Continual Inference: A Library for Efficient Online Inference with Deep Neural Networks in PyTorch

Lukas Hedegaard(✉) and Alexandros Iosifidis

Department of Electrical and Computer Engineering,
Aarhus University, Aarhus, Denmark
{lhm,ai}@ece.au.dk

Abstract. We present *Continual Inference*, a Python library for implementing Continual Inference Networks (CINs), a class of Neural Networks designed for redundancy-free online inference. This paper offers a comprehensive introduction and guide to CINs and their implementation, as well as best-practices and code examples for composing basic modules into complex neural network architectures that perform online inference with an order of magnitude less floating-point operations than their non-CIN counterparts. Continual Inference provides drop-in replacements of PyTorch modules and is readily downloadable via the Python Package Index and at www.github.com/lukashedegaard/continual-inference.

Keywords: Online inference · Continual Inference Network · Deep Neural Network · Python · PyTorch · Library

1 Introduction

Designing and implementing Deep Neural Networks, which offer good performance in online inference scenarios, is an important but overlooked discipline in Deep Learning and Computer Vision. Research in areas such as Human Activity Recognition focuses heavily on improving accuracy on select benchmark datasets with limited focus on computational complexity and still less on efficient online inference capabilities. Yet, important real-life applications such as human monitoring [13,18], driver assistance [3], and autonomous vehicles depend on performing predictions on a continual input stream with low latency and low energy consumption.

Continual Inference Networks (CINs) [7–9], are a recent family of Deep Neural Networks, which can accelerate a wide range of architectures for time-series processing (e.g., CNNs and Transformers) during online inference, even though source networks may have been trained exclusively for offline processing.

This paper provides comprehensive introduction to CINs (Sect. 2), the guiding principles of their design and implementation via the Continual Inference library (Sect. 4), and summarizes and compares achieved reductions in stepwise computational complexity and memory-usage using the library (Sect. 4).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

2 Continual Inference Networks

Originally introduced in [9] and subsequently elaborated in [7,8], Continual Inference Networks denote a variety of Neural Network, which can operate without redundancy during online inference on a continual input stream, as well as offline during batch inference. Specifically, CINs comply with Definition 1 [7]:

Definition 1 (Continual Inference Network). *A Continual Inference Network is a Deep Neural Network, which*

- *is capable of continual step inference without computational redundancy,*
- *is capable of batch inference corresponding to a non-continual Neural Network,*
- *produces identical outputs for batch inference and step inference given identical receptive fields,*
- *uses one set of trainable parameters for both batch and step inference.*

Many prior networks can be viewed as CINs. This includes networks, which perform their task within a single time-step (e.g., object detection and image recognition models), or which inherently process temporal data step-by-step (e.g., Recurrent Neural Networks such as LSTMs [10] and GRUs [2]). Some network types, however, are limited to batch inference exclusively. These include Convolutional Neural Networks (CNNs) with temporal convolutional components (e.g., 3D CNNs), as well as Transformers with tokens spanning the temporal dimension. While they can in principle be used for online inference, it is an inefficient process, where input steps are assembled to full (spatio-)temporal batches and fed to the network in a sliding window fashion, with many redundant intermediary computations as a result.

While some specialty architectures have been devised to let 3D convolutional network variants make predictions step by step [11,16], and accordingly also qualify as CINs, these were not weight-compatible with regular 3D CNNs. Recently, *Continual* 3D Convolutions [9] changed this. Through a reformulation of the 3D convolution to compute outputs for each time-step individually rather than for the whole spatio-temporal input at once, well-performing 3D CNNs such as X3D [4], Slow [5], and I3D [1] trained for Trimmed Activity Recognition were re-implemented to execute step by step without any re-training. Likewise, Spatio-temporal Graph Convolutional Networks for Skeleton-based Action Recognition [14,15,20], which originally operated only on complete sequences of skeleton graphs, were transformed to perform stepwise inference as well though a continual formulation of their Spatio-temporal Graph Convolution blocks [8]. Temporal Transformer networks had also been restricted to operate on batches until a *Continual* Multi-head Attention (*Co*MHA) [7] was introduced, which is weight-compatible with the original MHA [19], while being able to compute updated outputs for each time step.

