Predicting prosthetic finger kinematics in non-human primates using reinforcement learning

EECS 599 – Report

Sachin Salim
Mentor: Joseph Costello
Supervisor: Prof. Cynthia Chestek
CNPL - University of Michigan

I. Introduction

There has been considerable development and interest in brain-machine interfaces aimed at restoring motor function over the last decades. However, the capabilities of prosthetic limbs and fingers are still limited in replicating native function. One major limitation in achieving natural and rapid finger movements is the algorithm that converts brain signals into control signals for the prosthetic device. In order to address this limitation and improve the accuracy of prosthetic finger movements, researchers have been investigating the potential of machine learning algorithms [1] [2]. This research focuses on exploring the use of reinforcement learning methods to decipher finger movements and predict the kinematics in real-time, using a noisy kinematics data generated by a simulation. A closed-loop control system is used to update the position and velocity of the finger from the observation from the environment, which is one of the unique challenge this experiment tries to address. We believe that this study inspires further research on the use of neural networks in creating brain-controlled prostheses that can closely mimic natural movements.

II. Methods

A. Environment

1) Setup: A simulation environment is created for studying continuous finger movement target-acquisition tasks with variable degrees of freedom. The environment is responsible for generating the targets, providing an observation (current state) to the user, and in managing trails. The user may move using commands on position or velocity, and succeeds the experiment if the position is held within the size of target for a period specified by hold-time. The next target is shown as soon as the trial is completed. Optionally, perturbations may be added which could alter the position at a given probability during the hold period. The environment is implemented in Python gymnasium (Open AI) which provides standard APIs for implementing reinforcement learning.

2) State and Actions: The target positions for each degree-of-freedom are represented in one dimension (within [0, 1] by default). These targets may be chosen from a set of finite values or from a continuous range uniformly randomly. There is also an option to decide if the targets should alternate between the center position and rest of the positions so as to make a center-out movement. In each step, the user/agent takes an action as position or velocity which updates the current state. The position takes values in [0, 1] range whereas velocity lies in [-1, 1] range. If the current position stays within the size of the target for a sufficient period of time, the episode is deemed as successful. Depending on the experiment, the agent may observe various states such as the current position, velocity, or a noisy desired velocity.

3) Baseline Algorithm: In order to guide the agent towards taking the correct sequence of actions that lead to success in the environment, a baseline algorithm must first be defined for comparison purposes. Initially, a random action was considered as the baseline algorithm, but it did not yield success in a reasonable amount of time. Therefore, a new baseline algorithm was defined to take the action of the observed desired velocity from the environment. The desired velocity is calculated by the environment as a linear function of the distance remaining to the target, and some gaussian noise is added to make it more realistic. This noise factor was included because the desired velocity predicted by a model (such as an RNN) based on neural data may not be entirely precise. The standard deviation of the added noise was determined in such a way as to achieve a 90% success rate with the baseline algorithm.

B. Reinforcement Learning

To predict the optimal action based on observations, a Reinforcement Learning (RL) model was introduced. This was achieved using RLlib-Ray, an open-source library designed to support highly distributed RL workloads at a production-level scale. In implementing the RL model, the main challenges involved defining an appropriate reward function and configuring the right parameters for the algorithm.

1) Reward: Rewards and in particular the reward function is an important modeling component for any RL problem. Various ideas were attempted to achieve an efficient way for the model to train on. The ultimate goal was to encourage the agent to move towards the target and remain close to it. Additionally, the agent had to be discouraged from moving slowly, and from overshooting the target. Hence, a reward function that provides punishment which increases a function of the distance to the target was implemented. Some initial attempts to model the reward was later discarded because of its complex definition. A simpler reward function was finally decided as shown in equation 1.

$$r = \begin{cases} 1 & \text{if succeeds the episode} \\ 0 & \text{if in the target} \\ \frac{-1}{T} & \text{if outside the target} \end{cases} \tag{1}$$

where T is the number of timesteps it takes for the episode to timeout. To avoid overly punishing the agent, the negative reward for each timestep was scaled down. Specifically, the minimum return that the agent can receive in an episode was set to -1. This ensures that the return is positive only if the agent is successful. If the agent achieves a return close to 1, it means that it succeeded quickly in the task.

2) Building and training the algorithm: Since this is a problem involving continuous action spaces, PPO (Proximal Policy Optimization) was used to train the RL decoder. PPO is a reinforcement learning algorithm used to optimize policies for Markov decision processes (MDPs), proposed by OpenAI in 2017 as an improvement over previous policy gradient methods. The implementation of PPO was readily available in the algorithms package of RL-Lib.

The config settings of the algorithm were updated to extract maximum performance while training. The system used to train the model was CNPL-Brainiac, a Windows 10 machine with an intel core processor 3.6GHz 128GB RAM. The performance speed of training was compared by varying the number of rollout workers for parallel sampling between 1 and 8.

The algorithm was then instantiated using the config object's build method so that this can be trained. It is important to register the environment with ray using an environment specifier (a unique name) using the tune API. It was then trained using train API and the algorithm as well as the results were saved every 10 iterations. A learning rate of 10^{-4} and a discount factor of 0.99 was used. In 1 training iteration, 100 episodes were run.

III. Results

A. Training the algorithm

1) Performance speed: The performance speed of training was compared by varying the number of rollout workers between 1 and 8. The result is shown in Figure 1. It is observed that more rollout workers increased the speed of the training. There was a 38% decrease in the amount of time taken for training with 8 rollout workers compared to just one worker.

