4. Generative Model Based Simulators

4.1 Neural networks

Neural networks are the mathematical functions which maps an input value(s) to an output value(s). These functions are made of multiple perceptron which are arranged in layers and connected to each other.

4.1.1 Perceptron

A Perceptron is a linear binary classifier given by the following equation

Z = W.X+ B

Where w is the vector of weights w1, w2, w3…wn. x is the vector of inputs x1, x2, x3…xn. and b is the bias. The weights in the W signifies how much importance every input in vector x influences in the prediction. Figure 15 illustrates a neuron in perceptron with single output and multiple inputs.

4.1.2 Multi-layer perceptron

Neural networks is constructed by connection multiple layers of neurons in layers. The non liner function/ activation functions such as sigmoid, ReLU activation functions are used between the layers which maps the input to some non linear functions. Figure 16 shows the commonly used non linear activation functions. The process of calculating the output from this composite function for given input is called Forward Propagation.

Figure 16 Not linear activation function a) Sigmoid b) ReLU c) tanh

Neural networks are universal approximator which can approximate almost any continuous functions. Multiple layers of neuron and activation function makes it versatile to approximate complex functions. The weights and Bias associated with the neurons are initialized randomly and learned during the training. Figure 17 shows the architecture of a 2 layered Network.

Mathematically, the network function is represented as a proballistic function and the output of the model Y is given by

Y = p(y|x)

4.1.3 Loss functions

Loss function or cost function are the functions which calculates the deviation or difference between the original data point and the predicted data point. The metrics are used to train the network such that a network has to minimize this metric. Various loss functions are used depending upon the data and use case some of the popular loss functions are Mean square error, Entropy loss, KL divergence…

4.1.4 optimizers

Optimizers are the methods which are used to train the neural networks. The Network is trained by learning its parameters (weights and biases), where the cost function of the network is minimum. This is achieved by calculating the gradients of the parameters with respect to the loss function and updating the parameters by the value of step size in a way which its gradients shift in the direction of global minima using the equation

P\_new = p\_old − α∇L(p\_old)

Where ∇L(p\_old) is the gradient of loss function with respect to p\_old. Usually, in a multi layered network, the gradients of parameters are calculated using chain rule. The gradient equation of a parameter in layer n has the terms with gradients of parameters associated with neurons after it. α is the learning rate which signifies the rate at which the parameter update should happen. This rate is reduced over the training in some optimizer like ADAM (Adaptive Moment Estimation) to avoid skipping the minima. This entire process of updating the parameters of a network is called Back propagation. Various versions of optimization technique such as RMSprop, Adagrad, Adadelta are used in Neural network training.

4.1.4 Training

The process of training is iterating the backpropagation over the loop until this loss becomes low and stabilizes. The training data is segmented into batches making computationally effective and training faster. Backpropogation of all the batches in the training data marks an epoch. Various metrics such as accuracy are used to monitor the training. Moreover, a small set of data is subjected to validate the model which is not used for training.

4.1.5 Batch Normalization

Batch normalization is a technique in neural networks, improving training process by normalizing layer inputs within mini-batches. By calculating batch-wise mean and variance and normalizing inputs accordingly, it enables faster convergence, prevents vanishing or exploding gradients, and reduces the sensitivity to weight initialization.

4.2 Generative AI

Generative AI is the domain which uses the concept of artificial intelligence to create new instances of data. Initially, Various machine learning methods such as deep learning are used to classify or prediction. However, over the years these models are built to generate data. These Model are trained to learn the density of distributions of the training data effectively various types of architecture are used for various data distributions. For example, images, Convolutional neural networks effectively learn the pattern and features in the images. For data embedded with temporal information, Recurrent neural network is often used to learn the relations between various time steps. At once a model is trained to learn the distribution of the training data, new data points can be generated by sampling from the learned distribution. These generated are usually not in the training corpus and effectively trained generative models can generate data with features similar to training data. This method can be applied to many fields such as generating new images, music, text…

Based on the learning of density of data distribution, generative modelling can be classified in two types Implicit density explicit density

Explicit Density Models: These models explicitly learn and estimate the probability distribution of the input data. For example, Variational Autoencoders (VAEs) and traditional probabilistic models explicitly model the probability distribution of the data in a defined space. VAEs learn a probabilistic representation of the data, allowing them to generate new samples by sampling from this learned distribution.

