# Neurocoder: General-Purpose Computation Using Stored Neural Programs

**Hung Le** [1] **Svetha Venkatesh** [1]

## Abstract

Artificial Neural Networks are functionally equivalent to special-purpose computers. Their inter-neuronal connection weights represent the learnt Neural Program that instructs the networks on how to compute the data. However, without storing Neural Programs, they are restricted to only one, overwriting learnt programs when trained on new data. Here we design Neurocoder, a new class of general-purpose neural networks in which the neural network "codes" itself in a data-responsive way by composing relevant programs from a set of shareable, modular programs stored in external memory. This time, a Neural Program is efficiently treated as data in memory. Integrating Neurocoder into current neural architectures, we demonstrate new capacity to learn modular programs, reuse simple programs to build complex ones, handle pattern shifts and remember old programs as new ones are learnt, and show substantial performance improvement in solving object recognition, playing video games and continual learning tasks.

## 1. Introduction

From its inception in 1943 until recently, the fundamental architectures of Artificial Neural Networks remained largely unchanged - a program is executed by passing data through a network of artificial neurons whose inter-neuronal connection weights are learnt through training with data. These inter-neuronal connection weights, or Neural Programs, correspond to a program in modern computers (Schmidhuber, 1990). Memory Augmented Neural Networks (MANN) are an innovative solution allowing networks to access external memory for manipulating data (layer's output vectors) (Graves et al., 2014; 2016). But they were still unable to store Neural Programs (weight matrices) in such external memory, and this severely limits machine learning. Storing inter-neuronal connection weights only in their network does not permit modular separation of Neural programs and is analogous to a computer with one fixed program. Recent works introduce *conditional computation* via adjusting or activating parts of a network in an input-dependent manner (von der Malsburg, 1981; Schmidhuber, 1992; Bengio et al., 2013; Ha et al., 2017; Perez et al., 2018), but networks remain monolithic. Current networks forget when retrained, old inter-neuronal connection weights are merged with new ones or erased.

The brain is modular, not a monolithic system, and is divided into functional modules (Hubel, 1988). If the neural program for each module is kept in separate networks, networks proliferate. Modular neural networks, another form of conditional computation, combine the output of multiple expert networks, but as the experts grow, the networks grow drastically (Jacobs et al., 1991; Happel & Murre, 1994; Shazeer et al., 2017; Rosenbaum et al., 2018). This requires huge storage and introduces redundancy as these experts do not share common basic programs.

*A pathway out of this bind is to keep such basic programs in memory and combine them as required.* This brings neural networks towards modern general-purpose computers that use the stored-program principle (Turing, 1936; Von Neumann, 1993) to efficiently access reusable programs in external memory. Here we show how Neurocoder, a new neural framework, introduces a new class of general-purpose conditional computation machines in which a neural network can be "coded" in an input-dependent manner. Efficient decomposition of Neural Programs creates shareable modular components that can reconstruct the whole program space. These components change their "shapes" based on training and are stored in an external Program Memory. Then, in a data-responsive way, a Program Controller retrieves relevant components to build the Neural Program.

Using adaptive modular components vastly increases the learning capacity of the neural network by allowing re-utilisation of parameters, effectively curbing network growth as programs increase. More importantly, unlike pre-defined sub-networks or modules (Jacobs et al., 1991; Andreas et al., 2016) that combine at activation level, the construction of our modular components is dynamic and

[1]Applied AI Institute, Deakin University, Geelong, Australia. Correspondence to: Hung Le <thai.le@deakin.edu.au>.

*Proceedings of the 39th International Conference on Machine Learning*, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

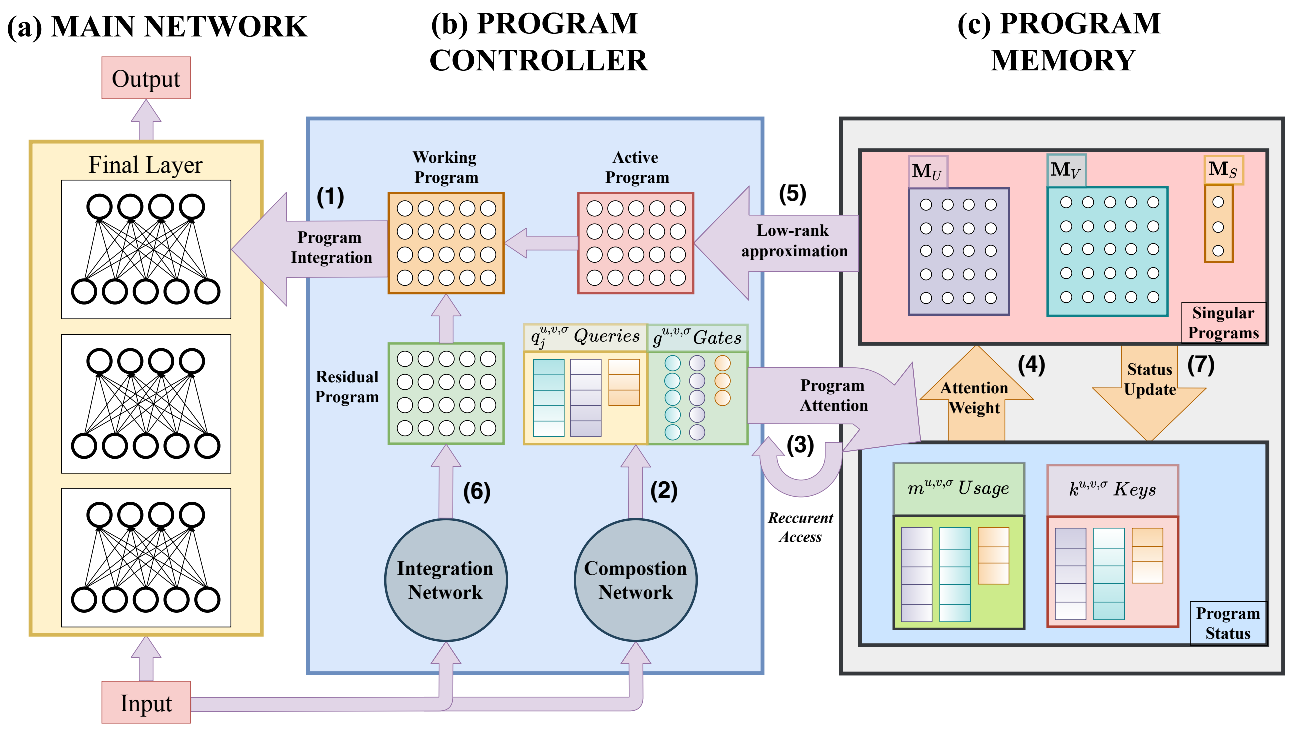

*Figure 1.* Neurocoder **(a)** The *Main Network* uses a *working program* to compute the output for the input. **(b)** The Program Controller's *composition network* controls access to the Program Memory. **(c)** The Program Memory stores the representations (singular programs) required to reconstruct the active program.

performed on the weight space. The Neural Program construction is learnt through training via traditional backpropagation as the architecture is end-to-end differentiable.

To demonstrate the flexibility of Neurocoder framework, we consider different learning paradigms: instance-based, sequential, multi-task and continual learning. We do not focus on breaking performance records by augmenting state-of-the-art models with Neurocoder. Instead, our inquiry is on re-coding feed-forward layers with the Neurocoder's programs and testing on varied data types to demonstrate its intrinsic properties, showing consistent improvement over standard backbones and methods. Our contributions are: (i) we provide a novel and efficient way to store programs/weights of the neural networks in an external memory, (ii) thanks to our general design of program memory, we can equip current neural networks with a new capability of conditional and modular computing, and (iii) we conduct experiments on various tasks, confirming the general-purpose property of our model.

## 2. Methods

### 2.1. System Overview

A Neurocoder is a neural network (Main Network) coupled to an external Program Memory through a Program Controller. The *working program* of the Main Network processes the input data to produce the output. This working program is "coded" by the Program Controller by creating an input-dependent *active program* from the Program Memory (Fig. 1). The following gives a high-level description of the Neurocoder framework and then the details.

**Neurocoder stores Singular Value Decomposition (SVD) of Neural Programs** The Neural Program needs to be stored efficiently in Program Memory. This is challenging as there may be millions of inter-neuronal connection weights, thus storing a Neural Program as a matrix weight directly (Le et al., 2020) is grossly inefficient. Instead, the Neurocoder forms the basis of a subspace spanned by Neural Programs and stores the singular values and vectors of this subspace in memory slots of the Program Memory (hereafter referred to as *singular programs*). Based on the input, relevant singular programs are retrieved, a new program is reconstructed and then loaded in the Main Network to process the input. This representational choice significantly reduces the number of stored elements and allows each singular program to effectively represent a unitary function of the active program.

For a neural network layer with $d_{in}$ and $d_{out}$ input and output units, respectively, the *active program* matrix $\mathbf{P} \in$

$\mathbb{R}^{d_{in} \times d_{out}}$ can be decomposed as

$$\mathbf{P} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathbf{T}} = \sum_{n}^{r_m} \sigma_n u_n v_n^{\top} \qquad (1)$$

where $\mathbf{U}$ and $\mathbf{V}$ are matrices of the left and right singular vectors, and $\mathbf{S}$ the matrix of singular values. $r_m$ is the total number of components we want to retrieve. $\{\sigma_n\}_{n=1}^{r_m}$ is the attended singular values, $\{u_n\}_{n=1}^{r_m}$ and $\{v_n\}_{n=1}^{r_m}$ the attended singular vectors of $\mathbf{S}$, $\mathbf{U}$, and $\mathbf{V}$, respectively. The Program Memory is then crafted as three *singular program memories* $\{\mathbf{M}_U, \mathbf{M}_V, \mathbf{M}_S\}$–each of their memory slot stores a singular component or singular program. *The process "codes" the active program using singular programs from* the program memories. The coding is conditioned on input $x_t$, yet here we drop index $t$ for notation simplification and leave the details until §2.2. The active program $\mathbf{P}$ will be used to compose the working program $W$ that is loaded to a layer of neural networks (§2.3).