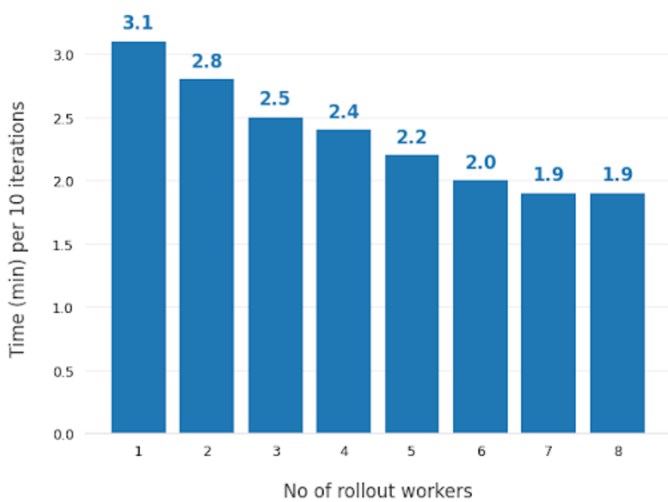


Fig. 1. Speed of training of PPO Algorithm for various number of rollout workers

2) Learning the action: The training process recorded the mean reward every 10 iterations, and it was observed that the algorithm struggled to learn accurate action predictions as the environment had more potential target positions. The corresponding figure (Figure 3) indicates that the algorithm achieved optimal learning after approximately 200 iterations when there were only 2 target positions. However, when the number of targets exceeded 4, the algorithm encountered difficulty in comprehending the correct actions.

B. Evaluating the algorithm

To evaluate the performance of the algorithm, 1000 episodes were run with the aaction being predicted by the algorithm. The percentage of episodes they were able to win is tabulated in I. The accuracy of the baseline algorithm is also provided for reference. To further visualize the results, the position of the finger is plotted for 60,000 timesteps with the desired velocity being predicted by the PPO algorithm. This is illustrated in 2.

Number of target positions	Success Rate
2	99.7%
3	99.1%
4	93.4%
5	7.3%
6	0.2%
Baseline	90.12%

TABLE I
Percentage of episodes the algorithm succeeds in; with different number of target positions

IV. Challenges faced

There were quite a few challenges while trying to setup RLLib. This was initially being trained using the free version of Google Colab, which disallowed the use of multiple rollout workers. The training was later shifted to a Linux Machine (CNPL-Parasite) which had trouble building the algorithm because of the incompatibility between the cuda versions of the system and the ray cluster. The environment implementation in Gym faced an additional challenge when it was moved to Gymnasium, resulting in several incompatibility errors that had to be addressed.

V. Other works

A. Tuning hyper-parameters of feed-forward neural network

In addition to the primary research on Reinforcement Learning, another task I had worked was to determine the optimal time-history and bin-size that would enhance the performance of a feed forward network. It involved modifying a Python Jupyter notebook that loaded neural and finger data, trained a recurrent neural network, and generated plots of

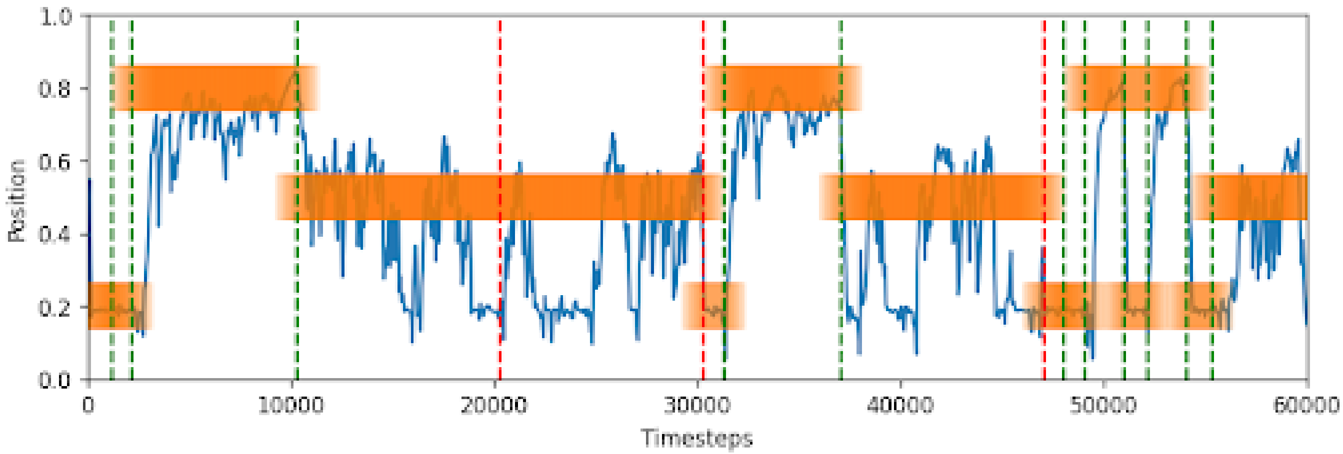


Fig. 2. The position of the finger with the desired velocity predicted by the RL algorithm. The target positions are represented by horizontal orange stripes, with the width of each stripe indicating the size of the corresponding target. The end of an episode is marked by vertical dashed lines, with green lines representing a successful episode and red lines indicating a failed one.

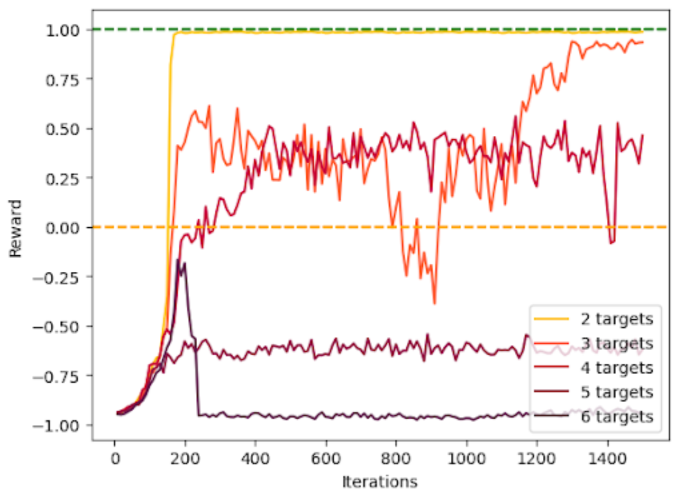


Fig. 3. Comparison of overall rewards (return) for environments with different number of possible target positions. When the training plot crosses the dashed orange line (reward 0), it indicates the algorithm has began winning episodes. When the plot reaches close to the dashed green line (reward 1), the algorithm is winning almost all the episodes.

the decoded finger kinematics. The objective was to replace the RNN with a feedforward network and conduct the performance analysis. A grid search was conducted on a predetermined set of time-history and bin-size values to evaluate the correlation and mean-squared error (MSE) of the model results. From the Figure 4, it can be seen that a bin size of around 200ms and a time history of around 5 bins yield the best results.

