DAT300 CHAPTER 16 – Modeling Sequential Data Using Recurrent Neural Netowrks

Jorid Holmen
Applied Robotics, Norwegian University of Life Sciences

This chapter is about **recurrent neural networks (RNN)**. Some applications of RNNs could be:

- **Document classification** and **timeseries classification**, such as identifying the topic of an article or the author of a book
- **Timeseries comparisons**, such as estimating how closely related two documents or two stock prices are
- **Sequence-to-sequence** learning, such as decoding an English sentence into French
- **Sentiment analysis**, such as classifying the sentiment of tweets or movie reviews as positive or negative
- **Timeseries forecasting**, such as predicting the future weather at a certain location, given recent weather data

Contents

1 Introducing sequential data **3**
- 1.1 Modeling sequential data - order matters . . . 3
- 1.2 Representing sequences . . . 3
- 1.3 The different categories of sequence modeling . . . 4

2 RNNs for modeling sequences **4**
- 2.1 Structure and flow of RNNs . . . 5
  - 2.1.1 Single layer RNN vs. Multilayer RNN . . . 5
  - 2.1.2 Computing activations in RNNs . . . 6
- 2.2 Hidden-recurrence vs. output-recurrence . . . 7
- 2.3 Working with text data in Keras . . . 8
  - 2.3.1 Word embeddings . . . 8

3 The Vanishing Gradient Problem **10**
- 3.1 Vanishing and exploding gradient problem with RNN . . . 10
- 3.2 Long Short-Term Memory (LSTM) . . . 11
- 3.3 Gated Recurrent Units (GRU) . . . 14

4 More on RNNs **15**
- 4.1 Advanced use of RNNs . . . 15
- 4.2 Sequence processing with CNN and RNN . . . 15
- 4.3 Training RNNs on time series data . . . 17
  - 4.3.1 A note about stationarity . . . 17
  - 4.3.2 Decomposition . . . 18
  - 4.3.3 Preprocessing . . . 18

5 Transformers **19**

6 Mathematics in RNNs **19**

7 Usefull pages and videos **19**

1 Introducing sequential data

Sequential data is more commonly known as sequence data or sequences. This chapter will take a look at the unique properties of sequences that makes them different to other kinds of data. Then we will see how we can represent sequential data and explore the variuous categories of models for sequential data, which are based on the input and output of a model. This will help us explore the relationship between RNNs and sequences.

1.1 Modeling sequential data - order matters

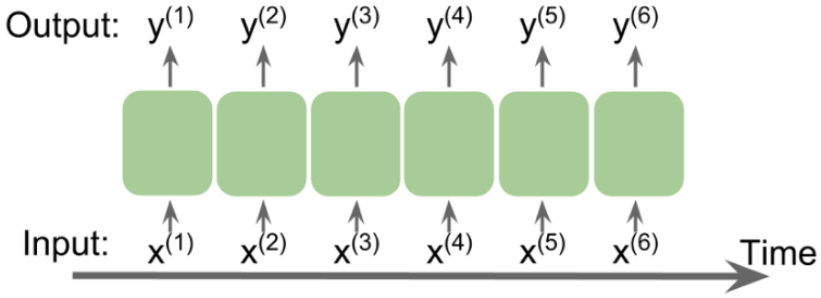
What makes sequences unique, compared to other types of data, is that elements in a sequence appear in a certain order and are not independent of each other. So far we have only worked with input data that is independent and identically distributed (IID), where the order didn't matter. This is not the case anymore. An example is when predicting the stock prices for the next three days, it would make sense to look at the most recent stock prices.

Time-series is a special type of sequential data, where each example is associated with time a dimension for time. Stock prices or speech records are examples of time-series data. Text data or DNA sequences are examples of sequences that are not time-series data.

1.2 Representing sequences

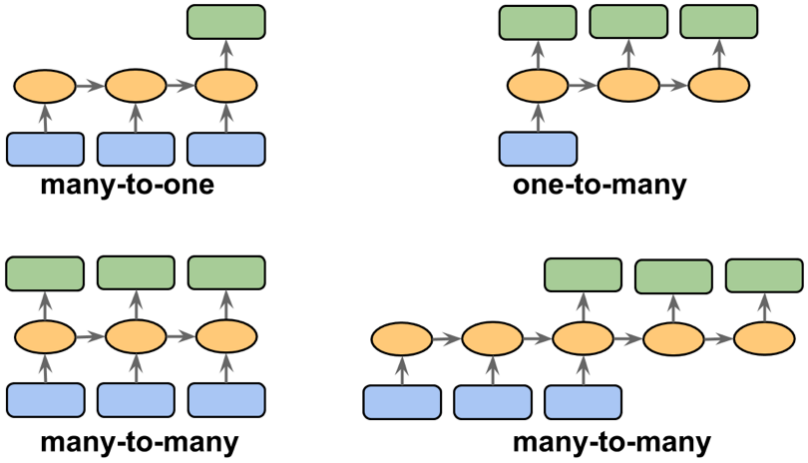
We've established that order amongst data points is important in sequential data, so next we need to find a way to leverage this ordering information in a machine learning model. Sequences will be represented by $(x^{(1)}, x^{(2)}, \ldots, x^{(T)})$. The superscript indices indicate the order of the instances, and the length of the sequences is T. For a sensible example of sequences, consider time-series data, where each example point $x^{(t)}$ belong to a particular time, t.

