Machine Translation Decoding beyond Beam Search

Rémi Leblond DeepMind
Jean-Baptiste Alayrac DeepMind
Laurent Sifre DeepMind
Miruna Pislar DeepMind
Jean-Baptiste Lespiau DeepMind
Ioannis Antonoglou DeepMind
Karen Simonyan DeepMind
Oriol Vinyals DeepMind

Abstract

Beam search is the go-to method for decoding auto-regressive machine translation models. While it yields consistent improvements in terms of BLEU, it is only concerned with finding outputs with high model likelihood, and is thus agnostic to whatever end metric or score practitioners care about. Our aim is to establish whether beam search can be replaced by a more powerful metric-driven search technique. To this end, we explore numerous decoding algorithms, including some which rely on a value function parameterised by a neural network, and report results on a variety of metrics. Notably, we introduce a Monte-Carlo Tree Search (MCTS) based method and showcase its competitiveness. We provide a blueprint for how to use MCTS fruitfully in language applications, which opens promising future directions. We find that which algorithm is best heavily depends on the characteristics of the goal metric; we believe that our extensive experiments and analysis will inform further research in this area.

1 Introduction

Sequence to sequence model decoding remains something of a paradox. The most widely adopted training method for these models is maximum likelihood estimation (MLE), which aims at maximising the probability of the ground truth outputs provided in the training datasets. Consequently, decoding from MLE-trained models is done by trying to find the output to which the model assigns maximum likelihood. Unfortunately, as models usually predict tokens one by one, exact search is not feasible in the general case and practitioners resort to heuristic mechanisms instead.

The most popular of these heuristics is beam search (Reddy, 1977), which maintains several hypotheses in parallel and is guaranteed to find a more likely output than the more basic greedy decoding. This approach has some obvious flaws: for one, it is completely agnostic to the actual metrics (or scores) practitioners actually want to optimise.

Even more crucially, in most cases beam search fails at the one thing it is supposed to do: finding the optimal output sequence (w.r.t the model), as shown by Stahlberg and Byrne (2019). Also alarming are the findings of Welleck et al. (2020), proving that traditional search mechanisms can yield infinite-length outputs, to which the model assigns zero probability. Finally, the use of likelihood as a training objective has a spectacular side-effect: it causes trained models to have an inordinate fondness for empty outputs. By using exact search on the output likelihood in machine translation, Stahlberg and Byrne (2019) show that in more than half of cases the highest scoring output according to the model is the empty sentence!

All told, we rely on models placing a surprising emphasis on empty outputs, and on a decoding mechanism which usually fails to find optimal outputs; and both ignore the relevant metrics. One can then justifiably wonder why we observe impressive MT results. Stahlberg and Byrne (2019) provide an apparently paradoxical explanation: it is precisely because the decoding mechanisms are imperfect that models produce outputs of high quality. Meister et al. (2020a) elaborate on this assumption; they show that beam search optimises for a slightly modified likelihood objective, promoting uniform distribution probability inside sentences.

This state of affairs seems highly unsatisfactory. While a whole body of work has been devoted to alleviating these issues, most approaches have been concerned with training (Bengio et al., 2015; Ranzato et al., 2016; Shen et al., 2016; Norouzi et al., 2016; Bahdanau et al., 2017; Edunov et al., 2018; Leblond et al., 2018), or making the search mechanism differentiable (Collobert et al., 2019). These have resulted in performance increase, but they still rely on likelihood as an objective for decoding. Further, Choshen et al. (2019) shows that

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 8410–8434
November 7–11, 2021.

performance improvements using RL are limited and poorly understood.

In this paper, we focus instead on contrasting the performance of beam search to alternative decoding algorithms aimed at optimising various metrics of interest directly, via a value function (or the metric itself when available). Notably, we experiment with variants of the powerful Monte Carlo Tree Search (MCTS) (Coulom, 2006; Kocsis and Szepesvári, 2006) mechanism, which has a proven track record in other sequential applications (Browne et al., 2012; Silver et al., 2017). We investigate whether, by optimising the metric of interest at test time, one can obtain improved performance compared to likelihood-based approaches, and whether performance scales with the amount of computation – as opposed to that of beam search which has been shown to degrade with large beam sizes (Cohen and Beck, 2019).

We concentrate on machine translation (MT), an emblematic, well-studied sequence to sequence task, with readily available data and benchmarks.

Contributions. (i) We recall that there are two different types of metrics: *reference-based* scores, which rely on ground truth translations, in contrast to *reference-less* ones. We design a new score, Multilingual BERTScore, as an imperfect but illustrative example of the latter. (ii) We introduce several new decoding algorithms, detailing their implementation and how best to use them for MT. We provide a blueprint for how to use MCTS profitably in NLP (with pseudocode for a batched Numpy-based (Harris et al., 2020) implementation), which opens the door for many exciting applications. (iii) We run extensive experiments to study the performance of decoding mechanisms for different metrics. We show that beam search is the best option only for *reference-based* metrics. For those, value-based alternatives falter as the value problem is too hard – since it ultimately relies on reconstructing hidden information. For reference-less scores, beam search is outperformed by its competitors, including MCTS.

Outline. We go over the related work in Section 2. In Section 3, we contrast several types of metrics, and introduce illustrative examples. We review beam search and introduce alternative algorithms in Section 4. We explain how we train the required value function for value-based methods in Section 5. In Section 6 go over experimental details and results. Finally, we discuss our results, their limitations and possible next steps in Section 7.

2 Related Work

Autoregressive models for sequence generation typically output more coherent sequences (Gu et al., 2018), as each token prediction takes into account its predecessors. However, this gain comes at a cost: finding the sequence with maximum probability according to the model – $\text{argmax}_{y \in \mathcal{Y}} \pi(y|x)$ – becomes an intractable search problem over the combinatorial space $\mathcal{Y}$. Given the size of the (token) action space $\mathcal{A}$, exact search appears out of the realm of possibility. So we resort to incremental prediction; picking individual tokens, without knowing how these choices will impact the final likelihood. We describe the three most widely used methods, which all pick tokens one by one from left to right.

The *sampling* method predicts tokens by directly sampling from the model policy $\pi(y_{t+1}|x, y_1...y_t)$, computed via a softmax operator applied to the model logits (Ackley et al., 1985) – possibly with a temperature parameter. The *greedy search* method incrementally picks the tokens with highest probability according to the model (akin to sampling with very low temperature). This inexpensive approach can be seen as a special case of the sampling method, with very low temperature. Finally, *beam search* maintains a beam of k possible translations, updating them incrementally by ranking their extensions via the model likelihood. While k times more expensive than the previous approaches, beam search has stood the test of time, resulting in steady performance improvements on MT tasks.

Building on these methods, a number of improvements have been proposed. Welleck et al. (2019) explore out-of-order decoding, where the model additionally learns the order in which to decode tokens. This provides benefits in a variety of tasks, but unfortunately not MT. Wang et al. (2020) use look-ahead in the beam search to take into account future likelihood, which yields improvements on low-data tasks, but again does not outperform beam search on MT. Meister et al. (2020b) speeds up beam search for monotonous scores.

Several works focus on the interplay between the incremental models and beam search. Cohen and Beck (2019) shows that performance is not monotonically increasing with beam size, but degrades after a fairly small value of k. Stahlberg and Byrne (2019) devise a clever exact search

mechanism, relying on the fact that likelihoods are monotonically decreasing with size. While still prohibitively expensive, this approach underlines several key facts. First, beam search does not recover $\text{argmax}_{y \in \mathcal{Y}} \pi(y|x)$ in most cases, whatever the computational budget. Second, $\text{argmax}_{y \in \mathcal{Y}} \pi(y|x)$ is the empty sequence more than half the time in MT. Eikema and Aziz (2020) propose an interesting explanation for this observation: while models are good at spreading probability mass over a large quantity of acceptable outputs, they are unable to effectively pick the best one. Indeed, the mode of the distribution might even be disjoint from the area where the models assigns the majority of probability mass. They propose using *minimum Bayes risk* decoding, which leverages the whole distribution rather than only its mode, and can outperform vanilla beam search in low-resource scenarios. Borgeaud and Emerson (2019), in a similar vein, develop an additional voting-based step on top of beam search to select more representative sequences, based on similarity measures.

A large body of work has been dedicated to improving sampling diversity, which plays a key role in many NLP applications – though not usually in machine translation. Fan et al. (2018) propose only sampling from the top k tokens according to the policy to avoid sampling from the tail of the distribution. Holtzman et al. (2019) adopt a similar approach, but instead of fixing k they fix p, the size of the 'nucleus' of the distribution from which sampling is allowed to select tokens. This performs better on open-ended tasks. Kool et al. (2019) propose a search mechanism in-between sampling and beam search, which produces provably unique samples by leveraging the Gumbel-Max trick (Gumbel, 1954). Yu et al. (2020) use a different, much more expensive flavor of MCTS to add diverse samples to a larger NMT system: instead of relying on direct value estimation, they rely on (expensive) rollouts to estimate node values.

Finally, the most closely related method to our proposed MCTS decoding is value-guided beam search, as developed by He et al. (2017); Ren et al. (2017) for MT and image captioning. Contrary to all other methods presented in this section, this approach does not solely rely on model likelihood. In both papers a value network – estimating the eventual score from an unfinished sample – is trained in addition to the policy network. Then instead of following the likelihood to select the hypotheses on the beam, one uses a linear combination of the policy logits and the value. This approach has shown improved performance compared to vanilla beam search; notably, it is less sensitive to the chosen beam size. While this method uses the value exclusively for one-step look-aheads, MCTS can be leveraged to explore further in the future. Additionally, it requires evaluating the value score of all tokens at each step, which can be prohibitively expensive if the action space is big (in MT, one routinely uses vocabularies of size larger than 30000).

3 Machine Translation metrics

There are two main evaluation strategies for MT outputs. The first one crucially relies on having access to a held-out test set of high quality (input, output) sentence pairs $(x, y_x)_{x \in \mathcal{X}}$. One can then compute a monolingual similarity score between the system's outputs $(\hat{y}_x)_{x \in \mathcal{X}}$ and the ground truth outputs $(y_x)_{x \in \mathcal{X}}$. Common metrics include BLEU (Papineni et al., 2002), METEOR (Denkowski and Lavie, 2014) which takes into account synonyms; or BERTScore (Zhang et al., 2019). This type of metrics are referred to as *reference-based*, as they require access to ground truth translations.

The second is concerned with assessing translation quality for source sentences for which one does not have reference translations. To determine whether machine-generated outputs are accurate enough or require human modification, one relies on multilingual quality estimation metrics (Specia et al., 2018). These do not rely on ground truth sequences; instead comparing produced samples to sources sentences directly. Expert human evaluation is perhaps the most relevant such score, but many automated alternatives exist (Martins et al., 2017); see e.g. Bhattacharyya et al. (2021). These metrics are referred to as *reference-less*.

Reference-based metrics provide high quality evaluation signal, and are well-suited to comparing average model performance (trusting that results on the unseen test set generalise to other domains of interest). However, they rely on the quality of the test set translations (which are usually unique, hence somewhat arbitrary), and cannot be used to evaluate the quality of models' prediction for specific unseen inputs. In contrast, reference-less metrics are harder to access or approximate but can be used without ground truth translations.

We use two reference-based metrics in our experiments: BLEU and BERTScore. We intro-

duce another, reference-less metric: Multilingual BERTScore. Note that while a translation model likelihood can be considered a reference-less metric, it comes with the unusual property that it is decomposable. We thus treat it as a special case.

BLEU (Papineni et al., 2002). The BLEU score computes modified precisions for n-grams (typically $1 \leq n \leq 4$) between a corpus of candidate sentences and a reference corpus. These precisions are averaged geometrically, and multiplied by a brevity penalty. This metric is meant to be used at the corpus level; it is unstable at the sentence level. It is the de facto gold standard for comparing MT algorithms, though as it crucially relies on access to a dataset of reference translations, it is not available to assess translation quality at decoding time.

BERTScore (Zhang et al., 2019). By contrast, BERTScore is a sentence-level metric to compare a candidate sentence to a reference translation. It relies on several consecutive steps: first, computing contextual embeddings for each token in both sentences with a shared BERT (Devlin et al., 2019) model; second, computing all pairwise cosine similarities between embeddings of the two sentences; third, greedily aligning tokens based on these similarities; finally, averaging the similarities of the aligned tokens. Compared to BLEU, BERTScore is found to correlate slightly better with human judgement. Importantly for decoding purposes, it is a sentence-level metric (which is averaged to produce a corpus-level statistic).

Multilingual BERTScore. While BERTScore is designed as a monolingual metric, we repurpose it as a multilingual one by using it to compare a candidate to its *source sentence* (instead of reference translation). Both sentences are in different languages, but this is fine as long as the underlying BERT model is itself multilingual.[1] We call this new metric Multilingual BERTScore. Its performance relies on the underlying BERT model's ability to map related tokens in different languages to similar embeddings. Because of the one-to-one nature of the alignment phase, we expect it to score more highly translation pairs that have a one-to-one token correspondence, rather than syntactically different pairs. We thus expect it to make sense for pairs of syntactically similar languages. We stress that we do not advocate widespread adoption of this imperfect score for MT; we consider it however a convenient illustrative example of a reference-less metric. In practice, we observe that it behaves reasonably for our two evaluation language pairs (WMT '14 English/German and English/French) as shown in Table 1. Interestingly, it scores trained model outputs higher than ground truth outputs. We hypothesise that the former follow the source sentence more closely than the latter.

[1] We use the multilingual, 12-layer, 768 hidden dimensionality BERT model available at https://github.com/google-research/bert (23/11/2018 entry).

Dataset	Random	GT	Greedy	Beam search
ENDE	68.13	81.36	83.02	83.40
ENFR	68.00	82.83	83.85	84.00

Table 1: Multilingual BERTScore for random text, ground truth sentences (GT), greedy and beam search decoding from a supervised model (guided by likelihood). We see that (i) random translation scores are lower than those of reference translations or that of supervised policy samples (though it is not 0, as for standard BERTScore, as both rely on continuous embeddings similarities); (ii) the scores are smaller than the corresponding monolingual BERTScore (87.88 and 90.55 for greedy decoding from a supervised model, respectively) and (iii) beam search outputs outperform greedy outputs consistently (as is the case for BLEU).

4 Decoding algorithms

In this section, we go over the details of each algorithm, and its adaptations to better suit reference-based or reference-less metrics. We separate them in three categories: (i) algorithms based on likelihood maximisation, (ii) value-based mechanisms which rely on approximating metrics via a value function, and (iii) ranking-based methods which access the metrics directly and pick the highest-scoring example out of a pool of finished candidates. Of course, ranking-based methods are only usable for reference-less metrics, as reference-based metrics are not computable at test time. We provide a high-level comparison of all algorithms in Table 2.

4.1 Likelihood-based decoding

Greedy decoding (GD) is our first baseline. It consists in picking the token with maximum likelihood at each step.

Sampling-based approaches – e.g. nucleus or top-k sampling – are not considered because they do not perform better than plain GD. Indeed, we found that the best hyperparameters for these methods make them equivalent to GD. This is not very surprising on a heavily conditioned task such as

MT.

Beam search (BS) maintains a beam of k possible translations prefixes at each time step t, $(p_t^i)_{i=1}^k$. Prefixes are updated incrementally as follows: for each prefix p_t^i one adds each of the corresponding k most probable tokens (given p_t^i), resulting in at most $k \times k$ new prefixes of size increased by 1. Then among these the k prefixes with the highest likelihood are selected, thus obtaining $(p_{t+1}^i)_{i=1}^k$. This method aims at optimising likelihood, and is agnostic to any metric of interest. It is therefore at a disadvantage if we change the objective of the search. Consequently, we also study the performance of value- or score-based variants.

4.2 Value-based decoding

To motivate the introduction of value functions in our decoding mechanisms, it is helpful to understand how machine translation can be construed of as a Reinforcement Learning task, with an underlying Markov Decision Process (MDP). In MT, we work with a vocabulary $\mathcal{V}$ of tokens, and a dataset contains pairs of sentences (x, y_x) where $x, y_x \in \mathcal{V}^+$. We can define a simple MDP, where:

- the states consist in a pair containing a source sentence $x \in \mathcal{V}^+$ and a sample in construction $\hat{y}_1 ... \hat{y}_t \in \mathcal{V}^+$,
- the action space $\mathcal{A}$ is the output vocabulary $\mathcal{V}$ (taking an action means adding a specific token to the sample),
- the transitions are deterministic: picking token $\hat{y}_{t+1} \in \mathcal{A}$ in state $s_t = (x, \hat{y}_1 ... \hat{y}_t)$ leads to the unique possible successor state $s_{t+1} = (x, \hat{y}_1 ... \hat{y}_t \hat{y}_{t+1})$,
- the reward is 0 for any non-terminal state; for terminal states, it is $m(y_x, \hat{y})$ for reference-based metrics and $m(x, \hat{y})$ for reference-less metrics of interest. Entering a terminal state is done by picking a special `<EOS>` token.

A *value function* v for a policy π approximates the final score one might expect to obtain, starting from a non-terminal state s, and following π thereafter. It thus provides *forward-looking* guidance *during* decoding, as opposed to likelihood (accessible during decoding but myopic) or a score (only computable on finished sentences).

Value-guided beam search (VGBS), as developed by He et al. (2017); Ren et al. (2017), augments the decision mechanism in beam search (when picking the top k prefixes amongst $k \times k$ candidates) with a value network v. The internal score is a linear combination between the (length-normalised) log-likelihood of a prefix and its value approximated by the value network, with a contribution factor α: $bs(s_t, a_t) = \frac{\alpha}{t} \log\big(\pi(s_t, a_t)\big) + (1-\alpha) v(s_t, a_t)$.[2] Note that this method does not use the score; thus it is applicable to both reference-less and reference-based metrics.

Value-guided MCTS (V-MCTS) indicates the version of Monte Carlo Tree Search as used by Silver et al. (2017). This search method combines both a policy π and a value network v. For every decoding step, a fixed budget of simulations is allocated to build a tree of possible future trajectories. Each simulation consists in 3 steps:

- selection: recursively picking children nodes according to the pUCT formula, starting at the root and until reaching an unopened (i.e. not expanded yet) node s_o:
$$\operatorname*{argmax}_{a \in \mathcal{A}} \left(Q(s,a) + c_{\text{puct}} \pi_\tau(a|s) \frac{\sqrt{\sum_b N_s^b}}{1 + N_s^a} \right)$$
where $Q(s,a)$ is a statistic representing the value of taking action a in state s, updated online during the search, c_{puct} is a tunable constant, τ is a temperature parameter applied to the policy $\pi_\tau(a|s) = {\pi(a|s)^{1/\tau}}/{\sum_b \pi(b|s)^{1/\tau}}$ and N_s^a is the number of times action a has been chosen from state s while building the tree (also called visit count);
- expansion: opening the selected node s_o by computing the policy $\pi(a|s_o)_{a \in \mathcal{A}}$ at the associated new state, as well as the value $v(s_o)$;
- backup: updating the Q statistics encountered during the tree traversal leading to s_o via an aggregation mechanism (such as *averaging* the previous statistic with $v(s_o)$, or taking their *maximum*: $Q \leftarrow \max(Q, v(s_o))$).

Once the tree is finished, the decision for the current decoding step is made according to the statistics of the root's children nodes. A popular option consists in picking the root child with the *most visit counts*, but one may also select the one with *maximum aggregated value* instead.

While it is customary to allow MCTS to use the score directly when encountering a terminal

[2] Using the logarithm of the value instead, as in He et al. (2017), yields no practical gains. So we opt for the simpler formulation.

state, we opt for a pure value implementation instead (i.e. using the value instead of the score on terminal nodes). This makes V-MCTS applicable to reference-based metrics, which it wouldn't be otherwise.

One of the keys to successful MCTS performance is properly balancing the breadth and depth of the exploratory trees. We found two adaptations to be helpful. First, we used an adaptive value scale as described by Schrittwieser et al. (2020, Appendix B): in the selection phase, we rescale $Q(s, a)$ in the $[0, 1]$ interval by replacing it with $\frac{Q(s,a)-\min Q}{\max Q-\min Q}$, where $\min Q$ and $\max Q$ correspond to the minimum and maximum value observed in the tree, updated online. Second, we tune the logits temperature τ jointly with the c_{puct} hyperparameter.

4.3 Reranking-based decoding

Value-driven decoding methods are well-suited to optimise metrics which we cannot evaluate at test time, such as reference-based metrics. One might also prefer them for especially expensive reference-less metrics, e.g. expert human evaluation. For tractable reference-less metrics though, we can directly compute the scores of finished candidate sentences, without having to resort to approximation. We study two specific decoding mechanisms that take advantage of this option.

Sampling and reranking (S+R, S+RV) consists in sampling a fixed number of finished candidate sentences $\hat{y}^1, ..., \hat{y}^n$ from the policy (with a carefully tuned temperature applied to its logits), scoring all of them and picking the highest-performing one: $\text{argmax}_{i=1}^{n} m(x, \hat{y}^i)$. To measure the loss of performance associated with using a value, we also introduce a variant, S+RV that ranks candidates according to the value (rather than the score).

MCTS with rollouts (MCTS+Roll) is a variant of V-MCTS where we replace the value approximation for a given node s by a more expensive one based off the actual score. From s, we perform a greedy rollout (w.r.t. the policy π) until we arrive at a terminal node s_T. We then compute the score with the finished sample and the source as inputs, use this scalar as the value of node s, and continue as in V-MCTS. Of course greedy rollouts are expensive in MT, so this method is not directly comparable to V-MCTS. It is however useful as a proof of concept which enables us to measure how much performance we lose by relying on a value function rather than directly on the score.

	Uses a value	Uses the score directly
Greedy	✗	✗
Beam Search	✗	✗
VGBS	✓	✗
V-MCTS	✓	✗
S + RV	✓	✗
S + R	✗	✓
MCTS + Roll	✗	✓

Table 2: Decoding algorithms characteristics.

4.4 Complexity analysis

For all the algorithms we consider, the complexity of the methods themselves – both in time and space – is negligible compared to the cost of running the necessary neural network inference steps. We can thus reduce the complexity analysis to this simple question: how many inference steps does each method require?

For a sequence of length T, greedy decoding requires only T inferences (here we're assuming that the lengths of both input and output sequences are equal, which is a reasonable assumption for comparing complexities). It is the cheapest method. Beam search requires $k \times T$ inferences, where k is the beam size, making it k times more expensive. Value-guided beam search adds calls to the value network for all considered extensions, which means its final complexity is $k^2 \times T$. V-MCTS on the other hand uses the same network call to compute both policy and value, and hence requires $S \times T$ inference steps, where S is the simulation budget per token. Sampling-based methods require $N \times T$ network inferences, where N is the number of samples taken per sequence.

To ensure fair comparison between methods we make it so the total number of inferences is the same across algorithms, except for methods whose performance degrades with more inferences (such as plain beam search).

5 Training a value network

Several of the algorithms we detail in Section 4 make use of a value network. We train these by:

- First, training a plain supervised policy model π_{sup} on our bilingual datasets.
- Second, updating each data item (x, y_x), which contains a source x and a reference sentence y_x, by replacing y_x with a sample $\hat{y}_x$ ob-

tained via greedy decoding[3] from our trained policy π_{sup}, and adding a score m comparing either $\hat{y}_x$ to y_x (for reference-based metrics) or $\hat{y}_x$ to x (for reference-less metrics).

- Finally, training a dual-headed network on the augmented dataset, with a shared transformer encoder-decoder torso (Vaswani et al., 2017) taking source x and sample $\hat{y}_x$ as inputs, and two heads, one predicting the policy π_d and the other the value v. This approach provides a powerful regulariser for the value, greatly reducing its tendency to overfitting (Silver et al., 2017).

The second step is mandatory to obtain a score distribution to train the value model on, in the case of reference-based metrics. Indeed, the scores of the optimal supervised policy are all perfect (comparing y_x to y_x), thus uninformative, making it impossible to train a value network on. Relying on a sample rather than on the ground truth sentence to compute the score has another advantage: the samples follow the policy π_{sup} so the value will be the one associated with a trained policy, as the one we use during decoding, rather than with the optimal supervised policy.

Losses. We train the policy by minimising its Kullback-Leibler divergence with the initial supervised policy π_{sup}: $\mathcal{L}_\pi = D_{\text{KL}}(\pi||\pi_{\text{sup}})$.

We reframe the value regression problem as classification by discretising the score interval into buckets. We emulate training our value function on unfinished samples by adding a value loss term at every step, and reusing the transformer decoder causality mask. The prediction target is the same across all time-steps: the one-hot vector indicating in which bucket the final score m falls. We use the cross-entropy loss function.

The trouble with reference-based metrics. In practice, we find that learning a value function for reference-based metrics (such as BLEU or BERTScore) is difficult. To understand why, we run an ablation to distinguish between the three subtasks a value function must perform: (i) approximate the score, (ii) predict the end of a trajectory from an unfinished prefix, and (iii) assess the translation quality of a pair of finished sentences in different languages. To separate concerns, we run the following experiments: for (i), we train our network to predict BLEU given ground truth targets (rather than source sentences) and finished samples (instead of prefixes). For (ii), we give the network ground truth targets and *unfinished samples*. Finally, for (iii), we give the network source sentences and finished samples (thus removing the need to predict the future of trajectories). We observe that: the error is very low for (i); higher, but significantly improved over the full setup for (ii); and surprisingly, roughly identical to the full setup for (iii). Thus the real difficulty lies in (iii).

	Decoder inputs	
Encoder inputs	Sample prefix	Finished sample
Source	0.112 (full setup)	0.111 (iii)
Reference Target	0.065 (ii)	0.002 (i)

Table 3: ℓ_1 error of BLEU value networks trained on different encoder inputs and outputs on our sample dataset based on WMT14 ENDE. Scores (and hence ℓ_1 error) are between 0 and 1.

One possible explanation for this result is that the value network is missing a key input. Indeed, in the case of reference-based metrics, the score is computed between a sample $\hat{y}_x$ and a ground truth reference y_x; but the value network only has access to the source sentence x and a prefix of $\hat{y}_x$. Thus before it can compute a precise score approximation, it first has to infer y_x from x. But of course, inferring y_x from x is exactly the original machine translation problem, which makes the value problem empirically harder than its policy counterpart on our dataset.

6 Experiments

We detail our general setup, then report results for all 3 metrics we consider, and finally study how they scale with increasing search budget. Detailed tuning methodology can be found in Appendix B.

6.1 Experimental setup

We consider two established machine translation datasets: WMT'14 English to German (ENDE) and WMT'14 English to French (ENFR). The first dataset contains roughly 4.5 million training sentence pairs, while the second is much bigger with just under 41 million training sentence pairs, which enables us to account for scale in our experiments. All dev and test sets contain about 3000 sentences.

Our joint policy/value model is based on the Transformer encoder-decoder (Vaswani et al., 2017), which is typically used in machine translation studies. Encoder and decoder have 6 atten-

[3] We also tried sampling one or more sentences instead, but did not detect any improvements from doing so.

	ENDE			ENFR		
Target score	BLEU	BERTScore	MLBERTScore	BLEU	BERTScore	MLBERTscore
Random	0	69.59	68.13	0	70.55	68.00
Ground Truth	100	100.0	81.36	100	100.0	82.83
Vaswani et al. (2017), beam search	27.30	–	–	38.10	–	–
Greedy	25.99	87.88	83.02	38.70	90.55	84.22
Beam search	**27.75**	**88.48**	83.40	39.24	90.76	84.48
VGBS	27.17	88.47	**85.10**	**39.33**	90.87	85.80
S+R (value)	26.03	88.39	84.49	38.67	**90.93**	85.68
V-MCTS	27.47	88.45	84.97	39.12	90.80	**86.31**
S+R (score)	–	–	85.11	–	–	86.12
MCTS + rollouts	–	–	**85.76**	–	–	**86.87**

Table 4: Comparison of decoding mechanisms on ENDE and ENFR. Top row contains general metric statistics and the original transformer baseline; the second row performance of supervised models with likelihood-based decodings; the third results for value-based algorithms with joint policy/value models (a specific one for each metric); the last one numbers for score-based methods. Best overall performance is in bold; best value-based performance in blue. Beam search performs strongly for reference-based metrics, while value-based methods prevail for reference-less scores. Score-based methods outperform their value-based counterparts, but V-MCTS remains competitive.

tion blocks, hidden dimensionality 512, 16 heads and our dictionary size is about 32k. As we test inference-intensive methods, we use a few adaptations detailed in Appendix A.

Finally, we allow a budget of 50 inferences per token in the sampled solutions for all methods; compared to 1 for greedy decoding, and 4 for beam search. We use incremental sampling for speed. Both running time and memory footprint are directly proportional to the amount of inferences for all methods.

6.2 Main results analysis

Reference-based metrics: BLEU and BERTScore. We report our results on reference-based metrics in Table 4. Plain beam search is a strong contender in this setup, often matching or outperforming other methods, while using a fraction of the inference budget (unfortunately performance degrades rapidly with larger beam sizes so we cannot leverage more compute). In this setup, value-based methods struggle to justify their higher complexity and cost.

Value-based algorithms for reference-less metrics. The results for this alternative use case, also presented in Table 4, paint a completely different picture. We see that while regular beam size obtains a small but consistent improvement, value-guided methods perform significantly better. Between the latter, MCTS is particularly promising, as its performance scales nicely with the size of the dataset.

We observe that the policies of our joint policy/value models perform slightly worse than their supervised counterparts (see Table 12 in Appendix). If we use the initial supervised model policy in conjunction with the multilingual value (see Table 13 in Appendix), we obtain promising results: notably 40.31 BLEU when optimising MLBERTScore with MCTS on the ENFR dataset – more than a full BLEU point above the performance of beam search. From a qualitative point of view, we see a confirmation of our conjecture: multilingual BERTScore is not perfectly aligned with BLEU. It seems to encourage word-for-word translations, which has a positive effect initially (more consistency between the source and the sample sentences), but ultimately leads to less natural translations if used with enough budget.

Score-based approaches for reference-less metrics. The bottom of Table 4 gives results when we allow direct access to our two reference-less metrics, without having to go through a value approximation. They reinforce our finding that the choice of algorithm heavily depends on the use case.

Two additional properties stand out. First, all the methods that access the score directly perform significantly better than their value-guided counterparts.

Second, the purely value-based V-MCTS is competitive with and can even outperform the score-based approach, S+R. This is promising, as MCTS is more widely applicable (as some scores are expensive to get). However S+R performs surprisingly well, which may warrant more explorations of sampling methods optimising for diversity (e.g. Fan et al. (2018); Holtzman et al. (2019); Kool et al. (2019)).

6.3 Scaling search with computating budget

We study how our decoding algorithms scale with their search computational budget. We report results on MLBERTScore in Table 5 and more de-

Data	ENDE			ENFR		
Alg.	VGBS	S+R	MCTS	VGBS	S+R	MCTS
1	82.82	81.77	82.82	84.22	82.94	84.22
10	84.48	84.25	84.47	85.38	85.35	85.76
25	84.87	84.77	84.81	85.69	85.82	86.10
50	85.10	85.10	84.97	85.80	86.12	86.31
75	85.34	85.31	85.07	85.75	86.33	86.44
100	85.39	85.40	85.14	85.78	86.44	86.51
200	85.64	85.69	85.27	85.77	86.76	86.62
300	85.79	85.89	85.27	85.79	86.90	86.66

Table 5: Comparison of how our methods scale with search budget (from 1 to 100 inferences per token) on ML-BERTScore.

tailed numbers in Appendix C. Our findings are fairly unsurprising: the more quality score data algorithms can leverage, the better they scale. We see that on reference-based metrics – where value networks are hard to train and thus quite imperfect – performance quickly stops increasing with more computation and start degrading instead. When using higher quality value networks (in the reference-less metrics setup), performance increases more steadily with computation (almost everywhere), plateauing rather than degrading. Finally, when accessing the score directly (for the ranking approaches), the more computation, the better and performance keeps increasing with more inferences.

7 Discussion

The main takeaway from our experiments is that which algorithm is best depends heavily on the metric to optimise. This reinforces the notion that one should carefully consider when picking a decoding mechanism for a machine translation pipeline, rather than default to beam search.

Second, we find that optimising reference-based metrics (e.g. BLEU) via a value function is surprisingly hard. While distinguishing large gaps in quality is easier than modelling the policy, discriminating between good candidates is in practice as hard as the policy problem, since a first required step is estimating the ground truth sentence. Indeed, empirically we observe relatively low quality value networks, and comparatively little improvements with value-based decoding methods (especially on the small ENDE dataset). Using a value function to optimise reference-less metrics is more promising.

Third, we show that MCTS is not only a valid way of decoding for machine translation tasks, but also the best option in some use cases. We study its strengths and weaknesses, and demonstrate that its performance is crucially linked to the ease of learning a good value function. We include pseudo-code for an easily reproducible Numpy implementation in Appendix F. All told, we provide a blueprint for how to use MCTS efficiently in NLP with state-of-the-art transformer models.

Finally and somewhat surprisingly, we find that whenever access to the score is possible, the simple S+R method performs well. More experimentation is required to understand why; but at any rate, it should be a strong contender in this specific setup.

Future directions. We have shown that optimising for reference-less metrics is easier than for reference-based ones. The ultimate reference-less metric for machine translation is human translation assessment. Thus it seems natural to consider training a score directly from human evaluation of translation pairs, and to later focus on optimising it via MCTS.

Another natural extension is a full-blown RL algorithm; iteratively improving policies via value-guided search and training value functions on search-improved policies, getting closer to the optimal policy and value at each step.

Acknowledgements

We want to thank our colleagues David Silver, Chris Dyer, Wang Ling, Julian Schrittwieser and Thomas Hubert for fruitful conversations, guidance, contributions to the paper and help with technical support.

References

David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. 1985. A learning algorithm for boltzmann machines. *Cognitive Science*.

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. 2017. An Actor-Critic Algorithm for Sequence Prediction. In *Proceedings of the 5^{th} International Conference on Learning Representations (ICLR)*.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems 28 (NIPS)*.

Sumanta Bhattacharyya, Amirmohammad Rooshenas, Subhajit Naskar, Simeng Sun, Mohit Iyyer, and Andrew McCallum. 2021. Energy-based reranking: Improving neural machine translation using energy-based models. *arXiv*.

Sebastian Borgeaud and Guy Emerson. 2019. Leveraging sentence similarity in natural language generation: Improving beam search using range voting. *arXiv*.

Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*.

Leshem Choshen, Lior Fox, Zohar Aizenbud, and Omri Abend. 2019. On the weaknesses of reinforcement learning for neural machine translation. In *Proceedings of the 8^{th} International Conference on Learning Representations (ICLR)*.

Eldan Cohen and J. Christopher Beck. 2019. Empirical analysis of beam search performance degradation in neural sequence models. In *Proceedings of the 36^{th} International Conference on Machine Learning (ICML)*.

Ronan Collobert, Awni Hannun, and Gabriel Synnaeve. 2019. A fully differentiable beam search decoder. In *Proceedings of the 36^{th} International Conference on Machine Learning (ICML)*.

Rémi Coulom. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5^{th} International Conference on Computers and Games (ICCG)*.

Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the EACL 2014 Workshop on Statistical Machine Translation*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*.

Sergey Edunov, Myle Ott, Michael Auli, David Grangier, and Marc'Aurelio Ranzato. 2018. Classical structured prediction losses for sequence to sequence learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Bryan Eikema and W. Aziz. 2020. Is map decoding all you need? the inadequacy of the mode in neural machine translation. *ArXiv*, abs/2005.10283.

Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

J. Gu, J. Bradbury, C. Xiong, V. O. Li, , and R. Socher. 2018. Non-autoregressive neural machine translation. In *Proceedings of the 6^{th} International Conference on Learning Representations (ICLR)*.

E. J. Gumbel. 1954. Statistical theory of extreme values and some practical applications: a series of lectures.

Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature*.

Di He, Hanqing Lu, Yingce Xia, Tao Qin, Liwei Wang, and Tie-Yan Liu. 2017. Decoding with value networks for neural machine translation. In *Advances in Neural Information Processing Systems*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. In *Proceedings of the 7^{th} International Conference on Learning Representations (ICLR)*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

L. Kocsis and C. Szepesvári. 2006. Bandit based monte-carlo planning. In *Proceedings of the 15^{th} European Conference on Machine Learning (ECML)*.

Wouter Kool, Herke Van Hoof, and Max Welling. 2019. Stochastic beams and where to find them: The Gumbel-top-k trick for sampling sequences without replacement. In *Proceedings of the 36^{th} International Conference on Machine Learning (ICML)*.

Rémi Leblond, Jean-Baptiste Alayrac, Anton Osokin, and Simon Lacoste-Julien. 2018. SEARNN: training rnns with global-local losses. In *Proceedings of the 6^{th} International Conference on Learning Representations (ICLR)*.

André F. T. Martins, Marcin Junczys-Dowmunt, Fabio N. Kepler, Ramón Astudillo, Chris Hokamp, and Roman Grundkiewicz. 2017. Pushing the limits of translation quality estimation. *Transactions of the Association for Computational Linguistics*.

Clara Meister, Ryan Cotterell, and Tim Vieira. 2020a. If beam search is the answer, what was the question? In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2173–2185. Association for Computational Linguistics.

Clara Meister, Tim Vieira, and Ryan Cotterell. 2020b. Best-first beam search. *Transactions of the Association for Computational Linguistics*.

Mohammad Norouzi, Sammy Bengio, Zhifeng Chen, Navdeep Jaitly, Mike Schuster, Yonghui Wu, and Dale Schuurmans. 2016. Reward Augmented Maximum Likelihood for Neural Structured Prediction. In *Advances in Neural Information Processing Systems 29 (NIPS)*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the* 40^{th} *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence Level Training with Recurrent Neural Networks. In *Proceedings of the* 5^{th} *International Conference on Learning Representations (ICLR)*.

Raj Reddy. 1977. *Speech understanding systems: A summary of results of the five-year research effort*. Carnegie Mellon University.

Z. Ren, X. Wang, N. Zhang, X. Lv, and L. Li. 2017. Deep reinforcement learning-based image captioning with embedding reward. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need. *arXiv*, abs/1911.02150.

Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. 2016. Minimum Risk Training for Neural Machine Translation. In *Proceedings of the* 53^{rd} *Annual Meeting of the Association for Computational Linguistics (ACL)*.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature*.

Lucia Specia, Carolina Scarton, and Gustavo Henrique Paetzold. 2018. Quality estimation for machine translation. *Synthesis Lectures on Human Language Technologies*.

Felix Stahlberg and Bill Byrne. 2019. On NMT search errors and model errors: Cat got your tongue? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

Yu-Siang Wang, Yen-Ling Kuo, and B. Katz. 2020. Investigating the decoders of maximum likelihood sequence models: A look-ahead approach. *ArXiv*, abs/2003.03716.

Sean Welleck, Kiante Brantley, Hal Daumé, III, and Kyunghyun Cho. 2019. Non-monotonic sequential text generation. In *Proceedings of the* 36^{th} *International Conference on Machine Learning (ICML)*.

Sean Welleck, Ilia Kulikov, Jaedeok Kim, Richard Yuanzhe Pang, and Kyunghyun Cho. 2020. Consistency of a recurrent language model with respect to incomplete decoding. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*.

Lei Yu, Laurent Sartran, Po-Sen Huang, Wojciech Stokowiec, Domenic Donato, Srivatsan Srinivasan, Alek Andreev, Wang Ling, Sona Mokra, Agustin Dal Lago, Yotam Doron, Susannah Young, Phil Blunsom, and Chris Dyer. 2020. The DeepMind Chinese–English document translation system at WMT2020. In *Proceedings of the Fifth Conference on Machine Translation*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2019. BERTScore: Evaluating text generation with BERT. In *Proceedings of the* 7^{th} *International Conference on Learning Representations (ICLR)*.

Outline

Appendix A provides details about the network architectures and the optimisation hyper-parameters used in our work. We also describe the hybrid architecture that is used to improve the training of value networks in the hard case of *reference-based* metrics. In Appendix B, we lay out the details of the tuning of the different decoding mechanisms used throughout the paper. In Appendix C, we give results about how performance of our decoding algorithms scale with more computational budget. In Appendix D, we discuss the trade offs between learning a joint policy and value network on sampled (or distilled) trajectories, versus training a separate value network to predict the value of a policy network trained on supervised trajectories. In Appendix E, we give a few examples of MCTS exploratory trees. Finally, in Appendix F we provide a simple implementation of a batched version of MCTS in plain Numpy.

A Network architectures and training

In this section, we detail our basic dual-headed architecture, our training regimen and our optimisation hyper-parameters. We also describe our hybrid architecture (which we use for reference-based metrics) in more depth.

Dual-head transformer architecture. We start from the original transformer encoder-decoder architecture (Vaswani et al., 2017), with a few modifications. Both encoder and decoder have $\text{num\_layers} = 6$ attention layers. The hidden size is 512. The embedding vocabulary size is just short of 32000 tokens for both language pairs. The unroll length of our models is 128. We use "normal GPT-2"-style initialisers, i.e. initial values are sampled from a Gaussian distribution with mean 0 and standard deviation $\frac{0.02}{\sqrt{\text{num\_layers}}}$. The one exception to this rule is for embeddings, where we use truncated normal initialisers with standard deviation 1.0.

On top of the decoder, we add two "heads". The first one is the policy head. It consists in a linear projection from the hidden dimensionality to the vocabulary size, followed by a softmax operator to output a distribution over the whole vocabulary. The second head is the value head: a linear projection from the hidden dimensionality to the amount of value buckets we define ($|\mathcal{B}| = 500$ in our experiments), followed by a softmax operator. We compute the value loss as the cross-entropy between the softmax distribution $(v_i)_{i\in\mathcal{B}}$ and a one-hot encoding of the target value of the same dimension. To output the value, we compute the sum of the softmax distribution multiplied by the average value in each bucket $\sum_{i\in\mathcal{B}} v_i \bar{b}_i$.

Compared to the original architecture, we apply several changes related to inference speed. First, instead of 8 attention heads we use 16. Second, the dimensionality of the keys and values is 128, compared to 64 in the original architecture, thus avoiding a costly padding operation on our hardware accelerators, TPUv3. Finally, we use multi-query attention (Shazeer, 2019), only computing a single set of keys and values per attention block and sharing them across all attention heads. This reduces the memory footprint of the keys and values by a factor of the number of attention heads (16 here), considerably decreasing the time spent reading and writing from memory, which ultimately results in a near-linear inference speedup with respect to the number of attention heads (as somewhat counter-intuitively, the performance bottleneck for small transformer architectures on our hardware of choice, TPUv3, is memory access by a very large margin). As this alternative attention mechanism requires less trainable weights than the more conventional one, we reallocate some of those in the feedforward layer of the attention blocks by using a bigger internal hidden dimensionality of 3072 instead of 2048.

Optimisation. We use the Adam (Kingma and Ba, 2015) optimiser with learning rate 0.001, and the following hyper-parameters: $b_1 = 0.9; b_2 = 0.98; \epsilon = 1e^{-9}$. Our batch size is 4096, and we train for 100000 steps for the ENDE dataset and 300000 steps for the larger ENFR dataset. As regularisation, we use dropout with a weight 0.1, but no weight decay. We also use label smoothing with hyper-parameter 0.1 (although we see little impact when removing it).

The last difference with the original transformer encoder-decoder is where we place the layer norm operator. We put it at the beginning of the attention and the feed-forward layers, rather than at the end, which allows for fully-residual layers.

Hybrid architecture for reference-based metrics. In Section 5, we show that learning good value functions on reference-based metrics such as BLEU and BERTScore is very difficult. This is mainly due to the fact that our value networks are

lacking access to the ground truth targets which are required for precise score computation. Our ablation study show that if we remove this difficulty by allowing the value model to "cheat" by using the ground truth targets rather than the source sentences as encoder inputs, we obtain much more precise values. Downstream results using MCTS with such a "cheating" value show very large BLEU improvements. Unfortunately, in practice one cannot rely on such a trick when decoding.

Our idea is to try to leverage "cheating" information indirectly at training time to shape the representation of a regular (i.e. non-cheating) value network. Another way to look at it is that we try to distill the knowledge of the cheating value model into the regular one.

To achieve this, we propose a new training regimen, as detailed in Figure 1. The basic idea is to compute the final layer of a cheating value model, and to use it as an auxiliary target for the final layer of a regular value model, in addition to its normal value loss. We thus have two pathways. On the left, the regular value model encoder receives the source sentence as input (which is available at test time). On the right, the cheating value model encoder receives the ground truth target sentence as input (which is **not** available at test time). For both pathways, the decoder's input is a sample sentence. Crucially, both pathways rely on the same transformer encoder-decoder: they share all weights, the only difference is in their inputs.

To train such a model, we use four losses. First, we apply the regular policy $\mathcal{L}_\pi$ and value loss $\mathcal{L}_v$ on the regular pathway. Second, we apply a value loss $\mathcal{L}_{v_c}$ on the cheating pathway. Finally, we add an ℓ_2 loss $\mathcal{D}$ between the final layer of both pathways, with a stop gradient for the cheating one – so that its representation is not directly affected by $\mathcal{D}$.

We do not add a policy loss on the cheating pathway. This seems natural, as such a loss would only encourage the model to reproduce its inputs exactly, effectively pushing it towards the identity function.

At inference time, we only compute the regular pathway, which does not cheat. In practice, with careful tuning of the loss hyper-parameters we are able to significantly reduce the gap in performance between this new hybrid model and the cheating one (as in (ii)), so we use this training regimen for our experiments on reference-based metrics.

Using such a hybrid architecture yields performance improvements when using the value model with MCTS, as shown in Table 6.

Architecture	Normal	Cheating	Hybrid
Greedy	26.11	–	25.99
MCTS	26.40	(38.52)	27.47
Greedy	87.92	–	87.88
MCTS	88.01	(90.70)	88.48

Table 6: Greedy vs MCTS (50 simulations) performance on the ENDE dataset for BLEU (top rows) and BERTScore (bottom rows). Using the normal architecture, improvements are very small. Using the hybrid architecture yields more significant improvements. The middle column contains (greyed-out) results when using a cheating model which takes ground truth targets as inputs. Improvements are enormous in this prohibited setting, which is unsurprising at the value function receives optimal output as its own inputs.

We find that to obtain best performance, three things need to be combined: (i) sharing weights across both pathways, (ii) the distillation ℓ_2 loss and (iii) the cheating value loss. Each loss is added with a linear weight. Proper tuning of these weights is important. We find that using weights 1.0 for $\mathcal{L}_\pi$ and $\mathcal{L}_{v_c}$, as well as 0.1 for $\mathcal{L}_v$ and $\mathcal{D}$ leads to best performance.

B Experimental details and ablations

B.1 Algorithms tuning and ablations

We detail how we tuned each algorithm for best performance in this section, yielding Table 4. We used the dev datasets to determine which options were best. Wherever we report numbers, we compute those on the test set for comparison purposes, but the experiments were run *after* dev set selection. In practice, we find a very good correlation between observations on the dev and on the test sets (although absolute values were lower on the dev set, rankings remained mostly unchanged). Unless otherwise indicated, our findings hold for both ENDE and ENFR datasets, and for all 3 metrics we consider (BLEU, BERTScore, Multilingual BERTScore).

Beam Search. As this method is known for degrading with large beam sizes, we add a length normalisation term, as advocated by Wu et al. (2016). The resulting score for a candidate $y_1..y_t$ is thus: $bs(y_1..y_t) = (\frac{6}{t+5})^\theta \log\big(\pi(y_1..y_t)\big)$

We tuned three hyper-parameters for beam

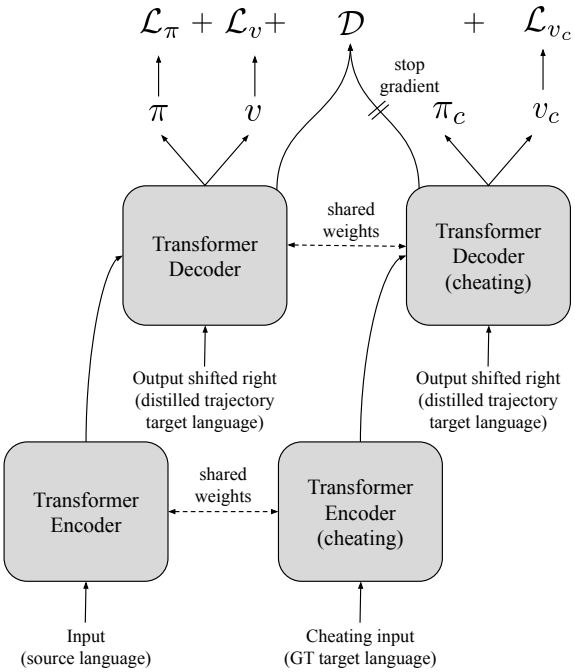

Figure 1: **Hybrid architecture.** Since learning a good value network is hard when dealing with *reference-based* metrics, we propose a training mechanism to distill the information from a *cheating* network that has access to the full *reference* information in order to predict the value score. In details, we have two encoder/decoder networks that share their weights. The first network (*left*) is a regular network that takes as input the input sentence in the source language and is trained to output (i) a policy predicting the words of the output sentence in the target language and (ii) a value score for that output via a policy and a value loss $\mathcal{L}_\pi$ and $\mathcal{L}_v$, respectively. The second network (*right*), dubbed cheating, is given as input the ground truth sentence in the target language and is trained to output the value score of the output sentence in the target language against that ground truth via the loss $\mathcal{L}_{v_c}$. This simplifies its task considerably, as it has direct access to the *reference* information to compute the value. A distillation loss $\mathcal{D}$ is added to transfer knowledge from the cheating model to the regular network. Results in Table 6 shows that such an approach yields significant improvements by easing the training of the value network.

	ENDE	ENFR
2	27.22	38.88
4	27.75	39.24
6	**28.03**	**39.37**
8	28.03	39.29
10	28.01	39.29
20	28.02	39.21

Table 7: BLEU results as a function of beam size for plain beam search.

search:

- the beam size: we tried 2, 4, 6, 8, 10 and 20. We find that the best performance is attained at 6, plateaus until 10 and starts slowly decreasing by 20 (see Table 7).

- the logits temperature: we tried 0.6, 0.8, 1.0, 1.2, 1.4. 1.0 performs best by a wide margin. Low values degrade to greedy search performance while high values yield non-sensical sentences.

- the normalisation temperature parameter θ: we tried 0.4, 0.6, 0.8 and 1.0. We find $\theta = 0.6$ performs best, as in the original paper ($\theta = 0.4$ is on par but slightly worse, and performance degrades as soon as $\theta \geq 0.8$).

We found that the best performance was achieved with the default hyper-parameters for most metrics.

Value-guided beam search. The score for a candidate $y_1..y_t$ for this method is:

$$bs(y_1..y_t) = \frac{\alpha}{t} \log\big(\pi(y_1..y_t)\big) + (1-\alpha) v(y_1..y_t).$$

The hyper-parameters to tune are slightly different than those of plain beam search. As we do not see performance decrease with larger beam size, we no longer need to find the optimal one. Which one to use is largely dependent on how much computation one can afford to use. Performance as a function of this quantity are reported in Section C.

Further, because of the additional value term, we have a new linear combination weight α to tune. Here are the hyper-parameter ranges we consider:

- the logits temperature: as for plain beam search, we tried 0.6, 0.8, 1.0, 1.2, 1.4. We find similar results: 1.0 is the best-performing

	ENFR	
Score	BERTScore	MLBERTScore
0.0	90.74	84.41
0.1	90.77	84.42
0.2	90.80	84.44
0.3	90.80	84.51
0.4	90.83	84.62
0.5	**90.84**	84.77
0.6	90.71	84.90
0.7	89.91	85.10
0.8	87.17	85.44
0.9	82.25	85.44
0.95	79.47	**85.55**
1.0	76.22	79.30

Table 8: VGBS performance on the ENFR dataset as a function of the linear weight α.

temperature by a significant margin; the reliance on the value mitigates the effect for reference-less metrics (where the value is a good approximation).

- the value normalisation: we tried using the logarithm of the value instead of the value itself, with appropriate scaling: $\log(\frac{v(y_1..y_t)-d}{D-d})$. We find that by carefully tuning the minimum and maximum bounds d and D we can match the performance we obtain with the plain value, but not outperform it. As a result we opt for the simpler formulation.

- the linear combination weights α: we swept between 0 and 1 by 0.1 increments (with an additional measure at 0.95). For reference-based metrics, we find $\alpha = 0.5$ to perform best (on both datasets). For Multilingual BERTScore on the other hand, much larger values are required to achieve best performance: $\alpha = 0.9$ for ENDE and $\alpha = 0.95$ for ENFR. Our hypothesis is that the quality of our value function is much higher for this last *reference-less* metric, which allows us to lean more heavily on its guidance. See Table 8 for illustrative results.

MCTS variants. MCTS is a complex algorithm with a large number of hyper-parameters. We found that three main aspects are important for performance: making sure the policy and the value terms are well-balanced in the UCT formula, picking the best value aggregation mechanism during the backup phase, and selecting the best acting criteria (once the tree is finished).

Action selection	argmax(vc)		argmax(v)	
Value aggregation	avg.	max.	avg.	max.
BLEU	**27.47**	26.82	22.48	25.52
MLBERTScore	83.76	83.81	84.08	**84.97**

Table 9: BLEU and Multilingual BERTScore performance when using different value aggregation mechanisms and action selection rules. We observe that on the reference-less metric, the best option rely more heavily on the value function; contrary to what we see for the reference-based metric.

To ensure balance in the UCT formula, we tuned two things:

- we optimised for the logits temperature τ and the multiplicative constant c_{puct} *jointly*. We tried temperatures 0.9, 1.1 and 1.3, in conjunction with c_{puct} in 1.0, 2.0, 3.0, 4.0, 6.0, 8.0. For reference-based metrics, the pair $(\tau = 0.9; c_{\text{puct}} = 3.0)$ performed best across both datasets and both metrics; while for Multilingual BERTScore the best performer was $(\tau = 1.1; c_{\text{puct}} = 8.0)$ across both datasets. Note that both larger temperature and larger c_{puct} encourage exploration in the UCT formula, thus reducing the relative weight of the policy in favour of the value; that we can use larger scalars for reference-less metrics is yet another indication that the associated value functions are more trustworthy.

- as we detailed in the main text, we rescale the values dynamically during the tree construction so that all the values encountered until the current step are more evenly distributed in the $[0, 1]$ interval by mapping the minimum value to 0 and the maximum value to 1.

We tested two value aggregation operators: running average and maximum. We also tried two action selection mechanisms: picking among the root's children nodes the one with maximum visit count, or the one with maximum aggregated value.

Our observation once again underline the contrast between reference-based and reference-less metrics, as is illustrated in Table 9. For the former, the best choice is to use the running average as value aggregation operator during the backup phase, and to pick the root child with maximum visit counts. Conversely, for Multilingual

	S+R	S+RV		
	MLBERT	BLEU	BERT	MLBERT
0.15	83.97	26.17	88.31	83.69
0.25	84.35	**26.23**	**88.32**	84.02
0.35	84.60	25.85	88.31	84.22
0.45	84.82	25.47	88.29	84.37
0.55	84.91	24.96	88.14	84.44
0.65	85.02	24.28	88.04	84.48
0.75	**85.11**	23.79	87.91	**84.48**
0.85	85.07	22.77	87.69	84.21
0.95	84.95	21.32	87.23	83.66

Table 10: Sampling + ranking performance as a function of policy logits temperature, for both score and value-based variants.

BERTScore we found that we obtained best performance with the maximum aggregation operator and by picking the root child with maximum aggregated value. We thus see that for our reference-less metric we can rely on the value function aggressively; while for reference-based metrics we need to limit our exposure to it.

Sampling + Ranking variants. For these algorithms, we really only have a single hyperparameter to tune: the policy temperature τ. Small temperatures lead to little diversity across different samples, but ensure that samples are highly ranked by the model and hence are syntactically correct. On the other hand, large temperature encourage diversity, at the price of correctness. We sweep over the [0.15, 0.95] interval by 0.1 increments, and report results in Table 10. We find that for the score-based S+R, balancing diversity with correctness means we have to use $\tau = 0.75$.

The story is more nuanced for S+RV. As this variant relies on the value function rather than the score to rerank samples, we can use it even for reference-based metrics. We observe that for these, the optimal temperature is much smaller ($\tau = 0.25$), which in effect means that the algorithm relies more heavily on the policy, compared to the value. The reason why is once again that the value for this type of metrics is of lower quality.

In contrast, the optimal temperature for our reference-less metric is $\tau = 0.75$, similar to what we find for the score-based S+R.

C Scaling search with computational budget

We present more detailed scaling results in this section. These confirm our main observation: the higher the quality of the metric signal we use, the better the method scales with additional computation.

We see for instance that on reference-based metrics, where value functions are hard to train, the performance of value-based methods reaches its peak quickly and start degrading. Comparatively, on reference-less metrics, value-based methods keep improving with more inferences, eventually plateauing. Finally, score-based methods do not even plateau. As a result, S+R ends up outperforming MCTS after 200 simulations per token, although MCTS remains the best performer under 100 simulations. This motivates investigating a variant of MCTS which is allowed to use the score on completed sentences (our current algorithm is purely value-based).

Beam search, which does not use the metric at all, behaves thus more similarly across all metrics, quickly reaching its peak performance and then plateauing. The length penalty is crucial to prevent performance degradation.

Another interesting observation is that S+RV, the value-based alternative of S+R, performs worse than VGBS or MCTS. It appears that the crucial ingredient to S+R good performance is direct access to the score, rather than its simple search mechanism.

Finally, we note that for VGBS, each token costs $k + k^2$ inferences (k to compute the policy for every beam, k^2 to compute the value for the k^2 possible follow-up tokens). As a result, we use the smallest k such that $k + k^2 \geq n$ when allowing n simulations for other decoding algorithms.

D Supervised policy vs Distilled policy

As we study decoding mechanisms on reference-based metrics based on the ground truth, we cannot train our value networks on the initial supervised dataset (where the value target would be 1 for all items, as the ground truth targets are considered optimal). As a result we go through an intermediate step, first training a supervised policy model, and then replacing ground truth targets by a greedy sample from the said policy to create a new distillation dataset.

	BLEU				BERTScore				MLBERTScore				
	BS	VGBS	S+RV	MCTS	BS	VGBS	S+RV	MCTS	BS	VGBS	S+R	S+RV	MCTS
1	38.16	38.16	28.54	38.16	90.52	90.52	87.98	90.52	84.22	84.22	82.94	82.97	84.22
10	**38.95**	39.30	34.46	38.96	90.71	90.83	90.23	90.81	84.36	85.38	85.35	85.07	85.76
25	38.81	39.25	35.01	**39.12**	**90.76**	**90.87**	**90.40**	**90.85**	84.42	85.69	85.82	85.44	86.10
50	38.67	39.33	35.19	38.95	90.76	90.85	90.37	90.80	84.46	85.80	86.12	85.68	86.31
75	38.62	39.41	**35.56**	38.89	90.76	90.86	90.31	90.74	84.48	85.75	86.33	85.86	86.44
100	38.55	**39.42**	35.13	38.84	90.76	90.86	90.37	90.67	84.48	85.78	86.44	85.95	86.51
200	38.54	39.39	35.31	38.30	90.76	90.84	90.26	90.33	84.48	85.77	86.76	86.22	86.62
300	38.45	39.36	35.00	37.70	90.74	90.83	90.16	89.83	84.49	**85.79**	**86.90**	**86.33**	**86.66**

Table 11: Comparison of how decoding algorithms scale with computational budget, on the ENFR dataset, for both reference-based and reference-less metrics.

Policy model	Supervised	Distilled
ENDE	**25.99**	25.95
ENFR	**38.70**	38.16

Table 12: Greedy decoding performance for plainly supervised policy model and dual-headed distilled ones.

We observe is the performance of the policy models trained on the distillation datasets is slightly lower than that of their plainly supervised counterparts, as illustrated in Table 12. The effect is larger for the bigger dataset, ENFR.

This prompts another line of investigation: what happens if at decoding time we use a value model (trained on a distillation dataset) together with a policy model trained on a supervised dataset? We present BLEU results for MCTS in Table 13. On the larger dataset, we see that MCTS decoding outperforms any other type of decoding. Interestingly, we obtain an improvement of more than 1 BLEU point over the supervised baseline when using a Multilingual BERTScore value function; and we obtain this result with a relatively low amount of simulations (25). Unfortunately adding more computational budget does not help, as the decoding is targeting a different metric than BLEU. But with a low enough amount of simulations, we see that trying to optimise our reference-less metric yields benefits.

While this result is close to the state of the art for such a small policy model, the comparison is not fair as the approach requires double the amount of parameters (since the value net is another network). This could be alleviated by training a supervised policy, fixing its weights, and adding a lightweight value head on top in a second training step. We leave this for future work.

	Beam search	Value type		
		BLEU	BERT	MLBERT
ENDE	27.75	27.33	27.42	27.17
ENFR	39.24	39.67	39.46	**40.31**

Table 13: MCTS (25 simulations) BLEU performance, using a supervised policy model and a distinct value model.

E MCTS examples

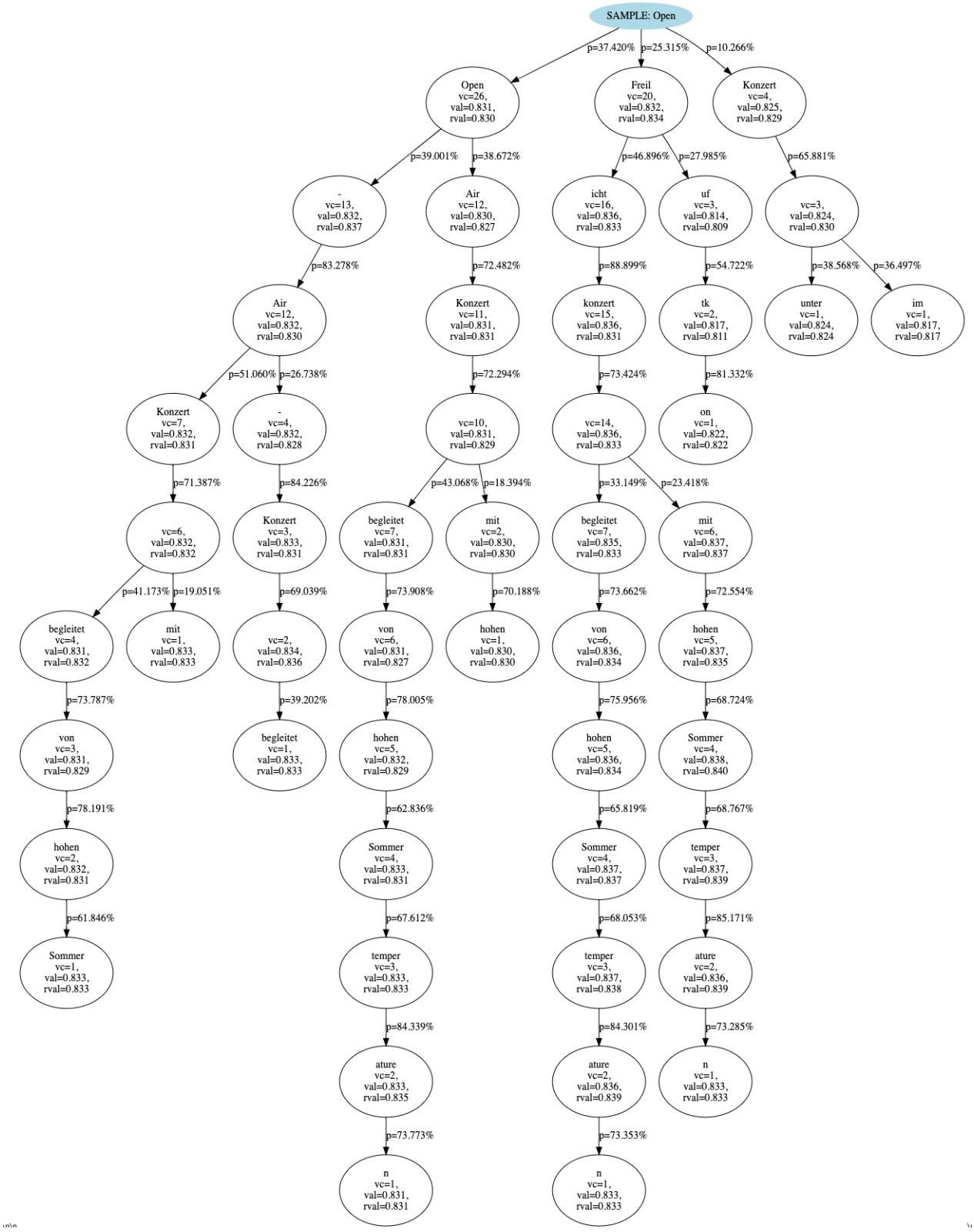

Figure 2: MCTS tree when translating “Open-air concert accompanied by high summer temperatures" from English to German, first step.

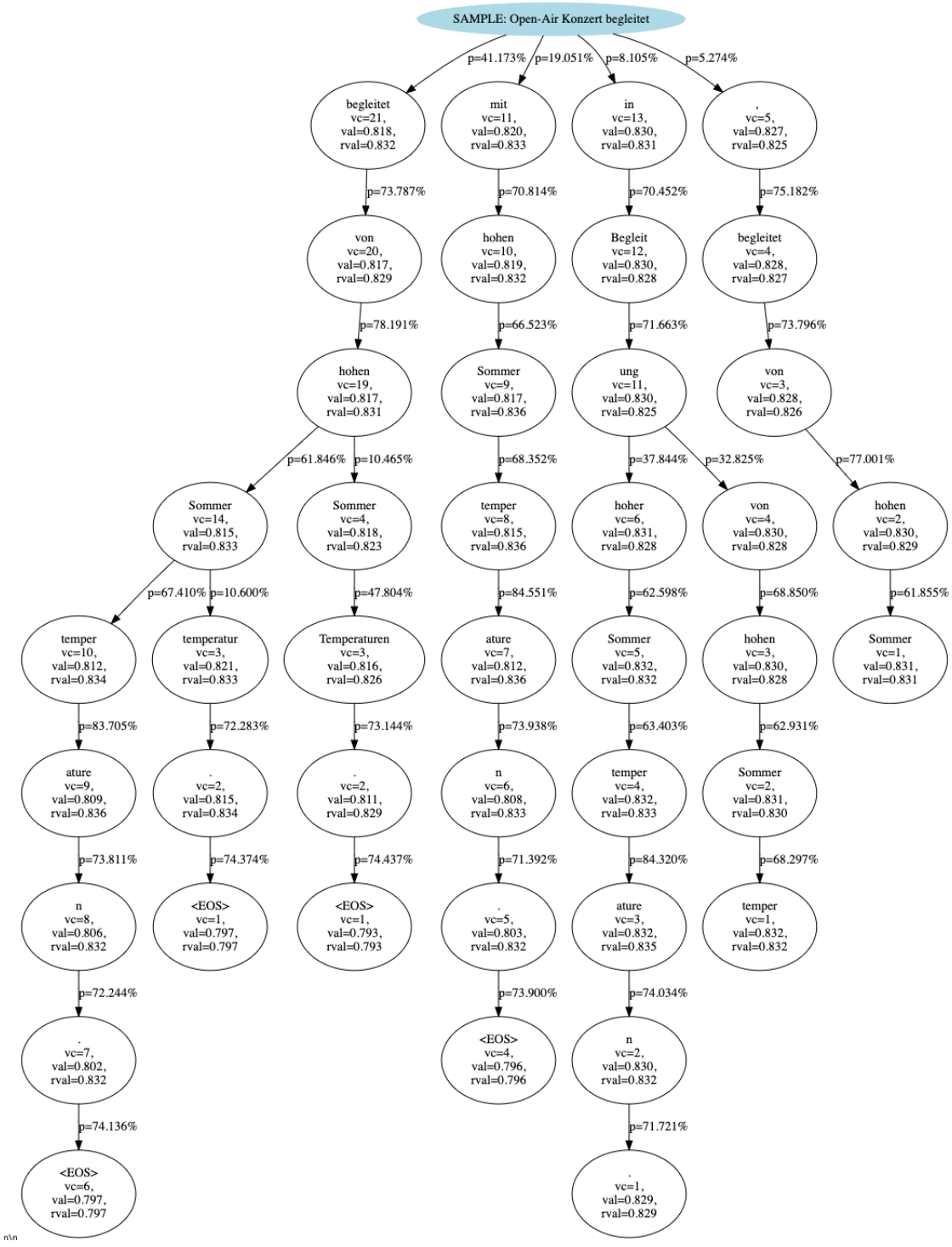

Figure 3: MCTS tree when translating “Open-air concert accompanied by high summer temperatures" from English to German, sixth step.

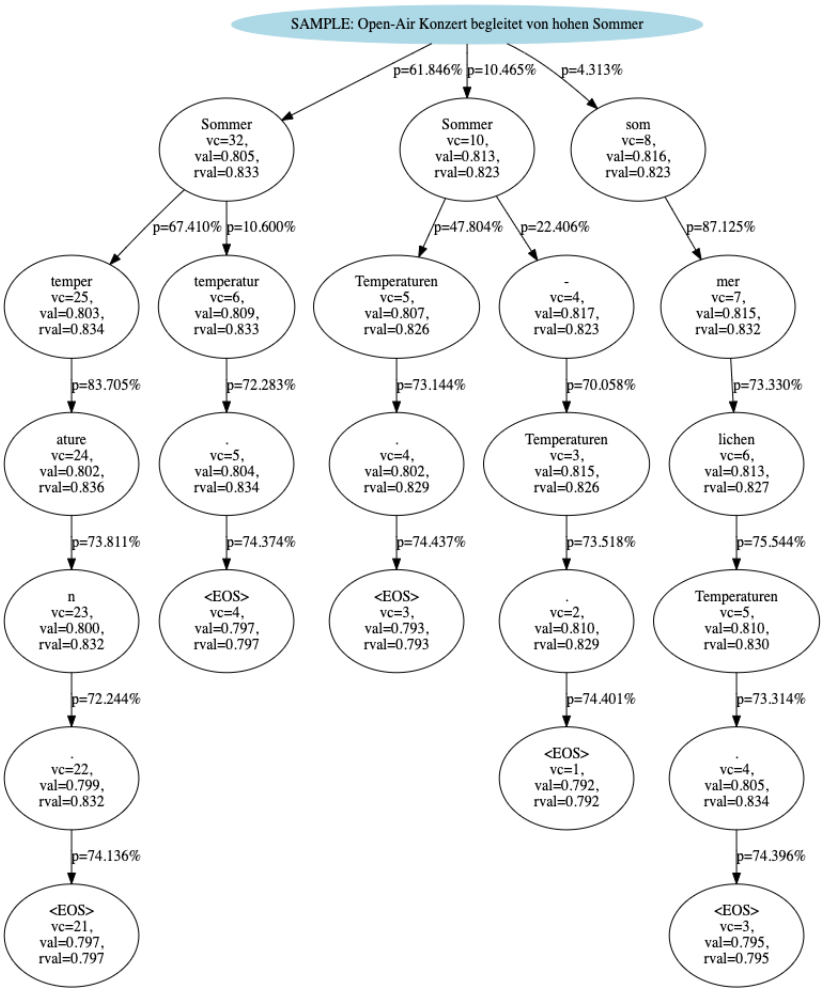

Figure 4: MCTS tree when translating “Open-air concert accompanied by high summer temperatures" from English to German, ninth step.

F Batched Numpy-friendly MCTS

Accelerator hardware such as GPUs or TPUs allow us to execute neural networks faster; but to fully leverage their computing power, we have to run on batches of several inputs. This is not very easily mixed with an algorithm such as MCTS, as it requires a queuing mechanism between the search itself and the neural network computations, potentially leading to inefficiencies. To circumvent this issue, we introduce a Numpy-compatible version of MCTS, which can then be run completely on the accelerator device.

The basic idea is that we use storage tensors which are indexed by the number of the current node or simulation in the MCTS tree. The root node has index 0 for all elements in a batch, and we then build all subsequent elements recursively.

We start by creating a `NumpyMCTS` object, whose fields store all the necessary tree information to compute a single batched instance of search (i.e. MCTS for one token, not MCTS applied to the full sequence). In details, for each node, for each item in the batch we store:

- `visit_counts`: the amount of times said nodes have been visited during the search,
- `raw_values`: the initial value of the node as returned by our value network,
- `values`: the aggregated value of the node at this point in the search,
- `parents`: which node is its parent in the tree,
- `action_from_parents`: which action was taken to transition from the parent to the node itself,
- `depth`: the tree depth of each node in the tree,
- `is_terminal`: whether or not they are a terminal node.

All these variables are tensors are of size (B, S) where B is the batch size and S is the amount of simulation plus one.

For ease of tree manipulation, we also store for each node the indices of its children, its prior over its possible children, the values of each child, and the visit count of each child. The associated tensors should be of shape (B, S, V), where V is the total number of possible actions. However this makes for large tensors, on which Numpy operations can become costly. To alleviate this issue we store a sparse version of these tensors instead, only keeping the top A children according to the policy for each node. The shapes are thus (B, S, A) instead. We maintain a mapping from 0 to $A - 1$ in the `topk_mapping` tensor, of shape (B, S, A) itself too.

Finally, the object also stores for each node its associated transformer state, so that we can use incremental inference during the search. These states can be kept on the accelerator device itself.

```
class NumpyMCTS():

 def __init__(self, root_fun, rec_fun, batch_size, num_simulations, num_actions, num_sparse_actions,
     pb_c_init):
  self._batch_size = batch_size
  self._num_simulations = num_simulations
  self._num_actions = num_actions
  self._num_sparse_actions = min(num_sparse_actions, num_actions)
  self._pb_c_init = pb_c_init

  self._root_fun = root_fun # a function called at the root
  self._rec_fun = rec_fun # a function called in the tree
  self._adaptive_min_values = np.zeros(batch_size, dtype=np.float32)
  self._adaptive_max_values = np.zeros(batch_size, dtype=np.float32)

  # Allocate all necessary storage.
  # For a given search associated to a batch-index, node i is the i-th node
  # to be expanded. Node 0 corresponds to the root node.
  num_nodes = num_simulations + 1
  batch_node = (batch_size, num_nodes)
  self._num_nodes = num_nodes
  self._visit_counts = np.zeros(batch_node, dtype=np.int32)
  self._values = np.zeros(batch_node, dtype=np.float32)
  self._raw_values = np.zeros(batch_node, dtype=np.float32)
  self._parents = np.zeros(batch_node, dtype=np.int32)
  # action_from_parents[b, i] is the action taken to reach node i.
```

```
  # Note that action_from_parents[b, 0] will remain -1, as we do not know,
  # when doing search from the root, what action led to the root.
  self._action_from_parents = np.zeros(batch_node, dtype=np.int32)
  # The 0-indexed depth of the node. The root is the only 0-depth node.
  # The depth of node i, is the depth of its parent + 1.
  self._depth = np.zeros(batch_node, dtype=np.int32)
  self._is_terminal = np.full(batch_node, False, dtype=np.bool)

  # To avoid costly numpy ops, we store a sparse version of the actions.
  # We select the top k actions according to the policy, and keep a mapping
  # of indices from 0 to k-1 to the actual action indices in the
  # self._topk_mapping tensor.
  batch_node_action = (batch_size, num_nodes, self._num_sparse_actions)
  self._topk_mapping = np.zeros(batch_node_action, dtype=np.int32)
  self._children_index = np.zeros(batch_node_action, dtype=np.int32)
  self._children_prior = np.zeros(batch_node_action, dtype=np.float32)
  self._children_values = np.zeros(batch_node_action, dtype=np.float32)
  self._children_visits = np.zeros(batch_node_action, dtype=np.int32)
  self._states = {}
  self._batch_range = np.arange(batch_size)
  self._reset_tree()

def _reset_tree(self):
  """Resets the tree arrays."""
  self._visit_counts.fill(0)
  self._values.fill(0)
  self._parents.fill(-1)
  self._action_from_parents.fill(-1)
  self._depth.fill(0)

  self._topk_mapping.fill(-1)
  self._children_index.fill(-1)
  self._children_prior.fill(0.0)
  self._children_values.fill(0.0)
  self._children_visits.fill(0)
  self._states = {} # Indexed by tuples (batch index, node index)
```

We now define a method to perform the search itself. As stated in the main text, MCTS consists in applying the same three steps for each simulation, so we iterate over S. First, we use the `simulate()` method to select which new nodes to explore. Second, we expand these new nodes (calling our neural network to compute both the policy and the value at these nodes). Finally, we back the newly computed values up the tree.

The `dense_visit_counts` method allows us to map back our sparse action representation into the original action space.

```
def search(self, raw_states):
  self._reset_tree()

  # Evaluate the root.
  prior, values, states = self._root_fun(raw_states)

  self._adaptive_min_values = values
  self._adaptive_max_values = values + 1e-6

  root_index = 0
  self._create_node(root_index, prior, values, states, np.full(self._batch_size, False, dtype=np.bool))

  # Do simulations, expansions, and backwards.
  leaf_indices = np.zeros((self._batch_size), np.int32)
  for sim in range(self._num_simulations):
    node_indices, actions = self.simulate()
    next_node_index = sim + 1 # root is 0, therefore we offset by 1.
    self.expand(node_indices, actions, next_node_index)
    leaf_indices.fill(next_node_index)
    self.backward(leaf_indices)

  return self.dense_visit_counts()

def dense_visit_counts(self):
  root_index = 0
  root_visit_counts = self._children_visits[:, root_index, :]
  dense_visit_counts = np.zeros((self._batch_size, self._num_actions))
  dense_visit_counts[self._batch_range[:, None], self._topk_mapping[:, root_index, :]] = root_visit_counts
  return dense_visit_counts
```

The simulate method consists in applying the UCT formula recursively until we have reached a new node to open for each element of the batch. The UCT formula itself can be computed in a fully batched fashion, as demonstrate by method `uct_select_action`.

```
def simulate(self):
  """Goes down until all elements have reached unexplored actions."""
  node_indices = np.zeros((self._batch_size), np.int32)
  depth = 0
  while True:
    depth += 1
    actions = self.uct_select_action(node_indices)
    next_node_indices = self._children_index[self._batch_range, node_indices, actions]
    is_unexplored = next_node_indices == -1
    if is_unexplored.all():
      return node_indices, actions
    else:
      node_indices = np.where(is_unexplored, node_indices, next_node_indices)

def uct_select_action(self, node_indices):
  """Returns the action selected for a batch of node indices of shape (B)."""
  node_children_prior = self._children_prior[self._batch_range, node_indices, :] # (B, A)
  node_children_values = self._children_values[self._batch_range, node_indices, :] # (B, A)
  node_children_visits = self._children_visits[self._batch_range, node_indices, :] # (B, A)
  node_visits = self._visit_counts[self._batch_range, node_indices] # (B)

  node_policy_score = np.sqrt(node_visits[:, None]) * self._pb_c_init * node_children_prior /
       (node_children_visits + 1) # (B, A)

  # Remap values between 0 and 1.
  node_value_score = node_children_values
  node_value_score = (node_value_score != 0) * node_value_score + (node_value_score == 0) *
       self._adaptive_min_values[:, None]
  node_value_score = (node_value_score - self._adaptive_min_values[:, None]) /
       (self._adaptive_max_values[:, None] - self._adaptive_min_values[:, None])

  node_uct_score = node_value_score + node_policy_score # (B, A)
  actions = np.argmax(node_uct_score, axis=1)
  return actions
```

Once we have selected nodes to expand, we can proceed. The `expand` method is where we call our neural networks to compute policies and values. We then create the nodes in the object fields through the `create_node` method. Finally, we update the tree topology to connect the new nodes to the tree.

```
def expand(self, node_indices, actions, next_node_index):
  """Creates and evaluate child nodes from given nodes and unexplored actions."""

  # Retrieve states for nodes to be evaluated.
  states = [self._states[(b, n)] for b, n in enumerate(node_indices)]
  previous_node_is_terminal = self._is_terminal[self._batch_range, node_indices[self._batch_range]] # (B)

  # Convert sparse actions to dense actions for network computation
  dense_actions = self._topk_mapping[self._batch_range, node_indices, actions]

  # Evaluate nodes.
  (prior, values, next_states, expanded_node_is_terminal) = self._rec_fun(states, dense_actions)

  # Create the new nodes.
  self.create_node(next_node_index, prior, values, next_states, expanded_node_is_terminal)

  # Update the min and max values arrays
  self._adaptive_min_values = np.minimum(self._adaptive_min_values, values)
  self._adaptive_max_values = np.maximum(self._adaptive_max_values, values)

  # Update tree topology.
  self._children_index[self._batch_range, node_indices, actions] = next_node_index
  self._parents[:, next_node_index] = node_indices
  self._action_from_parents[:, next_node_index] = actions
  self._depth[:, next_node_index] = self._depth[self._batch_range, node_indices] + 1

def create_node(self, node_index, prior, values, next_states, expanded_node_is_terminal):
  # Truncate the prior to only keep the top k logits
  prior_topk_indices = np.argpartition(prior, -self._num_sparse_actions, axis=-1)[:,
       -self._num_sparse_actions:]
  prior = prior[self._batch_range[:, None], prior_topk_indices] # (B, A)

  # Store the indices of the top k logits
  self._topk_mapping[self._batch_range, node_index, :] = prior_topk_indices

  # Update prior, values and visit counts.
  self._children_prior[:, node_index, :] = prior
  self._values[:, node_index] = values
  self._raw_values[:, node_index] = values
  self._visit_counts[:, node_index] = 1
  self._is_terminal[:, node_index] = expanded_node_is_terminal

  # Update states.
  for b, next_state in enumerate(next_states):
```

```
    self._states[(b, node_index)] = next_state
```

Finally, we back the newly computed values up using the `backward` method, which can again be done in a fully batched fashion.

```
def backward(self, leaf_indices):
  """Goes up and updates the tree until all nodes reached the root."""
  node_indices = leaf_indices # (B)
  leaf_values = self._values[self._batch_range, leaf_indices]
  while True:
    is_root = node_indices == 0
    if is_root.all():
      return
    parents = np.where(is_root, 0, self._parents[self._batch_range, node_indices])
    root_mask = 1.0 * is_root
    not_root_mask_int = (1 - is_root)
    not_root_mask = 1.0 - root_mask
    # Update the parent nodes iff their child is not the root.
    # We therefore mask the updates using not_root_mask and root_mask.
    self._values[self._batch_range, parents] = not_root_mask * (self._values[self._batch_range, parents] *
         self._visit_counts[self._batch_range, parents] + leaf_values) /
         (self._visit_counts[self._batch_range, parents] + 1.0) + root_mask *
         self._values[self._batch_range, parents]
    self._visit_counts[self._batch_range, parents] += not_root_mask_int
    actions = np.where(is_root, 0, self._action_from_parents[self._batch_range, node_indices])
    self._children_values[self._batch_range, parents, actions] = not_root_mask *
         self._values[self._batch_range,node_indices] + root_mask *
         self._children_values[self._batch_range, parents, actions]
    self._children_visits[self._batch_range, parents, actions] += not_root_mask_int

    # Go up
    node_indices = parents
```