Learned Dropout

> NB: LaTeX here is optimized for Github's Markdown, so please view it on Github. Also, Safari does not render Github's LaTeX and some SVG files well, so Chrome is advised.

$Dropout$ is a very effective yet simple regularization technique. However, its random implementation relegates it to training time only and renders it invariant to input. Here, I present $LearnedDropout$, a parametrized dropout module that learns the best dropout for each unique input (i.e. variant to input). Results demonstrate its efficacy and competitiveness with both the canonical $Dropout$ and MoE (Mixture of Experts).

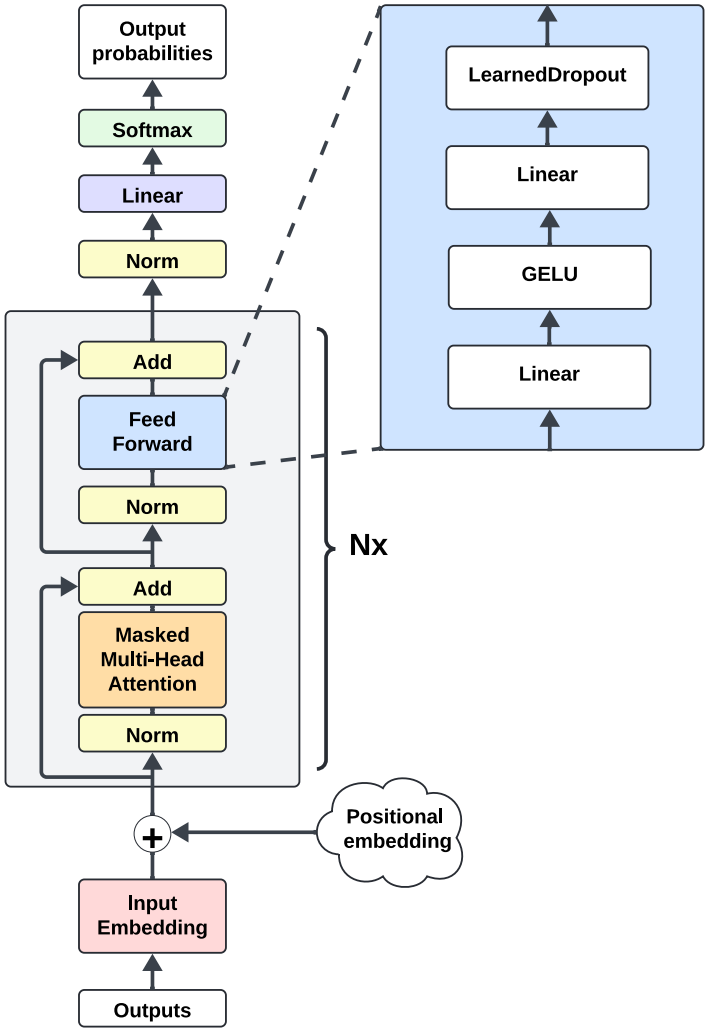
Motivations

$Dropout$ is a very popular technique that regularizes the model training to be more robust against overfitting and thus yields improved generalization. It simply works by randomly setting some values of a tensor to zero, with the ratio of zero values determined by a hyperparameter. When a value is set to zero, it becomes effectively detached from the computational graph, so all the parameters that contributed to that value will have a gradient of 0 w.r.t. that value. In doing so, $Dropout$ essentially creates a subgraph of the model because setting values to zeroes practically turns off part of the model. Given the randomness, every forward pass results in a different (transient) subgraph. Then, the final pre-trained model constitutes the ensemble of all the different subgraphs $Dropout$ created. Furthermore, observe that this outcome is not so conceptually removed from MoE's outcome. Each subgraph can be loosely thought of as an expert, and through these subgraphs, $Dropout$ (very weakly) partitions the model into different experts, like MoE.

Yet, unlike MoE, the random implementation means that 1) it is not useful during inference and 2) it is invariant to input. 1) limits the benefit of $Dropout$ to pre-training only, but 2) represents the larger reason why MoE produces better performance than $Dropout$. To overcome these deficits, the dropout module needs to be parametrized to permit the model to learn the best dropout, for every unique input. This change should make it a compelling alternative to both $Dropout$ and MoE.

Architecture

At the high-level, the architecture consists of a canonical decoder-only transformer with the new dropout module $LearnedDropout$ substituting for $Dropout$. $LearnedDropout$ is more computationally demanding than $Dropout$, so instead of a complete substitution (because of my limited compute budget), the model here limited it to the FeedForward blocks, after the last linear pass.



To encourage more dropout, a dropout L_1 norm penalty is added to the model loss.