Implicit Density Models: On the other hand, implicit density models do not explicitly define the probability distribution. Instead, they learn to generate new data points without directly modelling the probability distribution. Generative Adversarial Networks (GANs) are a prime example of implicit density models. GANs involve a generator network that learns to create samples without explicitly defining the underlying probability distribution. The generator aims to generate data that is indistinguishable from real data, but it doesn't directly model the probability distribution of the data.

The concept of Generative adversarial network was introduced in 2014 by goodfellow et.al has been a break through in the domain of Generative AI. GANs are initially used to generate images. Unlike VAE which generates blurry images, these GANs generate high quality images. However, these models suffer from various issues at the beginning such as Mode failure (inability to generate diverse data), instability in training… Later, various types of GANs had addressed this issue which made it a successful algorithm in the domain of Generative AI. Moreover, these models are generalized for various data making it versatile in this domain.

The ability of generating diverse and high-quality data has various application. In this proposed method these generated data are used to drive a simulator which trains autonomous vehicles. For effective training of a model which drives an autonomous vehicle, it has to trained on diverse scenarios. The existing simulator can offer a limited scenario for training which can be difficult for a model to generalize across diverse scenario. In the proposed method, the simulator is driven by a neural network model which is able to generate sensory data required by an autonomous vehicle. The driving model is trained on sensory data of diverse scenarios.

4.3 Data Generation

The Machine learning has 3 important paradigms called supervised, unsupervised and reinforcement learning. For training a ML model in supervised setup, training data is crucial. Especially generative deep learning model are data hungry and the performance of the model is proportional to the quantity of data on which the model is trained on.

For this research, transition of sensory data of autonomous vehicles with resect to time step along with the corresponding action which made the transition is required. This data is difficult to acquire from the available open source data sets in large quantity. In most of the dataset like [12],[13],[14],[15] the various sensory data captured at every time steps are available. However, the action label is missing.

In this research, the data is synthesized from simulator along with its action labels. These generated data is the based on the mathematical model of a 2d Lidar sensor on a vehicle driven in a closed environment. Each instances in the data consist of (Observation at timestep t, action label, Observation at timestep t+1 ) 150000 such datapoints are generated and used to train the generative model. The generative model is expected to generate new data points which are similar to one generated out of the simulator.

4.3.1 Mathematical model-based simulator

The simulator is developed in python. The environment is created using the floor plan of our university building. The wall in the plan are considered as lines within the simulator represented by it endpoint co-ordinates. The virtual vehicle which navigates with the environment is represented by a rectangular box according to the scale respect to the global environment boundaries and the change in position and orientation of the vehicle is simulated by updating the coordinates of the vehicle boundaries. The updating of the coordinates of the vehicle is done according to action commands. The action command is formulated as 3-dimensional vector where dimensions represent the velocity of the vehicle, turn angle of the vehicle and turn direction of the vehicle. The velocity of the vehicle is limited between -5 to 5 units and turn angle is restricted between 60 degrees on either side. The observation which mocks the 2d Lidar mounted on the vehicle is vector of size 360. Each entry in the vector accounts for the distance between the sensor and the closest distance of the obstacle around the vehicle in each angle (360 degree). This distance is calculated by projecting lines at each angle and this line is checked for intersection with all the wall with in the environment. The distance between the vehicle and the closest intersection point is assigned for that angle. The intersection points are checked by Cramer rule using the equations of angle line and wall line. This process is repeated for all 360 degrees and the observation is obtained. Figure 17 shows the vehicle within the simulator along with its projection line and wall.

