Recurrent Neural Networks and Language Modelling: Part 2

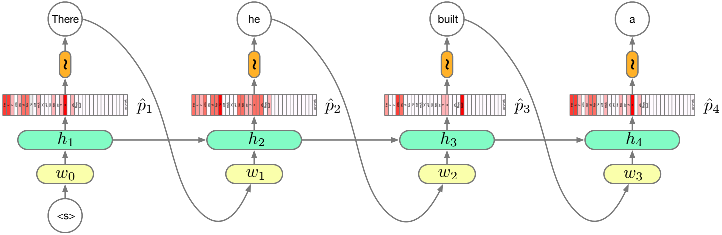
Phil Blunsom

phil.blunsom@cs.ox.ac.uk

Language Modelling: Review

Language models aim to represent the history of observed text $(w_1, \ldots, w_{t-1})$ succinctly in order to predict the next word (w_t):

- With count based n-gram LMs we approximate the history with just the previous n words.
- Neural n-gram LMs embed the same fixed n-gram history in a continues space and thus capture correlations between histories.
- With Recurrent Neural Network LMs we drop the fixed n-gram history and compress the entire history in a fixed length vector, enabling long range correlations to be captured.

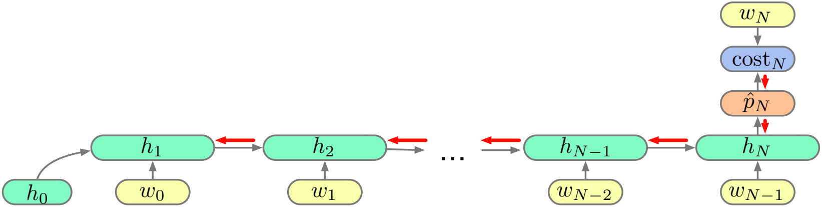


Capturing Long Range Dependencies

If an RNN Language Model is to outperform an n-gram model it must discover and represent long range dependencies:

p(sandcastle | Alice went to the beach. There she built a)

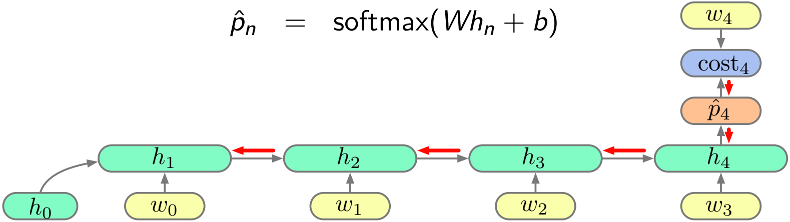
While a simple RNN LM can represent such dependencies in theory, can it learn them?



RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in cost_4 to changes in h_1:

$$\begin{aligned} h_n &= g(V[x_n; h_{n-1}] + c) \\ \hat{p}_n &= \text{softmax}(Wh_n + b) \end{aligned}$$

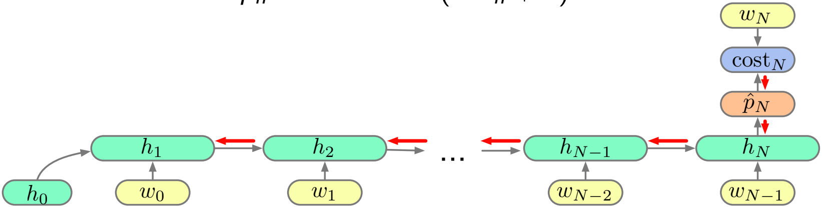


$$\frac{\partial \text{cost}_4}{\partial h_1} = \frac{\partial \text{cost}_4}{\partial \hat{p}_4} \frac{\partial \hat{p}_4}{\partial h_4} \frac{\partial h_4}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1}$$

RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in cost_N to changes in h_1:

$$\begin{aligned} h_n &= g(V[x_n; h_{n-1}] + c) \\ \hat{p}_n &= \text{softmax}(Wh_n + b) \end{aligned}$$

