**Due Date: Friday, 28th March, 11pm ET**

# Problem 1

**Implementing an LSTM (10pts)** In this problem, you will implement an LSTM model from scratch without using `torch.nn.LSTM`. Instead, you will manually implement the **gates and cell state updates** following the standard LSTM equations.

**LSTM Cell (3pts)**: An LSTM cell consists of the following key components:

- **Input gate**: Determines how much new information should be added to the memory.
- **Forget gate**: Determines how much of the previous memory should be retained.
- **Cell state update**: Updates the long-term memory.
- **Output gate**: Controls what part of the memory is exposed as output.
- **Hidden state update**: The new hidden state of the LSTM.

For each time step $t$, given the input $\mathbf{x}_t \in \mathbb{R}^{d_\text{x}}$ and the previous hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^{d_\text{h}}$ and cell state $\mathbf{c}_{t-1} \in \mathbb{R}^{d_\text{h}}$, the LSTM updates as follows:

$$\mathbf{i}_t = \sigma\left(\mathbf{W}_{[i]}\mathbf{x}_t + \mathbf{U}_{[i]}\mathbf{h}_{t-1} + \mathbf{b}_{[i]}\right) = \sigma\left(\begin{bmatrix}\mathbf{W}_{[i]} & \mathbf{U}_{[i]}\end{bmatrix}\begin{bmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{bmatrix} + \mathbf{b}_{[i]}\right) \quad \text{(Input gate)}$$

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_{[f]}\mathbf{x}_t + \mathbf{U}_{[f]}\mathbf{h}_{t-1} + \mathbf{b}_{[f]}\right) = \sigma\left(\begin{bmatrix}\mathbf{W}_{[f]} & \mathbf{U}_{[f]}\end{bmatrix}\begin{bmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{bmatrix} + \mathbf{b}_{[f]}\right) \quad \text{(Forget gate)}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{[o]}\mathbf{x}_t + \mathbf{U}_{[o]}\mathbf{h}_{t-1} + \mathbf{b}_{[o]}) = \sigma\left(\begin{bmatrix}\mathbf{W}_{[o]} & \mathbf{U}_{[o]}\end{bmatrix}\begin{bmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{bmatrix} + \mathbf{b}_{[o]}\right) \quad \text{(Output gate)}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{[c]}\mathbf{x}_t + \mathbf{U}_{[c]}\mathbf{h}_{t-1} + \mathbf{b}_{[c]}) = \sigma\left(\begin{bmatrix}\mathbf{W}_{[c]} & \mathbf{U}_{[c]}\end{bmatrix}\begin{bmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{bmatrix} + \mathbf{b}_{[c]}\right) \quad \text{(Candidate cell state)}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \text{(Cell state update)}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \text{(Hidden state update)}$$

where $\sigma$ is the sigmoid activation, and $\odot$ represents element-wise multiplication. For each $\mathbf{W} \in \mathbb{R}^{d_\text{h} \times d_\text{x}}$ and $\mathbf{U} \in \mathbb{R}^{d_\text{h} \times d_\text{h}}$, $\begin{bmatrix}\mathbf{W} & \mathbf{U}\end{bmatrix} \in \mathbb{R}^{d_\text{h} \times (d_\text{x}+d_\text{h})}$ is there concatenation along the second dimension (input dimension), and is treated in the code as a single linear transformation (with the corresponding bias), and the input of such linear layer is $\begin{bmatrix}\mathbf{x}_t \\ \mathbf{h}_{t-1}\end{bmatrix} \in \mathbb{R}^{d_\text{x}+d_\text{h}}$, the concatenation of the current input and previous hidden state.

1. In the file `lstm.py`, you are given a skeleton for an LSTM cell (class `LSTMCell`) that uses separate linear layers for each gate (forget, input, candidate, and output). Fill in the missing code in the forward pass, i.e., the `forward()` function. This function return $\mathbf{h}_t$ and $\mathbf{c}_t$.

**LSTM (4pts)**: In a multilayer LSTM, several LSTM cells are stacked on top of one another. Each layer processes the entire input sequence, and the hidden state outputs from one layer serve as inputs to the next layer. In Question 1, we defined an LSTM cell that computes the next hidden state and cell $(\mathbf{h}_t, \mathbf{c}_t)$ for a single time step $t$ given the input $\mathbf{x}_t$ at time step $t$ and previous state and cell $(\mathbf{h}_{t-1}, \mathbf{c}_{t-1})$. When processing a sequence $\mathbf{x} \in \mathbb{R}^{\mathtt{S} \times d_\mathrm{x}}$ of length $\mathtt{S}$, the LSTM cell is applied iteratively over time:

$$\mathbf{h}_t, \mathbf{c}_t = \mathrm{LSTMCell}(\mathbf{x}_t, (\mathbf{h}_{t-1}, \mathbf{c}_{t-1}))$$

for $t = 1, 2, \ldots, \mathtt{S}$. The output of each time step is collected to form the full sequence output. In a multilayer LSTM with $L$ layers, each layer processes the entire sequence one time step at a time. The process is as follows:

- Initialization : $\mathbf{h}_0^{(\ell)} \in \mathbb{R}^{\mathtt{B} \times d_\mathrm{h}}$ and $\mathbf{c}_0^{(\ell)} \in \mathbb{R}^{\mathtt{B} \times d_\mathrm{h}}$ are initialized (by default to 0) or provided, for each $\ell \in [L] := \{1, \ldots, L\}$.
- Layer 1:

$$\mathbf{h}_t^{(1)}, \mathbf{c}_t^{(1)} = \mathrm{LSTMCell}^{(1)}(\mathbf{x}_t, (\mathbf{h}_{t-1}^{(1)}, \mathbf{c}_{t-1}^{(1)})) \quad \forall t = 1, 2, \ldots, \mathtt{S}$$
$$\mathbf{h}^{(1)} = \mathrm{Dropout}(\mathbf{h}^{(1)}) \text{ if } L \geq 2$$

- Layer $\ell \geq 2$:

$$\mathbf{h}_t^{(\ell)}, \mathbf{c}_t^{(\ell)} = \mathrm{LSTMCell}^{(\ell)}(\mathbf{h}_t^{(\ell-1)}, (\mathbf{h}_{t-1}^{(\ell)}, \mathbf{c}_{t-1}^{(\ell)})) \quad \forall t = 1, 2, \ldots, \mathtt{S}$$
$$\mathbf{h}^{(\ell)} = \mathrm{Dropout}(\mathbf{h}^{(\ell)}) \text{ if } \ell < L$$

  Here, the input at time step $t$ for layer $\ell$ is the hidden state $\mathbf{h}_t^{(\ell-1)}$ produced by the previous layer at the same time step.

As you can see in the equations above, dropout is typically applied to the outputs between layers except the last (i.e., the hidden state outputs from one layer before they are fed into the next layer) to help prevent overfitting. Dropout is not applied on the outputs within a layer (i.e., across time steps) or on the final output layer.

For a batch of $\mathtt{B}$ sequences $\mathbf{x} \in \mathbb{R}^{\mathtt{B} \times \mathtt{S} \times d_\mathrm{x}}$, the outputs of the multilayer LSTM are :

- the full output for the sequence, $\left[\mathbf{h}_1^{(L)}, \ldots, \mathbf{h}_\mathtt{S}^{(L)}\right] \in \mathbb{R}^{\mathtt{B} \times \mathtt{S} \times d_\mathrm{h}}$
- the final hidden states and cell for each layers, $\left(\left[\mathbf{h}_\mathtt{S}^{(1)}, \ldots, \mathbf{h}_\mathtt{S}^{(L)}\right], \left[\mathbf{c}_\mathtt{S}^{(1)}, \ldots, \mathbf{c}_\mathtt{S}^{(L)}\right]\right) \in \mathbb{R}^{L \times \mathtt{B} \times d_\mathrm{h}} \times \mathbb{R}^{L \times \mathtt{B} \times d_\mathrm{h}}$

2. Using the LSTMCell from Question 1, complete the forward pass of the LSTM module that processes a sequence with a batch-first input. Your implementation should:

   - Initialize hidden and cell states to **zero** tensors if not provided: this is the part you need to implement (make sure they are on the same device and of the same type as the input tensor).
   - Loop over the time steps to update the hidden and cell states using the LSTMCell: this is the part you need to implement.
   - Apply dropout between layers (if dropout $> 0$) except for the last layer: this is already done for you.
   - Return the full output sequence and the final hidden and cell states: this is already done for you.

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

- $V$: Vocabulary size
- $d_e$: Embedding size
- $d_h$: Hidden size
- $L$: Number of LSTM layers
- $b \in \{0, 1\}$: Bias flag (1 if bias is enabled, 0 if disabled)

**Case 1: Bias = True**

| **Component** | **Parameter Count Expression** |
|---|---|
| Embedding Layer | $V \cdot d_e$ |
| LSTM Layer 1 (4 Gates) | $4(d_e + d_h)d_h + 4d_h$ |
| LSTM Layers 2 to $L$ | $(L-1)(8d_h^2 + 4d_h)$ |
| Classifier Layer | $V \cdot d_h + V$ |
| **Total** | $V \cdot d_e + 4(d_e + d_h)d_h + 4d_h + (L-1)(8d_h^2 + 4d_h) + V \cdot d_h + V$ |

Table 1: `bias` = `True`

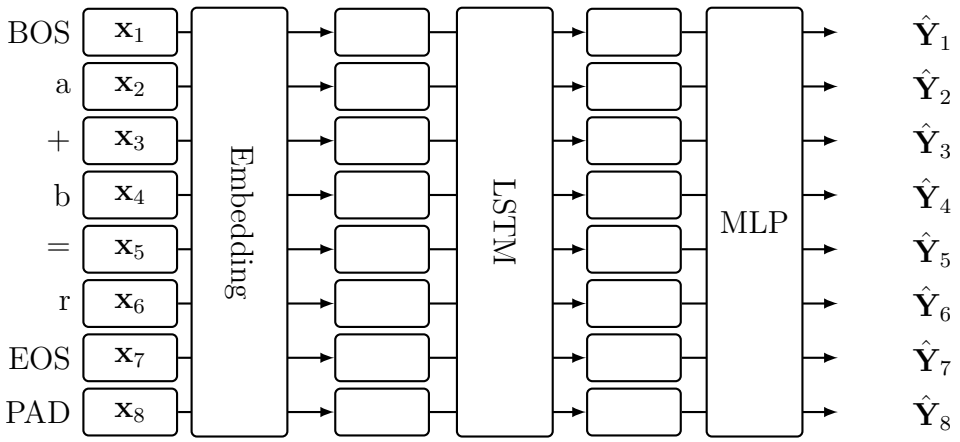
**Case 2: Bias = False**

| **Component** | **Parameter Count Expression** |
|---|---|
| Embedding Layer | $V \cdot d_e$ |
| LSTM Layer 1 | $4(d_e + d_h)d_h$ |
| LSTM Layers 2 to $L$ | $(L-1)(8d_h^2)$ |
| Classifier Layer | $V \cdot d_h$ |
| **Total** | $V \cdot d_e + 4(d_e + d_h)d_h + (L-1)(8d_h^2) + V \cdot d_h$ |

Table 2: `bias` = `False`.

**Full Language model (3pts)**: In the previous question, we saw that an LSTM takes an element in $\mathbb{R}^{\mathtt{S} \times d_x}$ as input. Except that we have a sequence $\mathbf{x} = \mathbf{x}_1 \cdots \mathbf{x}_\mathtt{S} \in \{0, \dots, \mathtt{V}-1\}^\mathtt{S}$, with $\mathtt{V}$ be the vocabulary size. We will equip our model with an Embedding layer to transform these tokens into vectors that the LSTM can understand. This embedding layer converts $\mathbf{x}$ into $\mathbf{C}(\mathbf{x}) = [\mathbf{C}(\mathbf{x}_1), \dots, \mathbf{C}(\mathbf{x}_\mathtt{S})] \in \mathbb{R}^{\mathtt{S} \times d_x}$, and it's this $\mathbf{C}(\mathbf{x})$ that's given to the LSTM. $\mathbf{C}(x) \in \mathbb{R}^{d_\mathtt{x}}$ is the unique embedding for the token $x \in \{0, \dots, \mathtt{V}-1\}$.

The architecture you will be asked to implement is the following:




Given the input sequence $\mathbf{x} = \mathbf{x}_1 \cdots \mathbf{x}_\mathtt{S} \in \{0, \dots, \mathtt{V}-1\}^\mathtt{S}$ of length $\mathtt{S}$, the model returns the logits $\hat{\mathbf{Y}} \in \mathbb{R}^{\mathtt{S} \times \mathtt{V}}$, such that $\hat{\mathbf{P}}_i = \text{softmax}\left(\hat{\mathbf{Y}}_i\right) \in [0,1]^\mathtt{V}$ represents the probability distribution over the vocabulary for the next token in the sequence given the context $\mathbf{x}_{\leq i} = \mathbf{x}_1 \cdots \mathbf{x}_i$. For example, on the figure above, $\hat{\mathbf{P}}_{ij} = \mathbb{P}[j \mid \text{BOS}, a, +, b, =] \in [0,1] \quad \forall j \in [\mathtt{V}]$.

In the file `lstm.py`, you are given an `LSTMLM` class containing all the blocks necessary to create this model. In particular, `self.embedding` is a `nn.Embedding` module that converts sequences of token indices into embeddings, `self.lstm` is a `LSTM` module that runs an LSTM over a sequence of vectors, and `self.classifier` is a 1-layer MLP responsible for classification.

4. Using these modules, complete the `forward()` function. This function must return the logits (and not the probabilities) of the next words in the sequence, as well as the final hidden states and cells of the LSTM. Note the input is a batch of sequence $\mathbf{x} \in \{0, \dots, \mathtt{V}-1\}^{\mathtt{B} \times \mathtt{S}}$.

# Problem 2

**Implementing a GPT (Generative-Pretrained Transformer) (31pts)** While typical RNNs "remember" past information by taking their previous hidden state as input at each step, recent years have seen a profusion of methodologies for making use of past information in different ways. The transformer[1] is one such fairly new architecture that uses several self-attention networks ("heads") in parallel, among other architectural specifics. Implementing a transformer is a fairly involved process, so we provide most of the boilerplate code, and your task is only to implement the multi-head scaled dot-product attention mechanism and the layer norm operation.

**Implementing Layer Normalization (5pts)**: You will first implement the layer normalization (LayerNorm) technique that we have seen in class. For this assignment, **you are not allowed** to use the PyTorch `nn.LayerNorm` module (nor any function calling `torch.layer_norm`).

As defined in the layer normalization paper, the layernorm operation over a minibatch of inputs $\mathbf{x}$ is defined as

$$\text{layernorm}(\mathbf{x}) = \frac{\mathbf{x} - \mathbb{E}[\mathbf{x}]}{\sqrt{\text{Var}[\mathbf{x}] + \epsilon}} * \texttt{weight} + \texttt{bias}$$

where $\mathbb{E}[\mathbf{x}]$ denotes the expectation over $\mathbf{x}$, $\text{Var}[\mathbf{x}]$ denotes the variance of $\mathbf{x}$, both of which are only taken over the last dimension of the tensor $\mathbf{x}$ here. `weight` and `bias` are learnable affine parameters.

[1] See https://arxiv.org/abs/1706.03762 for more details.

1. In the file `gpt.py`, implement the `forward()` function of the `LayerNorm` class. Pay extra attention to the lecture slides on the exact details of how $\mathbb{E}[\mathbf{x}]$ and $\mathrm{Var}[\mathbf{x}]$ are computed. In particular, PyTorch's function `torch.var` uses an unbiased estimate of the variance by default, defined as the formula on the left-hand side

$$\overline{\mathrm{Var}}(X)_{\text{unbiased}} = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2 \qquad \overline{\mathrm{Var}}(X)_{\text{biased}} = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2$$

whereas LayerNorm uses the biased estimate on the right-hand size (where $\overline{X}$ here is the mean estimate). Please refer to the docstrings of this function for more information on input/output signatures.

**Implementing the attention mechanism (17pts)**: You will now implement the core module of the transformer architecture – the multi-head attention mechanism. Assuming there are $m$ attention heads, the attention vector for the head at index $i$ is given by:

$$[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m] = \boldsymbol{Q}\boldsymbol{W}_Q + \boldsymbol{b}_Q \qquad [\boldsymbol{k}_1, \ldots, \boldsymbol{k}_m] = \boldsymbol{K}\boldsymbol{W}_K + \boldsymbol{b}_K \qquad [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m] = \boldsymbol{V}\boldsymbol{W}_V + \boldsymbol{b}_V$$

$$\boldsymbol{A}_i = \text{softmax}\left(\frac{\boldsymbol{q}_i \boldsymbol{k}_i^\top}{\sqrt{d}}\right)$$

$$\boldsymbol{h}_i = \boldsymbol{A}_i \boldsymbol{v}_i$$

$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m)\boldsymbol{W}_O + \boldsymbol{b}_O$$

Here $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are queries, keys, and values respectively, where all the heads have been concatenated into a single vector (e.g. here $\boldsymbol{K} \in \mathbb{R}^{\mathtt{S} \times md}$, where $d$ is the dimension of a single key vector, and `S` the length of the sequence). $\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V$ are the corresponding projection matrices (with biases $\boldsymbol{b}$), and $\boldsymbol{W}_O$ is the output projection (with bias $\boldsymbol{b}_O$). $\boldsymbol{Q}, \boldsymbol{K}$, and $\boldsymbol{V}$ are determined by the output of the previous layer in the main network. $\boldsymbol{A}_i$ are the attention values, which specify which elements of the input sequence each attention head attends to. We recommend you to check this tutorial for implementation details of GPT. **You are not allowed** to use the module `nn.MultiheadAttention` (or any function calling `torch.nn.functional.multi_head_attention_forward`). Please refer to the docstrings of each function for a precise description of what each function is expected to do, and the expected input/output tensors and their shapes.

2. The equations above require many vector manipulations in order to split and combine head vectors together. For example, the concatenated queries $\boldsymbol{Q}$ are split into $m$ vectors $[\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m]$ (one for each head) after an affine projection by $\boldsymbol{W}_Q$, and the $\boldsymbol{h}_i$'s are then concatenated back for the affine projection with $\boldsymbol{W}_O$. In the class `MultiHeadedAttention`, implement the utility functions `split_heads()` and `merge_heads()` to do both of these operations, as well as a transposition for convenience later. For example, for each $(i, j, k, l) \in \{0, \ldots, \texttt{B} - 1\} \times \{0, \ldots, \texttt{num\_heads} - 1\} \times \{0, \ldots, \texttt{S} - 1\} \times \{0, \ldots, d - 1\}$:

```
y = split_heads(x)  →  y[i, j, k, l] = x[i, k, num_heads * j + l]
x = merge_heads(y)  →  x[i, k, num_heads * j + l] = y[i, j, k, l]
```

These two functions are exactly inverse of one another. Note that the number of heads $m$ in the code is called `self.num_heads`, and the head dimension $d$ is `self.head_size`. Your functions must handle mini-batches of sequences of vectors; see the docstring for details about the input/output signatures.

