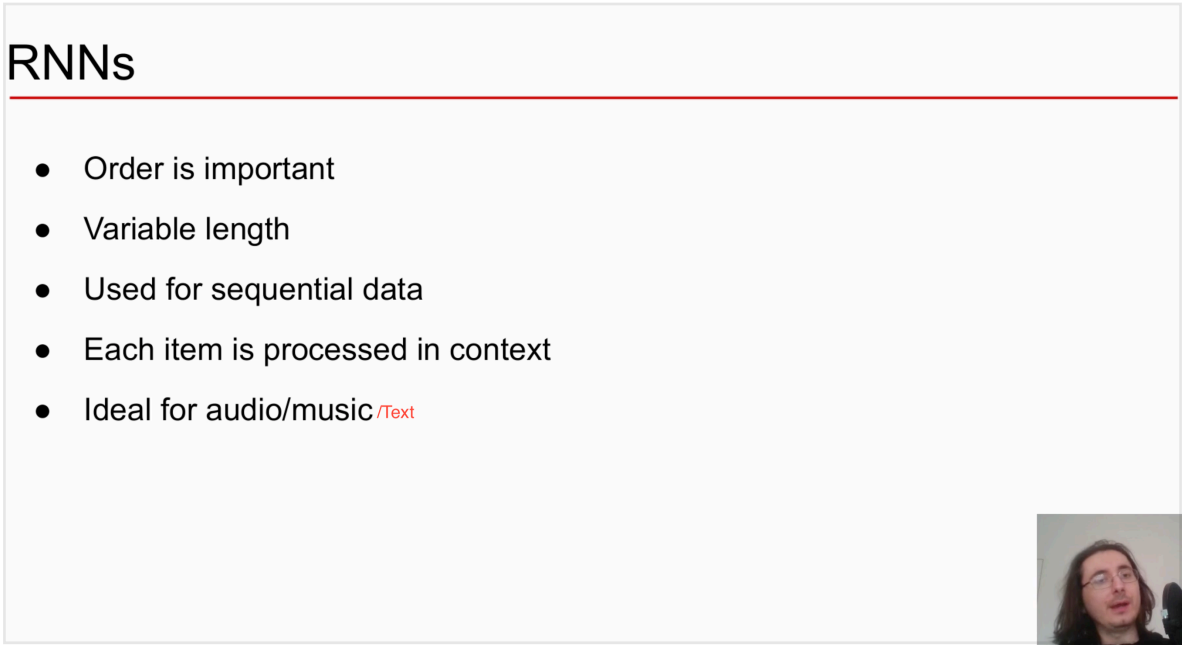
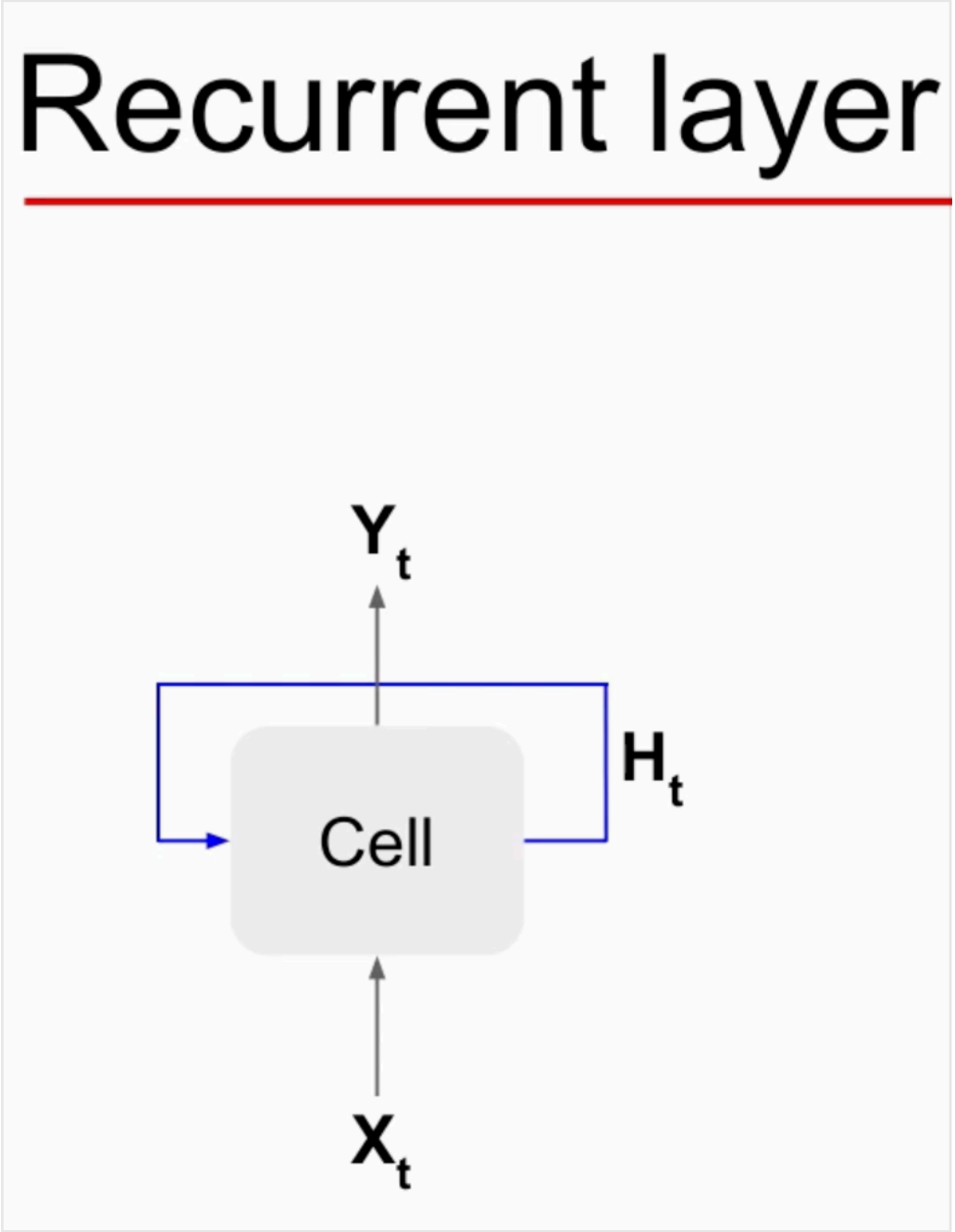
Notes 1 - Recurrent Neural Networks (RNN)

