a2c_on_gcp

June 26, 2024

1 Policy Gradients and A2C

In the previous notebook, we learned how to use hyperparameter tuning to help DQN agents balance a pole on a cart. In this notebook, we'll explore two other types of alogrithms: Policy Gradients and A2C.

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.

[ ]:
```
!pip install tensorflow==2.5
!pip install numpy==1.21.2
```

Note: Please ignore any incompatibility warnings and errors. Restart the kernel if required.

[ ]:
```
%%bash
BUCKET=<your-bucket-here> # Change to your bucket name
JOB_NAME=pg_on_gcp_$(date -u +%y%m%d_%H%M%S)
REGION='us-central1' # Change to your bucket region
IMAGE_URI=gcr.io/cloud-training-prod-bucket/pg:latest

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

[ ]:
```
!pip install gym==0.12.5 --user
```

Note: Restart the kernel if the above libraries needed to be installed

Thankfully, we can use the same environment for these algorithms as DQN, so this notebook will focus less on the operational work of feeding our agents the data, and more on the theory behind these algorthims. Let's start by loading our libraries and environment.

[1]:
```
import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras import backend as K

CLIP_EDGE = 1e-8

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))

env = gym.make('CartPole-v0')
```

1.2 The Theory Behind Policy Gradients

Whereas Q-learning attempts to assign each state a value, Policy Gradients tries to find actions directly, increasing or decreaing a chance to take an action depending on how an episode plays out.

To compare, Q-learning has a table that keeps track of the value of each combination of state and action:

	Meal	Snack	Wait
Hangry	1	.5	-1
Hungry	.5	1	0
Full	-1	-.5	1.5

Instead for Policy Gradients, we can imagine that we have a similar table, but instead of recording the values, we'll keep track of the probability to take the column action given the row state.

	Meal	Snack	Wait
Hangry	70%	20%	10%
Hungry	30%	50%	20%
Full	5%	15%	80%

With Q learning, whenever we take one step in our environment, we can update the value of the old state based on the value of the new state plus any rewards we picked up based on the Q equation:

Could we do the same thing if we have a table of probabilities instead values? No, because we don't have a way to calculate the value of each state from our table. Instead, we'll use a different Temporal Difference Learning strategy.

Q Learning is an evolution of TD(0), and for Policy Gradients, we'll use TD(1). We'll calculate TD(1) accross and entire episode, and use that to indicate whether to increase or decrease the probability correspoding to the action we took. Let's look at a full day of eating.

Hour	State	Action	Reward
9	Hangry	Wait	-.9
10	Hangry	Meal	1.2
11	Full	Wait	.5
12	Full	Snack	-.6
13	Full	Wait	1
14	Full	Wait	.6
15	Full	Wait	.2
16	Hungry	Wait	0
17	Hungry	Meal	.4
18	Full	Wait	.5

We'll work backwards from the last day, using the same discount, or `gamma`, as we did with DQNs. The `total_rewards` variable is equivalent to the value of state prime. Using the Bellman Equation, everytime we calculate the value of a state, st, we'll set that as the value of state prime for the state before, st-1.

[2]:
```
test_gamma = .5  # Please change me to be between zero and one
episode_rewards = [-.9, 1.2, .5, -.6, 1, .6, .2, 0, .4, .5]

def discount_episode(rewards, gamma):
    discounted_rewards = np.zeros_like(rewards)
    total_rewards = 0
    for t in reversed(range(len(rewards))):
        total_rewards = rewards[t] + total_rewards * gamma
        discounted_rewards[t] = total_rewards
    return discounted_rewards

discount_episode(episode_rewards, test_gamma)
```

[2]:
```
array([-0.16308594,  1.47382812,  0.54765625,  0.0953125 ,  1.390625  ,
        0.78125   ,  0.3625    ,  0.325     ,  0.65      ,  0.5       ])
```

Wherever our discounted reward is positive, we'll increase the probability corresponding to the action we took. Similarly, wherever our discounted reward is negative, we'll decrease the probabilty.

However, with this strategy, any actions with a positive reward will have it's probability increase, not necessarily the most optimal action. This puts us in a feedback loop, where we're more likely to pick less optimal actions which could further increase their probability. To counter this, we'll divide the size of our increases by the probability to choose the corresponding action, which will slow the growth of popular actions to give other actions a chance.

Here is our update rule for our neural network, where alpha is our learning rate, and pi is our optimal policy, or the probability to take the optimal action, a*, given our current state, s.

Doing some fancy calculus, we can combine the numerator and denominator with a log function. Since it's not clear what the optimal action is, we'll instead use our discounted rewards, or G, to increase or decrease the weights of the respective action the agent took. A full breakdown of the math can be found in this article by Chris Yoon.

Below is what it looks like in code. `y_true` is the one-hot encoding of the action that was taken. `y_pred` is the probabilty to take each action given the state the agent was in.

[3]:
```
def custom_loss(y_true, y_pred):
    y_pred_clipped = K.clip(y_pred, CLIP_EDGE, 1-CLIP_EDGE)
    log_likelihood = y_true * K.log(y_pred_clipped)
    return K.sum(-log_likelihood*g)
```

We won't have the discounted rewards, or `g`, when our agent is acting in the environment. No problem, we'll have one neural network with two types of pathways. One pathway, `predict`, will be the probability to take an action given an inputed state. It's only used for prediction and is not used for backpropogation. The other pathway, `policy`, will take both a state and a discounted reward, so it can be used for training.

The code in its entirety looks like this. As with Deep Q Networks, the hidden layers of a Policy Gradient can use a CNN if the input state is pixels, but the last layer is typically a Dense layer with a Softmax activation function to convert the output into probabilities.

[4]:
```
def build_networks(
        state_shape, action_size, learning_rate, hidden_neurons):
    """Creates a Policy Gradient Neural Network.

