Character-Level Language Modeling with Deeper Self-Attention

Rami Al-Rfou* **Dokook Choe*** **Noah Constant*** **Mandy Guo*** **Llion Jones***

Google AI
{rmyeid, choed, nconstant, xyguo, llion}@google.com

Abstract

LSTMs and other RNN variants have shown strong performance on character-level language modeling. These models are typically trained using truncated backpropagation through time, and it is common to assume that their success stems from their ability to remember long-term contexts. In this paper, we show that a deep (64-layer) transformer model [42] with fixed context outperforms RNN variants by a large margin, achieving state of the art on two popular benchmarks—1.13 bits per character on `text8` and 1.06 on `enwik8`. To get good results at this depth, we show that it is important to add auxiliary losses, both at intermediate network layers and intermediate sequence positions.

1 Introduction

Character-level modeling of natural language text is a challenging, for several reasons. First, the model must learn a large vocabulary of words "from scratch". Second, natural text exhibits dependencies over long distances of hundreds or thousands of time steps. Third, character sequences are longer than word sequences and thus require significantly more steps of computation.

In recent years, strong character-level language models typically follow a common template [32, 31, 40]. A recurrent neural net (RNN) is trained over mini-batches of text sequences, using a relatively short sequence length (e.g. 200 tokens). To capture context longer than the batch sequence length, training batches are provided in sequential order, and the hidden states from the previous batch are passed forward to the current batch. This procedure is known as "truncated backpropagation through time" (TBTT), because the gradient computation doesn't proceed further than a single batch [44]. A range of methods have arisen for unbiasing and improving TBTT [41, 20].

While this technique gets good results, it adds complexity to the training procedure, and recent work suggests that models trained in this manner don't actually make "strong" use of long-term context. For example Khandelwal et al. [22] find that a word-based LSTM language model only effectively uses ~200 tokens of context (even if more is provided), and that word order only has an effect within the last ~50 tokens.

In this paper, we show that a non-recurrent model can achieve strong results on character-level language modeling. Specifically, we use a deep network of transformer self-attention layers [42] with causal (backward-looking) attention to process fixed-length inputs and predict upcoming characters. The model is trained on mini-batches of sequences from random positions in the training corpus, with no information passed from one batch to the next.

Our primary finding is that the transformer architecture is well-suited to language modeling over long sequences and could replace RNNs in this domain. We speculate that the transformer's success

*Equal contribution.

Preprint. Work in progress.

here is due to its ability to "quickly" propagate information over arbitrary distances; by comparison, RNNs need to learn to pass relevant information forward step by step.

We also find that some modifications to the basic transformer architecture are beneficial in this domain. Most importantly, we add three auxiliary losses, requiring the model to predict upcoming characters (i) at intermediate sequence positions, (ii) from intermediate hidden representations, and (iii) at target positions multiple steps in the future. These losses speed up convergence, and make it possible to train deeper networks.

2 Character Transformer Model

Language models assign a probability distribution over token sequences $t_{0:L}$ by factoring out the joint probability as follows, where L is the sequence length:

$$\Pr(t_{0:L}) = P(t_0) \prod_{i=1}^{L} \Pr(t_i | t_{0:i-1}), \tag{1}$$

To model the conditional probability $\Pr(t_i | t_{0:i-1})$, we train a transformer network to process the character sequence $t_{0:i-1}$. Transformer networks have recently showed significant gains in tasks that require processing sequences accurately and efficiently.

Our character-level transformer architecture has 64 transformer layers. Following Vaswani et al. [42], by "transformer layer" we mean a block containing a multihead self-attention sub-layer followed by a feed-forward network of two fully connected sub-layers. For more details on the transformer architecture, refer to [42] and the `tensor2tensor` library[2]. To ensure that the model's predictions are only conditioned on past characters, we mask our attention layers with a causal attention, so each position can only attend leftward. This is the same as the "masked attention" in the decoder component of the original transformer architecture used for sequence-to-sequence problems [42].

Figure 1 shows our initial model with the causal attention mask limiting information flow from left to right. Each character prediction is conditioned only on the characters that appeared earlier.

