IFT6135A - Homework #2

Report for the Practical Part of the Homework

Université de Montréal

Felix Wilhelmy

April 5, 2025

Introduction

In this report, we explore and compare the performance and scaling behavior of two manually implemented sequential language models (LSTM and a GPT) on a modular arithmetic dataset. Our experiments aim to understand how different model configurations affect training dynamics, generalization performance, and robustness to overfitting.

The experiments were conducted on Google Colab Pro using a T4 GPU with CUDA. Although we experimented with various configurations for the number of workers, we ultimately set the number of workers to 0 for all experiments. The baseline training configuration includes the AdamW optimizer with a learning rate of 10^{-3} and a weight decay of 10^{0}. This setup provides a consistent foundation from which to explore how adjustments in model size and training dynamics influence performance. The report is organized as follows: we first describe the experimental setup and methodology, then present detailed analyses of model performance under different configurations, and finally, we discuss the implications of our findings for scaling language models.

Problem 1: LSTM Implementation

We first define the notation:

- d_x: Input dimension.
- d_h: Hidden dimension.
- L: Number of LSTM layers.
- $b \in \{0, 1\}$: Indicator for bias (1 if bias is True, 0 otherwise).

The `LSTM` module is implemented by stacking L `LSTMCell` modules. In the constructor of the `LSTM` class, the cells are created as follows:

```
# Create LSTM layers
self.layers = nn.ModuleList([
    # input_size for the first layer, hidden_size for the rest
    LSTMCell(input_size if layer == 0 else hidden_size, hidden_size, bias)
    for layer in range(num_layers)
])
```

This shows that the first cell receives an input of dimension d_x while the remaining $L-1$ cells receive an input of dimension d_h. Thus, we write the total number of learnable parameters as

$$P_{\text{LSTM}} = P_{\text{cell}}^{(1)} + (L-1)\, P_{\text{cell}}^{(2:L)}$$

where $P_{\text{cell}}^{(1)}$ is the parameter count for the first cell and $P_{\text{cell}}^{(2:L)}$ is for each subsequent cell.

Inside each `LSTMCell`, there are 4 linear layers corresponding to the input, forget, candidate (cell), and output gates. In the code, these are defined as:

```
self.input_gate = nn.Linear(combined_dim, hidden_size, bias=bias)       # [W_i, U_i] and b_i
self.forget_gate = nn.Linear(combined_dim, hidden_size, bias=bias)      # [W_f, U_f] and b_f
self.output_gate = nn.Linear(combined_dim, hidden_size, bias=bias)      # [W_o, U_o] and b_o
self.candidate_cell = nn.Linear(combined_dim, hidden_size, bias=bias)   # [W_c, U_c] and b_c
```

Here, `combined_dim` is equal to $d_{\text{in}} + d_h$ (with $d_{\text{in}} = d_x$ for the first cell and $d_{\text{in}} = d_h$ for the rest). For each gate, the weight matrix has size $(d_{\text{in}} + d_h) \times d_h$ and, if bias is used, a bias vector of size d_h. Therefore, the parameter count per gate is

$$\underbrace{(d_{\text{in}} + d_h) \times d_h}_{\text{weights}} + \underbrace{b \cdot d_h}_{\text{bias}}$$

Since there are 4 gates, the total number of parameters in a single `LSTMCell` is

$$P_{\text{cell}} = 4\Big[(d_{\text{in}} + d_h)\, d_h + b\, d_h\Big]$$

For the first layer, we have $d_{\text{in}} = d_x$. Thus, the parameter count for the first cell is

$$P_{\text{cell}}^{(1)} = 4\Big[(d_x + d_h)\, d_h + b\, d_h)\Big] = 4\, d_h\Big[d_x + d_h + b\Big]$$

For layers 2 through L, $d_{\text{in}} = d_h$ so that

$$P_{\text{cell}}^{(2:L)} = 4\Big[(d_h + d_h)\, d_h + b\, d_h\Big] = 4\, d_h\Big[(2\, d_h + b)\Big]$$

Combining these, the total number of learnable parameters in the LSTM module is

$$P_{\text{LSTM}} = 4\, d_h\Big[d_x + d_h + b + (L-1)\,(2\, d_h + b)\Big]$$

To summarize, we have

$$P_{\text{LSTM}} = \begin{cases} 4\, d_h\,[d_x + d_h + 1 + (L-1)\,(2\, d_h + 1)] & \textbf{With bias } (b = 1) \\ 4\, d_h\,[d_x + d_h + (L-1)\,(2\, d_h)] & \textbf{Without bias } (b = 0) \end{cases}$$

Problem 2: GPT Implementation

Problem 2.6: Multi-Head Attention Parameters

To determine the number of learnable parameters in the Multi-Head Attention (MHA) module, we first inspect its constructor:

```
assert d_model % num_heads == 0
self.head_size = d_model // num_heads
self.num_heads = num_heads

self.W_Q = nn.Linear(d_model, d_model, bias=bias)
self.W_K = nn.Linear(d_model, d_model, bias=bias)
self.W_V = nn.Linear(d_model, d_model, bias=bias)
self.W_O = nn.Linear(d_model, d_model, bias=bias)
```

Here, the key observation is that each linear layer projects from a dimension

$$d_{\text{model}} = d_k \times h$$

back to the same dimension, where d_k is the head size, h is the number of heads, and d_{model} is the model dimensionality

For a linear layer with input and output dimensions d_{model}, the number of learnable parameters is

$$\underbrace{d_{\text{model}} \times d_{\text{model}}}_{\text{weights}} + \underbrace{b \cdot d_{\text{model}}}_{\text{bias}}$$

where $b = 1$ if biases are enabled and $b = 0$ otherwise.

Since there are 4 such layers, the total number of parameters in the MHA module is

$$P_{\text{MHA}} = 4\Big[d_{\text{model}}^2 + b\, d_{\text{model}}\Big]$$

Expressing this in terms of the head size d_k and the number of heads h, with $d_{\text{model}} = d_k \cdot h$, we obtain

$$P_{\text{MHA}} = 4\Big[(d_k\, h)^2 + b\, d_k\, h\Big]$$

In summary, we have

$$P_{\text{MHA}} = \begin{cases} 4\Big[(d_k\, h)^2 + d_k\, h\Big] & \textbf{with bias } (b = 1) \\ 4\Big[(d_k\, h)^2\Big] & \textbf{without bias } (b = 0) \end{cases}$$

Problem 2.10: Decoder Parameters

In the decoder constructor

```
self.blocks = nn.ModuleList(
    [
        Block(d_model, num_heads, multiplier, dropout, non_linearity,  bias=bias)
        for _ in range(num_blocks)
    ]
)
```

We see that a decoder is composed of L blocks

$$P_{\text{Decoder}} = L \cdot P_{\text{Block}}$$

Bellow is a snippet of in the constructor of a Block module representing it's layers

```
self.self_attn = MultiHeadedAttention(d_model, num_heads, bias=bias)
self.self_attn_norm = LayerNorm(d_model)

d_ff = int(multiplier * d_model)
self.ffn = nn.Sequential(
    nn.Linear(d_model, d_ff, bias=False),
    non_linearities[non_linearity](),
    nn.Linear(d_ff, d_model, bias=False),
)
self.ffn_drop = nn.Dropout(p=dropout)
self.ffn_norm = LayerNorm(d_model)
```

where each block consists of a Multi-Head Attention, Feed-Forward Network and 2 Normalization layers.

Multi-Headed Attention

From Problem 2.6, the MHA parameter count is given by

$$P_{\text{MHA}} = 4\Big[d_{\text{model}}^2 + b\, d_{\text{model}}\Big]$$

with $b = 1$ if biases are enabled and $b = 0$ otherwise

Feed-Forward Network

The FFN consists of two linear layers with no bias and activation function in between the layers. The first layer projects from d_{model} to d_{ff} and the second projects from d_{ff} back to d_{model} with

$$d_{ff} = \text{m} \times d_{\text{model}}$$

Thus, the parameter count for the FNN is

$$P_{\text{FFN}} = 2\,m\,d_{\text{model}}^2$$

Parameters per Block

The total parameters for one block is the sum of the MHA, FNN and the two normalization layers parameters

$$P_{\text{Block}} = \underbrace{4\Big[d_{\text{model}}^2 + b\,d_{\text{model}}\Big]}_{\text{MHA}} + \underbrace{2\,d_{model}}_{\text{Norm}} + \underbrace{2\,m\,d_{\text{model}}^2}_{\text{FFN}} + \underbrace{2\,d_{model}}_{\text{Norm}}$$

Which can be expanded to

$$P_{\text{Block}} = d_{\text{model}}^2\Big(4 + 2\,m\Big) + d_{\text{model}}\Big(4\,b + 4\Big)$$

Decoder Total Parameter Count:

If the decoder is composed of L blocks, the total parameter count for the decoder is

$$P_{\text{Decoder}} = L\Big[d_{\text{model}}^2\Big(4 + 2\,m\Big) + d_{\text{model}}\Big(4\,b + 4\Big)\Big]$$

In summary

$$P_{\text{decoder}} = \begin{cases} L\Big[d_{\text{model}}^2\Big(4 + 2\,m\Big) + 8\,d_{\text{model}}\Big] & \textbf{with bias } (b = 1) \\ L\Big[d_{\text{model}}^2\Big(4 + 2\,m\Big) + 4\,d_{\text{model}}\Big] & \textbf{without bias } (b = 0) \end{cases}$$

Problem 4.1 & 4.2: Experiment 1 (Sanity check)

Problem 4.1

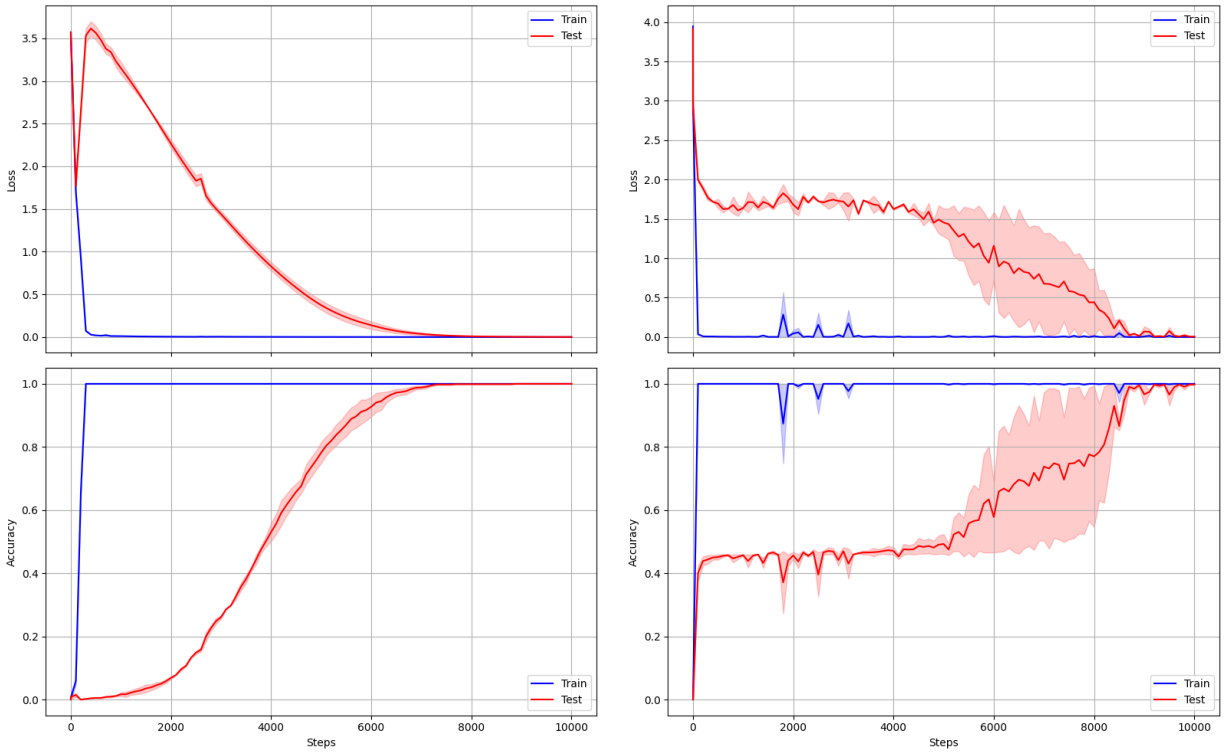

Figure 1: This figure contains four subplots detailing the loss (top) and it's corresponding accuracy (bottom) for the LSTM (left) and GPT (right) models. Each panel plots it's metrics versus the training steps with a blue curve for the training set and a red curve for test set, including shaded areas for standard deviation across runs.

Problem 4.2

		LSTM			**GPT**	
Train	$\mathcal{A}_{train} =$	1.0 ± 0.0	$t_f = 100$	$\mathcal{A}_{train} =$	1.0 ± 0.0	$t_f = 300$
	$\mathcal{L}_{train} =$	$1.54e^{-4} \pm 2.82e^{-7}$	$t_f = 9600$	$\mathcal{L}_{train} =$	$\mathbf{9.66e^{-5} \pm 1.42e^{-5}}$	$t_f = 10001$
Test	$\mathcal{A}_{val} =$	0.9979 ± 0.0021	$t_f = 9200$	$\mathcal{A}_{val} =$	1.0 ± 0.0	$t_f = 8900$
	$\mathcal{L}_{val} =$	$5.01e^{-3} \pm 4.26e^{-3}$	$t_f = 10000$	$\mathcal{L}_{val} =$	$\mathbf{8.88e^{-4} \pm 2.36e^{-4}}$	$t_f = 10001$
Δt	$\Delta t(\mathcal{A}) =$	$9200 - 100 = 9100$		$\Delta t(\mathcal{A}) =$	$8900 - 300 = 8600$	
	$\Delta t(\mathcal{L}) =$	$10000 - 9600 = 400$		$\Delta t(\mathcal{L}) =$	$10001 - 10001 = 0$	

Table 1: Performance metrics for LSTM and GPT models. Metrics are reported as mean ± std with the best step indicated to the right of the metrics as t_f.

