GPT-3 as a probability model

Andrei-Tiberiu Alexandru

January 2022

1 Introduction

Generative pre-trained transformers (GPTs) are a type of neural network for sequence modelling that is based on the transformer architecture. Recently, GPT-3, a large transformer model of around 175 billion parameters displayed state-of-the-art performance on several language tasks, and showcased the ability to generate coherent and convincing writing on different topics[BMR+20]. Framing text generation as a probability modelling task reveals that GPTs largely model the data in the same way as another class of network: the recurrent neural network (RNN). And yet, in practice, GPTs seem to outperform RNNs of comparable size. My hypothesis is that GPTs are better at fitting language data than RNNs because they have a useful inductive bias. That is, a particular quality of the transformer mechanism, one not found in the RNN, makes it easier to learn language data.

I believe this quality is related to how the two architectures use context to produce the next token in a sequence. RNNs summarise the sequence into a history digest at each step, using a recurrence relation to generate the next token. Transformers do not use recurrence, instead employing positional embeddings coupled with an attention mechanism. The probability distributions these mechanisms induce over the token to be predicted are conceptually the same, but in practice the types of distributions GPT finds are more like natural language than the ones RNNs do.

One scenario in which the difference between attention and history digests is salient is data with long-term dependencies. Intuitively, long-term, or non-local dependencies occur when one token in a sequence depends on a previous token that is not immediately adjacent. This type of non-local dependency is widely observed in language, where there are relationships between, e.g. a verb and an object ("I ate steak" vs "I ate a tender, dry-aged sirloin steak"). This ability to model distant dependencies better may be the source of the increased performance of GPTs over RNNs on language data.

In this paper, I propose a probabilistic interpretation of the GPT, contrasting it with that of an RNN. I also illustrate the behaviour of both on a toy dataset that contains non-local dependencies, and interpret the inductive bias hypothesis in light of the results.

2 Background

Sequence modelling is the general task of mapping an input sequence to an output sequence. There are many domains which fall under this category, like time series prediction, text or music generation, or even sequences of actions of a reinforcement learning agent. In the case of text prediction, the output sequence is a continuation of the input sequence, with one or more tokens – words or characters – having been appended. In general, the probability of a sequence is:

$$\begin{aligned}\Pr(\mathbf{x}) &= \Pr(X_1 X_2 \dots X_n) \\ &= \Pr_{X_1}(x_1)\Pr_{X_2}(x_2|X_1 = x_1) \dots \Pr_{X_t}(x_t|X_1 = x_1, X_2 = x_2, \dots, X_{t-1} = x_{t-1})\end{aligned} \tag{1}$$

One simplifying assumption that can be made is that the probability of each token is only dependent on the $n-1$ tokens that precede it, where n is less than or equal to the total number of tokens. This type of model is called an n-gram; for example, if we used a bigram model where only the token immediately preceding the token to be predicted is used, Equation 1 becomes:

$$\begin{aligned}\Pr(\mathbf{x}) &= \Pr(X_1 X_2 \dots X_n) \\ &= \Pr_{X_1}(x_1) \cdot \Pr_{X_2}(x_2|X_1 = x_1)\Pr_{X_3}(x_3|X_2 = x_2) \dots \Pr_{X_t}(x_t|X_{t-1} = x_{t-1})\end{aligned} \tag{2}$$

This is useful because the probabilities of pairs or triplets are much larger than of a sequence of many more tokens. (The joint probability of two random variables A and B is always larger than the joint probability of A and B and another random variable C). On the other hand, by only looking at a handful of previous tokens, everything else is discarded. By ignoring part of the context, the model loses information that could make the prediction more accurate. We would like a way to encode context without having to choose a size in advance.

This is where recurrent neural networks (RNNs) come in: instead of specifying what the context should be, a neural network is used to learn it from the data. In other words, RNNs are useful for this type of task because of their ability to represent contextual information. During training, an RNN receives sequences one token at a time. Given the token and a previous hidden state – a summary of the history the network has seen – the network outputs a new hidden state and a probability distribution over tokens. The next token is generated by sampling from this probability distribution, and the process is repeated until a special token is generated that marks the end of the sequence, or until stopped by the user.

