dqns_on_gcp

June 26, 2024

1 DQNs on GCP

Reinforcement Learning (RL) Agents can be quite fickle. This is because the environment for an Agent is different than that of Supervised and Unsupervised algorithms.

Supervised / Unsupervised	Reinforcement Learning
Data is previously gathered	Data needs to be simulated
Big Data: Many examples covering many siutations	Sparse Data: Agent trades off between exploring and exploiting
The environment is assumed static	The environment may change in response to the agent

Because of this, hyperparameter tuning is even more crucial in RL as it not only impacts the training of the agent's neural network, but it also impacts how the data is gathered through simulation.

1.1 Setup

Hypertuning takes some time, and in this case, it can take anywhere between **10 - 30 minutes**. If this hasn't been done already, run the cell below to kick off the training job now. We'll step through what the code is doing while our agents learn.

[1]:
```
%%bash
BUCKET=<your-bucket-here> # Change to your bucket name
JOB_NAME=dqn_on_gcp_$(date -u +%y%m%d_%H%M%S)
REGION='us-central1' # Change to your bucket region
IMAGE_URI=gcr.io/qwiklabs-resources/rl-qwikstart/dqn_on_gcp@sha256:
 ↪326427527d07f30a0486ee05377d120cac1b9be8850b05f138fc9b53ac1dd2dc

gcloud ai-platform jobs submit training $JOB_NAME \
    --staging-bucket=gs://$BUCKET \
    --region=$REGION \
    --master-image-uri=$IMAGE_URI \
    --scale-tier=BASIC_GPU \
    --job-dir=gs://$BUCKET/$JOB_NAME \
    --config=hyperparam.yaml
```

```
jobId: dqn_on_gcp_191025_173413
state: QUEUED
```

```
Job [dqn_on_gcp_191025_173413] submitted successfully.
Your job is still active. You may view the status of your job with the command

  $ gcloud ai-platform jobs describe dqn_on_gcp_191025_173413

or continue streaming the logs with the command

  $ gcloud ai-platform jobs stream-logs dqn_on_gcp_191025_173413
```

The above command sends a hyperparameter tuning job to the Google Cloud AI Platform. It's a service that sets up scaling distributed training so data scientists and machine learning engineers do not have to worry about technical infrastructure. Usually, it automatically selects the container environment, but we're going to take advantage of a feature to specify our own environment with Docker. Not only will this allow us to install our game environment to be deployed to the cloud, but it will also significantly speed up hyperparameter tuning time as each worker can skip the library installation steps.

The Dockerfile in this directory shows the steps taken to build this environment. First, we copy from a Google Deep Learning Container which already has Google Cloud Libraries installed. Then, we install our other desired modules and libraries. `ffmpeg`, `xvfb`, and `python-opengl` are needed in order to get video output from the server. Machines on the cloud don't typically have a display (why would they need one?), so we'll make a virtual display of our own.

After we copy our code, we tell the container to be configured as an executable so we can pass our hyperparameter tuning flags to it with the ENTRYPOINT command. In order to set up our virtual display, we can use the xvfb-run command. Unfortunately, Docker strips quotes from specified commands in ENTRYPOINT, so we'll make a super simple shell script, train_model.sh, to specify our virtual display parameters. The `"@"` parameter is used to pass the flags called against the container to our python module, `trainer.trainer`.

1.2 CartPole-v0

So what is the game we'll be solving for? We'll be playing with AI Gym's CartPole Environment. As MNIST is the "Hello World" of image classification, CartPole is the "Hello World" of Deep Q Networks. Let's install OpenAI Gym and play with the game ourselves!

[1]:
```
!python3 -m pip freeze | grep gym || python3 -m pip install --user gym==0.26.2
!python3 -m pip freeze | grep 'tensorflow==2.5\|tensorflow-gpu==2.1' || \
!python3 -m pip install -U tensorflow==2.3.0
!python3 -m pip install pygame
```

```
gym==0.17.2
```

Note: Restart the kernel if the above libraries needed to be installed. Please ignore incompatibility errors. The `gym` library hosts a number of different gaming environments that our agents (and us humans) can play around in. To make an environment, we simply need to pass it what game we'd like to play with the `make` method.

This will create an environment object with a number of useful methods and properties. * The `observation_space` parameter is the structure of observations about the environment. - Each "state" or snapshot or our environment will follow this structure * The `action_space` parameter is the possible actions the agent can take

So for example, with CartPole, there are 4 observation dimensions which represent `[Cart Position, Cart Velocity, Pole Angle, Pole Velocity At Tip]`. For the actions, there are 2 possible actions to take: 0 pushes the cart to the left, and 1 pushes the cart to the right. More detail is described in the game's code here.

[2]:
```
from collections import deque
import random

import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models


env = gym.make('CartPole-v0')
print("The observation space is", env.observation_space)
print("The observation dimensions are", env.observation_space.shape)
print("The action space is", env.action_space)
print("The number of possible actions is", env.action_space.n)
```

```
The observation space is Box(4,)
The observation dimensions are (4,)
The action space is Discrete(2)
The number of possible actions is 2
```

- The `reset` method will restart the environment and return a starting state.
- The `step` method takes an action, applies it to the environment and returns a new state. Each step returns a new state, the transition reward, whether the game is over or not, and game specific information. For CartPole, there is no extra info, so it returns a blank dictionary.

[3]:
```
def print_state(state, step, reward=None):
    format_string = 'Step {0} - Cart X: {1:.3f}, Cart V: {2:.3f}, Pole A: {3:.
↪3f}, Pole V:{4:.3f}, Reward:{5}'
    print(format_string.format(step, *tuple(state), reward))

state = env.reset()
step = 0
print_state(state, step)
```

```
Step 0 - Cart X: -0.021, Cart V: -0.045, Pole A: -0.030, Pole V:0.022,
Reward:None
```

[4]:
```
action = 0
state_prime, reward, done, info = env.step(action)
step += 1
print_state(state_prime, step, reward)
```

```python
print("The game is over." if done else "The game can continue.")
print("Info:", info)
```

```
Step 1 - Cart X: -0.022, Cart V: -0.239, Pole A: -0.030, Pole V:0.305,
Reward:1.0
The game can continue.
Info: {}
```

Run the cell below repeatedly until the game is over, changing the action to push the cart left (0) or right (1). The game is considered "won" when the pole can stay up for an average of steps 195 over 100 games. How far can you get? An agent acting randomly can only survive about 10 steps.

