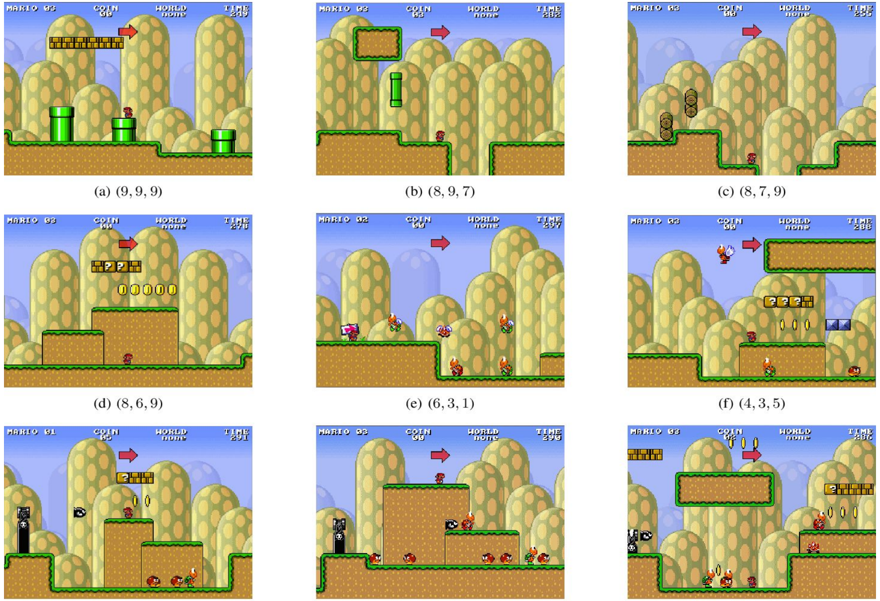
AI60201: Graphical And Generative Models For Machine Learning

# VGL-GAN: Video Game Level Generation using **Deep Convolutional Generative Adversarial Network**

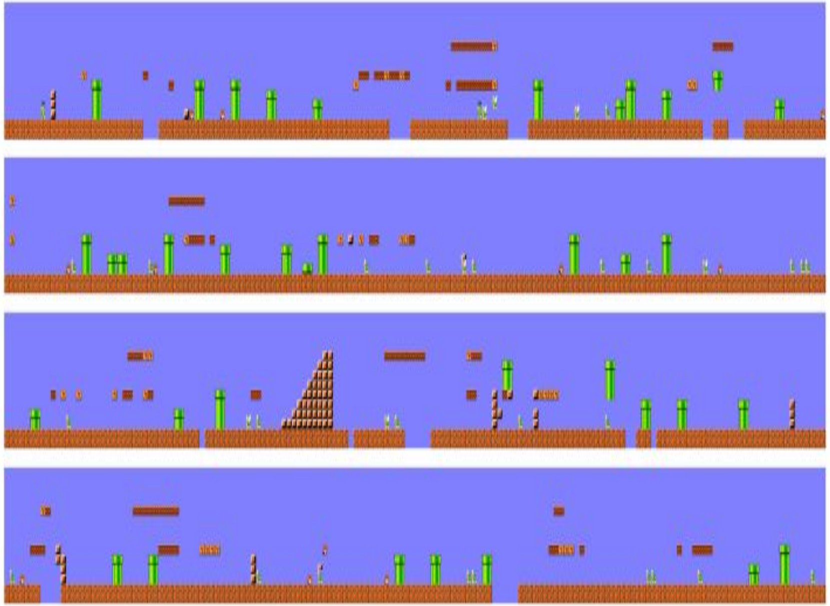
Abhinav Bohra, Maitreyi Swaroop, Mihir Shrivastava, Suryansh Kumar

# Introduction

## Procedural Content Generation

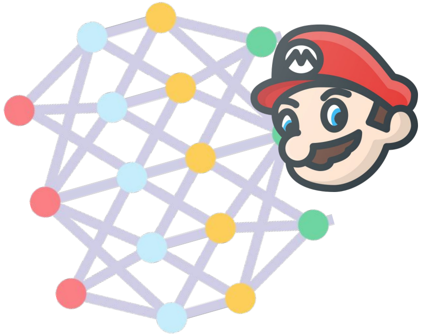


(a) (9, 9, 9)
(b) (8, 9, 7)
(c) (8, 7, 9)
(d) (8, 6, 9)
(e) (6, 3, 1)
(f) (4, 3, 5)
(g) (1, 1, 4)
(h) (6, 5, 7)
(i) (3, 3, 4)