VI. Discussion

The successful performance of Reinforcement Learning in accurately predicting the appropriate action and outperforming the baseline algorithm for a smaller number of target positions is a promising outcome. However, further investigation is necessary to determine the cause of its failure with a greater number of target positions. To address this, future research [3] will involve providing both position and velocity data to the RL decoder obtained from neural data using an alternate decoder like RNN. Additionally, the RL decoder could be directly fed with neural data to improve its ability to predict position accurately.

References

[1] M. S. Willsey, S. R. Nason-Tomaszewski, S. R. Ensel, H. Temmar, M. J. Mender, J. T. Costello, P. G. Patil, and C. A. Chestek, "Real-time brain-machine interface in non-human primates achieves high-velocity prosthetic finger movements using a shallow feedforward neural network decoder," *Nature Communications*, vol. 13, no. 1, p. 6899, 2022.

[2] M. N. Almani and S. Saxena, "Recurrent neural networks controlling musculoskeletal models predict motor cortex activity during novel limb movements," in *2022 44th Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pp. 3350–3356, IEEE, 2022.

[3] B. Girdler, W. Caldbeck, and J. Bae, "Neural decoders using reinforcement learning in brain machine interfaces: A technical review," *Frontiers in Systems Neuroscience*, vol. 16, 2022.

[4] S. R. Nason, A. K. Vaskov, M. S. Willsey, E. J. Welle, H. An, P. P. Vu, A. J. Bullard, C. S. Nu, J. C. Kao, K. V. Shenoy, *et al.*, "A low-power band of neuronal spiking activity dominated by local single units improves the performance of brain–machine interfaces," *Nature biomedical engineering*, vol. 4, no. 10, pp. 973–983, 2020.

[5] F. R. Willett, D. T. Avansino, L. R. Hochberg, J. M. Henderson, and K. V. Shenoy, "High-performance brain-to-text communication via handwriting," *Nature*, vol. 593, no. 7858, pp. 249–254, 2021.

[6] J. C. Sanchez, A. Tarigoppula, J. S. Choi, B. T. Marsh, P. Y. Chhatbar, B. Mahmoudi, and J. T. Francis, "Control of a center-out reaching task using a reinforcement learning brain-machine interface," in *2011 5th International IEEE/EMBS Conference on Neural Engineering*, pp. 525–528, IEEE, 2011.

VII. Appendix

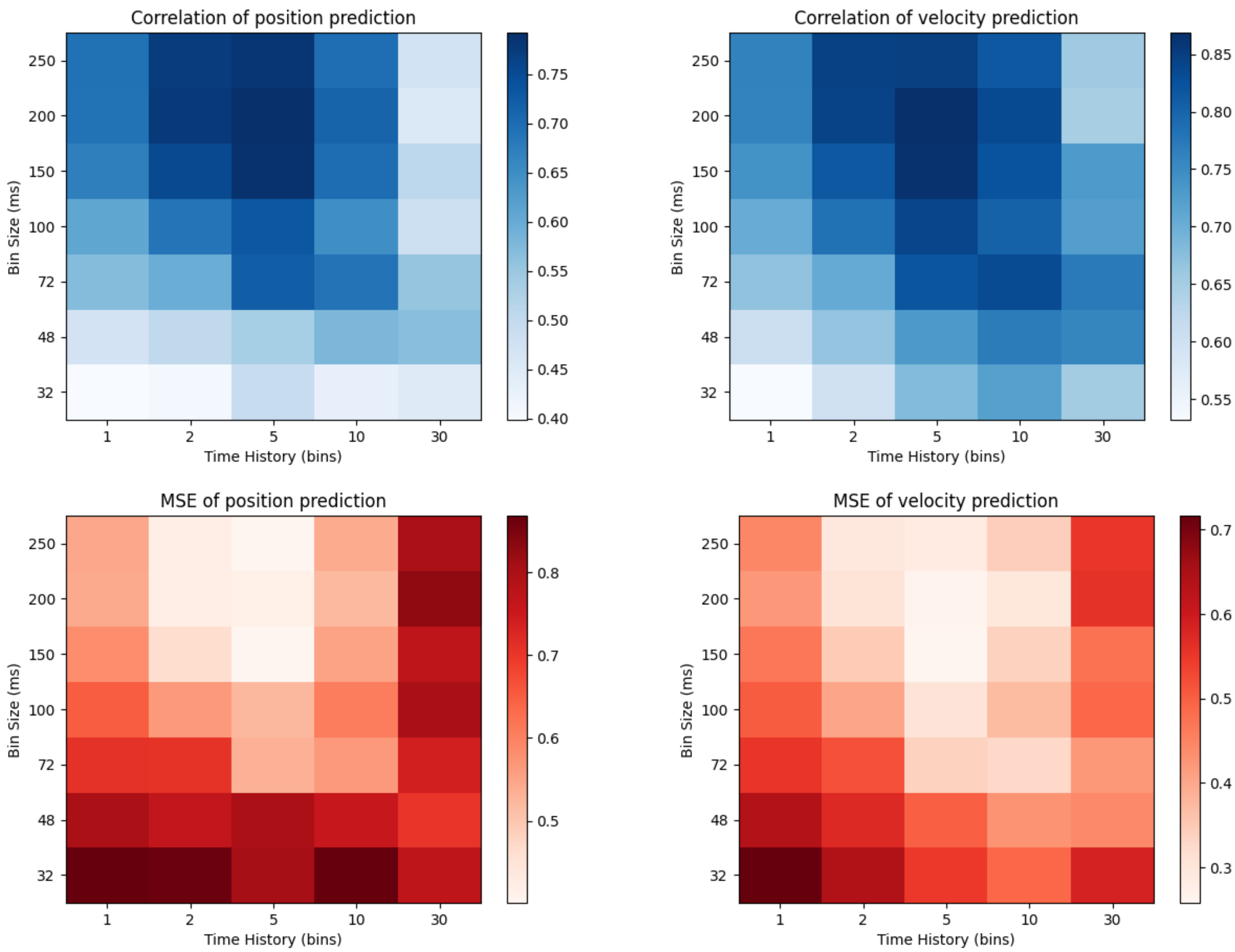


Fig. 4. Heatmap of correlation and MSE of decoding neural data using a feed forward neural network.

