OpenAI Requests for Research 2.0

⋆ ⋆ ⋆ Transfer Learning Between Different Games via Generative Models

PROJECT/PROGRESS REPORT

Authors:

- **Hamza Zafar (Project Lead)**
- **Ashba Jawed (Project Coordinator)**

Contents

1. Problem Statement:

We are always motivated to do breakthroughs in Artificial Intelligence (AI); and Artificial General Intelligence (AGI) reveals us the opportunity to dig down and create a strong AI to achieve our dream. Although the domain is quite vast and we can't consider all aspects of intelligence as the start of our work. So, we have decided to focus on a small portion of intelligence i.e. transfer learning where knowledge of one agent can be shared with the other agent.

Open AI's mission is to ensure that Artificial General Intelligence (AGI) benefits all of humanity. it has released its unsolved problems to be solved by anyone. We have found the following problem interesting:

⋆ ⋆ ⋆ **Transfer Learning Between Different Games via Generative Models.** Proceed as follows:

- Train 11 good policies for 11 Atari games. Generate 10,000 trajectories of 1,000 steps each from the policy for each game.
- Fit a generative model (such as the Transformer) to the trajectories produced by 10 of the games.
- Then fine-tune that model on the 11th game.
- Your goal is to quantify the benefit from pre-training on the 10 games. How large does the model need to be for the pre-training to be useful? How does the size of the effect change when the amount of data from the 11th game is reduced by 10x? By 100x?

2. Introduction:

This report describes in detail our approach for creating trajectories (data) and the selection of algorithms for training the policies to get trajectories in the Data Generation section. Then it discusses the algorithm development part of the data processing and it's training details in the Attention model section. Finally, Results are shown and the challenges are discussed for carrying out this research.

The general architecture of our work can be seen in fig.1. It can be seen that we have divided our work into two stages i.e. Data Generation and Algorithm Development. The Data Generation stage contains tasks like training of DQN on multiple games and using these trained network save trajectories i.e. state, action, and reward into the file system.

Then this trajectories data stored in the file system is processed to have a specific format. And it is fed as input to the algorithm. This can be seen in the Algorithm Development Stage in fig.1.

