CONVOLUTIONAL SEQUENCE MODELING REVISITED

Shaojie Bai
Carnegie Mellon University
Pittsburgh, PA 15213, USA
shaojieb@cs.cmu.edu

J. Zico Kolter
Carnegie Mellon University
Pittsburgh, PA 15213, USA
zkolter@cs.cmu.edu

Vladlen Koltun
Intel Labs

ABSTRACT

Although both convolutional and recurrent architectures have a long history in sequence prediction, the current "default" mindset in much of the deep learning community is that generic sequence modeling is best handled using recurrent networks. Yet recent results indicate that convolutional architectures can outperform recurrent networks on tasks such as audio synthesis and machine translation. Given a new sequence modeling task or dataset, which architecture should a practitioner use? We conduct a systematic evaluation of generic convolutional and recurrent architectures for sequence modeling. In particular, the models are evaluated across a broad range of standard tasks that are commonly used to benchmark recurrent networks. Our results indicate that a simple convolutional architecture outperforms canonical recurrent networks such as LSTMs across a diverse range of tasks and datasets, while demonstrating longer effective memory. We further show that the potential "infinite memory" advantage that RNNs have over TCNs is largely absent in practice: TCNs indeed exhibit *longer* effective history sizes than their recurrent counterparts. As a whole, we argue that it may be time to (re)consider ConvNets as the default "go to" architecture for sequence modeling.

1 INTRODUCTION

Since the re-emergence of neural networks to the forefront of machine learning, two types of network architectures have played a pivotal role: the convolutional network, often used for vision and higher-dimensional input data; and the recurrent network, typically used for modeling sequential data. These two types of architectures have become so ingrained in modern deep learning that they can be viewed as constituting the "pillars" of deep learning approaches. This paper looks at the problem of *sequence modeling*, predicting how a sequence will evolve over time. This is a key problem in domains spanning audio, language modeling, music processing, time series forecasting, and many others. Although exceptions certainly exist in some domains, the current "default" thinking in the deep learning community is that these sequential tasks are best handled by some type of recurrent network. Our aim is to revisit this default thinking, and specifically ask whether modern convolutional architectures are in fact just as powerful for sequence modeling.

Before making the main claims of our paper, some history of convolutional and recurrent models for sequence modeling is useful. In the early history of neural networks, convolutional models were specifically proposed as a means of handling sequence data, the idea being that one could slide a 1-D convolutional filter over the data (and stack such layers together) to predict future elements of a sequence from past ones (Hinton, 1989; LeCun et al., 1995). Thus, the idea of using convolutional models for sequence modeling goes back to the beginning of convolutional architectures themselves. However, these models were subsequently largely abandoned for many sequence modeling tasks in favor of recurrent networks (Elman, 1990). The reasoning for this appears straightforward: while convolutional architectures have a limited ability to look back in time (i.e., their receptive field is limited by the size and layers of the filters), recurrent networks have no such limitation. Because recurrent networks propagate forward a hidden state, they are theoretically capable of *infinite memory*, the ability to make predictions based upon data that occurred arbitrarily long ago in the sequence. This possibility seems to be realized even moreso for the now-standard architectures of Long Short-Term Memory networks (LSTMs) (Hochreiter & Schmidhuber, 1997), or recent incarnations such as the Gated Recurrent Unit (GRU) (Cho et al., 2014); these architectures aim to avoid the "vanish-

ing gradient" challenge of traditional RNNs and appear to provide a means to actually realize this infinite memory.

Given the substantial limitations of convolutional architectures at the time that RNNs/LSTMs were initially proposed (when deep convolutional architectures were difficult to train, and strategies such as dilated convolutions had not reached widespread use), it is no surprise that CNNs fell out of favor to RNNs. While there have been a few notable examples in recent years of CNNs applied to sequence modeling (e.g., the WaveNet (van den Oord et al., 2016) and PixelCNN (Oord et al., 2016) architectures), the general "folk wisdom" of sequence modeling prevails, that the first avenue of attack for these problems should be some form of recurrent network.

The fundamental aim of this paper is to revisit this folk wisdom, and thereby make a counterclaim. We argue that with the tools of modern convolutional architectures at our disposal (namely the ability to train very deep networks via residual connections and other similar mechanisms, plus the ability to increase receptive field size via dilations), in fact convolutional architectures *typically* outperform recurrent architectures on sequence modeling tasks, *especially* (and perhaps somewhat surprisingly) on domains where a long effective history length is needed to make proper predictions.

This paper consists of two main contributions. First, we describe a generic temporal convolutional network (TCN) architecture that is applied across all tasks. This architecture is informed by recent research, but is deliberately kept simple, combining some of the best practices of modern convolutional architectures. Second, and more importantly, we extensively evaluate the TCN model versus alternative approaches on a wide variety of sequence modeling tasks, spanning many domains and datasets that have typically been the purview of recurrent models, including word- and character-level language modeling, polyphonic music prediction, and other baseline tasks commonly used to evaluate recurrent architectures. Although our baseline TCN can be outperformed by specialized (and typically highly-tuned) RNNs in some cases, for the majority of problems the TCN performs best, with minimal tuning on the architecture or the optimization. This paper also analyzes empirically the myth of "infinite memory" in RNNs, and shows that in practice, TCNs of similar size and complexity may actually demonstrate *longer* effective history sizes. Our chief claim in this paper is thus an empirical one: rather than presuming that RNNs will be the default best method for sequence modeling tasks, it may be time to (re)consider ConvNets as a natural starting point in sequence tasks.

2 Related Work

In this section we highlight some of the key innovations in the history of recurrent and convolutional architectures for sequence prediction.

Recurrent networks broadly refer to networks that maintain a vector of hidden activations, which are kept over time by propagating them through the network. The intuitive appeal of this approach is that the hidden state can act as a sort of "memory" of *everything* that has been seen so far in a sequence, without the need for keeping an explicit history. Unfortunately, such memory comes at a cost, and it is well-known that the naïve RNN architecture is difficult to train due to the exploding/vanishing gradient problem (Bengio et al., 1994).

A number of solutions have been proposed to address this issue. More than twenty years ago, Hochreiter & Schmidhuber (1997) introduced the now-ubiquitous Long Short-Term Memory (LSTM) which uses a set of gates to explicitly maintain memory cells that are propagated forward in time. Other solutions or refinements include a simplified variant of LSTM, the Gated Recurrent Unit (GRU) (Cho et al., 2014), peephole connections (Gers et al., 2002), Clockwork RNN (Koutnik et al., 2014) and recent works such as MI-RNN (Wu et al., 2016) and the Dilated RNN (Chang et al., 2017). Alternatively, several regularization techniques have been proposed to better train LSTMs, such as those based upon the properties of the RNN dynamical system (Pascanu et al., 2013); more recently, strategies such as Zoneout (Krueger et al., 2017) and AWD-LSTM (Merity et al., 2017) were also introduced to regularize LSTM in various ways, and have achieved exceptional results in the field of language modeling.

While it is frequently criticized as a seemingly "ad-hoc" architecture, LSTMs have proven to be extremely robust and is very hard to improve upon by other recurrent architectures, at least for general problems. Jozefowicz et al. (2015) concluded that if there were "architectures much better than the LSTM", then they were "not trivial to find". However, while they evaluated a variety of

recurrent architectures with different combinations of components via an evolutionary search, they did not consider architectures that were fundamentally different from the recurrent ones.

