Procedural Level Generation for Sokoban via Deep Learning: An Experimental Study

Yahia Zakaria, Mayada Hadhoud and Magda Fayek

Abstract—Deep learning for procedural level generation has been explored in many recent works, however, experimental comparisons with previous works are rare and usually limited to the work they extend upon. This paper’s goal is to conduct an experimental study on four recent deep learning procedural level generators for Sokoban to explore their strengths and weaknesses. The methods will be bootstrapping conditional generative models, controllable & uncontrollable procedural content generation via reinforcement learning (PCGRL) and generative playing networks. We will propose some modifications to either adapt the methods to the task or improve their efficiency and performance. For the bootstrapping method, we propose using diversity sampling to improve the solution diversity, auxiliary targets to enhance the models’ quality and Gaussian mixture models to improve the sample quality. The results show that diversity sampling at least doubles the unique plan count in the generated levels. On average, auxiliary targets increases the quality by 24% and sampling conditions from Gaussian mixture models increases the sample quality by 13%. Overall, PCGRL shows superior quality and diversity while generative adversarial networks exhibit the least control confusion when trained with diversity sampling and auxiliary targets.

Index Terms—Procedural content generation, Level generation, Deep learning, Reinforcement learning, Generative models.

I. INTRODUCTION

PROCEDURAL content generation (PCG) is the process of automatically generating content using algorithms. For decades, PCG was a common tool in video games and it was recently employed to enhance the generalization in machine learning [1]. It is time and effort consuming to build a generator that yields useful, interesting and diverse content. Even if the generated content are unique, it can still suffer from the “10,000 bowls of oatmeal” problem [2] where every oatmeal bowl is unique but presents the same experience as every other bowl so there is no “Perceptual Uniqueness”.

One approach to PCG is Procedural Content Generation via Machine Learning (PCGML) [3] where models learn to generate the desired content. If the model is a deep neural network, then the algorithm is classified as Deep Learning. Given the subfield’s novelty, it is rare to find experimental comparisons between methods for multiple reasons. First of all, some aspects of generators are hard to compare such as diversity. Another reason is that some papers attend to novel problems that it is infeasible to compare their work with previous works. Moreover, papers rarely use common benchmarks (same dataset, level size, etc) to evaluate their methods so their results are usually incomparable.

Our main goal is to present an experimental comparison between four recent methods that employ deep learning to generate grid-based 2D games. The benchmark will be Sokoban level generation at size 7×7 . The methods will be bootstrapping conditional generative adversarial networks [4], controllable & non-controllable procedural content generation via reinforcement learning (PCGRL) [5], [6] and generative playing networks (GPN) [7]. The bootstrapping method will be used with Variational Autoencoders (VAE) [8], Generative Adversarial Networks (GAN) [9] and VAEGANs [10] (which combines VAEs and GANs). Throughout the paper, we will present the modifications to either adapt the methods, enhance their properties or improve their efficiency. Thus the contributions of this paper are as follows:

- 1) Adapt the bootstrapping method [4] and Generative Playing Networks [7] to Sokoban level generation.
- 2) Propose Diversity Sampling to improve the generators ability to generate levels with diverse solutions.
- 3) Propose using Auxiliary Targets to enhance the generators quality.
- 4) Propose sampling conditions from a Gaussian Mixture Model to improve the quality of the generated levels.
- 5) Compare between the four methods based on the experimental results.

In section II, we will mention related works, then we will define the level generation problem, state the objectives and explain Sokoban’s rules in section III. Section IV will briefly explain the bootstrapping method and propose modifications to improve its quality and diversity. Section V briefly explains PCGRL and discusses a trick to speedup the training. Section VI briefly explains generative playing networks and adapts it for puzzle generation. We will detail the experimental setup in section VII then show and discuss the results in section VIII. Finally, the conclusion will be presented in section IX.

II. RELATED WORKS

There are multiple recent reviews on PCG with different scopes. PCG via Machine Learning (PCGML) [3] was reviewed where various learning methods in addition to different data sources and representations were discussed. In a more recent work, Deep learning for PCG [11] was reviewed where they also discussed deep learning methods that are potentially useful for PCG. Another recent work [12] presented a review of PCG with a focus on puzzle generation. Our work differs in the fact that we focus on a smaller set of recent works but with the goals of presenting an experimental study rather than a literature review.

A variety of generative models has been applied to PCG such as GANs for Zelda level generation [4] and VAEs for generating and blending levels from multiple games [13]. Given the scarcity of level datasets, the bootstrapping method [4] scavenge the model output for playable levels to be used for training in the upcoming iterations. In addition, conditional embeddings [4] were proposed to improve the generator’s quality. To train without levels, Generative Playing Networks (GPN) [7] learns to generate levels using feedback from an agent trained to play the generated levels. Another approach is Procedural Content Generation via Reinforcement Learning (PCGRL) [5] where an agent learns to modify the level tiles via a reward proportional to the quality improvement. The work was extended [6] to include controls that specify the desired level properties.

III. PROBLEM DEFINITION

This section presents the problem formulation followed by the objectives we seek to optimize. Finally, we will describe Sokoban and the requirements for valid Sokoban levels.

A. Level Generation

A procedural level generator can be abstracted as a function that takes a random sample from the domain and returns a level as shown in (1) where l is the generated level, G is the generator and z is a random sample from the generator’s domain. In case of conditional generators, a condition u , that specifies the desired level properties, is supplied to the generator as shown in (2).

$$l = G(z) \quad (1)$$

$$l = G(z|u) \quad (2)$$

B. Objectives

A good generator should exhibit high quality, high diversity and high controllability (if it is controllable). In this section, we will discuss the objectives we seek to optimize.

1) *Quality*: Measuring the generators’ quality is relatively simple since the functional requirements for a level is verifiable via a solver. We use the percentage of playable levels in a generated sample as the measure of a generator’s quality.

