Transforming Game Play: A Comparative Study of DCQN and DTQN Architectures in Reinforcement Learning

*Note: Sub-titles are not captured in Xplore and should not be used

1st William A. Stigall
College of Computing and Software Engineering
Kennesaw State University
Marietta, United States of America
wstigall@students.kennesaw.edu

***Abstract*—In this study we investigate the performance of Deep Q-Networks utilizing Convolutional Neural Networks (CNNs) and Transformer architectures across 3 different Atari Games. The advent of DQNs have significantly advanced Reinforcement Learning, enabling agents to directly learn optimal policy from high dimensional sensory inputs from pixel or RAM data. While CNN based DQNs have been extensively studied and deployed in various domains Transformer based DQNs are relatively unexplored. Our research aims to fill this gap by benchmarking the performance of both DCQNs and DTQNs across the Atari games' Asteroids, SpaceInvaders and Centipede.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The field of Reinforcement Learning (RL) has witnessed substantial progress with the integration of deep learning techniques, particularly through the development of Deep Q-Networks (DQNs), which have revolutionized the way that Agents learn from and interact with their environments. Google Deepmind introduced the Deep Q-Network in 2013 as a means to learn control policies from high-dimensional sensory input in the form of Atari games [7]. This Deep Q-Network achieved human performance across 49 Atari games using a CNN architecture introduced in AlexNet in 2012 [5]. Our research utilizes the Arcade Learning Environment (ALE) framework and OpenAI Gym to access the ROMS for Atari 2600 games. OpenAI Gym provides a standardized easy-to-use interface for interacting with these game environments [1], [6]. Although Transformers have been introduced to RL, they typically have not been used in isolation and are often combined with convolutional or recurrent structures [2], [4]. Our study aims to extend this body of work by specifically evaluating the performance of DQNs that employ a Vision Transformer architecture as well as ones that in the context of Atari games, thus contributing to the understanding of Transformer-based DQNs without reliance on convolutions or recurrence.

Identify applicable funding agency here. If none, delete this.

II. Related Work

In this section we will provide an overview of the existing literature on Deep RL and how it's been used to train agents.

A. Deep Q-Networks (DQNs)

The beginnings of Deep RL can be largely attributed to the introduction of DQNs in the seminal work by Google Deepmind. The DQN by leveraging convolutional neural networks allowed to learn control policies directly from high-dimensional sensory input and the output of the network being the estimate of future rewards. The preprocessing for the algorithm utilizes frame stacking to provide temporal information for the network. Deepmind's DQN was able to surpass human experts in Beam Rider, Breakout, and Space Invaders [7]. This was built upon in "Human-level control through deep reinforcement learning", The DQN was evaluated largely, outperforming both the best linear agents, as well as professional human games testers across 49 games, marking it as the first artificial agent capable of learning and excelling at a diverse set of challenging tasks [8].

B. Advancements in DQN

There have been numerous improvements tot he original DQN architecture such as Double Q-Learning, a technique designed to mitigate overestimation bias by employing two separate estimators for action values [11]. Dueling, an architecture that decouples the value and advantage functions improving the efficiency of learning and improving training stability [13]. Additionally, Rainbow networks combining elements such as prioritized experience replay, distributional reinforcement learning, multi-step learning and noisy networks [10].

C. Transformers in Reinforcement Learning

Transformers introcued in the seminal work "Attention is All You Need", revolutionized the field of natural language processing by offering an architecture that exclusively relies on self-attention mechanisms, omitting the need for recurrent

layers. This architecture captures dependencies regardless of distance in the input sequence [12]. While Transformers where first found to make significant gains in natural language processing, they then had exceptional applications in computer vision demonstrating that Transformers could compete with CNNs in image recognition tasks. In reinforcement learning, Transformers were found to be difficult to optimize in RL objectives, the solution to this was introducing a Gated Transformer which could surpass LSTMs on the DMLab-30 benchmarks [9]. The Decision Transformer (Chen, R.T.Q., et al. 2019). Decision Transformers offer a unique perspective on RL tasks by framing them as conditional sequence modeling problems. By conditioning an autoregressive model on desired rewards, past states, and actions, the Decision Transformer effectively generates future actions that optimize returns [2].