Predicting prosthetic finger kinematics in non-human primates using reinforcement learning

Authors: Sachin Salim (sachinks), Joseph Costello (costellj)

In [1]:
```
ROOT_DIR = ''
```

In [2]:
```
# import all the libs
import gymnasium as gym
from gymnasium import spaces
from gymnasium.wrappers import EnvCompatibility
import numpy as np
import pandas as pd
import pdb
import torch

import ray
from ray.rllib.algorithms.ppo import PPOConfig
from ray import tune
import re
import matplotlib.pyplot as plt

from ipywidgets import Output
from IPython import display
import time

import warnings
warnings.filterwarnings('ignore')
```

Environment

Source code

In [3]:
```
class ProportionalUserStrategy:
    """
    The user moves toward the target with velocity proportional to distance. Each fing
    """

    def __init__(self, speed_scaler, maxspeed, dist_thresh=None):
        self.speed = speed_scaler
        self.maxspeed = maxspeed
        self.dist_thresh = dist_thresh

        # TODO: add option to add noise

    def get_velocity(self, state_dict):
        """
```

```
        Returns a velocity using the control strategy.
        state_dict (dict): contains 'position' and 'target_pos'
        """
        dist = state_dict['target_pos'] - state_dict['position']
        velocity = np.clip(self.speed * dist, -self.maxspeed, self.maxspeed)

        # stop moving if within thresh of the target
        if self.dist_thresh:
            velocity[np.abs(dist) < self.dist_thresh] = 0

        return velocity

class ProportionalUserStrategyWithNoise(ProportionalUserStrategy):
    """
    Same as ProportionalUserStrategy, but with noise added to the velocity. The user m
    proportional to distance. Each finger is calculated independently.
    """

    def __init__(self, speed_scaler, maxspeed, dist_thresh=None, noise_std=0.1):
        super().__init__(speed_scaler, maxspeed, dist_thresh)
        self.noise_std = noise_std

    def get_velocity(self, state_dict):
        velocity = super().get_velocity(state_dict)
        velocity += np.random.normal(0, self.noise_std, velocity.shape)
        return velocity
```