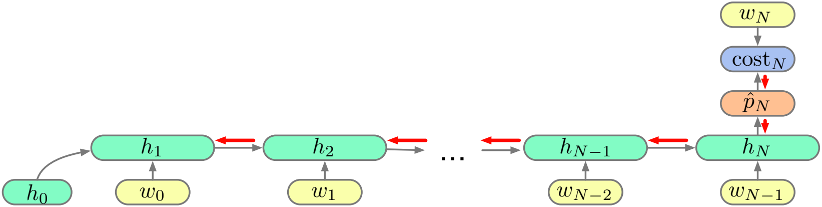


$$\frac{\partial \text{cost}_N}{\partial h_1} = \frac{\partial \text{cost}_N}{\partial \hat{p}_N} \frac{\partial \hat{p}_N}{\partial h_N} \left(\prod_{n \in \{N, \dots, 2\}} \frac{\partial h_n}{\partial h_{n-1}} \right)$$

RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in cost_N to changes in h_1:

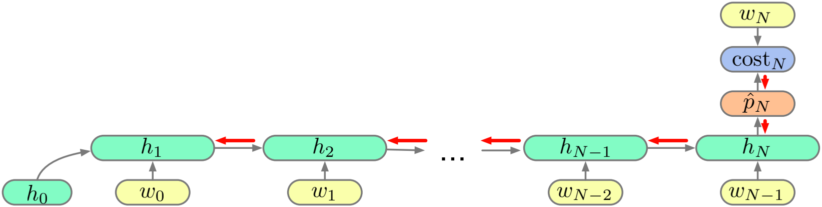
$$h_n = g(V[x_n; h_{n-1}] + c), \qquad \frac{\partial \text{cost}_N}{\partial h_1} = \frac{\partial \text{cost}_N}{\partial \hat{p}_N} \frac{\partial \hat{p}_N}{\partial h_N} \left(\prod_{n \in \{N,\ldots,2\}} \frac{\partial h_n}{\partial h_{n-1}} \right)$$



RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in cost_N to changes in h_1:

$$h_n = g(\underbrace{V_x x_n + V_h h_{n-1} + c}_{z_n}), \quad \frac{\partial \text{cost}_N}{\partial h_1} = \frac{\partial \text{cost}_N}{\partial \hat{p}_N} \frac{\partial \hat{p}_N}{\partial h_N} \left(\prod_{n \in \{N,\ldots,2\}} \frac{\partial h_n}{\partial z_n} \frac{\partial z_n}{\partial h_{n-1}} \right)$$

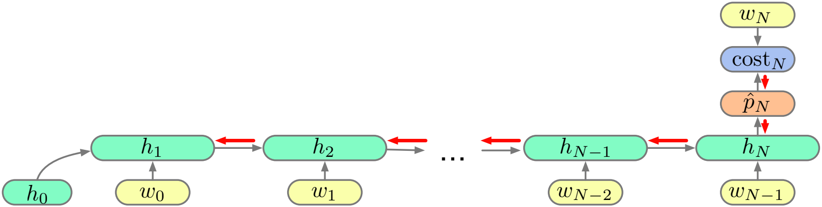


RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in cost_N to changes in h_1:

$$h_n = g(\underbrace{V_x x_n + V_h h_{n-1} + c}_{z_n}), \quad \frac{\partial \text{cost}_N}{\partial h_1} = \frac{\partial \text{cost}_N}{\partial \hat{p}_N} \frac{\partial \hat{p}_N}{\partial h_N} \left(\prod_{n \in \{N,\ldots,2\}} \frac{\partial h_n}{\partial z_n} \frac{\partial z_n}{\partial h_{n-1}} \right)$$

$$\frac{\partial h_n}{\partial z_n} = \text{diag}\left(g'(z_n)\right) \qquad \frac{\partial z_n}{\partial h_{n-1}} = V_h$$



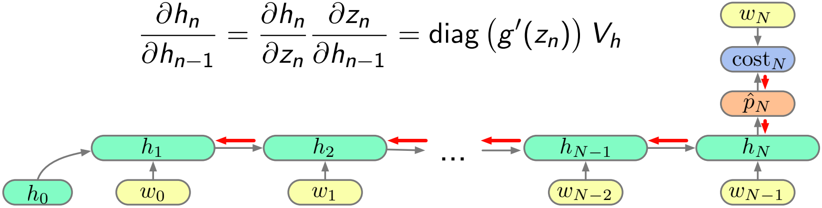
RNNs: Exploding and Vanishing Gradients

Consider the path of partial derivatives linking a change in cost_N to changes in h_1:

$$h_n = g(\underbrace{V_x x_n + V_h h_{n-1} + c}_{z_n}), \quad \frac{\partial \mathrm{cost}_N}{\partial h_1} = \frac{\partial \mathrm{cost}_N}{\partial \hat{p}_N}\frac{\partial \hat{p}_N}{\partial h_N}\left(\prod_{n\in\{N,\dots,2\}} \frac{\partial h_n}{\partial z_n}\frac{\partial z_n}{\partial h_{n-1}}\right)$$

$$\frac{\partial h_n}{\partial z_n} = \mathrm{diag}\left(g'(z_n)\right) \qquad\qquad \frac{\partial z_n}{\partial h_{n-1}} = V_h$$

$$\frac{\partial h_n}{\partial h_{n-1}} = \frac{\partial h_n}{\partial z_n}\frac{\partial z_n}{\partial h_{n-1}} = \mathrm{diag}\left(g'(z_n)\right) V_h$$



RNNs: Exploding and Vanishing Gradients

$$\frac{\partial \mathrm{cost}_N}{\partial h_1} = \frac{\partial \mathrm{cost}_N}{\partial \hat{p}_N} \frac{\partial \hat{p}_N}{\partial h_N} \left(\prod_{n \in \{N, \ldots, 2\}} \mathrm{diag}\left(g'(z_n)\right) V_h \right)$$

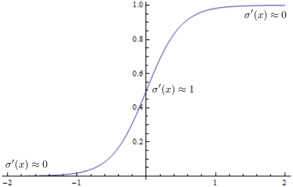
The core of the recurrent product is the repeated multiplication of V_h. If the largest eigenvalue of V_h is:

- 1, then gradient will propagate,
- > 1, the product will grow exponentially (explode),
- < 1, the product shrinks exponentially (vanishes).

RNNs: Exploding and Vanishing Gradients

Most of the time the spectral radius of V_h is small. The result is that the gradient vanishes and long range dependencies are not learnt.

Many non-linearities ($g(\cdot)$) can also shrink the gradient.



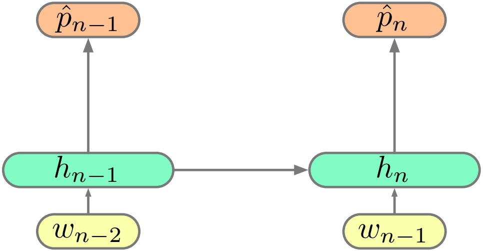
Second order optimisers ((Quasi-)Newtonian Methods) can overcome this, but they are difficult to scale. Careful initialisation of the recurrent weights can help.[1]

Here we will consider the most popular solution, changing the network architecture.

[1] Stephen Merity: Explaining and illustrating orthogonal initialization for recurrent neural networks. `smerity.com/articles/2016/orthogonal_init.html`

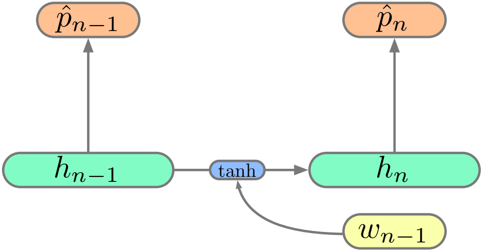
Long Short Term Memory (LSTM)

$$h_n = g(V[w_{n-1}; h_{n-1}] + c)$$



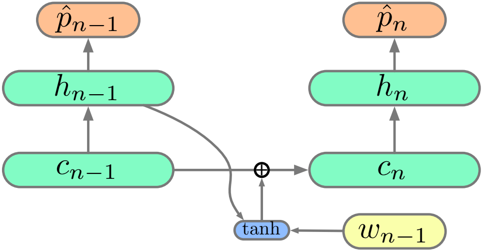
Long Short Term Memory (LSTM)

$$h_n = \mathrm{tanh}(V[w_{n-1}; h_{n-1}] + c)$$



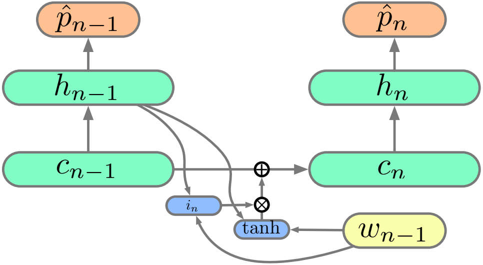
Long Short Term Memory (LSTM)

$$c_n \quad = c_{n-1} + \tanh(V[w_{n-1}; h_{n-1}] + b_c)$$



Long Short Term Memory (LSTM)

