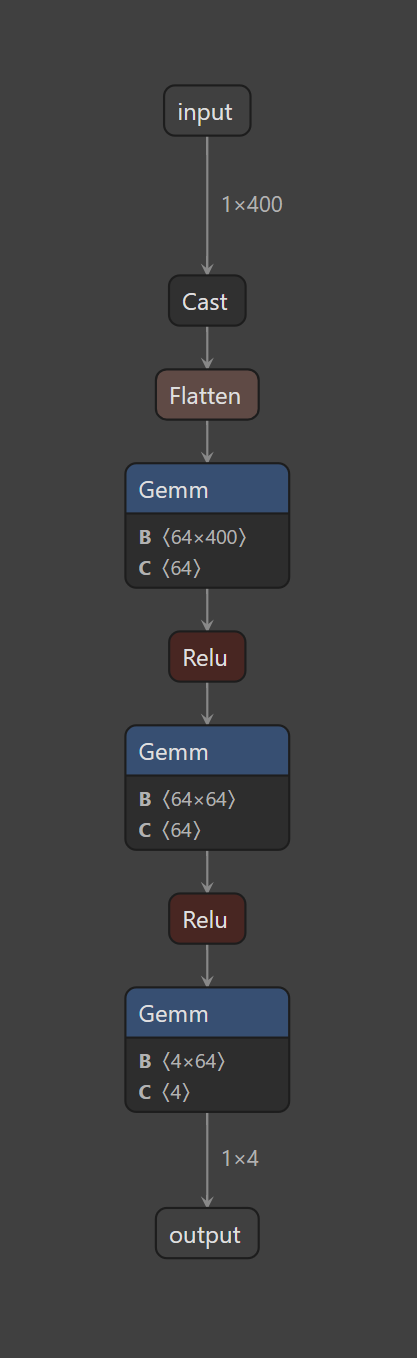
**Visualization of the DQN by Snake Game**

**performing RL on itself**



This network is the visualization of the model created by the game through RL by playing with itself close to 3200 episodes. DQN is Deep Q-Network pioneered by DeepMind to achieving or exceeding human-level performance in playing Atari games. It is a type of reinforcement learning algorithm that combines Q-Learning with Deep Neural Networks to handle high-dimensional state spaces.

*Instead of using a formal Q maximization and divergence minimization between Q predictions and the actual Q results in a game, I will avoid the hard-to-understand equations (as well as avoid needing to deal with Latex)*

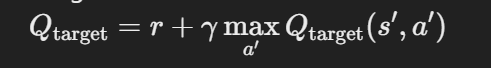
**Key Concepts of DQN:**

1. **Q-Learning**: A value-based reinforcement learning algorithm that estimates the optimal action-value function Q(s,a)Q(s, a)Q(s,a), which predicts the expected future rewards for taking action aaa in state sss.
2. **Deep Neural Networks**: Instead of maintaining a table of Q-values for all state-action pairs (which is infeasible in complex environments), DQN uses a deep neural network to approximate the Q-function.
3. **Experience Replay**: Instead of updating the Q-network based on consecutive experiences (which can cause instability), DQN stores past experiences in a **replay buffer** and samples mini-batches randomly to improve learning stability.
4. **Target Network**: To improve learning stability, DQN uses two neural networks:
   * **Online Network** (used for choosing actions)
   * **Target Network** (used to compute target Q-values) The target network is updated periodically to reduce instability.

In this experiment, we can see snake\_gameRL1.py use the following DQN steps:

**DQN Algorithm Steps:**

1. Initialize the Q-network with random weights.
2. Use the **ϵ-greedy policy** (exploration-exploitation trade-off) to choose an action.
3. Store the experience **(state, action, reward, next state)** in the replay buffer.
4. Sample a mini-batch of experiences from the buffer.
5. Compute the target Q-value using



1. Update the Q-network by minimizing the loss between predicted and target Q-values.
2. Periodically update the target network.

Using a rather simple DQN, it has produced this deep neural network architecture taking in 400 inputs (for the 20x20 grid Snake Environment) to approximate the calculation of all possible Q (rewards/outputs) for any action done on any given state on the 400 positions in the grid (Snake environment). See again the shape and parameters for the DNN in the first diagram above.

There are only 4 possible actions (snake moves up, down, left or right).

