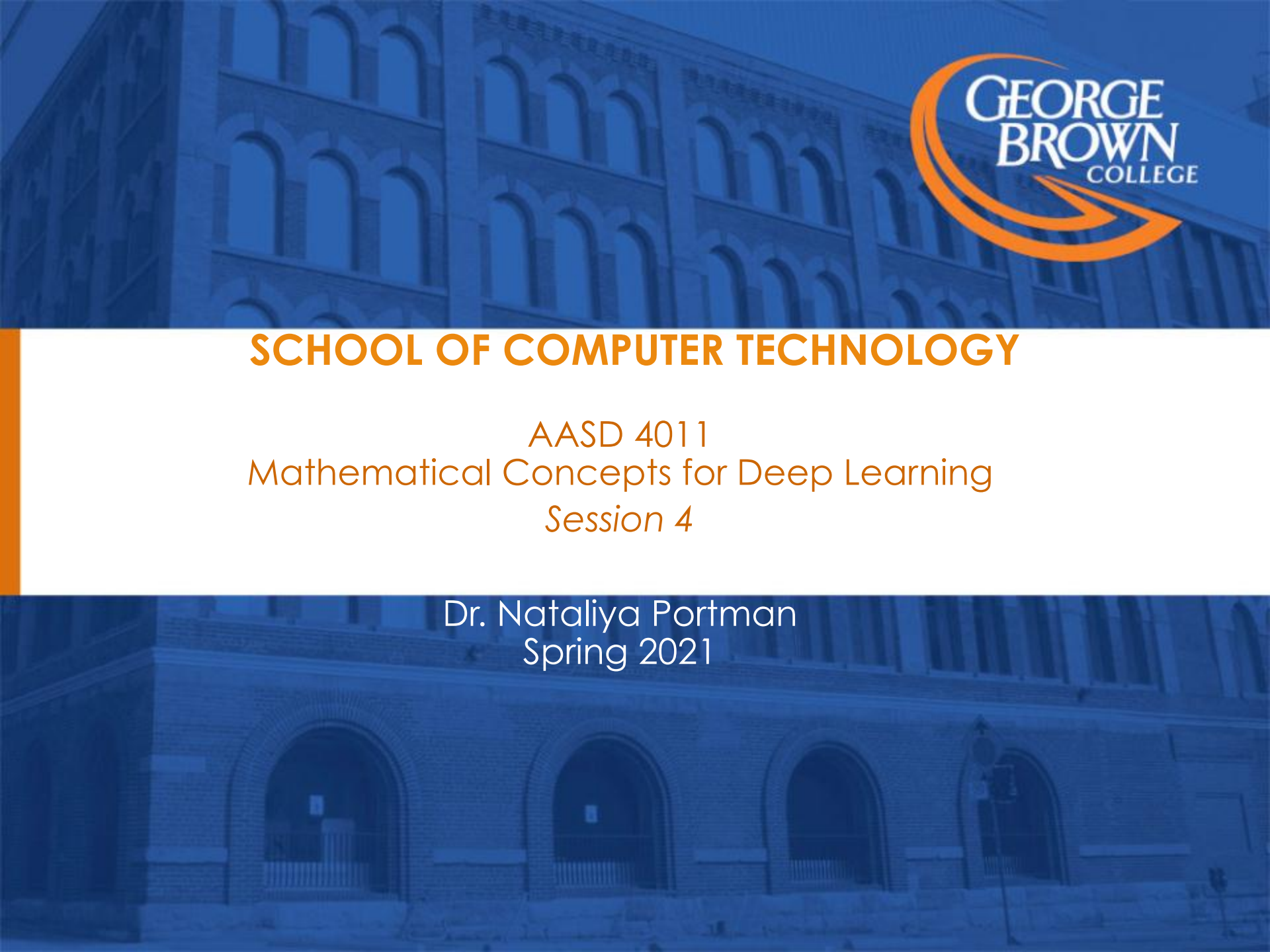


SCHOOL OF COMPUTER TECHNOLOGY

AASD 4011
Mathematical Concepts for Deep Learning
Session 4

Dr. Nataliya Portman
Spring 2021

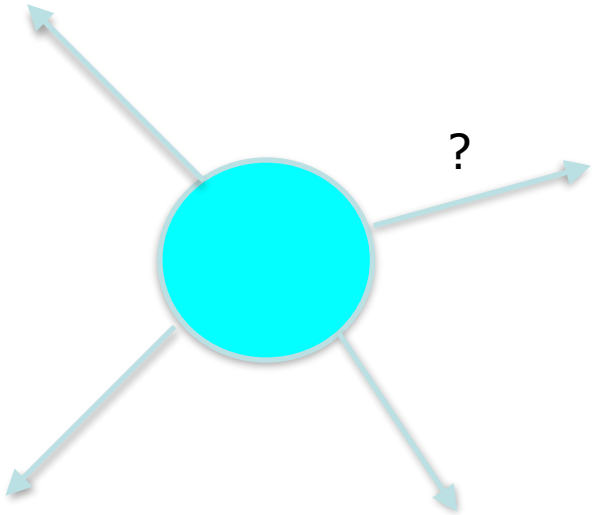
Overview

- Intro to Recurrent Neural Networks (RNN)
- Intuitive understanding of RNNs
- Vanishing and exploding gradient problems
- LSTMs

Motivation

RNNs solve problems involving sequential processing of data. This kind of tasks require a different NN architecture. What do we mean by sequential processing?

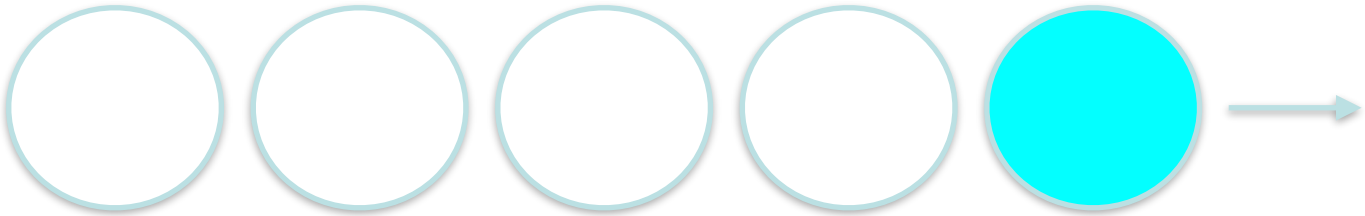
Suppose we have a picture of a ball and we want to predict where it will go next.



Without a prior information on the ball trajectory history or understanding the dynamics of its motion, our answer will be merely a guess.

Motivation

Suppose that in addition to the current position of the ball, we have the history of its previous locations. Now, the problem becomes much easier.

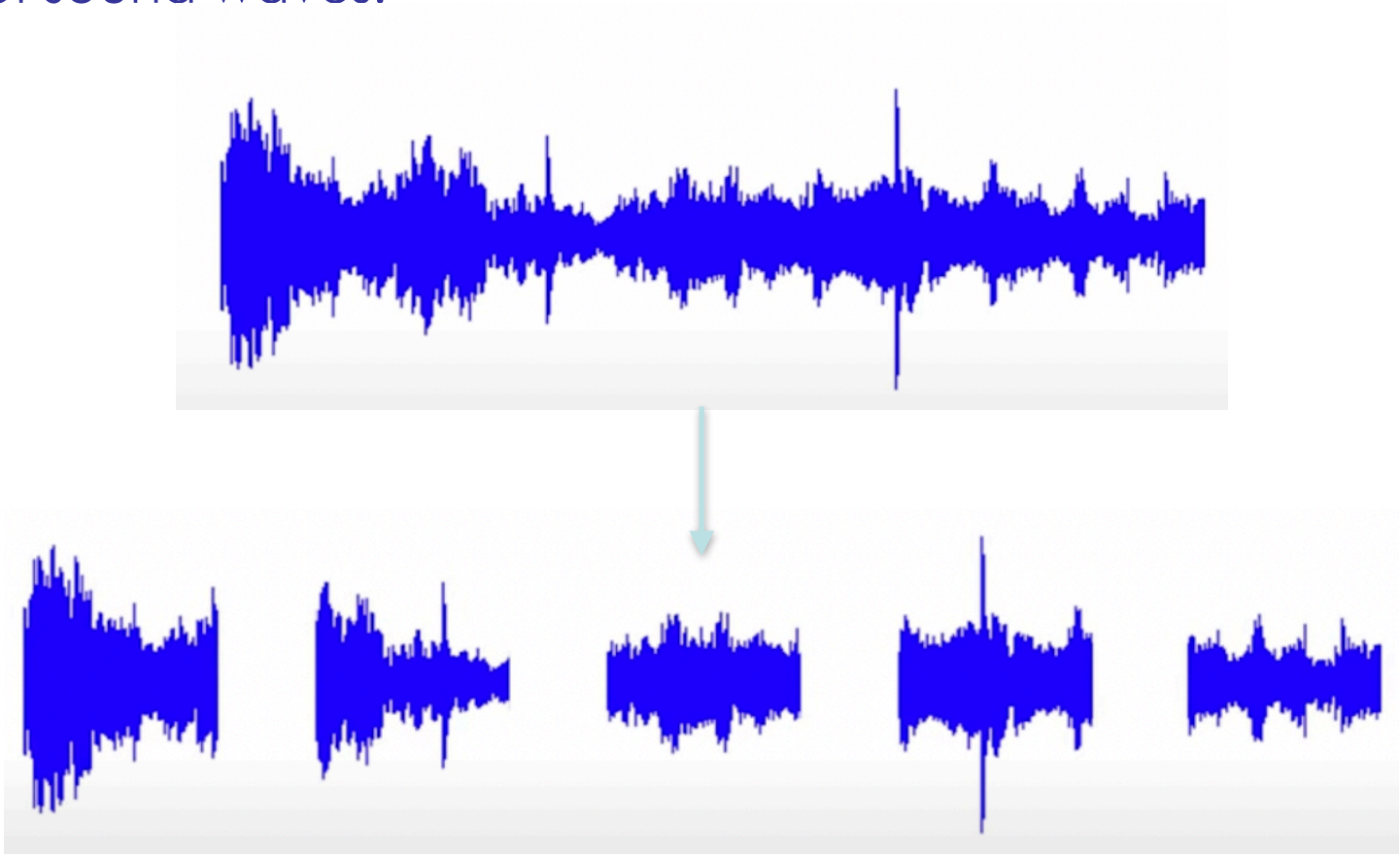


Now, we have a sense of where the ball is going to move to next and what we need for sequential modelling and sequential prediction.

Sequential data and these types of problems are all around us.

Motivation

Audio signal from the speech can be split up to a sequence of sound waves.



Motivation

And text can be split up into a sequence of different characters or words.

6.S191 Introduction to Deep Learning

Characters: 6 . S 1 9 1

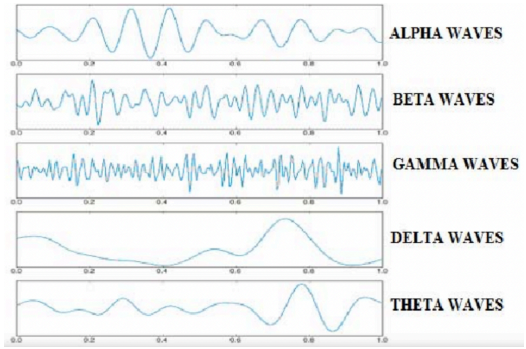
Words: Introduction to Deep Learning

Each of these individual characters or words can be thought of as a time step in a sequence.

There are many more cases where sequential processing may be useful.

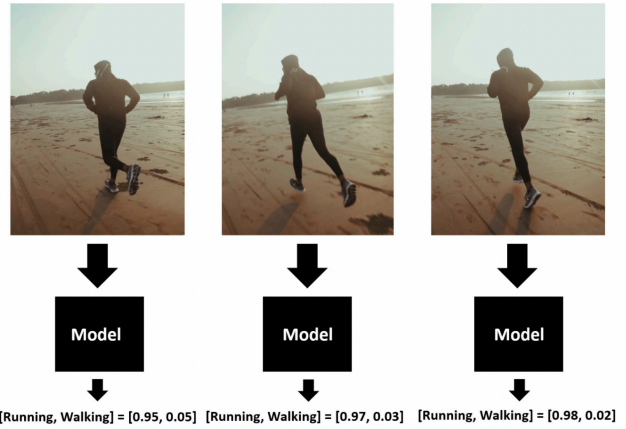
Motivation

Sequence modelling applications: neurological disorder diagnostics, financial time series prediction, speech recognition, music generation, sentiment classification, machine translation and video activity recognition.



"I love this movie.
I've seen it many times
and it's still awesome."

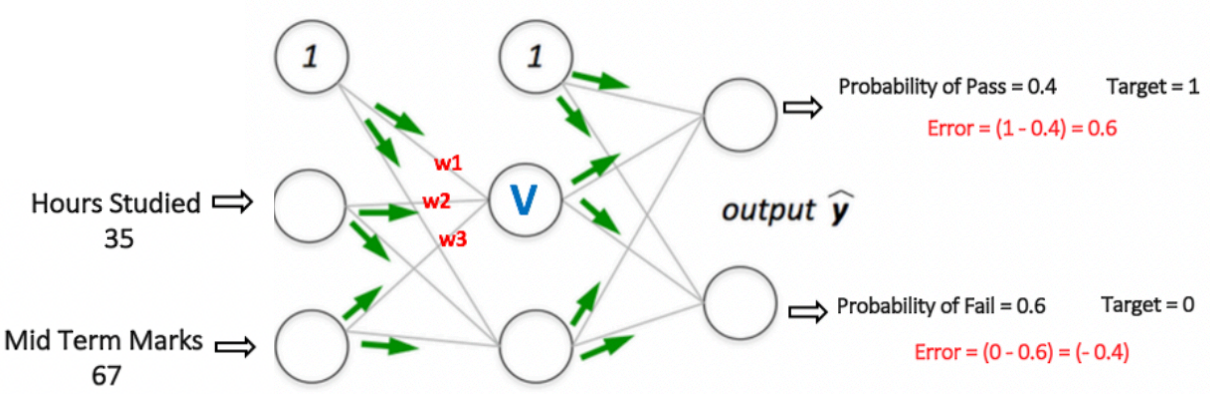
"This movie is bad.
I don't like it at all.
It's terrible."