[5]:
```python
action = 1  # Change me: 0 Left, 1 Right
state_prime, reward, done, info = env.step(action)
step += 1

print_state(state_prime, step, reward)
print("The game is over." if done else "The game can continue.")
```

```
Step 2 - Cart X: -0.026, Cart V: -0.044, Pole A: -0.024, Pole V:0.003,
Reward:1.0
The game can continue.
```

We can make our own policy and create a loop to play through an episode (one full simulation) of the game. Below, actions are generated to alternate between pushing the cart left and right. The code is very similar to how our agents will be interacting with the game environment.

[6]:
```python
# [0, 1, 0, 1, 0, 1, ...]
actions = [x % 2 for x in range(200)]
state = env.reset()
step = 0
episode_reward = 0
done = False

while not done and step < len(actions):
    action = actions[step]  # In the future, our agents will define this.
    state_prime, reward, done, info = env.step(action)
    episode_reward += reward
    step += 1
    state = state_prime
    print_state(state, step, reward)

end_statement = "Game over!" if done else "Ran out of actions!"
print(end_statement, "Score =", episode_reward)
```

```
Step 1 - Cart X: 0.047, Cart V: -0.171, Pole A: -0.028, Pole V:0.252, Reward:1.0
Step 2 - Cart X: 0.044, Cart V: 0.024, Pole A: -0.023, Pole V:-0.050, Reward:1.0
Step 3 - Cart X: 0.044, Cart V: -0.171, Pole A: -0.024, Pole V:0.236, Reward:1.0
Step 4 - Cart X: 0.041, Cart V: 0.025, Pole A: -0.019, Pole V:-0.065, Reward:1.0
Step 5 - Cart X: 0.041, Cart V: -0.170, Pole A: -0.020, Pole V:0.222, Reward:1.0
Step 6 - Cart X: 0.038, Cart V: 0.025, Pole A: -0.016, Pole V:-0.077, Reward:1.0
Step 7 - Cart X: 0.038, Cart V: -0.169, Pole A: -0.018, Pole V:0.211, Reward:1.0
Step 8 - Cart X: 0.035, Cart V: 0.026, Pole A: -0.013, Pole V:-0.088, Reward:1.0
Step 9 - Cart X: 0.036, Cart V: -0.169, Pole A: -0.015, Pole V:0.201, Reward:1.0
Step 10 - Cart X: 0.032, Cart V: 0.026, Pole A: -0.011, Pole V:-0.096,
Reward:1.0
Step 11 - Cart X: 0.033, Cart V: -0.169, Pole A: -0.013, Pole V:0.193,
Reward:1.0
Step 12 - Cart X: 0.029, Cart V: 0.027, Pole A: -0.009, Pole V:-0.104,
Reward:1.0
Step 13 - Cart X: 0.030, Cart V: -0.168, Pole A: -0.011, Pole V:0.186,
Reward:1.0
Step 14 - Cart X: 0.026, Cart V: 0.027, Pole A: -0.007, Pole V:-0.110,
Reward:1.0
Step 15 - Cart X: 0.027, Cart V: -0.168, Pole A: -0.010, Pole V:0.180,
Reward:1.0
Step 16 - Cart X: 0.024, Cart V: 0.027, Pole A: -0.006, Pole V:-0.116,
Reward:1.0
Step 17 - Cart X: 0.024, Cart V: -0.168, Pole A: -0.008, Pole V:0.175,
Reward:1.0
Step 18 - Cart X: 0.021, Cart V: 0.027, Pole A: -0.005, Pole V:-0.120,
Reward:1.0
Step 19 - Cart X: 0.021, Cart V: -0.168, Pole A: -0.007, Pole V:0.171,
Reward:1.0
Step 20 - Cart X: 0.018, Cart V: 0.028, Pole A: -0.004, Pole V:-0.124,
Reward:1.0
Step 21 - Cart X: 0.019, Cart V: -0.167, Pole A: -0.006, Pole V:0.167,
Reward:1.0
Step 22 - Cart X: 0.015, Cart V: 0.028, Pole A: -0.003, Pole V:-0.127,
Reward:1.0
Step 23 - Cart X: 0.016, Cart V: -0.167, Pole A: -0.006, Pole V:0.164,
Reward:1.0
Step 24 - Cart X: 0.012, Cart V: 0.028, Pole A: -0.002, Pole V:-0.130,
Reward:1.0
Step 25 - Cart X: 0.013, Cart V: -0.167, Pole A: -0.005, Pole V:0.162,
Reward:1.0
Step 26 - Cart X: 0.010, Cart V: 0.028, Pole A: -0.002, Pole V:-0.132,
Reward:1.0
Step 27 - Cart X: 0.010, Cart V: -0.167, Pole A: -0.004, Pole V:0.160,
Reward:1.0
Step 28 - Cart X: 0.007, Cart V: 0.028, Pole A: -0.001, Pole V:-0.134,
Reward:1.0
Step 29 - Cart X: 0.007, Cart V: -0.167, Pole A: -0.004, Pole V:0.158,
Reward:1.0
Step 30 - Cart X: 0.004, Cart V: 0.028, Pole A: -0.001, Pole V:-0.136,
Reward:1.0
