# Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Spring 2024

# Contents

CHAPTER 10

# Recurrent Neural Networks

### 10.0.1 Review: Neural Networks So Far

In this class, we design **models** we can **train** to handle different tasks.

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

- To create a neural network, we combine many smaller, simpler models together in a systematic, **non-linear** way.
- This creates the "**fully connected**" (FC) neural network.

Remember that by "systematic", we mostly just mean "organized":

The parts of our model are cleanly organized, to make our math easier.

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

- **Example:** FC networks don't encode information about which pixels are **close** or **far** from each other.

Our solution was the **convolutional neural network** (CNN):

- We used **convolution** as a way to represent which elements were "near" each other in space.

### 10.0.2 Time in a Neural Network

We've created models that can model *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:

The model changes while training, but the fully-trained model **doesn't** keep track of the past.

- As we discussed in the CNN chapter, the **order** of our input variables is mostly **ignored** by the model.

**Concept 1**
A traditional, fully connected **neural network** cannot easily use information about **time**, or the **past**.

Previously, we added structure to NNs using **convolution**. But this *doesn't* work as well in time as it does in space:

**Concept 2**
**Time** and **space** behave differently:

- Information can be spread out over **any direction** in space.
- However, information only travels **forward**, not backward, in time.

So, we need to model them differently.

Realistically, we want model that can keep track of time, and order of past events, for plenty of purposes:

- **Example:** Stocks, weather, choosing the best plan of action, etc.
  - "It rained yesterday" and "it rained last week" have very different effects on today's weather.

### 10.0.3 Our plan going forward (Optional)

Our new theme is **time**. There a several different ways to approach this topic, and we'll explore each over several chapters:

- **Ch.10: State Machines**
  - A "state machine" stores everything we need to know about the **past**, as a single "state".

  - Usually, rather than keeping track of the whole history, we just store what's **important** to remember.

- **Ch.10: Recurrent Neural Networks (RNNs)**
  - An RNN is a **model** that uses our new system of "states" to **store** the past, so we can use it in our model.
  - Our RNN makes predictions based the present (input $x_t$) **combined with** the past (state $s_{t-1}$).

- **Ch.11: Transformers**
  - Transformers introduce the idea of **attention**: "how much should we **pay attention** to other pieces of data, to **understand** this piece of data?"
  - We represent "time" by **masking**, or **hiding** the future data from our model.

- **Ch.12: Markov Decision Process**
  - We go back to using **states** to store past information.
  - With MDPs, our model represents different possible **decisions** it can make over time, with different possible **rewards**.

- **Ch.13: Reinforcement Learning**
  - Reinforcement learning (RL) often uses **MDPs** as a basis: using past data to make the **best** possible **decision**, via states.
  - In our notes, we often focus on situations where we have **incomplete** information about our system, and learn by interacting with it.

We can think of chapters **10**-**12**-**13** as a combined sequence of **state machines**, **MDPs** (which use state machines), and **RL** (which uses MDPs).

Chapter 10 and 11 both introduce models which allow us to process language, and similar sequential tasks.

## 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 3**
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 4**

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 5**

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** $\left[\textbf{position}, \textbf{velocity}\right]$ 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 6**

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 7**

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 $x$.

$$f_s(s, x)$$

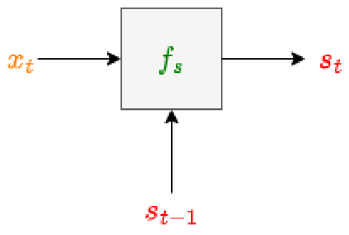
- We use this function at **every timestep** $t$ to get our next state, at time $t+1$.

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

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 8**

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 9**

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**.

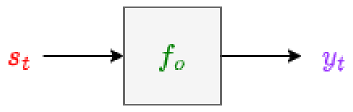
$$f_o(s_t) = y_t$$

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

Using sets, we can write this as:

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

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.

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

### 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 $\mathcal{Y}$.
  - 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 10**

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 11**

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:

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

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

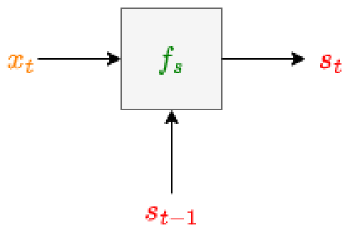
Finally, we'll create a visualization that represents our state diagram.

#### 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!

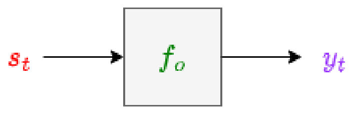
- We'll create a way to represent this later.

#### 10.1.14.2 Output Function

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

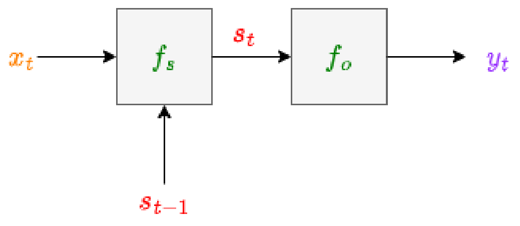
$$y_t = f_o(s_t) \tag{10.14}$$

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 $f_o$.

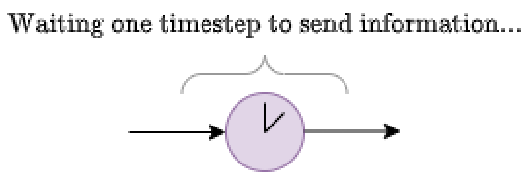


### 10.1.14.3 Time Delay

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 $s_{t+1}$.

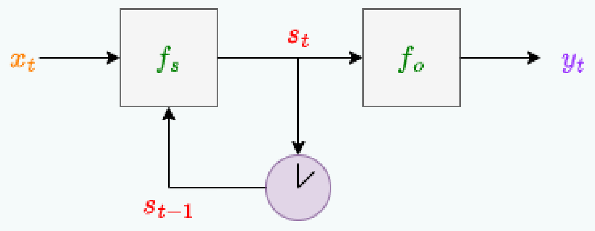
We don't want it to *immediately* send the state information back to $f_s$: we only use the function once per timestep. So, we'll *delay* by waiting one time step.

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



**Notation 12**

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 13**

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!

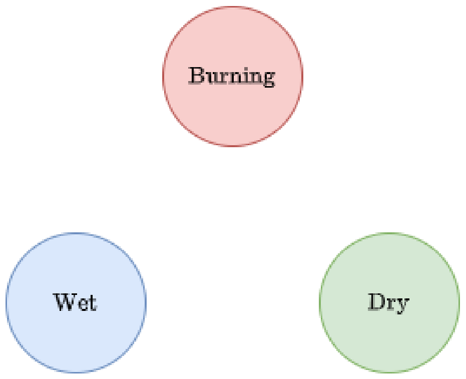
Even when a computer seems to be describing "infinite" collections of things, it only has a finite amount of space to represent them.

### 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".

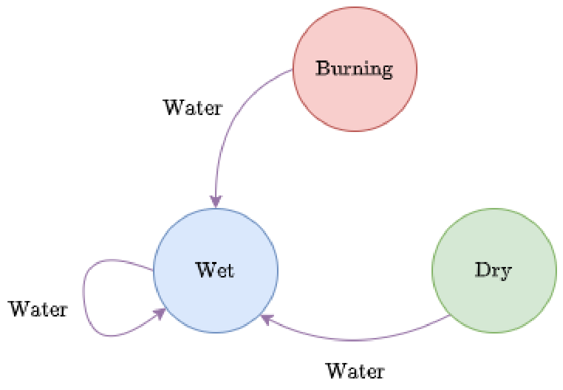


**Concept 14**

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.

