Practical Report

Assignment 2, Programming Part

RNN, Transformer, Optimization, Regularization, Normalization

Daniel Lofeodo
Student ID: 20226991

Submitted on: **April 4, 2025**

MILA, Université de Montréal
IFT 6135 - W2025
Representation Learning
Prof: Aaron Courville

Problem 1

Implementing an LSTM (10pts)

1. How many learnable parameters `P` does your module have, as a function of the input dimension d_{x}, the hidden dimension d_{h}, and the number of layers L? You should distinguish the case where `bias` = `True` and `bias` = `False`.

   Given a number of layers L, an input size d_x, and a hidden size d_h, the number of parameters p_{cell} in a single LSTM cell is given by the number of weight parameters w_{cell} and bias parameters b_{cell}.

   For individual gates (input, forget, output, candidate cell state), the number of weight parameters is $(d_x + d_h) \cdot d_h$ and of bias parameters is d_h. Thus, for all 4 gates, the total number of parameters in a LSTM cell is:

$$\begin{aligned} w_{cell} &= 4 \cdot (d_x + d_h) \cdot d_h \\ b_{cell} &= 4 \cdot d_h \end{aligned}$$

   In a complete LSTM module, there are L LSTM cells. But, the input size varies according to the layer:

$$\begin{aligned} &\text{input size} = d_x \text{ if } L = 1 \\ &\text{input size} = d_h \text{ if } L > 1 \end{aligned}$$

   Therefore, the amount of parameters p_{LSTM} in an LSTM module's weight and bias parameters are the following:

$$\begin{aligned} w_{LSTM} &= 4 \cdot ((d_x + d_h) \cdot d_h + (L-1) \cdot (d_h + d_h) \cdot d_h) \\ &= 4 \cdot \left((d_x + d_h) \cdot d_h + 2 \cdot (L-1) \cdot d_h^2\right) \\ &= 4 \cdot (d_x + d_h) \cdot d_h + 8 \cdot (L-1) \cdot d_h^2 \\ b_{LSTM} &= L \cdot b_{cell} \\ &= 4 \cdot L \cdot d_h \end{aligned}$$

   If `bias`=`True`:

$$\begin{aligned} p_{LSTM} &= w_{LSTM} + b_{LSTM} \\ &= 4 \cdot (d_x + d_h) \cdot d_h + 8 \cdot (L-1) \cdot d_h^2 + 4 \cdot L \cdot d_h \\ &= 4(d_x + L) \cdot d_h + 4d_h^2 + 8 \cdot (L-1) d_h^2 \\ &= 4 \cdot (d_x + L + (2L-1) \cdot d_h) \cdot d_h \end{aligned}$$

   If `bias`=`False`:

$$\begin{aligned} p_{LSTM} &= w_{LSTM} \\ &= 4 \cdot (d_x + d_h) \cdot d_h + 8 \cdot (L-1) \cdot d_h^2 \\ &= 4 \cdot (d_x + (2L-1) \cdot d_h) \cdot d_h \end{aligned}$$

Problem 2

Implementing a GPT (Generative-Pretrained Transformer) (31pts)

1. How many learnable parameters does your `MultiHeadedAttention` module have, as a function of `num_heads` and `head_size`? You should distinguish the case where `bias` = `True` and `bias` = `False`.

   There are four linear modules of input and output size `d_model`. Therefore, there are `d_model`^{2} weight parameters p_w and `d_model` bias parameters p_b per linear module. But,

   $$\texttt{d\_model} = \texttt{num\_heads} \times \texttt{head\_size}$$

   So, p_w and p_b can be rewritten as a function of `num_heads` and `head_size`:

   $$p_w = 4 \times (\texttt{num\_heads} \times \texttt{head\_size})^2$$
   $$p_b = 4 \times (\texttt{num\_heads} \times \texttt{head\_size})$$

   The total amount of parameters p when `bias` = `True`:

   $$p = 4 \times \big((\texttt{num\_heads} \times \texttt{head\_size})^2 + \texttt{num\_heads} \times \texttt{head\_size}\big)$$

   When `bias` = `False`:

   $$p = 4 \times (\texttt{num\_heads} \times \texttt{head\_size})^2$$

2. Above, you compute the number of parameters for a single `MultiHeadedAttention` module as a function of `num_heads` and `head_size`. Each block in the decoder is made of one `MultiHeadedAttention` and a two layers bias free MLP with input and output dimension $d_{\text{model}} = \texttt{num\_heads} \times \texttt{head\_size}$, and with $d_{\text{ff}} = \texttt{multiplier} \times d_{\text{model}}$ hidden neurons. A decoder is made of `L` of such blocks. How many learnable parameters `P` does your decoder have, as a function of d_{model}, `num_heads`, `multiplier` and `L`? You should distinguish the case where `bias` = `True` and `bias` = `False`.

   A single decoder block contains the following modules and parameter counts:

   (a) **MultiHeadedAttention**
      i. $P_{MHA,b} = 4 \times \texttt{d\_model}^2$
      ii. $P_{MHA,b} = 4 \times \texttt{d\_model}$

   (b) **LayerNorm** (bias always true)
      i. $P_{LN,w} = \texttt{d\_model}$
      ii. $P_{LN,b} = \texttt{d\_model}$

   (c) **Linear** (bias always false)
      i. $P_{Linear,w} = \texttt{multiplier} \times \texttt{d\_model}^2$

   (d) **Linear** (bias always false)
      i. $P_{Linear,w} = \texttt{multiplier} \times \texttt{d\_model}^2$

   (e) **LayerNorm** (bias always true)
      i. $P_{LN,w} = \texttt{d\_model}$
      ii. $P_{LN,b} = \texttt{d\_model}$

   Summing these, we obtain the following parameter counts when `bias` = `True`:

$$\begin{aligned} P_{Block} &= 4 \times \texttt{d\_model}^2 + 2 \times \texttt{multiplier} \times \texttt{d\_model}^2 + 8 \times \texttt{d\_model} \\ &= (4 + 2 \times \texttt{multiplier}) \times (\texttt{num\_heads} \times \texttt{head\_size})^2 + 8 \times \texttt{num\_heads} \times \texttt{head\_size} \\ P_{Decoder} &= L \times P_{Block} \\ &= L \times \big((4 + 2 \times \texttt{multiplier}) \times (\texttt{num\_heads} \times \texttt{head\_size})^2 + 8 \times \texttt{num\_heads} \times \texttt{head\_size}\big) \end{aligned}$$

   When `bias` = `False`:

$$\begin{aligned} P_{Block} &= 4 \times \texttt{d\_model}^2 + 2 \times \texttt{multiplier} \times \texttt{d\_model}^2 + 4 \times \texttt{d\_model} \\ &= (4 + 2 \times \texttt{multiplier}) \times (\texttt{num\_heads} \times \texttt{head\_size})^2 + 4 \times \texttt{num\_heads} \times \texttt{head\_size} \\ P_{Decoder} &= L \times P_{Block} \\ &= L \times \big((4 + 2 \times \texttt{multiplier}) \times (\texttt{num\_heads} \times \texttt{head\_size})^2 + 4 \times \texttt{num\_heads} \times \texttt{head\_size}\big) \end{aligned}$$

Problem 4

Training language models and model comparison (44pts) **Sanity check (3.5pts)**: Use the default hyperparameters above to train an LSTM and a GPT.

1. (0.25 × 4 = 1pts) Report performance. You need to make four figures: one for loss ($\mathcal{L}^{(t)}_{\text{train}}$ and $\mathcal{L}^{(t)}_{\text{val}}$) and one for accuracy ($\mathcal{A}^{(t)}_{\text{train}}$ and $\mathcal{A}^{(t)}_{\text{val}}$), for both models. You can directly use the function `plot_loss_accs` from `plotter.py`, customize it as needed, or write your code from scratch.

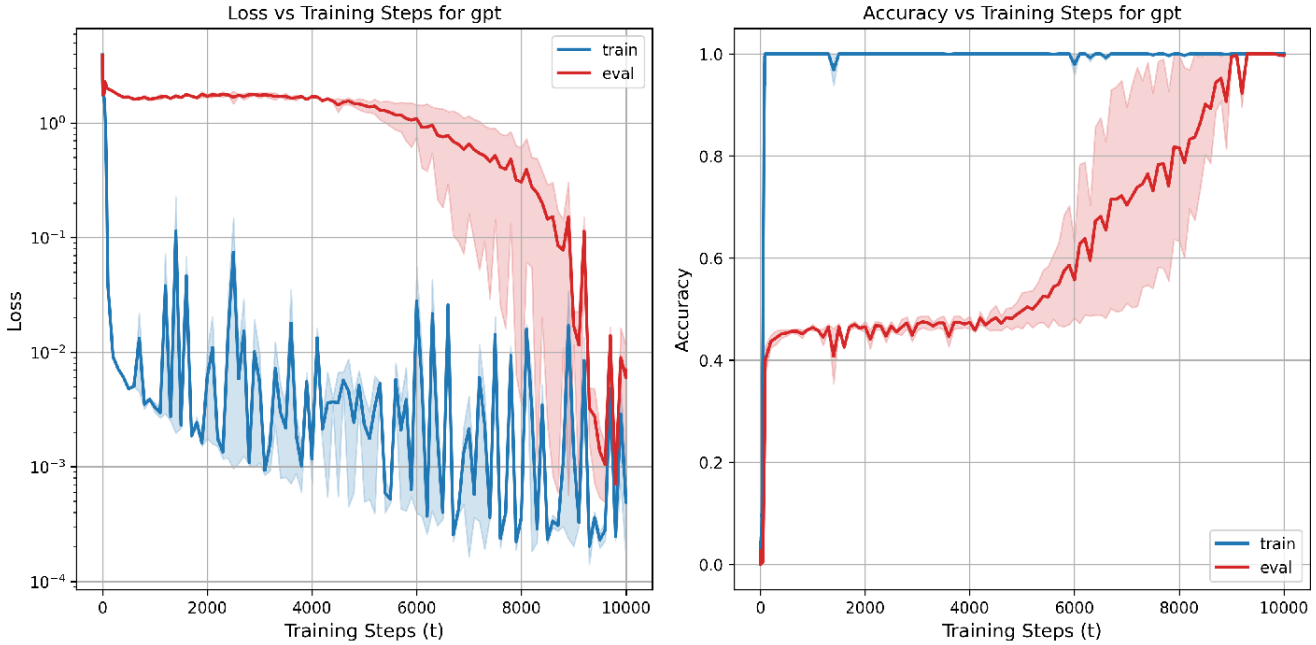

Figure 1: GPT loss and accuracy curves for training and evaluation

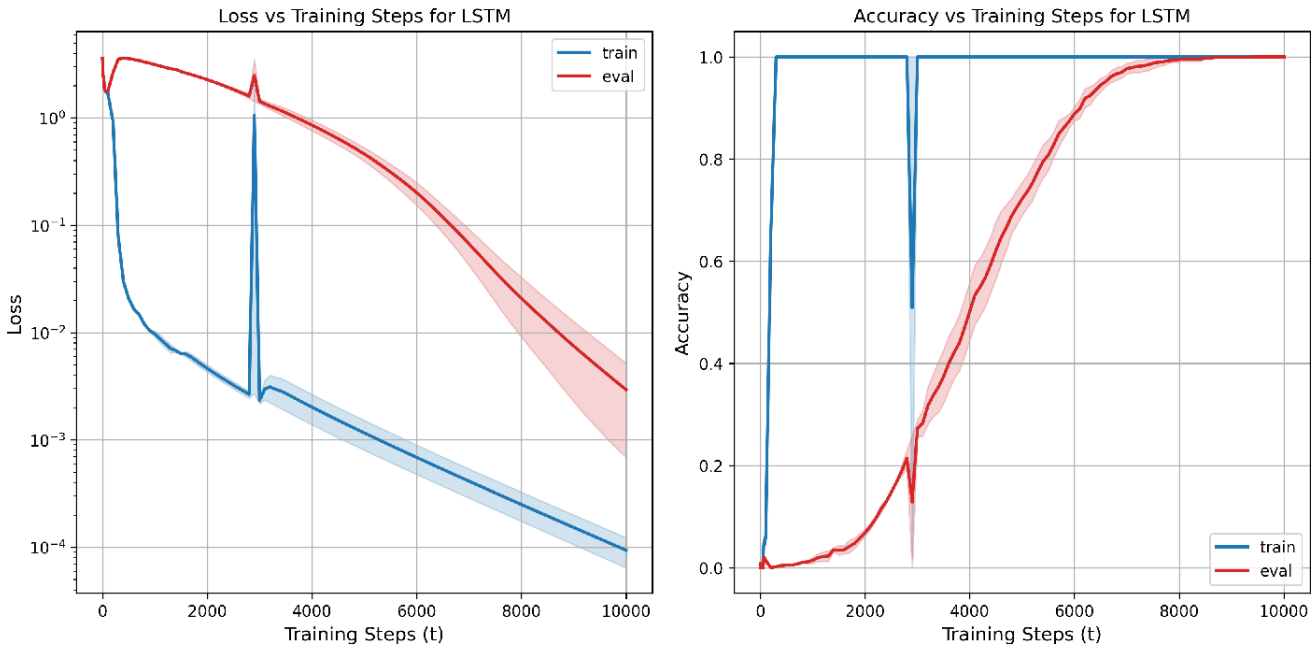

Figure 2: LSTM loss and accuracy curves for training and evaluation

2. (0.25 × 10 = 2.5pts) For each models, report $\mathcal{L}_{\text{train/val}}$, $\mathcal{A}_{\text{train/val}}$, $t_f(\mathcal{L}_{\text{train/val}})$, $t_f(\mathcal{A}_{\text{train/val}})$, $\Delta t(\mathcal{L})$ and $\Delta t(\mathcal{A})$. Make a table to compare results for LSTM and GPT (e.g., showing results side by side). Your result should be the mean and standard deviations, `mean` ± `std`.

