Adaptive Computation Time for Recurrent Neural Networks

Alex Graves
Google DeepMind
gravesa@google.com

Abstract

This paper introduces *Adaptive Computation Time* (ACT), an algorithm that allows recurrent neural networks to learn how many computational steps to take between receiving an input and emitting an output. ACT requires minimal changes to the network architecture, is deterministic and differentiable, and does not add any noise to the parameter gradients. Experimental results are provided for four synthetic problems: determining the parity of binary vectors, applying binary logic operations, adding integers, and sorting real numbers. Overall, performance is dramatically improved by the use of ACT, which successfully adapts the number of computational steps to the requirements of the problem. We also present character-level language modelling results on the Hutter prize Wikipedia dataset. In this case ACT does not yield large gains in performance; however it does provide intriguing insight into the structure of the data, with more computation allocated to harder-to-predict transitions, such as spaces between words and ends of sentences. This suggests that ACT or other adaptive computation methods could provide a generic method for inferring segment boundaries in sequence data.

1 Introduction

The amount of time required to pose a problem and the amount of thought required to solve it are notoriously unrelated. Pierre de Fermat was able to write in a margin the conjecture (if not the proof) of a theorem that took three and a half centuries and reams of mathematics to solve [35]. More mundanely, we expect the effort required to find a satisfactory route between two cities, or the number of queries needed to check a particular fact, to vary greatly, and unpredictably, from case to case. Most machine learning algorithms, however, are unable to dynamically adapt the amount of computation they employ to the complexity of the task they perform.

For artificial neural networks, where the neurons are typically arranged in densely connected layers, an obvious measure of computation time is the number of layer-to-layer transformations the network performs. In feedforward networks this is controlled by the network *depth*, or number of layers stacked on top of each other. For recurrent networks, the number of transformations also depends on the length of the input sequence — which can be padded or otherwise extended to allow for extra computation. The evidence that increased depth leads to more performant networks is by now inarguable [5, 4, 19, 9], and recent results show that increased sequence length can be similarly beneficial [31, 33, 25]. However it remains necessary for the experimenter to decide *a priori* on the amount of computation allocated to a particular input vector or sequence. One solution is to simply

make every network very deep and design its architecture in such a way as to mitigate the *vanishing gradient problem* [13] associated with long chains of iteration [29, 17]. However in the interests of both computational efficiency and ease of learning it seems preferable to dynamically vary the number of steps for which the network 'ponders' each input before emitting an output. In this case the effective depth of the network at each step along the sequence becomes a dynamic function of the inputs received so far.

The approach pursued here is to augment the network output with a sigmoidal *halting unit* whose activation determines the probability that computation should continue. The resulting *halting distribution* is used to define a mean-field vector for both the network output and the internal network state propagated along the sequence. A stochastic alternative would be to halt or continue according to binary samples drawn from the halting distribution—a technique that has recently been applied to scene understanding with recurrent networks [7]. However the mean-field approach has the advantage of using a smooth function of the outputs and states, with no need for stochastic gradient estimates. We expect this to be particularly beneficial when long sequences of halting decisions must be made, since each decision is likely to affect all subsequent ones, and sampling noise will rapidly accumulate (as observed for policy gradient methods [36]).

A related architecture known as *Self-Delimiting Neural Networks* [26, 30] employs a halting neuron to end a particular update within a large, partially activated network; in this case however a simple activation threshold is used to make the decision, and no gradient with respect to halting time is propagated. More broadly, learning when to halt can be seen as a form of *conditional computing*, where parts of the network are selectively enabled and disabled according to a learned policy [3, 6].

We would like the network to be parsimonious in its use of computation, ideally limiting itself to the minimum number of steps necessary to solve the problem. Finding this limit in its most general form would be equivalent to determining the Kolmogorov complexity of the data (and hence solving the halting problem) [21]. We therefore take the more pragmatic approach of adding a time cost to the loss function to encourage faster solutions. The network then has to learn to trade off accuracy against speed, just as a person must when making decisions under time pressure. One weakness is that the numerical weight assigned to the time cost has to be hand-chosen, and the behaviour of the network is quite sensitive to its value.

The rest of the paper is structured as follows: the Adaptive Computation Time algorithm is presented in Section 2, experimental results on four synthetic problems and one real-world dataset are reported in Section 3, and concluding remarks are given in Section 4.

2 Adaptive Computation Time

Consider a recurrent neural network $\mathcal{R}$ composed of a matrix of *input weights* W_x, a parametric *state transition model* $\mathcal{S}$, a set of *output weights* W_y and an *output bias* b_y. When applied to an input sequence $\mathbf{x} = (x_1, \ldots, x_T)$, $\mathcal{R}$ computes the *state sequence* $\mathbf{s} = (s_1, \ldots, s_T)$ and the output sequence $\mathbf{y} = (y_1, \ldots, y_T)$ by iterating the following equations from $t = 1$ to T:

$$s_t = \mathcal{S}(s_{t-1}, W_x x_t) \tag{1}$$

$$y_t = W_y s_t + b_y \tag{2}$$

The state is a fixed-size vector of real numbers containing the complete dynamic information of the network. For a standard recurrent network this is simply the vector of hidden unit activations. For a Long Short-Term Memory network (LSTM) [14], the state also contains the activations of the memory cells. For a memory augmented network such as a Neural Turing Machine (NTM) [10], the state contains both the complete state of the controller network and the complete state of the memory. In general some portions of the state (for example the NTM memory contents) will not be visible to the output units; in this case we consider the corresponding columns of W_y to be fixed to 0.

Adaptive Computation Time (ACT) modifies the conventional setup by allowing $\mathcal{R}$ to perform a variable number of state transitions and compute a variable number of outputs at each input step. Let $N(t)$ be the total number of updates performed at step t. Then define the *intermediate state sequence* $(s_t^1, \ldots, s_t^{N(t)})$ and *intermediate output sequence* $(y_t^1, \ldots, y_t^{N(t)})$ at step t as follows

$$s_t^n = \begin{cases} \mathcal{S}(s_{t-1}, x_t^1) \text{ if } n = 1 \\ \mathcal{S}(s_t^{n-1}, x_t^n) \text{ otherwise} \end{cases} \tag{3}$$

$$y_t^n = W_y s_t^n + b_y \tag{4}$$

where $x_t^n = x_t + \delta_{n,1}$ is the input at time t augmented with a binary flag that indicates whether the input step has just been incremented, allowing the network to distinguish between repeated inputs and repeated computations for the same input. Note that the same state function is used for all state transitions (intermediate or otherwise), and similarly the output weights and bias are shared for all outputs. It would also be possible to use different state and output parameters for each intermediate step; however doing so would cloud the distinction between increasing the number of parameters and increasing the number of computational steps. We leave this for future work.

To determine how many updates $\mathcal{R}$ performs at each input step an extra sigmoidal *halting* unit h is added to the network output, with associated weight matrix W_h and bias b_h:

$$h_t^n = \sigma\left(W_h s_t^n + b_h\right) \tag{5}$$

As with the output weights, some columns of W_h may be fixed to zero to give selective access to the network state. The activation of the halting unit is then used to determine the *halting probability* p_t^n of the intermediate steps:

$$p_t^n = \begin{cases} R(t) \text{ if } n = N(t) \\ h_t^n \text{ otherwise} \end{cases} \tag{6}$$

where

$$N(t) = \min\{n' : \sum_{n=1}^{n'} h_t^n >= 1 - \epsilon\} \tag{7}$$

the *remainder* $R(t)$ is defined as follows

$$R(t) = 1 - \sum_{n=1}^{N(t)-1} h_t^n \tag{8}$$

and ϵ is a small constant (0.01 for the experiments in this paper), whose purpose is to allow computation to halt after a single update if $h_t^1 >= 1 - \epsilon$, as otherwise a minimum of two updates would be required for every input step. It follows directly from the definition that $\sum_{n=1}^{N(t)} p_t^n = 1$ and $0 \leq p_t^n \leq 1 \; \forall n$, so this is a valid probability distribution. A similar distribution was recently used to define differentiable *push* and *pop* operations for neural stacks and queues [11].

