Supplementary Material

A Scrambling

We provide further details on the Scramble function in Algorithm 1, which is responsible of evaluating and possibly replacing one of the keys, at each time step. In particular, a generic i-th key is marked as not used enough if $\mu_i < \tau^\mu$, while it is too old if $\eta_i \geq \tau^\eta$, given two custom positive thresholds τ^μ and τ^η. We label as *weak* a key that fulfills both the conditions, and we consider replacing it with the current input x if it is significantly different from all the current keys.

Vector μ is initialized with zeros, while the entries in η are set to τ^η, so that, at the beginning, all keys are weak. Before updating keys with Eq. 6, if there exists at least a weak key and x falls "far away" from $K_\dagger$, i.e., the maximum similarity score returned by sim is smaller than τ^α (indicating that x is somewhat different from all the entries of K), the CNU replaces the weak key with x itself (actually $\psi(x)$), zeroing its usage counter.[3] Moreover, instead of randomly re-initializing the memory unit associated to the replaced key, the CNU initializes it to $M_\dagger$, to favour a warm-start of the memory values. Algorithm 2 formalizes the Scramble function.

Algorithm 2 The Scramble function of Algorithm 1.

```
function SCRAMBLE($x$, $K$, $M$, $s$, $\dagger$, $\mu$, $\eta$)
    $\mathcal{W} \leftarrow \{z\colon \mu_z < \tau^\mu \wedge \eta_z \geq \tau^\eta\}$        ▷ Set collecting indices of weak keys
    if $s_\dagger < \tau^\alpha \wedge \mathcal{W} \neq \emptyset$ then        ▷ If the current match is loose...
        $j \leftarrow \arg\max_{z \in \mathcal{W}}\{\eta_z\}$        ▷ The weakest key is the oldest
        $K_j \leftarrow \psi(x)$        ▷ Replace weakest key
        $M_j \leftarrow M_\dagger$        ▷ Warm-start for replaced memory unit
        $\mu_j \leftarrow 0$        ▷ Reset usage counter
        $\dagger \leftarrow j$        ▷ Update winning key index
    end if
return $K$, $M$, $\dagger$, $\mu$
end function
```

B Batched Computations

In Algorithm 3 we report the same operations of Algorithm 1 when a mini-batch of size b is provided by the stream at each time instant. The case of batched computations is pretty straightforward, with a major difference in the key scrambling routine. As a matter of fact, scrambling would require to serialize the processing of the element in the mini-batch, that is not a desirable feature. For this reason, we restrict to 1 the number of weak keys that can be potentially replaced for each mini-batch, selecting the one associated to the batch element that is returning the smallest similarity score with respect to K.

[3] There might be multiple weak keys. We select the oldest one.

