Recurrent Neural Networks

Milan Straka

April 1, 2025

Charles University

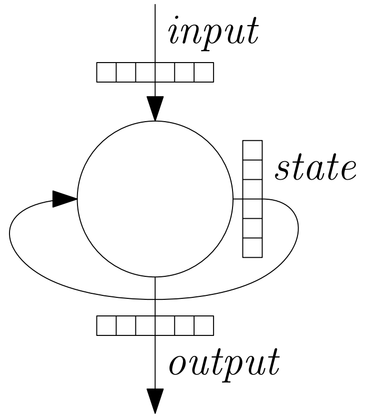
EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

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

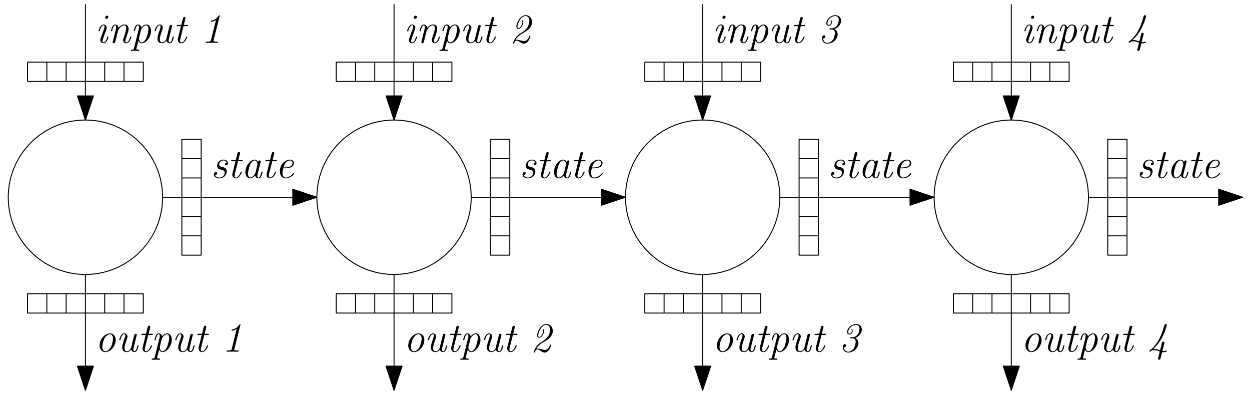
Recurrent Neural Networks

Recurrent Neural Networks

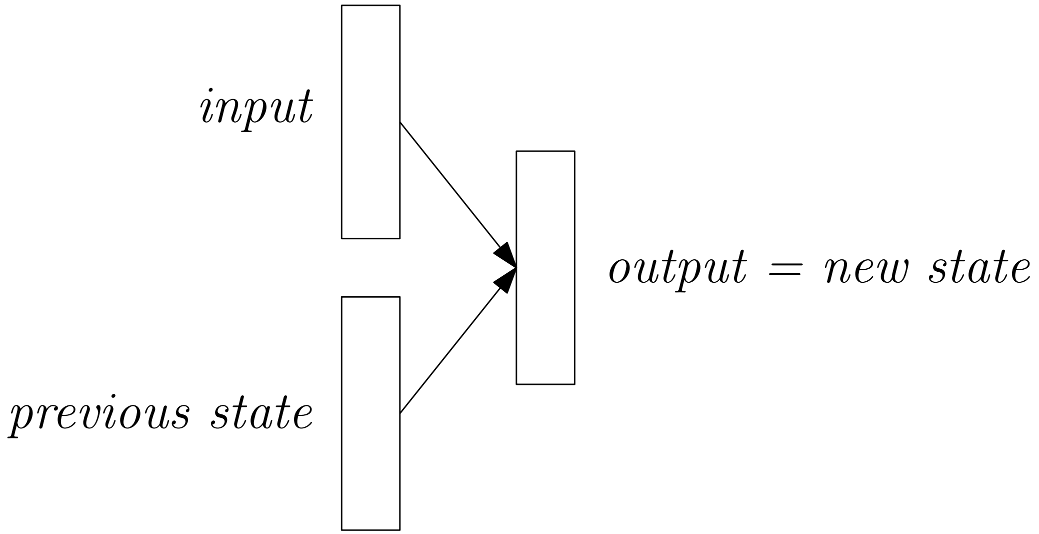
Single RNN cell

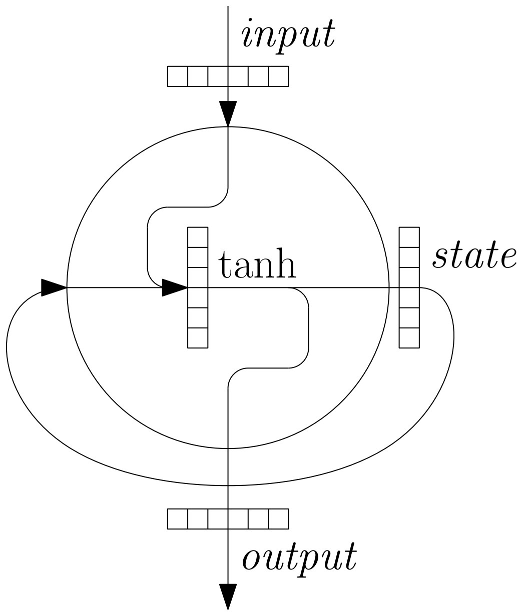


Unrolled RNN cells



Basic RNN Cell





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 `torch.nn.{RNN,RNNCell}` in PyTorch) 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 get that

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

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**.

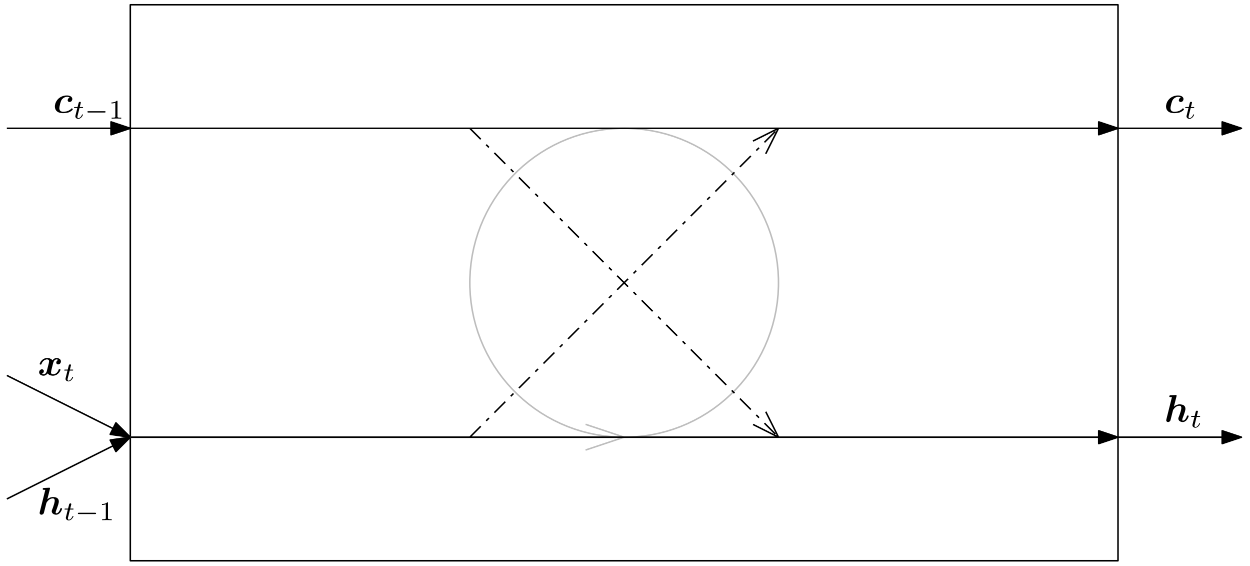
Long Short-Term Memory (LSTM)

Long Short-Term Memory

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

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

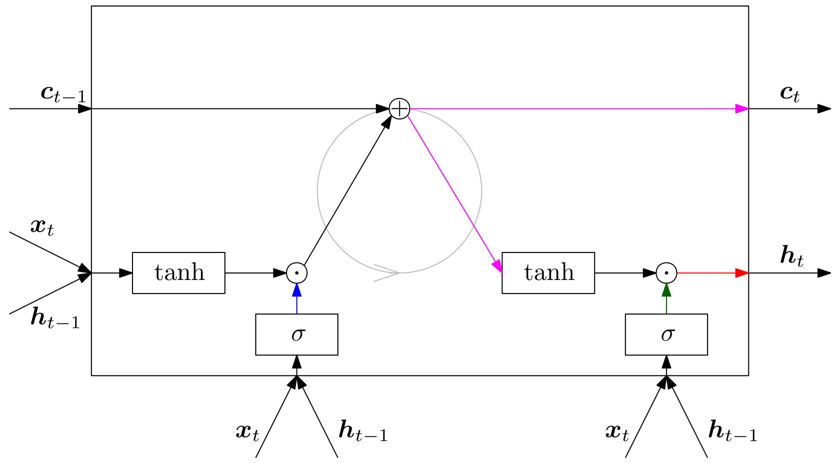
They propose to achieve that by a *constant error carrousel*.



Long Short-Term Memory

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

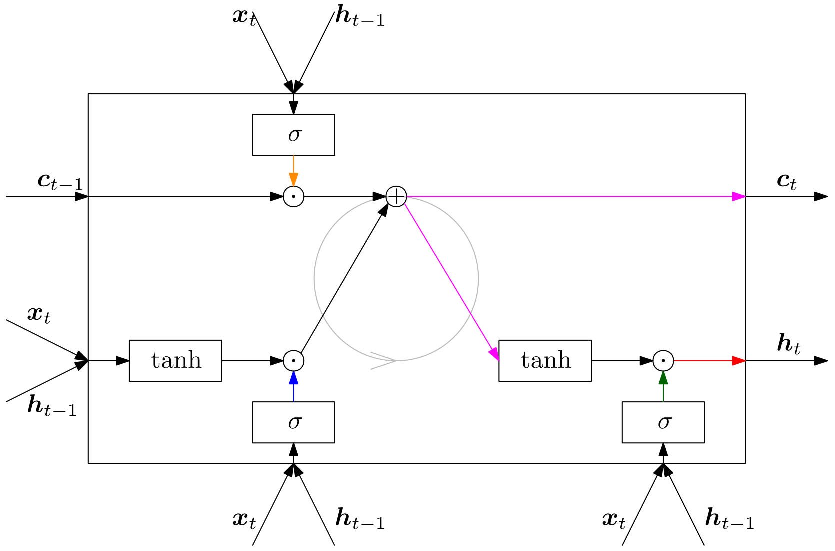
$$\boldsymbol{i}_t \leftarrow \sigma(\boldsymbol{W}^i \boldsymbol{x}_t + \boldsymbol{V}^i \boldsymbol{h}_{t-1} + \boldsymbol{b}^i)$$
$$\boldsymbol{o}_t \leftarrow \sigma(\boldsymbol{W}^o \boldsymbol{x}_t + \boldsymbol{V}^o \boldsymbol{h}_{t-1} + \boldsymbol{b}^o)$$
$$\boldsymbol{c}_t \leftarrow \boldsymbol{c}_{t-1}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