The sensor data calculated at consecutive two steps are stored as a tuple along with its action command. Each data point in represented as follows

(Ot, At, Ot+1)

Figure 18 show the change in data for two consecutive timesteps. Noises are injected in the data to make it robust and the vehicle is initialized at new location whenever it collides or crosses the wall. Data points are logged into the training corpus at random timestep and shuffled. 150,000 datapoints are logged and used for further training the generative model.

4.4 Generative Adversarial Network

Generative adversarial networks typically composed of 2 models Generator G, Discriminator D. In Classical GANs, the generator G generate new instances y from the noise z sampled from a random distribution, the generated data is represented by

Y = G (z)

However, in this way the user has no control over the content of generated data. In this work a conditional GAN is used where the output of the generator is conditioned on the inputs (observation at previous time step Ot and Action label At). The Generator generates the observation for the next time step given previous timestep and action taken

O t+1 = G(Ot, At)

The discriminator outputs the conditional probability of the observation at next time step O t+1 for given Ot and At.

D (Ot, Ot+1, At) = P(Ot+1|ot,At)

4.4.1 Generator

The generator adopts the encoder-decoder architecture. The encoder takes the previous timestep sensor observation Ot and the action label At as input. The encoder encodes the observation data into 32-dimension latent code the action label is passed through a linear layer and the output is concatenated with every layer of the encoder. Figure 18 shows the architecture of the encoder. A random 32-dimension noise is sampled from the normal distribution is passed through a linear layer and concatenated with the latent code. This random noise account for the generation of new objects in the output.

4.4.2 Decoder

The decoder takes the latent code along with the random noise as input and outputs the observation of the next timestep Ogt+1. The decoder has 4 fully connected layers where each layer is concatenated with the action label and the output of corresponding encoder layer as shown in figure 18. By employing these skip connections, the information of the inputs are preserved after crossing the bottleneck. Each layer in the encoder and the decoder are followed by ReLU activation function and a batch normalization.

4.4.3 Discriminator

The discriminator is a fully connected network which takes 3 inputs : the observation at timestep t Ot, the observation at timestep t+1 and the action label. Both of the observations are forward propagating through the network. The action label is passed through a dense layer and the output is concatenated to every layer in the network. The output layer consists of single neuron with sigmoid activation function. The network outputs the probability of the given observation pair with respect to the action label is real. Moreover, this network evaluates the realness of the generated data.

4.4.4 Training loop

The training process of the GAN happens in two distinct steps individual training of the Generator and the discriminator. A batch of data is fetched from the training corpus. The observation at previous time step Ot and action label At are forward propagated in the generator, which outputs the observation of the next timestep Ogt+1. The output of the generator along with the inputs (Ot, At) is fed into the discriminator which further predicts the probability of the realness of Ot+1 with respect to Ot and At. The label of 0 is given to the generated observations representing that it is fake and the cross-entropy loss is computed between the o labels and the discriminator output. Furthermore, mean square error is computed between Generated observation Og t+1 and actual observation O t+1. The gradients of the generator are computed with respect to the combined weighted loss as in equation and the gradients of the generator are updated.

Generator loss, Lg = Ld+ alpha\* MSE

The discriminator is fed with the actual data (Ot, Ot+1, at) and the output is compared with label ‘1’ representing the real data. Furthermore, the sum of discriminator loss from both generated and real data is calculated and the gradients of the discriminator is calculated and updated with respect to this combined discriminator loss

Discriminator Loss = Ld real+ Ld Generated

During the training of the generator the weights of the discriminator are frozen and vice versa which prevents either of the component become overpower. For every epoch the training of generator and the discriminator are done alternatively. The ADAM optimizer is used which adapts the learning rate and stabilizes the training.

**CLEAN TEXT**

4. Generative Model Based Simulators

4.1 Neural Networks

Neural networks embody mathematical functions that establish a mapping between input and output values. Comprising multiple perceptron organized in interconnected layers, these functions form the backbone of complex computation.