Step 31 - Cart X: 0.005, Cart V: -0.167, Pole A: -0.003, Pole V:0.157,
Reward:1.0
```

```
Step 32 - Cart X: 0.001, Cart V: 0.028, Pole A: -0.000, Pole V:-0.137,
Reward:1.0
Step 33 - Cart X: 0.002, Cart V: -0.167, Pole A: -0.003, Pole V:0.155,
Reward:1.0
Step 34 - Cart X: -0.001, Cart V: 0.028, Pole A: 0.000, Pole V:-0.138,
Reward:1.0
Step 35 - Cart X: -0.001, Cart V: -0.167, Pole A: -0.003, Pole V:0.154,
Reward:1.0
Step 36 - Cart X: -0.004, Cart V: 0.028, Pole A: 0.000, Pole V:-0.139,
Reward:1.0
Step 37 - Cart X: -0.004, Cart V: -0.167, Pole A: -0.002, Pole V:0.154,
Reward:1.0
Step 38 - Cart X: -0.007, Cart V: 0.028, Pole A: 0.001, Pole V:-0.140,
Reward:1.0
Step 39 - Cart X: -0.006, Cart V: -0.167, Pole A: -0.002, Pole V:0.153,
Reward:1.0
Step 40 - Cart X: -0.010, Cart V: 0.028, Pole A: 0.001, Pole V:-0.140,
Reward:1.0
Step 41 - Cart X: -0.009, Cart V: -0.167, Pole A: -0.002, Pole V:0.153,
Reward:1.0
Step 42 - Cart X: -0.013, Cart V: 0.028, Pole A: 0.001, Pole V:-0.140,
Reward:1.0
Step 43 - Cart X: -0.012, Cart V: -0.167, Pole A: -0.002, Pole V:0.153,
Reward:1.0
Step 44 - Cart X: -0.015, Cart V: 0.028, Pole A: 0.002, Pole V:-0.140,
Reward:1.0
Step 45 - Cart X: -0.015, Cart V: -0.167, Pole A: -0.001, Pole V:0.153,
Reward:1.0
Step 46 - Cart X: -0.018, Cart V: 0.028, Pole A: 0.002, Pole V:-0.140,
Reward:1.0
Step 47 - Cart X: -0.017, Cart V: -0.167, Pole A: -0.001, Pole V:0.153,
Reward:1.0
Step 48 - Cart X: -0.021, Cart V: 0.028, Pole A: 0.002, Pole V:-0.140,
Reward:1.0
Step 49 - Cart X: -0.020, Cart V: -0.167, Pole A: -0.001, Pole V:0.153,
Reward:1.0
Step 50 - Cart X: -0.024, Cart V: 0.028, Pole A: 0.002, Pole V:-0.140,
Reward:1.0
Step 51 - Cart X: -0.023, Cart V: -0.167, Pole A: -0.001, Pole V:0.154,
Reward:1.0
Step 52 - Cart X: -0.026, Cart V: 0.028, Pole A: 0.003, Pole V:-0.139,
Reward:1.0
Step 53 - Cart X: -0.026, Cart V: -0.167, Pole A: -0.000, Pole V:0.154,
Reward:1.0
Step 54 - Cart X: -0.029, Cart V: 0.028, Pole A: 0.003, Pole V:-0.138,
Reward:1.0
Step 55 - Cart X: -0.029, Cart V: -0.167, Pole A: 0.000, Pole V:0.155,
Reward:1.0
```

```
Step 56 - Cart X: -0.032, Cart V: 0.028, Pole A: 0.003, Pole V:-0.137,
Reward:1.0
Step 57 - Cart X: -0.031, Cart V: -0.167, Pole A: 0.000, Pole V:0.156,
Reward:1.0
Step 58 - Cart X: -0.035, Cart V: 0.028, Pole A: 0.004, Pole V:-0.136,
Reward:1.0
Step 59 - Cart X: -0.034, Cart V: -0.167, Pole A: 0.001, Pole V:0.158,
Reward:1.0
Step 60 - Cart X: -0.037, Cart V: 0.028, Pole A: 0.004, Pole V:-0.135,
Reward:1.0
Step 61 - Cart X: -0.037, Cart V: -0.167, Pole A: 0.001, Pole V:0.159,
Reward:1.0
Step 62 - Cart X: -0.040, Cart V: 0.028, Pole A: 0.005, Pole V:-0.133,
Reward:1.0
Step 63 - Cart X: -0.040, Cart V: -0.167, Pole A: 0.002, Pole V:0.161,
Reward:1.0
Step 64 - Cart X: -0.043, Cart V: 0.028, Pole A: 0.005, Pole V:-0.131,
Reward:1.0
Step 65 - Cart X: -0.042, Cart V: -0.167, Pole A: 0.002, Pole V:0.163,
Reward:1.0
Step 66 - Cart X: -0.046, Cart V: 0.028, Pole A: 0.006, Pole V:-0.129,
Reward:1.0
Step 67 - Cart X: -0.045, Cart V: -0.167, Pole A: 0.003, Pole V:0.166,
Reward:1.0
Step 68 - Cart X: -0.049, Cart V: 0.028, Pole A: 0.006, Pole V:-0.126,
Reward:1.0
Step 69 - Cart X: -0.048, Cart V: -0.168, Pole A: 0.004, Pole V:0.169,
Reward:1.0
Step 70 - Cart X: -0.051, Cart V: 0.028, Pole A: 0.007, Pole V:-0.123,
Reward:1.0
Step 71 - Cart X: -0.051, Cart V: -0.168, Pole A: 0.005, Pole V:0.172,
Reward:1.0
Step 72 - Cart X: -0.054, Cart V: 0.027, Pole A: 0.008, Pole V:-0.119,
Reward:1.0
Step 73 - Cart X: -0.054, Cart V: -0.168, Pole A: 0.006, Pole V:0.176,
Reward:1.0
Step 74 - Cart X: -0.057, Cart V: 0.027, Pole A: 0.009, Pole V:-0.114,
