Comparative Analysis of Hyperband, Random Search, and Bayesian Optimisation Algorithms for GAN Hyperparameter Tuning on MNIST Dataset

How do Hyperband, Random Search, and Bayesian Optimisation algorithms compare in optimizing hyperparameters for Generative Adversarial Networks trained on the MNIST dataset?

A Computer Science Extended Essay

3999 words

Contents

1 Introduction

With the rapid development of Machine Learning areas like Computer Vision, creating advanced images has emerged as a critical challenge. Addressing this, the General Adversarial Network (GAN) was unveiled in 2014 as a significant breakthrough.[1] GANs excel at creating highly realistic and varied images, becoming a core solution for many image-generation tasks.

Yet, when the goal is to create efficient GANs, the need for appropriate configuration of hyperparameters arises. Hyperparameters are external parameters, which do not belong to the model and cannot be predicted exclusively from the dataset. They are adjusted through methods like expertise—leveraging knowledge from past experiences to determine their settings—and *trial and error*,[2] which involves testing various configurations to identify the most effective one. These parameters significantly influence how the network will learn and capture sophisticated patterns and textures —proper hyperparameter tuning greatly increases the probability of GAN reaching the optimal or near-optimal solution.

This necessitates the exploration of effective hyperparameter optimization techniques to enhance GAN performance, rather than relying on manual adjustments. This is particularly crucial in broad hyperparameter search

[1] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

[2] H. Alibrahim and S. A. Ludwig. "Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization". In: *2021 IEEE Congress on Evolutionary Computation (CEC)*. 2021, pp. 1551–1559.

spaces, where manual selection demands excessive time and computational resources. Some of the key techniques for hyperparameter optimization across diverse neural network frameworks are often considered to be Hyperband, Bayesian Optimization, and Random Search. Nonetheless, examining this strategy on large and resource-demanding datasets requires sufficient computational resources, therefore one convenient way to study these optimisation techniques is by utilising the MNIST dataset with its relatively small size and low resolution of images (60000 28x28 grayscale handwritten digits distributed and labelled across 10 classes). Consequently, the research question that arises is: **How do Hyperband, Random Search, and Bayesian Optimisation algorithms compare in optimizing hyperparameters for Generative Adversarial Networks trained on the MNIST dataset?**

The findings indicated that, when assessing generated images' quality and diversity, Hyperband outperformed the others, making it the best-performing algorithm, followed by Bayesian Optimization, with Random Search performing the worst. Nonetheless, Hyperband lacked flexibility when setting the necessary number of trials (that is, the number of training processes with different hyperparameters), which made it the most time- and resource-intensive algorithm in this study, indicating that in the situations where the available execution time and computational resource usage are limited, Hyperband can be excessive and inefficient. Meanwhile, Random Search and Bayesian Optimization had similar resource efficiency. Their ability to be

run for different numbers of trials for the same number of epochs allows for more flexibility in adjusting for time- and computational resource constraints. This adaptability makes Bayesian Optimization (which compared to Random Search generates images of higher quality and diversity) a preferable option for small hyperparameter spaces in case of limited computational resources. As for large hyperparameter spaces and when computing resources usage is unimportant, Hyperband has higher chances to find the optimal or near-optimal solution (generate images of higher quality and diversity), thus making it the best option in this scenario.

2 Methodology and background information

2.1 Method

This paper explores the optimization of several hyperparameters, specifically focusing on the *dropout rate*—the probability with which certain neurons in the network are omitted from calculations;[3] the *activation function*—which acts on the hidden layers' output, assisting the neural network in learning complex data features;[4] and *batch normalization*—a technique for normaliz-

[3] Gonçalo Mordido, Haojin Yang, and Christoph Meinel. *Dropout-GAN: Learning from a Dynamic Ensemble of Discriminators*. 2020.

[4] Ameya D. Jagtap and George Em Karniadakis. "How important are activation functions in regression and classification? A survey, performance comparison, and future directions". In: *Journal of Machine Learning for Modeling and Computing* 4.1 (2023), pp. 21–75.

ing the activations in the network's intermediate layers.[5] These parameters are crucial as they significantly impact the results and are compatible with the hyperparameter optimization techniques discussed in this study, including Bayesian Optimization (BO), Hyperband, and Random Search. The hyperparameters will be tuned for the most commonly used *optimizer* (a neural network function that adjusts the weights to reduce losses[6]) in recent studies, the Adam optimizer, known for its efficiency.[7] A consistent *learning rate*—the speed of model learning from existing data—of 10^{-4} will be used, and the structure of the GAN will remain unchanged across experiments with different hyperparameter optimization techniques, facilitating a more accurate comparison.

Next, BO, Hyperband, and Random Search will be employed to determine which hyperparameter optimization technique most significantly enhances the GAN's performance. The evaluation will focus on the generated images' quality and diversity, alongside the model's general efficiency, taking into account the execution time and computational resources.

To evaluate the outcomes of this study, various metrics were utilized to determine the most effective models and, hence, the optimal hyperparameter optimization algorithms for our specific context. The metrics employed in

[5]Nils Bjorck et al. "Understanding batch normalization". In: *Advances in neural information processing systems*. Vol. 31. 2018.

[6]Chitra Desai. "Comparative Analysis of Optimizers in Deep Neural Networks". In: (Oct. 2020).

[7]Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

this analysis include:

- *Inception Score* (IS)[8] and *Fréchet Inception Distance* (FID),[9] which quantitatively measure the quality and diversity of generated images
- *Visual Inspection*, to qualitatively assess image quality
- *Generator and Discriminator loss*, to monitor the model's learning process
- *Execution time*, evaluating the model's efficiency in terms of computational resources use.
- The *confusion matrix*[10] and *discriminator accuracy* highlight how well the discriminator can differentiate between genuine and artificially generated images.

The selection of these metrics is driven by the complexity of evaluating GANs, given the limited number of metrics available for such assessments. The chosen metrics are widely recognized within the GAN and broader machine learning communities,[11][12] as they enable a comprehensive evaluation of both

[8] Tim Salimans et al. "Improved techniques for training GANs". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2234–2242.

[9] Martin Heusel et al. "GANs trained by a two time-scale update rule converge to a local Nash equilibrium". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6626–6637.

[10] Stephen Stehman. "Selecting and interpreting measures of thematic classification accuracy". In: *Remote Sensing of Environment* 62 (Oct. 1997), pp. 77–89.

[11] Yaniv Benny et al. "Evaluation Metrics for Conditional Image Generation". In: *International Journal of Computer Vision* 129 (Mar. 2021), 1712–1731. ISSN: 1573-1405.

[12] Alibrahim and Ludwig, "Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization".

the discriminator and generator's performance, as well as the overall output of the model.

It is crucial to note, however, that the most commonly used metrics for evaluation of generated images, IS and FID, were initially designed for color images, which are represented with three channels (RGB), in contrast to datasets like MNIST that use only one (grayscale). This difference could potentially impact the metrics' ability to accurately measure the similarity or diversity of grayscale images. Additionally, metrics such as *execution time* can vary depending on the processor's power, making them relative rather than absolute measures. Meanwhile, *generator and discriminator losses* offer a more direct comparison of model performance, rather than an explicit measure of image quality, thus influencing the precision of our evaluation regarding the best hyperparameter optimization techniques. These considerations highlight the importance of careful judgment and the integration of additional metrics, such as the *confusion matrix* or qualitative assessments, like *visual inception*, to accurately assess GAN performance. It also emphasizes the necessity of basing judgments on a variety of assessment techniques.

2.2 Existing research

Hyperband, an advancement on Random Search, is celebrated for its efficiency in exploring hyperparameter spaces by effectively allocating resources and implementing *early stopping* (stopping to train the model for hyperpa-

rameters which are not promising), often outperforming other methods in various scenarios.[13] BO has received attention for its precision in hyperparameter tuning,[14] owing to its methodical approach to exploring the hyperparameter space, making it a valuable tool not just in GANs but in a wide array of machine learning applications.[15] Random Search, on the other hand, despite its simplicity, remains a robust choice for hyperparameter discovery, outperforming manual approaches to hyperparameter tuning in various machine learning tasks such as Neural Architecture Search,[16] especially when under tight time restrictions. Although it may not achieve the sophistication of Hyperband or BO, its effectiveness in a broad spectrum of neural network types underscores its inclusion in this comparative study. Together, these optimization techniques—each with its unique strengths—combine into a comprehensive approach to the challenge of optimising the hyperparameters in machine learning models.

What is for evaluation metrics, Inception Score (IS)[17] and Fretchet Inception distance (FID)[18] have demonstrated their effectiveness in hyperparameter

[13]Liam Li et al. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *Journal of Machine Learning Research* 18-185 (2018), pp. 1–52.

[14]Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

[15]Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems*. Vol. 25. 2012.

[16]Petro Liashchynskyi and Pavlo Liashchynskyi. *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*. 2019.

[17]Salimans et al., "Improved techniques for training GANs".

[18]Heusel et al., "GANs trained by a two time-scale update rule converge to a local Nash equilibrium".

optimization and alignment with human judgment.[19] Additionally, metrics such as loss values and confusion matrices are invaluable for classifier assessment with modifications that can become applicable to GAN evaluations of both generator and discriminator.[20]

2.3 GAN

A GAN consists of two neural networks: the generator and the discriminator, which engage in a competitive interaction. This competition results in a scenario where the gain of the discriminator corresponds to a loss for the generator, and vice versa, fostering an adversarial learning dynamic. The main goal of the generator is to create data samples that are indistinguishable from genuine ones, while the discriminator's objective is to discern accurately if a sample is from the real dataset or is generated.[21]

To describe a GAN in more formal terms, let G represent the generative model, and D the discriminative model that assesses the likelihood of an image being authentic as opposed to being produced by G. Consequently, the G's goal is increasing the probability of D's mistake. As a consequence, D incurs a penalty through the loss function, which, in turn, promotes the improvement of both D and G (Figure 1).

[19] Karol Kurach et al. "A Large-Scale Study on Regularization and Normalization in GANs". In: *International Conference on Machine Learning*. 2019.

[20] Alibrahim and Ludwig, "Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization".

[21] Goodfellow et al., "Generative Adversarial Nets".