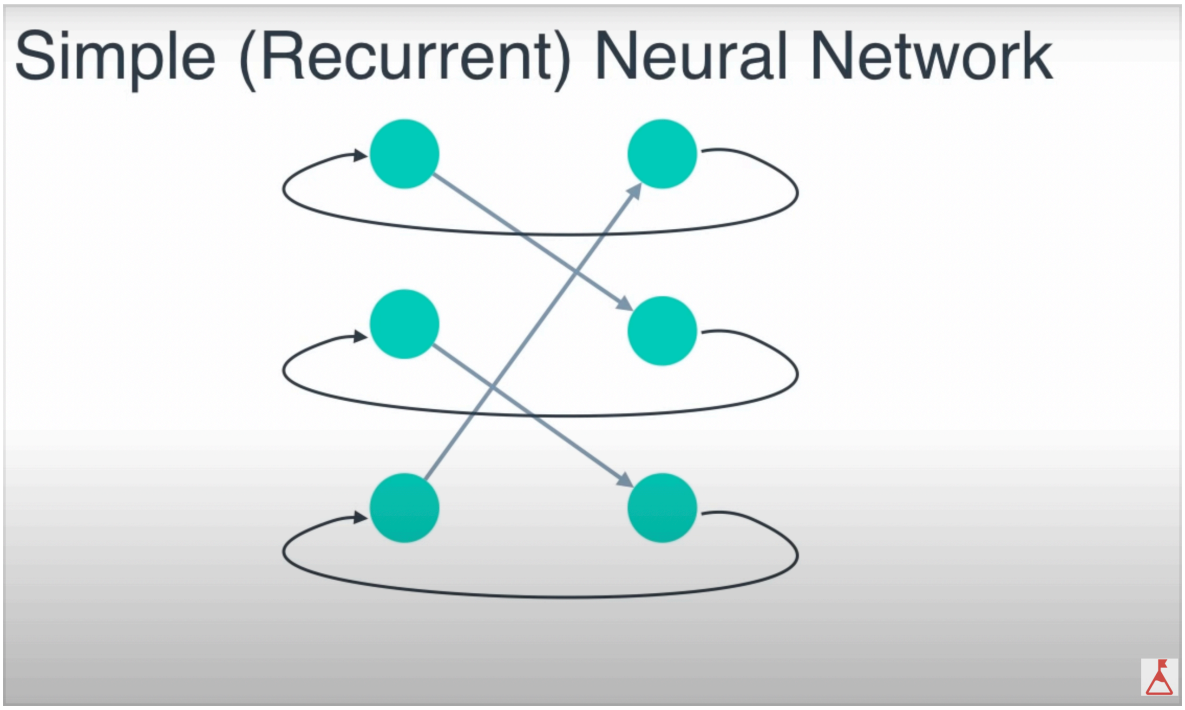
Source: Valerio Velardo's video on RNNs



- Series of points taken at different intervals of time
- Univariate time series: Taking only one value per event(per time step)
  - In context of audio, a data point with sampling rate=22050 and duration=9s, the dimension would be [22050*9, 1], 1 because we're sampling only amplitude at every instance
- Multivariate time series: Taking multiple samples per time interval(MFCCs for example)
  - In context of audio, Now the dimension would be, [sampling_rate/hop_length * duration, #MFCCs]