With these innovations, many existing DNNs can be converted to operate efficiently during online inference. In general, non-continual networks, which are transformed to continual ones attain reductions in per-step computational complexity in proportion to the temporal receptive field of the network. In some cases, these savings can amount to multiple orders of magnitude [8]. Still, the implementation of Continual Inference Networks with temporal convolutions and Multi-head Attention in frameworks such as PyTorch [12] requires deep knowledge and practical experience with CINs. With the Continual Inference library described in the next section, we hope to change this.

3 Library Design

3.1 Principles

The fundamental feature of CINs, that networks are flexible and perform well on both online inference and batch inference, is a guiding principle in the design of the Continual Inference library as well: Refactoring of existing implementations in pure PyTorch should be straightforward. In the following, we will adopt the Python import abbreviations `import continual as co` and `from torch import nn`. The library follows Principle 1 to ensure that `co` modules can be used as drop-in replacements for `nn` modules without behavior change:

Principle 1 (Compatibility with PyTorch). `co` *modules with identical names to* `nn` *modules also have:*

1. *identical* `forward`*,*
2. *identical model weights,*
3. *identical or extended constructors,*
4. *identical or extended supporting functions.*

Before proceeding to the enhanced functionality of `co` modules, let us state our assumption to the input format:

Assumption 1 (Order of input dimensions). *Inputs to* `co` *modules use the order* $(B, C, T, S_1, S_2, ...)$ *for multi-step inputs and* $(B, C, S_1, S_2, ...)$ *for single-step inputs, where* B *is the batch size,* C *is the input channel size,* T *is the temporal size, and* S_n *are additional optional dimensions.*

The core difference between Continual Inference Networks and regular networks is their ability to efficiently compute results for each time-step. Besides the regular `forward` function found in `nn` modules, `co` modules add multiple call modes that allow for continual inference with a simple interface:

Principle 2 (Call modes). `co` *modules provide three forward operations:*

1. `forward`*: takes a (spatio-) temporal input and operates identically to the* `forward` *of an* `nn` *module,*
2. `forward_step`*: takes a single time-step as input without a time-dimension and produces an output corresponding to* `forward`*, had it's input been shifted by one time-step, given identical prior inputs.*
3. `forward_steps`*: takes multiple time-steps as input and produces outputs identical to applying* `forward_step` *the number of times corresponding to the temporal size of the input.*

Furthermore, the `__call__` *method of* `co` *modules can be changed to use any of the three by either setting the* `call_mode` *attribute of the module or applying the* `co.call_mode()` *context with a string spelling out the wanted forward type.*

Let us exemplify Principle 2 in practice. Example 1.1 shows how the different forward functions introduced in Principle 2.1 can be used. Principle 2.2 is illustrated in Example 1.2.

```python
import torch
import continual as co

con = co.Conv3d(in_channels=4,
                out_channels=8,
                kernel_size=3)
assert con.delay == 2
assert con.receptive_field == 3

reg = torch.nn.Conv3d(in_channels=4,
                      out_channels=8,
                      kernel_size=3)
# Reuse weights
con.load_state_dict(reg.state_dict())

x = torch.randn((2, 3, 5, 6, 7))  # B,C,T,H,W
y = con.forward(x)
assert torch.equal(y, reg.forward(x))

# Multiple steps
firsts = con.forward_steps(x[:, :, :4])
assert torch.allclose(firsts, y[:, :, : con.delay])

# Single step
last = con.forward_step(x[:, :, 4])
assert torch.allclose(last, y[:, :, con.delay])
```

Example 1.1. Definition and usage of `co.Conv3d` and its forward modes.

```python
net(x)            # Invokes `forward` by default

net.call_mode = "forward_step"
net(x[:, :, 0])   # Invokes `forward_step`

with co.call_mode("forward_steps"):
    net(x)        # Invokes `forward_steps`

net(x[:, :, 0])   # Invokes `forward_step` again
```

Example 1.2. Changing the `call_mode` for a continual module `net`.

