Playing DOOM with Deep Reinforcement Learning

We train a Deep Reinforcement Learning model to play the DOOM basic level.

Take a look at this image. You have three moves available:

- Move left
- Move right
- Shoot

Most players would probably choose to move left, observing that the monster is on the left side of the screen, so it is more "rewarding" to move left towards the monster.

This is the idea of deep reinforcement learning: choosing the move that is most "rewarding".

Quantifying "rewards" with Gymnasium

Gymnasium is a Python library for Reinforcement learning. It provides environments (`gym.Env`) in which agents can take actions using `step()` from an `action_space` and receive feedback in the form of rewards defined by each environment.

For the DOOM basic environment, we use the Vizdoom library which registers the environment under the id `VizdoomBasic-v0` .

REWARDS:

- +106 for killing the monster
- -5 for every shot
- +1 for every tic the agent is alive

The episode ends after killing the monster or on timeout.

CONFIGURATION:

- 4 available actions: `action_space` is `Discrete(4)`
- 1 available game variable: ammo count

The episode timeouts after 300 frames.

In [ ]:
```
from vizdoom import gymnasium_wrapper
import gymnasium as gym

environment = gym.make('VizdoomBasic-v0', render_mode = 'rgb_array')
```

Training details

The Bellman equation

We have a function Q which takes the observation as an input and returns a list of n numbers, where n is the size of the action space. An optimal or near-optimal Q function will return the expected reward for each action.

The Bellman equation quantifies the training process as follows:

$$Q\left(S_t, A_t; \theta_t\right) \leftarrow Q\left(S_t, A_t; \theta_t\right) + \alpha\left(R_{t+1} + \gamma \underset{a}{\operatorname{argmax}} Q\left(S_{t+1}, a; \theta_t\right) - Q\left(S_t, A_t; \theta_t\right)\right)$$

It may look daunting, but we can break down the equation into its components.

Let's break the right side down component by component.

- $Q\left(S_t, A_t; \theta_t\right)$ gives the original prediction of the model Q given the state S_t, the action A_t, and based on a set of weights θ_t.
- $R_{t+1} + \gamma\left(\underset{a}{\operatorname{argmax}} Q^*\left(S_{t+1}, a\right)\right)$ is known as the `TD Target` (we denote this as T_t), and gives the reward of an action.
  - R_{t+1} is the immediate reward returned from `gym.Env.step(action)`
  - The remainder of the expression is the expected maximum reward of the next state as predicted by the model Q. The $\underset{a}{\operatorname{argmax}}$ selects the action a that will lead to the maximum value of $Q\left(S_{t+1}, a; \theta_t\right)$ (or the action that will maximize the expected reward). This reward is discounted by a factor γ.
- $T_t - Q\left(S_t, A_t; \theta_t\right)$ gives the residual, which is the amount by which the model has overestimated or underestimated the true reward. It is known as the `TD Error` (we denote this as T_e).

In the original expression, we write $Q\left(S_t, A_t; \theta_t\right) \leftarrow Q\left(S_t, A_t; \theta_t\right) + \alpha \cdot T_e$, assigning the value of the original prediction plus the residual multiplied by a learning rate α.

In practice, we would use a library like Tensorflow which provides automatic differentiation with `tf.GradientTape()` and backpropagate the TD Error through the network and update the model weights so that the model produces a closer prediction.

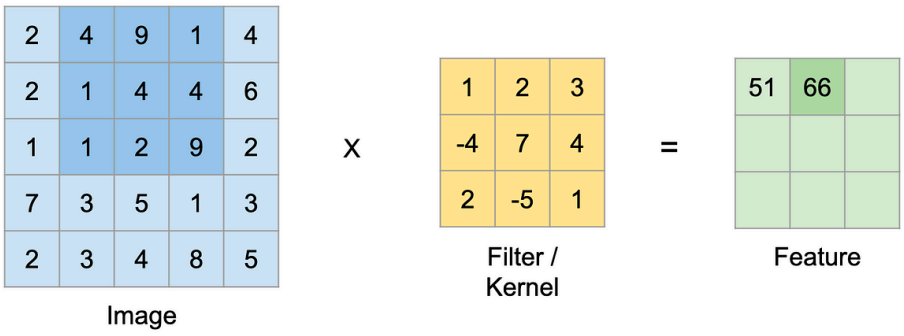
Convolutional Neural Networks

How will the model actually process inputs, though?

