A WORKING GUIDE FOR ENGINEERS

Deep Learning and Robotics

The ideas behind today's models and the robots they drive

Matthew Wnuk · www.matthewwnuk.com/guide.html

32 topics in five groups

From the training loop to the robot's hands, in the order the ideas build on one another.

Real measurements

Most charts come from real robot data or from experiments run for the guide. Illustrative ones are labelled.

Meant to be presented

Each slide has one message, a picture, and speaker notes. 27 notebooks reproduce the measurements.

OPENING

The map: 32 topics in five groups

from how a network learns, to the models, to robot policies, to bodies and data, to how the work is done

Foundations

- 1 Things I know
- 2 Foundations that survived
- 3 Sequence-to-sequence models
- 4 The transformer
- 5 Attention in depth

Models

- 6 Representation learning
- 7 Contrastive and multimodal learning
- 8 Retrieval and multimodal retrieval
- 9 Generative modeling
- 10 Stable Diffusion and latent diffusion
- 11 Foundation models and adaptation
- 12 Vision-language models
- 13 Perception and 3D

Robot learning

- 14 Robotics fundamentals
- 15 Robot learning basics
- 16 Reinforcement learning
- 17 Robot policies today
- 18 System 0, 1 and 2 architectures
- 19 Long-horizon tasks
- 20 World models

Data and hardware

- 21 Embodiments
- 22 Senses beyond vision
- 23 Dexterous hands
- 24 Data for robots
- 25 Data quality and curation
- 26 Data efficiency and generalisation
- 27 The data flywheel

Practice

- 28 Training at scale
- 29 Evaluation
- 30 Failure triage
- 31 Deployment on the robot
- 32 Understanding and safety

Each group builds on the one before it; the numbers are the section numbers in this deck

OPENING

How to use the guide

read it in order, or start from what you already know

You know machine learning

- Start with the robot notes: sections 14 to 20.
- Then bodies, senses and data: 21 to 27.
- Then practice: 29 to 32.

You know robots

- Start with how networks learn: sections 1 to 5.
- Then the model families: 6 to 12.
- Then robot policies (17) and training at scale (28).

Run it

- 27 notebooks at github.com/wnukmat/robot-learning-notebooks, each tied to a section of a note.
- Data and small checkpoints are on Hugging Face under mwnuk.
- Section title slides name the notebooks that go with them.

The deck gives the result; the note gives the derivation; the notebook lets you change it and rerun

FOUNDATIONS · TOPIC 1 OF 32

Things I know

Six habits that outlast any model, and how silent failures are caught.

KEY POINTS

- Data, scale and honest evaluation decide more than architecture does.
- Know how much a number moves when nothing changes before reading a difference into it.
- A falling loss proves the code runs, not that the model learned what you intended.

Run it: How much of a result is noise (github.com/wnukmat/robot-learning-notebooks)

Six principles that outlast any model

model names change every few months; these habits decide whether a project works more often than the architecture does

Data beats clever algorithms

- The scale, variety and quality of the data set the ceiling.
- A better dataset usually helps more than a new architecture.

General methods that scale win

- Sutton's “bitter lesson”: methods that improve with data and compute keep overtaking hand-encoded knowledge.
- Prefer designs that get better as the data grows.

All models are wrong

- A good score on held-out data is not a working robot.
- Judge a model in the loop it will actually run in.

Failing models don't throw exceptions

- A broken pipeline still produces a loss curve.
- The only way to know is to look at the inputs, labels and outputs yourself.

Evaluation leaks

- Every choice made against the test set moves a little of it into the model.
- Public benchmarks may already be in the pretraining data.

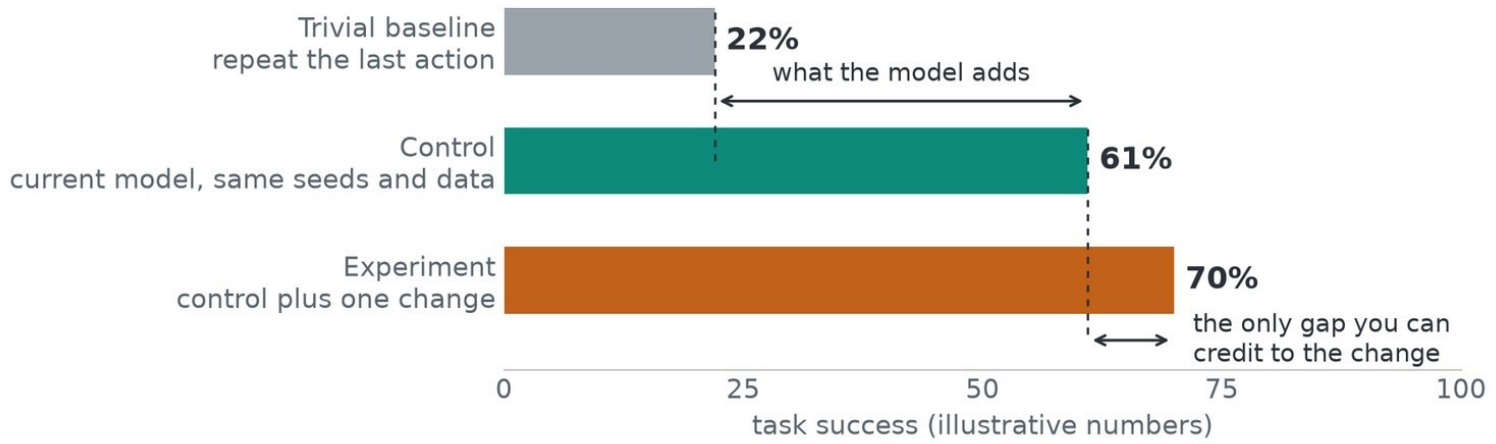
One variable per experiment

- Change one thing and keep a matched control.
- Know what a trivial baseline scores before you start.

Data, scale and honest evaluation decide more than the choice of architecture

Three numbers make a result readable

the last principle as a picture: a trivial baseline, a matched control and the experiment



Trivial baseline, 22%. What the simplest method scores. It shows how much of the result needed a model at all.

Control, 61%. A run identical to the experiment in every respect except the one thing being tested.

Experiment, 70%. It differs from the control in one way, so the marked gap belongs to that change.

Before the run. Know what the trivial baseline scores before you start, so the other two numbers have a floor.

An improvement can be attributed only when the control differs from the experiment in one way

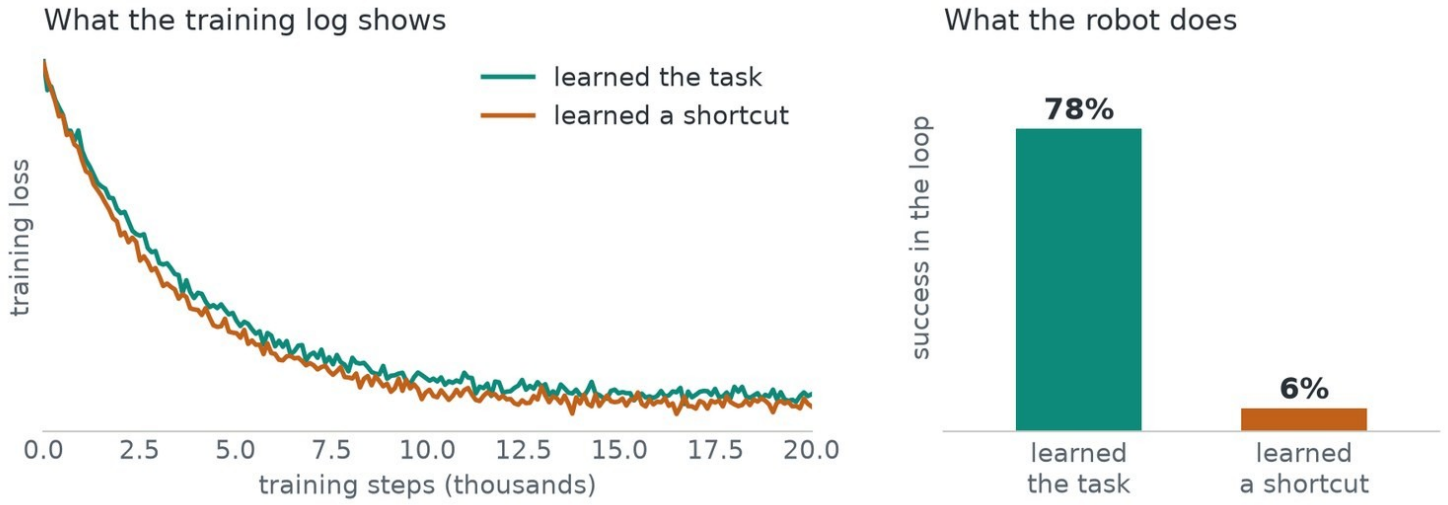
Silent failure: the logs look the same

an illustrative pair of runs: one learned the task, one learned a shortcut

Left: two loss curves. Both training losses fall almost identically. Nothing crashes and nothing in the log separates the runs.

Right: success on the robot. The model that learned the task succeeds 78% of the time. The one that learned a shortcut succeeds 6%.

What it means. The loss went down in both, and one model still learned the wrong thing.



Deep learning code fails quietly, and only running the policy tells the two runs apart

Four failures account for most silent cases

what went wrong in each, and the symptom it leaves in the numbers

Failure	What happened	Symptom
Label or input bug	Images and labels are misaligned, normalisation is wrong, or two cameras are swapped	The loss falls slowly and levels off high
Data leak	Test samples, or near-copies of them, are in the training set	The validation score looks too good
Shortcut learning	The model uses a cue that happens to correlate with the answer	Great on the benchmark, fails in new scenes
Evaluation bug	The metric is computed on the wrong split or the wrong output	The numbers do not match what you see

Each failure has a recognisable symptom; shortcut learning is the hardest of the four to spot

The copy shortcut, on real data

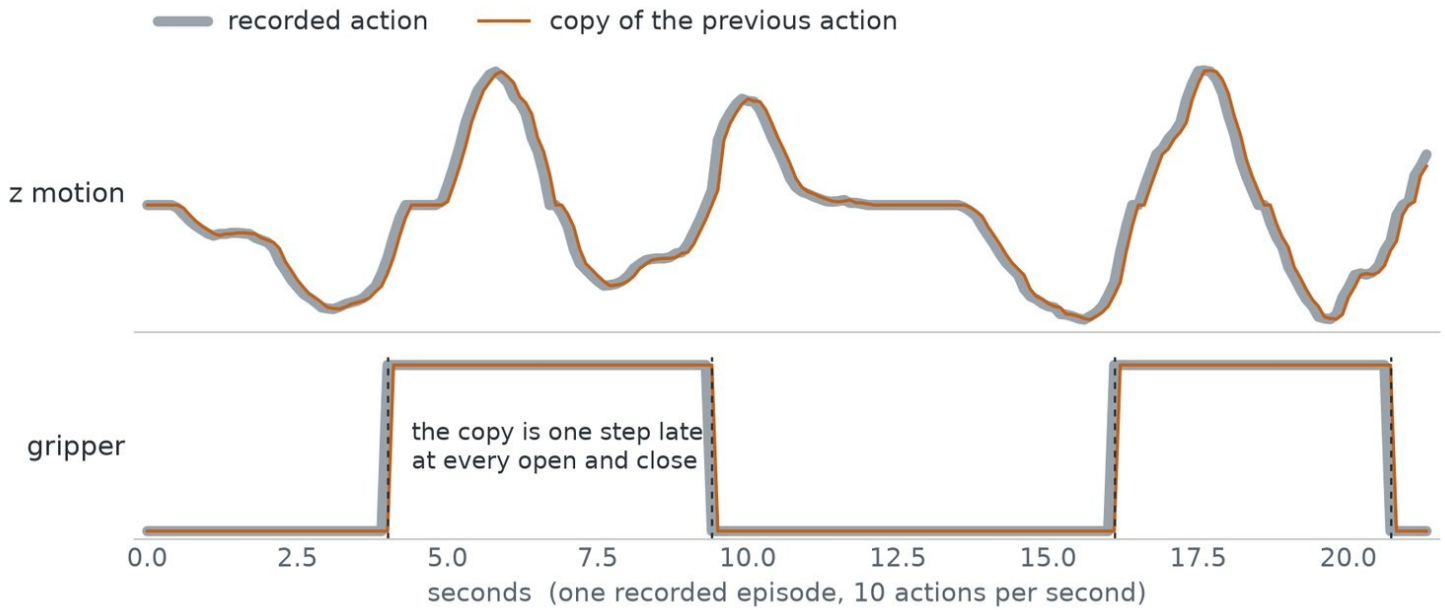
z-motion and gripper actions from one LIBERO demonstration of 214 steps, with a copy of the previous action on top

Nearly indistinguishable. The copy of the previous action lies almost exactly on the recorded action.

13 times lower error. Repeating the previous action gives a mean absolute error of 0.020. Predicting the average action gives 0.253.

Still cannot do the task. The copy never starts a motion and is late at all four gripper changes.

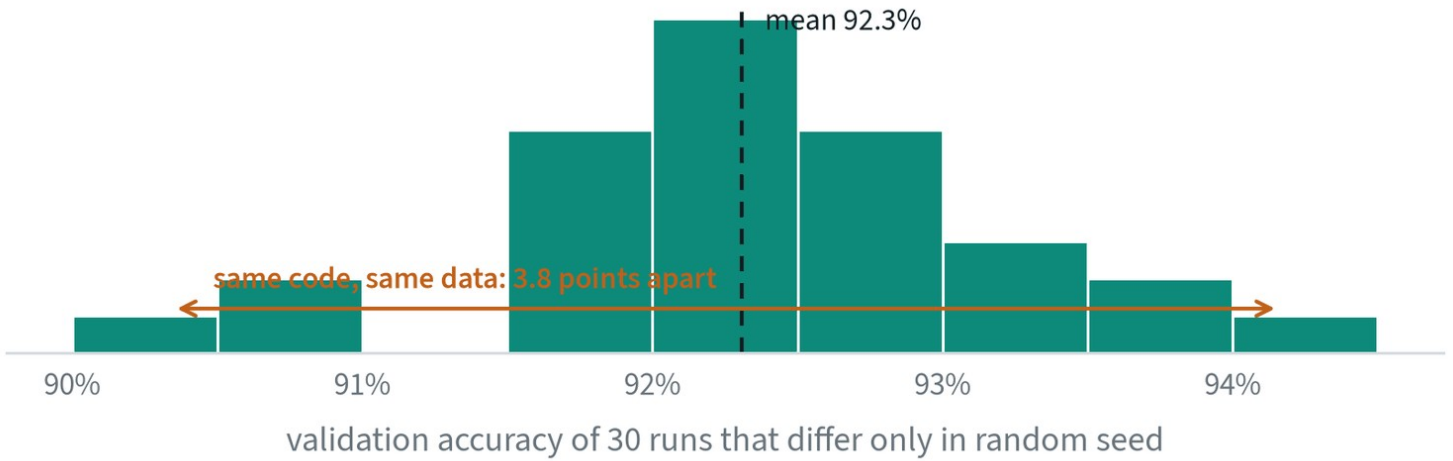
The dumb baseline. On recorded data, “repeat the last action” is a strong score to beat.



A model that only matches “repeat the last action” has learned nothing about the task

How much of a result is noise

measured for this note: one small classifier, one dataset, one configuration, 30 random seeds



The spread. Runs range from 90.3% to 94.2%, mean 92.3%, standard deviation 0.8. A 1.5-point gain from single runs means nothing.

Run several seeds. Report the mean and the spread. Three is a minimum; five or more for a claim that matters.

Compare the gap with the spread. A difference between two means is interesting only when it is large compared with $\sigma/\sqrt{n}$.

Fix what can be fixed. Use the same seeds, data order and evaluation set for both arms.

Know how much a number moves when nothing changes before reading a difference into it

Numbers you can check before trusting a run

several quantities have a known correct value; checking them costs minutes

Check	What it should be	What a wrong value means
Loss at the first step	ln C for C balanced classes. The small network above starts at 2.308; ln 10 = 2.303	Much larger: bad initialisation or unnormalised inputs. Much smaller: a label in the input
Overfitting a single batch	Loss goes to zero and the model reproduces those samples exactly	If it cannot, there is a bug in the model, the loss or the labels
A shuffled-label run	Validation performance at chance	Anything better means information is leaking from the evaluation set
Input ablation	Blanking an input the task needs makes performance collapse	If nothing changes, the model is not using that input
Predicted against recorded, by eye	Plots of predicted and recorded actions for whole episodes overlap	A constant offset or a lag points to normalisation or timing

Five cheap checks with known answers catch most broken pipelines

Leakage in robot data

consecutive frames are nearly identical, and episodes from one session share almost everything

Split by episode, never by frame

- Frames 100 and 101 of one episode are the same image.
- A random split by frame puts near-copies in training, and validation loss then measures memory.

Then ask what else is shared

- Episodes from the same scene, object set or operator share more than the split suggests.
- To claim generalisation to new scenes, hold out whole scenes.

Statistics from the training split only

- Normalisation means and ranges computed over all the data carry information from the validation set.

Count the choices made on the test set

- Every checkpoint, threshold or hyperparameter chosen on test results fits the test set a little.
- Keep a final set that is looked at once.

Split by episode, hold out whole scenes for scene claims, and keep a final set that is looked at once

Habits that catch it

five routines that find silent failures before they cost a training run

Look at the data

- View raw samples after every transform, not before.

Overfit one batch

- If the model cannot memorise ten samples, there is a bug.

Beat a dumb baseline

- Compare with “predict the mean” or “repeat the last action”.
- If you cannot beat it, stop and find out why.

Ask what shortcut exists

- Ask whether the model could score well without solving the task.
- Design a test that separates the two.

Test in the loop

- Run the policy. Do not only score it.

A falling loss proves that the code runs. It does not prove the model learned what you intended

Common mistakes

three reflexes that waste the most time when a result surprises you

Changing the model first

- When a result surprises you, check the data and the evaluation before touching the architecture.

Trusting an aggregate

- One average hides which cases fail.
- Break results down before drawing a conclusion.

Skipping the control

- Without a matched run that differs in one way, an improvement cannot be attributed to anything.

Check the data and the evaluation first, break results down, and always run the control

FOUNDATIONS · TOPIC 2 OF 32

Foundations that survived

The training loop, learning rates, losses, overfitting and the stabilisers that make deep networks trainable.

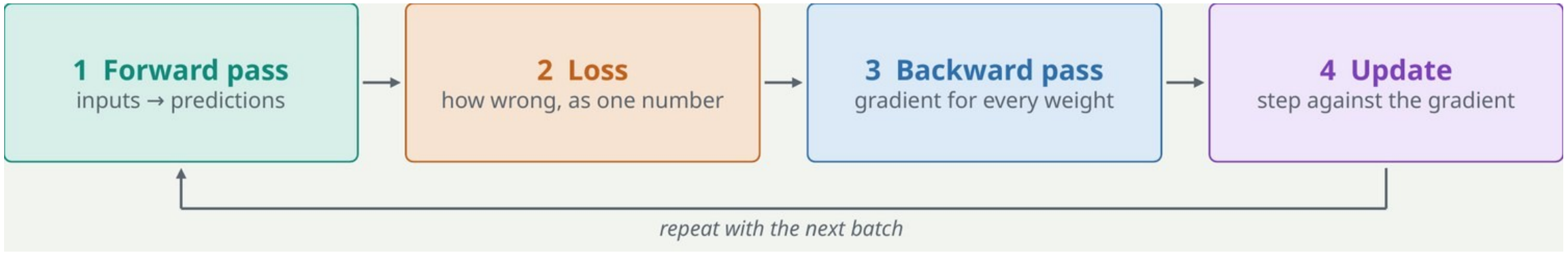
KEY POINTS

- Training is a search for weights that make the loss small on data the model has not memorised.
- If training diverges or stalls, look at the learning rate before anything else.
- Pick the loss that matches the kind of output: a class, a number, a distribution, or a similarity.
- Watch the gap between training and validation loss; that gap is overfitting.

Run it: The training loop, from scratch · Learning rate and optimisers · Why deep networks train at all (github.com/wnukmat/robot-learning-notebooks)

The training loop

every model in this guide, from a small classifier to a robot foundation model, is trained the same way



Forward pass and loss. A batch of inputs becomes predictions, and the loss scores how wrong they are as a single number.

Backward pass. Backpropagation applies the chain rule layer by layer to find how much each weight contributed. That is the gradient.

Update. Every weight is nudged a small step against its gradient, and the loop repeats with the next batch.

AdamW is the default optimiser. It gives every weight its own step size and applies weight decay as a separate step.

Training is a search for weights that make the loss small on data the model has not memorised

The same loop in symbols

a model f with parameters θ, a batch B, and the chain rule applied from the loss backwards

One step, and one layer of backpropagation

$$g_t = \nabla_\theta \frac{1}{|B|} \sum_{(x,y)\in B} \ell\big(f_\theta(x), y\big) \qquad \theta_{t+1} = \theta_t - \eta\, g_t$$

Average the loss over the batch, take its gradient, step against it by η

$$\frac{\partial \ell}{\partial h_{l-1}} = W_l^\top \left(\phi'(W_l h_{l-1}) \odot \frac{\partial \ell}{\partial h_l} \right)$$

For a layer $h_l = \varphi(W_l\, h_{l-1})$, the gradient at its input follows from the gradient at its output

$$\frac{\partial \ell}{\partial W_l} = \left(\phi'(W_l h_{l-1}) \odot \frac{\partial \ell}{\partial h_l} \right) h_{l-1}^\top$$

The same quantity times the layer's input gives the gradient for its weights

In words

Cost. The backward pass costs about the same as the forward pass, so a training step is roughly three times the cost of inference.

Fragile depth. If the per-layer terms are mostly smaller than one the product vanishes. If larger, it explodes.

Batch noise. The batch gradient is a noisy estimate of the full gradient, with noise that shrinks as $1/\sqrt{|B|}$.

The gradient at an early layer is a product of one term per later layer, which is why depth is fragile

Optimisers add memory and per-parameter scaling

plain gradient descent uses one step size for every parameter and forgets the past

Momentum and Adam

$$v_t = \mu\, v_{t-1} + g_t \qquad \theta_{t+1} = \theta_t - \eta\, v_t$$

Momentum: step along a running average of past gradients

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2$$

Adam: running averages of the gradient and of its square

$$\theta_{t+1} = \theta_t - \eta\, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Each parameter gets its own step: large where gradients were small, small where large



Same network and data, each optimiser at a reasonable learning rate, 30 passes. Validation loss: plain SGD 0.19 and still descending, momentum 0.049, AdamW 0.064.

No optimiser is best in general; AdamW is the transformer default because it forgives the learning rate

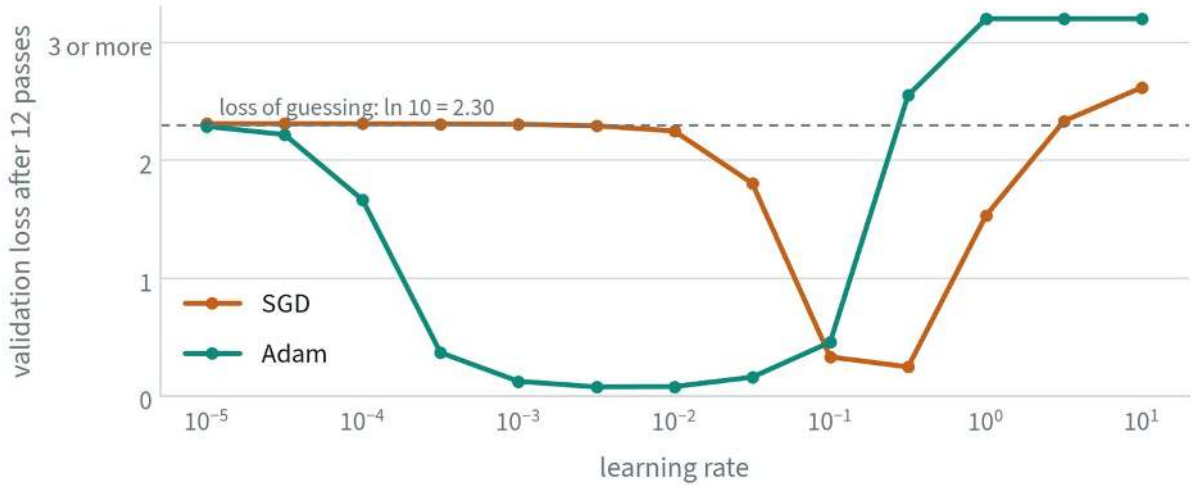
The learning rate is the most important number to tune

measured for this note on a small classifier: validation loss after 12 passes, mean of three runs per point

Warm up, then decay

$$\eta_t = \begin{cases} \eta_{\max}\, t / T_w & t < T_w \\ \frac{1}{2}\,\eta_{\max}\left(1 + \cos\frac{\pi\,(t - T_w)}{T - T_w}\right) & t \geq T_w \end{cases}$$

A linear ramp from zero over the first steps, then a decay along a cosine



Each optimiser has a basin. Adam works from about 0.0003 to 0.03 and diverges above 0.3. SGD needs a rate about a hundred times larger, near 0.1 to 0.3.

If training diverges or stalls, look at the learning rate before anything else

Loss functions tell the model what “wrong” means

the loss is the only signal the model gets, so pick the one that matches the kind of output

Loss	Use it when the output is	Intuition	Where you will see it
Cross entropy	One of several classes, or the next token	Penalises low probability on the correct answer	Classifiers, language models, action tokens
MSE	A continuous number	Penalises large errors heavily; predicts the average	Regression, diffusion and flow-matching targets
L1 or Huber	A continuous number with outliers	Penalises errors evenly	Action regression, depth, boxes
KL divergence	A probability distribution	How different two distributions are	VAEs, distillation from a teacher
Contrastive	An embedding	Pull matching pairs together, push the rest apart	CLIP-style models, retrieval

MSE predicts the mean, so a policy trained with it averages two valid ways of doing a task

Why these losses: maximum likelihood

each loss in the table says the same thing: make the training data as probable as possible under the model

Softmax and cross entropy, written out

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \qquad \ell = -\log p_y = -z_y + \log \sum_j e^{z_j}$$

Softmax turns logits z into probabilities; the loss is minus the log of the correct one

$$\frac{\partial \ell}{\partial z_i} = p_i - 1[i = y]$$

The gradient at the logits is the predicted probabilities minus the truth

In words

Squared error assumes one right answer blurred by bell-shaped noise. Fitted to two right answers, a single bell curve sits in the middle.

Every position. A language model applies cross entropy at each position, with the vocabulary as the classes. Action-token policies do the same.

Label smoothing trains toward 1 − ε on the correct class, which stops the logits growing without bound and improves calibration.

Huber is quadratic for small errors and linear for large ones, so a few bad labels do not dominate the gradient.

Cross entropy is maximum likelihood for classes; squared error is the same idea under bell-shaped noise

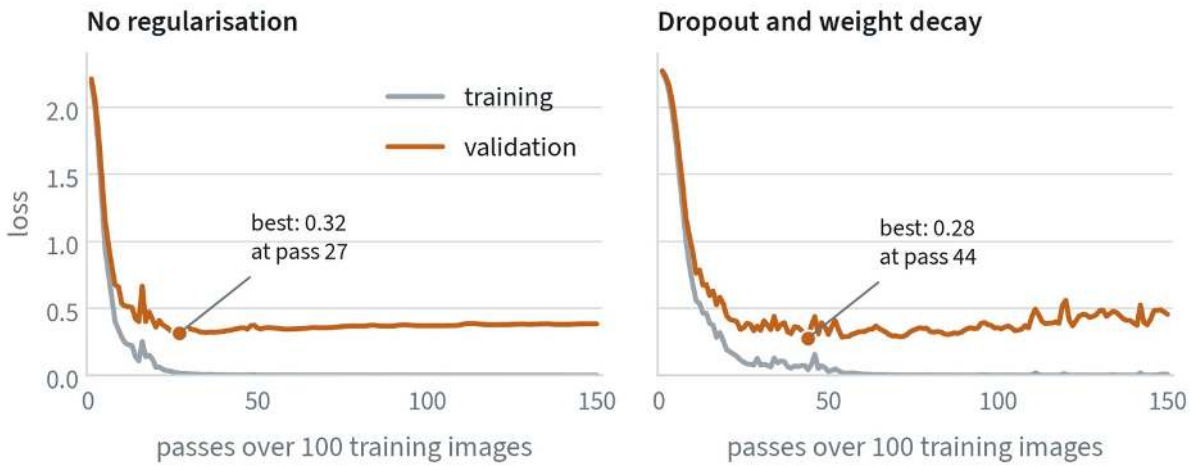
Overfitting and capacity

measured for this note: a network with far more parameters than its 100 training images, 150 passes

Bias, variance and noise

$$\mathbb{E}\left[(y - \hat{f}(x))^2\right] = \underbrace{\left(\mathbb{E}[\hat{f}(x)] - f(x)\right)^2}_{\text{bias}^2} + \underbrace{\mathrm{Var}\left[\hat{f}(x)\right]}_{\text{variance}} + \underbrace{\sigma^2}_{\text{noise}}$$

Expected error splits into a too-simple model, sensitivity to the sample, and noise



Training loss reaches zero in both runs. Unregularised, validation loss is lowest at pass 27, then creeps up. With dropout and weight decay the best is lower and later.

Watch the gap between training and validation loss; that gap is overfitting

Convolutions: the older building block that stayed

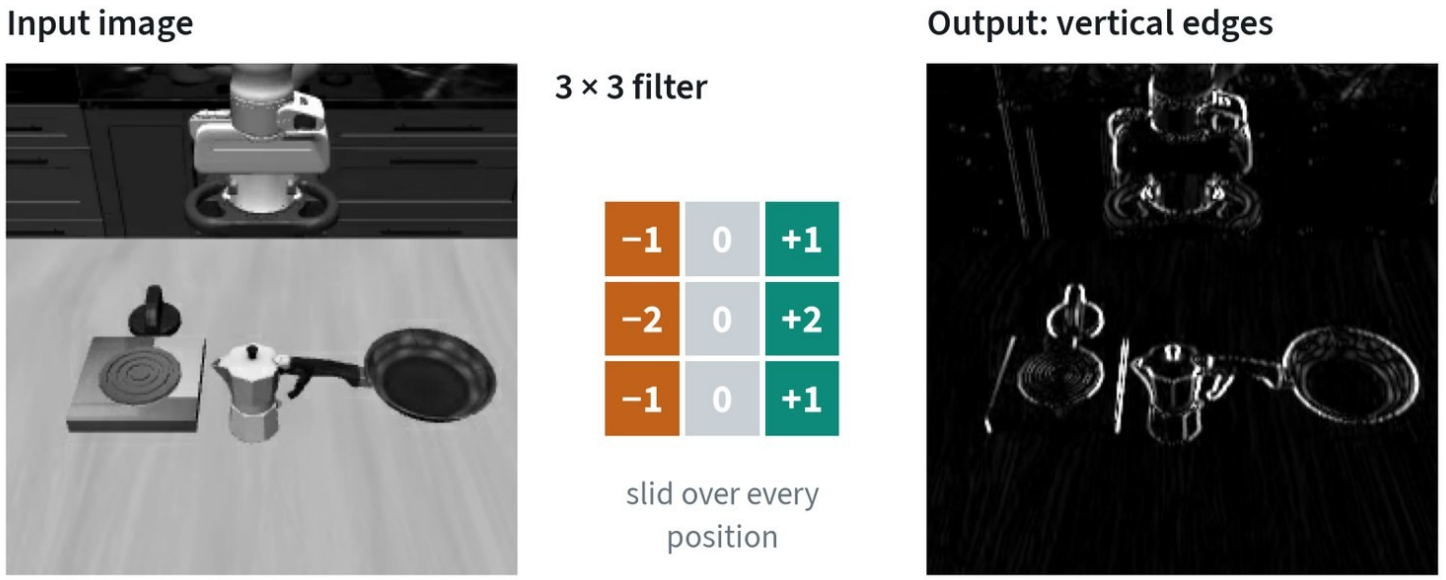
a real camera frame passed through one hand-written 3 × 3 filter

What it does. A small grid of weights slides over the image and outputs, at each position, how well the patch underneath matches.

In the figure. Weights are negative on the left and positive on the right, so the output is bright along vertical edges of the objects.

Shared and local. The same weights are used at every position, and each output sees a small neighbourhood. Edges first, then parts and objects.

Where they remain. The patch-token step is a convolution. Image and video autoencoders are convolutional, as are some policies' vision networks.



Transformers replaced convolutions as the main architecture, and convolutions are still in most systems

Why deep networks train at all

measured for this note: accuracy against depth, and the gradient reaching each layer of a 32-layer network before training

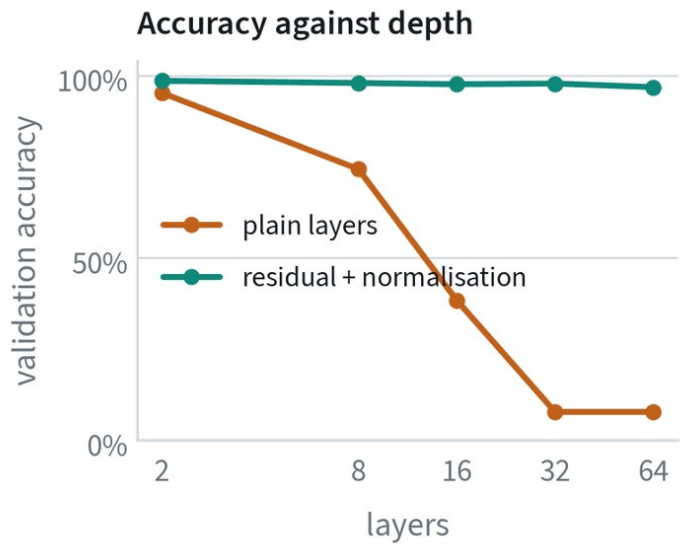
The stabilisers

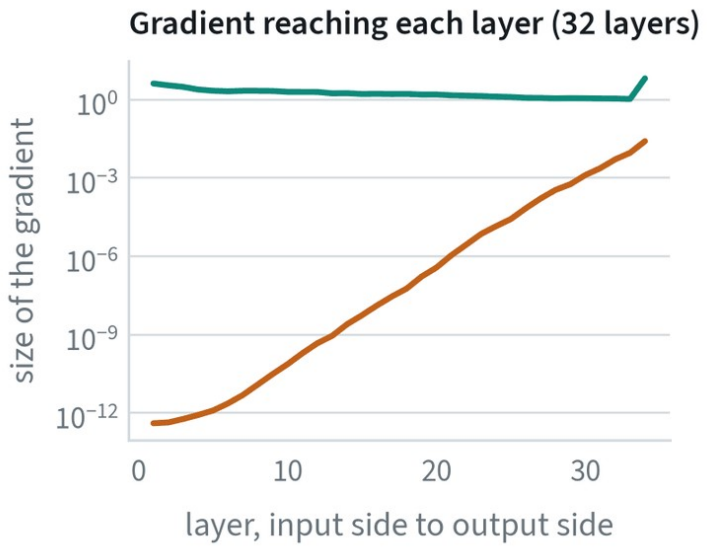
Plain stacks fail. 95% at 2 layers, 38% at 16, chance at 32. The gradient shrinks by eleven orders of magnitude on its way to the first layer.

Initialisation and normalisation. Var(W) = 2/n_in for a ReLU layer keeps signals the same size. LayerNorm or RMSNorm rescales activations inside the network.

Residual connections. h + F(h) has derivative I + ∂F/∂h. The identity term is the flat line: 97 to 99% accuracy up to 64 layers.

Regularisation. Weight decay, dropout, augmentation and early stopping keep the model from memorising.





Residual connections and normalisation make deep models trainable; regularisation makes them generalise

Common mistakes

three errors in reading and running a training job

Tuning everything except the learning rate

- If training diverges or stalls, look there first.

Stopping at the step budget

- A run is finished when the validation curve has flattened, not when the counter runs out.

Reading the training loss alone

- It always improves.
- Only the validation curve says whether the model is learning or memorising.

Look at the learning rate first, and read the validation curve, not the step counter

FOUNDATIONS · TOPIC 3 OF 32

Sequence-to-sequence models

One sequence in, another out: the bottleneck, attention, teacher forcing, decoding, and why robot control has the same shape.

KEY POINTS

- The question in every design is how the output side reaches the input side.
- A single vector between encoder and decoder fails on long inputs; attention removes the limit.
- Teacher forcing makes training fast and leaves the model unused to its own mistakes.

Run it: Sequence to sequence: the bottleneck and how attention removes it (github.com/wnukmat/robot-learning-notebooks)

One sequence in, another out: three designs

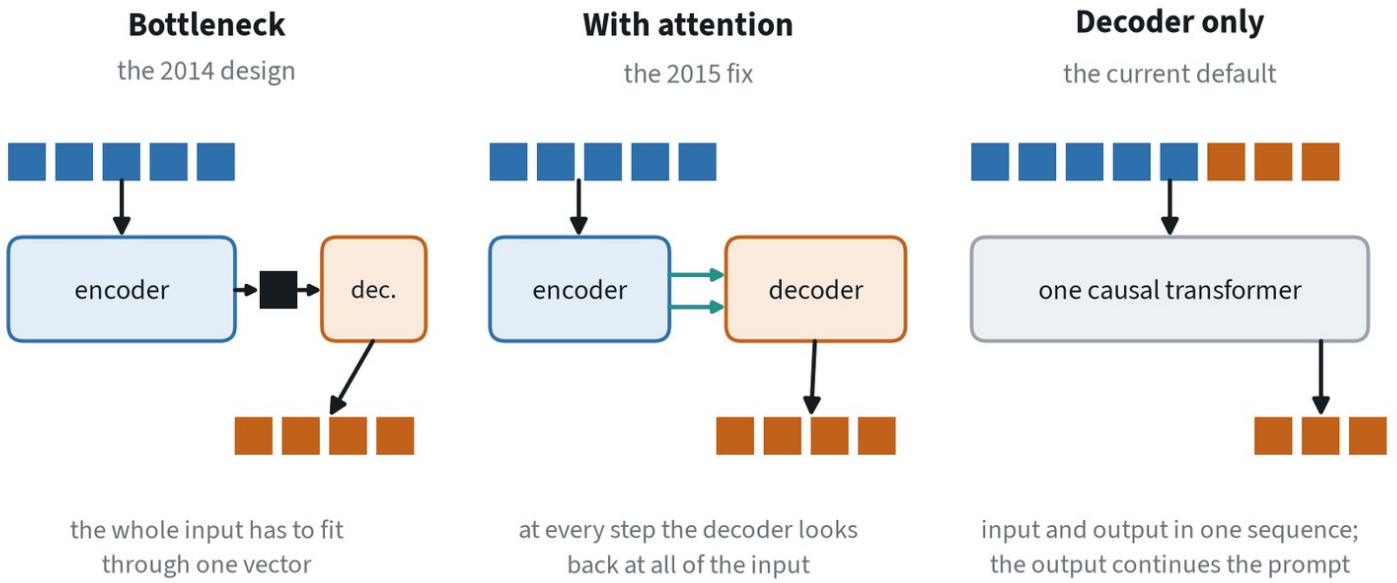
translation, summarising, transcription and robot control share one shape; the designs are in the order they appeared

No fixed alignment. The third output word need not match the third input word, and the output can be longer or shorter.

The bottleneck. An encoder compresses the input into one vector and a decoder writes from it. All it knows of the input must fit there.

Attention. Keep every encoder output and let the decoder look back at all of them at every step, weighted by relevance.

Decoder only. Put input and output in one sequence and train a single causal model to continue it. The input is a prompt.



The question in every design is how the output side reaches the input side

The model as a probability

the probability of an output sequence y given an input x, built one token at a time by the chain rule

Factorise, then maximise

$$p_\theta(y \mid x) = \prod_{t=1}^{T} p_\theta\left(y_t \mid y_{<t},\, x\right)$$

Each token is predicted from the input and the tokens before it

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log p_\theta\left(y_t \mid y_{<t},\, x\right)$$

Training maximises the log-probability of the correct output

In words

Each factor is a softmax over the vocabulary.

The loss is cross entropy summed over positions.

Perplexity is the exponential of the average per-token loss: roughly, how many equally likely choices the model hesitates between.

With an empty input the same factorisation is a language model, which is why the two share everything

The bottleneck, measured

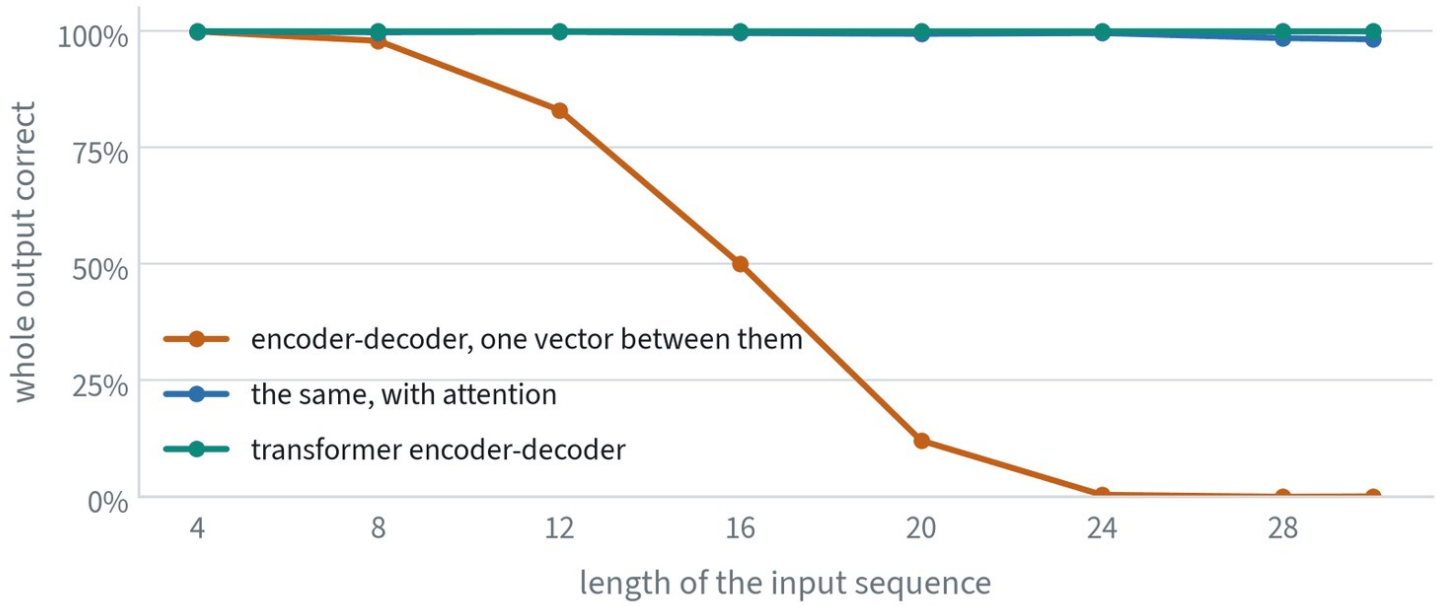
trained for this note: reverse the input sequence; lengths 3 to 30 seen in training; mean of three runs

One vector. Share of entirely correct outputs: 98% at length 8, 83% at 12, 50% at 16, 12% at 20, zero from 24.

Why. The encoder's single vector, 96 numbers here, cannot hold a long sequence precisely enough to reproduce it.

With attention. The same recurrent model plus about 30,000 parameters holds 98% or better at every length up to 30.

Transformer. Attention throughout, and no errors at all.



A single vector between encoder and decoder fails on long inputs; attention removes the limit

The attention step, written out

encoder outputs h_1 to h_n, decoder state s_t at step t

Score, normalise, average

$$e_{t,i} = \text{score}(s_t, h_i) \qquad \alpha_{t,i} = \frac{\exp e_{t,i}}{\sum_j \exp e_{t,j}}$$

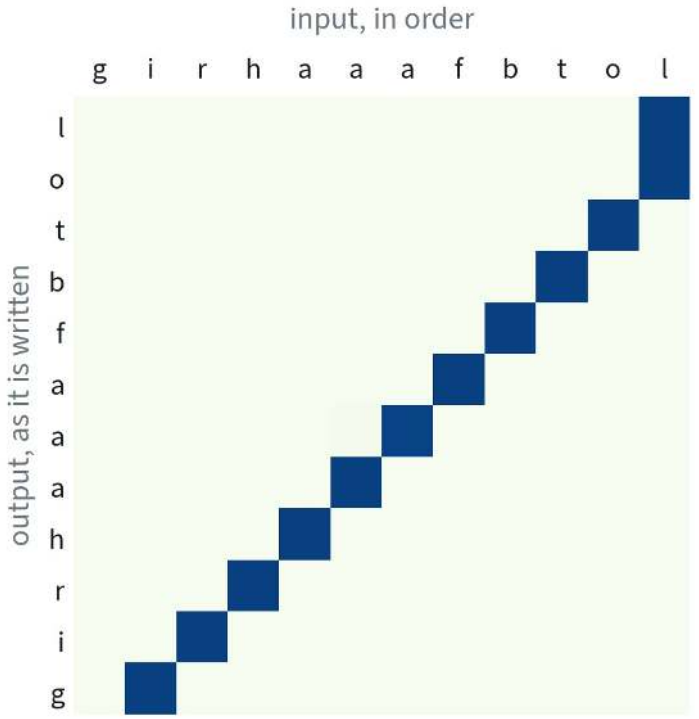
Score every input position against the current state, then normalise

$$c_t = \sum_i \alpha_{t,i}\, h_i$$

The context is the weighted average of the encoder outputs

$$v^\top \tanh(W_s s_t + W_h h_i) \quad \text{or} \quad s_t^\top W\, h_i$$

Two scores: additive, the original, and dot-product, which the transformer adopted



The weights α of the middle model reversing a twelve-symbol sequence. Each row is one output step. One dark cell per row: it walks backwards through the input.

Nobody told the model that reversal means reading backwards; the attention weights show it found out

Training: teacher forcing and its cost

the decoder predicts each token from the tokens before it; in training those come from the correct output

Teacher forcing

- Earlier tokens are taken from the correct output, not from the model's own predictions.
- Every position is trained in parallel in a single pass, which is why these models train efficiently.

Exposure bias

- In training the model always sees a perfect history. In use it sees its own output, mistakes included.
- One early error can send everything after it off course.

Remedies

- Train on more varied data.
- Occasionally feed the model its own predictions during training.
- Generate the output in a way that does not depend on a long chain of earlier choices.

Teacher forcing makes training fast and leaves the model unused to its own mistakes

Decoding: choosing the output

four ways to turn per-token probabilities into a sequence, and the length correction beam search needs

Beam search score

$$\text{score}(y) = \frac{1}{|y|^{\alpha}} \sum_{t=1}^{|y|} \log p_\theta\left(y_t \mid y_{<t},\, x\right)$$

Total log-probability divided by a function of the length, with α between 0.6 and 1

In words

Greedy. The most probable token at each step. Fast and fine with one right answer; an early poor choice cannot be undone.

Beam search. Keeps the k best partial outputs and extends them all. For translation and transcription; slower, tends to short, bland text.

Sampling. Draws from the probabilities, often with a temperature or cut-off. Gives variety; can go wrong and is not repeatable.

Constrained. Allows only tokens that keep the output valid: code, coordinates, actions. Needs a description of what is valid.

Beam width. Widths of 4 to 8 capture most of the gain. Much wider beams often make output worse.

For robot actions decoding is greedy or constrained, because a sampled outlier becomes a jerk of the arm

What the tokens are

real outputs of a sub-word tokenizer with a vocabulary of about 257,000 entries

Text	Tokens	Pieces
pick up the red cup	5	pick · up · the · red · cup
teleoperation	2	tele · operation
proprioception	3	propri · o · ception
unscrewing the moka pot	7	uns · cre · wing · the · mo · ka · pot
3.14159	7	3 · . · 1 · 4 · 1 · 5 · 9

Unusual words cost more tokens, and numbers are split digit by digit

Encoder-decoder or decoder only

two ways to arrange the same parts, compared row by row

	Encoder-decoder	Decoder only
Input is read	In both directions, by a dedicated encoder	Left to right, as part of the same sequence
Parameters	Two stacks, with cross-attention between them	One stack
Reusing the input	The encoder's output is computed once and attended to throughout	The key-value cache does the same job
Pretraining	Needs paired inputs and outputs, or a denoising task	Next-token prediction on any text
Where it is used	Translation, speech recognition, some robot policies	Language models, vision-language models, most new systems

Decoder-only models won mainly on pretraining: next-token prediction works on any text

Robot control as sequence-to-sequence

a sequence of observations in, a sequence of actions out

Action chunking with a transformer

- An encoder reads the camera images and joint positions.
- A decoder writes the next chunk of actions, all at once. The encoder-decoder shape applied directly.

Actions as tokens

- A decoder-only vision-language model continues its prompt with action tokens, exactly as it would continue a sentence.

Instructions to plans

- A sentence in, a list of sub-tasks out.

Video to actions

- A sequence of frames in, the actions that produced them out.

Bottlenecks, exposure bias and decoding were learned on text first, and each has a counterpart on a robot

Common mistakes

three errors in evaluating and running sequence models

Judging by per-token accuracy

- One wrong token can ruin an output.
- Score whole outputs.

Forgetting the mismatch

- A model that looks perfect with the correct history can still drift on its own output.

Sampling where one answer is right

- Use greedy or beam search for translation, transcription and actions.

Score whole outputs, test the model on its own history, and do not sample where one answer is right

FOUNDATIONS · TOPIC 4 OF 32

The transformer

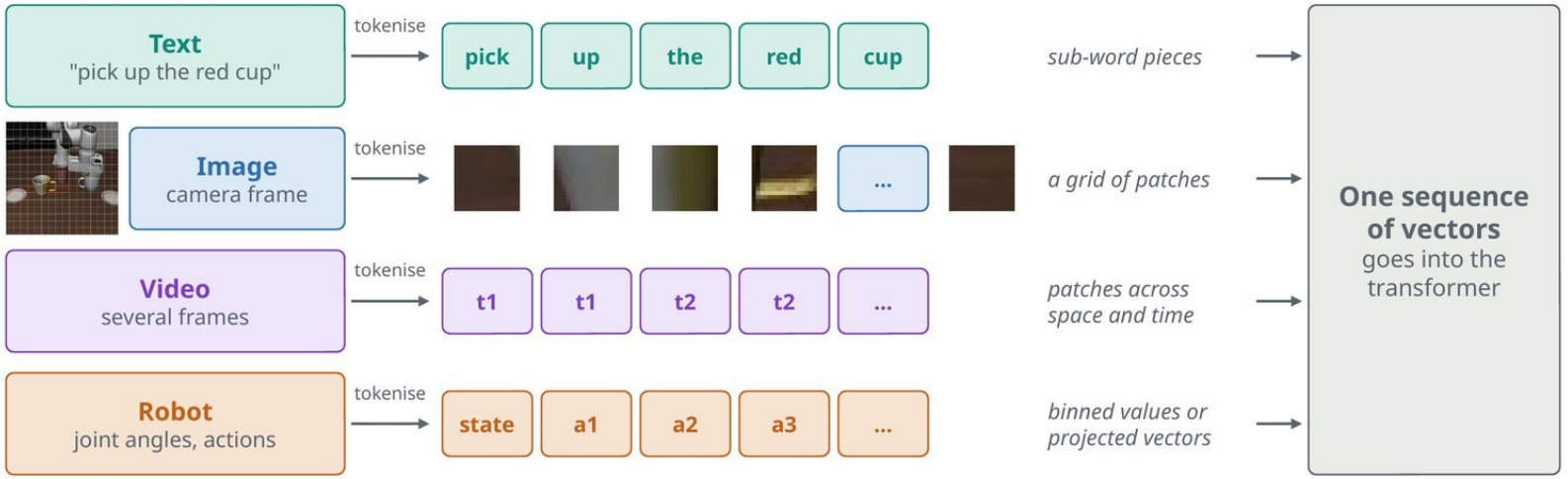
One block, stacked many times, that now reads text, images, video and robot actions.

KEY POINTS

- Everything is turned into tokens, and one block is repeated to process them.
- Attention moves information between tokens. The MLPs, which hold most of the parameters, transform it.
- A model of L layers and width d has about $12 \cdot L \cdot d^2$ parameters; cost scales with parameters times tokens.

Everything becomes tokens

a transformer reads one sequence of vectors; the first job of any model is to build that sequence



Text is split into sub-word pieces, and each piece is looked up in a table of learned vectors.

Images are cut into patches, for example 16 by 16 pixels, each flattened and projected. A 224-pixel image gives 14 × 14 = 196 tokens.

Video adds a time axis.

Robot data. Joint angles and actions are sorted into numbered bins, so they look like words, or projected as continuous vectors.

After tokenising, the model neither knows nor cares where a token came from

Attention: every token looks at every other token

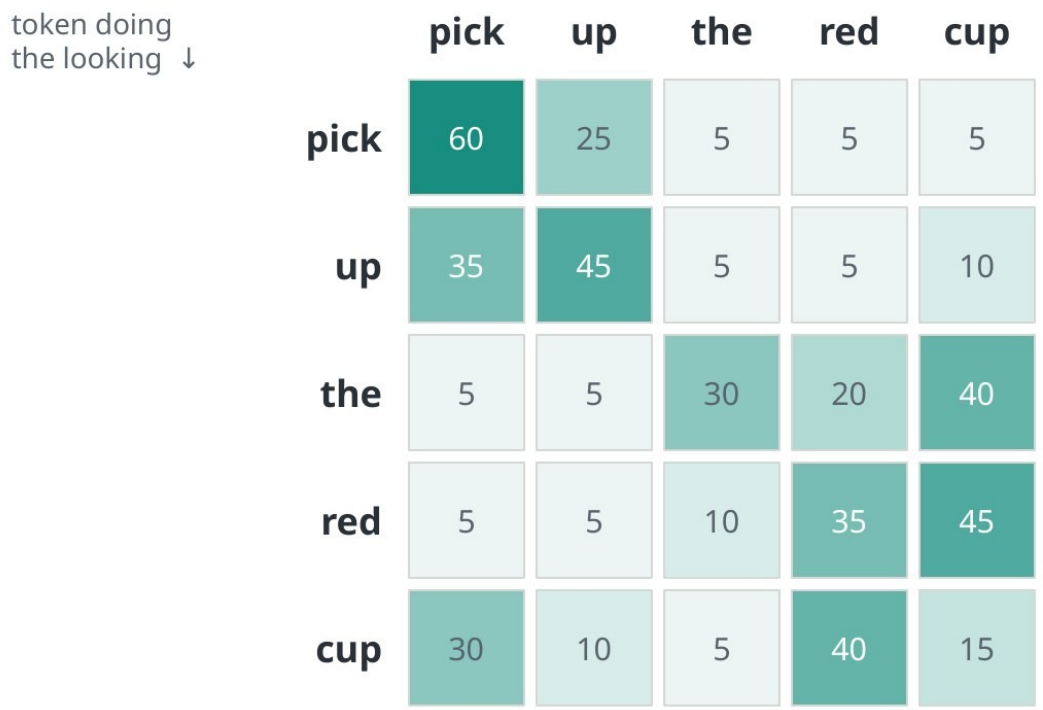
illustrative weights, as percentages, for the words pick, up, the, red, cup

Query, key, value. Each token says what it looks for, what it contains and what it passes on. Queries are compared with every key.

Each row is one token deciding where to look. A softmax makes the weights sum to one, and the new vector is the weighted sum of values.

Read the last row. “Cup” puts most of its weight on “red” and “pick”, so it now means something like the red cup that is to be picked.

Heads. Several of these run in parallel so that different heads can follow different relationships.



Read the last row: "cup" puts most of its attention on "red" and "pick". Its new vector now means "the red cup that is to be picked".

Numbers are percentages and each row sums to 100.

Attention is how a token gathers context, at a cost that grows with the square of the sequence length

Inside one block

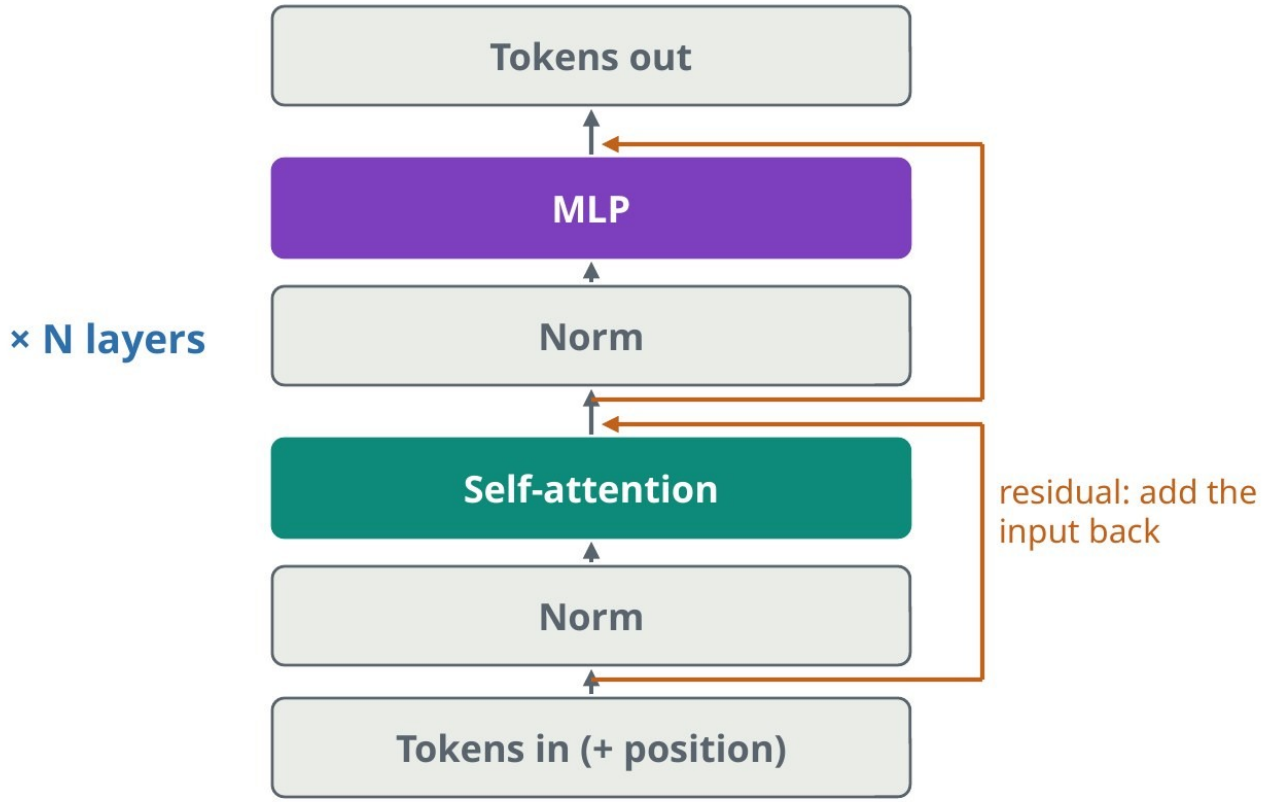
read from the bottom up: tokens, normalisation, self-attention, normalisation, MLP; orange lines are residual connections

Attention sub-layer. Moves information between tokens.

MLP sub-layer. A small network applied to each token on its own. It holds most of the model's parameters.

Normalise, then add back. The addition is the residual connection, a direct path for gradients that lets very deep stacks train.

Position. Attention has no sense of order. Rotary encoding rotates queries and keys by an angle that depends on position.



A transformer is the same block repeated: attention moves information, the MLP transforms it

One block in equations

X holds n tokens of width d; the now-standard pre-normalisation form is two residual updates

Two residual updates

$$X' = X + \mathrm{Attn}\big(\mathrm{Norm}(X)\big) \qquad X'' = X' + \mathrm{MLP}\big(\mathrm{Norm}(X')\big)$$

Each sub-layer normalises its input and adds its output back

$$\mathrm{MLP}(x) = W_2\,\sigma(W_1 x) \qquad W_1 \in \mathbb{R}^{d_{\mathrm{ff}} \times d},\ W_2 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$$

Per token: widen to d_ff, usually 4d, apply a non-linearity, narrow back

$$W_3\big(\sigma(W_1 x) \odot W_2 x\big)$$

The gated variant, with three matrices, common in current language models

In words

Pre-normalisation. The original transformer normalised after each sub-layer. Normalising before it avoids a delicate warmup.

The output. After the last block, a final normalisation and a linear layer map each token to scores over the vocabulary.

Weight tying. Usually that linear layer shares its weights with the input embedding table.

Normalising before each sub-layer keeps the residual path clean, so hundreds of layers train

Where the parameters are

attention has four d × d matrices: query, key, value and output; the MLP has two of size d × d_ff

Counting one block

$$\underbrace{4d^2}_{\text{attention}} + \underbrace{2 \cdot d \cdot 4d}_{\text{MLP}} = 12\,d^2 \text{ per block}$$

With d_ff = 4d, one block has about 12 d² parameters

$$\text{parameters} \approx 12\,L\,d^2$$

A model of L layers has about 12 L d² parameters

In words

Check on a real model. For a vision transformer with L = 12 and d = 768 the formula predicts 85 million. The real model has 87 million.

Two thirds MLP. Of the 12 d² in a block, 8 d² sit in the MLP.

Compute per token is about two floating-point operations per parameter, so cost scales with parameters times tokens.

Plus attention. On top of that comes the attention term that grows with the square of the sequence.

A model of L layers and width d has about 12·L·d² parameters, and two thirds of each block is MLP

The count in two real models

counted from CLIP ViT-B/32's image encoder, 87 million, and the Gemma 2B language model, 2.51 billion

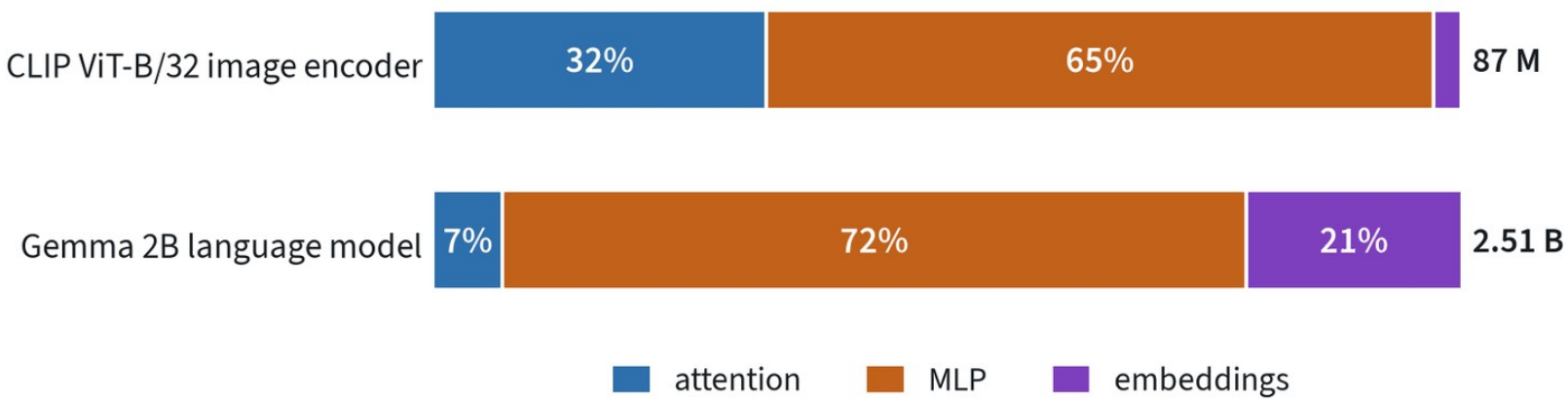


Image encoder. 32% attention and 65% MLP.

Language model. 7% attention, 72% MLP and 21% embeddings: one vector for each of 257,216 tokens.

Why attention is so small. 18 layers of width 2,048 with an MLP width of 16,384, and eight query heads sharing one key and value head.

Memory when generating. Each token stores 18 layers × 2 × 256 numbers: about 18 kilobytes at 16-bit precision.

Attention decides where information moves; the MLPs, where most weights sit, store and transform it

Three shapes, and how they are made fast

the shapes differ in what a token is allowed to see

Shape	What a token can see	Good at	Where it is used
Encoder	All tokens, in both directions	Understanding an input	Vision encoders, embeddings, classifiers
Decoder	Only earlier tokens	Generating a sequence step by step	Language models, vision-language models
Encoder-decoder	The decoder also reads the encoder's output	Mapping one sequence to another	Translation, some action policies

The KV cache makes the next token far cheaper than reprocessing the whole prompt

Common mistakes

three errors that follow from forgetting how the block works

Forgetting position

- Without position information a transformer treats its input as an unordered set.

Reading attention maps as explanations

- They show where weight went.
- That is a hint about the model's reasoning and not a proof.

Ignoring the quadratic cost

- Doubling image resolution gives four times the tokens and about sixteen times the attention cost.

Supply position, treat attention maps as hints, and count tokens before raising resolution

FOUNDATIONS · TOPIC 5 OF 32

Attention in depth

One operation, many forms: self, cross and shared attention, heads and masks, what it costs, and the linear recurrent variants.

KEY POINTS

- Every kind of attention is the same lookup. They differ in where queries, keys and values come from, and in the mask.
- Standard attention costs time that grows with the square of the sequence, and remembers everything.
- Linear recurrent forms cost the same per token forever, and remember a fixed-size summary.

Run it: Attention from scratch (github.com/wnukmat/robot-learning-notebooks)

One operation: a soft lookup

each token asks a question, every token offers an answer, and each leaves with a blend of the best matches

In symbols

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

Scores from queries and keys, turned into weights, applied to values

In words

Query, key, value. Three learned matrices turn each token into what it looks for, what it contains and what it passes on.

Q Kᵀ is an n × n table of scores, one for every pair of tokens.

Softmax turns each row into positive weights that sum to one: one token deciding where to look.