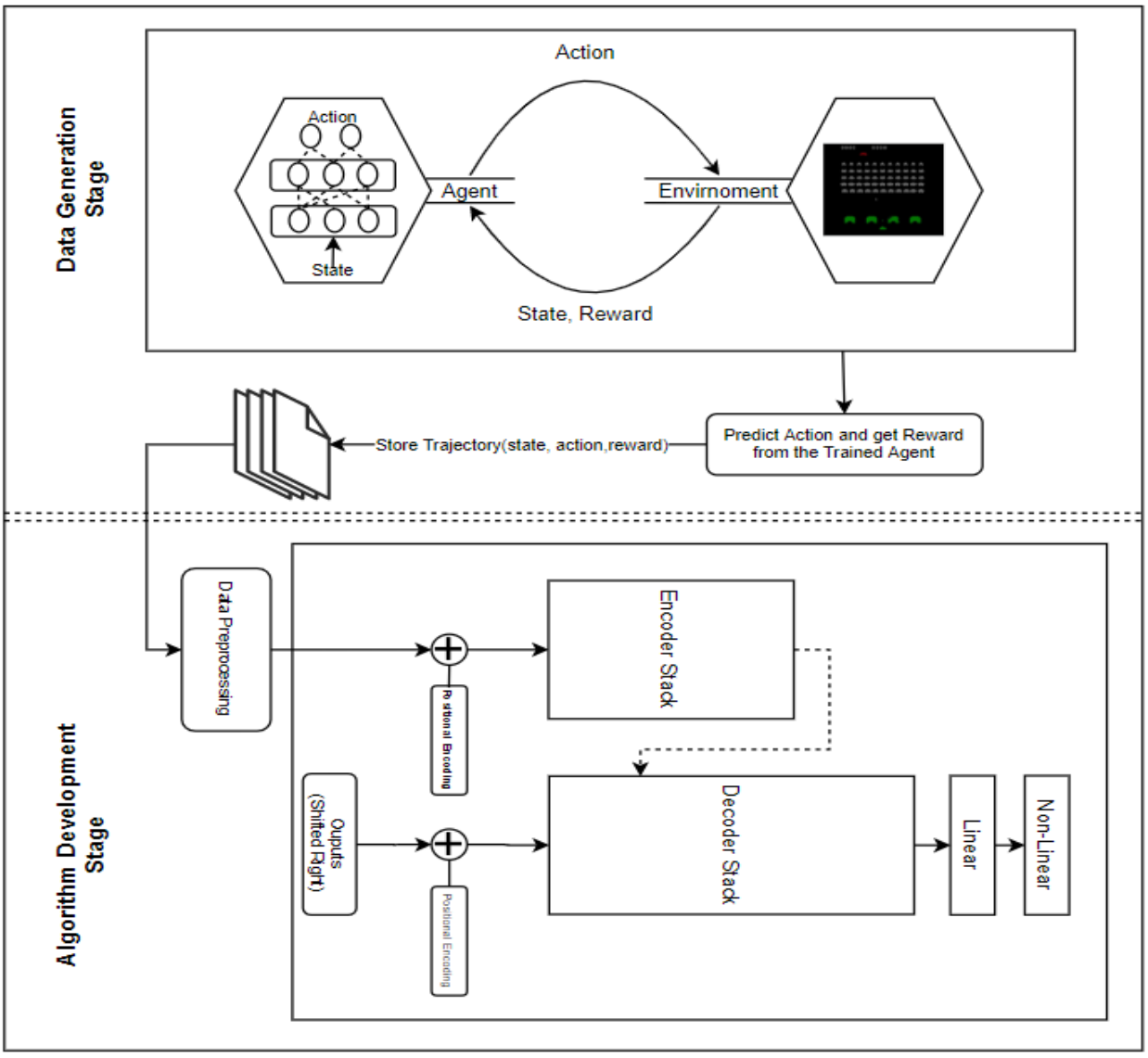

Figure 1. Transfer Learning End-to-End Architecture

3. Data Generation:

To train good policies and generate trajectories, we have selected DQN or deep Q Network as an algorithm and it's the open-source implementation in Keras implementation of which can be found from here. We trained DQN with little hyperparameter tuning on 11 Atari games named as:

1. AirRaid-v0
2. Assault-v0
3. BeamRiders-v0
4. Berzerk-v0
5. Bowling-v0
6. Breakout-v0
7. Carnival-v0
8. Centipede-v0
9. ChopperCommand-v0
10. DemonAttack-v0
11. Solaris-v0
12. SpaceInvaders-v0

Although, the original objective is to use 10 games, for experiment purposes we have also consider the games i.e. Bowling-v0, ChopperCommand-v0; that performed below the human level in 'Human-level control through deep reinforcement learning' research paper.

DQN model was trained for every game separately, the model with the best performance was saved which was used later to generate data for training. For each game, 10000 trajectories of 1000 steps are generated. At each step, a dictionary of current_state, action, and reward is recorded. Where the current state is the game state. That yielded the data in the shape of 10000x1000x3 for each game.

The parameters/hyper-parameters values for training DQN for data production purpose can be seen in the appendix section in Table 1.

3.1. Deep Q-Network:

DQN or deep Q Network was firstly introduced in 2013 in a research paper Playing Atari with Deep Reinforcement Learning on NIPS and was further discussed in 2015 in another paper Human-level control through deep reinforcement learning on Nature as in many games (Atari, chess, go, other board games, etc.) human experts were loosened by DQN.

A reinforcement learning algorithm that combines Q-Learning with deep neural networks to let RL work for complex, high-dimensional environments, like video games, or robotics. In Q-learning for all possible combinations of state (s) and action (a), their Q-value is stored in a memory table called Q-table represented as Q[s, a]. Whereas in DQN, all the actions an under all sates s are expresses by Q-Function which is approximated using a neural network or more specifically by convolutional neural network. An optimal policy is derived by this approximated Q function.

In DQN, a target network, which calculates a target value and is updated by the Q function at regular intervals, is introduced to stabilize the learning process.

4. Algorithm Development:

To fit a model on the generated data, we have selected the "Attention Is All You Need" research paper. Since the original paper focused language task with a sequence of the word, we have adapted most of the concept from the paper with some changes, as the steps in the single trajectory contains also sequence with time.

4.1. Attention Is All You Need:

In many tasks, such as machine translation or dialogue generation, we have a sequence of words as an input (e.g., an original text in English) and would like to generate another sequence of words as an output (e.g., a translation to Korean). Neural networks, especially recurrent ones Recurrent Neural Network (RNN), are well suited for solving such a task.

The transformer is a new encoder-decoder architecture that uses only the attention mechanism instead of RNN to encode each position, to relate two distant words of both the inputs and outputs w.r.t. itself, which then can be parallelized, thus accelerating the training.