D. Vision Transformers and Patch Embedding

Vision Transformers (ViTs) represent a significant shift in how images are processed for recognition tasks. Unlike CNNs, which extract local features through convolutional layers, ViTs treat an image as a sequence of patches. These patches are embedded into vectors and processed by the Transformer architecture, enabling the model to capture global dependencies across the image. This approach has shown promising results, with ViTs achieving competitive performance against traditional CNN architectures when trained on large datasets. The key to ViT's effectiveness lies in its patch embedding technique, which converts image patches into a format that the Transformer can process, allowing for a more flexible and efficient way to understand both local and global image features [3].

E. Reinforcement Learning Techniques

Experience Replay is a critical technique in the training of Deep Q-Networks (DQNs), enhancing both learning stability and efficiency. Initially introduced by Lin in 1992, the concept involves storing the agent's experiences at each timestep in a replay buffer and then randomly sampling mini-batches from this buffer to perform learning updates. This method addresses two main issues in reinforcement learning: the correlation between consecutive samples and the non-stationary distribution of data[reference]. The use of separate target and policy networks is a technique introduced to stabilize the training of DQNs. This approach, detailed in the work by Mnih et al. in 2015, involves maintaining two neural networks: the policy network, which is updated at every learning step, and the target network, which is updated less frequently The policy network is used to select actions, while the target network is used to generate the target Q-values for the learning updates. This separation helps to mitigate the issue of moving targets in the learning process, where updates to the Q-values could otherwise lead to significant oscillations or divergence in the learning process. By keeping the target Q-values relatively stable between updates, the training process becomes more stable and converges more reliably[reference]. Huber loss is a loss function introduced by Huber in 1964, which has found application in DQNs due to its advantages over other loss functions like the mean squared error (MSE) Huber loss is less sensitive to outliers in data than MSE, as it behaves like MSE for small errors and like mean absolute error (MAE) for large errors. This property makes it particularly useful in the context of DQNs, where the distribution of the temporal-difference (TD) error can have large outliers. By using Huber loss, DQNs can achieve more robust learning, as the loss function prevents large errors from disproportionately affecting the update of the network weights. This contributes to the overall stability and efficiency of the learning process in DQNs[reference].

III. Methodology

Describe your research methodology. Detail the procedures for data collection and analysis.

A. Enviornments

To streamline the learning process and enhance the efficiency of our RL agents we reduce the action space using the specifications provided by OpenAI's gym library for each game. This mitigates the complexity of the decision-making process. To ensure consistency and fairness across the comparative analysis between the DCQN and DTQN models, we use a fixed seed of 42 for the first environment of the training run, ensuring that all agents are exposed to an identical initial condition. Frame Stacking is implemented to create an explicit temporal dimension for Positional Embedding for the Transformer model.

1) State Preprocessing: For preparation for DQN training, we convert each frame from the environment to grayscale. The frames are resized to 110x84 and then center-cropped to 84x84 pixels before being normalized with a mean of 0.5 and a standard deviation of 0.5.

B. DCQN Architecture

Input Representation: Let $I \in \mathbb{R}^{B\times F\times H\times W}$ represent the input batch, where B is the batch size, F is the number of stacked frames (acting as channels), and H and W are the height and width of the frames, respectively. Each frame is an 84x84 grayscale image, so $H = 84$ and $W = 84$. The DCQN model processes I through a series of operations as follows:

Convolutional Layers: The input is passed through three convolutional layers. Each convolutional operation, denoted by C_i, includes a convolution with ReLU activation followed by batch normalization:

$$X_1 = \text{ReLU}\left(\text{BN}\left(C_1(I)\right)\right)$$
$$X_2 = \text{ReLU}\left(\text{BN}\left(C_2(X_1)\right)\right)$$
$$X_3 = \text{ReLU}\left(\text{BN}\left(C_3(X_2)\right)\right)$$

where C_i represents the i-th convolutional layer. The kernel sizes and strides are implicit in C_i but are not specified here to keep the representation general.