At this point we could proceed stochastically by sampling $\hat{n}$ from p_t^n and setting $s_t = s_t^{\hat{n}}$, $y^t = y_t^{\hat{n}}$. However we will eschew sampling techniques and the associated problems of noisy gradients, instead using p_t^n to determine mean-field updates for the states and outputs:

$$s_t = \sum_{n=1}^{N(t)} p_t^n s_t^n \qquad y_t = \sum_{n=1}^{N(t)} p_t^n y_t^n \tag{9}$$

The implicit assumption is that the states and outputs are approximately linear, in the sense that a linear interpolation between a pair of state or output vectors will also interpolate between the

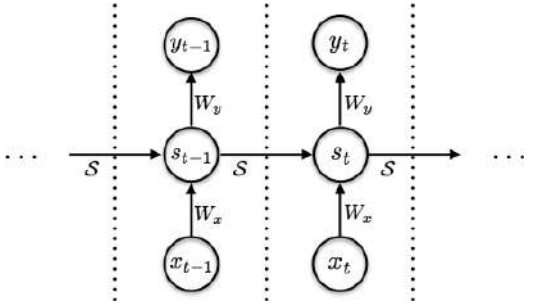

Figure 1: RNN Computation Graph. An RNN unrolled over two input steps (separated by vertical dotted lines). The input and output weights W_x, W_y, and the state transition operator $\mathcal{S}$ are shared over all steps.

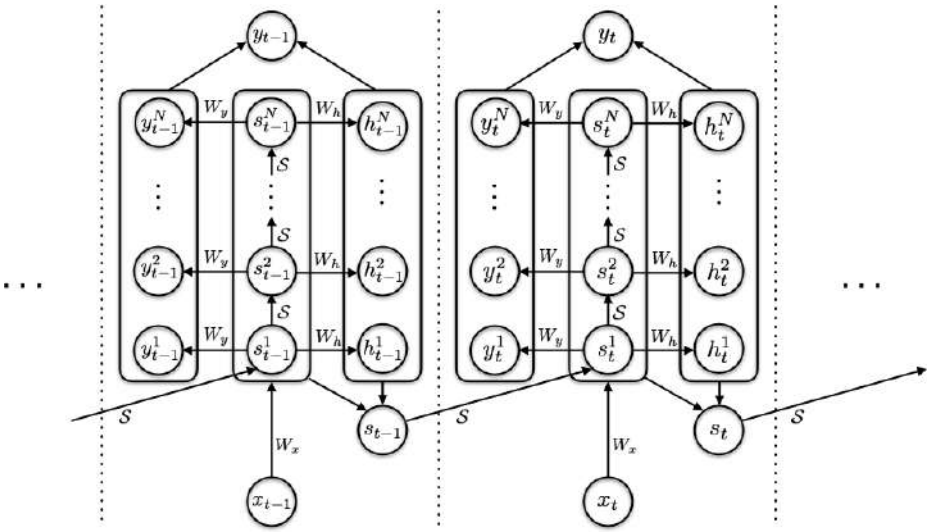

Figure 2: RNN Computation Graph with Adaptive Computation Time. The graph is equivalent to Figure 1, only with each state and output computation expanded to a variable number of intermediate updates. Arrows touching boxes denote operations applied to all units in the box, while arrows leaving boxes denote summations over all units in the box.

properties the vectors embody. There are several reasons to believe that such an assumption is reasonable. Firstly, it has been observed that the high-dimensional representations present in neural networks naturally tend to behave in a linear way [32, 20], even remaining consistent under arithmetic operations such as addition and subtraction [22]. Secondly, neural networks have been successfully trained under a wide range of adversarial regularisation constraints, including sparse internal states [23], stochastically masked units [28] and randomly perturbed weights [1]. This leads us to believe that the relatively benign constraint of approximately linear representations will not be too damaging. Thirdly, as training converges, the tendency for both mean-field and stochastic latent variables is to concentrate all the probability mass on a single value. In this case that yields a standard RNN with each input duplicated a variable, but deterministic, number of times, rendering the linearity assumption irrelevant.

A diagram of the unrolled computation graph of a standard RNN is illustrated in Figure 1, while Figure 2 provides the equivalent diagram for an RNN trained with ACT.

2.1 Limiting Computation Time

If no constraints are placed on the number of updates $\mathcal{R}$ can take at each step it will naturally tend to 'ponder' each input for as long as possible (so as to avoid making predictions and incurring errors). We therefore require a way of limiting the amount of computation the network performs. Given a length T input sequence $\mathbf{x}$, define the *ponder sequence* $(\rho_1, \dots, \rho_T)$ of $\mathcal{R}$ as

$$\rho_t = N(t) + R(t) \tag{10}$$

and the *ponder cost* $\mathcal{P}(\mathbf{x})$ as

$$\mathcal{P}(\mathbf{x}) = \sum_{t=1}^{T} \rho_t \tag{11}$$

Since $R(t) \in (0, 1)$, $\mathcal{P}(\mathbf{x})$ is an upper bound on the (non-differentiable) property we ultimately want to reduce, namely the total computation $\sum_{t=1}^{T} N(t)$ during the sequence[1].

We can encourage the network to minimise $\mathcal{P}(\mathbf{x})$ by modifying the sequence loss function $\mathcal{L}(\mathbf{x}, \mathbf{y})$ used for training:

$$\hat{\mathcal{L}}(\mathbf{x}, \mathbf{y}) = \mathcal{L}(\mathbf{x}, \mathbf{y}) + \tau \mathcal{P}(\mathbf{x}) \tag{12}$$

where τ is a *time penalty* parameter that weights the relative cost of computation versus error. As we will see in the experiments section the behaviour of the network is quite sensitive to the value of τ, and it is not obvious how to choose a good value. If computation time and prediction error can be meaningfully equated (for example if the relative financial cost of both were known) a more principled technique for selecting τ should be possible.

To prevent very long sequences at the beginning of training (while the network is learning how to use the halting unit) the bias term b_h can be initialised to a positive value. In addition, a hard limit M on the maximum allowed value of $N(t)$ can be imposed to avoid excessive space and time costs. In this case Equation (7) is modified to

$$N(t) = \min\{M, \min\{n' : \sum_{n=1}^{n'} h_t^n >= 1 - \epsilon\}\} \tag{13}$$

2.2 Error Gradients

The ponder costs ρ_t are discontinuous with respect to the halting probabilities at the points where $N(t)$ increments or decrements (that is, when the summed probability mass up to some n either decreases below or increases above $1 - \epsilon$). However they are continuous away from those points, as $N(t)$ remains constant and $R(t)$ is a linear function of the probabilities. In practice we simply ignore the discontinuities by treating $N(t)$ as constant and minimising $R(t)$ everywhere.

Given this approximation, the gradient of the ponder cost with respect to the halting activations is straightforward:

$$\frac{\partial \mathcal{P}(\mathbf{x})}{\partial h_t^n} = \begin{cases} 0 \text{ if } n = N(t) \\ -1 \text{ otherwise} \end{cases} \tag{14}$$

[1] For a stochastic ACT network, a more natural halting distribution than the one described in Equations (6) to (8) would be to simply treat h_t^n as the probability of halting at step n, in which case $p_t^n = h_t^n \prod_{n'=1}^{n-1} (1 - h_t^{n'})$. One could then set $\rho_t = \sum_{n=1}^{N(t)} n p_t^n$ — i.e. the expected ponder time under the stochastic distribution. However experiments show that networks trained to minimise *expected* rather than *total* halting time learn to 'cheat' in the following ingenious way: they set h_t^1 to a value just below the halting threshold, then keep $h_t^n = 0$ until some $N(t)$ when they set $h_t^{N(t)}$ high enough to ensure they halt. In this case $p_t^{N(t)} \ll p_t^1$, so the states and outputs at $n = N(t)$ have much lower weight in the mean field updates (Equation (9)) than those at $n = 1$; however by making the magnitudes of the states and output vectors much larger at $N(t)$ than $n = 1$ the network can still ensure that the update is dominated by the final vectors, despite having paid a low ponder penalty.

and hence

$$\frac{\partial \hat{\mathcal{L}}(\mathbf{x},\mathbf{y})}{\partial h_t^n} = \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial h_t^n} - \begin{cases} 0 \text{ if } n = N(t) \\ \tau \text{ otherwise} \end{cases} \tag{15}$$

The halting activations only influence $\mathcal{L}$ via their effect on the halting probabilities, therefore