× V replaces each token with the weighted average of the values.

Attention moves information between tokens. It cannot create any; that is the MLP's job

What it looks like in a real model

attention from three words of a robot instruction onto the camera image, in a public vision-language model



One sequence. The image is 256 tokens and the instruction follows it, so a word can attend to any patch.

It lands on the object. "Plate" sends 96% of its attention to image tokens, on the two plates; "mug" sends 91%.

Chosen to be readable. Most of the model's 144 heads are not, and some park weight on background patches.

Evidence, not explanation. Use maps to form a hypothesis about a model, then test it another way.

A map shows where information was read from, not what was done with it

Why divide by √d, and why several heads

two details of the formula that each fix a specific failure

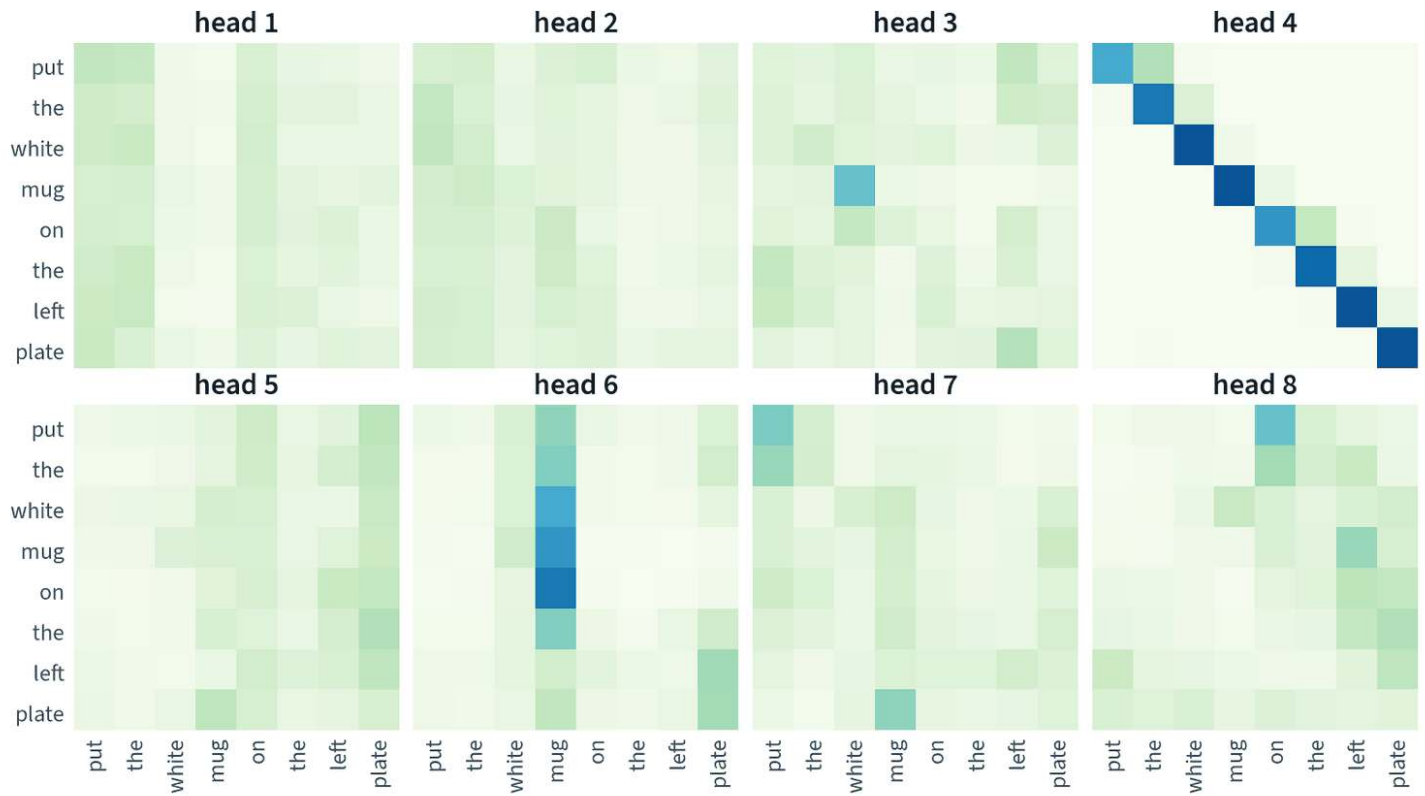
Scale and heads

Scale. A dot product of two width-d vectors has a typical size of √d. Unscaled, the softmax saturates to one token and passes no gradient.

Heads. The width is split into h slices, each with its own queries, keys and values, run in parallel for about the same cost.

Different jobs. Here one head copies each word forward and one gathers everything toward "mug".

Grouped-query attention shares keys and values across heads, shrinking the memory used during generation.



The scale keeps the softmax trainable; heads let one layer follow several relationships at once

What changes with depth

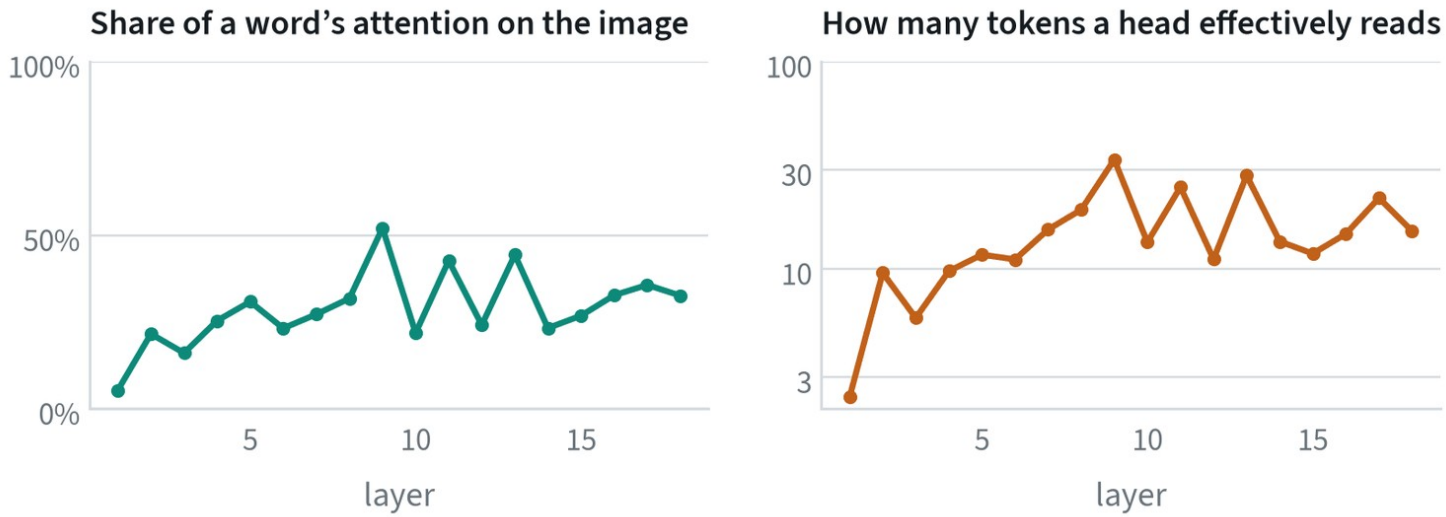
measured across the 18 layers of a real vision-language model

Layer 1 is local. A word sends 5% of its attention to the image and effectively reads two or three tokens.

Later layers mix. A fifth to a half of a word's attention is on the image, across ten to thirty tokens.

Effective count is the exponential of the entropy of a head's weights.

Reading it. Near 1, the head is a pointer to one token. Near the sequence length, it is averaging everything.



Early layers read locally; from the second layer on, words read the image

Masks decide who may look at whom

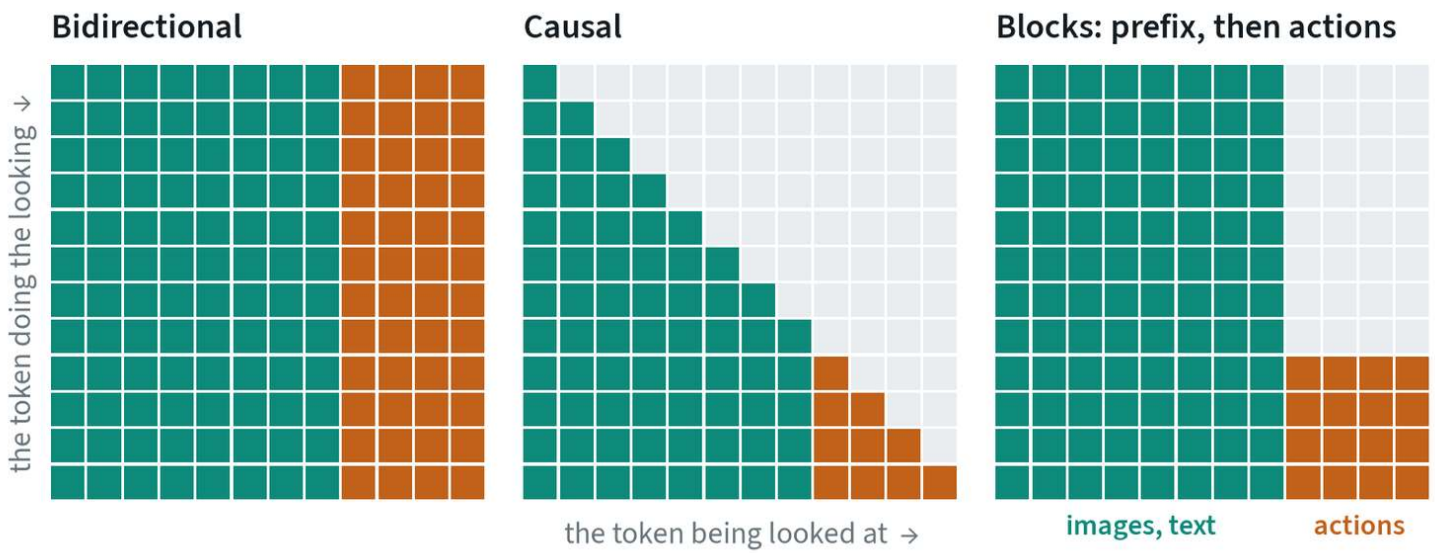
the mask, more than the weights, is what distinguishes one architecture from another

Bidirectional: every token sees every other. Used by encoders.

Causal: a token sees itself and earlier tokens. Used for generation, and it allows cached keys and values.

Blocks: groups with different rules, such as action tokens that see everything while the prefix cannot see them.

Caching follows the mask. A stored result is reusable only if those tokens could not see what came later.



The mask, more than the weights, is what separates an encoder from a generator

Self, cross and shared attention

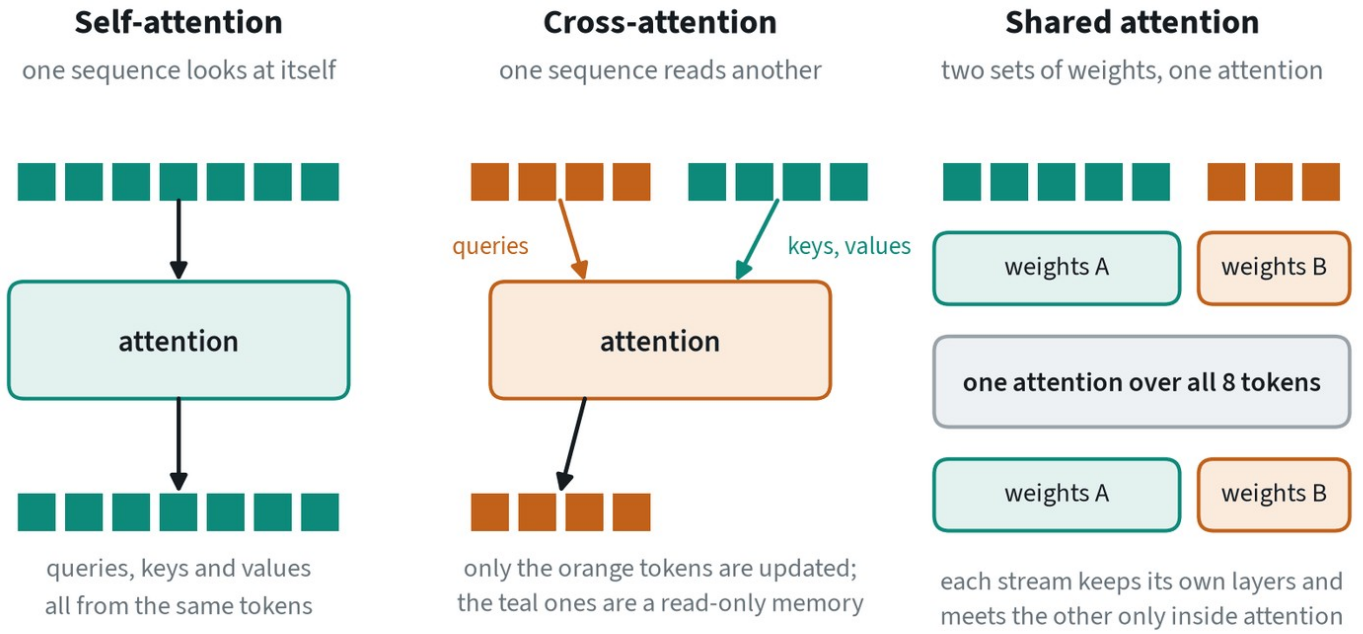
the formula does not say where the queries, keys and values come from

Self: queries, keys and values from one sequence. Every token is updated.

Cross: queries from A, keys and values from B. B is read-only, can be cached, and information flows one way.

Shared: two streams keep separate weights and concatenate their tokens inside attention only.

In robot policies the action network attends to the vision-language tokens, and a block mask stops them looking back.



Cross-attention conditions one model on another; shared attention lets two models meet with their own weights

What attention costs

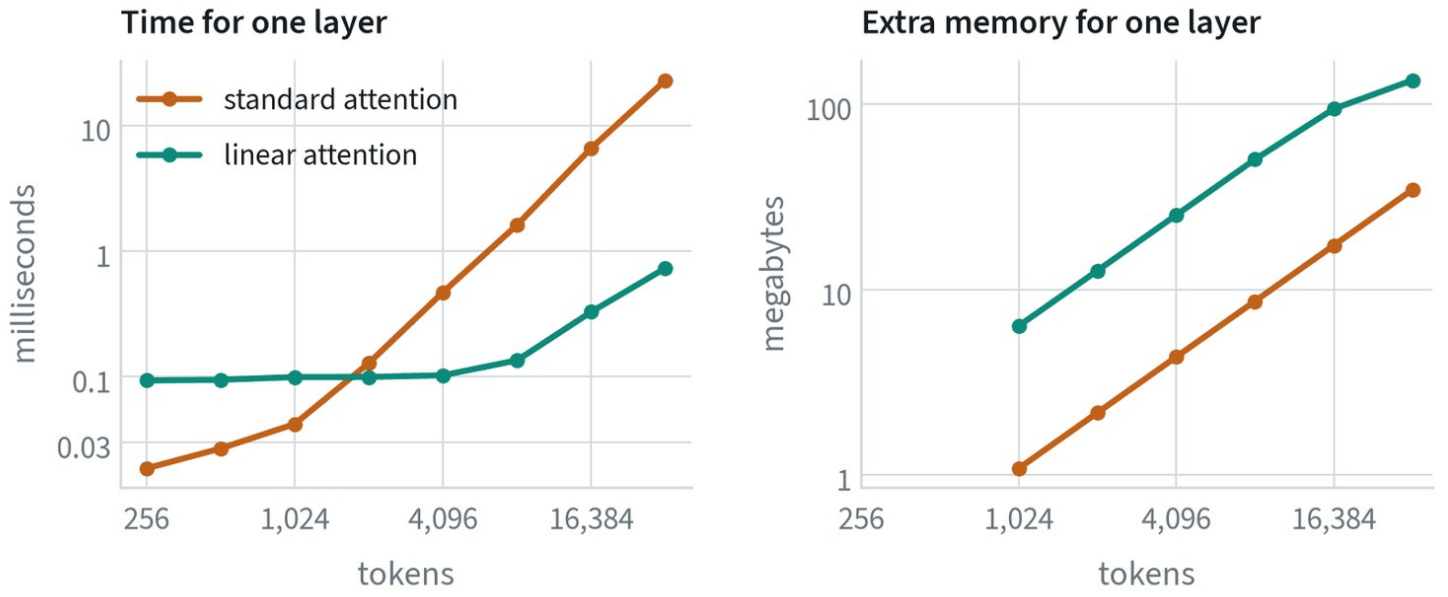
measured on a desktop GPU: one layer, eight heads, 16-bit precision

Quadratic time. The score table has n × n entries. Eight times the tokens took 49 times as long: 0.47 ms at 4,096 and 22.8 ms at 32,768.

Linear memory. FlashAttention computes the table in blocks and discards them, so the result is exact and memory grows only with n.

Key-value cache. During generation, earlier keys and values are stored and each new token computes one row. The cache is the main memory cost.

Ways out: fewer tokens, local windows, or an operation that needs no table.



Time grows with the square of the token count, which is why token counts are the first thing to cut

Linear attention is a recurrence

drop the softmax and the order of the sums can be changed

Without the softmax

$$o_i = \sum_j (q_i \cdot k_j)\, v_j = \Big(\sum_j v_j k_j^\top\Big) q_i = S\, q_i$$

One d × d matrix S answers every query, so the cost is proportional to n

$$S_t = S_{t-1} + v_t k_t^\top, \qquad o_t = S_t\, q_t$$

With a causal mask, S is a running sum: a recurrent state

In words

Why the table exists. The softmax must see a whole row of scores before it can normalise them.

Associative memory. Writing adds a key and value pair. Reading with a query returns the values stored under similar keys.

No growing cache. The state has a fixed size, however long the sequence.

The price. Standard attention can retrieve any past token exactly. A d × d state blurs once it holds more than about d pairs.

A fixed-size state costs the same per token forever, and holds about d clean associations

The delta rule and Gated DeltaNet

a plain running sum can only add; these two refinements let the memory replace and forget

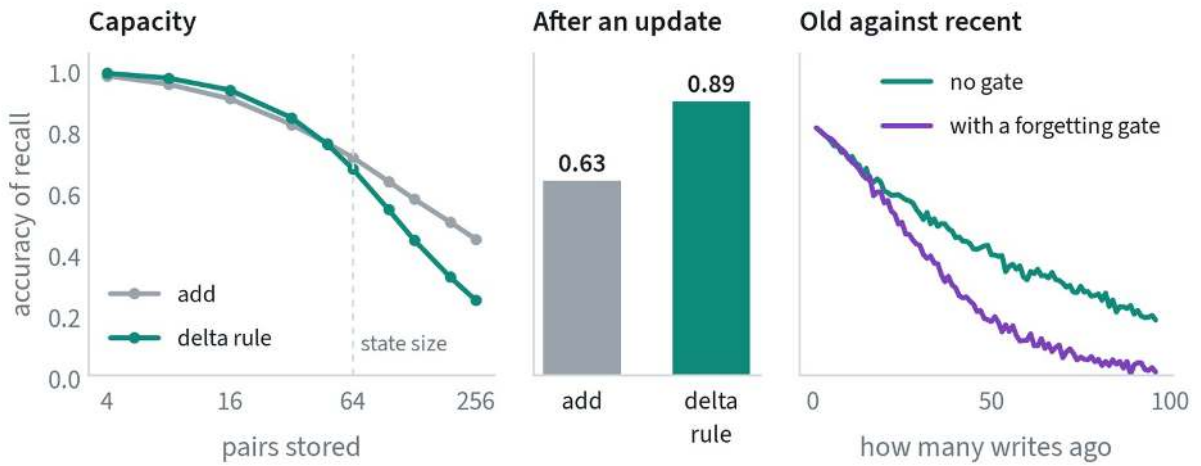
Replace, then forget

$$S_t = S_{t-1}\left(I - \beta_t k_t k_t^\top\right) + \beta_t v_t k_t^\top$$

Delta rule: erase what is stored along the key, then write the new value

$$S_t = S_{t-1}\,\alpha_t\left(I - \beta_t k_t k_t^\top\right) + \beta_t v_t k_t^\top$$

Gated DeltaNet: a learned gate α scales the whole previous state



Random pairs written into a 64 × 64 state. After keys are given new values, recall is 0.63 with plain addition and 0.89 with the delta rule.

The delta rule buys clean updates, not capacity; the gate clears the memory at a change of context

Choosing between them

standard attention remembers everything and pays in time; recurrent forms pay a fixed price and remember a summary

Situation	Reach for	Why
Up to a few thousand tokens	Standard attention	Exact retrieval of any token at an acceptable cost
One model conditioning another	Cross-attention	One-way, cacheable, cost set by the smaller side
A small specialist beside a large model	Shared attention with a block mask	Each keeps its own weights; the large side can be cached
Generation, one token at a time	Causal attention with a key-value cache	Each new token costs one row, not a table
Very long or unbounded streams	Linear recurrent layers, with some standard attention	Constant cost per token and no growing cache
Long video and world models	Compression first, then a mix	Token counts are the dominant cost

Mixed stacks take the cheap summary most of the time and the exact lookup where it matters

Common mistakes with attention

four errors that are easy to make and cheap to avoid

Reading a map as an explanation

- It shows where a head read from.
- Confirm by removing the input and seeing whether the output changes.

Forgetting the mask when caching

- A cached result is valid only if those tokens could not see what came later.

Treating linear attention as free

- It trades exact recall for a fixed-size summary.
- Check the task needs only the summary.

Counting only time

- During generation the key-value cache, not the arithmetic, is often what runs out.

Check conclusions from attention maps by removing the input and watching the output

MODELS · TOPIC 6 OF 32

Representation learning

How models learn useful features from data with no labels.

KEY POINTS

- Self-supervision is why models can train on billions of images and hours of video.
- Choose the encoder by what the features must capture: meaning in words, or geometry and detail.
- Good embeddings turn hard problems into nearest-neighbour lookups.

Run it: Linear probes: what do frozen features know? (github.com/wnukmat/robot-learning-notebooks)

Three recipes make labels from the data itself

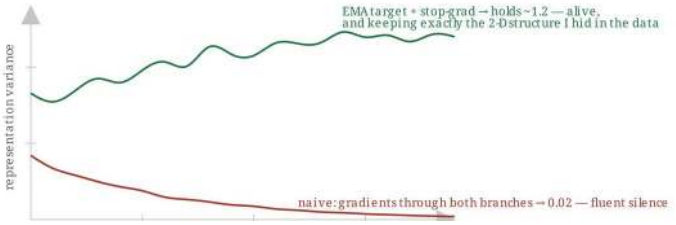
self-supervised learning invents a prediction problem whose answer is already in the data



Masked modeling: one visible context block and several hidden target blocks to predict.



Contrastive: eight images, eight captions. The diagonal is the truth; every other cell is rejected.



Self-distillation, toy run: with a slow teacher the features stay varied; without it they collapse.

Masked and contrastive. Predict hidden words or patches, or make the embeddings of things that belong together match and push the rest apart.

Self-distillation. A student predicts a slowly updated copy of itself that sees a different view. It gives the DINO family's features.

With no labels to pay for, models can train on billions of images and hours of video

Masked objectives: rebuild pixels or predict features

x is an image, m a set of hidden patches and f the encoder; each family is a different loss on the same data

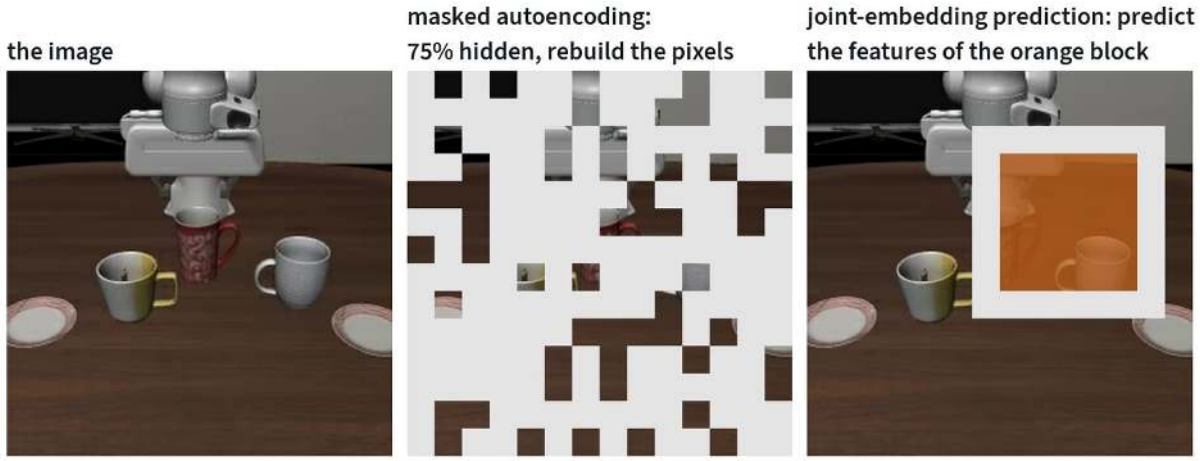
Two targets

$$\mathcal{L}_{\text{MAE}} = \frac{1}{|m|} \sum_{i \in m} \left\| g\left(f_\theta(x_{\setminus m})\right)_i - x_i \right\|^2$$

Masked autoencoding: a small decoder g rebuilds the hidden pixels from the visible patches

$$\mathcal{L}_{\text{JEPA}} = \frac{1}{|m|} \sum_{i \in m} \left\| h\left(f_\theta(x_{\setminus m})\right)_i - \bar{f}(x)_i \right\|^2$$

Joint-embedding prediction: a predictor h outputs the target encoder's features



One real frame. Masked autoencoding hides three quarters of the patches and rebuilds their pixels. Joint-embedding prediction hides a block and predicts its features.

The two losses have the same form. The target is pixels in one and features in the other

Self-distillation, and what stops it collapsing

the DINO family: a student and a teacher see two different crops of one image

Student and teacher

$$\mathcal{L}_{\text{DINO}} = -\sum_{k} p^{(k)}_{\text{teacher}}(x_1)\ \log p^{(k)}_{\text{student}}(x_2)$$

The student is trained to match the teacher's output distribution on another crop

$$\theta_{\text{teacher}} \leftarrow \tau\,\theta_{\text{teacher}} + (1-\tau)\,\theta_{\text{student}}$$

The teacher is an exponential moving average of the student

In words

The gap in the loss. Nothing in it stops both networks outputting the same thing for every image.

Centring. A running mean is subtracted from the teacher's outputs.

Sharpening. The teacher's outputs pass through a low temperature.

Slow teacher. Its weights trail the student's, which is what a momentum encoder is.

Centring, sharpening and a slow-moving teacher together keep the representation from collapsing

Vision encoders: the eyes of every modern system

almost all are Vision Transformers; how each was trained decides what its features are good at

Encoder family	Trained by	Features are good at	Typical use in robotics
CLIP and SigLIP	Matching images with captions	Meaning that lines up with language	Vision-language models, open-vocabulary recognition
DINO family	Self-distillation on images only	Fine spatial detail, parts, correspondence	Manipulation features, tracking, depth
Masked autoencoders	Reconstructing hidden patches	Low-level structure	Pretraining on unlabelled domain data
Video encoders	Masked or predictive training on clips	Motion and change over time	Dynamics, world models

Choose the encoder by what the features must capture: meaning in words, or geometry and detail

A language-aligned encoder classifies with no task training

three photos labelled mug, basketball and dog by comparing each image with text prompts



→ "mug"



→ "basketball"



→ "dog"

How it works. Embed the image, embed each candidate label as text, and pick the label whose embedding is closest.

Why it works. The encoder was trained to match images with captions, so its features line up with language.

What it lacks. A language-aligned encoder can be weak on fine spatial detail, which is where DINO-style features are strong.

In robot models. Many feed both encoders at once: one for meaning and one for spatial precision.

Zero-shot recognition is a comparison of two embeddings, with no training examples for the task

A representation is judged by what a linear layer can read

measured for this note on 1,516 robot frames, whole episodes held out, mean of eight runs

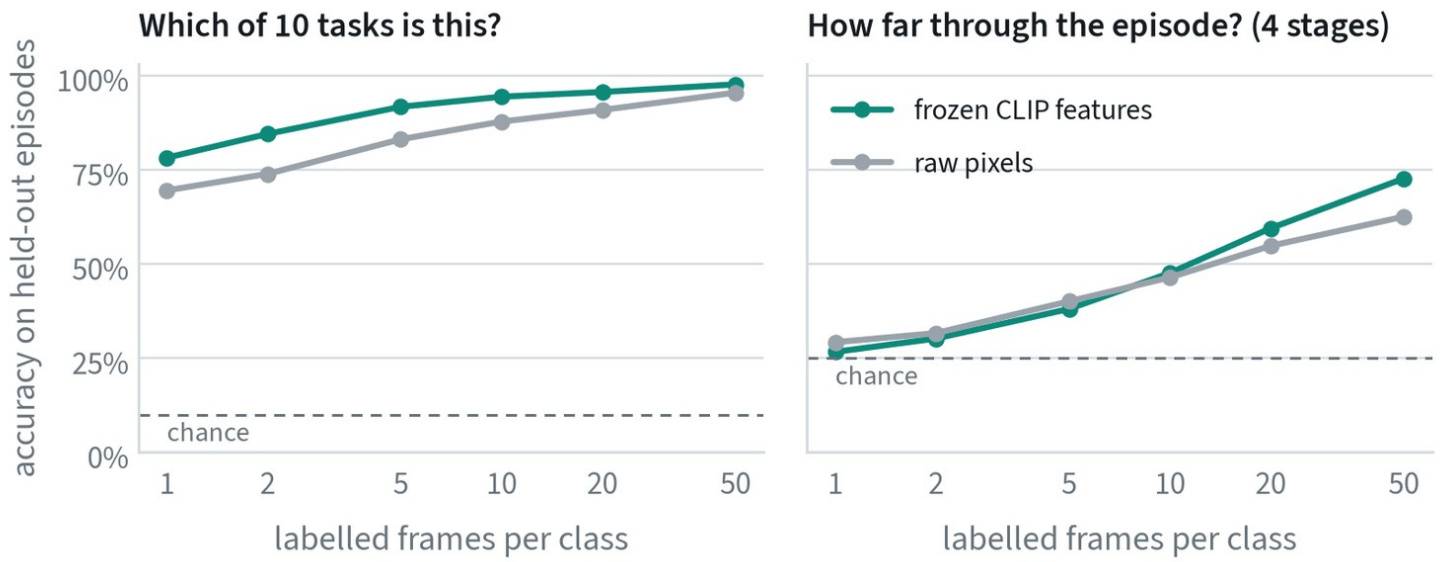
Linear probe

The test. Freeze the encoder, train one linear layer on its features, and report the accuracy.

Which of 10 tasks. With one labelled frame per task, frozen CLIP scores 78% and raw pixels 70%. They reach 98% and 95%.

Which of four stages. Both start near chance. CLIP pulls ahead only at twenty labels, reaching 73% against 63% at 50.

Reading it. CLIP knows what objects are in the scene. It was never trained to notice how far a gripper has closed.



Good features is always relative to a question, and the pixel baseline shows how easy the question is

Four protocols for judging a representation

ordered by how much is trained on top of the frozen encoder

Protocol	What is trained	What it measures
k-nearest neighbours	Nothing	Whether similar inputs are already close together
Linear probe	One linear layer	Whether the information is linearly readable
Attentive probe	A small attention layer over patch features	Whether it is present in the patches, if not in one summary vector
Fine-tuning	Everything	How good a starting point the weights are

Each protocol trains more on top of the encoder and answers a different question about it

Predict in feature space, then search by distance

a joint-embedding predictive architecture, or JEPA, and what a good embedding space is used for

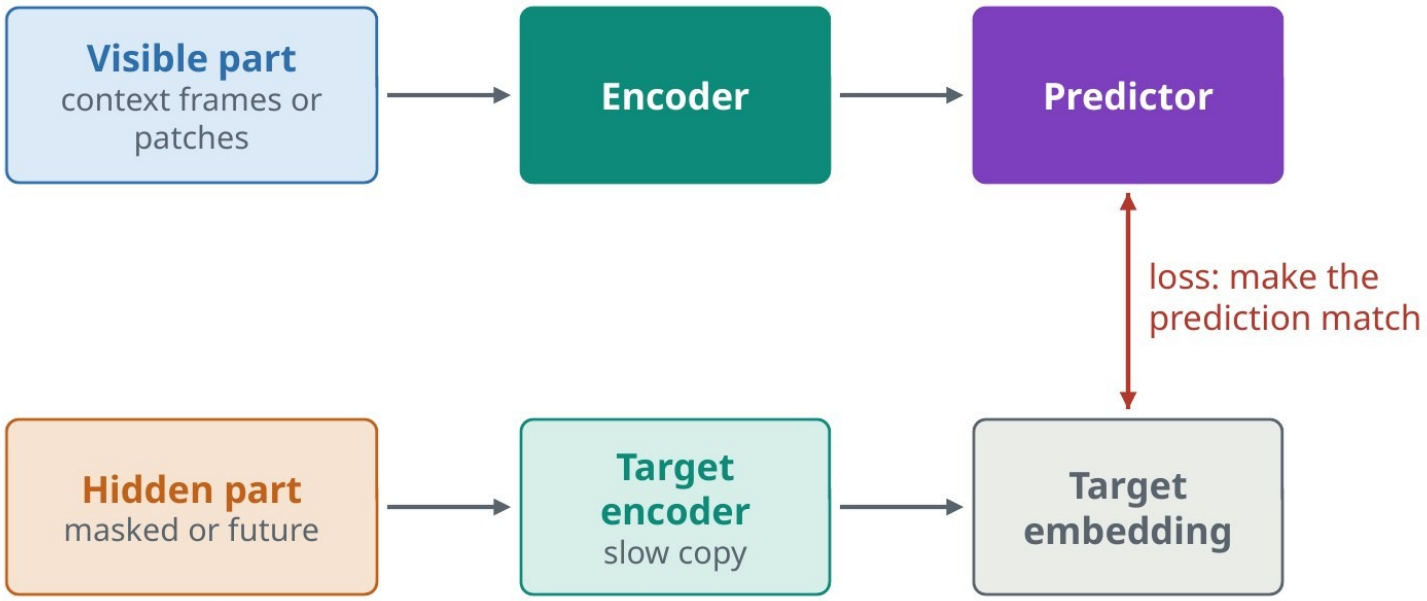
JEPA and retrieval

The problem with pixels. A model forced to predict every pixel spends capacity on texture and sensor noise that cannot be predicted.

The diagram. Visible input goes through an encoder and a predictor. Hidden input goes through a slow target encoder. The loss compares embeddings.

Hidden or future. The same design predicts the representation of a masked part or of what comes next.

Embeddings and retrieval. Once similar things land close together, search becomes finding the closest vectors.



No pixels are reconstructed. The loss compares two embeddings.

What matters for a decision is where the cup will be, not the grain of the table under it

Common mistakes with representations

three errors in choosing, trusting and training an encoder

Using one encoder for everything

- A language-aligned encoder can be weak on fine spatial detail, and the reverse.
- Choose by what the features must capture.

Assuming features transfer

- Features learned on web photos can miss what matters in your domain, such as packaging or radar imagery.

Ignoring collapse

- Self-supervised training can quietly converge to a constant output.
- Monitor the variance of the features.

Match the encoder to the question, test it on your own domain, and watch the feature variance

MODELS · TOPIC 7 OF 32

Contrastive and multimodal learning

Learning by comparison: triplet and batch-wide losses, mining, and how two kinds of data are brought into one space.

KEY POINTS

- A contrastive loss needs only to know what belongs together.
- Triplet losses depend on mining; batch-wide losses use every negative for free.
- A single shared vector is powerful for retrieval and too blunt for detailed instructions.

Run it: Contrastive losses: triplet, hard mining, and InfoNCE (github.com/wnukmat/robot-learning-notebooks)

Learning by comparison

a contrastive loss is told only which things belong together, and shapes an embedding space around that

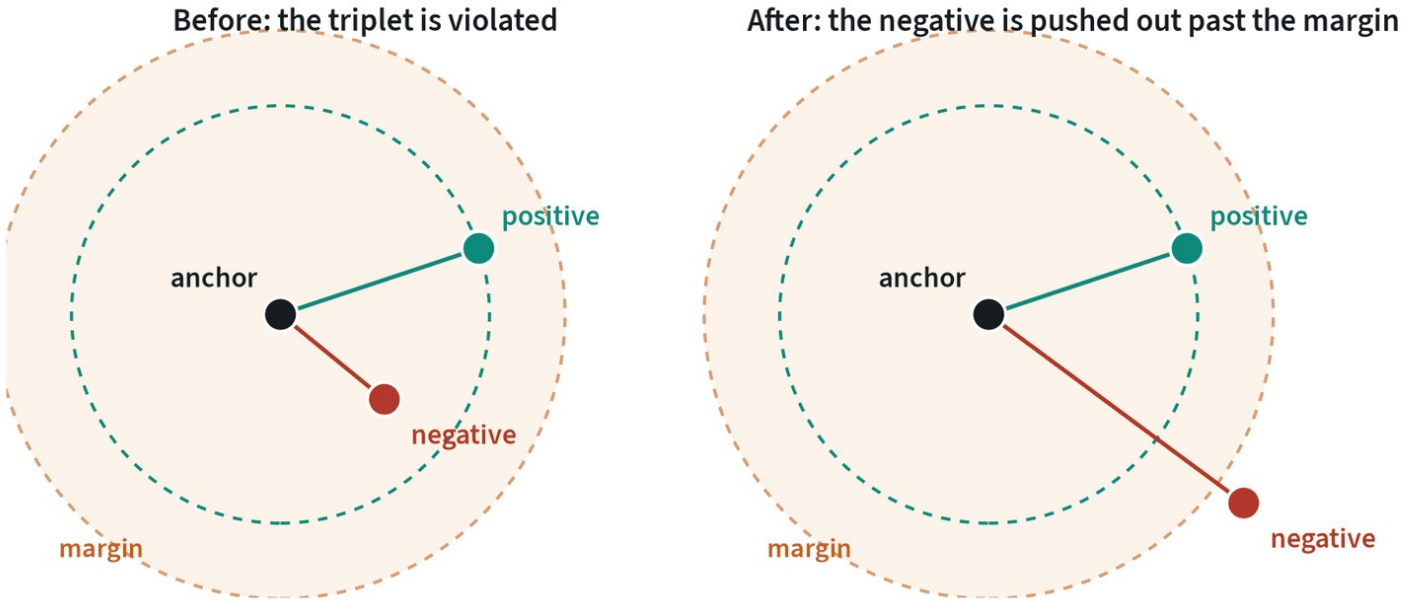
No answer needed

The goal. Things that belong together land close in the embedding space. Everything else lands far apart.

Cheap pairs. Two photos of one product, an image and its caption, a camera frame and the touch reading from the same instant.

Multimodal learning is the second and third case: teaching a model to relate different kinds of data.

A triplet. Training moves the points until the negative is further from the anchor than the positive, by at least the margin.



A contrastive loss needs only to know what belongs together, and that is cheap to obtain

Triplet loss and InfoNCE

three items at a time with a margin, or one positive against the whole batch as a classification

Two losses

$$L = \max\left(0,\ d(a,p) - d(a,n) + \text{margin}\right)$$

Triplet: zero once the negative n is further from anchor a than positive p, by the margin

$$L = -\log \frac{\exp(s_{ii}/\tau)}{\sum_j \exp(s_{ij}/\tau)}$$

InfoNCE: cross entropy over N candidates, with the true partner as the correct class

In words

The zero is a weakness. Most randomly chosen triplets already satisfy the margin and teach nothing.

Mining is the search for the triplets that do not. Triplet training depends on it.

Free negatives. In InfoNCE every other item in the batch is a negative, which is why these models train with batches of thousands.

Temperature τ. A low value concentrates the loss almost only on the hardest, most confusing negatives.

Triplet losses depend on mining; batch-wide losses use every negative for free

Five contrastive losses compared

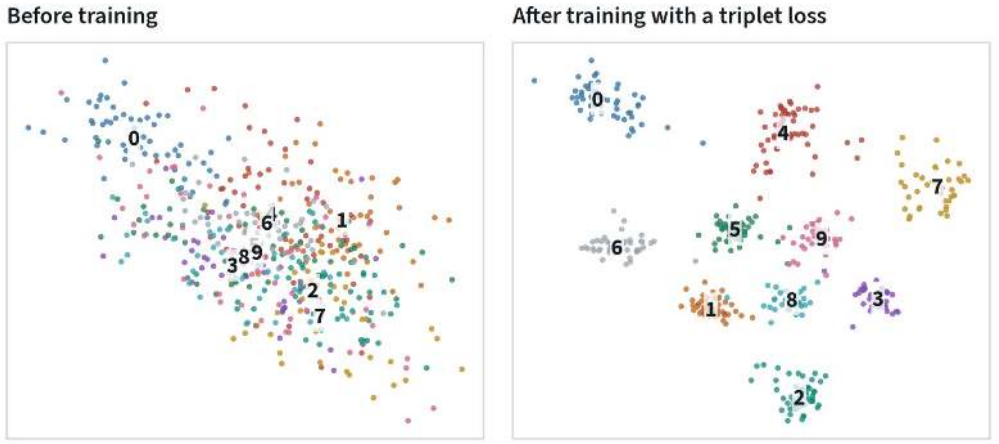
what each one compares, and what that buys and costs

Loss	Compares	Strength	Weakness
Contrastive (pairwise)	Two items: pull together if matched, push apart up to a margin if not	Simplest	Uses little of each batch
Triplet	Anchor, positive, negative	Directly states the ranking wanted	Most triplets are uninformative; needs mining
InfoNCE	One positive against every other item in the batch, as a classification	Uses every negative in the batch; no mining	Wants large batches; a temperature to tune
Supervised contrastive	As InfoNCE, with every same-class item as a positive	Uses labels when they exist	Needs labels
Sigmoid (SigLIP)	Every pair in the batch, each as its own yes or no	Works at small and very large batch sizes	Needs a bias term and care with the many negatives

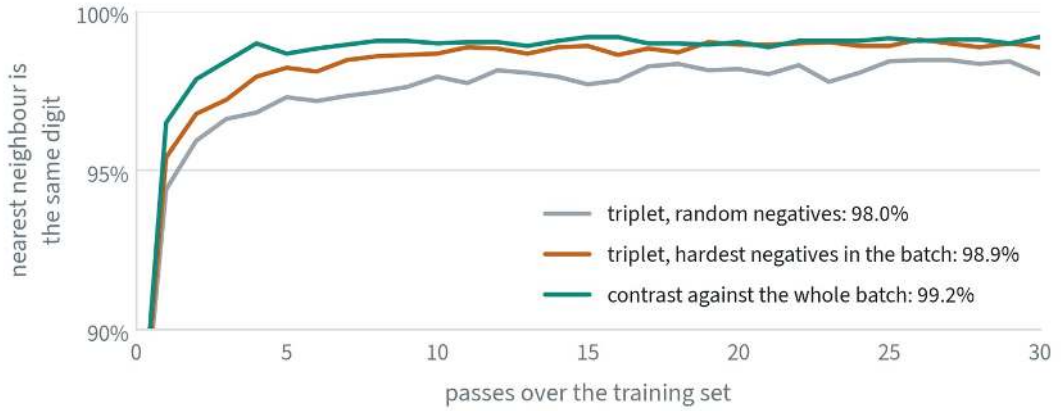
The losses differ mainly in how much of each batch they use as negatives

Three losses on the same data

run for this note on a public set of 1,797 handwritten digits with a small network



Two-dimensional embedding before and after triplet training. Only which images matched was given.



16-dimensional embedding, mean of five runs. Score: nearest neighbour is the same digit. Untrained: 84%.

Hard negatives. Taking the hardest in each batch cut the triplet error roughly in half: 98.0% to 98.9% accuracy, 2.0% to 1.1% error.

Whole batch. Contrasting against every item reached 99.2% and learned fastest, with nothing to mine.

Choosing negatives matters, and contrasting against the whole batch removes the need to choose

What the loss does to the space

with embeddings normalised to unit length, every point lies on a sphere and similarity is the cosine of an angle

Two goals at once

$$\mathcal{L}_{\text{align}} = \mathbb{E}_{(x,x^+)} \left\| f(x) - f(x^+) \right\|^2$$

Alignment pulls matched pairs together

$$\mathcal{L}_{\text{uniform}} = \log \mathbb{E}_{x,y}\, e^{-2\|f(x)-f(y)\|^2}$$

Uniformity spreads all points evenly over the sphere

In words

Uniformity keeps information. Spreading the points evenly preserves as much information as possible.

Inside InfoNCE. The numerator is the alignment term. The denominator, summed over the batch, is the uniformity term.

Collapse explained. With nothing pushing points apart, one point for everything satisfies alignment perfectly.

Alignment alone is minimised by mapping everything to one point; the negatives prevent that collapse

The CLIP and SigLIP objectives

image embeddings u, text embeddings v; the loss is applied in both directions and averaged

Softmax and sigmoid

$$\mathcal{L}_{\text{CLIP}} = -\frac{1}{2N}\sum_{i=1}^{N}\left[\log\frac{e^{u_i^\top v_i/\tau}}{\sum_j e^{u_i^\top v_j/\tau}} + \log\frac{e^{u_i^\top v_i/\tau}}{\sum_j e^{u_j^\top v_i/\tau}}\right]$$

InfoNCE from image to text and from text to image, with a learned temperature τ

$$\mathcal{L}_{\text{SigLIP}} = -\frac{1}{N}\sum_{i}\sum_{j}\log\sigma\left(z_{ij}\left(t\,u_i^\top v_j + b\right)\right)$$

Every pair is its own yes or no: z is +1 on the diagonal and −1 elsewhere

In words

No softmax. No sigmoid term depends on the rest of the batch, so the loss can be computed in pieces across machines.

Batch range. That is why it scales to very large batches and also works at small ones.

On the digits, no effect. Temperature from 0.01 to 1 moved accuracy between 98.5% and 99.1%. Batch 8 to 256: 98.2% to 98.9%.

Regime. A hyperparameter's reputation comes from the regime it was studied in.

Temperature and batch size matter at scale, where hard negatives in a batch limit learning

Two modalities, one space, and where it fails

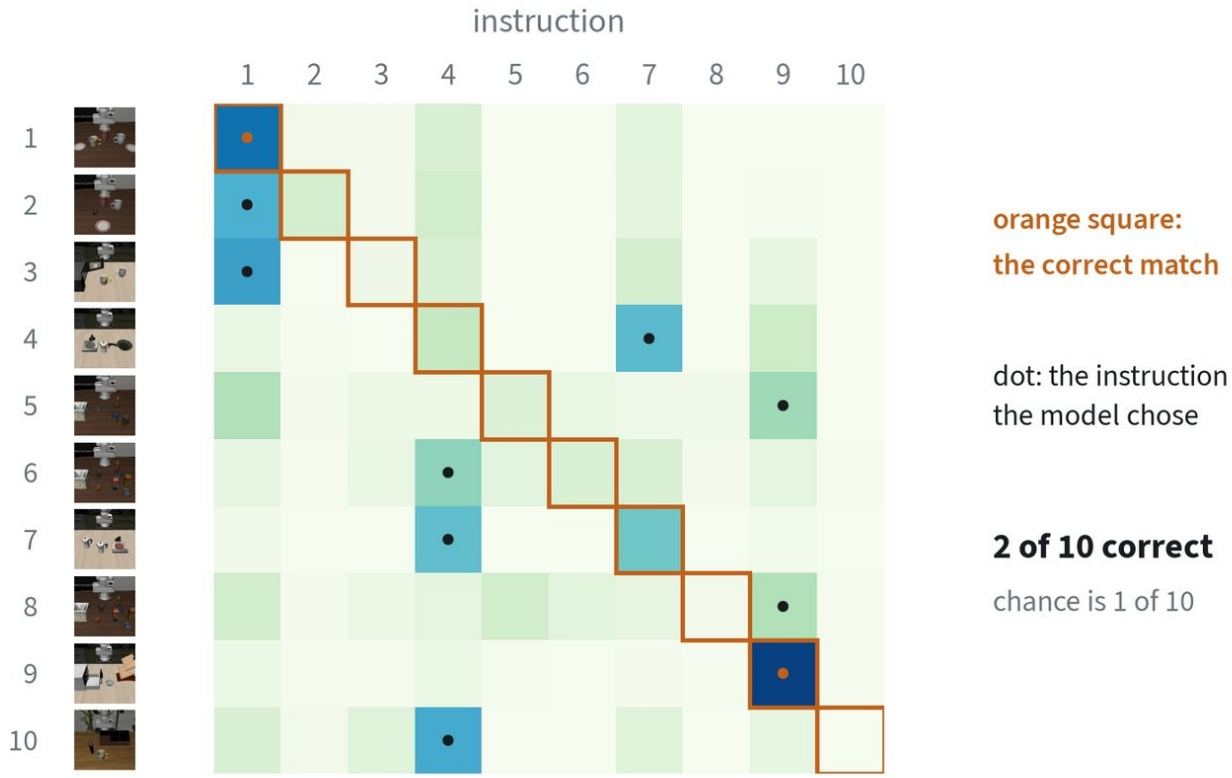
a real test for this note: a public CLIP model, ten LIBERO frames and their ten task instructions

Shared space. CLIP trains an image encoder and a text encoder so an image and its caption land at the same point.

2 of 10. Asked which instruction goes with which frame, it is right twice and gives several frames the same instruction.

Three reasons. Near-identical tabletops that differ in small objects, long compositional instructions, and simulated images.

The consequence. Robot models use the vision encoder's patch features inside a language model instead.



A single shared vector is powerful for retrieval and too blunt for detailed instructions

Beyond images and text

the same recipe works wherever two signals are known to belong together

More senses

- Sound, depth, touch and motion are each aligned with images.
- They become comparable with each other without ever being paired directly.

Time

- Nearby video frames are positives and distant frames are negatives.
- The embedding learns what changes slowly, often the task-relevant state.

Views

- Two cameras on the same moment form a positive pair.
- The features stop depending on viewpoint.

Touch and vision

- A fingertip image is paired with the camera image of the same contact.
- The touch encoder gets a start without scarce labelled touch data.

Anything paired with images becomes comparable with everything else paired with images

How contrastive training goes wrong

four failures that come from what the loss is and is not told

Failure	What happens	What to know
Collapse	Mapping everything to one point satisfies the loss if nothing pushes points apart	Negatives prevent it; methods without negatives need another device
False negatives	Two different images of the same thing in one batch are pushed apart as a mismatch	It worsens as batches grow
Shortcuts	Positives share something trivial, such as the same background or camera	The model learns that and nothing else
The modality gap	Even after training, images and texts tend to occupy separate regions of the shared space	Distances across modalities are meaningful only in relative terms

Check what the positives share and what the negatives contain before trusting the embedding

Common mistakes with contrastive learning

three errors in setting up the loss and in what is asked of the result

Training a triplet loss without mining

- Most triplets contribute nothing after the first few passes.

Leaving the temperature at a default

- It changes which negatives matter and is worth tuning.

Expecting one vector to carry a sentence

- Fine for "a photo of a mug".
- It does not hold a multi-step instruction.

Mine the triplets, tune the temperature, and do not ask one vector to carry a sentence

MODELS · TOPIC 8 OF 32

Retrieval and multimodal retrieval

Finding instead of generating: embeddings, ranking metrics, searching across text and images, large collections, and uses in robotics.

KEY POINTS

- Retrieval turns search into finding the nearest vectors.
- Always score a trivial baseline, and choose the metric by what the results are for.
- A shared embedding space lets text find images, and any sensor find another.

Run it: Retrieval on robot data (github.com/wnukmat/robot-learning-notebooks)

Finding instead of generating

retrieval answers a question by finding the right item in a collection

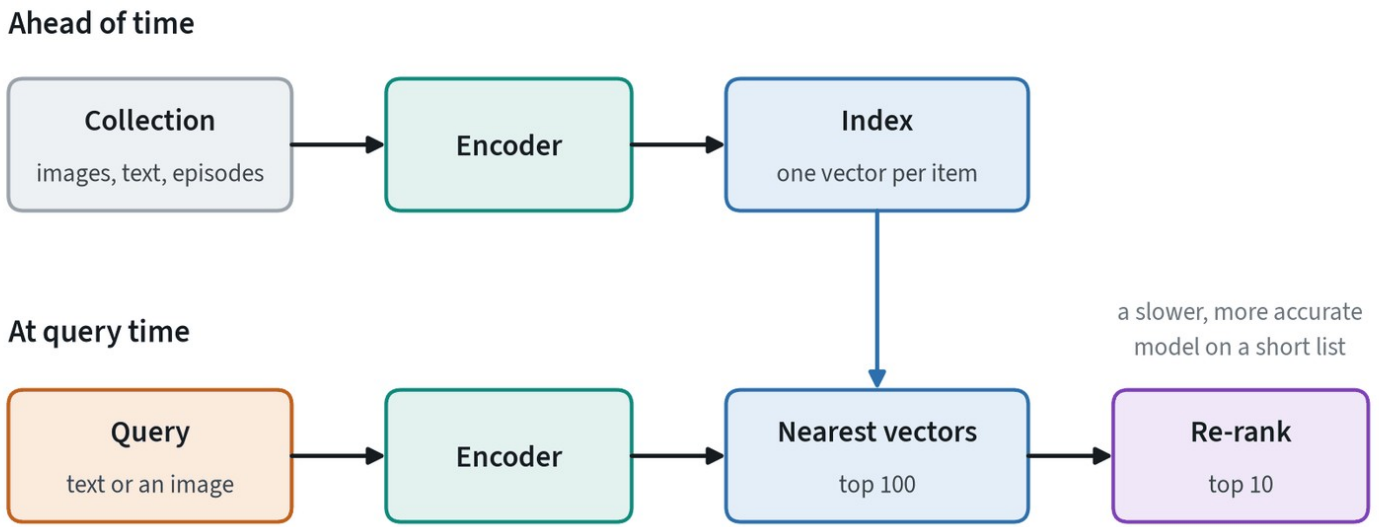
The standard pipeline

Generate or look up. A model that generates carries everything it knows in its weights. A model that retrieves can look things up.

Ahead of time. The collection is encoded once into an index holding one vector per item.

At query time. The query is encoded and the nearest stored vectors are looked up, here giving the top 100.

Re-ranker. A slower model optionally re-orders the short list, here down to the top 10.



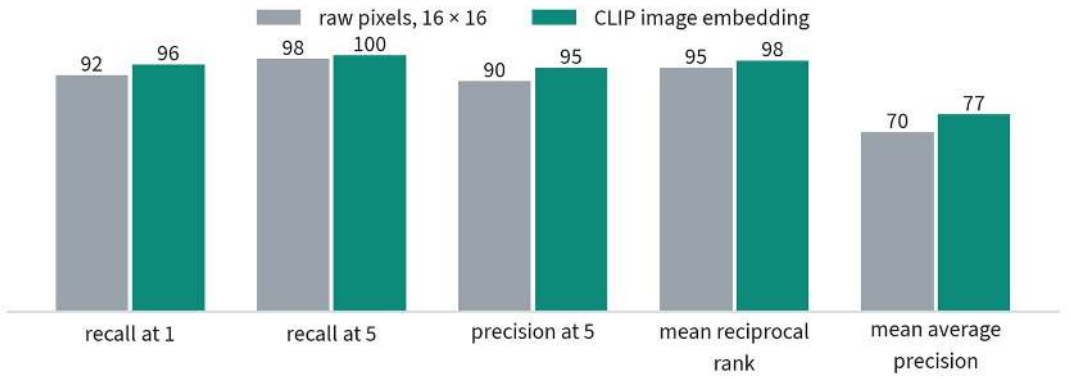
Retrieved knowledge can be inspected, corrected and extended without retraining

A worked example on robot data

1,516 frames, four from each of 379 LIBERO demonstrations, embedded with a public CLIP image encoder



Four queries and their five nearest neighbours from other episodes. Green means the same task.



The same test scored five ways, for CLIP and for each frame shrunk to 16 × 16 grey pixels.

The test. A query is one frame. A retrieved frame is relevant if it comes from a different episode of the same task.

The baseline. Recall at 1 is 96 for CLIP and 92 for raw pixels. Mean average precision is 77 against 70.

CLIP scores 96% and 256 grey pixels score 92%, so the high score says little about the encoder

Scoring a ranking

five metrics, each answering a different question about the same ordered list

Metric	Question it answers	Use it when
Recall at k	Is at least one relevant item in the top k?	One good hit is enough, as in looking up an answer
Precision at k	What share of the top k is relevant?	The user reads the whole list
Mean reciprocal rank	How high is the first relevant item? Scores 1, ½, ⅓ for ranks 1, 2, 3	Only the first hit matters and its position counts
Mean average precision	Are all the relevant items near the top?	There are many relevant items and all should be found
nDCG	As above, with graded relevance and a discount for lower ranks	Some results are better than others

Recall at 5 is 100% for CLIP while mean average precision is 77%: choose by what the results are for

The metrics, written out

for one query: rel is 1 if the item at rank k is relevant, and R is the number of relevant items in the collection

Per query

$$\mathrm{P@}k = \frac{1}{k}\sum_{i=1}^{k}\mathrm{rel}_i \qquad \mathrm{AP} = \frac{1}{R}\sum_{k}\mathrm{P@}k \cdot \mathrm{rel}_k$$

Precision at k, and average precision: precision taken at each relevant rank

$$\mathrm{RR} = \frac{1}{\text{rank of the first relevant item}}$$

Reciprocal rank looks only at the first relevant item

$$\mathrm{DCG@}k = \sum_{i=1}^{k}\frac{2^{g_i}-1}{\log_2(i+1)} \qquad \mathrm{nDCG@}k = \frac{\mathrm{DCG@}k}{\text{ideal DCG@}k}$$

Discounted gain, divided by the gain of the ideal ordering

In words

Means. Mean average precision and mean reciprocal rank are the averages of AP and RR over all queries.

Graded relevance. In nDCG, g is a relevance score, so a perfect match can count for more than a partial one.

Discount. The logarithm reduces the credit for items lower in the list.

Every ranking metric is a different weighting of the same list of relevant and irrelevant positions

Multimodal retrieval: text finds images

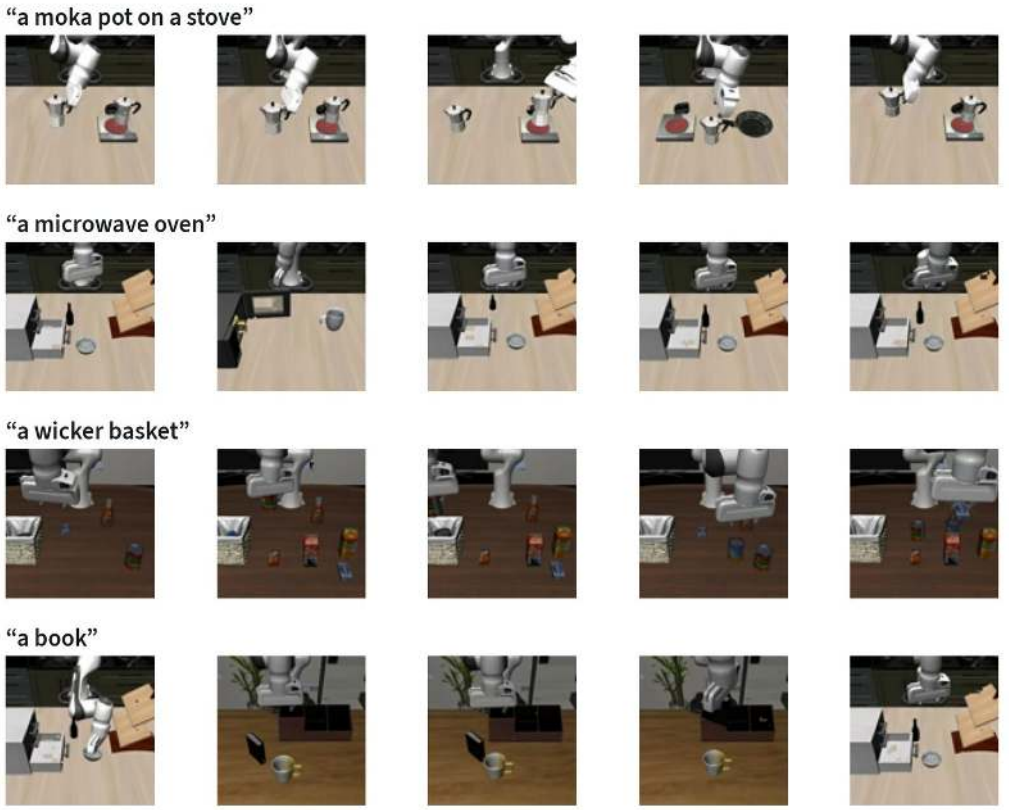
text queries against the same frames through CLIP's text encoder, top five from distinct episodes

Why it works. CLIP's text and image encoders share an embedding space, so a query of one kind retrieves items of the other.

Five of five. "A moka pot on a stove" and "a wicker basket" return the right scene every time.

One of five. "A microwave oven" returns the microwave once and a white cabinet four times.

Two of five. "A book" finds the book twice.



A shared embedding space lets a sentence search frames that were never labelled with those words

Kinds of multimodal retrieval

what the query is, what comes back, and an example of each

Kind	Query	Returns	Example
Text to image	A description	Images or video	Find every episode that contains a drawer
Image to image	A picture	Similar pictures	Find past scenes like the one in front of the robot
Image to text	A picture	Captions or documents	Find the manual page for this part
Composed	A picture and a modification in words	Images	Like this scene, with the door open
Cross-sensor	A reading from one sensor	Data from another	Find the camera frames that match this touch pattern

With a shared space, any sensor can find another

Searching large collections

measured for this note: exact search on a desktop processor, and a 32-cluster index on the robot frames

Exact and approximate

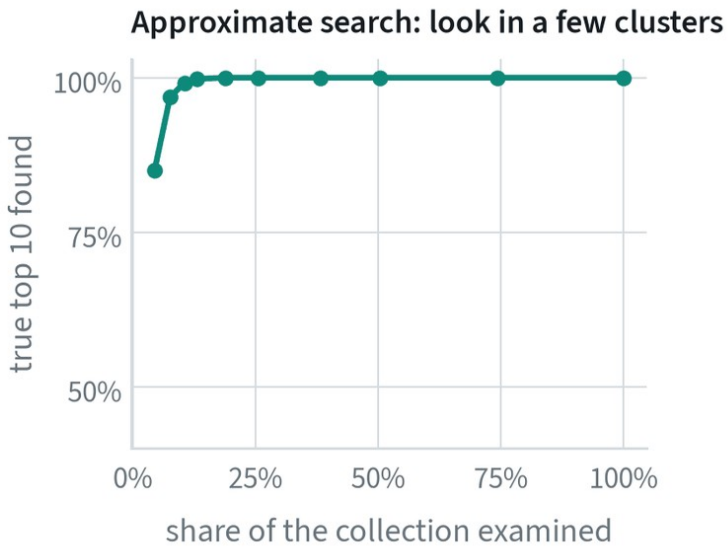
Exact search compares the query with every 512-number vector: 3 ms at 10,000 items, 9 ms at 100,000, 86 ms at a million.

Clustering (inverted file). Search only the clusters nearest the query. Examining 8% found 97% of the true top ten, and 11% found 99%.

Graphs (HNSW). Link each vector to its neighbours and walk toward the query. Fast and accurate, at the cost of memory for the links.

Product quantisation. Store each vector as a short code so far more fit in memory, with some loss of precision.





Below about 100,000 items exact search is usually fast enough and has nothing to tune

Compressing the vectors

measured on the robot-frame collection; queries stay exact and only the stored vectors are compressed

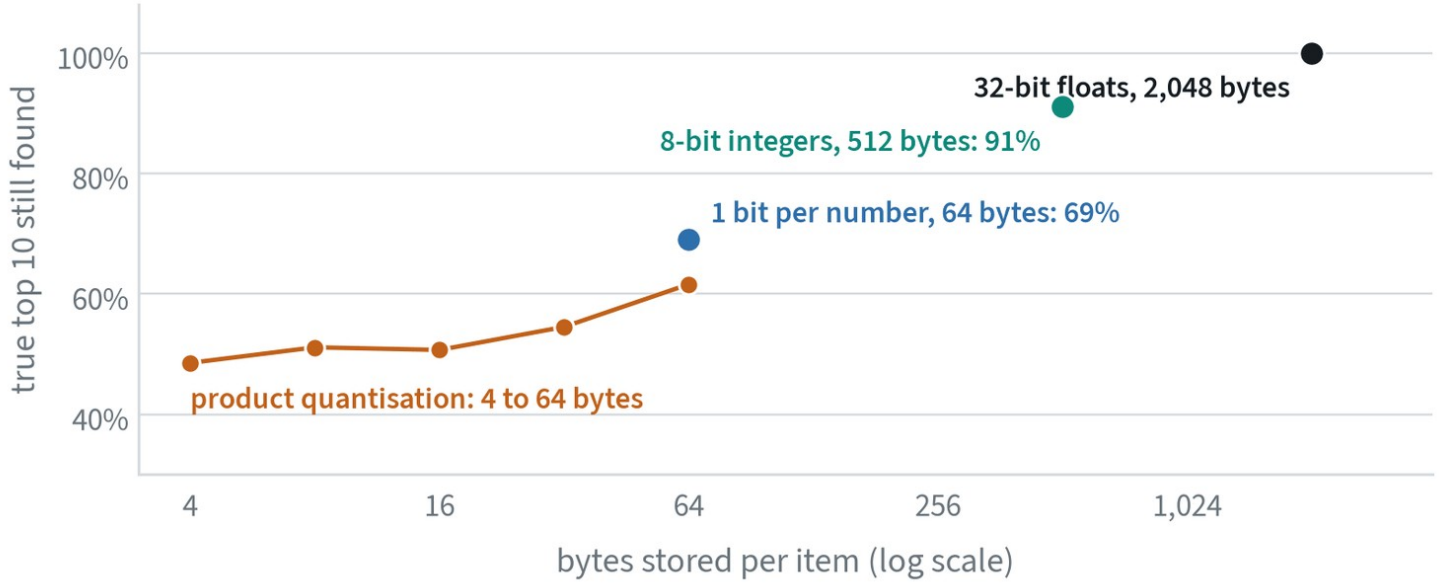
Bytes per item

The cost. A million 512-number embeddings as 32-bit floats take two gigabytes, 2,048 bytes per item.