3. In the class `MultiHeadedAttention`, implement the function `get_attention_weights()`, which is responsible for returning $\boldsymbol{A}_i$'s (for all the heads at the same time) from $\boldsymbol{q}_i$'s and $\boldsymbol{k}_i$'s. Remember that the language model here is auto-regressive (also sometimes called causal), meaning that the attention must be computed only on past inputs and not the future.

   Concretely, you will implement sequence masking to ensure that the model does not attend to future tokens when computing attention scores. This is an essential step in autoregressive models like GPT, where the prediction for a given token should only depend on previous tokens and not on future tokens. When using self-attention, each token in a sequence computes attention scores with every other token in the sequence. However, in autoregressive settings, we **cannot allow** a token at position $t$ to depend on tokens at positions $t+1, t+2, \ldots$, since that would provide information about future words, violating causality.

   To enforce this constraint, you need to **apply a lower triangular mask** to the attention scores. This mask ensures that each token can only attend to itself and previous tokens. Given the raw attention scores $\boldsymbol{s}_i = \frac{\boldsymbol{q}_i \boldsymbol{k}_i^\top}{\sqrt{d}} \in \mathbb{R}^{\mathtt{S}\times\mathtt{S}}$; you will introduce a mask $\mathbf{M} \in \mathbb{R}^{\mathtt{S}\times\mathtt{S}}$ where:

$$\mathbf{M}_{tk} = \begin{cases} 1 & \text{if } k \leq t \text{ (allow attention)} \\ 0 & \text{if } k > t \text{ (mask future tokens)} \end{cases}$$

   You will then apply this mask to the attention scores by setting masked positions ($\mathbf{M}_{tk} = 0$) to a very large negative value (you will use $-\infty$), which ensures they receive zero probability (attention weigths) after the softmax operation $\boldsymbol{A}_i = \text{softmax}(\boldsymbol{s}_i)$. You are strongly recommended to use vectorized operations as much as possible in order to speed up training in Problem 4.

   Note that this function also returns the attention weights $\boldsymbol{A}_i$, for each sequence in the mini-batch and for each head, so the output should have a shape $(\mathtt{B}, \mathtt{num\_heads}, \mathtt{S}, \mathtt{S})$ (see the variable `attn_weights`).

4. Using the functions you have implemented, complete the function `apply_attention()` in the class `MultiHeadedAttention`, which computes the vectors $\boldsymbol{h}_i$'s as a function of $\boldsymbol{q}_i$'s, $\boldsymbol{k}_i$'s and $\boldsymbol{v}_i$'s, and concatenates the head vectors.

$$\texttt{apply\_attention}(\{\boldsymbol{q}_i\}_{i=1}^m, \{\boldsymbol{k}_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m) \texttt{ = } \text{concat}(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_m).$$

5. Using the functions you have implemented, complete the function `forward()` in the class `MultiHeadedAttention`. You may implement the different affine projections however you want (do not forget the biases), and you can add modules to the `__init__()` function.

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

   Let:

   - $d_{\text{model}} = \texttt{num\_heads} \times \texttt{head\_size}$: total input/output dimension
   - $h$: number of heads
   - $d_h$: head size (i.e., dimensionality per head)
   - So, $d_{\text{model}} = h \cdot d_h$

   The multi-head attention mechanism includes four linear projections:

   - $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$: Query, Key, and Value projections
   - $\mathbf{W}_O$: Output projection

   Each of these has:

   - Weight matrix of shape $(d_{\text{model}}, d_{\text{model}})$
   - Optional bias vector of shape $(d_{\text{model}})$

   **Case 1: Bias = True**

| **Component** | **Parameter Count Expression** |
|---|---|
| Each Linear Projection | $d_{\text{model}}^2 + d_{\text{model}}$ |
| All 4 Projections | $4 \cdot (d_{\text{model}}^2 + d_{\text{model}})$ $= 4 \cdot [(h \cdot d_h)^2 + (h \cdot d_h)]$ |
| **Total** | $\boxed{4 \cdot \left[(h \cdot d_h)^2 + (h \cdot d_h)\right]}$ |

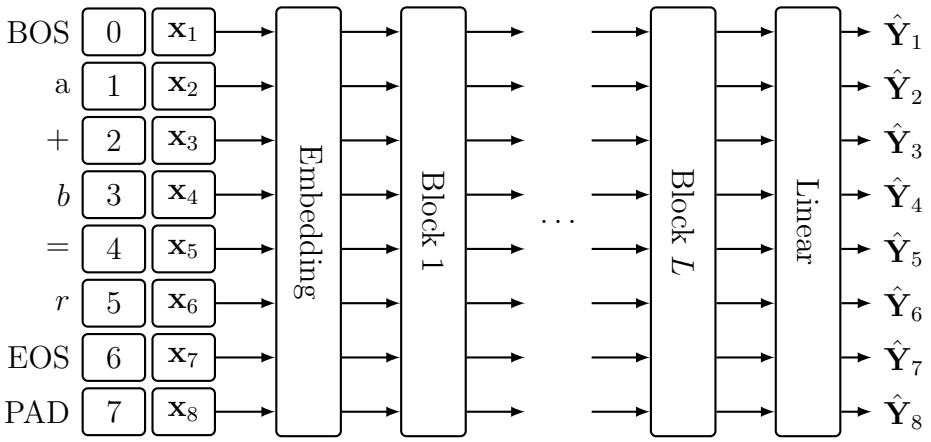
Table 3: Total learnable parameters in MHSA with bias

   **Case 2: Bias = False**

| **Component** | **Parameter Count Expression** |
|---|---|
| Each Linear Projection | $d_{\text{model}}^2$ |
| All 4 Projections | $4 \cdot d_{\text{model}}^2$ $= 4 \cdot (h \cdot d_h)^2$ |
| **Total** | $\boxed{4 \cdot (h \cdot d_h)^2}$ |

Table 4: Total learnable parameters in MHSA without bias

**The GPT forward pass (9pts)**: You now have all the building blocks to implement the forward pass of a miniature GPT model. You are provided a module `Block`, which corresponds to a full block with self-attention and a feed-forward neural network, with skip-connections, using the modules `LayerNorm` and `MultiHeadedAttention` you implemented before. You also have module `Decoder`, which contains a `nn.ModuleList` containing the different `Block` layers. As an illustration, the architecture of the GPT model is the following:




In this part of the exercise, you will fill in the classes `GPTEmbedding` and `GPT` in `gpt.py`. This module `GPT` contains all the blocks necessary to create the Transformer model. In particular, `self.embedding` is a module responsible for converting sequences of token indices into embeddings (using input and positional embeddings), `self.decoder` is a list of block of the Transformer (`Block`), and `self.classifier` is a linear layer responsible for classification.

7. Unlike recurrent models like LSTM, Transformers do not have an inherent sense of order in the input sequence, as they process tokens in parallel. To provide positional information to the model, you need to introduce positional encodings, which are added to the token embeddings. One effective way to achieve this is through sinusoidal positional encoding, defined as:

$$\texttt{PE}_{(\text{pos},i)} = \begin{cases} \sin\left(\frac{pos}{10000^{\frac{k}{d}}}\right) & \text{if } i = 2k \\ \cos\left(\frac{pos}{10000^{\frac{k}{d}}}\right) & \text{if } i = 2k+1 \end{cases}$$

where pos is the token's position in the sequence, $i$ indexes the embedding dimensions, and $d$ is the total embedding dimension. The alternating sine and cosine functions is used to create a unique representation for each position.

Unlike learned positional embeddings, sinusoidal embeddings are fixed and do not require training (hence the use of `register_buffer` to create this module in `GPTEmbedding`). They provide a continuous and smooth representation of positions, making interpolation possible for unseen sequence lengths. Plus, the formulation allows relative positional information to be easily learned.

In the class `GPTEmbedding`, complete the implementation of the `create_sinusoidal_embeddings` method, which constructs the positional encoding matrix. Ensure that the returned tensor has the prescribed shape.

8. The final embedding of an input sequence is the sum of the token embeddings, which provide semantic meaning, and the positional encodings, which provide order information. The forward method retrieves the token embeddings, extracts the corresponding positional encodings, and returns their sum.

   In the class `GPTEmbedding`, complete the `forward` method to correctly retrieve and combine the token embeddings and the positional encodings.

9. In the class `GPT`, complete the function `forward()` using the different modules described above. This function returns the logits (and not the log-probabilities as in the illustrative figure above) of the next words in the sequence. It also returns

   - the hidden states per layers, `hidden_states`, a tensor of shape `(B, num_layers, S, embedding_size)`
   - the attention weights per layers, `attentions`, a tensor of shape `(B, num_layers, num_heads, S, S)`

10. 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 $d_{\text{model}} =$ `num_heads` $\times$ `head_size`, and with $d_{\text{ff}} =$ `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 $d_{\text{model}}$, `num_heads`, `multiplier` and `L`? You should distinguish the case where `bias` = `True` and `bias` = `False`.

Each decoder block is composed of two main components:

- A `MultiHeadedAttention` module, consisting of four learnable linear projections: query ($W_Q$), key ($W_K$), value ($W_V$), and output ($W_O$).
- A bias-free feedforward network (FFN) composed of two linear layers with a non-linearity in between.

Let $d_{\text{model}} = d$ denote the dimensionality of the model, and let $d =$ `num_heads` $\times$ `head_size`. Let $m =$ `multiplier` be the expansion factor for the hidden layer in the FFN.

**1. Multi-Head Self-Attention (MHSA)**

Each of the four linear projections maps $\mathbb{R}^d \to \mathbb{R}^d$.

- With bias, each linear layer has $d \times d + d$ parameters.
- Without bias, each linear layer has $d \times d$ parameters.

Thus, the total number of parameters in the MHSA module is:

$$P_{\text{MHSA}}^{(\text{with bias})} = 4 \times (d^2 + d) = 4d^2 + 4d$$
$$P_{\text{MHSA}}^{(\text{no bias})} = 4 \times d^2 = 4d^2$$

**2. Feedforward Network (FFN)**

The FFN consists of two linear layers:

$$\text{Linear}_1 : \mathbb{R}^d \to \mathbb{R}^{md}, \quad \text{Linear}_2 : \mathbb{R}^{md} \to \mathbb{R}^d$$

Since both layers are defined without bias, the total parameter count is:

$$P_{\text{FFN}} = d \times md + md \times d = 2d^2 m$$

**3. Total Parameters per Block**

Summing the contributions from MHSA and FFN, we obtain:

- With bias:
$$P_{\text{block}}^{(\text{with bias})} = 4d^2 + 4d + 2d^2 m$$
- Without bias:
$$P_{\text{block}}^{(\text{no bias})} = 4d^2 + 2d^2 m$$

**4. Total Parameters for $L$ Blocks**

If the model contains $L$ identical transformer blocks, the total number of parameters scales linearly with $L$:

- With bias:
$$P_{\text{total}}^{(\text{with bias})} = L \times (4d^2 + 4d + 2d^2m)$$
- Without bias:
$$P_{\text{total}}^{(\text{no bias})} = L \times (4d^2 + 2d^2m)$$

# Problem 3

### Implementing the loss function (10pts)

Consider the input $\mathbf{X}$ of shape `(B, S)`, the targets $\mathbf{Y}$ of shape `(B, S)`, the equality positions $\mathbf{P}$ of shape `(B,)`, the mask $\mathbf{M}$ of shape `(B, S)`. Given the logits $\hat{\mathbf{Y}} := \hat{\mathbf{Y}}(\theta, \mathbf{X})$ of shape `(B, S, V)` produce from $\mathbf{X}$ by a model (LSTM or GPT) parametrized by $\theta$, you need to compute the loss (negative log-likelihood) and the accuracy on the right hand side of each equation (excluding the PAD tokens) of the batch, and average the result. In `(B, S, V)`, `B` is the mini batch size, `S` is the sequence length, and `V` is the vocabulary size.

More precisely, for a sequence $\mathbf{y}$ of shape length `S`, let $\texttt{RHS}(\mathbf{y}) = \{j \in [S] \mid [=] < \mathbf{y}_j < [\text{PAD}]\}$ be the right hand side of $\mathbf{y}$ (excluding the PAD tokens)[2], where $[\texttt{S}] := \{1, \dots, \texttt{S}\}$. The loss on the mini batch is

$$\mathcal{L}(\theta, \mathbf{X}, \mathbf{Y}) = -\frac{1}{\texttt{B}} \sum_{i=1}^{\texttt{B}} \frac{1}{|\texttt{RHS}(\mathbf{Y}_i)|} \sum_{\substack{j=1 \\ j \in \texttt{RHS}(\mathbf{Y}_i)}}^{\texttt{S}} \sum_{k=1}^{\texttt{V}} \mathbf{1}(k = \mathbf{Y}_{ij}) \times \log \hat{\mathbf{P}}_{ijk}$$

where $\hat{\mathbf{P}}_{ij} = \text{softmax}\left(\hat{\mathbf{Y}}_{ij}\right) \in [0,1]^{\texttt{V}}$ for all $(i,j) \in [\texttt{B}] \times [\texttt{S}]$; and $\mathbf{1}(k = \mathbf{Y}_{ij})$ is the indicator function which equals 1 if $k = \mathbf{Y}_{ij}$, and 0 otherwise. Here, for each valid token $\mathbf{Y}_{ij}$ (i.e $j \in \texttt{RHS}(\mathbf{Y}_i)$), we seek to maximize $\hat{\mathbf{P}}_{ijk}$ only for the expected token $k = \mathbf{Y}_{ij}$. Doing this automatically minimizes the other components of $\hat{\mathbf{P}}_{ij}$ because it belongs to the $(\texttt{V}-1)$-dimensional probability simplex:

$$\left\{ (p_1, \dots, p_{\texttt{V}}) \in \mathbb{R}^{\texttt{V}} \mid p_i \geq 0, \sum_{i=1}^{\texttt{V}} p_i = 1 \right\}$$

The accuracy on the mini-batch is

$$\mathcal{A}(\theta, \mathbf{X}, \mathbf{Y}) = \frac{1}{\texttt{B}} \sum_{i=1}^{\texttt{B}} \prod_{\substack{j=1 \\ j \in \texttt{RHS}(\mathbf{Y}_i)}}^{\texttt{S}} \mathbf{1}\left( \mathbf{Y}_{ij} = \arg\max_{k \in [\texttt{V}]} \hat{\mathbf{Y}}_{ijk} \right)$$

That means if all the tokens on the right-hand side are correct, then the equation is correct. Otherwise (i.e, if any token is incorrectly predicted), the entire equation is incorrect.

[2] If there is no padding (`operation_orders` equals 2 or 3, not (2, 3)), then $\texttt{RHS}(\mathbf{y}) = \{j \in [\texttt{S}] \mid [=] < \mathbf{y}_j\}$, i.e all the token after the equality tokens.

For standard auto-regressive training (next token prediction), $\texttt{RHS}(\mathbf{y}) = \{j \in [S] \mid \mathbf{y}_j < [\text{PAD}]\}$ (i.e all tokens but the PAD token), and if there is no padding, $\texttt{RHS}(\mathbf{y}) = [S]$, (i.e. the whole sequence).

1. In the file `trainer.py`, complete the function `get_loss_and_accuracy(logits, targets, eq_positions, mask)`. This function returns the mean negative log-likelihood and the accuracy of the right-hand side of each equation in the minibatch.

   - You are strongly recommended to use vectorized operations as much as possible in order to speed up training in Problem 4: `gather`, etc.
   - Since you don't need $\hat{\mathbf{P}}$, but just $\log \hat{\mathbf{P}}$, we encourage you to use the function `F.log_softmax` that takes $\hat{\mathbf{Y}}$ and return $\log \hat{\mathbf{P}}$.
   - You can use `F.nll_loss`, but be aware that it changes the normalization. In the equation above, we computed the loss for each sample by averaging over its valid (RHS) tokens and then averaged these per-sample losses. In contrast, when you use `F.nll_loss`, say with `ignore_index` (and `reduction` set to `"mean"`), it averages over all valid tokens across the batch. This token-level average can lead to a slightly different numerical value when sequences have different numbers of valid tokens. Even if the models trained with these two versions will generalize, we encourage you to implement the loss as described above to avoid problems when evaluating on Gradescope.

   This function accepts a parameter `reduction`. If it is `'mean'`, you need to average over the mini-batch dimension like in the previous formula. If it is `'sum'`, there is no need to divide the result by `B`. If it is `'none'`, the function must return the loss and accuracy per sample, not the mean/sum over the mini-batch (i.e. $\frac{1}{\mathtt{B}}\sum_{i=1}^{\mathtt{B}}$ is exclude from the formula above): it that last case it returns two one-dimensionnal tensor of lengths `B` each.

# Problem 4

**Training language models and model comparison (44pts)**

The purpose of this question is to perform model exploration, which is done using a validation set. You will train each of the architectures implemented above using `AdamW` optimizer (default learning rate and weights decay of $10^{-3}$ and $10^{0}$, respectively). For reference, you have the following options for running your experiments.

- Using `run_exp.py`. For example

  ```
  python run_exp.py --model gpt --optimizer adamw --n_steps 10000 --log_dir ../logs
  ```

- Using `train.py`. It takes no arguments but contains a class `Arguments` that lets you modify the hyperparameters by yourself. You can execute it directly with your code editor (e.g. VS Code) or call it with

  ```
  python train.py
  ```

- Or follow the instructions in `README.md` to do your training (recommended).

You are free to modify the scripts above as you deem fit. You do not need to submit code for this part of the assignment. However, you are required to create a report that presents the training curve comparisons and other results requested in the following questions (discussions, etc).

For each experiment, closely observe the training curves and report the lowest loss and the maximum accuracy across steps (not necessarily for the last steps). When you are asked to plot the training performances/statistics, you need to plot the training and validation loss and accuracies as a function of training steps (eg, losses on one figure, accuracies on another). Figures should have labeled axes, a legend, and an explanatory caption. For tables, always bold the best results when a comparison is asked[3]. The table should also have an explanatory caption and appropriate column and/or row headers. Any shorthand or symbols in the figure/table should be explained in the caption.

You will use the following notation:

- $\texttt{N}_{\text{train}} \simeq r_{\text{train}} \times \texttt{N}$ is the number of training samples (we will call it datasize for short in the following), where `N` is the number of samples in the whole dataset (we compute it as a function of $p$ and `operation_orders` at the beginning of the assignment). By default, $(p, \texttt{operation\_orders}, r_{\text{train}}) = (31, 2, 0.5)$, and the operator $\circ$ is the addition $(+)$.

[3]You can also make the table in LaTeX; for convenience, you can use tools like LaTeX table generator to generate tables online and get the corresponding LaTeX code.

