Neural Network Control of an Euler-Lagrange System

Jason Nezvadovitz

Controller Formulation

State Dynamics

$$M(q)\ddot{q} + V(q,\dot{q}) + G(q) + F(\dot{q}) = u$$

Tracking Error

$$e \coloneqq q_d - q$$

$$r \coloneqq \dot{e} + \alpha e$$

Error Dynamics

$$\dot{r} = \ddot{e} + \alpha\dot{e}$$

$$= \ddot{q}_d - \ddot{q} + \alpha\dot{e}$$

$$\therefore \quad M\dot{r} = M\ddot{q}_d - M\ddot{q} + M\alpha\dot{e}$$

$$= M\ddot{q}_d + M\alpha\dot{e} + V + G + F - u$$

Definition of NN

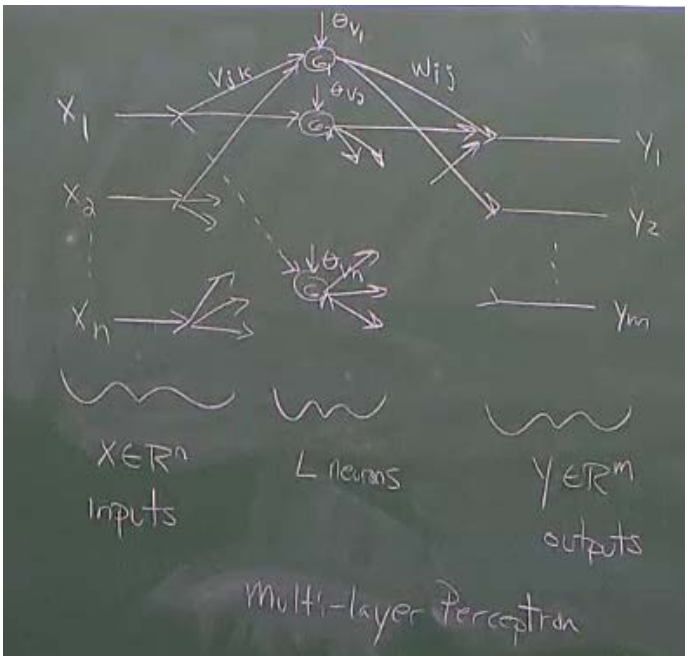
(n inputs, m outputs, 1 hidden layer with L neurons of a *differentiable* basis σ)

$$y \coloneqq W^T\sigma(V^T x) \quad \in R^m$$

$$x \coloneqq \begin{bmatrix} 1 \\ q \\ \dot{q} \end{bmatrix} \in R^{n+1}$$

$$V^T \coloneqq \begin{bmatrix} \theta_{V1} & V_{11} & \cdots & V_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{VL} & V_{L1} & \cdots & V_{Ln} \end{bmatrix} \in R^{L\times(n+1)}$$

$$W^T \coloneqq \begin{bmatrix} \theta_{W1} & W_{11} & \cdots & W_{1L} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{Wm} & W_{m1} & \cdots & W_{mL} \end{bmatrix} \in R^{m\times(L+1)}$$



Universal Approximation Assumption

Suppose that x starts and remains on a compact set. Then we can make y mimic any function up to some approximation error ϵ. Let's choose the function we want to approximate as,

$$f(x) \coloneqq M\ddot{q}_d + M\alpha\dot{e} + V + G + F$$

and intend for our NN output y to mimic f(x) up to ϵ as,

$$y = f(x) - \epsilon$$

This allows the error dynamics to be written as,

$$M\dot{r} = y + \epsilon - u$$

Approximation Estimation Errors

We do not initially know what choice of W and V will yield this ϵ-close approximation, so we define estimates of those "ideal" weights as,

$$\hat{y} \coloneqq \widehat{W}^T\sigma\left(\hat{V}^T x\right), \quad \text{where} \quad \widetilde{W} \coloneqq W - \widehat{W} \quad \text{and} \quad \tilde{V} \coloneqq V - \hat{V}$$

We now set out to select u such that x actually does remain on a compact set and also drives the tracking error r towards zero as quickly and accurately as possible.