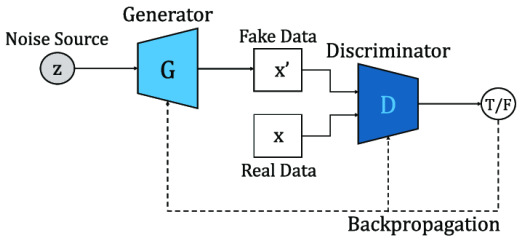


Figure 1: The structure of GAN[22]

2.4 Explanation of algorithms used

2.4.1 Hyperband

The Hyperband algorithm enhances the efficiency of *configuration evaluation* (configuration is determined by a set of hyperparameters for model training) by allocating more computational resources to the most promising configurations, rather than uniformly distributing resources across all options regardless of their performance outcomes.[23][24] This method differentiates from brute-force approaches by prioritizing configurations that show potential, thereby optimizing the use of resources and accelerating the discovery process.

[23]Li et al., "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization".

[24]Kevin Jamieson and Ameet Talwalkar. "Non-stochastic best arm identification and hyperparameter optimization". In: *International Conference on Artificial Intelligence and Statistics*. 2015.

2.4.2 Random Search

Random Search assesses the hyperparameter combinations with random selection. It applies to any case, be it discrete, continuous, or mixed space, and can be used both if the search space includes a variety of different values for different parameters and if the hyperparameter search space is relatively small. With Random Search the probability of fastly finding the optimal or near-optimal set of hyperparameters increases as it does not employ excessive iteration over all available configurations..[25]

2.4.3 Bayesian Optimization (BO)

Bayesian Optimization (BO) is a method for optimizing functions that are not directly accessible, utilizing a *probabilistic surrogate model* and an *acquisition function* to guide the search. At each step, BO constructs a surrogate model, which approximates the target function performance based on past data. Next, the acquisition function identifies the most promising parameters for further exploration, trying to determine the optimal ones. When choosing among different methods to decide where to look next, that is acquisition functions, Expected Improvement (EI) is often preferred (EI calculates the expected increase in evaluation that a new hyperparameter set provides as

[25] Liashchynskyi and Liashchynskyi, *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*.

compared to the best result found by this moment)[26][27]

2.5 Description of evaluation methods

2.5.1 Inception Score (IS)

For each subset i, the score can be expressed as:

$$IS_i = \exp\left(\frac{1}{|X_i|}\sum_{x\in X_i} D_{KL}(p(y|x)||p(y))\right)$$

where IS_i is the IS for subset i, X_i is the set of images in subset i, $p(y|x)$ represents the probability distribution of labels conditioned on the image x, while $p(y)$ denotes the overall probability distribution of labels, averaged over all images in the subset, $||$ means "or" in this notation, D_{KL} is the Kullback-Leibler divergence.[28][29]

The overall IS is then:

$$IS = \frac{1}{n}\sum_{i=1}^{n} IS_i$$

where n is the number of subsets (splits).

[26] Sebastian Ament et al. "Unexpected improvements to expected improvement for Bayesian optimization". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2024.

[27] Garnett, *Bayesian Optimization*.

[28] Salimans et al., "Improved techniques for training GANs".

[29] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79 –86.

2.5.2 Fréchet Inception Distance (FID)

FID quantifies the disparity between the feature representations of real (r) and generated (g) images as: ,

$$FID(r, g) = ||\mu_r - \mu_g||^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2})$$

Here, μ_r and μ_g symbolize the mean vectors of the real and generated images' features with $||\mu_r - \mu_g||^2$ representing a squared Euclidean distance between those vectors.[30] Σ_r and Σ_g denote the covariance matrices of the real and generated images' feature vectors, respectively. The trace operator (Tr) sums the diagonal elements of a matrix, providing a measure of total variance.[31]

2.5.3 Visual Inspection

Visual inspection serves as an intuitive evaluation metric for GAN performance by assessing if generated images resemble real ones. However, this method is subject to a significant degree of human bias.

2.5.4 Discriminator and generator loss

The efficiency of the discriminator and generator during GAN training is quantitatively assessed through their respective losses. The Binary Cross-Entropy loss function is utilized, comparing the predicted probability distri-

[30] D. Cohen. *Precalculus: A Problems-Oriented Approach*. Cengage Learning, 2004.

[31] Heusel et al., "GANs trained by a two time-scale update rule converge to a local Nash equilibrium".

bution by the model against the actual label distribution (real or generated for each image).

2.5.5 Execution time

The algorithm's efficiency can also be assessed by measuring the execution time, along with the ability to estimate this time and its variability based on different algorithmic parameters and time constraints, which could lead to a limit in total available processing power.

2.5.6 Confusion matrix and accuracy

A pre-trained MNIST classifier is used for evaluating the generative model's effectiveness in deceiving the discriminator. Lower accuracy in discriminator predictions indicates better generator performance. The confusion matrix is structured as follows:

	Predicted: Real Data	**Predicted: Generated**
Actual: Real Data	True Positive (TP)	False Negative (FN)
Actual: Generated	False Positive (FP)	True Negative (TN)

Table 1: Confusion Matrix

The accuracy of the discriminator's predictions is calculated using the for-

mula for binary classification accuracy:[32]

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

This metric assists in assessing the performance of the discriminator and generator in the optimally chosen model, also reflecting the hyperparameter optimization algorithm's effectiveness in selecting hyperparameters that contribute to an improved model.

3 Experiment

3.1 Implementation

In our study, a Google Colab environment equipped with a V100 GPU for computational support was utilized.

To implement the hyperparameter optimization strategies, the TensorFlow API, specifically Keras, utilizing the Keras Tuner tool for this purpose, was utilized (See the implementation in Appendix A.1.1)

The scope of our hyperparameter search consisted of several key parameters: the dropout rate, the activation function, and the application of batch normalization. Specifically, the dropout rate varied between 0.0 and 0.9, and the use of different activation functions including LeakyReLU, ReLU, Sigmoid,

[32] Alibrahim and Ludwig, "Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization".

and Softmax was explored, as well as whether to apply batch normalization, represented by Boolean values True or False, was considered.

In the figures presenting data from 15 epochs, the experiment was replicated 10 times (to enhance the objectivity of findings and minimize randomness) across 15 epochs for each of the 13 distinct hyperparameter combinations found by Random Search and BO. For Hyperband, the maximum number of epochs for training was also set to 15 epochs.

Regarding the figures depicting outcomes from 40 epochs, due to constraints in computational resources, these experiments were conducted only once. Both BO and Random Search were evaluated over 20 trials for a span of 40 epochs. Similarly, Hyperband was assessed with a preset maximum duration of 40 epochs.

This approach ensured that the runtime for all three algorithms was roughly equivalent, yet it still differed due to the complicated nature of Hyperband trials, where each trial has a different number of epochs to run for.

3.2 Results

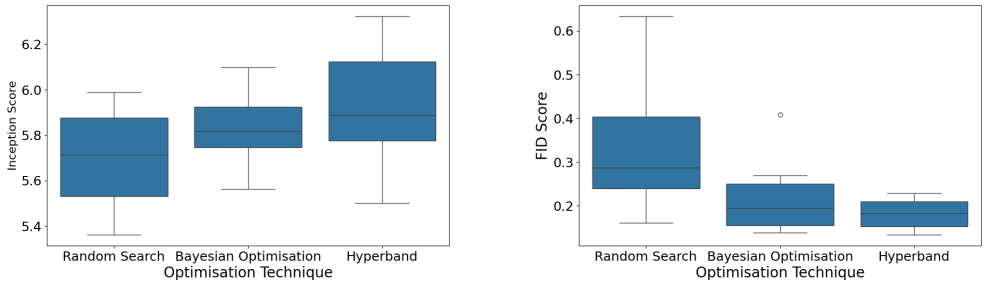
For ease of reference, this study denotes the optimal models obtained through the optimization processes of Hyperband, Random Search, and BO as M_H, M_R, and M_B, respectively.

3.2.1 Inception Score (IS) and Fréchet Inception Distance (FID)

To assess optimisation techniques' performance, this research will employ two metrics mentioned before, IS and FID. It has been determined that better results for both metrics align closely with outcomes evaluated through human observation—images with high IS and low FID are typically judged to resemble real images.[33] IS and FID focus on the direct assessment of diversity and visual quality of generated images through their quantitative comparison with ones from real datasets.[34][35]

Given that the hyperparameters selected by each of the three optimization techniques involve a degree of randomness, a higher level of result objectivity was attained by computing the best FID and IS scores obtained across 10 iterations of the model for 15 epochs (Figure 2). This methodology ensures a robust evaluation framework by mitigating the randomness in the optimization process (See the implementation of plot drawing in Appendix A.1.2, experimental data across three techniques in Appendix A.2).

[33] Kurach et al., "A Large-Scale Study on Regularization and Normalization in GANs".
[34] Salimans et al., "Improved techniques for training GANs".
[35] Heusel et al., "GANs trained by a two time-scale update rule converge to a local Nash equilibrium".



(a) IS distribution per 15 epochs (b) FID distribution per 15 epochs

Figure 2: Comparison of Inception score and FID Score distributions over 15 epochs

If we denote median IS for Random Search, BO, and Hyperband as $\widetilde{IS}_R$, $\widetilde{IS}_B$, and $\widetilde{IS}_H$, respectively, and, similarly, the median FID as $\widetilde{FID}_R$, $\widetilde{FID}_B$, and $\widetilde{FID}_H$, our analysis provides the following results from the box plots:

$$\widetilde{IS}_R < \widetilde{IS}_B < \widetilde{IS}_H$$

$$\widetilde{FID}_R > \widetilde{FID}_B > \widetilde{FID}_H$$

With $\widetilde{IS}_R \approx 5.72$, $\widetilde{IS}_B \approx 5.82$, $\widetilde{IS}_H \approx 5.89$, $\widetilde{FID}_R \approx 0.29$, $\widetilde{FID}_B \approx 0.19$, $\widetilde{FID}_H \approx 0.18$.

This indicates that, for the 15-epoch runs, Hyperband outperforms the other algorithms, with BO closely behind in both IS and FID, and Random Search exhibiting the least favourable results in these metrics.