Scalar quantisation. 8 bits per number: a quarter of the memory for 91% of the true top ten, with no training.

Binary codes. Keep only the sign: a thirty-second of the memory and 69%. Usually used to find candidates to re-score.

Product quantisation. m sub-vectors, each replaced by an index into a codebook of 256: 62% at 64 bytes, 49% at 4 bytes.



Memory, not arithmetic, limits large indexes: search compressed vectors, then re-score with exact ones

Beyond one vector, and retrieval in robotics

three ways to fix the mistakes of single-vector matching, and six places robots use retrieval

Re-ranking

- A slower model reads the query and each candidate together.
- Its cost applies only to the short list.

Late interaction

- One vector per token or patch, matched in detail at query time.
- More accurate on long, specific queries. Stores many times more data.

Hybrid search

- Embeddings find paraphrases.
- Keywords find exact names and part numbers.
- Each covers the other's blind spot.

Robots: curating data

- Retrieve episodes similar to a few demonstrations of a new task and train on both.
- Find and remove near-identical episodes.

Robots: checking coverage

- Retrieve the training episodes nearest a failure.
- If nothing is close, the policy was never shown that situation.

Robots: memory, facts, items

- Store what was seen, with where and when.
- Put retrieved documents in a language model's prompt.
- Match a camera view against a catalogue.

Retrieve a generous short list cheaply, then spend the expensive model only on that list

Common mistakes with retrieval

four errors in how a retrieval system is tested and built

Reporting a score with no baseline

- On an easy collection a trivial method scores highly too.

Letting the query retrieve itself

- Exclude the query, and near-copies such as frames of the same episode, from the results.

Choosing one metric by habit

- Recall at 1 and mean average precision can tell different stories about the same system.

Building an index too early

- Exact search is simpler and exact.
- Measure before replacing it.

Score a trivial baseline, exclude the query and its near-copies, and measure before adding an index

Generative modeling

Autoregression, diffusion and flow matching, and the latent spaces they work in.

KEY POINTS

- Autoregressive suits discrete sequences; diffusion and flow suit images, video and actions.
- The model never sees how to draw a whole sample; it only learns one small denoising step, applied repeatedly.
- Flow matching keeps diffusion's regression trick and removes the noise schedule.
- The quality ceiling of a latent generator is set by its autoencoder.

Run it: Flow matching against diffusion, on a problem small enough to see (github.com/wnukmat/robot-learning-notebooks)

Two ways to generate

a generative model learns a data distribution so that it can produce new samples from it

	Autoregressive	Diffusion and flow matching
How it generates	One piece at a time: predict the next token, append it, repeat	All at once: start from noise and refine the whole sample over several steps
Training loss	Cross entropy on the next token	Regression: predict the noise, or the direction toward clean data
Strengths	Simple, handles variable length	Sharp continuous outputs; represents several valid answers
Limits	One step per token; continuous values must be binned first	Needs several refinement steps per sample
Used for	Language, reasoning, tokenised actions	Images, video, robot action chunks

Autoregressive suits discrete sequences; diffusion and flow suit images, video and actions

Diffusion and flow matching, step by step

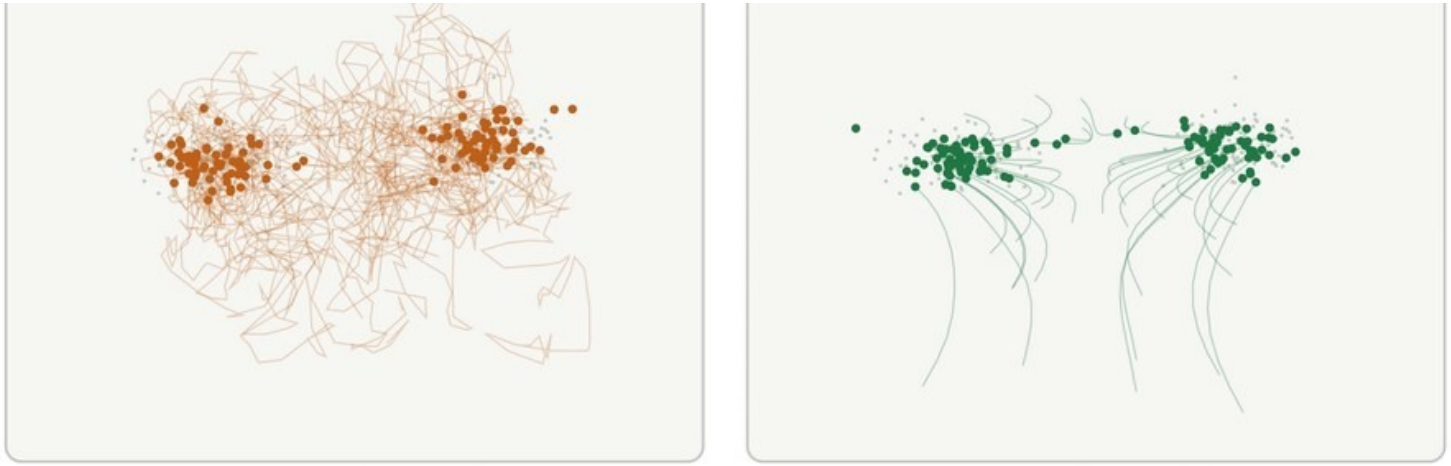
a real toy model: one small network trained two ways on 2-D data with two clusters

Training. Mix a real sample with noise at a random level. The network predicts the noise, or the velocity from noise toward data.

Sampling. Start from pure noise and step along the predicted direction.

Conditioning. Feed the network text, images or robot state, and the same noise becomes a sample that fits the condition.

The paths. Diffusion wanders over 50 steps because fresh noise is injected at each. Flow matching runs nearly straight in 10.



The model never learns to draw a whole sample; it learns one small denoising step, applied repeatedly

The mathematics: autoregression and diffusion

every generative model in use is fitted to maximise, or to bound, the probability of the data

Likelihood and noise

$$\log p_\theta(x) = \sum_t \log p_\theta\left(x_t \mid x_{<t}\right)$$

Autoregressive: the log-likelihood is a sum over tokens, each given the earlier ones

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon \qquad \epsilon \sim \mathcal{N}(0, I)$$

Diffusion forward process: any noise step can be reached directly from the data

$$\mathcal{L}_{\text{diffusion}} = \mathbb{E}_{x_0,\, \epsilon,\, t} \left|\left| \epsilon - \epsilon_\theta\left(x_t,\, t\right) \right|\right|^2$$

The network is trained to recover the noise that was added

In words

Autoregressive. The likelihood is exact, training is one parallel pass, and sampling takes one model call per token.

Schedule. The forward process adds Gaussian noise over T steps. ᾱ falls from 1 to nearly 0 as t grows.

A bound in disguise. The simple loss is a re-weighted form of a bound on the log-likelihood.

Score. Predicting the noise is equivalent, up to scale, to predicting the gradient of the log-density of the noisy data.

Diffusion trains by plain regression onto the noise that a fixed forward process added

Flow matching: a straight path and its velocity

x_0 is a noise sample and x_1 a data sample; the path between them is chosen, and the network learns its velocity

Path, loss, sampler

$$x_t = (1 - t)\, x_0 + t\, x_1$$

A straight path from noise to data

$$\mathcal{L}_{\text{flow}} = \mathbb{E}_{x_0,\, x_1,\, t} \left|\left| v_\theta(x_t,\, t) - (x_1 - x_0) \right|\right|^2$$

Regress the network onto the path's constant velocity, $x_1 - x_0$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v_\theta(x, t) \qquad x \leftarrow x + v_\theta(x, t)\, \Delta t$$

Sampling integrates an ODE from t = 0 to 1, in the simplest case with Euler steps

In words

One fewer thing to tune. The velocity of a straight path is constant, so there is no noise schedule.

An average. The learned velocity averages over all the straight paths through a point, so real trajectories curve a little.

Steps. The straighter the trajectories, the fewer sampling steps are needed.

Flow matching keeps diffusion's regression trick and removes the noise schedule

Flow matching against diffusion, measured

trained for this note: the same small network learns a two-crescent dataset with each loss

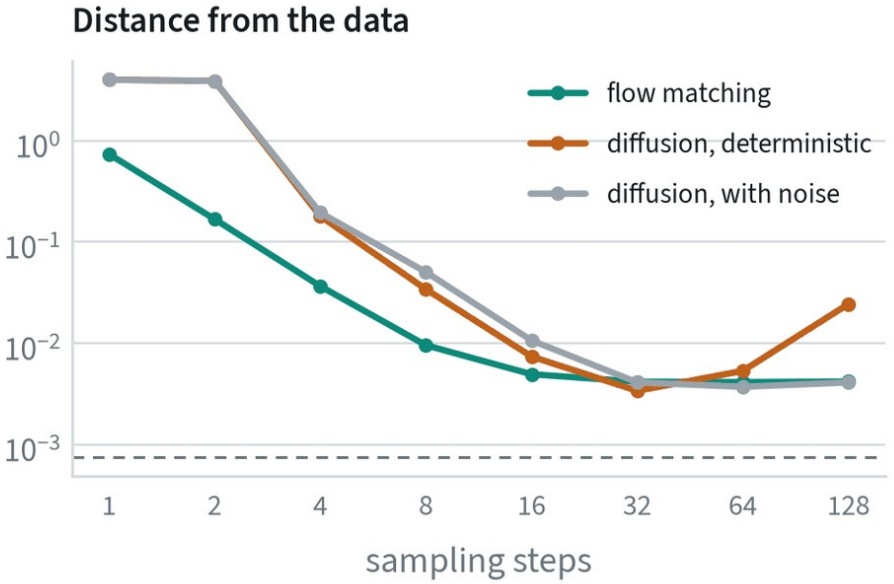
A fair comparison. Same architecture, same data, same number of training steps. Only the loss differs.

Few steps. At four steps flow matching is five times closer to the data. At one or two, diffusion produces nothing usable.

Why. The sixty trajectories on the left are nearly straight, so a few large steps follow them well.

More is not always better. The deterministic diffusion sampler gets worse again beyond 32 steps here.

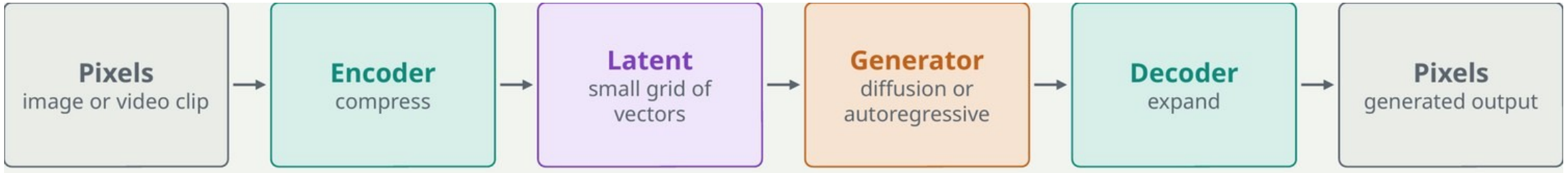




With few sampling steps flow matching is far closer to the data; with 32 or more the two are equal

Latent spaces: generate in a compressed world

modern generators work on a compact code and decode it to pixels at the end



Pixels are expensive. A short 480p video clip is tens of millions of numbers.

Compression. An autoencoder often shrinks it eight times in each spatial direction and four times in time.

The ceiling. If the autoencoder cannot reconstruct small objects, thin structures or fast motion, the generator cannot produce them.

For robots. Grippers and contact points are exactly that kind of detail. Reconstruct real data first and look at it.

The quality ceiling of a latent generator is set by its autoencoder

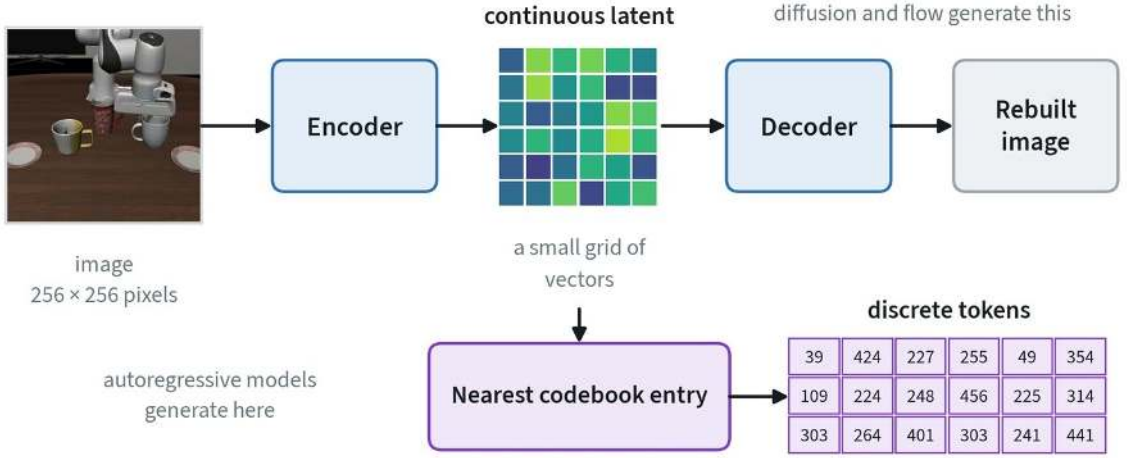
Autoencoders and tokenisers

an encoder squeezes the input into a small code and a decoder rebuilds it, trained together

The VAE bound

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] - \mathrm{KL}\big(q_\phi(z \mid x) \,\|\, p(z)\big)$$

The first term rewards reconstruction; the second keeps the codes close to a simple prior



One encoder, two kinds of code. The upper path keeps continuous vectors. The lower path replaces each vector with the number of its nearest codebook entry.

One encoder gives continuous latents for diffusion and flow, or discrete tokens for a language model

Making sampling controllable and fast

three techniques applied on top of a trained diffusion or flow model

Technique	What it does	What it costs or buys
Guidance	Run the model with and without the condition and exaggerate the difference	Samples follow the prompt more closely, at the cost of variety
Few-step sampling	Flow matching and better solvers cut fifty steps to under ten	Each step is a full network pass. Robot action heads commonly use four to ten
Distillation	Train a student to match many teacher steps in one	A little quality traded for a large speed-up

An action generator that needs fifty passes cannot run at 10 Hz. One that needs four can

Video generation, and scoring generated samples

video adds a choice that images do not have; neither has a single right answer to compare against

Whole clip at once

- Denoise all frames together.
- Consistent from start to end, with a fixed length.

Frame by frame

- Each frame is conditioned on the ones before.
- Any length, and new input can arrive mid-stream. Errors build up.

A noise level per frame

- Recent models combine the two.
- Frames already produced are clean and frames still to come are noisy.

FID and FVD

- Compare features of many real and many generated samples.
- Lower is better. They need thousands of samples to be stable.

FID and FVD measure realism, not correctness: track progress with them and look at the samples

Common mistakes with generative models

three errors in diagnosing and tuning a generator

Skipping the autoencoder check

- Do not judge a generator without checking its autoencoder.
- Blurry hands or grippers are often a reconstruction limit, not a generation failure.

Turning guidance up

- Past a point, outputs become saturated and unnatural.

Counting steps as free

- Every extra step is latency.

Check the autoencoder first, keep guidance moderate, and count every sampling step as latency

Stable Diffusion and latent diffusion

One real image generator taken apart: the autoencoder, the text encoder, the denoiser, guidance and steps, and why robotics builds on it.

KEY POINTS

- Latent diffusion is an autoencoder, a text encoder and a denoiser. Only the denoiser generates.
- The autoencoder sets what can be represented; the step count sets the speed.
- Robotics uses it as a world model, as a policy backbone and as a training recipe for actions.

Three parts, and only the denoiser generates

Stable Diffusion 1.5, the 2022 design called latent diffusion; parameter counts read from the model itself

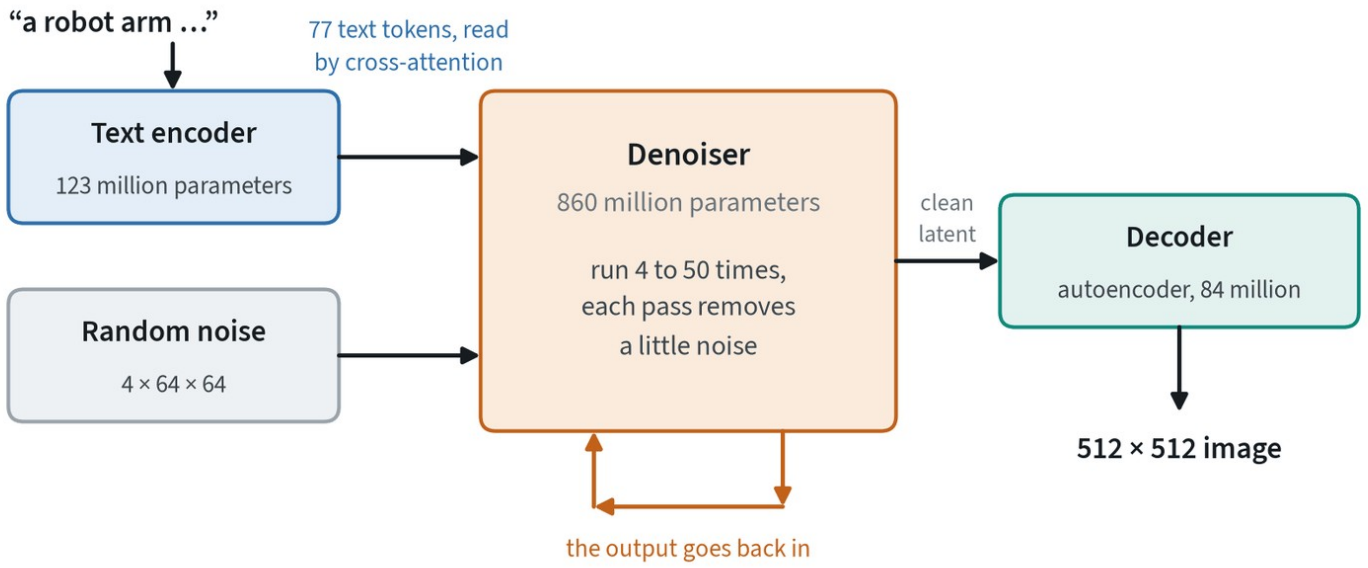
Latent diffusion

Autoencoder, 84 million. Compresses an image into a small grid of numbers, the latent, and rebuilds an image from one. Trained first, then frozen.

Text encoder, 123 million. The text half of a CLIP model, also frozen. It turns the prompt into 77 token vectors.

Denoiser, 860 million. Takes a noisy latent, the text vectors and the noise level, and predicts the noise. The only part trained for generation.

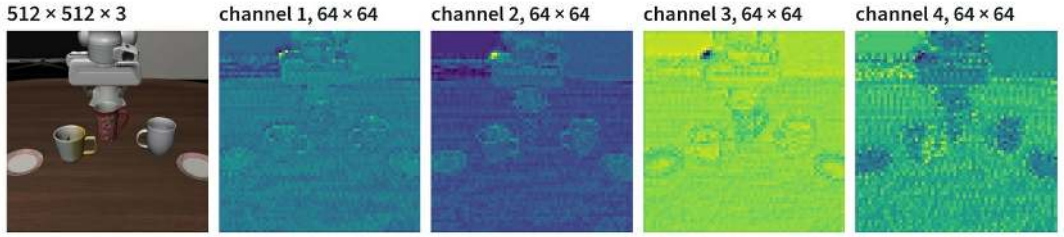
Generation runs the denoiser 4 to 50 times on a 4 × 64 × 64 latent that starts as pure noise, then decodes once to 512 × 512.



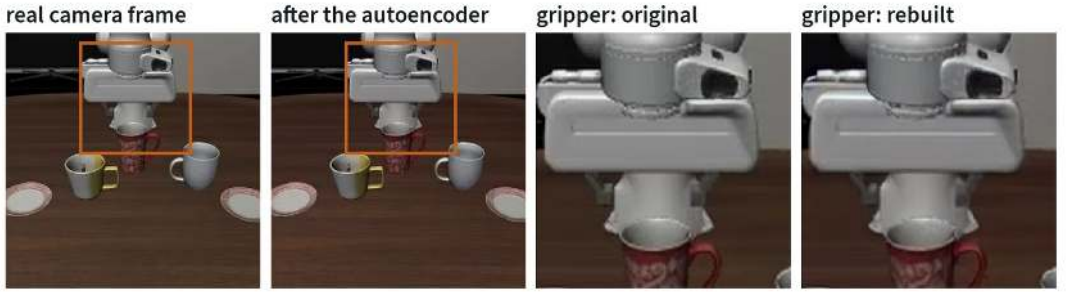
Latent diffusion is a frozen autoencoder, a frozen text encoder and a trained denoiser

Why it works in a latent, and what that costs

the autoencoder shrinks each side of the image by eight and keeps four channels, a 48 times reduction in data



A real robot camera frame and the four 64 × 64 channels of its latent: 786,432 numbers become 16,384



Encoded and decoded with no diffusion: 35.3 dB, with fine edges on the gripper slightly softened

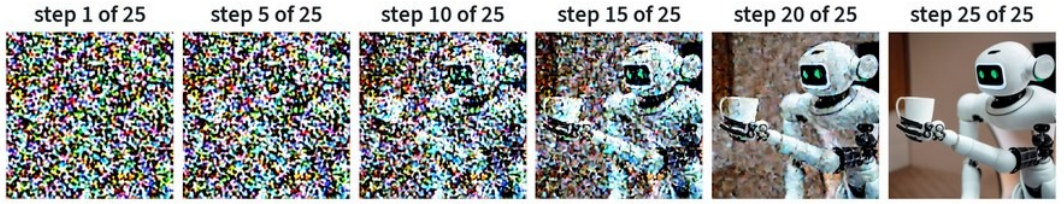
The gain. The denoiser works on a 64 × 64 grid instead of 512 × 512. That is the whole reason the model runs on an ordinary graphics card.

The price. The loss sits in small, sharp detail: edges of fingers, thin parts, small text. Encode and decode real target-domain images first.

Whatever the autoencoder cannot rebuild, the generator can never produce

Generation, one step at a time

at each step the denoiser estimates the noise present and a small rule, the sampler, removes part of it



One generation decoded at steps 1, 5, 10, 15, 20 and 25: noise, a rough shape at 10, a clear robot by 20



The same starting noise with 4, 10, 25 and 50 steps: 0.15, 0.37, 0.91 and 1.83 s on a desktop GPU

Early steps decide composition, late steps add texture and edges. Changing the prompt late alters detail and leaves the layout alone.

Time is proportional to steps. Four give a blurred smear, and 25 and 50 are nearly identical. Few-step quality takes a model trained for it.

Layout is settled in the middle steps and detail in the last; this model needs about 25 steps

The noise schedule never reaches pure noise

1,000 training steps with β rising from 0.00085 to 0.012, read from the model's own scheduler

Mixing and targets

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon$$

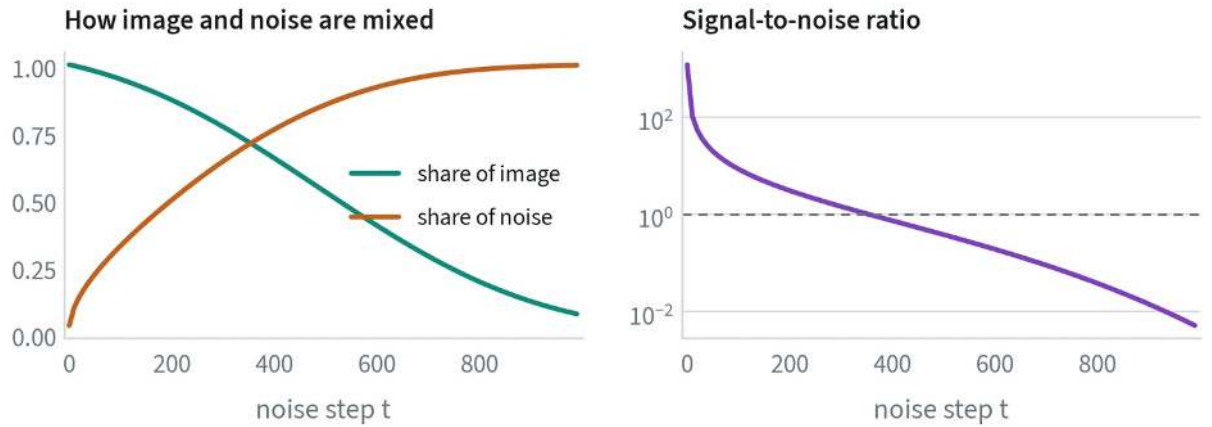
Training mixes the clean latent with noise in these proportions at a random step t

$$\mathrm{SNR}(t) = \frac{\bar{\alpha}_t}{1 - \bar{\alpha}_t}$$

Signal-to-noise ratio: the squared ratio of the two coefficients

$$v = \sqrt{\bar{\alpha}_t}\, \epsilon - \sqrt{1 - \bar{\alpha}_t}\, x_0$$

Velocity target, which behaves well at both ends of the schedule



Left: the image share falls from 1 to 0.07 as the noise share rises, crossing near step 350. Right: signal-to-noise falls from over 100 to below 0.01.

Training never shows the model pure noise, and sampling starts from pure noise

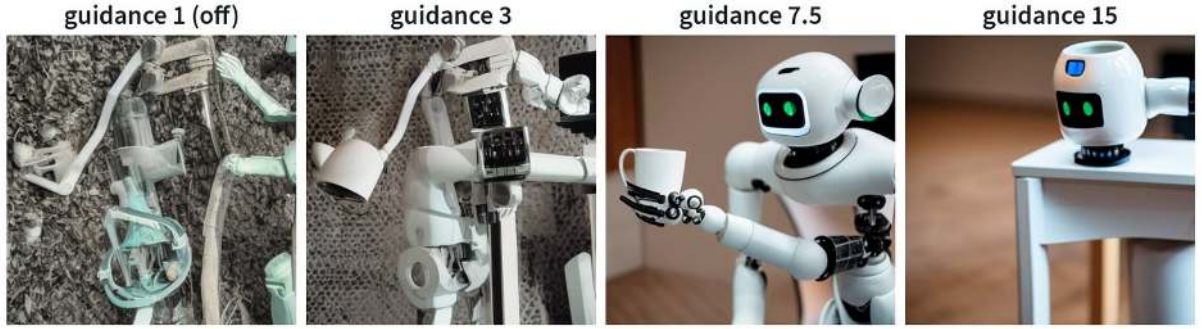
How the prompt gets in, and how guidance enforces it

cross-attention lets every latent position read the 77 text vectors; classifier-free guidance supplies the rest

Classifier-free guidance

$$\hat{\epsilon} = \epsilon_{\text{uncond}} + s\left(\epsilon_{\text{cond}} - \epsilon_{\text{uncond}}\right)$$

Push the prediction further in the direction that separates prompted from unprompted

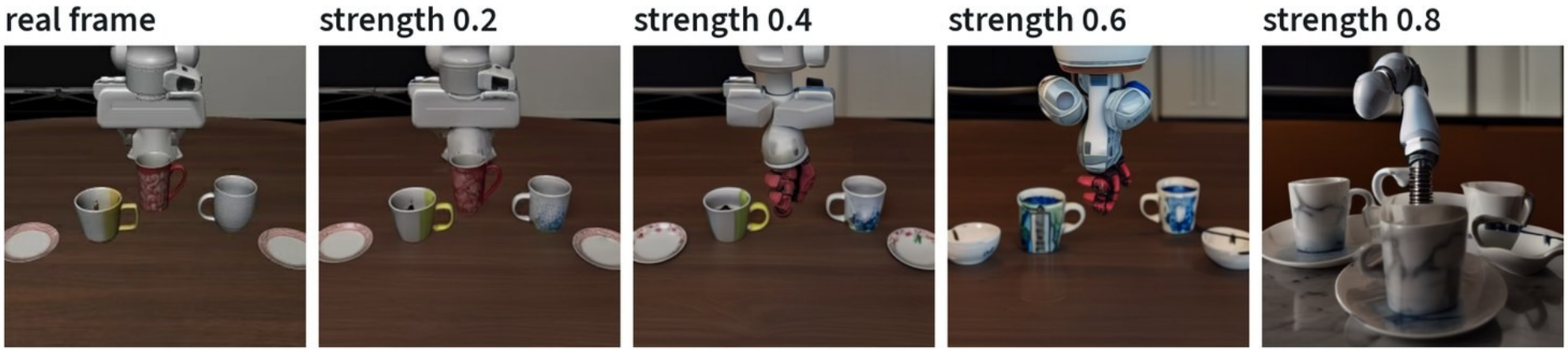


The same starting noise at scales 1, 3, 7.5 and 15: an incoherent tangle, a muddled machine, a clean robot holding a mug, then a simplified, oversaturated image.

Guidance trades variety for obedience and costs a second denoiser pass at every step

Image-to-image: how much of the original survives

a real robot frame is encoded, noised to an intermediate step and denoised with a prompt for a marble kitchen counter



Strength sets how much noise is added before denoising, and so how much of the original image survives.

0.2 and 0.4. The scene is nearly unchanged, then the mugs and plates gain new patterns. Nothing meaningful changes.

0.6 and 0.8. The gripper and objects change shape, then the scene becomes a different image of cups on a marble surface.

For robot data this trade-off makes plain image-to-image hard to use. Finer control is needed.

At high strength the gripper and objects change, so the recorded actions no longer match the picture

Conditioning on more than text

five mechanisms that give finer control, and that generated data for robots depends on

Mechanism	What it adds	Use
Inpainting	A mask: only the masked region is regenerated	Change a background or swap an object and leave the robot untouched
Structural control	An extra network that feeds an edge map, depth map or pose into the denoiser	Keep the geometry of a scene and change its appearance
Low-rank adapters	Small trained matrices added to the attention layers	Teach the model a new object or style from a few images
Image prompts	Features of a reference image passed through cross-attention	Generate variations of a specific object
Negative prompts	The unconditional branch of guidance is replaced with a prompt to avoid	Steer away from unwanted content

Mask out the robot and the manipulated object, then let everything else vary

What changed since

five generations of latent diffusion, from Stable Diffusion 1.x to video models

Generation	Denoiser	Text encoder	Objective	What it added
Stable Diffusion 1.x	Convolutional U-Net with attention	CLIP text encoder	Predict the noise	Latent diffusion, openly released
SDXL	A larger U-Net	Two CLIP text encoders	Predict the noise	Higher resolution and better composition
Diffusion transformers	A transformer over latent patches	Varies	Diffusion or flow	Scales like a language model; no convolutions in the denoiser
Stable Diffusion 3, FLUX	A transformer with joint text and image attention	CLIP plus a large language-model encoder	Flow matching	Better prompt following and rendered text; fewer steps
Video models	A transformer over space and time	A large language-model encoder	Mostly flow matching	An autoencoder that also compresses time

Denoisers became transformers, the objective became flow matching, and the text encoder grew

Why robotics builds on it

a model that generates plausible images has learned a good deal about how the world looks and, for video, moves

As a world model

- A video generator conditioned on the robot's actions predicts what the cameras will see next.

As a policy backbone

- World-action models start from a pretrained video generator and train it to produce actions together with future frames.

As a training recipe

- Diffusion Policy and flow-matching action heads apply the recipe to action chunks, with no image generation involved.

As a source of data

- Image and video generators change backgrounds, lighting and objects in existing demonstrations.

The autoencoder limits what can be shown, the step count limits speed, the training images limit knowledge

Common mistakes with latent diffusion

four errors that follow from forgetting how the model is built

Skipping the autoencoder check

- Encode and decode real frames from your domain before anything else.

Raising guidance to fix everything

- Past a point it damages the image without improving obedience.

Assuming few steps will do

- A model must be trained for few-step sampling.
- Otherwise quality falls off sharply.

Reading generated video as physics

- Plausible is not the same as correct.

Check the autoencoder on your own frames before anything else

Foundation models and adaptation

Pretrain once, adapt many times: the ladder from prompting to fine-tuning.

KEY POINTS

- Almost no one trains from scratch any more; the skill is choosing a starting model and adapting it well.
- Climb only as far as the task needs; each rung costs more data, compute and maintenance.
- Reasoning and tool use are the bridge from chat models to systems that do work, including robots.

Run it: Low-rank adaptation (github.com/wnukmat/robot-learning-notebooks)

Pretrain once, adapt many times

the economics of the field in one line: pretraining is done by a few groups, adaptation by everyone



Pretraining takes enormous data and compute. Adaptation takes a small dataset and a modest budget.

Language models are decoder transformers trained to predict the next token. They carry broad knowledge and follow instructions.

Vision-language models add a vision encoder in front: patches are encoded, projected into token space and placed beside the text.

Transfer cuts both ways. General features carry over and a little task data goes a long way. The model's gaps and biases transfer too.

Almost no one trains from scratch; the skill is choosing a starting model and adapting it well

Why scale works

an empirical regularity across model families, in three quantities: parameters, data and compute

A power law

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N} + L_\infty$$

Loss falls smoothly as a power law in parameters N; data and compute have similar forms

In words

Small exponents. Each constant improvement needs a multiplied budget, so progress has come from orders of magnitude.

Grow together. A large model on little data overfits. A small model on much data saturates.

In proportion. A model of a given size is best trained on an amount of data in proportion to it.

In robotics the same curves appear to hold for data variety.

Returns to scale are slow and predictable, which is what justified building very large models

The adaptation ladder

six ways to get a pretrained model to do a task, cheapest first; data counts are rough orders of magnitude

Method	What changes	Data needed	Reach for it when
Prompting	Only the input text	None	The model already knows how and needs instructions
In-context examples	The input, with a few worked examples	A handful	The output format or style is specific
Retrieval augmentation	The input, with documents fetched at run time	A document store	Answers depend on facts the model lacks
LoRA and adapters	Small added weight matrices; the base is frozen	Hundreds to thousands	A new skill or domain on a limited budget
Full fine-tuning	All the weights	Thousands and up	A large shift, such as a new robot or new kind of output
Distillation	A smaller student copies the adapted model	The teacher's outputs	The result must run on a small device

Try the cheap rungs first; each rung up costs more data, compute and upkeep

Low-rank adaptation in symbols

freeze the pretrained matrix W_0 and learn a correction that is the product of two thin matrices

LoRA

$$W = W_0 + \frac{\alpha}{r} BA$$

The frozen weight plus a scaled product of two small matrices

$$B \in \mathbb{R}^{d_{\text{out}} \times r}, \quad A \in \mathbb{R}^{r \times d_{\text{in}}}, \quad r \ll \min(d_{\text{in}}, d_{\text{out}})$$

The rank r is far smaller than either side of the matrix

$$d_{\text{in}}\, d_{\text{out}} \longrightarrow r\,(d_{\text{in}} + d_{\text{out}})$$

Trained parameters per matrix fall from the first count to the second

In words

B starts at zero, so training begins from exactly the pretrained model.

Free at inference. After training, the product can be added into W_0, so there is no extra cost.

Removable. Take the adapter away and the original model is back.

Swappable. Several adapters can be kept for one base model and swapped per task.

At rank 8, a 2,048 × 2,048 matrix trains 32,768 parameters in place of 4.2 million, under 1%

Low-rank adaptation, measured

measured for this note: a small network trained on one task is adapted to a second

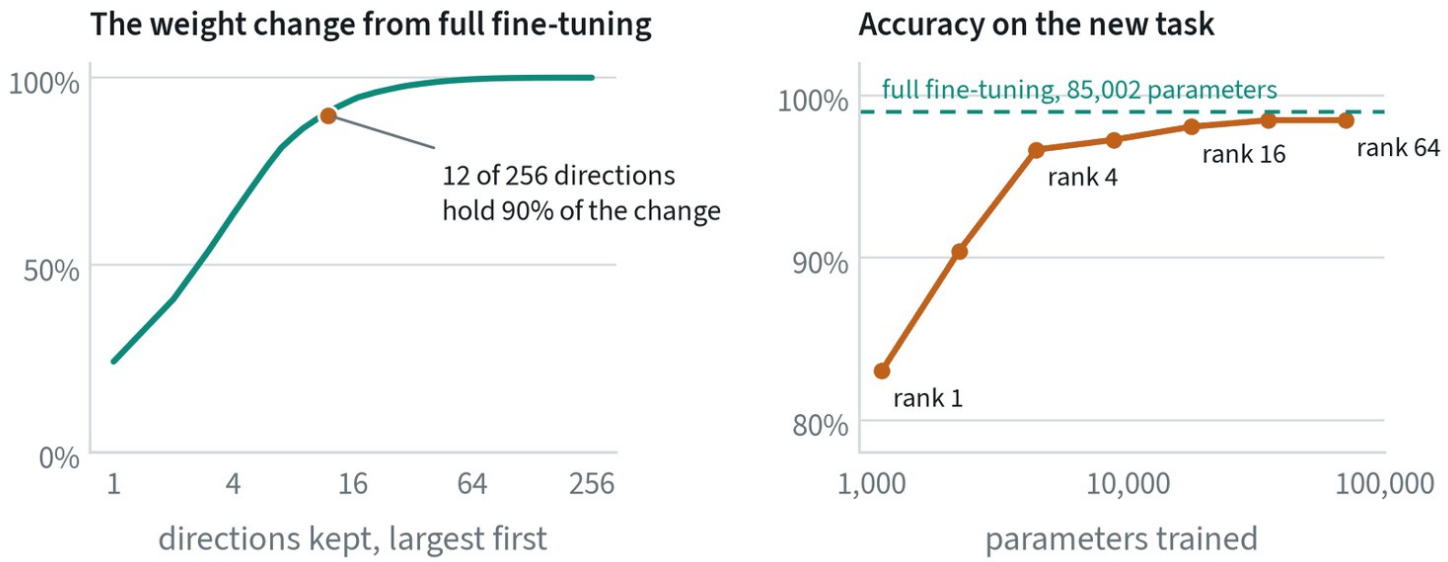
What the experiment shows

Left. 90% of the change that full fine-tuning made to the middle weight matrix lay in 12 of 256 possible directions.

Right. Rank 4 trains 4,392 parameters, 5% of the total, and reaches 96.6%. Rank 16 reaches 98.1%.

Full fine-tuning of all 85,002 parameters reaches 99.0%.

What it cost. The original task fell from 98.4% to 38.4% after full fine-tuning. An adapter avoids this for free.



The full fine-tuning update was already nearly low-rank, so restricting it loses little

Sampling is a setting: temperature

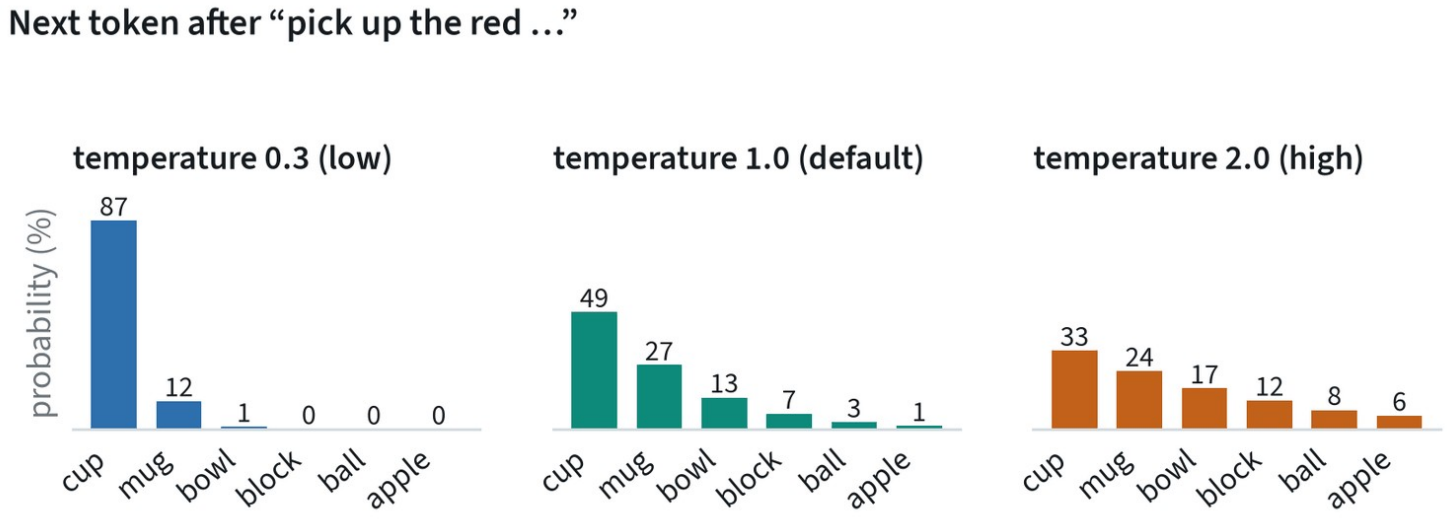
one set of scores for six candidate words, turned into probabilities at three temperatures

Temperature rescales the scores before they become probabilities. How a token is picked is separate from the model.

0.3, 1.0 and 2.0. The top word holds 87%, then 49%, then 33% with the others close behind.

The trade. Near zero the model almost always takes its top choice: repeatable, and sometimes dull. Higher values give variety and more mistakes.

Top-p sampling is a common companion. It discards the unlikely tail before picking.



For structured output and for robot actions written as tokens, use a low temperature

From answering to acting

three ideas that form the bridge from chat models to systems that do work

Reasoning

- Have the model write intermediate steps before its answer, often called chain of thought.
- This trades extra compute at run time for accuracy.

Tool use

- The model emits a structured call, such as a search or code to run, and the result is fed back as new input.
- It can then act on facts it was never trained on.

Agents

- A loop of observe, decide, act and observe again.
- Useful for multi-step work.

A robot is an agent whose tools are motors

Context length and made-up answers

two limits of every language model, and what to do about each

Context window

- How many tokens the model can read at once: instructions, documents, images and its own output so far.
- Anything outside it does not exist for the model.
- Long contexts cost more, respond more slowly, and recall the middle less reliably than the start and end.

Hallucination

- A language model always produces fluent text, including when it has nothing to base it on.
- Defences: supply facts through retrieval, ask for a form that can be checked, verify what matters.
- For a robot the check is physical: confirm the named object is in view before acting on it.

Give the model the facts, ask for checkable output, and verify anything that matters

Common mistakes with foundation models

three errors of order: doing the expensive thing before the cheap one

Fine-tuning first

- Many tasks are solved by a better prompt or by retrieval, at a fraction of the cost.

Forgetting what was lost

- Fine-tuning on a narrow dataset can erase general abilities the base model had.

Skipping the baseline

- Measure the unadapted model first, or you cannot say what adaptation bought.

Measure the unadapted model first, then climb the ladder one rung at a time

Vision-language models

How a language model is taught to read images, what the image costs, what these models get wrong, and how one becomes a robot policy.

KEY POINTS

A VLM is a language model with image patches added to its prompt.

Image tokens are the cost. Resolution and merging decide what the model can see and how fast it runs.

It brings knowledge of the world and is unreliable on counting, position and unfamiliar views.

Three ways to join vision and language

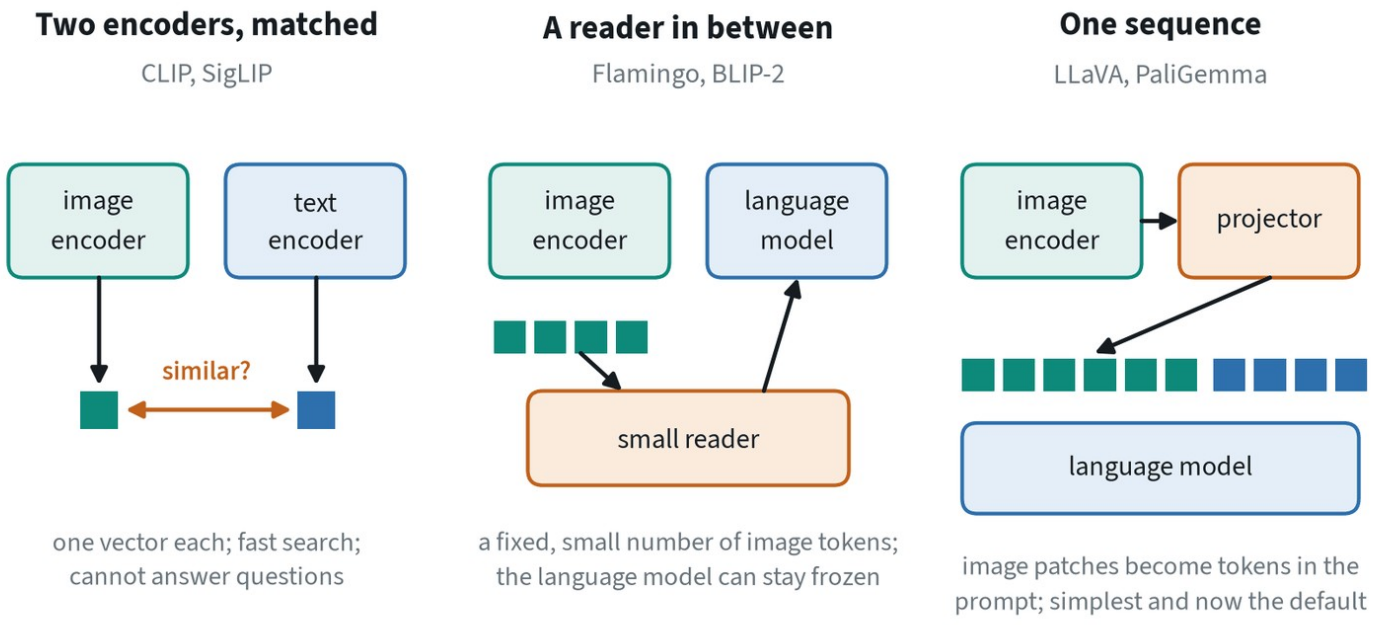
a VLM is a language model that can also read images; the three families are shown in the order they appeared

Two encoders, matched. An image encoder and a text encoder each produce one vector, and the vectors are compared.

A reader in between. A small module reads the image encoder's output and passes a small, fixed set of tokens to a language model.

One sequence. Image patches pass through a projector and join the text tokens in one sequence read by the language model.

Why robotics wants one. A VLM arrives knowing what a mug is and what "left of" means. A policy built on it only has to learn how to move.



Almost all current models, and almost all robot policies, put patches and text in one sequence

The three families compared

what each design costs, what it keeps of the image, and what it is used for

Property	Two encoders	A reader in between	One sequence
Image enters as	One vector	A small fixed set of tokens	Hundreds of patch tokens
Can generate text	No	Yes	Yes
New parts to train	Both encoders	The reader, and cross-attention layers	One small projector
Cost per image	Lowest	Low	Highest: the tokens fill the prompt
Fine spatial detail	Lost	Partly kept	Best kept
Used for	Retrieval, zero-shot labels	Many images or video on a budget	General assistants, robot policies

The single-sequence design won on simplicity, and pays for it in tokens

A real model, by the numbers

PaliGemma, counted from the model itself: two existing models joined by one linear layer

Tokens and objective

$$\left(\frac{224}{14}\right)^2 = 256 \text{ patch tokens}$$

A 224-pixel image in 14-pixel patches, placed at the start of the sequence

$$\mathcal{L} = -\sum_t \log p_\theta\left(y_t \mid I,\ \text{prefix},\ y_{<t}\right)$$

Predict each answer token from the image, the prefix and the answer so far



Vision encoder (SigLIP) 412 million parameters, projector 2.4 million, language model (Gemma) 2,509 million.

The projector that joins the two halves is under a tenth of a percent of the total

The image is the expensive part

tokens per image for six configurations, on a log scale; a sentence is a few dozen tokens

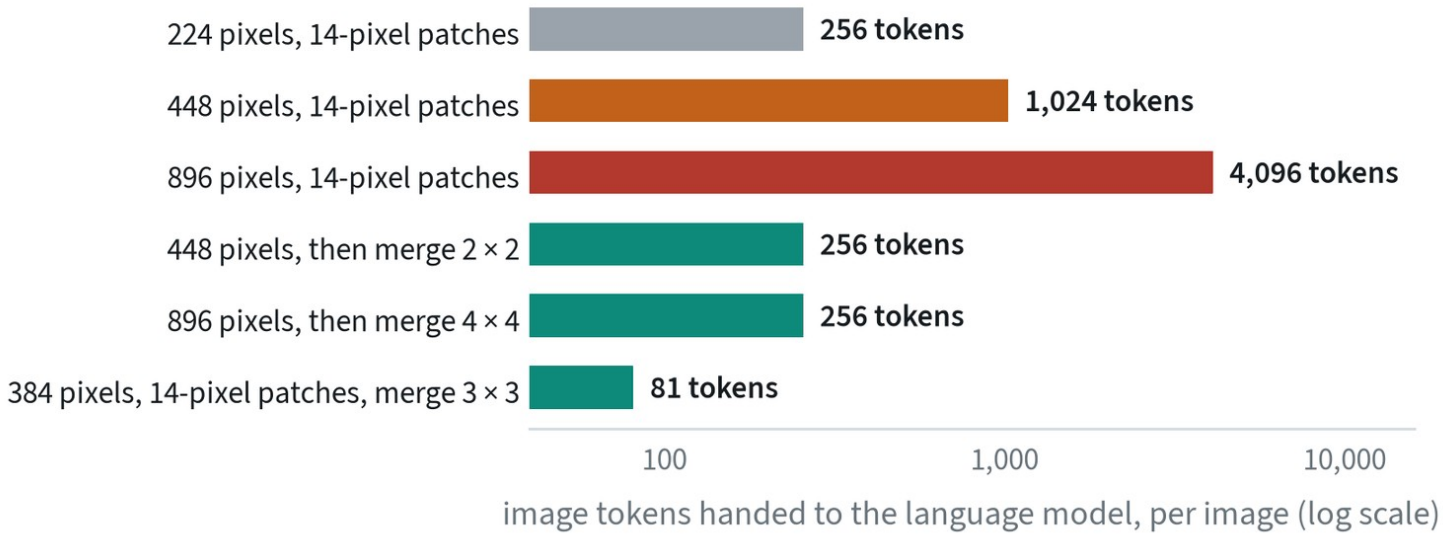
Spending image tokens

Resolution multiplies. With 14-pixel patches, 224 pixels is 256 tokens, 448 is 1,024 and 896 is 4,096: four times for each doubling.

Merging divides. 448 pixels merged two by two, or 896 merged four by four, is 256 again. 384 with three by three merging is 81.

Tiling and readers. Tiles plus a low-resolution overview keep detail and multiply tokens. A reader fixes the count and loses spatial precision.

Robot policies sit at the frugal end: low resolution or heavy merging, and often a compact model.



Every VLM design is a decision about how many image tokens to spend

What resolution buys

one region of a supermarket shelf photograph, as the vision encoder receives it at 224, 448 and 896 pixels

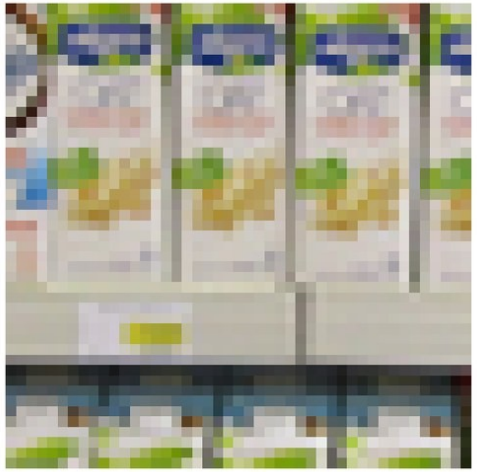
At 224 pixels the region is 45 pixels across and the product labels are unreadable blocks.

At 448 and 896 it is 89 and then 179 pixels across. The brand name can be made out, then the label text and price tag.

The bill. Each step multiplies the image tokens by four and the cost of the language model by more than that.

No recovery. No amount of language modelling can recover what the left-hand version has lost.

224 × 224 input · 256 tokens
this region: 45 × 45 pixels



448 × 448 input · 1,024 tokens
this region: 89 × 89 pixels



896 × 896 input · 4,096 tokens
this region: 179 × 179 pixels



Wrist cameras put the pixels where the detail is, instead of paying for resolution everywhere

How one is trained

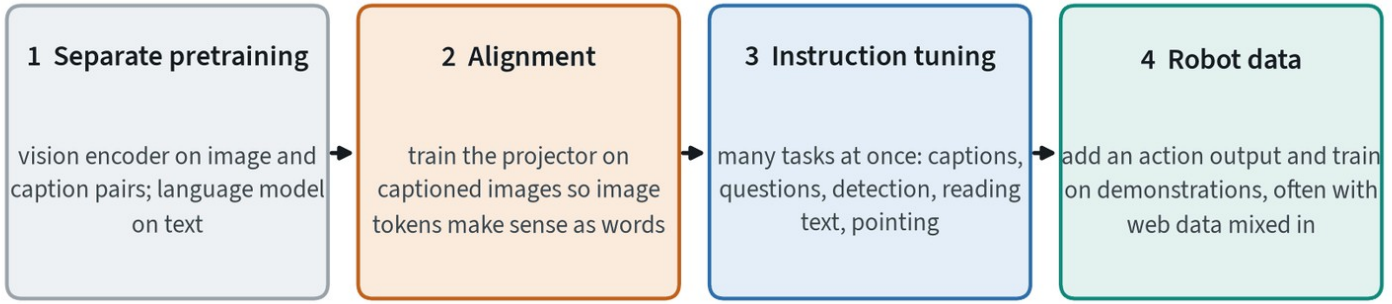
the usual sequence of four stages, each cheaper and using less data than the one before it

Separate pretraining. The vision encoder on image and caption pairs, usually contrastively, and the language model on text.

Alignment. Train only the projector, on captioned images, until image tokens are usable by the language model.

Instruction tuning. Train on a mixture of tasks phrased as instructions, with more of the model unfrozen.

Robot data. Add an action output and train on demonstrations.



Each stage starts from the weights of the one before. Stage 4 is what turns a vision-language model into a robot policy.

What gets frozen at each stage is a real decision between preserving features and adapting fully

Beyond text answers: locations as tokens

real PaliGemma output on a LIBERO frame; the boxes are decoded from the location tokens it wrote

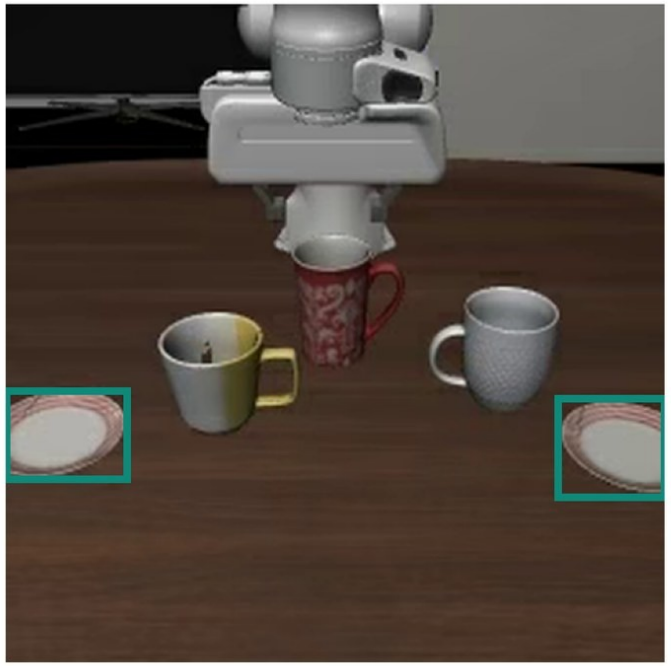
Everything is tokens. A language model can only emit tokens, so every other output is written as tokens.

Location tokens. Some models add a thousand or so values per axis to the vocabulary, and a box becomes four of them.

Alternatives. Other models write coordinates as ordinary numbers or output points.

In the frame. Asked for plate, the model boxes both plates. Asked for mug, it boxes the gripper and none of the three mugs.

“detect plate”: both found



“detect mug”: the gripper, not a mug



If a location can be a token, so can an action

What they get wrong

two public models asked about one frame; there are three mugs, and the one on the left is white and yellow

Question	Model	Answer
Describe the image	PaliGemma	"a robot's hand picks up a cup and puts it on a saucer"
Describe the image	SmolVLM2	"A coffee maker is on a table with two cups and a plate."
How many mugs are on the table?	PaliGemma	"couple"
How many mugs are on the table?	SmolVLM2	"There are two mugs on the table."
What colour is the mug on the left?	PaliGemma	"red"

Each answer is fluent, and each is wrong in a way that would matter to a robot

From a VLM to a robot policy

three ways in use to give the model an action output, and one practice that protects what it knew

Action tokens

- Write each action as tokens from the vocabulary and train on demonstrations.
- Nothing is added to the architecture.

An action head

- A small network reads the VLM's internal features and generates continuous actions, usually by flow matching.
- Language abilities stay largely intact.

A planner above a policy

- Leave the VLM as it is and have it name sub-tasks for a separate low-level policy.

Co-training

- Keep training on web image-and-text tasks alongside the robot data.

Learning to act should not erase what the model knew about the world

Common mistakes with vision-language models

three errors, one for each of the note's themes: tokens, reliability and adaptation

Choosing the resolution last

- It decides what the model can see and how fast it runs.
- Decide it first.

Trusting a description

- Check counts, colours and positions against the image before acting on them.

Fine-tuning on robot data alone

- The model's general knowledge erodes.
- Mix in the data it was built on.

Decide the resolution first, check what the model says, and keep web data in the mix

Perception and 3D

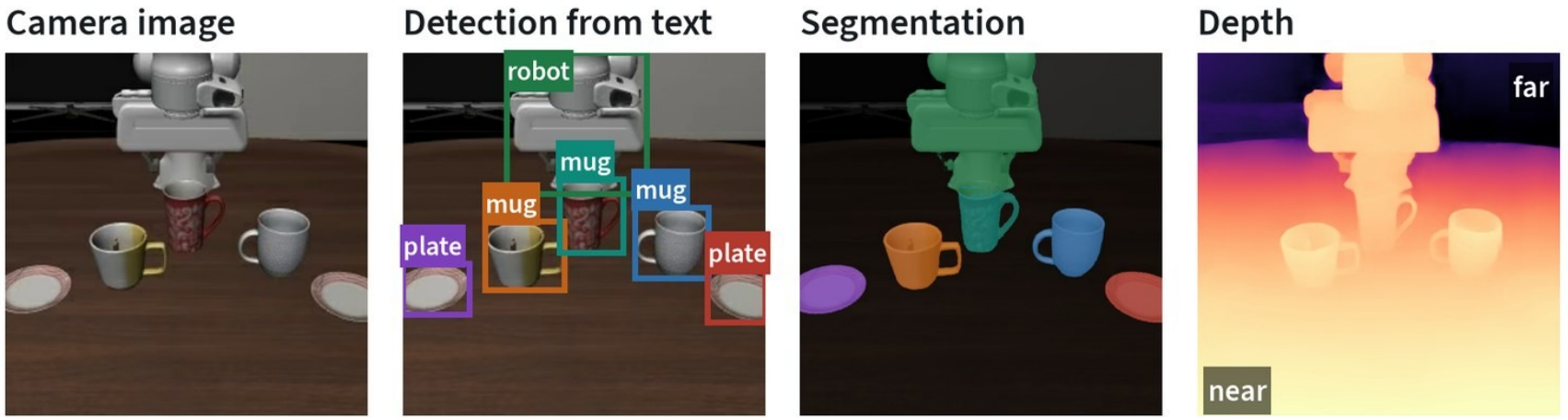
From pixels to objects, depth and maps, and when a robot still needs them.

KEY POINTS

- Open-vocabulary models find and segment objects from a text prompt, with no new training.
- Depth plus camera geometry turns an image into 3D points.
- Deployed robots usually combine a learned policy with explicit perception around it.

From pixels to things, with no new training

real output on one simulated robot frame: Grounding DINO boxes, SAM 2 masks and Depth Anything V2 depth



Why it still matters. Systems name objects, measure distances and build maps to label data, check success, monitor safety and plan.

Open vocabulary. The detector was given the words "mug, plate, robot", and the segmenter was prompted with its boxes.

The change. Older detectors knew a fixed list of classes. None of these three models was trained on this scene.

Where to check. On another frame of the same dataset, the detector labelled the gripper a moka pot with fair confidence.

With open vocabulary, a new object no longer needs a new labelled dataset

Four perception tasks

what each task outputs and what a robot system uses it for

Task	Output	Used for
Detection	A box and a label for each object	Counting, finding what to pick
Segmentation	The exact pixels of each object	Grasp planning, masking, success checks
Tracking	The same object followed across frames	Following moving items, video labelling
Pose estimation	An object's position and orientation in 3D	Precise placement and insertion

Detection finds, segmentation outlines, tracking follows, pose estimation places in 3D

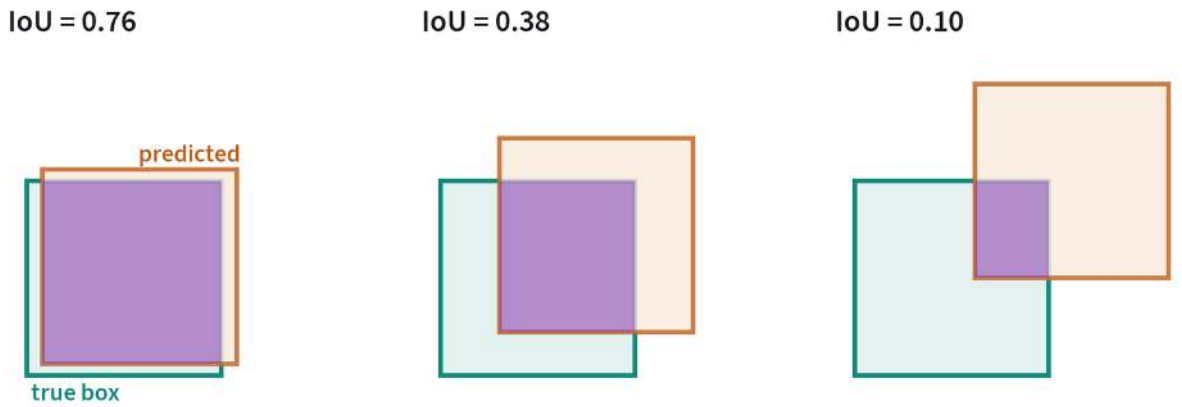
How detections are scored

a predicted box is compared with the true one by intersection over union

Intersection over union

$$\mathrm{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The area two regions share, divided by the area they cover together



Predicted boxes of the right size at increasing offsets score 0.80, 0.38 and 0.11. A detection commonly counts as correct above 0.5.

A shift of less than a third of the box width already drops the score to 0.38

From one overlap score to a benchmark number

how confidence, precision and recall are folded into average precision, and how duplicates are handled

Confidence threshold

- A detector outputs a confidence with each box.
- Lowering the threshold trades precision for recall.

Average precision and mAP

- Average precision is the area under the precision-recall curve. mAP is its mean over classes.
- The benchmark figure averages again over IoU thresholds from 0.5 to 0.95.

Non-maximum suppression

- Older detectors propose many overlapping boxes per object.
- Keep the most confident box, delete every box that overlaps it by more than a threshold, and repeat.

Transformer detectors

- They predict a fixed set of boxes, trained with one-to-one matching to true objects.
- No suppression step is needed.

The common benchmark figure rewards boxes that are tight as well as present

Depth and point clouds

every pixel of the camera frame, moved into 3D using its predicted depth and the pinhole model

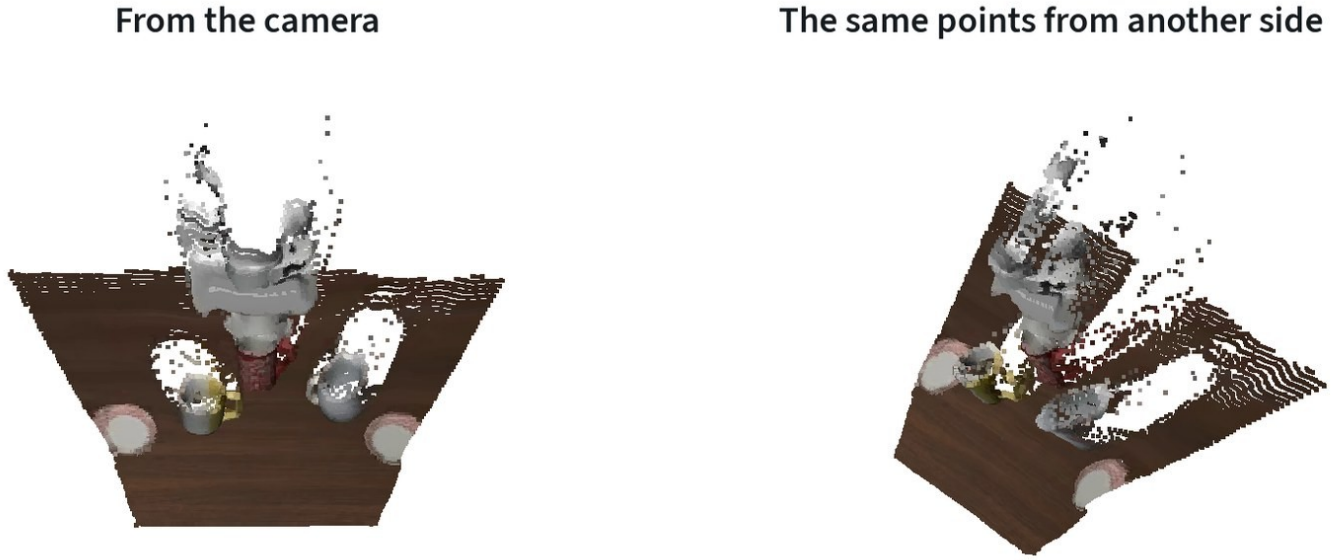
Depth turns an image into geometry

Sources. A sensor (two cameras, projected light or lidar) or a model that estimates depth from one image.

