Causal Scene Generation

Harish Ramani

Abstract

The project aims to procedurally generate 2D images where the causal model describes the relationship between the entities that generated the image. A causal generative model is built that understands the distribution of the images and thereby helping in reconstruction but also giving the capability to condition and intervene on the process that generated the image, thereby resulting in a new image for each conditioning and intervention queries.

1 Introduction

A causal model aims to represent a data generating process. There have been numerous attempts to model text to an image and it predominantly involves usage of a Generative Adversarial Network or GAN. [1] shows an existing GAN model which attempts to display an image that describes the text, however ineffectively. It doesn't capture the necessary relation between all the entities in the image. This is where a causal model would be highly beneficial, to capture the relationship between the different entities. To give an example, we could have a caption "*A man is walking towards a door*". If we have a causal model and a procedural image generation scheme, we could ask a counterfactual question, "*Had it been a woman*", what would the image look like? In this study, we are aiming to build a causal model that generates an image wherein we can condition and intervene on the causal model that describes the image generation process. Once this is developed, we can investigate as to whether this model is capable of addressing a counterfactual query. Later, textual captions that describe the image can be used along with the model developed here to produce counterfactual images from text.

2 Background

The current work is an extension of a project titled "*Causal Object Oriented Programming*" where, a proof-of-concept was created to answer probabilistic queries for different entities that are related using Pyro [2] This work utilized Avi Pfeffer's concept of Bayesian Object Oriented Programming[3] which was originally created in a programming language named Figaro using Scala. This work allowed us to represent real world entities and their relationship with other entities, probabilistically with the familiarity of Object-Oriented Programming. In this work, we considered a toy example found in [3] and reproduced the results in pyro. The example and the corresponding code base for this work could be found here [4]

3 Methods

The different components required to generate an image in this study are, a causal model that understands the relations between the entities that generated the image, a deep generative model that learns the aids in generating images and a procedural generation scheme to generate training samples for the generative model.

3.1 Procedural Generation

A procedural generation is an automated way of creating data. 2D images are automatically generated given the metadata about the image. In our example, we considered a universe of game characters [5] (currently two characters Satyr and Golem) doing a limited set of actions like Attack, Taunt, Walk etc. Each set of character has different variations where the armor, head color and the way they perform the actions varies. The procedural generation code creates different versions of the character which didn't exist in the original set.

HEAD HAIR TORSO WEAPON GENERATED CHARACTER

The above is an example of how a 2D image can be generated from different metadata. With the help of the above rendering mechanism, we can generate a scene by giving some information about the scene. We use a look up table to get to the corresponding images and merge them all together. Each image is of the same height and width. The procedural generation scheme is used to generate the training data needed for our causal model.

3.2 Causal Model

A causal model tries to capture the data generating process. The causal directed acyclic graph (DAG) [6] approach assumes that we will build a model that represents components of that system as a discrete set of variables. Those individual variables may be discrete or continuous. Their collective state represents a possible state of the overall data generating process. Further, we assume variables are causes/effects of other variables, and that these cause-effect relationships reflect true causality in the data generating process. All the variables in the DAG have a joint probability distribution and depending on the direction of the edge, we have a conditional probability distribution of the child node given the parent node.

Fig 1: A Directed acyclic graph with 3 variables.

For a set of three variables A, B, C all the below equations are acceptable as the joint probability distribution.

$$P(A, B, C) = P(A)P(B|A)\ P(C|A, B) \quad (1)$$

$$P(A, B, C) = P(B)P(B|A)\ P(C|A, B) \quad (2)$$

$$P(A, B, C) = P(C)P(A|C)\ P(B|A, C) \quad (3)$$

Algorithm 1 DAG Factorization

```
procedure DAG FACTORIZATION(DAG)
    for each root node x in DAG do
        return P(x)
    end for
    for each node y with parents in DAG do
        return P(y|parents(y))
    end for
end procedure
```

By using the above algorithm, the joint probability distribution using the DAG factorizes to,

$$P(A, B, C) = P(A)P(B|A)P(C|B) \quad (4)$$

The DAG encodes conditional independence assumptions and gives us a joint probability distribution that is different to the first three equations.

3.3 Conditioning and Intervention – Intuition

The real utility of building the causal models, under a probabilistic programming paradigm, is the ability to condition on observed data and infer on the latent factors that might have produced the data. In conditioning, we are interested in the conditional probability of the node given some evidence. From equation 4, we could ask a conditioning query like P(A=a|C=c) where the node C is taking discrete set of values and we observe that the value is c and we want to estimate the conditional probability of the node A taking the value a.

