

PID vs. Neural Network Control

How the control playground works: the robot models, the controllers, the training method and the MATLAB code

Akin Tatoglu, Ph.D. · Lecture notes · October 2026

1. What the playground does

Two controllers drive the same simulated robot side by side. The left one is a PID controller whose three gains you set by hand. The right one is a small neural network that learned to control the robot by trial and error. Both see the same sensors, drive the same motors and face the same stress tests: a heavier load, more friction, a push and a noisy sensor.

The playground offers four robots: a single arm joint, a self-balancing (Segway-style) robot, a two-link manipulator and a quadrotor drone. These notes give the math behind each part. Every number in them comes from the same simulation, and the MATLAB code in Section 8 reproduces it.

2. The simulation

Each robot is a set of differential equations stepped forward in time with semi-implicit Euler integration. Velocities update first, then positions use the new velocities, which keeps the motion stable at a 10 ms step (5 ms for the manipulator).

$$\omega_{k+1} = \omega_k + \dot{\omega}_k \Delta t, \quad \theta_{k+1} = \theta_k + \omega_{k+1} \Delta t$$

Every control command is limited to the motor's range before it is applied, as on real hardware.

Robot	Task	Control input	Key parameters
Arm joint	Hold 60°, then 100° at t = 2.5 s	Torque, $\pm 12 \text{ N}\cdot\text{m}$	$m = 1 \text{ kg}$, $l = 0.5 \text{ m}$, $b = 0.08 \text{ N}\cdot\text{m}\cdot\text{s}/\text{rad}$
Self-balancing robot	Stay upright; drive to 1 m, then back to 0 at t = 4 s	Wheel force, $\pm 20 \text{ N}$	Base $M = 1 \text{ kg}$, body $m = 0.6 \text{ kg}$ at $l = 0.4 \text{ m}$, drag $f = 0.3 \text{ N}\cdot\text{s}/\text{m}$
Two-link manipulator	Gripper to point A, then point B at t = 2.5 s	$\pm 25 \text{ N}\cdot\text{m}$ shoulder, $\pm 12 \text{ N}\cdot\text{m}$ elbow	Links 0.40 m / 1.2 kg and 0.35 m / 0.8 kg
Quadrotor drone	Hover at 1 m; then 2 m and 1 m right at t = 3 s	Thrust 0–25 N, tilt torque $\pm 1 \text{ N}\cdot\text{m}$	$m = 1 \text{ kg}$, $I = 0.01 \text{ kg}\cdot\text{m}^2$, drag $c_d = 0.15$

Arm joint

The angle θ is measured from hanging straight down, so gravity torque is largest at 90° . Here $d(t)$ is the push disturbance.

$$I\ddot{\theta} = \tau - b\dot{\theta} - mgl\sin\theta + d(t), \quad I = ml^2 + 0.02$$

Self-balancing robot

The robot is a cart-pole: a wheeled base with an inverted pendulum on top. The tilt ϕ is measured from upright, and the robot counts as fallen past 60° .

$$\ddot{\phi} = \frac{g\sin\phi - \cos\phi \frac{F - f\dot{x} + ml\dot{\phi}^2 \sin\phi}{M + m}}{I\left(\frac{4}{3} - \frac{m\cos^2\phi}{M + m}\right)}$$

$$\ddot{x} = \frac{F - f\dot{x} + ml\dot{\phi}^2 \sin\phi}{M + m} - \frac{ml\ddot{\phi}\cos\phi}{M + m}$$

To drive forward, the base must first roll backward so the body leans forward. This is called non-minimum-phase behavior.

Two-link manipulator

The shoulder angle q_1 is measured from straight down and the elbow angle q_2 is relative to the first link. With link centers of mass $c_i = L_i/2$ and $I_i = m_i L_i^2/12$:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) + B\dot{q} = \tau$$

$M_{11} = m_1 c_1^2 + I_1 + m_2(L_1^2 + c_2^2 + 2L_1 c_2 \cos q_2) + I_2$, $M_{12} = m_2(c_2^2 + L_1 c_2 \cos q_2) + I_2$, $M_{22} = m_2 c_2^2 + I_2$. Because M depends on q_2 , the torque one joint needs depends on what the other joint is doing.

