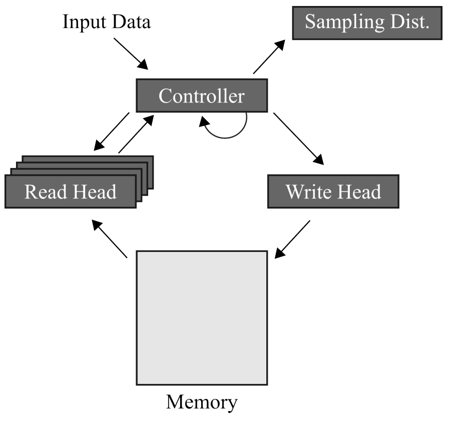
Supplementary Information

6.1. Additional model details

Our model is a variant of a Neural Turing Machine (NTM) from Graves et al. It consists of a number of differentiable components: a controller, read and write heads, an external memory, and an output distribution. The controller receives input data (see section 7) directly, and also provides an input to the output distribution. Each of these components will be addressed in turn.



(a)

Figure 7. MANN Architecture.

The controllers in our experiments are feed-forward networks or Long Short-Term Memories (LSTMs). For the best performing networks, the controller is a LSTM with 200 hidden units. The controller receives some concatenated input $(\mathbf{x}_t, \mathbf{y}_{t-1})$ (see section 7 for details) and updates its state according to:

$$\hat{\mathbf{g}}^f, \hat{\mathbf{g}}^i, \hat{\mathbf{g}}^o, \hat{\mathbf{u}} = \mathbf{W}^{xh}(\mathbf{x}_t, \mathbf{y}_{t-1}) + \mathbf{W}^{hh}\mathbf{h}_{t-1} + \mathbf{b}^h, \quad (9)$$

$$\mathbf{g}^f = \sigma(\hat{\mathbf{g}}^f), \quad (10)$$

$$\mathbf{g}^i = \sigma(\hat{\mathbf{g}}^i), \quad (11)$$

$$\mathbf{g}^o = \sigma(\hat{\mathbf{g}}^o), \quad (12)$$

$$\mathbf{u} = \tanh(\hat{\mathbf{u}}), \quad (13)$$

$$\mathbf{c}_t = \mathbf{g}^f \odot \mathbf{c}_{t-1} + \mathbf{g}^i \odot \mathbf{u}, \quad (14)$$

$$\mathbf{h}_t = \mathbf{g}^o \odot \tanh(\mathbf{c}_t), \quad (15)$$

$$\mathbf{o}_t = (\mathbf{h}_t, \mathbf{r}_t) \quad (16)$$

where $\hat{\mathbf{g}}^f$, $\hat{\mathbf{g}}^o$, and $\hat{\mathbf{g}}^i$ are the forget gates, output gates, and input gates, respectively, $\mathbf{b}^h$ are the hidden state biases, $\mathbf{c}_t$ is the cell state, $\mathbf{h}_t$ is the hidden state, $\mathbf{r}_t$ is the vector read from memory, $\mathbf{o}_t$ is the concatenated output of the controller, $\odot$ represents element-wise multiplication, and $(\cdot,\cdot)$ represents vector concatenation. $\mathbf{W}^{xh}$ are the weights from the input $(\mathbf{x}_t, \mathbf{y}_{t-1})$ to the hidden state, and $\mathbf{W}^{hh}$ are the weights between hidden states connected through time. The read vector $\mathbf{r}_t$ is computed using content-based addressing using a cosine distance measure, as described in the main text, and is repeated below for self completion.

The network has an external memory module, $\mathbf{M}_t$, that is both read from and written to. The rows of $\mathbf{M}_t$ serve as memory 'slots', with the row vectors themselves constituting individual memories. For reading, the controller cell state serves as a query for $\mathbf{M}_t$. First, a cosine distance measure is computed for the query key vector (here notated as $\mathbf{k}_t$) and each individual row in memory:

$$K\big(\mathbf{k}_t, \mathbf{M}_t(i)\big) = \frac{\mathbf{k}_t \cdot \mathbf{M}_t(i)}{\parallel \mathbf{k}_t \parallel \parallel \mathbf{M}_t(i) \parallel}, \quad (17)$$

Next, these similarity measures are used to produce a read-weight vector $\mathbf{w}_t^r$, with elements computed according to a softmax:

$$w_t^r(i) \leftarrow \frac{\exp\Big(K\big(\mathbf{k}_t, \mathbf{M}_t(i)\big)\Big)}{\sum_j \exp\Big(K\big(\mathbf{k}_t, \mathbf{M}_t(j)\big)\Big)}. \quad (18)$$

A memory, $\mathbf{r}_t$, is then retrieved using these read-weights:

$$\mathbf{r}_t \leftarrow \sum_i w_t^r(i)\mathbf{M}_t(i). \quad (19)$$

Finally, $\mathbf{r}_t$ is concatenated with the controller hidden state, $\mathbf{h}_t$, to produce the network's output $\mathbf{o}_t$ (see equation (16)). The number of reads from memory is a free parameter, and both one and four reads were experimented with. Four reads was ultimately chosen for the reported experimental results. Multiple reads is implemented as additional concatenation to the output vector, rather than any sort of combination or interpolation.