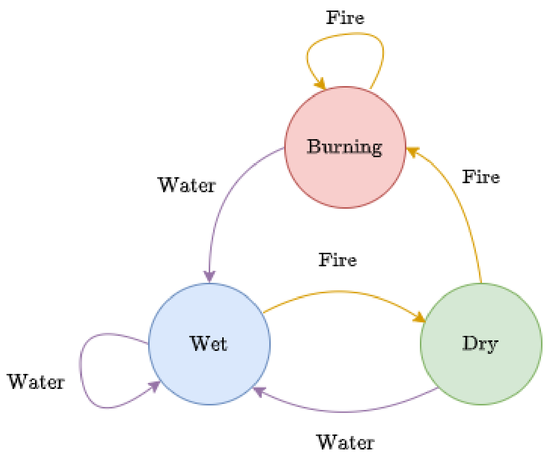
**Concept 15**

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 16**

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.

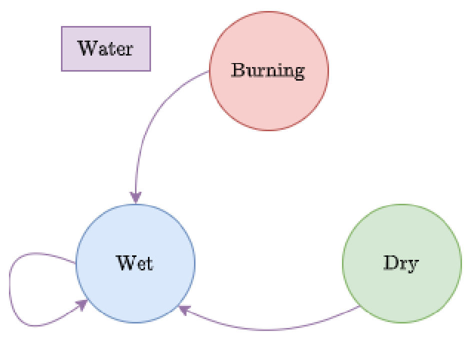
If you're not familiar with "nodes" or "edges", don't worry about it! For our purposes, "point" and "arrow" are good enough.

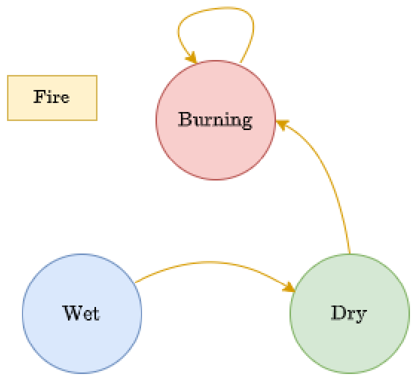
### 10.1.17 Simplified state transition diagrams: One-input graphs

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 17**

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 $\ell$.
- 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 18**

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 19**

**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 20**

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

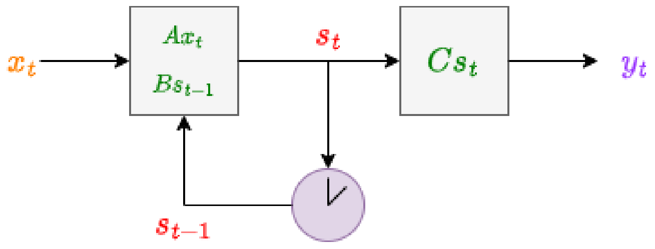
$$\mathcal{S} = \mathbb{R}^m \qquad \mathcal{X} = \mathbb{R}^\ell \qquad \mathcal{Y} = \mathbb{R}^n$$

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

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

$$y_t \quad = \quad f_o\Big(s_t\Big) \quad = \quad Cs_t$$

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.

In the next section, we'll add non-linear units to this linear model, and this will create a **Recurrent Neural Network**.

## 10.2 Recurrent Neural Networks

We've fully fleshed out our state machines above: we've developed a technique for managing time, using **memory**.

- This is similar to how, in the last chapter, we developed Convolution, to manage **space**.

Just as we did in convolution, we'll add state machines into our neural network system. This will give us the **Recurrent Neural Network (RNN)**.

### 10.2.1 Building up RNNs

How do we create a "network" using our state machine system?

- Well, a traditional NN applies a **linear** operation $z = W^T x + W_0$, and then a **nonlinear** operation, $a = f(z)$.
- We'll implement this process in our state machine, using $f_s$ and $f_o$.

The **linear time-invariant (LTI) system** we defined at the end of the last section gives us the simplest, most reduced version of this:

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

To replicate a neural network, we'll need to add two things.

**Concept 21**
An RNN modifies an LTI system in two ways:

- Includes **offset** terms in the linear part of $f_o$ and $f_s$
- Adds a **nonlinear** component after the linear component both functions.

### 10.2.2 Offset

Let's add that offset term:

$$f_s(s_{t-1}, x_t) = As_{t-1} + Bx_t + D \qquad f_o(s_t) = Cs_t + E$$

This is a bit cluttered. We'll rename all of these weights:

**Notation 22**

For our RNN, we'll label our **weights** as $W^{ab}$:

- $b$ represents the "**input**": what we're multiplying $W^{ab}$ with.
- $a$ represents the "**output**": what we're using $W^{ab}$ to compute.

We'll also use subscript $W_0$ for offset terms.

We can go through all of our weights like this.

**Notation 23**

We want to rewrite $f_s(s_{t-1}, x_t) \quad = \quad As_{t-1} + Bx_t + D$

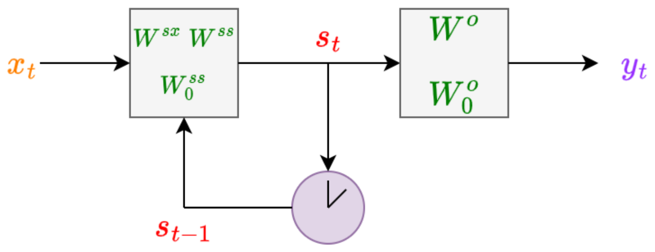
- $W^{ss}$ is combined with $s_{t-1}$, to compute $s_t$.
- $W^{sx}$ is combined with $x_t$, to compute $s_t$.
- $W_0^{ss}$ is the offset term that computes $s_t$.

$$s_t = W^{ss}s_{t-1} + W^{sx}x_t + W_0^{ss}$$

Now, we'll rewrite $f_o(s_t) \quad = \quad Cs_t + E$:

- $W^o$ is combined with $s_t$ to compute the $y_t$.
  - Based on conventions, you could also write it as $W^{os}$. But, since there's no $x$ term, this is unnecessary.
- $W_0^o$ is the offset term that computes $y_t$.

$$y_t = W^o s_t + W_0^o$$



### 10.2.3 Activation Function

Now, we've covered the linear part of our function. We'll apply a non-linear function $f$ and $g$ to each:

$$s_t = f\Big(W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss}\Big) \tag{10.18}$$

$$y_t = g\Big(W^{o} s_t + W_0^{o}\Big) \tag{10.19}$$

These are our **activation functions**.

$f$ and $g$ are pretty vague names, so it would be more helpful to name them according to what they're being used to compute: state and output.

- So, we could say $f = f_s$, and $g = f_o$.

**Clarification 24**

In previous sections, we've used $f_s$ and $f_o$ to indicate the **entire function** used to compute the state/output, including the **linear** part.

- However, we want to separate the linear and **non-linear** parts.