Effort Design

Consider the candidate Lyapunov function,

$$v \coloneqq \frac{1}{2} r^T M r + \frac{1}{2}\mathrm{tr}\left(\widetilde{W}^T K_W^{-1} \widetilde{W}\right) + \frac{1}{2}\mathrm{tr}\left(\tilde{V}^T K_V^{-1} \tilde{V}\right)$$

where K_W and K_V are positive-definite (L+1 by L+1) and (n+1 by n+1) gain matrices respectively and tr is the trace operator (sum of diagonal elements) which satisfies,

$$a^T b = \mathrm{tr}(\mathrm{ba^T}) \quad \text{and} \quad \mathrm{tr}(-\mathrm{a}) = -\mathrm{tr}(a)$$

The derivative is then,

$$\dot{v} = r^T M \dot{r} - \mathrm{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \mathrm{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

$$= r^T (y + \epsilon - u) - \mathrm{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \mathrm{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

Recognizing the need for feedback on r and feedforward with the estimated network, design,

$$\boxed{\begin{aligned} u &\coloneqq Kr + \hat{y} \\ &= Kr + \widehat{W}^T \sigma\left(\hat{V}^T x\right) \end{aligned}}$$

for some positive-definite gain matrix K. Thus the closed-loop Lyapunov derivative is,

$$\dot{v} = r^T \left(-Kr + \epsilon + W^T \sigma(V^T x) - \widehat{W}^T \sigma\left(\hat{V}^T x\right)\right) - \mathrm{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \mathrm{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

Adaptation Design

Adding and subtracting $W^T \sigma\left(\hat{V}^T x\right)$,

$$\dot{v} = r^T \left(-Kr + \epsilon + \widetilde{W}^T \sigma\left(\hat{V}^T x\right) + W^T \left(\sigma(V^T x) - \sigma\left(\hat{V}^T x\right)\right)\right) - \mathrm{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \mathrm{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

The factor $\sigma(V^T x) - \sigma\left(\hat{V}^T x\right)$ could be simplified if σ were linear, so we examine its Taylor series. For clarity we define,

$$\sigma_o \coloneqq \sigma(a)\Big|_{a = \hat{V}^T x} \quad \text{and} \quad \sigma_o' \coloneqq \frac{d\sigma}{da}(a)\Big|_{a = \hat{V}^T x}$$

so that its series expansion about $\hat{V}^T x$ is,

$$\sigma(V^T x) = \sigma_o + \sigma_o'\left(V^T x - \hat{V}^T x\right) + \cdots O^2\left(V^T x - \hat{V}^T x\right)$$

Remembering that,

$$V^T x - \hat{V}^T x = \tilde{V}^T x$$

we have,

$$\sigma(V^T x) - \sigma(\hat{V}^T x)$$

$$= \sigma_o + \sigma_o'(V^T x - \hat{V}^T x) + \cdots O^2(V^T x - \hat{V}^T x) - \sigma_o$$

$$= \sigma_o' \tilde{V}^T x + O^2(\tilde{V}^T x)$$

Therefore,

$$\dot{v} = r^T(-Kr + \epsilon + \widehat{W}^T \sigma_o + W^T \sigma_o' \tilde{V}^T x + W^T O^2) - \text{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \text{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

We now add and subtract $\widehat{W}^T \sigma_o' \tilde{V}^T x$ so that,

$$\dot{v} = r^T(-Kr + \epsilon + \widetilde{W}^T \sigma_o + \widehat{W}^T \sigma_o' \tilde{V}^T x + \widetilde{W}^T \sigma_o' \tilde{V}^T x + W^T O^2) - \text{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \text{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

