Essentials: Whirlwind Recap

EN.601.[4|6]98 Hands-on Robot Learning

Krishna Murthy Jatavallabhula

This lecture *is* dense

- We will step through a number of concepts (across mechanics, electronics, math)
- Most of these are intuitive. But, understanding and internalizing these takes substantial effort on your part.
- Payoff is high (you get an understanding of much of the robot stack, except for the robot learning bits, which the rest of the class is for!)
- If any of these are not quite clear, please reach out to the instructor during office hours, to discuss course preparedness
- You don't need to follow *every single detail*, but should be able to understand/appreciate the general intuition, and be able to apply this to coursework (given enough time)

By the end of the lecture, you'll be able to

- Understand the electromechanical construction of a robot manipulator (i.e., arm)
- Read a motor's datasheet and determine how it impacts your robot's abilities
- Recap forward and inverse kinematics, Jacobians, degrees of freedom, robot frames, singularities
- Establish a working intuition of basic feedback control
- Takeaway a list of ML/DL concepts you'll need to know to succeed in this class
- Map out the robot learning lifecycle, and how we'll step through this over the course of the semester

Administrivia

- Background and teaming survey has been posted on Canvas
- Please fill this out ASAP, or you will lose all influence in the teaming process
- There's a small break during class today. Please use that time to fill the survey out!
- Teams expected to be announced in time for next week's lecture (please fill out the survey)
- Course syllabus and guidelines+policies are posted on Canvas. Please talk to the instructor about any clarifications / concerns

Key questions we'll tackle

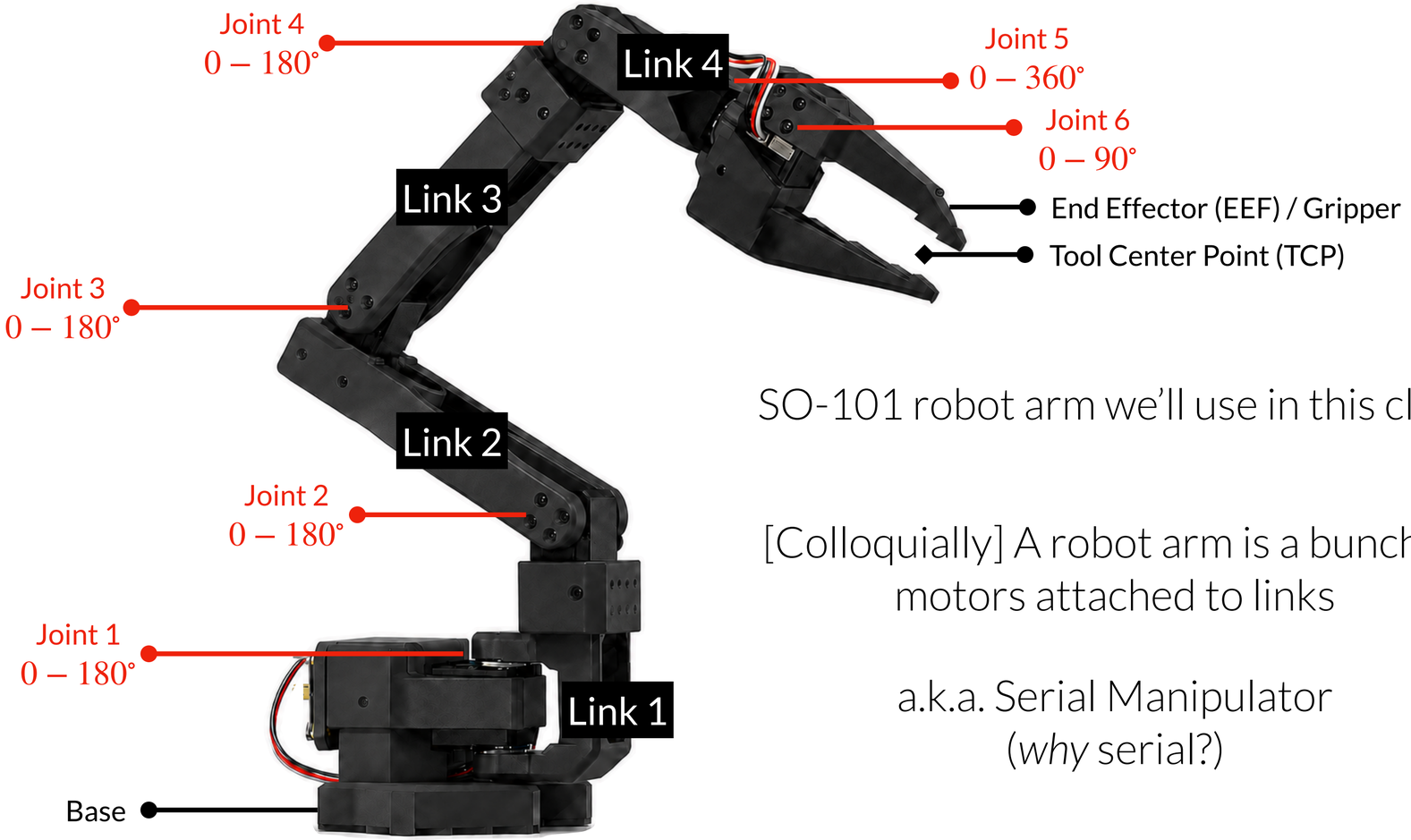
- How do we design and build a robot arm?
  - We will recap the anatomy of a *serial manipulator*
  - We will look at the electromechanical components of the arm — motors, circuits, etc.
  - And a quick overview of the SO-101 robot arm, and the jump from hobby to commercial arms
- How do we get the robot arm to move to where we'd like it to be?
  - a.k.a low-level robot *control*
  - We will recap simple, position-based control methods today
- How do we *learn* a neural network model that determines where the robot arm needs to move, to accomplish a task?
  - The rest of the semester will focus primarily on this

How do we design and build a robot arm?

It’s a pretty simple process

- Ask “*what is the arm for?*” (e.g., hobby tabletop arm, serious industrial arm, etc.)
- This gives you a rough sense of the kinds of motors you’d need (brushless DC, quasi-direct drive, smart servos, etc.)
- Use this to come up with initial designs (for hobby arms, a 3D printable design’s quite often the way to go)
- Then, ask “*what’s the budget like?*”
- Once reality hits hard, tear down the earlier design and build whatever the finances allow!
- Reality hit us pretty hard in this class, so we’ll stick to cheap, smart servos and 3D printed robot parts

What do we mean by ‘robot arm’?



SO-101 robot arm we’ll use in this class

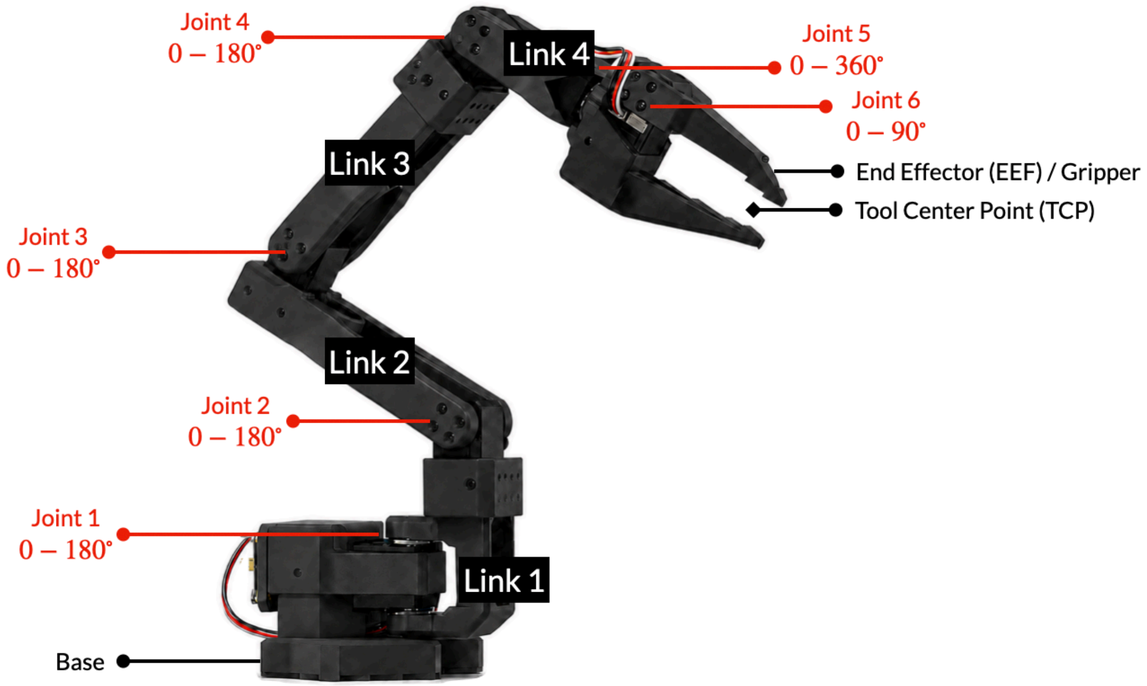
[Colloquially] A robot arm is a bunch of motors attached to links

a.k.a. Serial Manipulator
(*why* serial?)

[Less Colloquially] A robot arm is a set of rigid links connected by actuated joints. These joints are actuated by actuators (here, servo motors). The final link has an end effector that has a tool (typically a gripper) to interact with objects around it.