The history of convolutional architectures for time series is comparatively shorter, as they soon fell out of favor compared to recurrent architectures for these tasks, though are also seeing a resurgence in recent years. Waibel et al. (1989) and Bottou et al. (1990) studied the usage of time-delay networks (TDNNs) for sequences, one of the earliest local-connection-based networks in this domain. LeCun et al. (1995) then proposed and examined the usage of CNNs on time-series data, pointing out that the same kind of feature extraction used in images could work well on sequence modeling with convolutional filters. Recent years have seen a re-emergence of convolutional models for sequence data. Perhaps most notably, the WaveNet (van den Oord et al., 2016) applied a stacked convolutional architecture to model audio signals, using a combination of dilations (Yu & Koltun, 2016), skip connections, gating, and conditioning on context stacks; the WaveNet mode was additionally applied to a few other contexts, such as financial applications (Borovykh et al., 2017). Non-dilated gated convolutions have also been applied in the context of language modeling (Dauphin et al., 2017). And finally, convolutional models have seen a recent adoption in sequence to sequence modeling and machine translations applications, such as the ByteNet (Kalchbrenner et al., 2016) and ConvS2S architectures (Gehring et al., 2017).

Despite these successes, the general consensus of the deep learning community seems to be that RNNs (here meaning all RNNs including LSTM and its variants) are better suited to sequence modeling for two apparent reasons: 1) as discussed before, RNNs are theoretically capable of *infinite memory*; and 2) RNN models are inherently suitable for sequential inputs of varying length, whereas CNNs seem to be more appropriate in domains with fixed-size inputs (e.g., vision).

With this as the context, this paper reconsiders convolutional sequence modeling *in general*, first introducing a simple general-purpose convolutional sequence modeling architecture that can be applied in all the same scenarios as an RNN (the architecture acts as a "drop-in" replacement for RNNs of any kind). We then extensively evaluate the performance of the architecture on tasks from different domains, *focusing on domains and settings that have been used explicitly as applications and benchmarks for RNNs in the recent past*. With regard to the specific architectures mentioned above (e.g. WaveNet, ByteNet, gated convolutional language models), the primary goal here is to describe a simple, application-independent architecture that avoids much of the extra specialized components of these architectures (gating, complex residuals, context stacks, or the encoder-decoder architectures of seq2seq models), and keeps only the "standard" convolutional components from most image architectures, with the restriction that the convolutions be causal. In several cases we specifically compare the architecture with and without additional components (e.g., gating elements), and highlight that it does not seem to substantially improve performance of the architecture across domains. Thus, the primary goal of this paper is to provide a baseline architecture for convolutional sequence prediction tasks, and to evaluate the performance of this model across multiple domains.

3 Convolutional Sequence Modeling

In this section, we describe a generic architecture for convolutional sequence prediction, and generally refer to it as Temporal Convolution Networks (TCNs). We emphasize that we adopt this term not as a label for a truly new architecture, but as a simple descriptive term for this and similar architectures. The distinguishing characteristics of the TCN are that: 1) the convolutions in the architecture are *causal*, meaning that there is no information "leakage" between future and past; 2) the architecture can take a sequence of any length and map it to an output sequence of the same length, just as with an RNN. Beyond this, we emphasize how to build very long effective history sizes (i.e., the ability for the networks to look very far into the past to make a prediction) using a combination of very deep networks (augmented with residual layers) and dilated convolutions.

3.1 The Sequence Modeling Task

Before defining the network structure, we highlight the nature of the sequence modeling task. We suppose that we are given a sequence of inputs $x_0, \dots, x_T$, and we wish to predict some corresponding outputs $y_0, \dots, y_T$ at each time. The key constraint is that to predict the output y_t for some time t, we are constrained to only use those inputs that have been previously observed: $x_0, \dots, x_t$. For-

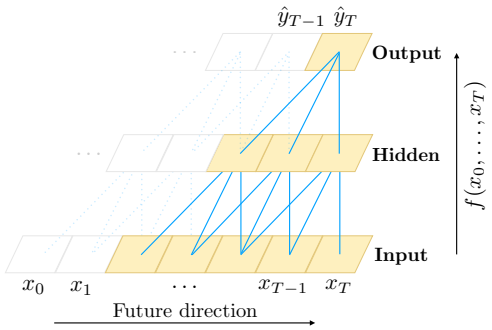


Figure 1: A simple causal convolution with filter size 3.

mally, a sequence modeling network is any function $f : \mathcal{X}^{T+1} \to \mathcal{Y}^{T+1}$ that produces this mapping

$$\hat{y}_0, \ldots, \hat{y}_T = f(x_0, \ldots, x_T) \tag{1}$$

if it satisfies the causal constraint that y_t depends only on $x_0, \ldots, x_t$, and not on any "future" inputs $x_{t+1}, \ldots, x_T$. The goal of learning in the sequence modeling setting is to find the network f minimizing some expected loss between the actual outputs and predictions $L(y_0, \ldots, y_T, f(x_0, \ldots, x_T))$ where the sequences and outputs are drawn according to some distribution.

This formalism encompasses many settings such as auto-regressive prediction (where we try to predict some signal given its past) by setting the target output to be simply the input shifted by one time step. It does *not*, however, directly capture domains such as machine translation, or sequence-to-sequence prediction in general, since in these cases the entire input sequence (including "future" states) can be used to predict each output (though the techniques can naturally be extended to work in such settings).

3.2 Causal Convolutions and the TCN

As mentioned above, the TCN is based upon two principles: the fact that the network produces an output of the same length as the input, and the fact that there can be no leakage from the future into the past. To accomplish the first point, the TCN uses a 1D fully-convolutional network (FCN) architecture (Long et al., 2015), where each hidden layer is the same length as the input layer, and zero padding of length $(\text{kernel size} - 1)$ is added to keep subsequent layers the same length as previous ones. To achieve the second point, the TCN uses *causal convolutions*, convolutions where a subsequent output at time t is convolved only with elements from time t and before in the previous layer.[1] Graphically, the network is shown in Figure 1. Put in a simple manner:

$$\text{TCN} = \text{1D FCN} + \text{causal convolutions}$$

It is worth emphasizing that this is essentially the same architecture as the time delay neural network proposed nearly 30 years ago by Waibel et al. (1989), with the sole tweak of zero padding to ensure equal sizes of all layers.

However, a major disadvantage of this "naïve" approach is that in order to achieve a long effective history size, we need an extremely deep network or very large filters, neither of which were particularly feasible when the methods were first introduced. Thus, in the following sections, we describe how techniques from modern convolutional architectures can be integrated into the TCN to allow for both very deep networks and very long effective history.

3.3 Dilated Convolutions

Through convolutional filters, as previously addressed, a simple causal convolution is only able to look back at a history with size linear in the depth of the network. This makes it challenging to apply

[1] Although many current deep learning libraries do not offer native support for these causal convolutions, we note that they can be implemented (with some inefficiency) by simply using a standard convolution with zero padding on both sides, and chopping off the end of the sequence.