Training an RNN is conceptually equivalent to maximum likelihood estimation. The network parameters are adjusted via gradient descent to minimise a loss function that corresponds to the negative log-likelihood of the data. One problem with RNNs is that as the sequence length increases, it becomes more and more difficult to propagate the loss back through the network due to decaying gradients. Depending on the size of the weights, gradients either explode or vanish [BSF94], which results in different failure scenarios, but just the same in a poorly-trained network. When predicting a token at time step t, if there is a dependency on a token at timestep $\tau \ll t$, with vanishing gradients it is difficult to use the token x_τ to predict x_t.

[HBF+01] summarises the difficulty with learning these long-term dependencies and proposes a set of possible remedies. Of those, only one is widely used today: the long short-term memory (LSTM) method[HS97]. In an LSTM, the simple nonlinearity in a RNN is replaced with a memory cell, which is designed to facilitate remembering across longer sequences. The architecture specifics aren't particularly relevant here; it suffices to know that the LSTM indeed outperforms a standard RNN empirically on different types of tasks. For example, [Gra13] applies it to text prediction on Wikipedia. This application is interesting because articles tend to introduce key information first, and then reference it later in the article. The author notes that although the capacity to memorise is markedly better (and although a larger network and more training could improve it even further), it seems unrealistic to expect coherence beyond short sentences. This is because the network hasn't experienced the concepts it is referring to — it knows of them, but does not understand them.

Machine translation is also a sequence modelling problem, where one input sequence in language A is translated to an output sequence in language B. There are several difficulties in machine translation. For example, the sequences may not be the same length (some languages are more verbose) or the order of the words is not the same (in French, adjectives go after a noun). One way to overcome these obstacles is to use RNNs in an encoder-decoder architecture [CVMG+14], where the input is first encoded into a context vector, then decoded into the target sequence.

[BCB14] discuss the limitations of an encoder-decoder with a fixed-size context vector: longer sequences are harder to encode into a single finite representation without losing information. Their solution works as follows. The encoder computes for each token x_j an annotation h_j, whose role is to summarise the rest of the input, not including the current token. To summarise both preceding and following tokens, the encoder uses a bidirectional RNN which reads the input sequence left-to-right, and again in reverse order. Given a set of annotations, the decoder then generates context vectors, one for each token, by weighting the annotations by some weight α_{ij}. These weights are part of an alignment model that is parameterised as a feedforward neural network, and essentially measure how well an input and an output token match.

This scoring is a form of attention, which is one of the defining features in another type of architecture, the transformer [VSP+17]. Another notable characteristic of the transformer is that it does not use recurrence to model sequential data. Instead, transformers use positional embeddings and self-attention to determine which parts of the input are relevant for predicting the next token. This is an advantage over RNNs: recurrence means that computation can be parallelised, which improves computational efficiency. Although the use of attention generates sequences which humans rate as more realistic, large-scale coherence is still a problem, so the earlier claim about language being grounded in sensory experience may have some credit.

3 Generative pre-trained transformers

The generative pre-trained transformer (GPT) is a variant of the transformer used for text generation. GPTs arose due to the scarcity of labelled data in downstream language tasks, which they circumvent by first training on large corpora of unlabelled text like CommonCrawl[RSR+19] and then fine-tune on the small datasets specific to the task to solve – a form of transfer learning. The authors note that the attention structure of the GPT helps transfer the learning from the initial task to the downstream task better than an LSTM would, and substantiate this with empirical ablation studies[RNSS18], which might hint at the idea that a network that attention simply handles longer sequences better.