2) *Diversity*: In [4], the lack of diversity was measured as the percentage of duplicate levels. In [6], diversity was measured as the average tile-wise hamming distance between all pairs of playable levels. We argue that both of them are flawed since usually, we can shift, rotate, flip the level or change most of its empty and wall tiles without affecting the level’s solution. Thus, we can have exponentially many levels with high tile diversity that are perceptually equivalent. On the other hand, modifying just a single tile can drastically change the solution. A better measure would be the number of unique generated solutions where a solution is defined as a sequence of actions to reach a goal state. Still, different solutions could boil down to the same high level plan so they would feel similar from a human perspective. In section IV-B, We will suggest another option inspired by hierarchical planning.

3) *Controllability*: A controllable generator should design levels whose properties are as close as possible to the supplied targets. Therefore, we will use the mean absolute error between the target and actual properties of the generated sample.

C. Sokoban

Sokoban is a deterministic 2D grid-based puzzle game. The player can move in 4 directions and push a single crate into an empty or a goal tile. The player wins if every crate is on a goal tile. Sokoban requires long-term planning since actions, that lead to dead ends, are common.

A Sokoban level is considered compilable if it satisfies the following requirements:

- There is only one player (no more, no less).
- The number of crates are equal to the number of goals.
- All tiles are in one of the 7 states shown in TABLE I.
- At least one crate is not on a goal tile since we are not interested in levels that are already solved.

A level is considered playable if it is compilable and solvable by the solver used in our experiments.

TABLE I
SOKOBAN TILESET

Tile Content	Sprite	Tile Content	Sprite
Empty		Wall
Goal		Crate
Player		Crate & Goal
Player & Goal

IV. BOOTSTRAPPING GENERATIVE MODELS

Generative models learn to capture the probability distribution of the training data, so they can overfit on small datasets and mostly generate copies from the training data. To mitigate this issue, the bootstrapping method [4] augments the training set with any new playable levels generated by the model. The augmentation process is run multiple times during training so the final model is trained on both the original dataset and the playable levels generated by its previous versions.

Alongside bootstrapping, conditional embeddings were proposed [4] to enhance the model’s ability to respect the tile frequency requirements of playable Zelda levels. Thus the conditional embeddings were derived from the frequency of each tile in the level. To adapt conditional embeddings for Sokoban, we use different inputs:

- 1) **Walls**: The wall count divided by the level’s area.
- 2) **Crates**: The crate count divided by the level’s area.
- 3) **Solution Length**: The minimum number of actions required to solve the level divided by the level’s area.
- 4) **Player distance to the center**: The Manhattan distance between the player and the level’s center. The location on each axis is normalized to be in the range $[-1, 1]$ where $(0, 0)$ is the center.

The walls and crates conditions are inspired by the original paper [4]. The solution length was added to vary and control the generated level's difficulty. The player distance to the center was added to resolve an issue where player was always located at a certain position in all the generated levels.

A. Generative Models

We test the bootstrapping method with three models: Generative Adversarial Networks (GAN) [9] (to which bootstrapping was originally applied), Variational Autoencoders (VAE) [8] and Variational Autoencoder Generative Adversarial Networks (VAEGAN) [10]. For GANs, we use the hinge adversarial loss [15] shown in (3) and (4) following the original paper [4]. L_D and L_G are the loss functions of the discriminator D and generator G respectively. The real data x is sampled from the training set P_x , the latent vector z is sampled from a standard normal distribution $\mathcal{N}(0, I)$ and u is the condition vector.

$$L_D = \mathbb{E}_{x \sim P_x} [\max(0, 1 - D(x|u))] + \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\max(0, 1 + D(G(z|u)|u))] \quad (3)$$

$$L_G = -\mathbb{E}_{z \sim \mathcal{N}(0, I)} [D(G(z|u)|u)] \quad (4)$$

For VAEs, the objective is to optimize the evidence lower bound loss shown in (5). To avoid confusion with the discriminator (which is denoted by D), we will use G to denote the decoder. The same loss L_E & L_G is applied to the encoder E and decoder G . The reconstruction term uses the cross-entropy loss $\mathcal{H}$ over the tiles. The latent vector z is sampled from the distribution predicted by the encoder.

$$L_E = L_G = -\mathbb{E}_{x \sim P_x} [\mathcal{H}(x, G(z|u)) + D_{KL}(E(x|u) \parallel \mathcal{N}(0, I))] , \quad z \sim E(x|u) \quad (5)$$

VAEGANs combine GAN and VAE losses but replace the tile-wise reconstruction loss with a feature-wise loss where the feature vector h_D is extracted from the discriminator's last hidden layer. Similar to the original VAEGAN, our motivation is to incentivize reconstructions that are feature-wise, rather than tile-wise, similar to the original. The VAEGAN losses are shown in (6), (7) and (8) where the parameter α controls the weight of the feature error loss in the generator loss.

$$L_D = \mathbb{E}_{x \sim P_x} [\max(0, 1 - D(x|u))] + 0.5 \mathbb{E}_{x \sim P_x} [\max(0, 1 + D(G(z_r|u)|u))] + 0.5 \mathbb{E}_{z_r \sim \mathcal{N}(0, I)} [\max(0, 1 + D(G(z_r|u)|u))] , \quad z_r \sim E(x|u) \quad (6)$$

$$L_G = 0.5 \mathbb{E}_{z \sim \mathcal{N}(0, I)} [-D(G(z|u)|u)] + 0.5 \mathbb{E}_{x \sim P_x} [-D(G(z_r|u)|u)] + \alpha \|h_D(x) - h_D(G(z_r|u))\|^2 , \quad z_r \sim E(x|u) \quad (7)$$

$$L_E = -\mathbb{E}_{x \sim P_x} [\|h_D(x) - h_D(G(z_r|u))\|^2 + D_{KL}(E(x|u) \parallel \mathcal{N}(0, I))] , \quad z_r \sim E(x|u) \quad (8)$$

B. Diversity Sampling

In the bootstrapping method, the model is trained on the dataset augmented by its previous versions' playable output. So if at any time, the model collapses into a single mode (which is likely), the upcoming training distributions will drift towards that mode causing a snowball effect. Our first attempt to mitigate this issue was by sampling conditions for augmentation from a uniform distribution ($U(\min u, \max u)$, $u \in$ training data conditions) rather than from the dataset to avoid oversampling from modes. However, the generator still created levels with similar solutions. After all, the generator can satisfy the crate condition by adding more already-on-goal crates. It can also vary the solution length by repositioning the player and adding walls to lengthen the path to the first push.

