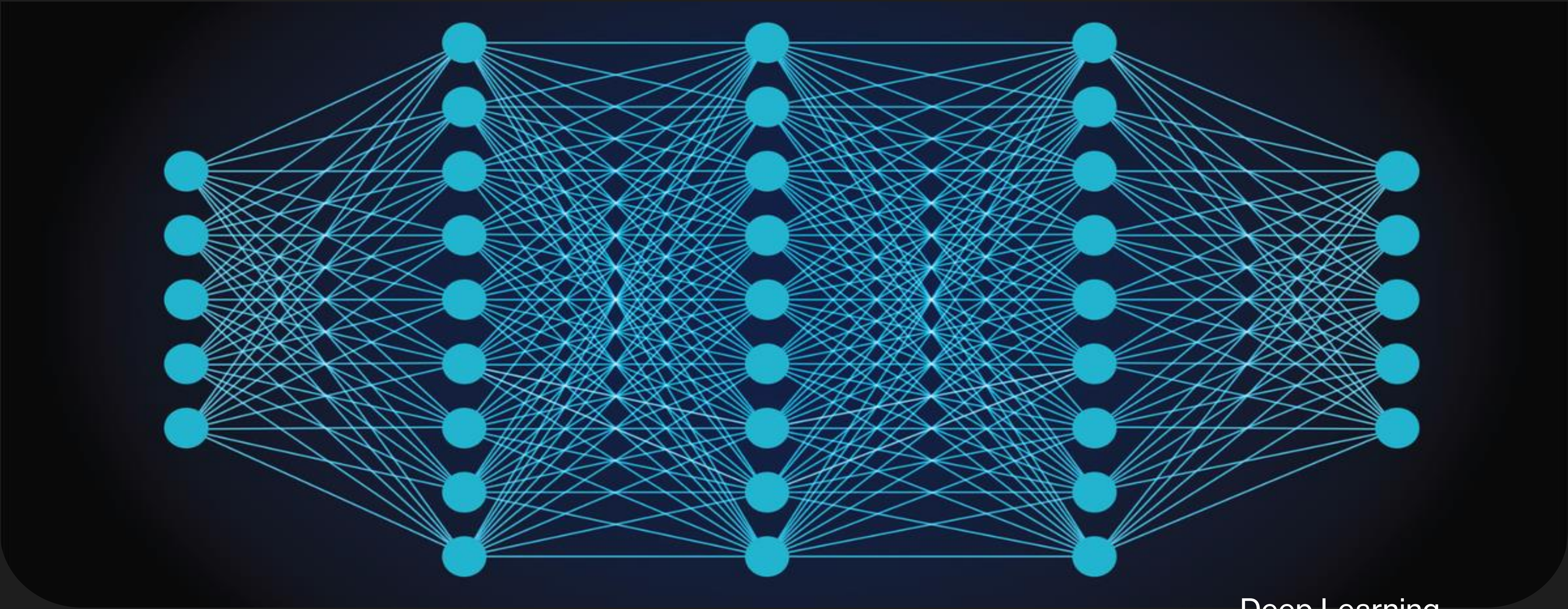
WGAN for MNIST and FINANCE



Deep Learning

Index

1 The Problem of Chaotic High Dimentionality Systems

The issue is that the real data distribution lies on a complex, high-dimensional manifold that standard statistical models fail to capture.

We do not know the physical equations governing handwritten digits or market returns.

Standard generative models often fail to converge on these complex distributions. We need a mathematically stable framework to ensure the model captures the structure without collapsing.

Use Wasserstein GAN to perform Implicit Density Estimation, learning to sample from P_{data}.

We validate the architecture on a controlled system. Generating sharp digits proves the WGAN can learn spatial correlations.

Stochastic Application:

Once validated, we transfer the architecture to Financial Time Series, creating a "Market Simulator" that learns temporal dependencies and volatility clustering from scratch.

2 Problems of GANs

The problem is the gradient vanished, which leads to instability in the convergence of the Jensen-Shannon Divergence:

$$\min_G \max_D V(D,G) = \mathrm{E}_{x\sim Pr}[\log D(x)] + \mathrm{E}_{z\sim Pz}[\log(1-D(G(z)))]$$

In this situation the Discriminator D acts as a Binary Classifier by a Sigmoid output.

This causes problems In high-dimensional spaces, Real and Generated distributions often have disjoint supports.

When distributions don't overlap, the JSD is constant, which leads to a zero gradient. In this situation the Generator stops learning

Dfferences with WGANs

Changing the Loss Function with the Wasserstain's one:

$$L = E_z[D(G(z))] - E_x[D(x)] + \lambda \cdot E_x\hat{}[(||\nabla_x D(x)||_2 - 1)^2]$$

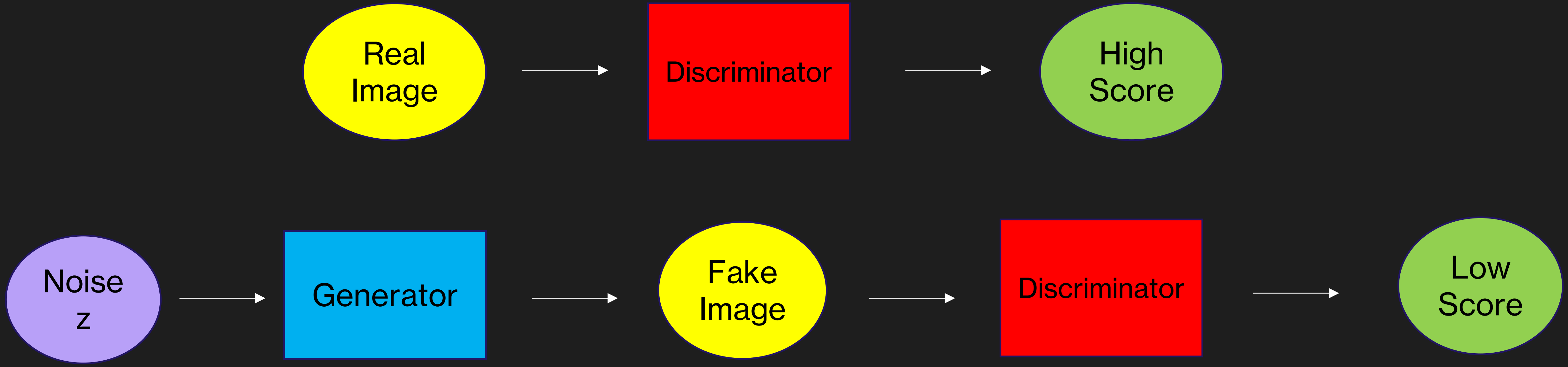
where the second term has to be there to apply the constrain that D must be Lipschitz.