Problem 4.3: Experiment 2 (Scaling Data Size)

Problem 4.3.a

LSTM

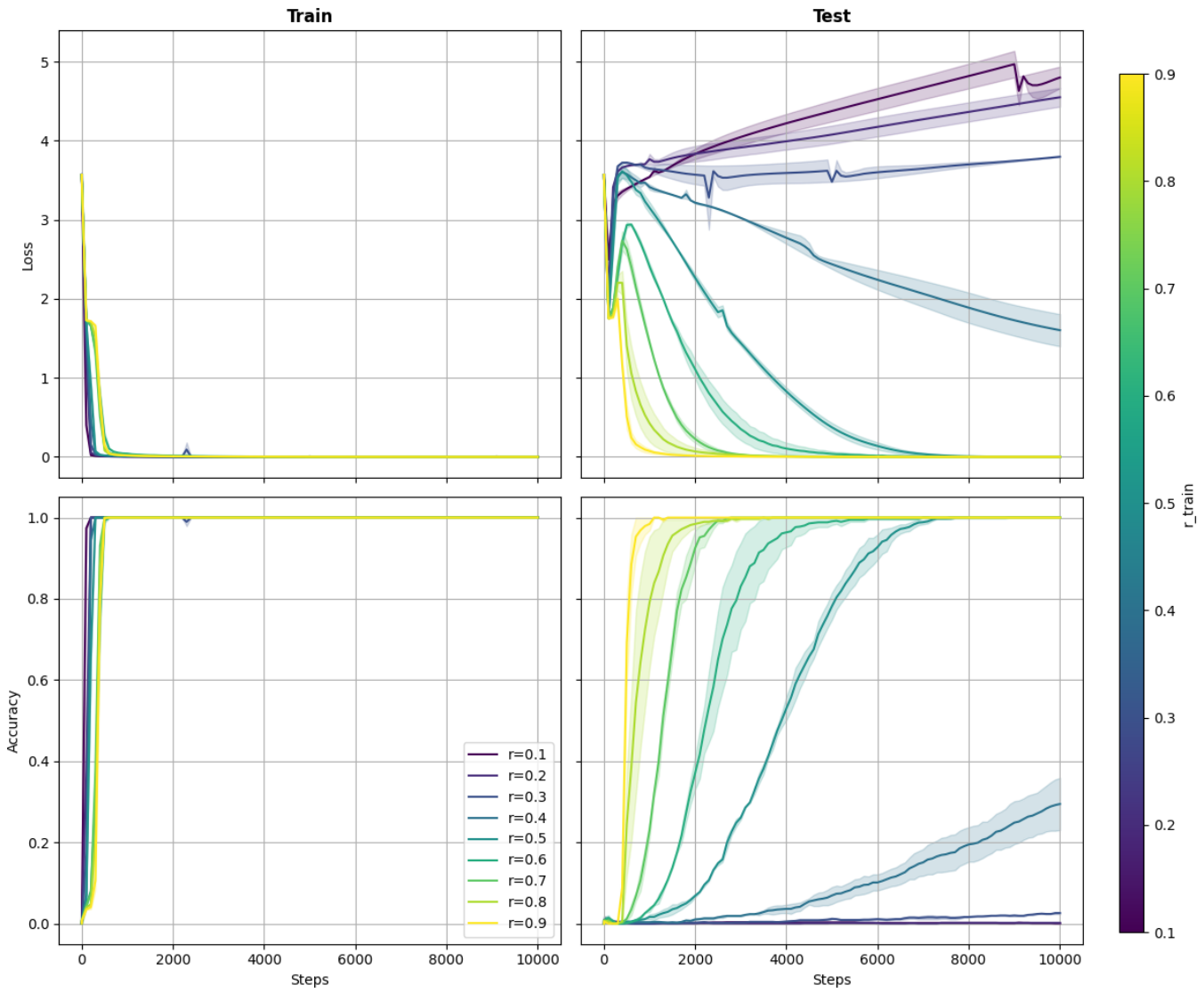

Figure 2: LSTM Model performance for different training fractions ($r_{\text{train}} \in \{0.1, \ldots, 0.9\}$). Top plots show $\mathcal{L}^{(t)}_{\text{train}}$ and $\mathcal{L}^{(t)}_{\text{val}}$, while bottom plots show $\mathcal{A}^{(t)}_{\text{train}}$ (with legend) and $\mathcal{A}^{(t)}_{\text{val}}$, all as functions of t. Curves are color-coded using the `viridis` colormap (with a color bar to the left mapping r_{train} values) and shaded areas indicate standard deviation.

GPT

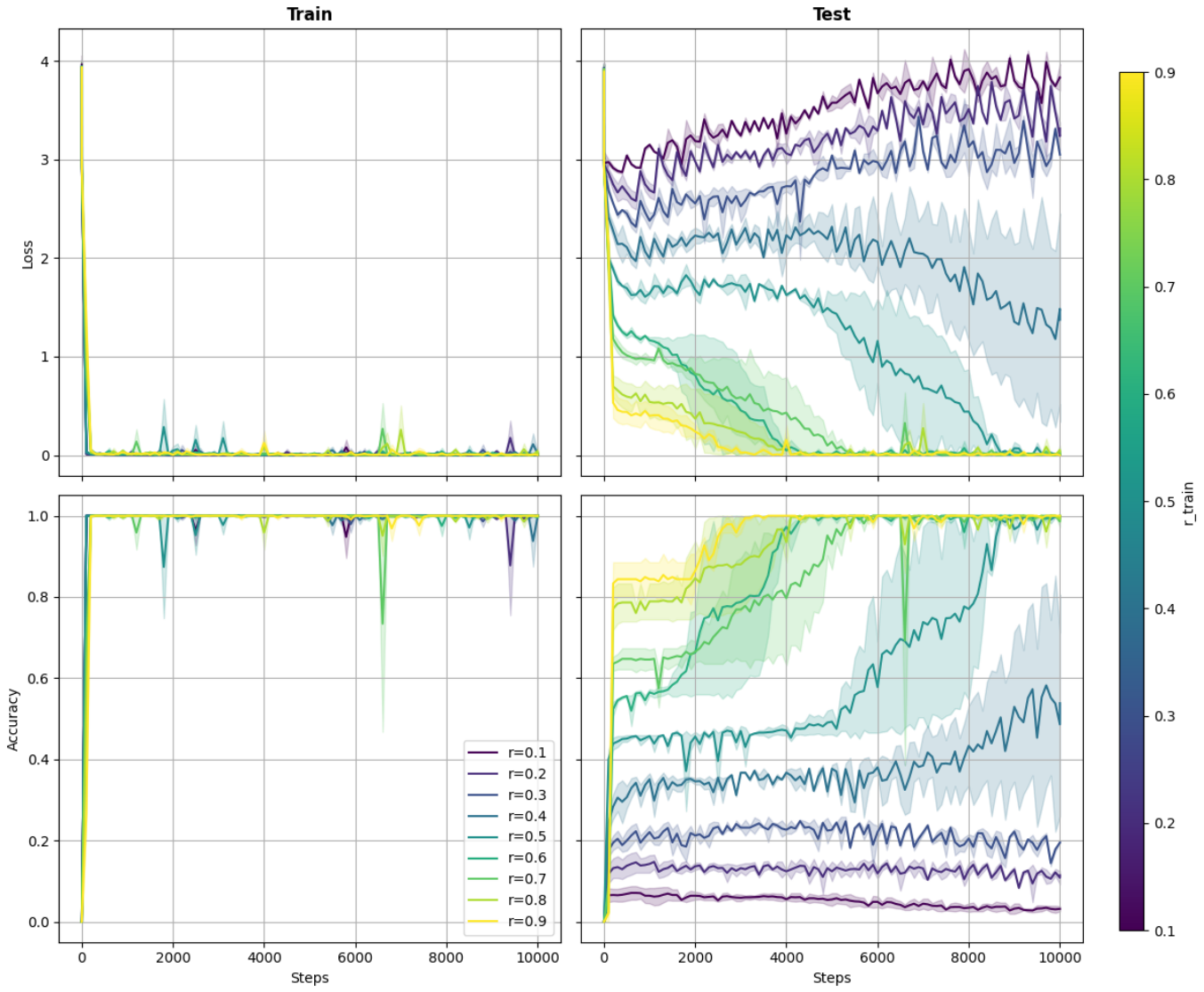

Figure 3: GPT performance for different training fractions ($r_{\text{train}} \in \{0.1, \ldots, 0.9\}$). Top plots show $\mathcal{L}^{(t)}_{\text{train}}$ and $\mathcal{L}^{(t)}_{\text{val}}$, while bottom plots show $\mathcal{A}^{(t)}_{\text{train}}$ (with legend) and $\mathcal{A}^{(t)}_{\text{val}}$, all as functions of t. Curves are color-coded using the `viridis` colormap (with a color bar to the left mapping r_{train} values) and shaded areas indicate standard deviation.

Problem 4.3.b

LSTM

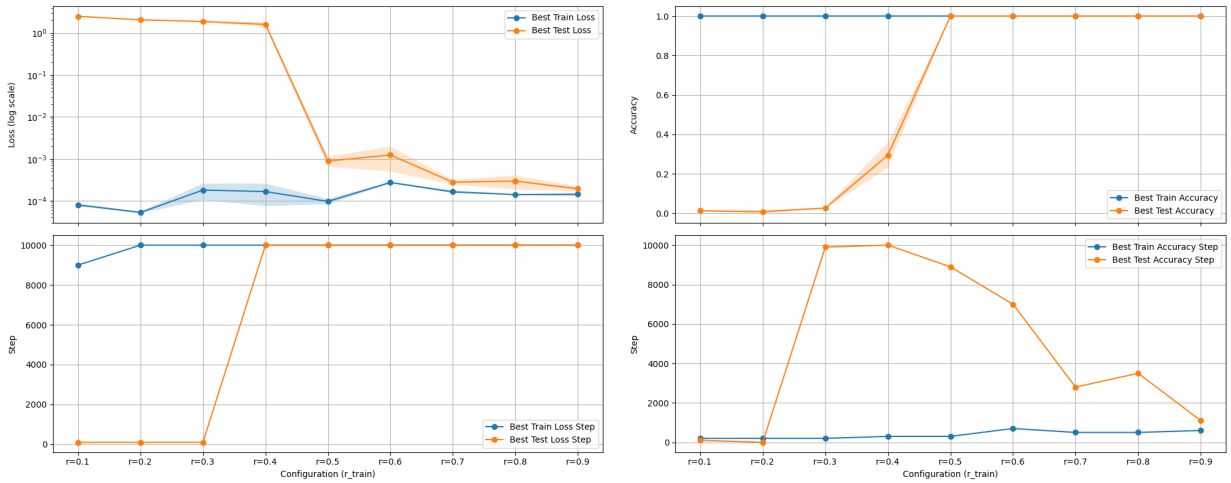

Figure 4: Comparative best performance metrics for the LSTM model. Left: best loss (train/test, log scale) and corresponding training steps across configurations. Right: best accuracy and corresponding steps. Shaded areas denote standard deviation.

GPT

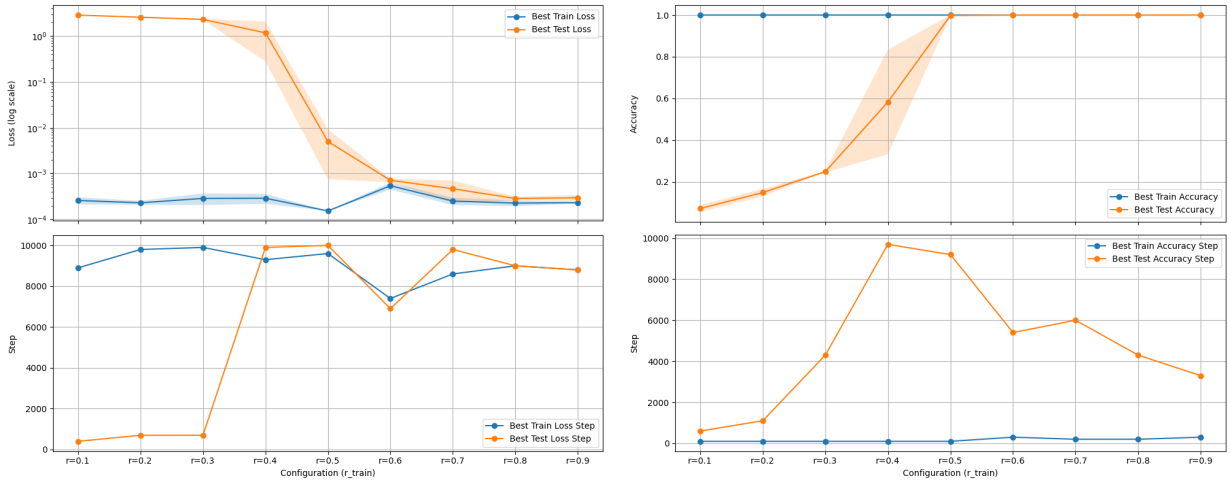

Figure 5: Comparative best performance metrics for the GPT model. Left: best loss (train/test, log scale) and corresponding training steps across configurations. Right: best accuracy and corresponding steps. Shaded areas denote standard deviation.

Problem 4.3.c

The experimental results show that the LSTM model scales better in terms of validation loss. As the training data fraction increases, the LSTM achieves a lower final $\mathcal{L}_{val}$. The training curves for LSTM are also less prone to instability, reflecting its simpler and more repetitive structure, which is well-suited to the straightforward nature of the task's data.

