State Gradients for Analyzing Memory in LSTM Language Models

Lyan Verwimp[a,*], Hugo Van hamme[a], Patrick Wambacq[a]

[a]*ESAT – PSI, KU Leuven, Kasteelpark Arenberg 10 - box 2441, 3001 Leuven, Belgium*

Abstract

Gradients can be used to train neural networks, but they can also be used to interpret them. We investigate how well the inputs of RNNs are remembered by their state by calculating 'state gradients', and applying SVD on the gradient matrix to reveal which directions in embedding space are remembered and to what extent. Our method can be applied to any RNN and reveals which properties in the embedding space influence the state space, without the need to know and label the properties beforehand. In this paper, we propose a normalization method that alleviates the influence of variance in embedding space on the state gradients and show the effectiveness of our method on a synthetic dataset. Additionally, the influence of several training settings on the RNN memory is investigated, and a more fine-grained analysis based on POS and word types shows that LSTM language models learn to model linguistic intuitions. Our code and datasets are publicly available.

Keywords: language modeling, neural networks, memory, gradients, interpreting

1. Introduction

Neural networks (NNs) are remarkably powerful models and have pushed the state of the art in several domains, including speech and language processing (e.g. language modeling [1, 2, 3], automatic speech recognition (ASR) [4, 5]). They typically consist of several large parameter matrices and non-linear functions, which transform the input of the network to an output. As a result, however, interpreting the reason why a specific output is predicted is not trivial. This is why NNs are often described as 'black-box' models. We usually have some intuitions about how they work, and proceed to propose new architectures, training regimes, data processing regimes . . . based on these intuitions. In this work, we take a step back and propose a method that can improve our understanding of how NNs, more specifically recurrent NNs (RNNs), work, which can potentially lead to better models in the future. Additionally, in certain scenarios such as diagnostic medical systems, interpretable models will be more easily accepted.

Our method is inspired by backpropagation, but instead of computing the gradients of the loss with respect to the parameters of the network, we compute the gradients of the hidden state of the network with respect to its current input or a previous input – the 'state gradients'. The gradient matrix can be decomposed with Singular Value Decomposition (SVD), resulting in singular values (SVs) that indicate the extent to which a specific direction in embedding space is preserved in state space. In this manner, we can investigate which properties that are encoded in the input embedding have an influence on the state, or in other words are remembered by the network, and to what extent. Our approach can by applied to any type of RNN[1], but in this work we apply it to Long Short-Term Memory (LSTM) [7] language models (LMs) [8].

*Corresponding author
Email address: `lyan.verwimp@kuleuven.be` (Lyan Verwimp)

[1]In this paper 'RNN' refers to all types of recurrent architectures, including LSTMs, whereas 'vanilla RNN' refers to the simple RNN or Elman network [6] as employed by among others Mikolov et al. [2].

The basics of our idea are explained in Verwimp et al. [9], while this paper presents the following additional contributions:

- We propose a normalization method for the state gradient matrix that makes sure that all directions in embedding space have equal variance (see Section 4.1). If the gradient matrix is not normalized, the singular values can be influenced by the fact that certain properties in embedding space require much larger steps than others.
- We demonstrate that the results of our approach are consistent with the predictions and the intermediate representations of an LSTM trained on a synthetic dataset (see Section 4.2).
- We investigate the influence of several training settings on the average memory of the model, namely different random initializations, a random versus a trained LSTM, different RNN cells (vanilla RNN, LSTM or Gated Recurrent Unit (GRU) [10]), different hidden state sizes, pre-trained embeddings vs training the embeddings from scratch, training on a different dataset and using a larger training set (see Section 4.3.2).
- We investigate whether LSTM LMs learn patterns that intuitively make sense from a linguistic point of view, by calculating the average memory for specific Part-of-Speech (POS) (Section 4.3.3) and word types (Section 4.3.4) and looking at how well specific POS are remembered by the RNN (Section 5).

Our code, that is based on TensorFlow [11], and the datasets that differ from publicly available corpora (Penn TreeBank with our own normalization, see Section 4.3.1, and the synthetic dataset used in Section 4.2) are available online: `https://github.com/lverwimp/state_gradients/`.

The remainder of this paper is structured as follows: we first present an overview of related work in Section 2. Next, we present the underlying concept of our approach in Section 3. In Section 4 we examine the average memory of LSTMs trained on synthetic and natural language data, while in Section 5 we examine how well specific properties are remembered by the LMs. We end with a conclusion and an outlook to future work in Section 6.

2. Related Work

Recently, the interest in gaining more insight in how NNs function has been rapidly growing. We make a distinction between several approaches to this topic.

Visualization. Dimensionality reduction techniques are commonly used to visualize embeddings in Natural Language Processing (NLP), e.g. Principal Component Analysis (e.g. in [12]), t-Distributed Stochastic Neighbor Embedding (t-SNE) [13] and bottleneck features [14]. Li et al. [15] use several techniques to visualize trained networks for sentiment analysis and sequence-to-sequence autoencoders: t-SNE visualization [13], a heatmap of the neuron activations over time and the variance of a specific word embedding with respect to an averaged word embedding. Strobelt et al. [16] release a tool that visualizes the evolution in LSTM hidden states for several text processing tasks, such as language modeling and biological sequence analysis. Karpathy et al. [17] visualize the activations of individual cells, tracking specific long-term dependencies aligned with the input text, for character-level LMs. They also plot gate activation statistics and perform error analysis. They show for example that an LSTM LM trained on Penn Treebank groups different types of phrases (noun/verb/adjective). Shi et al. [18] investigate why neural machine translation (MT) produces output sequences of the correct length seemingly effortlessly while statistical MT needs to model this explicitly. By visualizing activations and by training a classifier to predict string length based on the activations, they find that the length of the string is encoded by a specific subset of neurons. Ten Bosch and Boves [19] visualize the weights of Convolutional Neural Networks (CNNs) trained for keyword spotting, showing that they resemble spectro-temporal patterns and that increasing the number of convolutional matrices seems to lead to more specialized matrices. In this paper, we do not visualize embeddings or neuron activations but rather SVs that represent the extent to which the input embedding has an influence on the RNN state. Additionally, we visualize two measures of similarity between an embedding property and the embedding direction(s) with the largest influence on the state.

Methods based on derivatives/backpropagation. In computer vision, several backpropagation-based techniques for visualization and investigation of the inner workings of NNs have been proposed [20, 21]. For example, finding an image that maximizes the activation of a certain neuron can be done by optimizing using gradient ascent [20]. Simonyan et al. [21] propose two visualization techniques based on backpropagation, one which generates an image that maximizes the class score and one which visualizes the pixels in the image that are most informative for the prediction. This first-derivative saliency method is also applied by Li et al. [15], Aubakirova and Bansal [22] and Karlekar et al. [23]. These methods differ from our approach in the sense that they compute gradients of the output of the network with respect to the input, like in classical backpropagation, whereas we compute the gradients of the hidden state with respect to the input.

Rationale extraction. Another approach is trying to understand the reason why the network makes a certain prediction, by extracting the parts of the input that are important for predicting the output. A general framework to explain the predictions of a classifier is presented in [24], but it can only work with interpretable data representations such as binary vectors, whereas we present a framework that can deal with continuous (word) embeddings. Lei et al. [25] extract the parts of the input that are important for predicting the output, while Alvarez-Melis and Jaakkola [26] provide a general framework that can explain the predictions of models with structured input and output. Analyses can also be used to improve models, as Aubakirova and Bansal [22] demonstrate: they look at the activations of CNNs trained for politeness prediction and discover new features that can improve feature-based models. Related to rationale extraction, there is a line of work that tries to extract rules from a trained NN [27, 28, 29]. Our method does not provide ready-made explanations for certain predictions, but if for example a certain word has a large influence on the state a few timesteps later, we assume that that word plays an important role in the prediction of the LM.

Probing tasks. Embeddings can be evaluated by training logistic regression on them to predict a certain (linguistic) feature. For example, Adi et al. [30] test whether sentence embeddings contain information such as sentence length, word content and word order. A more extensive analysis of sentence embeddings is provided by Conneau et al. [31], involving different probing tasks and comparing different sentence embedding models. Shi et al. [32] investigate whether the encoder of a neural MT (NMT) system learns the syntax of the source sentence by training logistic regression on the sentence or word embeddings. Broere [33] investigates the syntactic properties of skip-thought sentence representations by predicting POS tags and dependency relations. Belinkov et al. [34, 35] extract word representations from NMT models and train classifiers to predict POS tags and morphological tags [34] or POS tags and semantic tags [35]. They report among others that character-based representations capture more morphology than word-based ones, that the lower layers of the network better capture syntactic structure while the higher layers better capture semantic information, and that mainly the encoder contains structural syntactic information. Blevins et al [36] investigate the syntactic properties of word representations extracted from several models (dependency parsing, semantic role labeling, MT and LMs) at several depths of the network. They train a feedforward Rectified Linear Unit (ReLU) classifier to predict the POS of the current word and its ancestors, and the presence or absence of a dependency arc between two words. They find that every model encodes syntax to some extent and that the optimal depth from which the representation is extracted depends on the type of model. Chrupała et al. [37] investigate whether the representations in a multi-modal model of speech and images encode features such as utterance length, the presence of a word and properties related to meaning and form, and find that the lower layers in the network are more specialized towards encoding formal properties, whereas the higher layers encode more semantic properties. For the same multi-modal model, Alishahi et al. [38] demonstrate based on several experiments that phonological information is best represented in the lower layers. Probing tasks have also shown that a small set of neurons encodes the length of a sequence [18, 39]. In a similar vein, we predict POS tags from word embeddings to verify whether specific classes are linearly separable in the embedding space. However, our probing experiment is not the main goal of this work but merely a sanity check to make sure that our definition of a property as a difference vector makes sense for that specific property (see Section 5 and Appendix 9).

Ablation studies. More insight in a specific model can be gained by removing/altering part of the model structure or the input data. Both approaches are used by Kuncoro et al. [40], who examine RNN grammars

(RNNGs): models that generate a parse tree and a sentence at the same time and that consist of three data structures: a stack of partially completed constituents, a buffer of generated terminal symbols and an action history list. Ablations to the RNNG itself involve removing one or two of the data structures to investigate which of them is most important, while ablations to the data involve removing non-terminal symbols to investigate the endocentric hypothesis (the function of a phrase is completely determined by its constituents). Hermans and Schrauwen [41] examine the influence of each layer in a character-level LM by setting its output to 0 and calculating the distance between hidden states after processing sequences that are identical except for one typo. Similarly, removing specific words from the input sentence and investigating the influence on the hidden state gives a measure of the importance of those words [42, 43]. Khandelwal et al. [44] go beyond the sentence level. They analyze the role of context in neural LMs by measuring the increase in perplexity when prior context words are dropped, shuffled or replaced. Their experiments show among others that on average the LM uses 200 context words (the 'effective context size'), and that there is a distinction between the nearby context, which is approximately 50 words, and the distant context. Target words seen in the nearby context can be readily copied by the LM, as opposed to the distant context that is modeled as a rough semantic representation – which is why a cache helps mostly for distant context words. The effective context size is larger for infrequent words and for content words. In Section 4.3.4, we demonstate that infrequent words have a larger influence on the state of the LSTM, which confirms the findings of Khandelwal et al. [44]. Zhu et al. [45] investigate semantic properties of several types of sentence embeddings by calculating cosine similarities for triplets of sentences. For example, the similarity of the original sentence with a sentence in which the verb is replaced by a synonym should be higher than with a negated version of the original sentence. Tüske et al. [46] limit the history of LSTM LMs and find that the performance plateaus at 40-grams for perplexity and 20-grams for ASR. Ablations to individual neurons have shown that grammatical number information in LMs is mainly encoded in only a few neurons [47]. In this work, we do not remove part of the model but alter the type of RNN cell or a specific hyperparameter and examine how this affects the average memory of the model.

Testing on special-purpose datasets. Instead of altering an existing corpus based on some simple rules as is done in ablation studies, new datasets can be created that can be used to test a specific feature. Early work by Gers and Schmidhuber [48] examines the ability of LSTMs to model context-free and context-sensitive languages by training on synthetic data. Weiss et al. [49] work along the same line by investigating the computational power of RNNs used in real applications. RNNs are Turing complete if they have infinite precision in the states and unbounded computation time, but in reality they have finite precision and computation time linear in the input length. Machines that can do unbounded counting are also able to recognize at least one context-free and one context-sensitive language. They show theoretically and empirically that LSTMs and vanilla RNNs with ReLU activation (that, however, easily have exploding gradients) can perform unbounded counting, while GRUs and vanilla RNNs with tanh activation cannot. More recently, Linzen et al. [50] and Bernardy and Lappin [51] investigate the ability of several models to learn subject – verb agreement by sampling sentences containing present tense verbs. Kuncoro et al. [52] work on the same dataset as Linzen et al. [50], comparing LSTM LMs with (several variants of) RNNGs. They find that LSTM LMs can predict number agreement quite well, but that RNNGs outperform them, probably because of their hierarchical structural bias. Gulordava et al. [53] construct a dataset that not only includes real corpus examples but also 'nonce' sentences, that are grammatically correct but semantically nonsense. They also focus on (long-distance) number agreement but include other cases than subject – verb and other languages than English. Just like Kuncoro et al. [52], but opposed to Linzen et al. [50] (whose LSTM possibly did not have optimal hyperparameters), they find that LSTM LMs have good performance on this task. Liu et al. [39] create several datasets that should contain the linguistic properties of natural language data to a specific extent: random sampling from a uniform distribution (no linguistic properties), sampling from a Zipfian distribution, sampling from a Zipfian distribution + Markovian dependencies between the words, and finally natural language. They show that the linguistic properties of the data have an impact on the memorization capability of LSTMs. LSTMs have been trained on a corpus of 'Dyck' language which is essentially a bracket completion task that should cover all context-free languages, to test their ability to learn recursion [54] and hierarchical structures [55]. In this paper, we create a synthetic dataset as a sanity check for our method (see Section 4.2).

