Efficient Real-Time Inference in Temporal Convolution Networks

Piyush Khandelwal, James MacGlashan, Peter Wurman, and Peter Stone

***Abstract*— It has been recently demonstrated that Temporal Convolution Networks (TCNs) provide state-of-the-art results in many problem domains where the input data is a time-series. TCNs typically incorporate information from a long history of inputs (the receptive field) into a single output using many convolution layers. Real-time inference using a trained TCN can be challenging on devices with limited compute and memory, especially if the receptive field is large. This paper introduces the RT-TCN algorithm that reuses the output of prior convolution operations to minimize the computational requirements and persistent memory footprint of a TCN during real-time inference. We also show that when a TCN is trained using time slices of the input time-series, it can be executed in real-time continually using RT-TCN. In addition, we provide TCN architecture guidelines that ensure that real-time inference can be performed within memory and computational constraints.**

I. INTRODUCTION

Many state-of-the-art solutions to problems operating on time-series input, such as human activity recognition [25] and heartbeat detection [12], make use of Deep Neural Networks (DNNs). Among different DNN architectures, *Temporal Convolution Networks* (TCNs) have achieved excellent results on both synthetic and real datasets [2, 24]. In TCNs, convolution operations are applied along the time dimension, i.e. data from many different time-steps is convolved together. By building layers of such temporal convolutions, a single output can incorporate a long history of input data. This input history is defined as the *receptive field* of the TCN. A 3-layer TCN is illustrated in Figure 1b.

In this paper, we specifically deal with the problem of calculating TCN outputs in real-time as time-series input becomes available. For the example TCN illustrated in Figure 1b, predictions are made every two samples after the first 15 samples are observed. Real-time execution implies that output y_{t-2} is computed as soon as input x_{t-2} is available at time $t-2$ and its calculation must be completed *before* the next prediction y_t is required at time t, or the system will gradually fall behind. Note that some of the individual convolutions required to compute y_t were computed previously while computing y_{t-2} (marked in dashed black). If the TCN is executed on a device with sufficient compute bandwidth and memory, the last T inputs can simply be stored in memory and these individual convolutions can be recomputed to compute the latest output. However, when resources are limited, executing a trained TCN in real-time requires a more sophisticated pipeline.

This paper introduces the *Real-Time TCN* (RT-TCN) algorithm for computing TCN outputs, which retains prior convolution outputs so that convolutions are not recomputed. Specifically, in the example TCN in Figure 1b, RT-TCN only computes the convolution shared between y_{t-2} and y_t once. For some TCN architectures, RT-TCN may also reduce the memory footprint compared to the straightforward approach of buffering the last T inputs. On devices with limited compute and memory, RT-TCN allows executing larger TCN architectures than would be otherwise possible.

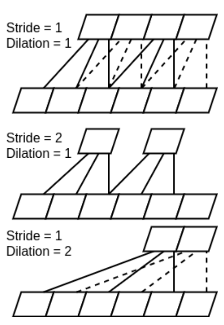
In addition to the main contribution of introducing RT-TCN, this paper makes two further contributions. A TCN typically operates *continually* on input sequences of arbitrary length. If the input length of the TCN is always equal to its receptive field size, then the TCN will generate only a single output, and we term such TCNs as *SingleWindowTCNs* (in Sections IV and V, we discuss why SingleWindowTCNs are useful). If a TCN only generates a single output, not all convolution operations it typically performs in a layer contribute to that single output. Thus, it is possible for a SingleWindowTCN to have a reduced network architecture than the corresponding continual TCN, and the second contribution of this paper is providing algorithms to convert between a SingleWindowTCN and the corresponding TCN. These algorithms recompute the rate at which convolutions should be performed at each layer.

The third and final contribution of this paper is providing architecture recommendations which enable RT-TCN to minimize its memory footprint and computational requirements. We provide results that show RT-TCN memory consumption and computational requirements (in terms of multiplicative FLOPS) across different network architectures. These results lead to guidelines that may help a user balance network architecture size with performance while using RT-TCN on a device with limited resources.

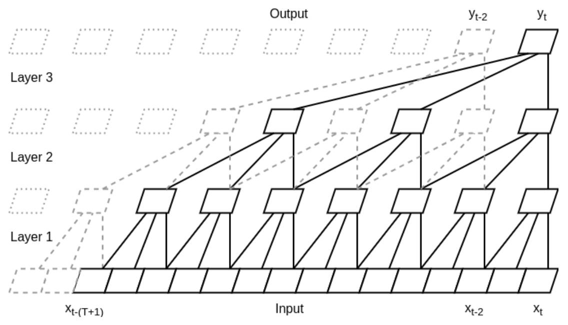
We believe that these contributions will be particularly of interest to the robotics community, as low-powered sensors and IoT devices that operate on real-world time-series data and use TCN architectures for estimation will benefit from faster inference methods.