LearnedDropout

Like every dropout implementation, the new $LearnedDropout$ module computes a dropout mask $M \in \{0, 1\}$ that is applied to the dropout input $X = \{x_1, x_2, \ldots, x_n\}$. The crux lies in the mask M's computation. The canonical $Dropout$ module randomly generates the dropout mask M from a Bernoulli distribution $M \sim \text{Bernoulli}(r)$, where r is the dropout rate hyperparameter. To enable learning, $LearnedDropout$ needs to generate the mask in a fully differentiable way. Normally, differentiability comes at the cost of loosing the $\in \{0, 1\}$ guarantee in favor of $M \in [0, 1]$. However, the implementation presented below suffers no such fate.

First, for a dropout to be highly variant to input X, it needs to leverage the causal dependencies between the input constituents $\{x_i \mid x_i \in X\}$ (i.e. across the T dimension). Therefore, a multi-headed attention operation is performed on the dropout input (without residual connection and other secondary operations). Stated more formally,

$$
\begin{aligned}
&W_Q, W_K, W_V := \text{attention weights} \\
&X := \text{input of } LearnedDropout
\end{aligned}
$$

$$
\begin{aligned}
&Q = X \cdot W_Q \\
&K = X \cdot W_K \\
&V = X \cdot W_V \\
&Attn = Q \cdot K^T \\
&out_{attn} = softmax(Attn) \cdot V
\end{aligned}
$$

Afterwards, the attention output out_{attn} is mapped to $[0, 1]$ with the following cosine function

$$M = 0.5 \cos(out_{attn} + B) + 0.5$$

where $B \in [0, \pi]$ is a shift bias term. This cosine function lies in the $[0, 1]$ range, and its recurrent property eliminates the risk of dropout becoming stuck in a local minimum, though at the cost of worse convergence.

Lastly, a rounding is applied to bring M to $\{0, 1\}$ to satisfy $M \in \{0, 1\}$. The rounding is important because, otherwise, the model might use the dropout module for computational ends (e.g. scaling of X). $LearnedDropout$ must remain a purely selective module. To minimize bias, the rounding rounds up or down M with a probability proportional to its values. In other words, given $M_{i,j}$, $P(M_{rounded_{(i,j)}} = 1) = M_{i,j}$ and $P(M_{rounded_{(i,j)}} = 0) = 1 - M_{i,j}$. Stated formally,

$$
\begin{aligned}
&N \coloneqq \text{a noise tensor where } N_{i,j} \sim Uniform[0,1] \\
&M_{complement} = (1 - M).detached() \\
&M_{rounded_{(i,j)}} = \begin{cases} M_{i,j} + M_{complement_{(i,j)}} & \text{if } N_{i,j} \leq M_{i,j} \\ M_{i,j} - M_{(i,j)}.detached() & \text{if } N_{i,j} > M_{i,j} \end{cases}
\end{aligned}
$$

The detachment is necessary because otherwise, the gradients always cancel out. Also, the probabilistic rounding is only used during training. During evaluation or inference, the rounding becomes deterministic in the following way

$$
M_{rounded_{(i,j)}} = \begin{cases} 1 & \text{if } M_{i,j} \geq 0.5 \\ 0 & \text{if } M_{i,j} < 0.5 \end{cases}
$$

though one could retain the probabilistic rounding even for inference as a way to increase model temperature.

In the end, the output of the module is the element-wise product between X and $M_{rounded}$

$$
X_{dropped} = X \odot M_{rounded}
$$

A faster implementation

Dropout masks $M_{rounded}$ are computed serially because they are dependent on the mask target X. However, pre-computing all $M_{rounded}$'s at the beginning of the forward pass would result in an obvious speedup. To do so, a close derivation of the input embeddings E can be used as a proxy for X. Let's denote this derivation $E_{dropout}$. Since X is a product of the attention mechanism, $E_{dropout}$ should be as well. Therefore, $E_{dropout}$ is computed as a multi-headed attention pass on E. More formally,

$W_Q, W_K, W_V :=$ attention weights
$E :=$ model input embedding, comprised of token and positional embedding

$$
\begin{aligned}
Q &= E \cdot W_Q \\
K &= E \cdot W_K \\
V &= E \cdot W_V \\
Attn &= Q \cdot K^T \\
out_{attn} &= softmax(Attn) \cdot V \\
E_{dropout} &= Linear(out_{attn})
\end{aligned}
$$

Next, use the same $E_{dropout}$ to pre-compute $M_{rounded}$ for all $LearnedDropout$ modules. Because each $LearnedDropout$ has its own attention weights, the pre-computed $M_{rounded}$ masks will still be different. Inexplicably, detaching $E_{dropout}$ before pre-computing $M_{rounded}$ resulted in slightly better performance (at least at a smaller scale). Then, at every layer, $LearnedDropout$ simply applies the pre-computed $M_{rounded}$ to X.

