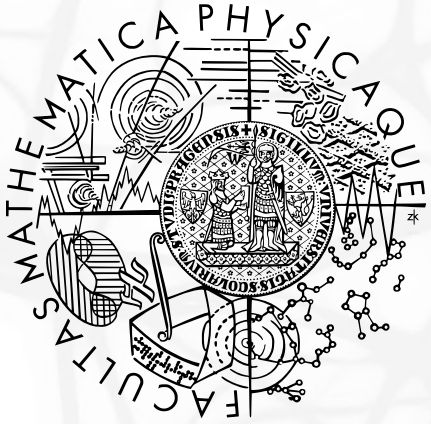
NPFL114, Lecture 08

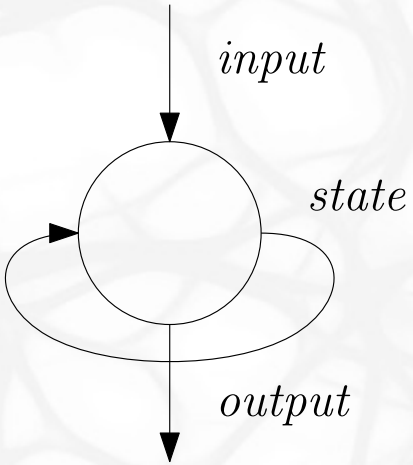
Recurrent Neural Networks II, Word Embeddings



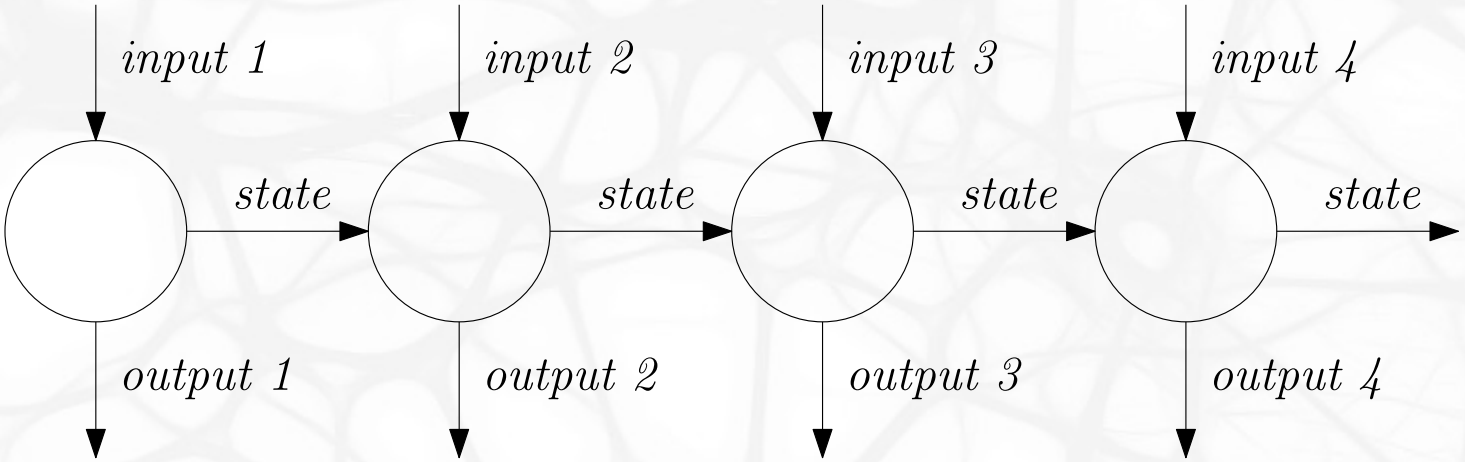
Milan Straka

Recurrent Neural Networks

Single RNN cell



Unrolled RNN cells



Challenge of Long-term Dependencies

Consider a RNN cell which given an input $\boldsymbol{x}^{(t)}$ and previous state $\boldsymbol{s}^{(t-1)}$ computes the new state as

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

Challenge of Long-term Dependencies

Consider a RNN cell which given an input $\boldsymbol{x}^{(t)}$ and previous state $\boldsymbol{s}^{(t-1)}$ computes the new state as

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

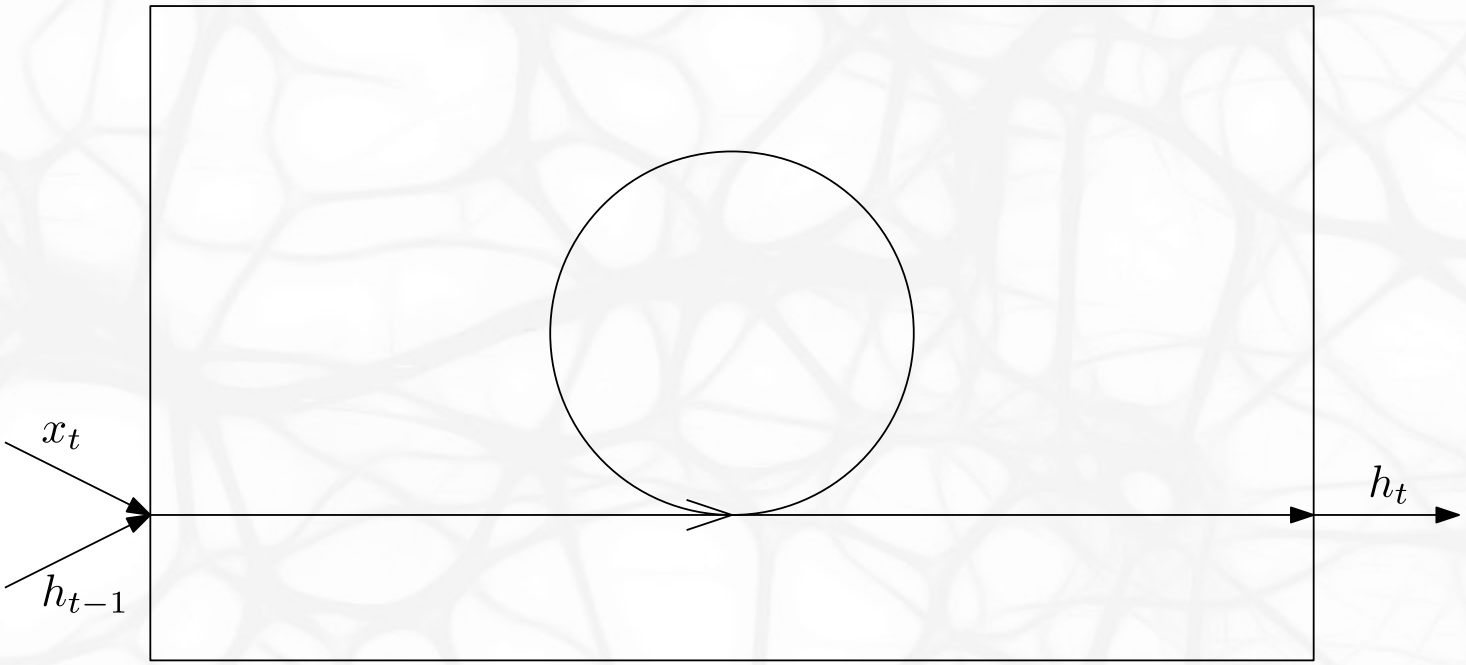
RNN cells generally suffer a lot from vanishing/exploding gradients (*the challenge of long-term dependencies*), a problem attributed to the fact that *same* function is iteratively applied many times.