In the next section, we provide a brief overview of TCNs, followed by a discussion of related work in Section III. In Section IV, we specify the RT-TCN algorithm. Next, we present results summarizing compute and memory footprints when using RT-TCN, and provide guidelines for different network architectures and inference approaches. Finally, we conclude in Section VI.

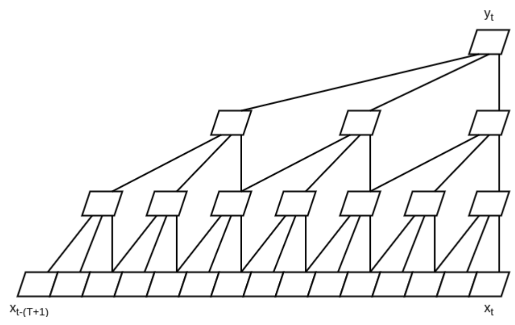
Piyush Khandelwal, James MacGlashan, Peter Wurman, and Peter Stone are with Sony AI, Sony Corporation of America, 25 Madison Avenue, New York, NY 10010, USA. Peter Stone is also with the Department of Computer Science, University of Texas at Austin, 2317 Speedway, Austin, TX 78712, USA. Email: {`piyush.khandelwal, james.macglashan, peter.wurman, peter.stone`}`@sony.com`



(a) Example s_i and d_i



(b) Continual Temporal Convolution Network (TCN)



(c) Reduced SingleWindowTCN with 1 output

Fig. 1: (a) shows examples of different values of strides (s_i) and dilation rates (d_i) for a TCN layer i in a network. (b) illustrates an example TCN with 3 layers and a receptive field size $T = 15$. The three layers apply a 1-d temporal convolution with kernel sizes $k = [3, 3, 3]$, network dilation rates $d = [1, 1, 2]$ and strides $s = [2, 1, 1]$, respectively. Given T inputs, this TCN computes 7, 5, and 1 convolution across the three layers, respectively. Only convolutions marked via solid lines contribute to output y_t, and the corresponding SingleWindowTCN in (c) only requires 3 convolutions at layer 2 (not 5).

II. TEMPORAL CONVOLUTION NETWORKS

Typically, convolution networks have been applied in image processing domains where 2-dimensional convolutions are applied across the width and height of an image. A TCN is a 1-dimensional convolution network where the input data is supplied as a time-series. A TCN takes a history of inputs, where the input at each time-step may consist of many different channels (i.e. features), and performs 1-d convolutions on this input data to generate an output. Similar to general convolution networks, a TCN also comprises many convolution layers, and the *depth* (n) of the TCN is defined by the number of layers inside it (in Figure 1b, $n = 3$).

Since this paper focuses on real-time execution of TCNs, it only considers TCNs that are made up of *causal convolutions*. A convolution operation is considered causal if the output of that operation is generated by convolving input data from the current time step or earlier. Causality ensures that a TCN does not depend on future data, making the TCN suitable for real-time execution. In Figure 1b, causality is illustrated by lining up the last input in a convolution with its output along the x-axis (time dimension).

For any given TCN layer, a convolution operation is defined via the following parameters:

- *Kernel Size* (k) - Kernel size is the number of time-steps that are convolved together in a single convolution operation. In Figure 1b, $k = 3$ for all layers.
- *Number of Filters* (n_f) - To obtain multiple features from the same convolution kernel input, multiple independent convolutions (filters) are applied on the same input kernel to generate a multi-channeled convolution output. Number of input channels (c) and n_f are not illustrated in Figure 1b. They are used by RT-TCN to compute the size of input and intermediate buffers.
- *Network Dilation Rate* (d) - Dilation controls the spacing between subsequent inputs supplied to a single convolution operation. Increasing dilation at any layer increases the receptive field size without increasing network depth of kernel size, and is useful in domains where a longer history is required to solve the task. In Figure 1b, the network dilation rates are 1, 1, and 2 for layers 1, 2, and 3, respectively.
- *Stride* (s) - Convolution kernels in a layer are applied repeatedly spaced apart by the convolution stride. The *output rate reduction* (r) of the the TCN, i.e. the output rate relative to the input rate, can be calculated as $r = \Pi_i^n s_i$, where s_i is the stride size for layer i. In Figure 1b, the strides are 2, 1, and 1 for layers 1, 2, and 3, respectively, and $r = 2 \times 1 \times 1 = 2$. Examples of different d and s are illustrated in Figure 1a.
- *Activation Function* (g) - The activation function of a layer is a non-linear function applied to the result of individual convolution operations, allowing the network to model complex non-linear relationships. The choices for activation functions in TCN are typically the same as those used for feed forward networks; e.g., ReLU, sigmoid, tanh, or ELU among others [1].
- *Dilated Kernel Size* (k') - At layer i, the number of timesteps used as input for a convolution is termed as dilated kernel size and computed as $k_i^{'} = d_i(k_i - 1) + 1$.