Super Mario Levels generated by AI powered models

# Problem Statement

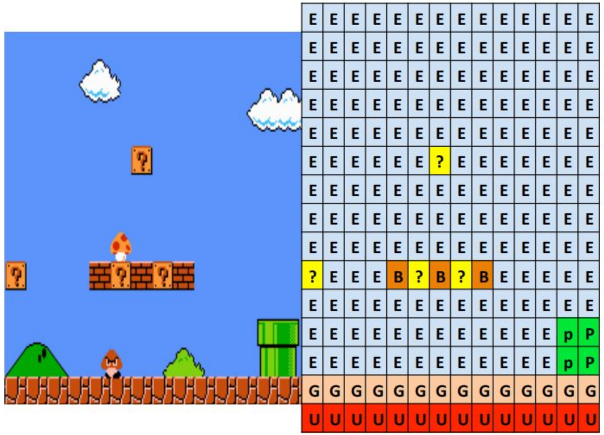


Explore and evaluate alternative GAN architectures applied to the creation of **playable** game levels

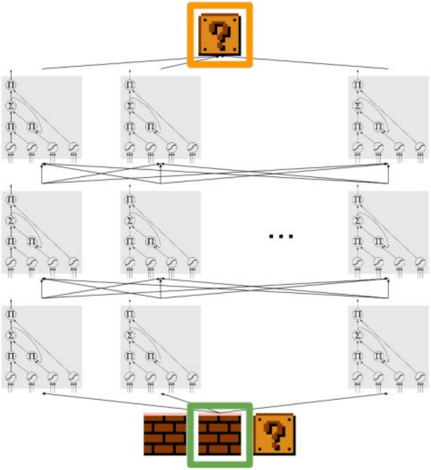
Compare **latent space search techniques** to optimise inputs to GAN from within its latent space

# Related Works

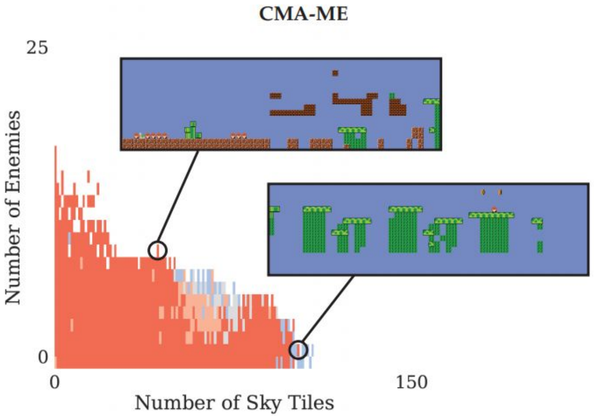
### Map Generation using Markov Chains[1]




