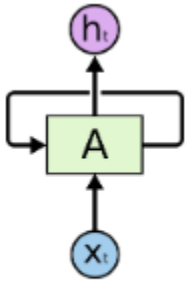
RNNs and LSTM

Traditional neural networks can’t use their previous knowledge about previous events to reason for new upcoming/unseen events, but **RNNs** (recurrent neural networks) address this issue. They are networks with loops in them, allowing information to persist.

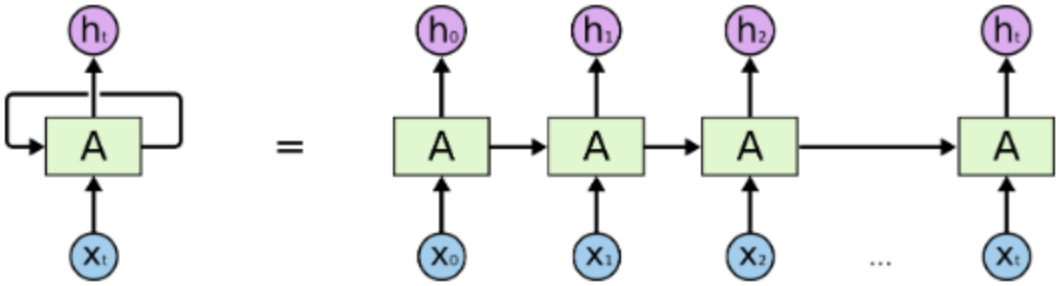
Recurrent Neural Networks (RNNs) differ from regular neural networks in how they process information. While standard neural networks pass information in one direction i.e from input to output, RNNs feed information back into the network at each step.

The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



Recurrent Neural Networks have loops.

They aren’t all that different from a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



An unrolled recurrent neural network.

RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.

This unrolling enables backpropagation through time (BPTT) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.

RNNs share similarities in input and output structures with other deep learning architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks where each dense layer has distinct weight matrices, RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs the hidden state H_i is calculated for every input X_i to retain sequential dependencies. The computations follow these core formulas:

1. Hidden State Calculation:

$$h = \sigma(U \cdot X + W \cdot h_{t-1} + B)$$

Where:

- h represents the current hidden state.
- U and W are weight matrices.
- B is the bias.

2. Output Calculation:

$$Y = O(V \cdot h + C)$$

The output Y is calculated by applying O an activation function to the weighted hidden state where V and C represent weights and bias.

3. Overall Function:

$$Y = f(X, h, W, U, V, B, C)$$

This function defines the entire RNN operation where the state matrix S holds each element s_i representing the network's state at each time step i.

At each time step RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory that retains information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

The current hidden state h_t depends on the previous state h_t−1 and the current input x_t and is calculated using the following relations:

1. State Update:

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

Where:

- h_t represents the current state.
- h_t-1 is the previous state.
- x_t is the input at the current time step.

2. Activation Function Application:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

Where, W_hh is the weight matrix for the recurrent neuron and W_xh is the weight matrix for the input neuron.

3. Output Calculation:

$$y_t = W_{hy} \cdot h_t$$

Where y_t is the output and W_hy is the weight at the output layer.

These parameters are updated using backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as backpropagation through time.

In BPTT, gradients are backpropagated through each time step. This is essential for updating network parameters based on temporal dependencies.

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, but it can't always do that. As the gap between context (the gap between the relevant information and the point where it is needed) grows, the RNNs become unable to learn (the vanishing gradient problem or the exploding gradient problem). In theory, RNNs are absolutely capable of handling such "long-term dependencies." Sadly, in practice, RNNs don't seem to be able to learn them.

- *Vanishing Gradient:* When training a model over time, the gradients which help the model learn can shrink as they pass through many steps. This makes it hard for the model to learn long-term patterns since earlier information becomes almost irrelevant.