Attention. For MT, it has been shown that NMT without attention fails on longer sentences [56]. Bahdanau et al. [57] introduce attention to overcome this problem and visualize the attention weights. Attention is also used and typically visualized in other NLP tasks, such as natural language inference [58] and speech recognition [59]. Kuncoro et al. [40] add attention to RNNGs to examine to what extent the linguistic rules of headedness are learned by the RNNG. In this paper, we do not make use of attention.

Examing the hidden representations. An important question in research about interpreting NNs involves what information is captured by the hidden representations, and whether different trained networks capture the same underlying representation. An information theoretic approach to understanding NNs is based on the Information Bottleneck principle [60], that models the training of NNs as finding the optimal tradeoff between the mutual information of the input and the hidden representation (compression, or discarding information in the input that is irrelevant) and the mutual information between the hidden representation and the output (prediction) [61, 62]. During training of multiple runs of the same network with Stochastic Gradient Descent (SGD), the same pattern emerged: the first, fast, phase consists of optimizing the network for prediction, whereas the second, much slower phase optimizes for compression. Deeper networks help because less epochs are needed for good generalization and because compression is more efficient if there are more layers. Ten Bosch and Boves [19] investigate the bottleneck representations in a phonetic autoencoder, showing that the representations of several trained networks can be 'morphed' into each other through a non-linear transformation. Throughout training the representations of phonetic classes become better separable, but this happens slower and less accurately for the more difficult classes such as nasals and liquids. Another approach to investigating the hidden representations is inspired by a neuroimaging method, the shared response model [63], that computes a mapping from the activation patterns of different trained networks to the same underlying shared representation [64]. In our work, we investigate the hidden states of the network by looking at which parts of the input embedding influence the state.

3. Singular Value Decomposition of State Gradients

We would like to know whether and how strong certain properties in embedding space are represented in the state space of the RNN. Given the fact that we can perform basic calculus operations on vectors, such as $\mathbf{v}_{king} - \mathbf{v}_{queen} = \mathbf{v}_{man} - \mathbf{v}_{woman}$, we assume that properties are represented as vectors in embedding space: we can find the embedding for e.g. *queen* by adding the *royal* vector to the *woman* vector. In Figure 1, we give an example of how the *gender* (P_1) and *royal* (P_2) properties could be represented in embedding space and in state space. When an input word is processed, the state vector will be modified, i.e. its word embedding will influence the state vector. If a word property, represented by an embedding component P is deemed important by the LM to predict future words, we expect the state vector to also shift by an amount P'. Irrelevant properties are not expected to affect the state. In the Figure, the length of P'_1 and P'_2 is non-zero, but it might very well be the case that those properties are not remembered at all by the RNN, resulting in zero-length vectors. The strength with which a specific direction is remembered by the RNN can be measured by calculating how a change of that direction in embedding space, $\Delta\mathbf{e}$, influences a change in state space, $\Delta\mathbf{s}$: the partial derivative of $\mathbf{s}$ with respect to $\mathbf{e}$, $\frac{\partial \mathbf{s}}{\partial \mathbf{e}}$.

We thus calculate the gradients of every component in the state $\mathbf{s}$ of the RNN with respect to every component in the input embedding $\mathbf{e}$ of timestep $t - \tau$, where t is the current timestep and τ the delay. For example, if the input sentence is *the cat lies on the mat* and we are currently processing the word *mat*, the input for a delay of 0 is *mat* itself, the input for a delay of 1 is *the* etc. For a specific timestep and specific delay, our gradients can be arranged in a gradient matrix $\mathbf{G}_{t,\tau} \in \mathbb{R}^{H \times E}$, where H is the size of the hidden state and E the size of the embedding . Averaging over all timesteps for a certain delay τ gives us the average gradient matrix for τ: $\bar{\mathbf{G}}_\tau$. This matrix hence contains the average influence of every component of the input embedding τ steps later on every component of the RNN state.

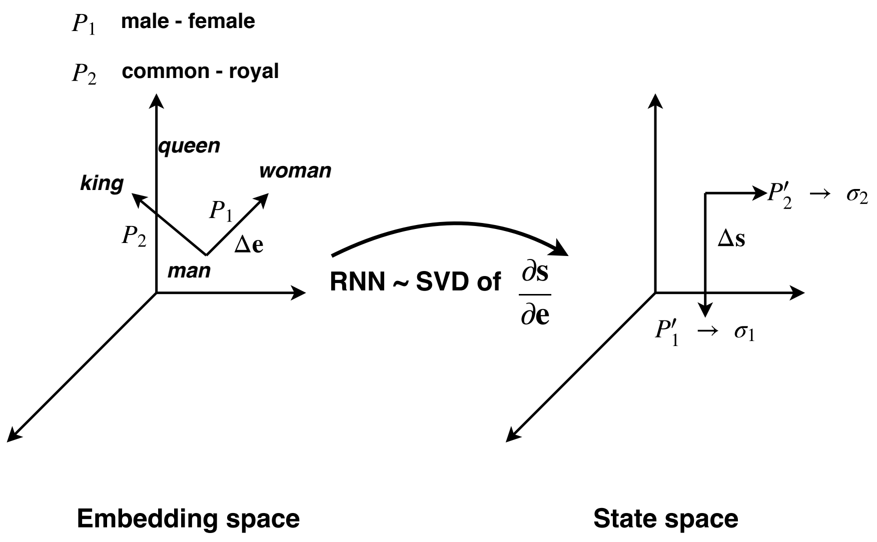


Figure 1: Example of how properties in embedding space are transformed to properties in state space by the RNN, and of how a step in state space $\Delta\mathbf{s}$ can be calculated based on the partial derivative of the state with respect to input embedding $\frac{\partial \mathbf{s}}{\partial \mathbf{e}}$ and the step in embedding space $\Delta\mathbf{e}$.

We decompose the average gradient matrix with reduced SVD:

$$\bar{\mathbf{G}}_\tau = \mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V}^T = \sigma_1\ \mathbf{u}_1\ \mathbf{v}_1^T + \sigma_2\ \mathbf{u}_2\ \mathbf{v}_2^T + \dots \tag{1}$$

in which $\mathbf{U}$ and $\mathbf{V}$ are matrices with orthonormal columns $\mathbf{u}_i$ and $\mathbf{v}_i$ respectively and $\mathbf{\Sigma}$ is a square diagonal matrix with the singular values σ_i. We can interpret $\mathbf{V}$ as directions in the embedding space, $\mathbf{\Sigma}$ as the extent to which the directions in the embedding space can be found in the hidden state space and $\mathbf{U}$ as corresponding directions in the hidden state space. For example in Figure 1, the male – female property P_1 in embedding space corresponds to $\mathbf{v}_1$, the first column of $\mathbf{V}$. It has the largest SV (lowest index) σ_1, and is hence the property that has the largest influence on the state space, or in other words, it is the property that is best remembered by the RNN. Notice that we make a linear approximation of the non-linear function of the RNN. One of the strengths of RNN memory analysis through state gradient matrices is that the embedding representation P of the properties need not be known, nor the properties themselves. The SVD will reveal which components in the embedding space affect the future state and to what extent. Another approach to analyzing the retained properties of the RNN would be building classifiers for those properties from the state representation, as is done in probing tasks. However, this approach assumes that we already know what those properties are and that we have appropriately labeled data, while it is possible that the RNN remembers things we did not initially consider to be important, and/or that the properties are not easily captured by labels. Our approach on the other hand does not make any a-priori assumptions about the retention properties of the RNN, and we will only apply probing procedures to interpret the embedding and state space representations the SVD analysis has come up with (see Section 5).

4. Average Memory of the RNN

In Section 4.1, we first explain how we calculate the average memory of a specific model (Section 4.1) and then present the results on a synthetic dataset (Section 4.2) and natural language datasets (Section 4.3).

4.1. Concept

As discussed in Section 3, the directions with the largest SVs are directions in embedding space that are best remembered by the RNN. In order to investigate how well the RNN remembers on average, we can average $\mathbf{G}$ over the whole dataset and can track the SVs with respect to the delay τ. We can also compare the SVs based on gradient matrices averaged over specific classes of words only, or even over the occurrences of individual words (the 'tokens' corresponding to a single word 'type').

There is however an issue with using the gradient matrix as is. To explain this issue, let us look at what the input to the RNN LM looks like:

$$\mathbf{e}_t = \mathbf{E}\ \mathbf{x}_t \tag{2}$$

In Equation 2, $\mathbf{x}_t$ is a one-hot encoding of the current input word, $\mathbf{E}$ the embedding matrix and $\mathbf{e}_t$ the resulting embedding. This embedding is given to the RNN, which multiplies it with a weight matrix $\mathbf{W}$:

$$\mathbf{s}_t = f(\mathbf{W}\ \mathbf{e}_t + \mathbf{H}\ \mathbf{s}_{t-1} + \mathbf{b}) \tag{3}$$

where $\mathbf{s}$ is the state of the network, f a non-linearity and $\mathbf{b}$ a bias vector. In the case of an LSTM or GRU, different weights $\mathbf{W}$, $\mathbf{H}$ and $\mathbf{b}$ will be trained for every gate.

Since we train our embeddings from scratch together with the rest of the LM and use no regularization on their weights, there are no constraints on the embedding space either. As a result, a specific property might be encoded as a larger step in embedding space than another property, and might have a smaller gradient with respect to the state, even though that does not imply that this property is less important. Ideally, we have an embedding space with equal variance to prevent such effects. If that is the case, a larger gradient effectively means that the property is more important.

To achieve this, we apply normalization to make sure that all directions in embedding space have equal variance. We can add additional weights $\mathbf{Z} \in \mathbb{R}^{E \times E}$ in Equation 3 as follows:

$$\mathbf{s}_t = f(\mathbf{W}\ \mathbf{Z}^{-1}\ \mathbf{Z}\ \mathbf{e}_t + \mathbf{H}\ \mathbf{s}_{t-1} + \mathbf{b}) \tag{4}$$

if $\mathbf{Z}^{-1}\ \mathbf{Z} = \mathbf{I}$. In Equation 4, we obtain a new embedding space spanned by $\mathbf{g}$:

$$\mathbf{g}_t = \mathbf{Z}\ \mathbf{e}_t \tag{5}$$

Notice that the weights $\mathbf{W}$ are also scaled by $\mathbf{Z}^{-1}$. We can now choose a $\mathbf{Z}$ to make sure that the variance of $\mathbf{g}$ is equal in all dimensions, by demanding that the covariance matrix of $\mathbf{g}$ is equal to the identity matrix:

$$\mathrm{cov}_{\mathbf{g}} = \mathbf{I} = \frac{1}{V} \sum_{i=1}^{V} (\mathbf{Z}\ \mathbf{e}_i)\ (\mathbf{Z}\ \mathbf{e}_i)^T = \mathbf{Z}\ \frac{1}{V} \sum_{i=1}^{V} (\mathbf{e}_i \mathbf{e}_i^T)\ \mathbf{Z}^T = \mathbf{Z}\ \mathrm{cov}_{\mathbf{e}}\ \mathbf{Z}^T \tag{6}$$

on the condition that all $\mathbf{e}$ have been normalized to have zero mean (V is the number of words in the vocabulary). If we decompose $\mathrm{cov}_{\mathbf{e}}$ through Cholesky decomposition into $\mathbf{D}\ \mathbf{D}^T$, Equation 6 becomes equal to:

$$\mathbf{I} = \mathbf{Z}\ \mathbf{D}\ \mathbf{D}^T\ \mathbf{Z}^T \tag{7}$$

One of the solutions to Equation 7 can be found by demanding that $\mathbf{Z}\ \mathbf{D} = \mathbf{I}$. As a result, we find that $\mathbf{Z} = \mathbf{D}^{-1}$.

Thus, we have a solution for the matrix $\mathbf{Z}$ that maps the word embeddings to a space with equal variance[2]. The partial derivative of the state with respect to the normalized embeddings is:

$$\frac{\partial\ \mathbf{s}}{\partial\ \mathbf{g}} = \frac{\partial\ \mathbf{s}}{\partial\ \mathbf{e}}\ \frac{\partial\ \mathbf{e}}{\partial\ \mathbf{g}} \tag{8}$$

[2]We have found the condition number of $\mathbf{Z}$ (the ratio of the largest SV w.r.t. the smallest SV) to be in the range 1–43. This implies that there are no numerical issues in this normalization.