The Program Memory also maintains the status for each singular program in terms of access and usage. To access a singular program, *program keys* ($k$) are used. These keys are low-dimensional vectors that represent the singular program function and computed by a neural network that effectively compresses the singular program. The *program usage* ($m$) measures memory utilisation, recording how much a memory slot is used in constructing a program. Fig. 1 depicts an overview of Neurocoder.

In this figure, the Main Network's final layer[1] is adaptively loaded with the *working program* (***1***). Other layers use traditional Neural Programs as connection weights (fixed-after-training). The Program Controller's *composition network* emits queries and interpolating gate control signals in response to the input (***2***). It then performs recurrent multi-head program attention to the Program Status ***(3)***, triggering attention weights to the Singular Programs ***(4)***. The attended Singular Programs form an *active program* using low-rank approximation (SVD) ***(5)***. *Residual program* produced by the Program Controller's *integration network* (**6**) plus the active program derives the *working program*. The Program Memory stores singular programs for constructing the active program. Access to the memory is controlled through the Program Status including keys ($k$), and slot usage ($m$) that are updated during the training and computation ***(7)***.

**Recurrent multi-head program attention mechanisms for program storage and retrieval** Neural networks use the concept of *differentiable attention* to access memory (Graves et al., 2014; Bahdanau et al., 2015). This defines a weighting distribution over the memory slots essentially weighting the degree to which each slot participates in a read or write operation. This is unlike conventional computers that use a unique address to access a single memory slot.

[1]For visualisation purpose only. Any or all layer can be used.

Here we use two kinds of attention. First is *content-based attention* (Graves et al., 2014; 2016) to ensure that the singular program is selected based on its functionality and the data input. This is achieved by producing a query vector based on the input and comparing it to the program keys ($k$) using cosine similarity. Higher cosine similarity scores indicate higher attention weights to the singular programs associated with those program keys. Second, to encourage better memory utilisation, higher attention weights are assigned to slots with lower program usage ($m$) through *usage-based attention* (Graves et al., 2016; Santoro et al., 2016). The attention weights from the two schemas are then combined using interpolating gates to compose the final attention weights to the Program Memory.

We adapt multi-head attention (Graves et al., 2014; Vaswani et al., 2017) that applies multiple attentions in parallel to retrieve $H$ singular components. Besides, we introduce a recurrent attention mechanism, in which multi-head access is performed recurrently in $J$ steps. The $j$-th set of $H$ retrieved components is conditioned on the previous ones. This recurrent, multi-head attention allows the composition network to incrementally search for optimal components for building relevant active programs.

**Neurocoder learns to "code" a relevant working program via training** The structure of the Program Memory and the role of the Program Controller facilitates automatic construction of working programs via training. The Program Controller controls memory access through its *composition network* that creates the *attention weight* defining how to weight the singular programs in the memories. A weighted summation of the singular programs results in the attended singular program. Applying the recurrent multi-head attention described earlier, multiple attended singular programs are retrieved to construct an active program (Eq. 1). Then the Program Controller generates a *residual program* using its *integration network*, adding to the active program to produce the working program of the Main Network. This addition enables creation of flexible higher-rank working programs, which compensates for SVD low-rank coding process. The Program Controller is illustrated in Fig. 1 (b).

The singular programs represent unitary functions necessary for any computation whilst the composition and integration networks select the relevant programs for the considering task. They are jointly trained end-to-end by the task loss. To ensure unitary functionality, we enforce orthogonality of stored singular vectors by minimising $\mathcal{L}_o = \left\|\mathbf{M}_U\mathbf{M}_U^{\top} - \mathbf{I} + \mathbf{M}_V\mathbf{M}_V^{\top} - \mathbf{I}\right\|_2$. Hence, the total loss becomes

$$\mathcal{L} = \mathcal{L}_{task} + a\mathcal{L}_o \qquad (2)$$

where $\mathcal{L}_{task}$ represents the supervised task loss and $\mathcal{L}_o$ represents the orthogonal loss weighted by a hyper-parameter

$a$ to control the unitary constraint.

## 2.2. Attention Mechanisms for Program Memory

Here we describe program attention mechanisms used in this paper. Given $w_{in}^{u}$, $w_{in}^{v}$, $w_{in}^{\sigma}$ (jointly denoted as $w_{in}^{u,v,\sigma}$)–the attention weight to the $i$-th slot of the singular program memories $\mathbf{M}_U$, $\mathbf{M}_V$ and $\mathbf{M}_S$, we retrieve the $n$-th singular vectors as follows,

$$u_n = \sum_{i=1}^{P_u} w_{in}^{u} \mathbf{M}_U (i) \qquad v_n = \sum_{i=1}^{P_v} w_{in}^{v} \mathbf{M}_V (i) \tag{3}$$

For the singular values, we need to enforce $\sigma_1 > \sigma_2 > ... > \sigma_{r_m} > 0$, thus we retrieve using

$$\sigma_n = \begin{cases} \text{softplus}\left(\sum_{i=1}^{P_s} w_{in}^{\sigma} \mathbf{M}_S (i)\right) & n = r_m \\ \sigma_{n+1} + \text{softplus}\left(\sum_{i=1}^{P_s} w_{in}^{\sigma} \mathbf{M}_S (i)\right) & n < r_m \end{cases} \tag{4}$$

Here, $P_u$, $P_v$ and $P_s$ are the number of memory slots of $\mathbf{M}_U$, $\mathbf{M}_V$ and $\mathbf{M}_S$, respectively. In this paper, we set $P = P_u = P_v = P_s$ as the number of memory slots of the Program Memory. Hence, $\mathbf{M}_U \in \mathbb{R}^{P \times d_{in}}$, $\mathbf{M}_V \in \mathbb{R}^{P \times d_{out}}$ and $\mathbf{M}_S \in \mathbb{R}^{P \times 1}$. We note that $u, v, \sigma, w$ notations are specified for some data input $x_t$ and the index $n$ later maps to an attention head $h$, and an attention step $j$, hence the full notation should be $w_{tijh}^{u,v,\sigma}$. To simplify notations, we will drop $u, v, \sigma$ from now and describe the computation of a representative $w_{tijh}$ for any of the three program memories in the following.

**Recurrent attention to the Program Memory via the composition network** To perform program attention, the Program Controller employs a composition network (denoted as $f_\theta$), which takes the current input $x_t$ and produce *program composition control signals* ($\boldsymbol{\xi}_t^p$). If $f_\theta$ performs all attentions concurrently via multi-head attention as in Graves et al. (2014); Vaswani et al. (2017), it may lead to program collapse (Le et al., 2020). Rolling out query heads rather than generating them in parallel allows us to ensure that each of the query heads is distinct from the rest and thereby avoiding program collapse. To this end, we implement $f_\theta$ as a recurrent neural network (LSTM (Hochreiter & Schmidhuber, 1997)) and let it access the program memory $J$ times, resulting in $\boldsymbol{\xi}_t^p = \left\{\boldsymbol{\xi}_{tj}^p\right\}_{j=1}^{J}$. At access step $j$, the recurrent network updates its hidden states and generates $\boldsymbol{\xi}_{tj}^p$ using recurrent dynamics as $\boldsymbol{\xi}_{tj}^p, h_j = f_\theta (x_t, h_{j-1})$ where $h_0$ is initialized as zeros and $\boldsymbol{\xi}_{tj}^p$ is the program composition control signal at step $j$ that depends on both on the input data $x_t$ and the the previous state $h_{j-1}$. Particularly, the control signal contains the queries and the interpolation gates for each head to compute the program attention weight: $\boldsymbol{\xi}_{tj}^p = \{q_{tjh}, g_{tijh}\}_{h=1}^{H}$. Here, at each attention step, we perform multi-head attention with $H$ as the number of attention heads and thus, each $\boldsymbol{\xi}_{tj}^p$ consists of $H$ pairs of queries and gates. Hence, the total number of retrieved components $r_m = J \times H$ and the index $n = j \times H + h$.

**Attending to programs by "name"** Inspired by the content-based attention mechanism for data memory (Graves et al., 2014), we use the query to look for the singular programs. In computer programming, to find the appropriate program for some computation, we often refer to the program description or at least the name of the program. Here, we create the "name" for our neural programs by compressing the program content to a low-dimensional key vector. As such, we employ a neural network ($f_\varphi$) to compute the program memory keys as $k_i = f_\varphi (\mathbf{M} (i))$ where $k_i \in \mathbb{R}^K$ and $i$ is the row index of the program memory (see Appendix A for details). As the singular programs evolve, their keys get updated. In this paper, we update the program keys after each learning iteration during training.

