Explanatory Notes for 6.3900

Shaunticlair Ruiz

Fall 2024

Contents

CHAPTER 10

Markov Decision Processes 0 - State Machines

10.1 State Machines

10.1.1 How to Model Time

How do we want to model time?

The simplest way is one we've used before: keeping track of the current **timestep** t.

- But this is too little information to be useful: it doesn't tell us much.
- **Example:** If I told you "the current time is $t = 1563$", that doesn't help you much with decision-making.

So, what would be a **useful** representation of time? We've already shown that we don't really care much about the exact **index** of time t.

Instead, we care about **what happened** in the past.

> **Concept 1**
> One simple way to record the past is to ask about **events**, and **when** they happened.

Example: You might keep track of a medical history, or the purchases made by a company over the last year.

10.1.2 States

Keeping a "history" of events is an **improvement**. In some contexts, though, it can become **expensive**: the **longer** our time frame, the more events will pile up.

We could ignore very **old** events, but whether an old event matters depends on the context.

- **Example:** If we omit all company profits/expenses from more than 3 years ago, we don't know our balance. What if we forgot a debt?

This particular example has a pretty simple solution: just keep track of the **total** amount of money you have.

And herein lies our *general* solution: rather than keeping track of every single event, we can keep track of the **state** that result from those past events.

Definition 2

A **state** represents information we use to keep track of the **current situation** you're in.

It allows us to store "**memory**" about the past:

- If an event changes our current situation, we'll **update** the state.
- Then, in future timesteps, we'll use that updated state.

A state can be almost **any information** that we want to keep.

- In practice, we want to exclude unhelpful, irrelevant, or outdated data.

Example: Suppose that you're an investor. Your state could include: 1. how much money you have, 2. the stocks you current own, and 3. whether the market seems to be going up or down.

- Notice that, while these variables don't give you exact time, they do **remember** past events: if you have $30, you at some point must have gotten those $30.

There are many other kinds of states: position and velocity of an object, or the progress on a project, etc.

10.1.3 How states are stored

Now that we've introduced the idea, we'll start formally notating it.

Notation 3

Typically, a **state** s stores our information as a **vector**.

We represent the **set** of all possible **states** as $\mathcal{S}$.

- We can have a **finite** or **infinite** set of states, depending on the situation.
- If s is one of our states, we can express that as $s \in \mathcal{S}$.

Our state at time t is s_t.

Our **initial state** ($t = 0$) is represented as s_0.

- Since it's a state, $s_0 \in \mathcal{S}$.

We now have two of the pieces of our state machine:

- $\mathcal{S}$ is a finite or infinite **set** of possible **states** s.
- $s_0 \in \mathcal{S}$ is the **initial state** of the machine.

10.1.4 State examples

Let's show a couple examples of what states different systems might have.

There are multiple different ways to represent the same set of states with a vector, so we won't specify the representation.

- The game of chess.
  - The **finite** set S is the set containing **every chess board**.
  - The initial state s_0 is the **board** when you first **start playing**.
- A ball moving in space, with coordinates.
  - The **infinite** set S contains **every pair** $\left[\text{position, velocity}\right]$ for the ball.
    - For example, the ball might be in state $\left[(1,2),(5,0)\right]$:
    - at position $(1,2)$,
    - with velocity $(5,0)$.
  - The initial state s_0 is the **position and velocity** when you first **release** the ball.
- A combination lock with 3 digits.
  - The **finite** set S contains every **sequence of 3 digits**, where only one sequence unlocks the lock.

    - For example: $[0,0,0]$, $[4,6,9]$, $[9,0,2]$, etc.

  - The initial state s_0 is the sequence when you **leave** your lock; maybe $[1,2,3]$.

10.1.5 Input

We now have a way to **store** our information in time. However, we need to know how to **update** our state: what happens if we learn new information?

We'll include some new variables to address this.

At each timestep, we get some kind of **input** x, which is our update: this is the newest information about our system. We'll *also* store this in a vector.

Definition 4

The **input** x represents **new information** we get from our system.

We represent the **set** of all possible **inputs** as $\mathcal{X}$.

- This set can be **finite** or **infinite**.
- We can say $x \in \mathcal{X}$.

Our input at time t is x_t.

10.1.6 Transition

Based on this new information, we need update the current **state** of the world.

- But often, this update depends both the new information, **and** the **old state**.