In [4]:

```
"""
This module contains simulation environments for BMI tasks. Each environment handles g
observation/current state to the user, and managing trials
"""


class TargetGenerator:
    def __init__(self, num_dof=1, center_out=False, is_discrete=False, discrete_targs=
        """
        :param num_dof (int):           Number of degrees of freedom (i.e. how many ta
        :param center_out (bool):       If True, alternates between the center positio
        :param is_discrete (bool):      If True, will choose targets from the discrete
                                        randomly choose targets from the continuous_ra
        :param discrete_targs (list):   List of target positions to choose from in dis
                                        within 0-1. Use the function setup_discrete_ta
        :param continuous_range (list): List with the upper and lower limits for conti
        """
        self.num_dof = num_dof
        self.center_out = center_out
        self.is_discrete = is_discrete
        self.discrete_targs = discrete_targs
        self.cont_range = continuous_range if continuous_range else [0, 1]

        self.at_center = False
        self.target_pos = None

    def _choose_targ(self):
        if self.is_discrete:
            # choose discrete target
            return np.random.choice(self.discrete_targs)
```

```
        else:
            # choose a continuous target
            return np.random.uniform(self.cont_range[0], self.cont_range[1])

    def reset(self):
        self.at_center = False
        self.target_pos = None

    def generate_targets(self):
        if self.center_out:
            if not self.at_center:
                self.target_pos = np.array([0.5 for _ in range(self.num_dof)])
                self.at_center = True
            else:
                self.target_pos = np.array([self._choose_targ() for _ in range(self.nu
                self.at_center = False

        else:
            self.target_pos = np.array([self._choose_targ() for _ in range(self.num_dc

        return self.target_pos


class TargetGeneratorDOFIndependent:
    """ Same as a target generator, but each DOF has its own target generator.
     This enables things like center-out for one DOF and random targets for another"""
    def __init__(self, target_gen_list):
        self.targ_gens = target_gen_list
        self.num_dof = len(target_gen_list)

    def reset(self):
        for gen in self.targ_gens:
            gen.reset()

    def generate_targets(self):
        return np.array([gen.generate_targets() for gen in self.targ_gens]).reshape((-


def setup_discrete_targets(num_targets, lowlim=0, uplim=1, remove_center=False):
    """ function to automatically calculate equally spaced targets """
    targets = list(np.linspace(lowlim, uplim, num_targets))
    if remove_center:
        targets = [target for target in targets if (target != 0.5)]
    return targets


class ContinuousBmiTaskEnv(gym.Env):
    """
    Environment for simulating a continuous movement target-acquisition task with vari
    user move using position or velocity commands, and requires a hold time on the tar
    Note: there is no delay/preparatory period - as soon as a trial is completed the n

    Also has an option for adding perturbations, i.e. jumps in the position, at a give
    hold period.

    Following the gym structure, has main functions: init, reset, and step.
    References for the gym api:
```

```
        https://www.gymlibrary.ml/content/environment_creation/
        https://www.gymlibrary.ml/content/api/
    """

    def __init__(self,
                 num_dof=2,
                 dt_ms=50,
                 target_size=0.12,
                 target_generator=None,
                 hold_time_ms=500,
                 trial_timeout_ms=10000,
                 target_in_obs=False,
                 use_velocity_action=True,
                 perturb_prob=0.0,
                 perturb_dict=None,
                 strategy=None,):
        """
        :param num_dof (int):           Number of fingers
        :param dt_ms (int):                 Milliseconds per timestep (the binsize)
        :param target_size (float):         Target size as proportion of full position
        :param target_generator:            A TargetGenerator object (which creates ta
        :param hold_time_ms (int):          Milliseconds for the hold time
        :param trial_timeout_ms (int):      Max number milliseconds before trial failu
        :param target_in_obs (bool):        If target position should be shown in the
        :param use_velocity_action (bool):  If True, the inputted action should be vel
                                            integrated to get the new positions. If Fa
                                            be positions.

        :param perturb_prob (float):        Probability of perturbing the target posit
        :param perturb_dict (dict):         Dictionary with the following keys:
                                                'magnitude': float, the magnitude of t
                                                'min_hold_time_ms': int, the point dur
                                                                    perturbation is ap

        """
        self.num_dof = num_dof
        self.dt_ms = dt_ms
        self.target_size = target_size

        self.targ_gen = target_generator
        self.hold_time_ms = hold_time_ms
        self.trial_timeout_ms = trial_timeout_ms
        self.target_in_obs = target_in_obs
        self.vel_action = use_velocity_action

        self.perturb_prob = perturb_prob
        self.attempted_perturb = False  # if a perturbation was tried this trial (the
        if perturb_prob > 0:
            self.perturb_mag = perturb_dict["magnitude"]
            self.perturb_time_hold_ms = perturb_dict["min_hold_time_ms"]

        self.current_trial = 0          # how many trials total
        self.t_millis = 0               # how many total ms (all trials)
        self.trial_t_ms = 0             # how many ms in this trial
        self.in_targ_ms = 0             # how many ms inside the target
        self.target_pos = None          # target position
        self.pos = None                 # dof position
        self.vel = None                 # dof velocity
```

```python
        self.acc = None                  # dof acceleration
        self.timed_out = False           # Whether the experiment is timed out
        self.strategy = strategy

        self.reset_full()

        # setup observation and action spaces (https://www.gymlibrary.ml/content/api/#
        if target_in_obs:
            self.observation_space = spaces.Dict({
                "target_pos": spaces.Box(low=0.0, high=1.0, shape=(num_dof,), dtype=np
                # "position": spaces.Box(low=0.0, high=1.0, shape=(num_dof,), dtype=np
                # "velocity": spaces.Box(low=-1.0, high=1.0, shape=(num_dof,), dtype=n
                "desired_pos": spaces.Box(low=0.0, high=1.0, shape=(num_dof,), dtype=n
                "desired_vel": spaces.Box(low=-1.0, high=1.0, shape=(num_dof,), dtype=
            })
        else:
            self.observation_space = spaces.Dict({
                # "position": spaces.Box(low=0.0, high=1.0, shape=(num_dof,), dtype=np
                # "velocity": spaces.Box(low=-1.0, high=1.0, shape=(num_dof,), dtype=n
                "desired_pos": spaces.Box(low=0.0, high=1.0, shape=(num_dof,), dtype=n
                "desired_vel": spaces.Box(low=-1.0, high=1.0, shape=(num_dof,), dtype=
            })

        if use_velocity_action:
            self.action_space = spaces.Box(low=-1.0, high=1.0, shape=(num_dof,), dtype
        else:
            self.action_space = spaces.Box(low=0, high=1.0, shape=(num_dof,), dtype=np

    def _get_obs(self):
        obs_dict = {}
        # obs_dict['position'] = self.pos
        # obs_dict['velocity']= self.vel
        if self.target_in_obs:
            obs_dict['target_pos'] = self.target_pos
        desired_vel = self.strategy.get_velocity(self.get_info())
        desired_pos = np.clip(self.pos + desired_vel, 0.0, 1.0)

        obs_dict['desired_vel'] = desired_vel
        obs_dict['desired_pos'] = desired_pos
        return obs_dict

    def get_info(self):
        return {
            'current_trial': self.current_trial,
            'total_t_ms': self.t_millis,
            'trial_t_ms': self.trial_t_ms,
            'target_pos': self.target_pos,
            'position': self.pos,
            'velocity': self.vel,
            'acceleration': self.acc,
            'timed_out': self.timed_out
        }

    def reset_full(self):
        self.current_trial = 0
        self.t_millis = 0
        self.trial_t_ms = 0
```

```
        self.in_targ_ms = 0
        self.targ_gen.reset()
        self.target_pos = self.targ_gen.generate_targets()
        self.pos = 0.5 * np.ones(self.num_dof)
        self.vel = np.zeros(self.num_dof)
        self.acc = np.zeros(self.num_dof)
        self.attempted_perturb = False
        return self._get_obs()

    def reset(self):
        self.current_trial += 1
        self.target_pos = self.targ_gen.generate_targets()
        self.trial_t_ms = 0
        self.in_targ_ms = 0
        self.attempted_perturb = False
        return self._get_obs()

    def _is_in_targ(self):
        return np.all(np.abs(self.pos - self.target_pos) < self.target_size)

    def _update_target_ms_count(self):
        in_targ = self._is_in_targ()
        if in_targ:
            self.in_targ_ms += self.dt_ms
        else:
            self.in_targ_ms = 0

    def _calc_reward(self, done):
        cur_pos, cur_vel, target_pos = self.pos, self.vel, self.target_pos
        if done and not self.timed_out:
            # trial success
            return 1

        # maximum number of time-steps
        T =  self.trial_timeout_ms / self.dt_ms

        in_targ = self._is_in_targ()
        if in_targ:
          return 0
        else:
          return -1/T

    def _add_perturbation(self):
        if (not self.attempted_perturb) and (self.in_targ_ms >= self.perturb_time_hold
            if np.random.rand() < self.perturb_prob:
                self.pos += np.random.choice([-1, 1], size=self.num_dof) * self.pertur
                # Note: each dof is not fully independent - either all or none are per
            self.attempted_perturb = True

    def step(self, action):
        """
        :param action (ndarray): velocity or position for each finger, depending on se
        :return: Tuple[observation, reward, done, info]
        """
        # update position
        prev_pos = self.pos
        prev_vel = self.vel
```

```python
        self.pos = self.pos + action if self.vel_action else action
        if self.perturb_prob > 0:
            self._add_perturbation()
        self.pos = np.clip(self.pos, 0, 1)
        self.vel = self.pos - prev_pos
        self.acc = self.vel - prev_vel

        # check if trial is done
        self.t_millis += self.dt_ms
        self.trial_t_ms += self.dt_ms
        self._update_target_ms_count()
        self.timed_out = self.trial_t_ms >= self.trial_timeout_ms
        if (self.in_targ_ms >= self.hold_time_ms) or self.timed_out:
            done = True
        else:
            done = False

        reward = self._calc_reward(done)
        # print('reward: ', reward)
        observation = self._get_obs()
        info = self.get_info()
        return observation, reward, done, info

    def render(self, mode="human"):
        dof = self.num_dof
        res = 40 # resolution
        for finger in range(dof):
          print(f"Finger {finger}")
          target = np.floor(res * self.target_pos[finger])
          pos = np.floor(res * self.pos[finger])
          for i in range(res+1):
            if i == target:
              if target == pos:
                print("&", end='')
              else:
                print("x", end='')
            elif i == pos:
              print("o", end='')
            else:
              print("=", end='')
          print()
```