Sequence modeling

Let's look at some concrete applications in which sequence modelling plays an important role.

We have seen feedforward NNs that work in a one-to-one manner, going from a fixed and static input to a fixed and static output.



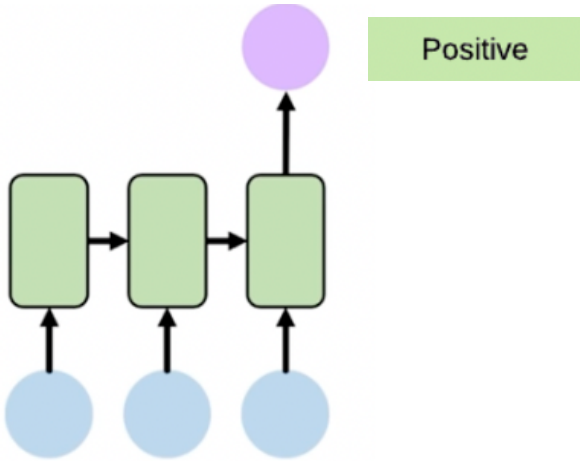
One-to-one

Feedforward NN for prediction of student fail or pass.

There is no inherent notion of sequential data or a time component. When we consider sequence modelling, we expand the range of possibilities to use cases that involve temporal inputs and outputs as well.

Sequence modeling

Let's consider the language processing problem where there is a sentence that is an input to our model. And that defines a sequence where the words are individual time steps. And in the end, our task is to predict one output which is going be a sentiment or a feeling associated with that sequence input.



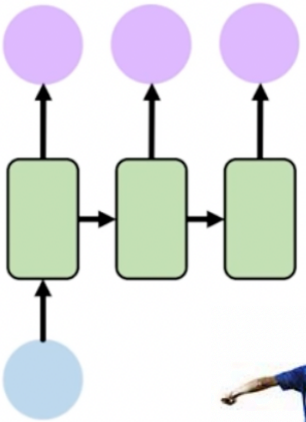
@DinahLady I too, liked the movie! I want to buy the DVD when it comes out

"Many-to-one" NN architecture for sentiment classification.

Sequence modeling

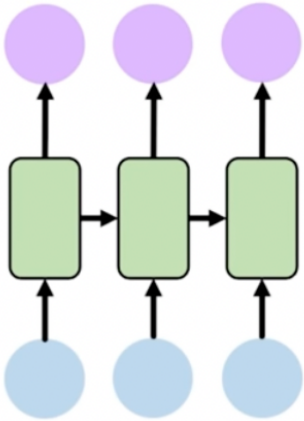
We can also consider the converse case when our input does not have the time dimension. For example, when we have a static image and our task is to produce a sequence that describes the content of the image such as an image caption that is a sequence of words. Another example is a "many-to-many" scenario, when we have to translate one sentence that is a sequence of words to another.

Image caption: A football player kicks the ball

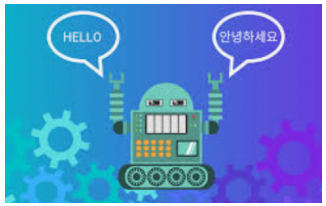


"One-to-many"

Machine translation



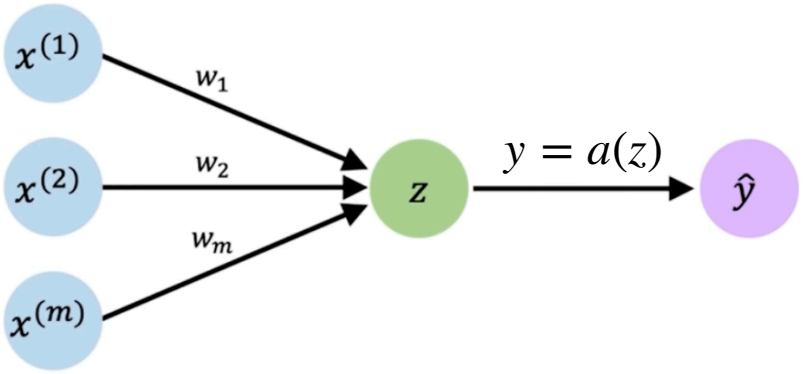
"Many-to-many"



Neurons with recurrence

How can we build NN to tackle this kind of problems? How can we add this temporal dimension to a model? We need to build a fundamental understanding of what changes need to be made to NN in order to tackle sequential data.

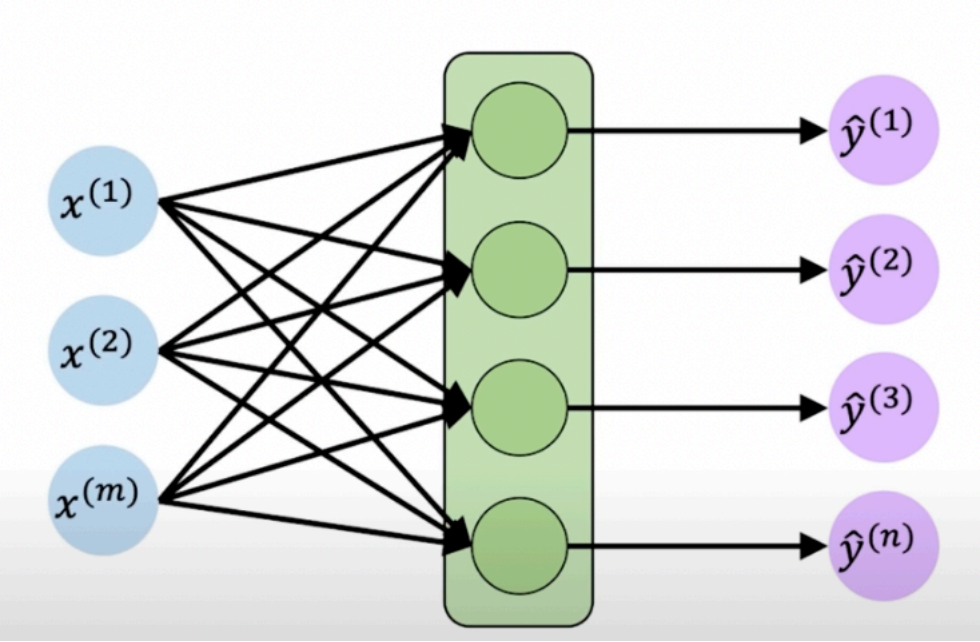
We start by revising the perceptron model.



Now, think of these inputs as being from a single time step in your sequence.

Neurons with recurrence

We also saw how we could build a complex neural network consisting of layers of perceptrons yielding a multidimensional output.



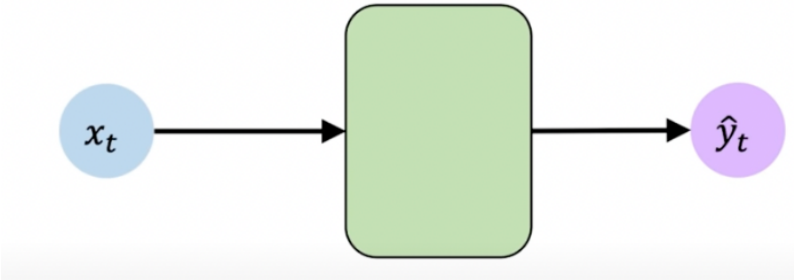
Feedforward neural network

Does this NN have a notion of time or a sequence?

Our inputs and outputs come from a fixed time step in a sequence. Moreover, they are independent.

Neurons with recurrence

Let's simplify this diagram. To do that we will hide the hidden layer in this green box.



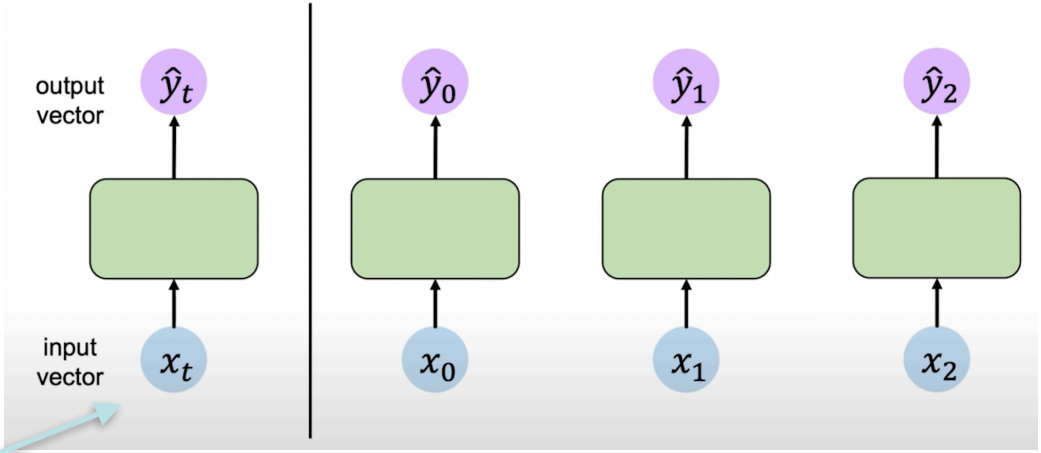
Our inputs are going be vectors of length m and outputs - vectors of length n. Here, we are considering an input at a specific time t which is nothing different from what we saw in the first session.

Even with this simplified feedforward NN model, we could naively feed the sequence to this model, and apply it repeatedly, once for each element x_t in the temporal sequence $\{x_{t1}, x_{t2}, \ldots, x_{tm}\}$.

Individual time steps

How can we handle these individual inputs across different time steps? Since we have a sequential data, we do not have just a single time step, we have multiple individual time steps which start at time $t = 0$. We can treat x_0 as an isolated point in time and feed it into our model to predict the output. And then we can take another input x_1, treat it as another isolated point in time, train a model to predict the corresponding output and repeat training for x_2. All of these models are just replicas of each other with different inputs at different time points.

We know that $\hat{y}$ is going to be a function of x_t at that time step.

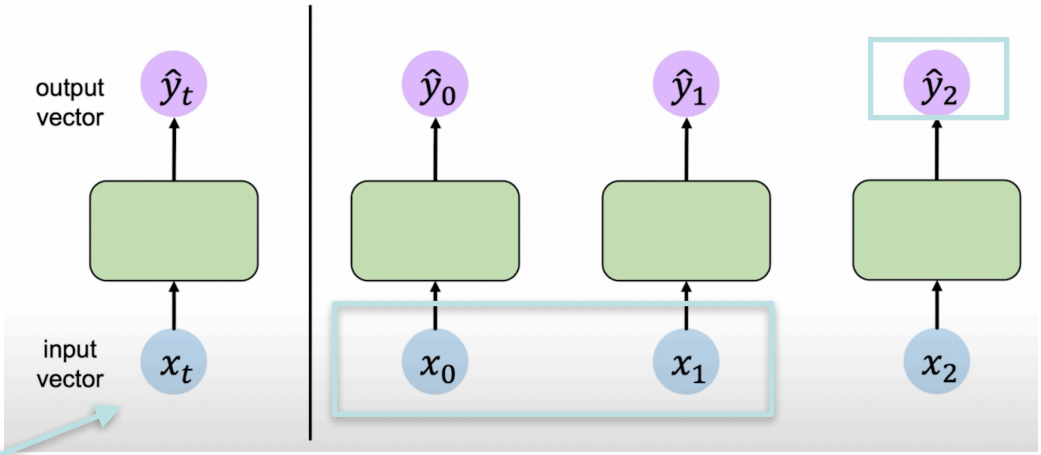


Input as a fixed time point

$$\hat{y}_t = F(x_t)$$

Individual time steps

If we are considering sequential data, then the label $\hat{y}_2$ at a later time step is going to be dependent on the inputs at the previous time steps. What we are missing here by treating these individual time steps as individual isolated time steps is the relationship that is inherent to sequence data between time steps that happened earlier in the sequence and what we want to predict later. How do we address this?



Input as a fixed time point

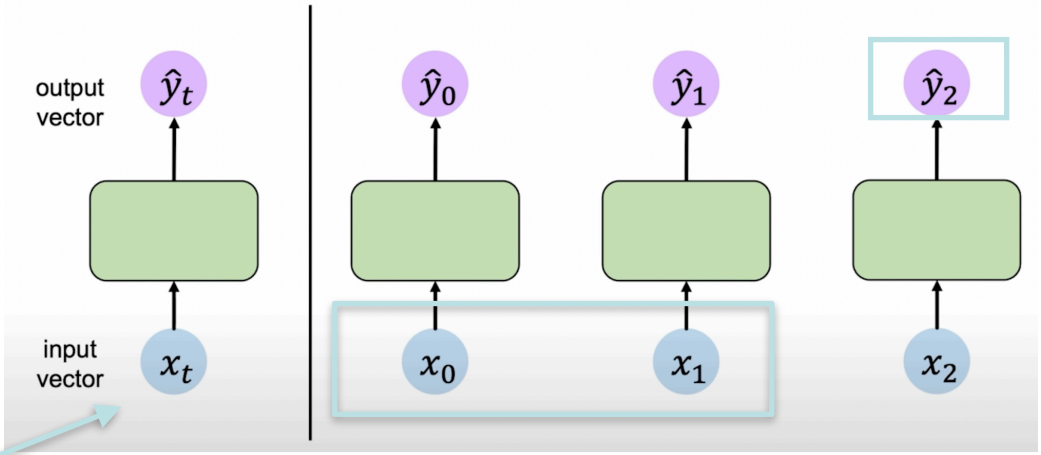
$$\hat{y}_t = F(x_t)$$

Individual time steps

What we need is to relate the way the neuronal operations that the network is doing at particular time step to both,

1. The prior history of network computation from previous time steps,
2. The input from these time steps.

We also need the sense of forward looking, to be able to pass current information onto future steps.



