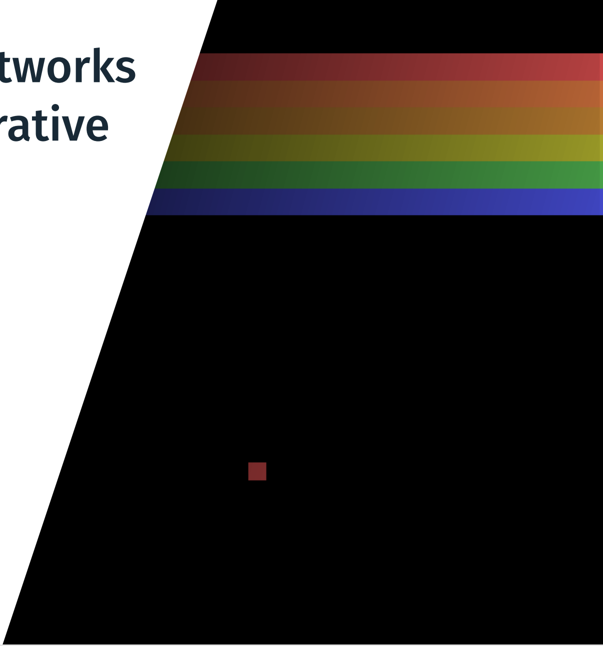
An Analysis of Deep Q-Networks and Applications of Generative Adversarial Networks in Reinforcement Learning

Adhokshaja V Madhwaraj
Prabuchandran K J

Indian Institute of Technology Dharwad
June 10, 2020



Introduction

What we are going to be looking at today

1. An introduction to the field of **Reinforcement Learning**
2. A **formal definition** of the problem we are trying to solve
3. **RL Algorithms** in the lead up to Deep Q-Networks
4. **Deep Q-Learning** and its variants
5. **Generative Adversarial Networks** and their uses
6. Applying GANs to RL - **Generating MDPs using GANs**

Introduction to Reinforcement Learning

Reinforcement Learning is Universal?

- Computer Science - We shall explore this further

Reinforcement Learning is Universal?

- Computer Science - We shall explore this further
- Neuroscience - How the brain takes decisions... and reacts to Rewards

Reinforcement Learning is Universal?

- Computer Science - We shall explore this further
- Neuroscience - How the brain takes decisions... and reacts to Rewards
- Psychology - Classical Conditioning

Reinforcement Learning is Universal?

- Computer Science - We shall explore this further
- Neuroscience - How the brain takes decisions... and reacts to Rewards
- Psychology - Classical Conditioning
- Economics - Game Theory

Reinforcement Learning is Universal?

- Computer Science - We shall explore this further
- Neuroscience - How the brain takes decisions... and reacts to Rewards
- Psychology - Classical Conditioning
- Economics - Game Theory
- Mathematics - Optimality and Operations

RL vs Supervised & Unsupervised Learning

RL has no direct *Supervisor*, but only a reward scheme which guides you in the correct direction.

RL vs Supervised & Unsupervised Learning

RL has no direct *Supervisor*, but only a reward scheme which guides you in the correct direction.

Feedback is not instantaneous, which makes predicting the future harder, thus picking actions harder.

RL vs Supervised & Unsupervised Learning

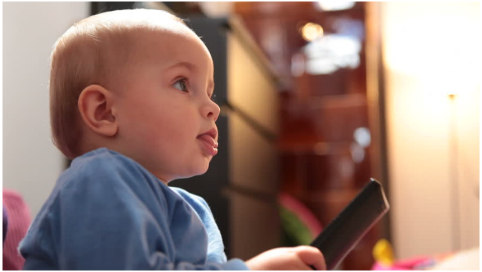
RL has no direct *Supervisor*, but only a reward scheme which guides you in the correct direction.

Feedback is not instantaneous, which makes predicting the future harder, thus picking actions harder.

