Implementation Guide: [Robot Control Using Trust Region Policy Optimization (TRPO) vs Proximal Policy Optimization (PPO)]:

Step 1

The following is a Linux command that installs the X Virtual Framebuffer (Xvfb) package:
to enables running graphical applications, such as gym, without the need for a physical display.

```
apt-get install -y xvfb
```

The following code installs the necessary dependencies for running a reinforcement learning algorithm on a virtual frame buffer:Its install gym==0.23.1, that provides a range of environments for developing and comparing reinforcement learning algorithms, pytorch-lightning==1.6: its library that simplifies the prototyping and research process for deep learning models.
pyvirtualdisplay: A Python library that acts as a wrapper for Xvfb and other virtual display libraries.
Install these packages together will allow you to run reinforcement learning algorithm on virtual frame buffer.

```
pip install gym==0.23.1 \
    pytorch-lightning==1.6 \
    pyvirtualdisplay
```

Step 2

Install the brax library from its Github repository.
This library is a set of utilities and environments for reinforcement learning, so this package will make it easier to use and work with reinforcement learning environments and methods in your code.

```
pip install git+https://github.com/google/brax.git@v0.0.11
```

Step 3

Imports a variety of libraries that are commonly used in machine learning, reinforcement learning, and data visualization. Some of the specific functions and classes that are imported:

```python
import copy
import torch
import random
import gym
import matplotlib
import functools
import itertools
import math
import numpy as np
import matplotlib.pyplot as plt
import torch.nn.functional as
from collections import deque, namedtuple
from IPython.display import HTML
from base64 import b64encode
from torch import nn
from torch.utils.data import DataLoader
from torch.utils.data.dataset import IterableDataset
from torch.optim import AdamW, Optimizer
from torch.distributions import Normal, kl_divergence
from pytorch_lightning import LightningModule, Trainer
import brax
from brax import envs
from brax.envs import to_torch
from brax.io import html
```

Step 4

The line of code device = 'cuda:0' is used to set the device to the first available GPU, with the index of 0, on which a tensor should be stored and operated on, It then gets the number of CUDA-enabled GPUs available on the system and assigns it to the num_gpus variable, then creates a 1-dimensional tensor of ones on the device specified in the device variable and assigns it to the v variable.

```
device = 'cuda:0'

num_gpus = torch.cuda.device_count()

v = torch.ones(1, device='cuda')
```

It's worth to mention that if you don't have GPU device on your system, this code will raise an error.

Step 5

In this step uses the PyTorch library to create video function: create_video.
This function takes an environment, the number of steps the agent takes in the environment as input. The function uses the samples actions from the environment's action space, then it takes these actions in the environment and collects the states of the environment in an array. Finally, it returns a rendered video of the agent's actions in the environment, which allows the user to see how the agent is behaving in the environment.

```
@torch.no_grad()

def create_video(env, episode_length, policy=None):

  qp_array = []

  state = env.reset()

  for i in range(episode_length):

    if policy:

      loc, scale = policy(state)

      sample = torch.normal(loc, scale)

      action = torch.tanh(sample)

    else:

      action = env.action_space.sample()

    state, _, _, _ = env.step(action)

    qp_array.append(env.unwrapped._state.qp)
```

Step 6

From PyTorch library create test_agent function to evaluate the performance of an agent in an environment. It takes an environment, the number of steps the agent takes in the environment, a policy function and the number of episodes as input. The function uses the policy to generate actions, then it takes these actions in the environment and accumulates the rewards. It repeats this process for a number of episodes, then it returns the average of the accumulated rewards as a performance metric of the agent. This function allows the user to evaluate the effectiveness of the agent's policy in the environment.

```
@torch.no_grad()
def test_agent(env, episode_length, policy, episodes=10):
  ep_returns = []
  for ep in range(episodes):
    state = env.reset()
    done = False
    ep_ret = 0.0
    while not done:
      loc, scale = policy(state)
      sample = torch.normal(loc, scale)
      action = torch.tanh(sample)
      state, reward, done, info = env.step(action)
      ep_ret += reward.item()
    ep_returns.append(ep_ret)
```