In contrast, the GPT model demonstrates a quicker improvement in its best validation accuracy at lower data ratios, indicating that it can generalize more effectively with less training data. However, the instability observed in the $\mathcal{A}_{val}^{(t)}$ curves suggests that the GPT's more complex structure may lead to overfitting and training instability.

Problem 4.3.d

The smallest training data fraction required for generalization is $r = 0.5$ for both models.

The largest r at which the model just overfits (i.e., $\mathcal{A}_{train} > 90\%$ and $\mathcal{A}_{val} < 50\%$) is **0.4** for the LSTM and **0.3** for the GPT.

There is only a small difference between the smallest r that allows generalization and the largest r where overfitting occurs, which is likely due to the simplicity and highly structured nature of the arithmetic dataset.

Problem 4.4: Experiment 3 (Binary and Ternary Operations)

Problem 4.4.a

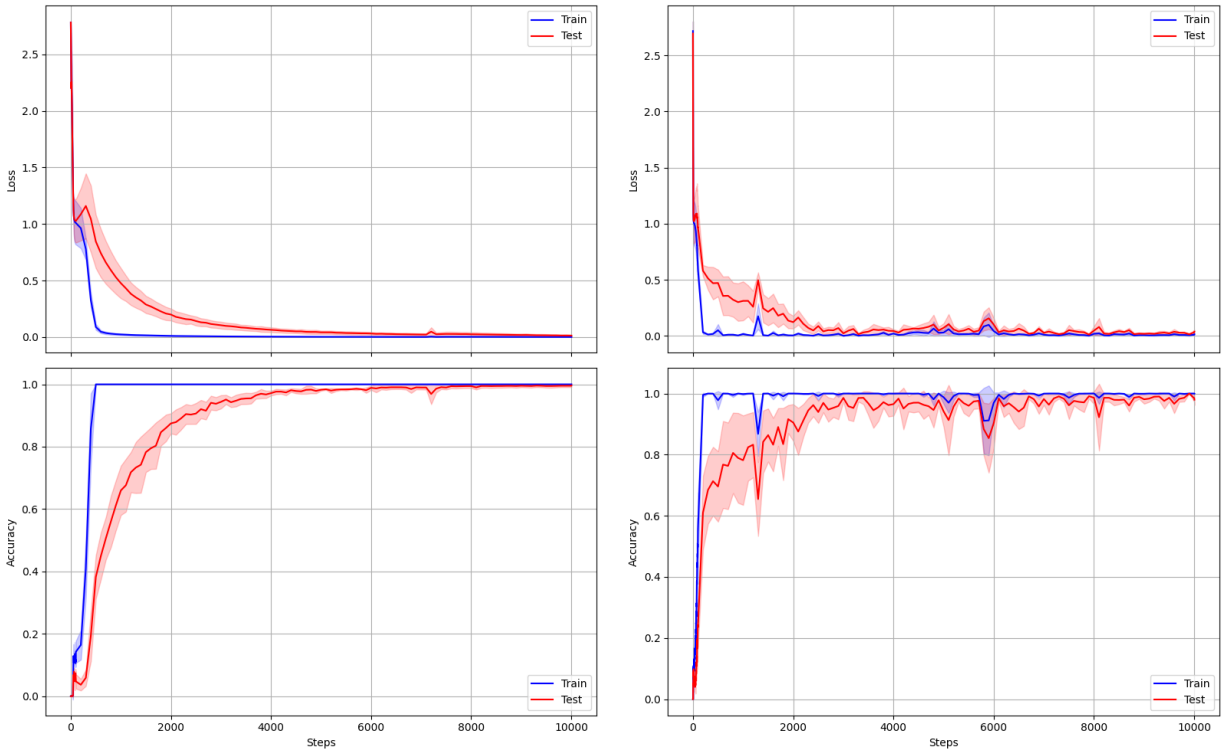

Figure 6: This figure presents a 2×2 grid of performance metrics versus training steps t. The top left panel shows $\mathcal{L}^{(t)}_{\text{train/val}}$ for the LSTM model (blue: train, red: test), and the bottom left displays $\mathcal{A}^{(t)}_{\text{train/val}}$ for LSTM. Similarly, the top right and bottom right panels present the loss and accuracy metrics for the GPT model. Shaded areas indicate standard deviation.

Problem 4.4.b

LSTM

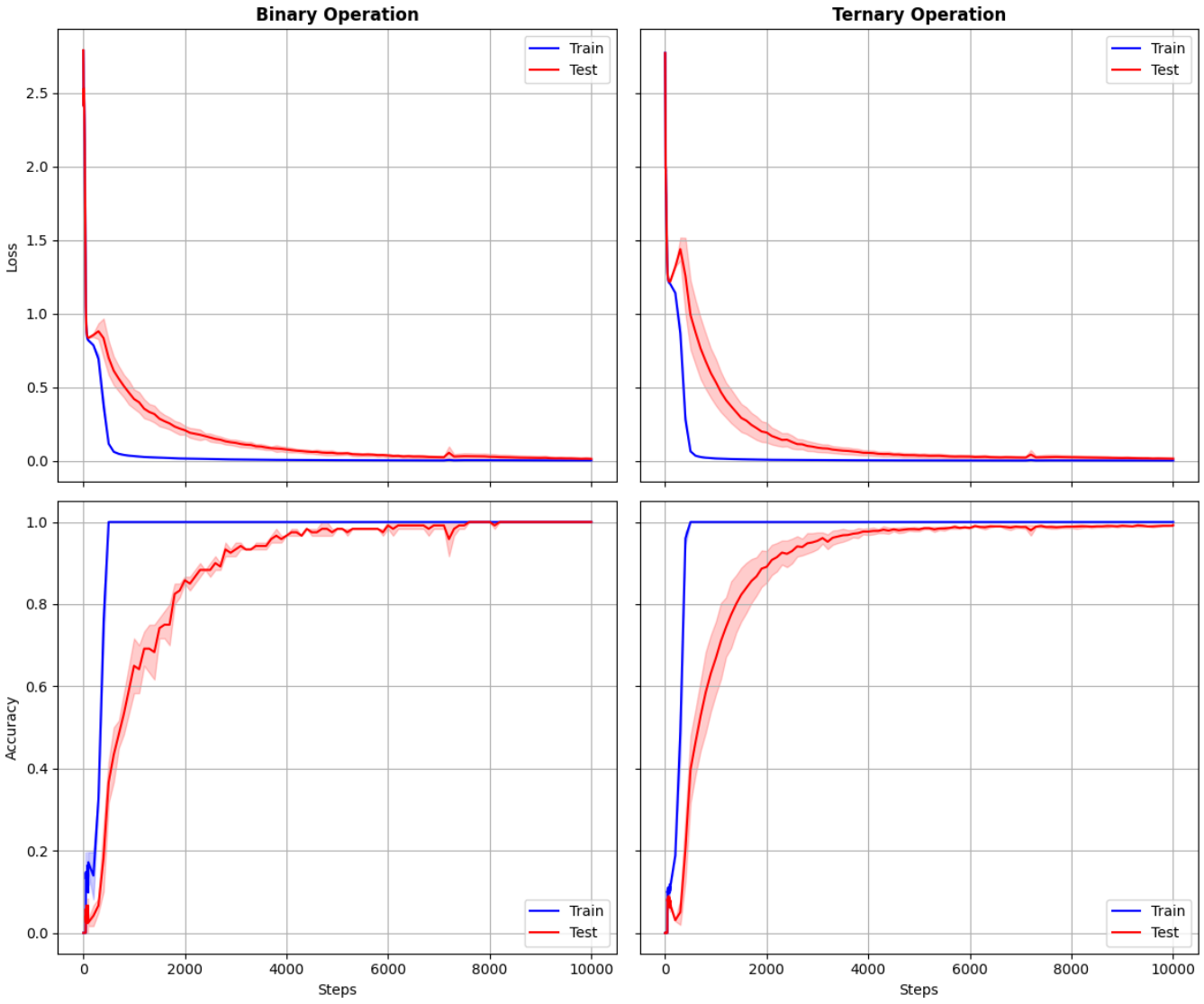

Figure 7: LSTM performance on binary (left column) and ternary (right column) operations. The top row plots $\mathcal{L}^{(t)}_{\text{train/val}}$ (blue: train, red: test), while the bottom row shows $\mathcal{A}^{(t)}_{\text{train/val}}$. Shaded areas represent standard deviation.

GPT

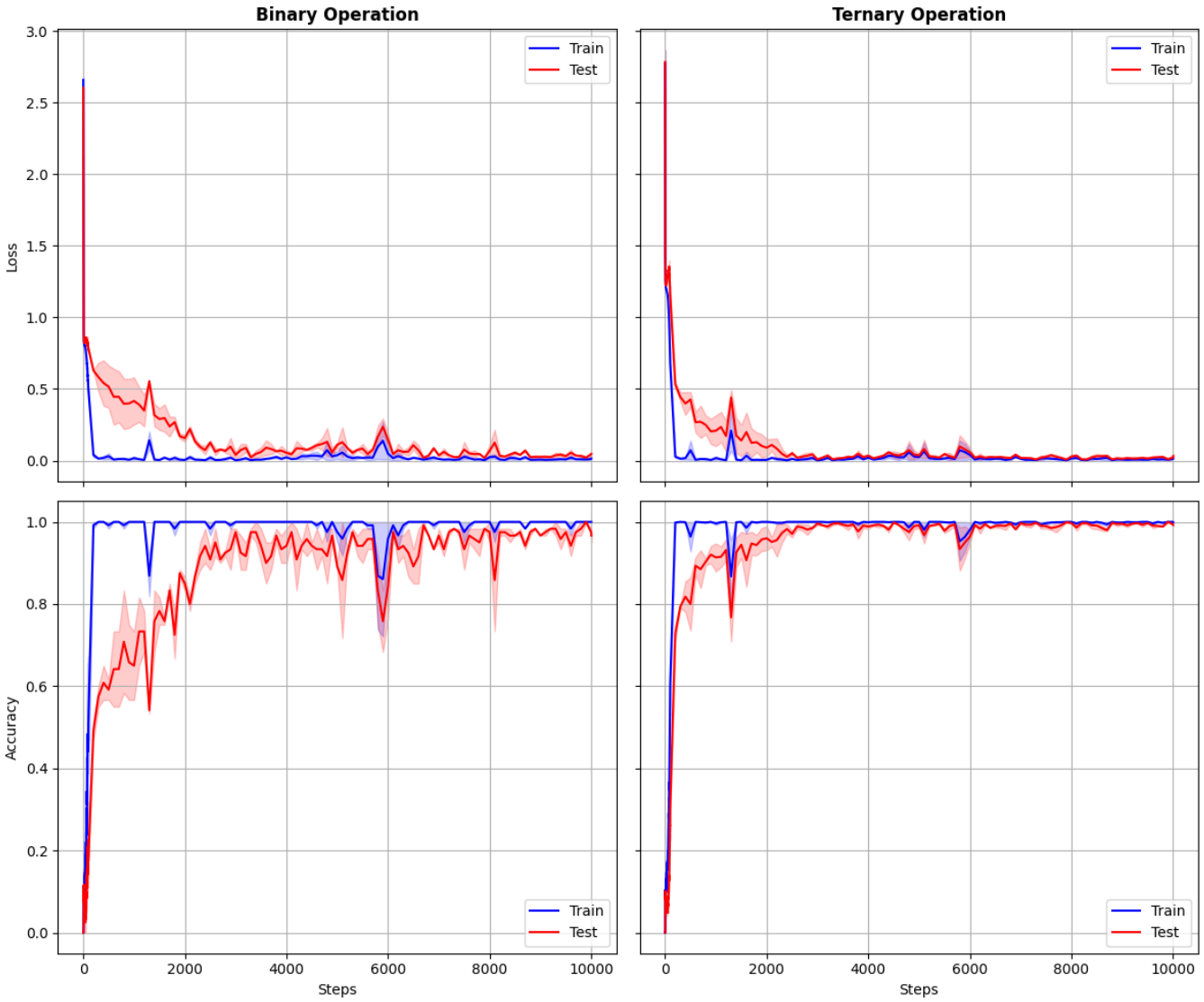

Figure 8: GPT performance on binary (left column) and ternary (right column) operations. The top row plots $\mathcal{L}^{(t)}_{\text{train/val}}$ (blue: train, red: test), while the bottom row shows $\mathcal{A}^{(t)}_{\text{train/val}}$. Shaded areas represent standard deviation.

Problem 4.4.c

My initial intuition was that the models would likely predict binary operations more quickly than ternary ones, as binary equations have a simpler structure with fewer tokens and less combinatorial complexity. This intuition was disproved by the experimental results showing that ternary operations actually converge more rapidly.

Even though the final accuracies are comparable, the experiments show that both models tend to overfit slightly more on binary operations, as evidenced by the more pronounced instability in the accuracy curves ($\mathcal{A}$) for binary data. In contrast, the ternary operations yield more stable accuracy curves and converge more rapidly, suggesting that the additional complexity in ternary equations provides a regularizing effect that facilitates faster generalization.