To write to memory, we implemented a new content-based access module called Least Recently Used Access (LRUA). LRUA writes to either the most recently read location, or the least recently used location, so as to preserve recent, and hence potentially useful memories, or to update recently encoded information. Usage weights $\mathbf{w}_t^u$ are computed each time-step to keep track of the locations most recently read or written to:

$$\mathbf{w}_t^u \leftarrow \gamma \mathbf{w}_{t-1}^u + \mathbf{w}_t^r + \mathbf{w}_t^w, \quad (20)$$

where γ is a decay parameter. The *least-used* weights, $\mathbf{w}_t^{lu}$, for a given time-step can then be computed using $\mathbf{w}_t^u$. First, we introduce the notation $m(\mathbf{v}, n)$ to denote the n^{th} smallest element of the vector $\mathbf{v}$. Elements of $\mathbf{w}_t^{lu}$ are set accordingly:

$$w_t^{lu}(i) = \begin{cases} 0 & \text{if } w_t^u(i) > m(\mathbf{w}_t^u, n) \\ 1 & \text{if } w_t^u(i) \leq m(\mathbf{w}_t^u, n) \end{cases}, \quad (21)$$

where n is set to equal the number of reads to memory.

To obtain the write weights $\mathbf{w}_t^w$, a learnable sigmoid gate parameter is used to compute a convex combination of the previous read weights and previous least-used weights:

$$\mathbf{w}_t^w \leftarrow \sigma(\alpha)\mathbf{w}_{t-1}^r + (1-\sigma(\alpha))\mathbf{w}_{t-1}^{lu}, \tag{22}$$

where α is a dynamic scalar gate parameter to interpolate between the weights. Prior to writing to memory, the least used memory location is computed from $\mathbf{w}_{t-1}^u$ and is set to zero. Writing to memory then occurs in accordance with the computed vector of write weights:

$$\mathbf{M}_t(i) \leftarrow \mathbf{M}_{t-1}(i) + w_t^w(i)\mathbf{k}_t, \forall i \tag{23}$$

6.2. Output distribution

The controller's output, $\mathbf{o}_t$, is propagated to an output distribution. For classification tasks using one-hot labels, the controller output is first passed through a linear layer with an output size equal to the number of classes to be classified per episode. This linear layer output is then passed as input to the output distribution. For one-hot classification, the output distribution is a categorical distribution, implemented as a softmax function. The categorical distribution produces a vector of class probabilities, $\mathbf{p}_t$, with elements:

$$p_t(i) = \frac{\exp(\mathbf{W}^{op}(i)\mathbf{o}_t)}{\sum_j \exp(\mathbf{W}^{on}(j)\mathbf{o}_t)}, \tag{24}$$

where $\mathbf{W}^{op}$ are the weights from the controller output to the linear layer output.

For classification using string labels, the linear output size is kept at 25. This allows for the output to be split into five equal parts each of size five. Each of these parts is then sent to an independent categorical distribution that computes probabilities across its five inputs. Thus, each of these categorical distributions independently predicts a 'letter,' and these letters are then concatenated to produce the five-character-long string label that serves as the network's class prediction (see figure 8).

A similar implementation is used for regression tasks. The linear output from the controller outputs two values: μ and σ, which are passed to a Gaussian distribution sampler as predicted mean and variance values. The Gaussian sampling distribution then computes probabilities for the target value y_t using these values.

6.3. Learning

For one-hot label classification, given the probabilities output by the network, $\mathbf{p}_t$, the network minimizes the episode loss of the input sequence:

$$\mathcal{L}(\theta) = -\sum_t \mathbf{y}_t^T \log \mathbf{p}_t, \tag{25}$$

where $\mathbf{y}_t$ is the target one-hot or string label at time t (note: for a given one-hot class-label vector $\mathbf{y}_t$, only one element assumes the value 1, and for a string-label vector, five elements assume the value 1, one per five-element 'chunk').

For string label classification, the loss is similar:

$$\mathcal{L}(\theta) = -\sum_t \sum_c \mathbf{y}_t^T(c) \log \mathbf{p}_t(c). \tag{26}$$

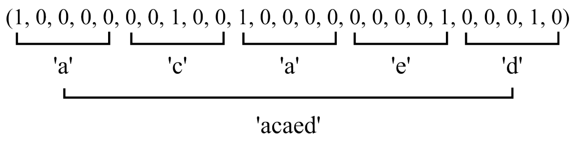
Here, the (c) indexes a five-element long 'chunk' of the vector label, of which there are a total of five.

For regression, the network's output distribution is a Gaussian, and as such receives two-values from the controller output's linear layer at each time-step: predictive μ and σ values, which parameterize the output distribution. Thus, the network minimizes the negative log-probabilities as determined by the Gaussian output distribution given these parameters and the true target y_t.