In Intervention, we artificially assign a value of a node such that the effect of the parents is negated. This is called as an ideal intervention where the effects of a node's parents are negated or mutilated. If we intervene on node B of the DAG as described in Fig 1, we get a new mutilated graph where the directed edge between node A and node B is mutilated and we get new conditional probabilities called as intervention probability distribution. It is denoted by the do operator. If we try to get the probability distribution of node C by intervening on node B, it is denoted by P(C|do(B=b)). The main difference between conditioning and intervention on the nodes is that, in intervention, the effects of the parents is mutilated whereas it is not in conditioning.

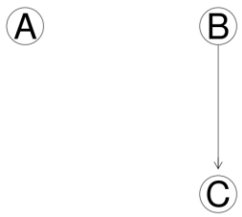


Fig 2: Intervention on Node B, mutilating the original DAG.

3.4 Variational Autoencoder

Autoencoder [7] is one such deep neural network architecture designed to learn representation for a set of data, typically in an unsupervised manner, by training by the network to ignore signal noise. In order to generate 2D images, we need a generative model that is able to understand the distribution of how the images are created. Variational Autoencoders [8] are directed probabilistic graphical models whose posteriors are approximated by a neural network with an auto encoder like architecture. The Autoencoder architecture comprises of an encoder unit, which reduces the large input space to a latent domain, usually of lower dimension than of input space and a decoder unit which reconstructs the input space from the latent representation as represented in Fig 3.

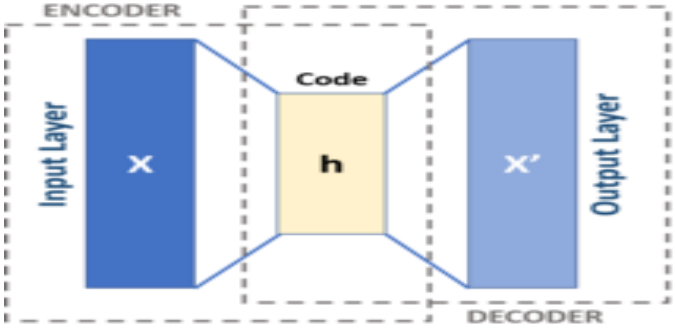
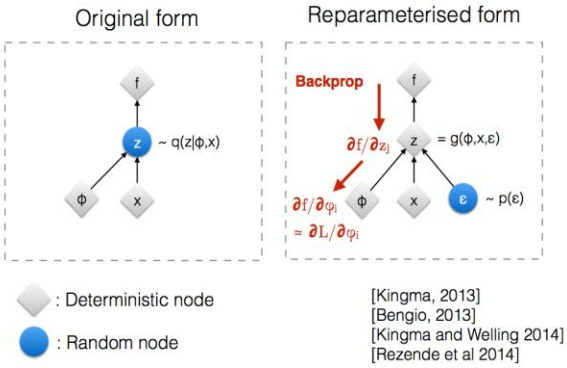


Fig 3: Autoencoder Architecture

After the encoder, there is a bottleneck layer which is represented as a latent representation of the data. Usually, this layer is represented by a Standard normal distribution. But this layer would be difficult to backpropagate as they



are stochastic. Hence, we use a reparameterization trick [9], where we introduce a new parameter ? so that it allows us to reparametrize our latent space z in a way that allows backpropagation to flow through deterministic nodes. In a variational autoencoder set up, the learned latent distribution can be sampled and along with a decoder can be used to generate new data points, in our case, images.

3.5 Statistical Motivation of Variational Inference

This section covers the statistical motivation behind the variational autoencoders and how it captures the distribution of the data that we are trying to model. Suppose there exists some latent variable z which generates an observation x as in Fig 2. In our case, X is our image and it can be observed whereas, the

latent variable isn't. Hence, we need to infer z from x which is basically the conditional distribution of z given x, P(Z|X). Using Bayes Rule,

$$p\left(z|x\right) = \frac{p\left(x|z\right)p\left(z\right)}{p\left(x\right)}$$

Unfortunately, computing p(x) is difficult and turns out to be an intractable distribution.

$$p\left(x\right) = \int p\left(x|z\right)p\left(z\right)dz$$

Fig4: Hidden Variable produces an observation

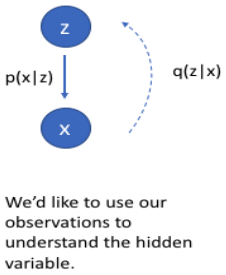
Hence, we use variational inference [10] to estimate this value. We do this by approximating p(z|x) by using another distribution q(z|x) such that it has a tractable distribution. If we define the parameters of q(z|x) such that it is very similar to p(z|x), we can use it to perform approximate inference of the intractable distribution. KL Divergence [11] is used to measure the difference between two probability distributions. Hence, if we wanted to ensure p(z|x) and q(z|x) are similar then we could minimize the KL Divergence between these two distributions.