Therefore, we propose diversity-sampling based on the level solution where the training data are grouped into clusters of similar solutions. To create a training batch for the model, we uniformly sample clusters, then randomly pick levels from them. The motivation is to show the model a diverse view of the solutions available in the dataset. To decide if two solutions are similar or not, we use a distilled version of the solution, that we call a "Signature", where solutions are considered similar if they have the same signature. It is computed as shown in Algorithm IV-B where the input is the solution as a string of actions which can be moves (denoted by $\{r, l, d, u\}$) or pushes (denoted by $\{R, L, D, U\}$). It is inspired by hierarchical planning where the solution is represented as high-level actions (HLA) and each HLA translates to "go to a crate and push it a number of times in a certain direction". In the signature, we only keep the push directions and ignore the other parameters (the crate and the number of pushes) to simplify clustering. To make the signature rotation and flipping invariant, we rotate it till the first action is "R". After that, if there are any vertical actions and the first one is "D", we flip the signature along the y-axis. While there are probably better representations, signatures were enough to significantly increase the unique generated solutions. Searching for better options is left for future work.

C. Auxiliary Targets

A Sokoban level is compilable if and only if the number of goals and crates are equal. GANs have a hard time learning this constraint by just observing the dataset, so we propose adding an extra target to the discriminator (inspired by Auxiliary Classifier GAN [16]) where it has to differentiate between compilable and uncompileable levels in real and fake data. In addition, we add regression targets for the object count (walls, players, crates and goals) in the level since they are crucial for detecting the level compilability. This extension proved useful for enhancing the quality of GANs and VAEGANs.

D. Sampling Conditions from a Gaussian Mixture Model

To sample from a conditional model, we need a latent vector and a condition. While latent vectors are sampled from a standard normal distribution, conditions have no parameterized distribution to sample from. The first option is to sample

Algorithm 1 Extract Signature from Solution

```

procedure EXTRACTSIGNATURE(actions)
  signature  $\leftarrow []$ 
  if actions[0] is a push then
    append actions[0] to signature
  end if
  for index from 1 to length(actions) - 1 do
    if actions[index] is a push then
      if actions[index]  $\neq$  actions[index - 1] then
        append actions[index] to signature
      end if
    end if
  end for
  while signature[0]  $\neq$  'R' do
    rotate signature by 90°
  end while
  for index from 1 to length(signature) - 1 do
    if signature[index] = 'U' then
      break
    else if signature[index] = 'D' then
      flip signature along y-axis
    end if
  end for
  return signature
end procedure

```

conditions from a uniform distribution. However, some regions in the distribution could be impossible to satisfy (e.g. one crate only with a solution of 100 steps). The second options is to sample from the dataset but this requires bundling all the training conditions (thousands of vectors) alongside the deployed model. Also, it provides no convenient way to sample conditions given a fixed set of user-supplied targets (e.g. the user may want to specify the difficulty but not care about the number of crates, walls or player position).

Therefore, we propose fitting the conditions of the final training dataset into a Gaussian mixture model (GMM). It is much more compact than storing all the conditions and can be sampled unconditionally as shown in (9) and conditionally as shown in algorithm IV-D where the query variables are b with value x_b and the unknown variables are a . The Gaussian mixture model is specified by n normal distributions $\mathcal{N}(\mu_k, \Sigma_k)$ where the weight of each distribution in the mixture is w_k .

$$u \sim \mathcal{N}(\mu_k, \Sigma_k), \quad k \sim [1, n], \quad P(k) = w_k \quad (9)$$

V. PROCEDURAL CONTENT GENERATION VIA REINFORCEMENT LEARNING

PCGRL [5] formulates level generation as a Markov decision process where the agent's actions modify the level and the reward is proportional to the level's quality improvement. A PCGRL Environment has multiple representations out of which we will focus on the following:

- 1) **Narrow:** The agent can modify the tile at its location while moving across the level in a scanline fashion.

Algorithm 2 Conditionally Sample from a Gaussian Mixture Model

```

procedure CONDITIONALSAMPLE( $x_b, \mu_{1..n}, \Sigma_{1..n}, w_{1..n}$ )
  for  $k$  from 1 to  $n$  do
     $\begin{bmatrix} \Sigma_{k_{aa}} & \Sigma_{k_{ab}} \\ \Sigma_{k_{ba}} & \Sigma_{k_{bb}} \end{bmatrix} \leftarrow \Sigma_k$ 
     $\begin{bmatrix} \mu_{k_a} \\ \mu_{k_b} \end{bmatrix} \leftarrow \mu_k$ 
     $\bar{\mu}_k \leftarrow \mu_{k_a} + \Sigma_{k_{ab}} \Sigma_{k_{bb}}^{-1} (x_b - \mu_{k_b})$ 
     $\bar{\Sigma}_k \leftarrow \Sigma_{k_{aa}} - \Sigma_{k_{ab}} \Sigma_{k_{bb}}^{-1} \Sigma_{k_{ba}}$ 
     $\hat{w}_k \leftarrow \frac{w_k}{\sqrt{|\Sigma_{k_{bb}}|}} e^{-\frac{1}{2} (x_b - \mu_{k_b})^\top \Sigma_{k_{bb}}^{-1} (x_b - \mu_{k_b})}$ 
  end for
  sample  $k \sim [1, n]$  where  $P(k) = \frac{\hat{w}_k}{\sum_{i=1}^n \hat{w}_i}$ 
  sample  $u_a \sim \mathcal{N}(\bar{\mu}_k, \bar{\Sigma}_k)$ 
  return  $\begin{bmatrix} u_a \\ x_b \end{bmatrix}$ 
end procedure

```

- 2) **Turtle:** The agent can modify the tile at its location but it can instead move to an adjacent tile.
- 3) **Wide:** The agent can modify any tile at any location in the level. The agent has no location.