Given a TCN, the first network output is not available until T input samples are available, where T is the receptive field size and can be calculated as follows:

$$T_n = k_n^{'}, \quad T_i = k_i^{'} + s_i(T_{i+1} - 1)\ \forall 1 \le i < n, \quad T = T_1$$

In Figure 1b, the receptive field size $T = 15$. *Padding* (p) defines the number of zero-padded inputs. If $p = 14$, the TCN can make a prediction immediately after the first input.

III. RELATED WORK

2-d convolution networks [14, 15] have been used for a long time to perform object recognition in images. Similarly, causal dependencies were learned in neural networks by supplying time-shifted input to recognize phonemes over 30 years ago [22]. In the last few years, Temporal Convolutional Networks (TCNs), which make use of convolutions across time-shifted features, have grown increasingly popular. TCNs have been shown to produce better performance than recurrent networks such as LSTMs [7] across different time-series benchmarks [2, 24] and forecasting [11, 23].

TCNs have also been successfully applied across many different application domains. The WaveNet architecture uses a TCN to generate raw audio waveforms [17]. Lea et al. perform action segmentation and detection in video by feeding spatiotemporal features extracted from individual video frames into two different TCN-based architectures, and show that TCNs can achieve state-of-the-art results while being trained an order of magnitude faster than LSTMs [13]. In [4], Chang et al. demonstrate TCNs with gated activations and residual connections outperform state-of-the-art LSTM networks on voice activity detection. Similarly, Lara-Benitez et al. show that TCNs outperform LSTMs in energy-related time series forecasting [10]. TinyRadarNN feeds the output of a conventional 2-d CNN into a TCN for hand gesture prediction in a battery operated wearable device [19].

Next, we discuss different frameworks for implementing TCNs on embedded devices. While TCN implementations using popular deep learning libraries are available [8, 18], the underlying libraries are unsuitable for operation on embedded devices. More recently, TensorFlow Lite Micro [5] provides a better framework for implementing neural networks on embedded devices. Similarly, Carreras et al [3] demonstrated FPGA-based acceleration for TCN inference. However, neither approach provides a mechanism for reusing convolutions during continual real-time TCN inference where convolutions are shared across subsequent evaluations. Finally, CMSIS-NN [9] discusses optimizing convolutions on Arm Cortex-M CPUs, and such advances can complement the convolution reuse proposed by RT-TCN.

IV. RT-TCN

In this section, we first describe RT-TCN, which enables executing a TCN continually in real-time without redundant computation. Next, we present algorithms that recompute the rate of computing convolutions when moving from a TCN to the corresponding SingleWindowTCN and vice-versa.

A. RT-TCN *Algorithm*

RT-TCN operates using these 2 principles:

- A convolution operation is performed as soon as a dilated kernel width of inputs are available at that layer.
- Each layer uses an independent buffer to maintain a history of inputs required for the next convolution.

RT-TCN consists of two parts:

1) Algorithm 1 is run once to allocate buffers to store input and intermediate data and initialize them.
2) Algorithm 2 is then supplied generated buffers and runs continually. Given the latest input, this algorithm performs the necessary individual convolutions in all layers, updates the necessary buffers, and returns the network output if it is available. The network output may not be available every time-step if:
    - The number of inputs processed is less than $T - p$.
    - Output rate reduction $r > 1$, as in Figure 1b.