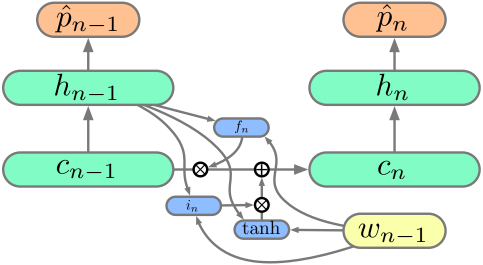
$$c_n \quad = c_{n-1} + \color{red}{i_n} \circ \tanh(V[w_{n-1}; h_{n-1}] + b_c)$$



$$i_n \quad = \sigma\left(W_i[w_{n-1}; h_{t-1}] + b_i\right).$$

Long Short Term Memory (LSTM)

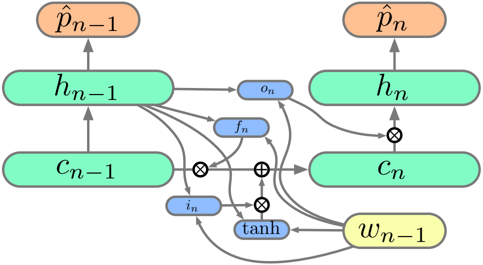
$$c_n \quad = f_n \circ c_{n-1} + i_n \circ \tanh(V[w_{n-1}; h_{n-1}] + b_c)$$



$$\begin{aligned} i_n \quad &= \sigma\left(W_i[w_{n-1}; h_{t-1}] + b_i\right), \\ f_n \quad &= \sigma\left(W_f[w_{n-1}; h_{t-1}] + b_f\right). \end{aligned}$$

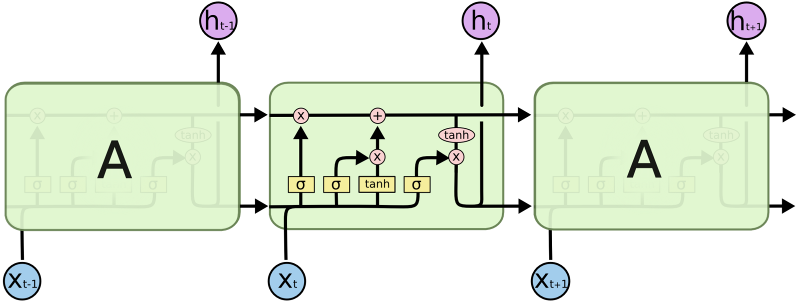
Long Short Term Memory (LSTM)

$$
\begin{aligned}
c_n &= f_n \circ c_{n-1} + i_n \circ \tanh(V[w_{n-1}; h_{n-1}] + b_c) \\
h_n &= o_n \circ \tanh\left(W_h c_n + b_h\right).
\end{aligned}
$$



$$
\begin{aligned}
i_n &= \sigma\left(W_i[w_{n-1}; h_{t-1}] + b_i\right), \\
f_n &= \sigma\left(W_f[w_{n-1}; h_{t-1}] + b_f\right), \\
o_n &= \sigma\left(W_o[w_{n-1}; h_{t-1}] + b_o\right).
\end{aligned}
$$

Long Short Term Memory (LSTM)



Christopher Olah: Understanding LSTM Networks
`colah.github.io/posts/2015-08-Understanding-LSTMs/`

LSTM LM

The LSTM cell,[2]

$$c_n = f_n \circ c_{n-1} + i_n \circ \hat{c}_n,$$

$$\begin{aligned}
\hat{c}_n &= \tanh\left(W_c[w_{n-1}; h_{t-1}] + b_c\right), \\
h_n &= o_n \circ \tanh\left(W_h c_n + b_h\right). \\
i_n &= \sigma\left(W_i[w_{n-1}; h_{t-1}] + b_i\right), \\
f_n &= \sigma\left(W_f[w_{n-1}; h_{t-1}] + b_f\right), \\
o_n &= \sigma\left(W_o[w_{n-1}; h_{t-1}] + b_o\right),
\end{aligned}$$

where h_n is the hidden state at time n, and i, f, o are the input, forget, and output gates respectively.[3]

[2] Long Short-Term Memory. Hochreiter and Schmidhuber, Neural Computation 1997.

[3] Optimizing Performance of Recurrent Neural Networks on GPUs. Appleyard et al., arXiv 2016.