Finally the content-based program memory attention $c_{tijh}$ is computed using cosine distance between the program keys $k_i$ and the queries $q_{tjh}$ as

$$c_{tijh} = \text{softmax}^{(i)} \left(\frac{q_{tjh} \cdot k_i}{||q_{tjh}|| \cdot ||k_i||}\right) \tag{5}$$

**Making every program count** Similarly to Graves et al. (2016); Santoro et al. (2016), in addition to the content-based attention, we employ a least-used reading strategy to encourage the Program Controller to assign different singular programs to different components. In particular, we calculate the memory usage for each program slot across attentions as $m_{tijh} = \max\limits_{\tilde{j} \leq j} \left(w_{ti\tilde{j}h}\right)$. Since we want to consider only $l_I$ amongst $P$ memory slots that have smallest usages, let $\hat{m}_{tjh}^{l_I}$ denote the value of the $l_I$-th smallest usage, then the least-used attention is computed as

$$l_{tijh} = \begin{cases} \max\limits_{i} (m_{tijh}) - m_{tijh} & ; m_{tijh} \leq \hat{m}_{tjh}^{l_I} \\ 0 & ; m_{tijh} > \hat{m}_{tjh}^{l_I} \end{cases} \tag{6}$$

The final program memory attention is computed as $w_{tijh} = \text{sigmoid} (g_{tijh}) c_{tijh} + (1 - \text{sigmoid} (g_{tijh})) l_{tijh}$. Since the usage record are computed along the memory accesses, the multi-step Neurocoder utilises this attention mechanism better than the single-step Neurocoder, creating different attention styles (see §3.2). The composition the active program $\mathbf{P}_t$ is illustrated in Appendix's Fig. 4.

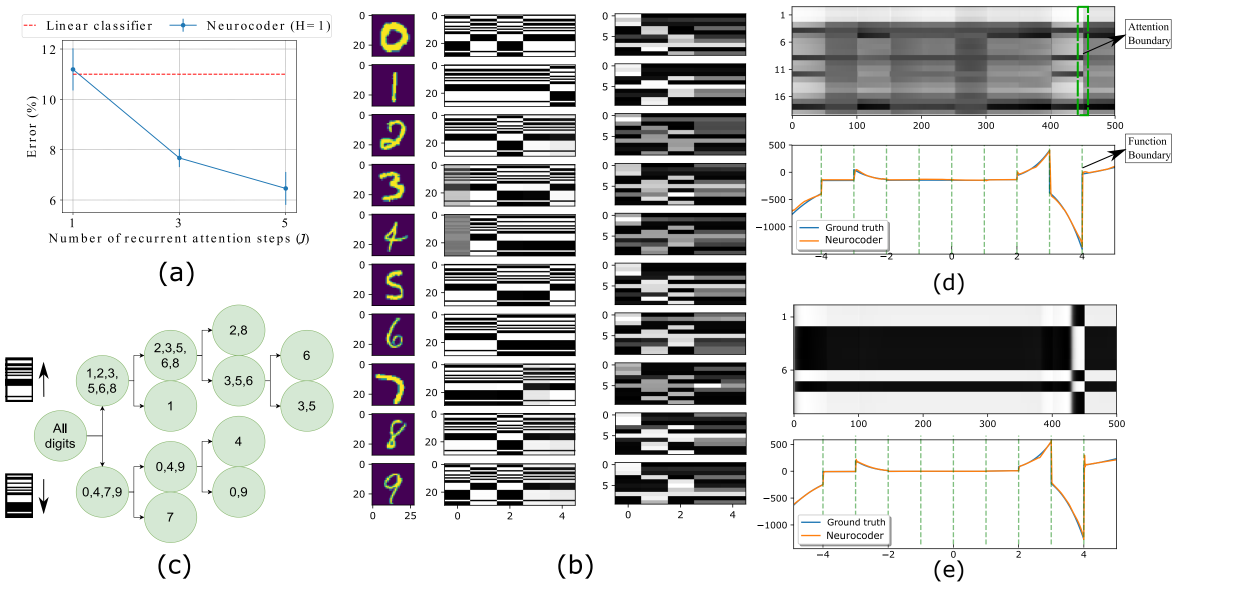

*Figure 2.* **(a)** MNIST test set classification error *vs* the number of steps ($J$) in Neurocoder (blue), compared with a linear classifier (red). **(b)** *1st column*: Digit images; *Middle column*: Single-step attention weights for 30 slots in $\mathbf{M}_U$ (vertical axis) for first 5 singular vectors (horizontal axis) for each digit; *Last column*: Multi-step attention weights for 10 slots in $\mathbf{M}_U$ (vertical axis) for first 5 singular vectors (horizontal axis). Multi-step attention is able to produce far more diverse patterns with fewer slots - 10 slots compared to single-step 30 slots. **(c)** Two attention patterns of single-step Neurocoder. The binary decision tree derived from single-step Neurocoder's attention patterns. The two patterns across components represent the decisions going up and down across the binary tree. Visualisation for **(d)** multi-step ($J = 5$, 20 memory slots) and **(e)** single-step ($J = 1$, 10 memory slots) cases showing while processing a sequence of the polynomial auto-regression task.

## 2.3. Program Integration via The Integration Network

Since the working program $\mathbf{P}_t$ only contains top $r_m$ principal components, it is low-rank and may be not flexible enough for sophisticated computation. We propose to enhance $\mathbf{P}_t$ with a residual program $\mathbf{R}$– a traditional connection weight trained as the integration network's parameters, which is constant after training w.r.t $t$. The residual program represents the sum of the remaining less important components. To this end, we suppress $\mathbf{R}$ with a multiplier that is smaller than $\sigma_{tr_m}$– the smallest singular value of the main components - resulting in the integration formula

$$W_t = \mathbf{P}_t + w_t^r \sigma_{tr_m} \mathbf{R} \tag{7}$$

where $w_t^r = \text{sigmoid}\left(f_\phi\left(x_t\right)\right)$ is an adaptive gating value that controls the contribution of the residual program. $f_\phi$ is the integration network in the Program Controller and hence, in our implementation, the integration control signal sent by the Program Controller is $\boldsymbol{\lambda}_t^p = \left\{w_t^r, \sigma_{tr_m}\right\}$. We note that in our experiments, the program integration can be disabled ($W_t$ is directly set to $\mathbf{P}_t$) to prove the contribution of $\mathbf{P}_t$ or reduce the number of parameters. The working program $W_t$ is then used by the Main Network to execute the input data $x_t$ (see (Fig. 1 (a))). For example, with linear classifier Main Network, the execution is $y_t = x_t W_t$. If multiple layers need to change program, we generate multiple $W_t$ accordingly. Appendix's Table 3 summarises Neurocoder's important parameter notations.

## 3. Experimental Results

We compare the performance of diverse Main Networks (MN) with and without Neurocoder. We also augment the Main Networks with other recent conditional computing methods, either modular (sparse Mixture of Experts, Neural Stored-program Memory) or monolithic (HyperNets, FiLM) to form stronger baselines. We always apply Neurocoder to all layers of multi-layer perceptrons (MLP) or just the final feed-forward layer of deep CNN networks (LeNet, DenseNet, ResNet), RNNs (GRU, LSTM), MANN (NTM). Other competitors such as MOE, NSM, HyperNet and FiLM are applied to the Main Networks in the same manner.

### 3.1. Instance-based Learning - Object Recognition

We tested Neurocoder on instance-based learning through classical image classification tasks using MNIST (LeCun et al., 1998) and CIFAR (Krizhevsky et al., 2009) datasets. The first experiment interpreted Neurocoder's behaviour in classifying digits into 10 classes ($0 - 9$) using linear classifier Main Network. With less parameters (around 7.3K vs 7.8K), Neurocoder using the recurrent attention surpasses

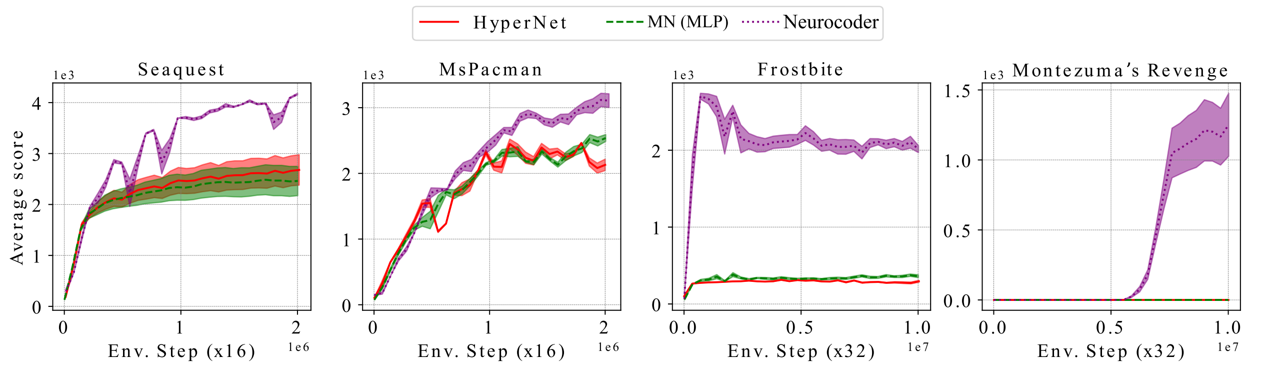

*Figure 3.* Learning curves (mean and std. over 5 runs) on representative Atari 2600 games. All baselines are applied to the actor/critic networks in the A3C agent.

linear classifier by up to $5\%$ (Fig. 2 (a)). This simple example showcases how a low-rank program is superior to the traditional weight.

