Recurrent Neural Networks (RNNs) and Transformers

CSC413H1S 2021 Tutorial 7

Haotian Cui

Based on the tutorials of CSC413 fall 2020

The Big Picture

Many domains feature sequences of data with temporal dependencies:

- Natural Language Processing (NLP)
- Time series forecasting (Healthcare, Finance, etc.)

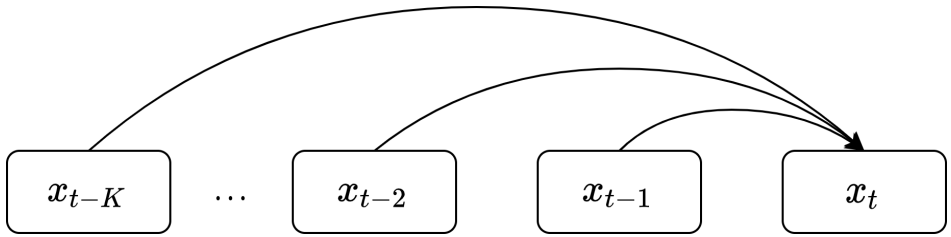
Common tasks:

- Predict the next value in a sequence
- Convert data sequence to equivalent sequence in another space (translation)
- Classify the entire sequence into specific class.

Related Methods

How do we model data which contains time dependency?

Autoregressive methods: Predict next data observation as a linear equation of previously observed data points.



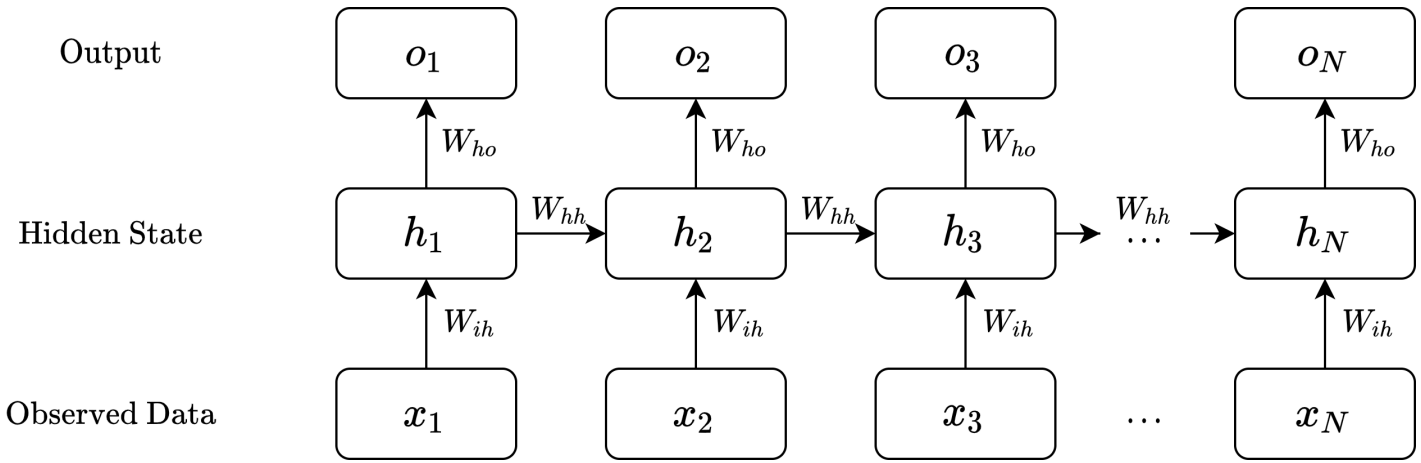
- Ex: $x_t = w_1 * x_{t-1} + w_2 * x_{t-2} + ... + w_K * x_{t-K}$
- Representational ability is limited. Only looks K steps back in time!

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) offer several advantages:

- Non-linear hidden state updates allows high representational power.
- Can represent long term dependencies in hidden state (theoretically).
- Shared weights, can be used on sequences of arbitrary length.