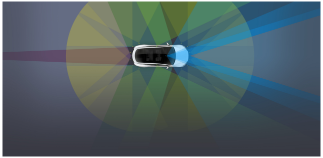
Highly correlated sequential data does not help, need i.i.d samples

An RL Example - Baby and the TV

- A baby learning to operate a TV
- An unresponsive TV with no colorful cartoons is a negative reward, while finding the channel with its favorite cartoon is the perfect positive reward.
- Actions leading to channels it likes, maximising the reward which is happiness here...



RL in the Real World



- Netflix A/B testing of alternate covers of movies
- AutoPilot features on new Tesla cars
- Autonomous training bots on the Stock Market

Formal Definition of the RL Problem

The Agent and the Environment

- The reward scheme, reward at time t is R_t

The Agent and the Environment

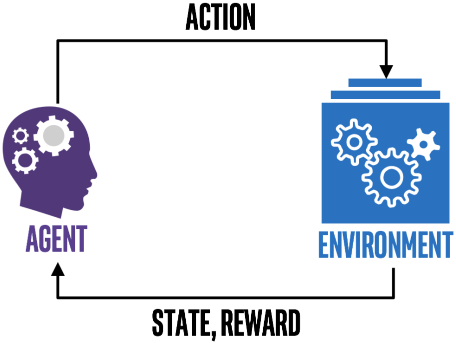
- The reward scheme, reward at time t is R_t
- Ultimate aim to maximise total rewards obtained from $t = [0, T]$

The Agent and the Environment

- The reward scheme, reward at time t is R_t
- Ultimate aim to maximise total rewards obtained from $t = [0, T]$
- The agent-environment interaction can be captured as a snapshot of the world at time t

The Agent and the Environment

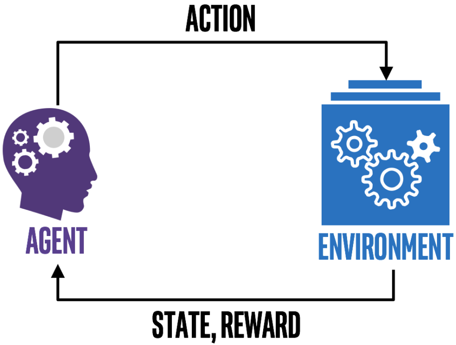
- The reward scheme, reward at time t is R_t
- Ultimate aim to maximise total rewards obtained from $t = [0, T]$
- The agent-environment interaction can be captured as a snapshot of the world at time t
- At each time step t, the agent observes a state S_t, takes an action A_t, to see the next state S_{t+1} and obtain reward R_t



The Agent and the Environment

- At each time step t, the agent observes a state S_t, takes an action A_t, to see the next state S_{t+1} and obtain reward R_t
- Forms a time-series

$$(S_0, a_0, r_0, S_1, a_1, r_1, S_2 ...)$$



Markov Property and Markov Decision Processes

MDPs can be defined by the tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- S : set of states
- A : set of actions
- P_{sa} : transition probabilities
- γ : discount factor
- $R : S \times A \rightarrow \mathbb{R}$: reward function

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

Goal in RL:

$$\max \mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots]$$

Components of an Agent

1. Policy : $\pi : S \to A$
2. Value Function $V(s)$:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

3. State-Action Value Function q_π :

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

Types of Agents

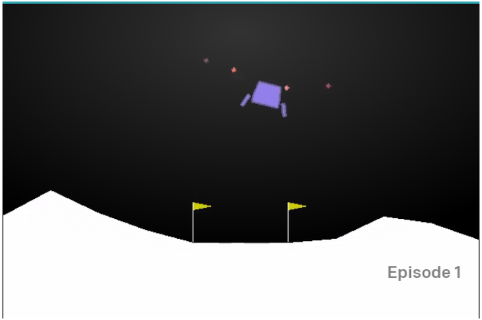
1. Policy Based Agents :
   Store the policy using some representation of it, directly modeling the policy
   Pros : Guaranteed Convergence
   Cons : Reach the goal policy very slowly

2. Value Based Agents :
   Agents that take decisions based on the Value Function, estimating the expected reward obtained by taking an action in a state
   Pros : Sample Efficient, and faster approach towards a good policy **(HOW?)**
   Cons : Can not ascertain convergence

Exploring Value Based Methods in our subsequent sections!