Anatomy of a robot manipulator

- A manipulator consists primarily of *links* and *joints*.
- In the course, *links* are always rigid bodies, and *joints* are always *revolute* and connect two links.
- Joints are actuated by low-cost servo motors
- Base frame defines the *world* (Cartesian) coordinate frame for the robot
- End-effector (EEF) is the final *flange* on which the gripper gets mounted



Terminology (EEF, gripper, flange, TCP)

- An *end-effector* (EEF) is the device at the end of the kinematic chain that interacts with the environment (e.g., a suction cup, a gripper, a probe, etc.)
- For the purposes of this class, a *gripper* is an EEF that grasps objects. The gripper has two *fingers/jaws*, one of which is stationary, and the other is driven by a motor
- A *flange* is the mechanical interface on the last link — the EEF attaches here. Forward kinematics is typically computed to the flange
- The *tool center point* (TCP) is a designated coordinate frame on the tool — commonly, this would be the center point between the two fingertips/jaws such that moving the gripper to TCP and closing it would result in an object situated at that point to be grasped

Common joint/actuator options for robot arms

Actuator / Joint	Commonly used in	Upsides	Downsides
Smart Servo	Hobby robots (e.g., SO-101)	Cheap, small, simple to work with	Backlash, heating, limited torque, more wear-and-tear
Servo joints for industrial robots	Many industrial arms	BLDC motor (precision + high-speed), high-res encoders and great braking	$$$, heavyweight, calibration is cumbersome
Quasi-direct / direct drive (QDD / DD)	Quadrupeds, Humanoids	Very low gear reduction ratios, very low backlash, backdrevable	Bulkier, high current draw, torque density (Nm/kg) is lower (much power draw supports motor weight alone)

Smart servo motors (used in SO-101)

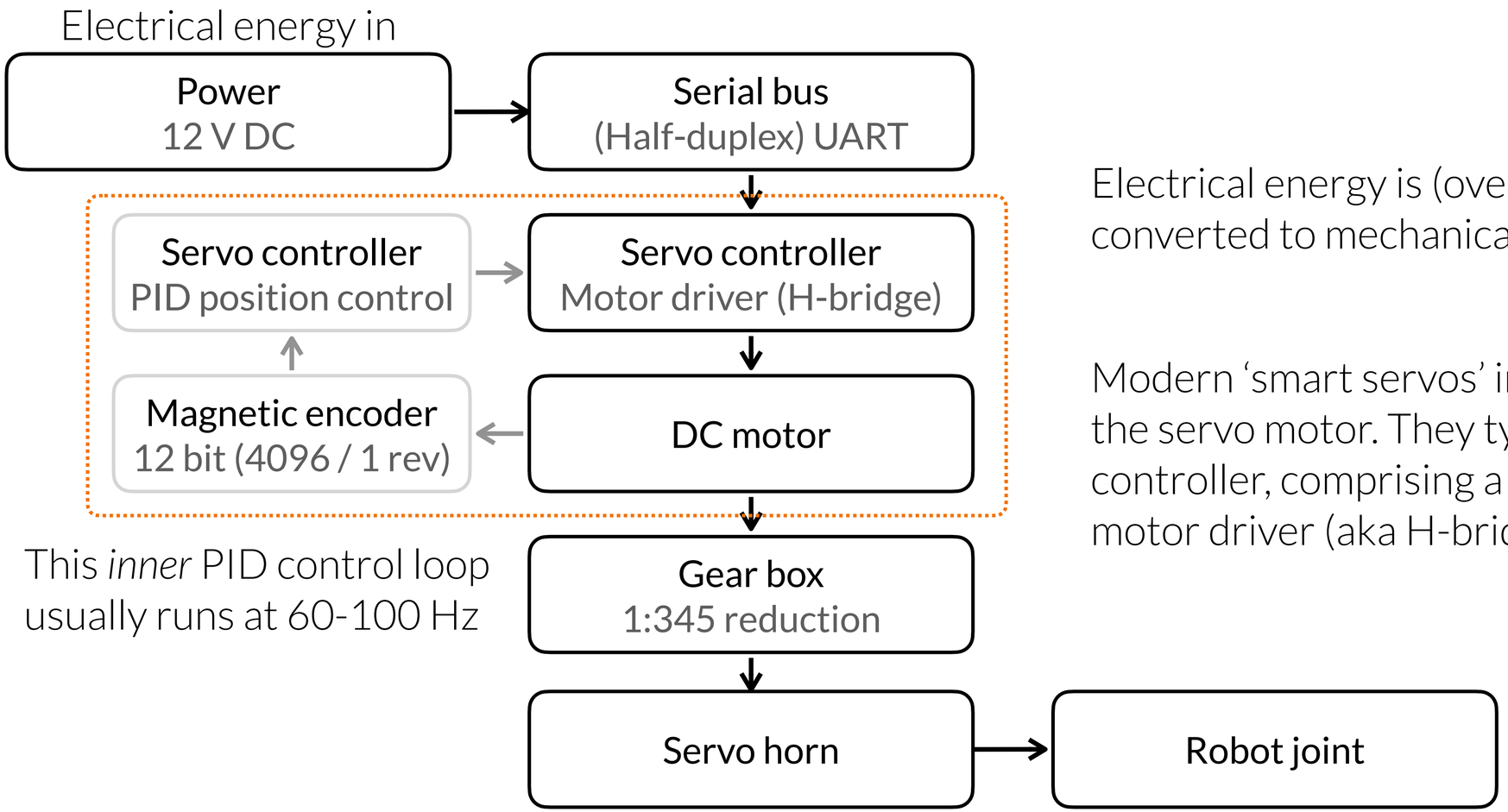
- The SO-101 arm is driven by servo motors (Feetech STS 3215)
- But, what are these motors? And how do we command them?
- If you look at the motor data sheet, you see lots of specs — what do they mean?
- 12 V nominal voltage (4-14V operating voltage)
- 10 kg.cm rated torque at 12 V
- 30 kg.cm peak stall torque at 12 V
- 2.7 A stall current at 12 V
- 4096 position (alternatively, 12-bit) magnetic encoder
- We will unpack all these terms

FeeTech STS3215-C018
(official product image)

https://robonine.com/testing-of-feetech-sts3215-servomotor-backlash-repeatability-and-torque/

https://core-electronics.com.au/attachments/uploads/sts3215-smart-servo-datasheet-translated.pdf

How a modern smart servo converts electrical energy to mechanical energy



This *inner* PID control loop usually runs at 60-100 Hz

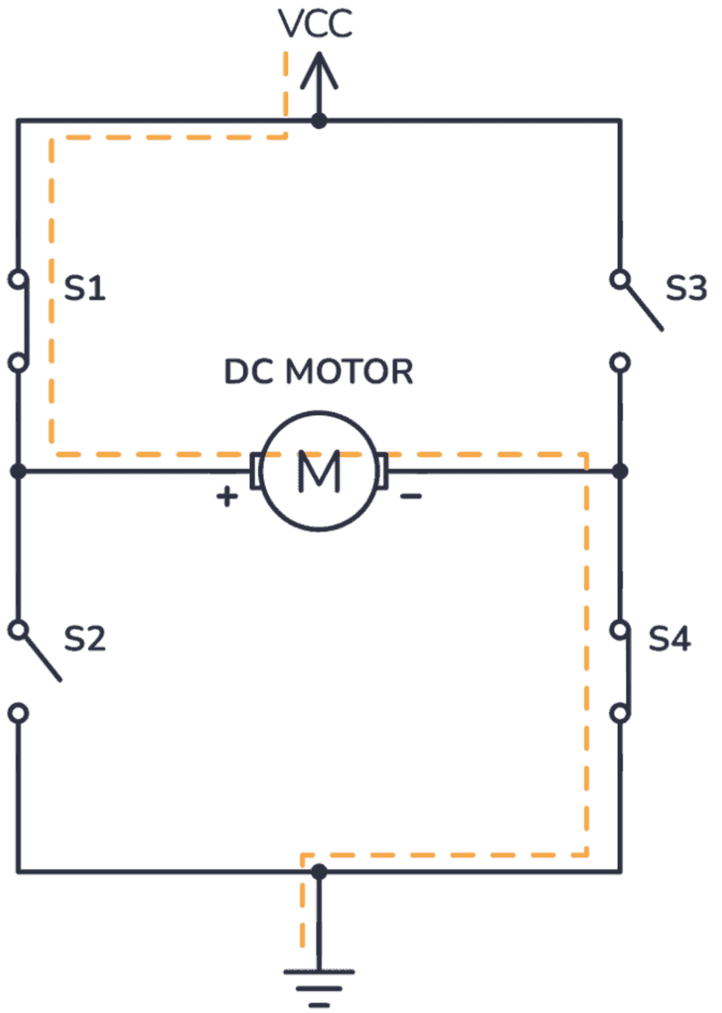
Electrical energy is (over a series of steps) converted to mechanical energy (robot movement)

Modern ‘smart servos’ include a lot more than just the servo motor. They typically contain a servo controller, comprising a position controller and a motor driver (aka H-bridge), and a gearbox

https://www.circuitbread.com/ee-faq/how-does-an-h-bridge-work

Aside: H-bridge

- Modern smart servos include one of these H-bridge circuits internally
- Allows the motor to spin both directions based on digital signals (square-pulse waveforms)
- Allows power-braking to get motors to a rapid stop
- Switch implementation matters (tiny delays needed to avoid accidental, instantaneous, short-circuiting)



https://www.circuitbread.com/ee-faq/how-does-an-h-bridge-work
One of the cleanest explanations out there on how an H-Bridge works

Servo motor specs - stall torque vs effective torque

- 30 kg . cm peak stall torque at 12 V (it's imprecise, but you'll see many data sheets say kg.cm for kgf.cm)
  - 1 kgf . cm = 0.0981 N . m => 30 kgf . cm = 2.94 N . m
- *But*, this does not mean that the motor lifts 30 kg!
- Then, how do we compute the actual mass the motor can support?
- Recall $m = \tau/(g \cdot r)$ where r is the length of the lever arm. So, for a 1 cm lever arm, a 30 kg mass is supported under stall torque (assuming all else is ideal)
- If the lever arm increases to 20 cm, mass supported at stall goes down to ~1.5 kg
- This means, the payload a robot arm can lift is dependent on it's current pose / configuration
- Common practice is to compute 'worst-case' payload (assume longest lever arm) and report that instead

Worst-case payload analysis for SO-101 follower arm

- The STS3215 motors are rated for a stall torque of 10 kgf.cm @ 7.4 V
- For the worst case, assume the arm is fully outstretched horizontally
- In this case, an object being picked up in the gripper will be (approximately) 40 cm away from the ‘shoulder’ joint.
- The rest of the arm (incl. the gripper) weighs 0.385 kg, and it’s center-of-mass is about 22.6 cm from the shoulder (in this pose)
- Torque on the motor, when grasping an object of mass m_{obj} is

$$\tau_{\text{full}} = \underbrace{0.385 \times 22.6}_{\text{arm}} + \underbrace{m_{\text{object}} \times 40}_{\text{obj}}$$