Dropout L1 norm penalty

Intuitively, more dropout (i.e. more 0s in M) is desirable. This intuition stems from the Occam's razor or Minimum Description Length principle. This is also analogous to desiring fewer experts per token in MoE. Yet, the model does not intrinsically favor more dropout. In fact, the opposite could happen because the next token prediction loss function can incentivize the model to use as much compute as possible, hence less dropout. To counter this, a dropout L_1 norm penalty is calculated in the following way for each $LearnedDropout$

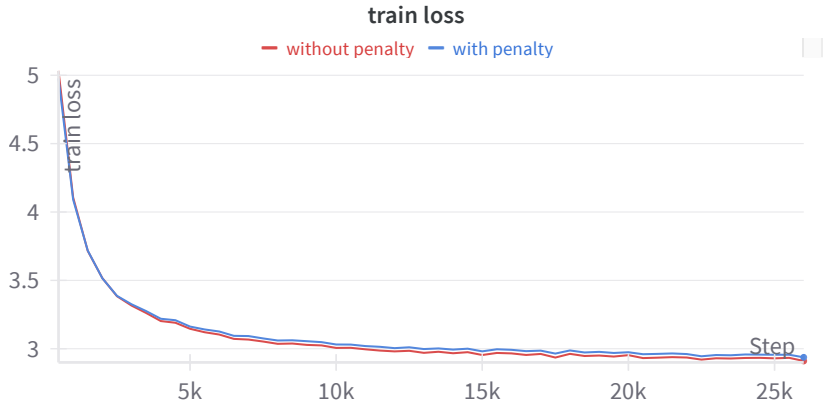
$$L_1\_norm\_penalty = \frac{M^2}{2}_1$$

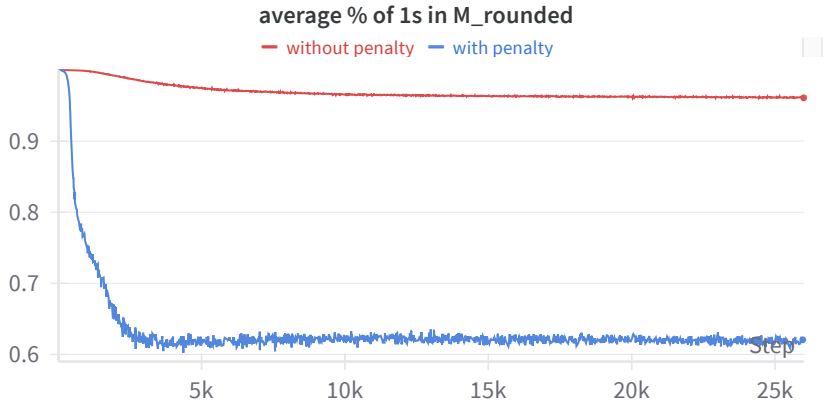
The final penalty added to the model loss is the average of all $L_1\_norm\_penalty$'s. Note that the unrounded M is used because it is deterministic. The squaring of M serves to create a non-linear penalty: as M approaches 0, the penalty should decay. The decay and the $\frac{1}{2}$ scaling ensure that the penalty does not take precedence over next token prediction.

Results

> All training runs below were done on a wikipedia dataset for 26k steps on a single A100 GPU, unless otherwise stated.
>
> Implementation of decoder-only transformer model (baseline) can be found in the `baseline_transformer` directory in this repo

First, the inclusion and exclusion of L_1 norm penalty were compared. Both had the shift bias B initialized to 0. The inclusion of the penalty outperformed its exclusion in validation loss but underperformed it in train loss. Surprisingly, the penalty absence did not detract the model from having more dropout over time, though at a much slower rate. This is promising because it means that more dropout is naturally aligned with better next token prediction, the primary objective.