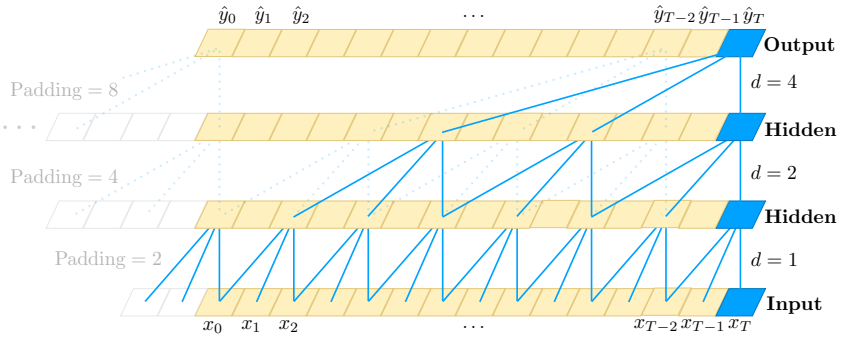


Figure 2: A dilated causal convolution with dilation factors $d = 1, 2, 4$ and filter size $k = 3$. The receptive field is able to cover all values from the input sequence.

the aforementioned causal convolution on sequence tasks, especially those requiring longer history. Our solution here, used previously for example in audio synthesis by van den Oord et al. (2016), is to employ dilated convolutions (Yu & Koltun, 2016) that enable an exponentially large receptive field. More formally, for a 1-D sequence input $\mathbf{x} \in \mathbb{R}^n$ and a filter $f : \{0, \dots, k-1\} \to \mathbb{R}$, the dilated convolution operation F on element s of the sequence is defined as

$$F(s) = (\mathbf{x} *_d f)(s) = \sum_{i=0}^{k-1} f(i) \cdot \mathbf{x}_{s-d \cdot i}$$

where d is the dilation factor and k is the filter size. Dilation is thus equivalent to introducing a fixed step between every two adjacent filter taps. When taking $d = 1$, for example, a dilated convolution is trivially a normal convolution operation. Using larger dilations enables an output at the top level to represent a wider range of inputs, thus effectively expanding the receptive field of a ConvNet.

This gives us two ways to increase the receptive field of the TCN: by choosing larger filter sizes k, and by increasing the dilation factor d, where the effective history of one such layer is $(k-1)d$. As is common when using dilated convolutions, we increase d exponentially with the depth of the network (i.e., $d = O(2^i)$ at level i of the network). This ensures that there is some filter that hits each input within the effective history, while also allowing for an extremely large effective history using deep networks. We provide an illustration in Figure 2. Using filter size $k = 3$ and dilation factor $d = 1, 2, 4$, the receptive field is able to cover all values from the input sequence.

3.4 Residual Connections

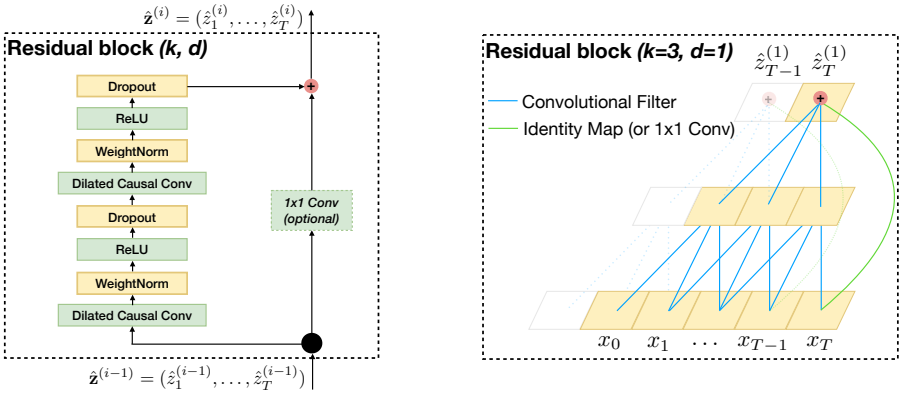
A residual block He et al. (2016) contains a branch leading out to a series of transformations $\mathcal{F}$, whose outputs are added to the input $\mathbf{x}$ of the block:

$$o = \text{Activation}(\mathbf{x} + \mathcal{F}(\mathbf{x})) \tag{2}$$

This effectively allows layers to learn modifications to the identity mapping rather than the entire transformation, which has repeatedly been shown to benefit very deep networks.

As the TCN's receptive field depends on the network depth n as well as filter size k and dilation factor d, stabilization of deeper and larger TCNs becomes important. For example, in a case where the prediction could depend on a history of size 2^{12} and a high-dimensional input sequence, a network of up to 12 layers could be needed. Each layer, more specifically, consists of multiple filters for feature extraction. In our design of the generic TCN model, we therefore employed a generic residual module in place of a convolutional layer.

The residual block for our baseline TCN is shown in figure 3a. Within a residual block, the TCN has 2 layers of dilated causal convolution and non-linearity, for which we used the rectified linear unit (ReLU) (Nair & Hinton, 2010). For normalization, we applied Weight Normalization (Salimans & Kingma, 2016) to the filters in the dilated convolution (where we note that the filters are essentially vectors of size $k \times 1$). In addition, a 2-D dropout (Srivastava et al., 2014) layer was added after each dilated convolution for regularization: at each training step, a whole channel (in the width dimension) is zeroed out.



(a) TCN residual block. An 1x1 convolution is added when residual input and output have different dimensions.

(b) An example of residual connection in TCN. The blue lines are filters in the residual function, and the green lines are identity mappings.

Figure 3: A visualization of the TCN residual block

However, whereas in standard ResNet the input is passed in and added directly to the output of the residual function, in TCN (and ConvNet in general) the input and output could have different widths. Therefore in our TCN, when the input-output widths disagree, we use an additional 1x1 convolution to ensure that element-wise addition $\oplus$ receives tensors of the same shape (see Figure 3a, 3b).

Note that many further optimizations (e.g., gating, skip connections, context stacking as in audio generation using WaveNet) are possible in a TCN than what we described here. However, in this paper, we aim to present a generic, general-purpose TCN, to which additional twists can be added as needed. As we are going to show in Section 4, this general-purpose architecture is already able to outperform recurrent units like LSTM on a number of tasks by a good margin.

3.5 Advantages of TCN Sequence Modeling

We conclude this section by listing several advantages and disadvantages of using TCNs for sequence modeling.

- **Parallelism**. Unlike in RNNs where the predictions for later timesteps must wait for their predecessors to complete, convolutions can be done in parallel since the same filter is used in each layer. Therefore, in both training and evaluation, a long input sequence can be processed as a whole in TCN, instead of sequentially as in RNN.
- **Flexible receptive field size**. A TCN can change its receptive field size in multiple ways. For instance, stacking more dilated (causal) convolutional layers, using larger dilation factors, or increasing the filter size are all viable options (with possibly different interpretations). TCN thus renders better control of the model's memory size, and is easy to adapt to different domains.
- **Stable gradients**. Unlike recurrent architectures, a TCN have a backpropagation path different from the temporal direction of the sequence. TCN thus avoids the problem of exploding/vanishing gradients, which is a major issue for RNNs (and which led to the development of LSTM, GRU, HF-RNN (Martens & Sutskever, 2011), etc.).
- **Low memory requirement for training**. Especially in the case of a long input sequence, LSTMs and GRUs can easily use up a lot of memory to store the partial results for their multiple cell gates. However, in a TCN the filters are shared across a layer, with backpropagation path depending only on network depth. Therefore in practice, we found gated RNNs likely to use up to a multiplicative factor more memory than TCNs.
- **Variable length inputs**. Just like RNNs, which model inputs with variable lengths in a recurrent way, TCNs can also take in inputs of arbitrary lengths by sliding the 1D convolutional kernels. This means that TCNs can be adopted as drop-in replacements for RNNs for sequential data of arbitrary length.