LSTM LM

The LSTM cell,[2]

$$c_n = f_n \circ c_{n-1} + (1 - f_n) \circ \hat{c}_n,$$

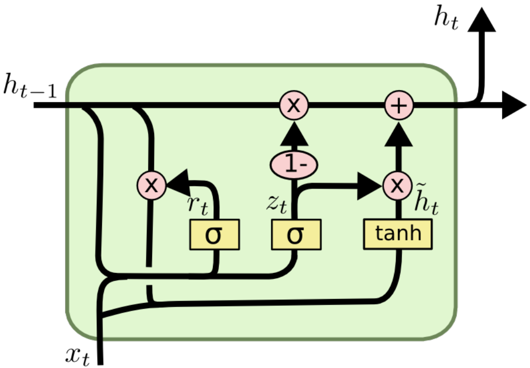
$$\begin{aligned}
\hat{c}_n &= \tanh\left(W_c[w_{n-1}; h_{t-1}] + b_c\right), \\
h_n &= o_n \circ \tanh\left(W_h c_n + b_h\right). \\
i_n &= \sigma\left(W_i[w_{n-1}; h_{t-1}] + b_i\right), \\
f_n &= \sigma\left(W_f[w_{n-1}; h_{t-1}] + b_f\right), \\
o_n &= \sigma\left(W_o[w_{n-1}; h_{t-1}] + b_o\right),
\end{aligned}$$

where h_n is the hidden state at time n, and i, f, o are the input, forget, and output gates respectively.[3]

[2] Long Short-Term Memory. Hochreiter and Schmidhuber, Neural Computation 1997.

[3] Optimizing Performance of Recurrent Neural Networks on GPUs. Appleyard et al., arXiv 2016.

Gated Recurrent Unit (GRU)



Christopher Olah: Understanding LSTM Networks
`colah.github.io/posts/2015-08-Understanding-LSTMs/`

Gated Recurrent Unit (GRU)

The GRU cell,[4]

$$h_n = (1 - z_n) \circ h_{n-1} + z_n \circ \hat{h}_n.$$

$$\begin{aligned}
z_n &= \sigma\left(W_z[x_n; h_{t-1}] + b_z\right), \\
r_n &= \sigma\left(W_r[x_n; h_{t-1}] + b_r\right), \\
\hat{h}_n &= \tanh\left(W_{\hat{h}}[x_n; r_n \circ h_{n-1}] + b_{\hat{h}}\right).
\end{aligned}$$

[4] Learning Phrase Representations using RNN EncoderDecoder for Statistical Machine Translation. Cho et al, EMNLP 2014.

LSTMs and GRUs

Good

- Careful initialisation and optimisation of vanilla RNNs can enable them to learn long(ish) dependencies, but gated additive cells, like the LSTM and GRU, often just work.
- The (re)introduction of LSTMs has been key to many recent developments, e.g. Neural Machine Translation, Speech Recognition, TTS, etc.

Bad

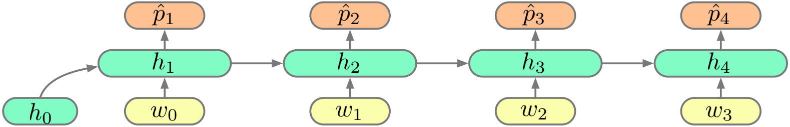
- LSTMs and GRUs have considerably more parameters and computation per memory cell than a vanilla RNN, as such they have less memory capacity per parameter.[5]

[5] Capcity and Trainability in Recurrent Neural Networks. Collins et al., arXiv 2016.

Deep RNN LMs

The memory capacity of an RNN can be increased by employing a larger hidden layer h_n, but a linear increase in h_n results in a quadratic increase in model size and computation.

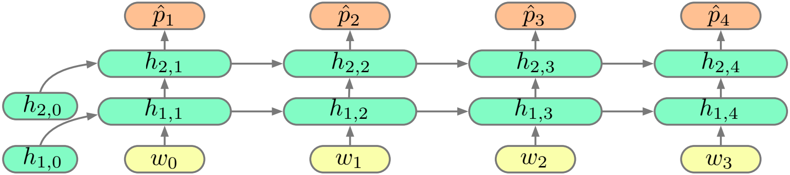
A Deep RNN increases the memory and representational ability with linear scaling.



Deep RNN LMs

The memory capacity of an RNN can be increased by employing a larger hidden layer h_n, but a linear increase in h_n results in a quadratic increase in model size and computation.