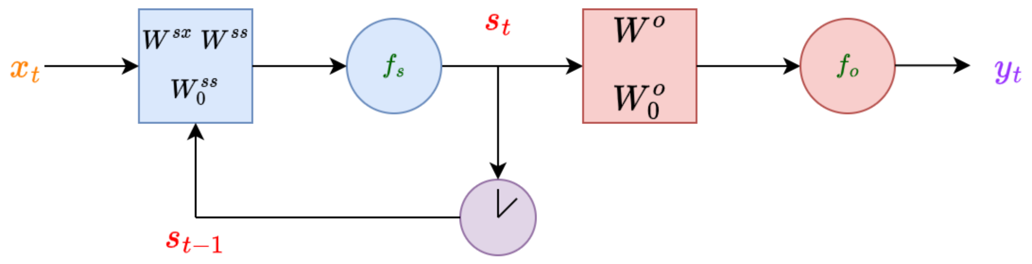
So, we'll switch conventions: $f_s$ and $f_o$ in sections 10.1 and 10.2 are **not the same**.

- $f_s$ and $f_o$ now represent these **activation functions**.

$$s_t = f_s\Big(W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss}\Big) \tag{10.20}$$

$$y_t = f_o\Big(W^{o} s_t + W_0^{o}\Big) \tag{10.21}$$

We'll insert these units into our diagram:



**Concept 25**
Just like in a traditional neural network, we apply our activation functions **element-wise**.

$$f\left(\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}\right) = \begin{bmatrix} f(a_1) \\ f(a_2) \\ f(a_3) \end{bmatrix}$$

### 10.2.4 Shape

We've the mechanics of our RNN sorted out. To do some quick book-keeping, we'll address the shapes of our various objects.

We'll keep our input, state, and output as vectors:

**Definition 26**
We define the dimensions of our input, output, and state as vectors:

$$\begin{matrix} \mathcal{X} = \mathbb{R}^{\ell} \\ \mathcal{S} = \mathbb{R}^{m} \\ \mathcal{Y} = \mathbb{R}^{n} \end{matrix} \quad \Longrightarrow \quad \begin{matrix} x_t & : & (\ell \times 1) \\ s_t & : & (m \times 1) \\ y_t & : & (n \times 1) \end{matrix}$$

Based on these vectors, we can derive our weight dimensions:

**Definition 27**
We define the dimensions of our RNN weights:

$$\begin{matrix} W^{sx} & : & (m \times \ell) \\ W^{ss} & : & (m \times m) \\ W^{ss}_0 & : & (m \times 1) \\ W^{o} & : & (n \times m) \\ W^{o}_0 & : & (n \times 1) \end{matrix}$$

### 10.2.5 Complete RNN

Now, we've done all the work we need to: We can define our RNN.

**Definition 28**

A **Recurrent Neural Network (RNN)** is a particular kind of **state machine** used as a neural network:

- We use a state machine so our network can remember and use past data, via the **state**.
- We call it "**recurrent**" because of our states. A state is created by the network to find the output, and then one timestep later, is **re-used** as a new input.

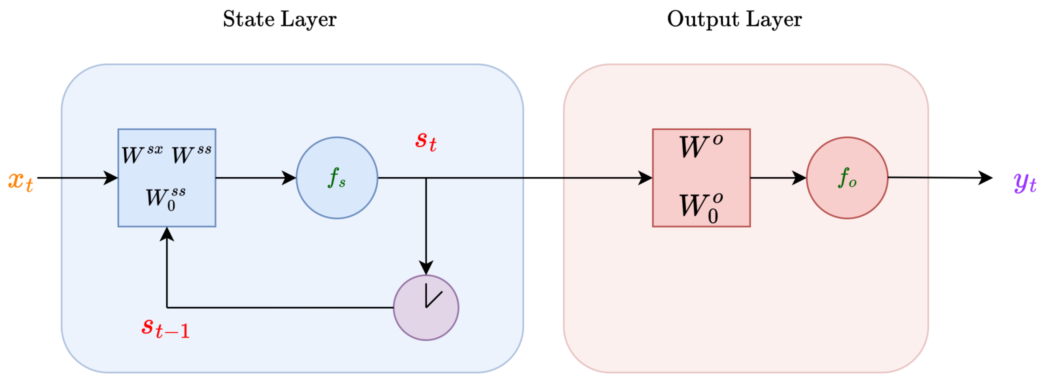
Our RNN manipulates input $x_t \in \mathbb{R}^\ell$, and state $s_t \in \mathbb{R}^m$, to create an output $y_t \in \mathbb{R}^n$.

Our state and output equations are given as:

$$s_t = f_s\left(W^{ss}s_{t-1} + W^{sx}x_t + W_0^{ss}\right)$$

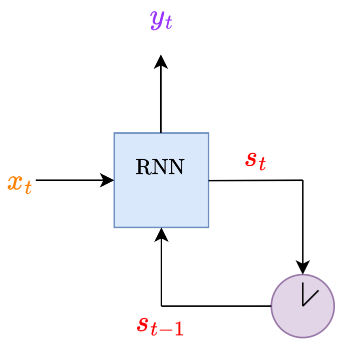
$$y_t = f_o\left(W^o s_t + W_0^o\right)$$

Where $f_s$ and $f_o$ are (typically non-linear) **activation functions**, and every weight $W$ is a **matrix** with the appropriate dimensions.



We can abstract away all the math with a simpler perspective:

Technically, this diagram works for any state machine.



We have the basic parts we care about: input, output, and state.
Our input and output are visible from outside, while the **state** is recycled within the system.

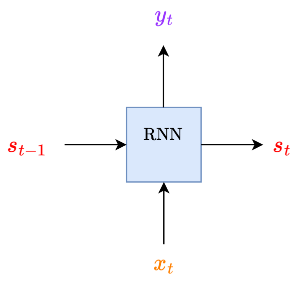
Take a moment to compare this model to the more complex one above: they're more similar than they seem.

### 10.2.6 RNN as a "network"

One issue we might have with the above diagram is it doesn't look very much like a **network**: at best, it seems like a very small network.

But the simplified diagram could inspire us: currently, the RNN "feeds into itself", using the state.

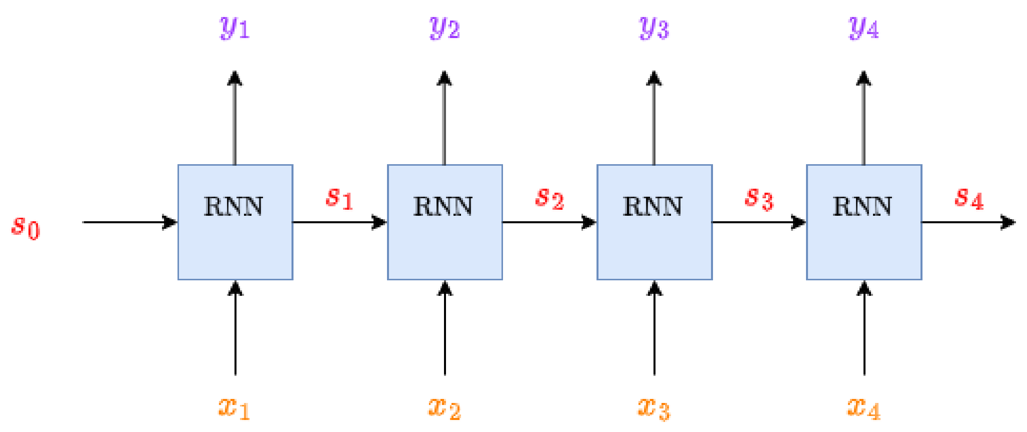
- In a different perspective, we could imagine that our first RNN unit is feeding into a second, **identical** unit.
- We'll show what we mean, by removing the clock:



We have an isolated, simple block.