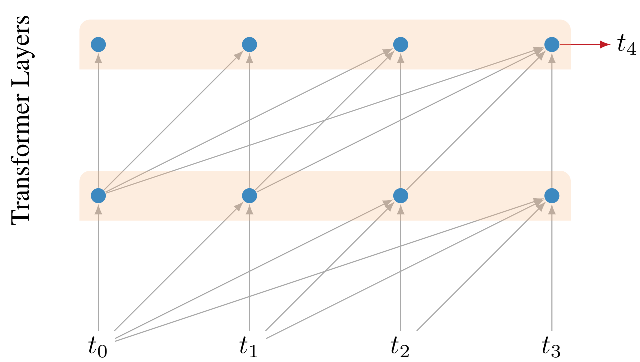


Figure 1: Character transformer network of two layers processing a four character sequence to predict t_4. The causal attention mask limits information to left-to-right flow. Red arrows highlight the prediction task the network has to learn.

2.1 Auxiliary Losses

Our network is, to our knowledge, deeper than any transformer network discussed in previous work. In initial experiments, we found training a network deeper than ten layers to be challenging, with slow convergence and poor accuracy. We were able to deepen the network to better effect through the addition auxiliary losses, which sped up convergence of the training significantly.

[2] `https://github.com/tensorflow/tensor2tensor`

We add several types of auxiliary losses, corresponding to intermediate positions, intermediate layers, and non-adjacent targets. We hypothesize that these losses not only speed up convergence but also serve as an additional regularizer. During training, the auxiliary losses get added to the total loss of the network with discounted weights. Each type of auxiliary loss has its own schedule of decay. During evaluation and inference time, only the prediction of the final position at the final layer is used.

One consequence of this approach is that a number of the network parameters are only used during training—specifically, the parameters in the output classification layers associated with predictions made from intermediate layers and predictions over non-adjacent targets. Thus, when listing the number of parameters in our models, we distinguish between "training parameters" and "inference parameters".

Multiple Positions – First, we add prediction tasks for each position in the final layer, extending our predictions from one per example to $|L|$ (sequence length). Note, predicting over all sequence positions is standard practice in RNN-based approaches. However in our case, since no information is passed forward across batches, this is forcing the model to predict given smaller contexts—sometimes just one or two characters. It is not obvious whether these secondary training tasks should help on the primary task of predicting with full context. However, we find that adding this auxiliary loss speeds up training and gives better results (see Section 4.1). Figure 2 illustrates the task of predicting across all sequence positions. We add these losses during training without decaying their weights.

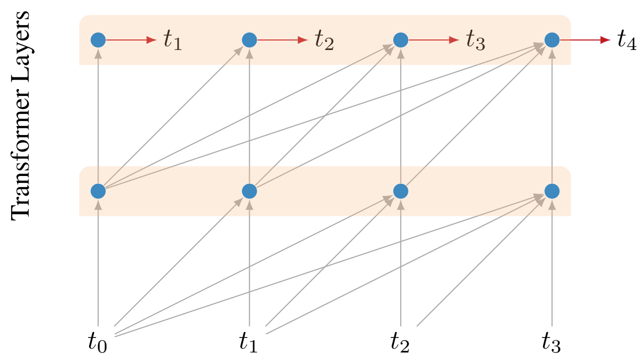


Figure 2: Adding the intermediate positions prediction tasks to our network. Now, we predict the final character t_4 and all intermediate characters $t_{0:3}$. t_3 has access only to $t_{0:2}$ because of the causal attention masks. All losses are equally added without weight decay during training.

Intermediate Layer Losses – In addition to the final prediction layer, we add predictions made from the output of each intermediate transformer layer. As with the final layer, we add predictions for all intermediate positions in the sequence (see Figure 3). Lower layers are weighted to contribute less and less to the loss as training progresses. If there are n layers total, then the l^{th} intermediate layer stops contributing any loss after finishing $l/2n$ of the training. This schedule drops all intermediate losses after half of the training is done.