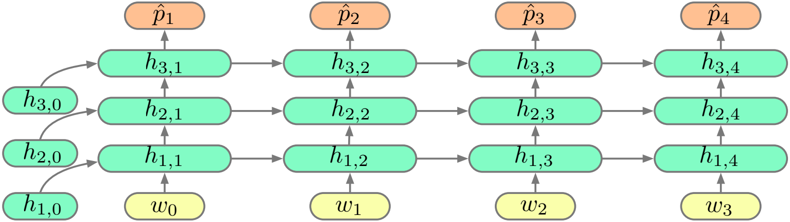
A Deep RNN increases the memory and representational ability with linear scaling.



Deep RNN LMs

The memory capacity of an RNN can be increased by employing a larger hidden layer h_n, but a linear increase in h_n results in a quadratic increase in model size and computation.

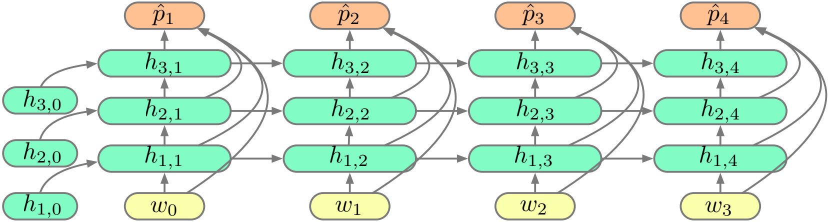
A Deep RNN increases the memory and representational ability with linear scaling.



Deep RNN LMs

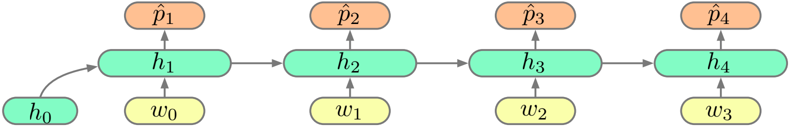
The memory capacity of an RNN can be increased by employing a larger hidden layer h_n, but a linear increase in h_n results in a quadratic increase in model size and computation.

A Deep RNN increases the memory and representational ability with linear scaling.



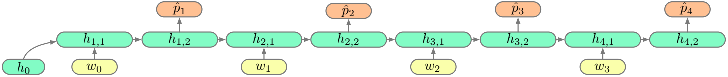
Deep RNN LM

Alternatively we can increase depth in the time dimension. This improves the representational ability, but not the memory capacity.



Deep RNN LM

Alternatively we can increase depth in the time dimension. This improves the representational ability, but not the memory capacity.



The recently proposed Recurrent Highway Network[6] employs a deep-in-time GRU-like cell with untied weights, and reports strong results on language modelling.

[6] Recurrent Highway Networks. Zilly et al., arXiv 2016.

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \mathrm{softmax}\left(W h_n + b \right)$$

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \text{softmax}\left(W h_n + b \right)$$

Solutions

Short-lists: use the neural LM for the most frequent words, and a traditional ngram LM for the rest. While easy to implement, this nullifies the neural LM's main advantage, i.e. generalisation to rare events.
Batch local short-lists: approximate the full partition function for data instances from a segment of the data with a subset of the vocabulary chosen for that segment.[7]

[7] On Using Very Large Target Vocabulary for Neural Machine Translation. Jean et al., ACL 2015

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \operatorname{softmax}\left(Wh_n + b\right)$$

Solutions

Approximate the gradient/change the objective: if we did not have to sum over the vocabulary to normalise during training it would be much faster. It is tempting to consider maximising likelihood by making the log partition function an independent parameter c, but this leads to an ill defined objective.

$$\hat{p}_n \equiv \exp\left(Wh_n + b\right) \times \exp(c)$$

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \mathrm{softmax}\left(Wh_n + b\right)$$

Solutions

Approximate the gradient/change the objective: Mnih and Teh use Noise Contrastive Estimation (NCE). This amounts to learning a binary classifier to distinguish data samples from (k) samples from a noise distribution (a unigram is a good choice):

$$p(\mathrm{Data} = 1 | \hat{p}_n) = \frac{\hat{p}_n}{\hat{p}_n + k p_{\mathrm{noise}}(w_n)}$$

Now parametrising the log partition function as c does not degenerate. This is very effective for speeding up training, but has no impact on testing time.[7]

[7] In practice fixing $c = 0$ is effective. It is tempting to believe that this noise contrastive objective justifies using unnormalised scores at test time. This is not the case and leads to high variance results.