    Creates a two hidden-layer Policy Gradient Neural Network. The loss
    function is altered to be a log-likelihood function weighted
    by the discounted reward, g.

    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')
    g = layers.Input((1,), name='g')

    hidden_1 = layers.Dense(hidden_neurons, activation='relu')(state_input)
    hidden_2 = layers.Dense(hidden_neurons, activation='relu')(hidden_1)
    probabilities = layers.Dense(action_size, activation='softmax')(hidden_2)

    def custom_loss(y_true, y_pred):
        y_pred_clipped = K.clip(y_pred, CLIP_EDGE, 1-CLIP_EDGE)
        log_lik = y_true*K.log(y_pred_clipped)
        return K.sum(-log_lik*g)

    policy = models.Model(
        inputs=[state_input, g], outputs=[probabilities])
    optimizer = tf.keras.optimizers.Adam(lr=learning_rate)
```

```
    policy.compile(loss=custom_loss, optimizer=optimizer)

    predict = models.Model(inputs=[state_input], outputs=[probabilities])
    return policy, predict
```

Let's get a taste of how these networks function. Run the below cell to build our test networks.

[5]:
```
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_policy, test_predict = build_networks(
    space_shape, action_size, test_learning_rate, test_hidden_neurons)
```

We can't use the policy network until we build our learning function, but we can feed a state to the predict network so we can see our chances to pick our actions.

[6]:
```
state = env.reset()
test_predict.predict(np.expand_dims(state, axis=0))
```

[6]:
```
array([[0.49907038, 0.5009296 ]], dtype=float32)
```

Right now, the numbers should be close to `[.5, .5]`, with a little bit of variance due to the randomization of initializing the weights and the cart's starting position. In order to train, we'll need some memories to train on. The memory buffer here is simpler than DQN, as we don't have to worry about random sampling. We'll clear the buffer every time we train as we'll only hold one episode's worth of memory.

[7]:
```
class Memory():
    """Sets up a memory replay buffer for Policy Gradient methods.

    Args:
        gamma (float): The "discount rate" used to assess TD(1) values.
    """
    def __init__(self, gamma):
        self.buffer = []
        self.gamma = gamma

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

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

```
    def sample(self):
        """Returns the list of episode experiences and clears the buffer.

        Returns:
            (list): A tuple of lists with structure (
                [states], [actions], [rewards]
            }
        """
        batch = np.array(self.buffer).T.tolist()
        states_mb = 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)
        self.buffer = []
        return states_mb, actions_mb, rewards_mb
```

Let's make a fake buffer to get a sense of the data we'll be training on. The cell below initializes our memory and runs through one episode of the game by alternating pushing the cart left and right.

Try running it to see the data we'll be using for training.

[8]:
```
test_memory = Memory(test_gamma)
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))
    step += 1
    state = state_prime

test_memory.sample()
```