- `P` denoted the number of model parameters, excluding all tokens and positional embeddings (you compute it for LSTM and GPT in the previous sections). You can compute it with the following code.

```
n_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
n_params_embeddings = sum(
    p.numel() for p in model.embedding.parameters()
    if p.requires_grad)
P = n_params - n_params_embeddings
```

- $\mathtt{C} \simeq \mathtt{P} \times \mathtt{B} \times \mathtt{T}$ is an estimate of the total non-embedding training compute, where `B` is the batch size and `T` is the number of training steps. Unless otherwise specified, use $\mathtt{B} = 2^9$, $\mathtt{T} = 10^4 + 1$, and `P` as the number of non embedding parameters of a 2 layers model with embedding and hidden size $2^7$ (4 heads and $d_{\text{ff}} = 4 \times d_{\text{model}}$ for GPT, i.e. `multiplier` $= 4$), `bias` $=$ `True` (for LSTM cells, Attention module, and classifiers).

- $\mathcal{L} \geq 0$ denotes the negative log-likelihood compute in Problem 3 (we will called it loss for short), and $\mathcal{L}^{(t)}_{\text{train/val}}$ denotes the training/validation loss at step $t$ of training.
  - If you are asked for the evolution of the loss during training, this is the evolution of $\mathcal{L}^{(t)}_{\text{train/val}}$ as a function of $t$ (the average if you did the same experiment for different random seeds, and a standard deviations around it).
  - On the other hand, if you're asked for the best loss for a training run, you'll have to return $\mathcal{L}_{\text{train/val}} := \min_{t \leq \mathtt{T}} \mathcal{L}^{(t)}_{\text{train/val}}$, depending on whether you're asked for the training loss or the validation loss (the average if you did the same experiment for different random seeds, and a standard deviations around it).

- $\mathcal{A} \in [0, 1]$ is the accuracy. It obeys the same notations as those of the loss $\mathcal{L}$, with $\mathcal{A}_{\text{train/val}} := \max_{t \leq \mathtt{T}} \mathcal{A}^{(t)}_{\text{train/val}}$.

- $t_f(\mathcal{L}_{\text{train/val}})$ is the step at which the model achieve the loss $\mathcal{L}_{\text{train/val}}$ for the first time; same for $t_f(\mathcal{A}_{\text{train/val}})$ with the accuracy. The functions `get_extrema_performance_steps` and `get_extrema_performance_steps_per_trials` in `checkpointing.py` is available to compute these quantities for you.

- $\Delta t(\mathcal{L}) = t_f(\mathcal{L}_{\text{val}}) - t_f(\mathcal{L}_{\text{train}})$ and $\Delta t(\mathcal{A}) = t_f(\mathcal{A}_{\text{val}}) - t_f(\mathcal{A}_{\text{train}})$. For the model that generalizes, these quantities represent the delay between memorization (overfitting) and generalization.

Each training session you perform saves the model's and optimizer's checkpoints every `save_model_step` step (see files `exp_name_state_t_acc=`$\mathcal{A}_{\texttt{val}}$`_loss=`$\mathcal{L}_{\texttt{val}}$`.pth` in the checkpoint path, `log_dir/exp_id`) and training statistics (training/validation loss/accuracy, etc) every `save_statistic_step` step (see the file `exp_name.pth` in the checkpoint path), which you can retrieve using the functions in `checkpointing.py` (see the readme).

If you're working on Google Colab, we advise you to retrieve and save the files `exp_name.pth` associated with each training before your session ends, as these will be used to create your training

curves and report your results (you can also save them in your Google Drive from Google Colab). You'll rarely need the checkpoints to answer the questions below, so you can set `save_model_step` to $\infty$, so that only the last checkpoint will be saved and your memory won't be saturated (but if a question requires you to inspect the state of your model during training, after training, you should use a low value, say `T`$/10^3$, for this parameter).

For any other information you wish to save during training ($\ell_2$ norm of model parameters, etc), you can go in the function `eval_model` in `trainer.py` and add them to the dictionary returned by the function. This information will be saved for you in `exp_name.pth` (or `all_metrics`), this every `eval_period` steps.

**Note** For each experiment, you need to do it for two different random seeds, 0 and 42. You can add more random seeds to make your results more consistent if your computational resources allow it, but it is not required; two are enough for this part.

**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.

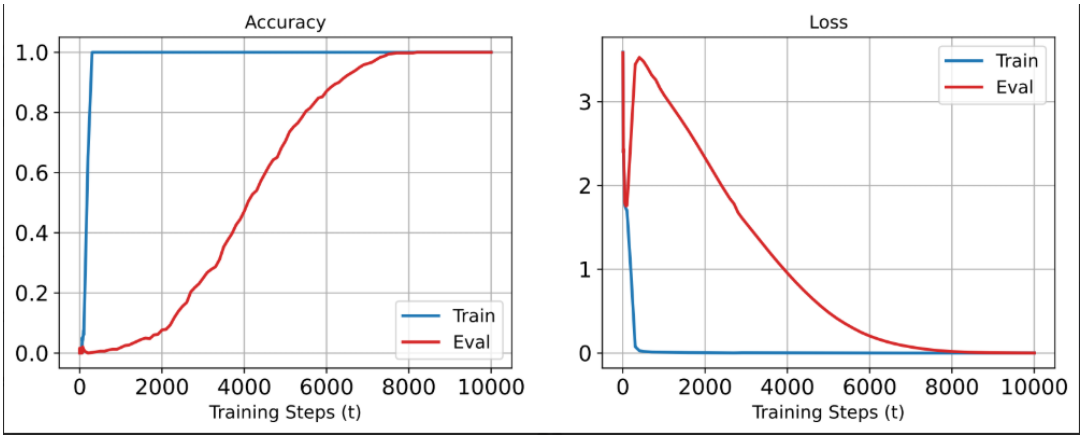



Figure 1: Loss and Accuracy of LSTM model.

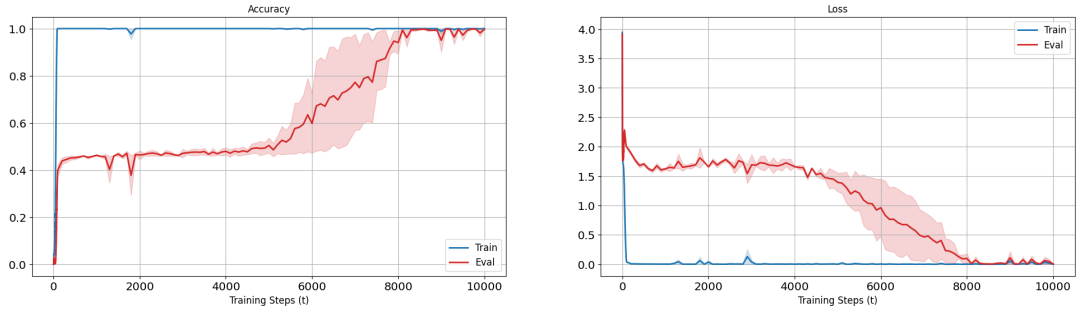



Figure 2: Loss and Accuracy of GPT model.

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` $\pm$ `std`.

**Results for LSTM**

| Metric | Value |
|---|---|
| $\mathcal{L}_{\text{train}}$ | $0.0001 \pm 0.0000$ |
| $\mathcal{L}_{\text{val}}$ | $0.0008 \pm 0.0004$ |
| $\mathcal{A}_{\text{train}}$ | $1.0000 \pm 0.0000$ |
| $\mathcal{A}_{\text{val}}$ | $1.0000 \pm 0.0000$ |
| $t_f(\mathcal{L}_{\text{train}})$ | $10001.0000 \pm 0.0000$ |
| $t_f(\mathcal{L}_{\text{val}})$ | $10001.0000 \pm 0.0000$ |
| $t_f(\mathcal{A}_{\text{train}})$ | $300.0000 \pm 0.0000$ |
| $t_f(\mathcal{A}_{\text{val}})$ | $7850.0000 \pm 350.0000$ |
| $\Delta t(\mathcal{L})$ | 0 |
| $\Delta t(\mathcal{A})$ | 7550 |

**Results for GPT**

| Metric | Value |
|---|---|
| $\mathcal{L}_{\text{train}}$ | $0.0001 \pm 0.0000$ |
| $\mathcal{L}_{\text{val}}$ | $0.0035 \pm 0.0031$ |
| $\mathcal{A}_{\text{train}}$ | $1.0000 \pm 0.0000$ |
| $\mathcal{A}_{\text{val}}$ | $0.9979 \pm 0.0021$ |
| $t_f(\mathcal{L}_{\text{train}})$ | $9750.0000 \pm 150.0000$ |
| $t_f(\mathcal{L}_{\text{val}})$ | $9100.0000 \pm 500.0000$ |
| $t_f(\mathcal{A}_{\text{train}})$ | $82.5000 \pm 5.5000$ |
| $t_f(\mathcal{A}_{\text{val}})$ | $8400.0000 \pm 200.0000$ |
| $\Delta t(\mathcal{L})$ | $-650$ |
| $\Delta t(\mathcal{A})$ | 8318 |

*This question is designed to familiarize you with the training and check that everything runs properly. Each training step (with evaluation) takes about on average $\tau = 0.05$ (resp. 0.06) seconds on Google Colab (with 1 GPU) for LSTM (resp. GPT). So for $T = 10^4 + 1$ steps, training could take around $\tau T \approx 5 \times 10^2$ (resp. $6 \times 10^2$) seconds ($\approx 10$ minutes); plus some latency due to checkpointing, etc. This information will not necessarily be the same for you (we provide a rough estimate to give you an idea). You need to use them to schedule your training sessions in the following questions (i.e., estimate the time you'll spend training the models). On Google Colab CPU (no GPU), we observe $\tau = 0.25$ (resp. 0.40), so you can always run the experiments on your own computer CPU, but each experiment can take $\tau T \approx 40$ (resp. 66) minutes for $T = 10^4 + 1$.*

**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 $r_{\text{train}}$).

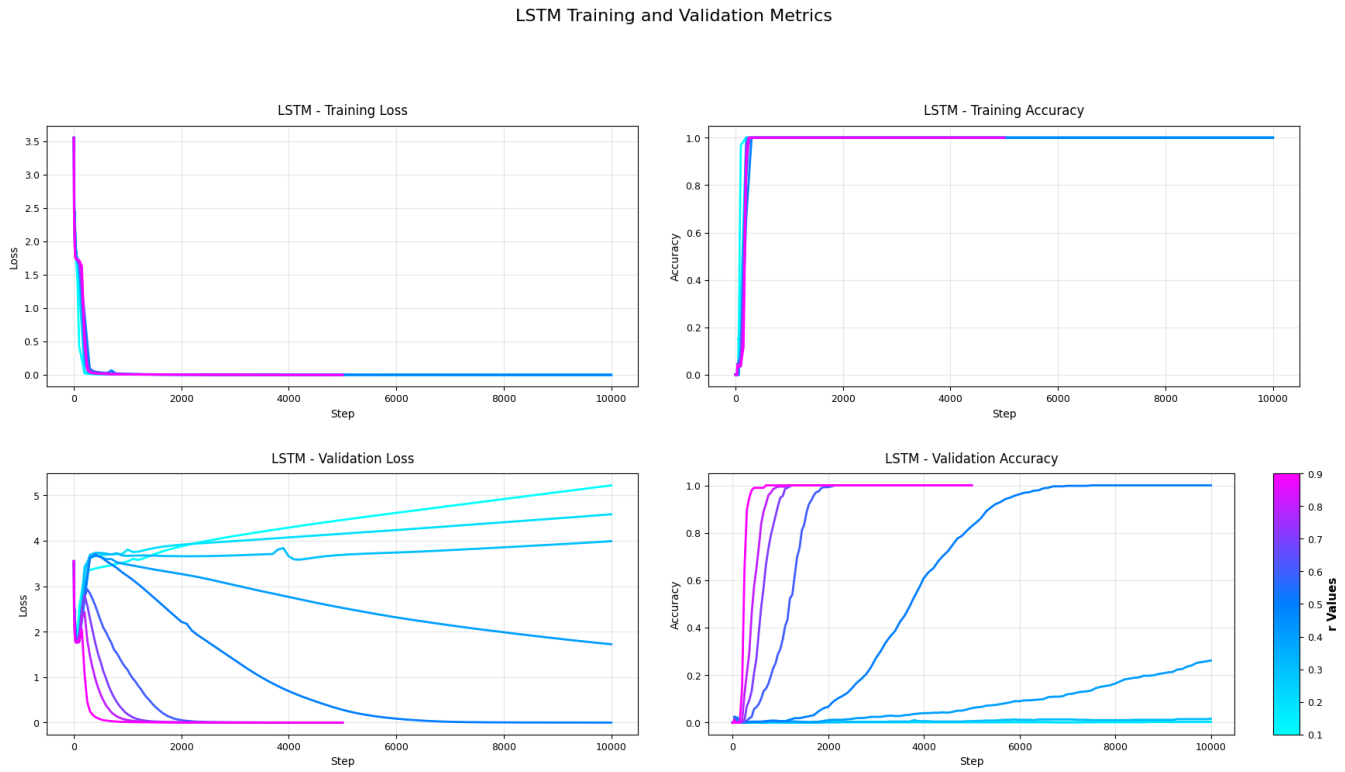



Figure 3: LSTM Training and Validation Metrics across different $r_{\text{train}}$ values. Gradient color encodes $r_{\text{train}}$ from 0.1 (light cyan) to 0.9 (magenta).

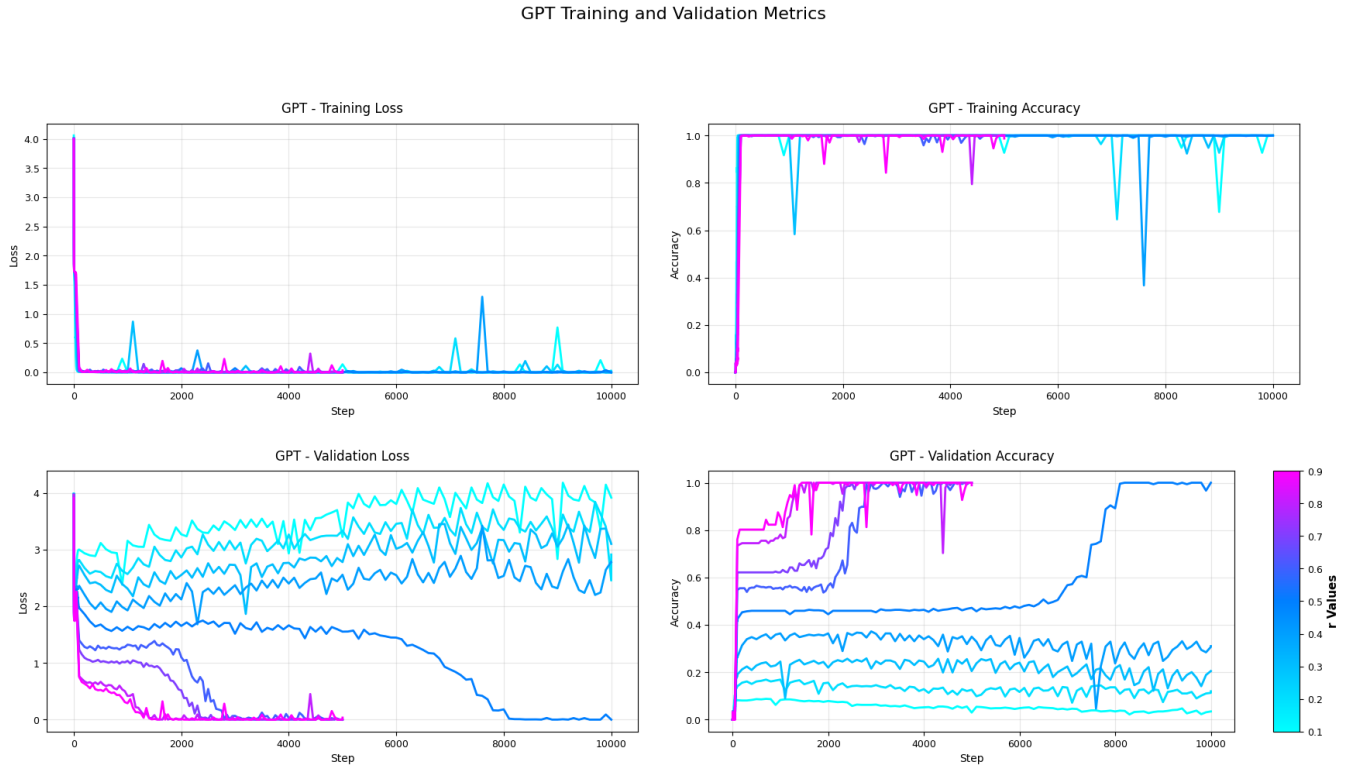



Figure 4: GPT Training and Validation Metrics across different $r_{\text{train}}$ values. Gradient color encodes $r_{\text{train}}$ from 0.1 (light cyan) to 0.9 (magenta).

(b) (0.25 × 8 = 2pts) 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 $r_{\text{train}}$. For $\mathcal{L}_{\text{train}}$ and $\mathcal{L}_{\text{val}}$, put the corresponding y-axis in log scale on your figures.

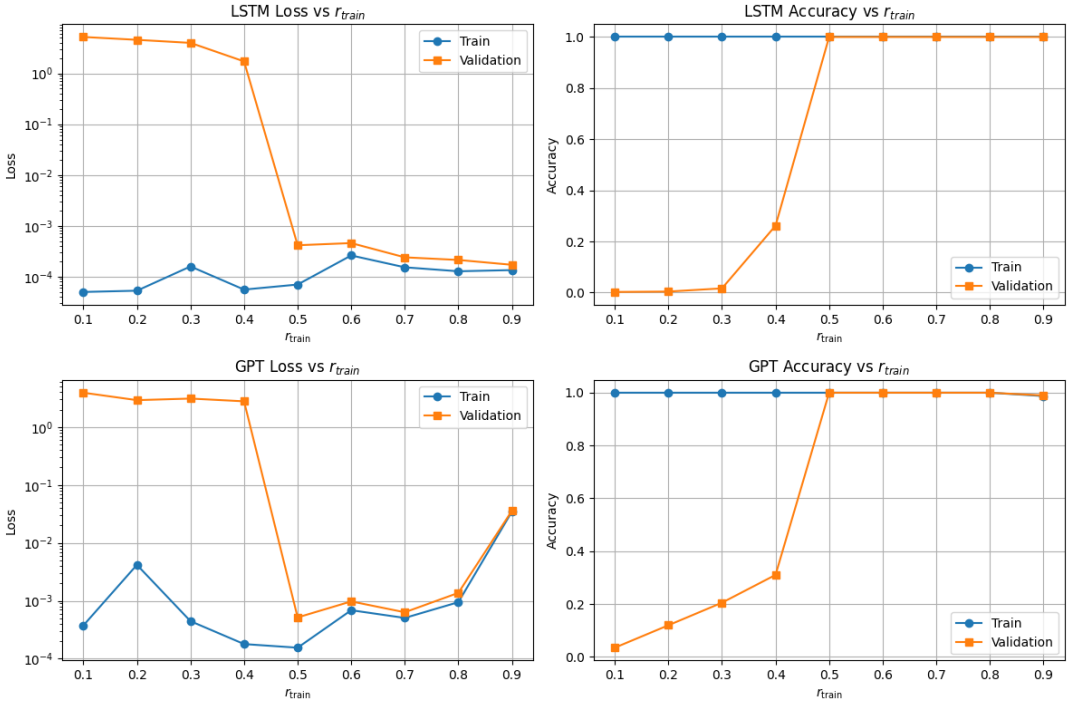



Figure 5: Comparative performance metrics versus training data ratio. Top: Loss trajectories (log scale). Bottom: Accuracy convergence.