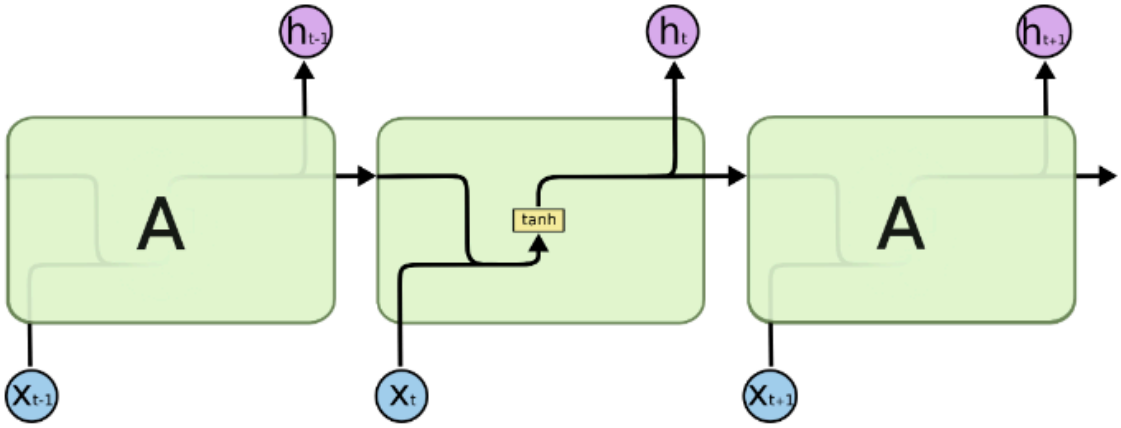
- *Exploding Gradient:* Sometimes gradients can grow too large causing instability. This makes it difficult for the model to learn properly as the updates to the model become erratic and unpredictable.

Both of these issues make it challenging for standard RNNs to effectively capture long-term dependencies in sequential data. To fix this problem, LSTM was created as a solution.

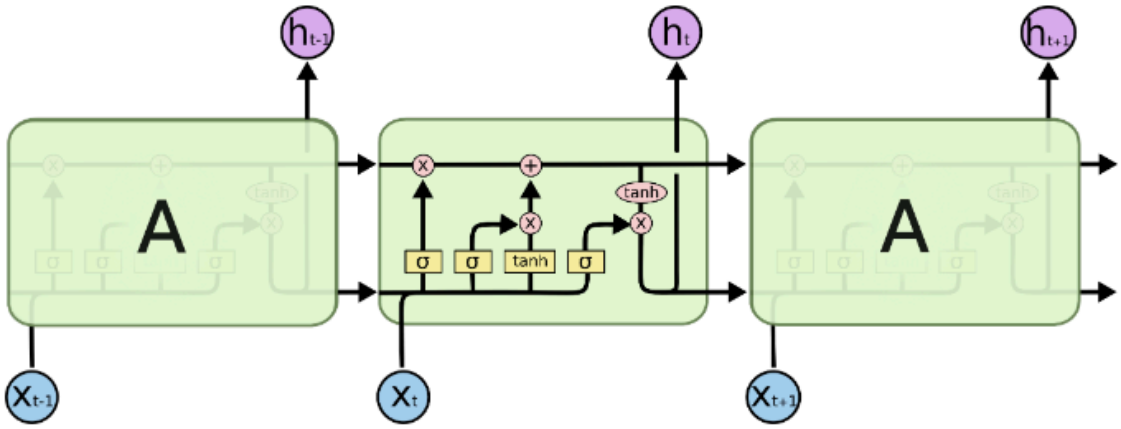
Long Short-Term Memory (LSTM) is an enhanced version of the Recurrent Neural Network (RNN) designed by Hochreiter and Schmidhuber, which can capture long-term dependencies in sequential data making them ideal for tasks like language translation, speech recognition and time series forecasting.

Unlike traditional RNNs which use a single hidden state passed through time, LSTMs introduce a memory cell that holds information over extended periods addressing the challenge of learning

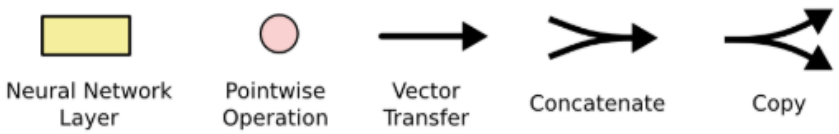
long-term dependencies. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!



The repeating module in a standard RNN contains a single layer.



The repeating module in an LSTM contains four interacting layers.



LSTM architectures involves the memory cell which is controlled by three gates:

1. Input gate: Controls what information is added to the memory cell.
2. Forget gate: Determines what information is removed from the memory cell.
3. Output gate: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Links:
https://colah.github.io/posts/2015-08-Understanding-LSTMs/
https://www.geeksforgeeks.org/machine-learning/introduction-to-recurrent-neural-network/
https://www.geeksforgeeks.org/deep-learning/deep-learning-introduction-to-long-short-term-memory/
https://distill.pub/2016/augmented-rnns/
https://karpathy.github.io/2015/05/21/rnn-effectiveness/