Class 3: Recurrent neural networks and transformers

Andrew Parnell
andrew.parnell@mu.ie

PRESS RECORD

https://andrewcparnell.github.io/intermediate_ML

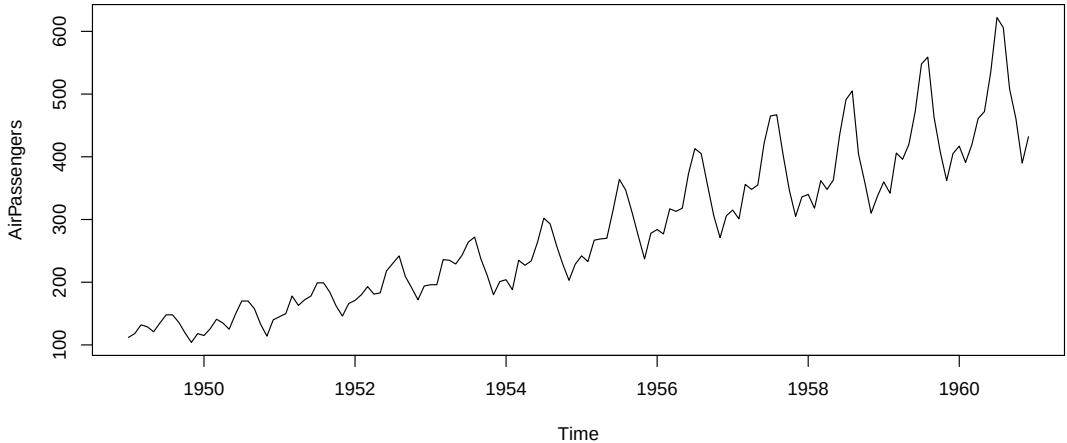
Learning outcomes

Revision from yesterday. Then:

- Learn how to analyse time series data using keras
- Learn how to fit Long Short Term Memory (LSTM) models
- Learn the basics of transformer models

Time series data

```
data(AirPassengers)
plot(AirPassengers)
```



From simple to recurrent neural networks (RNNs)

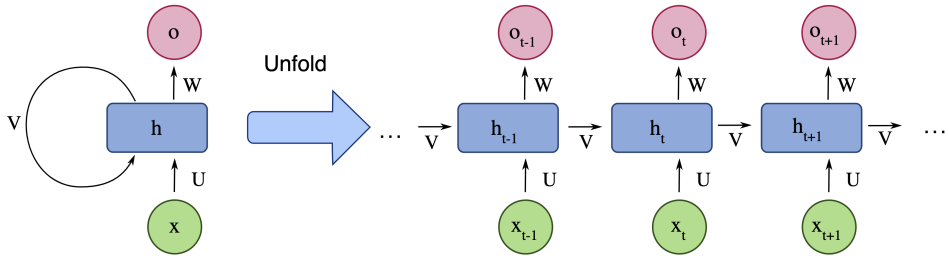
Three main differences:

1. RNNs incorporate loops to allow information persistence, enabling them to maintain a 'memory' of previous inputs in the sequence
2. The architecture of an RNN includes connections between hidden layers across time steps, forming a directed cycle within the network.
3. The training of an RNN requires a specific adaptation of the backpropagation algorithm known as Backpropagation Through Time (BPTT)

How to fit an RNN in keras

```
look_back <- 3
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 32,
                   input_shape = list(look_back, 1)) %>%
  layer_dense(units = 1)
```

Architecture of RNN



It's fundamentally the same as a standard neural network, but the hidden units have their own previous values as inputs as well

From https://en.wikipedia.org/wiki/Recurrent_neural_network

Backpropagation through Time (BPTT)

- Because the hidden units have themselves as previous values, standard backpropagation doesn't work
- Instead the network is 'unfolded' so that it looks like a standard NN
- This makes it computationally quite expensive, so often it's truncated at a certain level

Parameter set up in RNNs

- We have some common values we need to set just like in standard neural networks. This includes the number of hidden units, an activation function, batch size, learning rate, etc
- In particular we have to feed in a specific length of the time series into the model (the `look_back`) parameter in the above
- Dropout and regularisation gets very important. If you can ignore old values (or make them very small) the model can avoid over fitting
- It can also be important to initialise the recurrent weights (i.e. those that are specific to the recurrent loops) to avoid ...

