Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2024

Contents

APPENDIX C

Recurrent Neural Networks

This chapter was originally placed immediately after the Convolutional Neural Networks Chapter.

- Additionally, this chapter relies on having read the State Machines section of the Markov Decision Process (MDP 0).

C.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.

C.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.

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

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.

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.

C.1 State Machines

This section is identical to the notes from MDP 0. If you have already read that section, skip to C.2.

C.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)**.

C.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 **non-linear** 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 3

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.

C.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 4

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 5

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

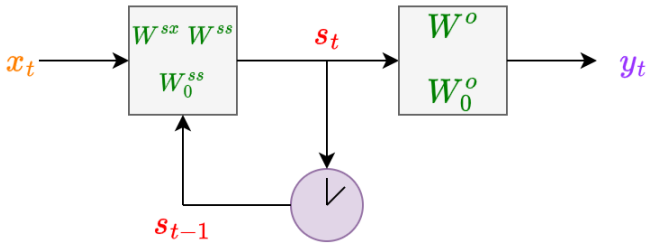
- W^{ss} is combined with s_{t-1}, to compute s_t.
- W^{sx} is combined with x_t, to compute s_t.
- W_0^{ss} is the offset term that computes s_t.

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

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

- 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.

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



C.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:

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

$$y_t = g\left(W^o s_t + W_0^o\right) \tag{C.2}$$

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 6

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.

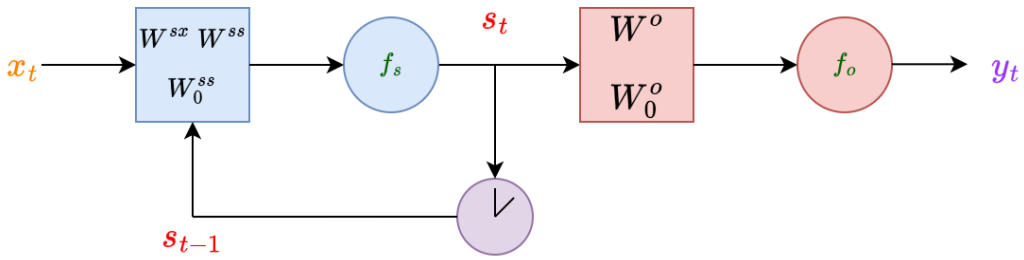
So, we'll switch conventions: f_s and f_o in sections 10.1 and 10.2 are **not the same**.

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

$$s_t = f_s\left(W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss}\right) \tag{C.3}$$

$$y_t = f_o\left(W^o s_t + W_0^o\right) \tag{C.4}$$

We'll insert these units into our diagram:



Concept 7

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

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

C.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 8

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

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

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

Definition 9

We define the dimensions of our RNN weights:

$$\begin{aligned} W^{sx} &: (m \times \ell) \\ W^{ss} &: (m \times m) \\ W_0^{ss} &: (m \times 1) \\ W^{o} &: (n \times m) \\ W_0^{o} &: (n \times 1) \end{aligned}$$

C.2.5 Complete RNN

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

Definition 10

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.

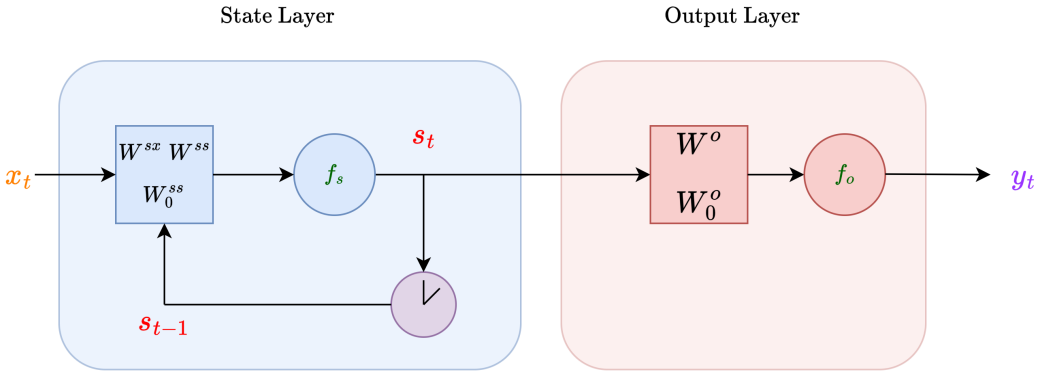
Our RNN manipulates input $x_t \in \mathbb{R}^\ell$, and state $s_t \in \mathbb{R}^m$, to create an output $y_t \in \mathbb{R}^n$.