(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 $\mathcal{A}_{\text{val}}$)?

**Answer:** The LSTM demonstrates superior scaling with increasing $r_{\text{train}}$, particularly in validation loss ($\mathcal{L}_{\text{val}}$) which decreases monotonically. For validation accuracy ($\mathcal{A}_{\text{val}}$), both models improve but the LSTM achieves more stable performance across all $r_{\text{train}}$ values.

(d) (2pts) What's the smallest $r_{\text{train}}$ 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)?

**Answer:** The LSTM first achieves $\mathcal{A}_{\text{val}} \geq 0.9$ at $r_{\text{train}} = 0.6$, while significant overfitting ($\mathcal{A}_{\text{train}} \approx 1.0$, $\mathcal{A}_{\text{val}} \leq 0.5$) occurs below $r_{\text{train}} = 0.3$. The 0.3 gap suggests the model needs substantial data to transition from memorization to genuine learning.

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.

```
import torch
from train import Arguments
from data import get_arithmetic_dataset

args = Arguments()

# Data
args.p=11
args.operation_orders = [2, 3]

(dataset, _), tokenizer, MAX_LENGTH, padding_index = get_arithmetic_dataset(
        args.p, args.p, args.operator, 1.0, args.operation_orders, seed=args.seed
)
vocabulary_size = len(tokenizer)

dataset_per_oders = {
    2 : torch.utils.data.Subset(
        dataset,
        [i for i in range(len(dataset)) if dataset[i][2] == 3]
    ), # a + b = r EOS PAD PAD
    3 : torch.utils.data.Subset(
        dataset,
        [i for i in range(len(dataset)) if dataset[i][2] == 5]
    ) # a + b + c = r EOS
}

# You can then split dataset_per_oders[2] and dataset_per_oders[3]
# based on 'args.r_train' with 'torch.utils.data.random_split',
# then combine each fraction (using 'torch.utils.data.ConcatDataset') to obtain
# "train_dataset" and "valid_dataset"
```

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

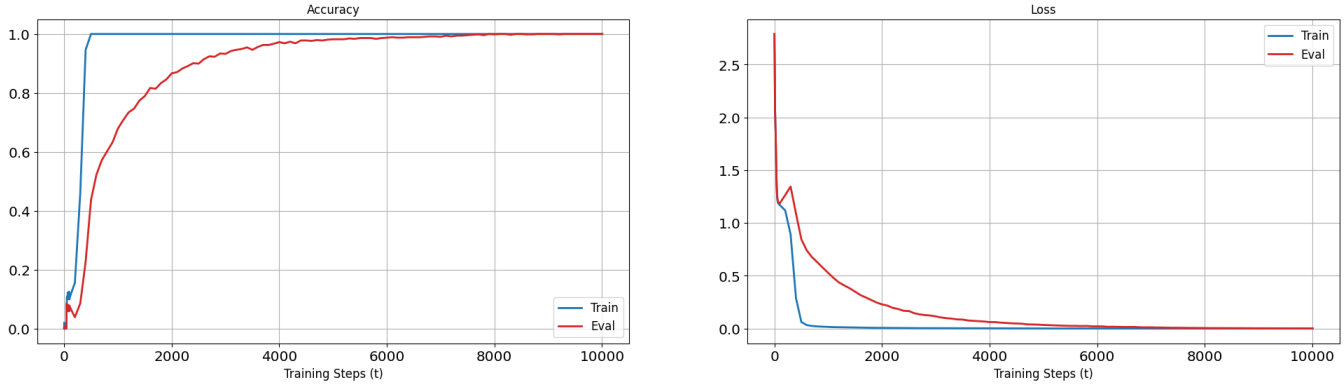



Figure 6: LSTM: Training and Validation Loss/Accuracy over time ($t$)

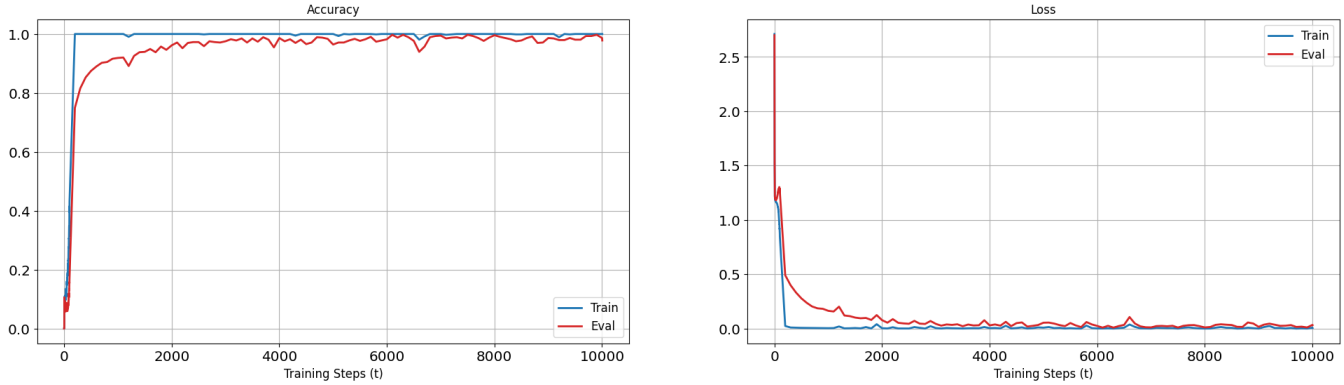



Figure 7: GPT: Training and Validation Loss/Accuracy over time ($t$)

(b) ($0.25 \times 4 \times 2 = 2$pts) 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.

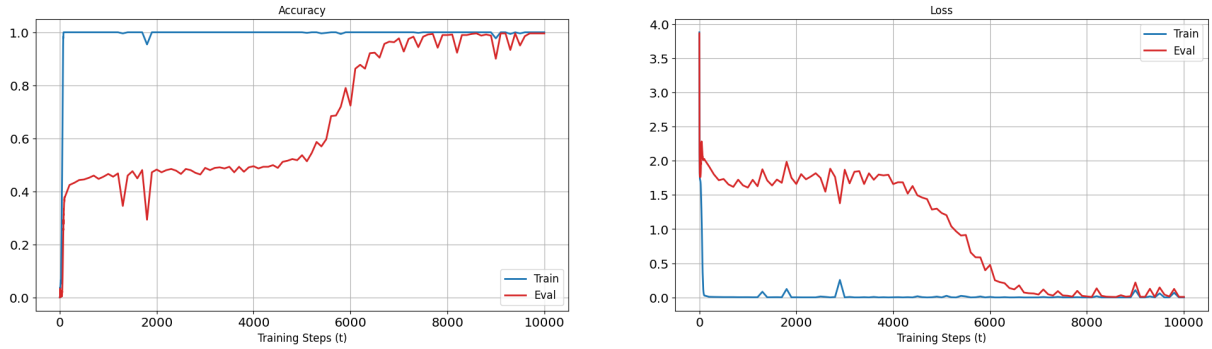



Figure 8: Accuracy and loss plots for the GPT model on binary operations.

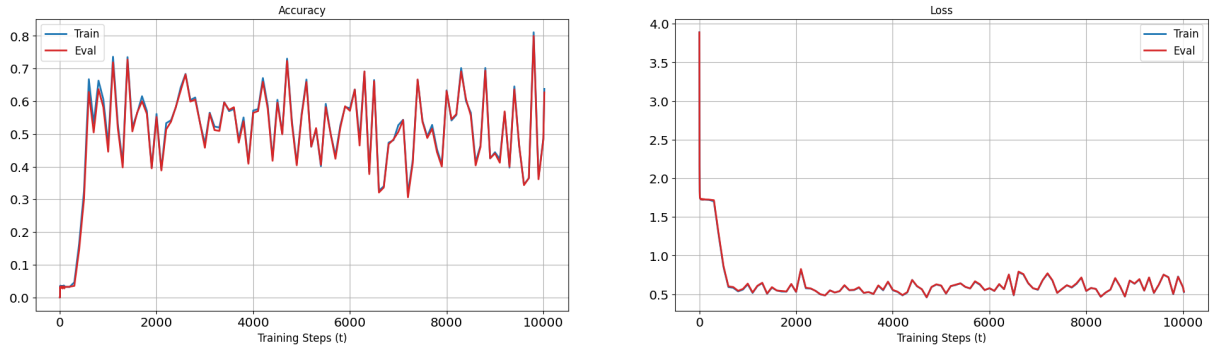



Figure 9: Accuracy and loss plots for the GPT model on ternary operations.

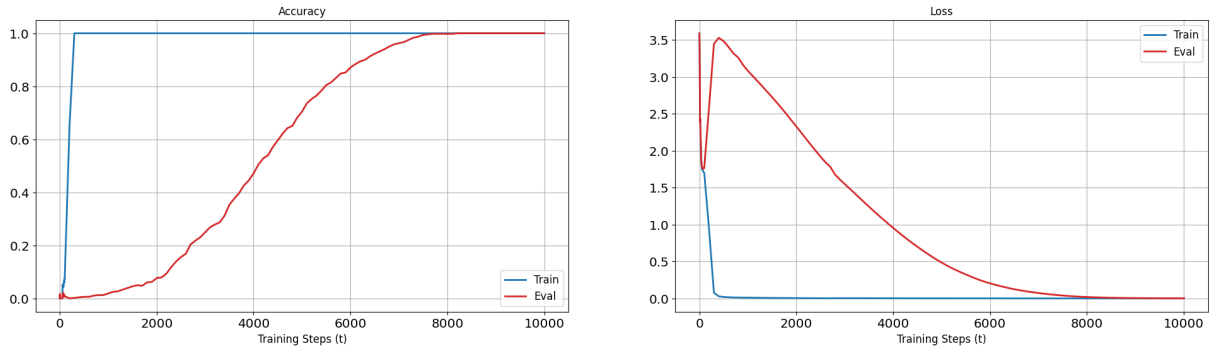



Figure 10: Accuracy and loss plots for the LSTM model on binary operations.

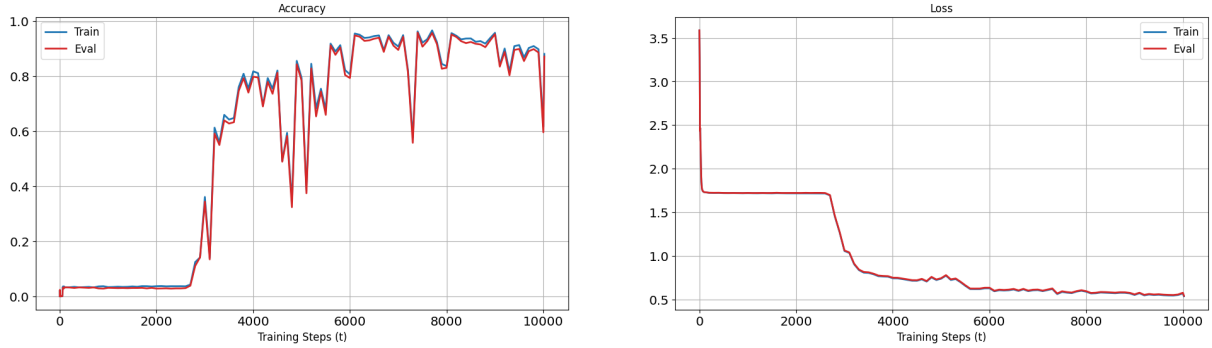



Figure 11: Accuracy and loss plots for the LSTM model on ternary operations.

(c) (1pt) During training, are certain types of operation predicted more quickly than others?
**Answer:** Binary operations are learned faster and more stably than ternary operations by the GPT model. In contrast, the LSTM model shows unstable learning overall, without clear preference or efficiency for either operation type.

Also, when you already combine the dataset, you don't need to divide it explicitly into two fractions to compute the loss and accuracy per sample. The function `get_loss_and_accuracy` receives a parameter `reduction`. By setting it to `'none'`, it returns the loss and accuracy per sample, not the mean/sum over the mini-batch. You can then calculate these metrics per sample, separate them according to `eq_positions` (3 or 5), and do the mean on each fraction obtained.

**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 $(L, d) \in \{1, 2, 3\} \times \{2^6, 2^7, 2^8\}$ ($3 \times 3$ models). For LSTM, you must fix the hidden size to $d$ each time.

(a) (0.25 × 4 × 3 = 3pts) 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.

We visualize the training and validation loss $\mathcal{L}^{(t)}_{\text{train}}, \mathcal{L}^{(t)}_{\text{val}}$ and accuracy $\mathcal{A}^{(t)}_{\text{train}}, \mathcal{A}^{(t)}_{\text{val}}$ as functions of training step $t$. For each model (LSTM and GPT), we generate one figure per layer $L \in \{1, 2, 3\}$, showing all four metrics. Each curve represents a batch size $d \in \{64, 128, 256\}$, and colorbars reflect $d$ in $\log_2$ scale.

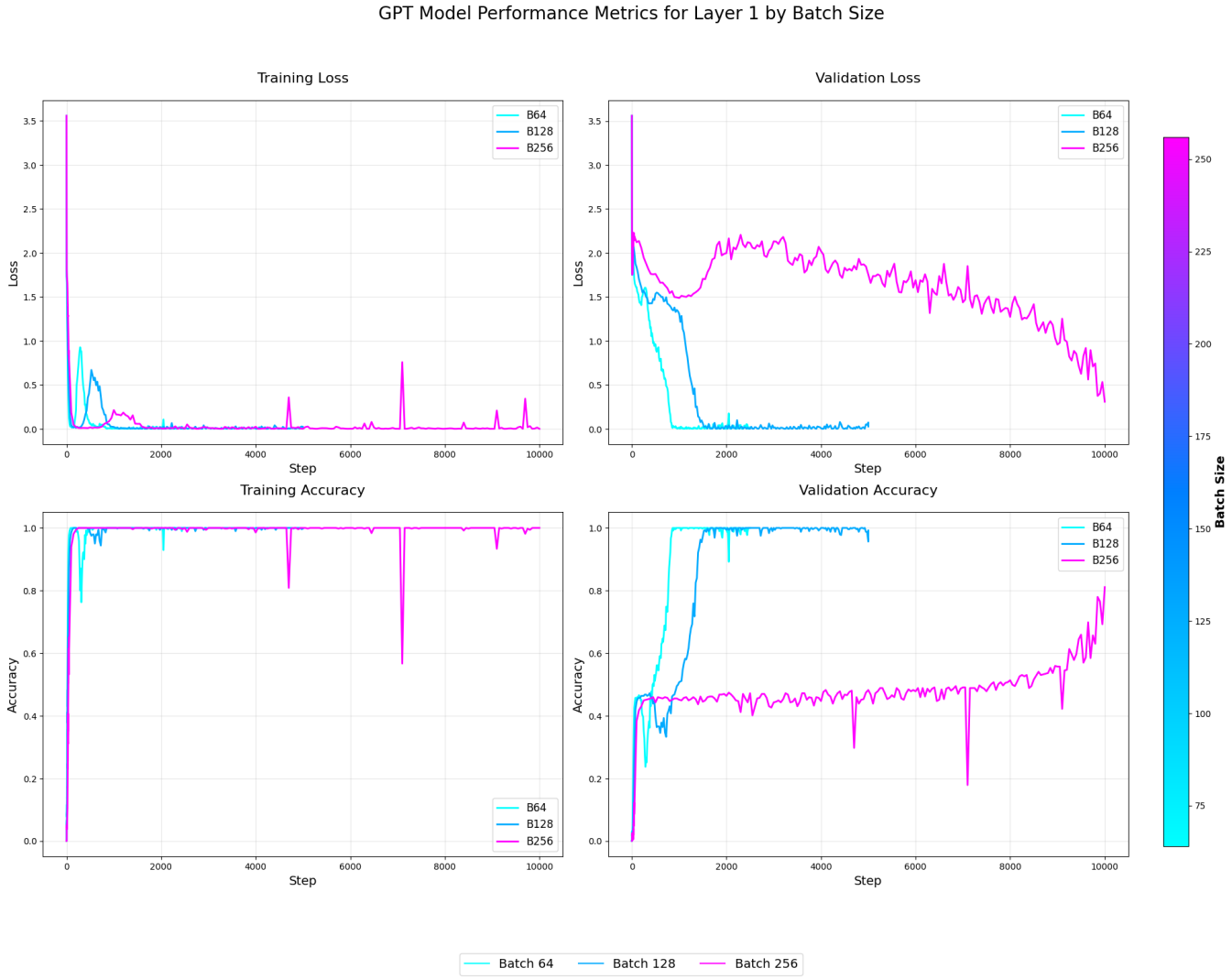



Figure 12: GPT, Layer 1: Rapid convergence for small $d$, but validation accuracy is unstable for $d = 256$.