Example: If your timestep update tells you "got 50 dollars", you need to know how much money you had before, to get your new total.

$$\overbrace{s_{t+1}}^{\text{New balance}} = \overbrace{s_t}^{\text{Old balance}} + \overbrace{x_t}^{\text{Money added}} \tag{10.1}$$

Here's a second example.

Example: Suppose you're taking care of a plant.

- If a plant is dry ($s_t = \text{Dry}$), then watering it will make it healthier ($s_{t+1} = \text{Healthy}$).
- If the plant is watered ($s_t = \text{Healthy}$), then watering it more might make it sick ($s_{t+1} = \text{Sick}$).

We're **transitioning** between states, so we use a **transition function**.

Definition 5

The **transition function** f_s tells us how to update our **state**, based on our new **input** information.

- Thus, our transition takes in two pieces of information: s and x.

$$f_s(s, x)$$

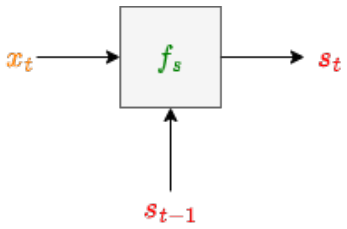
- We use this function at **every timestep** t to get our next state, at time $t+1$.

$$f_s(s_t, x_t) = s_{t+1}$$

We can treat each state-input **pair** as an object, (s, x). Thus, the set of all of these pairs is $\mathcal{S} \times \mathcal{X}$.

$$f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$$

We can visualize this as:



Now, we have two more pieces of our state machine:

- $\mathcal{X}$ is a finite or infinite set of possible **inputs** x.
- f_s is the **transition function**, which moves us from one state to the next, based on the input.

$$f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S} \tag{10.2}$$

10.1.7 Transition Examples

Now, we revisit our examples, and consider how they "transition":

- The game of chess.

  - The input x is the **choice** our player makes, **moving one piece** on the chess board according to the **rules**.
  - The transition function f_s applies this move to our current chess board, and produces a **new chess board**.
    * If we moved our pawn, the transition function outputs the board **after** that pawn is **moved**.

- A ball moving in space, with coordinates.
  - The input x might represent a **push** changing the ball's velocity.
  - The transition function f_s uses the push to change our **velocity**, and the velocity to change the ball's **position**.
    * If our ball wasn't moving before, and we **push** it, the new state is **moving** in that direction.

- A combination lock with 3 digits.
  - The input x is you **changing** one of the three digits on the lock: for example, **increasing** the first digit by 3.
  - The transition function f_s applies the **change** you make to the lock.
    * If the first digit was 2, and you **increase** it by 3, the new first digit is 5.

10.1.8 Output

We now have a system for keeping **track** of our state, and **updating** that state: this is a really powerful tool for managing time!

We're still missing something, though: why do we **care** about our state? Typically, there's some **result** we actually want from storing our state.

Just like how in CNNs, convolution wasn't the end goal: it was a transformation to help improve regression/classification.

Usually, the desired output is more simple than keeping track of everything we want to **remember**.

Example: If we're storing a bunch of information about the stock market, and our own money, we might simply return "invest in X" or "do not invest in X".

The is what we call our **output**.

Definition 6

The **output** y represents the **result of our current state**.

What we use as "output" depends on what we are **trying to predict/compute**.

- Sometimes, the output is the **only** thing (aside from input) we can **see**. This happens when the state is **hidden**!

In other words, while the state **stores** relevant information to keep track of the situation, the **output** is the decision based on this information.

We represent the **set** of all possible **outputs** as $\mathcal{Y}$.

- This set can be **finite** or **infinite**.
- If y is a possible output, we say $y \in \mathcal{Y}$.

Our output at time t is y_t.

Note that we use y_t: we don't necessary create a single output at the end of our runtime.

- Instead, we continuously create outputs at each timestep.

10.1.9 Output Function

So now, we need to actually **compute** our output. This will be based on all the data we have **stored** at the time we're asked for an output.

- We don't need to use the input, because the input data is already included in the state.

We can create an output for each timestep using the **output function**.

Definition 7

The **output function** f_o tells us what **output** we get based on our current **state**.

Thus, our **output function** only takes in the **state**.

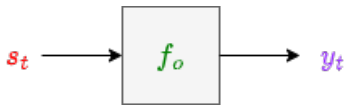
$$f_o(s_t) = y_t$$

It uses our current information (**state**) to produce a the result we're interested in (**output**).

