MotorNet: a Python Toolbox for Controlling Biomechanical Effectors with Artificial Neural Networks

Olivier Codol[12], Jonathan A Michaels[1234], Mehrdad Kashefi[1234], J Andrew Pruszyski[1234], Paul L Gribble[1235]

[1] Brain and Mind Institute **[2]** Department of Psychology **[3]** Department of Physiology & Pharmacology, Schulich School of Medicine & Dentistry, **[4]** Robarts Research Institute, at University of Western Ontario, ON, Canada; **[5]** Haskins Laboratories, in New Haven CT, USA

Problem

Artificial neural networks (ANNs) are a popular class of computational models for studying neural control of movement.

Typically, their use requires a biomechanical simulator and a neural network software to work together to efficiently train meaningful models.

This leads to two impracticalities:

(1) Researchers must rely on two different, independent platforms, forcing tinkering and preventing creation of streamlined software pipelines

(2) Effectors are not differentiable, constraining researchers to reinforcement learning algorithms to train networks.

Solution

Base requirements:

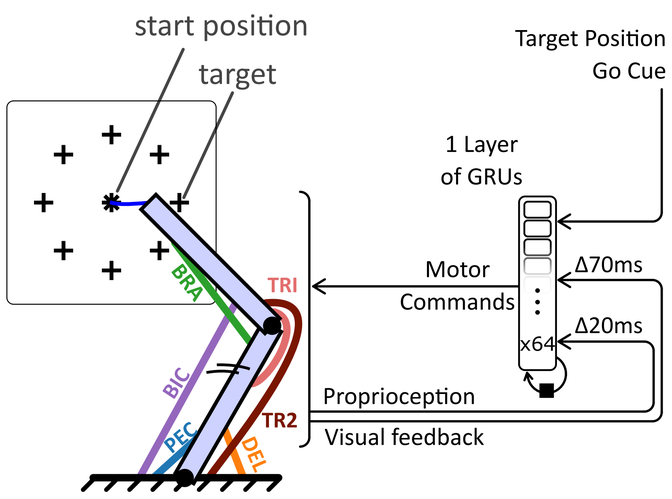
No dependency. Biomechanical simulation and ANN training on the same platform.

Differentiable effectors. To enable backpropagation through the plant.

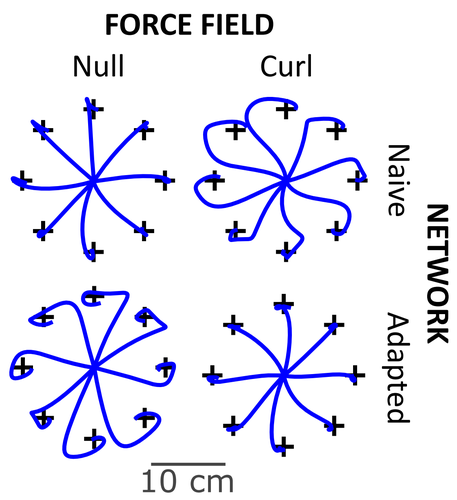
MotorNet was built to meet these requirements. It requires no dependency beyond typical pip or anaconda libraries, and all biomechanical models are built as tensors, allowing for backpropagation through the plant. Additionally:

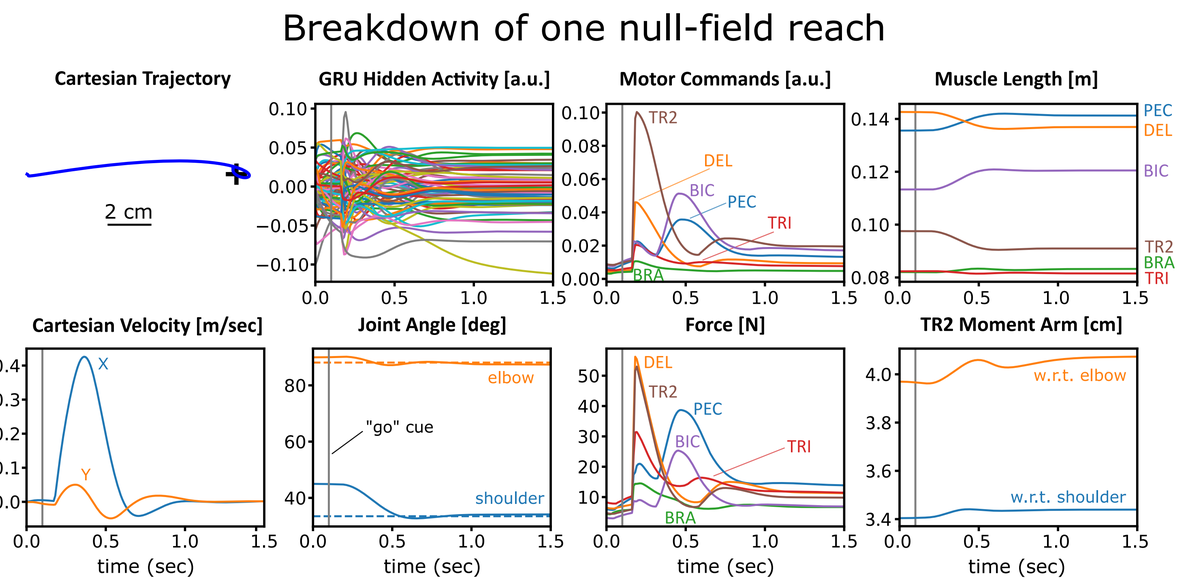
- It is open-source, so individual contributions are possible (and even welcome).
- Install simply using **pip install motornet**, useful for remote computing.
- High-level and fully documented API : **https://oliviercodol.github.io/MotorNet**
- Modular, easily customizable and expandable set of effectors.
- Based on TensorFlow. Anything TensorFlow can build can be used as a controller.

Example: Curl Field Adaptation in a Centre-Out Reaching Task



GRU: Gated Recurrent Unit





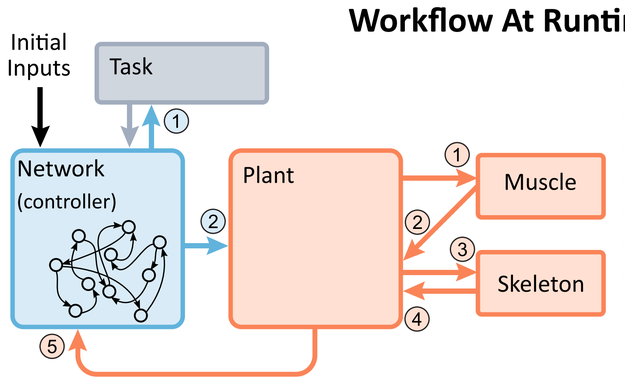
MotorNet's Architecture Is Modular

There are 5 base classes, each with pre-built subclasses, as indicated in the table below.

SKELETON	MUSCLE	PLANT	NETWORK	TASK
PointMass	ReLu (torque actuator)	RigidTendonArm26	GRUNetwork	CentreOutReach
Arm2	Rigid Tendon Hill Muscle [1]	ReluPointMass24		DelayedReach
	Rigid Tendon Hill Muscle [2]	CompliantTendonArm26		DelayedMultiReach
	Compliant Tendon Hill Muscle [1]			

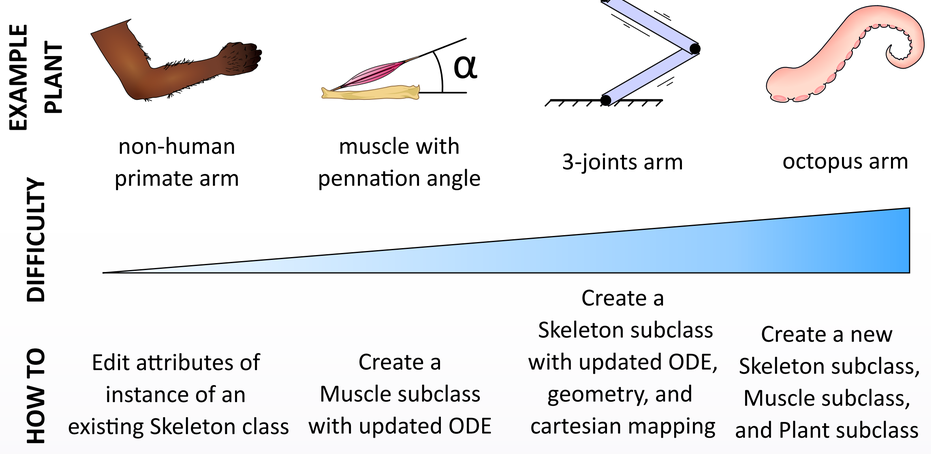
Based on the formalization by Kistemaker et al. (2010) [1] and Thelen (2003) [2]

Workflow At Runtime



1. recompute inputs [optional]
2. send new motor commands
1. pass on motor commands
2. return forces
3. send moment-adjusted forces
4. return new joint & cartesian states
5. return feedback states

This enables users to create their own biomechanical model, controller architecture, and task design by subclassing these object classes.