Our state and output equations are given as:

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

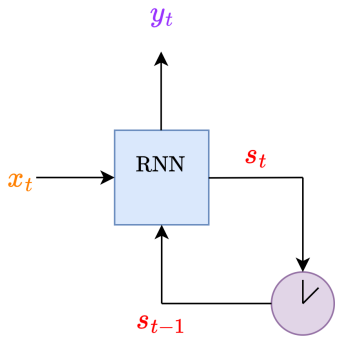
$$y_t = f_o\left(W^o s_t + W_0^o\right)$$

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.

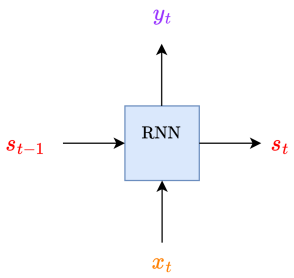
Take a moment to compare this model to the more complex one above: they're more similar than they seem.

C.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.

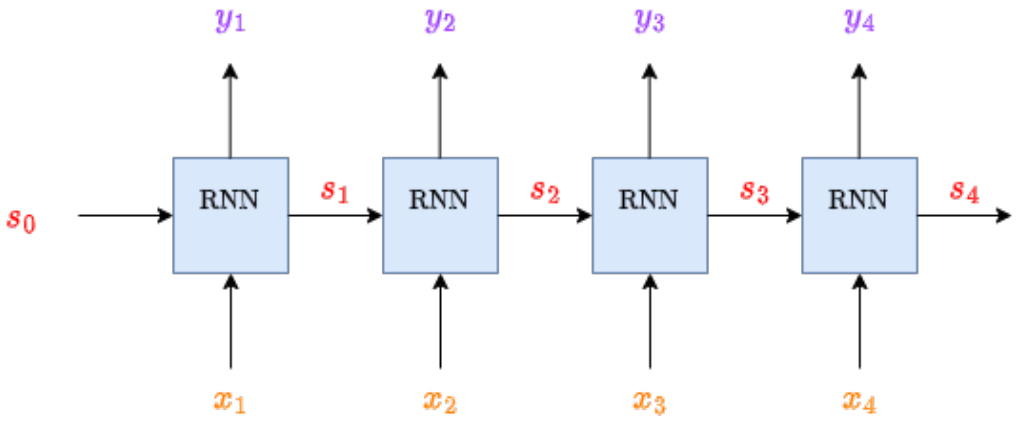
- In a different perspective, we could imagine that our first RNN unit is feeding into a second, **identical** unit.
- We'll show what we mean, by removing the clock:



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 11

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).

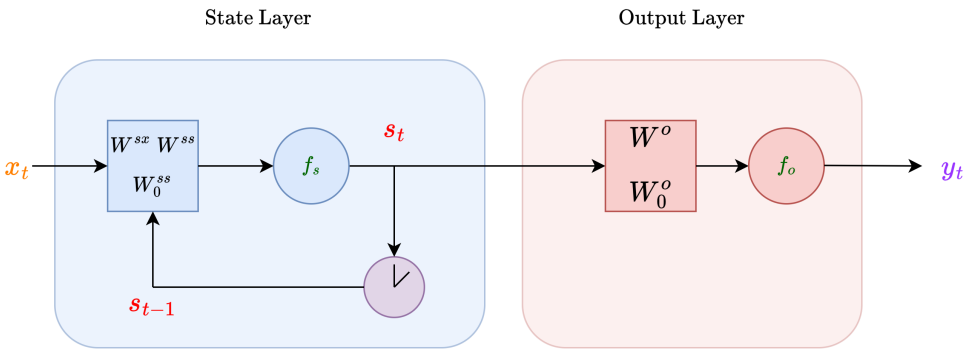
Of course, this version can be misleading: we don't actually have n copies of our RNN, we have one copy that we're using repeatedly.