Now, we can connect several of these in **series**: each one represents the input and output for the t^th timestep.

- The state $s_t$ feeds into the $(t+1)^{th}$ block.



Each of these RNNs is the same block: we've replaced our loop by copying the same RNN multiple times.

Now, it looks more like a network! This is fascinating: an RNN is like a network where we use the **same layer** over and over again!

- With the additional caveat that each layer has its own **input** and **output**.

**Concept 29**

One perspective on RNNs is to see them as a layered network, where each layer is the full RNN:

- Every layer has a **distinct** input and an output.
- Every layer is structurally **identical** (same weights, activation).

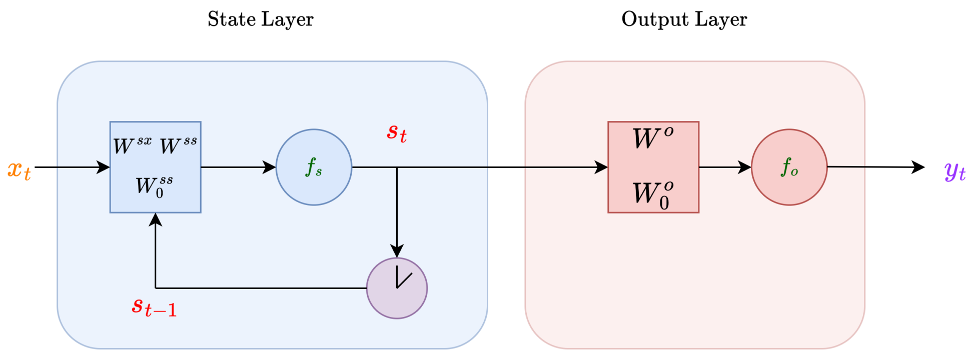
Of course, this version can be misleading: we don't actually have $n$ copies of our RNN, we have one copy that we're using repeatedly.

- However, we could think of the $x$-axis as representing "time": the same RNN reused at multiple times.

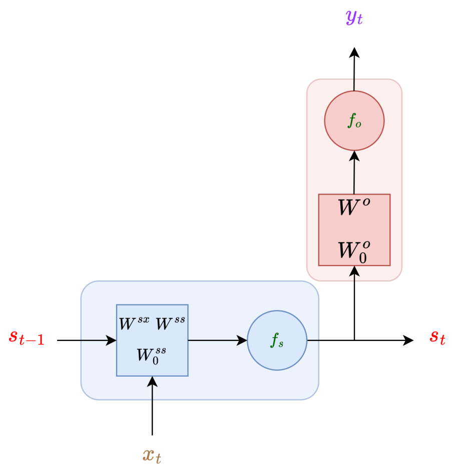
### 10.2.7 RNN fully unpacked

Now that we've introduced this perspective, let's use it for the "**unpacked**" version of our RNN, where we don't hide all of our inner functions. This will get a little messy.

First, we need to remove the clock:



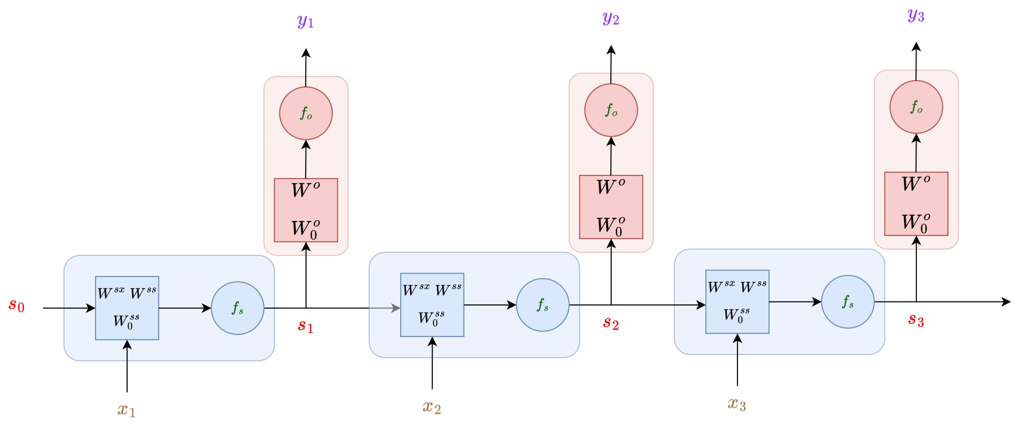
Our RNN, unpacked.



Our RNN, clock removed.

We had to rearrange things a bit to get the effect we wanted.

But now, we can stack these into a full "network":



Our RNN, unpacked.

This version looks complex, but it's just three copies of our previous model, side-by-side.

### 10.2.8 RNN Example 1 (Optional)

Let's try a very simple example: we'll do a **weighted average** of the last 3 inputs.

This is a linear operation, so we can ignore the activation functions. Thus, $f_s$ and $f_o$ are the identity function: $f_s(z) = f_o(z) = z$

$$s_t = W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss} \tag{10.22}$$

$$y_t = W^o s_t + W_0^o \tag{10.23}$$

Our input $x_t$ for each turn will be a single value, stored in a $(1 \times 1)$ matrix. Likewise, the "weighted average of 3 inputs" is a single value: another $(1 \times 1)$.

$$x_t = \begin{bmatrix} X_t \end{bmatrix} \qquad\qquad y_t = \begin{bmatrix} Y_t \end{bmatrix} \tag{10.24}$$

- Our state is based on the information we need to remember in order to compute the output.
- Thus, we'll store the **last three inputs**.

$$s_t = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} \tag{10.25}$$

**Concept 30**
Our **state vector** is typically chosen based on what is useful for finding the **output**.

So, our goal is to get the structure we gave $s_t$ above. We'll need to encode that in our equation.

We compute $s_t$ from $s_{t-1}$, $x_t$, and an offset.

- We're storing our past values, so we don't need an offset: $W_o^{ss} = 0$.

$$\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = W^{ss} s_{t-1} + W^{sx} x_t \quad \Longrightarrow \quad \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = W^{ss} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} + W^{sx} \begin{bmatrix} X_t \end{bmatrix} \tag{10.26}$$

Now, we can see that the information for $s_t$ is spread across both terms:

- The $s_{t-1}$ term contains $X_{t-1}$ and $X_{t-2}$
- The $x_t$ term contains $X_t$.

So, we want to end up with:

$$\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = \overbrace{\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix}}^{W^{ss}s_{t-1}} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ X_t \end{bmatrix}}^{W^{sx}x_t} \tag{10.27}$$

We can figure out our weight matrices $W^{ss}$ and $W^{sx}$, by comparing our input and output.

For starters, we showed above that the dimensions of $W^{sx}$ depend on $x_t$ and $s_t$.

$$\overbrace{\begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix} \tag{10.28}$$

To figure out $W_{ss}$, we go row-by-row, and figure out the correct values:

$$\overbrace{\begin{bmatrix} a & b & c \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix} \implies aX_{t-3} + bX_{t-2} + cX_{t-1} = X_{t-2} \tag{10.29}$$

We find $a = 0$, $b = 1$, $c = 0$. We can repeat this process for our other rows.