To observe the optimisation techniques' performance with longer training, M_H, M_R, and M_B were determined after 40 epochs per trial run again using

maximisation of IS as an objective:

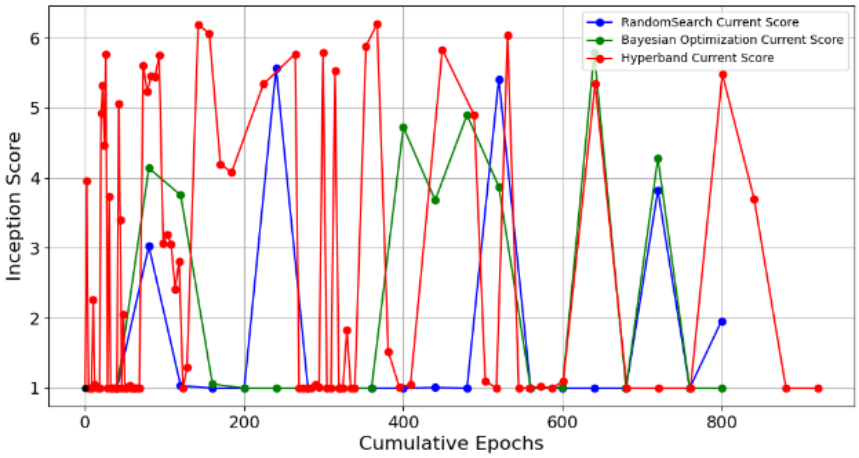


Figure 3: Evolution of Current IS per Epoch

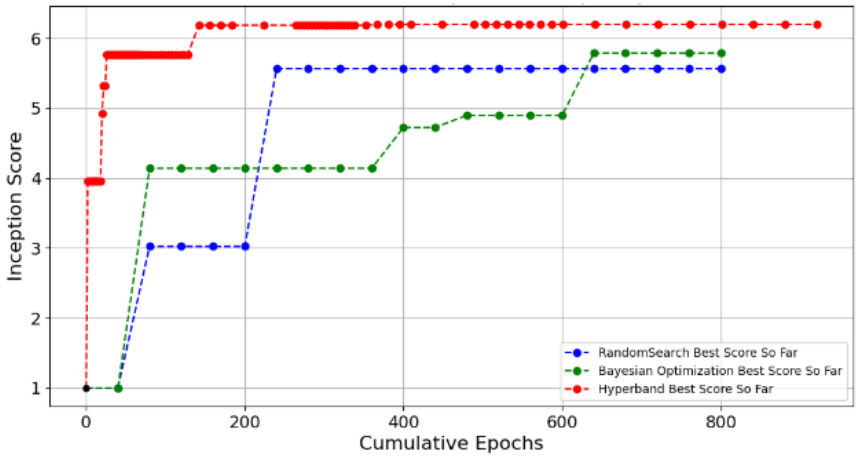


Figure 4: Evolution of Best IS per Epoch

Analysis of the evolution of current IS per epoch (Figure 3) reveals that Hyperband most frequently compared to other techniques explores models with high IS scores due to its focus on the most promising models over

a substantial number of trials. BO follows, targeting the most promising configurations with fewer trials, and Random Search trails due to its entirely random configuration selection.

Examining the evolution of the best IS per epoch (Figure 4), it can be concluded that BO shows steady improvement, which might be desirable if consistent progress over time is the goal. For the best IS achieved by the end of runs, scores for M_H are around 6.2, for M_R approximately 5.6, and for M_B about 5.8. This reflects advancement for each technique compared to the 15-epoch median scores while maintaining their relative performance order regarding the quality and diversity of generated images.

3.2.2 Visual Inspection

(a) 1 epoch (b) 15 epochs (c) 40 epochs

Figure 5: Transformation of images generated by the best model after 1, 15, and 40 epochs for Hyperband.

(a) 1 epoch (b) 15 epochs (c) 40 epochs

Figure 6: Transformation of images generated by the best model after 1, 15, and 40 epochs for Random Search.

(a) 1 epoch (b) 15 epochs (c) 40 epochs

Figure 7: Transformation of images generated by the best model after 1, 15, and 40 epochs for BO.

This visualization demonstrates that, as the number of epochs increases, the image quality improves across models optimized by each technique. However, the enhancement is most notable in models with Hyperband-optimized hyperparameters (Figure 5), where the images exhibit more defined shapes after 15 (Figure 5b) and 40 (Figure 5c) epochs of training, thereby resembling real photos more closely than ones hyperparameters of which were optimized by Random Search and BO.

In contrast, images from the generators, hyperparameters of which were found through Random Search, even after 15 and 40 epochs (Figures 6b, 6c respectively), show a tendency for backgrounds to appear blurred, leaning towards a grey hue rather than solid black. This suggests that images produced by models optimised by Random Search struggle with effectively transitioning the initial random noise to clearly distinct white digits with an intended black background, potentially contributing to its inferior performance in terms of IS and FID.

3.2.3 Generator and Discriminator losses

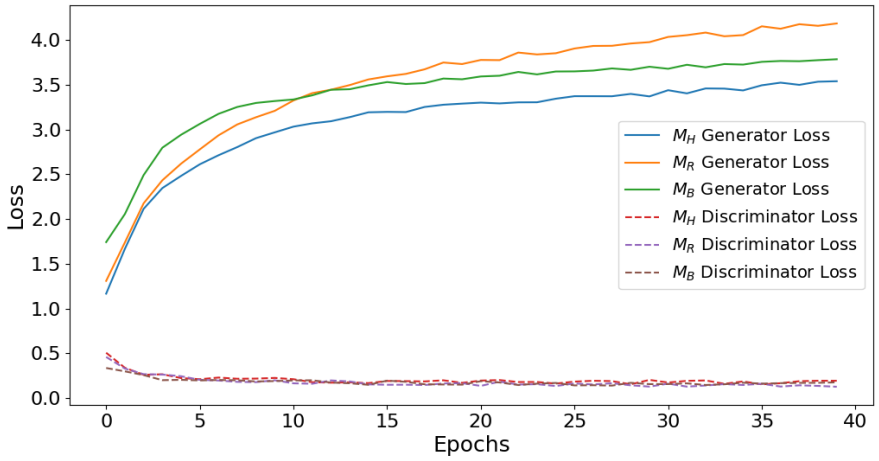


Figure 8: Discriminator and Generator losses per epoch for best models with hyperparameters generated by 3 hyperparameter optimisation techniques

Analyzing the graph data, we observe that the discriminator consistently outperforms the generator during the training process, as evidenced by the

generator's loss increasing while the discriminator's loss decreases. Notably, after a substantial period of training—specifically, 10 epochs the comparative relationship between losses remains the same—the generator of M_H has the lowest loss, followed by M_B's generator, with M_R's generator showing the highest loss.

This pattern indicates that M_H's generator is most effective at fooling its discriminator as it has the lowest loss, positioning it as the best model in this regard.[36] M_B's generator ranks second in this capability, while M_R's generator is the least effective. Despite this, the discrepancy in loss between the best (M_H) and the worst (M_R) models is relatively small, approximately 25%.

However, one limitation of using the generator loss as a metric is that the generators are assessed against their corresponding discriminators within the same model configuration, meaning that the discriminators of M_H, M_R, and M_B differ. Therefore, the generators' losses are influenced by the capabilities of these discriminators. Consequently, the variations in generator loss across different models can only be compared in terms of relative performance to their models.

[36] Goodfellow et al., "Generative Adversarial Nets".

3.2.4 Execution time

Execution time	Hyperband	Random Search	BO
15 epochs	33 min	29 min	30 min
40 epochs	130 min	121 min	123 min

Table 2: Execution times for different optimization methods over varying epochs

In the 15-epoch comparison, Random Search and BO each conducted 13 trials, with each trial testing a distinct set of hyperparameters determined by the respective algorithm. Hyperband, in contrast, executed 30 trials, varying the number of epochs per trial. The selection of trial numbers for Random Search and BO was based on trial and error to ensure their total execution time was roughly equivalent to that of Hyperband. Similarly, for the 40-epoch analysis, BO and Random Search were run for 20 trials to match Hyperband's execution duration (Table 2).

However, Hyperband's approach, involving a greater number of trials with variable epoch lengths and the application of early stopping, introduces complexity in predicting its total run time. This unpredictability could pose a disadvantage in scenarios with limited computing resources or strict time constraints, potentially making Random Search and BO more suitable choices.

Furthermore, in environments with smaller hyperparameter spaces, or where there should be a trade-off between computational time and quality final results, Random Search and BO might be more efficient, as these algorithms

allow for a predetermined number of trials, avoiding the need for the excessive experimentation that Hyperband's method entails.

3.2.5 Confusion matrix and discriminator accuracy

Each hyperparameter optimisation algorithm identified the model M with certain hyperparameters, which generates the best images (by the IS of these images) across the models optimized by this algorithm. This model consisted of the generator G and discriminator D. As mentioned previously, these models are M_H, M_R, and M_B. As such, to implement a confusion matrix for binary classification of generated images (real or generated). G of each technique generated 1500 images, which were further split into three batches of 500 images, where images from each batch were classified as real or fake by D of each technique to avoid possible bias—since a discriminator is presumably more effective at classifying images from its generator. Additionally, each D evaluated 1500 real images. The outcomes are as follows:

	Predicted: Real Data	**Predicted: Generated**
Actual: Real Data	1423	77
Actual: Generated	324	1176

Table 3: Confusion matrix for the best model found by Hyperband

	Predicted: Real Data	**Predicted: Generated**
Actual: Real Data	1395	105
Actual: Generated	102	1398

Table 4: Confusion matrix the best model found by Random Search

	Predicted: Real Data	**Predicted: Generated**
Actual: Real Data	1411	89
Actual: Generated	213	1287

Table 5: Confusion matrix for the best model found by Bayesian Optimisation

Based on this data (Tables 3, 4, 5), discriminator accuracy for each optimization technique is summarized as follows:

Optimization Technique	Accuracy
Hyperband	0.866
Random Search	0.921
BO	0.899

Table 6: Accuracy of Different Optimization Techniques

These accuracy levels, distant from the ideal (≈ 0.5) (Figure 11), suggest the generators are outperformed by their discriminators, a conclusion supported by earlier loss graph analysis. The discriminator accuracies (A_H, A_R, and A_B) for M_H, M_R, and M_B, respectively, are ranked as follows (Figure 11):

$$A_H < A_B < A_R$$

indicating that M_H's generator was the best at deceiving its discriminator, followed by M_B, with M_R's generator being the least effective.