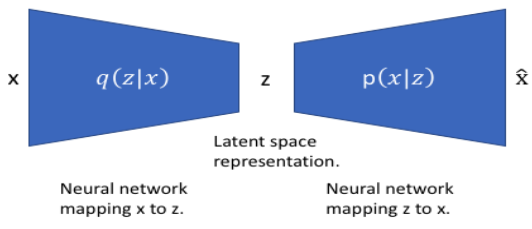
$$\min KL\left(q\left(z|x\right)||p\left(z|x\right)\right)$$

Minimizing the above equation is similar to maximizing the following.

$$E_{q(z|x)}\log p\left(x|z\right) - KL\left(q\left(z|x\right)||p\left(z\right)\right)$$

The first term represents the reconstruction likelihood and the second term ensures that the learned distribution q is similar to the true prior distribution p





We can use q to infer the possible latent state which was used to generate an observation. We can further construct this model into a neural network architecture where the encoder learns a mapping from x to z and the decoder learns the mapping from z back to x. Our loss function for this network will consist of two terms, one which penalizes reconstruction error and a second term which encourages our learned distribution q(z|x) to be similar to the true prior distribution p(z) which we will assume follows a unit gaussian distribution for each dimension j in the latent space.

$$\mathcal{L}(x, \hat{x}) + \sum_{j} KL(q_j(z|x) || p(z))$$

4 Implementation

To build a causal model, we decided to use the characters that appear in games. We chose two characters, Satyr and Golem and decided to give attributes like Strength, Defense and Attack that define their behavior. Each attribute can be either high or low. The causal model tries to represent what happens when these characters interact with each other. To model this, we first need the sample space of all the actions that they can do. The different activities that each character does are Walking, Taunting, Being Idle, Attacking, Getting Hurt if they're attacked and possibly die. Each Character has 3 different ways by which they can visually appear and they affect the character's attributes. All the different actions and the types that each character does correspond to a different image.

To make things simpler, each interaction has an actor, one who instigates or makes an action towards the other character based on his own attributes. The other character simply reacts to the action based on his own attributes. The probabilities in which each character is chosen and doing other actions are randomly assigned, for now, in the form of conditional probability tables and not learnt from data

4.1 DAG

The data generating process is encoded in the DAG. In this world of game characters, each image is composed of 2 characters. An actor and a reactor. Both of them are a random variable with equal chance of taking "Satyr" or "Golem" (the names of the characters). These characters can appear in various forms as shown in the below images. Their types obviously depend on their character and it influences their accessories, in terms of weapon of choice.

Fig 5: Three different types of Golem attacking.

Each character has individual attributes like strength, defense and attack which influences their actions. These attributes cannot be observed from image but can be reasoned about from a probability point of view. To make things simple for us, we define the probability of their individual attributes being either High or Low. We can think of it as a prior knowledge that we already know or assume it like it is. The Actor's attributes influence their action and the reactor's reaction is influenced by its attributes and the actor's action. The image is comprised of the actor and reactor's character, type and their respective action. The

action is a random variable with possible choices being Attack, Walk and Taunt. Reaction is also a random variable with possible choices being Hurt, Die, Walk and Attack.

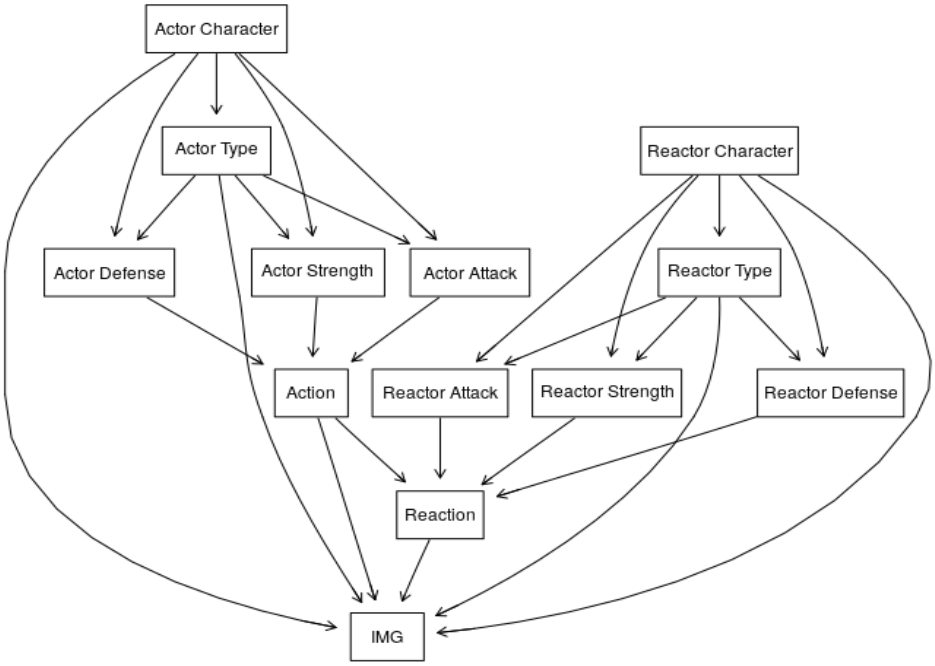


Fig6: Game Character Image DAG model.

