Learning from demonstrations

1 Introduction

Suppose we can guide a robot through a task, but we do not know how to write a controller that performs it reliably. We could record what the robot observes and what commands we give it, then train a function to predict those commands. This is *behavior cloning*: learning a policy by supervised learning on demonstrations.

A demonstration is a sequence of observation–action pairs,

$$(o_0, a_0), (o_1, a_1), \ldots, (o_{T-1}, a_{T-1}) \ .$$

The observation o_t might contain camera images, joint positions, and other sensor observations. The action a_t is a command sent to the robot, perhaps a desired hand pose and a gripper opening. A policy π_θ maps recent history of observations to the next action; θ denotes the parameters we fit, such as the weights of a neural network. Once trained, the robot repeatedly observes, applies its policy, and executes the resulting command.

Most typically we condition on a single observation, but that can be very limiting.

Although this can be framed as a classic supervised learning problem, there are several important differences.

These notes assume familiarity with fitting a neural-network model to labeled examples with both discrete and continuous outputs, basic derivatives, and vector arithmetic. We introduce the probability notation as we need it.

First, there may be multiple good actions to take given the same observation, and they may appear in our training data. Simply averaging these actions is generally not a good approach; we will instead model the *distribution* over actions that are taken by the demonstrator.

Second, the robot's predictions induce behavior that changes what it observes in subsequent steps. For example, a small error in placing a gripper changes the next camera image. The policy must then choose an action for that new situation, which may not have appeared in its training data.

We also have to be very aware that an observation need not reveal the full physical *state* s_t. For example, one image may not reveal an object's velocity or a contact hidden behind the hand. Including history can help. In most of the following we will write s as if we can observe the actual state of the world, but generally that won't be the case.

1.1 Averages, variance, and conditional predictions

We'll start with some useful probability background and reminders.

For a discrete random quantity X, its expectation is a probability-weighted average of its possible values. Thus

$$\mathbb{E}[X] = \sum_x p(x)x \ , \qquad \mathbb{E}[f(X)] = \sum_x p(x)f(x) \ .$$

We use capital letters for random variables and lowercase letters for their values: X is a random variable, and x is a possible value. When it's clear, we'll write $p(x)$ when we mean $\Pr(X = x)$.

The weights $p(x)$ are nonnegative and sum to one. For continuous quantities, a probability density replaces p and an integral replaces the sum. A sample average, $N^{-1} \sum_{i=1}^{N} f(x_i)$, estimates this expectation from data.

The notation $p(A \mid O = o)$ means the distribution over actions given an observation o and $\mathbb{E}[A \mid O = o]$ means the average action among cases with concrete observation o. Generally, we think of a policy π as mapping input o to some representation of $p(A \mid O = o)$.

We'll be a little notationally informal and use p for probabilities and densities.

Variance measures squared spread around a mean. For a scalar X with mean μ,

$$\mathrm{Var}(X) = \mathbb{E}[(X-\mu)^2] \ , \qquad \text{standard deviation} = \sqrt{\mathrm{Var}(X)} \ .$$

Variance has squared units; standard deviation has the same units as X. We will use two rules about variance:

$$\mathrm{Var}(cX + b) = c^2 \, \mathrm{Var}(X) \ ,$$
$$\mathrm{Var}(X \pm Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) \quad \text{if } X, Y \text{ are independent} \ .$$

Independence means that conditioning on one quantity does not change the distribution of the other. In particular, note that subtracting two independent measurement errors does not cancel their uncertainty.

1.2 Regression with squared loss

For predicting continuous actions, a standard choice is to let π_θ be a deterministic mapping from observations to actions with training loss

This is standard least-squares regression.

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|\pi_\theta(o_i) - a_i\|^2 \ .$$

For a vector v, $\|v\|^2$ is the sum of the squares of its coordinates. The prediction $\hat{a}$ that minimizes expected squared loss $\mathbb{E}[(\hat{a} - A)^2 \mid O = o]$ is $\mathbb{E}[A \mid O = o]$. Thus squared-error regression aims to produce the *conditional mean action*.

When is this a good idea? It's sensible when the variation in A that we see for the same or nearby values of O is due to simple noise or jitter and doesn't represent meaningful alternative strategies.

But consider a case where we want to grasp a mug and can approach it from the left or the right. For the same observation, we have 7 demonstrations that move sideways -2 cm and 3 demonstrations that move $+2$ cm. The average is -0.8 cm, which might completely fail. Or, see the figure involving going two different ways around an obstacle and imagine averaging the actions.



We describe these action distributions as *multimodal*: there is not a single peak in the density $p(A \mid O)$ and so simple averaging over samples of A doesn't make sense.

1.3 Likelihood and task success

A *distributional policy* predicts $p_\theta(A \mid O = o)$, from which we can sample an action. For independent labeled examples, the *likelihood* is $\prod_i p_\theta(a_i \mid o_i)$. Here we treat the examples as fixed and vary the parameters θ. *Maximum likelihood estimation* means choosing θ to make this product as large as possible. Taking a logarithm turns the product into a sum. Maximizing the likelihood is equivalent to minimizing the *negative log likelihood*

$$L_{\mathrm{NLL}}(\theta) = -\sum_i \log p_\theta(a_i \mid o_i) \ .$$

Also called a probabilistic policy.

There is an interesting question about whether you want to act exactly according to the distribution of actions in the data, or only do the most likely action, or something in between.

For continuous actions, these terms are probability densities, rather than probabilities of exact points. For the following discrete example, they are ordinary probabilities.

Discrete case Let's model the case of two grasp approaches to the mug as a discrete action choice. If n_L examples choose left and n_R choose right at the same observation, a model that assigns probability p to left has loss

$$L(p) = -n_L \log p - n_R \log(1-p) \ .$$

When both counts are positive, we can maximize the likelihood by minimizing $L(p)$. Setting its derivative to zero gives

$$-\frac{n_L}{p} + \frac{n_R}{1-p} = 0 \quad \Longrightarrow \quad p = \frac{n_L}{n_L + n_R} \ .$$

So maximum likelihood reproduces the observed frequencies when the model can represent them. A policy sampling the two approaches with probabilities 0.7 and 0.3 always succeeds under our assumptions. Always choosing the left approach also succeeds. This example does not establish that a controller must be random. It establishes that averaging valid commands can be a bad way to select one.

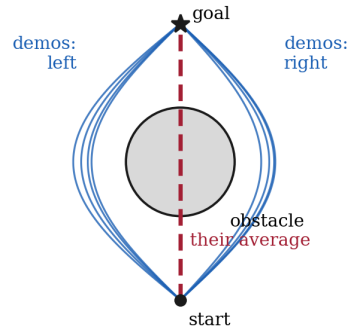


Figure 1: Two demonstrated routes go around an obstacle. Averaging corresponding waypoints can produce a route through it.

Continuous case Now suppose that, for a fixed observation o, we model the scalar action as a Gaussian with mean μ and variance $\sigma^2 > 0$:

$$A \mid O = o \sim \mathcal{N}(\mu, \sigma^2) \ .$$

Read this as: given that the observation is o, the action A has a Gaussian distribution with mean μ and variance σ^2. The vertical bar means "given," and $\sim$ means "is distributed as." Here o is a particular observation value (a lowercase letter, not zero).