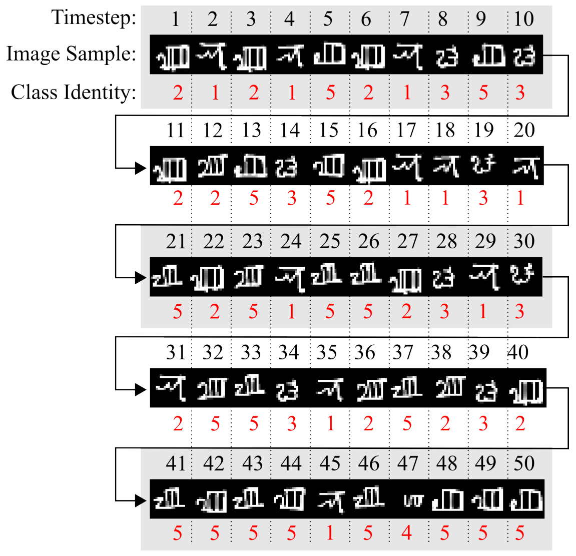
7. Classification input data

Input sequences consist of flattened, pixel-level representations of images $\mathbf{x}_t$ and time-offset labels $\mathbf{y}_{t-1}$ (see figure 8 for an example sequence of images and class identities for an episode of length 50, with five unique classes). First, N unique classes are sampled from the Omniglot dataset, where N is the maximum number of unique classes per episode. N assumes a value of either 5, 10, or 15, which is indicated in the experiment description or table of results in the main text. Samples from the Omniglot source set are pulled, and are kept if they are members of the set of n unique classes for that given episode, and discarded otherwise. $10N$ samples are kept, and constitute the image data for the episode. And so, in this setup, the number of samples per unique class are not necessarily equal, and some classes may not have any representative samples. Omniglot images are augmented by applying a random rotation uniformly sampled between $-\frac{\pi}{16}$ and $\frac{\pi}{16}$, and by applying a random translation in the x- and y- dimensions uniformly sampled between -10 and 10 pixels. The images are then downscaled to 20x20. A larger class-dependent rotation is then applied, wherein each sample from a particular class is rotated by either 0, $\frac{\pi}{2}$, π, or $\frac{3\pi}{2}$ (note: this class-specific rotation is randomized each episode, so a given class may experience different rotations from episode-to-episode). The image is then flattened into a vector, concatenated with a randomly chosen, episode-specific label, and fed as input to the network controller.

Class labels are randomly chosen for each class from episode-to-episode. For one-hot label experiments, labels are of size N, where N is the maximum number of unique classes that can appear in a given episode.



(a) String label encoded as five-hot vector



(b) Input Sequence

Figure 8. Example string label and input sequence.

8. Task

Either 5, 10, or 15 unique classes are chosen per episode. Episode lengths are ten times the number of unique classes (i.e., 50, 100, or 150 respectively), unless explicitly mentioned otherwise. Training occurs for 100 000 episodes. At the 100 000 episode mark, the task continues; however, data are pulled from a disjoint test set (i.e., samples from classes 1201-1623 in the omniglot dataset), and weight updates are ceased. This is deemed the "test phase."

For curriculum training, the maximum number of unique classes per episode increments by 1 every 10 000 training episodes. Accordingly, the episode length increases to 10 times this new maximum.

9. Parameters

9.0.1. Optimization

Rmsprop was used with a learning rate of $1\mathrm{e}^{-4}$ and max learning rate of $5\mathrm{e}^{-1}$, decay of 0.95 and momentum 0.9.

9.0.2. Free parameter grid search

A grid search was performed over number of parameters, with the values used shown in parentheses: memory slots (128), memory size (40), controller size (200 hidden units for a LSTM), learning rate ($1\mathrm{e}^{-4}$), and number of reads from memory (4). Other free parameters were left constant: usage decay of the write weights (0.99), minibatch size (16),

9.1. Comparisons and controls evaluation metrics

9.1.1. Human comparison

For the human comparison task, participants perform the exact same experiment as the network: they observe sequences of images and time-offset labels (sequence length = 50, number of unique classes = 5), and are challenged to predict the class identity for the current input image by inputting a single digit on a keypad. However, participants view class labels the integers 1 through 5, rather than one-hot vectors or strings. There is no time limit for their choice. Participants are made aware of the goals of the task prior to starting, and they perform a single, non-scored trial run prior to their scored trials. Nine participants each performed two scored trials.

9.1.2. kNN

When no data is available (i.e., at the start of training), the kNN classifier randomly returns a single class as its prediction. So, for the first data point, the probability that the prediction is correct is $\frac{1}{N}$ where N is number of unique classes in a given episode. Thereafter, it predicts a class from classes that it has observed. So, all instances of samples that are not members of the first observed class cannot be correctly classified until at least one instance is passed to the classifier. Since statistics are averaged across classes, first instance accuracy becomes $\frac{1}{N}(\frac{1}{N}+0)=\frac{1}{N^2}$, which is 4% and 0.4% for 5 and 15 classes per episode, respectively.