There are also 4 total possible states for any point in the grid:

1. Occupied by any part of the snake
2. Occupied by a trap/obstacle
3. Empty or Free
4. Occupied by an apple/fruit

Then there are 4 possible rewards from the combination of action performed on a given state and this reward system is essentially the Policy that drives the preferred action:

1. -100 if the snake collides with the wall or itself (episode termination/game over)
2. -10 when hitting a trap (and the snake’s length is cut into half)
3. -0.1 per normal move (it is slightly negative to encourage the move towards a fruit) or else it can just loop around hungry in an area where there is no traps and the game is pointless
4. + sum of (10 points + length of snake in grid count + number of traps to reflect difficulty bonus) when eating a fruit.

The resulting AI model avoids the need to create a 400x4x4x4 combinatoric system for maximizing Q by creating a 3-layer NN with 400 original possible spaces in the grid as inputs into 64 neurons and outputting eventually just 4 actions and corresponding Q values. All heuristics and activation logic for approximating the values are basic ML functions like ReLU.

Whether or not a training run of more episodes or other epsilon values or a more sophisticated DQN approach like double-DQN etc. would produce a significantly smarter game logic, is something to investigate but not very important at the moment.

Here below are print outs about the model as created, which are just some output of typical characteristics that describe the NN after loading the binary image of the model (created during the RL training session).

We are calling this model “DQN\_snake\_model1” as we anticipate producing other models with other RL sessions sometime for comparisons. This was also converted to .onnx format in the same python script to output the characteristics printed below. The above interpretation from Netron of the .onnx file appears to be a great visualization tool.

=== Model Policy Architecture ===

DQNPolicy(

(q\_net): QNetwork(

(features\_extractor): FlattenExtractor(

(flatten): Flatten(start\_dim=1, end\_dim=-1)

)

(q\_net): Sequential(

(0): Linear(in\_features=400, out\_features=64, bias=True)

(1): ReLU()

(2): Linear(in\_features=64, out\_features=64, bias=True)

(3): ReLU()

(4): Linear(in\_features=64, out\_features=4, bias=True)

)

)

(q\_net\_target): QNetwork(

(features\_extractor): FlattenExtractor(

(flatten): Flatten(start\_dim=1, end\_dim=-1)

)

(q\_net): Sequential(

(0): Linear(in\_features=400, out\_features=64, bias=True)

(1): ReLU()

(2): Linear(in\_features=64, out\_features=64, bias=True)

(3): ReLU()

(4): Linear(in\_features=64, out\_features=4, bias=True)

)

)

)

=== Q-Network Architecture ===

QNetwork(

(features\_extractor): FlattenExtractor(

(flatten): Flatten(start\_dim=1, end\_dim=-1)

)

(q\_net): Sequential(

(0): Linear(in\_features=400, out\_features=64, bias=True)

(1): ReLU()

(2): Linear(in\_features=64, out\_features=64, bias=True)

(3): ReLU()

(4): Linear(in\_features=64, out\_features=4, bias=True)

)

)

=== Layer Details and Weights ===

q\_net.0.weight: shape = (64, 400), requires\_grad = True

q\_net.0.bias: shape = (64,), requires\_grad = True

q\_net.2.weight: shape = (64, 64), requires\_grad = True

q\_net.2.bias: shape = (64,), requires\_grad = True

q\_net.4.weight: shape = (4, 64), requires\_grad = True

q\_net.4.bias: shape = (4,), requires\_grad = True

Using fallback dummy input of shape: (1, 400)

Running a forward pass through the Q-network to capture activations...

Output of Q-network forward pass:

tensor([[-34.8706, -38.9922, -36.1658, -44.3678]])

Captured Activations (layer: activation shape):

features\_extractor.flatten: (1, 400)

q\_net.0: (1, 64)

q\_net.1: (1, 64)

q\_net.2: (1, 64)

q\_net.3: (1, 64)

q\_net.4: (1, 4)

Exported graph: graph(%input : Float(1, 400, strides=[400, 1], requires\_grad=0, device=cpu),

%q\_net.0.weight : Float(64, 400, strides=[400, 1], requires\_grad=1, device=cpu),

%q\_net.0.bias : Float(64, strides=[1], requires\_grad=1, device=cpu),

%q\_net.2.weight : Float(64, 64, strides=[64, 1], requires\_grad=1, device=cpu),

%q\_net.2.bias : Float(64, strides=[1], requires\_grad=1, device=cpu),

%q\_net.4.weight : Float(4, 64, strides=[64, 1], requires\_grad=1, device=cpu),

%q\_net.4.bias : Float(4, strides=[1], requires\_grad=1, device=cpu)):