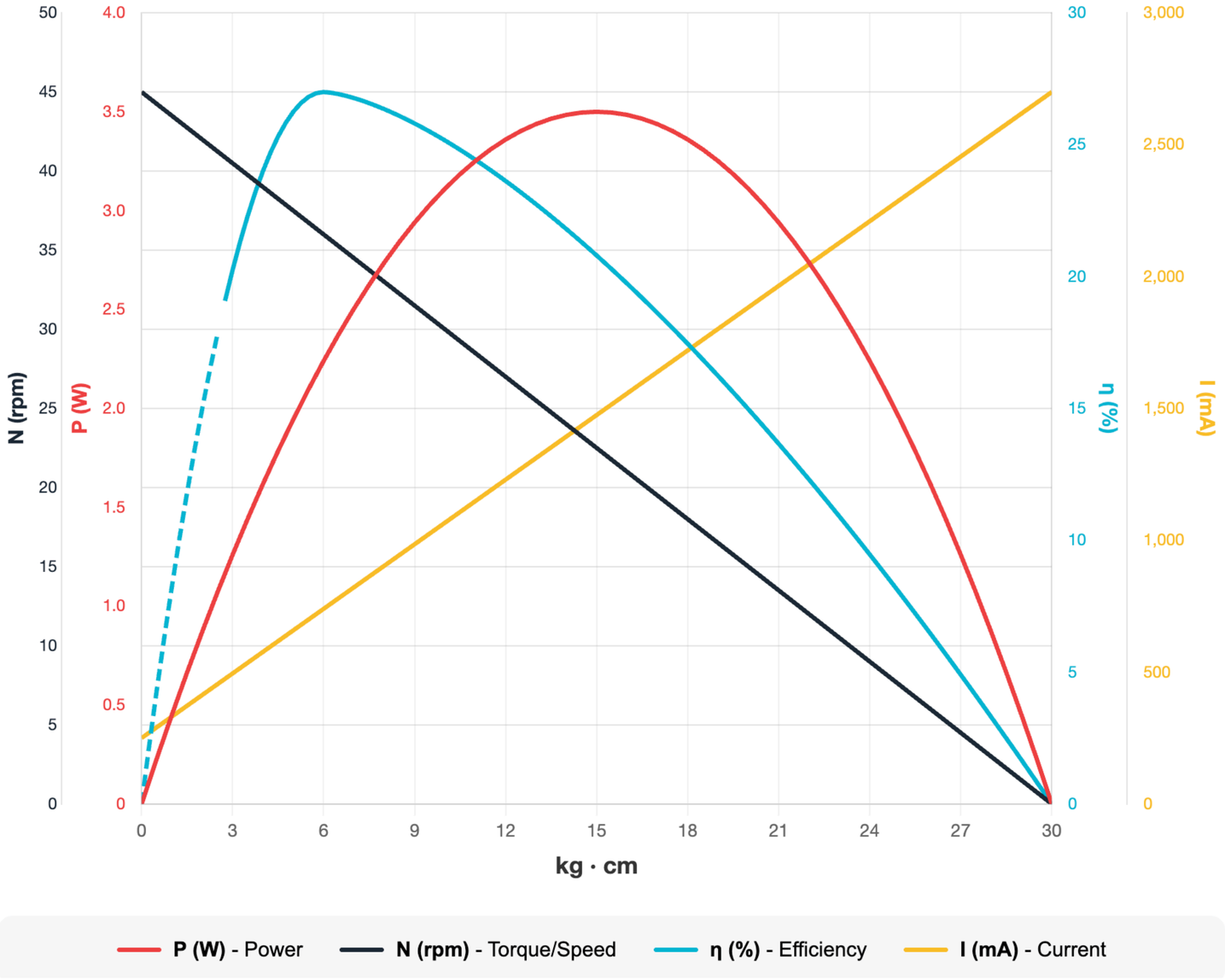
- If using 7.4 V (10 kgf.cm), this gives us an effective payload of 30 g (!!)
- If using 12 V (30 kgf.cm), this gives us an effective payload of ~500 g
- For a 15 kgf.cm rating, effective payload is ~150-160 g (more on the choice of 15 kgf.cm in the next few slides)

Instantaneous vs continuous payload

- The previous payload calculations were the theoretical maximum *instantaneous* payloads the arm could support
- However, in the real world, to support these payloads, motor draws more current
- This results in greater electrical power output (recall $P = IV$)
- For a 12 V power supply, and presuming each of the 6 motors draws it's max 2.7 A stall current, this totals a current of 16.2 A, and power of ~194 W.
- This excess power is dissipated as heat
- This presumes the power supply can provide 16.2 A current to begin with
- So, the theoretical max instantaneous payload is not what we should effectively expect in practice.
- Servo motor providers publish motor spec sheets / datasheets with torque curves, for this

How to interpret torque curves for a smart servo?

- As load / torque demand increases, current consumption increases (linearly)
- Speed decreases linearly
- Efficiency (least heat dissipation) peaks between 6 and 9 kgf.cm
- Power output peaks around 15 kgf.cm
- So, we'd ideally want worst case payloads to be around this ballpark
- In our case, this sweet spot is ~150 g



https://robonine.com/testing-of-feetech-sts3215-servomotor-backlash-repeatability-and-torque/

All the different ways in which you could render the SO-101 arm inoperable

- Flip the voltages (on one of the servos) ->
    - servos burn 🔥 🔥 -> teammates go 🫠😒😫😡
- Leader arm uses 7.4V while follower arm uses 12V (brilliant move)
    - Interchange leader and follower power adapters -> servos burn 🔥 🔥 -> robot dies -> teammates unhappy
- Hold one of the robot's powered links firm to prevent it from moving
    - Motor draws more current to prevent stalling -> eventually heats up enough to burn -> robot dies -> teammates complain to instructor
- Lift a heavy object (> 150 g) -> matter of time before motor overheats -> robot dies
- Many other ways: incorrect servo wire connections, leaving robot idling and on for too long, not powering down the robot before unplugging/replugging motors, (and many, many more)

What changes from a hobby robot to a serious robot?

	SO-101	**Franka / UR robot**
Encoder to sense motor position	12-bit encoder (0.088° / count)	Much higher (14-19 bits or more)
Backlash	~0.5°-0.9°	Near-zero
Low-level control targets	Position targets	Torque targets
Repeatability	O(1mm)	sub-mm (0.03-0.1 mm)
Control frequency	O(10 Hz)	1 KHz
Safety features for motors	Temperature / overheat sensing, overload sensing	Force/torque limits, velocity limits, and more
Compliant control	Only position-based control supported	Active impedance control, force control

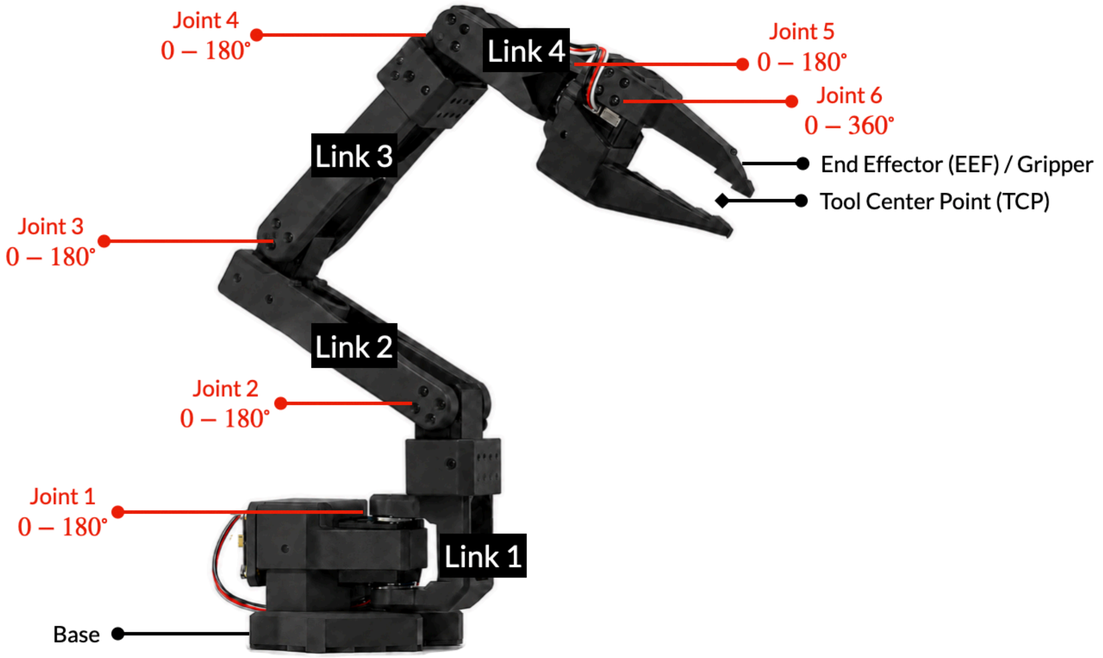
Subsection recap

- Robot -> links + actuators (joints)
- For hobby robots, links can simply be 3D printed (that’s what we do in this class)
- Smart servos and how they convert electrical energy into mechanical energy
- How to estimate the payload a design can support
- We looked at the multiple ways in which motors and robots get damaged

