Gauge dependence and structured-output corruption in sign-branched repetition penalties

Measurements across models, inference stacks, and alternative repetition controls

Peter Hollows

Abstract

The multiplicative repetition penalty shipped across the LLM inference ecosystem (HuggingFace, vLLM, llama.cpp, and a dozen further engines) branches on the *sign* of each raw logit (divide positives by θ, multiply negatives). But the softmax is unchanged by adding a constant to every logit, so a model's logit zero-point is arbitrary, and the sign-branch reads that arbitrary point. The sign-branch is itself the accepted fix for an earlier bug, so the accepted fix branches on a quantity the training objective leaves unconstrained. Two measurable consequences follow. (1) The penalty is not well-defined: re-centring a model's logits by a constant is a provable no-op at $\theta = 1$, yet at a routine $\theta = 1.3$ it changes 58–96% of greedy tokens, where subtractive and normalized penalties change none; real checkpoints sit at widely different zero-points, so a fixed `repetition_penalty` is a different operation on every model. (2) It corrupts structured output: on 200 real-world JSON schemas, $\theta = 1.3$ drops the rate of valid, schema-conformant output from 97% to 23%. In our measurements, applying the penalty to normalized log-probabilities instead of raw logits removes both effects. HuggingFace already ships that operator (`LogitNormalization`); today it is off by default and applied *after* the penalty. This note gives the mechanism, the measurements (five models up to 7B, base and RLHF, on WikiText-103 prefixes; two code models on HumanEval and JSONSchemaBench; both effects replicated inside vLLM and llama.cpp through their own samplers on the same inputs), and the normalized variant.

1 The penalty branches on an unconstrained coordinate

A repetition penalty is supposed to be a clean scalar knob: set `repetition_penalty` ≈ 1.1–1.3, get less repetition, and expect it to mean roughly the same thing on any model. The multiplicative form used almost everywhere (HuggingFace's `RepetitionPenaltyLogitsProcessor`, vLLM's `repetition_penalty`, llama.cpp's `repeat_penalty`, all inherited from CTRL [4]) does not have that property; Section 4 surveys how widely the operator ships. It transforms an already-seen token's logit z_i by

$$z_i' = \begin{cases} z_i/\theta & z_i \geq 0 \quad \text{(divide)}, \\ z_i \cdot \theta & z_i < 0 \quad \text{(multiply)}, \end{cases} \tag{1}$$

i.e. it branches on the *sign* of the raw logit. The verbatim implementation is three lines:

```
score = torch.gather(scores, 1, input_ids)
score = torch.where(score < 0, score * penalty, score / penalty)  # sign-branch
scores_processed = scores.scatter(1, input_ids, score)
```

The sign-branch is itself a fix: the naive version divided *every* seen logit by θ, but dividing a negative logit by $\theta > 1$ moves it toward zero and *raises* the token's probability, which is backwards; this was

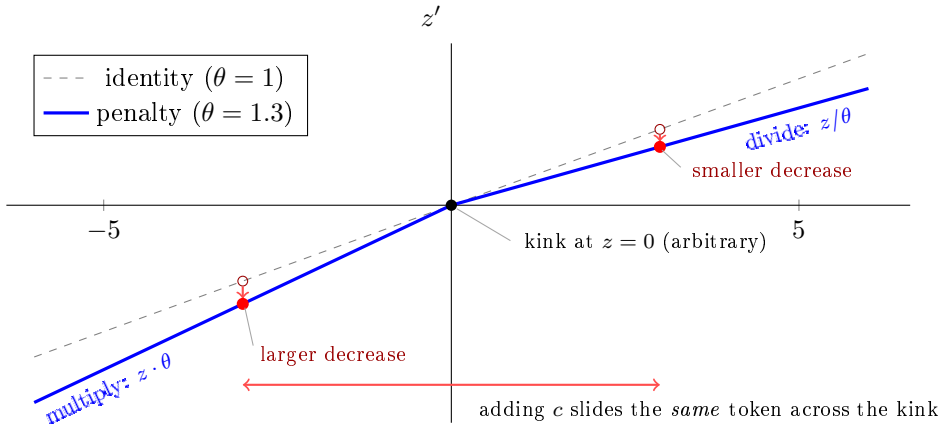

Figure 1: The penalty maps a seen token's logit z to z': positive logits are divided by θ (shallow slope), negative logits multiplied (steep slope), meeting at a kink at $z = 0$. Open marks show each token's unpenalized value on the identity line; the arrows show the penalty's decrease, smaller on the divide branch and larger on the multiply branch. Because the softmax does not change when a constant is added to every logit, the model can slide every seen token left or right across this kink without changing any output probability, so the *same* token can be divided under one gauge and multiplied under another and receive a different penalty.