[8]:
```
(array([[-1.02426745e-02,  1.07908761e-02, -9.08028707e-03,
         -4.79757078e-02],
        [-1.00268573e-02, -1.84199706e-01, -1.00398017e-02,
          2.41828531e-01],
        [-1.37108508e-02,  1.10642128e-02, -5.20323077e-03,
         -5.40042296e-02],
        [-1.34895667e-02, -1.83982745e-01, -6.28331536e-03,
          2.37032503e-01],
        [-1.71692222e-02,  1.12284059e-02, -1.54266518e-03,
         -5.76257259e-02],
```

```
[-1.69446543e-02, -1.83871388e-01, -2.69517978e-03,
  2.34570086e-01],
[-2.06220821e-02,  1.12889633e-02,  1.99622195e-03,
 -5.89617714e-02],
[-2.03963015e-02, -1.83861554e-01,  8.16986489e-04,
  2.34350309e-01],
[-2.40735337e-02,  1.12487162e-02,  5.50399255e-03,
 -5.80748022e-02],
[-2.38485578e-02, -1.83951721e-01,  4.34249640e-03,
  2.36339584e-01],
[-2.75275931e-02,  1.11079235e-02,  9.06928815e-03,
 -5.49704358e-02],
[-2.73054354e-02, -1.84142888e-01,  7.96987955e-03,
  2.40560070e-01],
[-3.09882928e-02,  1.08643146e-02,  1.27810808e-02,
 -4.95983213e-02],
[-3.07710059e-02, -1.84438556e-01,  1.17891142e-02,
  2.47089580e-01],
[-3.44597772e-02,  1.05130617e-02,  1.67309064e-02,
 -4.18515950e-02],
[-3.42495143e-02, -1.84844762e-01,  1.58938747e-02,
  2.56062776e-01],
[-3.79464105e-02,  1.00467019e-02,  2.10151300e-02,
 -3.15648839e-02],
[-3.77454758e-02, -1.85370207e-01,  2.03838330e-02,
  2.67673761e-01],
[-4.14528809e-02,  9.45498701e-03,  2.57373080e-02,
 -1.85109023e-02],
[-4.12637815e-02, -1.86026424e-01,  2.53670886e-02,
  2.82180041e-01],
[-4.49843109e-02,  8.72467831e-03,  3.10106911e-02,
 -2.39550718e-03],
[-4.48098183e-02, -1.86827973e-01,  3.09627801e-02,
  2.99908131e-01],
[-4.85463776e-02,  7.83927832e-03,  3.69609408e-02,
  1.71488058e-02],
[-4.83895913e-02, -1.87792704e-01,  3.73039171e-02,
  3.21260422e-01],
[-5.21454439e-02,  6.77869981e-03,  4.37291265e-02,
  4.05711532e-02],
[-5.20098694e-02, -1.88942149e-01,  4.45405506e-02,
  3.46724033e-01],
[-5.57887144e-02,  5.51887788e-03,  5.14750294e-02,
  6.84123784e-02],
[-5.56783378e-02, -1.90301836e-01,  5.28432801e-02,
  3.76881361e-01],
[-5.94843738e-02,  4.03133035e-03,  6.03809059e-02,
```

```
          1.01317212e-01],
        [-5.94037473e-02, -1.91901654e-01,  6.24072514e-02,
          4.12422299e-01],
        [-6.32417798e-02,  2.28268700e-03,  7.06556961e-02,
          1.40048638e-01],
        [-6.31961226e-02, -1.93776354e-01,  7.34566674e-02,
          4.54158932e-01],
        [-6.70716539e-02,  2.34215331e-04,  8.25398490e-02,
          1.85504705e-01],
        [-6.70669675e-02, -1.95965752e-01,  8.62499401e-02,
          5.03041863e-01],
        [-7.09862858e-02, -2.15859618e-03,  9.63107795e-02,
          2.38737836e-01],
        [-7.10294545e-02, -1.98515058e-01,  1.01085536e-01,
          5.60179174e-01],
        [-7.49997571e-02, -4.94630216e-03,  1.12289116e-01,
          3.00976813e-01],
        [-7.50986859e-02, -2.01474681e-01,  1.18308656e-01,
          6.26856506e-01],
        [-7.91281760e-02, -8.18545092e-03,  1.30845785e-01,
          3.73651028e-01],
        [-7.92918876e-02, -2.04899773e-01,  1.38318807e-01,
          7.04559207e-01],
        [-8.33898783e-02, -1.19379535e-02,  1.52409986e-01,
          4.58417088e-01],
        [-8.36286396e-02, -2.08848774e-01,  1.61578327e-01,
          7.94994712e-01],
        [-8.78056139e-02, -1.62694640e-02,  1.77478224e-01,
          5.57185948e-01],
        [-8.81310031e-02, -2.13380575e-01,  1.88621938e-01,
          9.00113404e-01],
        [-9.23986137e-02, -2.12461017e-02,  2.06624210e-01,
          6.72149956e-01]], dtype=float32),
 array([0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
        0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
        0], dtype=int8),
 array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32))
```

Ok, time to start putting together the agent! Let's start by giving it the ability to act. Here, we don't need to worry about exploration vs exploitation because we already have a random chance to take each of our actions. As the agent learns, it will naturally shift from exploration to exploitation. How conveient!

[9]:
```
class Partial_Agent():
    """Sets up a reinforcement learning agent to play in a game environment."""
```

```
    def __init__(self, policy, predict, memory, action_size):
        """Initializes the agent with Policy Gradient networks
            and memory sub-classes.

        Args:
            policy: The policy network created from build_networks().
            predict: The predict network created from build_networks().
            memory: A Memory class object.
            action_size (int): The number of possible actions to take.
        """
        self.policy = policy
        self.predict = predict
        self.action_size = action_size
        self.memory = memory

    def act(self, state):
        """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.

        Returns:
            (int) The index of the action to take.
        """
        # If not acting randomly, take action with highest predicted value.
        state_batch = np.expand_dims(state, axis=0)
        probabilities = self.predict.predict(state_batch)[0]
        action = np.random.choice(self.action_size, p=probabilities)
        return action
```

Let's see the act function in action. First, let's build our agent.

[10]:
```
test_agent = Partial_Agent(test_policy, test_predict, test_memory, action_size)
```

Next, run the below cell a few times to test the `act` method. Is it about a 50/50 chance to push right instead of left?

[11]:
```
action = test_agent.act(state)
print("Push Right" if action else "Push Left")
```

```
Push Right
```

Now for the most important part. We need to give our agent a way to learn! To start, we'll one-hot encode our actions. Since the output of our network is a probability for each action, we'll have a 1 corresponding to the action that was taken and 0's for the actions we didn't take.

That doesn't give our agent enough information on whether the action that was taken was actually a good idea, so we'll also use our `discount_episode` to calculate the TD(1) value of each step within the episode.

One thing to note, is that CartPole doesn't have any negative rewards, meaning, even if it does terribly, the agent will still think the run went well. To help counter this, we'll take the mean and standard deviation of our discounted rewards, or `discount_mb`, and use that to find the Standard Score for each discounted reward. With this, steps close to dropping the poll will have a negative reward.