Problem 4.5: Experiment 4 (Scaling Model Size)

Problem 4.5.a

LSTM (L=1)

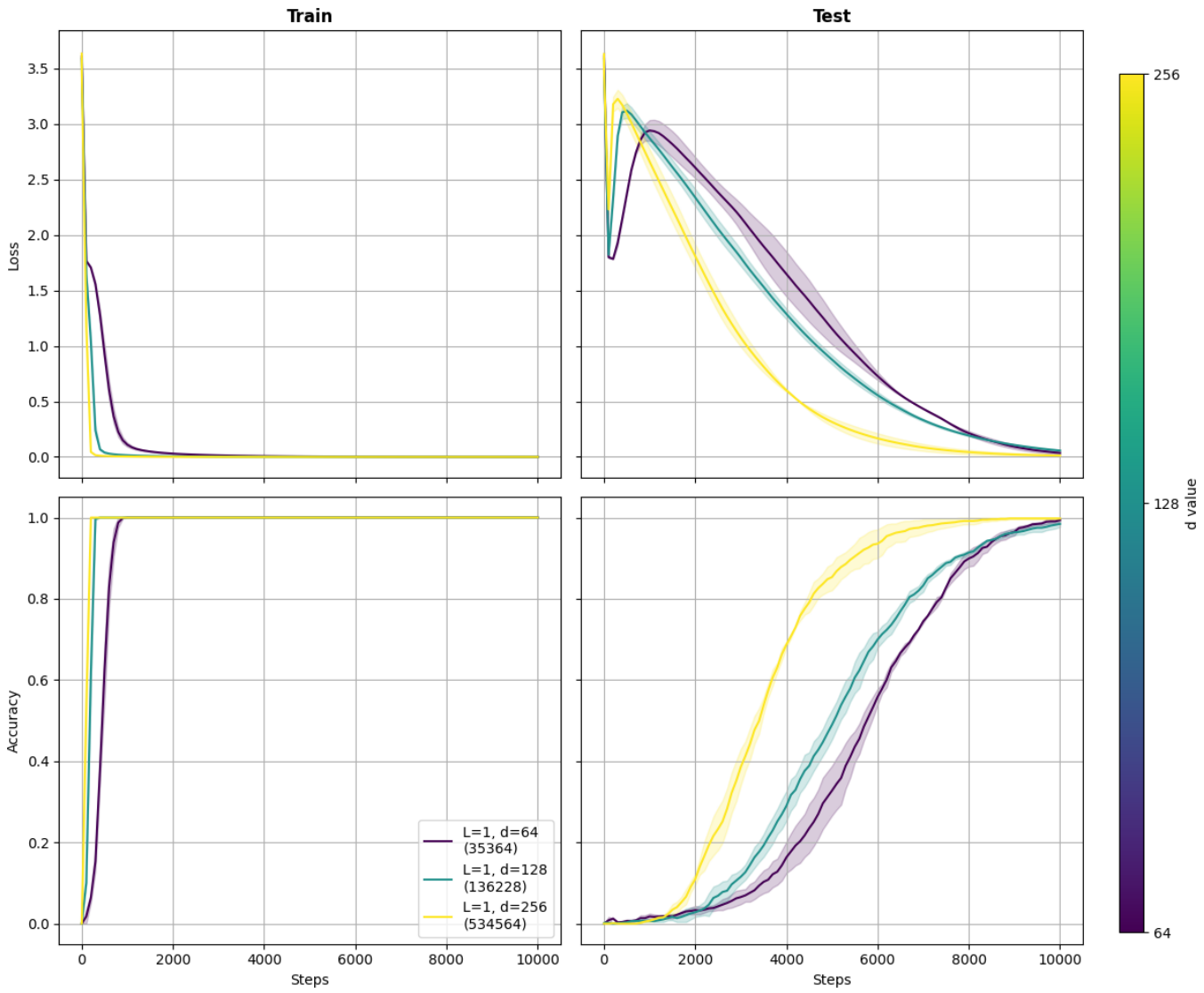

Figure 9: Model performance metrics for the LSTM model with $L = 1$. Plotted are $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ versus training steps t for different embedding dimensions d. Curves are color-coded by d on a $\log_2$ scale (see color bar), with shaded areas showing standard deviation.

LSTM (L=2)

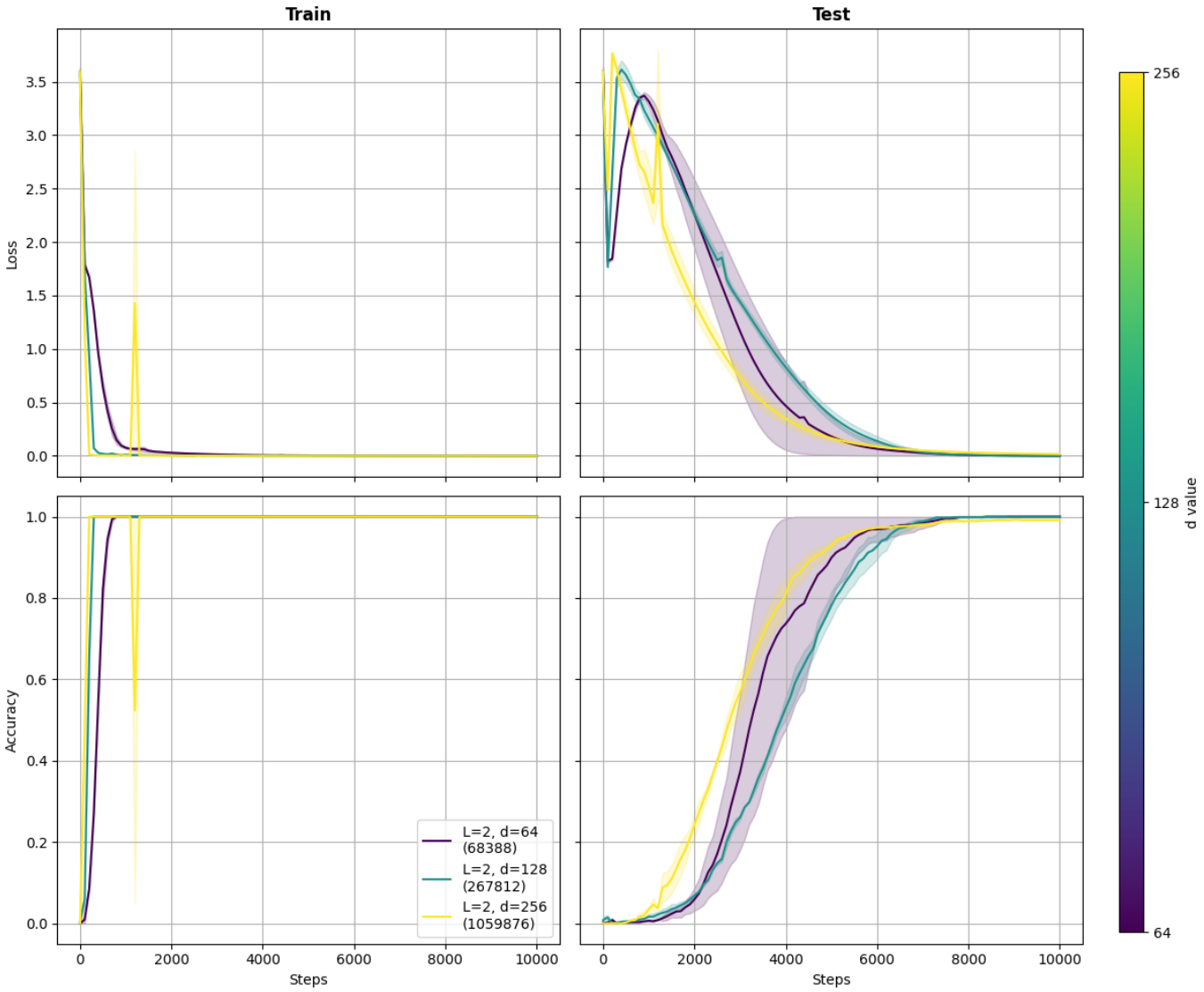

Figure 10: Model performance metrics for the LSTM model with $L = 2$. Plotted are $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ versus t for various d values. Curves are color-coded by d on a $\log_2$ scale (color bar provided) and shaded regions denote standard deviation.

LSTM (L=3)

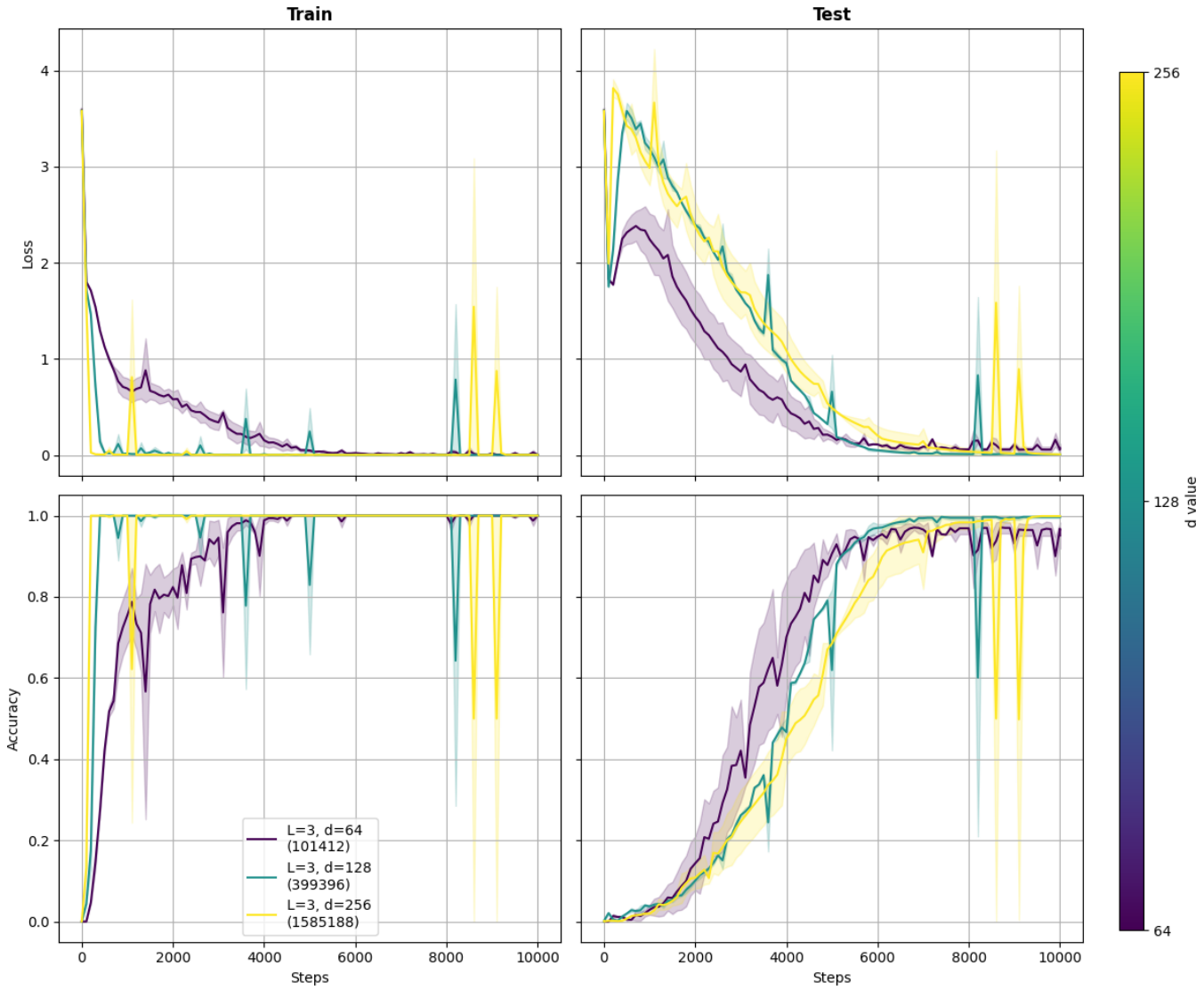

Figure 11: Model performance metrics for the LSTM model with $L = 3$. The plots show $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ as functions of t, with curves color-coded by embedding dimension d on a $\log_2$ scale (see color bar) and shaded areas indicating standard deviation.

GPT (L=1)

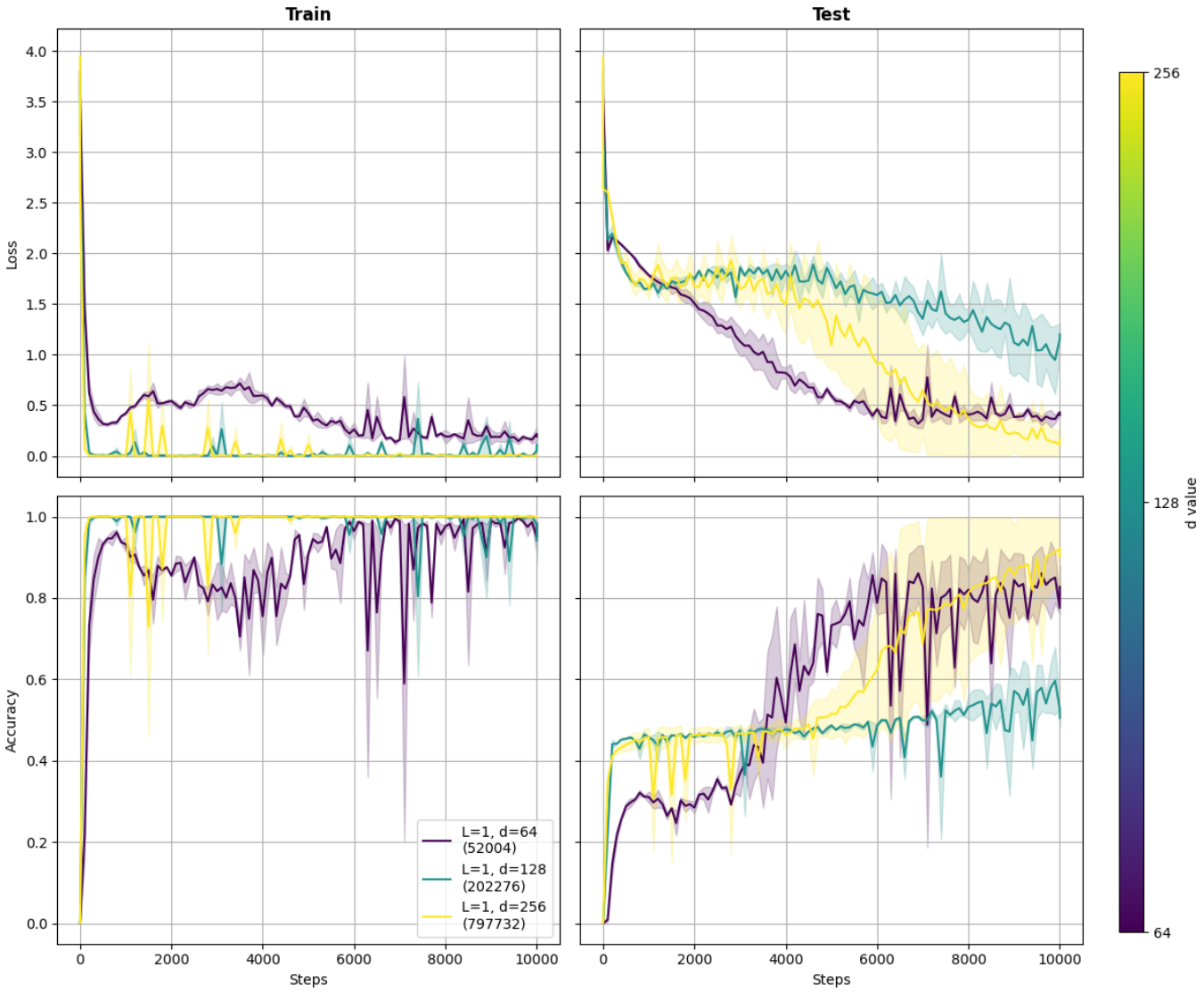

Figure 12: Model performance metrics for the GPT model with $L = 1$. Shown are $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ versus t for different d values, with curves color-coded by d on a $\log_2$ scale (see color bar) and shaded regions representing standard deviation.