Algorithm 3 Learning with a Continual Neural Unit when a batch of b samples is provided at each time instant by the stream. Notice that X is the mini-batch matrix $(b \times d)$, function α is intended to compute attention scores for each element of the batch, $\dagger$ and o are now arrays of length b. Scrambling involves up to 1 key for each mini-batch (function ScrambleOne).

```
Require: Stream $\mathcal{S}$; generic loss function loss(...), learning rates $\rho, \beta$; $K \leftarrow$ rand,
  $M \leftarrow$ rand, $\mu \leftarrow$ 0's, $\eta \leftarrow \tau^{\eta}$'s.
  while true do
      $X \leftarrow \text{next\_neuron\_inputs}(\mathcal{S})$    ▷ $X$ is a $b \times d$ matrix
      $A, S \leftarrow \alpha(X, K, \delta)$    ▷ $A$ and $S$ are $b \times m$ matrices (attentions and similarities)
      $\dagger_h \leftarrow \arg\max_{j \in \{1,\ldots,m\}}\{A_{hj}\},\ h = 1, \ldots, b$    ▷ Indices of the winning keys
      $K, M, \dagger, \mu \leftarrow \text{ScrambleOne}(X, K, M, S, \dagger, \mu, \eta)$ ▷ Possibly replace a weak key
      $K_{\dagger_h} \leftarrow K_{\dagger_h} + \beta \nabla \text{sim}(\psi(X_h), K_{\dagger_h}),\ h = 1, \ldots, b,$    ▷ Upd. winning keys, Eq. 6
      $A \leftarrow \alpha(X, K, \delta)$    ▷ Refresh attention (from scratch)
      $\mu_{\dagger_h} \leftarrow \mu_{\dagger_h} + 1, \quad \eta_{\dagger_h} \leftarrow 0, \quad h = 1, \ldots, b$ ▷ Incr. winning keys usages, reset ages
      $\eta = \eta + b$    ▷ Increase all ages
      $W \leftarrow AM$    ▷ Generate weights, Eq. 3, $W$ is a $b \times d$ matrix
      $o_h \leftarrow W_h X'_h, \quad h = 1, \ldots, b$    ▷ Compute output, Eq. 2
      $M_{\dagger} \leftarrow M_{\dagger} - \rho \nabla_{M_{\dagger}} \text{loss(o)}$    ▷ Update winning memory unit
  end while

  function ScrambleOne($X, K, M, S, \dagger, \mu, \eta$)
      $\mathcal{W} \leftarrow \{z \colon \mu_z < \tau^{\mu} \wedge \eta_z \geq \tau^{\eta}\}$    ▷ Set of weak keys (if any)
      $k \leftarrow \arg\min_{h \in \{1,\ldots,b\}}\{S_{h\dagger_h}\}$ ▷ Idx of sample with lowest sim. to its winning key
      if $S_{k\dagger_k} < \tau^{\alpha} \wedge \mathcal{W} \neq \emptyset$ then    ▷ If the current match is loose
          $j \leftarrow \arg\max_{z \in \mathcal{W}}\{\eta_z\}$    ▷ The weakest key is the oldest
          $K_j \leftarrow \psi(X_k)$    ▷ Replace weakest key with data sample
          $M_j \leftarrow M_{\dagger}$    ▷ Warm-start for replaced memory unit
          $\mu_j \leftarrow 0$    ▷ Reset usage counter
          $\dagger_k \leftarrow j$    ▷ Update winning key index
      end if
  return $K, M, \dagger, \mu$
  end function
```

C Metrics

Following the notation of Section 5, we are given a data stream $\mathcal{S}$ partitioned into non-overlapping time intervals, and we indicate with t_j the last time instant of the j-th interval I_j, with $t_N = T$. In the following description, we assume class indices to be ordered with respect to the time in which they become available, to keep the notation simple. We indicate with $p(x|I_i)$ the data distribution in the i-th interval, with θ_j is the model developed up to t_j (being it a CNU-based network or the network of another competitor), while D_i is an held out test set with data sampled from $p(x|I_i)$. Then, we indicate with

$$\mathrm{acc}_i^{\theta_j} = \mathrm{accuracy}(\mathcal{D}_i, \theta_j)$$

the accuracy on data sampled from $p(x|I_i)$ computed using the model parameters at t_j, i.e., θ_j. We collect the following matrix of accuracies during the learning procedure,

$$\left[\begin{array}{c|cccccc}
\text{Model/Test Data} & \mathcal{D}_1 & \mathcal{D}_2 & \dots & \mathcal{D}_j & \dots & \mathcal{D}_N \\
\hline
\theta_1 & \mathrm{acc}_1^{\theta_1} & \mathrm{acc}_2^{\theta_1} & \dots & \mathrm{acc}_h^{\theta_1} & \dots & \mathrm{acc}_N^{\theta_1} \\
\theta_2 & \mathrm{acc}_1^{\theta_2} & \mathrm{acc}_2^{\theta_2} & \dots & \mathrm{acc}_j^{\theta_2} & \dots & \mathrm{acc}_N^{\theta_2} \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\theta_j & \mathrm{acc}_1^{\theta_j} & \mathrm{acc}_2^{\theta_j} & \dots & \mathrm{acc}_j^{\theta_j} & \dots & \mathrm{acc}_N^{\theta_j} \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\theta_N & \mathrm{acc}_1^{\theta_N} & \mathrm{acc}_2^{\theta_N} & \dots & \mathrm{acc}_j^{\theta_N} & \dots & \mathrm{acc}_N^{\theta_N}
\end{array}\right]$$