Multiple Targets – At each position in the sequence, the model makes two (or more) predictions of future characters. For each new target we introduce a separate classifier. The losses of the extra targets get weighted by a multiplier of 0.5 before being added to their corresponding layer loss.

2.2 Positional Embeddings

In the basic transformer network described in [42], a sinusoidal timing signal is added to the input sequence prior to the first transformer layer. However, as our network is deeper (64 layers), we hypothesize that the timing information may get lost during the propagation through the layers. To address this, we replace the timing signal with a learned per-layer positional embedding added to the input sequence before each transformer layer. We are able to safely use positional embeddings for our task, as we don't require the model to generalize to longer contexts than those seen during training.

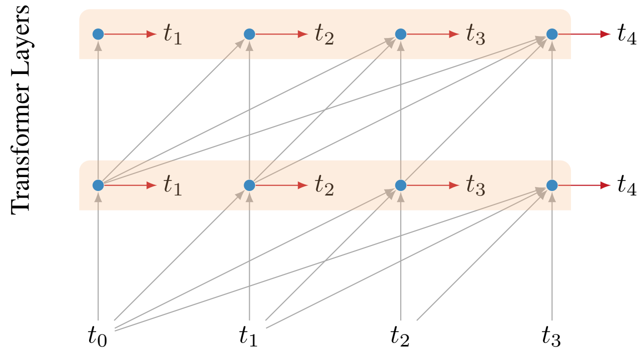


Figure 3: Our network after adding prediction tasks for the intermediate layers. For this example of two layers, the losses of the intermediate layer prediction tasks will be absent after finishing 25% of the training.

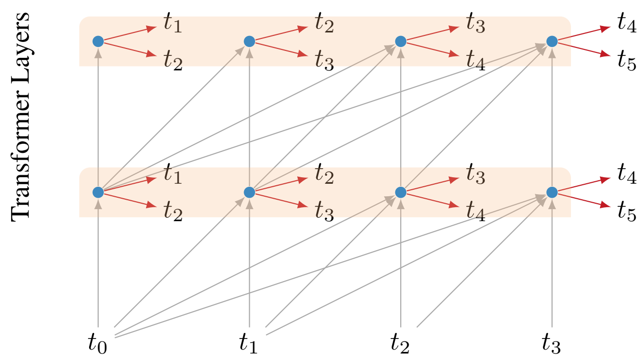


Figure 4: Our example network after adding two predictions per position.

3 Experimental Setup

3.1 Datasets

For evaluation we focus mainly on `text8` [29]. This dataset consists of English Wikipedia articles, with superfluous content removed (tables, links to foreign language versions, citations, footnotes, markup, punctuation). The remaining text is processed to use a minimal character vocabulary of 27 unique characters—lowercase letters `a` through `z`, and space. Digits are replaced by their spelled-out equivalents, so "20" becomes "`two zero`". Character sequences not in the range [a-zA-Z] are converted to a single space. Finally, the text is lowercased. The size of the corpus is 100M characters. Following [33, 49], we split the data into 90M characters for train, 5M characters for dev, and 5M characters for test.

To aid in comparison with other recent approaches, we also evaluate our model on `enwik8` [29] which is 100M bytes of unprocessed Wikipedia text, including markup and non-Latin characters. There are 205 unique bytes in the dataset. Following [5], and as in `text8`, we split the data into 90M, 5M and 5M for training, dev and test respectively.

3.2 Training

Compared to most models based on transformers [42, 37], our model is very deep, with 64 transformer layers and each layer using two attention heads. Each transformer layer has a hidden size of 512 and a filter size of 2048. We feed our model sequences of length 512. Each item in the sequence represents a single byte (or equivalently, one character in `text8`) which gets replaced by its embedding, a vector of size 512. We add to the byte embeddings a separate learned positional embedding for each of the 512 token positions. These positional embeddings are described in Section 2.2. We do the same addition at each layer activation throughout the network. The positional embeddings are not shared across the layers. With two predictions per position, each layer learns to predict 1024 characters. Because we are primarily interested in predicting the immediately following character (one step away), we halve the loss of predicting characters two steps away. The prediction layers are logistic regression layers over the full 256 outputs (the number of unique bytes). To demonstrate the generality of the model, we always train and predict over all 256 labels, even on datasets that cover a smaller vocabulary. Despite this, we found that in practice the model never predicted a byte value outside of the ones observed in the training dataset.