While it could be argued that overall accuracy is influenced by using different

discriminators for real images, thus possibly affecting the metric's relevance, recalculating accuracy based solely on generated images:

$$Accuracy = \frac{TN}{FP + TN}$$

provides accuracies for generated images (A'_H, A'_R, and A'_B) as 0.784, 0.932, and 0.858 for discriminators of M_H, M_R, and M_B, respectively (calculated from Tables 3, 4, 5, respectively).

4 Methodology evaluation

4.1 Advantages of the study

The use of multiple metrics ensured that our findings were robust, minimizing the impact of uncertainties, such as the random selection of initial hyperparameters. Ultimately, our method effectively addressed the research question, demonstrating that Hyperband is usually the best method of hyperparameter optimisation for GANs trained on the MNIST dataset, while also noting that in situations with smaller hyperparameter spaces where speed and flexibility are prioritized, BO and Random Search could be more advantageous.

4.2 Limitations of the study

Nevertheless, there were several limitations to our study, one of them being the restricted range of metrics available for GAN evaluation. Moreover, the

use of the grayscale MNIST dataset limited the effectiveness of certain evaluation techniques like IS and FID, as their performance might have been inaccurate, thus ineffectively addressing the research question. Additionally, limited computational resources constrained our exploration of a broader hyperparameter space, for which the results might have been different.

4.3 Improvements to the method

If this study was conducted again, it would be beneficial to explore a wider array of datasets (adding CIFAR-10 or CelebA, for instance) and utilize more powerful computational resources. This would allow for a deeper investigation into a broader range of hyperparameters and a comparison of optimization techniques across more varied conditions, leading to more generalizable conclusions. Furthermore, experimenting with different network architectures and adjusting the balance between generator and discriminator performance could potentially improve the training process, leading to more effective model optimization.

5 Conclusion

To address the research question, "How do Hyperband, Random Search, and Bayesian Optimization algorithms compare in optimizing hyperparameters for GANs trained on the MNIST dataset?", a detailed experimental analysis was carried out. This analysis examined the models generated by these

techniques using a range of metrics: Inception Score (IS), Fréchet Inception Distance (FID), visual inspection, losses of discriminator and generator, execution time, confusion matrix, and discriminator accuracy. This holistic approach allowed for a thorough evaluation of the quality and diversity of generated images, both qualitatively and quantitatively. The findings indicate that while Hyperband produces higher quality and more diverse images than BO, with Random Search being the least effective, BO and Random Search offer advantages in terms of computational efficiency and adaptability to time constraints, making them a better choice in case of small hyperparameter spaces' explorations with limited computational resource. These observations are consistent with prior research on hyperparameter optimization for deep learning,[37] where Hyperband outperformed BO, and on Neural Architecture Search,[38] which depicted BO's advantage over Random Search, yet consider the possibility of limited resources available.

The study effectively answered the research question by employing varied methods for GAN hyperparameter optimization. However, the scope was somewhat limited by available computational resources. Expanding the investigation to cover broader hyperparameter spaces, and different conditions, such as the number of trials, epochs and the datasets with colorful images could make the research more effective in answering the research question.

[37] Li et al., "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization".

[38] Liashchynskyi and Liashchynskyi, *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*.

The evaluation methodology, which incorporated multiple metrics to evaluate model performance and the quality of generated images, proved valuable. It offered a holistic perspective, with results combined from different metrics, enhancing the degree of certainty and confidence in the findings. Yet, it is important to recognize that metrics such as IS and FID were primarily designed for evaluating colorful images. This characteristic limited their reliability when applied to the grayscale MNIST dataset in our study. Additionally, the initial randomness in the selection of hyperparameters could have significantly influenced the outcomes, especially in runs spanning 40 epochs, where, due to computational resource constraints, hyperparameter optimization for each technique was conducted only a single time.

References

Alibrahim, H. and S. A. Ludwig. "Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization". In: *2021 IEEE Congress on Evolutionary Computation (CEC)*. 2021, pp. 1551–1559.

Ament, Sebastian et al. "Unexpected improvements to expected improvement for Bayesian optimization". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2024.

Benny, Yaniv et al. "Evaluation Metrics for Conditional Image Generation". In: *International Journal of Computer Vision* 129 (Mar. 2021), 1712–1731. ISSN: 1573-1405.

Bjorck, Nils et al. "Understanding batch normalization". In: *Advances in neural information processing systems*. Vol. 31. 2018.

Cohen, D. *Precalculus: A Problems-Oriented Approach*. Cengage Learning, 2004.

Desai, Chitra. "Comparative Analysis of Optimizers in Deep Neural Networks". In: (Oct. 2020).

Gabsi et al. *GAN Hyperparameter Tuning with Keras Tuner*. Medium. 2022.

Garnett, Roman. *Bayesian Optimization*. Cambridge University Press, 2023.

Goodfellow, Ian J. et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

Heusel, Martin et al. "GANs trained by a two time-scale update rule converge to a local Nash equilibrium". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6626–6637.

Jagtap, Ameya D. and George Em Karniadakis. "How important are activation functions in regression and classification? A survey, performance comparison, and future directions". In: *Journal of Machine Learning for Modeling and Computing* 4.1 (2023), pp. 21–75.

Jamieson, Kevin and Ameet Talwalkar. "Non-stochastic best arm identification and hyperparameter optimization". In: *International Conference on Artificial Intelligence and Statistics*. 2015.

Kingma, Diederik P and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

Kullback, S. and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79 –86.

Kurach, Karol et al. "A Large-Scale Study on Regularization and Normalization in GANs". In: *International Conference on Machine Learning*. 2019.

Li, Liam et al. "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". In: *Journal of Machine Learning Research* 18-185 (2018), pp. 1–52.

Liashchynskyi, Petro and Pavlo Liashchynskyi. *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*. 2019.

Mordido, Gonçalo, Haojin Yang, and Christoph Meinel. *Dropout-GAN: Learning from a Dynamic Ensemble of Discriminators*. 2020.

Salimans, Tim et al. "Improved techniques for training GANs". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2234–2242.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems*. Vol. 25. 2012.

Stehman, Stephen. "Selecting and interpreting measures of thematic classification accuracy". In: *Remote Sensing of Environment* 62 (Oct. 1997), pp. 77–89.

A Appendix

A.1 Code

A.1.1 Main code of the program

The model was created based on the GAN architecture proposed in Medium article.[39]

```
# -*- coding: utf-8 -*-
"""Extended Essay Code"""

"""
Counting Epochs for Hyperband
"""

epoch_counts_per_trial_RandomSearch = {}

epoch_counts_per_trial_BO = {}

epoch_counts_per_trial_Hyperband = {}

"""Import OS"""

import os

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

"""Loss Logger"""

import tensorflow as tf

class GANDashboard(tf.keras.callbacks.Callback):

    def __init__(self):
        super(GANDashboard, self).__init__()
        self.gen_losses = []
        self.disc_losses = []
```