Long Short-Term Memory

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

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

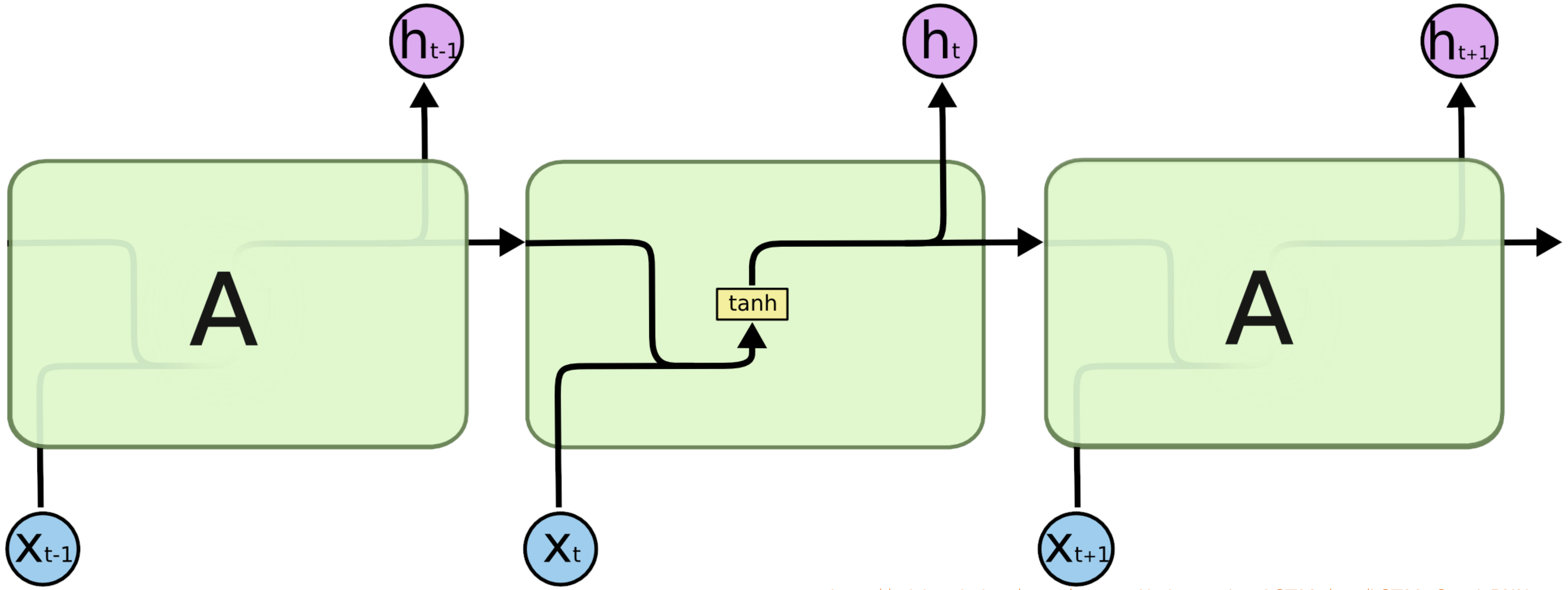


Note that since 2015, following the paper

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

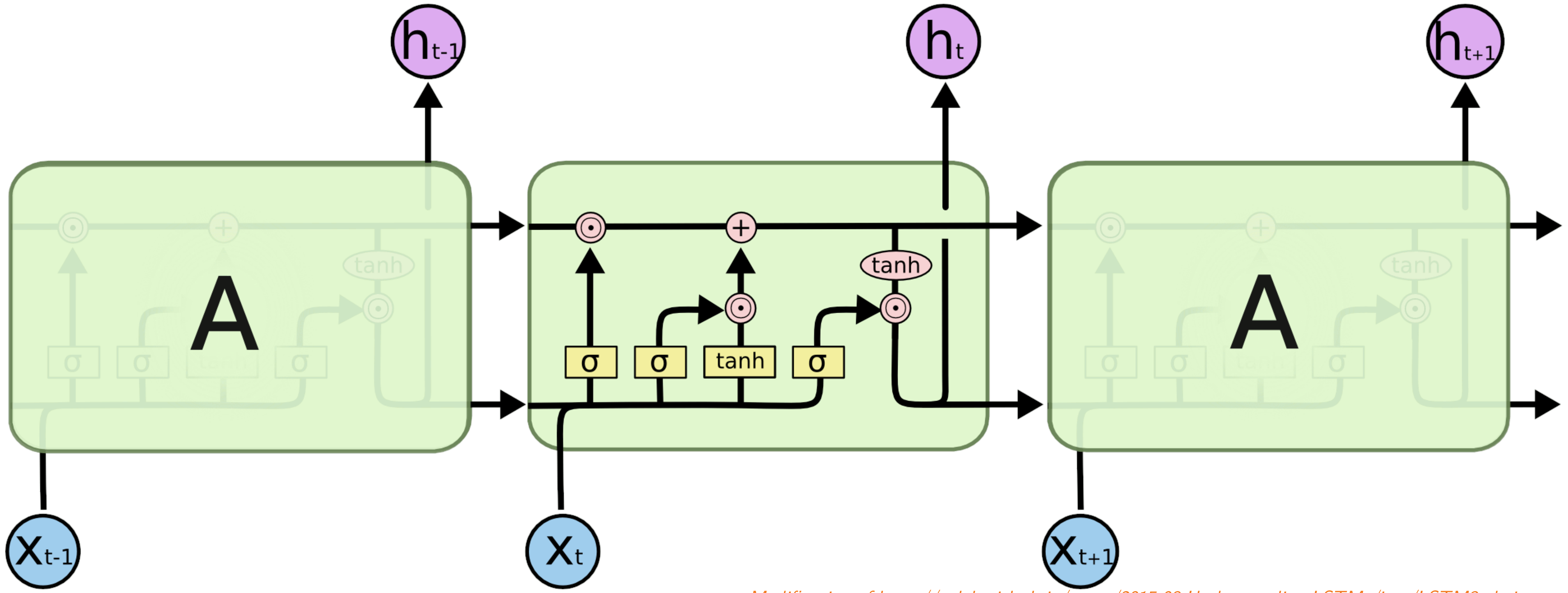
the forget gate bias $\boldsymbol{b}^f$ is usually initialized to 1, so that the forget gate is closer to 1 and the gradients can easily flow through multiple timesteps. (Gers et al. advocated this in the original paper already.) (BTW, I think 3 might be even better, as $\sigma(1) \approx 0.731$, $\sigma(3) \approx 0.953$.)

Long Short-Term Memory



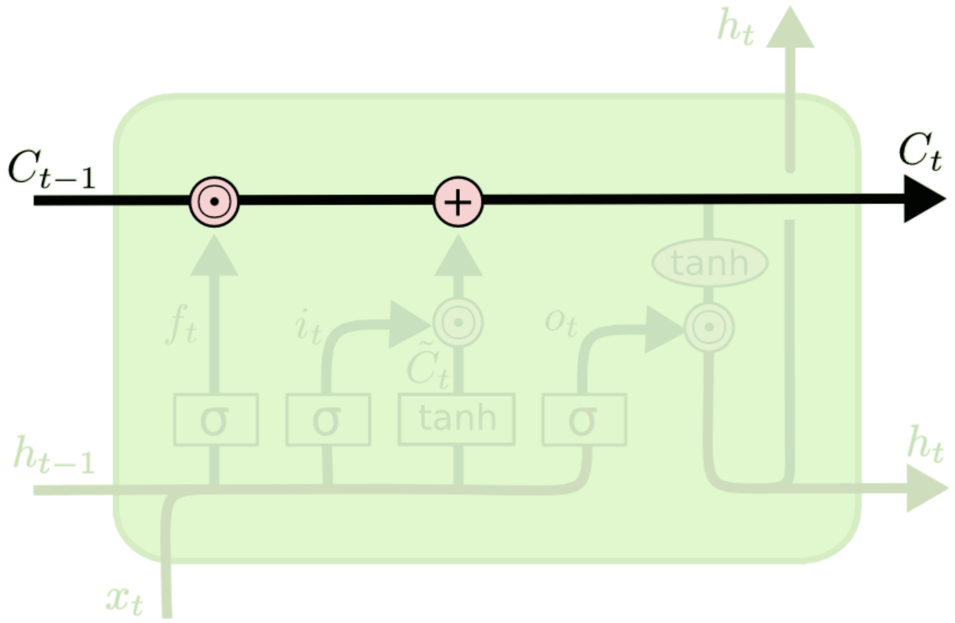
http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-SimpleRNN.png

Long Short-Term Memory



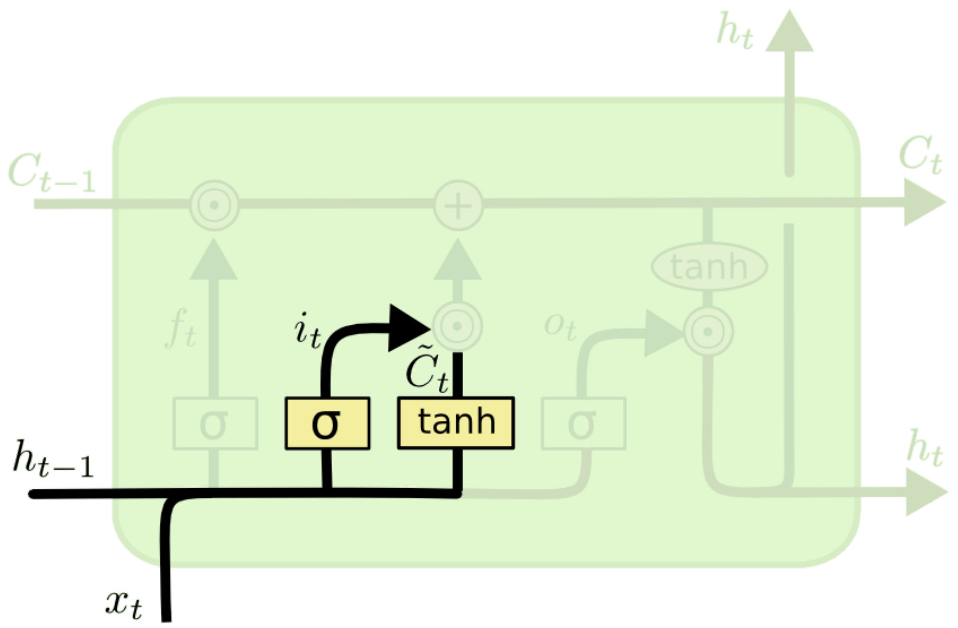
Modification of http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png

Long Short-Term Memory



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

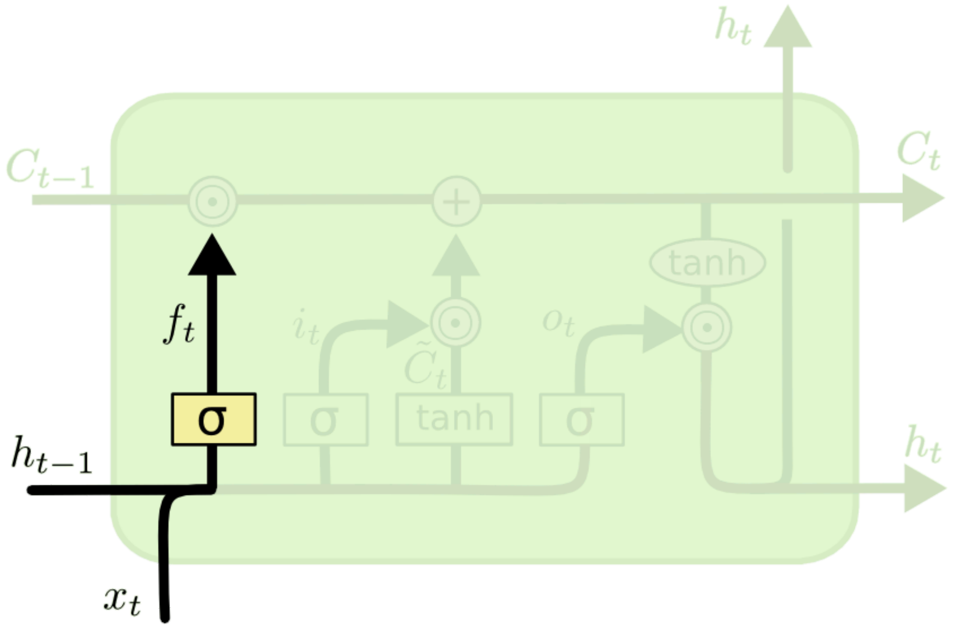
Long Short-Term Memory



$$i_t = \sigma\left(W_i\left[h_{t-1}, x_t\right] + b_i\right)$$
$$\tilde{C}_t = \tanh(W_C\left[h_{t-1}, x_t\right] + b_C)$$