The following figure shows an example of time-series data where both the input features (x's) and the target labels (y's) naturally follow the order according to their time axis. Therefore, both the x's and the y's are sequences.



1.3 The different categories of sequence modeling

Different types of sequence modeling tasks require appropriate models. If either the input or output is a sequence, the data will form one of the following three different categories.



Many-to-one: The input data is a sequence, but the ouput is a fixed size vector or scalar. For example, in sentiment analysis, the input is text-based and the output is a class label.

One-to-many: The input data is a standard format and not a sequence, but the output is a sequence. An example is image captioning - the input is an image and the output is an english phrase summarizing the content of that image.

Many-to-many: Both the input and the output arrays are sequences. This category can be further divided based on whether the input and output are **synchronized**. An example of synchonized many-to-many modeling is video classification, where each frame in a video is labeled. An example of **delayed** many-to-many modeling would be translating one language into another.

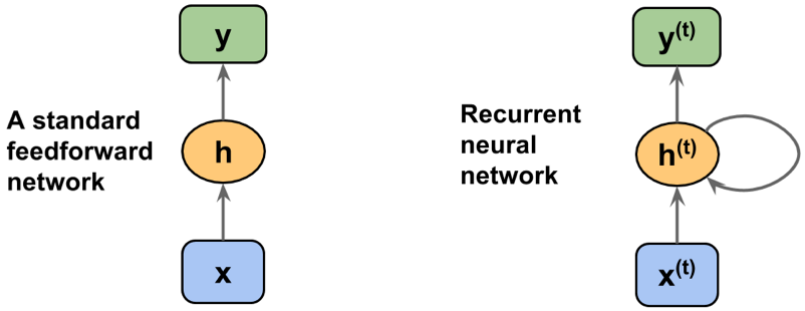
2 RNNs for modeling sequences

Standard neural network models that we have covered so far, such as MLPs (dense NN) and CNNs, are not capable of handling the order of input samples. Intuitively, one can say that such models do not have a memory of the past seen samples. With previous standard NNs, samples are passed through the feedforward and backpropagation steps, and the weights are updated independent of the order in which the sample is processed.

RNNs, by contrast, are designed for modeling sequences. RNNs are capable of remembering past information and processing new events accordingly.

2.1 Structure and flow of RNNs

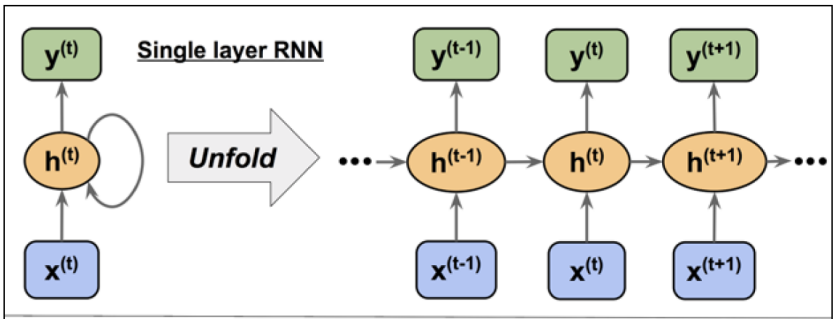
The following figure shows a standard feedforward NN and an RNN side by side for comparison:

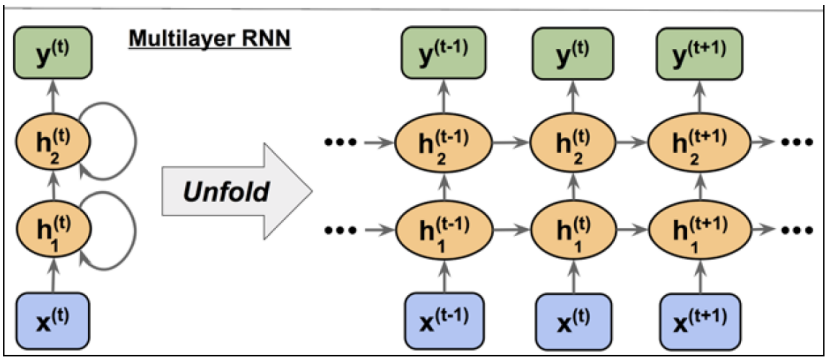


Both of the networks have only one hidden layer. In this representation, the units are not displayed, but we assume that the input layer x, the hidden layer h and the output layer o are vectors that contain many units.

In standard feedforward networks the information flows from the input to the hidden layer, and then from the hidden layer to the output layer. In RNN, the hidden layer recieves its input from both the input layer in the current time step, and from the hidden layer in the previous time step. The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation, which is how this general architecture got its name.