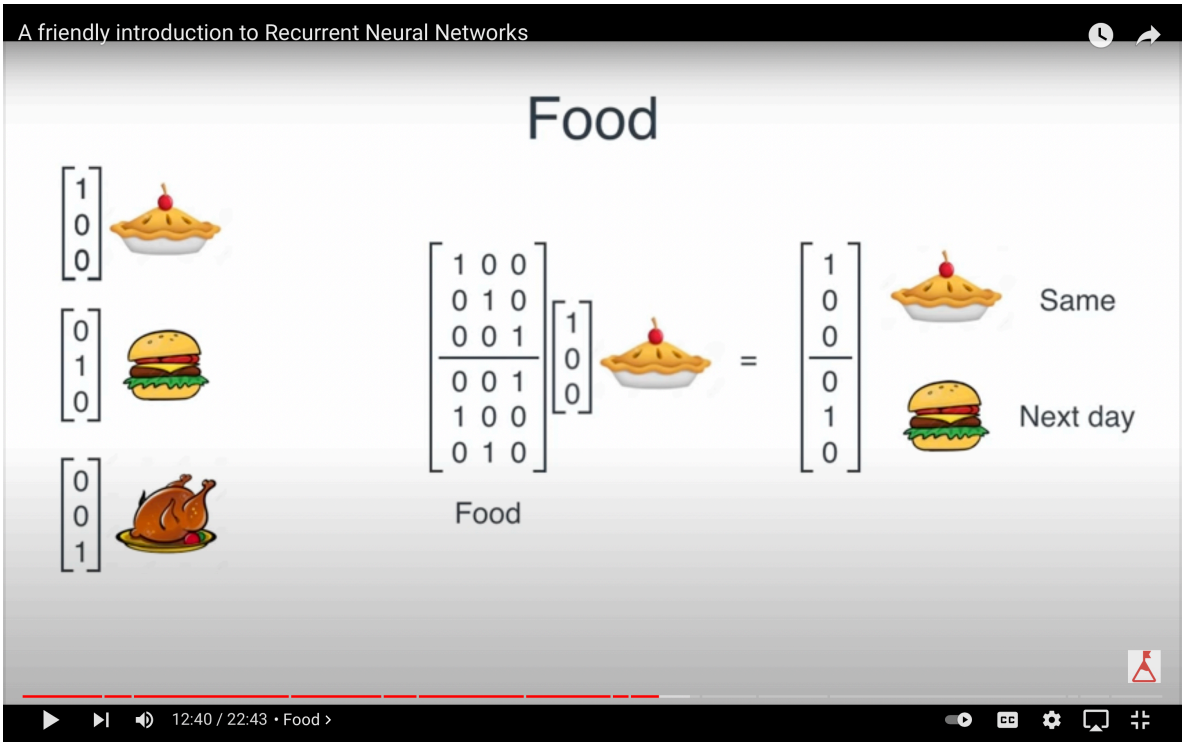


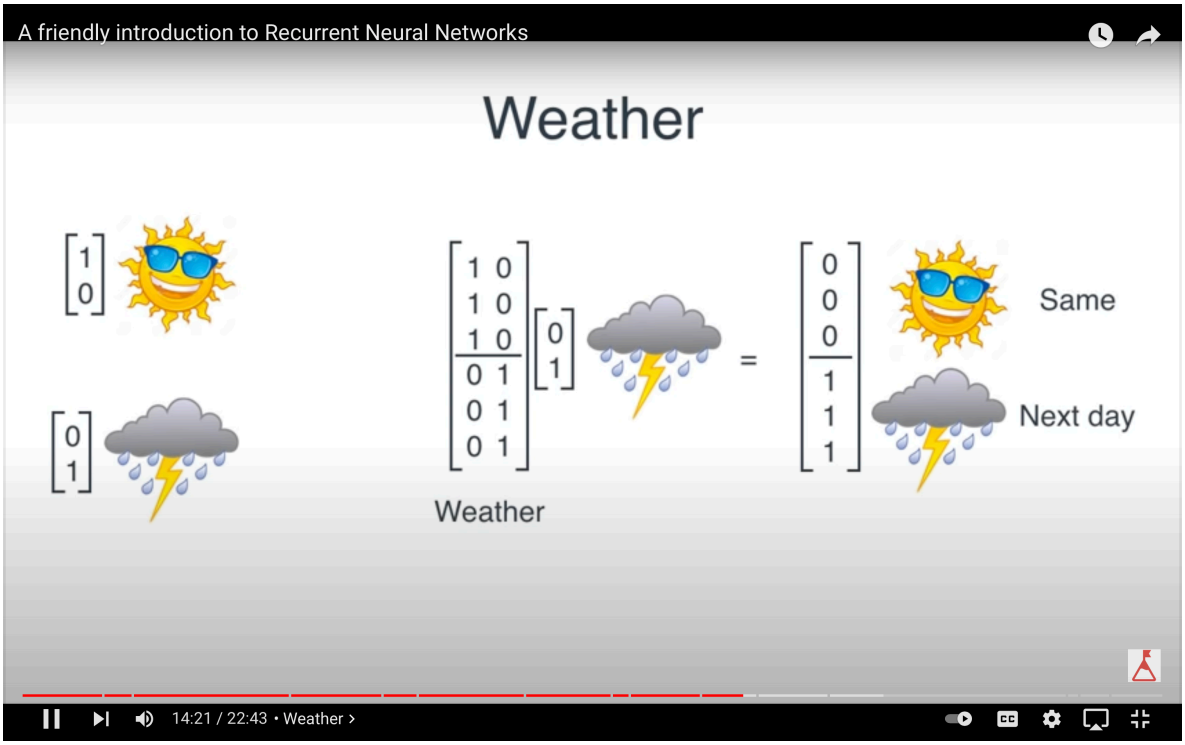
Pivoting to the next amazing video by Luis Serrano