Metric	**GPT**	**LSTM**
$\mathcal{L}_{\text{train}}$	$1.51 \times 10^{-4} \pm 1.01 \times 10^{-5}$	$9.36 \times 10^{-5} \pm 2.93 \times 10^{-5}$
$\mathcal{L}$val	$5.60 \times 10^{-4} \pm 9.11 \times 10^{-5}$	$2.93 \times 10^{-3} \pm 2.24 \times 10^{-3}$
$\mathcal{A}_{\text{train}}$	1.00 ± 0.00	1.00 ± 0.00
$\mathcal{A}_{\text{val}}$	1.00 ± 0.00	1.00 ± 0.00
$t_f(\mathcal{L}_{\text{train}})$	9650	10001
$t_f(\mathcal{L}_{\text{val}})$	9900.5	10001
$t_f(\mathcal{A}_{\text{train}})$	82.5	300
$t_f(\mathcal{A}_{\text{val}})$	8800	8700
$\Delta t(L)$	250.5	0
$\Delta t(A)$	8717.5	8400

Table 1: Comparison of GPT and LSTM models on different metrics.

Scaling data size (10pts): In this question, you will evaluate the evolution of the (comparative) performance of the models implemented above as a function of the amount of training data available.

3. (6pts) With the default parameters, train the models for $r_{\text{train}} \in \{0.1, \ldots, 0.9\}$.

   (a) ($0.25 \times 4 = 1$pts) Plot $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ as a function t (we advise you to consider a single curve for each of these metrics, and to use a color bar to distinguish r_{train}).

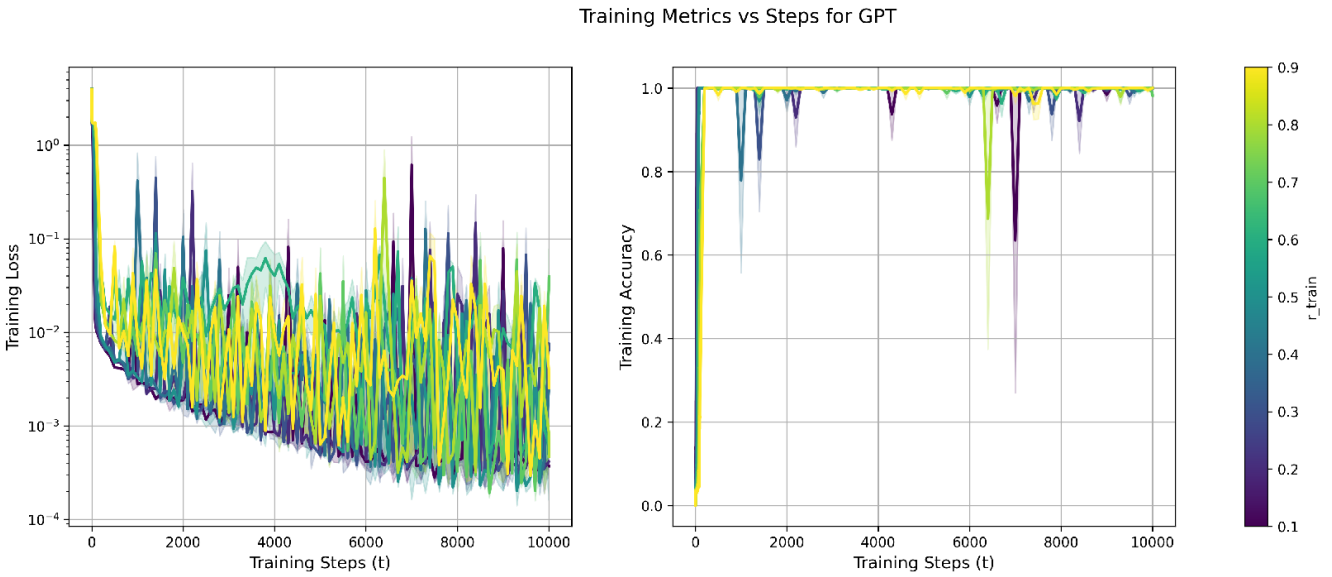

Figure 3: GPT training loss and accuracy for multiple `r_train` values.

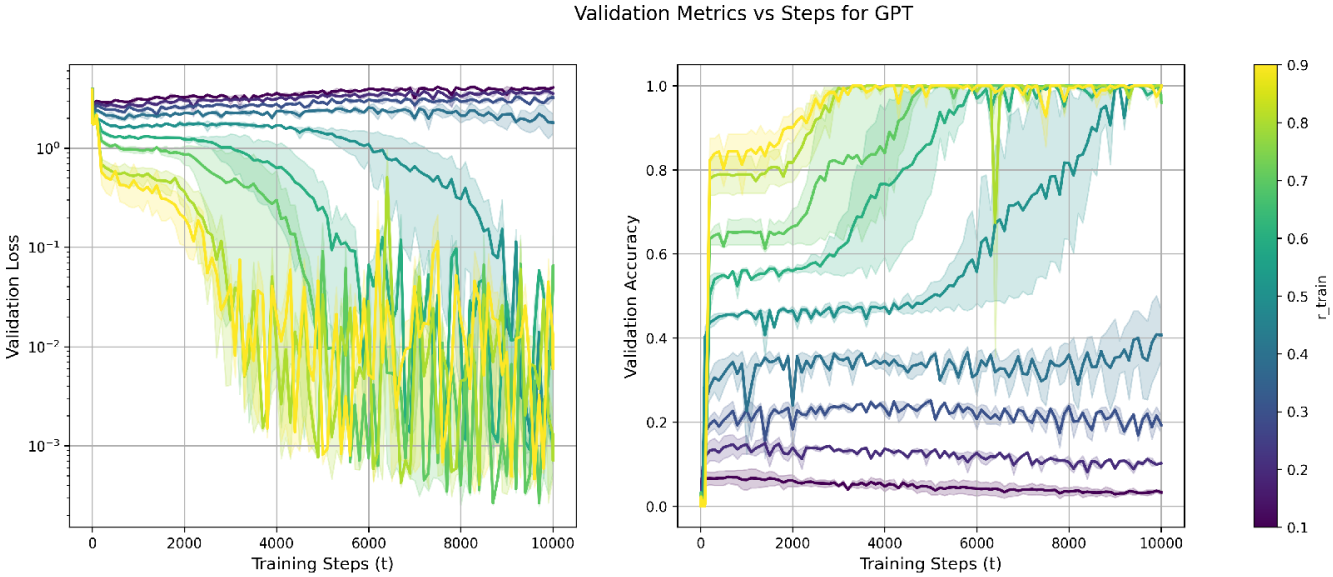

Figure 4: GPT validation loss and accuracy for multiple `r_train` values.

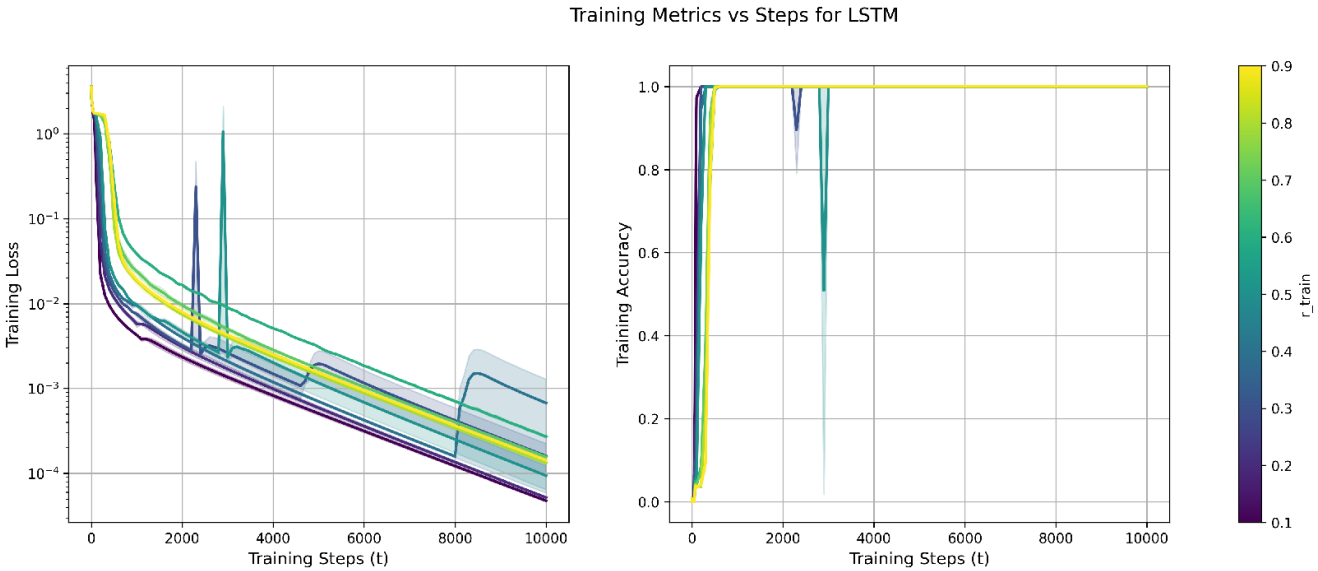

Figure 5: LSTM training loss and accuracy for multiple `r_train` values.

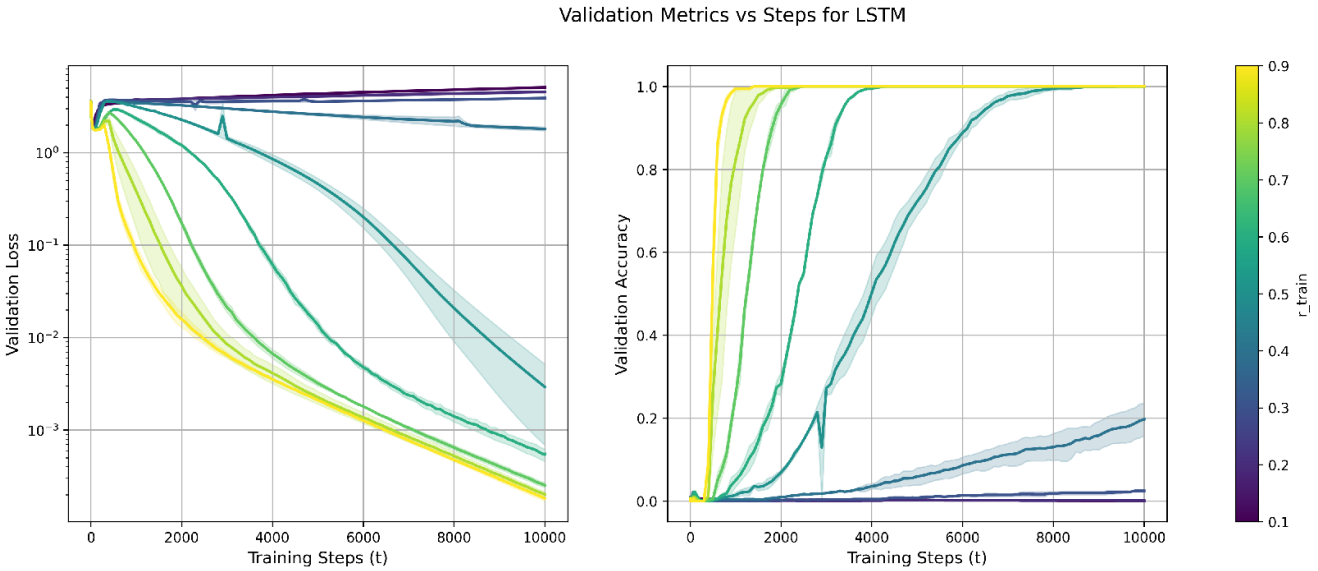

Figure 6: LSTM validation loss and accuracy for multiple `r_train` values.