The initial unsupervised pre-training objective is:

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta) \tag{3}$$

Where k is the size of a context window and the conditional probability P is modelled using a neural network parameterised by Θ. Then, the supervised fine-tuning is done given a labelled dataset consisting of pairs of input tokens $x_1, x_2, \ldots, x_m$ and a label y. Its objective is:

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y | x_1, \ldots, x_m) \tag{4}$$

Where the probability P is obtained by applying softmax to the output of the last transformer block multiplied by an added linear layer W_y:

$$P(y | x_1, \ldots, x_m) = \text{softmax}(h_m^l W_y) \tag{5}$$

L_1 and L_2 are combined into a final objective L_3, which the network actually optimises:

$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda \cdot L_1(\mathcal{C}) \tag{6}$$

These descriptions are from the GPT-1, the first iteration of the model[RNSS18]. As far as I can tell, the only modification in later versions of GPT has been that fine-tuning has been altogether eliminated leaving just the unsupervised learning objective L_1. Starting with GPT-2[RWC+19], fine-tuning is largely replaced by mechanism termed 'few-shot learning'. A simple way of thinking of few-shot learning is as learning from examples. This breaks down further into:

- **Few-shot learning**: given multiple examples, carry out a task described in the prompt (e.g. "here are three summaries of "Infinite Jest" by David Foster Wallace, generate an additional one");
- **One-shot learning**: given one example, carry out a task (e.g. "here is exactly one summary of the novel, generate an additional one");
- **Zero-shot learning**: without being given any examples, carry out a task described in language (e.g. "generate a summary of this novel");

These regimes can be thought of as an even more extreme version of the earlier fine-tuning scenario, where data was scarce. Zero-shot learning is naturally the most challenging: without any additional data, we want to get a language model to perform a downstream task. That is, we want to find:

$$Pr(c_{t+1} | c_1, \ldots, c_t, X) = Pr(c_{t+1} | c_1, \ldots, c_t; \Theta) \tag{7}$$

We can discard X, the original data, because the information about is encoded in our parameters, which cannot change. This is more like evaluating on a test example than fitting a model to a training example! What accounts for the ability of GPT to "learn" just from the context of a prompt, without modifying its weights is still an open question.

In terms of architecture, except for the self-attention, every other layer in GPT is in use in most neural networks: embeddings, dropout, fully-connected layers, layer normalisation, and a final fully-connected layer to project the output to the vocabulary size. An interesting note is that GPT also uses skip connections: some of the layers compute `x = x + some_fn(x)` instead of `x = some_fn(x)`, the idea being that it makes the network easier to train using gradient descent.

Presumably, the magic happens in the attention layer. But peering in, this is made up of the same building blocks as other layers: matrix multiplications. The point of attention layers is to attend to different parts of the sequence and see which are most relevant – by some measure of relevance. Each of these parts is scored using a method called scaled dot-product similarity, and the attention scores are propagated further into the network. This happens multiple times in parallel using several attention "heads", which look at different parts of the input.

Although it's hard to find a probabilistic interpretation of the attention mechanism, it's likely that the effect it has on the resulting probability distribution over the vocabulary is governed by its ability to handle long sequences. If an RNN has trouble with non-local dependencies, it will have a hard time understanding motifs, writing styles, summaries, references. It's like being able to remember only a few sentences at a time; from there, it's hard to extrapolate what an entire article or book should be like.

A lot of what GPT can do that RNNs seem to struggle with boils down to being able to remember very long sequences well. In the few-shot learning regime, it would be impossible to learn anything meaningful from context if the context size were limited to only a few words. For example, it would be impossible to do text summarisation if the network could not "see" the full text.

That said, the size of the language model seems to have a big impact on how well it does in the few-shot regime. In [BMR+20], GPT-3 is tested on the one of the classical tasks in NLP: predicting which word a pronoun refers to [SLBBC20]. The experiments show that in all three few-shot learning settings, a GPT of around 100 million parameters is only a few percentage points over the accuracy for random guessing. The recipe for success seems to be that attention scales really well.

4 Experiments

The idea of these experiments is to investigate how well the two types architectures (GPTs and RNNs) manage non-local dependencies. To do so, I'm looking at 2-digit integer addition. My data set consists of randomly generated permutations of 2-digit numbers (e.g. 63 and 21), where the ground truth is generated by adding the two numbers (= 84). The numbers are represented as sequences of digits: `[6, 3, 2, 1, 0, 8, 4]` with an extra zero to account for the possibility of the sum being a 3-digit number. Then, the network is trained on all but the last digit, and asked to predict this last digit. During testing, the prompts are only the first two numbers, and the network predicts the result, one digit at a time (i.e. given 4 digits, it predicts a 5th, which is then fed back in as another input, etc.)