that we indicate as continual confusion matrix (CCM, being CCM_j the matrix up to t_j), and we exploit it to compute the following measures. Notice that it is a square matrix.

- The average accuracy at t_z is defined as the average of the z-th row of the CCM, up to the z-th column (included),

$$\mathrm{avg\_accuracy}(\mathrm{CCM}_z) = \frac{1}{z} \sum_{i=1}^{z} \mathrm{acc}_i^{\theta_z},$$

  and we commonly measure the average accuracy (Acc of Section 5) at the end of training, $t_z = t_N$.
- The average forgetting at t_z can be defined as

$$\mathrm{avg\_forgetting}(\mathrm{CCM}_z) = \frac{1}{z-1} \sum_{i=1}^{z-1} \left(\mathrm{acc}_i^{\star} - \mathrm{acc}_i^{\theta_z} \right),$$

  where $\mathrm{acc}_i^{\star}$ is the best accuracy obtained on data $\mathcal{D}_i$ so far, i.e., $\max_{\theta_k \in \{\theta_1, \dots, \theta_{z-1}\}} \mathrm{acc}_i^{\theta_k}$ (maximum of the i-th column up to row $z-1$). We commonly measure the average forgetting (Forg of Section 5) at the end of training, $t_z = t_N$.

– Forward transfer measures how learning at checkpoint t_z (positively) influences predictions on data introduced in future intervals,

$$\text{forward}(\text{CCM}_z) = \frac{2}{z(z-1)} \sum_{i=1}^{z-1} \sum_{j=i+1}^{z} \text{acc}_i^{\theta_j},$$

i.e., it is the average of the upper-triangular portion of the CCM (excluding the diagonal). Similarly to the previous cases, we commonly measure it at $t_z = t_N$, yielding Fwd of Section 5.

D Computational Cost

A classic neuron model in a neural network requires u products to compute the output score, being u the size of the input, Eq. 1. We compare this cost in terms of the operations performed by CNUs, still using products as a reference. Computing the output of a CNU involves three main operations: (1) evaluating $\psi(x)$, whose cost is $C(\psi)$, that transforms the neuron input in a customizable manner, being $\tilde{u}$ the size of the ψ-output; (2) computing the attention scores by α, Eq. 4, with cost $m\tilde{u}$ plus the cost of the softmax$^\delta$ operation, that is δ; (3) blending memories, δu products, due to the sparsity of the attention scores; (4) computing the usual output function as in a classic neuron, u products. In total, we have $C(\psi) + m\tilde{u} + \delta + \delta u + u$. The cost of a layer of n classic neurons trivially becomes un, while the cost of a layer of CNUs that share the same K is

$$C(\psi) + m\tilde{u} + \delta + \delta un + un, \tag{7}$$

where only the last two terms depends on n, since the first three ones are about operations that are performed only once, being K shared. In order to reasonable relate the cost of classic and CM neurons, some basic considerations must be introduced. First of all, the cost $C(\psi)$ is expected to be way smaller than the cost of the whole layer. For example, when ψ is just limited to the L_2 normalization of x. Moreover, depending on the considered problem, there could be room for designing ψ such that $\tilde{u}$ is smaller than u. Of course, this does not always hold. It is reasonable to assume the term δ in Eq. 7 to be way smaller than the other ones (being it a strong sparsity index, always $< m$), thus we discard it. As a result, we can compute the ratio R between the cost of a CNU layer and the corresponding classic layer,

$$R_C = \frac{m\tilde{u} + (\delta+1)un}{un} = \frac{m\tilde{u}}{nu} + \delta + 1. \tag{8}$$