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial h_t^n} = \sum_{n'=1}^{N(t)} \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial p_t^{n'}} \frac{\partial p_t^{n'}}{\partial h_t^n} \tag{16}$$

Furthermore, since the halting probabilities only influence $\mathcal{L}$ via their effect on the states and outputs, it follows from Equation (9) that

$$\frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial p_t^n} = \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial y_t} y_t^n + \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial s_t} s_t^n \tag{17}$$

while, from Equations (6) and (8)

$$\frac{\partial p_t^{n'}}{\partial h_t^n} = \begin{cases} \delta_{n,n'} \text{ if } n' < N(t) \text{ and } n < N(t) \\ -1 \text{ if } n' = N(t) \text{ and } n < N(t) \\ 0 \text{ if } n = N(t) \end{cases} \tag{18}$$

Combining Equations (15), (17) and (18) gives, for $n < N(t)$

$$\frac{\partial \hat{\mathcal{L}}(\mathbf{x},\mathbf{y})}{\partial h_t^n} = \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial y_t} \left(y_t^n - y_t^{N(t)} \right) + \frac{\partial \mathcal{L}(\mathbf{x},\mathbf{y})}{\partial s_t} \left(s_t^n - s_t^{N(t)} \right) - \tau \tag{19}$$

while for $n = N(t)$

$$\frac{\partial \hat{\mathcal{L}}(\mathbf{x},\mathbf{y})}{\partial h_t^{N(t)}} = 0 \tag{20}$$

Thereafter the network can be differentiated as usual (e.g. with backpropagation through time [36]) and trained with gradient descent.

3 Experiments

We tested recurrent neural networks (RNNs) with and without ACT on four synthetic tasks and one real-world language processing task. LSTM was used as the network architecture for all experiments except one, where a simple RNN was used. However we stress that ACT is equally applicable to any recurrent architecture.

All the tasks were supervised learning problems with discrete targets and cross-entropy loss. The data for the synthetic tasks was generated online and cross-validation was therefore not needed. Similarly, the character prediction dataset was sufficiently large that the network did not overfit. The performance metric for the synthetic tasks was the *sequence error rate*: the fraction of examples where *any* mistakes were made in the complete output sequence. This metric is useful as it is trivial to evaluate without decoding. For character prediction the metric was the average log-loss of the output predictions, in units of bits per character.

Most of the training parameters were fixed for all experiments: Adam [18] was used for optimisation with a learning rate of 10^{-4}, the Hogwild! algorithm [24] was used for asynchronous training with 16 threads; the initial halting unit bias b_h mentioned in Equation (5) was 1; the ϵ term from Equation (7) was 0.01. The synthetic tasks were all trained for 1M iterations, where an iteration

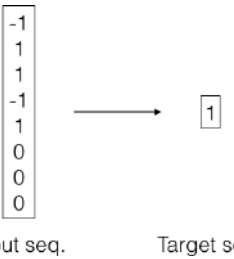

Figure 3: Parity training Example. Each sequence consists of a single input and target vector. Only 8 of the 64 input bits are shown for clarity.

is defined as a weight update on a single thread (hence the total number of weight updates is approximately 16 times the number of iterations). The character prediction task was trained for 10K iterations. Early stopping was not used for any of the experiments.

A logarithmic grid search over time penalties was performed for each experiment, with 20 randomly initialised networks trained for each value of τ. For the synthetic problems the range of the grid search was from $i \times 10^{-j}$ with integer i in the range 1–10 and the exponent j in the range 1–4. For the language modelling task, which took many days to complete, the range of j was limited to 1–3 to reduce training time (lower values of τ, which naturally induce more pondering, tend to give greater data efficiency but slower wall clock training time).

Unless otherwise stated the maximum computation time M (Equation (13)) was set to 100. In all experiments the networks converged on learned values of $N(t)$ that were far less than M, which functions mainly as safeguard against excessively long ponder times early in training.

3.1 Parity

Determining the parity of a sequence of binary numbers is a trivial task for a recurrent neural network [27], which simply needs to implement an internal switch that changes sign every time a one is received. For shallow feedforward networks receiving the entire sequence in one vector, however, the number of distinct input patterns, and hence difficulty of the task, grows exponentially with the number of bits. We gauged the ability of ACT to infer an inherently sequential algorithm from statically presented data by presenting large binary vectors to the network and asking it to determine the parity. By varying the number of binary bits for which parity must be calculated we were also able to assess ACT's ability to adapt the amount of computation to the difficulty of the vector.

The input vectors had 64 elements, of which a random number from 1 to 64 were randomly set to 1 or -1 and the rest were set to 0. The corresponding target was 1 if there was an odd number of ones and 0 if there was an even number of ones. Each training sequence consisted of a single input and target vector, an example of which is shown in Figure 3. The network architecture was a simple RNN with a single hidden layer containing 128 *tanh* units and a single sigmoidal output unit, trained with binary cross-entropy loss on minibatches of size 128. Note that without ACT the recurrent connection in the hidden layer was never used since the data had no sequential component, and the network reduced to a feedforward network with a single hidden layer.

Figure 4 demonstrates that the network was unable to reliably solve the problem without ACT, with a mean of almost 40% error compared to 50% for random guessing. For penalties of 0.03 and below the mean error was below 5%. Figure 5 reveals that the solutions were both more rapid and more accurate with lower time penalties. It also highlights the relationship between the time penalty, the classification error rate and the average ponder time per input. The variance in ponder time for low τ networks is very high, indicating that many correct solutions with widely varying runtime can be discovered. We speculate that progressively higher τ values lead the network to compute

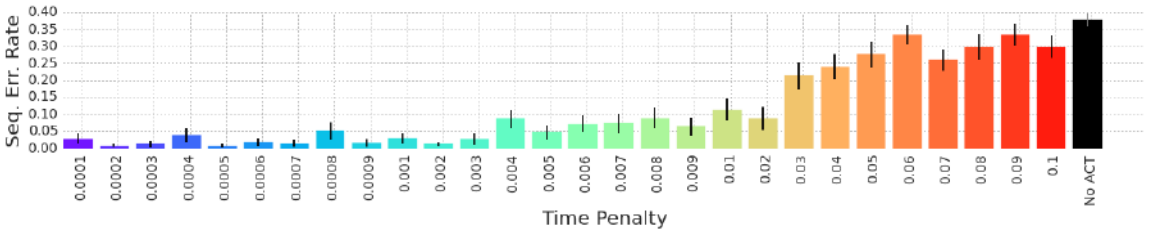

Figure 4: Parity Error Rates. Bar heights show the mean error rates for different time penalties at the end of training. The error bars show the standard error in the mean.

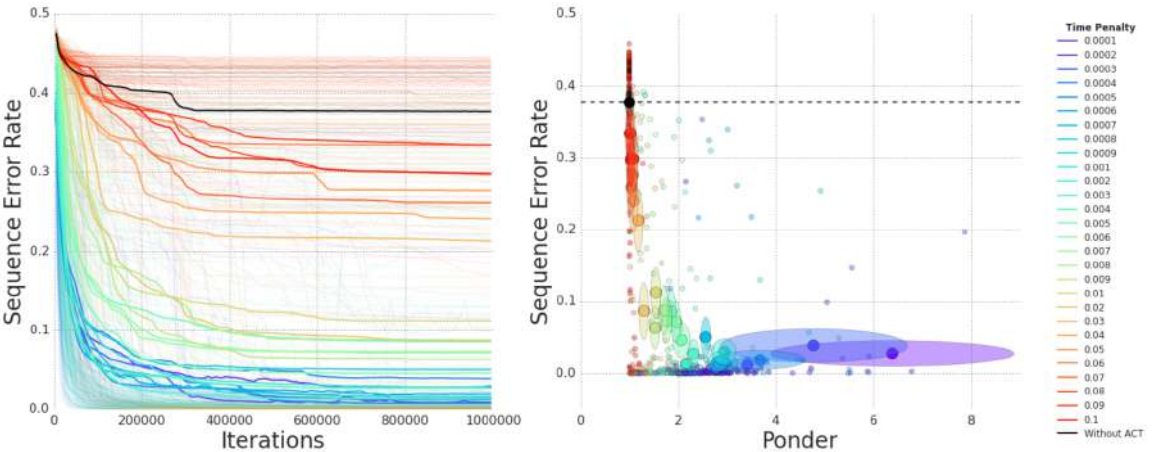

Figure 5: Parity Learning Curves and Error Rates Versus Ponder Time. Left: faint coloured curves show the errors for individual runs. Bold lines show the mean errors over all 20 runs for each τ value. 'Iterations' is the number of gradient updates per asynchronous worker. **Right:** Small circles represent individual runs after training is complete, large circles represent the mean over 20 runs for each τ value. 'Ponder' is the mean number of computation steps per input timestep (minimum 1). The black dotted line shows the mean error for the networks without ACT. The height of the ellipses surrounding the mean values represents the standard error over error rates for that value of τ, while the width shows the standard error over ponder times.

the parities of successively larger chunks of the input vector at each ponder step, then iteratively combine these calculations to obtain the parity of the complete vector.