Relative or metric. Relative models give the ordering of near and far. Metric models estimate actual distances.

In the figure. The table is a plane and the mugs stand on it. The ragged edges are depth errors at object boundaries.

Limits. Edges smear and the scale is an estimate. Sensors have their own failures on shiny, dark and transparent objects.



Depth plus camera intrinsics makes each pixel a 3D point: plausible shapes, estimated scale

The geometry underneath

three pieces of geometry that turn perception into something a robot can act on

Three problems

$$X = \frac{(u - c_x)\, Z}{f_x} \qquad Y = \frac{(v - c_y)\, Z}{f_y}$$

Back-projection: pixel (u, v) with depth Z becomes a 3D point in the camera frame

$$\min_{R,\,t} \sum_i \left\| \pi\left(R\, X_i + t\right) - u_i \right\|^2$$

Perspective-n-point: the pose (R, t) that minimises reprojection error

$$\min_{R,\,t} \sum_i \left\| R\, p_i + t - q_i \right\|^2$$

Registration: the rigid transform that best aligns matched pairs of points

In words

Back-projection made the point cloud. Applying the camera's pose then moves each point into the robot's base frame.

Pose from correspondences. Known 3D points on an object and where they appear in the image. Four determine the pose, and more make it robust to noise.

Iterative closest point alternates nearest-neighbour matching with a closed-form solve for the transform until it converges.

Starting guess. Registration needs a reasonable one, which is usually what a learned pose estimator provides.

Back-project pixels to points, solve pose from correspondences, align clouds by registration

Object pose and grasping

many industrial tasks reduce to knowing a part's pose precisely

6D pose

- Three numbers of position and three of orientation.
- Methods match the image against a known 3D model of the part.
- More recent ones estimate the pose of objects never seen before from a few reference views.

Grasp detection

- Grasp detectors take a point cloud and propose gripper poses directly.

Perceive, estimate pose, plan, move

- This pipeline is the classical alternative to a learned policy.
- It struggles with clutter, deformable items and anything it has no model for.

The classical pipeline is precise and easy to inspect when parts are rigid and known

Maps, navigation and learned 3D scenes

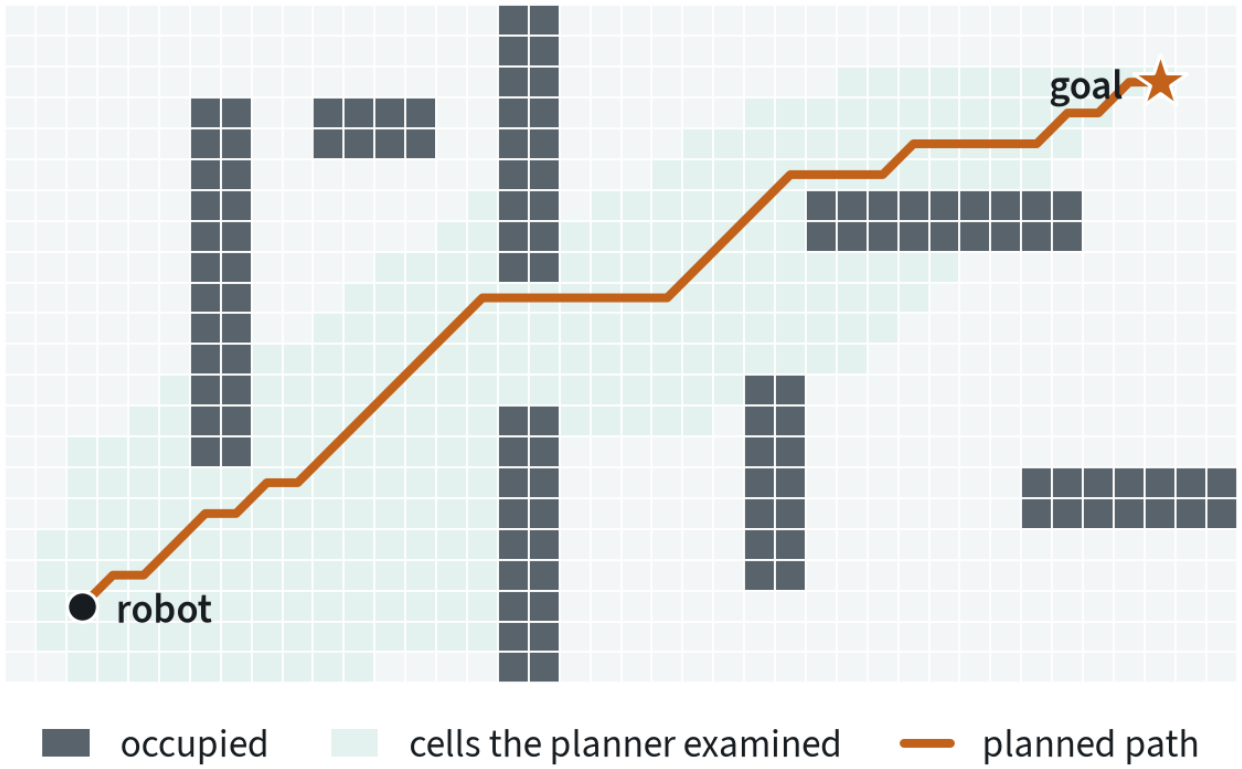
a computed example: A on an occupancy grid, from the robot at the lower left to a goal at the upper right*

Occupancy grid. The simplest map: free cells and occupied cells.

In the figure. The planner examined 284 of the 766 free cells and returned a 36-cell path around the obstacles.

Maps with meaning. Language-aligned features attached to each place let a robot act on "go to the kitchen" without anyone marking it.

Learned 3D scenes. Radiance fields and Gaussian splatting build a scene from photos at known positions and render it from any viewpoint.



SLAM builds the map and tracks the robot in it at once; a planner then searches it for a path

Modular or end-to-end

the classical pipeline and the learned policy, side by side

Aspect	Modular pipeline	End-to-end policy
How it works	Perceive, estimate, plan, then move	Pixels in, actions out
Strengths	Each stage can be tested and explained; precise with known objects	Handles clutter, deformables and tasks that are hard to specify
Weak points	Errors pass down the chain; every stage needs engineering	Needs demonstrations; failures are harder to diagnose
Common today	Industrial pick-and-place, navigation	Household and general manipulation

Most deployed systems mix the two: a learned policy with explicit perception around it

Common mistakes with perception

three assumptions that hold on a benchmark and fail on a real cell

Trusting a confidence score

- Detectors are overconfident on images unlike their training data.
- Check on your own scenes.

Forgetting calibration

- A perfect detection in the camera frame is useless with a wrong camera pose.

Expecting depth on glass and metal

- Test the actual objects, not a matte stand-in.

Test on your own scenes, your own calibration and the actual objects

ROBOT LEARNING · TOPIC 14 OF 32

Robotics fundamentals

Frames, kinematics, rotations, controllers and cameras: the classical layer every learned policy sits on.

KEY POINTS

Every position and rotation is relative to a frame, and transforms chain by multiplication.

The Jacobian links joint motion to hand motion and hand force to joint torque; where it degenerates, the arm is singular.

An action only has a meaning together with the controller that executes it.

Run it: Frames, kinematics and rotations · Controllers: from a target to motor torque (github.com/wnukmat/robot-learning-notebooks)

Everything is measured in a frame

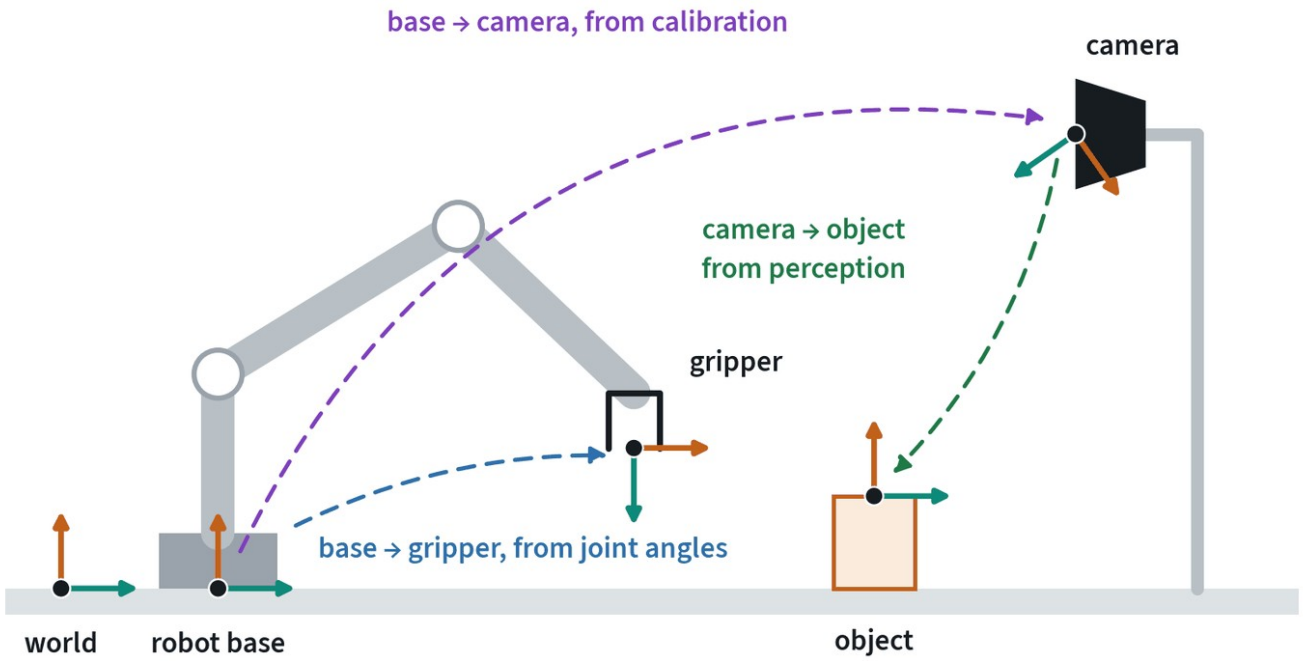
a robot knows where the cup is only relative to something: the camera, its base or the gripper

Frame and pose. A frame is a point with axes attached. A pose is the position and orientation of one frame as seen from another.

Three transforms. Base to gripper comes from joint angles, base to camera from calibration, camera to object from perception.

Errors pass through. Transforms chain, so a camera calibration that is 2 cm off gives a grasp that is 2 cm off.

Three checks. Which frame the number is expressed in, what the units are, and in which order the transforms are applied.



A frame error raises no exception. The robot just reaches for the wrong place

Transforms in symbols

a rotation R and a translation t packed into one 4 × 4 matrix, so that poses compose by multiplication

In symbols

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}, \qquad \begin{bmatrix} p' \\ 1 \end{bmatrix} = T \begin{bmatrix} p \\ 1 \end{bmatrix}$$

Apply the rotation, then add the translation, as a single matrix product

$$T^{-1} = \begin{bmatrix} R^\top & -R^\top t \\ 0 & 1 \end{bmatrix}$$

The inverse transform is built from the transpose of R

$$T_{\text{base, object}} = T_{\text{base, camera}}\, T_{\text{camera, object}}$$

Transforms chain by multiplication; the inner frame names match and cancel

In words

Notation. T_AB is the pose of frame B expressed in frame A.

Hand-eye calibration finds the fixed base-to-camera transform, or gripper-to-camera for a wrist camera, from several views of a known target.

A X = X B. Calibration reduces to this form. A and B are measured motions of the robot and of the target, and X is the unknown.

Nine entries, three freedoms. A rotation matrix must satisfy $R^T R = I$ and $\det R = 1$.

Write the frame names on every transform and the chain checks itself

Kinematics: joints and the hand

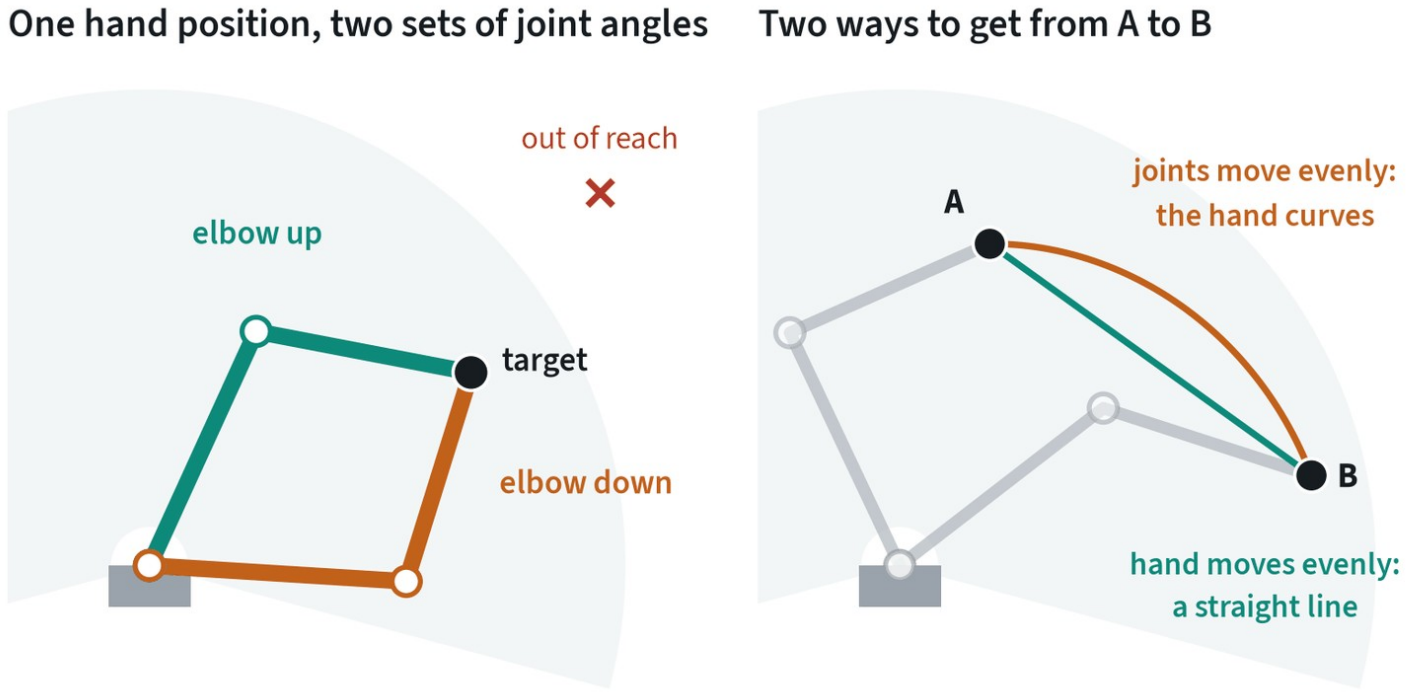
computed for a two-link arm: joint solutions on the left, two ways of interpolating on the right

Forward kinematics turns joint angles into the pose of the gripper. It is a fixed calculation with one answer.

Inverse kinematics can have several answers or none. Elbow up and elbow down reach the same target, and one marked point is out of reach.

Singularity. With the arm stretched out or two joints lined up, a small move of the hand needs a very large move of the joints.

Straight lines. Moving the joints evenly curves the hand's path from A to B. To slide along a table, plan in the hand's space.



A policy that outputs gripper poses relies on a solver; one that outputs joints is tied to one robot

The Jacobian links joints, hand and force

the derivative of forward kinematics x = f(q), relating small changes and, through them, velocities and forces

In symbols

$$\dot{x} = J(q)\,\dot{q}, \qquad \tau = J(q)^{\top} F$$

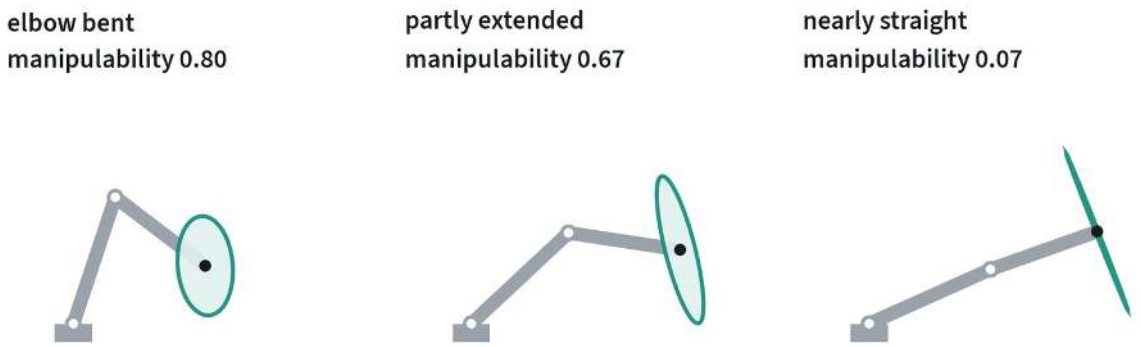
Joint speeds give the hand's velocity; a force at the hand maps to joint torques

$$w = \sqrt{\det\left(J J^{\top}\right)}$$

Manipulability: the size of the ellipse. Near zero the arm is at a singularity

$$\Delta q = J^{\top}\left(J J^{\top} + \lambda^{2} I\right)^{-1} \Delta x$$

Damped inverse kinematics step: a little accuracy traded for staying finite



Two-link arm. The ellipse of reachable hand velocities flattens as the arm straightens: manipulability 0.80 elbow bent, 0.67 partly extended, 0.07 nearly straight.

Where manipulability approaches zero, a small hand motion demands enormous joint speeds

Rotations are not ordinary numbers

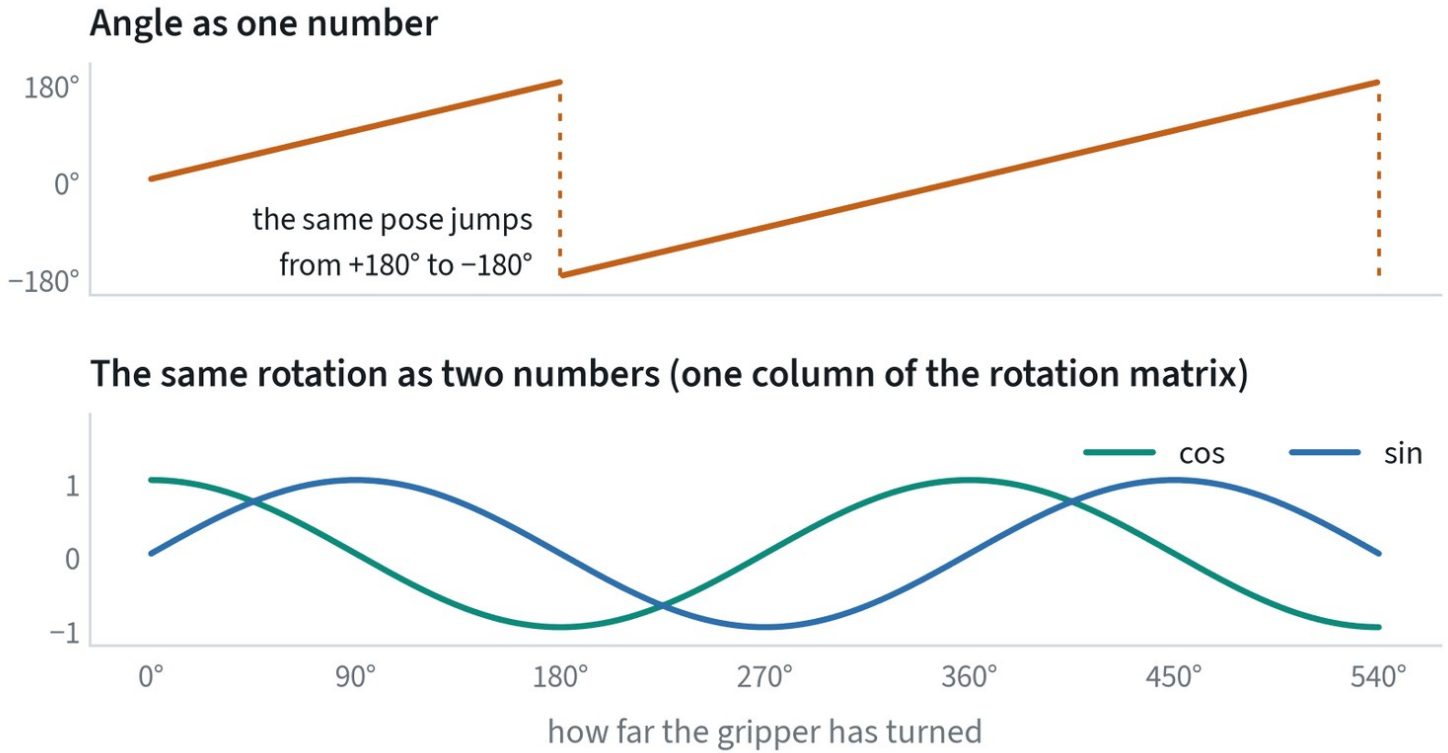
a gripper turning steadily, written as one angle and as a cosine and a sine

The jump. Written as one angle, the value rises steadily and then leaps from +180° to −180° where nothing physical happened.

Two numbers. The cosine and sine of the same rotation change smoothly throughout.

Hard to learn. A network asked to output the top curve must reproduce the jump exactly.

Averaging fails. The mean of two nearly identical poses on either side of the jump points the opposite way.



Every compact way to write a rotation breaks somewhere, so modern policies avoid single angles

Five ways to write a rotation

each representation, how many numbers it uses, where it breaks and where it turns up

Representation	Numbers	Its problem	Where you will see it
Euler angles	3	Depends on the axis order; wraps around; loses a degree of freedom at some poses	Datasets, user interfaces
Axis-angle	3	Wraps at half a turn	Many robot action formats
Quaternion	4	Two quaternions describe every rotation; must be kept at unit length	Simulators, robot drivers
Rotation matrix	9	Redundant; a network's output is not a valid matrix without clean-up	Maths and libraries
6D (two matrix columns)	6	Needs a small step to rebuild the matrix	Network outputs

Find out which representation a dataset uses, and compose rotations by multiplying

From an action to motor torque

a policy produces targets 10 to 50 times a second; a classical controller underneath runs at 500 to 1,000

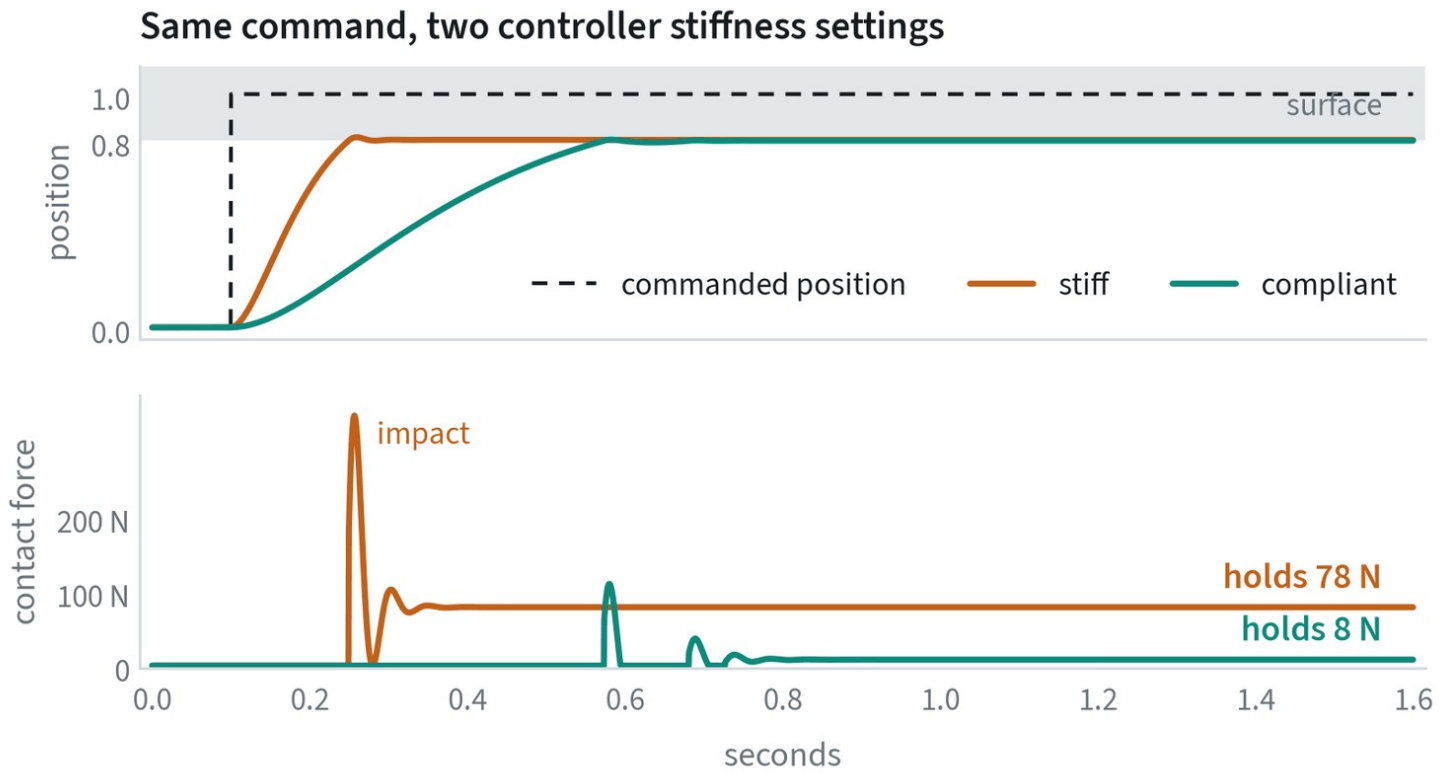
Three controllers

Position control. Go to the target and resist anything in the way. Precise in free space and harsh in contact: 78 N here.

Impedance control. Behave like a spring and damper around the target. Contact is gentle, 8 N here, and position is less exact.

Torque control. Command the motor torques directly. The most freedom, and it needs a fast loop and a good model of the robot.

Commanded or reached. The two positions differ, and a dataset may store either.



An action only has a meaning together with the controller that executed it

Controllers in symbols

from the lowest loop upward: error feedback, the arm's known dynamics, and a spring and damper at the hand

In symbols

$$u = K_p\, e + K_i \int e \,\mathrm{d}t + K_d\, \dot{e}$$

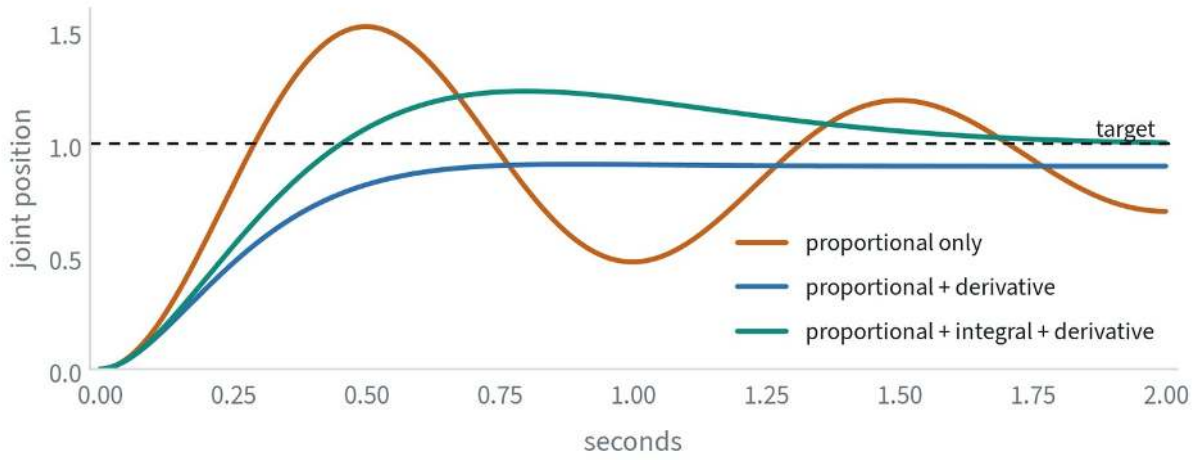
PID: a motor command from the position error e = q_d − q, its integral and its rate

$$M(q)\,\ddot{q} + C(q,\dot{q})\,\dot{q} + g(q) = \tau + J^{\top} F_{\text{ext}}$$

Arm dynamics: inertia M, velocity-dependent terms C and gravity g

$$F = K(x_d - x) + D(\dot{x}_d - \dot{x}), \qquad \tau = J^{\top} F + g(q)$$

Impedance: the hand acts as a spring and damper around its target



A simulated joint holding a constant load. Proportional control overshoots to 1.5 and oscillates. Adding a derivative term settles at 0.9. The integral term reaches 1.0.

Stiffness K decides the contact: a 2 cm error gives 8 N at K = 400 and 80 N at K = 4,000

Cameras, calibration and time

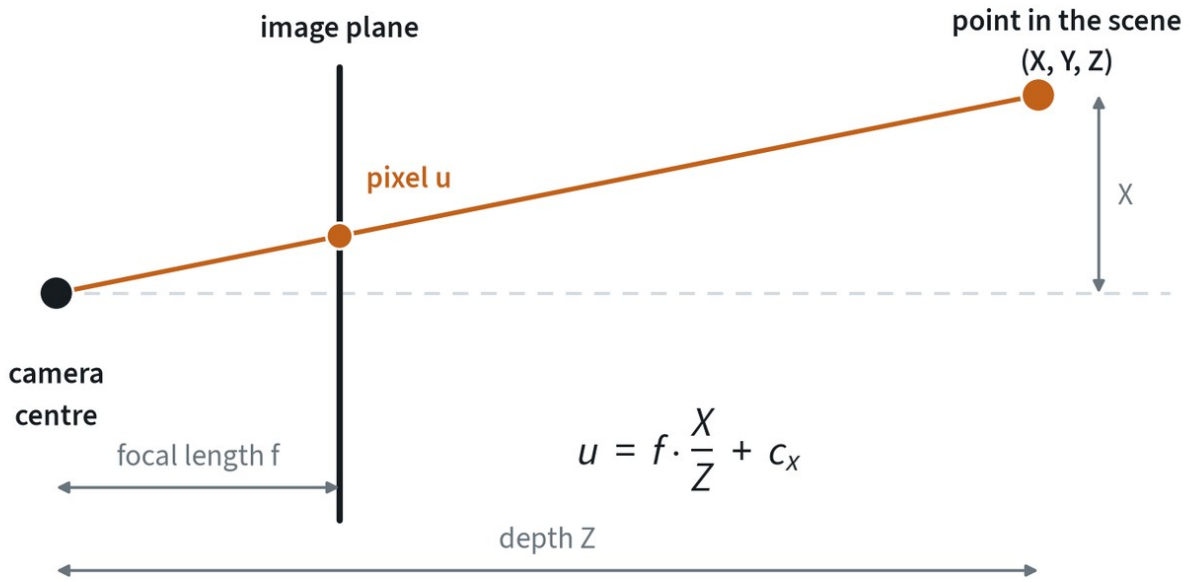
the pinhole model, the camera's pose relative to the robot, and the delay on every sensor and motor

Pinhole model. $u = f \cdot X / Z + c_x$. A pixel depends on sideways offset divided by depth, so one image cannot give distance.

Intrinsics and extrinsics. Focal length, centre and distortion turn a pixel into a ray. With the camera's pose and a depth, it is a reachable point.

Hidden dependency. An end-to-end policy absorbs the camera placement. A bump of a few centimetres is a quiet cause of lower success.

Time. Timestamp at the source and align streams by time. Keep the lag between images and joints the same in training and deployment.



Treat camera placement and timing as part of the model

Common mistakes in the classical layer

four errors that produce no exception and a wrong motion

Mixing frames or units

- A pose in the camera frame fed to a function that expects the base frame gives no error and a wrong reach.

Adding rotations

- Angles do not add.
- Compose rotations by multiplication.

Assuming the command was reached

- Check whether the dataset stores commanded or measured positions.

Moving the camera

- Treat camera placement as part of the model.
- Check it before blaming the policy.

Check frames, units, rotations and camera placement before blaming the policy

Robot learning basics

The control loop, action spaces, imitation learning and where reinforcement learning fits.

KEY POINTS

- Unlike a classifier, a policy's outputs change its own future inputs.
- Before training on any dataset, find out which action space it uses; mixing them silently breaks a policy.
- Predicting a chunk of future actions, not one, is the simple trick that made imitation learning work.
- Imitation leads in manipulation, RL in locomotion, and classical control runs under both.

Run it: Behaviour cloning and compounding error · Action spaces and normalisation, on real robot data (github.com/wnukmat/robot-learning-notebooks)

The control loop

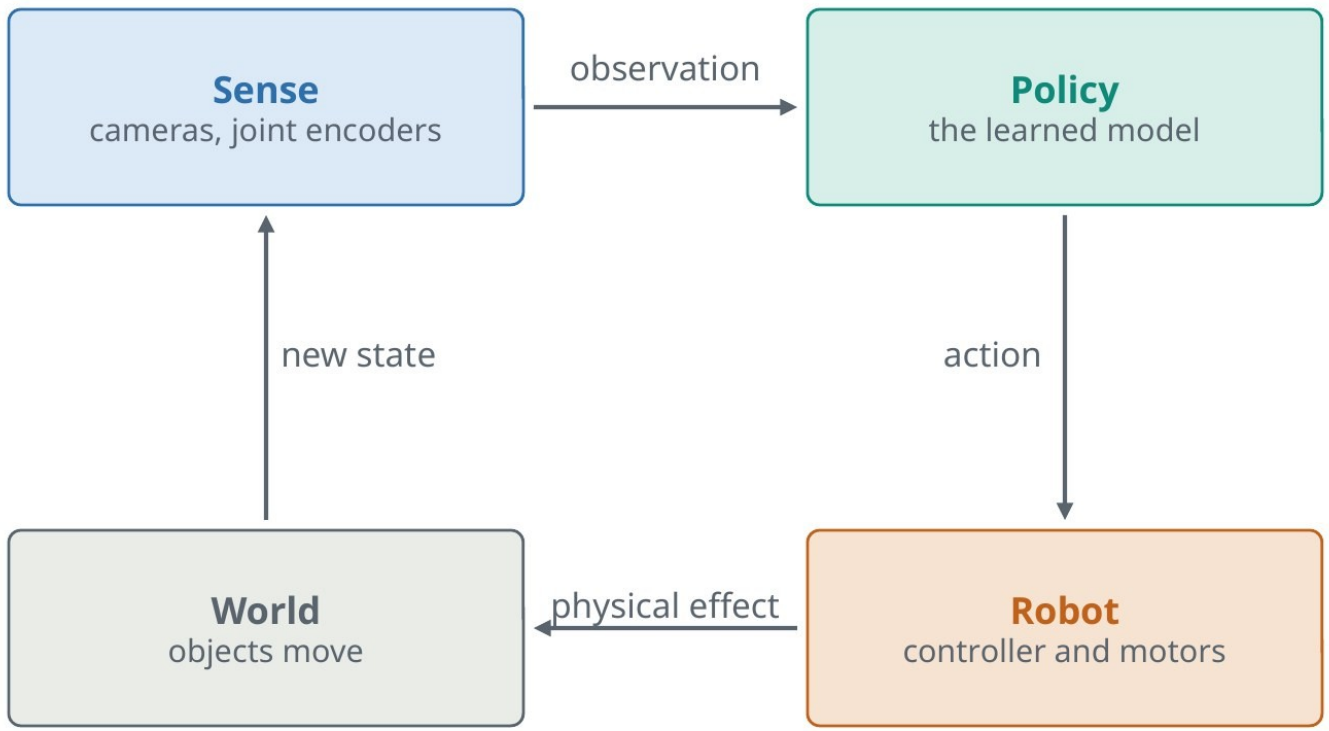
a policy is a function from what the robot senses to what it does next, called many times a second

Observation. Camera images plus the instruction.

Proprioception. The robot's own state: joint angles and how far the gripper is open.

Two rates. Learned policies typically run at 10 to 50 Hz, with a classical motor controller underneath at 500 Hz or more.

Closed loop. A classifier's test inputs do not depend on the classifier. A policy's outputs change its own future inputs.



A small mistake moves the robot somewhere the model may never have seen

Action spaces: joints or hand

"move the gripper to the cup" can be written as numbers in several ways; this is the first of four decisions

Aspect	Joint space	End-effector (Cartesian) space
The policy outputs	A target for every joint	A position and rotation for the hand
Advantages	Exact; no solver; the whole arm's shape is controlled, which matters near obstacles	Close to how tasks are described; the same numbers mean the same thing on different arms
Costs	Tied to one robot; the link between a joint and the task is indirect	Needs inverse kinematics; the elbow goes wherever the solver puts it; rotation formats differ
Typical use	Two-arm and humanoid systems trained on one body	Single arms and cross-robot datasets

The choice affects how easily the policy learns, how errors build up and whether data can be combined

Absolute or relative, position or velocity

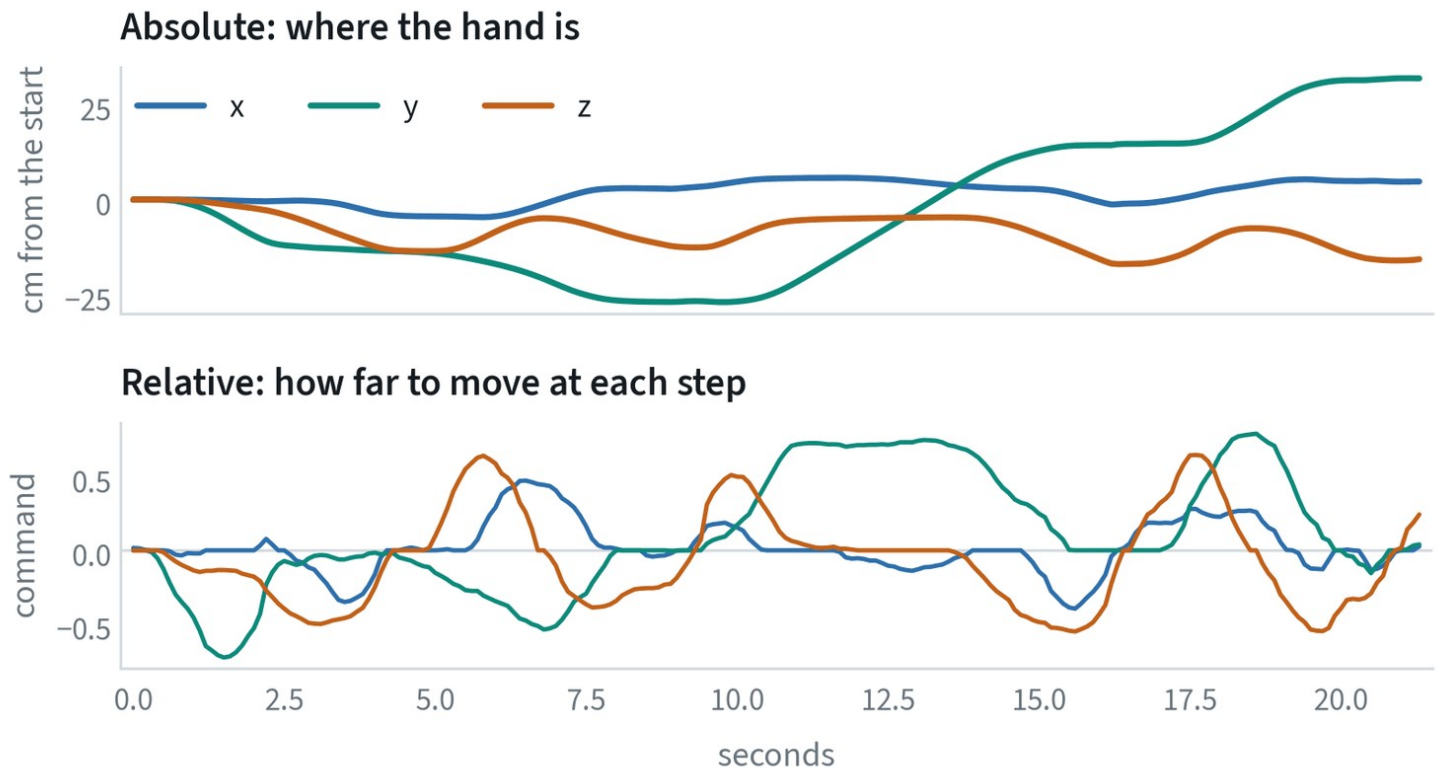
one real 21-second episode written both ways, in three axes

Absolute says where to be. Errors do not build up, and the policy has to learn the workspace: the curves drift up to 30 cm from the start.

Relative, also called a delta, says how far to move from here. It helps generalisation, and each small error is added to the last.

Position or velocity. Most policies output position targets. Velocities suit continuous motions and make the result depend on timing.

Commanded or reached. Training on reached positions asks for where the arm got to, always a little short, so the robot moves sluggishly.



The absolute form depends on where the task happens to be; the relative form looks the same anywhere

A delta needs a reference

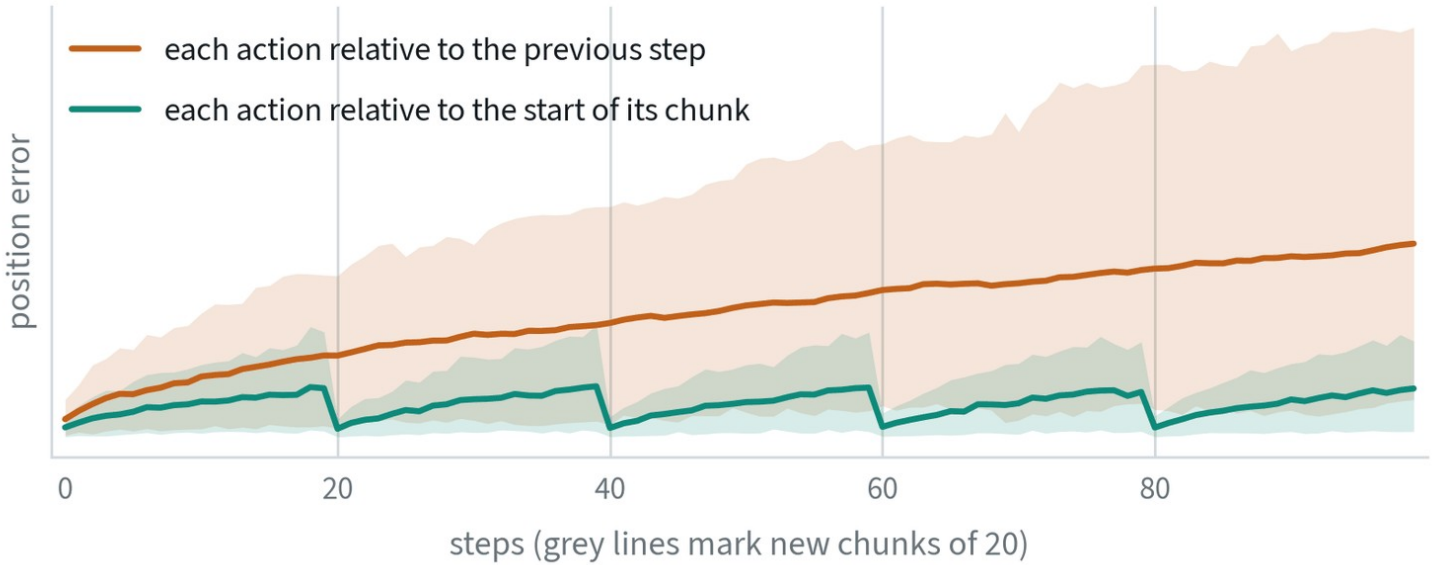
a simulation with the same small error on every predicted action, over 100 steps

Step to step. Each action is relative to the previous step, so the errors chain and add up without limit.

From the chunk start. Each action is relative to the pose at the moment of prediction. The error is wiped out at every 20-step chunk boundary.

Which frame. "Move 2 cm forward" in the base frame and in the gripper frame are different motions as soon as the gripper is tilted.

The gripper is its own small case: one number, open to closed, with the sign and the range differing between datasets.



Before training on any dataset, write down its answer to all four action-space questions

Normalise actions and state

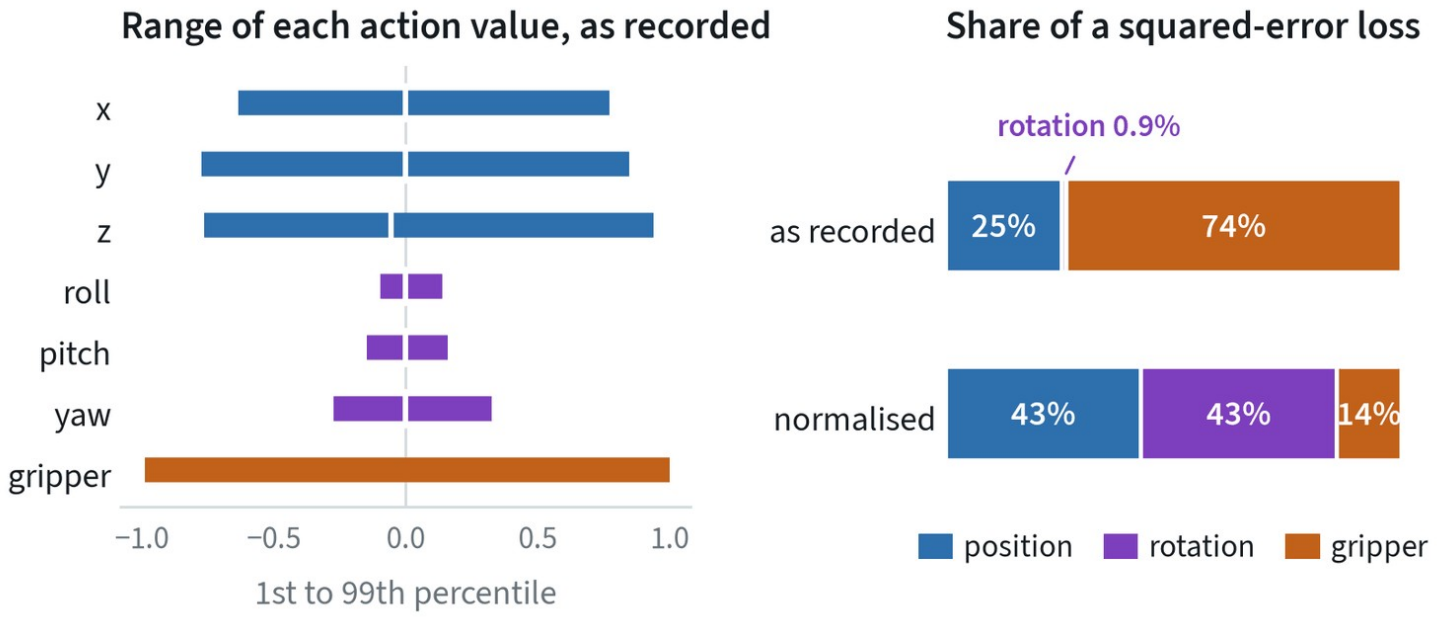
measured on 101,469 recorded actions from 379 LIBERO demonstrations

Unequal scales. The gripper command is −1 or +1, position values are a few tenths and rotation values a few hundredths.

As recorded, the gripper takes 74% of a squared-error loss, position 25% and rotation 0.9%. The wrist never learns to turn.

The fix. Rescale with training-set statistics: subtract the mean and divide by the spread, or map the 1st and 99th percentiles to −1 and +1.

Three rules. Save the statistics with the checkpoint, undo them on the way out, and normalise the body state too.



A model with the wrong statistics moves the robot smoothly to the wrong place

Behaviour cloning, and why errors compound

the dominant way to teach robots: record a person doing the task, then train the policy to copy

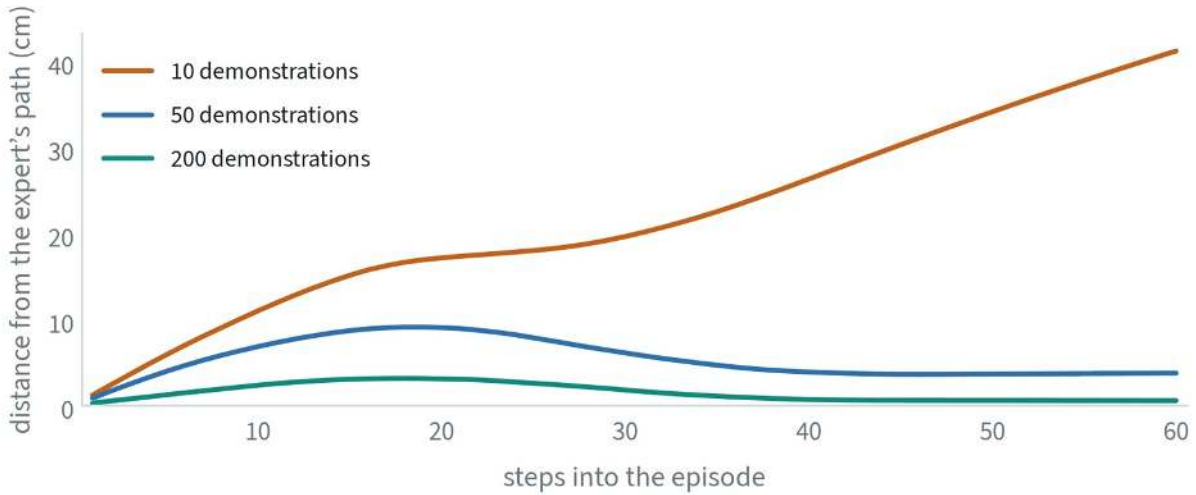
In symbols

$$\theta^* = \arg\max_\theta \; \mathbb{E}_{(o,a)\sim\mathcal{D}}\big[\log \pi_\theta(a \mid o)\big]$$

Maximum likelihood on the expert's states. With a fixed-variance Gaussian: squared error

cloning: $O(\epsilon T^2)$ with corrective data: $O(\epsilon T)$

Expected mistakes over T steps, for an error probability ε per step on expert states



Toy reaching task, mean of four runs and 300 starts. The gap to the expert reaches 41 cm at step 60 with 10 demonstrations, peaks at 9 cm with 50, stays under 3 with 200.

A one-step error of 1.8 cm becomes a 41 cm gap when the policy runs on its own

Action chunking

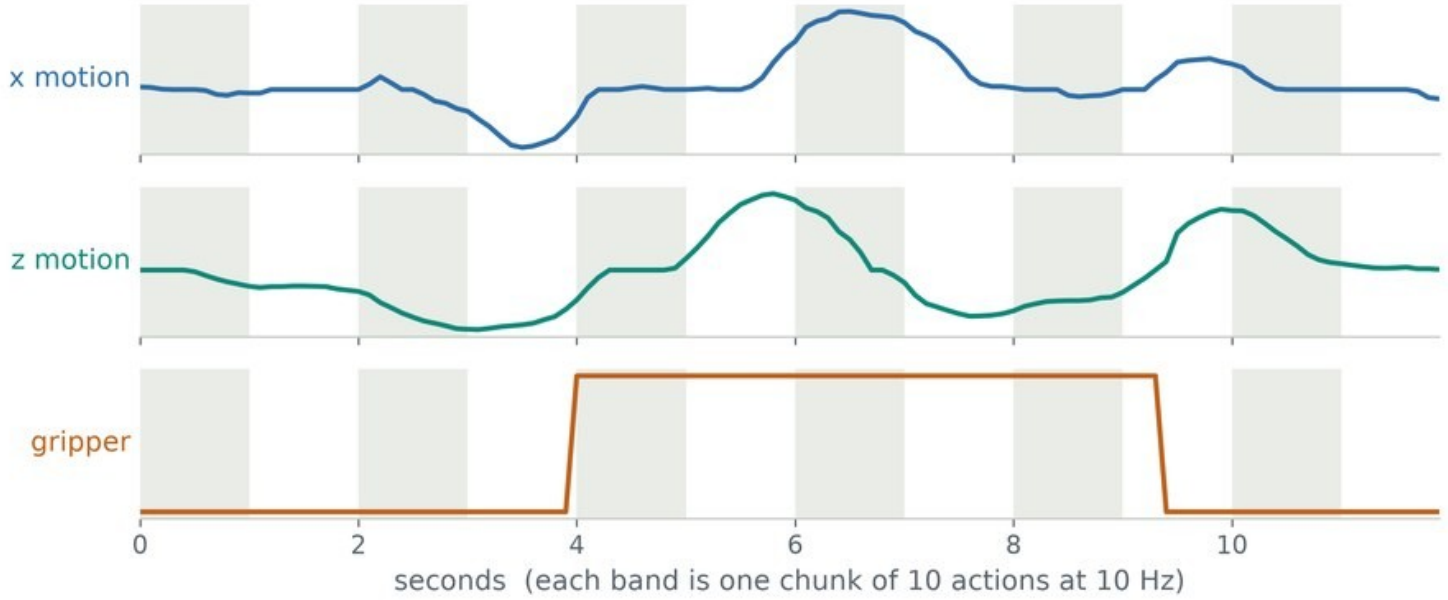
real actions from one LIBERO episode, x motion, z motion and gripper, cut into chunks of ten steps

One call, many actions. The policy predicts a short sequence of future actions instead of one.

Fewer decisions. A task takes fewer predictions, and the motion stays smooth.

Time to compute. A slow model works out the next chunk while the current one executes.

Stay responsive. Re-plan before the chunk ends.



Predicting a chunk is a simple trick, and a large part of why imitation learning started working

Chunking, measured

two parameters matter: the chunk length H, and how many actions run before the policy looks again

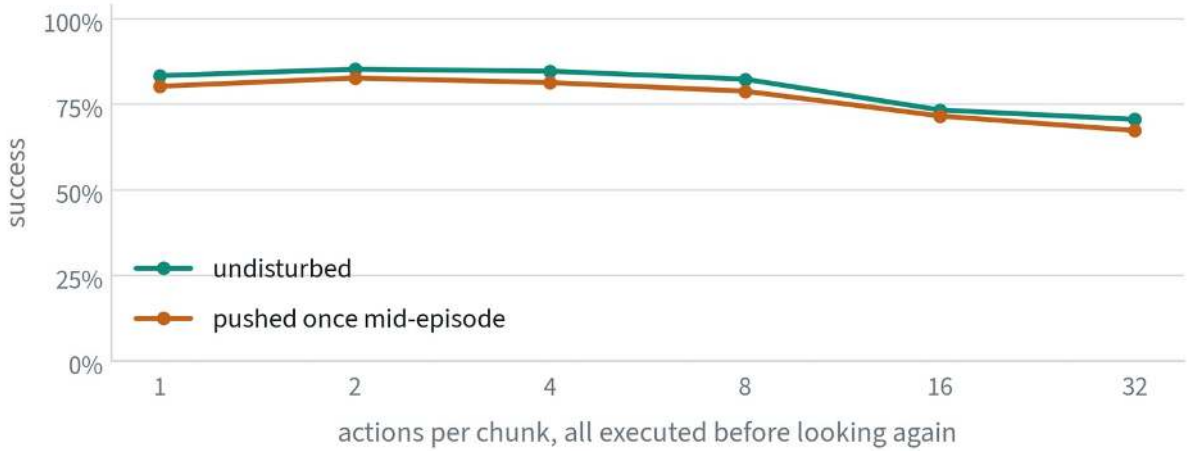
In symbols

$$\pi_\theta\left(a_{t:t+H} \mid o_t\right)$$

A chunking policy outputs H future actions from one observation

$$a_t = \frac{\sum_i w_i \, \hat{a}_t^{(i)}}{\sum_i w_i}, \qquad w_i = e^{-m\,i}$$

Temporal ensembling: average the overlapping predictions made i steps ago



Toy task, 60 demonstrations, every action in a chunk executed. Success is about 83 to 85% up to a chunk of 8 and 71% at 32. One push costs 3 to 4 points throughout.

On this toy, chunks of up to eight actions cost nothing and longer ones lose accuracy

History, reinforcement learning and the layer underneath

how much of the past to show, where trial and error fits, and what a learned policy sits on

Short history. One or two frames of images, the current body state and a predicted chunk. More frames invite "keep doing what I was doing".

RL needs trials. It learns from a reward by trial and error, so it works where a fast, accurate simulator exists: walking, running, balance.

Manipulation. Trials are slow or unsafe, rewards hard to write and contact hard to simulate. Imitation leads, with RL used afterwards.

The stack. Learned policy, then kinematics and planning, then the joint controller, then motors and safety limits.



Imitation leads in manipulation, RL in locomotion, and classical control runs under both

Common mistakes in robot learning

five errors, most of which raise no exception

Mixing action spaces

- The most common silent bug in robot learning.

Collecting only perfect demonstrations

- The policy then never sees how to recover from a mistake.

Judging by offline error

- Low error on recorded data says little about closed-loop behaviour.

Skipping or mismatching normalisation

- Check that the statistics used in training, saved in the checkpoint and applied on the robot are the same.

Chaining step-to-step deltas

- Over a long chunk the errors accumulate.
- Express a chunk's actions relative to its starting pose.

Check action space and normalisation first, and judge a policy in the loop

Reinforcement learning

Learning from reward by trial and error: where it works, why it is expensive, and how it is used with imitation.

KEY POINTS

Reinforcement learning needs no demonstrations, and pays for that in trials.

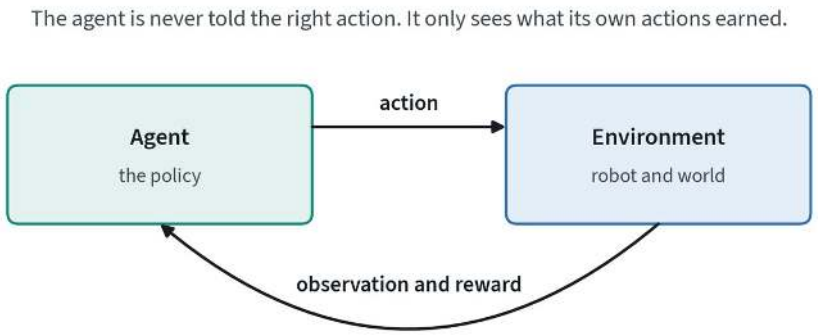
It leads where simulation is fast and accurate. Elsewhere it follows imitation.

The policy optimises the reward you wrote, not the task you meant.

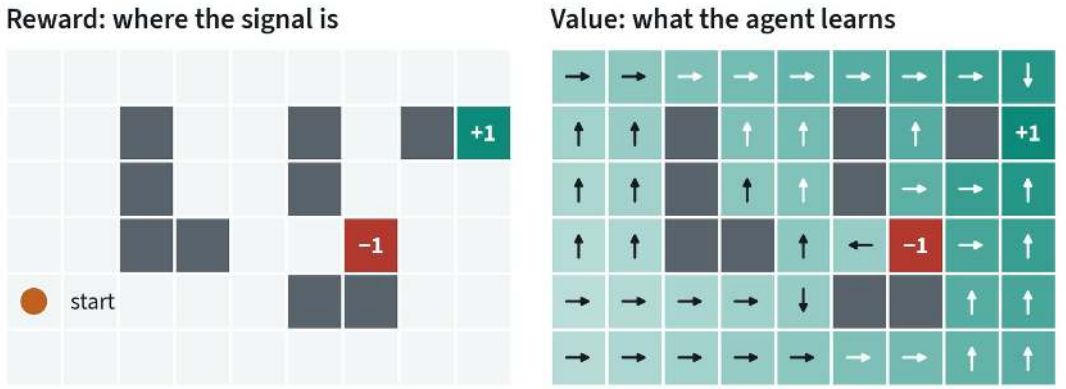
Run it: Reinforcement learning on a grid (github.com/wnukmat/robot-learning-notebooks)

Learning from reward

no demonstrator: the agent tries actions, receives a number called the reward, and adjusts to collect more



The whole setup: an action goes out, an observation and a reward come back. No labelled actions anywhere.



Value iteration on a small grid. Reward marks two cells; every cell gets a value and a best move.

Return. The agent maximises the sum of rewards over an episode, later ones usually counted a little less. It must explore to learn.

Value is the return expected from a state. Reward says what is good now; value says what is good in the long run.

The value function spreads a sparse reward over every state, and the policy follows from it

Three families of method

what each one learns, how it uses data, and where it is typically used

Family	What it learns	How it uses data	Typical use
Value-based (Q-learning)	The value of each action in each state	Reuses old experience	Discrete choices, games
Policy gradient (PPO)	The policy directly, nudged toward actions that did better than expected	Needs fresh experience after every update	Simulated locomotion at very large scale
Actor-critic, off-policy (SAC)	A policy and a value function together	Reuses old experience heavily	Real-robot learning, where every trial is costly

PPO needs fresh experience and runs in simulation; SAC reuses experience where trials are costly

The mathematics: return, Bellman and Q-learning

the setting is a Markov decision process: states s, actions a, a transition rule, a reward r(s, a) and a discount γ

In symbols

$$J(\pi) = \mathbb{E}_{\pi}\Big[\sum_{t=0}^{\infty} \gamma^{t}\, r(s_t, a_t)\Big]$$

The objective is the expected discounted return

$$Q^{*}(s,a) = r(s,a) + \gamma\, \mathbb{E}_{s'}\Big[\max_{a'} Q^{*}(s',a')\Big]$$

Bellman: immediate reward plus the discounted value of the best action afterwards

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$$

Q-learning: after each step, move the estimate toward the right-hand side

In words

Discount γ lies between 0 and 1.

Transition rule. P(s′ | s, a) gives the next state. Q-learning uses the observed next state in place of the expectation.

Temporal-difference error. The bracket: the gap between the estimate and the reward plus the discounted next estimate.

Used here. The grid experiments in this note run exactly this rule.

Q-learning is the Bellman equation turned into an update from single observed steps

Policy gradients and PPO

differentiate the objective with respect to the policy's parameters instead

In symbols

$$\nabla_\theta J = \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\ A(s_t, a_t) \right]$$

Raise each action's probability in proportion to its advantage

$$L^{\text{clip}} = \mathbb{E}\left[\min\left(\rho_t A_t,\ \text{clip}(\rho_t,\ 1-\epsilon,\ 1+\epsilon)\ A_t \right) \right]$$

PPO: a limit on how far one update may move the policy

$$\hat{A}_t = \sum_l (\gamma\lambda)^l\, \delta_{t+l}$$

The advantage is estimated as a weighted sum of temporal-difference errors

In words

Advantage. A(s, a) = Q(s, a) − V(s): how much better than expected the action turned out.

Ratio ρ. The probability of the action under the current policy divided by its probability under the old one.

λ trades bias against variance in the advantage estimate.

PPO is the method behind most simulated locomotion.

Raise the probability of actions that did better than expected, and limit each update

Why it needs so many trials

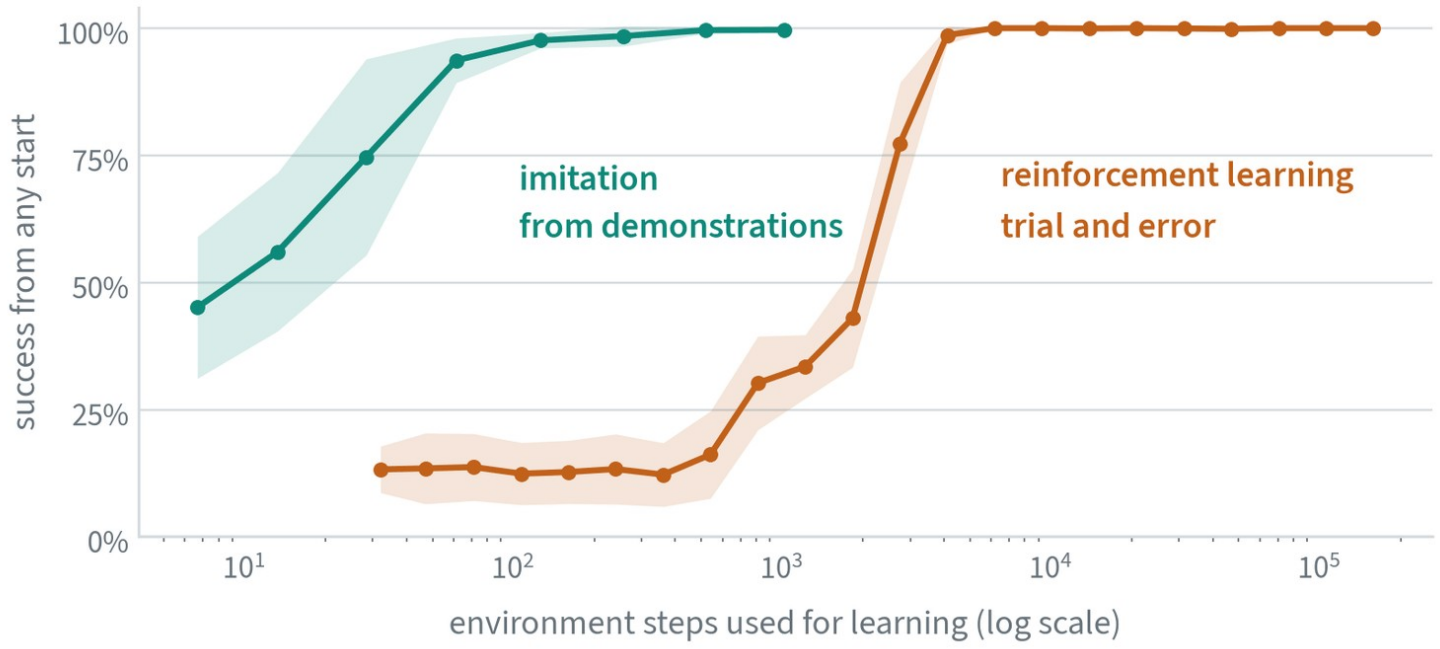
imitation and tabular Q-learning on the same 6 by 9 grid, averaged over 12 runs

Roughly 70 times more. Imitation reaches 90% success with about 60 steps of expert demonstration. Q-learning needs about 4,100 steps of its own.