I consider a sequence of `[6, 3, 2, 1, 0, 8, 4]` to not contain any non-local dependencies: all the relevant information is contained in the first four digits. To artificially create long-term dependencies, I'll add some "decoy" numbers between the two numbers that are actually to be summed, to see if this can distract the network. With one decoy, or a sequence length of 3, the above becomes `[6, 3, 0, 7, 2, 1, 0, 8, 4]`. We would like the network to still be able to predict 84 even though there is a 7 between the two terms to be summed.

The experiments look at the accuracy of GPTs and RNNs at predicting the three digits that make up the sum as more and more decoys are added. If the results supported the hypothesis that GPTs do better compared to RNNs in this scenario, we would see a drop in their performance that is less abrupt than the RNNs'. I expect the accuracy to drop across models regardless, as these dependencies are more challenging than the base case.

Experiments are run on networks with an approximately equal number of parameters. The smaller models are around 200,000 parameters, and the larger ones are 400,000. I run four types of networks for 20 and 50 epochs respectively, on exactly the same data. (The random dataset is generated deterministically using a seed.) There is one configuration for GPT that matches the implementation in the papers, with the code due to [Fal21, Kar21]. I use three RNNs: one with a default tanh nonlinearity, an LSTM network and a GRU network. For details on these architectures please see the code repo[1]. The experiments are run on a HPC comprising 10 AMD Epyc vCPUs, 2 NVIDIA A40 GPUs and 64Gb of RAM. I used Pytorch Lightning[FT19] for training using the DDP back-end and 16-bit floating-point precision.

[1] https://github.com/inwaves/prob-models/

5 Preliminary results

5.1 Full-size models

Table 1: Prediction accuracy for full-size models

2-digit addition		n = 2	n = 3	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9	n = 10
GPT (20 epochs)	**Training**	99.83%	99.07%	99.48%	82.58%	64.07%	27.25%	34.42%	25.50%	16.71%
	Test	**99.70%**	**99.23%**	**99.30%**	82.47%	61.20%	**26.66%**	**33.62%**	**24.47%**	**15.59%**
RNN (20 epochs)	**Training**	99.52%	98.55%	96.77%	84.80%	68.19%	6.41%	2.73%	4.31%	1.83%
	Test	99.20%	98.10%	96.07%	**84.67%**	**66.94%**	6.26%	2.37%	3.49%	1.74%
LSTM (20 epochs)	**Training**	99.17%	34.25%	29.37%	21.44%	11.46%	7.09%	0.93%	7.59%	1.06%
	Test	98.70%	28.53%	25.37%	17.47%	7.27%	5.43%	0.80%	6.17%	0.67%
GRU (20 epochs)	**Training**	99.71%	48.43%	59.34%	24.80%	21.41%	18.38%	16.46%	14.60%	12.44%
	Test	99.50%	41.23%	55.07%	19.17%	15.23%	12.57%	11.80%	10.40%	9.03%
GPT (50 epochs)	**Training**	100.00%	99.65%	99.99%	99.88%	100.00%	99.96%	99.80%	99.95%	99.59%
	Test	**100.00%**	**99.63%**	**99.90%**	**99.97%**	**100.00%**	**100.00%**	**99.79%**	**99.98%**	**99.55%**
RNN (50 epochs)	**Training**	100.00%	99.82%	99.64%	97.99%	94.91%	25.16%	24.15%	21.07%	14.58%
	Test	100.00%	99.47%	99.50%	97.90%	94.01%	18.86%	18.43%	15.12%	10.32%
LSTM (50 epochs)	**Training**	100.00%	62.92%	65.82%	49.57%	31.08%	18.05%	15.77%	13.63%	0.96%
	Test	99.80%	55.43%	59.87%	43.37%	21.57%	11.27%	10.50%	8.53%	0.70%
GRU (50 epochs)	**Training**	100.00%	87.20%	96.43%	58.87%	41.68%	39.24%	37.20%	33.39%	33.74%
	Test	100.00%	79.70%	92.97%	47.67%	29.70%	27.67%	25.80%	24.00%	23.47%