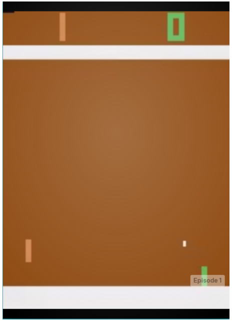
Environments experimented on (among others)

Simple Environment



- Lunar Lander Environment

Atari Environment



- Pong - AI playing Table Tennis

Reinforcement Learning Algorithms in the lead up to Deep Q-Learning

Quick Introduction to TD-Methods

- Temporal Difference (TD) Methods predict the value of the total reward
- This method follows the general trend of iterative gradient descent/ascent we see in Supervised Learning
- Each update happens with a single time-step delay (hence *Temporal*)

$$v(s_t) \leftarrow v(s_t) + \alpha \Big[R_{t+1} + \gamma v(s_{t+1}) - v(s_t) \Big]$$

Q-Learning

- Model-Free, and Off-Policy (What do these terms mean?)
- The Bellman Optimality Equation

$$q^*(s,a) = E_{s'}\left[r + \gamma max_{a'} q^*(s',a')\right]$$

- The Q-Learning Algorithm is based on a TD-variant of the Bellman Optimality Equation

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

Q-Learning Algorithm

Algorithm 1: Q-Learning (off-policy TD)

Parameters: γ, $\alpha \in (0, 1]$, $\epsilon > 0$;
for *each episode* **do**
  initialise start state s;
  while *not done* **do**
    choose action a using ϵ-greedy policy;
    take action a, observe r, s';
    $Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma max_{a'} Q(s', a') - Q(s, a)]$ $s \leftarrow s'$
  end
end

ϵ-Greedy Policy

- At each step, the agent picks an action based on the ϵ-greedy policy

$$action = \begin{cases} \text{random action}, & \text{with prob } \epsilon \\ \text{argmax}_a Q(s, a) & \text{with prob } (1 - \epsilon) \end{cases}$$

- This is used to balance the Exploration-Exploitation Dilemma

$$\epsilon = \begin{cases} \epsilon - \epsilon_{decay}, & \text{if } \epsilon > \epsilon_{min} \\ \epsilon_{min} & \text{otherwise} \end{cases}$$

The Need for Deep Q-Networks

- Q-Learning methods are highly Space-Intensive
- Not suitable for Continuous State Representations
- Mostly require hand-crafted feature sets, can not learn from visual/sensory inputs

Deep Q-Networks and their variants

Challenges faced by RL coupled with DL

Deep Learning

- Expects i.i.d (independent and identically distributed) samples
- Target should be constant for learning stability

Reinforcement Learning

- Highly correlated samples, as consequent actions generate states with minimal changes
- The Q-target is constantly changing, chasing a non-stationary target

Q-Learning Algorithm

Algorithm 2: Q-Learning (off-policy TD)

Parameters: γ, $\alpha \in (0, 1]$, $\epsilon > 0$;
for *each episode* **do**
    initialise start state s;
    while *not done* **do**
        choose action a using ϵ-greedy policy;
        take action a, observe r, s';
        $Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma max_{a'} Q(s', a') - Q(s, a)]$ $s \leftarrow s'$
    end
end

How do DQNs solve these challenges?