3.6 DISADVANTAGES OF TCN SEQUENCE MODELING

We also summarize two disadvantages of using TCN instead of RNNs.

- **Data storage in evaluation**. In evaluation/testing, RNNs only need to maintain a hidden state and take in a current input x_t in order to generate a prediction. In other words, a "summary" of the entire history is provided by the fixed-length set of vectors h_t, which means that the actual observed sequence can be discarded (and indeed, the hidden state can be used as a kind of encoder for all the observed history). In contrast, the TCN still needs to take in a sequence with non-trivial length (precisely the effective history length) in order to predict, thus possibly requiring more memory during evaluation.
- **Potential parameter change for a transfer of domain**. Different domains can have different requirements on the amount of history the model needs to memorize. Therefore, when transferring a model from a domain where only little memory is needed (i.e., small k and d) to a domain where much larger memory is required (i.e., much larger k and d), TCN may perform poorly for not having a sufficiently large receptive field.

We want to emphasize, though, that we believe the notable lack of "infinite memory" for a TCN is decidedly *not* a practical disadvantage, since, as we show in Section 4, the TCN method actually outperforms RNNs in terms of the ability to deal with long temporal dependencies.

4 EXPERIMENTS

In this section, we conduct a series of experiments using the baseline TCN (described in section 3) and generic RNNs (namely LSTMs, GRUs, and vanilla RNNs). These experiments cover tasks and datasets from various domains, aiming to test different aspects of a model's ability to learn sequence modeling. In several cases, specialized RNN models, or methods with particular forms of regularization can indeed vastly outperform both generic RNNs and the TCN on particular problems, which we highlight when applicable. But as a general-purpose architecture, we believe the experiments make a compelling case for the TCN as the "first attempt" approach for many sequential problems.

All experiments reported in this section used the same TCN architecture, just varying the depth of the network and occasionally the kernel size. We use an exponential dilation $d = 2^n$ for layer n in the network, and use the Adam optimizer (Kingma & Ba, 2015) with learning rate 0.002 for TCN (unless otherwise noted). We also empirically found that gradient clipping helped training convergence of TCN, and we picked the maximum norm to clip from $[0.3, 1]$. When training recurrent models, we used a simple grid search to find a good set of hyperparameters (in particular, optimizer, recurrent drop $p \in [0.05, 0.5]$, the learning rate, gradient clipping, and initial forget-gate bias), while keeping the network around the same size as TCN. No other optimizations, such as gating mechanism (see Appendix D), or highway network, were added to TCN or the RNNs. The hyperparameters we used for TCN on different tasks are reported in Table 2 in Appendix B. In addition, we conduct a series controlled experiments to investigate the effects of filter size and residual function on the TCN's performance. These results can be found in Appendix C.

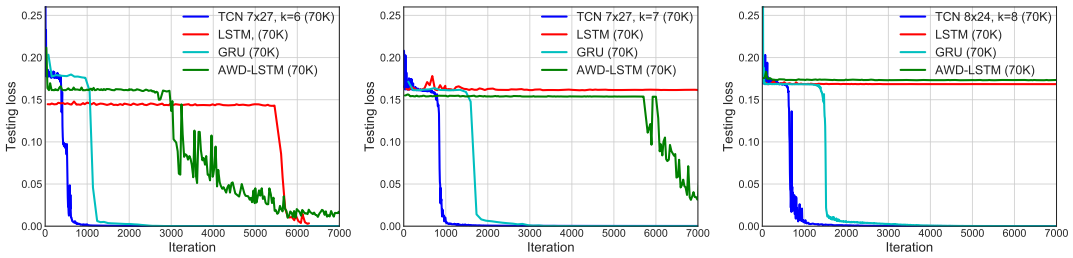
4.1 TASKS AND RESULTS SUMMARY

In this section we highlight the general performance of generic TCNs vs generic LSTMs for a variety of domains from the sequential modeling literature. A complete description of each task, as well as references to some prior works that evaluated them, is given in Appendix A. In brief, the tasks we consider are: the adding problem, sequential MNIST, permuted MNIST (P-MNIST), the copy memory task, the Nottingham and JSB Chorales polyphonic music tasks, Penn Treebank (PTB), Wikitext-103 and LAMBADA word-level language modeling, as well as PTB and text8 character-level language modeling.

A summary comparison of TCNs to the standard RNN architectures (LSTM, GRU, and vanilla RNN) is shown in Table 1. We will highlight many of these results below, and want to emphasize that for several tasks the baseline RNN architectures are still far from the state of the art (see Table 4), but in total the results make a strong case that the TCN architecture, as a generic sequence modeling framework, is often superior to generic RNN approaches. We now consider several of these

Table 1: Evaluation of TCNs and recurrent architectures on a variety of sequence modeling tasks. Current state-of-the-art results are listed in the Appendix. ^{h} means that higher is better. $^\ell$ means that lower is better.

Sequence Modeling Task	Model Size ($\approx$)	Models			
		LSTM	GRU	RNN	**TCN**
Seq. MNIST (accuracy^h)	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	**97.2**
Adding problem T=600 (loss$^\ell$)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T=1000 (loss)	16K	0.0204	0.0197	0.0202	**3.5e-5**
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	**8.10**
Music Nottingham (loss)	1M	3.29	3.46	4.05	**3.07**
Word-level PTB (perplexity$^\ell$)	13M	**78.93**	92.48	114.50	89.21
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^\ell$)	3M	1.41	1.42	1.52	**1.35**
Char-level text8 (bpc)	5M	1.52	1.56	1.69	**1.45**



(a) $T = 200$ (b) $T = 400$ (c) $T = 600$

Figure 4: Results of TCN vs. recurrent architectures on the adding problem

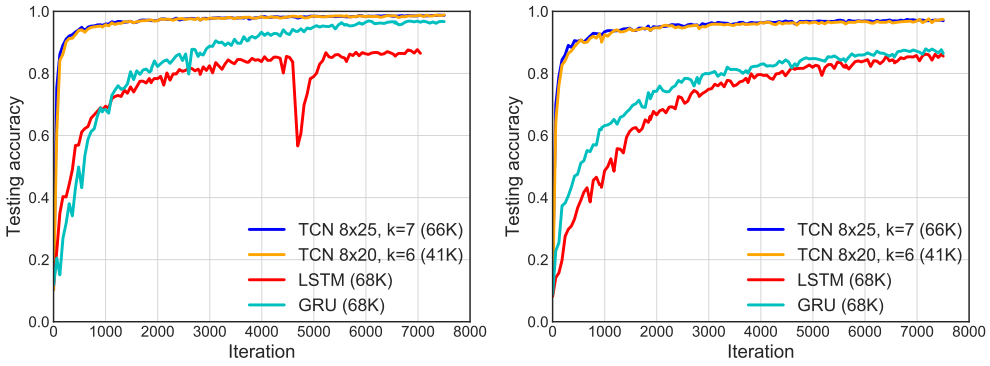
experiments in detail, generally distinguishing between the "recurrent benchmark" tasks designed to show the limitations of networks for sequence modeling (adding problem, sequential & permuted MNIST, copy memory), and the "applied" tasks (polyphonic music and language modeling).