Long Short-Term Memory

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

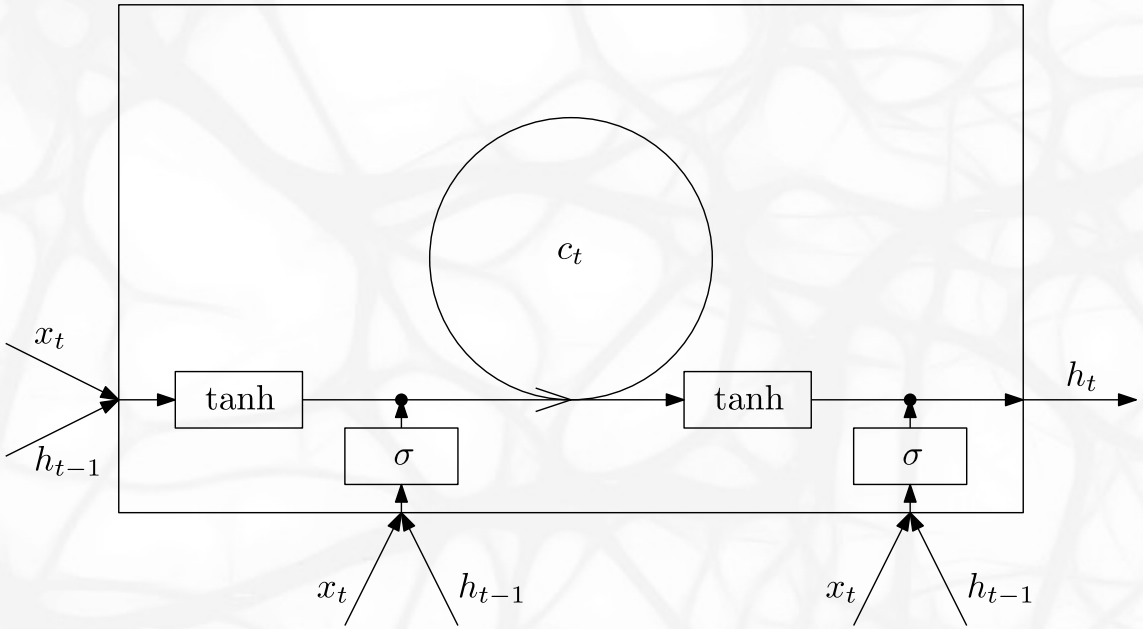
$$f' = \mathbf{1}.$$

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



Long Short-Term Memory

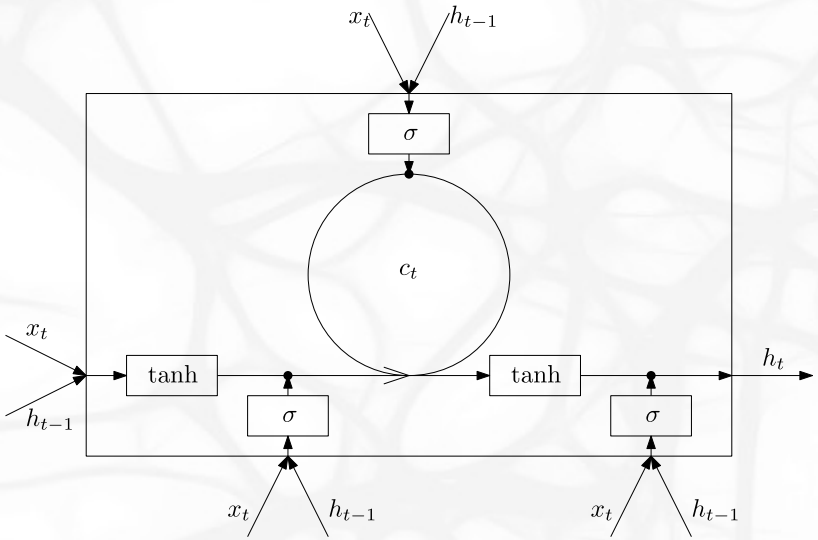
They also propose an *input* and *output* gates which control the flow of information into and out of the carrousel (*memory cell* $\boldsymbol{c}_t$).



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

Long Short-Term Memory

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

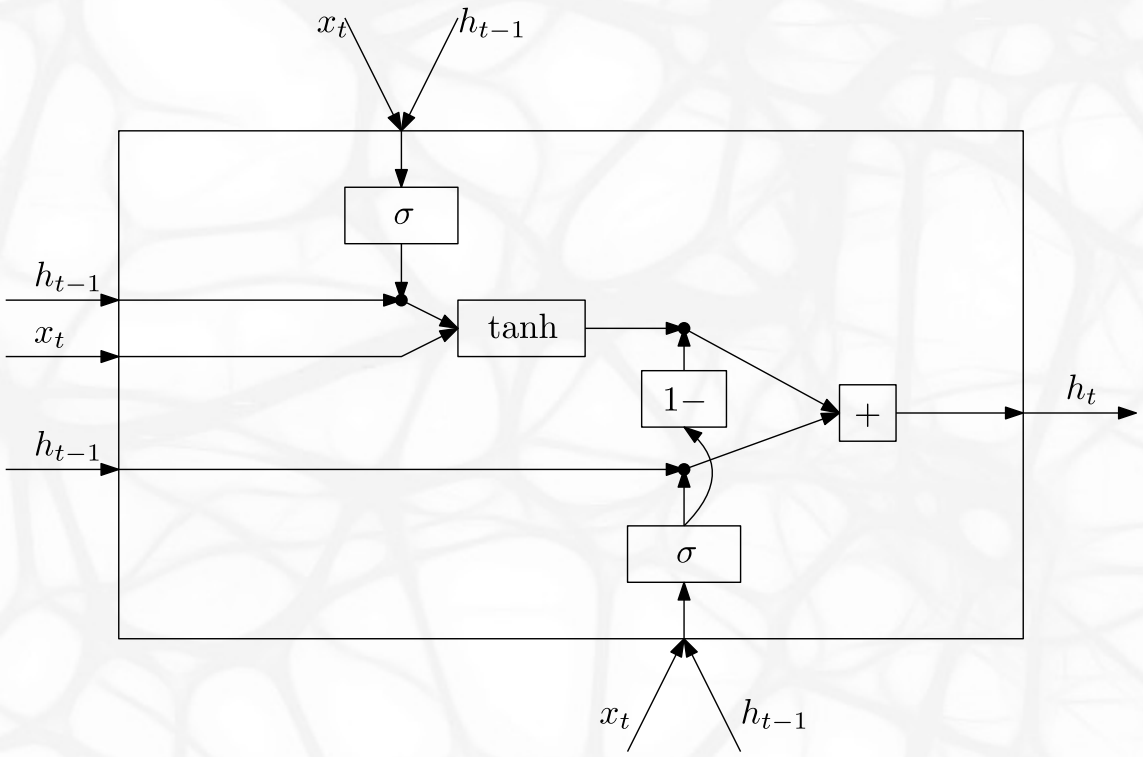


$$\boldsymbol{i}_t \leftarrow \sigma(\boldsymbol{W}^i \boldsymbol{x}_t + \boldsymbol{V}^i \boldsymbol{h}_{t-1} + \boldsymbol{b}^i)$$
$$\boldsymbol{f}_t \leftarrow \sigma(\boldsymbol{W}^f \boldsymbol{x}_t + \boldsymbol{V}^f \boldsymbol{h}_{t-1} + \boldsymbol{b}^f)$$
$$\boldsymbol{o}_t \leftarrow \sigma(\boldsymbol{W}^o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