Flat start. RL stays near 13% for hundreds of steps. Moving at random it almost never reaches the goal, so there is nothing to learn from.

It grows. A robot arm has a continuous state, a continuous action and a camera image, and the gap grows accordingly.

Not one-sided. Imitation needed an expert and can only match it. RL found the behaviour by itself and can go on to exceed the demonstrator.



The gap is the price of having no teacher

Where it works today: simulated locomotion

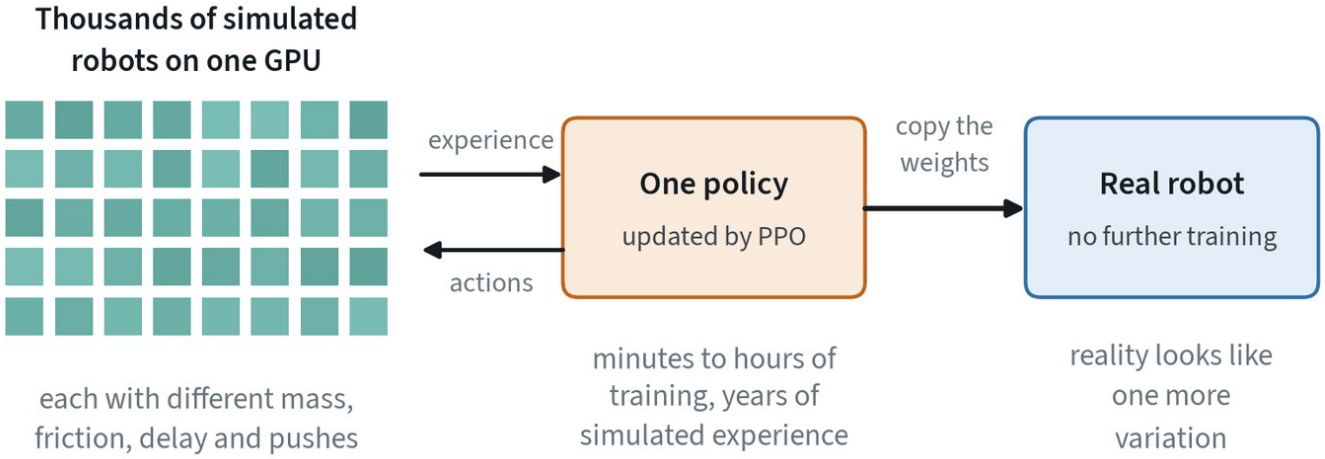
the standard recipe for walking and running robots

Parallel simulation. Thousands of copies of the robot run on one GPU, so years of experience take hours.

Domain randomisation. Each copy has different mass, friction, delay and pushes. The PPO policy transfers to hardware with no further training.

Three conditions. A fast simulator, foot and ground contact simple enough to simulate well, and a reward that is easy to write.

Manipulation fails all three. Deformable and small objects are hard to simulate, and "the table is tidy" is hard to turn into a number.



RL leads where simulation is fast and accurate, and follows imitation elsewhere

Used together with imitation

for manipulation, reinforcement learning is now mostly a second stage after imitation

Offline RL

- Learn from a fixed dataset with no new trials.
- It can use failed and mediocre episodes, because the reward tells it which were which.
- Its main danger is overrating actions the dataset never tried, so the methods stay close to the data.

Fine-tuning on the robot

- Start from an imitation policy, let it run, and reward success.
- A person steps in when it goes wrong, and those corrections become data too.
- Published systems report higher success and faster completion than the policy they started from.

Division of labour

- Imitation gets the policy to the point where it sometimes succeeds.
- That removes the long flat start of the reinforcement learning curve.
- Reward then polishes it.

Imitation removes the long flat start; reward then polishes the policy

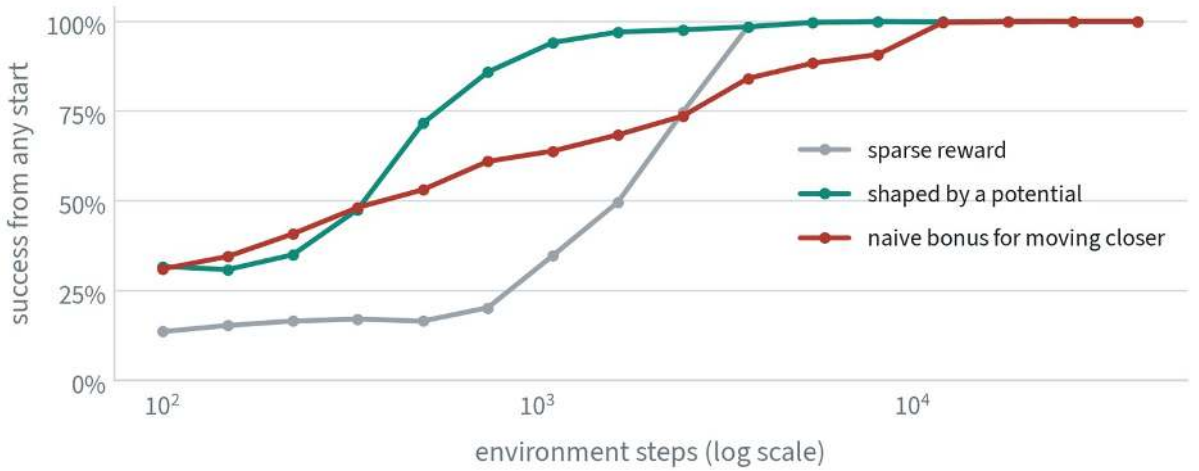
Reward shaping, measured

the same grid and the same Q-learning with three reward designs, mean of ten runs

The safe form

$$F(s, s') = \gamma\,\Phi(s') - \Phi(s)$$

Added to every reward. Φ is higher in better states: here, minus the distance to the goal



Sparse reward: under 20% for 700 steps, 99% after about 4,000. Potential-based: 94% after about 1,000. Naive bonus for moving closer: fast at first, then 84% against 98%.

Potential-based shaping speeds up learning and leaves the optimal policy unchanged

Reward is the hard part

four practical problems that sit outside the learning algorithm

Sparse or shaped

- A sparse reward, one point on success, is honest and slow to learn from.
- A shaped reward adds hints such as distance to the object. It is faster and easy to get wrong.

The agent optimises what you wrote

- If spinning in place earns shaped reward, the agent will spin.
- Watch the behaviour as well as the return.

Someone has to measure success

- On a real robot the reward comes from a person, a sensor or a learned success detector.
- Its mistakes become the policy's goals.

Resets and safety

- Each trial needs the scene put back, and exploration means trying things that fail.
- Both are cheap in simulation and expensive on hardware.

The policy optimises the reward you wrote, which may differ from the task you meant

Common mistakes with reinforcement learning

three errors of approach, each cheap to avoid

Reaching for RL first

- If demonstrations are available, start with imitation.
- Add reward when you need to exceed them.

Reading only the return curve

- A rising return can come from an exploit in the reward.
- Watch rollouts.

Trusting the simulator

- A policy can exploit simulator errors.
- Test on hardware early.

Start with imitation when demonstrations exist, and watch rollouts as well as curves

Robot policies today

How a model that understands images and language is turned into one that moves a robot.

KEY POINTS

- Generative action heads solved the "average of two good answers is a bad answer" problem.
- A VLA inherits what the internet taught the VLM, and learns only the acting part from robot data.
- Most of the parameters understand the scene; a small expert turns that understanding into motion.
- Token outputs keep the model simple; continuous heads give speed and precision. Many current systems use both.

Run it: Turning actions into tokens (github.com/wnukmat/robot-learning-notebooks)

Actions are generated, because regression averages

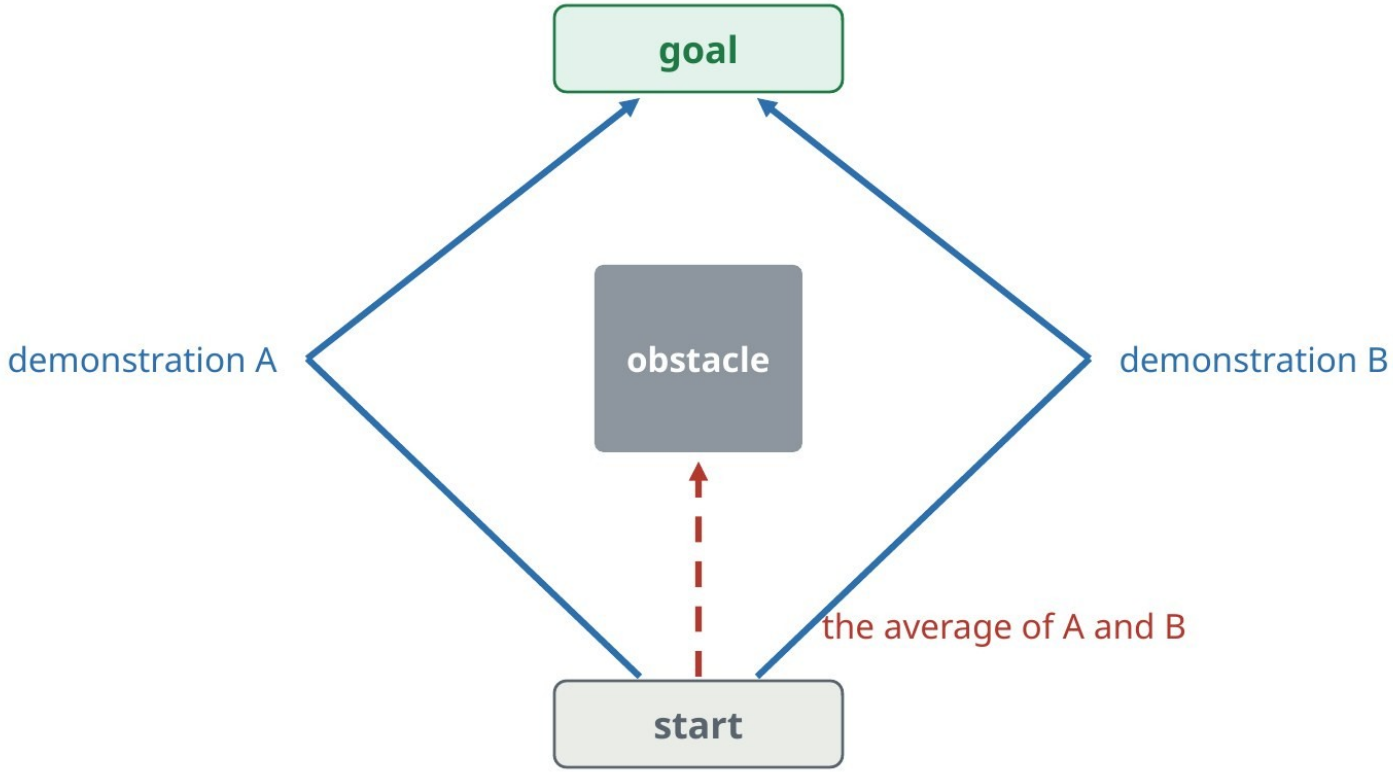
demonstrations contain more than one good answer, and mean squared error learns the average of them

Two good paths. One person steers left around the obstacle and another steers right. Their average goes straight through it.

Generation. Start from random noise and refine it into one consistent chunk of future actions, conditioned on recent images and robot state.

Diffusion Policy showed in 2023 that this gives large gains on manipulation tasks.

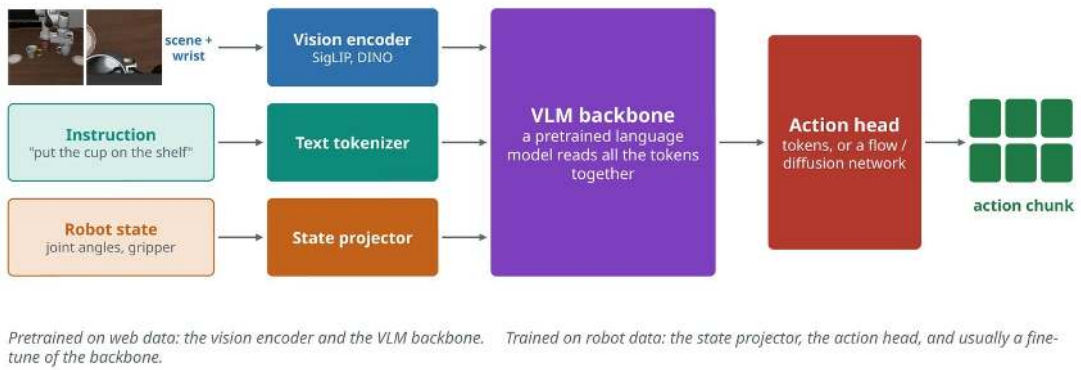
Flow matching does the same job in most recent systems because it needs fewer refinement steps.



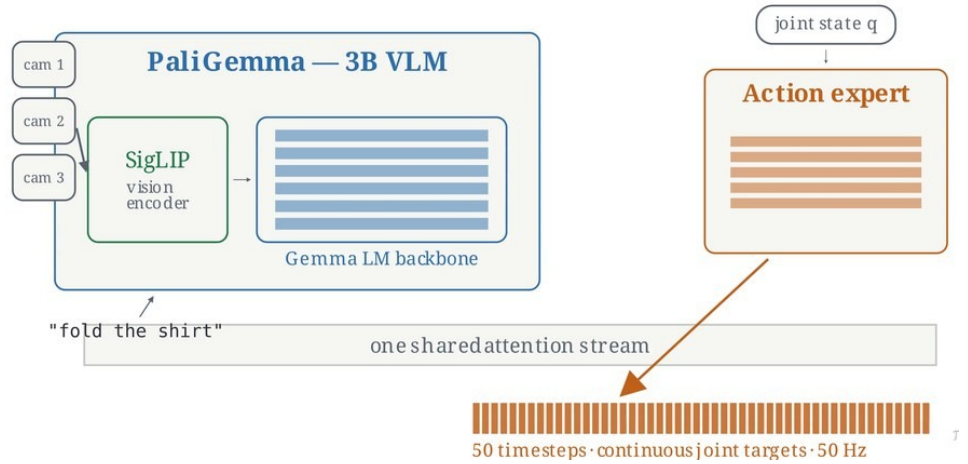
The average of two good answers is a bad answer

Anatomy of a vision-language-action model

a VLA is a vision-language model that has been taught to output robot actions



Inputs become tokens, a pretrained backbone reads them together, and an action head outputs a chunk.



π0 (Physical Intelligence, 2024): PaliGemma and a smaller action expert sharing one attention stream.

Division of labour. Encoder and backbone arrive trained on web-scale images and text. Robot data teaches only the acting part.

π0 in numbers. A backbone of about 3 billion parameters, an expert of about 300 million, and 50 actions in about ten steps: one second at 50 Hz.

Most of the parameters understand the scene; a small expert turns that understanding into motion

The flow-matching action head, written out

the chunk A holds H actions to predict; the observation o is images, instruction and body state

In symbols

$$A^\tau = (1-\tau)\,\epsilon + \tau A, \qquad \epsilon \sim \mathcal{N}(0, I)$$

A point on the straight line between noise and the true chunk

$$\mathcal{L} = \mathbb{E}_{A,\,\epsilon,\,\tau} \left\| v_\theta(A^\tau, \tau, o) - (A - \epsilon) \right\|^2$$

Training: the network outputs the direction of that line

$$A^{\tau + 1/K} = A^\tau + \tfrac{1}{K}\, v_\theta(A^\tau, \tau, o)$$

Run time: from noise, K Euler steps of size 1/K along the predicted direction

In words

Time τ is drawn between 0 and 1 for each training example.

Ten steps is typical. Each is a pass through the small action network only, because the observation is already encoded and cached.

Favour noisier τ. A common refinement over uniform sampling. For control, the coarse shape of the motion matters more than its last details.

Sampling costs K passes through the small action network; the observation is encoded once

Two ways to output actions

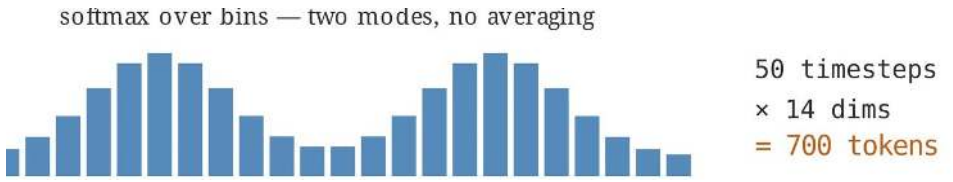
bins predicted like words, or a small network that refines a whole chunk from noise

Aspect	Discrete action tokens	Continuous action head
How	Sort each action value into a bin and predict the bins like words	A small network refines a whole chunk from noise, reading the backbone's features
Strengths	No new architecture; works with any language model; trains stably	Smooth and precise; a few steps yield many actions
Costs	One token per value is slow; bins limit precision	An extra module and loss; it must not degrade the backbone

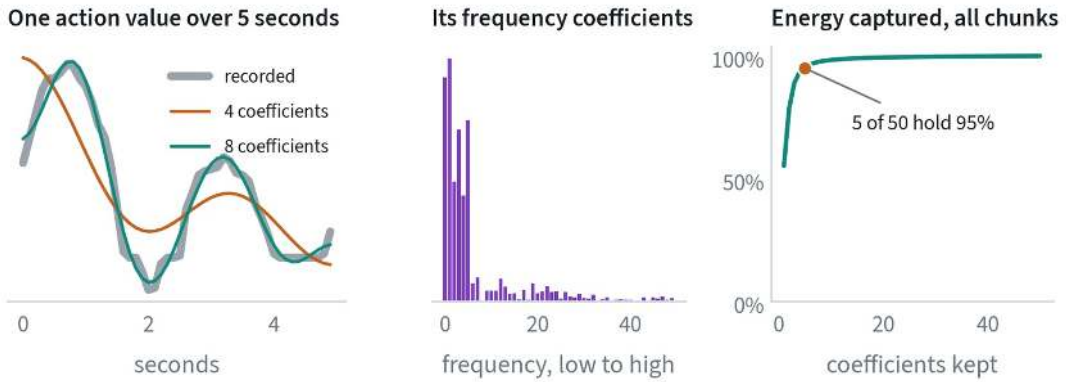
Neither has won outright, and several systems train with tokens but act with a continuous head

Action tokens, and compressing them

binning is precise enough; the trouble is length, and the remedy is a basis in which motion is sparse



Every value becomes a choice among bins. One second for a two-arm robot: 50 × 14 = 700 tokens.



200 LIBERO episodes under a discrete cosine transform: 5 of 50 coefficients hold 95% of the energy.

Bins. Mean rounding error on normalised LIBERO actions is 0.0015 with 256 bins and 0.026 with 16. A 50-step chunk of 7 values is 350 tokens.

Compression. Rounding the cosine coefficients leaves an average of 95 non-zero values out of 350, with an error of about 1% of the action range.

Frequency-space tokenisers let a token policy match flow matching on dexterous, high-rate tasks

One policy for many robots

no single robot has enough data, and most of what a policy learns is not about the body

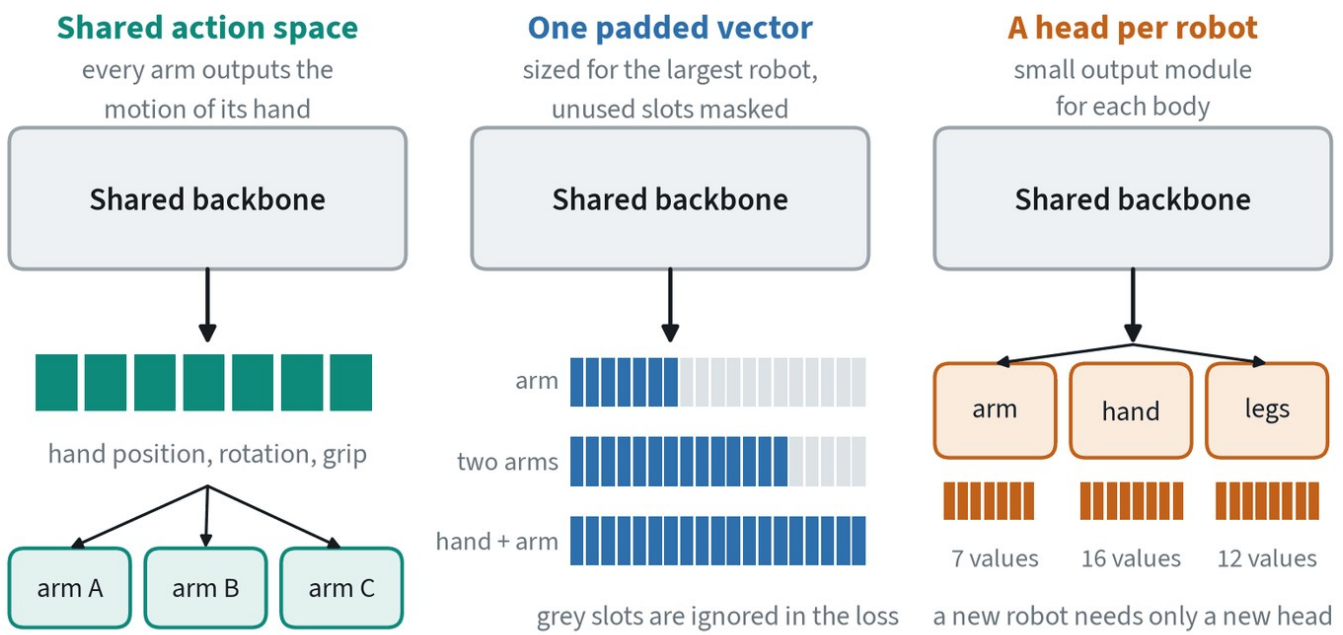
Three designs

Shared action space. Every arm outputs the motion of its hand in one seven-value format. It breaks down for hands, legs and mobile bases.

One padded vector. Size the action for the largest robot and mask unused slots. The model must learn from context which slots mean what.

A head per robot. Share the backbone and give each body a small input and output module. A new robot needs a new head and a little data.

Three details. Tell the model which robot it is, normalise each robot with its own statistics, and weight the mixture toward the target.



The body-specific part is the output, and the three designs are often combined

What transfers between robots

the realistic claim for cross-robot training is modest

Transfers well	Transfers poorly
Recognising objects and scenes	Reach and workspace
Understanding the instruction	Gripper or hand geometry
The order of steps in a task	Camera placement
Rough motion toward a target	Control rate, forces and contact

A new body needs far fewer demonstrations than starting from nothing

Keeping what the backbone knew

the action loss is new, its gradients are large and unlike anything in pretraining, and they flow into the backbone

Co-training

- Keep training on web image-and-text tasks alongside the robot data.
- The backbone is still held to what it knew.

Discrete targets for the backbone

- Train the backbone on tokenised actions with ordinary cross entropy, a loss of the kind it was pretrained with.
- Train the continuous action head on its features.

Stopping the gradient

- Do not let the action head's loss flow back into the backbone at all.
- The backbone learns from its own token loss, and the head adapts to whatever features result.

How the losses are routed matters as much as which architecture is used

Long tasks, and where the field is heading

a policy that maps the current image to the next second of motion can pick up a cup, and little more

Long tasks

- Clearing a table needs a plan and a memory.
- The methods are chunks, sub-task decomposition and reasoning before acting.

Dual-system designs

- A large, slow model understands the scene a few times a second.
- A small, fast model produces actions at control rate.

Embodied reasoning

- Before acting, the model states the sub-task, the objects involved and where they are.
- This helps on long tasks and makes failures easier to explain.

Cross-embodiment training

- One model is trained on data from many different robots, so a new robot needs less data of its own.

The direction of travel: one model, many robots, with slow thinking and fast acting separated

Common mistakes with robot policies

three errors that do not crash and do cost success

Mixing action spaces

- Absolute joint positions fed to a policy that expects relative hand motion will not crash.
- The robot will simply move wrongly.

Trusting offline error

- A policy with a low action error on held-out data can still fail when its own actions change what it sees next.

Adding refinement steps

- More steps cost latency, and a late action is a wrong action.

Check the action space, test in the loop, and count latency as part of accuracy

System 0, 1 and 2 architectures

Why robot policies are split into models that run at different speeds, what each layer does, and how they are connected.

KEY POINTS

- A slow model that understands and a fast model that reacts beat either alone.
- What the slow system hands to the fast one is the main design decision.
- System 0, 1 and 2 are published terms. “System 3” is informal: check what the author means.

One model cannot be both fast and thoughtful

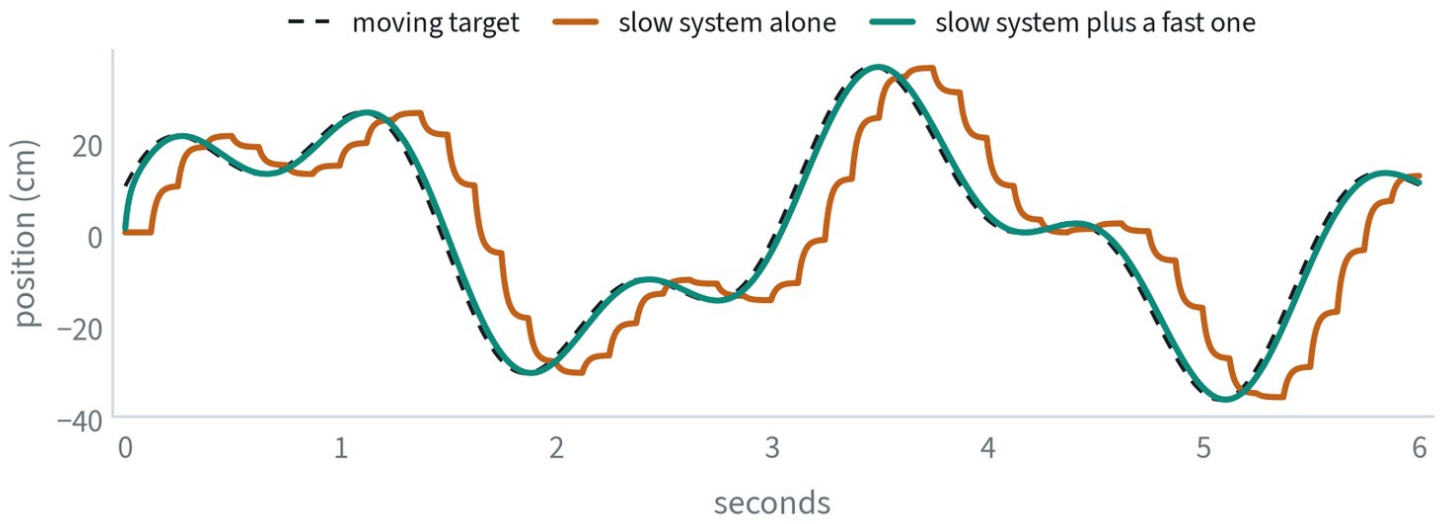
understanding takes a large model and a fraction of a second; correcting motion takes a small one, many times a second

The slow system alone. It looks 8 times a second and its answer arrives 120 ms late. It lags, moves in steps, and averages 10.4 cm of error.

With a fast loop beneath. A loop running 200 times a second under the same slow system brings the error down to 1.3 cm.

Out of date. A slow system acting alone is always answering a question that is no longer current.

The split. No single network does both jobs well, so the work is divided between models that run at different speeds.



A fast loop beneath the same slow system tracks eight times more closely

System 2, System 1, System 0

the layout and rates Figure has published for its Helix models; other systems use the same division with different numbers

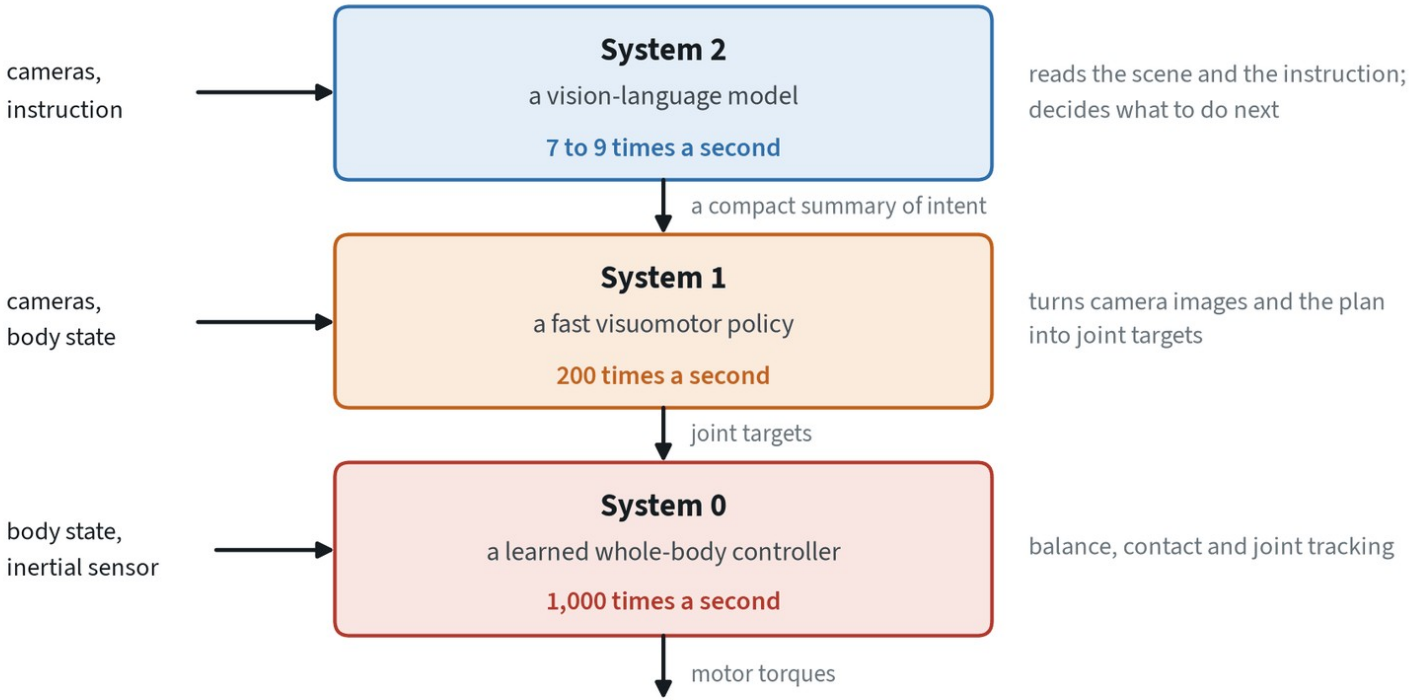
Three layers

System 2. A vision-language model takes the cameras and the instruction, runs 7 to 9 times a second, and passes down a summary of intent.

System 1. A fast visuomotor policy takes cameras and body state, runs 200 times a second, and outputs joint targets.

System 0. A learned whole-body controller takes body state and an inertial sensor, runs 1,000 times a second, and outputs motor torques.

A newer label. Most systems have the top pair. System 0 names a humanoid's balance and contact layer, once hand-engineered, now often learned.



Intent flows down as a compact summary, then joint targets, then motor torques

What each system is built from and trained on

the same three layers compared by job, construction, inputs, rate and training data

Aspect	System 2	System 1	System 0
Job	Understand the scene and the instruction; decide what to do	Turn vision and intent into motion	Keep the body balanced and the joints on target
Built from	A vision-language model	A smaller transformer, often with a flow-matching or diffusion action head	A small network trained by reinforcement learning in simulation, or a classical controller
Inputs	Cameras, language	Cameras, body state, System 2's output	Body state, inertial sensor, System 1's targets
Rate	A few times a second	Tens to hundreds of times a second	Around a thousand times a second
Trained on	Web data, then robot data	Robot demonstrations	Simulation

The slow layer learns from the web, the middle from demonstrations, the fastest from simulation

The rates on one axis, and the layer above System 2

decisions per second on a log axis, from the whole-body controller up to whatever sits above System 2

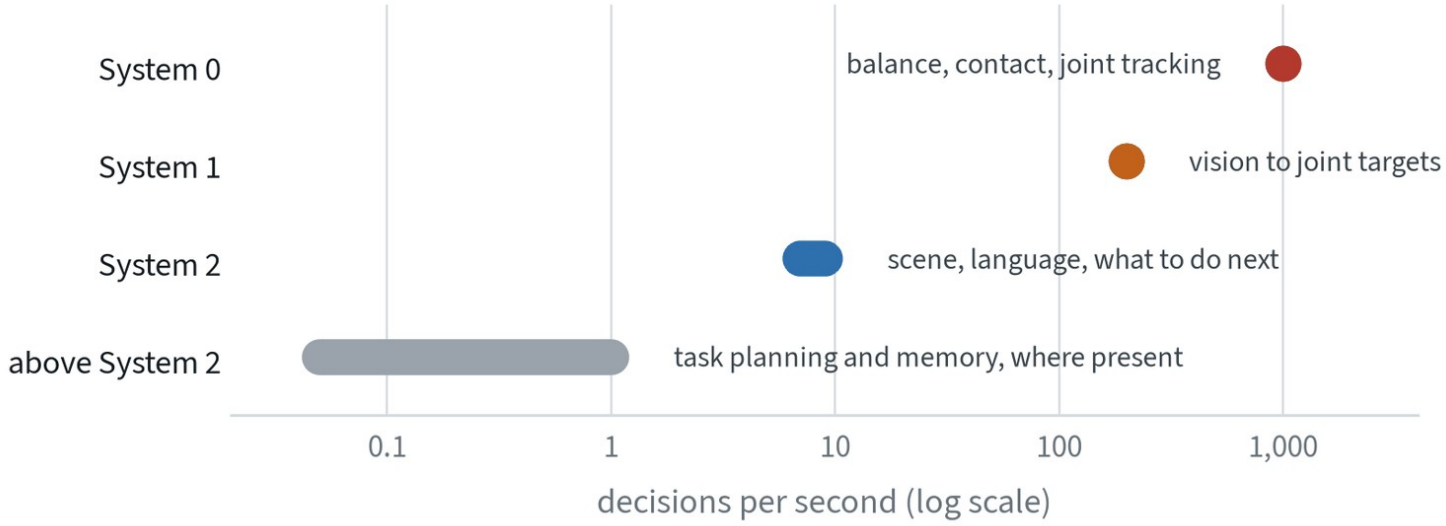
Rates and System 3

Ten to a hundred times. System 0 at 1,000 a second, System 1 at 200, System 2 at 7 to 9. Each is roughly that much faster than the layer above.

Above System 2. Where present, a layer decides once a second down to once every 20 seconds. Usually a large language model, often off the robot.

What lives there. Task planning over minutes, memory of what was done and seen, tools and knowledge, and coordination across robots or a shift.

Not an established term. The published systems found for this note describe Systems 0, 1 and 2. This layer is real and the least standardised.



System 3 is an informal label: check what the author means by it

How fast the fast loop needs to be

the tracking simulation swept over the rate of the fast loop and the delay of the slow system

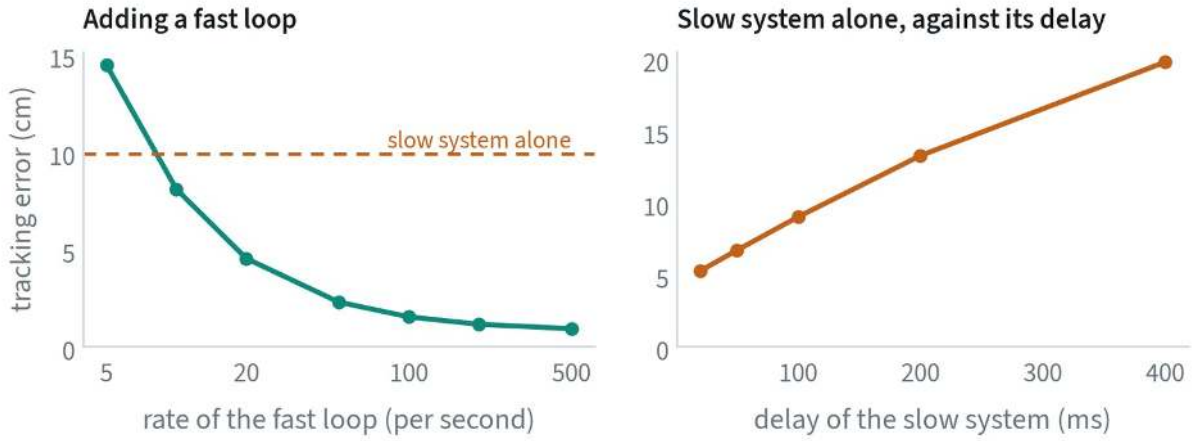
Delay sets the error

$$\text{error} \approx v\, d$$

A target moving at speed v, observed with total delay d, is mislocated by about v × d

$$d = d_{\text{sense}} + d_{\text{infer}} + \frac{1}{2f} + d_{\text{act}}$$

Camera, model, the wait for the next update at rate f, and the controller



Left: a fast loop at 5 a second gives 14.5 cm against 9.9 for the slow system alone; at 200 it gives 1.1. Right: slow system alone, 5.2 cm at 20 ms delay, 19.7 at 400 ms.

A fast loop that is too slow is worse than none; the gain starts near 10 a second and is mostly in by 100

The interface is the design

what System 2 hands to System 1 decides most of how a dual system behaves

Interface	What is passed down	Strength	Weakness
Language	A sub-task as text	Readable; the two models can be trained separately	Narrow: text cannot say exactly where or how
A latent vector	A learned summary of the scene and intent	Rich, and trained for the purpose	Opaque; the two halves must be trained together
Internal features	System 1 attends directly to System 2's layers	Nothing is lost in a bottleneck	Tightly coupled; System 1 depends on the exact System 2
Goals	A target image, points or a trajectory sketch	Concrete and checkable	Needs a model that can produce good goals

Shared attention blurs one model and two: the parts run at different rates and are trained as one

How the two systems are trained

four recipes, from fully separate to one loss through both systems

Recipe	How	Trade-off
Separately	Low-level policy on demonstrations labelled with sub-tasks; a vision-language model as planner	Simple and modular; the planner does not know what the policy can do
Jointly, end to end	One loss on actions flows back through both systems	The interface is learned for the task; the two halves cannot be swapped
Jointly, with insulated gradients	Slow model trained on its own token loss, fast one on actions, sharing features	Keeps the slow model's knowledge; needs care in routing the losses
Staged	Pretrain the slow model, freeze it, then train the fast model on its outputs	Stable; the slow model's features were not shaped for control

Whatever the recipe, train the fast system with the delay it will see in deployment

Running at different speeds

five consequences of putting two loops with different rates in one robot

The fast system uses a stale plan

- System 1 acts many times on each output of System 2.
- Untrained for that delay, it expects fresher guidance than it gets.

The fast system needs its own eyes

- If only System 2 sees the cameras, the fast loop is blind between updates.
- In the simulation, the fast loop's own view of the target closes the gap.

Asynchronous by design

- The two run in separate loops.
- System 1 uses whatever the latest output is.

Caching

- System 2's result is computed once and reused, as the deployment note measures.

Separate hardware

- The large model can run on a bigger processor, or off the robot.
- The fast loops stay on board and keep working if the link drops.

System 1 never waits for System 2, and it needs its own view of the scene

One model or several

what is gained and what is paid when a policy is split into systems

Aspect	One model	Two or three systems
Simplicity	One network, one training run	Interfaces, timing and several training recipes
Reaction time	Limited by the whole model	Set by the fast system
Reasoning	As good as the model, at every step	As good as System 2, a few times a second
Failure analysis	Hard to localise	A failure can often be assigned to a layer
Typical use	Arms at moderate rates, with action chunks covering the gap	Humanoids and anything needing balance or fast contact

A dual system makes the fast part closed-loop; the field has moved that way as tasks demand quicker reactions

Common mistakes with layered systems

three errors that follow directly from the timing and the naming

Giving only the slow system the cameras

- The fast loop then cannot react to anything it was not told about.

Training the fast system on fresh plans

- Train it with the delay it will see when deployed.
- A random offset between the two systems during training is the usual fix.

Assuming the labels are standard

- System 0, and anything called System 3, mean different things to different groups.

Give the fast loop its own cameras and train it on plans of realistic age

Long-horizon tasks

Action chunks, sub-task decomposition, chain of thought and memory: how robots get through tasks with many steps.

KEY POINTS

- Success on a long task is the product of its steps: ten steps at 95% is 60%.
- Chunks shorten the chain at the scale of seconds; sub-tasks at the scale of minutes.
- Reasoning before acting makes each decision more reliable and each failure easier to read.

Success on a long task is the product of its steps

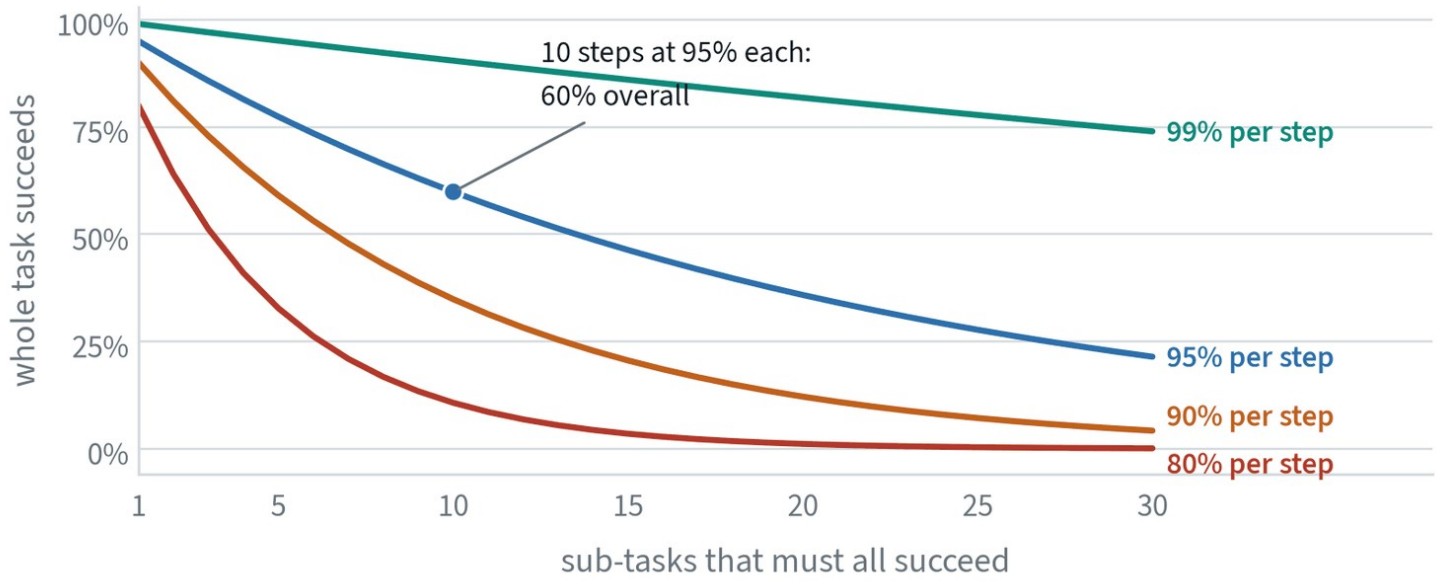
arithmetic, not an experiment: if every sub-task must succeed, the per-step rates multiply

95% per step sounds reliable. Ten such steps succeed 60% of the time, and thirty succeed 21%.

99% per step still gives only 74% over thirty steps.

Memory. The policy has to remember what it has already done, and it sees only the last frame or two.

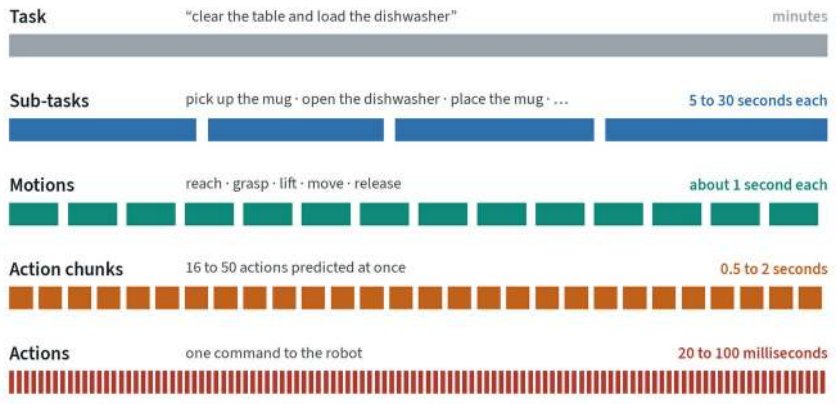
Vague instructions. The instruction no longer describes the next motion. "Clear the table" does not say which object to pick up first.



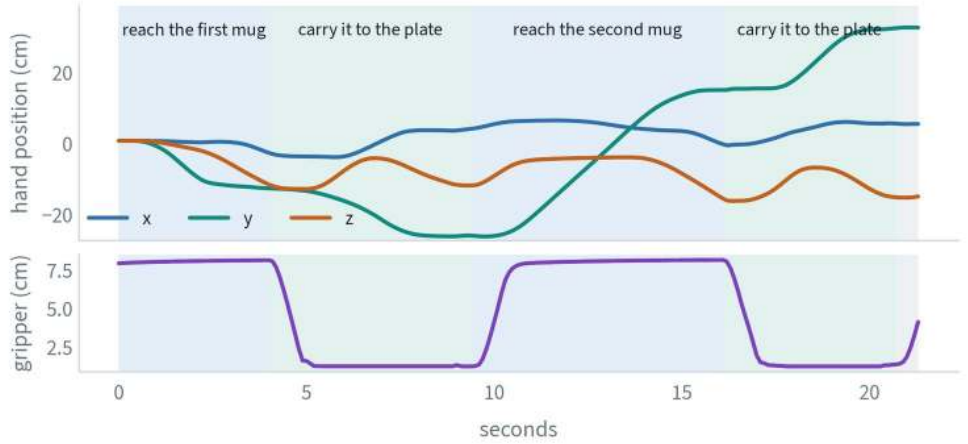
Three ways to attack it: make each step more reliable, make fewer steps, or recover when a step fails

One task at five levels

each method in this section works at one of the boundaries between these levels



Task: minutes. Sub-tasks: 5 to 30 s. Motions: about 1 s. Chunks: 0.5 to 2 s. Actions: 20 to 100 ms.



A real 21 s episode. The gripper closes at 4.0 s and 16.1 s and opens at 9.4 s and 20.7 s.

Four phases from one signal. The gripper command alone splits the episode: reach the first mug, carry it to the plate, reach the second, carry it.

Sub-task decomposition for free. The boundaries are already in the data, so this simplest form needs no labelling.

The simplest sub-task boundaries are free: the moments the gripper closes and opens

What long horizon means in a standard benchmark

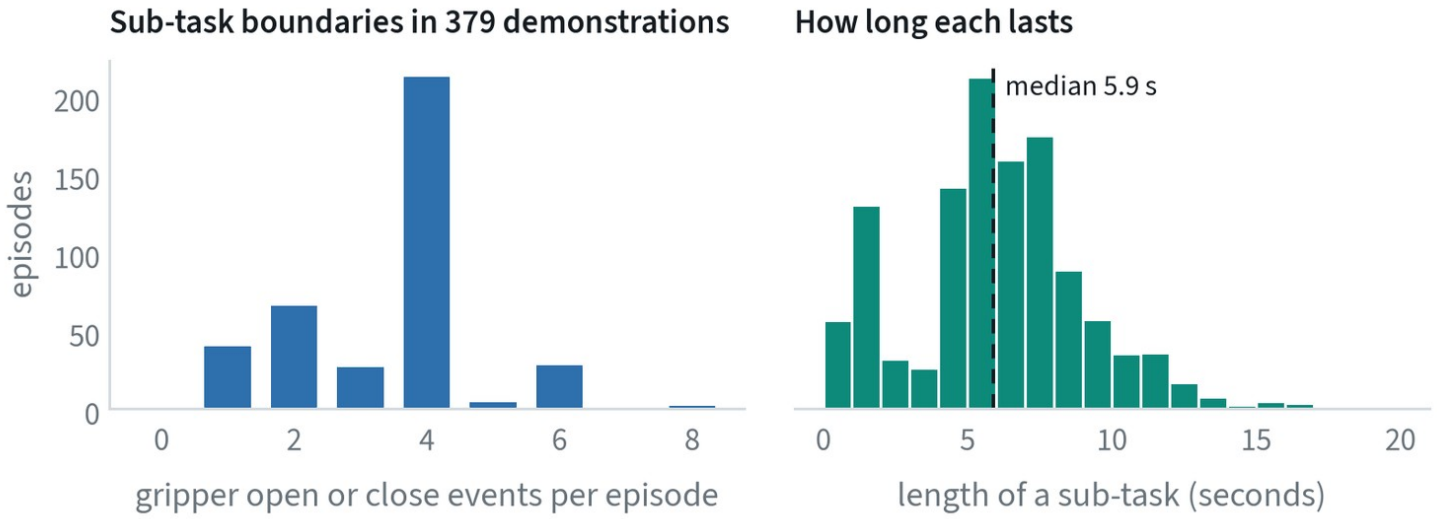
the same gripper-based segmentation applied to all 379 LIBERO demonstrations

Four events is typical. 212 of 379 episodes have four gripper events, which is two pick-and-place sub-tasks. 66 have two, 40 have one, 28 have six.

Short sub-tasks. The median sub-task lasts 5.9 seconds and the median episode 26.

In actions. At 10 actions a second that is roughly 60 actions per sub-task, or four chunks of 16.

The gap. Tasks that people consider simple, such as loading a dishwasher, run to dozens of sub-tasks.



Benchmark long-horizon tasks are two or three sub-tasks long; household tasks run to dozens

Action chunks: fewer, larger decisions

predicting a block of future actions in one call shortens the chain directly

The 214-step episode in chunks

- One action per call is 214 decisions. Chunks of 8 give 27, chunks of 16 give 14 and chunks of 50 give 5.
- A chunk of 16 covers 1.6 seconds; a chunk of 50 covers 5.

What it buys and what it costs

- Fewer decisions mean fewer chances to drift, and one pass gives smooth motion.
- Within a chunk the policy is not looking.

Execute part, re-plan early

- Predict 16 actions, execute 5, and predict again.
- The plan stays long and the reaction time stays short.

Blend overlapping chunks

- Average the predictions that successive chunks make for the same moment, which smooths the joins.

Plan around the delay

- Generate the next chunk knowing which of the current actions will already have run when it arrives.

Chunking solves the problem at the scale of seconds and does nothing for a task of minutes

Sub-task decomposition

a high-level model names the next sub-task in words and a low-level policy carries it out

Where sub-tasks come from	How it works	Trade-off
Written by hand	A fixed script or state machine lists the steps	Reliable and inflexible; one script per task
A language model as planner	Proposes the next step from the instruction and scene, often filtered by what the robot can do	Flexible; the planner can propose steps the robot cannot perform
A vision-language model trained for it	Fine-tuned on robot data labelled with sub-tasks; predicts the next one from the camera view	Grounded in what the robot sees; needs sub-task labels
Learned from the data's structure	Boundaries found automatically, from gripper events or from changes in the scene	Free labels; the segments may not match meaningful steps

A hierarchy works only if the high level knows when a sub-task is finished and can re-plan when it fails

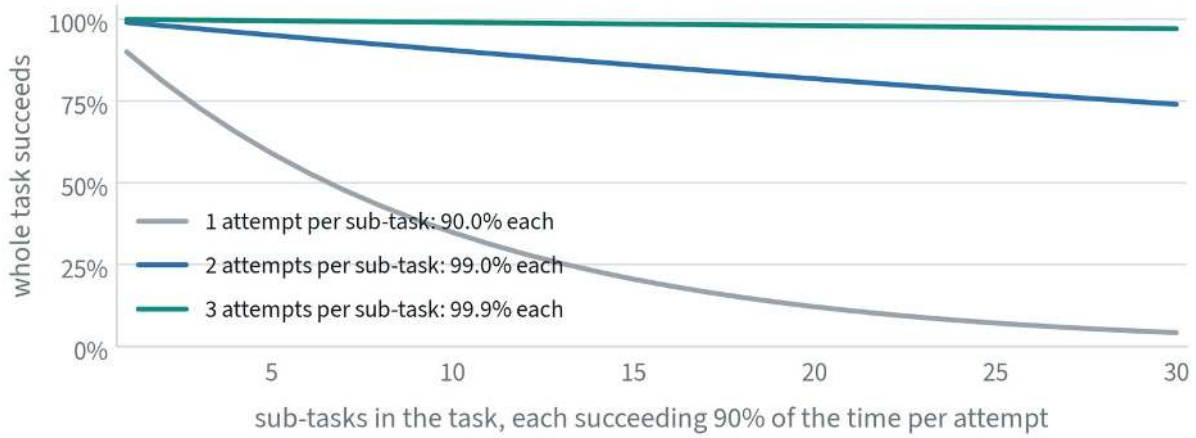
What recovery is worth

the compounding arithmetic again, with each sub-task allowed more than one attempt

With retries

$$p_{\text{eff}} = 1 - (1 - p)^k$$

Effective success of a sub-task with per-attempt rate p and up to k attempts



Thirty sub-tasks, each succeeding 90% of the time per attempt: 4% whole-task success with one attempt, 74% with two and 97% with three.

A second attempt turns 90% into 99%; no plausible improvement to the low-level policy does as much

Chain of thought: reasoning before acting

the model writes its intermediate reasoning as tokens before the action, and each step conditions the ones after it

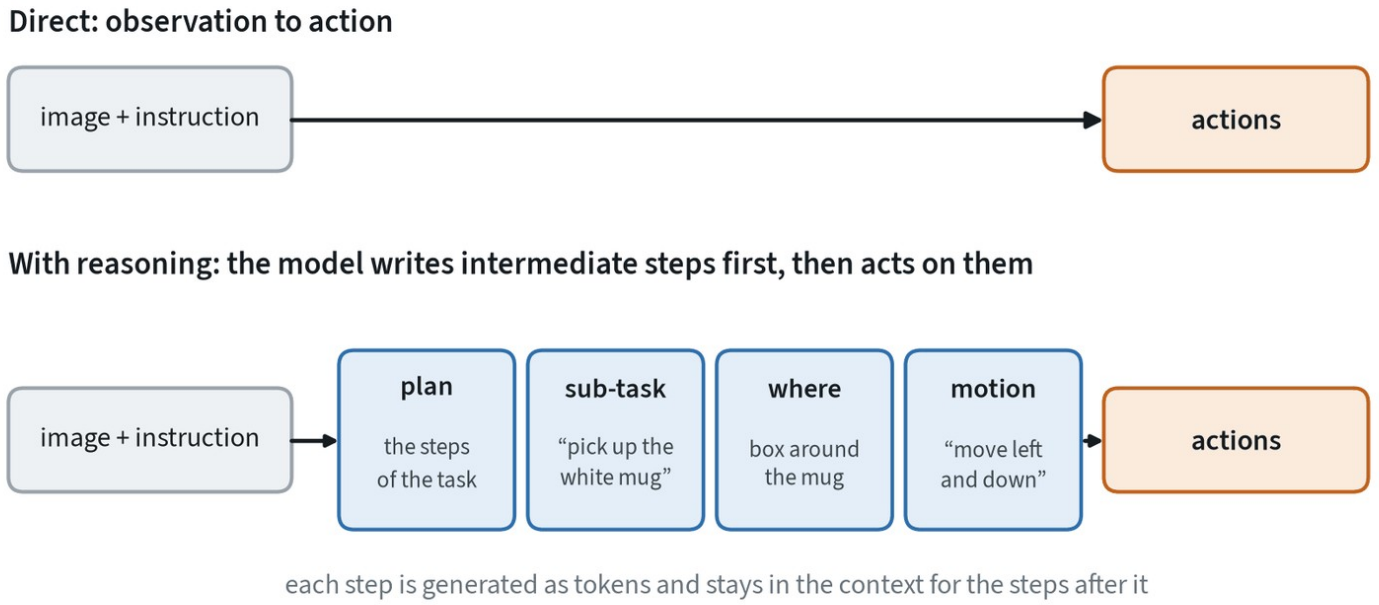
Embodied reasoning

Four steps first. The plan, the current sub-task, the object's location as a box, and the motion in words. Only then the action.

Reported gains. Published work reports clear gains in success on tasks that need generalisation.

Readable failures. The reasoning shows where it went wrong: the wrong sub-task, the wrong object, or the right intention and a bad motion.

Costs. The steps must be labelled in the training data, every token takes time, and fluent reasoning can precede an unrelated action.



Reasoning makes each decision more reliable and each failure easier to read

Memory

a policy that sees only the present cannot know whether it has already checked a drawer

A longer context

- Feed more past frames.
- Simple, expensive, and it invites the shortcut of copying past actions.

A written log

- The high-level model keeps short notes on what has been done, and reads them at each decision.

Keyframes

- Store a few informative past frames in place of all of them.

A recurrent state

- Carry a fixed-size summary forward in time, as the linear recurrent layers in the attention note do.

Memory is the least settled part: tasks that depend on something no longer visible remain hard

How the methods fit together

each method works at a different level and fights compounding in a different way

Method	Works at	Fights compounding by	Costs
Action chunks	Actions to motions	Fewer decisions	Reactivity within a chunk
Sub-task decomposition	Motions to task	Shorter chains per policy call; a place to re-plan	A second model, and a way to detect completion
Chain of thought	Inside one decision	Making each decision more reliable	Time, and labelled reasoning
Recovery and retries	Any level	Turning a failure into extra steps, where it would have been the end	A failure detector
Memory	Across the task	Avoiding repeated and skipped steps	Context, or added machinery

Current systems use most of these at once

Common mistakes with long tasks

four errors in how long-horizon systems are built and reported

Quoting per-step success

- Report success on the whole task.
- The chain is what matters.

Lengthening chunks to fix a long task

- Chunks help at the scale of seconds.
- Minutes need decomposition.

A hierarchy with no completion check

- The high level must know when to move on.

Assuming the reasoning caused the action

- Test it by editing the reasoning.

Report success on the whole task, and check that the system knows when a step has failed

World models

Models that predict what happens next, and models that predict and act together.

KEY POINTS

A policy answers "what should I do?". A world model answers "what will happen if I do this?".

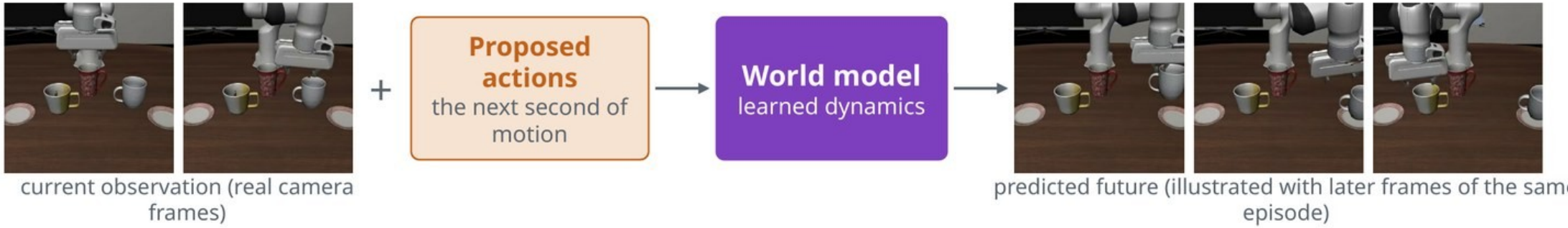
The bet: a model that can predict what its actions will cause has learned something a pure policy has not.

A world model is only as useful as it is faithful; always check it against real outcomes.

Run it: The baseline every world model must beat (github.com/wnukmat/robot-learning-notebooks)

Learning to predict what happens next

a world model takes the current situation and a proposed action and predicts the outcome



Action-conditioned video. The model takes recent frames and a sequence of robot actions and generates the frames that would follow.

An old idea at a new scale. Control has long planned with a model of the dynamics. The new part is learning it from video at scale.

Why robots want it. Actions can be tried in imagination before the real world, and plentiful video teaches physics with no action labels.

A high bar. The model must get contact, occlusion and object permanence right.

The model must show the consequences of the action, not just plausible video

The objectives

with o an observation, a an action, z = enc(o) a compact latent state and H the prediction horizon

Three losses

$$p_\theta\left(o_{t+1:t+H} \mid o_{\le t},\ a_{t:t+H}\right)$$

In pixels: a conditional generative model of future frames given actions

$$\mathcal{L}_{\text{latent}} = \left\| f_\theta(\text{enc}(o_t), a_t) - \overline{\text{enc}}(o_{t+1}) \right\|^2$$

In latents: regress the next state against a slow-moving target encoder

$$\mathcal{L}_{\text{WAM}} = \mathcal{L}_{\text{video}}(o_{t+1:t+H}) + \lambda\, \mathcal{L}_{\text{action}}(a_{t:t+H})$$

World-action model: one network, a joint loss on future frames and actions

In words

Pixel form. Usually trained by diffusion or flow matching in an autoencoder's latent space, with the actions as conditioning.

Latent form. The dynamics are $z_{t+1} = f(z_t, a_t)$, trained by regression in representation space.

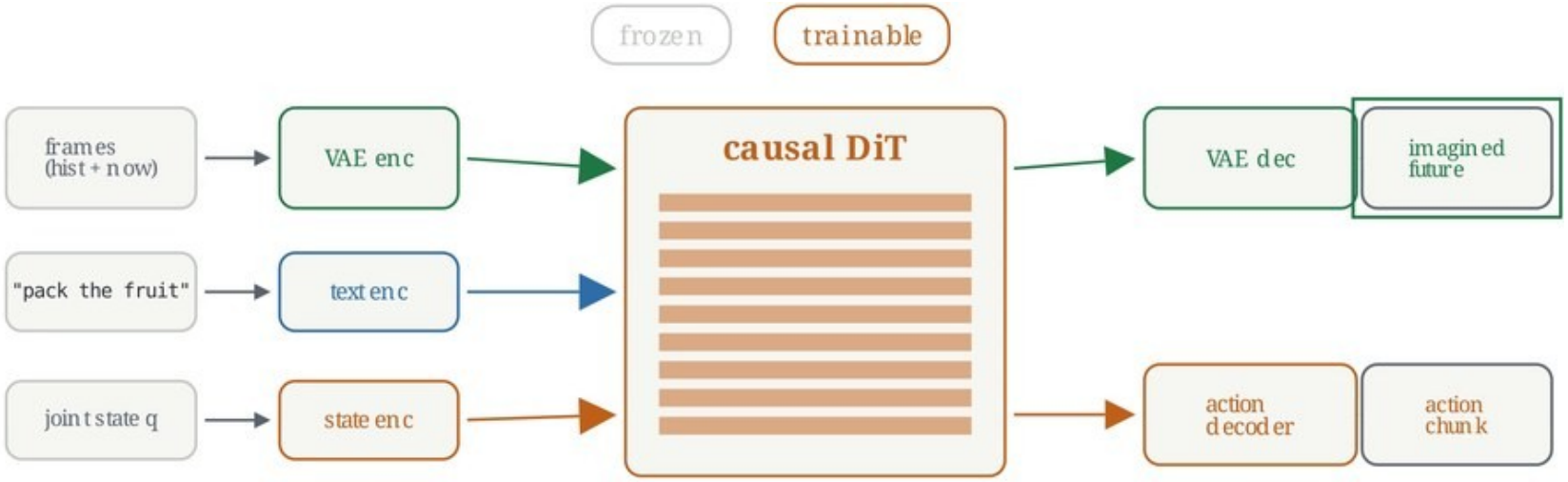
Joint form. Both terms are denoising losses of the same form, applied to different tokens of one sequence.

Inverse dynamics is the conditional the other way round, $p(a_t \mid o_t, o_{t+1})$. It is easier to learn because both ends of the motion are visible.

Predict pixels, predict latents, or predict the future and the actions together

World-action models: predict and act in one model

wiring of one published world-action model, DreamZero; older designs kept the policy and the world model separate



The wiring. Frames, instruction and joint state enter a causal diffusion transformer, which outputs an imagined future and an action chunk.

The bet. A model that can predict what its actions will cause has learned something a pure policy has not.

Denser signal. Predicting the future trains on every pixel or latent value of every frame. Imitating actions gives a few numbers per step.

What to predict. Pixels are easy to inspect and spend capacity on appearance. Latents are cheaper and focused, and harder to check.

