Recurrent Neural Networks

Milan Straka

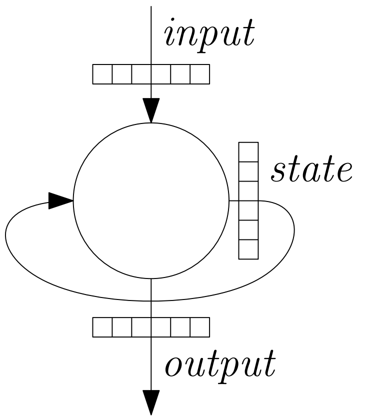
March 28, 2022

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

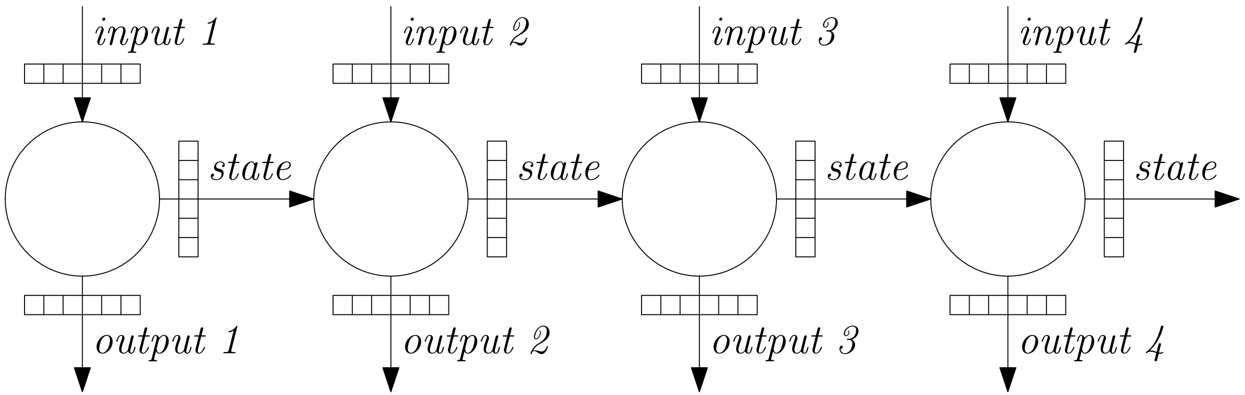
Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

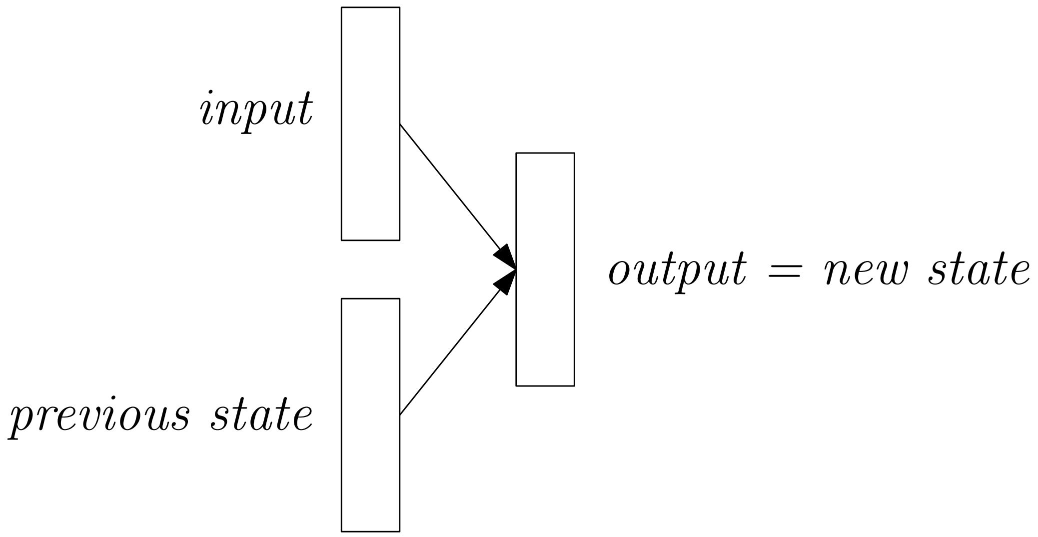
Recurrent Neural Networks

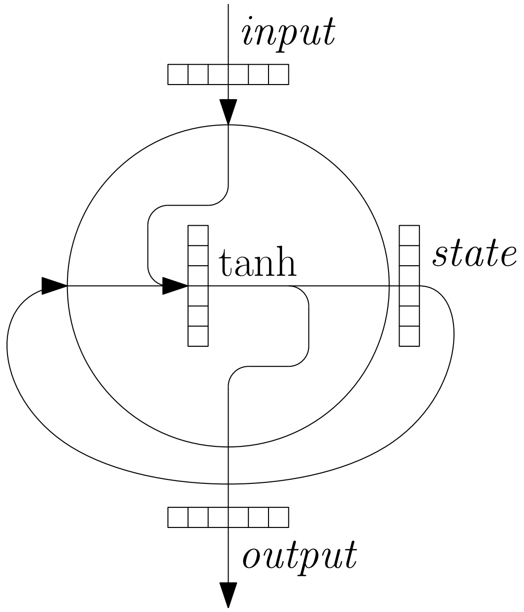
Single RNN cell



Unrolled RNN cells







Given an input $\boldsymbol{x}^{(t)}$ and previous state $\boldsymbol{h}^{(t-1)}$, the new state is computed as

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}).$$

One of the simplest possibilities (called `SimpleRNN` in TensorFlow) is

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{V}\boldsymbol{x}^{(t)} + \boldsymbol{b}).$$

Basic RNN Cell

Basic RNN cells suffer a lot from vanishing/exploding gradients (the so-called **challenge of long-term dependencies**).

If we simplify the recurrence of states to just a linear approximation

$$\boldsymbol{h}^{(t)} \approx \boldsymbol{U}\boldsymbol{h}^{(t-1)},$$

we get $\boldsymbol{h}^{(t)} \approx \boldsymbol{U}^t\boldsymbol{h}^{(0)}$.

If $\boldsymbol{U}$ has an eigenvalue decomposition of $\boldsymbol{U} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$, we therefore get

$$\boldsymbol{h}^{(t)} \approx \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\boldsymbol{h}^{(0)}.$$

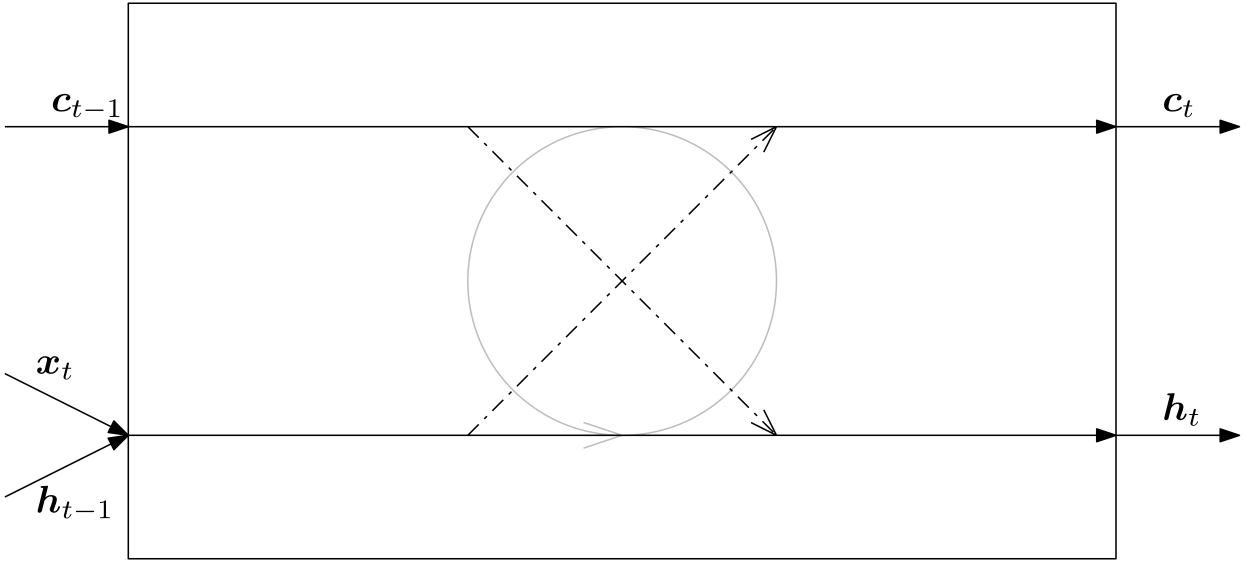
The main problem is that the *same* function is iteratively applied many times.

Several more complex RNN cell variants have been proposed, which alleviate this issue to some degree, namely **LSTM** and **GRU**.

Hochreiter & Schmidhuber (1997) suggested that to enforce *constant error flow*, we would like

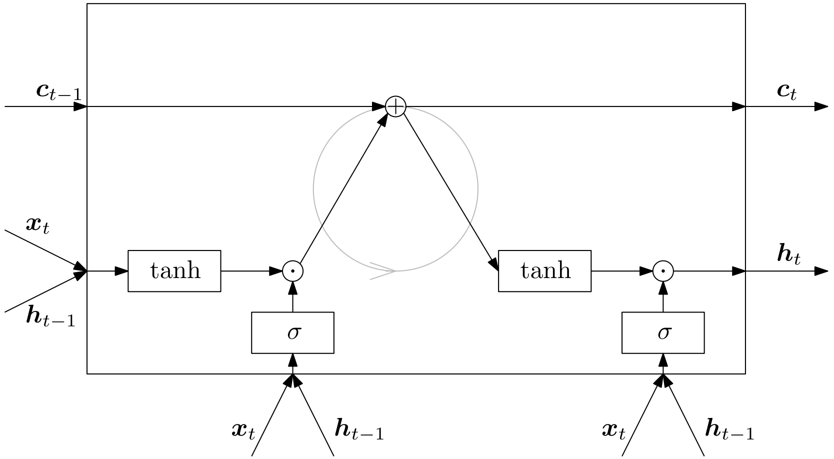
$$f' = \mathbf{1}.$$

They propose to achieve that by a *constant error carrousel*.



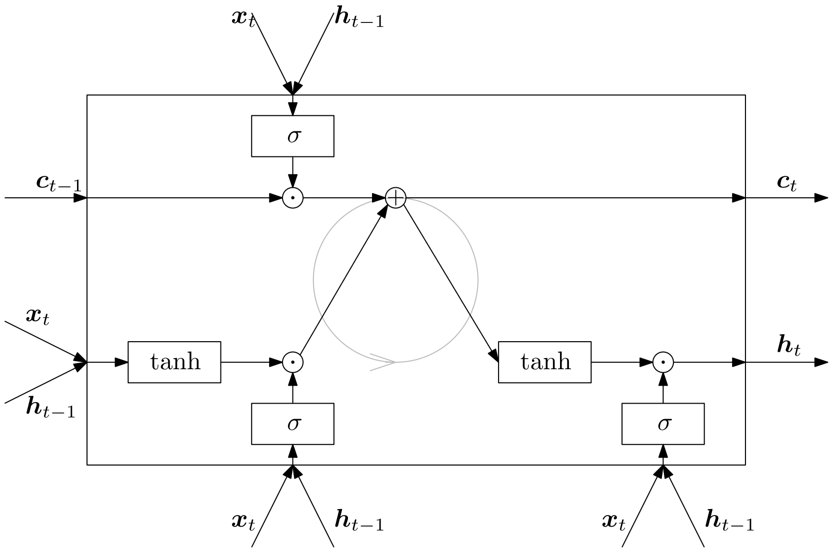
They also propose an **input** and **output** gates which control the flow of information into and out of the carrousel (**memory cell** $\boldsymbol{c}_t$).

$$\boldsymbol{i}_t \leftarrow \sigma(\boldsymbol{W}^i\boldsymbol{x}_t + \boldsymbol{V}^i\boldsymbol{h}_{t-1} + \boldsymbol{b}^i)$$
$$\boldsymbol{o}_t \leftarrow \sigma(\boldsymbol{W}^o\boldsymbol{x}_t + \boldsymbol{V}^o\boldsymbol{h}_{t-1} + \boldsymbol{b}^o)$$
$$\boldsymbol{c}_t \leftarrow \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}^y\boldsymbol{x}_t + \boldsymbol{V}^y\boldsymbol{h}_{t-1} + \boldsymbol{b}^y)$$
$$\boldsymbol{h}_t \leftarrow \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)$$



Later in Gers, Schmidhuber & Cummins (1999) a possibility to **forget** information from memory cell $\boldsymbol{c}_t$ was added.