Its density is

$$p(a \mid o; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right) \ .$$

For N independent demonstrated actions $a_1, \ldots, a_N$ at this observation, the negative log likelihood is

$$L(\mu, \sigma^2) = \frac{N}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^{N} (a_i - \mu)^2 \ .$$

When the examples are not all identical, we can maximize the likelihood by minimizing this loss. Setting the derivatives of L with respect to μ and σ^2 to zero gives

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} a_i \ , \qquad \hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N} (a_i - \hat{\mu})^2 \ .$$

Thus maximum likelihood fits the mean to the average demonstrated action and the variance to the average squared deviation from that mean. With a fixed variance, learning the mean by maximum likelihood is exactly squared-error regression.

To turn this into a policy for different observations, we can use a neural network that takes o as input and predicts a mean $\mu_\theta(o)$ and a variance $\sigma^2_\theta(o)$. We train its weights θ by minimizing the negative log likelihood over the demonstrated observation–action pairs.

To ensure a positive variance, let the network output an unrestricted real number ν and set $\sigma^2 = \exp(\nu)$. Thus the network predicts the log variance. Adding a small constant $\epsilon > 0$, giving $\sigma^2 = \exp(\nu) + \epsilon$, keeps the variance away from zero for numerical stability. The mean μ can be output directly.

For the earlier mug data, maximum likelihood gives $\hat{\mu} = -0.8$ cm and $\hat{\sigma}^2 = 3.36$ cm^2. The fitted Gaussian describes the spread, but its single peak lies between the two demonstrated actions. Importantly, predicting a mean and variance does not let one Gaussian represent two separate peaks.

2 *Flow matching and diffusion policies*

We have seen that a single Gaussian can be a poor model of the actions in our data. Our target, for each observation o, is the distribution of actions a demonstrator would take in that situation. We use the demonstration data to estimate it. In the mug example, the data suggest probability 0.7 for the left approach and 0.3 for the right. If the actions within each approach varied slightly, we would want two broader modes, each covering a cluster of similar demonstrated actions.

We could use a more complicated family of distributions, but then we would have to decide how to represent it and how to draw samples from it. Another approach is to learn a procedure that transforms a sample from a simple distribution into a sample from our estimate of the demonstration distribution.

We'll study *flow matching*. The idea is to start with a random value and make a sequence of small changes to it. A neural network tells us which way to move at each step, using the robot's observation as an input. We'll first work with a scalar action. The same procedure works for a vector, including a whole sequence of future actions. Figure 2 shows such a *flow field* and some paths that follow it.

2.1 *Training on paths from noise to actions*

$\mathcal{N}(0,1)$ is the Gaussian or *normal* distribution with mean 0 and variance 1.

Suppose we have a demonstrated action a at observation o. Draw a noise value z from $\mathcal{N}(0,1)$, independently of the demonstration, and consider the straight path

$$x_t = (1-t)z + ta \quad , \qquad 0 \leqslant t \leqslant 1 \quad .$$

At $t = 0$ we are at the noise value, and at $t = 1$ we are at the demonstrated action. The velocity along this path is

$$\frac{dx_t}{dt} = a - z \quad .$$

For example, if $z = -1$ and $a = -2$, then at time $t = 0.5$ the value is $x_{0.5} = -1.5$, and the velocity is -1 everywhere along this particular path.

Here t measures progress in generating an action. It is not time in the robot's task. Likewise, this velocity describes how we change a candidate action from a random draw to a sample from our target distribution, which has nothing to do with a robot's velocity.

So, if we can learn this velocity field, we can draw a random normal sample and trace its path to a value at time 1 representing a draw from our target distribution. We would like to learn this field conditioned on the observation o. We can do this with supervised learning with a completely standard neural network regression model!

It goes like this:

One flow-matching training update

1. Draw a demonstrated pair (o, a) from the dataset.
2. Independently draw noise z from $\mathcal{N}(0, I)$ and a time t uniformly between 0 and 1.
3. Form $x_t = (1-t)z + ta$ and the target velocity $a - z$.
4. Update θ to reduce $\|v_\theta(x_t, t, o) - (a - z)\|^2$.

For vector actions, I is the identity matrix: the noise coordinates are independent and each has variance one. The observation o is an input to the network; we do not add noise to it in this procedure.

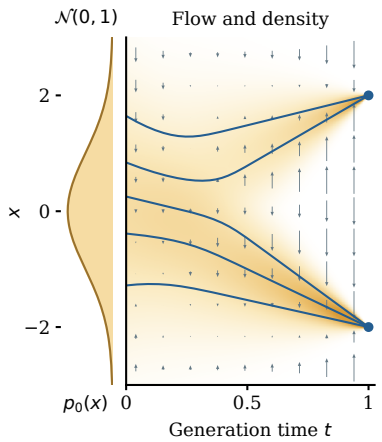


Figure 2: Flow for the two mug actions. Left: the Gaussian density at $t = 0$. Background shading shows the density at each time: darker means higher density, with the darkest color used for all densities above 2. Vertical arrows show velocity in action space. Their lengths are proportional to speed, capped at $|v| = 6$ to keep the figure readable. The blue curves show how samples change as time advances, ending at actions -2 or 2.

We can generate many training examples from the same demonstration by drawing different noise values and times. Each update requires only one point on a path; we don't have to simulate the whole path.

2.2 *Generating an action*

At execution time, we have an observation o and want to generate an action. We start with a fresh draw from the Gaussian distribution and repeatedly ask the network how to change the current candidate. We do so by taking an *Euler step*: we use the velocity at the current point

as an approximation to the velocity throughout a short interval to get a new value $x_{t+\delta t}$.

Sampling an action

1: Given trained neural network v_θ
2: Observe o, draw x_0 from $\mathcal{N}(0, I)$, and choose a number of steps K. Set $\Delta t = 1/K$.
3: For $k = 0, \ldots, K-1$, set $t = k/K$ and compute

$$x_{t+\Delta t} = x_t + \Delta t\, v_\theta(x_t, t, o) \ .$$

4: Return x_1 as the predicted action.

After the initial draw, these updates are deterministic. But different initial draws can produce different actions at the same observation. The network represents a distribution through this sampling procedure; it need not output a list of probabilities or a density formula.

The paths used to construct training examples are straight. The paths followed during sampling generally aren't: the predicted velocity can change as both x_t and t change. More, smaller steps give a better numerical approximation to following the learned velocity field, but take more computation. They do not fix errors in the learned model.

2.3 Why we don't just get the average action

Okay to skip this if you're not interested

We are using squared loss again. Why doesn't this lead to the same problem as directly predicting an action? To see what is being averaged, fix an observation o. Let A be a random demonstrated action at that observation, draw $Z \sim \mathcal{N}(0, 1)$ independently, and define

$$X_t = (1-t)Z + tA \ .$$

By the squared-loss argument in Section 1.2, the ideal network predicts

$$\begin{aligned} v^*(x, t, o) &= \mathbb{E}[A - Z \mid X_t = x, O = o] \\ &= \frac{\mathbb{E}[A \mid X_t = x, O = o] - x}{1-t} \ , \qquad t < 1 \ . \end{aligned} \tag{1}$$

The second line follows by solving $x = (1-t)Z + tA$ for Z and substituting into $A - Z$. The average depends on the current candidate x, as well as on the observation.