Quadrotor drone

The drone moves in a vertical plane. Total rotor thrust T acts along the body axis, and a torque τ tilts it. Tilting sends part of the thrust sideways, which is how it moves left or right. Here $w(t)$ is the wind gust.

$$m\ddot{x} = -T\sin\theta + w(t) - c_d\dot{x}, \quad m\ddot{y} = T\cos\theta - c_d\dot{y} - mg, \quad I\ddot{\theta} = \tau$$

3. PID control

A PID controller adds three reactions to the error $e = r - y$: one to the error now, one to its accumulated history and one to the rate of motion.

$$u(t) = K_p e(t) + K_i \int_0^t e(s) ds - K_d \dot{y}(t), \quad e = r - y$$

In discrete form, with saturation (sat) on both the integral and the output:

$$u_k = K_p e_k + K_i \text{sat}\left(\sum_{j \leq k} e_j \Delta t\right) - K_d \dot{y}_k, \quad u_k \leftarrow \text{sat}(u_k, \pm u_{\max})$$

- **Derivative on measurement.** Using $-K_d \dot{y}$ instead of $K_d \dot{e}$ avoids a spike in the motor command when the target jumps.
- **Anti-windup.** The integral is clamped (± 2 rad·s on the arm). Without a clamp it keeps growing while the motor is saturated and causes a large overshoot afterward.
- **Saturation.** The command is limited to the motor's range, so a gain set that looks perfect on paper can still fail.

Cascade control for unstable robots

The balancing robot and the drone move by tilting, so a single PID cannot control both position and balance. Each uses a fixed outer loop that turns position error into a target tilt, and the inner loop you tune tracks that tilt (balancing robot) or altitude (drone):

$$\phi_{\text{ref}} = \text{sat}(0.2(x_r - x) - 0.25\dot{x}, \pm 0.15), \quad F = K_p(\phi - \phi_{\text{ref}}) + K_i \int (\phi - \phi_{\text{ref}}) dt + K_d \dot{\phi}$$

$$T = \text{sat}(K_p e_y + K_i \int e_y dt - K_d \dot{y}, 0, T_{\max}), \quad \theta_{\text{ref}} = -\text{sat}(0.35 e_x - 0.5\dot{x}, \pm 0.3)$$

The drone's PID gets no hover thrust for free. At hover, T must equal $mg = 9.81$ N, and only the integral term can supply that steady part. The manipulator runs one PID per joint with the same gains, and the elbow uses 45% of them because it carries less load.

Change	Robot	Result in the simulation
$K_i = 0$	Arm joint	Stops about 9° short of 100°
$K_i = 0$	Manipulator	Gripper stops about 7 cm short of point B
$K_i = 0$	Drone	Hovers about 33 cm below 2 m
$K_d = 0$	Arm joint	Overshoots about 100% and never settles
$K_d = 0$	Self-balancing robot	Falls over at 0.9 s
$K_d = 0$	Drone	Overshoots about 80% on the first climb

4. The neural network controller

The network replaces the PID formula with a learned function. It has one hidden layer of tanh neurons, and its output passes through tanh again so the command stays within the motor's range.

$$u = u_{max} \tanh(W_2 \tanh(W_1 x + b_1) + b_2)$$

Robot	Inputs x	Hidden	Outputs	Weights
Arm joint	e, 0.3 ω , sin θ , cos θ , sin r	10	torque	71
Self-balancing	e _x , 0.5 $\dot{x}$, 3 ϕ , 0.5 $\dot{\phi}$, sin ϕ , 0.5r	12	wheel force	97
Manipulator	2 joint errors, 2 speeds, sin and cos of both link angles, sin of both target angles	14	2 torques	184
Drone	e _y , e _x , 0.3 $\dot{y}$, 0.3 $\dot{x}$, sin θ , cos θ , 0.1 $\dot{\theta}$, 0.5r _y	12	thrust, torque	134

The sine and cosine inputs let the network learn gravity directly, because gravity torque is proportional to sin θ . The network sees only the current state, though, not the history of the error. It has no integral term, so it cannot notice that a steady error has persisted and push harder.

5. Training by evolution strategies

The network is trained by trial and error. Each try is scored by the cost J of an episode, with random targets, on the normal robot only. The balancing robot also pays for tilt and pays heavily for falling; the drone pays for tilt and uses its distance to the target as e.

$$J(p) = \frac{1}{N} \sum_{k=1}^N (e_k^2 + 0.002 u_k^2) \text{ [+ tilt and fall penalties]}$$

Each training round makes n pairs of copies of the weights p, each nudged both ways along a random direction z_i, and scores them on the same episodes:

$$J_i^\pm = J(p \pm \sigma z_i), \quad z_i \sim \mathcal{N}(0, I), \quad i = 1, \dots, n$$

The scores are replaced by their ranks, which keeps one bad episode from dominating, and combined into a direction that improves the weights (Salimans et al., 2017):

$$g = \frac{1}{n\sigma} \sum_{i=1}^n (\rho(J_i^+) - \rho(J_i^-)) z_i, \quad \rho(J) = \frac{\text{rank}(J)}{2n-1} - \frac{1}{2}$$

The weights then take an Adam step (Kingma and Ba, 2015), where $\hat{m}$ and $\hat{v}$ are the bias-corrected averages. The pretrained networks used n = 20, $\sigma = 0.1$, $\eta = 0.05$ for 1,200 rounds (arm) or 800 rounds (the other robots).

$$m \leftarrow 0.9m + 0.1g, \quad v \leftarrow 0.999v + 0.001g^2, \quad p \leftarrow p - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + 10^{-8}}$$

Evolution strategies need only a score for each try, not a derivative of the simulator, which is why they suit a simulator written for clarity rather than for gradients.