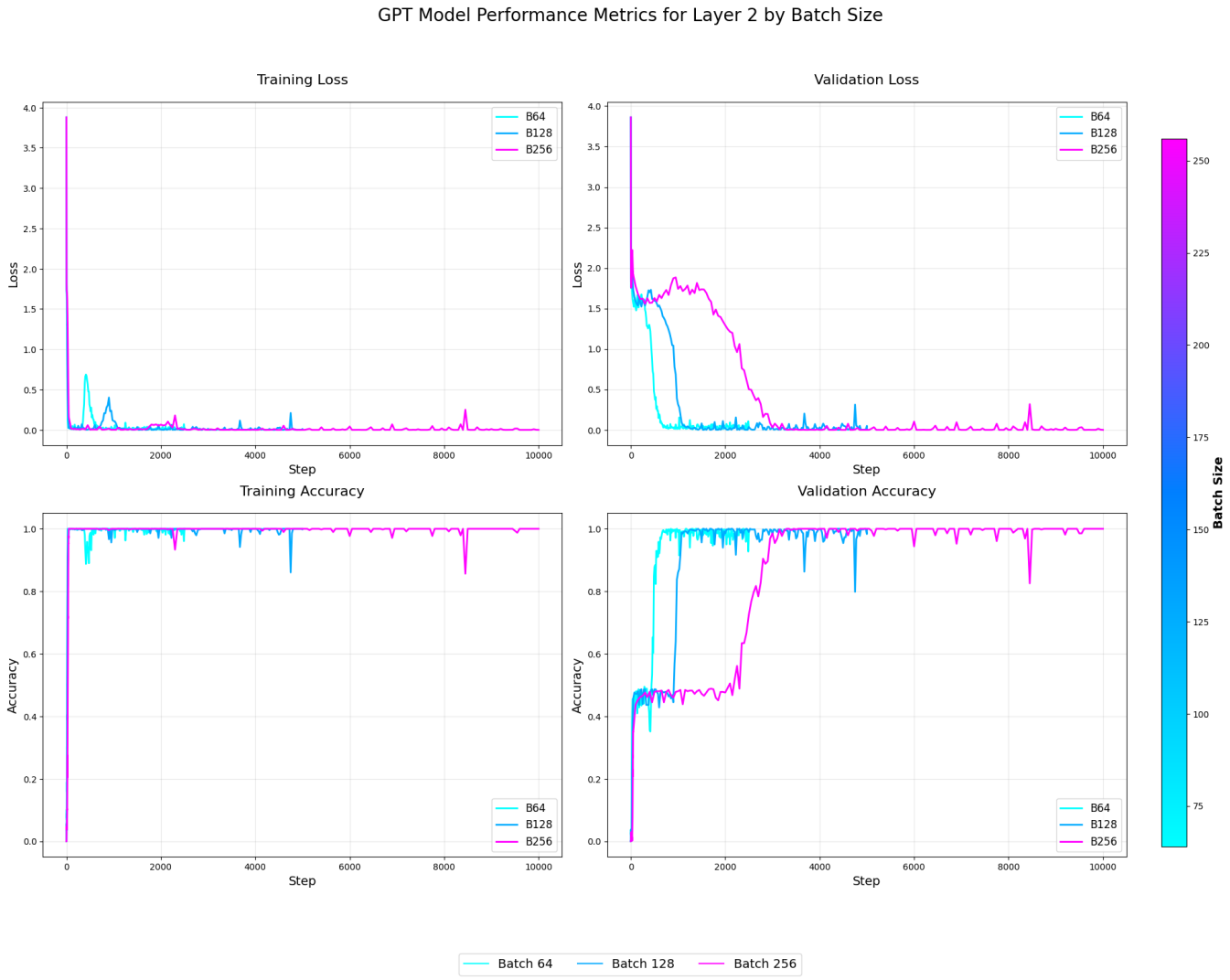



Figure 13: GPT, Layer 2: Stable convergence across all $d$. Validation loss drops smoothly; accuracy remains high.

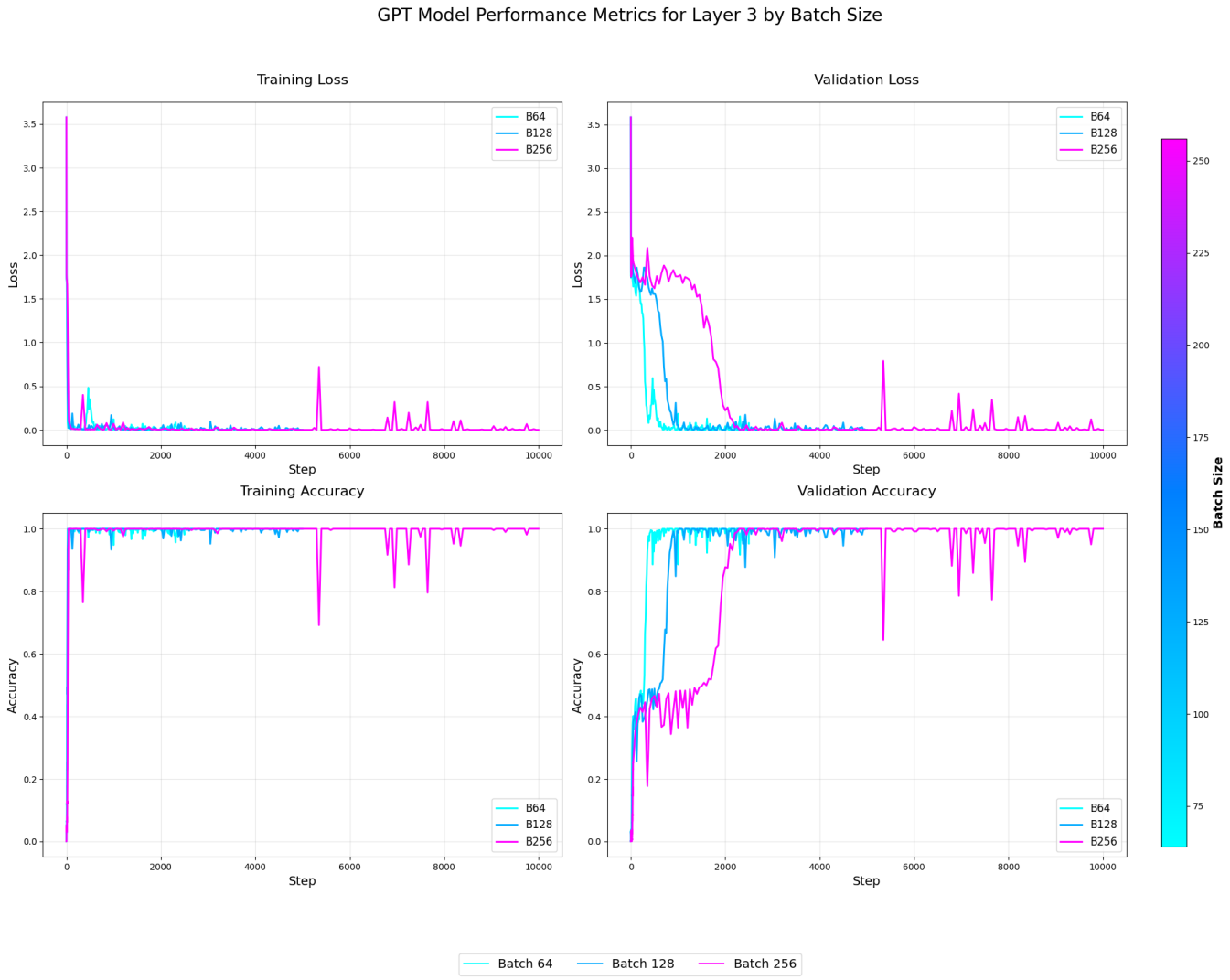



Figure 14: GPT, Layer 3: Similar trends as Layer 2. Validation accuracy saturates reliably for all batch sizes.

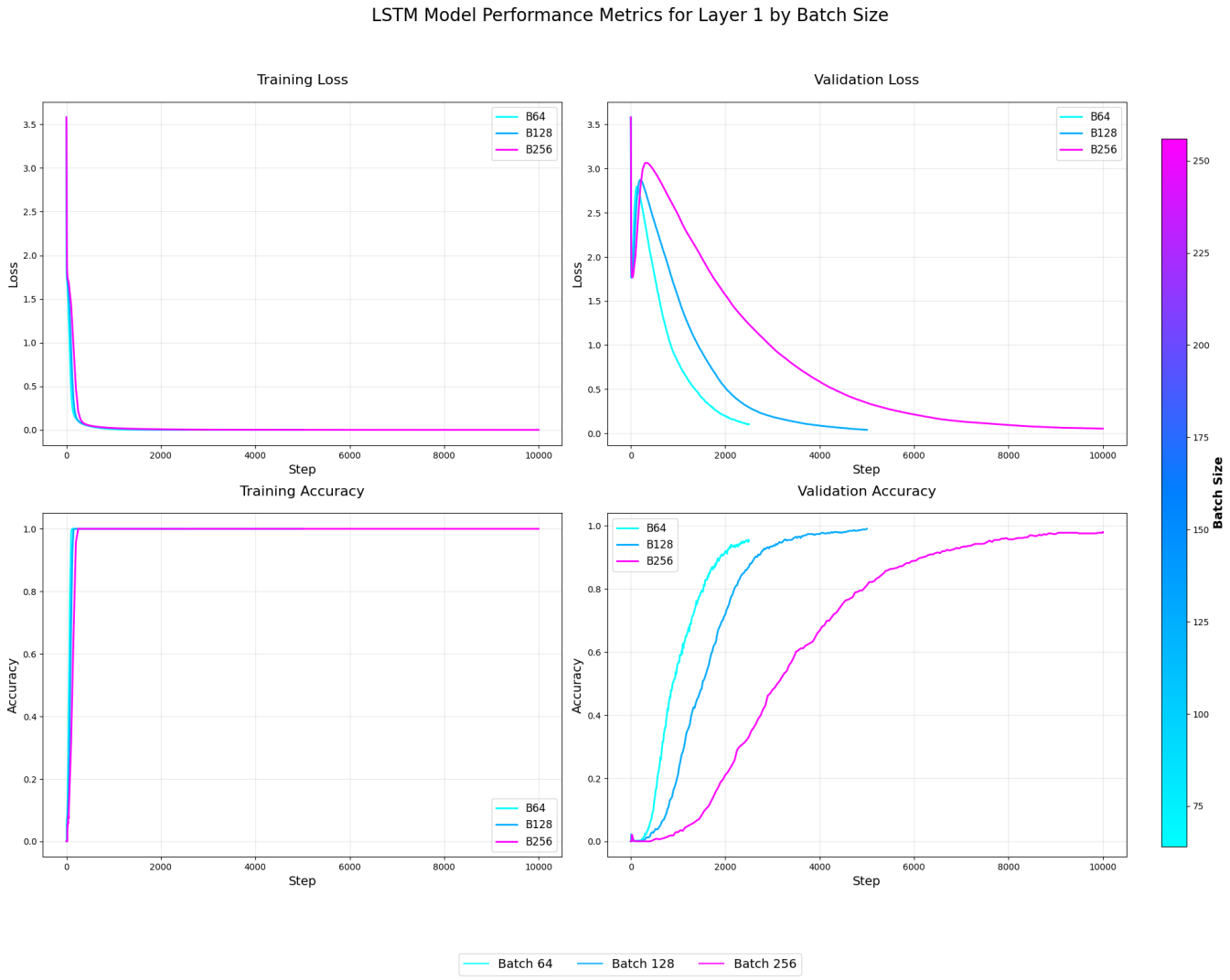



Figure 15: LSTM, Layer 1: Loss drops quickly; validation accuracy improves slowly with $d$.

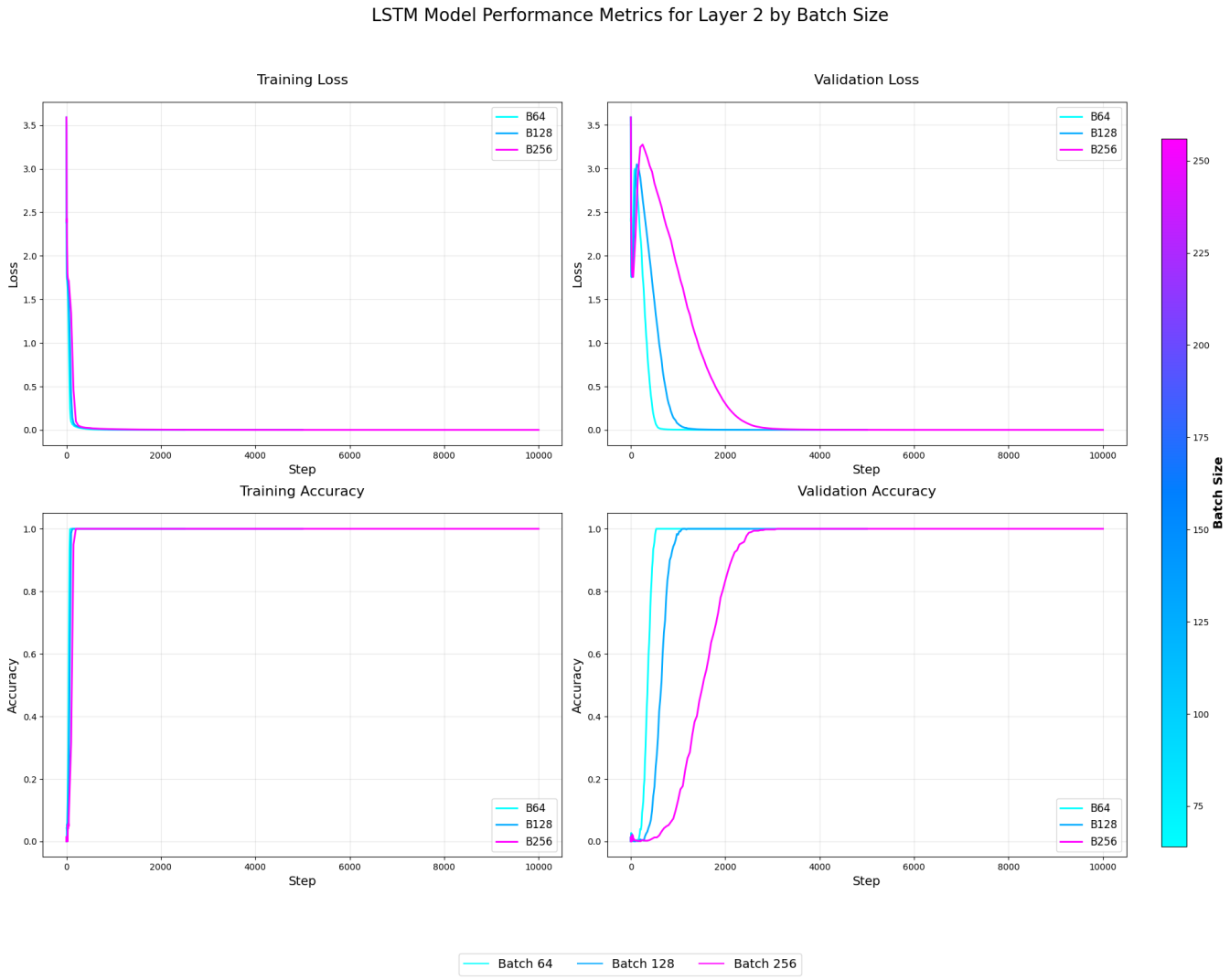



Figure 16: LSTM, Layer 2: Fast convergence and improved validation performance across all $d$.

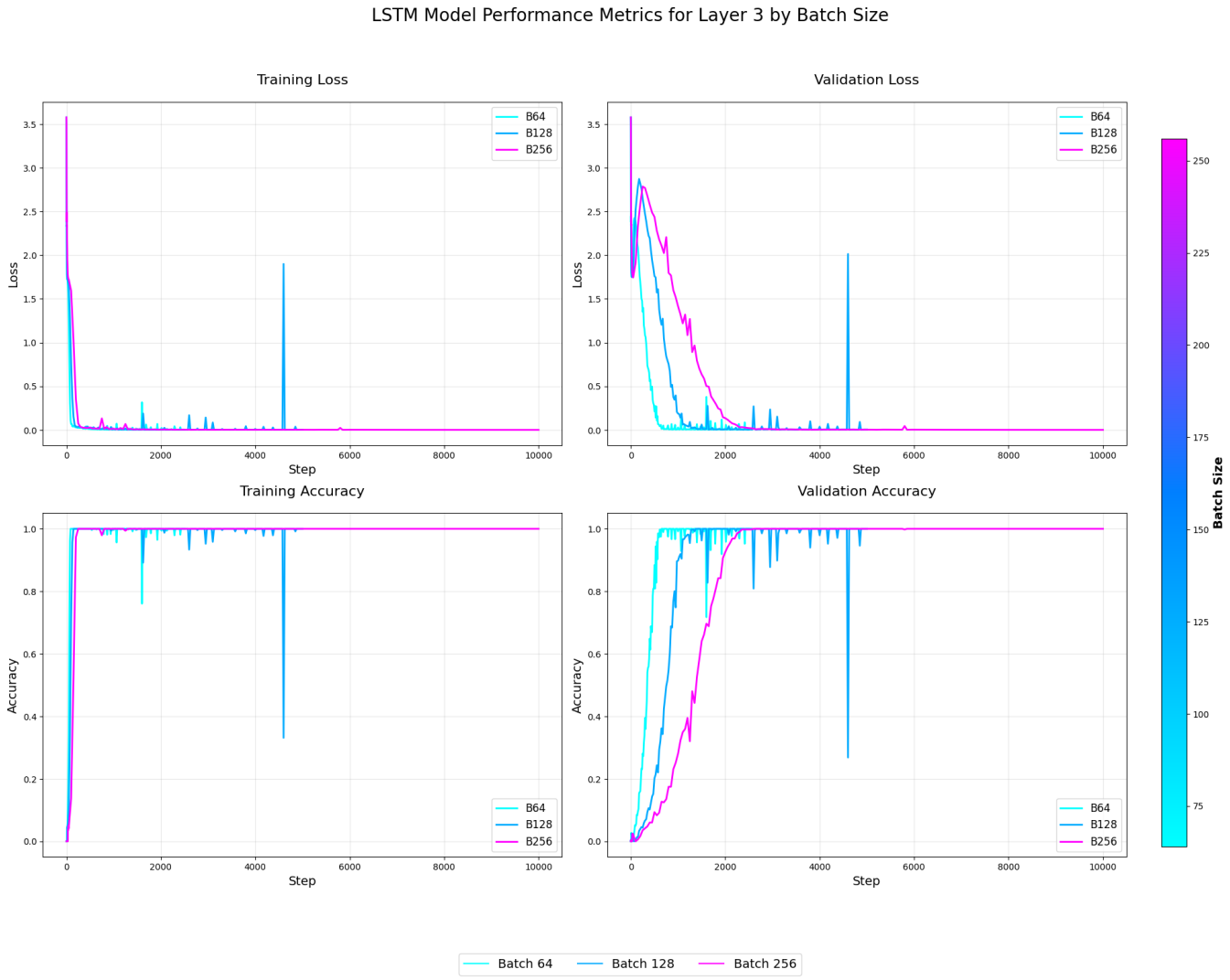



Figure 17: LSTM, Layer 3: More noise in metrics, but high accuracy is still achieved across batch sizes.

(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 $L$s, 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.

