

Lecture 6: recurrent neural networks, LSTMs, attention

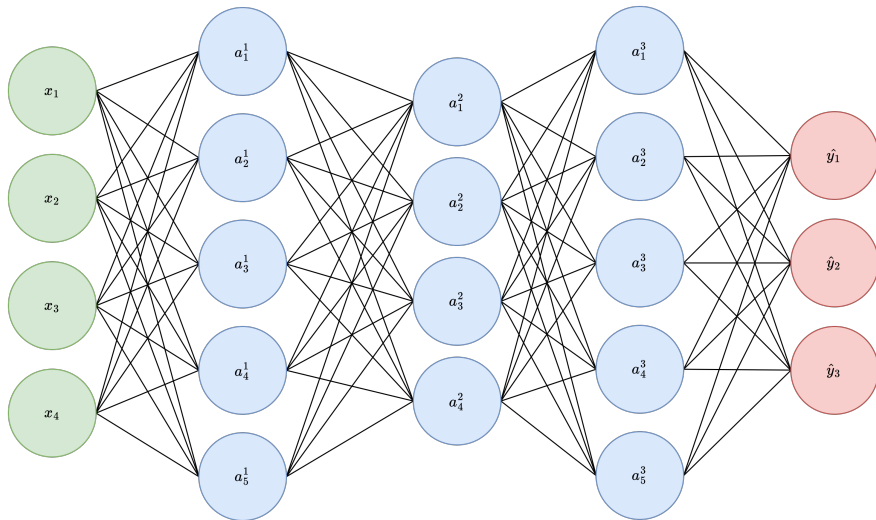
Jikael Gagnon (Based on slides by Sheheryar Parvaz and Shidan Javaheri)

McGill Artificial Intelligence Society

Winter 2026

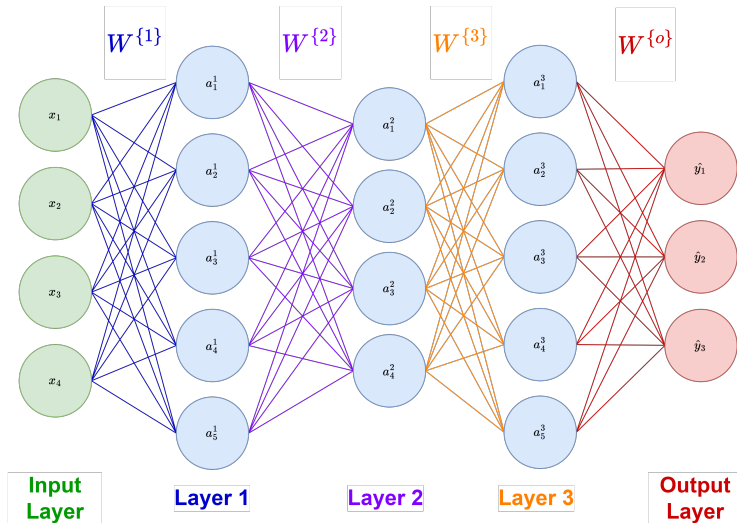


Neural Networks

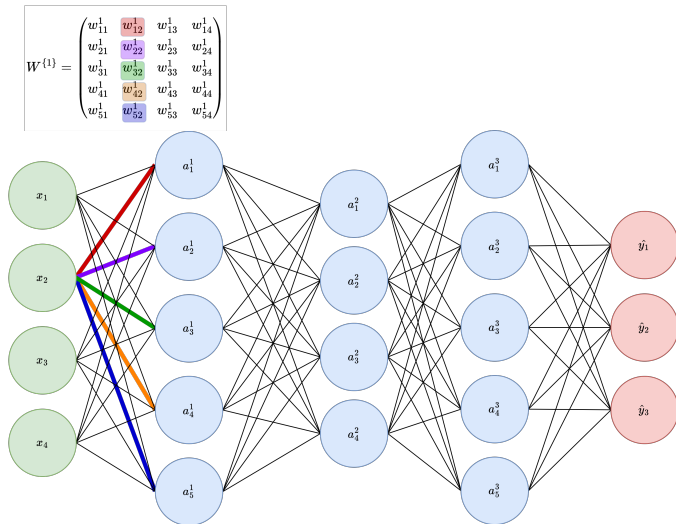


Warning: The notation used in the following diagrams is not the same as in lecture 4. For example, the roles of z and a have been interchanged. z now represents the *pre-activation* and a is the activation. Also, we are multiplying our weights from the left instead of the right.

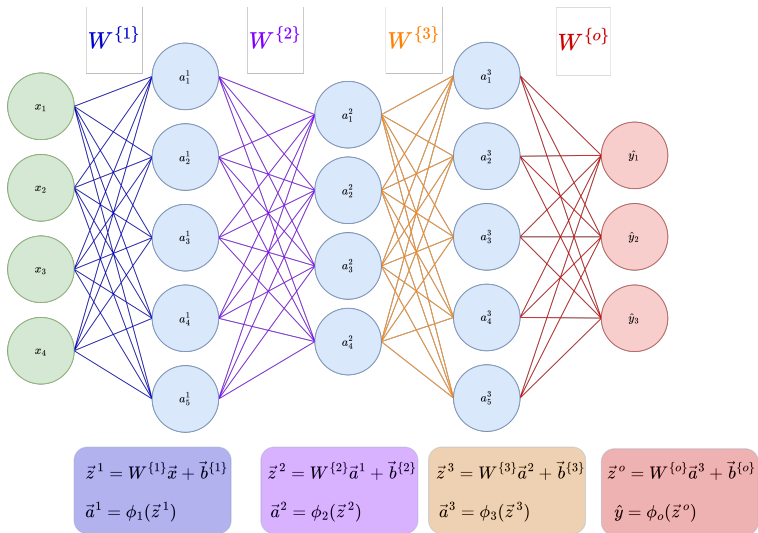
Neural Networks



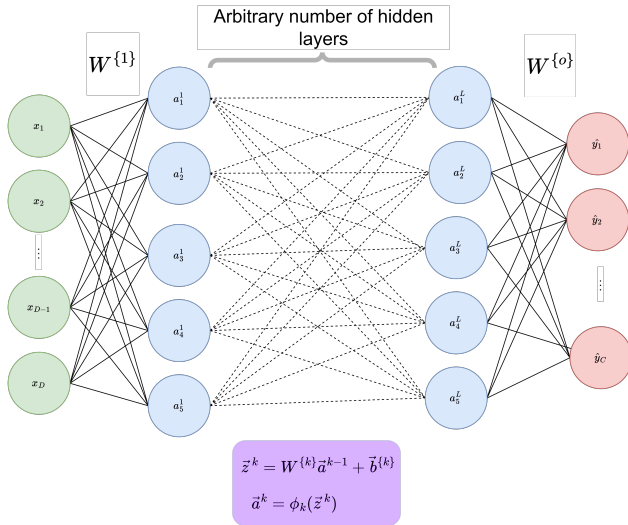
Neural Networks



Neural Networks



Neural Networks



Motivating RNNs

So far in the course we have (implicitly) made two assumptions about our data:

Motivating RNNs

So far in the course we have (implicitly) made two assumptions about our data:

1. Each input has the same number of features

Motivating RNNs

So far in the course we have (implicitly) made two assumptions about our data:

1. Each input has the same number of features
 - ▶ Eg. In CNNs, we assumed all input images have the same dimensions (eg. 64×64). It's impossible to provide images of any other size.

Motivating RNNs

So far in the course we have (implicitly) made two assumptions about our data:

1. Each input has the same number of features
 - ▶ Eg. In CNNs, we assumed all input images have the same dimensions (eg. 64×64). It's impossible to provide images of any other size.
2. The order of the features in our input are not meaningful (except in CNNs)

Motivating RNNs

So far in the course we have (implicitly) made two assumptions about our data:

1. Each input has the same number of features
 - ▶ Eg. In CNNs, we assumed all input images have the same dimensions (eg. 64×64). It's impossible to provide images of any other size.
2. The order of the features in our input are not meaningful (except in CNNs)
 - ▶ Eg. If we predict house price based on income and square footage, it doesn't matter whether we choose x_1 to be income or square footage (our initial ordering of columns in our dataset is irrelevant)

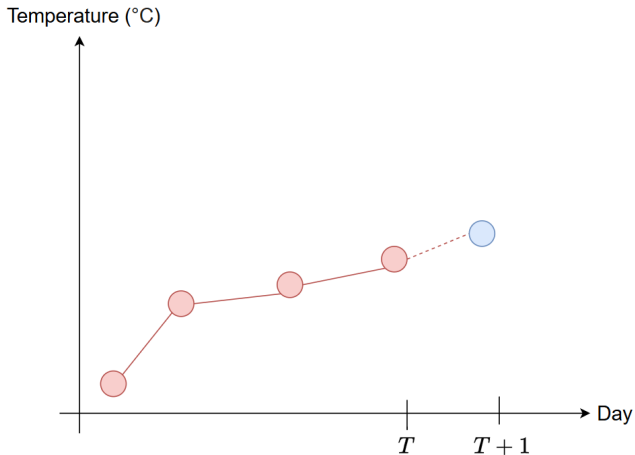
Motivating RNNs

Sometimes, these are bad assumptions to make. In particular, they are bad when we're dealing with *sequential data*.

Motivating RNNs

Example: Weather data

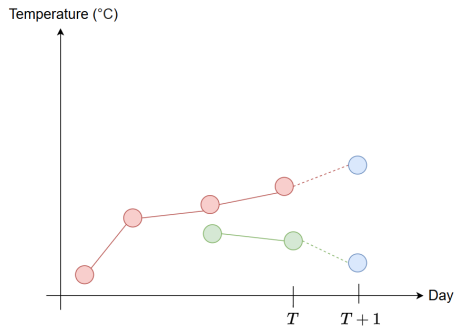
Suppose we want a model that can predict the temperature on day $t + 1$ given the temperatures on days $1, 2, \dots, t$.



Motivating RNNs

Issue 1: Sequence length

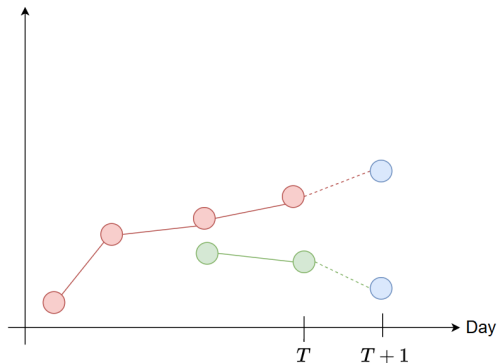
For this model to be useful, it should work for sequences of temperatures of *any* length. It would be pretty annoying if I told you “I have a model that can predict the weather tomorrow but only if you have exactly 37 days worth of data”...



Motivating RNNs

Issue 2: Order should be meaningful

Temperature (°C)

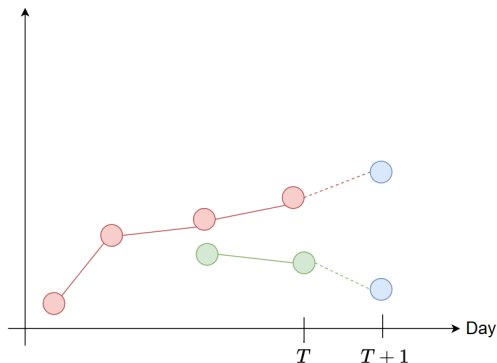


Our inputs are of the form $\vec{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \dots, x_T^{(i)}]$, where $x_t^{(i)}$ is the temperature on day t .

Motivating RNNs

Issue 2: Order should be meaningful

Temperature (°C)

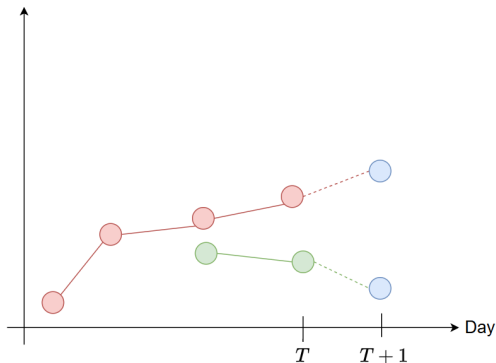


Our inputs are of the form $\vec{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \dots, x_T^{(i)}]$, where $x_t^{(i)}$ is the temperature on day t . Intuitively, the specific order of our inputs should matter in our prediction. If temperatures have been increasing lately, we might expect them to keep increasing (and vice versa).

Motivating RNNs

Issue 2: Order should be meaningful

Temperature (°C)



Our inputs are of the form $\vec{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}, \dots, x_T^{(i)}]$, where $x_t^{(i)}$ is the temperature on day t . Intuitively, the specific order of our inputs should matter in our prediction. If temperatures have been increasing lately, we might expect them to keep increasing (and vice versa). Our model needs to “understand” that the temperature $x_t^{(i)}$ comes *before* $x_{t+1}^{(i)}$.

Motivating RNNs

Thus, we have seen so far that our model should:

1. Work on sequences of any length
2. Take into account the order of the input features

Motivating RNNs

Note that sequence data are everywhere:

- Time data (eg. weather and stocks)
- Sentences (sequence of words)
- Videos (sequence of frames)
- DNA (sequence of nucleotides)
- etc...

Motivating RNNs

There are also a huge number of tasks we might perform using sequences:

- Predict the price of a stock tomorrow (seq2vec)
- Analyze the sentiment in a sentence (seq2vec)
- Caption an image (vec2seq)
- Translate a sentence from english to french (seq2seq)

(“seq” is short for sequence, “vec” is short for vector, and “x2y” means a task that requires mapping data type x to y)

Motivating RNNs

There are also a huge number of tasks we might perform using sequences:

- Predict the price of a stock tomorrow (seq2vec)
- Analyze the sentiment in a sentence (seq2vec)
- Caption an image (vec2seq)
- Translate a sentence from english to french (seq2seq)

(“seq” is short for sequence, “vec” is short for vector, and “x2y” means a task that requires mapping data type x to y)

Takeaway: Our models might need to learn very difficult tasks.