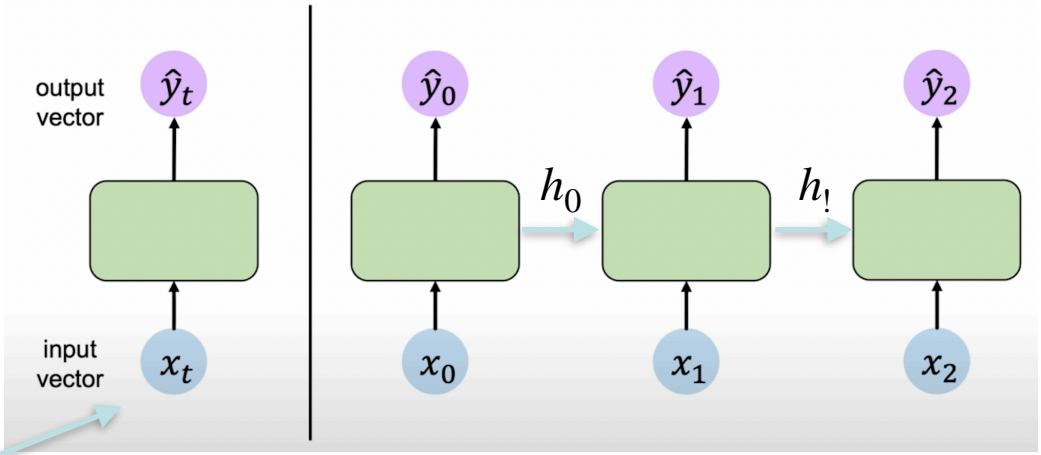
Input as a fixed time point

$$\hat{y}_t = F(x_t)$$

Individual time steps

We will consider linking of computation and information at different time steps to each other. Specifically, we will introduce this *internal memory* or a *cell state* that we denote here as h_t that is going to be maintained by the neurons and the network itself. And this state is going be passed from one time step to another in the course of time.

The key idea here is that by having this recurrence relation we are capturing a part of the memory of what the sequence looks like. This implies that the network output predictions and computations are not only a function of the input at a particular time step, but also the passed memory of the cell state.



Output depends on current inputs as well as the past computations and the past learning that occurred.

Input as a fixed time point

$$\hat{y}_t = F(x_t, h_{t-1})$$

F is a mapping that NN learns through standard neuronal operations.

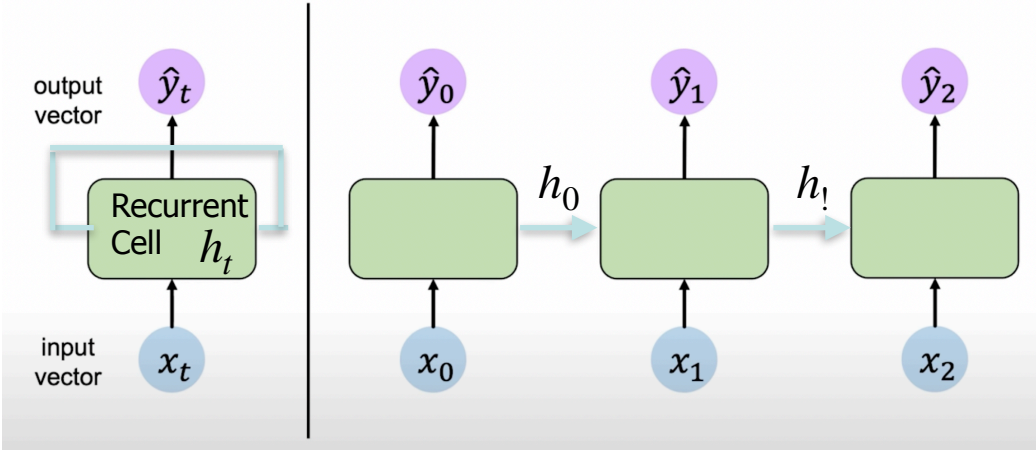
Neurons with recurrence

Our output will depend not only current input but also on the past memory.

$$\hat{y}_t = F(x_t, h_{t-1})$$

Output. Input Past Memory

Because the output is a function of the input and the past memory, this means that we can describe these neurons by a recurrence relation which means that we have the cell state that depends on the current input and prior cell states.



The diagram on the right shows individual time steps rolled out across time. The diagram on the left shows the same relationship as the cycle (shown by the loop) illustrating the concept of a recurrence relation.

Recurrent Neural Networks (RNN)

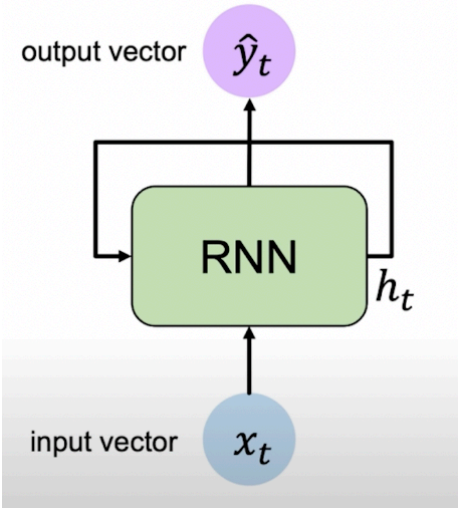
This is exactly this idea of recurrence that provides intuition of the key operations of the recurrent NN. We further build up our understanding of the mathematics of these recurrent relations and the operations that define RNN behaviour.

RNNs maintain the internal state, h_t, which is updated at each time step as the sequence is processed. And this is done by this *recurrence relation* which defines how the state is updated after each time step:

$$h_t = f_W(x_t, h_{t-1})$$

Internal cell state

Function with weights W



Weights W are unknown parameters that RNN will learn over the course of training. f_W will take as an input both, x_t and a prior state h_{t-1}.

How do we define and find this function?

Recurrent Neural Networks

f_W is parameterized by a set of weights that are going to be learned by RNN in the course of training.

Key feature of RNN: The same function and the set of parameters are applied at every time step of processing the sequence.

Weights are going to change over time and over the course of training. But at each iteration of the SGD algorithm, the same set of weights is going to be applied to each individual time step in the sequence.

Let's look at the algorithm of updating RNNs to gain a better understanding of how these networks work. Suppose our task is to predict the next word in a sentence. Firstly, we initialize our network, a hidden state and a sentence.

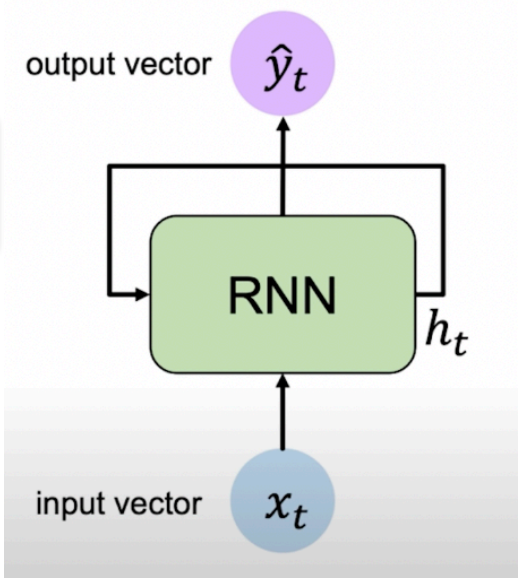
```
my_rnn=RNN()
hidden_state=[0,0,0,0]
sentence=["I","love","recurrent","neural"]
```

Recurrent Neural Networks

The RNN algorithm is executed as follows:

```
for words in sentence:
    prediction, hidden_state =my_rnn(word, hidden_state)

next_word_prediction=prediction
```



It loops through the words in the sentence,
and at each time step it takes the current word
and the hidden state at the previous time step.
And this is going to generate the prediction for the
next word as well as the update for the hidden state itself.

And finally, when these four words in a sentence are processed, RNN can generate the prediction of what the next word is by considering the RNN output after all the individual words have been fed through the model.

Mathematics of the hidden state

The RNN computation includes both, the internal cell state update to h_t as well as prediction itself.

Now we are going to walk though how each of these quantities is defined.

Output vector

$$\hat{y}_t = \boldsymbol{W}_{\boldsymbol{hy}}^{\boldsymbol{T}} h_t$$

Update hidden state

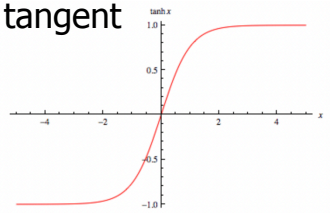
$$h_t = \tanh(\boldsymbol{W}_{\boldsymbol{hh}}^{\boldsymbol{T}} h_{t-1} + \boldsymbol{W}_{\boldsymbol{xh}}^{\boldsymbol{T}} x_t)$$

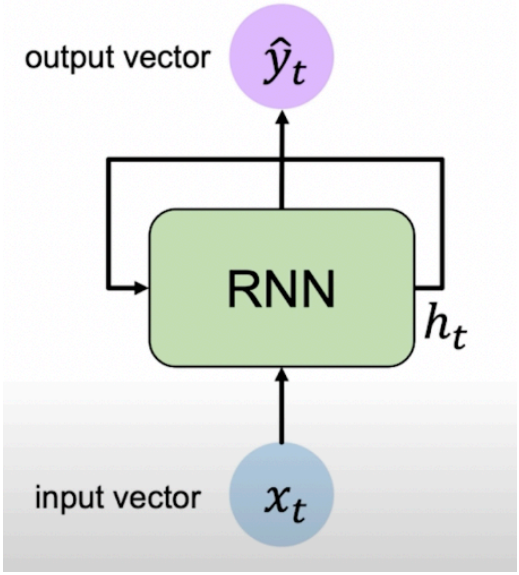
Input vector

x_t

To generate the output, we take that hidden state and multiply it by a separate weight matrix which produces a modified version of h_t .

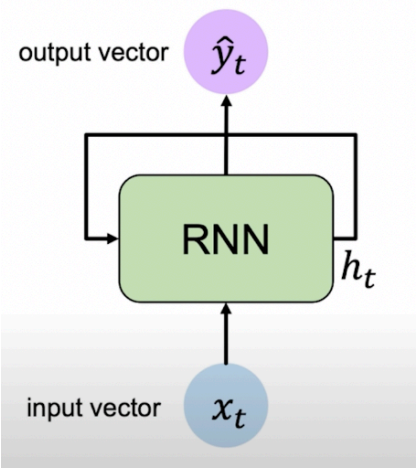
Hyperbolic tangent





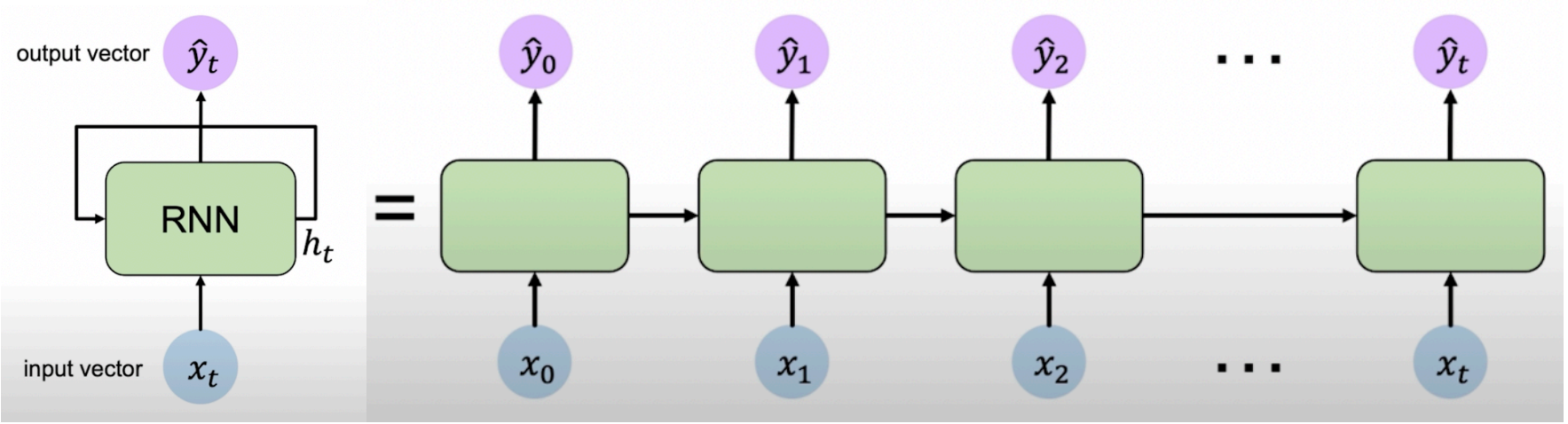
RNN: Computational Graph

We have seen that RNNs have internal loops that feedback on themselves. We have also seen how RNN loops have been realized in the course of time.



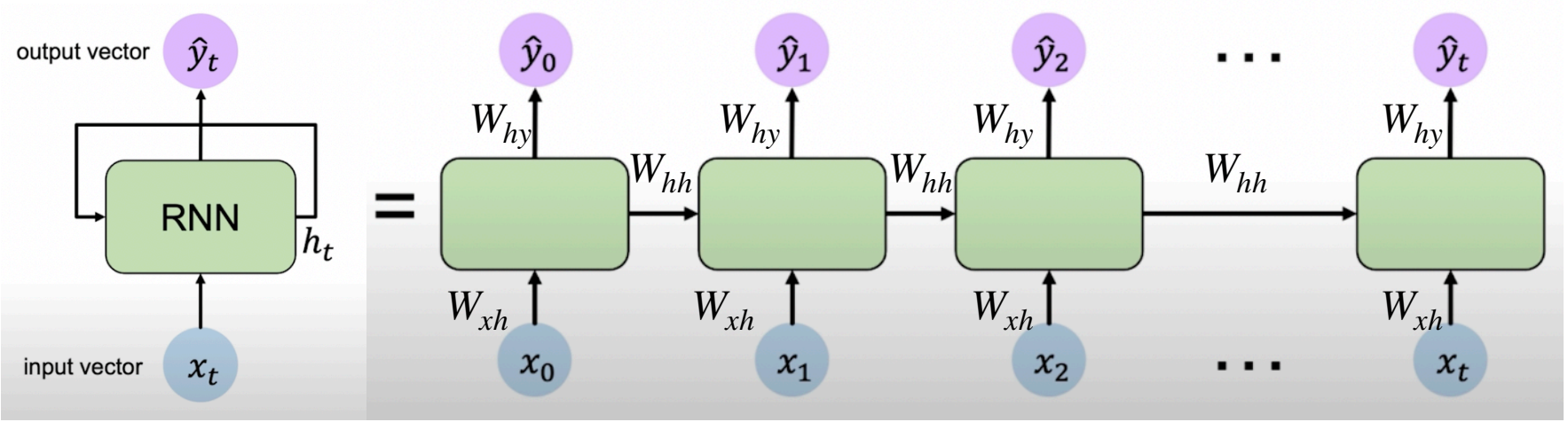
RNN: Computational Graph

We start from the first time step and continue to roll out the network across time up until the time step t.