If we can design $\dot{\widehat{W}}$ and $\dot{\hat{V}}$ in such a way that $\widetilde{W}$ and $\tilde{V}$ remain bounded, then the terms $\widetilde{W}^T \sigma_o' \tilde{V}^T x$ and $W^T O^2$ can be bounded by a constant times ||x||. This inclines using a projection for the update laws,

$$\dot{\widehat{W}} = proj(a) \quad \text{and} \quad \dot{\hat{V}} = proj(b)$$

where a and b will be decided upon later. The projection algorithm will be defined as,

$$\dot{\gamma} = proj(a) = \begin{cases} a, & \gamma_{min} < \gamma < \gamma_{max} \\ a, & \gamma = \gamma_{min} \ \ and \ \ a > 0 \\ a, & \gamma = \gamma_{max} \ \ and \ \ a < 0 \\ 0, & \gamma = \gamma_{min} \ \ and \ \ a < 0 \\ 0, & \gamma = \gamma_{max} \ \ and \ \ a > 0 \end{cases}$$

for some specified γ_{min} and γ_{max}. Thus for some constant vector c,

$$\dot{v} \leq r^T(-Kr + \epsilon + c||x|| + \widetilde{W}^T \sigma_o + \widehat{W}^T \sigma_o' \tilde{V}^T x) - \text{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) - \text{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

$$\leq -r^T K r + r^T \epsilon + r^T c||x|| + r^T \widetilde{W}^T \sigma_o - \text{tr}\left(\widetilde{W}^T K_W^{-1} \dot{\widehat{W}}\right) + r^T \widehat{W}^T \sigma_o' \tilde{V}^T x - \text{tr}\left(\tilde{V}^T K_V^{-1} \dot{\hat{V}}\right)$$

Applying the property of the trace operator previously defined,

$$r^T \widetilde{W}^T \sigma_o = \text{tr}(\widetilde{W}^T \sigma_o r^T)$$

$$r^T\widehat{W}^T\sigma_o'\tilde{V}^Tx = \mathrm{tr}\left(\tilde{V}^Txr^T\widehat{W}^T\sigma_o'\right)$$

Thus,

$$\dot{v} \leq -r^TKr + r^T\epsilon + r^Tc||x|| + \mathrm{tr}\left(\widetilde{W}^T\sigma_o r^T - \widetilde{W}^TK_W^{-1}\dot{\widehat{W}}\right) + \mathrm{tr}\left(\tilde{V}^Txr^T\widehat{W}^T\sigma_o' - \tilde{V}^TK_V^{-1}\dot{\hat{V}}\right)$$

which (substituting back the definition of σ_o) dictates the adaptation laws to be,

$$\dot{\widehat{W}} = proj\left(K_W\sigma\left(\hat{V}^Tx\right)r^T\right)$$

$$\dot{\hat{V}} = proj\left(K_Vxr^T\widehat{W}^T\sigma'\left(\hat{V}^Tx\right)\right)$$

(recall that),

$$\sigma'(a) = \begin{bmatrix} \frac{\partial\sigma_1}{\partial a_1} & \cdots & \frac{\partial\sigma_1}{\partial a_L} \\ \vdots & \ddots & \vdots \\ \frac{\partial\sigma_{L+1}}{\partial a_1} & \cdots & \frac{\partial\sigma_{L+1}}{\partial a_L} \end{bmatrix}$$

This leaves,

$$\dot{v} \leq -r^TKr + r^T\epsilon + r^Tc||x||$$

$$\leq -||K||||r||^2 + ||\epsilon||||r|| + c||r||||x||$$

The first two terms can be upperbounded by examining the binomial they complete,

$$\dot{v} \leq \frac{||\epsilon||^2}{4||K||} + c||r||||x||$$

Thus by Khalil's UUB theorem, the result is UUB for as long as K is chosen sufficiently based on the initial condition of x. This is a semi-global uniformly ultimately bounded result. A global result could be obtained had a DCAL approach been taken (define the neural network input as x_d instead of x) but this result, while more robust to large initial conditions, tends to provide worse adaptation because it maps desired states to actual dynamics instead of actual states to actual dynamics. Additionally, had we modified the control effort to,