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

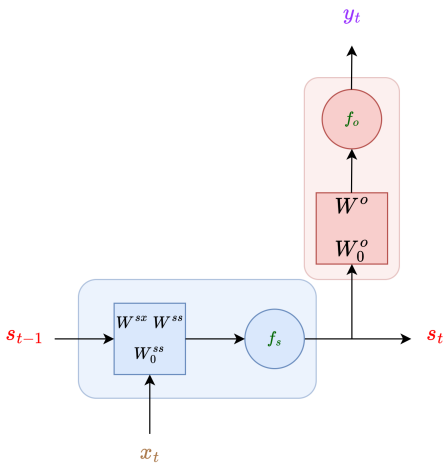
C.2.7 RNN fully unpacked

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:



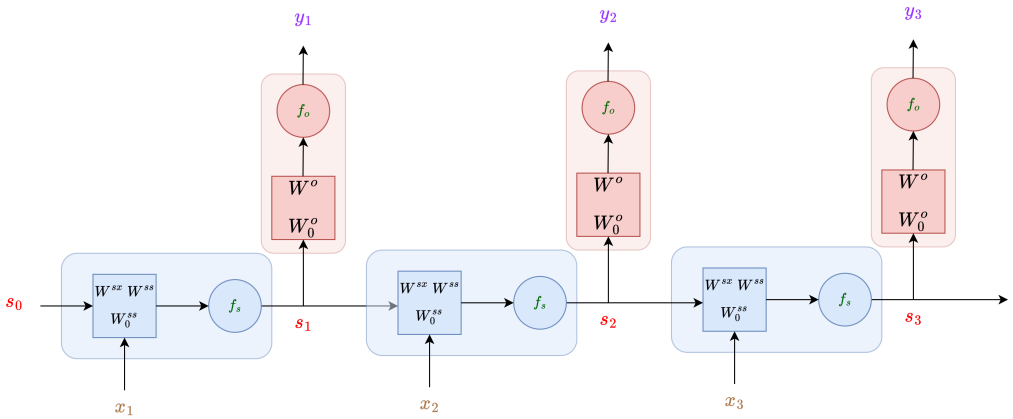
Our RNN, unpacked.



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.

C.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: $f_s(z) = f_o(z) = z$

$$s_t = W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss} \tag{C.5}$$

$$y_t = W^{o} s_t + W_0^{o} \tag{C.6}$$

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

$$x_t = \begin{bmatrix} X_t \end{bmatrix} \qquad y_t = \begin{bmatrix} Y_t \end{bmatrix} \tag{C.7}$$

- 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**.

$$s_t = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} \tag{C.8}$$

> **Concept 12**
> 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: $W_o^{ss} = 0$.

$$\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = W^{ss} s_{t-1} + W^{sx} x_t \quad \Longrightarrow \quad \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = W^{ss} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} + W^{sx} \begin{bmatrix} X_t \end{bmatrix} \tag{C.9}$$

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:

$$\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = \overbrace{\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix}}^{W^{ss}s_{t-1}} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ X_t \end{bmatrix}}^{W^{sx}x_t} \tag{C.10}$$

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.

$$\overbrace{\begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix} \tag{C.11}$$

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

$$\overbrace{\begin{bmatrix} a & b & c \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix} \implies aX_{t-3} + bX_{t-2} + cX_{t-1} = X_{t-2} \tag{C.12}$$

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

Once we do the rest, we find:

$$\overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} = \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ 0 \end{bmatrix} \tag{C.13}$$

We follow the same logic for the other example:

$$\overbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}^{W^{sx}} \begin{bmatrix} X_t \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ X_t \end{bmatrix} \tag{C.14}$$

Concept 13

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:

$$\begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} = \overbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}^{W^{ss}} \begin{bmatrix} X_{t-3} \\ X_{t-2} \\ X_{t-1} \end{bmatrix} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}^{W^{sx}} \begin{bmatrix} X_t \end{bmatrix} \tag{C.15}$$