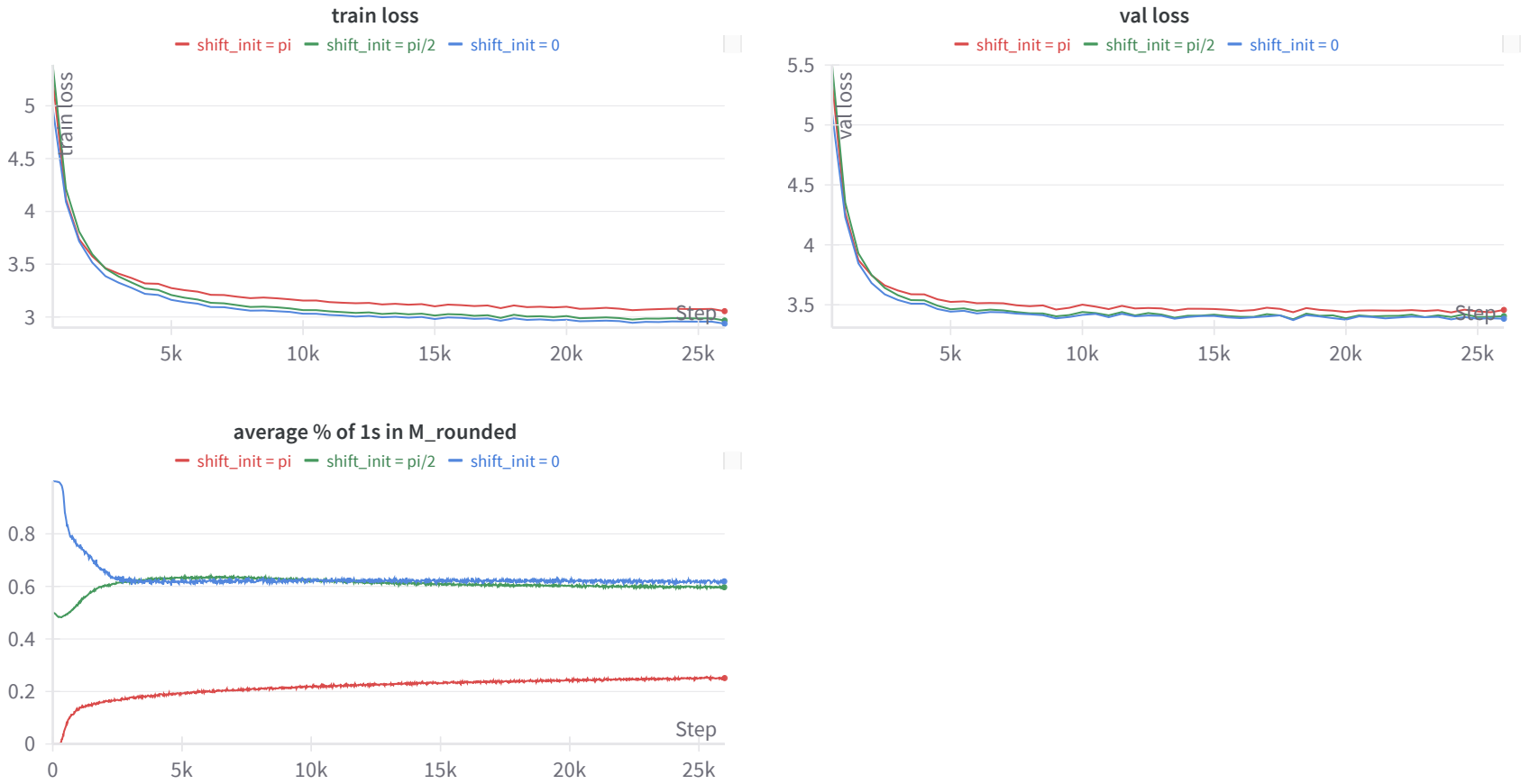




Safari may not render the charts above. Chrome is advised.

	Train loss	Val loss	average % of 1s in all $M_{rounded}$
with penalty (config)	2.937	**3.384**	**0.6167**
without penalty (config)	**2.911**	3.403	0.9609

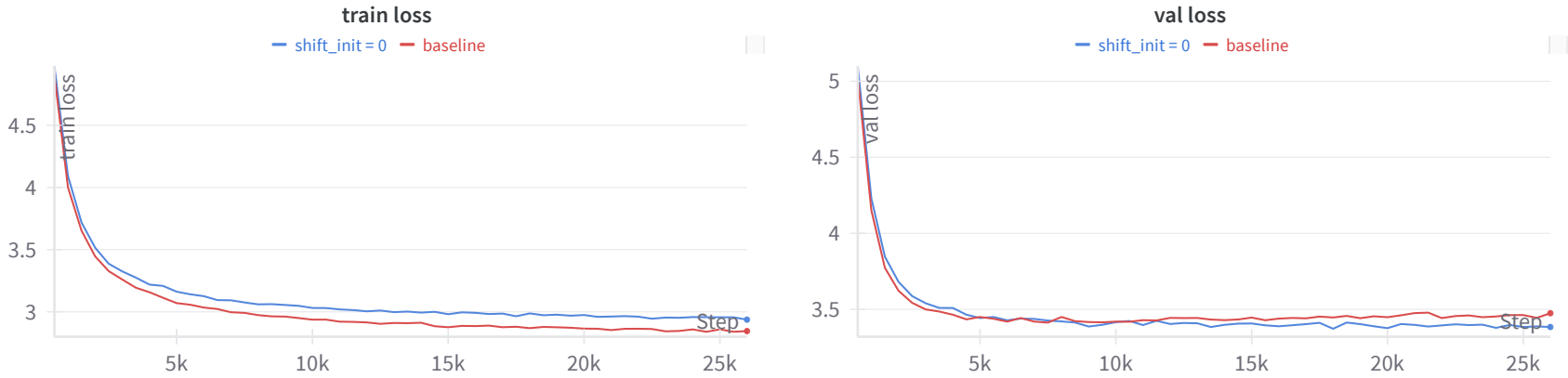
Next, using the L_1 norm penalty, different initialization values for the shift bias B were evaluated. The 0 initialization performed the best, followed by $\frac{\pi}{2}$ and π. This matches intuition because initializing with 0 means that M starts with values closer to 1, thus affording the model more compute. However, if training had run for longer, particularly with a very over-parametrized model, π initialization might have triumphed in the end. Nevertheless, the π initialization here performed very competitively despite dropping out about 75% of values on average.