Penalizing the gradient norm if it deviates from 1.

The Critic acts as a Potential Function with linear output, no Sigmoid.

This provides smooth, non-zero gradients everywhere.

The Structure of the Neural Network

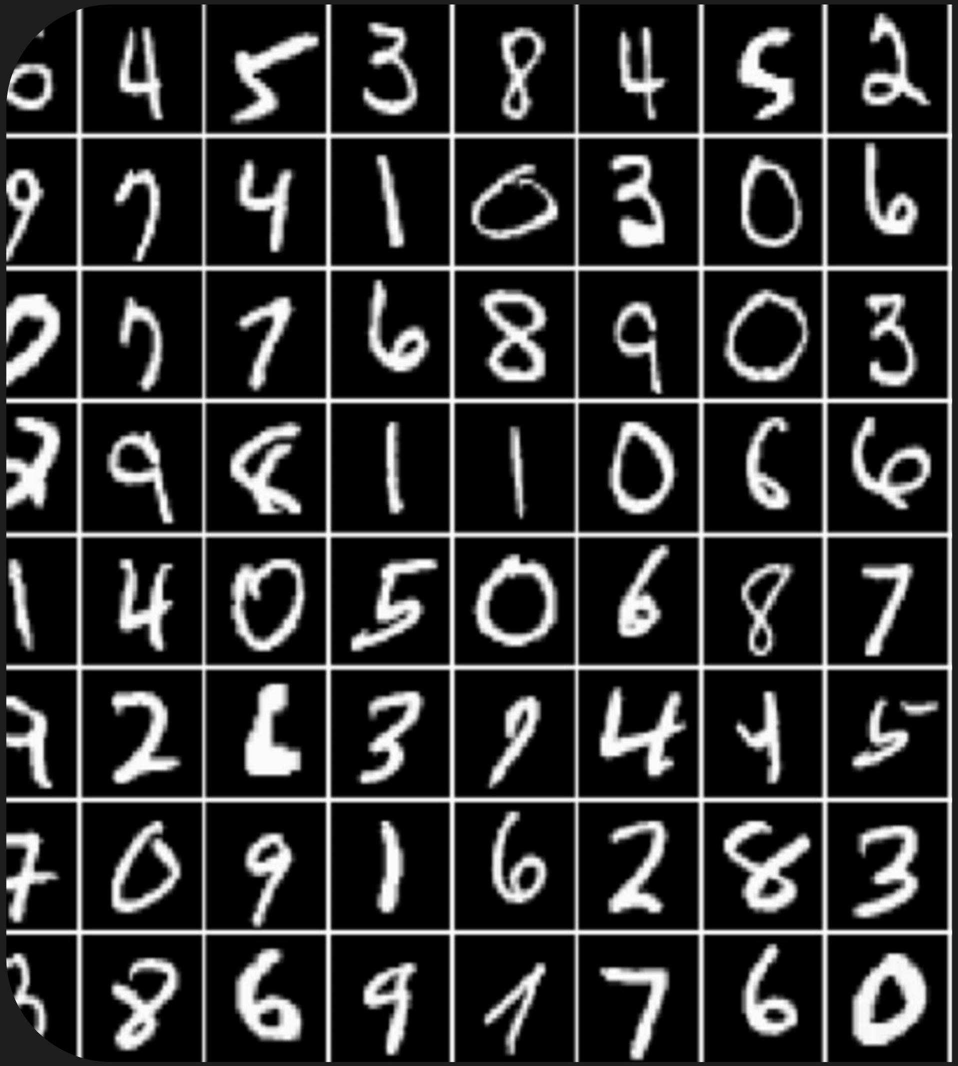


4 Data Implementation MNIST

The dataset was loaded via the tf.keras.datasets.mnist library

Input values in the range [0, 255] were converted to real values and normalized to the range [-1, 1] to match the tanh activation function in the generator

The dataset was divided into batches and shuffled to prevent correlation between samples