Let's use the mug example, measuring the action in centimeters. Suppose $A = -2$ with probability 0.7 and $A = 2$ with probability 0.3. At $t = 0$, $X_0 = Z$ tells us nothing about A, so

$$v^*(x, 0, o) = \mathbb{E}[A \mid O = o] - x = -0.8 - x \ .$$

If we took one Euler step of length 1, every initial sample would be sent to $x + (-0.8 - x) = -0.8$. So just taking one large step with this ideal predictor would indeed give us the average action.

But after a small step, we query the network at a new value and a new time. At $t > 0$, the input X_t contains information about the action. Conditional on $A = a$, it is Gaussian:

$$X_t \mid A = a, O = o \sim \mathcal{N}\big(ta, (1-t)^2\big) \ .$$

For $t < 1$, write the densities for the two possible actions as

$$g_L(x,t) = \mathcal{N}\big(x; -2t, (1-t)^2\big) \ ,$$
$$g_R(x,t) = \mathcal{N}\big(x; 2t, (1-t)^2\big) \ .$$

We are abusing notation here a little bit and now using $\mathcal{N}(x; \mu, \sigma^2)$ to stand for the Gaussian pdf with parameters (μ, σ^2) evaluated at x, rather than the whole Gaussian density as above.

Averaging over the two actions gives

$$p_t(x \mid o) = 0.7 g_L(x,t) + 0.3 g_R(x,t) \ .$$

Now apply Bayes' rule. The probability of the left action given the intermediate value is

$$w_L(x,t) = \Pr(A = -2 \mid X_t = x, O = o) = \frac{0.7 g_L(x,t)}{0.7 g_L(x,t) + 0.3 g_R(x,t)} \ .$$

We multiply each action's prior probability by the likelihood of seeing x if that action had been chosen, then divide by the sum. The right action has probability $1 - w_L(x,t)$. Substituting these probabilities into Equation (1) gives

$$v^*(x,t,o) = w_L(x,t)\frac{-2-x}{1-t} + (1 - w_L(x,t))\frac{2-x}{1-t} \ .$$

For example, at $t = 1/2$, the two Gaussian components have means -1 and 1, and variance $1/4$. At $x = -1$, almost all the posterior probability is on the left action, so the velocity is approximately $(-2-(-1))/(1/2) = -2$. At $x = 1$, almost all of it is on the right action, giving velocity approximately $(2-1)/(1/2) = 2$. The two candidates move in different directions.

As t increases, the components become narrower and the posterior usually puts more weight on one action. With the ideal velocity and exact integration, the distribution of the moving samples matches $p_t(x \mid o)$. In this example, as t approaches 1, 70% of the probability concentrates at -2 and 30% at 2. A fitted network and a finite number of sampling steps only approximate this result.

So we still take averages, but they are averages of velocities conditioned on the current intermediate value. The weights change during sampling. We never have to choose one fixed average action for all the samples.

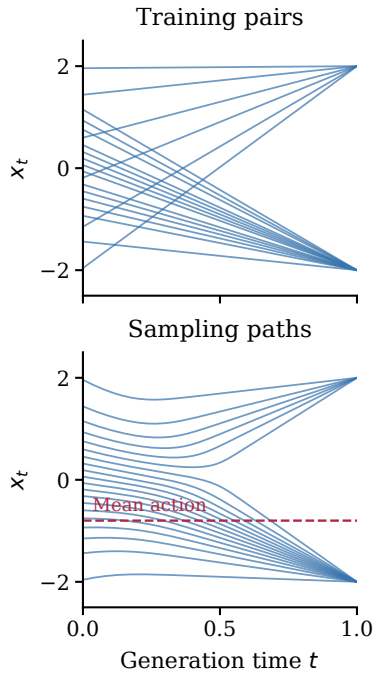


Figure 3: Top: example straight paths used to construct training pairs. Bottom: paths following the ideal conditional average velocity for the same noise distribution. Different starting values lead to different actions. The dashed line marks the mean action, -0.8.

The general result is that the conditional average velocity moves the whole distribution in the desired way. The flow-matching paper in the readings gives a proof. In a two-point example, densities and velocity formulas are used for $t < 1$; the endpoint distribution is their limit.

2.4 *Relation to diffusion*

Diffusion models use a closely related idea: corrupt demonstrated actions with noise, train a network to predict something about that corruption, and use repeated predictions to generate an action from noise. A common diffusion model predicts the added noise. Our flow model predicts the velocity along an interpolation.

For diffusion, let $t = 0, \dots, T$ count the steps of the noising process. Here X_0 is the clean action and X_T is nearly pure noise, so the time direction is opposite to the flow construction above. The forward noising process is

In both constructions, the time index describes generation, not robot time.

$$X_t = \sqrt{\alpha_t}\, X_{t-1} + \sqrt{1-\alpha_t}\, Z_t \quad , \qquad Z_t \sim \mathcal{N}(0,1) \quad .$$

Each Z_t is fresh independent noise, $0 < \alpha_t < 1$, and $\beta_t = 1 - \alpha_t$ is the variance of the noise added at that step. Writing $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, we can also sample directly at a chosen noise level:

$$X_t = \sqrt{\bar{\alpha}_t}\, X_0 + \sqrt{1-\bar{\alpha}_t}\, Z \quad , \qquad Z \sim \mathcal{N}(0,1) \quad . \tag{2}$$

This follows by expanding the recurrence and adding the variances of the independent noise terms. Conditional on $X_0 = x_0$, the mean is $\sqrt{\bar{\alpha}_t} x_0$ and the variance is $1 - \bar{\alpha}_t$. Here Z represents the combined noise, not the noise from one step. To start generation from a standard Gaussian, we choose $\bar{\alpha}_T$ close to zero so that little of the clean action remains.

For training, draw (o, x_0), a noise level t, and noise z. Form x_t using Equation (2) and minimize

$$\left\| \epsilon_\theta(x_t, t, o) - z \right\|^2 \quad .$$

The ideal prediction is $\mathbb{E}[Z \mid X_t = x_t, O = o]$. Solving the forward equation gives the corresponding clean-action estimate:

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\, \epsilon_\theta(x_t, t, o)}{\sqrt{\bar{\alpha}_t}} \quad .$$

With an ideal noise predictor, this is the conditional mean $\mathbb{E}[X_0 \mid X_t = x_t, O = o]$. At nearly pure noise, it can average incompatible actions, just as a single large step did in the flow example. The small denominator also amplifies prediction errors.

A diffusion sampler uses the prediction to take a step toward less noise, then predicts again. A *denoising diffusion probabilistic model* (DDPM) uses the prediction to set the mean of a Gaussian for the next candidate. It adds fresh Gaussian noise at the intermediate reverse steps; the last step returns the final action. So even from the same initial noise value, repeated runs can take different paths and produce different actions.

