Don't Read Too Much Into It:
Adaptive Computation for Open-Domain Question Answering

Yuxiang Wu   Pasquale Minervini   Pontus Stenetorp   Sebastian Riedel

University College London

{yuxiang.wu,p.minervini,p.stenetorp,s.riedel}@cs.ucl.ac.uk

Abstract

Most approaches to Open-Domain Question Answering consist of a light-weight retriever that selects a set of candidate passages, and a computationally expensive reader that examines the passages to identify the correct answer. Previous works have shown that as the number of retrieved passages increases, so does the performance of the reader. However, they assume all retrieved passages are of equal importance and allocate the same amount of computation to them, leading to a substantial increase in computational cost. To reduce this cost, we propose the use of *adaptive computation* to control the computational budget allocated for the passages to be read. We first introduce a technique operating on individual passages in isolation which relies on anytime prediction and a per-layer estimation of an early exit probability. We then introduce SKY-LINEBUILDER, an approach for dynamically deciding on which passage to allocate computation at each step, based on a resource allocation policy trained via reinforcement learning. Our results on SQuAD-Open show that adaptive computation with global prioritisation improves over several strong static and adaptive methods, leading to a 4.3x reduction in computation while retaining 95% performance of the full model.

1 Introduction

Open-Domain Question Answering (ODQA) requires a system to answer questions using a large collection of documents as the information source. In contrast to context-based machine comprehension, where models are to extract answers from single paragraphs or documents, it poses a fundamental technical challenge in *machine reading at scale* (Chen et al., 2017) .

Most ODQA systems consist of two-stage pipelines, where *1)* a context retriever such as BM25 (Robertson, 2004) or DPR (Karpukhin et al., 2020) first selects a small subset of passages that are likely to contain the answer to the question, and *2)* a machine reader such as BERT (Devlin et al., 2019) then examines the retrieved contexts to extract the answer. This two-stage process leads to a computational trade-off that is indicated in Fig. 1. We can run computationally expensive deep networks on a large number of passages to increase the probability that we find the right answer ("All Layers, All Passages"), or cut the number of passages and layers to reduce the computational footprint at the possible cost of missing an answer ("6 Layers, Top-2 Passages").

We hypothesise that a better accuracy-efficiency trade-off can be found if the computational budget is not allocated statically, but based on the complexity of each passage, see "Adaptive Computation" in Fig. 1. If a passage is likely to contain the answer, allocate more computation. If it isn't, allocate less. The idea of conditioning neural network computation based on inputs has been pursued in previous work on *Adaptive Computation* (Bengio et al., 2015; Graves, 2016; Elbayad et al., 2020), however how to apply this idea to ODQA is still an open research question.

In this work, we introduce two adaptive computation methods for ODQA: TOWERBUILDER and SKYLINEBUILDER. TOWERBUILDER builds a *tower*, a composition of transformer layers on a single passage, until an early stopping condition is met—we find that this method already helps reducing the computational cost required for reading the retrieved passages. Then, for coordinating the construction of multiple towers in parallel, we introduce a global method, SKYLINEBUILDER, that incrementally builds multiple towers one layer at a time and learns a policy to decide which tower to extend one more layer next. Rather than building single transformer towers in isolation, it constructs
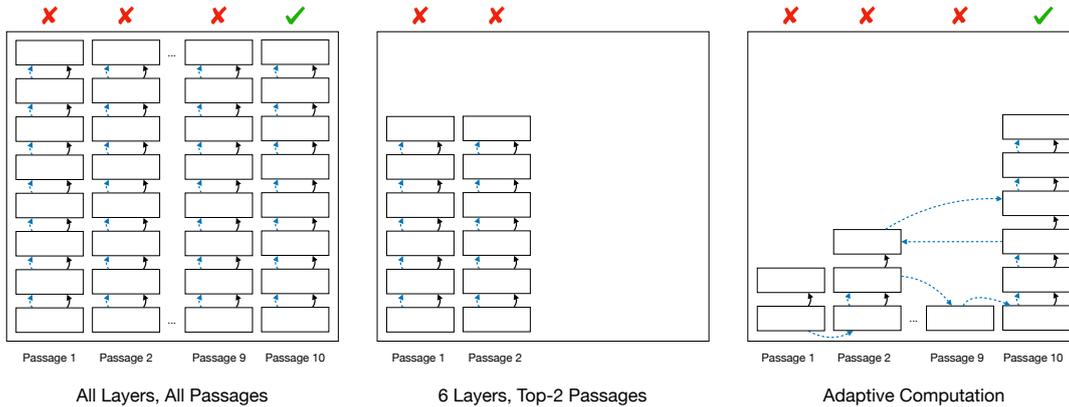
Figure 1: Static and adaptive computation for Open-Domain QA. Each block represents one layer of transformer computation on a passage. The solid arrows show how activations flow, and the dashed arrows indicate the order of computation. Only passage 10 contains the actual answer. Using all layers on all passages can find the answer, while processing only the top 2 retrieved passages with 6 layers is unable to find it. Adaptive computation can find the right passage, and allocates most computation budget to reading it.

a *skyline* of towers with different heights, based on which passages seem most promising to process further.