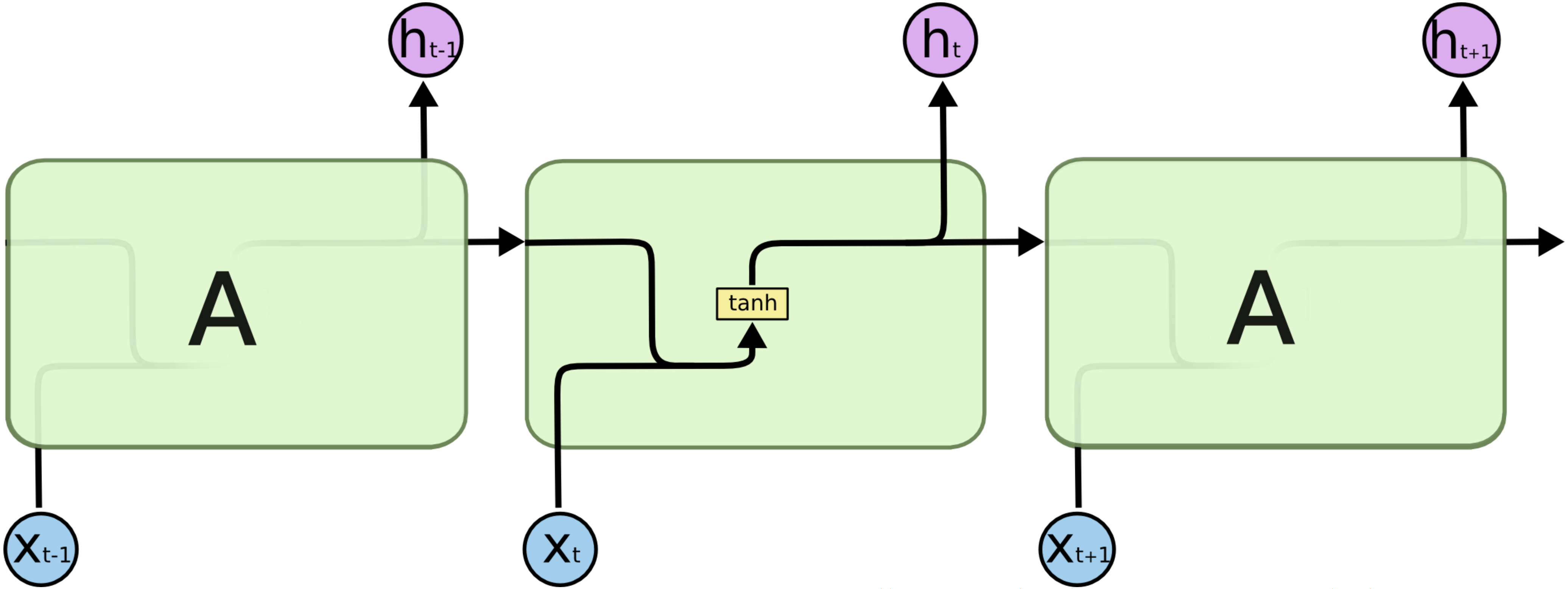
$$\boldsymbol{i}_t \leftarrow \sigma(\boldsymbol{W}^i \boldsymbol{x}_t + \boldsymbol{V}^i \boldsymbol{h}_{t-1} + \boldsymbol{b}^i)$$
$$\boldsymbol{f}_t \leftarrow \sigma(\boldsymbol{W}^f \boldsymbol{x}_t + \boldsymbol{V}^f \boldsymbol{h}_{t-1} + \boldsymbol{b}^f)$$
$$\boldsymbol{o}_t \leftarrow \sigma(\boldsymbol{W}^o \boldsymbol{x}_t + \boldsymbol{V}^o \boldsymbol{h}_{t-1} + \boldsymbol{b}^o)$$
$$\boldsymbol{c}_t \leftarrow \boldsymbol{f}_t \cdot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}^y \boldsymbol{x}_t + \boldsymbol{V}^y \boldsymbol{h}_{t-1} + \boldsymbol{b}^y)$$
$$\boldsymbol{h}_t \leftarrow \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)$$



Note that since 2015, following the paper

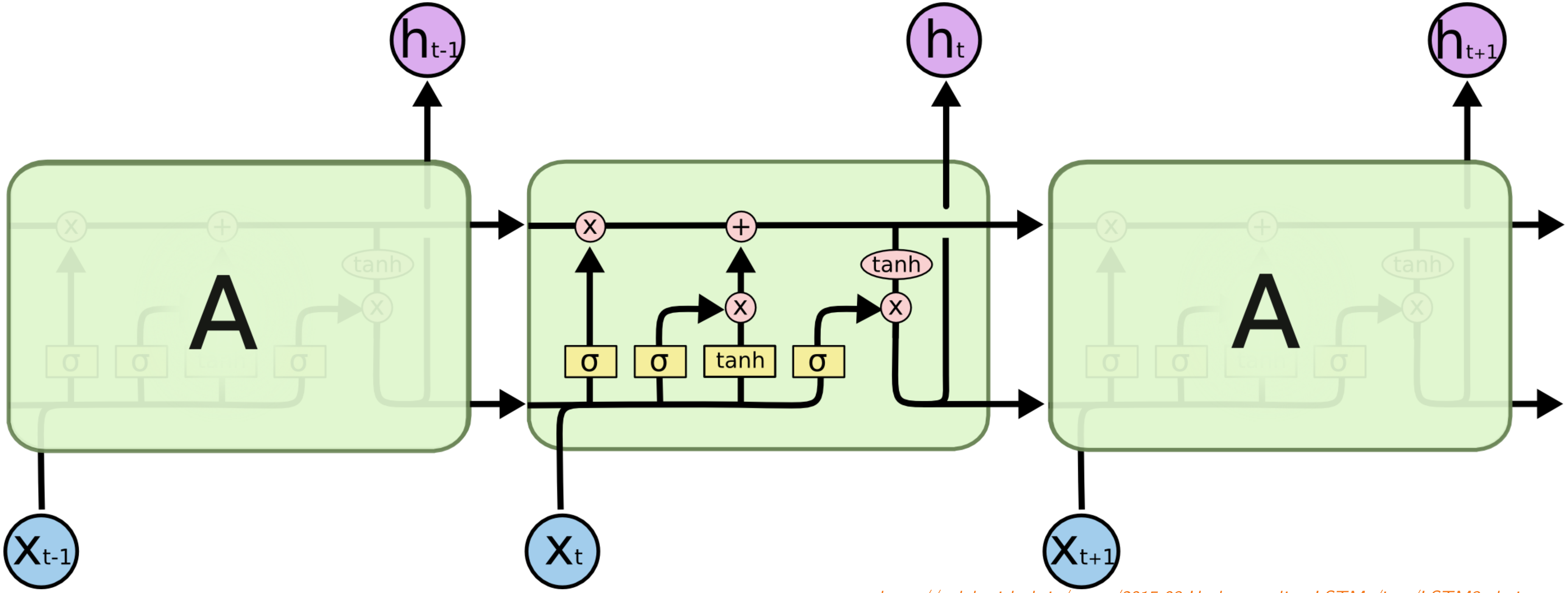
- R. Jozefowicz et al.: *An Empirical Exploration of Recurrent Network Architectures*

the forget gate bias is usually initialized to 1.