2.1.1 Single layer RNN vs. Multilayer RNN

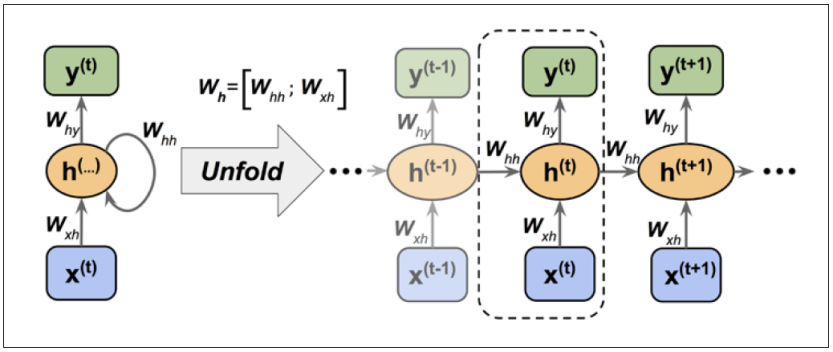




2.1.2 Computing activations in RNNs

Each directed edge (the connections between boxes) in an RNN in associated with a weight matrix. Those weights do not depend on time, t, therefore they are shared across the time axis. The different weight matrices in a single-layer RNN are as follows:

- W_{xh}: weight matrix between input $x^{(t)}$ and hidden layer h
- W_{hh}: weight matrix associated with recurrent edge. The connections between the units of the hidden layer across different time steps
- W_{hy}: weight matrix between hidden layer h and output layer y



Computing the activations is very similar to standard multilayer perceptrons and other types of feedforward NNs. For the hidden layer, the net input z_h (preactivation) is computed through a linear combination, that is, we compute the sum of the multiplications of the weight matrices with the corresponding vectors and add the bias unit:

$$z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h \tag{1}$$

$z_h^{(t)}$: net input
b_h: bias vector for hidden units

Then, the activations of the hidden units at the time step, t, are calculated as follows:

$$h^{(t)} = \phi_h(z_h^{(t)}) = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h) \tag{2}$$

Here, b_h is the bias vector for the hidden units and $\phi_h(\dot{})$ is the activation function of the hidden layer.

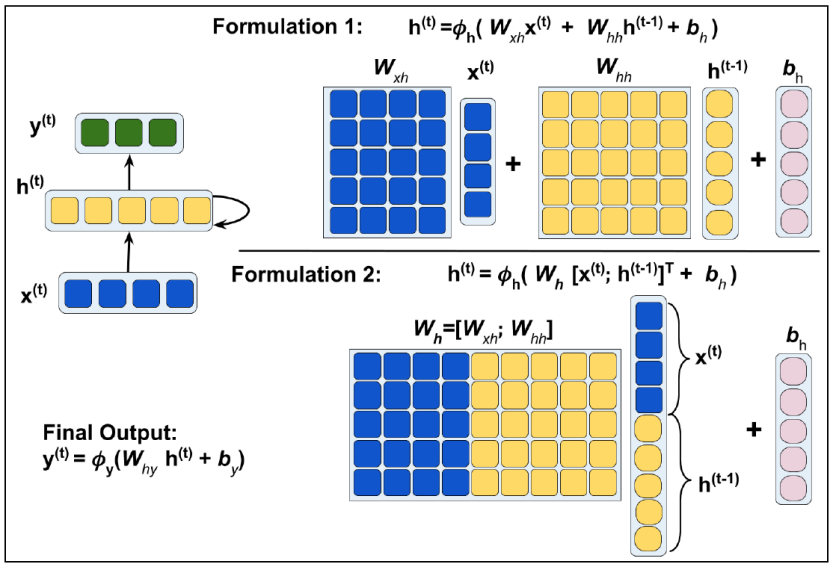
In case you want to use the concatenated weight matrix, $W_h = [W_{xh}; W_{hh}]$, the formula for computing hidden units will change as follows:

$$\boldsymbol{h}^{(t)} = \phi_h\left(\left[\boldsymbol{W}_{xh}; \boldsymbol{W}_{hh}\right]\begin{bmatrix}\boldsymbol{x}^{(t)} \\ \boldsymbol{h}^{(t-1)}\end{bmatrix} + \boldsymbol{b}_h\right)$$

Once the activation of the hidden units at the current time step are computed, then the activation of the output units will be computed, as follows:

$$o^{(t)} = \phi_o(W_{ho}h^{(t)} + b_o) \tag{3}$$

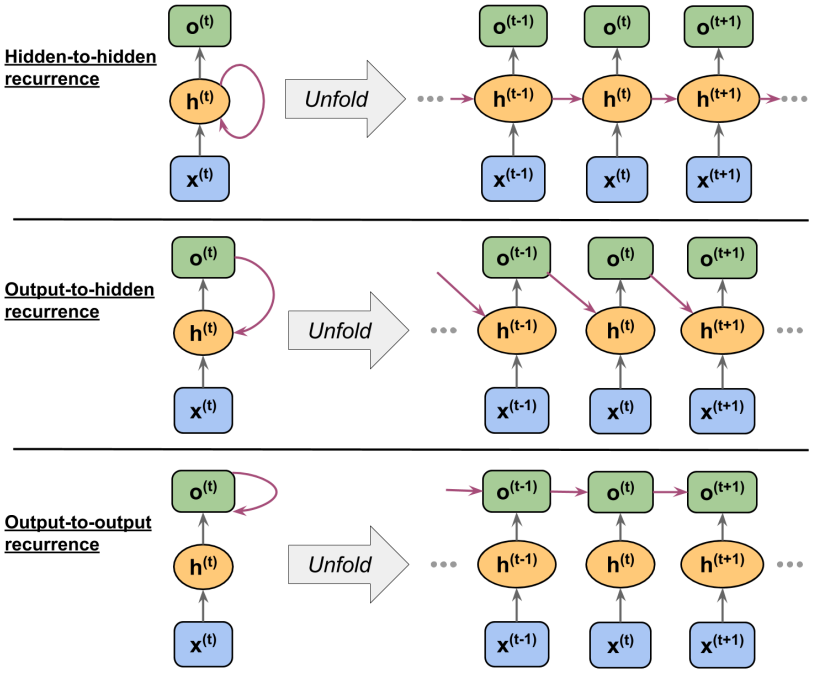
To help clarify further, the following figure shows the process of computing these activations with both formulations:



2.2 Hidden-recurrence vs. output-recurrence

So far, we have only seen recurrent networks in which the hidden layer has the recurrent property. However, note that there is an alternative model in which the recurrent connection comes from the output layer. In this case, the net activations from the output layer at the previous time step, o^{t-1}, can be added in one of two ways:

- To the hidden layer at the current time step, h^t
- To the output layer at the current time step, o^t



2.3 Working with text data in Keras

You have learned in previous lectures how to carry out sentiment analysis with scikit-learn and keras using ANN with dense layers. Now you will learn how to work with and preprocess text in keras before analysing it with RNN models built in keras. When preparing text data for RNN modeling, the Keras Tokenizer's `fit_on_text` method creates a word-to-index dictionary.

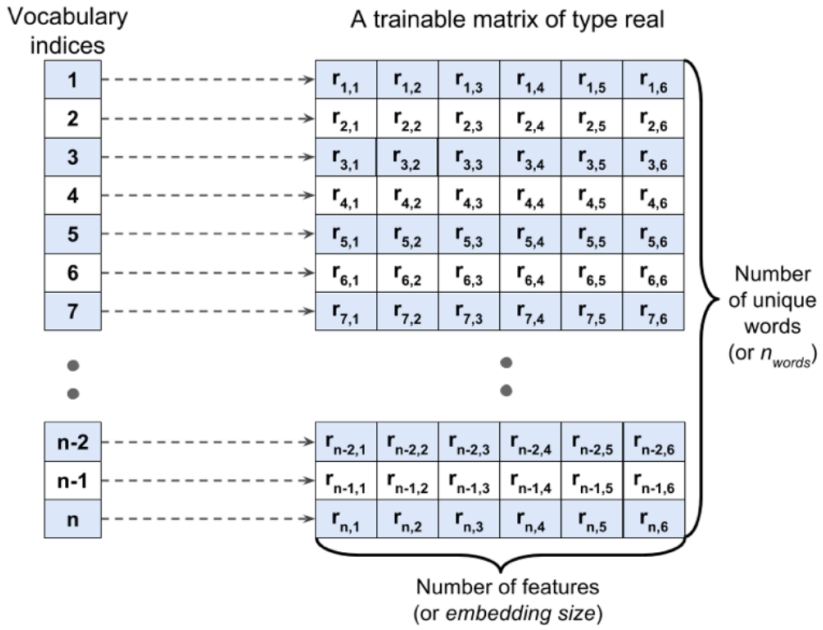
2.3.1 Word embeddings

One-hot encoding generates large sparse matrices (many zeros). A vocabulary of 20 000 words results in a sparse matrix containing 20 000 features, where each word is represented by a vector of dimension 20 000. Each vector contain 19 999 zeros except one dimension.

A more elegant way of representing words is embedding. It uses finite-sized vectors to represent words and these finite-sized vectors contain real numbers. The idea behind embedding, is that embedding is a feature learning technique. Embedding automatically learns salient features to represent a word in a dataset.

The advantaged to this approach is the reduction in dimensionality of the feature space to decrease the effect of the curse of dimensionality. In addition,

the extraction of salient features is made possible, since the embedding layer in a neural network is trainable.



There are several ways to obtain word embedding. One way is to learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of the neural network. Another way is to load into your model word embeddings that were precomputed using a different machine learning task than the one you're trying to solve. These are called pretrained word embeddings.

Word embedding with StatQuest: One super easy way to convert words into numbers is to just assign each word to a random number. Similar words that are used in similar ways could be given similar numbers, so that learning how to use one word will help learn how to use the other at the same time. And because some words can be used in different contexts, or made plural or used in some other way, it might be nice to assign each word more than one number, so that the NN can more easily adjust to different contexts. We can get a super simple neural network to do this job for us.

The first thing we do is to create inputs for each unique word. We connect each input to at least one activation function. The number of activation functions corresponds to how many numbers we want to associate with each word. The weights on these connections will, ultimately, be the numbers that we associate with each word. However, like always, these weights start out with random values that we will optimize with backpropagation. In order to do backpropagation, we have to make predictions. So, we'll use the input word to predict the next word in the phrase. If the phrase is 'Troll 2 is great!', we can

use the word 'troll' to predict the word 'is'. Put differently, the input for 'troll 2' is 1, and all the other inputs are 0, and in the output 'is' should have the largest value.

In order to make these predictions, we connect the activation functions to outputs, and we add weights to those connections with random initialization values. Then we run the outputs through the SoftMax function because we have multiple outputs for classification. That means we can use the cross entropy loss function for backpropagation.

If the weights aren't similar enough we need new weights. The new weights, after a trained neural network, on the connections from the inputs to the activation functions are the word embeddings.