The model has approximately 235 million parameters, which is larger than the number of characters in the `text8` training corpus. To regularize the model, we apply dropout in the attention and ReLU layers with a probability of 0.55. We use the momentum optimizer with 0.99 momentum. The learning rate is fixed during the training to 0.003. We train our model for 4 million steps, with each step processing a batch of 16 randomly selected sequences. We drop the intermediate layers losses consecutively as described in Section 2.1. Starting from the first layer, after every 62.5K ($= 4\text{M} \times \frac{1}{2*64}$) steps, we drop the losses introduced by the next layer. According to this schedule, after training is halfway complete, only the final layer losses are present.

3.3 Evaluation

At inference time, we use the model's prediction at the final position of the final layer to compute the probability of a character given a context of 512 characters. There is no state passed between predictions as would be the case with RNN models, so for each character predicted we have to process the context from scratch. Because there is no reused computation from previous steps, our model requires expensive computational resources for evaluation and inference. We measure the performance of training checkpoints (roughly every $10,000$ steps) by evaluating bits per character (bpc) over the entire the validation set, and save the parameters that perform the best. Our best model is achieved after around 2.5 million steps of training, which takes 175 hours on a single Google Cloud TPU v2.

4 Results

We report the performance of our best model (T64) on the validation and test sets. Table 1 compares our models against several recent results. On the test set, we achieve a new state of the art, 1.13 bpc. This model is 5x larger than previous models, which necessitated aggressive dropout rates of 0.55. For better comparison with smaller models, we also train a smaller model (T12) with 41M parameters. This model consists of 12 layers, and trained for 8M steps, with a reduced dropout rate of 0.2. All other settings were left the same as T64. Our smaller model still outperforms previous models, achieving 1.18 bpc on the test dataset. Increasing the depth of the network from 12 layers to 64 improved the results significantly, with the auxiliary losses enabling the training to better utilize the depth of the network. Note, our models do not use dynamic evaluation [24], a technique that adjusts model weights at test time by training on test data.

Model	Parameters (Millions) training	inference	bpc
LSTM [8]	-	-	1.43
BN-LSTM [8]	-	-	1.36
LayerNorm HM-LSTM [4]	35	35	1.29
Recurrent Highway Networks [51]	45	45	1.27
mLSTM [25]	45	45	1.27
T12 (ours)	44	41	**1.18**
T64 (ours)	235	219	**1.13**
mLSTM + dynamic eval [24]	45	-	1.19

Table 1: Comparison of various models on `text8` test.

Table 2 shows the performance of our model given different context sizes. We are able to achieve state-of-the-art results once the context increases beyond 128 characters, with the best performance of 1.06 bpc at 512 characters. As expected, the model performs better when it is given more context.

However this trend levels off after 512 characters; we do not see better results using a context of 1024.

	bpc		Accuracy (%)	
Context	dev	test	dev	test
32	1.25	1.34	72.8	71.1
64	1.17	1.26	74.8	73.0
128	1.12	1.20	76.1	74.4
256	1.09	1.16	76.9	75.3
512	1.06	1.13	77.3	75.9

Table 2: Bits per character (bpc) and accuracy of our best model on `text8` dev and test, for different context lengths.

Using the same hyperparameters and training procedure for `text8`, we also train and evaluate the T12 and T64 architectures on `enwik8`. Table 3 compares our models with recent results from the literature on the bits per byte (bpb) metric. Note, several previous authors discuss "bits per character" on `enwik8` but are in fact reporting bits per byte. Without retuning for this dataset, our models still achieve state-of-the-art performance.

	Parameters (Millions)		
Model	training	inference	bpb
Large FS-LSTM-4 [34]	47	-	1.25
Large mLSTM [25]	46	-	1.24
cmix v13 [23]	-	-	1.23
T12 (ours)	44	41	**1.11**
T64 (ours)	235	219	**1.06**
mLSTM + dynamic eval [24]	46	-	1.08

Table 3: Comparison of various models on `enwik8` test.