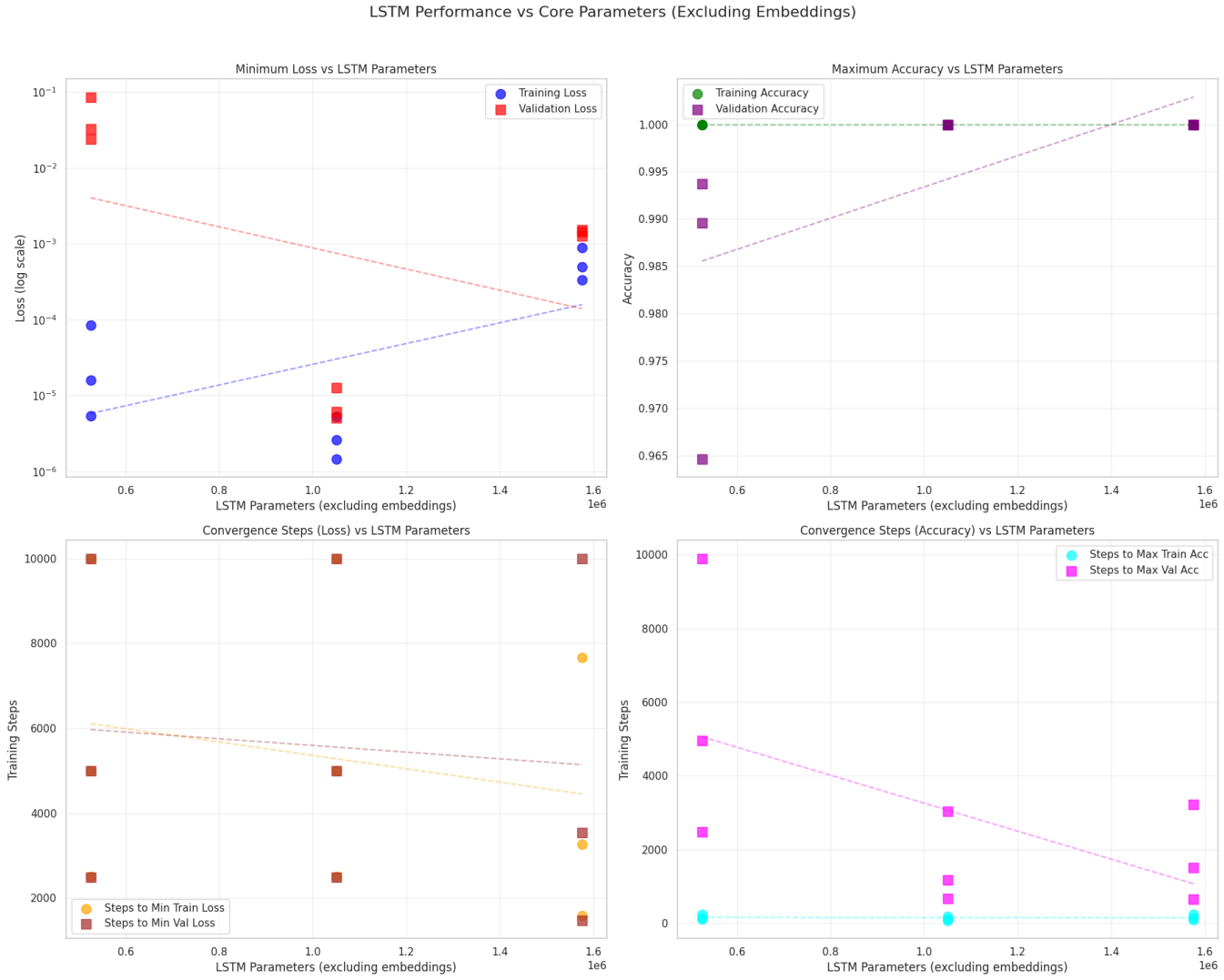



Figure 18: LSTM performance as a function of core parameters (excluding embeddings). Training and validation loss decrease with model size, while accuracy remains consistently high. Larger models converge faster, especially for validation metrics, though some variance remains across configurations.

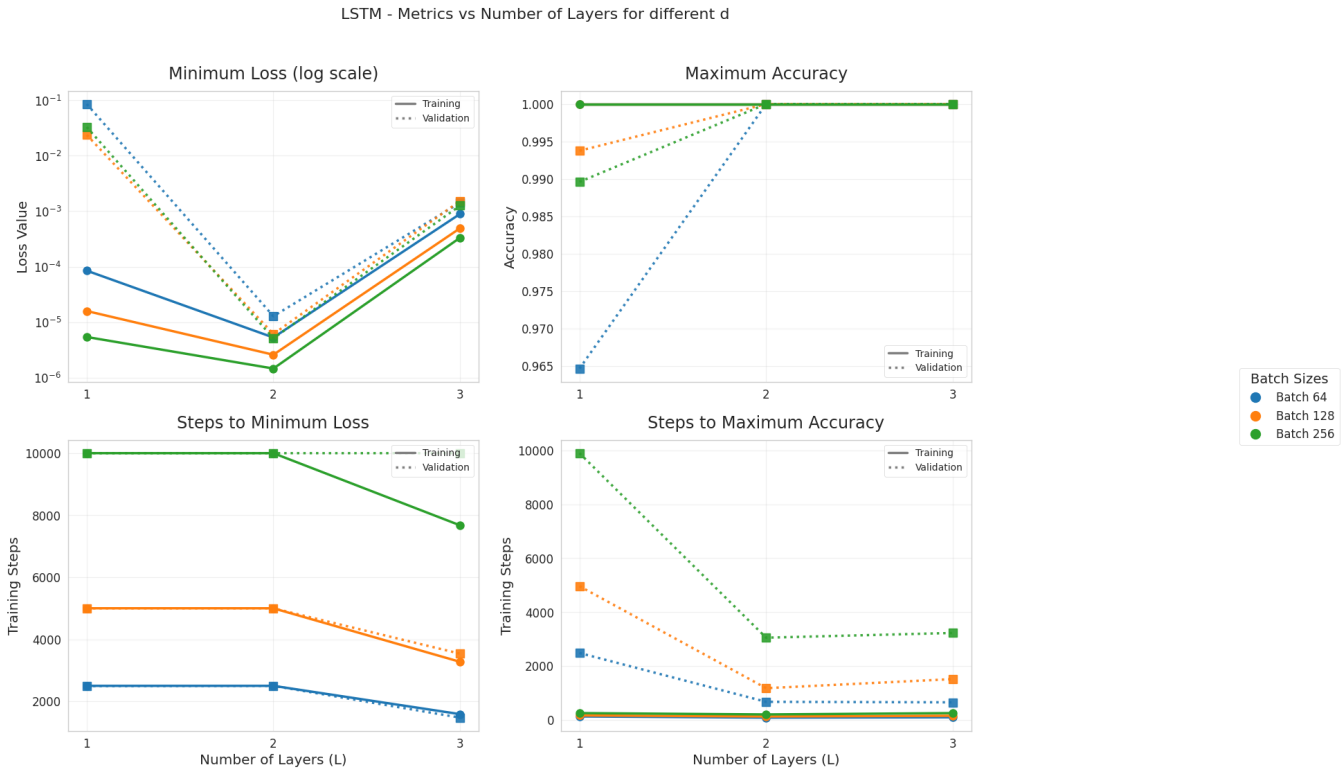



Figure 19: LSTM: Deeper models significantly reduce loss and boost accuracy. Training becomes more efficient with higher batch sizes, although validation stability varies more at lower layer counts.

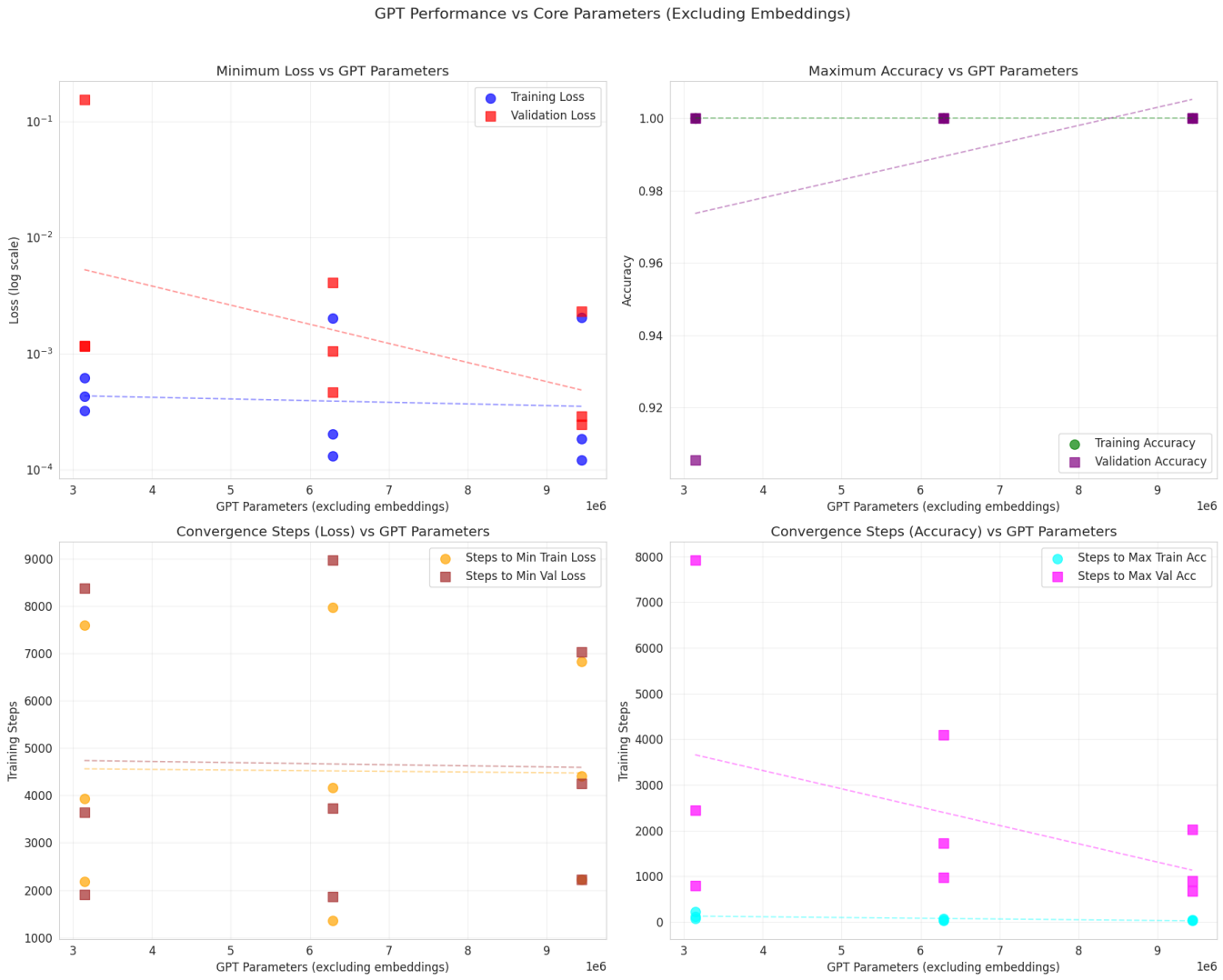



Figure 20: GPT performance as a function of core parameters (excluding embeddings). Larger models achieve lower loss and faster convergence, particularly in validation metrics. Accuracy improves with parameter count, and validation metrics show higher variance compared to training metrics.

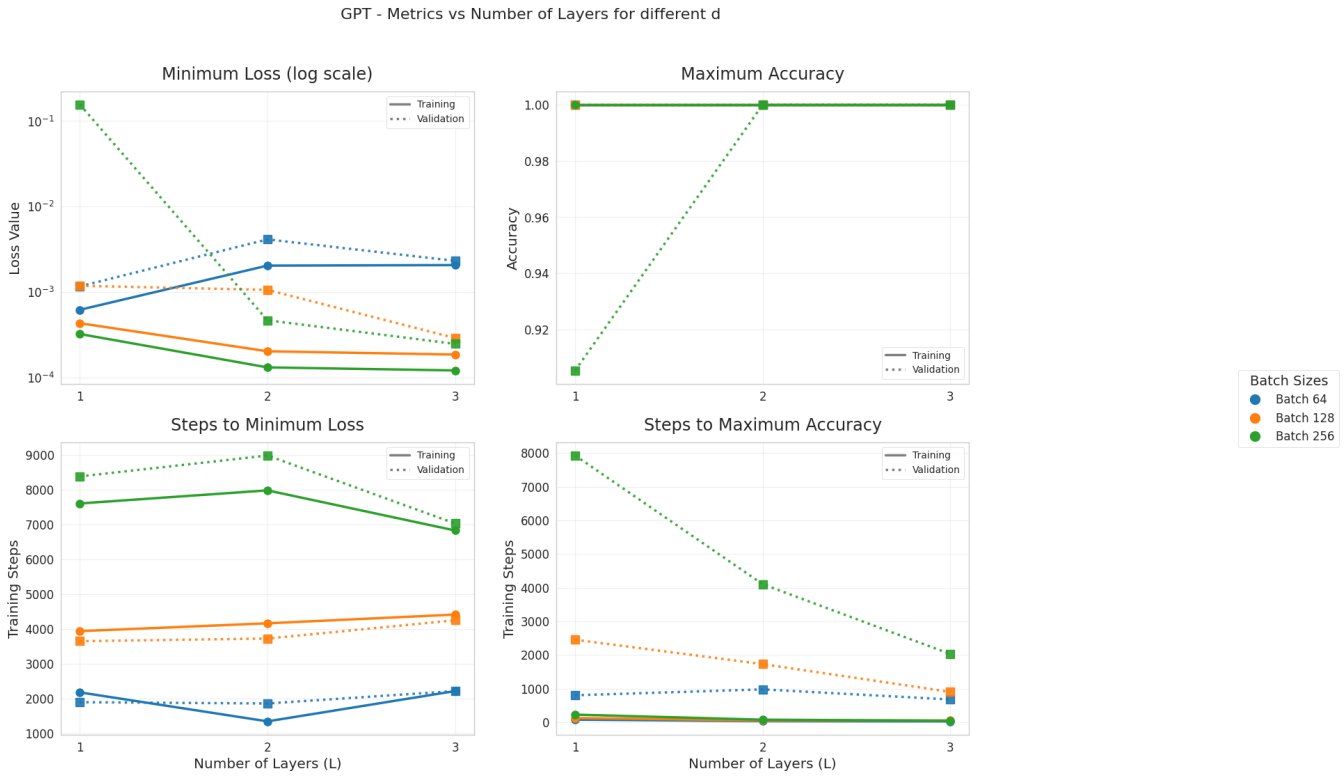



Figure 21: GPT: As the number of layers increases, both training and validation loss improve, with faster convergence in larger models. Accuracy reaches 100% across configurations, and higher batch sizes lead to more stable and faster optimization.

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

- **Scaling with $d$ (width) for fixed $L$:**
  - **GPT** scales better with increasing $d$, especially for validation accuracy ($\mathcal{A}_{\text{val}}$).
  - Larger $d$ leads to smoother and more stable generalization in GPT.
  - **LSTM** shows less consistent improvement and can overfit or become unstable as $d$ increases.
  - For validation loss ($\mathcal{L}_{\text{val}}$), GPT shows more reliable improvement with larger $d$ compared to LSTM.
- **Scaling with $L$ (depth) for fixed $d$:**
  - **GPT** benefits significantly from increased depth; deeper layers improve validation accuracy.
  - At fixed $d$, GPT leverages additional layers effectively to learn better representations.
  - **LSTM** struggles with deeper models; training becomes less stable and validation performance plateaus or worsens.
- **Scaling with number of parameters ($P$):**
  - **GPT** shows smoother gains in both $\mathcal{L}_{\text{val}}$ and $\mathcal{A}_{\text{val}}$ as $P$ increases.
  - **LSTM** gains from larger $P$ initially, but saturates earlier and is more sensitive to architecture.
  - Overall, GPT scales more efficiently with model size compared to LSTM.

**Scaling compute (5pts)**: We estimate the total non-embedding training compute by $\mathtt{C} \simeq \mathtt{P} \times \mathtt{B} \times \mathtt{T}$, where $\mathtt{B}$ is the batch size and $\mathtt{T}$ is the number of training steps. In practice, for a given budget $\mathtt{C}$, the aim is to find the triplet $(\mathtt{P}, \mathtt{B}, \mathtt{T})$ that gives the best generalization performance. For example, you may see the session time on Google Colab with computational resources as a budget $\mathtt{C}$, and your goal is to find the best model given this budget. If you have a law that for any triplet $(\mathtt{P}, \mathtt{B}, \mathtt{T})$ gives the generalization performance, you can, for a given $\mathtt{C}$, draw the level curve $\mathtt{P} \times \mathtt{B} \times \mathtt{T} = \mathtt{C}$, and find the point $(\mathtt{P}, \mathtt{B}, \mathtt{T})$ on that curve that gives you the best performance[4]. To avoid doing a lot of experiments, you'll focus on $(\mathtt{B}, \mathtt{T})$ and do something much more straightforward, keeping $\mathtt{P}$ fixed (use default parameters $(L, d) = (2, 2^7)$).

6. (5pts) Train the models for $\mathtt{B} \in \{2^5, 2^6, 2^7, 2^8, 2^9\}$ and $\mathtt{T} = 2 \times 10^4 + 1$. Now, for each $\mathtt{B}$ and for each $\mathtt{T}' \in \{\alpha\mathtt{T}, \alpha \in \{0.1, 0.2, \ldots, 0.9, 1.0\}\}$ 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 $\mathtt{T}' = \alpha\mathtt{T}$ steps (budget $\mathtt{C}' \simeq \alpha\mathtt{C}$), and you need to find the smallest loss and the largest accuracy under these $\mathtt{T}'$ 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`).

[4] If you're interested in reading about scaling laws for language models, have a look at the paper Scaling Laws for Neural Language Models. You can also check this one to have more understanding on the type of training and scaling curve you obtained: Emergent Abilities of Large Language Models.

(a) (0.25 × 4 = 1pts) 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 `B`). Also, always put the axis/colorbar corresponding to `B` in $\log_2$ scale.

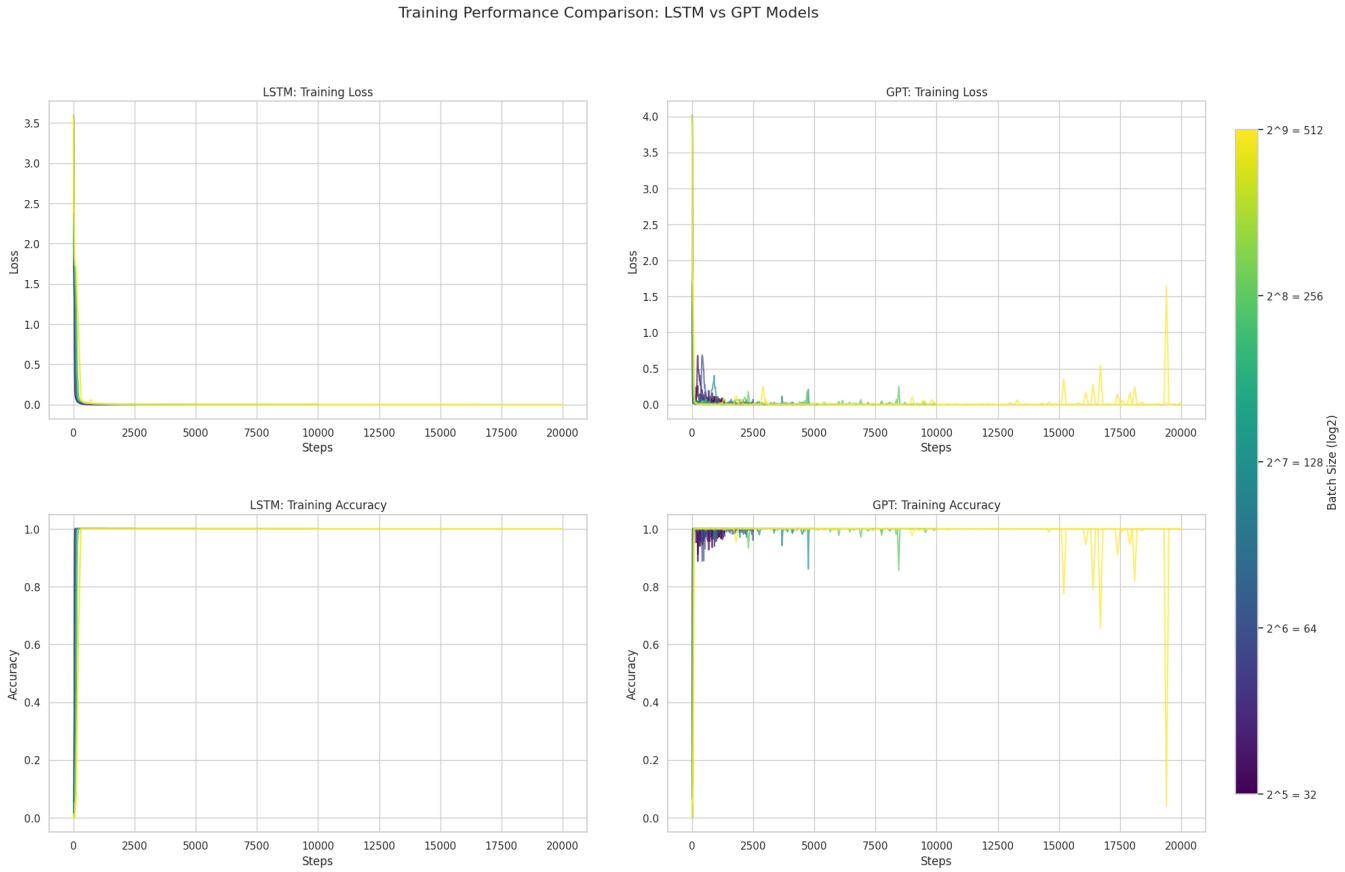



Figure 22: Training performance of LSTM and GPT models across various batch sizes. Loss and accuracy are shown over training steps, with color indicating $\log_2$(batch size). LSTMs converge quickly and stably, whereas GPTs exhibit more fluctuation with larger batch sizes.

