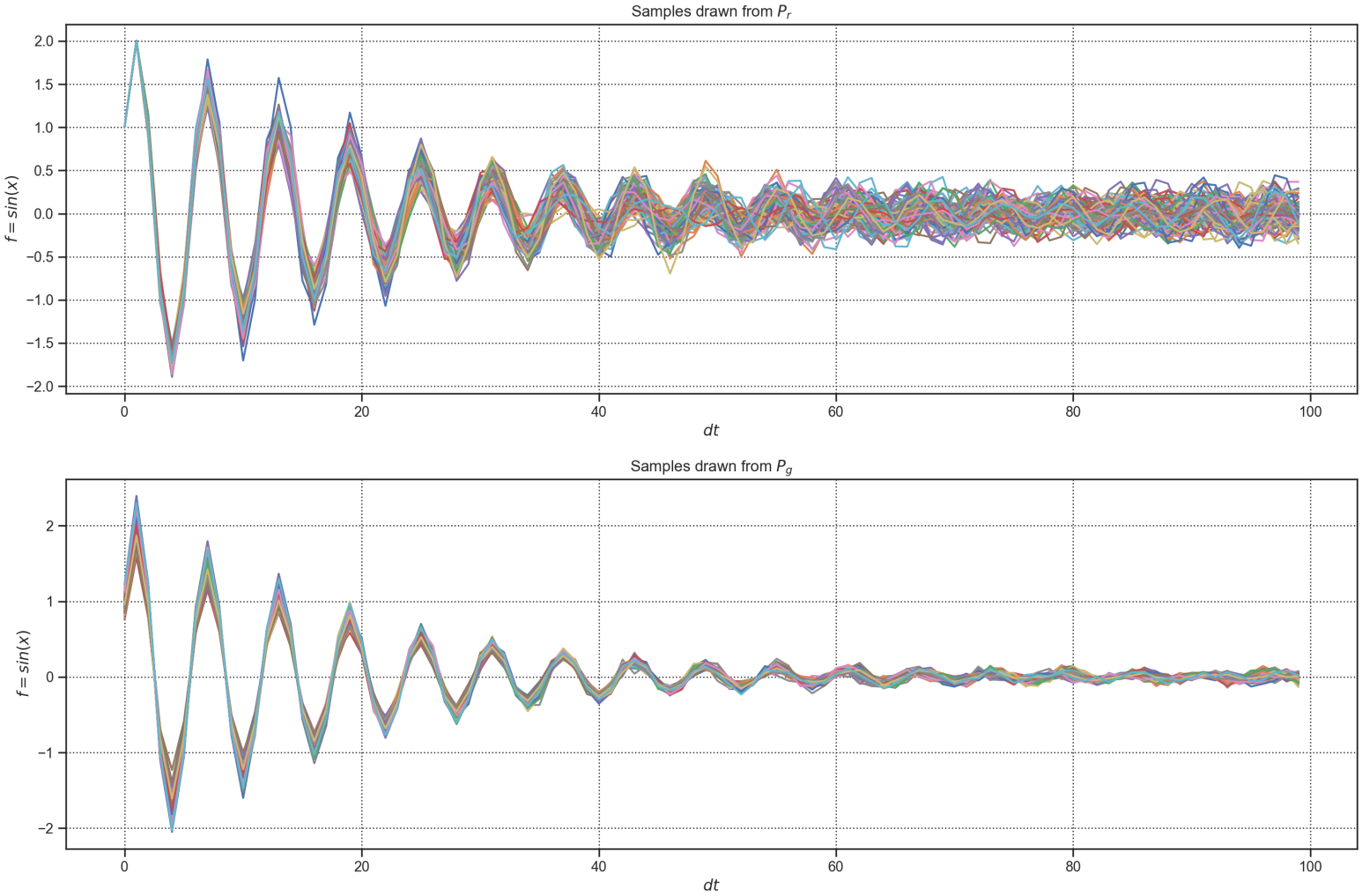


As before we see that the `Generator` is generating random noise when untrained. Let's start the training process and see if there are improvements:

```
In [ ]: # Define optimizers
```

```
        D_opti = tfk.optimizers.Adam(
            learning_rate=0.0002, beta_1=0.5, beta_2=0.9
        )

        G_opti = tfk.optimizers.Adam(
            learning_rate=0.0002, beta_1=0.5, beta_2=0.9
        )


        # Define losses
        def D_loss(Pr, Pg):
            #return - (tf.reduce_mean(Pg) - tf.reduce_mean(Pr))
            return tf.reduce_mean(Pg) - tf.reduce_mean(Pr)
        def G_loss(Pg):
            return -tf.reduce_mean(Pg)

        wgan.compile(D_opti, G_opti, D_loss, G_loss)

        history = wgan.fit(df.to_numpy().astype("float32"), batch_size= df.shape[0], epochs= 1000)
```

In [25]:
```
        # Sample real and generated data
        Pr_sampled= wgan.generate_data('Pr', n= 100, as_df= True)
        Pg_sampled= wgan.generate_data('Pg', n= 100, as_df= True)

        fig, ax= plt.subplots(2, 1, figsize=(15,10))
        ax[0].plot(Pr_sampled.T)
        ax[0].set_title(r"Samples drawn from $P_r$")
        ax[0].set_xlabel(r"$dt$")
        ax[0].set_ylabel(r"$y \sim AR(2)$")
        ax[1].plot(Pg_sampled.T);
        ax[1].set_title(r"Samples drawn from $P_g$")
        ax[1].set_xlabel(r"$dt$")
        ax[1].set_ylabel(r"$y \sim \hat{AR}(2)$");
        plt.tight_layout()
```



And again, we see that `Generator` has learned the real distribution, including it's mean reverting characteristic.

Chapter 7

Generating yield curves

Now we can move on to actual real-world data. We fetch daily historical yield curve data from the ECB. The data spans from 2004 all the way to 2022.

The data in its raw form is unsuitable to inject into WGANGP, so let's first do some cleaning up:

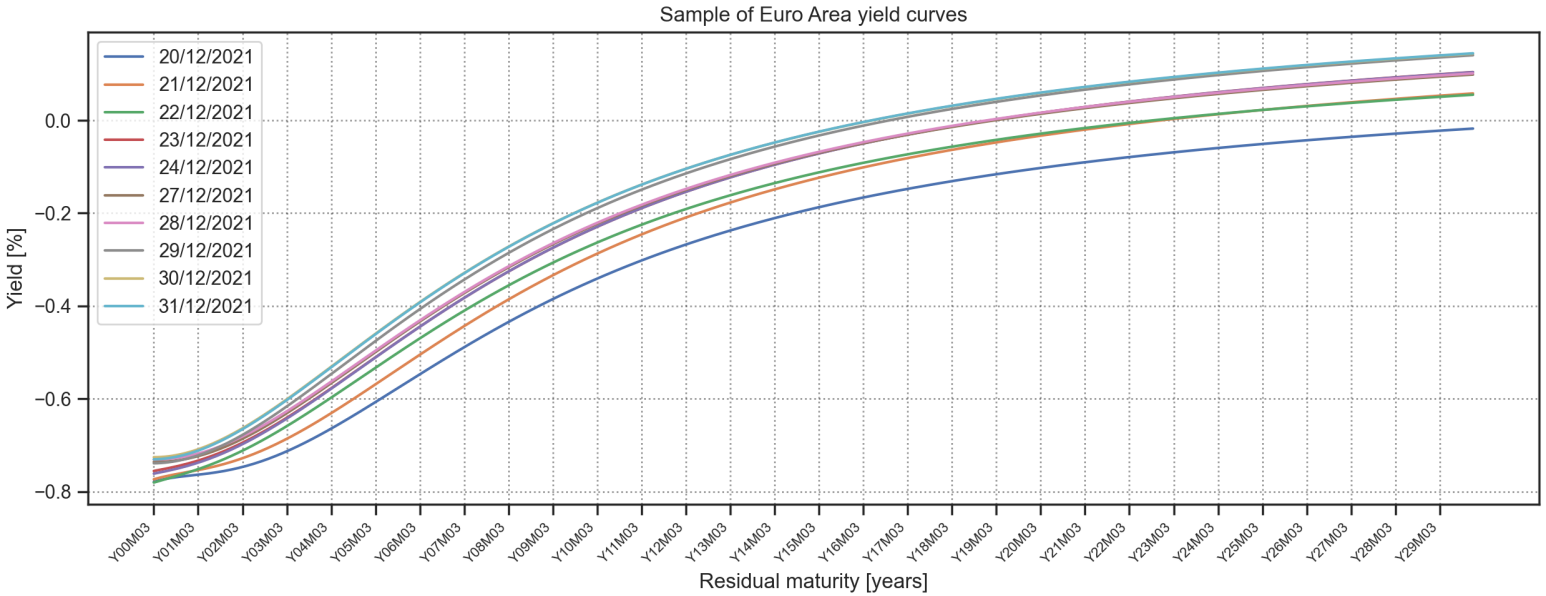
```
In [72]: file = r"C:\Users\aarabil\Desktop\data (2).csv"
         df = pd.read_csv(file, usecols=["TIME_PERIOD", "OBS_VALUE", "DATA_TYPE_FM", "INSTRUMENT_FM"])
         df = df.infer_objects()
         df = df[(df.DATA_TYPE_FM.str.startswith("SR_")) & (df.INSTRUMENT_FM == "G_N_A")]\
             .drop("INSTRUMENT_FM", axis=1)

In [73]: # Get Years
         df["y"] = df.DATA_TYPE_FM.str.extract(r"_(\d+)Y")
         df.y.fillna(0, inplace=True)

         # Get Months
         df["m"] = df.DATA_TYPE_FM.str.extract(r"[Y|_](\d+)M")
         df.m.fillna(0, inplace= True)

         # Create new variable
         df["SR"] = "SR_Y" + df.y.apply('{:0>2}'.format) + "M" + df.m.apply('{:0>2}'.format)
         df.drop(["y", "m", "DATA_TYPE_FM"], axis = 1, inplace= True)

         # index date as proper timeseries
         df.TIME_PERIOD = pd.to_datetime(df.TIME_PERIOD)
         df.set_index("TIME_PERIOD", inplace= True)
```

```
        # Pivot to wide format
        df = df.sort_values(by= "SR")
        df = df.pivot(columns= "SR", values= "OBS_VALUE")
```

After cleaning we're left with daily data on yield curves that cover maturities ranging from 3months to 30years. The total number of observations however is only 4430 which is quite limited when talking deep learning. Let's plot the last 10 days:

```
In [93]: df_plot = df.tail(10).T
         df_plot.index = df_plot.index.str[3:]
         df_plot.head()

         fig, ax = plt.subplots(1,1, figsize=(15,5))


         ax.plot(df_plot)


         # Major ticks every 20, minor ticks every 5
         major_ticks = np.arange(0, 12*30, 12)
         minor_ticks = np.arange(0, 101, 5)

         ax.set_xticks(np.arange(0, 12*30, 12), minor=False)
         # ax.yticks(major_ticks)
         # ax.yticks(minor=False)

         # And a corresponding grid
         ax.grid(which='both')

         # Or if you want different settings for the grids:
         ax.grid(which='minor', alpha=0.2)
         ax.grid(which='major', alpha=0.5)

         for ax in fig.get_axes():
             if ax.get_subplotspec().is_last_row():
                 for label in ax.get_xticklabels():
                     label.set_ha('right')
                     label.set_rotation(45.)
             else:
                 for label in ax.get_xticklabels():
```

```
            label.set_visible(False)
        ax.set_xlabel('')

ax.tick_params(axis='x', which='major', labelsize=8, labelbottom=True)
ax.set_title("Sample of Euro Area yield curves")
ax.set_xlabel("Residual maturity [years]")
ax.set_ylabel("Yield [%]")
ax.legend(df_plot.columns.strftime('%d/%m/%Y'));
```