RNN: Computational Graph

We can also make explicit weight matrices W_{xh}, starting from the weight matrix that defines how the inputs at each time step are being transformed in the hidden state computation. There are also weight matrices W_{hh} that define the relationship between prior hidden state and current hidden state.

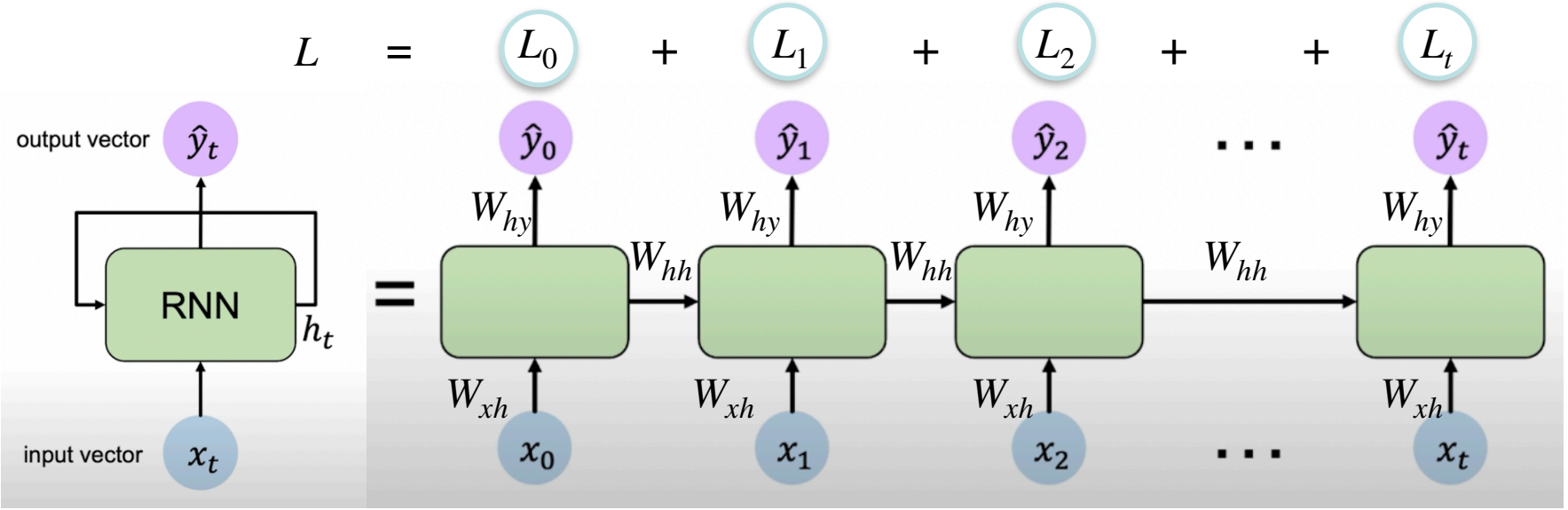


And finally, there are weight matrices W_{hy} that transform the hidden state to the output at a particular time step.

RNN reuses the same wight matrices at every time step.

RNN: Forward pass

When we make the forward pass through the network, we are going to generate outputs at each of those time steps. And from those individual outputs we can calculate the value for the loss function. Then we can sum all the individual losses to calculate the total loss that is used to train RNN.



So far, we have built a mathematical foundation on how to make a forward pass through RNN.

RNN implementation

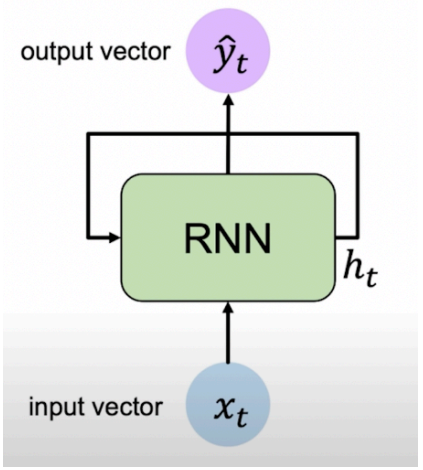
Here is a piece of Python code that shows how to implement RNN from scratch using Tensorflow (tf). RNN is defined using a layer

```
class MyRNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, input_dim, output_dim):
        super(MyRNNCell, self).__init__()

        # initialize wight matrices
        self.W_xh=self.add_weight([rnn_units,input_dim])
        self.W_hh=self.add_weight([rnn_units, rnn_units])
        self.W_hy=self.add_weight([output_dim, rnn_units])

        # initialize hidden state to zeros
        self.h=tf.zeros([rnn_units, 1])
```

```
def call(self, x):
    # update the hidden state
    self.h=tf.math.tanh(self.W_hh*self.h+self.W_xh*self.x)
    # compute the output
    output=self.W_hy*self.h
    # return the current output and the hidden state
    return output, self.h
```



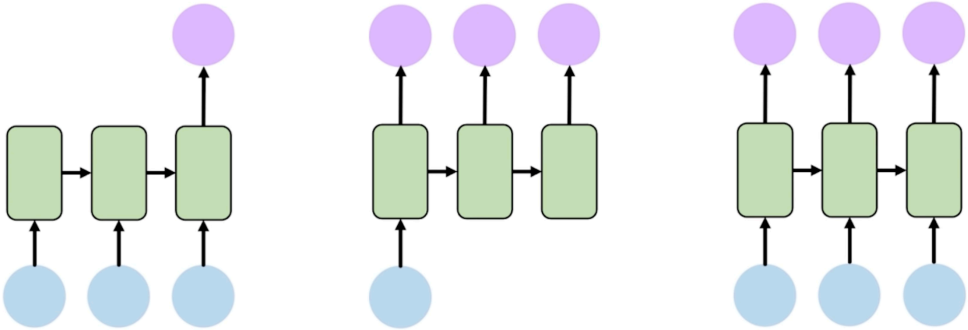
Call function is important as it describes how we can make a forward pass

Summary

Tensorflow has already implemented these type of RNN cell for us:

```
tf.keras.layers.SimpleRNN(rnn_units)
```

- RNNs are particularly well-suited for sequential data
- Typical RNN architectures are one-to-many, many-to-one and many-to-many



Beyond this, we can extend RNNs to many other applications in which sequential processing and sequential modelling may be useful.

Design criteria for sequence modelling

To really appreciate why RNNs are so powerful, we should consider a set of design criteria that we should keep in mind when thinking about sequence modelling problems. We need to ensure that our RNN

- Handles variable length sequences (not all sentences are going to have the same length),
- Tracks long-term dependencies and has a notion of memory,
- Maintains information about order (how are events that happened earlier in the sequence are going to affect later events),
- Has the above mentioned properties due to parameter sharing across the sequence.

RNNs do meet these sequence modelling criteria!

RNN application: Next word prediction

To understand these criteria, we will consider the following problem: Given a series of words in a sequence, our task is to predict the most likely next word.

Suppose we have this sentence as an example:

"This morning I took my cat for a walk".

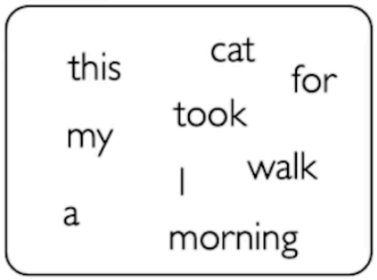
Given these words | Predict the next word

Our goal is to build a RNN to solve this problem. What is our first step? Before we get started with RNN modeling, we have to represent language to a Neural Network in a suitable(numerical) format. We need to represent words as arrays of numbers or vectors to enable mathematical operations of NN.

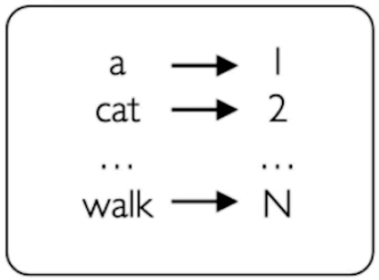
How exactly are we going to encode input words? The idea is to use *embedding* or transforming a set of identifiers for objects or indices into a vector of fixed length that captures exactly the content of the input.

Word embedding

One way of doing this is by generating sparse and binary vectors of the length equal to the number of unique words in our vocabulary. An alternative is learning word embeddings via Skip-gram NN model.

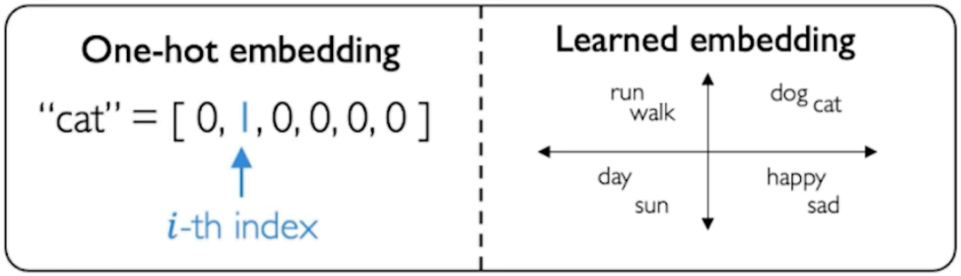


1. Vocabulary:
Corpus of words



2. Indexing:
Word to index

The nature of the word is encoded by its index.



3. Embedding:
Index to fixed-sized vector

Skip-gram model: Take the index mapping, feed it into a model that transforms this mapping across all the words in our vocabulary to a vector of a lower-dimensional space where Its components are learned

Handling variable sequence lengths

The first capability that we desired is handling variable sequence length. We can encounter very short sentences like the one below where words driving the meaning of our prediction are going to be very close to one another.

"The food was great".

We can also have a longer sequence

"We visited a restaurant for lunch"

and even longer sequence

"We were hungry but cleaned the house before eating"

where the information that is needed to predict the last word occurs in the sequence much earlier on.

The requirement that our RNN model should meet is the ability to handle such a variety of lengths.

Handling variable sequence lengths

Feedforward networks are unable to do this because they handle inputs of fixed dimensionality, and these inputs are passed on to the next layer.

In contrast, RNNs are able to handle variable sequence lengths. This is because the differences in sequence lengths are simply the differences in a number of time steps that are going to be input and processed by RNN.

So, RNNs meet the first design criterion.

Model Long-Term Dependencies

The following example shows that we will need a much earlier information in the sequence in order to predict the next word.

"France is where I grew up, but now I live in Boston and speak fluent __"

RNNs are able to achieve this because they have their way of updating the internal cell state by the recurrent relation discussed previously which incorporates the information from the past into the cell.

RNNs are able to achieve this criterion!

Differences in sequence order

Differences in sequence order could result in differences in overall meaning or property of the sequence. Here is an example of two sentences that have opposite semantic meaning but the same words:

"The food was good, not bad at all"

vs.

"The food was bad, not good at all"

The cell state is maintained by RNN, it depends on its past history which helps us maintain these sorts of differences, because we are holding information about its past history. We are also reusing the same weight matrices across each individual time step in our sequence to ensure that the information gets passed on to the current cell state.

Summary

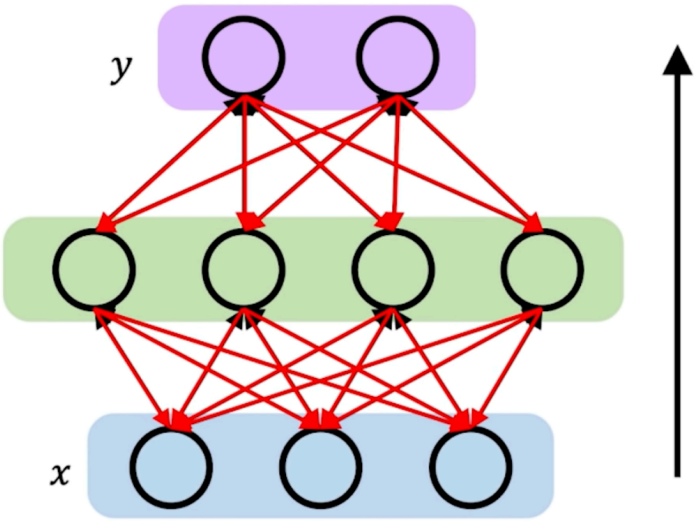
- Sequential data can be broadly represented and encoded into an RNN model
- RNNs can achieve all sequence modelling design criteria.

Specifically, RNN

- Handles variable length sequences (not all sentences are going to have the same length),
- Tracks long-term dependencies and has a notion of memory,
- Maintains information about order (how are events that happened earlier in the sequence are going to affect later events),
- Has the above mentioned properties due to parameter sharing across the sequence.