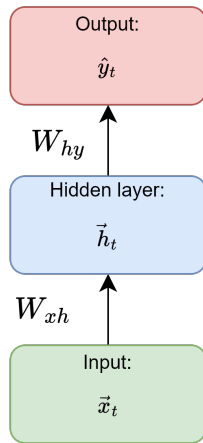
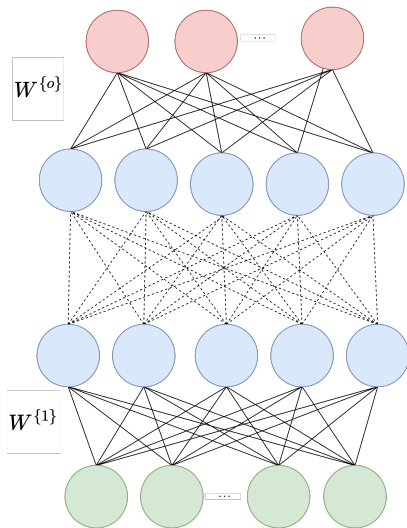
Motivating RNNs

Thus, we have seen so far that our model should:

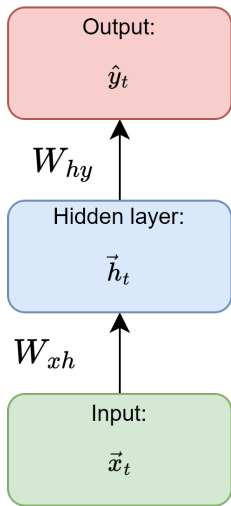
1. Work on sequences of any length
2. Take into account the order of the input features
3. Be able to learn complicated functions

Feed Forward Neural Networks: General Structure

First, let's simplify our diagram for feed-forward neural networks a bit to reduce clutter:



Feed Forward Neural Networks: General Structure

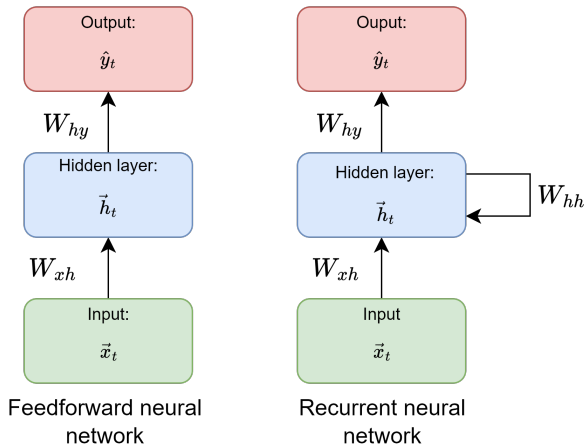


For simplicity, we will assume that there is a single hidden layer.

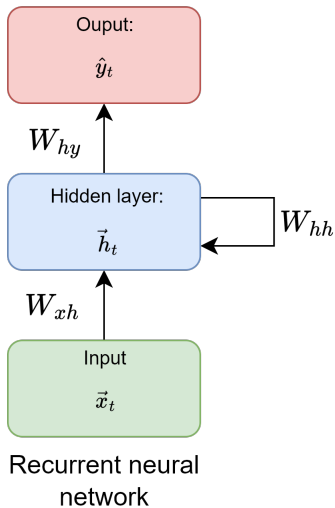
- W_{xh} are the weights that go from the input to first hidden layer (we previously called this $W^{\{1\}}$)
- W_{hy} are the weights that go from the last hidden layer to the output (we previously called this $W^{\{O\}}$)
- $\vec{h}_t$ are the activations in the hidden layer after feeding $\vec{x}_t$ as

RNNs: General Structure

So far, all the networks we have seen are *acyclic*, meaning none of the connections go backwards. In *recurrent neural networks*, we add a new connection that allows us to combine new inputs with the previous hidden state:



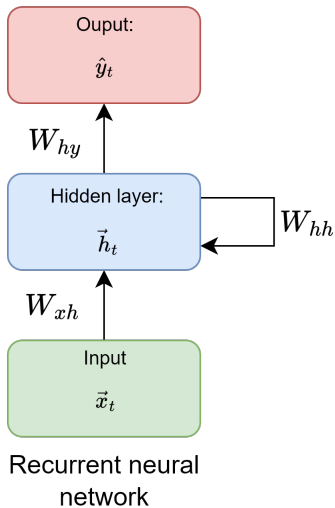
RNNs: General Structure



Note that the hidden layer now takes as input $W_{xh}\vec{x}_t$ and $W_{hh}\vec{h}_{t-1}$:

$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

RNNs: General Structure



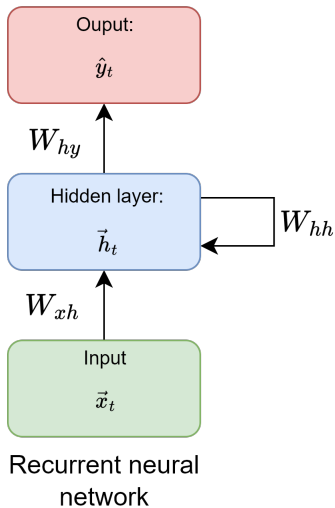
Note that the hidden layer now takes as input $W_{xh}\vec{x}_t$ and $W_{hh}\vec{h}_{t-1}$:

$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

The output layer is still the same as in feedforward networks:

$$\hat{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y)$$

RNNs: General Structure



Note that the hidden layer now takes as input $W_{xh}\vec{x}_t$ and $W_{hh}\vec{h}_{t-1}$:

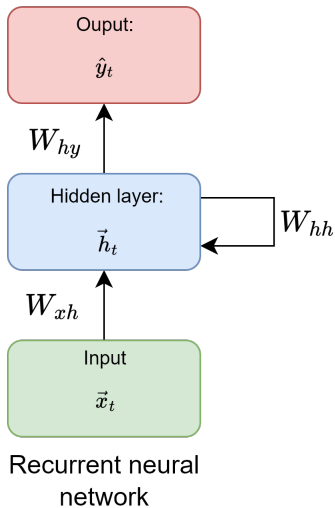
$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

The output layer is still the same as in feedforward networks:

$$\hat{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y)$$

For input $\vec{x}_t$ we can think of our network as having two outputs: $\vec{h}_t$ and $\vec{y}_t$, where $\vec{h}_t$ is fed back and combined with input $\vec{x}_{t+1}$.

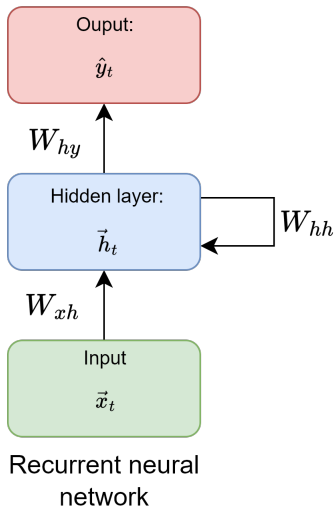
RNNs: General Structure



It is useful to think of the value of $\vec{h}_t$ as the *hidden state* of the RNN which is updated each time we receive a new input, $\vec{x}_{t+1}$.

In particular, we can think of this hidden state as a quantity that *summarizes* of the data we have seen so far.

RNNs: General Structure



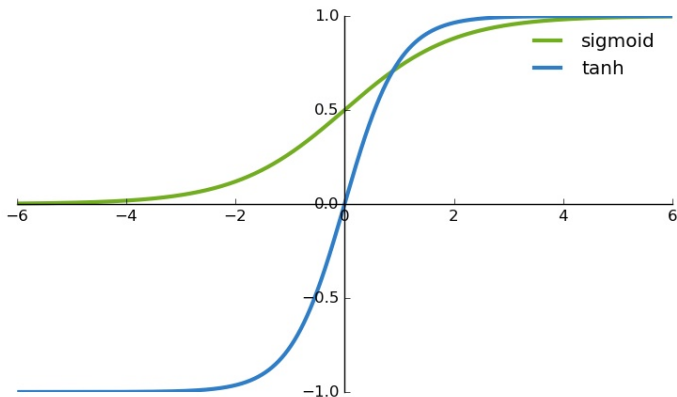
It is useful to think of the value of $\vec{h}_t$ as the *hidden state* of the RNN which is updated each time we receive a new input, $\vec{x}_{t+1}$.

In particular, we can think of this hidden state as a quantity that *summarizes* of the data we have seen so far.

That is, the state at time $t + 1$ is a function of input $\vec{x}_{t+1}$ and the previous state, h_t .

RNNs: General Structure

Note: For the hidden layer, it's common to use the $\phi_h(x) = \tanh(x)$ as the activation function in RNNs: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



```
class RNN(nn.Module):
    # ...

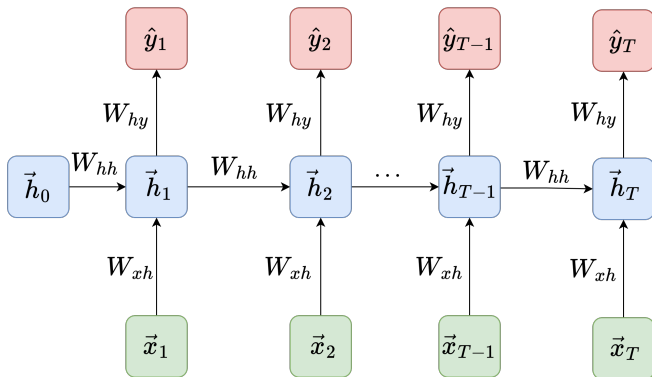
    def forward(self, x):
        # Update the hidden state:  $h_t = \tanh(W_{hh} * h_{t-1} + W_{xh} * x_t + b_h)$ 
        self.h = torch.tanh(self.W_hh @ self.h + self.W_xh @ x + self.b_h)

        # Compute the output vector:  $y_t = W_{hy} * h_t + b_y$ 
        y = self.W_hy @ self.h + self.b_y

    return y
```

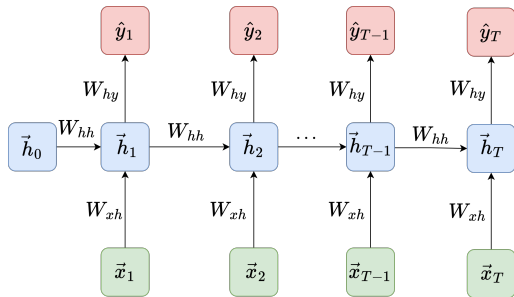
RNNs: General Structure

The recurrent structure can become a bit confusing to deal with. Instead, for a sequence of length T we can imagine making T copies of our neural network where copy t feeds the output of its hidden to copy $t + 1$:



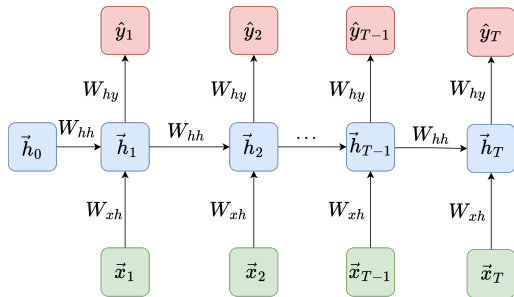
Important: The weights are the same across all the copies.

RNNs: General Structure



$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

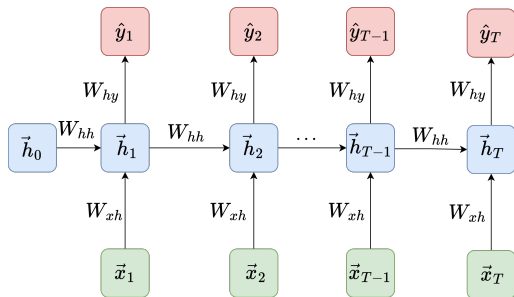
RNNs: General Structure



$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

$$\hat{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y)$$

RNNs: General Structure

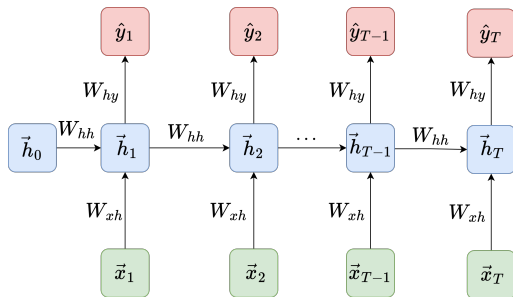


$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

$$\hat{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y)$$

$\vec{h}_0$ is some initial value (eg. $\vec{0}$)

RNNs: General Structure



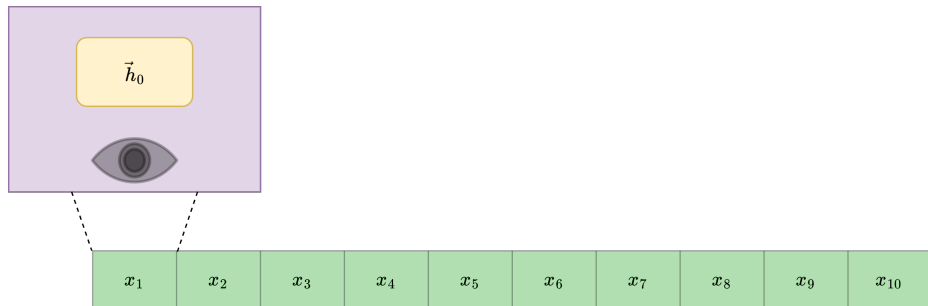
$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

$$\hat{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y)$$

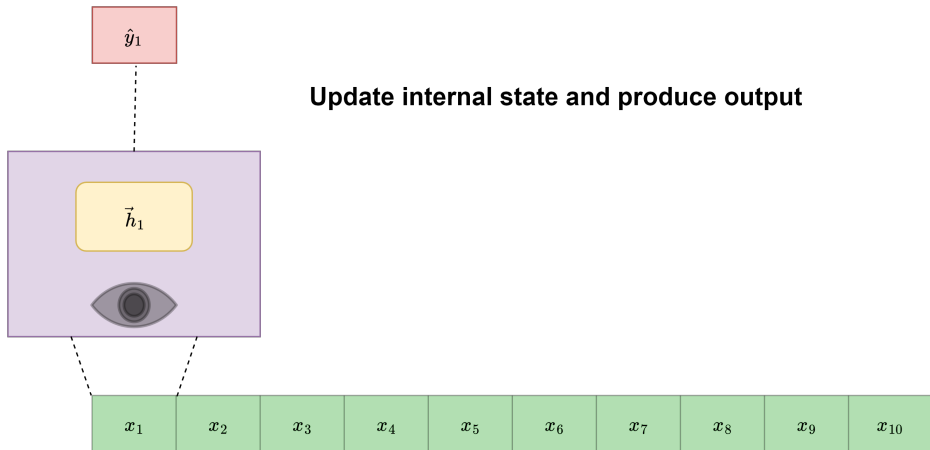
$\vec{h}_0$ is some initial value (eg. $\vec{0}$)

In this format we can see how the prediction for input t depends on all previous predictions.