Figure 6 shows that for the networks without ACT and those with overly high time penalties, the error rate increases sharply with the difficulty of the task (where *difficulty* is defined as the number of bits whose parity must be determined), while the amount of ponder remains roughly constant. For the more successful networks, with intermediate τ values, ponder time appears to grow linearly with difficulty, with a slope that generally increases as τ decreases. Even for the best networks the error rate increased somewhat with difficulty. For some of the lowest τ networks there is a dramatic increase in ponder after about 32 bits, suggesting an inefficient algorithm.

3.2 Logic

Like parity, the *logic* task tests if an RNN with ACT can sequentially process a static vector. Unlike parity it also requires the network to internally transfer information across successive input timesteps, thereby testing whether ACT can propagate coherent internal states.

Each input sequence consists of a random number from 1 to 10 of size 102 input vectors. The first two elements of each input represent a pair of binary numbers; the remainder of the vector is divided up into 10 chunks of size 10. The first B chunks, where B is a random number from

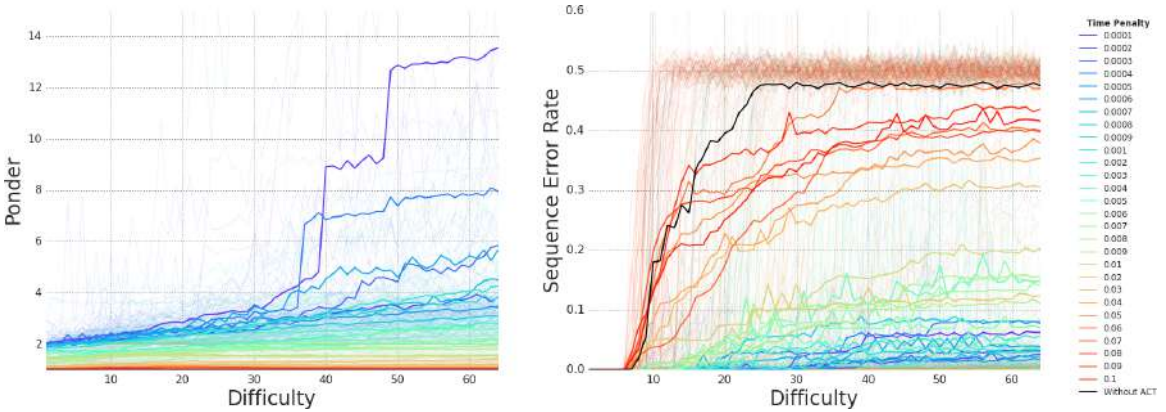

Figure 6: Parity Ponder Time and Error Rate Versus Input Difficulty. Faint lines are individual runs, bold lines are means over 20 networks. 'Difficulty' is the number of bits in the parity vectors, with a mean over 1,000 random vectors used for each data-point.

Table 1: Binary Truth Tables for the Logic Task

P	Q	NOR	Xq	ABJ	XOR	NAND	AND	XNOR	if/then	then/if	OR
T	**T**	F	F	F	F	F	T	T	T	T	T
T	**F**	F	F	T	T	T	F	F	F	T	T
F	**T**	F	T	F	T	T	F	F	T	F	T
F	**F**	T	F	F	F	T	F	T	T	T	F

1 to 10, contain one-hot representations of randomly chosen numbers between 1 and 10; each of these numbers correspond to an index into the subset of binary logic gates whose truth tables are listed in Table 1. The remaining $10 - B$ chunks were zeroed to indicate that no further binary operations were defined for that vector. The binary target b_{B+1} for each input is the truth value yielded by recursively applying the B binary gates in the vector to the two initial bits b_1, b_0. That is for $1 \leq b \leq B$:

$$b_{i+1} = T_i(b_i, b_{i-1}) \tag{21}$$

where $T_i(.,.)$ is the truth table indexed by chunk i in the input vector.

For the first vector in the sequence, the two input bits b_0, b_1 were randomly chosen to be false (0) or true (1) and assigned to the first two elements in the vector. For subsequent vectors, only b_1 was random, while b_0 was implicitly equal to the target bit from the previous vector (for the purposes of calculating the current target bit), but was always set to zero in the input vector. To solve the task, the network therefore had to learn both how to calculate the sequence of binary operations represented by the chunks in each vector, and how to carry the final output of that sequence over to the next timestep. An example input-target sequence pair is shown in Figure 7.

The network architecture was single-layer LSTM with 128 cells. The output was a single sigmoidal unit, trained with binary cross-entropy, and the minibatch size was 16.

Figure 8 shows that the network reaches a minimum sequence error rate of around 0.2 without ACT (compared to 0.5 for random guessing), and virtually zero error for all $\tau \leq 0.01$. From Figure 9 it can be seen that low τ ACT networks solve the task very quickly, requiring about 10,000 training iterations. For higher τ values ponder time reduces to 1, at which point the networks trained with ACT behave identically to those without. For lower τ values, the spread of ponder values, and hence computational cost, is quite large. Again we speculate that this is due to the network learning more or less 'chunked' solutions in which composite truth table are learned for multiple successive logic operations. This is somewhat supported by the clustering of the lowest τ networks around a ponder time of 5–6, which is approximately the mean number of logic gates applied per sequence,

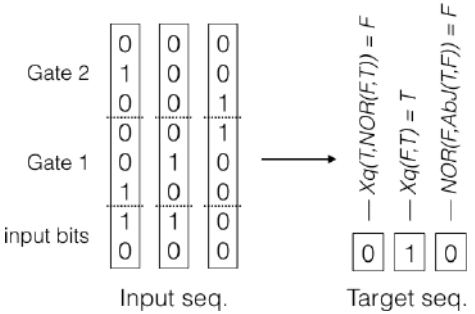

Figure 7: Logic training Example. Both the input and target sequences consist of 3 vectors. For simplicity only 2 of the 10 possible logic gates represented in the input are shown, and each is restricted to one of the first 3 gates in Table 1 (NOR, Xq, and ABJ). The segmentation of the input vectors is show on the left and the recursive application of Equation (21) required to determine the targets (and subsequent b_0 values) is shown in italics above the target vectors.

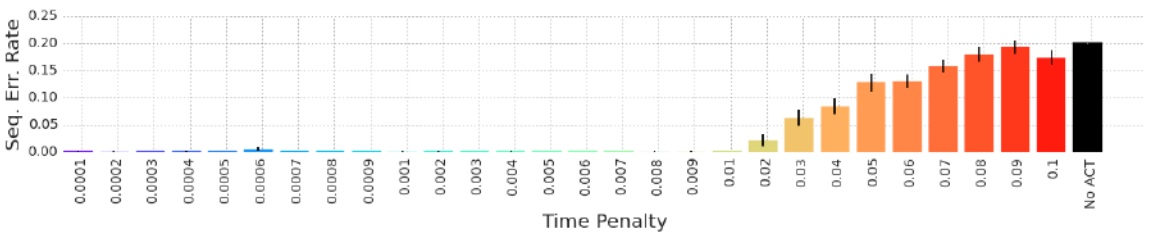

Figure 8: Logic Error Rates.

and hence the minimum number of computations the network would need if calculating single binary operations at a time.