Continual modules, which use information from multiple time-steps, are inherently stateful. Whenever `forward_step` or `forward_steps` is invoked, intermediary results needed for future step results are optimistically computed and stored. Principle 3 states the rules for state-manipulation and updates.

Principle 3 (State). *Module state is updated according to the following rules:*

- `forward_step` *and* `forward_steps` *use and update state by default.*
- *Step results may be computed without updating internal state by passing* `update_state=False` *to either* `forward_step` *or* `forward_steps`*.*
- `forward` *neither uses nor updates state.*
- *Module state can be wiped by invoking the* `clean_state()` *method.*
- *A module produces non-empty outputs after its has conducted a number of stateful forwards steps corresponding to its* `delay`*.*

Regular `nn` modules predominantly operate on input batches in an offline setting and do not have a built-in concept of delay. `co` modules on the other hand are designed to operate on time-series. Since `co` modules often integrate information over multiple time-steps and online operation is causal by nature, some modules produce the output corresponding to a given input only after observing additional steps. For instance, a `co.Conv1d` module with `kernel_size = 3` produces an output from the third input step as illustrated in Fig. 1. The delay of a module is calculated according to Principle 4:

Principle 4 (Delay). `co` *modules produce step outputs that are delayed by*

$$d = f - p - 1 \tag{1}$$

steps relative to the earliest input step used in the computation, where f is the receptive field and p is the temporal padding.

While padding is used in regular networks to retain the size of feature-maps in consecutive layers, this interpretation of temporal padding does not make sense in the context of an infinite, continual input, as handled by CINs. Instead, we may interpret padding as a reduction in delay. For instance, a `co.Conv1d` module with `kernel_size = 3` and `padding = 2` has a delay of zero, because the padded zeros already "saturated" the state before-hand. This is illustrated in Fig. 2. Considering, that `co` modules expect an infinite and continual input stream, end-padding padding is omitted by default. If an end-padding is required, the library supports its use by either passing manually defined zeros as steps or by setting `pad_end = True` for an invocation of the `forward_steps` function.

Similar to padding, the stride of a `co` module impacts the timing of the outputs. Specifically, stride results in empty outputs every $(s - 1)/s$ outputs, as well as larger delays for downstream network modules through increased receptive fields. This is stated in Principles 5 and 6.

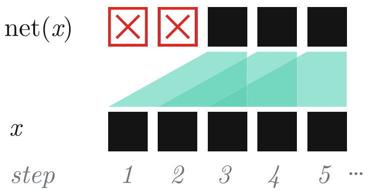


Fig. 1. Sketch of delay and receptive field. Here, the stepwise operation of a `co` module `net` with `receptive_field = 3` is illustrated. ■ are non-zero step-features and ☒ are empty outputs.

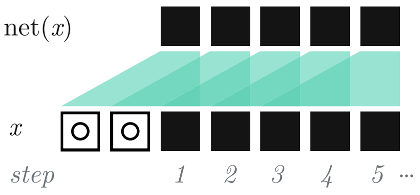


Fig. 2. Sketch of how padding reduces delay. Here, the stepwise operation of a `co` module `net` with `receptive_field = 3`, `padding = 2` is illustrated. ⊡ are padded zeros and ■ are non-zero step-features.

Principle 5 (Stride and prediction rate). *For neural network of N modules with strides $s^{(i)}, i \in \{1..N\}$, the accumulated stride at any given layer is*

$$s_{acc}^{(i)} = s^{(i)} \cdot s_{acc}^{(i-1)} \quad i \in 1..N \tag{2}$$

$$s_{acc}^{(0)} = s^{(0)}. \tag{3}$$

Equivalently, the resulting network stride is

$$s_{NN} = \prod_{i=1}^{N} s^{(i)}, \tag{4}$$

and the network prediction rate is

$$r_{NN} = 1/s_{NN}. \tag{5}$$

Accordingly, the outputs of a `co` *network are empty every $(s_{NN} - 1)/s_{NN}$ steps.*