Once we do the rest, we find:

$$\overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix} \tag{10.30}$$

We follow the same logic for the other example:

$$\overbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}^{W^{sx}} \begin{bmatrix} X_t \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ X_t \end{bmatrix} \tag{10.31}$$

**Concept 31**

In order to derive our weight matrix, we can go **row-by-row**:

- For each row, we figure out which choice of **weights** gives the desired output.

Taken together, we get:

$$\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = \overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}^{W^{sx}} \begin{bmatrix} X_t \end{bmatrix} \tag{10.32}$$

Now we can compute the weighted average. We'll pick some arbitrary numbers: 50% of $X_t$, 30% of $X_{t-1}$, and 20% of $X_{t-2}$.

$$Y_t = 0.5X_t + 0.3X_{t-1} + 0.2X_{t-2} \tag{10.33}$$

- Again, we don't need an offset $W_0^o = 0$.

$$\begin{bmatrix} Y_t \end{bmatrix} = W^o s_t \quad \Longrightarrow \quad W^o \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} \tag{10.34}$$

We can, again, figure out our weight matrix $W^o$ based on the desired result.

$$\overbrace{\begin{bmatrix} 0.5 & 0.3 & 0.2 \end{bmatrix}}^{W^o} \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = 0.5X_t + 0.3X_{t-1} + 0.2X_{t-2}$$

**Remark (Optional) 32**

Our final RNN comes out to:

$$f_s(z) = f_o(z) = z$$

$$W^{ss} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad W^{sx} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad W_0^{ss} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$W^o = \begin{bmatrix} 0.5 & 0.2 & 0.2 \end{bmatrix} \qquad W_0^o = \begin{bmatrix} 0 \end{bmatrix}$$

### 10.2.9 RNN Example 2 (Optional)

Let's run through a more concrete example.

For simplicity, $f_s$ and $f_o$ are the identity function: $f_s(z) = f_o(z) = z$.

Remember that this is the activation function, not the complete function we use

$$s_t = W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss} \tag{10.35}$$

$$y_t = W^o s_t + W_0^o \tag{10.36}$$

- Each **input** is one number: the amount of money you earn every month.

$$x_t = \begin{bmatrix} x_t^E \end{bmatrix} \tag{10.37}$$

- Our **state** will be two numbers: the money you have in the bank, and the money you've invested.

$$s_t = \begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} \tag{10.38}$$

- Your **output** is your net worth: including the bank, and the invested money.

$$y_t = \begin{bmatrix} y_t^T \end{bmatrix} \tag{10.39}$$

Each of these could be a vector of any length, depending on the problem.

First, we want to compute $s_t$. Just like we mentioned in **Example 1**, we can go row-by-row to figure out the equation for $s_t$.

Let's starting with your first row, $s_t^B$: your "savings" money.

- The money in the bank $s_{t-1}^B$ makes no interest.
- 10% of our investing $s_{t-1}^I$ goes into the bank.
- 80% of our earned money $x_t^E$ goes into the bank.
- We lose \$6000 of savings every month.

We have a pretty terrible bank.

$$s_t^B = s_{t-1}^B + 0.2 s_{t-1}^I + 0.8 x_t^E - 6000 \tag{10.40}$$

With this, we can write in vector form:

$$\begin{bmatrix} s_t^B \end{bmatrix} = \begin{bmatrix} 1 & 0.2 \end{bmatrix} \begin{bmatrix} s_{t-1}^B \\ s_{t-1}^I \end{bmatrix} + \begin{bmatrix} 0.8 \end{bmatrix} \begin{bmatrix} x_t^E \end{bmatrix} - 6000 \tag{10.41}$$

**Concept 33**

Just like in other neural networks, the **weights** in an RNN indicate how an **input** variable (ex: $x_t^E$) affects an **output** variable (ex: $s_t^B$ in our linear system)

Now, we'll compute $s_t^I$, your investment money:

- The money in the bank $s_{t-1}^B$ is not invested.
- Our invested money $s_{t-1}^I$ grows by 1%.
- 20% of our earned money $x_t^E$ is invested.
- No money is added beyond that.

$$s_t^I = 0s_{t-1}^B + 1.01s_{t-1}^I + 0.2x_t^E + 0 \tag{10.42}$$

In vector form:

$$\begin{bmatrix} s_t^I \end{bmatrix} = \begin{bmatrix} 0 & 1.01 \end{bmatrix} \begin{bmatrix} s_{t-1}^B \\ s_{t-1}^I \end{bmatrix} + \begin{bmatrix} 0.2 \end{bmatrix} \begin{bmatrix} x_t^E \end{bmatrix} + 0 \tag{10.43}$$

Now, we can create our full representation of $s_t$:

$$s_t = W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss} \tag{10.44}$$

Which becomes:

$$\begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & 0.2 \\ 0 & 1.01 \end{bmatrix}}^{W^{ss}} \begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} + \overbrace{\begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}}^{W^{sx}} \begin{bmatrix} x_t^E \end{bmatrix} + \overbrace{\begin{bmatrix} -6000 \\ 0 \end{bmatrix}}^{W_0^{ss}} \tag{10.45}$$

We've finished the state equation of our RNN.

The output will be a bit simpler: it's just your total net worth.

$$y_t = W^o s_t + W_0^o \tag{10.46}$$

- The output is the sum of your bank savings, and your investments.

- There's nothing to add beyond that.

$$\begin{bmatrix} y_t^T \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & 1 \end{bmatrix}}^{W^o} \begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} + \overbrace{\begin{bmatrix} 0 \end{bmatrix}}^{W_0^o} \tag{10.47}$$

**Remark (Optional) 34**

Our final RNN comes out to:

$$f_s(z) = f_o(z) = z$$

$$W^{ss} = \begin{bmatrix} 1 & 0.2 \\ 0 & 1.01 \end{bmatrix} \qquad W^{sx} = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} \qquad W_0^{ss} = \begin{bmatrix} -6000 \\ 0 \end{bmatrix}$$

$$W^o = \begin{bmatrix} 1 & 1 \end{bmatrix} \qquad W_0^o = \begin{bmatrix} 0 \end{bmatrix}$$

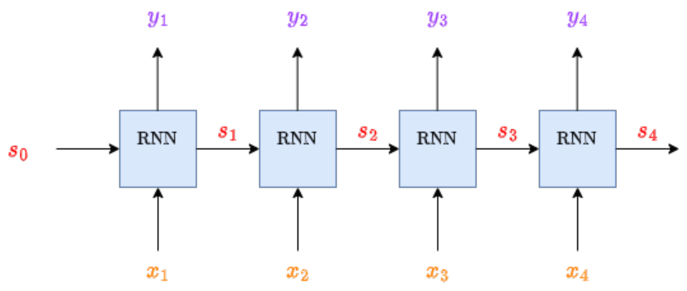
## 10.3 Sequence-to-sequence RNN

### 10.3.1 The sequence-to-sequence perspective

We've completely developed our RNN, a network designed out of a **state machine**.

- Our system takes one input, and produces one output, for each time step.

So far, we've been viewing each of these $x_t$ and $y_t$ terms separately. However, when we use our simplified perspective, things look a bit different:



In this view, we see a "sequence" of inputs $x_t$, and a "sequence" of outputs $y_t$.

This is why we might call the RNN problem a **sequence-to-sequence** problem.