Following Equation 5, we find that

$$\frac{\partial \mathbf{e}}{\partial \mathbf{g}} = \mathbf{Z}^{-1} = \mathbf{D} \tag{9}$$

This means that our normalized gradient matrix can be calculated as follows:

$$\mathbf{G}^{norm} = \mathbf{G}\ \mathbf{D} \tag{10}$$

In the remainder of this paper, we will always use the normalized gradient matrix, as opposed to [9], unless specified otherwise.

4.2. Experiments on synthetic data

In this Section, we first test our method on a synthetic dataset, in which there is a clear dependence between specific word pairs, and a clear independence between other pairs. For dependent word pairs, the input embedding of the first word should have a large influence on the state that is used to predict the second word, which should result in large SVs of the state gradient matrix. For independent word pairs, the gradients and subsequently the SVs should be significantly lower. We describe the synthetic dataset in Section 4.2.1 and the analysis in Section 4.2.2.

4.2.1. Setup

We generate the synthetic dataset in the following manner: we start by randomly drawing two tokens from the vocabulary (which contains 10 tokens), the first one is used only once and the second one 15 times. In the next segment, a new random token is drawn and repeated once, after which the first token of the previous segment is repeated 15 times:

A B B B B B B B B B B B B B B B C A A A A A A A A A A A A A A A D C C C C C C C
C C C C C C C C E D D D D D D D D D D D D D D D F E E E E . . .

We continue doing this until we have between 5 and 10 cycles, where a cycle is a combination of a single random token and the repetition of 15 tokens (the number of cycles is also randomly drawn from a uniform distribution), and then we generate an end-of-sentence token. In what follows, we refer to *token-A-1* as the first single occurrence of token A and to *token-A-2rep* as the sequence of 15 A's that can be predicted based on the occurrence of *token-A-1*. *token-<x>-{1/2rep}* refers to a general token/all instances of the specific pattern. Notice that *token-<x>-1* can never be predicted from the context, except by accident, while *token-<x>-2rep* always can, except for the very first occurrence (*token-B-2rep* in our example). The dependencies span across sentence boundaries, such that the *token-<x>-1* in the last cycle on a line will become *token-<x>-2rep* in the first cycle of the next line. The training set contains 16M tokens and the validation and test set each contain 160k tokens. We do not use an additional embedding layer, giving the one-hot vectors of the input tokens directly as input to the LSTM. The tokens in this dataset hence play the role of properties in natural language, which simplifies the analysis, since the input embeddings contain only one property. We train an LSTM with 1 hidden layer of 50 units with Adagrad [65] and a learning rate of 0.05. The perplexity (PPL) of this model is 1.2906/1.1932 for the validation/test set, which is close to the minimally possible PPL of 1.1375 (see Appendix 8).

4.2.2. Analysis

The LSTM needs to remember the first token of the previous cycle *token-<x>-1* in order to correctly predict it as the second element of the current cycle *token-<x>-2rep*. We expect large SVs for the embedding of this token with respect to all delays (at least) up to the timestep that the first of 15 repetitions should be predicted. Possibly, the SVs can remain large during *token-<x>-2rep*, but here the LSTM can rely on the elements of the repetition itself too, because once the first element of *token-<x>-2rep* is seen, the remaining ones can be predicted by copying the input token instead of relying on *token-<x>-1*. Additionally, the performance can be improved by counting how many tokens the LSTM has seen in *token-<y>-2rep*.

Table 1: Test accuracies (%) for LDA trained on the cell state and hidden state of the LSTM to predict the type of token ('Token ID'), the fact that the token is a new random token ('New token') or the count of the token ('Counting'). 'counting subpace'/'ID subspace' means that the cell states have been projected onto the 5-dimensional space which explains the majority of the variance for counting and its complement respectively (in the case of 'ID subspace'), found by the LDA trained on the original cell states to predict counting (first row, last column).

Representation	**Token ID**	**New token**	**Counting**
cell state	93	93	87
hidden state	89	93	80
cell state 'counting subspace'	11	93	86
cell state 'ID subpace'	93	93	17

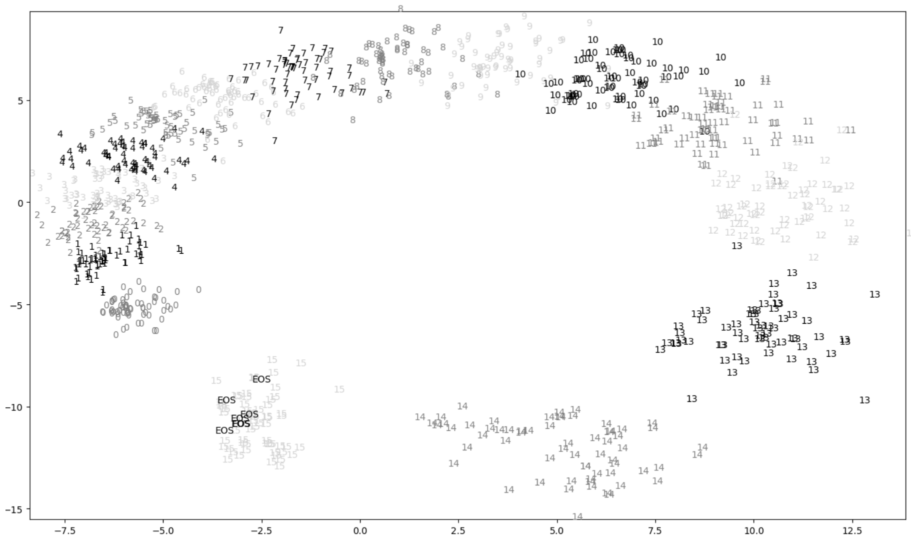


Figure 2: Cell states mapped to 2-dimensional space through LDA trained to predict the count of the input token associated with the cell state. '0' means the input token corresponds to *token-<x>-1*, '1' to the first element of *token-<y>-2rep*, and so on. 'EOS' corresponds to the end-of-sentence token.

If we take a closer look at the predictions of the LSTM, we see that it makes mostly 'expected' mistakes in predictions, i.e. when a new random token is chosen, the LSTM accordingly often predicts the wrong token. In only 0.009% (15 out of a total of 161,345) of all cases, the LSTM makes a mistake where we do not expect it – at positions other than a *token-<x>-1*. All of these 'unexpected' mistakes happen at the beginning of a new cycle, so when the LSTM sees *token-<y>-1* and should start predicting the first token of *token-<x>-2rep*. This means that in 0.009% of the cases, the LSTM is too slow to adapt to the beginning of a new cycle.

We examine whether the cell state and hidden state of the LSTM encode the properties that are needed to perform this synthetic task. To do this, we train Linear Discriminant Analysis (LDA) on the hidden representations recorded for the validation set, fitting a Gaussian density function to each token class under the assumption that all classes share the same covariance matrix. We deliberately kept the design of our synthetic dataset simple, expecting that the state space of the trained network will be interpretable. It should thus be possible to find clusters in the state space that can be described with a multivariate Gaussian, which is the reason why we chose LDA as classification method and not e.g. a multilayer neural network classifier which can give more complex decision boundaries. We reserve the last 10k tokens for testing the trained LDA parameters and use the rest (150k) for training. The results can be found in Table 1. Firstly, we try to predict the type of input token (10 target classes, see 'Token ID' in the Table) and we observe high

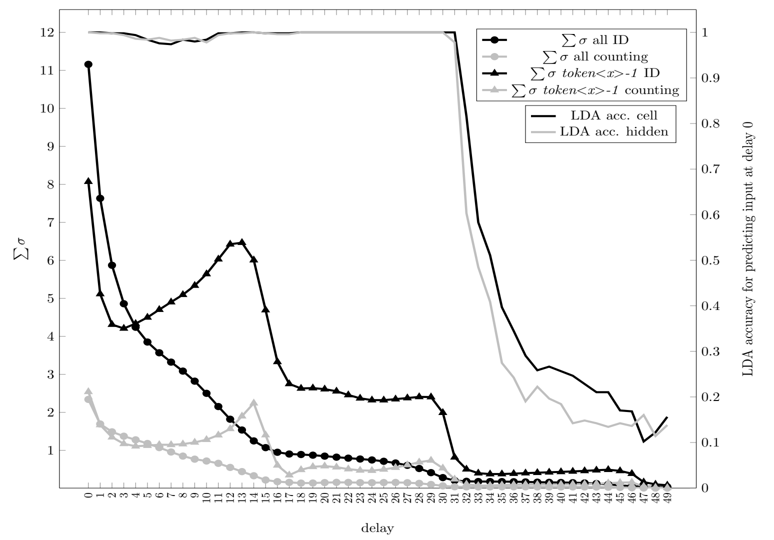


Figure 3: Left y-axis: $\sum \sigma$ of the gradient matrices of the cell state recorded for the validation set of the synthetic data, averaged over all tokens or over all positions where a new random token *token-<x>-1* is seen, w.r.t. consecutive tokens. 'counting' refers to the gradient matrix projected onto the 5-dimensional subspace that represents the counting information according to our LDA model, 'ID' refers to the gradient matrix projected onto the complement of this subspace (see Table 1). Right y-axis: average LDA accuracy for predicting the input at delay 0 based on the cell state or hidden state for several delays.

test accuracies for both the cell state and the hidden state. Secondly, we check whether the LSTM encodes the fact that a new random token is sampled ('New token'): we assign class label '1' to the new random tokens *token-<x>-1*, label 'EOS' to the special end-of-sentence tokens and '0' to all other tokens. Both representations encode this property with high accuracy. Finally, we check whether the counting behavior can be derived from the LSTM states: *token-<x>-1* receives label '0', the 15 tokens of *token-<y>-2rep* receive labels '1' until '15' and the end-of-sentence token receives label 'EOS'. The counting behavior can be derived with highest accuracy from the cell state, and we find that 5 dimensions in the 50-dimensional state are mostly responsible for this information (the eigenvalues of these dimensions cover 99%). This finding is in line with related work that shows that a small set of neurons can encode the length of a sequence [18, 39]. Mapping the cell states to the two most important LDA directions already gives reasonable accuracy, as the plot in Figure 2 shows: it can be seen that the counting is cyclical in nature, and that the classes become better separable when the end of the 15 repetitions is almost reached. Similarly, if we project the cell states onto all 5 counting dimensions instead of 2 dimensions as in Figure 2, and re-train LDA (cell state 'counting subspace' in the Table), we observe very low accuracy for token ID. On the other hand, projecting the cell state onto the 45 remaining dimensions ('ID subspace') gives high classification accuracy for token ID but low accuracy for counting. We can conclude that the LSTM is able to learn our synthetic task almost perfectly, and that it does this by remembering the identities and the count of the input tokens in different subspaces of the state space.

We now examine whether the dependencies between the new random token *token-<x>-1* and consecutive words can be detected in the SVs. In Figure 3, we plot the sum of the SVs $\sum \sigma$ of the gradient matrices averaged over all positions per delay ('all' in the Figure), and of the gradient matrices averaged over *token-<x>-1* per delay. We make a distinction between the gradients of directions associated with counting ('counting' in the Figure) and the gradients of the remaining directions ('ID' in the Figure), by projecting the gradient matrix onto the same subspaces as is done in Table 1. Additionally, we check how accurately the input at delay 0, if it is a new random token *token-<x>-1*, can be predicted based on the cell state or hidden state for a specific delay by training LDA. We observe that *token-<x>-1* can be accurately predicted until the end of *token-<x>-2rep* is reached, after which the LDA accuracy drops to just above chance level. This confirms our expectation that the identity of *token-<x>-1* is forgotten after *token-<x>-2rep*. The accuracy for the hidden state starts decreasing one timestep earlier, which is also in line with our expectations since the hidden state is used for prediction and there is no need to predict *token-<x>* when the last element of *token-<x>-2rep* is seen.

With respect to the SVs of the cell state, we see a decrease in $\sum \sigma$ for the gradient matrix averaged over all tokens, both for the gradients associated with counting and for those associated with token ID. The SVs averaged over the new random tokens *token-<x>-1* on the other hand, show distinct phases, decreasing at multiples of 16 steps, which reflects the fact that two data cycles of 16 tokens each need to be remembered to make accurate predictions. During the first phase, we observe a decrease followed by an increase of the SVs. The network seems to strengthen the influence of *token-<x>* when the moment that it should be predicted again, i.e. the start of *token-<x>-2rep*, draws closer. After step 32, only a minor influence of *token-<x>* is left, which agrees with classification rates dropping at that point – the fact that the influence of *token-<x>* does not disappear completely (especially its identity) agrees with classification rates slightly above chance level.

We conclude that for an LSTM that performs a synthetic task almost perfectly, large SVs for a specific input – state pair indicate that the input is still important at the timestep of the state, and decreasing SVs indicate decreasing importance of the input.