Reward:1.0
Step 75 - Cart X: -0.056, Cart V: -0.168, Pole A: 0.007, Pole V:0.181,
Reward:1.0
Step 76 - Cart X: -0.060, Cart V: 0.027, Pole A: 0.011, Pole V:-0.109,
Reward:1.0
Step 77 - Cart X: -0.059, Cart V: -0.168, Pole A: 0.009, Pole V:0.187,
Reward:1.0
Step 78 - Cart X: -0.063, Cart V: 0.027, Pole A: 0.012, Pole V:-0.103,
Reward:1.0
Step 79 - Cart X: -0.062, Cart V: -0.169, Pole A: 0.010, Pole V:0.193,
Reward:1.0
```

```
Step 80 - Cart X: -0.065, Cart V: 0.026, Pole A: 0.014, Pole V:-0.096,
Reward:1.0
Step 81 - Cart X: -0.065, Cart V: -0.169, Pole A: 0.012, Pole V:0.201,
Reward:1.0
Step 82 - Cart X: -0.068, Cart V: 0.026, Pole A: 0.016, Pole V:-0.088,
Reward:1.0
Step 83 - Cart X: -0.068, Cart V: -0.169, Pole A: 0.014, Pole V:0.210,
Reward:1.0
Step 84 - Cart X: -0.071, Cart V: 0.026, Pole A: 0.019, Pole V:-0.078,
Reward:1.0
Step 85 - Cart X: -0.071, Cart V: -0.170, Pole A: 0.017, Pole V:0.221,
Reward:1.0
Step 86 - Cart X: -0.074, Cart V: 0.025, Pole A: 0.022, Pole V:-0.067,
Reward:1.0
Step 87 - Cart X: -0.074, Cart V: -0.170, Pole A: 0.020, Pole V:0.233,
Reward:1.0
Step 88 - Cart X: -0.077, Cart V: 0.024, Pole A: 0.025, Pole V:-0.053,
Reward:1.0
Step 89 - Cart X: -0.077, Cart V: -0.171, Pole A: 0.024, Pole V:0.247,
Reward:1.0
Step 90 - Cart X: -0.080, Cart V: 0.024, Pole A: 0.029, Pole V:-0.038,
Reward:1.0
Step 91 - Cart X: -0.079, Cart V: -0.172, Pole A: 0.028, Pole V:0.263,
Reward:1.0
Step 92 - Cart X: -0.083, Cart V: 0.023, Pole A: 0.033, Pole V:-0.020,
Reward:1.0
Step 93 - Cart X: -0.082, Cart V: -0.173, Pole A: 0.033, Pole V:0.283,
Reward:1.0
Step 94 - Cart X: -0.086, Cart V: 0.022, Pole A: 0.039, Pole V:0.001, Reward:1.0
Step 95 - Cart X: -0.085, Cart V: -0.174, Pole A: 0.039, Pole V:0.305,
Reward:1.0
Step 96 - Cart X: -0.089, Cart V: 0.021, Pole A: 0.045, Pole V:0.025, Reward:1.0
Step 97 - Cart X: -0.089, Cart V: -0.175, Pole A: 0.045, Pole V:0.331,
Reward:1.0
Step 98 - Cart X: -0.092, Cart V: 0.020, Pole A: 0.052, Pole V:0.053, Reward:1.0
Step 99 - Cart X: -0.092, Cart V: -0.176, Pole A: 0.053, Pole V:0.362,
Reward:1.0
Step 100 - Cart X: -0.095, Cart V: 0.018, Pole A: 0.060, Pole V:0.086,
Reward:1.0
Step 101 - Cart X: -0.095, Cart V: -0.178, Pole A: 0.062, Pole V:0.397,
Reward:1.0
Step 102 - Cart X: -0.098, Cart V: 0.016, Pole A: 0.070, Pole V:0.125,
Reward:1.0
Step 103 - Cart X: -0.098, Cart V: -0.180, Pole A: 0.072, Pole V:0.438,
Reward:1.0
Step 104 - Cart X: -0.102, Cart V: 0.014, Pole A: 0.081, Pole V:0.169,
Reward:1.0
Step 105 - Cart X: -0.101, Cart V: -0.182, Pole A: 0.084, Pole V:0.486,
Reward:1.0
Step 106 - Cart X: -0.105, Cart V: 0.012, Pole A: 0.094, Pole V:0.221,
Reward:1.0
Step 107 - Cart X: -0.105, Cart V: -0.184, Pole A: 0.099, Pole V:0.542,
Reward:1.0
Step 108 - Cart X: -0.108, Cart V: 0.009, Pole A: 0.109, Pole V:0.282,
Reward:1.0
Step 109 - Cart X: -0.108, Cart V: -0.187, Pole A: 0.115, Pole V:0.607,
Reward:1.0
Step 110 - Cart X: -0.112, Cart V: 0.006, Pole A: 0.127, Pole V:0.353,
Reward:1.0
Step 111 - Cart X: -0.112, Cart V: -0.191, Pole A: 0.134, Pole V:0.683,
Reward:1.0
Step 112 - Cart X: -0.116, Cart V: 0.002, Pole A: 0.148, Pole V:0.435,
Reward:1.0
Step 113 - Cart X: -0.116, Cart V: -0.194, Pole A: 0.157, Pole V:0.771,
Reward:1.0
Step 114 - Cart X: -0.119, Cart V: -0.002, Pole A: 0.172, Pole V:0.531,
Reward:1.0
Step 115 - Cart X: -0.120, Cart V: -0.199, Pole A: 0.183, Pole V:0.873,
Reward:1.0
Step 116 - Cart X: -0.124, Cart V: -0.007, Pole A: 0.200, Pole V:0.642,
Reward:1.0
Step 117 - Cart X: -0.124, Cart V: -0.204, Pole A: 0.213, Pole V:0.991,
Reward:1.0
Game over! Score = 117.0
```