Using sets, we can write this as:

$$f_o : \mathcal{S} \to \mathcal{Y}$$

We visualize this unit as:



This gives us the last two parts of our state machine:

- $\mathcal{Y}$ is a finite or infinite set of possible **outputs** y.
- f_o is an **output function**, which gives us our output based on our state.

$$f_o : \mathcal{S} \to \mathcal{Y} \tag{10.3}$$

10.1.10 Output Examples

Again, we go to our examples, and give them outputs, completing our state machines:

- The game of chess.
  - The output y could be many things. But, what do we care about most: **winning**!
    - So, $\mathcal{Y}$ will have four options: "ongoing", "draw", "player 1 win", "player 2 win".
  - The output function f_o will give us our output. Thus, it represents the **chess rules** for whether there is a winner or a draw.
    - So, f_o looks at a board, and tells us whether someone has won, or there's a draw.

- A ball moving in space, with coordinates.
  - We want output y.
    - Sometimes, the **output** is the same as the **state**: all we want to know is what's **happening**!
    - In this case, we'll say our **output is the state**: we return the **position** and **velocity** of the ball.
  - If our state and output are the same, then the output function f_o should just **copy** the state it receives!
    - Our function is the **identity function**: $f_o(s) = s$.

We could have chosen a different output if we had a specific goal in mind!

- A combination lock with 3 digits.
  - We want our output y.
    - Our goal is more clear: we want the combination lock to be **open** or **closed**. So, those are our outputs y.
  - Our function f_o will tell us the lock is open if the current digits exactly **match the correct sequence**.

10.1.11 A Completed State Machine

Finally, we can assemble our completed state machine.

Definition 8

A **State Machine** can be formally defined as a collection of several objects

$$(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f_s, f_o)$$

We have three sets:

- $\mathcal{S}$ is a finite or infinite **set** of possible **states** s.
- $\mathcal{X}$ is a finite or infinite **set** of possible **inputs** x.
- $\mathcal{Y}$ is a finite or infinite **set** of possible **outputs** y.

And components to allow us to transition through time:

- $s_0 \in \mathcal{S}$ is the **initial state** of the machine.
- f_s is the **transition function**, which moves us from one state to the next, based on the input.
$$f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$$
- f_o is an **output function**, which gives us our output based on our state.
$$f_o : \mathcal{S} \to \mathcal{Y}$$

We have:

- Our **state** to store information,
- Our **input** to update information,
- Our **output** gives us the result of our information.

And to combine these, we need:

- Our **initial** state,
- How to **change** states,
- How to **get** an **output**.

10.1.12 Using a State Machine

How do we work with a state machine? Well, we have all of the tools we need.

We start with out initial state, s_0. For our **first** timestep, we get a new input: new **information**. We use this to get a new state.

$$s_1 = f_s(s_0, x_1) \tag{10.4}$$

With this state, we can now get our **output**.

$$y_1 = f_0(s_1) \tag{10.5}$$

We've calculated everything in our **first** timestep! Now, we can move on to our **second** timestep, and do the same thing.

In general, we'll repeatedly follow the process:

$$s_t = f_s(s_{t-1}, x_t) \tag{10.6}$$

$$y_t = f_o(s_t) \tag{10.7}$$

Concept 9

To move through time in a state machine, we follow these steps from $t = 1$:

- Use the input and state to get our new state.

$$s_t = f_s(s_{t-1}, x_t)$$

- Use the new state to get our output.

$$y_t = f_o(s_t)$$

- Increment the time from t to $t + 1$.

$$t_{new} = t_{old} + 1$$

- Repeat.

10.1.13 Example Run-Through of a State Machine

To make this more concrete, we'll build our own simple state machine and run a couple iteration steps.

Suppose you're saving up money to buy something. At each timestep, you gain or lose some money.

You want to know when you have enough money to buy it.

This example is simple enough that you might feel like a state machine is unnecessary. However, this is just for demonstration!

What are each of the parts of our state machine?

- The state s: how much money do we have right now?
- The input x: the money we add to our savings.
- The output y: we want to know when we have enough money. Maybe our goal is 10 dollars.
- Initial s_0: we start with 0 dollars.
- Transition f_s: we just add the new money to how much we have saved up.

$$f_s(s, x) = s + x \tag{10.8}$$

- Output f_o: do we have enough money?

$$f_o(s) = (s \geqslant 10) = \begin{cases} \text{True} & \text{If } s \geqslant 10 \\ \text{False} & \text{Otherwise} \end{cases} \tag{10.9}$$

We'll run through our state machine for the following input:

$$X = [x_1,\ x_2,\ x_3,\ x_4] = [4,\ 5,\ 6,\ -7] \tag{10.10}$$

Let's apply the steps above:

- Get new state from (old state, input).
- Get output from new state.
- Increment time counter.