Our experiments on the SQuAD-Open dataset show that our methods are very effective at reducing the computational footprint of ODQA models. In particular, we find that SKYLINEBUILDER retains 95% of the accuracy of a 24-layer model using only 5.6 layers on average. In comparison, an adaptation of the method proposed by Schwartz et al. (2020) requires 9 layers for achieving the same results. Improvements are even more substantial for smaller number of layers—for example, with an average of 3 layers SKYLINEBUILDER reaches 89% of the full performance, whereas the approach of Schwartz et al. (2020) yields 57% and a model trained to use exactly 3 layers reaches 65%. Finally, SKYLINEBUILDER retains nearly the same accuracy at full layer count.

To summarise, we make the following contributions: *1)* we are the first to explore adaptive computation for ODQA by proposing two models: TOWERBUILDER and SKYLINEBUILDER; *2)* we experimentally show that both methods can be used for adaptively allocating computational resources so to retain the predictive accuracy with a significantly lower cost, and that coordinating the building of multiple towers via a learned policy yields more accurate results; *3)* when compared to their non-adaptive counterparts, our proposed methods can reduce the amount of computation by as much as 4.3 times.

2   Background

We first give an overview of ODQA and the relevant work in adaptive computation.

2.1   Open Domain Question Answering

In ODQA we are given a natural language query $\mathbf{q}$ and a large number of passages C—for example, all paragraphs in Wikipedia. The goal is to use C to produce the answer $\mathbf{y}$. In extractive ODQA this answer corresponds to a span in one of the documents of C. The corpus C can be very large, and a common approach to reduce computational costs is to first determine a smaller document set $D_{\mathbf{q}} \subseteq C$ by retrieving the most relevant n passages using an information retrieval module. Then we run a neural reader model on this subset. In most works, the reader model extracts answers by applying a per-passage reader to each input passage $\mathbf{x}_1, \ldots, \mathbf{x}_n \in D_{\mathbf{q}}$ and then apply some form of aggregation function on the per-passage answers to produce a final answer. Note that the passage reader can either produce an answer span as output, or NoAnswer in case the passage does not contain an answer for the given question.

2.2   Transformers for ODQA

Most current ODQA models rely on transformer-based architectures (Vaswani et al., 2017), usually pre-trained, to implement the PReader passage reader interface. In such models, an input passage is processed via a sequence of transformer layers; in the following, we denote the i-th transformer layer in the sequence as TransformerLayer$_i$. Let

$\mathbf{h}_i$ be the input to the i-th transformer layer and $\mathbf{h}_{i+1} = \text{TransformerLayer}_i(\mathbf{h}_i)$ its output. We set $\mathbf{h}_1 = \mathbf{x}$ to be the input passage. In standard non-adaptive Transformer-based models, we incrementally build a *tower*—a composition of Transformer layers—until we reach some pre-defined height n and use an output layer to produces the final output, $\mathbf{y} = \text{OutputLayer}(\mathbf{h}_n)$. In this work, due to efficiency reasons, we restrict ourselves to pre-trained ALBERT (Lan et al., 2020) models. One critical property of these models is parameter tying across layers: $\text{TransformerLayer}_i(\mathbf{h}) = \text{TransformerLayer}_j(\mathbf{h})$ for any i, j.

2.3 Adaptive Computation

Our goal is to early-exit the iterative layer-by-layer process in order to save computation. We assume this can be happening adaptively, based on the input, since some passages might require less computation to produce an answer than others. Schwartz et al. (2020) show how this can be achieved for classification tasks. They first require internal layers to be able to produce outputs too, yielding an *anytime* algorithm. [1] This can be achieved with a suitable training objective. Next, for each candidate layer i, they calculate the exit probability given its hidden state $\mathbf{h}_i$, and use them for taking an early-exit decision: if the highest exit probability is above a global threshold τ, they return $\text{OutputLayer}(\mathbf{h}_i)$ otherwise they continue with the following layers.

The output layer probabilities are not calibrated for exit decisions, and hence Schwartz et al. (2020) tune them on an held-out validation set via temperature calibration (Guo et al., 2017; Desai and Durrett, 2020), where a temperature T is tuned to adapt the softmax output probabilities at each layer.

3 Adaptive Computation in ODQA

Our goal is to incrementally build up towers of transformer layers for all passages in $D_{\mathbf{q}}$ in a way that minimises unnecessary computation. Our algorithms maintain a state, or *skyline*, $S = (H, A)$, consisting of current tower heights $H = (h_1, \dots, h_n)$, indicating how many layers have been processed for each of the n towers, and the last representations $A = (\mathbf{a}_1, \dots, \mathbf{a}_n)$ computed for each of the towers. We want to build up the

skyline so that we reach an accurate solution fast and then stop processing.