[12]:
```
def learn(self, print_variables=False):
    """Trains a Policy Gradient policy network based on stored experiences."""
    state_mb, action_mb, reward_mb = self.memory.sample()
    # One hot enocde actions
    actions = np.zeros([len(action_mb), self.action_size])
    actions[np.arange(len(action_mb)), action_mb] = 1
    if print_variables:
        print("action_mb:", action_mb)
        print("actions:", actions)

    # Apply TD(1) and normalize
    discount_mb = discount_episode(reward_mb, self.memory.gamma)
    discount_mb = (discount_mb - np.mean(discount_mb)) / np.std(discount_mb)
    if print_variables:
        print("reward_mb:", reward_mb)
        print("discount_mb:", discount_mb)
    return self.policy.train_on_batch([state_mb, discount_mb], actions)

Partial_Agent.learn = learn
test_agent = Partial_Agent(test_policy, test_predict, test_memory, action_size)
```

Try adding in some print statements to the code above to get a sense of how the data is transformed before feeding it into the model, then run the below code to see it in action.

Finally, it's time to put it all together. Policy Gradient Networks have less hypertuning parameters than DQNs, but since our custom loss constructs a TensorFlow Graph under the hood, we'll set up lazy execution by wrapping our traing steps in a default graph.

By changing `test_gamma`, `test_learning_rate`, and `test_hidden_neurons`, can you help the agent reach a score of 200 within 200 episodes? It takes a little bit of thinking and a little bit of luck.

Hover the curser on this bold text to see a solution to the challenge.

[14]:
```
test_gamma = .5
test_learning_rate = .01
test_hidden_neurons = 100

with tf.Graph().as_default():
    test_memory = Memory(test_gamma)
    test_policy, test_predict = build_networks(
    space_shape, action_size, test_learning_rate, test_hidden_neurons)
```

```
    test_agent = Partial_Agent(test_policy, test_predict, test_memory,␣
↪action_size)
    for episode in range(200):
        state = env.reset()
        episode_reward = 0
        done = False

        while not done:
            action = test_agent.act(state)
            state_prime, reward, done, info = env.step(action)
            episode_reward += reward
            test_agent.memory.add((state, action, reward))
            state = state_prime

        test_agent.learn()
        print("Episode", episode, "Score =", episode_reward)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-
packages/tensorflow_core/python/ops/resource_variable_ops.py:1630: calling
BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops)
with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
Episode 0 Score = 11.0
Episode 1 Score = 11.0
Episode 2 Score = 24.0
Episode 3 Score = 27.0
Episode 4 Score = 24.0
Episode 5 Score = 30.0
Episode 6 Score = 24.0
Episode 7 Score = 30.0
Episode 8 Score = 27.0
Episode 9 Score = 33.0
Episode 10 Score = 36.0
Episode 11 Score = 40.0
Episode 12 Score = 34.0
Episode 13 Score = 32.0
Episode 14 Score = 99.0
Episode 15 Score = 61.0
Episode 16 Score = 70.0
Episode 17 Score = 46.0
Episode 18 Score = 90.0
Episode 19 Score = 125.0
Episode 20 Score = 132.0
Episode 21 Score = 154.0
Episode 22 Score = 200.0
Episode 23 Score = 200.0
Episode 24 Score = 200.0
```

```
Episode 25 Score = 200.0
Episode 26 Score = 93.0
Episode 27 Score = 188.0
Episode 28 Score = 134.0
Episode 29 Score = 126.0
Episode 30 Score = 157.0
Episode 31 Score = 169.0
Episode 32 Score = 161.0
Episode 33 Score = 128.0
Episode 34 Score = 115.0
Episode 35 Score = 110.0
Episode 36 Score = 167.0
Episode 37 Score = 200.0
Episode 38 Score = 102.0
Episode 39 Score = 109.0
Episode 40 Score = 61.0
Episode 41 Score = 32.0
Episode 42 Score = 14.0
Episode 43 Score = 18.0
Episode 44 Score = 10.0