$$u \coloneqq Kr + \hat{y} + K_s sgn(r)$$

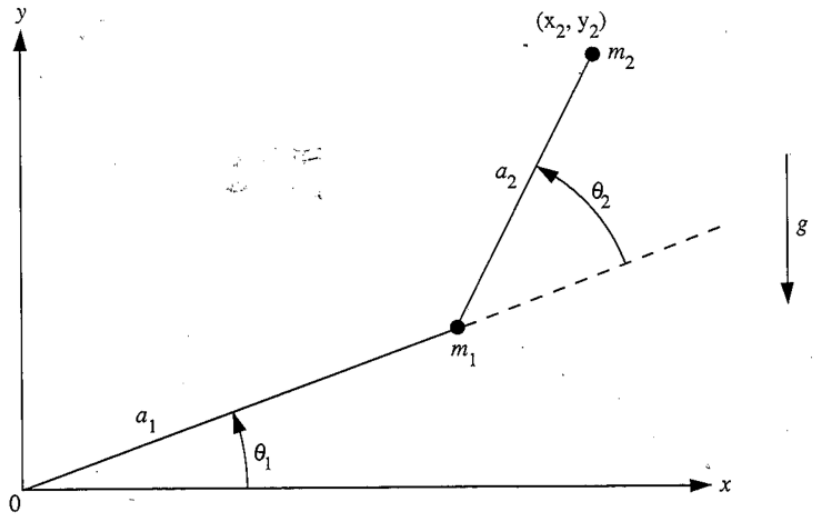
then a global asymptotic result could be obtained because the Lyapunov derivative would have an additional $r^TK_s sgn(r)$ term in it to dominate the residuals, however it would be a discontinuous controller that would be prone to jitter and harsh actuator efforts in practice.

Simulation

Model

We will use, as an example of an Euler-Lagrange system, the two-link upright robotic manipulator subject to gravity and joint friction.

The derivation of the dynamics can be found in a number of introductory mechanics and robotics texts.



Following the derivation outlined in "Robot Manipulator Control Theory and Practice" by Frank Lewis, we arrive at the following dynamics.

$$\begin{bmatrix} m_1 + m_2)\, a_1^2 + m_2 a_2^2 + 2m_2 a_1 a_2 \cos\theta_2 & m_2 a_2^2 + m_2 a_1 a_2 \cos\theta_2 \\ m_2 a_2^2 + m_2 a_1 a_2 \cos\theta_2 & m_2 a_2^2 \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix}$$

$$+ \begin{bmatrix} -m_2 a_1 a_2 (2\dot{\theta}_1 \dot{\theta}_2 + \dot{\theta}_2^2) \sin\theta_2 \\ m_2 a_1 a_2 \theta_1^2 \sin\theta_2 \end{bmatrix} + \begin{bmatrix} (m_1 + m_2)\, g a_1 \cos\theta_1 + m_2 g a_2 \cos(\theta_1 + \theta_2) \\ m_2 g a_2 \cos(\theta_1 + \theta_2) \end{bmatrix}$$

$$= \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}$$

where τ is the motor torque at each joint. If we define a position state vector,

$$q := \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

and control input vector,

$$u \coloneqq \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}$$

then these dynamics can be expressed concisely as a typical Euler-Lagrange system,

$$M(q)\ddot{q} + V(q, \dot{q}) + G(q) = u$$

However, this simple model does not account for joint friction. Suppose we have an arm with significant static friction effects in the joints. We reconcile this with a Coulomb friction term,

$$F(\dot{q}) \coloneqq \beta \tanh(c\dot{q})$$

$$\Rightarrow \ -\beta \leq F(\dot{q}) \leq \beta$$

$$\boxed{M(q)\ddot{q} + V(q, \dot{q}) + G(q) + F(\dot{q}) = u}$$

The simulation used the following true parameters:

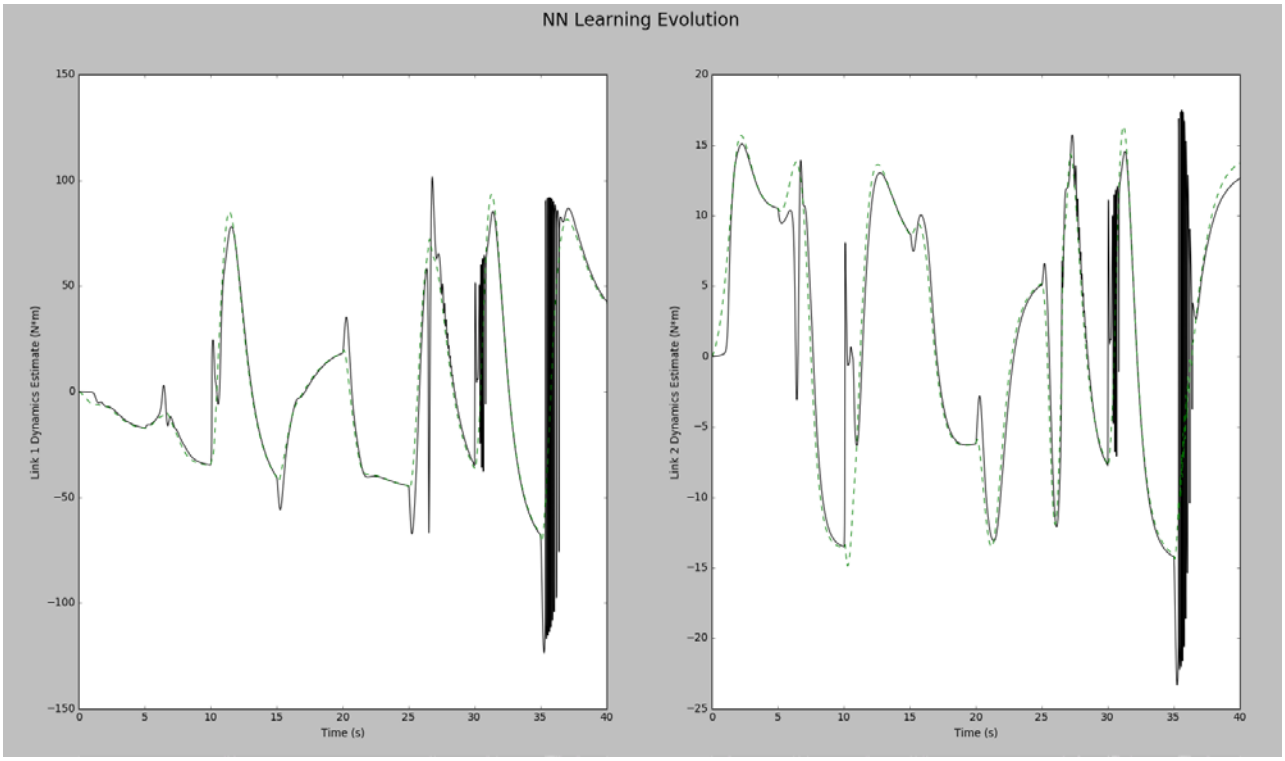
- m_1 = 5 kg
- m_2 = 3 kg
- a_1 = 1 m
- a_2 = 0.5 m
- g = 9.81 m/s^2
- β = [1, 1] N*m
- c = [2, 2] s/rad
- u_{limits} = [250, 30] N*m