Treat it as a bet and not a settled answer: the results are still coming in

Getting actions out of video

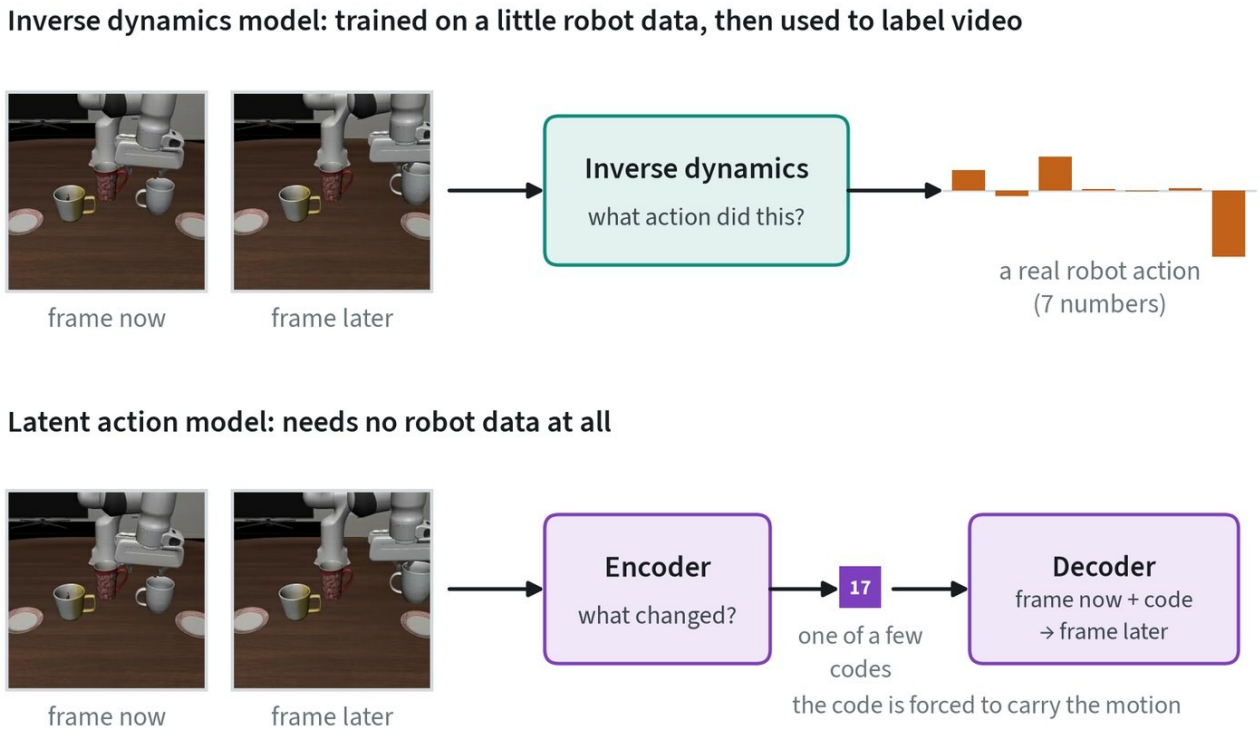
video is plentiful and has no action labels; two methods supply them

Inverse dynamics. Train on a little robot data to infer the action between two frames, here seven numbers, then run it over unlabelled video.

Easier than control. The inverse dynamics model sees the outcome.

Latent actions. An encoder turns the change between two frames into one discrete code. A decoder rebuilds the later frame from the earlier one and the code.

Invented codes. The codes come to mean things like "move left" or "close". A pretrained policy is mapped to real actions with a little robot data.



Both turn human and internet video into training data, and neither recovers force

Planning inside the model

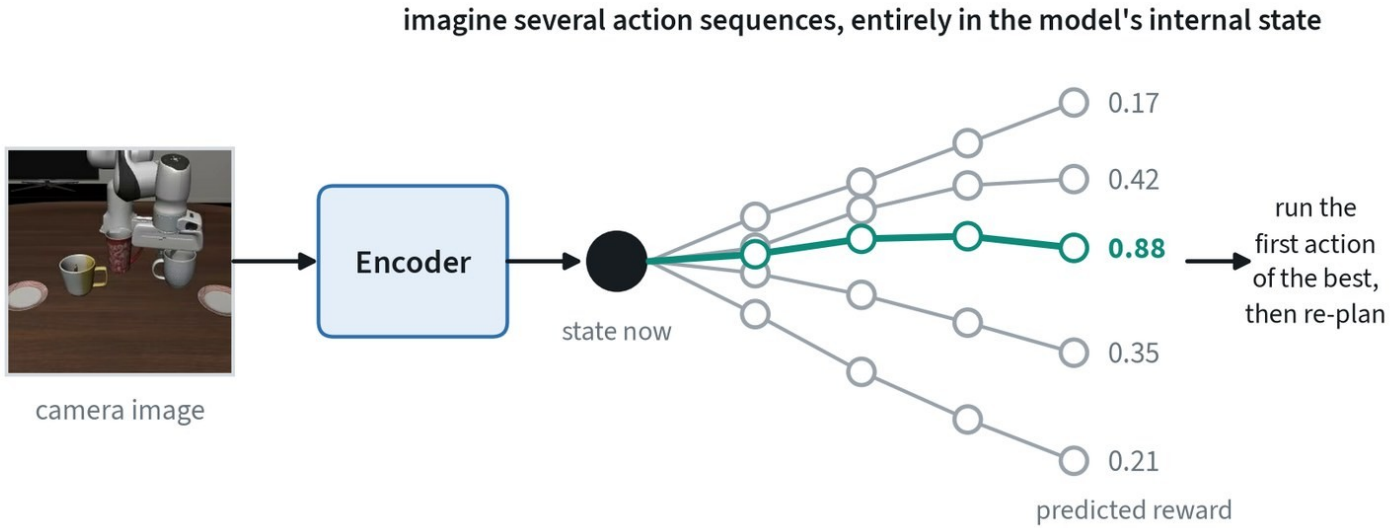
a latent dynamics model encodes the observation into a small state and predicts the next state and the reward directly

No video needed. The rollouts happen in the model's compact internal state, without rendering a single frame.

Plan at run time. Sample action sequences, score them in the model, execute the first action of the best, and repeat.

A known method. That loop is model-predictive control with a learned model.

Train in imagination. The other established use in reinforcement learning trains a policy entirely on imagined rollouts.



With no pixels to generate, thousands of imagined futures cost little

In manipulation video, almost nothing moves

two real frames one second apart, and maps of which pixels changed after one and after five seconds



After one second only the region of the robot arm has changed.

After five seconds the arm and the object it moved have changed. The rest of the map is still black.

Static background. The table, the stove and the background do not move.

What this implies. A useful evaluation weights or crops to the region that moves.

The pixels that change are the ones a policy cares about, and the hardest to predict

The baseline every world model must beat

measured on 60 real episodes: the score earned by copying the current frame forward

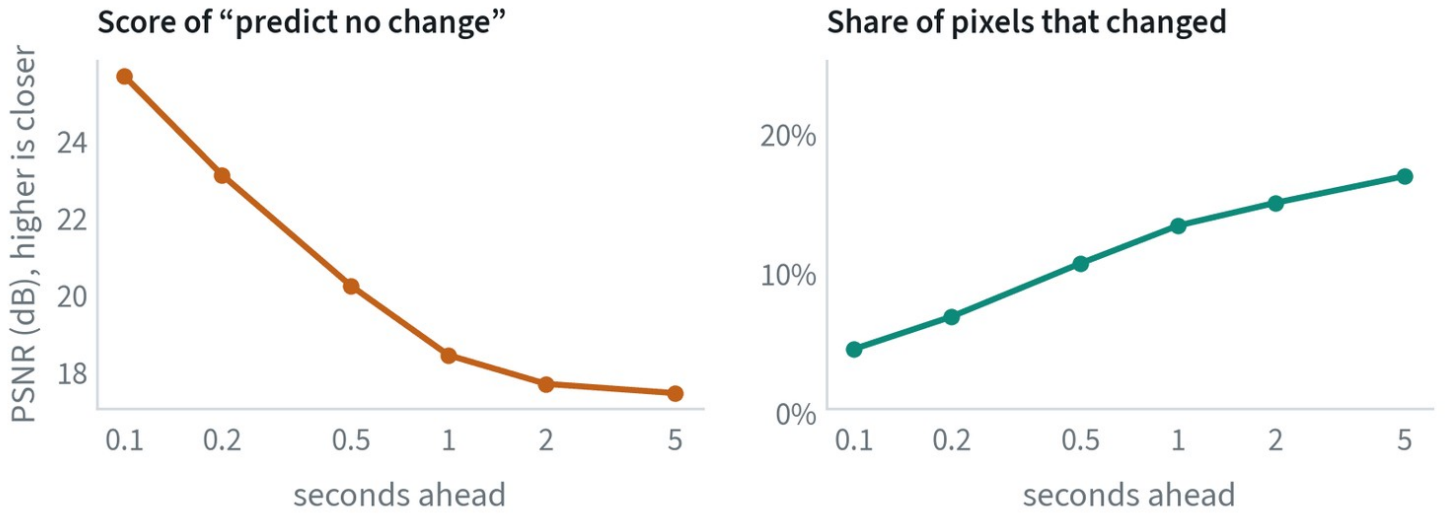
Predicting no change

Copying scores well. The unchanged current frame earns 25.6 dB at a tenth of a second, 20.1 at half a second, 18.3 at one second and 17.4 at five.

Little changes. The share of changed pixels is 4% at a tenth of a second, 13% at one second and 17% at five.

Most of the score is background. The baseline earns it without having predicted anything.

Alternative. Bypass pixels and ask whether the predicted video implies the right actions.



An image-similarity score means little without the persistence baseline beside it

Three uses, and one caution

what a learned world model is used for today, and the check that decides whether to trust it

Evaluate policies

- Roll a policy out inside the model, with no robot.
- Cheap, fast and safe, and useful for ranking checkpoints before real tests.

Generate data

- Produce new scenes, lighting and viewpoints to stretch scarce demonstrations.
- The generated video needs actions that match it.

Plan

- Imagine several action sequences, score the outcomes and execute the best.
- Slow today, and used mostly for high-level decisions.

The caution

- A learned model can produce physics that looks right and is wrong, and its errors grow with rollout length.
- An optimistic simulator will rank bad policies highly.

The check that matters: success inside the model predicts success on the robot, for the same policies

How world models are scored

four measures, from how each frame looks to whether the model ranks policies correctly

Measure	What it tells you	What it misses
Frame similarity (PSNR, SSIM, LPIPS)	How close each predicted frame is to the real one	Rewards blur; punishes a plausible future that differs from the recorded one
FVD	Whether the generated clips look like real ones as a set	Says nothing about following the action
Action following	Whether the video shows the commanded motion, for example by recovering the action from it	Needs a second model to judge
Policy ranking	Whether policies that do well in the model do well on the robot	Costs real robot trials to establish

Only policy ranking measures what a world model is for; the others are useful on the way there

Common mistakes with world models

three ways a convincing video leads to a wrong conclusion

Judging by how the video looks

- Plausible is not the same as correct.
- Check outcomes against reality.

Long rollouts

- Small errors compound, as they do for policies.

Ignoring the action

- A model that produces the same future whatever the action is a video model, not a world model.

Plausible is not the same as correct: check outcomes against reality

Embodiments

The bodies and sensors a policy actually controls.

KEY POINTS

- The body sets the action space, the data that can be collected and what can go wrong.
- Gearing trades strength for sensitivity: torque grows with the ratio and felt inertia with its square.
- Payload includes the hand, and repeatability is not accuracy.

Robots you will meet: three arms

the methods in this guide run on machines like these, from the standard lab arm to one built for work beside people



Franka research arm. Seven joints with torque sensing. The standard lab arm, and the arm simulated in LIBERO.



Flexiv Rizon. A seven-joint arm built around force control.



NEURA MAiRA. A collaborative arm with built-in sensing, for work beside people.

Sensing is part of the arm. The Franka senses torque in its joints, the Flexiv is built around force control, and the MAiRA has built-in sensing.

Where policies are strongest. Most public manipulation data comes from single arms with parallel grippers.

The bodies differ and the recipe is the same: cameras in, action chunks out

Robots you will meet: legs and wheels

a humanoid, a quadruped and a mobile manipulator



Unitree G1. A compact humanoid: whole-body control, two arms and hands.



Quadruped. Boston Dynamics Spot: 12 driven joints, three per leg.



Mobile manipulator. PAL Robotics TIAGo: arms and a head on a wheeled base.

More joints to coordinate. A quadruped has 12 driven joints. Humanoids typically have 20 to 50 or more, with balance and whole-body control to solve.

A moving base. It gives a large workspace, and the base and the arm must then be coordinated.

The body sets the action space, the data that can be collected and what can go wrong

Three machines by the numbers

figures from the makers' published specifications

Robot	Driven joints	Payload	Reach or size	Notes
Franka Research 3 (research arm)	7	3 kg	855 mm reach	A torque sensor in every joint
Universal Robots UR5e (collaborative arm)	6	5 kg	850 mm reach	Repeatability ±0.03 mm; force sensing at the tool flange
Boston Dynamics Spot (quadruped)	12	14 kg	32.5 kg; 1.6 m/s	Three joints per leg; an arm is an option

Payload is what the arm holds at full reach, and it includes the gripper

Reading a specification sheet

two numbers that are easy to misread, and why learned policies lean on them less

Payload

- What the arm can hold at full reach.
- It includes the gripper, so a 1 kg hand on a 3 kg arm leaves 2 kg.

Repeatability and accuracy

- Repeatability is how closely the arm returns to a position it has visited before.
- Accuracy is how close it gets to a commanded position it has never visited, and is typically several times worse.

What a learned policy needs

- Learned policies depend on neither as much as classical programs do.
- They close the loop through the camera.

Payload includes the hand, and repeatability is not accuracy

What the body decides

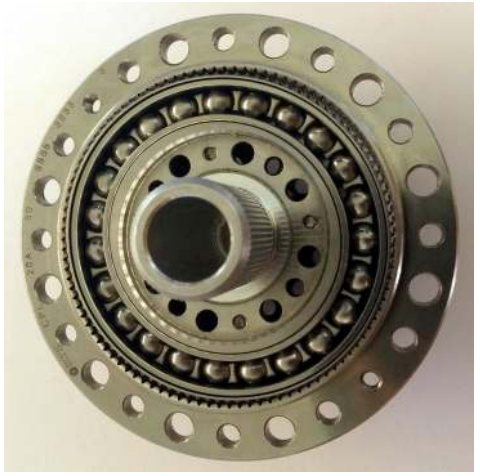
the same learning methods run on very different machines; degrees of freedom are typical ranges

Embodiment	Degrees of freedom	Strengths	Challenges for learning
Arms and collaborative robots	6 or 7, plus gripper	Precise, mature, safe to work beside	A fixed base limits reach and the tasks available
Parallel grippers	1	Simple, robust, easy to learn	Cannot re-orient an object in the hand
Dexterous hands	12 to 20 or more	Human-like manipulation	Hard to teleoperate, fragile, a large action space
Mobile manipulators	An arm on a moving base	Large workspace	Base and arm must be coordinated
Quadrupeds	12	Stable on rough terrain	Little manipulation unless an arm is added
Humanoids	20 to 50 or more	Fit human spaces and tools; can learn from human video	Balance, whole-body control, cost and data collection

Each body trades capability against how hard it is to control and to collect data for

The gear inside almost every robot arm

a motor is fast and weak, a joint needs to be slow and strong, and a strain-wave gear sits in between



A strain-wave gear, assembled. The fine teeth on the thin inner cup mesh with the outer ring.



Three parts: an elliptical bearing, a thin flexible cup, and a rigid ring with two more teeth than the cup.

How it works. The elliptical bearing deforms the cup so that it meshes with the ring at two points. One turn of the bearing advances the cup by two teeth.

Why arms use it. Ratios of 50 to 160 fit in a flat package with no backlash, so arms can be both precise and compact.

The gearbox is the least discussed and most consequential hardware choice in robot learning

Gearing trades strength for sensitivity

what a gearbox of ratio n does to torque, speed and the inertia felt at the joint

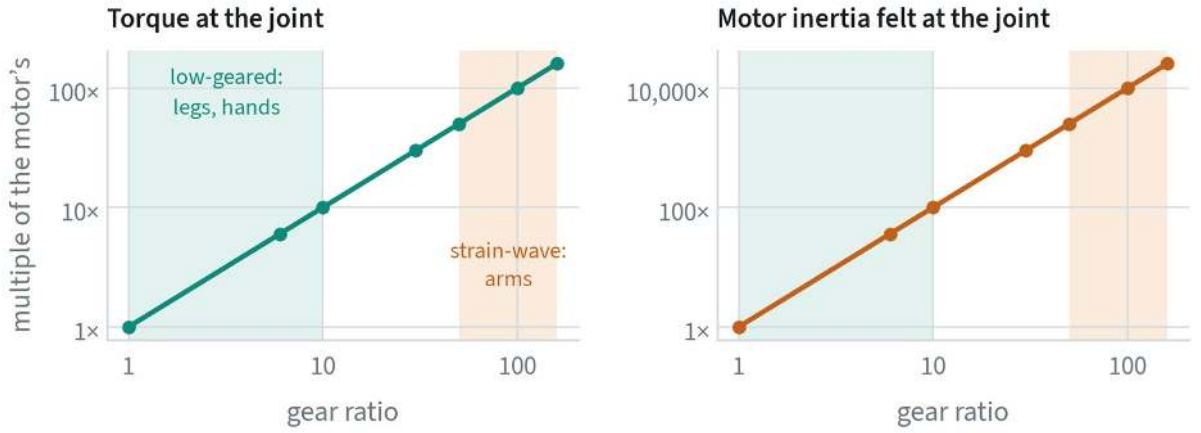
A gearbox of ratio n

$$\tau_{\text{joint}} = n\,\eta\,\tau_{\text{motor}} \qquad \omega_{\text{joint}} = \frac{\omega_{\text{motor}}}{n}$$

The motor's torque is multiplied by n and its speed is divided by n

$$I_{\text{reflected}} = n^2\, I_{\text{rotor}}$$

The motor's inertia, as felt at the joint, is multiplied by n squared



At a ratio of 100, torque is 100 times and felt inertia 10,000 times. Low-geared legs and hands sit below 10; strain-wave arms between 50 and 160.

At a ratio of 100 the joint behaves as though a heavy flywheel were attached

Actuators: what moves the joints

five designs and what each gives up

Design	Gear ratio	Strength	Weakness	Where
Strain-wave geared	50 to 160	Compact, precise, strong	High felt inertia; contact is invisible without a torque sensor	Arms; add a joint torque sensor to recover force control
Low-geared (quasi-direct drive)	6 to 10	Gives way on impact; motor current reveals contact forces	Large, heavy motors for a given torque; runs hot holding a load	Legged robots, some hands
Series elastic	High, with a spring in series	The spring measures force and absorbs shocks	The spring limits how fast force can change	Some humanoids and research arms
Hydraulic	None	Very high power for the weight	Pumps, hoses, leaks, noise	Large legged machines
Tendon-driven	Varies	Motors can sit away from the joint	Stretch, friction and wear	Hands

Demonstrations collected on a stiff, highly geared robot do not transfer cleanly to a compliant one

Power, heat and sensing on board

everything a mobile robot does runs from a battery, and a policy can only use what its sensors provide

Compute draws like the motors

- A GPU that runs a large policy draws tens to a few hundred watts continuously.
- That is comparable to the motors of a walking humanoid at rest.

What the budget explains

- Policies are compressed, the largest model is often run off the robot, and the split into slow and fast systems matters.

Heat

- Motors holding a load without moving, as a low-geared leg does when standing, turn current into heat.
- Motor temperature can limit a robot long before its battery is empty.

Sensing

- Scene cameras give context and wrist cameras a close view of the grasp.
- Body state is always available. Force, touch and sound matter once contact begins.

If a person cannot do the task from the robot's camera views alone, the policy cannot either

Common mistakes with embodiments

three assumptions about hardware that cost data

Moving a camera after collecting data

- The policy has learned the old viewpoint.

Expecting free transfer between bodies

- A different arm, gripper or camera layout usually needs new data.

Leaving out touch because data is scarce

- It is under-used because tactile data is hard to collect.
- That does not make it unimportant.

A policy is tied to the body and the camera layout it was trained on

Senses beyond vision

Touch, force, sound and body state: what each tells a robot, and how they are combined in one policy.

KEY POINTS

Cameras lead until contact. After that, force, touch and sound carry the information.

Physics sets the deadline: a slip must be answered in tens of milliseconds, faster than a chunked policy replans.

A sense that was added is not a sense that is used. Test with it removed.

Run it: Touch, force and sound (github.com/wnukmat/robot-learning-notebooks)

Vision is not enough

a schematic of which sense carries the information in each phase of a pick-and-place; the ratings are a judgement

Approach. Cameras are decisive while the hand reaches toward the object, and for that they are sufficient.

From contact onward. Once the hand closes, the camera's view is blocked. Force, torque and touch become decisive.

New questions. Whether the object is in the grip, whether it is slipping, and whether it has touched down.

Placement. Sound is decisive at the moment the object is put down.

	approach	contact	grasp and lift	carry	place
Scene camera	decisive	some	some	useful	useful
Wrist camera	useful	decisive	some	some	decisive
Body state	useful	useful	useful	decisive	useful
Force and torque		decisive	decisive	useful	decisive
Touch		decisive	decisive	decisive	useful
Sound		useful	some	some	decisive

Cameras lead until contact; after that, force, touch and sound carry the information

What each sense tells the robot

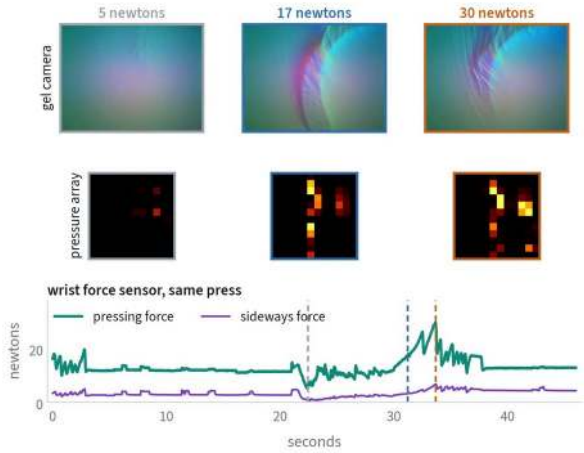
what is measured, what it is good for, and the usual way it goes wrong

Sense	What it measures	Good for	Watch out for
Body state (proprioception)	Joint angles and speeds, gripper opening	Knowing where the robot is; always available and precise	Policies lean on it and ignore the cameras
Joint torque and motor current	How hard each joint is working	Detecting contact and load with no extra sensor	Noisy; mixes contact with the arm's own weight and friction
Wrist force and torque	The total push and twist at the wrist	Insertion, wiping, pressing, knowing when to stop	One total, not where contact is; drifts and needs zeroing
Touch: camera-based, or pressure arrays	A fine image of the fingertip's skin deforming, or pressure over a grid of cells	Shape, texture and slip; with arrays, contact location over fingers and palms	Camera type is bulky, skin wears, units differ; arrays are coarse, weak on sideways force
Sound	Vibration through air, or through the gripper with a contact microphone	Contact events, slip, pouring, clicking into place	Motor and room noise; differs between sites
Inertial sensor	Acceleration and rate of turning	Balance and base motion on legged and mobile robots	Drifts when integrated over time
Depth and lidar	Distance to surfaces	Geometry, obstacles, navigation	Shiny, dark and transparent objects

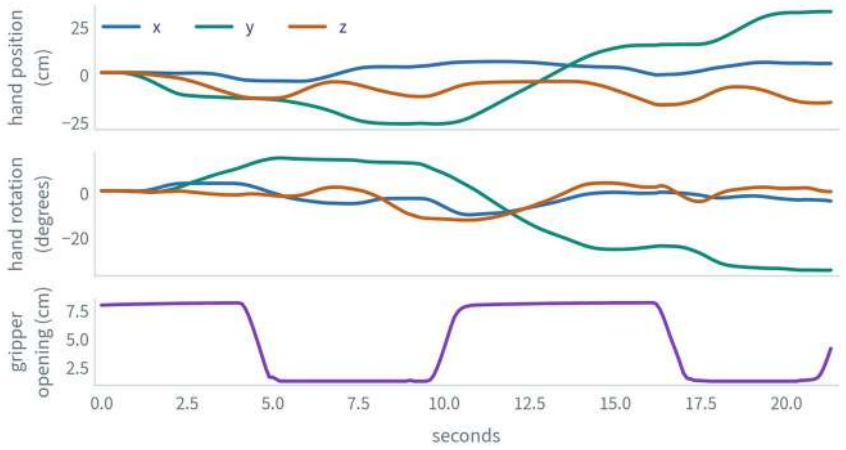
Every sense answers a different question, and every one has a characteristic failure

What the signals look like

real recordings: one press seen by three sensors, and body state through one robot episode



One press at 5, 17 and 30 N: gel fingertip camera, 12 × 12 pressure array, and wrist force over 46 s.



A 21 s episode: hand position, rotation and gripper opening. This robot reports 8 numbers per step.

Three answers about one contact. The gel camera shows the shape in fine detail, the array shows where, coarsely, and the force sensor gives one number for how hard.

Body state is the plainest signal of all: a few smooth numbers per step. That is why it is so easy for a policy to lean on.

It is easier to reason about a sense after seeing its raw data

Depth cameras

a stereo depth camera finds the same point in two images taken a known distance apart

Depth from disparity

$$Z = \frac{f\,B}{d}$$

Depth from the disparity d, with focal length f in pixels and baseline B

$$\delta Z \approx \frac{Z^2}{f\,B}\,\delta d$$

Depth error grows with the square of the distance



A stereo depth camera of the kind found on many research robots. The outer lenses are infrared cameras; the projector between them adds texture to blank surfaces.

A sensor accurate to a millimetre at 30 cm is accurate to about a centimetre at one metre

A closer look at sound

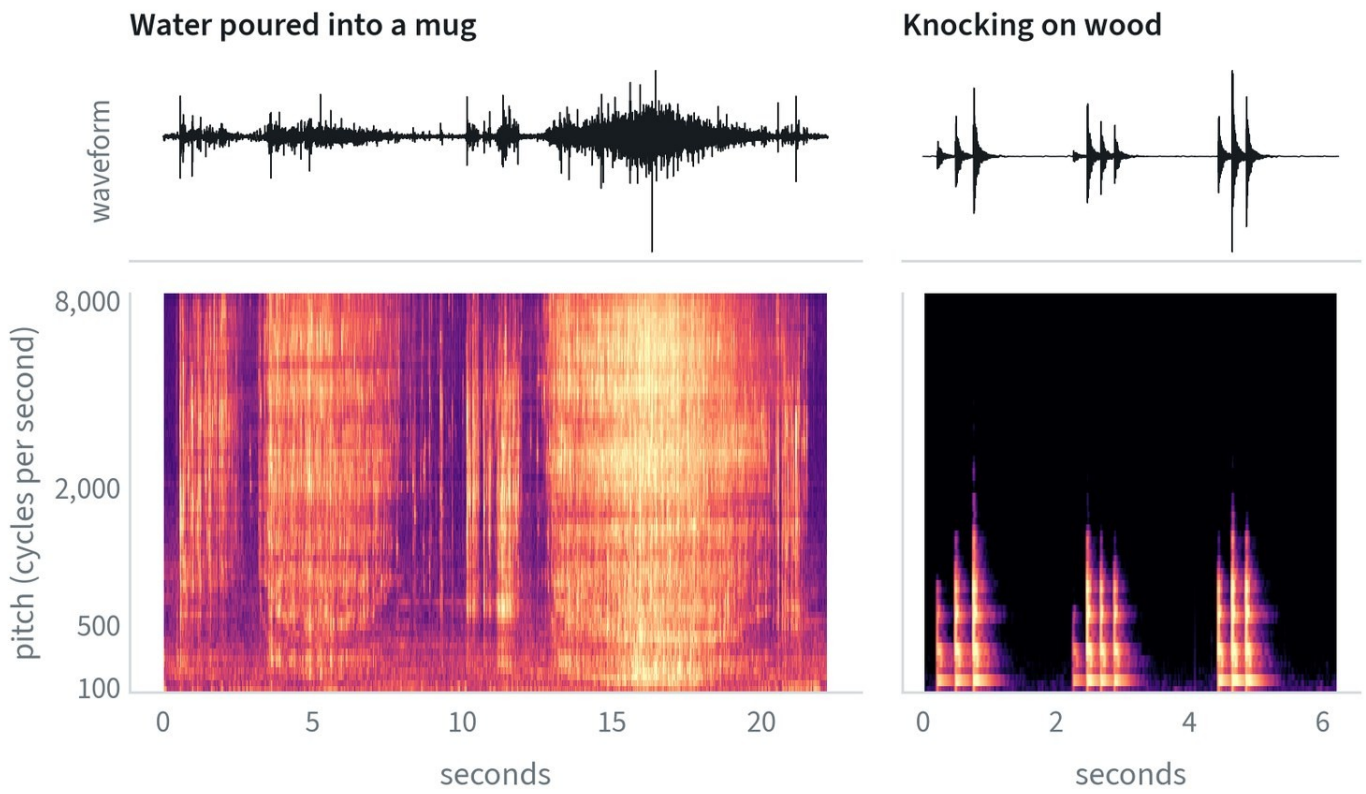
two public-domain recordings, each as a waveform above its spectrogram of pitch against time

Flow and contact look nothing alike. Pouring into a mug is a broad wash across all pitches that swells as it fills. A knock is a sharp burst.

Recognisable patterns. Each knock is a spike in the waveform. In the spectrogram it is a column of bands that rings and fades in a fraction of a second.

A contact microphone on the gripper hears the scrape of a spatula under food, the click of a plug seating, and the pitch rising as a container fills.

Conditions. Sound helps most in contact-rich tasks where vision is ambiguous, and needs the same microphone placement in collection and use.



Sound reports events a camera cannot see, at the moment they happen

From a waveform to a spectrogram to tokens

the short-time Fourier transform, the mel scale, and patches as for any image

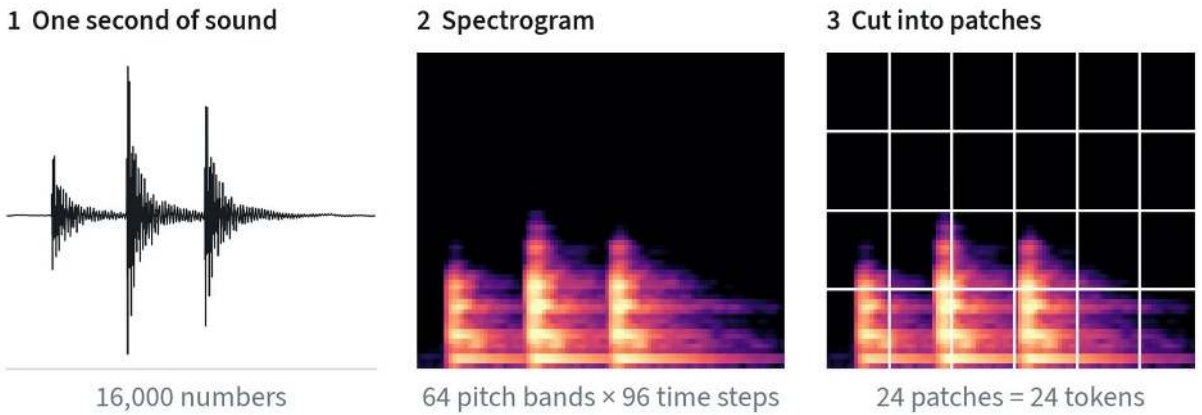
Computing a spectrogram

$$X[m,k] = \sum_{n=0}^{N-1} x[n+mH]\, w[n]\, e^{-2\pi i\, kn/N}$$

Overlapping windows of length N, tapered by w, hopped by H samples, then transformed

$$m(f) = 2595\, \log_{10}\left(1 + \frac{f}{700}\right)$$

Mel scale: roughly linear below 1,000 cycles a second and logarithmic above



One second of a knock is 16,000 numbers. As a spectrogram it is 64 pitch bands by 96 time steps. Cut into patches it is 24 tokens, handled like any image.

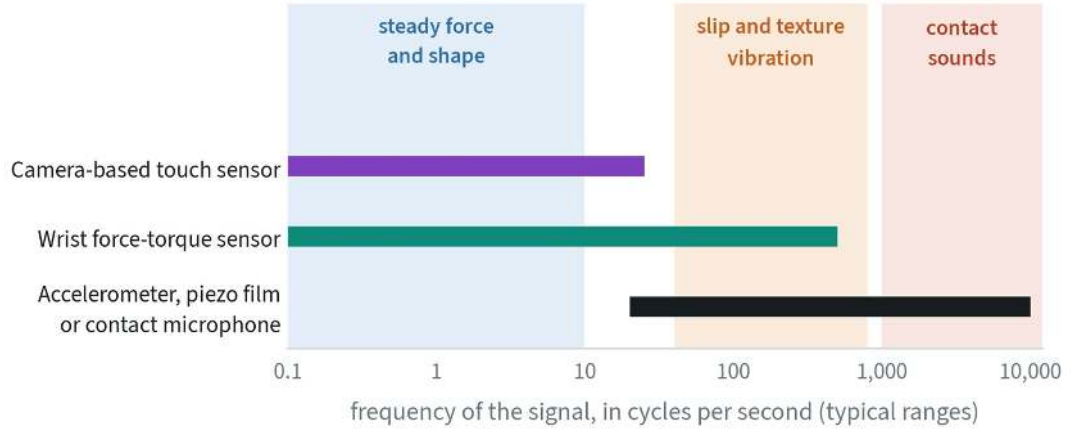
A long window resolves pitch finely and blurs timing; a short one does the reverse

Different senses, different clocks

typical reading rates of each sensor, and which sensor can carry which band of the touch signal



Cameras: tens a second. Joint, force and inertial: hundreds to thousands. Microphones: tens of thousands.



Force and shape: below about 10 cycles a second. Slip, texture: tens to hundreds. Contact sounds: thousands.

Windows, then tokens. A policy runs 10 to 50 times a second, so each fast signal is cut into a short window, such as the last tenth of a second.

Touch has two bands. Camera-based fingertips at 30 to 60 frames a second carry force and shape. Vibration needs hundreds or thousands of readings a second.

Keep a short window of every fast signal: the latest reading alone loses the spike and the vibration

How a model takes each sense in

cut the signal into pieces, turn each piece into a vector with an encoder built for it, and join the sequence

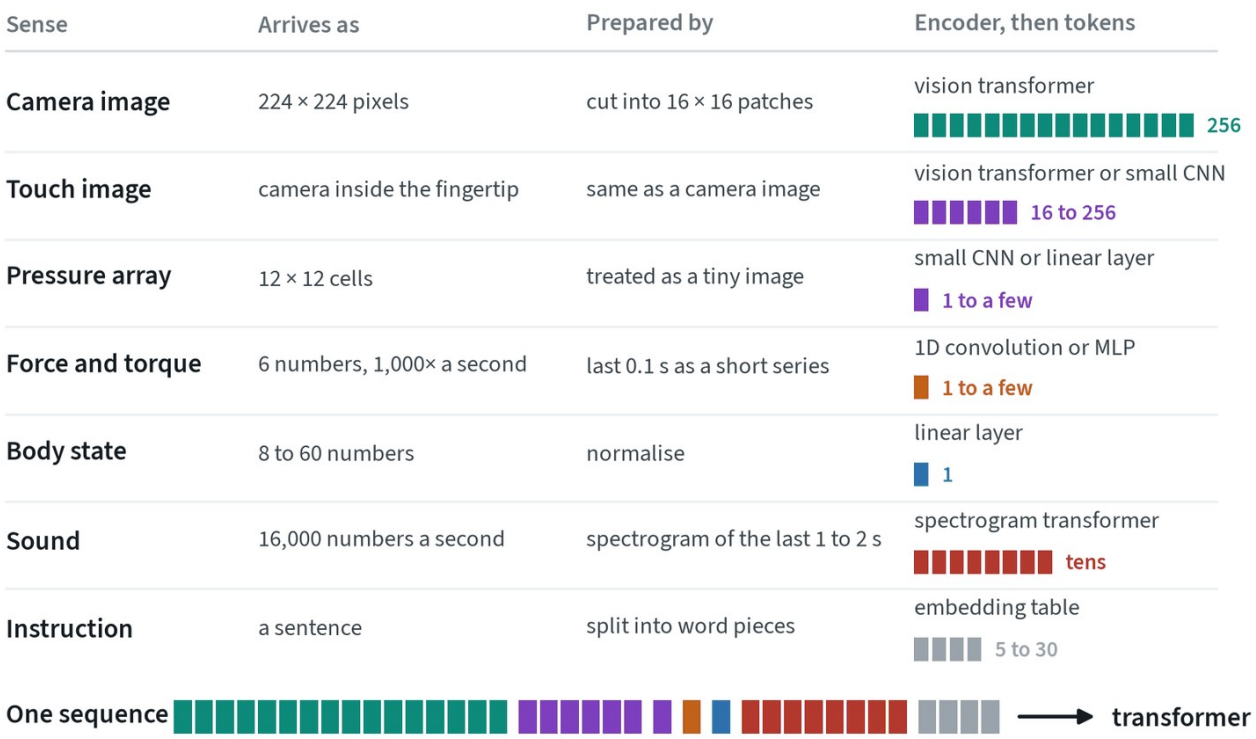
Everything becomes tokens

Images and touch images. Square patches, one token each: typically 256 for a camera image. A gel fingertip uses a smaller encoder trained on touch images.

Force and torque. The last tenth of a second or so of six-number readings goes through a small network that outputs one or a few tokens.

Body state and pressure. Body state is normalised and passed through one linear layer to give 1 token. A pressure array is a tiny image or one flat vector.

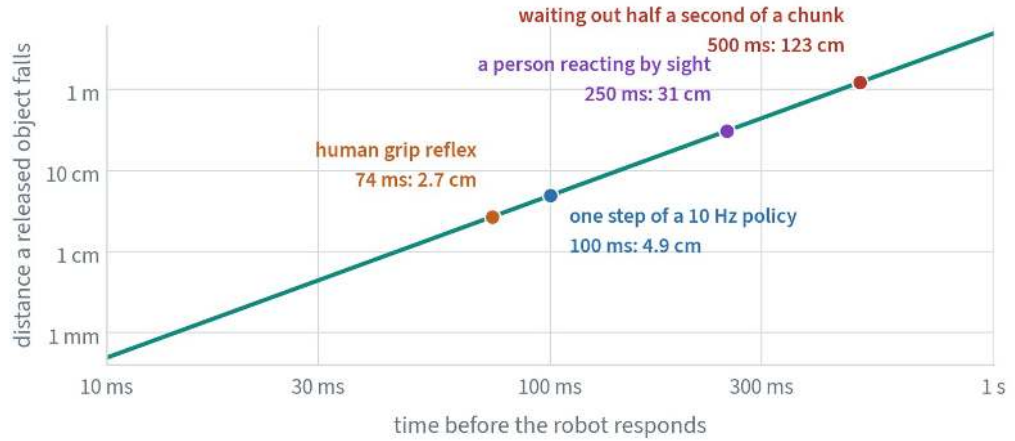
Instruction. Word pieces looked up in a table of vectors. "put the white mug on the plate" becomes seven tokens.



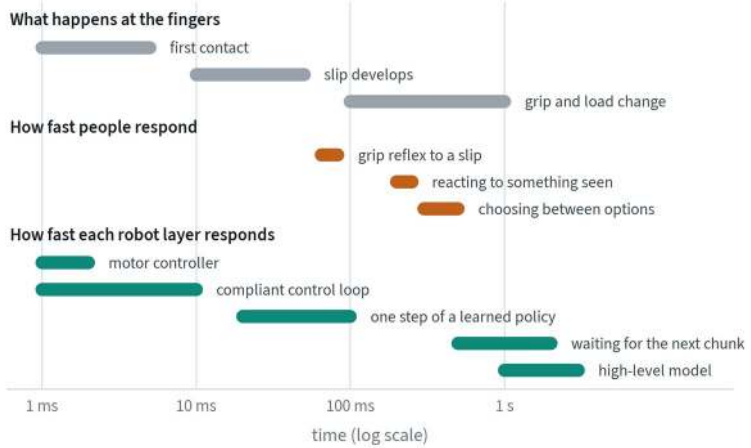
More tokens keep more detail and make every pass through the model slower

Deadlines: how fast is fast enough

how quickly a robot must act on a sense is set by physics; for a slipping object, gravity is the clock



Free fall, distance = ½ g t^2: 2.7 cm at 74 ms, 4.9 cm at 100 ms, 31 cm at 250 ms, 123 cm at 500 ms.



Events, human responses and robot layers on one axis, as typical ranges. A layer answers only slower events.

The grip reflex. People tighten their grip about 74 ms after a slip starts, through touch alone. Reacting to something seen takes three times as long.

Where to handle it. Under about 10 ms belongs in the controller. A slip, at tens of ms, needs a reflex or a policy that reacts between chunks.

A robot that notices a slip only when its next action chunk begins has already dropped the object

Combining senses in one policy

each sense gets a small encoder and its tokens join the image and language tokens; the difficulty is in the training

One sense dominates

- Vision predicts the action well for most of every episode, so the others get almost no weight.
- Remedy: randomly drop or blank a sense during training.

Check that it is used

- Run the policy with one sense removed or replaced by noise.
- If success does not change, that sense is not contributing.

Sensors differ

- Two touch sensors of the same model read differently, and a microphone moved by a centimetre hears something else.
- Calibrate, or collect across several units.

Little shared data

- Almost no large public dataset includes touch or sound.
- These encoders start from scratch while the vision encoder starts from the whole web.

The practical order

- Slipping calls for touch.
- Missed insertions call for force.
- Events hidden from the camera call for sound.

Add a sense when a specific failure calls for it, then test the policy with that sense removed

Common mistakes with extra senses

three errors in adding and feeding sensors beyond the cameras

Adding sensors before finding the failure

- Each one adds calibration, wear and a new way for training and deployment to differ.

Feeding only the latest reading

- Fast senses carry their information in how the signal changes over a short window.

Assuming a sense is used

- Test with it removed.

A sense that was added is not a sense that is used: test with it removed

Dexterous hands

Multi-fingered hands in depth: real hardware and its specifications, how a hand connects to the action head, retargeting, and using touch.

KEY POINTS

- A hand changes the hardware, the data collection, the sensing and the action space at once.
- How the hand’s action is written decides how much data is needed and what transfers.
- Touch is used twice: as input to the policy, and as a fast reflex underneath it.

A hand changes every part of the pipeline

a parallel gripper is two fingers and one number; these three hands show the main design decisions



Shadow Dexterous Hand. Tendon-driven: the cables running into the forearm are the tendons



Schunk SVH. Nine drives move twenty joints, so several joints move together



Allegro Hand fitted with uSkin touch sensors on each finger segment

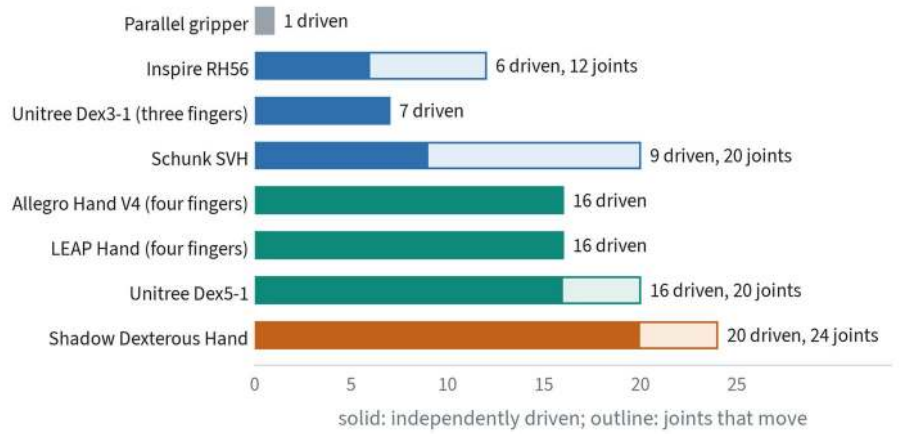
Joints and motors. A motor per joint places each segment. Fewer motors make fingers wrap. Forearm tendons give slim fingers and cables that stretch and wear.

Gearing and thumb. Low gearing lets the motor feel contact and give way. High gearing is strong, stiff and blind. Check how many thumb joints are driven.

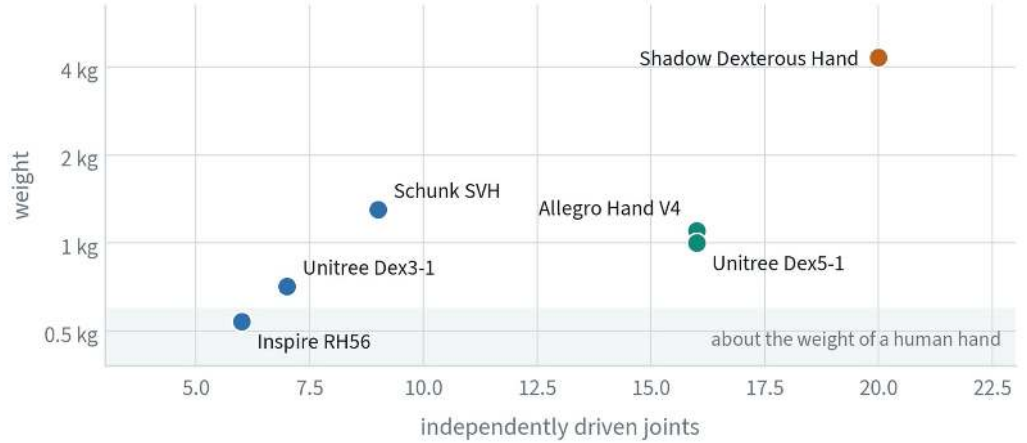
Driven joints, motor placement, gearing and the thumb account for most differences between hands

Real options: more driven joints cost weight

figures from makers' pages, the LEAP Hand paper and published listings; confirm against the current data sheet



Driven joints (solid) and moving joints (outline): 1 for a gripper, 20 of 24 for the Shadow hand



Five hands weigh 0.5 to 1.3 kg. The Shadow hand, with 20 driven joints, weighs 4.3 kg with its forearm

Start from the task. Picking, placing and holding tools need a secure grasp, and six motors is enough. Turning and re-gripping need independent joints.

Check the weight against the arm. A 1 kg hand on an arm rated for 3 kg leaves 2 kg for the object. Only six-motor hands approach a human hand's weight.

Choose from the task, then check weight, repair time, the control interface and a simulation model

Eight hands you can buy or build

driven joints against total joints, how they are driven, weight and touch sensing; specifications vary by source and version

Hand	Driven / joints	Drive	Weight	Touch sensing
Inspire RH56 (5 fingers)	6 / 12	Linear actuators and linkages	about 0.54 kg	On some versions
PSYONIC Ability Hand (5)	6	Motors in the fingers	about 0.49 kg	Fingertip sensors
Unitree Dex3-1 (3)	7 / 7	Motors in the hand	about 0.71 kg	33 sensors on the tactile version
Schunk SVH (5)	9 / 20	Motors in the hand, coupled joints	1.3 kg	None as standard
Allegro Hand V4 (4)	16 / 16	A motor at every joint	about 1.1 kg	Add-on skins available
LEAP Hand (4)	16 / 16	Off-the-shelf servos at every joint	not confirmed	None as standard
Unitree Dex5-1 (5)	16 / 20	Motors in the hand	about 1.0 kg	94 force sensors on the tactile version
Shadow Dexterous Hand (5)	20 / 24	Tendons from motors in the forearm	4.3 kg with forearm	Tactile fingertips on thumb and index

The choice is between few driven joints that are light and robust, and many that are heavy and capable

Two pieces of hand mechanics

tendon coupling explains why fingers wrap; the friction cone explains when a grasp holds

Coupling and friction

$$\Delta l = P\,\Delta q \qquad\qquad \tau = P^{\top} f$$

One coupling matrix P maps joint motion to tendon motion and tensions to joint torques

$$|f_t| \le \mu\, f_n$$

A fingertip holds only while tangential force is within friction times normal force

In words

Square, invertible P: the hand is fully actuated and every joint can be placed independently.

Fewer rows than columns: several joints share a motor, and their motion is determined only once the finger meets something.

Friction cone. The allowed forces form a cone around the surface normal with half-angle arctan μ.

Force closure. The fingers, each pushing within its cone, can resist a force or twist in any direction.

Slippery objects have narrow cones, so the fingers must oppose each other almost exactly. A grippy one leaves room for error.

A learned grasp must satisfy this geometry whether or not the policy represents it

Four ways to write the hand's action

one step for a seven-joint arm with a four-finger, 16-joint hand, or with a six-motor hand in the third row

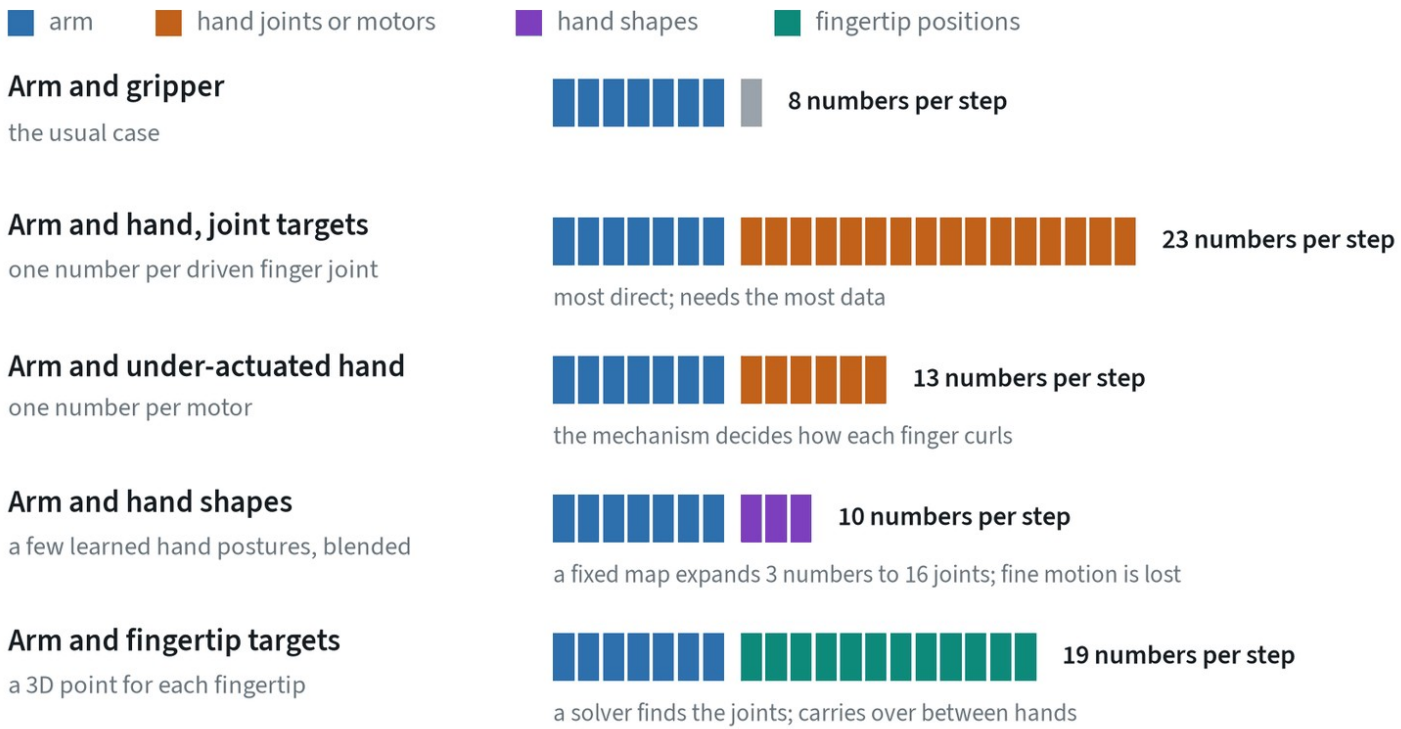
What the policy outputs

Joint targets, 23 numbers. An angle for every driven joint. Nothing is lost. It is the largest action and is tied to one hand.

Motor commands, 13. One number per motor of an under-actuated hand. The mechanism handles the wrapping, with no control of individual segments.

Hand shapes, 10. A few numbers blend learned whole-hand postures. Easy to learn from little data. Fine finger motion cannot be expressed.

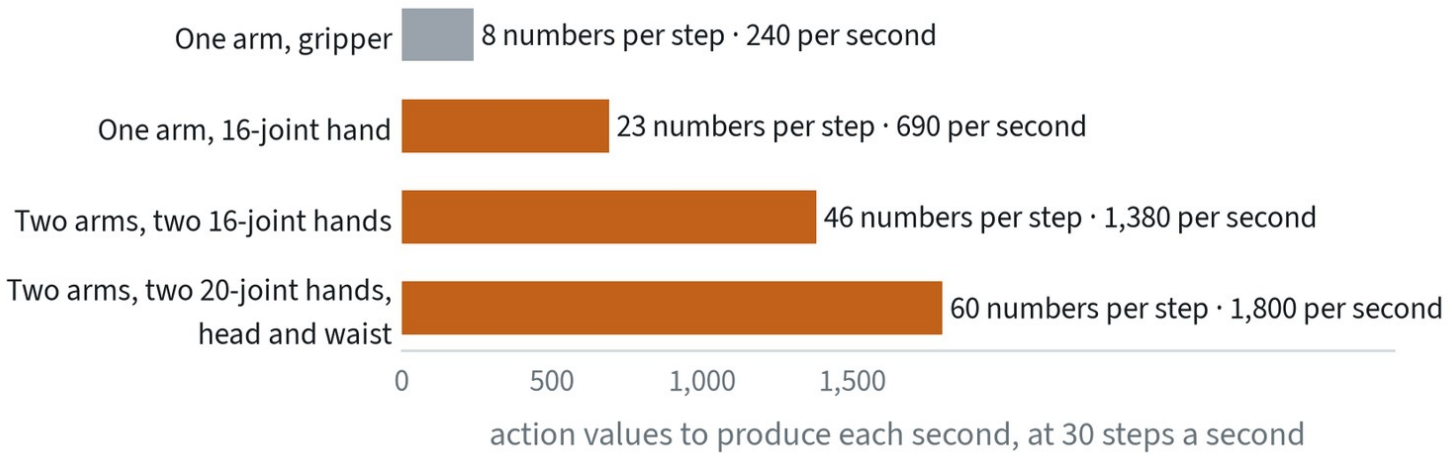
Fingertip targets, 19. A 3D point per fingertip, relative to the wrist, means the same on different hands. It needs a solver outside the policy.



Smaller actions learn faster and can do less, and no one representation has won

The action grows, and the action head has to keep up

action values to produce per second at 30 steps a second, from a gripper arm to a two-handed humanoid



240 to 1,800 per second. A gripper arm outputs 8 numbers a step. Two arms, two 20-joint hands, head and waist output 60.

The kind of head matters more. A token head pays for every number. A flow-matching or diffusion head produces the whole vector in each refinement step.

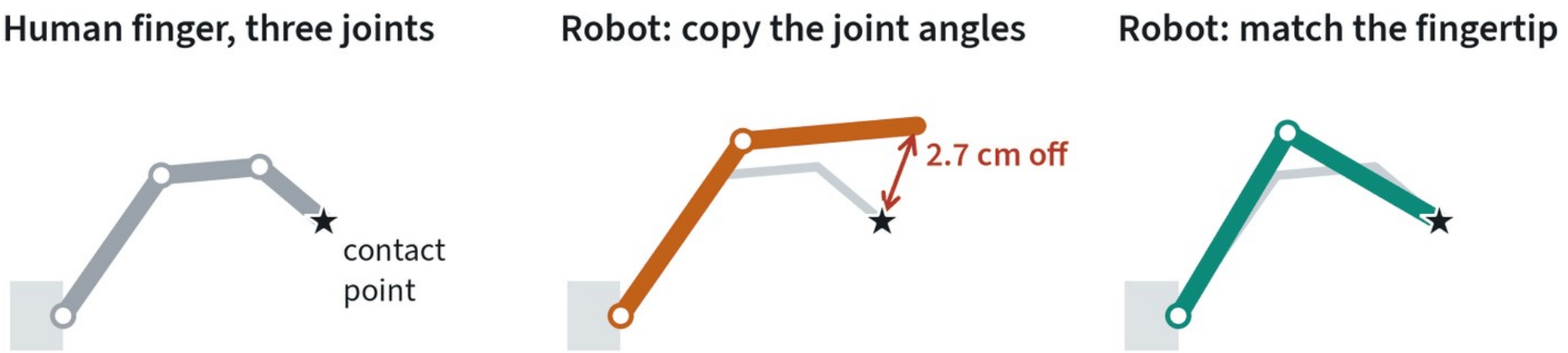
One action, normalised per joint. Arm and hand predicted in one chunk stay coordinated. Finger joints have very different ranges.

Gripper pretraining helps only partly. It teaches where to reach and what the task is. The finger motion has to come from hand data.

A two-handed humanoid produces seven or eight times as many numbers per second as a gripper arm

Retargeting: a human hand is not a robot hand

computed for a three-joint human finger and a two-joint robot finger of different lengths



Copying joint angles keeps the hand's shape and loses the contact. Here the robot fingertip ends 2.7 cm from the contact point.

Matching the fingertip position reaches the point with different joint angles. The contact is kept and the shape changes.

In practice most systems match fingertips, with extra terms to keep the thumb opposed to the fingers and to avoid the fingers colliding.

No mapping is exact. The leftover error is one reason hand policies need more data than gripper policies.

Matching fingertips keeps the contact, which is what manipulation depends on

Collecting data for hands

four sources of hand data, each with a different gap between the person and the robot

Method	How it works	Strength	Limit
Teleoperation	A glove or headset tracks the person's hand and the robot hand mirrors it	Real robot actions and observations	Slow; the operator feels nothing; needs the robot
Portable motion capture (DexCap)	Tracking gloves and a camera; the person works with their own hands	Fast, natural, collected anywhere	Human-hand data: must be retargeted, and the camera sees a hand
Wearable exoskeleton (DexUMI)	A frame limits the hand to motions the robot hand can make; video shows the robot hand	Closes the motion and visual gaps; the person feels the contact	One exoskeleton design for each robot hand
Reinforcement learning in simulation	The policy learns by trial and error on a simulated hand	No human data; has produced skilled in-hand rotation	Contact is hard to simulate; transfer needs care

The trend is toward capturing people directly: they work much faster and feel what they are doing

Touch is used twice

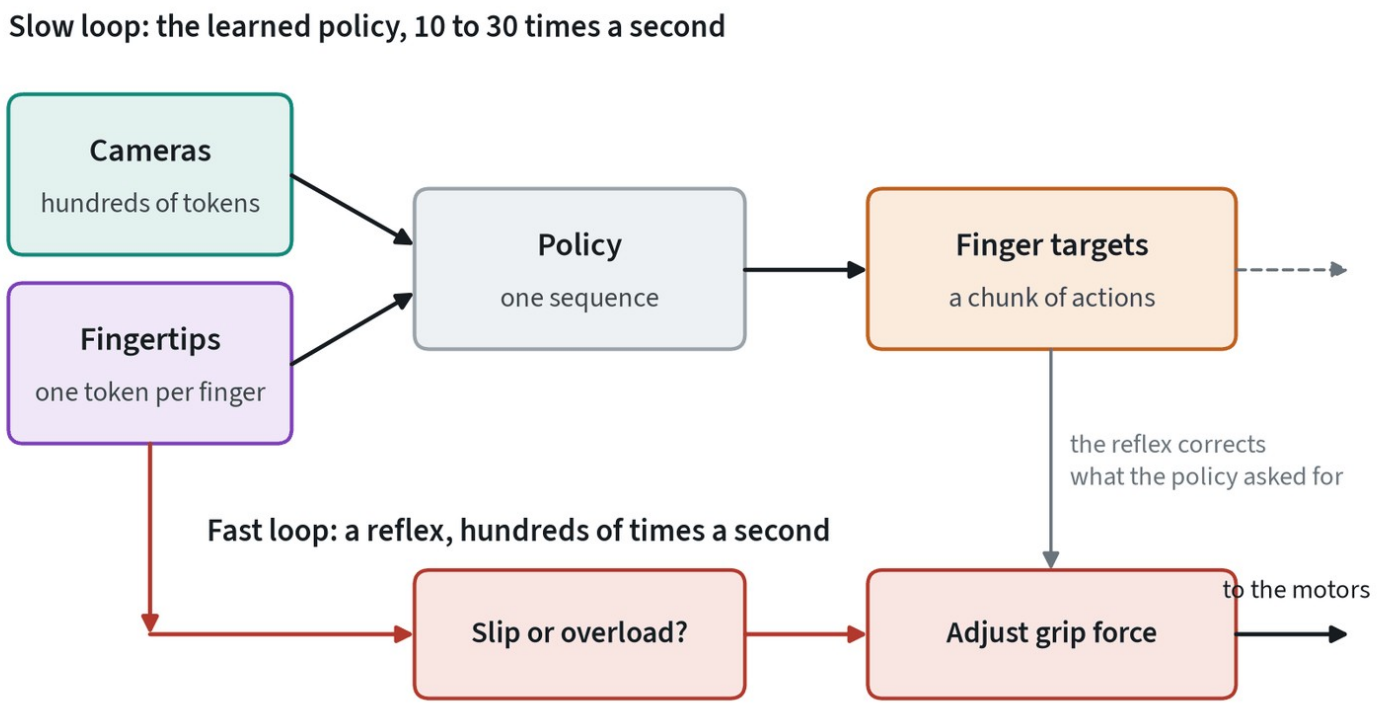
fingers cover the object they hold, so the camera loses sight of it exactly when it matters

As policy input. Each sensor patch becomes one or a few tokens, usually one group per finger. The policy learns seating, tilt and touchdown.

As a reflex. A slip must be answered within tens of milliseconds. A small rule tightens on vibration or sideways force and loosens above a force limit.

In collection and checking. Touch fed back to the operator gives gentler, more consistent demonstrations. Expected contact is a cheap success check.

Two cautions. Drop the camera input at random in training, then test with touch removed. Calibrate each sensor and expect to replace skins.



The policy sets the intent 10 to 30 times a second; the reflex keeps the grip hundreds of times a second

Open questions about hands

five things the field has not settled

Whether five fingers are needed

- Many useful tasks need two or three fingers.
- A human-like hand fits human tools and human video. How much benefit comes from which is unclear.

How the action should be written

- Joint angles, a handful of hand shapes, or fingertip positions.
- Smaller choices learn faster and can do less.

Whether human video can carry the load

- It is the only source large enough.
- How much robot data is needed on top is unsettled.

How much touch adds

- Results are strong on specific contact-rich tasks and mixed in general.
- There is no large shared touch dataset.

How success is measured

- There are few shared benchmarks.
- Results on different hands are hard to compare.

Results on different hands are hard to compare, because there are few shared benchmarks

Common mistakes with dexterous hands

four errors that cost data collection time

Choosing the most capable hand

- A simpler hand that survives a day of data collection teaches more.

Copying joint angles

- Check where the fingertips end up, not whether the pose looks right.

Treating the hand as a separate policy

- Arm and fingers move together and should be predicted together.

Adding touch and assuming it is used

- Test the policy with the touch input removed.
- If nothing changes, it was ignored.

A simpler hand that survives a day of data collection teaches more

Data for robots

Where robot training data comes from, and why quality matters more than volume.

KEY POINTS

Robot data comes in layers that trade volume against closeness to the target robot.

Push every skill as far down the pyramid as it will go.

Simulation and digital twins are for developing methods and screening policies, not for replacing real contact.

Run it: Inside a robot dataset (github.com/wnukmat/robot-learning-notebooks)

The data pyramid

language models had the web; robotics has no equivalent record of robot actions

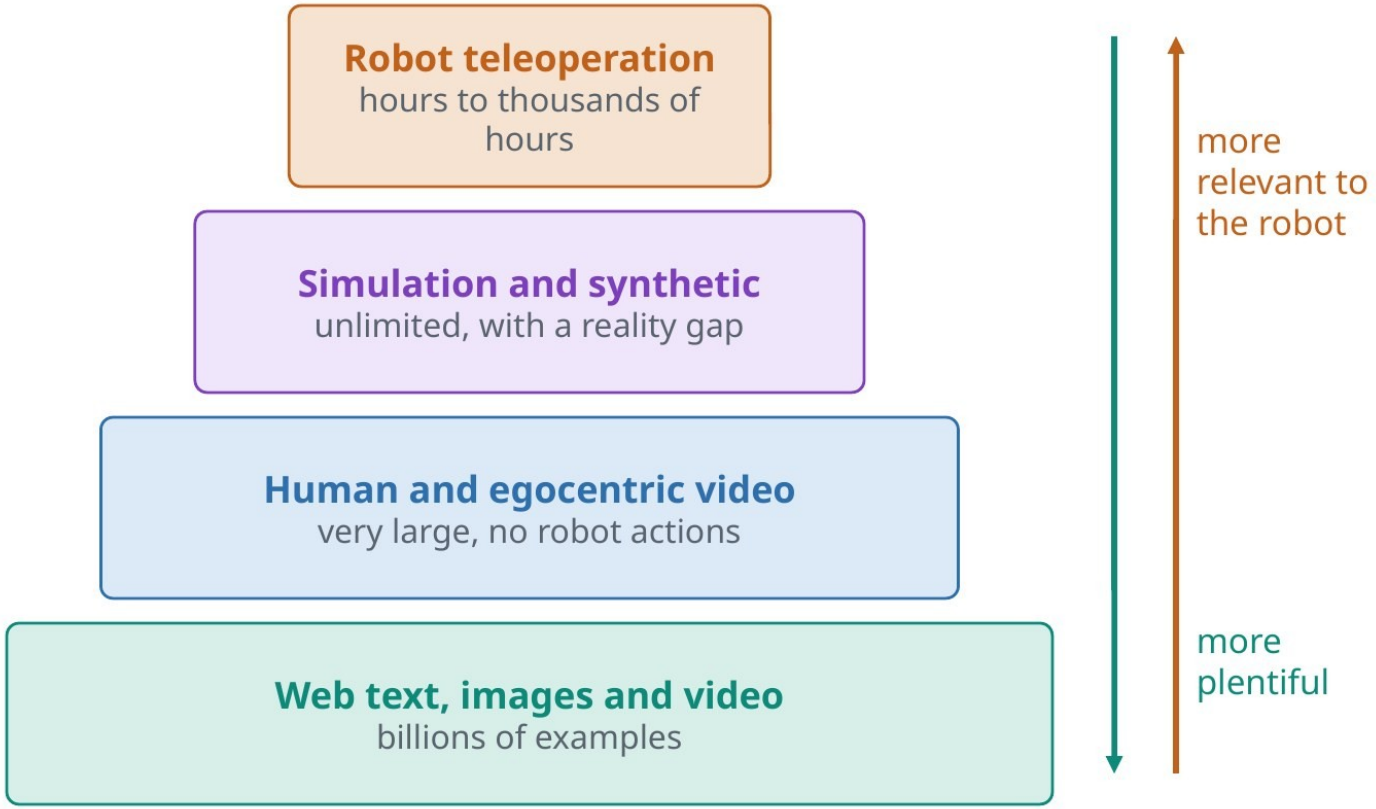
Four layers

Web data teaches language, objects and common sense, with no actions.