Example Use of the API

```python
import motornet as mn
import tensorflow as tf

# Create plant
skeleton = mn.plants.skeletons.TwoDofArm()
muscle = mn.plants.muscles.RigidTendonHillMuscle()
plant = mn.plants.RigidTendonArm26(
  muscle_type=muscle,
  skeleton=skeleton)

# Create network
network = mn.nets.layers.GRUNetwork(
  plant=plant,
  n_units=50,
  kernel_regularizer=10**-6)
task = mn.tasks.CentreOutReach(network=network)

# Map symbolic inputs to outputs
inputs = task.get_input_dict_layers()
state0 = task.get_initial_state_layers()
rnn = tf.keras.layers.RNN(
  cell=network,
  return_sequences=True)
state1 = rnn(inputs, initial_state=state0)
```

```python
# Create model
optimizer = tf.optimizers.Adam(clipnorm=1.)
model = mn.nets.MotorNetModel(
  inputs=[inputs, state0],
  outputs=state1,
  task=task)
model.compile(
  optimizer=optimizer,
  loss=task.losses,
  loss_weights=task.loss_weights)

# Train model
batch_size = 64
with tf.device('/device:GPU:0'):
    [x, y, init_states] = model.task.generate(
      n_timesteps=80,
      batch_size=500 * batch_size)
    model.fit(
      x=[x, init_states],
      y=y,
      epochs=1,
      batch_size=batch_size,
      shuffle=False)
```

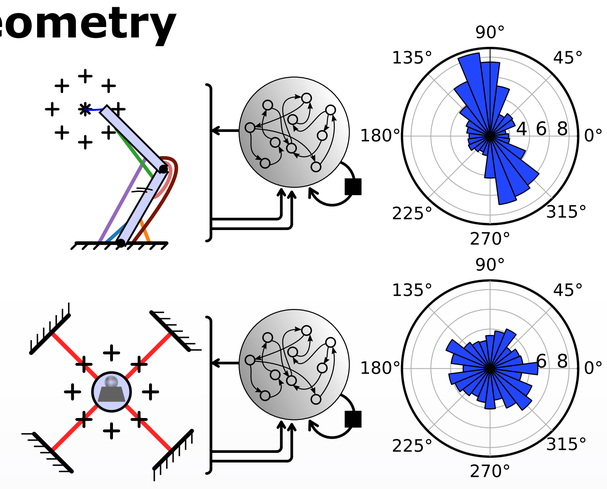
The above code would create and train a model for a two-joint planar arm with 6 muscles as in Kistemaker et al. (2010)

Replicating the Directional Representation Bias from Plant Geometry

Lillicrap & Scott (2013) show that the directional representation bias in the primary motor cortex is a direct consequence of the effector's geometry. Using MotorNet and a more recent GRU network architecture, we attempt to reproduce that finding.

In a 1-layer, 90-GRUs network, the representation bias occurs for a Plant with an arm-like geometry (top row), but not for one with a point-mass geometry (bottom row).

The polar histograms on the right indicate the average count of preferential movement direction across 8 networks trained to reach at random targets from a random start position across the full joint space.



Plant geometry that the network was trained to control in a centre-out reaching task

Polar histogram of preferential movement direction [GRU count]

REFERENCES

Kistemaker DA, Wong JD, Gribble PL (2010). The Central Nervous System Does Not Minimize Energy Cost in Arm Movements. J. Neurophysiol. 104, 2985–2994. **Lillicrap, TP, Scott SH (2013)**. Preference Distributions of Primary Motor Cortex Neurons Reflect Control Solutions Optimized for Limb Biomechanics. Neuron 77, 168–179. **Thelen DG (2003)**. Adjustment of Muscle Mechanics Model Parameters to Simulate Dynamic Contractions in Older Adults. J. Biomech. Eng. 125, 70–77.

NSERC grant # RGPIN-2018-05458 to PLG
CIHR grant # PJT-156241 to PLG
Correspondance should be addressed at codol.olivier@gmail.com