How to mathematically model a robot arm

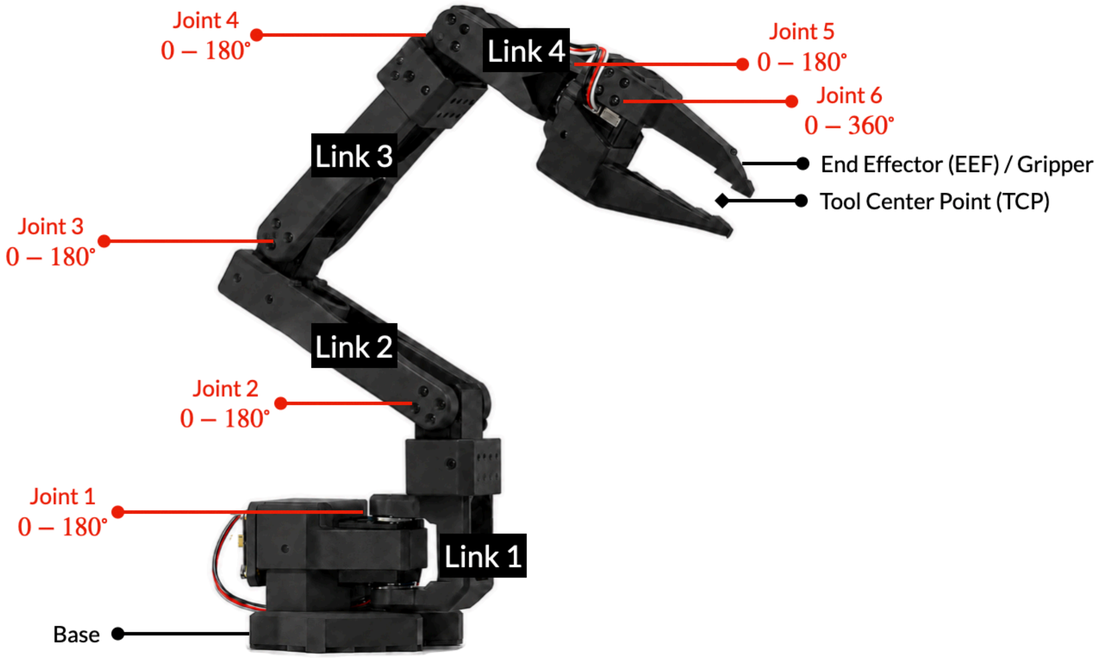
Joint space

- Joint coordinate: angle denoting where the (revolute) joint is at (e.g., for link 1, joint coordinate would be a value in 0-180)
- In this course, we will use q to denote joint coordinates
- Joint configuration: values of all of the robot's joint coordinates $q = [q_1, q_2, \cdot, q_n]$
- For SO-101, joint configuration specified 5 angles for the arm joints, plus one for the gripper
- Joint configuration uniquely specifies where each joint (and by extension, link) of the robot is, relative to the base.



Cartesian space, Task space, Degrees of Freedom

- Cartesian position: typically 3D coordinates (x, y, z) relative to the base
- This is used to express where the TCP is
- Task space: quantities we want to control — e.g., just tool position, or tool pose, etc.
- Degrees of Freedom (DoF): number of independent configuration coordinates (e.g., rigid body pose has 6 independent coordinates — 3 for translation, 3 for rotation)
- SO-101 has 6 motors. Does it mean it has 6 DoF for it's end-effector?

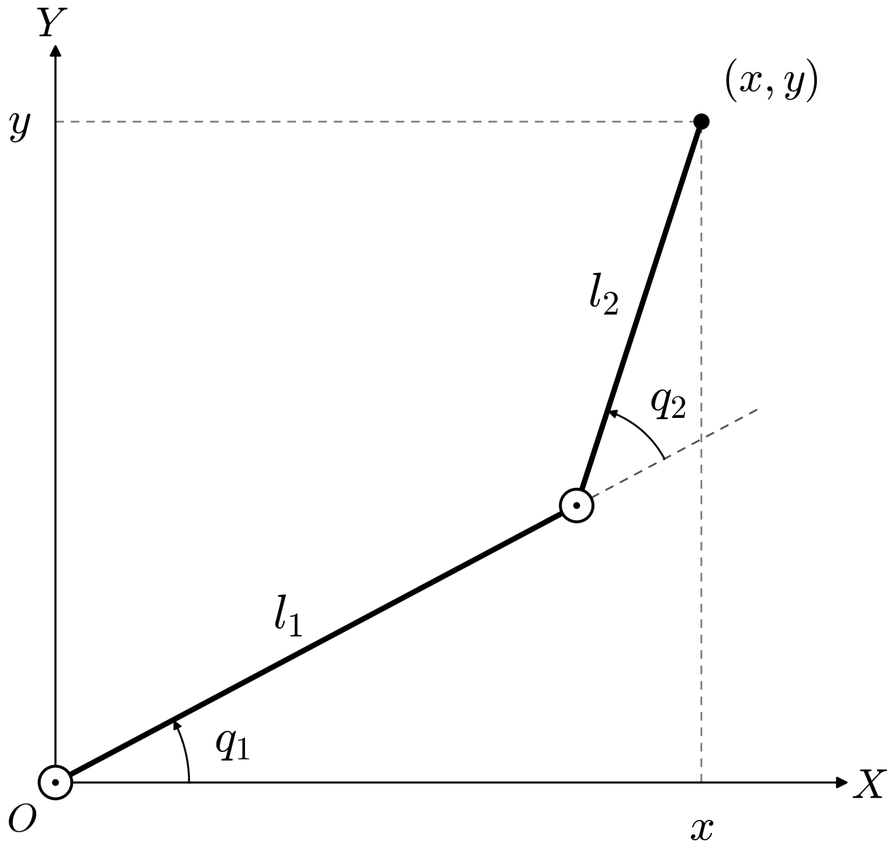


Forward kinematics (FK)

- Given a joint configuration, compute the cartesian pose (e.g., TCP pose)
- Depends on robot geometry (link dimensions, joint locations, axes of revolution, …)

$$\mathbf{x} = f(\mathbf{q})$$

- E.g., given joint encoders (present by default in each servo), calculate where gripper is
- For serial manipulators, FK is a composition of multiple transforms (one per joint)



$$x = l_1 \cos(q_1) + l_2 \cos(q_1 + q_2)$$
$$y = l_1 \sin(q_1) + l_2 \sin(q_1 + q_2)$$

Inverse kinematics (IK)

- Given a desired tool pose, find joint configuration(s) that achieve(s) it

$$\text{Find } \mathbf{q} \text{ such that } f(\mathbf{q}) = \mathbf{x}_{\text{desired}}$$

- Inherently ill-posed / ‘inverse’ problem
- IK may have no solution (target unreachable), several discrete solutions (distinct/unique joint configs), or a continuous family of solutions (redundant arms), depending on the specific robot and configuration
- Analytical IK: geometry/algebra to obtain explicit solutions
- Numerical IK: start with a joint configuration (i.e., guess a solution) and iteratively update it to reduce target error $\|f(\mathbf{q}) - \mathbf{x}_{\text{desired}}\|$
- To solve this optimization problem using gradient-based methods, we need the derivative of the FK function f — this derivative is the Jacobian

The Jacobian

- The Jacobian is the matrix of partial (first-order) derivatives of the forward kinematics function
- It describes how small changes in joint positions produces changes in the TCP pose

$$\dot{\mathbf{x}} = J(\mathbf{q})(\dot{\mathbf{q}}) \qquad J(\mathbf{q}) = \frac{\partial f}{\partial \mathbf{q}} \qquad J_{ij}(\mathbf{q}) = \frac{\partial x_i}{\partial q_j}$$

- For the planar 2-revolute-link (a.k.a 2R) arm we saw a couple slides ago,

$$J(q_1, q_2) = \begin{bmatrix} -l_1 \sin(q_1) - l_2 \sin(q_1 + q_2) & -l_2 \sin(q_1 + q_2) \\ l_1 \cos(q_1) + l_2 \cos(q_1 + q_2) & l_2 \cos(q_1 + q_2) \end{bmatrix}$$

- E.g., col 1 tells us the TCP / tip velocity when joint 1 moves at unit angular velocity (while holding joint 2 fixed)
- To get the actual TCP / tip velocity, sum the contribution due to each joint:

$$\dot{\mathbf{x}} = \underbrace{J_{:1}\dot{q}_1}_{\text{Joint 1 contrib}} + \underbrace{J_{:2}\dot{q}_2}_{\text{Joint 2 contrib}}$$

What's the Jacobian useful for?

- Numerical IK (gradient based methods need to recompute the Jacobian at each solver iteration)
  - Jacobian is used in a gradient-based update
  - However, at many joint configurations, the Jacobian is singular (rank-deficient) in one or more dimensions. This is known as a **kinematic singularity**, and is bad for your robot (IK may require large joint motion or velocity updates; may lose some instantaneous TCP motions)
  - E.g., in the 2R example, if the arm is stretched out so that both joints align, the joints no longer provide two independent TCP motion directions
- Task-space velocity control
  - If we rearrange the expression relating joint and Cartesian velocities, we get $\dot{\mathbf{q}} = J^{\dagger}(\mathbf{q})\dot{\mathbf{x}}$
- Characterizing the range of motion available at a particular joint configuration
  - Columns of Jacobian tell us the TCP velocities each joint can generate
  - The rank of the Jacobian tells us how many of these TCP motion directions are independent

Jacobians in the context of the SO-101 arm

- Recall: the SO-101 arm has 6 servo motors in all (5 for the arm, 1 for the gripper)
- This results in 5 degrees of freedom for the TCP, in the Cartesian space
- The Jacobian, in this case, is a 6 x 5 matrix (since the Cartesian velocities include 3 linear and 3 angular velocity components, and only 5 joints need to work to achieve them)
- What about the gripper?
- Gripper only opens and closes — it doesn't move the robot arm otherwise
- SO-101 Jacobian is rank deficient. This means it can't reach any arbitrary pose in its workspace all the time (needs to compromise on one of the dimensions in the pose)

Other concepts you should brush up on (we won't recap)

- Coordinate frames and change of coordinate frames
- Rotation representations: rotation matrices, Euler angles, quaternions
- Homogeneous transforms, transform composition, inversion

Break

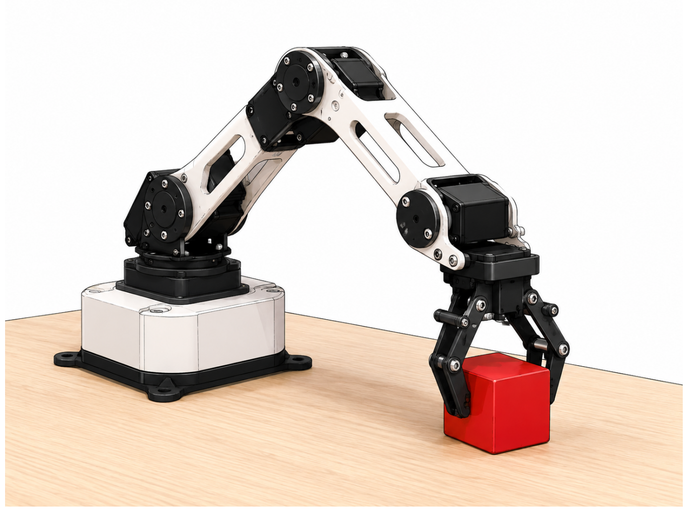
Reminder: fill out the background and teaming survey on Canvas

How do we get a robot arm to move to where we command it?