Architecture

The Generator

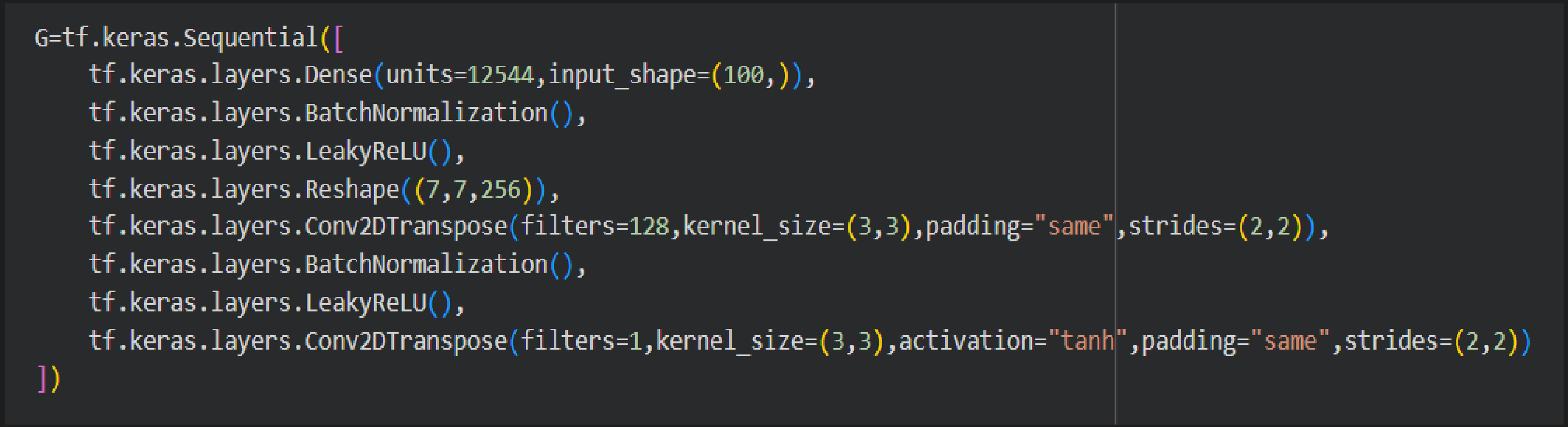
The Conv2DTranspose is used to expand the dimensions of the noise input.

The dimensions evolve as follows:

From 100 to 7 x 7 x 256 via the Reshape layer.

The Conv2DTranspose transforms it to 14 x 14 x 128.

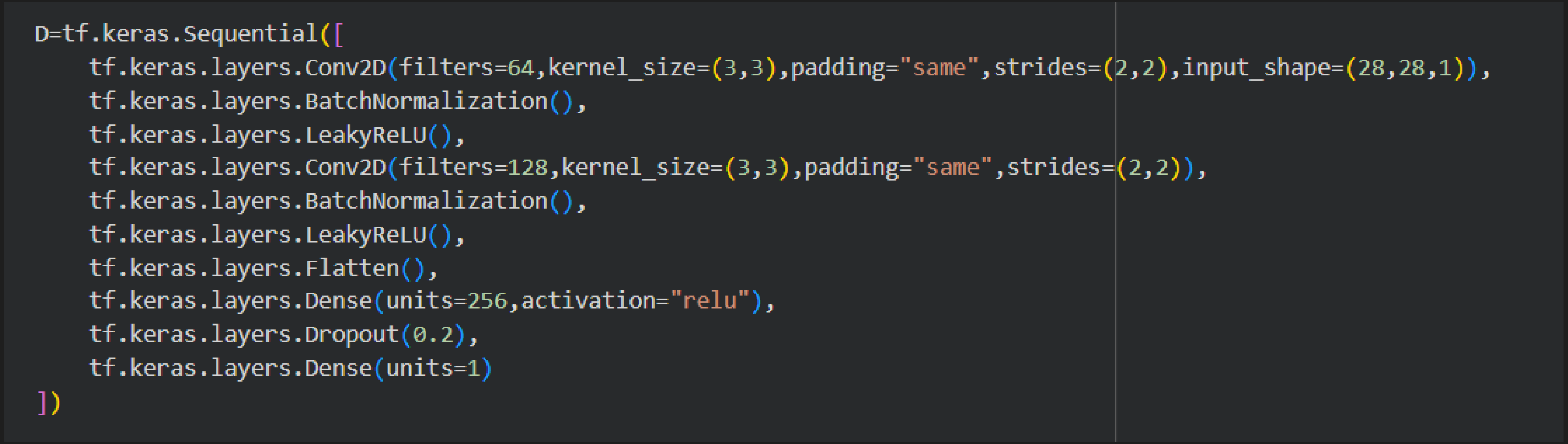
Finally, it reaches 28 x 28 x 1, creating the output image.



```
G=tf.keras.Sequential([
    tf.keras.layers.Dense(units=12544,input_shape=(100,)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Reshape((7,7,256)),
    tf.keras.layers.Conv2DTranspose(filters=128,kernel_size=(3,3),padding="same",strides=(2,2)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2DTranspose(filters=1,kernel_size=(3,3),activation="tanh",padding="same",strides=(2,2))
])
```