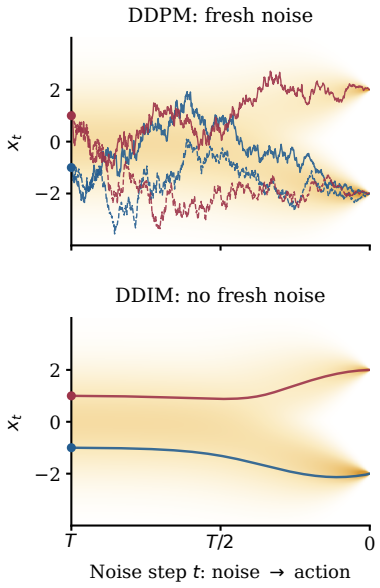


Figure 4: Two strategies for sampling from a diffusion model trained on the same mug distribution: probability 0.7 at -2 and 0.3 at 2. In each plot cases, we drew two random values from a Gaussian for the initial x_T, and for each of those random values, we ran the sampling "denoising" process twice. DDPM's runs differ; deterministic DDIM's coincide. Both use the exact noise predictor and 1000 steps. Shading shows the forward noising densities, with the same color scale as Figure 2. Read left to right, from $t = T$ to 0.

A *denoising diffusion implicit model* (DDIM) can use the same trained noise predictor with a different sampling rule. In its deterministic version, each next candidate is fixed by the current one: no fresh noise is added after the initial draw. Different initial draws can still give different actions. DDIM changes the rest of the update as well; it is not obtained by just deleting the noise term from a DDPM update.

Figure 4 compares the two samplers on our mug example. Here we can calculate the ideal noise prediction exactly, so no neural network is needed for the illustration. The paths show the effect of the sampling rules; with a learned predictor, there would also be prediction errors.

Flow matching is a reasonable default for a robot policy. At prediction time, it often uses many fewer network evaluations than the original DDPM sampler. DDIM can also use few evaluations; with the same network and number of evaluations, its cost is about the same as flow's. The relevant comparison is how many evaluations each method needs to produce good actions. Training is regression in both cases, without running the sampler. Flow matching's interpolation and target $a - z$ make it particularly simple to implement, but do not guarantee faster learning. Flow matching is now common in robot policies. Its simple training rule and effective sampling in a few steps help explain its appeal.

For example, π_0 uses 10 flow steps to generate actions, and GR00T N1 uses 4.

2.5 *Using the models on a robot*

For a robot, the output may be an *action chunk*: a sequence of future commands predicted together. We put all the coordinates of the chunk into one vector and apply the same training and sampling procedures. Although we start with independent noise in each coordinate, the network predicts the whole vector jointly. It can therefore learn relationships among commands, such as continuing along one route around an obstacle.

The robot observes, generates a chunk, executes the first part of it, and observes again. Within one generation procedure, the observation is fixed, so its neural-network features can be computed once and reused. Section 4.2 discusses why chunks are useful and how much of one to execute. *Diffusion Policy* uses this procedure with a diffusion model; a flow-matching policy can use the same observation and action representations.

For interesting studies of the properties of diffusion policies, see *Much Ado About Noising* and *Demystifying Diffusion Policies*.

3 *The DAgger problem*

Even a policy that represents the demonstrated action distribution well can fail when it controls the robot. Its actions determine which states it visits, and those states may differ from the ones in its training data. We now examine this problem and ways to collect the missing examples.

3.1 *The states in the training data*

In ordinary supervised learning, we usually assume that test inputs resemble training inputs. In behavior cloning, the robot's own actions affect what it will observe next. For example, demonstrations of grasping a mug may show the hand approaching along successful paths. If the learned policy moves the hand too far to one side, its next observation may show a configuration rarely seen in training.

Thus, a policy might predict the expert's actions accurately on held-out demonstrations yet perform poorly when controlling the robot. Supervised loss on the demonstrations tells us how well it acts in states visited by the expert; successful execution requires it to act well in the states it actually reaches. Further errors in these unfamiliar states can move the robot still farther from the demonstrated paths (Figure 5).

We call a run of a policy, including the states it visits and the actions it takes, a *rollout*. To see how this can affect a whole task, consider a rollout of T steps. The expert follows a successful path with cost 0 at every step. Until the learner takes a wrong turn, it follows this same path; suppose its probability of a wrong turn at each such step is at most ε. In this worst-case example, a wrong turn is unrecoverable and incurs cost 1 on that step and every remaining step.

There are at most T possible times for the first wrong turn. Adding their probabilities gives an upper bound of $T\varepsilon$ on the probability of any wrong turn during the rollout. Each failed rollout costs at most T, so the expected excess cost over the expert is at most $T \cdot T\varepsilon = T^2\varepsilon$. One factor of T comes from the opportunities to make a mistake; the other comes from how long its consequences can last. The total cost is also at most T, so the quadratic bound is useful only when $T\varepsilon < 1$.

This is a worst-case argument, not a prediction that every small error will grow. A stable controller may correct a disturbance quickly. Demonstrations that include recovery behavior can teach a learner to do the same. The difficulty is that expert trajectories may rarely contain the recovery states a learner needs. This suggests collecting expert action labels in states reached by the learner, including states where it must recover from its own mistakes. That is the idea behind DAgger, which we describe next.

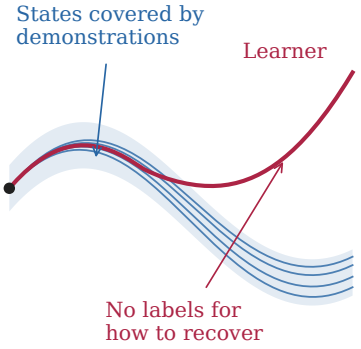


Figure 5: A learner can leave the region covered by demonstrations. Once there, another small prediction error may move it farther away.

3.2 *DAgger: collecting the missing labels*

DAgger, short for *Dataset Aggregation*, lets the learner influence which states receive expert labels. An expert must be available to answer what action should be taken at a state reached during training. That label can differ from the action actually executed there.

DAgger procedure

1. Collect demonstrations D_0 and fit an initial policy π_0.
2. Roll out the current policy and record the states before its actions.
3. Ask the expert for an action label at each recorded state.
4. Append these labeled examples to the existing dataset, retaining repeats.
5. Fit a new policy on the aggregate dataset and repeat from step 2.

Compare candidate policies using validation rollouts. The last fitted policy is not automatically the best one.

Here is a small example problem with states A and D and actions *align* and *insert*. In state A, a hand is aligned with a slot; in state D, it is displaced from the slot. Action *insert* succeeds from state A but leaves the hand stuck in state D if attempted there. Action *align* moves the hand from state D to state A. The expert labels are *insert* in state A and *align* in state D.

The initial dataset contains four copies of $(A, \textit{insert})$. A lookup policy fits those examples and defaults to *insert* at an unseen state. Now a rollout begins in state D, owing to an initial-position perturbation. The policy executes *insert* twice, remaining in state D both times. The expert labels both visits with *align*, so the new labeled examples are

$$(D, \textit{align}) \ , \quad (D, \textit{align}) \ .$$

After aggregation there are six examples: four from state A and two from state D. The old policy makes two mistakes on this dataset. A policy that chooses *insert* in state A and *align* in state D makes none. Its next rollout from state D executes *align* and then *insert*.

DAgger may also mix expert and learner actions during data collection. At iteration i, execute the expert's action with probability β_i and the learner's action otherwise, usually reducing β_i over time. The case $\beta_i = 0$ is a pure learner rollout. Mixing can keep early rollouts in useful regions while the learner is still poor. The training labels remain expert labels regardless of which policy supplied the executed action.