It's a challenge to get to 200! We could repeatedly experiment to find the best heuristics to beat the game, or we could leave all that work to the robot. Let's create an intelligence to figure this out for us.

1.3 The Theory Behind Deep Q Networks

The fundamental principle behind RL is we have two entities: the **agent** and the **environment**. The agent takes state and reward information about the envionment and chooses an action. The environment takes that action and will change to be in a new state.

RL assumes that the environment follows a Markov Decision Process (MDP). That means the state is dependent partially on the agent's actions, and partially on chance. MDPs can be represented by a graph, with states and actions as nodes, and rewards and path probabilities on the edges.

So what would be the best path through the graph above? Or perhaps a more difficult question, what would be our expected winnings if we played optimally? The probability introduced in this problem has inspired multiple strategies over the years, but all of them boil down to the idea of discounted future rewards.

Would you rather have `$100` now or `$105` a year from now? With inflation, there's no definitive answer, but each of us has a threshold that we use to determine the value of something now versus the value of something later. In psychology, this is called Delayed Gratification. Richard E. Bellman expressed this theory in an equation widely used in RL called the Bellman Equation.

Let's introduce some vocab to better define it.

Symbol	Name	Definition	Example
	agent	An entity that can act and transition between states	Us when we play CartPole
s	state	The environmental parameters describing where the agent is	The position of the cart and angle of the pole
a	action	What the agent can do within a state	Pushing the cart left or right
t	time / step	One transition between states	One push of the cart
	episode	One full simulation run	From the start of the game to game over
v, V(s)	value	How much a state is worth	V(last state dropping the pole) = 0
r, R(s, a)	reward	Value gained or lost transitioning between states through an action	R(keeping the pole up) = 1
	gamma	How much to value a current state based on a future state	Coming up soon
, (s)	policy	The recommended action to the agent based on the current state	(in trouble) = honesty

Bellman realized this: The value of our current state should the discounted value of the next state the agent will be in plus any rewards picked up along the way, given the agent takes the best action to maximize this.

Using all the symbols from above, we get:

However, this is assuming we know all the states, their corresponding actions, and their rewards. If we don't know this in advance, we can explore and simulate this equation with what is called the Q equation:

Here, the value function is replaced with the Q value, which is a function of a state and action. The learning rate is how much we want to change our old Q value with new information found during simulation. Visually, this results in a Q-table, where rows are the states, actions are the columns, and each cell is the value found through simulation.

	Meal	Snack	Wait
Hangry	1	.5	-1
Hungry	.5	1	0

	Meal	Snack	Wait
Full	-1	-.5	1.5

So this is cool and all, but how exactly does this fit in with CartPole? Here, MDPs are discrete states. CartPole has multidimensional states on a continuous scale. This is where neural networks save the day! Rather than categorize each state, we can feed state properties into our network. By having the same number of output nodes as possible actions, our network can be used to predict the value of the next state given the current state and action.

1.4 Building the Agent

These networks can be configured with the same architectures and tools as other problems, such as CNNs. However, the one gotcha is that uses a specialized loss function. We'll instead be using the derivative of the Bellman Equation. Let's go ahead and define our model function as it is in trainer/model.py

[7]:
```
def deep_q_network(
        state_shape, action_size, learning_rate, hidden_neurons):
    """Creates a Deep Q Network to emulate Q-learning.

    Creates a two hidden-layer Deep Q Network. Similar to a typical nueral
    network, the loss function is altered to reduce the difference between
    predicted Q-values and Target Q-values.

    Args:
        space_shape: a tuple of ints representing the observation space.
        action_size (int): the number of possible actions.
        learning_rate (float): the nueral network's learning rate.
        hidden_neurons (int): the number of neurons to use per hidden
            layer.
    """
    state_input = layers.Input(state_shape, name='frames')
    actions_input = layers.Input((action_size,), name='mask')

    hidden_1 = layers.Dense(hidden_neurons, activation='relu')(state_input)
    hidden_2 = layers.Dense(hidden_neurons, activation='relu')(hidden_1)
    q_values = layers.Dense(action_size)(hidden_2)
    masked_q_values = layers.Multiply()([q_values, actions_input])

    model = models.Model(
        inputs=[state_input, actions_input], outputs=masked_q_values)
    optimizer = tf.keras.optimizers.RMSprop(lr=learning_rate)
    model.compile(loss='mse', optimizer=optimizer)
    return model
```

Notice any other atypical aspects of this network?

Here, we take in both state and actions as inputs to our network. The states are fed in as normal, but the actions are used to "mask" the output. This is actually used for faster training, as we'd only want to update the nodes correspnding to the action that we simulated.

The Bellman Equation actually isn't in the network. That's because this is only the "brain" of our agent. As an intelligence, it has much more! Before we get to how exactly the agent learns, let's looks at the other aspects of its body: "Memory" and "Exploration".

Just like other neural network algorithms, we need data to train on. However, this data is the result of our simulations, not something previously stored in a table. Thus, we're going to give our agent a memory where we can store state - action - new state transitions to learn on.

Each time the agent takes a step in gym, we'll save `(state, action, reward, state_prime, done)` to our buffer, which is defined like so.

[8]:
```
class Memory():
    """Sets up a memory replay buffer for a Deep Q Network.

    A simple memory buffer for a DQN. This one randomly selects state
    transitions with uniform probability, but research has gone into
    other methods. For instance, a weight could be given to each memory
    depending on how big of a difference there is between predicted Q values
    and target Q values.

    Args:
        memory_size (int): How many elements to hold in the memory buffer.
        batch_size (int): The number of elements to include in a replay batch.
        gamma (float): The "discount rate" used to assess Q values.
    """
    def __init__(self, memory_size, batch_size, gamma):
        self.buffer = deque(maxlen=memory_size)
        self.batch_size = batch_size
        self.gamma = gamma

    def add(self, experience):
        """Adds an experience into the memory buffer.

        Args:
            experience: a (state, action, reward, state_prime, done) tuple.
        """
        self.buffer.append(experience)

    def sample(self):
        """Uniformally selects from the replay memory buffer.

        Uniformally and randomly selects experiences to train the nueral
        network on. Transposes the experiences to allow batch math on
        the experience components.

        Returns:
            (list): A list of lists with structure [
                [states], [actions], [rewards], [state_primes], [dones]
            ]
        """
        buffer_size = len(self.buffer)
        index = np.random.choice(
            np.arange(buffer_size), size=self.batch_size, replace=False)

        # Columns have different data types, so numpy array would be awkward.
        batch = np.array([self.buffer[i] for i in index]).T.tolist()
        states_mb = tf.convert_to_tensor(np.array(batch[0], dtype=np.float32))
        actions_mb = np.array(batch[1], dtype=np.int8)
        rewards_mb = np.array(batch[2], dtype=np.float32)
        states_prime_mb = np.array(batch[3], dtype=np.float32)
        dones_mb = batch[4]
        return states_mb, actions_mb, rewards_mb, states_prime_mb, dones_mb
```

Let's make a fake buffer and play around with it! We'll add the memory into our game play code to start collecting experiences.