Algorithm 1 Allocate and Initialize Buffers

```
Input: $n \leftarrow$ Network Depth/Number of layers
Input: $c \leftarrow$ Number of input channels
Input: $f[\,] \leftarrow$ List of number of filters for each layer
Input: $k[\,] \leftarrow$ List of kernel sizes for each layer
Input: $s[\,] \leftarrow$ List of strides for each layer
Input: $d[\,] \leftarrow$ List of dilation rates for each layer
Input: $p \leftarrow$ Number of zero-padded inputs
Output: $b[\,], b_h[\,]$ Lists of buffers and initial write heads
1: $n_c \leftarrow$ CONCATENATE$([c], f[1 : n-1])$
2: for $i = 1$ to $n$ do
3:     $k' \leftarrow d[i] \cdot (k[i] - 1) + 1$
4:     $b[i] \leftarrow$ Matrix$[n_c[i]][k']$
5:     $b_h[i] \leftarrow 0$
6: for $j = 1$ to $p$ do                              ▷ Apply padding
7:     $x \leftarrow$ Vector$[c]$
8:     FILL$(x, 0)$          ▷ Input with all channels set to zero
9:     RT-TCN$(x_t = x)$                      ▷ Apply Algorithm 2
```

In Algorithm 1, we first compute the number of input channels n_c at each layer on line 1 by combining the number of input channels and number of convolution filters from all but the last layer. On line 3, we compute the dilated kernel size k' for layer i, which is then used to initialize a single fixed size buffer $b[i]$ in "CW" layout (channels-width) on line 4. For a TCN, the width "W" in the "CW" layout references the time dimension [16]. On line 5, we initialize the write head $b_h[i]$ which identifies the column along "W" where the next input to that buffer should be written to. If padding p is specified, p *zero* inputs are fed into Algorithm 2 to populate the buffers appropriately (lines 6-9). Setting $p = T - 1$ ensures that an output is generated on the first input.

Algorithm 2 takes a new single multi-channeled input x_t and passes it through the network while performing individual convolutions as necessary. A new input can trigger at most 1 matrix multiplication per layer if the input buffer for that layer becomes full. If a convolution is triggered in the output layer, then a new network output gets generated.

On line 1, the new input sample x_t is put in a temporary buffer x'_t. On lines 4 and 5, x'_t is copied into the input buffer for the current layer. If the buffer is full, then a dilated convolution with an activation function is performed on lines 8 and 9 and the result is placed in x'_t to be processed as an input for the next layer. The input buffer is emptied by shifting it by the layer stride to eliminate the oldest values along the time dimension (line 10), and the write head is updated (line 11). Finally, if a convolution is performed in the output layer, the temporary buffer x'_t is copied to the output buffer on line 12. If the buffer is not full at any layer, then no output is returned (lines 14 and 15).

B. *SingleWindowTCN*

To train a TCN, the straightforward approach is to sample minibatches of whole input sequences with shape "NCW," where N is the minibatch size, C is the number of input channels, and W is the "width" of the sequence, and then perform standard stochastic gradient descent optimization

Algorithm 2 RT-TCN Algorithm

```
$n \leftarrow$ Network Depth/Number of layers
$s[\ ] \leftarrow$ List of stride for each layer
$d[\ ] \leftarrow$ List of dilation rates for each layer
$m_w[\ ] \leftarrow$ List of weight matrices for each layer
$m_b[\ ] \leftarrow$ List of bias matrices for each layer
$g[\ ] \leftarrow$ List of activations for each layer
$b[\ ] \leftarrow$ List of input buffers for each layer
$b_h[\ ] \leftarrow$ List of buffer write heads for each layer
Input: $x_t \leftarrow$ Next input data point to process at layer 0
Output: $o$ (Network output, none if not available)
1:  $x'_t \leftarrow x_t$
2:  for $i = 1$ to $n$ do
3:      $c \leftarrow$ NUMROWS($b[i]$)                ▷ Input channel size
4:      for $j = 0$ to $c$ do          ▷ Fill channels at buffer head
5:          $b[i][j][b_h[i]] \leftarrow x'_t[j]$
6:      $b_h[i] \leftarrow b_h[i] + 1$
7:      if $b_h[i] =$ NUMCOLS($b[i]$) then            ▷ Full buffer
8:          $x'_t \leftarrow$ CONV1D($b[i], m_w[i], m_b[i], d[i]$)
9:          $x'_t \leftarrow g[i](x'_t)$          ▷ Apply activation function
10:         $b[i] \leftarrow$ SHIFTCOLSLEFT($b[i], s[i]$)
11:         $b_h[i] \leftarrow b_h[i] - s[i]$
12:         if $i = n$ then $o \leftarrow x'_t$
13:     else
14:         $o \leftarrow$ none
15:         break
```

on those minibatches to compute weight and bias matrices. However, there is a potential drawback to this approach. If we train with only a few sequences of long lengths (small N and large W), then correlations within a single sequence may negatively impact the learning process. To address this issue, smaller random subsequences may be extracted and used for training, and the minimum size of such sequences is equal to the receptive field size.