Initializing environment

In [5]:

```python
num_dof = 1             # number of degrees of freedom
num_chans = 20          # number of channels
num_secs = 50          # number of seconds of data to simulate
binsize = 50            # binsize in ms
hold_time_ms = 1000      # hold time in ms
target_size = 0.08        # target size is used to calculate success
target_in_obs = False

train_val_test_split = [0.7, 0.1, 0.2]
batch_size = 64
conv_size = 20
```

```python
normalize_x = True      # normalize neural data
normalize_y = True      # normalize finger data
pred_type = 'pv'        # 'pv' means we predict position and velocity
```

In [6]:
```python
def get_params(env_version):
    params = {
        "speed_std": 0.036,
        "no_of_targets": 3
    }

    if env_version == "2.3.0.0":
        params["speed_std"] = 0.04
        params["no_of_targets"] = 6
    elif env_version == "2.3.0.2":
        params["speed_std"] = 0.04
        params["no_of_targets"] = 2
    elif env_version == "2.3.0.3":
        params["speed_std"] = 0.04
        params["no_of_targets"] = 3
    elif env_version == "2.3.0.31":
        params["speed_std"] = 0.036
        params["no_of_targets"] = 3
    elif env_version in ["2.3.0.41", "2.3.0.42"]:
        params["speed_std"] = 0.036
        params["no_of_targets"] = 4

    return params
```

In [7]:
```python
def init_env(env_version):
    params = get_params(env_version)
    strategy = ProportionalUserStrategyWithNoise(speed_scaler=0.15, maxspeed=0.2,
                                                 dist_thresh=0.02, noise_std=params["sp

    if True:
        targs = setup_discrete_targets(params["no_of_targets"], lowlim=0.2, uplim=0.8,
        targ_gen = TargetGenerator(num_dof=num_dof, center_out=False, is_discrete=True

    else:
        # option for random targets
        targ_gen = TargetGenerator(num_dof=num_dof, center_out=False, is_discrete=Fals


    # create an environment
    env = ContinuousBmiTaskEnv(num_dof=num_dof,
                               dt_ms=binsize,
                               target_size=target_size,
                               target_generator=targ_gen,
                               hold_time_ms=hold_time_ms,
                               trial_timeout_ms=10e3,
                               target_in_obs=target_in_obs,
                               use_velocity_action=True,
                               strategy = strategy)
    return env
```

Functions to run model and plot results

In [8]:
```
def run_model(model = None, num_episodes = 100, save_all_results = False, env=None):
  resultlist = []

  num_timesteps = 0
  # Collect all episode rewards here
  episode_rewards = []
  no_of_wins = 0

  env.reset_full()

  # Loop through episodes
  for ep in range(num_episodes):

      # Reset the environment at the start of each episode
      obs = env.reset()
      done = False
      episode_reward = 0.0

      # Loop through time steps per episode
      while True:
          # take random action, but you can also do something more intelligent
          # action = env.action_space.sample()
          # action = obs['desired_vel']
          if model is None:
            action = obs['desired_vel']
          else:
            action = model.compute_single_action(observation=obs, explore=False)

          # apply the action
          obs, reward, done, info = env.step(action)
          info['reward'] = reward
          info['done'] = done

          episode_reward += reward

          if save_all_results or ep < 20:
            # save only 1 episode unless save_all_results is True
            resultlist.append(pd.DataFrame([info]))

          # If the epsiode is up, then start another one
          num_timesteps += 1
          if done:
              if not info['timed_out']:
                # trial success
                no_of_wins += 1
              episode_rewards.append(episode_reward)
              break

  resultsdf = pd.concat(resultlist, ignore_index=True)

  # calculate mean_reward
  env_mean_random_reward = np.mean(episode_rewards)
  env_sd_reward = np.std(episode_rewards)
  # calculate number of wins
  total_reward = np.sum(episode_rewards)
```

```python
    print()
    print("**************")
    print(f"Mean Reward={env_mean_random_reward:.4f}+/-{env_sd_reward:.4f}")
    # print(f" (out of success={env_spec.reward_threshold})")
    print(f"got {total_reward:.2f} reward over {num_episodes} episodes ({num_timesteps}
    print(f"Approx {total_reward/num_episodes:.4f} reward per episode")
    print(f"won {no_of_wins} over {num_episodes} episodes")
    print("**************")

    return resultsdf
```