Word2vec with StatQuest: Now, so far, we have shown we can train a neural network to predict the next word in each phrase, but just predicting the next word does not give us a lot of context to understand each one. Word2vec is a popular method for creating word embeddings, that includes more context. Continuous bag of words is a type of word2vec that increases the context by using the surrounding words to predict what occurs in the middle. Another method is called skip-gram and uses the word in the middle to predict the surrounding words.

The total number of weights in a neural network that we need to optimize: (amount of words/phrases) * (number of weights each word has going to the activation function) * 2 (the weights that gets us from the activation functions to the outputs) = total number of weights.

One way that word2vec speeds things up is to use negative sampling. Negative sampling works by randomly selecting a subset of words that we do not want to predict for optimization.

3 The Vanishing Gradient Problem

The use of backpropagation to update weights across many layers leads to vanishing gradient problem.

3.1 Vanishing and exploding gradient problem with RNN

The more we unroll a recurrent neural network, the harder it is to train. This problem is called the vanishing/exploding gradient problem. It is about the weight that goes from the previous hidden layer, to the next hidden layer. Just a reminder: When we optimize neural networks with backpropagation, we first find the derivatives of gradients, for each parameter. We then plug those gradients into the gradient descent algorithm to find the parameter values that minimize a loss function, like the sum of the squared residuals.

The gradient will explode when we set the weight to any value larger than 1. If the weight is higher than 1, the input value will get bigger and bigger (exploding). If we tried to train this recurrent neural network with backpropagation,

the huge number would find its way into some of the gradients, which would make it hard to make small steps to find the optimal weights and biases. When finding the parameter values that give us the lowest value for the loss function, we usually want to take relatively small steps. If the gradient contains a huge number, we end up taking relatively large steps. Instead of finding the optimal parameter, we'll just bounce around a lot.

One way to prevent the exploding gradient problem would be to limit the weight to values less than 1. However, this results in the vanishing gradient problem. The input value will get smaller and smaller, and eventually get super close to 0. Now, when optimizing a parameter, instead of taking steps that are too large, we end up taking steps that are too small. As a result, we end up hitting the maximum number of steps we are allowed to take before we find the optimal value.

Various solutions exists, such as:

- Gradient clipping: solves the exploding gradient problem. Set a max value for gradients if they grow to large.
- Truncated Backpropagation through time (TBPTT): limits the number of time steps the signal can backpropagate each forward pass.
- Long-short-term-memory (LSTM) units: Architecture that avoids vanishing/exploding gradient problems.

Using gradient clipping we specofy a cut-off thershold value for the gradients, and we assign this cut-off value to gradient values that exceed this value.

In contrast, TBPTT simply limits the number of time steps stat the signal can backpropagate after each forward pass. For example, even if the sewquence has 100 elements or steps, we may only backpropagate the most recent 20 time steps.

3.2 Long Short-Term Memory (LSTM)

LSTM is a type of recurrent neural network that is designed to avoid the exploding/vanishing gradient problem using gated cells. The main idea behind how LSTM works is that instead of using the same feedback loop connection for events that happened long ago and events that just happened yesterday to make a prediciton about tomorrow, LSTM uses two seperate paths to make predictions about tomorrow. One path is for long-term memories, and one is for short-term memories.

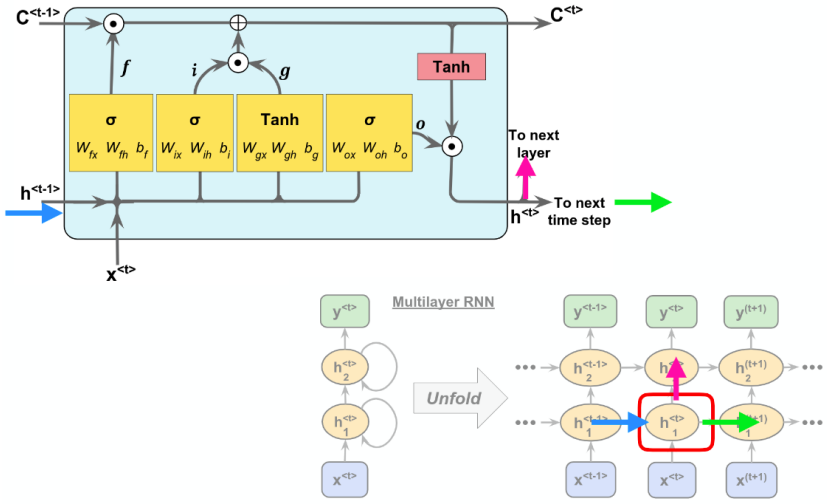
The building block of an LSTM is a memory cell, which essentially represents or replaces the hidden layer of standard RNNs. In each memory cell there is a recurrent edge that has the desirable weight, $w = 1$. The values associated with this recurrent edge are collectively called the cell state.

RNN unrolls from a relatively simple unit, but LSTM is based on a much more complicated unit. Note that LSTM uses sigmoid activation functions and

tanh activation functions. Sigmoid takes any x-axis coordinate and turns it into a y-axis coordinate between 0 and 1. In contrast, the tanh/hyperbolic tangent takes any x-axis coordinate and turns it into a y-axis coordinate between -1 and 1. There are two tanh activation functions in an LSTM cell. The first normalizes the cell, while the second one is used to generate the hidden state.