One final interpretation



One final interpretation



RNNs: seq2vec

Let's start with a simple seq2vec example.

Task: Sentiment analysis

- Inputs: Reviews of movies
- Outputs: Probability it's a positive review

RNNs: seq2vec

Let's start with a simple seq2vec example.

Task: Sentiment analysis

- Inputs: Reviews of movies
- Outputs: Probability it's a positive review

For simplicity, we will assume the reviews must be consist only of words from the following vocabulary: “it”, “was”, “good”, “bad”, “not”

eg. “it was good”

eg. “it was not bad”

RNNs: seq2vec

Our dataset looks something like the following:

#	Review ($\vec{x}$)	Sentiment (y)
1	“it was good”	Positive (1)
$\vdots$	$\vdots$	$\vdots$
15	“it was not good”	Negative (0)
$\vdots$	$\vdots$	$\vdots$

RNNs: seq2vec

We can treat each word as its own feature:

#	x_1	x_2	x_3	x_4	Sentiment (y)
1	“it	“was”	“good”	“”	Positive (1)
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
15	“it	“was”	“not”	“good”	Negative (0)
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

RNNs: seq2vec

We have one last problem: our inputs are text but our models work with numbers/vectors. We'll convert each word to its one-hot encoding.

RNNs: seq2vec

We have one last problem: our inputs are text but our models work with numbers/vectors. We'll convert each word to its one-hot encoding.

We have 5 words in our vocabulary so these will be 5 dimensional vectors:

$$\text{it} \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \text{was} \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \text{good} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{bad} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{not} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

We are predicting probabilities, so we will apply sigmoid to our output (since it maps values to $[0, 1]$). Thus, we will have

$$\vec{h}_t = \phi_h(W_{xh} \vec{x}_t + W_{hh} \vec{h}_{t-1} + \vec{b}_h) = \tanh(W_{xh} \vec{x}_t + W_{hh} \vec{h}_{t-1} + \vec{b}_h)$$

We are predicting probabilities, so we will apply sigmoid to our output (since it maps values to $[0, 1]$). Thus, we will have

$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h) = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

$$\vec{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y) = \sigma(W_{hy}\vec{h}_t + \vec{b}_y)$$

We are predicting probabilities, so we will apply sigmoid to our output (since it maps values to $[0, 1]$). Thus, we will have

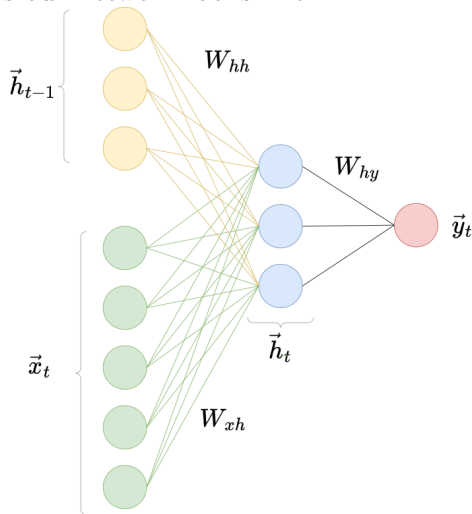
$$\vec{h}_t = \phi_h(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h) = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

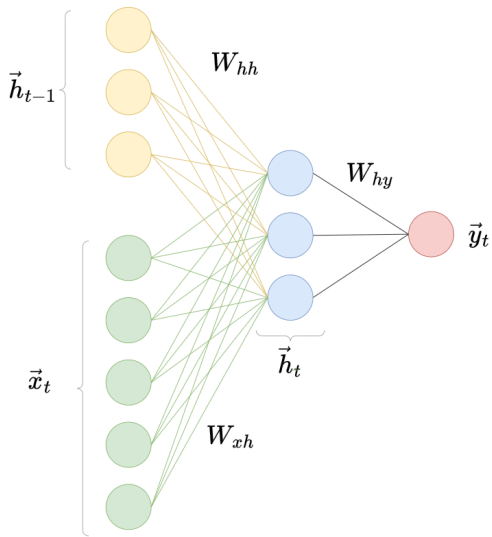
$$\vec{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y) = \sigma(W_{hy}\vec{h}_t + \vec{b}_y)$$

Since we are doing binary classification, our loss function will be the binary cross entropy loss function.

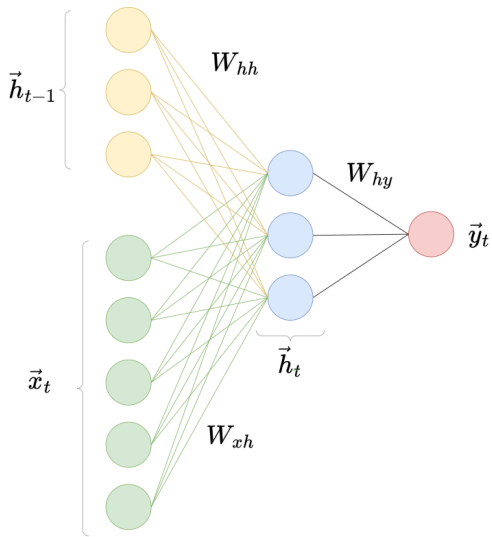
RNNs: seq2vec

Each input is a 5 dimensional vector and our output is a 1D probability vector.
With three hidden units our network looks like:



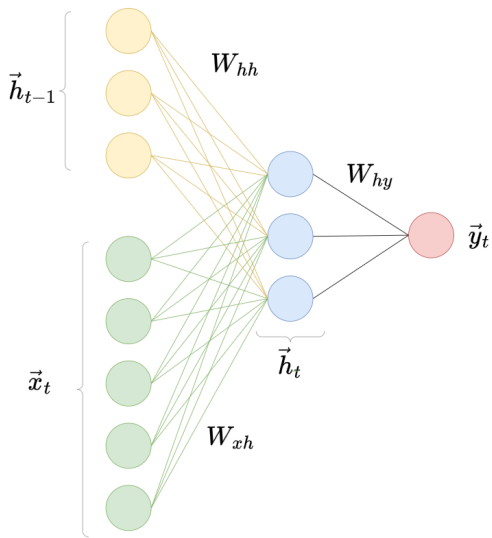


$$\vec{h}_t = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$



$$\vec{h}_t = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

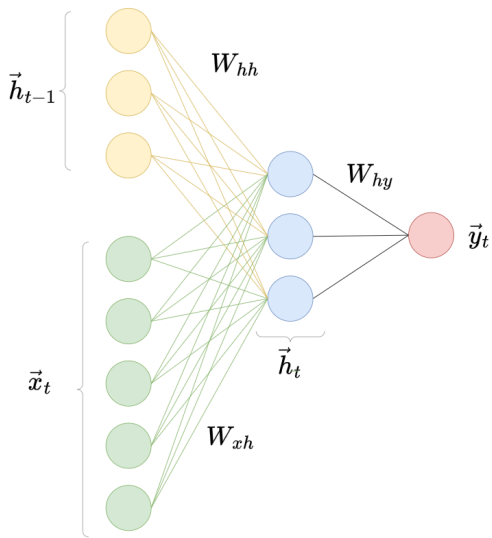
$$\vec{y}_t = \sigma(W_{hy}\vec{h}_t + \vec{b}_y)$$



$$\vec{h}_t = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

$$\vec{y}_t = \sigma(W_{hy}\vec{h}_t + \vec{b}_y)$$

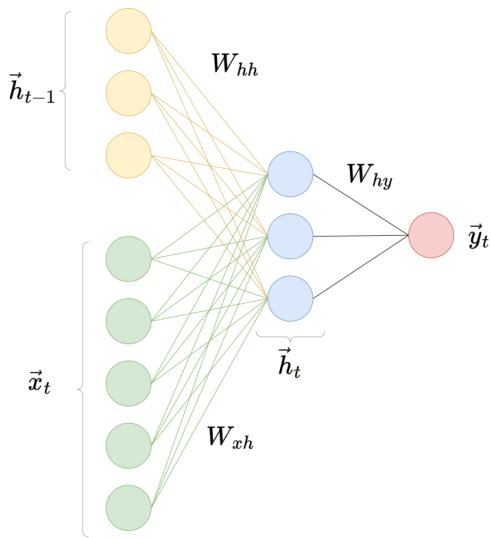
$\vec{h}_0$ is some initial value (eg. $\vec{0}$)



$$\vec{h}_t = \tanh(W_{xh}\vec{x}_t + W_{hh}\vec{h}_{t-1} + \vec{b}_h)$$

$$\vec{y}_t = \sigma(W_{hy}\vec{h}_t + \vec{b}_y)$$

$\vec{h}_0$ is some initial value (eg. $\vec{0}$)



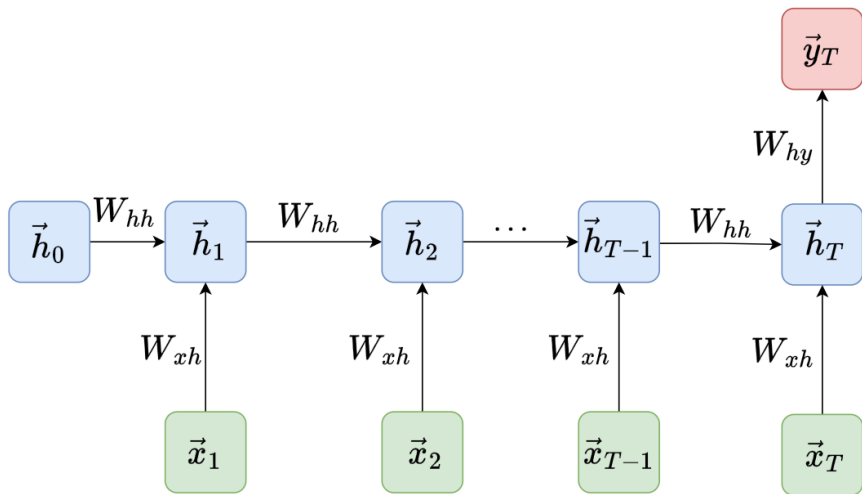
In practice, we can concatenate $\vec{h}_{t-1}$ and $\vec{x}_t$ into a big vector, and stack W_{hh} and W_{xh} into a big matrix:

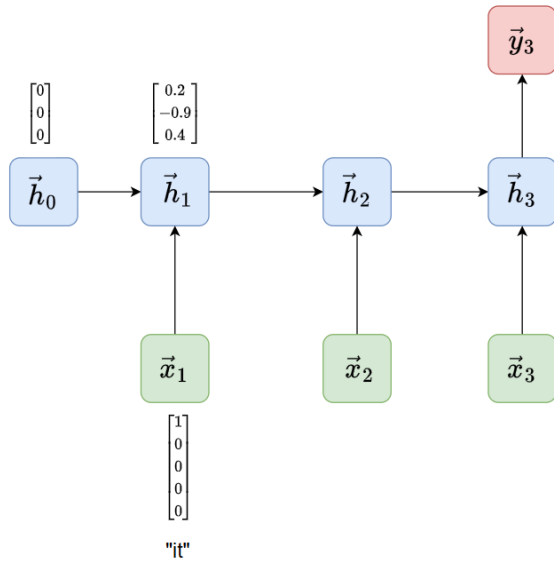
$$\begin{bmatrix} \vec{h}_{t-1} \\ \vec{x}_t \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} W_{hh} & W_{xh} \end{bmatrix}.$$

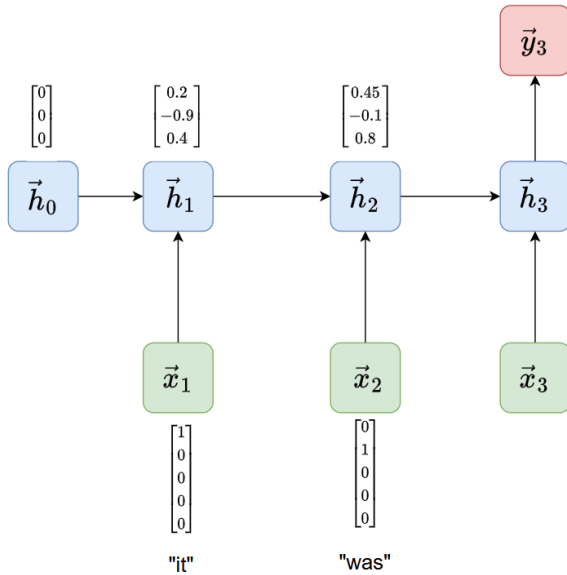
This allows us to compute the hidden state update as:

$$\vec{h}_t = \tanh \left(\begin{bmatrix} W_{hh} & W_{xh} \end{bmatrix} \begin{bmatrix} \vec{h}_{t-1} \\ \vec{x}_t \end{bmatrix} + \vec{b}_h \right).$$

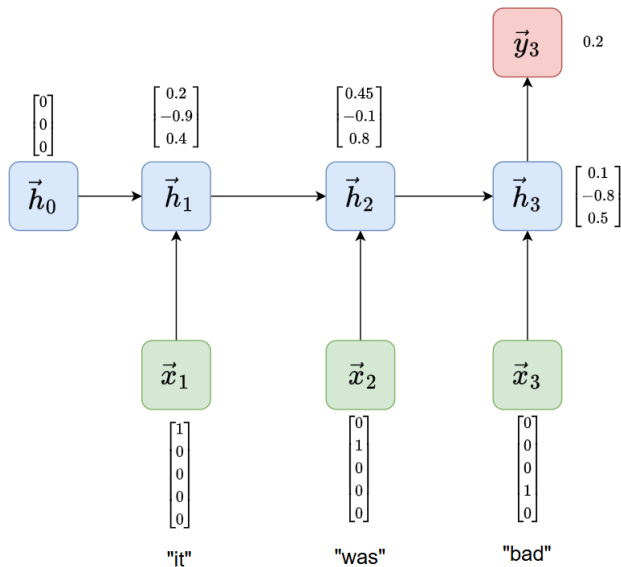
In this case we only care about the *final output*. If we unfold the network it looks like:







Our model predicts that there is a 20% probability that this is a positive review:



Note that we could've used a similar network for any task where we wanted to predict a vector using a sequence.