To differentiate the input, Neurocoder attends to different components of the active program to guide the decision-making process. Fig. 2 (b) shows single-step and multi-step attention to the first $5$ singular vectors for each digit across memory slots. Multi-step attention produces richer patterns compared to single-step Neurocoder that manages only $2$ attention weight patterns.

Fig. 2 (c) illustrates how Neurocoder performs modular learning by showing the attention assignment for top $5$ singular vectors as a binary decision tree. Digits under the same parental node share similar attention paths, and thereby similar active programs. Some digits look unique (e.g. $7$) resulting in active programs composed of unique attention paths, discriminating themselves early in the decision tree. Some digits (e.g. $0$ and $9$) share the same attention pattern for the first $5$ components and are thus unclassifiable. They can only be distinguished by considering more singular vectors.

We integrated Neurocoder with deep networks - $5$-*layer LeNet and* $100$-*layer DenseNet* - and tested on CIFAR datasets. Neurocoder significantly outperformed the Main Networks by $1-5\%$ accuracy. Compared with sparse Mixture of Experts (MOE (Shazeer et al., 2017)) and Neural Stored-program Memory (NSM (Le et al., 2020)), Neurocoder required a tenth of the number of parameters and performed better by up to $8-10\%$ (Appendix's Table 4).

### 3.2. Sequential Learning - Sequence Adaption and Game Playing in Reinforcement Learning

Recurrent neural networks (RNN) can learn from sequential data by updating the hidden states of the networks. However, this does not suffice when local patterns shift, as is often the case. We now demonstrate that Neurocoder helps RNNs overcome this limitation by composing diverse programs to handle sequence changes.

**Synthetic polynomial auto-regression** We created a simple auto-regression task in which data points are sampled from polynomial function chunks that change over time. The Main Network is a strong *RNN–Gated Recurrent Unit* (*GRU* (Cho et al., 2014)). We found that GRU integrated with a single-step or multi-step Neurocoder converged much faster than all other baselines. HyperNet (Ha et al., 2017) and FiLM (Perez et al., 2018)) adapt by re-scaling weights or activation of the GRU, which were shown inferior to our modular approach (Appendix's Fig. 5).

Figs. 2(d, e) visualise the first singular vector attention weights in $\mathbf{M}_U$ that form the first component of the active program are shown over sequence timesteps (*upper*) with Neurocoder's $y_t$ prediction (orange) and ground truth (blue) (*lower*). The vertical dash green lines separate polynomial chunks. Each chuck represents a local pattern, and thus ideally requires a specific active program to compute the input $x_t$. We find that the multi-step attention Neurocoder changes its attention following polynomial changes - it attends to the same singular program when processing data from the same polynomial and alters attention for data from a different polynomial (Fig. 2(d)). In contrast, the single-step Neurocoder only changes its attention when there is a remarkable change in $y$-coordinate values (Fig. 2(e)).

Single-step Neurocoder did not discover the underlying structure of the data, and thus underperformed the multi-step Neurocoder. We hypothesise that when recurrence is employed, usage-based attention takes effect, stipulating better memory utilisation and diverse attentions over timesteps. We ran multi-step Neurocoder without usage-based attention. The results were worse than multi-step Neurocoder, which confirms our hypothesis (Appendix's Fig. 5).

**Atari game reinforcement learning** We used reinforcement learning as a further testbed to show the ability to adapt

| Method | MN (MLP (Hsu et al., 2018)) | MN (MLP ours) | NSM | Neurocoder |
|---|---|---|---|---|
| Adam | 55.16±1.38 | 53.55±1.27 | 54.85±2.81 | **58.46±0.46** |
| Adagrad | 58.08±1.06 | 57.83±2.74 | 58.42±1.87 | **62.28±4.03** |
| L2 | 66.00±3.73 | 64.37±2.40 | 62.83±7.21 | **69.89±1.72** |
| SI | 64.76±3.09 | 64.41±3.36 | 64.36±2.99 | **67.96±3.22** |
| EWC | 58.85±2.59 | 58.41±2.37 | 58.12±3.24 | **65.66±1.25** |
| O-EWC | 57.33±1.44 | 57.78±1.84 | 58.55±3.40 | **73.97±1.50** |

*Table 1.* Incremental domain continual learning with Split MNIST. Final test accuracy (mean and std.) over 10 runs.

to environmental changes. We performed experiments on several Atari 2600 games (Bellemare et al., 2013) wherein the agent was implemented as the *Asynchronous Advantage Actor-Critic* (*A3C* (Mnih et al., 2016)). In the Atari platform, agents are allowed to observe the screen snapshot of the games and act to earn the highest score. We augmented the A3C by employing Neurocoder's working programs for feed-forward layers of the actor and critic networks, aiming to decompose the policy and value function into singular programs that were selected depending on the game state.

*Frostbite and Montezuma's Revenge*. These games are known to be challenging for A3C and other algorithms (Mnih et al., 2016). We trained A3C and HyperNet-based A3C for over 300 million steps, yet these models did not show any sign of learning, performing equivalently to random agents. For such complicated environments with sparse rewards, both the monolithic neural networks and the HyperNet's unstored fast-weights fail to learn (almost zero scores). In contrast, Neurocoder enabled A3C to achieve from $1,500$ to $3,000$ scores on these environments (Fig. 3), confirming the importance of decomposing a complex solution to smaller, simple stored programs.

### 3.3. Multi-task learning - Solving Mutliple Algorithms

Here we explore the modular learning capability of Neurocoder in multi-task setting. Inspired by algorithmic sequencing tasks (Le et al., 2020), we created a challenging sequential multi-task benchmark wherein the input sequence is a series of sub-sequences from 4 algorithms: Copy, Repeat Copy, Associative Recall and Priority Sort (Graves et al., 2014). Each sub-sequence, following a task identification vector, is the task input. In each input sequence, $n$ tasks were sampled from the set of 4 algorithms with replacement and the output sequences were created correspondingly.

We trained a *MANN–Neural Turing Machine (NTM* (Graves et al., 2014)*)* Main Network with FiLM, HyperNet and our Neurocoder augmentation on sequences of $n = 4$ tasks, and tested with sequences of $n = 4$ and $n = 8$ tasks. Appendix's Fig. 6 demonstrates that Neurocoder was performant in both test settings, not only achieving lowest error on $n = 4$, but also being the only one generalised well to $n = 8$ scenario, which was unseen during training.

### 3.4. Continual learning - Learning Tasks Sequentially

In continual learning, standard neural networks often suffer from "catastrophic forgetting" in which they cannot retain knowledge acquired from old tasks upon learning new ones (French, 1999). Our Neurocoder offers natural mitigation of such catastrophic forgetting in neural networks by attending to different singular programs whilst learning different tasks.

In this case, in addition to the Main Network, we examine several continual learning algorithms. These algorithms, including Elastic Weight Consolidation (EWC (Zenke et al., 2017)) and Synaptic Intelligence (SI (Zenke et al., 2017)), work by regularising the loss function and thus can be easily combined with Neurocoder by modifying the loss $\mathcal{L}_{task}$. To avoid catastrophic forgetting on Neurocoder through the residual weight $\mathbf{R}$, we excluded $\mathbf{R}$ in this experiment. We demonstrate that Neurocoder can improve these continual learning algorithms without requiring additional assumptions as in other approaches (Lopez-Paz & Ranzato, 2017; Shin et al., 2017; Serra et al., 2018) that either utilise task embedding or replay memory.

**Split MNIST** We first considered the split MNIST dataset–a standard continual learning benchmark wherein the original MNIST was split into a 5 2-way classification tasks, consecutively presented to a *Multi-layer Perceptron* Main Network (MLP). We followed the benchmarking as in Hsu et al. (2018) in which various optimisers and continual learning methods were examined under incremental task and domain scenarios. We measured the performance of the MLP versus Neurocoder and NSM. In both scenarios, Neurocoder was compatible with all continual leaning methods, demonstrating superior performance over MLP and NSM with performance gain between 1 to 16% (see Table 6 and Appendix's Table 1).

**Split CIFAR** We verified the scalability of Neurocoder to more challenging datasets. We split CIFAR datasets as in the split MNIST, resulting in 5-task 2-way split CIFAR10 and a 20-task 5-way split CIFAR100. We used Main Network *ResNet* (He et al., 2016)–a very deep CNN architecture.

| Configuration | Without $\mathbf{R}$ | | With $\mathbf{R}$ | |
|---|---|---|---|---|
| | $H=1$ | $H=3$ | $H=1$ | $H=3$ |
| CIFAR-10 | 77.4 | 80.8 | 88.9 | **95.6** |
| CIFAR-100 | 70.8 | 75.0 | 75.1 | **79.3** |

*Table 2.* CIFAR: accuracy of Neurocoder with different $H$ and $\mathbf{R}$.

When we stressed the orthogonal loss ($a = 10$) and used bigger program memory (100 slots), Neurocoder improved ResNet classification by 15% and 10% on CIFAR10 and CIFAR100, respectively. When we integrated Neurocoder with Synaptic Intelligence (SI (Zenke et al., 2017)), the performance was further improved, maintaining a stable performance above 80% accuracy for CIFAR10 and outperforming using SI alone by 10% for CIFAR100. We visualised the program attention over training time, and realize clear shift in attention pattern as new tasks appear. The details are given in Appendix B.4.