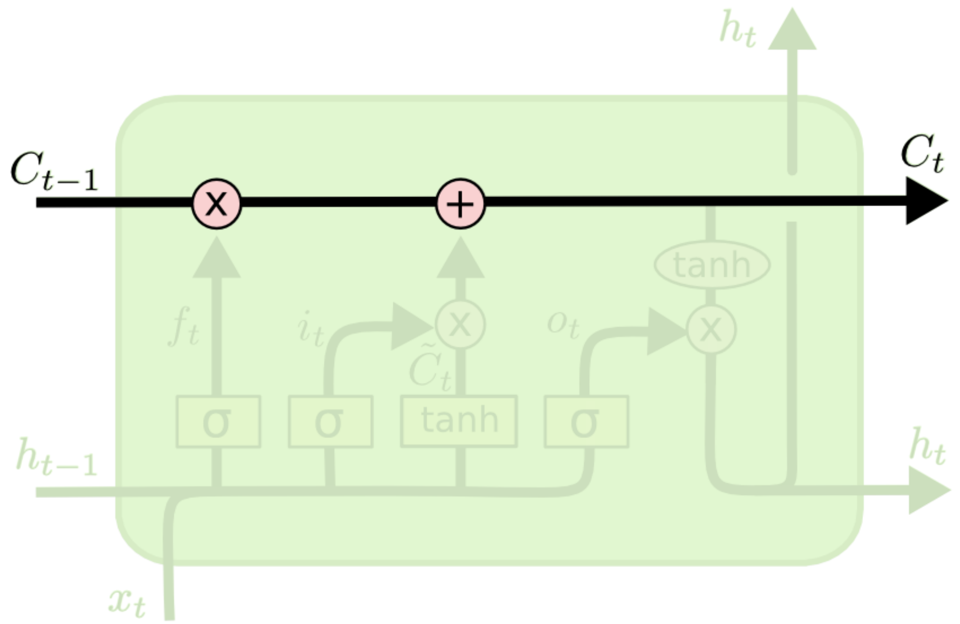


http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-SimpleRNN.png

Long Short-Term Memory

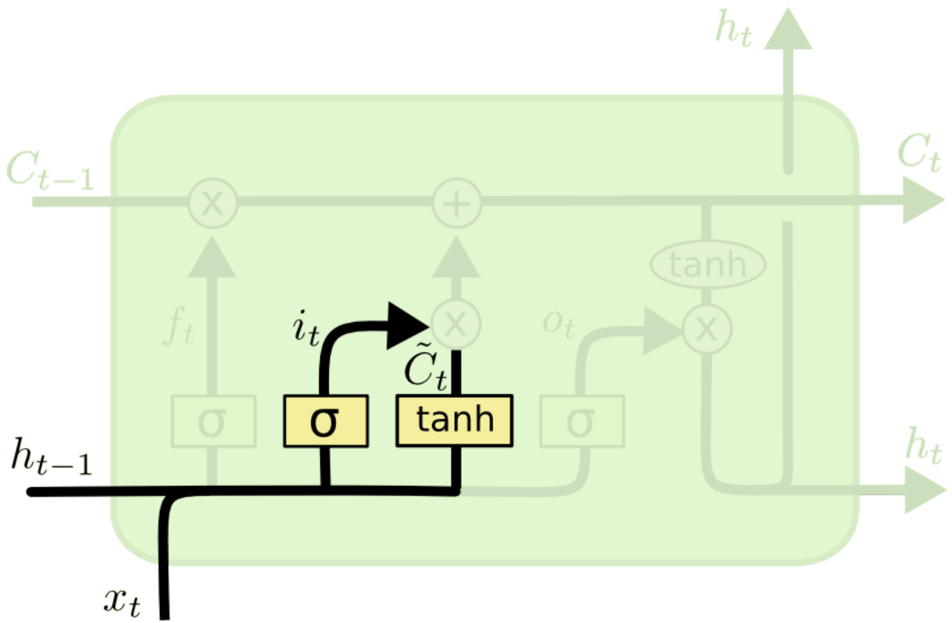


http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png



http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-C-line.png

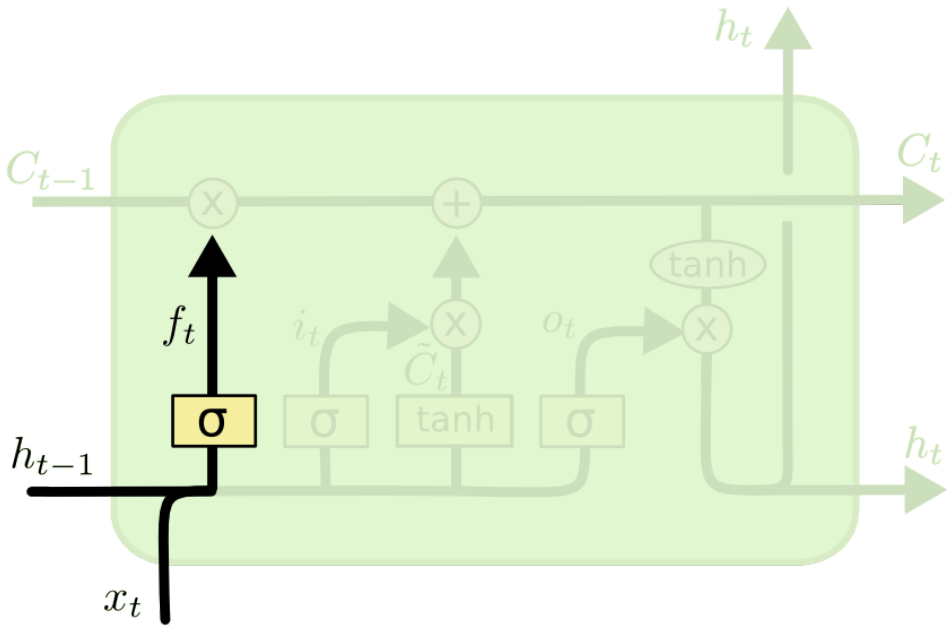
Long Short-Term Memory



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

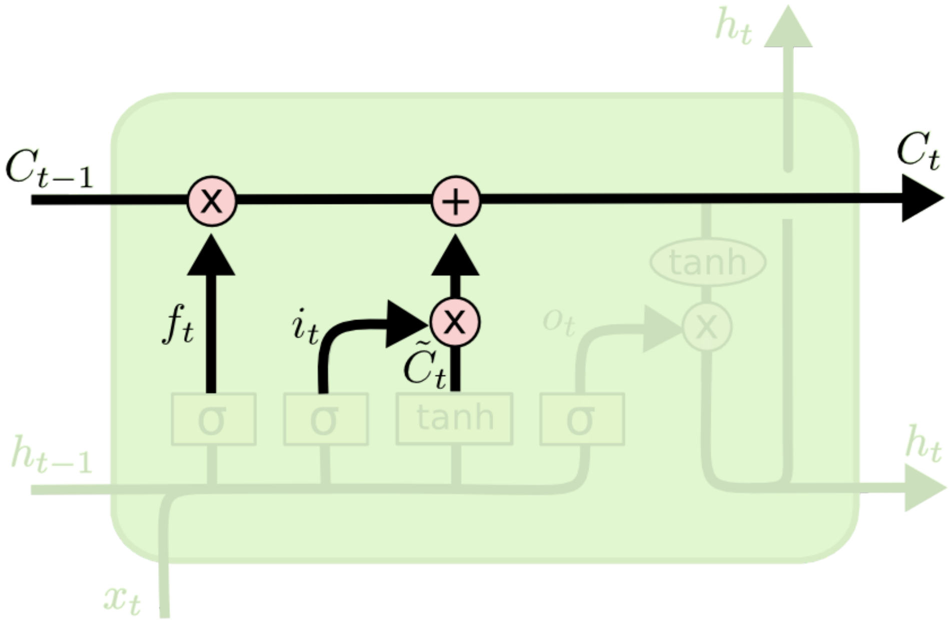
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \;+\; b_C)$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-i.png



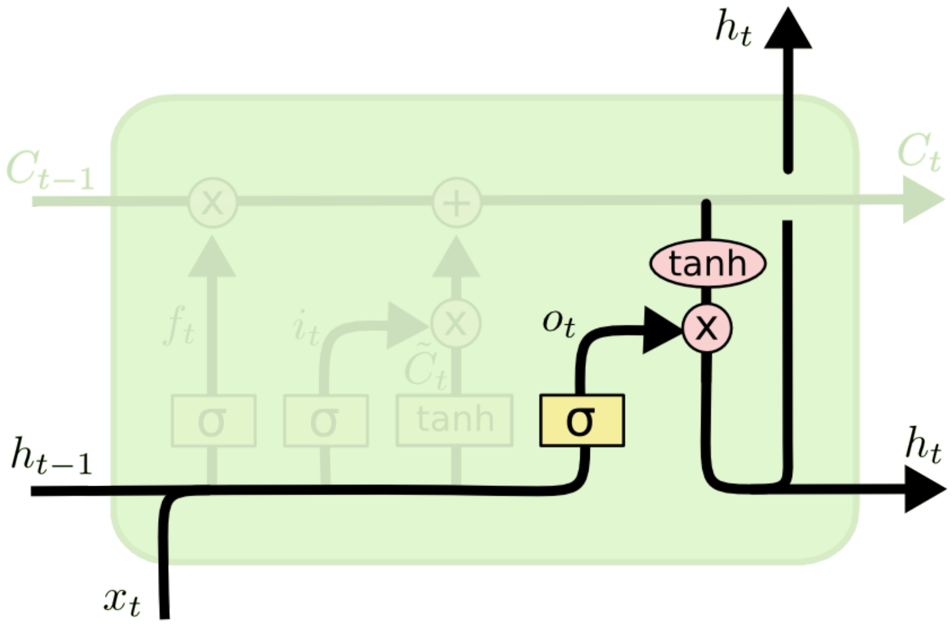
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-f.png



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-C.png



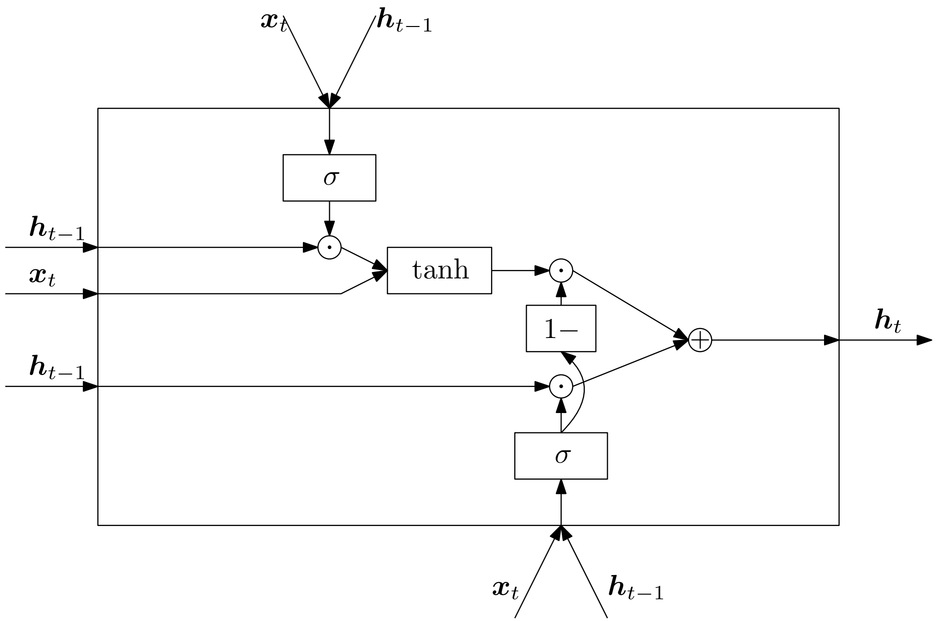
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-focus-o.png

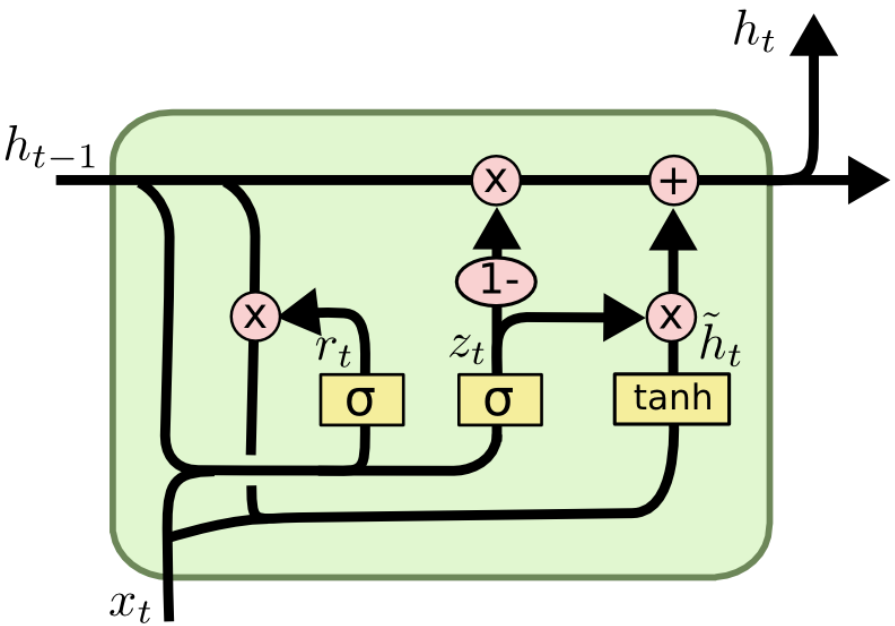
Gated recurrent unit (GRU) was proposed by Cho et al. (2014) as a simplification of LSTM. The main differences are

- no memory cell,
- forgetting and updating tied together.

$$\boldsymbol{r}_t \leftarrow \sigma(\boldsymbol{W}^r \boldsymbol{x}_t + \boldsymbol{V}^r \boldsymbol{h}_{t-1} + \boldsymbol{b}^r)$$
$$\boldsymbol{u}_t \leftarrow \sigma(\boldsymbol{W}^u \boldsymbol{x}_t + \boldsymbol{V}^u \boldsymbol{h}_{t-1} + \boldsymbol{b}^u)$$
$$\hat{\boldsymbol{h}}_t \leftarrow \tanh(\boldsymbol{W}^h \boldsymbol{x}_t + \boldsymbol{V}^h (\boldsymbol{r}_t \cdot \boldsymbol{h}_{t-1}) + \boldsymbol{b}^h)$$
$$\boldsymbol{h}_t \leftarrow \boldsymbol{u}_t \cdot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{u}_t) \cdot \hat{\boldsymbol{h}}_t$$



Gated Recurrent Unit



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

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

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

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

http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-var-GRU.png

GRU and LSTM Differences

The main differences between GRU and LSTM:

- GRU uses fewer parameters and less computation.
  - six matrices $\boldsymbol{W}$, $\boldsymbol{V}$ instead of eight
- GRU are easier to work with, because the state is just one tensor, while it is a pair of tensors for LSTM.
- In most tasks, LSTM and GRU give very similar results.
- However, there are some tasks, on which LSTM achieves (much) better results than GRU.
  - For a demonstration of difference in the expressive power of LSTM and GRU (caused by the coupling of the forget and update gate), see the paper
    - G. Weiss et al.: *On the Practical Computational Power of Finite Precision RNNs for Language Recognition* https://arxiv.org/abs/1805.04908
  - For a practical difference between LSTM and GRU, see for example
    - T. Dozat et al.: *Deep Biaffine Attention for Neural Dependency Parsing* https://arxiv.org/abs/1611.01734

Recall that when we approximate $\boldsymbol{h}^{(t)} \approx \boldsymbol{U}\boldsymbol{h}^{(t-1)}$, assuming the eigenvalue decomposition of $\boldsymbol{U} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$ we get

$$\boldsymbol{h}^{(t)} \approx \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\boldsymbol{h}^{(0)}.$$

This motivated a specific initialization scheme for the $\boldsymbol{U}$ matrix – this so-called **recurrent kernel** is initialized with a randomly generated orthogonal matrix.

This **orthogonal** initialization is used for all RNN cells in TensorFlow (via the `recurrent_initializer='orthogonal'` parameter of `SimpleRNN`, `GRU`, and `LSTM`).