- Predict if sentence is in english, french, or spanish: predict a 3D probability vector by applying softmax to output and use cross entropy loss

Now let's look at a seq2seq example. This time, we will train our model to predict the next letter in a word.

Now let's look at a seq2seq example. This time, we will train our model to predict the next letter in a word.

Task: Given the first l letters from a word, predict the $l + 1$ -th letter

- Input: Letter
- Output: Probabilities of the next letters

Our dataset is simply a bunch of english words:

#	Word ($\vec{x}$)
1	math
$\vdots$	$\vdots$
15	banana
$\vdots$	$\vdots$

This time we will split each word into its individual letters:

#	x_1	x_2	x_3	x_4	x_5	x_6
1	m	a	t	h		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
15	b	a	n	a	n	a
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

This time we will split each word into its individual letters:

#	x_1	x_2	x_3	x_4	x_5	x_6
1	m	a	t	h		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
15	b	a	n	a	n	a
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

But what are the y values...?

Our task is to predict the *next letter*. For example $\vec{x}$ = “math”.

$$\vec{x} = [\text{m}, \text{a}, \text{t}, \text{h}]$$

Our task is to predict the *next letter*. For example $\vec{x}$ = “math”.

$$\vec{x} = [\text{m}, \text{a}, \text{t}, \text{h}]$$

$$\vec{y} = [\text{a}, \text{t}, \text{h}, \langle \text{END} \rangle]$$

Our task is to predict the *next letter*. For example $\vec{x}$ = “math”.

$$\vec{x} = [\text{m}, \text{a}, \text{t}, \text{h}]$$

$$\vec{y} = [\text{a}, \text{t}, \text{h}, \text{<END>}]$$

For feature x_t , the correct prediction is y_t :

- For m the correct next letter is a
- For a the correct next letter is t
- For t the correct next letter is h
- For h the correct next letter is <END>, which signifies the string is terminated

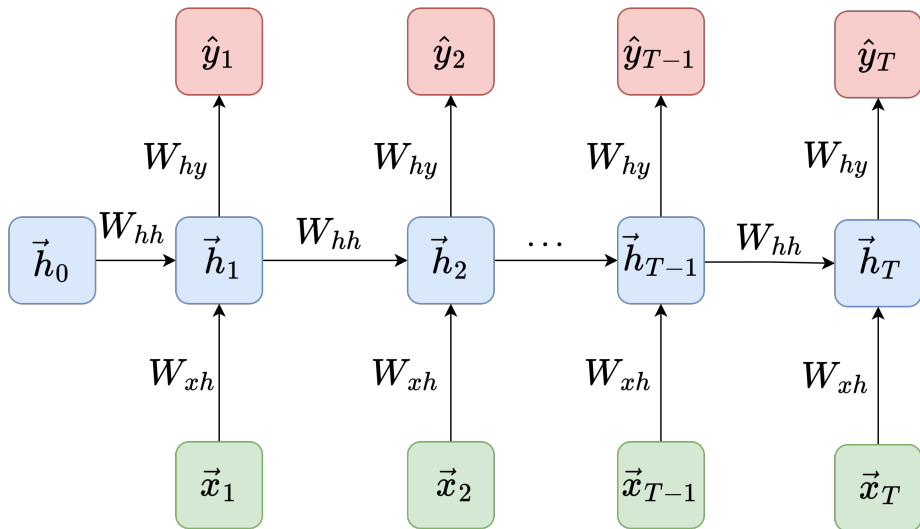
Our y values are thus the correct next letter:

#	x_1	x_2	x_3	x_4	x_5	x_6	y_1	y_2	y_3	y_4	y_5	y_6
1	m	a	t	h			a	t	h	<END>		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
15	b	a	n	a	n	a	a	n	a	n	a	<END>
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

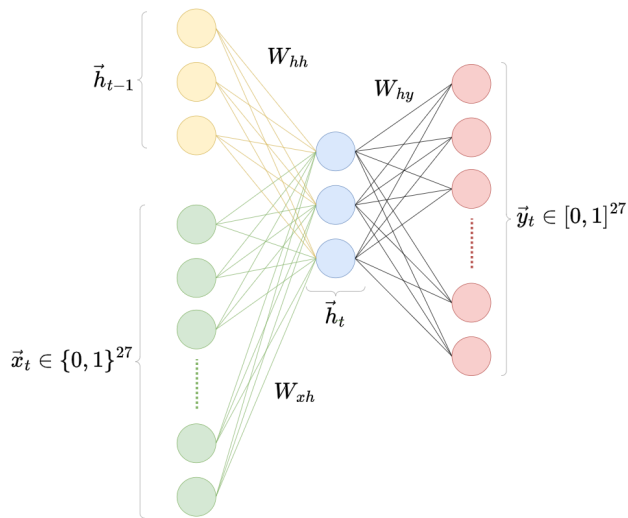
Again, we will use the one-hot encoding. This time, we have 27 dimensional vectors (one entry per letter of the alphabet + <END> token):

$$a \rightarrow \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \dots, z \rightarrow \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \quad \langle \text{END} \rangle \rightarrow \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

Unlike in sentiment analysis where we only cared about the last output, this time we care about all the outputs:

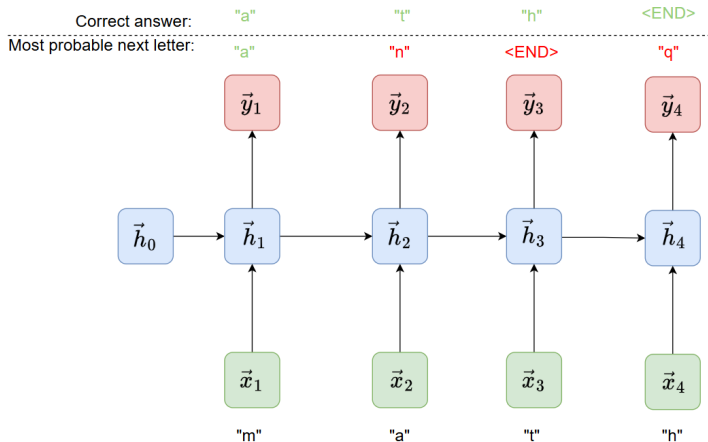


Again, we will assume we have 3 hidden units. Our input is a 27D one-hot vector and our output is a 27D probability vector representing the probability of each letter:



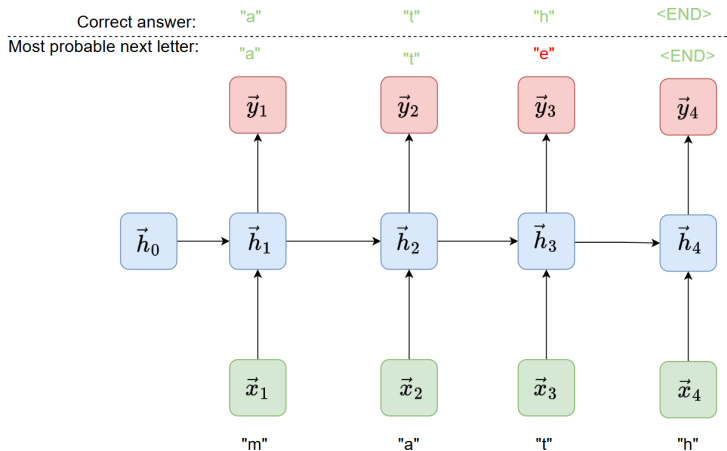
Training the network

To train this model we use cross entropy loss, where for each letter we compare the probabilities to the correct answer. At the start of training, the predictions will likely be nonsense...



Training the network

As the model improves, it will come up with more sensible answers:



Inference

Once the model is trained, we can use it for *inference*. In this case our model can generate new words!

Steps

1. Feed the network a starting letter as a one-hot encoding.
 - Eg. If I want it to generate a word starting with the letter “m” I can provide $x_1 = \text{one-hot encoding of “m”}$

Inference

Once the model is trained, we can use it for *inference*. In this case our model can generate new words!

Steps

1. Feed the network a starting letter as a one-hot encoding.
 - Eg. If I want it to generate a word starting with the letter “m” I can provide $x_1 = \text{one-hot encoding of “m”}$
2. The model will spit out a prediction of the probabilities of the next letter, $\vec{y}_1$. Choose the letter with the highest probability, set that to be x_2 and feed it back into the network.

Inference

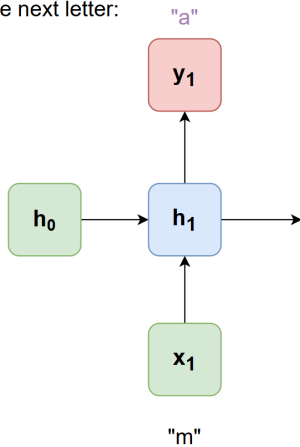
Once the model is trained, we can use it for *inference*. In this case our model can generate new words!

Steps

1. Feed the network a starting letter as a one-hot encoding.
 - Eg. If I want it to generate a word starting with the letter “m” I can provide $x_1 = \text{one-hot encoding of “m”}$
2. The model will spit out a prediction of the probabilities of the next letter, $\vec{y}_1$. Choose the letter with the highest probability, set that to be x_2 and feed it back into the network.
3. Repeat the above until the most probable letter is $\langle \text{END} \rangle$

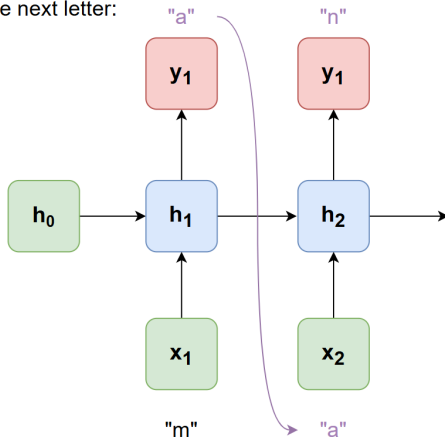
Inference

Most probable next letter:



Inference

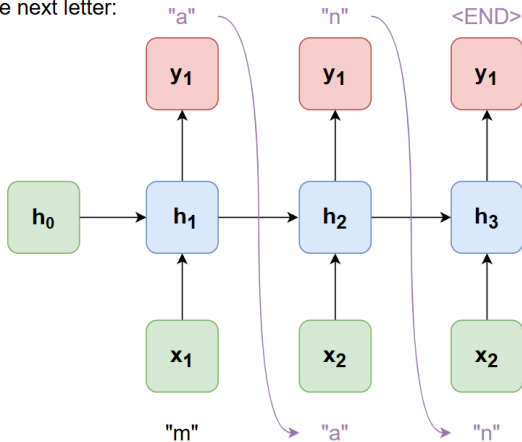
Most probable next letter:



Inference

In this case, our model generates the word “man”:

Most probable next letter:



Some notes:

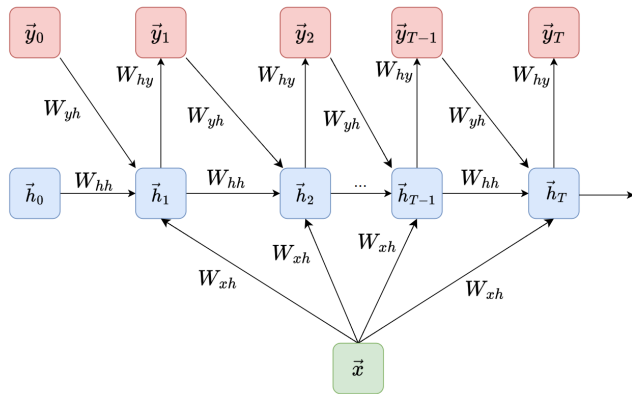
- This is quite similar to what ChatGPT does! First, it's pretrained on a bunch of data, then when you send it a prompt, it tries to complete the rest of the prompt using the strategy we saw for inference (ie. it predicts the next word one at a time).

Some notes:

- This is quite similar to what ChatGPT does! First, it's pretrained on a bunch of data, then when you send it a prompt, it tries to complete the rest of the prompt using the strategy we saw for inference (ie. it predicts the next word one at a time).
- The model we used is quite simple since it only generates single words. What if we wanted to translate words? In that case our output isn't necessarily the same size as our input... How do we train this? (We will come back to this...)

vec2seq

Due to time constraints, we will gloss over vec2seq relatively quickly. In general, they are given a single vector $\vec{x}$ as input which is provided to all cells of the RNN. The only other key difference is that we feed the prediction $\vec{y}_t$ as input at time $t + 1$:



Example: Image captioning

- Input: An embedding of an image $\vec{x}$
- Output: A sentence describing the image (each individual $\vec{y}_t$ is the probability of each possible next word in the sentence)

Example: Image captioning

- Input: An embedding of an image $\vec{x}$
- Output: A sentence describing the image (each individual $\vec{y}_t$ is the probability of each possible next word in the sentence)

variations on RNNs: encoder/decoder models

We can also combine these different strategies to create even more complex networks.

variations on RNNs: encoder/decoder models

We can also combine these different strategies to create even more complex networks.