GPT (L=2)

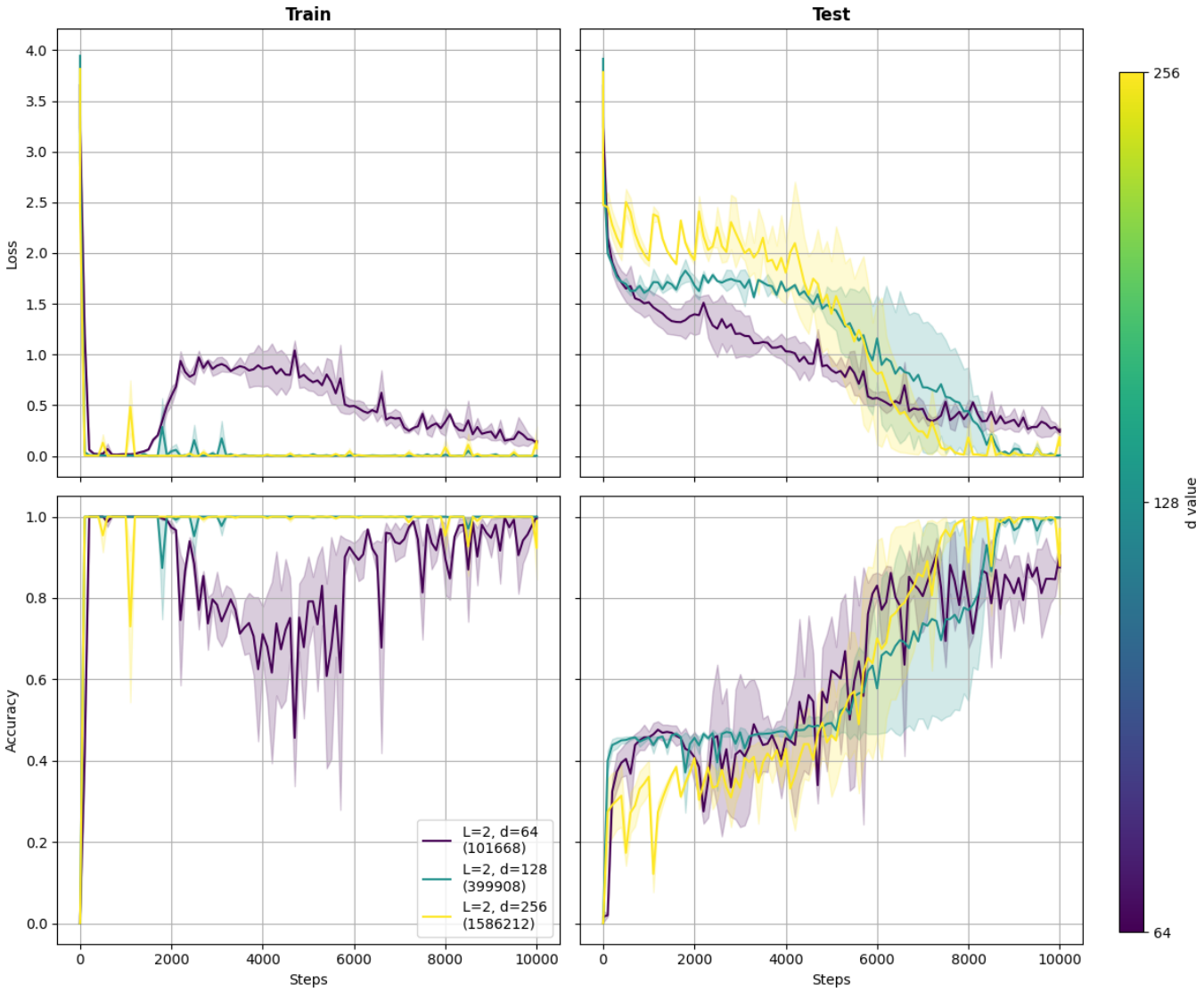

Figure 13: Model performance metrics for the GPT model with $L = 2$. The plots display $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ as functions of t for various d values. Colors ($\log_2$ scaled, per the color bar) denote d, with shaded areas indicating standard deviation.

GPT (L=3)

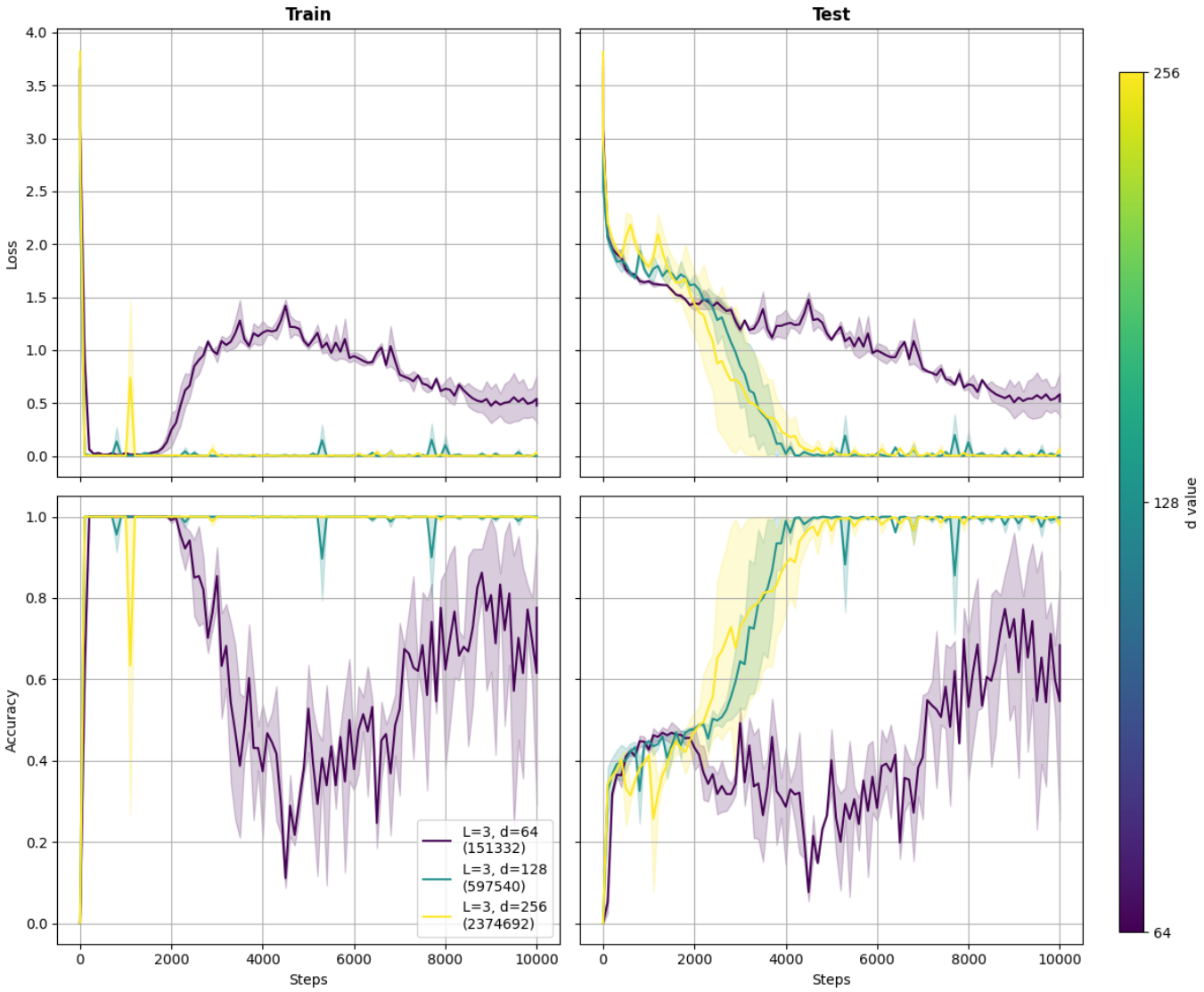

Figure 14: Model performance metrics for the GPT model with $L = 3$. Plots of $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ versus t are provided for different d values, with curves color-coded by d on a $\log_2$ scale (color bar) and standard deviation shown as shaded regions.

Problem 4.5.b: Best Metrics as a Function of (L, d)

LSTM

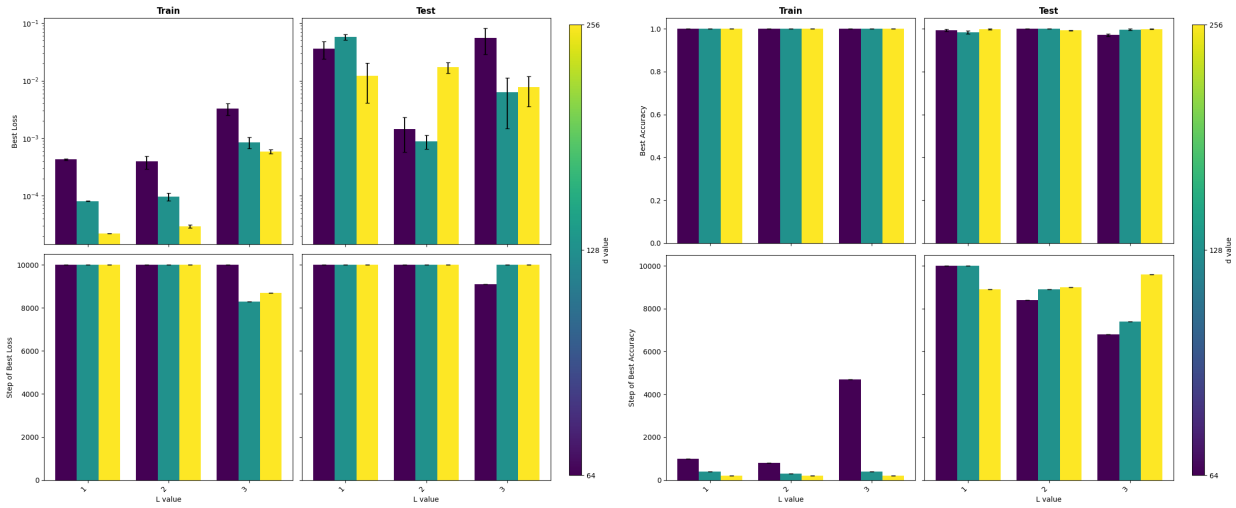

Figure 15: Comparative best performance metrics for the LSTM model as a function of L and d. The histogram subplots shows $\mathcal{L}_{\text{train/val}}$ (left) $\mathcal{A}_{\text{train/val}}$ (right) with L on the x-axis and different bars for each d. Colors are assigned using a $\log_2$ scale. With yerr indicating standard deviation.

GPT

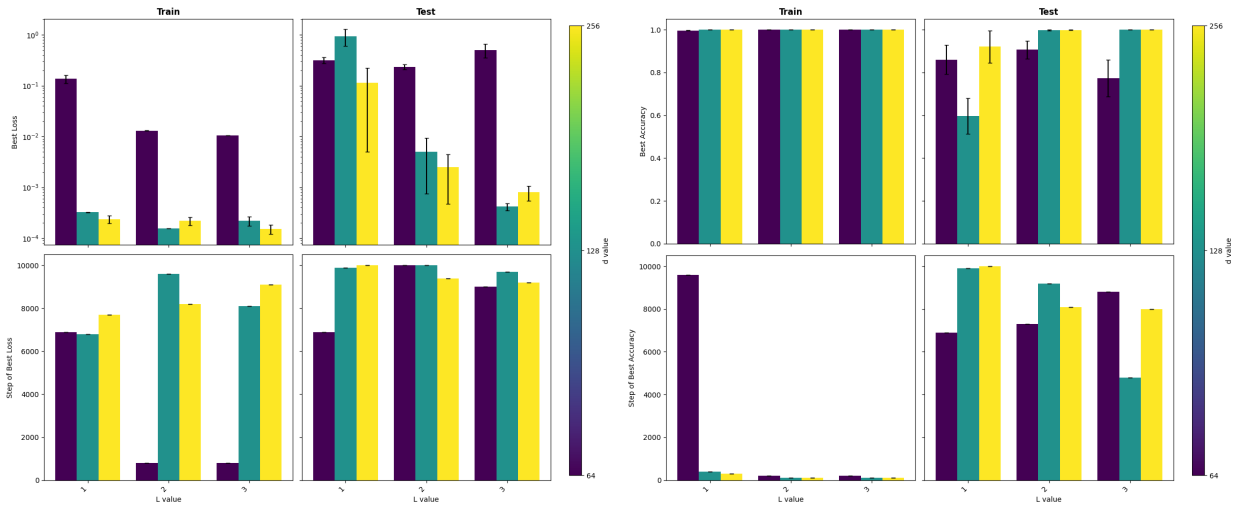

Figure 16: Comparative best performance metrics for the GPT model as a function of L and d. The histogram subplots shows $\mathcal{L}_{\text{train/val}}$ (left) $\mathcal{A}_{\text{train/val}}$ (right) with L on the x-axis and different bars for each d. Colors are assigned using a $\log_2$ scale. With yerr indicating standard deviation.