3.1 Early Exit with Local Exit Probabilities

Our first proposal is to extend the method from Schwartz et al. (2020) in order to build up the skyline S. In particular, we will process each passage $\mathbf{x}_i \in D_{\mathbf{q}}$ in isolation, building up height h_i and representation $\mathbf{a}_i$ until an *exit probability* reaches a threshold. For Schwartz et al. (2020) the exit probability is set to be the probability of the most likely class. While ODQA is not a classification problem per se, it requires solving one as a sub-step, either explicitly or implicitly: deciding whether a passage contains the answer. In turn, our first method TOWERBUILDER, uses the probability $1 - \text{HasAnswer}(\mathbf{a}_i)$ of the passage not containing the answer to calculate the exit probability at such given layer. In practice the probability $\text{HasAnswer}(\mathbf{a}_i)$ is calculated as the Sigmoid output of an MLP applied the representation of the CLS token in $\mathbf{a}_i$. Moreover, models are trained to produce HasAnswer probabilities for each layer using a per-layer loss. Following Schwartz et al. (2020), we also conduct temperature calibration for the HasAnswer modules using the development set.

When building up the towers, TOWERBUILDER produces early exit decisions for each tower in isolation. Once all towers have been processed, the method selects the highest m towers in the final S^* to produce the final answer, where m is a hyperparameter. Since some of the selected towers in S^* may not have full height, we will need to continue unrolling them to full height to produce an answer. We will call this the LastLayer strategy. Alternatively, we can return the solution at the current height, provided that we use an anytime model not just for HasAnswer predictions but also for answer extraction. We will refer to this strategy as AnyLayer. By default we use LastLayer but we will conduct ablation study of these two approaches in Section 5.3.

3.2 Global Scheduling

We can apply TOWERBUILDER independently to each passage $\mathbf{x}_i \in D_{\mathbf{q}}$. However, if we have already found an answer after building up one tower for a passage $\mathbf{x}_i$, we can avoid reading other passages. Generally, we imagine that towers that are more likely to produce the answers should be processed first and get more layers allocated to. To

---

[1] In practice, Schwartz et al. (2020) choose a subset of layers to be candidate output layers, so strictly speaking we cannot exit any time, but only when a candidate layer is reached.

assess if one tower is more likely to contain an answer, we need to compare them and decide which tower has highest *priority*. This type of strategy cannot be followed when processing passages in isolation, and hence we consider a global multi-passage view.

A simple approach for operating on multiple passages is to re-use information provided to the TOWERBUILDER method and select the next tower to extend using the HasAnswer probabilities. In particular, we can choose the next tower to build up as $j = \arg\max_i \text{HasAnswer}(\mathbf{a}_i)$, and then set $\mathbf{a}_j \leftarrow \text{TransformerLayer}(\mathbf{a}_j)$ and $h_j \leftarrow h_j + 1$ in the state S. To efficiently implement this strategy we use a priority queue. Every time a tower is expanded, its HasAnswer probability is re-calculated and used in a priority queue we choose the next tower from. Once we reach the limit of our computation budget, we can stop the reading process and return the results of the highest m towers S^* as inputs to its Output phase. The two aforementioned answer extraction methods (i.e., AnyLayer and LastLayer) also apply to this method.

3.3 Learning a Global Scheduler

Using HasAnswer probabilities to prioritise towers is a sensible first step, but not necessarily optimal. First, while the probabilities are calibrated, they are tuned for optimising the negative log-likelihood, not the actual performance of the method. Second, the HasAnswer probability might not capture everything we need to know about the towers in order to make decisions. For example, it might be important to understand what the rank of the tower's passage is in the retrieval result, as higher ranked passages might be more fruitful to expand. Finally, the HasAnswer probabilities are not learnt with the global competition of priorities across all towers, so they are not optimal for comparing priorities between towers that have different heights.

To overcome the above issues, we frame the tower selection process as a reinforcement learning (RL) problem: we consider each tower $i \in \{1, \ldots, n\}$ as a candidate action, and learn a policy $\pi(i|S)$ that determines which tower to expand next based on the current skyline. We present the corresponding details below.

3.3.1 Policy

Our policy calculates $\pi(i|S)$ using a priority vector $\mathbf{p}(S) \in \mathbb{R}^n$. The priority $p_i(S)$ of each tower i is calculated using a linear combination of the

HasAnswer probability of that tower and the output of a multi-layer perceptron MLP_θ. The perceptron is parametrised by θ and uses a feature representation $\mathbf{f}_i(S)$ of tower i in state S as input. Concretely, we have:

$$p_i(S) = \alpha \text{HasAnswer}(\mathbf{a}_i) + \text{MLP}_\theta(\mathbf{f}_i(S))$$

where α is a learnable mixture weight. As feature representation we use $\mathbf{f}_i(S) = [\text{HeightEmb}(h_i), \text{IndexEmb}(i), \text{HasAnswer}(\mathbf{a}_i)]$ where the tower height h_i and index i are represented using embedding matrices $\text{HeightEmb} \in \mathbb{R}^{l \times d}$ and $\text{IndexEmb} \in \mathbb{R}^{n \times d}$ respectively. When a tower is currently empty, an initial priority p_i^0 will be provided: it can either be a fixed value or a learnable parameter, and its impact is analysed in Section 5.2. Given the above priority vector, the policy simply maps per tower priorities to the probability simplex:

$$\pi(i|S) = \text{Softmax}_i(\mathbf{p}(S)).$$

The parameters (α, θ) introduced by this policy do not introduce much computational overhead: with embedding size $d = 8$ and using 32-dimensional hidden representations in the MLP, this model only introduces 1,039 new parameters, a small amount compared to ALBERT ($\approx$ 18M).