Our seq2seq model from earlier had a problem: the length of the x and y values in our dataset had to be the same

variations on RNNs: encoder/decoder models

We can also combine these different strategies to create even more complex networks.

Our seq2seq model from earlier had a problem: the length of the x and y values in our dataset had to be the same

Idea: If we feed the output of a seq2vec model into a vec2seq model, we get a seq2seq model which can produce sequences of arbitrary length!

variations on RNNs: encoder/decoder models

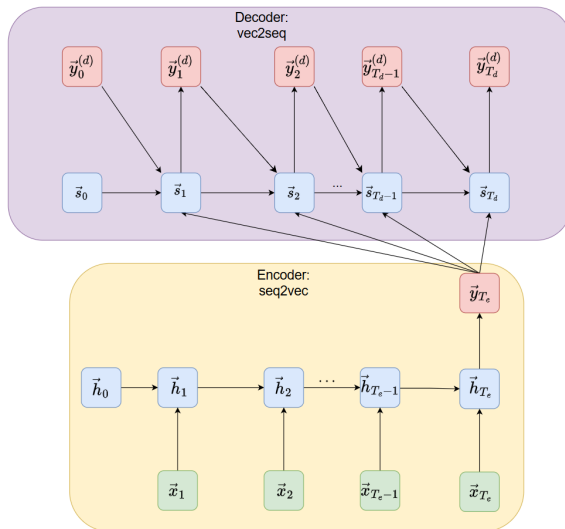
We can also combine these different strategies to create even more complex networks.

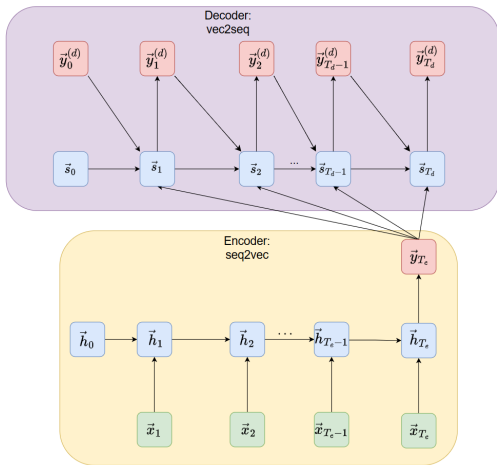
Our seq2seq model from earlier had a problem: the length of the x and y values in our dataset had to be the same

Idea: If we feed the output of a seq2vec model into a vec2seq model, we get a seq2seq model which can produce sequences of arbitrary length!

variations on RNNs: encoder/decoder models

A seq2vec + vec2seq model is typically called an encoder/decoder model:





Example

1. The seq2vec model *encodes* an english sentence into a single vector $\vec{y}_{T_e}$ which summarizes its content
2. The vec2seq model *decodes* $\vec{y}_{T_e}$ into a french sentence

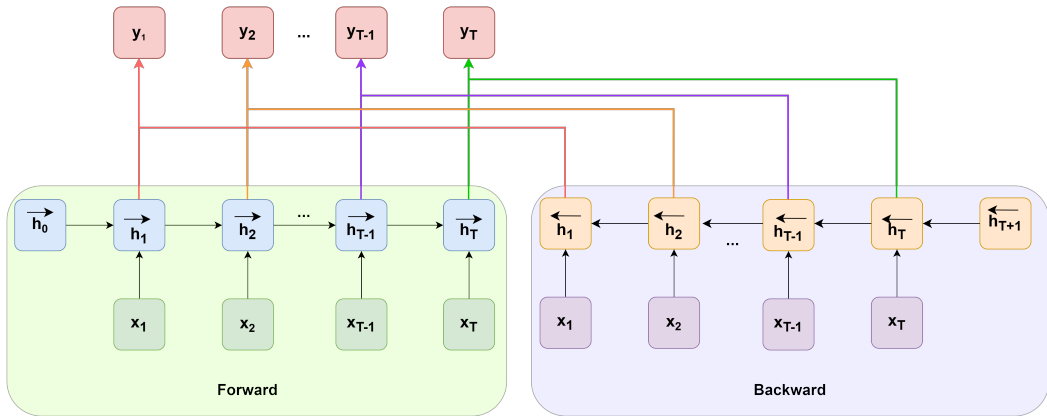
variations on RNNs: bidirectional RNNs

Another interesting variation is a *bidirectional RNN*.

variations on RNNs: bidirectional RNNs

Another interesting variation is a *bidirectional RNN*.

Idea: So far, our RNNs only predict values in the future based on values from the past. We could likely get better predictions if we could look at both the past and the future. Let's feed $\vec{x}_t$ forwards and backwards to our model and combine the results!



Let $\overleftarrow{h}_t$ be the hidden state at time t for the backward part of the network. Also, let f and b denote the forward part and backward part of the network respectively. Then,

$$\vec{h}_t = \phi_h(W_{xh}^{(f)} \vec{x}_t + W_{hh}^{(f)} \vec{h}_{t-1} + \vec{b}_h^{(f)})$$

$$\overleftarrow{h}_t = \phi_h(W_{xh}^{(b)} \overleftarrow{x}_t + W_{hh}^{(b)} \overleftarrow{h}_{t+1} + \overleftarrow{b}_h^{(b)})$$

Let $\overleftarrow{h}_t$ be the hidden state at time t for the backward part of the network. Also, let f and b denote the forward part and backward part of the network respectively. Then,

$$\vec{h}_t = \phi_h(W_{xh}^{(f)} \vec{x}_t + W_{hh}^{(f)} \vec{h}_{t-1} + \vec{b}_h^{(f)})$$

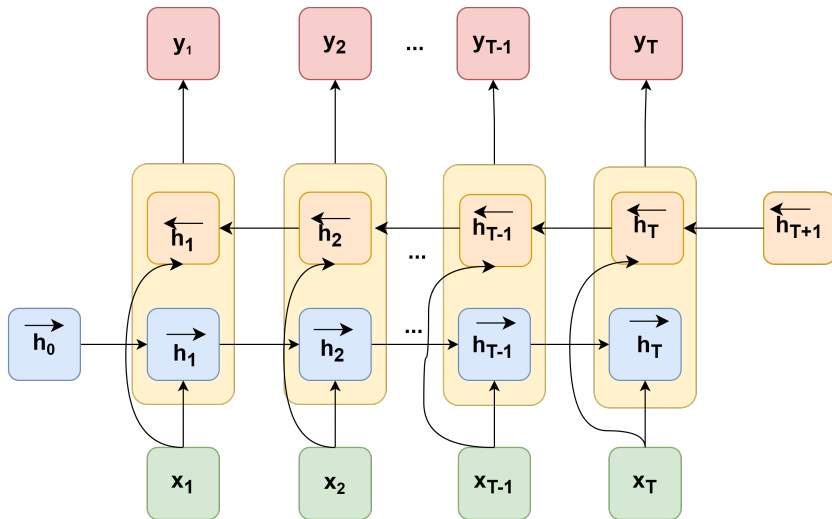
$$\overleftarrow{h}_t = \phi_h(W_{xh}^{(b)} \overleftarrow{x}_t + W_{hh}^{(b)} \overleftarrow{h}_{t+1} + \overleftarrow{b}_h^{(b)})$$

$$\mathbf{h}_t = [\vec{h}_t, \overleftarrow{h}_t]$$

Our prediction at time t is then:

$$\vec{y}_t = \phi_y(W_{hy} \mathbf{h}_t + \vec{b}_y)$$

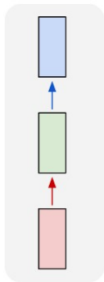
With this idea of concatenation in mind, we can instead draw the network in the following way which emphasizes which pairs are concatenated together:



summary

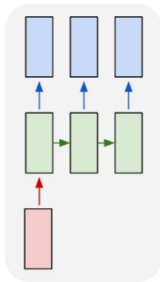
In general, RNNs fall into one of the following types, depending on the task:

one to one



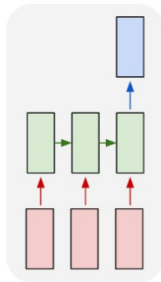
feed
forward NN

one to many



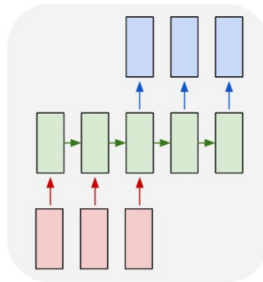
music
generation

many to one



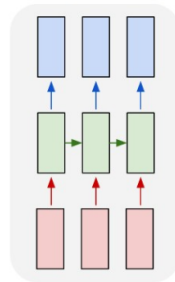
sentiment
analysis

many to many



translate
from EN to
FR

many to many



next letter
prediction

The problem with RNNs

The key motivation behind RNNs is that they can use previous inputs to make predictions about new inputs (they “remember” previous inputs by summarizing their information in the hidden state $\vec{h}_t$).

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

Ex: “The clouds are in the *sky*”

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

Ex: “The clouds are in the *sky*”

Here, it’s obvious that the next word must be “sky”.

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

Ex: “The clouds are in the *sky*”

Here, it’s obvious that the next word must be “sky”.

However, they begin to struggle as sentences get longer.

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

Ex: “The clouds are in the *sky*”

Here, it’s obvious that the next word must be “sky”.

However, they begin to struggle as sentences get longer.

Ex: “I grew up in France... I speak fluent *French*.”

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

Ex: “The clouds are in the *sky*”

Here, it’s obvious that the next word must be “sky”.

However, they begin to struggle as sentences get longer.

Ex: “I grew up in France... I speak fluent *French*.”

In this case, it’s obvious that the next word after “fluent” is the name of a language, but to know the correct language we need to remember that the person is from France.

We saw earlier that RNNs can be used to predict the next letter / word in a sentence. In general, they can be quite good with short sentences.

Ex: “The clouds are in the *sky*”

Here, it’s obvious that the next word must be “sky”.

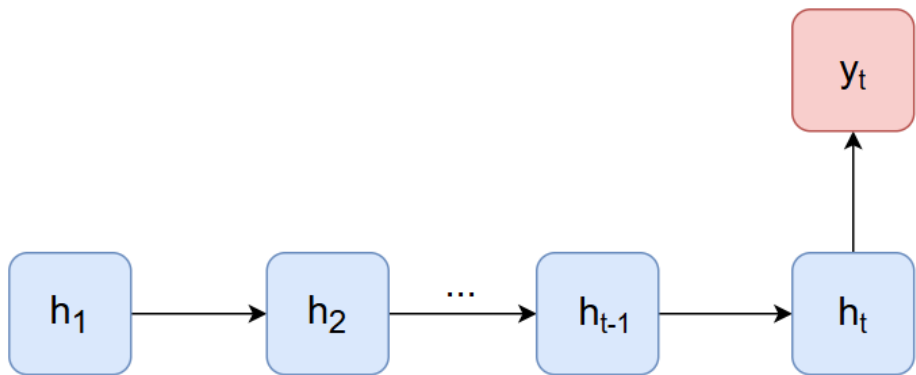
However, they begin to struggle as sentences get longer.

Ex: “I grew up in France... I speak fluent *French*.”

In this case, it’s obvious that the next word after “fluent” is the name of a language, but to know the correct language we need to remember that the person is from France.

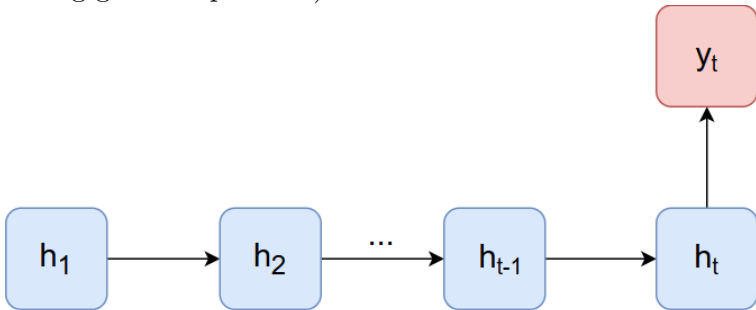
In short, RNNs struggle with *long term dependencies*.

To see why RNNs struggle with *long term dependencies*, let's take a look at a simplified RNN:



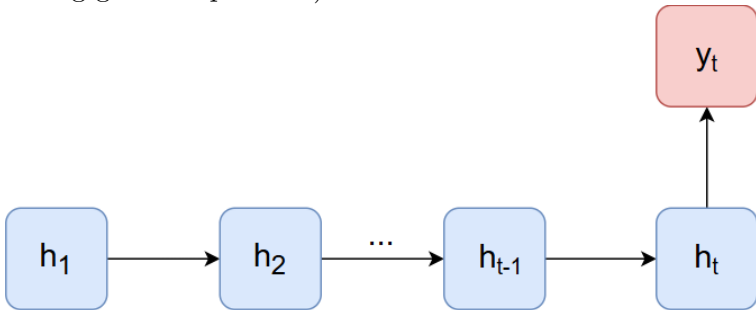
$$\frac{\partial y_t}{\partial h_1} = ?$$

If we multiply a bunch of numbers close to zero together, we get a really small number (vanishing gradient problem):



$$\frac{\partial y_t}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \dots \times \frac{\partial h_t}{\partial h_{t-1}} \times \frac{\partial y_t}{\partial h_t}$$

If we multiply a bunch of numbers close to zero together, we get a really small number (vanishing gradient problem):



$$\frac{\partial y_t}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \dots \times \frac{\partial h_t}{\partial h_{t-1}} \times \frac{\partial y_t}{\partial h_t}$$

The further you get from the output, the worse it gets (ie. h_1 is more affected than h_2 , h_2 more affected than h_3 , etc...)

$$\frac{\partial y_t}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \cdots \times \frac{\partial h_t}{\partial h_{t-1}} \times \frac{\partial y_t}{\partial h_t}$$

$\frac{\partial y_t}{\partial h_k}$ tells you how much a change to hidden state k will affect output t . If this is small, then long term dependencies have no effect on the output.

$$\frac{\partial y_t}{\partial h_1} = \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \cdots \times \frac{\partial h_t}{\partial h_{t-1}} \times \frac{\partial y_t}{\partial h_t}$$

$\frac{\partial y_t}{\partial h_k}$ tells you how much a change to hidden state k will affect output t . If this is small, then long term dependencies have no effect on the output.