If we feed only a receptive field size length of inputs into a TCN, some intermediate convolutions no longer contribute to the single output and can be removed. For example, Figure 1c shows a reduced SingleWindowTCN for the TCN in Figure 1b, and only 3 convolution operations need to be executed in Layer 2 instead of 5 when the input length is equal to the receptive field size. Popular deep learning libraries such as TensorFlow perform such pruning optimizations automatically during training [6], but such libraries cannot be used for inference on resource-constrained devices.

Note that network reduction is not necessary, as the TCN parameters s and d are sufficient to compute the output when the input sequence length is equal to its receptive field size. Some intermediate convolutions will not be used in the final output, and in Section V, we examine how removing such convolutions decreases computational requirements when RT-TCN is not used for inference. In Algorithm 3, we specify how a TCN can be reduced to a SingleWindowTCN by reducing the rate of convolutions in each layer. Additionally, if a trained SingleWindowTCN is used for inference continually via RT-TCN, it is necessary to recalculate the

Algorithm 3 Continual TCN to SingleWindowTCN

```
Input: $n \leftarrow$ Network Depth/Number of layers
Input: $s[\ ] \leftarrow$ Continual strides for each layer
Input: $d[\ ] \leftarrow$ Continual dilation rates for each layer
Output: $s_{sw}[\ ] \leftarrow$ SingleWindow strides for each layer
Output: $d_{sw}[\ ] \leftarrow$ SingleWindow dilation rates for each layer
1: $s_{sw}[n] \leftarrow 1$
2: $d_{sw}[n] \leftarrow 1$
3: $s_m \leftarrow d[n]$                    ▷ Stride rate multiplier
4: for $i = n - 1$ to 1 do
5:     $s_{sw}[i] \leftarrow s[i] \cdot s_m$
6:     $d_{sw}[i] \leftarrow d[i]$
7:     $s_m \leftarrow$ GCD($s_{sw}[i], d_{sw}[i]$)
8:     $s_{sw}[i] \leftarrow \frac{s_{sw}[i]}{s_m}$
9:     $d_{sw}[i] \leftarrow \frac{d_{sw}[i]}{s_m}$
```

rate at which convolutions need to be computed in every layer given output rate r (Algorithm 4).

We now describe Algorithm 3 in detail. In a SingleWindowTCN, since there is only 1 network output, stride at the output layer is insignificant and can be set to 1 (line 1). Similarly, since no other input apart from those being operated upon is necessary, dilation can be reduced to 1 (line 2). Decreasing dilation at the output layer increase strides in lower layers, and we calculate the striding multiplier s_m for the next layer (line 3) and update parameters for it on lines 5-6. Next, we optimize parameters at this layer by determining if any inputs are not needed. We compute the Greatest Common Divisor (GCD) of the new stride and dilation rate. GCD > 1 implies that some inputs at this layer are not needed, and the stride and dilation rate are optimized on lines 8-9; the stride rate multiplier for the next layer is set to the GCD (line 7). This process is repeated down to the input layer. If a reduction is performed at the input layer, then some inputs are no longer needed and inputs should be sub-sampled using the final value of s_m.

In Algorithm 4, we first initialize a dilation rate multiplier m_d to 1 on line 1. On line 4, we recompute strides for a given layer depending on whether the output needs to

Algorithm 4 SingleWindowTCN to Continual TCN

```
Input: $n \leftarrow$ Network Depth/Number of layers
Input: $r \leftarrow$ Desired Output Rate Reduction
Input: $s_{sw}[\ ] \leftarrow$ SingleWindow strides for each layer
Input: $d_{sw}[\ ] \leftarrow$ SingleWindow dilation rates for each layer
Output: $s[\ ] \leftarrow$ Continual strides for each layer
Output: $d[\ ] \leftarrow$ Continual dilation rates for each layer
1: $m_d \leftarrow 1$                      ▷ Dilation rate multiplier
2: $r' \leftarrow r$                          ▷ Temporary variable
3: for $i = 1$ to $n$ do
4:     $s[i] \leftarrow$ GCD($s_{sw}[i], r'$)
5:     $d[i] \leftarrow d_{sw}[i] \cdot m_d$
6:     $r' \leftarrow \frac{r'}{s[i]}$
7:     $m_d \leftarrow \frac{s[i]}{s_{sw}[i]} \cdot m_d$
8: $s[n] \leftarrow s[n] \cdot r'$
```

be calculated at a different rate than the stride specified by SingleWindowTCN. Next, we increase (as needed) the dilation rate based on the current value of the dilation rate multiplier. Finally, on lines 6 and 7, we update r to account whether the stride s at that layer accounts for a part of r rate reduction, and then recompute m_d depending on whether additional convolution outputs were inserted at the current layer, increasing the dilation rate at higher layers. If some portion of r was not accounted by the stride at any layer, it implies that striding exists at the output layer and the stride rate is updated on line 8.