3.3.2 Training

While executing a policy, the scheduler needs to make discrete decisions as which tower to pursue. These discrete decisions mean we cannot simply frame learning as optimising a differentiable loss function. Instead we use the REINFORCE algorithm (Williams, 1992) for training our policy, by maximising the expected cumulative reward. For us, this reward is defined as follows. Let $\mathbf{i}_1^m = i_1, \ldots, i_m$ and $\mathbf{S}_1^m = S_1, \ldots, S_m$ be a trajectory of (tower selection) actions and states, respectively. We then set the cumulative reward to $R(\mathbf{i}_t^m, \mathbf{S}_t^m) = r(i_t, S_t) + \gamma R(\mathbf{i}_{t+1}^m, \mathbf{S}_{t+1}^m)$ where $r(i_t, S_t)$ is a immediate per-step reward we describe below, and γ is a discounting factor.

We define an immediate per-step reward $r(i, S)$ of choosing tower i in state S as $r(i, S) = r - c$ where $r = 1$ if the selected tower contains an answer and $r = 0$ otherwise. $c \in \mathbb{R}_+$ is a penalty cost of taking a step. In our experiments, we set $c = 0.1$.

4 Related Work

Adaptive Computation One strategy to reduce a model's complexity consists in dynamically deciding which layers to execute during inference (Bengio et al., 2015; Graves, 2016). *Universal transformers* (Dehghani et al., 2019) can learn after how many layers to emit an output conditioned on the input. Elbayad et al. (2020) generalise universal transformers by also learning which layer to execute at each step. Schwartz et al. (2020); Liu et al. (2020) propose methods that can adaptively decide when to early stop the computation in sentence classification tasks. To the best of our knowledge, previous work has focused adaptive computation for a single input. We are the first to learn how to prioritise computation across instances in the context of ODQA.

Smaller Networks Another strategy consists in training smaller and more efficient models. In *layer-wise dropout* (Liu et al., 2018), during training, layers are randomly removed, making the model robust to layer removal operations. This idea was expanded Fan et al. (2020) to modern Transformer-based models. Other methods include *Distillation* (Hinton et al., 2015) of a teacher model into a student model, *Pruning* of architectures after training (LeCun et al., 1989) and *Quantisation* of the parameter space (Wróbel et al., 2018; Shen et al., 2019; Zafrir et al., 2019). These methods are not adaptive, but could be used in concert with the methods proposed here.

Open Domain Question Answering Most modern ODQA systems adopt a two-stage approach that consists of a retriever and a reader, such as DrQA (Chen et al., 2017), HardEM (Min et al., 2019), BERTserini (Yang et al., 2019), Multi-passage BERT (Wang et al., 2019), and PathRetriever (Asai et al., 2020). As observed by Chen et al. (2017); Yang et al. (2019); Karpukhin et al. (2020); Wang et al. (2019), the accuracy of such two-stage models increases with more passages retrieved. But it remains a challenge to efficiently read a large number of passages as the reader models are usually quite computationally costly.

5 Experiments

Dataset SQuAD-Open (Chen et al., 2017) is a popular open-domain question answering dataset based on SQuAD. We partition the dataset into four subsets: training set, two development sets

SQuAD-Open	train	dev_0	dev_1	test
Size	78,839	4,379	4,379	10,570
Hits@30	71.2%	72.7%	72.1%	77.9%

Table 1: Dataset sizes and retriever performances.

(dev_0 and dev_1), and test set, and their details are summarised in Table 1.

Experimental Setup We follow the preprocessing approached proposed by Wang et al. (2019) and split passages into 100-word long chunks with 50-word long strides. We use a BM25 retriever to retrieve the top n passages for each question as inputs to the reader and the Wikipedia dump provided by Chen et al. (2017) as source corpus. Following Wang et al. (2019), we set $n = 5$ for training and $n = 30$ for test evaluations. Table 1 shows the Hits@30 results of our BM25 retriever on the dataset and they are comparable with previous works (Yang et al., 2019; Wang et al., 2019).

Reader Model For all our experiments, we fine-tune a pre-trained ALBERT model (Lan et al., 2020), consisting of 24 transformer layers and cross-layer parameter sharing. We do *not* use global normalisation (Clark and Gardner, 2018) in our implementation, but our full system (without adaptive computation) achieves an EM score of 52.6 and is comparable to Multi-passage BERT (Wang et al., 2019) which uses global normalisation.

Training Pipeline The anytime reader models are first trained on training set and validated on dev_0. Then we conduct temperature calibration on dev_0. For SKYLINEBUILDER, the scheduler model is trained on dev_0 with the calibrated anytime model, and validated with dev_1.

Baselines Following Schwartz et al. (2020), we use three types of baselines: *1*) the *standard baseline* that reads all passages and outputs predictions at the final layer, *2*) the *efficient baseline* that always exits at a given intermediate layer for all passages, and is optimised to do so, *3*) the *top-k baseline* that only reads the k top ranked passages and predicts the answer at their final layers.

Evaluation protocol Our goal is to assess the computational efficiency of a given method in terms of accuracy vs. computational budget used. We follow Fan et al. (2020) and consider the computation of one layer as a unit of computational

(a) SKYLINEBUILDER vs. baselines

(b) Local vs. Global Models

(c) AnyLayer vs. LastLayer

(d) Learnt vs. Fixed Initial Priorities
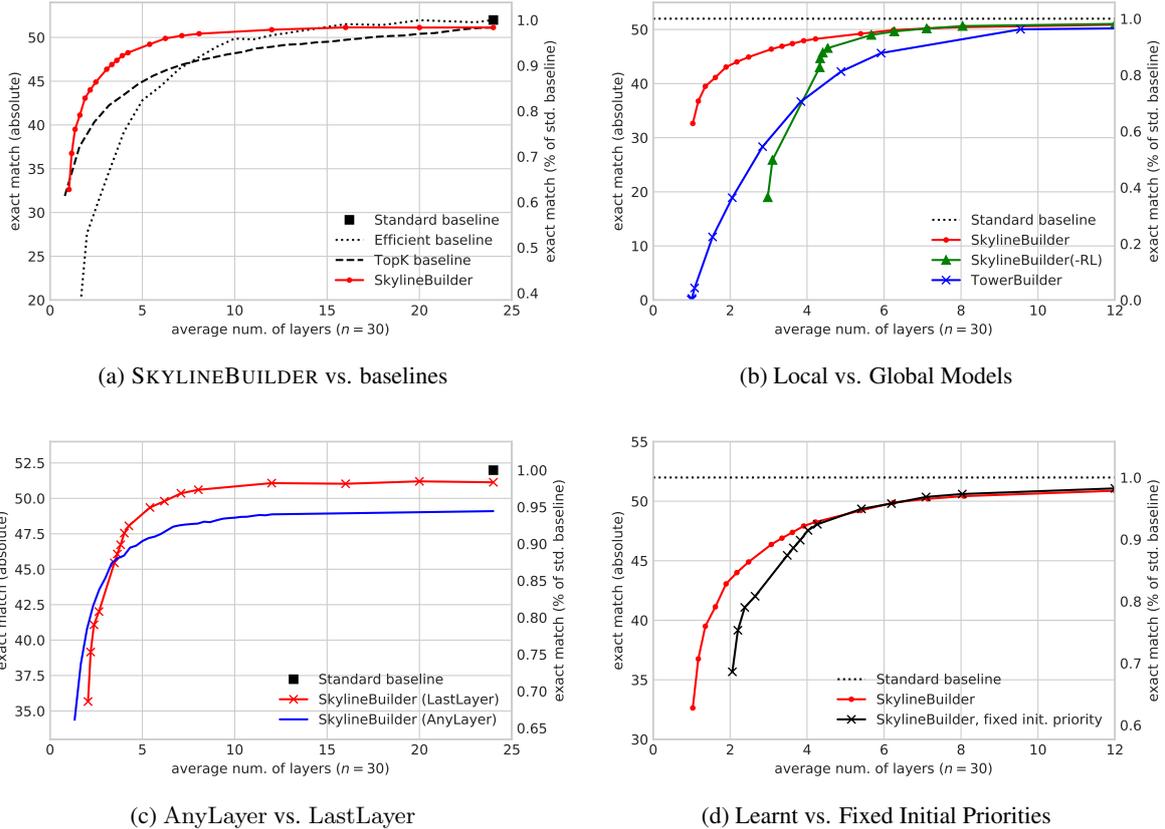
Figure 2: Evaluation results on the SQuAD-Open test set with 30 passages.

cost. In particular, we will assess how many layers, on average, each method builds up for each passage. Similarly to Schwartz et al. (2020), we show the accuracy-efficiency trade-off for different strategies by showing the computation cost on the x-axis, and the Exact Match (EM) [2] score on the y-axis.

5.1 Static vs. Adaptive Computation

We first investigate how adaptive computation compares to the static baselines. We will focus on a single adaptive method, SKYLINEBUILDER, and assess different adaptive variants later.