4.2. Data Preprocessing:

The data that was generated from the DQN implementation contains states of shape (4, 110, 84) we have flattened it that yielded a row vector of 36960 feature set. Thus created a very computational expensive feature set. To cope up with our limited hardware, PCA is used with fixed number N of components to be retrieved after the operation. In our case, N is used as 256. So, initially, we have a large feature set ie. 36960 is reduced to 256 to be fetched as the input to encoder stack.

4.3. Training Details:

To have control over the algorithm to fit now or for future changes, we have written the whole attention model from scratch in TensorFlow with a bit of change for training trajectories instead of the text data.

Our current implementation uses a single stack of encoder and decoder with n number of heads. And we have used 8 attention heads in the encoder and decoder both. As explained in the data preprocessing section, the data set is flattened to 256, the input to an algorithm for a single trajectory is (1000, 256). Where 1000 is the steps for trajectory and 256 is a new feature set after applying PCA.

With the Attention model, we want to learn attention on action for each state at time step i. We have placed each action for the particular state at the output layer. And want to analyze the agent taking action on a particular state.

In the appendix section Table 2. shows the parameters we have selected during training and most of them are set as defined in the attention paper as of the base model:

Although the original objective of the research was to use 10 games with 10,000 sample set. The code can only train a single sample due to challenges describes in the challenges section. ‘

5. Result:

We have validated our Transformer’s algorithm implementation from scratch on Tensorflow, on the randomly created sample data. And then have used data set (one sample) generated from the DQN.

The sample data contains the same shape as described in the above data section but the only difference is the randomly assigned input values that range from 0-255 and output values that are 0 and 1.

Instead of considering 1000 trajectories from our sample data, we have only considered 128 and 256 steps and that can be seen in fig.2, fig.3 respectively. The y-axis represents the Mean Cross Entropy Sigmoid error in the logarithmic scale and the x-axis shows epochs of training.

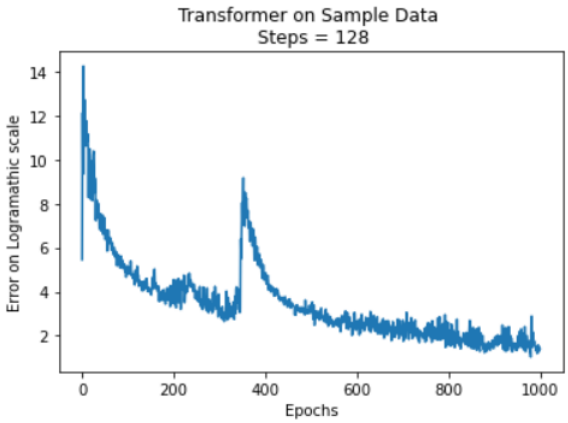

Figure 2. Error curve for steps=128 & Epoch=1000

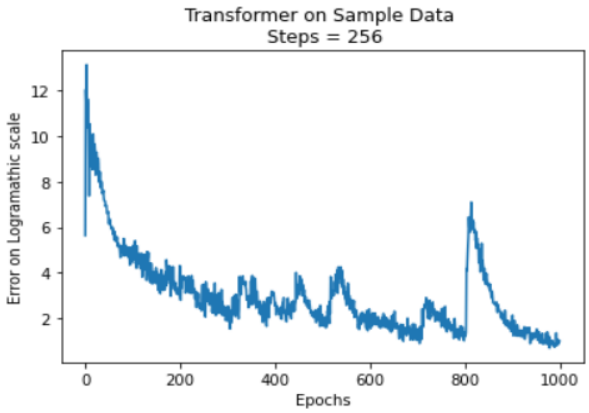

Figure 3. Error curve for steps=256 & Epoch=1000

Each of trajectory in fig.2, fig.3 is trained of 1000 epochs. And both of figure shows the convergence of transformer algorithm. Thus gives us confidence that the algorithm behavior is normal as it reduces error while epoch increases. [Note: due to our current limitation/challenges the algorithm is trained only one sample]

We have then extended our analysis via considering real game (AirRaid-V0) data and used one sample set from the data to observe algorithm convergences on different steps and different epochs.