### 3.5. Ablation Study

Here we conduct more ablation studies considering the impact of the number of program attention heads ($H$) and residual weight ($\mathbf{R}$) in computer vision tasks. In particular, under the same training as in §3.1 we ran Neurocoder (DenseNet backbone) on CIFAR10 and CIFAR100 datasets using different $H$ and with/without $\mathbf{R}$. The result in Table 2 confirms the necessity of having multiple attention heads and residual weight, which may help Neurocoder capture better fine-grained details of the images for classification.

However, the residual weight is not always beneficial, especially in continual learning. As normal, $\mathbf{R}$ suffers from catastrophic forgetting. We verify that in the Split MNIST experiment. Both settings of incremental task and domain show a clear drop of performance, up to 2% and 8%, respectively, when Neurocoder is equipped with $\mathbf{R}$, as shown in Appendix's Tables 7 and 1.

We note that other studies regarding multi-step attention ($J$), usage-based attention, the number of programs ($P$) and the orthogonal coefficient ($a$), were already embedded in the above experimental sections (§3.1, §3.2, and §3.4, respectively). These studies collect evidences showing that our proposed components are critical for improving Neurocoder's performance in a variety of tasks.

## 4. Related Works

Memory-augmented neural networks (MANN) treats neural activations as data stored in an external memory, resembling Turing Machines (Graves et al., 2014; 2016; Weston et al., 2014; Sukhbaatar et al., 2015). Sophisticate models (Andreas et al., 2016; Le et al., 2020) step further by employing a memory of weights, which is analogous to the program memory concept in modern computers and thus, can perform adaptive computation by switching program during processing. Sharing similar ideas, conditional computation methods also use a bank of separate big programs, each of which is a weight matrix containing numerous parameters (Jacobs et al., 1991; Shazeer et al., 2017; Le et al., 2020). These attempts are not only inefficient in terms of storage, but also introduce redundancy and program collapse as the number of programs increases. Recent works extend the concept to routing networks, trained with reinforcement learning (Rosenbaum et al., 2018). Unlike them, in its program memory, Neurocoder maintains only shareable, smaller components that can reconstruct the whole program space, thereby heavily utilising the parameters and preventing the model from proliferating. On the application side, our solution offers a single framework that is scalable and adaptable to various problems and learning paradigms, which is not limited to single domain as in prior works.

Another way to achieve adaptive computation is via modifying the neural weight. HyperNet (Ha et al., 2017) and FiLM (Perez et al., 2018) rescale the weight conditioned on the input using multiplicative operations (similar to the residual weight $\mathbf{R}$ in Neurocoder). However, our method focuses more on modular representation of the weights, and can compose new weight from stored singular programs. Although less related, neural program synthesis (Reed & De Freitas, 2015; Shu & Zhang, 2017) shares with Neurocoder the goal to generate program from input-output examples. However, Neurocoder's program is not a real program with proper programming syntax. Instead, it is a metaphor for the weight matrix of neural networks and thus, simple and easy to combine with other Main Networks. Our model is also orthogonal to approaches employing tensor decomposition to reduce the number of parameters or hasten the computation (Novikov et al., 2015; Lebedev et al., 2015). Neurocoder composes rather than decompose the neural weights. Our aim is not only to enable efficient parameter usage, but also achieve general-purpose computing power that enables generalisation to various domains.

## 5. Discussion

Our experiments demonstrate that Neurocoder is capable of re-coding Neural Programs in distinctive neural networks, amplifying their capabilities in diverse learning scenarios: instance-based, sequential, multi-task and continual learning. This consistently results in significant performance increase, and further creates novel robustness to pattern shift and catastrophic forgetting. This ability for each architecture to re-code itself is made possible without changing the way it is trained, or majorly increasing the number of parameters it needs to learn (see Appendix Table 9).

The MNIST problem illustrates the reasoning process of

Neurocoder when classifying digit images wherein its singular program assignment resembles a binary tree decision-making process - it shows how some singular programs are shared, others are not. The sequential problems highlight the importance of efficient memory utilisation in re-constructing the working program enabling discovery of hidden structures in sequential data and allowing RL agents to solve complex games wherein traditional methods fail or learn slowly. Neurocoder also works well with multi-task setting, as shown in the challenging multi-algorithm benchmark. Finally, continual learning problems show that Neurocoder mitigates catastrophic forgetting efficiently under different learning settings/algorithms.

One limitation of this work is the number of additional hyperparameters, which prevents us from fully tuning Neurocoder. In future work, we will extend Neurocoder's application beyond feed-forward layers. It would be interesting to efficiently replace all neural layers including CNN or Transformer by Neurocoder's programs.

## Acknowledgements

This research was partially funded by the Australian Government through the Australian Research Council (ARC). Prof Venkatesh is the recipient of an ARC Australian Laureate Fellowship (FL170100006).

## References

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 39–48, 2016.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015. URL http://arxiv.org/abs/1409.0473.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734. Association for Computational Linguistics, October 2014. doi: 10.3115/v1/D14-1179. URL https://www.aclweb.org/anthology/D14-1179.

French, R. M. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Ha, D., Dai, A. M., and Le, Q. V. Hypernetworks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rkpACe1lx.

Happel, B. L. and Murre, J. M. Design and evolution of modular neural network architectures. *Neural networks*, 7(6-7):985–1004, 1994.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hsu, Y.-C., Liu, Y.-C., Ramasamy, A., and Kira, Z. Re-evaluating continual learning scenarios: A categorization and case for strong baselines. In *NeurIPS Continual learning Workshop*, 2018. URL https://arxiv.org/abs/1810.12488.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Hubel, D. *Eye, Brain, and Vision*. Scientific American Library series. Scientific American Library, 1988. URL https://books.google.com.au/books?id=HaHmtwEACAAJ.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. *TR-2009*, 2009.

Le, H., Tran, T., and Venkatesh, S. Neural stored-program memory. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkxxA24FDr.

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *International Conference on Learning Representations*, 2015.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lopez-Paz, D. and Ranzato, M. Gradient episodic memory for continual learning. In *Advances in neural information processing systems*, pp. 6467–6476, 2017.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. Tensorizing neural networks. In *Advances in neural information processing systems*, pp. 442–450, 2015.

Perez, E., Strub, F., De Vries, H., Dumoulin, V., and Courville, A. FiLM: Visual Reasoning with a General Conditioning Layer. In *AAAI Conference on Artificial Intelligence*, New Orleans, United States, February 2018. URL https://hal.inria.fr/hal-01648685.

Reed, S. and De Freitas, N. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Rosenbaum, C., Klinger, T., and Riemer, M. Routing networks: Adaptive selection of non-linear functions for multi-task learning. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=ry8dvM-R-.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., and Lillicrap, T. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pp. 1842–1850, 2016.

Schmidhuber, J. Making the world differentiable: On using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environm nts. *TR FKI-126-90*, 1990.

Schmidhuber, J. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

Serra, J., Suris, D., Miron, M., and Karatzoglou, A. Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*, pp. 4548–4557, 2018.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

Shin, H., Lee, J. K., Kim, J., and Kim, J. Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems*, pp. 2990–2999, 2017.

Shu, C. and Zhang, H. Neural programming by example. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Turing, A. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, 1936.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

von der Malsburg, C. The correlation theory of brain function, 1981. URL http://cogprints.org/1380/.

Von Neumann, J. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

Weston, J., Chopra, S., and Bordes, A. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Zenke, F., Poole, B., and Ganguli, S. Continual learning through synaptic intelligence. *Proceedings of machine learning research*, 70:3987, 2017.

## Appendix

### A. Implementation details

In this section, we describe how we implement $f_\varphi$ in practice. Assume that the size of memories $\mathbf{M}_U, \mathbf{M}_V$ and $\mathbf{M}_S$are , $P \times d_1$ , $P \times d_2$ and $P \times 1$, respectively; and the key size is $K$. Then $f_\varphi$ is a neural network with an input size of $d_1 + d_2 + 1$ and output size of $3K$ ($K \ll d_{1,2}$). $f_\varphi$ takes the concatenation (detached) $[\mathbf{M}_U\left[i\right], \mathbf{M}_V\left[i\right], \mathbf{M}_S\left[i\right]$ ] as input where $i$ is the row index, and outputs 3 key vectors. We then perform attention to each of the 3 key vectors to get the attention weight to each slot in the memories. The weights are used to retrieve the components as in Eq. 3-4.

### B. Details of experiments

#### B.1. Instance-based learning experiments

***Image classification-linear Main Network*** We used the standard training and testing set of MNIST dataset. To train the models, we used the standard SGD with a batch size of 32. Each MNIST image was flattened to a 768-dimensional vector, which requires a linear classifier of $7,680$ parameters to categorise the inputs into 10 classes. For Neurocoder, we used Program Memory with $P = 6$ and $K = 2$. The Program Controller's composition network was an LSTM with a hidden size of 8. We controlled the number of parameters of Neurocoder, which included parameters for the Program Memory and the Program Controller by reducing the input dimension using random projection $z_t = x_t U$ with $U \in \mathbb{R}^{768\times200}$ initialised randomly and fixed during the training. We also excluded the program integration to eliminate the effect of the residual program $\mathbf{R}$. Given the flattened image input $x_t$, Neurocoder generated the active program $\mathbf{P}_t$, predicting the class of the input as $y_t = argmax\left(x_t \mathbf{P}_t\right)$. The performance of the linear classifier was imported from (LeCun et al., 1998) and confirmed by our own implementation.