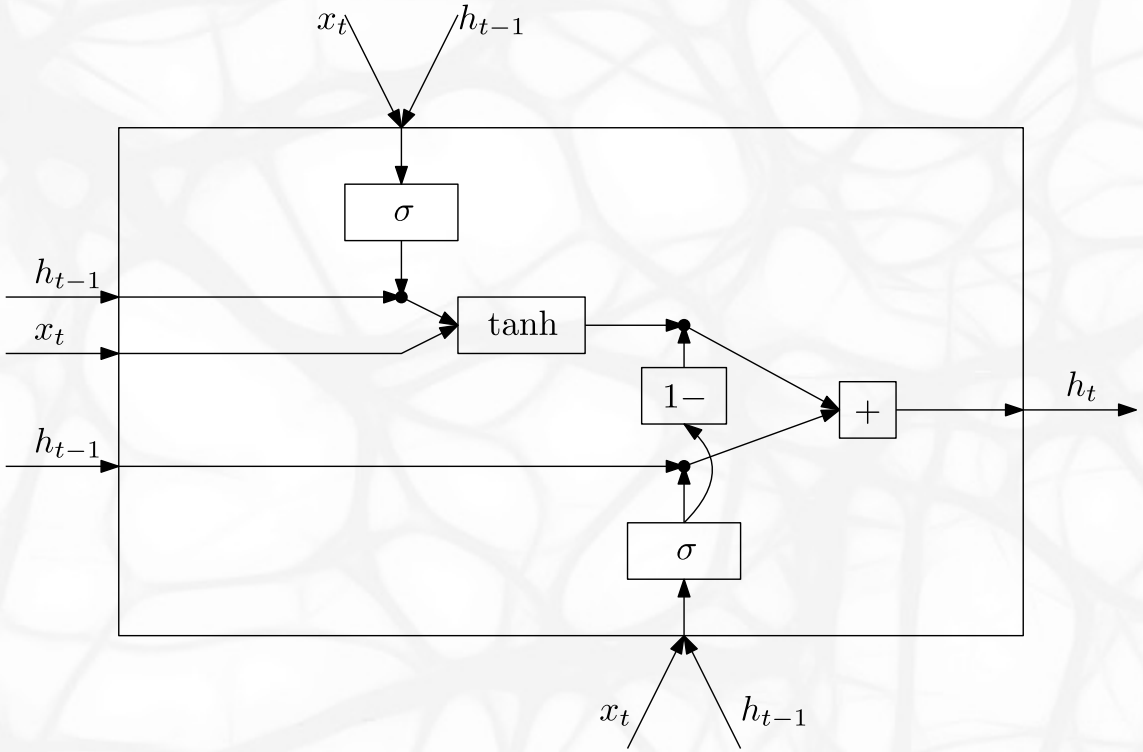
Gated Recurrent Unit

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



Gated Recurrent Unit



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

Word Embeddings

Word Embeddings

One-hot encoding considers 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.

Word Embeddings

One-hot encoding considers 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.

Word Embeddings

One-hot encoding considers 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.

Unsupervised Word Embeddings

The embeddings can be trained for each task separately.

Unsupervised Word Embeddings

The embeddings can be trained for each task separately.

However, a method of precomputing word embeddings have been proposed, based on *distributional hypothesis*:

Words that are used in the same contexts tend to have similar meanings.

The distributional hypothesis is usually attributed to Firth (1957).

Word2Vec

Mikolov et al. (2013) proposed two very simple architectures for precomputing word embeddings, together with a C multi-threaded implementation `word2vec`.

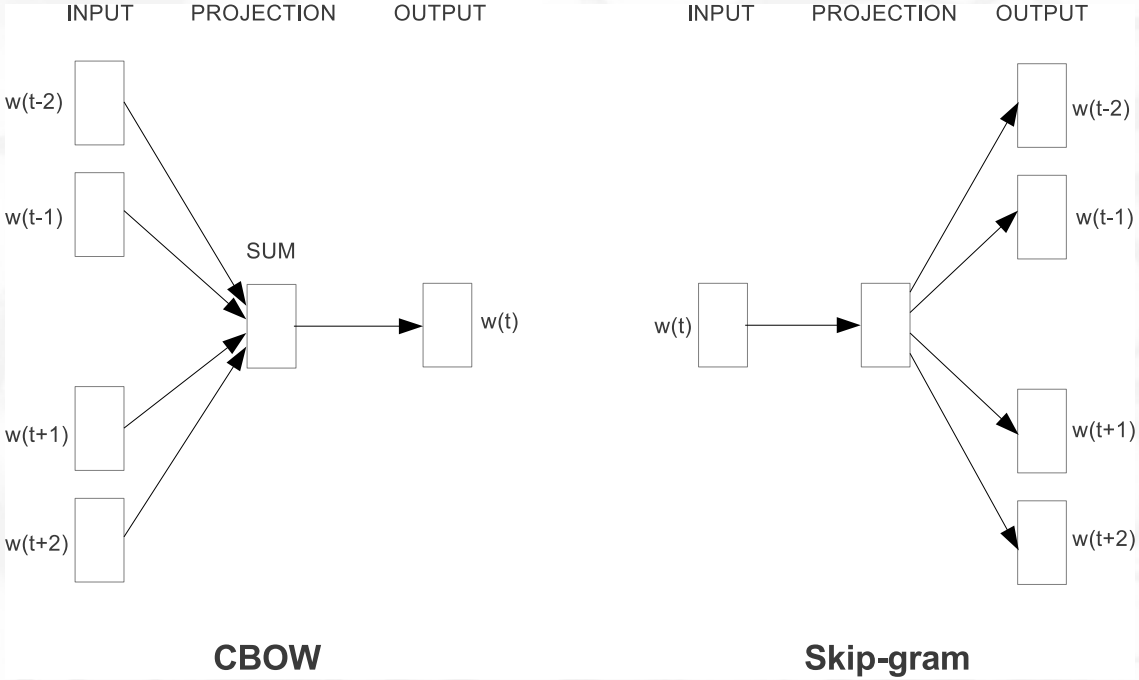
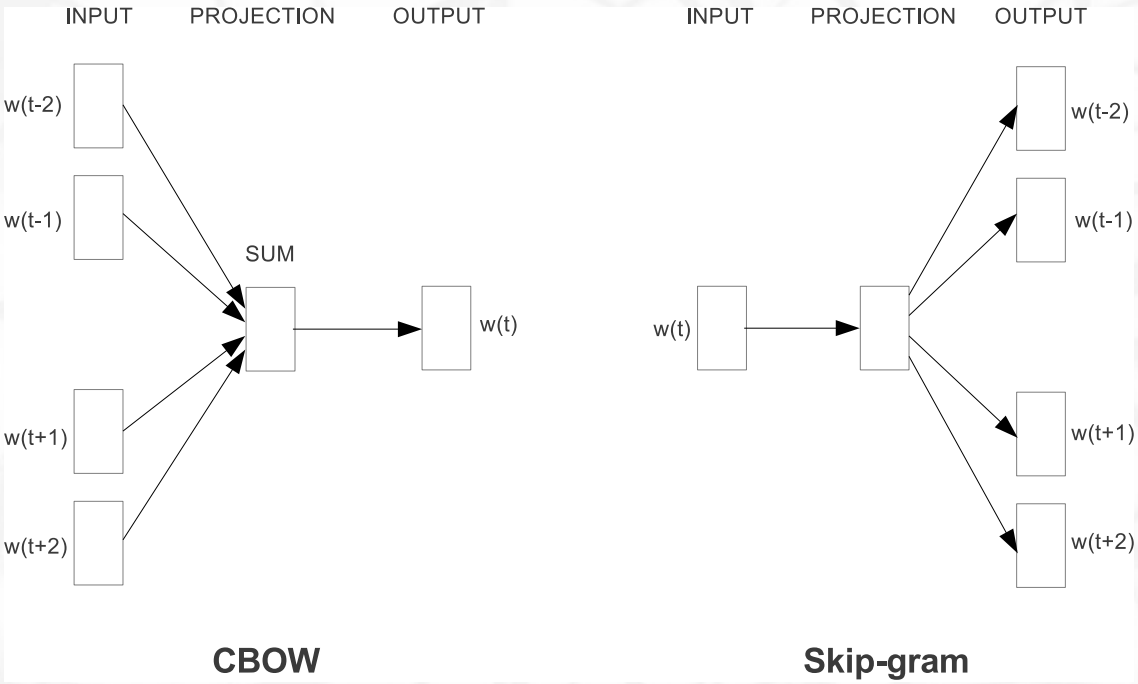