Backpropagation through time

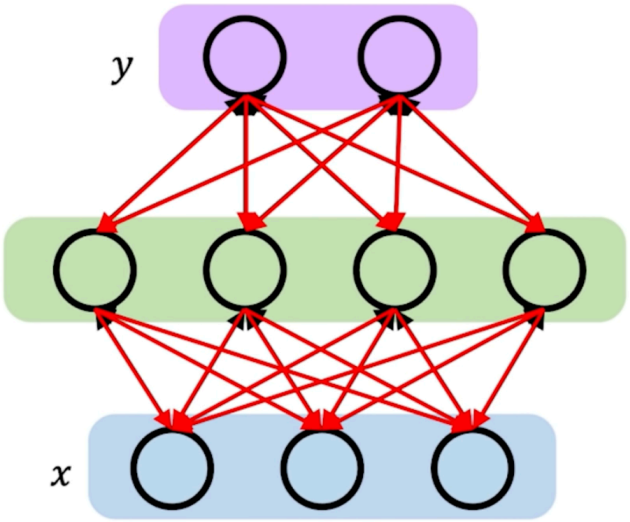
Recall how we used backpropagation for feedforward NN



Forward pass through the network

Backpropagation through time

Recall how we used backpropagation for feedforward NN

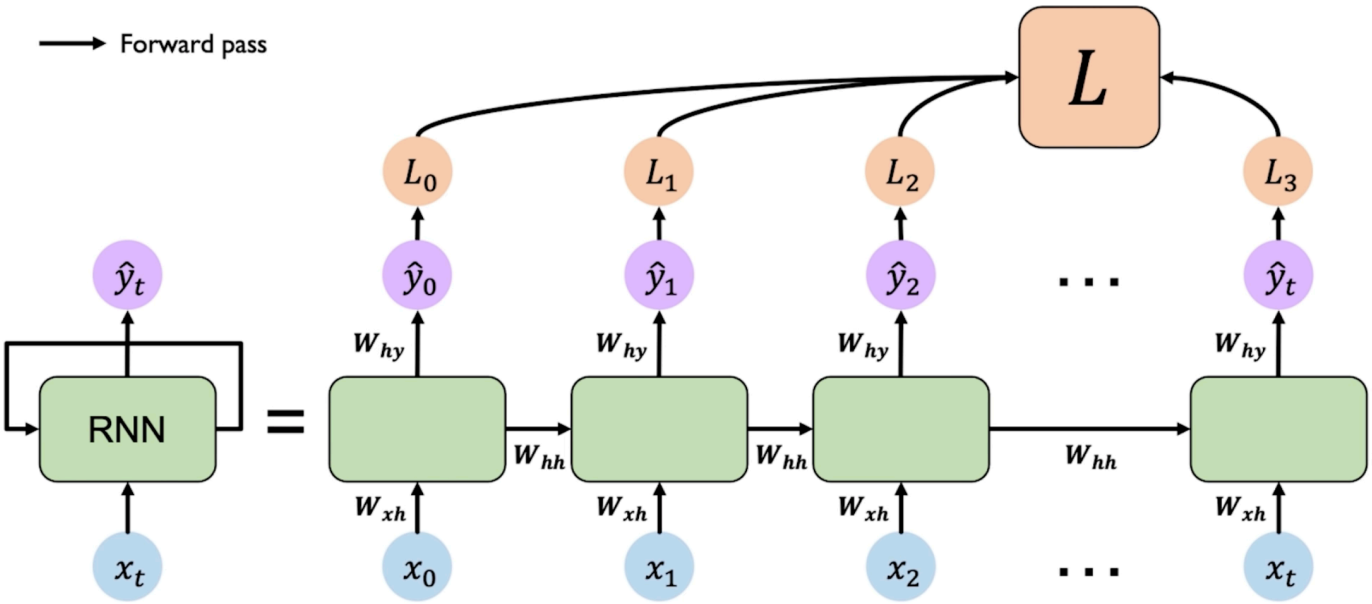


Backpropagation algorithm:

1. Take the derivative (gradient) of the loss with respect to each parameter
2. Shift parameters in order to minimize loss

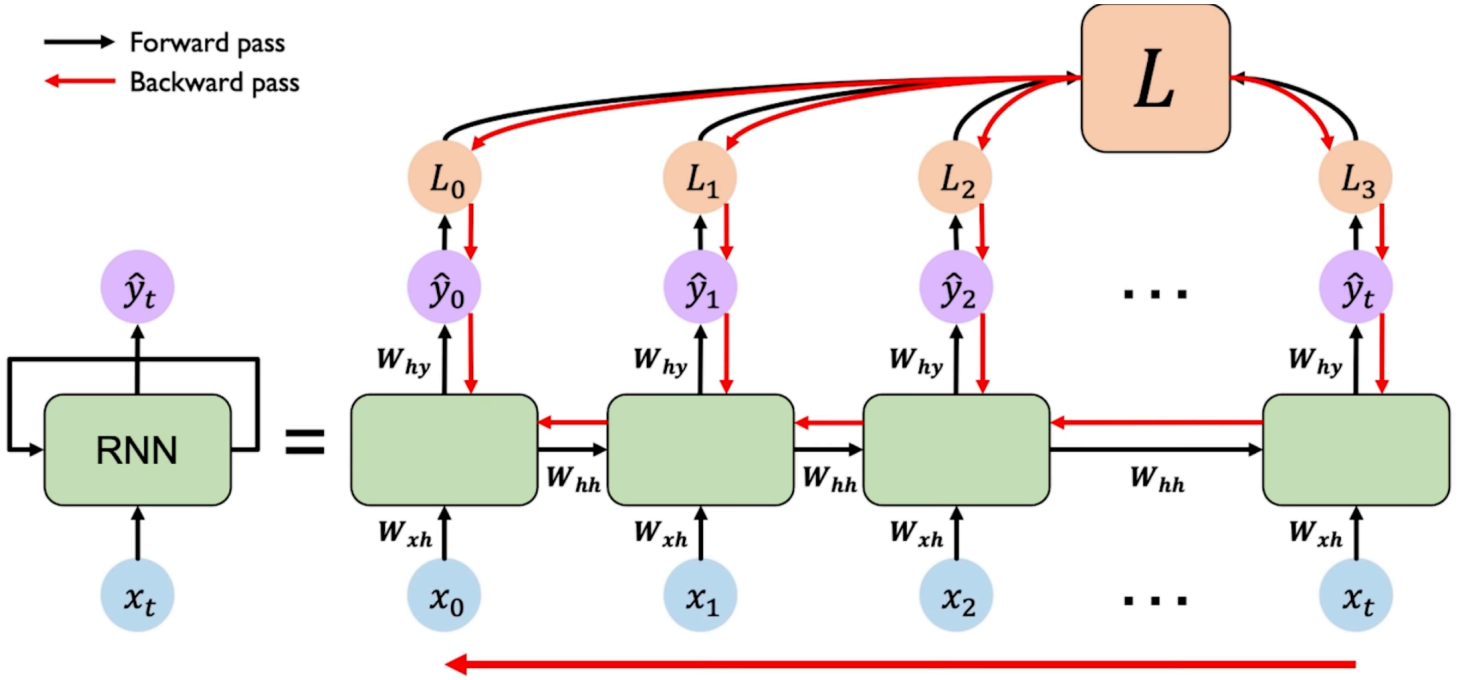
Backpropagation through time

The forward pass involves going forward across time and updating of the cell state based on the input as well as the previous cell state generating an output and fundamentally computing the loss values at the individual time steps in our sequence, and finally summing those to get the total loss.



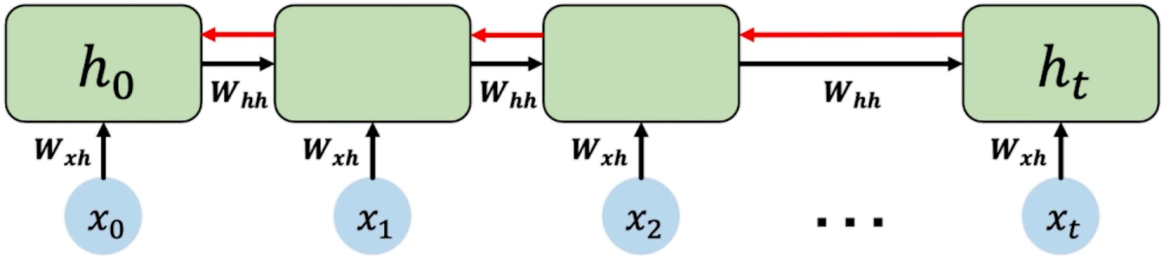
Backpropagation through time

Instead of back propagating errors through a single feedforward network at a single time step, those errors are going to be back propagated from the overall loss to each individual time-step and then across the time steps all the way from where we are currently in the sequence back to the beginning. This is the reason why it is call back propagation through time.



Bacjkpropagation through time

If we take a closer look at how the gradients flow across this chain of repeating RNN modules, we can see that between the time steps, we have to perform this matrix multiplication that involves the weight matrix W_{hh}.



Computing the gradient with respect to initial cell state h_0 is going to involve many factors of W_{hh} and also repeated computation of the gradient. This can be problematic for a couple of reasons.

When we have many matrix multiplications with values >1 we can run into an exploding gradient problem, when the gradients are going to be extremely large and we cannot optimize. And the solution here is to do a so called gradient clipping, that effectively scales back the values of particularly large gradients.

Alternatively, we can run into a vanishing gradient problem with matrix element values <1.

Vanishing gradient problem

There are three ways to tackle the vanishing gradient problem:

1. Activation function
2. Weight initialization
3. Network architecture

Vanishing gradient problem

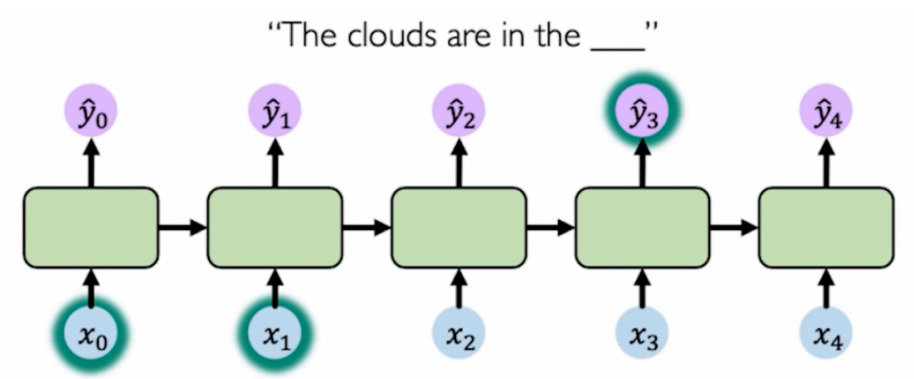
Why are vanishing gradients a problem?

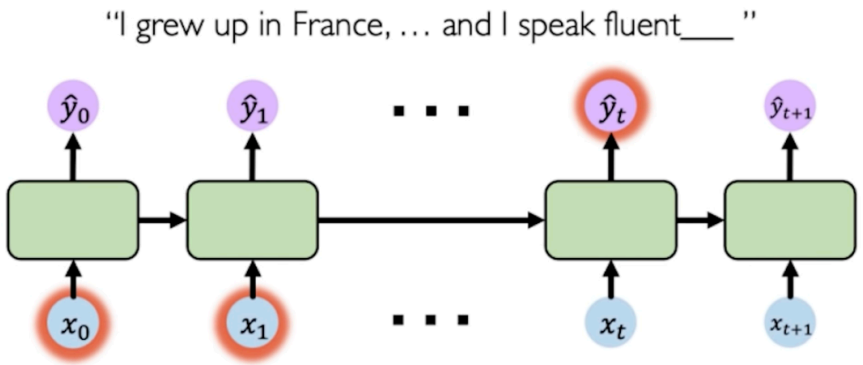
Multiply many small numbers together

Errors due to further back time steps

Have smaller and smaller gradients

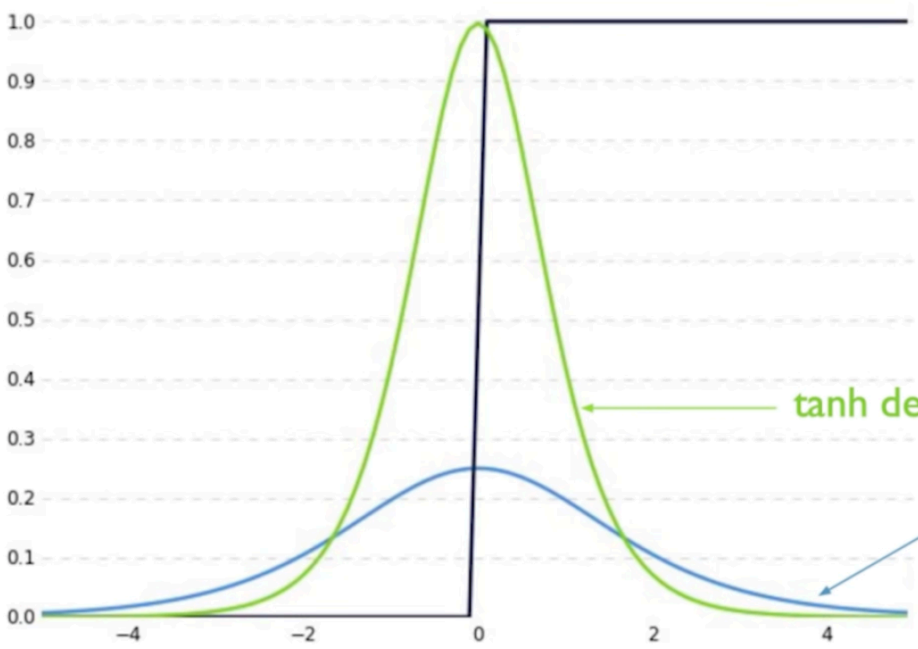
Bias parameters to capture short-term dependencies





Vanishing gradient problem

Trick #1: Activation functions



Using ReLU prevents f' from shrinking the gradients when $x > 0$

Vanishing gradient problem

Trick #2: Parameter initialization

Initialize weights to identity matrix

Initialize biases to zero.

This helps prevent weights from shrinking to zero.

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Vanishing gradient problem

The most robust solution is to introduce and to use a more complex recurrent unit that can more effectively track long-term dependencies. Intuitively, you can think of it as controlling what information is passed through and what information is used to updated the actual cell state. Specifically, we are going to use what is called a *gated cell*.