4.1.1 Perceptron

A Perceptron stands as a linear binary classifier expressed through the equation:

\[Z = W \cdot X + B\]

Here, \(W\) denotes the weight vector \(w\_1, w\_2, w\_3, ... w\_n\), \(X\) signifies the input vector \(x\_1, x\_2, x\_3, ... x\_n\), and \(B\) represents the bias. The weights within \(W\) symbolize the significance of each input in vector \(X\) toward the prediction. Refer to Figure 15 for an illustration of a neuron in a perceptron with a single output and multiple inputs.

4.1.2 Multi-layer Perceptron

Neural networks are constructed by interconnecting multiple layers of neurons, integrating non-linear activation functions such as sigmoid and ReLU between these layers (Figure 16). The process of computing the output from this composite function for a given input is termed Forward Propagation.

Figure 16: Non-linear activation functions - a) Sigmoid b) ReLU c) tanh

Neural networks, as universal approximators, efficiently approximate nearly any continuous function. The incorporation of multiple layers of neurons and activation functions enables them with the ability to approximate intricate functions. The weights and biases associated with the neurons imitated randomly and is learned during training. Figure 17 illustrates a two-layered network).

Figure 17: Two layered fully connected network

Mathematically, the network function is represented as a probabilistic function, with the output \(Y\) expressed as \(p(y|x)\).

4.1.3 Loss Functions

Loss functions, also termed cost functions, evaluate the deviation between the original and predicted data points. These metrics guide network training towards minimizing this deviation. Popular loss functions include Mean Square Error, Entropy loss, and KL divergence. The loss function is selected based on the data and specific use case.

4.1.4 Optimizers

Optimizers represent the methods used in neural network training. They facilitate learning the network's parameters (weights and biases) to minimize the network's cost function. This is achieved by computing parameter gradients with respect to the loss function and updating these parameters by a step size in a manner that shifts their gradients towards the global minimum:

\[P\_{\text{new}} = P\_{\text{old}} - \alpha \nabla L(P\_{\text{old}})\]

Here, \(\nabla L(P\_{\text{old}})\) denotes the gradient of the loss function with respect to \(P\_{\text{old}}\). Typically, in multi-layered networks, parameter gradients are calculated using the chain rule. The learning rate (\(\alpha\)) signifies the speed of parameter updates, often diminished during training, as in ADAM (Adaptive Moment Estimation), to prevent skipping the minimum. This iterative parameter update process called as Backpropagation. Various optimization techniques such as RMSprop, Adagrad, and Adadelta are used in neural network training.

4.1.4 Training

The training process includes iteratively backpropagating through the network until the loss is minimized and stabilizes. Segmentation of the training data into batches enhances computational efficiency and expedites training. The backpropagation across all batches in the training data constitutes an epoch. Metrics like accuracy monitor the training progress. Furthermore, a small set of training data which are not utilized for training, is employed to validate the model during the training.

4.1.5 Batch Normalization

Batch normalization is a technique in neural networks, optimizing the training process by normalizing layer inputs within mini-batches. Computation of batch-wise mean and variance enables normalization of inputs, resulting in faster convergence, mitigating vanishing or exploding gradients, and reducing sensitivity to weight initialization.

4.2 Generative AI

Generative AI represents a domain uses artificial intelligence concepts to generate novel data instances. Initially, Various machine learning methods such as deep learning are used for classification and prediction. However, over the years these models have evolved to create data instances. Trained to learn the density of distributions within the training data, these models include various architectures tailored to specific data. For instance, Convolutional Neural Networks performs well in learning patterns and features within images, while Recurrent Neural Networks proficiently learn temporal relationships in data sequences.

Once the model learned the distribution of the training data, these models possess the capability to generate new data points by sampling from the learned distribution. These generated data points typically diverge from the training corpus, yet effectively trained generative models exhibit the ability to generate data resembling the features found in the training data. This versatile methodology spans diverse fields, including image, music, and text generation.

Generative modelling, founded on learning the density of data distribution, categorized into two categories: Implicit and Explicit Density Models.

Explicit Density Models:

Explicit Density Models explicitly learn and estimate the probability distribution of input data. Variational Autoencoders (VAEs) and traditional probabilistic models is a typical example of this approach by explicitly modelling the probability distribution within a defined space. VAEs acquire a probabilistic representation of the data, enabling the generation of new samples through sampling from this learned distribution.