reported to HuggingFace in 2019 [8], and the sign-branch is the accepted remedy. The problem is one level up. The softmax obeys $\mathrm{softmax}(z+c) = \mathrm{softmax}(z)$ for any scalar $c \in \mathbb{R}$, where $z+c$ adds c to every coordinate of the logit vector. A uniform shift therefore leaves the next-token distribution unchanged, so the greedy token is unchanged exactly, and sampled output is unchanged token-for-token under a fixed random seed. We call a choice of the constant c a *gauge*: the model's behavior fixes the distribution but not the gauge. So the location of the sign-branch's kink, $z_i = 0$, is a coordinate not constrained by training: the loss depends on the logits only through the softmax. The accepted fix for the naive penalty branches on this unconstrained coordinate (Figure 1).

2 Consequence 1: the penalty is not well-defined (A1)

The operator branches on the sign of each raw logit, and where a model's logits sit relative to zero is a quantity not constrained by training. It differs widely across real checkpoints: measured at decode time on five checkpoints (Table 1), the fraction of seen-token logits that are positive, and so *divided* rather than multiplied, ranges from 0.17 on gpt2 to 0.95 on Qwen2.5-Coder-7B. The same `repetition_penalty=1.3` therefore takes the multiply branch for most seen tokens on the first model and the divide branch for nearly all on the second. A fixed `repetition_penalty` is a different operation on every model.

Within one model this becomes a controlled experiment. Adding a constant c to every logit has no effect on the model's output distribution (Section 1), so we decode the *same* model twice at the same θ, once with $c = +5$ and once with $c = -5$ added to the logits (equivalently, `lm_head.bias` $+= c$), and count the fraction of greedy positions where the two runs differ: the flip rate. For a well-defined intervention the flip rate is zero, and for the other standard repetition controls it is: with a subtractive presence penalty ($z_i - \alpha$ on seen tokens, $\alpha = 1$) the two runs are token-for-token

Table 1: Where five real checkpoints sit relative to the penalty's sign boundary, at decode time. This set spans the divide-fraction range and includes the two code models used in Consequence 2; the controlled flip experiment (Table 2) uses the five base/RLHF models. The divide-fraction spans 0.17–0.95, and re-centring each model by its own median logit (a behaviorally invisible gauge) flips most of its greedy tokens at $\theta = 1.3$. gpt2's per-position logits are bimodal (top-1 deciles span roughly -230 to $+150$), so a large negative median coexists with a positive divide-fraction.

Model	frac. seen logit > 0 (divided)	median top-1 logit	flip @ own-median gauge
gpt2	0.17	-161.4	0.71
gpt2-large	0.52	13.3	0.96
starcoder2-7b	0.78	18.8	0.96
pythia-2.8b	0.88	19.7	0.96
Qwen2.5-Coder-7b	0.95	24.8	0.97

Table 2: Gauge flip rate: fraction of greedy tokens that differ between $c = +5$ and $c = -5$, two configurations with identical output distributions. A well-defined intervention flips nothing; the subtractive presence penalty and the normalized multiplicative penalty flip none of the 40,000 positions on any model, while the CTRL sign-branch flips most tokens. The θ=1 column is the harness control: any implementation error would flip tokens there.