4.2 Sample Image

We randomly sample the probabilistic model to get an actor's action and reactor's action along with the respective character and type. Based on this metadata, we draw an image of the scene using the procedural generation scheme as shown below.

Fig7: Golem is attacking Satyr and it got hurt in the process.

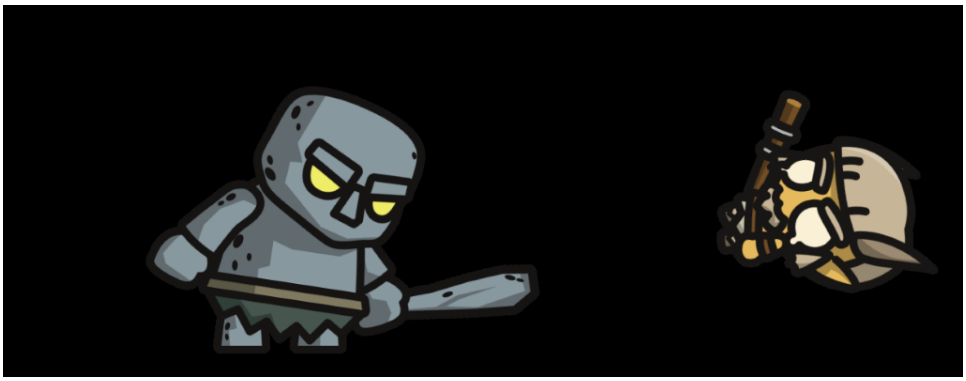


Fig8: Golem is attacking Satyr and it died in the process

4.3 Causal Variational Auto Encoder

In this section, we will see how the different parts, autoencoder architecture, stochastic variational inference optimizer, causal model via the DAG, are put together. The model is implemented using Pyro, a probabilistic programming language from Uber. Pyro works in conjunction with PyTorch, a deep learning framework. Hence, this programming language is the perfect choice, as of now, to implement models of these type. The goal of the causal variational auto encoder is to produce images like Fig 7 and Fig 8 when we condition and intervene few nodes in the DAG.

4.3.1 Pyro and its convention

All probabilistic programs are built up by composing primitive stochastic functions and deterministic computation. In our case, the data generating process is encoded into the DAG and is implemented to a stochastic function, conventionally named, model. In this stochastic function, we define each of the node in the DAG to be sampled from a specific distribution. In our case, all the nodes except the image are discrete variables and hence sampled from a categorical distribution. Since, we have training labels we can use them in conjunction while sampling from the nodes. If there's any learnable parameters, as in our case the encoder and decoder neural networks, we need another stochastic function named guide to help learn these parameters. Inference algorithms in pyro, such as stochastic variational inference [12], use the guide functions as approximate posterior distributions. Guide [13] functions must satisfy two criteria to be valid approximations of the model. One, all the unobserved sample statements that appear in the model must appear in the guide. Second, the guide has the same signature as that of the model, i.e. it takes the same arguments.

4.3.2 Conditional Probability Query from gRain Package

The nodes of the DAG are sampled from a categorical distribution as each node apart from the image are discrete variables. We need a conditional probability table so that we can use them in our sample statements in our model and guide. From the DAG mentioned in Fig 6, if we factorize the DAG, we get the joint probability distribution to be

*P(Actor Character) * P(Actor Type | Actor Character) * P (Actor Strength | Actor Character, Actor Type) * P(Actor Defense | Actor Character, Actor Type) * P(Actor Attack | Actor Character, Actor Type) * P(Reactor Character) * P(Reactor Type |Reactor Character) * P(Reactor Strength | Reactor Character, Reactor Type) * P(Reactor Defense | Reactor Character, Reactor Type) * P(Reactor Attack | Reactor*

Character, Reactor Type) P(Action | Actor Defense, Actor Strength, Actor Attack) * P(Reaction | Reactor Strength, Reactor Defense, Reactor Attack, Action) * P(Image | Action, Reaction, Actor Type, Reactor Type, Actor Character, Reactor Character)*

Apart from *P(Image | Action, Reaction, Actor Type, Reactor Type, Actor Character, Reactor Character)*, we can reason about the rest of the probability distributions and assume a prior conditional probability distribution. We make use of R package bnlearn [14] to create the DAG with assumed prior distributions. We then make use of the gRain [15] package to query the created DAG to infer about the probabilities of unobserved entities (from image) *Actor Strength, Actor Defense, Actor Attack, Reactor Strength, Reactor Defense and Reactor Attack*. This would be particularly useful in our guide function as these entities won't be observed in our model. From the DAG, the probabilities we find using gRain are