1. Experience Replay
   - Each entry of the Experience Replay consists of the following

$$(s_t, a_t, r_t, s_{t+1}, done)$$

2. Target Network
   - Target Network follows the Training Network behind by c steps
   - Two sets of parameters, θ (Training Network) and θ^- (Target Network)

DQN Algorithm

Algorithm 2: Deep Q-Learning with Experience Replay

Initialization;
γ, $\alpha \in (0, 1]$, $\epsilon > 0$;
Initialize Experience Replay Buffer M with capacity N;
Initialize training and target networks, Q and $\hat{Q}$, with weights θ and θ^-, $\theta = \theta^-$;
for *each episode* **do**
    Initialise start state/image x, and preprocess to obtain s;
    while *not done* **do**
        Choose action a using ϵ-greedy policy;
        Execute action a, observe r, and new state/image x';
        Preprocess x' to obtain s';
        Store transition $(s, a, r, s', done)$ in M;
        Sample random mini-batch of transitions $(s_i, a_i, r_i, s_{i+1}, done_i)$;

$$y_i = \begin{cases} r_i, & \text{if } done_i \\ r_i + \gamma max_{a'} \hat{Q}(s_{i+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

        Perform Gradient Descent step on $\left(y_i - Q(s_i, a_i; \theta)\right)^2$ wrt network weights θ;
        Every c steps, reset $Q = \hat{Q}$
    end
end

Double Q-Learning and Double-DQN

Why Double Q-Learning?

- Using common network, shown to cause overestimation of Q-values
- De-coupling of action Selection and Evaluation

Two networks Q_1 with parameters θ_1, and Q_2 with parameters θ_2.

Double Q-Learning and Double-DQN - Update Rule

Double Q-Learning Update Rule

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q_2(S_{t+1}, argmax_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \Big]$$

Corresponding Double-DQN Update Rule

$$y_i^{DoubleDQN} = \begin{cases} r_i, & \text{if terminal step} \\ r_i + \gamma \hat{Q}\left(s_{i+1}, argmax_a Q(s_{t+1}, a, \theta_i); \theta_i^-\right) & \text{otherwise} \end{cases}$$

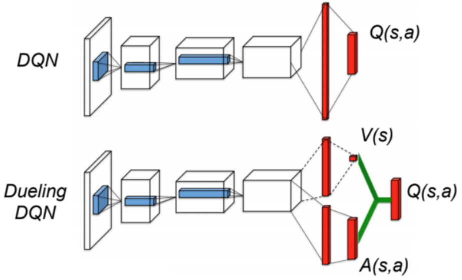
Dueling DQN presents a new architecture

The Dueling network contains two streams:

1. Value Stream
2. Advantage Stream

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$
$$Q^{\pi}(s,a) = A^{\pi}(s,a) + V^{\pi}(s)$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right) \tag{1}$$



Prioritised Experience Replay

A unique sampling method as compared to Uniform sampling strategy employed thus far.

- Key idea to use samples that contribute more to learning more often
- Measure of importance? TD-Error δ
- Unique Data-structure to sample based on priority defined by δ
- Constant sampling of high-error terms may lead to *overfitting*
  - Mitigated by using a stochastic sampling method
  - Each sample assigned $p_i = \delta + e$ value
  - Probability of sampling a transition i, can be written as,

$$P(i) = \frac{p_i^\alpha}{\sum_k p_i^\alpha}$$

  - $\alpha = 0$ - purely random; $\alpha = 1$ - purely greedy.

Training Details

We trained the models for 3 variants - DQN, Double DQN and Dueling DQN. Each training session took approximately:

- Simple Environments
  - Mountain Car : $\sim$ 25 minutes for 500 episodes (one task)
  - Lunar Lander : $\sim$ 50-60 minutes for 500 episodes (one task)
  - Cart Pole : $\sim$ 15 minutes for 500 episodes (one task)
  - Acrobot : $\sim$ 30 minutes for 500 episodes (one task)
- Atari Environments
  - Pong : $\sim$ 4-6 hours for 500 episodes (one task) on a Tesla K80 GPU
- Hyperparameters and Archtitecture Details are presented in the Report in detail.