Table 1 reports the results of the experiments for sequence lengths (n) ranging from 2-10. A sequence length of 2 means there are no decoys in the input – just the two numbers to add – whereas $n = 10$ means there were 8 numbers that were not included in the sum. What is immediately obvious from all runs is that there is a decrease in accuracy as the sequence length increases. I interpret this to mean that the decoys are working — they're throwing off the network's ability to predict the sum, even though the terms of the sum are always the first and last numbers in the sequence. To a human, this is an obvious pattern, but perhaps as the sequence length or number of digits are increased, we would have trouble noticing it too. The decrease in accuracy is plotted in Figure 1.

The second observation is that performance for the GPT and RNN decrease almost in lockstep until n = 6, with the RNN seemingly outperforming for n = 5 and 6, but not by much. From n = 7 onward however, both models' accuracies drop off abruptly, from 61% to 26% for the GPT and 66% to 6% for the RNN. The latter seems to lose accuracy faster, even though there is no particularity of the implementation that should correlate with that exact sequence length.

Some results were unexpected: sequence length 6 onward for the RNN and n = 7 onward for GPT seemed to show abrupt decreases in accuracy. This behaviour was consistent over multiple repetitions of those experiments. There is nothing about the implementations that would warrant such a sudden drop, but it's likely that as n increases network just need more time to converge to 100% training set accuracy.

At 50 epochs, the GPT converges for all n and its performance almost does not decrease at all as we vary the sequence length. The RNN's performance is also more stable, maintaining $> 94\%$ accuracy until sequence length 6. From $n = 7$ onward, the accuracy abruptly decreases to 18% on the test set – a less dramatic decrease than on 20 epochs, but in stark contrast to the GPT nonetheless.

The LSTM does very poorly when trained for only 20 epochs, and seems to be more robust during 50-epoch runs. The accuracy of the GRU sits somewhere in-between the LSTM and the RNN, and exhibits the same decrease in performance as the two.

5.2 Half-size models

Table 2 reports the behaviour for half-size models (∼200k parameters each), which is broadly the same as the full-sized models. There is a relatively steady drop in performance up to a point, where the accuracy completely drops off to $< 10\%$. Interestingly, at 20 epochs the GPT seems to be more robust when half-size, maintaining $> 89\%$ accuracy up to sequence length 7, then jumping to only 7%. The RNN is less robust, with relatively accurate predictions up to $n = 5$, then nonsense (the full-sized model still output reasonable predictions for $n = 6$).

At 50 epochs, the half-sized GPT is less robust than the full-size: predictions for $n = 9$ and 10 are not at all accurate. The half-sized RNN is much less robust than its full-size counterpart, with predictions from $n = 6$ onward being $< 1\%$ accurate, where previously they were at least 10%