In case of multi-layer nets, with ℓ layers, we have

$$R_C = \frac{\sum_{i=1}^{\ell} m_{(i)}\tilde{u}_{(i)} + (\delta_{(i)}+1)d_{(i)}n_{(i)}}{\sum_{i=1}^{\ell} d_{(i)}n_{(i)}}, \tag{9}$$

being i the layer index. In terms of memory consumption, a layer of n CNUs stores matrix K and n matrices of memory units (M), that is a total of $m\tilde{u}+mun$ floating point numbers, while in a classic layer only the weight matrix is stored (un floating point values). The ratio R_M for ℓ layer is then,

$$R_M = \frac{\sum_{i=1}^{\ell} m_{(i)}\tilde{u}_{(i)} + m_{(i)}u_{(i)}n_{(i)}}{u_{(i)}n_{(i)}} \tag{10}$$

while the additional memory usage introduced by ℓ CNU layers is

$$U_M = m_{(i)}\tilde{u}_{(i)} + (m_{(i)} - 1)u_{(i)}n_{(i)}. \tag{11}$$

A candidate way to compare CNU-based net with models that replay data from memory buffers, is to use the exact same network architecture, using classic neurons in place of CNU. Then, we allow replay-based methods to sample $R_C - 1$ examples from the buffer at each time step. In fact, these buffer-based models make a prediction on a mini-batch of buffer data in addition to the currently streamed sample, according to the continual online learning setting experimented in this paper. Of course, when comparing with models that have more layers that the CNU-net, it is harder to keep a perfect balance in term of computational cost, so we allowed competitors to have a cost that is slightly larger than the one of the CNU-net, making the comparison more challenging. Moreover, we recall that the replay-based methods learn by exploiting the label-related information they store on the replay-buffer, while no-label-information is stored by the CNU-net (this the comparison becomes unfair when using very large replay buffers).

E Hyper-parameters

We evaluated multiple combinations of values for the main hyper-parameters of CNUs and competitors, that we summarize in the following, in addition to the already described parameter values of the main paper. In the case of CNU-based nets, in MODES and MOONS, we selected $m = 8$ memory units with $\delta = 2$, while we tested $\beta \in \{10^{-4}, 10^{-3}, 10^{-2}, 1\}$, $\tau_\mu \in \{50, 200\}$, $\tau_\eta \in \{50, 200\}$, $\tau_\alpha \in \{0.85, 0.95\}$, $\gamma \in \{1, 5, 25\}$. In NS-IMAGENET we considered $m \in \{10, 25, 50, 100\}$, $\delta \in \{2, 5\}$, $\beta \in \{10^{-3}, 10^{-2}\}$, $\tau_\mu \in \{50, 500, 5000\}$, $\tau_\eta \in \{50, 500, 5000\}$, $\tau_\alpha \in \{0.7, 0.85, 0.95\}$, $\gamma \in \{1, 5, 25\}$. The hidden layer size has been evaluated in $h \in \{5, 25\}$ for 2D data, while in $h \in \{50, 100\}$ for NS-IMAGENET. In all the models, we considered a learning rate $\rho \in \{10^{-4}, 10^{-3}, 10^{-2}, 1\}$, and trained with fixed-step-size gradient descent. We also evaluated the case of Adam, which yielded lower results on average. Indeed, adopting optimizers with memory such as Adam may be tricky: at every step, the model might select a different set of weights to be updated, making the statistics of the optimizer invalid. We leave the investigation about the effect of such optimizers for future work, restricting our analysis to memoryless optimizers, which do not suffer from this issue. We also considered a weight decay factor DECAY for the optimizers $\in \{10^{-4}, 10^{-3}, 0\}$. We trained GDumb for 10 epochs on the buffer data. Other minor internal

Table 2: Optimal parameters. The best selected hyperparameters for the proposed CNU model, drawn from the grids described in the text, for the datasets. See the code for further details.