Highway Networks

Highway Networks

For input $\boldsymbol{x}$, fully connected layer computes

$$\boldsymbol{y} \leftarrow H(\boldsymbol{x}, \boldsymbol{W}_H).$$

Highway networks add residual connection with gating:

$$\boldsymbol{y} \leftarrow H(\boldsymbol{x}, \boldsymbol{W}_H) \cdot T(\boldsymbol{x}, \boldsymbol{W}_T) + \boldsymbol{x} \cdot (1 - T(\boldsymbol{x}, \boldsymbol{W}_T)).$$

Usually, the gating is defined as

$$T(\boldsymbol{x}, \boldsymbol{W}_T) \leftarrow \sigma(\boldsymbol{W}_T \boldsymbol{x} + \boldsymbol{b}_T).$$

Note that the resulting update is very similar to a GRU cell with $\boldsymbol{h}_t$ removed; for a fully connected layer $H(\boldsymbol{x}, \boldsymbol{W}_H) = \tanh(\boldsymbol{W}_H \boldsymbol{x} + \boldsymbol{b}_H)$ it is exactly it, apart from copying $\boldsymbol{x}$ instead of $\boldsymbol{h}_{t-1}$.

Highway Networks on MNIST

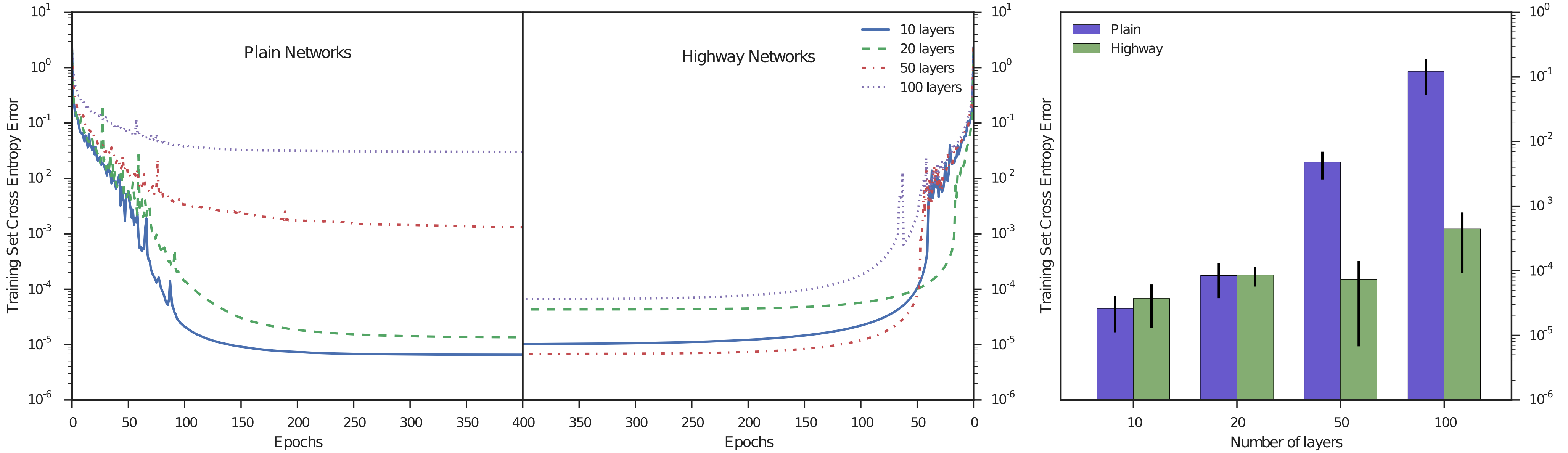


Figure 1: Comparison of optimization of plain networks and highway networks of various depths. *Left:* The training curves for the best hyperparameter settings obtained for each network depth. *Right:* Mean performance of top 10 (out of 100) hyperparameter settings. Plain networks become much harder to optimize with increasing depth, while highway networks with up to 100 layers can still be optimized well. Best viewed on screen (larger version included in Supplementary Material).

Figure 1 of "Training Very Deep Networks", https://arxiv.org/abs/1507.06228

Highway Networks

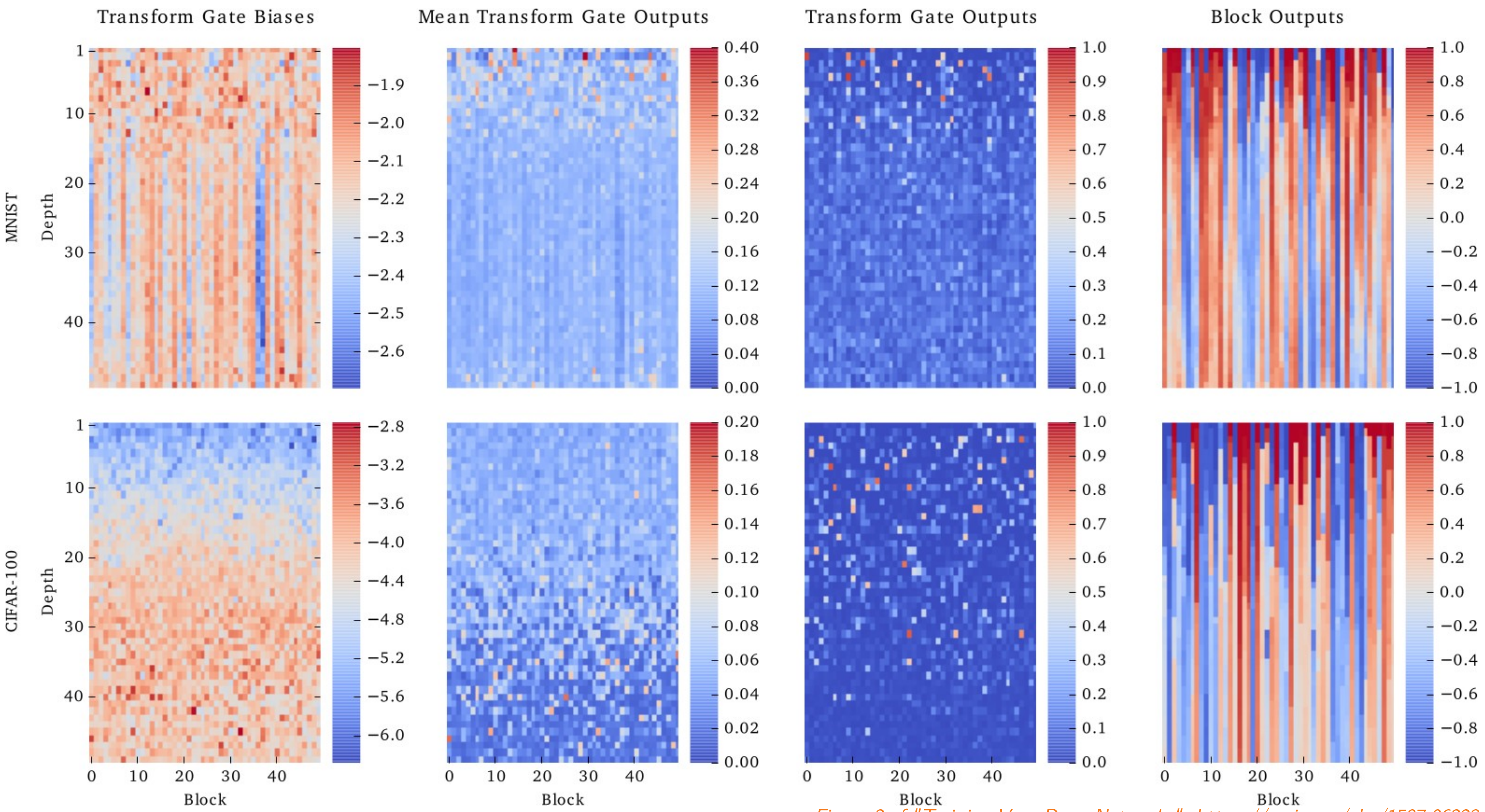


Figure 2 of "Training Very Deep Networks", https://arxiv.org/abs/1507.06228

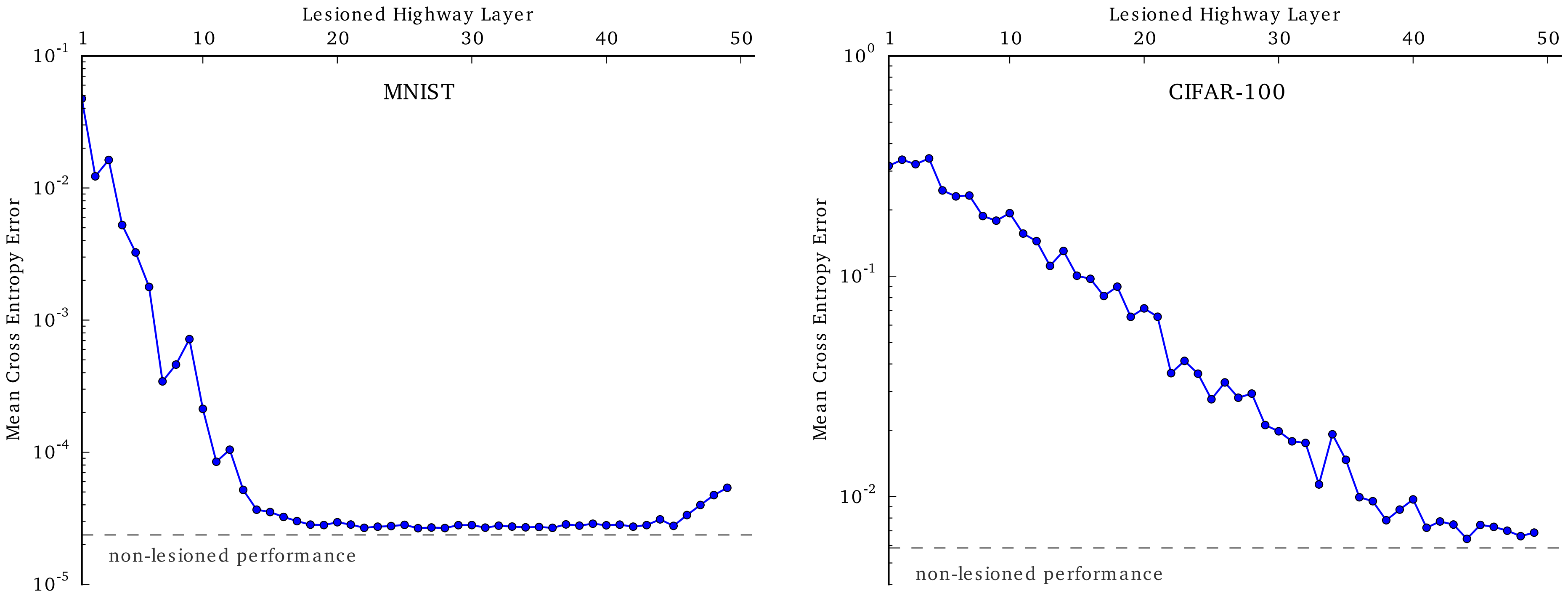


Figure 4: Lesioned training set performance (y-axis) of the best 50-layer highway networks on MNIST (left) and CIFAR-100 (right), as a function of the lesioned layer (x-axis). Evaluated on the full training set while forcefully closing all the transform gates of a single layer at a time. The non-lesioned performance is indicated as a dashed line at the bottom.