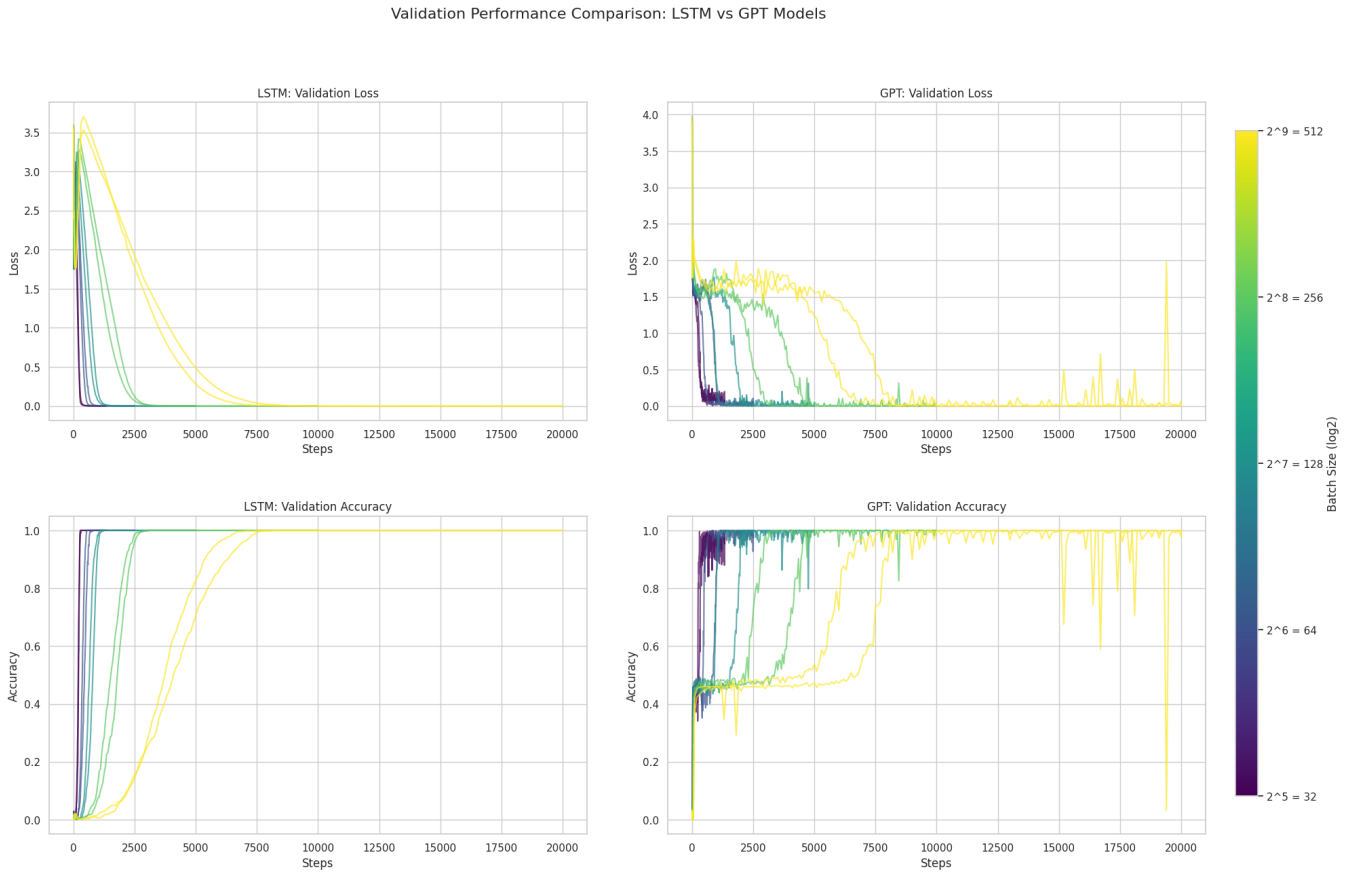



Figure 23: Validation performance of LSTM and GPT models across various batch sizes. Loss and accuracy are shown over validation steps. LSTM models generalize well early, while GPTs require more steps for stability, especially at larger batch sizes.

(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 $\alpha$). You can also consider LSTM and GPT on the same curve for comparison purposes (or side-by-side curves).

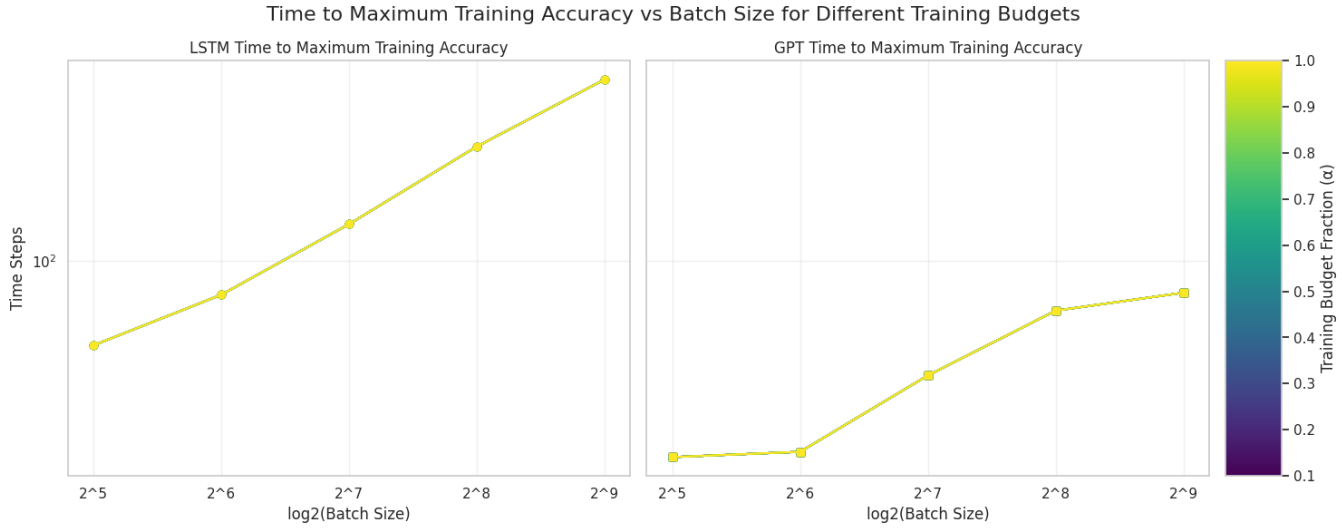



Figure 24: Time steps to reach maximum training accuracy for LSTM and GPT models, plotted against batch size for different training budget fractions $\alpha$. LSTMs require significantly more steps as batch size increases, while GPTs remain relatively stable for smaller batches.

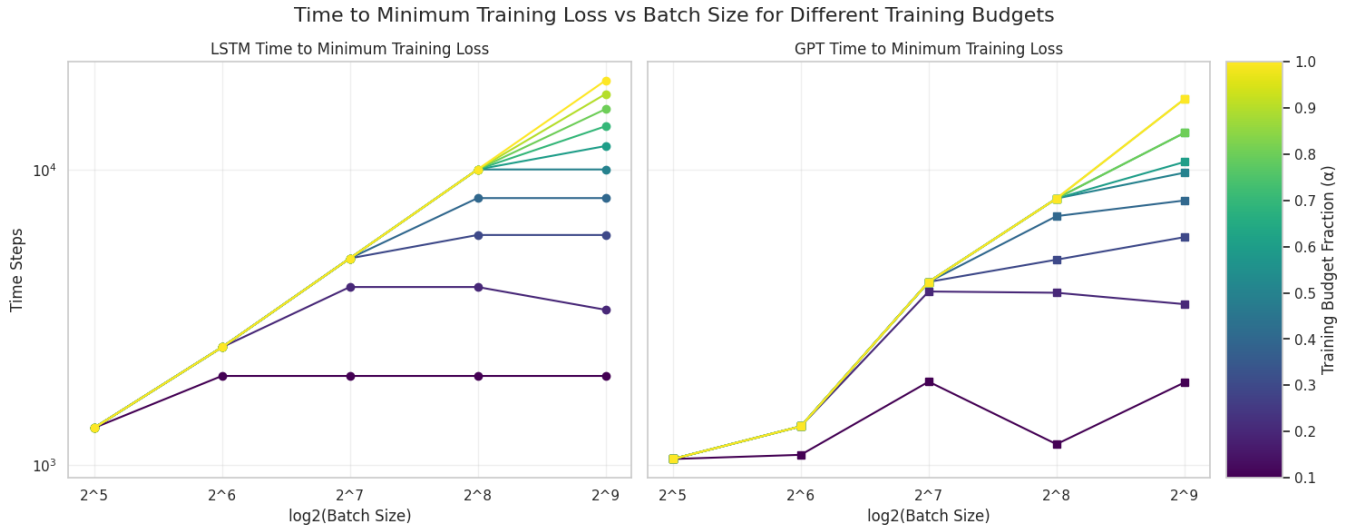



Figure 25: Time to reach minimum training loss for LSTM and GPT models across batch sizes and training budgets. LSTMs benefit from smaller batches under low budget regimes, whereas GPTs show more variance with batch size.

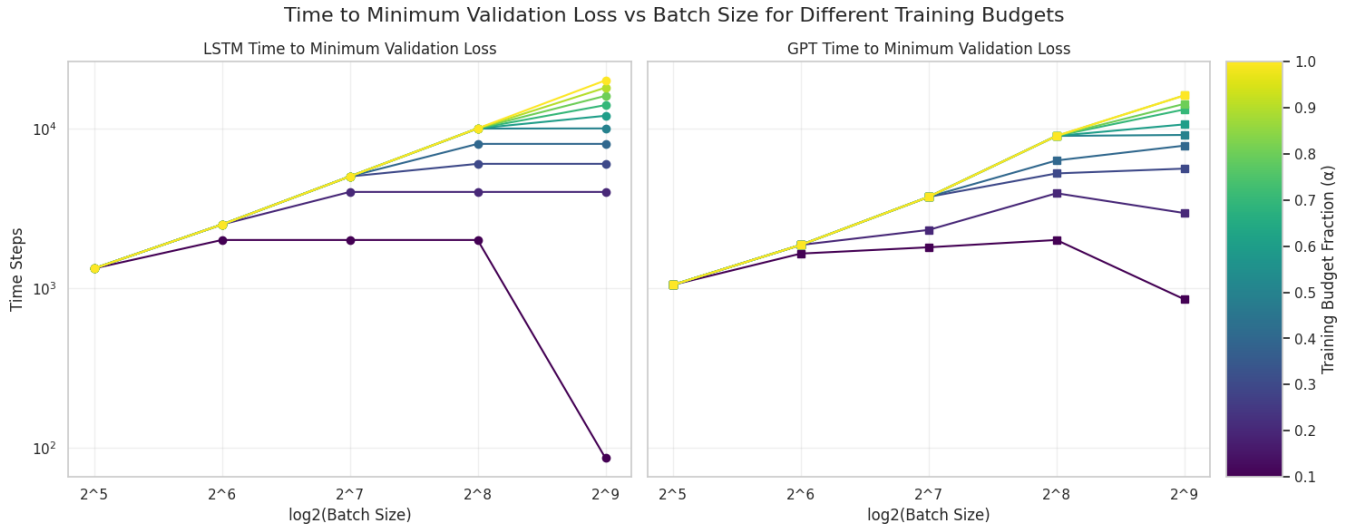



Figure 26: Time steps required to reach minimum validation loss across batch sizes and training budgets. LSTM curves exhibit a more predictable scaling with batch size, while GPT performance varies more under lower budget fractions.

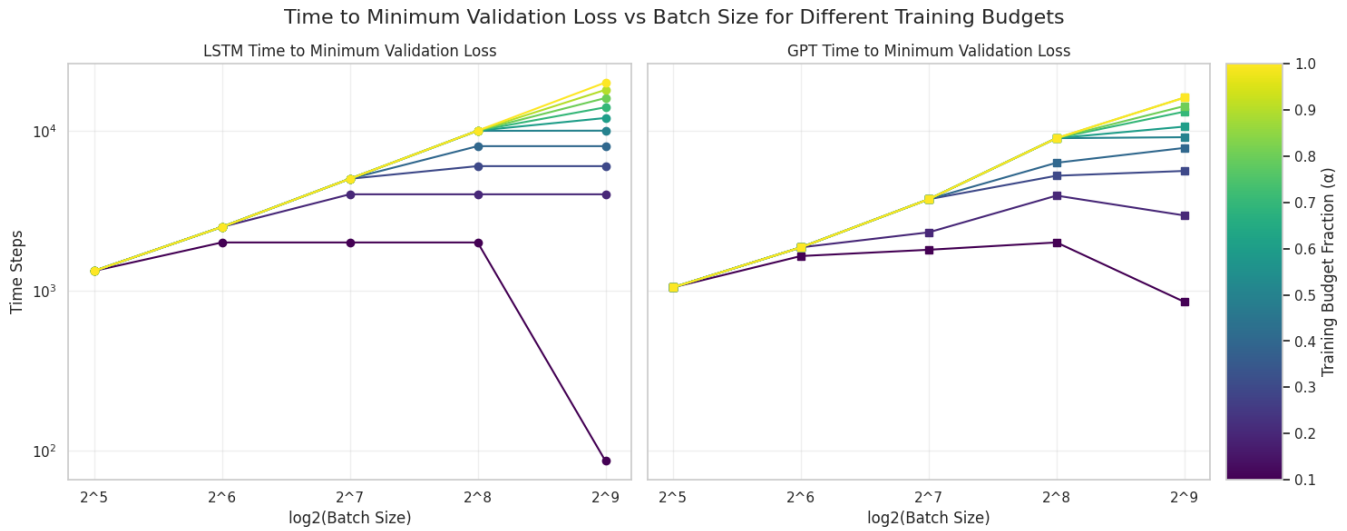



Figure 27: Time steps required to reach minimum validation loss across batch sizes and training budgets. LSTM curves exhibit a more predictable scaling with batch size, while GPT performance varies more under lower budget fractions.

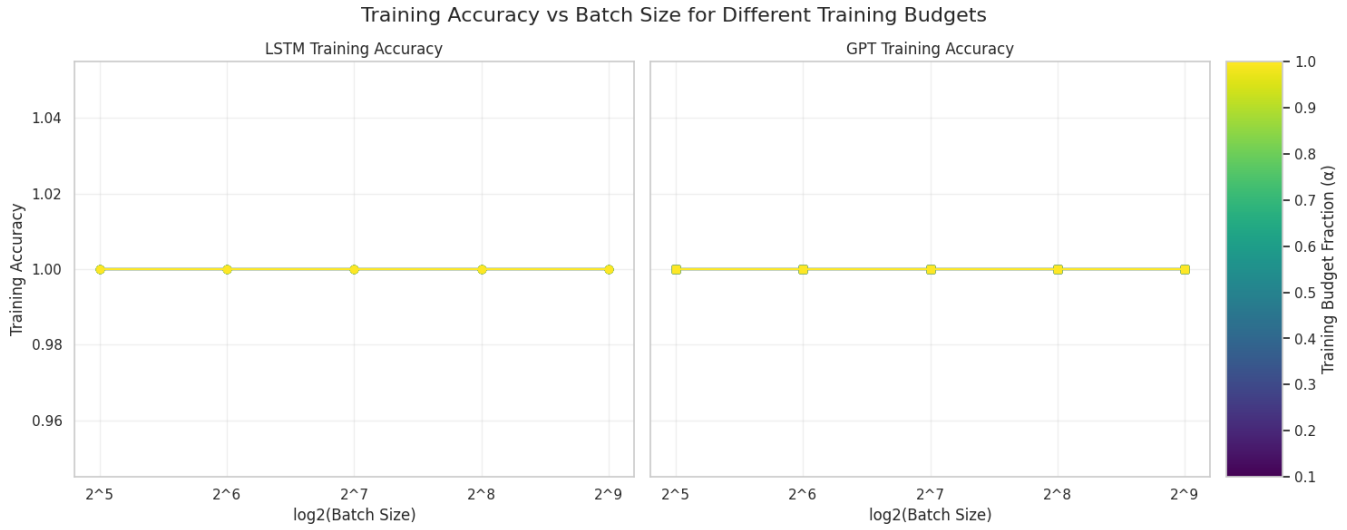



Figure 28: Final training accuracy vs. batch size for LSTM and GPT across different training budget fractions $\alpha$. Both models consistently reach perfect training accuracy regardless of batch size.