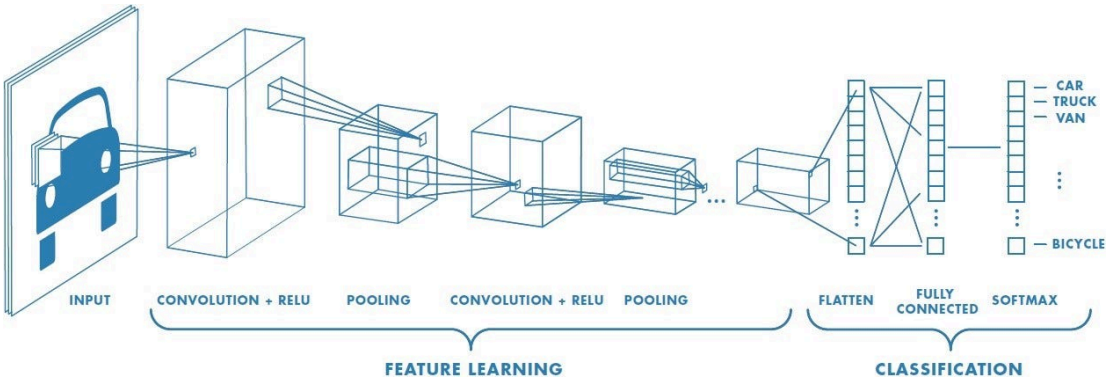
- Input space: 84×84 image (7056-dimensional, $\approx 4.2 \cdot 10^{50977}$ observations)
- Output space: 4 actions (4-dimensional)

It is evidently unreasonable to enumerate the reward of each possible input; this would take up many more orders of magnitude of storage than there are atoms in the universe. Instead, we resort to a well-known architecture in deep learning known as Convolutional Neural Networks.

A convolutional neural network applies an operation known as a convolution (hence the name) to a rectangular region of spatially-structured data using a "kernel" of custom size, with the result of the operation being the sum of the element-wise product of the kernel's values and the data.



Most convolutions will reduce the size of an image; by applying multiple such kernels, each with different values, a convolutional neural network reduces size while expanding "channels". In the image below, you can see this as boxes getting smaller but deeper progressing through the network.



Apart from the filter count and kernel dimensions, the strides parameter can also be adjusted for each layer, which decides the downsampling factor (how much the image is scaled down).

Epsilon-greedy sampling

There is a tradeoff in reinforcement learning between exploration and exploitation; the two terms are briefly explained below.

- Exploration: the agent tries something new to avoid falling into a "local minimum" which seems optimal but falls short to another undiscovered strategy
- Exploitation: the agent exploits the best strategy previously-found rather than taking a risk with exploration.

These two must be balanced in order to have a good agent: an agent with too much exploration will fail to settle on the best strategy, whereas an agent with too much exploitation may settle into a local minimum and fail to find a better strategy that an exploring agent could have found.

Epsilon-greedy sampling is a method where the agent will try a random action with probability epsilon, versus the "best" action it has found otherwise. In other words, the selected action a from an action space A is given as follows.

$$a = \begin{cases} a \sim A & \text{W.P. } \epsilon \\ \underset{a}{\operatorname{argmax}}\, Q\left(S_t, a\right) & \text{W.P. } 1 - \epsilon \end{cases}$$

We will implement a convolutional neural network called `DeepQ` using the `keras` library, which provides the machine learning interface. This `DeepQ` class has adjustible filter counts, kernel sizes, and strides in its convolution section, followed by two dense layers that output the values. Note that the last dense layer should have the same number of neurons as the number of output classes, or for reinforcement learning, the number of possible discrete actions.

In [ ]:
```
import keras
import random
import numpy as np

@keras.saving.register_keras_serializable()
class DeepQ(keras.Layer):
    def __init__(self,
```

```
                 filters,
                 kernels,
                 strides,
                 dense_units,
                 **kwargs): # Constructor
        super(DeepQ, self).__init__() # Initialize keras.Layer
        self.filters, self.kernels, self.strides, self.dense_units = filt

        # Construct convolution layers
        for i in range(len(filters)):
            conv = keras.layers.Conv2D(filters = filters[i],
                                       kernel_size = kernels[i],
                                       strides = strides[i],
                                       padding = 'same')
            self._layers.append(conv)

        # Flatten downsampled tensor
        self._layers.append(keras.layers.Flatten())

        # Construct dense layers
        for i in range(len(dense_units)):
            dense = keras.layers.Dense(units = dense_units[i])
            self._layers.append(dense)
        return None

    def call(self, inputs): # Forward pass
        x = inputs
        for layer in self._layers:
            x = layer(x) # Apply all layers
        return x

    def get_config(self): # Save state
        config = {
            'filters': self.filters,
            'kernels': self.kernels,
            'strides': self.strides,
            'dense_units': self.dense_units,
        }
        base_config = super(DeepQ, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))

    @classmethod
    def from_config(cls, config): # Build from save state
        filters = config.pop('filters')
        kernels = config.pop('kernels')
        strides = config.pop('strides')
        dense_units = config.pop('dense_units')
        layer = cls(filters = filters,
                    kernels = kernels,
                    strides = strides,
                    dense_units = dense_units,
                    **config)
        return layer

def sample_action(env: gym.Env, function: keras.Model, state, epsilon, ve
    if random.random() < epsilon:
        return env.action_space.sample() # Exploration: random action
    else:
        return np.argmax(function.predict(state, verbose = verbosity)) #
```

