Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

Contents

CHAPTER 9

Recurrent Neural Networks - Explanatory Notes

9.0.1 Review: Neural Networks So Far

In this class, we have built up some powerful tools, and built them into **models** we can teach to handle different tasks.

All of this has culminated in the **neural network**: a class of models that can handle a huge number of interesting problems.

- All we had to do was combine many simple models together in a systematic, **nonlinear** way!

Remember that by "systematic", we mostly just mean "organized": that way, we can do math easier!

We then discovered a *weakness* of neural networks: they don't understand **space** very well!

- Our models had trouble recognizing that pixels in an image close to each other were more *related* than pixels far away, for example.

The solution was the **convolutional** neural network: we used *convolution* as a way to represent which elements were "near" each other in space.

9.0.2 Time in a Neural Network

If we want our models to be able to handle *space*, we might also wonder: can we make it so they understand **time** as well?

Right now, our neural networks have no built-in way to represent *time*: each data point stands by itself.

Note that we're focused on the finished model: sure, time passes while training, but the final model doesn't keep track of the past.

For example, suppose we have two data points $x^{(1)}$ and $x^{(2)}$. To our current neural network, there's **no difference** if we **switched** the order of those data points.

But what if it *does* matter? In real life, past information can matter a lot, as can the order.

Example: If you injured your leg yesterday, you probably don't want to walk around on it too much today. If you injured it a month ago, you might not care.

Concept 1

A traditional **neural network** cannot easily use information about **time**, or the **past**.

9.0.3 Our plan going forward

Now that we've settled on approaching the problem of **time**, we'll spend the next couple chapters building ways to tackle this problem:

- **Ch.9:** First, we'll build something called a "**state machine**" to record information about time.
  - We'll use this "machine" to modify our neural networks, creating something called a **recurrent neural network** (RNN) that can analyze information over time.
- **Ch.10:** Then, we'll create a model that makes **decisions** over time: a "**Markov Decision Process**" (MDP).
- **Ch.11:** Finally, we'll use state machines to **learn** how to make good decisions in an unknown environment: this is **Reinforcement Learning**.

9.1 State Machines

9.1.1 Possible ways to model time

We said that we want to think about time, but what does that really mean?

Example: In the simplest case, we'd just keep track of the *current* timestep t.

This is too *little* information. It doesn't really tell us that much: if I told you "the current time is t = 1563", that doesn't help you make decisions without more context.

Let's use the *injured leg* example:

Example: If you injured your leg yesterday, we might want to remember the **event**, and when the event **happened**.

But eventually, your leg *heals*, and that event doesn't **matter** anymore. As time continues, we'd gather more and more events... that could get **expensive** to keep track of.

So, remembering every event is too **much** information.

9.1.2 States

At this point, someone might get annoyed: "just tell me whether my leg hurts or not!" And therein lies our solution.

Example: The only information we care about is the current **state** of your leg: is it injured or is it not?

Thus, we only store information about our current condition that **matters**: we need to update this over time, of course.

This is called a **state**.

Definition 2

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

A state allows us to keep some "**memory**" of the past, but only the parts that matter:

If an event has **changed** the situation, it'll change the **state**.

A state can be almost **any information** that we want to keep. It can contain **multiple** different pieces of information, as well.

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.

9.1.2.1 How states are stored

Now, we want to know how to notate this consistently:

Notation 3

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

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

- We can have a **finite** or **infinite** set of states, depending on the situation.
- Since $\mathcal{S}$ contains all of our states, we can say $s \in S$.

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.

9.1.2.2 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 that 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 [position, velocity] for the ball.
    - For example, the ball might be:
    - 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 $[1,2,3]$.

9.1.3 Inputs and Transitions

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?

In order to do this, we'll create a few more variables.

9.1.3.1 Input

First, we get some kind of **input** x, which is our update: this is the most recent information we have. This is *also* stored 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.

9.1.3.2 Transition

Now, how do we use this input? Well, our new information will affect our **state**. But, often, **how** it affects our state depends on the state itself.

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:

$$\begin{matrix}\text{State}\\ \text{Input}\end{matrix} \longrightarrow \boxed{f_s} \longrightarrow \text{New state} \tag{9.1}$$

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.

$$f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S} \tag{9.2}$$

9.1.3.3 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 frm-e, and you increase it by 3, the new first digit is 5.

9.1.4 Output and Output Function

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? We should have some sort of **reason** for wanting to store the state.

9.1.4.1 Output

Usually, that information is more simple than all of the parts of the state we want to **remember**.