Human video shows how tasks are done by hands, but the actions have to be inferred.

Simulation and synthetic data have perfect labels and unlimited scale, but they differ from reality.

Teleoperation is exactly the right data, the robot's own sensors and actions, and it is slow and costly to collect.



Use the cheap layers to learn about the world and the expensive one to learn how to act

The pyramid for a humanoid

chosen by one question: how close is this data to the robot we ship, and what does an hour of it cost

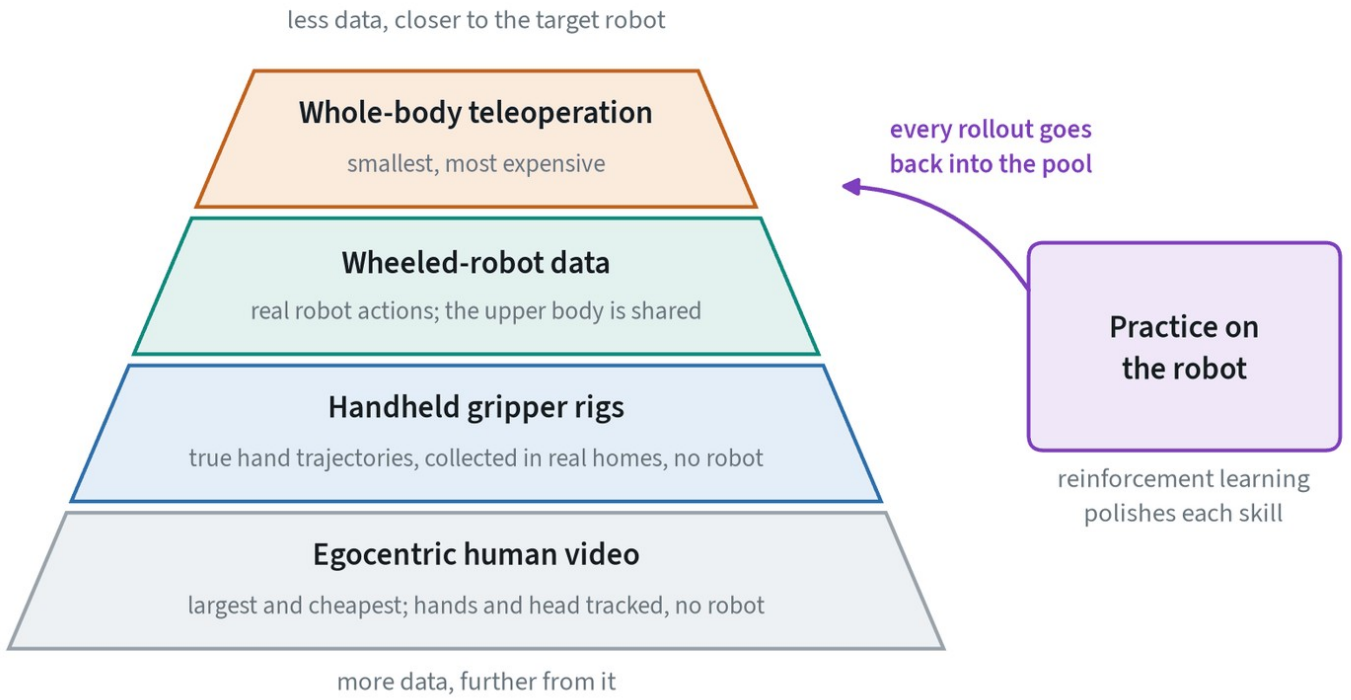
Five sources

Egocentric human video. Volume and variety. It lacks robot actions, forces and the robot's own view.

Handheld gripper rigs. True hand trajectories with the robot's gripper and camera, collected in real homes. They lack the arm and body.

Wheeled-robot data. A wheeled base with the same arms shares the upper body, so manipulation carries over. No walking or balance.

Whole-body teleoperation. The full robot in the task. Slowest and most expensive, so kept for skills only a biped has, and for recovery.



Push every skill as far down the pyramid as it will go

Collecting and storing demonstrations

four ways to record a person's motion, and the two formats the result is usually stored in

Leader-follower arms

- The operator moves a small copy of the arm and the robot mirrors it.
- This made two-arm data collection cheap.

VR headsets and controllers

- These track the operator's hands.

Handheld grippers

- People collect manipulation data with no robot present at all.

Exoskeletons and gloves

- Used for humanoids and dexterous hands.

LeRobot format or RLDS

- Each records observations, actions, timestamps and the task description.
- Converting between them is a routine chore.

A large share of the bugs on a robot learning project live in the data plumbing

What a real dataset contains

the LIBERO-10 dataset used for figures throughout this guide: 379 demonstrations of ten tasks

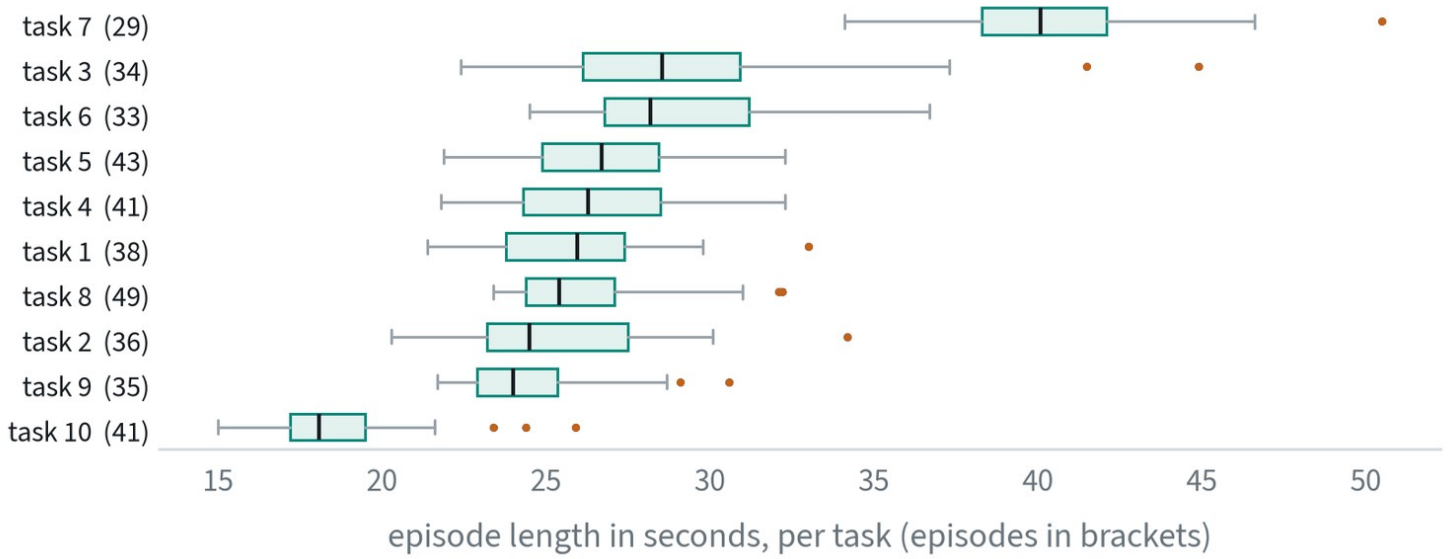
One dataset, concretely

Size. 379 demonstrations, 29 to 49 per task, and 101,469 frames at 10 per second: 2.8 hours of robot time.

Length. Episodes last 15 to 50 seconds with a median of 26. Task medians run from about 18 to 40 seconds.

Fields. Scene and wrist cameras at 256 × 256 × 3, a state of 8 numbers, an action of 7, the instruction, and timestamps.

The scale problem. A language model trains on text a person would need tens of thousands of years to read. This is an afternoon's work.



Three hours is small, and typical of what one lab collects for one study

Three public collections, and a storage trap

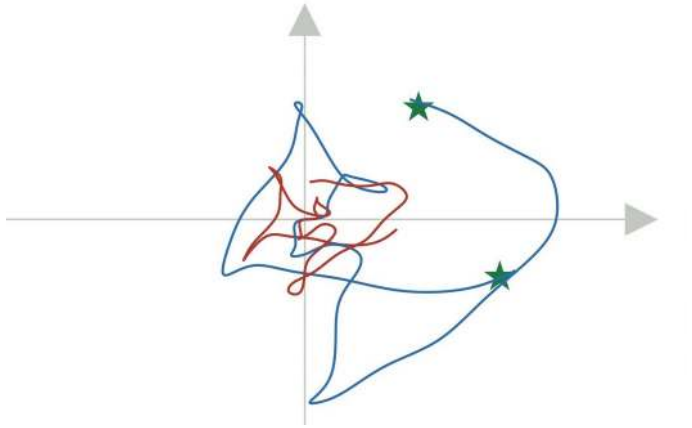
scale as reported by each dataset's authors

Dataset	Scale	What it is for
LIBERO	130 tasks in simulation, in four suites	Comparing methods under identical conditions
Open X-Embodiment	Data from 22 robots, pooled across 21 institutions, covering 527 skills	Pretraining generalist policies across robots
DROID	About 76,000 trajectories, 350 hours, in 564 scenes	Variety of real scenes with one robot setup

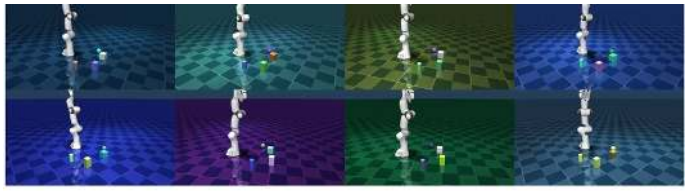
Timestamps, not frame counts, are the reference when matching images to actions

Simulation and the reality gap

a measured toy example: a controller planned in a simulator with the wrong friction



Blue is what the simulator predicted and red is what the real system did



Domain randomisation: the same scene under randomised appearance



Mean miss on the real system: plain 0.82, narrow randomisation 0.79, wide 0.39. Lower is better

The gap. Rendered images differ from camera images, simulated contact and friction are approximations, and real sensors have delay and noise.

Domain randomisation. Vary lighting, textures, masses and friction in training so that reality looks like one more variation.

Randomising helps only when the real value falls inside the randomised range

Turning human video into robot data

people have recorded far more manipulation than robots have; three routes make it usable

Track the hand and retarget

- Estimate the hand's pose in each frame and convert it to a gripper pose.
- Works best with head-mounted or wrist-mounted cameras that see what a robot would see.

Infer actions with a model

- Label the video with an inverse dynamics model or with latent actions.
- The world models note shows both.

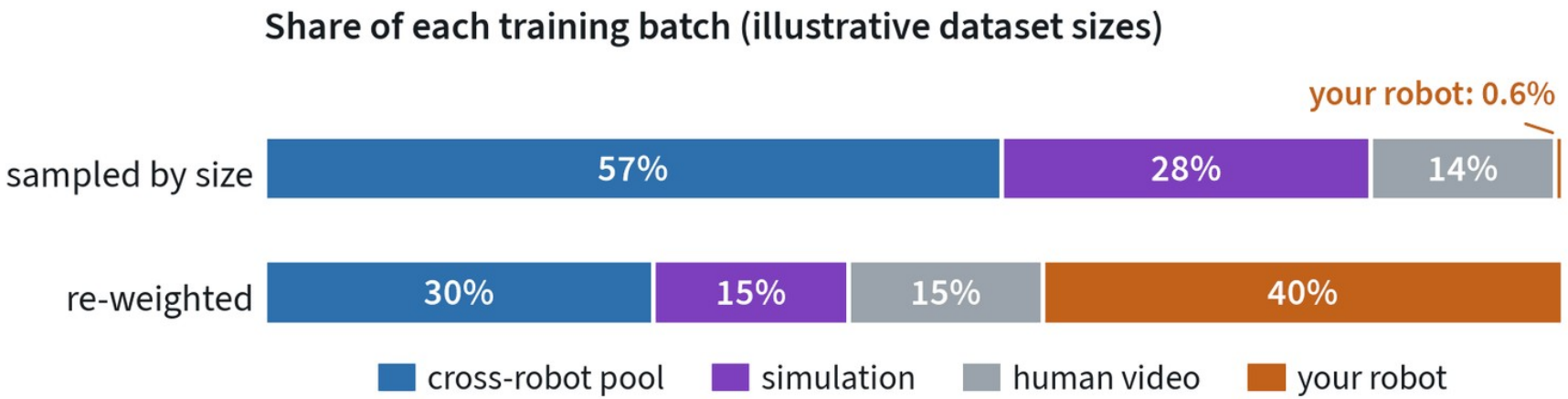
Train on the video as video

- Use it to pretrain the visual and predictive parts of the model.
- Learn actions from robot data alone.

Human video is strongest for what to do and where, and weakest for how hard to press

Mixing datasets

illustrative sizes: 2,000, 1,000 and 500 hours of broad data and 20 hours from the target robot



Sampled by size, the 20-hour target dataset is 0.6% of each batch among 3,500 hours of other data. The model learns everyone else's robot.

Re-weighted, it makes up 40% of each batch, while the broad data still provides variety.

Normalise each dataset with its own statistics.

Agree on one action convention, or tell the model which one each sample uses.

Set the mixture by hand so the target robot is a large share of every batch

Labels, generated variations and digital twins

three ways to get more from demonstrations and robot time that already exist

Language labels are training data

- One fixed sentence per task teaches that sentence. Vague or wrong labels teach the model to ignore language.
- Write several phrasings, label sub-tasks in long episodes, and check a sample by eye.

Generated variations

- In simulation, a handful of demonstrations can be transformed to new object positions and replayed into thousands.
- On real data, generative models change backgrounds, lighting and distractors while the motion stays the same.

Digital twins

- A simulated copy of one specific robot and workspace, checked against it.
- Develop and debug methods, screen candidate policies, and re-run the same starting conditions exactly.

Variation adds visual and spatial variety. It adds no behaviour that was not demonstrated

Common mistakes with robot data

three errors that follow from treating data as a quantity

Counting hours

- A hundred varied, clean hours can beat a thousand repetitive ones.

Trusting labels

- Task descriptions attached to the wrong episodes are common in pooled datasets.

Randomising blindly

- If the real value lies outside the randomised range, randomisation does not help.

A hundred varied, clean hours can beat a thousand repetitive ones

Data quality and curation

What makes a good demonstration, how a large dataset is checked and scored, and how to measure what data is worth.

KEY POINTS

A good demonstration is consistent, decisive, feasible and cleanly recorded.

Past a few hundred episodes, quality has to be scored by program and reviewed by exception.

Measure data by what it adds to evaluation results, not by hours collected.

Run it: Scoring data quality (github.com/wnukmat/robot-learning-notebooks)

What makes a good demonstration

seven properties, most of them decided by the operator instructions before collection starts

Property	What it means	Why it matters
Consistent	One strategy per task, written into the operator instructions	Two strategies for one situation give the policy two answers to average
Observable	The operator acts only on what the robot's cameras show	Otherwise the data contains decisions the policy cannot reproduce
Decisive	Steady speed, with no idle pauses or fidgeting	A policy copies hesitation; pauses teach that doing nothing is valid
Feasible	Inside the robot's joint, speed and gripper limits	Motions at the limits are clipped by the controller and recorded wrongly
Includes recovery	A small share of episodes show a slip or bad grasp being corrected	Otherwise the policy has never seen how to get back
Diverse where it matters	Objects, layouts and starting poses vary. The strategy does not	Variety in the scene generalises. Variety in the method confuses
Cleanly recorded	Sensors in sync, steady frame timing, correct labels and sub-task boundaries	Every one of these, when wrong, is a silent bug

Vary what the world can vary. Hold fixed what the operator chooses

Past a few hundred episodes, score by program

real data: every episode in the LIBERO set scored on two cheap measures

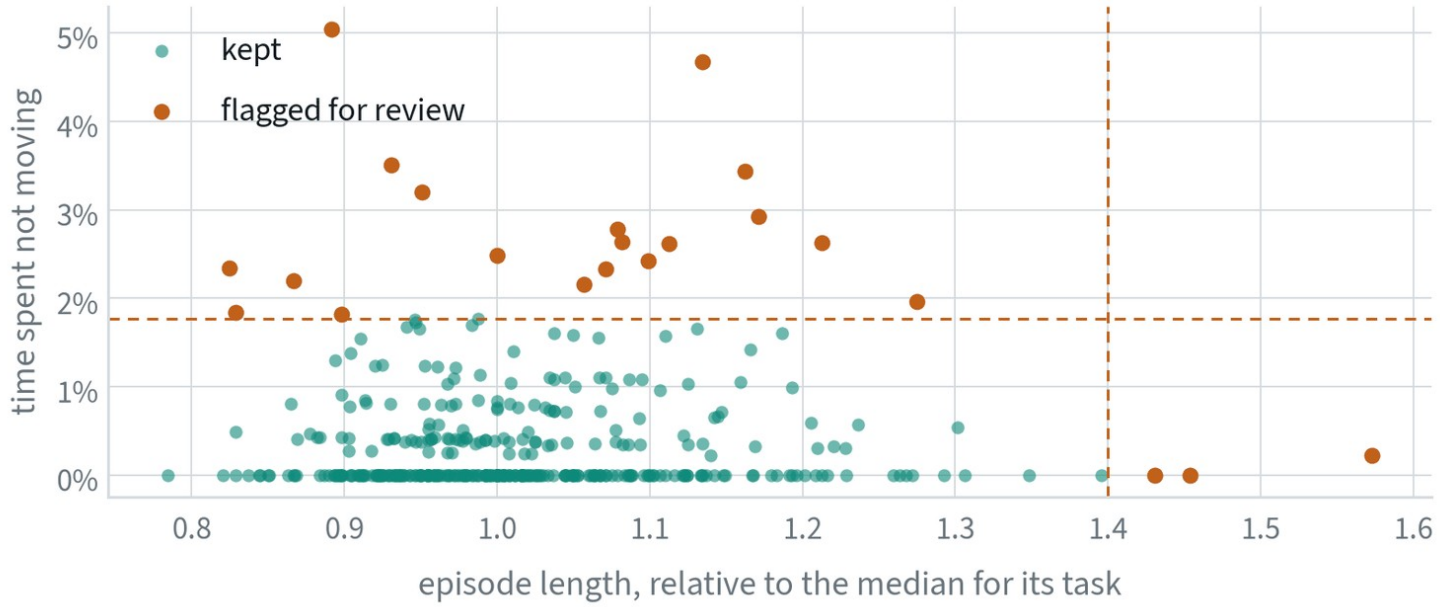
Review by exception

Two cheap measures. Episode length relative to the median for its task, and the share of time spent not moving.

Most episodes cluster near a relative length of 1.0 with almost no idle time.

22 of 379 are flagged, with thresholds at 1.4 times the task's median length and at the 95th percentile of idle time.

A person reviews the 22, because past a few hundred episodes nobody can watch everything.



Programs score every episode, and people look at what the programs flag

The curation pipeline

seven stages, from automatic rejection to planning the next round of collection

Stage	What it does
Sanity checks	Timestamp gaps, dropped frames, sensors out of sync, calibration drift. Reject or repair
Episode scoring	Success label, duration against similar episodes, jerkiness, pauses, repeated grasps
Consistency checks	Group the demonstrations of one task and flag outlier strategies and outlier operators
De-duplication	Embed every episode and drop near-copies, which add volume and no information
Coverage mapping	Find what is missing among objects, scenes and layouts, and plan the next collection
Labelling	A vision-language model drafts instructions and sub-task boundaries. People check a sample
Closing the loop	Failures found in evaluation become the brief for the next round of collection

Closing the loop turns curation from cleaning into a strategy

Scores computed on every episode

simple functions of the recorded motion, with p the hand position at each step and Δt the time between steps

Directness and smoothness

$$\text{path length} = \sum_{t} \|p_{t+1} - p_t\|$$

The total distance the hand travels: a measure of directness

$$\text{jerk} = \sqrt{\frac{1}{T}\sum_{t}\left\|\frac{p_{t+3} - 3p_{t+2} + 3p_{t+1} - p_t}{\Delta t^3}\right\|^2}$$

The root mean square of the third derivative of position: a measure of smoothness

In words

Jerk is large for a hesitant or corrected motion even when its path is short.

Idle time is the share of steps with a near-zero command.

Duration is the episode length.

Relative scores. A value of 1.0 means typical for that task, so one threshold serves every task.

Divide each score by the median for its task, because tasks differ

What the scores find, and why one is not enough

real data: two demonstrations of one task seen from above, and all 379 episodes on two scores

Typical. The left path is 137 cm long and sits inside the envelope of 25 other demonstrations of the task.

Flagged. The middle path is 195 cm, 42% longer than the task median, and wanders outside the others.

Recovery or hesitation. The first is valuable and the second is not, and only a person looking at it can say which.

Different scores, different episodes. Path length and jerk correlate at 0.05. Path length and duration correlate at 0.63.



The scores decide what gets looked at. A person decides what it is

Measuring consistency

resample every demonstration of a task to a common number of steps, then compute the spread at each step

Spread across demonstrations

$$\sigma(s) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \left\| p_i(s) - \bar{p}(s) \right\|^2}$$

At step s, the root mean square distance of N demonstrations from their mean path

In words

Where to look. The spread should be low at a grasp or an insertion.

Align in time first. Warping the time axis so matching events line up removes differences that are only in speed.

Compare operators. This shows whether the spread comes from the task or from one person's habits.

A learned alternative. Ask how predictable each demonstration's actions are from its observations, given the rest of the dataset.

Low spread where the task needs precision is the sign of a clean dataset

The value of one more demonstration falls

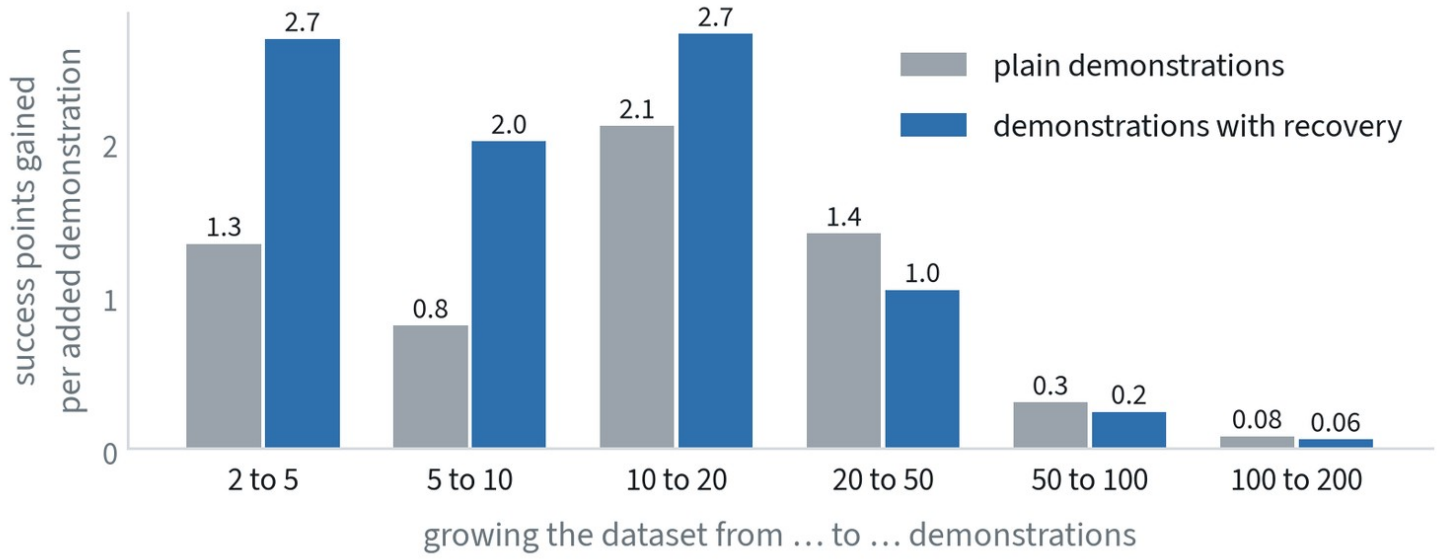
success points gained per added demonstration, from the experiment in the data-efficiency note

Scarce data. Up to 20 demonstrations, each added one is worth between 1 and 3 points of success.

Plentiful data. Between 100 and 200 the gain is under 0.1 points each, a fall of more than twenty times.

Recovery data is worth more early and no more later.

Track a curve per source. Plot the gain as each source grows and notice where it flattens.



Spend on the source whose curve is still climbing

Measuring what data is worth

ways to put a number on a batch or a source before and after paying for it

Ablate it

- Train with and without a batch or a source and compare on the evaluation suite.
- This is the only direct measurement.

Compare properly

- Same starting conditions, trials interleaved in time, intervals reported.
- A data ablation judged on 20 trials measures noise.

Use cheap stand-ins first

- Fine-tune a small model on subsets before committing to a full training run.

Estimate influence

- Newer methods estimate which demonstrations helped or hurt specific outcomes without retraining.
- Approximate, and good for finding harmful episodes.

Count in money

- Points per dollar lets a cheap, weak source be compared with an expensive, strong one.

The useful unit is evaluation points gained per dollar

Common mistakes with data quality

three ways curation goes wrong even when the scores are computed

Managing by hours collected

- Track what each source adds to evaluation results per dollar.

Reviewing by sampling alone

- A random sample finds common problems.
- Rare ones need a score computed on every episode.

Filtering without measuring

- Check that removing the flagged episodes improves the policy.
- Sometimes the unusual ones were the recovery data.

Check that removing flagged episodes improves the policy before removing them for good

Data efficiency and generalisation

How much data a robot needs, what makes it need less, and what the result carries over to.

KEY POINTS

Data efficiency is a curve, not a number. Compare methods at several data sizes.

Efficiency comes from knowledge that is cheaper than new demonstrations.

Spend the collection budget on variety. Name the axis when you say a model generalises.

Run it: Data efficiency: learning curves, a power law, and what pretraining buys (github.com/wnukmat/robot-learning-notebooks)

A learning curve you can run yourself

a small experiment run for this note: a point robot must reach a target without touching the obstacle

Two questions, one experiment

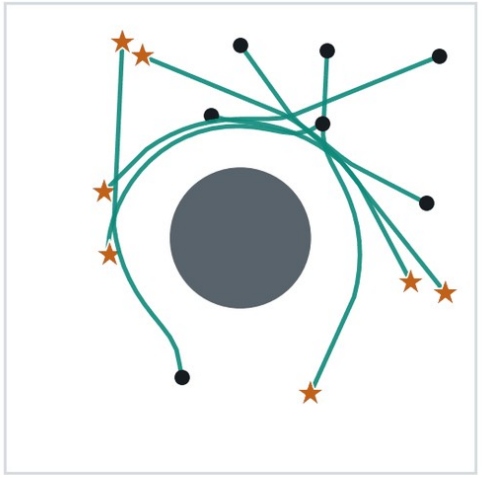
Two questions follow every result. How much data it took is data efficiency. What else it works on is generalisation.

The setup. A small network trained by behaviour cloning, run in the loop on 300 unseen start and target pairs. Mean of six runs.

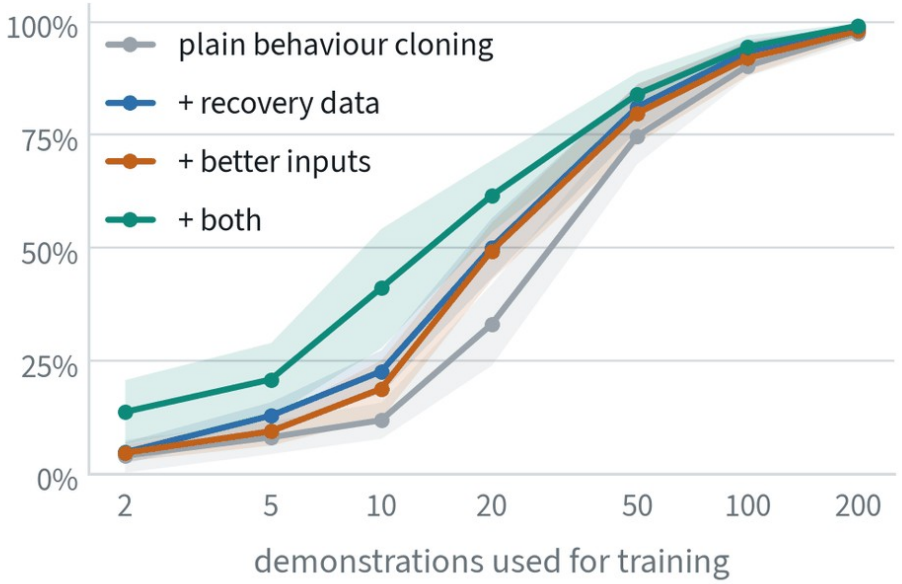
Recovery data. The demonstrator was nudged off course while recording, so the data shows how to get back.

Better inputs. The policy is given the target and obstacle positions relative to itself, the form of the problem that matters.

The task: reach the star, avoid the obstacle



Success when the policy runs in the loop



Data efficiency is a curve, success against the amount of data, and never a single number

The additions matter most when data is scarce

closed-loop success in the toy experiment, by number of demonstrations and training setup

Demonstrations	Plain	+ recovery data	+ better inputs	+ both
10	12%	23%	19%	41%
20	33%	50%	49%	62%
50	75%	81%	80%	84%
200	98%	99%	98%	99%

A method that only looks better at one data size has not been shown to be more efficient

The shape of the curve is a power law

the plain behaviour-cloning curve, replotted as failure rate on log axes

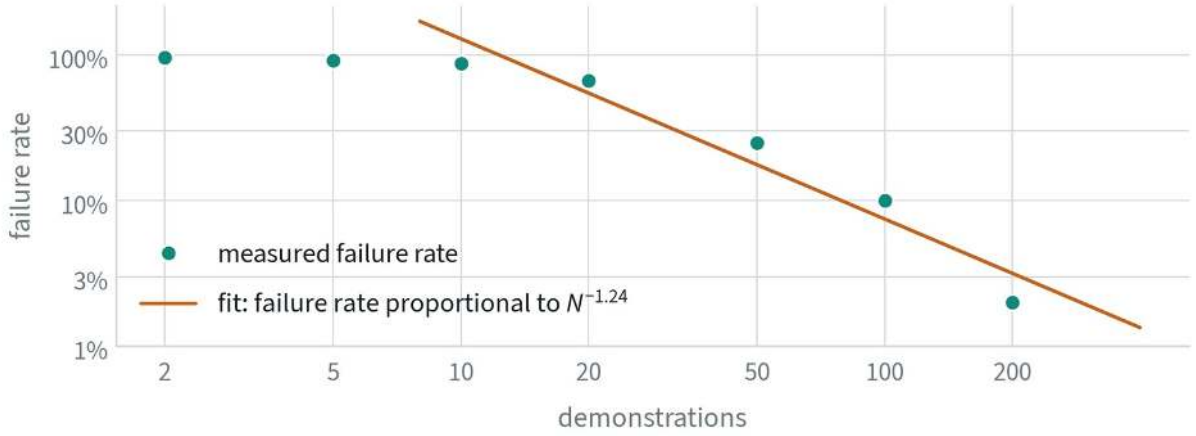
Failure against data

$$\text{failure rate} \approx c\, N^{-\alpha}$$

Failure falls as a fixed power of the number of demonstrations N. Here α is 1.24

$$\log(\text{failure}) \approx \log c - \alpha \log N$$

On log axes the same relation is a straight line with slope minus α



From 10 demonstrations on, the points lie close to a straight line with a fitted slope of minus 1.24. Below about ten the policy fails almost always.

The exponent α is the single number that summarises data efficiency

What buys efficiency

every lever supplies knowledge from somewhere cheaper than new demonstrations

Lever	Where the knowledge comes from	Limit
Pretrained vision and language	Web images and text: what objects are and what words mean	Knows nothing about forces or the robot's body
Pretrained robot policy; co-training	Thousands of hours of other robots' data, or larger related datasets trained alongside	Helps most when robot and tasks resemble the old; the mixture needs careful weights
Action chunking	Fewer decisions per task, so fewer chances to drift	Less reactive within a chunk
Augmentation	Known invariances: lighting, small shifts, colour	Cannot invent new behaviour
Recovery data and corrections	States the policy actually reaches when it errs	Needs a person in the loop
Better observations	A wrist camera, relative actions, body state	Has to be decided before data collection
Simulation, generated data, human video	Physics engines and generative models; people doing the task	The gap to reality; video has no robot actions, so they must be inferred

The first two rows are why "pretrain once, adapt many times" took over robotics

What pretraining buys, measured

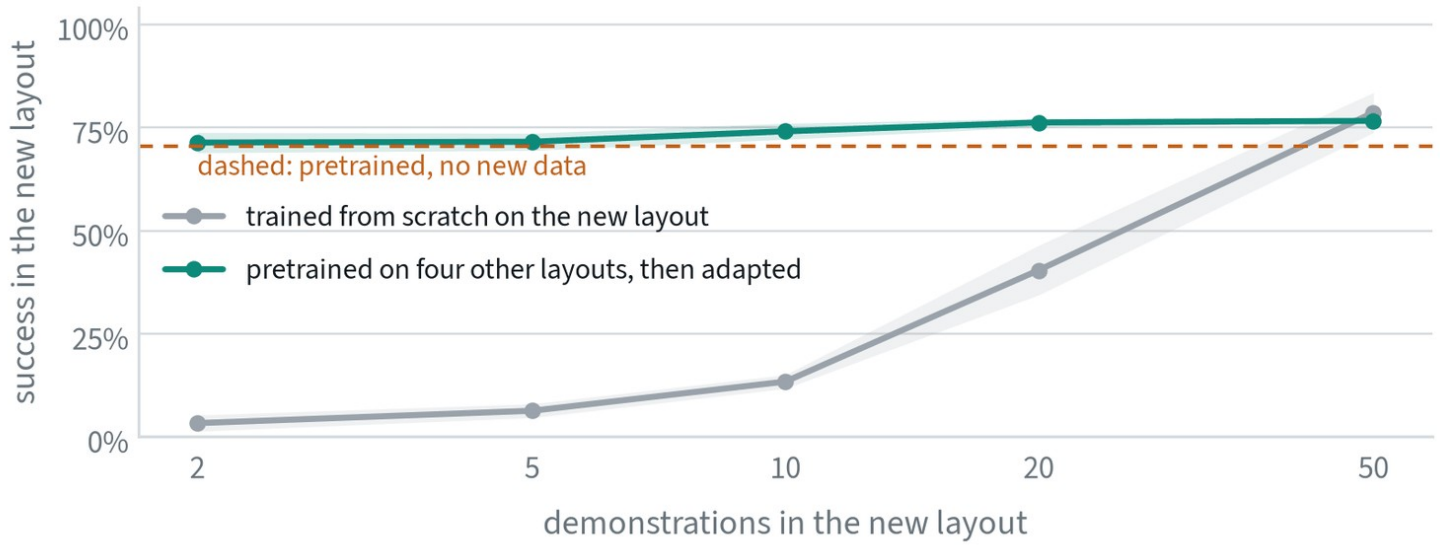
the toy task with the obstacle in a new place; pretraining used 240 demonstrations from four other layouts

Before any new data, the pretrained policy succeeds 70% of the time in the new layout, and 71% after two demonstrations.

From scratch, two demonstrations give 3%, ten give 13%, twenty give 40%, and it takes about 50 to catch up.

The pretrained curve is nearly flat. It gains only six points from 50 new demonstrations, reaching 77% against 79% from scratch.

Why flat. It was adapted conservatively, with the old data mixed in to avoid forgetting.



Pretraining buys a starting level. The last points still come from data in the target setting

How much data is typical

published results give rough orders of magnitude, not rules

One task from scratch, one robot

- Usually from tens to a few hundred demonstrations.

Adapting a pretrained generalist

- A similar count for a new task.
- It tends to give a more robust result.

Places the robot has never seen

- A different matter. It depends on variety far more than on volume.
- Lin and colleagues (2024), with over 40,000 demonstrations: success grew roughly as a power law in the number of different environments and objects.

Spend the collection budget on variety

Generalisation is several things

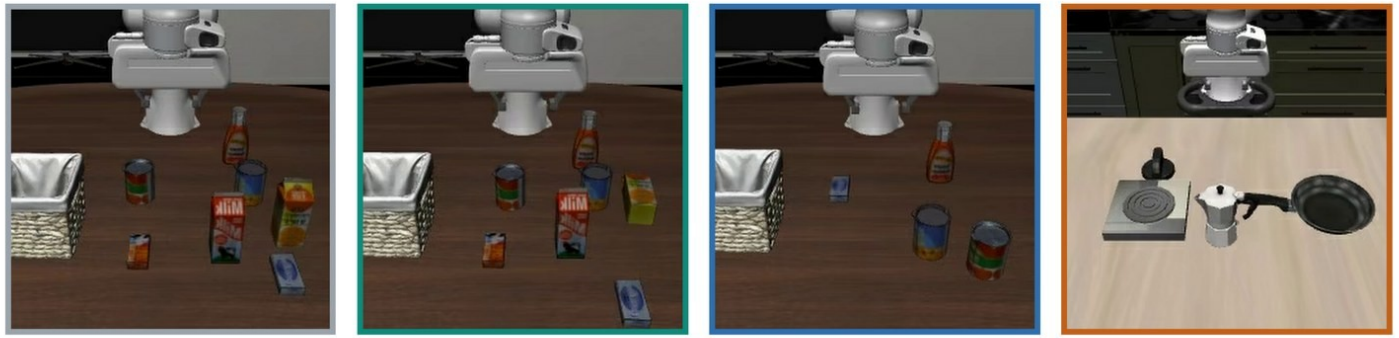
frames from the LIBERO dataset, used here to show the kinds of change

Training scene. Groceries and a basket, as the policy saw them.

Same objects, moved. The policy may have memorised where things were.

A different instruction. The same scene now asks for something else.

A different scene. A stove and a moka pot. Each step to the right asks more of the policy.



Seen in training

“put both the alphabet soup and the tomato sauce in the basket”

New positions

same instruction, objects moved

New instruction

“put both the alphabet soup and the cream cheese box in the basket”

New scene and objects

“turn on the stove and put the moka pot on it”

close to the training data → far from it

"The model generalises" means little until the axis is named

Seven axes of change

what can differ between training and test, why it is hard, and what helps

What changes	Why it is hard	What helps
Object positions	The policy may have memorised where things were	Vary positions during collection; use a wrist camera
Object instances	A new mug has a new shape, colour and handle	Many instances per category; pretrained vision
Scene and lighting	Backgrounds leak into what the policy attends to	Many environments; augmentation
Instruction wording	The same task can be phrased many ways	A language-model backbone; paraphrased labels
Camera placement	The mapping from pixels to motion shifts	Fix the cameras, or vary them on purpose
Task	New combinations of known skills, or new skills	Broad multi-task data; the hardest of all
Robot body	Different joints, reach and gripper	Cross-embodiment training; shared action spaces

Policies today are strongest on the first four rows and weakest on the last two

Measuring efficiency and generalisation

four reporting habits that make a result comparable

Report the curve

- Give success at several data sizes, not one.
- State how many demonstrations and how many hours.

Hold out one axis at a time

- Test new positions, new objects and new scenes separately, so a failure can be traced.

Compare at equal data

- A method given more or better data has not been compared.

Count pretraining

- A model that needs 50 demonstrations after 10,000 hours of pretraining is efficient in new data, not in total data.

Say whether a model is efficient in new data or in total data

Common mistakes with data efficiency

three ways to overstate what a policy has learned

Quoting one data size

- Efficiency is the shape of the curve.
- One point cannot show it.

Collecting volume without variety

- A thousand demonstrations in one scene teach that scene.

Testing on the training layout

- Success where the objects stood during collection says little about anywhere else.

A thousand demonstrations in one scene teach that scene

The data flywheel

Turning what a deployed robot gets wrong into the data that fixes it: what to capture, how to use it, and how to keep the loop honest.

KEY POINTS

- Corrections collected where the policy fails are worth more than the same amount of ordinary demonstrations.
- Train on old and new data together, and keep every fixed failure as a permanent test.
- Measure the loop’s speed and yield, not the size of the dataset.

Run it: The data flywheel (github.com/wnukmat/robot-learning-notebooks)

What a data flywheel is

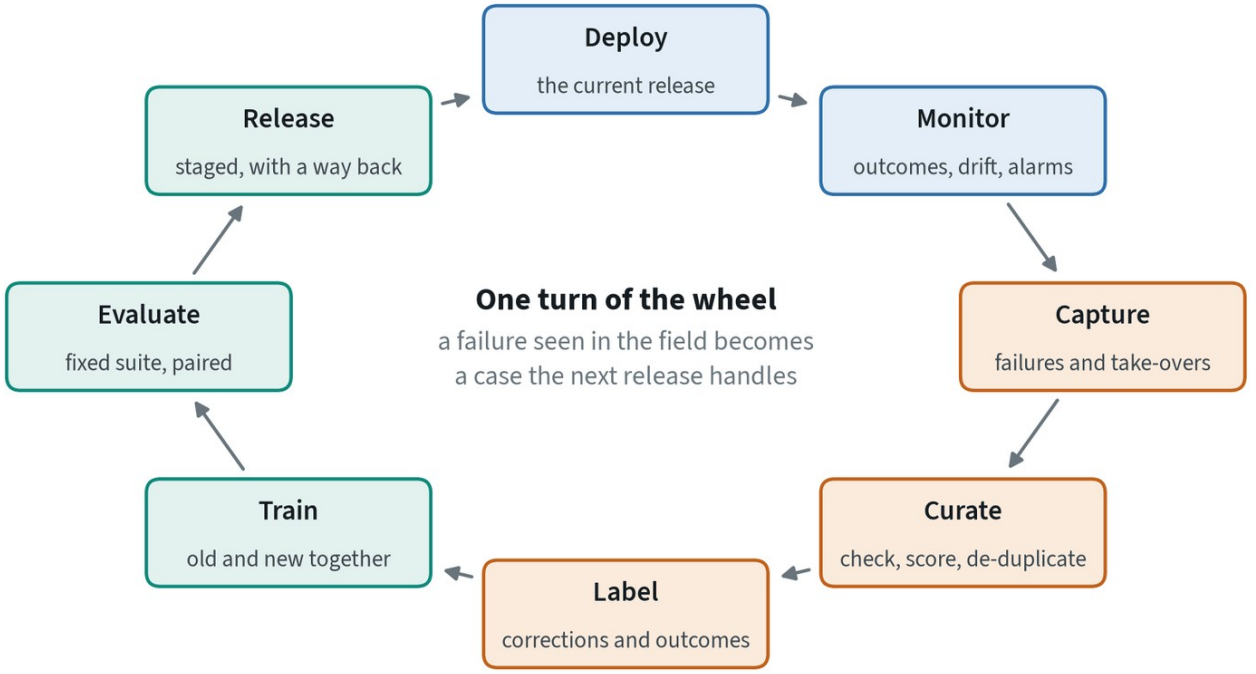
the loop that turns what a deployed policy meets in the field into training data

One turn of the wheel

Eight stages. Deploy, monitor, capture, curate, label, train on old and new together, evaluate on a fixed suite, release in stages.

It feeds itself. A better policy can be trusted with more work, more work produces more data, and more data makes a better policy.

Where the effort goes. The stages in orange take most of the engineering, and are where most flywheels stall.



The loop is simple to draw and hard to run, because each stage has a way of quietly failing

What to capture

a fleet produces far more data than anyone can store, review or train on, and most of it shows nothing new

Keep	Why it is valuable	How it is detected
Take-overs	Marks the exact moment the policy was wrong; what the person does next demonstrates the fix	The operator's control input
Failures	Shows what the policy cannot do	Success checks, timeouts, safety stops, operator reports
Near misses	Recovered slips and retries are failures that almost happened	Repeated grasps, long episodes, force spikes
Unfamiliar situations	Places the training data did not reach	Distance from training data, low confidence, disagreement between predicted chunks
A random sample of everything	An unbiased picture of normal operation, for monitoring and for evaluation	Chosen by chance, not by a trigger

A small random sample is the only way to notice the failures no monitor was built for

Corrections are worth more than new demonstrations

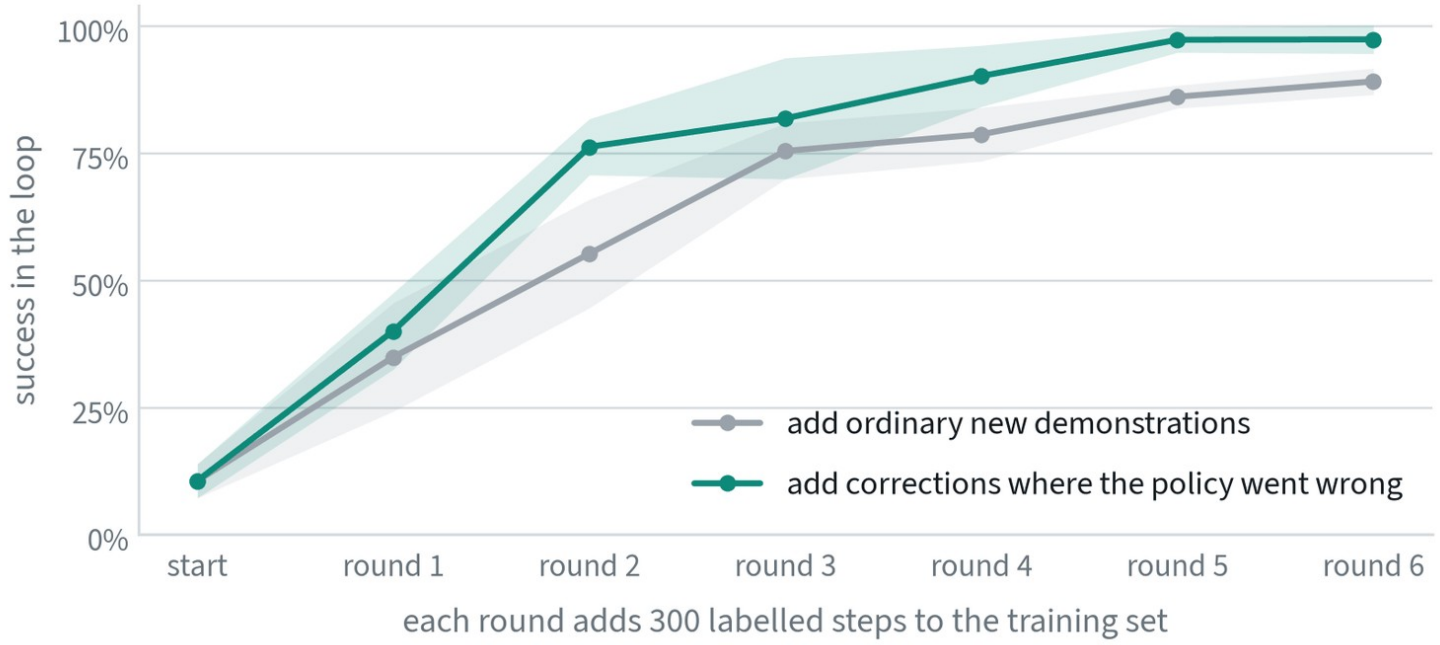
an experiment run for this note: both strategies start from 10 demonstrations and add 300 labelled steps per round

Same expert effort. After two rounds corrections reach 76% and ordinary demonstrations 55%. After five, 97% against 86%.

Why. An expert rarely gets into trouble. The policy's mistakes take it to states the expert never saw, and those need the help.

DAgger. This idea began a family of methods, and it is why take-overs are the most valuable thing a fleet produces.

The built-in limit. The failure rate during collection fell from 86% in the first round to 2% in the last, so each turn finds less.



Corrections put the data exactly where the policy's errors are

The loop in symbols

π is the policy after each round, π the expert, D the dataset, and d the states the current policy visits*

DAgger, then deployment

$$\mathcal{D}_{i+1} = \mathcal{D}_i \cup \left\{ \left(s,\ \pi^*(s) \right) : s \sim d_{\pi_i} \right\}$$

Add expert labels for states drawn from where the current policy goes

$$\pi_{i+1} = \arg\min_{\pi} \sum_{(s,a) \in \mathcal{D}_{i+1}} \ell\left(\pi(s),\ a\right)$$

Retrain the policy on everything collected so far

$$\mathcal{L} = \sum_{\mathcal{D}_{\text{demo}}} \ell + w_{\text{int}} \sum_{\mathcal{D}_{\text{intervention}}} \ell + w_{\text{auto}} \sum_{\mathcal{D}_{\text{autonomous}}} \ell$$

In deployment, interventions and autonomous data get their own weights

In words

The subscript is the whole idea. States are sampled from the current policy's distribution. The experiment implements exactly this.

In deployment the expert cannot label every state. The policy runs until a person takes over, and their actions are recorded.

Interventions are weighted up, because they are scarce and mark exactly where the policy was wrong.

Autonomous data before a take-over is weighted down or dropped. Successful stretches can be kept at a low weight.

The new labels are collected where the policy goes, not where the expert goes

From captured episodes to a training set

illustrative numbers for one week, on a log scale

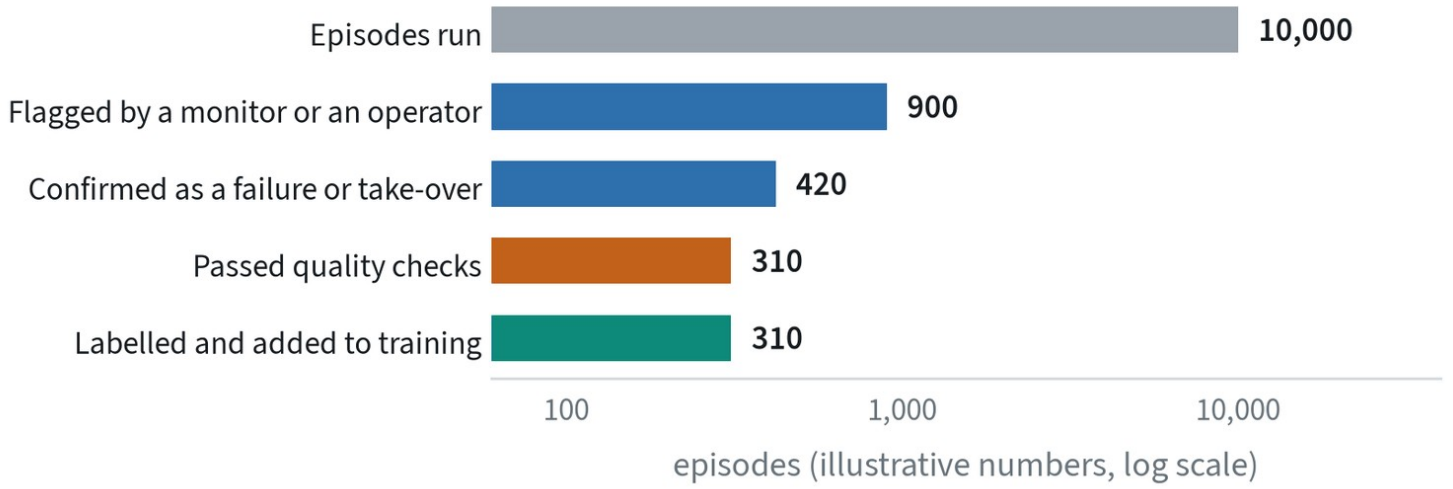
Five steps

The funnel. 10,000 episodes run, 900 flagged, 420 confirmed as a failure or take-over, 310 pass quality checks and are labelled.

Confirm and curate. Monitors raise false alarms, so a person or a second model checks each flag. Then the usual dataset checks apply.

Cut and label. Keep a few seconds before a take-over and the correction after. An autonomous failure has no labels of its own.

File it. Tag every episode with the release, the task and the failure type. Without the release, nobody can tell if a problem was fixed.



About three episodes in a hundred end up in the training set

Training on it without breaking what worked

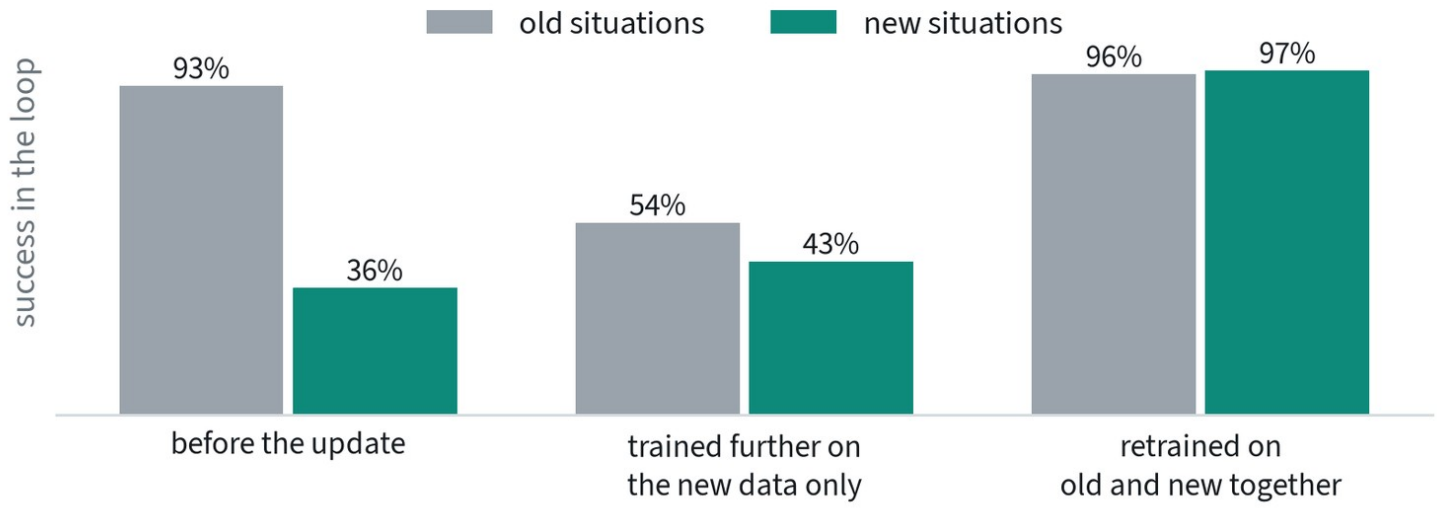
a second experiment: a policy trained on one half of the workspace is updated with data from the other half

New data only. Success on old situations fell from 93% to 54%, and new situations rose only from 36% to 43%.

Old and new together. 96% on old and 97% on new. Retrain from the pretrained start, or mix old data into every batch.

Weight the new data up, within reason. Corrections are few, so sample them more often. Too much overfits to last week's failures.

Keep every earlier fix in the test suite. A failure that was fixed once should stay fixed.



Training on the new data alone damaged the old cases and did not fix the new ones

Choosing what to label

active learning: three signals that rank episodes when labelling capacity is the bottleneck

Signal	How it is computed	Finds
Uncertainty	Disagreement between an ensemble of policies, or between samples from one generative policy	States where the policy does not know what to do
Novelty	Distance from the nearest training examples in an embedding space	Situations the data never covered
Outcome	Failures, take-overs, long or repeated attempts	States where the policy was demonstrably wrong

Outcome is the most reliable signal, because it does not depend on the model's opinion of itself

Keeping the loop honest

a flywheel feeds a model's own behaviour back into its training, which makes it easy to fool yourself

Risk	What happens	Guard
The test set leaks	Field failures are added to training, and the same failures measure improvement	A held-out evaluation set, never trained on, refreshed when looked at too often
Old fixes come undone	A release repairs this month's failure and reintroduces last month's	A regression suite: every fixed failure becomes a permanent test
The policy shapes its own data	It stops attempting what it is bad at, so those cases vanish and look solved	The random sample; track what is attempted, not only what succeeds
Conditions changed, not the policy	Success rises because the week's work was easier	Compare releases side by side on the same conditions, with intervals

Every fixed failure becomes a permanent test, and the evaluation set is never trained on

Measuring the flywheel itself

five numbers that show whether the wheel is turning; teams tend to measure the size of the dataset

Loop time

- From a failure being seen in the field to a release that handles it reaching the fleet.

Yield

- The share of captured episodes that end up in training.
- A low yield means noisy triggers or slow curation.

Interventions per hour, by release

- The outcome the whole loop exists to move.

Failures by type, by release

- Whether each turn removed the failure it targeted.

Cost per point

- What a point of improvement cost in robot hours, review time and compute, against other uses of the same money.

Loop time is the flywheel's speed, and usually the number most worth improving

Common mistakes with the data flywheel

four ways the loop stalls or misreports its own progress

Storing everything, reviewing nothing

- Data that is never curated or labelled does not improve anything.

Fine-tuning on the newest data alone

- It forgets.
- Train on old and new together.

Measuring on the failures you trained on

- Keep a held-out set and a regression suite.

Expecting the wheel to speed up by itself

- A better policy fails less, so each turn takes more deployment to supply.

A better policy fails less, so each turn takes more deployment to supply

PRACTICE · TOPIC 28 OF 32

Training at scale

A recipe for starting any training project, and the engineering around a large run.

KEY POINTS

- Build up from something simple that works; never start with the full system.
- A result you cannot reproduce or compare is not a result.

Build up from something simple that works

the 2019 recipe by Andrej Karpathy, still the best checklist for starting a training project



Know the data. Look at hundreds of samples. Find the duplicates, the bad labels and the imbalance before the model does.

Skeleton and baseline. Full training and evaluation loop, tiny model, fixed seed. Check the starting loss is what chance would give.

Overfit. Make the model big enough to memorise the training set. If it cannot, there is a bug.

Regularise, then tune. Add data first, then augmentation, dropout and weight decay. Search key settings one at a time.

The lesson is the ordering: never start with the full system

The engineering around a run

large training runs are software projects; six practices keep them fast, comparable and recoverable

Practice	What it is	Watch for
Mixed precision	Compute in 16-bit numbers, keep the sensitive parts in 32-bit. Roughly halves memory	Overflow and NaNs
Distributed training	Split each batch across several GPUs for a shorter wall-clock time	Data loading becomes the bottleneck
Experiment tracking	Log every run: configuration, code version, metrics and sample outputs	Run names you cannot decode in a year
Pinned configurations	Set every hyperparameter explicitly; compare the full configuration with the reference run	Defaults that changed between code versions
Checkpointing	Save model, optimiser and step regularly; keep the best checkpoint by validation score	A resumed run that differs from an uninterrupted one
Convergence	A run is done when the validation curve has flattened	A curve that flattened only because the learning rate reached zero

A result you cannot reproduce or compare is not a result

Two small additions that prevent large failures

a small regression model trained with noisy gradients, run for this note

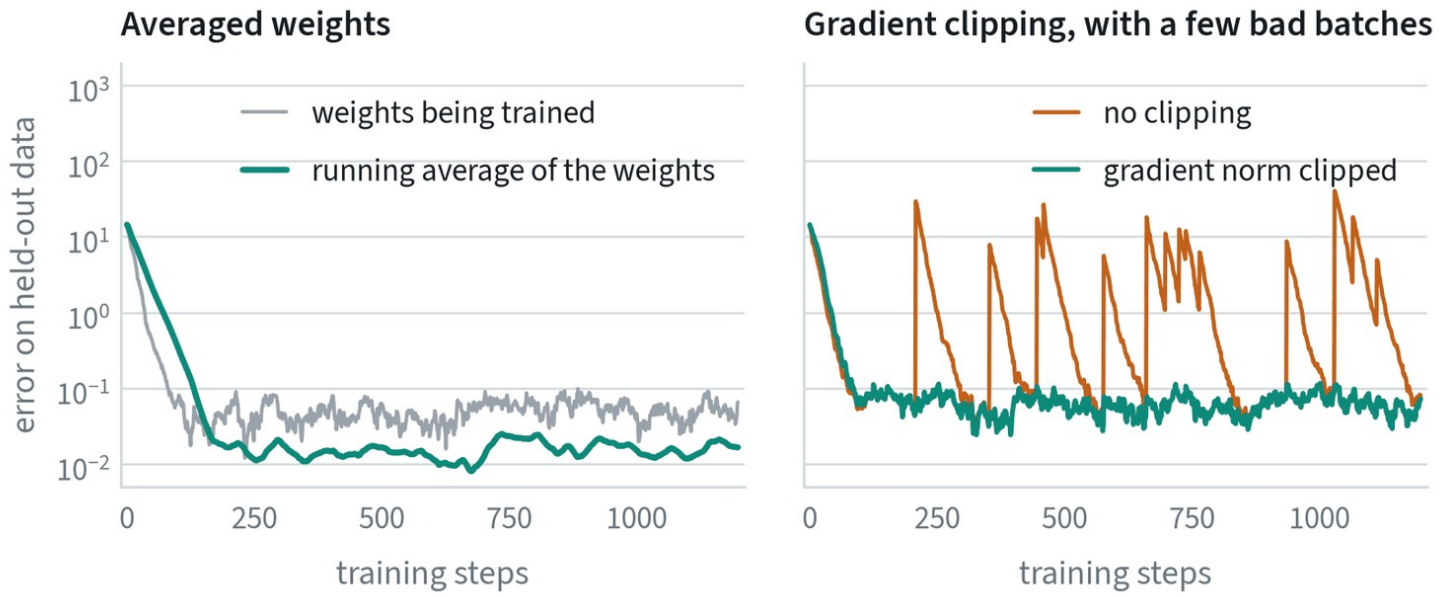
Averaging and clipping

Averaged weights (EMA). A second copy of the model, a slow running average of the trained weights. Its error here is about a third of the raw weights'.

Standard for diffusion and flow matching. For these policies the averaged copy makes a visible difference to the motion.

Gradient clipping. An unusually large gradient is scaled down to a fixed size, so one bad batch costs one ordinary step.

Without it. With 1% of batches corrupted, each bad batch raised the error several hundredfold and took a hundred steps to recover.



Evaluate and deploy the averaged weights, and clip every gradient

Batch size, learning rate and the data loader

a larger batch gives a less noisy gradient, and the slow part of a robot-learning run is often the cpu

Scale batch and rate together

- A larger batch gives a less noisy gradient, which permits a larger learning rate.
- Scaling the two together is a common starting rule.

Gradient accumulation

- When the batch does not fit in memory, add up the gradients of several small batches before each update.
- Same result, longer run.

The GPU may be idle

- Decoding video and resizing images on the CPU can leave the GPU waiting most of the time.

If utilisation is low

- Add loader workers.
- Decode at a lower resolution.
- Pre-compute and cache what does not change.

Check GPU utilisation first; if it is low, the fix is in the loader

Where the time and memory go

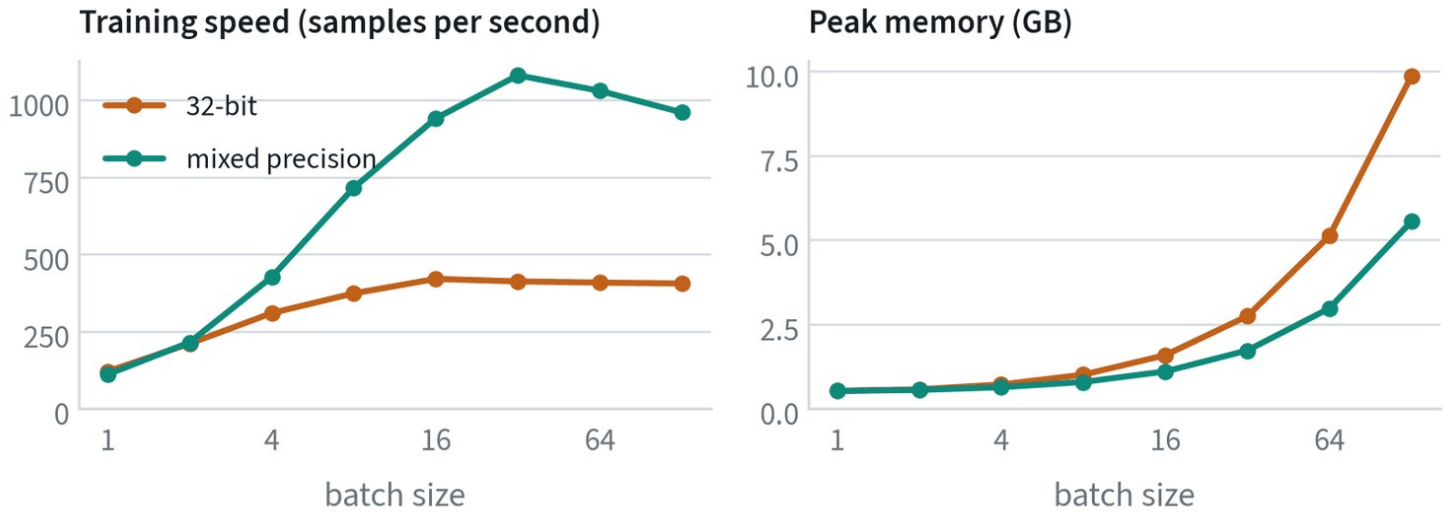
measured on a desktop GPU: one training step of an eight-layer transformer on sequences of 256 tokens

Throughput saturates. At 32 bits it levels off near 420 samples a second from a batch of 16. A larger batch then buys no speed.

Mixed precision. About 1,080 samples a second at a batch of 32, and 5.6 GB against 9.9 GB at a batch of 128. The price is the occasional overflow.

Memory is a straight line. Weights, gradients and optimiser state are fixed. Activations scale with batch × sequence length × layers × width.

Levers before a bigger GPU. Accumulation, gradient checkpointing for about a third more compute, or several GPUs.



Mixed precision is the cheapest speed-up there is: one line of code for 2.6 times the speed

The numbers that go with batch size

three hyperparameters are tied to the batch, and changing one without the others breaks a working configuration

Averaging and clipping, in symbols

$$\bar{\theta}_t = \beta\,\bar{\theta}_{t-1} + (1-\beta)\,\theta_t$$

The averaged copy keeps a fraction β of itself and takes the rest from the new weights

$$g \leftarrow g \cdot \min\left(1,\ \frac{c}{\|g\|}\right)$$

A gradient whose norm exceeds the threshold c is rescaled to norm c

In words

Learning rate. A batch k times larger has √k less gradient noise. The rule of thumb scales the rate with the batch, up to a limit, with a longer warmup.