Figure 10 shows a surprisingly high ponder time for the least difficult inputs, with some networks taking more than 10 steps to evaluate a single logic gate. From 5 to 10 logic gates, ponder gradually increases with difficulty as expected, suggesting that a qualitatively different solution is learned for the two regimes. This is supported by the error rates for the non ACT and high τ networks, which increase abruptly after 5 gates. It may be that 5 is the upper limit on the number of successive gates the network can learn as a single composite operation, and thereafter it is forced to apply an iterative algorithm.

3.3 Addition

The addition task presents the network with a input sequence of 1 to 5 size 50 input vectors. Each vector represents a D digit number, where D is drawn randomly from 1 to 5, and each digit is drawn randomly from 0 to 9. The first $10D$ elements of the vector are a concatenation of one-hot encodings of the D digits in the number, and the remainder of the vector is set to 0. The required output is the cumulative sum of all inputs up to the current one, represented as a set of 6 simultaneous classifications for the 6 possible digits in the sum. There is no target for the first vector in the sequence, as no sums have yet been calculated. Because the previous sum must be carried over by the network, this task again requires the internal state of the network to remain coherent. Each classification is modelled by a size 11 softmax, where the first 10 classes are the digits and the 11^{th} is a special marker used to indicate that the number is complete. An example input-target pair is shown in Figure 11.

The network was single-layer LSTM with 512 memory cells. The loss function was the joint cross-entropy of all 6 targets at each time-step where targets were present and the minibatch size

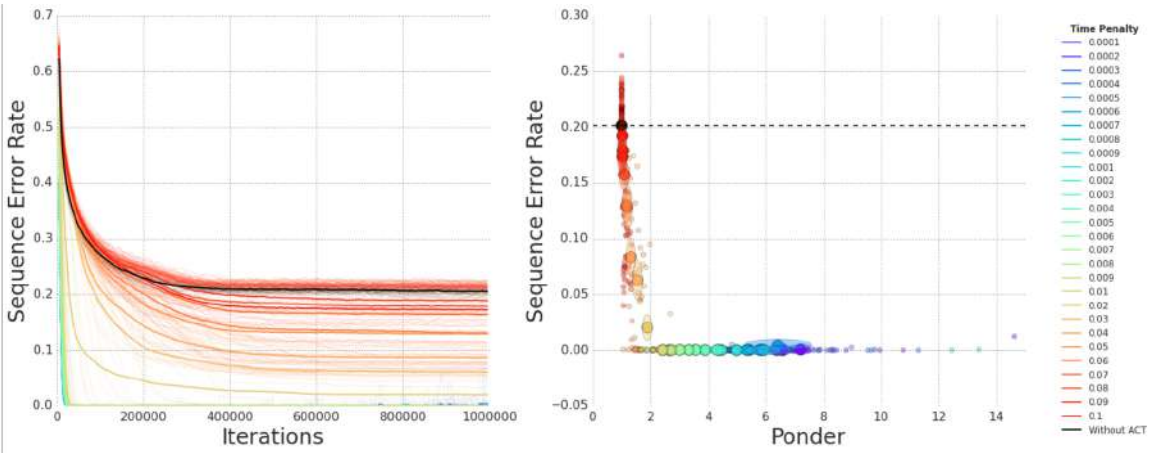

Figure 9: Logic Learning Curves and Error Rates Versus Ponder Time.

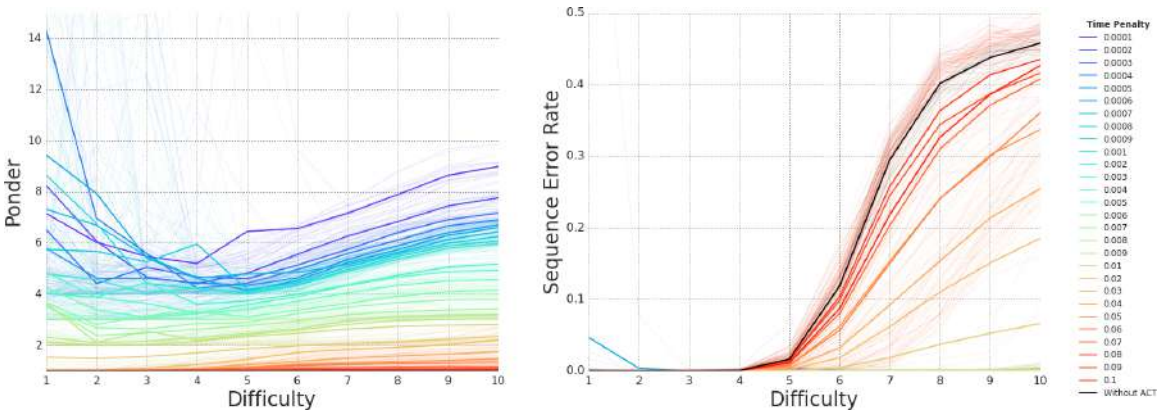

Figure 10: Logic Ponder Time and Error Rate Versus Input Difficulty. 'Difficulty' is the number of logic gates in each input vector; all sequences were length 5.

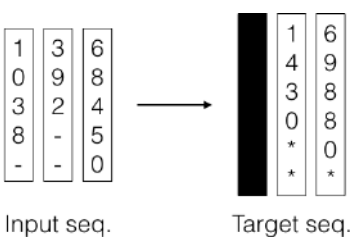

Figure 11: Addition training Example. Each digit in the input sequence is represented by a size 10 one hot encoding. Unused input digits, marked '-', are represented by a vector of 10 zeros. The black vector at the start of the target sequence indicates that no target was required for that step. The target digits are represented as 1-of-11 classes, where the 11^th class, marked '*', is used for digits beyond the end of the target number.

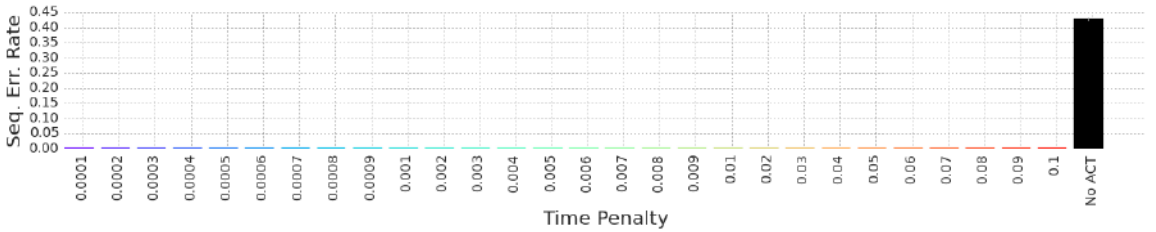

Figure 12: Addition Error Rates.

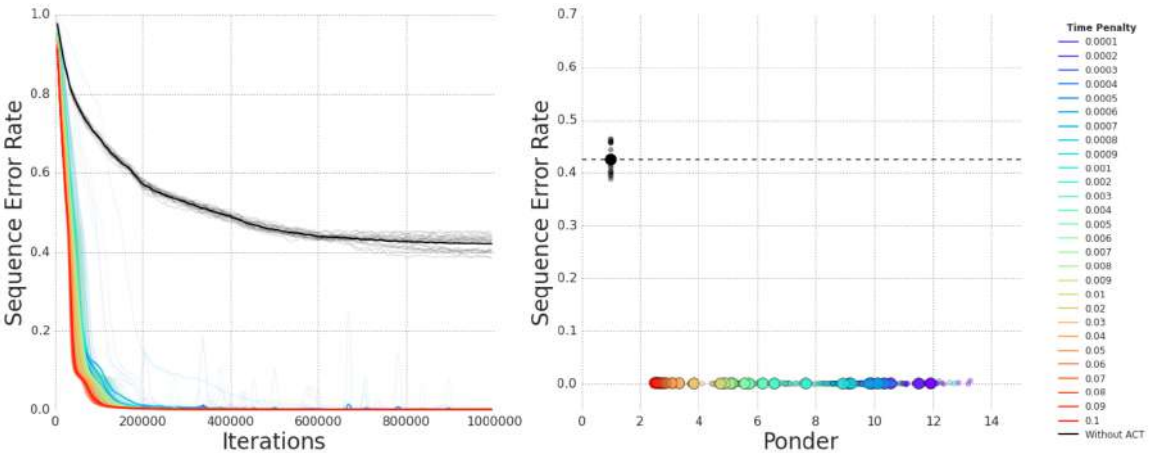

Figure 13: Addition Learning Curves and Error Rates Versus Ponder Time.

was 32. The maximum ponder M was set to 20 for this task, as it was found that some networks had very high ponder times early in training.