In [9]:
```python
def plot_simulated_data(df, t_max=40e3, targetsize = target_size, posvel='pos'):
    t = np.stack(df.total_t_ms.to_numpy())              # shape (num_steps,)
    target_pos = np.stack(df.target_pos.to_numpy())      # shape (num_steps, num_dof)
    finger_pos = np.stack(df['position'].to_numpy())
    finger_vel = np.stack(df['velocity'].to_numpy())
    success_trials = df.query(
        'done == True and timed_out == False')[['total_t_ms', 'target_pos']]
    failure_trials = df.query(
        'done == True and timed_out == True')[['total_t_ms', 'target_pos']]
    num_dof = finger_pos.shape[1]

    fig = plt.figure(figsize=(10,3), dpi=120)

    # multi-dof plot
    for i in range(num_dof):
        if posvel =='pos':
            plt.plot(t, finger_pos[:, i])
        elif posvel =='vel':
            plt.plot(t, finger_vel[:, i])
        else:
            pass

        # target position
        if posvel =='pos':
            y = target_pos[:, i]
        elif posvel =='vel':
            y = 0.4*target_pos[:, i]-0.2
        else:
            pass
        plt.plot(t, target_pos[:, i], linewidth=0, marker='s',
                 markersize=targetsize*262, alpha=0.05)

    for i in range(num_dof):
      for _, trial in success_trials.iterrows():
        plt.axvline(x=trial['total_t_ms'],
                    linestyle='--',
                    # ymin = trial['target_pos'][i] - targetsize,
                    # ymax = trial['target_pos'][i] + targetsize,
                    color='g')

      for _, trial in failure_trials.iterrows():
        plt.axvline(x=trial['total_t_ms'],
                    linestyle='--',
                    color='r')

    plt.xlabel("Timesteps")
```

```
    plt.ylabel("Position")
    plt.xlim((0, t_max))
    plt.ylim((0, 1))
    plt.show()
```

Train model

In [10]:
```
def init_config(env_version):
    algo_config = {}
    algo_config['evaluation_num_workers'] = 0
    algo_config['evaluation_parallel_to_training'] = False
    algo_config['num_gpus'] = 1
    algo_config['num_rollout_workers'] = 8
    algo_config['num_envs_per_worker'] = 1

    # Change config settings
    # Create a PPOConfig object
    ppo_config = PPOConfig()\
        .environment(env=f"bmi-v-{env_version}")\
        .framework(framework="torch")\
        .debugging(seed=415, log_level="ERROR")\
        .evaluation(
            evaluation_interval=15,
            evaluation_duration=5,
            evaluation_num_workers=algo_config['evaluation_num_workers'],
            evaluation_parallel_to_training=algo_config['evaluation_parallel_to_traini
            evaluation_config = dict(
                explore=False,
                num_workers=4,
            ),)\
        .rollouts(
            num_rollout_workers=algo_config['num_rollout_workers'],
            num_envs_per_worker=algo_config['num_envs_per_worker'],)\
        .training(
            gamma=0.99,
            lr=1e-4)\
        .resources(
            num_gpus=algo_config['num_gpus']
            )
    return ppo_config
```

In [11]:
```
def train_model(model_config, end_it, start_it = 1, env_version = "0"):
    num_iterations = end_it
    ppo_algo = model_config.build()

    checkpoint_dir = f'{ROOT_DIR}saved_runs/ppo_{env_version}/'

    if start_it > 1:
        checkpoint =f"{checkpoint_dir}checkpoint_{(start_it-1):06d}"
        ppo_algo.restore(checkpoint)

    f_reward_path = f'{ROOT_DIR}reward_data/v{env_version}.txt'

    start_time = time.time()
    ppo_rewards  = []
```

```
    with open(f_reward_path,"a+") as f_reward:
        for i in range(start_it, end_it):
            # Call its `train()` method
            result = ppo_algo.train()

            # Extract reward from results.
            ppo_rewards.append(result["episode_reward_mean"])

            # checkpoint and evaluate every 10 iterations
            if ((i % 10 == 0) or (i == num_iterations-1)):
                line_str = f"Iteration={i}, Mean Reward={result['episode_reward_mean']
                try:
                    line_str += f"+/-{np.std(ppo_rewards ):.4f}"
                except:
                    pass
                # save checkpoint file
                checkpoint_file = ppo_algo.save(checkpoint_dir)
                line_str += f"\nCheckpoints saved at {checkpoint_file}\n"

                f_reward.write(line_str)
                print(line_str, end="")
                # evaluate the policy
                eval_result = ppo_algo.evaluate()

    # To stop the Algorithm (and Env) and release its blocked resources, use:
    ppo_algo.stop()

    # convert num_iterations to num_episodes
    num_episodes = len(result["hist_stats"]["episode_lengths"]) * num_iterations
    # convert num_iterations to num_timesteps
    num_timesteps = sum(result["hist_stats"]["episode_lengths"] * num_iterations)
    # calculate number of wins
    num_wins = np.sum(result["hist_stats"]["episode_reward"])

    # train time
    secs = time.time() - start_time
    print(f"PPO won {num_wins:.2f} times over {num_episodes} episodes ({num_timesteps}
    print(f"Approx {num_wins/num_episodes:.4f} wins per episode")
    print(f"Training took {secs:.2f} seconds, {secs/60.0:.2f} minutes")
```

In [12]:
```
def load_model(config, env_version, checkpoint_version):
    checkpoint_dir = f'{ROOT_DIR}saved_runs/ppo_{env_version}/'

    checkpoint =f"{checkpoint_dir}checkpoint_{(checkpoint_version):06d}"
    print(f"\n{checkpoint}")

    algo = config.build()
    algo.restore(checkpoint)

    return algo
```

Main code

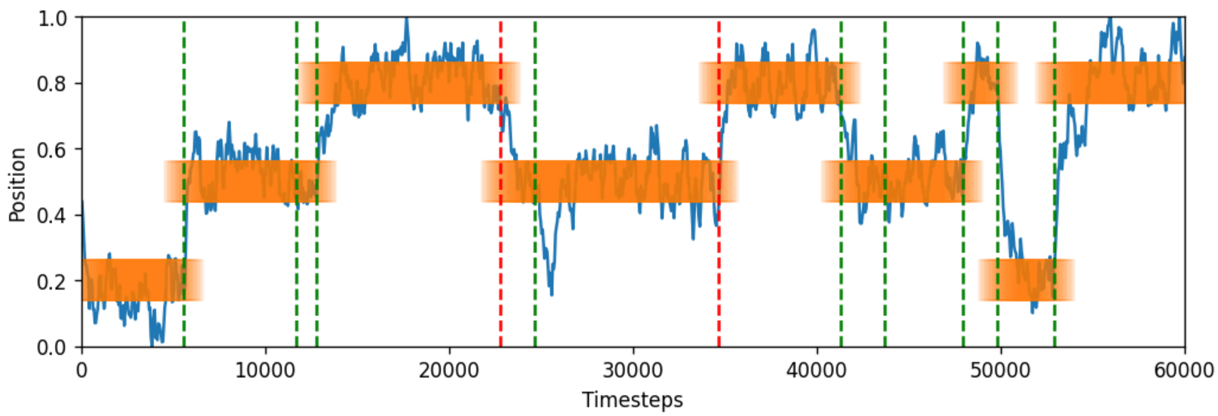
In [13]:
```
env_version = "2.3.0.3"

# To start fresh, restart Ray in case it is already running
if ray.is_initialized():
    ray.shutdown()

env = init_env(env_version)
```