	MODES			MOONS			NS-IMAGENET
Parameters	CI	CDI	CDID	CI	CDI	CDID	-
δ	2	2	2	2	2	2	5
β	10^{-2}	10^{-2}	10^{-2}	10^{-2}	10^{-2}	10^{-1}	10^{-3}
ρ	10^{-2}	10^{-1}	10^{-2}	10^{-1}	10^{-2}	10^{-2}	10^{-4}
γ	25	25	25	25	25	5	1
m	8	8	8	8	8	8	100
τ_α	0.95	0.95	0.95	0.95	0.95	0.85	0.7
τ_η	50	50	50	50	200	200	5000
τ_μ	50	50	50	50	50	50	500
DECAY	0.	0.	0.	0.	0.	10^{-3}	10^{-3}

parameters of the competitors were set to the values suggested in the respective papers. The results reported in the main paper are averaged over three runs with different seeds in $\{1234, 123456, 12346578\}$. For all the experiments in this work we used PyTorch, running on a Linux machine–NVIDIA GeForce RTX 3090 GPU (24 GB).

E.1 Optimal Hyper-parameters

We report in Table 2 the best selected hyperparameters for the CNU model in all the considered datasets and settings described in the main paper.

F Networks of CNUs

We report in Figure 5 a more detailed view of the process of input projection, memory blending and WTA update occurring in each CNU. We now briefly discuss the specific case of convolutional layers and on networks with multiple stacked CNU-based layers.

In a convolutional layer, each spatial coordinate is associated with a neuron[4] whose input is differently shifted with respect to every other neuron of the layer. At a first glance, this makes less obvious how to share keys (if need) among the neurons of the layer. However, neurons are still expected to share the same weights/filter, thus the key-attention scores should be the same for all of them, in order to coherently blend memories and generate the same filter. In this case, it is convenient to go back to the standard definition of convolution operation, where the whole input map of the layer is one of its operands, thus $\psi(x)$ is a

[4] We consider a single filter/output-feature-map in this description, for simplicity. For the same reason, and without any loss of generality, we describe infinitely supported filters.

projection of the whole input of the layer, and it is the same for all the neurons. As a result, the attention scores are computed only once per layer, independently of the resolution of the input map.

Layers of CNUs can be stacked into multi-layer networks, as usual. However, while the input of the first layer is not affected by CNU dynamics, the input of any other layer comes from the CNUs of the layer below. The proposed WTA key update scheme is not gradient-based, so that we also avoid gradients to propagate through the key-matching process, i.e., we consider $\hat{w}(x, K, M)$ of Eq. 2 to not depend on x for gradient computation purposes. As a result, gradients flow from the output layer down to the output of the layer below, as usual, and not through $\hat{w}$. Another intuition that we followed is that CNUs belonging to the lower layers in a deep convolutional network should have a smaller number of memory units than CNUs on the top layers, since lower-level features are likely to be more shared across inputs than the higher-level ones, where the semantics emerge (e.g., edge-like filters in the lowest levels vs. objects/object-parts related filters in the highest ones). Of course, in non-convolutional nets this is harder to say in advance, but we still follow the same intuition motivated by the need of reducing the variability in the outputs of the lower layers, to favor stability in the learning process of the upper layers. Investigating the interaction among layer requires specific studies that go beyond the scope of this paper. Indeed, we focused on networks in which the input of the CNUs was not affected by the learning dynamics, both when using backbones with several layers or when directly classifying data in their original representations. Each CNU virtually and softly partitions its input space, dynamically learning how to do it and how to behave in each partition. In multiple layers of CNUs, the progressive-compositional development of such partitions can lead to instabilities or difficulties in quickly learning in an online manner. Preliminary investigations on fully