Figure 4 of "Training Very Deep Networks", https://arxiv.org/abs/1507.06228

Dropout

- Using dropout on hidden states interferes with long-term dependencies.
- However, using dropout on the inputs and outputs works well and is used frequently.
  - In case residual connections are present, the output dropout needs to be applied before adding the residual connection.
- Several techniques were designed to allow using dropout on hidden states.
  - Variational Dropout
  - Recurrent Dropout
  - Zoneout

Variational Dropout

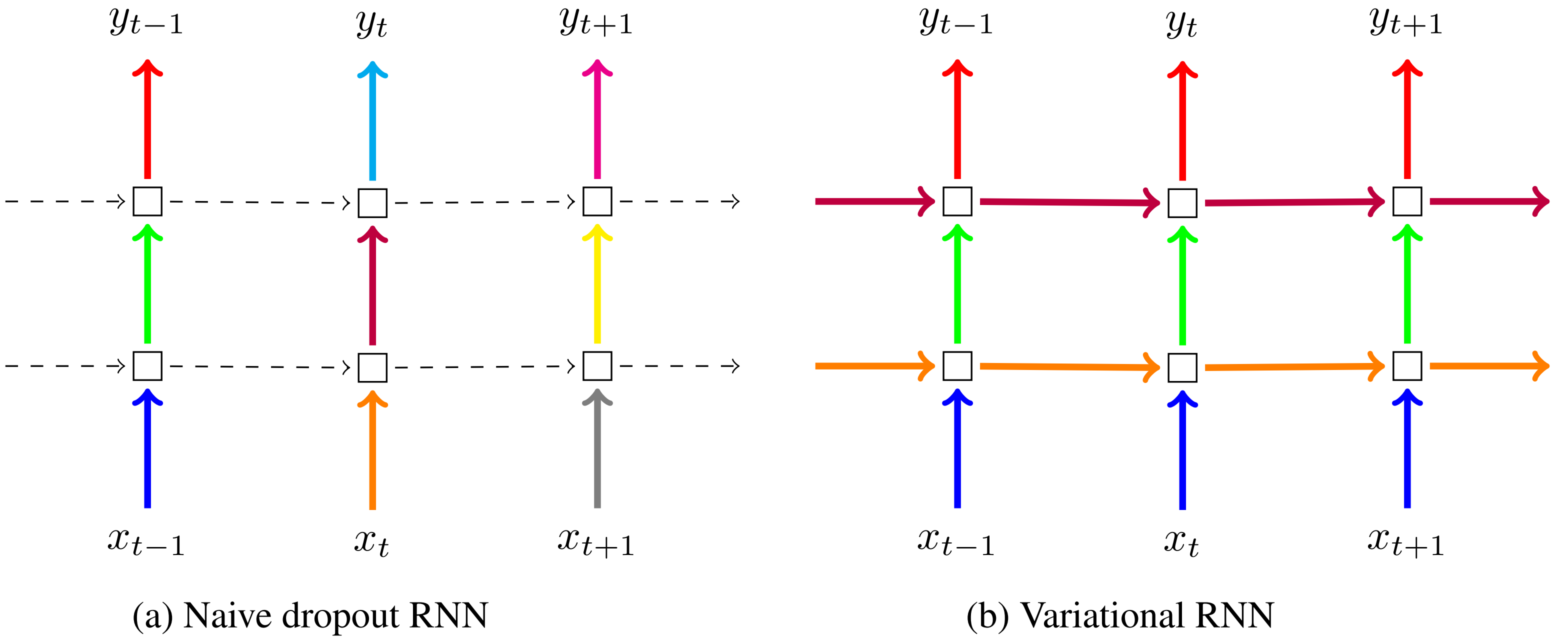


Figure 1 of "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks", https://arxiv.org/abs/1512.05287.pdf

To implement variational dropout on inputs in TensorFlow, use `noise_shape` of `tf.keras.layers.Dropout` to force the same mask across time-steps. The variational dropout on the hidden states can be implemented using `recurrent_dropout` argument of `tf.keras.layers.{LSTM,GRU,SimpleRNN}{,Cell}`.

Regularizing RNNs

Recurrent Dropout

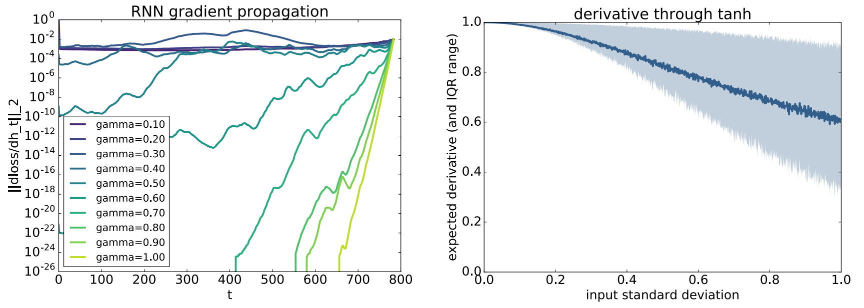
Dropout only candidate states (i.e., values added to the memory cell in LSTM and previous state in GRU), independently in every time-step.

Zoneout

Randomly preserve hidden activations instead of dropping them.

Batch Normalization

Very fragile and sensitive to proper initialization – there were papers with negative results (*Dario Amodei et al, 2015: Deep Speech 2* or *Cesar Laurent et al, 2016: Batch Normalized Recurrent Neural Networks*) until people managed to make it work (*Tim Cooijmans et al, 2016: Recurrent Batch Normalization*; specifically, initializing $\gamma = 0.1$ did the trick).



(a) We visualize the gradient flow through a batch-normalized tanh RNN as a function of γ. High variance causes vanishing gradient.

(b) We show the empirical expected derivative and interquartile range of tanh nonlinearity as a function of input variance. High variance causes saturation, which decreases the expected derivative.

Figure 1 of "Recurrent Batch Normalization", https://arxiv.org/abs/1603.09025

Batch Normalization

Neuron value is normalized across the minibatch, and in case of CNN also across all positions.

Layer Normalization

Neuron value is normalized across the layer.

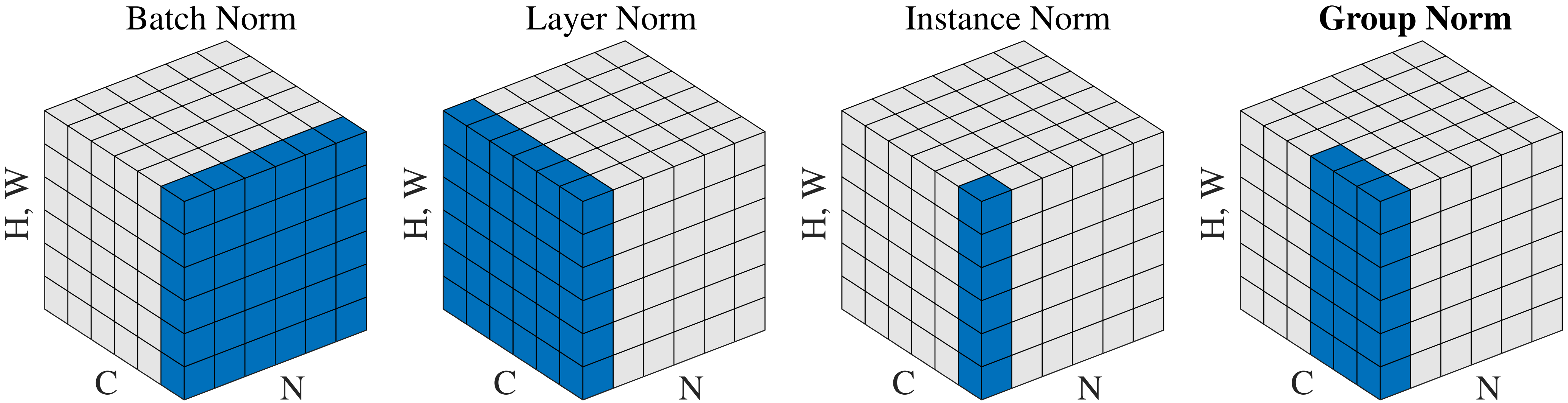


Figure 2 of "Group Normalization", https://arxiv.org/abs/1803.08494

Layer Normalization

Consider a hidden value $\boldsymbol{x} \in \mathbb{R}^D$. Layer normalization (both during training and during inference) is performed as follows.

Inputs: An example $\boldsymbol{x} \in \mathbb{R}^D$, $\varepsilon \in \mathbb{R}$ with default value 0.001
Parameters: $\boldsymbol{\beta} \in \mathbb{R}^D$ initialized to $\mathbf{0}$, $\boldsymbol{\gamma} \in \mathbb{R}^D$ initialized to $\mathbf{1}$
Outputs: Normalized example $\boldsymbol{y}$

- $\mu \leftarrow \frac{1}{D} \sum_{i=1}^D x_i$
- $\sigma^2 \leftarrow \frac{1}{D} \sum_{i=1}^D (x_i - \mu)^2$
- $\hat{\boldsymbol{x}} \leftarrow (\boldsymbol{x} - \mu)/\sqrt{\sigma^2 + \varepsilon}$
- $\boldsymbol{y} \leftarrow \boldsymbol{\gamma}\hat{\boldsymbol{x}} + \boldsymbol{\beta}$

Layer Normalization

Much more stable than batch normalization for RNN regularization.

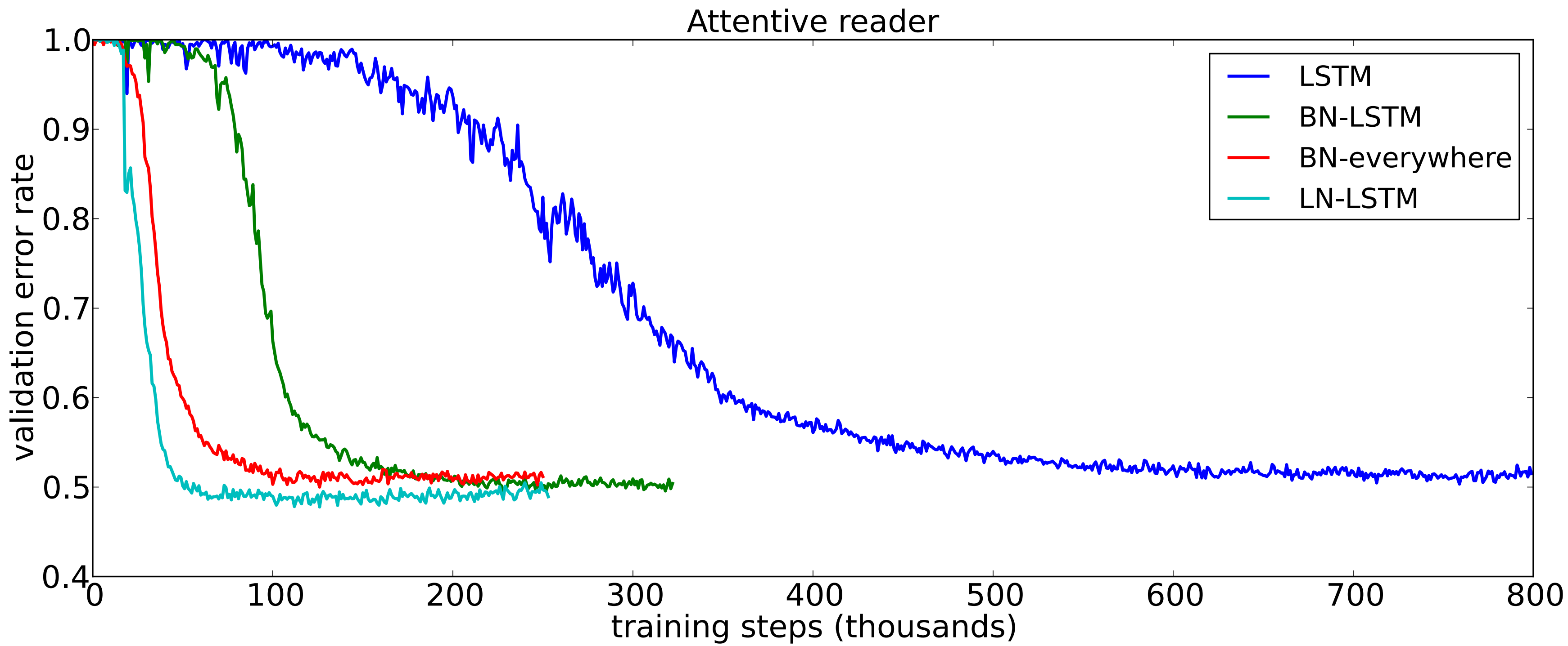
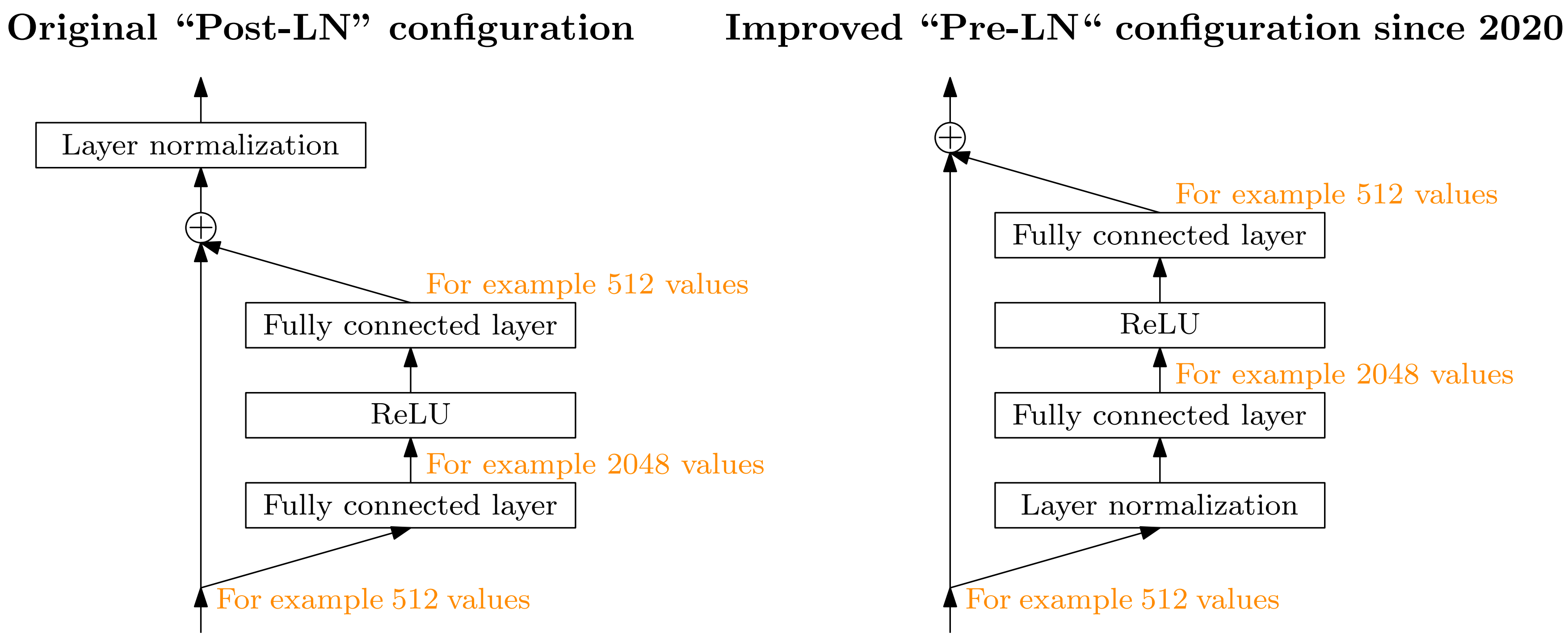