Recurrent Neural Networks



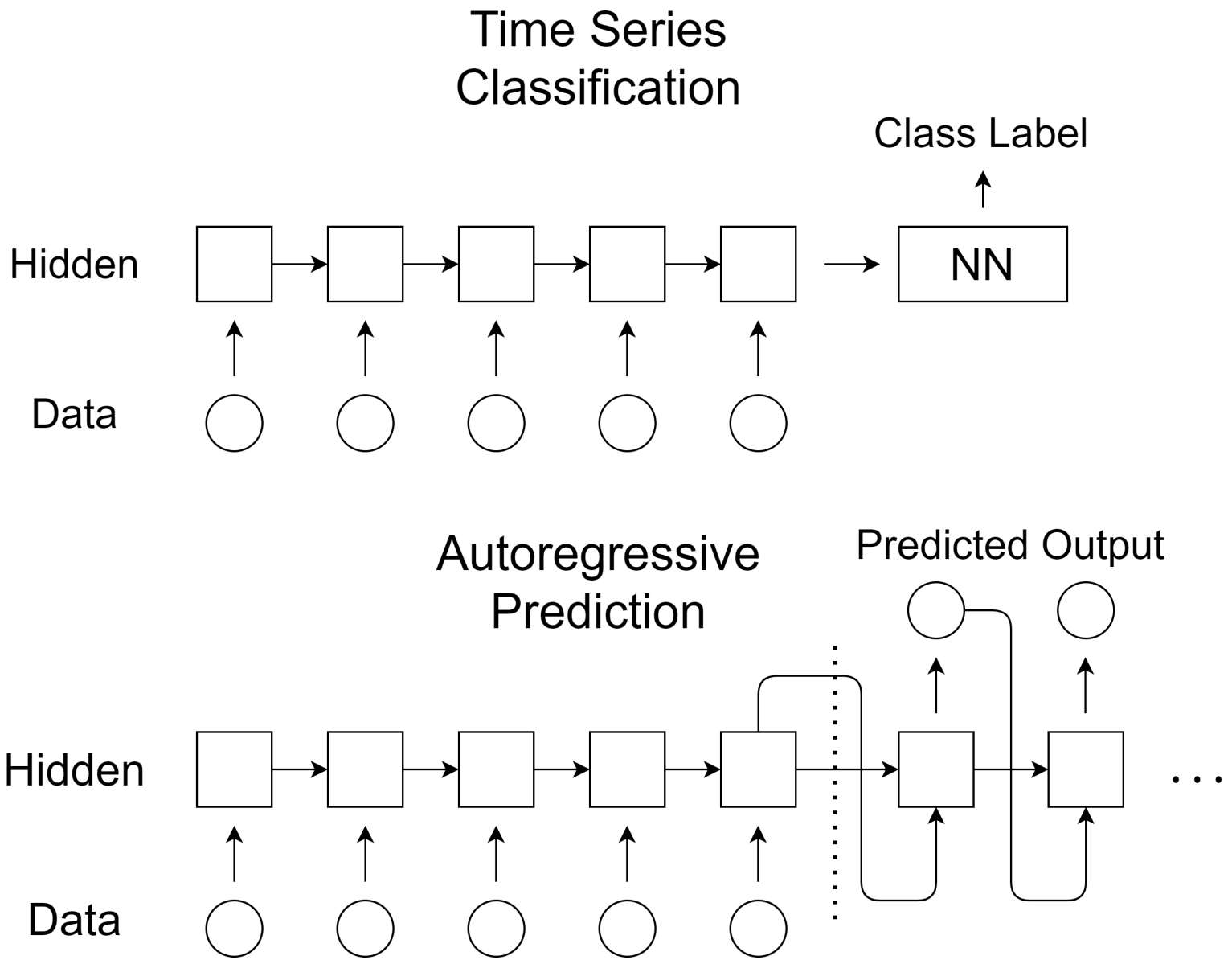
$$\mathbf{h_t} = W_{ih}\ \mathbf{x_t} + W_{hh}\ \mathbf{h_{t-1}} + b_{ih} + b_{hh} \tag{1}$$

$$\mathbf{a_t} = \tanh(\mathbf{h_t}) \tag{2}$$

$$\mathbf{o_t} = \text{softmax}(W_{ho}\ \mathbf{a_t} + b_{ho}) \tag{3}$$

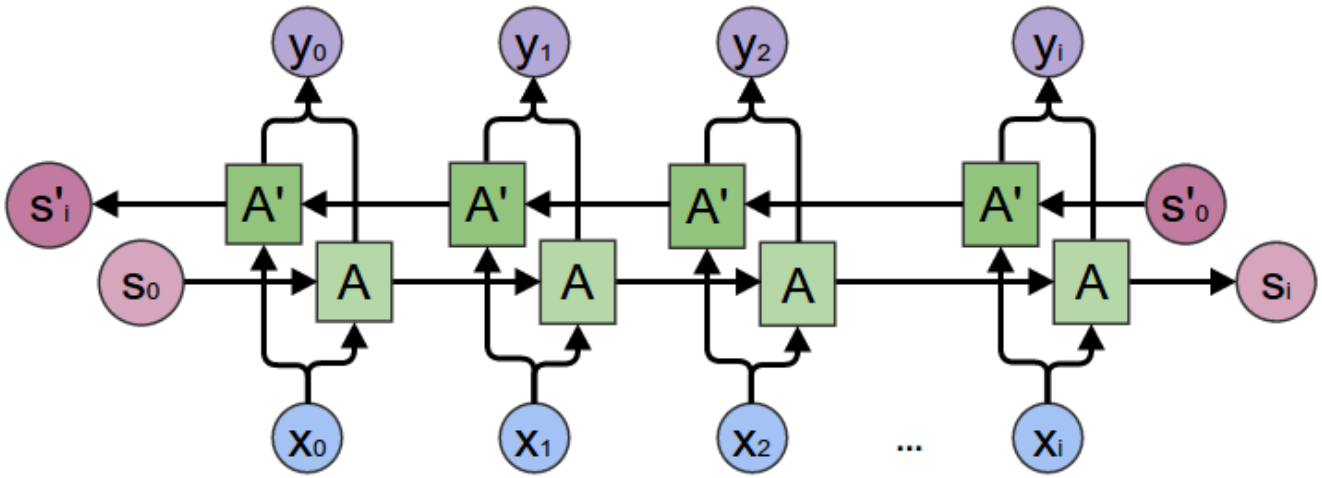
Weight matrices are shared, meaning sequence can be arbitrary length.