Fig. 2a shows the accuracy of SKYLINEB-UILDER at different budgets when compared to the standard, efficient, and top-k baselines. We note that it reaches the similar results of the static baselines with much fewer layers. In particular, it yields substantially higher performance than static methods when the computational budget is smaller than ten layers. For example, when given four layers on average, SKYLINEBUILDER achieves EM

Method	Avg. #layers	Reduction
Standard baseline	24	1.0x
Efficient baseline	9.5	2.5x
Top-k baseline	14.4	1.7x
TOWERBUILDER	9.0	2.7x
SKYLINEBUILDER(-RL)	6.1	3.9x
SKYLINEBUILDER	**5.6**	**4.3x**

Table 2: Reduction in layer computations while achieving 95% of the accuracy of the standard baseline.

score 48.0, significantly outperforming EM score 44.2 of the top-k baseline.

In Table 2 we consider a setting where SKY-LINEBUILDER and the static baseline reach comparable (95%) performance of the full 24-layer model. We see that simply reducing the number of passages to process is giving a poor accuracy-efficiency trade-off, requiring 14.4 layers (or 18 passages) to achieve this accuracy. The efficient baseline fares better with 9.5 layers, but it is still outperformed by SKYLINEBUILDER, that only needs 5.6 layers on average to reach the desired accuracy.

	Var(h)	Avg(rank)	Flips	$h_+ - h_-$	HAP	Exact Match
Efficient Baselines	0.00	14.50	-	0.00	6.1%	23.47
TOWERBUILDER	11.05	13.38	-	3.68	22.0%	17.10
SKYLINEBUILDER(-RL)	7.46	13.06	13.37	3.46	27.4%	27.95
SKYLINEBUILDER	12.71	8.78	6.48	5.99	40.5%	**33.60**

Table 3: Quantitative analysis on SQuAD Open dev_1 set with top 30 passages and two layers of computation per passage on average.

5.2 Local vs. Global Models

What is the impact of globally selecting which towers to extend, rather than taking early-exit decisions on a per-tower basis? To answer this question, we consider two global methods: SKYLINEBUILDER and SKYLINEBUILDER(-RL), the method in Section 3.2 that uses HasAnswer probabilities as priorities without any RL-based selection policy. We compare both to the local method TOWERBUILDER.

Fig. 2b shows that, while for very low budgets TOWERBUILDER outperforms SKYLINEBUILDER(-RL), with a budget larger than 4 layers it is not the case anymore. This may be due to a tendency of SKYLINEBUILDER(-RL) spending an initial computation budget on exploring many towers—in Fig. 3 we show examples of this behaviour. It is also shown that SKYLINEBUILDER considerably outperforms both TOWERBUILDER and SKYLINEBUILDER(-RL). Along with the results in Table 2, the comparisons above indicate that 1) global scheduling across multiple towers is crucial for improving efficiency, and 2) optimising the adaptive policy with RL manage to exploit global features for tower selection, leading to further improvements.

5.3 Ablation Studies

Any Layer vs. Last Layer Model For comparing the LastLayer and the AnyLayer strategies introduced in Section 3.1, we show the behaviour of these methods for the SKYLINEBUILDER scheduling algorithm in Fig. 2c. Using an anytime answer extraction model has a negative effect on accuracy. We see this clearly at 24 layers where AnyLayer lags substantially behind the standard baseline while LastLayer almost reaches it. We see this gap across the whole budget spectrum, leading to less accurate results except for very small budgets.

Learning Initial Priorities SKYLINEBUILDER uses a learnt initial priority for each tower. This not only enables it learn which towers to process first at the beginning, but also how long to wait until other towers are visited. Fig. 2d shows the benefit gained from adopting this strategy: without training the initialisation priorities, SKYLINEBUILDER spend more computation on passages that are likely not needed. Once an average of 4 layers have been added, the benefit disappears as SKYLINEBUILDER with learnt initial priorities will try to visit more candidates itself.

5.4 Quantitative Analysis

This section aims at understanding where and how our adaptive strategies behave differently, and what contributes to the gain in the accuracy-efficiency trade-off. We propose the following quantitative metrics: *1)* Var(h): variance of the heights of the towers. *2)* Avg(rank): average rank of the tower when the method chooses which tower to build on. *3)* Flips: how often does the strategy switch between towers, measuring the exploration-exploitation trade-off of a method. *4)* $h_+ - h_-$: h_+ (resp. h_-) is the average height of towers with (resp. without) an answer. Their difference measures the difference in amount of computation between passages with the answer and the ones without an answer. *5)* HasAnswer Precision (HAP): how often a tower selection action selects a tower whose passage contains the answer.

We analyse our proposed methods along with static baselines on the SQuAD development set; results are outlined in Table 3. Overall, the higher the HasAnswer Precision, the more accurate the method. This finding matches with our intuition that, if a tower selection strategy can focus its computation on passages that contain the answer, it yields more accurate results with smaller computation budgets.
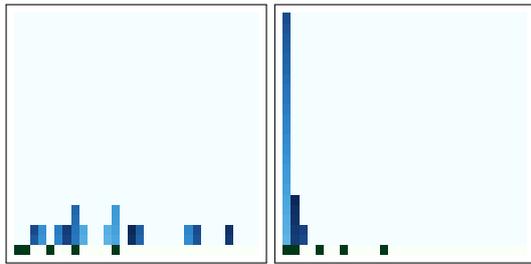
(a) Example 1
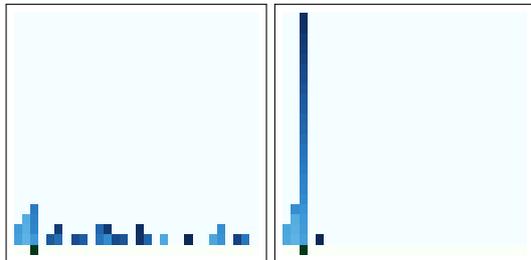
(b) Example 2
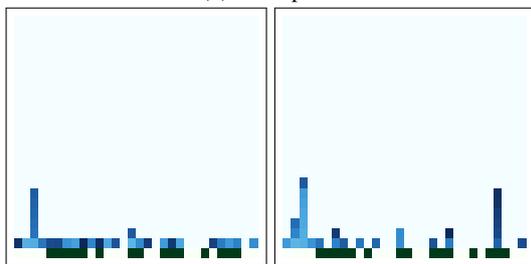
(c) Example 3

Figure 3: Examples of the skylines built by SKYLINEBUILDER(-RL) (left) and SKYLINEB-UILDER (right), with two layers per passage on average. The green blocks indicate towers that contain the answer.

Comparing SKYLINEBUILDER(-RL) and SKY-LINEBUILDER gives more insights regarding what the RL training scheme learns. SKYLINEBUILDER learns a policy with the highest Var(h), the lowest Avg(rank), and the lowest number of tower flips, suggesting that *1)* it focuses on a few towers rather than distributing its computation over all passages, *2)* it is more likely to select top-ranked passages, and *3)* it switches less between towers, and tends to build one tower before switching to another. SKY-LINEBUILDER also yields the highest HasAnswer Precision and $h_+ - h_-$, meaning that tends to pri-oritise the passages containing the answer.

5.5 Qualitative Analysis and Visualisation

Here we analyse how different methods build the skyline. Fig. 3 shows some examples of skylines built by SKYLINEBUILDER(-RL) and SKYLINEB-
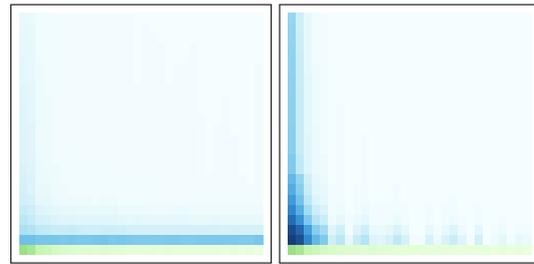

Figure 4: Heatmap of the tower selections by SKYLINEBUILDER(-RL) (left) and SKYLINEB-UILDER (right). The colour gradient of the blues blocks reflects their selection frequencies.

UILDER. The towers are ordered by the rank of their associated passages in the retrieval results from left to right, and are built bottom-up. The colour gradient of the blues blocks reflects the or-der in which the layers are built: darker cells corre-spond to layers created later in the process.

In Fig. 3a and Fig. 3b we can see that SKY-LINEBUILDER tends to focus on one or two towers, whereas SKYLINEBUILDER(-RL) has a more even distribution of computation across different towers. In Fig. 3b, even when only one tower contains the answer, SKYLINEBUILDER manages to locate it and build a full-height tower on it.

Fig. 3c shows a case where none of the top 4 passages contains the answer. SKYLINEBUILDER goes over these irrelevant towers quickly and start exploring later towers, until it reaches the tower with rank 27 and becomes confident enough to keep building on it. These examples shows how SKYLINEBUILDER learns an efficient scheduling algorithm to locate passages containing the answer with very limited budgets.

To understand how our proposed methods work at macro level, we use heat-maps (Fig. 4) for showing how frequently each block is selected. The green row at the bottom indicates the fre-quency of each passage containing the answer. SKYLINEBUILDER(-RL) explores all passages quite evenly, whereas SKYLINEBUILDER learns to prioritise top-ranked towers. This preference is reasonable because, as shown by the green row at the bottom, top-ranked towers are more likely to contain the answer. Also note that SKYLINEB-UILDER does not naively process towers from left to right like the top-k baseline does, but instead it learns a trade-off between *exploration and exploita-tion*, leading to the significant improvement over the top-k baseline shown in Fig. 2a.

Models	Num. layers	EM
DistilBERT (Sanh et al., 2019)	6	40.5
SKYLINEBUILDER	1.6	41.1
SKYLINEBUILDER	3	46.4
SKYLINEBUILDER	6	**49.7**

Table 4: Comparing adaptive computation with distillation on SQuAD-Open test set.