Controllable PCGRL [6] extends the observation by adding extra channels containing +1, -1 or 0 to specify whether the corresponding level property should increase, decrease or stay still respectively. For Sokoban, the controllable properties are the number of crates and the solution length.

The PCGRL reward function for Sokoban motivates the agent to create a solvable level with one connected region, a crate count within a specific range, and to increase the solution length. In Controllable PCGRL, the reward function motivates creating levels with the specified targets.

To maintain diversity, PCGRL limits the number of allowed changes. Otherwise, the agent would overwrite the whole input and memorize a single high-reward level. On the other hand, Controllable PCGRL left the limit at 100% of the map.

A. Reducing the Training Time

For training PCGRL on Sokoban, the bottleneck is the reward calculation since it requires the solution length. Sokoban solving is time consuming since the search space is usually huge. The first optimization was to cache the solutions since Turtle and Narrow agents spend most of the time moving rather than applying changes. Caching almost halved the training time but it was still slow to run for 10^8 steps.

We noticed that the CPU utilization was low which implied that most of the cores stay idle while a few cores were still searching. Since rewards are not needed during experience collection and can be delayed till the optimization phase, we compute the solutions asynchronously while the experience collection resumes. It is noteworthy that we remove the solution length threshold from the termination condition. Before the agent optimization phase, the algorithm halts till the solvers

finish. This change decreased the training time from 9 days to just 2 days and 8 hours on our machine.

Unfortunately, Controllable PCGRL environments require the solution to compute the next observation, so we had to rewrite its solver using the C programming language which ran $400\times$ faster than the original solver. Thus the aforementioned idea is only useful for uncontrollable PCGRL with games that have no fast solver.

VI. GENERATIVE PLAYING NETWORKS

GPNs [7] are like GANs but the discriminator is replaced by a reinforcement learning (RL) agent and the generator trains to minimize the absolute of the generated level's predicted value. In other words, it trains to generate hard yet playable levels, while the RL agent trains to win the generated levels in the least number of steps. Although GPNs do not require a dataset, the agent can be pretrained on a few levels to facilitate the learning. The reward function is defined in (10) where t_n is the episode length, t_{max} is the maximum allowed episode length and r is the reward returned by the GVGAI environment.

$$R_{action}(S_n) = \begin{cases} 1 - \frac{t_n}{t_{max}} & \text{if player wins} \\ -1 + \frac{t_n}{t_{max}} & \text{if player loses} \\ \frac{1}{t_{max}} & r > 0 \end{cases} \quad (10)$$

This method was used to generate Zelda levels only but it is applicable to Sokoban after modifying the reward function. The reward function (10) is suitable for action games since the agent is rewarded for surviving longer if it loses. But in Sokoban, the only losing condition is to run out of steps before solving the level, so the survival-time reward should be omitted from the reward function as shown in (11).

$$R_{puzzle}(S_n) = \begin{cases} 1 - \frac{t_n}{t_{max}} & \text{if player wins} \\ -1 & \text{if player loses} \\ \frac{1}{t_{max}} & r > 0 \end{cases} \quad (11)$$

VII. EXPERIMENTAL SETUP

In this section, we will introducing the dataset, discuss the network architectures used by the four methods. Finally, we will detail the rest of the training and generation configuration.

A. Dataset

The dataset shown in Fig.1 was used for bootstrapping the generative models and pretraining the GPN agent. It contains 12 levels with varying degrees of difficulty. While pretraining the GPN, we randomly flip and rotate the level after loading it. When used for bootstrapping generative models, the dataset is expanded adding the intermediate states along the solution path of every level into the dataset to augment it with varying values for the solution length condition. The expanded dataset contains 167 levels with 145 unique solutions and 16 unique signatures. The maximum solution length found in the dataset is 31 steps. The expanded dataset is distributed in the behavior space as shown in Fig.2.

Fig. 1. Sokoban Dataset

Fig. 2. Dataset Distribution

B. Network Architecture

For the bootstrapping method, we use the same generator architecture for all three generative models. The encoders and discriminators are also the same except for the last layer. All the networks utilizes self-attention and instance normalization with learnable affine parameters. In addition, we use spectral normalization on every convolution layer in the discriminator. A skip connection and a trainable parameter γ was added to the self-attention module as shown in Fig.3 to match the original self-attention GAN [17]. All the internal activation functions are leaky ReLUs with 0.2 negative slope. The four conditions are treated similarly in all modules where it goes through two feed-forward layers before being concatenated with the input and also supplied to all the self-attention modules.

For PCGRL, We follow the agent architectures found in the official repository with only one more convolution layer (3×3) and ReLU activation added to each network. For GPN, the agent and the generator networks were redesigned to fit 7×7 levels. Following the GPN repository, we used a Residual Network [18] followed by a single GRU unit [19] for the agent. The generator uses Convolution Transpose (3×3), Batch Normalization, ReLU and Dropout.

Fig. 3. Self-Attention Module

C. Training and Generation Configuration

For the bootstrapping method, the models are trained for 10^4 iterations with 32 batch size and RMSProp [20] at 10^{-3} learning rate. Every 100 iterations, we scavenge new playable levels from a generated sample of 128 levels. For VAEGAN, the feature loss factor α is 10^{-2} . For auxiliary targets, the compilability loss is binary cross entropy while the regression losses are L1 with weights 1/49, 1/5, 1/5, 1 for the walls, goals, crates and players respectively. For fitting the conditions to a GMM, we used a Bayesian Gaussian mixture model [21] with 16 components. To verify levels, we use breadth-first search with a limit of 10^7 iterations.