(b) ($0.25 \times 8 = 2$pts) Plots the comparative performances ($\mathcal{L}_{\text{train/val}}$, $\mathcal{A}_{\text{train/val}}$, $t_f(\mathcal{L}_{\text{train/val}})$ and $t_f(\mathcal{A}_{\text{train/val}})$) of the models as a function of r_{train}. For $\mathcal{L}_{\text{train}}$ and $\mathcal{L}_{\text{val}}$, put the corresponding y-axis in log scale on your figures.

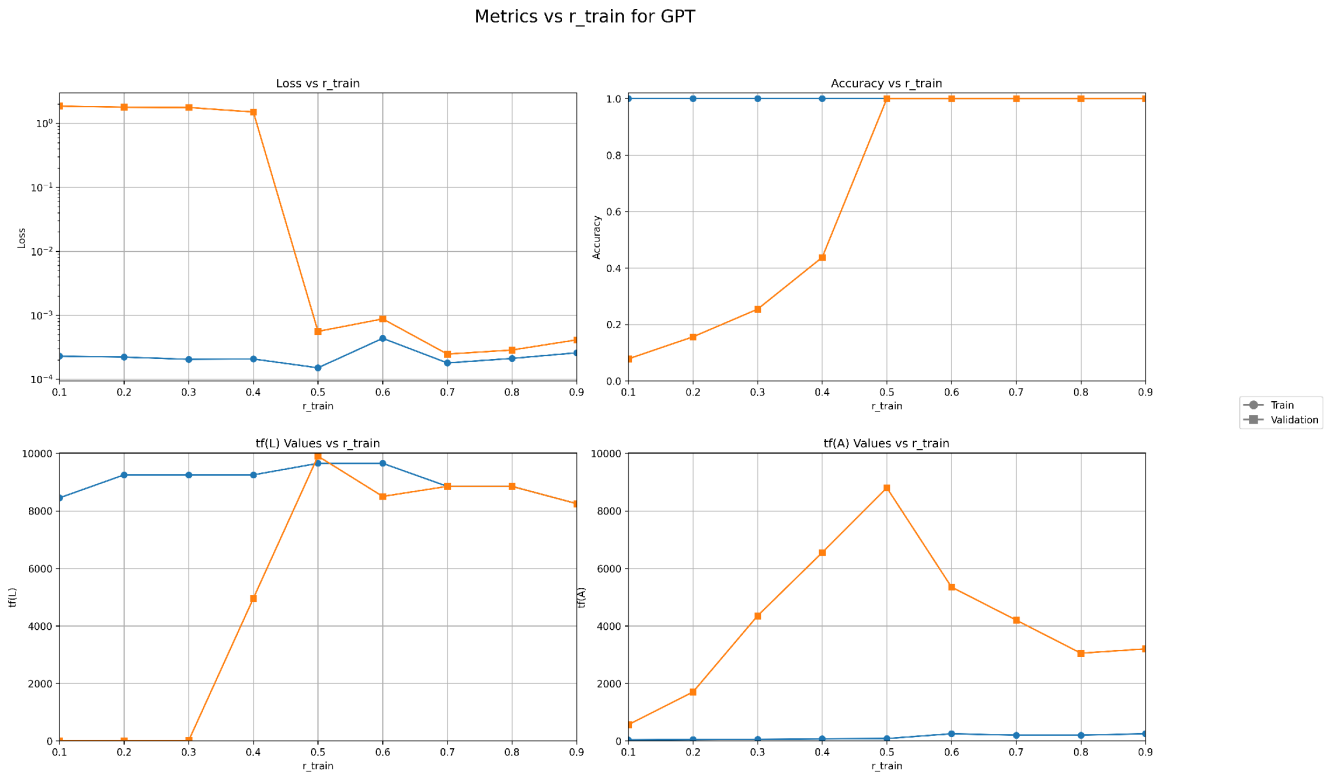

Figure 7: GPT training and validation loss and accuracy as a function of `r_train` values.

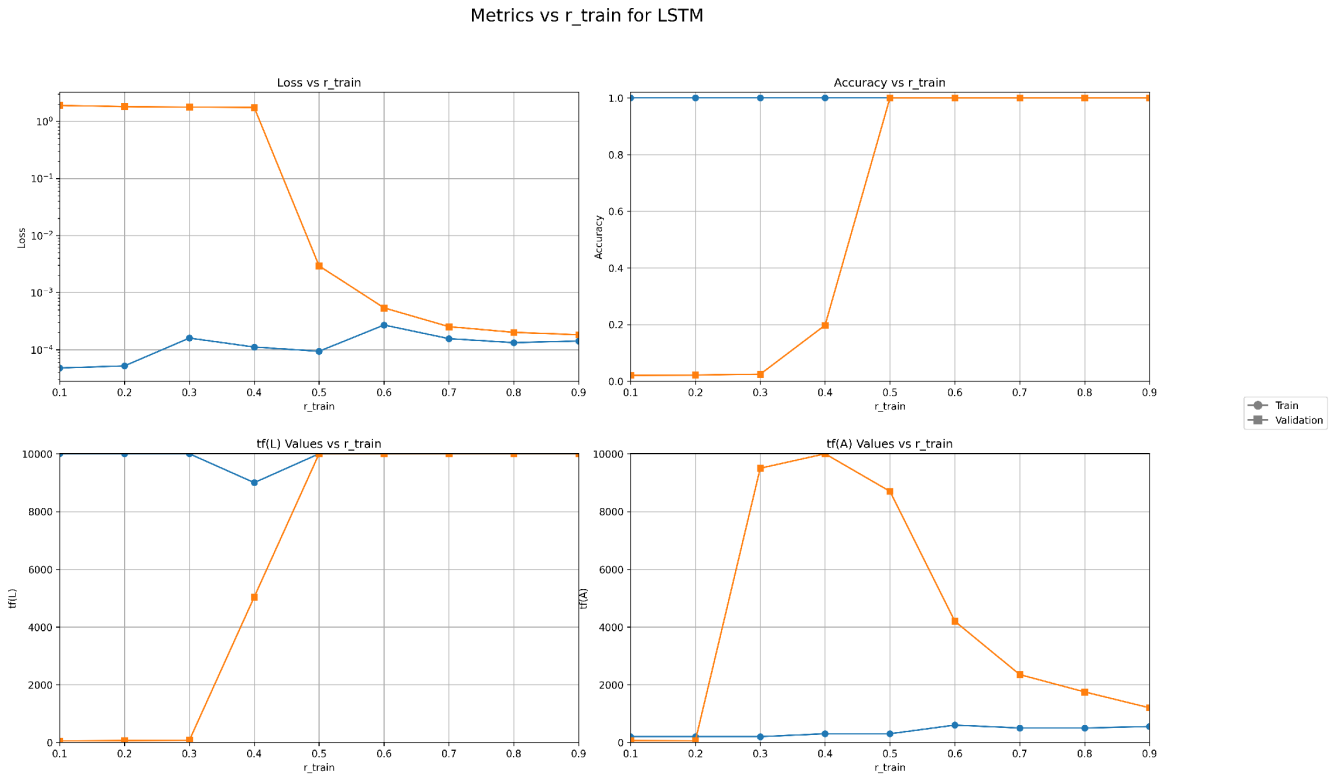

Figure 8: LSTM training and validation loss and accuracy as a function of `r_train` values.

(c) (1pt) Does one model scale better with $r_{\text{train}} \in (0, 1)$ than the other (you need to answer separately in terms of $\mathcal{L}_{\text{val}}$ and $\mathcal{A}_{\text{val}}$).

i. $\mathcal{A}_{\text{val}}$
Both GPT and LSTM models respond favorably to increasing the ratio of available training data. Both reach a peak validation accuracy of 100% when the ratio reaches 0.5.
The GPT model responds quicker to increases in `r_train`, with steady increases in performance as the ratio increases from 0.1 to 0.5.
In contrast, the LSTM model does not show much of an improvement in performance in accuracy with initial increments of `r_train`, with validation accuracy stagnating under 20% until `r_train` reaches 0.5 and accuracy shoots up to 100%.

ii. $\mathcal{L}_{\text{val}}$
In constrast to validation accuracy, the GPT model's and LSTM model's validation loss behaviour as a function of `r_train` is quite similar. The loss stagnates around 1 until `r_train` reaches a value of 0.5, at which point the loss shoots down very rapidly.
The GPT model's validation loss reaches a minimum value when `r_train` $= 0.7$ and begins steadily augmenting afterwards. Further, the validation loss curves for the GPT model become increasingly noisy and unstable as the training ratio increases.
The LSTM model's validation loss continues descending for all values of `r_train`. In addition, the LSTM's validation loss curves maintain a smooth and stable descent as

training ratio increases. This asymmetry in behaviour between the GPT and LSTM indicates that the LSTM scales better to higher values of `r_train`.

(d) (2pts) What's the smallest r_{train} value that allows generalization ($\mathcal{A}_{\text{val}} \geq 0.9$), and the largest value for which the model just overfits the training data ($\mathcal{A}_{\text{train}} \approx 1.0$ and $\mathcal{A}_{\text{val}} \leq 0.5$)? Is there a big difference between these values? If there's not much difference between these two values, what do you think can be the reason (2 sentences maximum)?

The smallest value of `r_train` that allows for generelizatkon is 0.6. In both the GPT and the LSTM, no overfitting occurs for any value of `r_train`. In fact, overfitting decreases as `r_train` increases ($\Delta t_f(\mathcal{A}/\mathcal{L})$ decreases) which makes sense because increasing the quantity of training data will improve a model's ability to generalize, not hinder it.

4. (4pts) Train the two models (LSTM and GPT) with $p = 11$ and `operation_orders` $= [2, 3]$ (and $r_{\text{train}} = 0.5$, its default value), and report performance. You need to make sure that half of the ternary operations (a+b+c=r) go into the training dataset and the other half into the validation dataset. The same applies to operations of binary operation (a+b=r). Binary sample have `eq_positions` $= 3$, and ternary sample have `eq_positions` $= 5$. You can, therefore, first take the data with `r_train=1.0` (i.e., just get `train_dataset`), then split it into two according to the order (2 or 3), take half from each half, and combine.

   (a) ($0.25 \times 4 = 1$pts) You need to plot $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ as a function t.

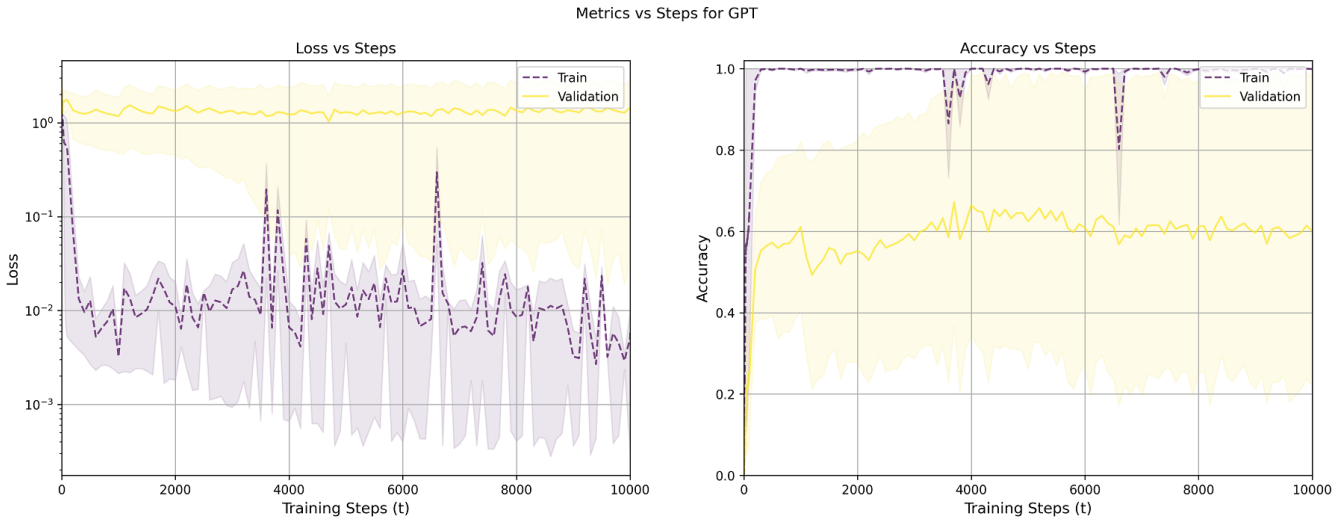

Figure 9: GPT training and validation loss and accuracy for combined operation orders.

