Add Q- function Part here --

Preprocessing:

To overcome the complexity and high-dimensionality of Atari 2600 games, The first step involves converting the RGB representation of the frames to grayscale and down-sampling the image to a size of 110x84 pixels (or 105x80 in some implementations) from the original size of 210x160 pixels. The down-sampling is done to reduce the number of pixels in the image while still maintaining the relevant information for gameplay.

Next, a cropping stage is applied to obtain a final image size of 84x84 pixels, which is a square input required for the GPU implementation of 2D convolutions used in the Deep Q-Network. This stage involves cropping a central region of the downscaled image to keep the relevant parts of the gameplay.

The final preprocessing step involves normalizing the image pixel values to the range of [0,1] and converting them to the uint8 data type to reduce memory usage. The rewards are clipped to a range of [-1,1] to limit the scale of error derivatives and allow the use of a single learning rate across multiple games. This step could affect the agent's performance as it cannot differentiate between rewards of different magnitudes.

Also, we can stack the last four preprocessed frames to produce the input to the Q-function to give the agent information about the recent history of the game. Overall, the preprocessing steps allow for more efficient training of the Deep Q-Network on Atari games and improve the agent's ability to learn and play the game at a superhuman level.

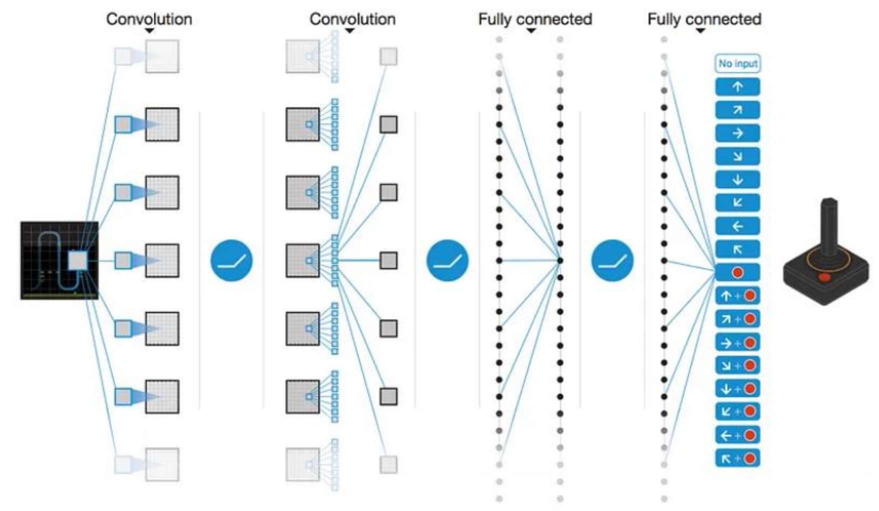
Add Deep Q- learning part ---

Model Definition:

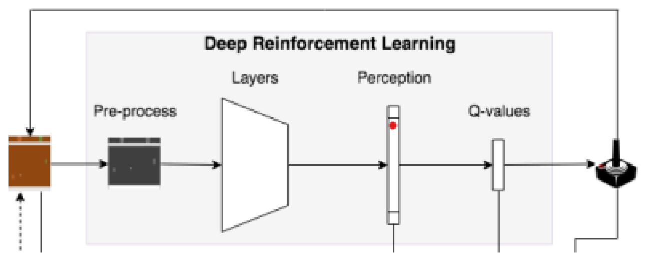
The input to the neural network is a preprocessed image with dimensions 84×84×4, obtained by converting the raw RGB frames to grayscale and down-sampling them to 110x80 followed by cropping a 84x84 region. The first hidden layer of the network is a convolutional layer that convolves the input image with 16 8×8 filters with a stride of 4. The output is then passed through a rectifier nonlinearity (ReLU). The second hidden layer is also a convolutional layer, but this time it convolves the output of the first layer with 32 4×4 filters with a stride of 2, again followed by a ReLU nonlinearity.

The final hidden layer is a fully connected layer consisting of 256 rectifier units. The output layer is a fully connected linear layer with a single output for each valid action. In this experiment, the researchers used the RMSProp algorithm with minibatches of size 32 for training the network.

Overall, the model architecture is designed to extract relevant features from the preprocessed images and predict the Q-values for each action given the current state of the game. The use of convolutional layers allows the model to learn spatial features of the game environment, while the fully connected layers enable it to learn more abstract features and combine them for accurate Q-value predictions. The choice of RMSProp and minibatches helps to stabilize the training process and improve convergence.



A visual representation of the network.



DQN Model:

```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
      end for
  end for
```

The DQN model described incorporates experience replay as a key method to prevent the network from diverging. The idea behind experience replay is to store relevant information (current state, next state, action taken, reward, and whether the next state is terminal) from each step of the game in a finite-size memory, rather than updating the model based on just the last step. During each Q-learning iteration, a batch of 32 elements is sampled from this memory and used to update the model.

To ensure that the network does not forget what it's like to be in states that it hasn't seen in a while, we need to prefill the memory with a random policy up to a certain number of elements (50,000 in their case). This approach is helpful because successive states in reinforcement learning are often highly similar, making it easy for the network to forget about less frequently encountered states. Replaying experience ensures that the network is still showing old frames, helping to prevent this issue.

However, experience replay comes with a significant memory cost. Each preprocessed 110x80 frame takes a minimum of around 9KB to be stored, and with experience replay, the need to store a million of these frames, which amounts to 9GB of RAM. Therefore, it is important to be careful about how memory is used to prevent the DQN from constantly crashing. It is recommended to store frames using the np.uint8 type and converting them to floats in the [0, 1] range at the last moment, as using np.float32 will use 4x as much space and np.float64 (the default float type) will use 8x as much space.

To prevent unnecessary memory usage, it is also recommended to ensure that each unique frame uses the same memory in each state that it is a part of and not calling np.array() multiple times on a given frame or using copy.copy. They also suggest being careful when serializing memory, as most serialization libraries will create multiple copies of the same frame when deserialized.

Add results or conclusion here ---

Future Research:

First, unlike DQNs, target networks do not have the "chasing the target" issue. Without a set goal, the network's predictions can drastically change with each update, making convergence challenging. Target networks minimize the "chasing" effect and produce more stable updates by allowing a fixed target to be used for a set number of iterations. In the end, this results in faster convergence and better performance.

Second, Huber loss is a more reliable loss function for training DQNs than mean squared error (MSE). Mean absolute error (MAE) is not differentiable at zero, whereas mean standard error (MSE) can severely penalize large errors and slow learning. Huber loss offers a middle ground between MSE and MAE, penalizing large errors more severely than MAE but less severely than MSE while remaining globally differentiable. Faster convergence results from more reliable updates as a result.

Third, utilizing larger networks and giving them more training time is a tip for enhancing DQN performance. By adding an additional layer before the first dense layer and increasing the number of filters in each convolutional layer, the DeepMind team was able to double the size of the network. Additionally, they increased the quantity of units in the concealed dense layer. They trained the network for 50 million frames, five times as many as in the previous iteration, to handle the larger network.

Overall, using target networks, Huber loss and Bigger Networks, Longer Training can greatly improve the performance and training efficiency of DQNs.

Add References ---