Table 8: *Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).*

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Word2Vec – SkipGram Model



Considering input word w_i and output w_o, the Skip-gram model defines

$$p(w_o|w_i) \stackrel{\text{def}}{=} \frac{e^{\boldsymbol{W}'_{w_o}\boldsymbol{V}_{w_i}}}{\sum_w e^{\boldsymbol{W}'_w\boldsymbol{V}_{w_i}}}.$$

Word2Vec – Hierarchical Softmax

Instead of a large softmax, we construct a binary tree over the words, with a sigmoid classifier for each node.

If word w corresponds to a path $n_1, n_2, \ldots, n_L$, we define

$$p_{\text{HS}}(w|w_i) \overset{\text{def}}{=} \prod_{j=1}^{L-1} \sigma([+1 \text{ if } n_{j+1} \text{ is right child else -1}] \cdot \boldsymbol{W}_{n_j}^\top \boldsymbol{V}_{w_i}).$$

Word2Vec – Negative Sampling

Instead of a large softmax, we could train individual sigmoids for all words.

We could also only sample the *negative examples* instead of training all of them.

This gives rise to the following *negative sampling* objective:

$$l_{\text{NEG}}(w_o, w_i) \stackrel{\text{def}}{=} \log \sigma(\boldsymbol{W}_{w_o}^\top \boldsymbol{V}_{w_i}) - \sum_{j=1}^{k} \mathbb{E}_{w_j \sim P(w)} \log \sigma(-\boldsymbol{W}_{w_j}^\top \boldsymbol{V}_{w_i}).$$

Word2Vec – Negative Sampling

Instead of a large softmax, we could train individual sigmoids for all words.

We could also only sample the *negative examples* instead of training all of them.

This gives rise to the following *negative sampling* objective:

$$l_{\text{NEG}}(w_o, w_i) \stackrel{\text{def}}{=} \log \sigma(\boldsymbol{W}_{w_o}^\top \boldsymbol{V}_{w_i}) - \sum_{j=1}^{k} \mathbb{E}_{w_j \sim P(w)} \log \sigma(-\boldsymbol{W}_{w_j}^\top \boldsymbol{V}_{w_i}).$$

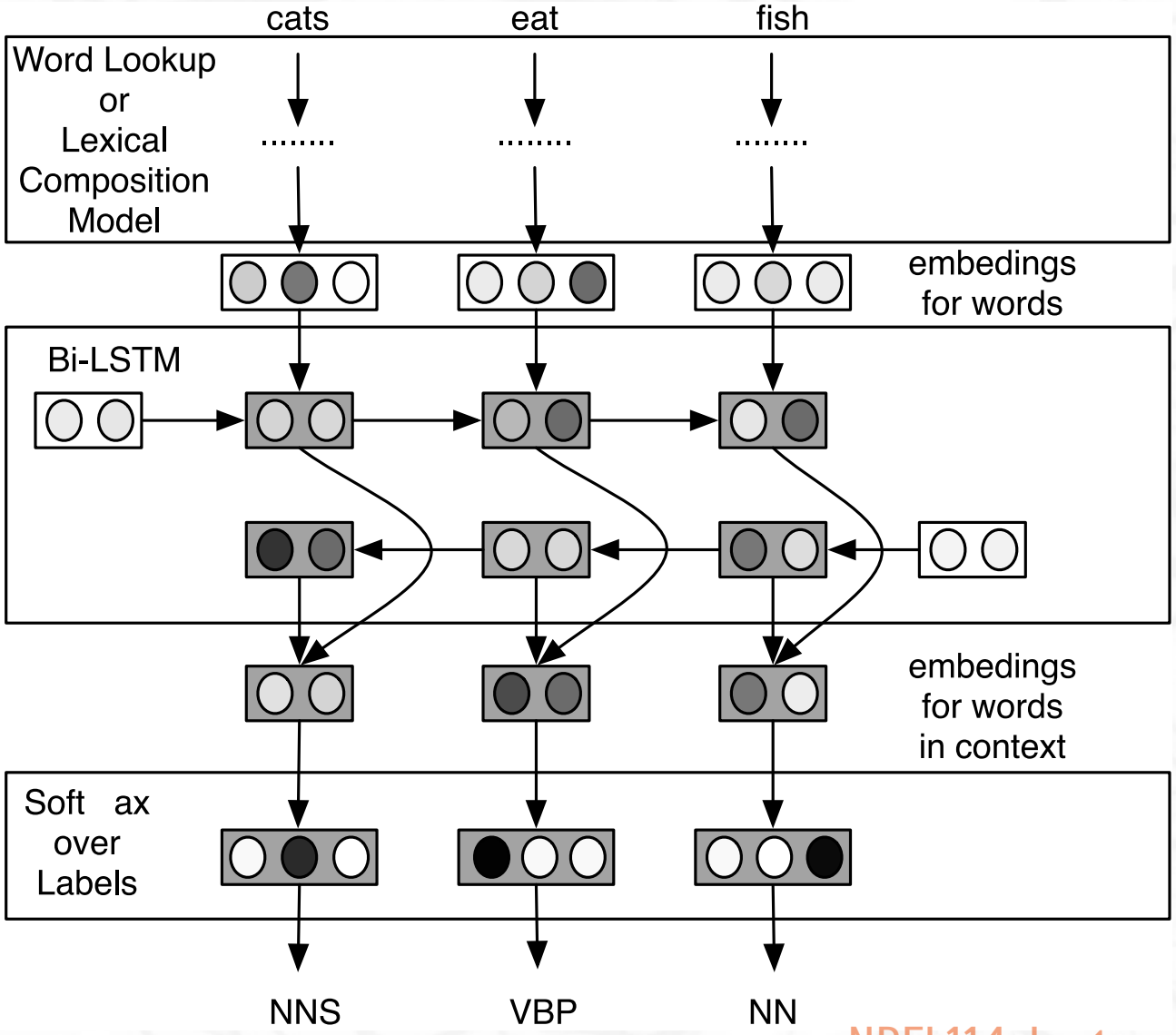
For $P(w)$, both uniform and unigram distribution $U(w)$ work, but

$$U(w)^{3/4}$$

outperforms them significantly.