4.1 Ablation Experiments

To better understand the relative importance of the several modifications we proposed, we run an ablation analysis. We start from our best model T64 and then remove one modification at a time. For example, when we disable *Multiple Positions*, the model gets trained with only the last position loss for each layer. This corresponds to calculating $\{L(t_4 \mid t_{0:3}), L(t_5 \mid t_{0:3})\}$ in the example shown in Figure 4 for both the first and the second layers. When disabling *Positional Embeddings*, we add the default transformer sinusoidal timing signal before the first layer.

Description	bpc	Δbpc
T64 (Baseline)	1.062	-
T64 without Multiple Positions	2.482	1.420
T64 without Intermediate Layer Losses	1.158	0.096
T64 without Positional Embeddings	1.069	0.007
T64 without Multiple Targets	1.068	0.006
T64 with SGD Optimizer	1.065	0.003

Table 4: Evaluation of T64 on `text8` dev with context set to 512. Disabling each feature or loss lowers the quality of the model. The biggest win comes from adding multiple positions and intermediate layers losses.

For the ablation experiments, we reuse the hyperparameters from our best model to avoid a prohibitively expensive parameter search for each ablation. The only exception is the SGD experiment, where we vary the learning rate. The analysis shows that the biggest advantage comes from multiple positions and intermediate layers losses. Predicting all the intermediate positions leads to significant speed up in convergence, since the model sees more effective training examples per batch. Adding losses at the intermediate layers acts in the same spirit by forcing more predictions per training step.

Finally, we replace momentum with SGD as our optimizer, using a range of learning rates (0.3, 0.1, 0.03, 0.01, 0.003, 0.001). This ablation shows that SGD produces competitive models, with learning rate 0.1 giving the best performance. Despite the depth of our network, SGD is able to train the network efficiently with the help of our auxiliary losses.

4.2 Qualitative Analysis

To probe the strengths and weaknesses of our best model (T64), we run the model forward, starting with the sequence of 512 characters in Figure 5, taken from the `text8` test set. Figure 6 shows several per-character metrics for the model's predictions over the true continuation of this seed text. At each position, we measure i) the model's prediction entropy in bits across all 256 output classes, ii) its loss—the negative log probability of the target label, i.e. the "bits per character" for this position, and iii) the rank of the target in the list of output classes sorted by likelihood. Unsurprisingly, the model is least certain when predicting the first character of a word, and becomes progressively more confident and correct as subsequent characters are seen.

> *mary was not permitted to see them or to speak in her own defence at the tribunal she refused to offer a written defence unless elizabeth would guarantee a verdict of not guilty which elizabeth would not do although the casket letters were accepted by the inquiry as genuine after a study of the handwriting and of the information contained therein and were generally held to be certain proof of guilt if authentic the inquiry reached the conclusion that nothing was proven from the start this could have been pr*

Figure 5: A seed sequence of 512 characters taken from `text8` test set.