Table 2: Prediction accuracy for half-size models

2-digit addition		n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
GPT (20 epochs)	**Training**	86.52%	98.47%	93.39%	86.36%	97.30%	90.04%	8.04%	8.76%	6.35%
	Test	86.80%	**98.70%**	94.33%	86.33%	**97.37%**	**89.33%**	**7.43%**	**8.67%**	**6.27%**
RNN (20 epochs)	**Training**	99.72%	96.78%	95.25%	88.90%	0.92%	0.93%	0.90%	0.95%	0.89%
	Test	**99.80%**	96.33%	**94.43%**	**89.77%**	0.83%	0.80%	0.83%	0.90%	0.83%
LSTM (20 epochs)	**Training**	98.25%	30.80%	23.55%	18.22%	9.50%	8.00%	0.86%	5.19%	0.99%
	Test	97.40%	26.60%	19.77%	13.93%	7.93%	6.93%	0.80%	4.03%	0.67%
GRU (20 epochs)	**Training**	99.72%	39.90%	27.21%	17.81%	15.50%	13.36%	11.27%	7.89%	1.01%
	Test	99.20%	36.13%	21.93%	13.37%	11.57%	9.57%	7.33%	6.63%	0.70%
GPT (50 epochs)	**Training**	99.32%	99.02%	99.41%	100.00%	97.33%	96.06%	99.80%	7.91%	10.09%
	Test	99.80%	99.03%	**99.63%**	**100.00%**	**97.70%**	**96.30%**	**100.00%**	**9.13%**	**9.37%**
RNN (50 epochs)	**Training**	100.00%	100.00%	99.57%	96.55%	0.89%	0.90%	0.99%	0.92%	9.37%
	Test	**100.00%**	**99.93%**	98.97%	96.23%	0.67%	0.70%	0.60%	0.73%	0.77%
LSTM (50 epochs)	**Training**	99.98%	48.21%	23.55%	18.22%	9.50%	8.00%	0.86%	5.19%	0.99%
	Test	99.80%	26.60%	19.77%	13.93%	7.93%	6.93%	0.80%	4.03%	0.67%
GRU (50 epochs)	**Training**	100.00%	75.15%	59.32%	27.90%	27.33%	22.13%	21.41%	10.04%	0.94%
	Test	**100.00%**	69.17%	52.13%	21.07%	19.23%	15.70%	14.80%	7.17%	0.60%

accurate. The LSTM follows the same behaviour with an abrupt drop in accuracy early in the sequence progression. Figure 2 plots the evolution of accuracy as sequence length is increased.

6 Discussion

The preliminary results seem to support the hypothesis that attention indeed confers the ability to remember and use more of the tokens in a long sequence. All networks were presented with inputs of the same size, but after a point the RNNs are unable to use the first two tokens of the sequence, and cannot compute the sum. It seems like there is a hard cut-off for the RNNs – somewhere around $n = 6$ – after which they just cannot predict correctly. This isn't the case; experiments with much larger RNNs ($\sim$1,000,000 parameters) trained for longer show that they *can* achieve similar performance as GPT, but they need more computation. This makes the difference between the two types of architectures quantitative: an RNN can generate the types of sequences GPT does, just not as easily.

So a key point of the performance GPTs achieve in the few-shot context is likely due to this bias that lets it extract information from large bodies of text. Still, this doesn't explain zero-shot prompts like "translate the word 'cheese' to French". Maybe this is possible because in the text in the data set, the words 'cheese' and 'French' were associated with 'fromage'. But this is a just-so explanation: they could just as well be surrounded by a recipe for an omelette, with no reference to the translation. How exactly prompting works is not yet known, and prompt engineering – designing prompts such that they elicit the answer we want – is emerging as an active area of research.

