Effective Theory of Transformers



Emily Dinan, Sho Yaida, Susan Zhang [arXiv:2304.02034]

Meta

(Spinoff from)
Effective Theory of Deep Neural Networks



Dan Roberts, Sho Yaida, Boris Hanin [arXiv:2106.10165; Cambridge University Press]

Meta

(Branch of)
Physics of Machine Learning

Order-one fraction of people in this room

Meta

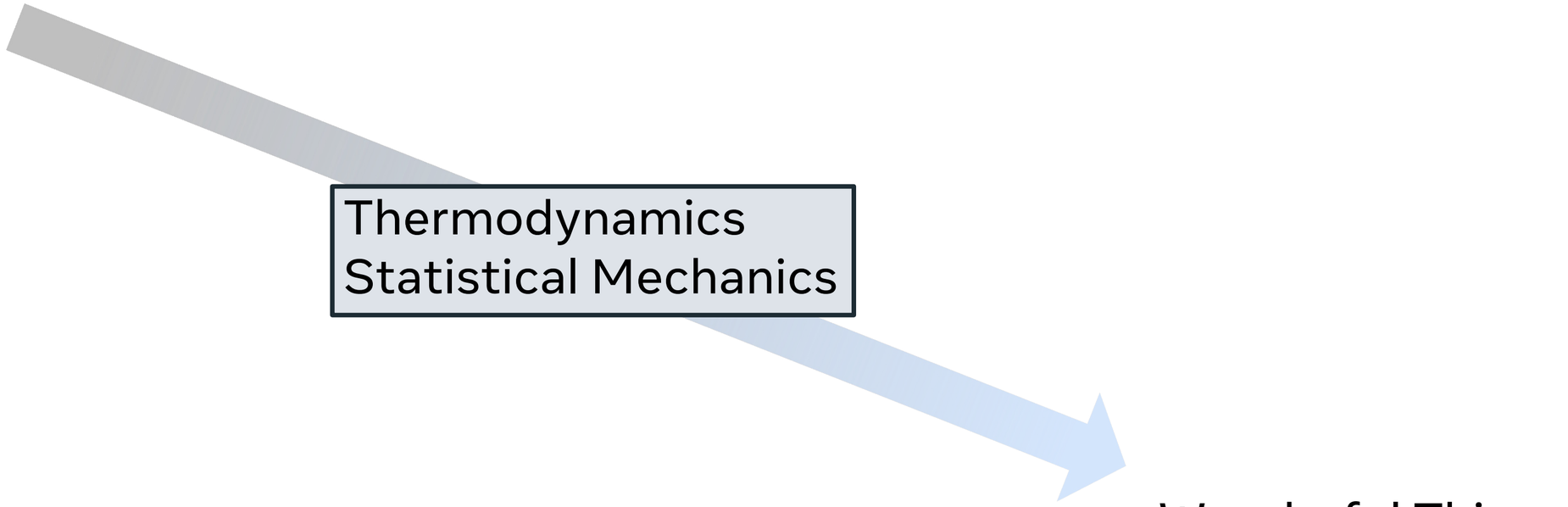
Historical Motivation

Steam Engine

Wonderful Things

Historical Motivation

Steam Engine



Wonderful Things

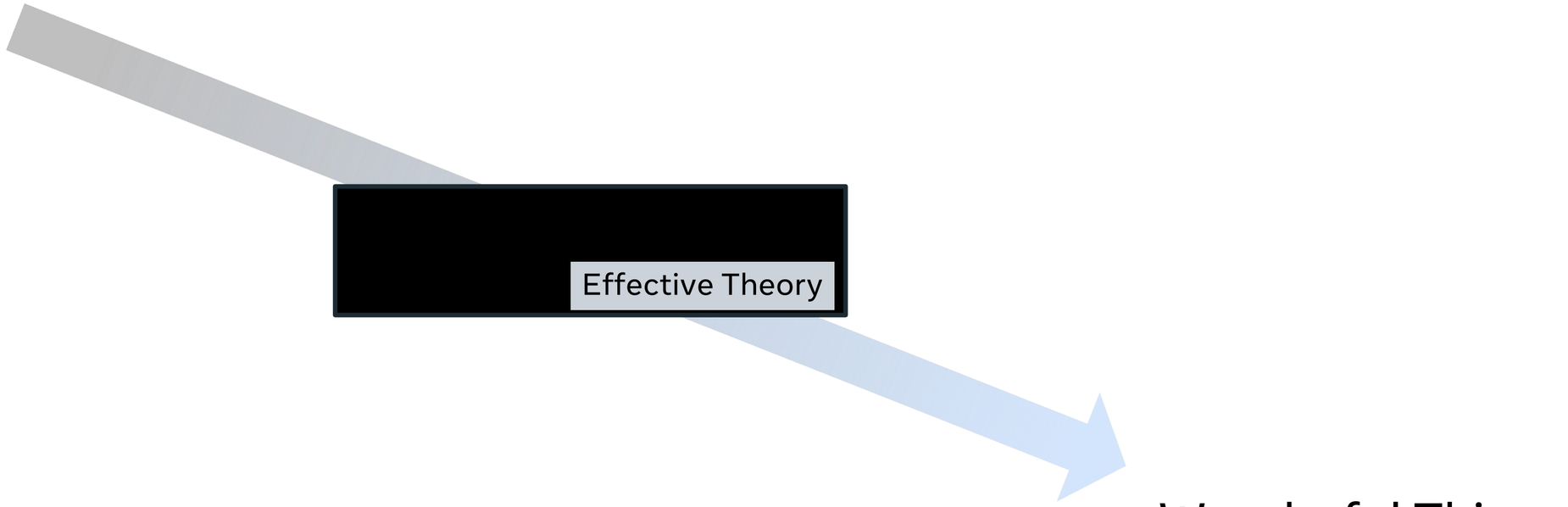
Historical Motivation

Deep Learning

Wonderful Things

Historical Motivation

Deep Learning



Wonderful Things

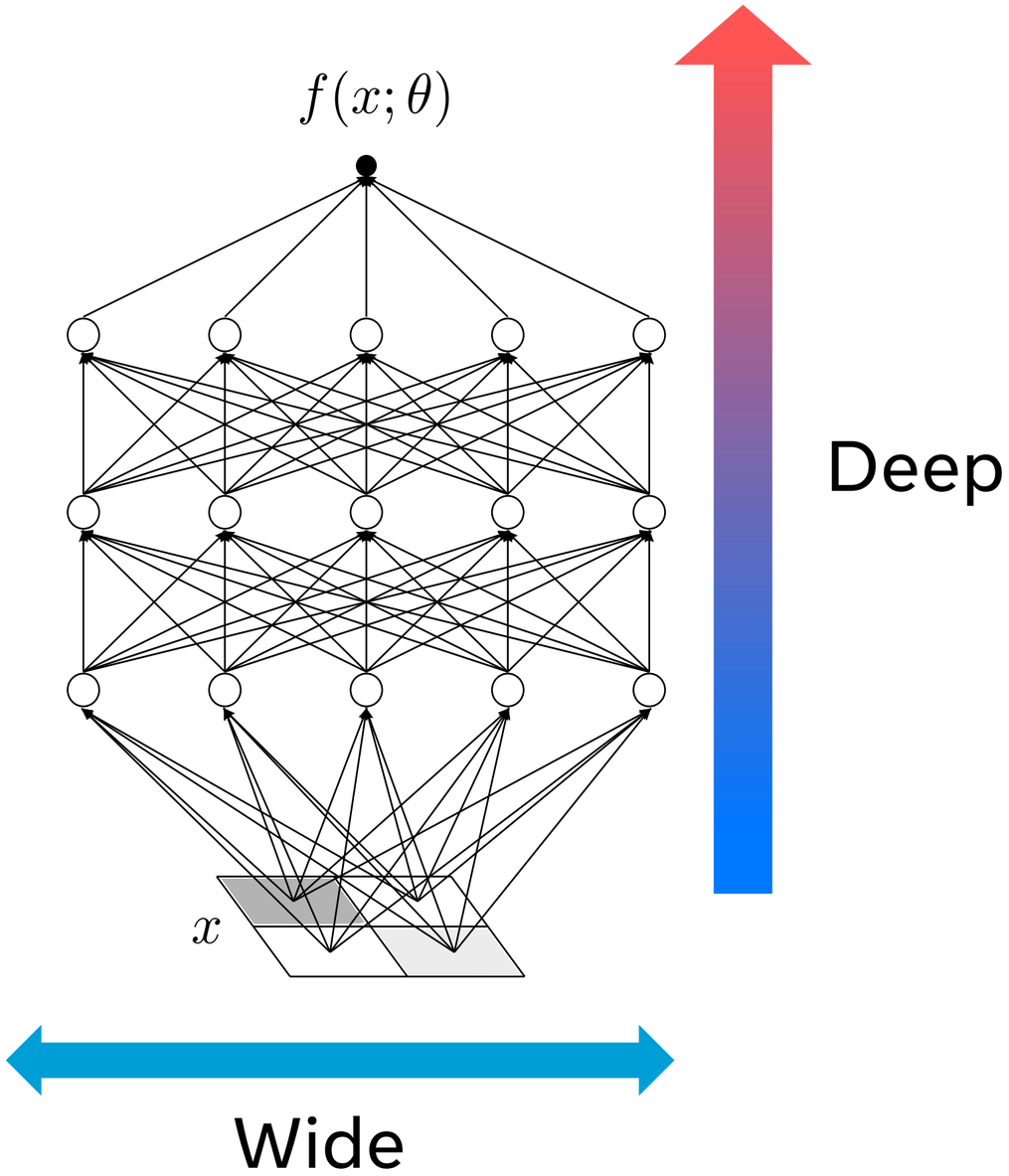
Outline

1. Simplification @Scale

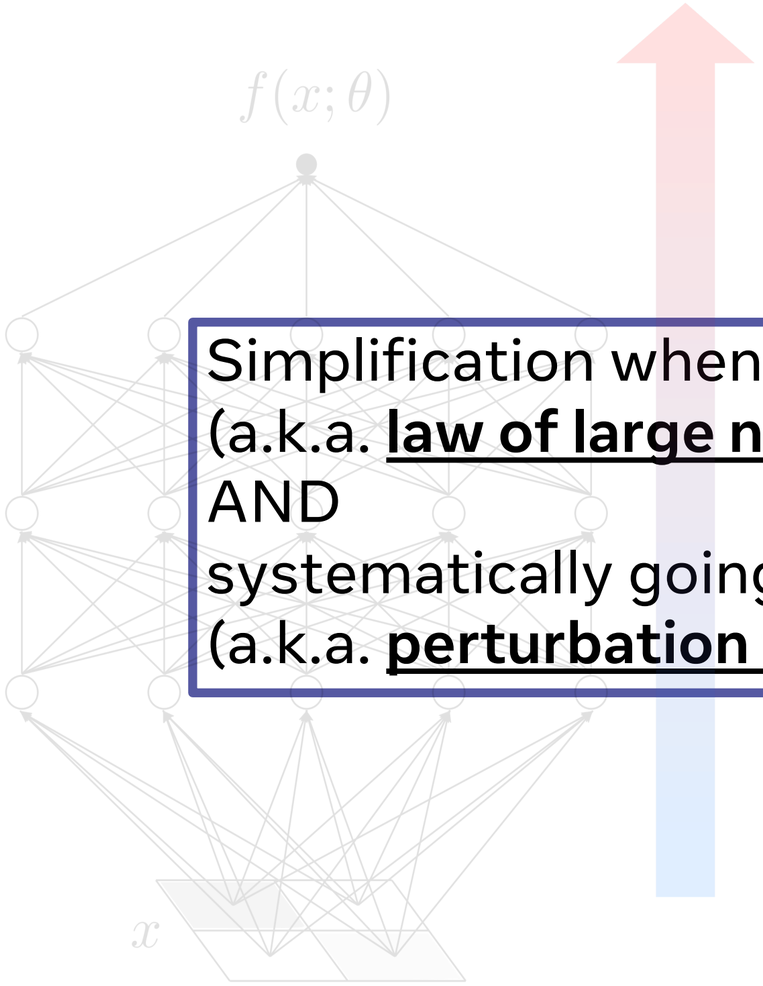
2. Hyperparameter Scaling for Transformers

1. Simplification @Scale

Physics of Wide & Deep Neural Networks



Physics of Wide & Deep Neural Networks

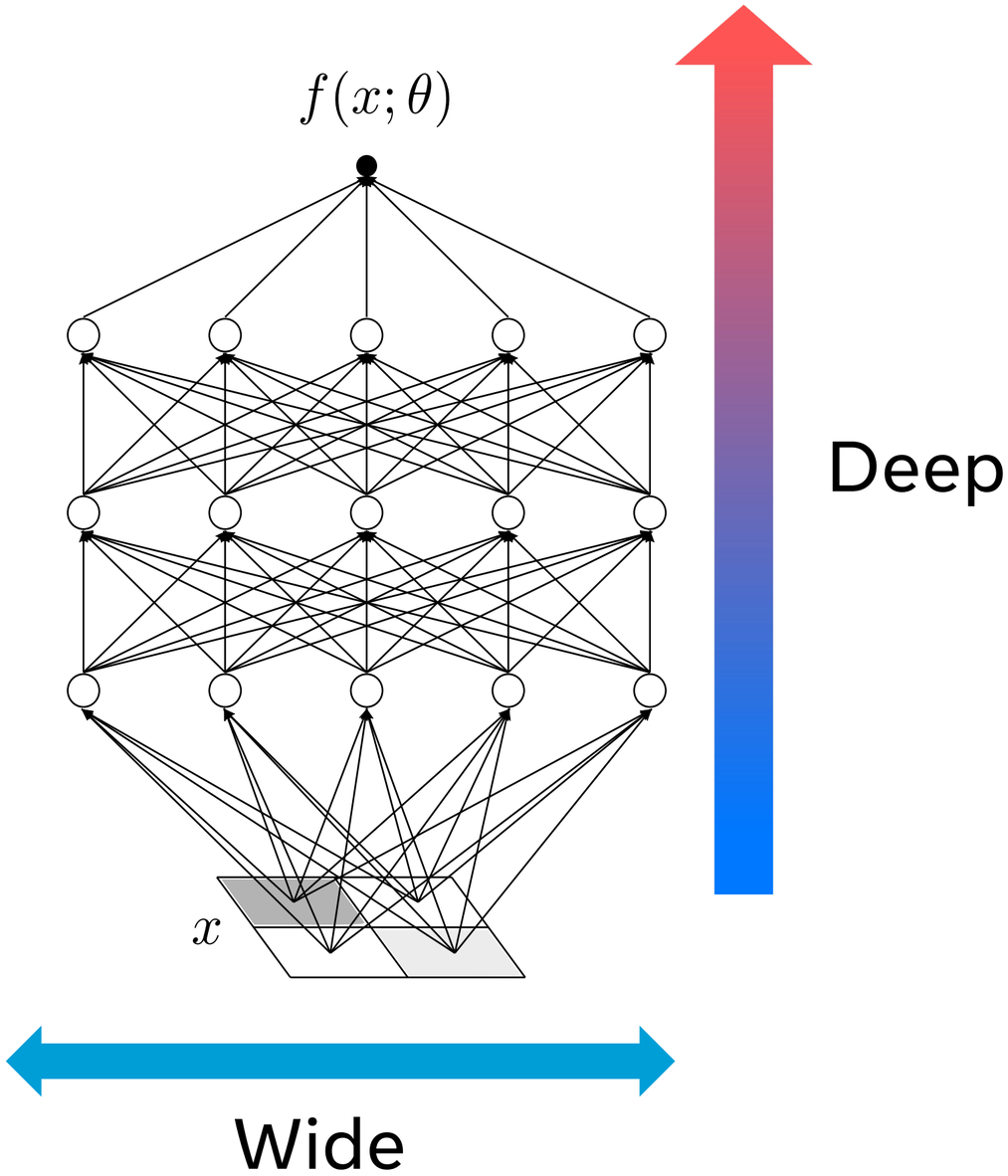


Deep

Wide

Simplification when there are infinitely-many neurons in hidden layers
(a.k.a. **law of large numbers**; a *free theory* at $\text{width} = \infty$)
AND
systematically going beyond that idealized limit
(a.k.a. **perturbation theory**; a *weakly-interacting theory* when $\text{width} \gg \text{depth}$)

Physics of Wide & Deep Neural Networks

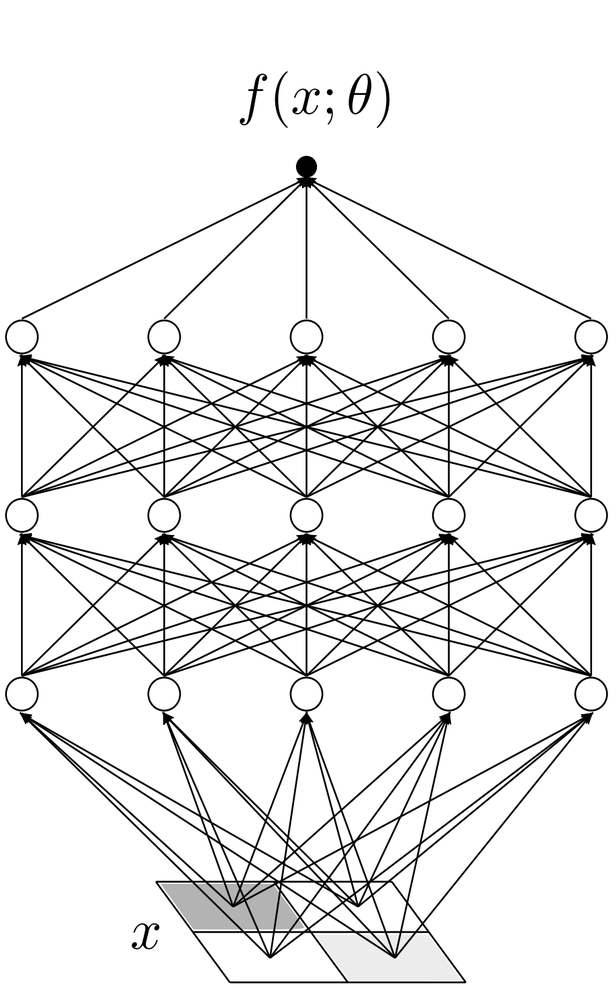


Effective theory works well when

$$\text{width}, \text{depth} \gg 1 \quad \& \quad \text{“}\frac{\text{depth}}{\text{width}} \ll 1\text{”}$$

i.e., typical large-scale setups!

Physics of Wide & Deep Neural Networks



Deep

Wide

Effective theory works well when

$$\text{width}, \text{depth} \gg 1 \quad \& \quad \text{“}\frac{\text{depth}}{\text{width}} \ll 1\text{”}$$

i.e., typical large-scale setups!

[D. Roberts, S. Yaida, B. Hanin: arXiv:2106.10165; Cambridge University Press
+many many many others]

Applied Physics of Wide & Deep Neural Networks

Effective theory ⇨ how to scale initialization/training hyperparameters

⇨ Bringing theory & practice closer together @ large scales

Applied Physics of Wide & Deep Neural Networks

Effective theory ⇒ how to scale initialization/training hyperparameters

⇒ Bringing theory & practice closer together @ large scales

- For Transformers: w/ Emily Dinan and Susan Zhang [arXiv:2304.02034]

- …

#TL;DR

- **(Meta-)Principle of Criticality**:

  “every preactivation/activation should be of order one, i.e., not too big, not too small”

  → `stddev ~ np.sqrt(1.0/fan_in)` for weights @ initialization

`fan_in` = number of signals coming into a layer

`fan_out` = number of signals coming out of a layer

#TL;DR

- **(Meta-)Principle of Criticality**:

  “every preactivation/activation should be of order one, i.e., not too big, not too small”

  → `stddev ~ np.sqrt(1.0/fan_in)` for weights @ initialization

- **(Meta-)Principle of Equivalence**:

  “every layer should contribute equally to learning (that is, to the neural tangent kernel)”

  → `lr ~ 1.0/fan_in` for SGD

  → `lr ~ 1.0/(fan_in*np.sqrt(fan_out))` for AdamW

`fan_in` = number of signals coming into a layer

`fan_out` = number of signals coming out of a layer

#TL;DR

- **(Meta-)Principle of Criticality**:

  “every preactivation/activation should be of order one, i.e., not too big, not too small”

  → `stddev ~ np.sqrt(1.0/fan_in)` for weights @ initialization

- **(Meta-)Principle of Equivalence**:

  “every layer should contribute equally to learning (that is, to the neural tangent kernel)”

  → `lr ~ 1.0/fan_in` for SGD

  → `lr ~ 1.0/(fan_in*np.sqrt(fan_out))` for AdamW

* Caveat 1: we need to be careful with “`fan_in`” for sparse inputs.
* Caveat 2: stem—head weight tying requires us to rescale output logits.
* Caveat 3: we can further fine-tune “1.0” but that’s largely moot due to `LayerNorms`.

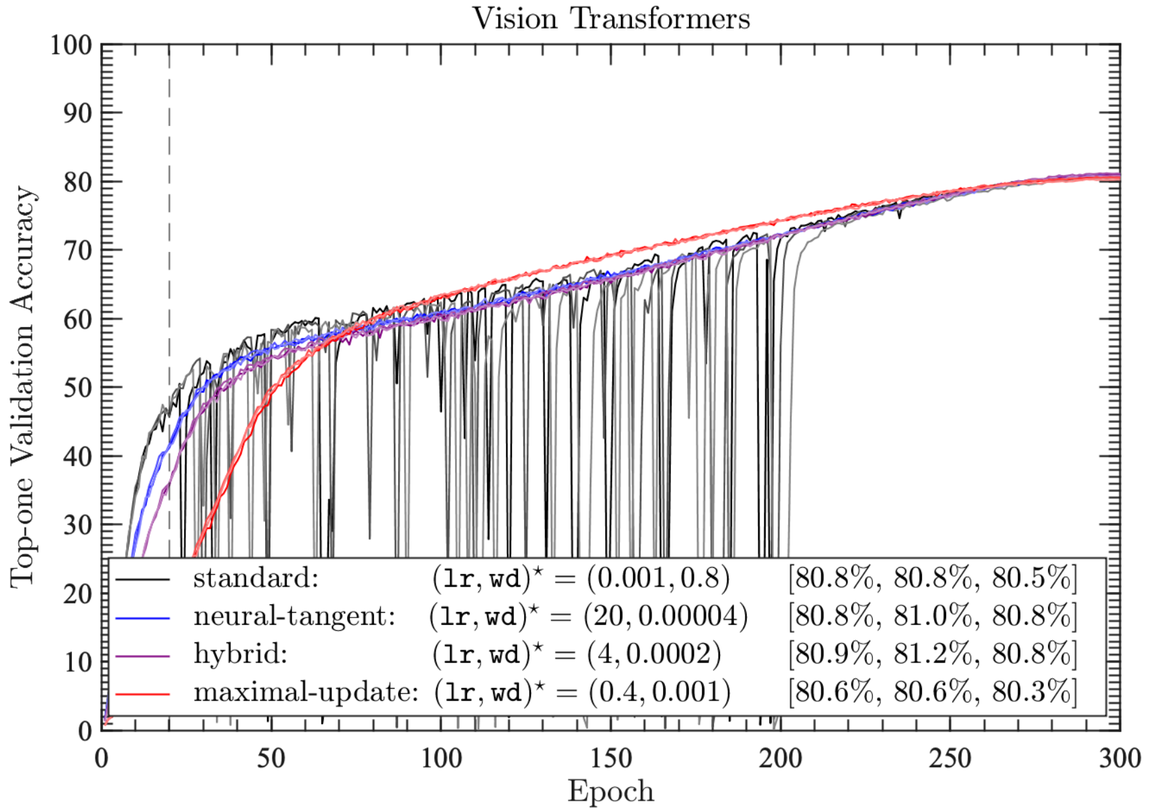
* Meta-Caveat:
preceding and following discussions assume the *neural-tangent* scaling of various hyperparameters;
for more general cases including the *maximal-update* scaling (proposed by Greg Yang & Edward Hu), see

arXiv:2210.04909 (S. Yaida, “Meta-Principled Family of Hyperparameter Scaling Strategies”)
or
various footnotes in our paper [arXiv:2304.02034] under presentation

2. Hyperparameter Scaling for Transformers

`bias=elementwise_affine=False`

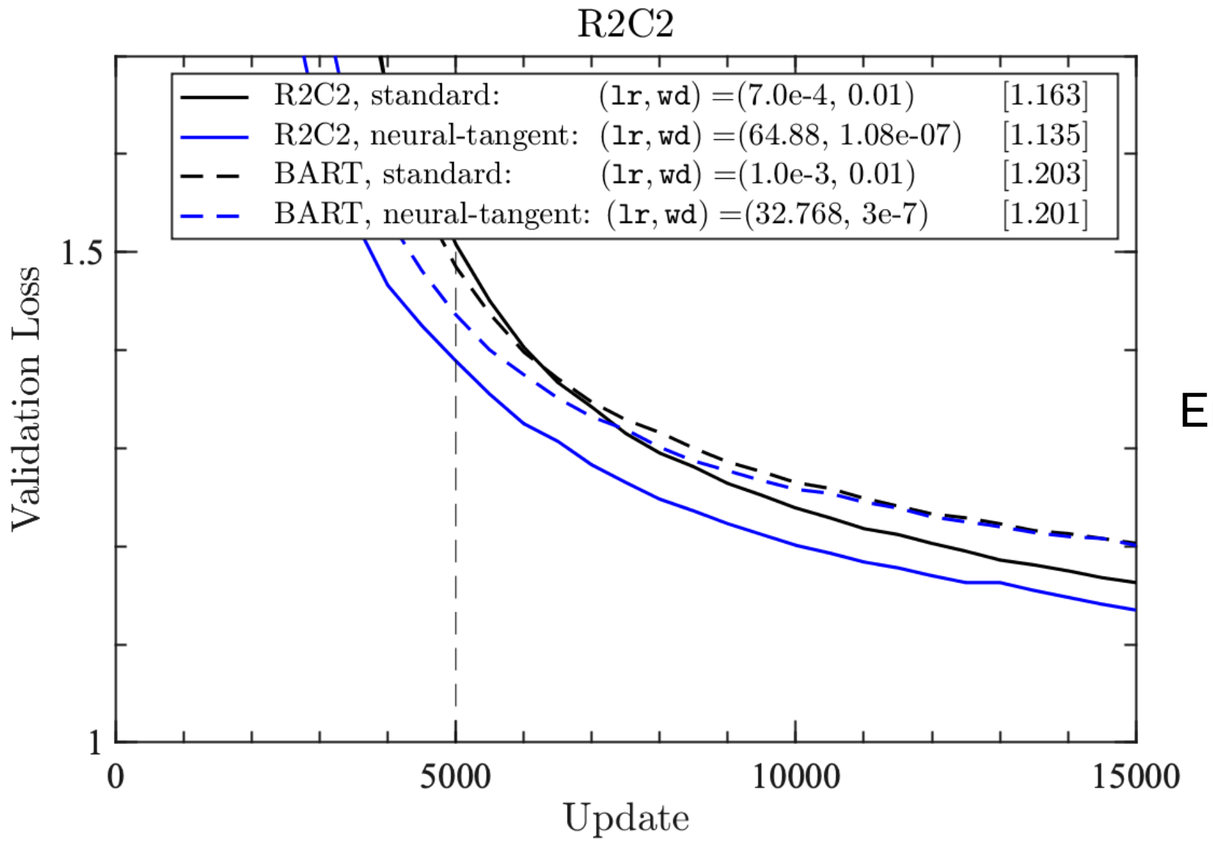
Does it work?



It seems to work well here. ☺

One curious observation:
neural-tangent scaling *increases* the learning rate for positional-embedding parameters by ~*1000 relative to the standard uniform scaling, and it makes the training *stabler*.

Does it work?



Encouraging but not conclusive…

Technical-looking Summary 1: Initialization Hyperparameters

`stddev ~ np.sqrt(1.0/fan_in)`

that their covariances should be scaled as

$$\mathbb{E}\left[W^{\text{patch}}_{i_1 j_1} W^{\text{patch}}_{i_2 j_2}\right] = \left(\frac{C_{\text{patch}}}{n_{\text{patch}}}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \quad \Big|\Big| \quad \mathbb{E}\left[W^{\text{WE}}_{i_1 j_1} W^{\text{WE}}_{i_2 j_2}\right] = \left(C_{\text{WE}}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \tag{1.25}$$

$$\mathbb{E}\left[b^{\text{PE}}_{t_1;i_1} b^{\text{PE}}_{t_2;i_2}\right] = \left(C_{\text{PE}}\right) \delta_{t_1 t_2} \delta_{i_1 i_2} \,, \tag{1.26}$$

$$\mathbb{E}\left[Q^{h_1}_{c_1 i_1} Q^{h_2}_{c_2 i_2}\right] = \left(\frac{C_Q}{n}\right) \delta_{c_1 c_2} \delta_{i_1 i_2} \delta^{h_1 h_2} \,, \tag{1.27}$$

$$\mathbb{E}\left[K^{h_1}_{c_1 i_1} K^{h_2}_{c_2 i_2}\right] = \left(\frac{C_K}{n}\right) \delta_{c_1 c_2} \delta_{i_1 i_2} \delta^{h_1 h_2} \,, \tag{1.28}$$

$$\mathbb{E}\left[V^{h_1}_{c_1 i_1} V^{h_2}_{c_2 i_2}\right] = \left(\frac{C_V}{n}\right) \delta_{c_1 c_2} \delta_{i_1 i_2} \delta^{h_1 h_2} \,, \tag{1.29}$$

$$\mathbb{E}\left[U^{h_1}_{i_1 c_1} U^{h_2}_{i_2 c_2}\right] = \left(\frac{C_U}{n}\right) \delta_{i_1 i_2} \delta_{c_1 c_2} \delta^{h_1 h_2} \,, \tag{1.30}$$

$$\mathbb{E}\left[W_{i_1 j_1} W_{i_2 j_2}\right] = \left(\frac{C_W}{n}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \tag{1.31}$$

$$\mathbb{E}\left[X_{i_1 j_1} X_{i_2 j_2}\right] = \left(\frac{C_X}{Mn}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \tag{1.32}$$

$$\mathbb{E}\left[W^{\text{head}}_{i_1 j_1} W^{\text{head}}_{i_2 j_2}\right] = \left(\frac{C_{\text{head}}}{n}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \;\; b^{\text{head}}_i = 0 \quad \Big|\Big| \quad \mathcal{N}_{\text{rescale}} = \sqrt{\frac{1}{n}} \,, \tag{1.33}$$

where $\mathbb{E}\left[\cdot\right]$ denotes an expectation value with respect to the initialization distribution, δ_{ij} is the Kronecker delta (i.e., $\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ when $i \neq j$), and all the initialization hyperparameters C_G are order-one numbers for each group G of model parameters. Here, by

Technical-looking Summary 2: Training Hyperparameters (LR)

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

of $\sim |g^\mu|$ modifies the above relative learning-rate factors to

$$\widetilde{\lambda}_{W^{\text{patch}}} = \left(\frac{1}{n_{\text{patch}}\sqrt{n}}\right)\widetilde{\Lambda}_{W^{\text{patch}}} \quad \Big\| \quad \widetilde{\lambda}_{W^{\text{WE}}} = \left(\frac{1}{\sqrt{n}}\right)\widetilde{\Lambda}_{W^{\text{WE}}}\,, \tag{1.108}$$

$$\widetilde{\lambda}_{b^{\text{PE}}} = \left(\frac{1}{\sqrt{n}}\right)\widetilde{\Lambda}_{b^{\text{PE}}}\,, \tag{1.109}$$

$$\widetilde{\lambda}_{Q} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{Q}\,, \tag{1.110}$$

$$\widetilde{\lambda}_{K} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{K}\,, \tag{1.111}$$

$$\widetilde{\lambda}_{V} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{V}\,, \tag{1.112}$$

$$\widetilde{\lambda}_{U} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{U}\,, \tag{1.113}$$

$$\widetilde{\lambda}_{W} = \left(\frac{1}{n\sqrt{Mn}}\right)\widetilde{\Lambda}_{W}\,, \tag{1.114}$$

$$\widetilde{\lambda}_{X} = \left(\frac{1}{Mn\sqrt{n}}\right)\widetilde{\Lambda}_{X}\,, \tag{1.115}$$

$$\widetilde{\lambda}_{W^{\text{head}}} = \left(\frac{1}{n\sqrt{n_{\text{out}}}}\right)\widetilde{\Lambda}_{W^{\text{head}}}\,, \quad \widetilde{\lambda}_{b^{\text{head}}} = \left(\frac{1}{\sqrt{n_{\text{out}}}}\right)\widetilde{\Lambda}_{b^{\text{head}}}\,, \tag{1.116}$$

where – like the order-one initialization hyperparameters C_G's – $\widetilde{\Lambda}_G$'s are order-one training hyperparameters which we could in principle tune but won't.[25]

```
stddev ~ np.sqrt(1.0/fan_in)
```

```
lr ~ 1.0/(fan_in*np.sqrt(fan_out))
```

Attention Blocks

$$\text{stddev} \sim \text{np.sqrt}(1.0/\text{fan\_in})$$

$$\text{lr} \sim 1.0/(\text{fan\_in}*\text{np.sqrt}(\text{fan\_out}))$$

The query, key, value, and out_proj weights all have (*)

$$\text{fan\_in}=\text{embedding\_dim}$$

$$\text{fan\_out}=\text{embedding\_dim}$$

(*Ok, with the traditional usage of terminology, strictly speaking, it depends on how you code QKV, but it doesn't really matter much here.)

Attention Blocks

$$\text{stddev} \sim \text{np.sqrt}(1.0/\text{fan\_in})$$

$$\text{lr} \sim 1.0/(\text{fan\_in}*\text{np.sqrt}(\text{fan\_out}))$$

The query, key, value, and out_proj weights all have (*)

```
fan_in=embedding_dim
fan_out=embedding_dim
```

```
stddev=np.sqrt(1.0/embedding_dim)
lr=np.sqrt(1.0/embedding_dim**3)
```

(*Ok, with the traditional usage of terminology, strictly speaking, it depends on how you code QKV, but it doesn't really matter much here.)

Attention Blocks

`stddev ~ np.sqrt(1.0/fan_in)`

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

One comment about the famous query—key dot product normalization:

$$\frac{1}{\sqrt{C}}\mathbf{q}\cdot\mathbf{k} \quad \text{where} \quad C = \#\ \text{of channels per head} = \frac{\texttt{embedding\_dim}}{\texttt{n\_heads}}$$

Attention Blocks

`stddev ~ np.sqrt(1.0/fan_in)`

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

One comment about the famous query—key dot product normalization:

$$\frac{1}{\sqrt{C}}\mathbf{q}\cdot\mathbf{k} \quad \text{where} \quad C = \#\ \text{of channels per head} = \frac{\texttt{embedding\_dim}}{\texttt{n\_heads}}$$

This is theoretically correct (while Tensor Program V paper does something else).

FC Blocks

$$stddev \sim np.sqrt(1.0/fan\_in)$$

$$lr \sim 1.0/(fan\_in*np.sqrt(fan\_out))$$

The 1st layer

```
fan_in=embedding_dim
fan_out=M*embedding_dim
```

The 2nd layer

```
fan_in=M*embedding_dim
fan_out=embedding_dim
```

FC Blocks

$$stddev \sim np.sqrt(1.0/fan\_in)$$

$$lr \sim 1.0/(fan\_in*np.sqrt(fan\_out))$$

The 1st layer

```
fan_in=embedding_dim
fan_out=M*embedding_dim
```

```
stddev=np.sqrt(1.0/embedding_dim)
lr=np.sqrt(1.0/(M*(embedding_dim**3)))
```

The 2nd layer

```
fan_in=M*embedding_dim
fan_out=embedding_dim
```

```
stddev=np.sqrt(1.0/(M*embedding_dim))
lr=np.sqrt(1.0/((M**2)*(embedding_dim**3)))
```

Word-Embedding Parameters

$\texttt{stddev} \sim \texttt{np.sqrt(1.0/fan\_in)}$

$\texttt{lr} \sim \texttt{1.0/(fan\_in*np.sqrt(fan\_out))}$

Naively...,

```
fan_in=n_vocab
fan_out=embedding_dim
```

Word-Embedding Parameters

`stddev ~ np.sqrt(1.0/fan_in)`

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

Because of one-hot-ness,

~~`fan_in=n_vocab`~~ `fan_in=1`

`fan_out=embedding_dim`

Word-Embedding Parameters

stddev ~ np.sqrt(1.0/fan_in)

lr ~ 1.0/(fan_in*np.sqrt(fan_out))

Because of one-hot-ness,

~~fan_in=n_vocab~~ fan_in=1

fan_out=embedding_dim

stddev=1.0

lr=np.sqrt(1.0/embedding_dim)

Word-Embedding Parameters

`stddev ~ np.sqrt(1.0/fan_in)`

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

Because of one-hot-ness,

~~`fan_in=n_vocab`~~ `fan_in=1`

`fan_out=embedding_dim`

```
stddev=1.0
lr=np.sqrt(1.0/embedding_dim)
```

* Lesson 1: be mindful with sparse features.
* Lesson 2: the learning rate is relatively much higher than those for other layers.

(Word-Embedding Parameters)^T

$\texttt{stddev} \sim \texttt{np.sqrt(1.0/fan\_in)}$

$\texttt{lr} \sim \texttt{1.0/(fan\_in*np.sqrt(fan\_out))}$

Because of stem—head weight tying,

$$\texttt{logit}_i = \sum_{j=1}^{\texttt{embedding\_dim}} (\texttt{WE.weight})_{ji} (\texttt{last\_features})_j$$

(Word-Embedding Parameters)^T

$\texttt{stddev} \sim \texttt{np.sqrt(1.0/fan\_in)}$

$\texttt{lr} \sim \texttt{1.0/(fan\_in*np.sqrt(fan\_out))}$

Because of stem—head weight tying,

$$\texttt{logit}_i = \sum_{j=1}^{\texttt{embedding\_dim}} (\texttt{WE.weight})_{ji} (\texttt{last\_features})_j \sim O(\sqrt{\texttt{embedding\_dim}})$$

(Word-Embedding Parameters)^T

`stddev ~ np.sqrt(1.0/fan_in)`

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

Because of stem—head weight tying,

$$\texttt{logit}_i = \sum_{j=1}^{\texttt{embedding\_dim}} (\texttt{WE.weight})_{ji}(\texttt{last\_features})_j \sim O(\sqrt{\texttt{embedding\_dim}})$$

```
logit=logit/np.sqrt(embedding_dim)
```

Positional-Embedding Parameters

$$\text{stddev} \sim \text{np.sqrt}(1.0/\text{fan\_in})$$

$$\text{lr} \sim 1.0/(\text{fan\_in}*\text{np.sqrt}(\text{fan\_out}))$$

Finally, quickly, for positional-embedding (additive, like bias), we have

```
fan_in=1
fan_out=embedding_dim
```

```
sttdev=1.0 or 0.0 or something
lr=np.sqrt(1.0/embedding_dim)
```

Technical-looking Summary 1: Initialization Hyperparameters

`stddev ~ np.sqrt(1.0/fan_in)`

that their covariances should be scaled as

$$\mathbb{E}\left[W^{\text{patch}}_{i_1 j_1} W^{\text{patch}}_{i_2 j_2}\right] = \left(\frac{C_{\text{patch}}}{n_{\text{patch}}}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \quad \Big|\Big| \quad \mathbb{E}\left[W^{\text{WE}}_{i_1 j_1} W^{\text{WE}}_{i_2 j_2}\right] = \left(C_{\text{WE}}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \tag{1.25}$$

$$\mathbb{E}\left[b^{\text{PE}}_{t_1;i_1} b^{\text{PE}}_{t_2;i_2}\right] = \left(C_{\text{PE}}\right) \delta_{t_1 t_2} \delta_{i_1 i_2} \,, \tag{1.26}$$

$$\mathbb{E}\left[Q^{h_1}_{c_1 i_1} Q^{h_2}_{c_2 i_2}\right] = \left(\frac{C_Q}{n}\right) \delta_{c_1 c_2} \delta_{i_1 i_2} \delta^{h_1 h_2} \,, \tag{1.27}$$

$$\mathbb{E}\left[K^{h_1}_{c_1 i_1} K^{h_2}_{c_2 i_2}\right] = \left(\frac{C_K}{n}\right) \delta_{c_1 c_2} \delta_{i_1 i_2} \delta^{h_1 h_2} \,, \tag{1.28}$$

$$\mathbb{E}\left[V^{h_1}_{c_1 i_1} V^{h_2}_{c_2 i_2}\right] = \left(\frac{C_V}{n}\right) \delta_{c_1 c_2} \delta_{i_1 i_2} \delta^{h_1 h_2} \,, \tag{1.29}$$

$$\mathbb{E}\left[U^{h_1}_{i_1 c_1} U^{h_2}_{i_2 c_2}\right] = \left(\frac{C_U}{n}\right) \delta_{i_1 i_2} \delta_{c_1 c_2} \delta^{h_1 h_2} \,, \tag{1.30}$$

$$\mathbb{E}\left[W_{i_1 j_1} W_{i_2 j_2}\right] = \left(\frac{C_W}{n}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \tag{1.31}$$

$$\mathbb{E}\left[X_{i_1 j_1} X_{i_2 j_2}\right] = \left(\frac{C_X}{Mn}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \tag{1.32}$$

$$\mathbb{E}\left[W^{\text{head}}_{i_1 j_1} W^{\text{head}}_{i_2 j_2}\right] = \left(\frac{C_{\text{head}}}{n}\right) \delta_{i_1 i_2} \delta_{j_1 j_2} \,, \;\; b^{\text{head}}_i = 0 \quad \Big|\Big| \quad \mathcal{N}_{\text{rescale}} = \sqrt{\frac{1}{n}} \,, \tag{1.33}$$

where $\mathbb{E}\left[\cdot\right]$ denotes an expectation value with respect to the initialization distribution, δ_{ij} is the Kronecker delta (i.e., $\delta_{ij} = 1$ when $i = j$ and $\delta_{ij} = 0$ when $i \neq j$), and all the initialization hyperparameters C_G are order-one numbers for each group G of model parameters. Here, by

Technical-looking Summary 2: Training Hyperparameters (LR)

`lr ~ 1.0/(fan_in*np.sqrt(fan_out))`

of $\sim |g^\mu|$ modifies the above relative learning-rate factors to

$$\widetilde{\lambda}_{W^{\text{patch}}} = \left(\frac{1}{n_{\text{patch}}\sqrt{n}}\right)\widetilde{\Lambda}_{W^{\text{patch}}} \quad \Big\| \quad \widetilde{\lambda}_{W^{\text{WE}}} = \left(\frac{1}{\sqrt{n}}\right)\widetilde{\Lambda}_{W^{\text{WE}}}\,, \tag{1.108}$$

$$\widetilde{\lambda}_{b^{\text{PE}}} = \left(\frac{1}{\sqrt{n}}\right)\widetilde{\Lambda}_{b^{\text{PE}}}\,, \tag{1.109}$$

$$\widetilde{\lambda}_{Q} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{Q}\,, \tag{1.110}$$

$$\widetilde{\lambda}_{K} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{K}\,, \tag{1.111}$$

$$\widetilde{\lambda}_{V} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{V}\,, \tag{1.112}$$

$$\widetilde{\lambda}_{U} = \left(\frac{1}{n\sqrt{n}}\right)\widetilde{\Lambda}_{U}\,, \tag{1.113}$$

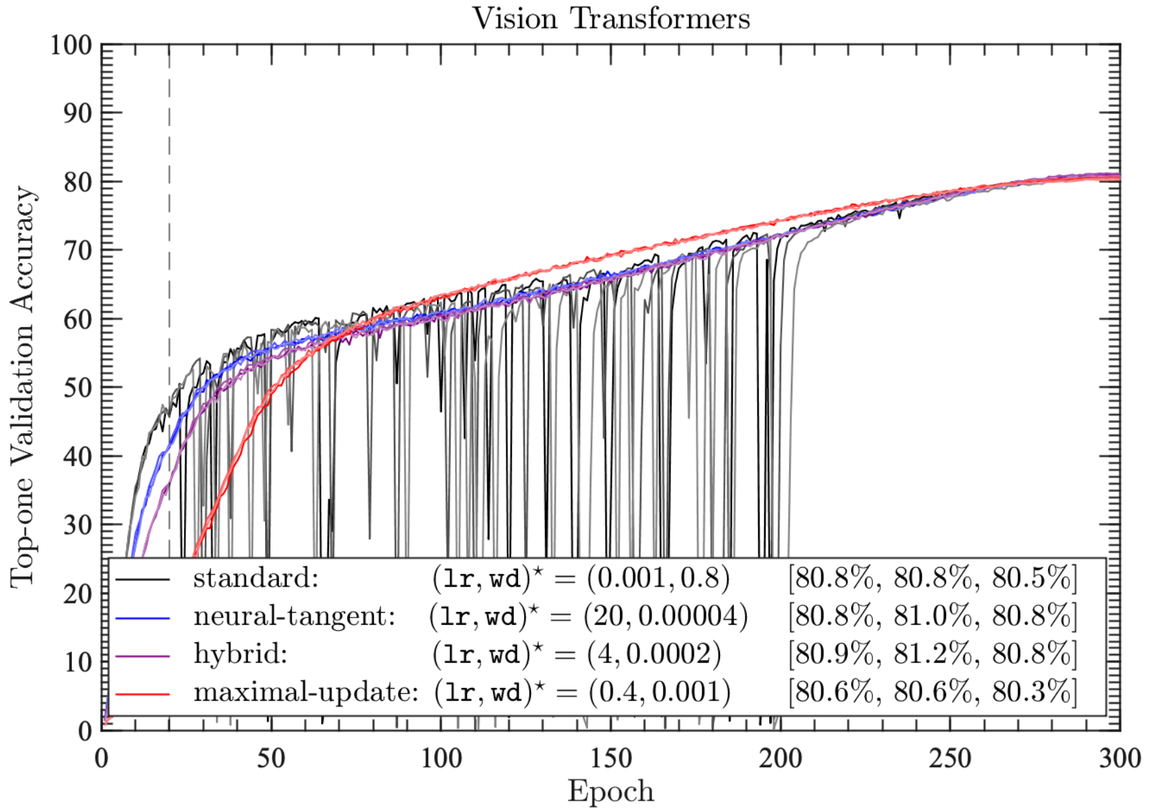
$$\widetilde{\lambda}_{W} = \left(\frac{1}{n\sqrt{Mn}}\right)\widetilde{\Lambda}_{W}\,, \tag{1.114}$$

$$\widetilde{\lambda}_{X} = \left(\frac{1}{Mn\sqrt{n}}\right)\widetilde{\Lambda}_{X}\,, \tag{1.115}$$

$$\widetilde{\lambda}_{W^{\text{head}}} = \left(\frac{1}{n\sqrt{n_{\text{out}}}}\right)\widetilde{\Lambda}_{W^{\text{head}}}\,, \quad \widetilde{\lambda}_{b^{\text{head}}} = \left(\frac{1}{\sqrt{n_{\text{out}}}}\right)\widetilde{\Lambda}_{b^{\text{head}}}\,, \tag{1.116}$$

where – like the order-one initialization hyperparameters C_G's – $\widetilde{\Lambda}_G$'s are order-one training hyperparameters which we could in principle tune but won't.[25]

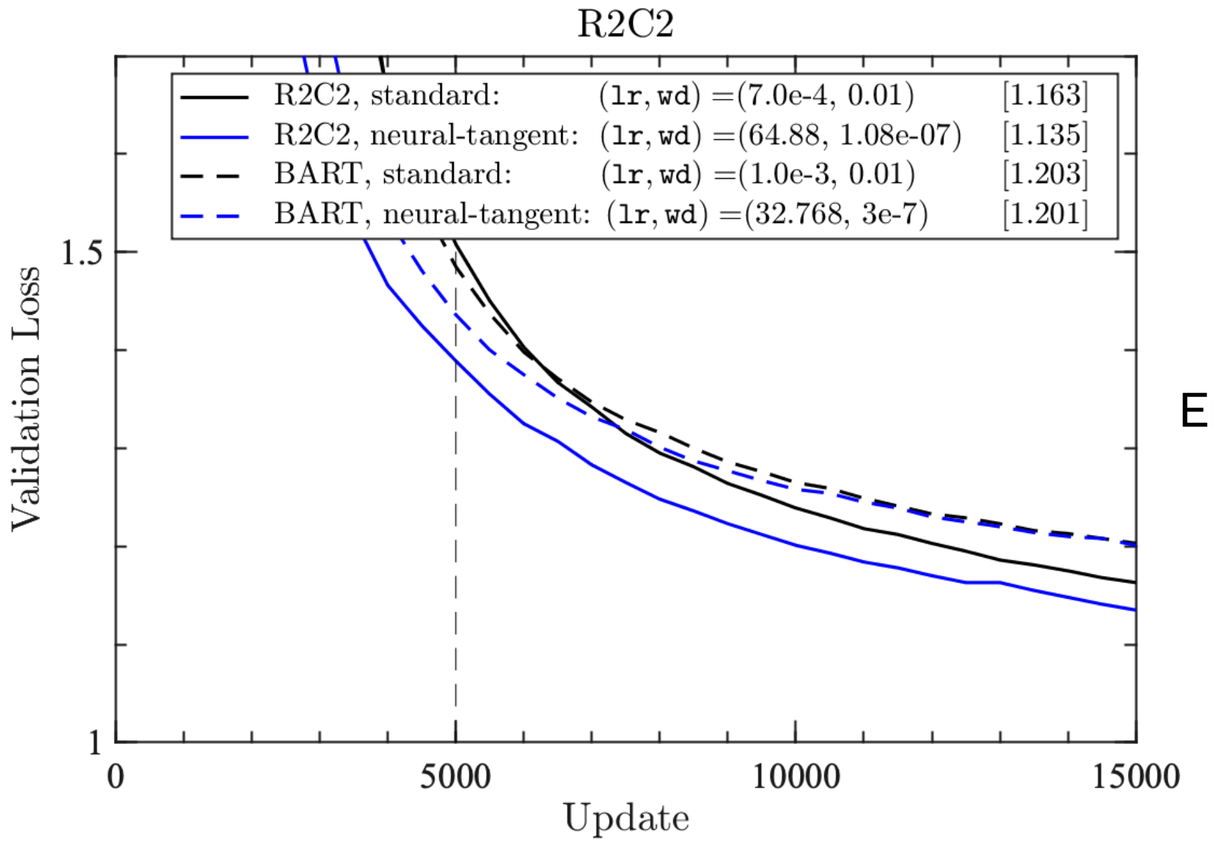
Does it work?



It seems to work well here. ☺

One curious observation:
neural-tangent scaling *increases* the learning rate for positional-embedding parameters by ~*1000 relative to the standard uniform scaling, and it makes the training *stabler*.

Does it work?



Encouraging but not conclusive…

Does it work?

lots of encouraging/discouraging anecdata to come — from you