Problem 4.5.b: Best Metrics as a Function of the Number of Parameters

LSTM

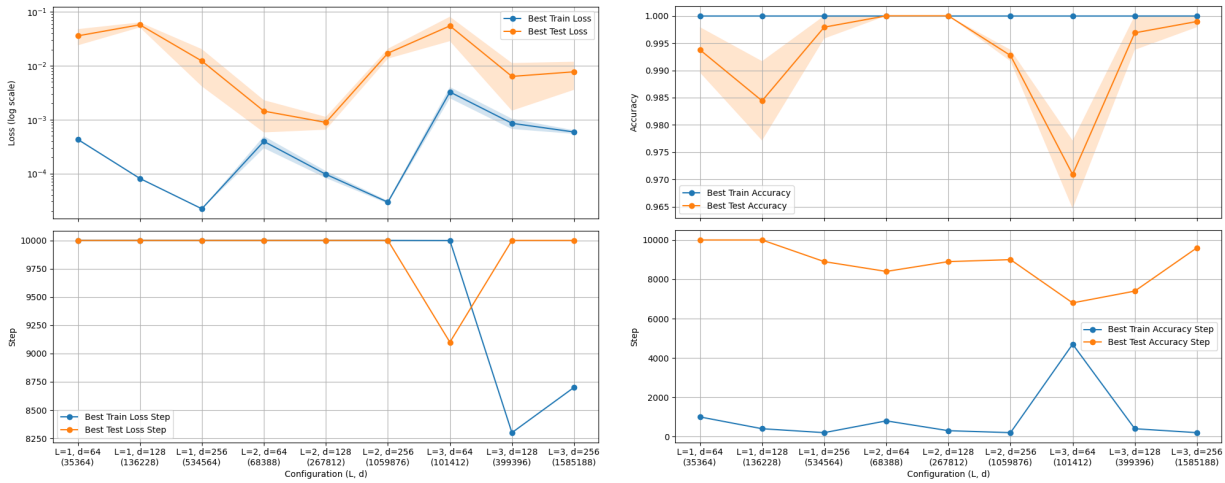

Figure 17: Comparative best performance metrics for the LSTM model plotted against the number of non-embedding parameters. The left panel shows best loss values and the right panel shows best accuracy values, with error bars indicating standard deviation.

GPT

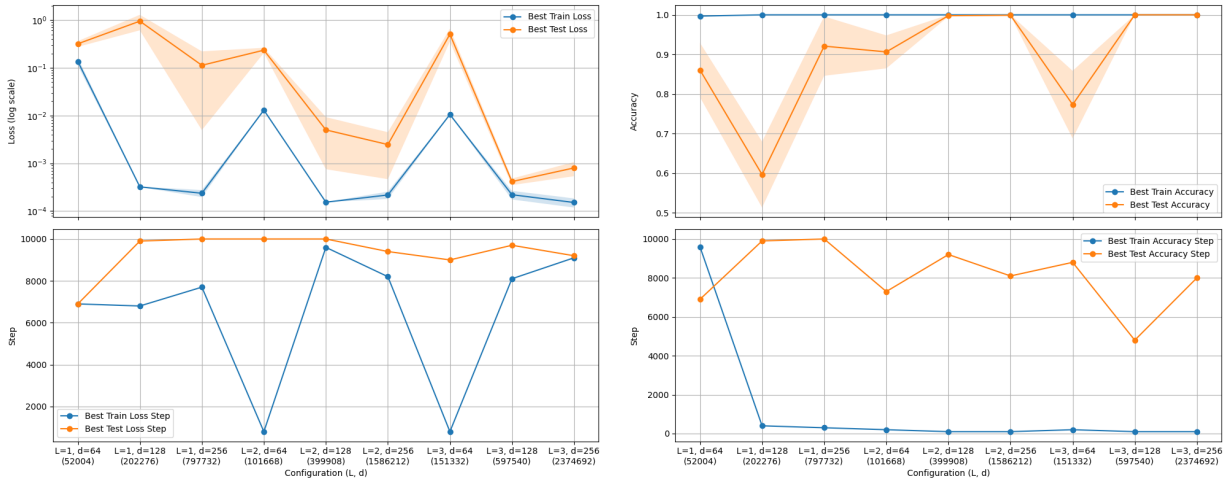

Figure 18: Comparative best performance metrics for the GPT model as a function of the number of non-embedding parameters. The left panel plots best loss values and the right panel plots best accuracy values versus parameter count, with error bars representing standard deviation.

Problem 4.5.c

Fixed L

For a fixed number of layers, as the hidden size d increases, the GPT model shows a clear improvement in validation performance: its best validation loss $\mathcal{L}_{val}$ decreases and its validation accuracy $\mathcal{A}_{val}$ improves. This is likely because its self-attention mechanism can better exploit high-dimensional representations.

In contrast, the LSTM model tends to exhibit an increase in its best $\mathcal{L}_{val}$ as d grows, although its $\mathcal{A}_{val}$ remains high; this suggests that while the LSTM may over-parameterize in terms of loss minimization, its classification performance is less affected by the increased capacity.

Fixed d

When the hidden size is held constant, increasing the number of layers L has divergent effects on the two architectures. The LSTM becomes increasingly unstable and shows a higher $\mathcal{L}_{val}$ with additional layers; nevertheless, it manages to maintain a stable high $\mathcal{A}_{val}$ across configurations.

Conversely, the GPT model benefits from additional depth, as evidenced by a lower $\mathcal{L}_{val}$ and improved $\mathcal{A}_{val}$, indicating that the transformer architecture can more effectively leverage extra layers to learn robust representations.

Conclusion

Overall, the experimental data suggest that the GPT model scales better with the number of parameters, particularly in terms of $\mathcal{L}_{val}$. In contrast, the LSTM's performance degrades as its size increases, although it manage to maintain a high $\mathcal{A}$.

Problem 4.6: Experiment 5 (Scaling Compute)

Problem 4.6.a

LSTM

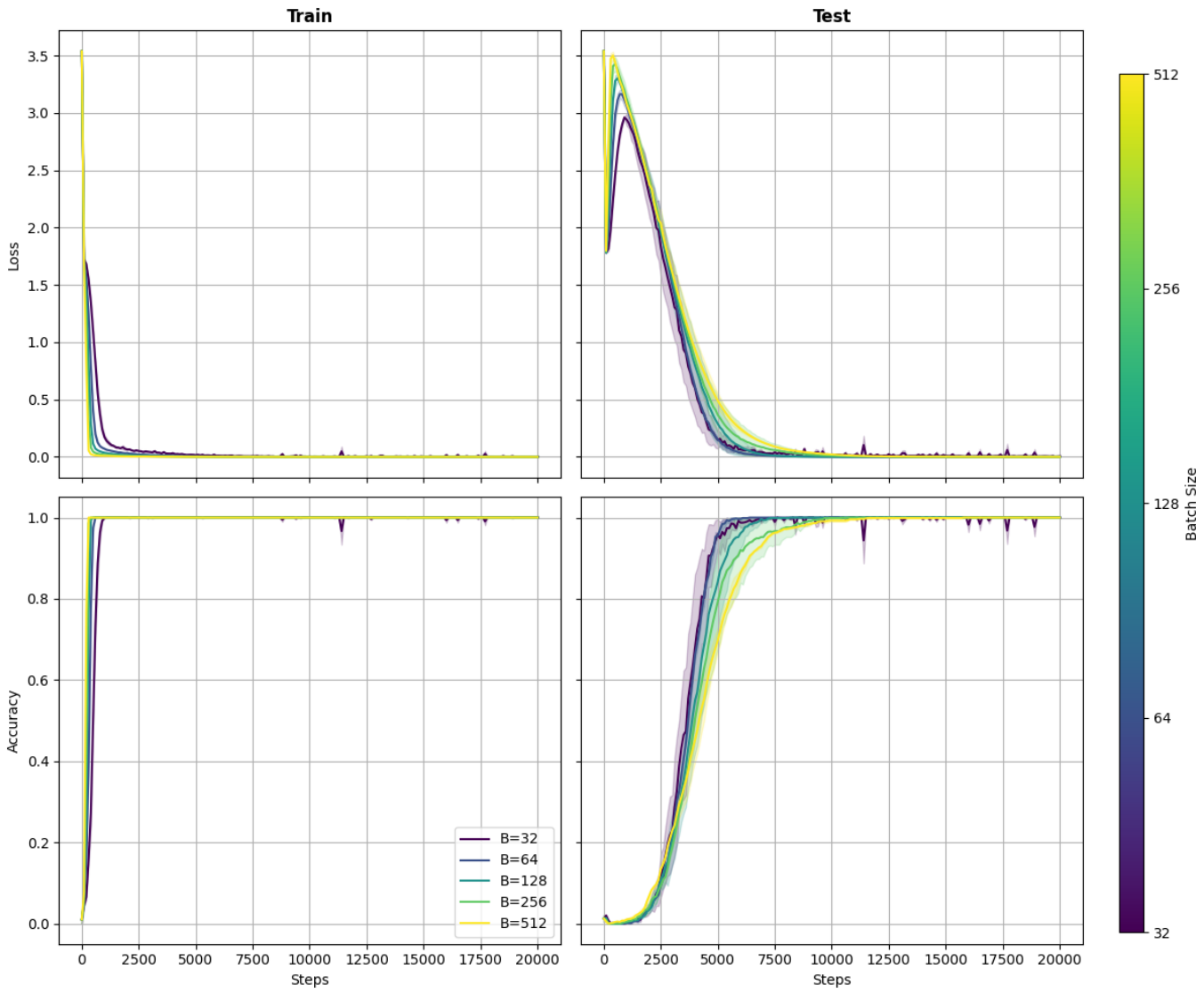

Figure 19: Performance metrics for the LSTM model. These 2×2 plots show $\mathcal{L}^{(t)}_{\text{train}}$ (top left), $\mathcal{L}^{(t)}_{\text{val}}$ (top right), $\mathcal{A}^{(t)}_{\text{train}}$ (bottom left) and $\mathcal{A}^{(t)}_{\text{val}}$ (bottom right) for various batch sizes $B \in \{2^5, 2^6, 2^7, 2^8, 2^9\}$. Curves are color-coded by B on a $\log_2$ scale (see color bar), and shaded regions indicate standard deviation.

GPT

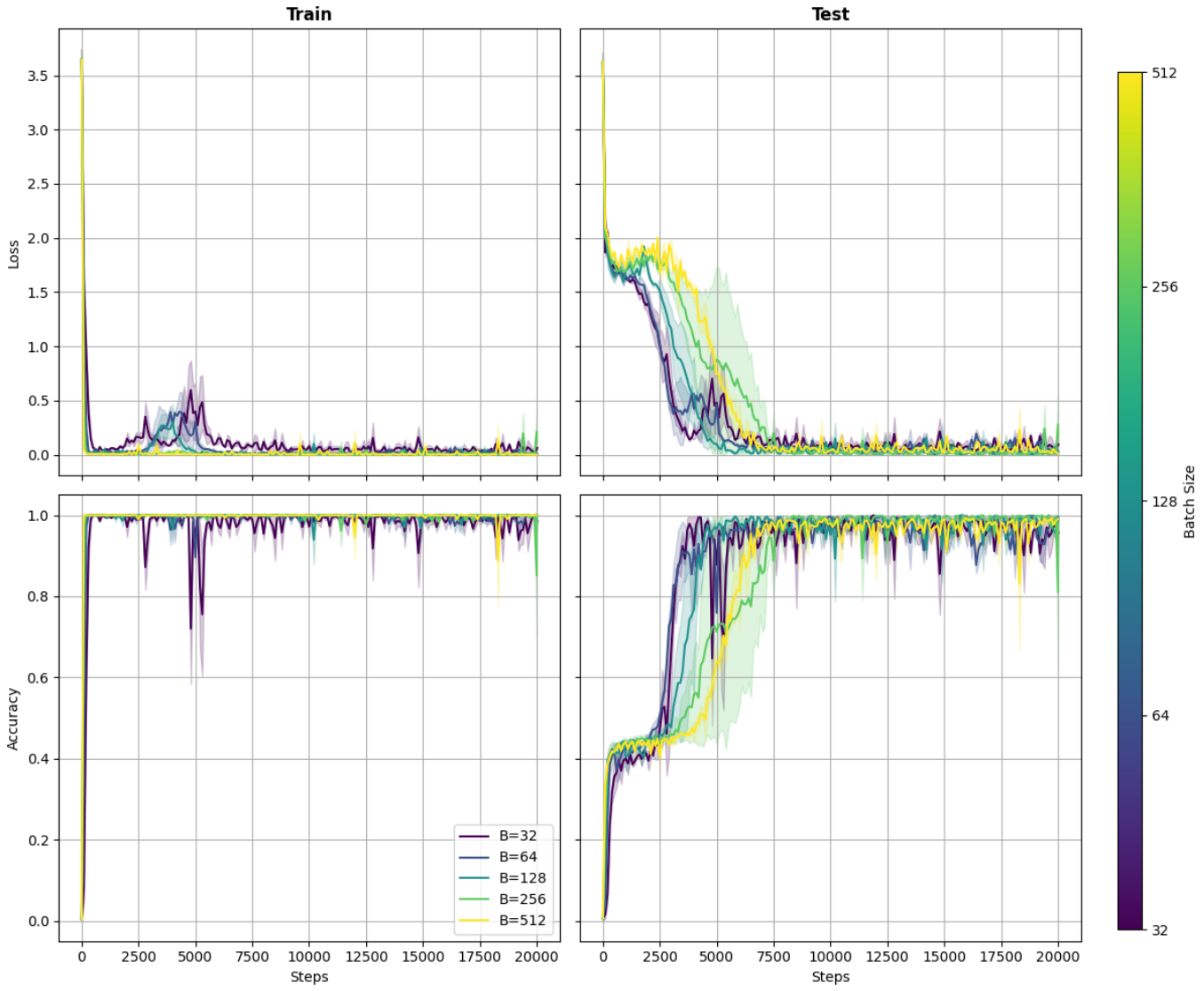

Figure 20: Performance metrics for the GPT model. These 2×2 plots show $\mathcal{L}_{\text{train}}^{(t)}$ (top left), $\mathcal{L}_{\text{val}}^{(t)}$ (top right), $\mathcal{A}_{\text{train}}^{(t)}$ (bottom left) and $\mathcal{A}_{\text{val}}^{(t)}$ (bottom right) for various batch sizes $B \in \{2^5, 2^6, 2^7, 2^8, 2^9\}$. Curves are color-coded by B on a $\log_2$ scale (see color bar), and shaded regions indicate standard deviation.