V. RESULTS

In this section, we evaluate how well RT-TCN performs on metrics representing compute and memory when compared to 2 other baseline approaches for TCN inference.

A. Experimental Setup, Approaches, and Metrics

We use a TCN network similar to Figure 1. Unless otherwise specified, it has the following parameters:

- The depth of the network $n = 3$.
- All 3 layers have a kernel size of 3, i.e. $k = [3, 3, 3]$.
- The output layer has 1 convolution filter whereas other layers have 6, i.e. $n_f = [6, 6, 1]$.
- The network dilation rates $d = [1, 1, 2]$.
- The convolution strides $s = [2, 1, 1]$. This s results in an output reduction rate $r = 2$.
- ReLU is used as the non-linear activation function g in all layers. ReLU is a good activation function in devices with constrained compute since it doesn't require any FLOPs. Existing work on efficiently approximating *tanh* [21], *sigmoid* [21], and ELU [20] is available.
- Input sampling rate is assumed to be 100Hz. Compute for all approaches scales linearly with this rate.
- All values are stored in 4 byte floating-point.

Note that the receptive field size $T = 15$ at these settings.

Given a TCN, the following approaches can be used for computing TCN outputs in real-time:

1) SIMPLE - The last T inputs are buffered in memory and used to compute the output by computing each intermediate layer completely prior to the next layer.
2) SINGLEWINDOW - Same as SIMPLE, except strides and dilations computed by reducing the network using Algorithm 3 are used to compute network output.
3) RT-TCN - Given a TCN, different buffers are initialized for the input and intermediate layers via Algorithm 1. During inference, these generated buffers are used by RT-TCN (Algorithm 2) to compute the output.

If the TCN has been trained as a SingleWindowTCN, it must be converted to a continual TCN using Algorithm 4 when SIMPLE or RT-TCN are used to compute the TCN output, and vice versa when SINGLEWINDOW is used for computation (using Algorithm 3). Furthermore, since both SIMPLE and SINGLEWINDOW require buffering the last T inputs, they always require the same running memory.

We measure TCN inference performance for these approaches using the amount of running memory required for buffering data, and the number of multiplicative FLOPS required to compute convolutions. Note that the amount of read-only memory to store convolution parameters is same across all approaches, and is not included. We now vary some network parameters and show metrics for all 3 approaches, and outline guidelines using those results.

B. Varying Convolution Filters (n_f)

We vary the number of convolution filters (n_f) in the first 2 layers in tandem, and plot metrics in the 2 graphs on the left end of Figure 2. As n_f increases, the computation time for all 3 approaches increases on the order of n^2, as both the number of individual convolutions and inputs at the next layer increase. At the default setting ($n_f = [6, 6, 1]$), RT-TCN requires less compute than the other approaches while requiring the same amount of memory. While the relative ratios between the approaches are maintained as we vary n_f, the memory required by RT-TCN increases linearly while SINGLEWINDOW and SIMPLE use constant memory as the input buffer size does not change. This result is not unexpected, as RT-TCN requires intermediate layer buffers and the size of those buffers depend on n_f.

From these results, we can state the guideline that if memory is not constrained, or the number of input channels $c \gtrsim \text{AVERAGE}(n_f)$, RT-TCN can reduce computation compared to SINGLEWINDOW within a similar memory footprint. Furthermore, RT-TCN memory can be reduced by reducing filters in layers with high dilation rates, assuming network output quality is not significantly impacted.

C. Varying Network Dilation Rate (d)

As network dilation rate d is varied in tandem across all 3 layers (results in center left graphs in Figure 2), compute for SINGLEWINDOW and RT-TCN is not impacted, whereas compute for SIMPLE increases linearly. SIMPLE is inefficient at high dilation as it computes numerous intermediate convolutions that do not contribute to the single output. In fact, all graphs in Figure 2 demonstrate that SINGLEWINDOW is better than SIMPLE, as it is always better not to compute unnecessary convolutions. Additionally, RT-TCN requires less memory relative to SINGLEWINDOW as d increases. The receptive field size T increases super-linearly with d and SINGLEWINDOW requires more memory relative to RT-TCN, giving RT-TCN a slight edge at high dilation rates.