For our first step, we get:

$$\begin{aligned} s_1 &= 4 + 0 = 4 \\ y_1 &= (4 \geqslant 10) = \text{False} \end{aligned} \tag{10.11}$$

For the others, we get:

$$\begin{matrix} s_2 = 9 \\ y_2 = \text{False} \end{matrix} \longrightarrow \begin{matrix} s_3 = 15 \\ y_3 = \text{True} \end{matrix} \longrightarrow \begin{matrix} s_4 = 8 \\ y_4 = \text{False} \end{matrix} \tag{10.12}$$

Though our transition and output functions might become more complicated, this is the basic idea behind all state machines.

10.1.14 State Machine Diagram

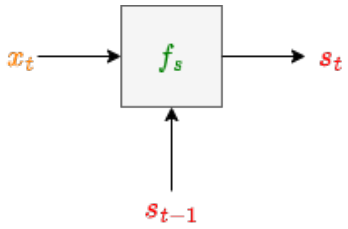
Finally, we'll create a visualization that represents our state diagram.

10.1.14.1 Transition Function

Our transition function follows this format:

$$s_t = f_s(s_{t-1}, x_t) \tag{10.13}$$

We can diagram this component as:



Note that the state appears **twice**: once as an input, once as an output.

In the *next* timestep, s_t will be the **input** to f_s, even though it's currently the **output**.

If $t = 10$, then s_{10} is the output. If $t = 11$, then s_{10} is the input!

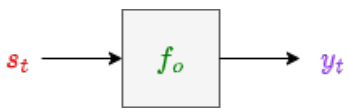
- We'll create a way to represent this later.

10.1.14.2 Output Function

Our **output function** takes in the state we just got from the transition function:

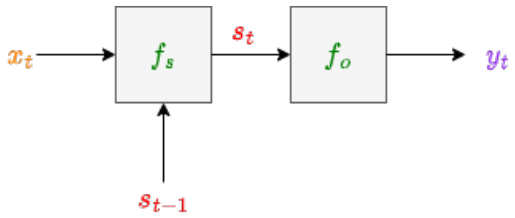
$$y_t = f_o(s_t) \tag{10.14}$$

So, we diagram it accordingly:



As we mentioned, the **output** function takes in the state as its input.

- That means that the **output** of f_s, is the **input** of f_o.

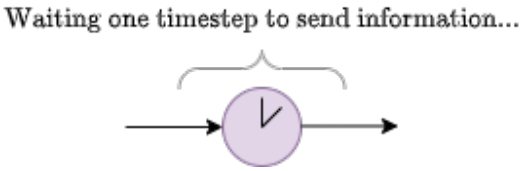


10.1.14.3 Time Delay

Only one thing is missing: we know that our current state s_t needs to be reused **later**: we'll need it to compute our *new* state s_{t+1}.

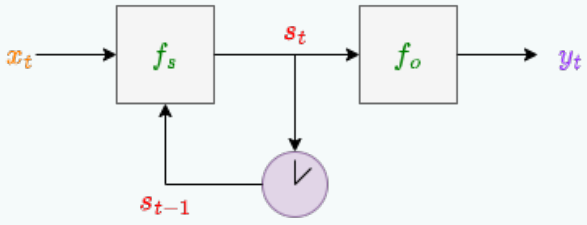
We don't want it to *immediately* send the state information back to f_s: we only use the function once per timestep. So, we'll *delay* by waiting one time step.

We'll use a little clock symbol to represent this fact.



Notation 10

We can depict a **state machine** using the following diagram:



At every timestep, we use x_t and s_{t-1} to calculate our new state, and our new output.

The circular "clock" element represents our **delay**: s_t becomes the input to f_s on the **next** timestep.

10.1.15 Finite State Machines

To get used to state machines, we'll start with a simpler, special case, the **finite state machine**.

Definition 11

A **finite state machine** is a state machine where

- The set of states $\mathcal{S}$
- The set of inputs $\mathcal{X}$
- The set of outputs $\mathcal{Y}$

Are all **finite**. Meaning, the total space of our state machine is **limited**.

Each aspect of our state machine can be put into a finite list of elements: this often makes it easier to *fully* describe our state machine.