Applications of RNN



RNN Modifications: Bidirectional RNNs

Bidirectional RNNs (Schuster and Paliwal 1997)



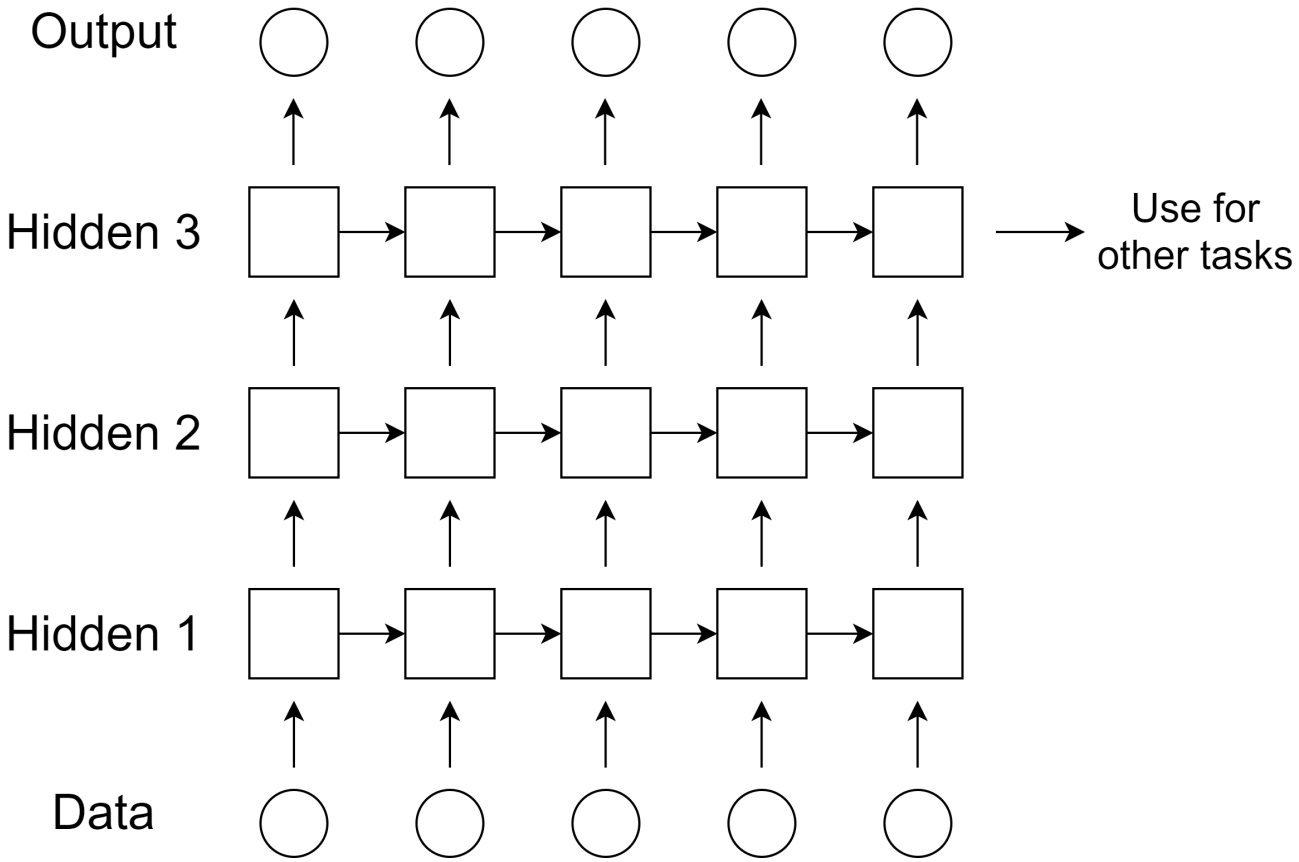
Source: `http://colah.github.io/posts/2015-09-NN-Types-FP/`

Runs two separate RNN in opposite directions, and concatenate output.