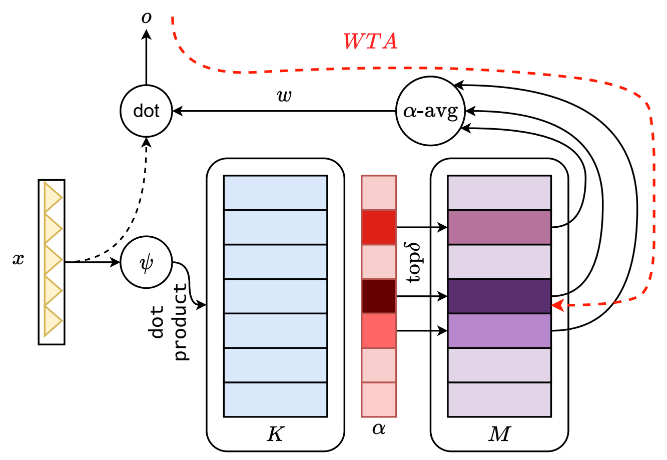

Fig. 5: Larger instance of the contents of Fig. 1, for better readability.

connected networks, obtained by stacking multiple layers of CNUs, did not result in significant gain with respect to shallow architectures. Interestingly, when multiple layers are stacked, on average we observe a slight decrease in accuracy, both for our method and for the competitors. This finding confirms that larger models tend to perform better than deeper models for continual learning, due to exploding gradients leading to forgetting in the earliest layers, as discussed in [27,28].

G Additional Experimental Results

In Fig. 6 we report the upper-bound of the results presented in the main paper, obtained by selecting the best-performing model on the test data. Of course, these results are only intended to be a reference to understand the maximum performance each model could achieve, and not a way to make comparisons across different approaches. We remark that this is different from what we did in the main paper (Fig. 3), where we cross-validated the hyper-parameter values on the validation part of the stream and evaluated performance on the out-of-sample test sets.

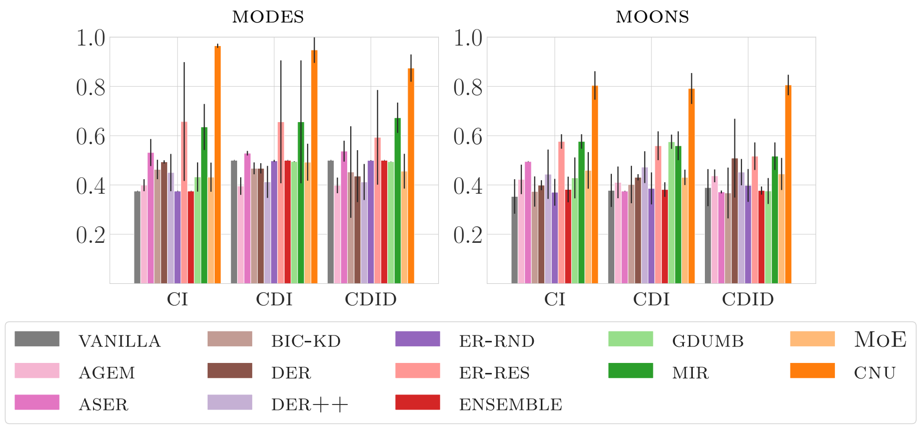

Fig. 6: MODES and MOONS data, **best** test accuracy (reference only) and std in the three setting we analyzed (CI, CDI, CDID).

Comparing Fig. 6 with Fig. 3, we notice that the CNU-based net is actually able to reach similar performance, thus being able to make the most out of the validation procedure. Since the validation set is limited to the first part of the streamed data, this results is very promising in terms of what can be achieved when working on longer streams with limited time-span dedicated to the validation of the parameters.