P(Actor Strength | Actor Type, Actor Character, Action), P(Actor Defense| Actor Type, Actor Character, Action), P(Actor Attack| Actor Type, Actor Character, Action), P(Reactor Strength | Reactor Type, Reactor Character, Reaction), P(Reactor Defense | Reactor Type, Reactor Character, Reaction), P(Reactor Attack | Reactor Type, Reactor Character, Reaction)

4.3.3 Data Pre-Processing

Using our procedural image generation scheme, we produce all the different possible combinations of images (2 characters with 3 types each and the actor having 3 actions and the reactor having 4 reactions to choose from) We produce a total of 432 different images. The data pre-processing step includes resizing the image to 400*400 pixels and normalizing the RGB values by dividing each pixel by 255.

4.3.4 Encoder and Decoder Network Architecture

The variational autoencoder, as explained in earlier sections, consists of an encoder which converts an image to latent space and a decoder which converts the latent space to an image. Since, we are working with images we need convolution layers in the encoder to capture the distribution of the data, and using convolution neural networks for images is a well-established concept at this point in time [16]. The decoder model takes in an encoded representation of the image and also the observed labels to generate an image back.

```
VAE(
  (encoder): Encoder(
    (cnn): Sequential(
      (0): Conv2d(3, 8, kernel_size=(5, 5), stride=(2, 2))
      (1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
      (3): Conv2d(8, 16, kernel_size=(5, 5), stride=(2, 2))
      (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU()
      (6): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2))
      (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): ReLU()
      (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2))
      (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (11): ReLU()
      (12): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2))
      (13): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (14): ReLU()
      (15): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2))
      (16): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (17): ReLU()
      (18): Conv2d(256, 512, kernel_size=(2, 2), stride=(2, 2))
      (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (20): ReLU()
      (21): Conv2d(512, 1024, kernel_size=(2, 2), stride=(1, 1))
      (22): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (23): ReLU()
      (24): Flatten()
    )
```

Fig 9: Encoder CNN architecture

In the encoder architecture, we use multiple blocks of convolution, batch normalization and ReLU activation units. Each block works with certain image channels as we keep on increasing the channel depth and decreasing the image height and width. We represent a 400*400 image with 3 channels as a tensor with 1024 elements.

In the decoder architecture, we start with the 1024 element tensor and we apply convolution transpose to decrease the channel size and increase the height and width of the resulting image back to the size of the original image. The kernel and stride are chosen so that the image shapes are consistent.

```
(decoder): Decoder(
  (cnn_decoder): Sequential(
    (0): UnFlatten()
    (1): ConvTranspose2d(1024, 512, kernel_size=(7, 7), stride=(2, 2))
    (2): ReLU()
    (3): ConvTranspose2d(512, 256, kernel_size=(7, 7), stride=(2, 2))
    (4): ReLU()
    (5): ConvTranspose2d(256, 128, kernel_size=(7, 7), stride=(2, 2))
    (6): ReLU()
    (7): ConvTranspose2d(128, 64, kernel_size=(7, 7), stride=(2, 2))
    (8): ReLU()
    (9): ConvTranspose2d(64, 32, kernel_size=(7, 7), stride=(2, 2))
    (10): ReLU()
    (11): ConvTranspose2d(32, 16, kernel_size=(5, 5), stride=(2, 2))
    (12): ReLU()
    (13): ConvTranspose2d(16, 8, kernel_size=(13, 13), stride=(1, 1))
    (14): ReLU()
    (15): ConvTranspose2d(8, 4, kernel_size=(11, 11), stride=(1, 1))
    (16): ReLU()
    (17): ConvTranspose2d(4, 3, kernel_size=(2, 2), stride=(1, 1))
    (18): Sigmoid()
  )
```

Fig 10: Decoder CNN architecture

4.3.5 Final Model Overview

Causal Variational Autoencoder consists of a model and guide stochastic function where the model uses the decoder network to produce the image and the guide uses the encoder network to produce the latent representation that could have produced the image. In the training mode, the learnable weights of the encoder and decoder are fine-tuned through backpropagation by using the Adam optimizer. [17]