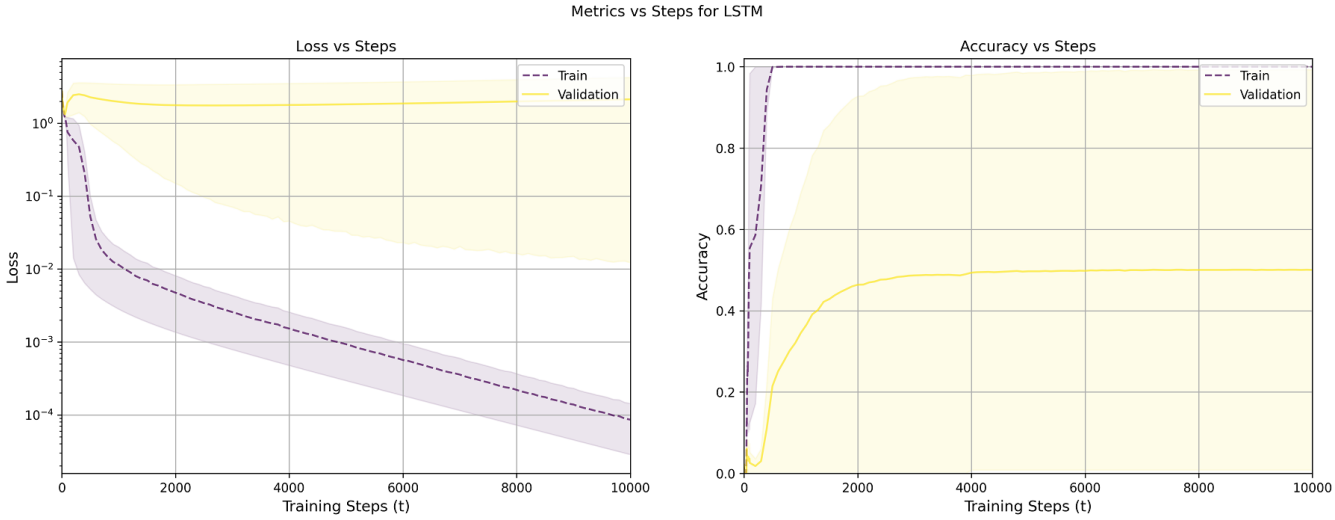

Figure 10: LSTM training and validation loss and accuracy for combined operation orders.

(b) (0.25 × 4 × 2 = 2pts) Compute $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ separately on binary and ternary operation as a function of t, and plot it.

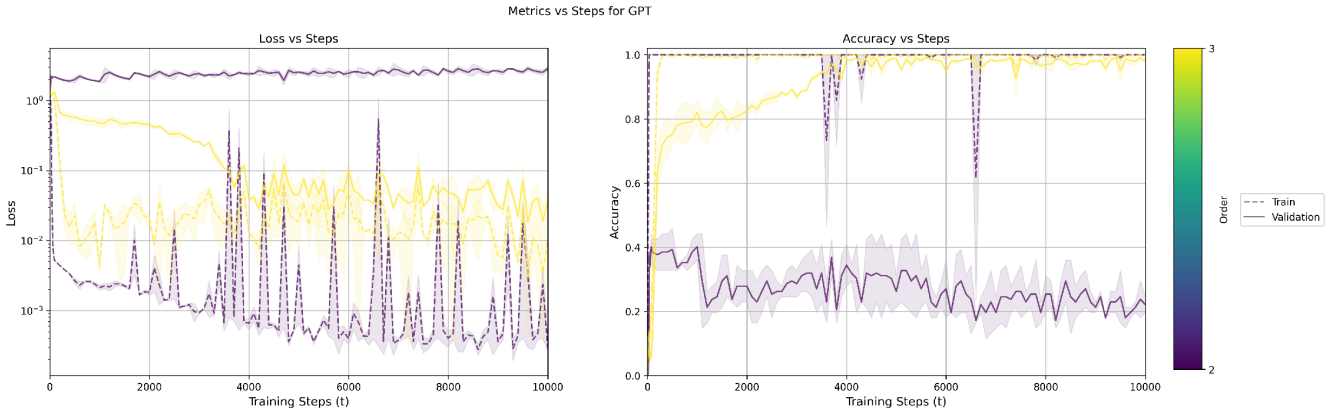

Figure 11: GPT training and validation loss and accuracy for multiple `operation_order` values.

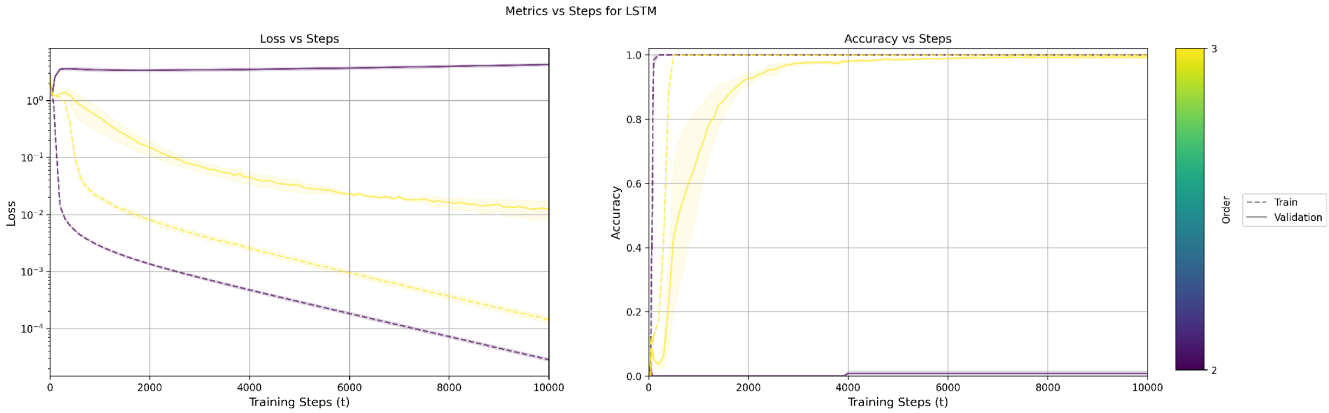

Figure 12: LSTM training and validation loss and accuracy for multiple `operation_order` values.

(c) (1pt) During training, are certain types of operation predicted more quickly than others?

We can see from the figures separating the model metrics by operation orders that the models learn much quicker and more effectively for an operation order value of 3. This is counter intuitive due to the more complex nature of using three operands over 2, but it is explainable by the fact that there are more data to train on for operation order 3 than for 2.

In this question, we set the p value to 11. This means that the maximum number that an operand can take is 11. When operation order is 2, there are then 11^2 data that are generated, whereas when operation order is 3 there are 11^3 data generated. Hence, the model is trained on significantly more data when operation order = 3, leading to quicker generalization.

Scaling model size (10pts): In this question, you will evaluate the evolution of the (comparative) performance of the models as a function of their size. Both LSTM and GTP have as hyperameters the number of layers (L) and the embedding dimension (d), defaulted above to $(L, d) = (2, 2^7)$.

5. (10pts) Train each model for $(L, d) \in \{1, 2, 3\} \times \{2^6, 2^7, 2^8\}$ (3×3 models). For LSTM, you must fix the hidden size to d each time.

   (a) ($0.25 \times 4 \times 3 = 3$pts) Plot $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ as a function t. For a given L, we advise you to consider a single curve for each of these metrics and to use a color bar to distinguish d (so make a different figure for each L, showing the four metrics). Also, always put the axis/colorbar corresponding to d in $\log_2$ scale.

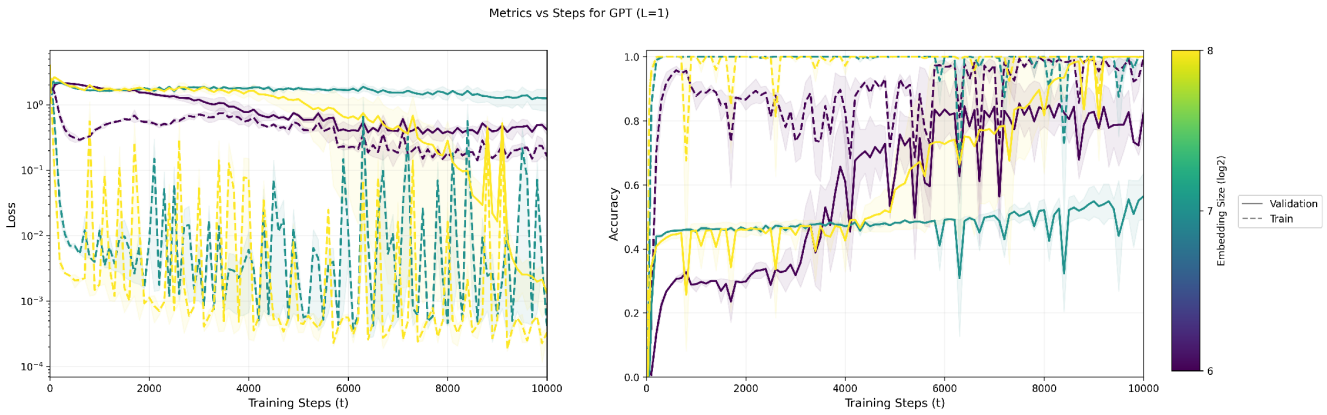

Figure 13: GPT training and validation loss and accuracy for multiple embedding sizes when $L = 1$.

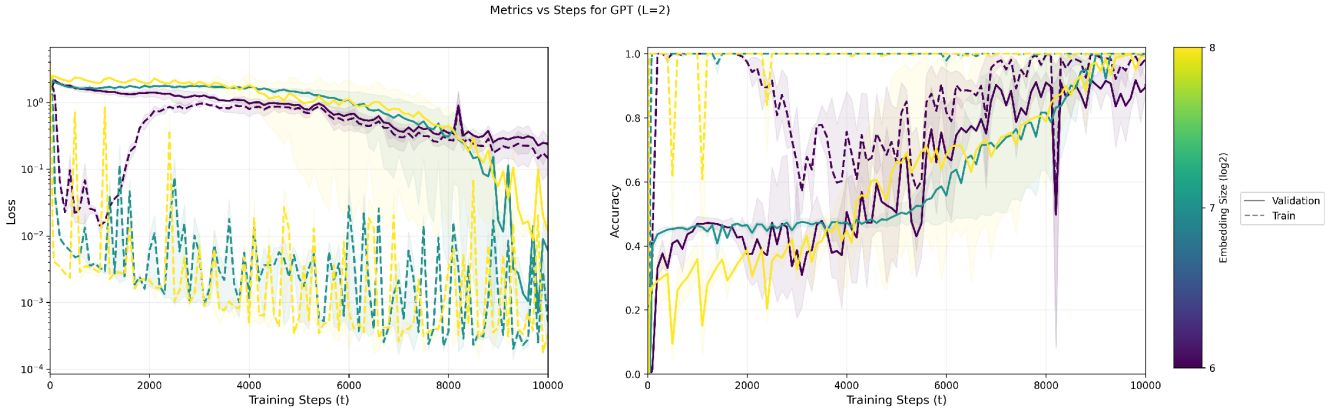

Figure 14: GPT training and validation loss and accuracy for multiple embedding sizes when $L = 2$.

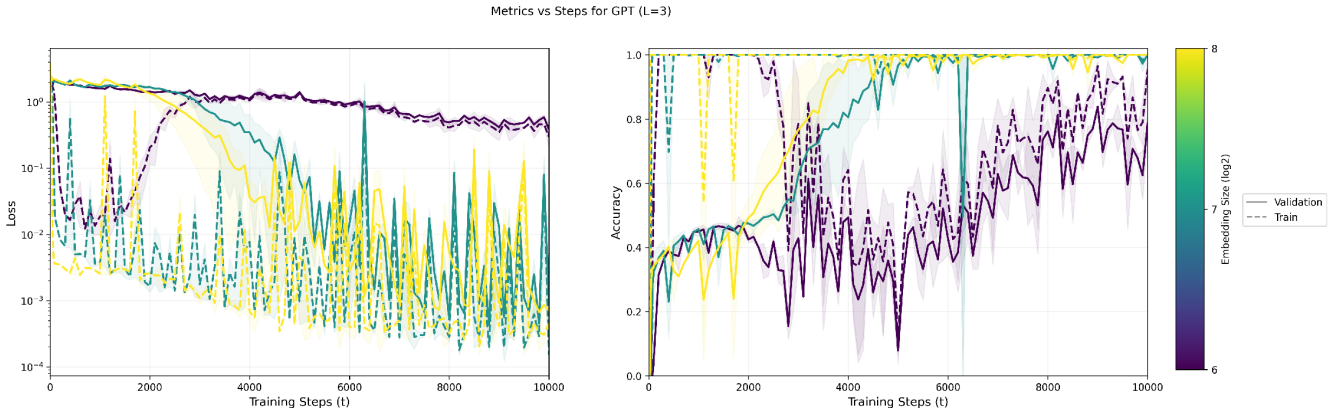

Figure 15: GPT training and validation loss and accuracy for multiple embedding sizes when $L = 3$.

