16-264 Notes on Learning Transformations

What are transformations?

Agents often have to convert points (vectors) in one space to another space. The example we are using in class is that eyes see things in visual space, and an arm needs commanded joint angles (or some other arm-related variables). To be specific, we will say that the visual space is 2D coordinates (x, y). We will initially work with a planar arm with two joints (shoulder θ_1 and elbow θ_2): Reaching out

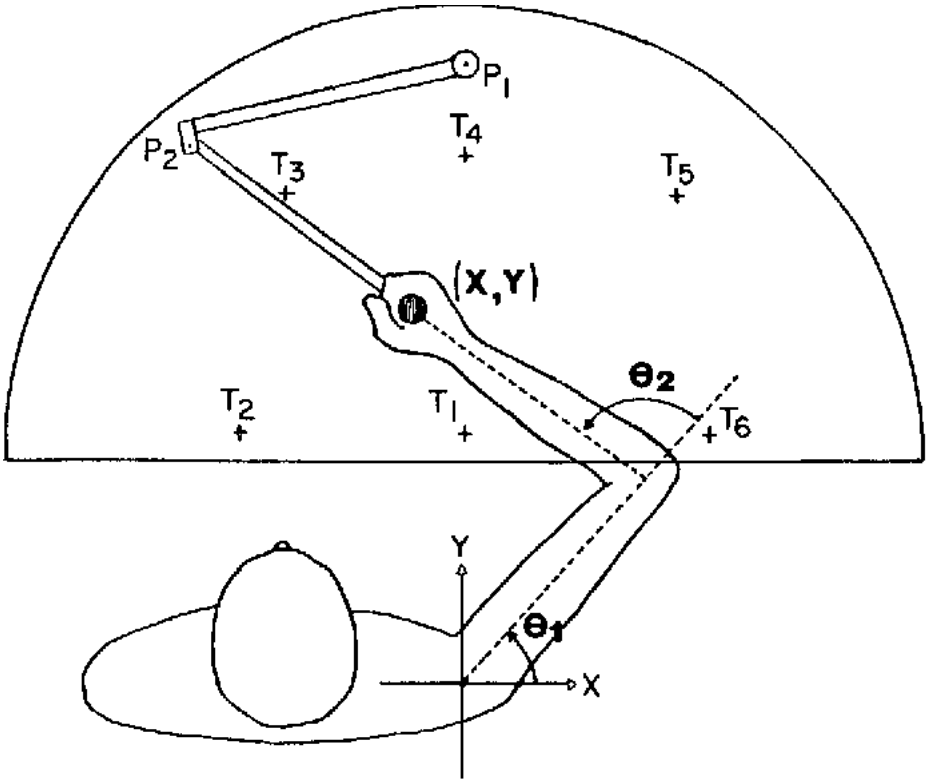


Figure 2. Experimental apparatus for measuring arm trajectories in a

to touch an (x, y) target involves generating the desired joint angles (θ_1, θ_2). This transformation is known as "inverse kinematics". You use the reverse transformation, "forward kinematics", to know where your hand is in space based on your sensed joint angles when you can't see your hand.

Real life kinematics is more complicated. Humanoids might have 50 joints and have many ways to touch the same target (redundancy). We are going to ignore that for now.

Other transformations a humanoid might learn are transformations between tasks or problems to solve and commands or solutions. The key issue for supervised learning is "Do you have a lot of examples of inputs and outputs (training data)?"

Learning transformations = function approximation

Learning transformations can be viewed as function approximation:

$$\text{outputs} = f(\text{inputs}, \text{parameters}) \tag{1}$$

where the parameters are adjusted until the function $f()$ correctly maps the inputs to the outputs.

What does an agent learn from?

"Self-supervised" learning is when an agent can create its own training data that includes both possible inputs and desired outputs. This is called "labelled" training data, as the desired outputs are known. For learning kinematics, labelled training data is available continuously. Whenever a humanoid can see their hand, a combination of joint angles and visual location of the wrist or hand is available:

$$(x, y, \theta_1, \theta_2) \tag{2}$$

To learn inverse kinematics, x and y are inputs, and θ_1 and θ_2 are the outputs. To learn forward kinematics, θ_1 and θ_2 are the inputs, x and y are the outputs.

What are possible representations?

There are popular choices for the type of functions (representations) used in learning transformations.

One class is domain-specific parameteric functions based on engineering or other forms of knowledge, where the structure of the function is motivated by knowledge about a particular domain. An example is the following parametric function for forward kinematics you might see in a robot textbook:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) \\ l_1 \sin(\theta_2) + l_2 \sin(\theta_1 + \theta_2) \end{pmatrix} \tag{3}$$

θ_1 and θ_2 are the inputs, l_1 and l_2 are the parameters, and x and y are the outputs. In a real implementation, one might add additional parameters such as bias and scale parameters that affect each joint angle.