D. Varying Network Depth (n)

Next, we vary the network depth (n) from 2 layers to 8 layers (results in center right graphs in Figure 2). As we vary network depth, it is necessary to select kernel sizes, number of filters, strides and dilation rates for all layers appropriately. All layers always use a kernel size of 3 and 6 filters regardless of the value of n. Layer 1 has a stride of 2, and all remaining $n - 1$ layers have a stride of 1, which ensures that the output rate reduction is always 2 regardless of n. Similar to the default settings, layer n has a dilation of 2, and all other layers have a dilation rate of 1.

The graphs demonstrate one key result of this paper. As the number of layers increases, the CPU consumption

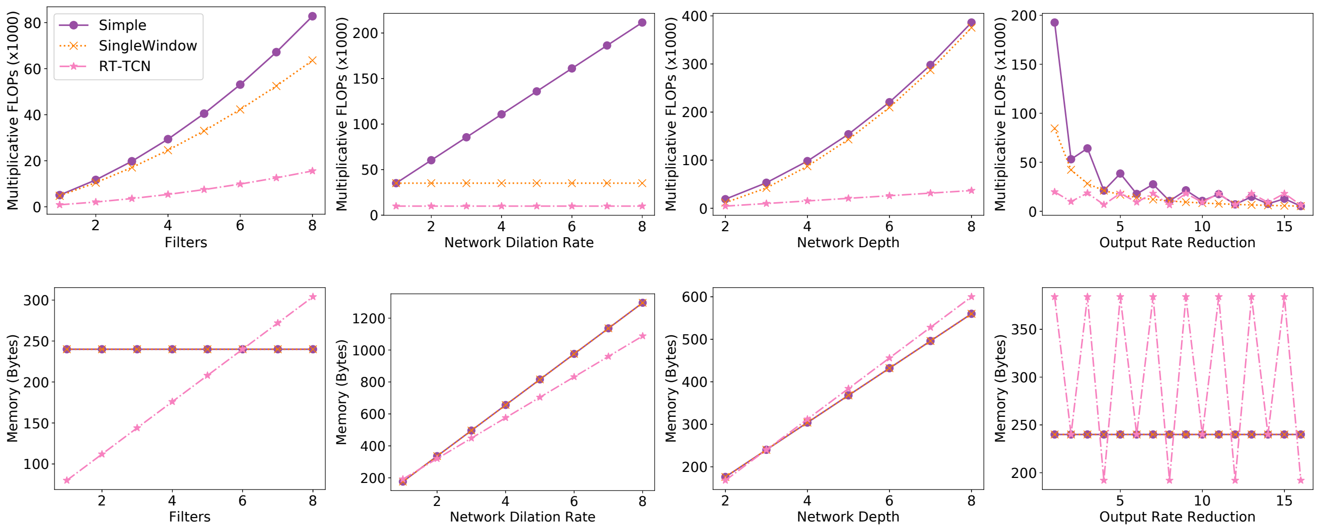


Fig. 2: We vary TCN parameters and plot multiplicative FLOPS as a measure of compute requirements (top row) and memory consumption in bytes as a measure of memory requirements (bottom row). At the left end, we vary number of filters (n_f) across all layers apart from the output layer in tandem. At the center-left, we vary network dilation rates (d) across all layers in tandem. At the center-right, we vary network depth (n). At the right end, we change the output rate reduction (r) assuming that the network was trained as a SingleWindowTCN.

for RT-TCN increases linearly with the number of layers because RT-TCN only performs at most 1 convolution per layer. In contrast, CPU consumption for both SIMPLE and SINGLEWINDOW increases super-linearly as increasing the network depth additionally increases the receptive field size, and more convolutions are performed at lower layers to compute the SingleWindowTCN output. In terms of memory consumption, the memory consumption for all 3 approaches increases linearly with the network depth. As the number of layers increase, the overhead for maintaining more intermediate buffers in RT-TCN is slightly more than the increasing receptive field size, so SIMPLE and SINGLEWINDOW consume slightly less memory relatively.

E. *Varying Output Rate Reduction (r)*

Assuming that the TCN has been trained as a SingleWindowTCN, and the frequency at which network output needs to be computed is a choice for a specific problem domain, we can induce different continual TCN architectures by varying the output reduction rate r (right end of Figure 2). Varying r changes the stride s and network dilation rate d of the resulting TCN (computed in Algorithm 4). As we vary r, the CPU required by SINGLEWINDOW is inversely proportional to r, and since the receptive field size T does not change, the memory requirement for SINGLEWINDOW is also constant.