Episode 45 Score = 11.0
Episode 46 Score = 10.0
Episode 47 Score = 12.0
Episode 48 Score = 17.0
Episode 49 Score = 10.0
Episode 50 Score = 8.0
Episode 51 Score = 9.0
Episode 52 Score = 13.0
Episode 53 Score = 10.0
Episode 54 Score = 9.0
Episode 55 Score = 8.0
Episode 56 Score = 10.0
Episode 57 Score = 9.0
Episode 58 Score = 10.0
Episode 59 Score = 10.0
Episode 60 Score = 10.0
Episode 61 Score = 10.0
Episode 62 Score = 10.0
Episode 63 Score = 9.0
Episode 64 Score = 10.0
Episode 65 Score = 8.0
Episode 66 Score = 10.0
Episode 67 Score = 10.0
Episode 68 Score = 9.0
Episode 69 Score = 9.0
Episode 70 Score = 10.0
Episode 71 Score = 9.0
Episode 72 Score = 9.0
```

```
Episode 73 Score = 10.0
Episode 74 Score = 8.0
Episode 75 Score = 11.0
Episode 76 Score = 10.0
Episode 77 Score = 8.0
Episode 78 Score = 10.0
Episode 79 Score = 9.0
Episode 80 Score = 10.0
Episode 81 Score = 9.0
Episode 82 Score = 9.0
Episode 83 Score = 10.0
Episode 84 Score = 8.0
Episode 85 Score = 10.0
Episode 86 Score = 10.0
Episode 87 Score = 9.0
Episode 88 Score = 9.0
Episode 89 Score = 10.0
Episode 90 Score = 9.0
Episode 91 Score = 10.0
Episode 92 Score = 9.0
Episode 93 Score = 9.0
Episode 94 Score = 10.0
Episode 95 Score = 9.0
Episode 96 Score = 11.0
Episode 97 Score = 8.0
Episode 98 Score = 10.0
Episode 99 Score = 8.0
Episode 100 Score = 10.0
Episode 101 Score = 10.0
Episode 102 Score = 9.0
Episode 103 Score = 9.0
Episode 104 Score = 8.0
Episode 105 Score = 10.0
Episode 106 Score = 11.0
Episode 107 Score = 8.0
Episode 108 Score = 10.0
Episode 109 Score = 9.0
Episode 110 Score = 9.0
Episode 111 Score = 9.0
Episode 112 Score = 9.0
Episode 113 Score = 10.0
Episode 114 Score = 11.0
Episode 115 Score = 9.0
Episode 116 Score = 9.0
Episode 117 Score = 8.0
Episode 118 Score = 9.0
Episode 119 Score = 10.0
Episode 120 Score = 10.0
```

```
Episode 121 Score = 9.0
Episode 122 Score = 9.0
Episode 123 Score = 9.0
Episode 124 Score = 9.0
Episode 125 Score = 9.0
Episode 126 Score = 8.0
Episode 127 Score = 8.0
Episode 128 Score = 8.0
Episode 129 Score = 8.0
Episode 130 Score = 10.0
Episode 131 Score = 9.0
Episode 132 Score = 9.0
Episode 133 Score = 8.0
Episode 134 Score = 9.0
Episode 135 Score = 10.0
Episode 136 Score = 9.0
Episode 137 Score = 8.0
Episode 138 Score = 8.0
Episode 139 Score = 10.0
Episode 140 Score = 10.0
Episode 141 Score = 10.0
Episode 142 Score = 9.0
Episode 143 Score = 9.0
Episode 144 Score = 10.0
Episode 145 Score = 9.0
Episode 146 Score = 10.0
Episode 147 Score = 9.0
Episode 148 Score = 9.0
Episode 149 Score = 9.0
Episode 150 Score = 10.0
Episode 151 Score = 9.0
Episode 152 Score = 9.0
Episode 153 Score = 8.0
Episode 154 Score = 9.0
Episode 155 Score = 10.0
Episode 156 Score = 9.0
Episode 157 Score = 8.0
Episode 158 Score = 9.0
Episode 159 Score = 9.0
Episode 160 Score = 10.0
Episode 161 Score = 10.0
Episode 162 Score = 9.0
Episode 163 Score = 10.0
Episode 164 Score = 9.0
Episode 165 Score = 8.0
Episode 166 Score = 10.0
Episode 167 Score = 9.0
Episode 168 Score = 10.0
```

```
Episode 169 Score = 10.0
Episode 170 Score = 10.0
Episode 171 Score = 8.0
Episode 172 Score = 10.0
Episode 173 Score = 9.0
Episode 174 Score = 9.0
Episode 175 Score = 9.0
Episode 176 Score = 10.0
Episode 177 Score = 9.0
Episode 178 Score = 9.0
Episode 179 Score = 9.0
Episode 180 Score = 9.0
Episode 181 Score = 9.0
Episode 182 Score = 10.0
Episode 183 Score = 9.0
Episode 184 Score = 9.0
Episode 185 Score = 10.0
Episode 186 Score = 10.0
Episode 187 Score = 8.0
Episode 188 Score = 9.0
Episode 189 Score = 9.0
Episode 190 Score = 8.0
Episode 191 Score = 8.0
Episode 192 Score = 10.0
Episode 193 Score = 9.0
Episode 194 Score = 10.0
Episode 195 Score = 10.0
Episode 196 Score = 9.0
Episode 197 Score = 9.0
Episode 198 Score = 9.0
Episode 199 Score = 10.0
```