**Concept 35**

Rather than seeing each $x_t$ term as an isolated input, we could consider our input the full **sequence**

$$x = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \end{bmatrix}$$

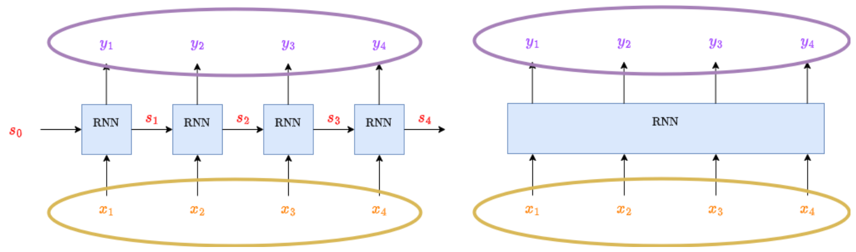
Our RNN takes the sequence $x$ and returns a paired, output sequence $y$:

$$y = \begin{bmatrix} y_1 & y_2 & y_3 & \cdots & y_n \end{bmatrix}$$

In this view, we can think of our RNN as a machine that takes in one sequence, and outputs a second, **equal-length** sequence.

- Notice that $x_t$ and $y_t$ can be **vectors**: $x$ and $y$ may need to be complete **matrices**, to store all these vectors.

In this view, we "lump together" all of our inputs and outputs as a single object:



If we ignore the internal states, our RNN just turns one sequence into another.

We sometimes call this process **transduction**.

**Definition 36**
We say can that our RNN **transduces** our **input** sequence into our **output** sequence.

Now, we can have **multiple sequences**: for example, we could have our input be, $x = [1, 2, 3, 4]$, or $x = [2, 4, 6, 7]$. Both are valid inputs.

- And in each of those sequences, you have **multiple timesteps**.

And each vector $x_t$ can have multiple elements! We'll ignore this last bit, for our sanity.

We'll need to distinguish between these two: difference sequences, versus different timesteps, within a sequence.

- For this purpose, we re-use data point notation $x^{(i)}$ that we developed in earlier chapters, Regression and Classification.

**Notation 37**
We use $x_t$ to distinguish between inputs in the **same sequence**.

- We'll represent a whole sequence with $x$.

We'll use $x^{(i)}$ to distinguish between **different sequences**.

- **Example:** $x_3^{(2)}$ is the 3rd timestep, of the 2nd sequence ("data point").

### 10.3.2 Sequence length

One important observation: we **need** an input $x_t$ in order to proceed to the next output.

- So, we only have as many outputs as we have inputs.

**Concept 38**

The **input** and **output** sequences to an RNN will be the **same length**.

What about two different input sequences?

- Our RNN is capable of taking in sequences of **any length**: if a sequence is longer, we just run our RNN for more timesteps.

So, each sequence our RNN receives can whatever length it wants to be: they don't have to match length.

**Concept 39**

An RNN can receive input sequences of **any length**.

In fact, the length can be different between **different input sequences**.

- So, sequence $x^{(1)}$ and sequence $x^{(2)}$ can be used by the **same RNN**, even if they have different lengths.

This means that each data point needs a separate variable for length: the length of $x^{(i)}$ is $n^{(i)}$

However, we need to be careful of the difference between these two ideas:

**Clarification 40**

The output sequence $y^{(i)}$ **must** have the **same length** as its input $x^{(i)}$: they're **paired** together.

However, different inputs ($x^{(i)}$ and $x^{(j)}$) can have **different lengths**.

This also means that inputs and outputs which are **not from the same pair** ($x^{(i)}$ and $y^{(j)}$) can have different lengths.

Note that this also means that different outputs can have different lengths, as well.

### 10.3.3 Training data

Just like any other NN, we usually are training our RNN for a **task**. What kind of task?

- The output of our RNN is a **sequence**: so, our goal will be to take the input sequence $x^{(i)}$, and give the *desired* sequence $y^{(i)}$.

**Concept 41**

Training our RNN is similar to our previous model training problems, like **regression**:

- Given a particular input sequence $x^{(i)}$, we want to teach our model to produce the output sequence $y^{(i)}$.

This is similar to regression, where we want to take an input vector, and get a real number as an output.

We want to use this model for **supervised** learning: we know the sequence we want to get as an output.

**Definition 42**

Our RNN is trained with a **training set** with $q$ data points:

$$\begin{bmatrix} (x^{(1)}, y^{(1)}) & (x^{(2)}, y^{(2)}) & \cdots & (x^{(q)}, y^{(q)}) \end{bmatrix}$$

Where, due to the behavior of RNNs (described above), we require:

- Each element $x^{(i)}$ or $y^{(j)}$ is a **sequence**.
- Elements $(x^{(i)}, y^{(i)})$ from the **same pair** must have the **same length** $n^{(i)}$.
- **Different pairs** may have **different lengths**.
  - Meaning, $n^{(i)}$ and $n^{(j)}$ are allowed to be different.

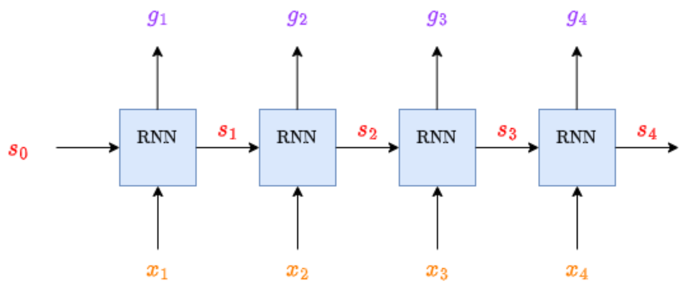
One important note: Now, we're using $y$ to represent the **desired** output, which is not necessarily the same as what our RNN actually gives us.

- We'll need separate notation for this.

**Notation 43**

From this point on, we'll use $y$ to indicate the **desired, correct** output, and $g$ to represent the **current RNN** output.

- So, our goal is to make $g$ and $y$ as **similar** as possible.



Very little changes: we just replace $y$ with $g$.

### 10.3.4 Training and Evaluation

The desired training output is $y^{(i)}$: we will **predict** it using the RNN output, $g^{(i)}$.

- For training and evaluation, we'll need a **loss function** to indicate how wrong our guess $g^{(i)}$ is.

This loss will be indicated by $\mathcal{L}_{seq}\left(g^{(i)}, y^{(i)}\right)$: typically, it will tell us how **different** our sequences are.

- The easiest way to compare two sequences is to compare them **element-wise**: compare the t$^{\text{th}}$ element of $y^{(i)}$ to the t$^{\text{th}}$ element of $g^{(i)}$.

**Definition 44**

We compute the **loss** $\mathcal{L}_{seq}$ of our sequence by adding up the loss $\mathcal{L}_{elt}$ for each **element** in our sequence.

$$\mathcal{L}_{seq}\left(g^{(i)}, y^{(i)}\right) = \sum_{t=1}^{n^{(i)}} \mathcal{L}_{elt}\left(g_t^{(i)}, y_t^{(i)}\right)$$

The choice of loss function $\mathcal{L}_{elt}$ depends on the data type of $y^{(t)}$.

- Note that for sequence $g^{(i)}$, we have $n^{(i)}$ timesteps.