Principle 6 (Accumulated delay). *The accumulated receptive field of a downstream module i in a network of N modules is given by:*

$$f_{acc}^{(i)} = f^{(i)} + (f_{acc}^{(i-1)} - 1)s^{(i)}, \quad i \in 1..N \tag{6}$$

$$f_{acc}^{(0)} = f^{(0)}. \tag{7}$$

The accumulated delay of layer i in a network is

$$d^{(i)} = f_{acc}^{(i)} - p_{acc}^{(i)} - 1, \tag{8}$$

where the accumulated padding p_{acc} is given by

$$p_{acc}^{(i)} = p^{(i)} \cdot s_{acc}^{(i-1)}, \quad i \in 1..N, \tag{9}$$

$$p_{acc}^{(0)} = p^{(0)}. \tag{10}$$

Figure 3 illustrates a mixed example, where the first layer of a two-layer network has `padding = 2` and `stride = 2`. Noting layer attributes in consecutive order, and using Eqs. 2 to 10, the example has the following network attributes:

$$\begin{aligned}
s &= \{2, \quad 1\} \\
p &= \{2, \quad 0\} \\
s_{acc} &= \{2, \quad 2 \cdot 1 = 2\} \\
p_{acc} &= \{2, \quad 2 + 2 \cdot 0 = 2\} \\
f_{acc} &= \{3, \quad 3 + (3-1) \cdot 2 = 7\} \\
d_{acc} &= \{3 - 2 - 1 = 0, \quad 7 - 2 - 1 = 4\} \\
s_{NN} &= s_{acc}^{(1)} = 2 \\
r_{NN} &= 1/s_{NN} = 1/2 \\
d_{NN} &= d_{acc}^{(1)} = 4.
\end{aligned}$$

Before continuing onto the specific modules, we have to discuss a final principle of CINs, namely that of parallel modules.

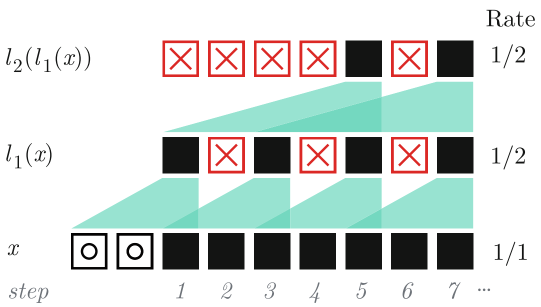


Fig. 3. A mixed example of delay and outputs under padding and stride. Here, we illustrate the stepwise operation of two `co` module layers, l_1 with `receptive_field = 3`, `padding = 2`, and `stride = 2` and l_2 with `receptive_field = 3`, no padding and `stride = 1`. ⊡ denotes a padded zero, ■ is a non-zero step-feature, and ☒ is an empty output.

Principle 7 (Parallel modules). *Modules can be arranged in parallel to execute on each their separate stream of data under the following rules:*

- *Parallel modules follow the same global clock.*
- *The delay of a collection of parallel modules is the maximum delay of any module in the collection.*
- *If the merger of parallel step values includes an empty value, then the resulting step output of the merger is also empty.*

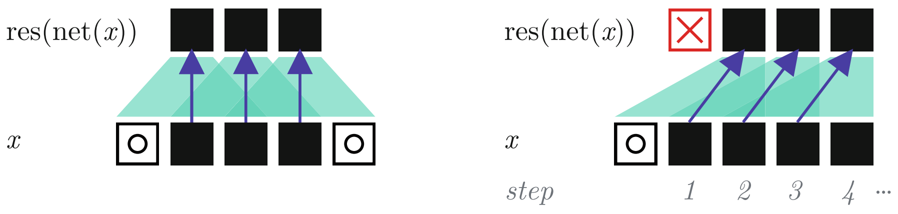
A discussion of residual connections provides a practical example for Principle 7.