2 The Theory Behind Actor - Critic

Now that we have the hang of Policy Gradients, let's combine this strategy with Deep Q Agents. We'll have one architecture to rule them all!

Below is the setup for our neural networks. There are plenty of ways to go combining the two strategies. We'll be focusing on one varient called A2C, or Advantage Actor Critic.

Here's the philosophy: We'll use our critic pathway to estimate the value of a state, or V(s). Given a state-action-new state transition, we can use our critic and the Bellman Equation to calculate the discounted value of the new state, or r + * V(s').

Like DQNs, this discounted value is the label the critic will train on. While that is happening, we can subtract V(s) and the discounted value of the new state to get the advantage, or A(s,a). In human terms, how much value was the action the agent took? This is what the actor, or the policy gradient portion or our network, will train on.

Too long, didn't read: the critic's job is to learn how to asses the value of a state. The actor's job is to assign probabilities to it's available actions such that it increases its chance to move into a higher valued state.

Below is our new `build_networks` function. Each line has been tagged with whether it comes from Deep Q Networks (`# DQN`), Policy Gradients (`# PG`), or is something new (`# New`).

[15]:
```
def build_networks(state_shape, action_size, learning_rate, critic_weight,
                   hidden_neurons, entropy):
    """Creates Actor Critic Neural Networks.

    Creates a two hidden-layer Policy Gradient Neural Network. The loss
    function is altered to be a log-likelihood function weighted
    by an action's advantage.

    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.
        critic_weight (float): how much to weigh the critic's training loss.
        hidden_neurons (int): the number of neurons to use per hidden layer.
        entropy (float): how much to enourage exploration versus exploitation.
    """
    state_input = layers.Input(state_shape, name='frames')
    advantages = layers.Input((1,), name='advantages')  # PG, A instead of G

    # PG
    actor_1 = layers.Dense(hidden_neurons, activation='relu')(state_input)
    actor_2 = layers.Dense(hidden_neurons, activation='relu')(actor_1)
    probabilities = layers.Dense(action_size, activation='softmax')(actor_2)

    # DQN
    critic_1 = layers.Dense(hidden_neurons, activation='relu')(state_input)
    critic_2 = layers.Dense(hidden_neurons, activation='relu')(critic_1)
    values = layers.Dense(1, activation='linear')(critic_2)

    def actor_loss(y_true, y_pred):  # PG
        y_pred_clipped = K.clip(y_pred, CLIP_EDGE, 1-CLIP_EDGE)
        log_lik = y_true*K.log(y_pred_clipped)
        entropy_loss = y_pred * K.log(K.clip(y_pred, CLIP_EDGE, 1-CLIP_EDGE)) ␣
↪# New
        return K.sum(-log_lik * advantages) - (entropy * K.sum(entropy_loss))

    # Train both actor and critic at the same time.
    actor = models.Model(
        inputs=[state_input, advantages], outputs=[probabilities, values])
    actor.compile(
        loss=[actor_loss, 'mean_squared_error'],  # [PG, DQN]
```

```
        loss_weights=[1, critic_weight],  # [PG, DQN]
        optimizer=tf.keras.optimizers.Adam(lr=learning_rate))

    critic = models.Model(inputs=[state_input], outputs=[values])
    policy = models.Model(inputs=[state_input], outputs=[probabilities])
    return actor, critic, policy
```

The above is one way to go about combining both of the algorithms. Here, we're combining training of both pwathways into on operation. Keras allows for the training against multiple outputs. They can even have their own loss functions as we have above. When minimizing the loss, Keras will take the weighted sum of all the losses, with the weights provided in `loss_weights`. The `critic_weight` is now another hyperparameter for us to tune.

We could even have completely separate networks for the actor and the critic, and that type of design choice is going to be problem dependent. Having shared nodes and training between the two will be more efficient to train per batch, but more complicated problems could justify keeping the two separate.

The loss function we used here is also slightly different than the one for Policy Gradients. Let's take a look.

[16]:
```
def actor_loss(y_true, y_pred):  # PG
    y_pred_clipped = K.clip(y_pred, 1e-8, 1-1e-8)
    log_lik = y_true*K.log(y_pred_clipped)
    entropy_loss = y_pred * K.log(K.clip(y_pred, 1e-8, 1-1e-8))  # New
    return K.sum(-log_lik * advantages) - (entropy * K.sum(entropy_loss))
```

We've added a new tool called entropy. We're calculating the log-likelihood again, but instead of comparing the probabilities of our actions versus the action that was taken, we calculating it for the probabilities of our actions against themselves.

Certainly a mouthful, but the idea is to encourage exploration: if our probability prediction is very confident (or close to 1), our entropy will be close to 0. Similary, if our probability isn't confident at all (or close to 0), our entropy will again be zero. Anywhere inbetween, our entropy will be non-zero. This encourages exploration versus exploitation, as the entropy will discourage overconfident predictions.

Now that the networks are out of the way, let's look at the `Memory`. We could go with Experience Replay, like with DQNs, or we could calculate TD(1) like with Policy Gradients. This time, we'll do something in between. We'll give our memory a `batch_size`. Once there are enough experiences in the buffer, we'll use all the experiences to train and then clear the buffer to start fresh.

In order to speed up training, instead of recording state_prime, we'll record the value of state prime in `state_prime_values` or `next_values`. This will give us enough information to calculate the discounted values and advantages.

[17]:
```
class Memory():
    """Sets up a memory replay for actor-critic training.

    Args:
        gamma (float): The "discount rate" used to assess state values.
```

```
        batch_size (int): The number of elements to include in the buffer.
    """
    def __init__(self, gamma, batch_size):
        self.buffer = []
        self.gamma = gamma
        self.batch_size = batch_size

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

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

    def check_full(self):
        return len(self.buffer) >= self.batch_size

    def sample(self):
        """Returns formated experiences and clears the buffer.

        Returns:
            (list): A tuple of lists with structure [
                [states], [actions], [rewards], [state_prime_values], [dones]
            ]
        """
        # Columns have different data types, so numpy array would be awkward.
        batch = np.array(self.buffer).T.tolist()
        states_mb = 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)
        dones_mb = np.array(batch[3], dtype=np.int8)
        value_mb = np.squeeze(np.array(batch[4], dtype=np.float32))
        self.buffer = []
        return states_mb, actions_mb, rewards_mb, dones_mb, value_mb
```

Ok, time to build out the agent! The `act` method is the exact same as it was for Policy Gradients. Nice! The `learn` method is where things get interesting. We'll find the discounted future state like we did for DQN to train our critic. We'll then subtract the value of the discount state from the value of the current state to find the advantage, which is what the actor will train on.