Motion detection: frame stacking

The current model still suffers from an inability to sense motion. Think about playing a video game, especially one where the player must track a moving target. Motion is necessary to determine which direction, and what speed, to move at. However, it is impossible to determine motion with just a single frame of information. Therefore, this next section introduces a technique called frame stacking.

Typical images are rendered in the three RGB channels, one each for red, green, and blue. If an image has dimensions of $m \times n$, the tensor representing that image's pixels would have a shape of (m, n, 3). Color is unnecessary information in Doom, so we can grayscale each frame in the preprocessing phase. Similarly, in the basic gamemode the roof contains no information, so it is also cropped. The image is then resized to 84×84, with a tensor representation shape of (84, 84, 1). To stack multiple frames, we can simply take several (I use four) consective screen readings:

```
[(84, 84, 1),
 (84, 84, 1),
 (84, 84, 1),
 (84, 84, 1)]
```

Then we can stack them on the channels axis (-1), which gives one tensor of shape (84, 84, 4) which is like one image with "four colors" where each color represents a different point in time.

In [ ]:
```python
from collections import deque
import skimage

def preprocess(frame): # Preprocess an environment observation
    reduce_dims = frame[0] # Input frame shape is (1, ...); this removes
    gs = np.mean(reduce_dims, -1) / 255 # Grayscale
    cf = np.array(gs)[30:-10, 30:-30] # Crop
    result = skimage.transform.resize(cf, [84, 84]) # Resize
    result = np.expand_dims(result, 0) # Re-inserting first dimension
    return result

def stack_state(stack, state, is_new): # Stack a frame `state` onto a deq
    state = preprocess(state) # Grayscale, crop, resize
    if is_new: # The state is the first; it should be stacked 4 times
        stack = deque([np.zeros((84, 84)) for i in range(4)], maxlen = 4)
        for _ in range(4):
            stack.append(state)
    else:
        stack.append(state) # State appended to stack (automatic size man
    tensor = np.stack(stack, -1) # Build the stacked tensor
    return tensor, stack
```

Experience replay

As a network model undergoes training, its weights more closely reflect recent changes. In other words, similar to a human, the model remembers its latest

examples the best. As humans learning a skill or subject, we cannot simply leave past content unreviewed; over time, learning new content, we would forget old content. Similarly, if a model is simply trained on the latest data, it will eventually "forget" about how to deal with old scenarios. To address this issue, the following section introduces a technique known as experience replay.

Experience replay uses a replay buffer, which stores a certain set amount of experiences for future replaying. The exact size of the buffer can be adjusted by the user depending on tradeoffs between system memory and past experience persistence.

For this project, experience replay is implemented using a class wrapping a list with two operations: insert and sample.

Empty memory problem

At the very beginning of training, the memory buffer is empty as the model has not yet engaged with the environment. To address this issue, the memory buffer is initialized to have a preliminary set of experiences for the model to work on.

In [ ]:
```
class ReplayBuffer:
    def __init__(self, buffer_size): # Constructor
        self.elements = []
        self.buffer_limit = buffer_size
        return None
    def insert(self, element): # Insert an element
        if len(self.elements) == self.buffer_limit:
            self.elements.pop(random.randint(0, self.buffer_limit - 1)) #
            self.elements.append(element)
        else:
            self.elements.append(element)
        return None
    def sample(self, count = 1): # Sample `count` elements
        result = []
        for i in range(count):
            result.append(self.elements[random.randint(0, len(self.elemen
        return result

def initialize_memory(env: gym.Env, buffer_size): # Empty memory problem
    buffer = ReplayBuffer(buffer_size) # Initialize buffer
    stack = deque([np.zeros((84, 84)) for i in range(4)], maxlen = 4) # I

    # Sampling from environment
    state, info = env.reset()
    state = state["screen"]
    state = np.expand_dims(state, 0)
    state, stacked_states = stack_state(stack, state, True)
    for i in range(1, buffer_size):
        action = env.action_space.sample()
        next_state, reward, terminated, truncated, info = env.step(action
        next_state = next_state["screen"]
        next_state = np.expand_dims(next_state, 0)
        if terminated or truncated:
            next_state = np.zeros((state.shape))
            buffer.insert((state, action, reward, next_state, True))
            state, info = env.reset()
```