6. The scoreboard

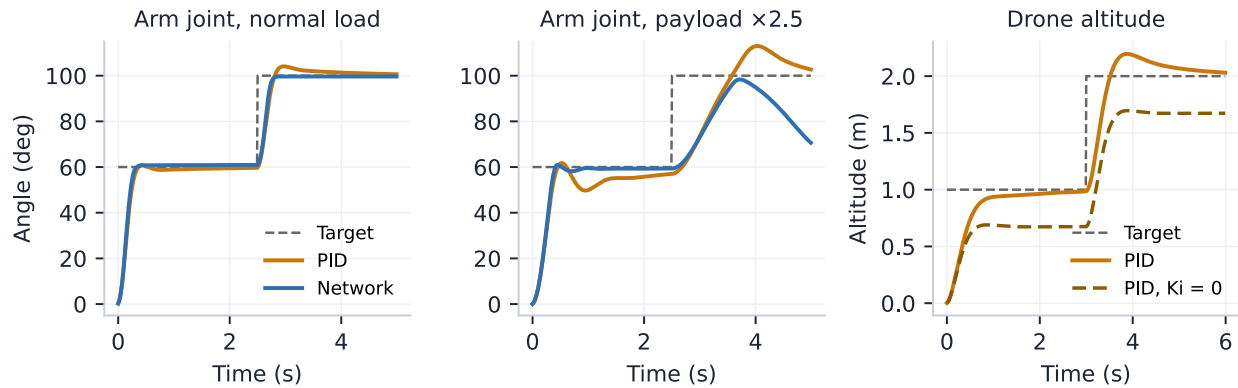
Overshoot and settling time are measured on the first move. Settling time is the last time the output is outside a band around the first target (1.2° for the arm, 5 cm for the balancing robot and drone, 2 cm for the manipulator). Final error is the mean error over the last 0.3 s.

$$OS = \max\left(0, \frac{\max_k (y_k - r_1)}{r_1 - y_0}\right) \times 100\%$$

$$e_{RMS} = \sqrt{\frac{1}{N} \sum_k (r_k - y_k)^2}, \quad \text{jitter} = \frac{100}{T} \sum_k |u_k - u_{k-1}| \quad [\%/s]$$

Motor jitter adds up how much the normalized command changes each step. High jitter means the motor is switched back and forth rapidly, which heats motors, wears gearboxes and drains batteries.

7. What the simulations show



Left and center: on the arm, the network tracks well at normal load but stops far short with a 2.5× payload, while PID's integral term recovers. Right: without K_i , the drone hovers below its target.

Robot	Condition	Measure	PID	Network
Arm joint	Normal	Final error	0.7°	0.4°
Arm joint	Payload ×2.5	Final error	3.7°	25.7°
Arm joint	Noisy sensor	Motor jitter	1,284 %/s	7,171 %/s
Arm joint	Push	Overshoot	18%	1%
Self-balancing	Normal	Final error	0.02 m	0.14 m
Self-balancing	Rough ground	Final error	0.73 m	0.14 m
Self-balancing	Noisy sensor	Motor jitter	1,279 %/s	6,607 %/s
Manipulator	Normal	Tracking error (RMS)	13.8 cm	10.3 cm
Manipulator	Normal	Motor jitter	89 %/s	1,035 %/s
Manipulator	Heavy object	Settling time	not settled	0.78 s
Drone	Normal	Settling time	1.56 s	0.50 s
Drone	Wind gust	Settling time	1.56 s	2.47 s
Drone	Noisy sensor	Motor jitter	492 %/s	2,157 %/s

Neither controller wins everywhere. Three patterns explain most of the table:

1. **Integral action beats learned compensation off-design.** The network learned gravity for the load it trained on. With a new load, its compensation is wrong and nothing corrects it. PID's integral term keeps adding torque until the error is gone.
2. **Learned controllers chatter.** Training rewarded fast tracking and barely penalized effort, so the networks learned near bang-bang commands, and sensor noise makes it worse.
3. **A fixed design can be the weak point.** On rough ground, the balancing robot's PID fails because its fixed outer loop was tuned for low drag. The network learned position and balance together and adapts better.

8. MATLAB code

The MATLAB files run the same robots and controllers as the playground. They need MATLAB R2019b or newer and no toolboxes. Unzip `arai_pid_vs_nn.zip`, open the folder in MATLAB, and run one section at a time with Ctrl+Enter.