- Access to the future values can improve RNN representations.
- Example: The _____ is a flightless bird that lives in Antarctica.

RNN Modifications: Stacked RNNs

Stacked RNNs



PyTorch Implementation

CLASS `torch.nn.RNN(*args, **kwargs)` [SOURCE]

Applies a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

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

where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then ReLU is used instead of $\tanh$.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*). Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

Long Term Dependencies

Prediction tasks in time series often requires long term information from observations ago.

Example: "The flamingo is a pink bird which lives in warmer regions of the world, and they like to speak in run-on sentences for the sake of this example. Surprisingly, _____ are not naturally pink, but rather appear pink because they are always embarrassed."

Task: Fill in the blank. The RNN needs to store information about the subject for an arbitrarily long length. Experiments show RNNs have a hard time remembering.

Gradient Issues

Consider: Deepest feed forward models contain up to ~ 150 layers, but the type of sequential data used in RNNs can easily exceed this in length. What happens to the gradient?

Some intuition:

- Backprop is chain rule, i.e., recursive multiplication of many VJPs.
- The derivative of the Tanh / Sigmoid activation is always less than 1.
- Multiplying gradient with enough activation Jacobians will cause the gradient will go to 0.
- Gradients can explode with ReLU activations (since its unbounded).

Hacky fixes: Gradient clipping to prevent explosion.

LSTMs

Long Short-Term Memory (LSTM) units introduce long term cell state, allowing gradients to flow without being forced to change.

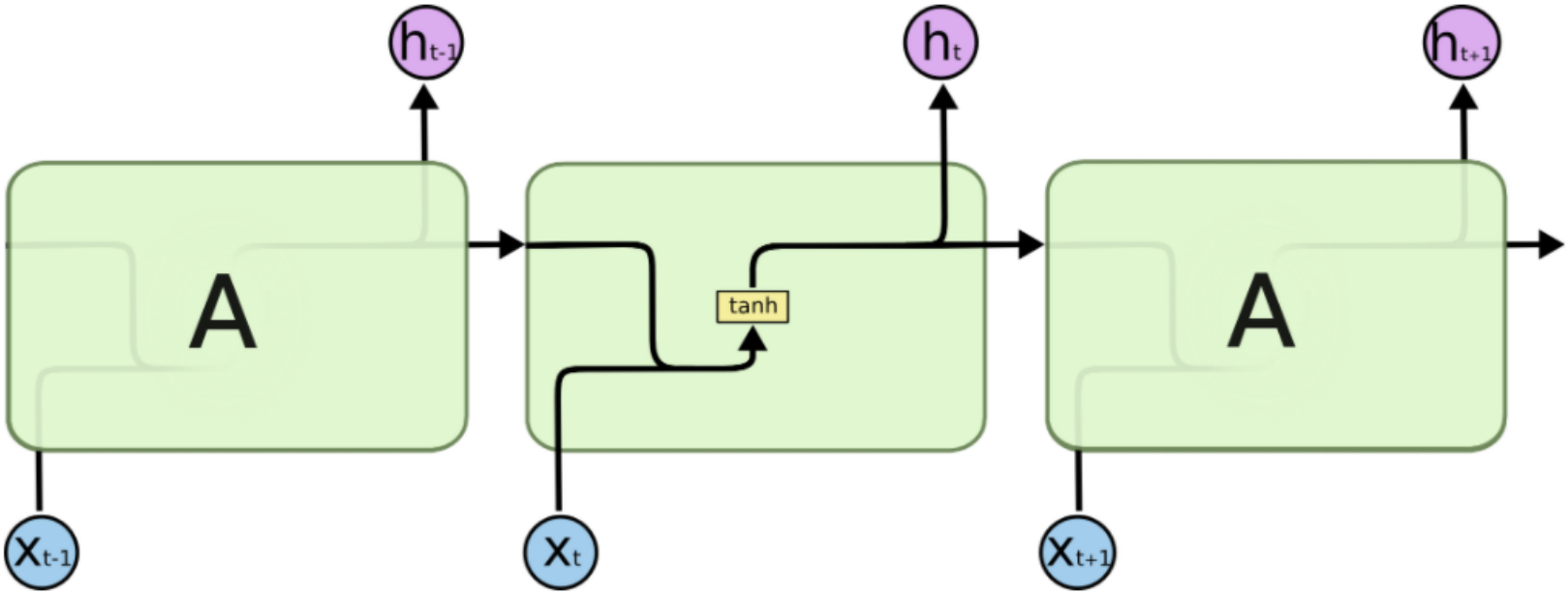
- Well, that description was unclear. Lets break it down!

The following figures are directly taken from Chris Olah's blog: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

- Avoiding citing each image to save space, but I claim no credit!
- Side note: his blog contains many top tier tutorials, and is worth checking out.