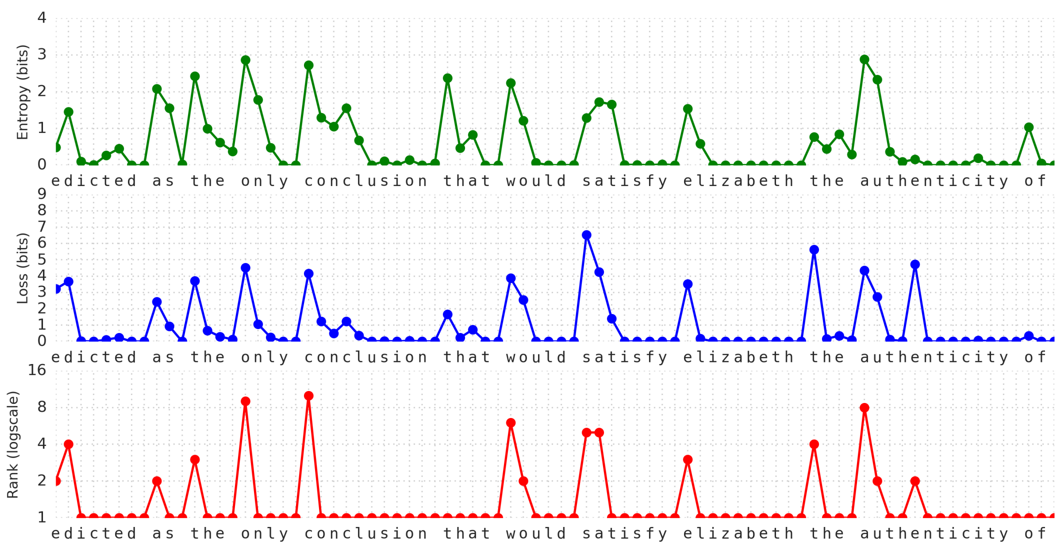


Figure 6: Per-character entropy, loss and rank assigned by T64 after seeding on the 512 character sequence from Figure 5.

To investigate the degree to which our model prefers actual English words over non-existent words, we compute the likelihood the model assigns to all continuations after the seed. We cut off continuations when they reach a space character, or when the total probability of the continuation falls below 0.001. Figure 7 shows the entire set of word completions, in order of probability, where the initial `pr-` from the seed is repeated for readability. Note that these are all real or plausible (`proofed`) English words, and that even short but bad continuations like `prz` are assigned a lower cumulative probability than long realistic word completions like `predictable`.

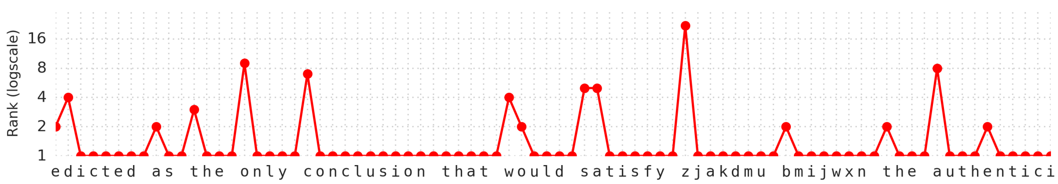
We expect that the transformer self-attention should make it easy for our model to copy sequences observed in the context over long distances (up to the context size of 512 characters). To test this expectation, we corrupt the seed and continuation from above by introducing a fake name `zjakdmu bmijwxn`. Specifically, we change the first occurrence of `elizabeth` in the seed to `zjakdmu` *proven, proved, proof, prevented, presented, problematic, probably, provided, practical, provoked, preceded, predicted, previously, presumed, praised, proposed, practicable, produced, present, preserved, precisely, prior, protected, probable, prompted, proofed, properly, practiced, prohibited, profound, preferable, proceeded, precise, predictable, practically, prevalent*

Figure 7: All word completions assigned cumulative probability above 0.001 to follow the seed from Figure 5, in order from most likely (0.529) to least likely (0.001).

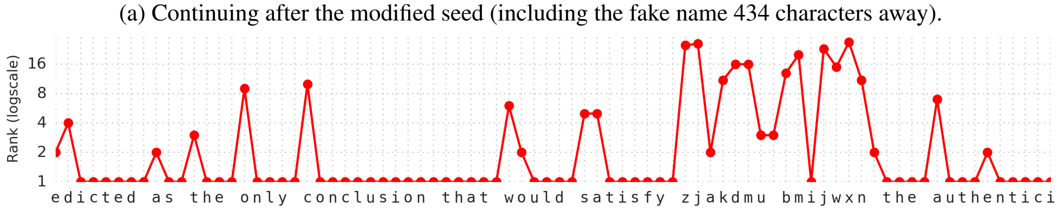
`bmijwxn`, and the second occurrence to `she`. Similarly, in the continuation, we change `elizabeth` to `zjakdmu bmijwxn`. The resulting distance between the two occurrences of the fake name is 434 characters.