File	What it does
lab1_pid_arm.m ... lab6_drone.m	Ready-to-run examples: PID tuning, training a network, stress tests, balancing robot, manipulator, drone
simulate_robot.m	Equations of motion for all four robots
controller_step.m	PID and network control laws
nn_features.m, nn_forward.m	Network inputs and forward pass
train_es.m, episode_cost.m	Evolution-strategies training and its score
compute_metrics.m	Scoreboard numbers
pretrained_weights.m	The playground's pretrained networks

A minimal session compares the default PID with the pretrained network on the arm:

```
P = arai_params('arm');
cond = default_conditions();
trP = simulate_robot(P, make_pid(P.pid), cond);
trN = simulate_robot(P, make_nn(pretrained_weights('arm')), cond);
plot_compare(P, {trP, trN}, {'PID', 'Network'});
print_metrics(P, {'PID', 'Network'}, [compute_metrics(P, trP), compute_metrics(P, trN)]);
```

Training a new network from scratch takes one line; 300 rounds take a few minutes on a laptop:

```
[p, loss] = train_es('arm', 300, 1, struct('pop', 12, 'eps', 2));
```

The main files follow. Results match the playground exactly, except runs with a noisy sensor, which use MATLAB's random number generator.

simulate_robot.m

```
function tr = simulate_robot(P, ctrl, cond, seed)
%SIMULATE_ROBOT Run one episode of a robot under a controller.
% tr = simulate_robot(P, ctrl, cond, seed)
% P robot settings from arai_params
% ctrl controller struct: make_pid(...) or make_nn(...)
% cond stress tests from default_conditions
% seed random seed for sensor noise (optional)
%
% tr.t time (s)
% tr.y main output (arm: angle rad, segway: position m, twolink: gripper distance m, drone:
altitude m)
% tr.ref target for tr.y
% tr.u normalized motor command, -1..1 (twolink: mean of both joints)
% tr.x state for plotting (arm: th, segway: [x tilt], twolink: [q1 q2], drone: [x y tilt xref])
% tr.fell true if the balancing robot fell over; tr.fellAt = time

if nargin < 4, seed = 7; end
rs = RandStream('mt19937ar', 'Seed', seed); % own stream: does not disturb rand/randn elsewhere
n = round(P.T/P.dt);
dt = P.dt;
tr.t = zeros(n,1); tr.y = zeros(n,1); tr.ref = zeros(n,1);
tr.u = zeros(n,1); tr.x = zeros(n,4); tr.e2 = zeros(n,1);
tr.fell = false; tr.fellAt = NaN;
st.i = [0 0]; % integral memory for PID
```

```

switch P.key
% ----- arm
case 'arm'
    m = 1*cond.payload; l = 0.5; I = m*l^2 + 0.02;
    b = 0.08; if cond.friction, b = 0.6; end
    th = 0; w = 0;
    for k = 1:n
        t = (k-1)*dt;
        ref = P.refs(1); if t >= P.switchT, ref = P.refs(2); end
        o.ref = ref; o.th = th; o.w = w;
        if cond.noise
            o.th = th + 0.03*randn(rs);
            o.w = w + 0.24*randn(rs);
        end
        o.err = ref - o.th;
        [u, st] = controller_step(P, ctrl, o, st);
        u = clampv(u, -1, 1);
        tau = u*P.umax(1);
        dist = 0; if cond.push && t >= 1.2 && t < 1.3, dist = -25; end
        w = w + (tau + dist - b*w - m*9.81*l*sin(th))/I*dt;
        th = th + w*dt;
        tr.t(k) = t; tr.y(k) = th; tr.ref(k) = ref; tr.u(k) = u; tr.x(k,1:2) = [th 0];
    end

% ----- segway
case 'segway'
    M = 1.0; m = 0.6*cond.payload; l = 0.4; g = 9.81;
    fr = 0.3; if cond.friction, fr = 3.0; end
    x = 0; v = 0; th = 3*pi/180; w = 0;
    for k = 1:n
        t = (k-1)*dt;
        ref = P.refs(1); if t >= P.switchT, ref = P.refs(2); end
        u = 0;
        if ~tr.fell
            o.ref = ref; o.x = x; o.v = v; o.th = th; o.w = w;
            if cond.noise
                o.x = x + 0.01*randn(rs); o.v = v + 0.05*randn(rs);
                o.th = th + 0.015*randn(rs); o.w = w + 0.1*randn(rs);
            end
            o.err = ref - o.x;
            [u, st] = controller_step(P, ctrl, o, st);
            u = clampv(u, -1, 1);
            F = u*P.umax(1);
            push = 0; if cond.push && t >= 2.0 && t < 2.1, push = 1.5; end
            tm = M + m; s = sin(th); c = cos(th);
            tmp = (F - fr*v + m*l*w^2*s)/tm;
            thacc = (g*s - c*tmp)/(l*(4/3 - m*c^2/tm)) + push/(m*l^2);
            xacc = tmp - m*l*thacc*c/tm;
            v = v + xacc*dt; x = x + v*dt;
            w = w + thacc*dt; th = th + w*dt;
            if abs(th) > 60*pi/180
                tr.fell = true; tr.fellAt = t;
            end
        else
            th = clampv(th + sign(th)*0.05, -pi/2, pi/2); % tips over
        end
        tr.t(k) = t; tr.y(k) = x; tr.ref(k) = ref; tr.u(k) = u; tr.x(k,1:2) = [x th];
    end

% ----- two-link
case 'twolink'
    m1 = 1.2; m2 = 0.8 + 0.6*(cond.payload - 1);
    L1 = P.L1; L2 = P.L2; c1 = L1/2; c2 = L2/2*(1 + 0.3*(cond.payload - 1));
    I1 = m1*L1^2/12; I2 = m2*L2^2/12; g = 9.81;
    b = 0.1; if cond.friction, b = 1.2; end
    q1 = 0; q2 = 0; w1 = 0; w2 = 0;