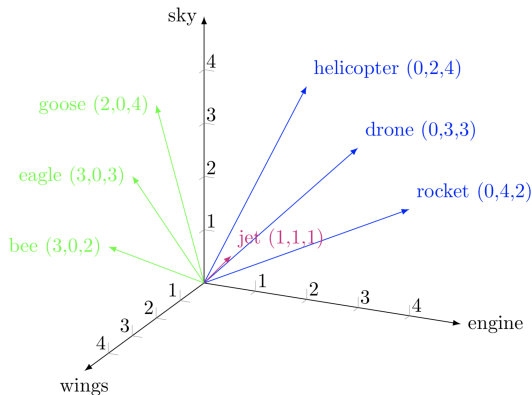
If the model predicted “I grew up in France... I speak fluent *Mandarin*” and got the answer wrong, then it wouldn’t be able to correctly update the weights to account for the long-term dependency on the word “France”.

Solving the long-term dependency issue

Long Short-Term Memory (LSTM) networks were created to solve the long-term dependency issue in RNNs. These networks do not suffer from the vanishing/exploding gradient problem and have some pretty cool concepts.

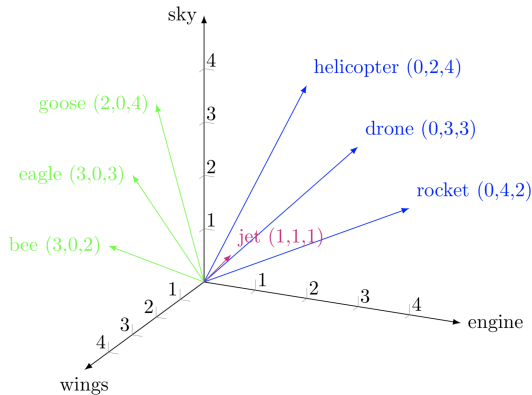
Learning representations

To make the following explanation of LSTMs easier to follow, we will assume that the vectors store information in some interpretable way.



Learning representations

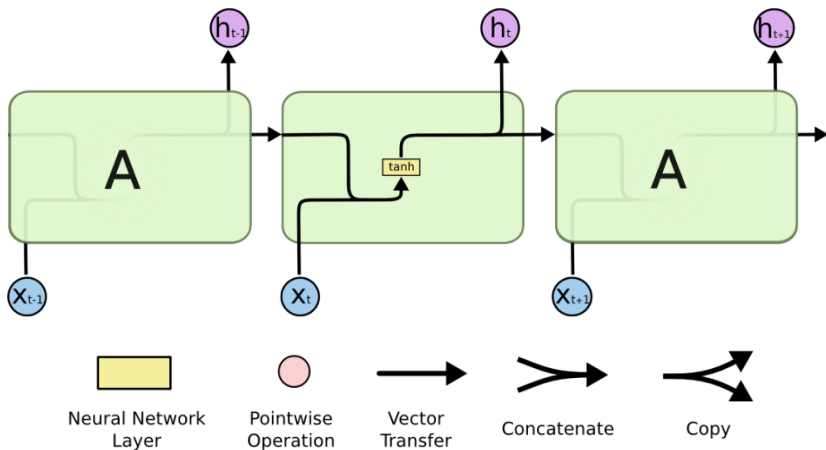
To make the following explanation of LSTMs easier to follow, we will assume that the vectors store information in some interpretable way.



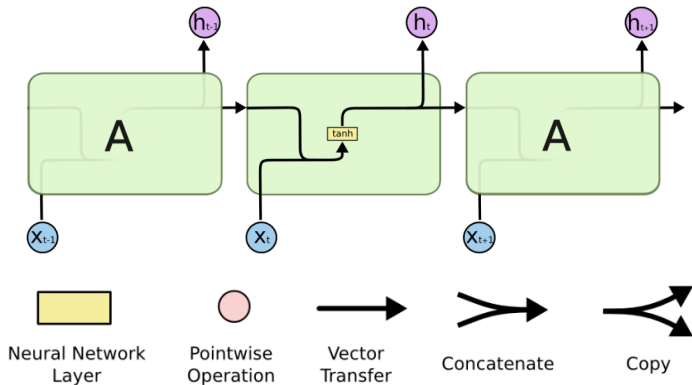
Key Idea: We can have vectors where each dimension represents certain information.

In our example, we will use an LSTM to predict the next word in a sentence. We will assume that cell states (explained soon) encode information about things like the gender of the subject, whether the subject is singular or plural, etc...

The following is a simple RNN (note that in these diagrams the output is $\vec{h}_t$, not $\vec{y}_t$. this is equivalent since in practice our output is just some linear layer + activation applied to $\vec{h}_t$):

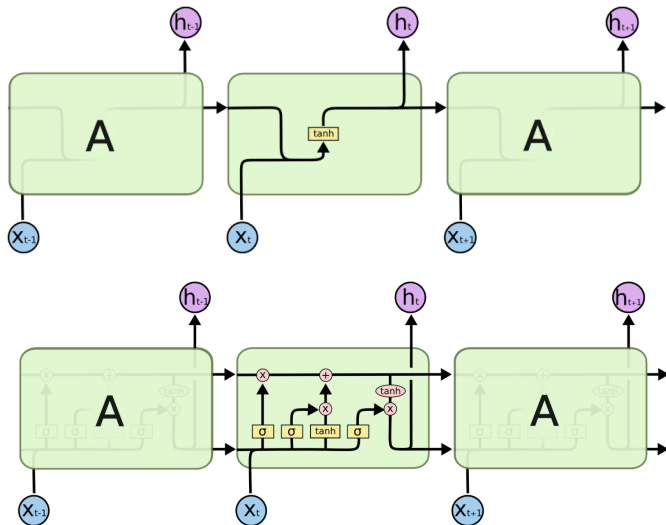


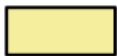
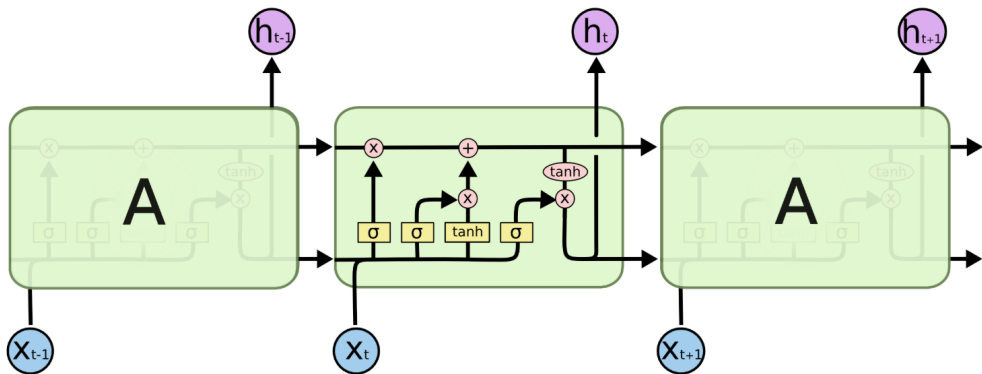
Let's understand the notation and how it fits in with what we've seen so far:



$$\vec{h}_t = \tanh \left(\begin{bmatrix} W_{hh} & W_{xh} \end{bmatrix} \begin{bmatrix} \vec{h}_{t-1} \\ \vec{x}_t \end{bmatrix} + \vec{b}_h \right) \quad \vec{y}_t = \phi_y(W_{hy}\vec{h}_t + \vec{b}_y)$$

Let's compare this to an LSTM:





Neural Network
Layer



Pointwise
Operation



Vector
Transfer



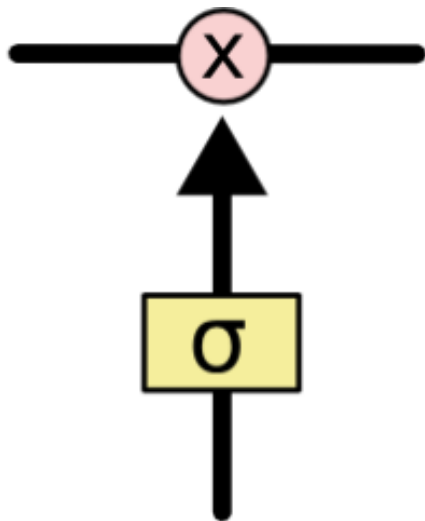
Concatenate



Copy

The key mechanism that sets LSTMs apart is their ability to *decide which information to remember*.

This is achieved using a “forget gate”, which consists of a sigmoid function and pointwise multiplication:



Recall that sigmoid maps values to the range $[0, 1]$.

$$\sigma(\vec{a}) \otimes \vec{b}$$

($\otimes$ denotes pointwise multiplication)

We can think of $\sigma(\vec{a})$ as telling us “what percent of each element of $\vec{b}$ should I remember?”

Example

Let $\sigma(\vec{a}) = [0, 0.9]$ and $\vec{b} = [\text{subject gender}, \text{subject plural}] = [13.2, 10]$

Example

Let $\sigma(\vec{a}) = [0, 0.9]$ and $\vec{b} = [\text{subject gender}, \text{subject plural}] = [13.2, 10]$

$\sigma(\vec{a})$ tells us “remember 0% of the information about the gender” and “remember 90% of the information about being plural”.

$$\sigma(\vec{a}) \otimes \vec{b} = [0 \times 13.2, 0.9 \times 10] = [0, 9]$$

Example

Let $\sigma(\vec{a}) = [0, 0.9]$ and $\vec{b} = [\text{subject gender}, \text{subject plural}] = [13.2, 10]$

$\sigma(\vec{a})$ tells us “remember 0% of the information about the gender” and “remember 90% of the information about being plural”.

$$\sigma(\vec{a}) \otimes \vec{b} = [0 \times 13.2, 0.9 \times 10] = [0, 9]$$

Whenever you see sigmoid being multiplied with a value in this network, think of it this way!

Example

Let $\sigma(\vec{a}) = [0, 0.9]$ and $\vec{b} = [\text{subject gender}, \text{subject plural}] = [13.2, 10]$

$\sigma(\vec{a})$ tells us “remember 0% of the information about the gender” and “remember 90% of the information about being plural”.

$$\sigma(\vec{a}) \otimes \vec{b} = [0 \times 13.2, 0.9 \times 10] = [0, 9]$$

Whenever you see sigmoid being multiplied with a value in this network, think of it this way!

This will allow our network to “choose” which information is worth remembering at each step.

From our previous example, the network could choose to remember that the subject grew up in France in the long term...

Note: In the following sections I will omit the arrow above vectors to make the notation less cluttered. Assume that any variable (other than the weight matrices denoted by W) are *vectors*.

Each cell of an LSTM has the following *inputs*:

- C_{t-1} : the long term memory up to time step $t - 1$
- h_{t-1} : the hidden state from time step $t - 1$
- x_t : the input at time t

and the following *outputs*:

- C_t : the long term memory up to time step t
- h_t : the hidden state from time step t

Each cell of an LSTM has the following *inputs*:

- C_{t-1} : the long term memory up to time step $t - 1$
- h_{t-1} : the hidden state from time step $t - 1$
- x_t : the input at time t

and the following *outputs*:

- C_t : the long term memory up to time step t
- h_t : the hidden state from time step t

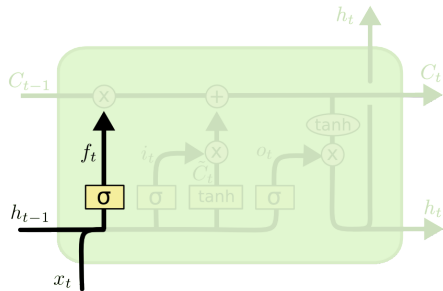
The key idea in LSTMs is that in each cell we will update the hidden state and long term memory by:

- Forgetting information (using “forget gate”)
- Adding new information (using vector addition)

In this way, the network “chooses” which information is relevant at each step.

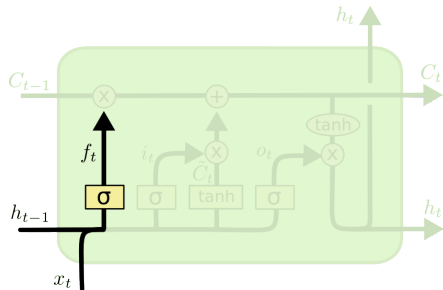
Let's look at what an LSTM does step by step...

First, it combines h_{t-1} and x_t into a “forget gate”. The output of this gate will be applied to C_{t-1} (ie. we will use it to forget information from C_{t-1}):



$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

First, it combines h_{t-1} and x_t into a “forget gate”. The output of this gate will be applied to C_{t-1} (ie. we will use it to forget information from C_{t-1}):

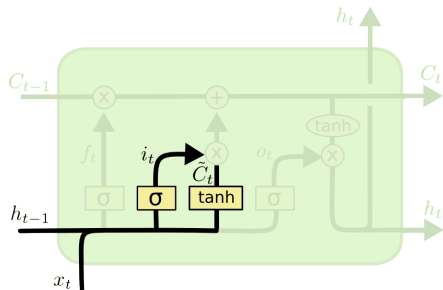


$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

Example

The current long term memory (C_{t-1}) might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to *forget the gender of the old subject*.