Residual Connections. The residual connection is a simple but crucial tool for avoiding vanishing and exploding gradients; by adding the input of a module to its output, gradients can flow freely through models with hundreds of layers. Without exaggeration, we can state that almost all recent deep architectures at the time of writing use some form of residual connection [5,6,19,20]. Yet, their implementation in Continual Inference Networks may not follow common intuition in all cases. Let us first consider the residual connection during regular `forward` operation as found in a non-continual residual shown in Fig. 4a. Here, the wrapped module will almost always use padding to ensure equal input and output shapes (known as "equal padding"). For a module with receptive field three, we would thus have a padding of one. In this case, the `forward` computation of the residual amounts to adding the input to the output of the convolution. However, the implementation of `forward_step` illustrated in Fig. 4b is different. Since the first output uses information from the second step, the module has a delay of one. Accordingly, the residual connection requires a delay of one as well.

Now consider the same scenario but without padding. This will be quite foreign to many Deep Learning practitioners, and it is not clear how exactly to align residuals. We will use a separate module to shrink the residual by an equivalent amount as the wrapped module. Of the possible alignment choices, a sensible approach is to discard the border values to align the feature maps on *center*. Contrary to other alignment forms, this has the benefit of weight-compatibility between the no-padding case and the case with equal padding described in the former paragraph. The outputs of step 3 in Figs. 4b and 5b are equal given the same weights and inputs. However, two issues arise:

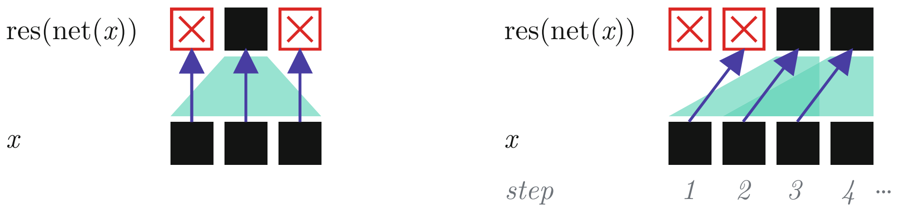
1. Delay mismatch: While the residual connection has a delay of one, the wrapped module has a delay of two.
2. Mix of empty and non-empty results: C.f. the differences in delay, the residual will start producing non-empty outputs before the wrapped module.

Principle 7 helps us navigate this. Despite the internal delay mismatch, the delay of the whole residual module corresponds to the largest delay, in this case two. Consequently, the whole residual module produces outputs from the third step, despite the fact that the delayed input already has non-empty outputs from the second step. Both of these issues can also be avoided if we force residuals to employ the same delay as the wrapped module. This corresponds to a *lagging* alignment. However, using such a strategy breaks weight compatibility between the same residual modules with and without padding.



(a) `forward` (b) `forward_step`

Fig. 4. Residual connections ↑ over a module with receptive field of size ▲ and padding one ("equal padding") ⊡. ☒ are empty outputs.



(a) `forward` (b) `forward_step`

Fig. 5. Centered residual connections ↑ over a module with receptive field of size ▲ and no padding. ☒ are empty outputs.

3.2 Core Modules

Designed as an augmentation of PyTorch, the Continual Inference library provides a collection of basic building blocks for composing neural networks. Following Principle 1, we use the same public interfaces as PyTorch, i.e. class constructor, function names and arguments, and attribute names, to ensure that `co` modules can be used as drop-in replacements for `nn` modules. The basic modules can be categorized as follows:

- Convolutions [9]: `co.Conv1d`, `co.Conv2d`, ...
- Pooling: `co.AvgPool1d`, `co.MaxPool1d`, ...
- Linear: `co.Linear`.
- Transformer [7]: `co.TransformerEncoder`, ...
- Shape: `co.Delay`, `co.Reshape`.
- Arithmetic: `co.Lambda`, `co.Add`, ...

Here, the `MultiheadAttention` implementation is a special case, which features two distinct versions of continual operation: 1) "single-output", where only the attention output corresponding to the latest input is produced, and 2) "retrospective", where updates to prior outputs are also produced retrospectively. The details of this are explained in greater detail in [7]. Linear `co` modules follow the `nn` modules closely, but ensure compatibility of dimension c.f. Assumption 1. `co.Delay` adds a specified delay to the input stream. This is handy for aligning

Table 1. Composition modules.