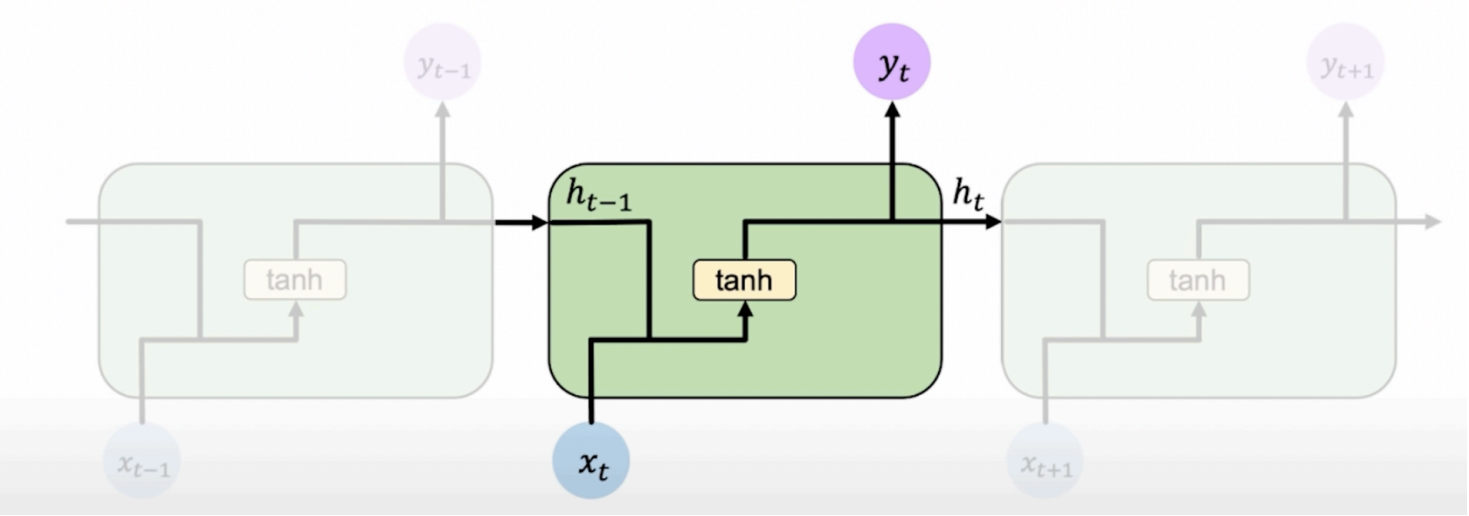
We will focus on one particular type of the gated cell that is commonly used in RNNs. It called a *Long Short Term Memory* Unit.

What is remarkable about LSTMs is that networks that are built with these units better maintain long-term dependencies in the data and retain information across multiple time steps to try to overcome the vanishing gradient problem. More, importantly, LSTMs more effectively model sequential data.

LSTMs are the workhorse of the deep learning community for most sequential modelling tasks.

Intuitive understanding of LSTMs

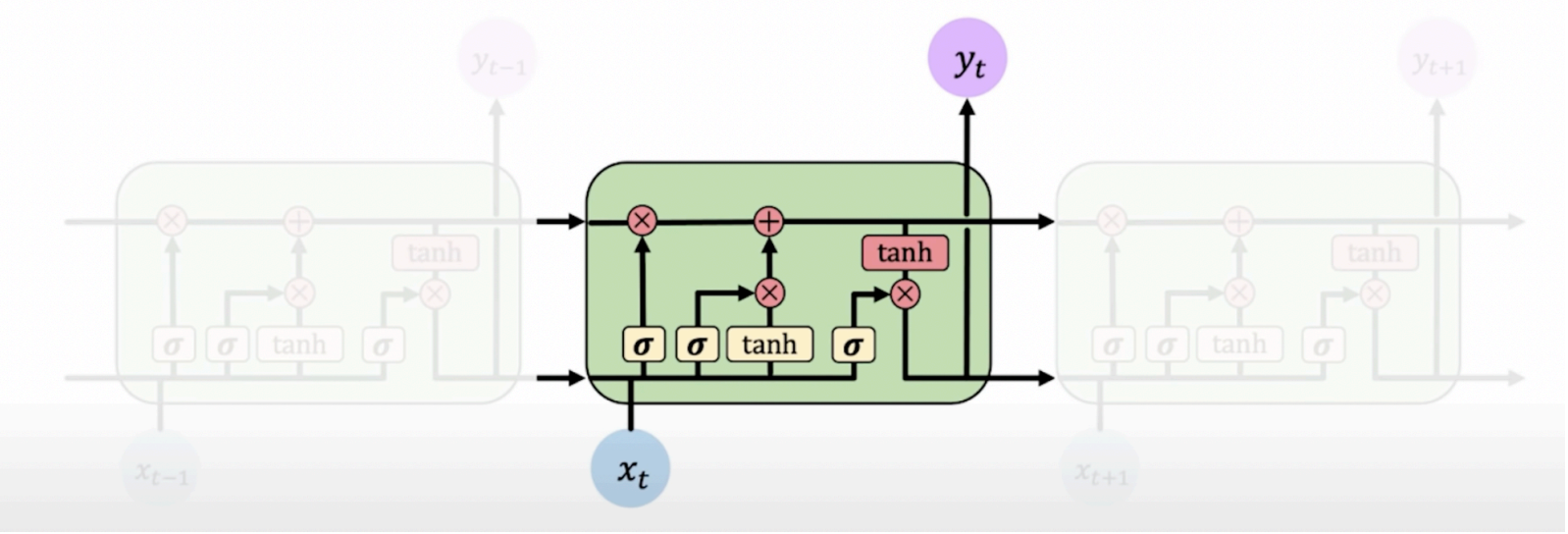
Lets revisit how standard RNNs work. We build RNN by a repeating module linked across time. Here is a representation of RNN that illustrates operations that define the state cell and output update functions. Black lines capture weight matrix multiplications and yellow rectangles show nonlinear applications of activation functions. A repeated module of RNN contains a simple computation node consisting of a tank activation function layer. Here, we are performing an update of the internal state h_t which depends on the previous cell state h_{t-1} and the current input x_t.



RNN cell structure

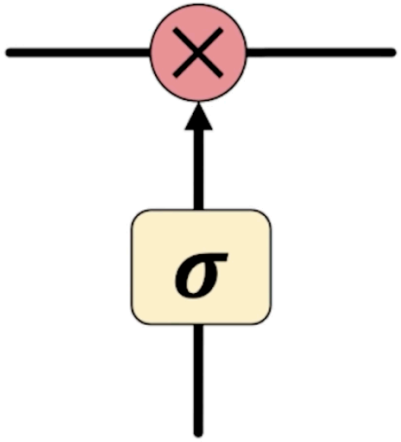
Intuitive understanding of LSTMs

LSTMs also have a chain-like structure, but an internal repeating module, that recurrent unit, is slightly more complex. In LSTMs the repeating recurrent unit contains different interacting layers which are defined by standard NN operations like sigmoid and tanh and weight matrix multiplications. What is cool about these interacting layers is that they can effectively control the flow of information through an LSTM cell LSTMs are able to track information through many time steps.



LSTM

Information is selectively added or removed to the internal cell state through structures called *gates*.

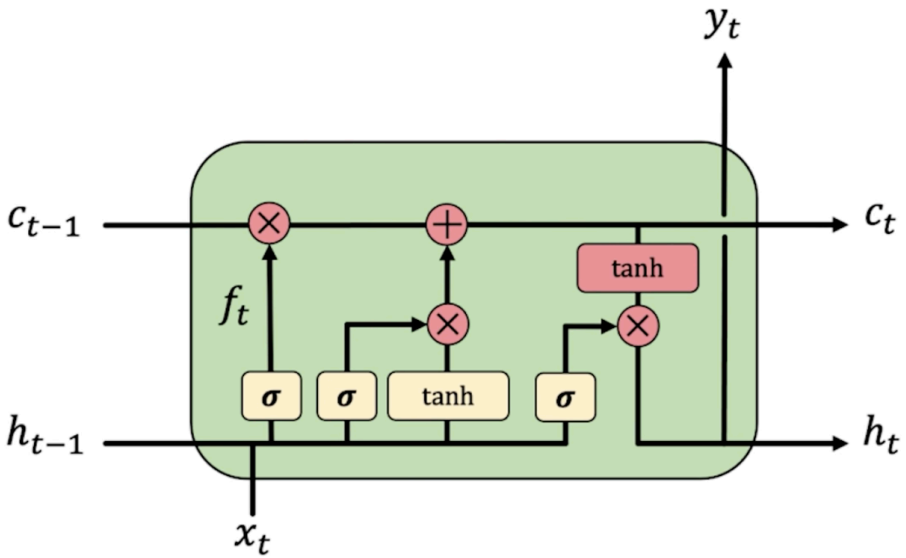


These gates consist of a standard NN layer (like a sigmoid) and element wise multiplication. Let's take a moment to think about what a gate like this can do. Because we have a sigmoid activation function, this will force everything that passes through that gate to be a real number between 0 and 1. Essentially, the gates capture how much of an input should be passed through between 0 and 1(everything) which gates the flow of information.

LSTM

Let's understand how LSTM processes information. Firstly, it forgets irrelevant history, secondly, it stores relevant new information, thirdly, it updates the internal state and finally, generates the output.

1) Forget 2) Store 3) Update 4) Output

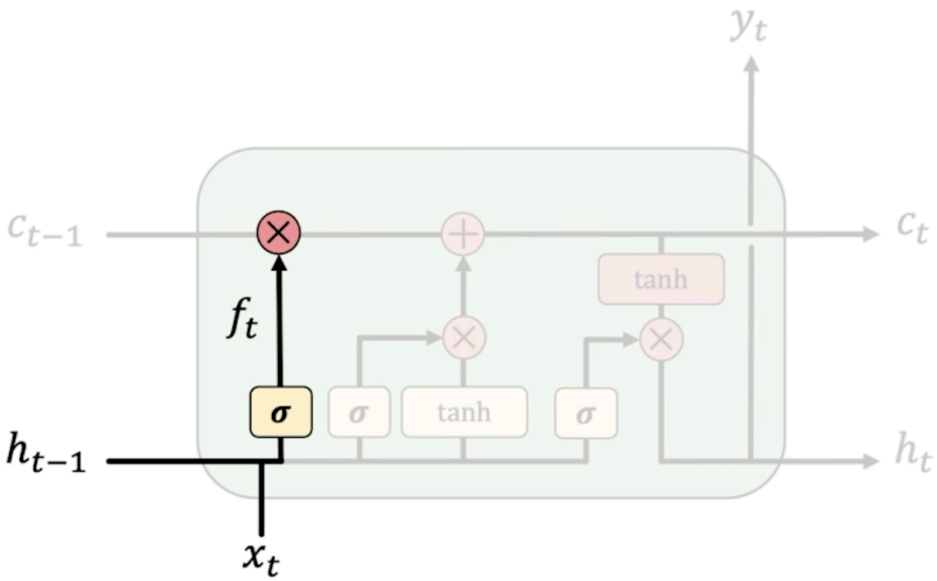


LSTM

The first step is to forget irrelevant parts of the previous state by taking the previous state and passing it through the sigmoid gates.

1) Forget 2) Store 3) Update 4) Output

LSTMs **forget irrelevant** parts of the previous state

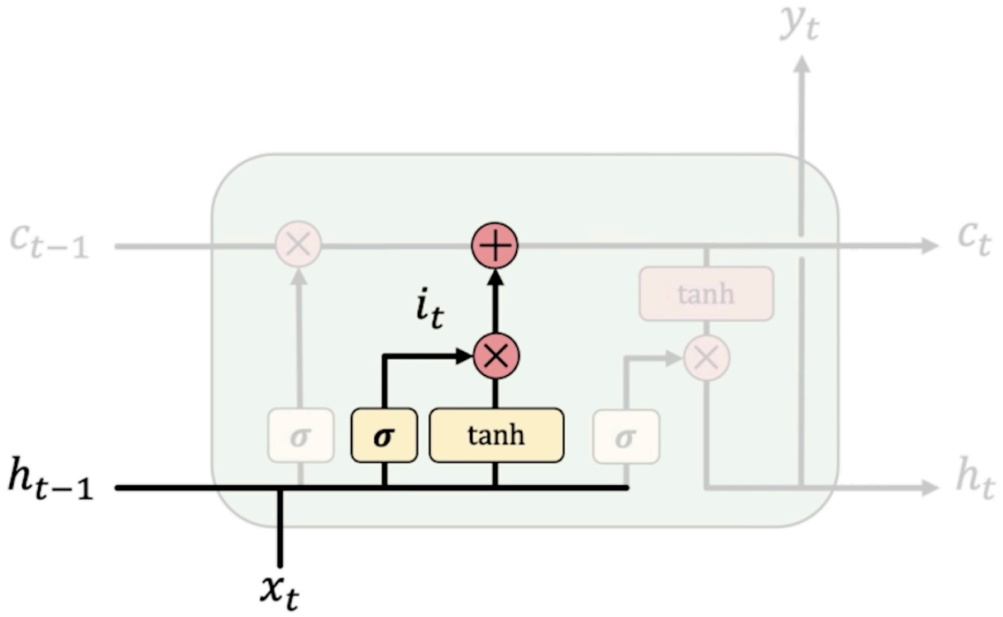


LSTM

The next step is to determine what part of a new and old information is relevant.