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}.

$$Y_t = 0.5X_t + 0.3X_{t-1} + 0.2X_{t-2} \tag{C.16}$$

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

$$\begin{bmatrix} Y_t \end{bmatrix} = W^o s_t \quad \Longrightarrow \quad W^o \begin{bmatrix} X_{t-2} \\ X_{t-1} \\ X_t \end{bmatrix} \tag{C.17}$$

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) 14

Our final RNN comes out to:

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

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

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

C.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

$$s_t = W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss} \tag{C.18}$$

$$y_t = W^{o} s_t + W_0^{o} \tag{C.19}$$

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

$$x_t = \begin{bmatrix} x_t^E \end{bmatrix} \tag{C.20}$$

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

$$s_t = \begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} \tag{C.21}$$

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

$$y_t = \begin{bmatrix} y_t^T \end{bmatrix} \tag{C.22}$$

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.

$$s_t^B = s_{t-1}^B + 0.2 s_{t-1}^I + 0.8 x_t^E - 6000 \tag{C.23}$$

With this, we can write in vector form:

$$\begin{bmatrix} s_t^B \end{bmatrix} = \begin{bmatrix} 1 & 0.2 \end{bmatrix} \begin{bmatrix} s_{t-1}^B \\ s_{t-1}^I \end{bmatrix} + \begin{bmatrix} 0.8 \end{bmatrix} \begin{bmatrix} x_t^E \end{bmatrix} - 6000 \tag{C.24}$$

Concept 15

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.

$$s_t^I = 0 s_{t-1}^B + 1.01 s_{t-1}^I + 0.2 x_t^E + 0 \tag{C.25}$$

In vector form:

$$\begin{bmatrix} s_t^I \end{bmatrix} = \begin{bmatrix} 0 & 1.01 \end{bmatrix} \begin{bmatrix} s_{t-1}^B \\ s_{t-1}^I \end{bmatrix} + \begin{bmatrix} 0.2 \end{bmatrix} \begin{bmatrix} x_t^E \end{bmatrix} + 0 \tag{C.26}$$

Now, we can create our full representation of s_t:

$$s_t = W^{ss} s_{t-1} + W^{sx} x_t + W_0^{ss} \tag{C.27}$$

Which becomes:

$$\begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} = \overbrace{\begin{bmatrix} 1 & 0.2 \\ 0 & 1.01 \end{bmatrix}}^{W^{ss}} \begin{bmatrix} s_t^B \\ s_t^I \end{bmatrix} + \overbrace{\begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}}^{W^{sx}} \begin{bmatrix} x_t^E \end{bmatrix} + \overbrace{\begin{bmatrix} -6000 \\ 0 \end{bmatrix}}^{W_0^{ss}} \tag{C.28}$$

We've finished the state equation of our RNN.

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

$$y_t = W^o s_t + W_0^o \tag{C.29}$$

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

- There's nothing to add beyond that.

$$\left[y_t^T\right] = \overbrace{\begin{bmatrix}1 & 1\end{bmatrix}}^{W^o}\begin{bmatrix}s_t^B \\ s_t^I\end{bmatrix} + \overbrace{\begin{bmatrix}0\end{bmatrix}}^{W_0^o} \tag{C.30}$$

Remark (Optional) 16

Our final RNN comes out to:

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

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

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

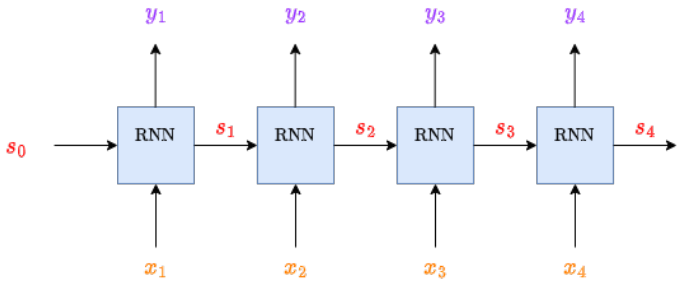
C.3 Sequence-to-sequence RNN

C.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 17

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

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

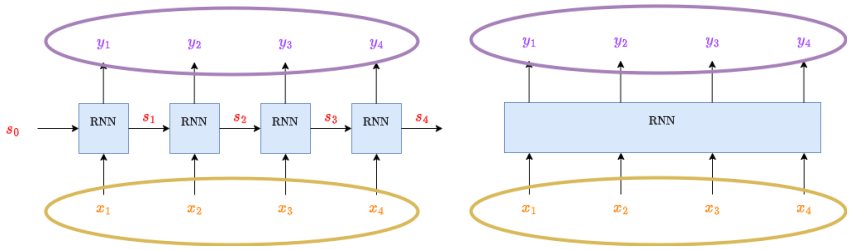
Our RNN takes the sequence x and returns a paired, output sequence y:

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

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 18

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 19

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").

C.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 20

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 21

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 22

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.

C.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 23

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 24

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

$$\left[\quad (x^{(1)}, y^{(1)}) \qquad (x^{(2)}, y^{(2)}) \qquad \cdots \qquad (x^{(q)}, y^{(q)}) \quad \right]$$

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.

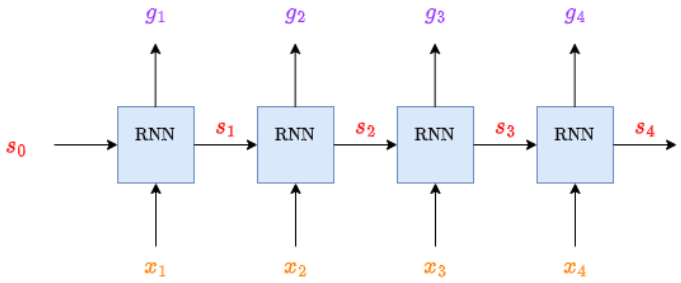
One important note: Now, we're using y to represent the **desired** output, which is not necessarily the same as what our RNN actually gives us.

- We'll need separate notation for this.

Notation 25

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.

C.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^{th} element of $y^{(i)}$ to the t^{th} element of $g^{(i)}$.

Definition 26

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:

$$g^{(i)} = \begin{bmatrix} 1 & 4 & 5 \end{bmatrix} \qquad y^{(i)} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \qquad \mathcal{L}_{elt}(a, b) = (a - b)^2 \tag{C.31}$$

If compute the total loss, we get

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

$$\mathcal{L}_{seq}\left(g^{(i)}, y^{(i)}\right) = 8 \tag{C.32}$$

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

Notation 27

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 28

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

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

C.3.5 Activation Functions

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

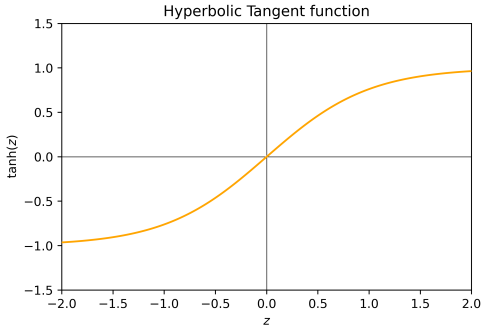
f_s is used for our **state**: because our state doesn't directly compute our output, we tend to use the same f_s for different problems:

Concept 29

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 30

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

- Data type
- Range
- Sensitivity

And other properties.

C.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 31

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.

C.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 32

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

Some systems, like chatgpt, 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**:

$$\begin{array}{ccc} 1 & 2 & 3 \\ \mathrm{I} & \mathrm{love} & \mathrm{dogs} \end{array} \tag{C.33}$$

$$\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \mathrm{I} & \_ & \mathrm{l} & \mathrm{o} & \mathrm{v} & \mathrm{e} & \_ & \mathrm{d} & \mathrm{o} & \mathrm{g} & \mathrm{s} \end{array} \tag{C.34}$$