The Discriminator

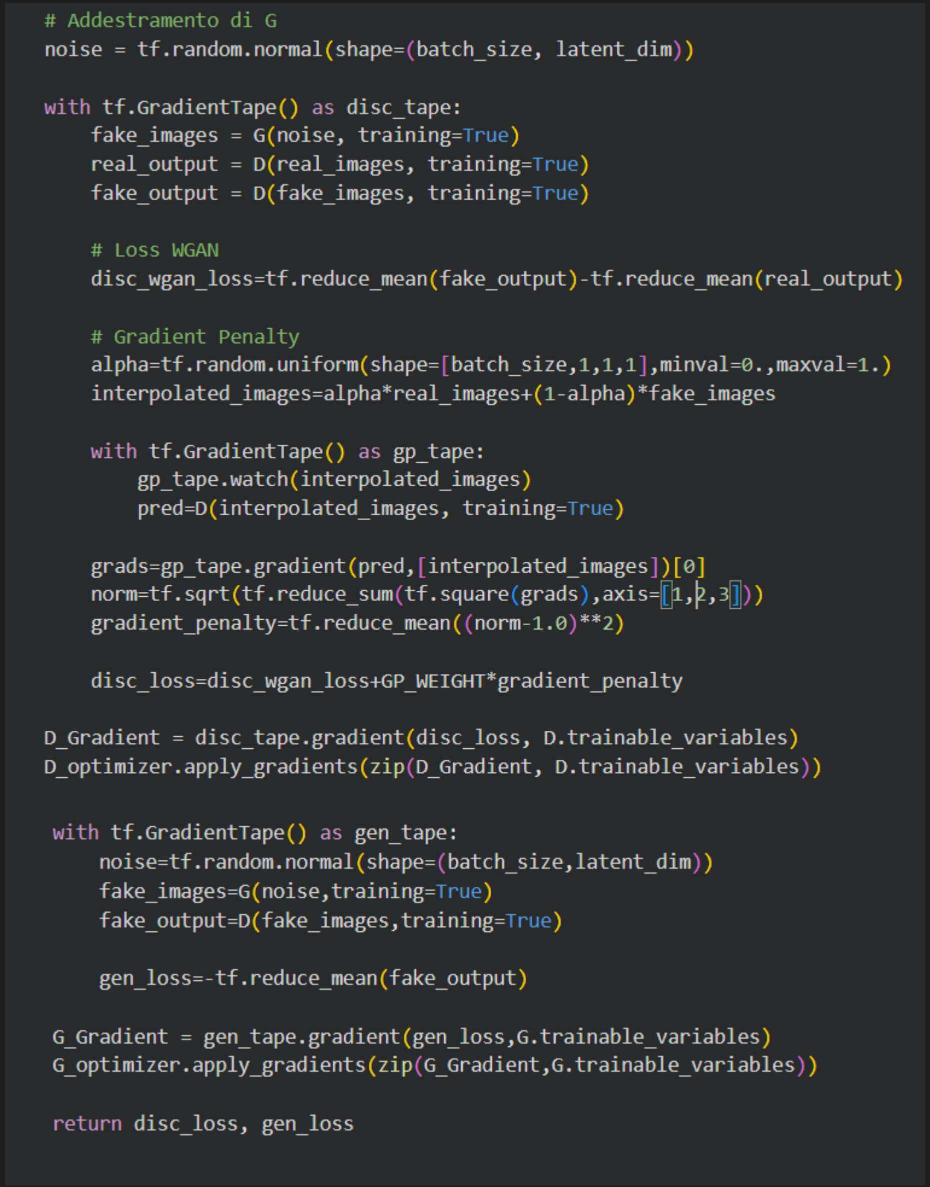
It compresses the input image to evaluate the final output.
The main difference between standard GAN and WGAN is the absence of the Sigmoid function at the end.
Consequently, the final output is not a probability in [0,1], but a value ranging in (-∞,+ ∞).



```python
D=tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=64,kernel_size=(3,3),padding="same",strides=(2,2),input_shape=(28,28,1)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv2D(filters=128,kernel_size=(3,3),padding="same",strides=(2,2)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=256,activation="relu"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=1)
])
```

Training

The Generator and Critic engage in an adversarial minimax game to minimize their respective loss functions. This iterative process improves the Generator's ability to approximate the real data distribution.
The 1-Lipschitz constraint is enforced via Gradient Penalty.
The gradient norm || ∇D || is constrained to 1 by sampling interpolated points long the lines connecting real and generated data.



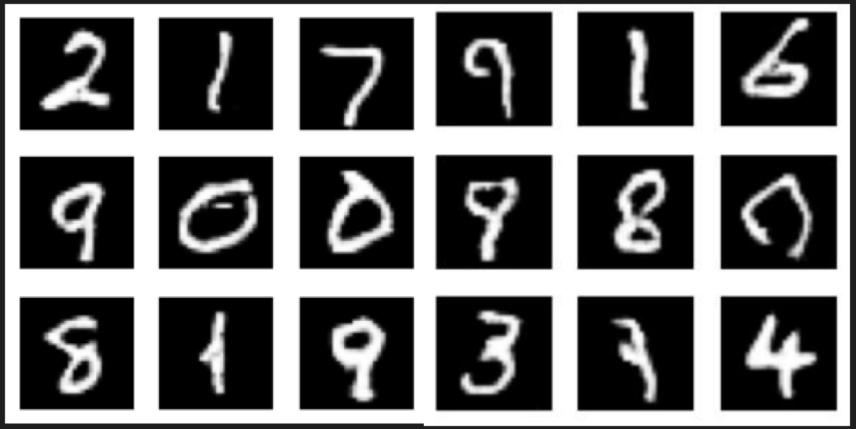
```python
# Addestramento di G
noise = tf.random.normal(shape=(batch_size, latent_dim))

with tf.GradientTape() as disc_tape:
    fake_images = G(noise, training=True)
    real_output = D(real_images, training=True)
    fake_output = D(fake_images, training=True)

    # Loss WGAN
    disc_wgan_loss=tf.reduce_mean(fake_output)-tf.reduce_mean(real_output)

    # Gradient Penalty
    alpha=tf.random.uniform(shape=[batch_size,1,1,1],minval=0.,maxval=1.)
    interpolated_images=alpha*real_images+(1-alpha)*fake_images

    with tf.GradientTape() as gp_tape:
        gp_tape.watch(interpolated_images)
        pred=D(interpolated_images, training=True)

    grads=gp_tape.gradient(pred,[interpolated_images])[0]
    norm=tf.sqrt(tf.reduce_sum(tf.square(grads),axis=[1,2,3]))
    gradient_penalty=tf.reduce_mean((norm-1.0)**2)

    disc_loss=disc_wgan_loss+GP_WEIGHT*gradient_penalty

D_Gradient = disc_tape.gradient(disc_loss, D.trainable_variables)
D_optimizer.apply_gradients(zip(D_Gradient, D.trainable_variables))

 with tf.GradientTape() as gen_tape:
     noise=tf.random.normal(shape=(batch_size,latent_dim))
     fake_images=G(noise,training=True)
     fake_output=D(fake_images,training=True)

     gen_loss=-tf.reduce_mean(fake_output)

 G_Gradient = gen_tape.gradient(gen_loss,G.trainable_variables)
 G_optimizer.apply_gradients(zip(G_Gradient,G.trainable_variables))

 return disc_loss, gen_loss
```