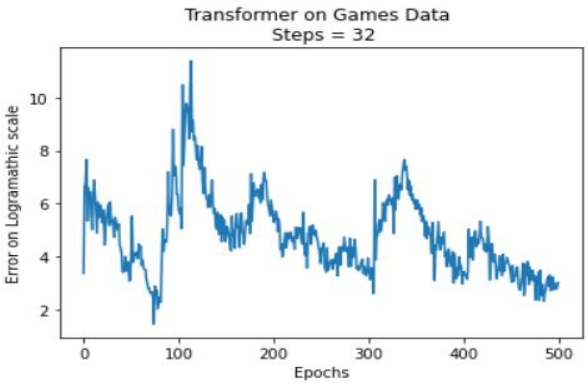

Figure 4. Error curve for steps=32 & Epoch=500

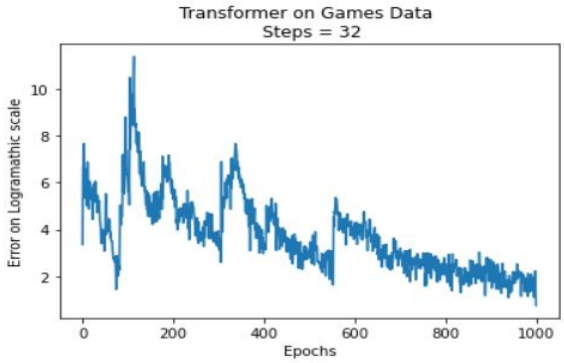

Figure 5. Error curve for steps=32 & Epoch=1000

From fig.4 to fig. 10 we have used Steps 32, 64, 128, 256, and 512 respectively. And we have seen convergences in each of the steps except on trajectory steps 128. The algorithm was unable to reduce the error. We were supposing this behavior because of the size of trajectory steps increases it becomes harder for the algorithm to converge.

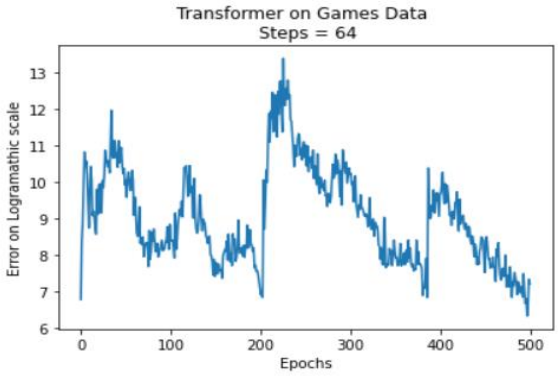

Figure 6. Error curve for steps=64 & Epoch=1000

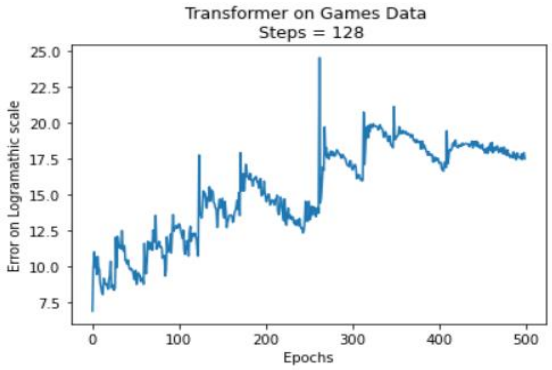

Figure 7. Error curve for steps=128 & Epoch=1000

But in cases of Figures No.8, 9, and 10, the same thing does not hold true i.e. for higher trajectory steps 256, 512 the algorithm still shows convergence.