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

$$c = \begin{bmatrix} c_1 & c_2 & \cdots & c_k \end{bmatrix} \tag{C.35}$$

C.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:

$$\text{RNN}\left(\begin{bmatrix} c_1 \end{bmatrix}\right) = \begin{bmatrix} G_2 \end{bmatrix} \tag{C.36}$$

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 33

Our **prediction** for the n^{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:

$$\text{RNN}\left(\begin{bmatrix} c_1 & c_2 \end{bmatrix}\right) = \begin{bmatrix} G_2 & G_3 \end{bmatrix} \tag{C.37}$$

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 34

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

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

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

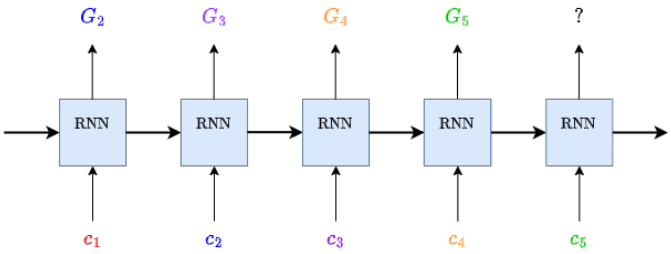
- Otherwise, our model could cheat, and predict by looking at the answer!

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

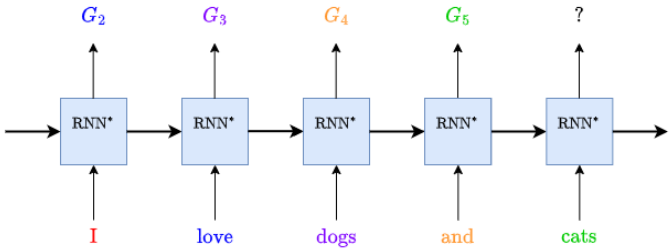
$$\text{RNN}\left(\begin{bmatrix} c_1 & c_2 & \cdots & c_{k-1} & c_k \end{bmatrix}\right) = \begin{bmatrix} G_2 & \cdots & G_{k-1} & G_k & ? \end{bmatrix} \tag{C.38}$$

Let's show this in our simplified RNN diagram.

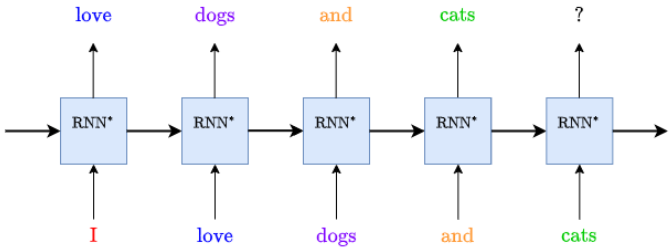
Remember, the arrows left-to-right are states: they represent all of the previous words in the sentence.



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.

C.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:

$$x = \begin{bmatrix} \textbf{START} & c_1 & c_2 & \cdots & c_k \end{bmatrix} \tag{C.39}$$

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.

$$\text{RNN}\left(\begin{bmatrix} \textbf{START} \end{bmatrix} \right) = \begin{bmatrix} g_1 \end{bmatrix} \tag{C.40}$$

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 35

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.

$$y = \begin{bmatrix} c_1 & c_2 & \cdots & c_k & \textbf{END} \end{bmatrix} \tag{C.41}$$

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

Definition 36

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

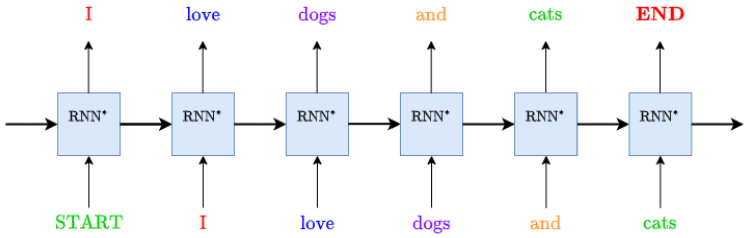
$$y = \begin{bmatrix} c_1 & c_2 & \cdots & c_k & \textbf{END} \end{bmatrix}$$

Thus, we've fully formed our problem statement:

Concept 37

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).

C.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 38

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 39

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.

C.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 40

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**.

C.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