Implicit Density Models:

In contrast, Implicit Density Models do not explicitly define the probability distribution. Instead, they focus on generating new data points without directly modelling the probability distribution. Generative Adversarial Networks (GANs) fall in this category. GANs employ a generator network tasked with creating samples without explicitly defining the underlying probability distribution. The generator's objective is to generate data indistinguishable from real data, without directly modelling of the probability distribution.

The introduction of Generative Adversarial Networks in 2014 by Goodfellow et al. marked a pivotal breakthrough in Generative AI. Initially applied for image generation, GANs stood out for producing high-quality images in contrast to the blurry images generated by VAEs. Early challenges such as Mode failure and training instability were addressed in subsequent GAN variants, solidifying its prominence in the domain. These models exhibit versatility across various data types such as videos, audios, signals…

The capacity to generate diverse and high-quality data finds various applications. In a proposed approach, these generated data drive a simulator used for training autonomous vehicles. Effective training demands exposure to diverse scenarios, which existing simulators might limit. In this proposed methodology, a neural network model drives the simulator, generating requisite sensory data for autonomous vehicle training. This approach facilitates training the driving model on diverse scenarios, augmenting the vehicle's ability to generalize across various environments.

4.3 Training Data

In the domain of machine learning, three fundamental paradigms stand out: supervised learning, unsupervised learning, and reinforcement learning. The training of a machine learning model, particularly in the domain of generative deep learning, heavily relies on data. The model's performance scales proportionally with the volume of training data on which it is trained on.

In this research, the synthesis of sensory data of autonomous vehicles alongside corresponding action labels proves challenging to obtain in substantial quantities from existing open-source datasets. While datasets like [12], [13], [14], and [15] provide comprehensive sensory data at each time step, they lack the corresponding action labels.

Addressing this challenge, this research synthesizes data derived from a simulator, incorporating action labels. The generated data is rooted in the mathematical model of a 2D Lidar sensor installed on a vehicle navigating within a confined environment. Each instance in the dataset comprises (Observation at time step \(t\), action label, Observation at time step \(t+1\)). A dataset consisting of 150,000 such data points is generated and used to train the generative model. The anticipated outcome is the generation of new data points closely resembling those produced within the simulator environment.

4.3.1 Mathematical Model-based Simulator

The simulator is constructed using Python, featuring an environment mapped to the floor plan of our university building. The walls in the floor plan are represented as lines within the simulator, defined by their endpoint coordinates. A virtual vehicle navigating this environment is represented by a rectangular box scaled in accordance with the global environment boundaries. Simulated changes in the vehicle's position and orientation are executed by updating the vehicle boundary coordinates based on action commands.

The action command is formulated as a three-dimensional vector which includes the vehicle's velocity, turn angle, and turn direction. Vehicle velocity ranges between -5 to 5 units, while turn angle confined within 60 degrees on either side. The simulated 2D Lidar, mimicking observations from the vehicle, comprises a 360-sized vector. Each vector entry signifies the distance between the sensor and the closest obstacle around the vehicle for a given angle (360 degrees). This distance calculation entails projecting lines at each angle and checking for intersection with the environment's walls. The Cramer rule facilitates identifying intersection points between angle lines and wall lines, enabling the determination of closest distances of the obstacles from the vehicle. This iterative process across 360 degrees yields the observation at given timestep. Refer to Figure 18 illustrating the vehicle, its projection lines, and the walls within the simulator.

Figure 18: Observation of the environment within the simulator

Successive sensor data calculated across two time steps are stored as a tuple alongside the corresponding action command, constituting each data point: \((O\_t, A\_t, O\_{t+1})\)

Figure 19 visualizes the data variation across two consecutive time steps. Gaussian noise are injected in the data to make it robust, and the vehicle initializes at a new location upon collision or crossing walls. Randomly logged data points across time steps are shuffled, culminating in the logging of 150,000 data points for further training the generative model.