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

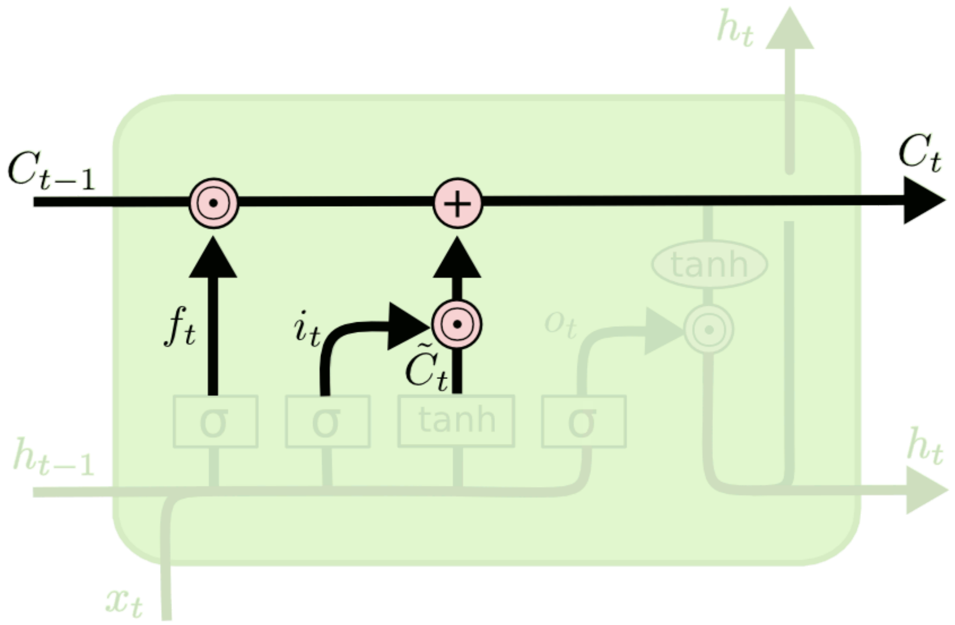
Long Short-Term Memory



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

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

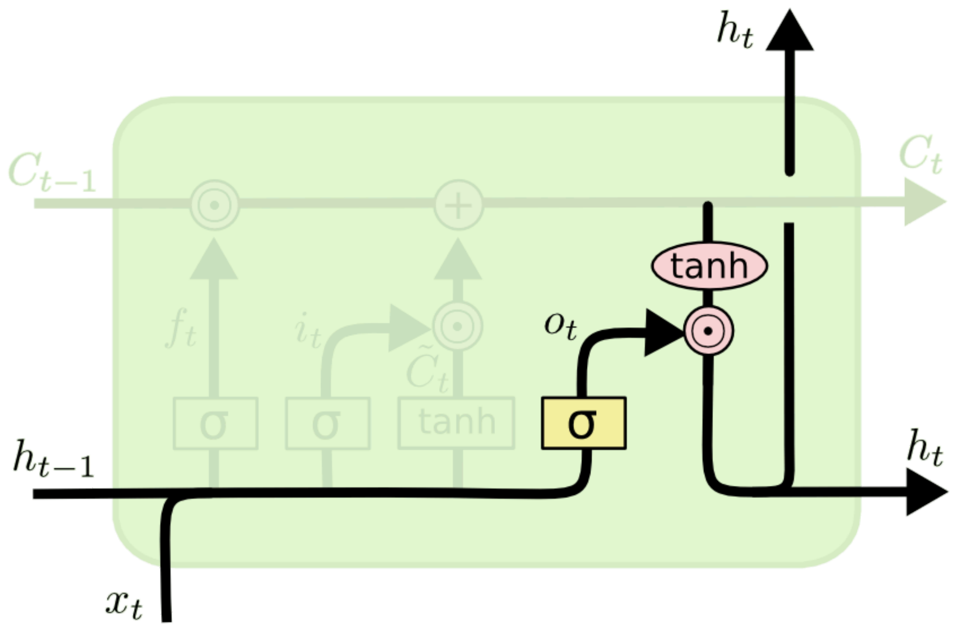
Long Short-Term Memory



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

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

Long Short-Term Memory



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

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

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

Gated Recurrent Unit (GRU)

Gated Recurrent Unit

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

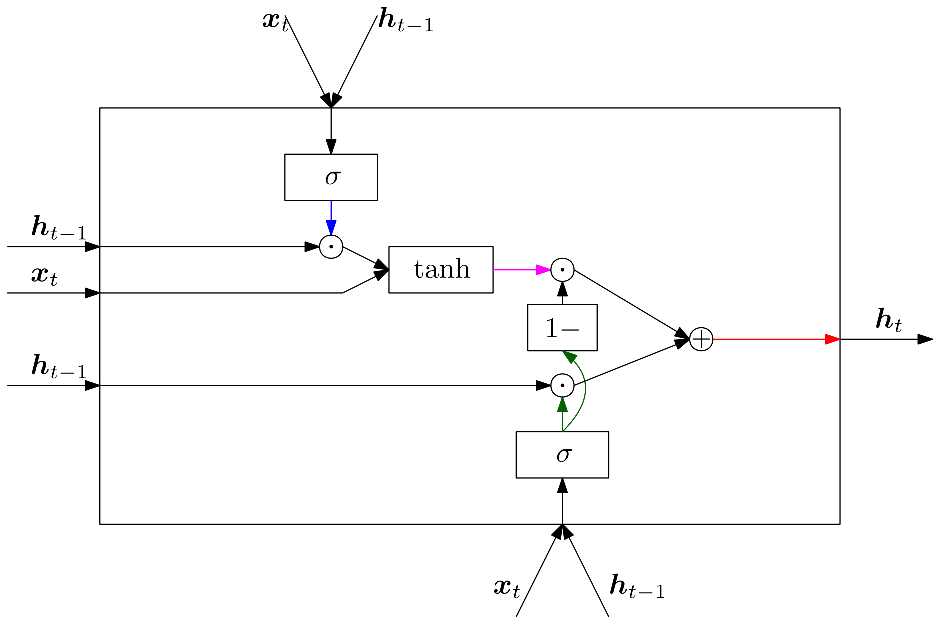
- no memory cell,
- forgetting and updating tied together.

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

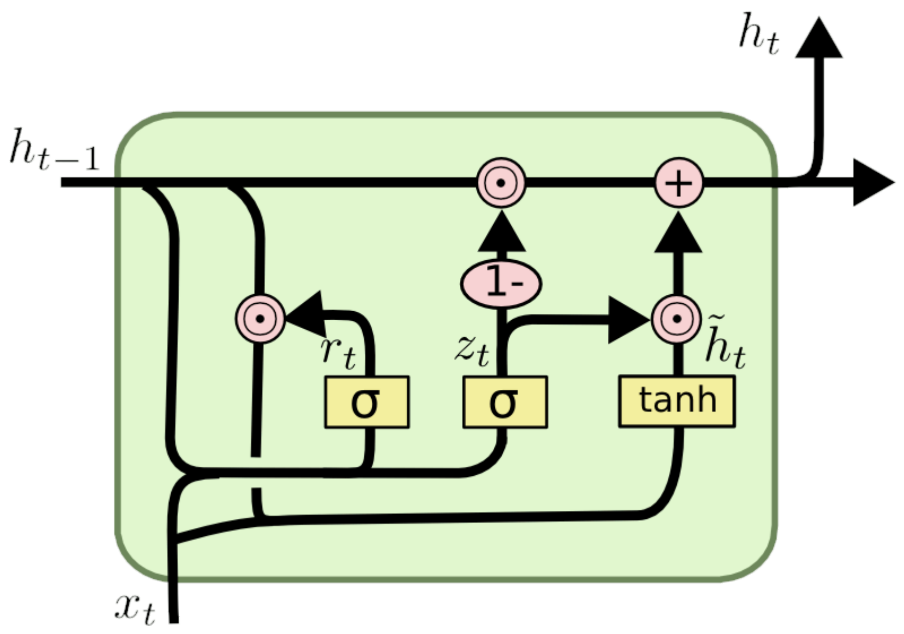
$$\boldsymbol{u}_t \leftarrow \sigma(\boldsymbol{W}^u \boldsymbol{x}_t + \boldsymbol{V}^u \boldsymbol{h}_{t-1} + \boldsymbol{b}^u)$$

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

$$\boldsymbol{h}_t \leftarrow \boldsymbol{u}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{u}_t) \odot \hat{\boldsymbol{h}}_t$$



Gated Recurrent Unit



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

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

$$\tilde h_t = \tanh\left(W\left[r_t \odot h_{t-1}, x_t\right]\right)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde h_t$$

Modification of 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 difference between LSTM and GRU on a real-word task, see for example
    - T. Dozat et al.: *Deep Biaffine Attention for Neural Dependency Parsing* https://arxiv.org/abs/1611.01734

GRU and LSTM Comparison

Arch.	Arith.	XML	PTB
Tanh	0.29493	0.32050	0.08782
LSTM	0.89228	0.42470	0.08912
LSTM-f	0.29292	0.23356	0.08808
LSTM-i	0.75109	0.41371	0.08662
LSTM-o	0.86747	0.42117	0.08933
LSTM-b	0.90163	0.44434	0.08952
GRU	0.89565	0.45963	0.09069
MUT1	**0.92135**	**0.47483**	0.08968
MUT2	0.89735	**0.47324**	0.09036
MUT3	0.90728	0.46478	**0.09161**

Table 1. Best next-step-prediction *accuracies* achieved by the various architectures (so larger numbers are better). LSTM-{f,i,o} refers to the LSTM without the forget, input, and output gates, respectively. LSTM-b refers to the LSTM with the additional bias to the forget gate.

Table 1 of "An Empirical Exploration of Recurrent Network Architectures" by Jozefowicz et al.

Arch.	5M-tst	10M-v	20M-v	20M-tst
Tanh	4.811	4.729	4.635	4.582 (97.7)
LSTM	4.699	4.511	4.437	4.399 (81.4)
LSTM-f	4.785	4.752	4.658	4.606 (100.8)
LSTM-i	4.755	4.558	4.480	4.444 (85.1)
LSTM-o	4.708	4.496	4.447	4.411 (82.3)
LSTM-b	4.698	4.437	4.423	**4.380 (79.83)**
GRU	4.684	4.554	4.559	4.519 (91.7)
MUT1	4.699	4.605	4.594	4.550 (94.6)
MUT2	4.707	4.539	4.538	4.503 (90.2)
MUT3	4.692	4.523	4.530	4.494 (89.47)

Table 3. Perplexities on the PTB. The prefix (e.g., 5M) denotes the number of parameters in the model. The suffix "v" denotes validation negative log likelihood, the suffix"tst" refers to the test set. The perplexity for select architectures is reported in parentheses. We used dropout only on models that have 10M or 20M parameters, since the 5M models did not benefit from dropout at all, and most dropout-free models achieved a test perplexity of 108, and never greater than 120. In particular, the perplexity of the best models without dropout is below 110, which outperforms the results of Mikolov et al. (2014).

Table 3 of "An Empirical Exploration of Recurrent Network Architectures" by Jozefowicz et al.

SimpleRNN, GRU, and LSTM Initialization

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 in Keras and TensorFlow – this so-called **recurrent kernel** (the concatenation of all the $\boldsymbol{V}^i$, $\boldsymbol{V}^f$, $\boldsymbol{V}^o$, $\boldsymbol{V}^y$ matrices) is initialized with a randomly generated orthogonal matrix. This **orthogonal** initialization is available as `torch.nn.init.orthogonal_` in PyTorch, but it is not used as the default initialization of RNNs.

Our `npfl138` module changes the RNN initialization defaults to:

- initialize the recurrent kernel using orthogonal initialization,
- initialize the non-recurrent kernel using Glorot (Xavier) initialization,
- initialize biases to zero, with the exception of forget cell bias in LSTM initialized to 1.

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:

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

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 $\boldsymbol{h}_{t-1}$.

Analogously to LSTM, the transform gate bias $\boldsymbol{b}_T$ should be initialized to a negative number.