Tuning

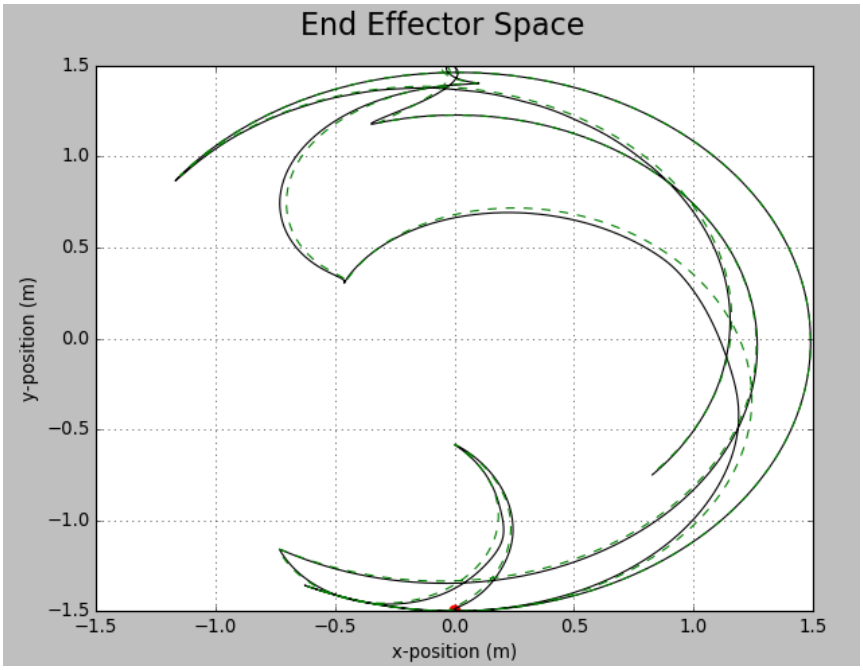
A randomly generated C2 continuous path was generated for the joint angles to track (a sequence of linear model-references tracking random waypoints). While attempting to track this trajectory, the following tuning was found to work very well (without sliding mode).

- K = 100*eye(2)
- α = eye(2)
- $\sigma(\cdot)$ = [tanh(·)] * 10 neurons
- K_V = 10*eye(5)
- K_W = 10*eye(11)