Safari may not render the charts above. Chrome is advised.

	Train loss	Val loss	average % of 1s in all $M_{rounded}$
shift_init = 0 (config)	**2.937**	**3.384**	0.6167
shift_init = pi/2 (config)	2.967	3.405	0.5955
shift_init = pi (config)	3.055	3.457	**0.2507**

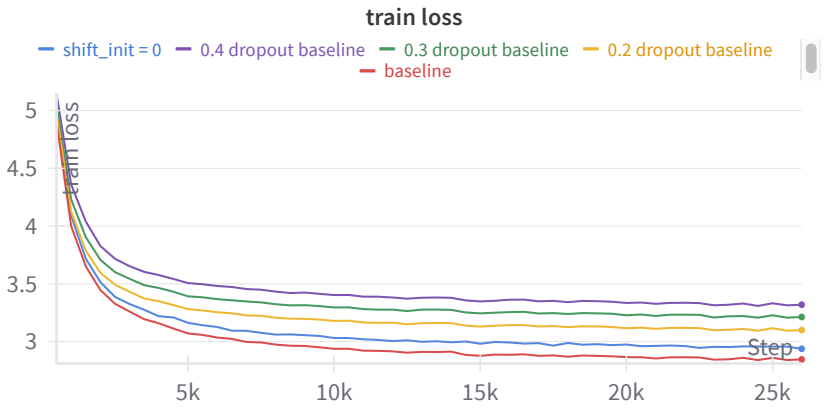
Compared to a canonical decoder-only transformer (baseline) with no dropout, the new model outperformed the baseline in validation loss only. Both completed in a similar amount of time with similar memory demands, but the baseline had more parameters.

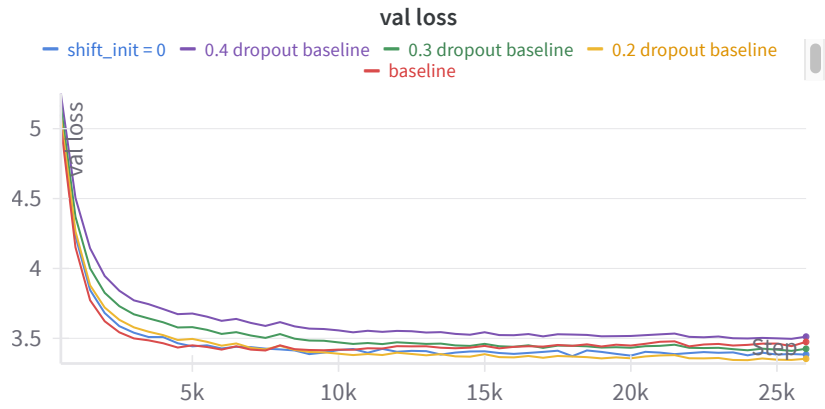


Safari may not render the charts above. Chrome is advised.

	Train loss	Val loss	average % of 1s in all $M_{rounded}$	Size (params)
shift_init = 0 (config)	2.937	**3.384**	0.6167	15,335,424
baseline (config)	**2.845**	3.475	NA	15,441,192

Three more baselines with $Dropout$ were compared: "0.2 dropout baseline", "0.3 dropout baseline", and "0.4 dropout baseline". The new model outperformed them in validation loss except for "0.2 dropout baseline", but it did outperform them all in train loss (excluding the baseline with no dropout). This is a very competitive result for $LearnedDropout$, particularly because the new model only has one $LearnedDropout$ per layer and no other dropouts (technically, other dropouts are present in the code but were inactive), while the baseline has 3 $Dropout$'s per layer.