This seemingly limited tool is more powerful than it seems: **all computers** can be described as finite state machines!

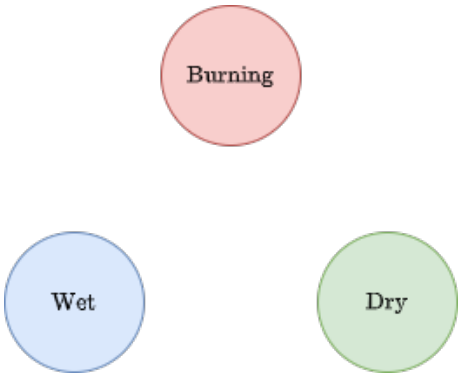
Even when a computer seems to be describing "infinite" collections of things, it only has a finite amount of space to represent them.

10.1.16 State Transition Diagrams

One nice thing about the simplicity of a finite state machine is that we can represent it **visually**.

Let's build one up: we'll pick a simple, though not entirely realistic example.

Example: We have a blanket. It can be in three states: either wet, dry, or burning. We can represent each state as a "node".

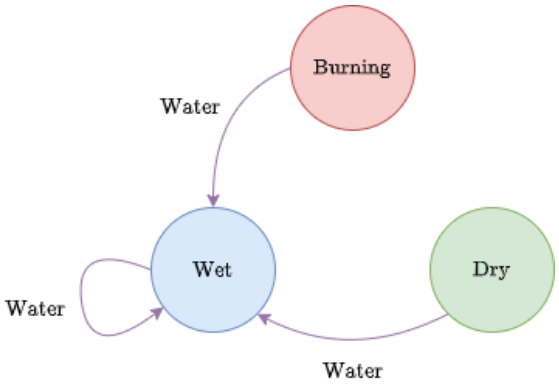


Concept 12

In a **state transition diagram**, states are represented as **nodes**, or points on the graph.

We have our states down. The other important thing is our **transitions**. How do we go between states?

Well, one input could be **water**: it would stop the blanket from burning. In any case, the blanket will be wet.



Now, we can see: each arrow represents a **transition** between two states. Each **input** gets its own transition.

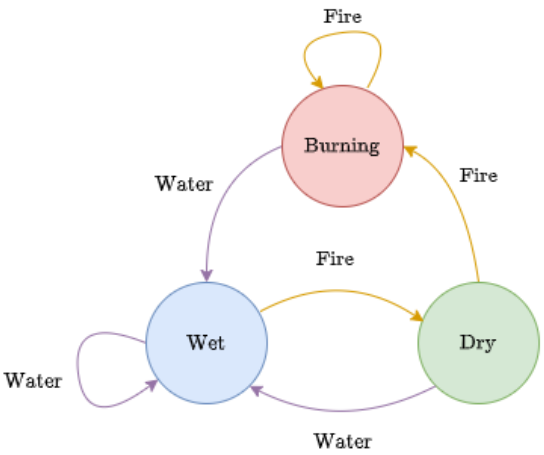
Concept 13

In a **state transition diagram**, transitions are represented as **arrows** between states.

We usually label these with whichever **input** will cause that transition.

Also notice that a state can transition to itself: a wet blanket **stays wet** when you add water.

What if we add **fire**? That would make a dry blanket **burn**. But, we could also use it to **dry off** the wet blanket!



And now, we have a simple **state transition diagram**!

Note that our diagram doesn't show the output. In this case, that's not a problem: the output is the state.

Each transition, as usual, is based on two things: the **current** state (where the arrow starts) and the **input**(which arrow you follow).

Definition 14

A **state transition diagram** is a **graph** of

- Nodes (points) representing states
- Directed edges (arrows) representing transitions

Where each input-state pair has one arrow associated with it.

These arrows show one transition, with the properties:

- The start and end states represented by the start and the end of the arrow
- The input that causes this transition is labelled.

This diagram does not have to show the input.