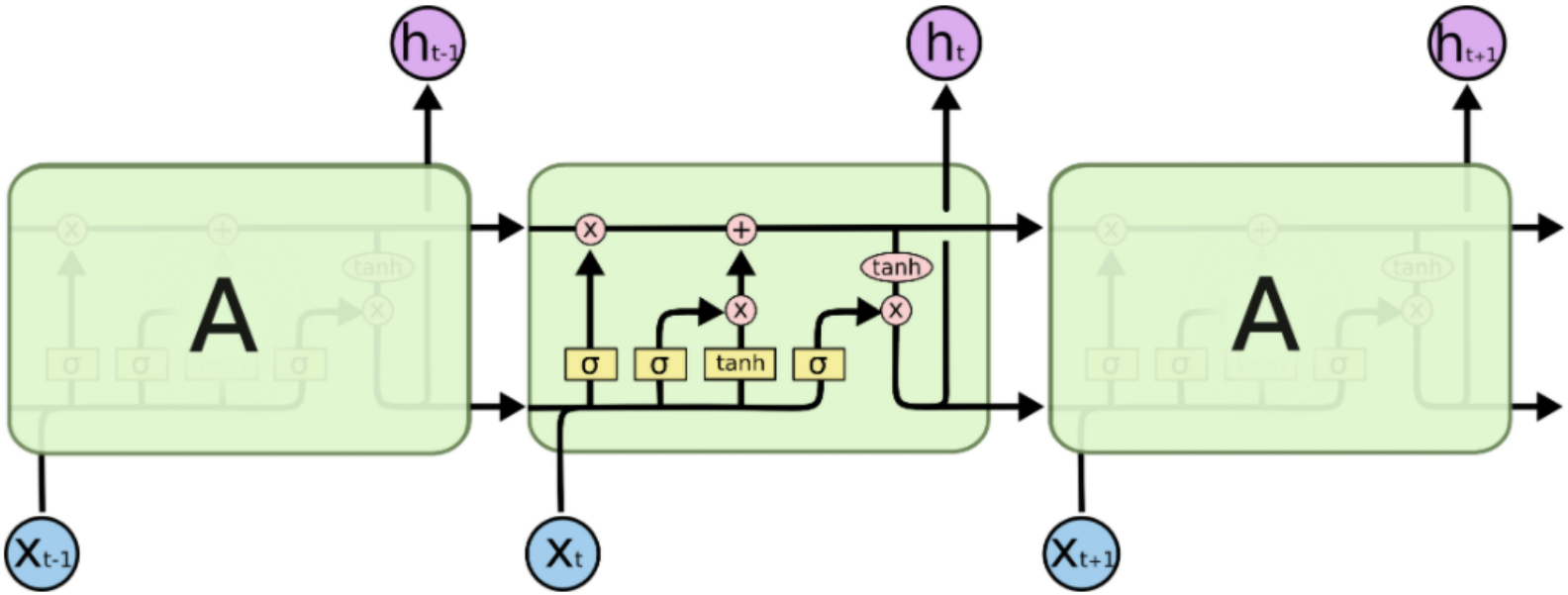
LSTMs

RNN Diagram



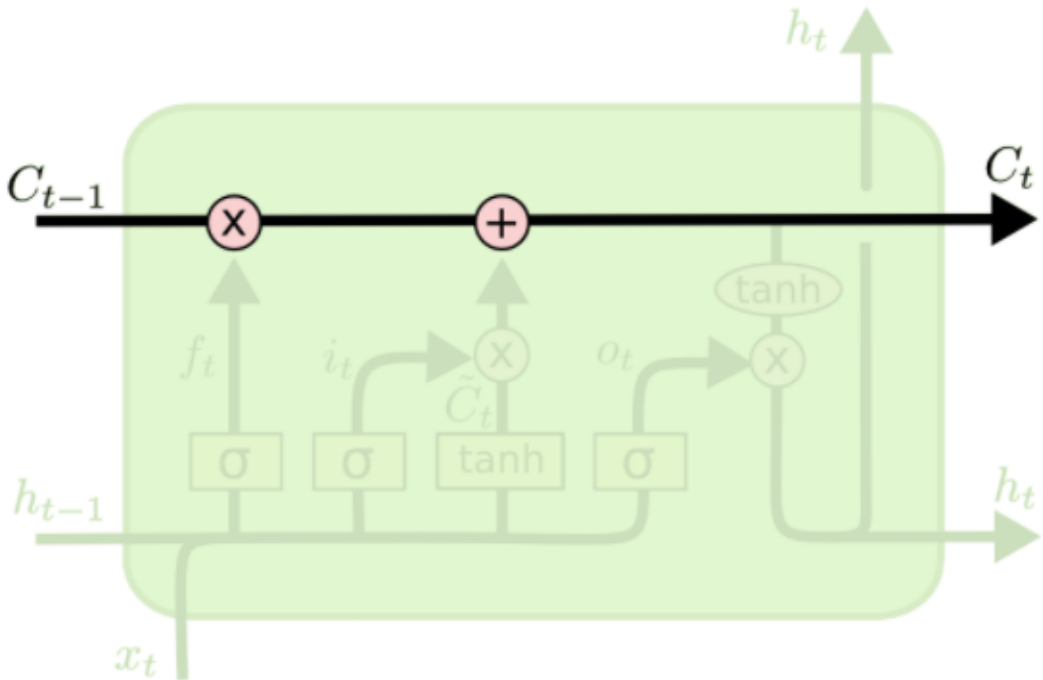
LSTMs

LSTM Diagram



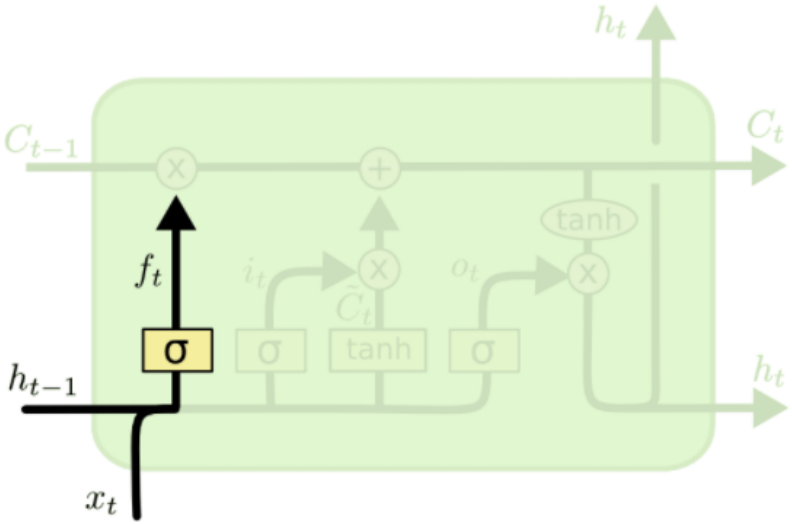
LSTMs

Personally, I think of cell state as long-term memory. Protected by gates (next slides) from unwanted gradient updates.



LSTMs

Forget Gate: Deletes information from cell state.

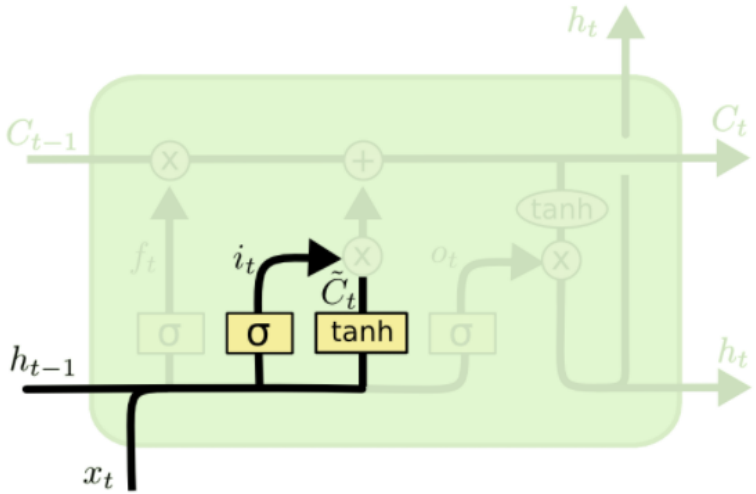


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- Takes linear combination of x_t and $h_{t=1}$.