In LSTM, the gate is updated at every time step, which contributes to maintaining the long-term dependencies.

An LSTM in divided into a hidden state and a cell state. The cell state carries long-term information, while the hidden state carries short-term information. The hidden state is typically used for redictions at each time step. There are two hyperbolic tangent (tanh) activation functions in an LSTM cell, the first tanh function normalizes the cell state, while the second one is used to generate the hidden state.



Notice that the cell state from the previous time stec $C^{(t-1)}$ is modified to get the cell state at the current time step $C^{(t)}$, without being multiplied directly with any weight factor. The flow of information in this memory cell is controlled by several computation units, often called gates.

In an LSTM cell, there are three different types of gates, which are known as the forget gate, the input gate and the output gate.

- **The forget gate** (f_t) allows the memory cell to reset the cell state without growing indefinitely. In fact, the forget gate decides which information is allowed to go through and which information to suppress.
- **The input gate** (i_t) and **canditate value** $(\tilde{C}_t)$ are responsible for updating the cell state, $C^{(t)}$. It decides what new information to add to the cell state.

- **The output gate (o_t)** decides how to update the values of the hidden units, $h^{(t)}$.

LSTM with **StatQuest**:

The green line from the first image under is the Cell State and represents the long-term memory. Long-term memory can be modified through a multiplication, and later by an addition, but there are no weight or biases that can modify it directly. The lack of weights allows the long-term memories to flow through a series of unrolled units without causing the gradient to explode or vanish.

The pink line, is called the Hidden State, and represents the Short-Term memories. The short term memories are directly connected to the weights that can modify them. This state is typically used for predictions at each time step.

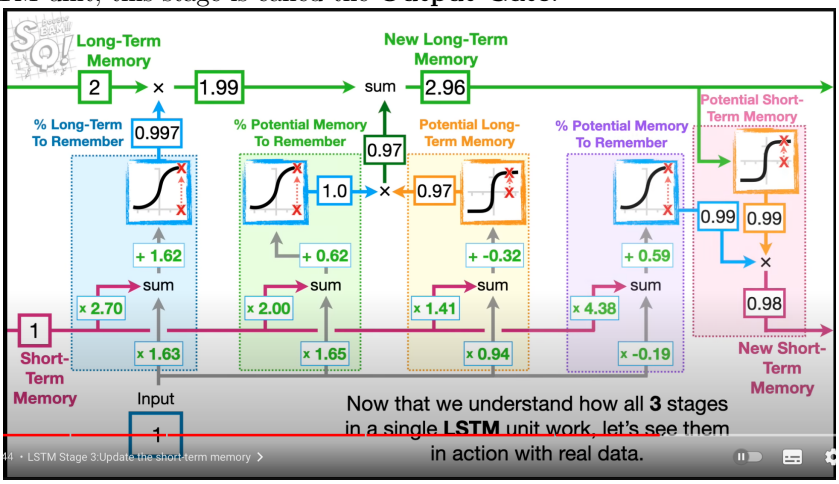
We start by multiplying the short-term memory with its weight, then the input with its weight, sum these two together, and add the bias. This sum represents the x-axis coordninate for a sigmoid activation function. The y-axis coordinate we get from the sigmoid activation function is multiplied with the long term memory. The output from the sigmoid activation function determines what percentage of the long-term memory is remembered. Even though this part of the LSTM determines what percentage of the long-term memory will be remembered, it is usually called the **Forget Gate**. In other words it decides which information should be discarder from the cell state.

The next stage combines the short-term memory and the input to create a potential long-term memory and what percentage of that potential memory to add to the long-term memory. This happens in two 'blocks'. We start with the block with the tanh activation function. We multiply the short-term memory and input by their respective weights, sum these two, and add the bias, like in the first step, but with different weights. This is x-coordinates for the tanh activation function. We get a y-axis corrdinate between -1 and 1, which becomes the **potential memory**. To determine how much of this potential memory to save, we repeat everything, but with different weights, and send it through a sigmoid function. The y-axis coordinate is the percentage, which is multiplied with the the potential memory.

The potential memory multiplied by the percentage, is added to the existing long-term memory, calculated in the first stage (forget stage). Then we get a new long term memory. This part of the LSTM unit determines how we should update the long-term memory, and is called the **Input Gate**. In other words, it decides what new information to add to the cell state.

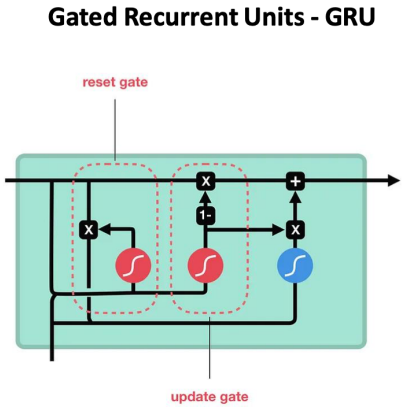
The final stage of LSTM updates the short-term memory. We start with the new long-term memory, and use it as input to the tanh acitvation function. The y-axis coordinates from tanh represents the potential short-term memory. Now the LSTM has to decide how much of the potential short term memory to pass on. This is done using the exact same method as we used two times earlier, with a sigmoid function. The new short-term memory is created by multiplying the

potential short-term memory, by the percentage from the sigmoid activation function. Because the new short-term memory is the ouput from this entire LSTM unit, this stage is called the **Output Gate**.