Model	CTRL, θ=1 (gate)	CTRL, θ=1.3	subtractive, α=1	normalized
gpt2-large	0.000	0.96	0.000	0.000
pythia-2.8b	0.000	0.94	0.000	0.000
gpt2	0.000	0.58	0.000	0.000
Qwen2.5-7B (base)	0.000	0.92	0.000	0.000
Qwen2.5-7B-Instruct	0.000	0.92	0.000	0.000

identical on every model, and the same holds for the multiplicative penalty applied to normalized log-probabilities (Section 5). Prompts follow the standard open-ended-generation protocol: 200 32-token prefixes sampled from the WikiText-103 test set [5, 6], 200 greedy tokens each, i.e. 40,000 compared positions per model. At $\theta = 1$ the CTRL runs are also identical (a provable no-op). At $\theta = 1.3$ the CTRL operator flips 58–96% of greedy tokens (Table 2); this holds at 7B and is undiminished by RLHF. The probe needs no synthetic constant either: re-centring each model by its own median logit (a gauge it could equally have shipped) flips 71–97% (Table 1, last column). The own-median offsets are larger than the ± 5 probe (for gpt2, 161 logit units), which is why gpt2's 0.71 there exceeds its 0.58 in Table 2.

Reading the flip rate. A flip is not by itself a quality claim: a penalty is supposed to change tokens, and in open-ended prose there is often no single correct continuation, so we make no claim that either gauge's output is worse. The controls fix the number's meaning. Up to the first divergence the two runs share an identical context, so the first differing token cannot be inherited from earlier differences; it can only come from the gauge changing that one greedy decision. The control operators run the same 200-token greedy process without one such flip in 40,000 positions. Under the CTRL operator, divergence reaches most prefixes (200/200 on four models, 180/200 on gpt2), with median first divergence at position 4–10 (62 on gpt2); once diverged, two continuations stay $\sim$99% different, as any two different texts do, so the headline rate reflects early and pervasive

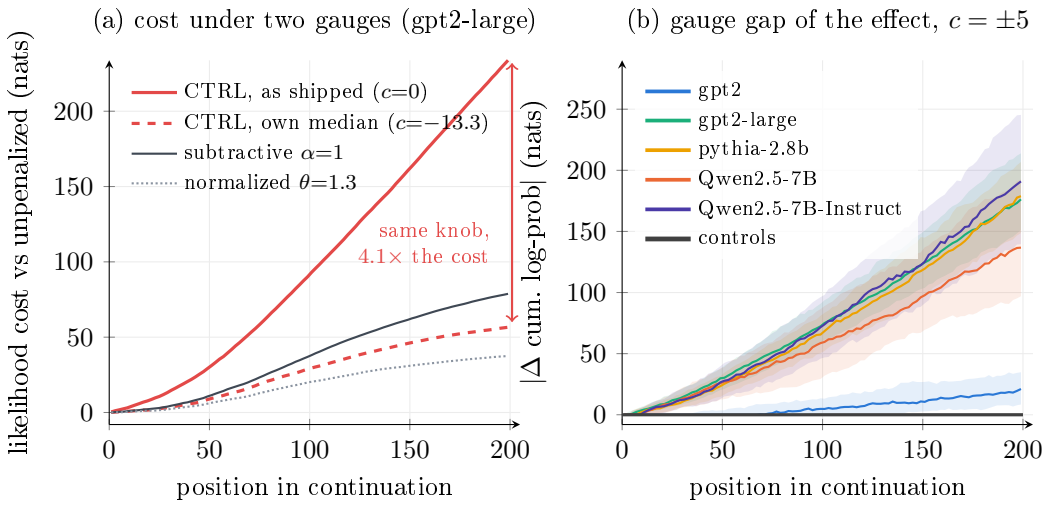

Figure 2: The cost of each intervention in model log-likelihood: traces scored token-by-token under the unpenalized distribution, mean over the 200 WikiText-103 prefixes. (a) gpt2-large under two equally-legitimate gauges of the same checkpoint (as shipped, and re-centred by its own median logit): the same $\theta = 1.3$ costs 4.1× more in one gauge than the other, and the two prices straddle the subtractive penalty's. The controls each have a single, gauge-independent cost curve. (b) Cumulative log-probability gap between the $c = \pm 5$ gauge runs (median and IQR over prefixes): it grows without saturating on every model, while the two control operators (the subtractive presence penalty, $\alpha = 1$, and the normalized penalty) sit at exactly zero at every position.

seeding rather than a per-position disagreement rate. The gauge also sets how strongly the penalty acts: the size of the push depends on where a seen logit sits relative to the kink (Figure 1), so the same θ can over- or under-suppress depending on the zero-point. Where correctness is defined, the flips stop being neutral: that is Consequence 2. `repetition_penalty=1.3` does not name one behavior; the default penalty is an inference-time intervention whose effect depends in part on the logit zero-point, a quantity training sets only incidentally.