1) Forget **2) Store** 3) Update 4) Output

LSTMs **store relevant** new information into the cell state

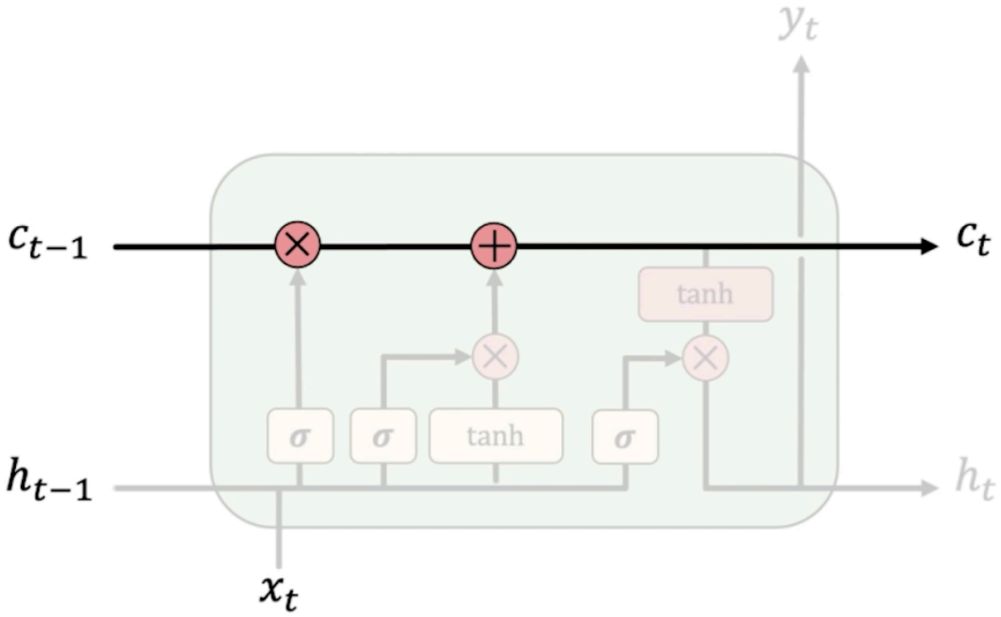


LSTM

LSTM maintains the separate value c_t of the cell state. c_t is selectively updated by gatewise operations.

1) Forget 2) Store **3) Update** 4) Output

LSTMs **selectively update** cell state values

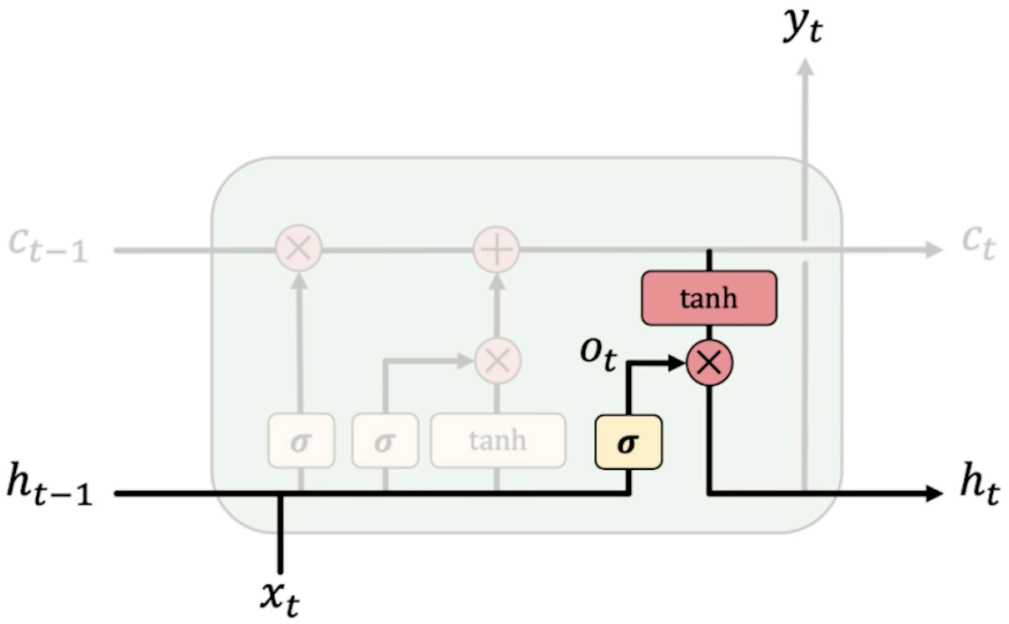


LSTM

Finally, LSTM returns the output using an interacting layer that is an output gate that controls what information encoded into a cell state can be output and sent to the network.

1) Forget 2) Store 3) Update **4) Output**

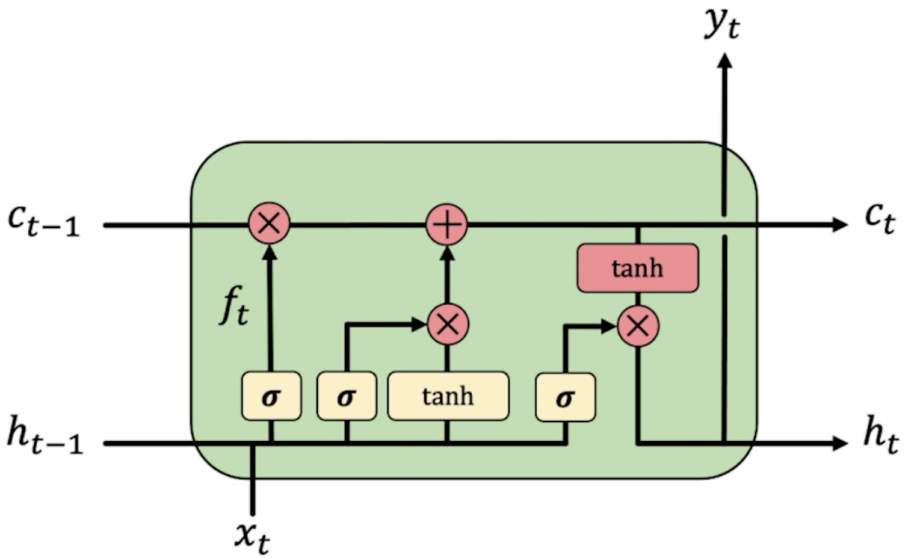
The **output gate** controls what information is sent to the next time step



LSTM

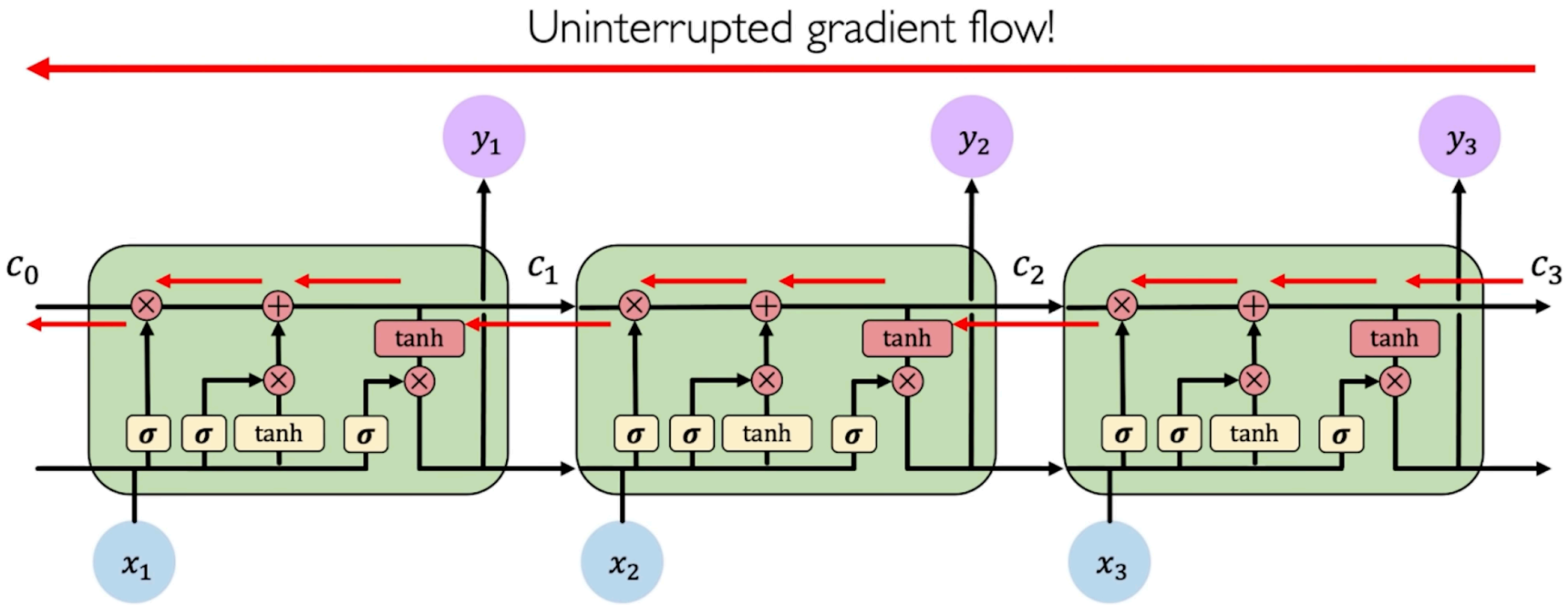
The key takeaway is that LSTM can effectively regulate information flow and storage. By doing this, they can effectively capture long-term dependencies and help us train the networks and overcome the vanishing gradient problem.

1) Forget 2) Store 3) Update 4) Output



LSTM Gradient flow

All these different gating mechanisms work to allow an uninterrupted flow of gradients. This is done by maintaining a cell separate state c_t across which actual gradient computations take place.



LSTMs: Key concepts

1. Maintain a separate cell state from what is outputted.
2. Use gate to control the flow of information
   - Forget gate forgets irrelevant information from the past
   - Stores relevant information from current input
   - Selectively update the cell state
   - Output gate returns a filtered version of the cell state
3. Backpropagation through time with uninterrupted gradient flow allowing for more effective training.

For these reasons LSTMs are very commonly used.

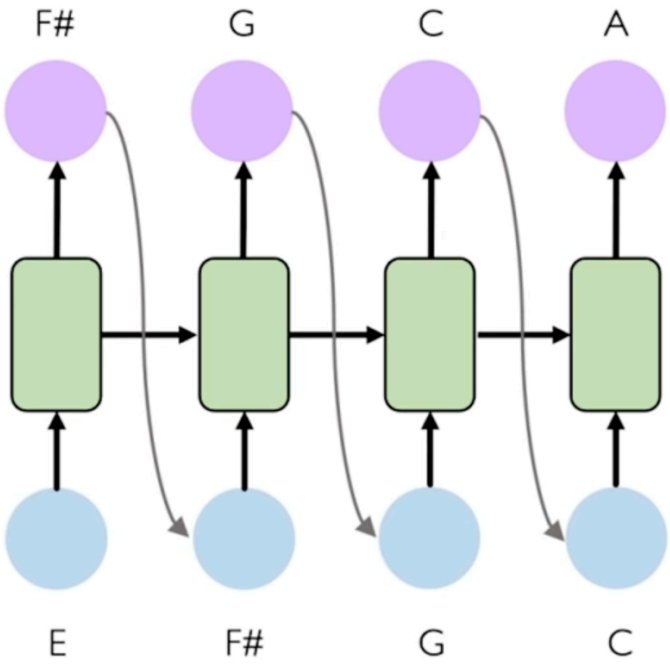
LSTM applications

How LSTMs can be deployed for sequential modelling ? Lets consider a few practical examples.

1. Music generation

Input: Sheet music

Output: Next character in sheet music



Moreover, we want to take the trained model and use it to generate music sequences that have never been heard before.

Essentially, we want to task the model to generate a new composition during testing.

What is the line between the AI and human creativity?

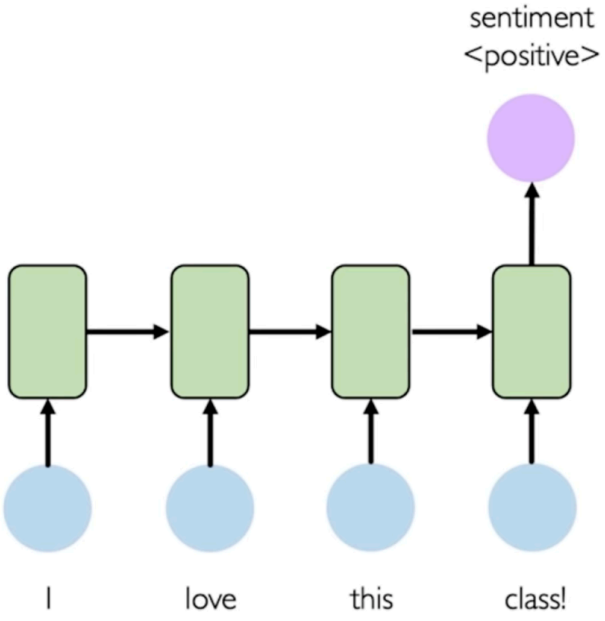
LSTM applications

Example task: Sentiment classification

Input: sequence of words

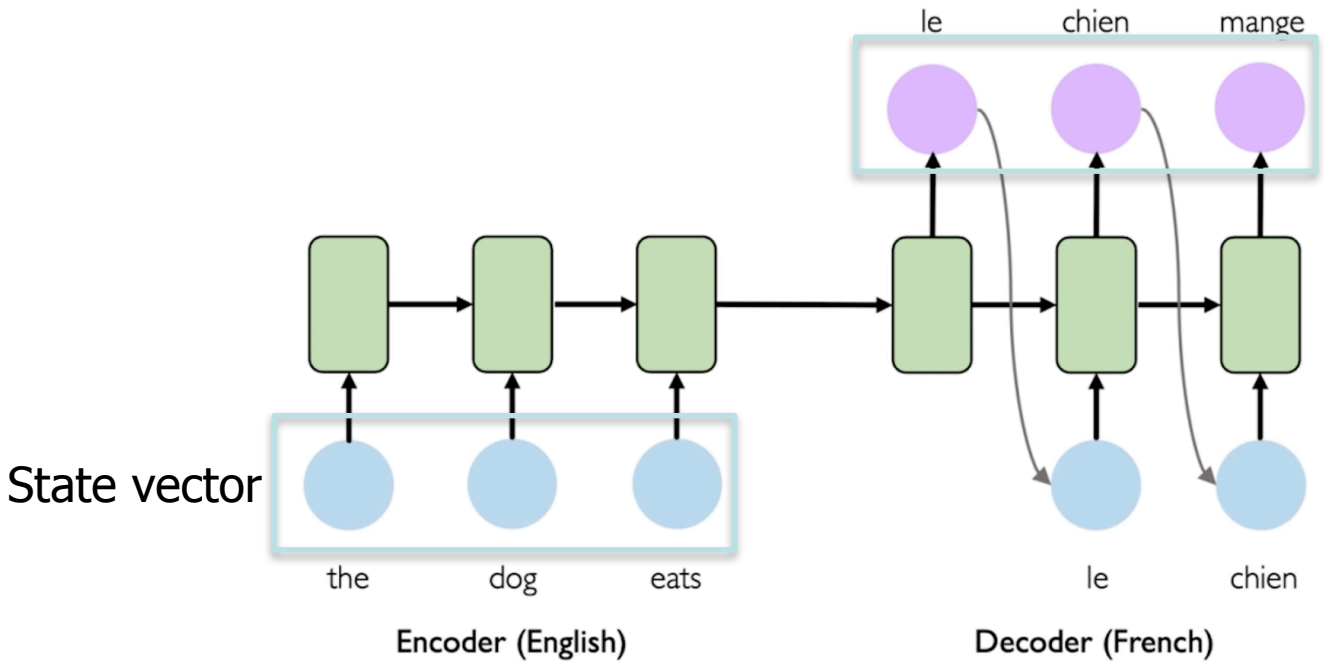
Output: probability of having positive sentiment

Train RRN to predict emotion associated with a sentence and classify tweets.