Figure 19: Observations at two consecutive timesteps

4.4 Conditional Generative Adversarial Networks

In classical Generative Adversarial Networks (GANs), the typical setup involves two models: the Generator \(G\) and the Discriminator \(D\). Traditionally, the generator \(G\) produces new instances \(y\) based on noise \(z\) sampled from a random distribution, and the generated data is represented as \(Y = G(z)\). However, this approach lacks control over the content of the generated data.

In this work, a conditional GAN architecture is employed, altering the generator's output to be conditioned on specific inputs—namely, the observation at the previous time step \(O\_t\) and the action label \(A\_t\). The Generator, in this context, generates the observation for the next time step based on the given previous timestep and action: \(O\_{t+1} = G(O\_t, A\_t)\).

Conversely, the discriminator evaluates the conditional probability of the observation at the next time step \(O\_{t+1}\) given \(O\_t\) and \(A\_t\):

\[D(O\_t, O\_{t+1}, A\_t) = P(O\_{t+1} | O\_t, A\_t)\]

This conditional setup allows for more controlled generation of data, where the generator's output is influenced by specific inputs, leading to more tailored and context-aware generation.

4.4.1 Generator

The generator adopts an encoder-decoder architecture. Taking the previous timestep's sensor observation \(O\_t\) and action label \(A\_t\) as input, the encoder translates the observation data into a 32-dimensional latent code. Simultaneously, the action label is passed through a Dense layer, and its output is concatenated with every layer of the encoder (refer to Figure 20). Additionally, a random 32-dimensional noise sampled from a normal distribution is passed through a linear layer and concatnated with the latent code, introducing noise for the generation of new objects in the output.

4.4.2 Decoder

The decoder accepts the latent code and random noise as input, producing the observation for the next timestep \(O\_{gt+1}\) as output. Comprising four fully connected layers, each layer is concatenated with the action label and the output of the corresponding encoder layer (see Figure 20). By implementing these skip connections, information from the inputs is preserved after passing through the bottleneck. ReLU activation functions and batch normalization follow each layer in both the encoder and decoder.

4.4.3 Discriminator

The discriminator is a fully connected network processing three inputs: observation at timestep \(t\) \(O\_t\), observation at timestep \(t+1\) \(O\_{t+1}\), and the action label \(A\_t\),. Both observations propagate forward through the network, while the action label undergoes pass via a dense layer and is concatenated with every layer in the network. The output layer consists of a single neuron with a sigmoid activation function. This network outputs the probability that the given observation pair concerning the action label is real. Furthermore, the network evaluates the authenticity of generated data.

4.4.4 Training Loop

The GAN's training occurs in two distinct steps: the individual training of the Generator and the Discriminator. A batch of data is fetched from the training corpus. The observation at the previous time step \(O\_t\) and action label \(A\_t\) are propagated through the generator, resulting in the observation for the next timestep \(O\_{gt+1}\). This output, combined with generator inputs \((O\_t, A\_t)\), is then fed into the discriminator. The discriminator evaluates the probability of the authenticity of \(O\_{gt+1}\) for given \(O\_t\) and \(A\_t\). Generated observations are assigned a 'fake' label ('0'), prompting computation of cross-entropy loss between these labels and the discriminator output. Additionally, mean square error is calculated between the generated observation \(O\_{gt+1}\) and the actual observation \(O\_{t+1}\). The generator gradients are computed with respect to the combined weighted loss, as expressed in the equation, and the generator gradients are updated:

\[Generator\ loss, L\_g = L\_d + \alpha \times MSE\]

For the discriminator, actual data \((O\_t, O\_{t+1}, A\_t)\) is utilized, compared against the 'real' label ('1'). The sum of discriminator loss from both generated and actual data is calculated. Subsequently, the discriminator gradients are computed and updated concerning this combined discriminator loss:

\[Discriminator\ Loss = L\_d\ real + L\_d\ Generated\]

During generator training, discriminator weights are frozen, and vice versa, preventing either component from becoming overpowering. This alternating training of the generator and discriminator occurs for each epoch, facilitated by the use of the ADAM optimizer at adaptive learning rate