```

```

for k = 1:n
    t = (k-1)*dt;
    ref = P.refs(1,:); if t >= P.switchT, ref = P.refs(2,:); end
    o.ref = ref; o.q1 = q1; o.q2 = q2; o.w1 = w1; o.w2 = w2;
    if cond.noise
        o.q1 = q1 + 0.02*randn(rs); o.q2 = q2 + 0.02*randn(rs);
        o.w1 = w1 + 0.2*randn(rs); o.w2 = w2 + 0.2*randn(rs);
    end
    o.err = [ref(1) - o.q1, ref(2) - o.q2];
    [u, st] = controller_step(P, ctrl, o, st);
    u = clampv(u, -1, 1);
    t1 = u(1)*P.umax(1); t2 = u(2)*P.umax(2);
    push = 0; if cond.push && t >= 1.2 && t < 1.3, push = -30; end
    cs = cos(q2); sn = sin(q2);
    M11 = m1*c1^2 + I1 + m2*(L1^2 + c2^2 + 2*L1*c2*cs) + I2;
    M12 = m2*(c2^2 + L1*c2*cs) + I2;
    M22 = m2*c2^2 + I2;
    h = m2*L1*c2*sn;
    C1 = -h*(2*w1*w2 + w2^2);
    C2 = h*w1^2;
    G1 = (m1*c1 + m2*L1)*g*sin(q1) + m2*c2*g*sin(q1 + q2);
    G2 = m2*c2*g*sin(q1 + q2);
    r1 = t1 + push - b*w1 - C1 - G1;
    r2 = t2 - b*w2 - C2 - G2;
    dt_ = M11*M22 - M12^2;
    a1 = (M22*r1 - M12*r2)/dt_;
    a2 = (-M12*r1 + M11*r2)/dt_;
    w1 = w1 + a1*dt; w2 = w2 + a2*dt;
    q1 = q1 + w1*dt; q2 = q2 + w2*dt;
    p = tip_position(P, q1, q2);
    pr = tip_position(P, ref(1), ref(2));
    tr.t(k) = t; tr.y(k) = norm(p - pr); tr.ref(k) = 0;
    tr.u(k) = mean(abs(u)); tr.x(k,1:2) = [q1 q2];
end
% ----- drone
case 'drone'
    m = 1.0*cond.payload; I = 0.01*(1 + 0.5*(cond.payload - 1)); g = 9.81;
    cdrag = 0.15; if cond.friction, cdrag = 0.9; end
    x = 0; y = 0; vx = 0; vy = 0; th = 0; w = 0;
    for k = 1:n
        t = (k-1)*dt;
        ref = P.refs(1,:); if t >= P.switchT, ref = P.refs(2,:); end
        u = [0 0];
        if ~tr.fell
            o.ref = ref; o.x = x; o.y = y; o.vx = vx; o.vy = vy; o.th = th; o.w = w;
            if cond.noise
                o.x = x + 0.02*randn(rs); o.y = y + 0.02*randn(rs);
                o.vx = vx + 0.1*randn(rs); o.vy = vy + 0.1*randn(rs);
                o.th = th + 0.02*randn(rs); o.w = w + 0.2*randn(rs);
            end
            o.err = [ref(1) - o.y, ref(2) - o.x];
            [u, st] = controller_step(P, ctrl, o, st);
            u = clampv(u, -1, 1);
            T = (u(1) + 1)/2*P.umax(1); tau = u(2)*P.umax(2);
            wind = 0; if cond.push && t >= 1.5 && t < 2.0, wind = 6; end
            ax = (-T*sin(th) + wind - cdrag*vx)/m;
            ay = (T*cos(th) - cdrag*vy)/m - g;
            vx = vx + ax*dt; vy = vy + ay*dt; w = w + tau/I*dt;
            x = x + vx*dt; y = y + vy*dt; th = th + w*dt;
            if y < 0, y = 0; vy = max(vy, 0); end % ground
            if abs(th) > 80*pi/180
                tr.fell = true; tr.fellAt = t; % crashed
            end
        else
            vy = vy - g*dt; y = max(0, y + vy*dt);
            th = clampv(th + sign(th)*0.05, -pi/2, pi/2);
        end
    end
end