On the other hand, the memory and compute requirements for RT-TCN can change dramatically as we vary r. For instance, a value of $r = 3$ effectively induces a dilation rate of 4 and a stride of 3 in the output layer, and requires the same memory and almost the same amount of computation as when $r = 1$. In fact it is better to choose a higher output frequency rate by setting $r = 2$, as in the induced continual TCN architecture the output layer has a dilation of 2 and stride of 1, and the TCN is much more efficient to compute in real-time. The compute requirements for SIMPLE are similarly affected as RT-TCN. These results demonstrate that it is necessary to choose a suitable value of output rate reduction r if that parameter can be freely selected. Additionally, in situations where the output is not needed often when compared to the input rate ($r \gtrsim T/2$) it is faster to compute the network output from buffered inputs directly, and SINGLEWINDOW should be used.

VI. CONCLUSION

In this work, we have presented RT-TCN, an algorithm which buffers the output of intermediate convolutions to ensure no redundant computation is performed when a TCN is evaluated continually in real-time. Through empirical results we have demonstrated settings at which RT-TCN can reduce the computational cost for inference compared to the SINGLEWINDOW approach, where a time-slice of time-series data is passed through a convolution network layer-by-layer, and intermediate convolution operations are not retained in memory even if they will be needed to compute future outputs. We have also presented guidelines and results that are intended to help with selection of network architecture and inference approach when using TCNs. We believe the work presented in this paper will help in the application of TCNs for solving problems on devices with limited resources.

In future work, we aim to extend RT-TCN to more complex and deeper network architectures. While not every TCN architecture is suited for the input and intermediate convolution reuse proposed by RT-TCN, it should be possible to incorporate some common TCN techniques such as residual connections in RT-TCN which we would like to explore.

References

[1] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. A survey on modern trainable activation functions. *arXiv preprint arXiv:2005.00817*, 2020.

[2] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[3] Marco Carreras, Gianfranco Deriu, Luigi Raffo, Luca Benini, and Paolo Meloni. Optimizing temporal convolutional network inference on FPGA-based accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2020.

[4] Shuo-Yiin Chang, Bo Li, Gabor Simko, Tara N Sainath, Anshuman Tripathi, Aäron van den Oord, and Oriol Vinyals. Temporal modeling using dilated convolution and gating for voice-activity-detection. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[5] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, et al. Tensorflow lite micro: Embedded machine learning on tinyml systems. *arXiv preprint arXiv:2010.08678*, 2020.

[6] Google. *TensorFlow graph optimization with Grappler*, Accessed August 5, 2020. `https://www.tensorflow.org/guide/graph_optimization`.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[8] Carnegie Mellon University Locus Lab. *Sequence Modeling Benchmarks and Temporal Convolutional Networks (TCN)*, Accessed August 5, 2020. `https://github.com/locuslab/TCN`.

[9] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient neural network kernels for arm Cortex-M cpus, 2018.

[10] Pedro Lara-Benítez, Manuel Carranza-García, José M Luna-Romera, and José C Riquelme. Temporal convolutional networks applied to energy-related time series forecasting. *Applied Sciences*, 2020.

[11] Pedro Lara-Benıtez, Manuel Carranza-Garcıa, and Jose C Riquelme. An experimental review on deep learning architectures for time series forecasting. *International Journal of Neural Systems*, 2020.

[12] Siddique Latif, Muhammad Usman, Rajib Rana, and Junaid Qadir. Phonocardiographic sensing using deep learning for abnormal heartbeat detection. *IEEE Sensors Journal*, 2018.

[13] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks for action segmentation and detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[14] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, pages 541–551.

[15] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*. Springer, 1999.

[16] MathWorks. *Data Layout Considerations in Deep Learning*, Accessed August 5, 2020. `https://www.mathworks.com/help/gpucoder/ug/data-layout-considerations-gpu-deep-learning.html`.

[17] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[18] Philippe Rémy. *Keras TCN*, Accessed August 5, 2020. `https://github.com/philipperemy/keras-tcn`.

[19] Moritz Scherer, Michele Magno, Jonas Erb, Philipp Mayer, Manuel Eggimann, and Luca Benini. Tinyradarnn: Combining spatial and temporal convolutional neural networks for embedded gesture recognition with short range radars. *arXiv preprint arXiv:2006.16281*, 2020.

[20] Nicol N Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 1999.

[21] P Sibi, S Allwyn Jones, and P Siddarth. Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, 2013.

[22] 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*, 1989.

[23] Renzhuo Wan, Shuping Mei, Jun Wang, Min Liu, and Fan Yang. Multivariate temporal convolutional network: A deep neural networks approach for multivariate time series forecasting. *Electronics*, 2019.

[24] Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline. In *International joint conference on neural networks (IJCNN)*. IEEE, 2017.

[25] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. Deep convolutional neural networks on multichannel time series for human activity recognition. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.