Highway Networks on MNIST

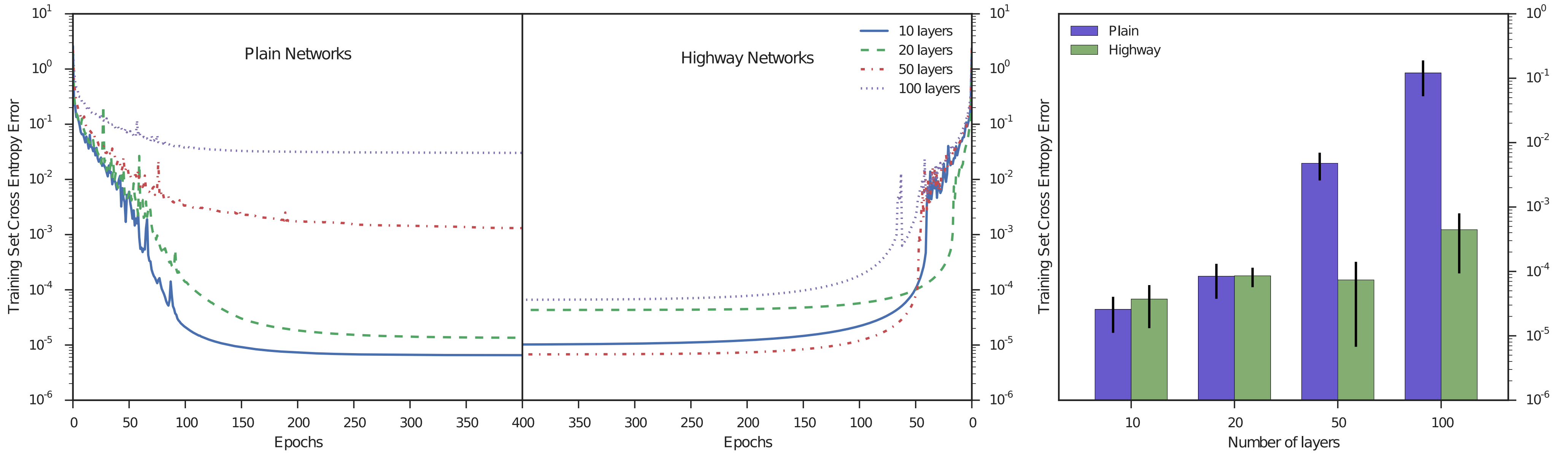


Figure 1: Comparison of optimization of plain networks and highway networks of various depths. *Left:* The training curves for the best hyperparameter settings obtained for each network depth. *Right:* Mean performance of top 10 (out of 100) hyperparameter settings. Plain networks become much harder 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

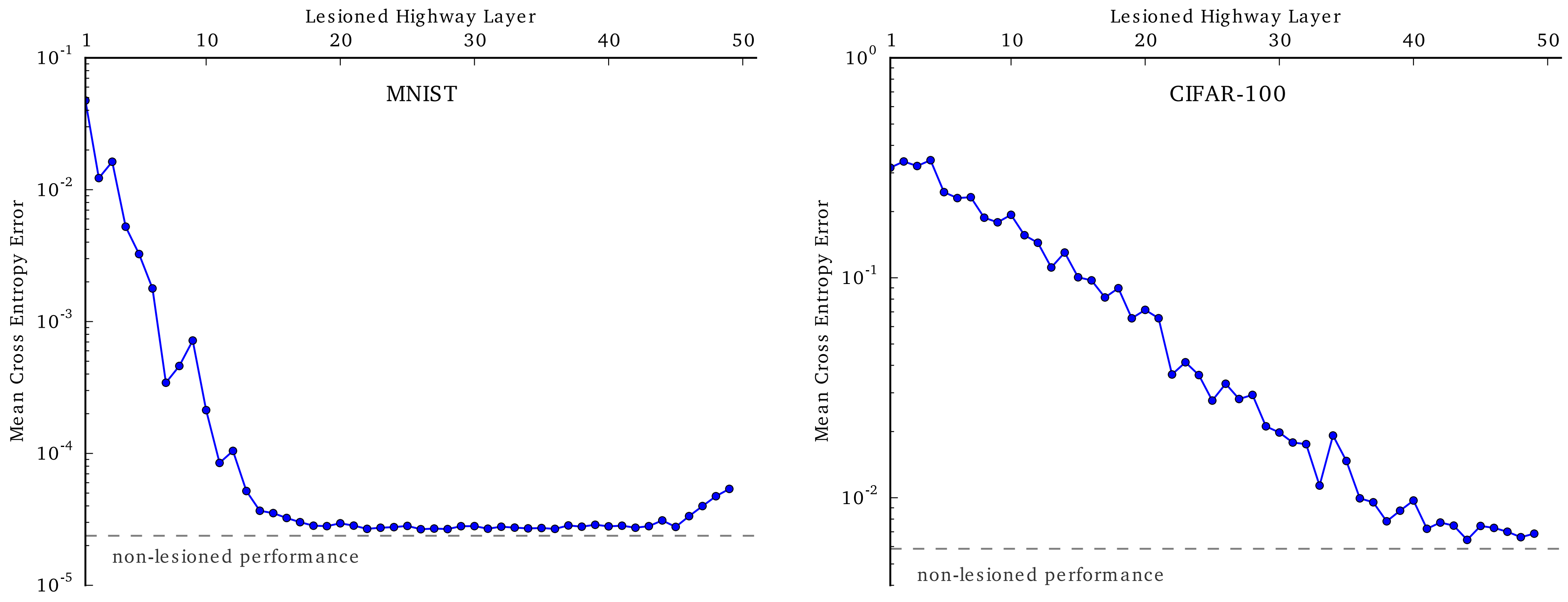


Figure 4: Lesioned training set performance (y-axis) of the best 50-layer highway networks on MNIST (left) and CIFAR-100 (right), as a function of the lesioned layer (x-axis). Evaluated on the full training set while forcefully closing all the transform gates of a single layer at a time. The non-lesioned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

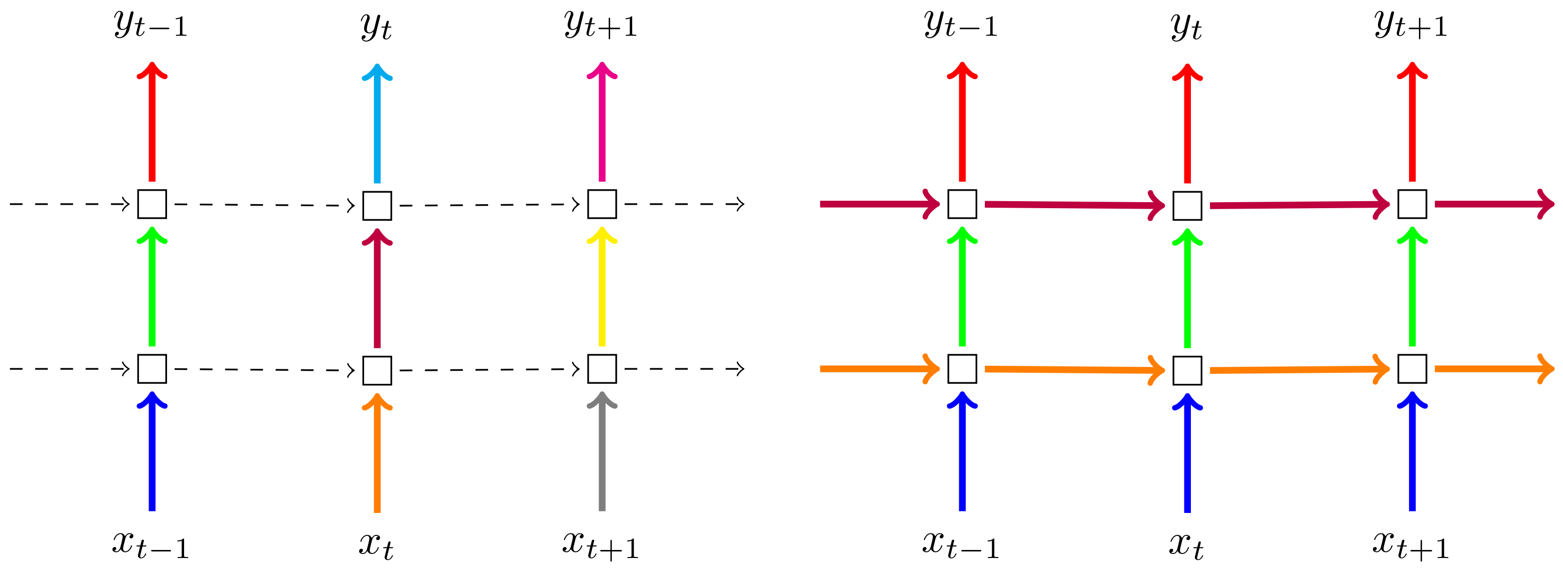
Figure 4 of "Training Very Deep Networks", https://arxiv.org/abs/1507.06228

Regularizing RNNs

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.
  - In PyTorch, `torch.nn.{RNN,LSTM,GRU}` has a parameter `dropout`, which adds a dropout layer with given dropout probability on the output of *all but the last RNN layers*, i.e., on the places that you cannot place it manually.
    - *However, using a multi-layer `torch.nn.{RNN,LSTM,GRU}` module does not use residual connections, so personally I never use the multi-layer variant.*
- Several techniques were designed to allow using dropout on hidden states.
  - Variational Dropout
  - Recurrent Dropout
  - Zoneout

Variational Dropout



(a) Naive dropout RNN (b) Variational RNN

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, the same dropout mask must be used for all time steps (`torch.nn.Dropout1d` in PyTorch; using `noise_shape` in Keras).

In practice, the variational dropout on the hidden states is not frequently used, because it is not supported by GPU-accelerated algorithms.

Regularizing RNNs

Recurrent Dropout

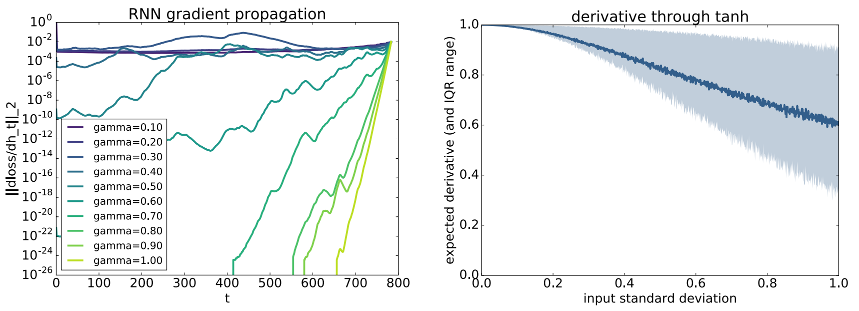
Dropout only candidate states (i.e., values added to the memory cell in LSTM and previous state in GRU), independently in every time-step.

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

Layer Normalization

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.

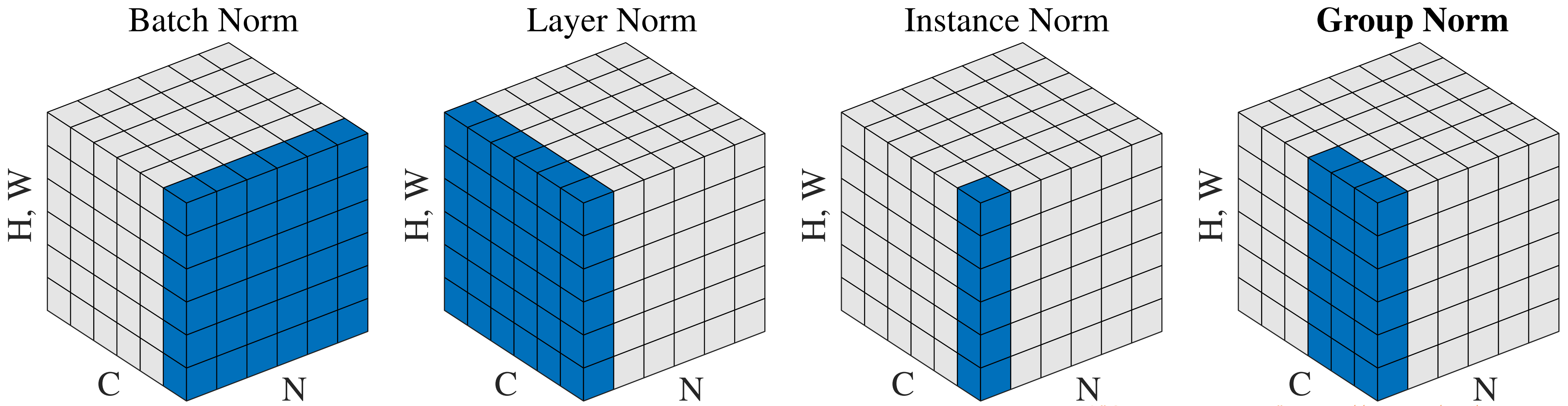


Figure 2 of "Group Normalization", https://arxiv.org/abs/1803.08494

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 $\boldsymbol{y}$

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

When applying layer normalization after convolutions on an image, we want it to be positional invariant (the same as with batch normalization); therefore, we have parameters only for channels, i.e., $\beta \in \mathbb{R}^C$ and $\gamma \in \mathbb{R}^C$, but we compute μ and $\boldsymbol{\sigma}^2$ over the whole image.