The likelihood cost of the intervention. Scoring every trace token-by-token under the model's unpenalized distribution prices each intervention in log-likelihood (Figure 2). The question is whether that price is well-defined. For the controls it is: one deterministic cost curve each. For the CTRL operator it is not: on gpt2-large at $\theta = 1.3$, the model as shipped pays 235 nats over 200 tokens, and the same model re-centred by its own median logit pays 57; the subtractive penalty's 79 lies between the two. Across models, the cumulative log-likelihood gap between the $c = \pm 5$ gauge runs grows roughly linearly, reaching 140–190 nats by position 200 on four of five models (Figure 2b); for the controls it is zero at every position. The settings are routine values, not calibrated to each other: $\theta = 1.3$ is the setting tested throughout, and $\alpha = 1$ is mid-range of the hosted-API presence-penalty interval. Cost levels across *different* operators therefore are not comparable; the comparisons that do not depend on these choices are within the CTRL pair and against zero.

3 Consequence 2: it corrupts structured output (A2)

Structured output has repetition the grammar *requires*: JSON must reclose `}`, reseparate with `,`, requote with `"`; code must re-indent. Those delimiters have appeared before, so the penalty pushes

them down, and it cannot tell "the grammar needs this" from "the model is looping." At a delimiter where the correct token is the confident top choice (logit $z_{\text{top}} > 0$) with the runner-up a gap g below (and itself unpenalized; positions where both are penalized obey a two-sided variant), the penalty ($z_{\text{top}} \mapsto z_{\text{top}}/\theta$) flips the greedy choice to the runner-up exactly when

$$z_{\text{top}}/\theta < z_{\text{top}} - g \iff g < z_{\text{top}}\,(1 - 1/\theta). \tag{2}$$

On StarCoder2-7B, decoding the 164 HumanEval prompts [1], this closed form predicts the observed penalty-induced flips with balanced accuracy 0.999 over the 48,919 positions where its assumptions hold (top token penalized on the divide branch, runner-up unpenalized; pooled over $\theta \in \{1.1, \ldots, 1.5\}$), and all 27,720 flips there land on the pre-penalty runner-up (this replicates on Qwen2.5-Coder-7B). End to end: generating complete JSON objects against 200 real-world schemas sampled from JSONSchemaBench [2], a routine $\theta = 1.3$ takes the rate of parseable, schema-valid output from 97% to 23% (Table 4); the surviving valid outputs are dominated by near-empty schemas with no required properties, i.e. instances with few grammar-obligatory delimiters to corrupt.[1]

4 The operator ships across the inference ecosystem

Both effects replicate inside vLLM and llama.cpp. The main experiments use the HuggingFace implementation; we re-ran A1 and A2 inside both other stacks, on the same benchmark inputs, through each stack's own sampler and its own knob: vLLM 0.8.5.post1 (pinned by the test host's driver; the operator in `vllm/main` at `e196268` is mathematically identical, so the pin does not narrow the finding) and llama.cpp at master commit `4fc4ec55`. The A1 gauge probe passes its validity gate in both stacks (exactly 0 of 40,000 greedy tokens flip at $\theta = 1$) and reproduces the divergence at $\theta = 1.3$ on gpt2-large: flip rate 0.964 in both, matching HuggingFace, and the vLLM run is token-identical to the HuggingFace run at every compared position. The A2 JSON task (Qwen2.5-Coder-7B, same schemas and validator) falls from 0.97 valid at $\theta = 1$ to 0.24 (vLLM) and 0.29 (llama.cpp) at $\theta = 1.3$ (HF: 0.23). llama.cpp's default 64-token penalty window makes this worse, not better (0.12 at $\theta = 1.3$): on multi-hundred-token structured outputs the sliding window concentrates the penalty on the most recent, grammar-obligatory tokens.[2] Scripts and raw outputs are in the repository (`code/smoke_*`, `results/smoke_*_bench`).