Another class is domain-independent parametric functions. "Neural networks" are in this class, as are some kinds of tables, linear, and polynomial models. More on neural networks later.

Another class is memory-based models, where the training data is stored, and an output vector is constructed at query time by interpolating training data similar or "near" to a query. Google "locally weighted learning"

What about search, or trial and error?

It is true that an agent can try sets of joint angles out on a real robot arm, and generate new angles and try them until the target is touched. Given we can turn off the lights so visual feedback cannot be used in actual search, and still do pretty well, suggests humans have some learned kinematics function.

It is also true that an agent can try sets of joint angles out in a simulation with a good forward kinematics model, and generate new angles and try them until the target is touched. This can be viewed as internal search or optimization to generate a solution. In this case the forward kinematics model for the simulation must be represented and learned.

Curve fitting using functions that are linear in the adjustable parameters (the batch case)

A special case is fitting a data set with a function that is linear in the unknown parameters. Let's say that x is an input and that y is the output. $\mathbf{p}$ is a vector of parameters to be learned.

$$y = f(x, \mathbf{p}) \tag{4}$$

Examples include polynomials such as a line:

$$y = c_1 * x + c_0 \tag{5}$$

or a parabola (quadratic function):

$$y = c_2 * x^2 + c_1 * x + c_0 \tag{6}$$

Note that the equation above is nonlinear in x (x is squared, which is not a line but a parabola), but it is linear in the unknown coefficients c_i.

We can write this in the following vector equation:

$$\begin{pmatrix} 1 & x & x^2 & \cdots & x^n \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = y \tag{7}$$

This allows us to add more training data points, indexed by i:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ 1 & x_3 & x_3^2 & \cdots & x_3^n \\ & & \vdots & & \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} \tag{8}$$

A compact version of this equation is

$$\mathbf{Ap} = \mathbf{b} \tag{9}$$

This is a set of equations for $n+1$ unknowns. If there are $n+1$ equations, we can solve it using a matrix inverse or your favorite equation solver: $\mathbf{p} = \mathbf{A}^{-1}\mathbf{b}$

However, the data is typically noisy and we want to use more than $n+1$ data points. In this case, what we do is multiply both sides by $\mathbf{A}^{\mathrm{T}}$:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{p} = \mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{10}$$

resulting in:

$$\mathbf{p} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b} \tag{11}$$

In this case learning involves a cycle of collecting more data, adding it to the training set, solving the above equations, and doing it again.

This is minimizing the sum of squared prediction errors. Physical analogy: springs from data points vertically to a line or plane, minimize potential energy.

Iterative curve fitting using gradients

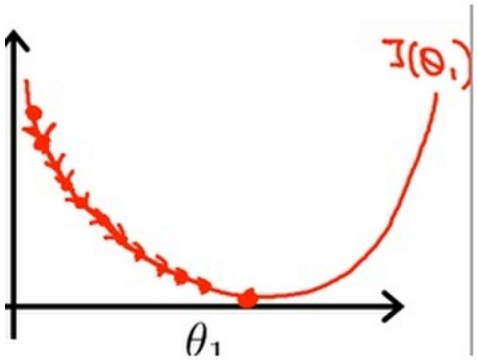
What do we do when functions are nonlinear, or the batch equations are too expensive to solve?

Gradient descent can minimize a total cost $C(\mathbf{p})$ by adjusting the parameter vector $\mathbf{p}$ in a direction that reduces the cost:

$$\Delta\mathbf{p} = -\epsilon\frac{\partial C(\mathbf{p})}{\partial\mathbf{p}} \tag{12}$$

where ϵ is a step size.

$$\theta_1 := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_1)$$



Using the current parameter vector, we can predict the y values:

$$\hat{y}_i = f(x_i, \mathbf{p}) \tag{13}$$

The error $\hat{y}_i - y_i$ may be made smaller by changing the parameter vector $\mathbf{p}$. The total cost of a parameter vector $\mathbf{p}$ is:

$$C(\mathbf{p}) = \sum_i (\hat{y}_i - y_i)^2 = \sum_i (f(\mathbf{x}_i, \mathbf{p}) - y_i)^2 \tag{14}$$

$$\frac{\partial C(\mathbf{p})}{\partial\mathbf{p}} = \sum_i (f(\mathbf{x}_i, \mathbf{p}) - y_i)\frac{\partial f(\mathbf{x}_i, \mathbf{p})}{\partial\mathbf{p}} \tag{15}$$

This is the sum of the prediction error on each training data point times an estimate of the effect of each parameter on the prediction at that point.