```

```

        tr.t(k) = t; tr.y(k) = y; tr.ref(k) = ref(1);
        tr.u(k) = mean(abs(u)); tr.x(k,:) = [x y th ref(2)];
        tr.e2(k) = norm([ref(1) - y, ref(2) - x]);
    end
end
end

```

controller_step.m

```

function [u, st] = controller_step(P, ctrl, o, st)
%CONTROLLER_STEP One control update. Returns normalized command(s) in -1..1.
% ctrl.type = 'pid' uses ctrl.gains = [Kp Ki Kd]
% ctrl.type = 'nn' uses ctrl.weights (row vector)

switch ctrl.type
    case 'pid'
        kp = ctrl.gains(1); ki = ctrl.gains(2); kd = ctrl.gains(3);
        switch P.key
            case 'arm'
                e = o.err;
                st.i(1) = clampv(st.i(1) + e*P.dt, -2, 2); % anti-windup
                u = (kp*e + ki*st.i(1) - kd*o.w)/P.umax(1); % derivative on measurement
            case 'segway'
                % Outer loop (fixed): position error -> target lean angle
                lean = clampv(0.2*(o.ref - o.x) - 0.25*o.v, -0.15, 0.15);
                % Inner loop (you tune): PID on tilt
                e = o.th - lean;
                st.i(1) = clampv(st.i(1) + e*P.dt, -1, 1);
                u = (kp*e + ki*st.i(1) + kd*o.w)/P.umax(1);
            case 'twolink'
                share = [1 0.45]; % elbow gets a smaller share of the gains
                w = [o.w1 o.w2];
                u = zeros(1,2);
                for j = 1:2
                    e = o.err(j);
                    st.i(j) = clampv(st.i(j) + e*P.dt, -1, 1);
                    u(j) = share(j)*(kp*e + ki*st.i(j) - kd*w(j))/P.umax(j);
                end
            case 'drone'
                % Altitude PID (you tune). No hover thrust is given: the integral must find it.
                e = o.err(1);
                st.i(1) = clampv(st.i(1) + e*P.dt, -2, 2);
                T = clampv(kp*e + ki*st.i(1) - kd*o.vy, 0, P.umax(1));
                % Fixed lateral cascade: x error -> target tilt -> torque
                thRef = -clampv(0.35*o.err(2) - 0.5*o.vx, -0.3, 0.3);
                tau = 0.5*(thRef - o.th) - 0.08*o.w;
                u = [T/P.umax(1)*2 - 1, tau/P.umax(2)];
        end
    case 'nn'
        x = nn_features(P, o);
        u = nn_forward(ctrl.weights, x, P.nn).';
    otherwise
        error('Unknown controller type "%s".', ctrl.type);
end
end

```

nn_features.m

```

function x = nn_features(P, o)
%NN_FEATURES What the network sees, as a column vector.
switch P.key
    case 'arm'
        x = [o.ref - o.th; o.w*0.3; sin(o.th); cos(o.th); sin(o.ref)];
    case 'segway'
        x = [clampv(o.ref - o.x, -1.5, 1.5); o.v*0.5; o.th*3; o.w*0.5; sin(o.th); o.ref*0.5];
    case 'twolink'

```

```

        r = o.ref;
        x = [r(1) - o.q1; r(2) - o.q2; o.w1*0.2; o.w2*0.2; ...
            sin(o.q1); cos(o.q1); sin(o.q1 + o.q2); cos(o.q1 + o.q2); ...
            sin(r(1)); sin(r(1) + r(2))];
    case 'drone'
        r = o.ref;
        x = [clampv(r(1) - o.y, -1.5, 1.5); clampv(r(2) - o.x, -1.5, 1.5); o.vy*0.3; o.vx*0.3; ...
            sin(o.th); cos(o.th); o.w*0.1; r(1)*0.5];
    end
end

```

nn_forward.m

```

function u = nn_forward(p, x, sz)
%NN_FORWARD One-hidden-layer tanh network. u = tanh(W2*tanh(W1*x + b1) + b2)
% p weight vector (row or column), layout matches the online playground
% x input column vector
% sz [nin nh nout]
nin = sz(1); nh = sz(2); nout = sz(3);
p = p(:);
W1 = reshape(p(1:nin*nh), nin, nh).'; % nh x nin
b1 = p(nin*nh + (1:nh));
o0 = nin*nh + nh;
W2 = reshape(p(o0 + (1:nh*nout)), nh, nout).'; % nout x nh
b2 = p(o0 + nh*nout + (1:nout));
u = tanh(W2*tanh(W1*x + b1) + b2);
end