### Level Generation Via LSTMs[2]



### GANs & Latent Space Illumination[3]

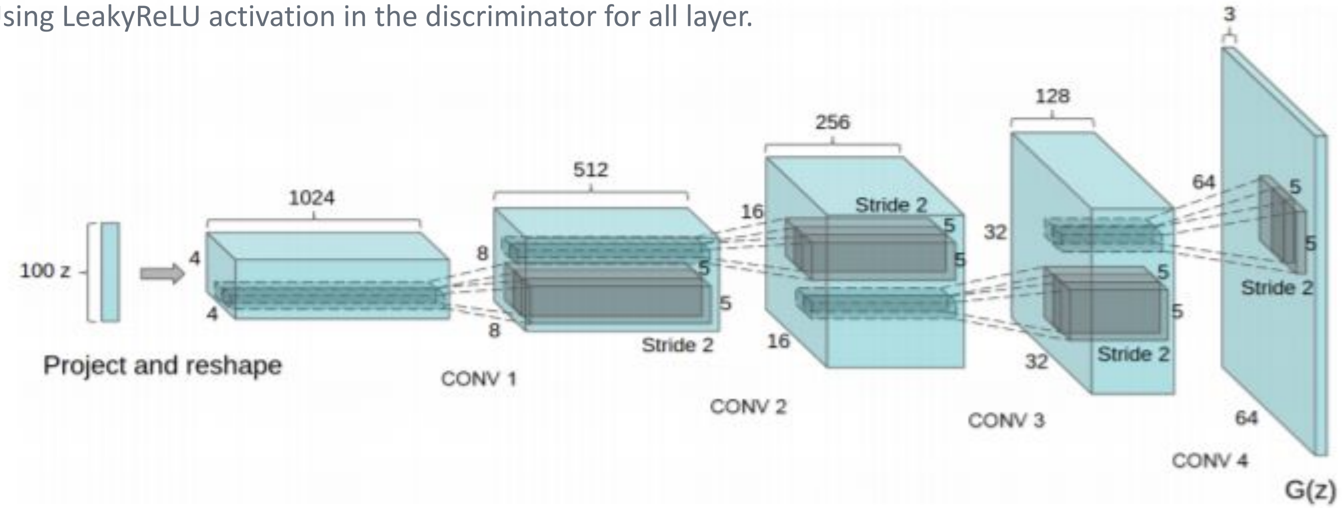




[1] Snodgrass, S. and Ontañón, S., 2014. Experiments in map generation using Markov chains. In FDG
[2] Summerville, A. and Mateas, M., 2016. Super mario as a string: Platformer level generation via lstms
[3] Fontaine, M.C et al 2021, May. Illuminating mario scenes in the latent space of a generative adversarial network. In Proceedings of the AAAI Conference on AI

# DC-GAN

- Replacing any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Using batchnorm in both the generator and the discriminator.
- Removing fully connected hidden layers for deeper architectures.
- Using ReLU activation in generator for all layers except for the output, which uses tanh.
- Using LeakyReLU activation in the discriminator for all layer.

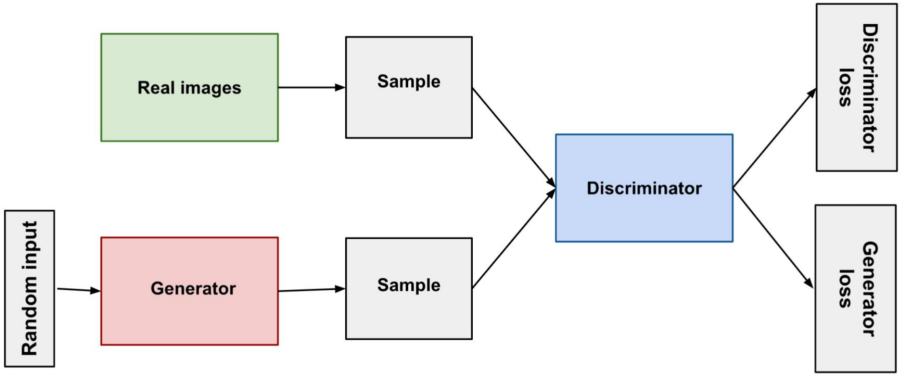



**DC-GAN Generator Model Architecture[4]**

[4] Radford, A., Metz, L. and Chintala, S., 2015. Unsupervised representation learning with deep convolutional generative adversarial networks