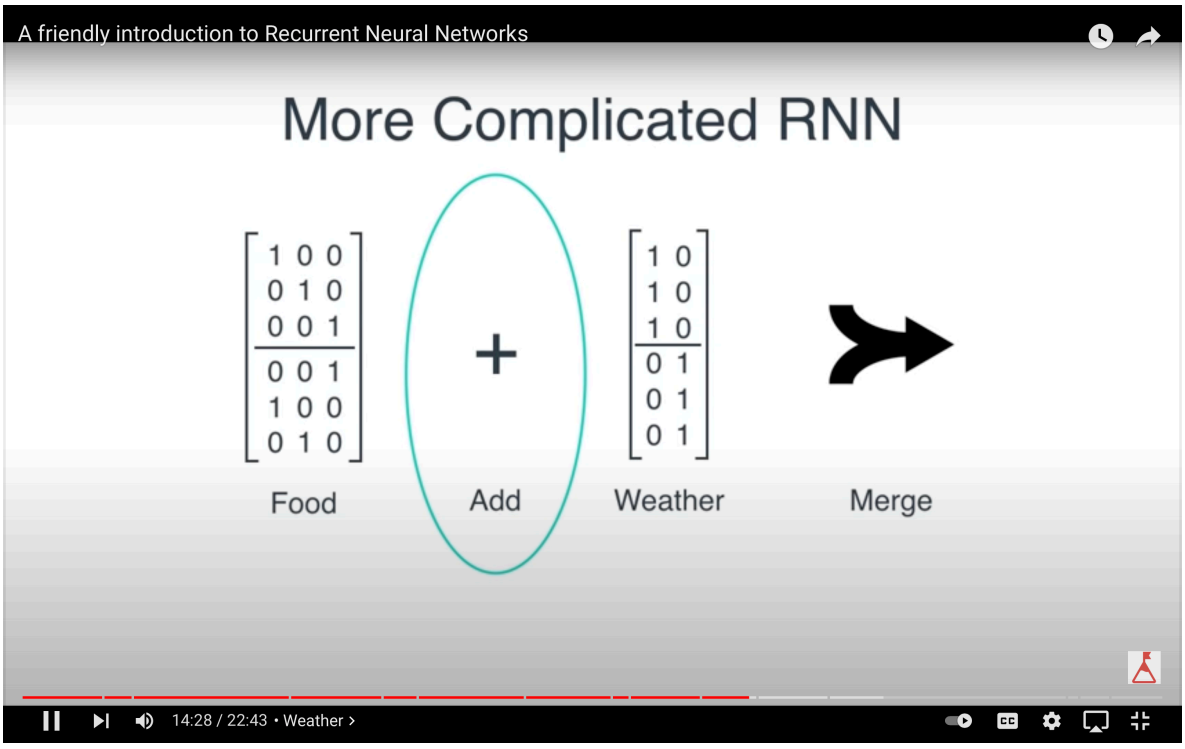


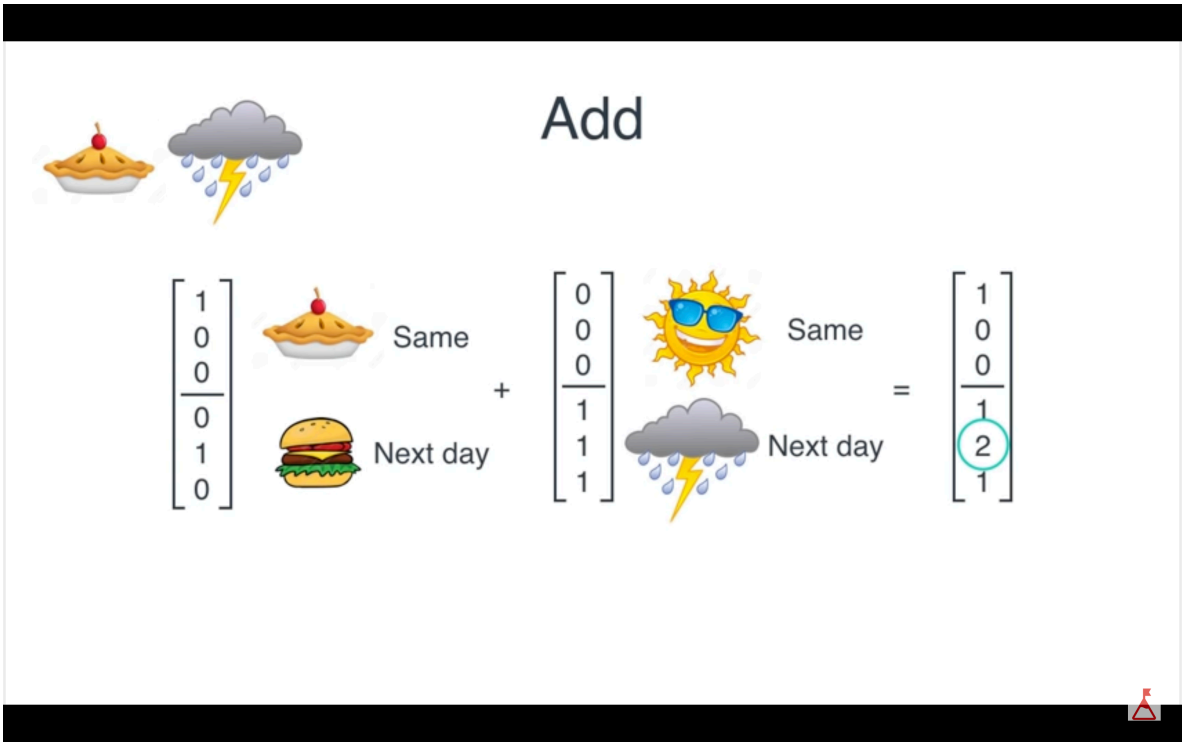
The perfect roommate example:)