Step 7

Create the RunningMeanStd class to keep track of the running mean and standard deviation of a stream of data. It is a way to calculate the mean and standard deviation of a large dataset, by processing the data in small segments and updating the running mean and standard deviation

```
class RunningMeanStd:
    def __init__(self, epsilon=1e-4, shape=()):
        self.mean = torch.zeros(shape, dtype=torch.float32).to(device)
        self.var = torch.ones(shape, dtype=torch.float32).to(device)
        self.count = epsilon
    def update(self, x):
        batch_mean = torch.mean(x, dim=0)
        batch_var = torch.var(x, dim=0)
        batch_count = x.shape[0]
        self.update_from_moments(batch_mean, batch_var, batch_count)
    def update_from_moments(self, batch_mean, batch_var, batch_count):
        self.mean, self.var, self.count = update_mean_var_count_from_moments(
            self.mean, self.var, self.count, batch_mean, batch_var, batch_count
        )
def update_mean_var_count_from_moments(
    mean, var, count, batch_mean, batch_var, batch_count
):
    delta = batch_mean - mean
    tot_count = count + batch_count
    new_mean = mean + delta * batch_count / tot_count
    m_a = var * count
    m_b = batch_var * batch_count
    M2 = m_a + m_b + torch.square(delta) * count * batch_count / tot_count
    new_var = M2 / tot_count
    new_count = tot_count
    return new_mean, new_var, new_count
  return sum(ep_returns) / episodes
```

Step 8

Define the class "NormalizeObservation" is to normalize the observations coming from a gym environment by using the running mean and standard deviation. It wraps around a gym environment and normalizes the observations obtained from the environment before returning them.

```
class NormalizeObservation(gym.core.Wrapper):
    def __init__(self, env, epsilon=1e-8):
        super().__init__(env)
        self.num_envs = getattr(env, "num_envs", 1)
        self.obs_rms = RunningMeanStd(shape=self.observation_space.shape[-1])
        self.epsilon = epsilon
    def step(self, action):
        obs, rews, dones, infos = self.env.step(action)
        obs = self.normalize(obs)
        return obs, rews, dones, infos
    def reset(self, **kwargs):
        return_info = kwargs.get("return_info", False)
        if return_info:
            obs, info = self.env.reset(**kwargs)
        else:
            obs = self.env.reset(**kwargs)
        obs = self.normalize(obs)
        if not return_info:
            return obs
        else:
            return obs, info
    def normalize(self, obs):
        self.obs_rms.update(obs)
        return (obs - self.obs_rms.mean) / torch.sqrt(self.obs_rms.var + self.epsilon)
```

Step 9

This code defines a function called create_env which takes three parameters env_name, num_envs and episode length, The function creates an instance of the gym environment by calling the gym.make() function with the given and the number of environments and the length of the episode as arguments. Then it wraps the environment with the "NormalizeObservation" class defined earlier. This class normalizes the observations coming from the environment by using the running mean and standard deviation, the function returns the wrapped environment. Then creates an environment for running the 'ant environment with a total of 10 parallel environments. The env.reset() function is then called, which resets the environment and returns the initial observation of the environment.

```
def create_env(env_name, num_envs=256, episode_length=1000):
  env = gym.make(env_name, batch_size=num_envs, episode_length=episode_length)
  env = NormalizeObservation(env)
  return env
env = create_env('ant', num_envs=10)
obs = env.reset()
```

We have completed the main implementation steps of our project. The remaining details and procedures for execution can be found in the accompanying repository for reference, In order to implement the TRPO agent, we first implemented its optimizer and associated dataset. We then proceeded to implement the training code. Similarly, for the PPO agent, we first implemented the agent's data pipeline. Utilizing the TensorBoard tool, to visualize the results of both learned agents and compare the two results.