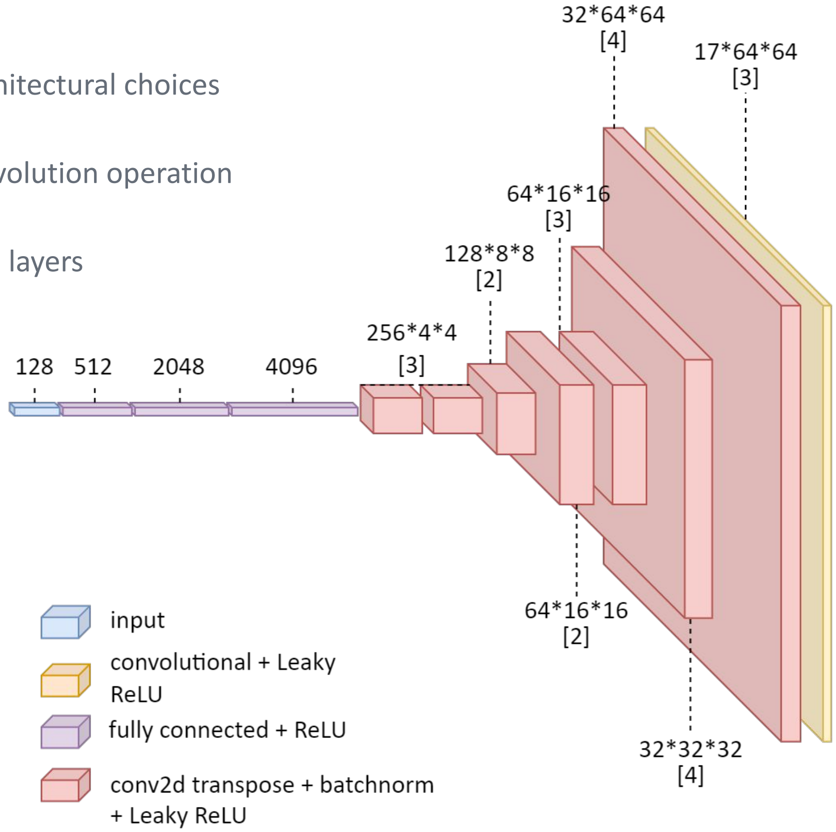
# VGL-DCGAN (Proposed Model)

- Built using PyTorch framework.
- Architecture based on Deep Convolutional Networks
- Uses randomly initialized input vectors to synthesize levels using the generator.
- Discriminator tries to distinguish between synthesized and actual levels.
- Model parameters updated through Wasserstein Loss.
- Model has to be robust so as to prevent mode collapse and produce diverse levels.
- Playability is an important factor, overly elaborate levels need to be suppressed.



# VGL-DCGAN (Generator Architecture)

- The generator has certain unique quirks to its design.
- Mode collapse is a common problem for GANs, certain architectural choices can help alleviate this issue.
- We use three dense, fully connected layers before any convolution operation which helps in increasing the sparsity of the input
- Cold-starting is also used where first two conv2d transpose layers do not up-sample the input.
- Adding these features prevented mode collapse.



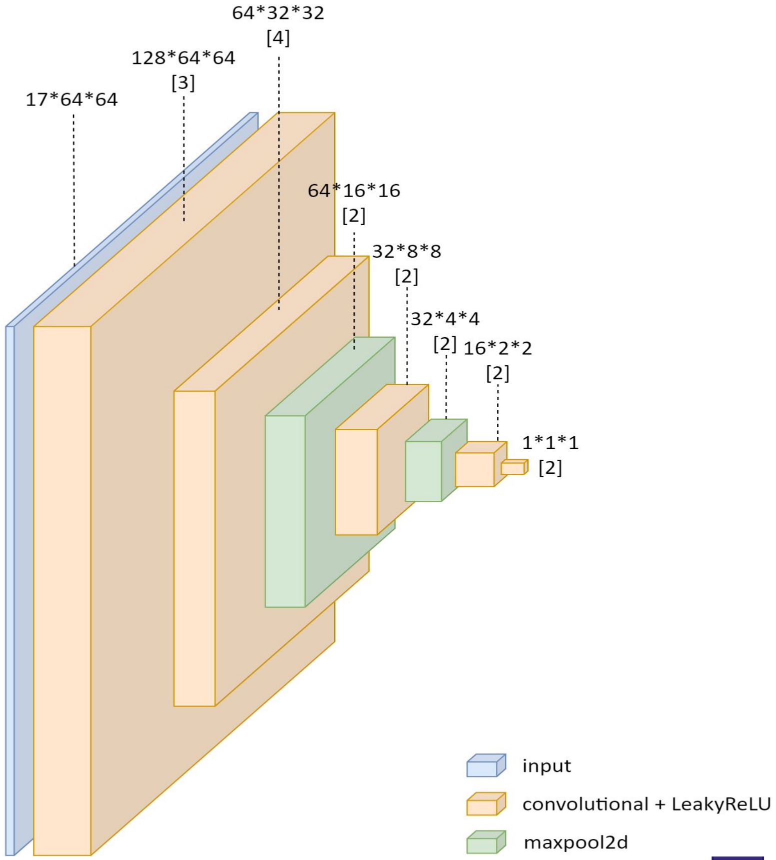
# VGL-DCGAN (Mode Collapse)


Mode collapsed output


Output after adding dense layers and cold starting

# VGL-DCGAN (Discriminator Architecture)

- The discriminator uses a simple architecture with convolution2d layers and max pooling.
- It takes a 17-channel input due to there being 17 unique tiles.