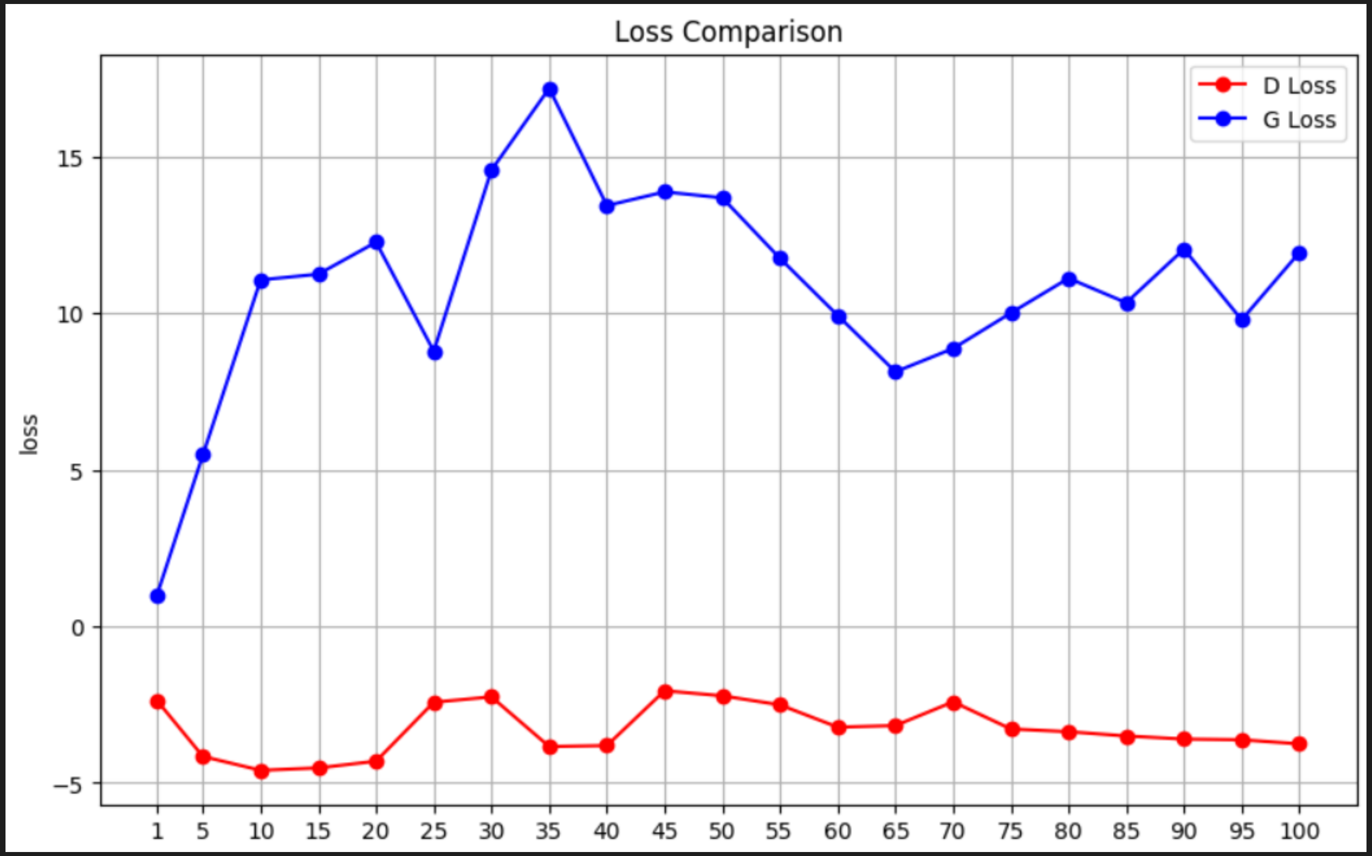
HyperParameters

The following configuration was adopted for the training phase:

- Learning Rate: η = 0.0001
- Learning rate first momentum: $\beta_1=0.5$
- Epochs: n=150
- Batch Size: n_B=256
- GP weight: λ=10 (standard approach)

4 MNIST Results





The FID of 37.86 confirms high-quality generation: the model successfully learned the digit structure.

Note the stable descent of the Critic Loss: this proves the Vanishing Gradient is solved, whereas the absolute value of the Generator Loss is irrelevant for monitoring convergence.

5 Financial Application

Now, we apply the same approach to a completely different domain: the financial market.
The goal is to generate different scenarios of log returns for the S&P 500 that possess the same statistical properties. We are not predicting future values, but simulating possible scenarios to perform statistical analysis.For this application, daily logarithmic returns were used:

$$r_t = \ln\left(\frac{P_t}{P_{t-1}}\right)$$

Dataset Implementation

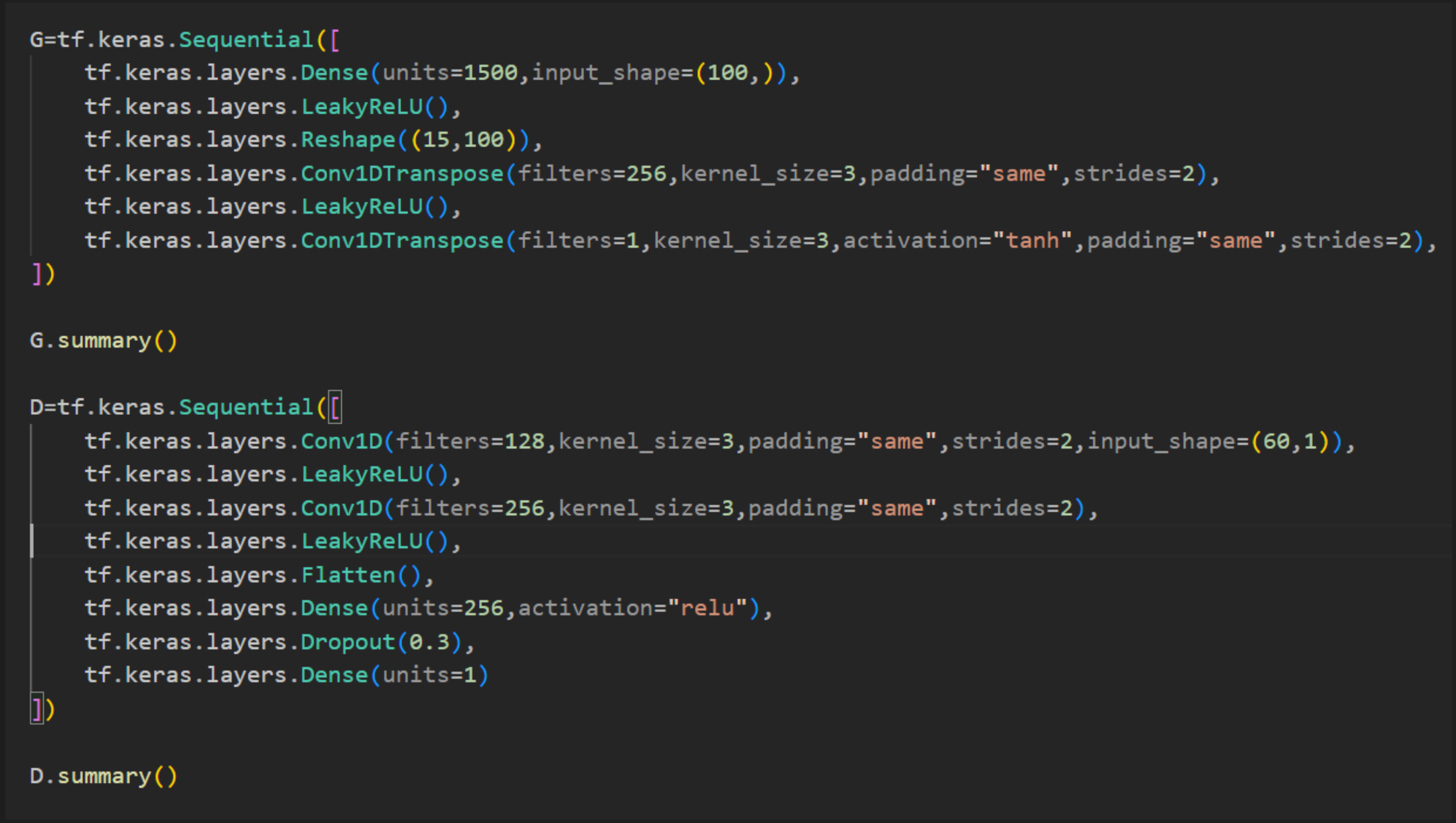
Data Acquisition and Context:The dataset was sourced from the S&P 500 index using the yfinance library. The selected period, ranging from 2000 to 2023, ensures the inclusion of diverse market conditions.

Stationarity and Segmentation: The continuous time series was segmented into rolling windows of 60 days. This structure allows the model to learn temporal dependencies and short-term volatility patterns.

Normalization: Normalization was applied to map all values into the interval [-1, 1].

Architecture

To adapt to financial time series, the architecture replaces spatial 2D layers with 1D Convolutions, enabling the network to capture temporal dependencies rather than spatial shapes.