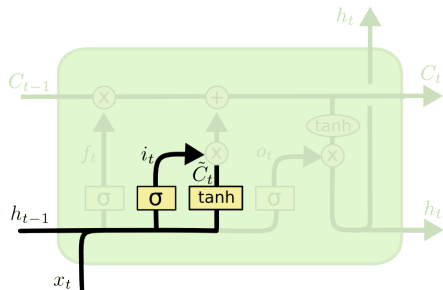
Next, we compute a new “candidate” long term memory $\tilde{C}_t$. We also pass h_{t-1} and x_t into another “forget gate”; we call the result i_t . We will combine these two results to decide which of the “candidate” long term memory should be included in the actual long term memory.



$$i_t = \sigma (W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Next, we compute a new “candidate” long term memory $\tilde{C}_t$. We also pass h_{t-1} and x_t into another “forget gate”; we call the result i_t . We will combine these two results to decide which of the “candidate” long term memory should be included in the actual long term memory.



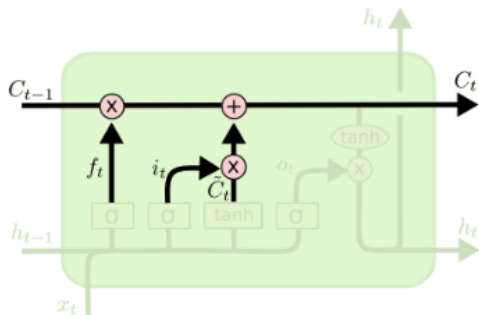
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Example

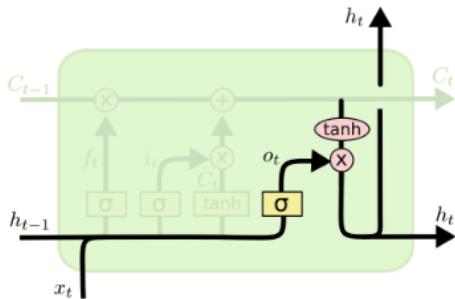
We might want to add the gender of the new subject to the long term memory, to replace the old one we're forgetting.

Now we update the long term memory by forgetting information and adding new information:



$$C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}$$

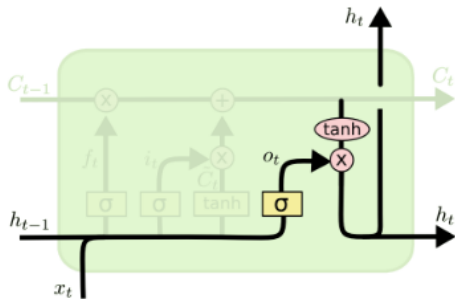
Finally, we use one last “forget gate” to compute o_t and compute the new hidden state by “forgetting” some information from our newly computed long term memory h_t :



$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \otimes \tanh(C_t)$$

Finally, we use one last “forget gate” to compute o_t and compute the new hidden state by “forgetting” some information from our newly computed long term memory h_t :



$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \otimes \tanh(C_t)$$

Example

For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

Some notes:

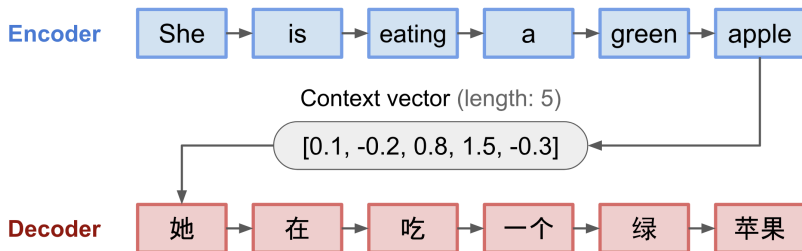
- One can show that the LSTM helps a lot (but doesn't entirely prevent) the vanishing gradient problem. Thus, LSTMs are basically always used instead of RNNs in practice.

Some notes:

- One can show that the LSTM helps a lot (but doesn't entirely prevent) the vanishing gradient problem. Thus, LSTMs are basically always used instead of RNNs in practice.
- This is a basic (“vanilla”) LSTM. There are many variants that exist. I *highly* recommend reading this blog post by Christopher Olah, (one of the founders of Anthropic, which is the company that makes Claude), which explains LSTMs and their variants in more detail. (I basically stole this entire section from his blog).

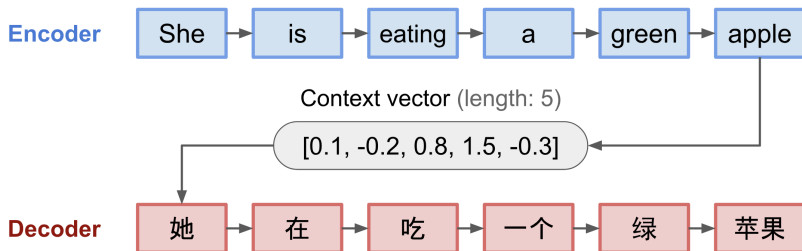
attention

Recall the encoder/decoder idea from earlier: to translate a sentence, we can use a seq2vec model to *encode* a sentence into a single vector (called a “context vector”), then we feed that single vector into a vec2seq model which spits out a the sentence in a new language:



attention

Recall the encoder/decoder idea from earlier: to translate a sentence, we can use a seq2vec model to *encode* a sentence into a single vector (called a “context vector”), then we feed that single vector into a vec2seq model which spits out a the sentence in a new language:



Problem: These context vectors are often bad at summarizing long sentences. In particular, they tend to “forget” about content at the beginning of the sentence.

Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .



Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .

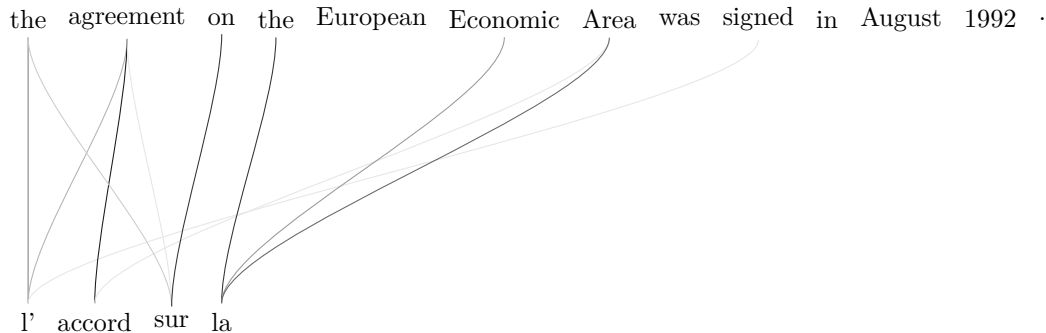


Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .

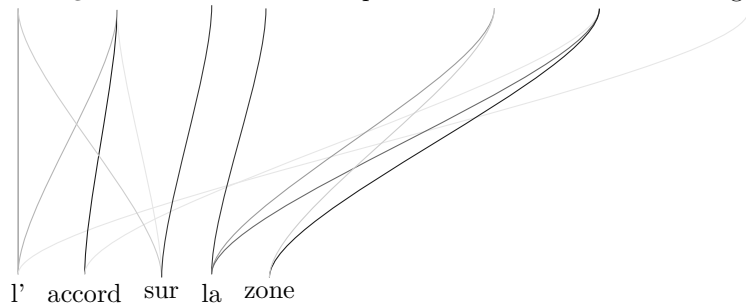


Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)



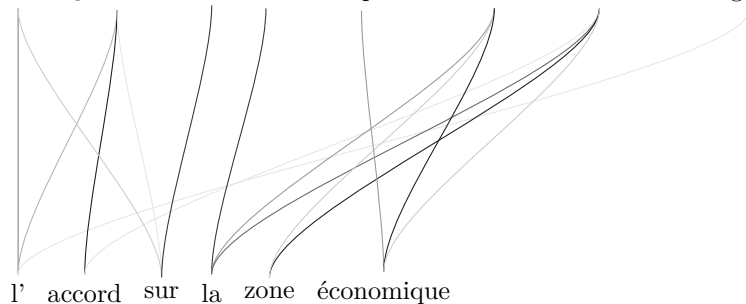
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .



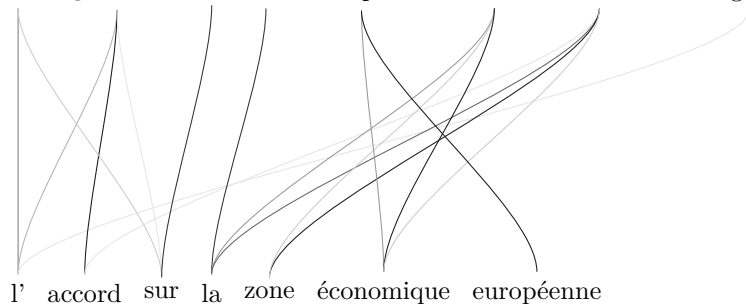
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .



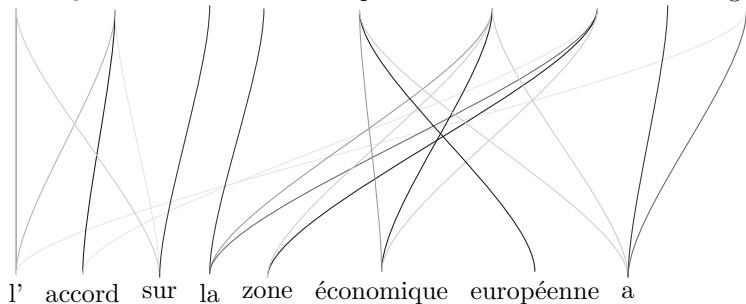
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .



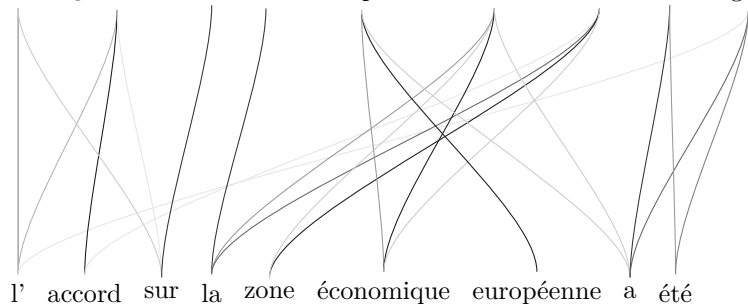
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .



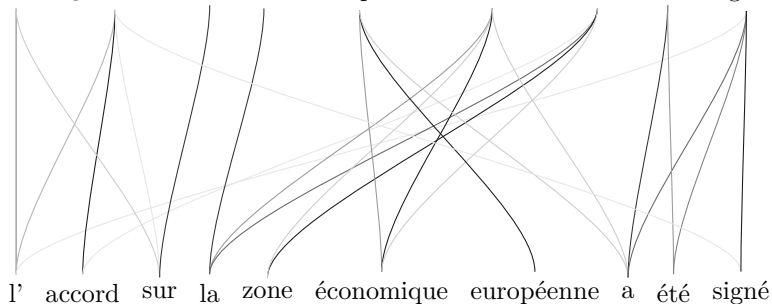
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

the agreement on the European Economic Area was signed in August 1992 .

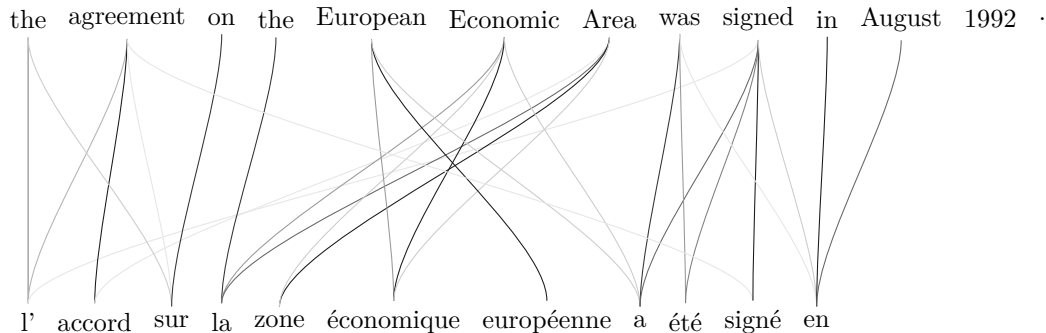


Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)

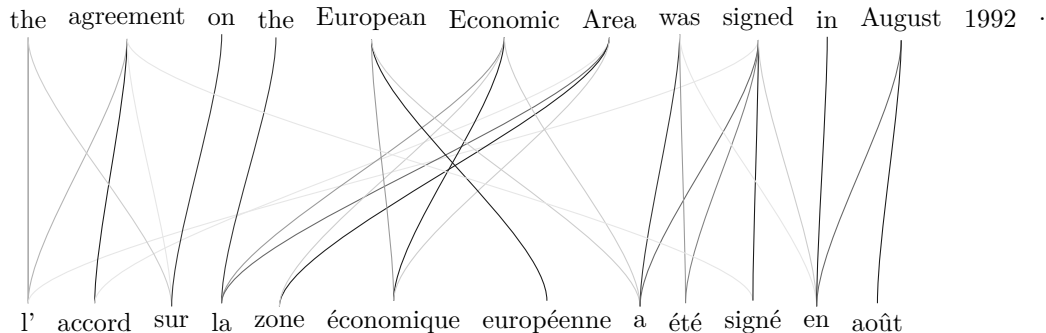
the agreement on the European Economic Area was signed in August 1992 .