[9]:
```
test_memory_size = 20
test_batch_size = 4
test_gamma = .9  # Unused here. For learning.

test_memory = Memory(test_memory_size, test_batch_size, test_gamma)
```

[10]:
```
actions = [x % 2 for x in range(200)]
state = env.reset()
step = 0
episode_reward = 0
done = False

while not done and step < len(actions):
    action = actions[step]  # In the future, our agents will define this.
    state_prime, reward, done, info = env.step(action)
    episode_reward += reward
    test_memory.add((state, action, reward, state_prime, done)) # New line here
    step += 1
    state = state_prime
    print_state(state, step, reward)

end_statement = "Game over!" if done else "Ran out of actions!"
print(end_statement, "Score =", episode_reward)
```

```
Step 1 - Cart X: -0.044, Cart V: -0.162, Pole A: 0.047, Pole V:0.315, Reward:1.0
Step 2 - Cart X: -0.048, Cart V: 0.033, Pole A: 0.053, Pole V:0.037, Reward:1.0
Step 3 - Cart X: -0.047, Cart V: -0.163, Pole A: 0.054, Pole V:0.346, Reward:1.0
Step 4 - Cart X: -0.050, Cart V: 0.031, Pole A: 0.061, Pole V:0.071, Reward:1.0
Step 5 - Cart X: -0.050, Cart V: -0.165, Pole A: 0.062, Pole V:0.383, Reward:1.0
Step 6 - Cart X: -0.053, Cart V: 0.029, Pole A: 0.070, Pole V:0.110, Reward:1.0
Step 7 - Cart X: -0.052, Cart V: -0.167, Pole A: 0.072, Pole V:0.424, Reward:1.0
Step 8 - Cart X: -0.056, Cart V: 0.027, Pole A: 0.081, Pole V:0.155, Reward:1.0
Step 9 - Cart X: -0.055, Cart V: -0.169, Pole A: 0.084, Pole V:0.472, Reward:1.0
Step 10 - Cart X: -0.059, Cart V: 0.025, Pole A: 0.093, Pole V:0.207, Reward:1.0
Step 11 - Cart X: -0.058, Cart V: -0.171, Pole A: 0.097, Pole V:0.528,
Reward:1.0
Step 12 - Cart X: -0.061, Cart V: 0.022, Pole A: 0.108, Pole V:0.267, Reward:1.0
Step 13 - Cart X: -0.061, Cart V: -0.174, Pole A: 0.113, Pole V:0.592,
Reward:1.0
Step 14 - Cart X: -0.064, Cart V: 0.019, Pole A: 0.125, Pole V:0.337, Reward:1.0
Step 15 - Cart X: -0.064, Cart V: -0.177, Pole A: 0.132, Pole V:0.666,
Reward:1.0
Step 16 - Cart X: -0.068, Cart V: 0.016, Pole A: 0.145, Pole V:0.418, Reward:1.0
Step 17 - Cart X: -0.067, Cart V: -0.181, Pole A: 0.154, Pole V:0.753,
Reward:1.0
Step 18 - Cart X: -0.071, Cart V: 0.012, Pole A: 0.169, Pole V:0.512, Reward:1.0
Step 19 - Cart X: -0.071, Cart V: -0.186, Pole A: 0.179, Pole V:0.853,
Reward:1.0
Step 20 - Cart X: -0.074, Cart V: 0.007, Pole A: 0.196, Pole V:0.621, Reward:1.0
Step 21 - Cart X: -0.074, Cart V: -0.190, Pole A: 0.208, Pole V:0.968,
Reward:1.0
Step 22 - Cart X: -0.078, Cart V: 0.001, Pole A: 0.228, Pole V:0.748, Reward:1.0
Game over! Score = 22.0
```

Now, let's sample the memory by running the cell below multiple times. It's different each call, and that's on purpose. Just like with other neural networks, it's important to randomly sample so that our agent can learn from many different situations.

The use of a memory buffer is called Experience Replay. The above technique of a uniform random sample is a quick and computationally efficient way to get the job done, but RL researchers often look into other sampling methods. For instance, maybe there's a way to weight memories based on their rarity or loss when the agent learns with it.

[11]:
```
test_memory.sample()
```

[11]:
```
(<tf.Tensor: id=0, shape=(4, 4), dtype=float32, numpy=
 array([[-0.05801004, -0.17125413,  0.09742869,  0.527557  ],
        [-0.07442356,  0.00676632,  0.19591288,  0.62101716],
        [-0.06098768, -0.17411232,  0.11332171,  0.5917883 ],
        [-0.06143512,  0.02237173,  0.10797983,  0.26709434]],
       dtype=float32)>,
 array([1, 0, 1, 0], dtype=int8),
 array([1., 1., 1., 1.], dtype=float32),
 array([[-0.06143512,  0.02237173,  0.10797983,  0.26709434],
        [-0.07428823, -0.19047384,  0.20833322,  0.96844834],
        [-0.06446993,  0.01925603,  0.12515749,  0.33684152],
        [-0.06098768, -0.17411232,  0.11332171,  0.5917883 ]],
       dtype=float32),
 [False, False, False, False])
```

But before the agent has any memories and has learned anything, how is it supposed to act? That comes down to Exploration vs Exploitation. The trouble is that in order to learn, risks with the unknown need to be made. There's no right answer, but there is a popular answer. We'll start by acting randomly, and over time, we will slowly decay our chance to act randomly.

Below is a partial version of the agent.