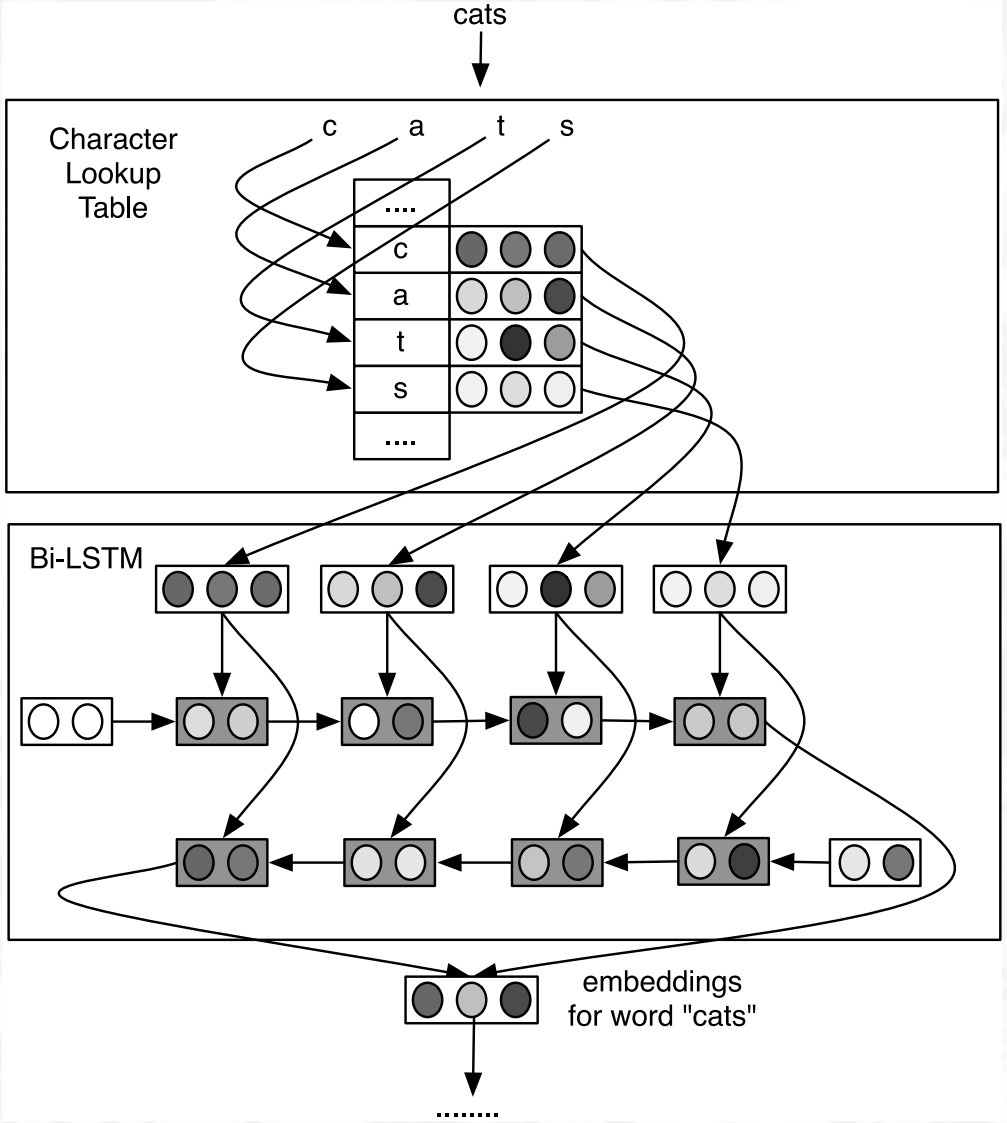
Basic NLP Processing

Bidirectional RNN



Word Embeddings for Unknown Words

Recurrent Character-level WEs

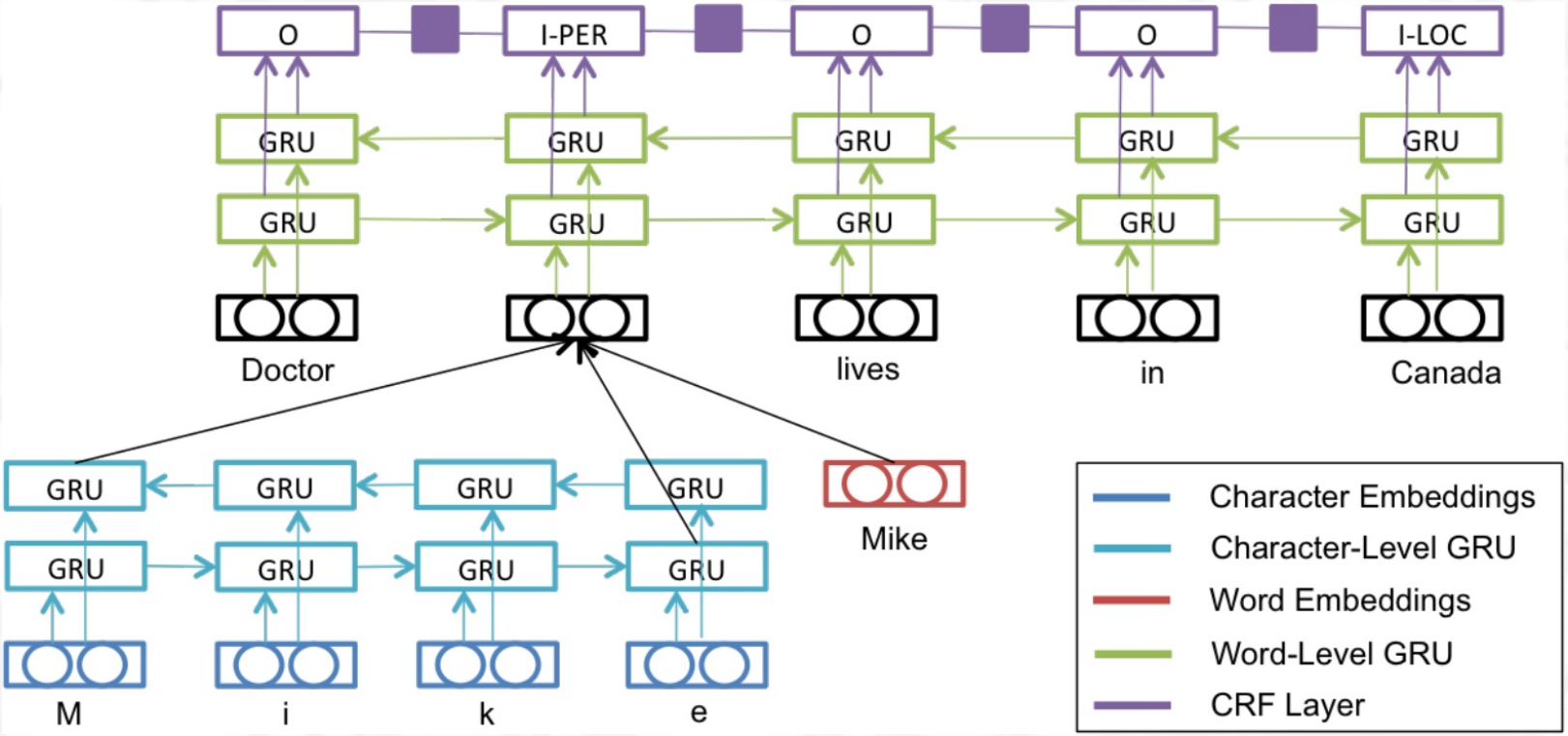


Recurrent Character-level WEs

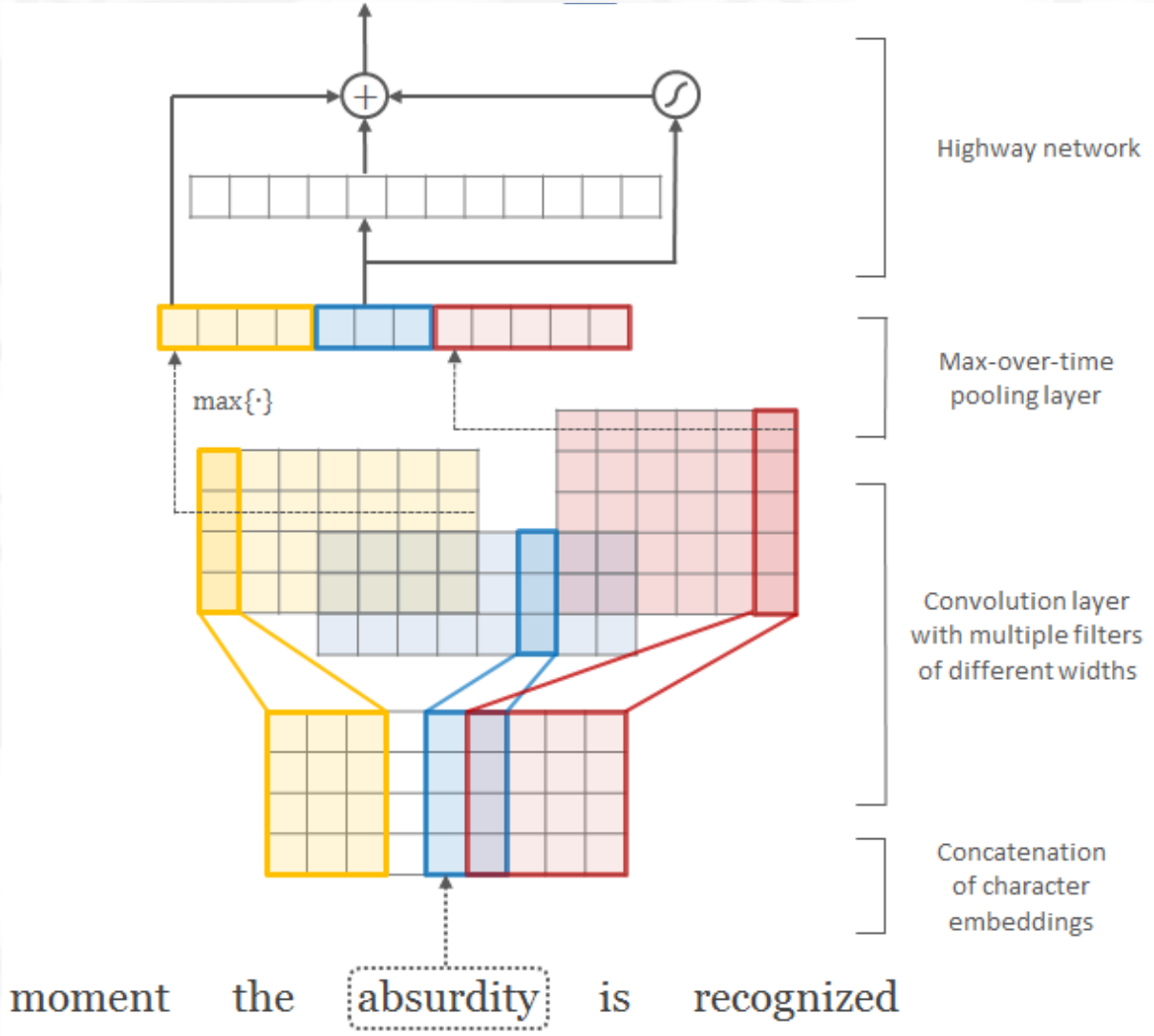
increased	*John*	*Noahshire*	*phding*
reduced	Richard	Nottinghamshire	mixing
improved	George	Bucharest	modelling
expected	James	Saxony	styling
decreased	Robert	Johannesburg	blaming
targeted	Edward	Gloucestershire	christening

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.

Recurrent Character-level WEs



Convolutional Character-level WEs



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.

Character N-grams

Another simple idea appeared simultaneously in three nearly simultaneous publications as Charagram, Subword Information or SubGram.

A word embedding is a sum of the word embedding plus embeddings of its character *n*-grams. Such embedding can be pretrained using same algorithms as `word2vec`.

Character N-grams

Another simple idea appeared simultaneously in three nearly simultaneous publications as Charagram, Subword Information or SubGram.

A word embedding is a sum of the word embedding plus embeddings of its character *n*-grams. Such embedding can be pretrained using same algorithms as `word2vec`.

The implementation can be

- dictionary based: only some number of frequent character *n*-grams is kept
- hash-based: character *n*-grams are hashed into K buckets (usually $K \sim 10^6$ is used)

Charagram WEs

query	tiling	tech-rich	english-born	micromanaging	eateries	dendritic
`sisg`	tile flooring	tech-dominated tech-heavy	british-born polish-born	micromanage micromanaged	restaurants eaterie	dendrite dendrites
`sg`	bookcases built-ins	technology-heavy .ixic	most-capped ex-scotland	defang internalise	restaurants delis	epithelial p53

Table 7: Nearest neighbors of rare words using our representations and `skipgram`. These hand picked examples are for illustration.