DAgger gives the learner examples of what to do in situations caused by its own actions, including its mistakes. If it can learn the expert's corrections and get back on track quickly, one mistake need not spoil the rest of the task. This can limit the accumulation of errors described in Section 3.1. Success still depends on the learner's ability to learn from the new labels and on whether the task allows recovery; more labels cannot undo an irreversible mistake. Ross, Gordon, and Bagnell analyze the conditions under which DAgger improves performance in the original DAgger paper; see that paper for the assumptions and mathematical details.

3.3 *Active querying: deciding when to ask for help*

A problem with the basic DAgger procedure is that it does not know how to ask selectively for new labels just when it needs them. It requests an expert label for every visited state, even when the learner already predicts the action well. Since expert labels cost time, we would like to identify situations where asking is likely to help. Two useful signals are disagreement among learned models and distance from the training observations.

Model disagreement and action diversity We could train a set of K models, for example with different initializations or resampled training datasets. Such a collection is called an *ensemble*. Let p_k be member k's predicted probability of taking a particular action at an observation. Define

$$\bar{p} = \frac{1}{K}\sum_{k=1}^{K} p_k \ , \qquad U = \frac{1}{K}\sum_{k=1}^{K} (p_k - \bar{p})^2 \ .$$

The idea is that, for parts of the space where we have enough data to really determine the output of the model, all the models should roughly agree. So, if they disagree, it's a sign that the predictions are not reliable.

If two members both predict a left-route probability of 0.7, then $\bar{p} = 0.7$ and $U = 0$. They agree on a distribution that permits both routes. If instead they predict 0.4 and 1.0, the mean is still 0.7 but $U = (0.3^2 + 0.3^2)/2 = 0.09$. The distribution averaged across the models is the same, but their disagreement differs.

The first case can reflect variation in valid demonstrated behavior, sometimes called *aleatoric uncertainty*. The second suggests uncertainty about the model, called *epistemic uncertainty*, which more informative data may reduce. Even if all the models agree on a prediction it might not be right; for example, a shared training bias can make every ensemble member wrong in the same way.

Distance from known observations Let $\phi(o)$ be a feature vector representing observation o. For example, it could be an image encoder's output together with joint positions. Define the nearest-training-observation distance

$$d(o) = \min_{i \in D} \|\phi(o) - \phi(o_i)\| \ .$$

A large distance indicates that the feature vector is unlike those seen in training. One query rule is

$$\text{ask if } d(o) > \tau_d \ ,$$

where τ_d is a threshold that can be picked via cross validation.

Labels obtained at these selected states can be added to the training dataset before refitting the policy, just as in DAgger.

4 Representations for actions

So far, we have treated an action as a quantity or vector to predict. We must also decide exactly what quantities to predict and how to encode them, taking into account both the process of model learning and of controlling the robot. There are many choices to be made, but we will explore three of them: whether to command positions or velocities, whether to predict one command or a sequence, and whether to express pose targets in world coordinates or relative to the current hand pose.

4.1 Position targets and velocity commands

A policy can predict either a desired position or a commanded velocity. There are two difficulties with using velocities as the output representation.

First, in teleoperated data, it can be hard to estimate a velocity to use as a training label. If we record hand positions, we must estimate velocity from their change over time. Small errors in the measured positions can become large errors in the estimated velocity.

Second, at execution time, errors in velocity commands can accumulate into increasingly large position errors before feedback corrects them. A position command instead gives the robot a target: a feedback controller that tracks this target tends to damp deviations by moving the robot back toward it, even between policy predictions. A velocity policy can also learn to correct errors from its next observation. The distinction is where and how often that correction happens. We examine these two effects below.

Noise in differentiated measurements Suppose measured hand positions are $\widetilde{X}_t = x_t + E_t$, where E_t has mean zero and variance σ^2. Estimate

velocity by

$$\widetilde{V}_t = \frac{\widetilde{X}_{t+1} - \widetilde{X}_t}{\Delta t} = \frac{x_{t+1} - x_t}{\Delta t} + \frac{E_{t+1} - E_t}{\Delta t} \quad .$$

If successive measurement errors are independent, the variance of the velocity-label error is

$$\mathrm{Var}\left(\frac{E_{t+1} - E_t}{\Delta t}\right) = \frac{2\sigma^2}{\Delta t^2} \quad .$$

With $\sigma = 0.002$ m and $\Delta t = 0.01$ s, its standard deviation is $\sqrt{2}(0.002)/0.01 \simeq 0.283$ m/s, even for a stationary hand. Adjacent velocity estimates share a position measurement, so their errors are correlated with each other even though the position errors were independent.

Position labels avoid this numerical differentiation. This argument does not apply unchanged if velocity commands are recorded directly from a teleoperator, or if velocity is estimated with a filter and a motion model. The measurement process determines which noise calculation is appropriate.

Bias and feedback Assume the desired position is zero. If an open-loop velocity prediction has a constant error b_v, the idealized discrete dynamics are

$$x_{t+1} = x_t + \Delta t\, b_v \quad \Longrightarrow \quad x_n = x_0 + n\Delta t\, b_v \quad .$$

The position error grows with time. There is no correction in this law. For instance, $b_v = 0.005$ m/s accumulates 0.025 m in five seconds.

A position policy might instead output a biased but fixed target b_x. Suppose a low-level feedback controller, or *servo*, follows it using

$$x_{t+1} = x_t + \alpha(b_x - x_t) \quad .$$

Define error relative to the commanded target as $e_t = x_t - b_x$. Then

$$e_{t+1} = (1-\alpha)e_t \quad , \qquad e_n = (1-\alpha)^n e_0 \quad .$$

The error decays precisely when $|1-\alpha| < 1$, or $0 < \alpha < 2$. The hand converges to b_x. Its error relative to the desired position zero is then b_x, which is bounded in this model.

The velocity example above omits feedback. A learned velocity policy might observe the displacement caused by its previous commands and predict a velocity that moves the hand back. With such feedback, velocity commands need not cause unbounded drift. The practical advantage of a position target is that a fast, low-level controller can correct position errors while the policy is still processing an image or

executing a chunk. If position-error correction relies on the next policy prediction, observation and computation delays postpone it. A velocity controller that only tracks the commanded velocity does not correct an accumulated position error; it needs position feedback to do that. So the comparison depends on the feedback available and how quickly it is used.

We will study the details of control in a few weeks.

4.2 *Action chunking*

An *action chunk* is a sequence of H future commands predicted jointly from the current observation. The robot executes the first K commands, where $1 \leqslant K \leqslant H$, then observes again and predicts a new chunk. This is receding-horizon execution. It separates how far ahead we predict from how long we act before using new feedback.

The Diffusion Policy paper gives three reasons for this design.[1]

[1] Chi et al., *Diffusion Policy*, Sections 4.3 and 5.3.

- **Consistent actions.** Near the start of a motion around an obstacle, both routes can remain plausible after one step. Predicting each next action separately can repeatedly switch the chosen route. A joint sequence represents one coherent continuation.
- **Moving through pauses.** Demonstrations often include pauses. A single-step policy can repeatedly predict an idle action; a sequence can include both the pause and the motion that follows it.
- **Tolerating inference delay.** Image processing and policy inference take time. Predicting future commands provides actions for the controller between policy evaluations and can accommodate that delay.