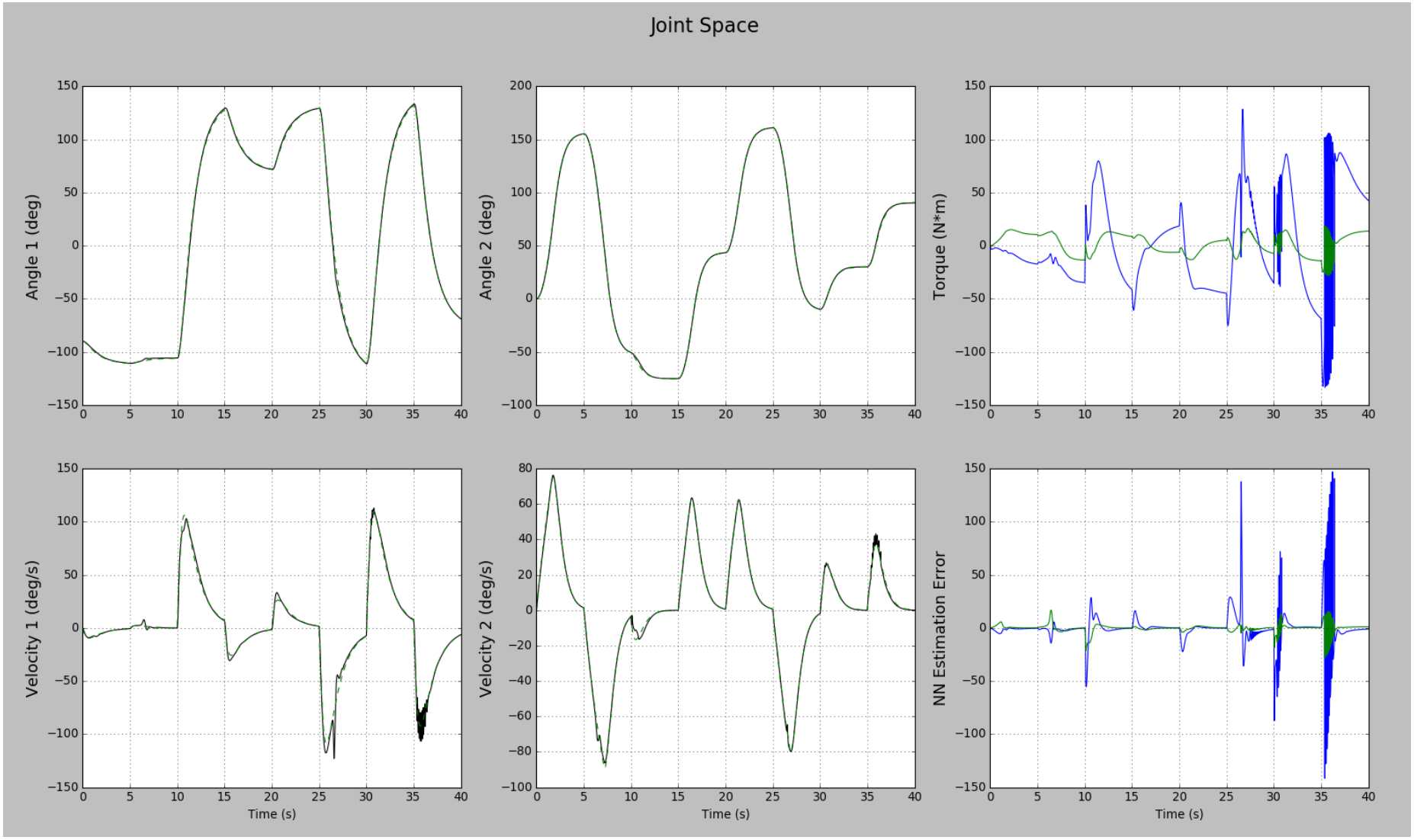
Results



The green line is the true value of f(x) during the simulation, while the black line is the value of $\hat{y}$. Evidently, the NN very quickly adapts to match the system dynamics.



The green is the desired path, while the black line is the actual path followed. Good tracking.


Joint Space
Angle 1 (deg)
Angle 2 (deg)
Torque (N*m)
Velocity 1 (deg/s)
Velocity 2 (deg/s)
NN Estimation Error
Time (s)
Time (s)
Time (s)

The addition of the sliding mode term barely influenced the result, adding only what seemed like a small layer of jitter over the already pretty-darn-close tracking. Plots are omitted, seriously they look extremely similar. Additionally, with 10 neurons there are nearly 100 different weights, which would be ridiculous to plot. Their result (the first plot) is better.

Code Excerpts

Controller

```
107
108     def get_effort(self, q, dt):
109         """
110         Returns the vector of torques as a PD controller plus
111         a feedforward term that uses an estimate of the system's
112         dynamics. The output is saturated at umax as
113         specified by the user previously. Before returning the
114         torques, the latest dynamics estimate is also updated.
115 
116         """
117         # Tracking errors
118         E = self.qref[:2] - q[:2]
119         Edot = self.qref[2:] - q[2:]
120         r = self.kr*E + Edot
121 
122         # Control law
123         u = self.kp*E + self.kd*Edot + self.y
124 
125         # Adapt NN
126         if not self.saturated:
127             x = np.concatenate(([1], q))
128             VTx = self.V.T.dot(x)
129             Wdot = self.kw.dot(np.outer(self.sig(VTx), r))
130             Vdot = self.kv.dot(np.outer(x, r).dot(self.W.T).dot(self.sigp(VTx)))
131             self.W = self.W + Wdot*dt
132             self.V = self.V + Vdot*dt
133             self.y = self.W.T.dot(self.sig(self.V.T.dot(x)))
134 
135         # Update reference trajectory and controller life time
136         self.update_ref(dt)
137         self.time = self.time + dt
138 
139         # Safety saturation of output
140         self.saturated = False
141         for i, mag in enumerate(abs(u)):
142             if mag > self.umax[i]:
143                 u[i] = self.umax[i] * np.sign(u[i])
144                 self.saturated = True
145 
146         # Return effort torques
147         return u
148 
```

Reference Generator

```
150
151     def update_ref(self, dt):
152         """
153         Updates the reference state qref depending on the
154         settings created in set_path. In every case, a
155         spring-damper tuned to vmax and amax is used to
156         generate the profile between each discontinuous target.
157 
158         'train': sequence of random joint-space configurations
159 
160         """
161         self.path_time = self.path_time + dt
162 
163         if self.path_type == 'train':
164             Eref = self.target[:2] - self.qref[:2]
165             Erefdot = -self.qref[2:]
166             uref = self.kp*Eref + self.kd*Erefdot
167             self.qref = self.qref + self.reference_dynamics(self.qref, uref)*dt
168             if self.path_time > 5:
169                 self.set_path(self.qref, 2*np.pi*(np.random.rand(2) - 0.5), 'train', dt)
170 
171         else:
172             raise ValueError("Invalid path_type.")
173 
```