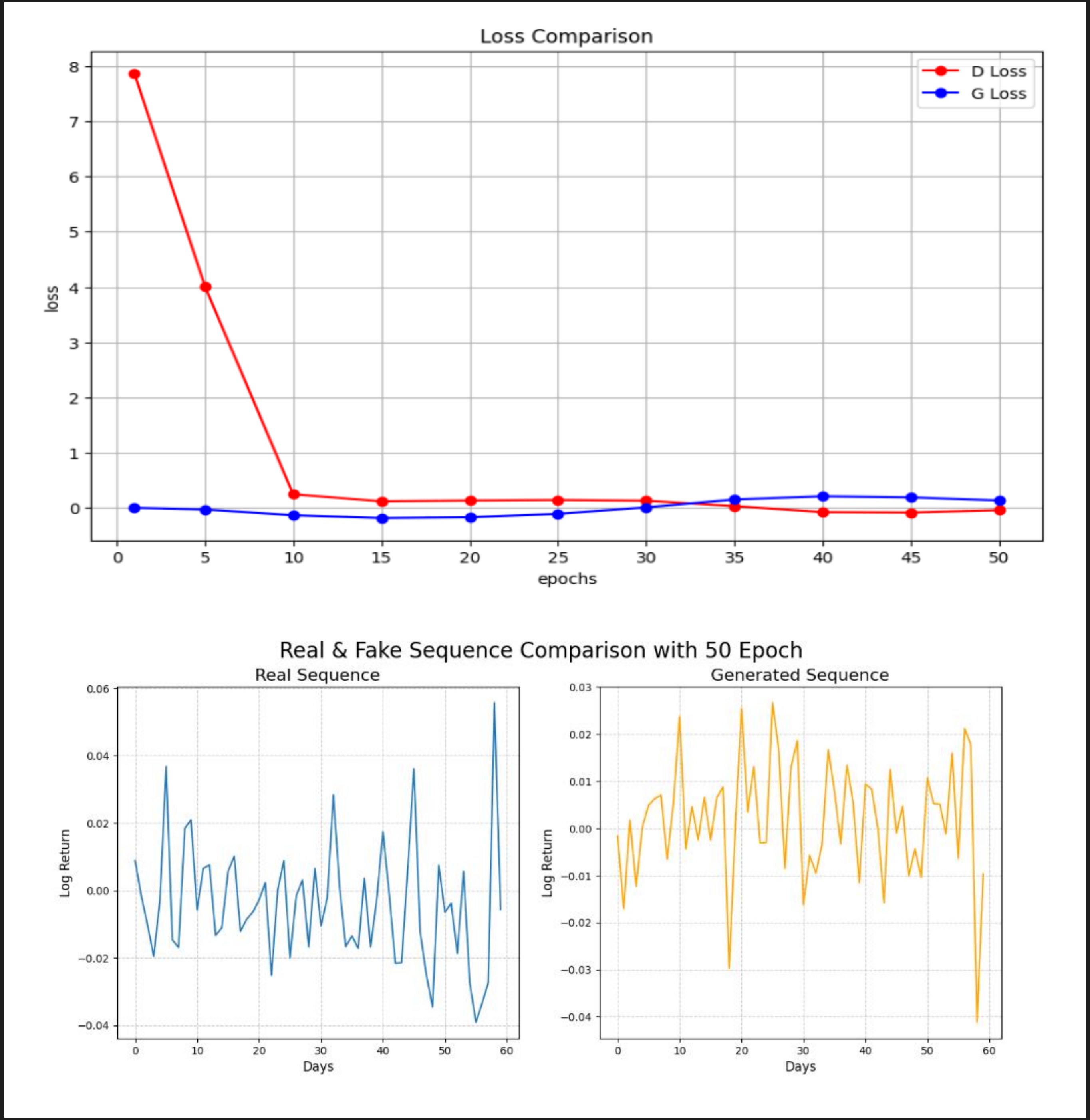


```
G=tf.keras.Sequential([
    tf.keras.layers.Dense(units=1500,input_shape=(100,)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Reshape((15,100)),
    tf.keras.layers.Conv1DTranspose(filters=256,kernel_size=3,padding="same",strides=2),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv1DTranspose(filters=1,kernel_size=3,activation="tanh",padding="same",strides=2),
])

G.summary()

D=tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=128,kernel_size=3,padding="same",strides=2,input_shape=(60,1)),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Conv1D(filters=256,kernel_size=3,padding="same",strides=2),
    tf.keras.layers.LeakyReLU(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=256,activation="relu"),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(units=1)
])

D.summary()
```

Financial Results

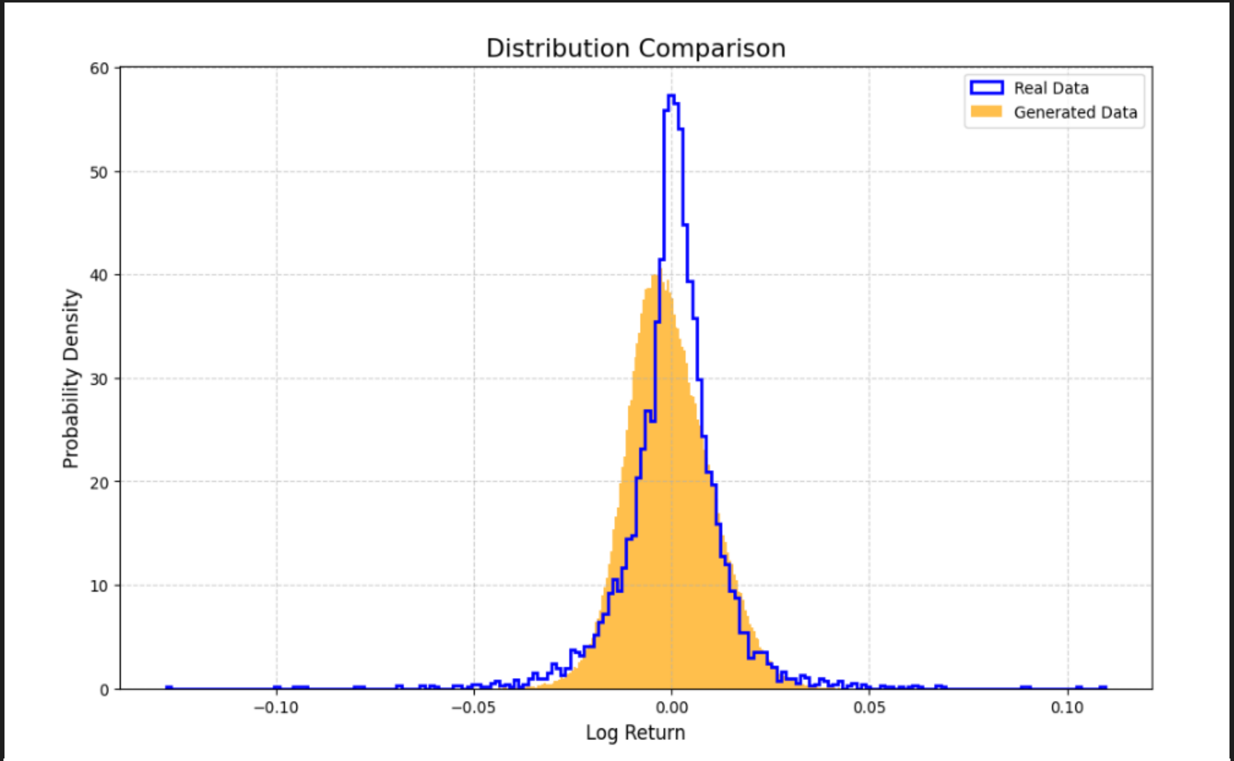
The Critic Loss convergence confirms training stability, while the generated plot visually reproduces volatility clustering. The model successfully matches Mean, Std Dev, and Kurtosis, proving it can generate realistic individual trajectories with correct local properties.