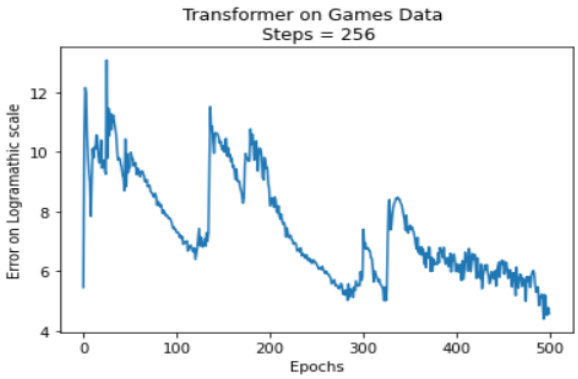

Figure 8. Error curve for steps=256 & Epoch=1000

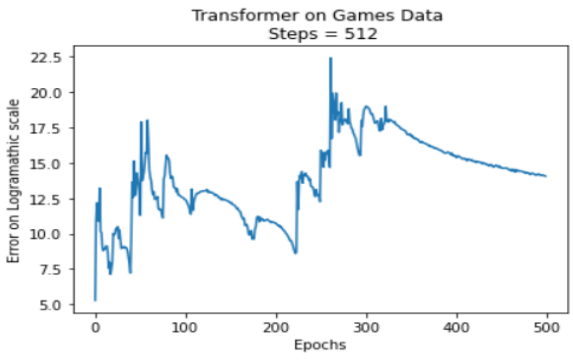

Figure 9. Error curve for steps=512 & Epoch=1000

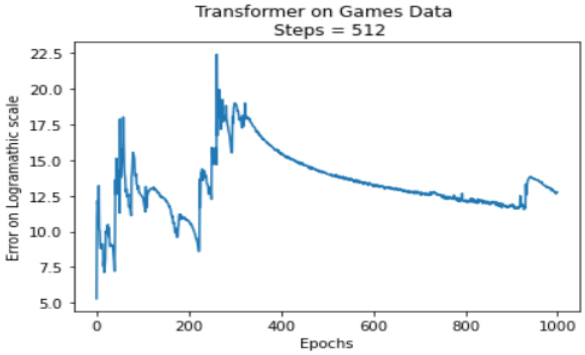

Figure 10. Error curve for steps=512 & Epoch=1000

Although, the convergence of steps 512 is not good when it is trained of 500 epochs in fig. 9, when the epoch size is increased to 1000 we can observe further convergence in fig.10 and we may say

that the unusual behavior in the trajectory steps 128 is by chance, and the algorithm implementation is cable of convergence when trajectory steps are higher.

6. Challenges/Constraints:

Below are the challenges we faced during solving the Research problem:

- **Time:** As both of us were working on this problem in our spare time. It hinders our progress towards producing good results. This research requires more time to be focused and dedicated.
- **Hardware Resources:** We didn't have high computational resources. Only data generation took 3-4 months due to this reason. Google Collab and Kaggle kernel didn't help as much as they have set limits for continuous use. We can't even improve our data quality as of the hardware limitations. The same limitation is also there when training the Attention model on the data.
- **Supervisor:** Writing a bug-free algorithm and working such difficult and complex algorithms like DQN and transformer, it is necessary to have a supervisor who can validate our results. And give us directions to optimize it.

7. Future Work:

Due to challenges, we have discussed above in Section 6, we will like to direct the research project as of the following direction:

- Tensorflow graph is running on Single training example, change the code so that it should take m multiple example.
- One stack of encoder and decoder is implemented. In future we want to increase number of stacks to improve training performance.
- Right now on output layer, model has only action values. Want to investigate the behavior of attention by including both reward and action as output.
- Create CNN at the input before positional encoding
- Dealing with high variance and bias in the model and best fit the model
- Ensure that the agent should play game for 1000 steps in each of trajectory.

8. References:

1. https://github.com/danielegrattarola/deep-q-atari
2. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
3. https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4
4. https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b
5. https://www.kdnuggets.com/2019/08/deep-learning-transformers-attention-mechanism.html
6. https://medium.com/@joealato/attention-in-nlp-734c6fa9d98

9. Appendix:

Table 1. Hyper Parameters for DQN training

Parameter	Value	Description
epoch	50-70	
replay_memory_size	1e6	
minibatch_size	32	number of sample to train the DQN at each update
replay_memory_size	1e6	number of samples stored in the replay memory
target_network_update_freq	10e3	frequency (number of DQN updates) with which the target DQN is updated
avg_val_computation_freq	50e3	frequency (number of DQN updates) with which the average reward and Q value are computed
discount_factor	0.99	
update_freq	4	frequency (number of steps) with which to train the 'DQN
learning_rate	0.00025	
epsilon	1	initial exploration rate for the agent'
min_epsilon	0.1	final exploration rate for the agent
epsilon_decrease	9e-7	rate at which to linearly decrease epsilon
replay_start_size	50e3	minimum number of transitions (with fully random policy) to store in the replay memory before starting training
initial_random_actions	30	number of random actions to be performed by the agent at the beginning of each episode
dropout	0	dropout rate for the DQN
max_episodes	1000	maximum number of episodes that the agent can experience before quitting
max_episode_length	1000	maximum number of steps in an episode
max_frames_number	50e6	maximum number of frames during the whole algorithm

Table 2. Parameters Values for training Transformer

	epoch	$\mathbf{d_{ff}}$	**h**	$\mathbf{d_k}$	$\mathbf{d_v}$	$\mathbf{P_{drop}}$
Base model	1000/500	2048	8	64	64	0.1