For uncontrollable PCGRL, the agent trains for 10^8 steps with the solver power set to 10^4 iterations. For controllable PCGRL, the agent trains for 5×10^8 steps with the solver power set to 2×10^4 iterations. The tileset was constrained to only tiles with a single object, otherwise the agent always adds a goal in the same tile with every crate. The agents were trained using Proximal Policy Optimization [22] with Adam [23] at 10^{-4} learning rate. We use separate networks for the actor and the critic, since it improved the training process. The change percentage was set to 40% and 100% for uncontrollable and controllable PCGRL respectively. During generation, the change percentage was set to 100% for both. For uncontrollable PCGRL, the maximum crate count was 5. For controllable PCGRL, the targets range was $[1, 10]$ for the crates and $[1, 105]$ for the solutions. We test each agent twice; once as a deterministic policy and once as a stochastic policy. For deterministic policies, the episode terminates as soon as a state is revisited as it means the agent is stuck in a loop.

For GPN, we pretrain the agent for 2×10^7 steps with the episode limited to 50 steps. After pretraining, the GPN is trained for 100 epochs where in each epoch, the generator is trained to optimize the utility for 10 iterations, then trained to increase the diversity for 90 iterations followed by training the agent for 10^6 iterations with a 50% chance of sampling the training level from the elites. The generator's batch size is 128 and the latent size is 512.

As a baseline, two random generators are used. The first is a "Naive" level generator where tiles are selected randomly

with some weights similar to the initialization of PCGRL environments. The second is a "Compilable" level generator where the sampling process ensures level compilability.

For evaluation, each generator creates 10,000 levels from random inputs. To evaluate the controllability, we request 100 levels for every control target in the grid: $crates \in [1, 10]$, $solution\ length \in [1, 100]$. For generative models, the other conditions (player distance to center and wall ratio) are sampled from the GMM given the target conditions. Since generative models are fast to train, we run each configuration 5 times then we report the mean and the standard deviation of the results. All the experiments were conducted on a machine equipped with a 6 core CPU and an Nvidia GTX 980 Ti GPU.

VIII. RESULTS AND DISCUSSION

In this section, we will show the results and compare the methods in each of the following aspects: Quality, Diversity, Training time, Generation time and Controllability.

A. Quality and Diversity

TABLE II shows the statistics related to each generator's quality and diversity. DS, AT and GMM denote Diversity Sampling, Auxiliary Targets and Gaussian Mixture Models respectively. Copies are the percentage of playable levels copied from the initial dataset. Tile diversity is the average hamming distance between all pairs of playable levels. Among baseline generators, the compilable level generator created 524 playable levels with 524 unique solutions and 195 unique signatures which outperforms the solution diversity of many generative models despite having only a few playable levels.

For GANs, VAEs and VAEGANs, the first row in their table section includes the results of running without bootstrapping or any other modification. For GAN and VAEGAN, training without bootstrapping decreases both the quality and the diversity. For VAE, the playability percentage is higher without bootstrapping but at a significant decrease in unique signatures. For all models, the duplicates and copies are significantly higher without bootstrapping.

From the results, we can notice the following:

- Diversity sampling always increases the unique solutions and signatures. However, it increases the duplication rate and usually decreases the playability percentage.
- Auxiliary targets always increase the playability percentage and decrease the duplicates. When used with diversity sampling, it increases the unique solutions and signatures.
- Sampling conditions from a GMM always increases the playability percentage.

Among the generative models, VAEGANs have the highest playability percentage but is outperformed by GANs and VAEs in unique solutions and signatures. VAEs outperforms GANs in generating unique solutions and signatures except when GANs use both diversity sampling and auxiliary targets.

Fig.4 shows a random sample (without duplicates) of levels generated by GANs (with auxiliary targets) with and without diversity sampling. While tiles vary in both samples, the solutions vary significantly when diversity sampling is applied.

TABLE II
QUALITY AND DIVERSITY STATISTICS
(DS: DIVERSITY SAMPLING, AT: AUXILIARY TARGETS, GMM: SAMPLING CONDITIONS FROM THE GAUSSIAN MIXTURE MODEL)
(THE SYMBOL $\sim$ MEANS THAT THE GENERATION WAS DONE VIA A STOCHASTIC POLICY)