4.2 Baseline Recurrent Tasks

We first compare the results of the TCN architecture to those of RNNs on the toy baseline tasks that have been frequently used to evaluate sequential modeling (Hochreiter & Schmidhuber, 1997; Martens & Sutskever, 2011; Pascanu et al., 2013; Le et al., 2015; Cooijmans et al., 2016; Zhang et al., 2016; Krueger et al., 2017; Wisdom et al., 2016; Arjovsky et al., 2016).

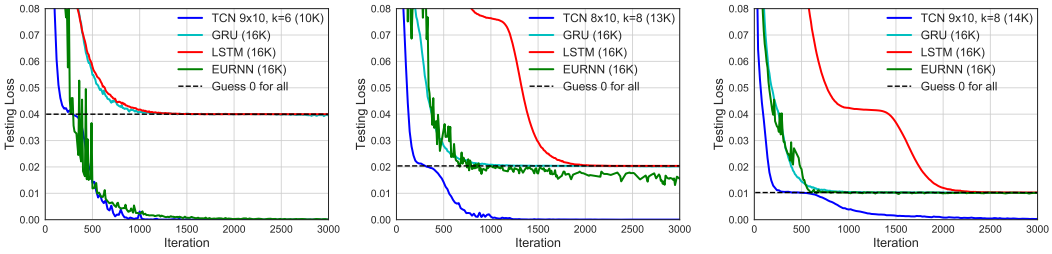
The Adding Problem. Convergence results for the adding problem, for problem sizes $T = 200, 400, 600$, are shown in Figure 4; all models were chosen to have roughly 70K parameters. In all three cases, TCNs quickly converged to a virtually perfect solution (i.e., an MSE loss very close to 0). LSTMs and vanilla RNNs performed significantly worse, while on this task GRUs also performed quite well, even though their convergence was slightly slower than TCNs.

Sequential MNIST and P-MNIST. Results on sequential and permuted MNIST, run over 10 epochs, are shown in Figures 5a and 5b; all models were picked to have roughly 70K parameters. For both problems, TCNs substantially outperform the alternative architectures, both in terms of convergence time and final performance level on the task. For the permuted sequential MNIST, TCNs outperform state of the art results using recurrent nets (95.9%) with Zoneout+Recurrent BatchNorm (Cooijmans et al., 2016; Krueger et al., 2017), a highly optimized method for regularizing RNNs.



(a) Sequential MNIST (b) P-MNIST

Figure 5: Results of TCN vs. recurrent architectures on the Sequential MNIST and P-MNIST



(a) $T = 500$. (b) $T = 1000$. (c) $T = 2000$.

Figure 6: Result of TCN vs. recurrent architectures on the Copy Memory Task, for different T

Copy Memory Task. Finally, Figure 6 shows the results of the different methods (with roughly the same size) on the copy memory task. Again, the TCNs quickly converge to correct answers, while the LSTM and GRU simply converge to the same loss as predicting all zeros. In this case we also compare to the recently-proposed EURNN (Jing et al., 2017), which was highlighted to perform well on this task. While both perform well for sequence length $T = 500$, the TCN again has a clear advantage for $T = 1000$ and $T = 2000$ (in terms of both loss and convergence).

4.3 Results on Polyphonic Music and Language Modeling

Next, we compare the results of the TCN architecture to recurrent architectures on 6 different real datasets in polyphonic music as well as word- and character-level language modeling. These are areas where sequence modeling has been used most frequently. As domains where there is considerable practical interests, there have also been many specialized RNNs developed for these tasks (e.g., Zhang et al. (2016); Ha et al. (2017); Krueger et al. (2017); Grave et al. (2017); Greff et al. (2017); Merity et al. (2017)). We mention some of these comparisons when useful, but the primary goal here is to compare the generic TCN model to other generic RNN architectures, so we focus mainly on these comparisons.

Polyphonic Music. On the Nottingham and JSB Chorales datasets, the TCN with virtually no tuning is again able to beat the other models by a considerable margin (see Table 1), and even outperforms some improved recurrent models for this task such as HF-RNN (Boulanger-Lewandowski et al., 2012) and Diagonal RNN (Subakan & Smaragdis, 2017). Note however that other models such as the Deep Belief Net LSTM (Vohra et al., 2015) perform substantially better on this task; we believe this is likely due to the fact that the datasets involved in polyphonic music are relatively small, and thus the right regularization method or generative modeling procedure can improve performance significantly. This is largely orthogonal to the RNN/TCN distinction, as a similar variant of TCN may well be possible.

Word-level language modeling. Language modeling remains one of the primary applications of recurrent networks and many recent works have focused on optimizing LSTMs for this task (Krueger et al., 2017; Merity et al., 2017). Our implementation follows standard practice that ties the weights of encoder and decoder layers for both TCN and RNNs (Press & Wolf, 2016), which significantly reduces the number of parameters in the model. For training, we use SGD and anneal the learning rate by a factor of 0.5 for both TCN and RNNs when validation accuracy plateaus.

On the smaller PTB corpus, an optimized LSTM architecture (i.e., with recurrent and embedding dropout, etc.) outperforms the TCN, while the TCN outperforms both GRU and vanilla RNN. However, on the much larger Wikitext-103 corpus and the LAMBADA dataset (Paperno et al., 2016), without any hyperparameter search, the TCN outperforms state-of-the-art LSTM results by Grave et al. (2017), achieving much lower perplexities.

Character-level Language Modeling. The results of applying TCN and alternative models on PTB and text8 data for character-level language modeling are shown in Table 1, with performance measured in *bits per character* (bpc). While beaten by the state of the art (see Table 4), the generic TCN outperforms regularized LSTM and GRU as well as methods such as Norm-stabilized LSTM (Krueger & Memisevic, 2015). Moreover, we note that using a filter size of $k \leq 4$ works better than larger filter sizes in character-level language modeling, which suggests that capturing short history is more important than longer dependencies in these tasks.

4.4 Memory Size of TCN and RNNs

Finally, one of the important reasons why RNNs have been preferred over CNNs for general sequence modeling is that *theoretically*, recurrent architectures are capable of an infinite memory. We therefore attempt to study here how much memory TCN and LSTM/GRU are able to actually "backtrack", focusing on 1) the copy memory task, which is a stress test designed to evaluate long-term, distant information propagation in recurrent networks, and 2) the LAMBADA task, which tests both local and non-local textual understanding of computational models.

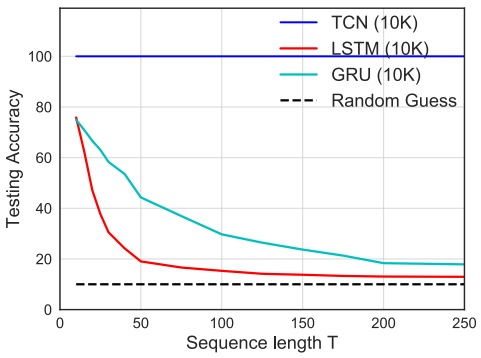


Figure 7: Accuracy on the copy memory task for varying sequence length T.