Quick recap of (position-based) control

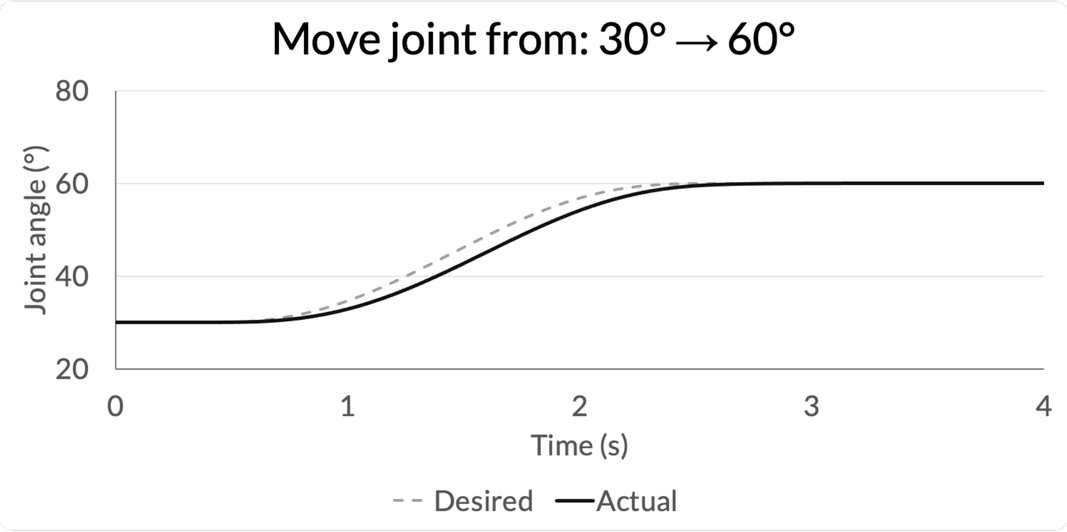
Scenario: Get the robot to pick up the cube

- Assume: we know the coordinates of the center of the cube w.r.t. the robot's base frame
- Reasonable first attempt / idea: Let's get the TCP / tool to move to that known cube-center; and then close the gripper
- But the center is known in Cartesian space — how do we get it to be in joint space? *Inverse Kinematics*
- This gives us *where the robot joints need to be*
- Now, *how do we move the robot joints from its current to the target positions*?
- Position-based control!



Position-based control

- Given a target joint position (i.e., angle the servo shaft needs to be in), actually compute the underlying signals (pulse-widths) that get the joint there
- Constraints: joint limits, velocity (and/or acceleration) limits
- A reference trajectory $q_d(t)$ specifies joint positions over time
- Controller attempts to follow (a.k.a. *track*) this reference trajectory over time
- Absolute position control (e.g., “got to 60 degrees”) is also possible (in fact our STS3215 servos support that)



Desired trajectories don’t always fully follow the actual trajectories

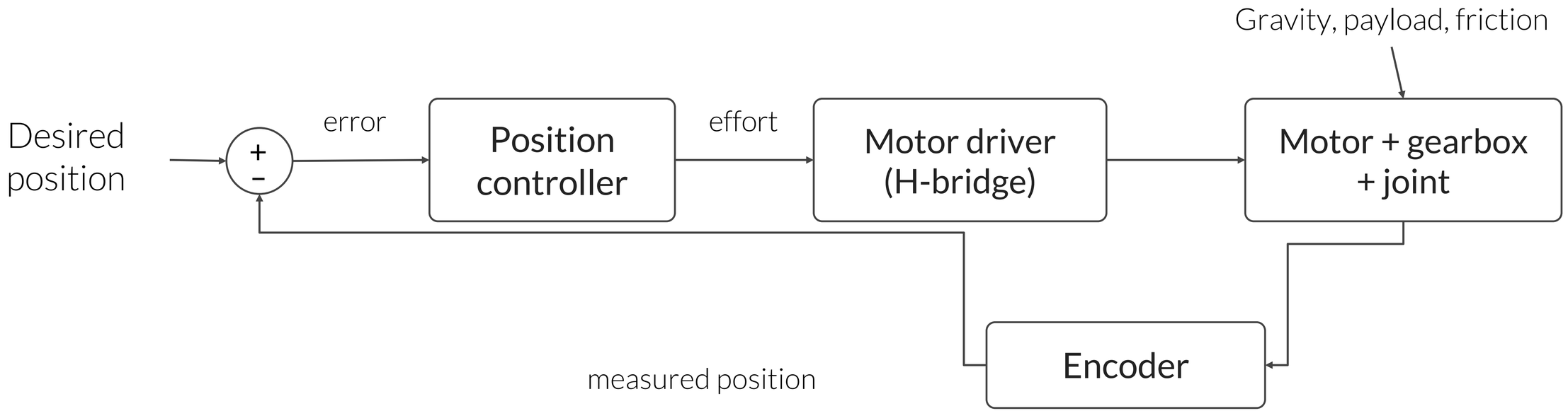
$q_d(t)$: desired

q(t): actual

Position feedback control

$$e(t) = q_d(t) - q_{measured}(t)$$

60° − 45° = 15° of position error



- Measure position (from servo encoder) → compute error → adjust motor effort → repeat
- Note: motor effort also depends on payload (not just position) heavier payload → more effort to hold position

PID position control

$$u(t) = K_P e(t) + K_I \int_0^t e(s)\,ds + K_D \dot{e}(t)$$

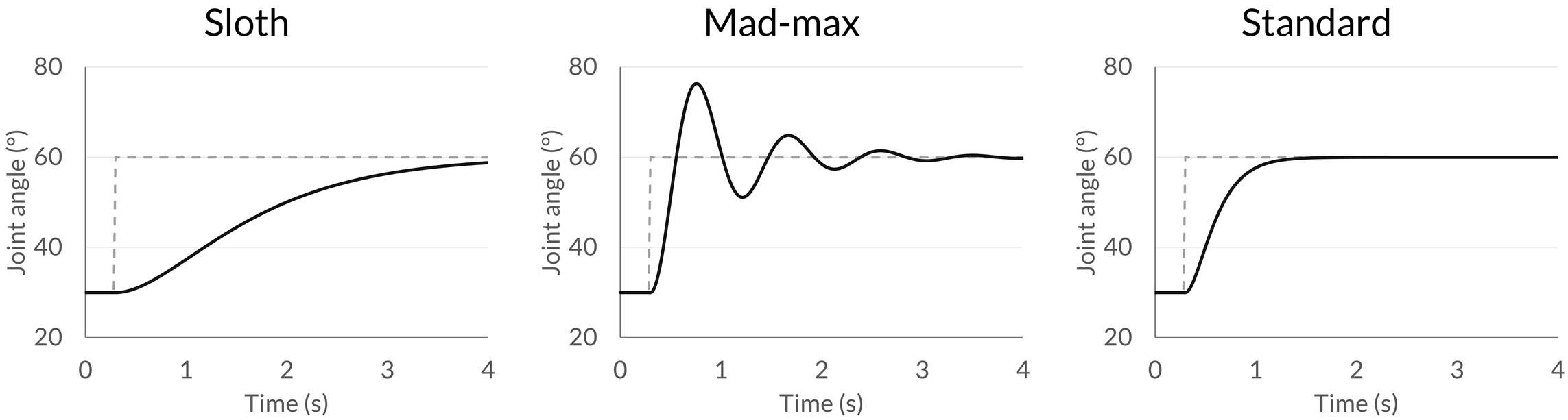
u(t): internal motor effort; the computer can still send position targets.

	What we measure	**What it means**
P — proportional	Current error	Bigger error → stronger correction
I — integral	Accumulated error	Corrects a persistent offset under load
D — derivative	How error is changing	Adds damping; reduces overshoot

Often, controllers the SO-101 (and other) commodity robot arms use are often just PD (no I)

What does a good response look like?

Controller gain tuning – fortunately, you don't have to do it for this class 🎉🎉



Gray: target Black: actual

When tuning a PID controller, monitor speed, overshoot, settling time, and steady-state error.

Which response would you want while carrying a cup of coffee?

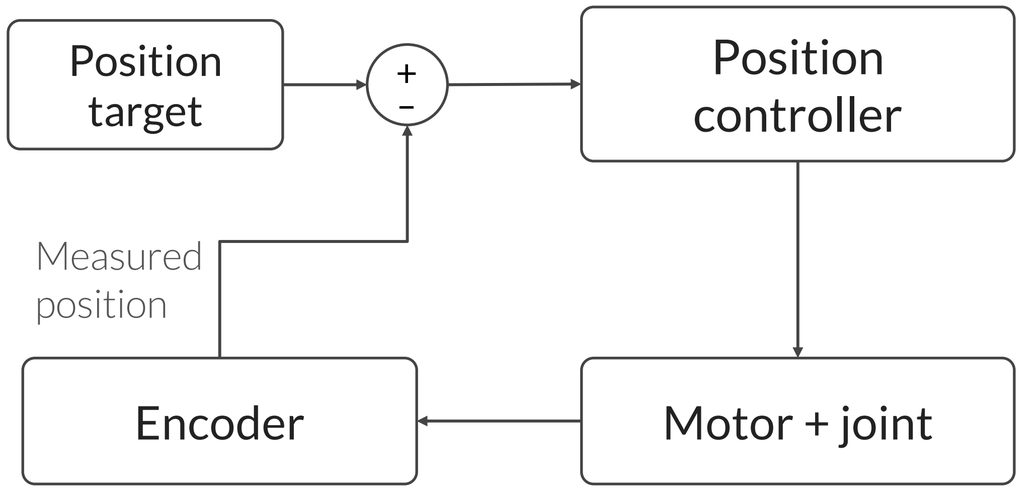
(Not to imply that FSD modes on a Tesla are literally tweaking PID coefficients)

Position control in the SO-101 context

Outside the servo (i.e., on your computer)

- Specify joint targets (IK, teleoperator, learned policy, …)
- Send position commands over the servo bus (3-wire cable)
- Read joint positions (and cameras / other sensors)

Inside each STS3215 smart servo



Goal_Position: write target

Present_Position: read encoder

Rate at which you send commands from your computer is usually *much* slower than the internal servo control rate

What pure position-based feedback control can-vs-can't do

- PID / PD gains tuned for free-space don't always work across tasks (e.g., heavier payload => more holding force => PD controller over/under shoots)
- This is why feedback is needed to begin with
- Open-loop: execute commands w/o measuring position / error (e.g., replay a previous trajectory)
- Closed-loop: commands take measured position / error into account
- Intuition check:
  - What if payload increases, and is beyond motor capacity?
  - What if calibration is incorrect (i.e., the motor shaft is off by 2 degrees)?
  - What if the cube starts moving?