Problem 4.6.b

LSTM

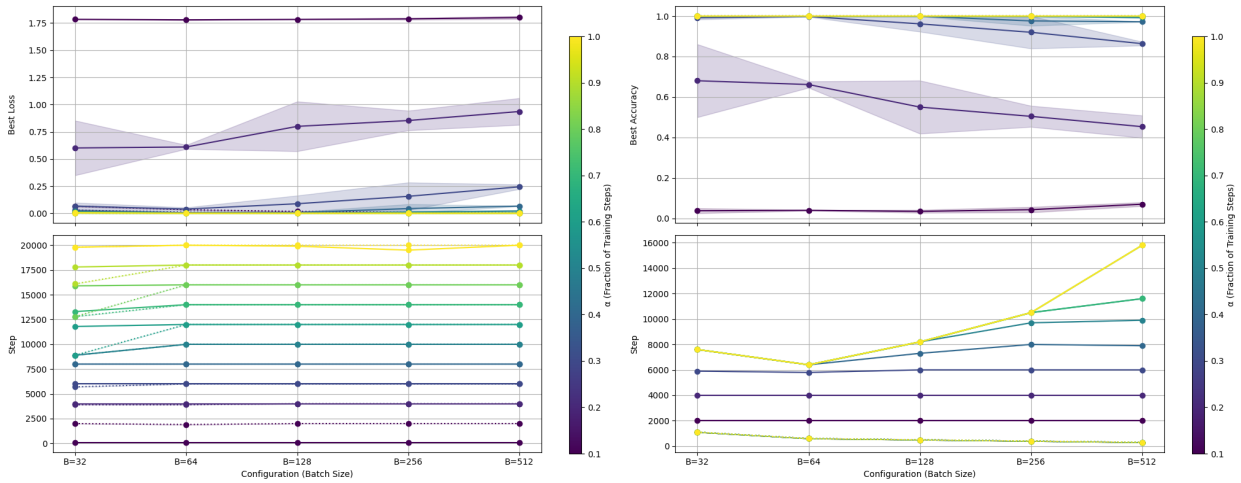

Figure 21: Best performance metrics for the LSTM model from data sliced at $\alpha \in \{0.1, \ldots, 1.0\}$. Left: best loss ($\mathcal{L}_{\text{train/val}}$) and $t_f(\mathcal{L})$ (train: dotted, val: solid). Right: best accuracy ($\mathcal{A}_{\text{train/val}}$) and $t_f(\mathcal{A})$ (same style). Curves are plotted versus configuration with colors (and color bar) denoting α.

GPT

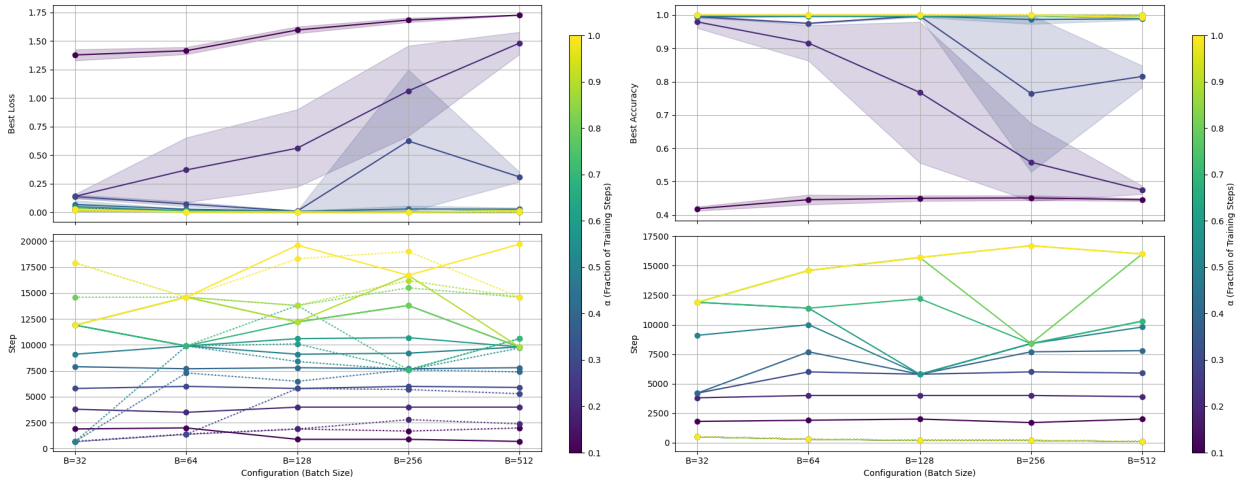

Figure 22: Best performance metrics for the GPT model from data sliced at $\alpha \in \{0.1, \ldots, 1.0\}$. Left: best loss ($\mathcal{L}_{\text{train/val}}$) and $t_f(\mathcal{L})$ (train: dotted, val: solid). Right: best accuracy ($\mathcal{A}_{\text{train/val}}$) and $t_f(\mathcal{A})$ (same style). Curves are plotted versus configuration with colors (and color bar) denoting α.

Problem 4.6.c

Increasing α consistently improves both $\mathcal{L}_{val}$ and $\mathcal{A}_{val}$ until overfitting sets in, while the effect of B is more nuanced. Overall, it's impact on generalization is a balance between gradient stability and the risk of converging to sharp, less generalizable minima. Notably, the GPT model is particularly sensitive to batch size, as its deep self-attention architecture benefits from the gradient noise of smaller batches to avoid converging to sharp minima.

Validation Loss ($\mathcal{L}_{val}$)

Larger batch sizes, despite providing stable gradient estimates, require a higher α (i.e., longer effective training) to minimize the loss, since very large batches may converge to sharper, less robust minima that generalize poorly.

Validation Accuracy ($\mathcal{A}_{val}$)

Conversely, smaller batch sizes can help the model escape sharp minima and quickly boost accuracy with a lower compute budget, although their convergence may be noisier overall.

Problem 4.7: Experiment 6 (Regularization)

Problem 4.7.a

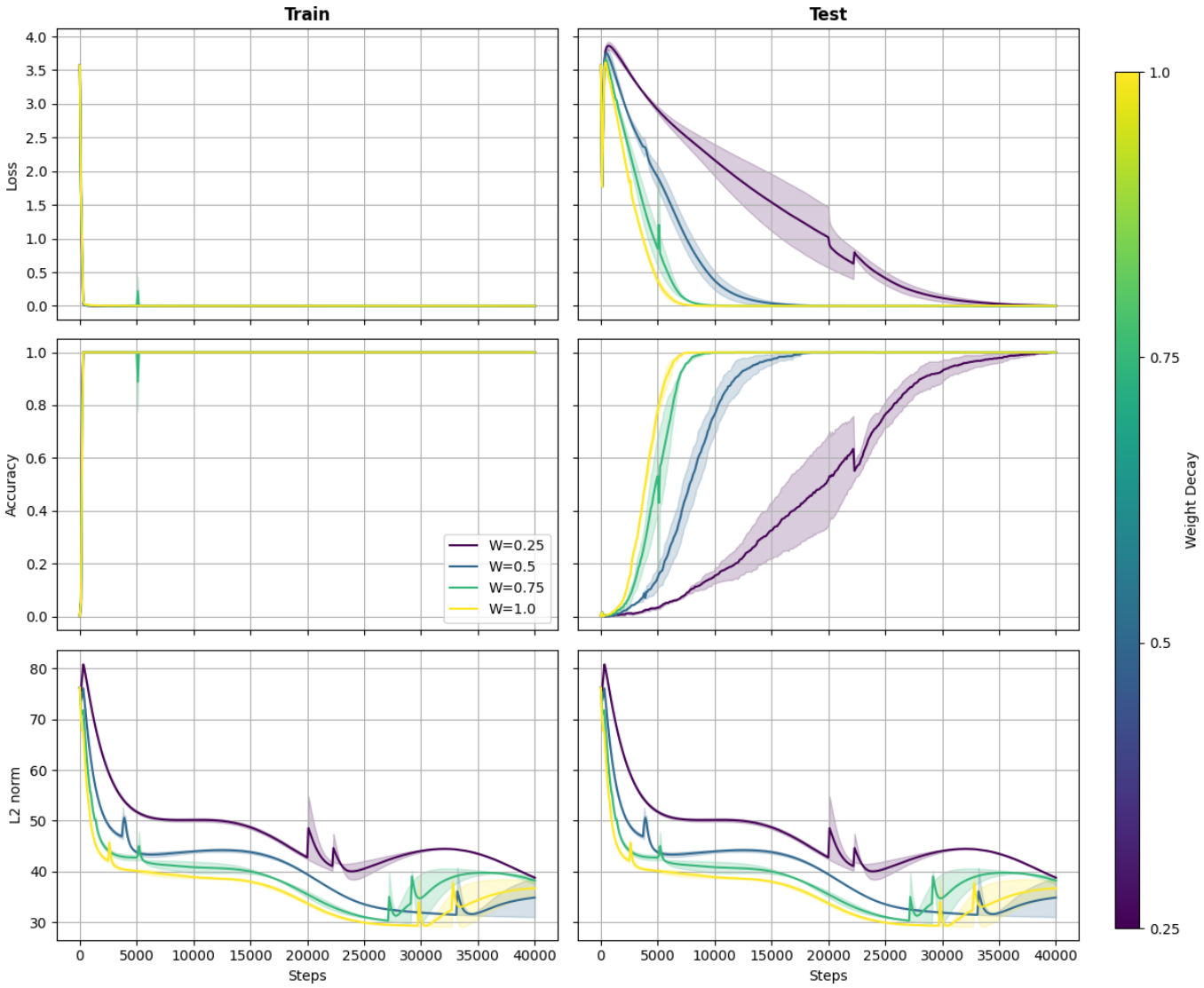

Figure 23: This 3×2 figure displays $\mathcal{L}^{(t)}_{\text{train/val}}$, $\mathcal{A}^{(t)}_{\text{train/val}}$, and the ℓ_2 norm $\|\theta^{(t)}\|_2$ of model parameters $\theta(t)$. The left column corresponds to training metrics and the right column to validation metrics, with rows showing loss (top), accuracy (middle), and parameter norm (bottom). Note that the parameter norm is duplicated for both train and validation sets solely for comparison with the other metrics. Curves are color-coded by weight decay $\in \{0.25, 0.5, 0.75, 1.0\}$ (see the color bar), and shaded areas indicate standard deviation.

Problem 4.7.b

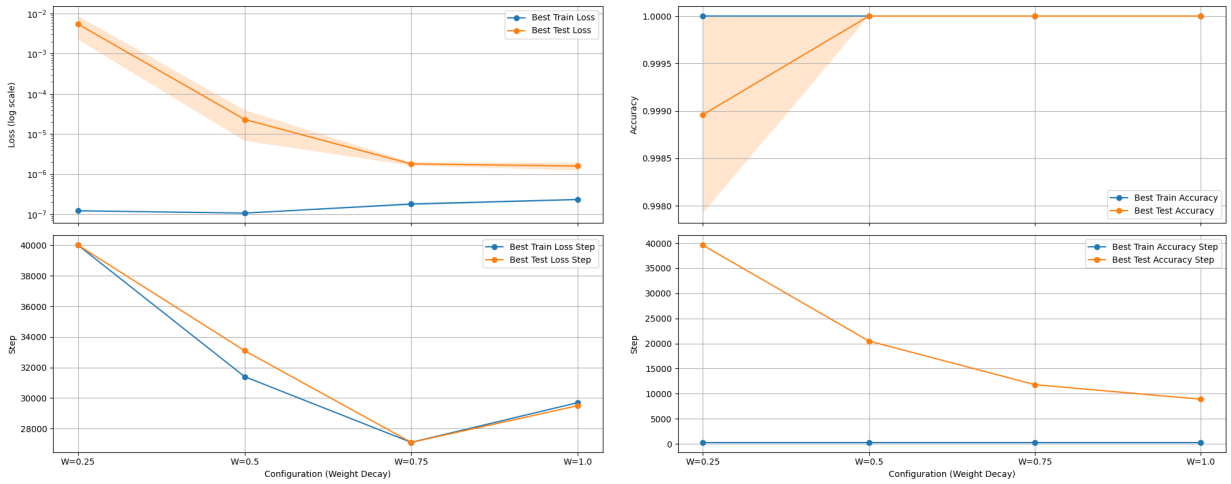

Figure 24: Comparative best performance metrics for the LSTM model as a function of weight decay. Left: best loss values $\mathcal{L}_{\text{train/val}}$ (on a log scale) and corresponding $t_f(\mathcal{L})$. Right: best accuracy values $\mathcal{A}_{\text{train/val}}$ and corresponding $t_f(\mathcal{A})$.

