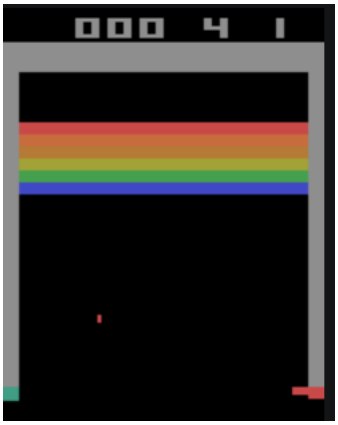
Training agent using DQN on Atari Breakout

Following the paper: "Human-level control through deep reinforcement learning"

December 9, 2025

Figure 1: 

Setup:

We will try to train the agent in the Atari game breakthrough. As we know DQN is a algorithm designed to study the Q values. Q values are the Q function output for given (state, action) pairs. Q output is the maximum accumulated (discounted) reward we could achieve continuing forward given that we are in state - 's' and doing action 'a'. The way that it is done is specified in the paper, in the best way so all I'll explain here is how to implement it on the Atari game Breakout.

Preprocessing:

As the article states the raw Atari 2600 frames are 210X160 pixel images with 128- color palette, this could be demanding. thats why will apply some preprocessing step aimed at reducing the input dimention and dealing with some artefacts of the Atari 2600 emulator. First, to encode a single frame we take the maximum value for each pixel color value over the frame being encoded and the previous frame. This is necessary to remove flickering that is present in games where some object appear only in even frames while others in odd frames. For example the ball in the game could appear in even frames while the paddle could appear in odd ones and it flickers like that. to solve it you can just take the maximum value of the pixel. since the background is black in the game and artefacts/sprites are colored it will work. Second, we extract the Y channel, also known as luminance, from RGB frame and rescale it to 84x84.- so instead of using colored frames we only use the brightness making it colorless and rescale the image. To apply that to the code theres the wrapper module which make preprocessing very easy to achieve all that was said above all we need to do is:

```
env = AtariPreprocessing(
    env,
    screen_size=84
    frame_skip=4,
    grayscale_obs=True,
    scale_obs=True
    noop_max= NO_OP_MAX
) #this are the defualt for this function - just wrote them for clarity

env = FrameStackObservation(env, stack_size=4)
```

Model architecture:

There are several possible ways of parameterizing Q using a neural network. Because Q maps history–action pairs to scalar estimates of their Q-value, the history and the action have been used as inputs to the neural network by some previous approaches24,26. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. We instead use an architecture in which there is a separate output unit for each possible action , and only the state representation is an input to the neural network.. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. -(Human-level control through deep reinforcement learning)

$$\begin{pmatrix} * & * & * & * & * \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ & \dots & \dots & \dots & * \end{pmatrix}_{84\times84} \Rightarrow Neuralnetwork \Rightarrow \begin{pmatrix} value \\ value \\ value \\ \dots \\ value \end{pmatrix}_{6\times1}$$

each index in the output vector represent an action, so each value correspond to the state/input and action.

Neural network description from the paper:

"Model architecture. There are several possible ways of parameterizing Q using a neural network. Because Q maps history–action pairs to scalar estimates of their Q-value, the history and the action have been used as inputs to the neural network by some previous approaches[24,26] *. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. We instead use an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. The exact architecture, shown schematically in Fig. 1, is as follows. The input to the neural network consists of an 84 × 84 × 4 image produced by the preprocessing map w. The first hidden layer convolves 32 filters of 8 × 8 with stride 4 with the input image and applies a rectifier nonlinearity*[31,32] *. The second hidden layer convolves 64 filters of 4 × 4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3 × 3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a*

single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered"

Lets go over the implementation: The code:

```
#creating the neural network
class ConvolutionalNetwork(nn.Module):
    def __init__(self, env):
        super().__init__()

        # Convolutional layers
        self.conv = nn.Sequential(
            # Conv1: 32 filters, 8x8, stride 4
            nn.Conv2d(4, 32, kernel_size=8, stride=4),
            nn.ReLU(),

            # Conv2: 64 filters, 4x4, stride 2
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),

            # Conv3: 64 filters, 3x3, stride 1
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        # Calculate flattened size after conv layers
        # Input: 84x84x4
        # After Conv1 (stride 4): ~20x20x32
        # After Conv2 (stride 2): ~9x9x64
        # After Conv3 (stride 1): ~7x7x64 = 3136

        # Fully connected layers
        self.fc = nn.Sequential(
            # FC1: 512 units
            nn.Linear(7 * 7 * 64, 512),
            nn.ReLU(),

            # Output: one per action
            nn.Linear(512, env.action_space.n)
        )

    def forward(self, x):
        # x shape: (batch, 4, 84, 84)
        x = self.conv(x)  # (batch, 64, 7, 7)
        x = x.view(x.size(0), -1)  # Flatten: (batch, 3136)
        x = self.fc(x)  # (batch, num_actions)
        return x
```

Following the stated in the paper we created the network.

Training details:

"Training details. We performed experiments on 49 Atari 2600 games where results were available for all other comparable methods12,15 . A different network was trained on each game: the same network architecture, learning algorithm and hyperpara- meter settings (see Extended Data Table 1) were used across all games, showing that our approach is robust enough to work on a variety of games while incorporating only minimal prior knowledge (see below). While we evaluated our agents on unmodified games, we made one change to the reward structure of the games during training only. As the scale of scores varies greatly from game to game, we clipped all positive rewards at 1 and all negative rewards at -1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude. For games where there is a life counter, the Atari 2600 emulator also sends the number of lives left in the game, which is then used to mark the end of an episode during training. In these experiments, we used the RMSProp (see http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) algorithm with minibatches of size 32. The behaviour policy during training was e-greedy with e annealed linearly from 1.0 to 0.1 over the first million frames, and fixed at 0.1 thereafter. We trained for a total of 50 million frames (that is, around 38 days of game experience in total) and used a replay memory of 1 million most recent frames. Following previous approaches to playing Atari 2600 games, we also use a simple frame-skipping technique[15] . More precisely, the agent sees and selects actions on every k'th frame instead of every frame, and its last action is repeated on skipped frames. Because running the emulator forward for one step requires much less computation than having the agent select an action, this technique allows the agent to play roughly k times more games without significantly increasing the runtime. We use k = 4 for all games. The values of all the hyperparameters and optimization parameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost. These parameters were then held fixed across all other games. The values and descriptions of all hyperparameters are provided in Extended Data Table 1."

we're implementing this in the the hyper parameters section and the section below the preprocessing, here a clip of the code:

```
#Setting up the replay buffer & reward buffer for tracking
replay_buffer = deque(maxlen = BUFFER_SIZE)
rew_buffer = deque([0.0], maxlen =100)
episode_reward = 0.0

online_net = ConvolutionalNetwork(env)
target_net = ConvolutionalNetwork(env)


#As the paper stated we'll use RMSprop optimizer
optimizer = torch.optim.RMSprop(
    online_net.parameters(),
    lr=LEARNING_RATE,           # 0.00025
    alpha=GRADIENT_MOMENTUM,    # 0.95 (decay factor)
    eps=MIN_SQUARED_GRADIENT,   # 0.01 (epsilon)
    momentum=0,                 # RMSprop doesn't use momentum
```

```
    centered=False
)
```