The results in Figure 12 show that the task was perfectly solved by the ACT networks for all values of τ in the grid search. Unusually, networks with higher τ solved the problem with fewer training examples. Figure 14 demonstrates that the relationship between the ponder time and the number of digits was approximately linear for most of the ACT networks, and that for the most efficient networks (with the highest τ values) the slope of the line was close to 1, which matches our expectations that an efficient long addition algorithm should need one computation step per digit.

Figure 15 shows how the ponder time is distributed during individual addition sequences, providing further evidence of an approximately linear-time long addition algorithm.

3.4 Sort

The *sort* task requires the network to sort sequences of 2 to 15 numbers drawn from a standard normal distribution in ascending order. The experiments considered so far have been designed to favour ACT by compressing sequential information into single vectors, and thereby requiring the use of multiple computation steps to unpack them. For the sort task a more natural sequential representation was used: the random numbers were presented one at a time as inputs, and the required output was the sequence of indices into the number sequence placed in sorted order; an example is shown in Figure 16. We were particularly curious to see how the number of ponder steps scaled with the number of elements to be sorted, knowing that efficient sorting algorithms have $O(N \log N)$ computational cost.

The network was single-layer LSTM with 512 cells. The output layer was a size 15 softmax,

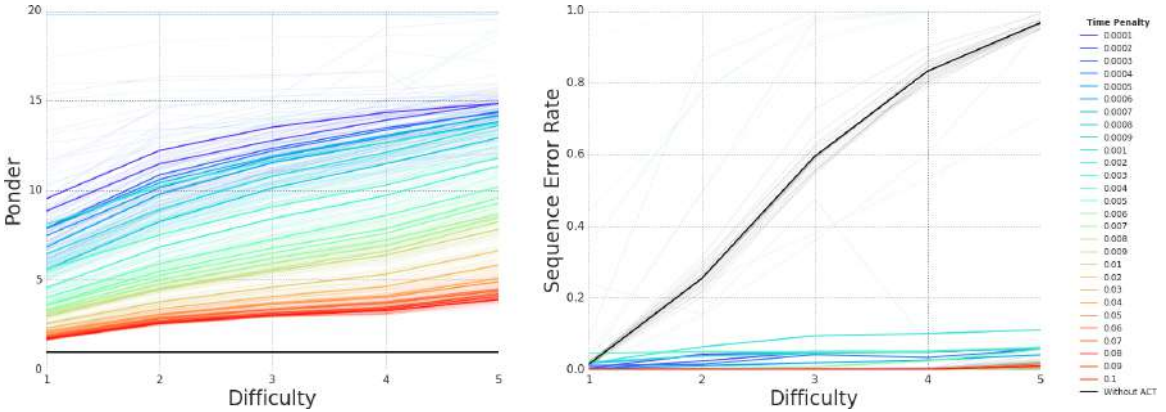

Figure 14: Addition Ponder Time and Error Rate Versus Input Difficulty. 'Difficulty' is the number of digits in each input vector; all sequences were length 3.

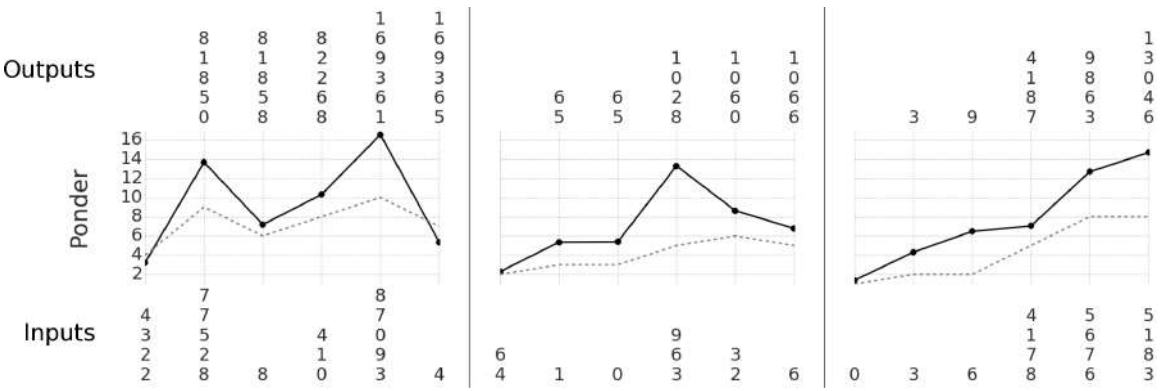

Figure 15: Ponder Time During Three Addition Sequences. The input sequence is shown along the bottom x-axis and the network output sequence is shown along the top x-axis. The ponder time ρ_t at each input step is shown by the black lines; the actual number of computational steps taken at each point is ρ_t rounded up to the next integer. The grey lines show the total number of digits in the two numbers being summed at each step; this appears to give a rough lower bound on the ponder time, suggesting an internal algorithm that is approximately linear in the number of digits. All plots were created using the same network, trained with $\tau = 9e^{-4}$.

trained with cross-entropy to classify the indices of the sorted inputs. The minibatch size was 16.

Figure 17 shows that the advantage of using ACT is less dramatic for this task than the previous three, but still substantial (from around 12% error without ACT to around 6% for the best τ value). However from Figure 18 it is clear that these gains come at a heavy computational cost, with the best networks requiring roughly 9 times as much computation as those without ACT. Not surprisingly, Figure 19 shows that the error rate grew rapidly with the sequence length for all networks. It also indicates that the better networks had a sublinear growth in computations per input step with sequence length, though whether this indicates a logarithmic time algorithm is unclear. One problem with the sort task was that the Gaussian samples were sometimes very close together, making it hard for the network to determine which was greater; enforcing a minimum separation between successive values would probably be beneficial.

Figure 20 shows the ponder time during three sort sequences of varying length. As can be seen, there is a large spike in ponder time near (though not precisely at) the end of the input sequence, presumably when the majority of the sort comparisons take place. Note that the spike is much higher for the longer two sequences than the length 5 one, again pointing to an algorithm that is nonlinear

Figure 16: Sort training Example. Each size 2 input vector consists of one real number and one binary flag to indicate the end of sequence to be sorted; inputs following the sort sequence are set to zero and marked in black. No targets are present until after the sort sequence; thereafter the size 15 target vectors represent the sorted indices of the input sequence.

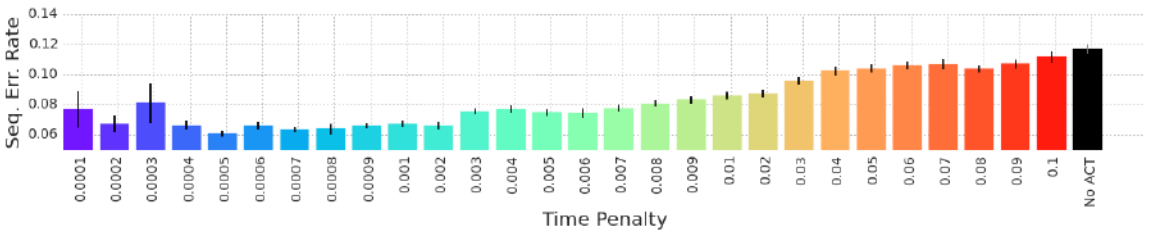

Figure 17: Sort Error Rates.