4.3. Experiments on natural language data

4.3.1. Setup

For our experiments on natural language data, our baseline LM has the following hyperparameters: it is an LSTM network containing an embedding layer of dimension 64 and 1 layer of 256 LSTM cells. Using a larger embedding size in combination with dropout [66] on the embeddings gives slightly better results, but we choose to not use dropout on the embeddings/input of the LSTM cell since we are interested in seeing what the model retains from the input, and masking the input might have unexpected effects on our analysis. The purpose of this work is not to train the best possible LM, but rather to investigate the inner workings of LMs with reasonable accuracy, and to examine the influence of changing a specific setting or hyperparameter on the inner workings. We do apply dropout on the output of the LSTM cells with a probability of 50%. The norm of the gradients is clipped at 5 for each mini-batch (during training, not while generating the state gradients). We train on mini-batches of size 20, each batch element containing a text sequence of 200 words. The network is initialized with random values drawn from a uniform distribution between -0.05 and 0.05. The LSTM is trained with SGD, starting with a learning rate of 1 for the first 6 epochs, after which it is exponentially decreased with a decay of 0.8. Notice that our baseline setup is the same as in [9], except for the fact that the length of our batch and subsequently the number of steps that the network is unrolled for (truncated) backpropagation through time is 200 instead of 50. Since we want to inspect the results for delays up to 50, training on batches of length 50 has the disadvantage that for a delay of 50 there are much less data points to calculate the gradients, since calculating gradients across batches is (as far as we know) not possible in TensorFlow. Hence our decision to train on batches of length 200 instead of 50.

We train the baseline LSTM LM on the widely used Penn TreeBank (PTB) benchmark [67], that contains 900k word tokens for training, 70k word tokens as validation set and 80k words as test set. The PPLs of three runs of the same network can be found in the first rows of Table 2, the second one will be used as baseline since it has best PPL. We chose PTB because it contains manually assigned POS tags that we will use to define specific syntactic properties (see Section 5 and Appendix 9), but 900k words is quite small. Therefore, we also train embeddings on Wall Street Journal (WSJ), which encompasses PTB and is hence in-domain data, but is much larger: we use the CSR LM-1 corpus (obtained from LDC) with non-verbalized punctuation (years 87–94) which contains 110M words. Embeddings trained on this dataset give better classification results than embeddings trained on PTB only [9] and are hence better linearly separable. Thus, we want to use pre-trained WSJ embeddings in our PTB LMs too. However, since the version of PTB that is commonly used for language modeling [2] is normalized differently than WSJ, the vocabularies of the two datasets do not match. If we want to use pre-trained WSJ embeddings, we need an embedding for every word in the PTB corpus. Hence, we chose to do some additional normalization on PTB and WSJ (see [9] for a description of the normalization). The WSJ embeddings are trained with word2vec [68] (cbow) with the default parameters. Using these pre-trained embeddings improves the PPL of the baseline model (see Table 2).

Table 2: Perplexities of the different LMs trained on PTB and WikiText.

Model	**Size hidden layer**	**Valid PPL**	**Test PPL**
PTB LSTM seed 1	256	107.6	104.9
PTB LSTM seed 2 (baseline)	256	103.3	103.8
PTB LSTM seed 3	256	106.4	104.2
PTB LSTM cbow WSJ	256	99.1	95.7
PTB GRU	256	110.4	108.2
PTB RNN	256	686.4	648.3
PTB LSTM	128	119.1	115.3
PTB LSTM	512	101.1	99.8
Wiki-2	256	126.2	118.0
Wiki-2	512	119.5	111.8
Wiki-103	512	43.4	42.1

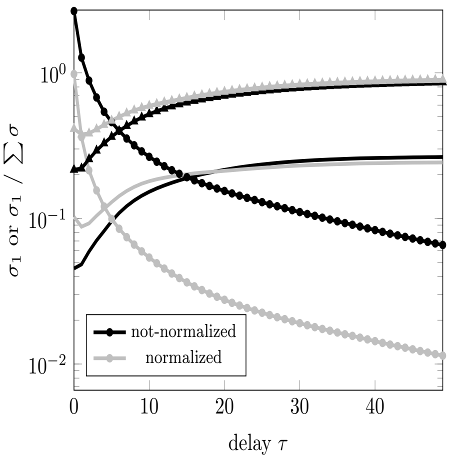
We compare the LSTM LM with a GRU [10] and a vanilla RNN with the same hyperparameters, which means that the size of the embedding and the hidden size are equal, but the number of trainable parameters will be larger for the LSTM than for the GRU, which in turn has more parameters than the vanilla RNN. We also compare with LSTMs with a smaller ('128h') and larger ('512h') hidden size. In Table 2, the validation and test PPLs for all models can be found. For each setting, we train three networks with a different random seed, and choose the network with the best results. Notice that even though we picked the best model out of three separate runs, the vanilla RNN still gives extremely bad PPL results. Moreover, the PPLs of the three runs were very different, showing the lack of convergence of the vanilla RNN.

We additionally train LMs on the WikiText dataset [69], which consists of WikiText-2 ('Wiki-2'), with a vocabulary of 33k words, 2M words for training, 200k words for validation and 240k words for testing, and the larger WikiText-103 ('Wiki-103'), which contains 100M words for training. Wiki-103 has the same validation and test sets as Wiki-2, which allows us to investigate the impact of the size of the training data on the memory. As opposed to previous work [69], we use the same vocabulary for Wiki-103 as for Wiki-2 to make sure that the only difference between the two datasets is their size.

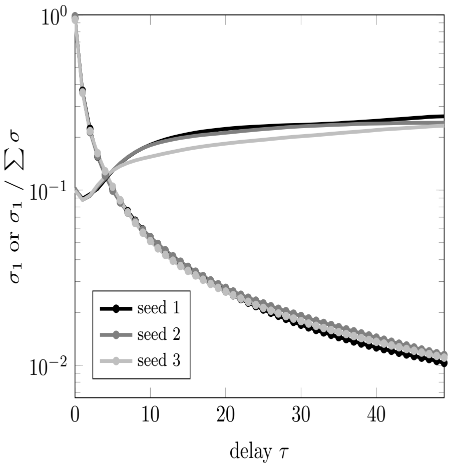
4.3.2. Analysis: Average over all words

In order to investigate the average memory of a LM, we calculate the gradient matrices for all words in the validation set for delays up to 50. The gradient matrices are averaged over all words for a specific delay, and SVD is applied to calculate the SVs. Unless specified otherwise, the 'state' that we are examining is the cell state or internal memory of the LSTM. We inspect the largest SV σ_1 (dotted lines in all plots) with respect to the delay as a measure of the influence of a word on the state after that delay. We observe similar trends for inspecting other SVs or the sum of all SVs. We also inspect the ratio of the largest SV with respect to the sum of all SVs $\sigma_1/\sum\sigma$ (solid lines in all plots), which gives an indication of how selective the model is: if the ratio becomes larger, the number of directions in embedding space that have a meaningful impact on the state becomes smaller. Notice that the y-axes of the plots have logarithmic scales.

Effect of normalization. We first examine the effect of normalizing the average gradient matrix. In Figure 4a, we plot σ_1 (dotted lines) with respect to the delay for $\bar{\mathbf{G}}$ and $\bar{\mathbf{G}}^{norm}$. The normalization introduces a scaling (and rotation which is not visible in the plots): we observe that the normalized values are lower. However, the qualitative trend is similar with and without normalization: there is a fast, but not exponential, decrease in σ_1. Thus, the influence of a word on the state decreases quickly when new words are fed to the network. We also plot the ratio of the largest SV (solid lines) and the sum of the 5 largest SVs (lines with triangles) with respect to the sum of all SVs. The scaling introduced by the normalization is not equally large in all directions, and as a result the difference between the normalized and non-normalized plots is smaller for the ratios of SVs than for the largest SV. We observe that in general the memory becomes more selective when



(a) Normalized vs non-normalized.



(b) Different random seed.

Figure 4: 4a: Compare memory for normalized (see Section 4.1) and not-normalized gradient matrices of the cell state of the baseline LSTM LM trained on PTB. The dotted lines indicate the largest SV σ_1 (average memory), the solid lines plot the ratio $\sigma_1 \ / \sum \sigma$ (selectiveness), the lines with triangles the ratio $\sum_{i=1}^{5} \sigma_i \ / \sum \sigma$ (selectiveness).
4b: Compare average memory and selectiveness for the same baseline model trained with a different seed.

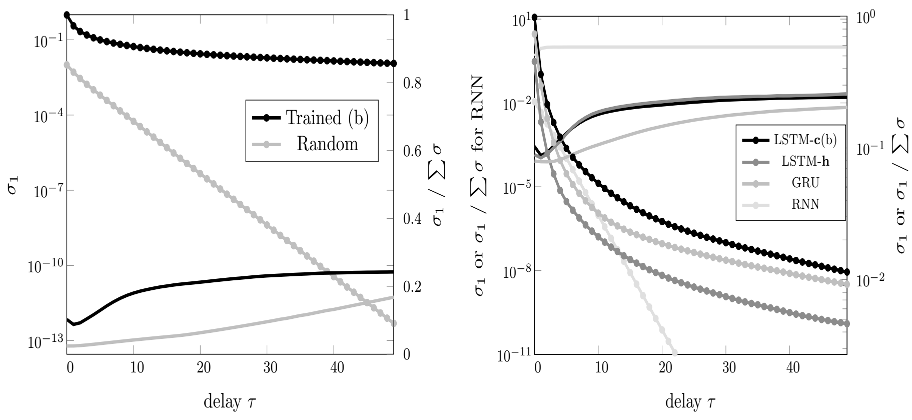
the delay increases. Notice that the value of $\sum_{i=1}^{5} \sigma_i \ / \sum \sigma$ is close to 1 for large delays, which indicates that what the LSTM LM remembers from words seen more than 40 timesteps ago is restricted to roughly 5 directions. In the experiments that follow, all gradient matrices have been normalized.

Effect of different initializations. In Figure 4b, we compare the results for the cell state $\mathbf{c}_t$ of the baseline LSTM LM trained with different random initializations. Both qualitatively and quantitatively, we observe no large difference between the three models. The influence of a word on the state quickly decays, but even after seeing more than 40 consecutive words, the LSTM state still contains some information about the word. The ratio of the largest SV with respect to the sum of all SVs increases when the delay increases, which shows that the memory becomes more selective over time: the LSTM remembers only one or a few of the properties from the word it has seen more than 20 timesteps ago. The baseline ('b') results shown in the subsequent experiments correspond to the best model in terms of PPL, 'seed 2'.

Effect of training. We are interested in how an LSTM with random weights would perform according to our metrics. In Figure 5a, we compare the trained baseline model with the same network with random weights. We observe that with random weights there is an exponential decay of σ_1, while training the LSTM as a LM results in larger SVs and a slower decay. The ratio $\sigma_1 / \sum \sigma$ (y-axis to the right) is also larger for the trained model, which implies that even though an LSTM LM has learned to remember important properties from its input embeddings, it has also learned to more quickly forget information – we hypothesize that it has learned that this information is irrelevant for prediction.

Effect of cell/state type. We now compare the cell state $\mathbf{c}_t$ of the LSTM with its hidden state $\mathbf{h}_t$ and the states of a GRU (y-axis to the right) and a vanilla RNN (y-axis to the left) in Figure 5b. The largest SV for the hidden state is smaller than for the cell state. This is not surprising given the fact that the hidden state is the result of multiplying the output gate with a tanh function on the cell state – the gradient of the tanh function is maximum 1 so it makes sense that the gradients (and the SVs) become smaller. The largest SV decays similarly, albeit slightly slower, for the GRU cell and lies between the cell state and the hidden state of the LSTM. The GRU state becomes selective more slowly, which seems plausible after closer inspection of how its state is calculated:

$$\mathbf{h}_t = \mathbf{z}_t \ \odot \ \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \ \odot \ \tanh(\mathbf{W} \ \mathbf{x}_t + \mathbf{H} \ (\mathbf{r}_t \ \mathbf{h}_{t-1}) + \mathbf{b}) \tag{11}$$



(a) Baseline PTB LSTM LM vs the same LSTM with random weights.

(b) LSTM hidden state, cell state, GRU and vanilla RNN of same size (PTB).

Figure 5: Compare memory: σ_1 = dotted lines (average memory) and $\sigma_1 / \sum \sigma$ = solid lines (selectiveness).

where $\odot$ is the element-wise product, $\mathbf{z}_t$ the output of the update gate and $\mathbf{r}_t$ the output of the reset gate. If the GRU wants to remember the new input/add something to its state (second element of the sum), it needs to forget something too (first element of the sum) since $\mathbf{z}_t$ is used as an interpolation weight. In an LSTM, there is no direct connection between remembering something new and having to forget something:

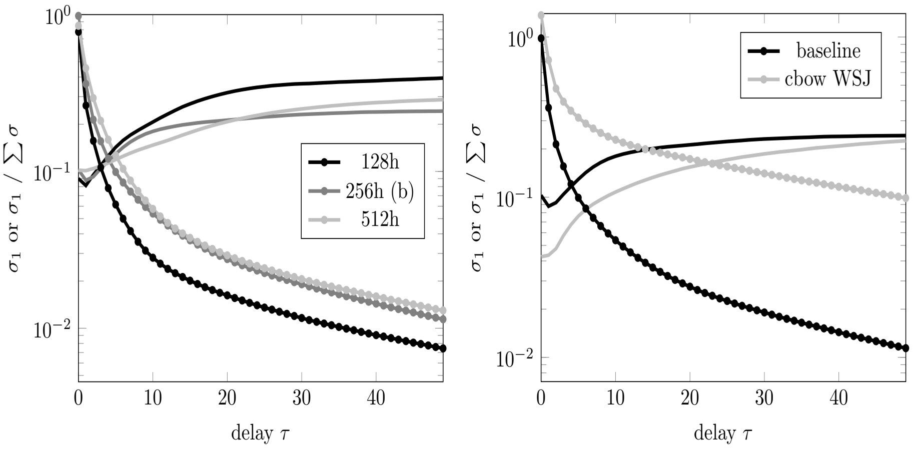
$$\begin{aligned}\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}\ \mathbf{x}_t + \mathbf{H}\ \mathbf{h}_{t-1} + \mathbf{b}) \\ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c_t})\end{aligned} \tag{12}$$

In the Equation above, $\mathbf{c}_t$ is the cell state, $\mathbf{h}_t$ the hidden state and $\mathbf{o}_t$, $\mathbf{f}_t$ and $\mathbf{i}_t$ the outputs of the output, forget and input gate respectively. The forget gate decides what should be forgotten from the previous cell state, while a different gate, the input gate, decides what should be added. We thus hypothesize that the LSTM cell state (and as a consequence, its hidden state) is more selective since it is more flexible.