[39] Gabsi et al. *GAN Hyperparameter Tuning with Keras Tuner*. Medium. 2022.

```
    def on_epoch_end(self, epoch, logs=None):
        logs = logs or {}
        gen_loss = logs.get('g_loss')
        disc_loss = logs.get('d_loss')
        if gen_loss is not None and disc_loss is not None:
          self.gen_losses.append(min(max(gen_loss, 0), 5))
          self.disc_losses.append(min(max(disc_loss, 0), 5))

    def plot_losses(self, type_of_losses, num_of_epochs=None, num_of_trials=None):
        if type_of_losses == "epochs":
            plt.figure(figsize=(10, 5))
            plt.plot(self.gen_losses, label='Generator Loss')
            plt.plot(self.disc_losses, label='Discriminator Loss')
            plt.title('GAN Losses Over Epochs')
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.legend()
            plt.savefig("/content/drive/My Drive/Combined_tuner/GAN_losses_over_epochs.png")
            plt.show()
        elif type_of_losses == "trials":
            # Validate num_of_epochs
            if num_of_epochs is None or num_of_epochs <= 0:
                print("Please provide a valid number of epochs per trial.")
                return

            # Select losses for plotting: every [num_of_epochs] starting from [num_of_epochs-1]
            indices = range(num_of_epochs - 1, len(self.gen_losses), len(self.gen_losses))
            selected_gen_losses = [self.gen_losses[i] for i in indices]
            selected_disc_losses = [self.disc_losses[i] for i in indices]
            print(selected_gen_losses)
            print(selected_disc_losses)

            # Plotting
            plt.figure(figsize=(10, 5))
            plt.plot(selected_gen_losses, label='Generator Loss')
            plt.plot(selected_disc_losses, label='Discriminator Loss')
            plt.title('GAN Losses Over Trials')
            plt.xlabel('Trials')
            plt.ylabel('Loss')
            plt.legend()
            plt.savefig("/content/drive/My Drive/Combined_tuner/GAN_losses_over_trials.png")
            plt.show()

"""Epoch Logger"""

class EpochsLogger(tf.keras.callbacks.Callback):
    def __init__(self, trial_epoch_dict):
```

```
        super(EpochsLogger, self).__init__()
        self.epoch_count = 0
        self.trial_epoch_dict = trial_epoch_dict  # Reference to the global dictionary

    def on_epoch_begin(self, epoch, logs=None):
        self.epoch_count += 1

    def on_trial_end(self, trial, logs=None):
        # Here, we update the global dictionary
        self.trial_epoch_dict[trial.trial_id] = self.epoch_count
        print(f"Trial {trial.trial_id} completed with {self.epoch_count} epochs.")
        self.epoch_count = 0  # Reset the count for the next trial

"""Default GAN structure"""

"""
The structure of the model as well as the implementation of hyperparameter
techniques is adapted from the "GAN Hyperparameter Tuning with Keras Tuner"
Medium article by Alya Gabsi, Ali ALSAMURAEE, Crisitinaesposito, Arpitnagpal
Link: https://medium.com/@siraj.hatoum/gan-hyperparameter-tuning-with-keras-tuner-81e00ad1d6be
"""
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import keras_tuner as kt
import numpy as np

class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

    @property
    def metrics(self):
        return [self.gen_loss_tracker, self.disc_loss_tracker] #maybe introduce fid tracker

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, real_images):
```

```
        batch_size = tf.shape(real_images)[0]

        # Sample random points in the latent space
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        # Decoding them to fake images
        generated_images = self.generator(random_latent_vectors)
        # Combineing them with real images
        combined_images = tf.concat([generated_images, real_images], axis=0)

        # Assembling labels discriminating real from fake images
        labels = tf.concat(
            [tf.ones((batch_size, 1)), tf.zeros((real_images.shape[0], 1))], axis=0
        )
        # Adding random noise to the labels - important trick!
        labels += 0.05 * tf.random.uniform(labels.shape)

        # Training the discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_weights))

        # Sample random points in the latent space
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        # Assemble labels that say "all real images"
        misleading_labels = tf.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        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(grads, self.generator.trainable_weights))

    # print(self.gen_loss_tracker.result())
    # print(self.disc_loss_tracker.result())

        # Monitor loss.
        # print(self.gen_loss_tracker.update_state(g_loss))
        # print(self.disc_loss_tracker.update_state(d_loss))
        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)

        return {
```

```
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }

import tensorflow as tf
from tensorflow import keras
import numpy as np
from tensorflow.keras import layers

"""MNIST Classifier"""

# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Scaling images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Making sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train␣shape:", x_train.shape)
print(x_train.shape[0], "train␣samples")
print(x_test.shape[0], "test␣samples")


# converting class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

batch_size = 128
```

```
epochs = 15

model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)

from numpy import log, mean, std, floor, exp, expand_dims

"""Hypermodel"""

class HyperGAN(kt.HyperModel):

  def __init__(self, class_model, gan_dashboard_cb=None, epochs_logger=None):
        super(HyperGAN, self).__init__()
        self.class_model = class_model
        self.gan_dashboard_cb = gan_dashboard_cb
        self.epochs_logger = epochs_logger

      # Building discriminator function
  def make_discriminator_model(self, hp_drop_rate):
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                  input_shape=[28,28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(hp_drop_rate))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(hp_drop_rate))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model

    # Build generator
  def make_generator_model(self, latent_dim, batch_norm, activation_function):
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=[latent_dim]))
    if batch_norm == True:
      model.add(layers.BatchNormalization())
    model.add(activation_function)
    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) #None corresponds to the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same'))
    assert model.output_shape == (None, 7, 7, 128)
    if batch_norm == True:
```

```
    model.add(layers.BatchNormalization())

  model.add(activation_function)

  model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same'))
  assert model.output_shape == (None, 14, 14, 64)
  if batch_norm == True:
    model.add(layers.BatchNormalization())

  model.add(activation_function)

  model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',  activation='tanh'))
  assert model.output_shape == (None, 28, 28, 1)

  return model


def build(self, hp):

  #hp_latent_dim = hp.Int('latent_dim', min_value = 10, max_value = 500, step = 32)
  hp_drop_rate = hp.Float('dropout rate', min_value = 0, max_value = 0.9)
  batch_norm = hp.Boolean('batch normalization')
  activation_function = hp.Choice('activation function', ['relu', 'leaky_relu', 'sigmoid', 'softmax'])
  hp_latent_dim = 100

  activation_dict = {
      'leaky_relu': layers.LeakyReLU(),
      'relu': layers.ReLU(),
      'sigmoid': layers.Activation('sigmoid'),
      'softmax': layers.Activation('softmax'),
  }

  self.discriminator = self.make_discriminator_model(hp_drop_rate)
  self.generator = self.make_generator_model(hp_latent_dim, batch_norm, activation_dict[activation_function])


  model_gan = GAN(self.discriminator, self.generator, hp_latent_dim)

  adam_optimizer = tf.keras.optimizers.legacy.Adam(1e-4)
  binary_crossentropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
  model_gan.compile(adam_optimizer,adam_optimizer,binary_crossentropy)
  return model_gan


# assumes images have the shape 299x299x3, pixels in [0,255]
def calculate_score(self, model, images, n_split=10, eps=10**-16):
  # loading classification model
```

```
    model = self.class_model
    model = model
    # converting from uint8 to float32
    processed = images.astype("float32")
    # pre-processing raw images for inception v3 model
    # Scaling images to the [0, 1] range
    images = images.astype("float32") / 255
    # Mak ingsure images have shape (28, 28, 1)
    images = np.expand_dims(images, -1)
    # predicting class probabilities for images
    yhat = model.predict(processed)
    # enumerating splits of images/predictions
    scores = list()
    n_part = floor(images.shape[0] / n_split)

    for i in range(n_split):
      # retrieve p(y|x)
      ix_start, ix_end = int(i * n_part), int(i * n_part + n_part)
      p_yx = yhat[ix_start:ix_end]
      # calculate p(y)
      p_y = expand_dims(p_yx.mean(axis=0), 0)
      # calculate KL divergence using log probabilities
      kl_d = p_yx * (log(p_yx + eps) - log(p_y + eps))
      # sum over classes
      sum_kl_d = kl_d.sum(axis=1)
      # average over images
      avg_kl_d = mean(sum_kl_d)
      # undo the log
      is_score = exp(avg_kl_d)
      # store
      scores.append(is_score)

    # average across images
    is_avg, is_std = mean(scores), std(scores)

    return is_avg, is_std



def fit(self, hp, model, x, **kwargs):
    # Using the gan_dashboard_cb attribute
    callbacks = kwargs.get('callbacks', [])
    if self.gan_dashboard_cb:
        callbacks.append(self.gan_dashboard_cb)
    kwargs['callbacks'] = callbacks

    # Proceed with training
```

```
        self.history = model.fit(x, **kwargs)
        #print(self.history)

        random_latent_vectors = tf.random.normal(shape=(500, model.latent_dim))
        generated_images = model.generator.predict(random_latent_vectors)
        mean_score, std_score = self.calculate_score(self.class_model, generated_images)

        return -mean_score
"""Classes to store losses"""

gan_dashboard_cb_RandomSearch = GANDashboard() #GANDashboard for Random Search

gan_dashboard_cb_BO = GANDashboard() # GANDashboard for Bayesian Optimisation

gan_dashboard_cb_Hyperband = GANDashboard() # GANDashboard for Hyperband

"""Classes to store epochs"""

epochs_logger_RandomSearch = EpochsLogger(epoch_counts_per_trial_RandomSearch)

epochs_logger_BO = EpochsLogger(epoch_counts_per_trial_BO)

epochs_logger_Hyperband = EpochsLogger(epoch_counts_per_trial_Hyperband)

"""Calculation for the 1st tuner (RandomSearch)"""

import time

start_time = time.time()


trials = 20
epochs = 40

tuner_RandomSearch = kt.RandomSearch(
    hypermodel=HyperGAN(model, gan_dashboard_cb_RandomSearch, epochs_logger_RandomSearch),
    max_trials=trials,
    overwrite=True,
    directory="my_dir",
    project_name="custom_eval_RandomSearch",
)

tuner_RandomSearch.search(
    x = x_train, epochs=40 #should be 40 epochs
    )

end_time = time.time()
```

```
# Calculating the total time taken
total_time_taken = end_time - start_time

# Converting the time to a readable format
hours, rem = divmod(total_time_taken, 3600)
minutes, seconds = divmod(rem, 60)
print("Total Time Taken: {:0>2}:{:0>2}:{:05.2f}".format(int(hours),int(minutes),seconds))

tuner_RandomSearch.results_summary()

"""Calculation for the 2nd tuner (Bayesian optimisation)"""

import time

start_time = time.time()


trials = 20
epochs = 40

tuner_BO = kt.BayesianOptimization(
    hypermodel=HyperGAN(model, gan_dashboard_cb_BO, epochs_logger_BO),
    overwrite=True,
    max_trials=trials,
    directory="my_dir",
    project_name="custom_eval_bo",
)

tuner_BO.search(
    x = x_train, epochs=40
    )

end_time = time.time()
# Calculating the total time taken
total_time_taken = end_time - start_time

# Converting the time to a readable format
hours, rem = divmod(total_time_taken, 3600)
minutes, seconds = divmod(rem, 60)
print("Total Time Taken: {:0>2}:{:0>2}:{:05.2f}".format(int(hours),int(minutes),seconds))

tuner_BO.results_summary()

"""Calculation for the 3rd tuner (Hyperband)"""

import time
```

```
start_time = time.time()


trials = 20
epochs = 40

tuner_Hyperband = kt.Hyperband(
    hypermodel=HyperGAN(model, gan_dashboard_cb_Hyperband, epochs_logger_Hyperband),
    max_epochs=40,
    factor=3,
    overwrite=True,
    directory="my_dir",
    project_name="custom_eval_hyperband",
)

tuner_Hyperband.search(
    x = x_train, epochs=40
    )

end_time = time.time()
# Calculating the total time taken
total_time_taken = end_time - start_time

# Converting the time to a readable format
hours, rem = divmod(total_time_taken, 3600)
minutes, seconds = divmod(rem, 60)
print("Total Time Taken: {:0>2}:{:0>2}:{:05.2f}".format(int(hours),int(minutes),seconds))

tuner_Hyperband.results_summary()