In [14]:
```
# baseline model
resultsdf_base = run_model(num_episodes = 100, env=env)
plot_simulated_data(resultsdf_base, t_max=60e3)
```

```
**************
Mean Reward=0.6230+/-0.5083
got 62.30 reward over 100 episodes (10714 timesteps)
Approx 0.6230 reward per episode
won 80 over 100 episodes
**************
```



In [15]:
```
# Registering in Ray
tune.register_env(f"bmi-v-{env_version}", lambda config: EnvCompatibility(env))

ppo_config = init_config(env_version)

no_of_iterations = 1500
# train_model(ppo_config, end_it=1+no_of_iterations, env_version=env_version)
```

Evaluate model

In [16]:
```
algo = load_model(ppo_config, env_version, checkpoint_version=no_of_iterations)
```

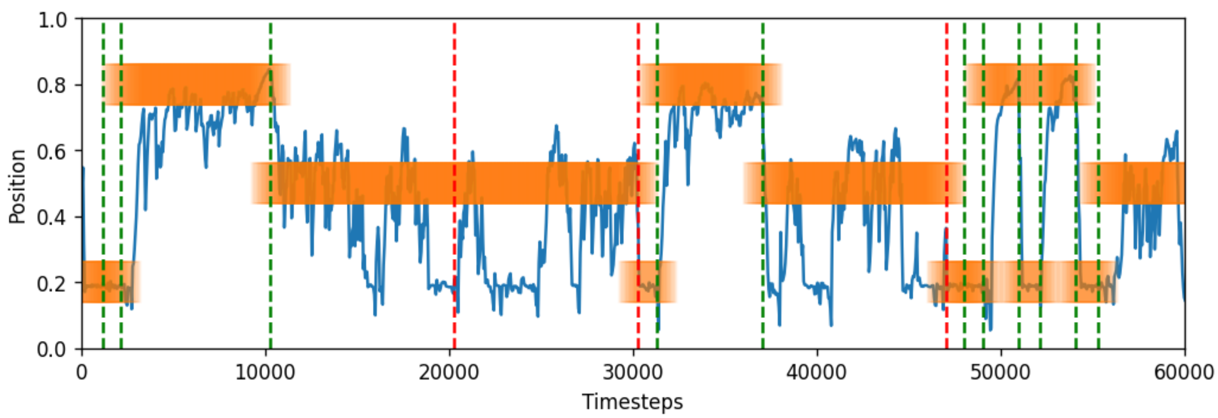
```
saved_runs/ppo_2.3.0.3/checkpoint_001500
```

```
2023-04-27 16:18:18,370 INFO worker.py:1553 -- Started a local Ray instance.
2023-04-27 16:18:27,836 INFO trainable.py:172 -- Trainable.setup took 11.942 seconds.
If your trainable is slow to initialize, consider setting reuse_actors=True to reduce
actor creation overheads.
2023-04-27 16:18:27,846 WARNING util.py:67 -- Install gputil for GPU system monitorin
g.
2023-04-27 16:18:27,981 INFO trainable.py:791 -- Restored on 127.0.0.1 from checkpoin
t: saved_runs\ppo_2.3.0.3\checkpoint_001500
2023-04-27 16:18:27,981 INFO trainable.py:800 -- Current state after restoring: {'_it
eration': 1500, '_timesteps_total': None, '_time_total': 16850.89587712288, '_episode
s_total': 55268}
```

In [17]:

```
resultsdf_model = run_model(model=algo, num_episodes = 100, env=env)
plot_simulated_data(resultsdf_model, t_max=60e3)
```

```
**************
Mean Reward=0.3785+/-0.7732
got 37.85 reward over 100 episodes (9798 timesteps)
Approx 0.3785 reward per episode
won 66 over 100 episodes
**************
```



Plotting rewards v/s iterations

In [20]:

```
def plot_rewards(filename, title, label=None, color_label='b', horiz_line = True):
  text_file = open(f'{ROOT_DIR}reward_data/{filename}', "r")
  text = text_file.read()
  text_file.close()

  text_list = text.rstrip().split('\n')
  it_list = []
  mu_list = []
  std_list = []

  for line in text_list[::2]:
    it = int(re.search(r'Iteration=(\d*),', line).group(1))
    reward_mu = float(re.search(r'Reward=(.*)\+\/\-', line).group(1))
    reward_std = float(re.search(r'\+\/\-(.*)', line).group(1))
    if len(it_list) and it <= it_list[-1]:
      it_list, mu_list, std_list = [], [], []
    it_list.append(it)
    mu_list.append(reward_mu)
```

```
    std_list.append(reward_std)

  it_list = np.array(it_list)
  mu = np.array(mu_list)
  std = np.array(std_list)

  plt.plot(it_list, mu, color=color_label, label=label)
  if horiz_line:
    plt.axhline(y = 1, color = 'g', linestyle = '--')
    plt.axhline(y = 0, color = 'orange', linestyle = '--')

  plt.xlabel('Iterations')
  plt.ylabel('Reward')
  plt.title(title)
```

In [21]:

```
plot_rewards(filename='v3.2.txt', title='', label="2 targets", color_label='#ffc300')
plot_rewards(filename='v3.3.txt', title='', label="3 targets", color_label='#ff5733')
plot_rewards(filename='v3.4.txt', title='', label="4 targets", color_label='#c70039')
plot_rewards(filename='v3.5.txt', title='', label="5 targets", color_label='#900c3f')
plot_rewards(filename='v3.6.txt', title='', label="6 targets", color_label='#581845')

plt.legend(loc='lower right')
plt.show()
```