Layer Normalization

Much more stable than batch normalization for RNN regularization.

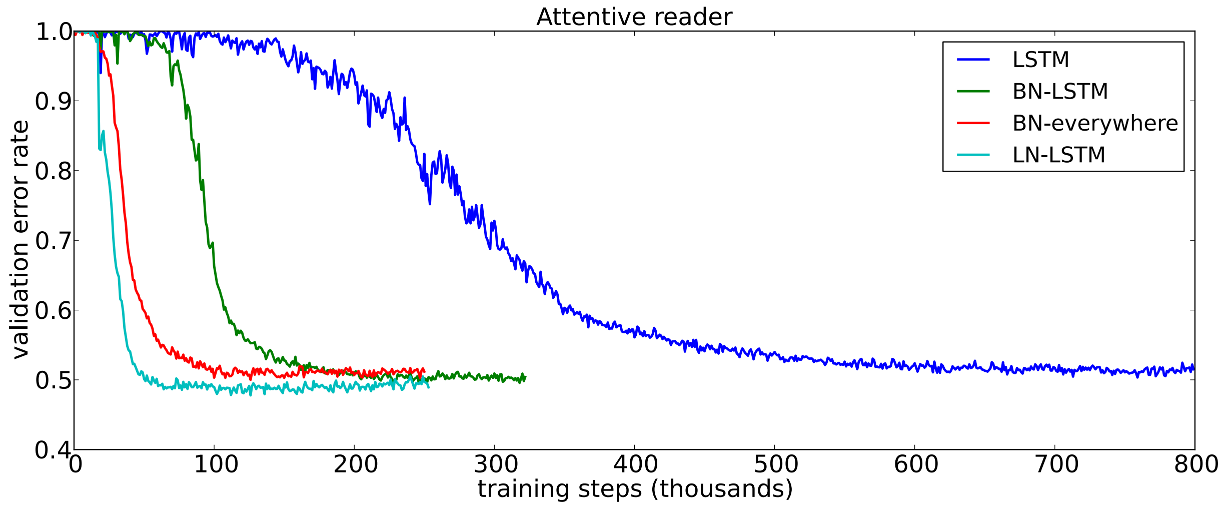


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

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

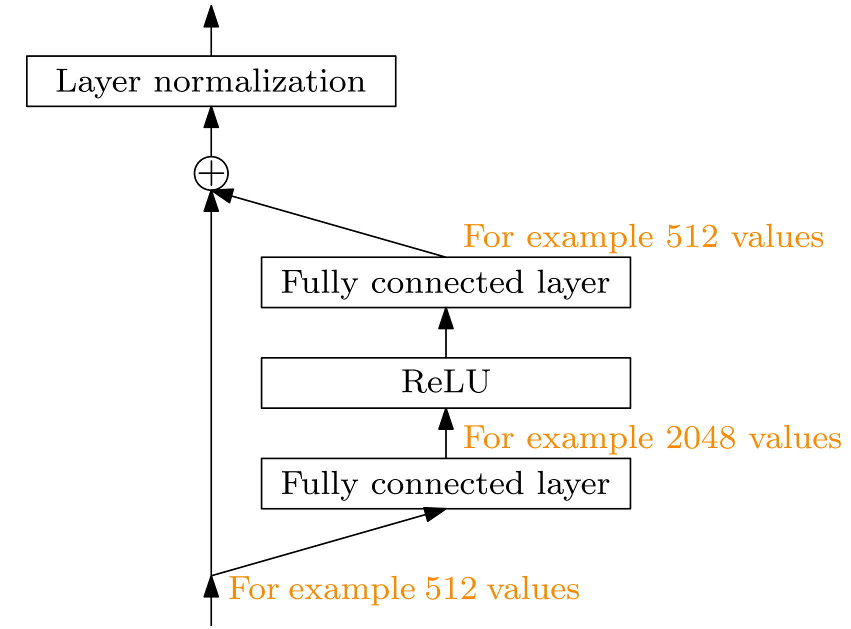
	Weight matrix re-scaling	Weight matrix re-centering	Weight vector re-scaling	Dataset re-scaling	Dataset re-centering	Single training case re-scaling
Batch norm	Invariant	No	Invariant	Invariant	Invariant	No
Weight norm	Invariant	No	Invariant	No	No	No
Layer norm	Invariant	Invariant	No	Invariant	No	Invariant

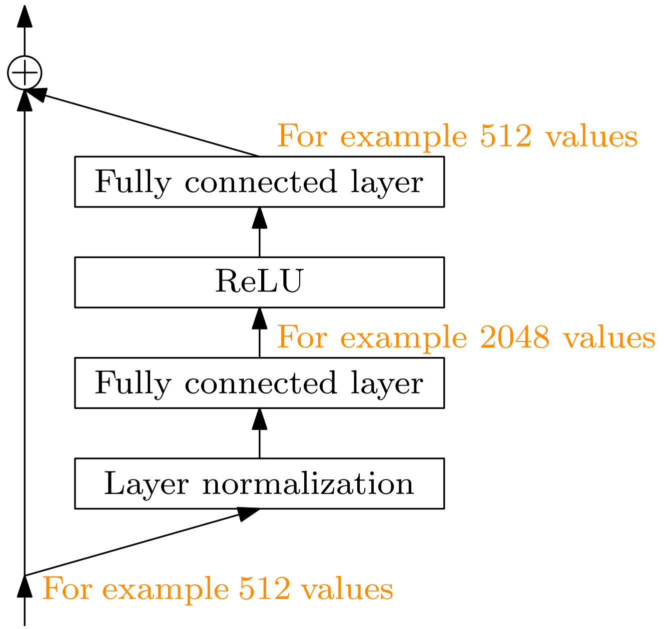
Table 1 of "Layer Normalization", https://arxiv.org/abs/1607.06450

Layer Normalization

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

Original "Post-LN" configuration **Improved "Pre-LN" configuration since 2020**





This could be considered a ResNet-like alternative of residual connections for fully-connected layers (better than highway networks). Note the architecture can be interpreted as a variant of a mobile inverted bottleneck 1×1 convolution block.

Group Normalization

Group Normalization is analogous to Layer normalization, but the channels are normalized in groups (by default, $G = 32$).

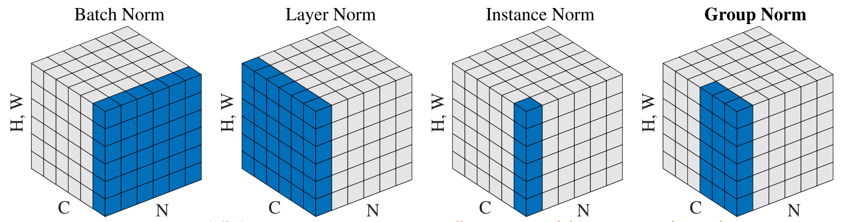


Figure 2 of "Group Normalization", https://arxiv.org/abs/1803.08494

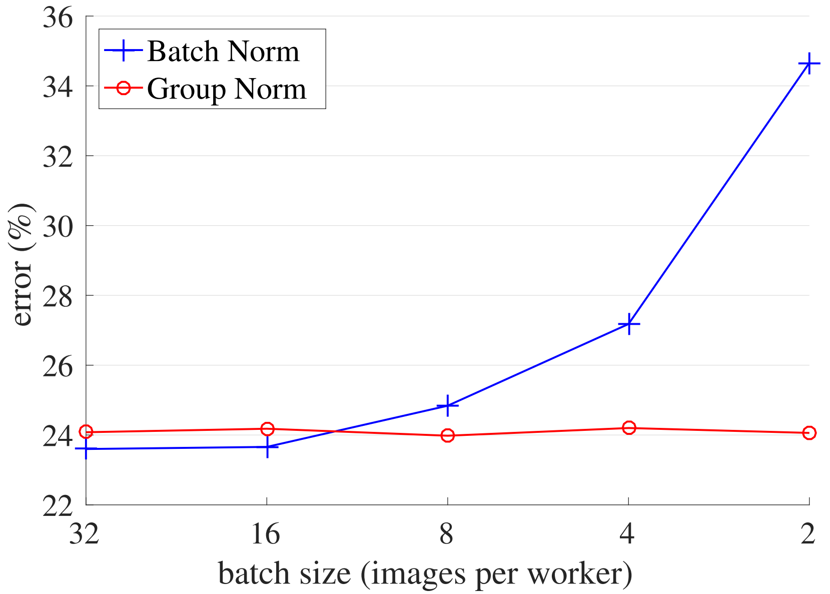


Figure 1 of "Group Normalization", https://arxiv.org/abs/1803.08494

Group Normalization

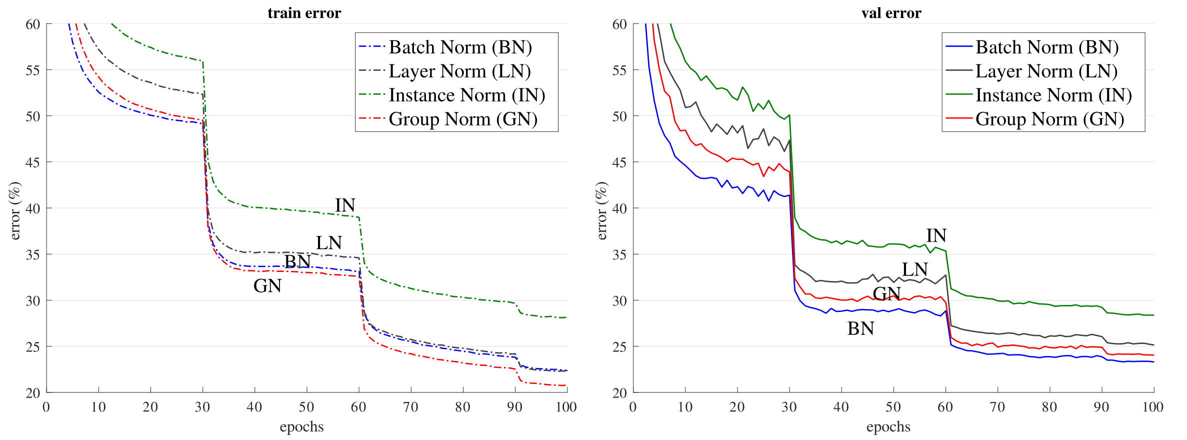


Figure 4. Comparison of error curves with a batch size of 32 images/GPU. We show the ImageNet training error (left) and validation error (right) vs. numbers of training epochs. The model is ResNet-50.