3.3 Gated Recurrent Units (GRU)

GRUs have a simpler architecture than LSTMs. It has fewer parameters which leads to it being computationally more efficient. For some tasks the performance is comparable to those of LSTM. There are reports exhibiting that GRUs have better performance on smaller datasets.



There are two main gates in a GRU. The update gate determines how much of the past information needs to be passed. It's like deciding how much of the previous memory to keep in the new state. The input (or reset) gate determines how much of the past information to forget/discard during the combination with the current input. A reset and an update gate is used to update the hidden state.

4 More on RNNs

4.1 Advanced use of RNNs

There are several ways to improve the performance and generalisation power of RNNs.

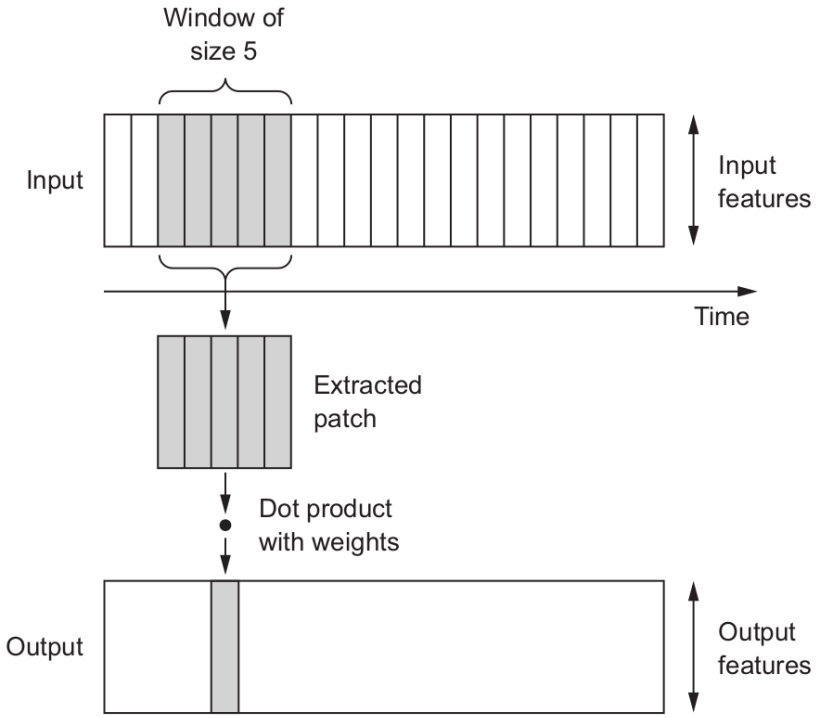
- Recurrent dropout: this is a specific, built-in way to use dropout to fight overfitting in recurrent layers.
- Stacking current layers: this increases the representational power of the network (at the cost of higher computational costs).
- Bidirectional recurrent layers: these present the same information to a recurrent netowrk in different ways, increasing accuracy and mitigating forgetting issues.

4.2 Sequence processing with CNN and RNN

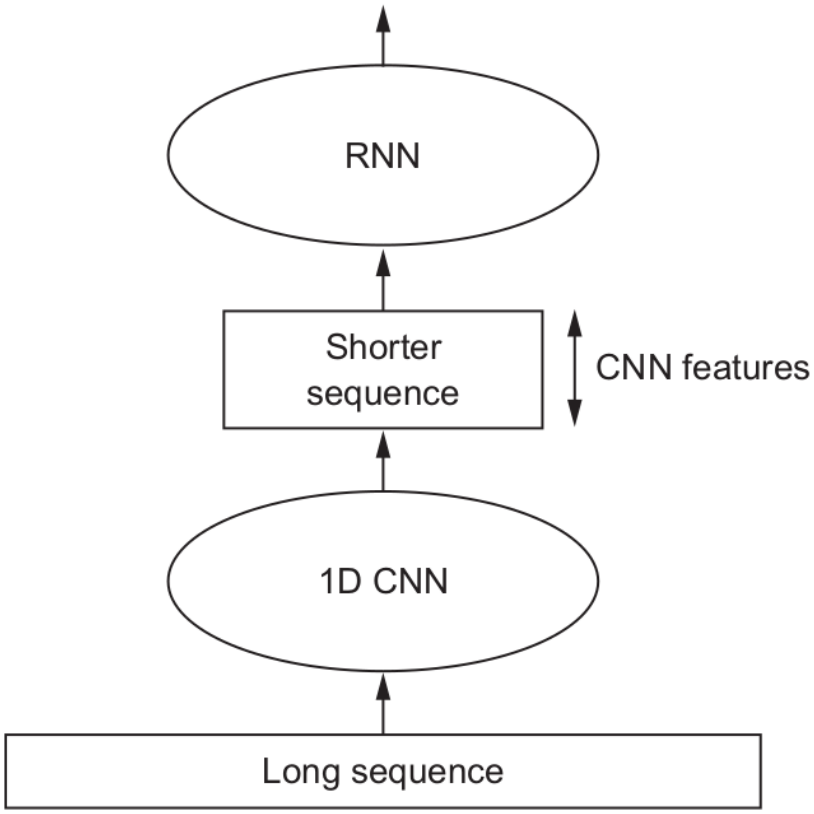
CNN performs particularly well on computer vision problems, with the ability to operate convolutionally, extract features from local input patches and allowing for representation modularity and data efficiency. The same properties that make CNNs excel at computer vision also makes them highly relevant to sequence processing, since time can be treated as a spatial dimension, like the height or width of a 2D image.