Module	Description
`Sequential`	Arrange modules sequentially
`Broadcast`	Broadcast one stream to multiple parallel streams
`Parallel`	Apply modules in parallel, each on a separate stream
`Reduce`	Reduce multiple input streams into one
`Residual`	Add a residual connection for a wrapped module
`Conditional`	Conditionally invoke a module (or another) at runtime

the delay of multiple streams as required by residual connections (see Sect. 3.1). `co.Lambda` allows a user to pass in functions and functors that are applied stepwise to the inputs. Besides the above list of tailor-made modules, the Continual Inference library has interoperability with most activation functions (`nn.ReLU`, `nn.Softmax`, etc.), normalisation layers (`nn.BatchNorm1d`, `nn.LayerNorm`, etc.), and `nn.DropOut` when used within the composition modules as presented in Sect. 3.3. The full list of compatible modules can be found at www.github.com/lukashedegaard/continual-inference.

3.3 Composition Modules

In PyTorch, modules are composed by either by using the `nn.Sequential` container or by creating a new class which inherits from `nn.Module` and manually controls data flow within the `forward` function. While the latter is commonly used to handle complex modules in a simple and easily debuggable manner, it is not necessarily the simplest approach for implementing complex Continual Inference Networks. In addition to defining the basic forward flow, a CIN implementation also needs to handle stepwise computations, which require meticulous alignment of delays if Principle 2 is to be kept.

Instead, we expand the container interface of PyTorch to include modules for parallel and conditional processing (Table 1). While each module is simple in nature, they can be used to compose complex neural network architectures, which retain all the principles in Sect. 3.1 without explicitly needing to consider them. A brief overview and description of each `co` container module is given in Sect. 3.3. To get a practical understanding of these, we will give implementation examples of two common architecture blocks, the residual connection as discussed in Sect. 3.1 and an Inception module [17].

Example 1.3 shows three equivalent implementations of a residual 3D convolution block. `res1` is the verbose version, in which `co.Broadcast` is used to split a single input into two parallel stream, `co.Parallel` specifies that `conv` handles the first stream, while a delay is used on the second. `co.Reduce` merges the streams via an add reduce operation. Due to the commonality of broadcast-apply-reduce operations, the library features a `co.BroadcastReduce` shorthand to specify such composition more succinctly. Even shorter, `co.Residual` can

```
conv = co.Conv3d(1, 1, kernel_size=3, padding=1)

res1 = co.Sequential(co.Broadcast(2),
                     co.Parallel(conv, co.Delay(1)),
                     co.Reduce("sum"))

res2 = co.BroadcastReduce(conv, co.Delay(1))

res3 = co.Residual(conv)
```

Example 1.3. Equivalent implementations of a residual block.

automatically infer the needed delay from the module it wraps. Other reduction functions can be specified in `co.BroadcastReduce` and `co.Residual` using the `reduce` argument, which is `"sum"` by default. The code in Example 1.3 correspond to Fig. 4. The centered residual module in Fig. 5 is easily specified as `co.Residual(conv, residual_shrink=True)` where `conv` has `padding = 0`.

```
def norm_relu(conv):
    return co.Sequential(conv,
                         nn.BatchNorm3d(conv.out_channels),
                         nn.ReLU())

inception_module = co.BroadcastReduce(
    co.Conv3d(192, 64, 1),
    co.Sequential(
        norm_relu(co.Conv3d(192, 96, 1)),
        norm_relu(co.Conv3d(96, 128, 3, padding=1)),
    ),
    co.Sequential(
        norm_relu(co.Conv3d(192, 16, 1)),
        norm_relu(co.Conv3d(16, 32, 5, padding=2))
    ),
    co.Sequential(
        co.MaxPool3d(kernel_size=(1, 3, 3),
                     padding=(0, 1, 1),
                     stride=1),
        norm_relu(co.Conv3d(192, 32, 1)),
    ),
    reduce="concat",
)
```

Example 1.4. *Continual* Inception module using a mix of `co` and `nn` modules.