Flattening: The output of the last convolutional layer X_3 is flattened to form a vector X_{flat}.

Fully Connected Layers: The flattened vector X_{flat} is then passed through three fully connected layers with ReLU activations for the first two layers:

$$X_4 = \text{ReLU}\left(F_1(X_{\text{flat}})\right)$$
$$X_5 = \text{ReLU}\left(F_2(X_4)\right)$$
$$X_6 = F_3(X_5)$$

where F_i denotes the i-th fully connected layer operation. The final output X_6 represents the Q-values for each action, thus $X_6 \in \mathbb{R}^{N_{\text{actions}}}$, where N_{actions} is the environment-dependent number of possible actions.

C. DTQN Architecture

The DTQN model is designed to leverage the Transformer architecture for processing sequences of stacked frames from the environment. It consists of several key components: dimensionality reduction, positional embeddings, a Transformer encoder, and fully connected layers for action value prediction. The model can be formalized as follows:

Input Representation: Let $X \in \mathbb{R}^{B\times(F\cdot 84\cdot 84)}$ represent the input batch, where B is the batch size and F is the number of stacked frames, each flattened from an 84x84 grayscale image.

Patch Embedding: Given our input tensor $X \in \mathbb{R}^{B\times(F\cdot 84\cdot 84)}$, where:

The input is transformed into a sequence of embedded patches $X_p \in \mathbb{R}^{B\times(F\cdot N)\times E}$, where $N = \left(\frac{H}{p}\right)\left(\frac{W}{p}\right)$ is the number of patches per frame. This transformation involves unfolding the input into patches, linearly projecting each patch to an embedding space, and reshaping to match the Transformer's input format.

where $W_{\text{proj}} \in \mathbb{R}^{(p^2)\times E}$ and b_{proj} represent the weights and bias of the linear projection layer, respectively. The resulting tensor, x_{final}, with dimensions $[B, F, N, E]$, effectively encodes each patch into an embedding vector of size E, ready for processing by the Transformer encoder; this process is heavily inspired by Vision Transformers (Doistevsky et al., 2021) [3].

Transformer Encoder and Action Value Prediction: The Transformer encoder with gating mechanisms processes X_{pos}, yielding X_{enc}, which is then flattened and passed through fully connected layers to predict the Q-values for each action. The inclusion of gating mechanisms allows the DTQN model to effectively leverage the Transformer's capabilities for sequence processing while maintaining stability and enhancing learning, enabling it to learn complex policies based on sequences of environmental frames.

The gated Transformer encoder processes the sequence X_{pos}, applying self-attention with gating and feed-forward with gating at each layer, yielding X_{enc}:

$$X_{\text{enc}} = \text{GatedTransformerEncoder}(X_{\text{pos}})$$

where the GatedTransformerEncoder function represents the application of multiple layers of the GatedTransformerXL-Layer, each consisting of a self-attention mechanism with a gating mechanism followed by a feed-forward network with another gating mechanism. The gating mechanisms are implemented using linear layers that combine the inputs and outputs of the self-attention and feed-forward sublayers, respectively, allowing for controlled information flow and stabilization of the learning process [9].