***Image classification-deep Main Network*** We used the standard training and testing sets of CIFAR datasets. For most experiments, we use Adam optimiser with a batch size of 128. *The deep Main Networks were adopted from the original papers, resulting in* 3*-layer MLP,* 5*-layer LeNet* (LeCun et al., 1998) *and* 100*-layer DenseNet* (Huang et al., 2017)[2]. The other baselines for this task included a recent sparse Mixture of Experts (MOE (Shazeer et al., 2017)) and the Neural Stored-program Memory (NSM (Le et al., 2020)). For this case, we employed the program integration with the residual program $\mathbf{R}$ to flexibly fit to the data distribution.

[2]Only for experiments with DenseNet, to closely match the reported results, we followed the original training with SGD optimizer, scheduling learning rate and batch size of 32.

#### B.2. Sequential learning experiments

***Synthetic polynomial auto-regression*** A sequence was divided into $n_{pa}$ chunks, each of which associated with a randomly generated polynomial. The degree and coefficients of each polynomial were sampled from $U \sim [2, 10]$ and $U \sim [-1, 1]$, respectively. Each sequence started from $x_1 = -5$ and ended with $x_T = 5$, equally divided into $n_{pa}$ chunks. Each chunk contained several consecutive points $(x_t, y_t)$ from the corresponding polynomial, representing a local transformation from the input to the output. Given previous points $(x_{<t}, y_{<t})$ and the current $x$-coordinate $x_t$, the task was to predict the current $y$-coordinate $y_t$. To be specific, at each timestep, the Main Network GRU was fed with $(x_t, y_{t-1})$ and trained to predict $y_t$ by minimizing the mean square error $1/T\sum_{t=1}^{T}\left(\hat{y}_t - y_t\right)^2$ where $y_0 = 0$, $\hat{y}_t$ is the prediction of the network and $y_t$ the ground truth.

We augmented *GRU by applying Neurocoder* and HyperNet (Ha et al., 2017) *to the output layer of the GRU*. Here, the HyperNet baseline (Ha et al., 2017) generated adaptive scales for the output weight while the FiLM baseline (Perez et al., 2018) modulates the activation of the output layer. We trained the networks with Adam optimiser with a batch size of 128. To balance the model size, we used GRU's hidden size of 32, 28, 32, 16 and 8 for the original Main Network, HyperNet, FiLM, single-step and multi-step Neurocoder, respectively. We also excluded program integration phase in Neurocoders to keep the model size equivalent to or smaller than that of the Main Network.

We compared three configurations of Neurocoder - single-step, multi-head ($J = 1$, $H = 15$), multi-step, single-head ($J = 5$, $H = 1$) and multi-step without usage-based attention- against the original GRU with output layer made by MLP, HyperNet and FILM. We found that MLP failed to learn and converge within $10,000$ learning iterations. In contrast, both Neurocoders learn and converge, in as little as only $2,000$ iterations with the multi-step Neurocoder. HyperNet and FiLM converged much slower than Neurocoders and could not minimize the predictive error as well as Neurocoders when Gaussian noise (mean 0, variance $0.3 \times \max_t y_t$) is added or the number of polynomials ($n_{pa}$) is doubled (see Fig. 5).

***Atari 2600 games*** We used OpenAI's Gym environments to simulate Atari games. We used the standard environment settings, employing no-frame-skip versions of the games. The picture of the game snapshot was preprocessed by CNNs and the A3C agent was adopted from the original paper with default hyper-parameters as in Mnih et al. (2016). *The actor/critic network of A3C was LSTM whose output layer's working program was provided by Neurocoder* or HyperNet. The hidden size of the LSTM was 512 for all baselines.

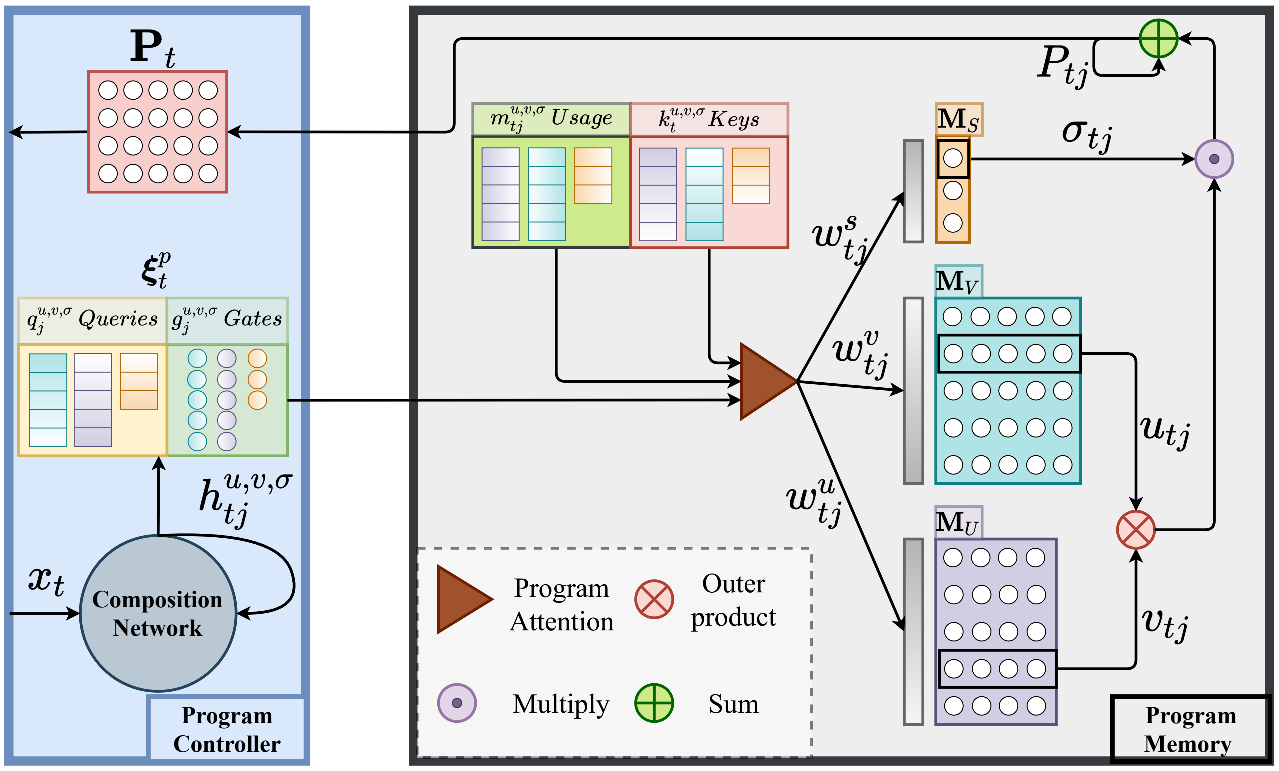

*Figure 4.* Active program coding. The Program Controller uses the composition network (a recurrent neural network) to process the input $x_t$ and generate composition signal $\boldsymbol{\xi}_t^p$, which is composed of the queries ($q$) and the interpolating gates ($g$). The similarity of the query to program memory keys ($k$) is then computed together with the memory usage ($m$) from which attention weights for the Program Memory are derived. The active program $\mathbf{P}_t$ is then "coded" through low-rank approximation using the $j$-th component accessed by recurrent attentions. For simplicity, one attention head is shown ($H = 1$).

| Notation | Meaning | Location | |
|---|---|---|---|
| | | Program Controller | Program Memory |
| | **Trainable parameters** | | |
| $\theta^{u,v,\sigma}$ | Composition network | ✓ | |
| $\phi$ | Integration network | ✓ | |
| $\varphi^{u,v,\sigma}$ | Key generator network | | ✓ |
| $\mathbf{R}$ | Residual program (optional) | ✓ | |
| $\mathbf{M}_U$ | Memory of left singular vectors | | ✓ |
| $\mathbf{M}_V$ | Memory of right singular vectors | | ✓ |
| $\mathbf{M}_S$ | Memory of singular values | | ✓ |
| | **Control variables** | | |
| $\boldsymbol{\xi}_t^p$ | Composition control signal | ✓ | |
| $\boldsymbol{\lambda}_t^p$ | Integration control signal | ✓ | |
| $k^{u,v,\sigma}$ | Program keys | | ✓ |
| $m^{u,v,\sigma}$ | Program usages | | ✓ |
| | **Hyper-parameters** | | |
| $P$ | Number of memory slots | | ✓ |
| $K$ | Key dimension | | ✓ |
| $l_I$ | Number of considered least-used slots | | ✓ |
| $J$ | Number of recurrent attention steps | ✓ | |
| $H$ | Number of attention heads | ✓ | |
| $a$ | Orthogonal loss weight | | ✓ |

*Table 3.* Important parameters of Neurocoder.

| Architecture | Task | Original | MOE | NSM | Neurocoder |
|---|---|---|---|---|---|
| MLP | CIFAR10 | 52.06 | 50.76 | 52.76 | **54.86** |
| | CIFAR100 | 23.31 | 22.79 | 25.65 | **26.24** |
| LeNet | CIFAR10 | 75.71 | 75.88 | 75.45 | **78.92** |
| | CIFAR100 | 42.73 | 42.47 | 43.14 | **47.21** |
| DenseNet | CIFAR10 | 93.61 | 80.61 | 94.24 | **95.61** |
| | CIFAR100 | 78.11 | 69.48 | 71.76 | **79.34** |