- Sigmoid activation squashes to range 0 (forget) to 1 (remember).
- Output multiplied element-wise with cell state to forget certain pieces of long term information (e.g., if the subject switches).

LSTMs

Input Gate: Adds information to cell state.



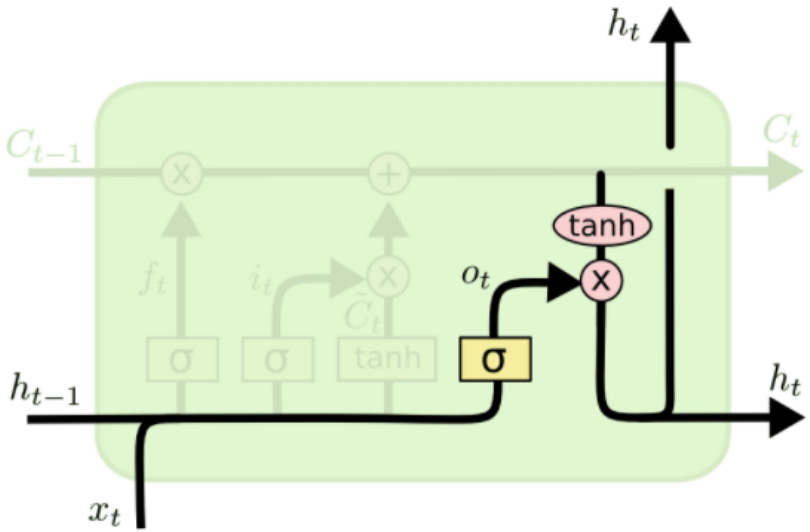
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

- First function determines which cell dimensions to update.
- Second function determines what values to update cell state with.
- Output of input gate is added to cell state.