The copy memory task is a simple but perfect task to examine a model's ability to pick up its memory from a (possibly) distant past (by varying the value of sequence length T). However, different from the setting in Section 4.2, in order to compare the results for different sequence lengths, here we only report the accuracy on the *last 10 elements* of the output sequence. We used a model size of 10K for both TCN and RNNs.

The results of this focused study are shown in Figure 7. TCNs consistently converge to 100% accuracy for all sequence lengths, whereas LSTMs and GRUs of the same size quickly degenerate to random guessing as the sequence length T grows. The accuracy of the LSTM falls below 20% for $T < 50$, while the GRU falls below 20% for $T < 200$. These results indicate that TCNs are able to maintain a longer effective history than their recurrent counterparts.

This observation is also backed up by the experiments of TCN on the LAMBADA dataset, which is specifically designed to test a model's textual understanding in a broader discourse. The objective of LAMBADA dataset is to predict the last word of the target sentence given a sufficiently long context (see Appendix A for more details). Most of the existing models fail to guess accurately on this task. As shown in Table 1, TCN outperforms LSTMs by a significant margin in perplexity on LAMBADA, with a smaller network and virtually no tuning (State-of-the-art results on this dataset are even better, but only with the help of additional memory mechanisms Grave et al. (2017)).

5 DISCUSSION

In this work, we revisited the topic of modeling sequence predictions using convolutional architectures. We introduced the key components of the TCN and analyzed some vital advantages and disadvantages of using TCN for sequence predictions instead of RNNs. Further, we compared our generic TCN model to the recurrent architectures on a set of experiments that span a wide range of domains and datasets. Through these experiments, we have shown that TCN with minimal tuning can outperform LSTM/GRU of the same model size (and with standard regularizations) in most of the tasks. Further experiments on the copy memory task and LAMBADA task revealed that TCNs actually has a better capability for long-term memory than the comparable recurrent architectures, which are commonly believed to have unlimited memory.

It is still important to note that, however, we only presented a generic architecture here, with components all coming from standard modern convolutional networks (e.g., normalization, dropout, residual network). And indeed, on specific problems, the TCN model can still be beaten by some specialized RNNs that adopt carefully designed optimization strategies. Nevertheless, we believe the experiment results in Section 4 might be a good signal that instead of considering RNNs as the "default" methodology for sequence modeling, convolutional networks too, can be a promising and powerful toolkit in studying time-series data.

REFERENCES

Moray Allan and Christopher Williams. Harmonising chorales by probabilistic inference. In *Neural Information Processing Systems (NIPS)*, 2005.

Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2016.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 1994.

Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. Conditional time series forecasting with convolutional neural networks. *arXiv:1703.04691*, 2017.

Léon Bottou, F Fogelman Soulie, Pascal Blanchet, and Jean-Sylvain Liénard. Speaker-independent isolated digit recognition: Multilayer perceptrons vs. dynamic time warping. *Neural Networks*, 3 (4), 1990.

Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv:1206.6392*, 2012.

Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark Hasegawa-Johnson, and Thomas Huang. Dilated recurrent neural networks. *arXiv:1710.02224*, 2017.

Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv:1409.1259*, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv:1412.3555*, 2014.

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

Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. In *International Conference on Learning Representations (ICLR)*, 2016.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International Conference on Machine Learning (ICML)*, 2017.

Jeffrey L Elman. Finding structure in time. *Cognitive Science*, 14(2), 1990.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning (ICML)*, 2017.

Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research (JMLR)*, 3, 2002.

Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. In *International Conference on Learning Representations (ICLR)*, 2017.

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10), 2017.

David Ha, Andrew Dai, and Quoc V Le. HyperNetworks. In *International Conference on Learning Representations (ICLR)*, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition (CVPR)*, 2016.

Geoffrey E Hinton. Connectionist learning procedures. *Artificial intelligence*, 40(1-3), 1989.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997.

Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (EUNN) and their application to RNNs. In *International Conference on Machine Learning (ICML)*, 2017.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning (ICML)*, 2015.

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

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. A clockwork rnn. In *International Conference on Machine Learning (ICML)*, 2014.

David Krueger and Roland Memisevic. Regularizing rnns by stabilizing activations. *arXiv:1511.08400*, 2015.

David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Hugo Larochelle, Aaron C. Courville, and Chris Pal. Zoneout: Regularizing RNNs by randomly preserving hidden activations. In *International Conference on Learning Representations (ICLR)*, 2017.

Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv:1504.00941*, 2015.

Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. In *The handbook of brain theory and neural networks*, volume 3361, pp. 1995. 1995.

Yann Lecun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pp. 2278–2324, 1998.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19(2), 1993.

James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *International Conference on Machine Learning (ICML)*, 2011.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv:1609.07843*, 2016.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and Optimizing LSTM Language Models. *arXiv:1708.02182*, 2017.

Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *Preprint*, 2012.

Yasumasa Miyamoto and Kyunghyun Cho. Gated word-character recurrent language model. *arXiv:1606.01700*, 2016.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *International Conference on Machine Learning (ICML)*, 2010.

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. In *Neural Information Processing Systems (NIPS)*, 2016.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. *arXiv:1606.06031*, 2016.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2013.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv:1608.05859*, 2016.

Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Neural Information Processing Systems (NIPS)*, 2016.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15(1), 2014.

Y Cem Subakan and Paris Smaragdis. Diagonal RNNs in symbolic music modeling. *arXiv:1704.05420*, 2017.

Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. WaveNet: A generative model for raw audio. *arXiv:1609.03499*, 2016.

Raunaq Vohra, Kratarth Goel, and JK Sahoo. Modeling temporal dependencies in data using a DBN-LSTM. In *Data Science and Advanced Analytics (DSAA)*, 2015.

Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3), 1989.

Scott Wisdom, Thomas Powers, John Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks. In *Neural Information Processing Systems (NIPS)*, 2016.

Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Neural Information Processing Systems (NIPS)*, 2016.

Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. Breaking the softmax bottleneck: A high-rank RNN language model. *International Conference on Learning Representations (ICLR)*, 2018.

Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *International Conference on Learning Representations (ICLR)*, 2016.

Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan R Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. In *Neural Information Processing Systems (NIPS)*, 2016.

A DESCRIPTION OF BENCHMARK TASKS

The Adding Problem: In this task, each input consists of a length-n sequence of depth 2, with all values randomly chosen in $[0, 1]$, and the second dimension being all zeros expect for two elements that are marked by 1. The objective is to sum the two random values whose second dimensions are marked by 1. Simply predicting the sum to be 1 should give an MSE of about 0.1767. First introduced by Hochreiter & Schmidhuber (1997), the addition problem have been consistently used as a pathological test for evaluating sequential models (Pascanu et al., 2013; Le et al., 2015; Zhang et al., 2016; Arjovsky et al., 2016).

Sequential MNIST & P-MNIST: Sequential MNIST is frequently used to test a recurrent network's ability to combine its information from a long memory context in order to make classification prediction (Le et al., 2015; Zhang et al., 2016; Cooijmans et al., 2016; Krueger et al., 2017; Jing et al., 2017). In this task, MNIST (Lecun et al., 1998) images are presented to the model as a 784×1 sequence for digit classification In a more challenging setting, we also permuted the order of the sequence by a random (fixed) order and tested the TCN on this permuted MNIST (P-MNIST) task.