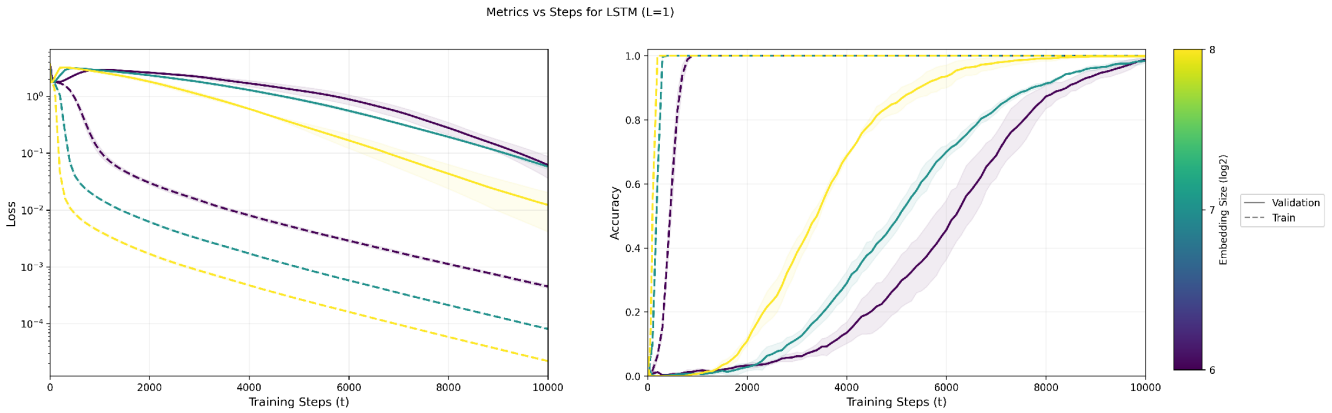

Figure 16: LSTM training and validation loss and accuracy for multiple embedding sizes when $L = 1$.

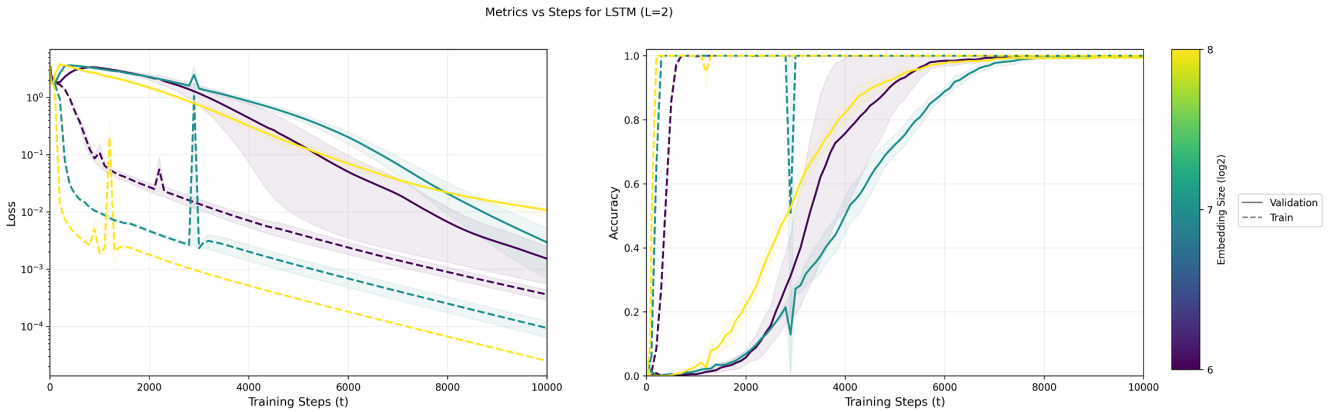

Figure 17: LSTM training and validation loss and accuracy for multiple embedding sizes when $L = 2$.

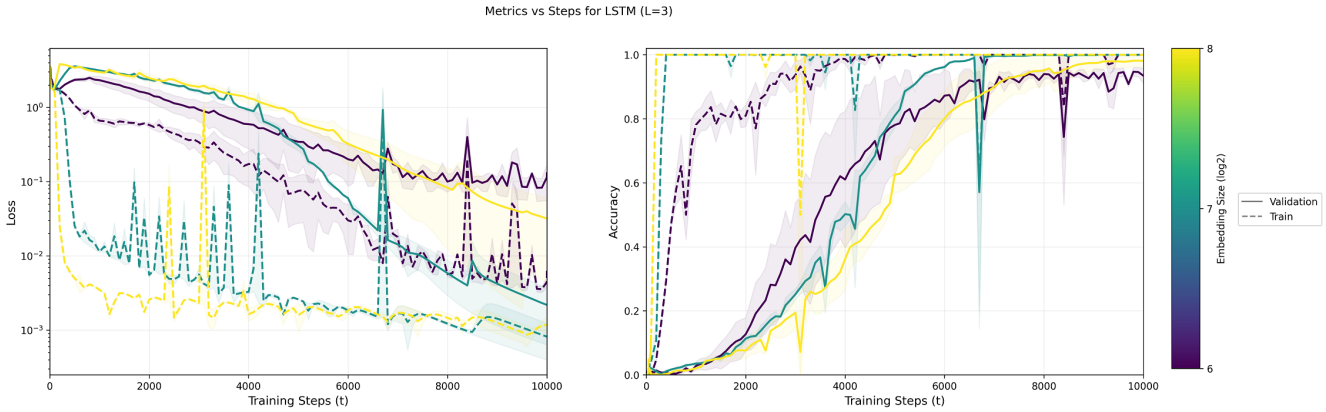

Figure 18: LSTM training and validation loss and accuracy for multiple embedding sizes when $L = 3$.

(b) ($0.25 \times 8 \times 2 = 4$pts) Plot $\mathcal{L}_{\text{train/val}}$, $\mathcal{A}_{\text{train/val}}$, $t_f(\mathcal{L}_{\text{train/val}})$ and $t_f(\mathcal{A}_{\text{train/val}})$ as a function of d, L, and the number of parameters. When counting the number of model parameters, you should exclude all vocabulary and positional embeddings (for the loss, put the corresponding y-axis in log scale on your figures). You need to combine L and d on the same figure (either for each metric, put the metric on the y-axis, then have d on the x-axis and use a color bar for different Ls, or have L on the x-axis and use different color bars for d). The figure for metrics as a function of the number of parameters should be separate.

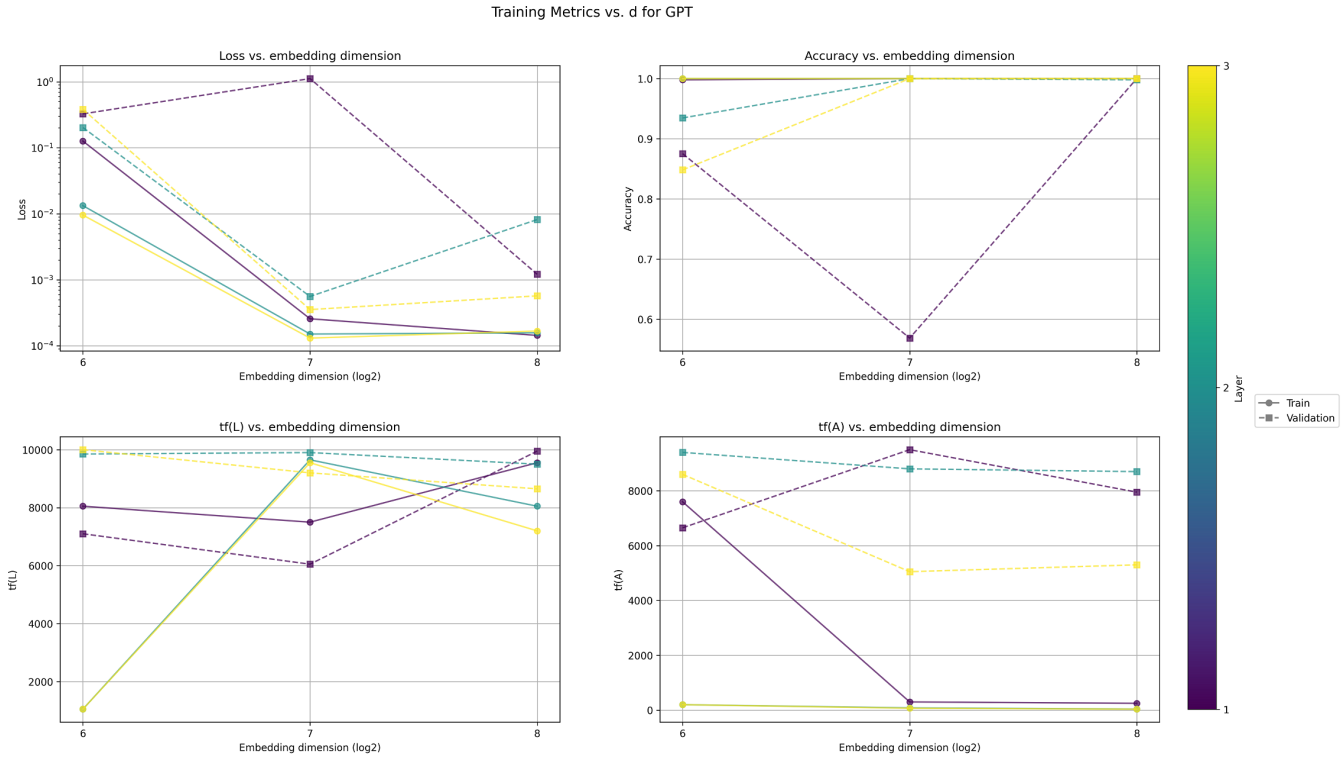

Figure 19: GPT training and validation loss, accuracy, $t_f(\mathcal{L})$ and $t_f(\mathcal{A})$ for multiple L values as a function of embedding size.

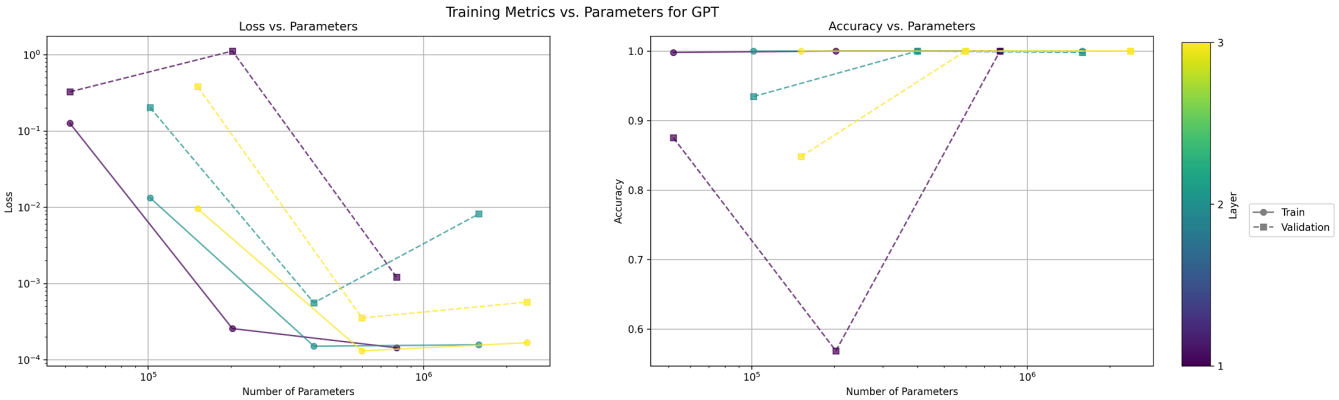

Figure 20: GPT training and validation loss and accuracy for multiple L values as a function of model parameter count.