When the decision for “what food to cook today” is depending on both the food for today AND the weather

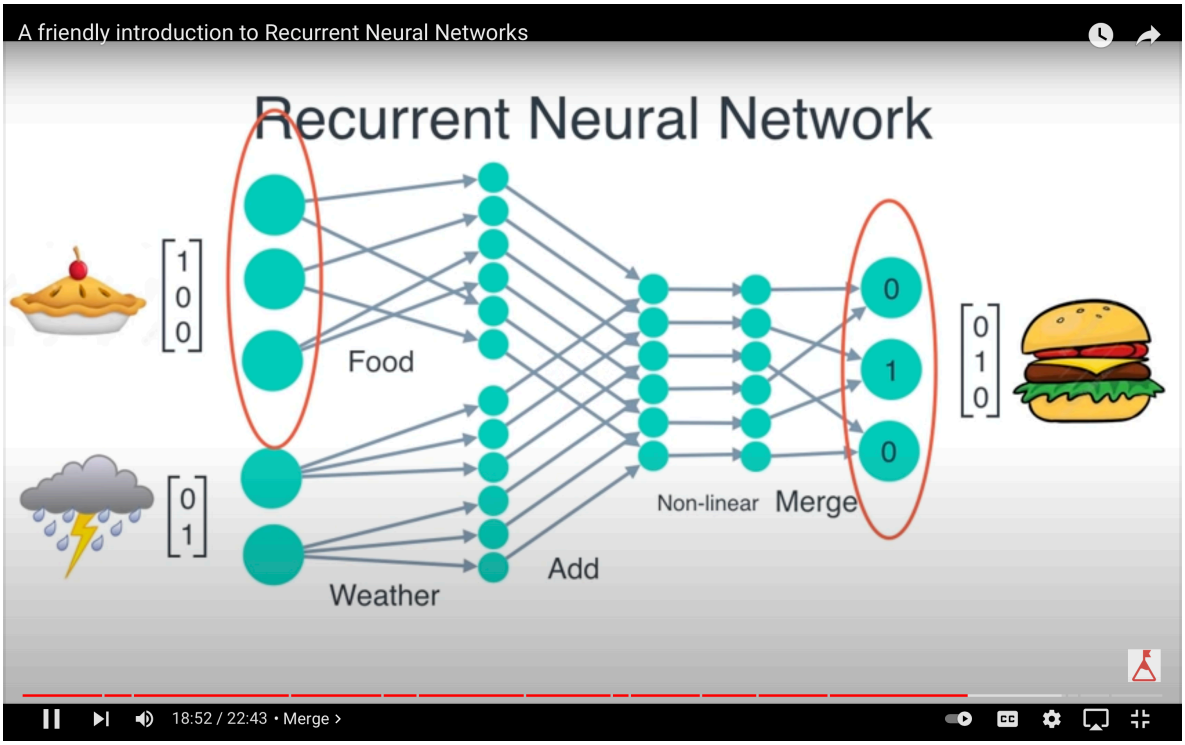


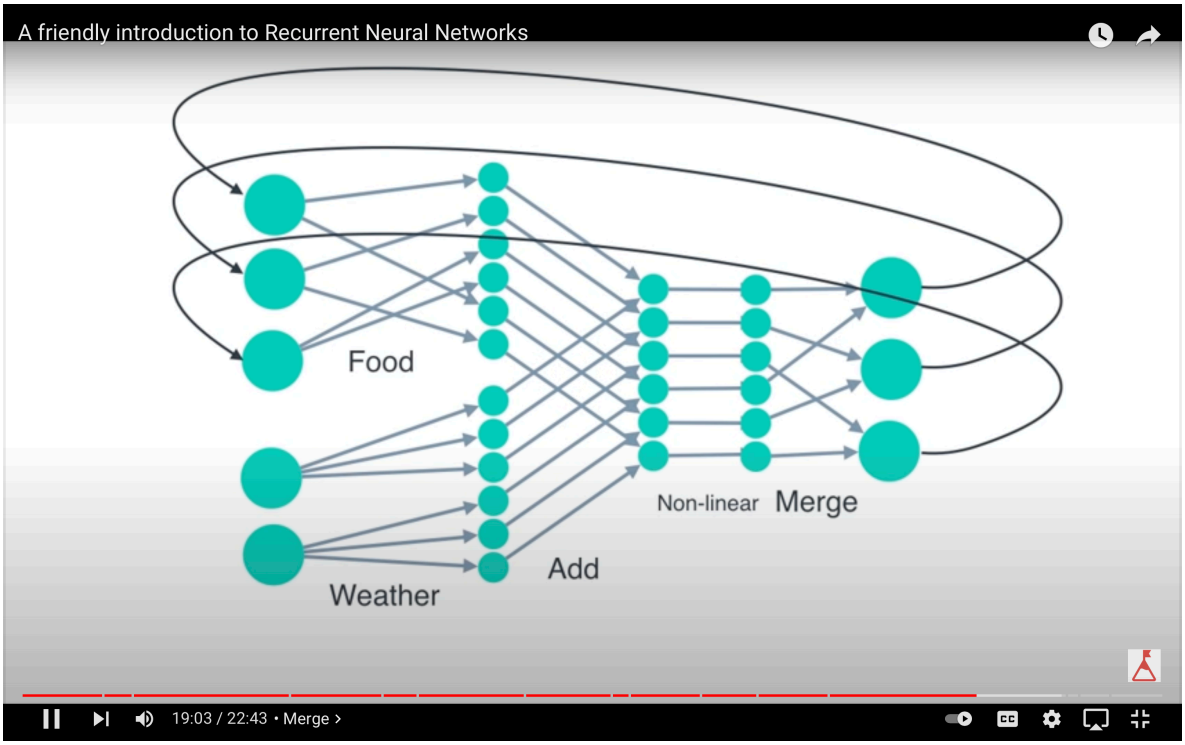

A friendly introduction to Recurrent Neural Networks
Weather
Same
Next day
Weather
14:21 / 22:43 • Weather ›


A friendly introduction to Recurrent Neural Networks
More Complicated RNN
Food
Add
Weather
Merge
14:28 / 22:43 • Weather ›



And the output of course again is the food(context for next decisions)

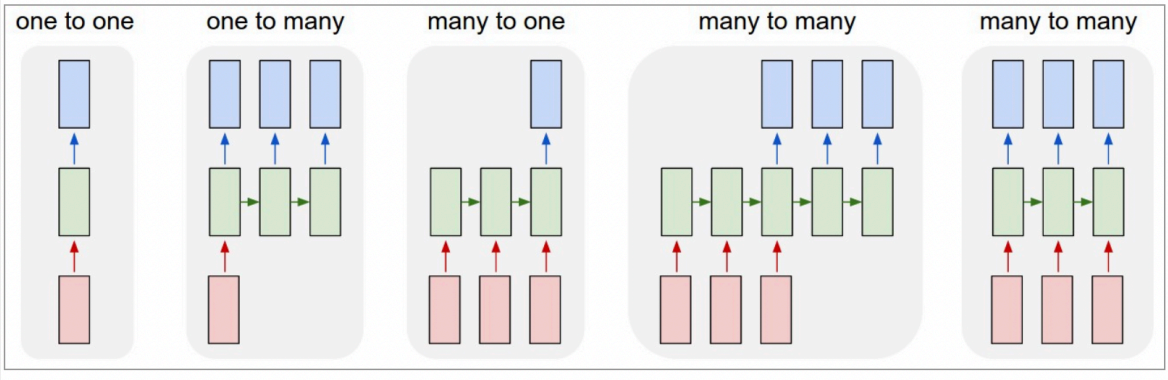




Makes sense, right!

Now getting a bit more real about it, the final source:
Karpathy's blog: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sequences. Depending on your background you might be wondering: *What makes Recurrent Networks so special*? A glaring limitation of Vanilla Neural Networks (and also Convolutional Networks) is that their API is too constrained: they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes). Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model). The core reason that recurrent nets are more exciting is that they allow us to operate over *sequences* of vectors: Sequences in the input, the output, or in the most general case both. A few examples may make this more concrete:



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

“As you might expect, the sequence regime of operation is much more powerful compared to fixed networks that are doomed from the get-go by a fixed number of computational steps, and hence also much more appealing for those of us who aspire to build more intelligent systems. Moreover, as we’ll see in a bit, RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector. This can in programming terms be interpreted as running a fixed program with certain inputs and some internal variables. Viewed this way, RNNs essentially describe programs. In fact, it is known that RNNs are Turing-Complete in the sense that they can to simulate arbitrary programs (with proper weights). But similar to universal approximation theorems for neural nets you shouldn’t read too much into this. In fact, forget I said anything.
If training vanilla neural nets is optimization over functions, training recurrent nets is optimization over programs.
”
RNNs for non-sequential data

“The takeaway is that even if your data is not in form of sequences, you can still formulate and train powerful models that learn to process it sequentially. You’re learning stateful programs that process your fixed-sized data.”