Fully Connected Layers: Finally, X_{enc} is flattened and passed through fully connected layers to predict the Q-values for each action:

$$X_{\text{flat}} = X_{\text{enc}}.\text{view}(B, -1)$$
$$X_{\text{fc1}} = \text{ReLU}(W_{\text{fc1}}X_{\text{flat}} + b_{\text{fc1}})$$
$$X_{\text{fc2}} = \text{ReLU}(W_{\text{fc2}}X_{\text{fc1}} + b_{\text{fc2}})$$
$$X_{\text{fc3}} = \text{ReLU}(W_{\text{fc3}}X_{\text{fc2}} + b_{\text{fc3}})$$
$$Q = W_{\text{out}}X_{\text{fc3}} + b_{\text{out}}$$

where $W_{\text{fc1}}, W_{\text{fc2}}, W_{\text{fc3}}, W_{\text{out}}$ and $b_{\text{fc1}}, b_{\text{fc2}}, b_{\text{fc3}}, b_{\text{out}}$ are the weights and biases of the respective fully connected layers, and $Q \in \mathbb{R}^{B\times A}$ represents the predicted Q-values for A actions.

D. Training

Let $\mathcal{M}$ represent the memory buffer of size N, where an experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ is stored at time t. Sampling a minibatch of size k from $\mathcal{M}$ can be mathematically represented by $B = \{e_1, e_2, ..., e_k\} \subseteq \mathcal{M}$.

For the ϵ-greedy strategy: The action a_t at time t is chosen by

$$a_t = \begin{cases} \text{random action from } \mathcal{A}', & \text{with probability } \epsilon \\ \arg\max_{a\in\mathcal{A}'} Q(s_t, a), & \text{with probability } 1-\epsilon \end{cases}$$

where ϵ is the exploration rate, and $Q(s_t, a)$ represents the action-value function estimate for state s_t and action a.

The epsilon decay process is defined to gradually decrease the exploration rate from an initial value of ϵ_{start} to a minimum value of ϵ_{end} over time, encouraging the agent to explore the environment initially and gradually exploit its learned knowledge. The decay is mathematically represented as:

$$\epsilon \leftarrow \max(\epsilon_{\text{end}}, \epsilon \cdot \text{decay\_rate})$$

where decay_rate is a factor less than 1 applied to decrease ϵ after each episode.

Following the selection of an action a_t using the ϵ-greedy strategy, the agent receives a reward r_{t+1} and observes a new state s_{t+1}. The Q-values are updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right)$$

where α is the learning rate, γ is the discount factor, and a' represents all possible actions in the state s_{t+1}.

The loss function used to update the neural network parameters is the Huber loss, which is defined as:

$$\text{Loss} = \frac{1}{k}\sum_{i=1}^{k} L_\delta\left(y_i - Q(s_t(i), a_t(i))\right)$$

where $L_\delta(a)$ is the Huber loss, which is defined as:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2, & \text{for } |a| \le \delta \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

and $y_i = r_{t+1}(i) + \gamma \max_{a'} Q(s_{t+1}(i), a')$ is the target Q-value for the i-th sample in the minibatch.

E. Experimental Setup

Our research

IV. Results

Present the findings of your research. Use tables, figures, and graphs to display data clearly.

A. Model Behavior

1) Centipede: The DTQN Agent learned quickly in Centipede that an easy way to maximize points was to sit in one area the whole time and continuously shoot. Since the Centipede is guaranteed to move side to side across the screen shooting quickly upwards allowed for large sections and points to be accumulated at one time. Not moving is also an advantage because the Spider is more likely to cross the firing path than it is for the Agent to collide with it, resulting in more chances to reap rewards. This allows it to outperform the DCQN over 10000 episodes in this game. The DCQN Agent learns to dodge occasionally, performing best when the Centipede consists of a larger number of pixels (less of it has been destroyed). However, the DCQN Agent also shows signs of the same camping strategy.

V. Discussion

Interpret the results, discuss their implications, and explain how they fit into the broader context of your research area. Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

A. Limitations

1) Patch Embedding Size: Our Transformer implementation uses the same 16x16 patch size as ViT[]. However, in the original paper, the 16x16 patch was designed for 224x224 images; in this study, we use a 16x16 patch on an 84x84 image which is not proportional segmentation of information. Additionally, most implementations like Deit and other lightweight models rely on ViT Feature Extractor to prepossess the images since it would be computationally intensive to do so inside of the model the reason we do not use a 6x6 patch size, which would line up more with ViT in terms of proportionality is that it increases the param size by approximately 700 percent.