Other control modes to be aware of

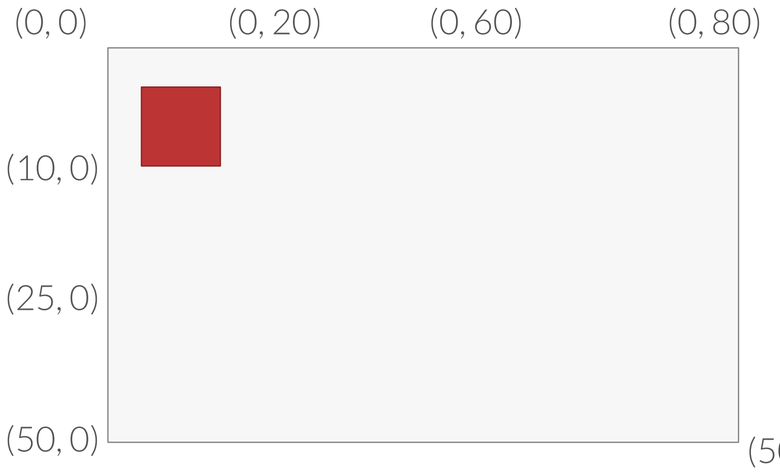
- Joint velocity control — works in the velocity space (i.e., how fast should each of the joints be rotating)
- Cartesian / task-space position control — similar to joint position control in how feedback is computed, but everything's done in Cartesian coordinates
  - This means we constantly need to compute joint targets (via IK or Jacobian pseudo-inversion)
  - Quite useful in teleoperation of robot manipulators
- Torque control / current control
  - Enables compliant / force control (e.g., Franka, UR robots)
  - Can control 'how stiff' each joint should behave like
- In this course, we'll only use position control — and for the most part, we won't even touch the controller code

Modern machine / deep learning: A quick recap

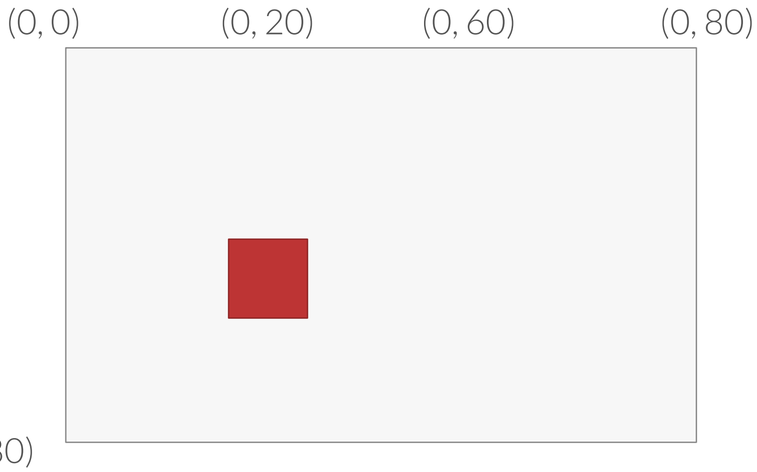
Note: some of the slides in this sub-section have illustrations generated using GPT 6 Astra

Supervised learning

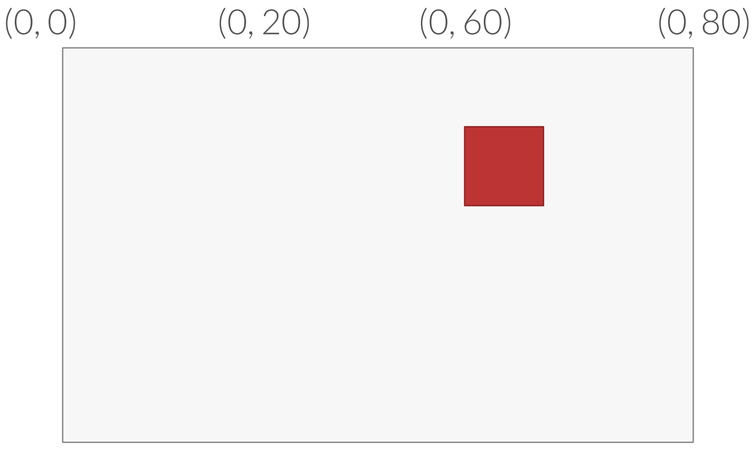
Learn a mapping from labeled examples.



y = (5, 10)



y = (25, 18)



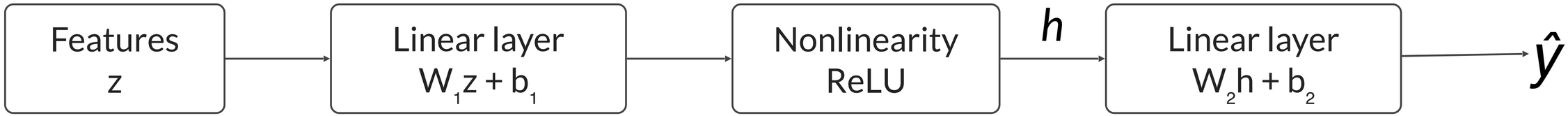
y = (10, 60)

Learned mapping is $\mathbf{y} = \mathbf{f}_{\theta}(\mathbf{I})$ I: input image y = (u, v): labeled cube position

θ: the model parameters we learn from the examples

A neural network is a parameterized function

- The function computes an output quantity by composing multiple computations
- Computations are organized into *layers* (layers compose simple operations, such as affine transformation, nonlinearities, etc.)
- Parameters θ are the weights W and biases b.

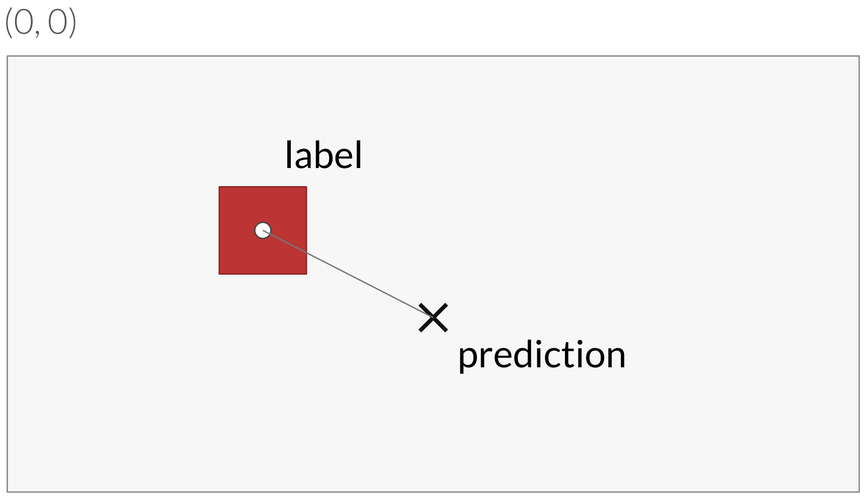


ReLU(z) = max(0, z), applied to each feature

- Nonlinearities let the network represent more complex function maps (w/o a nonlinearity, the entire sequence of operations collapses into a 'linear' map)
- These computations are generally thought of as 'graphs' (computation graphs)

Loss functions

Loss functions compare predictions (typically from a neural network) with the true labels aka (‘ground truth’) and provide a scalar value (lower is better)



Label: y = (0.3, 0.4)

Prediction: $\hat{y}$ = (0.5, 0.6)

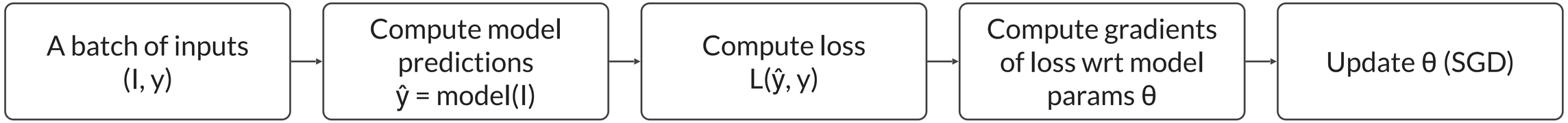
$$\ell = \|\hat{y} - y\|^2$$

$$= 0.2^2 + 0.2^2 = 0.08$$

Loss is typically computed over a ‘batch’ of inputs

	Output	**Common loss function(s)**
Regression	Continuous values, e.g. (u, v)	Squared error
Classification	Scores for discrete classes	Cross-entropy

How do we *train* these neural networks?



(Pseudo) code

```
optimizer.zero_grad()
prediction = model(images)
loss = loss_fn(prediction, labels)
loss.backward()
optimizer.step()
```

- Batch: the examples used for one update.
- Epoch: one pass through the training dataset.