"""Save tuner models"""

import shutil
import os

source_path = '/content/my_dir/custom_eval_hyperband'
destination_path = "/content/drive/MyDrive/CombinedTuner/Hyperband/custom_eval_hyperband"

if os.path.exists(destination_path):
    shutil.rmtree(destination_path)

# Copying the entire directory of tuner results
shutil.copytree(source_path, destination_path)

source_path = '/content/my_dir/custom_eval_bo'
destination_path = "/content/drive/MyDrive/CombinedTuner/BO/custom_eval_bo"
```

```
if os.path.exists(destination_path):
    shutil.rmtree(destination_path)

# Copying the entire directory of tuner results
shutil.copytree(source_path, destination_path)

source_path = '/content/my_dir/custom_eval_RandomSearch'
destination_path = "/content/drive/MyDrive/CombinedTuner/RandomSearch/custom_eval_RandomSearch"

if os.path.exists(destination_path):
    shutil.rmtree(destination_path)

# Copying the entire directory of tuner results
shutil.copytree(source_path, destination_path)

"""Calculating Hyperband distribution of epochs per trial"""

trials = tuner_Hyperband.oracle.trials

hyperband_epochs_distribution = [] #distribution of epochs to further calculate inception score over epochs

# Iterate over the trial objects correctly
for trial_id, trial in trials.items():
    # print(trial_id)  # Now 'trial_id' is the string ID directly
    # Extract the number of epochs this trial was run for
    epochs = trial.hyperparameters.get('tuner/epochs')
    hyperband_epochs_distribution.append(epochs)
    # print(f"Trial {trial_id} ran for {epochs} epochs.")

#print(hyperband_epochs_distribution)

"""Plotting and showing images function"""

import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

# function to plot the images
def plot_multiple_images(images, search_type, n_cols=None):
    n_cols = n_cols or len(images)
    n_rows = (len(images) - 1) // n_cols + 1
    if images.shape[-1] == 1:
        images = np.squeeze(images, axis=-1)
    plt.figure(figsize=(n_cols, n_rows))
    for index, image in enumerate(images):
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(image*127.5+127.5, cmap="gray")
        plt.axis("off")
```

```
        if search_type == 'RandomSearch':
            plt.savefig('/content/drive/My␣Drive/CombinedTuner/RandomSearch/image_at_epoch_{:04d}.png'.format(epochs))
        elif search_type == 'BO':
            plt.savefig('/content/drive/My␣Drive/CombinedTuner/BO/image_at_epoch_{:04d}.png'.format(epochs))
        elif search_type == 'Hyperband':
             plt.savefig('/content/drive/My␣Drive/CombinedTuner/Hyperband/image_at_epoch_{:04d}.png'.format(epochs))



def tuner_show_images(tuner, search_type):

    best_model = tuner.get_best_models()[0]
    if search_type == 'RandomSearch':
        model.save('/content/drive/My␣Drive/CombinedTuner/RandomSearch/Random_Search_model.h5')
    elif search_type == 'BO':
      model.save('/content/drive/My␣Drive/CombinedTuner/BO/BO_model.h5')
    elif search_type == 'Hyperband':
      model.save('/content/drive/My␣Drive/CombinedTuner/Hyperband/Hyperband_model.h5')

    random_latent_vectors = tf.random.normal(shape=(20, best_model.latent_dim))

    generated_images = best_model.generator.predict(random_latent_vectors)
    figure(figsize = (10, 6), dpi = 80)
    plot_multiple_images(generated_images,search_type, 5)
    plt.show()

"""Show images for Random Search"""

tuner_show_images(tuner_RandomSearch, 'RandomSearch') #show images for RandomSearch

"""Show images for Bayesian Optimisation"""

tuner_show_images(tuner_BO, 'BO') #show images for Bayesian Optimisation

"""Show images for Hyperband"""

tuner_show_images(tuner_Hyperband, 'Hyperband') # show images for Hyperband

"""Plotting Inception Score separately for each technique"""

import matplotlib.pyplot as plt

def plot_inception_score(tuner, save_directory):

    list_of_trials = tuner.oracle.get_best_trials(len(tuner.oracle.trials))

    # Extracting trial scores and sorting them in ascending order since it's a minimization problem
```

```
    trial_scores = [-trial.score for trial in list_of_trials]
    trial_ids = [trial.trial_id for trial in list_of_trials]

    correct_order_of_trial_ids = []
    for i in range(len(tuner.oracle.trials)):
        if i >= 0 and i <= 9:
            correct_order_of_trial_ids.append(f'{0}{i}')
        else:
            correct_order_of_trial_ids.append(i)

    # Converting trial IDs to integers for correct comparison and indexing
    trial_ids_int = [int(id) for id in trial_ids]
    sorted_indices = sorted(range(len(trial_ids_int)), key=lambda k: trial_ids_int[k])
    sorted_scores = [trial_scores[i] for i in sorted_indices]

    # Calculating  best scores so far in the order of trials executed
    best_scores_so_far_original_order = [max(sorted_scores[:i+1]) for i in range(len(sorted_scores))]

    plt.figure(figsize=(10, 6))
    # Plotting original scores in their execution order with markers
    plt.plot(correct_order_of_trial_ids, [trial_scores[i] for i in sorted_indices], marker='o', linestyle='-',
             color='r')
    # PLotting best scores evolution in original execution order
    plt.plot(correct_order_of_trial_ids, best_scores_so_far_original_order, marker='o', linestyle='--', color='g')
    plt.title('Inception␣Score␣Evolution', fontsize=18)
    plt.xlabel('Trial␣ID', fontsize=16)
    plt.ylabel('Inception␣Score', fontsize=16)
    plt.xticks(rotation=45, fontsize=14)
    plt.yticks(fontsize=14)
    plt.legend(['Inception␣Score␣per␣Trial', 'Best␣Score␣So␣Far'])
    plt.grid(True)
    plt.savefig(save_directory)
    plt.show()

"""Plot Inception Score for Random Search"""

plot_inception_score(tuner_RandomSearch,
                     "/content/drive/My␣Drive/CombinedTuner/RandomSearch/IndividualInceptionScore.png")

"""Plot Inception Score for Bayesian Optimisation"""

plot_inception_score(tuner_BO, "/content/drive/My␣Drive/CombinedTuner/BO/IndividualInceptionScore.png")

"""Plot Inception score for Hyperband"""

plot_inception_score(tuner_Hyperband, "/content/drive/My␣Drive/CombinedTuner/Hyperband/IndividualInceptionScore.png")
```

```
"""Plotting the Inception Scores together"""

import matplotlib.pyplot as plt
import numpy as np

print(epochs)

def plot_scores_with_lines_from_start_point(tuners, tuner_names, epochs, hyperband_epochs_distribution):
    starting_point = (0, 1)  # Define the starting point

    # Prepareing cumulative epochs for Bayesian Optimization and Random Search
    cumulative_epochs_rs_bo = np.arange(1, len(tuners[0].oracle.trials) + 1) * epochs

    colors = ['blue', 'green', 'red']  # Defining colors for each technique for consistency

    # Plot for Current Scores with lines from starting point
    plt.figure(figsize=(12, 6))
    for i, (tuner, name, epochs_distribution) in enumerate(zip(tuners,
            tuner_names, [cumulative_epochs_rs_bo, cumulative_epochs_rs_bo, hyperband_epochs_distribution])):
        trials = list(tuner.oracle.trials.values())
        trial_scores = [-trial.score for trial in trials]  # Assuming minimization objective

        if name == 'Hyperband':
            cumulative_epochs = np.cumsum(epochs_distribution)
        else:
            cumulative_epochs = epochs_distribution

        # Plotting the line from the starting point to the first data point using the same color
        if len(cumulative_epochs) > 0:
            plt.plot([starting_point[0], cumulative_epochs[0]],
                     [starting_point[1], trial_scores[0]], linestyle='-', color=colors[i])

        plt.plot(cumulative_epochs, trial_scores, marker='o', linestyle='-', color=colors[i],
                 label=f'{name} Current Score')

    plt.plot(starting_point[0], starting_point[1], marker='o', markersize=5, color='black')
    plt.title('Evolution of Current Inception Score per Epoch', fontsize=18)
    plt.xlabel('Cumulative Epochs', fontsize=16)
    plt.ylabel('Inception Score', fontsize=16)
    plt.xticks(fontsize=14)
    plt.yticks(fontsize=14)
    plt.legend()
    plt.grid(True)
    plt.savefig("/content/drive/MyDrive/CombinedTuner/Current_Inception_Score_Evolution.png")
    plt.show()
```

```
    # Ploting for Best Scores So Far with lines from starting point
    plt.figure(figsize=(12, 6))
    for i, (tuner, name, epochs_distribution) in enumerate(zip(tuners, tuner_names,
                                                               [cumulative_epochs_rs_bo,
                                                                cumulative_epochs_rs_bo,
                                                                hyperband_epochs_distribution])):
        trials = list(tuner.oracle.trials.values())
        trial_scores = [-trial.score for trial in trials]
        best_scores_so_far = np.maximum.accumulate(trial_scores)

        if name == 'Hyperband':
            cumulative_epochs = np.cumsum(epochs_distribution)
        else:
            cumulative_epochs = epochs_distribution

        # Plot the line from the starting point to the first data point using the same color
        if len(best_scores_so_far) > 0:
            plt.plot([starting_point[0], cumulative_epochs[0]],
                     [starting_point[1], best_scores_so_far[0]], linestyle='--', color=colors[i])

        plt.plot(cumulative_epochs, best_scores_so_far, marker='o',
                 linestyle='--', color=colors[i], label=f'{name} Best Score So Far')

    # Plot the starting point after drawing lines to ensure it's on top
    plt.plot(starting_point[0], starting_point[1], marker='o', markersize=5, color='black')
    plt.title('Evolution of Best Inception Score per Epoch', fontsize=18)
    plt.xlabel('Cumulative Epochs', fontsize=16)
    plt.ylabel('Inception Score', fontsize=16)
    plt.xticks(fontsize=14)
    plt.yticks(fontsize=14)
    plt.legend()
    plt.grid(True)
    plt.savefig("/content/drive/MyDrive/CombinedTuner/Best_Inception_Score_Evolution.png")
    plt.show()


# Assuming tuners and epochs variable are defined
plot_scores_with_lines_from_start_point([tuner_RandomSearch, tuner_BO, tuner_Hyperband],
                                        ['RandomSearch', 'Bayesian Optimization', 'Hyperband'],
                                        epochs,
                                        hyperband_epochs_distribution)

"""Calculating FID"""

import numpy as np
from scipy.linalg import sqrtm
```

```
from numpy import cov, trace, iscomplexobj

def calculate_fid(model, real_images, generated_images, eps=1e-8):
    # Function to resize and colorize images
    def resize_color_images(images):
        images_resized = np.array([resize(image, (299, 299, 3)) for image in images])
        return images_resized

    # Resizing and preprocessing images
    real_images_resized = resize_color_images(real_images)
    generated_images_resized = resize_color_images(generated_images)
    real_images_preprocessed = preprocess_input(real_images_resized)
    generated_images_preprocessed = preprocess_input(generated_images_resized)

    # Calculateing ctivations
    act1 = model.predict(real_images_preprocessed)
    act2 = model.predict(generated_images_preprocessed)

    # Calculat ingmean and covariance statistics
    mu1, sigma1 = act1.mean(axis=0), cov(act1, rowvar=False)
    mu2, sigma2 = act2.mean(axis=0), cov(act2, rowvar=False)

    # Adding epsilon to the diagonal for numerical stability
    sigma1 += np.eye(sigma1.shape[0]) * eps
    sigma2 += np.eye(sigma2.shape[0]) * eps

    # Calculating sum squared difference between means
    ssdiff = np.sum((mu1 - mu2)**2.0)

    # Calculating sqrt of product between covariances, adding epsilon for stability
    covmean = sqrtm(sigma1.dot(sigma2))

    # Check and correct complex numbers from sqrt
    if np.iscomplexobj(covmean):
        if not np.allclose(np.imag(covmean), 0, atol=1e-10):
            raise ValueError("Significant␣complex␣components␣found␣in␣sqrtm␣result")
        covmean = np.real(covmean)

    # Calculate FID
    fid = ssdiff + trace(sigma1 + sigma2 - 2.0 * covmean)
    return fid

import numpy as np
import tensorflow as tf
from scipy.linalg import sqrtm
from skimage.transform import resize
from numpy import cov, trace, iscomplexobj, asarray
```

```
from numpy.random import random
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
import tensorflow as tf
from tensorflow.keras.applications.inception_v3 import InceptionV3, preprocess_input
from tensorflow.keras.datasets import mnist
import numpy as np
from scipy import linalg

"""Loading Inception Model"""

inception_model = InceptionV3(include_top=False, pooling='avg', input_shape=(299,299,3))

num_samples = 500

# Determine necessary parameters for FID calculation
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
indices = np.random.choice(range(len(x_train)), num_samples, replace=False)
train_sampled = train_images[indices]
real_images = train_sampled

"""Printing FID"""

def print_fid(tuner, model, real_images):
    best_model = tuner.get_best_models()[0]
    random_latent_vectors = tf.random.normal(shape=(num_samples, best_model.latent_dim))