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \mathrm{softmax}\left(Wh_n + b\right)$$

Solutions

Approximate the gradient/change the objective: NCE defines a binary classification task between true or noise words with a logistic loss. An alternative proposed by Jozefowicz et al.[7], called Importance Sampling (IS), defines a multiclass classification problem between the true word and noise samples, with a Softmax and cross entropy loss.

[7] Exploring the Limits of Language Modeling. Jozefowicz et al., arXiv 2016.

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \mathrm{softmax}\left(Wh_n + b\right)$$

Solutions

Factorise the output vocabulary: One level factorisation works well (Brown clustering is a good choice, frequency binning is not):

$$p(w_n|\hat{p}_n^{\text{class}}, \hat{p}_n^{\text{word}}) = p(\text{class}(w_n)|\hat{p}_n^{\text{class}}) \times p(w_n|\text{class}(w_n), \hat{p}_n^{\text{word}}),$$

where the function class($\cdot$) maps each word to one class. Assuming balanced classes, this gives a $\sqrt{V}$ speedup.

Scaling: Large Vocabularies

Much of the computational cost of a neural LM is a function of the size of the vocabulary and is dominated by calculating:

$$\hat{p}_n = \text{softmax}\left(Wh_n + b\right)$$

Solutions

Factorise the output vocabulary: By extending the factorisation to a binary tree (or code) we can get a $\log V$ speedup,[7] but choosing a tree is hard (frequency based Huffman coding is a poor choice):

$$p(w_n|h_n) = \prod_i p(d_i|r_i, h_n),$$

where d_i is i^{th} digit in the code for word w_n, and r_i is the parameter vector for the i^{th} node in the path corresponding to that code.

Recently Grave et al. proposed optimising an n-ary factorisation tree for both perplexity and GPU throughput.[8]

[7] A scalable hierarchical distributed language model. Mnih and Hinton, NIPS'09.

[8] Efficient softmax approximation for GPUs. Grave et al., arXiv 2016

Scaling: Large Vocabularies

Full Softmax

Training: Computation and memory $O(V)$,
Evaluation: Computation and memory $O(V)$,
Sampling: Computation and memory $O(V)$.

Balanced Class Factorisation

Training: Computation $O(\sqrt{V})$ and memory $O(V)$,
Evaluation: Computation $O(\sqrt{V})$ and memory $O(V)$,
Sampling: Computation and memory $O(V)$ (but average case is better).

Balanced Tree Factorisation

Training: Computation $O(\log V)$ and Memory $O(V)$,
Evaluation: Computation $O(\log V)$ and Memory $O(V)$,
Sampling: Computation and Memory $O(V)$ (but average case is better).

NCE / IS

Training: Computation $O(k)$ and Memory $O(V)$,
Evaluation: Computation and Memory $O(V)$,
Sampling: Computation and Memory $O(V)$.

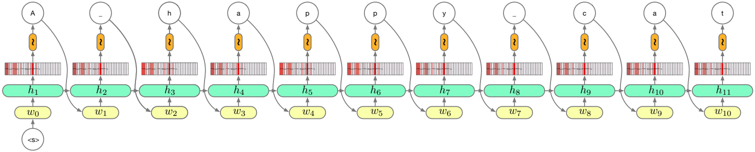
Sub-Word Level Language Modelling

An alternative to changing the softmax is to change the input granularity and model text at the morpheme or character level.

This results in a much smaller softmax and no unknown words, but the downsides are longer sequences and longer dependencies.

This also allows the model to capture subword structure and morphology: *disunited* ↔ *disinherited* ↔ *disinterested*.

Charater LMs lag word based models in perplexity, but are clearly the future of language modelling.



Regularisation: Dropout

Large recurrent networks often overfit their training data by memorising the sequences observed. Such models generalise poorly to novel sequences.