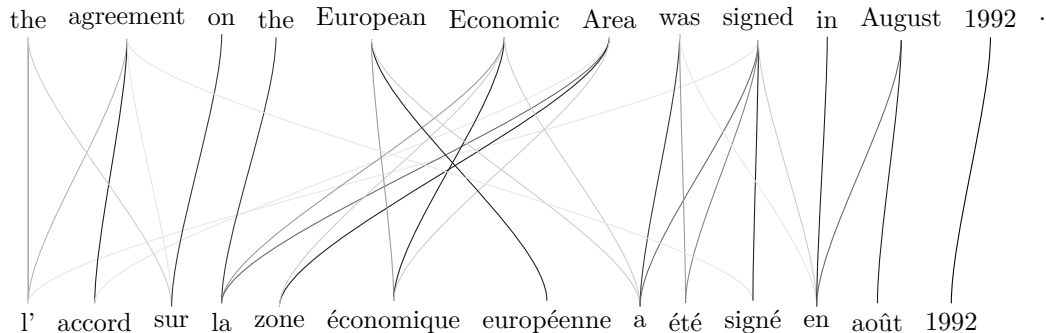
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)



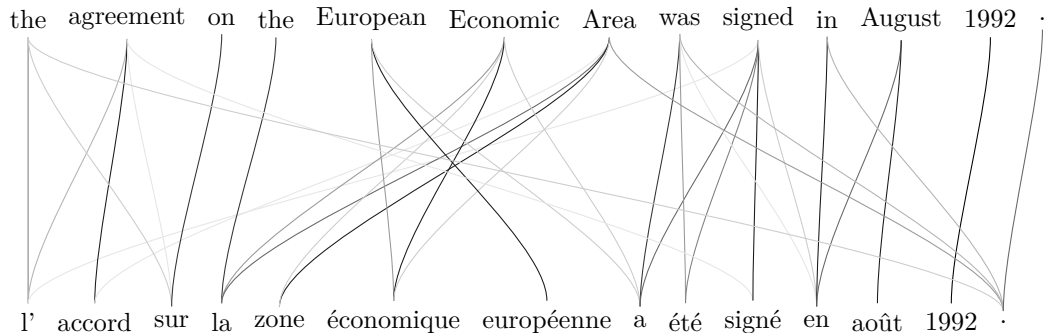
Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)



Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)



Idea: Our network should somehow look at how related the predicted words (french) are to each input word (english)



How can we build this in to our network?

How can we build this in to our network?

We will use the infamous “attention” mechanism.

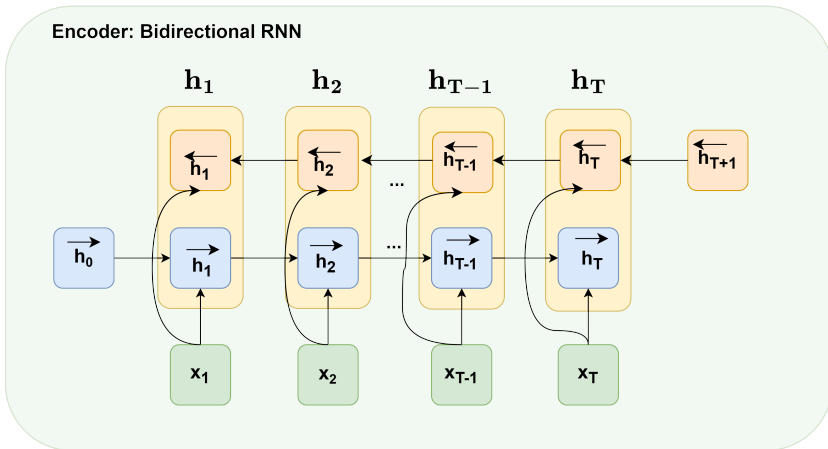
The idea is that our prediction for the next french word will depend on a weighted sum of all the words in the english sentence. English words that are more relevant will be assigned higher weight than those that are irrelevant.

Additive Attention: General Outline

Model: Encoder-Decoder with Bidirectional RNN Encoder

1. Encode each English word into its own vector using the hidden states of a bidirectional RNN.
2. Before predicting a French word, compute the relevance of each English word using the encodings from step 1.
3. Compute a context vector which is a weighted sum of English word encodings, where weights depend on results of step 2.
4. Feed context vector into a vec2seq model to predict new French word. This context vector will be used only for predicting *the current French word*. To predict the next word, we will compute a new context vector.

Step 1: Encode each input into a vector using the hidden states of a bidirectional RNN



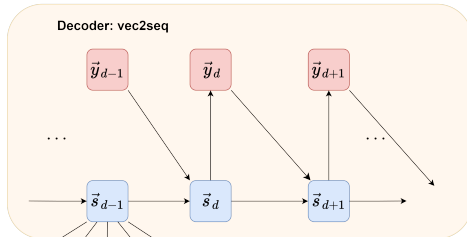
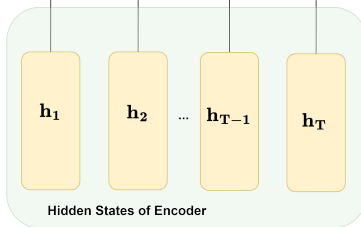
Step 2: Compute relevance of each input by comparing encoder hidden states to previous hidden state of decoder

Relevance Scores:

$\vec{e}_d =$



$$e_{dt} = \text{score}(\vec{s}_{d-1}, h_t)$$

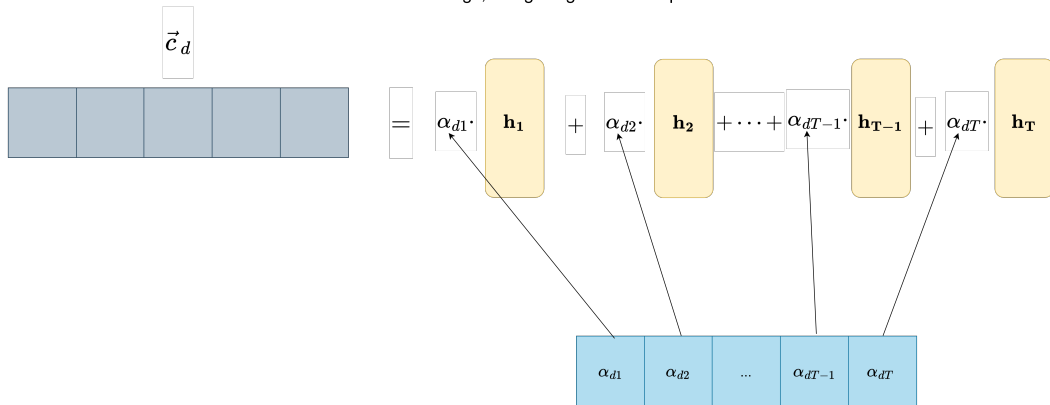


Step 2 (cont.): Use relevance scores to compute weights using softmax

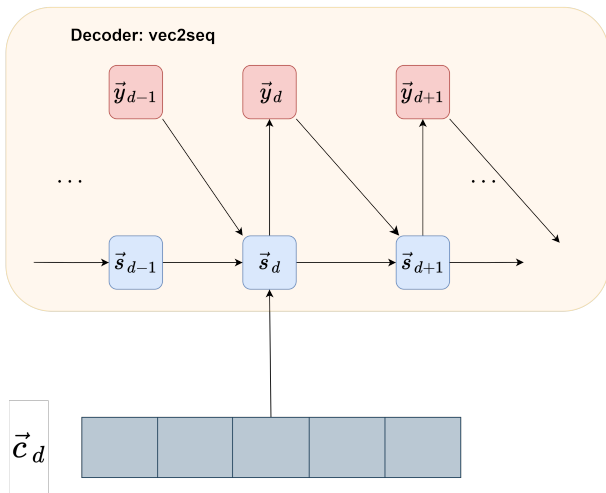
The diagram illustrates the computation of attention weights $\vec{\alpha}_d$ from relevance scores e_d . On the left, a horizontal row of five light blue boxes contains the relevance scores α_{d1} , α_{d2} , an ellipsis, α_{dT-1} , and α_{dT} . Above this row is a vertical white box containing the vector $\vec{\alpha}_d$. To the right of the blue boxes is an equals sign, followed by a vertical white box containing the word "softmax". To the right of "softmax" is a large opening parenthesis. Inside the parenthesis is a horizontal row of five light purple boxes containing the relevance scores e_{d1} , e_{d2} , an ellipsis, e_{dT-1} , and e_{dT} . The parenthesis closes after the last box. To the right of the closing parenthesis is another vertical white box.

$$\vec{\alpha}_d = \text{softmax} \left(\begin{matrix} e_{d1} & e_{d2} & \dots & e_{dT-1} & e_{dT} \end{matrix} \right)$$

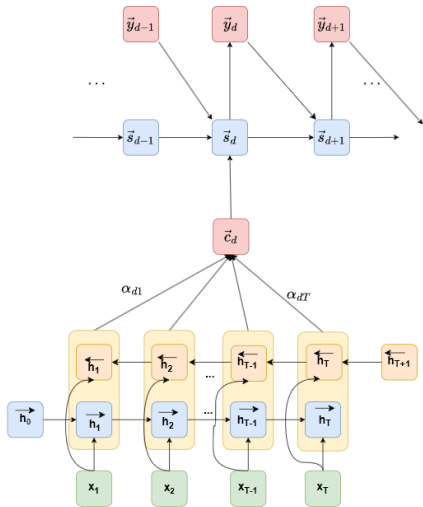
Step 3: Compute context vector as weighted sum of encodings, using weights from step 2



Step 4: Feed the context vector into the decoder to make the d-th prediction



Putting it all together



- $e_{dt} = \text{score}(\vec{s}_{d-1}, \vec{h}_t)$
- $\vec{e}_d = [e_{d1}, e_{d2}, \dots, e_{dT}]$
- $\vec{\alpha}_d = \text{softmax}(\vec{e}_d) = [\alpha_{d1}, \dots, \alpha_{dT}]$
- $\vec{c}_d = \sum_{t=1}^T \alpha_{dt} \vec{h}_t$

- $e_{dt} = \text{score}(\vec{s}_{d-1}, \mathbf{h}_t)$
 - Score that tells us “given that we already know $\vec{s}_{d-1}$, how important is $\mathbf{h}_t$ in predicting $\vec{s}_d$ ”

- $e_{dt} = \text{score}(\vec{s}_{d-1}, \mathbf{h}_t)$
 - ▶ Score that tells us “given that we already know $\vec{s}_{d-1}$, how important is $\mathbf{h}_t$ in predicting $\vec{s}_d$ ”
- $\vec{e}_d = [e_{d1}, e_{d2}, \dots, e_{dT}]$
 - ▶ Compute score between $\vec{s}_{d-1}$ and all encoder hidden states

- $e_{dt} = \text{score}(\vec{s}_{d-1}, \mathbf{h}_t)$
 - ▶ Score that tells us “given that we already know $\vec{s}_{d-1}$, how important is $\mathbf{h}_t$ in predicting $\vec{s}_d$ ”
- $\vec{e}_d = [e_{d1}, e_{d2}, \dots, e_{dT}]$
 - ▶ Compute score between $\vec{s}_{d-1}$ and all encoder hidden states
- $\vec{\alpha}_d = \text{softmax}(\vec{e}_d) = [\alpha_{d1}, \dots, \alpha_{dT}]$
 - ▶ Turns vector of scores into weights, where higher scores get higher weights.

- $e_{dt} = \text{score}(\vec{s}_{d-1}, \mathbf{h}_t)$
 - ▶ Score that tells us “given that we already know $\vec{s}_{d-1}$, how important is $\mathbf{h}_t$ in predicting $\vec{s}_d$ ”
- $\vec{e}_d = [e_{d1}, e_{d2}, \dots, e_{dT}]$
 - ▶ Compute score between $\vec{s}_{d-1}$ and all encoder hidden states
- $\vec{\alpha}_d = \text{softmax}(\vec{e}_d) = [\alpha_{d1}, \dots, \alpha_{dT}]$
 - ▶ Turns vector of scores into weights, where higher scores get higher weights.
- $\vec{c}_d = \sum_{t=1}^T \alpha_{dt} \mathbf{h}_t$
 - ▶ Compute *unique* context vector for prediction d , where hidden states $\mathbf{h}_t$ with higher scores get higher weight (influence) on the result.

In the paper, they compute scores as

$$e_{dt} = \text{score}(\vec{s}_{d-1}, \mathbf{h}_t) = \vec{v}_a^T \tanh(W_a[\vec{s}_{d-1}; \mathbf{h}_t])$$

where $\vec{v}_a$ and W_a are learned weights.

In the paper, they compute scores as

$$e_{dt} = \text{score}(\vec{s}_{d-1}, \mathbf{h}_t) = \vec{v}_a^T \tanh(W_a[\vec{s}_{d-1}; \mathbf{h}_t])$$

where $\vec{v}_a$ and W_a are learned weights.

This is known as *additive attention*, but many kinds exist.

What was the point of all this?

- Before, we used our encoder network to predict a *single* context vector summarizing the entire sentence which we fed as input to the network.
- Now, to predict token d , we use our encoder network to compute a *unique context vector* $\vec{c}_d$
- The value of this context vector is a weighted sum of the outputs of the encoder $\mathbf{h}_1, \dots, \mathbf{h}_T$, where the weight of each $\mathbf{h}_t$ depends on how related it is to $\vec{s}_{d-1}$
- In a sense this tells the decoder which parts of the input to pay *attention* to when making a prediction

bibliography & further reading

- StatQuest: Recurrent Neural Networks (RNNs), Clearly Explained!!! - <https://youtu.be/AsNTP8Kwu80?si=ly0IqlC5ftzp1U0F>
- Probabilistic Machine Learning: An Introduction (Chapter 15.2) - <https://probml.github.io/pml-book/book1.html>
- Stanford CS224N - Lecture 7: Vanishing Gradients and Fancy RNNs - <https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture07-fancy-rnn.pdf>
- Dive into Deep Learning (Chapters 9.4, 10.1) - <https://d2l.ai/index.html>
- The Unreasonable Effectiveness of Recurrent Neural Networks - <https://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- Understanding LSTM Networks - <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- Attention? Attention! - <https://lilianweng.github.io/posts/2018-06-24-attention/#whats-wrong-with-seq2seq-model>