Copy Memory Task: In copy memory task, each input sequence has length $T + 20$. The first 10 values are chosen randomly from digit [1-8] with the rest being all zeros, except for the last 11 entries which are marked by 9 (the first "9" is a delimiter). The goal of this task is to generate an output of same length that is zero everywhere, except the last 10 values after the delimiter, where the model is expected to repeat the same 10 values at the start of the input. This was used by prior works such as Arjovsky et al. (2016); Wisdom et al. (2016); Jing et al. (2017); but we also extended the sequence lengths to up to $T = 2000$.

JSB Chorales: JSB Chorales dataset (Allan & Williams, 2005) is a polyphonic music dataset consisting of the entire corpus of 382 four-part harmonized chorales by J. S. Bach. In a polyphonic music dataset, each input is a sequence of elements having 88 dimensions, representing the 88 keys on a piano. Therefore, each element x_t is a chord written in as binary vector, in which a "1" indicates a key pressed.

Nottingham: Nottingham dataset [2] is a collection of 1200 British and American folk tunes. Nottingham is a much larger dataset than JSB Chorales. Along with JSB Chorales, Nottingham has been used in a number of works that investigated recurrent models' applicability in polyphonic music (Greff et al., 2017; Chung et al., 2014), and the performance for both tasks are measured in terms of negative log-likelihood (NLL) loss.

PennTreebank: We evaluated TCN on the PennTreebank (PTB) dataset (Marcus et al., 1993), for both character-level and word-level language modeling. When used as a character-level language corpus, PTB contains 5059K characters for training, 396K for validation and 446K for testing, with an alphabet size of 50. When used as a word-level language corpus, PTB contains 888K words for training, 70K for validation and 79K for testing, with vocabulary size 10000. This is a highly studied dataset in the field of language modeling (Miyamoto & Cho, 2016; Krueger et al., 2017; Merity et al., 2017), with exceptional results have been achieved by some highly optimized RNNs.

Wikitext-103: Wikitext-103 (Merity et al., 2016) is almost 110 times as large as PTB, featuring a vocabulary size of about 268K. The dataset contains 28K Wikipedia articles (about 103 million words) for training, 60 articles (about 218K words) for validation and 60 articles (246K words) for testing. This is a more representative (and realistic) dataset than PTB as it contains a much larger vocabulary, including many rare vocabularies.

LAMBADA: Introduced by Paperno et al. (2016), LAMBADA is a dataset consisting of 10K passages extracted from novels, with on average 4.6 sentences as context, and 1 target sentence whose last word is to be predicted. This dataset was built so that human can guess naturally and perfectly when given the context, but would fail to do so when only given the target sentence. Therefore, LAMBADA is a very challenging dataset that evaluates a model's textual understanding and ability to keep track of information in the broader discourse. Here is an example of a test in the LAMBADA dataset, where the last word "miscarriage" is to be predicted (which is not in the context):

[2] See http://ifdo.ca/∼seymour/nottingham/nottingham.html

Context: "Yes, I thought I was going to lose the baby.""I was scared too." he stated, sincerity flooding his eyes. "You were?""Yes, of course. Why do you even ask?""This baby wasn't exactly planned for."

Target Sentence: "Do you honestly think that I would want you to have a _____"

Target Word: miscarriage

This dataset was evaluated in prior works such as Paperno et al. (2016); Grave et al. (2017). In general, better results on LAMBADA indicate that a model is better at capturing information from longer and broader context. The training data for LAMBADA is the full text of 2,662 novels with more than 200M words [3], and the vocabulary size is about 93K.

text8: We also used text8[4] dataset for character level language modeling (Mikolov et al., 2012). Compared to PTB, text8 is about 20 times as large, with about 100 million characters from Wikipedia (90M for training, 5M for validation and 5M for testing). The corpus contains 27 unique alphabets.

[3]LAMBADA and training dataset can be found at http://clic.cimec.unitn.it/lambada/

[4]Available at http://mattmahoney.net/dc/text8.zip

B HYPERPARAMETERS SETTINGS

B.1 HYPERPARAMETERS FOR TCN

In this supplementary section, we report in a table (see Table 2) the hyperparameters we used when applying the generic TCN model on the different tasks/datasets. The most important factor for picking parameters is to make sure that the TCN has a *sufficiently large* receptive field by choosing k and n that can cover the amount of context needed for the task.

Table 2: TCN parameter settings for experiments in Section. 4

TCN SETTINGS							
Dataset/Task	**Subtask**	k	n	**Hidden**	**Dropout**	**Grad Clip**	**Note**
The Adding Problem	$T = 200$	6	7	27	0.0	N/A	
	$T = 400$	7	7	27			
	$T = 600$	8	8	24			
Seq. MNIST	-	7	8	25	0.0	N/A	
		6	8	20			
Permuted MNIST	-	7	8	25	0.0	N/A	
		6	8	20			
Copy Memory Task	$T = 500$	6	9	10	0.05	1.0	RMSprop 5e-4
	$T = 1000$	8	8	10			
	$T = 2000$	8	9	10			
Music JSB Chorales	-	3	2	150	0.5	0.4	
Music Nottingham	-	6	4	150	0.2	0.4	
Word-level LM	PTB	3	4	600	0.5	0.4	Embed. size 600
	Wiki-103	3	5	1000	0.4		Embed. size 400
	LAMBADA	4	5	500			Embed. size 500
Char-level LM	PTB	3	3	450	0.1	0.15	Embed. size 100
	text8	2	5	520			

As previously mentioned in Section 4, the number of hidden units was chosen based on k and n such that the model size is approximately at the same level as the recurrent models. In the table above, a gradient clip of N/A means no gradient clipping was applied. However, in larger tasks, we empirically found that adding a gradient clip value (we randomly picked from $[0.2, 1]$) helps the training convergence.

B.2 HYPERPARAMETERS FOR LSTM/GRU

We also report the parameter setting for LSTM in Table 3. These values are picked from hyper-parameter search for LSTMs that have up to 3 layers, and the optimizers are chosen from {SGD, Adam, RMSprop, Adagrad}.

GRU hyperparameters were chosen in a similar fashion, but with more hidden units to keep the total model size approximately the same (since a GRU cell is smaller).

B.3 COMPARE TO THE STATE OF THE ART RESULTS

As previously noted, TCN can still be outperformed by optimized RNNs in some of the tasks, whose results are summarized in Table 4 below. The same TCN architecture is used across all tasks.

Note that the size of the SoTA model may be different from the size of the TCN.

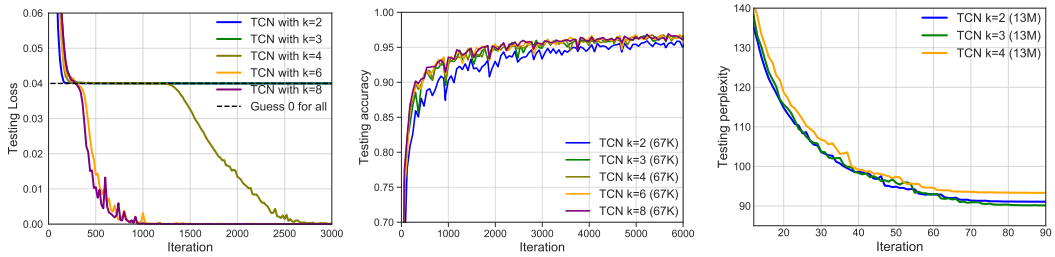
Table 3: LSTM parameter settings for experiments in Section 4.