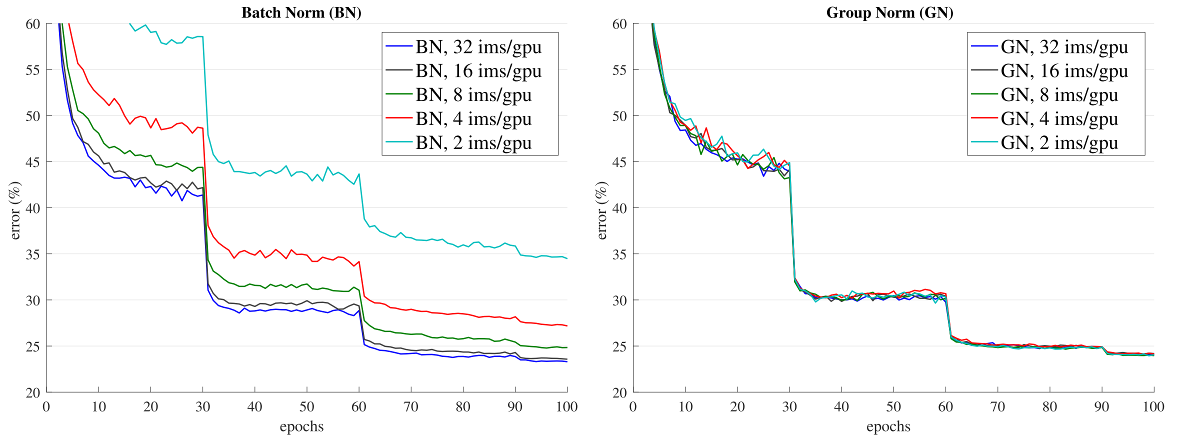
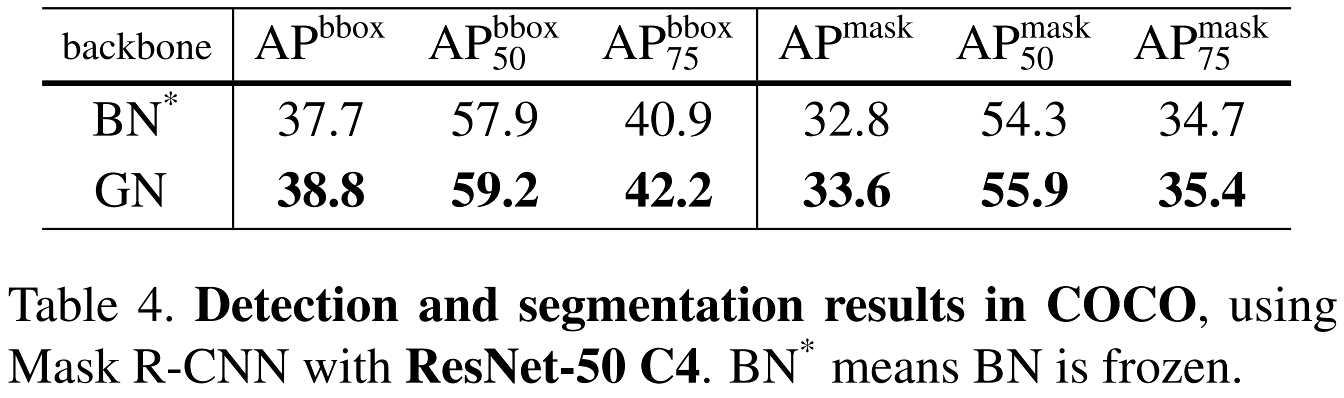


Figure 5. Sensitivity to batch sizes: ResNet-50's validation error of BN (left) and GN (right) trained with 32, 16, 8, 4, and 2 images/GPU.

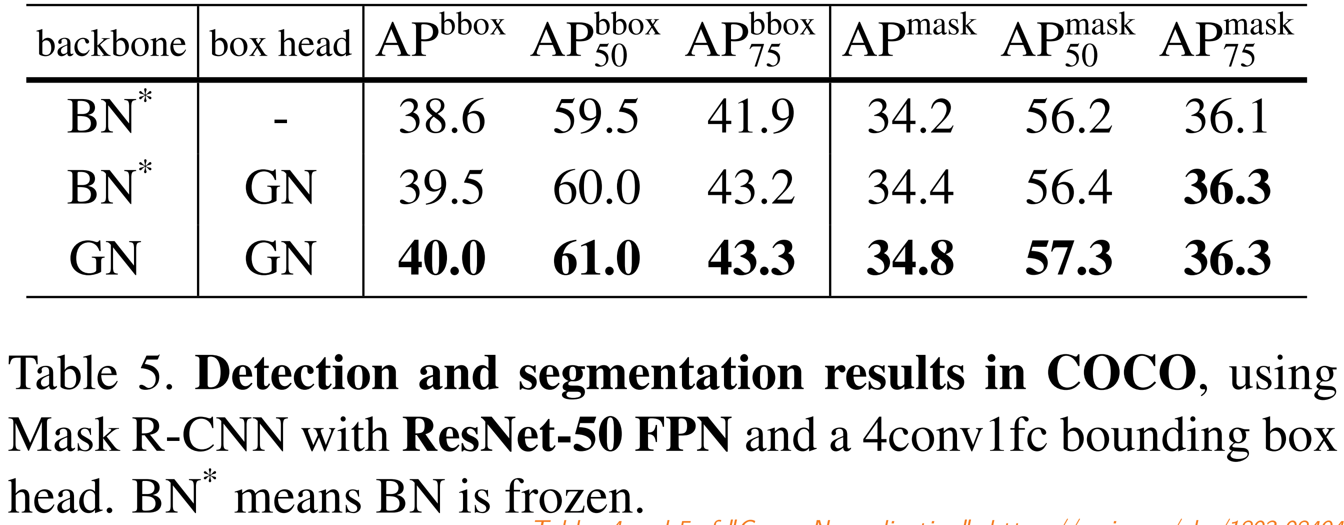
Figures 4 and 5 of "Group Normalization", https://arxiv.org/abs/1803.08494

Group Normalization



backbone	AP^{bbox}	AP^{bbox}_{50}	AP^{bbox}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
BN*	37.7	57.9	40.9	32.8	54.3	34.7
GN	**38.8**	**59.2**	**42.2**	**33.6**	**55.9**	**35.4**

Table 4. **Detection and segmentation results in COCO**, using Mask R-CNN with **ResNet-50 C4**. BN* means BN is frozen.



backbone	box head	AP^{bbox}	AP^{bbox}_{50}	AP^{bbox}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
BN*	-	38.6	59.5	41.9	34.2	56.2	36.1
BN*	GN	39.5	60.0	43.2	34.4	56.4	**36.3**
GN	GN	**40.0**	**61.0**	**43.3**	**34.8**	**57.3**	**36.3**

Table 5. **Detection and segmentation results in COCO**, using Mask R-CNN with **ResNet-50 FPN** and a 4conv1fc bounding box head. BN* means BN is frozen.

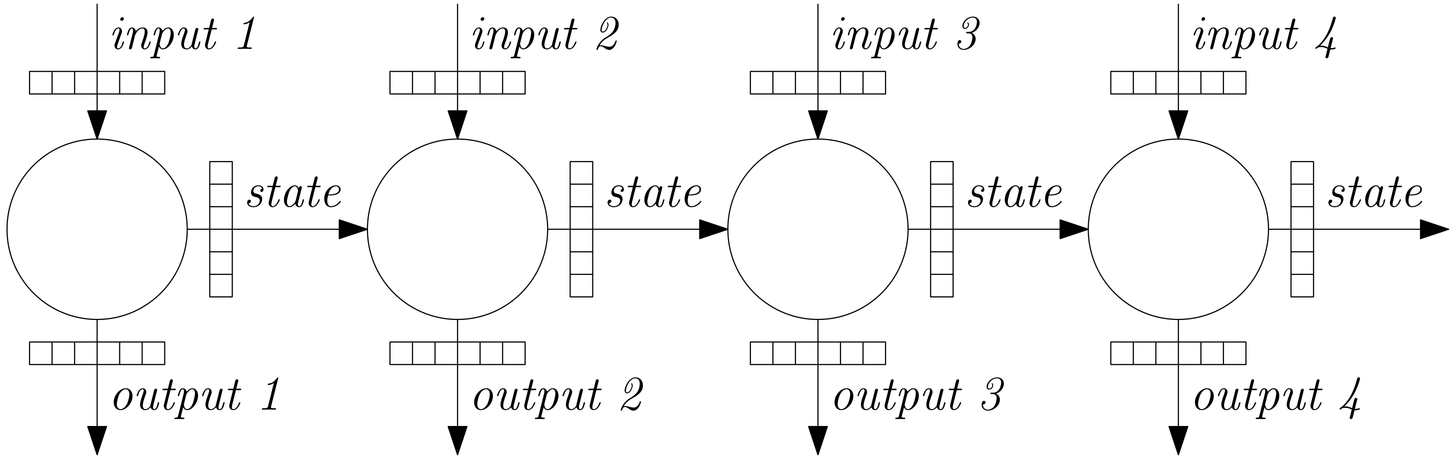
Tables 4 and 5 of "Group Normalization", https://arxiv.org/abs/1803.08494

RNN Architectures and Tasks

Basic RNN Architectures and Tasks

Sequence Element Representation

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

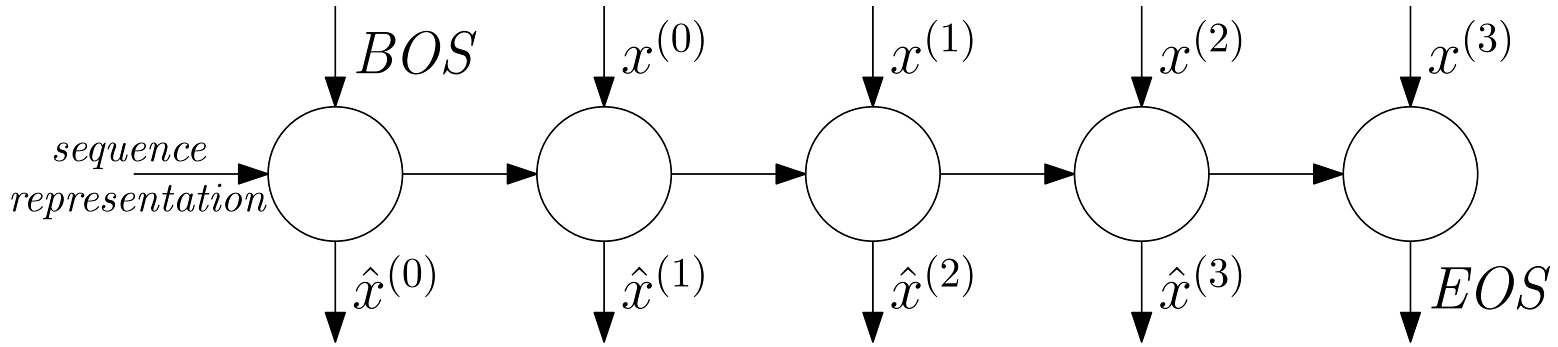


Sequence Representation

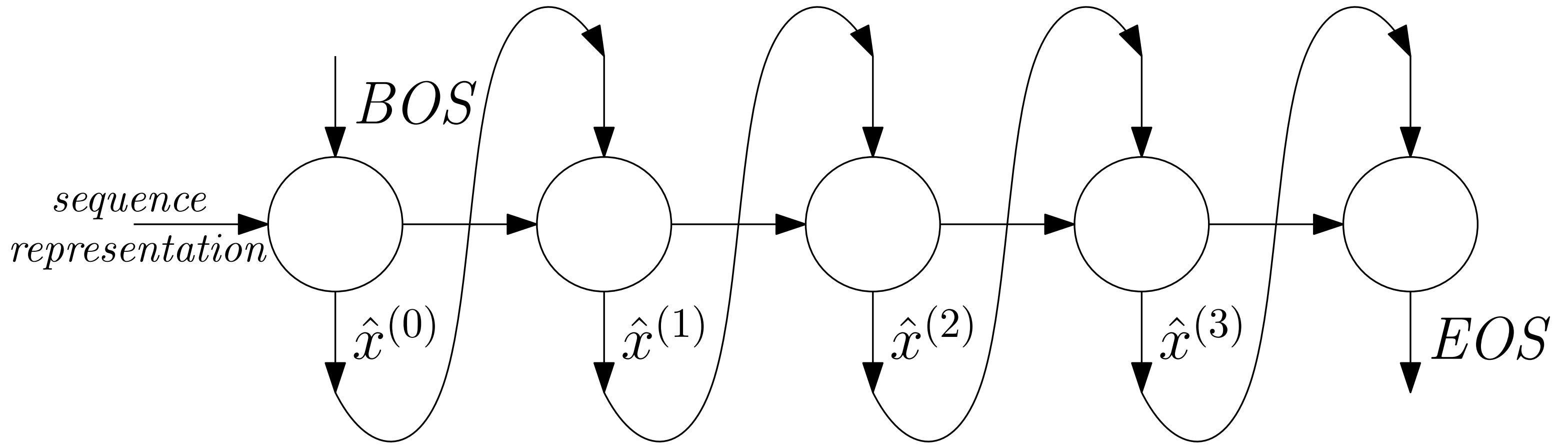
Generate a single output for the whole sequence (either the last output or the last state).