```

train_es.m

```

function [p, lossHist] = train_es(key, gens, seed, opts)
%TRAIN_ES Train a network controller by evolution strategies (trial and error).
% [p, lossHist] = train_es('arm', 300, 1)
% [p, lossHist] = train_es('arm', 300, 1, struct('pop',12,'eps',2,'p0',p_start))
%
% Each round: make POP pairs of slightly changed copies of the weights,
% score each copy on EPS episodes with random targets, rank them, and move
% the weights toward the better copies (Adam step). Only the normal robot is
% used for training, so every stress test is new to the network.
%
% Method: Salimans et al. (2017), arXiv:1703.03864. Step size: Kingma & Ba (2015).

if nargin < 3 || isempty(seed), seed = 1; end
if nargin < 4, opts = struct(); end
pop = getopt(opts, 'pop', 12);
nEp = getopt(opts, 'eps', 2);
sig = getopt(opts, 'sigma', 0.1);
lr = getopt(opts, 'lr', 0.05);
showEvery = getopt(opts, 'showEvery', 10);

P = arai_params(key);
NP = P.nn(1)*P.nn(2) + P.nn(2) + P.nn(2)*P.nn(3) + P.nn(3);
rng(seed);
if isfield(opts, 'p0'), p = opts.p0(:); else, p = 0.25*randn(NP,1); end
mA = zeros(NP,1); vA = zeros(NP,1);
cond = default_conditions();
lossHist = zeros(gens,1);

t0 = tic;
for g = 1:gens
    % the same random targets for every copy this round (a fair comparison)
    targets = cell(1, nEp);
    for j = 1:nEp, targets{j} = random_targets(P); end

    Z = randn(NP, pop);
    F = zeros(2*pop, 1);

```

```

    for k = 1:pop
        F(2*k-1) = score(p + sig*Z(:,k));
        F(2*k)   = score(p - sig*Z(:,k));
    end
    % rank-based shaping: best = -0.5, worst = +0.5
    [~, order] = sort(F);
    rk = zeros(2*pop,1);
    rk(order) = (0:2*pop-1)/(2*pop-1) - 0.5;
    grad = Z*(rk(1:2:end) - rk(2:2:end))/(pop*sig);

    % Adam step (gradient descent on the ranked score)
    mA = 0.9*mA + 0.1*grad;
    vA = 0.999*vA + 0.001*grad.^2;
    mh = mA/(1 - 0.9^g); vh = vA/(1 - 0.999^g);
    p = p - lr*mh./(sqrt(vh) + 1e-8);

    lossHist(g) = mean(F);
    if mod(g, showEvery) == 0 || g == gens
        fprintf('round %4d of %d    error %.4f    (%.0f s)\n', g, gens, lossHist(g), toc(t0));
    end
end
p = p.';

function c = score(q)
    c = 0;
    for jj = 1:nEp
        Pj = P; Pj.refs = targets{jj};
        tr = simulate_robot(Pj, make_nn(q), cond, jj);
        c = c + episode_cost(P, tr);
    end
    c = c/nEp;
end

function v = getopt(s, name, default)
if isfield(s, name), v = s.(name); else, v = default; end
end

```

episode_cost.m

```

function c = episode_cost(P, tr)
%EPISODE_COST Training score for one episode (lower is better).
% Mean squared tracking error + a small penalty on motor effort.
% The balancing robot also pays for tilt and pays heavily for falling.
if strcmp(P.key, 'twolink')
    e = tr.y*3;
elseif strcmp(P.key, 'drone')
    e = tr.e2;
else
    e = tr.ref - tr.y;
end
c = e.^2 + 0.002*tr.u.^2;
if strcmp(P.key, 'segway')
    c = c + 2*tr.x(:,2).^2;
end
if strcmp(P.key, 'drone')
    c = c + 0.5*tr.x(:,3).^2;
end
c = mean(c);
if tr.fell
    c = c + 10*(1 + (P.T - tr.fellAt));
end
end