**Example:** If our sequence is a series of numbers, we could take the **squared error** between the elements:

$$g^{(i)} = \begin{bmatrix} 1 & 4 & 5 \end{bmatrix} \qquad y^{(i)} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \qquad \mathcal{L}_{elt}(a, b) = (a - b)^2 \tag{10.48}$$

If compute the total loss, we get

$$\mathcal{L}_{seq}\left(g^{(i)}, y^{(i)}\right) = \sum_{t=1}^{3}\left(g_t^{(i)} - y_t^{(i)}\right)^2 = (1-1)^2 + (4-2)^2 + (5-3)^2$$

$$\mathcal{L}_{seq}\left(g^{(i)}, y^{(i)}\right) = 8 \tag{10.49}$$

Next, we'll compute the overall performance of our RNN. First, some notation:

**Notation 45**

We'll collectively represent all of our weights with a $W$:

$$W = \left(W^{sx}, W^{ss}, W^{o}, W_0^{ss}, W_0^{o}\right)$$

These are the **parameters** of our RNN – they're used for computing $g^{(i)}$. Meanwhile, $x^{(i)}$ is the **input**, so we find:

$$g^{(i)} = \text{RNN}\left(x^{(i)}; W\right)$$

Just like in other problems, we evaluate our model by taking the **average** of all of our losses:

**Definition 46**

The **objective function** $J(W)$ of our RNN is given by **averaging** the loss for each of our $q$ data points:

$$J(W) \quad = \quad \frac{1}{q}\sum_{i=1}^{q}\mathcal{L}_{seq}\left(g^{(i)}, y^{(i)}\right) \quad = \quad \frac{1}{q}\sum_{i=1}^{q}\mathcal{L}_{seq}\left(\overbrace{\text{RNN}\left(x^{(i)}, W\right)}^{g^{(i)}}, y^{(i)}\right)$$

## 10.3.5 Activation Functions

Lastly, we want to address our activation functions, $f_s$ and $f_o$.

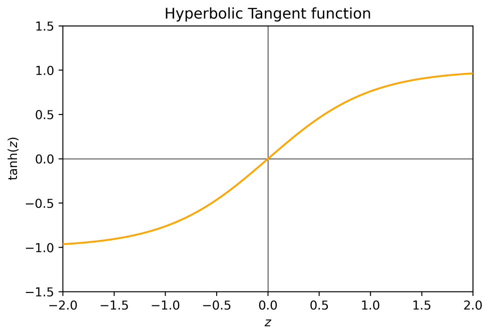
$f_s$ is used for our **state**: because our state doesn't directly compute our output, we tend to use the same $f_s$ for different problems:

**Concept 47**

Our most typical choice for $f_s$ is the **hyperbolic tangent** function tanh:

$$f_s(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

A function whose output ranges over $(-1, +1)$.



A reminder of how tanh appears: it looks relatively similar to sigmoid, with a different output range.

Meanwhile, $f_o$ directly allows us to compute the output, so our choice of $f_o$ depends on our problem and data type.

**Concept 48**

Just like in regular supervised learning, $f_o$ is chosen based on the problem at hand, considering:

- Data type
- Range
- Sensitivity

And other properties.

## 10.4 RNN as a language model

Human language is written/spoken **sequentially**, with each character/word/syllable coming in a particular **order**.

Thus, we might expect RNNs, a "sequential" model, to be suited for this task.

Not nearly as well as transformers, but we'll discuss that in the Transformers chapter.

The task in question is **predictive text**:

**Definition 49**
In the **predictive text** problem, you're given the "past" sequence of text, and you're supposed to predict "future" text.

**Example:** Autocorrect is a common application: based on the words you've typed so far, your phone will predict the most likely next word.

RNNs can be trained to accomplish this type of task.

### 10.4.1 Tokens

Our goal is to **correctly** predict the next word in a sentence, based on the previous words in a sentence.

First, we'll break up our sentence into a sequence of **elements**: these elements will be called **tokens**.

**Definition 50**
A **token** is a single **unit** of our text: this might be a single letter/**character**, or a single **word**.

Some systems, like chat-gpt, will use several characters as a single "token": this set of characters might not line up with each word!

**Example:** The following sentence contains 3 tokens if a token is a **word**, and 11 tokens if a token is a **character**:

$$\begin{matrix} 1 & 2 & 3 \\ \text{I} & \text{love} & \text{dogs} \end{matrix} \tag{10.50}$$

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \text{I} & \_ & l & o & v & e & \_ & d & o & g & s \end{matrix} \tag{10.51}$$

We can represent our sentence $c$ as sequence of tokens $c_i$:

$$c = \begin{bmatrix} c_1 & c_2 & \cdots & c_k \end{bmatrix} \tag{10.52}$$

### 10.4.2 Predicting tokens

We want our RNN to predict **future** tokens in the sentence, based on **past** tokens.

- Let's start by feeding in our first token:

$$\text{RNN}\left(\begin{bmatrix} c_1 \end{bmatrix}\right) = \begin{bmatrix} G_2 \end{bmatrix} \tag{10.53}$$

We want our RNN to predict the **next** character, so the **output** will be our prediction for $c_2$: we'll call it $G_2$.

**Notation 51**

Our **prediction** for the $n^{\text{th}}$ token, $c_n$, is $G_n$.

– Thus, $G_n$ is the token we consider **most likely**for $c_n$.

Alternatively, $G_n$ could be a vector, giving the **probability** for each possible token that $c_n$ could be.

This would be useful for evaluating our model: how sure was it of the right answer?

- Let's try our second token:

$$\text{RNN}\left(\begin{bmatrix} c_1 & c_2 \end{bmatrix}\right) = \begin{bmatrix} G_2 & G_3 \end{bmatrix} \tag{10.54}$$

Note that our model gets to see the correct $c_2$, **after** making its prediction, $G_2$.

- That means that our RNN has only seen $c_1$ when it predicts $G_2$: it doesn't know what the correct character, $c_2$, is yet.
- Meanwhile, $G_3$ is generated with knowledge of $c_1$ and $c_2$: the RNN knows what the first two characters are.

**Concept 52**

When our RNN **guesses** the $t^{\text{th}}$ token, $G_t$, it can only see the **first $t-1$ inputs**:

$$\begin{bmatrix} c_1 & c_2 & \cdots & c_{t-1} \end{bmatrix} \overset{\text{RNN}}{\Longrightarrow} G_t$$

Information stored in $c_t$ or $c_{t+1}$, for example, has **no effect** on $G_t$.

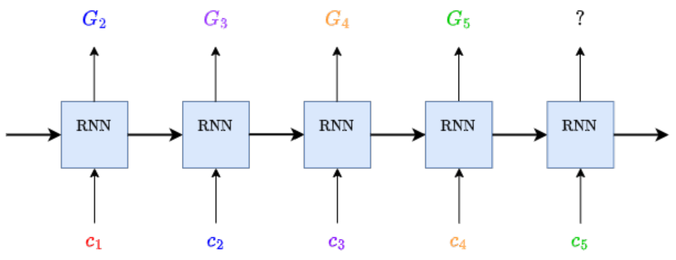
- Otherwise, our model could cheat, and predict by looking at the answer!