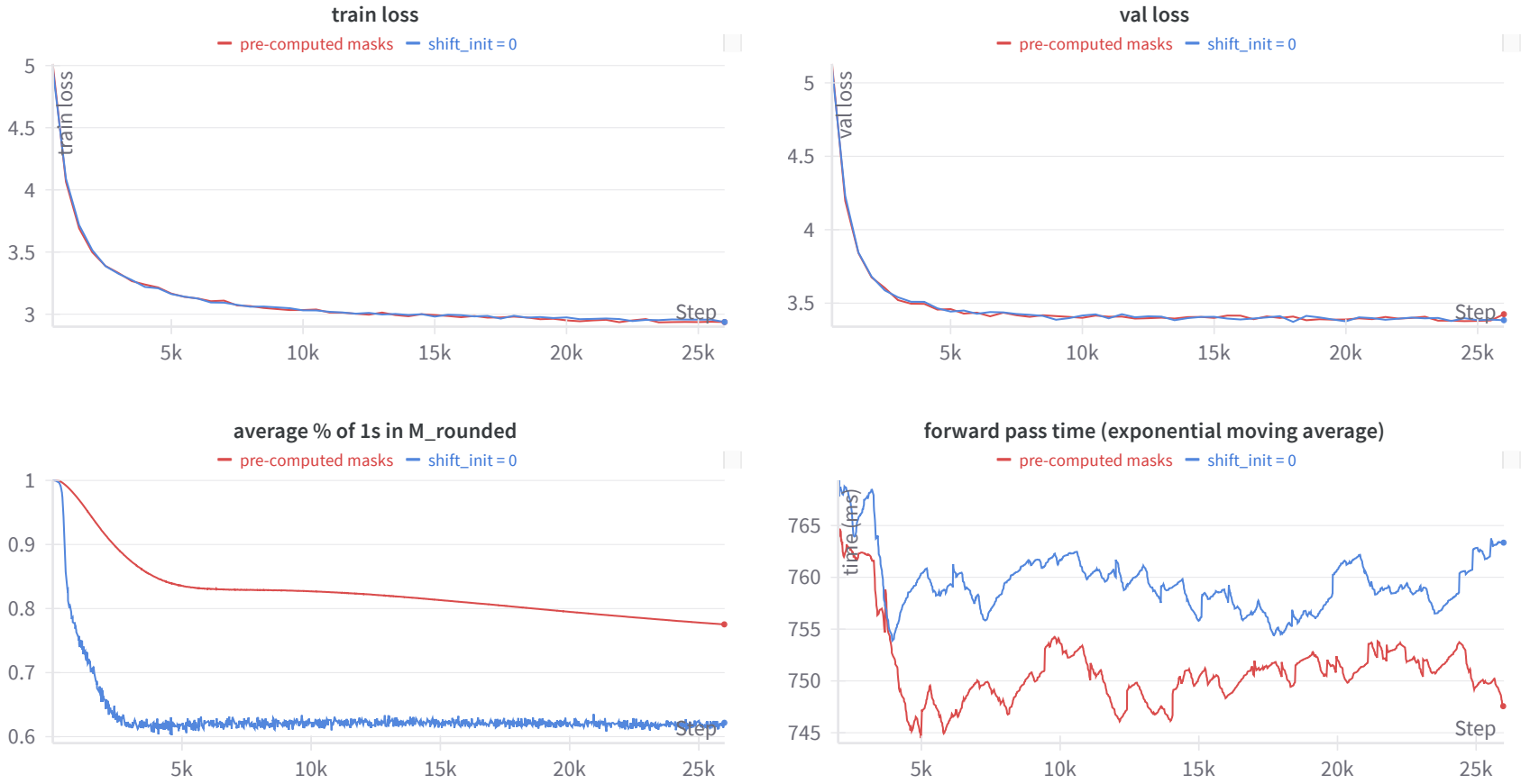




Safari may not render the charts above. Chrome is advised.

	Train loss	Val loss	average % of 1s in all $M_{rounded}$	Size (params)
shift_init = 0 (config)	2.937	3.384	0.6167	15,335,424
baseline (config)	**2.845**	3.475	NA	15,441,192
0.2 dropout baseline (config)	3.1	**3.354**	NA	15,441,192
0.3 dropout baseline (config)	3.213	3.425	NA	15,441,192
0.4 dropout baseline (config)	3.319	3.512	NA	15,441,192

Finally, the faster implementation with $E_{dropout}$ was compared. It had comparable train loss performance but underperformed on val loss. It also had less dropout on average. As predicted, it had a faster forward pass time, despite having more parameters because of the extra attention operation to compute $E_{dropout}$. Also, this speedup gain simply resulted from `torch.compile` (which both models had), so more can probably be gained from explicit code optimization.