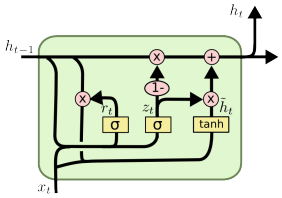
Vanishing and Exploding Gradients

- When unfolding the network, the derivative of the loss function can end up getting very small or very large - this is the *vanishing* or *exploding* gradient problem
- It's because you end up multiplying lots of numbers together. As soon as they're not 1 they will explode or vanish
- A common solution to the problem is to change the network slightly to a *Long Short-Term Memory* (LSTM) or *Gated Recurrent Units* (GRU) model

LSTM and GRU

- We will explore four different ways to fit supervised time series type models in R: RNNs (covered), GRUs, LSTMs, and Transformers
- LSTMs have a slightly more complex network structure with different gates to control how much previous information is used in the forecasts
- GRUs are a compromise between the simplicity of RNNs and the capability of LSTMs but with two gates (reset and update gates), and are a bit more computationally efficient than LSTMs

GRUs - picture



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Inputs are x_t, outputs are h_t, W are weights, $\sigma()$ are activiation functions, the . is element-wise multiplication and * is matrix multiplication

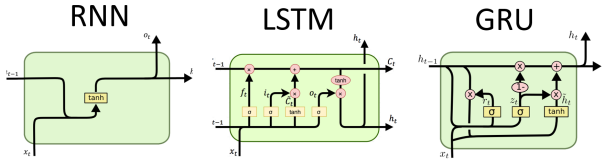
From http://colah.github.io/posts/2015-08-Understanding-LSTMs/

More on GRUs

Two key gates:

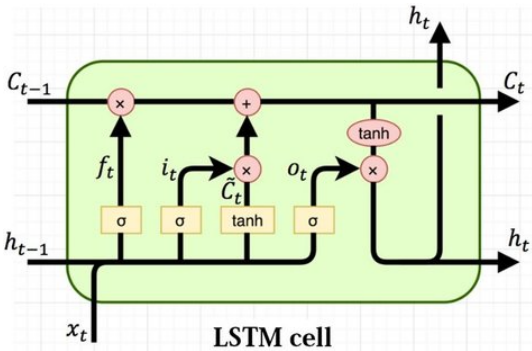
- Update Gate: The update gate in a GRU determines how much of the previous hidden state should be retained and how much of the new candidate hidden state should be accepted. By performing this weighted combination, it facilitates the model in capturing dependencies over various time scales, allowing it to decide whether to keep the existing memory or update it with new information.
- Reset Gate: The reset gate assists in controlling how much of the previous hidden state will influence the candidate hidden state for the current time step. By allowing the model to forget certain portions of the previous hidden state selectively, it provides the flexibility to make the current hidden state less dependent on the historical context when necessary, effectively enabling the model to capture both short-term and long-term dependencies.

RNNs vs LSTMs vs GRUs



From http://dprogrammer.org/rnn-lstm-gru

LSTM formula



$$
\begin{aligned}
i_t &= \sigma\left(x_t U^i + h_{t-1} W^i\right) \\
f_t &= \sigma\left(x_t U^f + h_{t-1} W^f\right) \\
o_t &= \sigma\left(x_t U^o + h_{t-1} W^o\right) \\
\tilde{C}_t &= \tanh\left(x_t U^g + h_{t-1} W^g\right) \\
C_t &= \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right) \\
h_t &= \tanh(C_t) * o_t
\end{aligned}
$$

From https://www.researchgate.net/publication/329362532_Designing_neural_network_based_decoders_for_surface_codes

LSTMs - the three gates

- Forget Gate: assesses which information from the memory cell is no longer necessary and should be discarded, allowing the LSTM to forget irrelevant parts of the sequence, which is essential in maintaining the memory cell's relevance and efficiency as it processes new inputs.
- Input Gate: decides which information is retained from the current input and previous hidden state, and adds this selected information to the cell
- Output Gate: determines which part of the current state will be used to compute the hidden state of the current time step

Fitting RNNs, LSTMs and GRUs

- All very similar code:

```
model_rnn <- keras_model_sequential() %>%
  layer_simple_rnn(units = 50,
                   input_shape = list(look_back, 1)) %>%
  layer_dense(units = 1)
model_gru <- keras_model_sequential() %>%
  layer_gru(units = 50,
            input_shape = c(look_back, 1)) %>%
  layer_dense(units = 1)
model_lstm <- keras_model_sequential() %>%
  layer_lstm(units = 50,
             input_shape = list(look_back, 1)) %>%
  layer_dense(units = 1)
```

A final note: time series classification

- Once you've got the basics of these models it's pretty easy to fit them to a data set
- See `stockmarket_keras_gru_classification.R` or `earthquakes_keras_lstm_classification.R` for an example that fits a time series classification model
- Here the data are stock market data and we are predicting whether the values go up or down