For the vanilla RNN cell, σ_1 is much smaller and becomes virtually zero ($< 10^{-10}$) when the delay is larger than 20. Our results thus confirm that the memory of the LSTM is effectively longer than the memory of the vanilla RNN (see [70, 71] for a mathematical explanation of why this happens), and this has a considerable impact on the performance, as the results in Table 2 show.

Effect of hidden state size. Next, we would like to investigate the influence of the size of the hidden state on its capacity to remember properties from the input embeddings. We expect that increasing the hidden size will make the model less selective, since it has more representational power to encode different directions. In Figure 6a, we compare our baseline model with a model with smaller hidden size and larger hidden size. The small LSTM has lower values for σ_1 overall and its memory is more selective, which confirms our expectations. Given that its PPL is significantly worse than the baseline (see Table 2), these results give a plausible explanation why a hidden size of 128 is too small for this dataset: the LSTM is not capable of remembering relevant information long enough since its state cannot accommodate all relevant properties long enough. The selectiveness of the large model initially increases less quickly than for the baseline, but for long delays the large model is more selective and still shows an upward trend, as opposed to the baseline. This observation contradicts our expectation that a larger hidden state will automatically be less selective. Instead, we hypothesize that the large model can make more fine-grained distinctions in which properties should be remembered and which not, even at long delays. This results in slightly better performance, so we can conclude that there is no linear relationship between a more selective model and a better PPL.

Effect of pre-trained embeddings. Next, we compare our baseline model, in which the word embeddings are trained jointly with the rest of the model, with an LM with embeddings pre-trained on the much larger in-



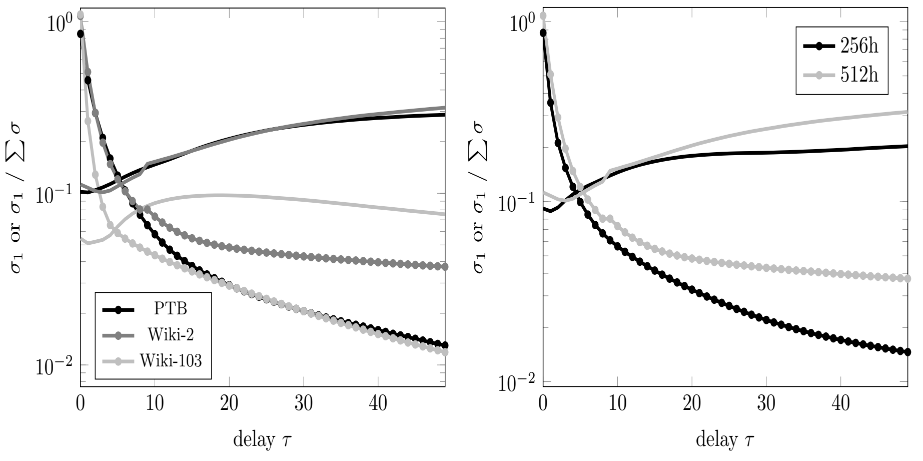
(a) Change hidden size for the baseline PTB LSTM LM.
(b) Pre-trained or jointly trained embeddings (= baseline PTB LSTM LM).

Figure 6: Compare memory: σ_1 = dotted lines (average memory) and $\sigma_1 / \sum \sigma$ = solid lines (selectiveness).

domain WSJ dataset[3]. In Figure 6b, it can be seen that σ_1 is larger for the LM with pre-trained embeddings and decreases more slowly, which means that the input embeddings have a larger and longer influence on the LSTM state. Given the fact that the input embeddings are meaningful from the very first training iteration as opposed to the baseline, in which the input embeddings are initially random and only gradually encode meaning throughout training, we assume that the LSTM LM learns to better make use of its input embeddings if they are pre-trained. The fact that the memory of the LM with pre-trained embeddings is less selective supports this hypothesis: plausibly, the LSTM can retain more useful information from the pre-trained embeddings for a longer time, and thus less quickly forgets that information. Notice that before normalization, the absolute values of σ_1 were larger for the baseline model without pre-trained embeddings: σ_1 for a delay of 0 is 2.66 for the baseline LM compared to 0.55 for the LM with pre-trained embeddings, which demonstrates the added value of normalizing the gradient matrices.

Effect of type and size of data. We now examine the influence of training (and validation) data on the average memory. In Figure 7a, we compare the PTB model with a hidden size of 512 with the same architecture trained on the small and large WikiText datasets. The WikiText datasets should contain more long-term dependencies since they consist of Wikipedia articles about the same topic, so we expect to observe a longer memory. While Wiki-2 and Wiki-103 share the same vocabulary, type of training data and validation and test sets, the main similarity between PTB and Wiki-2 as opposed to Wiki-103 is the size of the training data (1M and 2M as opposed to 100M words). Interestingly, the selectiveness plots of PTB and Wiki-2 are more similar than those of Wiki-2 and Wiki-103. The σ_1 plots of PTB and Wiki-2 are close for short delays, but start to diverge for delays longer than 10: the model trained on Wiki-2 has larger SVs, which confirms our expectation of a longer memory. σ_1 of Wiki-103 decays more quickly than for the models trained on much less data, while the selectiveness curve is considerably lower and stops increasing after a delay of about 10 words. The curve descends again after a delay of 10, which is caused by the fact that σ_1 decays more quickly during short delays than $\sum \sigma$ for this model. This seems to imply that upon seeing more data, the model learns to focus less strongly on only a few important directions on the long term.

[3] A similar comparison is made in [9], but the models in this paper differ in two aspects. Firstly, we fixed a preprocessing bug that produces two end-of-sentence symbols at the end of each sentence instead of one. Since predicting a second end-of-sentence symbol after the first one is an easy task, the PPLs reported in [9] are lower. The second difference is the fact that we train with batches of 200 words instead of 50 words (see Section 4.3.1 for the motivation of this choice). In the original setting (50 words per batch and double end-of-sentence symbols), the model with pre-trained embeddings has worse PPL (83.0 compared to 78.8 for the baseline), while in this setting the model with pre-trained embeddings has better PPL (95.7 compared to 103.8). However, the relative difference is due to the length of the batch and not due to the preprocessing bug, because the model with pre-trained embeddings is also worse if we train with the correct preprocessing and a batch size of 50 (101.3 compared to 95.9). Additionally, the results in this paper are for normalized gradient matrices.



(a) LSTM of size 512 trained on PTB, Wiki-2 and Wiki-103.

(b) Different hidden size for LSTM trained on Wiki-2.

Figure 7: Compare memory: σ_1 = dotted lines (average memory) and $\sigma_1/\sum\sigma$ = solid lines (selectiveness). The PTB LM in Figure 7a is the same model as in Figure 6a).

Effect of hidden size on WikiText. We also investigate the influence of increasing the hidden size on the WikiText dataset, since we expect that a larger hidden size and thus a better memory capacity will have a larger impact for WikiText than for PTB. In Figure 7b, we indeed observe a larger difference than in Figure 6a: the largest SV for the model with size 512 is consistently higher than the largest SV for the baseline model, even at long delays. We see a similar trend for the ratio σ_1 / $\sum\sigma$ as with PTB – the selectiveness of the larger model keeps on increasing while for the medium-sized model, the curve initially increases and is flat for long delays –, but the difference between the models is larger for WikiText.

4.3.3. Analysis: Average over specific POS

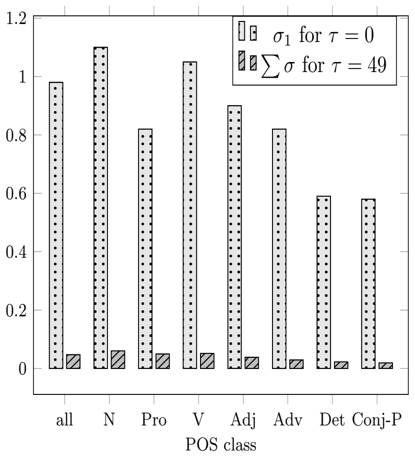
To investigate how long a specific class/POS is remembered on average, we average the gradient matrices of the baseline model per POS. In Figure 8a, we show the largest SV σ_1 for a delay of 0 and the sum of all SVs $\sum\sigma$ for the largest delay, namely 49, for several POS. Firstly, we observe that the relative relationships between the different classes remain the same across delays: the classes with the largest σ_1 at a delay of 0 still have the largest $\sum\sigma$ for a delay of 49. The classes that have the strongest impact on the state are nouns and verbs, which intuitively makes sense since these classes are the main pivots, both semantically and syntactically, of a sentence. The class of adjectives, that also carry some meaning but that will probably not be important on the long term, has the third largest σ_1, followed by adverbs and pronouns, that mainly carry semantic and syntactic meaning respectively. Conjunctions, prepositions and determiners have the smallest impact on the state, which also seems plausible given the fact that those classes mainly have consequences for short-term syntactic patterns while being relatively straightforward semantically.

4.3.4. Analysis: Average over all occurrences of a specific word

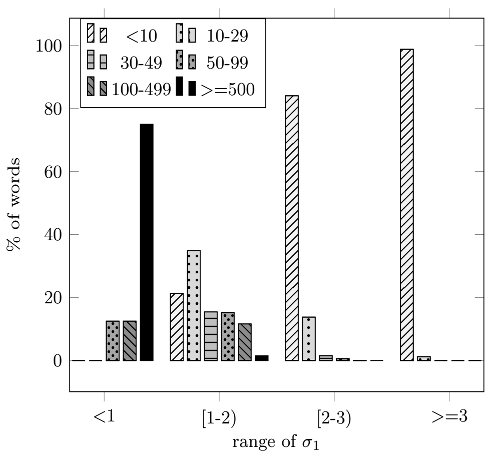
For a more fine-grained analysis, we now proceed to examine the average memory per word type. We average the gradient matrices over all occurrences of a word in the validation set and inspect its SVs.

Effect of frequency on σ_1. The frequency of a word in the training data has a large influence on the value of its largest SV, or in other words, how large its impact on the state of the LM is. In Figure 8b, we show the relationship between the binned value of σ_1 (x-axis) of the gradient matrix for delay 0 (we observe similar tendencies over all delays), and the percentage of words that have a certain binned frequency in the training set. For example, 99% of the infrequent words (frequency < 10) have a SV larger than or equal to 3, while 75% of the very frequent words (frequency $>= 500$) have a SV smaller than 1. It is clear from Figure 8b that frequent words tend to have low SVs, while infrequent words tend to have high SVs.

Frequent words often have mainly syntactic functionality, while infrequent words are mostly semantically relevant. If we take a look at the top 20 words with lowest SVs, these include determiners such as *an, a, the,*



(a) Average over POS.



(b) Average over word types.

Figure 8: 8a: average the gradient matrices for the baseline PTB LSTM LM over specific POS. 'N' = nouns, 'Pro' = pronouns, 'V' = verbs, 'Adj' = adjectives, 'Adv' = adverbs, 'Det' = determiners, 'Conj-P' = conjunctions and prepositions.
8b: distribution of word frequencies (labels) across the range of σ_1 values for the same baseline LM and a delay of 0.

the end-of-sentence symbol, prepositions such as *of, in, on, for, at, to*, frequent (auxiliary) verbs such as *is, was, has, would*, pronouns such as *one, it, its, them, any, that* and conjunctions such as *because, while, that.* The words with highest SVs generally have high semantic content, e.g. *reunification, earthquake, plaintiff, bizarre, turnpike.* This confirms the results for averaging over POS and again intuitively makes sense: words carrying much meaning should be remembered better since they are important clues for the subsequent words.