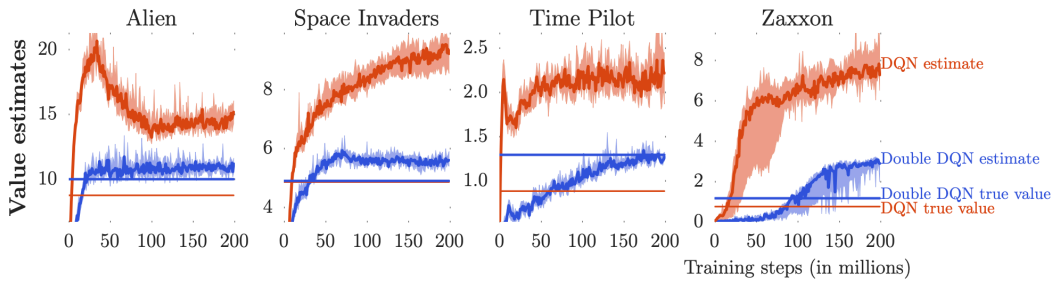
```
            state = state["screen"]
            state = np.expand_dims(state, 0)
            state, stacked_states = stack_state(stack, state, True)
        else:
            next_state, stacked_states = stack_state(stacked_states, next
            buffer.insert((state, action, reward, next_state, False))
            state = next_state
    return buffer, stack
```

Double Q learning

The issue with Deep Q Learning as originally introduced in section 2 above is the overestimation problem. Since the TD Target (T_t) portion of the Bellman equation involves taking an argmax of estimated values, and the model used to predict the value while simultaneously being updated, this means that the model will gradually start to lean toward higher values. Consider the following cause-and-effect process:

- The model takes the maximum reward of the next state, which is higher than the actual maximum reward.
- The maximum reward happens to be higher than the current state estimation.
- The model reads a positive TD error T_e and backpropagates this error through the weights so that they produce a higher reward.

Overestimation occurs when the maximum reward of the next state happens to be higher than the actual maximum reward and the current state estimation, leading to a positive TD error. It may cause the model to overestimate reward values over time which can be problematic. Hasselt et al. illustrates the issue:



Instead, Double Q Learning makes a slight adjustment to the Bellman equation. The original equation is:

$$Q\left(S_t, A_t; \theta_t\right) \leftarrow Q\left(S_t, A_t; \theta_t\right) + \alpha\left(R_{t+1} + \gamma \operatorname*{argmax}_a Q\left(S_{t+1}, a; \theta_t\right) - Q\left(S_t, A_t; \theta_t\right)\right)$$

Double Q Learning uses off-policy: different sets of weights to be trained and for prediction, respectively:

$$Q\left(S_t, A_t; \theta_t\right) \leftarrow Q\left(S_t, A_t; \theta_t\right) + \alpha\left(R_{t+1} + \gamma \operatorname*{argmax}_a Q\left(S_{t+1}, a; \theta_t^*\right) - Q\left(S_t, A_t; \theta_t\right)\right)$$

The only difference between the two versions of the equation is the $\underset{a}{\operatorname{argmax}} Q\left(S_{t+1}, a; \theta_t^*\right)$ which uses θ_t^* instead of θ_t for value estimation. The value estimation weights are fixed for a certain interval of time during which the trained weights can converge upon the model's previous estimation of the true Q function. By freezing the value estimation weights, the Deep Q Network is allowed to stabilize and converge.

With that, here is the training function.