    generated_images = best_model.generator.predict(random_latent_vectors)

    fid_score = calculate_fid(model, real_images, generated_images)
    print(f"FID score: {fid_score}")

"""Printing FID for different models"""

tuners = [tuner_RandomSearch, tuner_BO, tuner_Hyperband] #the tuners of optimisations created earlier
for tuner in tuners:
  print_fid(tuner, inception_model, real_images)

"""GANDashBoards"""

dashboards = [gan_dashboard_cb_RandomSearch, gan_dashboard_cb_BO, gan_dashboard_cb_Hyperband]

"""Plot losses function"""

def plot_losses(type_of_losses, gan_dashboard_cb, num_of_epochs=None, num_of_trials=None):
        if type_of_losses == "epochs":
            plt.figure(figsize=(10, 5))
            plt.plot(gan_dashboard_cb.gen_losses, label='Generator Loss')
```

```
            plt.plot(gan_dashboard_cb.disc_losses, label='Discriminator Loss')
            plt.title('GAN Losses Per Epoch')
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.legend()
            #plt.savefig("/content/drive/My Drive/BayesianOptimisation/GAN_losses_over_epochs.png")
            plt.show()
        elif type_of_losses == "trials":
            num_of_epochs = len(gan_dashboard_cb.gen_losses) // 21
            # Validate num_of_epochs
            if num_of_epochs is None or num_of_epochs <= 0:
                print("Please provide a valid number of epochs per trial.")
                return

            # Select inglosses for plotting: every [num_of_epochs] starting from [num_of_epochs-1]
            indices = range(num_of_epochs - 1, len(gan_dashboard_cb.gen_losses), num_of_epochs)
            # print(indices)
            selected_gen_losses = [gan_dashboard_cb.gen_losses[i] for i in indices]
            selected_disc_losses = [gan_dashboard_cb.disc_losses[i] for i in indices]
            # print(selected_gen_losses)
            # print(selected_disc_losses)

            # Plotting
            plt.figure(figsize=(10, 5))
            plt.plot(selected_gen_losses, label='Generator Loss')
            plt.plot(selected_disc_losses, label='Discriminator Loss')
            plt.title('GAN Average Losses Per Trial')
            plt.xlabel('Trials')
            plt.ylabel('Loss')
            plt.legend()
            #plt.savefig("/content/drive/My Drive/BayesianOptimisation/GAN_losses_over_trials.png")
            plt.show()

"""Plotting losses"""

for dashboard in dashboards: # iterating over all dadshboards


    print(dashboard.gen_losses)
    print(dashboard.disc_losses)
    #dashboard.plot_losses("epochs")
    #dashboard.plot_losses("trials", 40, 20)
    plot_losses('epochs', dashboard)
    #plot_losses('trials', dashboard)

"""PLotting combined losses per epochs"""
```

```
def plot_combined_losses(dashboards, type_of_loss):
    plt.figure(figsize=(10, 5))
    for dashboard in dashboards:
        if type_of_loss == 'generator':
            plt.plot(dashboard.gen_losses, label=f'1 Generator Loss')
        elif type_of_loss == 'discriminator':
            plt.plot(dashboard.disc_losses, label=f'1 Discriminator Loss')
    plt.title(f'Combined GAN {type_of_loss.capitalize()} Losses Per Epoch')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

"""Plotting combined loss functions"""

plot_combined_losses(dashboards, 'generator')
plot_combined_losses(dashboards, 'discriminator')

"""Plotting all losses combined"""

import matplotlib.pyplot as plt

def plot_combined_losses(dashboards):
    plt.figure(figsize=(12, 6))

    # Plotting Generator Losses
    for i, dashboard in enumerate(dashboards):
        if i == 0:
            plt.plot(dashboard.gen_losses, label='Random Search Generator Loss')
        elif i == 1:
            plt.plot(dashboard.gen_losses, label='Bayesian Optimisation Generator Loss')
        else:
            plt.plot(dashboard.gen_losses, label='Hyperband Generator Loss')

    # Plotting Discriminator Losses
    for i, dashboard in enumerate(dashboards):
        if i == 0:
            plt.plot(dashboard.disc_losses, '--', label='Random Search Discriminator Loss')
        elif i == 1:
            plt.plot(dashboard.disc_losses,'--', label='Bayesian Optimisation Discriminator Loss')
        else:
            plt.plot(dashboard.disc_losses,'--', label='Hyperband Discriminator Loss')

    plt.title('Combined GAN Generator and Discriminator Losses Per Epoch')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
```

```
    plt.show()

plot_combined_losses(dashboards)

"""Saving losses"""

import csv
import os

def save_losses(dashboard, name, save_dir):
    # Ensuring the save directory exists
    os.makedirs(save_dir, exist_ok=True)

    # Defining file paths
    gen_loss_path = os.path.join(save_dir, f'{name}_gen_losses.csv')
    disc_loss_path = os.path.join(save_dir, f'{name}_disc_losses.csv')

    # Saving generator losses
    with open(gen_loss_path, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Epoch', 'GenLoss'])
        for i, loss in enumerate(dashboard.gen_losses):
            writer.writerow([i, loss])

    # Save discriminator losses
    with open(disc_loss_path, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Epoch', 'DiscLoss'])
        for i, loss in enumerate(dashboard.disc_losses):
            writer.writerow([i, loss])

    print(f'Losses␣saved␣for␣{name}␣to␣{save_dir}')

dashboards = [
    (gan_dashboard_cb_RandomSearch, 'RandomSearch'),
    (gan_dashboard_cb_BO, 'BayesianOptimization'),
    (gan_dashboard_cb_Hyperband, 'Hyperband')
]

for dashboard, name in dashboards:
    if name == 'RandomSearch':
        save_losses(dashboard, name, "/content/drive/MyDrive/CombinedTuner/RandomSearch/losses")
    elif name == 'BayesianOptimization':
        save_losses(dashboard, name, "/content/drive/MyDrive/CombinedTuner/BO/losses")
    elif name == 'Hyperband':
        save_losses(dashboard, name, "/content/drive/MyDrive/CombinedTuner/Hyperband/losses")
```

```
"""Calculating confusion matrix"""

import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

import numpy as np

def remove_last_dim(arr):
    """
    Converts an array of shape (1500, 28, 28, 1) to (1500, 28, 28) by removing the last dimension.
    """
    assert arr.ndim == 4  # Ensure the array has four dimensions
    return arr.reshape(arr.shape[0], arr.shape[1], arr.shape[2])

def add_last_dim(arr):
    """
    Converts an array of shape (1500, 28, 28) to (1500, 28, 28, 1) by adding a new last dimension.
    """
    assert arr.ndim == 3  # Ensure the array has three dimensions
    return np.expand_dims(arr, axis=-1)

"""
Here we identify 300 with one discriminator, 500 with other,  with third and
then sum to try to identify which model performs best on all generators combined
"""

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_test = x_test[:1500]


def create_confusion_matrix(tuner, tuner_Hyperband, tuner_BO, tuner_RandomSearch, x_test, save_directory):
    # Identifying best generator model
    best_trial = tuner.oracle.get_best_trials(num_trials=1)[0]
    best_model = tuner.get_best_models(num_models=1)[0]

    # Loading best models from each tuner
    Hyperband_best_model = tuner_Hyperband.get_best_models(num_models=1)[0]
    BO_best_model = tuner_BO.get_best_models(num_models=1)[0]
    RandomSearch_best_model = tuner_RandomSearch.get_best_models(num_models=1)[0]

    # Generating 1500 images with best generator
    num_images_to_generate = 1500  # Total images to generate
    random_latent_vectors = tf.random.normal(shape=(num_images_to_generate, best_model.latent_dim))
```

```
generated_images = best_model.generator(random_latent_vectors, training=False)

# Split generated images into 3 groups of 500
images_group_1 = generated_images[:500].numpy()
images_group_2 = generated_images[500:1000].numpy()
images_group_3 = generated_images[1000:].numpy()

#images_group_1 = remove_last_dim(images_group_1)
#images_group_2 = remove_last_dim(images_group_2)
#images_group_3 = remove_last_dim(images_group_3)

#images_group_1 = images_group_1 * 127.5 + 127.5
#images_group_2 = images_group_2 * 127.5 + 127.5
#images_group_3 = images_group_3 * 127.5 + 127.5



#images_group_1 = np.squeeze(images_group_1, axis=-1)
#images_group_2 = np.squeeze(images_group_2, axis=-1)
#images_group_3 = np.squeeze(images_group_3, axis=-1)


# Classifying real images through the best model's discriminator
real_images = x_test[:num_images_to_generate]
real_images = (real_images-127.5)/127.5
print(real_images[0])
print(images_group_1[0])
real_predictions = best_model.discriminator.predict(real_images)

# Classifying generated images through different discriminators
fake_predictions_1 = Hyperband_best_model.discriminator.predict(images_group_1)
fake_predictions_2 = BO_best_model.discriminator.predict(images_group_2)
fake_predictions_3 = RandomSearch_best_model.discriminator.predict(images_group_3)

# Concatenating all fake predictions altogether
fake_predictions = np.concatenate([fake_predictions_1, fake_predictions_2, fake_predictions_3])
print(images_group_1.shape)
print(fake_predictions)
print(real_images.shape)
print(real_predictions)

# Preparing labels for the confusion matrix
true_labels = np.append(np.ones(len(real_predictions)), np.zeros(len(fake_predictions)))  # Note: Real=1, Fake=0
predictions_binarized = np.append((real_predictions > 0.5).astype(int), (fake_predictions > 0.5).astype(int))

# Generating confusion matrix
conf_matrix = confusion_matrix(true_labels, predictions_binarized)
```

```
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=['Fake', 'Real'],
                yticklabels=['Real', 'Fake'])
    plt.title('Confusion Matrix for Discriminator')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.savefig(save_directory)
    plt.show()

def create_confusion_matrix(tuner, tuner_Hyperband, tuner_BO, tuner_RandomSearch, x_test, save_directory, ):
    # Identifying the best generator model
    best_trial = tuner.oracle.get_best_trials(num_trials=1)[0]
    best_model = tuner.get_best_models(num_models=1)[0]

     # Loading best models from each tuner
    Hyperband_best_model = tuner_Hyperband.get_best_models(num_models=1)[0]
    BO_best_model = tuner_BO.get_best_models(num_models=1)[0]
    RandomSearch_best_model = tuner_RandomSearch.get_best_models(num_models=1)[0]

    # Generating 1500 images with the best generator
    num_images_to_generate = 1500  # Total images to generate
    random_latent_vectors = tf.random.normal(shape=(num_images_to_generate, best_model.latent_dim))
    generated_images = best_model.generator(random_latent_vectors, training=False)

    images_group_1 = generated_images[:500].numpy()
    images_group_2 = generated_images[500:1000].numpy()
    images_group_3 = generated_images[1000:].numpy()

    # Normingalizing the real images the same way as the generated images
    x_test = (x_test - 127.5) / 127.5

    # Classifying real images through the best model's discriminator
    real_predictions = best_model.discriminator.predict(x_test)

    # Classifying generated images through the best model's discriminator
    fake_predictions_1 = Hyperband_best_model.discriminator.predict(images_group_1)
    fake_predictions_2 = BO_best_model.discriminator.predict(images_group_2)
    fake_predictions_3 = RandomSearch_best_model.discriminator.predict(images_group_3)

    fake_predictions = np.concatenate([fake_predictions_1, fake_predictions_2, fake_predictions_3])



    # Binarizing predictions based on a 0.5 threshold (>0/5 - real, <0.5 - fake
    real_predictions_binarized = (real_predictions > 0.5).astype(int)
    fake_predictions_binarized = (fake_predictions <= 0.5).astype(int)
```

```
    # Concatenaing (combining) the true labels and predicted labels for both real and fake images
    true_labels = np.concatenate((np.ones(len(real_predictions)), np.zeros(num_images_to_generate)))
    predicted_labels = np.concatenate((real_predictions_binarized, fake_predictions_binarized))

    # Generating confusion matrix
    conf_matrix = confusion_matrix(true_labels, predicted_labels, labels=[1, 0])

    # Visualizing the confusion matrix
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=['Real', 'Fake'],
                yticklabels=['Real', 'Fake'])
    plt.title('Confusion Matrix for Discriminator')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.savefig(save_directory)
    plt.show()

"""Creating confusion matrix for Random Search"""

create_confusion_matrix(tuner_RandomSearch, tuner_Hyperband, tuner_BO, tuner_RandomSearch, x_test,
                        "/content/drive/MyDrive/CombinedTuner/RandomSearch/confusion_matrix.png")

"""Creating confusion matrix for Bayesian Optimisation"""

create_confusion_matrix(tuner_BO, tuner_Hyperband, tuner_BO, tuner_RandomSearch, x_test,
                        "/content/drive/MyDrive/CombinedTuner/BO/confusion_matrix.png")

"""Creating confusion matrix for Hyperband"""

create_confusion_matrix(tuner_Hyperband, tuner_Hyperband, tuner_BO, tuner_RandomSearch, x_test,
                        "/content/drive/MyDrive/CombinedTuner/Hyperband/confusion_matrix.png")
```