*Table 4.* Best test accuracy over 5 runs on image classification tasks comparing original architecture, Mixture of Experts (MOE), Neural Stored-program Memory (NSM) and our architecture (Neurocoder). Three architectures of the Main Network of Neurocoder were considered: 3-layer perceptron (MLP), 5-layer CNN (LeNet (LeCun et al., 1998)) and very deep Densely Connected Convolutional Networks (DenseNet (Huang et al., 2017)). We employed two classical image classification datasets: CIFAR10 and CIFAR100.

| Tasks | Configuration | Input $\rightarrow$ Output |
|---|---|---|
| Copy | Sequence length range: [1, 3] | $x_1, ..., x_T \rightarrow x_1, ..., x_T$ |
| Repeat Copy | Sequence length range: [1, 3]<br>#Repeat range $n$: [1, 2] | $n, x_1, ..., x_T \rightarrow [x_1, ..., x_T] \times n$ |
| Associative Recall | #Item range: [2, 3]<br>Item length: 2 | $[x_{1,1}, x_{1,2}], ..., [x_{T,1}, x_{T,2}], [x_{i,1}, x_{i,2}] \rightarrow [x_{i+1,1}, x_{i+1,2}]$ |
| Priority Sort | #Item: 3<br>#Sorted Item: 2 | $[x_1, p_1], [x_2, p_2], [x_3, p_3] \rightarrow x_{i_1}, x_{i_2}$ s.t. $p_{i_1} \geq p_{i_2} \geq p_{i_3}$ |

*Table 5.* Algorithmic tasks used in multi-task learning.

*Seaquest and MsPacman*. The original A3C agent was able to learn and obtain a moderate score of around $2,500$ after 32 million environment steps. We also equipped A3C with HyperNet-based actors/critics, however, the performance remained unchanged, with scores of about $2/3$ of Neurocoder-based agent's.

### B.3. Multi-task learning experiments

Table 5 lists the input-output structure and configuration of the algorithmic tasks: Copy, Repeat Copy, Associative Recall and Priority Sort. Basically, each timestep of the input sequence presents a binary vector of 8 bits. In our multi-task setting, the input-output structure is $t_1, x^1_{1...I_1}, t_2, x^2_{1...I_2}, t_3, x^3_{1...I_3}, t_4, x^4_{1...I_4} \rightarrow y^1_{1...O_1}, y^2_{1...O_2}, y^3_{1...O_3}, y^4_{1...O_4}$ where $\left\{t_i, x^i_{1...I_i}, y^i_{1...O_i}\right\}^4_{i=1}$ is the task identification, input and output sequence of the task, respectively. The multi-task learning problem was challenging because the models must distinguish tasks by remembering the task identifications and learn to solve different algorithms by generating different interface programs in accordance with each task.

Following Le et al. (2020), we used the same NTM with 1 read and 1 write head, and applied Neurocoder and other conditional computing methods to the interface network of the NTM, which is a single-layer MLP with $\tanh$ activation. We used single-step attention Neurocoder for this task to keep the number of parameters comparable with other models. We trained the models with RMSProp optimiser with learning rate of $10^{-4}$ and batch size of 64 to minimise the cross-entropy loss of the ground truth output and the predicted one. The evaluation metric was $\%$ bit error, which was computed for a sequence as $\frac{\#\text{ wrong bits}}{\#\text{ total bits}} \times 100$.

### B.4. Continual learning experiments

We evaluate the models under incremental task and domain setting. The former is easier as each task uses its own dedicated prediction head (catastrophic forgetting only happens at lower layers) while the latter shares the prediction head across task.

***Split MNIST*** We used the same 2-layer MLP and continual learning baselines as in Hsu et al. (2018). Here, we again excluded program integration to avoid catastrophic forgetting happening on the residual program $\mathbf{R}$. Remarkably, the NSM with much more parameters could not improve MLP's performance, illustrating that simple modular conditional computation is not enough for continual learning (see Table 6).

***Split CIFAR*** The 18-layer *ResNet* implementation was adopted from Pytorch's official release whose weights was pretrained with ImageNet dataset. When performing continual learning with CIFAR images, we froze all except for the output layers of ResNet, which was a 3-layer MLP. We only tuned the hyper-parameters of SI and Neurocoder for this task.

In the CIFAR10 task, compared to the monolithic ResNet, the Neurocoder-augmented ResNet could achieve much higher accuracy when we finished the learning for all 5 tasks (55% versus 70%, respectively). Also, we realised that stressing the orthogonal loss further improved the performance. When we employed Synaptic Intelligence (SI (Zenke et al., 2017)), the performance of ResNet improved, yet it still dropped gradually to just above 70%. In contrast, the Neurocoder-augmented ResNet with SI maintained a stable performance above 80% accuracy (see Fig. 7 (left)). To show the adaption of program attention, we visualised the attention pattern over training time in Fig. 8. Overall, Neurocoder can learn to find different programs as the task's data shift in distribution. Thus, knowledge of old tasks is preserved.

In the CIFAR100 task, Neurocoder alone with a bigger program memory slightly exceeded the performance of SI, which was about 10% better than ResNet. Moreover, Neurocoder plus SI outperformed using only SI by another 10% of accuracy as the number of seen tasks grew to 20 (see Fig. 7 (right)).

## C. Training procedure and hyper-parameter selections

For all experiments, Neurocoder was jointly trained with Main Networks. We trained all the models using single GPU NVIDIA V100-SXM2. Running time depends on task, the longest task is multi-task learning with MN as NTM, which took 1 day for 1 training run with Neurocoder. Adding Neurocoder makes the training slower about 30%, yet still faster than MOE or NSM. However, compared to HyperNet or FILM, Neurocoder is still slower by 15%, which is the limitation of Neurocoder.

The learning rate of optimisers was set to default value unless stated otherwise. The Main Network's hyper-parameters were fixed and we only tuned the hyper-parameters of Neurocoder and its competitors: MOE, NSM, HyperNet and FILM. In particular, for Neurcocoder, main hyper-parameters such as number of memory slots ($P$), recurrent steps ($J$), and heads ($H$) were selected from $\{10, 20, 30, 50, 80, 100\}$, $\{1, 5\}$ and $\{1, 5, 15\}$, respectively. Hyper-parameters such as number of least-used slots ($l_I$) key dimension ($K$), orthogonal loss weight ($a$) was selected from $\{2, 5\}$, $\{3, 5\}$ and $\{0.1, 10\}$, respectively.

For MOE, we tuned the total number of experts and top-$k$ chosen experts from range $\{10, 50, 80, 100\}$ and $\{1, 5, 10\}$, respectively. For NSM, we tuned the number of program memory slots $\{5, 10, 50\}$. Other hyper-parameters of MOE and NSM were kept as in the original papers. For HyperNet and FiLM, chosen as MLPs (ReLU activation), we tuned the number of layers $\{1, 2\}$ and hidden size $\{64, 128, 256\}$.

We report details of best hyper-parameters and model size for each tasks in Table 8 and 9, respectively. Readers are referred to Table 3 for the complete list of parameters in Neurocoder.

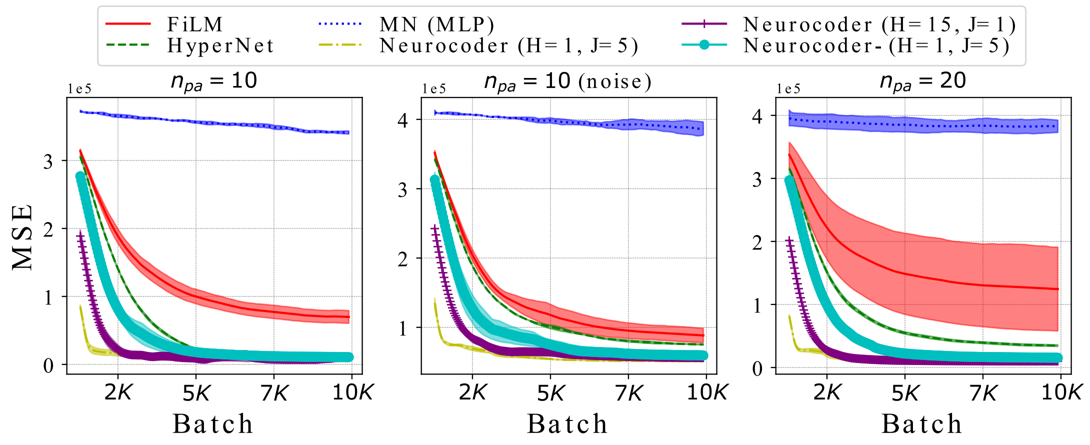

*Figure 5.* Polynomial auto-regression: mean square error (MSE) over training iterations with a batch size of 128 comparing FiLM, HyperNet, Main Network (MLP), single-step, multi-step Neurocoders. - denotes the ablated Neurocoder without usage-based attention. The learning curves are taken average over 5 runs.