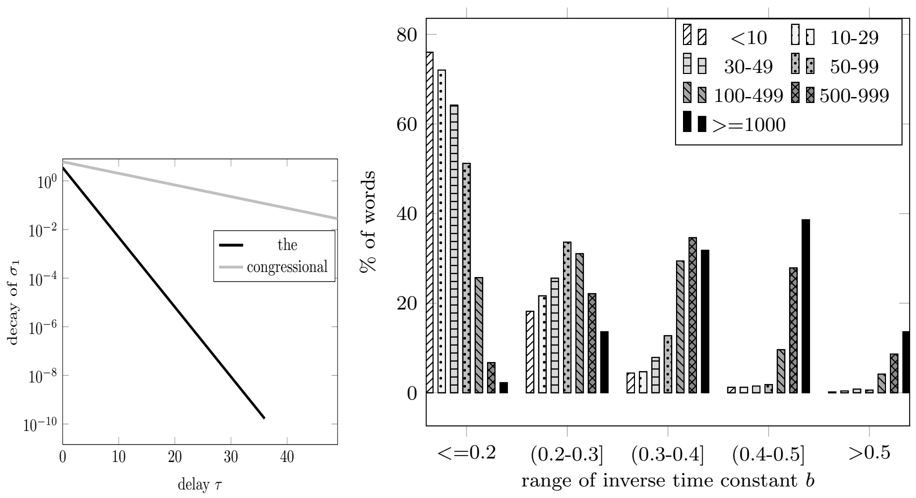
Effect of frequency on decay of σ_1. Additionally, one would expect that words with high semantic content would be remembered longer, while function words are typically only important on the short term. In other words, one would expect that the decay of the SVs would be stronger for frequent words. It has for example been shown that for combinations of LMs with cache models, which function as a sort of external memory for the LM, the cache model is not beneficial for function words and can receive very low weight in the interpolation [72]. To investigate whether this hypothesis is true, we try to fit an exponentially decaying function to the values of σ_1 for a specific word:

$$\sigma_1 = a * e^{-b*\tau} \tag{13}$$

In the Equation above, a and b are parameters for which we try to find optimal values. $1/b$ can be interpreted as the time constant, which indicates how fast the function decays – larger b means faster decay. The optimal values of the parameters are found with the help of non-linear least squares[4]. In Figure 9a, we give as example the results for the function word *the* and the content word *congressional.* It can be seen that the largest SV for *the* decays more quickly then for *congressional* (and becomes zero for delays of 37 and longer), which confirms our expectation.

For a quantitative analysis, we investigate the relationship between b and the frequency of the words. In Figure 9b, we plot on the x-axis the binned optimal b values of the words, and on the y-axis the percentage of words with a specific binned frequency (labels) that have this value for b. For example, 76% of the words with a frequency of less than 10 – which are typically content words – have a b value that is lower than or equal to 0.2, which corresponds to a slow decay. On the other hand, high-frequency (function) words have mostly high values for b. Thus, we can conclude that frequent words typically have a smaller impact on the state of the LSTM than infrequent words, and their impact also decays more quickly.

[4]This is implemented in `scipy.optimize.curve_fit`.



(a) Examples fit curves (eq. 13). (b) Relationship between frequency and b.

Figure 9: 9a: Optimal values for b in Equation 13 are 0.66 for *the* and 0.11 for *congressional*, based on gradient matrices averaged over all occurrences of those tokens, for the baseline PTB LSTM LM.
9b: distribution of the frequency range of the words (labels) with respect to the fit value of b/inverse time constant according to Equation 13 for the same baseline LM.

5. Tracking Specific Properties

In Section 5.1, we describe methods for tracking how well specific properties encoded in the input embedding are remembered by an RNN, which are applied to our baseline LSTM LM in Section 5.2.

5.1. Concept

If we want to know to what extent a specific property encoded in the input embedding is remembered in the RNN state, we will compare the vector encoding this property to the directions in $\mathbf{V}$ corresponding to the largest SVs of the average gradient matrix, for which we propose two methods.

We first define a specific syntactic or semantic property, e.g. the difference between singular ('sg') and plural ('pl') nouns, as the difference between the averaged embeddings for the classes separated by that property, normalized to unit length and normalized such that all directions in embedding space have equal variance:

$$\mathbf{d}_{a-b} = \mathbf{Z} \; \frac{\bar{\mathbf{e}}_a - \bar{\mathbf{e}}_b}{\| \; \bar{\mathbf{e}}_a - \bar{\mathbf{e}}_b \; \|} \tag{14}$$

where $\bar{\mathbf{e}}_a$ and $\bar{\mathbf{e}}_b$ are the result of averaging all embeddings of words belonging to classes a and b respectively and $\mathbf{Z}$ is the weight matrix as defined in Section 4.1. Before defining a specific property as the difference between the average embeddings, we will first check whether the embeddings of the two classes are linearly separable by training a linear classifier (see Appendix 9). We propose two methods to investigate the extent to which a property is remembered.

Compare with the n best remembered directions, without taking into account the strength with which they are remembered. The first possibility is comparing the difference vector $\mathbf{d}$ with the n best remembered directions. To compare $\mathbf{d}$ with a set of vectors, we first define $\mathcal{H}_n$ as the subspace of the embedding space spanned by the directions that are best remembered, the n largest right-singular vectors. Next, we make the orthogonal projection of $\mathbf{d}$ on $\mathcal{H}_n$. Since the n largest right-singular vectors are orthonormal, we can define the orthogonal projection as follows:

$$\mathbf{y} = \mathrm{proj}_{\mathcal{H}_n} \; \mathbf{d} = \mathbf{V}_n \; \mathbf{V}_n^T \; \mathbf{d} \tag{15}$$

where $\mathbf{V}_n$ is the matrix containing the n first columns of $\mathbf{V}$. The angle between $\mathcal{H}_n$ and $\mathbf{d}$ is equal to the angle between $\mathbf{y}$ and $\mathbf{d}$ and hence the cosine similarity is equal to:

$$\cos(\mathbf{d},\mathbf{y}) = \frac{\mathbf{d}^T\ \mathbf{V}_n\ \mathbf{V}_n^T\ \mathbf{d}}{\|\mathbf{d}\|\ \ \|\mathbf{y}\|} = \left\|\mathbf{V}_n^T\ \mathbf{d}\right\| \tag{16}$$

The cosine similarity between $\mathbf{d}$ and $\mathcal{H}_n$ is a measure of how close $\mathbf{d}$ is to the top n directions that have the largest influence on the RNN state: the closer to 1, the better the property specified by $\mathbf{d}$ is remembered if only n directions are effectively remembered.

Compare with all directions, taking into account the SV of the best remembered direction. A second option is comparing $\mathbf{d}$ with the direction in embedding space that is *best* remembered. Looking back at Figure 1, a measure of how well a direction in embedding space is remembered can be defined as follows:

$$\|\Delta\mathbf{s}\| \approx \left\|\frac{\partial\mathbf{s}}{\partial\mathbf{e}}\ \Delta\mathbf{e}\right\| \tag{17}$$

Since the partial derivatives are captured by the gradient matrix $\bar{\mathbf{G}}_\tau$, and $\Delta\mathbf{e}$ by $\mathbf{d}$, we can define r as a measure of the extent to which the difference vector is remembered by the state:

$$r = \left\|\bar{\mathbf{G}}_\tau \times \mathbf{d}\right\| \tag{18}$$

The result is a vector of the size of the state, in which each component captures the influence of the difference vector on that specific component of the state.

If $\mathbf{d}$ would be the embedding direction that has the largest influence on the state, then it would be equal to $\mathbf{v}_1$ and r would be equal to σ_1:

$$r = \left\|\bar{\mathbf{G}}_\tau \times \mathbf{v}_1\right\| = \left\|\sigma_1\ \mathbf{u}_1\ \mathbf{v}_1^T\ \mathbf{v}_1 + \sigma_2\ \mathbf{u}_2\ \mathbf{v}_2^T\ \mathbf{v}_1 + \ldots\right\| = \sigma_1 \tag{19}$$

because the columns of $\mathbf{V}$ are orthonormal. In order to get a relative measure of how well a specific direction is remembered, we compare r with σ_1 and obtain an 'extent to which the property is remembered, relative to the property that is best remembered', which we will henceforth refer to as the 'relative memory' m:

$$m = \frac{r}{\sigma_1} \tag{20}$$

Notice that the cosine similarity between $\mathbf{d}$ and $\mathcal{H}_n$ would be equal to the relative memory if the first n SVs have the same values, $\sigma_n = \sigma_1$, and if all other SVs are 0, $\sigma_{n+1} = 0$. In such a scenario, the gradient matrix would be equal to the following:

$$\bar{\mathbf{G}} = \mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V}^T = \mathbf{U}_n\ \mathbf{\Sigma}_n\ \mathbf{V}_n^T = \sigma_1\ \mathbf{U}_n\ \mathbf{I}\ \mathbf{V}_n^T = \sigma_1\ \mathbf{U}_n\ \mathbf{V}_n^T \tag{21}$$

Replacing $\bar{\mathbf{G}}$ in Equation 18 with the equality in Equation 21 gives the following result:

$$r = \left\|\bar{\mathbf{G}} \times \mathbf{d}\right\| = \left\|\sigma_1\ \mathbf{U}_n\ \mathbf{V}_n^T\ \mathbf{d}\right\| = \ \sigma_1\ \left\|\mathbf{U}_n\ \mathbf{V}_n^T\ \mathbf{d}\right\| = \sigma_1\ \left\|\mathbf{V}_n^T\ \mathbf{d}\right\| \tag{22}$$

Replacing the numerator of Equation 20 with the result of Equation 22, we see that the relative memory m equals the cosine similarity if $\sigma_n = \sigma_1$ and $\sigma_{n+1} = 0$.

$$m = \frac{\left\|\bar{\mathbf{G}} \times \mathbf{d}\right\|}{\sigma_1} = \frac{\sigma_1\ \left\|\mathbf{V}_n^T\ \mathbf{d}\right\|}{\sigma_1} = \left\|\mathbf{V}_n^T\ \mathbf{d}\right\| \tag{23}$$

Thus, both measures capture the extent to which a certain property is remembered, but the cosine similarity only compares $\mathbf{d}$ with the first n right-singular vectors while not taking into account the strength with which those directions are remembered, whereas the relative memory compares with all right-singular vectors but is a measure relative to the strength of the direction that is best remembered.

5.2. Experiments on natural language data

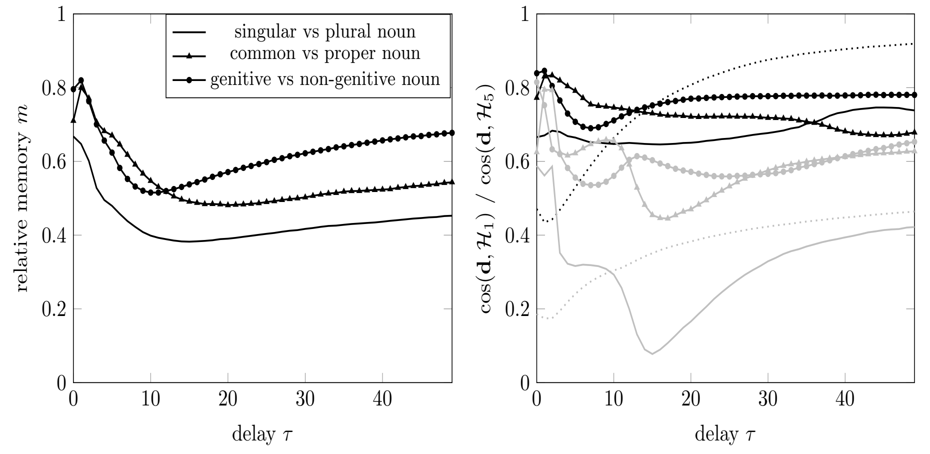
5.2.1. Setup

All experiments in this Section are carried out on the baseline LSTM LM trained on PTB, as described in Section 4.3.1 ('PTB LSTM seed 2' in Table 2). We will focus on syntactic properties that can be derived from the PTB POS tags, since this allows us to define difference vectors without manual labeling. Moreover, semantic properties are typically more ambiguous – a word might have a different meaning depending on the context in which it occurs, e.g. ***spring** is my favorite season* vs *it is **spring** tide* – and more difficult to capture as a difference vector since they might not be binary, e.g. positive vs negative: words such as *moderate* and *ok* are neither positive nor negative. The properties that are analyzed in this Section have been tested for linear separability (see Appendix 9).

5.2.2. Analysis

In Figure 10a, we show the relative memory for the difference vectors separating singular from plural nouns (e.g. *cat* vs *cats*), common from proper nouns (e.g. *cat* vs *adams*) and genitive nouns from non-genitive forms (e.g. *japan's* vs *japan*). We compare with the gradient matrix averaged over all nouns.

At a delay of 0, the genitive property is best remembered by the state with a relatively high m of almost 0.8. Its relative memory slightly increases at a delay of 1, followed by a gradual decrease at the medium-range delays and an increase for longer delays. The importance of the genitive property on the short term makes intuitively sense, since genitives have to be followed by a noun phrase and are thus an important syntactic signal. The difference between common and proper nouns is the second most important property, and becomes the most important property for the medium-range delays, which seems plausible since proper nouns mainly have semantic content. The difference vector of singular versus plural nouns has the lowest relative memory, and decays gradually if the delay increases – again, this intuitively makes sense since grammatical number has mainly an impact on the short term (e.g. subject – verb agreement).



(a) Relative memory m. (b) Cosine similarities.