Figure 8a confirms that the model can successfully copy over this long distance. While the initial `z` in `zjakdmu` is unexpected, the model immediately chooses to copy the remainder of this word from the context, as opposed to predicting any real `z-` words learned during training. Similarly, while the model is somewhat unsure whether the fake surname `bmijwxn` will appear (assigning the initial `b` a rank of two), it immediately picks up on the correspondence after the `b` is observed, correctly predicting the remainder of the fake surname.

For comparison, Figure 8b shows how the model would rank the targets in our fake continuation if the original seed with `elizabeth` were used. This confirms that the fake name is not predictable based on knowledge gained through training, and is indeed being copied from the preceding context.



(a) Continuing after the modified seed (including the fake name 434 characters away).



(b) Continuing after the original seed from Figure 5.

Figure 8: Per-character rank assigned by T64 to a fake continuation, after being seeded on either (a) the fake context where `elizabeth` is replaced with `zjakdmu bmijwxn`, or (b) the original context.

5 Related Work

Character-level modeling has shown promise in many areas such as sentiment analysis [35], question answering [21] and classification [50], and is an exciting area due to its simplicity and the ability to easily adapt to other languages. Neural network based language modeling has been heavily researched since its effectiveness was shown by Bengio et al. in [2]. By far, the most popular architecture in this area is the RNN and variants, first studied in [32].

Many authors explore RNN cell designs, such as Multiplicative Integration [46], MuFuRU [43], and Hypernetworks [13], including automatically searching for superior RNN cell designs [52]. Others experiment with modifying how the RNN cells are interconnected [6, 4]. Giving RNNs access to external memory has also shown improvements in the language modeling domain [17, 47].

Much of the progress in this area has been made by mitigating the vanishing gradients problem [14] by architectures such as LSTMs [15], GRU [3], Recurrent Highway Networks [51], Unitary RNNs [1] and others. This is an issue that transformers do not have, due to attention allowing short paths to all inputs. Methods of normalizing activation functions, such as Batch Normalization [16, 30] and Layer Normalization [27] have also demonstrated improvements on language modeling tasks. As with this work, progress has been made with discovering ways to regularize sequential architectures, with techniques such as Recurrent Dropout [48, 11] and Zoneout [26, 36].

A closely related architecture is the Neural Cache Model [12], where the RNN is allowed to attend to all of its previous hidden states at each step. Another similar model is used in [9] where a key-value attention mechanism similar to transformers is used. Both approaches show improvements on word level language modeling. Memory Networks [45] have a similarity to the transformer model in design as it also has layers of attention for processing a fix memory representing the input document and has been shown to be effective for language modeling in [39]. ByteNet [19], which is related but uses layers of dilated convolutions rather than attention, showed promising results on byte level language modeling. Gated Convolutional Networks [10] was an early non-recurrent model to show superior performance on word level language modelling.

Language models are not usually very deep due to computational constraints of training RNNs, and this also limits the number of parameters. The transformer architecture allowed us to build very deep (64 layer) models with a large number of parameters. A recent CNN model for Text Classification [7] at 29 layers is considered deep in the NLP community. A Sparsely-Gated Mixture-of-Experts Layer [38] allowed language modeling experiments with a greatly increased number of parameters by only accessing a small portion of parameters every time step, showing a reduction in bits per word. In Exploring the Limits of Language Modeling [18], an increase in the number of parameters was achieved by mixing character-level and word level models, using specialized softmaxes and using a large amount of computational resources to train. IndRNN [28] uses a simplified RNN architecture that allows deeper stacking with 21-layers, achieving near SOTA character-level language modeling. Fast-Slow Recurrent Neural Networks [34] also achieved near SOTA by increasing the number of recurrent steps for each character processed.

6 Conclusion

Character language modeling has been dominated by recurrent network approaches. In this paper, we show that a network of 12 stacked transformer layers achieves state-of-the-art results on this task. We gain further improvements in quality by deepening the network to 64 layers, utilizing capacity and depth efficiently. The use of auxiliary losses at intermediate layers and positions is critical for reaching this performance, and these losses allow us to train much deeper transformer networks. Finally, we analyze the behavior of our network and find that it is able to exploit dependencies in structure and content over long distances, over 400 characters apart.