LSTM SETTINGS (KEY PARAMETERS)							
Dataset/Task	**Subtask**	n	**Hidden**	**Dropout**	**Grad Clip**	**Bias**	**Note**
The Adding Problem	$T = 200$	2	77	0.0	50	5.0	SGD 1e-3
	$T = 400$	2	77		50	10.0	Adam 2e-3
	$T = 600$	1	130		5	1.0	-
Seq. MNIST	-	1	130	0.0	1	1.0	RMSprop 1e-3
Permuted MNIST	-	1	130	0.0	1	10.0	RMSprop 1e-3
Copy Memory Task	$T = 500$	1	50	0.05	0.25	-	RMSprop/Adam
	$T = 1000$	1	50		1		
	$T = 2000$	3	28		1		
Music JSB Chorales	-	2	200	0.2	1	10.0	SGD/Adam
Music Nottingham	-	3	280	0.1	0.5	-	Adam 4e-3
		1	500		1	-	
Word-level LM	PTB	3	700	0.4	0.3	1.0	SGD 30, Emb. 700, etc.
	Wiki-103	-	-	-	-	-	Grave et al. (2017)
	LAMBADA	-	-	-	-	-	Grave et al. (2017)
Char-level LM	PTB	2	600	0.1	0.5	-	Emb. size 120
	text8	1	1024	0.15	0.5	-	Adam 1e-2

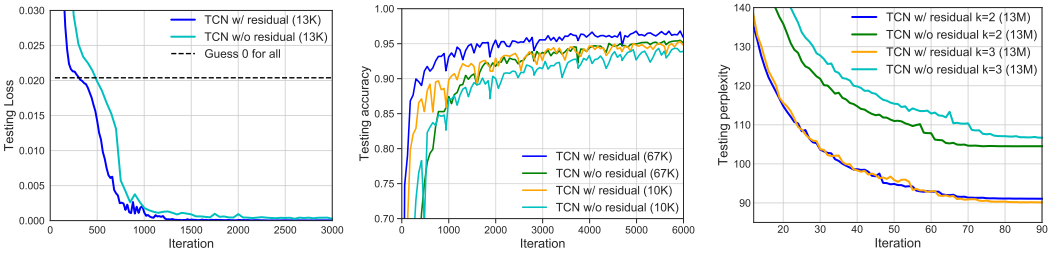
Table 4: State-of-the-art (SoTA) results for tasks in Section 4.

TCN VS. SOTA RESULTS					
Task	**TCN Result**	**Size**	**SoTA**	**Size**	**Model**
Seq. MNIST (acc.)	99.0	21K	99.0	21K	Dilated GRU (Chang et al., 2017)
P-MNIST (acc.)	97.2	42K	95.9	42K	Zoneout (Krueger et al., 2017)
Adding Prob. 600 (loss)	5.8e-5	70K	5.3e-5	70K	Regularized GRU
Copy Memory 1000 (loss)	3.5e-5	70K	0.011	70K	EURNN (Jing et al., 2017)
JSB Chorales (loss)	8.10	300K	3.47	-	DBN+LSTM (Vohra et al., 2015)
Nottingham (loss)	3.07	1M	1.32	-	DBN+LSTM (Vohra et al., 2015)
Word PTB (ppl)	89.21	13M	47.7	22M	AWD-LSTM-MoS + Dynamic Eval. (Yang et al., 2018)
Word Wiki-103 (ppl)	45.19	148M	40.4	>300M	Neural Cache Model (Large) (Grave et al., 2017)
Word LAMBADA (ppl)	1279	56M	138	>100M	Neural Cache Model (Large) (Grave et al., 2017)
Char PTB (bpc)	1.35	3M	1.22	14M	2-LayerNorm HyperLSTM (Ha et al., 2017)
Char text8 (bpc)	1.45	4.6M	1.29	>12M	HM-LSTM (Chung et al., 2016)

C THE EFFECT OF FILTER SIZE AND RESIDUAL BLOCK ON TCN



(a) Different k on Copy Memory Task (b) Different k on P-MNIST (c) Different k on PTB (word)



(d) Residual on Copy Memory Task (e) Residual on P-MNIST (f) Residual on PTB (word)

Figure 8: Controlled experiments that examine different components of the TCN model

In this section we briefly study, via controlled experiments, the effect of filter size and residual block on the TCN's ability to model different sequential tasks. Figure 8 shows the results of this ablative analysis. We kept the model size and the depth of the networks exactly the same within each experiment so that dilation factor is controlled. We conducted the experiment on three very different tasks: the copy memory task, permuted MNIST (P-MNIST), as well as word-level PTB language modeling.

Through these experiments, we empirically confirm that both filter sizes and residuals play important roles in TCN's capability of modeling potentially long dependencies. In both the copy memory and the permuted MNIST task, we observed faster convergence and better result for larger filter sizes (e.g. in the copy memory task, a filter size $k \geq 3$ led to only suboptimal convergence). In word-level PTB, we find a filter size of $k = 3$ works best. This is not a complete surprise, since a size-k filter on the inputs is analogous to a k-gram model in language modeling.

Results of control experiments on the residual function are shown in Figure 8d, 8e and 8f. In all three scenarios, we observe that the residual stabilizes the training by bringing a faster convergence as well as better final results, compared to TCN with the same model size but no residual block.

D EXPERIMENTS: GATING MECHANISM ON TCN

One component that has shown to be effective in adapting a convolutional architecture to language modeling is the gating activation, which we have chosen not to include in the generic TCN model. However, even among the prior works that adopted gated activations (e.g., van den Oord et al. (2016); Dauphin et al. (2017)), different gating mechanisms were chosen as well. In particular, Dauphin et al. (2017) empirically compared the effects of gated linear unit (GLU) and gated tanh unit (GTU), and adopted GLU in their non-dilated gated ConvNet model to control the information flow. Following the same choice, we compared TCNs using ReLU and gating, represented by an elementwise product between two convolutional layers, with one of them also passing through a sigmoid function $\sigma(x)$. Note that this introduces approximately twice as many convolutional layers as in ReLU-TCN.

The results shown in Table 5, where we kept the number of model parameters at about the same size. By comparison, we notice that GLU does further improve the TCN results on certain language modeling datasets like PTB, which agrees with prior works. However, we do not observe such benefits to exist in general, such as on polyphnic music datasets, or the simpler synthetic stress tests that require longer information retention. On the copy memory task with $T = 1000$, we found that TCN with gating converged to a result that is substantially worse than TCN with ReLU (though still better than recurrent models).

Table 5: TCN with Gating Mechanism within Residual Block

RELU TCN VS. GATED TCN RESULTS		
Task	**TCN**	**TCN + Gating**
Sequential MNIST (acc.)	**99.0**	**99.0**
Permuted MNIST (acc.)	**97.2**	96.9
Adding Problem $T = 600$ (loss)	**5.8e-5**	**5.6e-5**
Copy Memory $T = 1000$ (loss)	**3.5e-5**	0.00508
JSB Chorales (loss)	**8.10**	8.13
Nottingham (loss)	**3.07**	3.12
Word-level PTB (ppl)	89.21	**87.94**
Char-level PTB (bpc)	1.35	**1.343**
Char text8 (bpc)	**1.45**	1.485