2) Model Size: In our study we only tested the DQN and DTQN models in the 36M-39M parameter range, it should be considered that one model might perform differently with a different level of complexity even with the same architecture.In other studies

3) Model Architecture: Our proposed DTQN is one of many possible implementations of Transformers in RL, based on the existing literature, different implementations lead to different findings. For our specific architecture replacing the patch embedding and using Convolutions to extract features from the model before applying the positional embedding leads to a 300 percent reduction in the number of parameters. As a result of this the Embed Size of the Transformer can be doubled to get back to our 30 million parameter class model.

VI. Conclusion

.

VII. Future Work

A. Architectural Improvements

It is possible to use a projection layer on the input before processing to solve our Model Size issue instead of using convolutional layers. This requires disabling gradient calculation on the Linear layer and directly projecting it to the embed size of the Transformer. This method requires extensive testing to determine if this is an acceptable method.

Acknowledgment

The authors would like to thank nobody as there is nobody to thank at this time

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[2] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

[3] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[4] Kevin Esslinger, Robert Platt, and Christopher Amato. Deep transformer q-networks for partially observable reinforcement learning, 2022.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1097–1105, Red Hook, NY, USA, 2012. Curran Associates, Inc.

[6] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, pages 523–562, 2018.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, and Shane Legg. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[9] Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant M. Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, Matthew M. Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing transformers for reinforcement learning, 2019.
[10] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.
[11] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.
[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.
[13] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

APPENDIX A
SEQUENTIAL REPLAY BUFFER

The Sequential Replay Buffer is a modified version of the standard experience replay buffer, designed to handle sequences of observations, actions, rewards, and terminal states. This buffer is particularly useful for training agents that rely on temporal information, such as recurrent neural networks or agents that operate on sequences of observations.

The key components of the Sequential Replay Buffer are as follows:

A. Initialization

The Sequential Replay Buffer is initialized with the following parameters:

$$\begin{aligned}\mathcal{C} &= \text{capacity of the replay buffer}\\ \mathcal{S} &= \text{shape of the state/observation}\\ \mathcal{B} &= \text{batch size}\\ \mathcal{L} &= \text{sequence length}\end{aligned}$$

The buffer is implemented using a *LazyMemmapStorage* to efficiently store the experiences, and a *TensorDictReplayBuffer* to manage the sampling and retrieval of experiences.

B. Adding Experiences

To add a sequence of experiences to the buffer, the `add` method is used:

$$\text{add}(\mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}', \mathbf{d})$$

where:

$$\begin{aligned}\mathbf{s} &= \text{sequence of states/observations} \in \mathbb{R}^{\mathcal{L}\times\mathcal{S}}\\ \mathbf{a} &= \text{sequence of actions} \in \mathbb{Z}^{\mathcal{L}\times 1}\\ \mathbf{r} &= \text{sequence of rewards} \in \mathbb{R}^{\mathcal{L}\times 1}\\ \mathbf{s}' &= \text{sequence of next states/observations} \in \mathbb{R}^{\mathcal{L}\times\mathcal{S}}\\ \mathbf{d} &= \text{sequence of terminal states} \in \{0,1\}^{\mathcal{L}\times 1}\end{aligned}$$

The method ensures that the input tensors have the correct shape and stores the sequence in the replay buffer.

C. Sampling Experiences

To sample a batch of sequences from the replay buffer, the `sample` method is used:

$$\text{sample}() \rightarrow \{\mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}', \mathbf{d}\}$$

The method returns a dictionary containing the sampled sequences of states, actions, rewards, next states, and terminal states.

D. Implementation Details

The Sequential Replay Buffer is implemented using the *LazyMemmapStorage* and *TensorDictReplayBuffer* from the `torchrl` library. The storage keys are initialized with the appropriate shapes to accommodate the sequence-level data. The `add` and `sample` methods are implemented to handle the sequence-level data accordingly.