Results of Simple Environments

- Fixed 3 models - The initial model, final model, and the best model.
- Used two policies for picking actions - Absolute and ϵ-greedy

Mountain Car

Algorithm	Model	μ-Absolute	σ-Absolute	μ-ϵ-greedy	σ-ϵ-greedy
dqn	initial	-200.0	0.0	-200.0	0.0
.	final	-156.375	34.651	-153.445	37.0
.	best	-187.355	26.564	-185.96	27.271
double-dqn	initial	-200.0	0.0	-200.0	0.0
.	final	-187.515	34.641	-193.19	25.753
.	best	-126.32	38.507	-130.43	38.044
Dueling-dqn	initial	-200.0	0.0	-200.0	0.0
.	final	-128.19	32.072	-130.245	32.34
.	best	-161.045	36.481	-162.565	35.962

Table: Mountain Car - Results

Lunar Lander

Algo	Model	μ-Absolute	σ-Absolute	μ-ϵ-greedy	σ-ϵ-greedy
dqn	initial	-162.482	36.233	-158.887	45.988
	final	264.237	40.226	267.897	30.14
	best	-51.298	172.811	-32.329	181.31
double-dqn	initial	-577.074	168.104	-573.091	163.269
	final	249.505	51.21	257.166	31.338
	best	267.475	22.958	266.129	23.009
Dueling-dqn	initial	-865.927	557.288	-195.776	108.549
	final	232.966	87.964	241.798	75.058
	best	239.86	62.36	227.823	65.275

Table: Lunar Lander - Results

Cart Pole

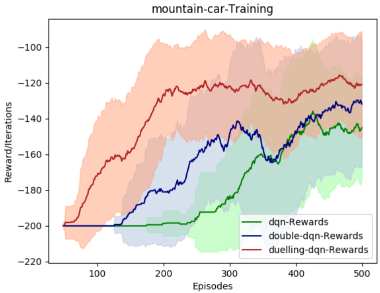
Algo	Model	μ-Absolute	σ-Absolute	μ-ϵ-greedy	σ-ϵ-greedy
dqn	initial	10.29	2.756	10.445	2.887
	final	500.0	0.0	500.0	0.0
	best	500.0	0.0	500.0	0.0
double-dqn	initial	9.7	1.96	9.78	1.795
	final	500.0	0.0	500.0	0.0
	best	500.0	0.0	500.0	0.0
Dueling-dqn	initial	9.5	0.762	23.115	11.339
	final	462.97	80.979	472.0	72.861
	best	500.0	0.0	500.0	0.0

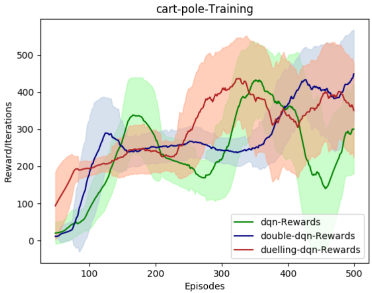
Table: Cart Pole - Results

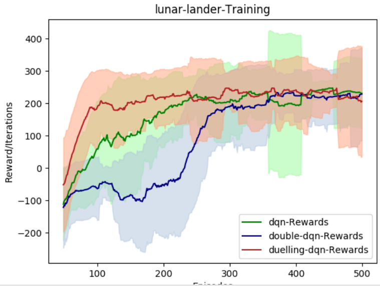
Acrobot

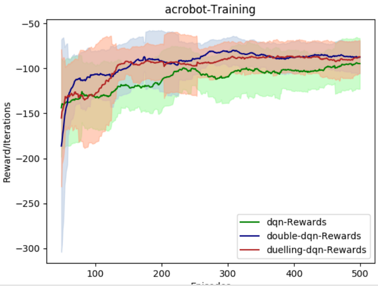
Algo	Model	μ-Absolute	σ-Absolute	μ-ϵ-greedy	σ-ϵ-greedy
dqn	initial	-94.275	62.759	-92.52	41.861
	final	-85.74	20.574	-83.365	19.055
	best	-87.105	33.714	-88.96	27.481
double-dqn	initial	-499.695	2.669	-498.35	11.227
	final	-82.585	16.323	-82.855	14.88
	best	-75.74	11.979	-79.675	16.401
Dueling-dqn	initial	-500.0	0.0	-455.56	70.344
	final	-83.07	15.807	-85.86	22.765
	best	-84.475	16.827	-83.665	15.378