Charagram WEs

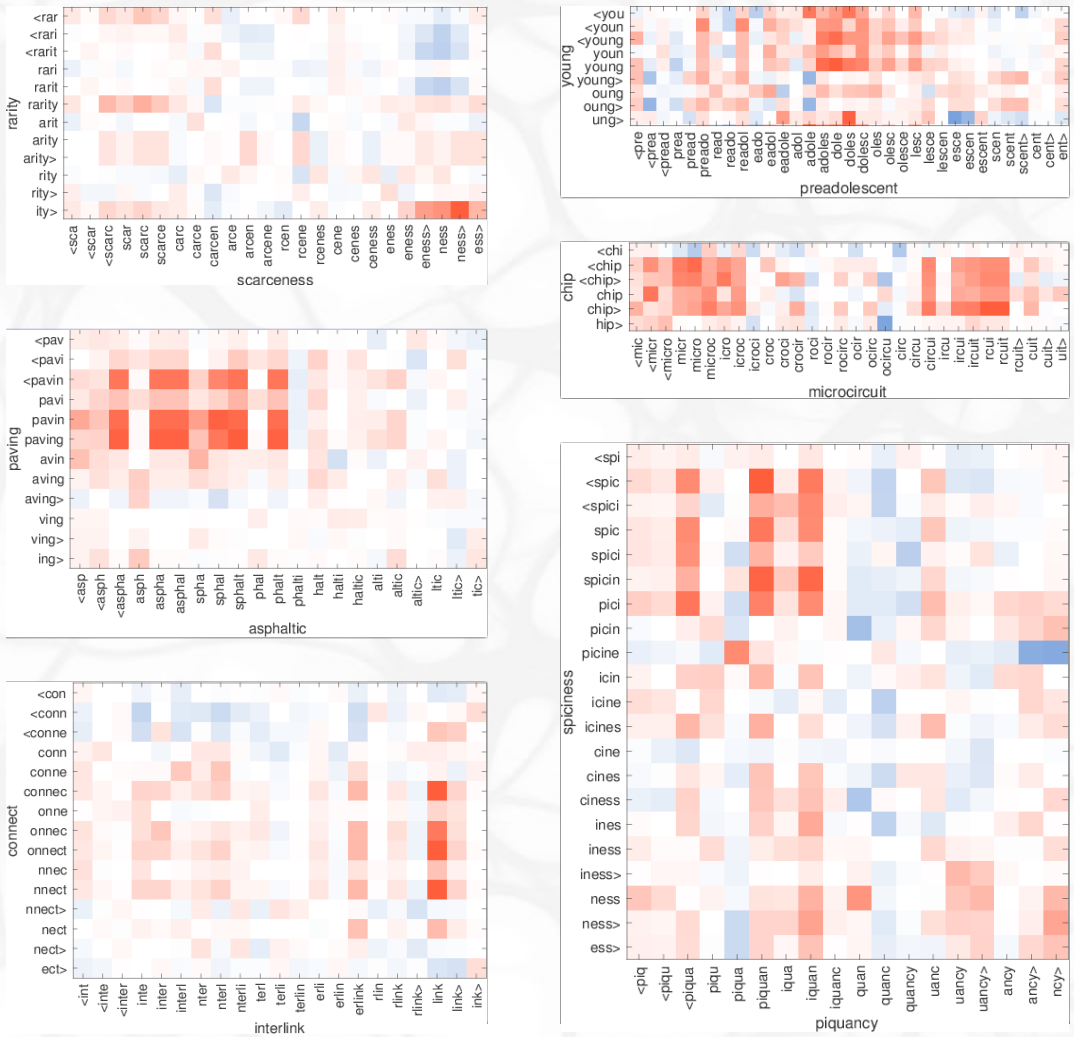


Figure 2: Illustration of the similarity between character n-grams in out-of-vocabulary words. For each pair, only one word is OOV, and is shown on the x axis. Red indicates positive cosine, while blue negative.

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.

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.

TF – RNN and Variable Length Sequences

tf.nn.dynamic_rnn

```
outputs, state = tf.nn.dynamic_rnn(cell,
                                   inputs,
                                   dtype=None,
                                   initial_state=None,
                                   sequence_length=None,
                                   ...)
```

TF – RNN and Variable Length Sequences

tf.nn.dynamic_rnn

```
outputs, state = tf.nn.dynamic_rnn(cell,
                                   inputs,
                                   dtype=None,
                                   initial_state=None,
                                   sequence_length=None,
                                   ...)
```

The `sequence_length` parameter allows sequences of different length. If used, the `outputs` past the sequence lengths are zeros and the `state` is the state after processing last element of each sequence.

TF – RNN and Variable Length Sequences

Losses

```
outputs, state = tf.nn.sparse_softmax_cross_entropy(
  labels,
  logits,
  weights=1.0,
  ...)
```

TF – RNN and Variable Length Sequences

Losses

```
outputs, state = tf.nn.sparse_softmax_cross_entropy(
  labels,
  logits,
  weights=1.0,
  ...)
```

The `weights` can be used to mask some of the labels and logits.

Generating the weights

```
tf.sequence_masks(lengths, maxlen=None, dtype=tf.bool)

weights = tf.sequence_masks(lengths, dtype=tf.float32)
```

TF – Bidirectional RNNs

tf.nn.bidirectional_dynamic_rnn

```
(outputs_fwd, outputs_bwd), (state_fwd, state_bwd) = \
  tf.nn.bidirectional_dynamic_rnn(cell_fwd,
                                  cell_bwd,
                                  inputs,
                                  dtype=None,
                                  initial_state_fw=None,
                                  initial_state_bw=None,
                                  sequence_length=None,
                                  time_major=False,
                                  ...)
```

Highway Networks

Highway Networks

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

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

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:

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

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

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

Highway Networks

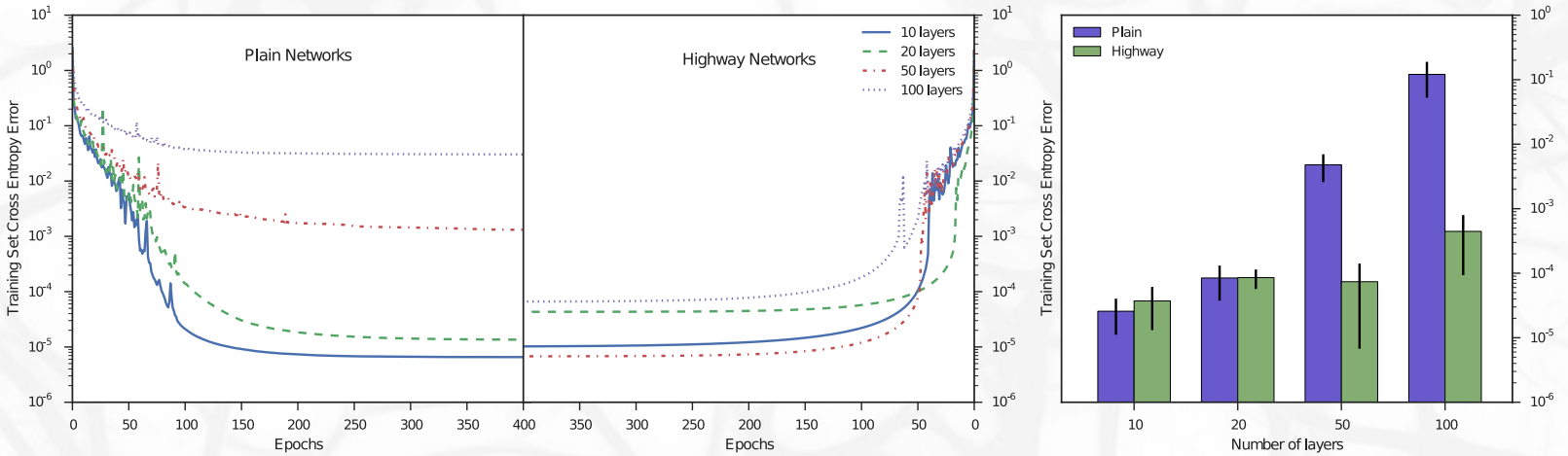
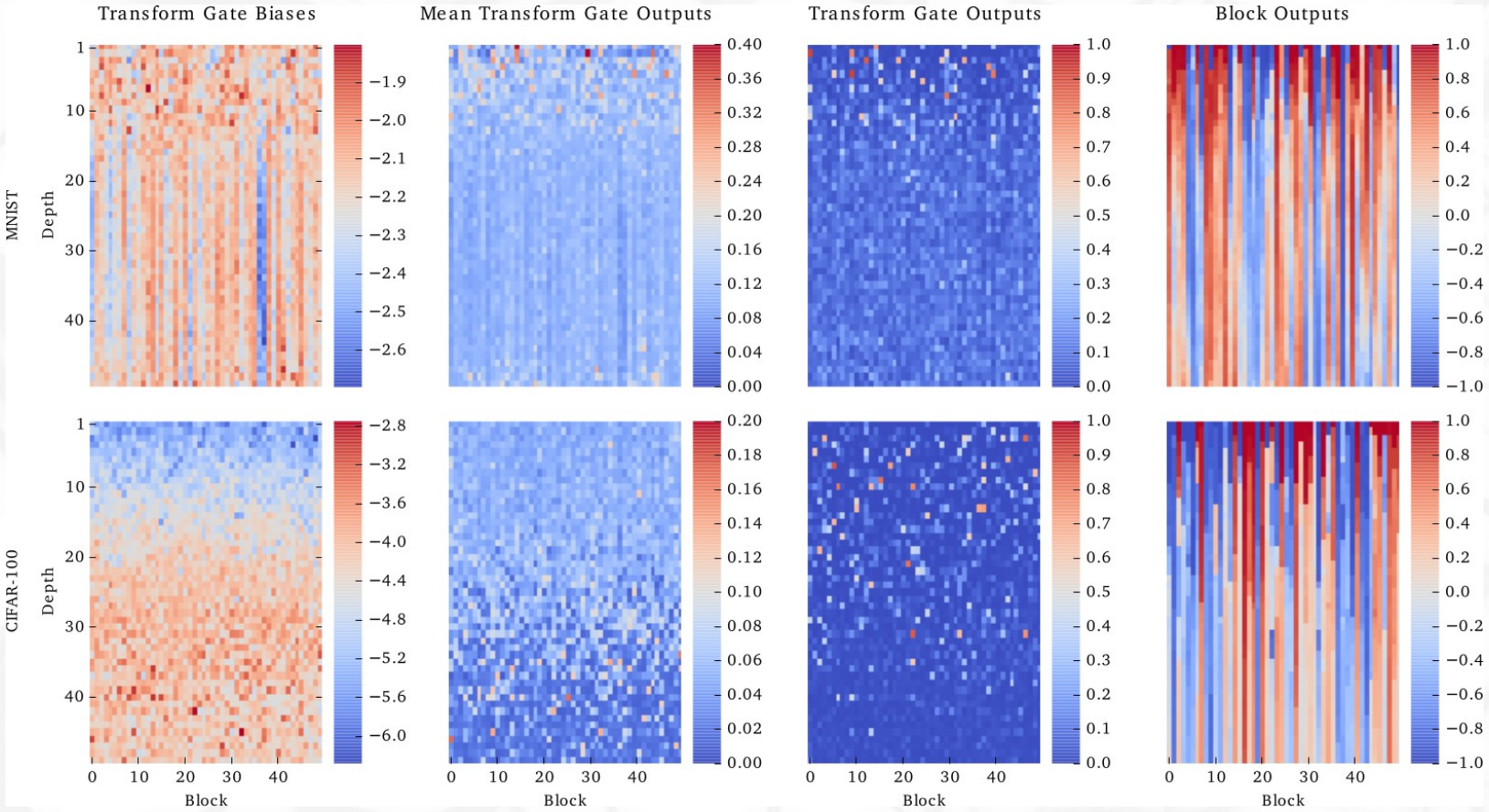


Figure 1: Comparison of optimization of plain networks and highway networks of various depths. *Left:* The training curves for the best hyperparameter settings obtained for each network depth. *Right:* Mean performance of top 10 (out of 100) hyperparameter settings. Plain n tworks become much harder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

Highway Networks



Highway Networks

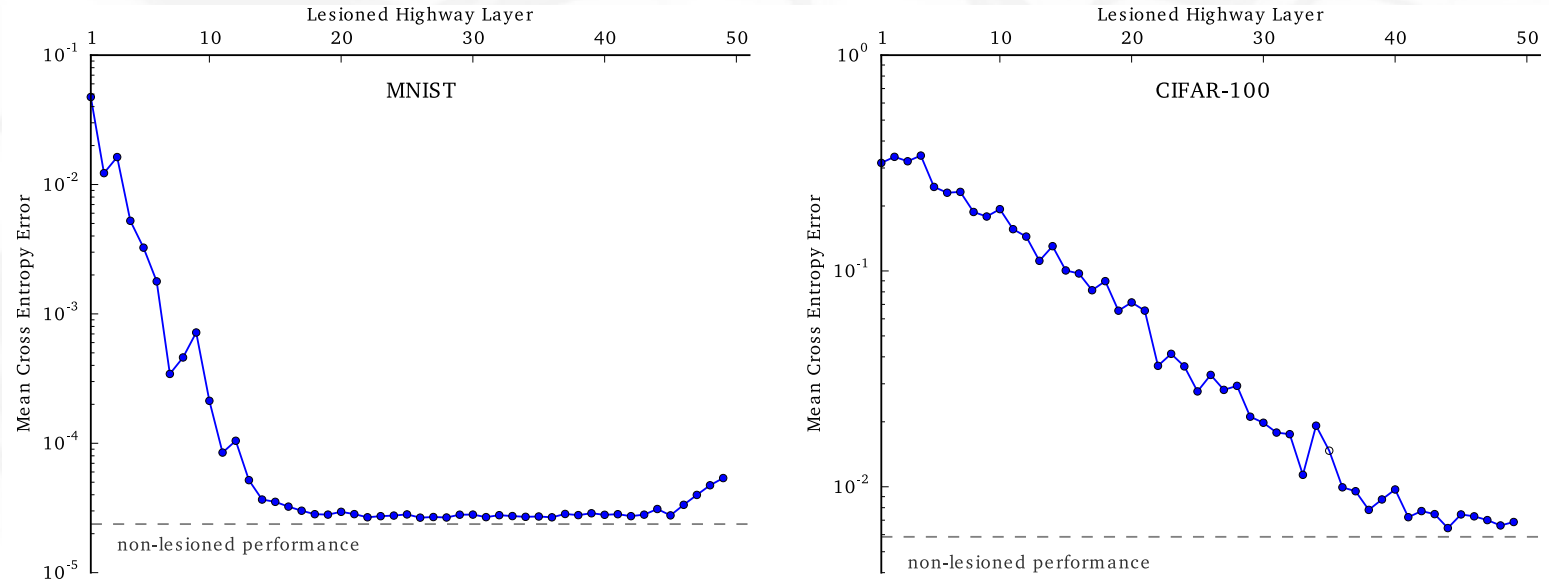


Figure 4: Lesioned training set performance (y-axis) of he best 50-layer highway networks on MNIST (left) and CIFAR-100 (right), as a function of the lesioned layer (x-axis). Evaluated on the full training set while forcefully closing all the transform gates of a single layer at a time. The non-lesioned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