[12]:
```
class Partial_Agent():
    """Sets up a reinforcement learning agent to play in a game environment."""
    def __init__(self, network, memory, epsilon_decay, action_size):
        """Initializes the agent with DQN and memory sub-classes.

        Args:
            network: A neural network created from deep_q_network().
            memory: A Memory class object.
            epsilon_decay (float): The rate at which to decay random actions.
            action_size (int): The number of possible actions to take.
        """
        self.network = network
        self.action_size = action_size
        self.memory = memory
        self.epsilon = 1  # The chance to take a random action.
        self.epsilon_decay = epsilon_decay

    def act(self, state, training=False):
        """Selects an action for the agent to take given a game state.

        Args:
            state (list of numbers): The state of the environment to act on.
            traning (bool): True if the agent is training.

        Returns:
            (int) The index of the action to take.
        """
        if training:
            # Random actions until enough simulations to train the model.
            if len(self.memory.buffer) >= self.memory.batch_size:
                self.epsilon *= self.epsilon_decay

            if self.epsilon > np.random.rand():
                print("Exploration!")
                return random.randint(0, self.action_size-1)

        # If not acting randomly, take action with highest predicted value.
        print("Exploitation!")
        state_batch = np.expand_dims(state, axis=0)
        predict_mask = np.ones((1, self.action_size,))
        action_qs = self.network.predict([state_batch, predict_mask])
        return np.argmax(action_qs[0])
```

Let's define the agent and get a starting state to see how it would act without any training.

[13]:
```
state = env.reset()

# Define "brain"
space_shape = env.observation_space.shape
action_size = env.action_space.n

# Feel free to play with these
test_learning_rate = .2
test_hidden_neurons = 10
test_epsilon_decay = .95

test_network = deep_q_network(
    space_shape, action_size, test_learning_rate, test_hidden_neurons)

test_agent = Partial_Agent(
    test_network, test_memory, test_epsilon_decay, action_size)
```

Run the cell below multiple times. Since we're decaying the random action rate after every action, it's only a matter a time before the agent exploits more than it explores.

[14]:
```
action = test_agent.act(state, training=True)
print("Push Right" if action else "Push Left")
```

```
Exploration!
Push Left
```

Memories, a brain, and a healthy dose of curiosity. We finally have all the ingredient for our agent to learn. After all, as the Scarecrow from the Wizard of Oz said:

"Everything in life is unusual until you get accustomed to it."
~L. Frank Baum

Below is the code used by our agent to learn, where the Bellman Equation at last makes an appearance. We'll run through the following steps.

1. Pull a batch from memory
2. Get the Q value (the output of the neural network) based on the memory's ending state
    - Assume the Q value of the action with the highest Q value (test all actions)
3. Update these Q values with the Bellman Equation
    - `target_qs = (next_q_mb * self.memory.gamma) + reward_mb`
    - If the state is the end of the game, set the target_q to the reward for entering the final state.
4. Reshape the target_qs to match the networks output

   - Only learn on the memory's corresponding action by setting all action nodes to zero besides the action node taken.
5. Fit Target Qs as the label to our model against the memory's starting state and action as the inputs.

[15]:
```
def learn(self):
    """Trains the Deep Q Network based on stored experiences."""
    batch_size = self.memory.batch_size
    if len(self.memory.buffer) < batch_size:
        return None

    # Obtain random mini-batch from memory.
    state_mb, action_mb, reward_mb, next_state_mb, done_mb = (
        self.memory.sample())

    # Get Q values for next_state.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([next_state_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
    target_qs = (next_q_mb * self.memory.gamma) + reward_mb
    target_qs = tf.where(done_mb, reward_mb, target_qs)

    # Match training batch to network output:
    # target_q where action taken, 0 otherwise.
    action_mb = tf.convert_to_tensor(action_mb, dtype=tf.int32)
    action_hot = tf.one_hot(action_mb, self.action_size)
    target_mask = tf.multiply(tf.expand_dims(target_qs, -1), action_hot)

    return self.network.train_on_batch(
        [state_mb, action_hot], target_mask, reset_metrics=False
    )

Partial_Agent.learn = learn
test_agent = Partial_Agent(
    test_network, test_memory, test_epsilon_decay, action_size)
```

Nice! We finally have an intelligence that can walk and talk and… well ok, this intelligence is too simple to be able to do those things, but maybe it can learn to push a cart with a pole on it. Let's update our training loop to use our new agent.

Run the below cell over and over up to ten times to train the agent.

[16]:
```
state = env.reset()
step = 0
episode_reward = 0
done = False
```

```
while not done:
    action = test_agent.act(state, training=True)
    state_prime, reward, done, info = env.step(action)
    episode_reward += reward
    test_agent.memory.add((state, action, reward, state_prime, done)) # New␣
 ↪line here
    step += 1
    state = state_prime
    print_state(state, step, reward)

print(test_agent.learn())
print("Game over! Score =", episode_reward)
```

```
Exploration!
Step 1 - Cart X: -0.038, Cart V: 0.221, Pole A: 0.038, Pole V:-0.281, Reward:1.0
Exploration!
Step 2 - Cart X: -0.034, Cart V: 0.025, Pole A: 0.032, Pole V:0.023, Reward:1.0
Exploration!
Step 3 - Cart X: -0.033, Cart V: -0.170, Pole A: 0.033, Pole V:0.326, Reward:1.0
Exploration!
Step 4 - Cart X: -0.037, Cart V: -0.366, Pole A: 0.039, Pole V:0.629, Reward:1.0
Exploration!
Step 5 - Cart X: -0.044, Cart V: -0.171, Pole A: 0.052, Pole V:0.349, Reward:1.0
Exploration!
Step 6 - Cart X: -0.048, Cart V: 0.023, Pole A: 0.059, Pole V:0.073, Reward:1.0
Exploration!
Step 7 - Cart X: -0.047, Cart V: -0.173, Pole A: 0.060, Pole V:0.383, Reward:1.0
Exploitation!
Step 8 - Cart X: -0.051, Cart V: -0.369, Pole A: 0.068, Pole V:0.694, Reward:1.0
Exploitation!
Step 9 - Cart X: -0.058, Cart V: -0.565, Pole A: 0.082, Pole V:1.008, Reward:1.0
Exploration!
Step 10 - Cart X: -0.069, Cart V: -0.761, Pole A: 0.102, Pole V:1.325,
Reward:1.0
Exploitation!
Step 11 - Cart X: -0.084, Cart V: -0.957, Pole A: 0.128, Pole V:1.647,
Reward:1.0
Exploitation!
Step 12 - Cart X: -0.104, Cart V: -1.154, Pole A: 0.161, Pole V:1.977,
Reward:1.0
Exploitation!
Step 13 - Cart X: -0.127, Cart V: -1.350, Pole A: 0.201, Pole V:2.315,
Reward:1.0
Exploitation!
Step 14 - Cart X: -0.154, Cart V: -1.546, Pole A: 0.247, Pole V:2.662,
Reward:1.0
0.54204845
```

```
Game over! Score = 14.0
```