Dynamics

```
def dynamics(q, u):
    """
    Returns state derivative (qdot).
    Takes control input (u) and current state (q).

    """
    # Externally set parameters
    global L, m, g, b, c, umax

    # Mass matrix M(q)
    M = np.zeros((2, 2))
    M[0, 0] = (m[0]+m[1])*L[0]**2 + m[1]*L[1]**2 + 2*m[1]*L[0]*L[1]*np.cos(q[1])
    M[0, 1] = m[1]*L[1]**2 + m[1]*L[0]*L[1]*np.cos(q[1])
    M[1, 0] = M[0, 1]  # symmetry
    M[1, 1] = m[1]*L[1]**2

    # Centripetal and coriolis vector V(q)
    V = np.array([
                  -m[1]*L[0]*L[1]*(2*q[2]*q[3]+q[3]**2)*np.sin(q[1]),
                   m[1]*L[0]*L[1]*q[2]**2*np.sin(q[1])
                ])

    # Gravity vector G(q)
    G = np.array([
                  g*(m[0]+m[1])*L[0]*np.cos(q[0]) + m[1]*g*L[1]*np.cos(q[0]+q[1]),
                  m[1]*g*L[1]*np.cos(q[0]+q[1])
                ])

    # Joint damping D(q)
    D = np.array([
                  d[0]*q[2],
                  d[1]*q[3]
                ])

    # Joint friction
    F = np.array([
                  b[0]*np.tanh(c[0]*q[2]),
                  b[1]*np.tanh(c[1]*q[3])
                ])

    # Record joint friction for examining the unstructured disturbance
    global unstruct_history, i
    unstruct_history[i, :] = F + D

    # Vibration noise introduced in an industrial environment
    f = vibe_mean + vibe_stdv*np.random.randn(2)

    # Actuator saturation
    for i, mag in enumerate(abs(u)):
        if mag > umax[i]:
            u[i] = umax[i] * np.sign(u[i])

    # [theta1dot, theta2dot] = [w1, w2]   and   [w1dot, w2dot] = (M^-1)*(u-V-G-D-F)
    return np.concatenate((q[2:], np.linalg.inv(M).dot(u + f - V - G - D - F)))
```

Discussion

- Tuning was surprisingly easy. I have been using K=100 alpha=1 for a while now with this system, not that a PD controller alone yields anything remotely close to tracking. When I threw in the NN, I just decided arbitrarily to make the learning gains 10 each, pick 10 tanh neurons, and of course try K_s = 0 to start with. This worked shockingly well, I literally ran out of my room and immediately showed all of my roommates. I was so surprised at how incredibly well the NN predicted f(x) that I got greedy. Let's try 50 neurons, 100! It went unstable. I found that upping the number of neurons (probably due to overfit?) causes it to become extremely sensitive, just like upping the learning gains. I have a feeling that the performance was so good because my learning gains (10) are really pretty high, and that in the real world noise would cause it to freak-out. But even lowering them, the learning is still very good.
- Like you saw, the tracking error was very small as soon as the NN (almost immediately) picked up on the dynamics. However, this tight tracking came quickly due to high learning gains. In real life we wouldn't want it to be so sensitive.
- Most NN train on data and optimize over their data fit. This method uses only an error based adaptation law, and thus it is *drastically faster*.
- Compared to linear regression techniques (Y*theta), this method is far more general and easier to derive (same for every time-invariant system!), however, to get good adaptation it has to be way more sensitive. For example, in project one, noise and random disturbances didn't destroy tracking, but the NN doesn't like that at all.
- Programming was actually much easier than, say, concurrent learning. There are no lofty regressors, no history stacks or integral stacks to manage, and the adaptation laws are pretty simple especially if you just use Euler integration.