[18]:
```
class Agent():
    """Sets up a reinforcement learning agent to play in a game environment."""
    def __init__(self, actor, critic, policy, memory, 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.actor = actor
        self.critic = critic
        self.policy = policy
        self.action_size = action_size
        self.memory = memory

    def act(self, state):
        """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 not acting randomly, take action with highest predicted value.
        state_batch = np.expand_dims(state, axis=0)
        probabilities = self.policy.predict(state_batch)[0]
        action = np.random.choice(self.action_size, p=probabilities)
        return action

    def learn(self, print_variables=False):
        """Trains the Deep Q Network based on stored experiences."""
        gamma = self.memory.gamma
        experiences = self.memory.sample()
        state_mb, action_mb, reward_mb, dones_mb, next_value = experiences

        # One hot enocde actions
        actions = np.zeros([len(action_mb), self.action_size])
        actions[np.arange(len(action_mb)), action_mb] = 1

        #Apply TD(0)
        discount_mb = reward_mb + next_value * gamma * (1 - dones_mb)
        state_values = self.critic.predict([state_mb])
        advantages = discount_mb - np.squeeze(state_values)
        if print_variables:
            print("discount_mb", discount_mb)
            print("next_value", next_value)
            print("state_values", state_values)
            print("advantages", advantages)
        else:
```

```
            self.actor.train_on_batch(
                [state_mb, advantages], [actions, discount_mb])
```

Run the below cell to initialize an agent, and the cell after that to see the variables used for training. Since it's early, the critic hasn't learned to estimate the values yet, and the advatanges are mostly positive because of it.

Once the crtic has learned how to properly assess states, the actor will start to see negative advantages. Try playing around with the variables to help the agent see this change sooner.

[19]:
```
# Change me please.
test_gamma = .9
test_batch_size = 32
test_learning_rate = .02
test_hidden_neurons = 50
test_critic_weight = 0.5
test_entropy = 0.0001

with tf.Graph().as_default():
    test_memory = Memory(test_gamma, test_batch_size)
    test_actor, test_critic, test_policy = build_networks(
        space_shape, action_size,
        test_learning_rate, test_critic_weight,
        test_hidden_neurons, test_entropy)
    test_agent = Agent(
        test_actor, test_critic, test_policy, test_memory, action_size)

    state = env.reset()
    episode_reward = 0
    done = False

    while not done:
        action = test_agent.act(state)
        state_prime, reward, done, _ = env.step(action)
        episode_reward += reward
        next_value = test_agent.critic.predict([[state_prime]])
        test_agent.memory.add((state, action, reward, done, next_value))
        state = state_prime
        test_agent.learn(print_variables=True)
```

Have a set of variables you're happy with? Ok, time to shine! Run the below cell to see how the agent trains.