Table: Acrobot - Results


mountain-car-Training
Reward/Iterations
Episodes
dqn-Rewards
double-dqn-Rewards
duelling-dqn-Rewards


cart-pole-Training
Reward/Iterations
Episodes
dqn-Rewards
double-dqn-Rewards
duelling-dqn-Rewards


lunar-lander-Training
Reward/Iterations
dqn-Rewards
double-dqn-Rewards
duelling-dqn-Rewards


acrobot-Training
Reward/Iterations
dqn-Rewards
double-dqn-Rewards
duelling-dqn-Rewards

Preprocessing for Atari Games

The preprocessing in Atari games is multifold:

- SkipFrames
- Image Processing
  - Conversion to Grayscale
  - Cropping off of unnecessary area
  - Resize to 80×80
- Scaling and Buffering

The hyperparameters and Architecture are explained in detail in the Report

Introduction to Generative Adversarial Networks and their Applications

Structure of a Generative Adversarial Network

A GAN is made from two networks -

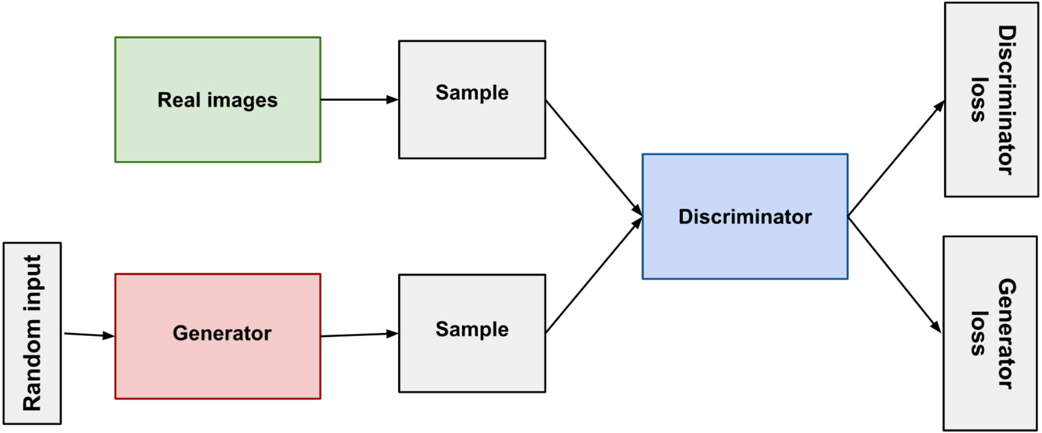
1. The Generator

   Random Input Vector $\rightarrow$ Generator Model $\rightarrow$ Generated Example

2. The Discriminator

   Real and Fake Training Data $\rightarrow$ Generator Model $\rightarrow$ Binary Classifier Outputs

Structure of a Generative Adversarial Network



GANs as a Two-Player Game

- Minimax game, rather than an optimization problem, and have a value function that one agent seeks to maximize and the other seeks to minimize.
- The Generator is trained to become better at generating more plausible samples, in other words, fool the Discriminator.
- The Discriminator is trained and updated to get better at classifying samples

In an ideal scenario where we have a perfect Generator, we should see a prediction probability of 50%

Loss Function of GANs

- We define a distribution $p_z(z)$ over the latent space, and define the Generator's distribution as $p_g(x)$.
- $G(z;\theta_g)$ - function represented by the Generator network with parameters θ_g
- $D(x;\theta_d)$ - Discriminator probability that x is generated from the data rather from p_g
- Minimax Value Function to optimize

$$\min_G \max_D V(D,G) = \mathbb{E}_x\left[logD(x)\right] + \mathbb{E}_z\left[log(1 - D(G(z)))\right]$$

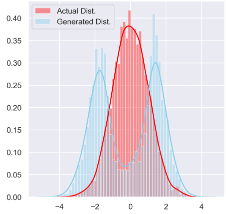
Training GANs

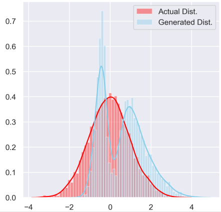
The GAN training process is not very straightforward. The combined GAN network has to separately handle the training of both Generator and Discriminator in an asynchronous way.