In [ ]:
```
import time
import math

def train(env: gym.Env,
          episodes: int,
          episode_length: int,
          input_shape: tuple,
          conv_filters: list,
          conv_kernels: list,
          conv_strides: list,
          dense_units: list,
          buffer_size: int,
          learning_rate: float = 0.0005,
          epsilon: float = 0.01,
          epsilon_decay: float | None = None,
          gamma: float = 0.999,
          batch_size: int = 32,
          reset_frequency: int = 16,
          verbosity = 0,
          id: int | str = 0) -> keras.Model: # Train a Double Deep Q Netw

    # Conditions: all conv parameters homogeneous length, at least one de
    assert(len(conv_filters) == len(conv_kernels) == len(conv_strides))
    assert(len(dense_units) > 0)
    assert(0 <= epsilon <= 1)

    _w = math.floor(math.log10(episodes)) + 1 # Printing detail

    # Q(S, A, theta)
    input = keras.layers.Input(shape = input_shape)
    q = DeepQ(filters = conv_filters,
            kernels = conv_kernels,
            strides = conv_strides,
            dense_units = dense_units)
    output = q(input)
    function = keras.models.Model(input, output)

    # Q(S, A, theta*)
    target_input = keras.layers.Input(shape = input_shape)
    target_q = DeepQ(filters = conv_filters,
                    kernels = conv_kernels,
                    strides = conv_strides,
                    dense_units = dense_units)
    target_output = target_q(target_input)
    target_function = keras.models.Model(target_input, target_output)

    # Compile functions for training
    function.compile(optimizer = keras.optimizers.Adam(learning_rate), lo
```

```
    target_function.compile(optimizer = keras.optimizers.Adam(learning_ra

    episode_rewards = []

    best_reward = 0.0

    buffer, stack = initialize_memory(env, buffer_size)

    for episode in range(episodes):
        _st = time.time()

        episode_rewards.append(0.0)
        state, info = env.reset()
        state = state["screen"]
        state = np.expand_dims(state, 0)
        state, stacked_states = stack_state(stack, state, True)
        for step in range(episode_length):
            if epsilon_decay is not None: # Reduce epsilon if applicable;
                epsilon *= math.exp(-epsilon_decay)

            # Experience Replay
            # Take an action and observe the results; stack the frames an
            action = sample_action(env, function, state, epsilon, verbosi
            next_state, reward, terminated, truncated, info = env.step(ac
            next_state = next_state["screen"]
            next_state = np.expand_dims(next_state, 0)
            episode_rewards[-1] += reward
            if terminated or truncated:
                next_state = np.zeros((state.shape))
                buffer.insert((state, action, reward, next_state, True))
                state, info = env.reset()
                state = state["screen"]
                state = np.expand_dims(state, 0)
                state, stacked_states = stack_state(stack, state, True)
                continue
            else:
                next_state, stacked_states = stack_state(stacked_states,
                buffer.insert((state, action, reward, next_state, False))
                state = next_state

            # Training
            # Extract experiences from memory and separate into states, a
            batch = buffer.sample(batch_size)
            states = np.array([each[0] for each in batch], ndmin = 3)
            actions = np.array([each[1] for each in batch])
            rewards = np.array([each[2] for each in batch])
            next_states = np.array([each[3] for each in batch], ndmin = 3
            dones = np.array([each[4] for each in batch])

            states = np.squeeze(states)
            next_states = np.squeeze(next_states)
            if batch_size == 1:
                states = np.expand_dims(states, 0)
                next_states = np.expand_dims(next_states, 0)
            q_states = function.predict(states, verbose = verbosity)

            for i in range(batch_size):
                # For each experience in the batch:
                state, action, reward, next_state, done = states[i], acti
                q_state = q_states[i]
```

```
                q_target = reward

                state = np.expand_dims(state, 0)
                next_state = np.expand_dims(next_state, 0)

                if not done: # Add the discounted next reward (theta*); e
                    q_target += gamma * np.amax(target_function.predict(n
                q_state[action] = q_target
                # Fit the model to better approximate the actual Q values
                function.fit(state, np.expand_dims(q_state, 0), verbose =

            # Printing details
            _was_done_str = "env terminated or truncated" if terminated o
            _et = time.time() - _st
            _ec = episode + 1
            _er = episodes - _ec
            _eta = _et / _ec * _er
            print(f"Episode {(episode + 1): {_w}}/{episodes}\t"
                  f"[{'=' * math.floor(step * 25 / episode_length)}>"
                  f"{'-' * (25 - math.floor(step * 25 / episode_length))}
                  f"\tFrame {step + 1} of {episode_length}, "
                  f"{int(_et)}s elapsed); {_was_done_str}", end = '\r')

            if (episode * episode_length + step) % reset_frequency == 0:
                target_function.set_weights(function.get_weights())

        print(f"Episode {episode + 1: {_w}}/{episodes}\t[{'=' * 26}], "
              f"{int(time.time() - _st)}s elapsed{' ' * 50}")

        # Save model if it does well
        if episode_rewards[-1] > best_reward:
            print(f'Reward improved from {best_reward: .3f} to {episode_r
                  f'saving model to file "policy_{id}.keras"')
            best_reward = episode_rewards[-1]
            function.save(f'policy_{id}.keras')
        else:
            print(f'Reward of {episode_rewards[-1]: .3f} did not improve

    return function
```