Acknowledgements

For useful discussion and comments, we are grateful to Orhan Firat, Łukasz Kaiser, Slav Petrov, Noam Shazeer and Jakob Uszkoreit.

References

[1] Martín Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. *CoRR*, abs/1511.06464, 2015.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[3] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[4] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

[5] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, pages 2067–2075, 2015.

[6] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. *CoRR*, abs/1502.02367, 2015.

[7] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann LeCun. Very deep convolutional networks for natural language processing. *CoRR*, abs/1606.01781, 2016.

[8] Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron C. Courville. Recurrent batch normalization. *CoRR*, abs/1603.09025, 2016.

[9] Michal Daniluk, Tim Rocktäschel, Johannes Welbl, and Sebastian Riedel. Frustratingly short attention spans in neural language modeling. *CoRR*, abs/1702.04521, 2017.

[10] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *CoRR*, abs/1612.08083, 2016.

[11] Y. Gal and Z. Ghahramani. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. *ArXiv e-prints*, December 2015.

[12] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *CoRR*, abs/1612.04426, 2016.

[13] David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. *CoRR*, abs/1609.09106, 2016.

[14] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[15] Sepp Hochreiter and Jrgen Schmidhuber. Long short-term memory. 9:1735–80, 12 1997.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[17] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *CoRR*, abs/1503.01007, 2015.

[18] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[19] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[20] Nan Rosemary Ke, Anirudh Goyal, Olexa Bilaniuk, Jonathan Binas, Laurent Charlin, Chris Pal, and Yoshua Bengio. Sparse attentive backtracking: Long-range credit assignment in recurrent networks. *arXiv preprint arXiv:1711.02326*, 2017.

[21] Tom Kenter, Llion Jones, and Daniel Hewlett. Byte-level machine reading across morphologically varied languages. 2018.

[22] Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. Sharp nearby, fuzzy far away: How neural language models use context. In *Association for Computational Linguistics (ACL)*, 2018.

[23] Bryon Knol. cmix - http://www.byronknoll.com/cmix.html, 2017.

[24] Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. *arXiv preprint arXiv:1709.07432*, 2017.

[25] Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

[26] David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Chris Pal. Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*, 2016.

[27] J. Lei Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *ArXiv e-prints*, July 2016.

[28] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building A longer and deeper RNN. *CoRR*, abs/1803.04831, 2018.

[29] MultiMedia LLC. Large text compression benchmark, 2009.

[30] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *CoRR*, abs/1708.02182, 2017.

[31] T. Mikolov, S. Kombrink, L. Burget, J. ernock, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531, May 2011.

[32] Tomas Mikolov, Martin Karafit, Luks Burget, Jan Cernock, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH*, pages 1045–1048. ISCA, 2010.

[33] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8, 2012.

[34] Asier Mujika, Florian Meier, and Angelika Steger. Fast-slow recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 5915–5924, 2017.

[35] Alec Radford, Rafal Józefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment. *CoRR*, abs/1704.01444, 2017.

[36] Kamil M Rocki. Surprisal-driven feedback in recurrent networks. *arXiv preprint arXiv:1608.06027*, 2016.

[37] Tim Salimans, Han Zhang, Alec Radford, and Dimitris N. Metaxas. Improving gans using optimal transport. *CoRR*, abs/1803.05573, 2018.

[38] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

[39] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448, 2015.

[40] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[41] Corentin Tallec and Yann Ollivier. Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*, 2017.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[43] Dirk Weissenborn and Tim Rocktäschel. Mufuru: The multi-function recurrent unit. *CoRR*, abs/1606.03002, 2016.

[44] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[45] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014.

[46] Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Advances in neural information processing systems*, pages 2856–2864, 2016.

[47] Dani Yogatama, Yishu Miao, Gabor Melis, Wang Ling, Adhiguna Kuncoro, Chris Dyer, and Phil Blunsom. Memory architectures in recurrent neural network language models. 2018.

[48] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[49] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan R Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1822–1830, 2016.

[50] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *CoRR*, abs/1509.01626, 2015.

[51] Julian G. Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *CoRR*, abs/1607.03474, 2016.

[52] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.