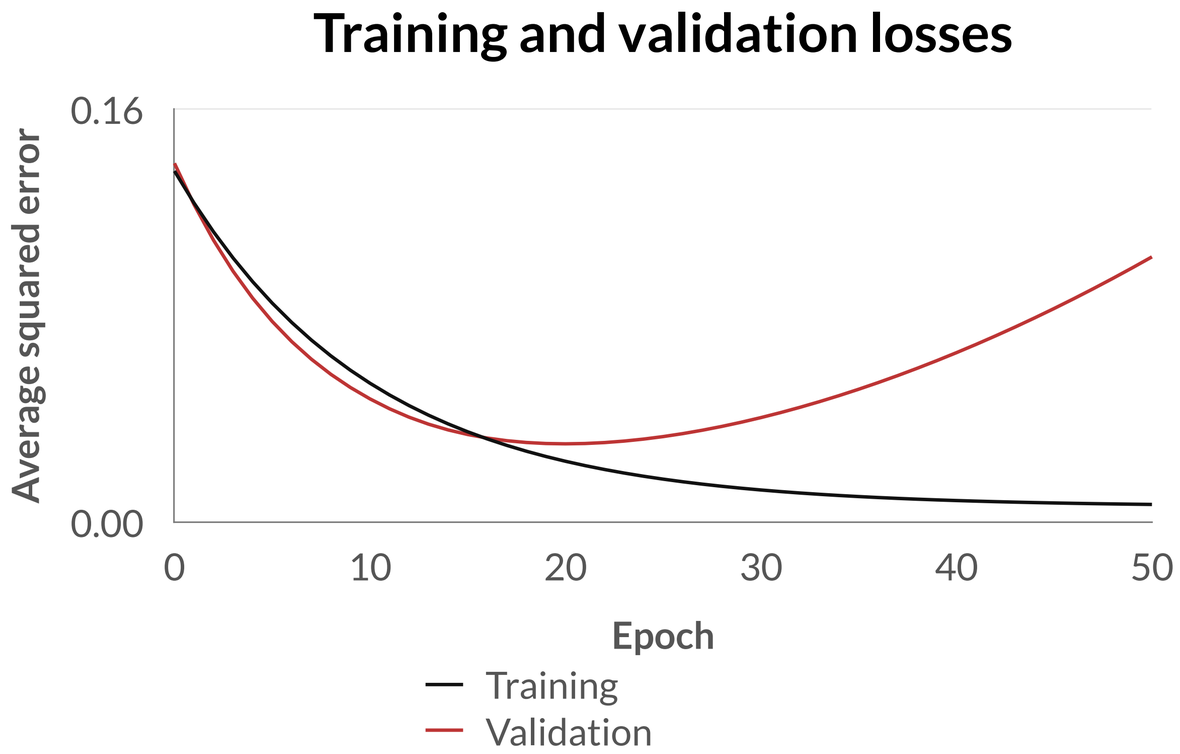
Gradients are computed using modern automatic differentiation libraries

The *optimizer* updates parameters.

$$\theta \leftarrow \theta - \eta \nabla L(\theta)$$

Stochastic Gradient Descent(SGD) update
η denotes the *learning rate*
Higher learning rate causes faster updates

Training vs Validation



Training set

Fit the model parameters.

Validation set

Choose settings and checkpoints.

Test set

Evaluate the selected model on held-out data.

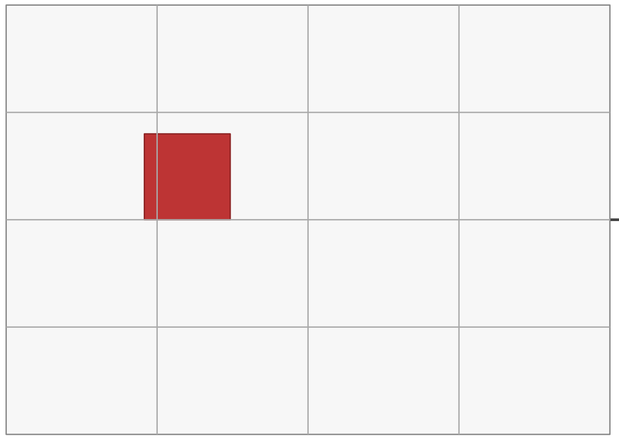
What could be going on here?

Overfitting

Modern deep learning: *Tokens*

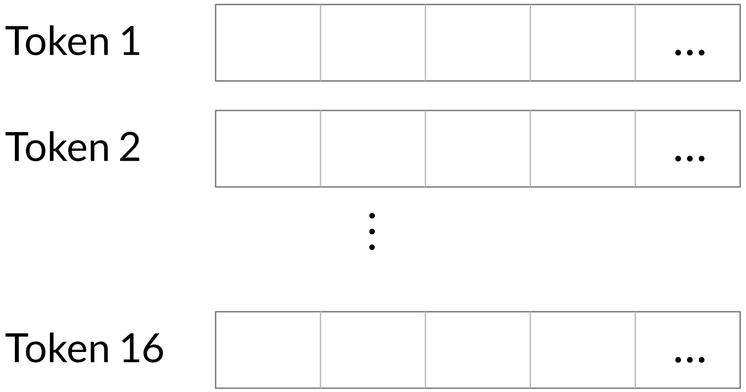
A token is represented by a vector of features.

E.g., 4 × 4 image patches



Flatten each patch
+ learned projection

One vector per patch



16 tokens × d features

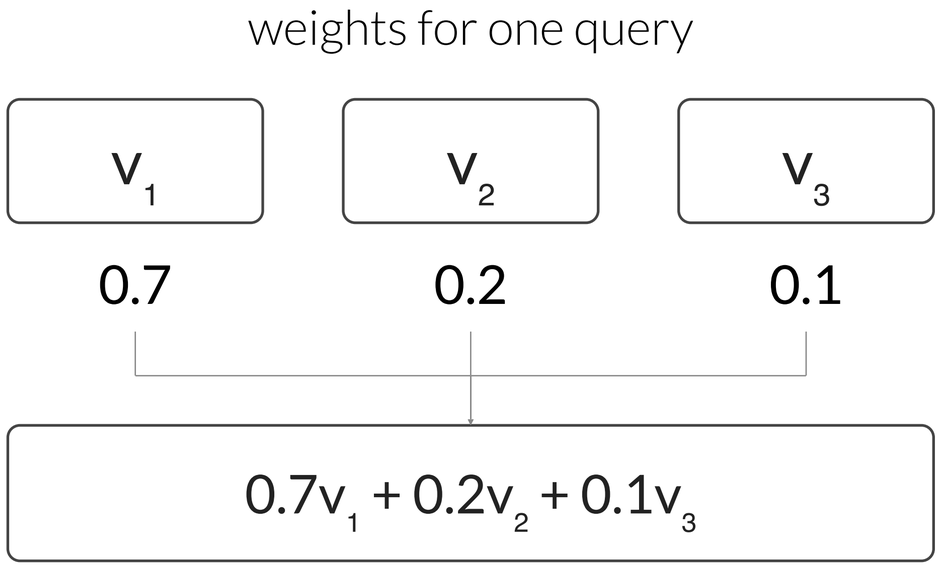
- Add position information so the model knows where each patch came from.
- Text tokens and joint-state vectors can also become token representations.

Attention mixes information across tokens

Each token combines information from other tokens, with input-dependent weights.

Query–key similarities → softmax weights → weighted sum of values

	Role
Query **q**	What information would help?
Key **k**	Describes this token for matching
Value **v**	What information gets combined?

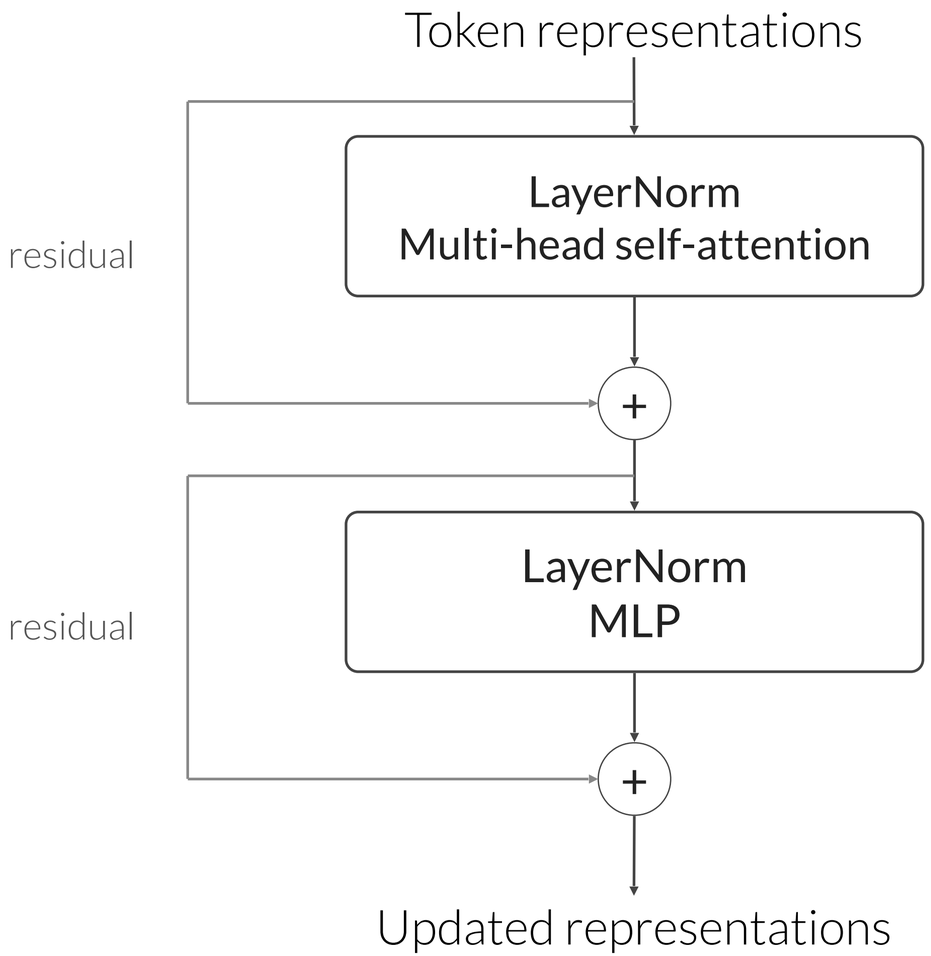


Queries, keys, and values come from learned linear projections.