Momentums	Real	Generated
Mean	-0.004167	0.001780
Dev Std	0.017681	0.012558
Excess Kurtosis	1.495116	1.459760



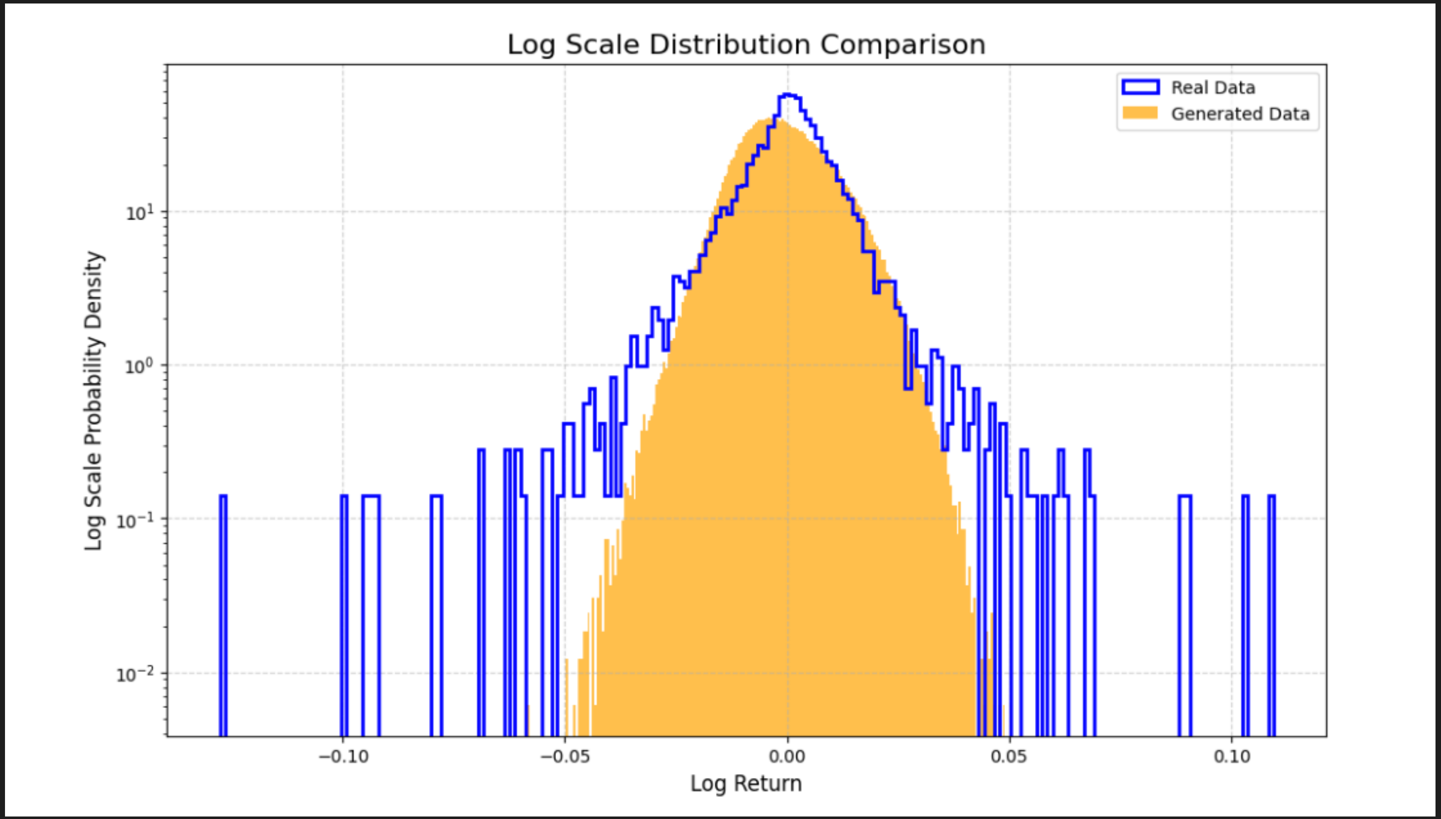
The Differences between the two distributions

The distributions overlap almost perfectly in the center. The model successfully captures the diffusive dynamics.
Mean and Std Dev are well-aligned with real market data.



Momentums	Real	Generated
Mean	0.000197	-0.000512
Std Dev	0.012379	0.010467
Excess Kurtosis	10.283440	0.234558

2nd Graph: The logarithmic view reveals a limitation. Real data exhibits heavy tails, representing "Black Swan" events. The Generated data, however, approximates a Gaussian distribution, effectively modeling the volatility but underestimating extreme market crashes.



Limitations and Future Progresses

LIMITATIONS

While the WGAN successfully learns Mean and Standard Deviation, it currently underestimates the Kurtosis. The generated distribution is closer to a Gaussian Distribution than the real Heavy-Tailed distribution.
The Generator learns most of the market behavior but struggles to capture critical events as they are rare and energetically expensive to learn.

The model relies on Log-Returns stationarity. It does not yet account for long-term structural regime changes, for instance low vs. high inflation eras.

FUTURE DEVELOPMENTS

Replace the Gaussian Noise input $z \sim \mathcal{N}(0, 1)$ with t-Student or Lévy Stable noise to inject fat tails directly into the latent manifold.

In addition, modify the Loss Function by adding a penalty term for higher-order moments mismatch