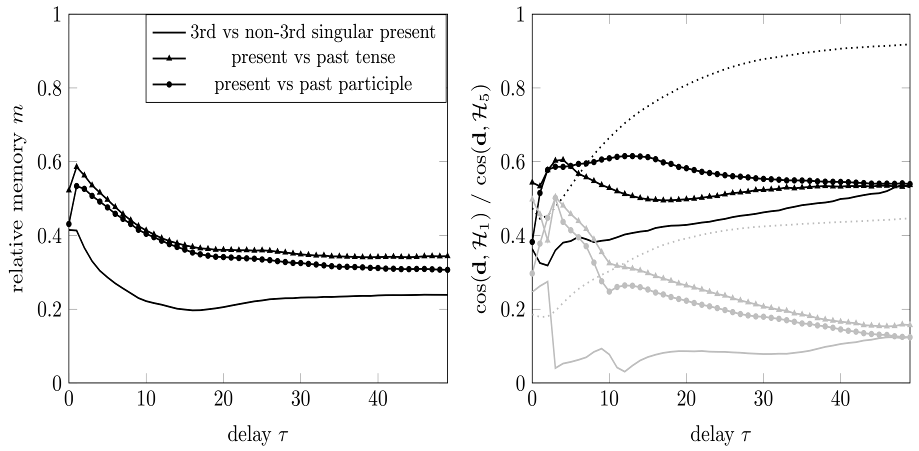
Figure 10: Memory of difference vectors separating different **noun** classes for the baseline PTB LSTM LM; gradient matrices are averaged over all nouns. Figure 10b uses the same legend as Figure 10a. For the cosine similarities, gray lines indicate the similarity of the difference vector with the first column of $\mathbf{V}$ and black lines with the 5 first columns of $\mathbf{V}$. The dotted lines are $\sigma_1 / \sum \sigma$ (gray) and $\sum_{i=1}^{5} \sigma_i / \sum \sigma$ (black).

We now compare the results for relative memory with those for cosine similarity between the difference vector and the subspace spanned by the top n most important directions. In Figure 10b, we plot the cosine similarity for $n = 5$ (black plots) and for $n = 1$ (gray plots), as well as the ratio of respectively the top 5 SVs (dotted black plot) and the largest SV (dotted gray plot) with respect to the sum of all SVs as a measure of the proportion of the total embedding space that we are comparing against. The ratio for the top 5 SVs becomes larger than 90% for long delays, which means that we are calculating the cosine similarity of an embedding vector with more than 90% of the original embedding space, so we need to be careful in drawing conclusions based on the cosine similarity for long delays. We compare the results for $\mathcal{H}_1$ and $\mathcal{H}_5$ because for $\mathcal{H}_1$, this ratio is smaller and because a large difference between these two measures for a specific property indicates that even though the property is not close to the best remembered direction, it does belong to the top 5 best remembered directions.

Looking at the cosine similarity with $\mathcal{H}_1$, we observe a similar pattern as for the relative memory: the genitive and common – proper directions have the highest similarity with the most important direction, each taking turns depending on the delay. For the grammatical number direction, we see a similar decay in similarity when the delay increases. The fact that the cosine similarity increases again after a delay of 16 seems to coincide with an increase in the ratio $\sigma_1 / \sum \sigma$. The curves for the cosine similarities with $\mathcal{H}_5$ are smoother, but the same relationships between the different properties apply.

Let us now look at the different properties of verbs: the difference between 3rd and non-3rd singular present (e.g. *eats* vs *eat*), between the present and past tense (e.g. *eat* vs *ate*), and between the present and past participle (e.g. *eating* vs *eaten*). In Figure 11a, we plot the relative memory for these difference vectors calculated based on the gradient matrix averaged over all occurrences of verbs.

Firstly, we observe that overall the m values are lower for these properties than for the nominal properties shown in Figure 10a. If we calculate the relative memory with respect to the gradient matrix averaged over all words in the validation set instead of over only the verbs or nouns, the nominal properties still have larger m values: the average σ_1 at delay 0 for the nominal properties is 0.72, while the average σ_1 for the verbal properties is 0.47. Given the fact that this also applies to m calculated for the POS-specific gradient matrices, this seems to imply that the LSTM LM pays more attention to the syntactic properties of nouns than to the syntactic properties of verbs. We hypothesize that this might be due to the word order in English, which is SVO – subject - verb - object. In order for the LM to predict the correct verbal conjugation, it needs to remember the syntactic properties of the subject. However, once the LM has seen the verb, it does not need to remember for example whether this verb was a third person conjugation or not in order to predict the object, which can explain why the difference vector 3rd – non-3rd singular present has the lowest relative memory. The difference between present and past tense/participle might be important to predict tense in subsequent sentences, so intuitively it makes sense that these properties are better remembered. The results for the cosine similarity are largely consistent with the results for the relative memory (see Figure 11b). Notice also that it is more difficult to classify the embeddings for verbs according to the syntactic properties that we are investigating, as opposed the properties of nouns (see Table 3).



(a) Relative memory m. (b) Cosine similarities.

Figure 11: Memory of difference vectors separating different **verb** classes for the baseline PTB LSTM LM; gradient matrices are averaged over all nouns. Figure 11b uses the same legend as Figure 11a. For the cosine similarities, gray lines indicate the similarity of the difference vector with the first column of $\mathbf{V}$ and black lines with the 5 first columns of $\mathbf{V}$. The dotted lines are $\sigma_1 / \sum \sigma$ (gray) and $\sum_{i=1}^{5} \sigma_i / \sum \sigma$ (black).

If we take a look at the syntactic properties of adjectives captured by the PTB POS tags, namely base form, comparative or superlative, we observe low relative memory and almost no difference between the different properties: 0.27 and 0.28 at a delay of 0 for the difference vectors base form vs comparative and base form vs superlative respectively. The cosine similarities show the same trend: 0.34 and 0.26 for similarity with the subspace spanned by the top 5 most important directions. This is slightly surprising since the presence of a comparative or superlative adjective can be a signal that respectively *than* and *of* will follow. However, if we look at the frequency of these patterns, we see that firstly comparative and superlative adjectives constitute only 0.6% and 0.4% of words in the training data, and only in 18% and 7% of the cases the adjectives are actually followed by the prepositions mentioned. Thus, it is very well possible that there is not enough training data for the LSTM to properly learn this distinction.

6. Conclusion and Future Work

We described a framework that is based on SVD of state gradients which can be used to analyze the influence of input embeddings on the state of RNNs, or in other words, how well properties encoded in the input embeddings are remembered by RNNs. The state gradients method allows to perform the RNN memory analysis of properties without knowing what those properties are exactly. This paper extends work described in [9], but adds a normalization method that ensures that all directions in embedding space have equal variance, since large variance differences might affect the gradients.

The method is tested on an LSTM trained on synthetic data, containing dependent word pairs and completely independent word pairs. We show that the dependent word pairs have large SVs, that the SVs decrease drastically when an input token does not need to be remembered anymore and that the intermediate representations of the LSTM effectively capture information that is needed to perform the task well.

We also apply our method to RNN LMs trained on natural language with different training settings. Firstly, we observe that it is robust against different random initializations. Secondly, an LSTM with random weights has exponentially decaying memory and a slow increase in memory selectivity, while training the LSTM as LM implies that the model learns to remember for a longer time, but also becomes more selective in what it remembers for longer delays. For GRU LMs, the impact of the input on the state is smaller than for the LSTM cell state, but larger than for its hidden state, while both LSTM states are more selective than the GRU state. We confirm a well-known property of vanilla RNNs, namely that their memory is in practice much more limited than that of GRUs and LSTMs. Increasing the hidden size results in less selective memory, but the relationship is not linear and depends on the dataset: increasing the hidden size from 128 to 256 for PTB improves results considerably and makes the memory much less selective. Increasing it further to 512 results in smaller gains for PTB than for WikiText. Using embeddings pre-trained on a large in-domain dataset results in a LM with longer and less selective memory. If we compare LMs trained on PTB and WikiText, we observe that in terms of selectiveness, the size of the dataset seems to have a larger impact than the type of data, whereas in terms of largest SV, the type of data is more important – models trained on WikiText have longer memory.

The results of applying a more-fine grained analysis are consistent with our expectations about how LMs function: frequent words, which typically are function words that are mainly important on the short term, have a smaller impact on the state and their impact decays more quickly. Nouns and verbs have on average the largest influence on the state, but the different syntactic properties of nouns are remembered better than the syntactic properties of verbs.

For future work, it would be interesting to examine whether errors made by the LM coincide with specific patterns in the state gradients and their SVs. Other interesting test cases include looking at models with several hidden layers, comparing the average memory for different languages, and applying our framework to other NLP tasks than language modeling.

7. Acknowledgements

This research is funded by the Flemish government agency IWT (project 130041, SCATE).

8. Appendix: Perplexity of Synthetic Dataset

The perplexity of seeing any cycle in context, e.g. $C\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A\ A$ in the example in Section 4.2.1, except for the very first cycle can be calculated as follows:

$$
\begin{aligned}
PPL(cycle|h) &= P(CAA\dots A)^{-\frac{1}{N}} \\
&= (P(C|h) * P(A|h) * P(A|h) * \dots * P(A|h))^{-\frac{1}{16}} \\
&= (\frac{1}{8} * 1 * 1 * \dots * 1)^{-\frac{1}{16}} = 1.1388
\end{aligned} \tag{24}
$$

where h denotes the entire history and corresponds to $A\ B\ B\ B\ B\ B\ B\ B\ B\ B\ B\ B\ B\ B\ B\ B$ for $P(C|h)$, $N = 16$ is the number of tokens in the cycle, $P(C|h) = \frac{1}{8}$ since there are 10 possible tokens and C cannot be equal to A or to B, and the probabilities of seeing every A are 1 since they can be predicted completely from the context.

The PPL of seeing a line with 5 cycles, l_5, can then be calculated as follows:

$$PPL(l_5) = ((\frac{1}{8})^5)^{-\frac{1}{5*16+1}} = (\frac{1}{8})^{-\frac{5}{81}} = \sqrt[81]{8^5} = 1.1369 \tag{25}$$

$N = 5 * 16 + 1$ because there are 5 cycles of 16 tokens and 1 end-of-sentence token, which has probability 1 since we know that it will follow after 5 cycles.

Since the length of each line is sampled from a uniform distribution in the range [5-10], the probability of seeing a line with length x is $\frac{1}{6}$ and the final PPL of the dataset is:

$$
\begin{aligned}
PPL = &\frac{1}{6} * PPL(l_5) + \frac{1}{6} * PPL(l_6) + \frac{1}{6} * PPL(l_7) \\
&+ \frac{1}{6} * PPL(l_8) + \frac{1}{6} * PPL(l_9) + \frac{1}{6} * PPL(l_{10}) = 1.1375
\end{aligned} \tag{26}
$$

9. Appendix: Linear Classification

Before investigating the extent to which a difference vector for a certain property is remembered in the cell state, we first check whether it makes sense to categorize a certain property as a difference vector by verifying whether the two classes are separable by a linear classifier.

The linear classifiers are trained with scikit-learn [73], with a grid search over several hyperparameters: regularization (L1, L2), regularization strength, initialization seed, frequency-based class weights or not, one vs all or multinomial loss and optimization algorithm.

In Table 3, we present the validation and test accuracies for logistic regression trained on the embeddings of the baseline LM to predict specific POS classes as defined in PTB (see [67] for a full description of the tag set).

The first part of the Table contains results for fine-grained distinctions. Trying to predict all POS tags jointly (first row) is clearly too difficult, probably because the number of classes is large (34) and because there are quite some infrequent classes. Distinguishing between the different sub-types for a class, e.g. sg (NN), pl (NNS), sg proper (NNP) and pl proper (NNPS) for the class of nouns, is usually easier. The second part of the Table contains results for more coarse-grained distinctions, e.g. 'N-V' is a classifier that tries to predict whether the embeddings belong to a noun or a verb. In general, we observe that the classifier achieves reasonable accuracies if the number of classes to separate is limited.

The final part of the Table are examples of specific properties that can be derived from the PTB POS tags. For the class of nouns, we add the distinction between genitive and non-genitive nouns (e.g. *cat's* vs *cat*), for which there are no POS tags but which can be found automatically (words ending on *'s* and their counterparts without *'s*). All nominal properties achieve high classification accuracies, while the verbal properties (next part of the Table) achieve slightly lower but still acceptable accuracies.

Table 3: Validation and test accuracies of logistic regression to predict POS-based classes. Numbers between brackets = number of target classes. N=nouns (tags NN, NNP, NNS, NNPS), V=verbs (tags VBG, VBD, VBN, VBP, VBZ), Adj=adjectives (tags JJ, JJR, JJS), Adv=adverbs (tags RB, RBR, RBS), Pro=pronouns (tags PRP, PRP$, WP, WP$). The first part of the Table makes a distinction between POS tags of the same word class (except for the first row), the second part of the Table between word classes, defined as the union of all POS tags of that word class (e.g. nouns = NN+NNP+NNS+NNPS), while the tags used for the last two parts are specified in the first column.

Classes	**Valid**	**Test**
all (34 tags)	42.0	40.8
nouns (4 tags)	70.8	72.0
verbs (6 tags)	37.9	39.6
adjectives (3 tags)	96.3	95.9
adverbs (3 tags)	100	94.4
N-V-Adj-Adv-Pro	67.8	66.7
N-V-Adj-Adv	67.9	66.9
N-V-Adj	69.3	68.0
N-V	78.2	77.2
N-Adj	84.2	81.1
N-Adv	94.6	93.4
V-Adj	82.4	80.7
V-Adv	94.6	94.2
Adj-Adv	84.4	86.9
sg-pl (NN+NNP - NNS+NNPS)	85.7	88.2
common-proper (NN+NNS - NNP+NNPS)	83.3	82.0
genitive-non-genitive (self-defined)	93.0	89.4
3rd - non-3d (VBP - VBZ)	70.5	71.5
present - past tense (VBP+VBZ - VBD)	76.4	80.0
present - past participle (VBG - VBN)	73.9	79.6

References

References

[1] Y. Bengio, R. Ducharme, P. Vincent, C. Jauvin, A Neural Probabilistic Language Model, Journal of Machine Learning Research 3 (2003) 1137–1155.