We can showcase a more advanced application of parallel streams by considering an Inception module [17]. An Inception module broadcasts the input into four streams and applies convolution of varying kernel sizes in parallel before concatenating the channels to produce one output. Without the `co` container modules, it would be complicated to keep track of and align delays of the different branches to create valid `forward`, `forward_step`, and `forward_steps` methods. Using `co.Sequential`, which automatically sums up delays, and `co.BroadcastReduce`, which automatically adds delays to match the branch with highest inherent delay, the implementation becomes simple as shown in Example 1.4.

Table 2. Dataset performance, parameter count, maximum allocated memory (Max mem.), and floating-point operations (FLOPs) of continual and non-continual models on video and spatio-temporal graph classification datasets. Subscript$_{xx}$ denotes expanded temporal average pooling, b1 and b2 denote one and two block transformer decoders, and superscript* indicates architectures where network stride was reduced to one. Parentheses show the improvement/deterioration of the continual model relative to the corresponding non-continual model. The noted metrics were originally presented in [7–9].

Model	Dataset performace (%)		Params (M)	Max mem. (MB)	FLOPs (G)
	Kinetics-400 (Acc.)				
X3D-L	69.3		06.2	240.7	19.17
*Co*X3D-L$_{64}$	71.6 (+2.3)		06.2	184.4 0(75%)	01.25 0(↓ 15.34×)
X3D-M	67.2		03.8	126.3	04.97
*Co*X3D-M$_{64}$	71.0 (+3.8)		03.8	069.0 0(55%)	00.33 0(↓ 15.06×)
X3D-S	64.7		03.8	061.3	02.06
*Co*X3D-S$_{64}$	67.3 (+2.6)		03.8	042.0 0(69%)	00.17 0(↓ 12.12×)
Slow-8×8	67.4		32.5	266.0	54.87
*Co*Slow$_{64}$	73.1 (+5.7)		32.5	176.4 0(66%)	06.90 00(↓ 7.95×)
I3D	64.0		28.0	191.6	28.61
*Co*I3D$_{8}$	59.6 (−4.4)		28.0	235.9 (123%)	05.68 00(↓ 5.04×)
	THUMOS14 (mAP)	TVSeries (mcAP)			
OadTR-b2	64.2	89.0	15.9	067.6	01.08
*Co*OadTR-b2	64.4 (+0.2)	88.2 (−0.8)	15.9	071.7 (106%)	00.41 00(↓ 2.61×)
OadTR-b1	64.4	89.1	09.6	043.3	00.67
*Co*OadTR-b1	64.5 (+0.1)	88.0 (−1.1)	09.6	045.1 (104%)	00.01 0(↓ 63.49×)
	NTU RGB+D 60 (Acc.) *X-Sub*	*X-View*			
ST-GCN	86.0	93.4	03.1	045.3	16.73 ↓ 000.0×
*Co*ST-GCN*	86.3 (+0.3)	93.8 (+0.4)	03.1	036.1 0(80%)	00.16 (↓ 107.7×)
AGCN	86.4	94.3	03.5	048.4	18.69 ↓ 000.0×
*Co*AGCN*	84.1 (−2.3)	92.6 (−1.7)	03.5	037.4 0(77%)	00.17 (↓ 108.8×)
S-TR	86.8	93.8	03.1	074.2	16.14 ↓ 000.0×
*Co*S-TR*	86.3 (−0.3)	92.4 (−1.4)	03.1	036.1 0(49%)	00.15 (↓ 107.6×)

4 Performance Comparisons

Using the basic `co` modules and composition building blocks, continual versions of advanced neural networks have been implemented in multiple recent works with manyfold speedups and significant reductions in memory consumption during online inference [7–9]. Specifically, the 3D-CNNs *Co*X3D, *Co*I3D, and *Co*Slow for video-based Human Activity Recognition were proposed in [9]; the Transformer *Co*OadTR for Online Action Detection in [7]; and Spatio-temporal Graph Convolutional Networks *Co*ST-GCN, *Co*AGCN, and *Co*S-TR for Skeleton-based Action Recognition in [8]. While direct conversion from regular to continual versions of the above noted architectures works well in accelerating inference in itself, further improvements can be achieved by exploiting some core characteristics of CINs: in [9], accuracy was improved by increasing model receptive fields through expansions of temporal global average pooling to 64 steps, and in [8], the stride of temporal convolutions was reduced to one to increase prediction rates. Table 2 presents a summary of benchmark performance, computational complexity, and maximum allocated memory on GPU for each of these networks alongside with their non-continual counterparts [7–9].