LSTM applications

Example task: Machine translation

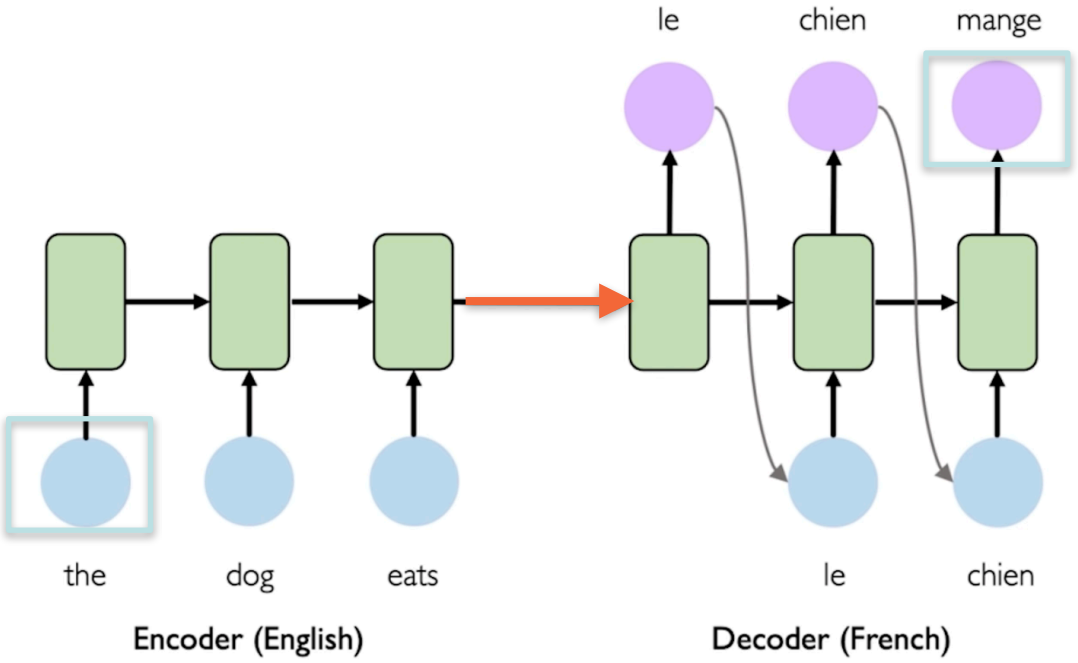


LSTM applications

Example task: Machine translation

Potential issues:

1. Encoding bottleneck
2. Slow, no parallelization, backpropagation though time is very expensive.
3. Limited memory capacity



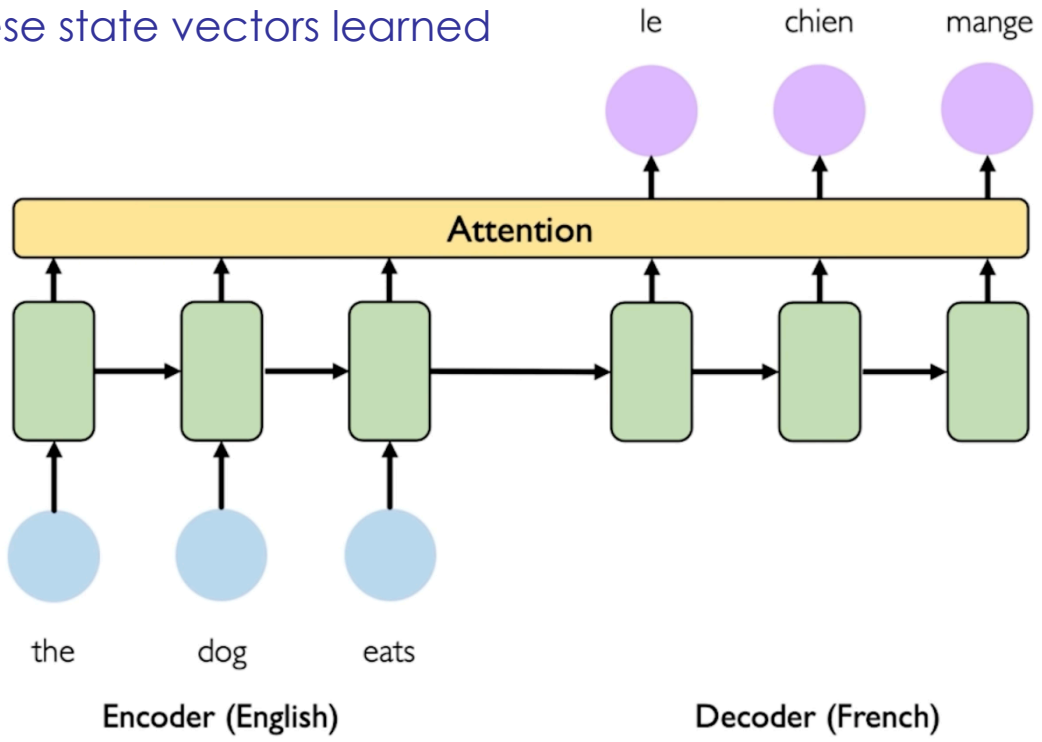
LSTM applications

Example task: Machine translation for large-scale sequences

Attention method: Instead of decoder component having access to the final state of the encoder (state vector), it has access to states after each of the time steps in the original sentence.

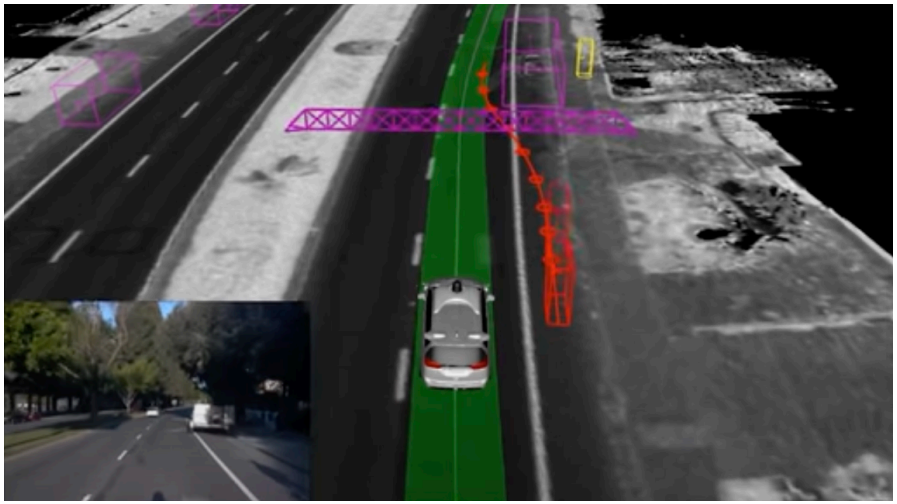
Attention is the weighting of these state vectors learned by the network over the course of training.

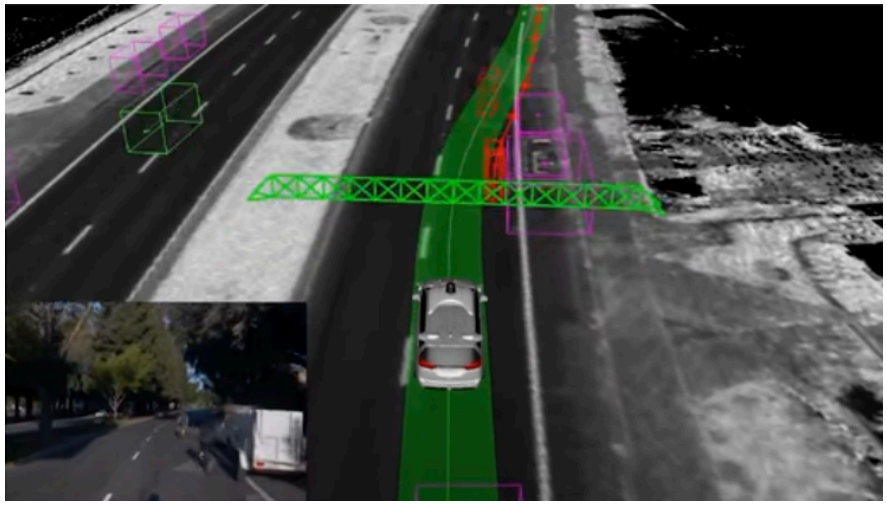
It makes it capable of capturing long-term dependencies. To train such a network only requires a single pass through attention module. Attention mechanisms provide learnable memory access.



LSTM applications

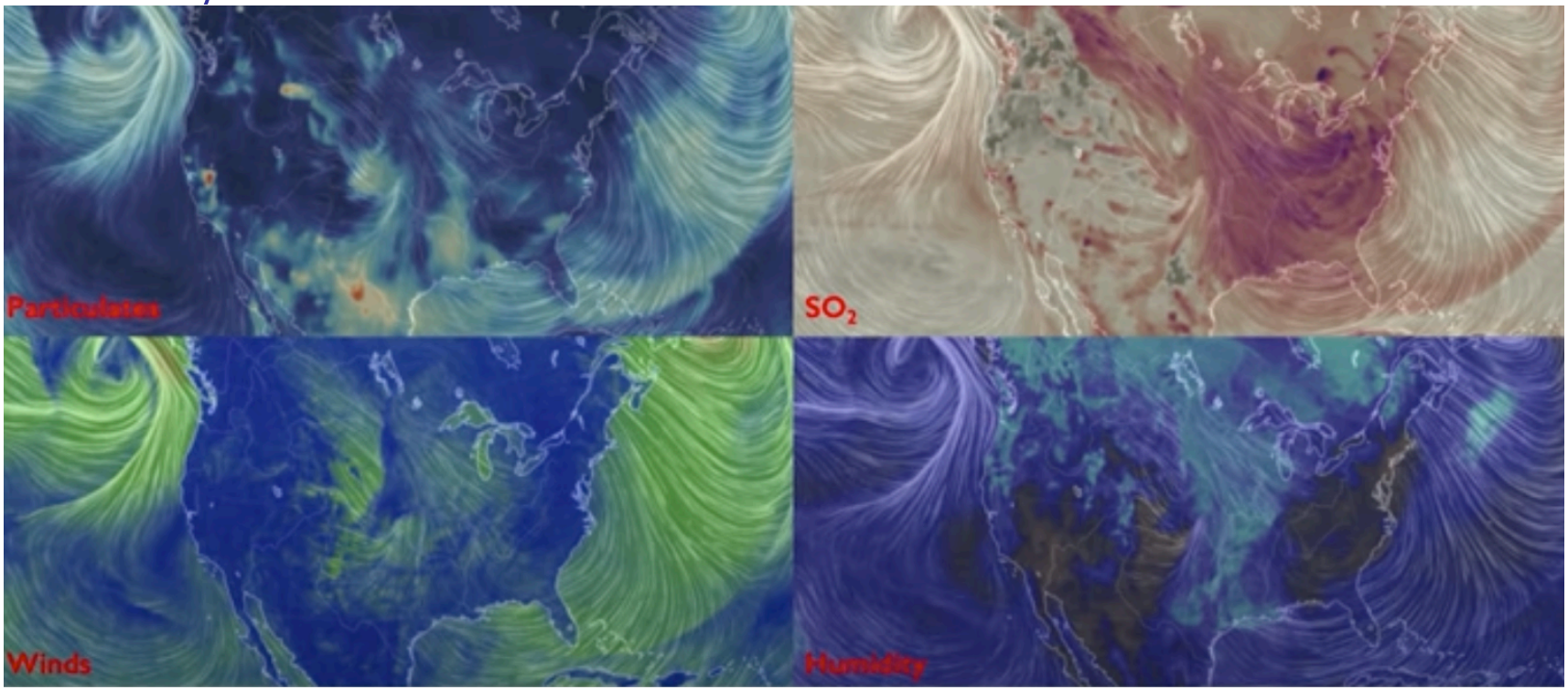
Attention consideration is very important in other applications such as trajectory prediction for self-driving cars.





LSTM applications

Example: Environmental modelling (climate pattern analysis and prediction)



Summary

1. RNNs are well-suited for sequence modelling tasks
2. Temporal dimension is incorporated via a recurrence relation
3. RNNs are trained via backpropagation through time
4. Gated cells like LSTMs model long-term dependencies
5. RNNs have a wide range of applications (music generation, machine translation and more)

Let's open iPython notebook and see theory in practice by building a simple RNN model for a sine-wave sequence from scratch.