```

compute_metrics.m

```
function m = compute_metrics(P, tr)
%COMPUTE_METRICS Scoreboard numbers, in plotting units (deg, m or cm).
% m.os overshoot on the first move (%), NaN for the two-link arm
% m.settle settling time on the first move (s), NaN = not settled
% m.ss final error, mean over the last 0.3 s
% m.rms RMS tracking error over the whole run
% m.jitter total change in motor command per second (% of full command)
% m.fell true if the robot fell over

n = numel(tr.t);
k1 = round(P.switchT/P.dt);
y0 = tr.y(1); r1 = tr.ref(1);

if strcmp(P.key, 'twolink')
    m.os = NaN; % distance to target has no overshoot sign
else
    d = sign(r1 - y0); if d == 0, d = 1; end
    peak = max(d*(tr.y(1:k1) - r1));
    m.os = max(0, peak/abs(r1 - y0)*100);
end

out = find(abs(tr.y(1:k1) - tr.ref(1:k1)) > P.band, 1, 'last');
if isempty(out), settle = 0; else, settle = out*P.dt; end
if settle >= P.switchT - 0.05 || tr.fell, settle = NaN; end
m.settle = settle;

last = n - round(0.3/P.dt) + 1 : n;
m.ss = mean(abs(tr.ref(last) - tr.y(last)))*P.yScale;
m.rms = sqrt(mean((tr.ref - tr.y).^2))*P.yScale;
if tr.fell, m.ss = NaN; m.rms = NaN; end
m.jitter = sum(abs(diff(tr.u)))/P.T*100;
m.fell = tr.fell;
end
```

arai_params.m

```
function P = arai_params(key)
%ARAI_PARAMS Robot settings for the PID vs. Neural Network lab.
% P = arai_params('arm') single robot arm joint
% P = arai_params('segway') self-balancing (Segway-style) robot
% P = arai_params('twolink') two-link manipulator
% P = arai_params('drone') planar quadrotor drone
%
% ARAI Lab, University of Hartford. Matches the online playground.

d2r = pi/180;
P.key = key;
switch key
case 'arm'
    P.name = 'Robot arm joint';
    P.T = 5; % episode length (s)
    P.dt = 0.01; % time step (s)
    P.umax = 12; % motor torque limit (N*m)
    P.refs = [60 100]*d2r; % target 1, target 2 (rad)
    P.switchT = 2.5; % time the target changes (s)
    P.band = 1.2*d2r; % settling band (rad)
    P.yScale = 1/d2r; % plot in degrees
    P.yLabel = 'Joint angle (deg)';
    P.pid = [30 20 4]; % default Kp Ki Kd
    P.nn = [5 10 1]; % inputs, hidden neurons, outputs
case 'segway'
    P.name = 'Self-balancing robot';
    P.T = 8;
    P.dt = 0.01;
```

```

P.umax    = 20;           % wheel force limit (N)
P.refs    = [1.0 0.0];   % position targets (m)
P.switchT = 4;
P.band    = 0.05;        % m
P.yScale   = 1;
P.yLabel   = 'Position (m)';
P.pid      = [50 10 18];
P.nn       = [6 12 1];

case 'twolink'
    P.name    = 'Two-link manipulator';
    P.T       = 5;
    P.dt      = 0.005;
    P.umax    = [25 12];   % shoulder, elbow torque limits (N*m)
    P.refs    = [100 -60; 130 -50]*d2r; % each row: [q1 q2] for point A, point B
    P.switchT = 2.5;
    P.band    = 0.02;      % m
    P.yScale   = 100;      % plot in cm
    P.yLabel   = 'Gripper distance to target (cm)';
    P.pid      = [70 30 8];
    P.nn       = [10 14 2];
    P.L1      = 0.40;
    P.L2      = 0.35;

case 'drone'
    P.name    = 'Quadrotor drone';
    P.T       = 6;
    P.dt      = 0.01;
    P.umax    = [25 1.0];  % max total thrust (N), max tilt torque (N*m)
    P.refs    = [1.0 0.0; 2.0 1.0]; % each row: [altitude x] targets (m)
    P.switchT = 3;
    P.band    = 0.05;      % m
    P.yScale   = 1;
    P.yLabel   = 'Altitude (m)';
    P.pid      = [30 20 8]; % altitude PID
    P.nn       = [8 12 2];

otherwise
    error('Unknown robot "%s". Use arm, segway, twolink or drone.', key);
end
end

```

9. References

1. Ziegler, J. G., & Nichols, N. B. (1942). Optimum settings for automatic controllers. *Transactions of the ASME*, 64, 759-768.
2. Åström, K. J., & Hägglund, T. (2006). *Advanced PID Control*. ISA.
3. Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5), 834-846.
4. Hunt, K. J., Sbarbaro, D., Żbikowski, R., & Gawthrop, P. J. (1992). Neural networks for control systems: A survey. *Automatica*, 28(6), 1083-1112.
5. Grasser, F., D'Arrigo, A., Colombi, S., & Rufer, A. C. (2002). JOE: A mobile, inverted pendulum. *IEEE Transactions on Industrial Electronics*, 49(1), 107-114.
6. Spong, M. W., Hutchinson, S., & Vidyasagar, M. (2020). *Robot Modeling and Control* (2nd ed.). Wiley.
7. Salimans, T., Ho, J., Chen, X., Sidor, S., & Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. arXiv:1703.03864.
8. Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *ICLR*. arXiv:1412.6980.

© 2026 Akin Tatoglu, ARAI Lab, University of Hartford. Free for teaching and learning with attribution.