Example: In the above case, where we wanted to know if our leg was injured, the real thing we cared about was: **can I walk or not**?

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

Definition 6

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

What we consider the "output" often depends on what we are **focused on**.

- 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 result of this information.

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

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

Our output at time t is y_t.

9.1.4.2 Output Function

How do we get this output? Well, we use all the information we've **gathered**.

We could include both the input and the state, but the state has already been **updated** to reflect the input. So, we only need that.

This reflects what we said before: our state exists to store **memory**, and use the information to create an **output**.

We turn state into the output using an **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 new 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:

$$\text{State} \longrightarrow \boxed{f_o} \longrightarrow \textbf{Output}$$

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.

$$f_o : \mathcal{S} \to \mathcal{Y} \tag{9.3}$$

9.1.4.3 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.
  - The output y again depends on what we care about.
    - 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 shouldn't alter the state it receives!
    - Our function is the identity: $f_o(s) = s$.
- A combination lock with 3 digits.
  - We want our output y.

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

    * Our goal is more clear: we want the combination lock to be **open** or **closed**. So, those are our outputs $\mathcal{Y}$.

  - Our function f_o will tell us the lock is open if the current digits exactly match the correct sequence.

9.1.5 Our 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 preserve 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**.

9.1.6 How to use 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.

$$s_1 = f_s(s_0, x_1) \tag{9.4}$$

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

$$y_1 = f_0(s_1) \tag{9.5}$$

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:

$$s_t = f_s(s_{t-1}, x_t) \tag{9.6}$$

$$y_t = f_o(s_t) \tag{9.7}$$

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.

9.1.6.1 Example of 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.

$$f_s(s, x) = s + x \tag{9.8}$$

- Output f_o: do we have enough money?

$$f_o(s) = (s \geqslant 10) = \begin{cases} \text{True} & \text{If } s \geqslant 10 \\ \text{False} & \text{Otherwise} \end{cases} \tag{9.9}$$

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

$$X = [x_1,\ x_2,\ x_3,\ x_4] = [4,\ 5,\ 6,\ -7] \tag{9.10}$$

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:

$$s_1 = 4 + 0 = 4$$
$$y_1 = (4 \geqslant 10) = \text{False} \tag{9.11}$$

For the others, we get:

$$\begin{matrix} s_2 = 9 \\ y_2 = \text{False} \end{matrix} \longrightarrow \begin{matrix} s_3 = 15 \\ y_3 = \text{True} \end{matrix} \longrightarrow \begin{matrix} s_4 = 8 \\ y_4 = \text{False} \end{matrix} \tag{9.12}$$

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

9.1.7 State Machine Diagram

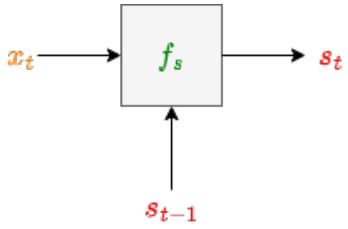
Finally, we'll create a visualization that represents our state diagram.

9.1.7.1 Transition Function

Our transition function follows this format:

$$s_t = f_s(s_{t-1}, x_t) \tag{9.13}$$

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!

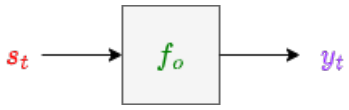
We'll create a way to represent this later.

9.1.7.2 Output Function

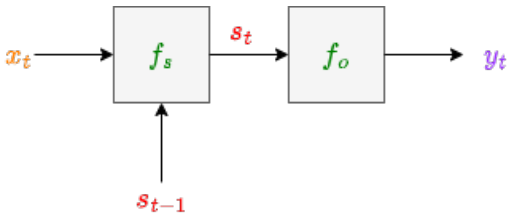
Our output function takes in the state we just got from the transition function:

$$y_t = f_o(s_t) \tag{9.14}$$

So, we diagram it accordingly:



As we mentioned, the **output** function takes in the **transition** function as its input: let's depict that! We'll combine our two units.

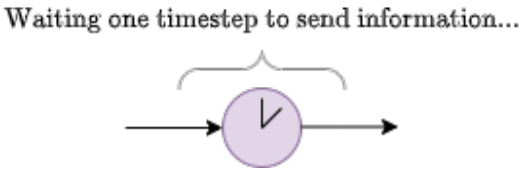


9.1.7.3 Time Delay

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}.