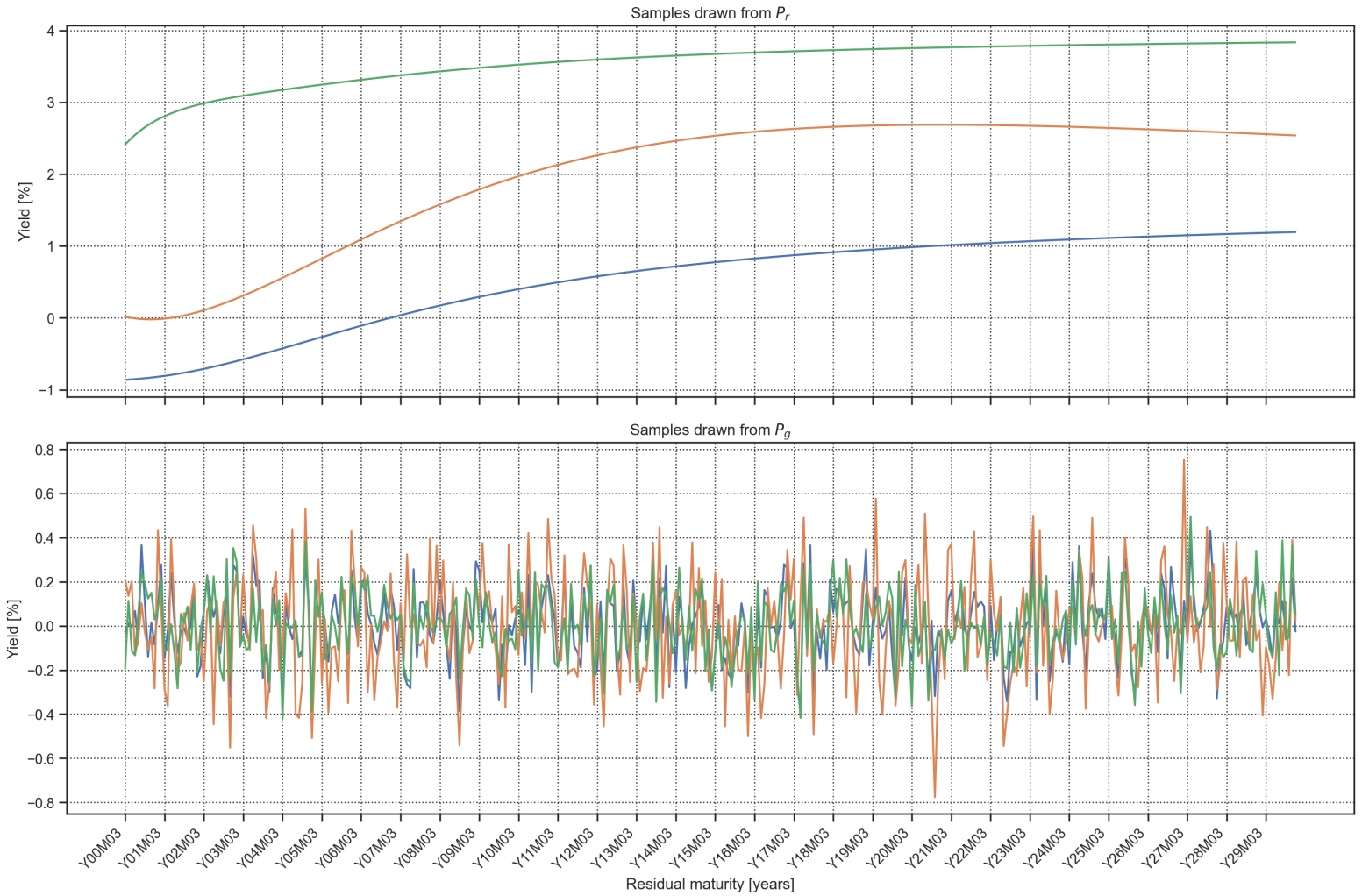


With the data in the right format, we can simply keep using the same procedure as before to set up our `WGANGP`:

```
In [101]: df.reset_index(drop=True, inplace= True)
          df.columns = [col[3:] for col in df.columns]

In [104]: wgan = WGANGP(Pr= df, dim_latent_space= 128)
          wgan.init_discriminator()
          wgan.init_generator(output_activation= tfk.activations.linear)
          wgan.init_latent_space("Gaussian")
```

```
        # Sample real and generated data
        Pr_sampled= wgan.generate_data('Pr', n= 3, as_df= True)
        Pg_sampled= wgan.generate_data('Pg', n= 3, as_df= True)

        fig, ax= plt.subplots(2, 1, figsize=(15,10))
        ax[0].plot(Pr_sampled.T)
        ax[0].set_xticks(np.arange(0, 12*30, 12), minor=False)
        #ax[0].tick_params(axis='x', which='major', labelsize=8, labelbottom=False)
        ax[0].set_title(r"Samples drawn from $P_r$")
        ax[0].set_xlabel(r"Residual maturity [years]")
        ax[0].set_ylabel(r"Yield [%]")
        ax[1].plot(Pg_sampled.T);
        ax[1].set_xticks(np.arange(0, 12*30, 12), minor=False)
        #ax[1].tick_params(axis='x', which='major', labelsize=8, labelbottom=False)
        ax[1].set_title(r"Samples drawn from $P_g$")
        ax[1].set_xlabel(r"Residual maturity [years]")
        ax[1].set_ylabel(r"Yield [%]");

        for ax in fig.get_axes():
            if ax.get_subplotspec().is_last_row():
                for label in ax.get_xticklabels():
                    label.set_ha('right')
                    label.set_rotation(45.)
            else:
                for label in ax.get_xticklabels():
                    label.set_visible(False)
                    ax.set_xlabel("")


        plt.tight_layout()
```

```
2022-04-04 12:35:19 | DEBUG | src.model.wgan_gp:__init__ | line 29 | Initialize tfk.Model
2022-04-04 12:35:19 | DEBUG | src.model.gan_base:__init__ | line 34 | Initialize GANBase class
2022-04-04 12:35:19 | DEBUG | src.model.wgan_gp:__init__ | line 34 | Initialize WGANGP
```