A larger K gives more time for inference and preserves the chosen sequence longer, but delays the next response to an unexpected change. Chunking trades temporal consistency and tolerance of computation delay against responsiveness; it does not make feedback unnecessary.

4.3 *Absolute and relative hand poses*

A hand *pose* describes where the hand is and how it is oriented. Suppose our policy predicts an action chunk consisting of H future hand poses. We use Drake's monogram notation, as in the problem sets: ${}^{W}X^{G}$ is the rigid transform from world frame W to gripper frame G, or the pose of G in W. It combines a translation and a rotation. When applied to point coordinates, it converts coordinates in G to coordinates in W. Let G_t be the measured gripper frame when we predict a chunk, and let G_{t+j} denote its target frame j steps into the future. The target frame is where we want the hand to be; the actual hand may not reach it exactly.

Here X and R are the conventional symbols for transforms and rotations, not random variables. See the problem sets' transform-notation appendix or Drake's pose and transform notation.

An absolute representation specifies the targets in world coordinates: ${}^{W}X^{G_{t+1}}, \ldots, {}^{W}X^{G_{t+H}}$. An alternative, *relative*, strategy is to express every target relative to the current gripper frame G_t:

$$ {}^{G_t}X^{G_{t+j}} = \left({}^{W}X^{G_t}\right)^{-1} {}^{W}X^{G_{t+j}} \quad , \qquad j = 1, \ldots, H \quad . $$

This describes a translation along the hand's current axes and a rotation relative to its current orientation. The same local motion can then be represented by the same chunk from different starting hand poses.

During training, we compute the relative transforms from recorded hand poses. At execution time, the policy predicts the sequence

$$ \left({}^{G_t}X^{G_{t+1}}, \ldots, {}^{G_t}X^{G_{t+H}}\right) \quad . $$

We reconstruct each world target using the measured ${}^{W}X^{G_t}$:

$$ {}^{W}X^{G_{t+j}} = {}^{W}X^{G_t}\, {}^{G_t}X^{G_{t+j}} \quad . $$

The adjacent G_t labels match, just as in the transform compositions in the problem sets.

The reference frame G_t stays fixed while this chunk is executed. Each target is relative to that one frame, even after the hand has moved. For a numerical example, keep the gripper axes aligned with the world axes and let the hand move along one axis. Write ${}^{W}p^{G_t}$ for its position along that axis. Suppose ${}^{W}p^{G_t} = 0.30$ m and the desired future positions are $0.40, 0.50, 0.60$ m. With these fixed, aligned orientations, the translation parts of the relative transforms are just differences:

$$ \left({}^{G_t}p^{G_{t+1}}, {}^{G_t}p^{G_{t+2}}, {}^{G_t}p^{G_{t+3}}\right) = (0.10, 0.20, 0.30)\ \text{m} \quad . $$

To recover each world target, add the same initial position:

$$ {}^{W}p^{G_{t+j}} = {}^{W}p^{G_t} + {}^{G_t}p^{G_{t+j}} \quad . $$

This gives $0.40, 0.50, 0.60$ m. Adding each offset to the preceding target would instead give $0.40, 0.60, 0.90$ m and execute the wrong motion. The training labels and execution code must agree that the predictions are ${}^{G_t}X^{G_{t+j}}$, all relative to the same frame, rather than transforms between successive frames ${}^{G_{t+j-1}}X^{G_{t+j}}$. For full poses, we use transform multiplication rather than adding pose coordinates.

As in Section 4.2, we may execute only the first K targets before observing again and predicting a new chunk. At that point, we measure the actual gripper frame G_{t+K} and express the next chunk relative to it. This uses the pose the robot reached, which may differ from the preceding chunk's target. Keeping one reference frame within a chunk gives its pose sequence a consistent interpretation; measuring a new reference frame between chunks lets the next prediction respond to what the robot actually did.

Encoding 3D rotations. Euler angles can jump at angle wraparounds; quaternions use q and $-q$ for the same rotation. A useful alternative represents a rotation by the first two columns of its matrix: six numbers. Train two predicted 3-vectors u, v to match these columns. To obtain a valid rotation, normalize the first vector, remove its component from the second, and normalize the result:

$$ r_1 = \frac{u}{\|u\|} \quad , \qquad w = v - (r_1^{\mathsf{T}} v) r_1 \quad , $$

$$ r_2 = \frac{w}{\|w\|} \quad , \qquad r_3 = r_1 \times r_2 \quad . $$

The columns of the rotation matrix are r_1, r_2, r_3. This encoding changes continuously as the desired rotation changes, avoiding jumps in the labels. Normalization needs care if u is near zero or u, v are nearly parallel. See Zhou et al., *On the Continuity of Rotation Representations in Neural Networks*.

5 Data and evaluation

The training data determine which behaviors a policy can learn, and execution tests tell us whether those behaviors solve the task. We consider how to collect and weight demonstrations, then how to evaluate the resulting policy.

5.1 Data collection and quality

Teleoperation on the target robot provides paired observations and commands directly. Other sources require more processing. Human video shows useful motions but does not directly specify robot commands. Handheld demonstrations require pose estimation and a consistent mapping to robot actions. Simulation can supply many labeled examples, but their value depends on whether the simulated observations and contacts resemble deployment.

For every source, record the meaning and timing of an action, its units and coordinate frame, camera and robot identities, and the intended task. Include variations in initial conditions and recoveries where possible. An action recorded before an image but paired as if it followed the image teaches a different mapping. Relative coordinates cannot repair an incorrect time alignment.

Behavior cloning learns to reproduce the actions in its training data. Those actions may come from demonstrators with different levels of expertise. An experienced operator may reliably center a gripper over an object, while a beginner repeatedly approaches too far to one side. If beginners supply most of the data, treating every example equally can make their errors dominate what the policy learns.

Expertise is only one reason to weight examples differently. Here are two other cases where some data may be more useful for the robot we want to deploy:

- **What the demonstrator can see.** Purely observational data, such as video of a person doing a task, usually give us less useful supervision than teleoperation data with recorded robot commands. In *third-person* teleoperation, the demonstrator watches the robot with their own eyes and may use information that the robot's cameras do not capture. In *first-person* teleoperation through VR, the demonstrator sees what the robot will see. Other things being equal, this is more useful for learning an observation-to-action mapping: the action choices are based on the information the policy will have. We might therefore give observational data less weight than third-person teleoperation, and third-person data less weight than first-person data.

- **Which robot collected the data.** Demonstrations from the same robot type as the deployment robot are usually more directly useful than demonstrations from a different robot. Differences in grippers, kinematics, and cameras can change which actions are possible and what the robot observes. We may downweight data from other robot types even after mapping their actions into a common representation.

We can give more influence to demonstrations we have reason to trust. Let $\ell_i(\theta)$ be the prediction loss on example i for policy parameters θ, and assign each example a nonnegative weight w_i. The weighted training loss is

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} w_i \ell_i(\theta) \ .$$

At least one weight must be positive. Setting every weight to 1 recovers the usual average loss. A weight of zero excludes an example, and a larger weight makes fitting that example count more in training. We can weight whole demonstrations or individual observation–action pairs.