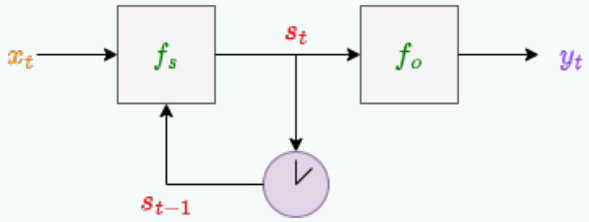
We don't want it to *immediately* send the state information back to f_s: we only use the function once per timestep. So, we'll *delay* by waiting one time step.

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 our **delay**: s_t becomes the input to f_s on the **next** timestep.

9.1.8 State Transition Diagrams

9.1.8.1 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!

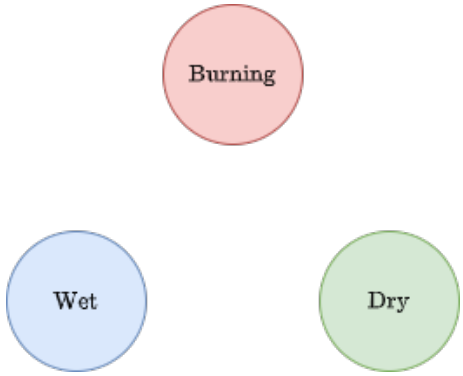
Even when a computer seems to be describing "infinite" collections of things, it only has a finite amount of space to represent them.

9.1.8.2 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.

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

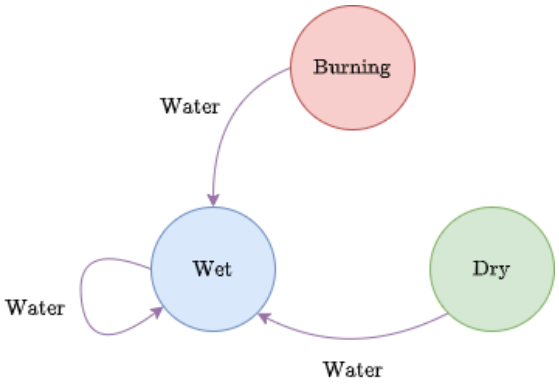


Concept 12

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.

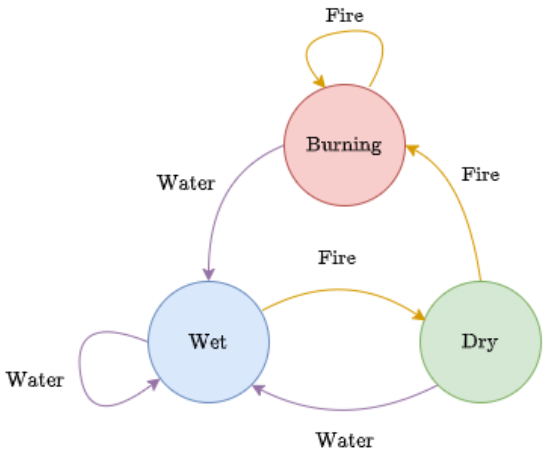
Concept 13

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**!

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

Note that our diagram doesn't have to show the output: the output is given by the state, so each node has its own output

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.

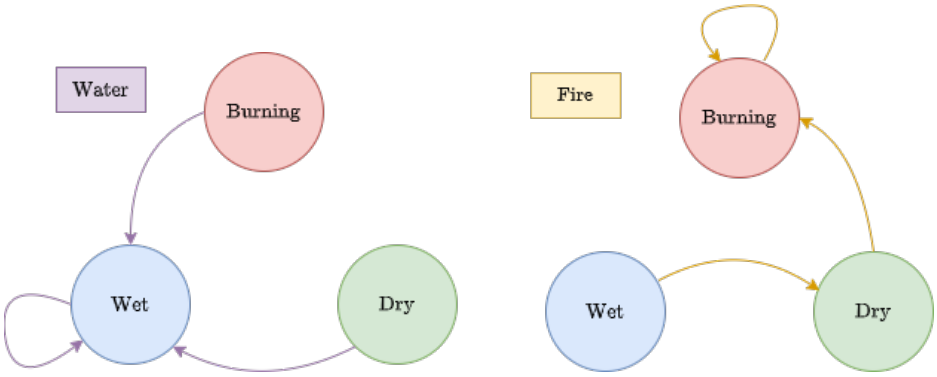
If you're not familiar with "nodes" or "edges", don't worry about it! For our purposes, "point" and "arrow" are good enough.

9.1.8.3 Simplifying 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.