Pseudocode for an RNN step:

RNN computation. So how do these things work? At the core, RNNs have a deceptively simple API: They accept an input vector `x` and give you an output vector `y`. However, crucially this output vector's contents are influenced not only by the input you just fed in, but also on the entire history of inputs you've fed in in the past. Written as a class, the RNN's API consists of a single `step` function:

```
rnn = RNN()
y = rnn.step(x) # x is an input vector, y is the RNN's output vector
```

The RNN class has some internal state that it gets to update every time `step` is called. In the simplest case this state consists of a single *hidden* vector `h`. Here is an implementation of the step function in a Vanilla RNN:

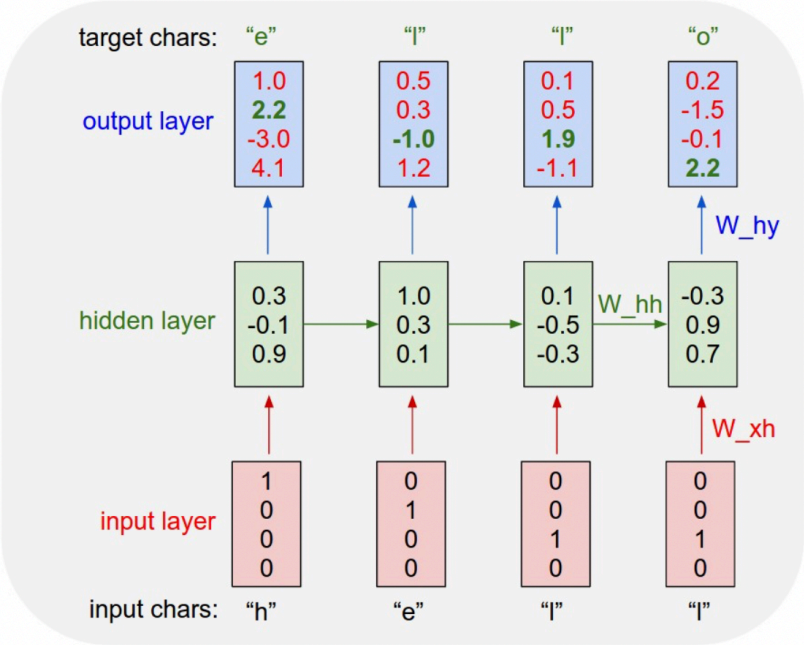
```
class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

The above specifies the forward pass of a vanilla RNN. This RNN's parameters are the three matrices `W_hh, W_xh, W_hy`. The hidden state `self.h` is initialized with the zero vector. The `np.tanh` function implements a non-linearity that squashes the activations to the range `[-1, 1]`. Notice briefly how this works: There are two terms inside of the tanh: one is based on the previous hidden state and one is based on the current input. In numpy `np.dot` is matrix multiplication. The two intermediates interact with addition, and then get squashed by the tanh into the new state vector. If you're more comfortable with math notation, we can also write the hidden state update as $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$, where tanh is applied elementwise.

We initialize the matrices of the RNN with random numbers and the bulk of work during training goes into finding the matrices that give rise to desirable behavior, as measured with some loss function that expresses your preference to what kinds of outputs `y` you'd like to see in response to your input sequences `x`.

And finally for some more intuition on one RNN step:

Concretely, we will encode each character into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary), and feed them into the RNN one at a time with the `step` function. We will then observe a sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence. Here's a diagram:



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.

For example, we see that in the first time step when the RNN saw the character "h" it assigned confidence of 1.0 to the next letter being "h", 2.2 to letter "e", -3.0 to "l", and 4.1 to "o". Since in our training data (the string "hello") the next correct character is "e", we would like to increase its confidence (green) and decrease the confidence of all other letters (red). Similarly, we have a desired target character at every one of the 4 time steps that we'd like the network to assign a greater confidence to. Since the RNN consists entirely of