Adaptive optimisers are less sensitive than plain gradient descent, and a square-root scaling is often safer for them.

Averaging window. About 1 / (1 − β) steps: 1,000 at β = 0.999. Double the batch and halve the steps, and β must change to keep the window in samples.

Clipping threshold. Set it from a logged distribution of gradient norms, a little above the usual value. Do not copy it from another project.

Change the batch and the learning rate, the averaging window and the clipping threshold all move

When the loss becomes NaN

some value overflowed or an impossible operation was attempted, and it has spread to everything

Check	What to look for
The learning rate	Halve it. If the problem goes away, it was too high, or the warmup was too short
The data	NaN or infinite inputs; actions never normalised; division by a zero spread when normalising
Mixed precision	Switch it off. If the run survives, a value overflowed in 16 bits
The first bad step	Log the gradient size every step and find where it jumps. Inspect that batch
The loss itself	A logarithm of zero or a division by a very small number

Work through the causes in order of likelihood, starting with the learning rate

Common mistakes in training

four errors that waste runs

Starting with the full system

- When it fails, there are too many places to look.

Changing several things at once

- The result cannot then be attributed to any of them.

Declaring victory at the step limit

- A run is done when the validation curve has flattened, whatever the step count says.

Evaluating raw diffusion-policy weights

- Use the averaged copy.

Check the curve, not the counter

PRACTICE · TOPIC 29 OF 32

Evaluation

How to tell whether a model, or a robot, is actually better.

KEY POINTS

- Accuracy hides which kind of mistake the model makes; precision and recall show it.
- Ten trials cannot tell 70% from 50%. Report the number of trials and an interval, not just a percentage.
- Break results down before you believe them, and look at the failures yourself.

Run it: Evaluation statistics for small numbers of trials (github.com/wnukmat/robot-learning-notebooks)

Start by defining success

written down before any trial is run, so that two people would score the same episode the same way

The end state

- “The mug is upright on the shelf and the gripper has released it”, not “the mug was put away”.

The limits

- A time limit, a number of allowed retries, and whether a person may reset anything.

What disqualifies

- Knocking over another object, excessive force, leaving the workspace.

Who decides

- A person following the written rule, or an automatic check that has itself been tested against people.

A success definition decided afterwards drifts toward whatever the favoured model did

Four counts and the ratios built on them

for any yes-or-no decision, four counts tell the whole story and every named metric is a ratio of them

Three ratios

$$\text{Precision} = \frac{TP}{TP + FP}$$

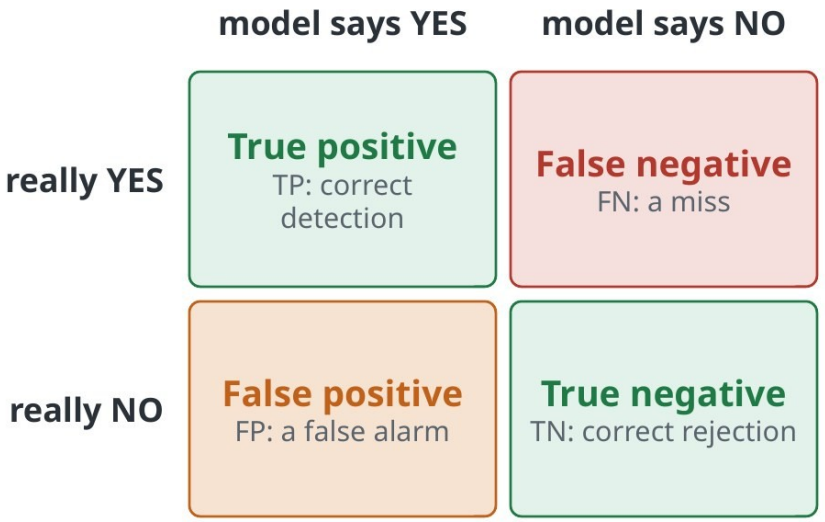
When the model says yes, how often it is right

$$\text{Recall} = \frac{TP}{TP + FN}$$

Of the real positives, how many it found

$$\text{Specificity} = \frac{TN}{TN + FP}$$

Of the real negatives, how many it rejected



Predictions against reality. F1 is the harmonic mean of precision and recall.

Accuracy hides which kind of mistake a model makes; precision and recall show it

Count your trials

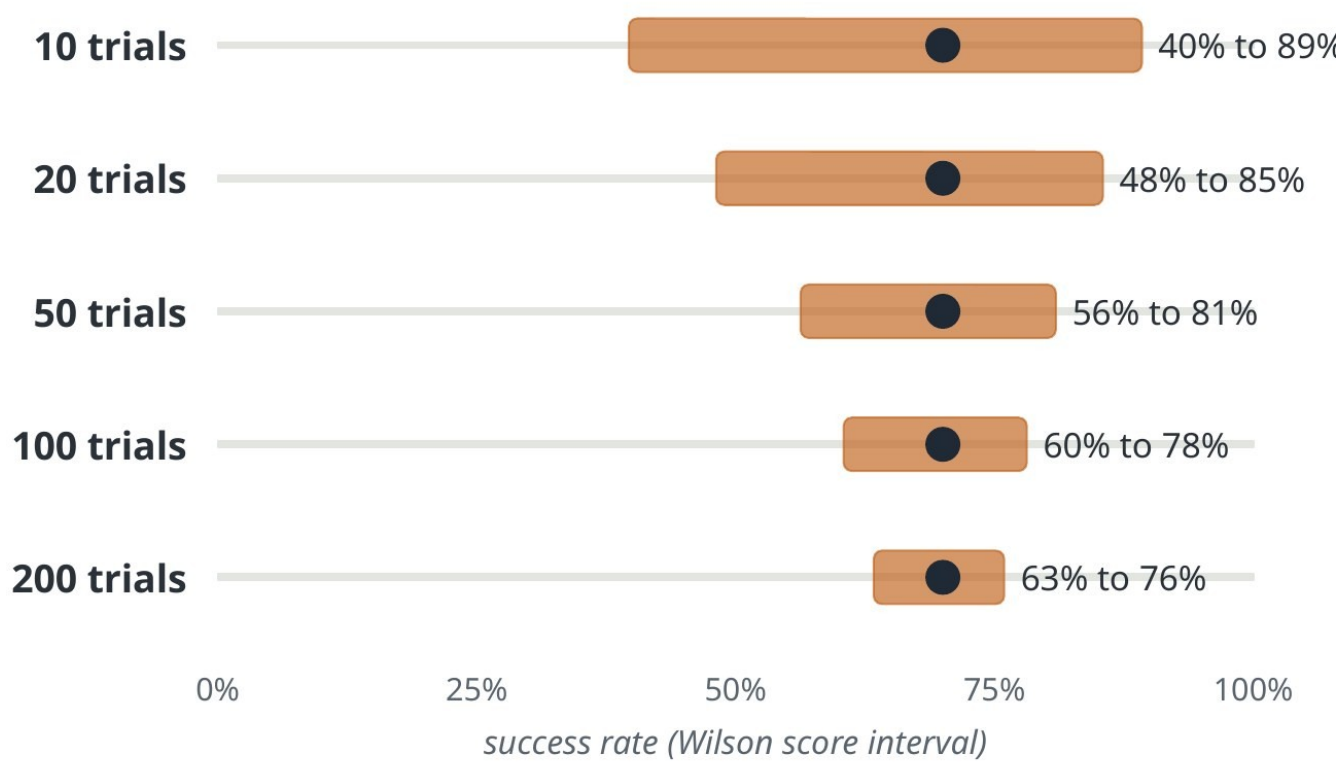
the 95% Wilson interval around a measured success rate of 70%, from 10 to 200 trials

7 of 10 gives an interval of 40% to 89%. Robot evaluations are slow, so samples are small and error bars wide.

It narrows slowly. 48% to 85% at 20 trials, 56% to 81% at 50, 60% to 78% at 100 and 63% to 76% at 200.

Report both. Give the number of trials and an interval with every success rate.

Protocol before the first run. The tasks, the object positions and what counts as success, the same for every model.



Ten trials cannot tell 70% from 50%

The interval, written out

k successes in n trials give an observed rate p̂ = k / n; z = 1.96 for 95%

Textbook and Wilson

$$\hat{p} \pm z\sqrt{\frac{\hat{p}\,(1-\hat{p})}{n}}$$

Textbook: the observed rate plus and minus a multiple of its standard error

$$\frac{1}{1+\frac{z^2}{n}}\left(\hat{p}+\frac{z^2}{2n} \pm z\sqrt{\frac{\hat{p}\,(1-\hat{p})}{n}+\frac{z^2}{4n^2}}\right)$$

Wilson: the centre shifts toward one half and the width never collapses



Computed exactly for 20 trials. Wilson contains the truth 92% to 100% of the time. The textbook interval: 88% at a true rate of 90%, as low as 33% near the extremes.

Use the Wilson interval: 20 out of 20 is 84% to 100%, the honest summary of twenty successes

Comparing two policies needs far more trials

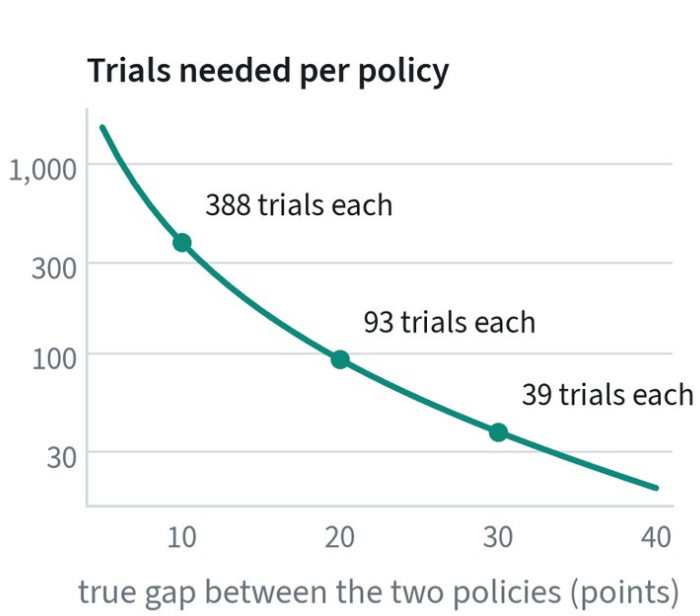
calculated for a baseline at 50% success, 80% chance of detecting the gap at the 5% significance level

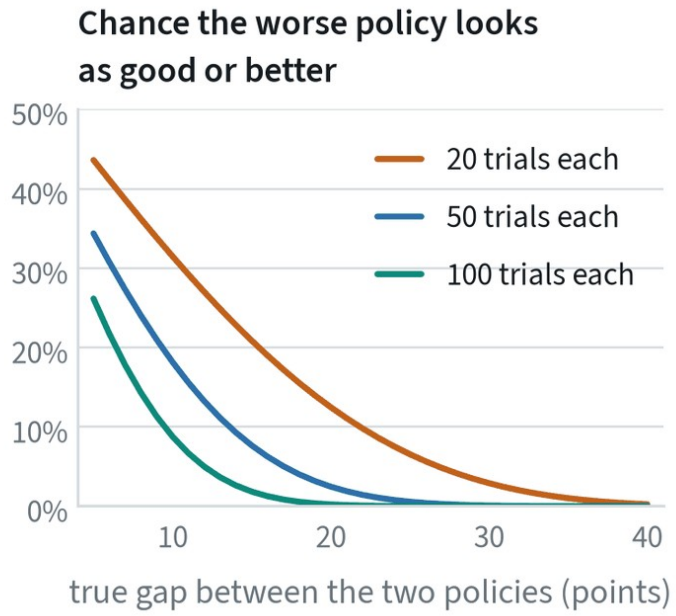
Trials per policy. About 390 to detect a real 10-point improvement, about 90 for 20 points and about 40 for 30 points.

Pair the trials. Run both policies from the same starting layouts and compare layout by layout, so layout differences cancel.

Interleave. Alternate the policies during the session so lighting, battery level and wear affect both alike.

Call small gaps ties. If the intervals overlap heavily, the honest report is “no measurable difference”.





With 20 trials each, a policy that is truly 10 points worse ties or wins 31% of the time

Testing a difference

an interval for the difference between two independent success rates, and the trials needed to detect it

Difference and sample size

$$(\hat{p}_1 - \hat{p}_2) \pm z\sqrt{\frac{\hat{p}_1(1-\hat{p}_1)}{n_1} + \frac{\hat{p}_2(1-\hat{p}_2)}{n_2}}$$

The policies differ if this interval excludes zero

$$n \approx \frac{(z_{\alpha/2} + z_{\beta})^2 \left[p_1(1-p_1) + p_2(1-p_2) \right]}{\delta^2}$$

Trials per policy to detect a true difference δ

$$n \approx \frac{7.85 \times 2\,\bar{p}\,(1-\bar{p})}{\delta^2}$$

The same at 80% power and the 5% level; δ = 0.1 near 50% gives about 390

In words

δ^2 is the whole difficulty. The trials needed grow with the inverse square of the difference.

McNemar's test. With paired trials, count only the disagreements, b and c. Under no difference, b of b + c is a fair coin.

Fix n in advance. Testing every few trials and stopping when it looks significant inflates false positives well beyond 5%.

Many comparisons. Twenty checkpoints or tasks compared at the 5% level produce one false positive by chance.

Halving the difference to be detected quadruples the trials

Partial credit

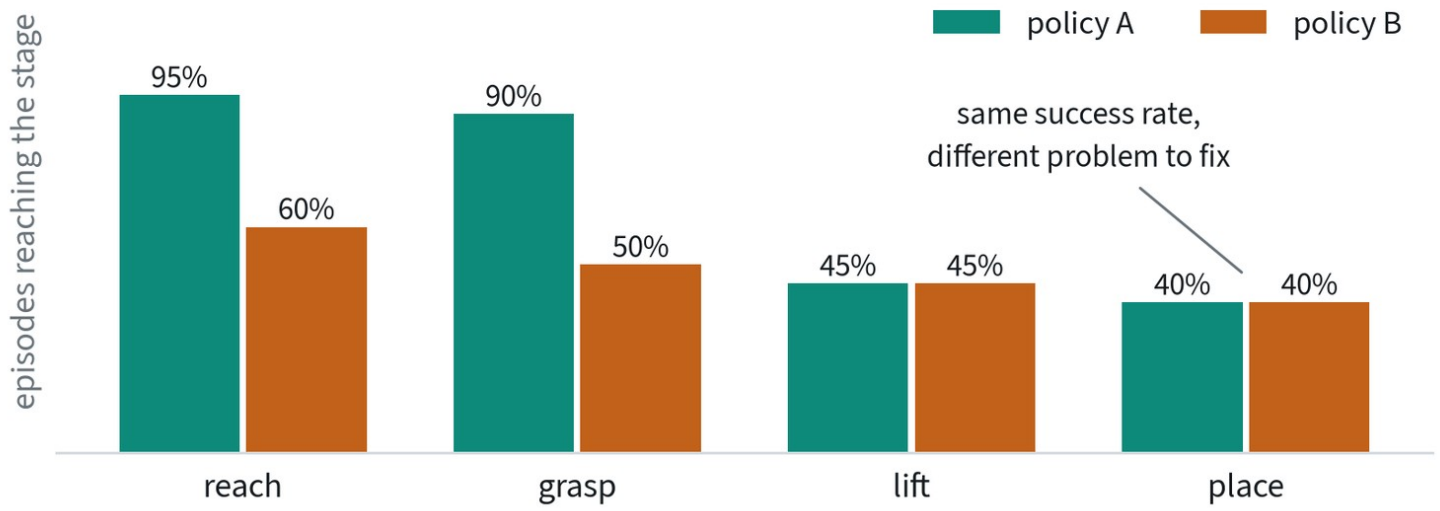
illustrative numbers: the share of episodes reaching each of four stages, for two policies

Same total. Both policies succeed 40% of the time, so the success rate cannot tell them apart.

Policy A reaches and grasps in over 90% of episodes and falls to 45% at lifting. It loses the object while lifting.

Policy B loses 40% of episodes at reaching. It often fails to get to the object at all.

Define the stages first. Reached, grasped, lifted, placed: fixed before testing, like everything else in the protocol.



Stage scores give a finer measure from the same trials and say where to work next

What a single number hides, and cheaper stand-ins

break results down before believing them, and check any proxy against real trials on your own setup

Break results down

- By new object positions, new objects of a familiar kind, new scenes and lighting, and new wording.
- Label each failure, count, and work on the largest group.

Three benchmark traps

- Saturation: above 95%, differences are mostly noise.
- Contamination: test tasks may be in the pretraining data.
- Simulation only: scores may not carry over.

Error on held-out demonstrations

- Quick, and it catches broken runs.
- A weak predictor: a policy can match recorded actions closely and still fail in the loop.

Simulation

- Thousands of repeatable trials.
- Benchmarks such as LIBERO compare methods under identical conditions.

Test the stand-in

- Rank several policies by the cheap measure and by real trials, and see whether the rankings agree.

Held-out scenes

- Keep some scenes, objects and layouts out of all training and tuning.
- The gap to familiar scenes is the most useful single measure of generalisation.

Until its ranking has been checked against real trials, a cheap number is a filter and not a verdict

Deployment metrics

a deployed robot is judged on how much work it does and how much attention it needs

Metric	What it measures
Interventions per hour	How often a person had to step in. The headline number for autonomy
Time between interventions	The same thing from the other side: how long the robot runs alone
Throughput	Tasks completed per hour, usually against a person doing the same job
Success over long runs	Success across hours of continuous work, where small failure rates compound
Recovery rate	The share of its own errors the robot corrects without help
Safety events	Stops triggered, limits reached, contacts above threshold
Uptime	The share of scheduled time the system was able to work at all

A policy that succeeds slightly less often and recovers by itself may be the one to ship

Common mistakes in evaluation

four ways a reported success rate misleads

A percentage with no count

- “80% success” means little without the number of trials.

Unequal conditions

- Comparing models tested on different days, positions or lighting.

Tuning on the test tasks

- Each choice made against the test set leaks a little of it into the model.

Declaring a winner on 20 trials

- A 10-point gap needs about 390 trials of each policy to detect reliably.

Work out how many trials the gap you care about needs before running any

Failure triage

When the robot fails: how to tell whether the controller, the pipeline, the data or the policy is at fault.

KEY POINTS

Check the controller, then the pipeline, then the data, and only then the policy.

Replay a recorded episode. If that fails, the policy is not the problem.

A coverage gap needs more data. A consistency problem needs cleaner data.

Run it: Failure triage on real data (github.com/wnukmat/robot-learning-notebooks)

Find the layer before fixing anything

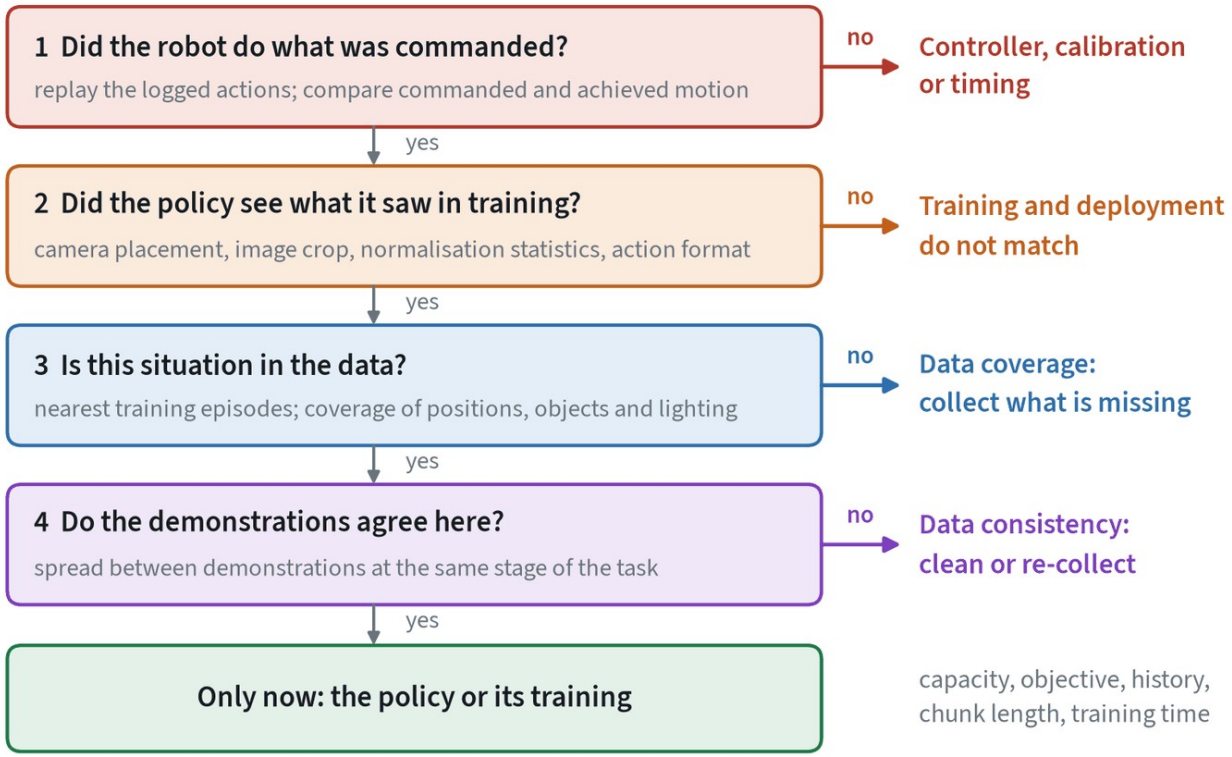
a learned policy sits on a stack: a controller underneath, a data pipeline feeding it and a dataset behind it

Retraining is the instinct. It is the slowest fix and often the wrong one. A fault in any layer looks, from outside, like a bad policy.

Four questions. Did the robot do what was commanded, did the policy see what it saw in training, is the situation in the data, does the data agree.

Ordered by cost. The first two take minutes and need no training run. The last can take days.

Each rules out a layer. Only if all four answers are yes is the fault in the policy or its training.



Work down the list and stop at the first “no”

Check 1: commanded against achieved motion

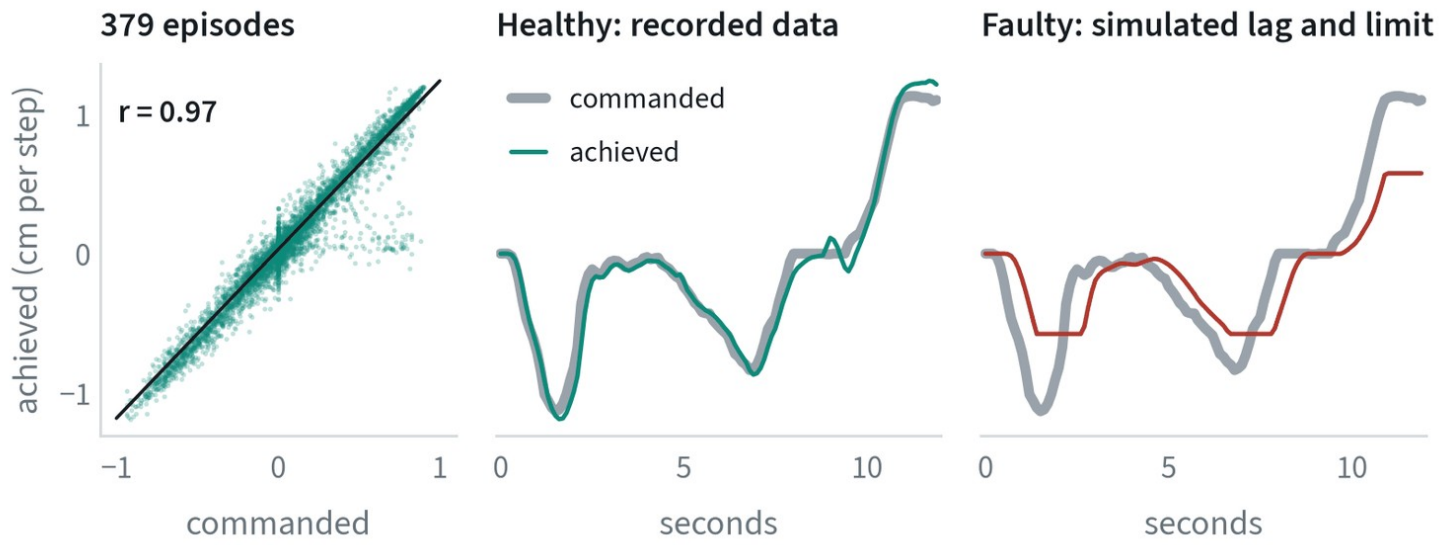
log the actions the policy sent and the motion the robot actually made, and compare them

Healthy. Across 379 LIBERO episodes the achieved motion follows the command with correlation 0.97, as in the middle panel.

Lag. The achieved motion trails the command, as in the simulated fault on the right.

Clipping and offset. Motion cut off at a limit the policy does not know about, or shifted by a constant, which suggests calibration.

Replay is decisive. Send a recorded demonstration's actions to the robot with no policy involved.



Replay a recorded episode. If that fails, the policy is not the problem

Check 2: the policy sees what it saw in training

everything between the sensors and the network; these faults are silent

Check	How to check it
Camera placement and order	Overlay a live frame on a training frame. Confirm wrist and scene cameras are not swapped
Image processing	Same crop, resize, colour order and compression as in training
Normalisation statistics	The checkpoint's statistics are the ones applied, in both directions
Action convention	Same frame, same units, absolute or relative, same gripper sign
Timing	Same control rate; images and body state no staler than in training

Feed a training episode through the deployment code path and compare with the recorded actions

Checks 3 and 4: coverage and consistency

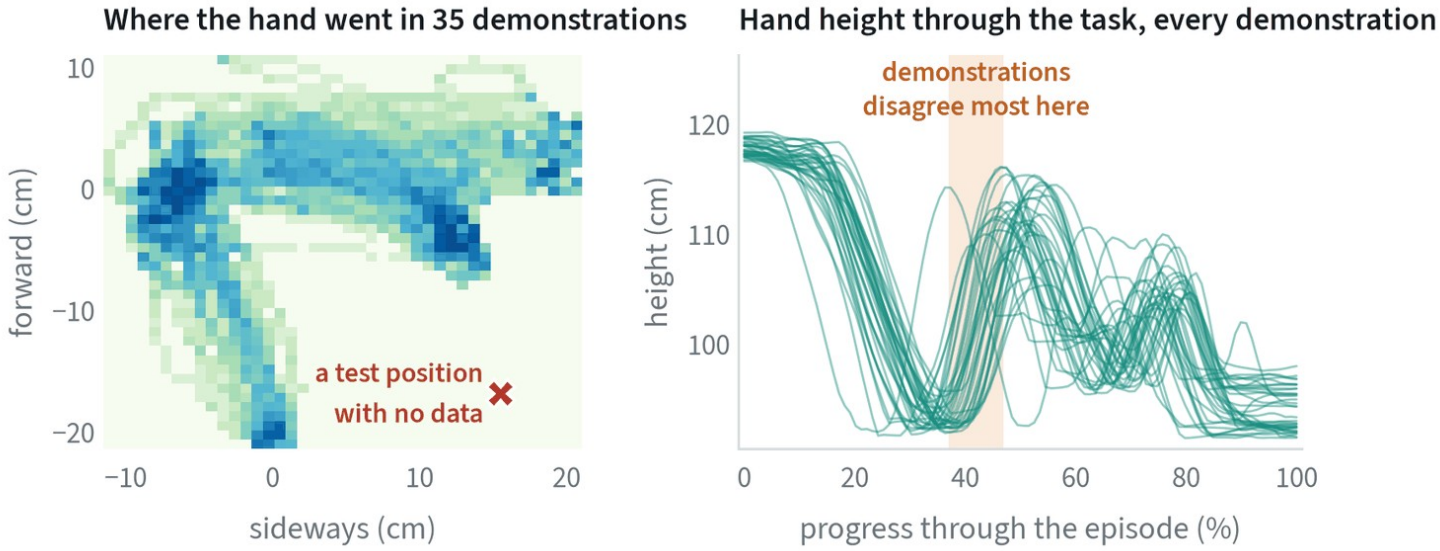
real data: 35 demonstrations of one LIBERO task, where the hand went and how its height varied

Coverage. Find the nearest training episodes to the failure, by object position or image embedding. If they are far away, collect that case.

Left panel. A hypothetical test position sits in a region the 35 demonstrations never visited.

Consistency. Around 40% of the way through, the demonstrations differ by several centimetres. The policy is being taught several answers.

Opposite cures. More data fixes a coverage gap and can make a consistency problem worse. Clean or re-collect instead.



A coverage gap needs more data. A consistency problem needs cleaner data

Coverage can be answered with a number

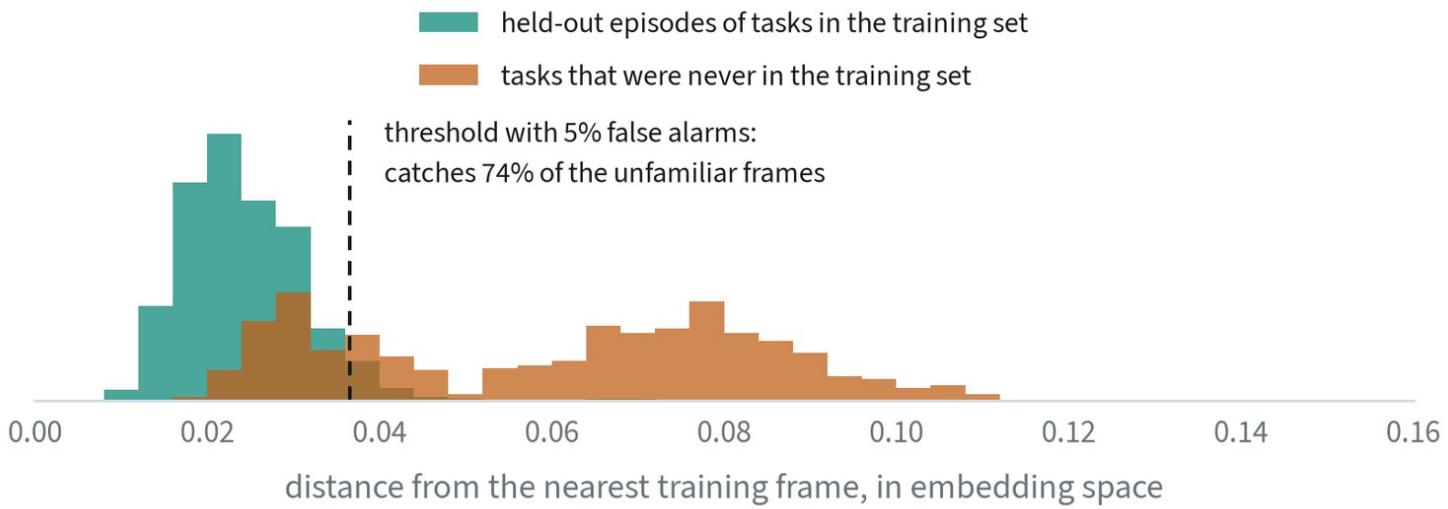
distance from each frame to its nearest training frame, in the embedding of a pretrained image encoder

It separates the groups. Familiar frames cluster near 0.02. Frames from three left-out tasks spread to 0.11. Area under the curve: 0.93.

At a threshold of 0.036, 5% of familiar frames raise a false alarm and 74% of unfamiliar ones are caught.

The missed quarter are new-task frames that look like old ones, typically the start of an episode.

Two blind spots. It ignores changes the encoder ignores, such as a small shift in position, and says nothing about data quality.



Frames far from the training set mean coverage; frames as close as held-out ones mean look elsewhere

Only then the policy, and the habits behind triage

if all four answers are yes, the fault is in the model or its training

Check 5: the policy

- Too little capacity, an objective that averages over valid choices, too much or too little history.
- A chunk too long to react, or a run that had not converged.

Log everything, every episode

- Images, body state, commanded and achieved actions, timestamps and the release that was running.

Keep a failure taxonomy

- Label each failure with a stage and a cause, and count.
- The largest group is the next piece of work.

Reproduce before fixing

- Find the starting conditions that fail repeatedly.
- A fix cannot be verified against a failure that happened once.

Change one thing, then re-run the same failing cases

Common mistakes in failure triage

three ways to spend days on the wrong layer

Retraining first

- It is the most expensive test and it changes several things at once.

More data for a consistency problem

- More conflicting demonstrations do not resolve a conflict.

Diagnosing from the success rate

- The number shows that something went wrong.
- The cause is in the recorded episodes.

The success rate says that something failed. Only the logs say what

Deployment on the robot

Making the model small enough and fast enough, then learning from the field.

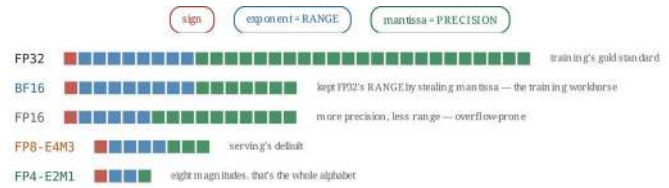
KEY POINTS

- Quantisation is the first tool to reach for; always re-measure task success afterwards.
- Latency is a correctness issue: an action computed from an old image is an action for a world that no longer exists.
- The team that turns field failures into training data fastest improves fastest.

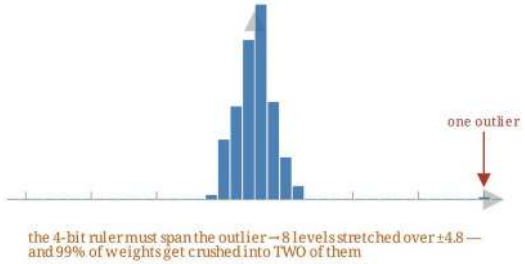
Run it: Quantising a policy until it breaks (github.com/wnukmat/robot-learning-notebooks)

Making the model fit

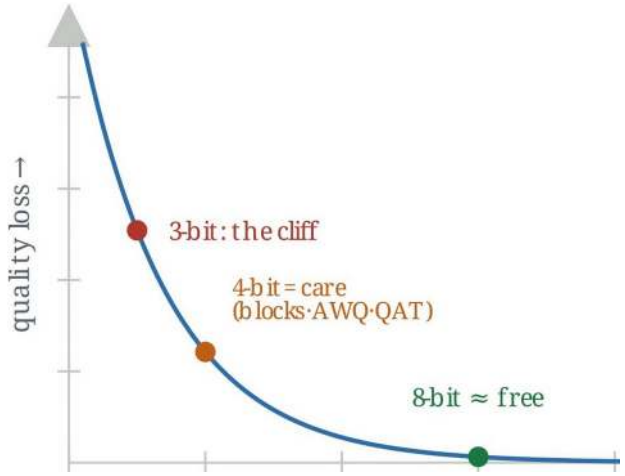
a model that needs a data-centre GPU is no use on a battery-powered robot



Number formats, bit by bit. Fewer bits mean less memory, faster inference, and less range or precision.



One outlier stretches a 4-bit scale so far that almost all the weights are crushed into two levels.



The trade-off, drawn schematically: nearly free at 8 bits, careful work at 4, a cliff at 3.

Three ways to shrink. Quantisation uses fewer bits per number, pruning removes weights, heads or layers, distillation trains a small student.

How low. Sixteen-bit inference is usually free. Eight-bit integers need representative calibration data. Below that, care is needed.

Quantisation is the first tool to reach for; always re-measure task success afterwards

Quantisation, written out and measured

uniform quantisation maps a real number x to one of 2^b integer levels with a scale s and an offset z

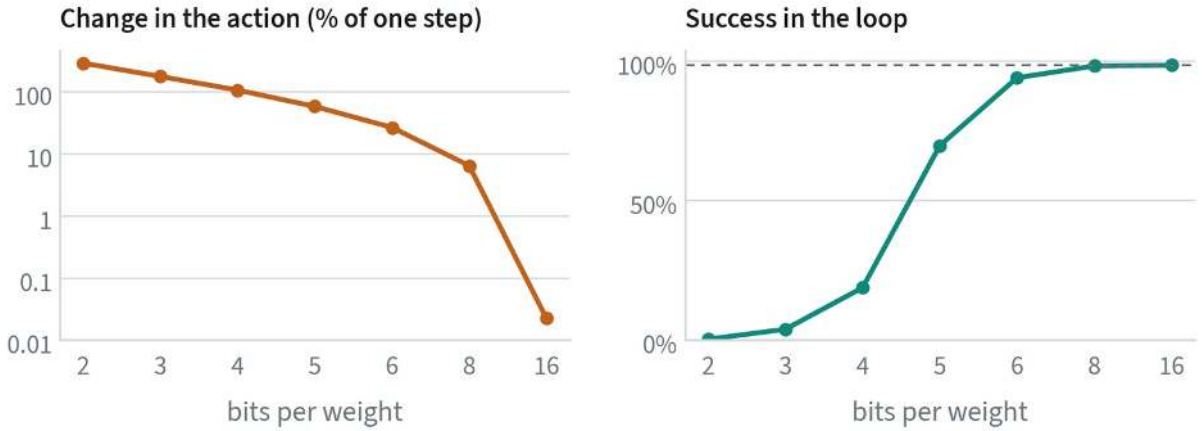
Round, clip, restore

$$q = \mathrm{clip}\Big(\mathrm{round}\big(\tfrac{x}{s}\big) + z,\ 0,\ 2^b - 1\Big)$$

Divide by the scale, round, shift by the offset and clip to the available levels

$$\hat{x} = s\,(q - z)$$

The value recovered; the rounding error is at most s / 2



Measured for this note, a small policy rounded with no retraining. Success: 99% at 16 bits, 98% at 8, 94% at 6, 70% at 5, 19% at 4, 0% at 2.

The error on recorded data does not predict where the cliff is. Only running the policy does

The latency budget

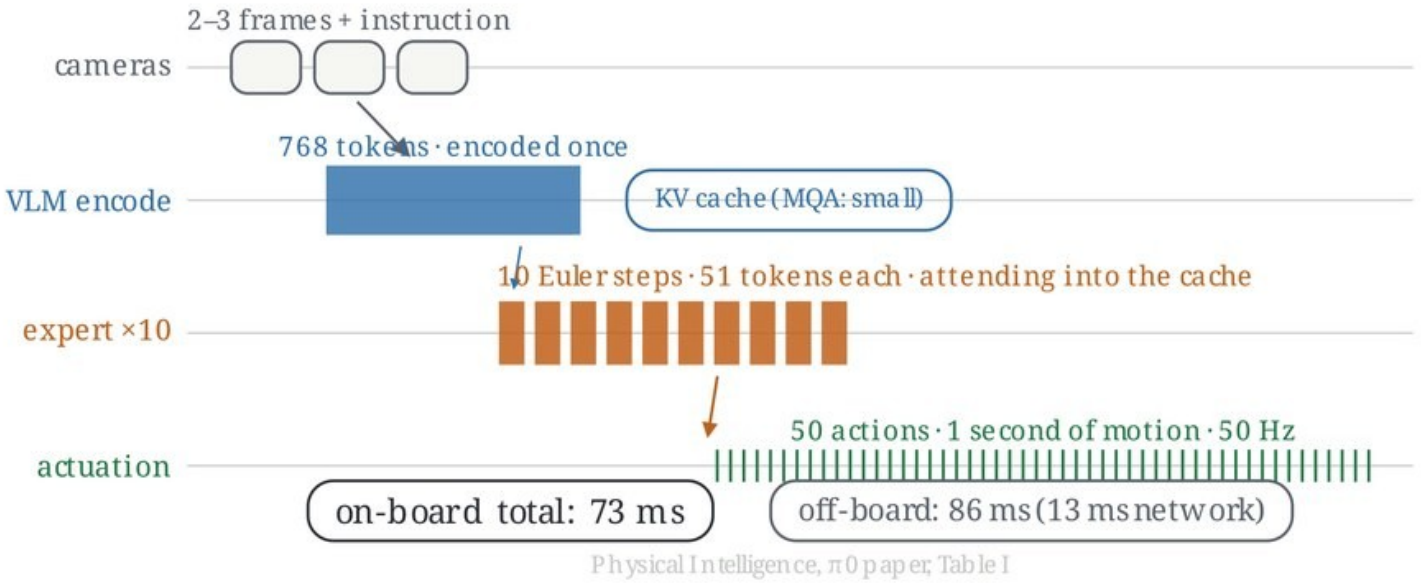
one control cycle of the π_0 model, from the timing table in its paper

73 ms for one second. Images are encoded once and cached, the action expert runs ten short steps, and fifty actions come out at 50 Hz.

Chunk and overlap. Each call returns a stretch of motion, so the next call runs while this one executes and motion never pauses.

Real-time chunking. Plan the new chunk knowing it will arrive late, and blend it into the actions already committed.

Split the work. Run the large model a few times a second and only the small action network every cycle.



An action computed from an old image is an action for a world that no longer exists

Calculating latency: count the tokens

sensing, preparing inputs, running the model and sending the command; the model usually dominates

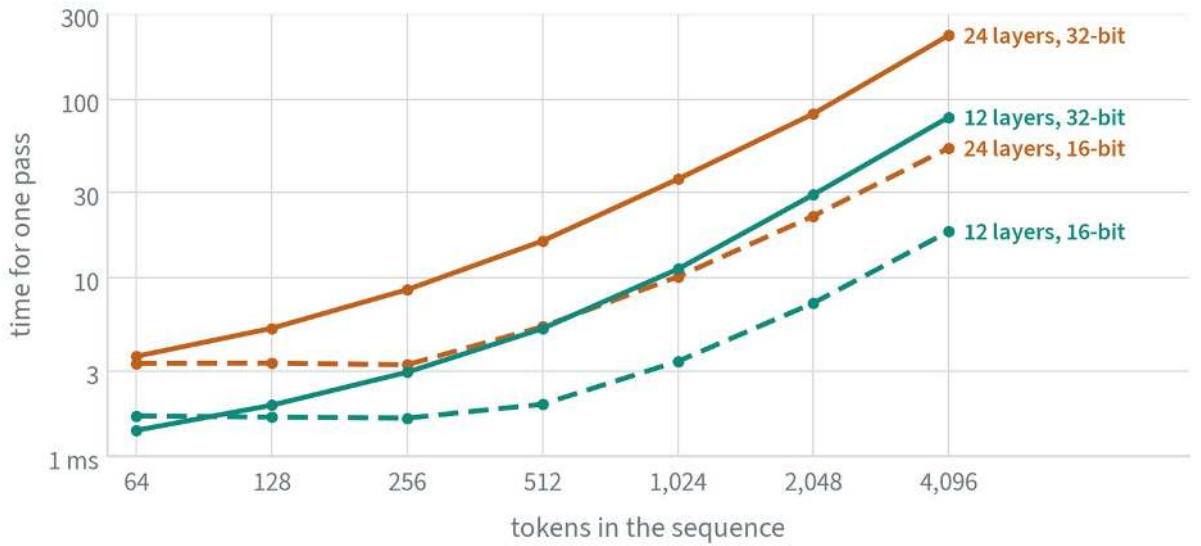
Estimate before building

$$\text{model time} = \text{passes} \times \text{time per pass}$$

Time per pass is set mainly by the number of tokens in the sequence

$$\text{tokens} = \text{cameras} \times \text{frames} \times \frac{\text{height} \times \text{width}}{(\text{patch size})^2}$$

A 224-pixel image in 14-pixel patches is 16 × 16 = 256 tokens; two cameras give 512



Measured on a desktop GPU (RTX 4080). The 24-layer model at 32-bit: 16 ms at 512 tokens, 228 ms at 4,096. Numbers will differ on robot hardware; the shapes will not.

Below a few hundred tokens trimming buys nothing; above that, time climbs faster than the count

The knobs that control it

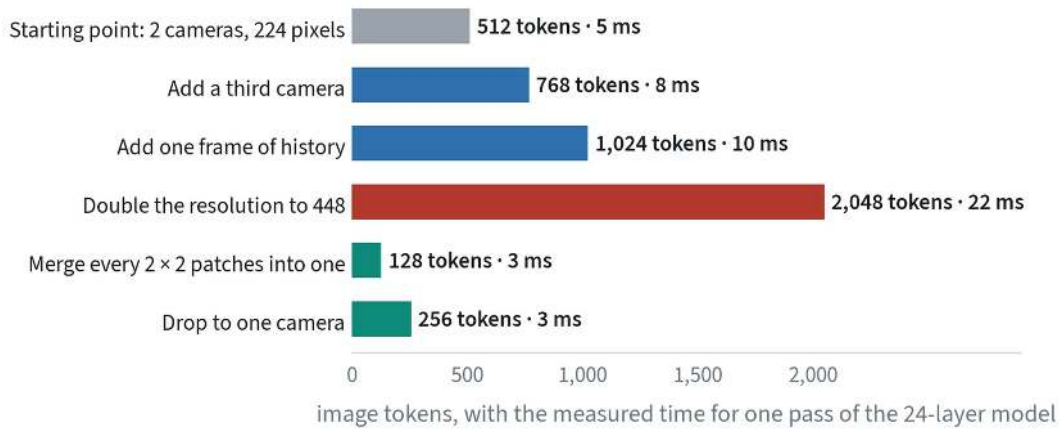
each choice changes the token count, the number of passes or the time per pass, and each has a price

Knob	Effect on time	What it costs
Cameras, frames of history	Tokens grow in proportion	Viewpoints and occluded detail; motion cues and short-term memory
Image resolution	Tokens grow with the square	Small objects and fine detail
Patch size, merging, video compression	Tokens fall, with the square for patch size	Fine detail, or detail the autoencoder cannot rebuild
Refinement steps	Passes grow in proportion	Sample quality, below a handful of steps
Depth and width	Time grows with depth, and faster with width	Capability
Number precision	16-bit was 3 to 4 times faster here; 8-bit goes further	Accuracy, if calibration is poor
Caching	Cameras are encoded once, not once per step	Nothing, if the architecture allows it

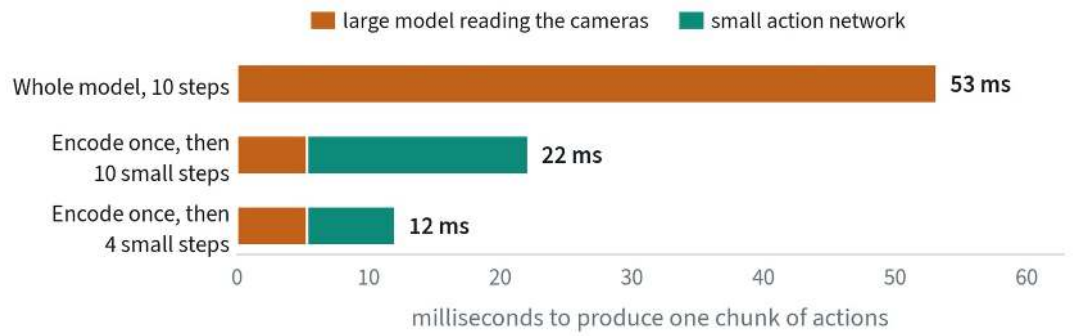
Resolution costs with the square; cameras and history cost their share

Token counts and the time for one chunk

measured for the 24-layer model at 16-bit precision, then combined into the time to produce a chunk of actions



Two cameras at 224 px: 512 tokens, 5 ms. Double the resolution: 2,048, 22 ms. Merge 2 × 2 patches: 128, 3 ms.



Whole model for 10 steps: 53 ms. Encode once, small network for 10 steps: 22 ms. Four steps: 12 ms.

Read once, refine cheaply. A large model reads the scene once, and a small one turns it into actions over several quick steps.

Reaction time is longer. A robot that executes half a second of a chunk before re-planning can take that plus the model time to respond.

Caching and step count cut the time by more than four with the same large model

Where the model runs

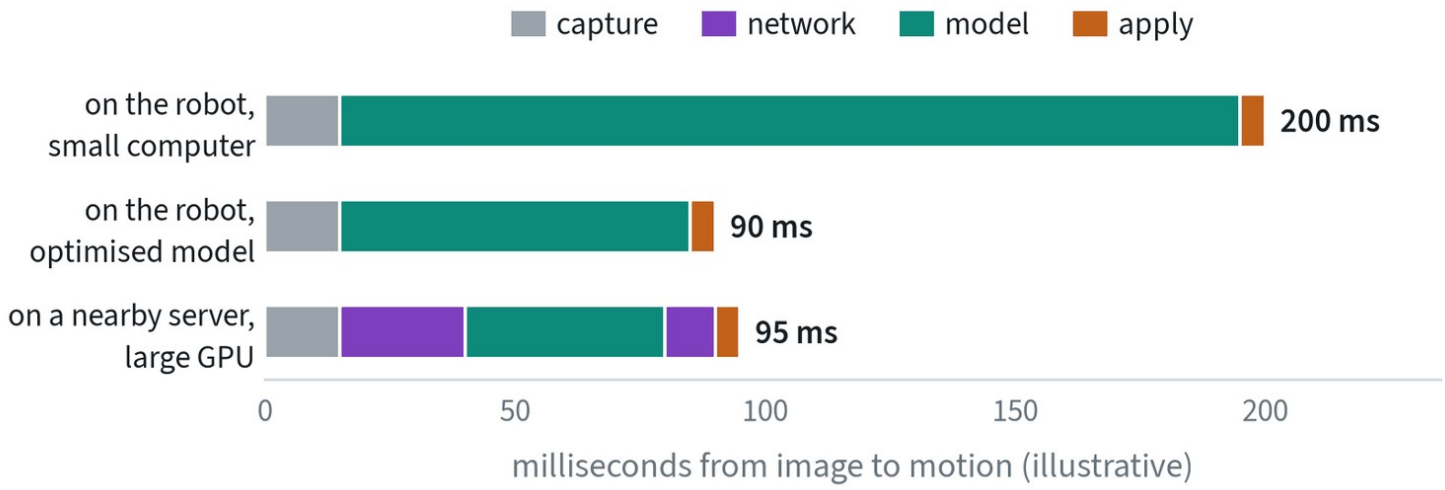
illustrative timings for one inference, from image to motion

Three cases. A small on-board computer: 200 ms, almost all in the model. Optimised on board: 90 ms. Nearby server: 95 ms, 35 of them network.

On the robot. Compute is limited by power, weight and heat. Latency is fixed and predictable. Every robot must be updated.

On a nearby server. Compute is as large as needed and updates happen in one place. Latency depends on the network, and varies.

If the link drops, nothing changes on board. With a server, the robot must stop safely.



Split the work so that a network stall degrades the plan and not the motion

Export, logging and releases

the framework a model is trained in is rarely what runs on the robot, and what runs is more than the weights

Export and runtimes

Unsupported operations. Find out before the architecture is final.

Fixed shapes. Many runtimes fix batch, image size and sequence length at compile time.

Numerical differences. Compare outputs with the original, then compare task success.

Profile and log

Profile each stage separately. The slowest is often not the one expected; CPU preprocessing is a frequent culprit.

Log every episode with images, body state, commanded and achieved actions, timestamps and the release.

Versions and rollback

One unit. Weights, normalisation statistics, preprocessing, action convention and controller settings.

Roll out in stages. One robot, then a few, then the fleet, compared with the previous release.

Keep the way back. Returning to the previous release is one command.

A mismatch between any two parts of a release is a failure that looks like a bad model

Distillation

a small student trained on a mixture of the true labels and the teacher's softened output distribution

The student's loss

$$\mathcal{L} = (1-\lambda)\,\mathrm{CE}(y,\,p_s) + \lambda\,T^2\,\mathrm{KL}\big(p_t^{(T)}\,\|\,p_s^{(T)}\big)$$

Cross-entropy on the true labels, mixed with the divergence from the teacher

$$p^{(T)} = \mathrm{softmax}(z/T)$$

Both distributions are softened by a temperature T

In words

Classifiers and token-based policies use this loss directly.

Generative policies. The student learns to produce in one or two steps what the teacher produces in ten. Few-step action heads come from this.

No labels at all. Run the large policy, record what it does, and train the small one on those recordings.

The catch. The student sees more data than any demonstration set contains, and inherits the teacher's mistakes with it.

The teacher's distribution says which wrong answers are nearly right, which a label cannot

Common mistakes in deployment

three checks that are skipped most often

Checking output difference only

- After quantising, measure whether the task still succeeds.

Measuring the model alone

- The camera, preprocessing and communication all spend the budget too.

Shipping without monitoring

- Then the first sign of a problem is a complaint.

Measure the task and the whole pipeline, then keep watching after release

PRACTICE · TOPIC 32 OF 32

Understanding and safety

Looking inside a model, knowing when not to trust it, and keeping people safe regardless.

KEY POINTS

- Interpretability tools suggest hypotheses; an ablation or a controlled test confirms them.
- Never rely on the policy to be safe; put hard limits underneath it and a human above it.

Run it: Calibration, and knowing when not to trust a model (github.com/wnukmat/robot-learning-notebooks)

Looking inside the model

four tools for asking what a model has learned, and how far each can be trusted

Tool	What it does	How far to trust it
Probing	A small classifier trained on internal features. If it reads out object position, it is encoded	A cheap first check of what a layer knows
Attention and saliency maps	Show which pixels or tokens influenced an output	Good for spotting shortcuts; suggestive, not proof
Sparse autoencoders	Decompose activations into many simple features, each often with a readable meaning	An active research area
Ablation	Removes or zeroes a component, or masks an input such as one camera, and measures the change	The most direct causal evidence

Interpretability tools suggest hypotheses; an ablation or a controlled test confirms them

A probe, run for real

whether a pretrained image encoder's features contain the gripper state, which it was never trained to notice

Features	Accuracy at telling whether the gripper is closed
Always guessing the more common answer	69%
Raw pixels, 16 × 16	81%
Frozen CLIP image embedding	86%

A probe without a baseline shows little, and a successful one shows presence, not use

Knowing when not to trust it

a policy facing an unfamiliar scene should slow down or stop, and to do that the system has to notice

Expected calibration error

$$\mathrm{ECE} = \sum_{m} \frac{|B_m|}{n} \left|\mathrm{acc}(B_m) - \mathrm{conf}(B_m)\right|$$

The gap between accuracy and confidence, averaged over confidence bins

In words

Uncertainty. Use an ensemble, several sampled outputs or calibrated confidence scores, so the system can ask for help.

Overconfident. A small classifier trained on 150 images was right 94.4% of the time while reporting 96.5% confidence.

Temperature scaling. Dividing the logits by T = 1.81 gave 92.6% confidence and cut the error from 2.7% to 2.2%. No prediction changes.

Out-of-distribution detection. Nearest-neighbour distance in embedding space: area under the curve 0.93, 74% caught at 5% false alarms.

The design problem. Stopping on 5% of normal frames is unusable at 10 frames a second, so monitors average over a window.

Calibration holds only on familiar inputs, so a separate check for unfamiliarity is needed

Physical safety sits outside the model

safety layers, outermost first: human supervisor, runtime monitor, learned policy, low-level controller, hardware

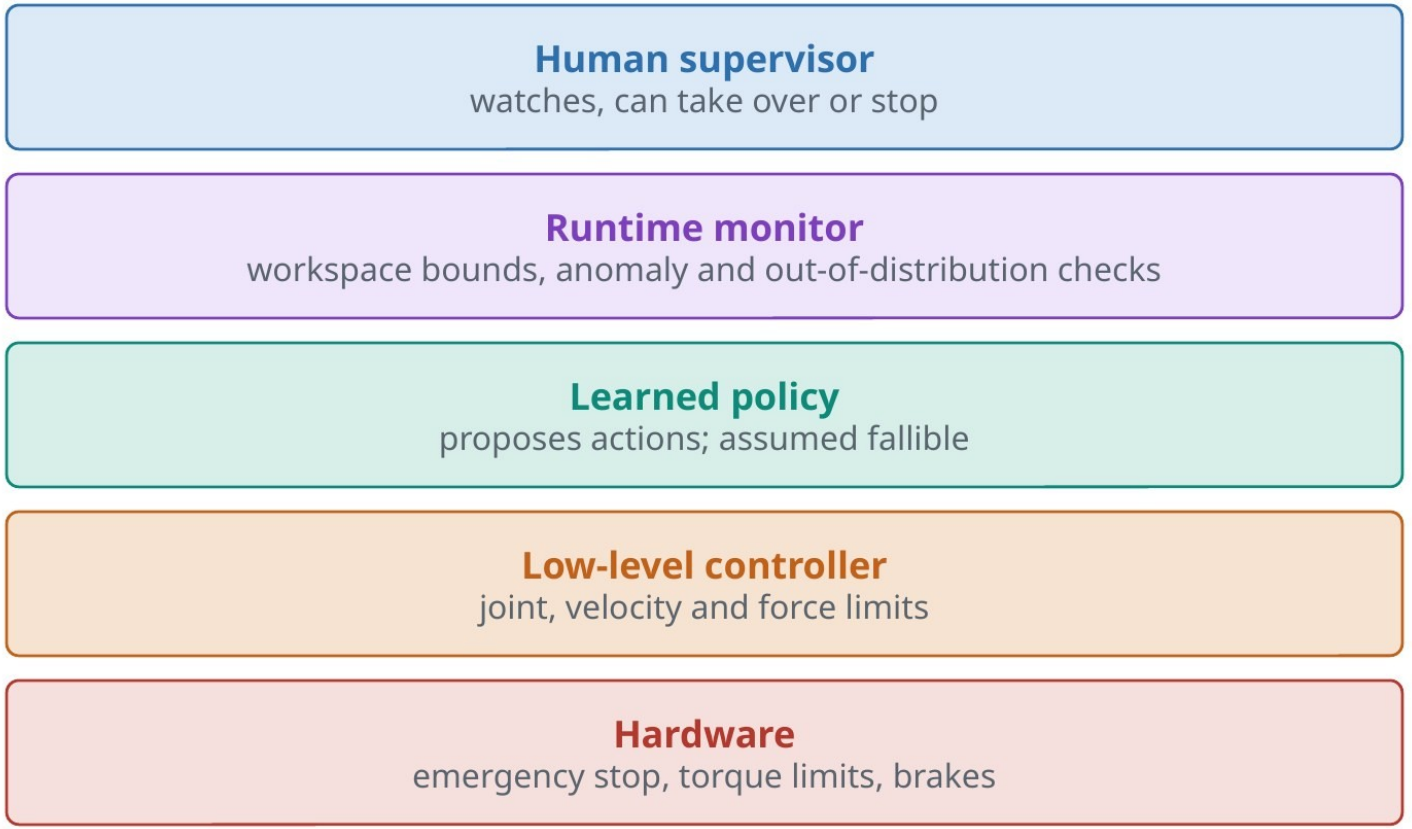
Defence in depth

Independent layers. Use several, each able to stop the robot. The policy is one layer, and it is assumed to be fallible.

Lower layers are simpler. Limits in firmware and hardware are far easier to verify than a neural network.

Fail safe and limit energy. Stop on lost communication or stale commands. Cap speed and force near people.

Test the stop. Exercise the emergency stop regularly, and make sure new team members know where it is.



Never rely on the policy to be safe: hard limits underneath it and a human above it

The standards

robot safety is governed by published standards; this table is an orientation, not compliance advice

Standard	Covers
ISO 12100	How to assess and reduce risk for any machine
ISO 10218, parts 1 and 2	Industrial robots and their systems. The 2025 edition covers working alongside people
ISO/TS 15066	Collaborative operation, with contact force and pressure limits. Folded into ISO 10218 in 2025
ISO 13482	Personal care robots, including mobile and wearable ones

A learned policy does not change which standards apply

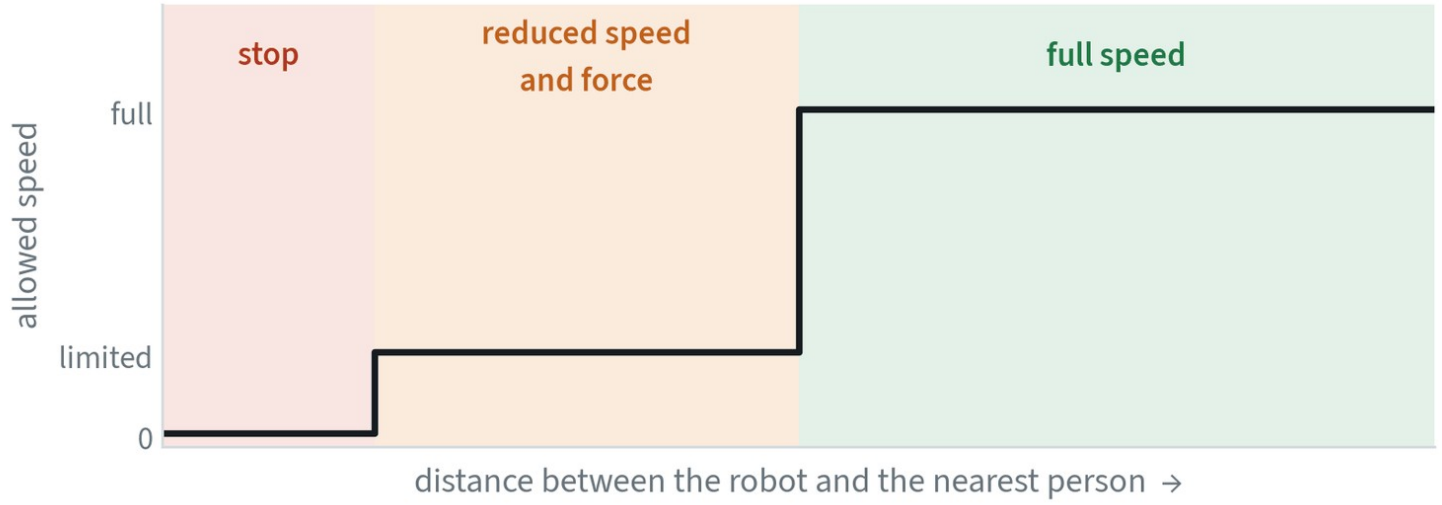
Two standard ways to share a space

allowed robot speed against distance to the nearest person, under speed and separation monitoring

Speed and separation monitoring slows and then stops the robot as a person approaches.

Three zones. Speed is zero inside a stop zone, limited in a middle zone and full beyond it.

Power and force limiting lets contact happen and keeps it below levels that cause harm.



Both are enforced by certified safety hardware, below the learned policy and independent of it

Run-time monitors and instructions as an attack surface

a policy does not announce that it is about to fail, and a robot that follows language can be told the wrong thing

Consistency of its own plans

- Successive action chunks that disagree sharply suggest the policy is unsure.

Distance from the training data

- Compare the embedding of the current image with those seen in training.

A second model as observer

- A vision-language model asked “is the task progressing?” at intervals.

Plain limits

- A time limit per sub-task, and a count of retries.
- When any monitor fires: slow down, stop, or ask a person.

Instructions can be hostile

- They come from a person or from text in the camera view. A printed sign reading “ignore previous instructions” is an input like any other.

Structural defences

- An approved list of tasks.
- Confirmation for anything risky.
- Physical limits where no instruction can reach them.

Measure a monitor's false alarms on normal runs before relying on it

Common mistakes in understanding and safety

three places where a model is given more credit than it has earned

A saliency map as an explanation

- It shows where the model looked, not why it decided.

Trusting raw confidence

- Uncalibrated scores can be high on inputs the model has never seen.

Putting safety in the policy

- Enforce limits outside the model, in layers that can be verified.

A constraint the network learned is a tendency, not a guarantee

What to remember

If the detail fades, keep these six

Data sets the ceiling

Scale, variety and quality of data matter more than the architecture, and quality is measurable.

Tokens, attention, residuals

One block, stacked, reads text, images, video, touch and actions.

Pretrain, then adapt

Start from a foundation model and climb the adaptation ladder only as far as the task needs.

The classical layer still decides

Frames, controllers, timing and the action space fix what a learned action physically does.

Count your trials

Report an interval with every success rate, pair comparisons, and know the trivial baseline.

Find the layer before fixing

Controller, inputs, data, then the policy. A falling loss proves only that the code runs.

CREDITS

Sources of the data, models and measurements

where the numbers and model outputs in this deck come from

Robot camera frames, states and actions: the LIBERO benchmark (Liu et al., 2023), via the lerobot/libero_10 dataset, Apache-2.0.

Touch, pressure and force recordings: the TaF tactile-force dataset (Voxel51/TaF-Tactile-Force), MIT.

Sound clips: public-domain recordings from PDSounds via Wikimedia Commons: pouring water by earthcalling, knocking by stephan.

Handwritten digits: the scikit-learn digits set.

Perception figures: the output of Grounding DINO, SAM 2 and Depth Anything V2 run on a LIBERO frame.

Attention maps: real weights from the public PaliGemma model run on a LIBERO frame.

Generated images: Stable Diffusion 1.5 (CreativeML Open RAIL-M licence), run for the guide.

Toy-model figures (diffusion and flow samples, feature collapse, the simulator gap, domain randomisation, number formats) are from my own technical videos.

Charts described as measured, computed or run for a note were produced for the guide. Charts marked illustrative show the typical shape of a curve, not measured data.

Photograph credits

all photographs are from Wikimedia Commons, cropped

Robots: Unitree G1 by Sayanesy (CC0); Franka by Ims, Flexiv Rizon by MichelleACipriano and MAiRA by Anna Schneider2021 (all CC BY-SA 4.0); Spot by Jonte and TIAGo by PAL Robotics (CC BY-SA 4.0).

Hands: Shadow Dexterous Hand by Richard Greenhill and Hugo Elias (CC BY-SA 3.0); Schunk SVH by NearEMPTiness and Allegro hand with uSkin by XELA Robotics (both CC BY-SA 4.0).

Mechanisms and sensors: strain-wave gear by wdwd, gear set by Pieceofmetalwork and depth camera by Marc Auledas (all CC BY-SA 4.0).

Stock photos inside the CLIP and I-JEPA figures: Pexels.

The full guide, with references for every section, is at www.matthewwnuk.com/guide.html.