Sequence Prediction

During training, predict next sequence element.

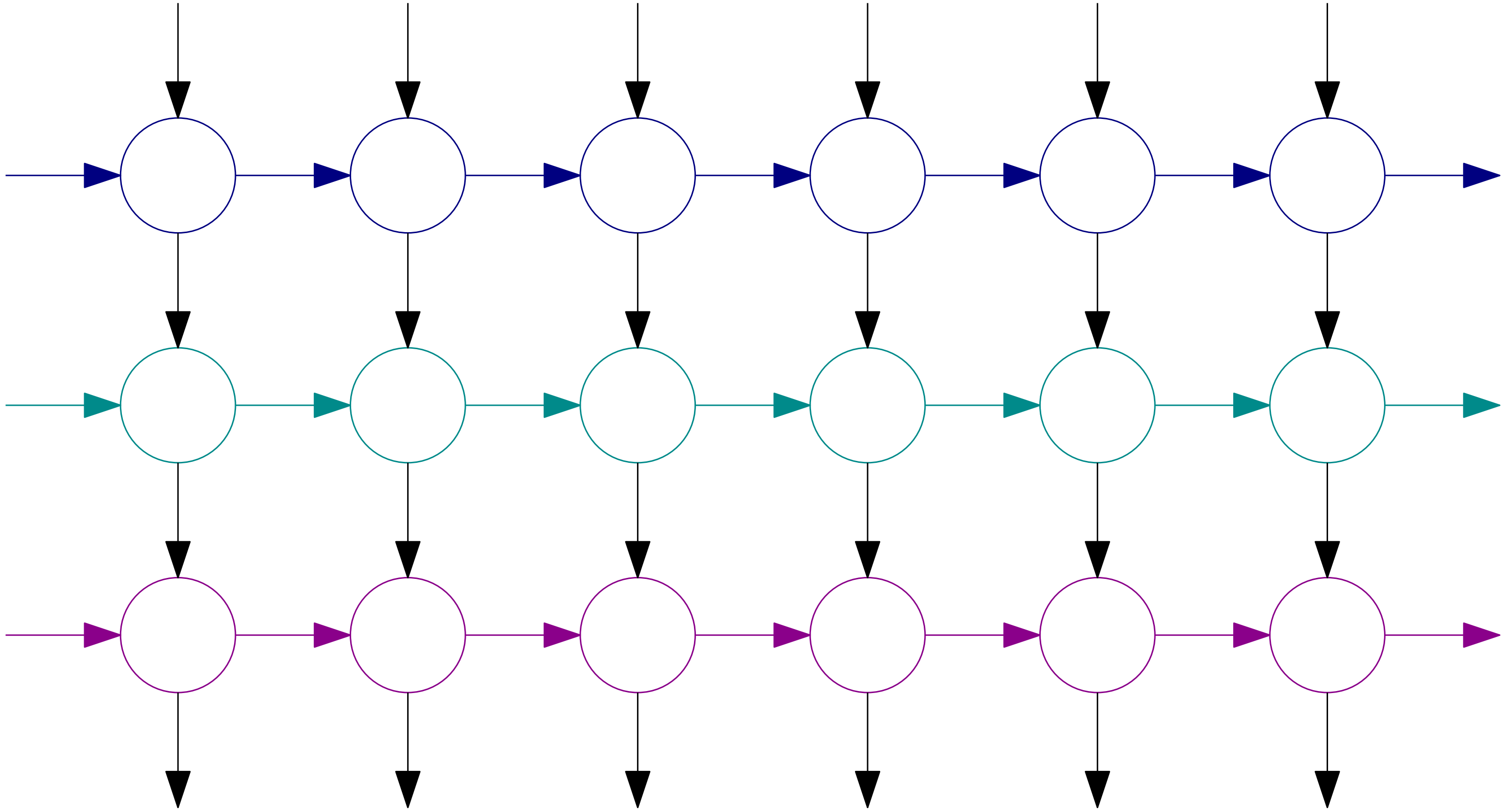


During inference, use predicted elements as further inputs.



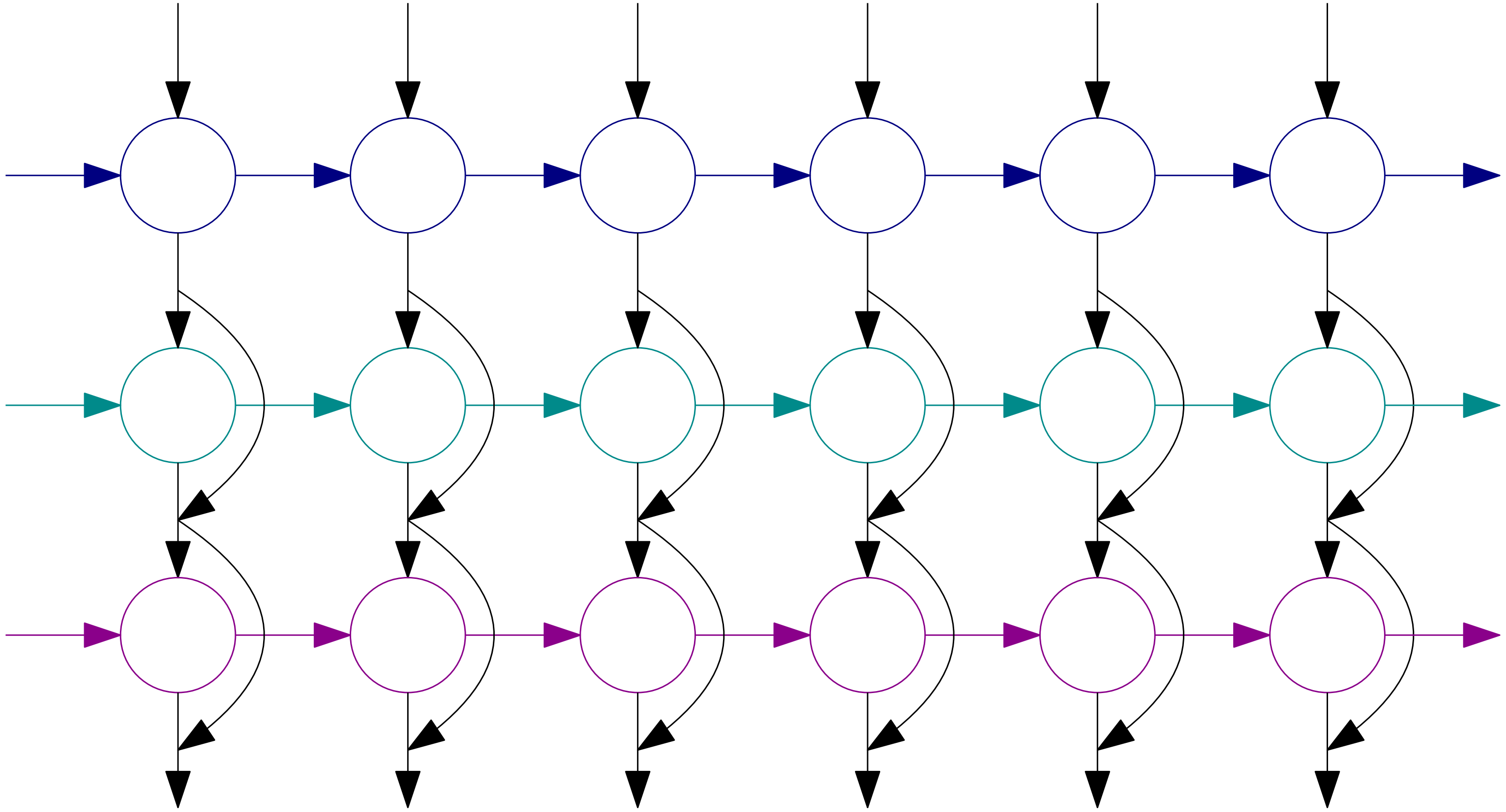
Multilayer RNNs

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



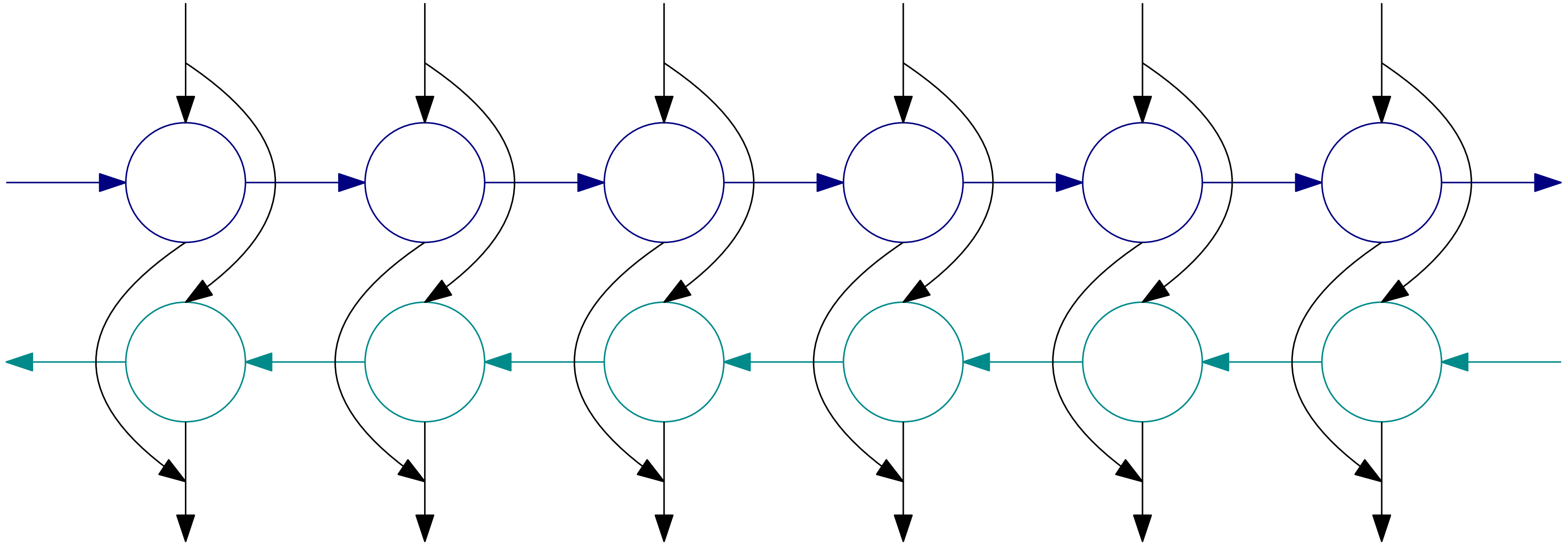
Multilayer RNNs

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

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.