LSTMs

Output Gate: Decides what information to output from cell state.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

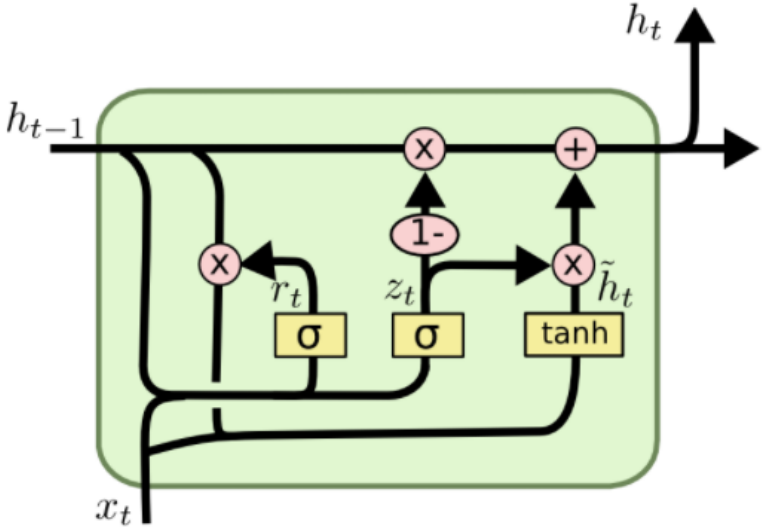
$$h_t = o_t * \tanh\left(C_t\right)$$

- Afterwards, hidden and cell states passed to next cell.

GRU

LSTMs are pretty complex, and require many weights.

Gated Recurrent Units (GRUs) (Cho et al. 2014) simplify LSTMs, and should perform roughly as well.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Merge cell and hidden states, but keep the concept of gating updates to hidden state.

Conclusions

So do LSTMs actually solve the vanishing gradient problem? Kinda!

- Many deployed real-world applications.
  Powered Google Translate for many years.
- Long term dependencies still challenging in reality.

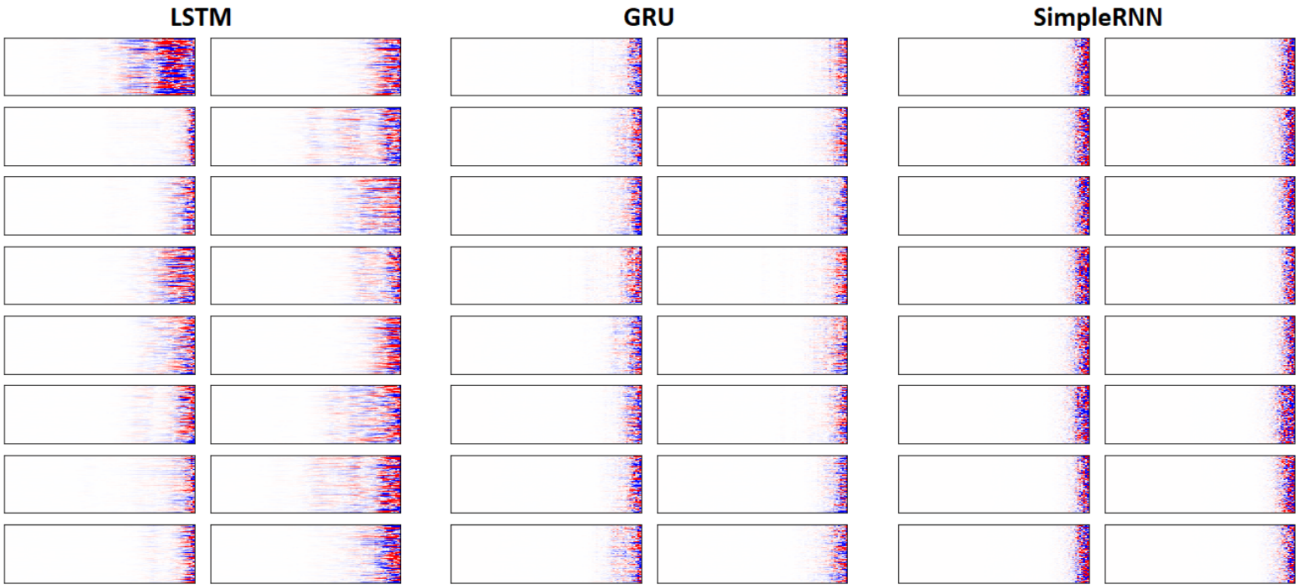


Figure: Heatmap of gradient flow mapped out by depth.
Source: `https://github.com/OverLordGoldDragon/see-rnn`

References

Zhengping Che et al. "Recurrent neural networks for multivariate time series with missing values". In: *Scientific reports* 8.1 (2018), pp. 1–12.

Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

We'd like an architectural primitive that is:

- Ideally feed-forward
- Can facilitate between-token interactions
- Can model long dependences easily.

Attention to the rescue!

- There are many forms of attention. Today we'll focus on **scaled dot product attention.**

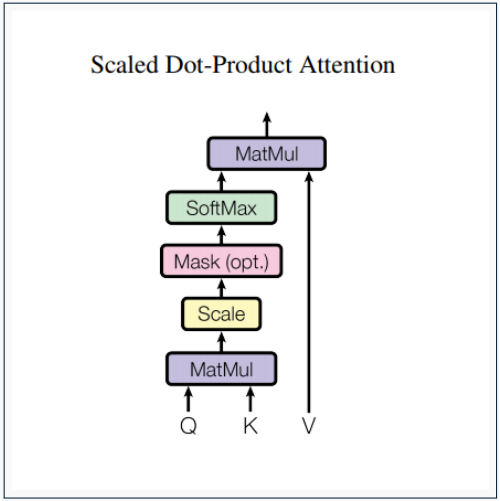
Attention

- **Similarity:** Dot product between keys and queries.
- **Interesting theorem:** In high dimensions, two randomly sampled [1] vectors are almost always approximately perpendicular to each other.
- **Normalization:** Softmax along the keys/values!
- **Result:** Scaled dot product attention.
- We get the following attention mechanism:

$$\mathbb{A}(Q, K, V) = V(softmax(\frac{K^T Q}{d_{kq}})) \tag{2}$$

[1] From, lets say, a isotropic multivariate Gaussian distribution.

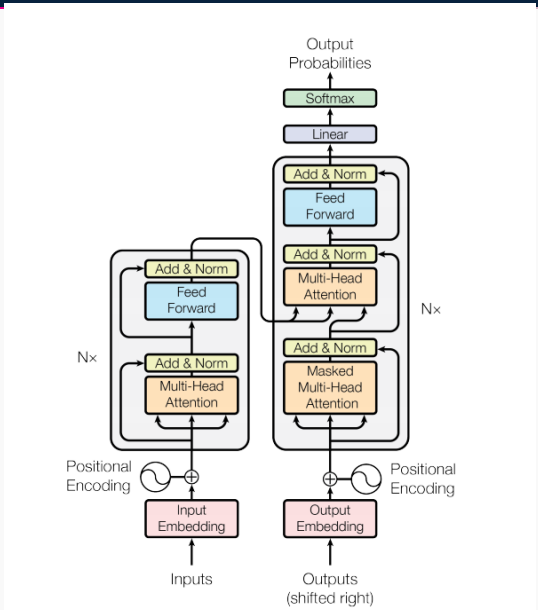
Attention


Scaled Dot-Product Attention
MatMul
SoftMax
Mask (opt.)
Scale
MatMul
Q
K
V

Self-Attention

- What is self-attention?
- Use the same tensor for keys, values and queries!
- What are the keys/queries/values in a self attention layer processing sentence
- The features corresponding to each token!

Transformers



Pretraining Language Models

- Can we use large amounts of text data to pretrain language models?
- Considerations:
  - How can we fuse both left-right and right-left context?
  - How can we facilitate non-trivial interactions between input tokens?
- Previous approaches:
  - ELMO (Peters. et. al., 2017): Bidirectional, but shallow.
  - GPT (Radford et. al., 2018): Deep, but unidirectional.
  - BERT (Devlin et. al., 2018): Deep and bidirectional!

BERT Workflow

- The BERT workflow includes:
  - Pretrain on generic, self-supervised tasks, using large amounts of data (like all of Wikipedia)
  - Fine-tune on specific tasks with limited, labelled data.
- The pretraining tasks (will talk about this in more detail later):
  - Masked Language Modelling (to learn contextualized token representations)
  - Next Sentence Prediction (summary vector for the whole input)

Resources

The transformer section of this tutorial is influenced by the fantastic talk by Lukasz Kaiser on transformers:
https://www.youtube.com/watch?v=rBCqOTEfxvgt=1704s

Hugging Face and tutorial notebooks: https://huggingface.co/transformers/notebooks.html

The illustrated transformers series: http://jalammar.github.io/illustrated-bert/