Safari may not render the charts above. Chrome is advised.

	Train loss	Val loss	average % of 1s in all $M_{rounded}$	Size (params)
shift_init = 0 (config)	**2.937**	**3.384**	**0.6167**	15,335,424

	Train loss	Val loss	average % of 1s in all $M_{rounded}$	Size (params)
pre-computed masks (config)	**2.937**	3.425	0.7752	15,418,368

Next steps

These are some further things to look forward to:

- substitute all $Dropout$'s for $LearnedDropout$
- scale the L_1 norm penalty on an exponential schedule: the penalty is smaller earlier in the training and grows larger later in the training
  - this is already implemented. I just didn't have enough compute to test this
- implement tensor operations that take advantage of highly sparse tensors caused by $LearnedDropout$. This should greatly reduce compute time
  - this probably won't be a small effort since `torch.compile` does not currently support sparse tensors
- instead of a single cosine function that maps values to $[0, 1]$, use a Fourier series
- try bigger models, at least GPT-2 size
- run training for longer to observe long-term behavior
- evaluate on different datasets
- evaluate on non-language tasks
- empirically evaluate $LearnedDropout$ against MoE

Conclusions

Instead of setting a hyperparameter like dropout rate or top-k experts in MoE, $LearnedDropout$ permits the model to learn the best dropout/experts for each unique input. As demonstrated by the results, $LearnedDropout$ produces very competitive performances with a high dropout percentage.

The result of $LearnedDropout$ is adaptive sparsity, very similar to MoE. Unlike MoE, though, it doesn't require dictating a pre-defined sparsity target. This adaptive sparsity should enable a training paradigm that starts with a very over-parametrized dense model and makes it more sparse over time with $LearnedDropout$. Furthermore, more sparsity makes fine-tuning easier.

Alas, the principal limitation is my personal compute budget, so this project cannot avail itself of further analysis and experimentation.

Citation

If you use this codebase, or otherwise found my work valuable, please cite:

```
@misc{yan2024learned-dropout,
  title={Learned Dropout},
  author={Yan, Yifei},
  year={2024},
  url={https://github.com/yiphei/yif-AI/tree/main/learned_dropout}
}
```

Appendix

Run configs

"without penalty"

```
batch_size: 50
beta1: 0.9
beta2: 0.95
decay_lr: true
est_interval: 500
```

```
est_steps: 200
gradient_accumulation_steps: 16
lr: 0.0009
lr_decay_iters: 700000
min_lr: 9.0e-05
model_config:
  context_size: 200
  dropout_rate: 0
  l1_norm_penalty_type: null
  learned_dropout_config:
    dropout_input_type: "HIDDEN_STATE"
    mask_rounding_type: "NOISE_AND_LINEAR"
    n_head: 9
    shift_init: 0
    use_bias: false
    use_detached_input: false
  n_embed: 144
  n_head: 9
  n_layer: 26
  use_bias: false
  use_dropout_entropy_penalty: false
  use_dropout_l1_norm_penalty: false
train_steps: 26000
warmup_iters: 300
weight_decay: 0.1
```

"with penalty / shift_init = 0"

```
batch_size: 50
beta1: 0.9
beta2: 0.95
decay_lr: true
est_interval: 500
est_steps: 200
```

```
  gradient_accumulation_steps: 16
  lr: 0.0009
  lr_decay_iters: 700000
  min_lr: 9.0e-05
  model_config:
    context_size: 200
    dropout_l1_norm_coeff_config:
      max_coeff: 0.1
    dropout_rate: 0
    l1_norm_penalty_type: "SQUARED"
    learned_dropout_config:
      dropout_input_type: "HIDDEN_STATE"
      mask_rounding_type: "NOISE_AND_LINEAR"
      n_head: 9
      shift_init: 0
      use_bias: false
      use_detached_input: false
    n_embed: 144
    n_head: 9
    n_layer: 26
    use_bias: false
    use_dropout_entropy_penalty: false
    use_dropout_l1_norm_penalty: true
  train_steps: 26000
  warmup_iters: 300
  weight_decay: 0.1
```