One way to choose weights is to specify how much influence each group of demonstrators should have. Suppose 20% of our examples come from experienced operators and 80% from beginners. We decide that experienced operators should account for 80% of the total example weight and beginners 20%. Let $q(c)$ be the fraction of available examples from group c, and $p(c)$ the desired fraction. The *importance weight*

$$w(c) = \frac{p(c)}{q(c)}$$

converts the available mix into the desired mix. Here,

$$w(\text{experienced}) = \frac{0.8}{0.2} = 4 \ , \qquad w(\text{beginner}) = \frac{0.2}{0.8} = 0.25 \ .$$

Each experienced example therefore receives sixteen times the weight of each beginner example. Choosing the desired mix is a judgment about data quality; calculating the ratio implements that choice.

To see the effect on a prediction, consider ten examples with the same observation. The two experienced examples recommend a scalar position command $a = 1$, while the eight beginner examples recommend $a = -1$. Assume that the experienced command is correct and the beginner command is a systematic error. With squared loss, the unweighted prediction is $[2(1) + 8(-1)]/10 = -0.6$. The weighted prediction is

$$\hat{a} = \frac{2(4)(1) + 8(0.25)(-1)}{2(4) + 8(0.25)} = 0.6 \ .$$

Weighting moves the prediction toward the experienced command, although it does not remove the beginners' influence entirely.

More generally, let $\ell(c)$ be a fixed policy's average loss within group c. Then

$$\sum_c q(c) w(c) \ell(c) = \sum_c p(c) \ell(c) \ .$$

Thus, with exact weights, averaging the weighted loss over repeated datasets sampled from q gives the loss for the desired mix p. This is an *unbiased estimate*. It assumes that we change only the group proportions, keeping the distribution of examples within each group the same. The identity also applies to other ways of grouping examples, such as by state. The resulting loss measures agreement with the chosen mixture of demonstrations; task success still needs to be tested by running the policy.

5.2 Evaluating a policy

Evaluating robot policies is very important and very hard to do well! To get comparable experimental conditions, we need a way of resetting the environment, and everything takes a lot of time and usually human supervision. We also need to be careful about whether we have enough data to support conclusions we might want to make about, for example, whether one method is better than another.

Choose the distribution of initial configurations, the definition of success, and the time limit before testing. Use the same conditions for both methods and randomize their testing order. Keep the final test trials separate from the trials used to choose models and tune parameters. Be careful that you don't make changes in lighting or camera calibration between trials for one method and trials for the other. For now, assume trials are independent; we will discuss paired trials below.

Success and failure Let's compare methods A and B, judging on each trial whether the robot successfully completed the task. Suppose we run N trials of each and observe s_A successes for A and s_B for B. Let p_A and p_B be their underlying success probabilities under our testing conditions. To ask whether A is better, we test the *null hypothesis* $H_0 : p_A \leqslant p_B$ against $H_1 : p_A > p_B$.

A classical choice, especially with few trials, is *Fisher's exact test*. At the boundary of the null hypothesis, $p_A = p_B$. Conditional on the total number of successes $s = s_A + s_B$, every assignment of those successes to the $2N$ trials is equally likely. We calculate the probability that A would get at least as many successes as we observed:

$$p_{\text{value}} = \sum_{k=s_A}^{\min(N,s)} \frac{\binom{N}{k}\binom{N}{s-k}}{\binom{2N}{s}} \ .$$

The binomial coefficients count the ways to assign k successes to A's trials and the remaining successes to B's trials. This is a *one-sided* test: it looks for evidence in the direction specified before testing. If either direction would be interesting, use a *two-sided* test instead.

For a 2×2 table of success and failure counts, software can do this calculation directly; see SciPy's `fisher_exact`. Choose the direction of the test before looking at which method did better.

A small p-value means that a result at least this favorable to A would be unusual under the null hypothesis. It is not the probability that the null hypothesis is true. We might choose a significance level $\alpha = 0.05$ and reject H_0 when $p_{\text{value}} \leqslant \alpha$. Under the test's assumptions, this procedure falsely declares A better in at most 5% of repeated experiments where H_0 is true.

So, if A succeeds $7/10$ times and B succeeds $9/10$ times, what can we conclude? The observed rates favor B, by 20 percentage points, but the two-sided Fisher p-value is about 0.58. Even if we had specified in advance that we would test whether B is better, the one-sided p-value would be about 0.29. Ten trials per method do not establish that either is better at the 0.05 level. This also does not establish that they are equally good: the experiment leaves substantial uncertainty.

For larger samples, a *normal approximation* gives a simpler calculation. Let $\hat{p}_A = s_A/N$ and $\hat{p}_B = s_B/N$ be the observed success rates. Under the hypothesis $p_A = p_B$, we pool the trials to estimate their common success probability as

$$\hat{p} = \frac{s_A + s_B}{2N} \quad .$$

A success rate estimated from N independent binary trials has variance $p(1-p)/N$. The difference of two independent rates has twice that variance. Substituting our pooled estimate gives the test statistic

$$z_{\text{obs}} = \frac{\hat{p}_A - \hat{p}_B}{\sqrt{2\hat{p}(1-\hat{p})/N}} \quad .$$

This measures the difference in success rates relative to the variation we would expect from random trials if the methods were equally good. Under that hypothesis, the standardized difference is approximately standard normal. Let $\Phi(z) = \Pr(Z \leqslant z)$ for $Z \sim \mathcal{N}(0,1)$. Then the approximate p-values are

$$\begin{aligned} p_{\text{one-sided}} &\approx 1 - \Phi(z_{\text{obs}}) && \text{for } p_A > p_B \quad , \\ p_{\text{two-sided}} &\approx 2\big[1 - \Phi(|z_{\text{obs}}|)\big] && \text{for } p_A \neq p_B \quad . \end{aligned}$$

This is called the *two-proportion z-test*. For a two-sided test at level 0.05, we reject when $|z_{\text{obs}}| > 1.96$.

See NIST's normal approximation for comparing proportions. Fisher's test fixes the total number of successes; the normal test above is a large-sample approximation to the same comparison, not the exact calculation.

The approximation needs enough successes *and* failures. A common minimum check is that the expected counts in each group, $N\hat{p}$ and $N(1-\hat{p})$, are both at least 5. For the $7/10$ versus $9/10$ example, $\hat{p} = 0.8$, $z_{\text{obs}} \approx -1.12$, and the approximate two-sided p-value is 0.26. The expected failure count is only 2 per group, so this approximation is poor; use the exact Fisher result, 0.58, for this small experiment.

Task progress Now suppose each trial receives a score between 0 and 1, according to how far it progressed toward success. A human or a vision-language model (VLM) might assign the score using a fixed rubric. Use the same rubric and judge for both methods, ideally without revealing which method produced the trial; statistical testing cannot correct a judge's systematic preference for one method.

Let the observed scores be $r_{A,1}, \ldots, r_{A,N}$ and $r_{B,1}, \ldots, r_{B,N}$. To compare *mean progress*, a standard choice is *Welch's two-sample* t*-test*. Compute the sample means $\bar{r}_A, \bar{r}_B$ and sample variances $\hat{\sigma}_A^2, \hat{\sigma}_B^2$ (using $N - 1$ in the variance denominator). The test statistic is

$$t_{obs} = \frac{\bar{r}_A - \bar{r}_B}{\sqrt{\hat{\sigma}_A^2/N + \hat{\sigma}_B^2/N}} \ .$$

The denominator estimates the uncertainty in the difference of means. Thus, the statistic measures the observed difference relative to the variation between trials. Software compares it with an appropriate Student t distribution to obtain a p-value, using a one-sided test for $\mu_A > \mu_B$ or a two-sided test for $\mu_A \neq \mu_B$, where the μ's are the true mean scores.