[21]:
```
with tf.Graph().as_default():
    test_memory = Memory(test_gamma, test_batch_size)
    test_actor, test_critic, test_policy = build_networks(
        space_shape, action_size,
        test_learning_rate, test_critic_weight,
        test_hidden_neurons, test_entropy)
```

```
    test_agent = Agent(
        test_actor, test_critic, test_policy, test_memory, action_size)
    for episode in range(200):
        state = env.reset()
        episode_reward = 0
        done = False

        while not done:
            action = test_agent.act(state)
            state_prime, reward, done, _ = env.step(action)
            episode_reward += reward
            next_value = test_agent.critic.predict([[state_prime]])
            test_agent.memory.add((state, action, reward, done, next_value))

            #if test_agent.memory.check_full():
            #test_agent.learn(print_variables=True)
            state = state_prime
        test_agent.learn()
        print("Episode", episode, "Score =", episode_reward)
```

```
Episode 0 Score = 13.0
Episode 1 Score = 13.0
Episode 2 Score = 13.0
Episode 3 Score = 11.0
Episode 4 Score = 10.0
Episode 5 Score = 9.0
Episode 6 Score = 11.0
Episode 7 Score = 9.0
Episode 8 Score = 11.0
Episode 9 Score = 10.0
Episode 10 Score = 10.0
Episode 11 Score = 9.0
Episode 12 Score = 9.0
Episode 13 Score = 9.0
Episode 14 Score = 10.0
Episode 15 Score = 10.0
Episode 16 Score = 9.0
Episode 17 Score = 9.0
Episode 18 Score = 10.0
Episode 19 Score = 10.0
Episode 20 Score = 10.0
Episode 21 Score = 10.0
Episode 22 Score = 9.0
Episode 23 Score = 9.0
Episode 24 Score = 10.0
Episode 25 Score = 10.0
Episode 26 Score = 9.0
Episode 27 Score = 8.0
```

```
Episode 28 Score = 9.0
Episode 29 Score = 10.0
Episode 30 Score = 10.0
Episode 31 Score = 10.0
Episode 32 Score = 9.0
Episode 33 Score = 9.0
Episode 34 Score = 10.0
Episode 35 Score = 10.0
Episode 36 Score = 10.0
Episode 37 Score = 10.0
Episode 38 Score = 10.0
Episode 39 Score = 11.0
Episode 40 Score = 9.0
Episode 41 Score = 10.0
Episode 42 Score = 10.0
Episode 43 Score = 9.0
Episode 44 Score = 11.0
Episode 45 Score = 10.0
Episode 46 Score = 9.0
Episode 47 Score = 10.0
Episode 48 Score = 10.0
Episode 49 Score = 9.0
Episode 50 Score = 9.0
Episode 51 Score = 9.0
Episode 52 Score = 9.0
Episode 53 Score = 10.0
Episode 54 Score = 9.0
Episode 55 Score = 9.0
Episode 56 Score = 10.0
Episode 57 Score = 9.0
Episode 58 Score = 10.0
Episode 59 Score = 9.0
Episode 60 Score = 10.0
Episode 61 Score = 9.0
Episode 62 Score = 8.0
Episode 63 Score = 10.0
Episode 64 Score = 11.0
Episode 65 Score = 9.0
Episode 66 Score = 9.0
Episode 67 Score = 10.0
Episode 68 Score = 10.0
Episode 69 Score = 8.0
Episode 70 Score = 10.0
Episode 71 Score = 9.0
Episode 72 Score = 9.0
Episode 73 Score = 9.0
Episode 74 Score = 8.0
Episode 75 Score = 9.0
```

```
Episode 76 Score = 8.0
Episode 77 Score = 9.0
Episode 78 Score = 10.0
Episode 79 Score = 9.0
Episode 80 Score = 9.0
Episode 81 Score = 10.0
Episode 82 Score = 11.0
Episode 83 Score = 9.0
Episode 84 Score = 8.0
Episode 85 Score = 9.0
Episode 86 Score = 8.0
Episode 87 Score = 10.0
Episode 88 Score = 10.0
Episode 89 Score = 9.0
Episode 90 Score = 10.0
Episode 91 Score = 10.0
Episode 92 Score = 10.0
Episode 93 Score = 9.0
Episode 94 Score = 10.0
Episode 95 Score = 8.0
Episode 96 Score = 9.0
Episode 97 Score = 10.0
Episode 98 Score = 9.0
Episode 99 Score = 10.0
Episode 100 Score = 9.0
Episode 101 Score = 11.0
Episode 102 Score = 9.0
Episode 103 Score = 9.0
Episode 104 Score = 9.0
Episode 105 Score = 10.0
Episode 106 Score = 9.0
Episode 107 Score = 10.0
Episode 108 Score = 9.0
Episode 109 Score = 9.0
Episode 110 Score = 10.0
Episode 111 Score = 10.0
Episode 112 Score = 9.0
Episode 113 Score = 9.0
Episode 114 Score = 8.0
Episode 115 Score = 9.0
Episode 116 Score = 9.0
Episode 117 Score = 8.0
Episode 118 Score = 10.0
Episode 119 Score = 8.0
Episode 120 Score = 10.0
Episode 121 Score = 10.0
Episode 122 Score = 9.0
Episode 123 Score = 10.0
```

```
Episode 124 Score = 8.0
Episode 125 Score = 9.0
Episode 126 Score = 9.0
Episode 127 Score = 10.0
Episode 128 Score = 10.0
Episode 129 Score = 10.0
Episode 130 Score = 10.0
Episode 131 Score = 8.0
Episode 132 Score = 10.0
Episode 133 Score = 9.0
Episode 134 Score = 9.0
Episode 135 Score = 9.0
Episode 136 Score = 11.0
Episode 137 Score = 9.0
Episode 138 Score = 10.0
Episode 139 Score = 10.0
Episode 140 Score = 9.0
Episode 141 Score = 10.0
Episode 142 Score = 8.0
Episode 143 Score = 10.0
Episode 144 Score = 10.0
Episode 145 Score = 8.0
Episode 146 Score = 9.0
Episode 147 Score = 9.0
Episode 148 Score = 10.0
Episode 149 Score = 10.0
Episode 150 Score = 9.0
Episode 151 Score = 10.0
Episode 152 Score = 8.0
Episode 153 Score = 10.0
Episode 154 Score = 10.0
Episode 155 Score = 9.0
Episode 156 Score = 10.0
Episode 157 Score = 9.0
Episode 158 Score = 9.0
Episode 159 Score = 10.0
Episode 160 Score = 9.0
Episode 161 Score = 9.0
Episode 162 Score = 10.0
Episode 163 Score = 10.0
Episode 164 Score = 9.0
Episode 165 Score = 10.0
Episode 166 Score = 10.0
Episode 167 Score = 8.0
Episode 168 Score = 10.0
Episode 169 Score = 10.0
Episode 170 Score = 10.0
Episode 171 Score = 9.0
```

```
Episode 172 Score = 9.0
Episode 173 Score = 11.0
Episode 174 Score = 8.0
Episode 175 Score = 10.0
Episode 176 Score = 9.0
Episode 177 Score = 10.0
Episode 178 Score = 11.0
Episode 179 Score = 9.0
Episode 180 Score = 10.0
Episode 181 Score = 9.0
Episode 182 Score = 9.0
Episode 183 Score = 10.0
Episode 184 Score = 10.0
Episode 185 Score = 10.0
Episode 186 Score = 8.0
Episode 187 Score = 9.0
Episode 188 Score = 9.0
Episode 189 Score = 9.0
Episode 190 Score = 9.0
Episode 191 Score = 10.0
Episode 192 Score = 9.0
Episode 193 Score = 9.0
Episode 194 Score = 9.0
Episode 195 Score = 9.0
Episode 196 Score = 8.0
Episode 197 Score = 9.0
Episode 198 Score = 11.0
Episode 199 Score = 9.0
```

Any luck? No sweat if not! It turns out that by combining the power of both algorithms, we also combined some of their setbacks. For instance, actor-critic can fall into local minimums like Policy Gradients, and has a large number of hyperparameters to tune like DQNs.

Time to check how our agents did in the cloud! Any lucky winners? Find it in your bucket to watch a recording of it play.