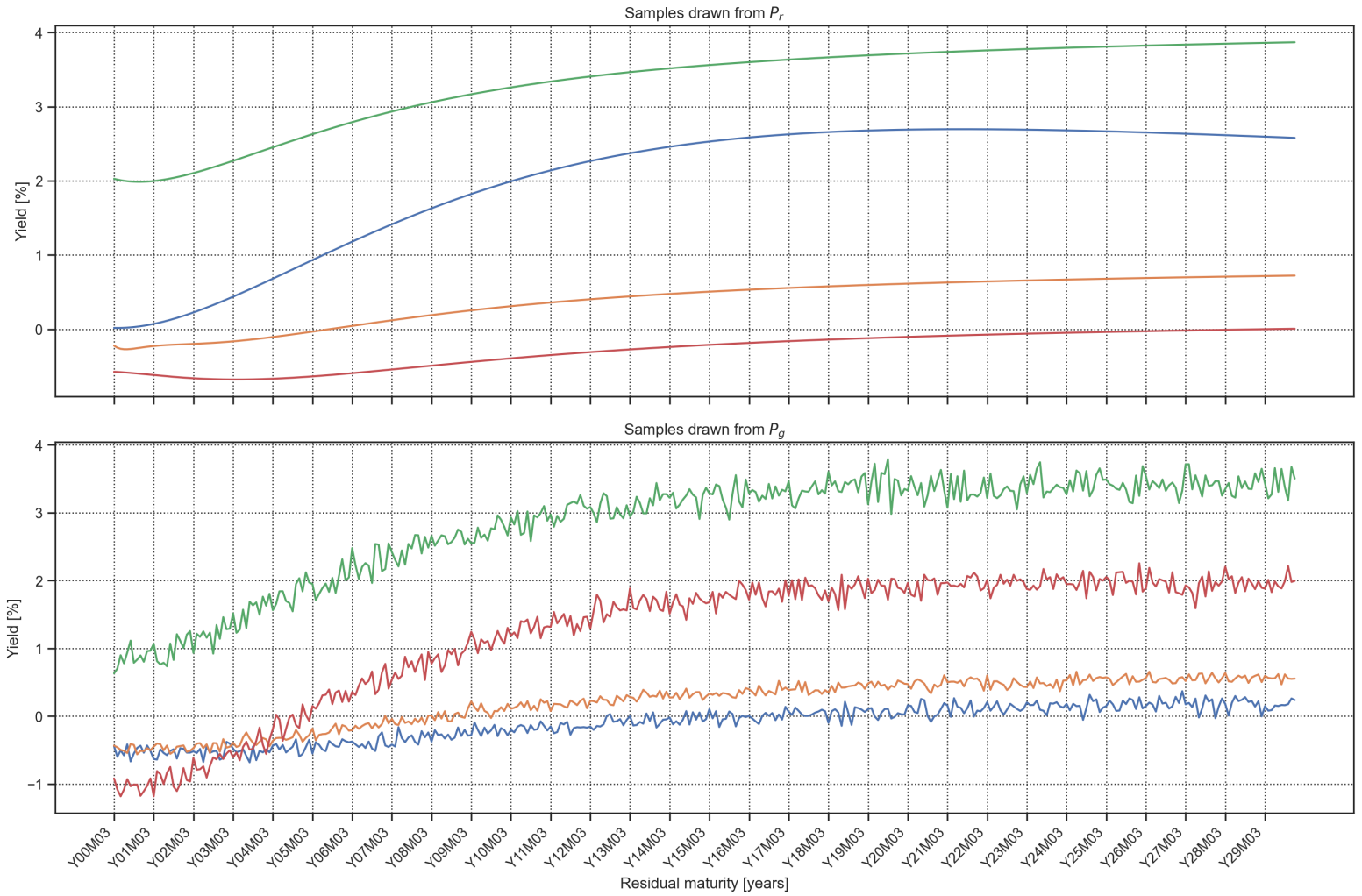
Again pure noise from the side of the `Generator` before training. Let's start the training now:

```
In [ ]: # Define optimizers
        D_opti = tfk.optimizers.Adam(
```

```
        learning_rate=0.0002, beta_1=0.5, beta_2=0.9
    )

    G_opti = tfk.optimizers.Adam(
        learning_rate=0.0002, beta_1=0.5, beta_2=0.9
    )


    # Define losses
    def D_loss(Pr, Pg):
        #return - (tf.reduce_mean(Pg) - tf.reduce_mean(Pr))
        return tf.reduce_mean(Pg) - tf.reduce_mean(Pr)
    def G_loss(Pg):
        return -tf.reduce_mean(Pg)

    wgan.compile(D_opti, G_opti, D_loss, G_loss)

    history = wgan.fit(df.to_numpy().astype("float32"), batch_size= df.shape[0], epochs= 1000)
```

In [112]:
```
# Sample real and generated data
Pr_sampled= wgan.generate_data('Pr', n= 4, as_df= True)
Pg_sampled= wgan.generate_data('Pg', n= 4, as_df= True)

fig, ax= plt.subplots(2, 1, figsize=(15,10))
ax[0].plot(Pr_sampled.T)
ax[0].set_xticks(np.arange(0, 12*30, 12), minor=False)
#ax[0].tick_params(axis='x', which='major', labelsize=8, labelbottom=False)
ax[0].set_title(r"Samples drawn from $P_r$")
ax[0].set_xlabel(r"Residual maturity [years]")
ax[0].set_ylabel(r"Yield [%]")
ax[1].plot(Pg_sampled.T);
ax[1].set_xticks(np.arange(0, 12*30, 12), minor=False)
#ax[1].tick_params(axis='x', which='major', labelsize=8, labelbottom=False)
ax[1].set_title(r"Samples drawn from $P_g$")
ax[1].set_xlabel(r"Residual maturity [years]")
ax[1].set_ylabel(r"Yield [%]");

for ax in fig.get_axes():
    if ax.get_subplotspec().is_last_row():
```

```
        for label in ax.get_xticklabels():
            label.set_ha('right')
            label.set_rotation(45.)
    else:
        for label in ax.get_xticklabels():
            label.set_visible(False)
            ax.set_xlabel("")


plt.tight_layout()
```



Seems like `WGANGP` is capable of training a `Generator` that can output realistic yield curves, albeit a bit noisy.