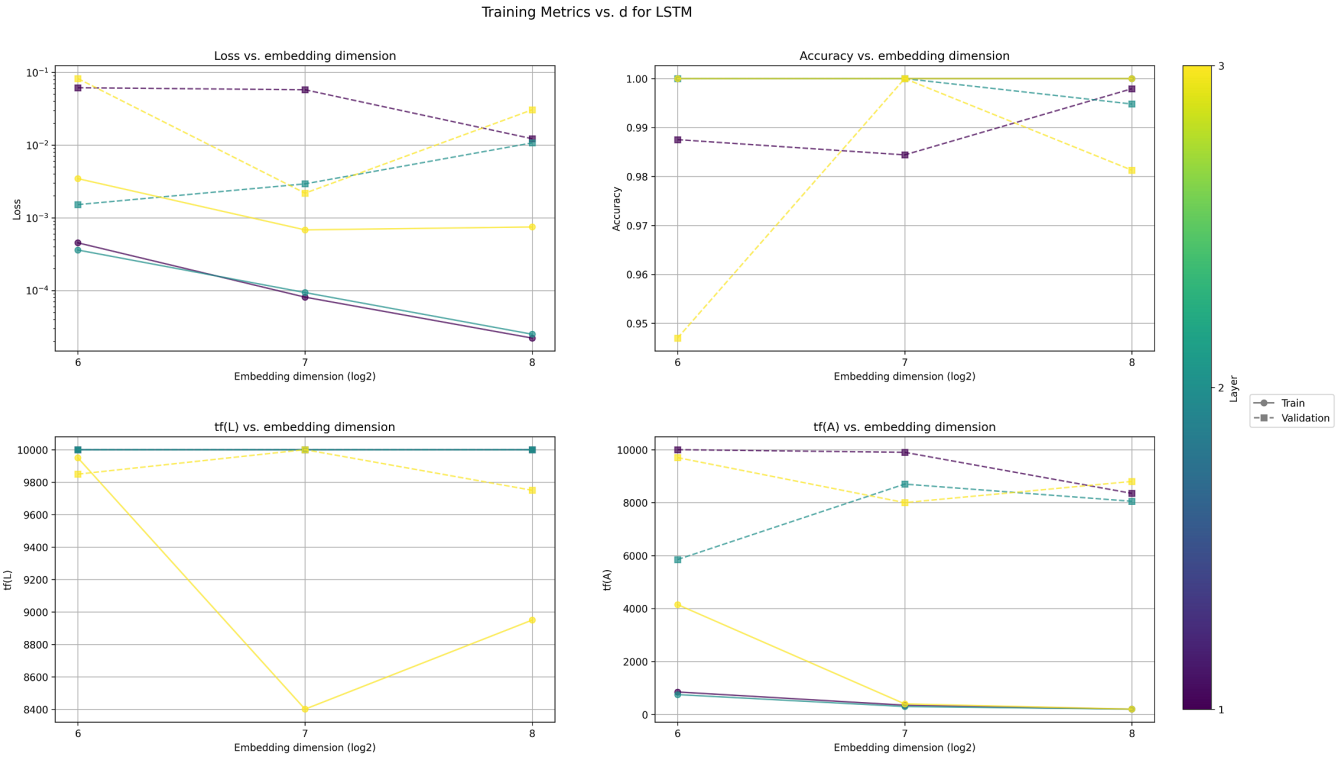

Figure 21: LSTM training and validation loss, accuracy, $t_f(\mathcal{L})$ and $t_f(\mathcal{A})$ for multiple L values as a function of embedding size.

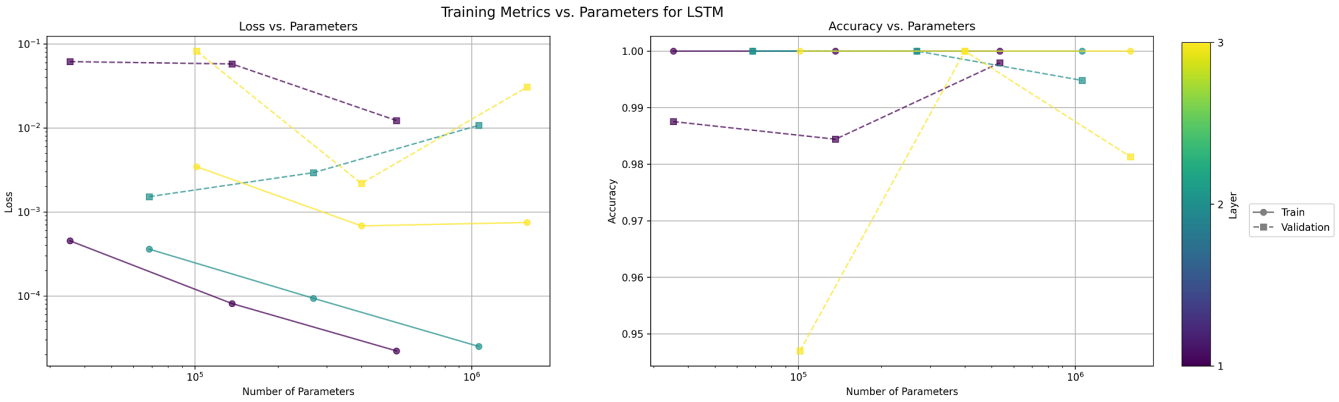

Figure 22: LSTM training and validation loss and accuracy for multiple L values as a function of model parameter count.

(c) (3pts) For L fixed, which model scales better with d? For d fixed, which model scales better with L?Which model scales better with the number of parameters (LSTM or GPT)? You need to answer separately in terms of $\mathcal{L}_{\text{val}}$ and $\mathcal{A}_{\text{val}}$.

i. Fixed L:

A. $\mathcal{L}_{\text{val}}$ The GPT shows a clear progression in loss minimization as d increases. With the LSTM, the validation loss is higher for $d = 2^8$ than it is for $d = 2^7$ for all layer sizes.
Overall, the **GPT** scales better with d for a fixed L than the LSTM in validation loss.

B. $\mathcal{A}_{\text{val}}$ Similarly, the validation accuracy displays a better performance increase in the GPT for an increase d when L is constant.
In the LSTM, the accuracy generally does scale with d, except in the case of $L = 3$, where $d = 2^7$ outperforms $d = 2^8$.
In general, it can be said that the validation accuracy scales better as a function of d for a fixed L for the **GPT** than for the LSTM.

ii. Fixed d:

A. $\mathcal{L}_{\text{val}}$ For the GPT with $d > 2^6$, the loss scales well with an increase in layer size. The loss descends faster and lower for higher values of L.
For the LSTM, the loss only scales well with L for $d = 2^7$. For other d values, the loss either does not improve or worsens.
Overall, the **GPT**'s validation loss scales better with L.

B. $\mathcal{A}_{\text{val}}$ The GPT's validation accuracy scales very well with an increased layer size L for a constant d.
The LSTM's validation accuracy does not scale much with an increase in L for a fixed d. The performance is relatively constant across different L values.
Overall, the **GPT**'s validation accuracy scales much better with an increase in layer size than does the LSTM.

iii. Number of parameters:

A. $\mathcal{L}_{\text{val}}$ The **GPT**'s validation loss scales better with number of parameters than does the LSTM.

The LSTM actually shows somewhat of an inverse relationship between loss minimization and number of parameters.

B. $\mathcal{A}_{\text{val}}$ In terms of validation accuracy, the **GPT** displays better scaling with number of parameters. The number of parameters leads to a steady increase in performance in GPT, whereas in the LSTM, an increase in parameters tends to lead to a decrease in accuracy.

Scaling compute (5pts): We estimate the total non-embedding training compute by $\texttt{C} \simeq \texttt{P} \times \texttt{B} \times \texttt{T}$, where $\texttt{B}$ is the batch size and $\texttt{T}$ is the number of training steps. In practice, for a given budget $\texttt{C}$, the aim is to find the triplet $(\texttt{P}, \texttt{B}, \texttt{T})$ that gives the best generalization performance. For example, you may see the session time on Google Colab with computational resources as a budget $\texttt{C}$, and your goal is to find the best model given this budget. If you have a law that for any triplet $(\texttt{P}, \texttt{B}, \texttt{T})$ gives the generalization performance, you can, for a given $\texttt{C}$, draw the level curve $\texttt{P} \times \texttt{B} \times \texttt{T} = \texttt{C}$, and find the point $(\texttt{P}, \texttt{B}, \texttt{T})$ on that curve that gives you the best performance. To avoid doing a lot of experiments, you'll focus on $(\texttt{B}, \texttt{T})$ and do something much more straightforward, keeping $\texttt{P}$ fixed (use default parameters $(L, d) = (2, 2^7)$).

6. (5pts) Train the models for $\texttt{B} \in \{2^5, 2^6, 2^7, 2^8, 2^9\}$ and $\texttt{T} = 2 \times 10^4 + 1$. Now, for each $\texttt{B}$ and for each $\texttt{T}' \in \{\alpha\texttt{T}, \alpha \in \{0.1, 0.2, \ldots, 0.9, 1.0\}\}$ compute the performances ($\mathcal{L}_{\text{train/val}}$ and $\mathcal{A}_{\text{train/val}}$). The point here is to consider that you only trained the model for $\texttt{T}' = \alpha\texttt{T}$ steps (budget $\texttt{C}' \simeq \alpha\texttt{C}$), and you need to find the smallest loss and the largest accuracy under these $\texttt{T}'$ training steps (the functions `get_extrema_performance_steps` and `get_extrema_performance_steps_per_trials` in `checkpointing.py` accept a parameter `T_max`, and the results return take into account just the training statistics before the training step `T_max`).

   (a) ($0.25 \times 4 = 1$pts) Plot $\mathcal{L}^{(t)}_{\text{train/val}}$ and $\mathcal{A}^{(t)}_{\text{train/val}}$ as a function $t \in [\texttt{T}]$ (we advise you to consider a single curve for each of these metrics, and to use a color bar to distinguish $\texttt{B}$). Also, always put the axis/colorbar corresponding to $\texttt{B}$ in $\log_2$ scale.

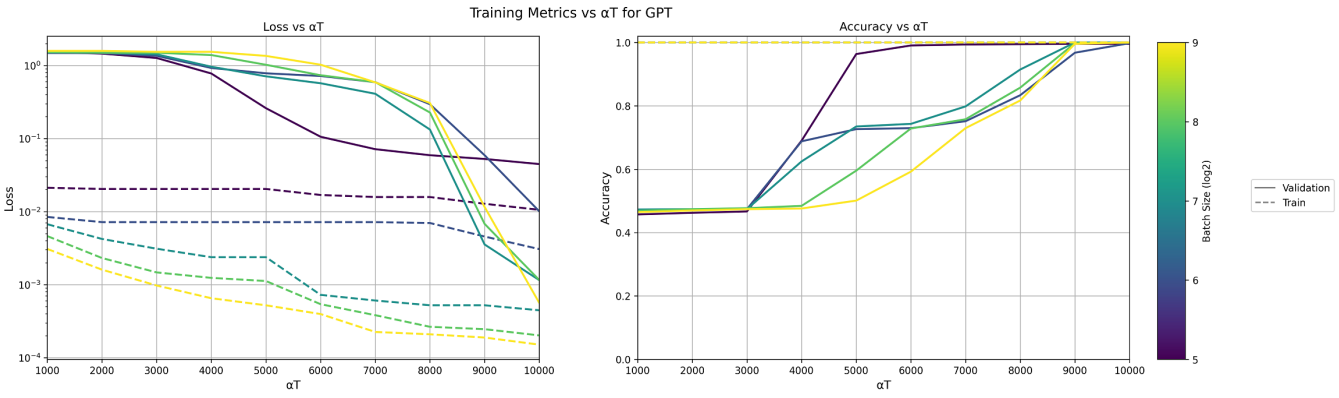

Figure 23: GPT training and validation loss and accuracy for multiple batch sizes as a function of αT.

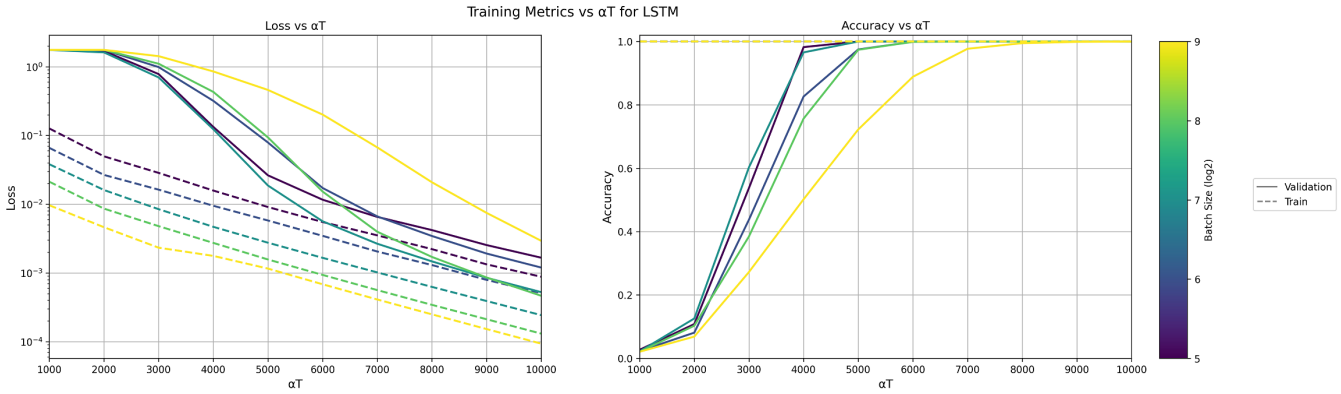

Figure 24: LSTM training and validation loss and accuracy for multiple batch sizes as a function of αT.