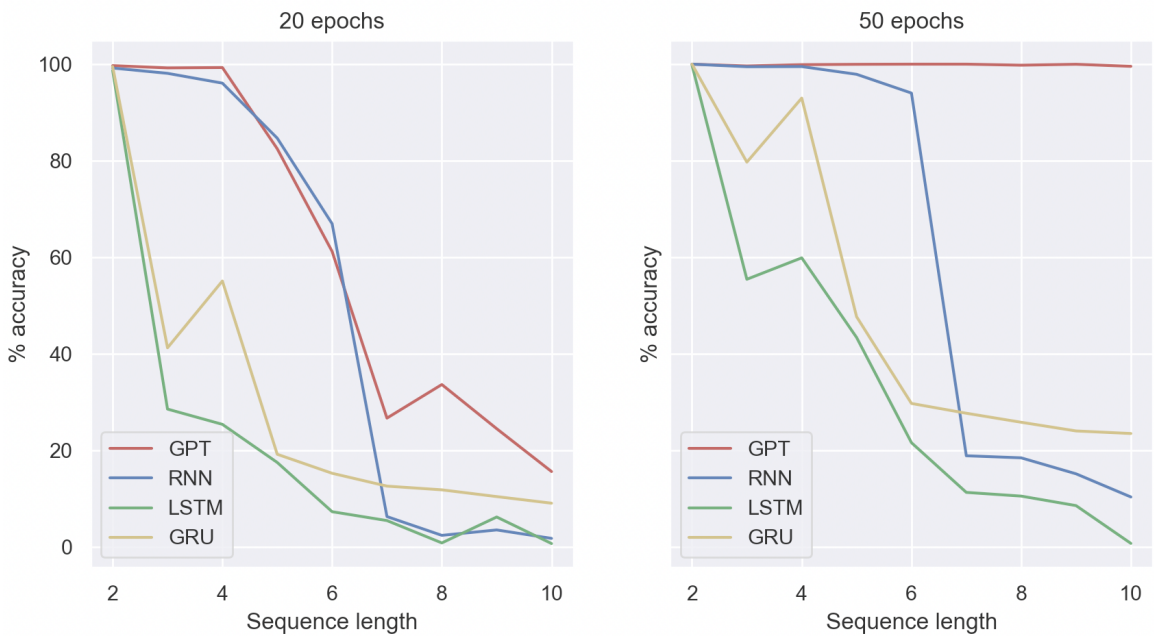

Figure 1: Test accuracy of full-size model as sequence length is increased.

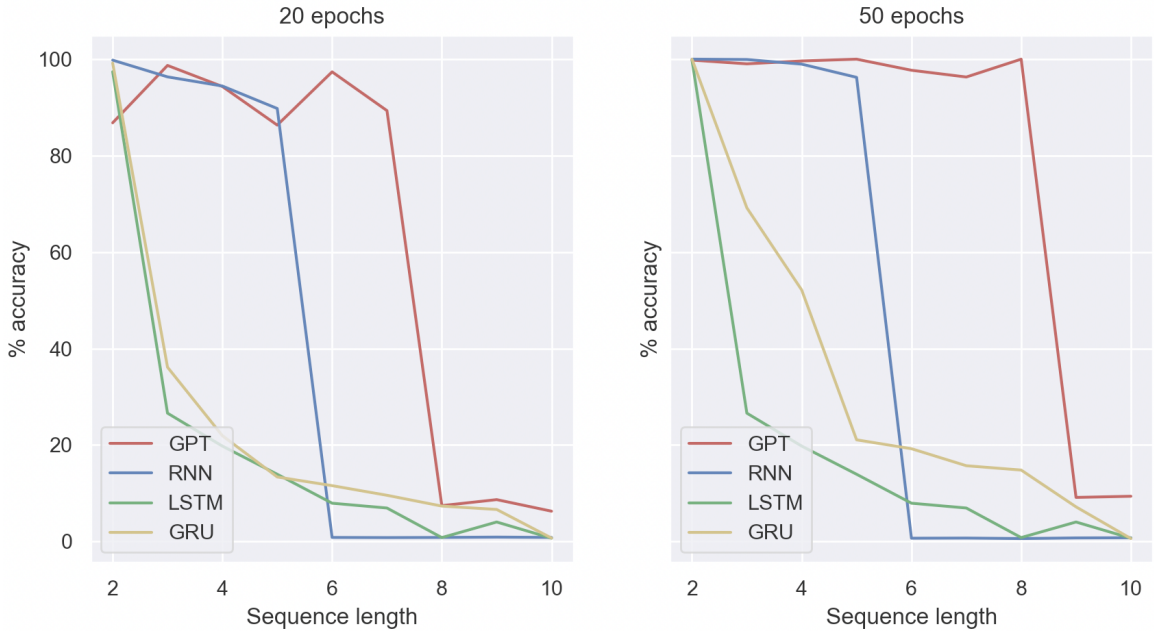

Figure 2: Test accuracy of half-size model as sequence length is increased.

References

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[BMR+20] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[Fal21] William Falcon. mingpt: A minimal pytorch lightning re-implementation of the openai gpt (generative pretrained transformer) training. https://github.com/williamFalcon/minGPT, 2021.

[FT19] William Falcon and The PyTorch Lightning team. PyTorch Lightning, 3 2019.

[Gra13] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[HBF+01] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[Kar21] Andrej Karpathy. mingpt: A minimal pytorch re-implementation of the openai gpt (generative pretrained transformer) training. https://github.com/karpathy/minGPT, 2021.

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[RSR+19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[SLBBC20] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8732–8740, 2020.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.