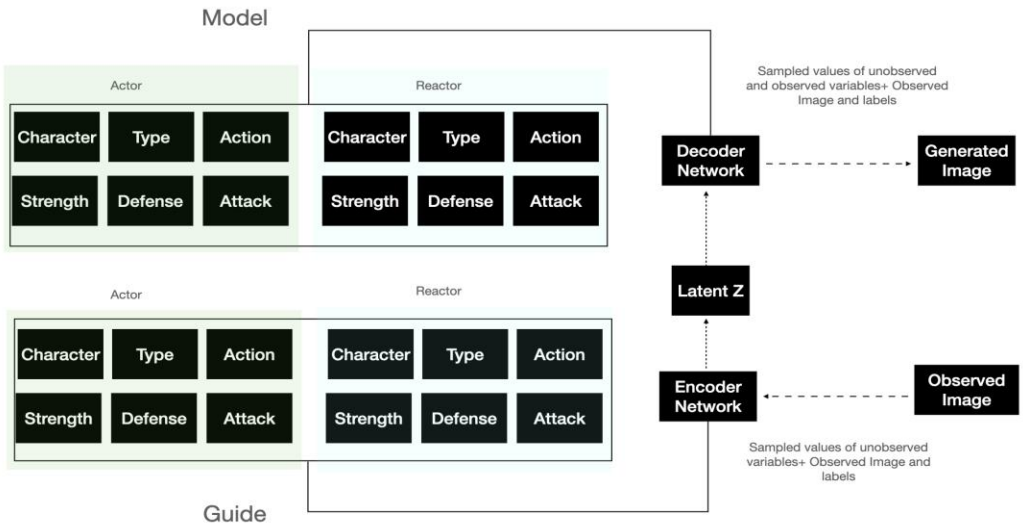


Fig 11: Model Overview during training mode

In Training mode, we use both the model and the guide. The model and the decoder work in conjunction whereas the guide and the encoder work together. In model, we sample all the nodes of the DAG conditioned on the labels and observed image. In guide we only sample the unobserved attributes of both actor and reactor like their strength, attack and defense by making use of the conditional probability distribution we got using gRain.

In inference mode, there's no need for a guide as the weights of the encoder and decoder are learnt during training. Hence, we can directly sample from the latent space and make certain condition and intervention queries on the model and pass both them both to the decoder to generate the image.

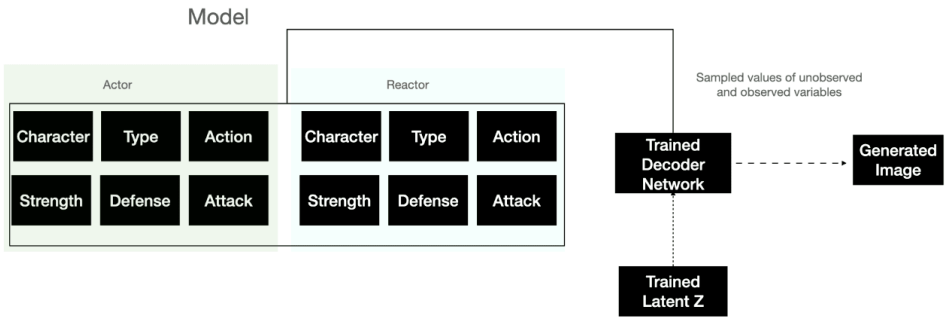


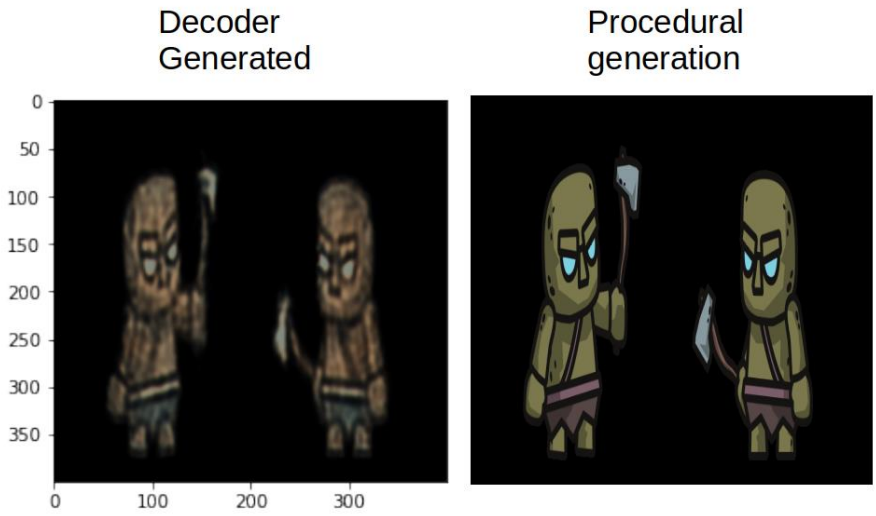
Fig 12: Model overview during inference mode

5 Results

The model is trained and the learnable weights are saved and now we use the inference model where we just use sample the nodes according to the conditional probability tables and generate an image. After generating the image from the decoder, other sampled values like the actor character, type, action etc. are also obtained and given to the procedural generation scheme to get the original image.