In this way, we can supply our entire input, and get a full vector of predictions:

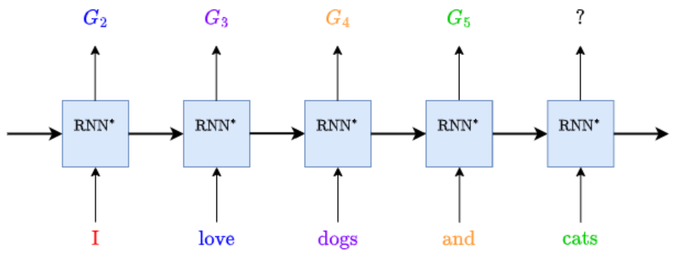
$$\text{RNN}\left(\begin{bmatrix} c_1 & c_2 & \cdots & c_{k-1} & c_k \end{bmatrix}\right) = \begin{bmatrix} G_2 & \cdots & G_{k-1} & G_k & ? \end{bmatrix} \tag{10.55}$$

Let's show this in our simplified RNN diagram.

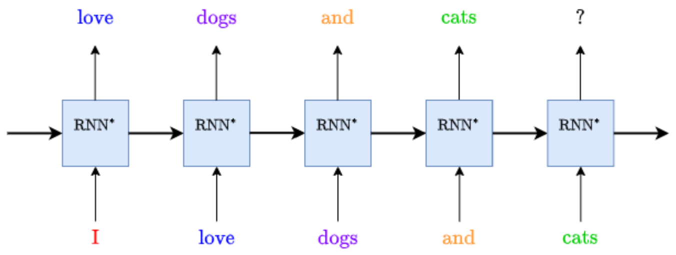
Remember, the arrows left-to-right are states: they represent all of the previous words in the sentence.



Here's an example sentence we could try to predict:



And now, here's an ideal case, where our RNN performs perfectly (we'll call it RNN*):



Our RNN predicts the right word, right before it appears.

### 10.4.3 Start token and end token

There are two problems we'd like to fix.

- We'd like to try predicting the first character, $c_1$, rather than **skipping** it.
- If we start with $c_2$, there are only $k - 1$ characters we need to predict.

  - We have $k$ inputs, and therefore $k$ outputs: we have **one more output** than we need.

Our first solution is to add a special "**START**" token to the beginning of our input:

$$x = \begin{bmatrix} \textbf{START} & c_1 & c_2 & \cdots & c_k \end{bmatrix} \tag{10.56}$$

What does this do for us?

- During our first timestep, our RNN has nothing other than the **START** token, so it's able to spent that timestep **predicting** $c_1$.

$$\text{RNN}\left( \begin{bmatrix} \textbf{START} \end{bmatrix} \right) = \begin{bmatrix} g_1 \end{bmatrix} \tag{10.57}$$

Now, our output starts with $G_1$.

$$\text{RNN}\left( \begin{bmatrix} \textbf{START} & c_1 & c_2 & \cdots & c_{k-1} & c_k \end{bmatrix} \right) = \begin{bmatrix} G_1 & G_2 & \cdots & G_{k-1} & G_k & ? \end{bmatrix}$$

**Definition 53**

The **input** to our **sentence-prediction RNN** starts with the special **START** token, followed by all of the tokens in our sentence $c$.

$$x = \begin{bmatrix} \textbf{START} & c_1 & c_2 & \cdots & c_k \end{bmatrix}$$

When our RNN receives this **START** token, it has an opportunity to predict the **first word** in the sentence, with no context.

This hasn't fixed our other problem, though: now we have $k+1$ slots, but only $k$ outputs we need to predict.

We'll solve that by adding an **END** character at the end of the output.

$$y = \begin{bmatrix} c_1 & c_2 & \cdots & c_k & \textbf{END} \end{bmatrix} \tag{10.58}$$

- Now, we have a desired final output: we want our RNN to predict when the sentence ends, and return **END**.

**Definition 54**

The **optimal output** of our **sentence-prediction RNN** starts with all the tokens in our sentence $c$, followed by the special **END** token.

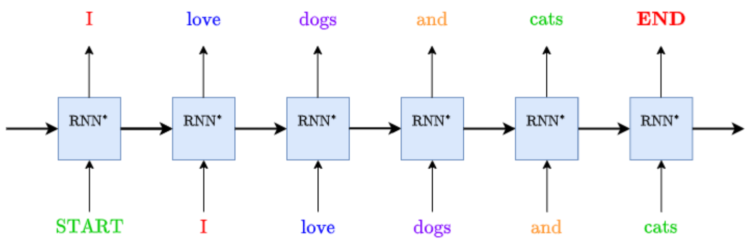
$$y = \begin{bmatrix} c_1 & c_2 & \cdots & c_k & \textbf{END} \end{bmatrix}$$

Thus, we've fully formed our problem statement:

**Concept 55**

The goal of our sentence-prediction RNN is to **replicate the sentence** $c$, token-by-token, with two caveats:

- The input starts with the special **START** token, to give your model one timestep to **predict the first token** $c_1$.
- The output should terminate with the special **END** token: your model should **predict when the sentence ends**.



Here's what our system looks like, now that we've added the START and END tokens.

Let's review what happens at each timestep.

- On the bottom, we **input** one word in the sentence.
- On the top, we **output** the predicted next word.
- After an input, we update our **state**, to include new info. This is **used** by the RNN in the next timestep (left-to-right).

### 10.4.4 Why we might use RNNs for language

The general reason we decided to try using RNNs for language is pretty basic:

- "RNNs **output sequences**. Text is a **sequence of tokens**".

But there's a bit more to it than that: RNNs allow our model to remember the **structure** of our sentence, using our **state** $s_t$.

- The newest token $c_t$ might be related to one that we saw **earlier**: for example, 5 tokens ago.
- Our state can treat words that are **closer**, differently from words which are **further away**.

This way, our **state** allows us to keep track of sentence structure: grammar, the meaning of each word, tone, etc.

**Concept 56**

When we use an RNN as a **language model**, we're hoping that it can distinguish between "**nearby**" words, and "**farther away**" words.

- Words which are closer/further can contribute differently to the sentence: this gives us **context**.

Convolution has a similar effect, but in a more **discrete** way: in convolution, we have a window of a **fixed size**.

- If $n$ is our window size, but two tokens are $n + 1$ units apart, then they won't show up together in convolution.

Meanwhile, RNNs are more flexible:

**Concept 57**

Depending on how your **RNN state** works, it could preserve/accumulate data over **longer, non-fixed** distances than **convolution**.

- **Example:** A **running average** could factor in newer information, with older information, without completely "forgetting" that old information.

### 10.4.5 Why RNNs don't work (well) for language

Unfortunately, RNNs tend not to work well enough.

- Their state $s_t$ can only store a **limited** amount of data.
- So, the longer the RNN runs, the more it forgets.

**Concept 58**

Over time, an RNN will "**forget**" information it learned in **earlier** timesteps.

- It's replaced by **newer** data.

Language requires this sort of longer-term memory.

- The longer the text prompt becomes, the worse the RNN performs.

In the next chapter, we'll design a model to overcome this problem: the **transformer**.

## 10.5 Terms

- Timestep $t$(Review)
- State $s_t$
- Input $x_t$
- Transition function $f_s$
- Output $y_t$
- Output function $f_o$
- State Machine
- Finite State Machine
- State Transition Diagram
- Linearity
- Time-Invariance
- Linear Time-Invariant System (LTI)
- Recurrent Neural Network (RNN)
- Weights $W^{ss}$, $W^{sx}$, $W_0^{ss}$, $W^o$, $W_0^o$
- Transduction
- $x_t$ notation
- $x^{(i)}$ notation (Review)
- Sequence loss $\mathcal{L}_{seq}$
- Hyperbolic Tangent Function tanh (Review)
- Predictive Text
- Token
- Start Token
- End Token