1. Discriminator trained separately with equal samples from a frozen Generator and the real data source. Losses calculated are backpropagated only through the Discriminator
2. Discriminator frozen, Generator trained. Loss obtained from the entire network is backpropagated through the Generator

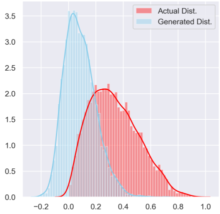
Experiments on simple GANs

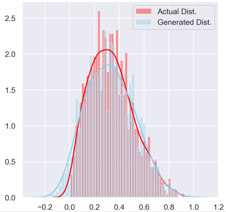
Normal Distribution



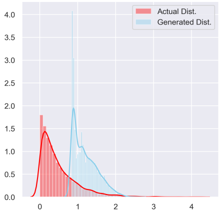


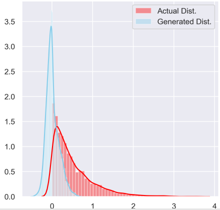
Beta Distribution





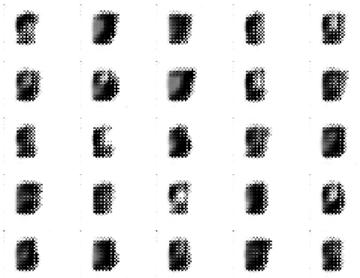
Exponential Distribution



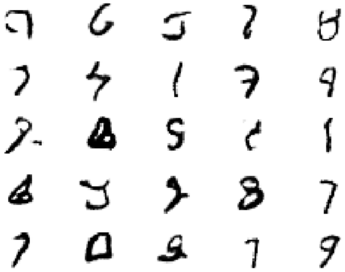


Experiments of GANs on Images

Untrained MNIST-GAN



Trained MNIST-GAN



Initially the output is just noise, but as the network learns from the data, it learns a mixture of features of the digits. Not Good!

Solution: Conditional GANs

With an extra input of class label, create data conditioned on the class label

Untrained Conditional GAN

Trained Conditional GAN

Each column has images of a certain class only

Applying GANs to RL - Generating MDPs using GANs

Generation of MDPs using GANs

Consider a state-transition table having 4 states

	S_0	S_1	S_2	S_3
S_0	0.6	0.1	0.2	0.1
S_1	0.2	0.5	0.2	0.1
S_2	0.2	0.0	0.6	0.2
S_3	0.1	0.1	0.1	0.7

The (i, j) entry of the matrix is the probability of moving from state S_i to state S_j, and is equal to P_{S_i,S_j}.

Implementation

- Train individual GANs for each state, to obtain the transition probabilities for each state.
- For each state S_i, we train the GAN to obtain the probabilities $P_{S_i,S_j}, S_j \in S$
- GANs do not work well with discrete data, hence One-Hot encoded the states $S_0 = [1,0,0,0]$, $S_1 = [0,1,0,0]$ and so on.
- Model Architectural Details are presented in the Report in detail.
- The best Generator is frozen, and samples are generated

Results

Let's compare the Original and GAN-generated Probabilities

	S_0	S_1	S_2	S_3
S_0	0.6	0.1	0.2	0.1
S_1	0.2	0.5	0.2	0.1
S_2	0.2	0.0	0.6	0.2
S_3	0.1	0.1	0.1	0.7

	S_0	S_1	S_2	S_3
S_0	0.611	0.105	0.203	0.075
S_1	0.182	0.591	0.217	0.010
S_2	0.247	0.000	0.753	0.000
S_3	0.098	0.121	0.000	0.781

- The transition probabilities obtained from the GANs almost matched the original transition probabilities of the model
- This is a valid method to generate transition probabilities of simple MDPs

Thank You!