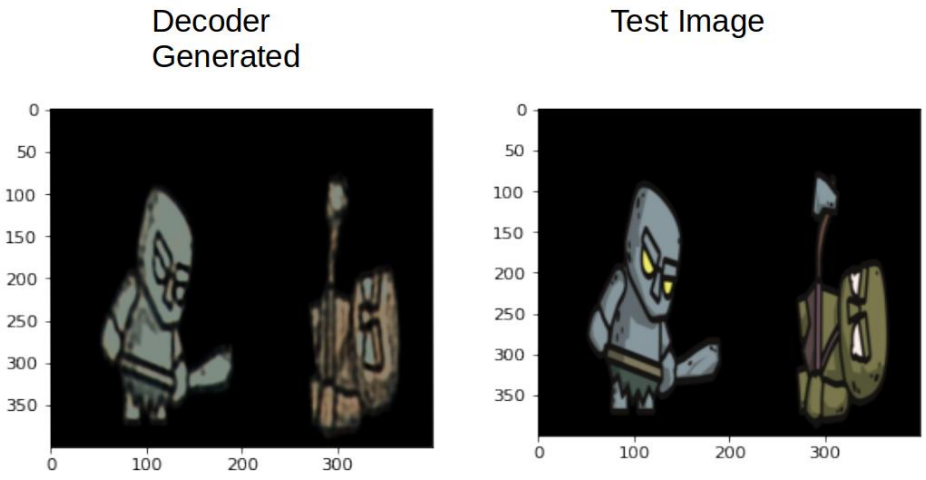
5.1 Random Sample from causal model

In this experiment, we don't condition or intervene any nodes and let the inference model sample and produce an image.



5.2 Reconstructing Test Images

In this experiment, we test out the model's reconstruction capabilities. We use both the encoder and decoder here as we have the image and the label already.



5.3 Condition Query

In this experiment, we condition on few nodes and generate an image from the decoder.

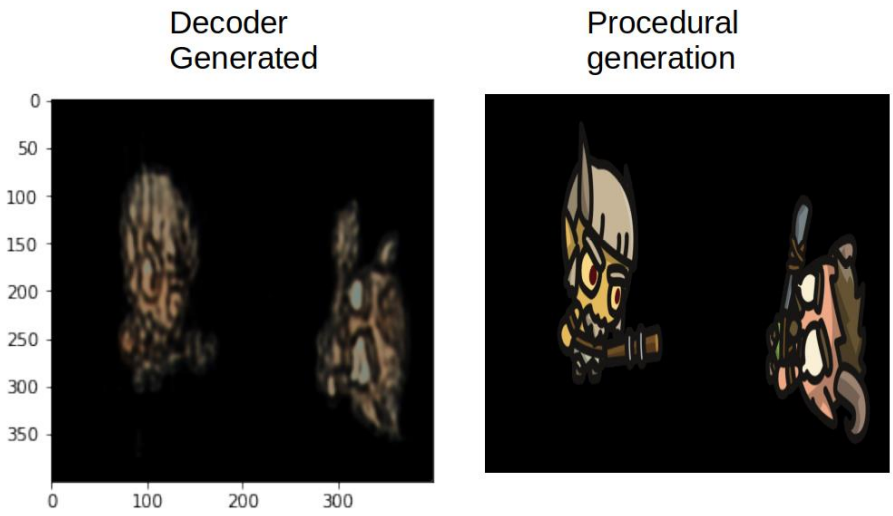


Fig 13: Reactor Reaction conditioned on Dying.

We conditioned on the reactor dying and the most probable action to cause this effect is for the actor to attack according to our conditional probability distribution.

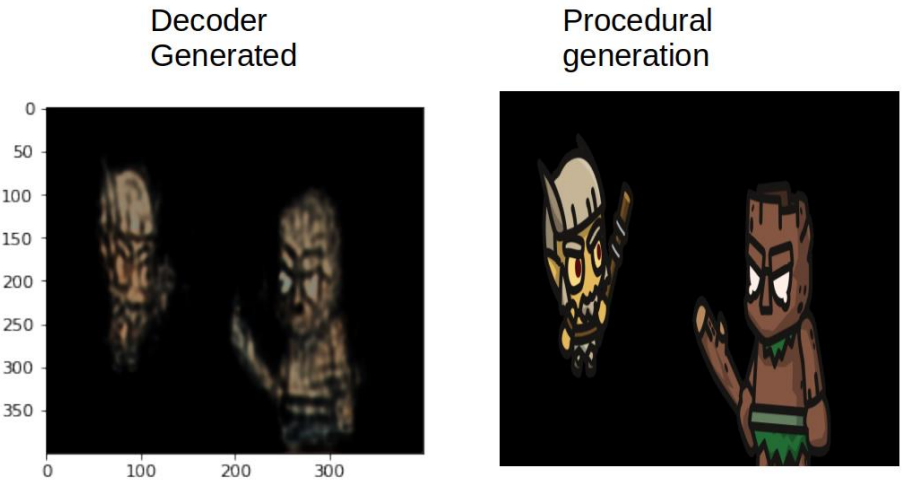


Fig 14: Actor conditioned to Satyr Character and Reactor to Golem

In the above image, we conditioned the actor node to take the satyr character and the reactor character to take the Golem character.