A.1.2 Code for generating box plots and determining medians

This code generated box plots and determined medians across 10 experiments, where Random Search and Bayesian Optimisation were run for 13 trials for 15 epochs, and Hyperband algorithm's maximum number of epochs was also set to 15 (see experimental data in Appendix A.2).

```
# -*- coding: utf-8 -*-

"""Distribution Scores for Inception Score and FID for 15-epoch runs"""
```

```
import os

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

"""Inception Score distribution plot for 15 epochs"""

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import statistics

# Data from the table
data = {
    'Random Search': [5.812, 5.988, 5.619, 5.558, 5.885,
                      5.491,5.523, 5.930, 5.851, 5.362],
    'Bayesian Optimisation': [5.639, 5.811, 5.741, 5.767, 5.822,
                              5.563, 5.852, 5.950, 6.098, 5.971],
    'Hyperband': [6.151, 5.922, 6.098, 6.132, 5.856,
                  5.774, 5.787, 5.614, 5.501, 6.323]
}

print(statistics.median(data["Random Search"]))
print(statistics.median(data["Bayesian Optimisation"]))
print(statistics.median(data["Hyperband"]))

# Converting the data to a DataFrame for easier plotting
df = pd.DataFrame(data)

# Melting the DataFrame to long format for use with seaborn's boxplot function
df_melted = df.melt(var_name='Optimization Technique', value_name='Score')

# Creating the score distribution plot
plt.figure(figsize=(10, 6))
sns.boxplot(x='Optimization Technique', y='Score', data=df_melted)
plt.ylabel('Inception Score', fontsize=16)
plt.xlabel('Optimisation Technique', fontsize=20)
plt.xticks(fontsize=18)  # Adjust fontsize as needed for x-axis tick labels
plt.yticks(fontsize=18)  # Adjust fontsize as needed for y-axis tick labels
plt.savefig("/content/drive/My Drive/CombinedTuner/Distribution_plots/Inception_score_15_epochs")
plt.show()

"""FID distribution plot for 15 epochs"""

import matplotlib.pyplot as plt
import seaborn as sns
```

```
import pandas as pd

# Data from the table
data = {
    'Random Search': [0.25745988697477945, 0.16151168902086305,
                      0.37865820506836695, 0.31615434164471246,
                      0.2571413383895816, 0.48529925075207664,
                      0.17522923881749775, 0.633282757014247,
                      0.2340476889242242, 0.412567555609656],
    'Bayesian Optimisation': [0.40841507059196525, 0.26940141502528586,
                              0.2536681669748142, 0.15168068253572906,
                              0.18868168262761814, 0.14104479071688855,
                              0.23845517670152544, 0.13817984940046527,
                              0.1993634588653225, 0.16458319022707746],
    'Hyperband': [0.18576973585718668, 0.17874622957705775,
                  0.1477036950397352, 0.2025964932898846,
                  0.21232590411035538, 0.14550430901701067,
                  0.21853788726546153, 0.133597468329683,
                  0.16847209031126947, 0.2288876621773397]
    }

print(statistics.median(data["Random Search"]))
print(statistics.median(data["Bayesian Optimisation"]))
print(statistics.median(data["Hyperband"]))




# Converting the data to a DataFrame for easier plotting
df = pd.DataFrame(data)

# Melting the DataFrame to long format for use with seaborn's boxplot function
df_melted = df.melt(var_name='Optimization Technique', value_name='Score')

# Creating the FID score distribution plot
plt.figure(figsize=(10, 6))
sns.boxplot(x='Optimization Technique', y='Score', data=df_melted)
plt.ylabel('FID Score', fontsize=20)
plt.xlabel('Optimisation Technique', fontsize=20)
plt.xticks(fontsize=18)  # Adjust fontsize as needed for x-axis tick labels
plt.yticks(fontsize=18)  # Adjust fontsize as needed for y-axis tick labels
plt.savefig("/content/drive/My Drive/CombinedTuner/Distribution_plots/FID_15_epochs")
plt.show()
```

A.2 Experimental data

Tables of statistical data for 10 experiments, where Random Search and Bayesian Optimisation were run for 13 trials for 15 epochs, and Hyperband algorithm's maximum number of epochs was also set to 15.

A.2.1 Experimental data for Hyperband

Number of the experiment	Inception Score	Fréchet Inception Distance
1	6.151	0.1858
2	5.922	0.1787
3	6.098	0.1477
4	6.132	0.1455
5	5.856	0.2123
6	5.774	0.2026
7	5.787	0.2185
8	5.614	0.2289
9	5.501	0.1336
10	6.323	0.1685

Table 7: Table of experimental data for 15-epoch run for Hyperband

A.2.2 Experimental data for Random Search

Number of the experiment	Inception Score	Fréchet Inception Distance
1	5.812	0.2575
2	5.988	0.1615
3	5.619	0.3787
4	5.558	0.3162
5	5.885	0.2571
6	5.491	0.4853
7	5.523	0.1752
8	5.930	0.6333
9	5.851	0.2340
10	5.362	0.4126

Table 8: Table of experimental data for 15-epoch run for Random Search

A.2.3 Experimental data for Bayesian Optimisation

Number of the experiment	Inception Score	Fréchet Inception Distance
1	5.639	0.4084
2	5.811	0.2694
3	5.741	0.2537
4	5.767	0.1517
5	5.822	0.1886
6	5.563	0.2385
7	5.852	0.1410
8	5.950	0.1382
9	6.098	0.1646
10	5.971	0.1994

Table 9: Table of experimental data for 15-epoch run for Bayesian Optimisation