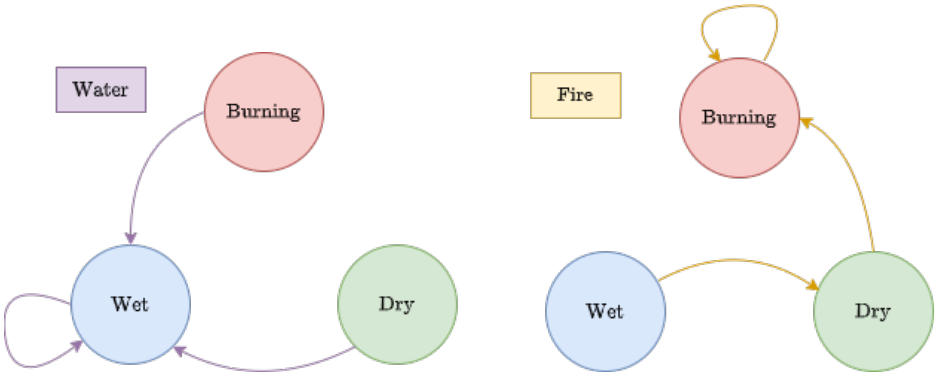
If you're not familiar with "nodes" or "edges", don't worry about it! For our purposes, "point" and "arrow" are good enough.

10.1.17 Simplified state transition diagrams: One-input graphs

One more consideration: the graph above is helpful, but it's a bit **complicated**.

In fact, if we added more **states**, or more **inputs**, it could get too complicated to read!

Our solution: if a system is too complicated, we create a separate state-transition diagram for **each input**.



The left diagram only uses **water** as an input, while the right diagram only uses **fire** as an input.

Each of our diagrams is much more readable now! Not only do we have less arrows, but we don't have to label each arrow.

As a tradeoff, we have two graphs to keep track of, instead of one. However, this is usually

necessary.

In the next chapter, MDPs, we'll need this!

Concept 15

We can simplify our **state transition diagrams** by creating a **separate diagram** for each input.

This makes it easier to visualize what's going on.

10.1.18 Linear Time-Invariant Systems (LTI)

A wide range of problems can be modelled by a simplified, **linear** version of this system.

That means we'll work entirely with vectors and matrices: no non-linear functions.

- Our **states** are all vectors of length m.
- Our **inputs** are all vectors of length ℓ.
- Our **outputs** are all vectors of length n.

$$\mathcal{S} = \mathbb{R}^m \qquad \mathcal{X} = \mathbb{R}^\ell \qquad \mathcal{Y} = \mathbb{R}^n \tag{10.15}$$

To transition between states, we'll **linearly** combine state s_{t-1}, with our input x_t: A and B are **matrices**.

$$s_t = As_{t-1} + Bx_t \tag{10.16}$$

Notice that we **exclude the offset terms**: this is due to our definition of **linear**.

Clarification 16

There are two related, but **distinct** definitions for what it means to be **linear**:

- The kind of linear we're more used to: "an equation that draws a line". This is allowed to have an **offset**.

$$f(x) = W^T x + W_0$$

- The kind of linearity used in **linear combinations**, where you're only allowed to **scale** and **add** the inputs together: **no offset**.

$$f(x) = W^T x$$

The latter allows us to use the **linearity** property:

$$f(a+b) = f(a) + f(b) \qquad f(ca) = cf(a)$$

While the former definition, with the offset, does not.

Our output will simply be a **linear** scaling of our state: C is a matrix.

$$y_t = Cs_t \tag{10.17}$$

We'll also find a second, interesting property:

Definition 17

Time-invariance is the property of our input having the **same** effect on our system, no matter what **time** we apply it.

Thus, we call this restricted model a **Linear Time-Invariant System (LTI)**.

Definition 18

A **Linear Time-Invariant System (LTI)** is a variant of a **state machine**, where

- Our input x, state s, and output y are all vectors

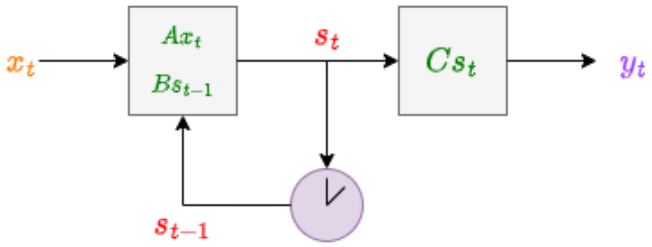
$$\mathcal{S} = \mathbb{R}^m \qquad \mathcal{X} = \mathbb{R}^\ell \qquad \mathcal{Y} = \mathbb{R}^n$$

- Our transition and output functions are linear (with A, B, and C being matrices)

$$s_t = f_s(s_{t-1}, x_t) = As_{t-1} + Bx_t$$

$$y_t = f_o(s_t) = Cs_t$$

We can depict this with a modified version of our state machine diagram from above:



This kind of model is often an excellent approximation of simple systems in physics, signals, and other sciences.