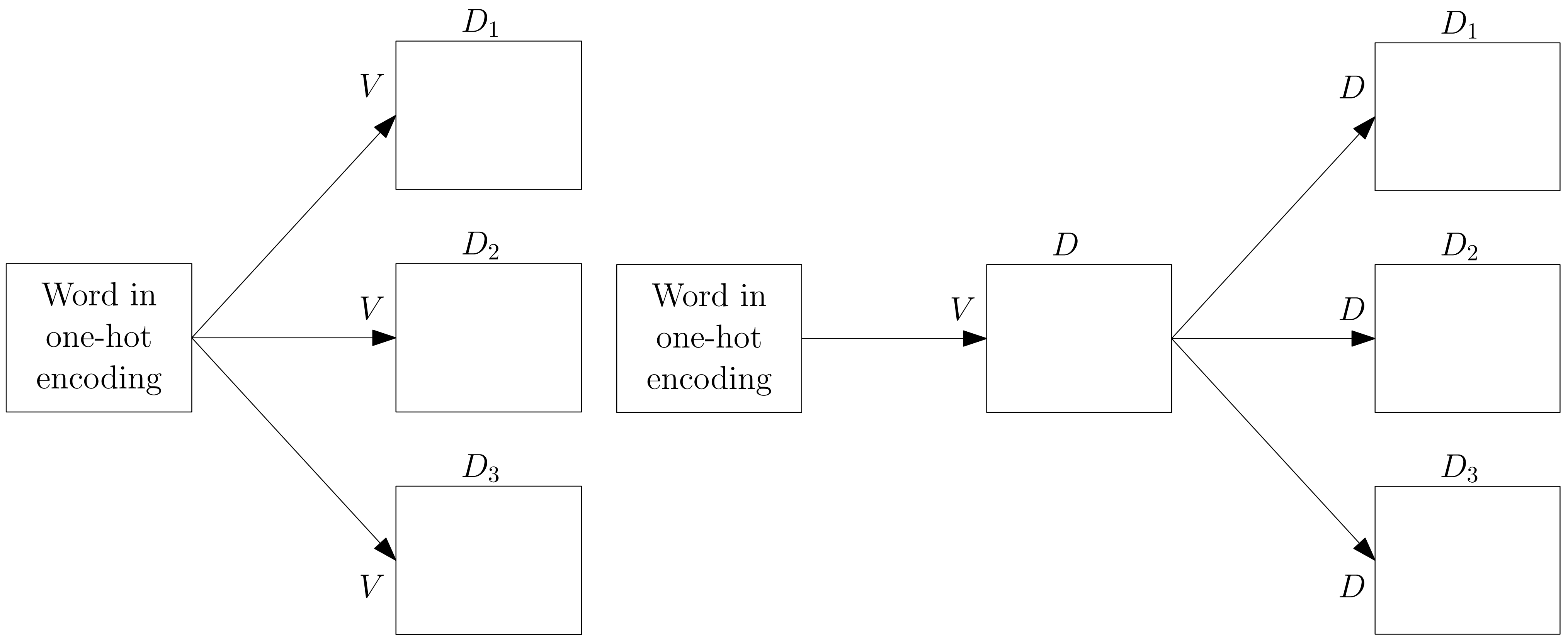
Instead, the so-called **embedding** layer is used, which is much more efficient. When a matrix is multiplied by an one-hot encoded vector (all but one zeros and exactly one 1), the row corresponding to that 1 is selected, so the embedding layer can be implemented only as a simple lookup.

In PyTorch, the embedding layer is available as

```
torch.nn.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.



Character-Level Word Embeddings

Word Embeddings for Unknown Words

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.

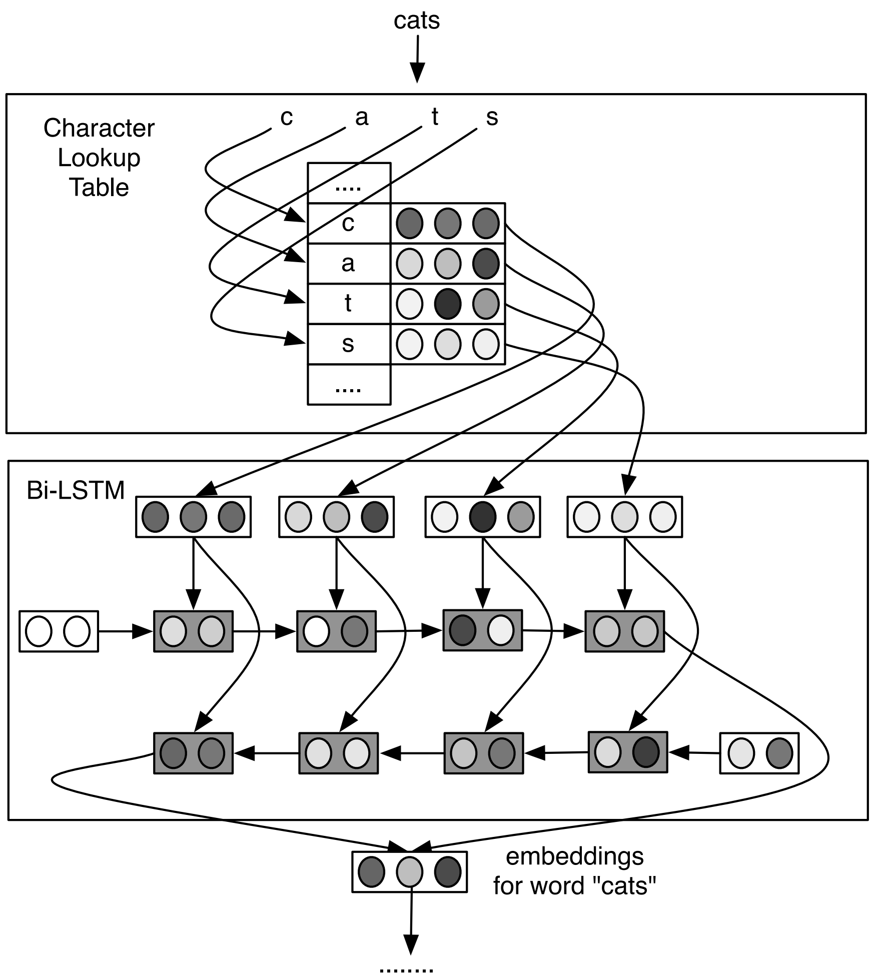


Figure 1 of "Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation", https://arxiv.org/abs/1508.02096

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

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).

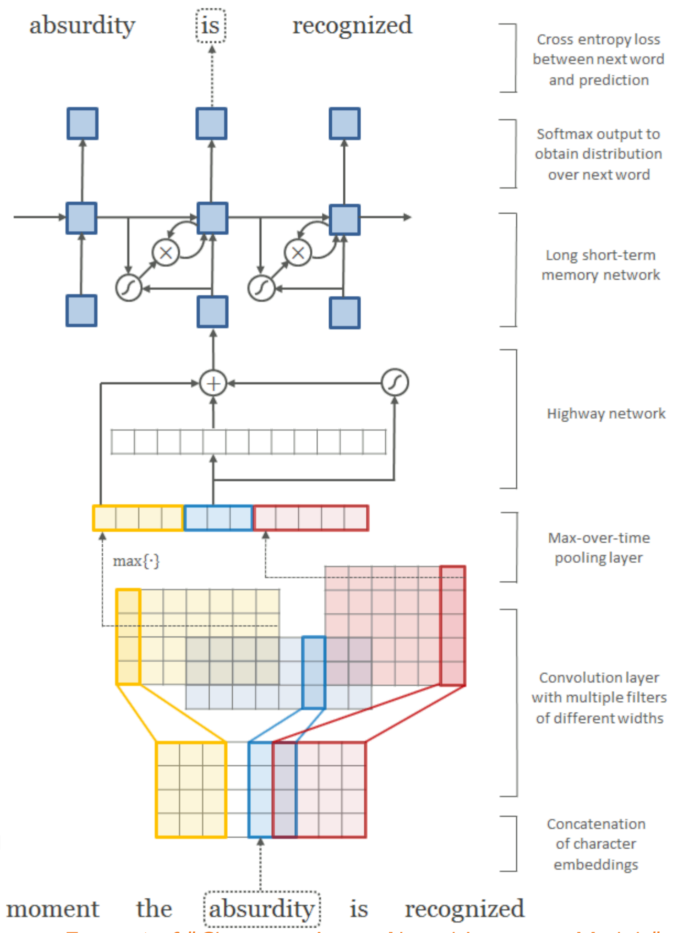

absurdity
is
recognized
Cross entropy loss between next word and prediction
Softmax output to obtain distribution over next word
Long short-term memory network
Highway network
max{·}
Max-over-time pooling layer
Convolution layer with multiple filters of different widths
Concatenation of character embeddings
moment
the
absurdity
is
recognized

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

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

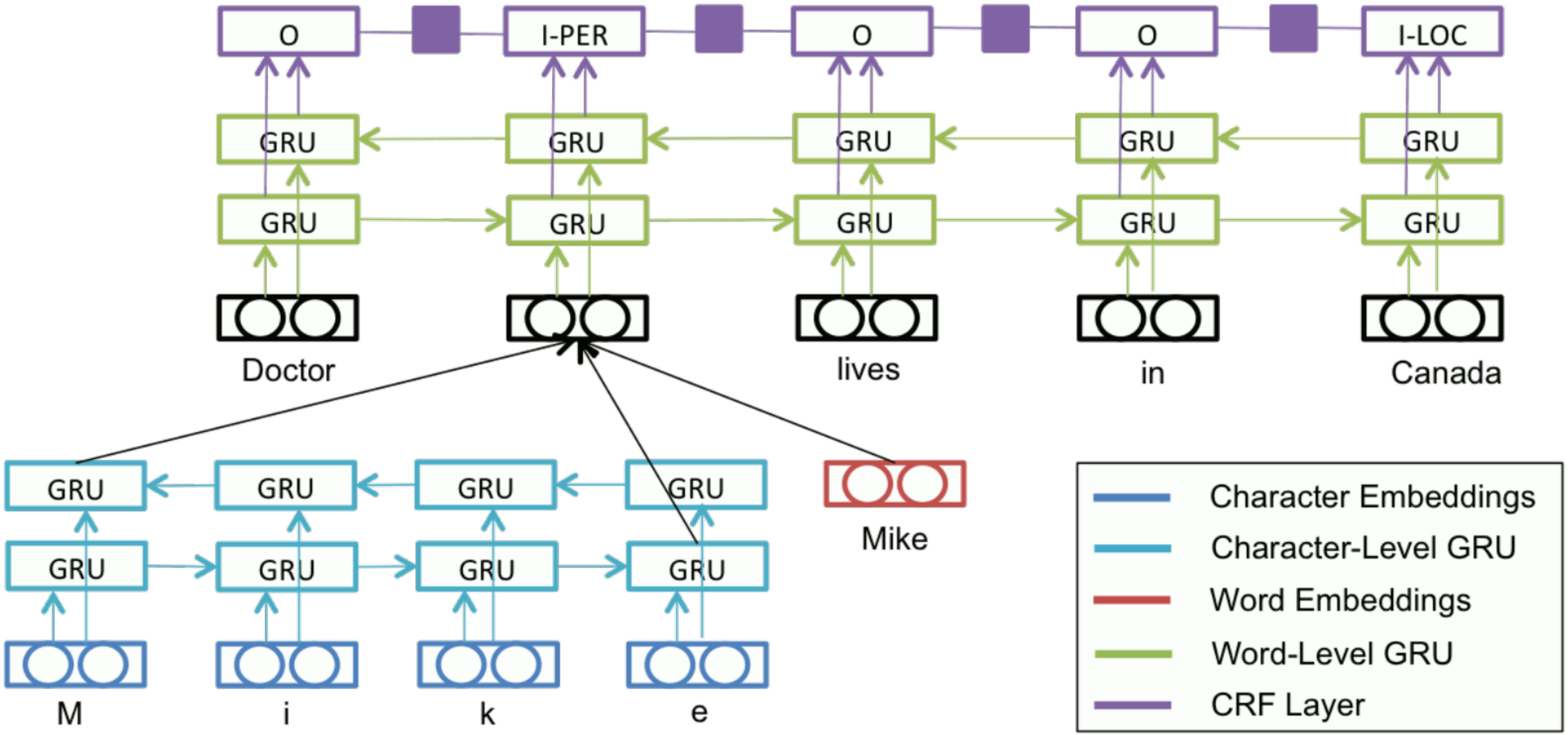


Figure 1 of "Multi-Task Cross-Lingual Sequence Tagging from Scratch", https://arxiv.org/abs/1603.06270