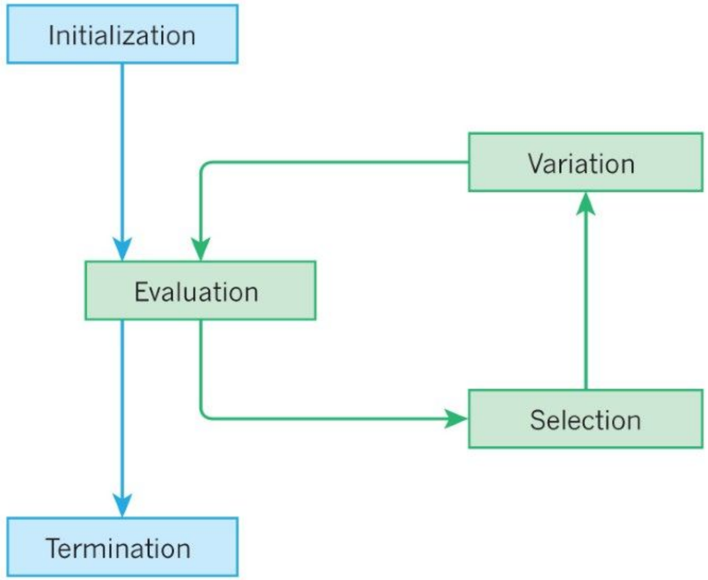
# VGL-DCGAN Training Details

- We used an Adam optimizer with a Wasserstein loss metric.
- The value for the learning rate and the beta parameters were fine-tuned through extensive experimentation.
- Model was trained for 2500 epochs using a batch size of 32.
- The both the discriminator and generator losses exhibited sustained variation across at certain points during training, indicating overcoming of mode collapse.

| Component | Learning Rate | beta -1 | beta-2 |
| --- | --- | --- | --- |
| Generator | 0.0001 | 0.65 | 0.99 |
| Discriminator | 0.00005 | 0.85 | 0.999 |

# Evolutionary Latent Space Search Techniques

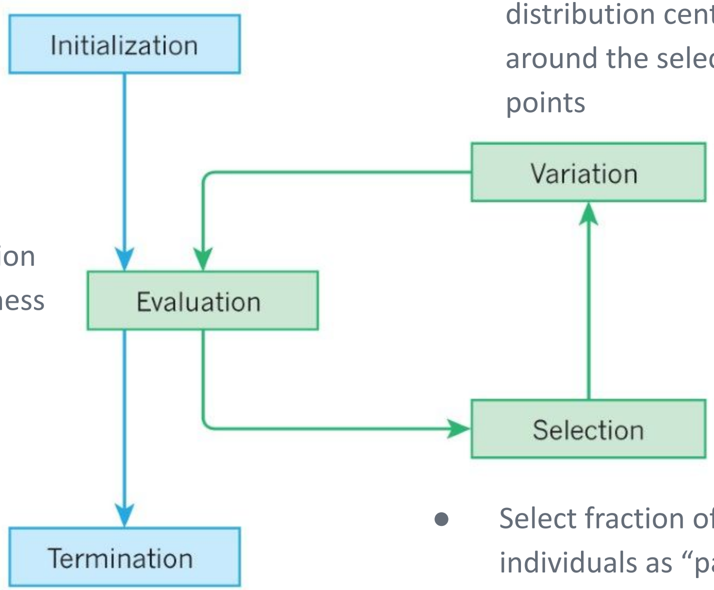
- Genetic Algorithms  and Evolutionary Search Strategies for optimisation
- Generate a seed population
- Define fitness function to evaluate individuals
- Rules for evolution of the population



# Guided Random Search

- Gradient free, randomised search
- Needs a large population for successful search

- Assume initial distribution parameters to be standard normal
- Draw initial population