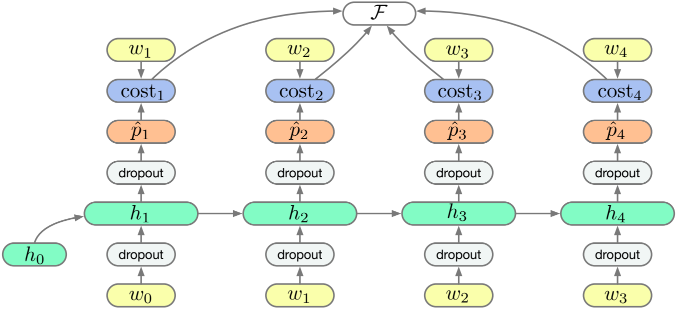
A common approach in Deep Learning is to overparametrise a model, such that it could easily memorise the training data, and then heavily regularise it to facilitate generalisation.

The regularisation method of choice is often Dropout.[9]

[9] Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Srivastava et al. JMLR 2014.

Regularisation: Dropout

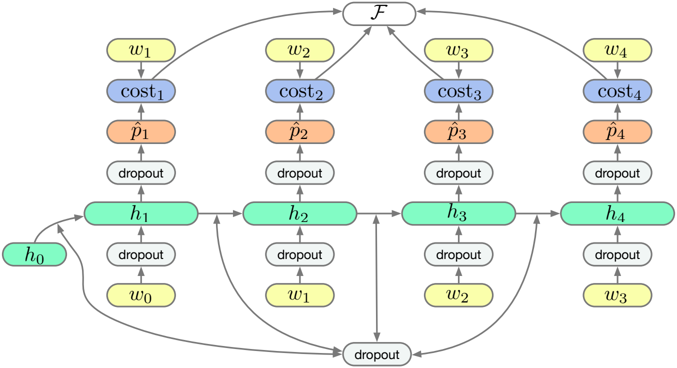
Dropout is ineffective when applied to recurrent connections, as repeated random masks zero all hidden units in the limit. The most common solution is to only apply dropout to non-recurrent connections.[10]



[10] Recurrent neural network regularization. Zaremba et al., arXiv 2014.

Regularisation: Bayesian Dropout (Gal)

Gal and Ghahramani[11] advocate tying the recurrent dropout mask and sampling at evaluation time:



[11] A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. Gal and Ghahramani, NIPS 2016.

Summary

Long Range Dependencies

- The repeated multiplication of the recurrent weights V lead to vanishing (or exploding) gradients,
- additive gated architectures, such as LSTMs, significantly reduce this issue.

Deep RNNs

- Increasing the size of the recurrent layer increases memory capacity with a quadratic slow down,
- deepening networks in both dimensions can improve their representational efficiency and memory capacity with a linear complexity cost.

Large Vocabularies

- Large vocabularies, $V > 10^4$, lead to slow softmax calculations,
- reducing the number of vector matrix products evaluated, by factorising the softmax or sampling, reduces the training overhead significantly.
- Different optimisations have different training and evaluation complexities which should be considered.

References

Papers

On the difficulty of training recurrent neural networks. Pascanu et al., ICML 2013.

Long Short-Term Memory. Hochreiter and Schmidhuber, Neural Computation 1997.

Learning Phrase Representations using RNN EncoderDecoder for Statistical Machine Translation. Cho et al, EMNLP 2014

A scalable hierarchical distributed language model. Mnih and Hinton, NIPS 2009.

A fast and simple algorithm for training neural probabilistic language models. Mnih and Teh, ICML 2012.

On Using Very Large Target Vocabulary for Neural Machine Translation. Jean et al., ACL 2015

Exploring the Limits of Language Modeling. Jozefowicz et al., arXiv 2016.

Efficient softmax approximation for GPUs. Grave et al., arXiv 2016

Notes on Noise Contrastive Estimation and Negative Sampling. Dyer, arXiv 2014.

Pragmatic Neural Language Modelling in Machine Translation. Baltescu and Blunsom, NAACL 2015

A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. Gal and Ghahramani, NIPS 2016.

Recurrent Highway Networks. Zilly et al., arXiv 2016.

Capcity and Trainability in Recurrent Neural Networks. Collins et al., arXiv 2016.

Optimizing Performance of Recurrent Neural Networks on GPUs. Appleyard et al., arXiv 2016.

Blog Posts

Christopher Olah: Understanding LSTM Networks
`colah.github.io/posts/2015-08-Understanding-LSTMs/`

Yarin Gal: Uncertainty in Deep Learning `mlg.eng.cam.ac.uk/yarin/blog_2248.html`

The End

Next week we will cover representation learning for classification and accelerating deep networks using GPUs.

DeepMind

UNIVERSITY OF OXFORD

DEPARTMENT OF COMPUTER SCIENCE