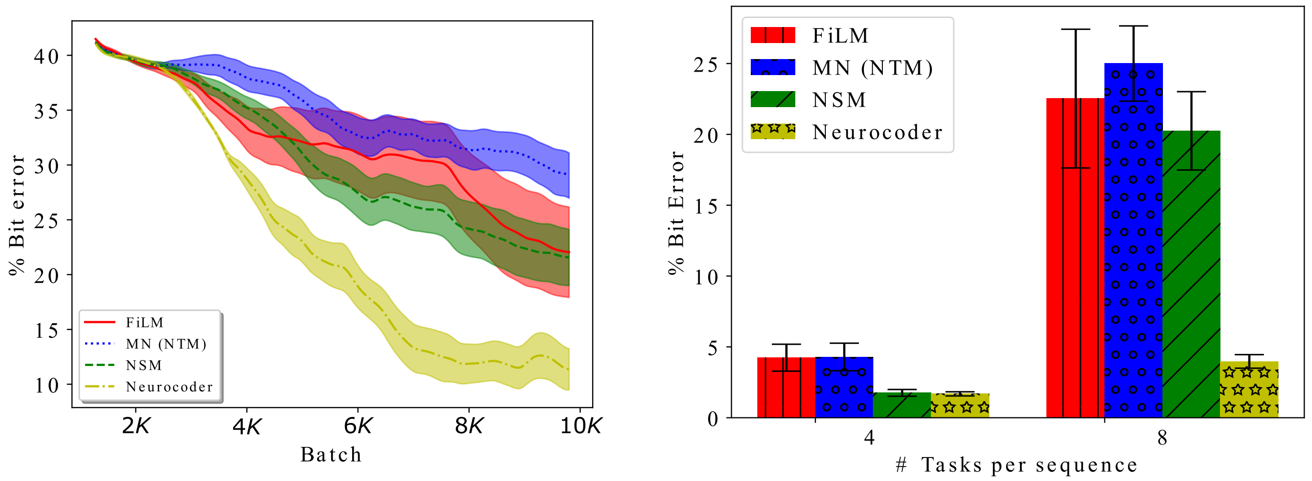

*Figure 6.* Multi-algorithm learning task (mean and std. over 5 runs). Left: Bit error over training steps (8 tasks per sequences). Right: Average bit error on different testing settings measured by best checkpoints. Lower is better.

| Method | MN (MLP (Hsu et al., 2018)) | MN (MLP ours) | NSM | Neurocoder ($\mathbf{R}$) | Neurocoder (no $\mathbf{R}$) |
|---|---|---|---|---|---|
| Adam | 93.46±2.01 | 93.75±3.28 | 87.55± 4.38 | 94.91± 3.29 | **96.54±1.39** |
| Adagrad | 98.06±0.53 | 98.02±0.89 | 96.63±1.49 | 98.04±0.85 | **99.01±0.19** |
| L2 | 98.18±0.96 | 98.14±0.43 | 91.44± 3.80 | 98.13± 0.81 | **98.35±0.74** |
| SI | 98.56±0.49 | 98.69±0.20 | 98.87±0.20 | 98.72±0.21 | **99.14±0.24** |
| EWC | 97.70±0.81 | 97.00±1.10 | 93.94±2.36 | 97.04±0.07 | **97.88±0.22** |
| O-EWC | 98.04±1.10 | 98.23±1.17 | 96.11±1.27 | 98.27±1.73 | **98.30±1.48** |

*Table 6.* Incremental task continual learning with Split MNIST. Final test accuracy (mean and std.) over 10 runs. We use 2 variants of Neurocoder: with and without the residual weight $\mathbf{R}$

| Method | MN (MLP (Hsu et al., 2018)) | MN (MLP ours) | NSM | Neurocoder ($\mathbf{R}$) | Neurocoder (no $\mathbf{R}$) |
|---|---|---|---|---|---|
| Adam | 55.16±1.38 | 53.55±1.27 | 54.85±2.81 | 54.72± 1.37 | **58.46±0.46** |
| Adagrad | 58.08±1.06 | 57.83±2.74 | 58.42±1.87 | 58.59±2.38 | **62.28±4.03** |
| L2 | 66.00±3.73 | 64.37±2.40 | 62.83±7.21 | 64.18± 1.33 | **69.89±1.72** |
| SI | 64.76±3.09 | 64.41±3.36 | 64.36±2.99 | 65.98±4.74 | **67.96±3.22** |
| EWC | 58.85±2.59 | 58.41±2.37 | 58.12±3.24 | 60.82±2.63 | **65.66±1.25** |
| O-EWC | 57.33±1.44 | 57.78±1.84 | 58.55±3.40 | 65.17±2.45 | **73.97±1.50** |

*Table 7.* Incremental domain continual learning with Split MNIST. Final test accuracy (mean and std.) over 10 runs. We use 2 variants of Neurocoder: with and without the residual weight $\mathbf{R}$

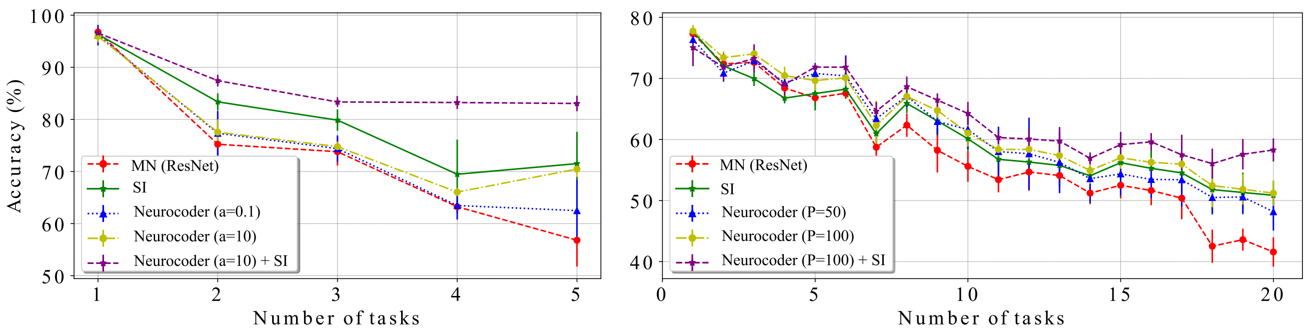

*Figure 7.* Incremental task continual learning with Split CIFAR10 (left) and CIFAR100 (right). Average classification accuracy with error bar over all learned tasks as a function of number of tasks.

| Task | Neurocoder | | Use $\mathbf{R}$ |
|---|---|---|---|
| MNIST | $P=5, J=5, H=1$ <br> $K=2, l_I=2, a=0.1$ | | ✗ |
| CIFARs | $P=30, J=5, H=3$ <br> $K=5, l_I=5, a=0.1$ | | ✓ |
| Polynomial auto-regression | $P=10, J=1, H=15$ <br> $K=3, l_I=0, a=0.1$ | $P=20, J=5, H=1$ <br> $K=3, l_I=2, a=0.1$ | ✓ |
| Atari games | $P=80, J=1, H=15, K=3, l_I=5, a=0.1$ | | ✓ |
| Multi-algorithm | $P=30, J=1, H=5, K=5, l_I=2, a=10$ | | ✓ |
| Split MNIST | $P=50, J=1, H=10, K=5, l_I=5, a=10$ | | ✗ |
| Split CIFARs | $P=100, J=1, H=10, K=5, l_I=5, a=10$ | | ✗ |

*Table 8.* Best hyper-parameters of Neurocoder in all experiments. For polynomial auto-regression task, two Neurocoder configurations are included, corresponding to single-step and multi-step Neurocoder.

| Task | Main Network | Original | MOE | NSM | HyperNet | FiLM | Neurocoder |
|---|---|---|---|---|---|---|---|
| MNIST | Linear classifier | 7.8K | – | – | – | – | 7.3K |
| | 3-layer MLP | 1.7M | 15.4M | 21.2M | – | – | 1.9M |
| CIFARs | LeNet | 2.1M | 12.3M | 27.1M | – | – | 2.3M |
| | DenseNet | 7.0M | 20.5M | 16.7M | – | – | 7.3M |
| Polynomial auto-regression | GRU | 3.4K | – | – | 3.5K | 3.6K | 3.6K 2.1K |
| Atari games | LSTM | 3.2M | – | – | 3.6M | – | 3.3M |
| Multi-algorithm | NTM | 308K | – | 264K | – | 254K | 255K |
| Split MNIST | 2-layer MLP | 328K | | 2.3M | | – | 348K |
| Split CIFARs | ResNet | 12.6M | – | – | – | – | 12.6M |

*Table 9.* Number of parameters of machine learning models in all experiments. The parameter count includes the parameter of the Main Network and the conditional computing model. – denotes not available. For tasks that contain different datasets, leading to slightly different model size, the numbers of parameters are averaged. For polynomial auto-regression task, two Neurocoder configurations are included, corresponding to single-step and multi-step Neurocoder.

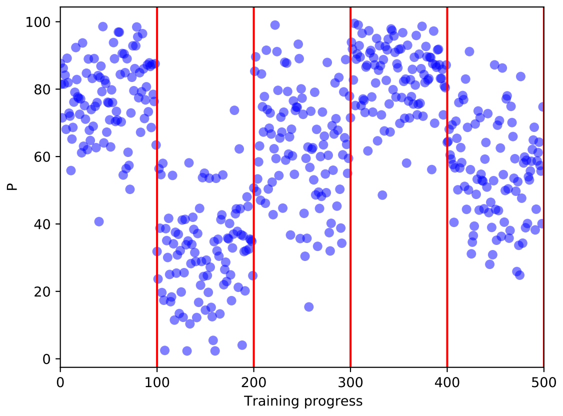

*Figure 8.* Split CIFAR10 attention during training. At any training step, we can find the mostly attended program memory slot in $\mathbf{M}_U$ by measuring the attention weight $w$. A blue point represents the index of the the top attended slot (over 100 slots). During the training of each task, we sample the top attended slots at 100 training steps, resulting in a total of 500 points in the plot. Vertical red lines separate 5 tasks.