- Sort population based on fitness value.
- Draw new population from distribution centered around the selected fittest points
- Select fraction of fittest individuals as “parents”

# Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

- Search over non-smooth, non-convex solution space
- Gradient free method
- Update both mean vector and covariance matrix
- Update mean - to focus search to points closer to already known good solutions
- Update covariance matrix - want the covariance matrix to approximate the contours of the fitness function
- Analogy to quasi-Newton methods and finding inverse of the Hessian
- Requires smaller population than random search

**Algorithm 1** CMA-ES Algorithm

`Define:` evolutionary search parameters $\lambda, \mu, \kappa_1, \kappa_2, \sigma, c_\mu$. `Intialise:`the mean vector $\mathbf{m}^{(0)}$ and covariance matrix $\boldsymbol{C}^{(0)} = \boldsymbol{I}$. $g \leftarrow 1$

**while** not converged **do**

Sample search points $\{\mathbf{x}_k^{(g+1)}\}_{k=1}^{\lambda}$ for the given generation $g$.

$$z_k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$\mathbf{y}_k = z_k \sim \mathcal{N}(\mathbf{0}, \boldsymbol{C})$$

$$\mathbf{x}_k^{(g+1)} = \mathbf{m} + \sigma \mathbf{y}_k \sim \mathcal{N}(\mathbf{m}^{(g)}, (\sigma^{(g)})^2 \boldsymbol{C}^{(g)})$$

$$k = 1, ..., \lambda$$

(Here $\boldsymbol{B}, \boldsymbol{D}$ are obtained from the eigendecomposition of $\boldsymbol{C}$ by Spectral Theorem).

Select $\mu$ points from the sample, in decreasing order of fitness (i.e. increasing order of objective function) as $\{\hat{\mathbf{x}}\}_{i=1}^{\mu}$. Here $f(\hat{\mathbf{x}}_1) \leq f(\hat{\mathbf{x}}_2) \cdots \leq f(\hat{\mathbf{x}}_\mu)$ Update the mean of the search distribution

$$\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + \gamma \sum_{i=1}^{\mu} (w_i \hat{\mathbf{x}}_i^{(g+1)} - \mathbf{m}^{(g)})$$

where $\sum_{i=1}^{\mu} w_i = 1$, and $w_i > 0$ are arranged in non-increasing order. $\gamma$ is the learning rate.

Further, the covariance matrix is updated by:

$$\boldsymbol{C}_\mu^{(g+1)} = (1 + \kappa_1)\boldsymbol{C} + \kappa_2 + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\,\lambda} \mathbf{y}_{i:\,\lambda}^T$$

where

$g \leftarrow g + 1$

**end while**

# Covariance Matrix Adaptation MAP-Elites

- CMA-ME = CMA-ES + Directional Search among Map-Elites
- Behaviour Characteristics and Map-Elites
- Update both mean vector and covariance matrix according to CMA-ES rules
- Sample point from resultant distribution + add a vector in the direction of  a second elite

# Evaluation Techniques

The generated output should meet these two requirements:

- Levels must be new while having similar properties to existing human-generated game levels
    - Enforced by the objective function characteristic of the GAN architecture
- Levels should be playable by a human
    - Ideally, this will be answered by human annotators, but here that is infeasible,
    - So, we use standard methods such as playing against an A* agent.

The search techniques against which we evaluate our model include:

- Stochastic Hill Climbing
- CMA-ES
- CMA-ME

# Experiments: Data sources

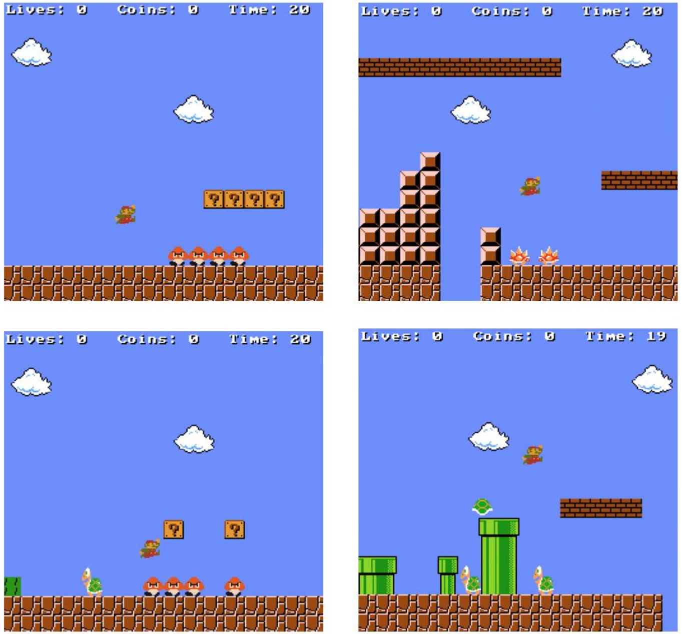
**Mario-AI-Framework[4] -** an open source repository that helps with the seamless deployment of AI models with a version of Super Mario Bros.

The framework includes:

- Several AI planning agents
- Java applet that converts the ASCII output of the generator into playable levels

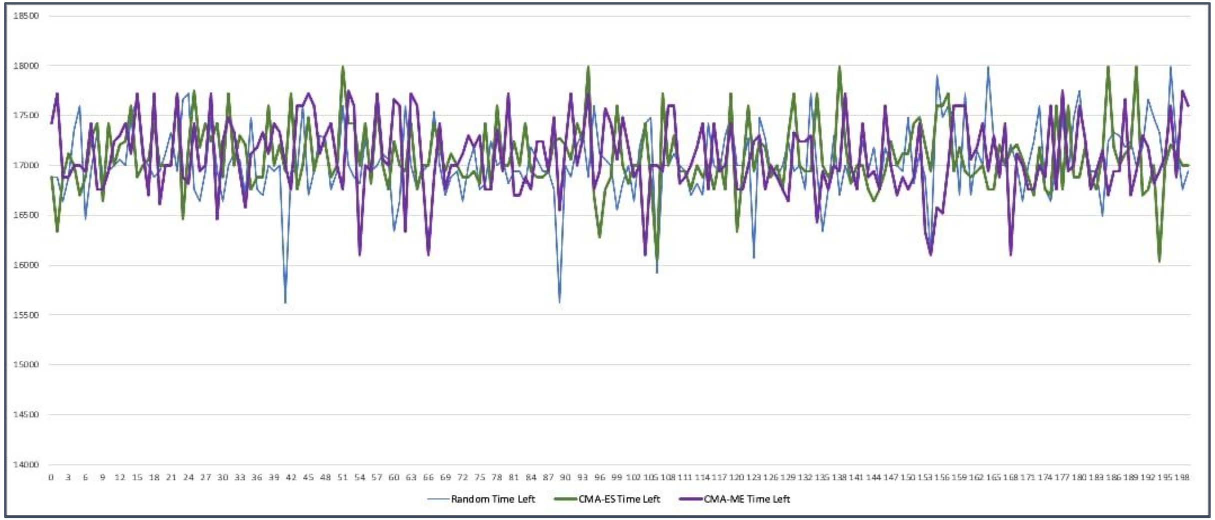
The framework provides us with:

- A straightforward mechanism of running levels generated by our model
- Feedback from the agent that includes relevant parameters like level completion status, number of jumps, etc.




[4] https://github.com/amidos2006/Mario-AI-Framework

# Experiments: Results




- Common measure of playability:
  - the amount of time left after the A* agent is done playing the game
  - any non-zero value indicates that the agent was able to complete the level and the higher the value the easier the level is to finish

- Observation: all three of the search methods behave similarly over the iterations when searching for playable levels

# Conclusion

- Built a framework for generating human-playable levels for Super Mario Bros. using Deep Convolutional Generative Networks enhanced by latent space exploration techniques.
- Deployed a DCGAN model, different from the earlier implementations that succeeded in efficiently generating new levels for the game.
- Applied different latent search exploration methods with the aim of enhancing the quality of generated levels by finding an optimized latent vector for querying to the generator.
- Through rigorous experimentation, we arrived at a framework that is able to generate diverse and human-playable levels without them being extremely contrived and unplayable.

# Thank you!

**It's been a great learning experience**.

Abhinav Bohra, Maitreyi Swaroop, Mihir Shrivastava, Suryansh Kumar