Name	DS	AT	GMM	Playable	Duplicates	Tile Diversity	Unique Solutions	Unique Sig.	Copies
Random	Naive			0.01%	0.0%	0.00	1.0	1.0	-
	Compilable			5.2%	0.0%	0.38	524.0	195.0	-
GAN	No Bootstrapping			21.5% $\pm$ 21.2%	74.8% $\pm$ 42.2%	0.11 $\pm$ 0.13	22.8 $\pm$ 18.2	7.0 $\pm$ 5.9	50.6% $\pm$ 47.3%
	.	.	.	28.0% $\pm$ 10.6%	39.0% $\pm$ 9.2%	0.35 $\pm$ 0.05	175.0 $\pm$ 93.3	70.2 $\pm$ 28.2	0.22% $\pm$ 0.30%
	.	.	✓	54.1% $\pm$ 11.7%	42.0% $\pm$ 7.6%	0.33 $\pm$ 0.04	182.4 $\pm$ 102.6	73.6 $\pm$ 38.4	0.20% $\pm$ 0.27%
	.	✓	.	73.8% $\pm$ 4.1%	18.2% $\pm$ 9.4%	0.40 $\pm$ 0.04	136.0 $\pm$ 45.0	41.8 $\pm$ 16.8	0.07% $\pm$ 0.07%
	.	✓	✓	81.3% $\pm$ 3.8%	17.9% $\pm$ 8.9%	0.37 $\pm$ 0.03	124.0 $\pm$ 41.7	36.2 $\pm$ 17.9	0.11% $\pm$ 0.10%
	✓	.	.	29.7% $\pm$ 8.2%	67.0% $\pm$ 7.1%	0.29 $\pm$ 0.08	234.6 $\pm$ 39.4	151.2 $\pm$ 32.1	0.01% $\pm$ 0.02%
	✓	.	✓	37.7% $\pm$ 9.8%	64.1% $\pm$ 10.5%	0.29 $\pm$ 0.06	244.8 $\pm$ 43.4	154.8 $\pm$ 36.5	0.02% $\pm$ 0.05%
	✓	✓	.	57.0% $\pm$ 12.1%	37.9% $\pm$ 5.1%	0.36 $\pm$ 0.07	665.8 $\pm$ 184.3	456.6 $\pm$ 124.7	0.00% $\pm$ 0.00%
VAEGAN	No Bootstrapping			38.0% $\pm$ 5.5%	86.2% $\pm$ 2.8%	0.32 $\pm$ 0.03	134.0 $\pm$ 45.7	32.6 $\pm$ 12.2	67.7% $\pm$ 9.9%
	.	.	.	53.1% $\pm$ 8.2%	32.9% $\pm$ 8.7%	0.30 $\pm$ 0.04	149.6 $\pm$ 88.4	46.0 $\pm$ 33.2	0.24% $\pm$ 0.19%
	.	.	✓	79.0% $\pm$ 5.4%	37.8% $\pm$ 9.3%	0.29 $\pm$ 0.04	135.6 $\pm$ 79.9	42.8 $\pm$ 26.5	0.32% $\pm$ 0.26%
	.	✓	.	75.1% $\pm$ 12.6%	27.6% $\pm$ 8.4%	0.33 $\pm$ 0.04	134.2 $\pm$ 61.6	37.2 $\pm$ 29.9	0.01% $\pm$ 0.02%
	.	✓	✓	87.6% $\pm$ 4.8%	23.5% $\pm$ 4.2%	0.32 $\pm$ 0.02	100.2 $\pm$ 38.4	23.8 $\pm$ 13.2	0.04% $\pm$ 0.06%
	✓	.	.	49.6% $\pm$ 10.6%	50.7% $\pm$ 16.3%	0.32 $\pm$ 0.02	251.0 $\pm$ 71.6	126.4 $\pm$ 21.1	0.14% $\pm$ 0.08%
	✓	.	✓	57.7% $\pm$ 9.7%	53.0% $\pm$ 18.1%	0.32 $\pm$ 0.02	237.0 $\pm$ 76.9	117.4 $\pm$ 27.2	0.17% $\pm$ 0.12%
	✓	✓	.	69.5% $\pm$ 7.8%	51.8% $\pm$ 16.6%	0.30 $\pm$ 0.01	331.6 $\pm$ 200.2	151.0 $\pm$ 96.6	0.00% $\pm$ 0.01%
VAE	No Bootstrapping			34.3% $\pm$ 4.0%	72.5% $\pm$ 4.0%	0.37 $\pm$ 0.01	406.0 $\pm$ 38.9	49.8 $\pm$ 8.3	70.1% $\pm$ 4.6%
	.	.	.	30.4% $\pm$ 6.7%	28.0% $\pm$ 4.2%	0.30 $\pm$ 0.02	492.6 $\pm$ 130.1	171.0 $\pm$ 70.4	0.22% $\pm$ 0.11%
	.	.	✓	52.5% $\pm$ 5.0%	29.4% $\pm$ 5.7%	0.28 $\pm$ 0.02	519.0 $\pm$ 134.7	142.2 $\pm$ 53.0	0.25% $\pm$ 0.03%
	✓	.	.	32.4% $\pm$ 5.2%	39.1% $\pm$ 7.5%	0.31 $\pm$ 0.01	623.4 $\pm$ 123.7	262.0 $\pm$ 100.6	0.17% $\pm$ 0.08%
PCGRL	✓	.	✓	46.4% $\pm$ 0.8%	37.6% $\pm$ 6.5%	0.29 $\pm$ 0.01	675.6 $\pm$ 140.3	256.0 $\pm$ 105.8	0.26% $\pm$ 0.12%
	Narrow			58.4%	0.0%	0.42	5758.0	1869.0	-
	Narrow $\sim$			65.8%	0.0%	0.40	6527.0	2552.0	-
	Turtle			63.2%	0.0%	0.50	6316.0	2513.0	-
	Turtle $\sim$			85.9%	0.0%	0.43	8592.0	4287.0	-
	Wide			0.1%	0.0%	0.52	5.0	4.0	-
	Wide $\sim$			79.1%	4.9%	0.18	6234.0	1792.0	-
	C. Turtle			68.3%	0.1%	0.54	6769.0	216.0	-
GPN	C. Turtle $\sim$			86.2%	3.7%	0.49	7955.0	162.0	-
				48.7%	99.4%	0.08	5.0	2.0	37.63%

(a) GAN with Auxiliary Targets but without Diversity Sampling

(b) GAN with Auxiliary Targets and Diversity Sampling

Fig. 4. Samples to Demonstrate the Effect of Diversity Sampling

For uncontrollable PCGRL, all agents generate high quality levels with significantly more unique solutions and signatures than any generative models. The only exception is the wide deterministic agent where it gets stuck in a loop after 2 or 3 steps. In general, acting stochastically improves the results for all agents. Among the three agents, the turtle agent has the highest quality and solution diversity.

The controllable PCGRL agent achieved the highest playability percentage among the PCGRL agents, but with significantly less unique signatures. Fig.5 shows a random sample of the agent's output where it shows that the agent usually adds crates and goals in adjacent pairs. Therefore, most of the signatures are expected to be repeated strings of the action "U". Limiting the change percentage during training, like with the other PCGRL agents, may help fix this issue.

Fig. 5. Samples of Solvable Levels from Controllable PCGRL

For GPNs, the playability percentage is not low, however, most levels are duplicates and both the tile and solution diversity are very low. By inspecting a generated sample shown in Fig.6, it seems that the levels are minor modifications of two levels from the original dataset. By testing the GPN

agent, it only solved the 5 trivial levels (out of the 12 dataset levels) among which two levels require the most steps. Thus the generator prefers to imitate these two levels. We believe the bottleneck is the agent since it is hard to train an RL agent to play Sokoban. Given these results, we did not attempt to train a GPN from nothing.

