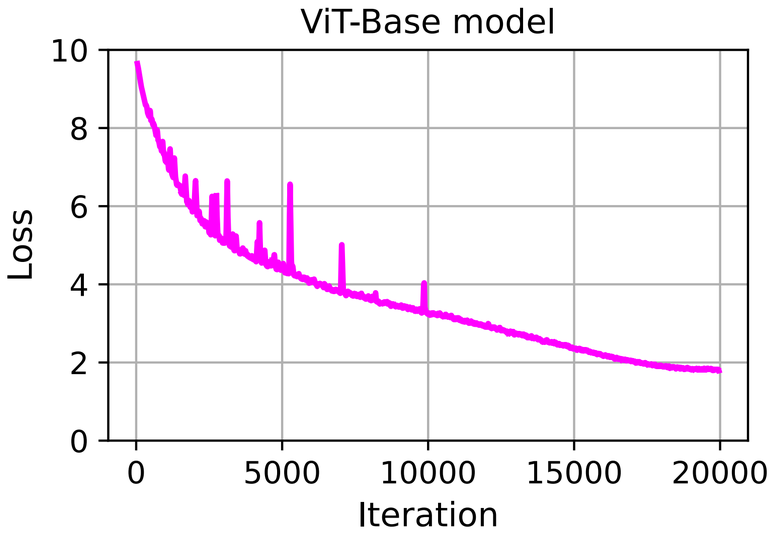
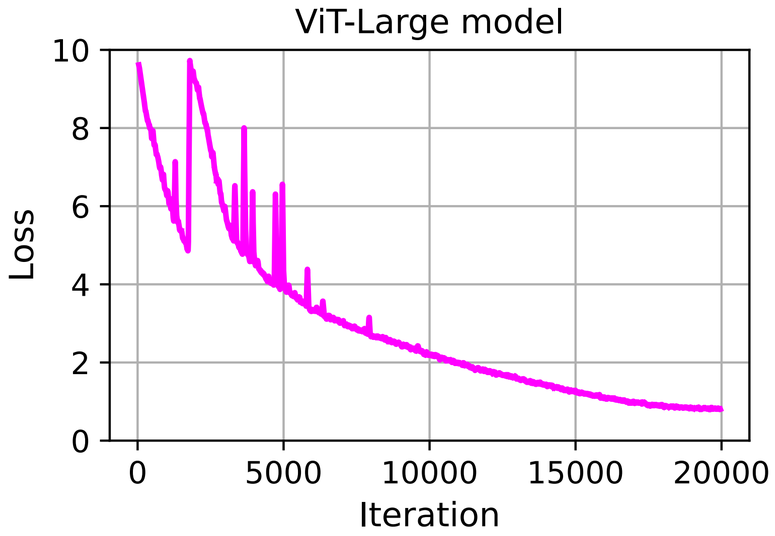
Adam - what can go wrong and how to fix it

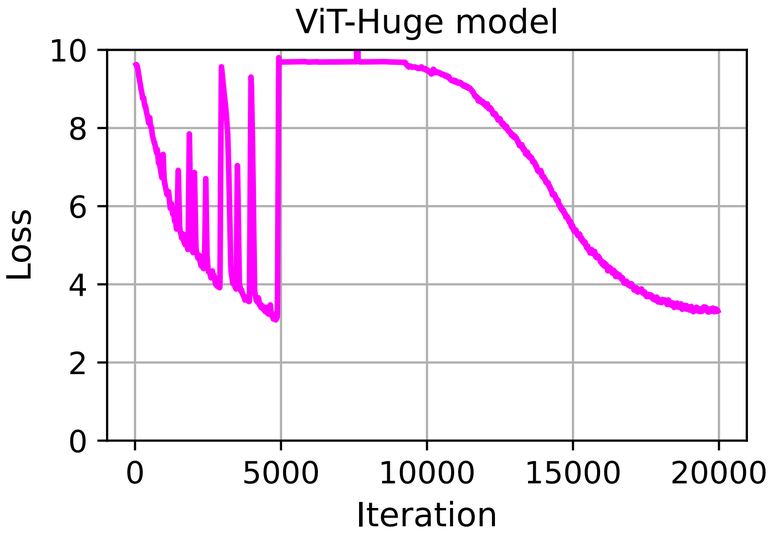
Mitchell

Based on part of an upcoming research project with Tim Dettmers, Luke Zettlemoyer, Ari Morcos, Ali Farhadi, Ludwig Schmidt

What can go wrong with Adam?

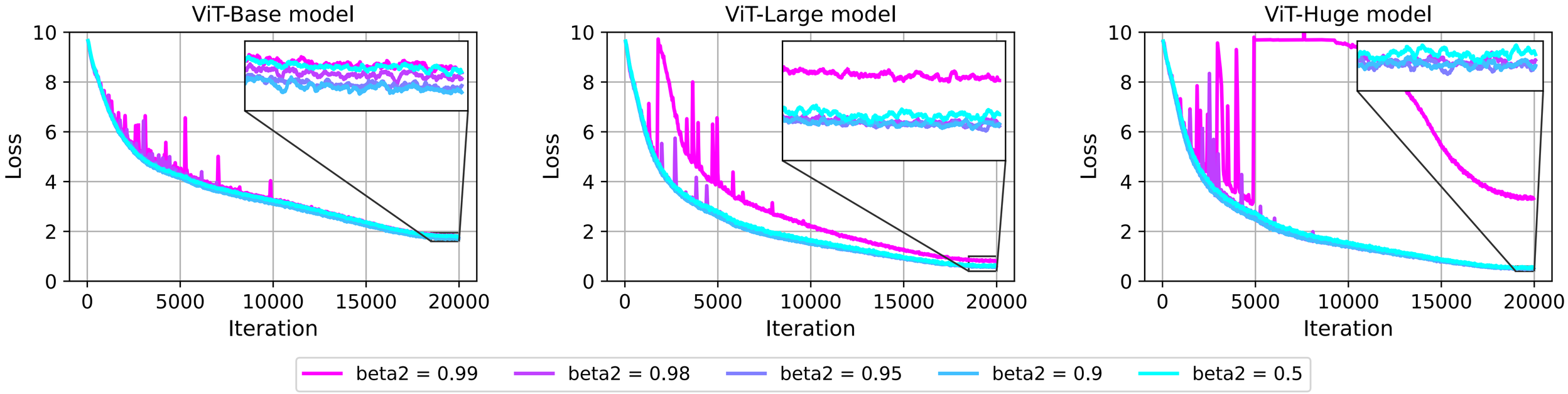






Beta1 = 0.9, Beta2 = 0.99

Tuning more AdamW beta2 values

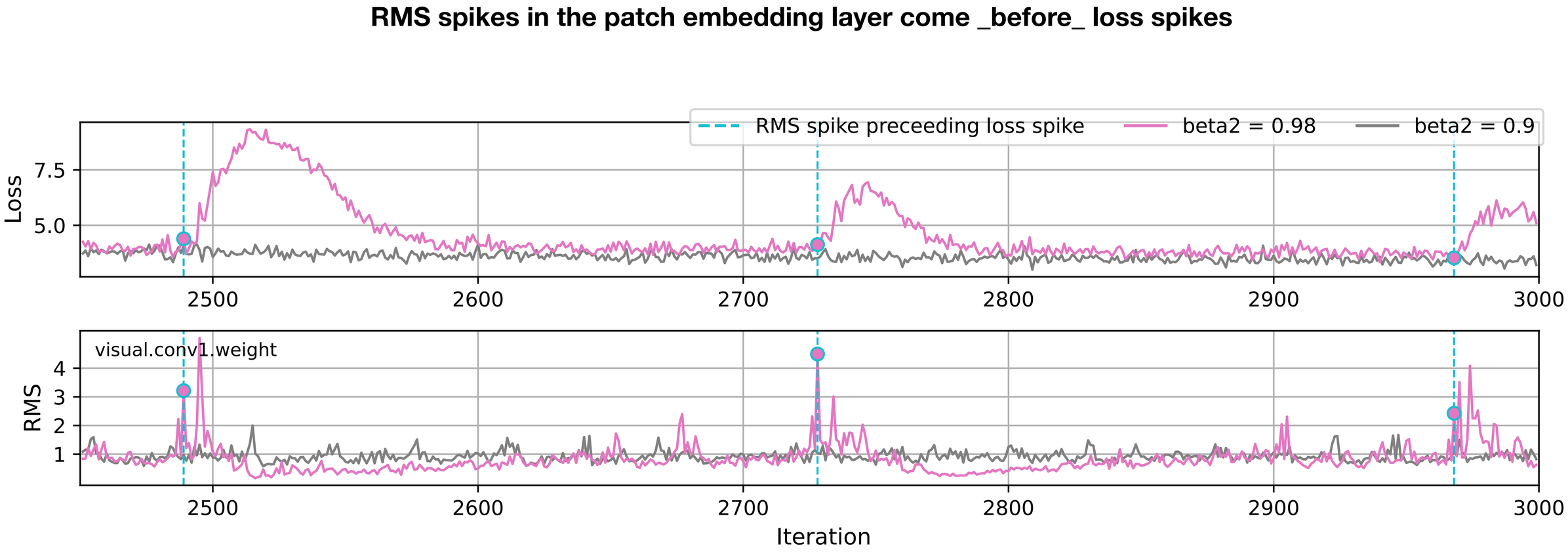


The second moment estimator becomes out-of-date, leading to updates which are too large.

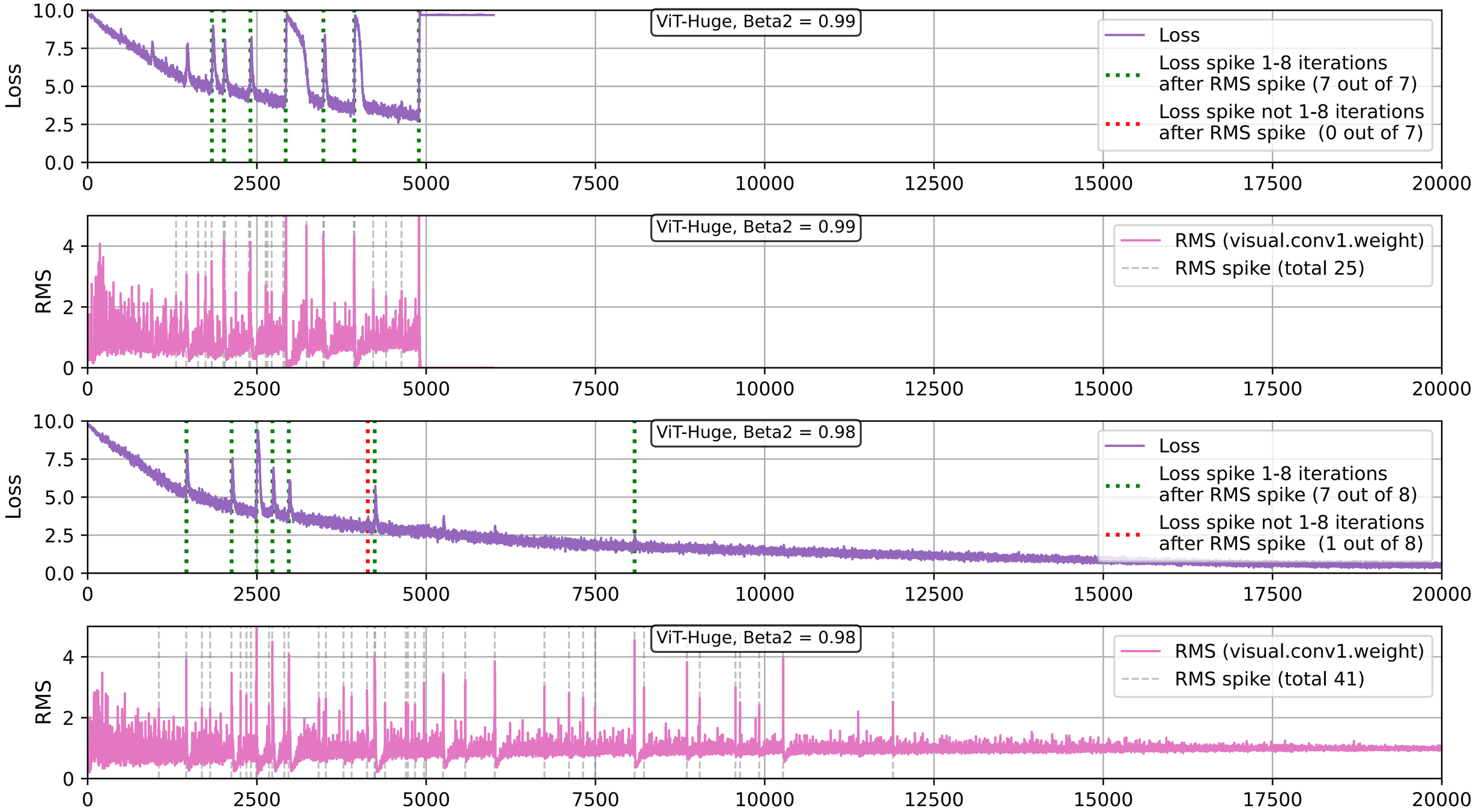
Let's track how good of an estimator we have for the squared gradient… the following aggregate quantity should be approx 1

$$\text{RMS}_t = \sqrt{\mathbb{E}\left[\frac{g_t^2}{u_t}\right]}$$

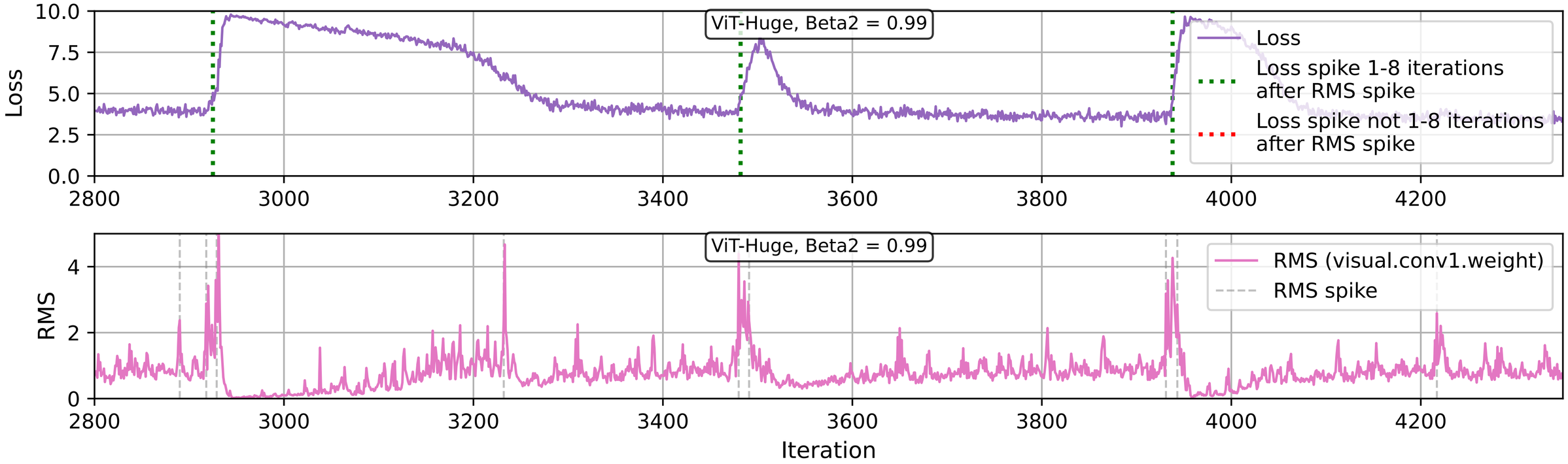
Can we search for a causal relationship between an RMS spike and a loss spike?


RMS spikes in the patch embedding layer come _before_ loss spikes
RMS spike preceeding loss spike
beta2 = 0.98
beta2 = 0.9
Loss
7.5
5.0
visual.conv1.weight
RMS
4
3
2
1
2500
2600
2700
2800
2900
3000
Iteration

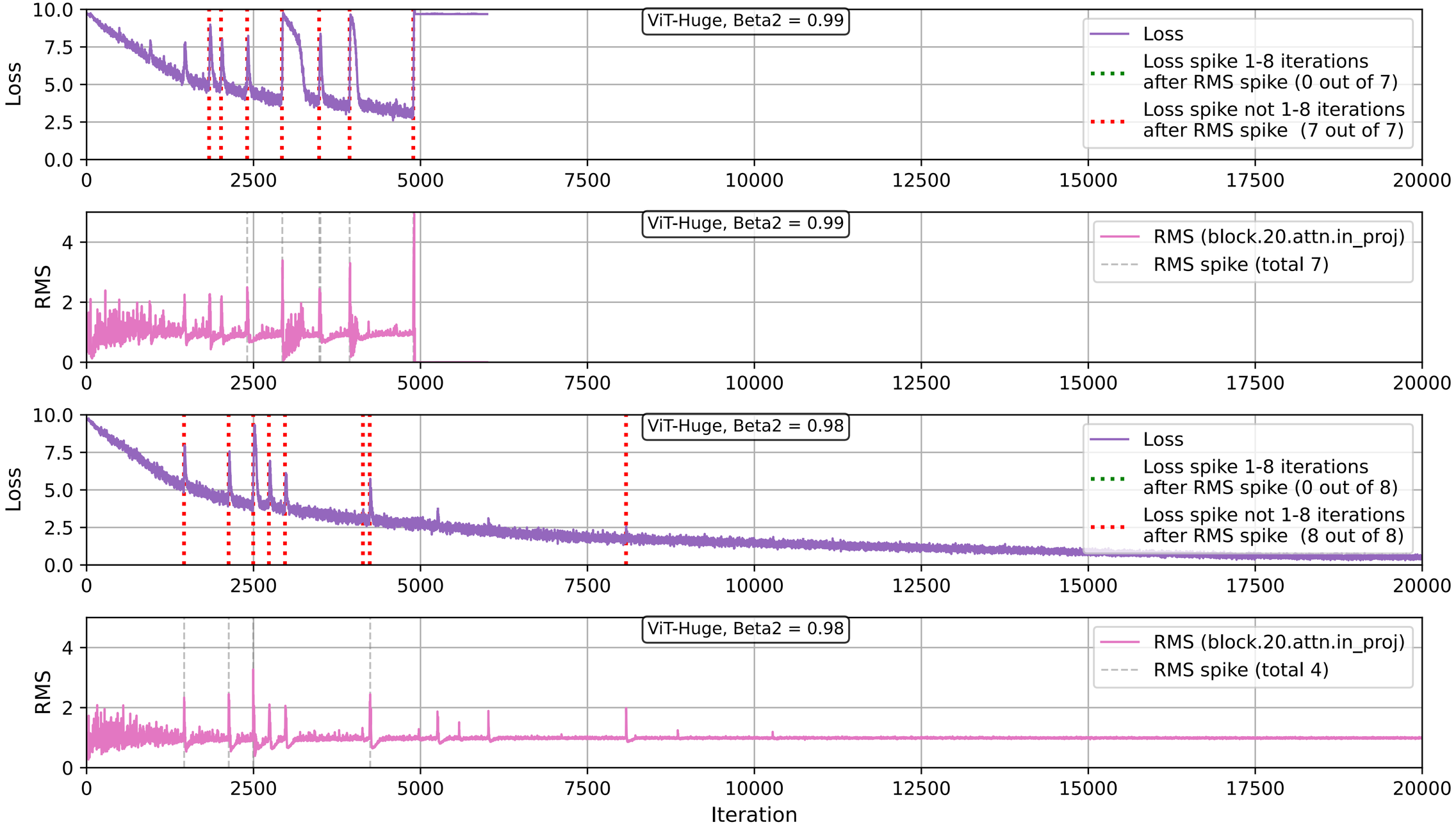
This is just one example… let’s come up with heuristic for documenting loss/rms spikes and use this for analysis



Chance of a loss spike randomly following an RMS spike is 1.0%


ViT-Huge, Beta2 = 0.99
Loss
Loss spike 1-8 iterations after RMS spike
Loss spike not 1-8 iterations after RMS spike
Loss
10.0
7.5
5.0
2.5
0.0
2800
3000
3200
3400
3600
3800
4000
4200
ViT-Huge, Beta2 = 0.99
RMS (visual.conv1.weight)
RMS spike
RMS
4
2
0
Iteration

For later layers, RMS spikes are postdictive not predictive of loss spikes


ViT-Huge, Beta2 = 0.99
Loss
Loss spike 1-8 iterations after RMS spike (0 out of 7)
Loss spike not 1-8 iterations after RMS spike (7 out of 7)
Loss
ViT-Huge, Beta2 = 0.99
RMS (block.20.attn.in_proj)
RMS spike (total 7)
RMS
ViT-Huge, Beta2 = 0.98
Loss
Loss spike 1-8 iterations after RMS spike (0 out of 8)
Loss spike not 1-8 iterations after RMS spike (8 out of 8)
ViT-Huge, Beta2 = 0.98
RMS (block.20.attn.in_proj)
RMS spike (total 4)
RMS
Iteration

- Has this ever been encountered before?
  - Yes, Shazeer and Stern found an out of date second moment estimator when developing AdaFactor and running experiments without warm-up
  - They propose a fix called “update clipping”
- Does anyone use “update clipping”?
  - No, because people do not get good performance with AdaFactor, but this is for other reasons (most likely the factored moments)
- So… let’s try AdamW + update clipping and refer to the result as StableAdamW

Slow down learning (reduce learning rate) if second moment estimator out of date

Algorithm 2 StableAdamW $\left(\{\alpha_t\}_{t=0}^{T}, \beta_1, \beta_2, \epsilon\right)$

$v_0, u_0 = \mathbf{0}$

for $t = 1$ **to** T **do**

$g_t = \nabla f(\theta_t)$

// apply correction term to debias moving avg.

$\hat{\beta}_1 = \beta_1 \cdot \frac{1-\beta_1^{t-1}}{1-\beta_1^t}$

$\hat{\beta}_2 = \beta_2 \cdot \frac{1-\beta_2^{t-1}}{1-\beta_2^t}$

// update moving averages

$v_t = \hat{\beta}_1 v_{t-1} + (1-\hat{\beta}_1) g_t$

$u_t = \hat{\beta}_2 u_{t-1} + (1-\hat{\beta}_2) g_t^2$

// for implementation convenience, the steps

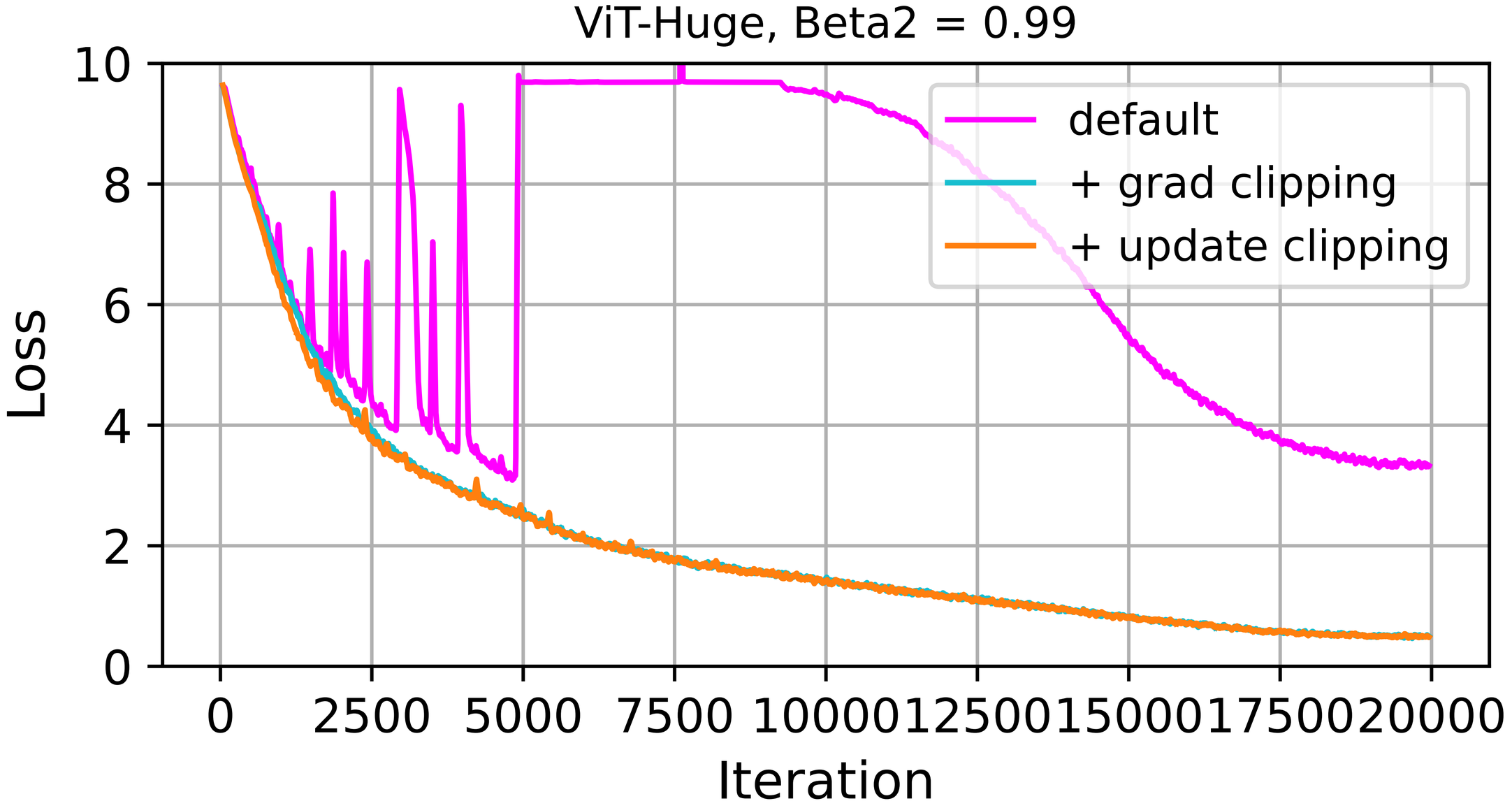
// below occur independently for each tensor

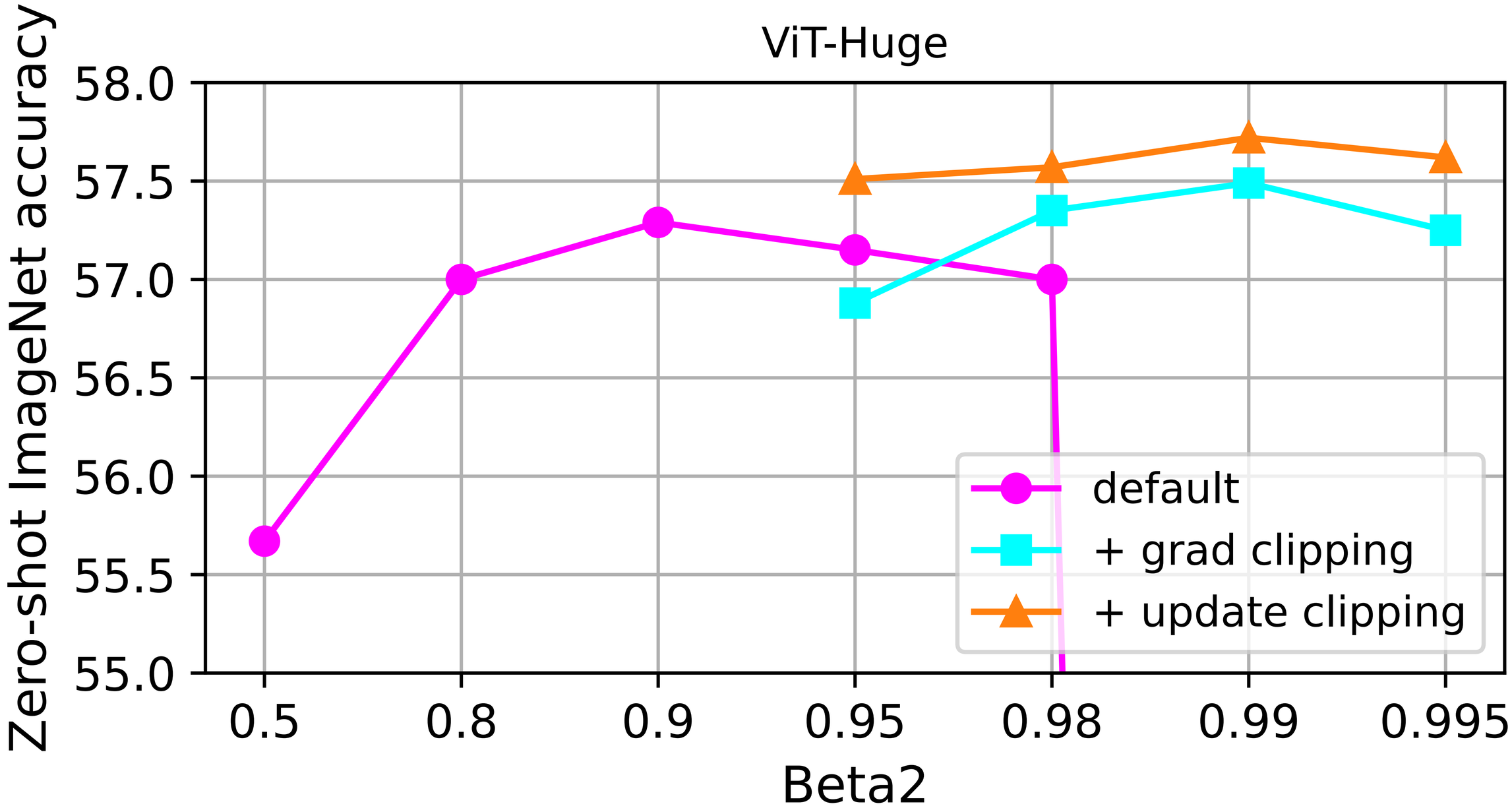
$\mathsf{RMS}_t = \sqrt{\mathbb{E}\left[g_t^2 / u_t\right]}$

// update parameters

$\eta_t = \alpha_t / \max\left(1, \mathsf{RMS}_t\right)$

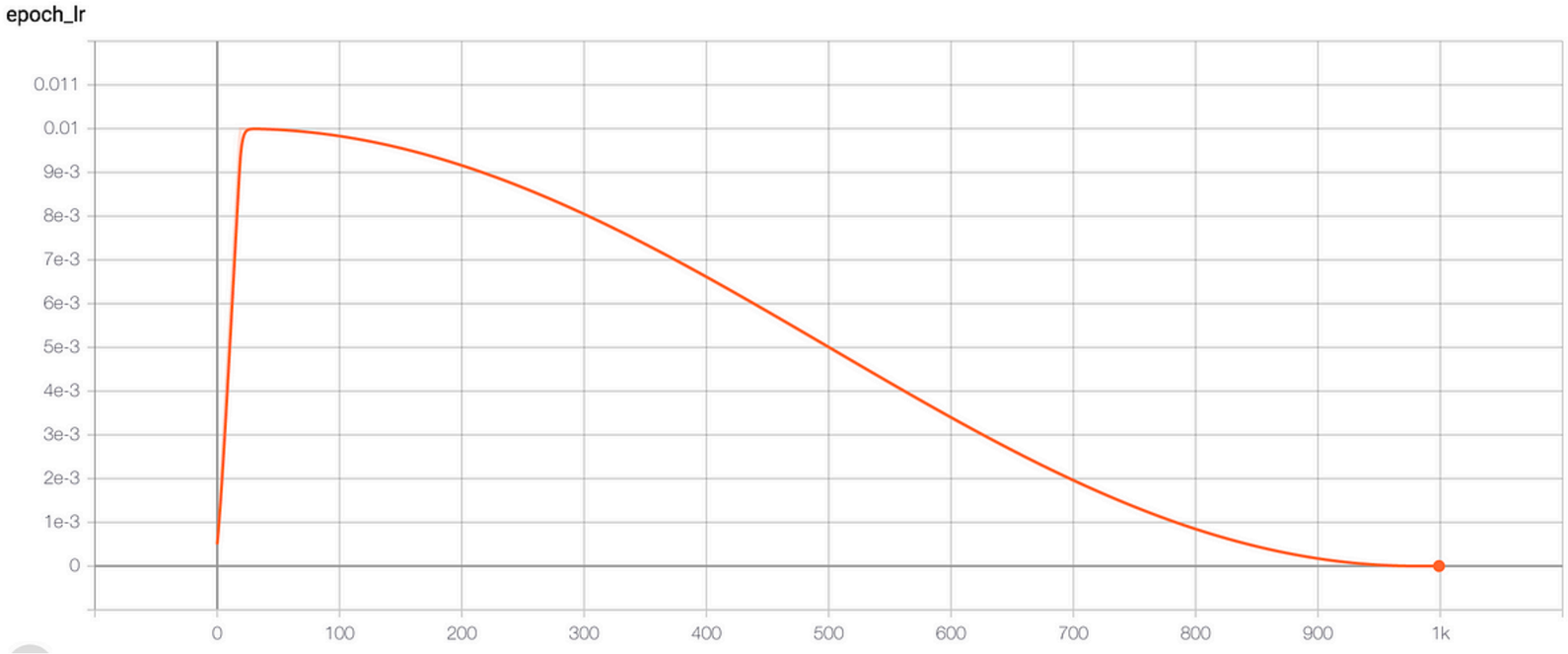
$\theta_t = \theta_{t-1} - \alpha_t \lambda \theta_{t-1} + \eta_t v_t / \left(\sqrt{u_t} + \epsilon\right)$


ViT-Huge, Beta2 = 0.99
Loss
Iteration
0
2
4
6
8
10
0
2500
5000
7500
10000
12500
15000
17500
20000
default
+ grad clipping
+ update clipping


ViT-Huge
Zero-shot ImageNet accuracy
58.0
57.5
57.0
56.5
56.0
55.5
55.0
0.5
0.8
0.9
0.95
0.98
0.99
0.995
Beta2
default
+ grad clipping
+ update clipping

Other tips & tricks for optimization

- Use around 5000 iterations of linear warm-up for the learning rate
- After that, use cosine-decay



In things still go wrong…

- Inspect your data
- Use a smaller learning rate
- Using bf16 instead of fp16

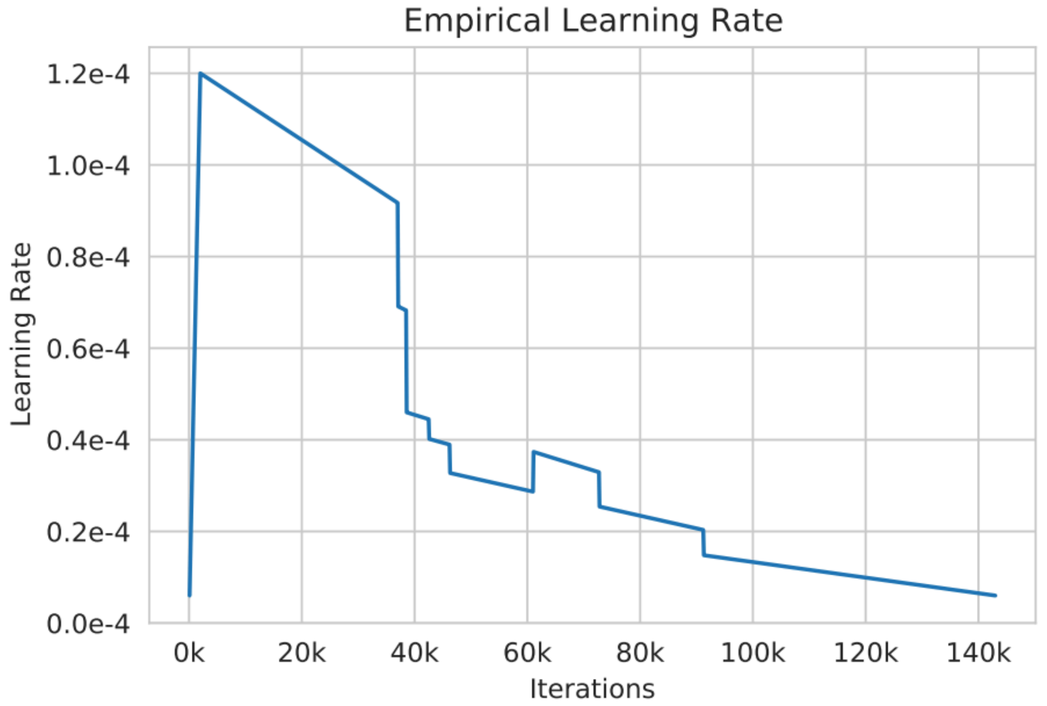


Figure 1: **Empirical LR schedule.** We found that lowering learning rate was helpful for avoiding instabilities.

What we didn’t cover and may be important

- Scaling learning rate and initialization based on width!

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

- There are other fixes for this, including scaling LR by RMS(weights) (see PaLM paper), or “principled” approaches such as mu-transfer

Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer

Greg Yang[*,×] **Edward J. Hu**[*,×,†] **Igor Babuschkin**[°] **Szymon Sidor**[°] **Xiaodong Liu**[×]
David Farhi[°] **Nick Ryder**[°] **Jakub Pachocki**[°] **Weizhu Chen**[×] **Jianfeng Gao**[×]
[×]Microsoft Corporation [°]OpenAI

Abstract

Hyperparameter (HP) tuning in deep learning is an expensive process, prohibitively so for neural networks (NNs) with billions of parameters. We show that, in the recently discovered Maximal Update Parametrization (μP), many optimal HPs remain stable even as model size changes. This leads to a new HP tuning paradigm we call *μTransfer*: parametrize the target model in μP, tune the HP indirectly on a smaller model, and *zero-shot transfer* them to the full-sized model, i.e., without directly tuning the latter at all. We verify μTransfer on Transformer and ResNet. For example, 1) by transferring pretraining HPs from a model of 13M parameters, we outperform published numbers of BERT-large (350M parameters), with a total tuning cost equivalent to pretraining BERT-large once; 2) by transferring from 40M parameters, we outperform published numbers of the 6.7B GPT-3 model, with tuning cost only 7% of total pretraining cost. A Pytorch implementation of our technique can be found at `github.com/microsoft/mup` and installable via `pip install mup`.