1D covnets can be competetive with RNN's on certain sequence-processing problems, usually at a considerably cheaper computational cost. Small 1D covnets can offer a fast alternative to RNNs for simple tasks such as text classification and timeseries forecasting.

1D convolutions extract local 1D patches (sub-sequences) from sequences. Such 1D convolution layers can recognize local patterns in a sequence. The same input transoformations in performed on every patch. A pattern learned at a certain position in a sentence, can later be recognized at a different position, and this makes the 1D covnets translation invariant (for temporal translations).



In previous discussions, we explored 2D pooling operations like 2D average pooling and max pooling, commonly used in convolutional neural networks (convnets) to spatially downsample image tensors. Notably, there exists a 1D equivalent to the 2D pooling operation, which involves extracting 1D patches (or subsequences) from an input and then outputting either the maximum value (max pooling) or the average value (average pooling). This 1D pooling process, similar to its 2D counterpart, is employed to reduce the length of 1D inputs through a subsampling mechanism.



4.3 Training RNNs on time series data

4.3.1 A note about stationarity

Stationarity in a time series refers to a property where the statistical properties of the data remain constant over time.

- **Constant Mean:** The time series data should remain consistent and not exhibit any significant trend or systematic change over time.
- **Constant Variance:** The variance of the data should remain stable. Fluctuations around the mean should not systematically change over time.
- **Constant Autocovariance:** The autocovariance, which measures how the series values at different time points correlate with each other, should not depend on time.

Achieving stationarity is crucial for many time series analyses because non-stationary data can lead to unreliable results and misleading conclusions. Trans-

formations, differencing, and other techniques are often applied to make a time series stationary before performing statistical modeling or forecasting.

4.3.2 Decomposition

Decomposing time series into trend, seasonal components, and residual is a widely employed technique in financial analysis, empowering financial experts to identify enduring trends, spot recurrent market patterns, and uncover anomalies, thereby enriching their capacity to make well-informed investment choices and assess risks.

Furthermore, from a data scientist's point of view, it is useful for guiding the choice of modeling techniques. Seasonal and trend components might be better suited to certain models, while the residuals, representing the unexplained part, can be useful for diagnosing model fit and identifying any patterns that were not captured.

Other useful properties for DS:

- Reveal underlying patterns and structures within a time series
- Identify the long-term direction of the time series
- Helps to identify anomalies or outliers in the data. Unusual patterns that cannot be explained by the trend or seasonality might be indicative of irregular events, errors, or other significant factors.
- Clearer visualizations make it easier to interpret the data and communicate insights to stakeholders.

4.3.3 Preprocessing

Splitting historical stock prices into test and training datasets can be challenging due to the unique characteristics of time series data:

1. **Temporal Dependencies:** Time series data has temporal dependencies, meaning that each data point is influenced by its preceding points. When splitting, it's essential to maintain this chronological order to avoid leaking future information into the past, which can lead to over-optimistic model evaluation.
2. **Non-IID Data:** Time series data is often not independently and identically distributed (non-IID).
3. **Changing Patterns:** Financial markets are dynamic, and stock prices exhibit changing patterns over time due to economic factors.

Given these challenges, practitioners often rely on specialized techniques like

- **Sequential Splitting**

- **Cross-Validation:** Create multiple training and test sets by sliding a window of fixed size over the data.
- **Out-of-Sample Testing:** Remove a certain portion of the data and then forecast a future period.
- **Rolling Window Approach:** Iteratively train the model on a fixed-size window.

5 Transformers

6 Mathematics in RNNs

If the input sequence to an RNN is of length 10, the hidden layer has 5 neurons, and the output layer has 3 neurons, what are the dimensions of the weight matrix Whh (hidden–¿hidden)? Answer: 5x5

You are working on a text classification project where you process batches of sentences using a 3-dimensional tensor named X. The tensor X has a shape of (B, T, F), where B is the batch size, T is the number of timesteps corresponding to the maximum sentence length, and F is the number of features representing each word using one-hot encoded vectors. If the batch size is 32, the maximum sentence length is 10 words, and the vocabulary size for one-hot encoding is 1000, what are the dimensions of the tensor X? Answer: (32, 10, 1000)

In a text classification project, you are using a 3-dimensional tensor X to process a batch of sentences. The shape of X is (B, T, F), where B is the batch size, T is the number of timesteps corresponding to the maximum sentence length, and F is the number of features representing each word using one-hot encoded vectors. Given that the batch size B is 32, the maximum sentence length T is 10 words, and the vocabulary size F for one-hot encoding is 1000, the tensor X has the dimensions (32, 10, 1000). Now, suppose you add a hidden layer of size 16 to the network. What would be the dimensions of the weight matrix Wxh, which connects the input layer to this hidden layer? Answer: (32, 16)

7 Usefull pages and videos

Playlist from StatQuest with Josh Starmer - video 15-22