Attention weights can change while the trained model parameters stay fixed.

Inside a transformer block

One common arrangement (pre-norm)



Multi-head attention

Several attention computations exchange information across tokens.

MLP

Transforms each token's features using shared weights.

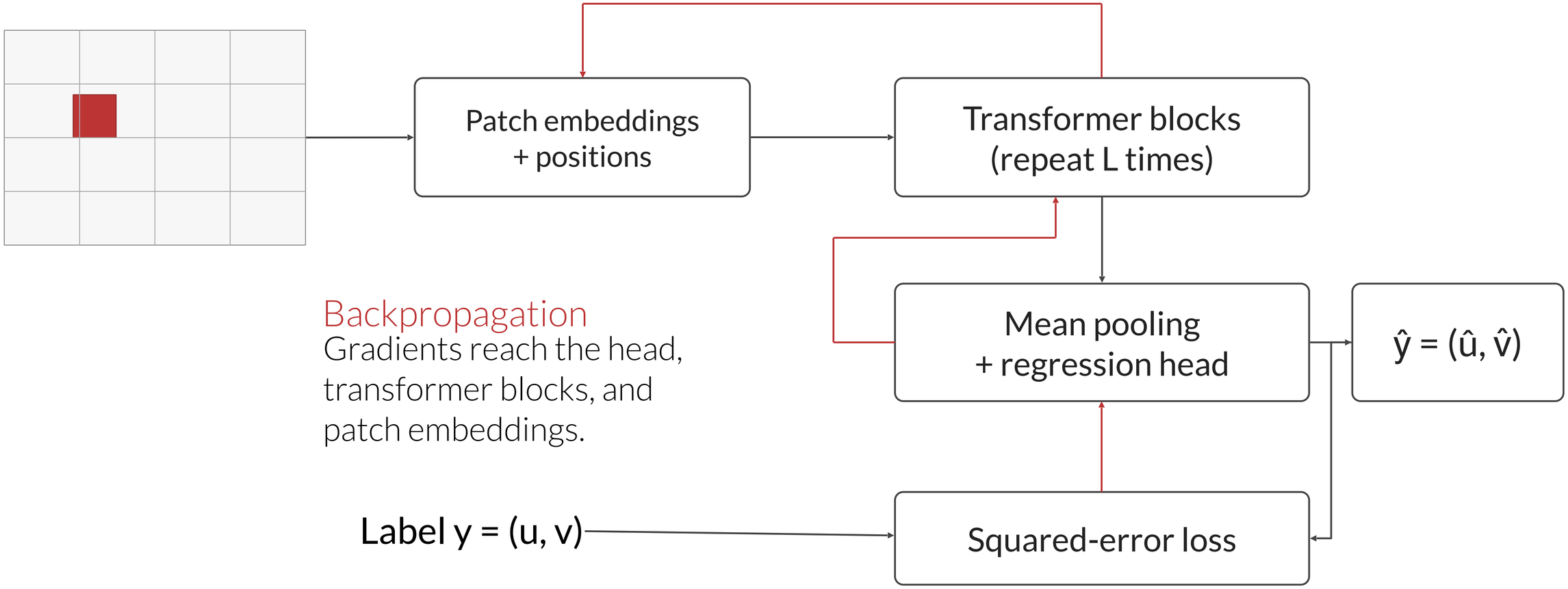
Residual connections

Add the incoming representation back.

LayerNorm

Normalizes features within each token.

Train a transformer on a supervised learning problem



Backpropagation
Gradients reach the head, transformer blocks, and patch embeddings.

The labels supervise the output. Each attention *learns for itself* what it should attend to.

Training, fine-tuning, and inference

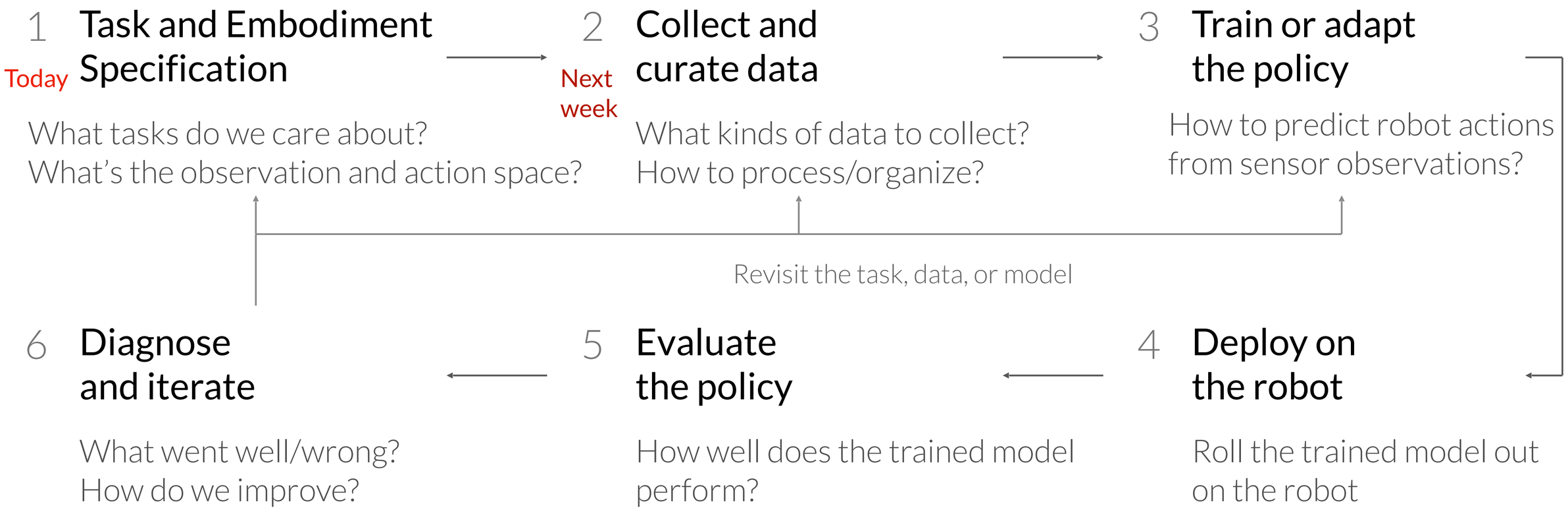
Mode	Starting point	What does this mean
Training from scratch	Randomly initialized parameters	All parameters are being updated
Fine-tuning	A pretrained model	A subset of parameters are updated (others are fixed)
Inference	A pretrained model	No parameters are updated
Low-rank adaptation	A pretrained model	A *very small subset* of parameters are updated (model performance on older tasks is preserved)

Attention weights changed for a new image. Does this mean the model learned something new?

If all this is new to you, please brush up on: Neural networks, backpropagation, stochastic gradient descent, optimizer variants, positional encodings, encoder/decoder architectures, causal masking, etc.

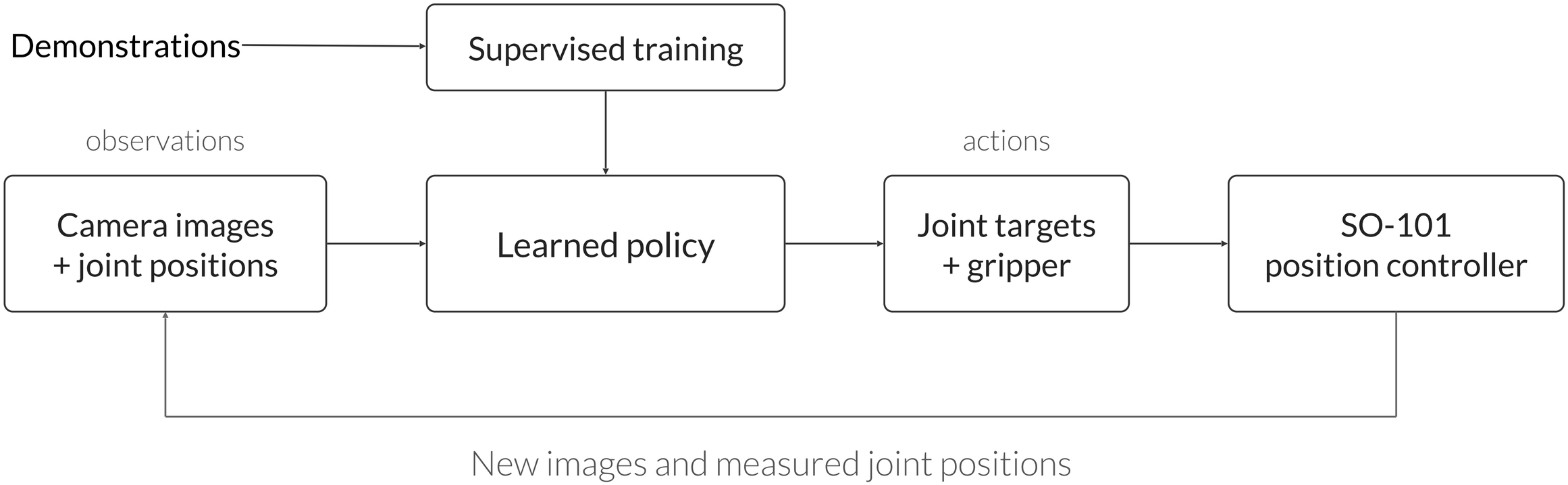
The *modern* robot learning workflow

The *modern* robot learning lifecycle



Putting it all together

In this course, we will train learned models/policies that compute *position* commands for robots



Important takeaway/note: Policies don't live in a *void*. They're always viewed through the lens of a low-level *controller*. (i.e., embrace the *full-stack* view of robot learning)

In the rest of this course, we will build

	How are actions computed	**Focus / learning objective**
Handcrafted policy	You'll design code that determines how a robot responds to observations	Traditional Sense-Plan-Act approach
Single-task imitation	A neural network *policy* that is trained to mimic teleoperated demos	Data collection, training, and rollout evaluation
VLA/WAM/multi-task policies	A generalist policy trained on multiple tasks	Generalization

Policy training workflow

Questions?