1. Supplementary Material

1.1. Implementation details

Unless otherwise stated, all models use the default parameters from the official implementation of MAC \(^1\). Notable differences between these and those in the original paper [3] include using different dropout rates for different sections of the network (92% on output of biLSTM, 82% on convolutional filters), initializing weights by sampling from the Xavier uniform distribution [2], setting the question as the first control, and using a LR scheduler. We use a gate bias of 1.0 for the gated MAC as this achieves the best performance in the original paper; however, unless stated otherwise, we do not bias our adaptive computation algorithms.

Experimental results in Section 4 are obtained by training from scratch the MAC network three times and then using the resulting weights to train multiple other models. Each of the pre-trained MACs are further trained to obtain: four variants of DACT, corresponding to different values of \(\lambda\); and seven variants of ACT, six with different \(\lambda\)'s and one without ponder penalty. Training the variants is done by resetting all schedulers and optimizers and then reinitializing a new MAC with its corresponding gating mechanism. We then overwrite the pertinent weights with the pre-trained ones and train for another 30 epochs saving the best model according to the monitored precision. This is done to reduce the time needed to train all models while still providing significant mean and variance values for relevant metrics (i.e. precision and steps). Figure 1 shows how the average number of steps used by DACT-MACs as training progresses. For reference, we include the case of standard non-adaptive MAC as a dotted line (12 steps). Note that in the first epoch the model uses more steps than the value to which it converges, following a behavior similar to the pre-trained MAC, while subsequent epochs reduce computation. Using higher ponder costs translates into less variability in the number of steps used, shown as translucent bands of the same color.

Few changes were needed in order to apply adaptive MACs to GQA [4]. Instead of using regions of the image (top-down attention), our knowledge-base is composed of region detections (bottom-up) [1]. We also use 1x1 convolution filters instead of 3x3 for the same reason. Additionally, we use a 4 step non-adaptive MAC during the pre-training phase as this is the recommended number of steps for the model [4] and also yielded the best results in our experiments. The resulting accuracies, along with the mean number of steps used to attain them, are shown in Table 1.

Each question in GQA has been assigned one of 105 question types according to the operations needed to answer the question. Therefore, as was the case with question families in CLEVR, we group questions by type with the underlying assumption being that all questions of a type have similar complexities. Figure 2 validates that, as in CLEVR, DACT is adapting computation, and the total amount of computation varies following the time penalty.

1.2. Question Families

For any given synthetic image in the CLEVR dataset, a series of queries are generated by chaining a sequence

---

\(^1\)https://github.com/stanfordnlp/mac-network
Table 1. Our proposed method (DACT) achieved better accuracy than existing adaptive algorithms on the GQA test-dev set, while also adapting computation coherently to the values taken by the ponder cost hyper-parameter. However, the task did not benefit from increased computation, so all adaptive models incur in a small metric loss compared to non-adaptive variants.

![Image](image.png)

Figure 2. The figure shows the the distribution of the number of steps used by DACT for each one of the 105 different question types in the GQA dataset. In order from top (a) to bottom (c) we show how decreasing the time penalty ($5 \times 10^{-2}, 1 \times 10^{-2}, 5 \times 10^{-3}$ for a,b, c respectively) results in increased total computation.

of modular operations such as count, filter, compare, etc. These functional programs can then be expressed in natural language in multiple ways, for instance translating count(filtercolor(red, scene())) into “How many <C> <M> things are there?”, a translation which is accomplished by instantiating the text templates specific to each program following [5]. As a result, questions with the same functional program can be clustered together into question families that share a similar complexity. Figure 3 includes a text template for each of the question families present in CLEVR, sorted by the average number of steps used for validation questions belonging to the specific family.

### 1.3. Proofs

In this section we prove that our method for building the final answer $Y$ can be interpreted as attending the intermediate outputs $y_n$, with attention weights that follow a valid probability distribution. We include two proofs by induction to show that, for any $n$, the accumulated answer $a_n$ can be expressed as a weighted sum of all intermediate outputs up to the $n$th step, and that these weights always add up to one.

**Proposition.** Every accumulated answer $a_n$ can be expressed as a weighted sum of all intermediate outputs up to the $n$th step.

**Proof.** Assume $\alpha_i$ exists for each $y_i$ such that every $a_{n-1} = y_{n-1}\alpha_{n-1} + \cdots + y_0\alpha_0$. This is trivial to prove for $n = 1$ as $p_0 = 1$ makes $a_1 = y_1p_0 + a_0(1 - p_0)$ become $a_1 = y_1$.

$$a_n = y_n p_{n-1} + a_{n-1}(1 - p_{n-1})$$

$$= y_n p_{n-1} + (\alpha_{n-1}y_{n-1} + \cdots + \alpha_0y_0)(1 - p_{n-1})$$

$$= y_n p_{n-1} + \sum_{i=0}^{n-1} y_i(\alpha_i(1 - p_{n-1}))$$

**Proposition.** Every accumulated answer $a_n$ can be expressed as a weighted sum of all intermediate outputs up to the $n$th step, and the sum of the weights is equal to one.

**Proof.** The base case is again trivial to prove since $p_0 = 1$ when $n = 1$. Using the proof above we define $\beta_i$ to be the weights used to express $a_n$ as a weighed sum of $y_i$ $\forall i \in [1, n]$.

$$\beta_i = \begin{cases} p_{n-1} & \text{if } i = n \\ \alpha_i(1 - p_{n-1}) & \text{otherwise} \end{cases}$$

Assume $\alpha_i$ exists for each $y_i$ such that every $a_{n-1} = \alpha_{n-1}y_{n-1} + \cdots + \alpha_0y_0$ and $\sum_{i=0}^{n-1} \alpha_i = 1$.

$$\sum_{i=0}^{n} \beta_i = p_{n-1} + \sum_{i=0}^{n-1} \alpha_i(1 - p_{n-1})$$

$$= p_{n-1} + \sum_{i=0}^{n-1} \alpha_i - p_{n-1} \sum_{i=0}^{n-1} \alpha_i$$

$$= p_{n-1} + 1 - p_{n-1}$$

$$= 1$$
Figure 3. The image above shows the average number of steps used by DACT-MAC (λ = 5 × 10^{-3}) for each of the question families present in CLEVR, along with one template for each family to typify the whole group. Following [5], each question is generated by replacing <Z>, <C>, <M>, and <S> with the size, color, material and/or shape of objects present in the image. Note that families with fewer supporting objects are more likely to be answered in less steps, and finding the number of objects that possess a pair of qualities (both) is regarded as generally easier than finding those that possess (either).
References