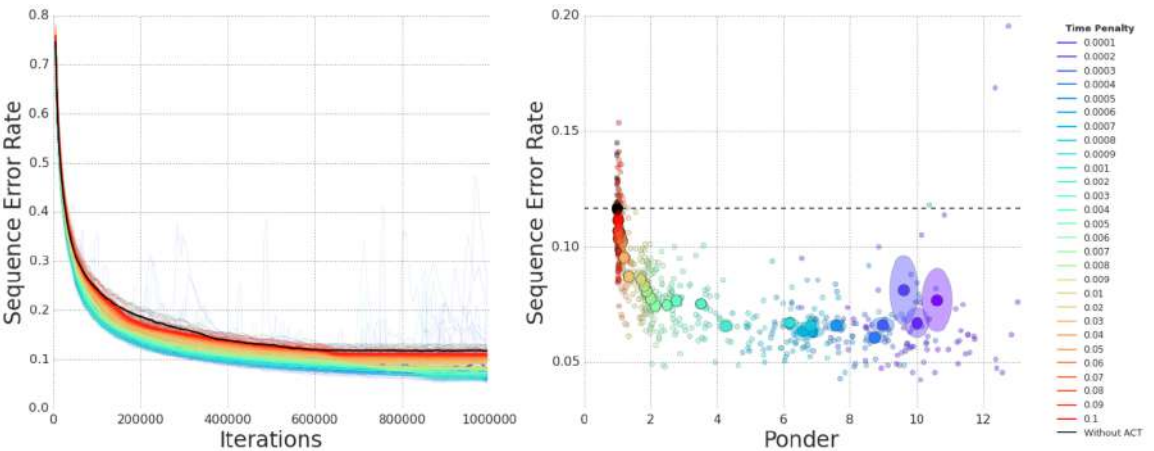

Figure 18: Sort Learning Curves and Error Rates Versus Ponder Time.

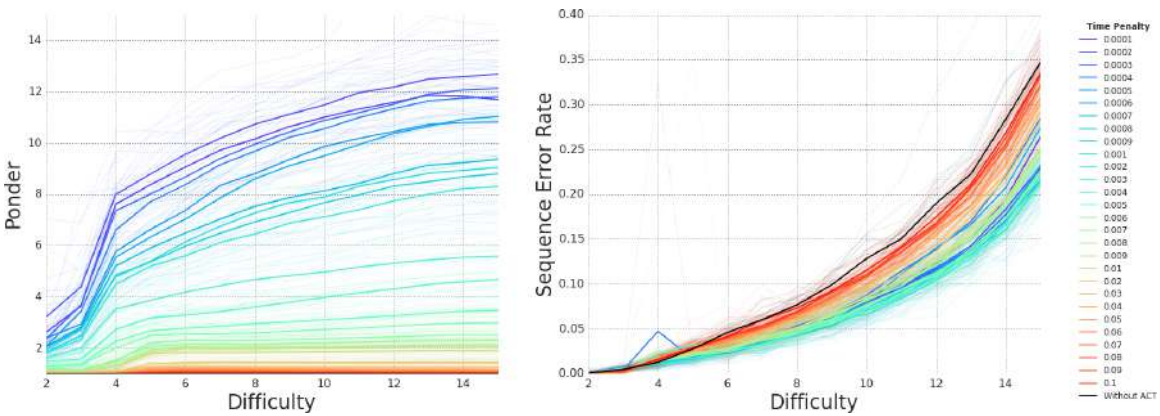

Figure 19: Sort Ponder Time and Error Rate Versus Input Difficulty. 'Difficulty' is the length of the sequence to be sorted.

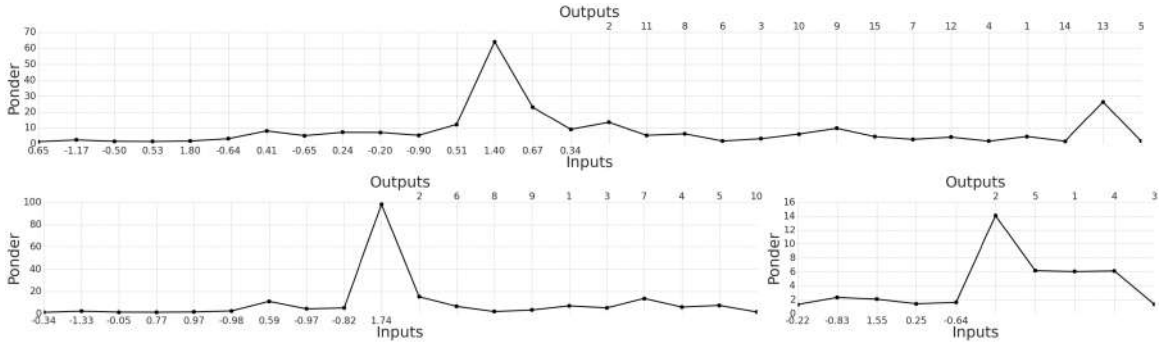

Figure 20: Ponder Time During Three Sort Sequences. The input sequences to be sorted are shown along the bottom x-axes and the network output sequences are shown along the top x-axes. All plots created using the same network, trained with $\tau = 10^{-3}$.

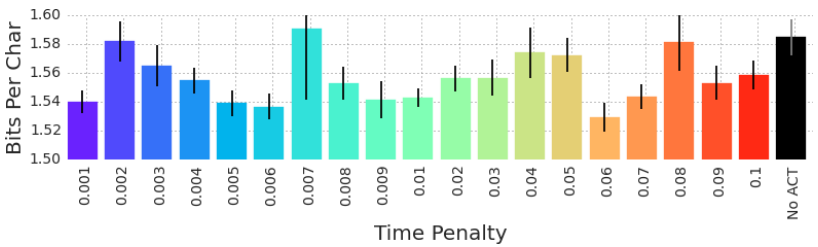

Figure 21: Wikipedia Error Rates.

in sequence length (the average ponder per timestep is nonetheless lower for longer sequences, as little pondering is done away from the spike.).

3.5 Wikipedia Character Prediction

The *Wikipedia* task is character prediction on text drawn from the Hutter prize Wikipedia dataset [15]. Following previous RNN experiments on the same data [8], the raw unicode text was used, including XML tags and markup characters, with one byte presented per input timestep and the next byte predicted as a target. No validation set was used for early stopping, as the networks were unable to overfit the data, and all error rates are recorded on the training set. Sequences of 500 consecutive bytes were randomly chosen from the training set and presented to the network, whose internal state was reset to 0 at the start of each sequence.

LSTM networks were used with a single layer of 1500 cells and a size 256 softmax classification layer. As can be seen from Figures 21 and 22, the error rates are fairly similar with and without ACT, and across values of τ (although the learning curves suggest that the ACT networks are somewhat more data efficient). Furthermore the amount of ponder per input is much lower than for the other problems, suggesting that the advantages of extra computation were slight for this task.

However Figure 23 reveals an intriguing pattern of ponder allocation while processing a sequence. Character prediction networks trained with ACT consistently pause at spaces between words, and pause for longer at 'boundary' characters such as commas and full stops. We speculate that the extra computation is used to make predictions about the next 'chunk' in the data (word, sentence, clause), much as humans have been found to do in self-paced reading experiments [16]. This suggests that ACT could be useful for inferring implicit boundaries or transitions in sequence data. Alternative measures for inferring transitions include the next-step prediction loss and predictive entropy, both of which tend to increase during harder predictions. However, as can be seen from the figure, they

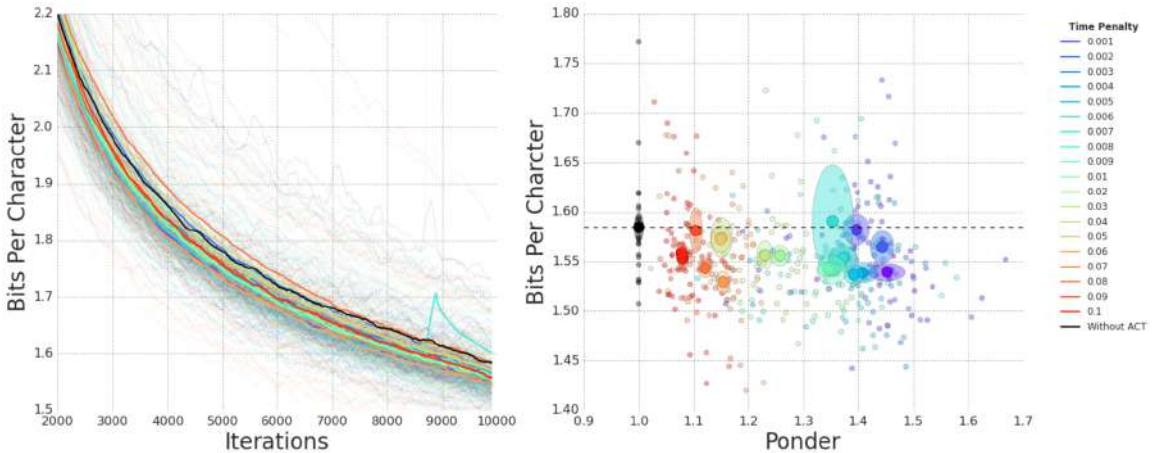

Figure 22: Wikipedia Learning Curves (Zoomed) and Error Rates Versus Ponder Time.