Problem 4.7.c

As the weight decay increases, the validation loss $\mathcal{L}_{val}$ drops faster and reaches a lower minimum, while the validation accuracy $\mathcal{A}_{val}$ improves earlier. In other words, both $t_f(\mathcal{L}_{val})$ and $t_f(\mathcal{A}_{val})$—the training steps at which the best $\mathcal{L}_{val}$ and $\mathcal{A}_{val}$ are first achieved—are lower with higher weight decay. This tells us that higher weight decay imposes stronger regularization, forcing the model to learn more robust features sooner, which leads to earlier improvements.In contrast

In contrast, with a lower weight decay (e.g., 0.25), the model takes longer to reach a good $\mathcal{L}_{val}$ and $\mathcal{A}_{val}$. This suggests that with a lower weight decay, the model is less constrained early on and may tend to overfit the training data, causing $\mathcal{L}_{val}^{(t)}$ to improve more slowly.

Problem 4.7.d

Before the model starts to generalize, it tends to fit the training data by increasing the magnitudes of its weights—so the ℓ_2 norm of the parameters is relatively high. Then, once the model begins to generalize well, the ℓ_2 norm starts decreasing and stabilizes at a low value. This happens because the regularization (weight decay) forces the weights to be smaller and discourages overly complex solutions, which in turn helps the model generalize better.

Then, as the model begins to generalize, there is a noticeable drop in the norm, indicating that the regularizer is forcing the weights to shrink and thus encouraging simpler, more robust representations that generalize better.

We can observe spikes in the ℓ_2 norm of the parameters, this is explicitly the weight decay in action. At those moments, the model begins to assign large weight values in an attempt to overfit the training data, which also causes the $\mathcal{L}_{val}^{(t)}$ and $\mathcal{A}_{val}^{(t)}$ to crash. The model is then immediately penalized by the regularization (specifically, the weight decay), effectively stomping the spike and guiding the model back onto a good generalization path.

Problem 4.8: Interpretability

Problem 4.8.a

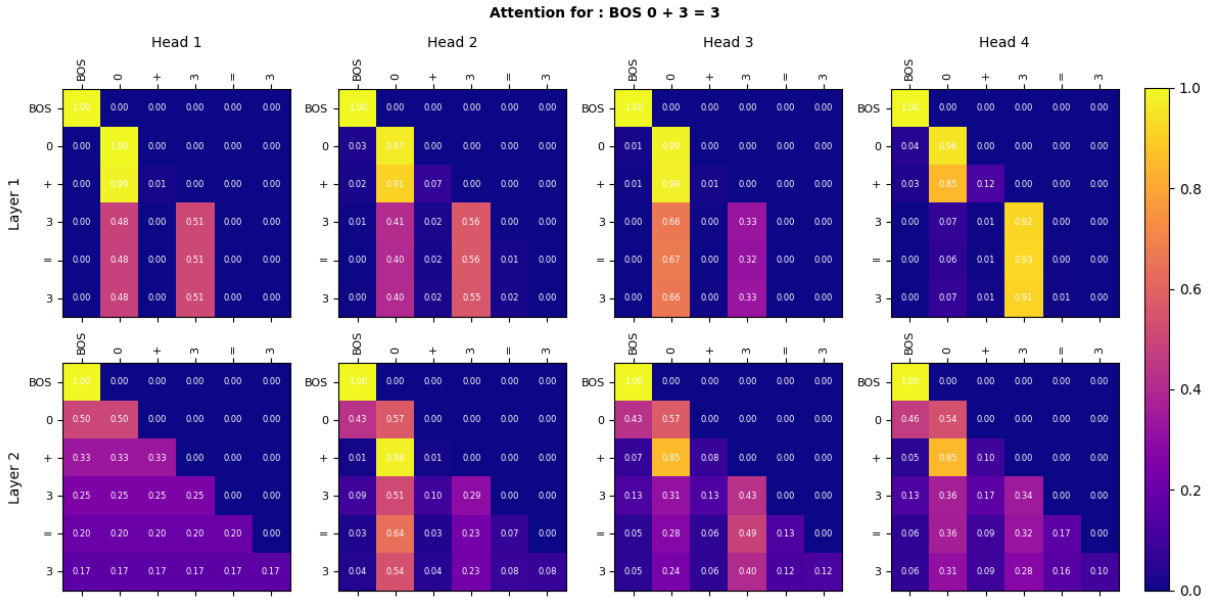

Figure 25: Attention heatmaps for the GPT from experiment 1. The grid displays a separate heatmap for each attention head across the layers. Rows correspond to different layers (with Layer 1 at the top), while columns correspond to different heads (Head 1, Head 2, etc.). Each cell shows a heatmap of size $S \times S$, where S is the sequence length, with tokens (obtained via the tokenizer) labeling both axes. The color intensity represents the attention weight (ranging from 0 to 1), and numerical values overlaid on the heatmap indicate the exact weight in each cell.

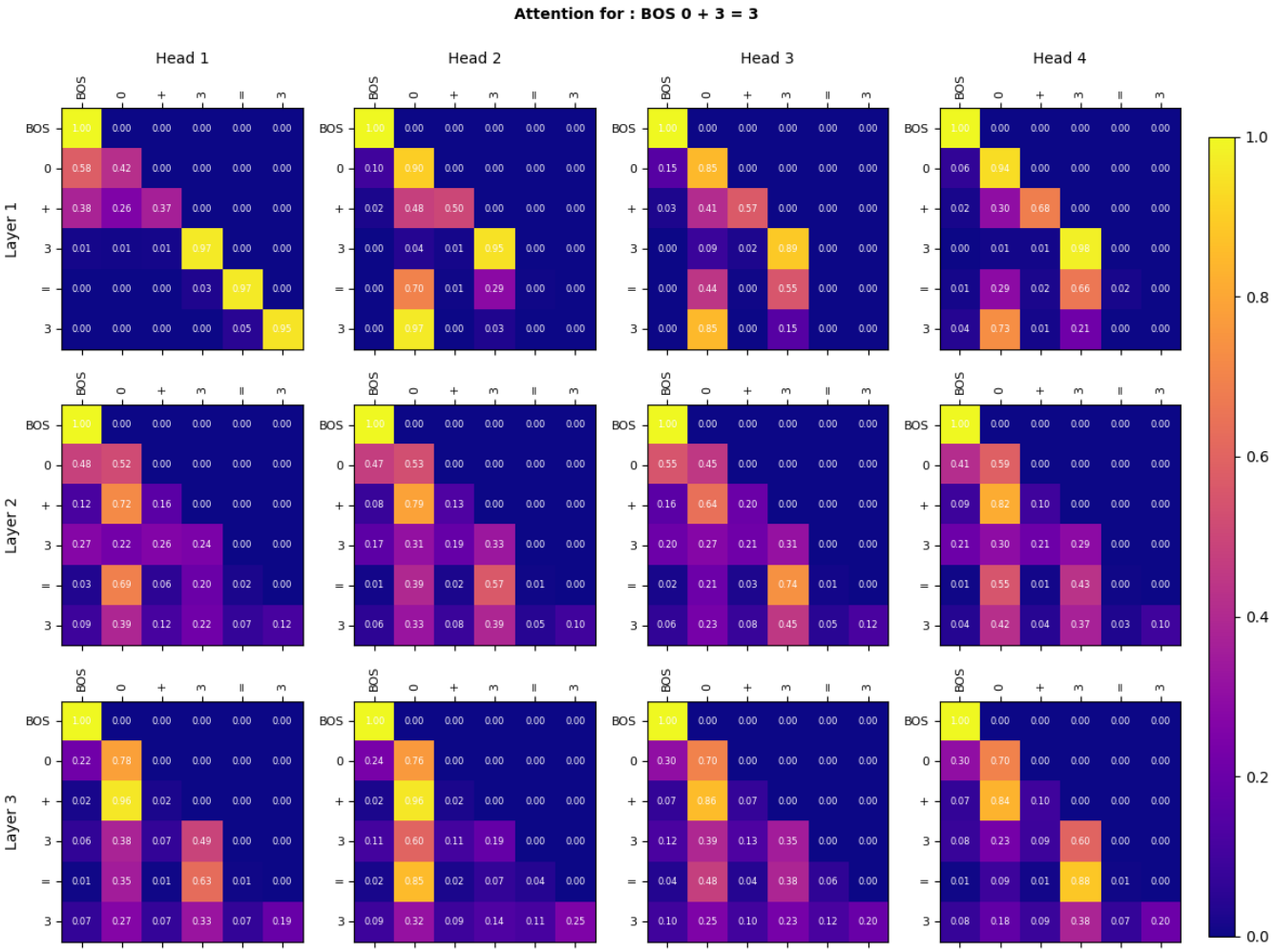

Figure 26: Attention heatmaps for the GPT model from experiment 3. This model was trained on both binary and ternary order operations, resulting in a sequence length of 8. For binary operations, padding tokens are added to reach this uniform length. As in the other figures, the grid displays a separate heatmap for each attention head across the layers, with rows corresponding to layers and columns to attention heads. The x- and y-axes are labeled using the tokenized input (including padding where applicable), and each cell's color intensity (ranging from 0 to 1) reflects the attention weight, with numerical values overlaid to indicate the precise weights.

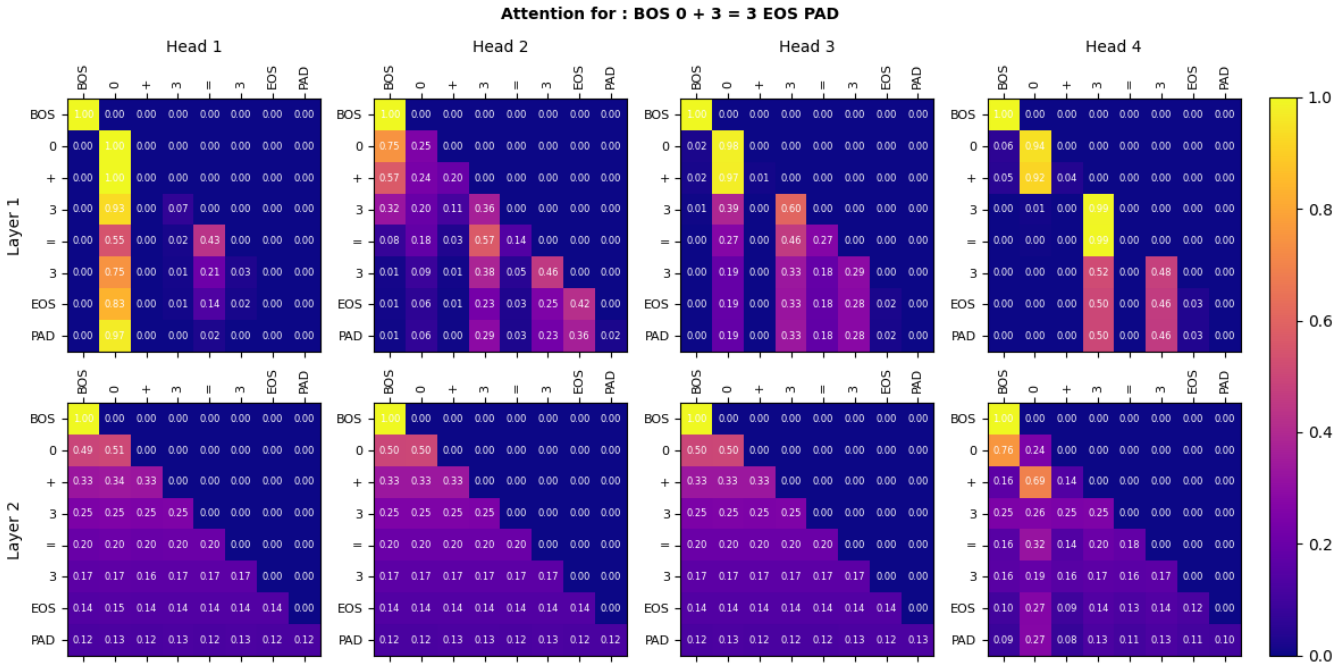

Figure 27: Attention heatmaps for a GPT model from experiment 4. This model is configured with 3 layers and both an embedding size and hidden size equal to 2^8. As with the other models, the grid displays a separate heatmap for each attention head across the layers, with rows representing layers and columns representing heads. The heatmaps illustrate the attention weights (ranging from 0 to 1) with numerical values overlaid on each cell.

Problem 4.8.b

After analyzing the attention maps across several examples, we can notice that tokens representing the numbers (especially the one immediately preceding the '=') consistently receive high attention. This is not uniform across all heads and layers; rather, it is more pronounced in the heads that seem to specialize in capturing local, arithmetic-relevant information.

The explanation for this behavior is straightforward: the operator (e.g., '+') in the dataset is constant, so it provides no new information for predicting the outcome. Instead, the model learns that the variable numerical tokens are the key elements for computation. By focusing on the numbers before the '=', the model can effectively extract the critical information needed to compute the result.

Thus, the attention mechanism adapts by prioritizing the numbers (tokens) that are essential for obtaining the correct answer.