[2] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, S. Khudanpur, Recurrent neural network based language model, in: Proceedings Interspeech, 2010, pp. 1045–1048.

[3] R. Jozefowicz, W. Zaremba, I. Sutskever, An Empirical Exploration of Recurrent Network Architectures, in: Proceedings of the International Conference on Machine Learning (ICML), 2015, pp. 2342–2350.

[4] F. Seide, G. Li, D. Yu, Conversational Speech Transcription Using Context-Dependent Deep Neural Networks, in: Proceedings Interspeech, 2012, pp. 437–440.

[5] H. Sak, A. Senior, F. Beaufays, Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling, in: Proceedings Interspeech, 2014, pp. 338–342.

[6] J. L. Elman, Finding structure in time, Cognitive Science 14 (2) (1990) 179–211.

[7] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780.

[8] M. Sundermeyer, R. Schlüter, H. Ney, LSTM Neural Networks for Language Modeling, in: Proceedings Interspeech, 2012, pp. 1724–1734.

[9] L. Verwimp, H. Van hamme, V. Renkens, P. Wambacq, State Gradients for RNN Memory Analysis, in: Proceedings Interspeech, 2018, pp. 1467–1471.

[10] K. Cho, B. van Merriënboer, C. Gulcehre, F. Bougares, H. Schwenk, Y. Bengio, Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1724–1734.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, G. I. Andrew Harp, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng., TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, Software available from tensorflow.org.

[12] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed Representations of Words and Phrases and their Compositionality, in: Advances in Neural Information Processing Systems (NIPS), 2013, pp. 3111–3119.

[13] L. V. der Maaten, G. Hinton, Visualizing Data using t-SNE, Journal of Machine Learning Research 9 (2008) 2579–2605.

[14] Z. Bai, P. Weber, M. R. Peter Jančovič, Exploring how phone classification neural networks learn phonetic information by visualising and interpreting bottleneck features, in: Proceedings Interspeech, 2018, pp. 1472–1476.

[15] J. Li, X. Chen, E. Hovy, D. Jurafsky, Visualizing and Understanding Neural Models in NLP, in: Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL), 2016, pp. 681–691.

[16] H. Strobelt, S. Gehrmann, B. Huber, H. Pfister, A. M. Rush, LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks, IEEE Transactions on Visualization and Computer Graphics 24 (2018) 667–676.

[17] A. Karpathy, J. Johnson, F.-F. Li, Visualizing and Understanding Recurrent Networks, in: International Conference on Learning Representations (ICLR): Workshop track, 2016.

[18] X. Shi, K. Knight, D. Yure, Why Neural Translations are the Right Length, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2016, pp. 2278–2282.

[19] L. ten Bosch, L. Boves, Information encoding by deep neural networks: what can we learn?, in: Proceedings Interspeech, 2018, pp. 1457–1461.

[20] D. Erhan, Y. Bengio, A. Courville, P. Vincent, Visualizing Higher-Layer Features of a Deep Network, Tech. rep., Université de Montréal (2009).

[21] K. Simonyan, A. Vedaldi, A. Zisserman, Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, in: International Conference on Learning Representations (ICLR), 2014.

[22] M. Aubakirova, M. Bansal, Interpreting Neural Networks to Improve Politeness Comprehension, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2016, pp. 2035–2041.

[23] S. Karlekar, T. Niu, M. Bansal, Detecting Linguistic Characteristics of Alzheimer's Dementia by Interpreting Neural Models, in: Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL), 2018, pp. 701–707.

[24] M. T. Ribeiro, S. Singh, C. Guestrin, "Why Should I Trust You?": Explaining the Predictions of Any Classifier, in: Proceedings of the International Conference on Knowledge Discovery and Data Mining, 2016, pp. 1135–1144.

[25] T. Lei, R. Barzilay, T. Jaakkola, Rationalizing Neural Predictions, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2016, pp. 107–117.

[26] D. Alvarez-Melis, T. S. Jaakkola, A causal framework for explaining the predictions of black-box sequence-to-sequence models, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2017, pp. 412–421.

[27] G. Towell, J. W. Shavlik, Interpretation of Artificial Neural Networks: Mapping Knowledge-Based Neural Networks into Rules, in: Advances in Neural Information Processing Systems (NIPS), 1991, pp. 977–984.

[28] S. B. Thrun, Extracting Provably Correct Rules from Artificial Neural Networks, Tech. rep., University of Bonn (1993).

[29] M. Sushil, S. Šuster, W. Daelemans, Rule induction for global explanation of trained models, in: Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, 2018, pp. 82–97.

[30] Y. Adi, E. Kermany, Y. Belinkov, O. Lavi, Y. Goldberg, Fine-grained Analysis of Sentence Embeddings Using Auxiliary Prediction Tasks, in: International Conference on Learning Representations (ICLR), 2017.

[31] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, M. Baroni, What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Long Papers), 2018, pp. 2126–2136.

[32] X. Shi, I. Padhi, K. Knight, Does String-Based Neural MT Learn Source Syntax?, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), 2016, pp. 1526–1534.

[33] B. Broere, Syntactic properties of skip-thought vectors, Master's thesis, Tilburg University (2017).

[34] Y. Belinkov, N. Durrani, F. Dalvi, H. Sajjad, J. Glass, What do Neural Machine Translation Models Learn about Morphology?, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Long Papers), 2017, pp. 861–872.

[35] Y. Belinkov, L. Màrquez, H. Sajjad, N. Durrani, F. Dalvi, J. Glass, Evaluating layers of representation in neural machine translation on part-of-speech and semantic tagging tasks., in: Proceedings of the International Joint Conference on Natural Language Processing (IJCNLP) (Long Papers), 2017, pp. 1–10.

[36] T. Blevins, O. Levy, L. Zettlemoyer, Deep RNNs Encode Soft Hierarchical Syntax, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Short Papers), 2018, pp. 14–19.

[37] G. Chrupała, L. Gelderloos, A. Alishahi, Representations of language in a model of visually grounded speech signal, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL), 2017, pp. 613–622.

[38] A. Alishahi, M. Barking, G. Chrupała, Encoding of phonology in a recurrent neural model of grounded speech, in: Proceedings of the Conference on Computational Natural Language Learning (CoNLL), 2017, pp. 368–378.

[39] N. F. Liu, O. Levy, R. Schwartz, C. Tan, N. A. Smith, LSTMs Exploit Linguistic Attributes of Data, in: Proceedings of the Workshop on Representation Learning for NLP (RepL4NLP), 2018, pp. 180–186.

[40] A. Kuncoro, M. Ballesteros, L. Kong, C. Dyer, G. Neubig, N. A. Smith, What Do Recurrent Neural Network Grammars Learn About Syntax?, in: Proceedings of the European Chapter of the Association for Computational Linguistics (EACL) (Long Papers), 2017, pp. 1249–1258.

[41] M. Hermans, B. Schrauwen, Training and Analysing Deep Recurrent Neural Networks, in: Advances in Neural Information Processing Systems (NIPS), 2013, pp. 190–198.

[42] J. Li, W. Monroe, D. Jurafsky, Understanding Neural Networks through Representation Erasure, arXiv:1612.08220.

[43] A. Kádár, G. Chrupała, A. Alishahi, Representation of linguistic form and function in recurrent neural networks, Computational Linguistics 43 (4) (2017) 761–780.

[44] U. Khandelwal, H. He, P. Qi, D. Jurafsky, Sharp Nearby, Fuzzy Far Away: How Neural Language Models Use Context, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Long Papers), 2018, pp. 284–294.

[45] X. Zhu, T. Li, G. de Melo, Exploring Semantic Properties of Sentence Embeddings, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Short Papers), 2018, pp. 632–637.

[46] Z. Tüske, R. Schlüter, H. Ney, Investigation on LSTM Recurrent N-gram Language Models for Speech Recognition, in: Proceedings Interspeech, 2018, pp. 3358–3362.

[47] Y. Lakretz, G. Kruszewski, T. Desbordes, D. Hupkes, S. Dehaene, M. Baroni, The emergence of number and syntax units in LSTM language models, in: Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL), 2019.

[48] F. A. Gers, J. Schmidhuber, LSTM Recurrent Networks Learn Simple Context Free and Context Sensitive Languages, Transactions on Neural Networks 12 (6) (2001) 1333–1340.

[49] G. Weiss, Y. Goldberg, E. Yahav, On the Practical Computational Power of Finite Precision RNNs for Language Recognition, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Short Papers), 2018, pp. 740–745.

[50] T. Linzen, E. Dupoux, Y. Goldberg, Assessing the ability of LSTMs to learn syntax-sensitive dependencies, Transactions of the Association for Computational Linguistics 4 (2016) 521–535.

[51] J.-P. Bernardy, S. Lappin, Using Deep Neural Networks to Learn Syntactic Agreement, Linguistic Issues in Language Technology (LiLT) 15 (2).

[52] A. Kuncoro, C. Dyer, J. Hale, D. Yogatama, S. Clark, P. Blunsom, LSTMs Can Learn Syntax-Sensitive Dependencies Well, But Modeling Structure Makes Them Better, in: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (Long Papers), 2018, pp. 1426–1436.

[53] K. Gulordava, P. Bojanowski, E. Grave, T. Linzen, M. Baroni, Colorless green recurrent networks dream hierarchically, in: Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL), 2018, pp. 1195–1205.

[54] J.-P. Bernardy, Can Recurrent Neural Networks Learn Nested Recursion?, Linguistic Issues in Language Technology (LiLT) 16 (1).

[55] L. Sennhauser, R. C. Berwick, Evaluating the Ability of LSTMs to Learn Context-Free Grammars, in: Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, 2018, pp. 115–124.

[56] K. Cho, B. V. Merriënboer, D. Bahdanau, Y. Bengio, On the Properties of Neural Machine Translation: Encoder-Decoder Approaches, in: Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8), 2014, pp. 103–111.

[57] D. Bahdanau, K. Cho, Y. Bengio, Neural Machine Translation by Jointly Learning to Align and Translate, in: International Conference on Learning Representations (ICLR), 2015.

[58] T. Rocktäschel, E. Grefenstette, K. M. Hermann, T. Kočiský, P. Blunsom, Reasoning about Entailment with Neural

Attention, in: International Conference on Learning Representations (ICLR), 2016.

[59] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, Y. Bengio, Attention-Based Models for Speech Recognition, in: Advances in Neural Information Processing Systems (NIPS), 2015, pp. 577–585.

[60] N. Tishby, F. C. Pereira, , W. Bialek, The information bottleneck method, in: Proceedings of the Annual Allerton Conference on Communication, Control and Computing, 1999, pp. 368–377.

[61] N. Tishby, N. Zaslavsky, Deep learning and the information bottleneck principle, in: Information Theory Workshop (ITW), 2015, pp. 1–5.

[62] R. Schwartz-Ziv, N. Tishby, Opening the black box of Deep Neural Networks via Information, in: Why & When Deep Learning works: Looking Inside Deep Learning (ICRI-CI), 2017.

[63] P.-H. Chen, J. Chen, Y. Yeshurun, U. Hasson, J. V. Haxby, P. J. Ramadge, A Reduced-Dimension fMRI shared response model, in: Advances in Neural Information Processing Systems (NIPS), 2015, pp. 460–468.

[64] Q. Lu, P.-H. Chen, J. W. Pillow, P. J. Ramadge, K. A. Norman, U. Hasson, Shared Representational Geometry Across Neural Networks, in: Integration of Deep Learning Theories workshop, Conference on Neural Information Processing Systems (NeurIPS), 2018.

[65] J. Duchi, E. Hazan, Y. Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Journal of Machine Learning Research 12 (2011) 2121–2159.

[66] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research 15 (2014) 1929–1958.

[67] M. P. Marcus, M. A. Marcinkiewicz, B. Santorini, Building a Large Annotated Corpus of English: the Penn Treebank, Computational Linguistics 19 (1993) 313–330.

[68] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, arXiv:1301.3781.

[69] S. Merity, C. Xiong, J. Bradbury, R. Socher, Pointer Sentinel Mixture Models, in: International Conference on Learning Representations (ICLR), 2017.

[70] Y. Bengio, P. Frasconi, P. Simard, The Problem of Learning Long-Term Dependencies in Recurrent Networks, in: Proceedings IEEE International Conference on Neural Networks, 1993, pp. 1183–1195.

[71] R. Pascanu, T. Mikolov, Y. Bengio, On the difficulty of training recurrent neural networks, in: Proceedings of the International Conference on Machine Learning (ICML), 2013, pp. 1310–1318.

[72] L. Verwimp, J. Pelemans, H. Van hamme, P. Wambacq, Information-Weighted Neural Cache Language Models for ASR, in: Proceedings of the IEEE Workshop on Spoken Language Technology (SLT), 2018, pp. 756–762.

[73] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.