Fig. 6. Samples of Solvable Levels from GPN

For controllable models, GANs with diversity sampling and auxiliary targets present a good compromise among the generative models and outperform controllable PCGRL in signature diversity. When no control is required, the PCGRL turtle agent significantly outperforms other models' diversity while being on par with their best playability percentage.

Fig.8 shows the expressive range of every generative model (without GMM) where we plot the average of the 5 runs. The differences across runs are minor except for VAEGANS and GANs, when trained without diversity sampling. Compared to the initial dataset distribution in Fig.2, all the generative models have ranges that expand beyond the initial dataset.

Out of all models, GANs with auxiliary targets and diversity sampling has the widest expressive range. Diversity sampling also shifts the mode from the bottom-left bin to a higher solution length and crate count. However, the effect of diversity sampling is less prominent on VAEs compared to GANs. For GANs and VAEGANs, models trained with auxiliary targets exhibit more expansion on the crates axis.

Fig. 8. Expressive Range of Generative Models (DS: Diversity Sampling, AT: Auxiliary Targets)

Fig. 7. Expressive Range of PCGRL (Stochastic)

TABLE III
TRAINING AND GENERATION TIME

Name	DS	AT	Training	Inference	Inferences / Level	Level Generation
GAN	.	.	15 : 48s ± 06 : 56s	2.08ms	1	2.08ms
	.	✓	9 : 40s ± 03 : 35s	2.08ms	1	2.08ms
	✓	.	20 : 28s ± 09 : 24s	2.08ms	1	2.08ms
	✓	✓	21 : 41s ± 11 : 36s	2.08ms	1	2.08ms
VAEGAN	.	.	18 : 37s ± 00 : 16s	2.08ms	1	2.08ms
	.	✓	20 : 32s ± 00 : 17s	2.08ms	1	2.08ms
	✓	.	18 : 40s ± 00 : 09s	2.08ms	1	2.08ms
	✓	✓	20 : 30s ± 00 : 23s	2.08ms	1	2.08ms
VAE	.	.	7 : 14s ± 01 : 16s	2.08ms	1	2.08ms
	✓	.	8 : 10s ± 02 : 42s	2.08ms	1	2.08ms
PCGRL	Narrow					
	Narrow (Stochastic)		4 days 00 : 15 : 19s	0.82ms	127.0 ± 28.3 1110.1 ± 315.2	104.1ms ± 23.2ms 910.3ms ± 258.5ms
	Turtle					
	Turtle (Stochastic)		2 days 08 : 02 : 17s	0.83ms	54.8 ± 15.8 2381.7 ± 203.8	45.5ms ± 13.1ms 1976.8ms ± 169.2ms
	Wide					
GPN	Wide (Stochastic)		3 days 01 : 18 : 56s	1.28ms	2.4 ± 0.7 2397.4 ± 58.8	3.1ms ± 0.9ms 3068.7ms ± 75.3ms
	C. Turtle					
	C. Turtle (Stochastic)		4 days 18 : 39 : 05s	0.83ms	70.2 ± 27.8 1013.9 ± 963.7	58.3ms ± 23.1ms 840.8ms ± 799.9ms
			2 days 18 : 19 : 48s	1.76ms	1	1.76ms

For uncontrollable PCGRL, all representations have mostly similar ranges as seen in Fig.7 and they barely extends beyond 5 crates which is the limit set in the configuration. In Fig.7d, the expressive range of the controllable turtle agent shows more expansion on the crates axis up to 10 crates but it fails to generate levels with long solutions using a few crates.

B. Training and Generation Time

TABLE III shows the training and generation time for each generator. The bootstrapping method is significantly faster to train where it finishes in less than an hour while PCGRL and GPN requires days. For PCGRL, the training time mainly depends on the frequency of edit actions so the narrow and wide agents took more time since their action space contains edit actions only. The controllable PCGRL agent is trained for $5\times$ the steps used for uncontrollable agents, however, it was the only experiment that used the C solver so it only took nearly twice the training time of the uncontrollable turtle agent. GPN took 2 days and 18 hours due to the large network architectures and the relatively slow GVGAI environments.

TABLE III also includes the inference times for a single network evaluation call which is calculated by supplying a random input (batch size = 1) for 10^4 iterations. GANs, VAEs & VAEGANs have the same time since they share the same generator architecture. While acting deterministically, the turtle agent requires the least steps, since it reaches its target tiles faster. While acting stochastically, the narrow agent requires the least steps. The controllable turtle also finish faster since it terminates as soon as the targets are achieved. Overall, deterministic policies finish fast since they are terminated as soon as they get stuck in a loop. Overall, the generative models generate levels faster since they require only one call per level batch.

C. Controllability

TABLE IV shows the control error for controllable models. The error is the absolute difference between the target and the actual level properties. The crates error is calculated for all the generated levels while the solution length error is calculated for playable levels only. It is notable that the increase in the playability percentage usually leads to an increase in the solution-length error. For the models with relatively high playability, GANs with diversity sampling and auxiliary targets have the lowest solution length error. In comparison, PCGRL presents a significant playability improvement but at a notable increase in the solution-length error. It seems that using auxiliary targets without diversity sampling and vice versa increase the solution length error. And for PCGRL, acting stochastically improves both the control error and the playability.