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

Figure 2 of "Layer Normalization", https://arxiv.org/abs/1607.06450

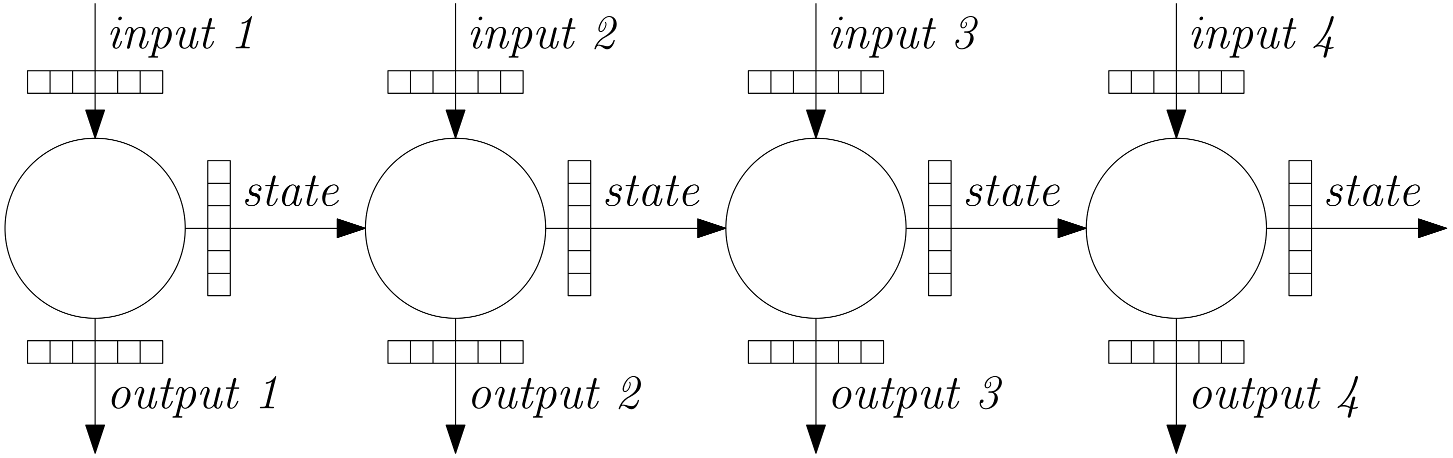
In an important recent architecture (namely Transformer), many fully connected layers are used, with a residual connection and a layer normalization.



This could be considered an alternative to highway networks, i.e., a suitable residual connection for fully connected layers. Note the architecture can be considered as a variant of a mobile inverted bottleneck 1×1 convolution block.

Sequence Element Representation

Create output for individual elements, for example for classification of the individual elements.

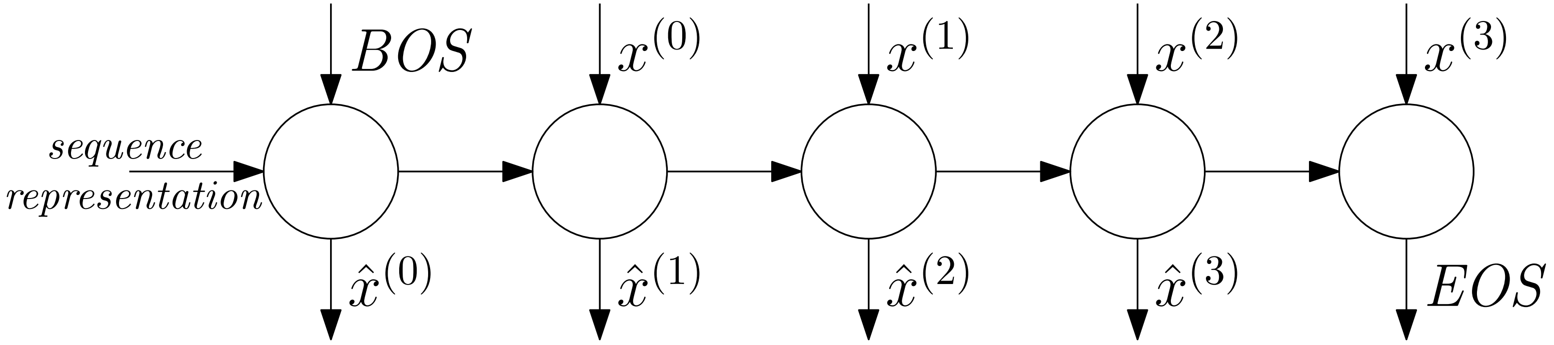


Sequence Representation

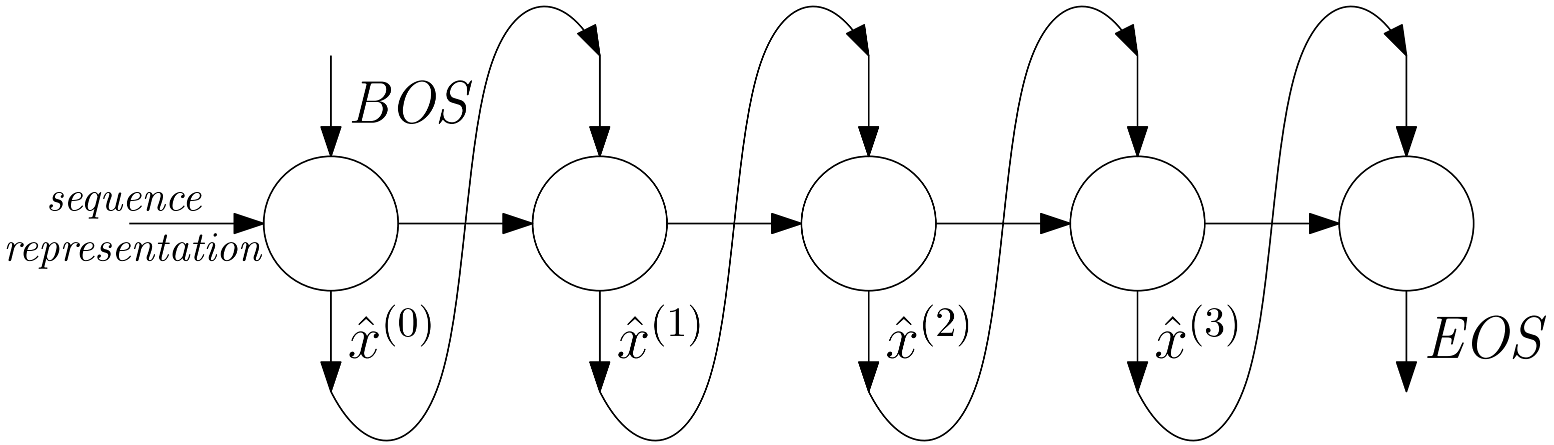
Generate a single output for the whole sequence (either the last output or the last state).

Sequence Prediction

During training, predict next sequence element.

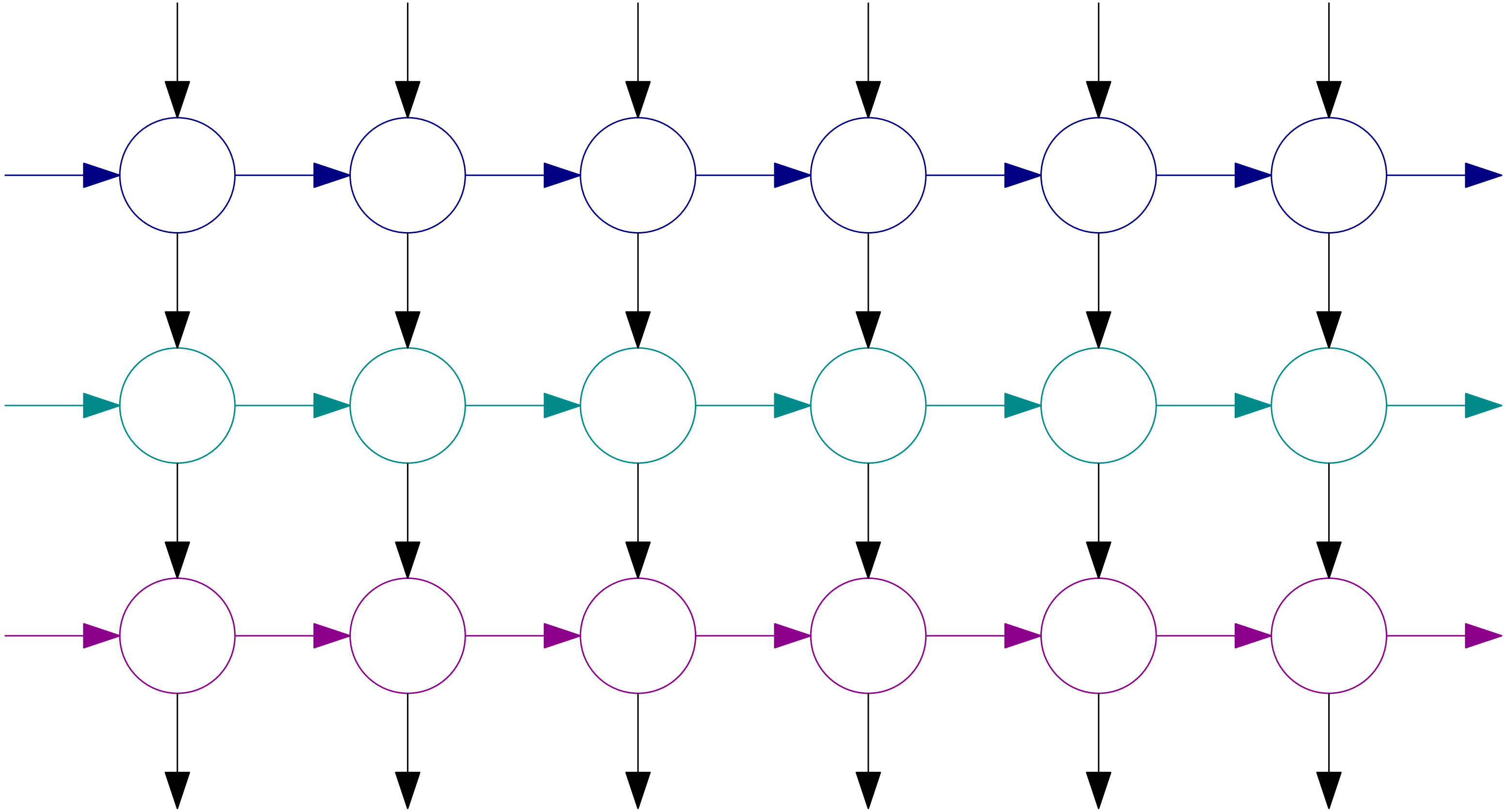


During inference, use predicted elements as further inputs.