"shift_init = pi/2"

```
  batch_size: 50
  beta1: 0.9
  beta2: 0.95
  decay_lr: true
  est_interval: 500
```

```
  est_steps: 200
  gradient_accumulation_steps: 16
  lr: 0.0009
  lr_decay_iters: 700000
  min_lr: 9.0e-05
  model_config:
    context_size: 200
    dropout_l1_norm_coeff_config:
      max_coeff: 0.1
    dropout_rate: 0
    l1_norm_penalty_type: "SQUARED"
    learned_dropout_config:
      dropout_input_type: "HIDDEN_STATE"
      mask_rounding_type: "NOISE_AND_LINEAR"
      n_head: 9
      shift_init: 1.57079
      use_bias: false
      use_detached_input: false
    n_embed: 144
    n_head: 9
    n_layer: 26
    use_bias: false
    use_dropout_entropy_penalty: false
    use_dropout_l1_norm_penalty: true
  train_steps: 26000
  warmup_iters: 300
  weight_decay: 0.1
```

"shift_init = pi"

```
  batch_size: 50
  beta1: 0.9
  beta2: 0.95
  decay_lr: true
```

```
  est_interval: 500
  est_steps: 200
  gradient_accumulation_steps: 16
  lr: 0.0009
  lr_decay_iters: 700000
  min_lr: 9.0e-05
  model_config:
    context_size: 200
    dropout_l1_norm_coeff_config:
      max_coeff: 0.1
    dropout_rate: 0
    l1_norm_penalty_type: "SQUARED"
    learned_dropout_config:
      dropout_input_type: "HIDDEN_STATE"
      mask_rounding_type: "NOISE_AND_LINEAR"
      n_head: 9
      shift_init: 3.14159
      use_bias: false
      use_detached_input: false
    n_embed: 144
    n_head: 9
    n_layer: 26
    use_bias: false
    use_dropout_entropy_penalty: false
    use_dropout_l1_norm_penalty: true
  train_steps: 26000
  warmup_iters: 300
  weight_decay: 0.1
```

"baseline"

```
  batch_size: 50
  beta1: 0.9
  beta2: 0.95
```

```
decay_lr: true
est_interval: 500
est_steps: 200
gradient_accumulation_steps: 16
lr: 0.0009
lr_decay_iters: 700000
min_lr: 9.0e-05
model_config:
  context_size: 200
  dropout_rate: 0
  n_embed: 156
  n_head: 12
  n_layer: 26
  use_bias: false
train_steps: 26000
warmup_iters: 300
weight_decay: 0.1
```

"0.2 dropout baseline"

```
batch_size: 50
beta1: 0.9
beta2: 0.95
decay_lr: true
est_interval: 500
est_steps: 200
gradient_accumulation_steps: 16
lr: 0.0009
lr_decay_iters: 700000
min_lr: 9.0e-05
model_config:
  context_size: 200
  dropout_rate: 0.2
```

```
    n_embed: 156
    n_head: 12
    n_layer: 26
    use_bias: false
  train_steps: 26000
  warmup_iters: 300
  weight_decay: 0.1
```

"0.3 dropout baseline"

```
batch_size: 50
beta1: 0.9
beta2: 0.95
decay_lr: true
est_interval: 500
est_steps: 200
gradient_accumulation_steps: 16
lr: 0.0009
lr_decay_iters: 700000
min_lr: 9.0e-05
model_config:
  context_size: 200
  dropout_rate: 0.3
  n_embed: 156
  n_head: 12
  n_layer: 26
  use_bias: false
train_steps: 26000
warmup_iters: 300
weight_decay: 0.1
```

"0.4 dropout baseline"

```
batch_size: 50
beta1: 0.9
beta2: 0.95
decay_lr: true
est_interval: 500
est_steps: 200
gradient_accumulation_steps: 16
lr: 0.0009
lr_decay_iters: 700000
min_lr: 9.0e-05
model_config:
  context_size: 200
  dropout_rate: 0.4
  n_embed: 156
  n_head: 12
  n_layer: 26
  use_bias: false
train_steps: 26000
warmup_iters: 300
weight_decay: 0.1
```

"pre-computed masks"

```
batch_size: 50
beta1: 0.9
beta2: 0.95
decay_lr: true
est_interval: 500
est_steps: 200
gradient_accumulation_steps: 16
lr: 0.0009
```

```
lr_decay_iters: 700000
min_lr: 9.0e-05
model_config:
  context_size: 200
  dropout_l1_norm_coeff_config:
    max_coeff: 0.1
  dropout_rate: 0
  l1_norm_penalty_type: "SQUARED"
  learned_dropout_config:
    dropout_input_type: "EMBED"
    mask_rounding_type: "NOISE_AND_LINEAR"
    n_head: 9
    shift_init: 0
    use_bias: false
    use_detached_input: true
  n_embed: 144
  n_head: 9
  n_layer: 26
  use_bias: false
  use_dropout_entropy_penalty: false
  use_dropout_l1_norm_penalty: true
train_steps: 28000
warmup_iters: 300
weight_decay: 0.1
```