TABLE IV
CONTROL ERROR
("∼" MEANS THAT GENERATION WAS DONE VIA A STOCHASTIC POLICY)

Name	DS	AT	Error		Playable
			Crate	Sol-Length	
GAN	.	.	2.8 ± 1.3	5.8 ± 3.2	21.7% ± 20.5%
	.	✓	2.2 ± 0.7	16.7 ± 18.7	45.2% ± 26.5%
	✓	.	2.9 ± 0.6	11.6 ± 3.8	25.8% ± 13.3%
	✓	✓	2.3 ± 0.4	10.1 ± 3.0	48.1% ± 19.3%
VAEGAN	.	.	3.5 ± 0.6	4.8 ± 1.2	20.3% ± 5.5%
	.	✓	7.3 ± 7.0	21.4 ± 17.9	39.3% ± 29.9%
	✓	.	3.0 ± 0.4	8.7 ± 4.6	28.7% ± 5.6%
	✓	✓	3.1 ± 0.3	10.5 ± 4.5	49.3% ± 11.6%
VAE	.	.	3.1 ± 0.5	6.2 ± 2.0	23.7% ± 8.0%
	✓	.	2.6 ± 0.3	7.1 ± 1.3	24.8% ± 4.8%
PCGRL	C. Turtle		2.6	21.9	68.9%
	C. Turtle ∼		2.5	17.5	86.6%

Fig. 9. Crates Confusion Matrices (DS: Diversity Sampling, AT: Auxiliary Targets)

Next, we visualize the confusion matrices for the crates in Fig.9 and for the solution lengths in Fig.10. The matrices include all the generated levels, where the first column, in the solution length matrices, is dedicated for unplayable levels.

Fig.9 shows that for GANs, auxiliary targets decreases the confusion while diversity sampling increases it. For VAEs, diversity sampling seems to decrease the confusion instead. In all cases, VAEGANs seems to fail at controlling the crates. The PCGRL agent has some control over the crates, but the most probable output for every input lies around the range 3–5 and 5–7 for deterministic and stochastic policies respectively. Crate control is easier for generative models since they can put crates and goals on the same tile. Overall, GANs with auxiliary targets have the best crate control.

For the solution length, most models are performing well but they tend to fail if the requested solution length is high as seen in Fig.10. Also, training with auxiliary targets without diversity sampling leads to the least solution-length control performance. Overall, GANs with diversity sampling and auxiliary targets have the least confusion over the solution-length target if we ignore the unplayable levels.

IX. CONCLUSION

In this paper, we conducted an experimental study on four recent methods for Sokoban level generation. We adapted bootstrapping conditional GANs [4] for Sokoban then applied its methodology to VAEs and VAEGANs. To improve the diversity, we proposed diversity sampling where the model

trains on batches containing diverse solutions. To increase the quality, we proposed training with auxiliary targets and sampling conditions from GMMs for generation. For PCGRL, we discussed a trick to speedup the training and For GPN, we discussed a modification to the reward function for games where the player cannot die. The results showed that uncontrollable PCGRL achieves superior quality and diversity with the main drawbacks being uncontrollability and the long training and generation time. When control is required, GANs, with diversity sampling and auxiliary targets, presented a good compromise between quality and diversity while also having good control over their output. We also showed that diversity sampling consistently increases the solution diversity while auxiliary targets and GMM condition sampling consistently improved the quality.

REFERENCES

- [1] S. Risi and J. Togelius, “Increasing generality in machine learning through procedural content generation,” *Nature Machine Intelligence*, pp. 1–9, 2019.
- [2] K. Compton. (2016, Feb.) So you want to build a generator.... [Online]. Available: <https://galaxykate0.tumblr.com/post/139774965871/so-you-want-to-build-a-generator>
- [3] A. Summerville, S. Snodgrass, M. J. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, “Procedural content generation via machine learning (pcgml),” *IEEE Transactions on Games*, vol. 10, pp. 257–270, 2018.
- [4] R. Rodríguez Torrado, A. Khalifa, M. Cerny Green, N. Justesen, S. Risi, and J. Togelius, “Bootstrapping conditional gans for video game level generation,” in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 41–48.

Fig. 10. Solution Length Confusion Matrices (DS: Diversity Sampling, AT: Auxiliary Targets)

- [5] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, pp. 95–101, Oct. 2020. [Online]. Available: <https://ojs.aaai.org/index.php/AIIDE/article/view/7416>
- [6] S. Earle, M. Edwards, A. Khalifa, P. Bontrager, and J. Togelius, "Learning Controllable Content Generators," in *2021 IEEE Conference on Games (CoG)*, 2021.
- [7] P. Bontrager and J. Togelius, "Learning to Generate Levels From Nothing," in *2021 IEEE Conference on Games (CoG)*, 2021.
- [8] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2014. [Online]. Available: <http://arxiv.org/abs/1312.6114>
- [9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: <https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afcf3-Paper.pdf>
- [10] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," *ArXiv*, vol. abs/1512.09300, 2016.
- [11] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, vol. 33, no. 1, pp. 19–37, oct 2021. [Online]. Available: <https://link.springer.com/article/10.1007/s00521-020-05383-8>
- [12] B. De Kegel and M. Haahr, "Procedural puzzle generation: A survey," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 21–40, 2020.
- [13] A. Sarkar, Z. Yang, and S. Cooper, "Conditional level generation and game blending," *ArXiv*, vol. abs/2010.07735, 2020.
- [14] M. Suleman, F. H. Syed, T. Q. Syed, S. Arfeen, S. I. Behlim, and B. Mirza, "Generation of sokoban stages using recurrent neural networks," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 3, 2017. [Online]. Available: <http://dx.doi.org/10.14569/IJACSA.2017.080364>
- [15] J. H. Lim and J. C. Ye, "Geometric gan," *ArXiv*, vol. abs/1705.02894, 2017.
- [16] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier GANs," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 2642–2651. [Online]. Available: <https://proceedings.mlr.press/v70/odena17a.html>
- [17] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," in *36th International Conference on Machine Learning, ICML 2019*, 2019.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
- [19] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.
- [20] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSE: Neural Networks for Machine Learning, 2012.
- [21] S. Roberts, D. Husmeier, I. Rezek, and W. Penny, "Bayesian approaches to gaussian mixture modeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 11, pp. 1133–1142, 1998.
- [22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *ArXiv*, vol. abs/1707.06347, 2017.
- [23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1412.6980>