5 Conclusion

We presented Continual Inference, an easy-to-use library for implementing Continual Inference Networks in Python. Following interfaces closely, the components provided in the library are backwards-compatible drop-in replacements for PyTorch modules, which add the capability of redundancy-free online inference without the need for intimate knowledge of CINs nor their meticulous low-level implementation. Having shown the vast computational advantages of CINs over regular neural networks in multiple settings of video and spatio-temporal graph classification, we hope that this library will contribute to the adoption of CINs and the advancement of use-cases requiring low-latency online inference under recourse constraints in general.

Acknowledgement. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871449 (OpenDR).

References

1. Carreira, J., Zisserman, A.: Quo Vadis, action recognition? A new model and the kinetics dataset. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4724–4733 (2017)
2. Cho, K., van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: encoder-decoder approaches. In: Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, pp. 103–111 (2014)

3. Enkelmann, W.: Video-based driver assistance-from basic functions to applications. Int. J. Comput. Vis. (IJCV) **45**(3), 201–221 (2001)
4. Feichtenhofer, C.: X3D: expanding architectures for efficient video recognition. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2020)
5. Feichtenhofer, C., Fan, H., Malik, J., He, K.: SlowFast networks for video recognition. In: IEEE/CVF International Conference on Computer Vision (ICCV), pp. 6201–6210 (2019)
6. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
7. Hedegaard, L., Bakhtiarnia, A., Iosifidis, A.: Continual transformers: redundancy-free attention for online inference. In: International Conference on Learning Representations (ICLR) (2023)
8. Hedegaard, L., Heidari, N., Iosifidis, A.: Online skeleton-based action recognition with continual spatio-temporal graph convolutional networks. Preprint arXiv:2203.11009 (2022)
9. Hedegaard, L., Iosifidis, A.: Continual 3D convolutional neural networks for real-time processing of videos. In: Avidan, S., Brostow, G., Cissé, M., Farinella, G.M., Hassner, T. (eds.) ECCV 2022. LNCS, vol. 13664, pp. 369–385. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19772-7_22
10. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**, 1735–1780 (1997)
11. Köpüklü, O., Hörmann, S., Herzog, F., Cevikalp, H., Rigoll, G.: Dissected 3D CNNs: temporal skip connections for efficient online video processing. Preprint arXiv:2009.14639 (2020)
12. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, vol. 32, pp. 8024–8035. Curran Associates, Inc. (2019)
13. Pigou, L., van den Oord, A., Dieleman, S., Van Herreweghe, M., Dambre, J.: Beyond temporal pooling: recurrence and temporal convolutions for gesture recognition in video. Int. J. Comput. Vis. (IJCV) **126**(2), 430–439 (2018)
14. Plizzari, C., Cannici, M., Matteucci, M.: Skeleton-based action recognition via spatial and temporal transformer networks. Comput. Vis. Image Underst. **208**, 103219 (2021)
15. Shi, L., Zhang, Y., Cheng, J., Lu, H.: Two-stream adaptive graph convolutional networks for skeleton-based action recognition. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 12026–12035 (2019)
16. Singh, G., Cuzzolin, F.: Recurrent convolutions for causal 3D CNNs. In: IEEE/CVF International Conference on Computer Vision Workshop (ICCVW), pp. 1456–1465 (2019)
17. Szegedy, C., et al.: Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1–9 (2015)
18. Tavakolian, M., Hadid, A.: A spatiotemporal convolutional neural network for automatic pain intensity estimation from facial dynamics. Int. J. Comput. Vis. (IJCV) **127**(10), 1413–1425 (2019)
19. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems (NeurIPS), vol. 30, pp. 5998–6008 (2017)
20. Yan, S., Xiong, Y., Lin, D.: Spatial temporal graph convolutional networks for skeleton-based action recognition. In: AAAI Conference on Artificial Intelligence, pp. 7444–7452 (2018)