1.5 Hypertuning

Chances are, at this point, the agent is having a tough time learning. Why is that? Well, remember that hyperparameter tuning job we kicked off at the start of this notebook?

The are many parameters that need adjusting with our agent. Let's recap: * The number of `episodes` or full runs of the game to train on * The neural networks `learning_rate` * The number of `hidden_neurons` to use in our network * `gamma`, or how much we want to discount the future value of states * How quickly we want to switch from explore to exploit with `explore_decay` * The size of the memory buffer, `memory_size` * The number of memories to pull from the buffer when training, `memory_batch_size`

These all have been added as flags to pass to the model in `trainer/trainer.py`'s `_parse_arguments` method. For the most part, `trainer/trainer.py` follows the structure of the training loop that we have above, but it does have a few extra bells and whistles, like a hook into TensorBoard and video output.

[17]:
```python
def _parse_arguments(argv):
    """Parses command-line arguments."""
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--game',
        help='Which open ai gym game to play',
        type=str,
        default='CartPole-v0')
    parser.add_argument(
        '--episodes',
        help='The number of episodes to simulate',
        type=int,
        default=200)
    parser.add_argument(
        '--learning_rate',
        help='Learning rate for the nueral network',
        type=float,
        default=0.2)
    parser.add_argument(
        '--hidden_neurons',
        help='The number of nuerons to use per layer',
        type=int,
        default=30)
    parser.add_argument(
        '--gamma',
        help='The gamma or "discount" factor to discount future states',
        type=float,
        default=0.5)
    parser.add_argument(
        '--explore_decay',
        help='The rate at which to decay the probability of a random action',
        type=float,
        default=0.1)
    parser.add_argument(
        '--memory_size',
        help='Size of the memory buffer',
        type=int,
        default=100000)
    parser.add_argument(
        '--memory_batch_size',
        help='The amount of memories to sample from the buffer while training',
        type=int,
        default=8)
    parser.add_argument(
        '--job-dir',
        help='Directory where to save the given model',
        type=str,
        default='models/')
    parser.add_argument(
        '--print_rate',
        help='How often to print the score, 0 if never',
        type=int,
        default=0)
    parser.add_argument(
        '--eval_rate',
        help="""While training, perform an on-policy simulation and record
        metrics to tensorboard every <record_rate> steps, 0 if never. Use
        higher values to avoid hyperparameter tuning "too many metrics"
        error""",
        type=int,
        default=20)
    return parser.parse_known_args(argv)
```

Geez, that's a lot. And like with other machine learning methods, there's no hard and fast rule and is problem dependent. Plus, there are many more paramaters we could explore, like the number of layers, learning rate decay, and so on.

We can tell Google Cloud how to explore the hyperparameter tuning space with a config file. The `hyperparam.yaml` file in this directory is exactly that. It specifies which parameter to tune on (in this case, the `episode_reward`) and the range for the different flags we want to tune on.

In our code, we'll add the following

```
import hypertune  #From cloudml-hypertune library

hpt = hypertune.HyperTune()  # Initialized before looping through episodes

# Placed right before the end of the training loop hpt.report_hyperparameter_tuning_metric(
hyperparameter_metric_tag='episode_reward',     metric_value=reward,
global_step=episode)
```

This way, at the end of every episode, we can send information to the tuning service on how the agent is doing. The service can only handle so much information being thrown at it at once, so we'll add a `eval_rate` flag to throttle information to every `eval_rate` episodes.

It is definately a worthwhile exercise to try and find the optimal set of parameters on one's on, but if life is too short, and there isn't time for that, the hyperparameter tuning job should now be complete. Head on over to Google Cloud's AI Platform to see the job labeled `dqn_on_gcp_<time_this_lab_was_started>`

Click on the job name to see the results. Information comes in as each trial is complete, and the best performing trial will be listed on the top.

Logs can be invaluable when debugging. Click the three dots to the right of one of the trials to filter logs by that particular trial.

At last, let's see the results of the best trial. Keep in mind the best trial number and navigate over to your bucket. The results will be in a file with the same Job Name as your hyperparameter tuning job. In that folder, there will be a number of subfolders equal to the number of hyperparameter tuning trials. Select the folder with your best performing `Trial Id`

There should be a number of goodies in the file including TensorBoard information in `/train`, a saved model in `saved_model.pb`, and a recording of the model in `recording.mp4`.

Open the Google Cloud Shell and run Tensorboard with

```
tensorboard --logdir=gs://<your-bucket>/<job-name>/<path-best-trial>
```

The episode rewards and training loss are displayed for the trial in intervals of 20 episodes.

Click `recording.mp4` in your bucket to visually see how the model performed! How did it do? If you're not proud of your little robot, check out the recordings of the other trials to see how it decimates the competition.

Congratulations on making a Deep Q Agent! That's it for now, but this is just scratching the surface for Reinforcement Learning. AI Gym has plenty of other environments, see if you can conquer them with your new skills!

[ ]: