Notes

Idea

Weak recurrence in hidden states or in attention, the hidden state of the previous step is sitting right above you. Meaning in parallel computation of the transformer the example predicting token n is directly under the example predicting token n-1. Downshift matrix by 1. Hadamard, Matmul, conv, what? Lstm research relevant here.

So the “Mary had a little _” example gets to incorporate the “Mary had a __” example.

In recurrence language we consider the prediction of token n as the current example / input, and the prediction of token n-1 as the state of the previous time step.

Will take a look at LSTM architectures and try out some variations of gating / combining/ forgetting / concatenating current input with the representations of the previous time step.

This seems simplest to try incorporating between the transformer block and feedforward block. So inside of the feedforward block before actual feedforward we’ll modify the input to incorporate the previous time step.

Other:

OK some tempering: we already have this kind of weak recurrence in transformers. the attention and hidden state of time t for a given token get shown to all the other tokens at time t+1. So example 3 does get to see example 2's hidden state at the next layer's QKV. The griffin paper calls multihead attention a "temporal mixing block", a function template where RNN can also be put.
Cross attention but using lstm mechanisms

Other:
expand on cross-example:
like cross-example attention: maybe at the very end, after all layers, you just get to see a weighted example of the representations of previous examples. So there's some masking to prevent looking at future tokens, but you get to see the hidden representations of previous token prediction computation. the cross-example attention idea is kind of like weakly recurrent version: you see the intermediate hidden state of the computation for processing previous words.
EDIT: it's more like second order recurrence. so MHA is itself a kind of recurrence, or rather relation to previous time steps. at layer t, information is mixed, and at layer t+1 when information

is mixed again each token gets to indirectly see how other tokens attended through this t+1 representation. if you do att@att it's like a second order example mixing
so it doesn't quite work doing att@att. but what about the intermediate representations in the ff?

Other:
There's some other idea of projecting all past examples such that each example can see what all others attended to. att@att does this in cross attention. You need to apply appropriate masking and need to do a few projections - my problem trying to build this is it always looks a lot like self attention again, maybe minus one or two projections. The only difference here would be that we look at the hidden representation state of other examples instead of their token representations (following e.g. QKV projections)

X = T D
W1 = D D
P1 = X.masked@W1 = T D each row is each example + it peeks at past representation space (not token) of examples
OUT = gate * X + 1-gate * P1

This is probably most straightforwardly viewed as a convolution

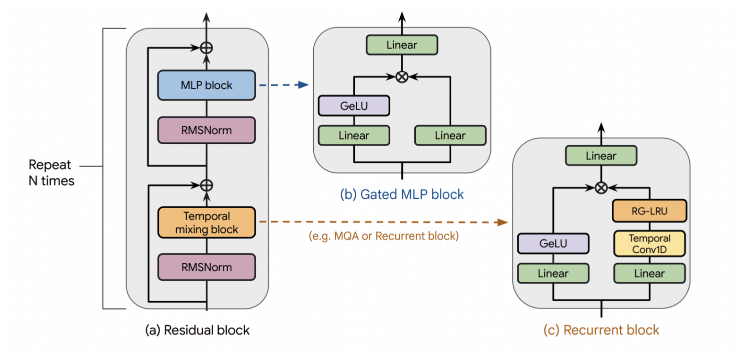
Design questions / variations for SSMs

There are a few dimensions along which we can create variations, and having now looked at some recurrence and SSM literature after the initial experiments it seems like people are basically trying out the different variations in this search space:

- **Where in architecture** is weak recurrence applied? In the attention block, feedforward block? And where inside of it
    - Starting with the beginning of feedforward block
    - TODO review KV caching: am i building something which does not allow for KV caching?
- **Which previous states** are used? One previous state, multiple, all?
- **How are the previous state(s) aggregated**? Convolution, addition, concatenation, linear map, custom kernel, etc. How to effectively concentrate or represent the previous state seems like a key area of research
- By **what operation** is the previous state combined with the current input?
- How are the previous state and current input **gated/weighted** under this combination operation? Is the gating input-dependent?

Interestingly it feels (early now) that you can basically differentiate and describe each of these new SSM/recurrence models by how they answer these questions.

For example, now that I've read the Griffin paper:
Where? In place of self attention



Which states? It seems like they look at the previous 4

Recurrent block Our recurrent block (Figure 2(c)) is similar to the GSS block (Mehta et al., 2022) and the block used by Mamba (Gu and Dao, 2023). We take the input of dimension D and apply two linear layers with output dimension D_{RNN} in parallel, creating two branches. On the first branch, we apply a small separable Conv1D layer, inspired by the Shift-SSM in H3 (Dao et al., 2022b), with a temporal filter dimension of 4. Note that this Conv1D layer is very small, with just $4D_{RNN}$ parameters. We follow the Conv1D layer with our proposed RG-LRU layer (defined below.) On the second branch we apply a GeLU nonlinearity and then merge the branches by element-wise multiplication. We then apply a final linear layer with output dimension D.

Aggregated how? 1d convolution filter

Gated and combined with input how?

$$\begin{aligned} r_t &= \sigma(W_a x_t + b_a), && \textit{recurrence gate} \\ i_t &= \sigma(W_x x_t + b_x), && \textit{input gate} \\ a_t &= a^{c r_t}, \\ h_t &= a_t \odot h_{t-1} + \sqrt{1 - a_t^2} \odot (i_t \odot x_t). \end{aligned}$$

Related works

An incredible amount of work on adding recurrent components into transformer architectures has come out in just the last two weeks (today is 4/16/24). It looks like Griffin and Megalodon and a bunch of others have the same overarching idea of gating "previous time step" and combining with the current time step.

The griffin paper diagram gives a slot to "temporal mixing block", and says that self attention or whatever their special block is goes there. That's a good term, because these are all ways of mixing information across examples / tokens / time steps, so self attention and recurrent and state space methods work here

****

Ok, looked at the hazy research blogs on SSMs to get a brief overview on their genesis. (Didn't realize Chris Re and snorkel were in this space). The "convolution is recurrence is SSM" is a really nice perspective.

The annotated S4 looks good but too deep for now. The Mamba paper also gives a very good overview of what's going on in the space. Albert Gu is a very good explainer, these posts and the mamba paper are extremely clear and concise.

It seems the idea of just pulling in the previous example is a very low-tech version of the SSM and recurrence ideas, but the mechanical questions I have of how to incorporate it are all in line with a lot of the more recent papers.

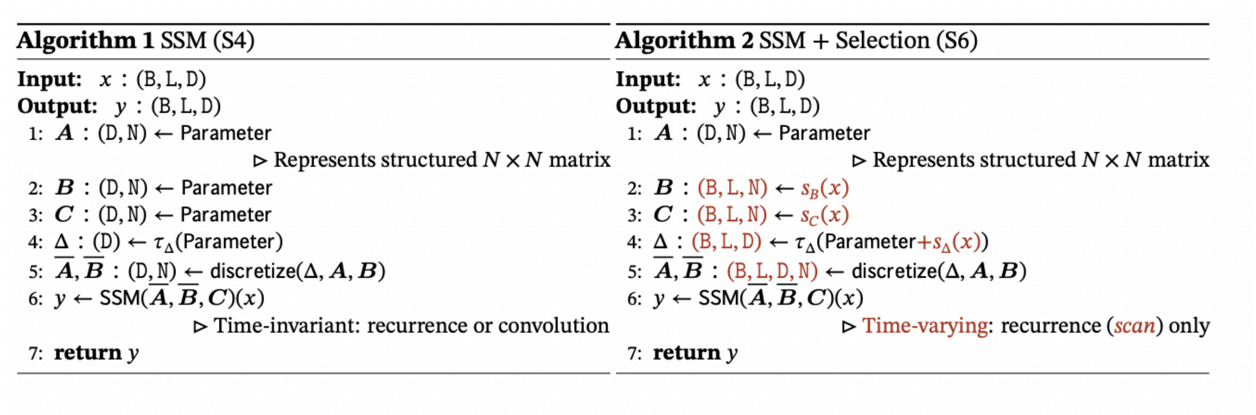
Looking at the Griffin paper blocks confirms this more. Look at the previous 4 steps using convolution and provide a kind of GRU-like gate to combine with input.

There's a lot of reapplication of old LSTM mechanisms.

I'm surprised there's not more discussion or looking-inside of the mechanisms over training. Like, what is the behavior of the learnable parameters, specifically gate parameters, over time?

Griffin
Mamba
S4
GRU
RetNet?
Megaladon

On further reading of Mamba, it looks like the main point is: S4 models are OK but they don't have selection criteria over the past state, so let's make incorporation of the past state input dependent



Algorithm 1 SSM (S4)

```
Input: $x$ : (B, L, D)
Output: $y$ : (B, L, D)
1: $\boldsymbol{A}$ : (D, N) ← Parameter
        ▷ Represents structured $N \times N$ matrix
2: $\boldsymbol{B}$ : (D, N) ← Parameter
3: $\boldsymbol{C}$ : (D, N) ← Parameter
4: $\Delta$ : (D) ← $\tau_\Delta$(Parameter)
5: $\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}$ : (D, N) ← discretize($\Delta, \boldsymbol{A}, \boldsymbol{B}$)
6: $y$ ← SSM($\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C}$)($x$)
        ▷ Time-invariant: recurrence or convolution
7: return $y$
```

Algorithm 2 SSM + Selection (S6)

```
Input: $x$ : (B, L, D)
Output: $y$ : (B, L, D)
1: $\boldsymbol{A}$ : (D, N) ← Parameter
        ▷ Represents structured $N \times N$ matrix
2: $\boldsymbol{B}$ : (B, L, N) ← $s_B(x)$
3: $\boldsymbol{C}$ : (B, L, N) ← $s_C(x)$
4: $\Delta$ : (B, L, D) ← $\tau_\Delta$(Parameter$+s_\Delta(x)$)
5: $\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}$ : (B, L, D, N) ← discretize($\Delta, \boldsymbol{A}, \boldsymbol{B}$)
6: $y$ ← SSM($\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C}$)($x$)
        ▷ Time-varying: recurrence (scan) only
7: return $y$
```

This comes with implementation problems that seem to be a focus of the paper.
[we create projections of input data, inevitably looking more and more like attention…]

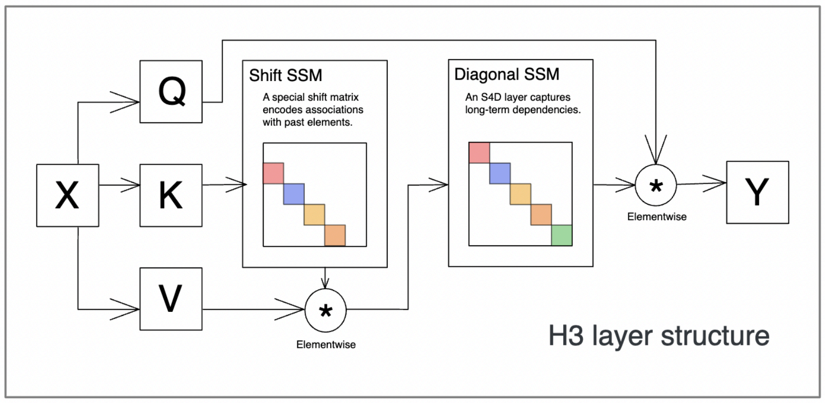
https://arxiv.org/pdf/2404.08819.pdf
"The Illusion of State in State-Space Models"
No idea how relevant this paper is practically (how important the theoretical class of complexity an algorithm handles is), but says that the diagonalized version of the selection gating / input dependence is insufficient to perform certain tasks where a non-diagonalized version can. The fix is making A non-diagonal:

$$h_i = A(x_i)h_{i-1} + Bx_i.$$

https://twitter.com/lambdaviking/status/1780246366075171032
Q: “Wait isn't the discrete A input dependent in Mamba?”
A: “Yes but it’s diagonal and only input-dependent through delta. Turns out this isn’t enough to get greater expressive power”

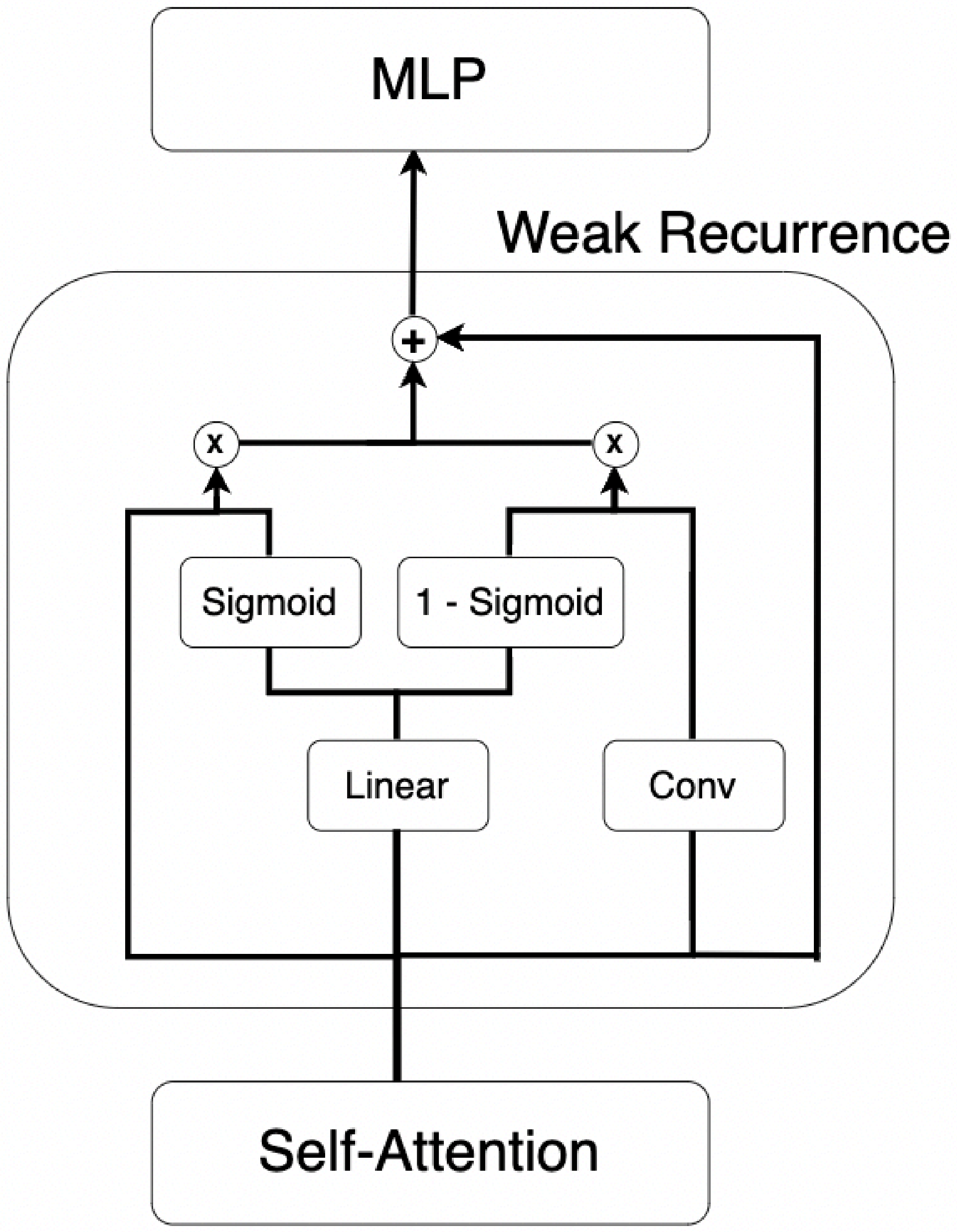
****

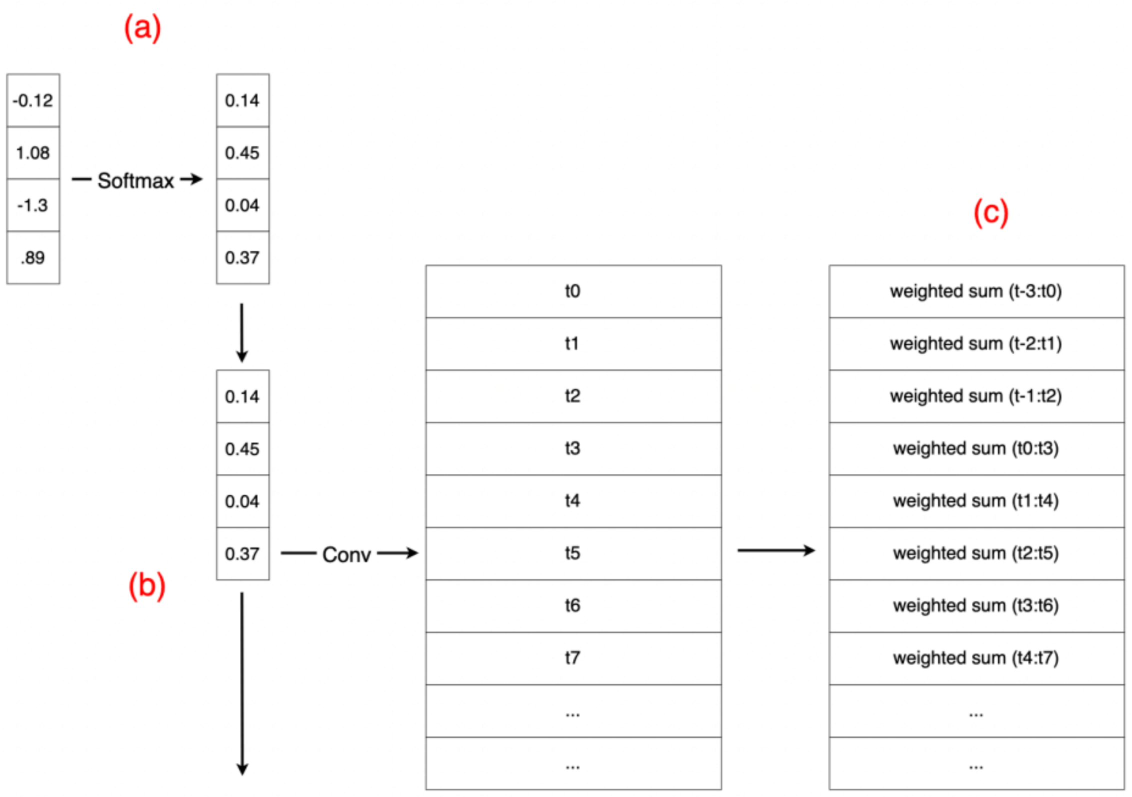
H3 looks like something quite similar to the idea here: includes a shift matrix. I’m not sure of the details and am not getting a concise explanation / implementation. Will look at later

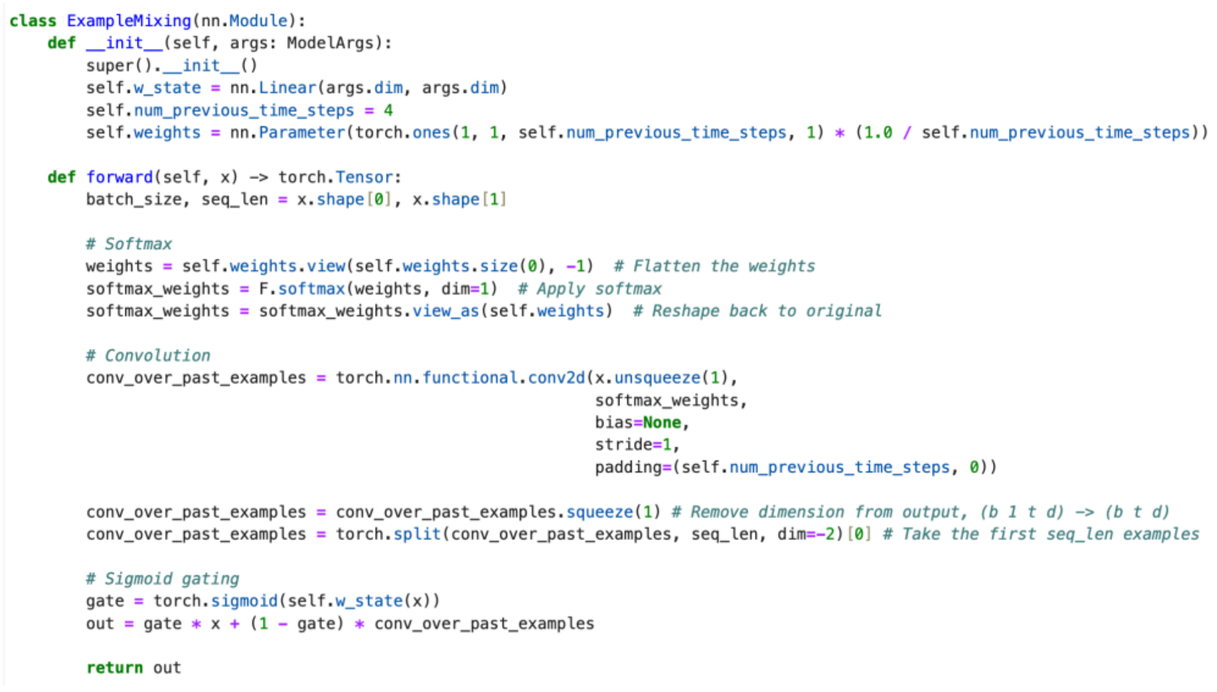

X
Q
K
V
Shift SSM
A special shift matrix encodes associations with past elements.
Diagonal SSM
An S4D layer captures long-term dependencies.
*
Elementwise
*
Elementwise
Y
H3 layer structure

From https://iclr-blogposts.github.io/staging/blog/2022/hippo-to-h3/

My diagrams:


MLP
Weak Recurrence
+
x
x
Sigmoid
1 - Sigmoid
Linear
Conv
Self-Attention





```python
class ExampleMixing(nn.Module):
    def __init__(self, args: ModelArgs):
        super().__init__()
        self.w_state = nn.Linear(args.dim, args.dim)
        self.num_previous_time_steps = 4
        self.weights = nn.Parameter(torch.ones(1, 1, self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))

    def forward(self, x) -> torch.Tensor:
        batch_size, seq_len = x.shape[0], x.shape[1]

        # Softmax
        weights = self.weights.view(self.weights.size(0), -1)  # Flatten the weights
        softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
        softmax_weights = softmax_weights.view_as(self.weights)  # Reshape back to original

        # Convolution
        conv_over_past_examples = torch.nn.functional.conv2d(x.unsqueeze(1),
                                                             softmax_weights,
                                                             bias=None,
                                                             stride=1,
                                                             padding=(self.num_previous_time_steps, 0))

        conv_over_past_examples = conv_over_past_examples.squeeze(1) # Remove dimension from output, (b 1 t d) -> (b t d)
        conv_over_past_examples = torch.split(conv_over_past_examples, seq_len, dim=-2)[0] # Take the first seq_len examples

        # Sigmoid gating
        gate = torch.sigmoid(self.w_state(x))
        out = gate * x + (1 - gate) * conv_over_past_examples

        return out
```

Example Mixing Block

How to create the previous example tensor?

1) We began with inplace operations but this breaks the graph

```
      # previous_state = torch.zeros_like(x)
      # previous_state[:, 1:, :] = x[:, :-1, :]
```

2) Changed it to copy the input data to a tensor of zeros and then downshift the data

```
      # previous_state = torch.cat((torch.zeros(batch_size, 1, args.dim),
x[:,:-1,:]), dim=1)
```

3) Lastly:

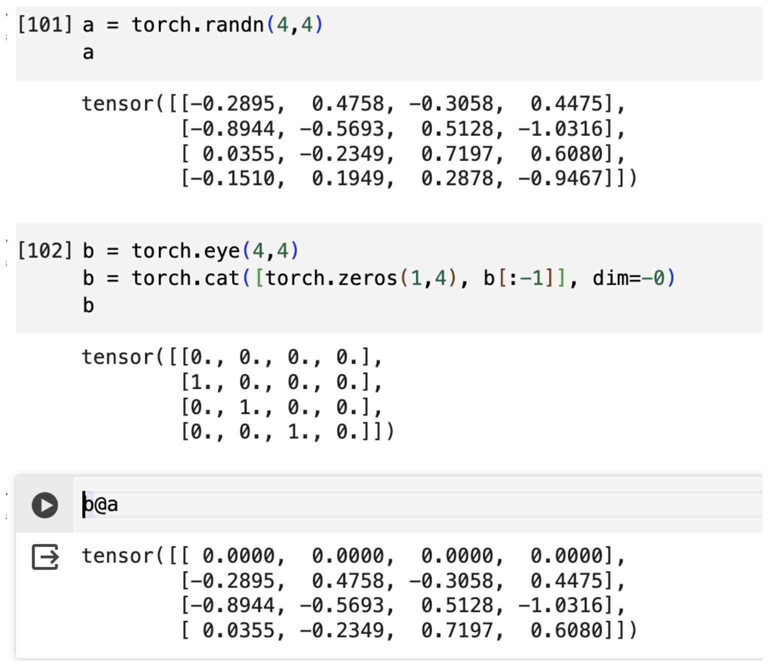
"Wait, instead of breaking the computation graph with inplace or copying the tensor which breaks backprop, it's like:

x@torch.eye(downshifted one)"

So that the previous_state tensor becomes a function of the input rather than a copy of it, allowing backprop to flow through the previous_state tensor.

```
      downshift_matrix = torch.eye(seq_len, seq_len, device=device)
      downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
      downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
      previous_state = downshift_matrix @ x
```

Toy example:



```
[101] a = torch.randn(4,4)
      a

      tensor([[-0.2895,  0.4758, -0.3058,  0.4475],
              [-0.8944, -0.5693,  0.5128, -1.0316],
              [ 0.0355, -0.2349,  0.7197,  0.6080],
              [-0.1510,  0.1949,  0.2878, -0.9467]])
```

```
[102] b = torch.eye(4,4)
      b = torch.cat([torch.zeros(1,4), b[:-1]], dim=-0)
      b

      tensor([[0., 0., 0., 0.],
              [1., 0., 0., 0.],
              [0., 1., 0., 0.],
              [0., 0., 1., 0.]])
```

```
b@a

tensor([[ 0.0000,  0.0000,  0.0000,  0.0000],
        [-0.2895,  0.4758, -0.3058,  0.4475],
        [-0.8944, -0.5693,  0.5128, -1.0316],
        [ 0.0355, -0.2349,  0.7197,  0.6080]])
```

So there's three versions of creating the downshifted matrix of previous example

- In place ops to create previous example tensor (breaks graph)
- Copy input over to create previous example tensor (works but no backprop to data)
- Create downshift linear transformation (works and backprops back to data)

In practice 1 doesn't work, 2 and 3 work about the same (testing at small scale)

****

I've been calling this a "downshift" matrix, which of course turns out to exist and have the same name: https://en.wikipedia.org/wiki/Shift_matrix. I would guess I'd heard this term long ago? But I don't recall seeing the matrix before, it's just obvious that you can create a shift matrix.

Residual Notes

Important note

My terminology has been backwards:

Given something like x_2 = x_1 + f(x_1),

- X_1 is the identity mapping
- f(x_1) is the computation

f(x_1) is referred to as the residual, not x_1. In the log I've been incorrectly referring to the identity/skip connection as the "residual."

So we should call it: the residual and the identity/skip. From here on (in the notes/sections outside of experiment notes) we'll refer to it as residual and skip.

Inside of the experiment logs, whenever I am saying "residual" I mean "skip/identity." In either case the code is well-documented and referenced in the notes so the code is the ground truth.

Related works

This stanford talk by Hyung Won Chung https://www.youtube.com/watch?v=3gb-ZkVRemQ&t=139s is a good explanation of my intuition for this kind of thing. Residual connections were added to help stabilize training, but in the long term it seems like quite restrictive to enforce that each layer sum one part identity with one part residual. Yes, the model in theory can accommodate this and aprpxoimate this - the model is equivalent ot one with learned skip weights - but it seems more general purpose to allow the model to adjust this relationship directly.

So anything that kind of *looks* hardcoded is probably worth revisiting, like the scaling factor in attention, or residuals. It's not really changing the model in any deep way, it's just allowing the model more degrees of freedom to change itself. A good quote from the resnet paper is "Although both forms should be able to asymptotically approximate the desired functions (as hypothesized), **the ease of learning might be different**."

The other piece of intuition is that learned scalable parameters lead to better efficiency. Model a is multiplying two matrices A@B. If A has a single scalar weight (a*A)@B it allows for the weights of A to focus on variance and expressitivity instead of scale. Factoring out a multiplicative scalar means the numbers can become more easily fine-tuned, instead of the case where values are blowing up in one matrix - you could test and see if all matrices take on the same mean/variance while the multiplicative scalars handle the other part

https://arxiv.org/pdf/1512.03385 kaiming he, resnet

- The initial story is the degradation problem: accuracy of a deep model is saturated at some middle layer and then actually becomes worse through the later layers. Surely the later layers could just learn to become identity functions and pass the accurate representation through, rather than degrading it; a model with performing at a certain accuracy should in theory be no worse if we stack additional layers on top of it. And yet it is.
- The whole idea is to facilitate training, not to change the architecture. Reformulate the function to be learned as a residual function. "Although both forms should be able to asymptotically approximate the desired functions (as hypothesized), the ease of learning might be different."

- The key theory: “if the added layers can be constructed as identity mappings, a deeper model should have training error no greater than its shallower counterpart. The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers. With the residual learning reformulation, if identity mappings are optimal, the solvers may simply drive the weights of the multiple nonlinear layers toward zero to approach identity mappings. In real cases, it is unlikely that identity mappings are optimal, but our reformulation may help to precondition the problem. If the optimal function is closer to an identity mapping than to a zero mapping, it should be easier for the solver to find the perturbations with reference to an identity mapping, than to learn the function as a new one.”

https://arxiv.org/pdf/1505.00387.pdf

Highway networks

- Gate = sigmoid(w_gate(x))
- Y = relu(w_transform(x)) * gate + x * (1-gate)
- Propose initializing the w_gate as negative, but note that it is sigmoid(w_gate) so this is always positive “such that the network is initially biased towards carry [skip] behavior”
- Good analysis of what’s going on that I could try to replicate

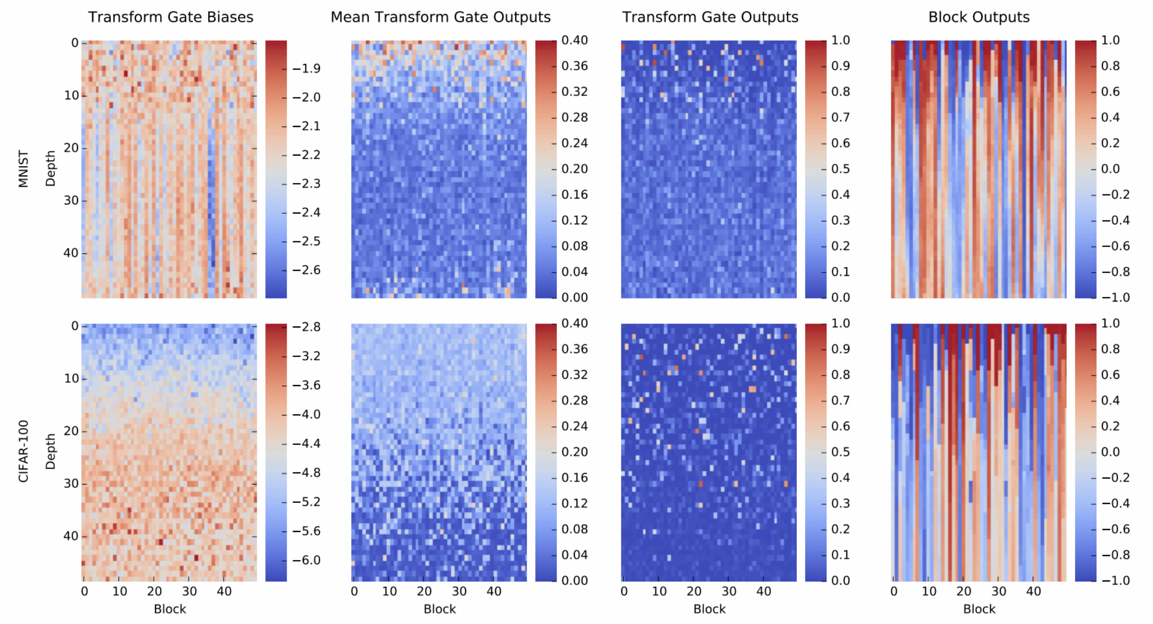


Figure 2. Visualization of certain internals of the blocks in the best 50 hidden layer highway networks trained on MNIST (top row) and CIFAR-100 (bottom row). The first hidden layer is a plain layer which changes the dimensionality of the representation to 50. Each of the 49 highway layers (y-axis) consists of 50 blocks (x-axis). The first column shows the transform gate biases, which were initialized to -2 and -4 respectively. In the second column the mean output of the transform gate over 10,000 training examples is depicted. The third and forth columns show the output of the transform gates and the block outputs for a single random training sample.

-

https://arxiv.org/pdf/1603.05027.pdf

Identity Mappings in Deep Residual Networks by He

- They actually test out a variety of gating mechanisms and scaling options and find that original fixed 1 * skip is the best:

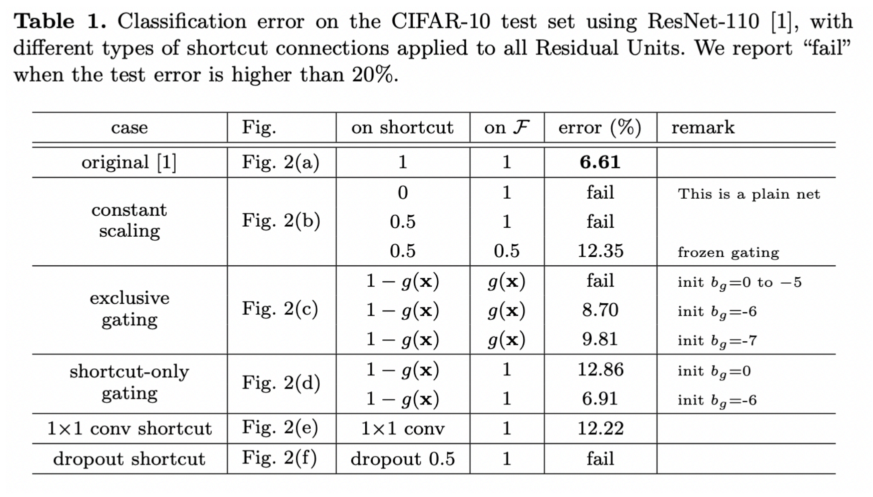
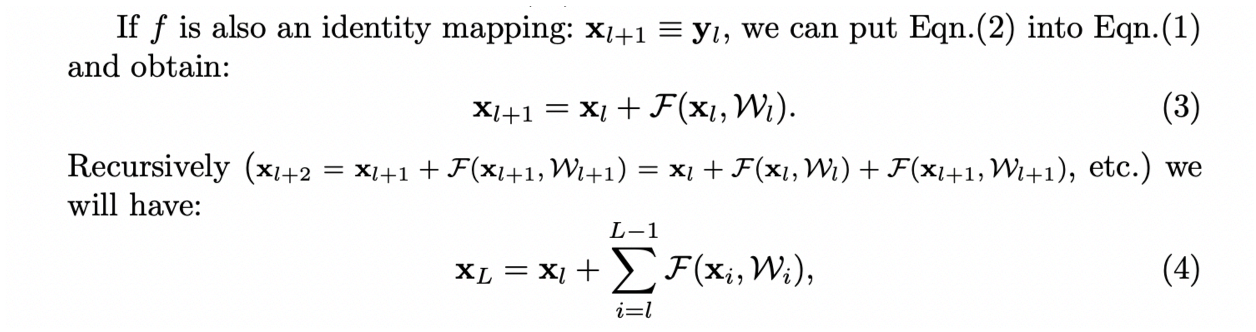


Table 1. Classification error on the CIFAR-10 test set using ResNet-110 [1], with different types of shortcut connections applied to all Residual Units. We report "fail" when the test error is higher than 20%.

case	Fig.	on shortcut	on $\mathcal{F}$	error (%)	remark
original [1]	Fig. 2(a)	1	1	**6.61**	
constant scaling	Fig. 2(b)	0	1	fail	This is a plain net
		0.5	1	fail	
		0.5	0.5	12.35	frozen gating
exclusive gating	Fig. 2(c)	$1-g(\mathbf{x})$	$g(\mathbf{x})$	fail	init b_g=0 to -5
		$1-g(\mathbf{x})$	$g(\mathbf{x})$	8.70	init b_g=-6
		$1-g(\mathbf{x})$	$g(\mathbf{x})$	9.81	init b_g=-7
shortcut-only gating	Fig. 2(d)	$1-g(\mathbf{x})$	1	12.86	init b_g=0
		$1-g(\mathbf{x})$	1	6.91	init b_g=-6
1×1 conv shortcut	Fig. 2(e)	1×1 conv	1	12.22	
dropout shortcut	Fig. 2(f)	dropout 0.5	1	fail	

-
- Tested out skip = .5 fixed and found it worse
- "...the shortcut connections are the most direct paths for the information to propagate. Multiplicative manipulations (scaling, gating, 1×1 convolutions, and dropout) on the shortcuts can hamper information propagation and lead to optimization problems."
- Note that these are all fixed instead of parameterized scalars



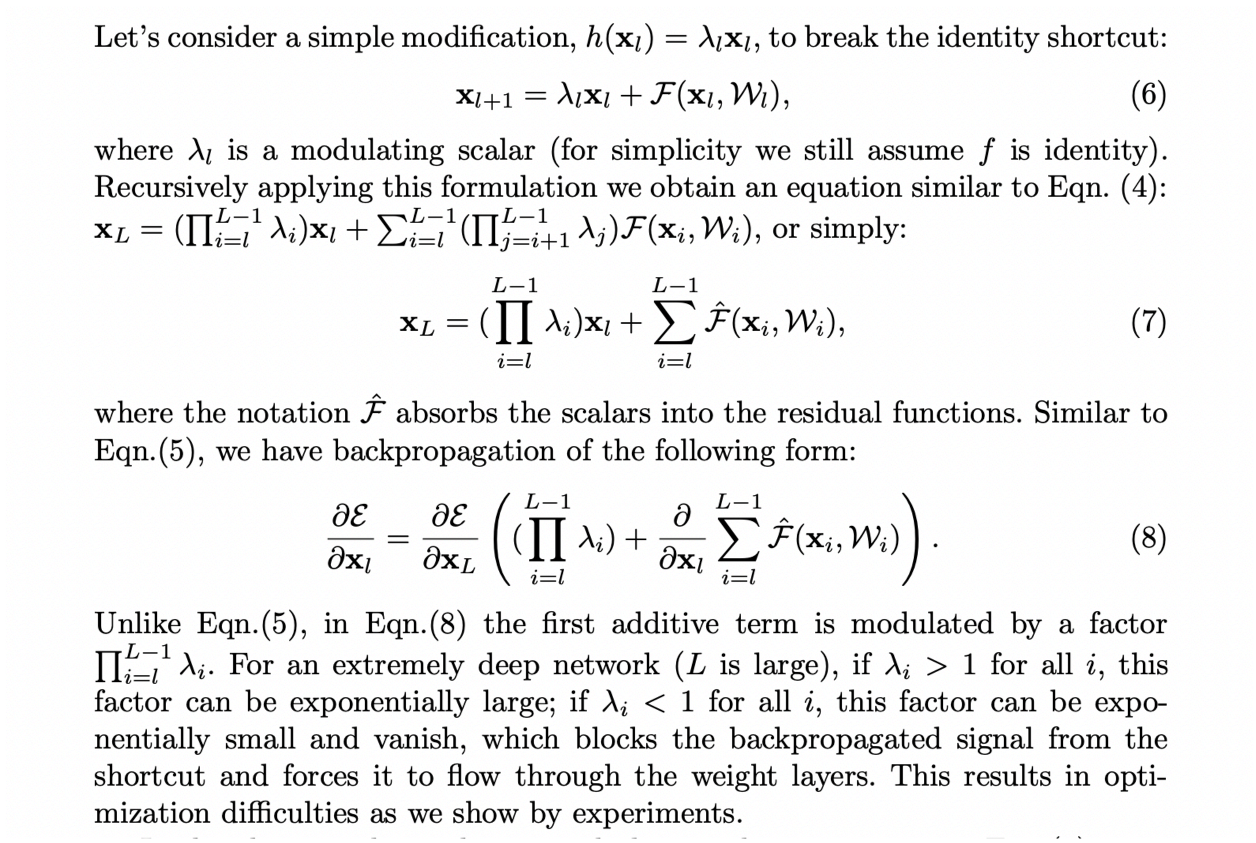
If f is also an identity mapping: $\mathbf{x}_{l+1} \equiv \mathbf{y}_l$, we can put Eqn.(2) into Eqn.(1) and obtain:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l). \tag{3}$$

Recursively $(\mathbf{x}_{l+2} = \mathbf{x}_{l+1} + \mathcal{F}(\mathbf{x}_{l+1}, \mathcal{W}_{l+1}) = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l) + \mathcal{F}(\mathbf{x}_{l+1}, \mathcal{W}_{l+1})$, etc.) we will have:

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}_i, \mathcal{W}_i), \tag{4}$$

-



Let's consider a simple modification, $h(\mathbf{x}_l) = \lambda_l \mathbf{x}_l$, to break the identity shortcut:

$$\mathbf{x}_{l+1} = \lambda_l \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l), \tag{6}$$

where λ_l is a modulating scalar (for simplicity we still assume f is identity). Recursively applying this formulation we obtain an equation similar to Eqn. (4): $\mathbf{x}_L = (\prod_{i=l}^{L-1} \lambda_i)\mathbf{x}_l + \sum_{i=l}^{L-1} (\prod_{j=i+1}^{L-1} \lambda_j)\mathcal{F}(\mathbf{x}_i, \mathcal{W}_i)$, or simply:

$$\mathbf{x}_L = (\prod_{i=l}^{L-1} \lambda_i)\mathbf{x}_l + \sum_{i=l}^{L-1} \hat{\mathcal{F}}(\mathbf{x}_i, \mathcal{W}_i), \tag{7}$$

where the notation $\hat{\mathcal{F}}$ absorbs the scalars into the residual functions. Similar to Eqn.(5), we have backpropagation of the following form:

$$\frac{\partial \mathcal{E}}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L} \left((\prod_{i=l}^{L-1} \lambda_i) + \frac{\partial}{\partial \mathbf{x}_l} \sum_{i=l}^{L-1} \hat{\mathcal{F}}(\mathbf{x}_i, \mathcal{W}_i) \right). \tag{8}$$

Unlike Eqn.(5), in Eqn.(8) the first additive term is modulated by a factor $\prod_{i=l}^{L-1} \lambda_i$. For an extremely deep network (L is large), if $\lambda_i > 1$ for all i, this factor can be exponentially large; if $\lambda_i < 1$ for all i, this factor can be exponentially small and vanish, which blocks the backpropagated signal from the shortcut and forces it to flow through the weight layers. This results in optimization difficulties as we show by experiments.

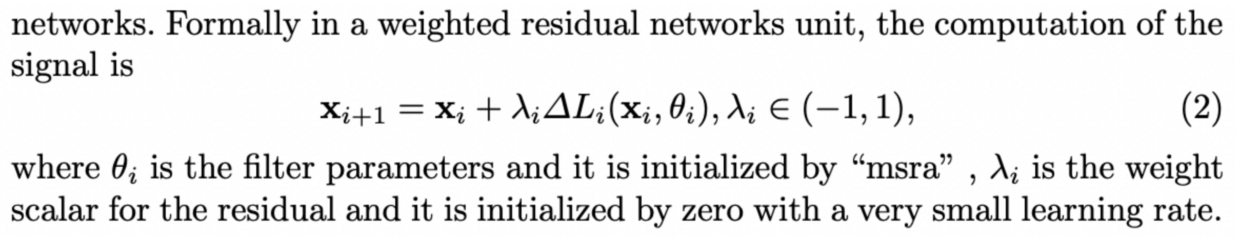
-
    - Where F is the layer residual and x_l is a layer input
    - The point is that depending on lambda's initialization the skip connection can explode or vanish. The bad result is that models fail to train.

https://arxiv.org/pdf/2003.04887 rezero

- Add a learnable weight to the residual that is initialized at zero
- Mostly building on the theory of dynamical isometry: the singular values of the Jacobian are close to one - that this is key for trainability.
-

https://arxiv.org/pdf/1605.08831.pdf

- Basically the same as ReZero but before it? Not cited in ReZero.
- Scalar weight, but not for the skip, for the residual.
-



networks. Formally in a weighted residual networks unit, the computation of the signal is

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i \Delta L_i(\mathbf{x}_i, \theta_i), \lambda_i \in (-1, 1), \tag{2}$$

where θ_i is the filter parameters and it is initialized by "msra" , λ_i is the weight scalar for the residual and it is initialized by zero with a very small learning rate.

- Mostly interested in showing that it performs slightly better and that it allows the training of much deeper (1000s of layers) networks

OK back to negative skip weights. Something bothering me is that the activation functions are not symmetric around zero. So doesn't the sign actually matter? And aren't the outputs of a layer almost always more positive, therefore a negative skip weight means we're subtracting representation? So it seems like it is significant that the skip weights go negative? However in a FF and glu variants there is always one last matmul around the activation function, like (W2(RELU(W1(x))).

- "...For a residual unit, a central question is how to combine the residual signal and the highway signal, where element-wise addition was proposed in [3]. A natural idea is to perform addition after ReLU activation. However, this leads to a nonnegative output from residual branch, which limits the representative ability of the residual unit meaning that it can only enhance the highway signal…."

https://conferences.computer.org/iccst/pdfs/ICCST2020-2x2ntQfxuA3yvTHh2xbxHC/813800a192/813800a192.pdf
- Some discussion of weighted skip connections, but bounded between 0 and 1
- No analysis of what these skip connections do

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8472970/
- Somewhat technical gateing function on the skip connection

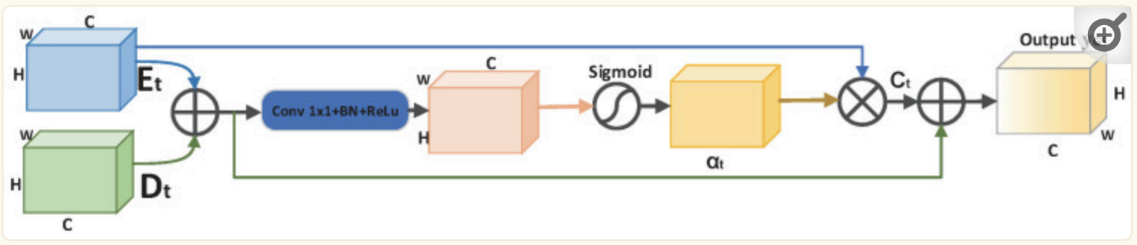


Figure 2

Structure of the gated skip-connection.

-

https://arxiv.org/pdf/2303.13563.pdf
- Spiking neural networks (intersting approach)
- Basically parameterize the position, type, and number of skip connections over a network. This is a large permutation search space, so bayesian optimization to tune them like hyperparameters.
- No analysis of the skip values
- Helps their network, they generalize to saying that more connections help. The work seems confined to spiking neural networks

Finally some background:

https://aclanthology.org/2020.coling-main.320.pdf

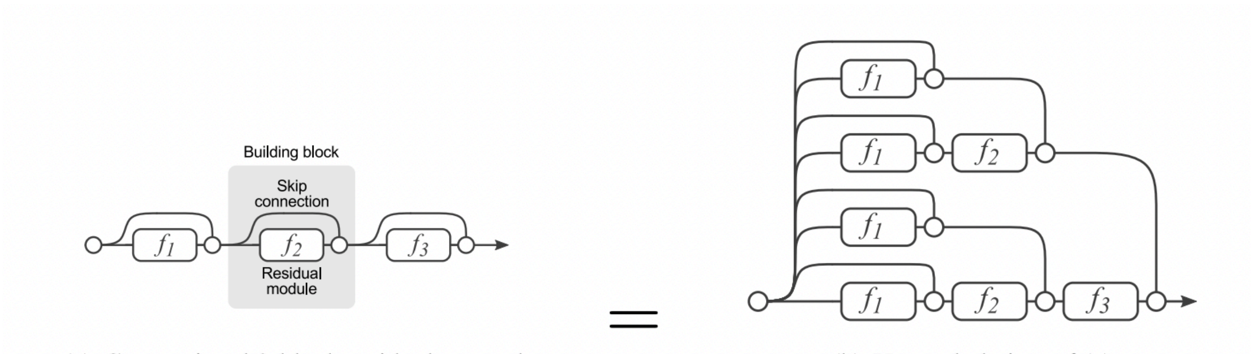
- "...it is intuitive that the modulating factor λ may not always be one, especially when F is not well trained and not sufficient in representation ability. In addition, the intuition is supported by Srivastava et al. (2015) who found that if using a gate mechanism to determine the balance between the two components, the skip component wins in almost all situations. However, in the study of He et al. (2016b)2 , which discussed the combination of batch normalization, learned λ, and convolutional neural networks, they found empirically that the best performance is achieved when λ is fixed as one, suggesting the shortcut and the transformed input should be of equal importance."
- The work is reevaluating where and how to weight the skip connection given the introduction of layernorm
- They mostly focus on position, and set the weight of the skip connection fixed at 2 or 3, actually finding that a higher weight is better?
- At the end of 4.3, they discuss "what if we set the skip weight to be learnable" and try scalar/vector learnable weight, but "However, the findings are not in favor of this approach over the fixed scalar version"
- On the connection problem Highway network (Srivastava et al., 2015) built a highway connection from the input to the output, similar to the skip connection. However, a transform gate was proposed to control the balance of the input x and the transformed input F(x, W), instead of using identity mapping to combine the input and the transformed input. He et al. (2016b) designed five types of skip connections and discussed the possible skip connections in detail. Based on their theory and experiments, they argued that it is likely for a model to perform the best when the skip connection is directly combined into the output without any additional scaling such that the skip and the transformed input have equal contribution to the output. The reason is that with scaling, the gradient of the skip suffers from the gradient exploding or vanishing problem, which hinders the deep neural network from efficient optimization.
-
- …Following Srivastava et al. (2015), we mainly experiment with λ > 1. (lambda is identity/skip scalar)

-

https://aclanthology.org/C16-1020.pdf

- Thorough look at skip connections with LSTMs
- Nothing too relevant that I can see

https://arxiv.org/pdf/1605.06431

- Residual Networks Behave Like Ensembles of Relatively Shallow Networks
- 

(a) Conventional 3-block residual network (b) Unraveled view of (a)
- Residual networks behave like ensembles: if you delete or remove one or multiple residual blocks but leave the skip intact, the model loss doesn't explode (as it does for non-residual networks (VGG)) but instead smoothly increases.`
- Looking at all paths through the network, gradient is concentrated on shorter paths. In fact, deep paths provide no gradient "...Thus, residual networks do not resolve the vanishing gradient problem by preserving gradient flow throughout the entire depth of the network…"
-

https://arxiv.org/pdf/1710.04773
https://arxiv.org/pdf/1612.07771

Hypothesis

My working hypothesis is that previous studies of this (how much to weigh the skip connection vs the residual, whether to make it learnable, etc.) took place a while ago when training networks was fundamentally more unstable, so all experiments basically led to optimization problems. These questions were taken for granted or ignored because they've been tested, but today with more stable networks, the question needs to be asked again. Today, giving the skip connections learnable weights can have a positive effect.
[Follow up: I'm not sure how much new tech has been added since residual connections: in the original paper they talk about norms. Not sure if Adam or LR scheduling was around?]

Setup

“Standard” Transformer model where we swap out the FeedForward block for the FeedForward variations below. These variations give each token/example some type of combination (concatenation, linear transformation, convolution, etc.) with the n previous examples/tokens. Following this, the data is fed through a normal SwiGLU MLP block.

Notebook:
weak recurrence.ipynb

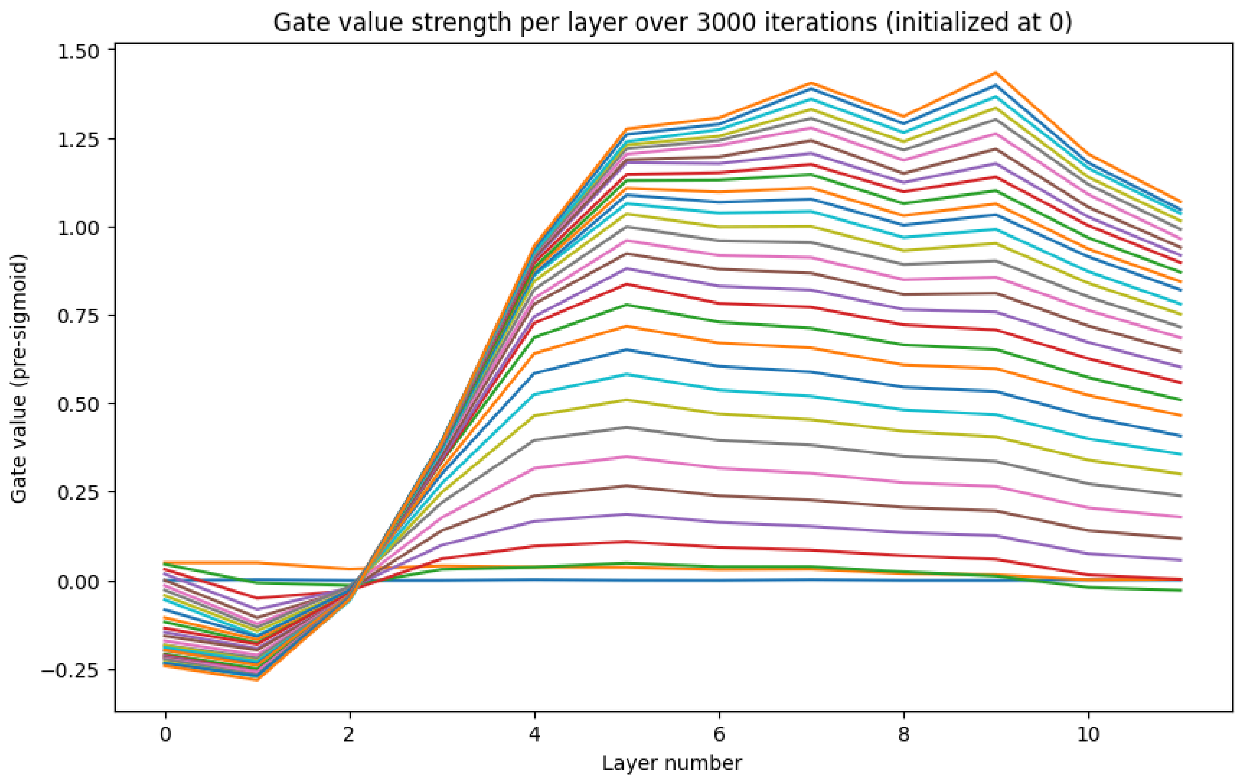
Wandb:
https://wandb.ai/nickcdryan/weak-recurrent-ff

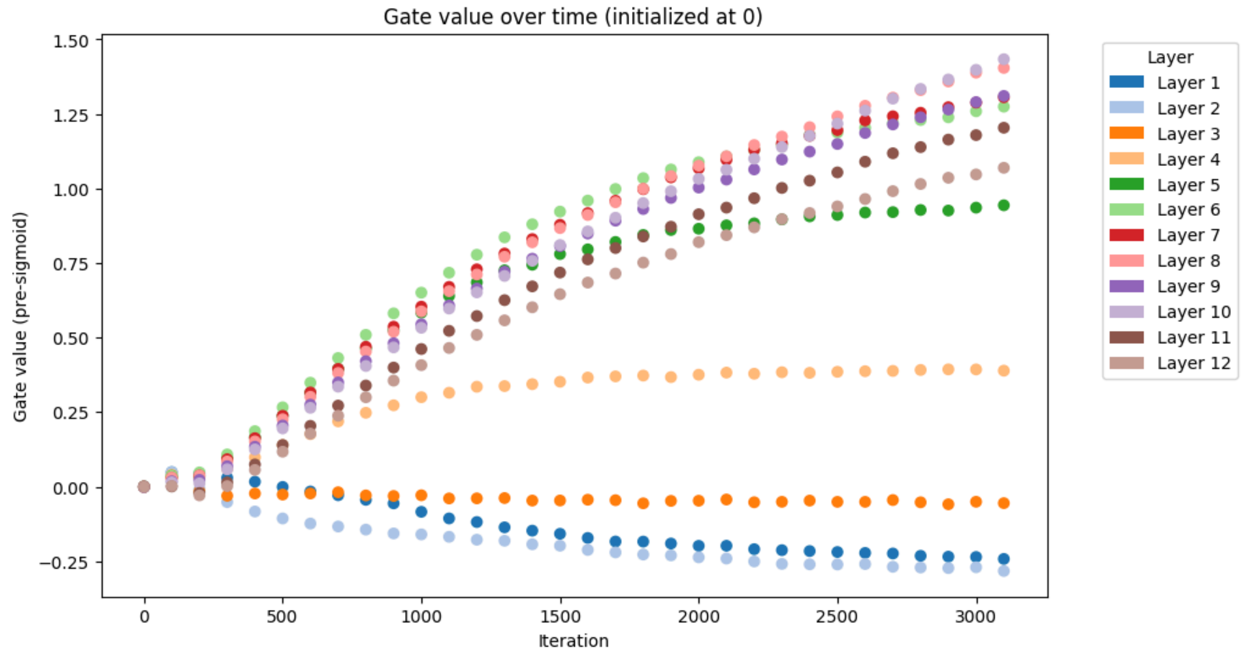
Experiments

Variation 1

```
# VARIATION 1
# Sigmoid gate the previous state

# data = torch.randn(8,512,24)
# # chop off last row, downshift, add zeros as first row. Each row in data
is paired with its previous row
# new = torch.cat((torch.zeros(8,1,24), data[:,:-1,:]), dim=1)
# # original representation gated with previous time step representation
# torch.sigmoid(gate) * data + (1-torch.sigmoid(gate)) * new

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
```

```
        self.gate = nn.Parameter(torch.zeros(1))

    def forward(self, x) -> torch.Tensor:
        batch_size, seq_len = x.shape[0], x.shape[1]

        # copied
        # previous_state = torch.zeros_like(x)
        # previous_state[:, 1:, :] = x[:, :-1, :]

        # in-place
        # previous_state = torch.cat((torch.zeros(batch_size, 1, args.dim),
x[:,:-1,:]), dim=1)

        # backprop
        downshift_matrix = torch.eye(seq_len, seq_len, device=device)
        downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
        downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
        previous_state = downshift_matrix @ x

        gate_value = torch.sigmoid(self.gate)
        x = gate_value * x + (1 - gate_value) * previous_state
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```





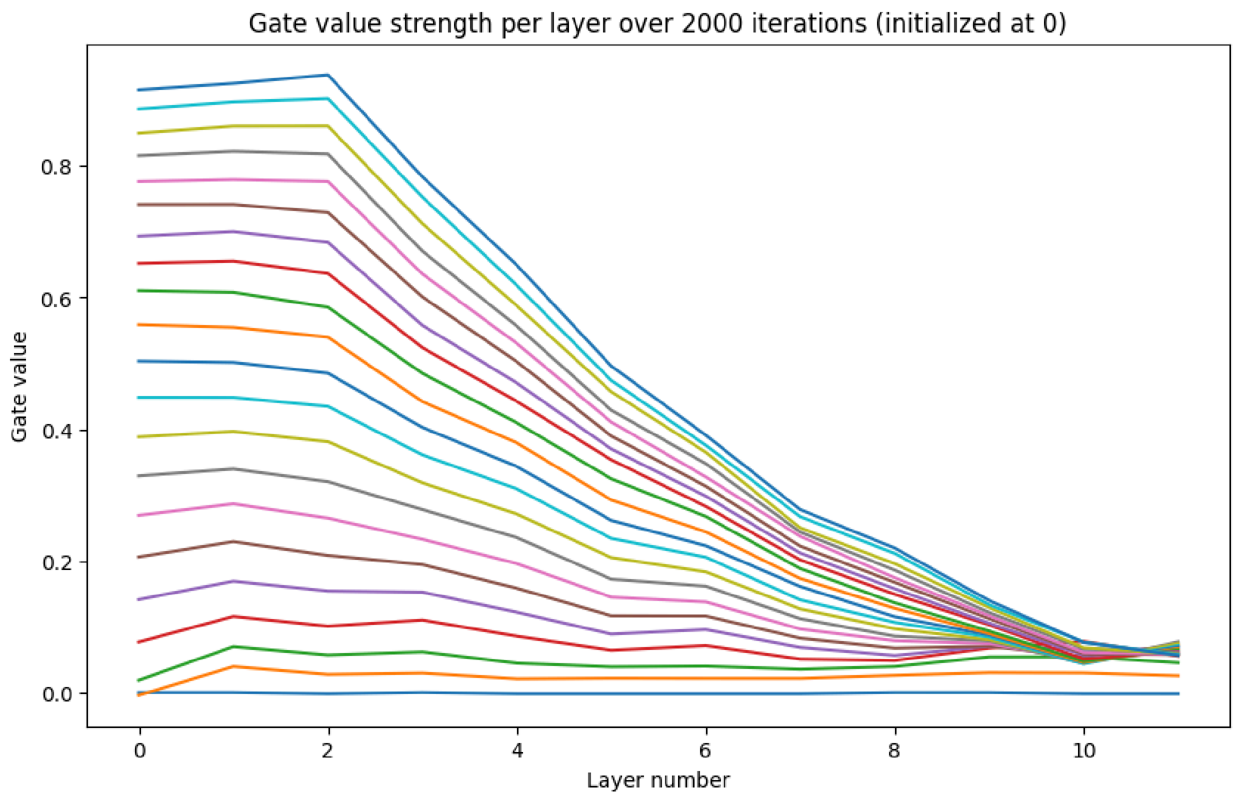
- The relevant code:
    - x = gate_value * x + (1 - gate_value) * previous_state
- The lower layers actually more heavily weight the previous state
- The higher layers more heavily weight the original input

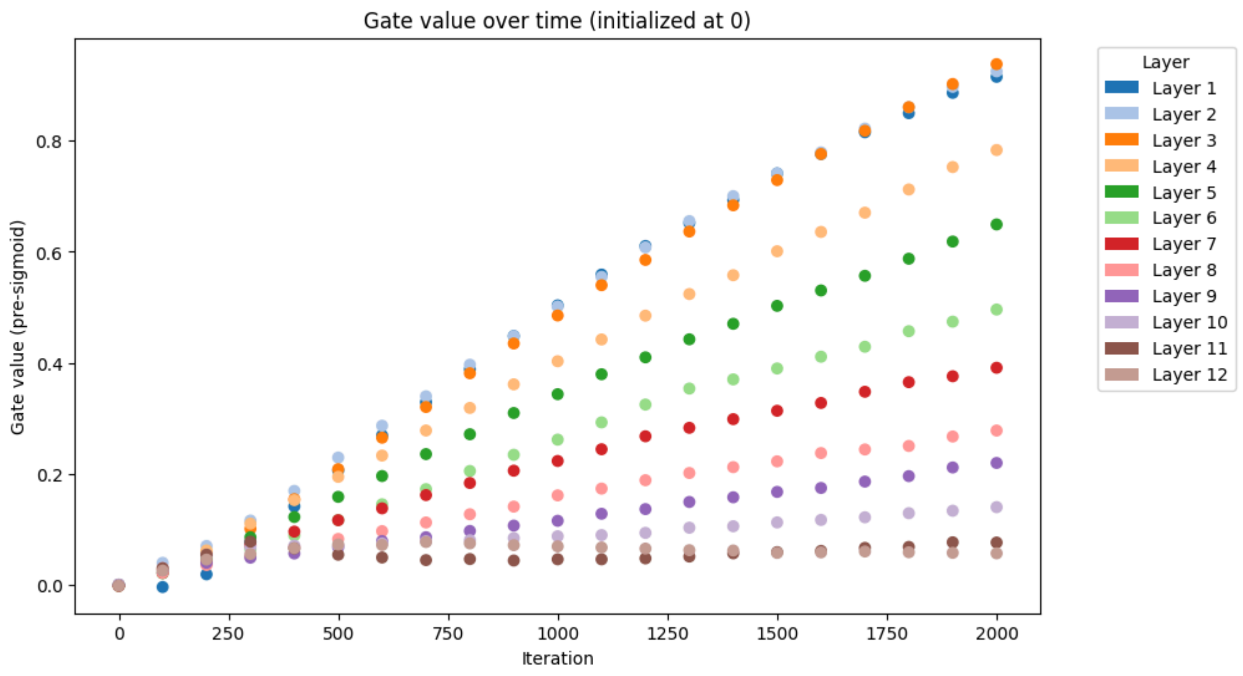
Variation 2

```
# VARIATION 2
# Concat and with previous state and project back to args.dim

# data = torch.randn(8,512,24)
# # chop off last row, downshift, add zeros as first row. Each row in data
is paired with its previous row
# new = torch.cat((torch.zeros(8,1,24), data[:,:-1,:]), dim=1)
# concat = torch.cat([data, new], dim=-1)
# w = torch.randn(48, 24)
# out = concat @ w


class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim * 2, args.dim)

   def forward(self, x) -> torch.Tensor:
       # need to shift matrix down one and add zeros in first row
       previous_state = torch.zeros_like(x)
       previous_state[:, 1:, :] = x[:, :-1, :]
       #previous_state = torch.cat((torch.zeros(batch_size, 1, args.dim),
x[:,:-1,:]), dim=1)
       concat = torch.cat([x, previous_state], dim=-1)
       #print ("concat", concat.shape)
       x = self.w_state(concat)
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Variation 3

```
# VARIATION 3
# Concat, project, and gate with previous state

# data = torch.randn(8,512,24)
# # chop off last row, downshift, add zeros as first row. Each row in data
is paired with its previous row
# new = torch.cat((torch.zeros(8,1,24), data[:,:-1,:]), dim=1)
# concat = torch.cat([data, new], dim=-1)
# w = torch.randn(48, 24)
# out = concat @ w
# torch.sigmoid(gate) * data + (1-torch.sigmoid(gate)) * out


class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim * 2, args.dim)
       self.gate = nn.Parameter(torch.zeros(1))

   def forward(self, x) -> torch.Tensor:
       # need to shift matrix down one and add zeros in first row
       previous_state = torch.zeros_like(x)
       previous_state[:, 1:, :] = x[:, :-1, :]
       #previous_state = torch.cat((torch.zeros(batch_size, 1, args.dim),
x[:,:-1,:]), dim=1)
       concat = torch.cat([x, previous_state], dim=-1)
       x_updated = self.w_state(concat)
       gate_value = torch.sigmoid(self.gate)
       x = gate_value * x + (1 - gate_value) * x_updated
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```





- Relevant line:
  - x = gate_value * x + (1 - gate_value) * x_updated
- All layers are learning to prefer the original input, though early layers much more so

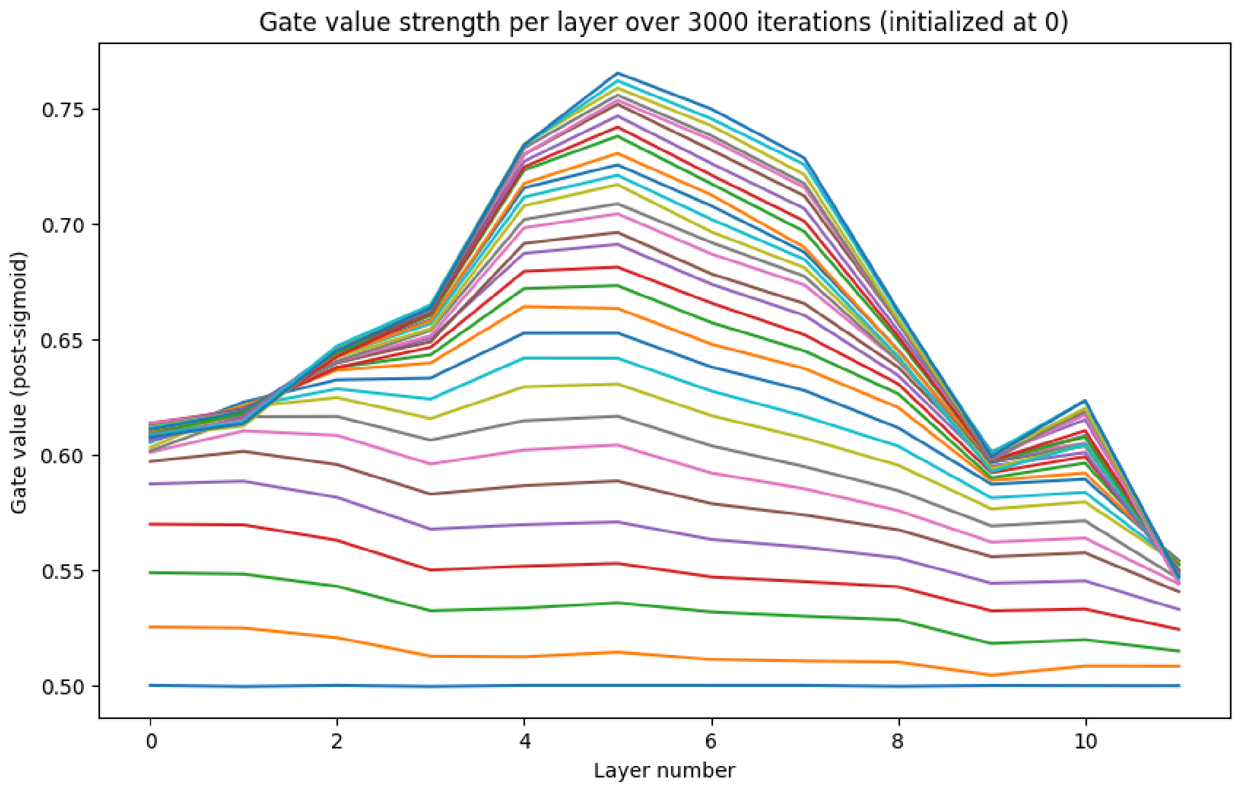
Variation 3.1 (Mamba input-dependent gating)

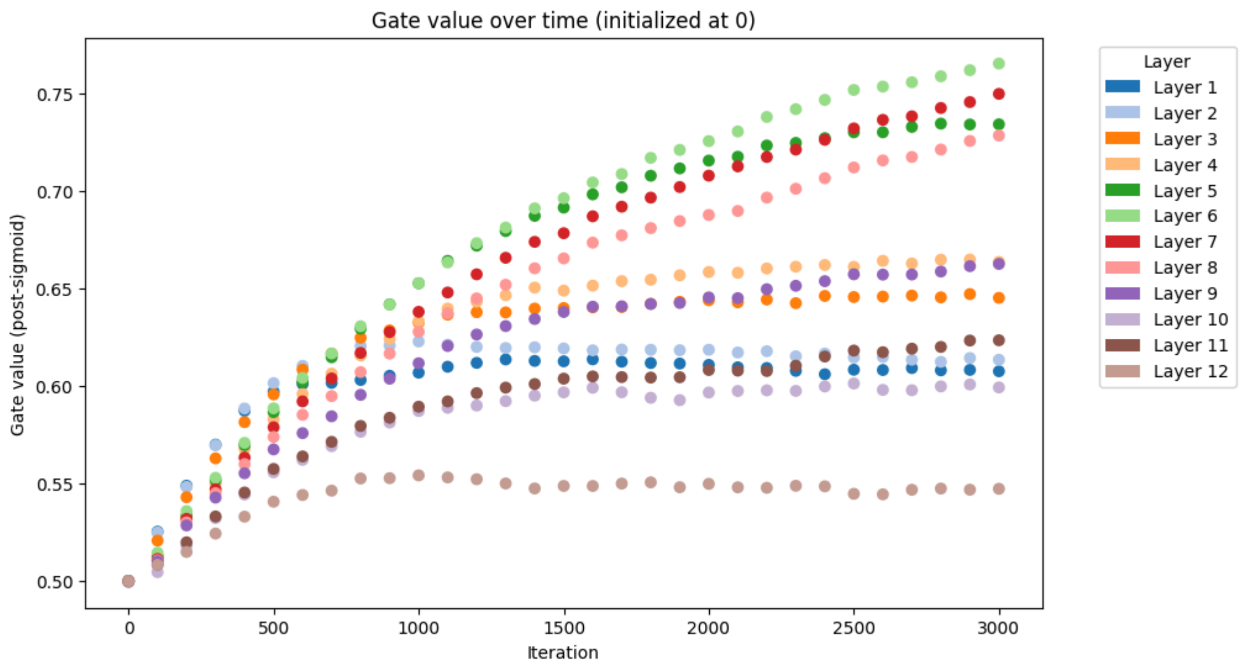
```
# VARIATION 3.1
# Pass input through a matrix and use this matrix as the gate value
# Basically from mamba 3.5.1, making the gating input/data dependent
https://arxiv.org/pdf/2312.00752.pdf


class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       downshift_matrix = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
       downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
       previous_state = downshift_matrix @ x

       # tensor gate
       gate = torch.sigmoid(self.w_state(x))

       x = gate * x + (1 - gate) * previous_state
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Variation 3.2 (Mamba over previous 2 examples)

```
# VARIATION 3.2
# Pass input through a matrix and use this matrix as the gate value
# This gate value controls a sigmoid (scalar) gated combination of
previous and previous_previous
# Basically from mamba 3.5.1, making the gating input/data dependent
https://arxiv.org/pdf/2312.00752.pdf


class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.gate = nn.Parameter(torch.zeros(1))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       downshift_matrix = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
       downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
       previous_state = downshift_matrix @ x

       downshift_matrix_2 = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix_2 = torch.cat([torch.zeros(2, seq_len,
device=device), downshift_matrix_2[:-2]], dim=0)
       downshift_matrix_2 = downshift_matrix_2.repeat(batch_size, 1, 1)
       previous_previous_state = downshift_matrix_2 @ x

       # gated combination of the states
       gate_value = torch.sigmoid(self.gate)
       combined_states = gate_value * previous_state + (1-gate_value) *
previous_previous_state

       # tensor gate
       gate_tensor = torch.sigmoid(self.w_state(x))
```

```
        x = gate_tensor * x + (1 - gate_tensor) * combined_states
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```





- Note that the gate here just controls the balance of previous and previous_previous states. It does not affect the current input's weight, which is controlled with a tensor of gate values

Variation 4 (GRU)

```
# VARIATION 4
# Basically GRU

# data = torch.randn(8,512,24)
# # chop off last row, downshift, add zeros as first row. Each row in data
is paired with its previous row
# old = torch.cat((torch.zeros(8,1,24), data[:,:-1,:]), dim=1)
# concat = torch.cat([data, old], dim=-1)
# wr = torch.randn(48, 24)
# wz = torch.randn(48, 24)
# r = torch.sigmoid(concat @ wr)
# z = torch.sigmoid(concat @ wz)
# w = torch.randn(48, 24)
# h = torch.cat([r * old, data], dim=-1) @ w
# h = torch.nn.functional.silu(h)
# h_out = (1 - z) * old + z * h
# h_out.shape


class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim * 2, args.dim)
       self.wr = nn.Linear(args.dim * 2, args.dim)
       self.wz = nn.Linear(args.dim * 2, args.dim)

   def forward(self, x) -> torch.Tensor:
       # need to shift matrix down one and add zeros in first row
       previous_state = torch.zeros_like(x)
```

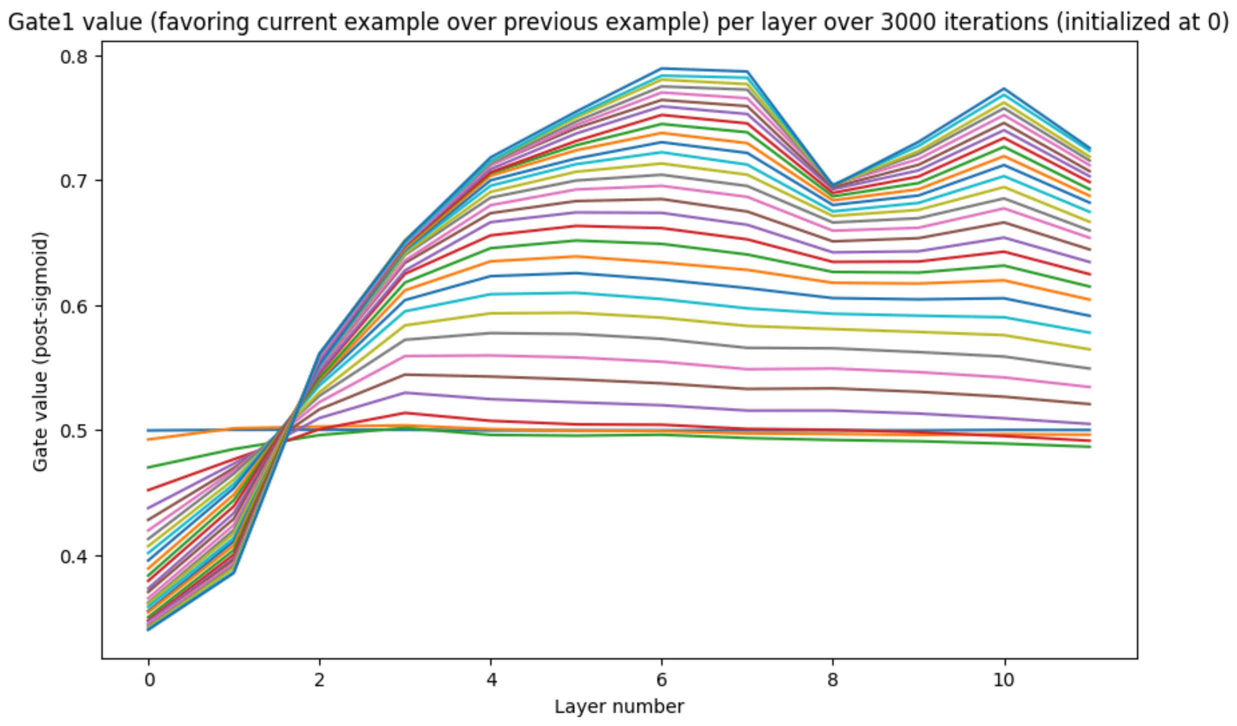
```
        previous_state[:, 1:, :] = x[:, :-1, :]
        #previous_state = torch.cat((torch.zeros(batch_size, 1, args.dim),
x[:,:-1,:]), dim=1)
        concat = torch.cat([x, previous_state], dim=-1)
        r = torch.sigmoid(self.wr(concat))
        z = torch.sigmoid(self.wz(concat))
        h = self.w_state(torch.cat([r * previous_state, x], dim=-1))
        h = torch.nn.functional.silu(h)
        x = (1-z) * previous_state + z * h
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Variation 5

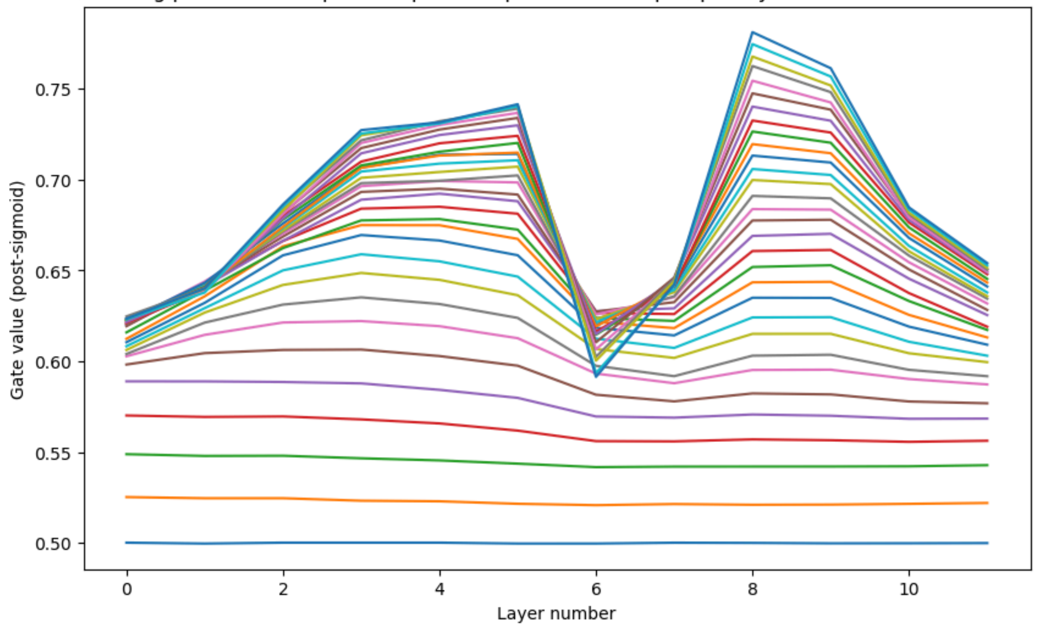
Same as variation 1 but a learned weighted sum of the current example plus previous example plus previous previous example

```
# VARIATION 5
# Gate the previous two states

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.gate1 = nn.Parameter(torch.zeros(1))
       self.gate2 = nn.Parameter(torch.zeros(1))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       # previous state
       downshift_matrix = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
       downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
       previous_state = downshift_matrix @ x
```

```
        # previous previous state
        downshift_matrix_2 = torch.eye(seq_len, seq_len, device=device)
        downshift_matrix_2 = torch.cat([torch.zeros(2, seq_len,
device=device), downshift_matrix_2[:-2]], dim=0)
        downshift_matrix_2 = downshift_matrix_2.repeat(batch_size, 1, 1)
        previous_previous_state = downshift_matrix_2 @ x

        # gate values
        gate1_value = torch.sigmoid(self.gate1)
        gate2_value = torch.sigmoid(self.gate2)
        gate_x = gate1_value
        gate_previous = (1 - gate1_value) * gate2_value
        gate_previous_previous = (1 - gate1_value) * (1 - gate2_value)

        x = (gate_x * x) + (gate_previous * previous_state) +
(gate_previous_previous * previous_previous_state)
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```



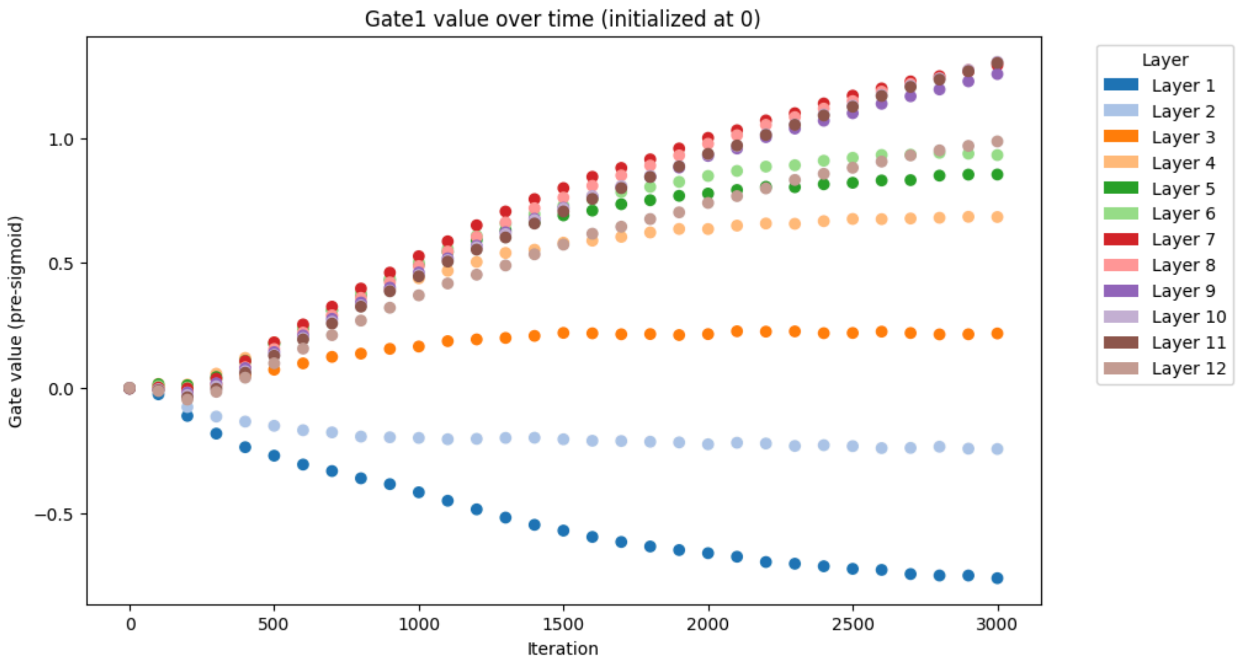
Gate2 value (favoring previous example over previous previous example) per layer over 3000 iterations (initialized at 0)



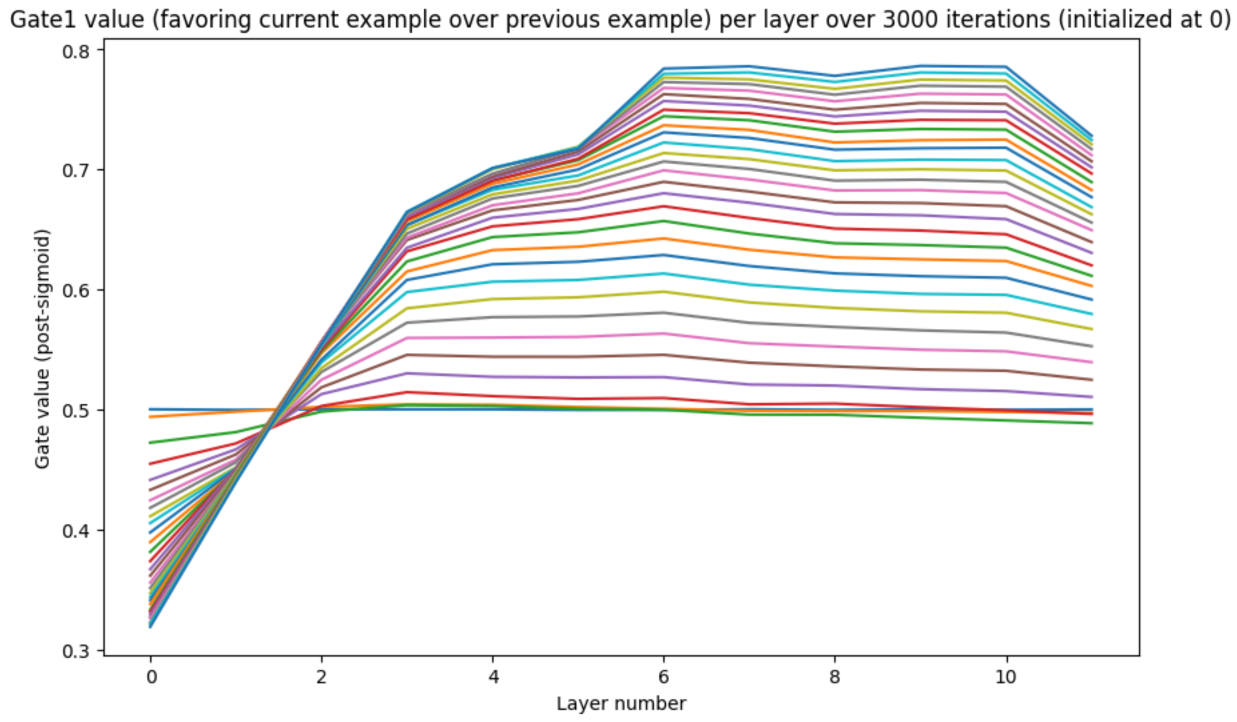
- To be clear these graphs are POST sigmoid where all previous graphs are pre sigmoid. This is because with three values to distribute it's easier to just see the probabilities
    - So gate1 is the weight given to the current example. Gate2 is the weight given to the previous example * (1 - gate1), and the previous previous example gets weight (1 - gate2) * (1 - gate1)`
- For early layers, the current example is weighed less than previous examples, and between previous examples, the previous is weighed more heavily than the previous previous example
- For later layers, the current example is weighed more heavily, and the remaining weight favors the previous example over the previous previous example
- Strange dynamics at layer 5, 6 and 8, where we change how much we care about previous example vs previous previous example

Variation 6

```
# VARIATION 6
# Combine the previous two states and gate with current input

class FeedForward(nn.Module):
    # SwiGLU
    def __init__(self, args: ModelArgs):
        super().__init__()

        self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
        self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.gate1 = nn.Parameter(torch.zeros(1))
        self.gate2 = nn.Parameter(torch.zeros(1))

    def forward(self, x) -> torch.Tensor:
        batch_size, seq_len = x.shape[0], x.shape[1]

        # previous state
        downshift_matrix = torch.eye(seq_len, seq_len, device=device)
        downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
        downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
        previous_state = downshift_matrix @ x

        # previous previous state
        downshift_matrix_2 = torch.eye(seq_len, seq_len, device=device)
        downshift_matrix_2 = torch.cat([torch.zeros(2, seq_len,
device=device), downshift_matrix_2[:-2]], dim=0)
        downshift_matrix_2 = downshift_matrix_2.repeat(batch_size, 1, 1)
        previous_previous_state = downshift_matrix_2 @ x

        # gated combination of the states
        gate2_value = torch.sigmoid(self.gate2)
        combined_states = gate2_value * previous_state + (1-gate2_value) *
previous_previous_state

        gate1_value = torch.sigmoid(self.gate1)
        x = gate1_value * x + (1-gate1_value) * combined_states
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

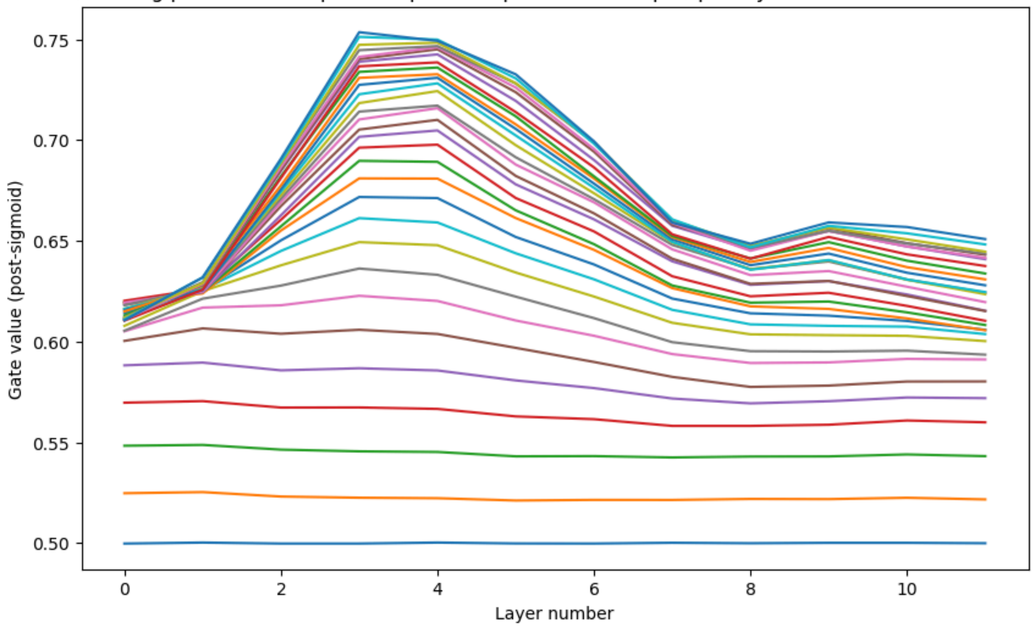


Gate1 value over time (initialized at 0)
Gate value (pre-sigmoid)
Iteration
Layer
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer 6
Layer 7
Layer 8
Layer 9
Layer 10
Layer 11
Layer 12



Gate1 value (favoring current example over previous example) per layer over 3000 iterations (initialized at 0)
Gate value (post-sigmoid)
Layer number

Gate2 value (favoring previous example over previous previous example) per layer over 3000 iterations (initialized at 0)



- First gate looks the same as usual
- The second gate is strange. Why favor previous_state more in the early layers than in the later layers? I have no obvious explanation, and would also like to train longer/deeper to watch the gates evolve
- Just eyeballing the proportion for each t, t-1, t-2:
  - Layer 0
    - T : .30
    - T-1: .42
    - T-2: .28
  - Layer 2
    - T : .50
    - T-1: .34
    - T-2: .16
  - Layer 4
    - T : .70
    - T-1: .22
    - T-2: .08
  - Layer 8
    - T : .8
    - T-1: .13
    - T-2: .07
  - Layer 11
    - T : .75
    - T-1: .16
    - T-2: .09

Variation 6.1

```
# VARIATION 6.1
# Combine the previous two states naively and gate with current input

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.gate1 = nn.Parameter(torch.zeros(1))
       self.gate2 = nn.Parameter(torch.zeros(1))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       # previous state
       downshift_matrix = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
       downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
       previous_state = downshift_matrix @ x

       # previous previous state
       downshift_matrix_2 = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix_2 = torch.cat([torch.zeros(2, seq_len,
device=device), downshift_matrix_2[:-2]], dim=0)
       downshift_matrix_2 = downshift_matrix_2.repeat(batch_size, 1, 1)
       previous_previous_state = downshift_matrix_2 @ x


       # just add the states
       combined_states = (previous_state + previous_previous_state) / 2.0

       gate1_value = torch.sigmoid(self.gate1)
       x = gate1_value * x + (1-gate1_value) * combined_states
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Slightly worse performance than 6, suggesting gating rather than naive addition between the previous_state and Previous_previous_state helps.

Variation 7 (Griffin-like)

Griffin style gating, but lambda is a single scalar.

Also, we're not caching previous state, but just injecting conv_over_past_examples as the h_prev.

So essentially we're just trying out the LR-GRU gating mechanism

We can build out the full one but I just want to try the gating mechanism inside the current framework. Only applies to 1 previous state.

Shows norms commented out

```
# VARIATION 7
# Griffin-like
# We're not doing a tensor/diagonal lambda/a matrix
# Just a scalar

# Lambda init:

#
https://github.com/google-deepmind/recurrentgemma/blob/e4939f9b7edf8baa1d5
12fb86bfc2e206044d66b/recurrentgemma/torch/layers.py#L193
# "We initialize Λ such that a**c is uniformly distributed between 0.9 and
0.999 at the start of training, similar to Orvieto et al. (2023b)."

# #.99 mimicing if we sampled uniformly between .9 and .999
# a = torch.tensor(.99**2)
# # ?
# a.log_().mul_(0.5)
# # inverse softplus (inplace operations work left to right)
# a.neg_().exp_().sub_(1.0).log_()
# a = -4.59

class FeedForward(nn.Module):
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.wr = nn.Linear(args.dim, args.dim, bias=True)
       self.wi = nn.Linear(args.dim, args.dim, bias=True)
       self.lambda_p = nn.Parameter(torch.tensor(-4.59))
       #self.griffin_norm = RMSNorm(args.dim, eps=args.norm_eps) ######

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       downshift_matrix = torch.eye(seq_len, seq_len, device=device)
       downshift_matrix = torch.cat([torch.zeros(1, seq_len,
device=device), downshift_matrix[:-1]], dim=0)
       downshift_matrix = downshift_matrix.repeat(batch_size, 1, 1)
       previous_state = downshift_matrix @ x

       r = self.wr(x)
       i = self.wi(x)

       # stable impelmentation, Appendix A
       log_a = -8.0 * torch.nn.functional.softplus(self.lambda_p) * r
       a = torch.exp(log_a)
       a_squared = torch.exp(log_a * 2.)
       x = a * previous_state + ( (1 - a_squared) * i * x )
       #x = self.griffin_norm(x) ######

       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

First run is actually better than all variation up to now, but

```
step 0 :training loss: 11.042474746704102
step 100 :training loss: 2.3072938919067383
step 200 :training loss: 2.256641387939453
step 300 :training loss: 1.8106539249420166
step 400 :training loss: 1.640507459640503
step 500 :training loss: 1.6591455936431885
step 600 :training loss: nan
step 700 :training loss: nan
step 800 :training loss: nan
```

Debugging

1) Let's check the parameter weights specific to griffin to see if anything's amiss
2) Also maybe need to apply a norm post griffin and pre ffn. My first thought is there's a lot of computation in the griffin block, the output of which goes straight into FF. usually there's a norm direclty before attention and directly before FF, but here we take the output of attention, norm it, then do a bunch of griffin computation before sending it straight to FF, so a third norm might be needed post griffin.

```
for i in range(12):
   print (model.layers[i].feed_forward.lambda_p)
Parameter containing:
tensor(-4.5145, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.4586, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.4302, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.3845, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.3741, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.3588, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.3288, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.3174, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.2991, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.2954, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.2991, device='cuda:0', requires_grad=True)
Parameter containing:
tensor(-4.2924, device='cuda:0', requires_grad=True)
```

- Doesn't seem to be the lambda

WR:

```
for i in range(12):
    mean = torch.mean(model.layers[i].feed_forward.wr.weight.detach().cpu())
    var = torch.var(model.layers[i].feed_forward.wr.weight.detach().cpu())
    print (i, "mean", mean)
    print (i, "var", var, "\n")
```

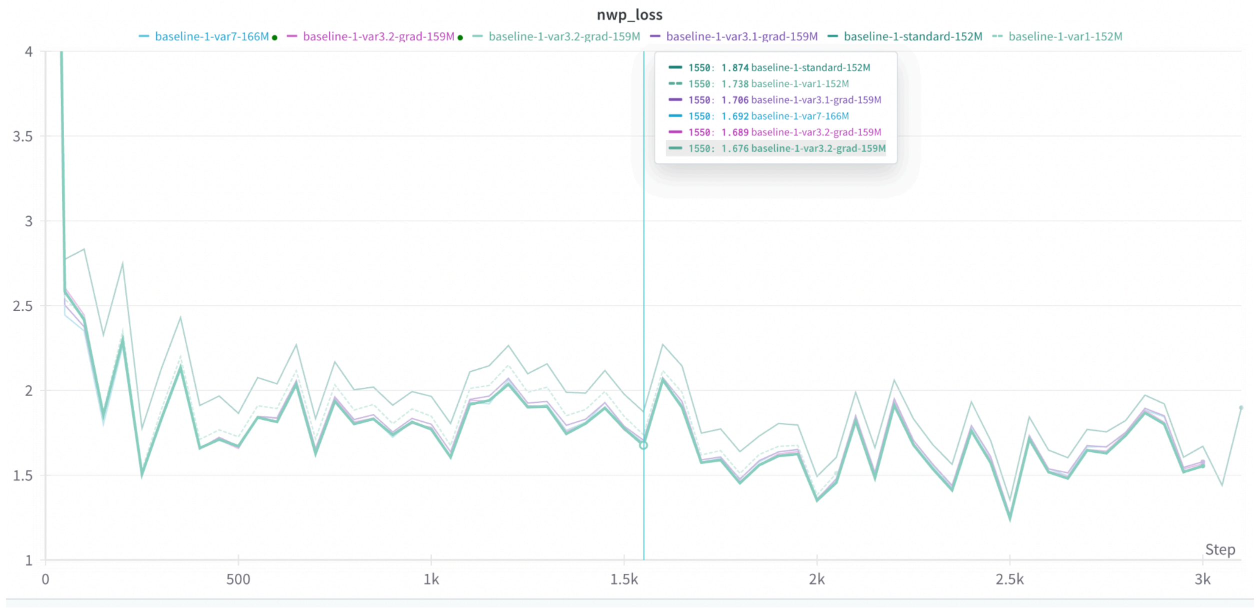
```
0 mean tensor(8.8023e-05)
0 var tensor(0.0008)

1 mean tensor(0.0002)
1 var tensor(0.0009)

2 mean tensor(0.0002)
2 var tensor(0.0009)

3 mean tensor(0.0001)
3 var tensor(0.0009)

4 mean tensor(0.0001)
4 var tensor(0.0010)

5 mean tensor(0.0001)
5 var tensor(0.0010)

6 mean tensor(-7.1553e-05)
6 var tensor(0.0010)

7 mean tensor(-5.3694e-05)
7 var tensor(0.0010)

8 mean tensor(-4.3079e-05)
8 var tensor(0.0010)

9 mean tensor(-1.8247e-05)
9 var tensor(0.0010)

10 mean tensor(-7.0893e-05)
10 var tensor(0.0009)

11 mean tensor(-3.0589e-05)
11 var tensor(0.0007)
```

WI

```
0 mean tensor(-4.7073e-05)
0 var tensor(0.0007)

1 mean tensor(-1.2726e-05)
1 var tensor(0.0009)

2 mean tensor(-8.9185e-05)
2 var tensor(0.0009)

3 mean tensor(-1.5085e-05)
3 var tensor(0.0009)

4 mean tensor(-9.1646e-05)
4 var tensor(0.0009)

5 mean tensor(4.6323e-05)
5 var tensor(0.0010)

6 mean tensor(7.1575e-05)
6 var tensor(0.0010)

7 mean tensor(6.9633e-05)
7 var tensor(0.0010)

8 mean tensor(3.5336e-05)
8 var tensor(0.0010)

9 mean tensor(6.6895e-05)
9 var tensor(0.0010)

10 mean tensor(9.9952e-05)
10 var tensor(0.0009)

11 mean tensor(4.7659e-05)
11 var tensor(0.0007)
```

- I don't see anything obviously wrong with griffin parameters, no order of magnitude blow ups

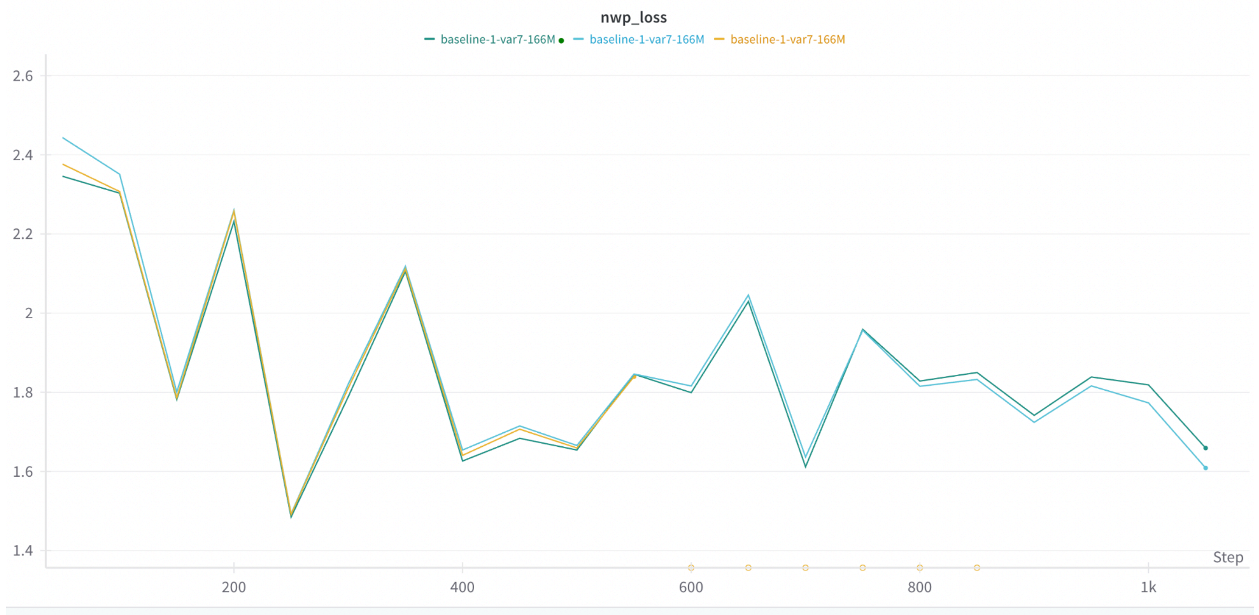
Let's just try adding a norm:
Ok, adding a norm stabilizes training. The result (light blue, var7) is a bit worse than something like 3.2 but it's still good. 3.1 is probably a fair comparison, both look at the previous time step, though 3.1 has slightly fewer parameters. Over the course of the run it's about on par with 3.1.



However, the verison without norm (before it NaN'd) was slightly better than the version with norm, so it's possible the norm degrades performance, although the difference is slight and could just be noise.

Let's try another round without norm to see the numbers, at least before it NaNs.

OK, this lasted 1000 iters before NaN



Yellow = first run (no norm)
Blue = second run (norm
Green = third run (no norm)

- No norm versions do well in the first few iterations, but the norm version starts to pull away quickly.
    - BUT: the pull away in performance may just be because the green run is about to diverge and Nan
    - Overall, the margin between these runs isn't enormous, so I'm going to hold off trying to fix the no-norm version, at least until we finish experiments at this small scale and make a selection of what to run at larger scale

Variation 7.1

Added the 8.2 style softmax convolution in

Works very well, but without norm it Nans

Variation 7.2

Decided to add in the more elements of Griffin:

- Still no caching (requires a lot of reworking and introduces actual recurrence, which I'm trying to avoid)
- Put in the branches in the recurrent block: two branches of linear projection (rnn_width, width, variable names like that: always larger than the argos.dim) of input, 1 branch just gets Gelu, the other branch gets 1d convolution then undergoes the LR-GRU gating with previous state, at the end they're hadamard combined and linear projected out.
- It ends up adding so many parameters, I'm skeptical it's going to work so haven't tried yet

Variation 8 (convolution)

```
# VARIATION 8
# Generalization of 3.1 / 3.2
# Instead of only looking at only previous time step or handcoded previous
two steps (previous_state + previous_previous_state)
# Apply convolution over past n examples
# Gate convolution of previous states with current state gate matrix
# If num_previous_time_steps = 1, this is equivalent to 3.1
# If num_previous_time_steps = 2, this is almost equivalent to 3.2 except
that 3.2 involves sigmoid weighting and this is linear weighting of
preivous states
```

```
class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.num_previous_time_steps = 4 # i.e. kernel size for conv

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       # convolution over n rows at a time
       # padded such that the output at each row index of x is exactly the
n rows above (previous)
       # the first row of output is 0s since there is no previous time
step
       conv = torch.nn.Conv2d(in_channels=1,
                              out_channels=1,
                              kernel_size=(self.num_previous_time_steps,
1),
                              padding = (self.num_previous_time_steps, 0),
                              stride = (1,1),
                              bias=True,
                              device=device)

       conv_over_past_examples = conv(x.unsqueeze(1)) # add dimension to x
(b t d) -> (b 1 t d)
       conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
       conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

       # tensor gate
       gate = torch.sigmoid(self.w_state(x))

       x = gate * x + (1 - gate) * conv_over_past_examples
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

```
# TOY DEMONSTRATION OF CONV
# kernel rows corresponds to how many previous time steps we want to see
# kernel_rows = 3
# data_rows = 10
# batches = 2
# dim = 5

# data = torch.ones(batches * data_rows * dim).float().view(batches, 1,
data_rows, dim)

# conv = nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=(kernel_rows, 1), padding = (kernel_rows, 0), stride=(1, 1),
bias=False)
# with torch.no_grad():
#     conv.weight[0, 0, :, :] = 1. # set to 1s so we can verify easily

# output = conv(data)
# output = output.squeeze()  # Should bring it down to a shape of (1, 7,
1)
# print (data)
# print(output.shape)
# print (torch.split(output, 10, dim=-2)[0])
```

Tried with num_previous_time_steps = 2 and it was about the same as transformer baseline, which is much worse than other variations we've created.

With 4 time steps it's about the same performance as with 2. In theory, time steps = 2 should be almost identical to variation 3.2 which is much better performing, save that we sigmoid gate the time steps together instead of linear convolution combination.

It's odd that a sigmoid there would have such a large effect. The best interpretation I have is that it is not that we sigmoid gated previous state and previous previous state (gate * PS + (1-gate) * PPS, it's that we applied a nonlinearity of any kind.

So let's do the same thing, but we'll do sigmoid(conv_over_past_examples)

Variation 8.1 (sigmoid the convolution output)

OK, this also didn't work.

The other way to create this is something like:
- set up an empty conv layer,
- initialize some parameter scalars,
- and then the softmax(scalars) are used as weights of the conv layer.

This seems closest to what the sigmoid gating over previous states does.

Let's do that:

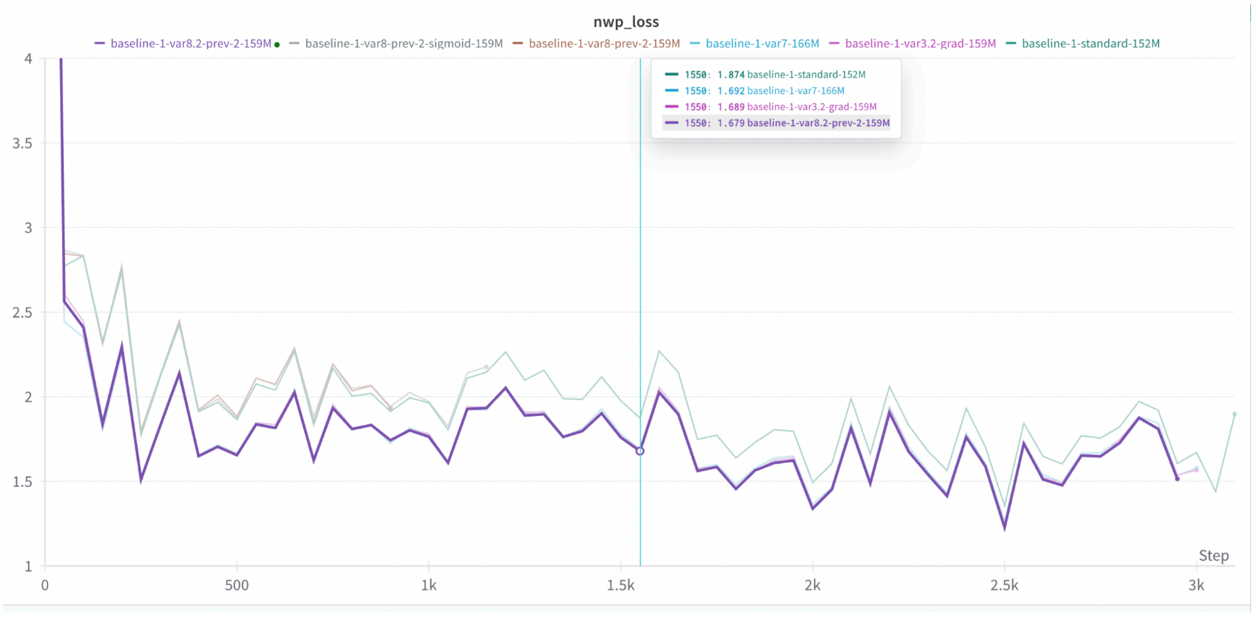
Variation 8.2 (softmax the conv weights prior to conv)

The convolution kernel weights are first passed through softmax before being applied. You can think of this almost as a (softmax) weighted average of n past examples

```
# VARIATION 8.2

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.num_previous_time_steps = 2
       self.weights = nn.Parameter(torch.randn(1, 1,
self.num_previous_time_steps, 1))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
       softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
       softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape
```

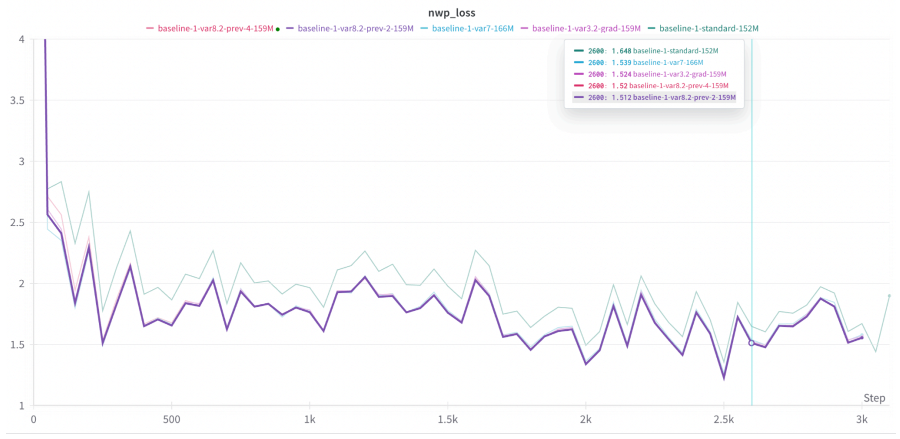
```
        # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
        conv_over_past_examples =
torch.nn.functional.conv2d(x.unsqueeze(1),

softmax_weights,
                                                                      bias=None,
                                                                      stride=1,

padding=(self.num_previous_time_steps, 0))


        conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
        conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

        # tensor gate
        gate = torch.sigmoid(self.w_state(x))

        x = gate * x + (1 - gate) * conv_over_past_examples
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

OK, this works as expected. The performance of this, with last 2 states, matches 3.2 which it should be basically equivalent to (it's softmax over 2 values vs sigmoid over one value and 1 - sigmoid(value).



So this looks like it (purple) works well for 2 states, on par / beating 3.2 and 7. Let's increase the previous states it can see.

OK, using the previous 4 states is pretty good but worse than only using the previous 2 states:

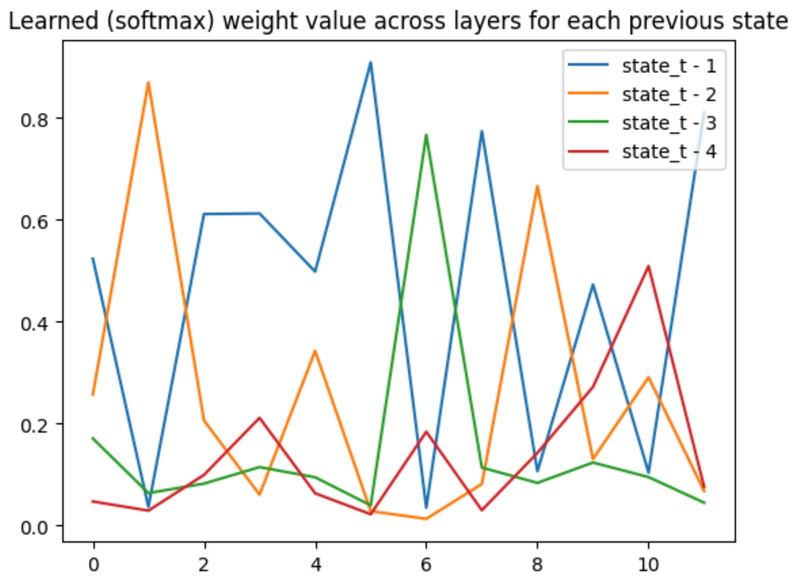


The behavior of the convolution weights:
After 3000 iters, the weights for four previous examples at each layer.

```
vals = []
for i in range(12):
    # !! Note i've flipped these so that the first weight will be the
preivous example, second weight is previous previous example, etc.
    conv_weights =
F.softmax(model.layers[i].feed_forward.weights.detach().cpu(),
dim=-2).squeeze().flip(0)
    vals.append(conv_weights)
vals
```

```
[tensor([0.5240, 0.2573, 0.1713, 0.0475]),
 tensor([0.0369, 0.8697, 0.0639, 0.0295]),
 tensor([0.6115, 0.2064, 0.0825, 0.0996]),
 tensor([0.6126, 0.0608, 0.1149, 0.2116]),
 tensor([0.4986, 0.3429, 0.0949, 0.0636]),
 tensor([0.9092, 0.0285, 0.0399, 0.0224]),
 tensor([0.0355, 0.0134, 0.7668, 0.1844]),
 tensor([0.7740, 0.0817, 0.1140, 0.0303]),
 tensor([0.1067, 0.6666, 0.0839, 0.1428]),
 tensor([0.4732, 0.1307, 0.1239, 0.2723]),
 tensor([0.1041, 0.2912, 0.0953, 0.5095]),
 tensor([0.8108, 0.0680, 0.0450, 0.0762])]
```

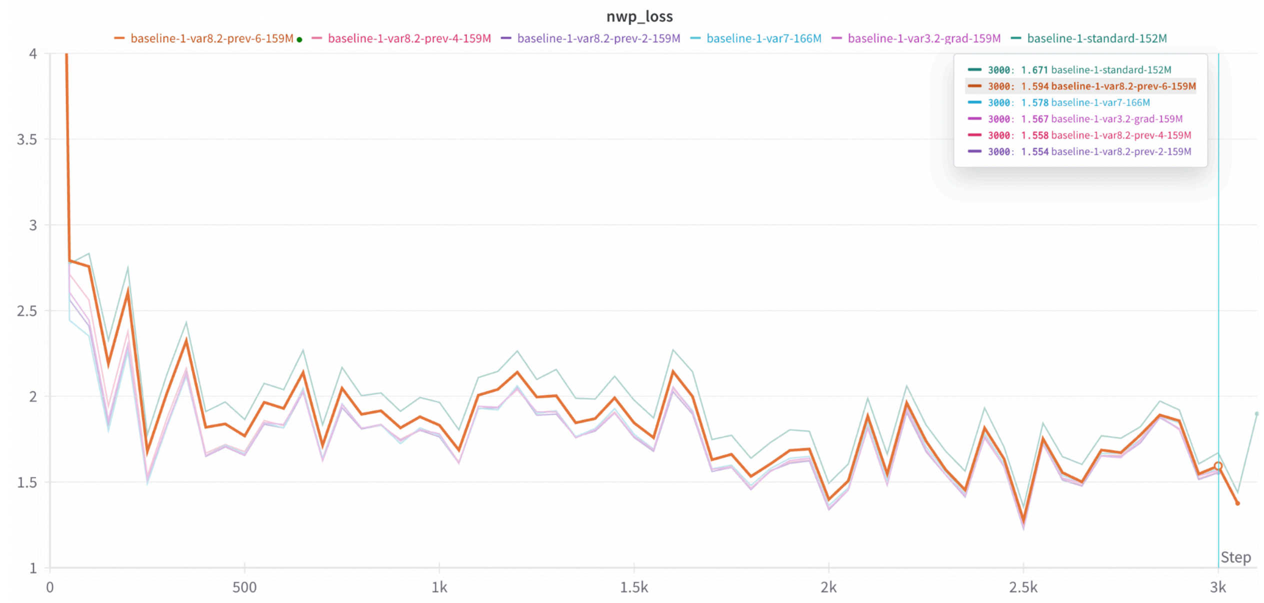
```
for i in range(vals[0].shape[0]):
    plt.plot([x[i] for x in vals], label=f'state_t - {i + 1}')
    plt.legend()
plt.show()
```



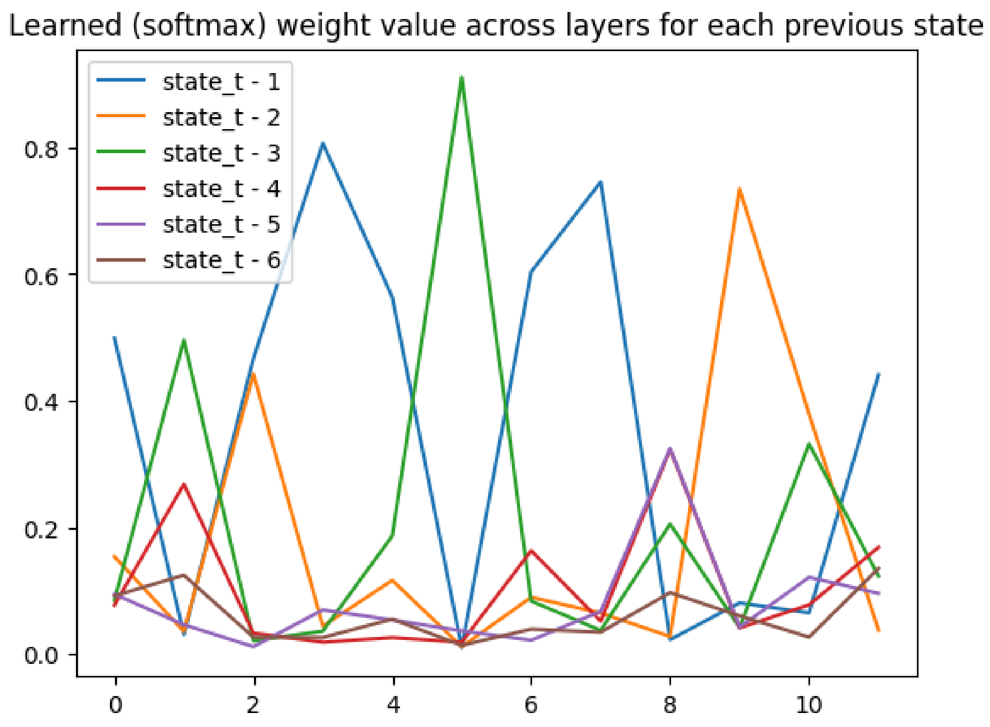
- So: the previous example (blue line) tends to have the highest weight, then the example before that.
- The t-3 stays almost at zero but at layer 6 becomes very important?
- T-4 stays low but becomes important in the last layers.
- The interpretability is probably also confounded by these being randomly initialized instead of something simple where we'd see a smoother evolution, but maybe not.
- TODO: pull these values and chart them over time.

6 previous steps (orange/brown line) beats baseline but is worse than fewer steps, though the difference seems to shrink as training goes on.

I don't think it's worth testing previous step sizes until we're on a larger model.



Examining the convolution weights per layer:

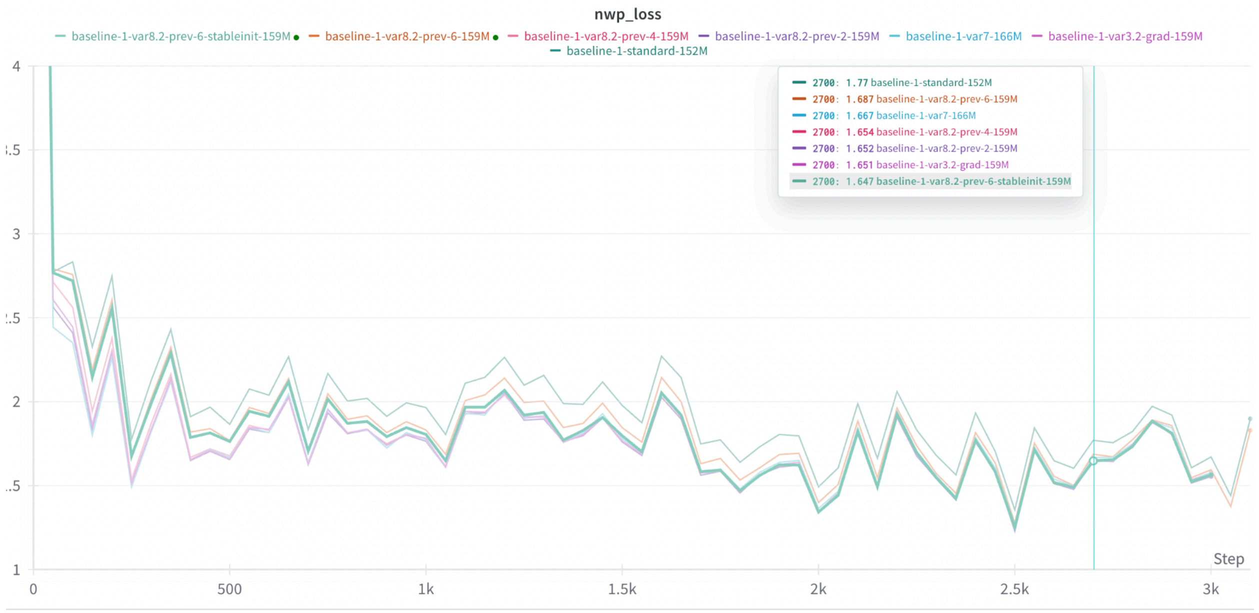


- Somewhat in line with what we think: the further back the state the less weight it has throughout.
- Interestingly one of these steps will spike at a given layer and the others will dip to 0.
    - T-3 spikes at layer 5 just as all others go to 0
    - Same for t-2 at layer 9
    - Same(ish) for t-5 at layer 8.
    - So in a way each of the previous states has one layer focusing on it more than others.
    - Again: this could just be due to wonky weight initialization, so let’s rerun this with a flat weight initialization across layers.

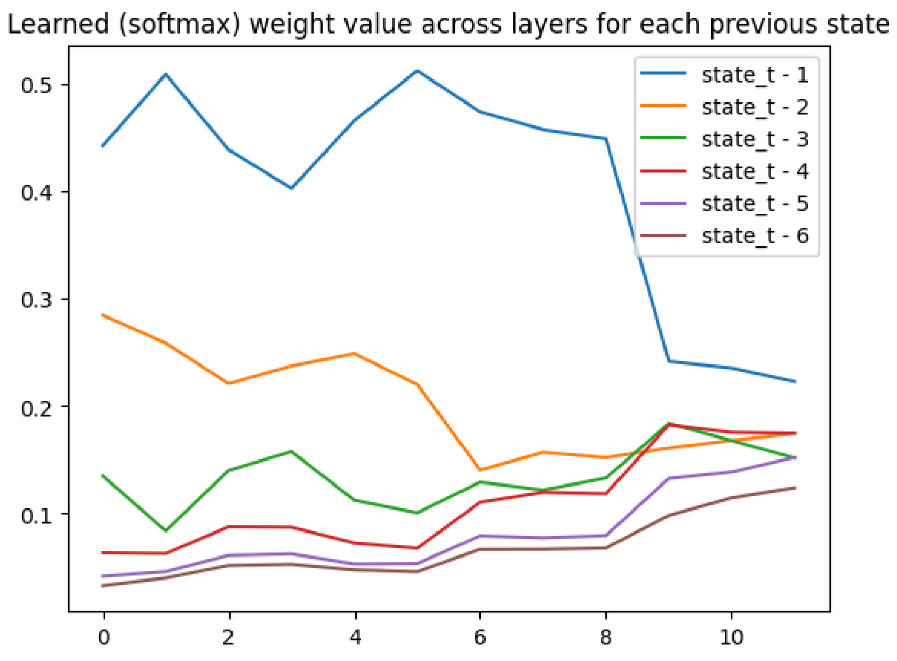
Next experiment:

Stable initialization of the convolution weights (6 steps, so each weight is initialized as .166 roughly ⅙)

Better performance:



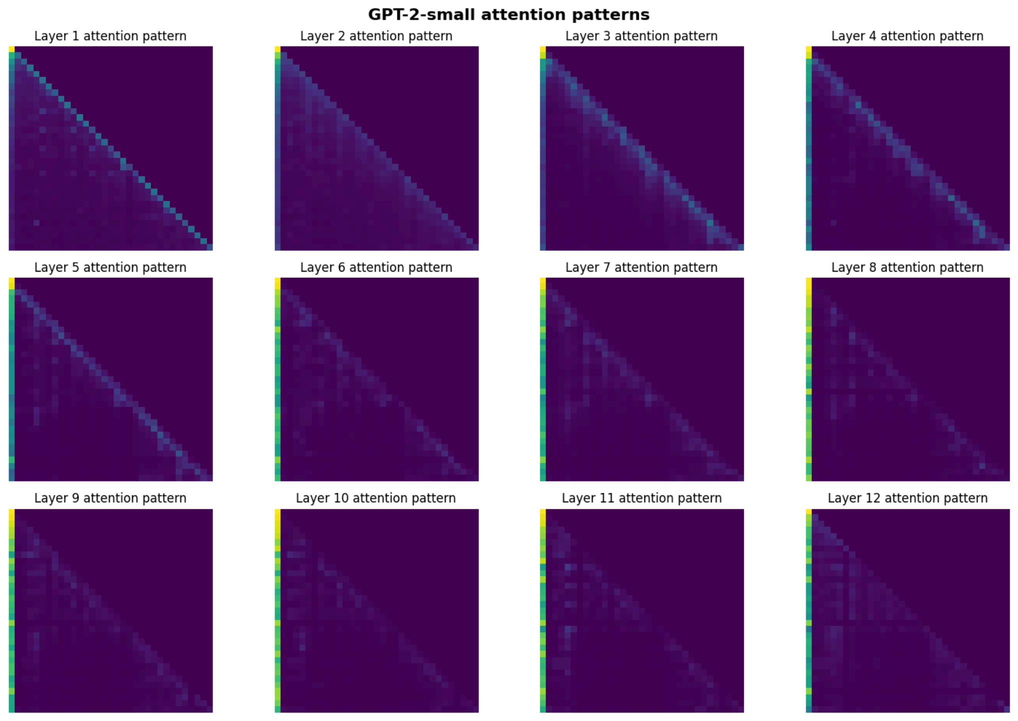
- This starts out worse than alternatives but eventually has the lowest loss, more or less tied with 8.2 when we only look at the last 2 steps.
- So initialization of the conv weights ends up mattering a lot.



- This is very well behaved (3000 iterations of training)
- Closer states get more weight, but all states converge to same weight in later layers
- This is an interesting graph. It might sound naive but we should ask and examine: why does a given example/state prefer to look at its immediate predecessors over very

distant ones? There is no bias towards recency built into the model: it (previous word as strong predictor of next word) is only in the data.

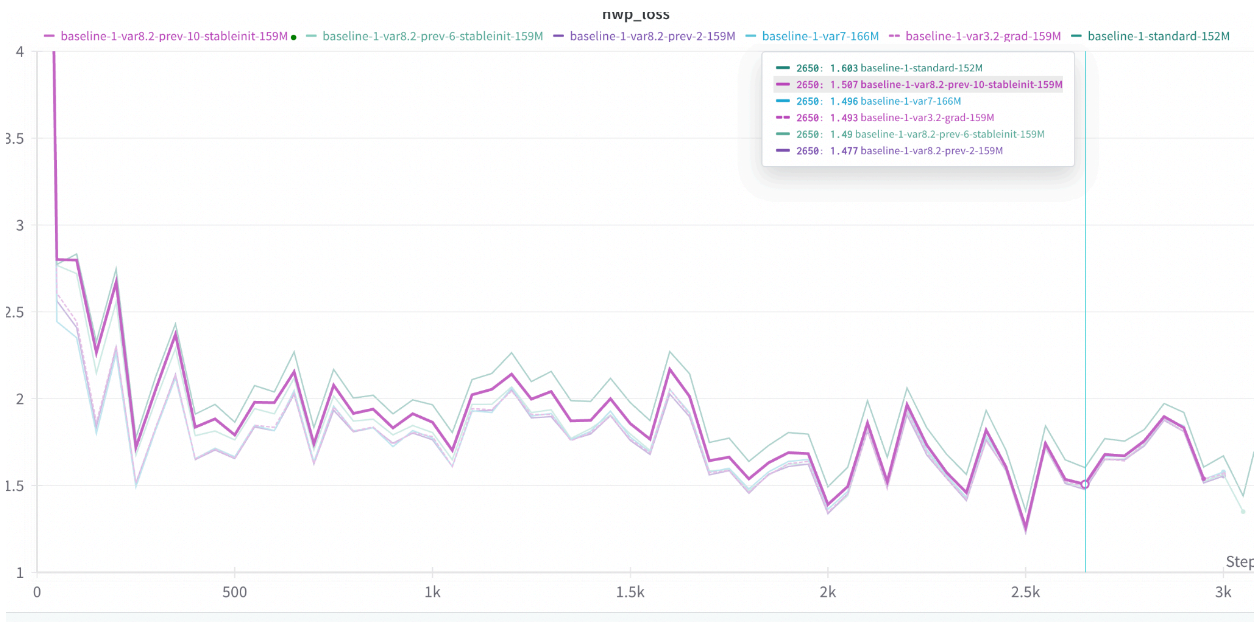
- Note also the difference across layers. In early layers information about immediately recent context is important, but towards later layers the context window gets larger and larger until at the end each state is almost equally useful to see. So as the data flows through the model it starts by looking at recent context and gradually incorporating more distant context.
- Note difference / similarity with self attention:
    - Difference: each token attends to each other token in a way that is unbiased by time (of course there is position bias almost always built in, but even without position bias decoder models learn sequence order (NoPE paper, others). So one gets attention patterns that are (in theory) based in semantics rather than recency: attention strength should just be affinity
    - Similarity: these similarities/affinities/attention patterns end up in practice being heavily influenced by the time dimension. More recent tokens play a more important factor, on average, than distant ones.
        - As we go deeper into layers, the attention patterns start out heavily focused on recent tokens and gradually focus grows more diffusely across distant context
        - Our previous state mechanism can be thought of as very close to sliding window attention.



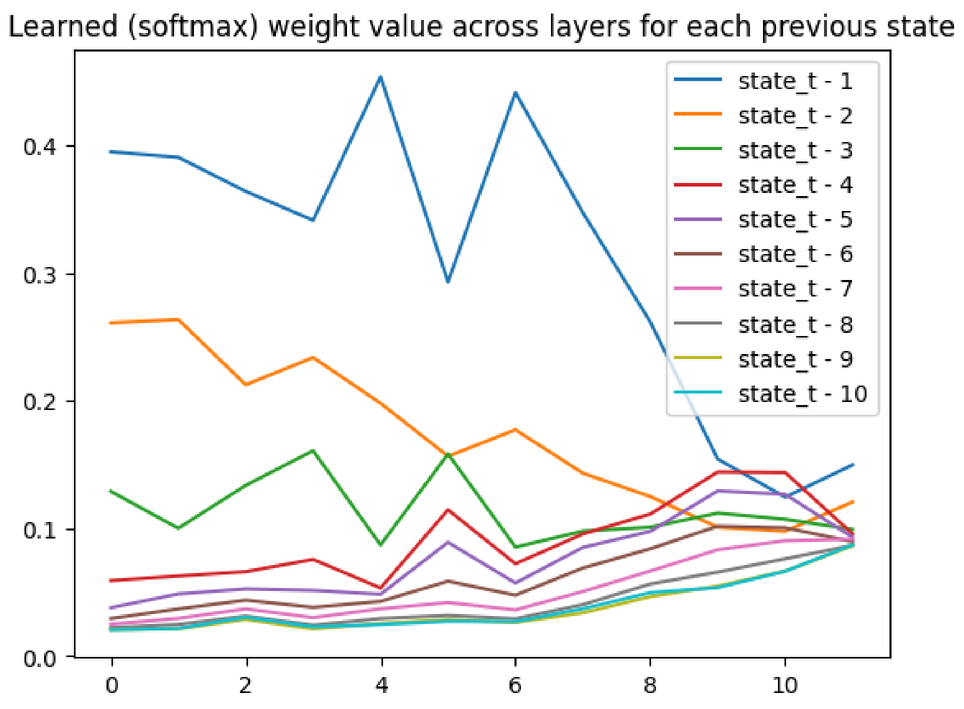
(attention patterns over a sequence fed into GPT-2, built quickly on top of the TransformerLens demo notebook.)

We would probably like to keep track of the balance between previou state and current state (torch.sigmoid(self.w_state(x)).

Let’s look at 10 steps:



- Performance starts out poor but gets better over time, but mostly as all variations converge a little



- Very interpretable and in line with past (fewer steps)

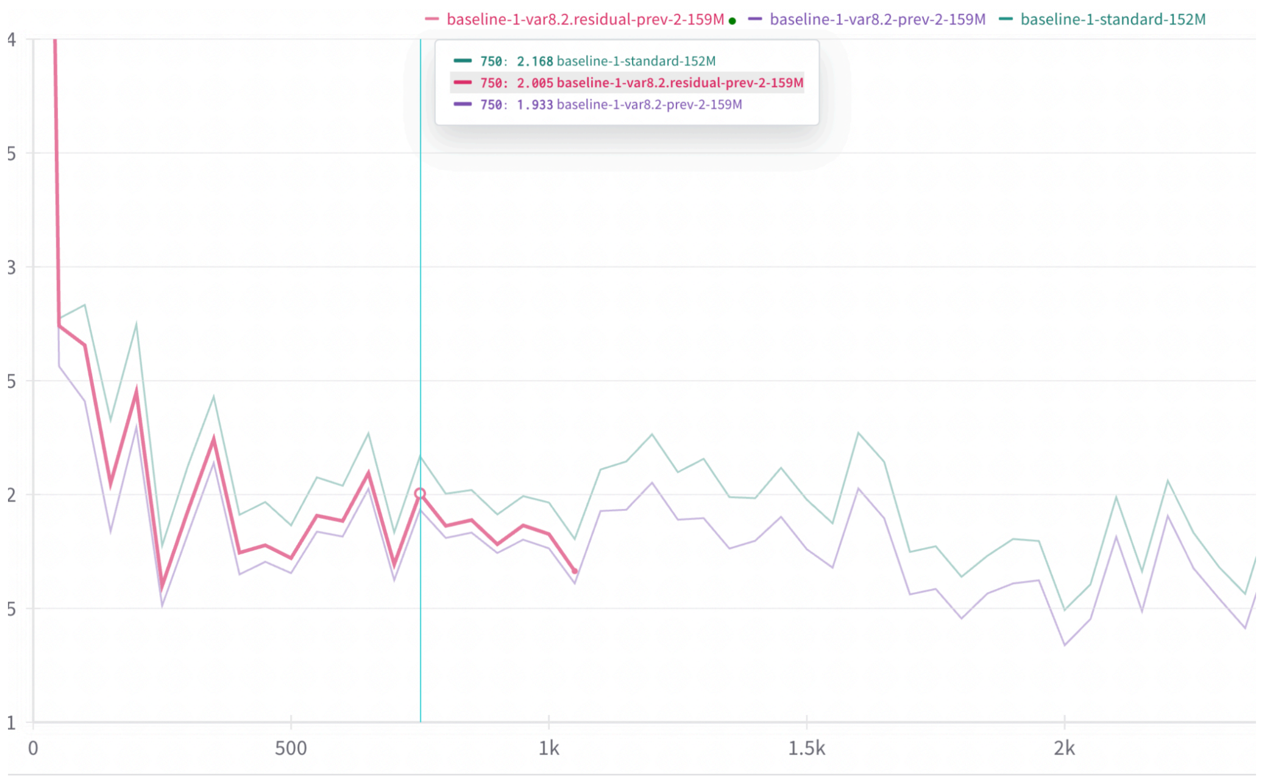
8.2 Residual

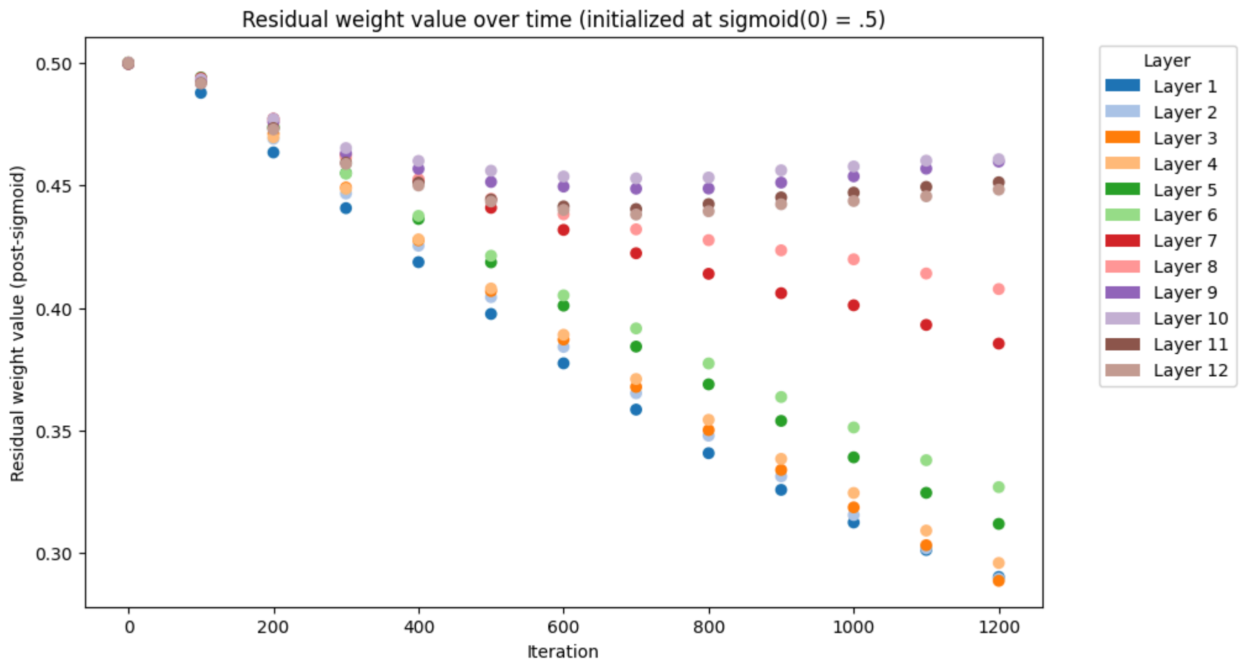
Simply add a sigmoid residual weight to the input vs the recurrent chunk so we can see how heavily the model uses it. Actually, we should track the residual strength for all blocks: attention, recurrence, and FF to see how much each is used over time and over layers.

```
# VARIATION 8.2.residual
# Residual weight
# For analysis
# Same as 8.2 but with residual connection and weight to see how much data
is actually passed through the recurrence block.


class FeedForward(nn.Module):
    # SwiGLU
    def __init__(self, args: ModelArgs):
        super().__init__()

        self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
        self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w_state = nn.Linear(args.dim, args.dim)
        self.num_previous_time_steps = 2
        self.weights = nn.Parameter(torch.ones(1, 1,
self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))
        self.residual_weight = nn.Parameter(torch.zeros(1))

    def forward(self, x) -> torch.Tensor:
        batch_size, seq_len = x.shape[0], x.shape[1]

        weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
        softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
        softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

        # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
        conv_over_past_examples =
torch.nn.functional.conv2d(x.unsqueeze(1),

softmax_weights,
                                                              bias=None,
                                                              stride=1,

padding=(self.num_previous_time_steps, 0))


        conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
        conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

        # tensor gate
        gate = torch.sigmoid(self.w_state(x))

        recurrent = gate * x + (1 - gate) * conv_over_past_examples
        x = (torch.sigmoid(self.residual_weight) * x) + ((1 -
torch.sigmoid(self.residual_weight)) * recurrent)
```

```
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

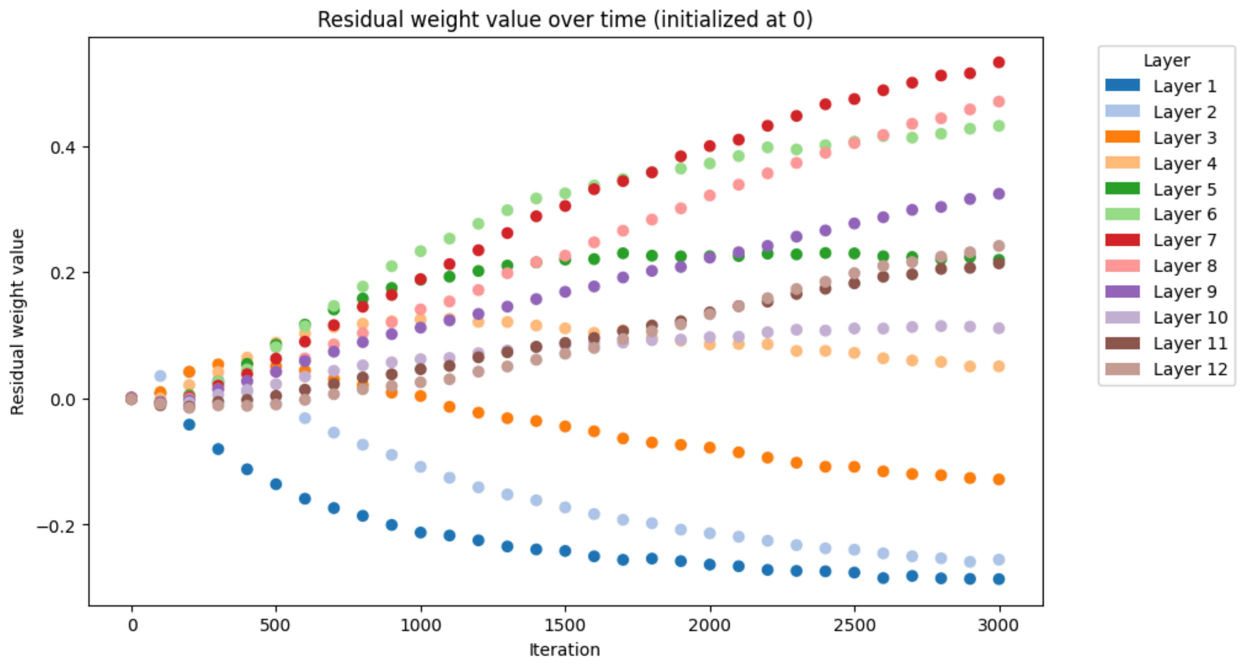
Interestingly the model is much worse. Perhaps a sigmoid weight is incorrect and we should just do a naked linear weight on the residual. We'll stop and run a normal linear weight





With linear weight (x = residual_weight * x + recurrence(x))

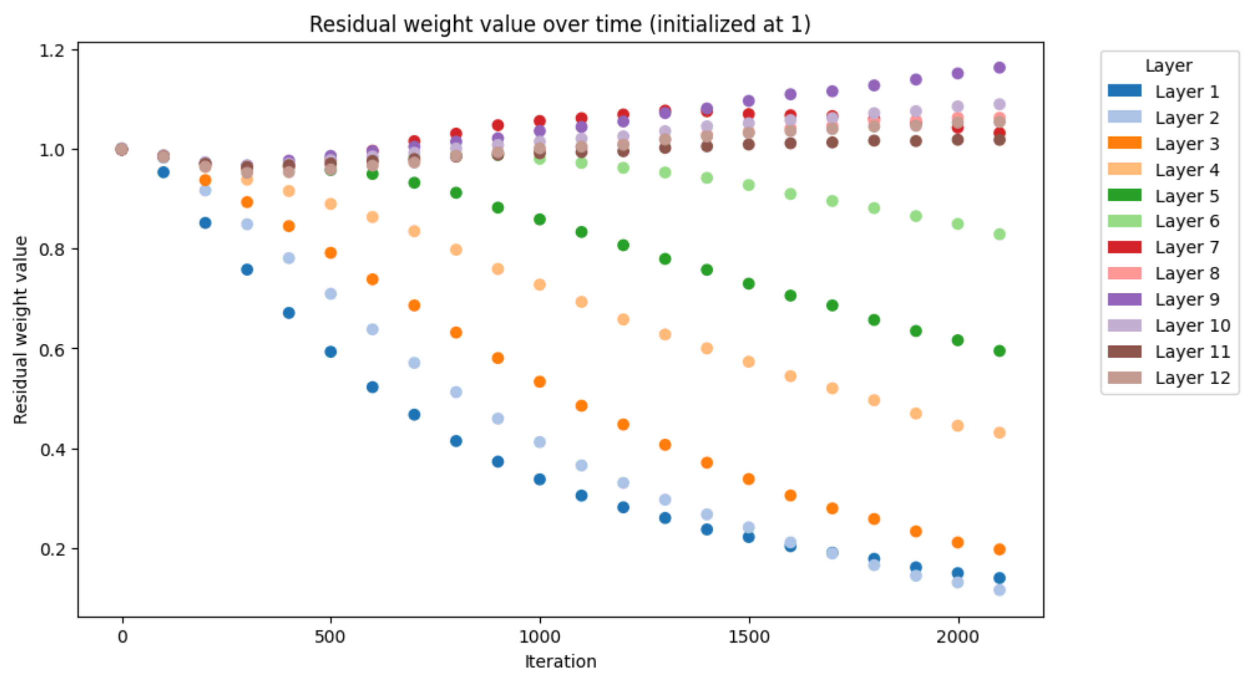
I realized the weight was still initialized at 0. Even still it does just as well as 8.2, if not a little better.

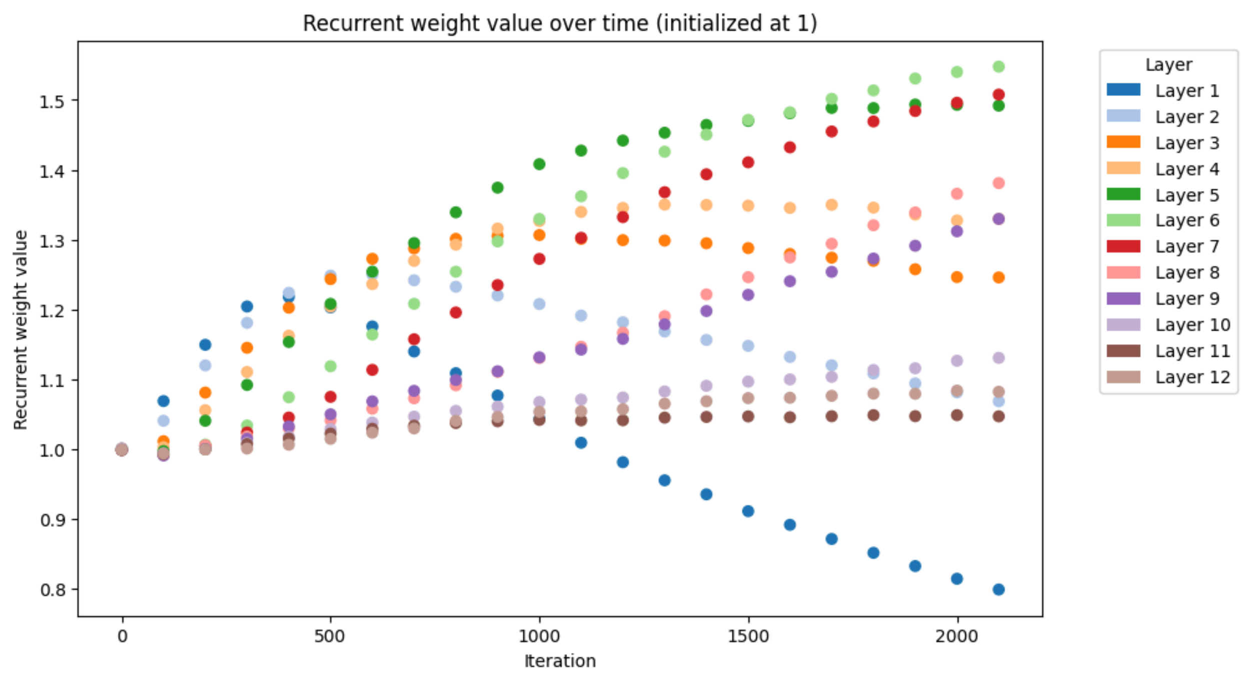


- This (initializing a residual weight at 0) can be interpreted as beginning with no residual connection and learning one if desired.
- Interestingly the value goes negative for the early layers.
    - Why wouldn't it just stay at 0?
    - Negative implies maybe that it hurts but seomthign more interesting: that subtracting the input representation from the transformed representation actually helps

OK, with linear weight initialized at 1. In fact this is:
X = residual_weight * x + recurrent_weight * recurrence(x), where both weights are a single scalar

Residual





Variation 8.2 All Residual - transformer block and recurrent

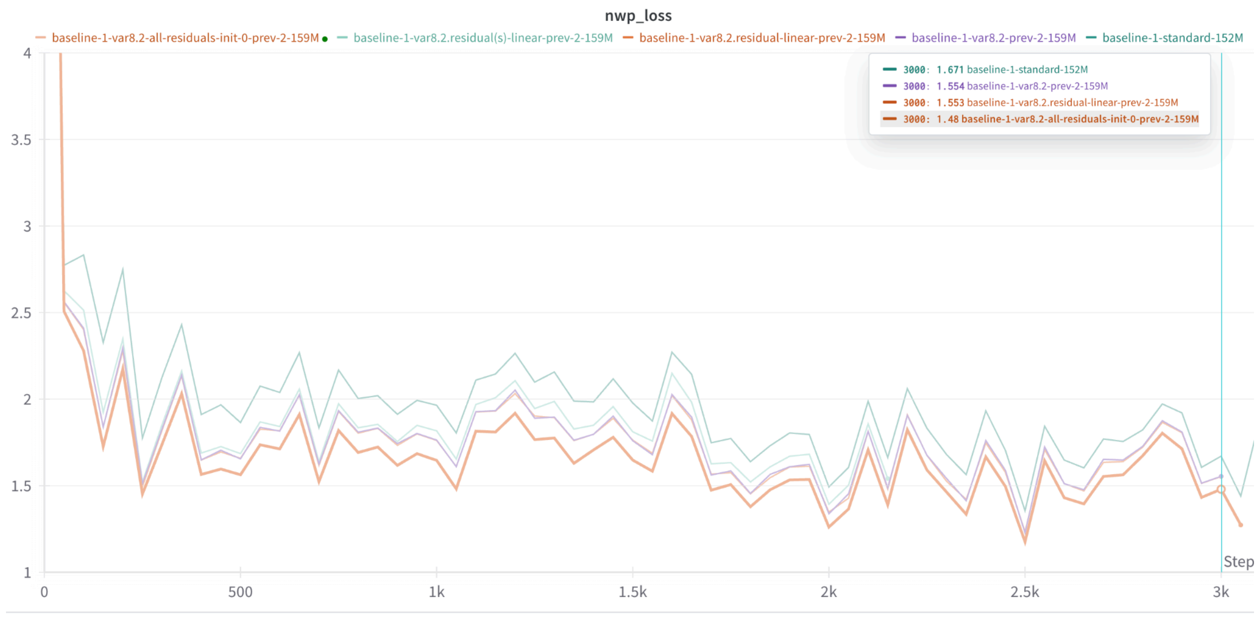
Inspired by the behavior when we accidentally initialized learned residual connections at 0, let's set the recurrence residual weight at 0 and the transformer block residual weight at 0. That is:

we initialize all residual connections at 0 and give the model the ability to parameterize this weight.

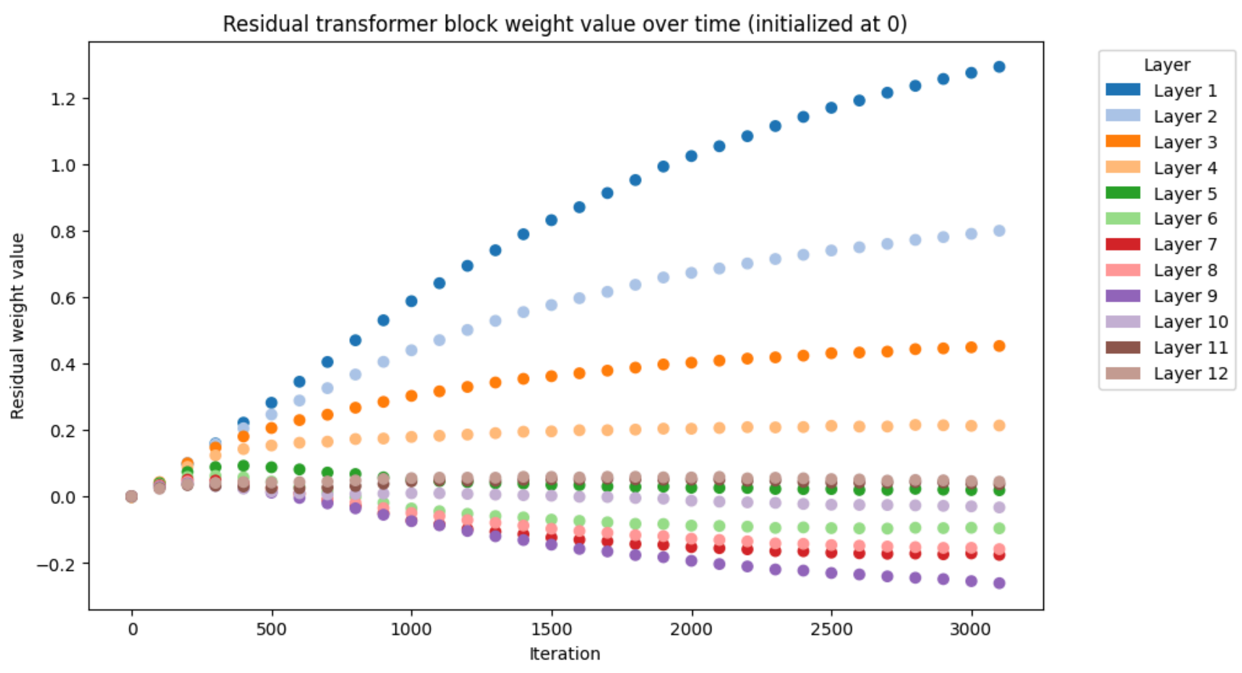
In the transformer block, we have a residual weight initialized at 0.

```
        r = x + self.attention.forward(self.attention_norm(x),
rotary_emb_fn)
        h = self.feed_forward.forward(self.ffn_norm(x))
        x = self.residual_weight * x
        out = x + h + r

        return out
```

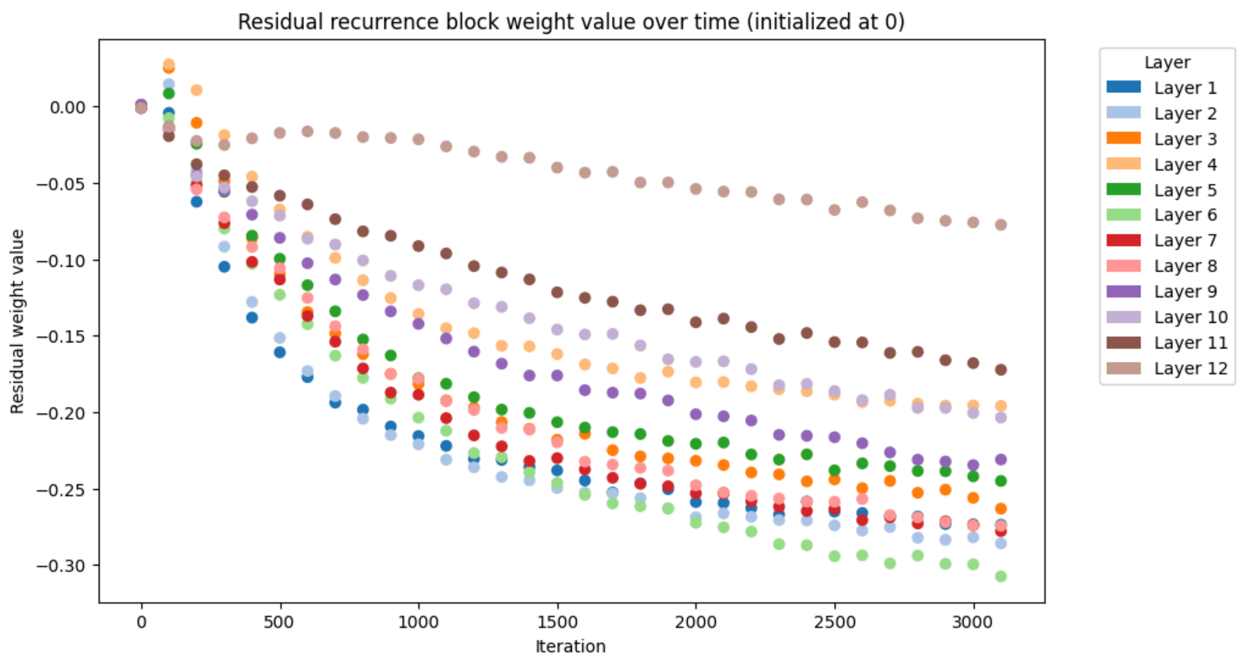
First, the performance is notably better from this small change:



- Throughout training, by adding these few weights, the model is much better throughout all steps
- If the transformer block residuals are initialized at 1, performance is not nearly as good as when they are initialize at 0



- The above is the residual for the entire transformer block
- Early layers greatly prefer to pass through their data in the residual stream rather to use the transformed data.
    - TODO: is there any merit in having narrower early layers, or having an inverted pyramid structure to LLMs to more efficiently use parameters?
- Later layers not only prefer the transformed data, they actually would like to subtract the input representation (this is how I read it)! Very interesting.

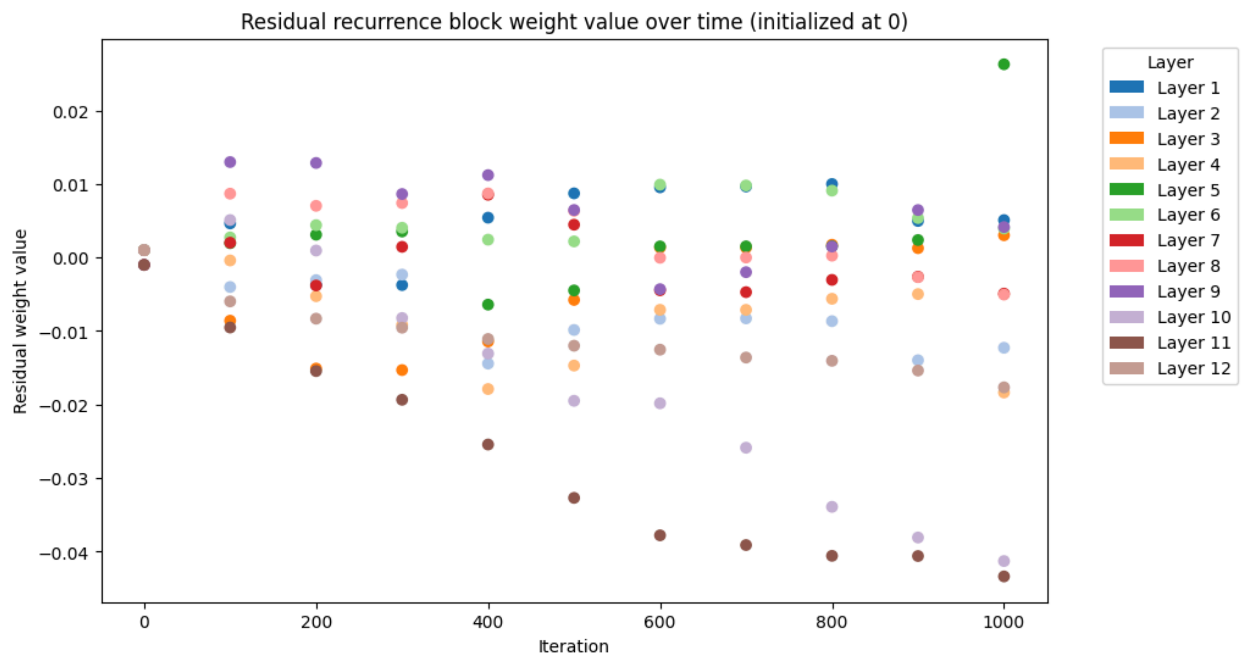


- Here as well in the recurrence block, the model actively prefers to subtract the input representation

Adding residual weight to self attention component

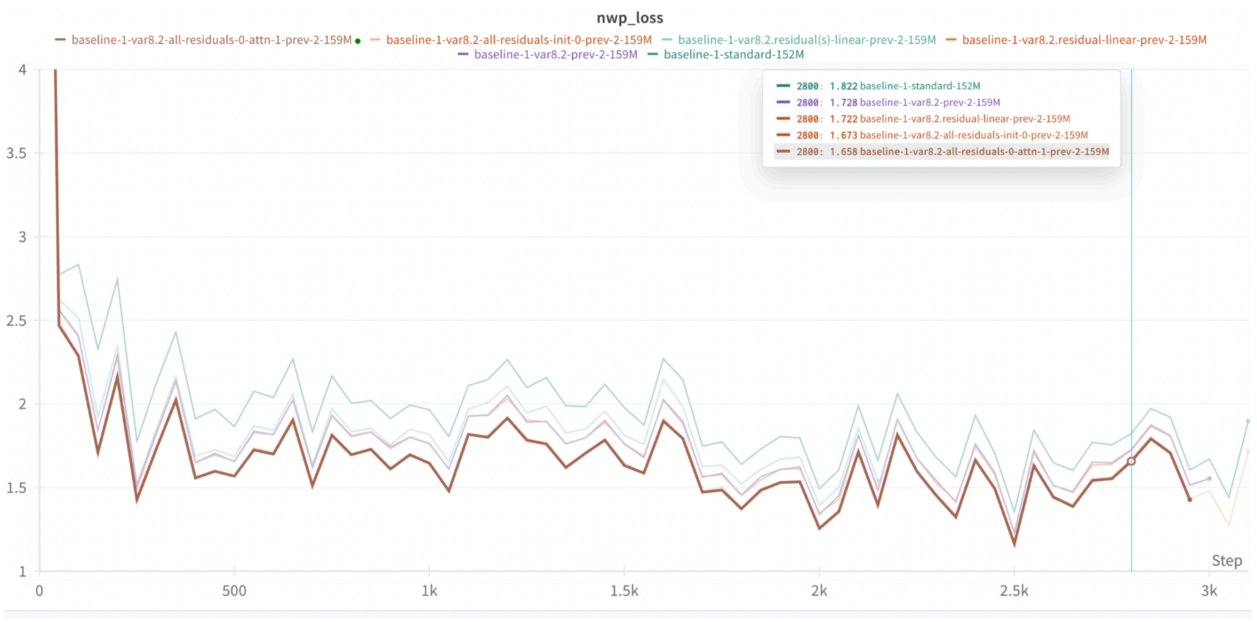
If you add a 0 residual weight to attention, loss is bad and everything goes strange actually:

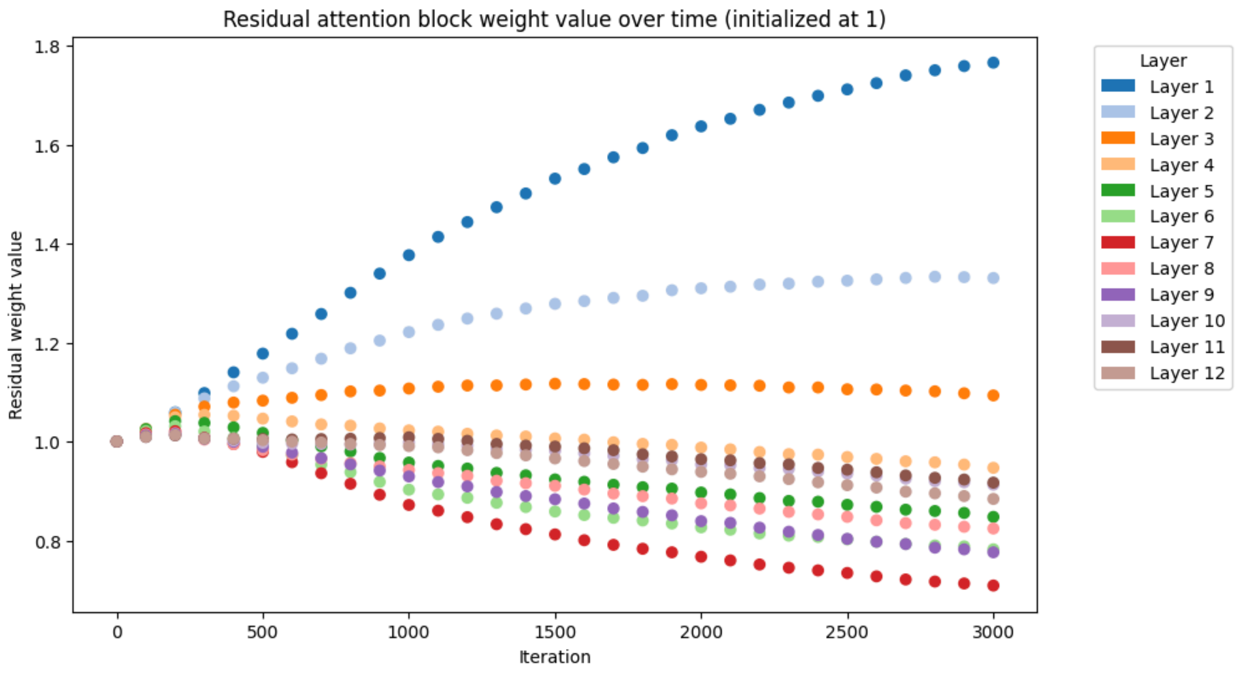
```
step 0 :training loss: 10.984538078308105
step 100 :training loss: 4.680907726287842
step 200 :training loss: 5.1240034103393555
step 400 :training loss: 4.418183326721191
step 500 :training loss: 4.562368869781494
step 600 :training loss: 4.919055461883545
step 700 :training loss: 4.79615592956543
step 800 :training loss: 4.84099817276001
step 900 :training loss: 4.5317535400390625
step 1000 :training loss: 4.914287567138672
```



OK, if we keep the weight on the attention residual but initialize at 1.

Performance is about the same as without an attention residual weight, but perhaps a little better:

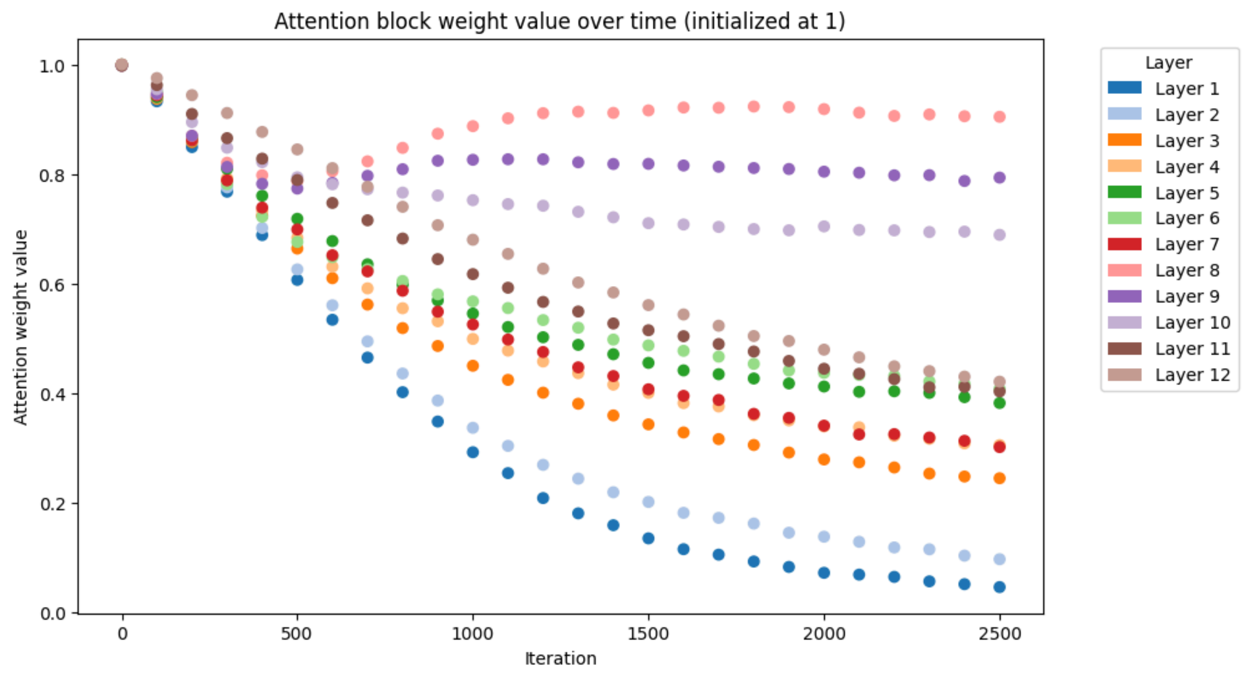




- For the first few layers, the residual connection is very important
- The residual here seems constantly important across layers and time

Let’s try a weight on the attention itself, rather than the attention residual

Performance is basically on par or slighty better than with the weight on the residual.



- Low layers focus on passing residuals through
- Middle layers (8,9,10) are a sudden jump up to heavily focused on attention, interestingly
- The jump between layer 7 and 8 is strange
- End layers (11, 12) much less dependent on attention computation.

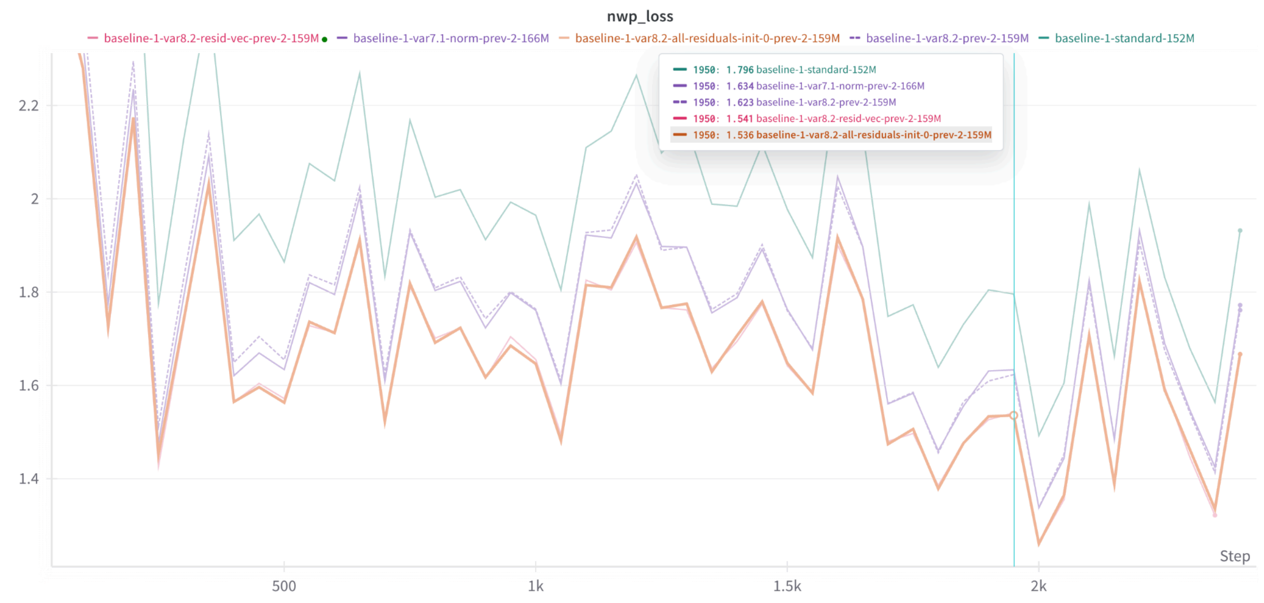
Args.dim residual weight

The residual weight on the transformer block

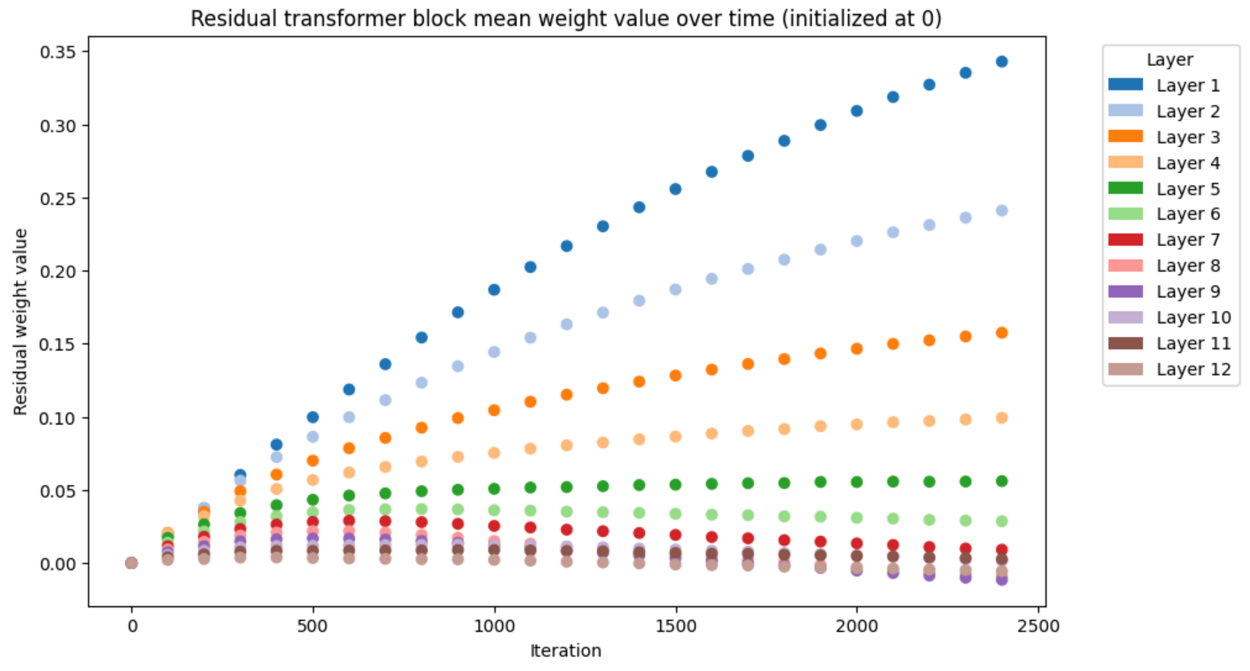
```
class TransformerBlock(nn.Module):
    def __init__(self, args: ModelArgs):
        super().__init__()
        self.n_heads = args.n_heads
        self.dim = args.dim
        self.attention = Attention(args)
        self.feed_forward = FeedForward(args=args)
        self.attention_norm = RMSNorm(args.dim, eps=args.norm_eps)
        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)
        self.args = args
        #self.residual_weight = nn.Parameter(torch.zeros(1))
        self.residual_weight = nn.Parameter(torch.zeros(args.dim))

        #self.attention_residual = nn.Parameter(torch.ones(1))

    def forward(
        self, x: torch.Tensor,
        rotary_emb_fn
    ) -> torch.Tensor:
        # PALM parallel layers (pre-norm):
https://arxiv.org/pdf/2204.02311.pdf
        #r = x + (self.attention_residual *
self.attention.forward(self.attention_norm(x), rotary_emb_fn))
        r = x + + self.attention.forward(self.attention_norm(x),
rotary_emb_fn)
        h = self.feed_forward.forward(self.ffn_norm(x))
        x = self.residual_weight * x
        out = x + h + r

        return out
```

Is what made the big performance difference. So here we will make it a (1,args.dim) vector of 0s instead of a (1,1) scalar of 0s.

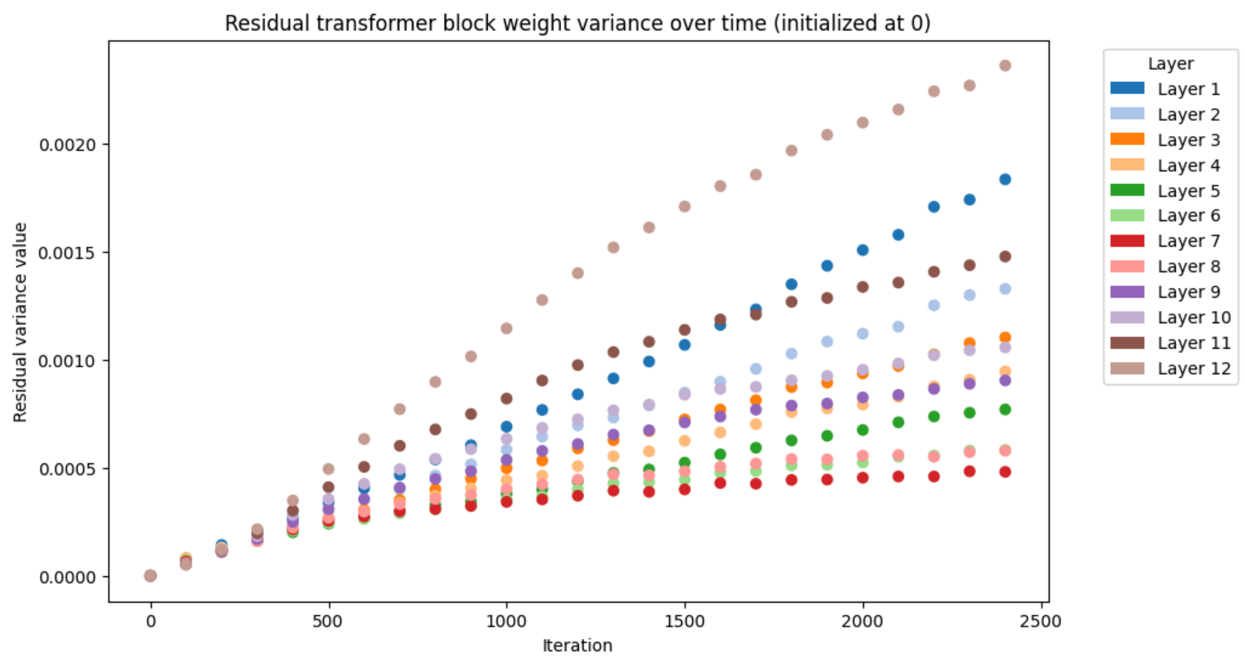
Surprisingly using a args.dim size weight is basically the same performance as with a scalar. This run (pink) is basically overlaid on the scalar weight run (brownish-orange)



Looking at the mean value of these vectors:



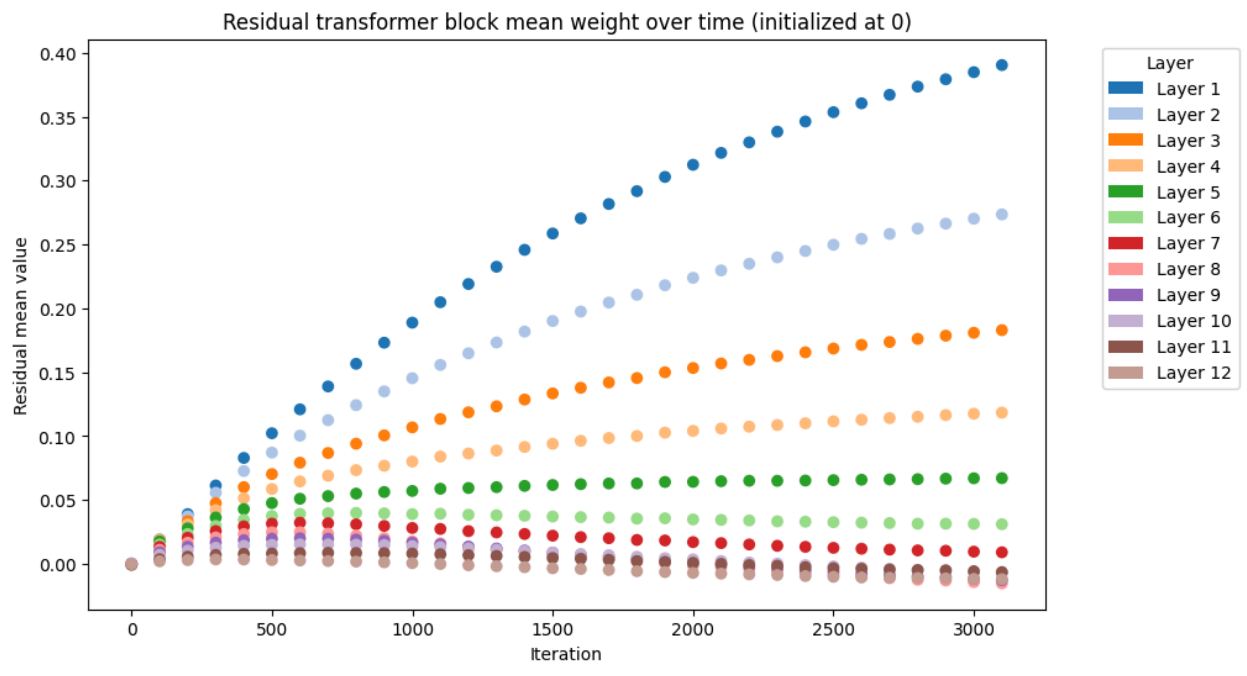
- Much gentler slope going negative

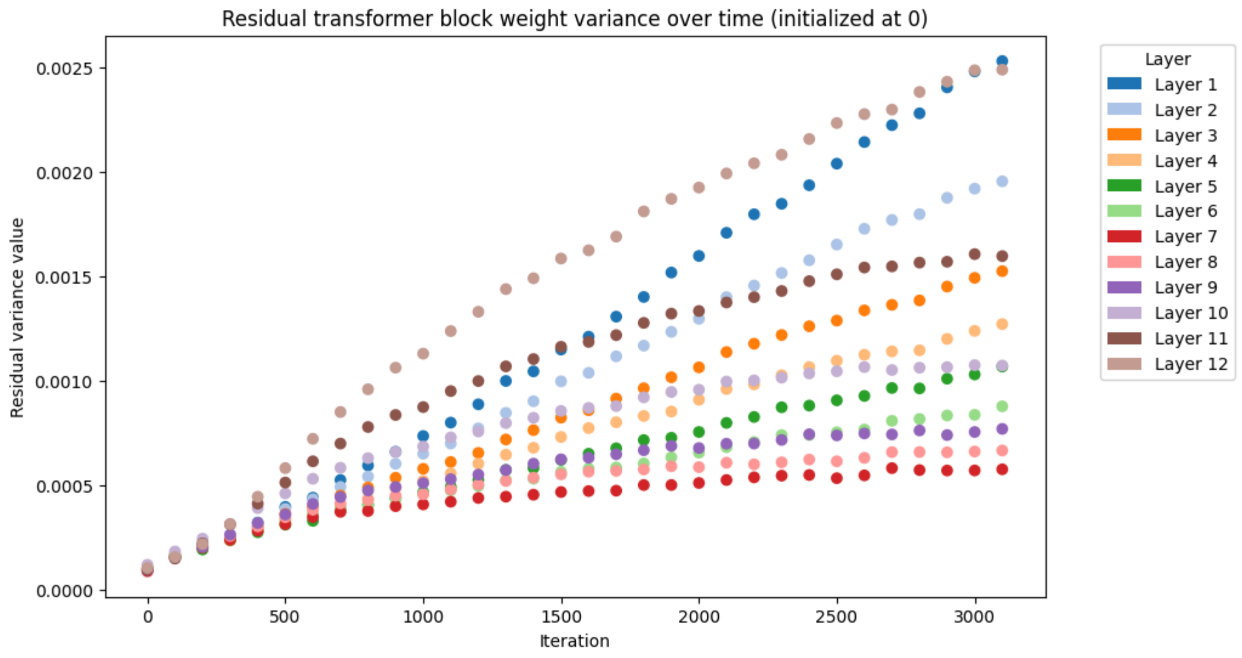


- Looking at the variance here, it seems the later layer residual vectors do start to take on more variance. Maybe we use some random initialization around 0? I think it makes sense that a vector would be better than a scalar for this reason…

We'll add a little noise to the zero vector (+ torch.randn(args.dim) / 100) and see what happens..

Eh, very similar





Create a skip weight for: block (identity), ff, attention, recurrence

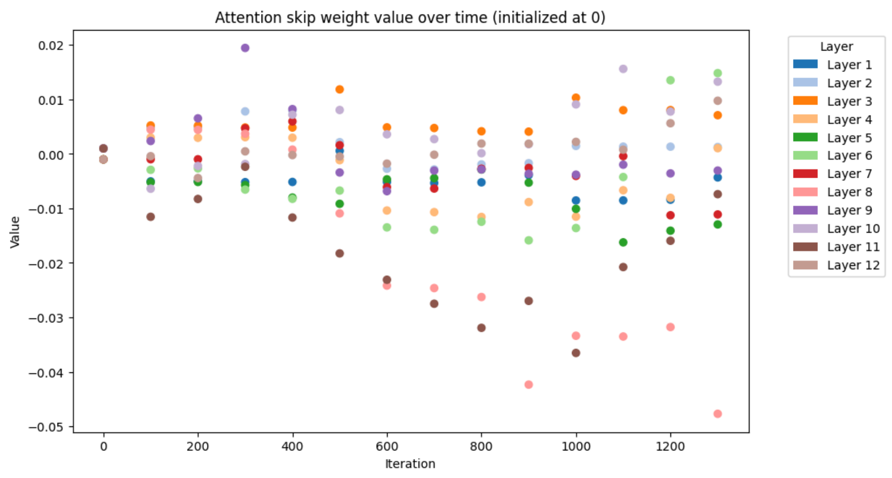
We want to see the weight of the skip connection as a proxy for how "important" the associated residual is, like measuring how much each residual computation is valued by the model.

We can also add a linear weight to the residual itself, but we'll start here.

Init all at 0

Performance is not good

All the skip weights look like this, no real pattern:

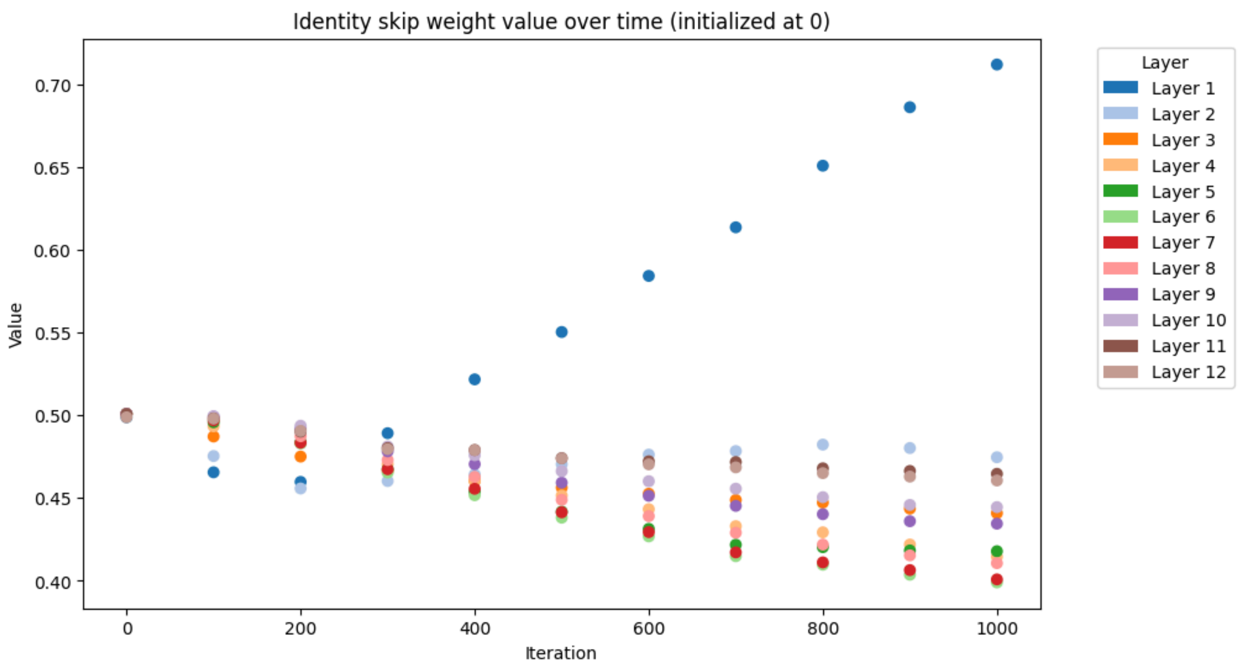


So clearly the model's having trouble nad some of these skip weights need to be initialized higher.

Init all at .5

If we see the direction the skips head when initialized at .5 (up/down) then we can initialize better

OK, ff, attention and identity skip all have the exact same values. Why?

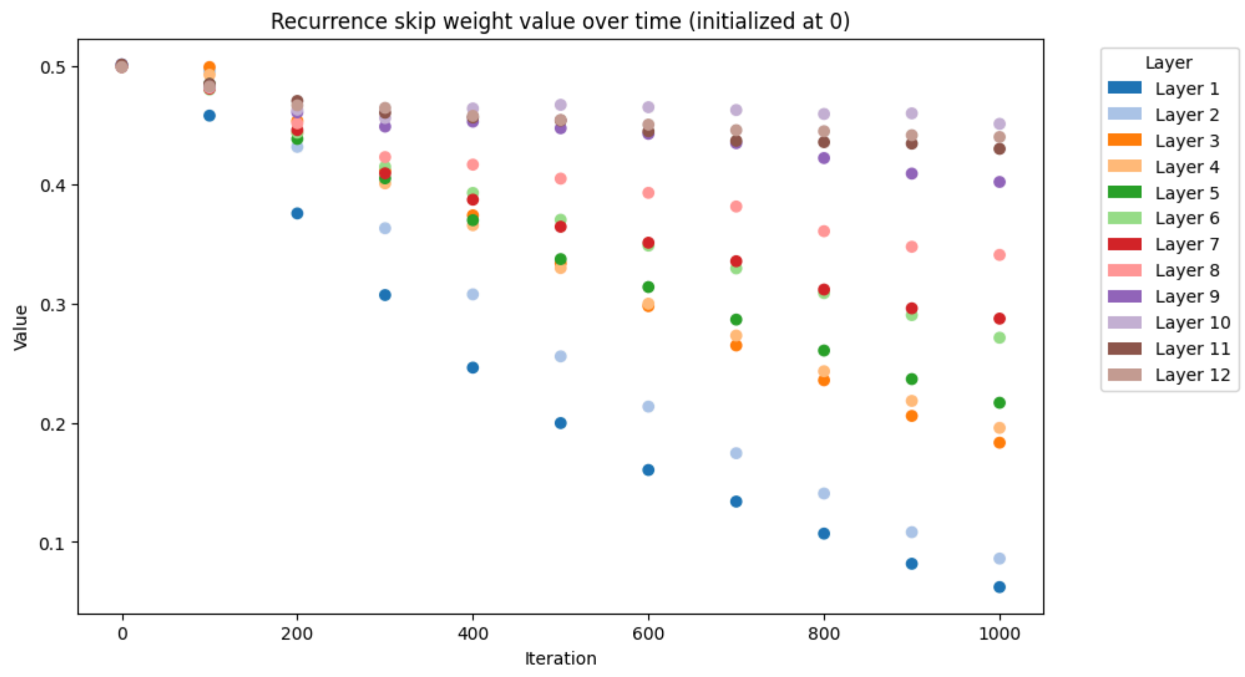


- Identity, ff, and attention all have the exact same values

```
        r = r + self.attention_skip * x
        h = h + self.ff_skip * x # adding a ff skip
        x = self.identity_skip * x

        out = x + h + r

        return out
```

If you write it out, it's just fusing these as one weight on the skip connection.

Recurrence weight



- Heading to 0 for early layers, about .5 for later layers.

Init: Custom1 - all residuals + skips

We're actually going to attach weights to the residual (computation) instead of the skip. Ignore the variable name, this is just easier to run quickly.

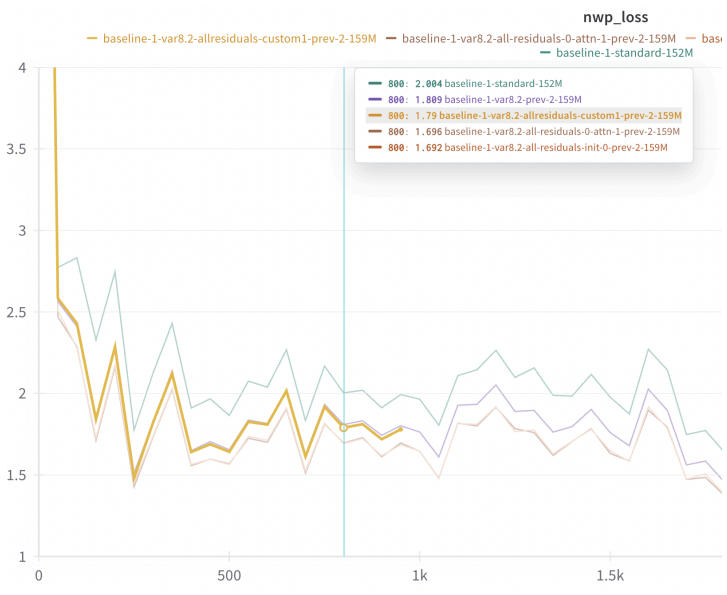
- Attention, ff are init at 1,
- identity and recurrence are init at 0
- Recurrence still weighs the skip connection, not the residual

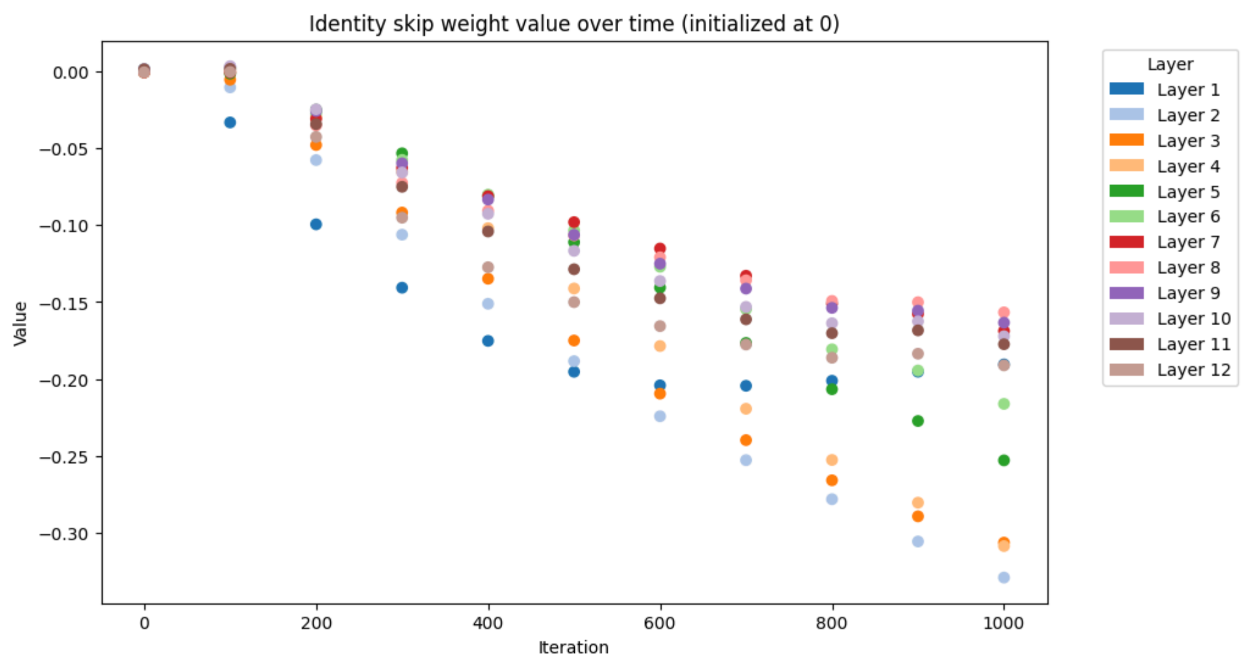
```
        r = r * self.attention_skip + x
        h = h * self.ff_skip + x # adding a ff skip
        x = self.identity_skip * x

        out = x + h + r

        return out
```

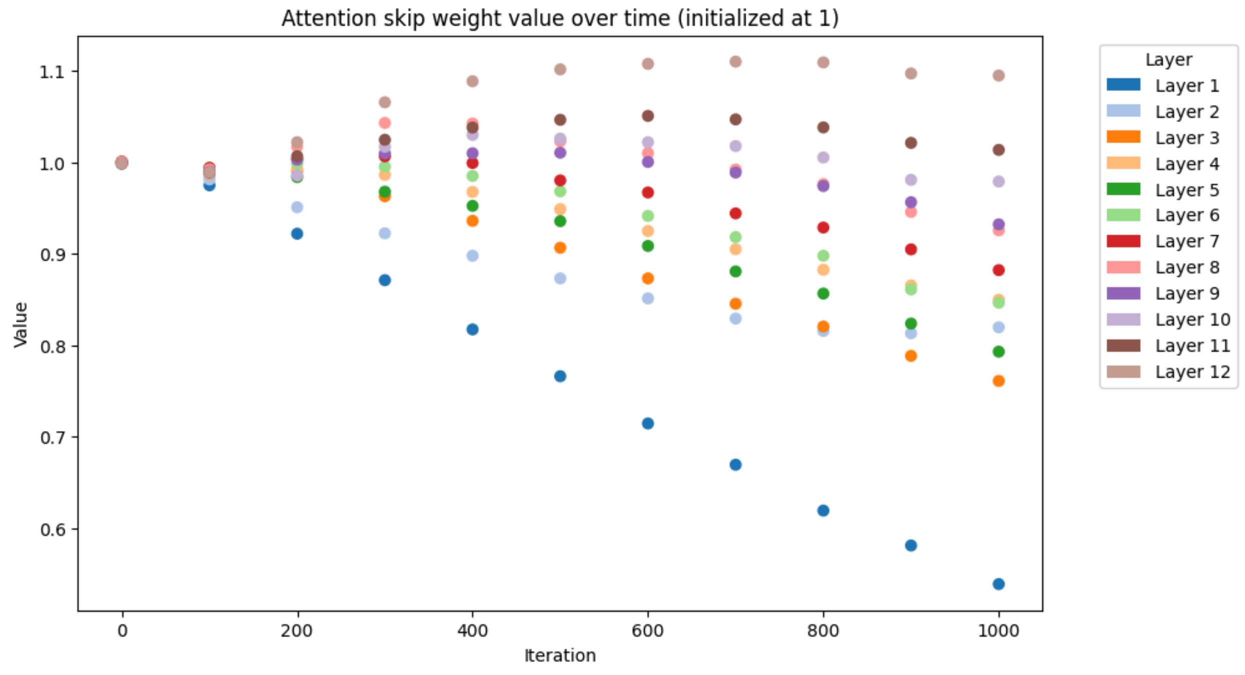
The idea is adding all of these:
Identity * identity_wieght
Attention * attention_weight
Ff * ff_weight
Recurrence * recurrence_weight - but recurrence has its own identity weight inside of ff

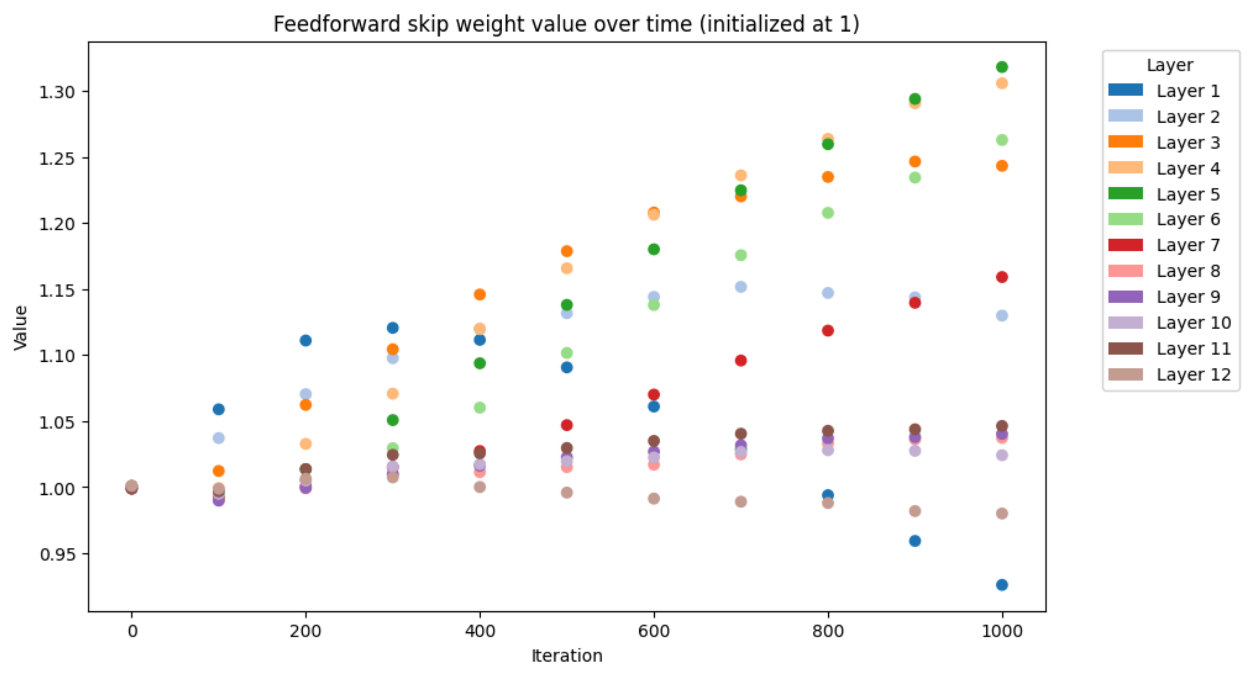
We should really parse out recurrence from ff…

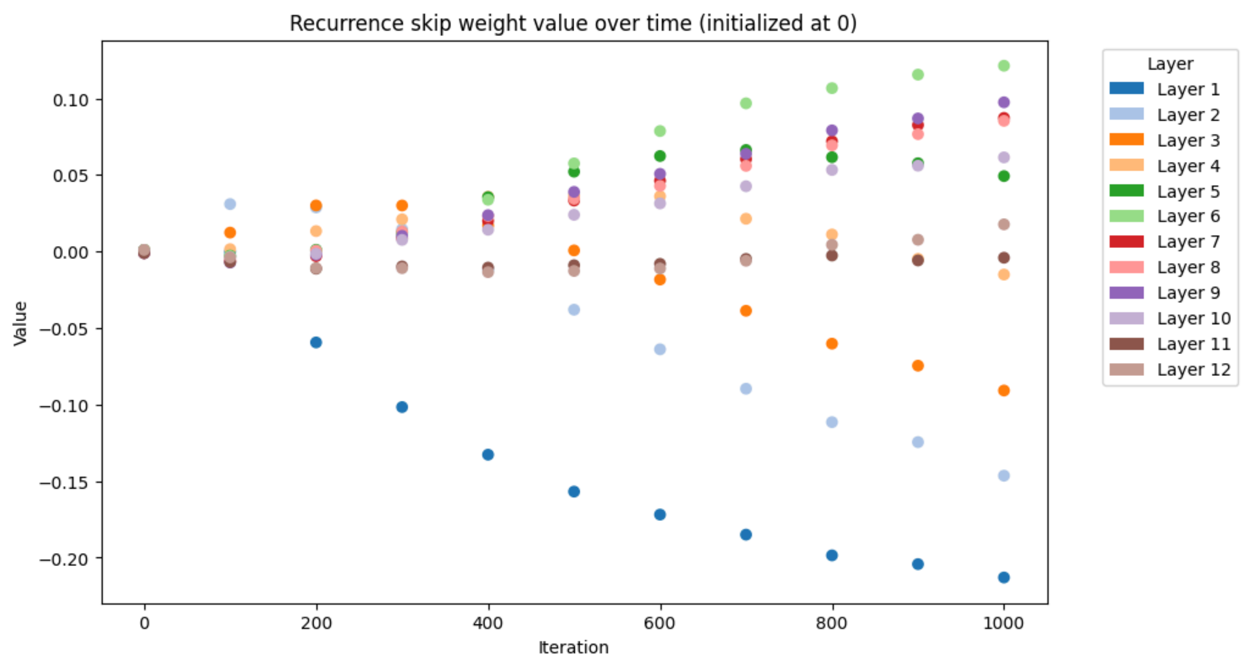
Performance is about the same as 8.2 and worse than the versions where we do simple skip connections on the recurrence and block @ 0.




Identity skip weight value over time (initialized at 0)
Value
Iteration
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
−0.30
0
200
400
600
800
1000
Layer
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer 6
Layer 7
Layer 8
Layer 9
Layer 10
Layer 11
Layer 12


Attention skip weight value over time (initialized at 1)
Value
Iteration
1.1
1.0
0.9
0.8
0.7
0.6
0
200
400
600
800
1000
Layer
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer 6
Layer 7
Layer 8
Layer 9
Layer 10
Layer 11
Layer 12





Init: Custom2 - block skip only

Now we only have a skip connection init at 0 for the transformer block, so we're isolating what this does for the model as a whole.

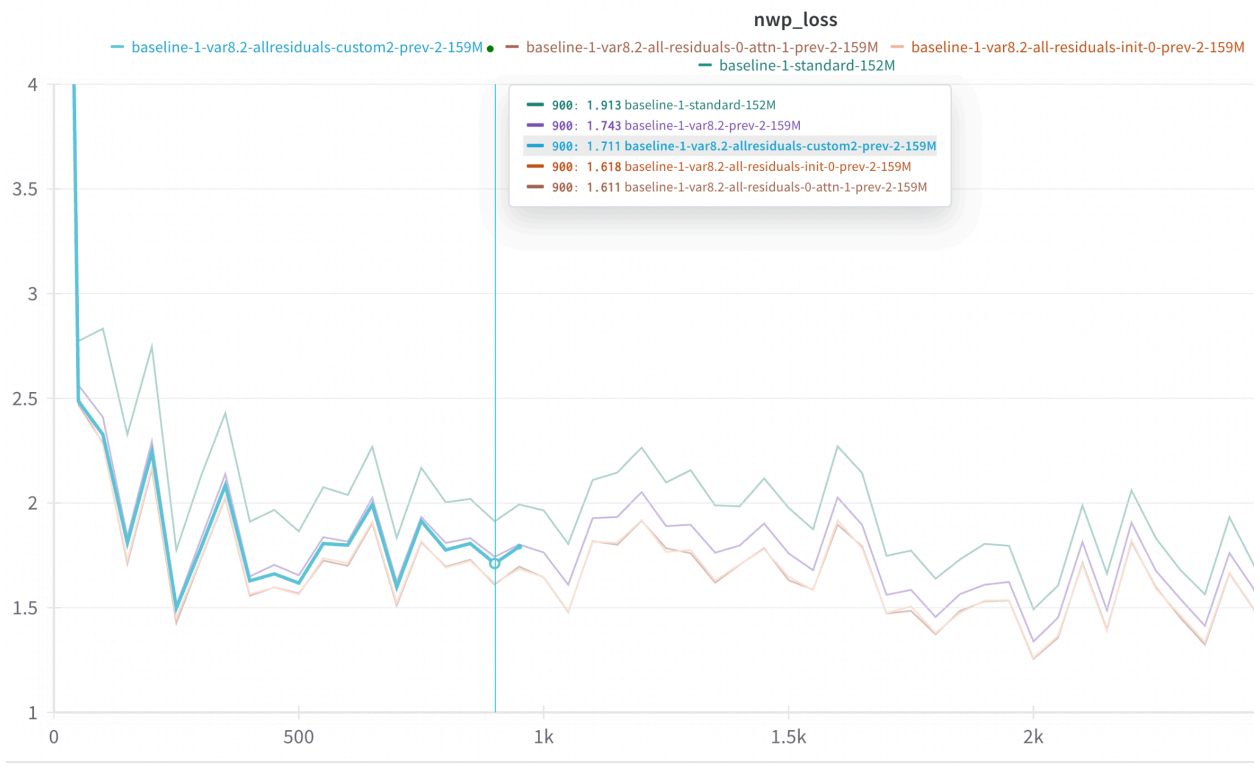
```
        r = x + self.attention.forward(self.attention_norm(x),
rotary_emb_fn)
```

```
        h = self.feed_forward.forward(self.ffn_norm(x))

        x = self.identity_skip * x

        out = x + h + r

        return out
```

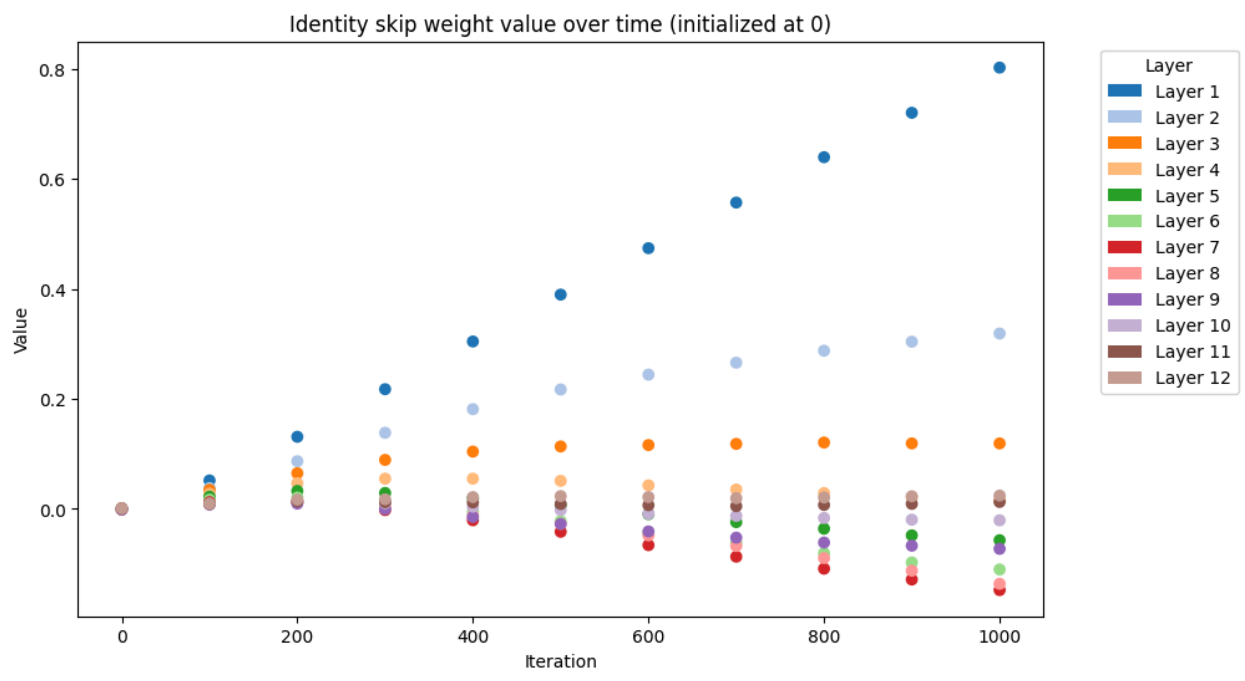
The recurrent block ahs no residual either:

```
        recurrent = gate * x + (1 - gate) * conv_over_past_examples

        #x = (self.recurrence_skip * x) +  recurrent
        x = x + recurrent
```

For performance, it's better than 8.2 plain by a bit but worse than when both the block and recurrence have 0 init skip weights:



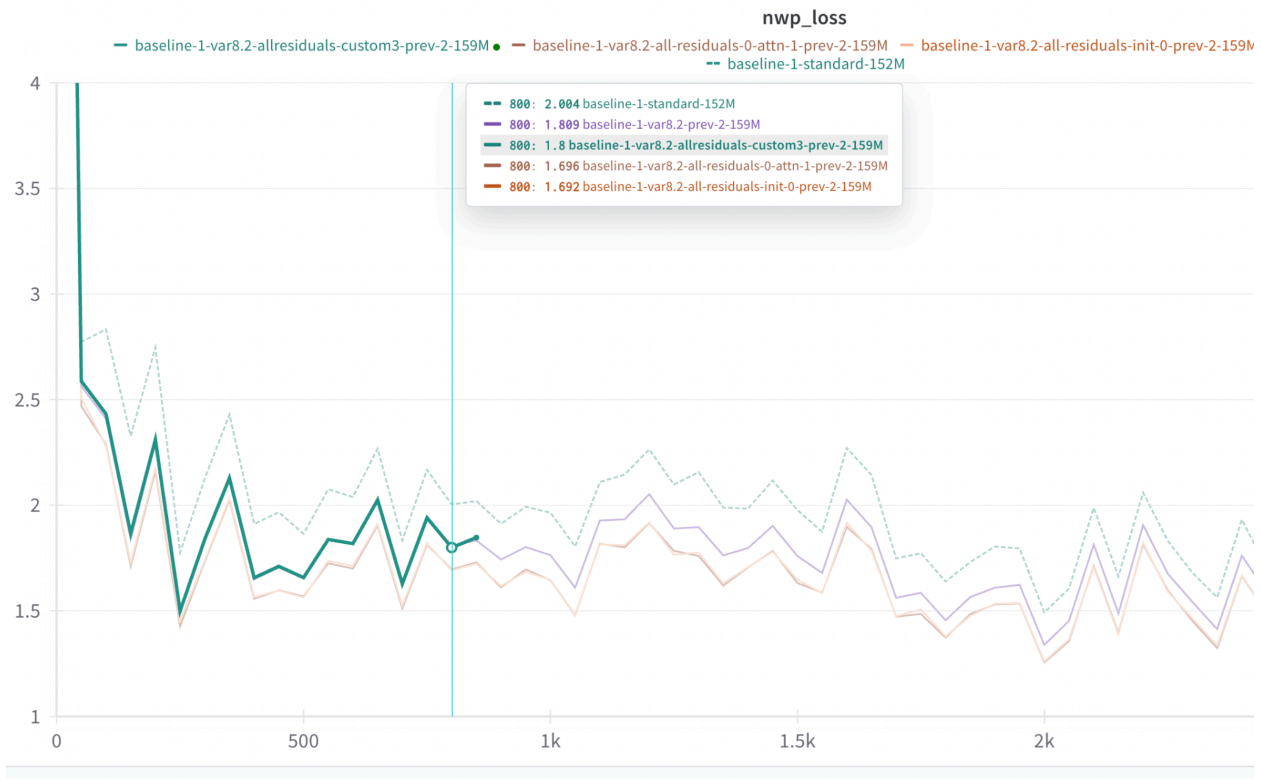
What we saw in other versions:

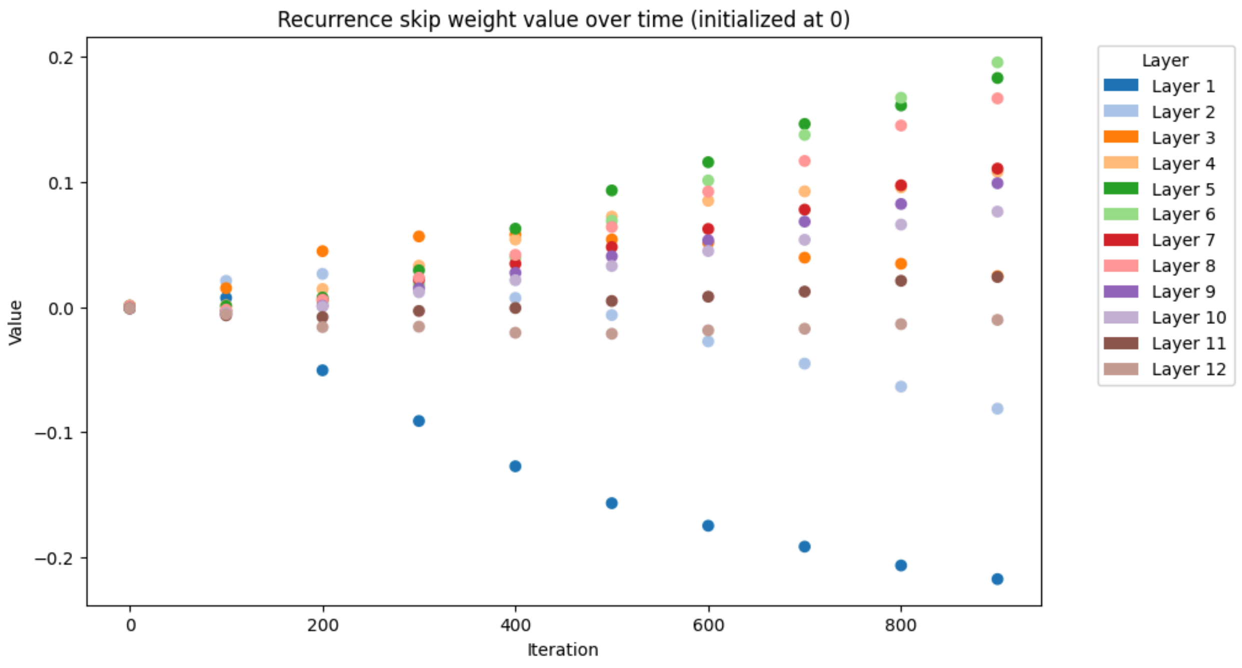


Init: Custom3 - recurrence skip only

Let’s isolate the effect of the recurrence skip weight only
[We can consider just removing it also…]

There is almost no effect here, it looks just like 8.2 normal:
Suggesting that the effect of having both skip weights is not just the additive sum of having a skip weight on either, but rather due to some interaction when both are available.
Recurrence skip alone effect = 0
Block skip alone effect = little
Recurrence and block skip together = a lot

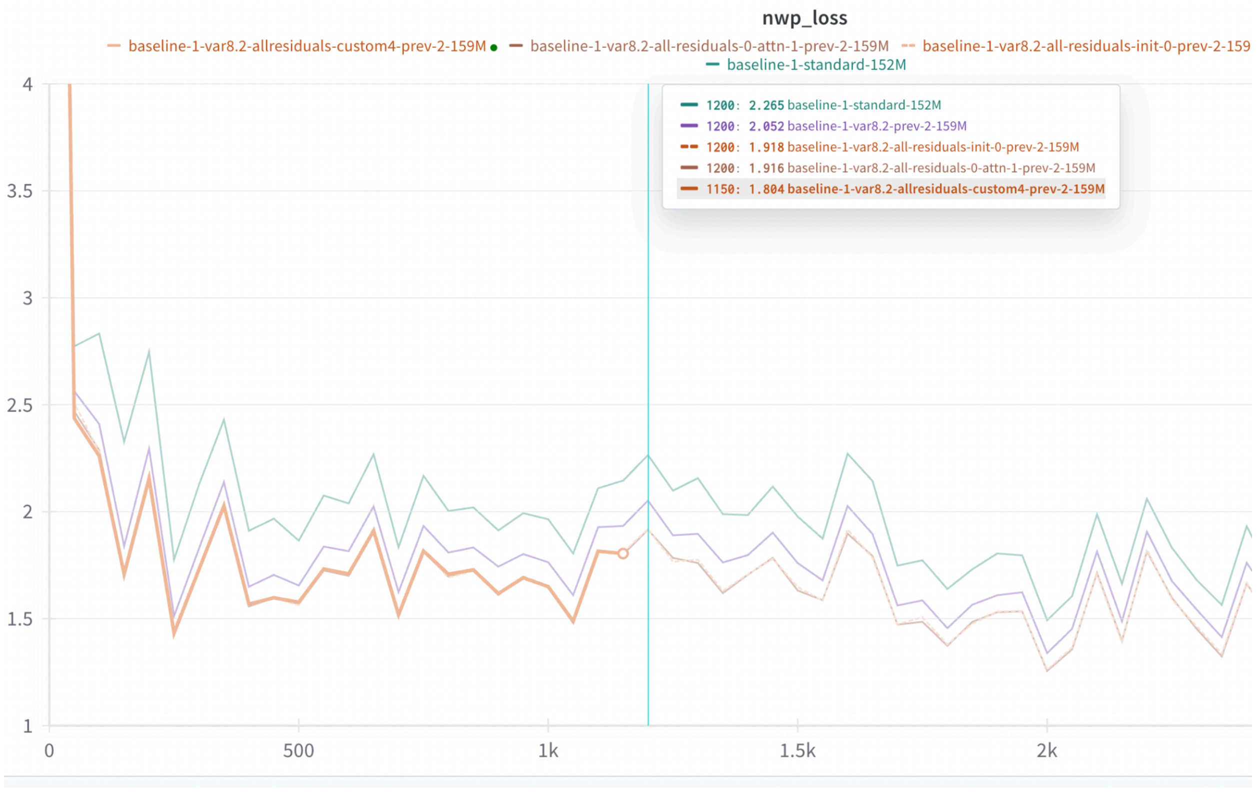




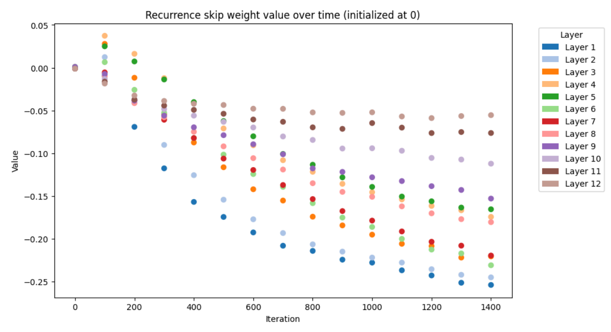
- There is the opposite effect of the identity/block skip: the early layers here subtract the skip, whereas in the identity/block skip the first layers are used heavily

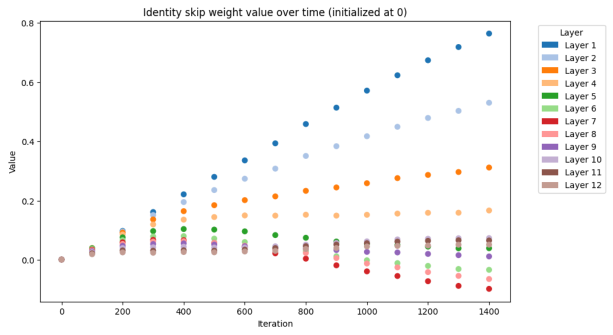
Init: Custom4 - recurrence skip and block skip, both init @ 0

Ok, here is the good performance we saw a few days ago again:



The two skips in combination are greater than the sum of their parts, which is very interesting.





- We can notice that for both of these, the curve of the first layer is much less dramatic on both vs. when compared to the skip weight in custom 2 and custom 3 (when it's one skip at a time)

Init: Custom5 - recurrence skip, block skip, and attention skip all init @ 0

This blows up. All the skip weights also lose all their patterns

Init: Custom6 - recurrence skip, block skip init @ 0. Add skip inside attention init @ 0

```
        y = att@val

        y = y.transpose(1, 2).contiguous().view(batch_size, seq_len, 
self.n_heads * self.args.head_dim) # re-assemble all head outputs side by 
side
        return self.wo(y) + (x * self.attention_skip_output)
```

I don't think this does anything different?

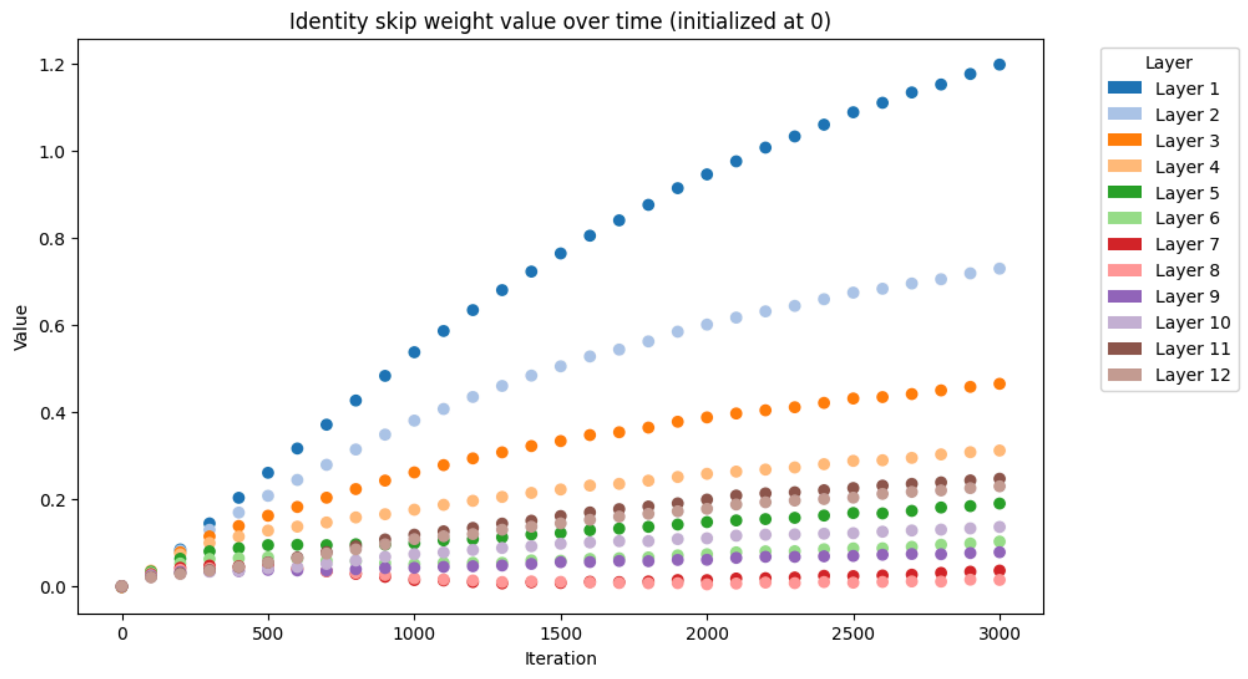
Performance is the same as Custom4.

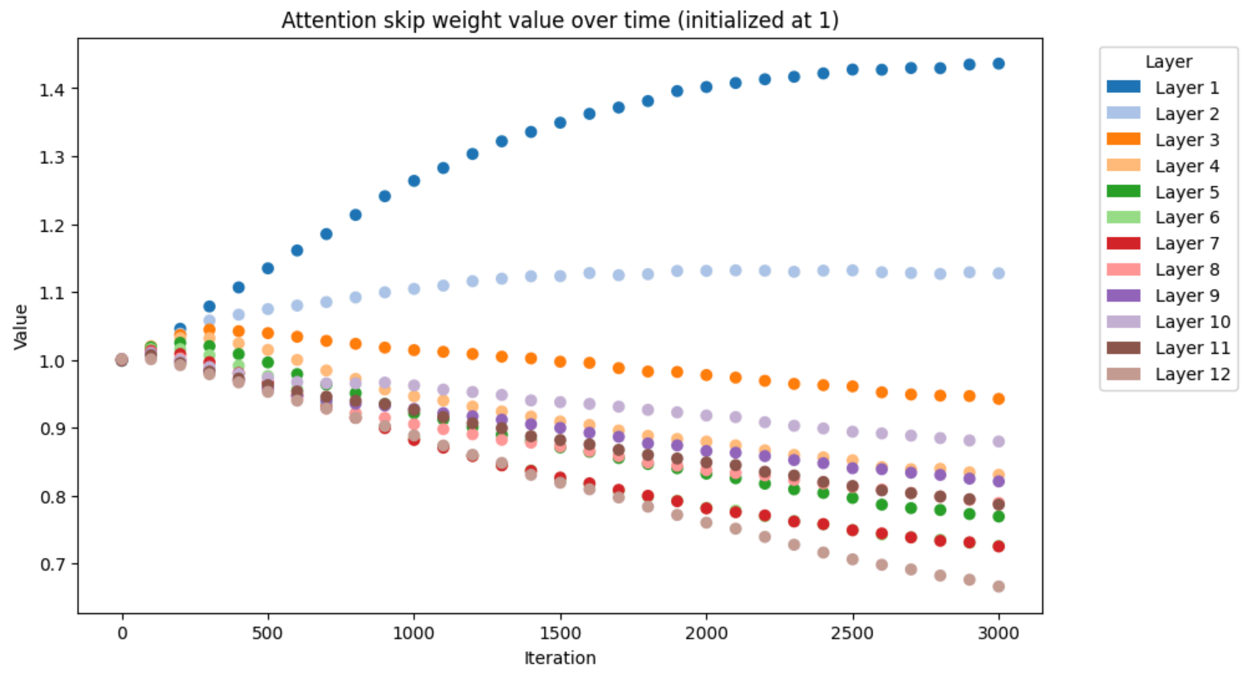
Actually the attention weight and identity weight end up being slightly different. I don't totally understand why they're different here but the same when you add it inside the transformer block…

Init: Custom6 - recurrence skip, block skip init @ 0. Add residual weight to attention inside block init @ 1

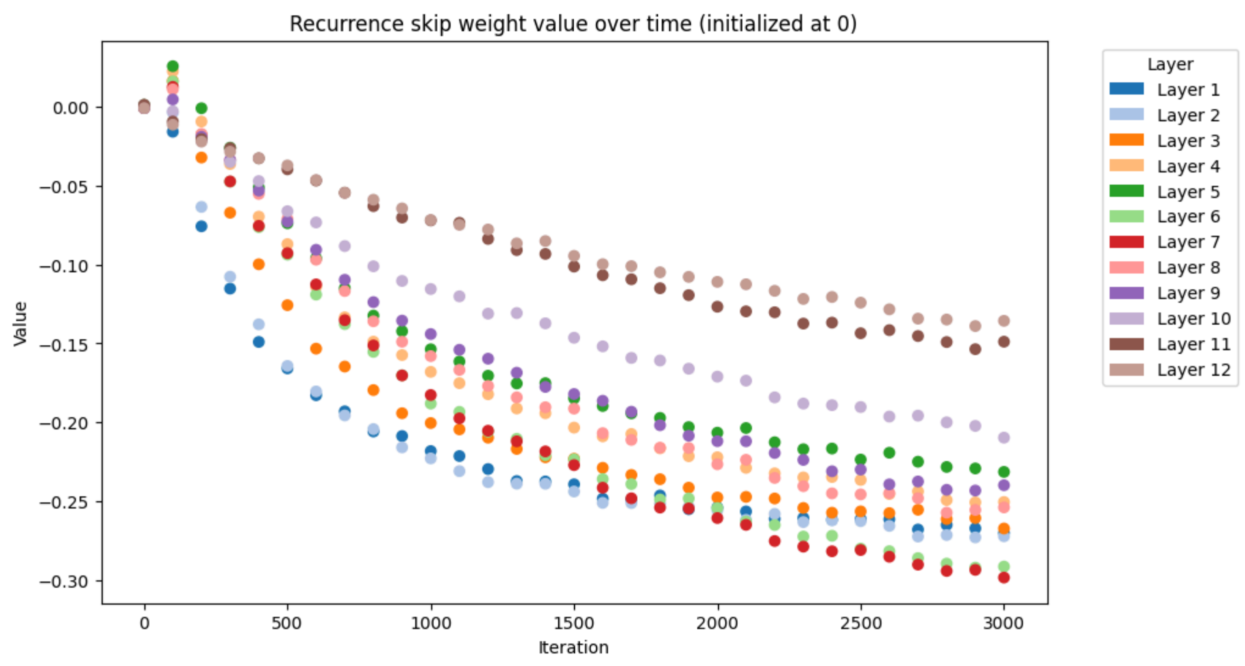
So this is just custom4, which works well, but we're adding a parametrizable weight on attention.

Performance is the same as custom4



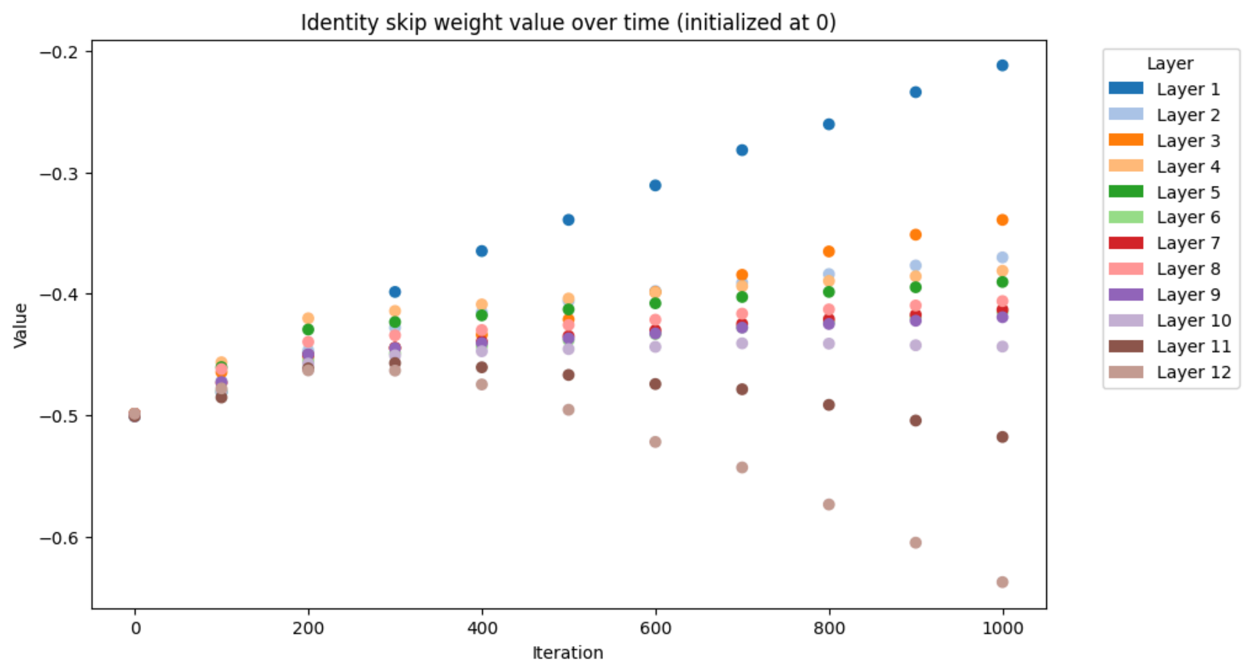


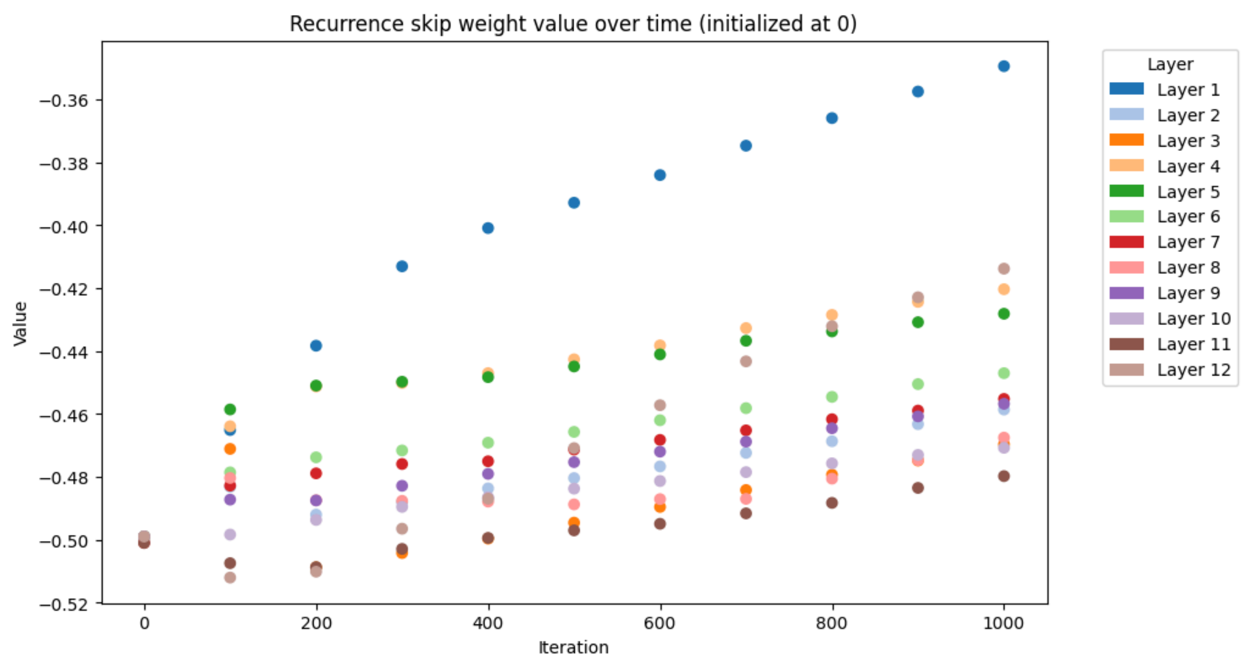
- Note that this is not for the skip weight, this is for the residual; it is a multiplier on the attention computation itself



Init: Custom7 - recurrence skip, block skip init @ -.5

Performance is bad, but the skip weights tell us something





All those values are too low for recurrence skip weight
For identity skip weight some of these are too low but some are not even low enough.

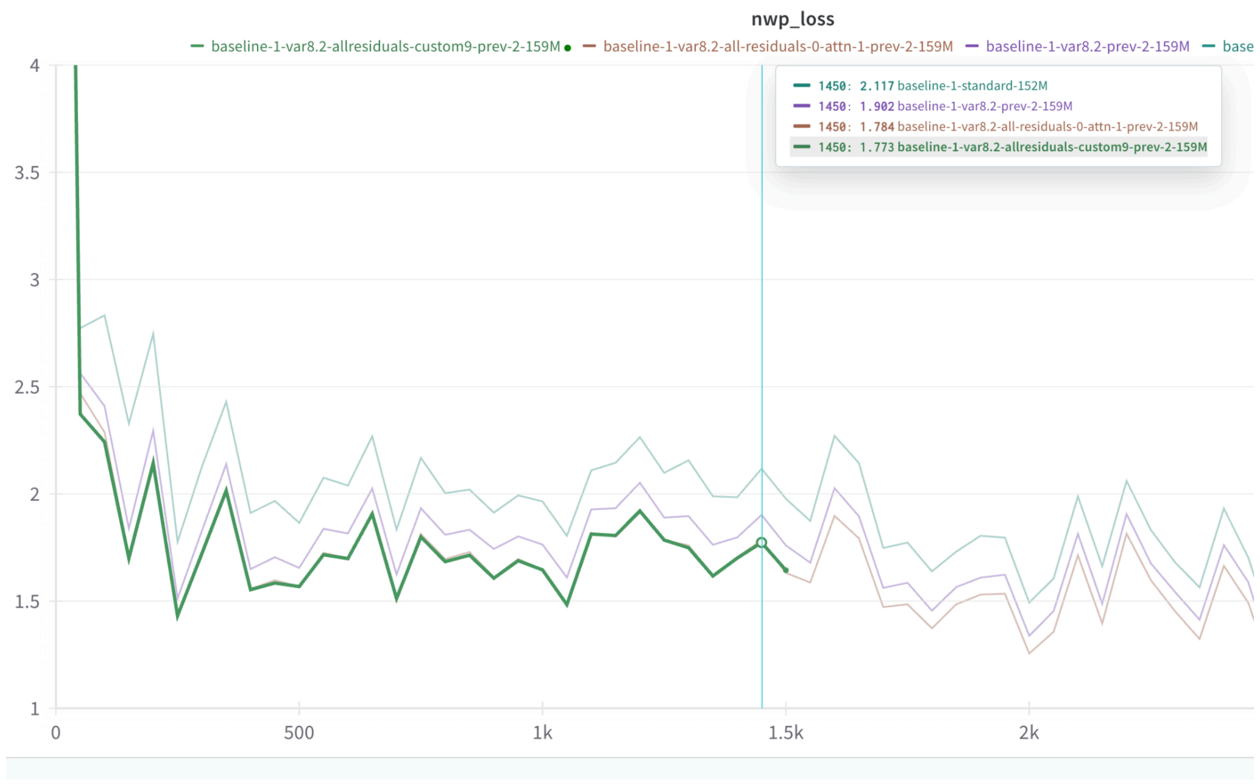
Should we consider initializing these based on layer depth?

Init: Custom8 - recurrence skip, block skip init @ 0, block skip layer 1 and 2 init at .5

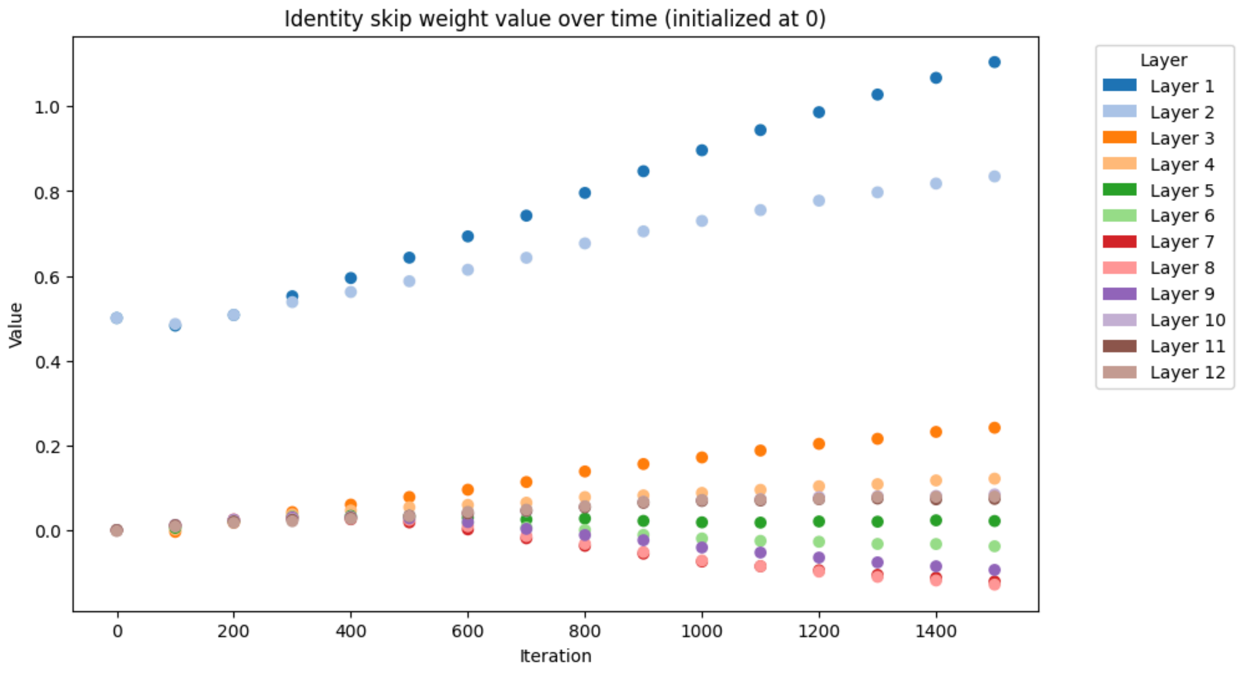
```
# with torch.no_grad():
#     model.layers[0].identity_skip += .5
#     model.layers[1].identity_skip += .5
```

Let’s boost the identity skip weight for the frist two layers since they want to pass through identity information and see what happens.

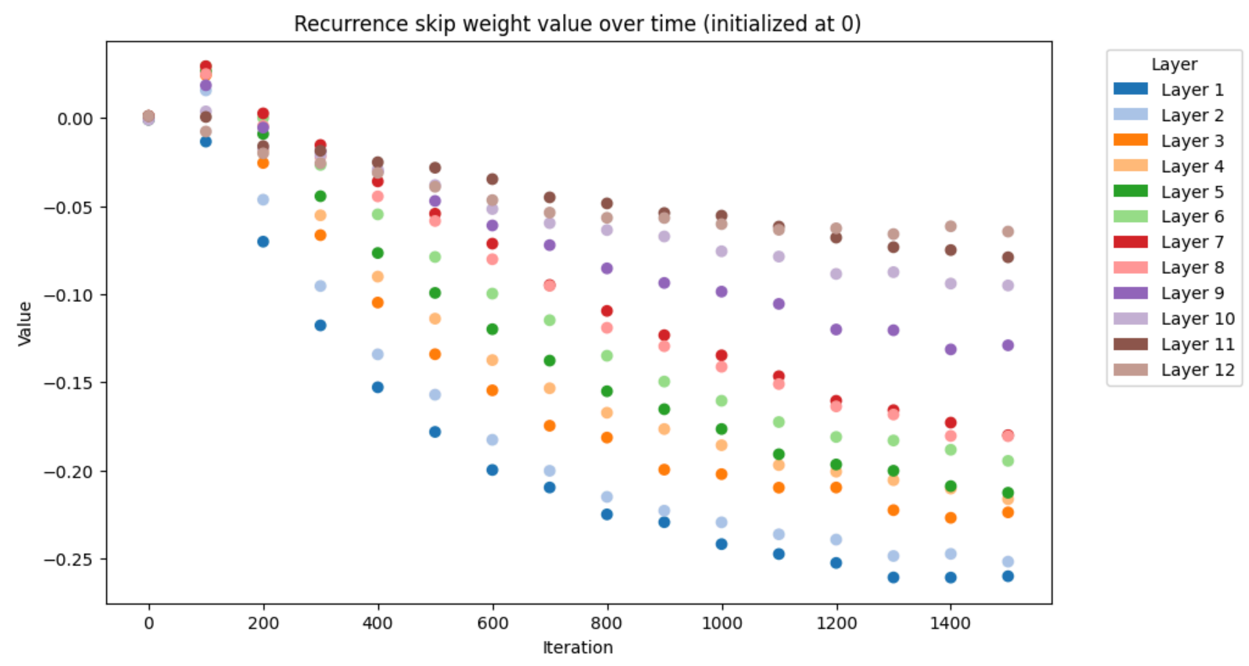
Performance is basically the same as custom4



However it does have the desired effect:



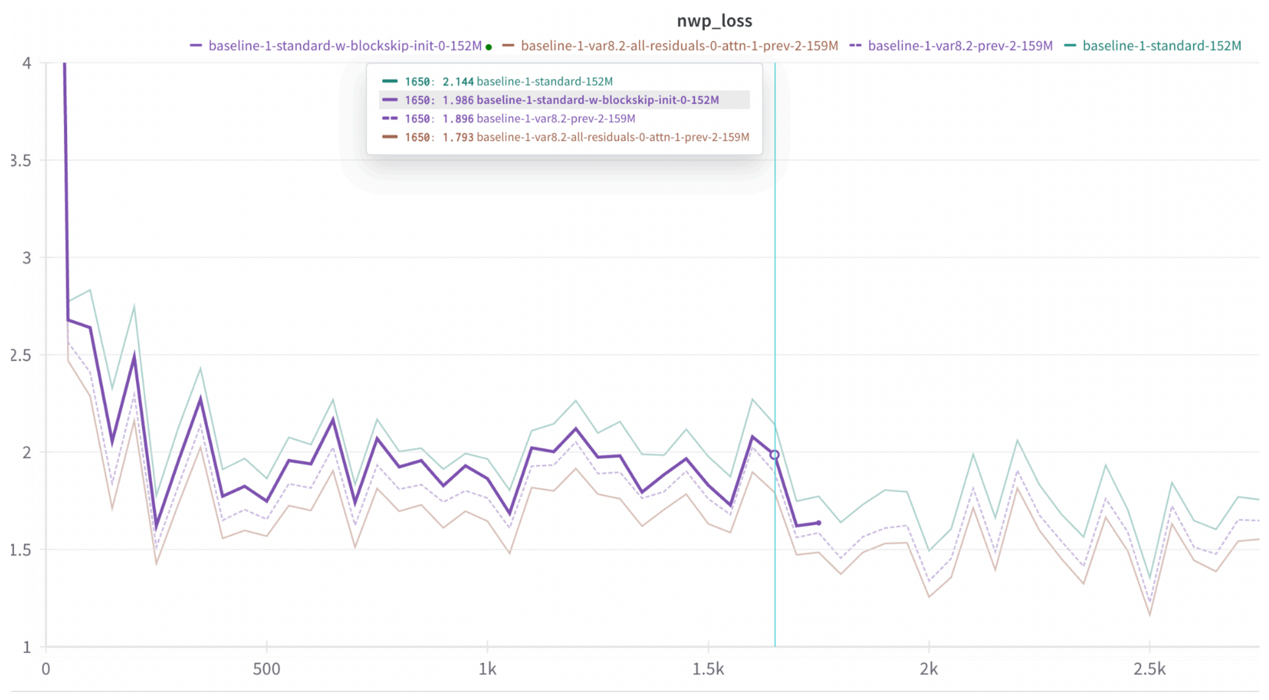
The recurrence weights, untouched, do the same thing

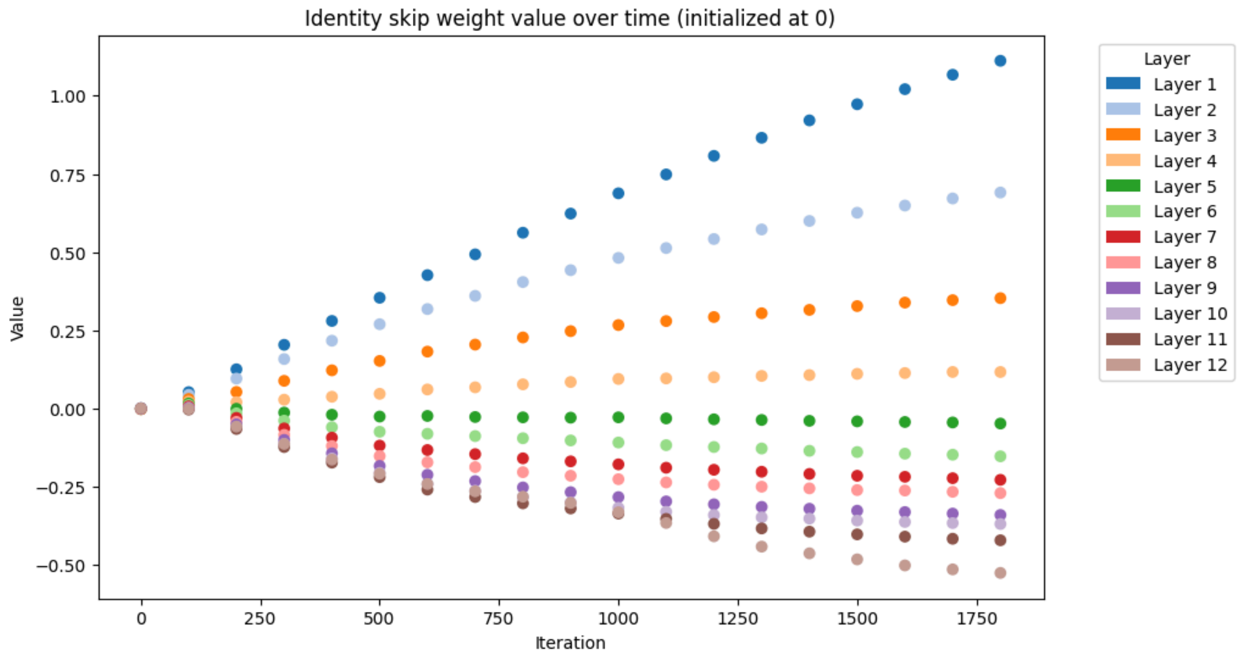


Residual baseline - init @ 0

Let's run a standard transformer with identity weight at 0

OK, this residual/skip phenomenon isnt specific to recurrence model, it's the same for vanilla transformer: Solid purple (this run) vs green (vanilla transformer)





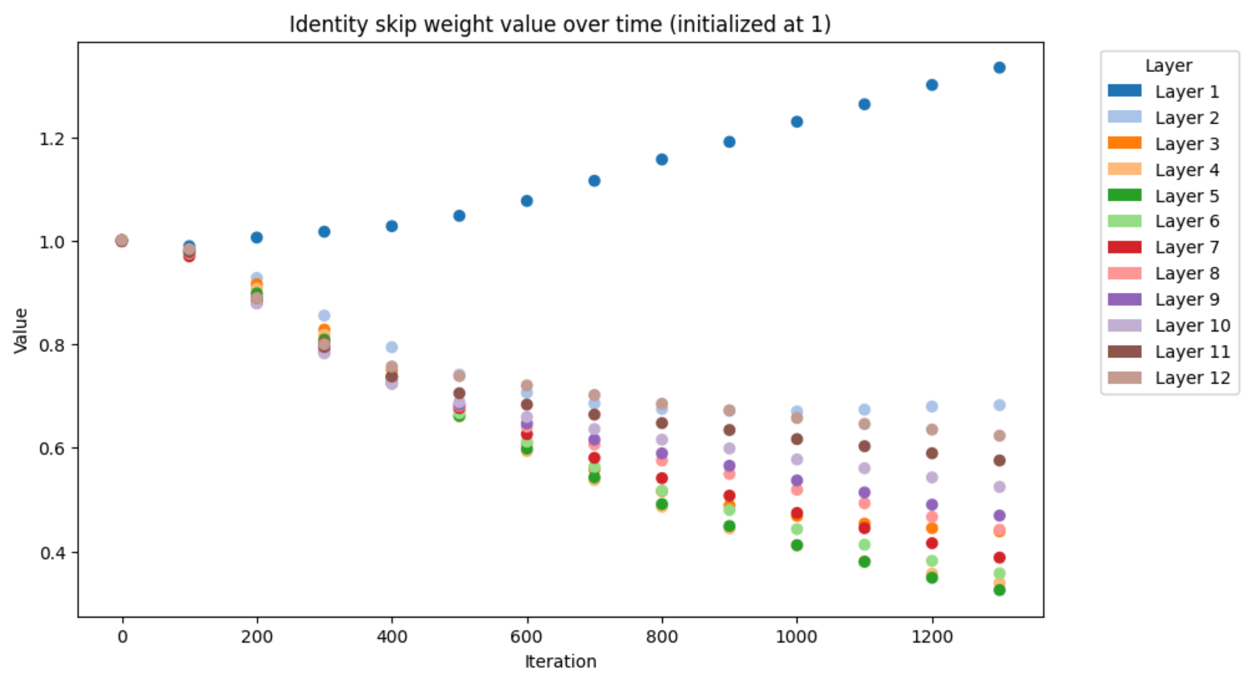
- The differences are even more exaggerated here

Residual baseline - init @ 1

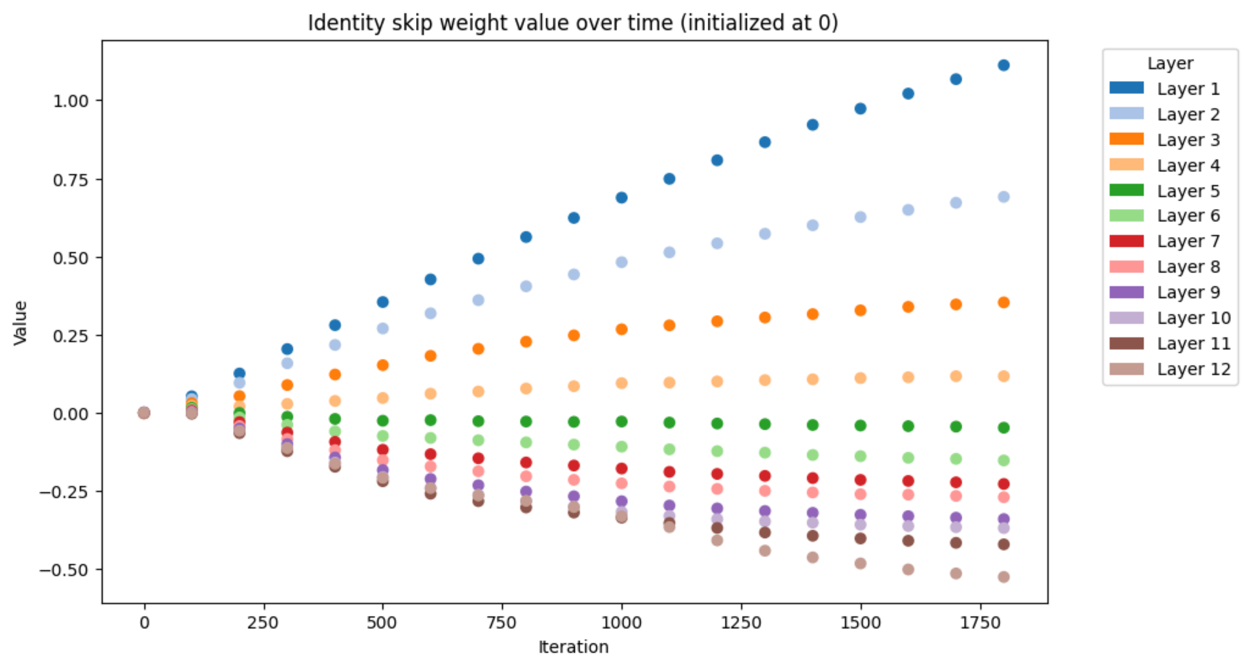
The story as I can tell so far is basically:

The skip connection is important for early layers (and redundant layers if they exist) to let the network pass the identity through. But it actually hurts middle/later layers because the residual is important, and diluting the residual with identity hurts performance. For this reason the middle layers actually want to boost their residual signal by actually **subtracting** the identity. Adding a learnable multiplicative weight to skip connection allows the network to do both. Note that the model is not setting the identity connection to 0: it is actually subtracting a portion of the input representation from the result of the residual computation, as if to erase the effect of the input.

It's quite sensitive to initialization as well. If you initialize the weight at 1.0, which makes the most sense, the model actually does slightly worse. So it seems like a trick that does nothing. But you will notice the following pattern:

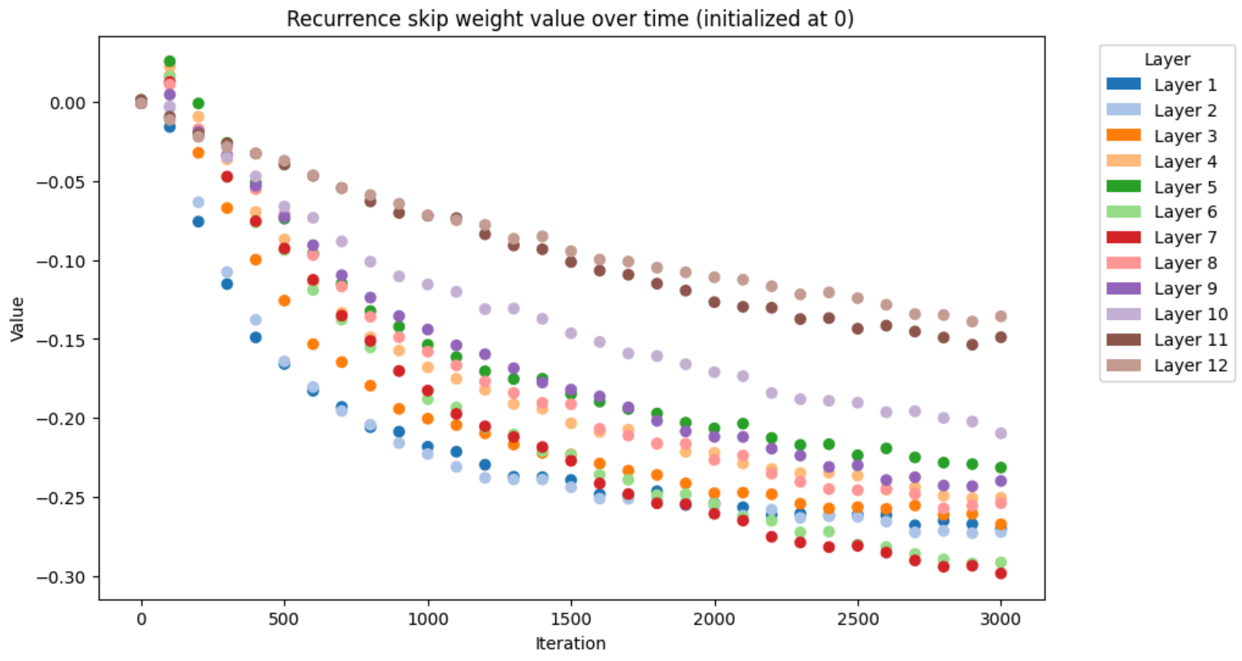


Initializing at 0 makes hte model perform considerably better, and you see the following



It's not clear why initializing at 0 allows it to perform considerably better. Perhaps the benefit of having a strong early layer identity signal is much less important than being able to subtract the identity in later layers. Initializing the early layers at higher value doesn't do much to performance save at the very start of training, so that supports this view. This is the working hypothesis.

Other residuals have even stronger preference to subtract the identity. In an experimental architecture that is a vanilla transformer which includes a style of recurrent computation between attention and feed forward at each layer, the recurrent skip connection is even more dramatic:

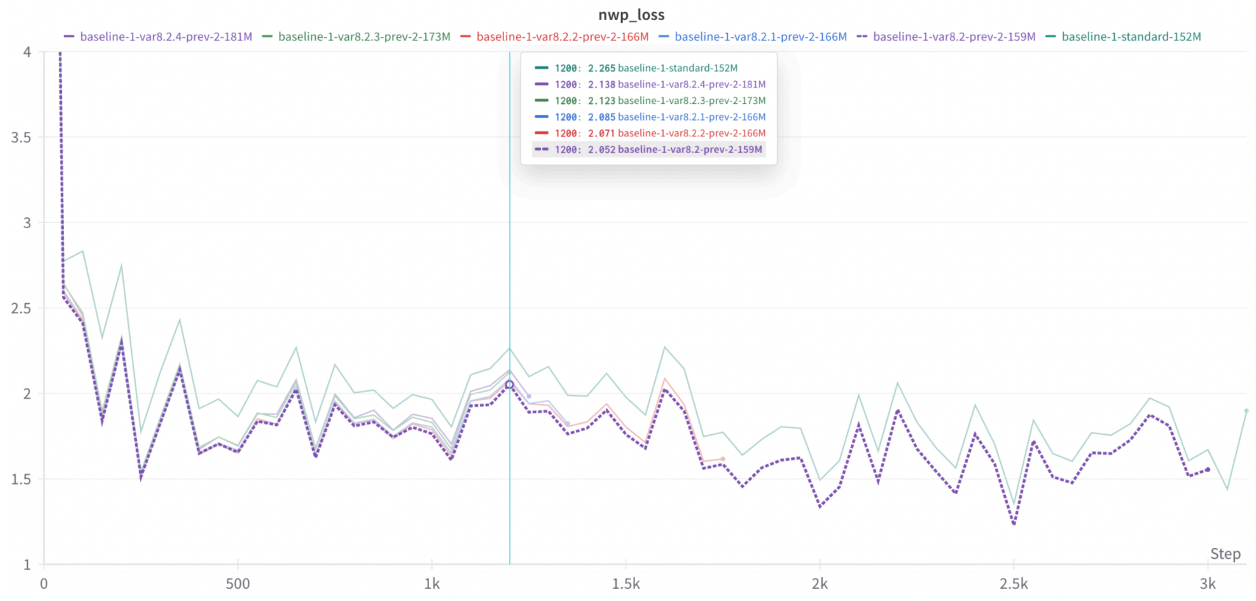


Where the early layers even more so than the later layers prefer to subtract the identity.

8.2.X

Variation 8.2.1 - 8.2.4 Performance

All somewhat worse than 8.2 original and all with higher parameter count



```
1200: 2.265 baseline-1-standard-152M
1200: 2.138 baseline-1-var8.2.4-prev-2-181M
1200: 2.123 baseline-1-var8.2.3-prev-2-173M
1200: 2.085 baseline-1-var8.2.1-prev-2-166M
1200: 2.071 baseline-1-var8.2.2-prev-2-166M
1200: 2.052 baseline-1-var8.2-prev-2-159M
```

Variation 8.2.1

```
# VARIATION 8.2.1
# Add a weight matrix to h after convolution

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.h_state = nn.Linear(args.dim, args.dim)
       self.num_previous_time_steps = 2
       self.weights = nn.Parameter(torch.ones(1, 1,
self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
       softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
       softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

       # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
       conv_over_past_examples =
torch.nn.functional.conv2d(x.unsqueeze(1),

softmax_weights,
                                                                bias=None,
                                                                stride=1,

padding=(self.num_previous_time_steps, 0))
```

```
        conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
        conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

        # tensor gate
        w_x = self.w_state(x)

        # apply h_state post convolution
        conv_over_past_examples = self.h_state(conv_over_past_examples)
        gate = torch.sigmoid(w_x)

        x = gate * x + (1 - gate) * conv_over_past_examples
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Variation 8.2.2

```
# VARIATION 8.2.2
# Add a weight matrix to h before convolution

class FeedForward(nn.Module):
    # SwiGLU
    def __init__(self, args: ModelArgs):
        super().__init__()

        self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
        self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w_state = nn.Linear(args.dim, args.dim)
        self.h_state = nn.Linear(args.dim, args.dim)
        self.num_previous_time_steps = 2
        self.weights = nn.Parameter(torch.ones(1, 1,
self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))

    def forward(self, x) -> torch.Tensor:
        batch_size, seq_len = x.shape[0], x.shape[1]

        weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
```

```
        softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
        softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

        # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
        # apply h_state before convolution
        conv_over_past_examples =
torch.nn.functional.conv2d(self.h_state(x).unsqueeze(1),

softmax_weights,
                                                              bias=None,
                                                              stride=1,

padding=(self.num_previous_time_steps, 0))


        conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
        conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

        # tensor gate
        w_x = self.w_state(x)

        gate = torch.sigmoid(w_x)

        x = gate * x + (1 - gate) * conv_over_past_examples
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

As the best performing 8.2.x variation (though not better than plain 8.2), also tried scaling down this h_state(x) by sqrt(args.dim) but performance was worse.

Variation 8.2.3

```
# VARIATION 8.2.3
# Make the previous state input dependent (selective re Mamba)
# Concatenate h and x and linear transform before gating

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.concat_out = nn.Linear(args.dim * 2, args.dim)
       self.num_previous_time_steps = 2
       self.weights = nn.Parameter(torch.ones(1, 1,
self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
       softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
       softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

       # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
       conv_over_past_examples =
torch.nn.functional.conv2d(x.unsqueeze(1),

softmax_weights,
                                                            bias=None,
                                                            stride=1,

padding=(self.num_previous_time_steps, 0))



       conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
       conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

       # tensor gate
```

```
        w_x = self.w_state(x)
        # concatenate (b t 2d) and transform to (b t d)
        conv_over_past_examples = self.concat_out(torch.cat([x,
conv_over_past_examples], dim=-1))

        gate = torch.sigmoid(w_x)

        x = gate * x + (1 - gate) * conv_over_past_examples
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Variation 8.2.4

```
# VARIATION 8.2.4
# Make the previous state input dependent (selective re Mamba)
# Add a weight matrix to h before convolution
# Concatenate h and x and linear transform before gating

class FeedForward(nn.Module):
    # SwiGLU
    def __init__(self, args: ModelArgs):
        super().__init__()

        self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
        self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
        self.w_state = nn.Linear(args.dim, args.dim)
        self.h_state = nn.Linear(args.dim, args.dim)
        self.concat_out = nn.Linear(args.dim * 2, args.dim)
        self.num_previous_time_steps = 2
        self.weights = nn.Parameter(torch.ones(1, 1,
self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))

    def forward(self, x) -> torch.Tensor:
        batch_size, seq_len = x.shape[0], x.shape[1]

        weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
        softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
        softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

        # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
        conv_over_past_examples =
torch.nn.functional.conv2d(x.unsqueeze(1),

softmax_weights,
                                                                    bias=None,
                                                                    stride=1,

padding=(self.num_previous_time_steps, 0))


        conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
        conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

        # tensor gate
        w_x = self.w_state(x)
        # apply h_state post convolution
        conv_over_past_examples = self.h_state(conv_over_past_examples)
        # concatenate (b t 2d) and transform to (b t d)
        conv_over_past_examples = self.concat_out(torch.cat([x,
conv_over_past_examples], dim=-1))

        gate = torch.sigmoid(w_x)

        x = gate * x + (1 - gate) * conv_over_past_examples
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Variation 8.2.1

Variation 8.3

We could also have larger convolution kernels, so this kernel is essentially providing a single scalar (softmaxed) weight for each example, whereas we could provide a larger convolutional kernel, or a (num_past_examples, dimension) convolution.

You need to adjust num_groups as well as previous time steps here.

```
# VARIATION 8.3
# Generalization of 8.2 to groups:
# Instead of a kernel of size (num_previous_time_steps, 1),
# we effectively create kernels of size (num_previous_time_steps, n) where
n is some divisor of dim
# The result is that we can effectively have a richer convolution: kernel
weights for each dimension, or dimension / m where m is some divisor

# Note this requires moving around our input in order to treat each
dimension like its own in_channel


class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.num_previous_time_steps = 2
       self.num_groups = 1
       self.weights = nn.Parameter(torch.ones(args.dim, int(args.dim /
self.num_groups), self.num_previous_time_steps, 1) * (1.0 /
self.num_previous_time_steps))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
       softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
       softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

       conv_over_past_examples =
torch.nn.functional.conv2d(x.permute(0,2,1).unsqueeze(-1), # reshaping the
input

softmax_weights,
                                                            bias=None,
                                                            stride=1,

padding=(self.num_previous_time_steps, 0),

groups=self.num_groups)


       conv_over_past_examples =
conv_over_past_examples.squeeze(-1).permute(0,2,1) # returning to shape
       conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

       # tensor gate
       gate = torch.sigmoid(self.w_state(x))

       x = gate * x + (1 - gate) * conv_over_past_examples
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))

# RuntimeError: Given groups=1, weight of size [768, 1, 2, 1], expected
input[16, 768, 512, 1] to have 1 channels, but got 768 channels instead
#Given groups=1, weight of size [768, 8, 2, 1], expected input[16, 768,
512, 1] to have 8 channels, but got 768 channels instead


# TOY EXAMPLE OF GENERALIZED GROUP CONVOLUTION OVER PAST TIME STEPS

# num_groups is the important variable, informing what size convolution we
have
# num_groups = dim means weights for each dimension

# batch_size = 3
# seq_len = 10
# num_previous_examples = 4
# dim = 8
# num_groups = 8
# assert dim % num_groups == 0, "dim not divislbe by num_groups"


# orig_data = torch.arange(batch_size * seq_len *
dim).float().view(batch_size, seq_len, dim)

# # we're treating each dimension as an in_channel of (row, 1)
# data = orig_data.permute(0,2,1).unsqueeze(-1) # (minibatch, in_channel,
H, W)


# # (out_channel, in_channel/groups, H, W)
# kernel = torch.ones(dim, int(dim/num_groups), num_previous_examples, 1)


# output = F.conv2d(data,
#                   kernel,
#                   padding=(num_previous_examples, 0),
#                   groups=num_groups)

# output = output.squeeze(-1).permute(0, 2, 1)
```

Here we’re interested in playing around with group size.

Part of the motivation in fact is that having more capacity to weigh the previous states will make increasing the number of previous states more effective

If num_groups = args.dim then we’re basically creating convolution filter size (num_previous_time_steps, args.dim) that passes over timesteps and outputs a (1, args.dim) output. So instead of weighting each dimension of the previous examples with a single scalar, we are weighting it with args.dim scalars.

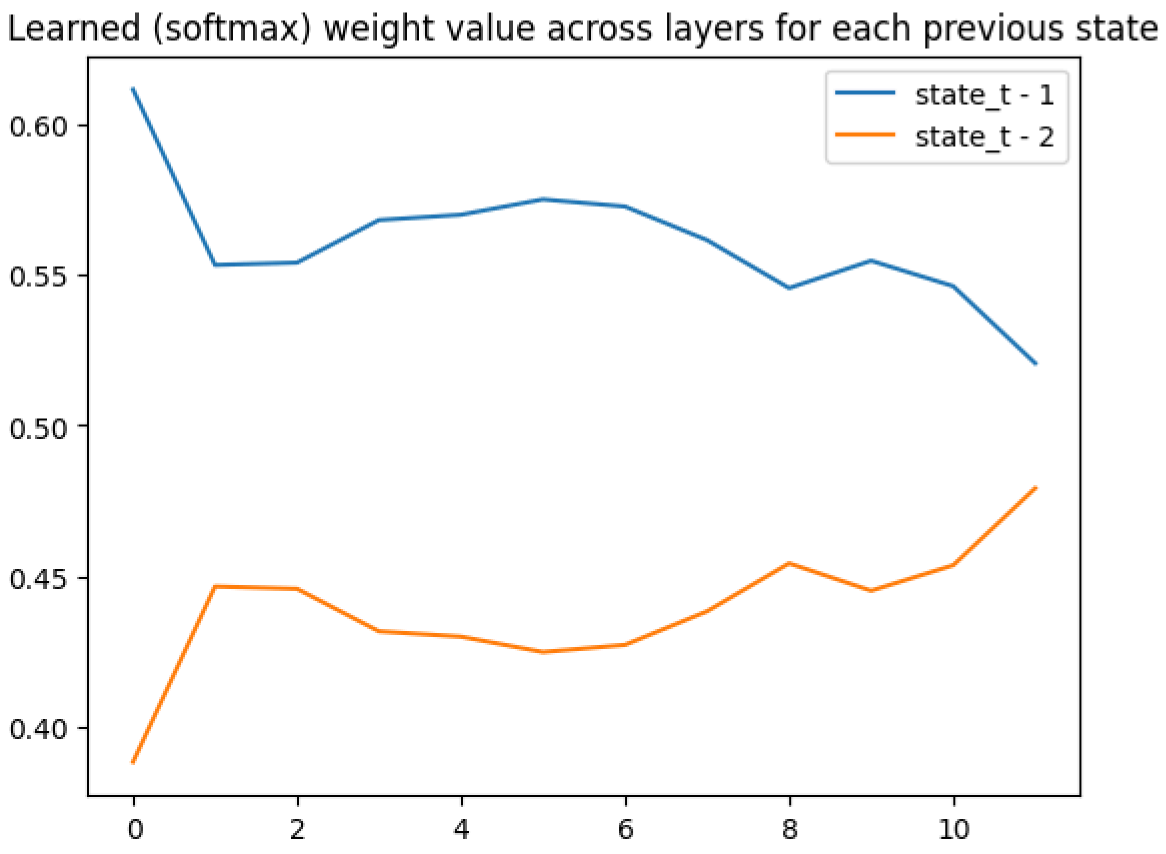
num_groups = 1 doesn’t work well (same as baseline) - we’re creating kernels of (args.dim, args.dim) which is enormous. We’re primarily interested in:

num_groups = args.dim: creates kernels of (1, args.dim)

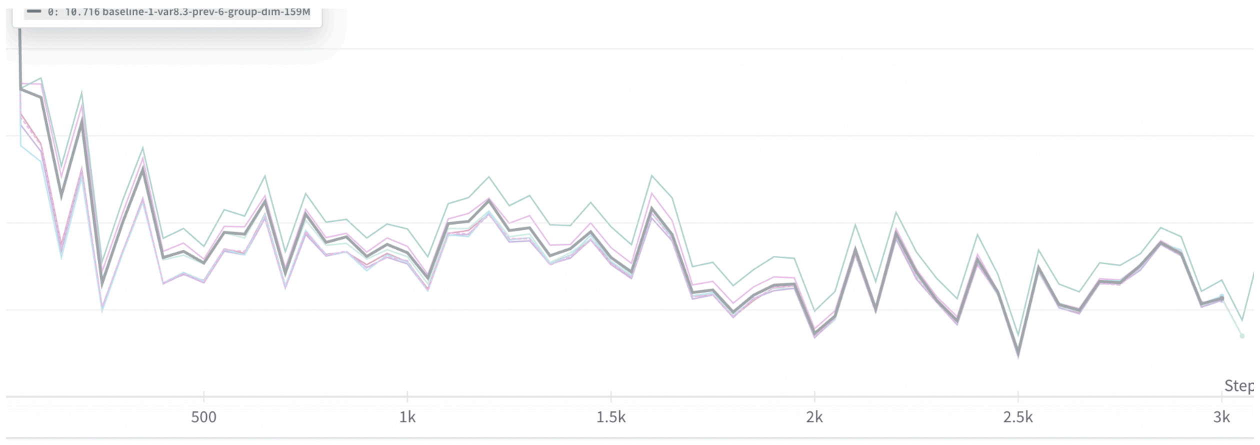
But we can increase the size and see what happens.

With num_groups = args.dim, performance is pretty good. It starts out in the middle of the pack and ends up being second best.

Plot: after 3000 iters. Take the mean across dimension of each layer per previous state

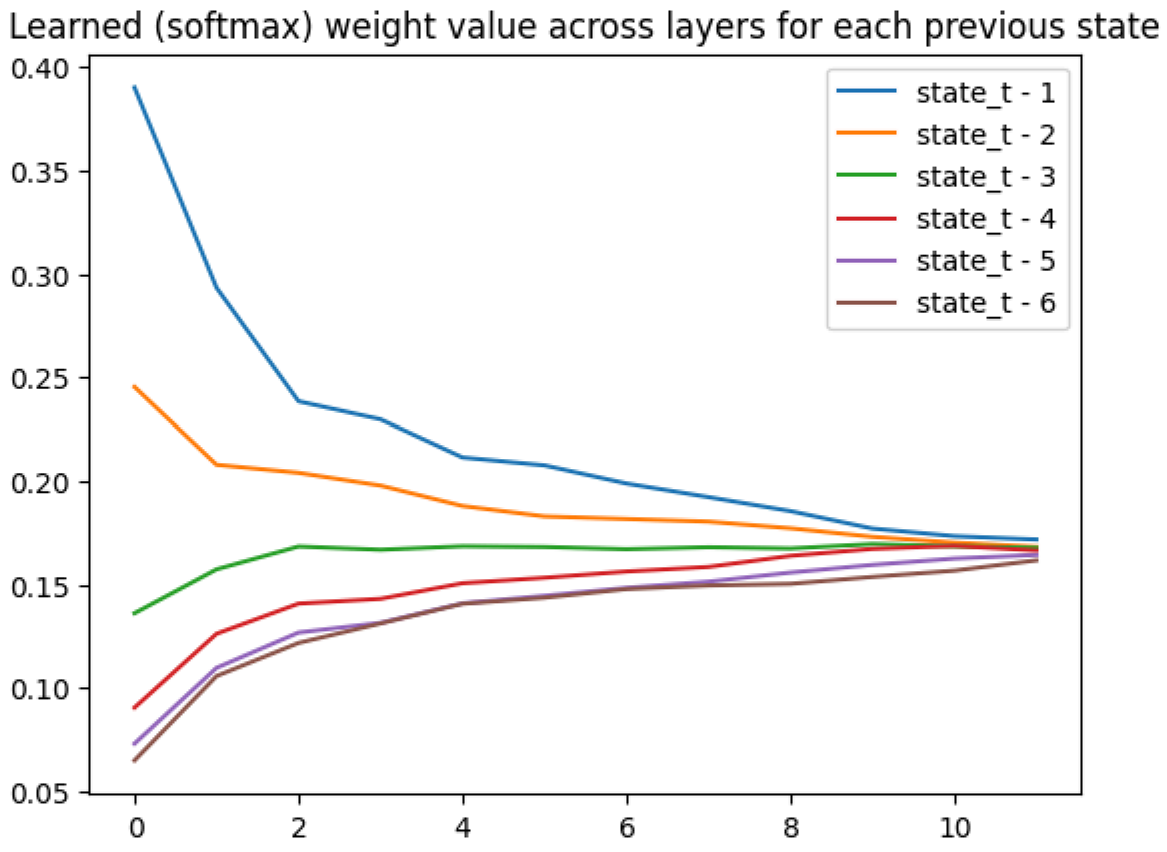


For 6 previous states, this does fine



Although basically everything is converging: variations are converging together, and baseline is also converging with variations though at a slower rate. Or (we're just eyeballing the graph above): all these variations are better than baseline and they're all just converging together.

3000 iters,the previous states all converge over layers towards even distribution (6 states, approximately .16)



Variation 9

```
# VARIATION 9

# h = A(x)h
# Hardcoded weight matrix A_state of (t d)

class FeedForward(nn.Module):
   # SwiGLU
   def __init__(self, args: ModelArgs):
       super().__init__()

       self.w1 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w2 = nn.Linear(args.hidden_dim, args.dim, bias=False)
       self.w3 = nn.Linear(args.dim, args.hidden_dim, bias=False)
       self.w_state = nn.Linear(args.dim, args.dim)
       self.A_state = nn.Linear(512, args.dim)
       self.concat_out = nn.Linear(args.dim * 2, args.dim)
       self.num_previous_time_steps = 2
       self.weights = nn.Parameter(torch.ones(1, 1,
self.num_previous_time_steps, 1) * (1.0 / self.num_previous_time_steps))

   def forward(self, x) -> torch.Tensor:
       batch_size, seq_len = x.shape[0], x.shape[1]

       weights = self.weights.view(self.weights.size(0), -1)  # Flatten
the weights
       softmax_weights = F.softmax(weights, dim=1)  # Apply softmax
       softmax_weights = softmax_weights.view_as(self.weights)  # Reshape
back to original shape

       # Note for F.conv2d unlike nn.conv2d, you are required to pass in
the weights:
       conv_over_past_examples =
torch.nn.functional.conv2d(x.unsqueeze(1),

softmax_weights,
                                                            bias=None,
                                                            stride=1,
padding=(self.num_previous_time_steps, 0))


       conv_over_past_examples = conv_over_past_examples.squeeze(1) #
remove dimension from output, (b 1 t d) -> (b t d)
       conv_over_past_examples = torch.split(conv_over_past_examples,
seq_len, dim=-2)[0] # take the first seq_len examples

       # input modifies weight matrix that modifies state
       # h @ (x.T @ A)
       inter = self.A_state(x.transpose(-1,-2)) # b d d
       conv_over_past_examples = conv_over_past_examples @ inter # b t d

       # tensor gate
       w_x = self.w_state(x)
       gate = torch.sigmoid(w_x)

       x = gate * x + (1 - gate) * conv_over_past_examples
       return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))
```

Loosely incorporating ideas from

https://arxiv.org/pdf/2404.08819.pdf
"The Illusion of State in State-Space Models"

First try NaNs

Debugging

The first idea is that there are too many matmuls blowing up the numbers

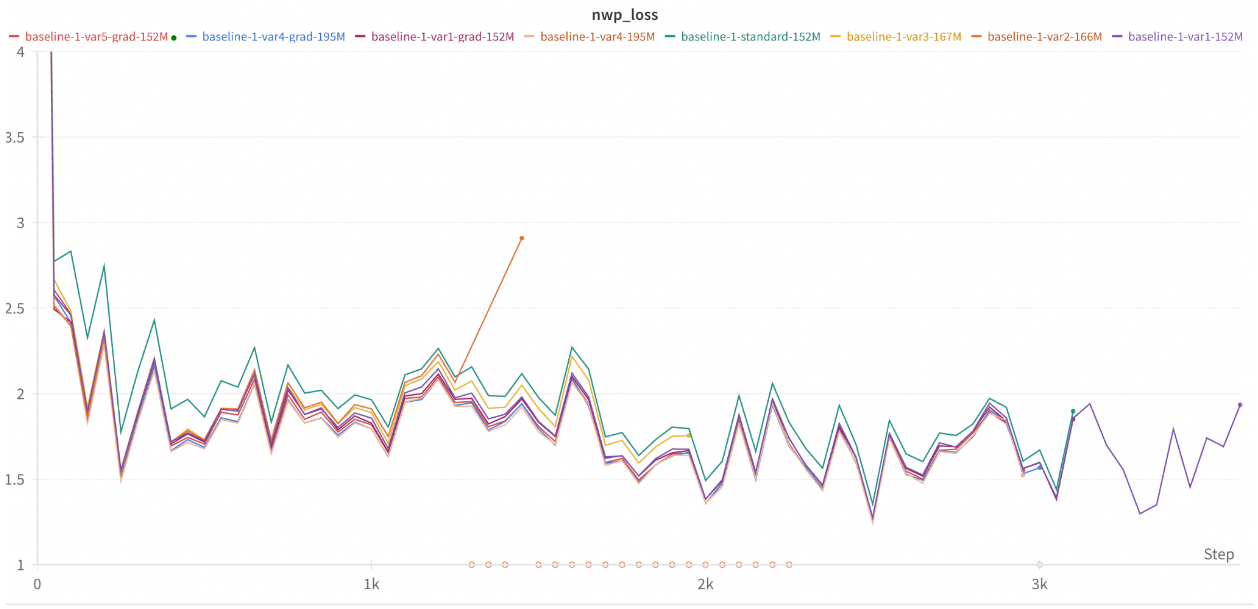
```
        inter = self.A_state(x.transpose(-1,-2)) # b d d
        conv_over_past_examples = conv_over_past_examples @ inter # b t d
```

So if we divide each of these terms by sqrt(args.dim) the model runs. Performance isn't great. The remaining variations would be to just try norming only the first of these, then only the second of these.

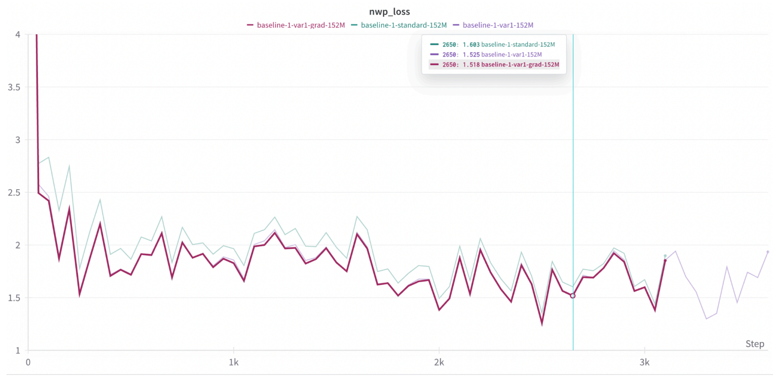
Results

Day 1 results / initial

All variations so far

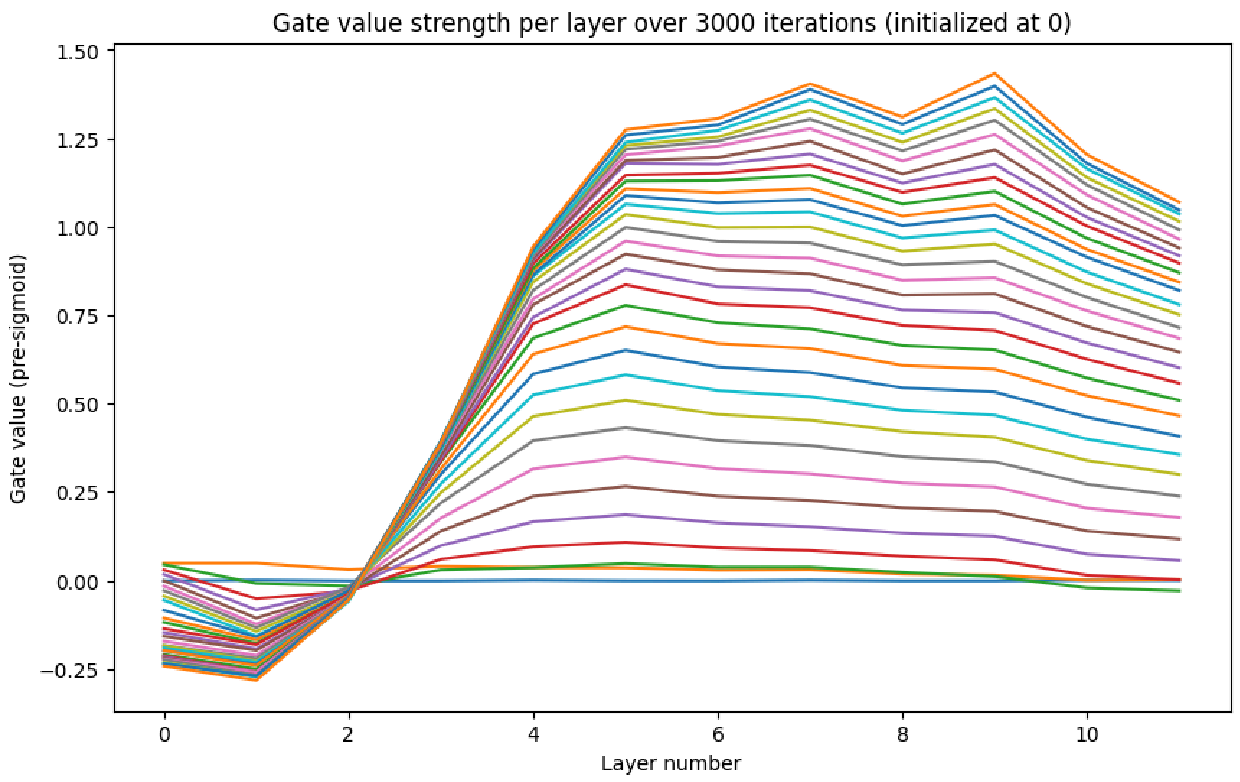


Takeaways:

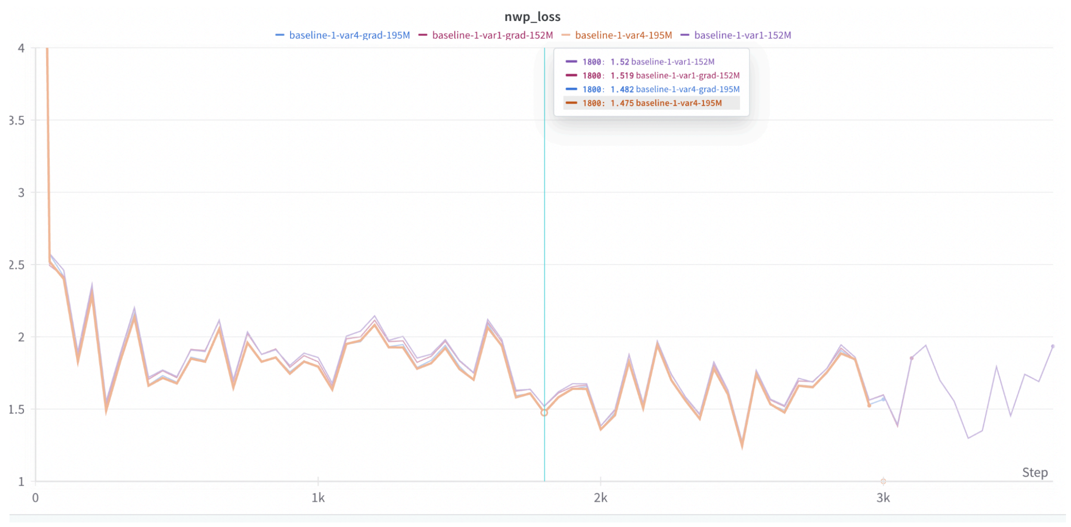
- Variation 1 is the most promising and should be tested at larger scales. It simply gives the model the choice, during feedforward layer, to incorporate through weighted (gate) addition the hidden representation of the previous time step (previous example)
    - Simple implementation
    - No noticeable slowdown (there's an extra matmul)
    - No extra parameters (1 gate parameter per layer)
    - Significantly lower loss (~.1)
    - 

2650: 1.603 baseline-1-standard-152M
2650: 1.525 baseline-1-var1-152M
2650: 1.518 baseline-1-var1-grad-152M

- Models tend to actually favor the data from the previous example in early layers, and in later layers favor the current example (see graphs of gate weights), e.g. from variation 1:

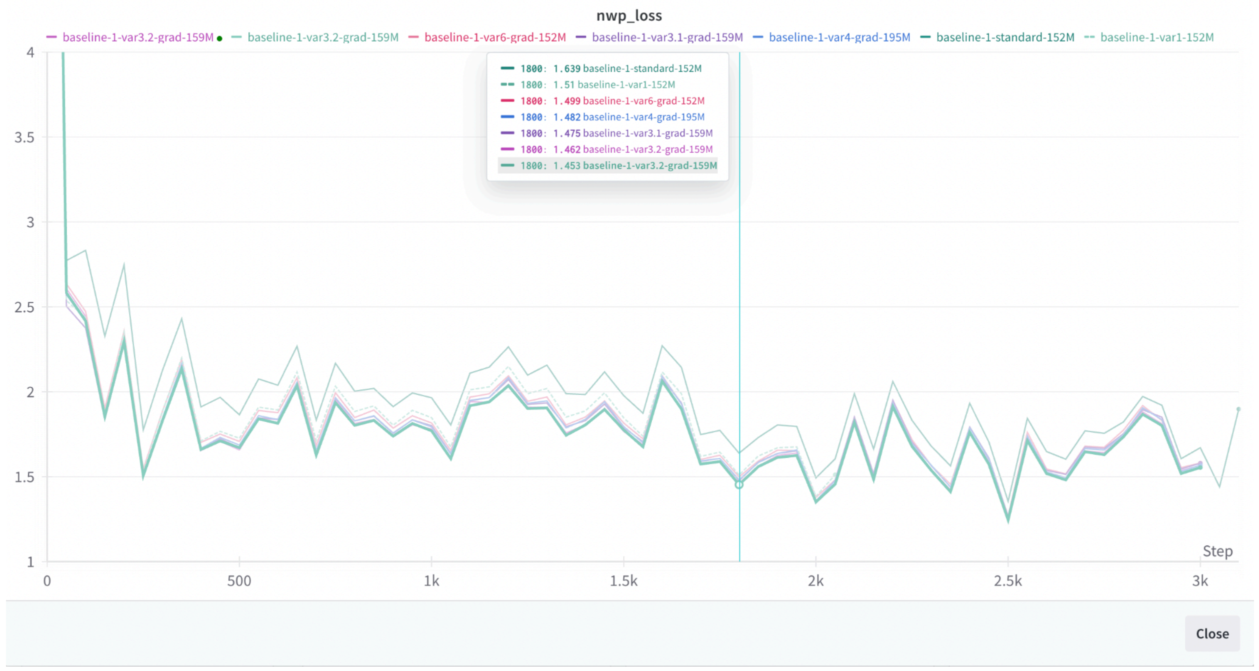


-
- Grad version of creating the preivous_state tensor doesn't seem to make a difference. We can create a matrix of the previous examples either in a way that breaks or doesn't break backprop down through the matrix. There doesn't seem to be a noticeable difference, for a given variation, when including or excluding this.
- Having the model look at additional previous examples (variation 5) doesn't seem to help performance
- Variation 2 Nans eventually, but prior to that performance looked slightly worse than variation 1
- Variation 3 is slightly better than baseline but worse than Variation 1
- Variation 4 (basically GRU translated to this data format) outperforms variation 1 (and baseline) by a small amount, but at the cost of significantly more parameters (195M vs 152M)
    - TODO: parameter-parity benchmark: test Variation 4 against a baseline model of 195M parameters



-

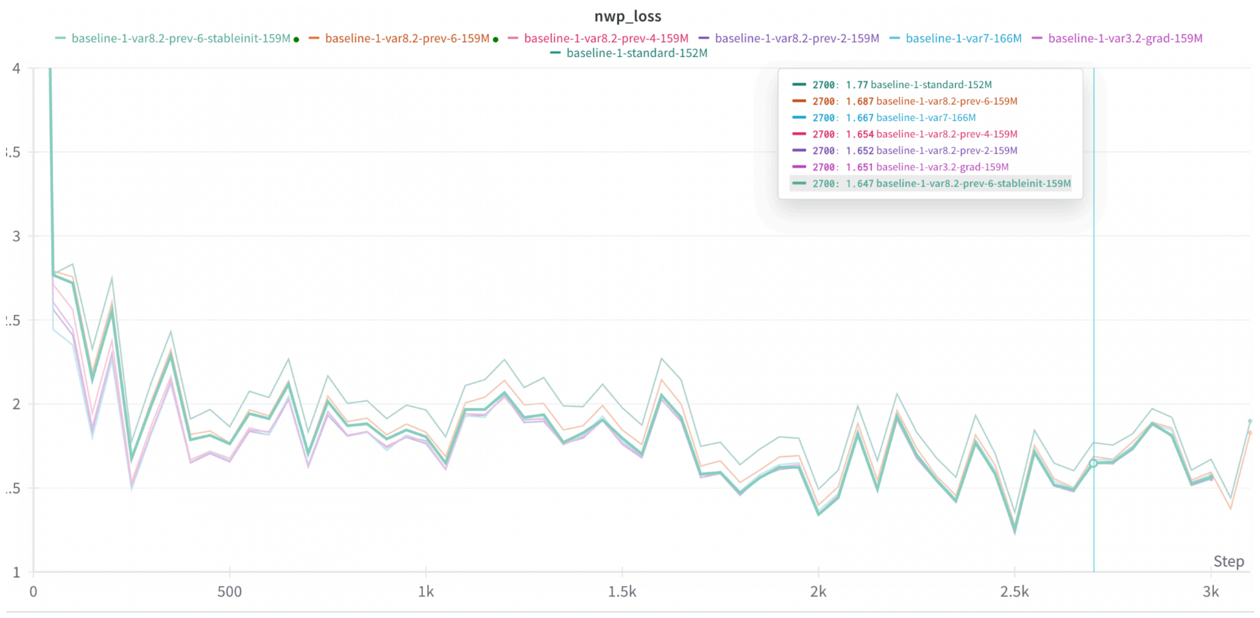
Day 2 results



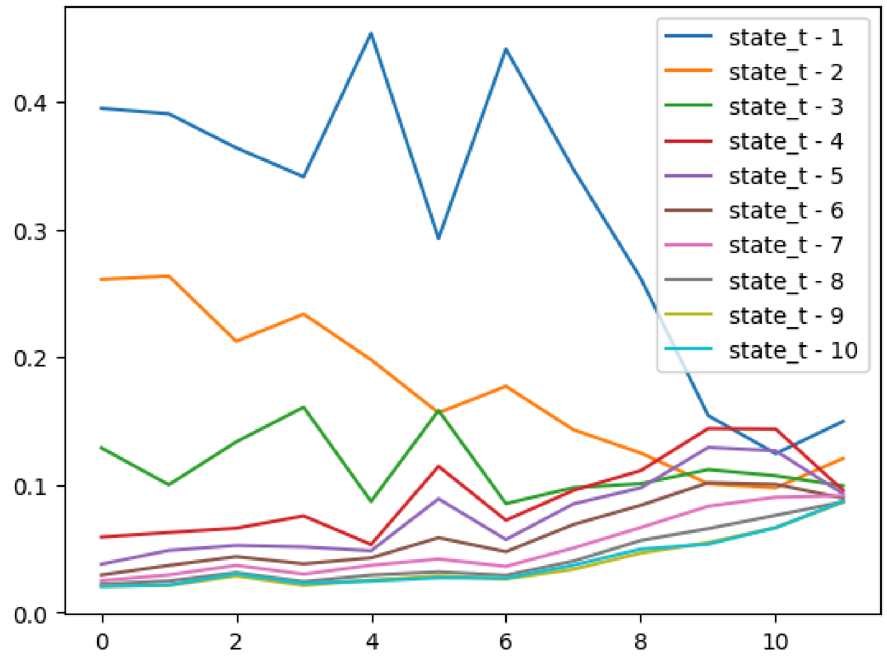
- 3.1, 3.2 added. Both are quite good with 3.2 being consistently very good, significantly below baseline and much below others. This very simply adds a gate that is not a single parameterized scalar but a) data dependent b) weight matrix. (Mamba inspired). 3.2 expands this to the previous two examples and shows improvement over 3.1, which only looks at the previous state. Overall extremely simple and effective

Day 3 results

- 7, 8, 8.1, 8.2 added.
- 7 is Griffin-like, though it doesn't implement the full lambda weights (we basically just borrow the gating structure) and our RG-LRU blocks aren't in the same place in the architecture. This version does well, though it NaNs after a while. ADding a norm after RG-LRU and before the FF stabilizes the training. The norm version and non-norm version perform slightly differently, but I'm not sure it's not just noise. Also useless to rigorously test at this small of a scale
- 8 variations are interesting. The idea is basically like version 3, where we're doing a version of convolution, and this just translates this into convolution operation so that we can expand our previous state information as many time steps back as we want (rather than hand coding out each one with a downshift matrix).
    - Plain convolution doesn't work well
    - Nor does adding a nonlinearity post convolution
    - However softmaxing some parameterized weights before passing them in to be used as convolution kernel weights works well - equivalent ot hwat we had in 3.2 which was the winner. So the weights need to be either nonlinearized, sum to 1, or both.
    - 8.2 reflects these changes and seems like the best
    - It's also important to intialize these kernel weights evenly. Looking at the weight values post training shows distribution fo weight over past tokens looks sensical / well-ordered if we initialize them sensibly, but if you initialize them randomly you get strange looking weight distributions (see visualizations in 8.2)



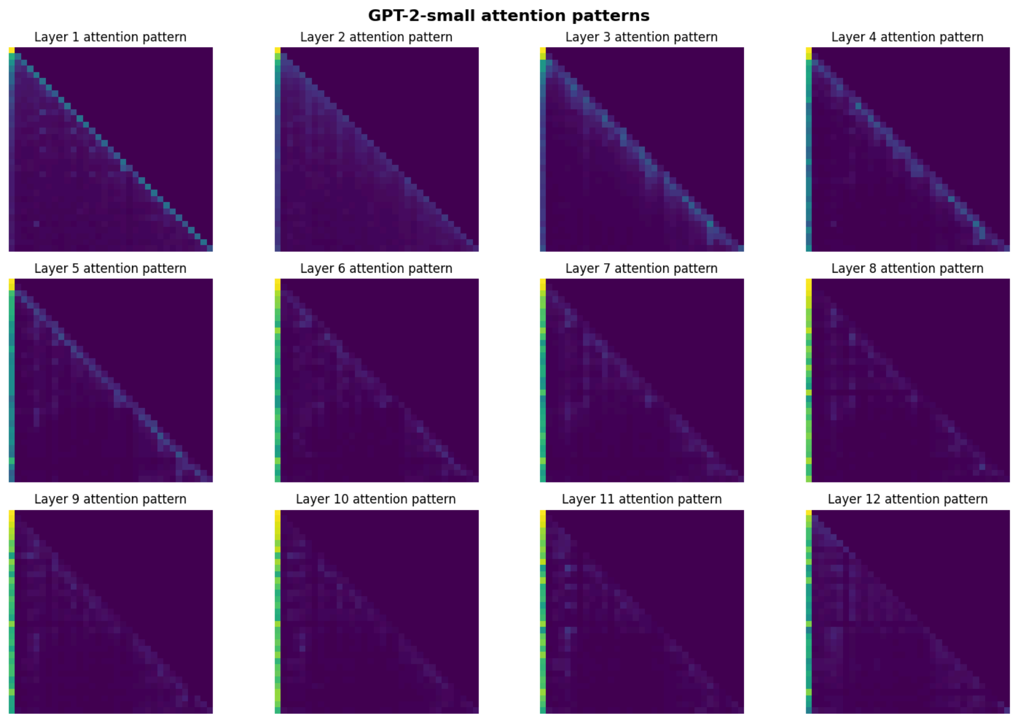
Learned (softmax) weight value across layers for each previous state



Interesting result here. Notes pulled straight from section 8.2 repeated here:

- This is very well behaved (3000 iterations of training)
- Closer states get more weight, but all states converge to same weight in later layers
- This is an interesting graph. It might sound naive but we should ask and examine: why does a given example/state prefer to look at its immediate predecessors over very distant ones? There is no bias towards recency built into the model: it (previous word as strong predictor of next word) is only in the data.
    - Note also the difference across layers. In early layers information about immediately recent context is important, but towards later layers the context window gets larger and larger until at the end each state is almost equally useful to see. So as the data flows through the model it starts by looking at recent context and gradually incorporating more distant context.
    - Note difference / similarity with self attention:
        - Difference: each token attends to each other token in a way that is unbiased by time (of course there is position bias almost always built in, but even without position bias decoder models learn sequence order (NoPE paper, others). So one gets attention patterns that are (in theory) based in semantics rather than recency: attention strength should just be affinity
        - Similarity: these similarities/affinities/attention patterns end up in practice being heavily influenced by the time dimension. More recent tokens play a more important factor, on average, than distant ones.

- As we go deeper into layers, the attention patterns start out heavily focused on recent tokens and gradually focus grows more diffusely across distant context
- Our previous state mechanism can be thought of as very close to sliding window attention.



(attention patterns over a sequence fed into GPT-2, built quickly on top of the TransformerLens demo notebook.)

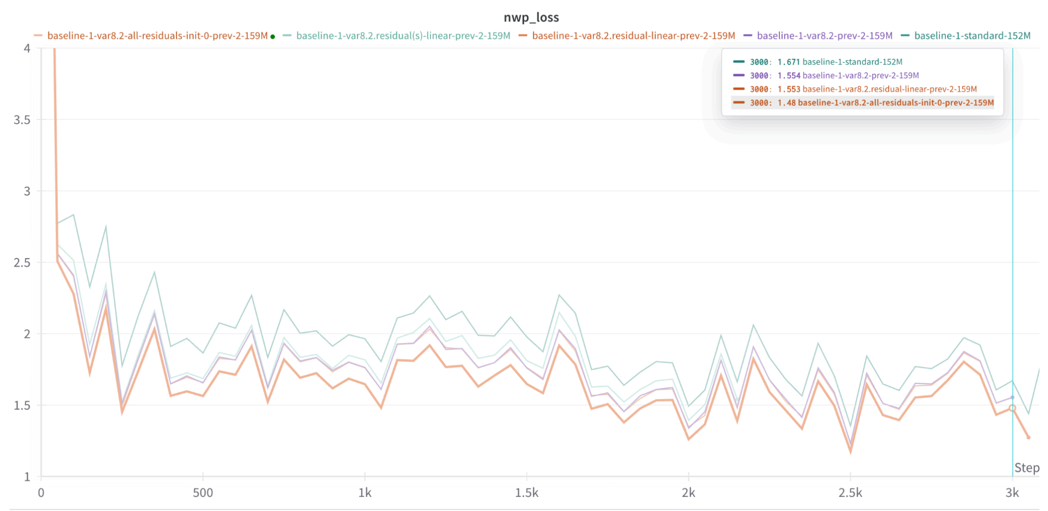
Day 4 results

- 8.3 works fine but not great. There's still more testing to do with group size for the convolution.
- At this point it's maybe worth doing another one or two small versions, but it's time to take the best candidates and scale them up.

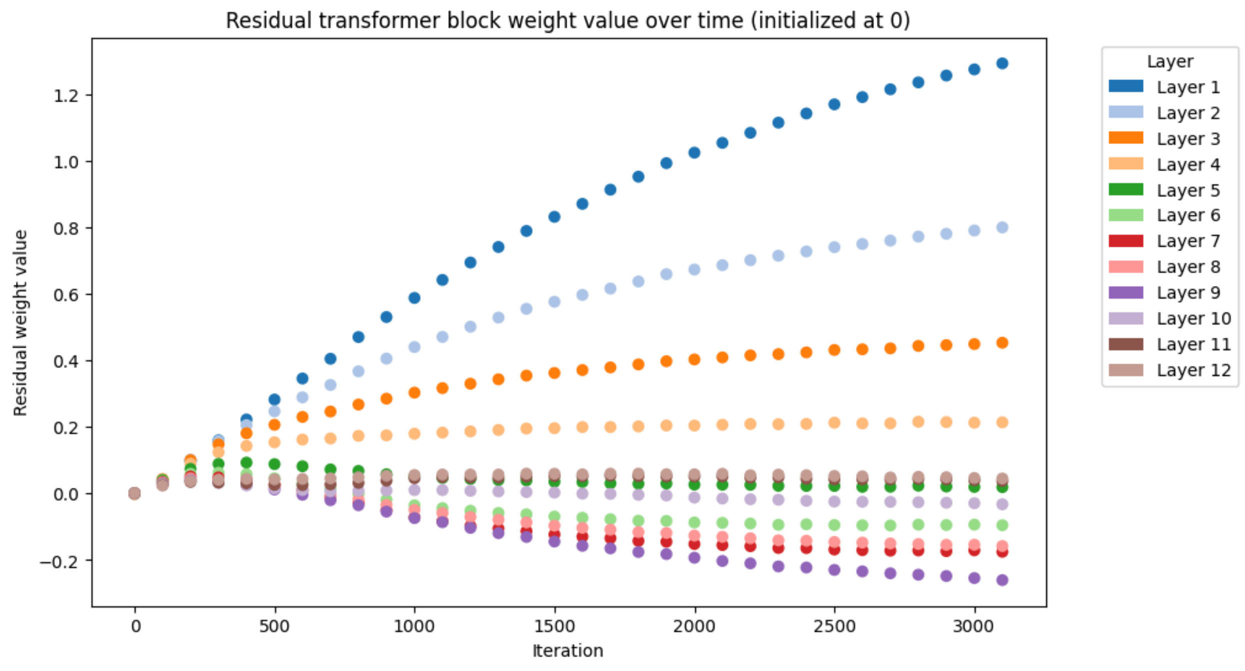
Day 5 results - subtractive skip connections

- 8.2.1-8.2.4 variations didn't do well - all were variations on attempts to more thoroughly incorporate input data as a selector gate over the previous states
- Variation 9 NaNs, we can revisit: I suspect repeated matmuls are blowing the numbers up, so a norm or division by sqrt(dim)

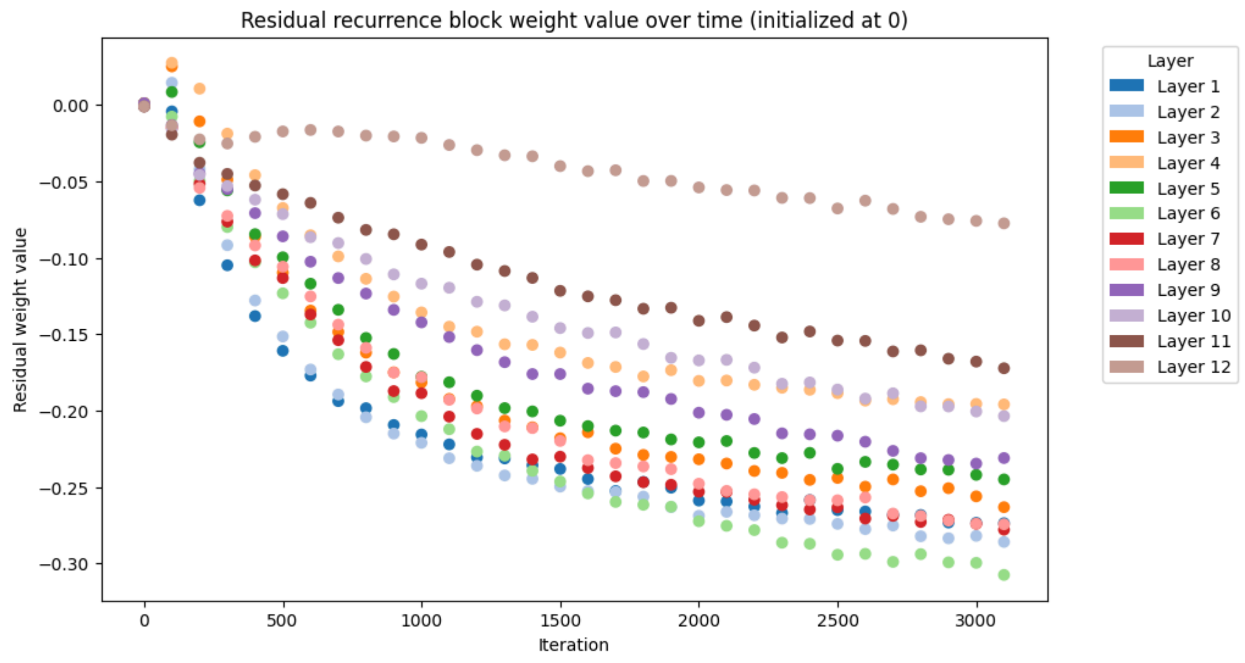
- 8.2 with residual weights very good results. So: add residual to the recurrent component and add a weight to the recurrent stream. We also add a weight to the residual component of the transformer block
    - The main result is that **initializing the residual streams to 0** provides a significant performance improvement



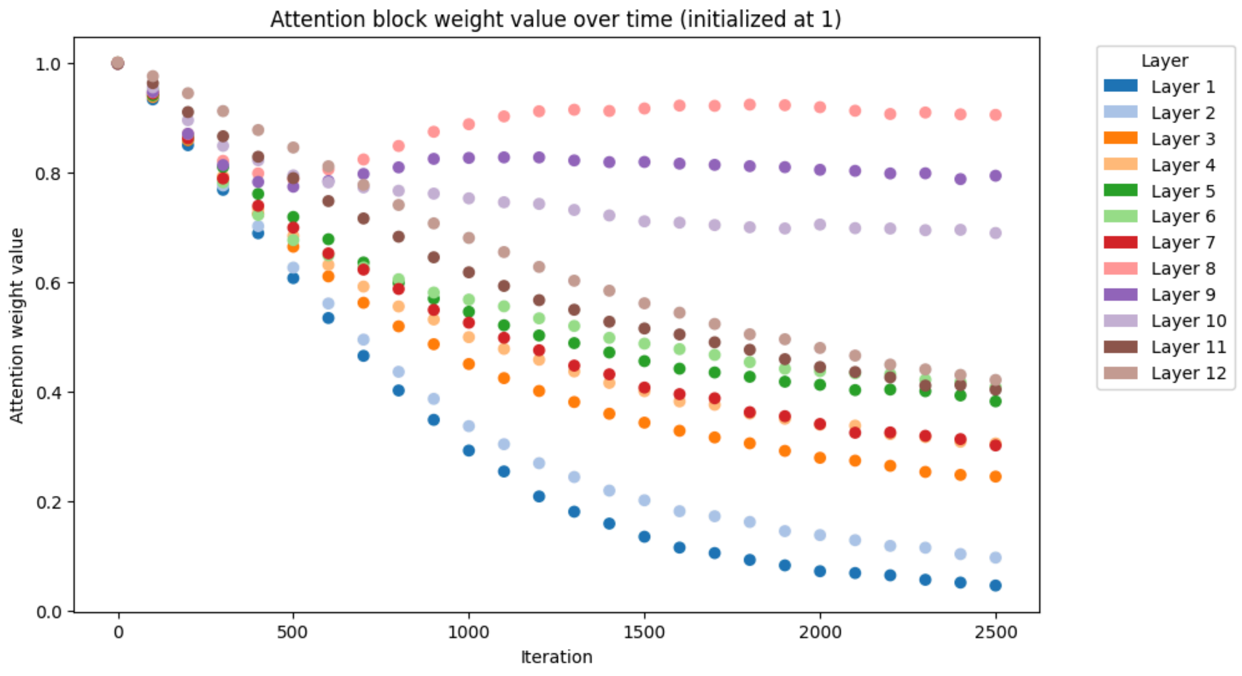
    -
    - Initializing the attention specific residual at 0 does not work well, but initializing at 1 possibly helps, or esle is equal.
    - The analysis of these residual weights is very revealing



- The above is the residual for the entire transformer block
- Later layers not only prefer the transformed data, they actually would like to subtract the input representation (this is how I read it)! Very interesting.



- Here as well in the recurrence block, the model actively prefers to subtract the input representation. Note that is not more heavily weighting the recurrence vs. the residual, it is actively subtracting the residual
- Lastly, we attach a weight to the attention block itself (not the residual attached tot he attention block)



-
- Very interesting behavior in the 8-10 layers, note the discontinuous jump between layer 7 and 8.

- The overall more important point is that the model prefers to reduce the amount that attention influences the computation. I would have expected attention weight to be > 1.0 but for all layers and all time steps it is < 1.0, and in many layers < .5.
    - TODO: test this on a regular transformer. The decreased weight might be because the necessary computation is taken over sufficiently by the weak recurrence blocks.

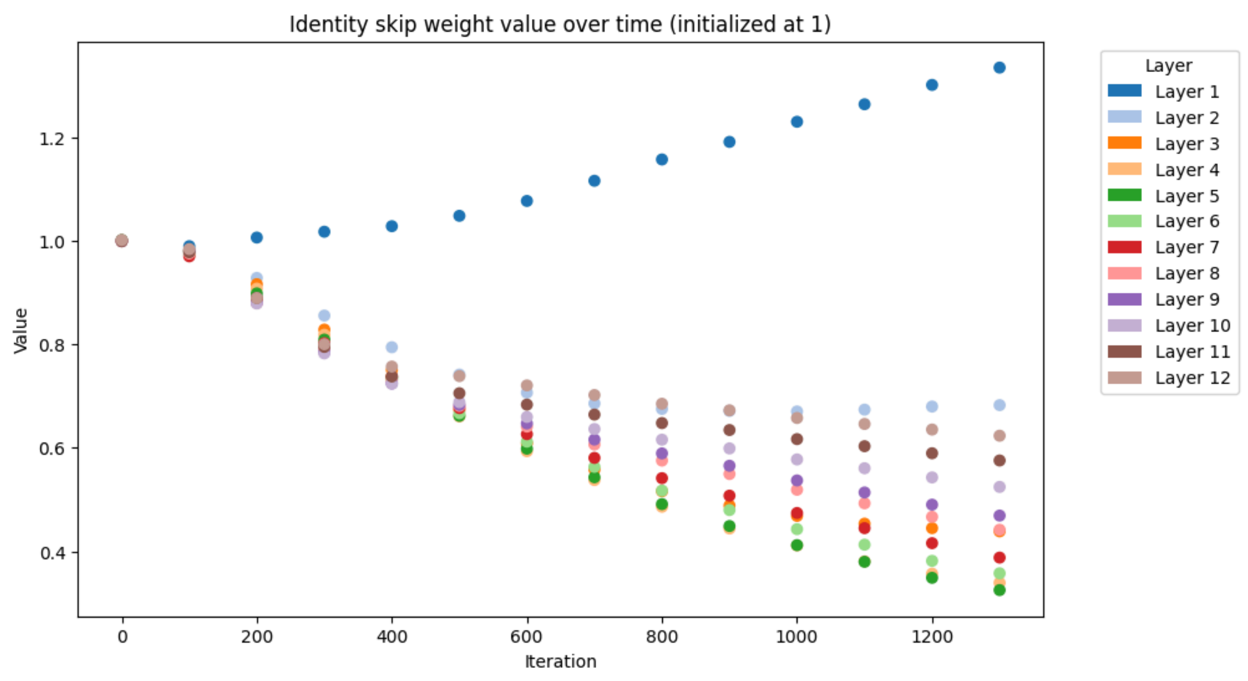
Day 6 results - subtractive skip connection

- Added a more “proper” version of the Griffin computation (7.2) but it ends up adding so many parameters, and we’re not doing the cache operation, so it seems just like a very overweight version of the 8.x models that we already have
- Lots of testing of skip and residual weights.
    - Anything other than identity/block skip weight and recurrence skip weight both init @ 0 seems worse. This includes intiializing at 1, adding skip weights to other parts of computation, adding weights to the residuals of ff, attention, recurrence, block…
    - So **the best combo is identity and recurrence skip weights @ 0**. **Is this good performance just a sum of its parts? No. Testing each individually shows that the recurrence weight @ 0 has essentially no effect, and the identity weight @ 0 on its own has a small effect. So there is some benefit beyond just the additive effect of these skip connections**
- Tested the skip connection on a standard vanilla transformer
- Init @ 1 results in worse performance
- Init @ 0 results in a large boost.

The story as I can tell so far is basically:

The skip connection is important for early layers (and redundant layers if they exist) to let the network pass the identity through (this has been identified before: as the network “warms up” these early layers just want to pass the input through). But it actually hurts middle/later layers because the residual is important, and diluting the residual with identity hurts performance. For this reason the middle layers want to boost their residual signal by actually **subtracting** the identity. Adding a learnable multiplicative weight to skip connection allows the network to do both. Note that the model is not setting the identity connection to 0: it is actually subtracting a portion of the input representation from the result of the residual computation, as if to erase the effect of the input leaving an even more exaggerated residual.

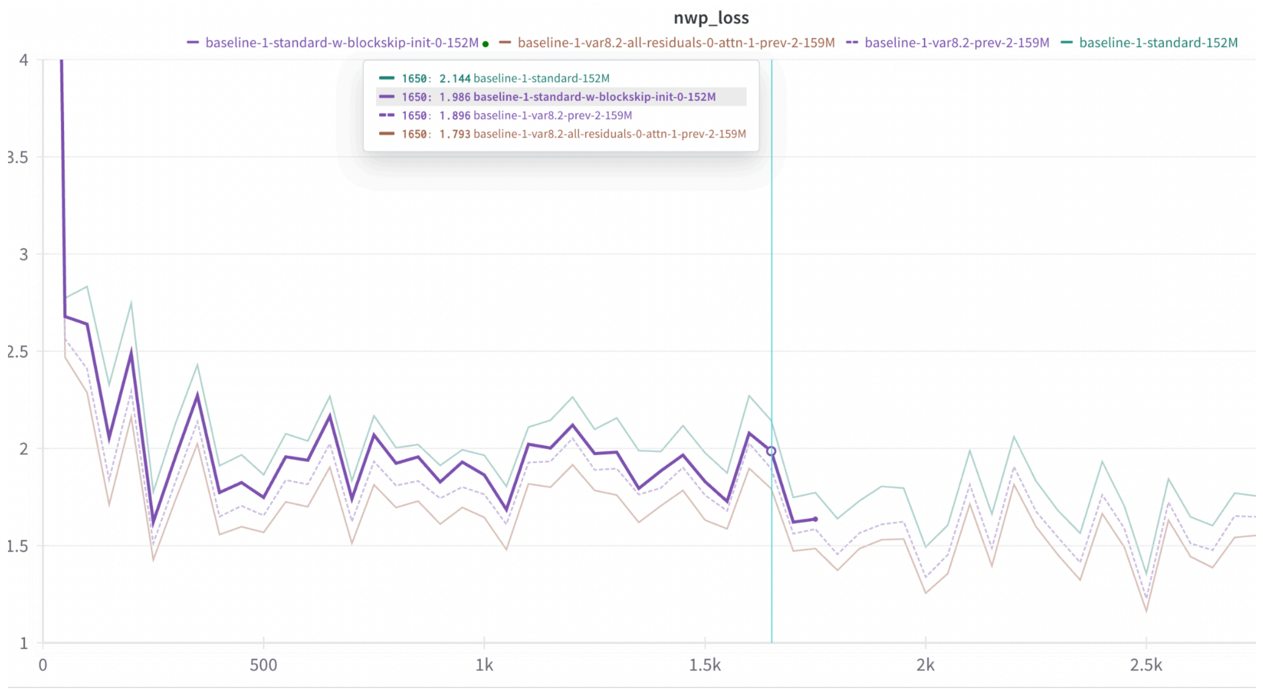
It’s quite sensitive to initialization as well. If you initialize the weight at 1.0, which makes the most sense, the model actually does slightly worse. So it seems like a modification that does nothing. But you will notice the following pattern:



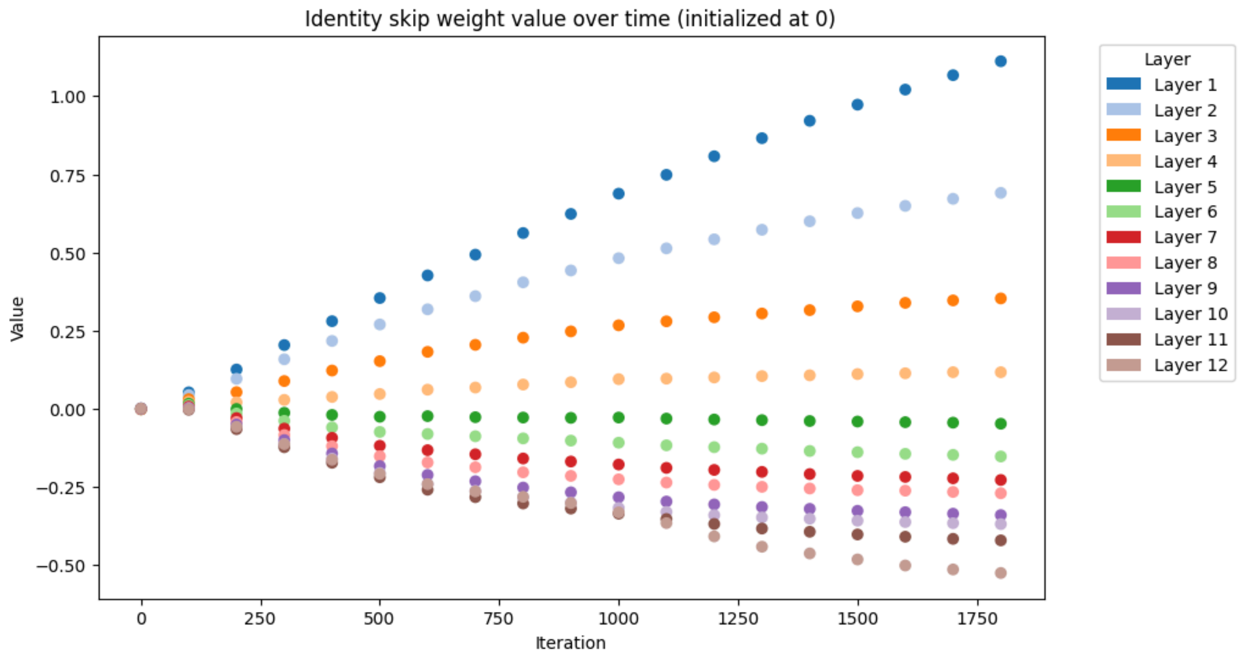
So the early layers want stronger identity connections, less so for the later layers.

[It's a bad habit to talk about what the model "wants" or what it "prefers" but we can pull some broad generalizations here. Note also that just because these values are heading up or down does not mean that in all cases they converge to a "preferred" value, and it's just a matter of finding the value they converge to and initializing it as such.
Different initializations lead to different convergences on "preferred" values, different networks lead to different convergences: the system balances itself. That said, as we will see, some initializations are better than others.]

Initializing learnable block/identity skip connections @ 0 makes the model perform considerably better (solid purple) than vanilla (green)

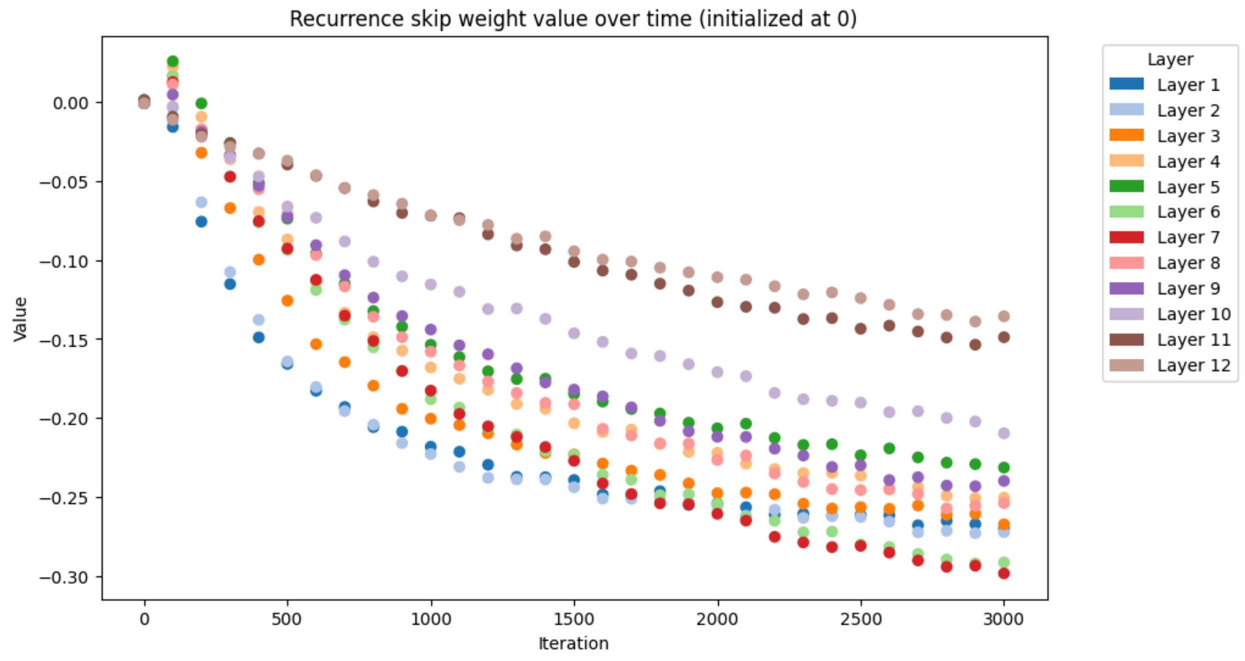


And you see the following:



It's not clear why initializing at 0 allows it to perform considerably better. Perhaps the benefit of having a strong early layer identity signal is much less important than being able to subtract the identity in later layers. Initializing the early layers at higher value (first two layers at .5, e.g.) doesn't do much to performance, so that supports this view. So this is the working hypothesis.

Other residuals have even stronger preference to subtract the identity. In an experimental architecture that is a vanilla transformer which includes a style of recurrent computation between attention and feed forward at each layer, the recurrent skip connection is even more dramatic:



Where early layers, even more so than the later layers, prefer to subtract the identity.