The operator is not confined to those three. A source survey in the repository finds it in twelve further engines (TGI, SGLang, TensorRT-LLM, ExLlamaV2, mlx-lm, LMDeploy, aphrodite-engine, KoboldCpp, mistral.rs, candle, text-generation-webui, and Ollama), in every case a sign-branch on raw logits with no normalization; Ollama enables the penalty by default (`repeat_penalty` = 1.1). These engines carry both effects by operator form; we have not run the probes inside them (Table 3). A git-history trace in the repository shows the sign-branch was reached independently at least twice, in HuggingFace's 2019 fix for the naive divide and in an unrelated 2023 re-derivation in llama.cpp from the CTRL paper, while the surveyed engines each copied a prior implementation rather than deriving it anew; neither derivation records the gauge dependence. Because each engine keeps its own copy rather than importing a shared one, no single upstream patch reaches them all.

[1] We also pre-registered, and our data refuted, two sharper claims: that corruption hits the *most confident* tokens first (it is gap-driven, so it hits *less* confident tokens) and that it is strongly code-specific (the code-vs-prose ratio is real but model-dependent, 2.3× on StarCoder2 vs 5.6× on Qwen). Details and the full audit are in the repository.

[2] Measuring inside llama.cpp also surfaced an unrelated defect: `-repeat-last-n -1` is documented as "context size" but the sampler clamps the value to 0, silently disabling the penalty (`src/llama-sampler.cpp`, `std::max(penalty_last_n, 0)`); reported as ggml-org/llama.cpp#25388.

Table 3: Penalty form by stack. The multiplicative CTRL form (sign-branch on raw logits) is subject to both effects; the subtractive presence/frequency form subtracts a constant, is shift-invariant, and is immune. The three multiplicative stacks are measured directly (HuggingFace in the main experiments; vLLM and llama.cpp in this section); the surveyed row covers the twelve engines named in the text, classified by operator form, not probed directly.

Stack / knob	Penalty form	Normalizes first?	A1?	A2?
HF `repetition_penalty`	multiplicative (CTRL)	no (default)	yes	yes
llama.cpp `repeat_penalty`	multiplicative (CTRL)	no	yes	yes
vLLM `repetition_penalty`	multiplicative (CTRL)	no	yes	yes
twelve surveyed engines	multiplicative (CTRL)	no	by form	by form
Hosted-API `presence_penalty`	subtractive (constant)	n/a	no	no
`frequency_penalty` (API/vLLM)	subtractive (count-scaled)	n/a	no	no

Table 4: Normalize-before-penalize, measured. Applying the penalty to $\log\mathrm{softmax}(z)$ removes A1 gauge divergence, reduces delimiter corruption $\sim$80$\times$, and returns JSON validity to its $\theta = 1$ level. The delimiter-flip rate counts flips landing on structural tokens as a fraction of all generated code tokens (mean over generations, θ pooled over 1.1–1.5); it is a different population from the formula-exact positions of Section 3.

Metric	raw logits (default)	normalized
Gauge flip rate @θ=1.3 (gpt2-large / pythia / gpt2)	0.96 / 0.94 / 0.58	0.00 / 0.00 / 0.00
Delimiter-flip rate (code, StarCoder2-7B)	0.139	0.0017
JSON schema-valid rate @θ=1.3 (Qwen2.5-Coder-7B)	0.23	0.97

5 Normalizing before the penalty

Both problems have one root, branching on the sign of an un-normalized logit. Shift-invariant repetition controls already exist (the subtractive family of Table 3); as an exploration, this section measures what changes if the multiplicative operator itself reads a shift-invariant coordinate, i.e. the penalty applied to $\ell = \log\mathrm{softmax}(z)$ instead of to z. The measurements concern the two effects; we make no claim about output quality. Two things happen. (a) $\log\mathrm{softmax}$ is itself shift-invariant, so the arbitrary gauge c is gone and A1 disappears by construction. (b) Every $\ell_i \le 0$, so the sign-branch stops branching: every seen token takes the multiply path, $\ell_i \mapsto \theta\,\ell_i$. Since $\theta \log p_i = \log p_i^\theta$, this is $p_i \mapsto p_i^\theta$ in probability space: the normalized penalty tempers each seen token's probability downward by a common power. It is a monotone, well-defined suppression that reads a coordinate the model's distribution determines (where $\sum_i e^{\ell_i} = 1$) rather than one the training objective leaves free.