5.6 Adaptive Computation vs. Distillation

Distillation is another orthogonal approach to reduce computational cost. We compare our adaptive computation method SKYLINEBUILDER with a static DistilBERT (Sanh et al., 2019) baseline, and the results are shown in Table 4. Our method significantly outperforms DistilBERT while computing much fewer layers.

6 Discussions and Future Works

In this paper, we focus on reducing the number of layers and operations of ODQA models, but the actual latency improvement also depends on the hardware specifications. On GPUs we cannot expect a reduction in the number of operations to translate 1:1 to lower execution times, since they are highly optimised for parallelism. [3] We leave the parallelism enhancements of SKYLINEBUILDER for future work.

We also notice that the distillation technique is complementary to the adaptive computation methods. It will be interesting to integrate these two approaches to achieve further computation reduction for ODQA models.

7 Conclusions

In this work we show that adaptive computation can lead to substantial efficiency improvements for ODQA. In particular, we find that it is important to allocate budget dynamically across a large number of passages and prioritise different passages according to various features such as the probability that the passage has an answer. Our best results emerge when we learn prioritisation policies using reinforcement learning that can switch between exploration and exploitation. On our benchmark, our method achieves 95% of the accuracy of a 24-layer model while only needing 5.6 layers on average.

---

[3]When evaluated on an NVIDIA TITAN X GPU, our proposed SKYLINEBUILDER achieves approximately 2.6x latency reduction while retaining 95% of the performance.

References

Akari Asai, Kazuma Hashimoto, Hannaneh Hajishirzi, Richard Socher, and Caiming Xiong. 2020. Learning to retrieve reasoning paths over wikipedia graph for question answering. In *ICLR*. OpenReview.net.

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. 2015. Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297.

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading wikipedia to answer open-domain questions. In *ACL (1)*, pages 1870–1879. Association for Computational Linguistics.

Christopher Clark and Matt Gardner. 2018. Simple and effective multi-paragraph reading comprehension. In *ACL (1)*, pages 845–855. Association for Computational Linguistics.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2019. Universal transformers. In *ICLR (Poster)*. OpenReview.net.

Shrey Desai and Greg Durrett. 2020. Calibration of pre-trained transformers. *CoRR*, abs/2003.07892.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. 2020. Depth-adaptive transformer. In *ICLR*. OpenReview.net.

Angela Fan, Edouard Grave, and Armand Joulin. 2020. Reducing transformer depth on demand with structured dropout. In *ICLR*. OpenReview.net.

Alex Graves. 2016. Adaptive computation time for recurrent neural networks. *CoRR*, abs/1603.08983.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On calibration of modern neural networks. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 1321–1330. PMLR.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *CoRR*, abs/2004.04906.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A lite BERT for self-supervised learning of language representations. In *ICLR*. OpenReview.net.

Yann LeCun, John S. Denker, and Sara A. Solla. 1989. Optimal brain damage. In *NIPS*, pages 598–605. Morgan Kaufmann.

Liyuan Liu, Xiang Ren, Jingbo Shang, Xiaotao Gu, Jian Peng, and Jiawei Han. 2018. Efficient contextualized representation: Language model pruning for sequence labeling. In *EMNLP*, pages 1215–1225. Association for Computational Linguistics.

Weijie Liu, Peng Zhou, Zhiruo Wang, Zhe Zhao, Haotang Deng, and Qi Ju. 2020. Fastbert: a self-distilling BERT with adaptive inference time. In *ACL*, pages 6035–6044. Association for Computational Linguistics.

Sewon Min, Danqi Chen, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2019. A discrete hard EM approach for weakly supervised question answering. In *EMNLP/IJCNLP (1)*, pages 2851–2864. Association for Computational Linguistics.

Stephen Robertson. 2004. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of Documentation*, 60(5):503–520.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108.

Roy Schwartz, Gabriel Stanovsky, Swabha Swayamdipta, Jesse Dodge, and Noah A. Smith. 2020. The right tool for the job: Matching model and instance complexities. In *ACL*, pages 6640–6651. Association for Computational Linguistics.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. Q-BERT: hessian based ultra low precision quantization of BERT. *CoRR*, abs/1909.05840.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*, pages 5998–6008.

Zhiguo Wang, Patrick Ng, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. 2019. Multi-passage BERT: A globally normalized BERT model for open-domain question answering. In *EMNLP/IJCNLP (1)*, pages 5877–5881. Association for Computational Linguistics.

R. J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Krzysztof Wróbel, Marcin Pietron, Maciej Wielgosz, Michal Karwatowski, and Kazimierz Wiatr. 2018. Convolutional neural network compression for natural language processing. *CoRR*, abs/1805.10796.

Wei Yang, Yuqing Xie, Aileen Lin, Xingyu Li, Luchen Tan, Kun Xiong, Ming Li, and Jimmy Lin. 2019. End-to-end open-domain question answering with bertserini. In *NAACL-HLT (Demonstrations)*, pages 72–77. Association for Computational Linguistics.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8BERT: quantized 8bit BERT. *CoRR*, abs/1910.06188.