Introduction to Transformers

- In 2017 the paper 'Attention is all you need' came out, by a number of authors from Google, which introduces a new type of layer / structure called a Transformer
- The main use case was on language data; simply predicting the next word (or token) in a sentence
- This was actually a simplification of other models previously used for NLP, but works better and led to all of the amazing Transformer models we see today
- Whilst originally built for text data, they can be adapted for any sequential learning task

Key advantages of Transformers

- **Self-Attention Mechanism**: allows simultaneous consideration of all parts of the input sequence, and can the capture long-range dependence
- **Parallelization and Efficiency**: permit parallelisation across the entire sequence, avoiding the need to complex unstacking like in RNNs
- **Scalability and Pre-Training**: new models like BERT and GPT can be pre-trained on vast datasets and fine-tuned for specific tasks

Queries, keys, and attention scores

Underlying the attention mechanism are sets of vectors that capture the attention mechanism. The three key parts are:

- **Query Vector**: associated with the specific position in the sequence that you are focusing on. If the query vector corresponds to a particular word, it will interact with the key vectors to determine how much attention that word should pay to every other word in the sequence.
- **Key Vector**: corresponds to all the positions in the sequence (e.g. words in a sentence), including the one associated with the query vector.
- **Attention scores**: computed via the dot product of the query vector with the key vectors, followed by some standardisation and applying and activation function.

The attention scores signify the importance or relevance of other words in the sequence to the word represented by the query vector.

Attention is all you need

- The self-attention mechanism computes attention scores by taking the dot product of a query vector with the key vectors from all other positions in the sequence, followed by scaling and applying a softmax function
- These attention scores are then used to help the model to focus more on parts of the input sequence that are relevant to the particular context or position (or word) being evaluated
- Capturing Dependencies: By considering all parts of the input simultaneously, self-attention allows the model to capture long-range dependencies and intricate relationships within the text, which contributes to the transformer's ability to understand and generate coherent and contextually relevant language.

An extra wrinkle: embeddings

- In transformer models, input text is mapped from the discrete textual elements (tokens) into a continuous vector using an **embedding layer**
- Alongside the embedding **positional encodings** are included so that the model can recognise the order of the words in the sentence
- It is these embeddings and positional encodings that are fed into the attention mechanism, not the word values
- All of these embeddings and encodings are often high dimensional to capture as much information as possible

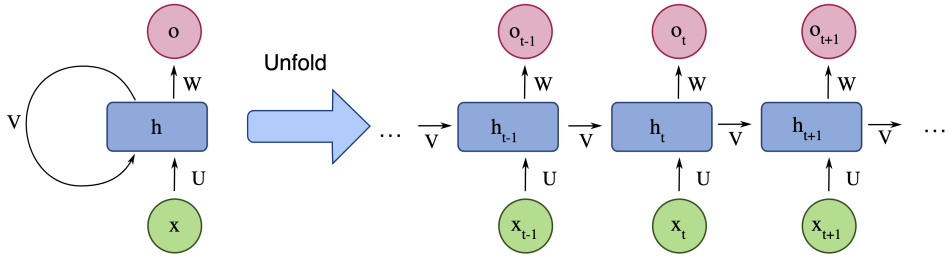
Multi-Head Attention

- Rather than just use one attention mechanism, the Transformer model uses multiple attention layers (called 'heads') to capture all of the different types of dependencies that can occur in text (or time series)
- Once all the multiple heads attention scores are calculated the values are all summed (with more weights), and transformed back into probabilities for the next forecast word / data point

Pre-training

- Pre-training is commonly (but not necessarily) used with Transformers
- It involves fitting a model on a large dataset before fine-tuning for a specific task
- We first learn about features (or embeddings) from unsupervised or lightly supervised data. For example you might remove a word from some of the sentences in your training data and ask it to predict it, or remove a patch from your image and ask it to predict that
- The idea is to reduce the amount of labelled data required to fit the model

Transformers in pictures



From https://arxiv.org/abs/1706.03762

Different types of transformers

The two most famous ones are:

- **BERT (Bidirectional Encoder Representations from Transformers)**. Uses bidirectional self-attention to pre-train representations from unlabeled text, followed by fine-tuning for specific tasks.
- **GPT (Generative Pre-trained Transformer)**. Trained as a language model in an unsupervised manner, then fine-tuned for various tasks; uses a unidirectional (left-to-right) attention mechanism

Implementing Transformers in R

A time series classification example: see ‘FordA_keras_transformer.R’

Summary

- We have covered lots of different models for time series models, including RNNs, GRUs, LSTMs, and Transformers
- For many smaller data sets, the simpler models will be fine (also don't forget standard statistical models such as ARIMA)
- Transformers are very new, and code/descriptions here are likely to go out of date quickly
- Many more Transformer models are possible but not widely available in R yet: https://github.com/huggingface/transformers