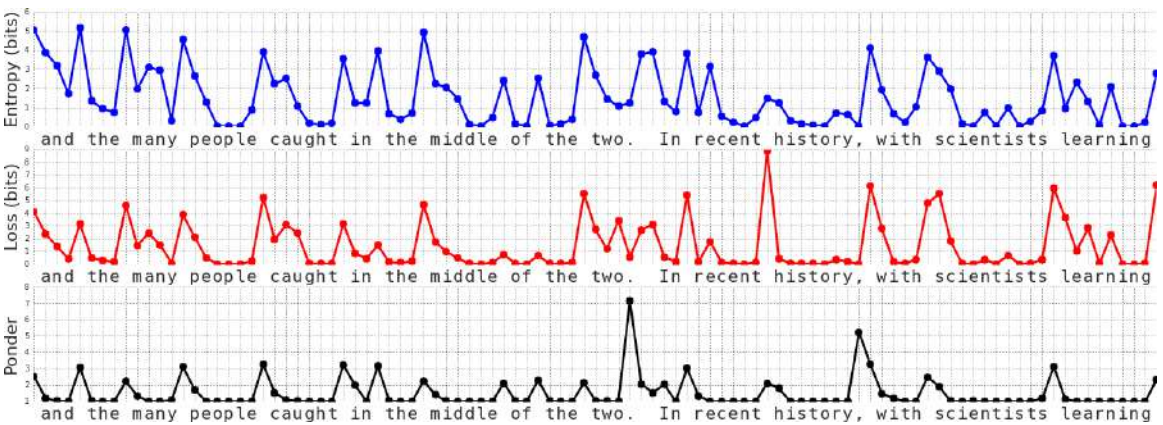

Figure 23: Ponder Time, Prediction loss and Prediction Entropy During a Wikipedia Text Sequence. Plot created using a network trained with $\tau = 6e^{-3}$

are a less reliable indicator of boundaries, and are not likely to increase at points such as full stops and commas, as these are invariably followed by space characters. More generally, loss and entropy only indicate the difficulty of the current prediction, not the degree to which the current input is likely to impact future predictions.

Furthermore Figure 24 reveals that, as well as being an effective detector of non-text transition markers such as the opening brackets of XML tags, ACT does not increase computation time during random or fundamentally unpredictable sequences like the two ID numbers. This is unsurprising, as doing so will not improve its predictions. In contrast, both entropy and loss are inevitably high for unpredictable data. We are therefore hopeful that computation time will provide a better way to distinguish between structure and noise (or at least data perceived by the network as structure or noise) than existing measures of predictive difficulty.

4 Conclusion

This paper has introduced Adaptive Computation time (ACT), a method that allows recurrent neural networks to learn how many updates to perform for each input they receive. Experiments on

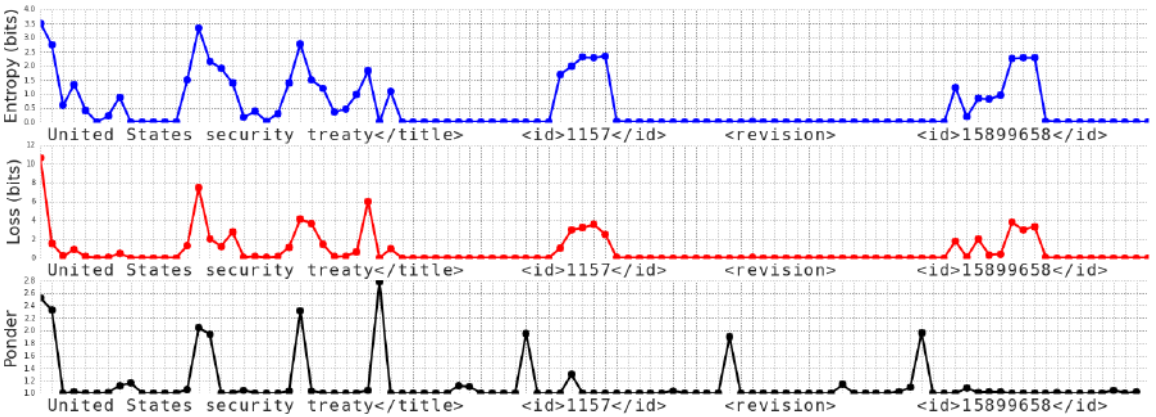

Figure 24: Ponder Time, Prediction loss and Prediction Entropy During a Wikipedia Sequence Containing XML Tags. Created using the same network as Figure 23.

synthetic data prove that ACT can make otherwise inaccessible problems straightforward for RNNs to learn, and that it is able to dynamically adapt the amount of computation it uses to the demands of the data. An experiment on real data suggests that the allocation of computation steps learned by ACT can yield insight into both the structure of the data and the computational demands of predicting it.

ACT promises to be particularly interesting for recurrent architectures containing soft attention modules [2, 10, 34, 12], which it could enable to dynamically adapt the number of glances or internal operations they perform at each time-step.

One weakness of the current algorithm is that it is quite sensitive to the time penalty parameter that controls the relative cost of computation time versus prediction error. An important direction for future work will be to find ways of automatically determining and adapting the trade-off between accuracy and speed.

Acknowledgments

The author wishes to thank Ivo Danihleka, Greg Wayne, Tim Harley, Malcolm Reynolds, Jacob Menick, Oriol Vinyals, Joel Leibo, Koray Kavukcuoglu and many others on the DeepMind team for valuable comments and suggestions, as well as Albert Zeyer, Martin Abadi, Dario Amodei, Eugene Brevdo and Christopher Olah for pointing out the discontinuity in the ponder cost, which was erroneously described as smooth in an earlier version of the paper.

References

[1] G. An. The effects of adding noise during backpropagation training on a generalization performance. *Neural Computation*, 8(3):643–674, 1996.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. abs/1409.0473, 2014.

[3] E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.

[4] D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *arXiv:1202.2745v1 [cs.CV]*, 2012.

[5] G. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30 –42, jan. 2012.

[6] L. Denoyer and P. Gallinari. Deep sequential neural network. *arXiv preprint arXiv:1410.0510*, 2014.

[7] S. Eslami, N. Heess, T. Weber, Y. Tassa, K. Kavukcuoglu, and G. E. Hinton. Attend, infer, repeat: Fast scene understanding with generative models. *arXiv preprint arXiv:1603.08575*, 2016.

[8] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[9] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

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

[11] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1819–1827, 2015.

[12] K. Gregor, I. Danihelka, A. Graves, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[13] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

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

[15] M. Hutter. *Universal artificial intelligence*. Springer, 2005.

[16] M. A. Just, P. A. Carpenter, and J. D. Woolley. Paradigms and processes in reading comprehension. *Journal of experimental psychology: General*, 111(2):228, 1982.

[17] N. Kalchbrenner, I. Danihelka, and A. Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

[18] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014.

[21] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[23] B. A. Olshausen et al. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

[24] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[25] S. Reed and N. de Freitas. Neural programmer-interpreters. Technical Report arXiv:1511.06279, 2015.

[26] J. Schmidhuber. Self-delimiting neural networks. *arXiv preprint arXiv:1210.0118*, 2012.

[27] J. Schmidhuber and S. Hochreiter. Guessing can outperform many long time lag algorithms. Technical report, 1996.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[29] R. K. Srivastava, K. Greff, and J. Schmidhuber. Training very deep networks. In *Advances in Neural Information Processing Systems*, pages 2368–2376, 2015.

[30] R. K. Srivastava, B. R. Steunebrink, and J. Schmidhuber. First experiments with powerplay. *Neural Networks*, 41:130–136, 2013.

[31] S. Sukhbaatar, J. Weston, R. Fergus, et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, pages 2431–2439, 2015.

[32] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[33] O. Vinyals, S. Bengio, and M. Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[34] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2674–2682, 2015.

[35] A. J. Wiles. Modular elliptic curves and fermats last theorem. *ANNALS OF MATH*, 141:141, 1995.

[36] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. *Back-propagation: Theory, architectures and applications*, pages 433–486, 1995.