Welch's test allows the two methods to have different score variances. In SciPy's `ttest_ind`, set `equal_var=False`. The NIST handbook gives the reference distribution and its degrees of freedom.

Scores restricted to $[0, 1]$ are clearly not normally distributed: a normal distribution with nonzero variance assigns some probability to values outside that interval. Clipping values to the interval can also create concentrations exactly at 0 and 1. Nevertheless, a test based on a normality assumption can sometimes work well. With enough independent trials, the *sample means* can be approximately normal even when the individual scores are not. This is a consequence of the central limit theorem, and helps explain why a t-test can still be useful. With few trials, especially when most scores are at an endpoint, the approximation can be poor.

Another approach is to use a *non-parametric* test, which does not require choosing a distribution family such as the Gaussian for the individual scores. There are many such tests, including tests based on the ranks of the observations. They still have assumptions, and they do not all test the same hypothesis. Here we can use a *permutation test*: pool the scores, repeatedly shuffle the A/B labels into two groups of size N, and compare the observed difference of means with the differences after shuffling. This tests a null hypothesis under which the labels are interchangeable: the two methods have the same score distribution. It does not require normally distributed scores.

See SciPy's `permutation_test`. For a two-sided comparison, count shuffled absolute differences at least as large as the observed one. With M random shuffles and b such results, use $(b + 1)/(M + 1)$.

Looking at whole distributions Two methods can have the same mean but behave quite differently. A method that always completes half the task and one that succeeds fully on half the trials and does nothing on the rest both have mean progress 0.5. A *violin plot* helps show this

distinction. Its vertical axis gives the score, and its width shows a smoothed estimate of how often scores near that value occur.

Figure 6 shows an example from the TRI large behavior models paper. The gray single-task baseline often makes relatively little progress; the red and yellow models concentrate more trials near completion but still have lower tails. These violins show variation between trials, not uncertainty about the mean. The shape of a violin also depends on the smoothing used; its overlap with another violin is not a statistical test.

To test whether the *whole score distributions* differ, we can use a statistic such as the largest gap between their empirical cumulative distributions: at each score, compare the fractions of trials scoring at or below it, and take the largest absolute difference. This is the Kolmogorov–Smirnov statistic. Calibrating it by shuffling method labels, as above, gives a permutation test of equal distributions, including when scores have ties. A difference detected this way could concern spread or failure frequency, rather than a higher mean.

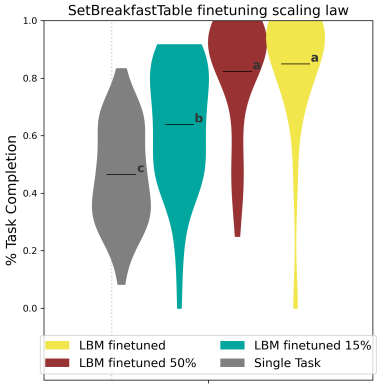


Figure 6: Task-completion distributions for setting a breakfast table, reproduced from Figure 5 of the TRI LBM Team's *A Careful Examination of Large Behavior Models for Multitask Dexterous Manipulation*. Each policy was evaluated on 50 real-world trials. Percentages refer to the task-specific training data used for finetuning. Horizontal marks are means; letters summarize the paper's statistical comparisons of means. A shared letter indicates no detected difference.

Paired trials If we generate a random initial configuration and run both A and B from that same configuration, the trials are *paired*. Reset between the runs and randomize which method goes first. For progress scores, compute the difference within each pair,

$$d_i = r_{A,i} - r_{B,i} \ .$$

A *paired* t*-test* uses

$$t_{obs} = \frac{\bar{d}}{s_d/\sqrt{N}} \ ,$$

where $\bar{d}$ is the mean difference and s_d is its sample standard deviation. The reference t distribution has $N-1$ degrees of freedom. Here the pairs should be independent; with few pairs, the differences should be approximately normally distributed.

See the NIST discussion of paired observations. For paired binary outcomes, use a paired test such as McNemar's exact test, based on pairs where one method succeeds and the other fails. Independent-sample Fisher testing would discard the pairing.

When difficult configurations tend to be difficult for both methods, taking differences removes much of that shared variation. We can then often detect a difference with fewer total trials. Pairing helps to the extent that it makes the differences less variable; it is not an automatic reduction in the required sample size.

There are lots of subtleties to be aware of, and it is worth asking a statistician when they arise. For example, decide in advance how to handle crashes and time-outs. A policy-caused crash or failure to finish within the task's time limit is usually a failed trial; silently dropping it would favor the unreliable method. An unrelated equipment failure may need different treatment. Testing many methods or repeatedly checking whether we can stop after a significant result also requires an analysis that accounts for those choices.

Statistical significance and *practical significance* are different. A small p-value is evidence against the null hypothesis under the test's assumptions; it does not say that the improvement is large enough to matter for our application. Report the size of the effect and an uncertainty interval as well as the p-value. A tiny improvement can be statistically significant with enough trials, while a useful improvement may remain uncertain with too few.

Finally, inspect failures. A missed grasp due to a shifted camera, a wrong coordinate convention, and an unseen recovery state suggest very different changes. An overall success percentage alone cannot tell us which data or model change will help.

Reading and PS5 guide

The table points to background for each bookwork problem. The worked examples use different numbers so that the problems still require applying the ideas.

PS5	Topic	Notes
1	Discrete and continuous action representations	§1, §2.4
2	Observation history, pauses, and action chunks	§1, §4.2
3	Regression, distributions, and coherent action chunks	§1, §4.2
4	Learner rollouts, expert labels, and aggregation	§3.2
5	Noise in velocities and position feedback	§4.1
6	Hand-relative pose targets, observations, and output ranges	§4.3
7	Sharing robot data and importance weights	§5.1
8	Ensemble disagreement and distance from training data	§3.3
9	Flow matching and conditional velocities	§2.1–§2.3
10	Binary outcomes, sample size, and statistical significance	§5.2
11	Rotation representations and orthonormalization	§4.3

- Lipman et al. (2023), *Flow Matching for Generative Modeling*. Training velocity fields and the conditional averaging result.
- Ross, Gordon, and Bagnell (2011), *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*. The DAgger algorithm and its analysis.
- Chi et al., *Diffusion Policy: Visuomotor Policy Learning via Action Diffusion*. Action diffusion and robot execution.
- Pan et al., *Much Ado About Noising: Dispelling the Myths of Generative Robotic Control*. Ablations of policy representation and training.
- He et al., *Demystifying Diffusion Policies: Action Memorization and Simple Lookup Table Alternatives*. Memorization and retrieval in small-data policy learning.
- Ho, Jain, and Abbeel (2020), *Denoising Diffusion Probabilistic Models*; Song, Meng, and Ermon, *Denoising Diffusion Implicit Models*. Further derivations of the sampling procedures.
- Song et al. (2021), *Score-Based Generative Modeling through Stochastic Differential Equations*. The deterministic-flow connection to diffusion.