Measured, normalizing first removes both effects (Table 4): A1 divergence collapses to 0 at every θ on every model, delimiter corruption drops $\sim$80$\times$, and JSON validity at $\theta = 1.3$ returns from 23% to 97%, its $\theta = 1$ level. It still breaks degenerate loops (repetition rate falls monotonically in θ). One caveat: because it acts on log-probabilities ($O(1)$) rather than raw logits ($O(10)$), a given θ is gentler, so existing tuned values must be re-set; in exchange, the same θ denotes the same operation on every model.

HuggingFace already ships this normalization as `LogitNormalization`; it is off by default and, when enabled, runs after the penalty. The same reorder applies to every stack carrying the multiplicative form (Section 4). Standardizing on the subtractive family, or on a redesigned penalty

such as the LZ penalty [3], avoids the operator entirely; which path to take, if any, is a call for each stack's maintainers.

6 Scope

We study the multiplicative CTRL operator of Eq. (1); subtractive penalties are immune and stacks that already normalize before penalizing are exempt. A1 is measured on five models up to 7B (base and RLHF) over 200 WikiText-103 test prefixes, with subtractive and normalized penalties as measured controls; A2 on two 7B code models over the 164 HumanEval prompts and 200 JSONSchemaBench schemas; both replicate inside vLLM and llama.cpp through each stack's own sampler, on the same inputs. The original pre-registered runs used 16 fixed prompts and 6 hand-written schemas; their frozen decision rules, verdicts, and raw outputs remain in the repository, and their verdicts are consistent with the benchmark numbers reported here. All experiments are inference-only, greedy, on a single GPU; the harness, pinned environment (`transformers 5.11.0`), pre-registered decision rules, and a record of what did and did not replicate are in the repository, alongside a longer version with confidence intervals and the full audit.

7 Conclusion

Every repetition penalty is a workaround: an inference-time intervention correcting behavior the model itself gets wrong. Workarounds differ in their sharp edges. The default multiplicative operator branches on the logit zero-point, a coordinate training sets only incidentally; in consequence, a fixed setting is a different operation on every model, equivalent gauges of one model decode and cost differently at the same θ, and routine strengths corrupt structured output. Tuning per model and per task, the standard advice, presupposes that these edges are known. We found one prior observation of the mechanism, a 2023 llama.cpp issue noting the kink's conflict with softmax shift-invariance, closed without action [7], and no documentation of either consequence; this note documents them. They belong to this operator, not to repetition control in general: the subtractive penalties avoid them by construction, though not every stack exposes one (HuggingFace's `generate` ships only the multiplicative form), and the normalized variant of Section 5 removed them in every measurement we ran.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. HumanEval.

[2] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Jsonschemabench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868*, 2025. https://github.com/guidance-ai/jsonschemabench.

[3] Antonio A. Ginart, Naveen Kodali, Jason Lee, Caiming Xiong, Silvio Savarese, and John R. Emmons. LZ penalty: An information-theoretic repetition penalty for autoregressive language models. *arXiv preprint arXiv:2504.20131*, 2025.

[4] Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. CTRL: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019.

[5] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICLR)*, 2017. WikiText-103. arXiv:1609.07843.

[6] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[7] tysam-code. Suggested fixes for mathematical inaccuracy in `llama_sample_repetition_penalty` function, 2023. llama.cpp issue #2970, September 2023; closed as stale. https://github.com/ggml-org/llama.cpp/issues/2970.

[8] Patrick von Platen. Repetition penalty work falsely in case the logit of the token is negativ. https://github.com/huggingface/transformers/issues/2302, 2019. HuggingFace Transformers issue #2302 (December 2019), reporting that dividing a negative logit by a penalty > 1 raises its probability; fixed by the author in PR #2303 with the sign-branched divide/multiply form.