(b) ($0.25 \times 8 = 2$pts) For each metric ($\mathcal{L}_{\text{train/val}}$, $\mathcal{A}_{\text{train/val}}$, and the corresponding t_f), make a curve with the `B` values as the x-axis, metric values as the y-axis, and different colors for each $\alpha \in \{0.1, 0.2, \ldots, 0.9, 1.0\}$ (don't forget to put a color bar next the curves for α). You can also consider LSTM and GPT on the same curve for comparison purposes (or side-by-side curves).

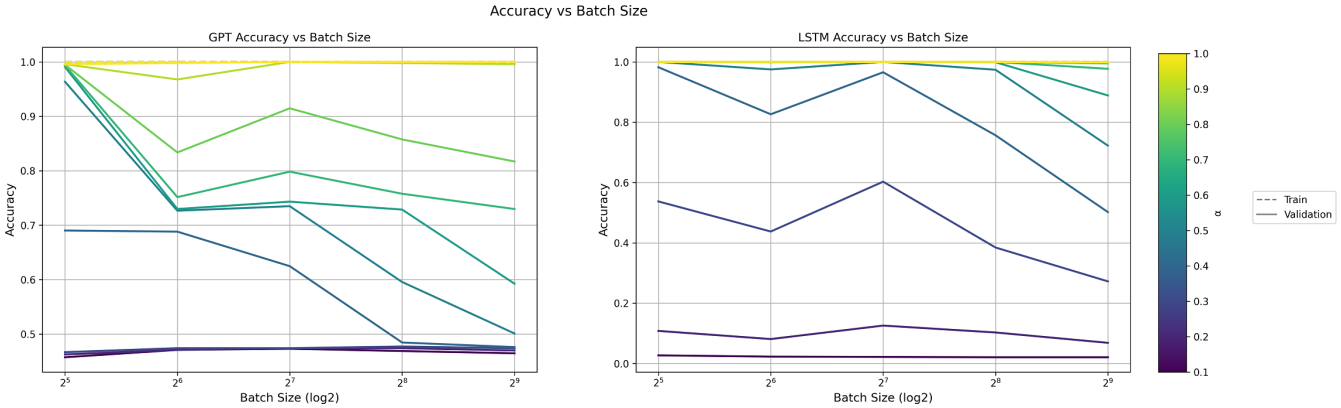

Figure 25: GPT and LSTM training and validation accuracy for α values as a function of batch size.

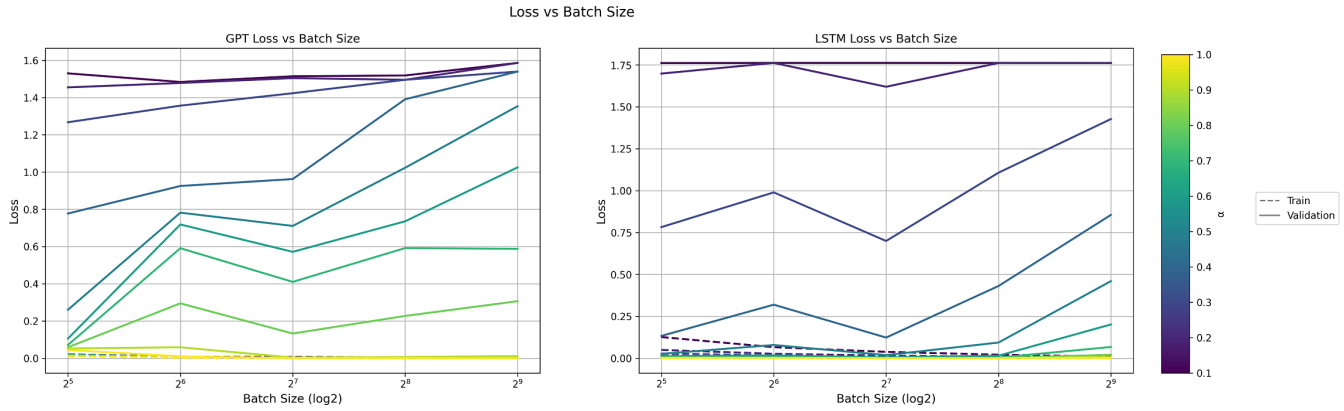

Figure 26: GPT and LSTM training and validation loss for α values as a function of batch size.

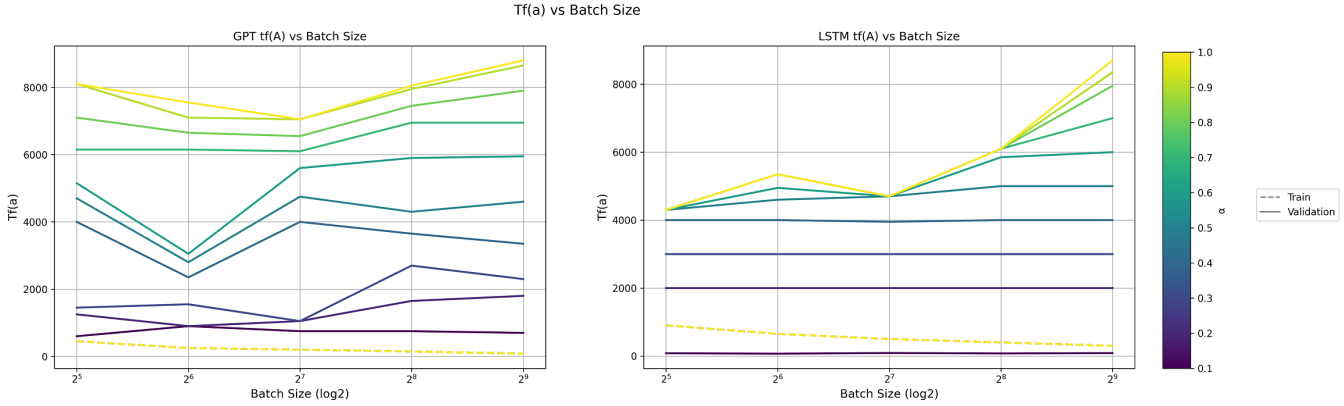

Figure 27: GPT and LSTM training and validation $t_f\mathcal{A}$ for α values as a function of batch size.

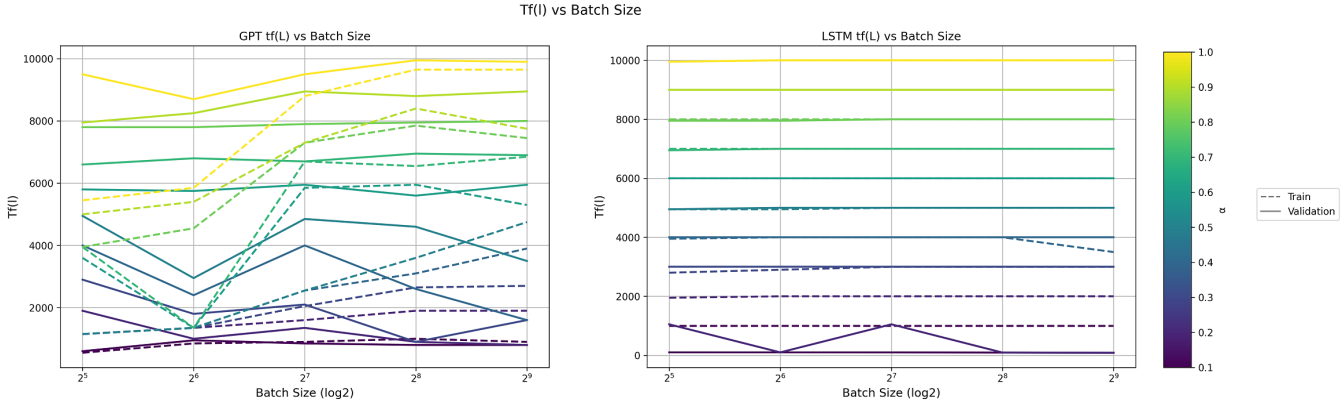

Figure 28: GPT and LSTM training and validation $t_f\mathcal{L}$ for α values as a function of batch size.

(c) (2pts) Do the generalization (validation) performances improve as `B` and/or α increases? You need to answer separately in terms of $\mathcal{L}_{\text{val}}$ and $\mathcal{A}_{\text{val}}$.

i. $\mathcal{L}_{\text{val}}$

A. αT

The validation loss consistently diminishes as αT increases for both the GPT and LSTM models.

B. B

The validation loss peaks at a minimum value for the LSTM generally at a batch size of 2^7. Afterwards, the loss continues to increase.

For the GPT, the smallest batch size yields the lowest loss, with loss increasing alongside batch size.

ii. $\mathcal{A}_{\text{val}}$

A. αT

The validation accuracy consistently increases as a function of αT for both GPT and LSTM models.

B. B

The LSTM's validation accuracy tends to peak at a batch size of 2^5 or 2^7. But generally, with an increase in batch size the accuracy diminishes.

The GPT's accuracy also decreases with batch size, with the peak value occuring at the smallest batch size of 2^5.

Regularization (6.5pts):

7. (6.5pts) With $\texttt{T} = 4 \times 10^4 + 1$, train an LSTM with `weight_decay` in $\{1/4, 1/2, 3/4, 1\}$, and also track the ℓ_2 norm $\|\theta^{(t)}\|_2$ of model parameters $\theta^{(t)}$ as a function of training steps t (you can modify the function `eval_model` in `trainer.py` for that).

   (a) ($0.25 \times 6 = 1.5$pts) Plot $\mathcal{L}^{(t)}_{\text{train/val}}$, $\mathcal{A}^{(t)}_{\text{train/val}}$ and $\|\theta^{(t)}\|_2$ as a function t (we advise you to consider a single curve for each of these metrics, and to use a color bar to distinguish `weight_decay`).

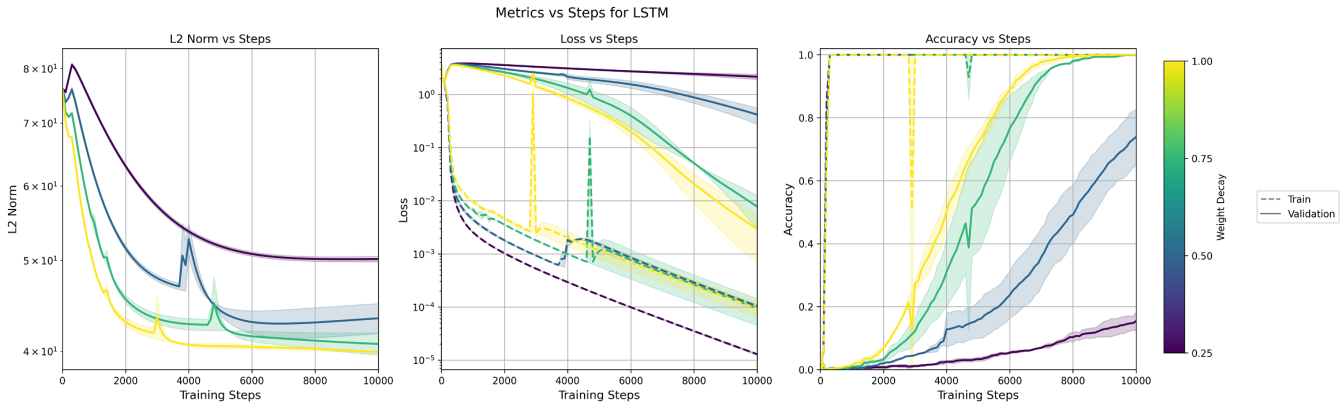

Figure 29: LSTM training and validation loss and accuracy and L2 parameter norm for multiple weight decay values.

   (b) ($0.25 \times 8 = 2$pts) Plot $\mathcal{L}_{\text{train/val}}$ and $\mathcal{A}_{\text{train/val}}$ (and the corresponding t_f) as a function of `weight_decay` (log scale for loss).