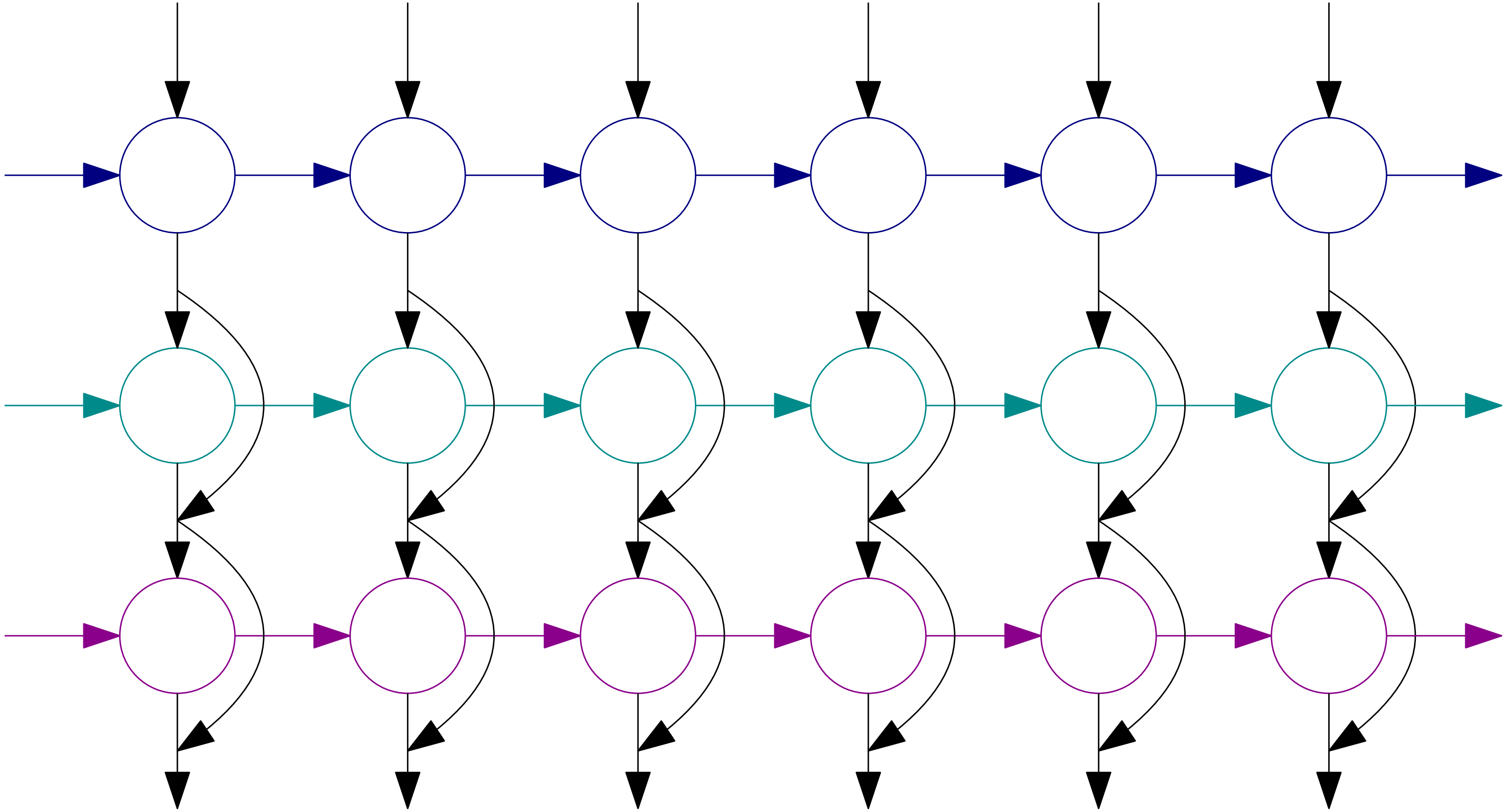
Multilayer RNNs

We might stack several layers of recurrent neural networks. Usually using two or three layers gives better results than just one.



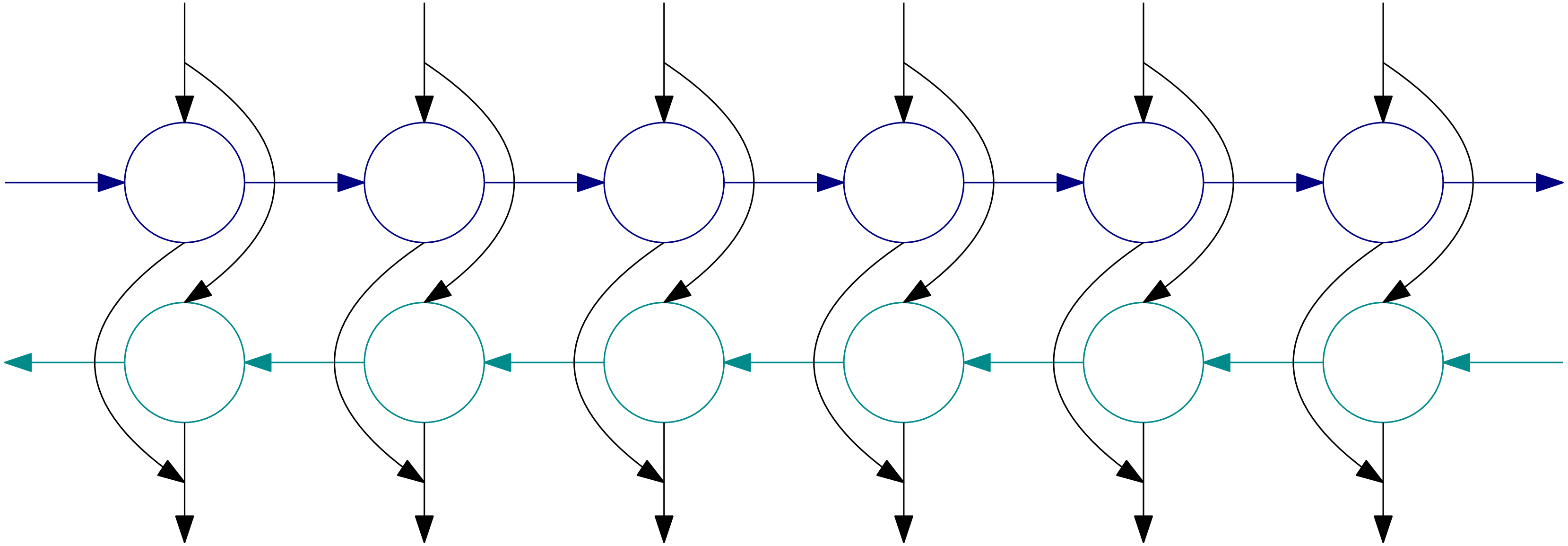
Multilayer RNNs

In case of multiple layers, residual connections usually improve results. Because dimensionality has to be the same, they are usually applied from the second layer.



Bidirectional RNN

To consider both the left and right contexts, a **bidirectional** RNN can be used, which consists of parallel application of a **forward** RNN and a **backward** RNN.



The outputs of both directions can be either **added** or **concatenated**. Even if adding them does not seem very intuitive, it does not increase dimensionality and therefore allows residual connections to be used in case of multilayer bidirectional RNN.

Word Embeddings

We might represent **words** using one-hot encoding, considering all words to be independent of each other.

However, words are not independent – some are more similar than others.

Ideally, we would like some kind of similarity in the space of the word representations.

Distributed Representation

The idea behind distributed representation is that objects can be represented using a set of common underlying factors.

We therefore represent words as fixed-size **embeddings** into $\mathbb{R}^d$ space, with the vector elements playing role of the common underlying factors.

These embeddings are initialized randomly and trained together with the rest of the network.

Word Embeddings

The word embedding layer is in fact just a fully connected layer on top of one-hot encoding. However, it is not implemented in that way.

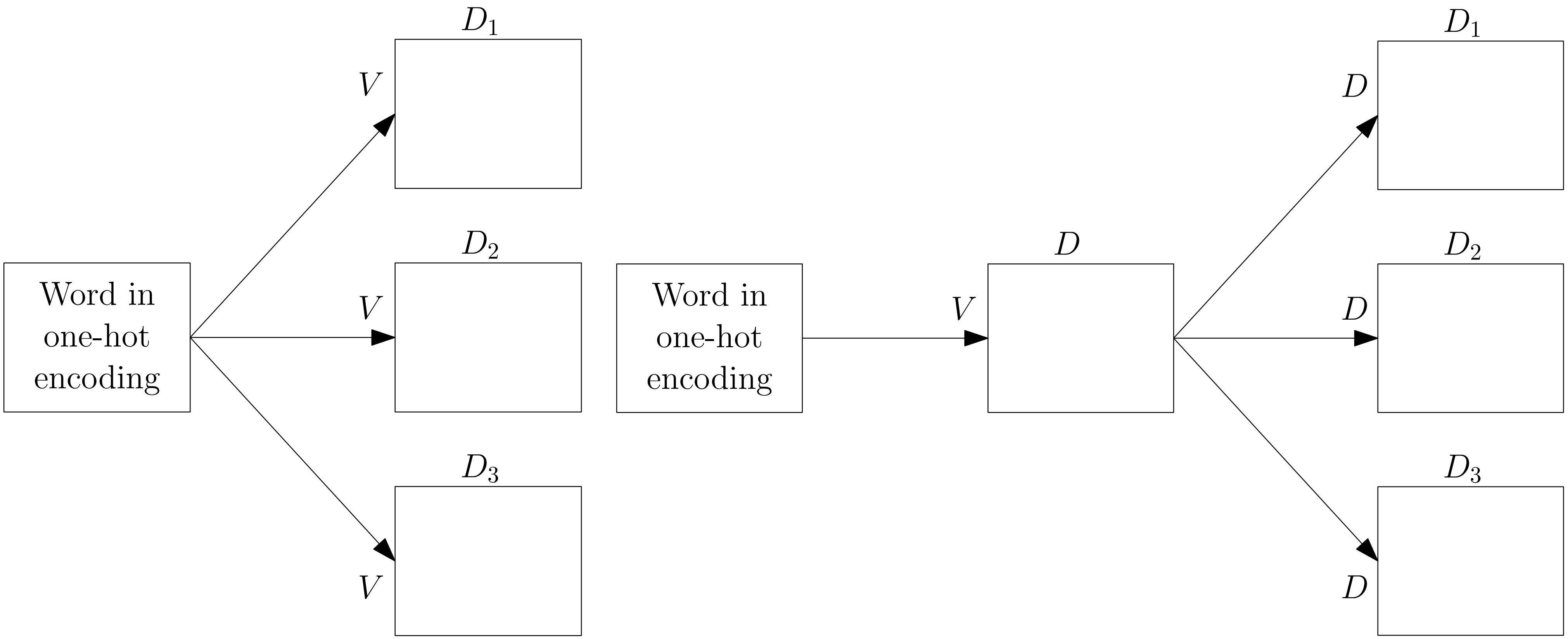
Instead, the so-called **embedding** layer is used, which is much more efficient. When a matrix is multiplied by an one-hot encoded vector (all but one zeros and exactly one 1), the row corresponding to that 1 is selected, so the embedding layer can be implemented only as a simple lookup.

In TensorFlow, the embedding layer is available as

```
tf.keras.layers.Embedding(input_dim, output_dim)
```

Word Embeddings

Even if the embedding layer is just a fully connected layer on top of one-hot encoding, it is important that this layer is *shared* across the whole network.



Recurrent Character-level WEs

In order to handle words not seen during training, we could find a way to generate a representation from the word **characters**.