5.4 Intervention Query

In this model, we test out the model's intervention capabilities. By intervening the nodes, we mutilate the DAG so that the parent's effect is negated. The actor has initially a low chance of not causing a death when attacking and the reactor has a high chance of not dying when getting attacked. But by intervening on the actor's attack to be High and the reactor's defense to be Low, we severe the effects the type and character on the action and reaction and we observe that the reactor dies after the actor attacked.

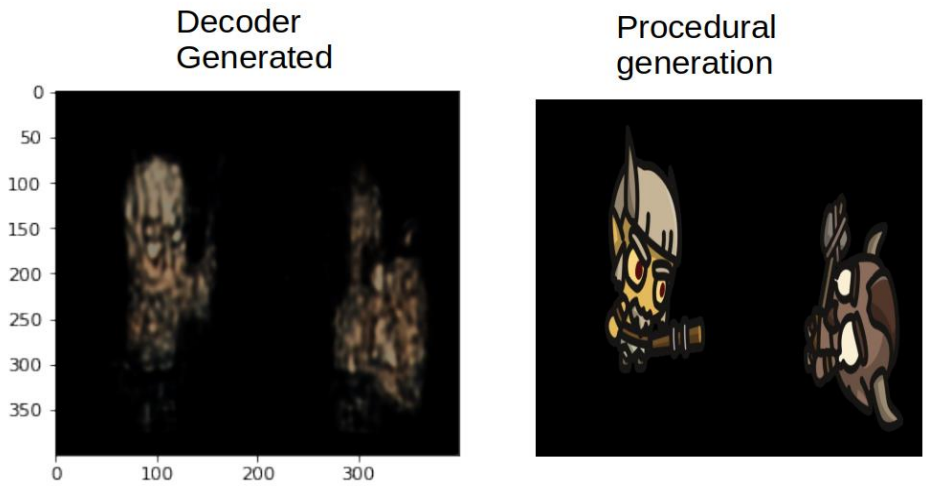


Fig 15: Intervention on Actor Attacking and Reactor's Defense.

6 Discussion and Future Work

We have presented a causal variational autoencoder to produce a 2D image and demonstrated that conditioning and intervention works for the above-mentioned cases. A known issue with variational autoencoder is that the quality of the images is very poor, as seen from above images, and the next step is to implement a Generative Adversarial Network (GAN) based architecture in place of the decoder network to produce high quality images.

Loss	Initial ELBO Loss	Final ELBO Loss	Num Epochs
Train	354007.8538	54203.3490	500
Loss	342033.2888	55323.4191	500

We see that the loss has reduced almost 84% during our training but it still hasn't captured some of the types of game characters. Some of the generated images are non-decipherable to the naked eye. A possible reason to this issue might be that variational auto encoder needs more training images to capture the distribution effectively but we used only around 360 images to train this model as the sample space for this DAG was intentionally limited. The final ELBO loss is still fairly high and more experiments needs to be conducted to investigate whether changing the encoder and decoder architecture helps to reduce the loss even more.

References

[1] https://experiments.runwayml.com/generative_engine/

[2] http://pyro.ai/examples/intro_part_i.html

[3] https://www.manning.com/books/practical-probabilistic-programming

[4] https://github.com/robertness/causalML/tree/master/projects/causal%20OOP/causal_oop_social_media/ppp_replication

[5] https://www.pinterest.com/pin/648518415079978646/

[6] https://en.wikipedia.org/wiki/Directed_acyclic_graph

[7] https://en.wikipedia.org/wiki/Autoencoder

[8] https://www.jeremyjordan.me/variational-autoencoders/

[9] https://towardsdatascience.com/reparameterization-trick-126062cfd3c3

[10] https://medium.com/@jonathan_hui/machine-learning-variational-inference-273d8e6480bb

[11] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

[12] http://pyro.ai/examples/intro_part_ii.html#Parametrized-Stochastic-Functions-and-Variational-Inference

[13] http://pyro.ai/examples/svi_part_i.html#Guide

[14] https://www.bnlearn.com/documentation/

[15] https://cran.r-project.org/web/packages/gRain/vignettes/grain-intro.pdf

[16] https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7

[17] https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c