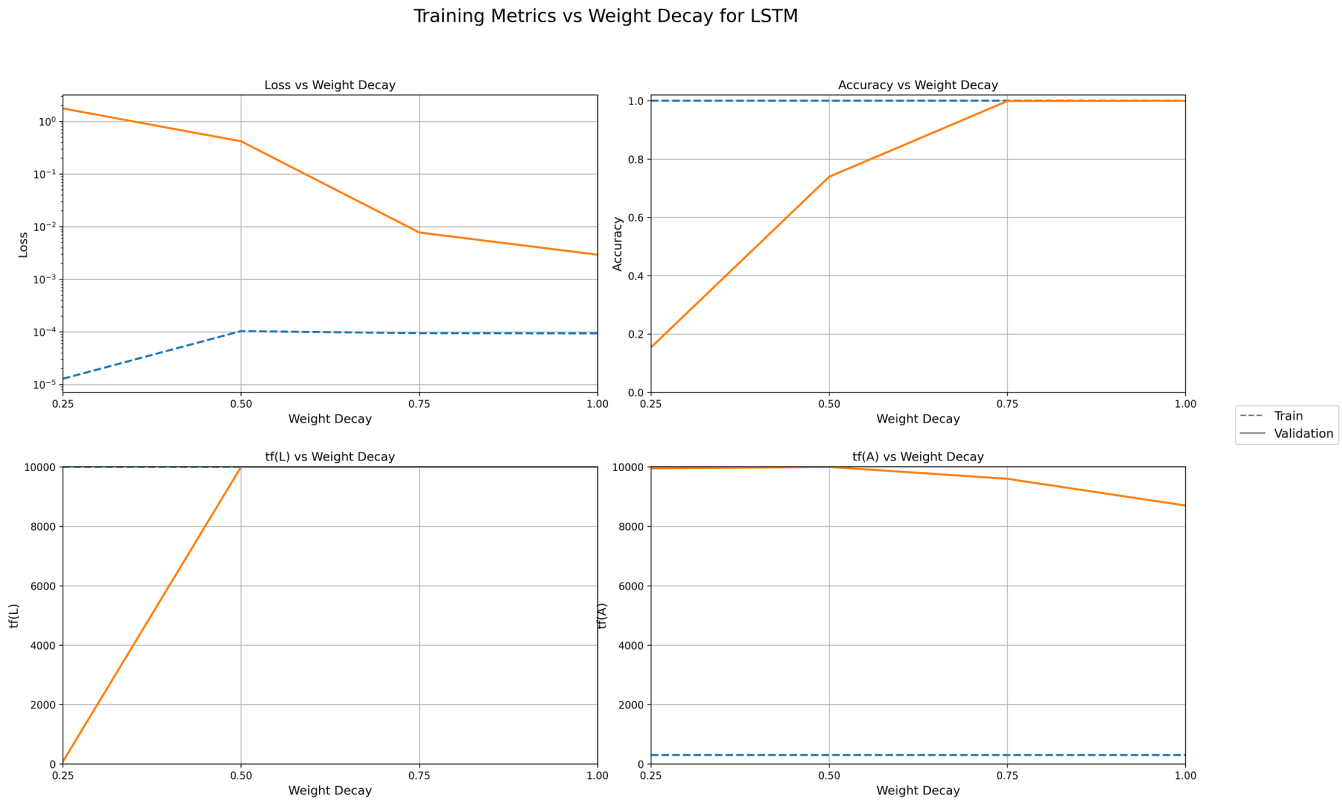

Figure 30: LSTM training and validation loss, accuracy, $t_f(\mathcal{L})$ and $t_f(\mathcal{A})$ as a function of weight decay.

(c) ($0.25 \times 4 = 1$pt) How does $\mathcal{L}_{\text{val}}$, $\mathcal{A}_{\text{val}}$, $t_f(\mathcal{L}_{\text{val}})$ and $t_f(\mathcal{A}_{\text{val}})$ change with the `weight_decay`?

i. $\mathcal{L}_{\text{val}}$
The validation loss decreases quicker to lower values with an increase in weight decay.

ii. $\mathcal{A}_{\text{val}}$
The validation accuracy increases quicker to higher values with an increase in weight decay.

iii. $t_f(\mathcal{L}_{\text{val}})$
The minimum loss value is reached at later steps with in an increase in weight decay, plateauing at the 10000^{th} step for values of weight decay >= .50.

iv. $t_f(\mathcal{A}_{\text{val}})$
The maximum accuracy is reached at earlier time steps for greater weight decay values.

(d) (2pts) Did you observe any change in $\|\theta^{(t)}\|_2$ before and when the models generalize? If so, describe it. How can you explain your observations?

$\|\theta^{(t)}\|_2$ begins at relatively large values at early time steps. As the model generalizes, the magnitude of weights decreases, more so with higher weight decay values. This correlates to an improvement in the validation loss and accuracy.

With higher values of $\|\theta^{(t)}\|_2$, models tend to be more complex and overfit to data. As the magnitude of weight decreases, the model generalizes better because smaller values of $\|\theta^{(t)}\|_2$ promote robustness to unseen data.

Interpretability (9pts): Consider a GPT model that generalized (e.g., the model trained in Question 1). Choose $\texttt{B} = 2$ samples of your choice from the training set (the sources sequences "BOS a + b = r"), and compute the attention weights $\boldsymbol{A}$ on these samples. The GPT model takes an input $\mathbf{x} = \mathbf{x}_1 \cdots \mathbf{x}_\texttt{S} \in \{0, \ldots, \texttt{V} - 1\}^{\texttt{B} \times \texttt{S}}$, and returns, in addition to the logits, the hidden states per layers, `hidden_states`, a tensor of shape `(B, num_layers, S, embedding_size)`; and the attention weights per layers, `attentions`, a tensor of shape `(B, num_layers, num_heads, S, S)`.

8. (9pts) The tensor to consider here is $\boldsymbol{A} = \texttt{attentions} \in [0, 1]^{\texttt{B} \times \texttt{num\_layers} \times \texttt{num\_heads} \times \texttt{S} \times \texttt{S}}$.

   (a) ($0.25 \times 8 \times 2 = 4$pts) For each sample $k \in [\texttt{B}]$ you choose, you need to create a grid of `num_layers` rows and `num_heads` columns. On each cell $(i, j) \in [\texttt{num\_layers}] \times [\texttt{num\_heads}]$ of the grid, display the attention weights $\boldsymbol{A}_{kij} \in [0, 1]^{\texttt{S} \times \texttt{S}}$ (with imshow or anything similar). Label your axes. The Figure **??** illustrates what you need to do; for the input sequence "BOS a + b = r", with the attention scores randomly generated. You can use `tokenizer.encode` (resp. `tokenizer.decode`) to convert sequences of tokens into token indexes (resp., vice versa), where `tokenizer` is the tokenizer returned by the function `get_arithmetic_dataset` during data creation.

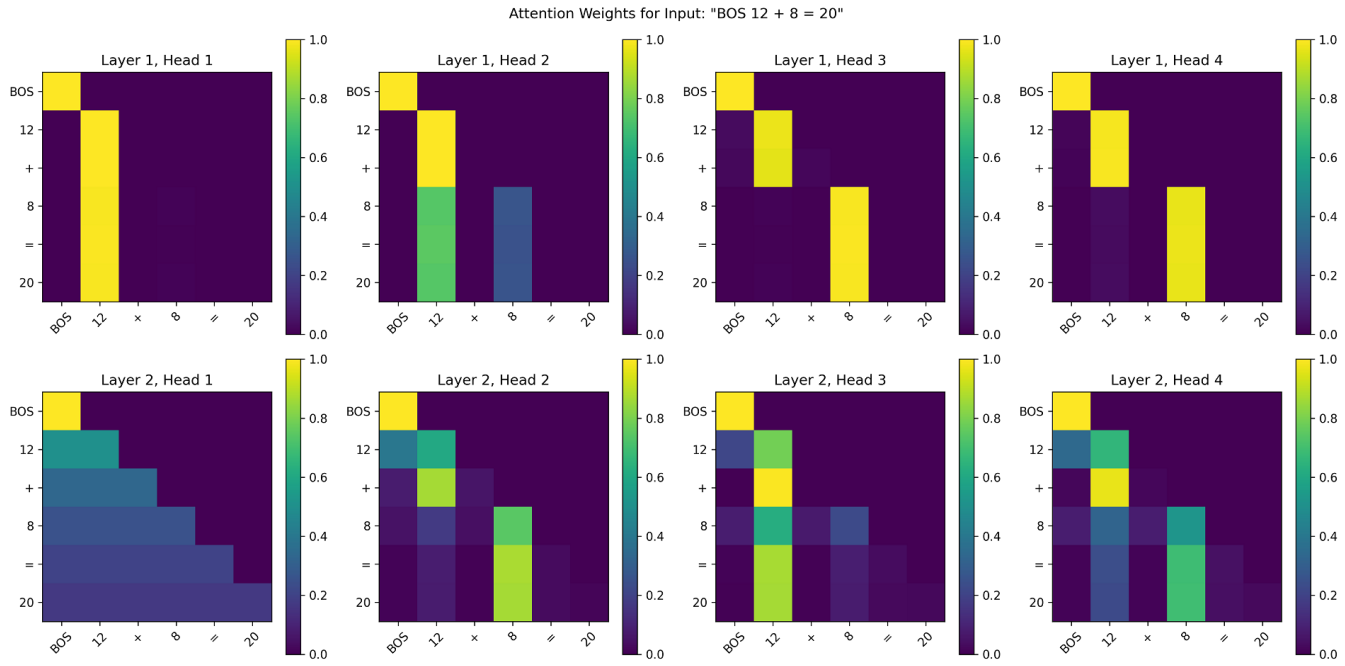

Figure 31: GPT attention weights for input sequence "BOS 12 + 8 = 20".

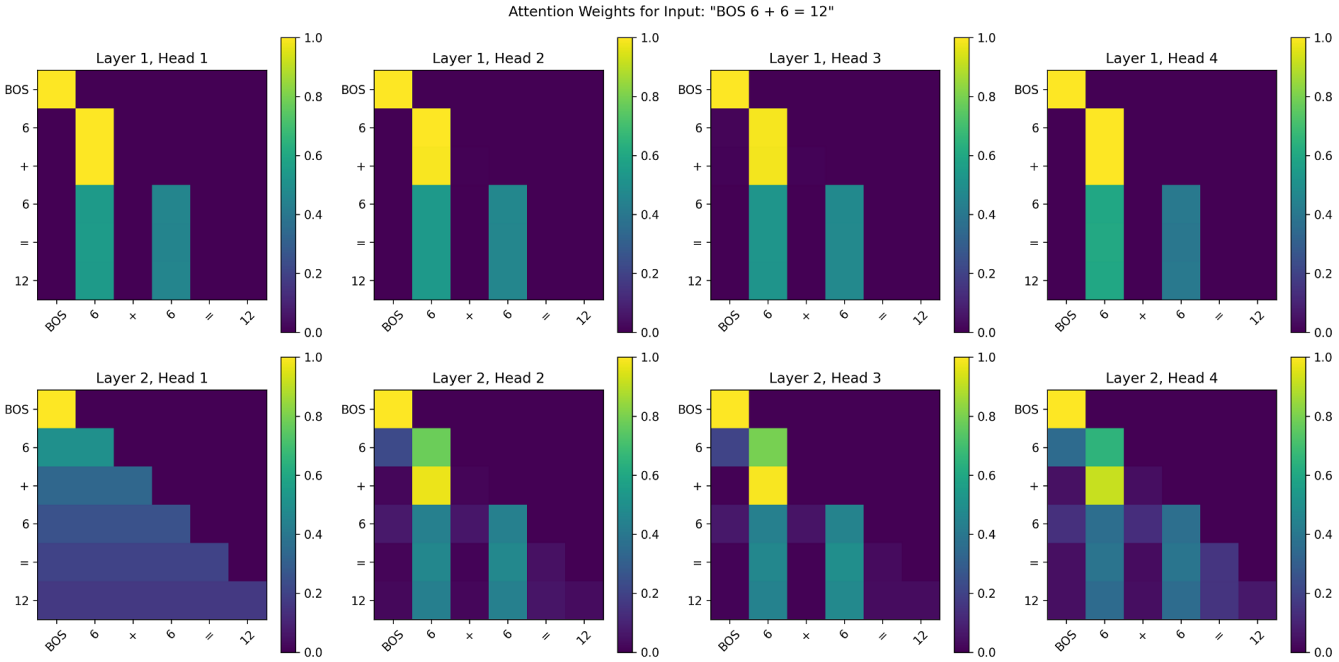

Figure 32: GPT attention weights for input sequence "BOS 6 + 6 = 12".

(b) (5pts) Are there tokens in particular positions that other tokens focus must attention on? If so, which ones? Does this apply to all heads/layers or just to particular heads/layers? How can you explain this? Be brief in you answer.

The operand tokens tend to receive more attention, in particular from the "=" token which relies on them to compute the final result. The operators also receive attention, but more so in layer 2 than in layer 1, suggesting deeper of the overall operation in higher layers.

The attention patterns are not consistent across layers, with layer 2 demonstrating a more diverse attention pattern. This reflects the hierarchical processing in the attention layers of transformers. Lower layers pick up on basic interactions (e.g. operand pairs) whereas higher layers pick up on global context.