A possible way to compose the representation from individual characters is to use RNNs – we embed *characters* to get character representation, and then use an RNN to produce the representation of a whole *sequence of characters*.

Usually, both forward and backward directions are used, and the resulting representations are concatenated/added.

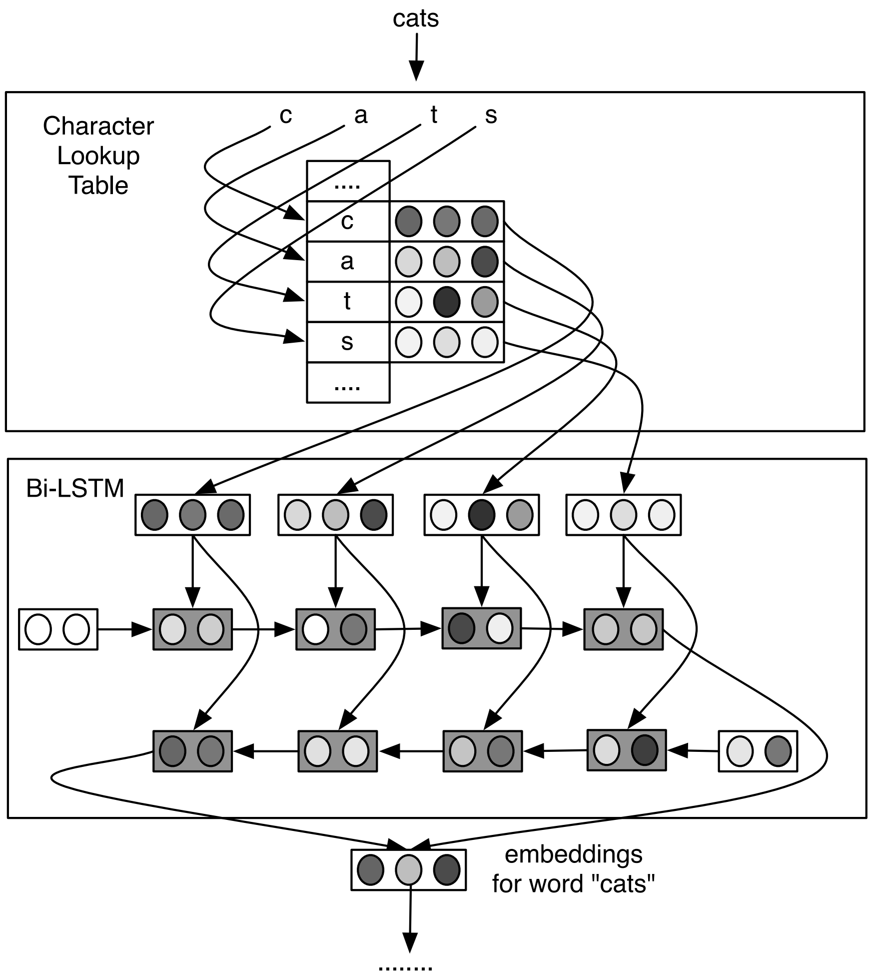


Figure 1 of "Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation", https://arxiv.org/abs/1508.02096

Convolutional Character-level WEs

Alternatively, 1D convolutions might be used.

Assume we use a 1D convolution with kernel size 3. It produces a representation for every input word trigram, but we need a representation of the whole word. To that end, we use *global max-pooling* – using it has an interpretable meaning, where the kernel is a *pattern* and the activation after the maximum is a level of a highest match of the pattern anywhere in the word.

Kernels of varying sizes are usually used (because it makes sense to have patterns for unigrams, bigrams, trigrams, …) – for example, 25 filters for every kernel size $(1, 2, 3, 4, 5)$ might be used.

Lastly, authors employed a highway layer after the convolutions, improving the results (compared to not using any layer or using a fully connected one).

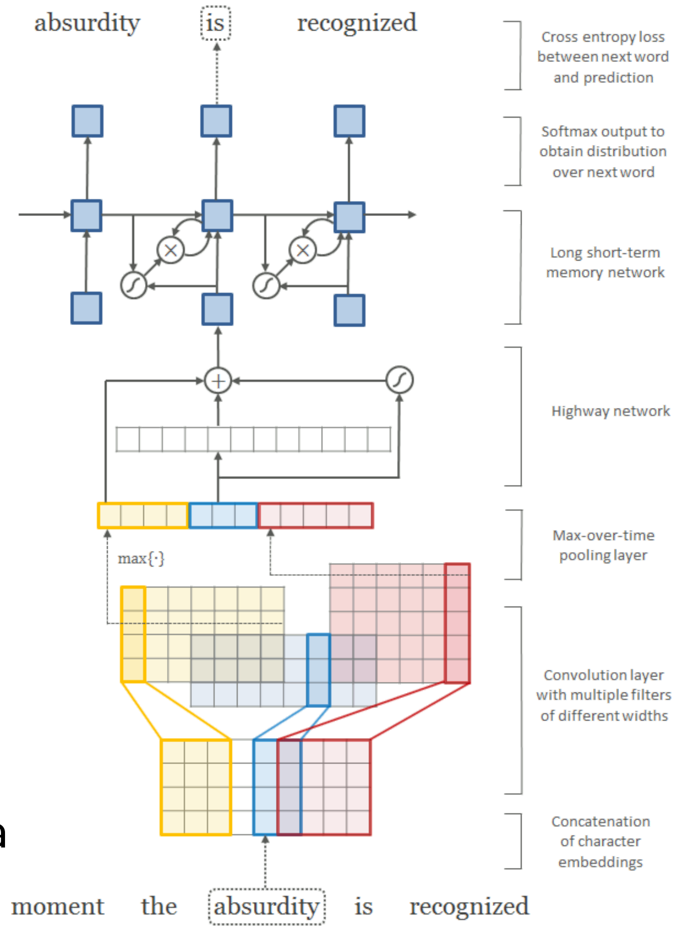


Figure 1 of "Character-Aware Neural Language Models", https://arxiv.org/abs/1508.06615

Examples of Recurrent Character-level WEs

increased	*John*	*Noahshire*	*phding*
reduced	Richard	Nottinghamshire	mixing
improved	George	Bucharest	modelling
expected	James	Saxony	styling
decreased	Robert	Johannesburg	blaming
targeted	Edward	Gloucestershire	christening

Table 2: Most-similar in-vocabular words under the C2W model; the two query words on the left are in the training vocabulary, those on the right are nonce (invented) words.

Table 2 of "Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation", https://arxiv.org/abs/1508.02096

Examples of Convolutional Character-level WEs

	In Vocabulary					Out-of-Vocabulary		
	while	*his*	*you*	*richard*	*trading*	*computer-aided*	*misinformed*	*looooook*
LSTM-Word	*although*	*your*	*conservatives*	*jonathan*	*advertised*	–	–	–
	letting	*her*	*we*	*robert*	*advertising*	–	–	–
	though	*my*	*guys*	*neil*	*turnover*	–	–	–
	minute	*their*	*i*	*nancy*	*turnover*	–	–	–
LSTM-Char (before highway)	*chile*	*this*	*your*	*hard*	*heading*	*computer-guided*	*informed*	*look*
	whole	*hhs*	*young*	*rich*	*training*	*computerized*	*performed*	*cook*
	meanwhile	*is*	*four*	*richer*	*reading*	*disk-drive*	*transformed*	*looks*
	white	*has*	*youth*	*richter*	*leading*	*computer*	*inform*	*shook*
LSTM-Char (after highway)	*meanwhile*	*hhs*	*we*	*eduard*	*trade*	*computer-guided*	*informed*	*look*
	whole	*this*	*your*	*gerard*	*training*	*computer-driven*	*performed*	*looks*
	though	*their*	*doug*	*edward*	*traded*	*computerized*	*outperformed*	*looked*
	nevertheless	*your*	*i*	*carl*	*trader*	*computer*	*transformed*	*looking*

Table 6: Nearest neighbor words (based on cosine similarity) of word representations from the large word-level and character-level (before and after highway layers) models trained on the PTB. Last three words are OOV words, and therefore they do not have representations in the word-level model.

Table 6 of "Character-Aware Neural Language Models", https://arxiv.org/abs/1508.06615

Character-level WE Implementation

Training

- Generate unique words per batch.
- Process the unique words in the batch.
- Copy the resulting embeddings suitably in the batch.

Inference

- We can cache character-level word embeddings during inference.

NLP Processing with CLEs

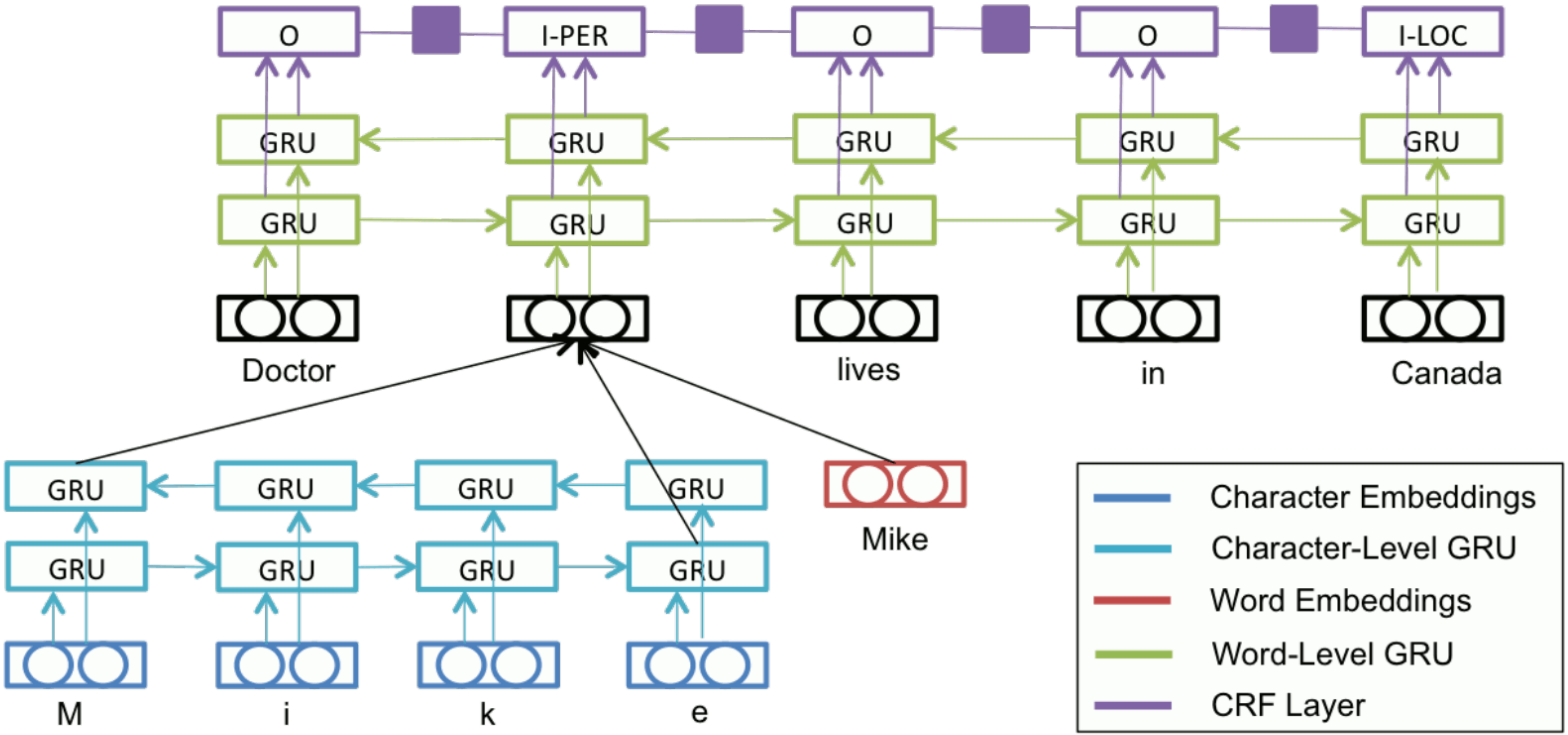


Figure 1 of "Multi-Task Cross-Lingual Sequence Tagging from Scratch", https://arxiv.org/abs/1603.06270