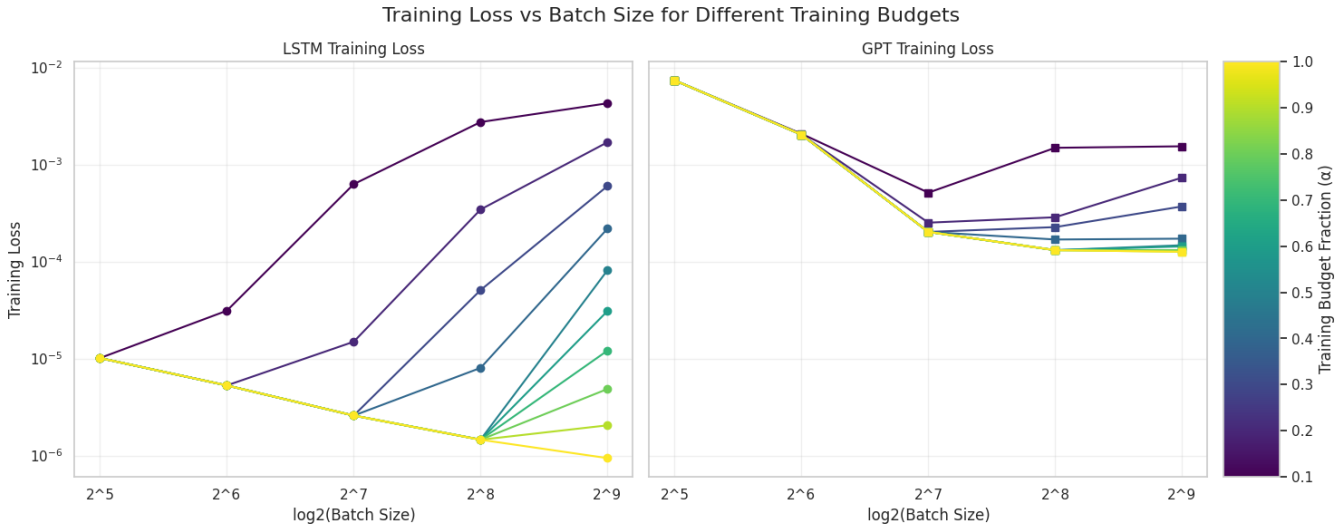



Figure 29: Training loss as a function of batch size under varying training budgets. LSTM models show increased loss with larger batches under constrained budgets. GPT models show greater resilience across the batch size spectrum.

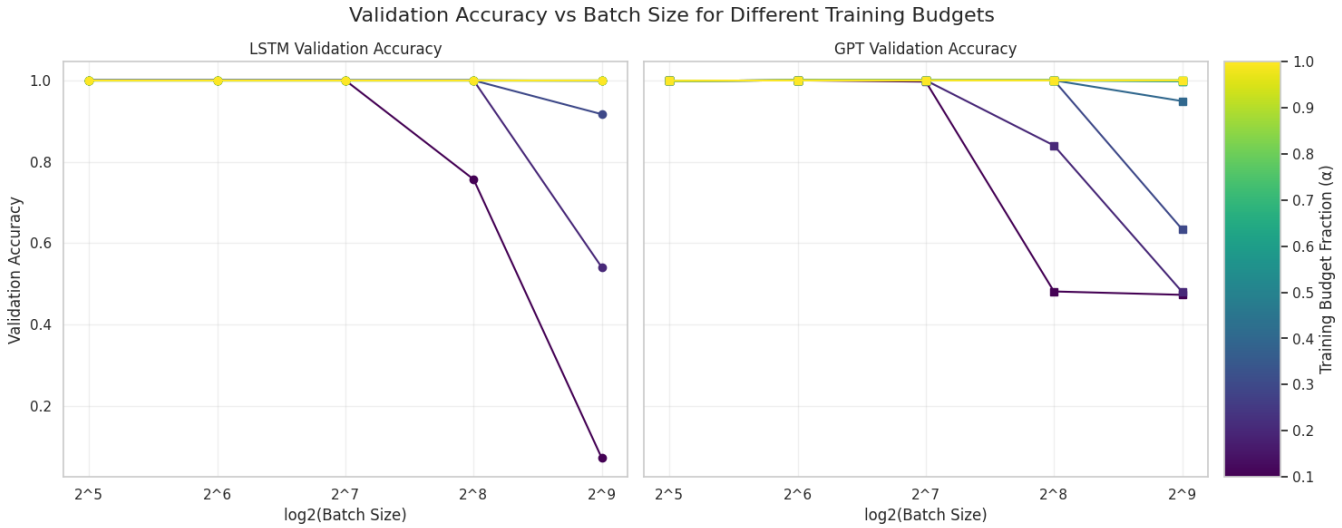



Figure 30: Final validation accuracy vs. batch size for LSTM and GPT under different training budgets. LSTM performance drops with larger batch sizes at lower budgets. GPT maintains higher accuracy across a wider range.

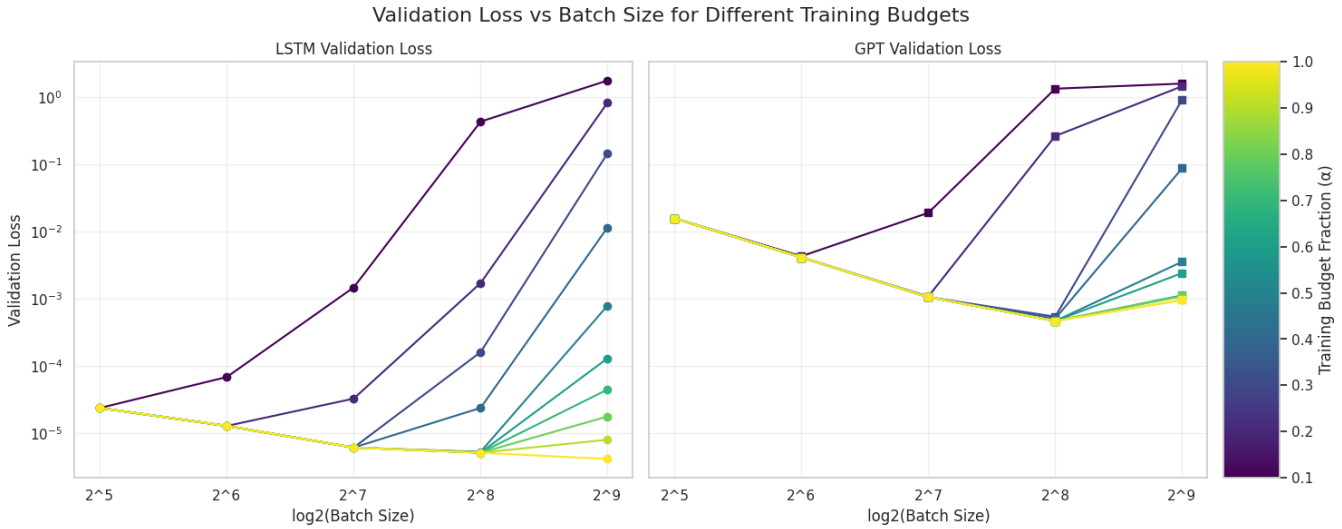



Figure 31: Validation loss across batch sizes and training budgets. LSTM loss increases more steeply with batch size under budget constraints, while GPT is less sensitive to this variation.

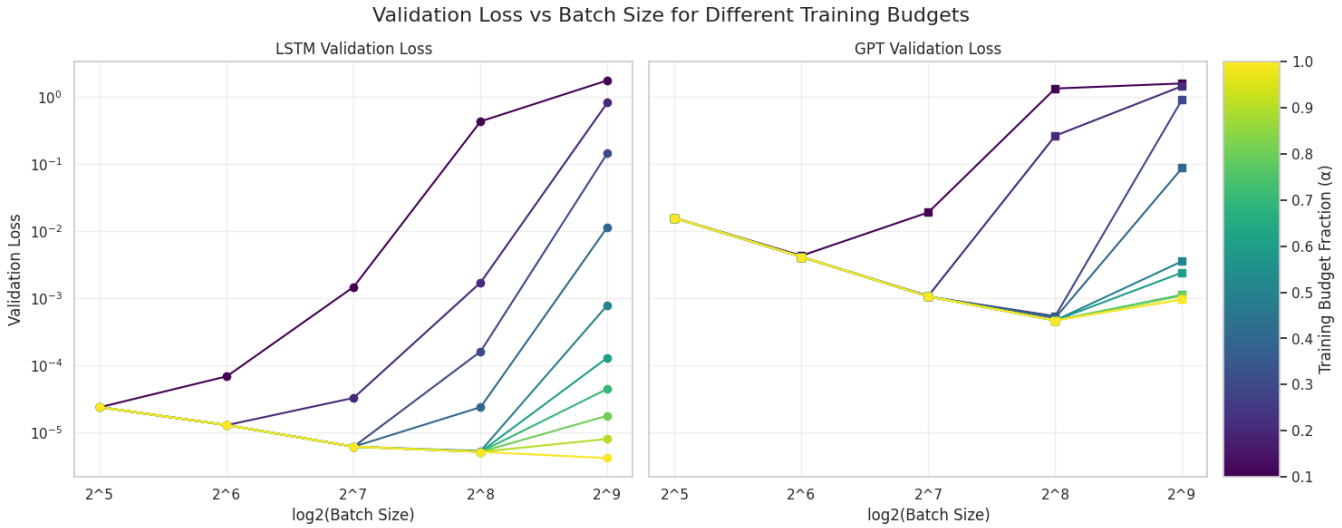



Figure 32: Validation loss across batch sizes and training budgets. LSTM loss increases more steeply with batch size under budget constraints, while GPT is less sensitive to this variation.

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

- Increasing batch size (`B`) leads to higher validation loss and lower validation accuracy, especially when the training budget fraction ($\alpha$) is small.
- Increasing the training budget fraction ($\alpha$) improves generalization performance across all batch sizes, reducing validation loss and increasing validation accuracy.
- Validation loss ($\mathcal{L}_{\text{val}}$) is lowest when training with small batch sizes and large $\alpha$. At low $\alpha$, larger batch sizes result in significantly higher $\mathcal{L}_{\text{val}}$ for both LSTM and GPT models.
- Validation accuracy ($\mathcal{A}_{\text{val}}$) remains high across all batch sizes when $\alpha$ is large. Under low $\alpha$, accuracy drops sharply with increasing `B`, with LSTM models being more affected than GPT.

**Regularization (6.5pts)**:

7. (6.5pts) With `T` $= 4 \times 10^4 + 1$, train an LSTM with `weight_decay` in $\{1/4, 1/2, 3/4, 1\}$, and also track the $\ell_2$ 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`).

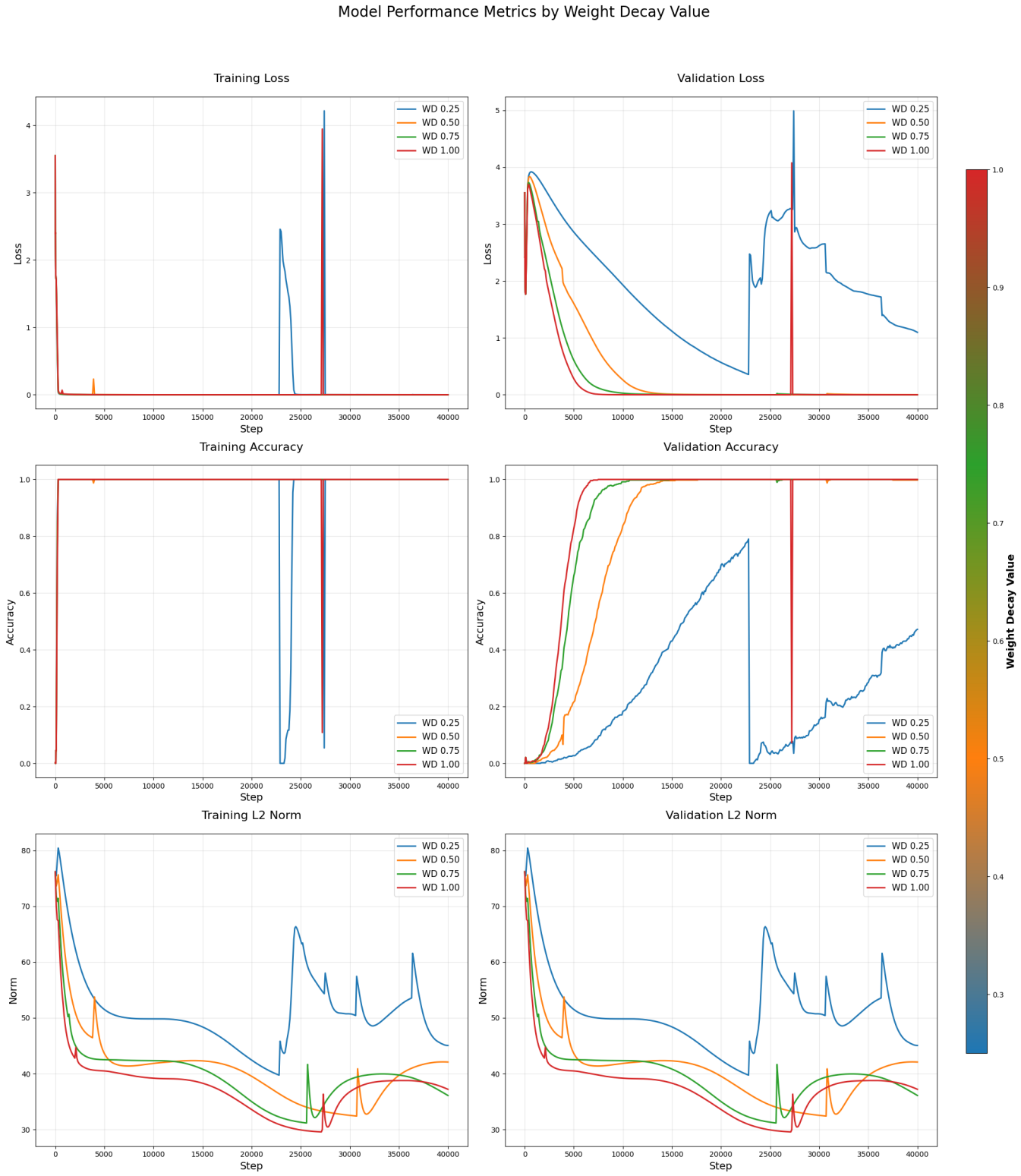



Figure 33: Training and validation metrics (loss, accuracy, L2 norm) for various weight decay values over training steps. Models with higher weight decay (e.g., 0.75, 1.0) converge faster and generalize better, while lower decay (0.25) shows overfitting and instability.

(b) (0.25 × 8 = 2pts) 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).

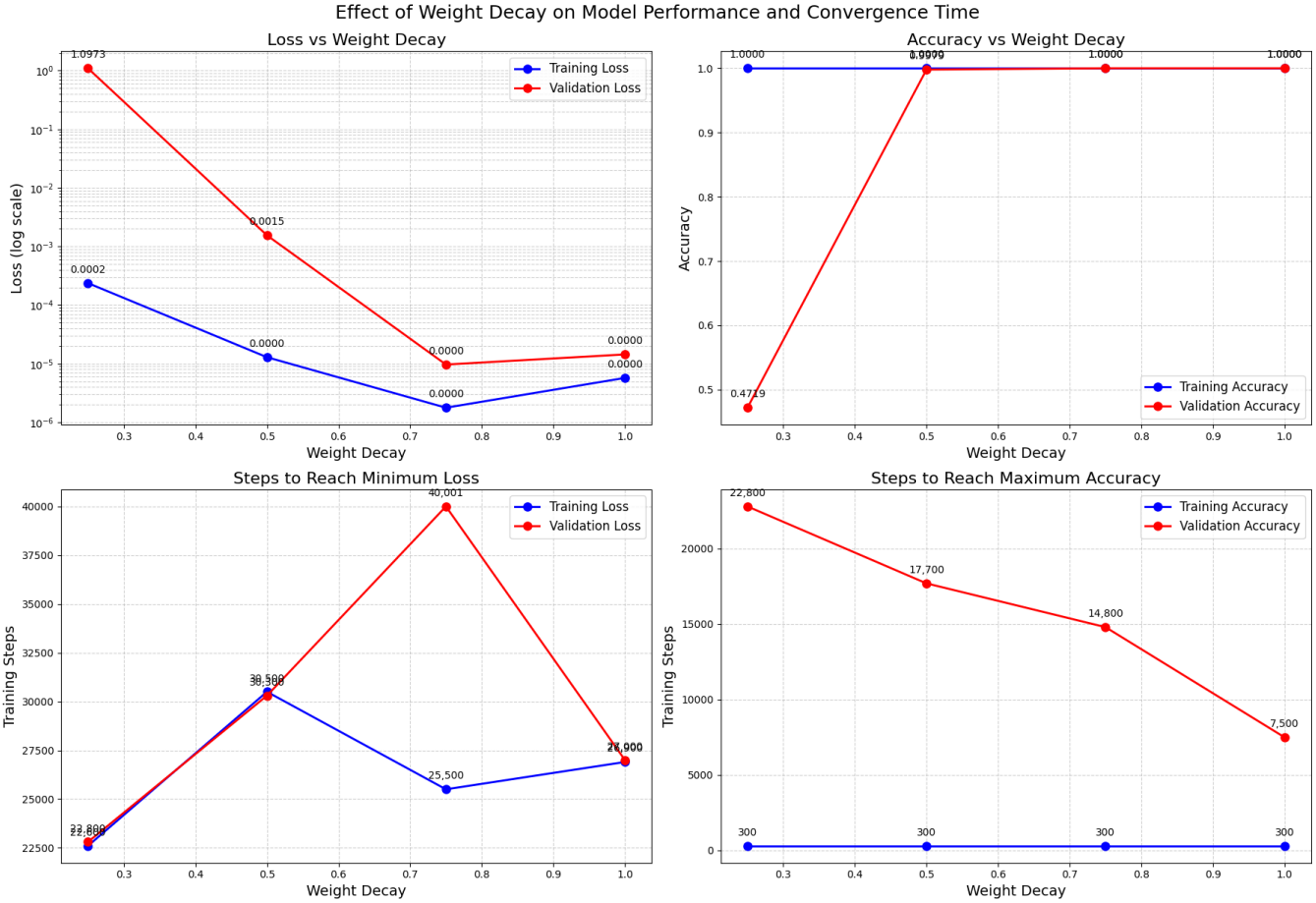



Figure 34: Effect of weight decay on loss, accuracy, and convergence. Loss decreases and accuracy improves with higher weight decay. Moderate decay speeds up generalization while maintaining performance.

(c) $(0.25 \times 4 = 1\text{pt})$ 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`?

- $\mathcal{L}_{\text{val}}$ decreases sharply as weight decay increases up to 0.7, then slightly increases beyond that point.
- $\mathcal{A}_{\text{val}}$ improves with increasing weight decay, reaching near-perfect accuracy from 0.5 onward.
- $\texttt{Steps}_{\mathcal{L}_{\text{val}}}$ peaks at weight decay 0.7, indicating delayed convergence to minimum validation loss at this setting.
- $\texttt{Steps}_{\mathcal{A}_{\text{val}}}$ decreases steadily with higher weight decay, reaching fastest convergence at decay = 1.0.

(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?

- For low `weight_decay` (e.g., 0.25), $\|\theta^{(t)}\|_2$ stays high and fluctuates. Generalization is slower or fails.
- For higher values (0.5–1.0), $\|\theta^{(t)}\|_2$ drops quickly and stabilizes. Generalization happens earlier.
- This happens because weight decay penalizes large weights, shrinking $\|\theta^{(t)}\|_2$ and helping the model avoid overfitting, which improves generalization.

**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)`.

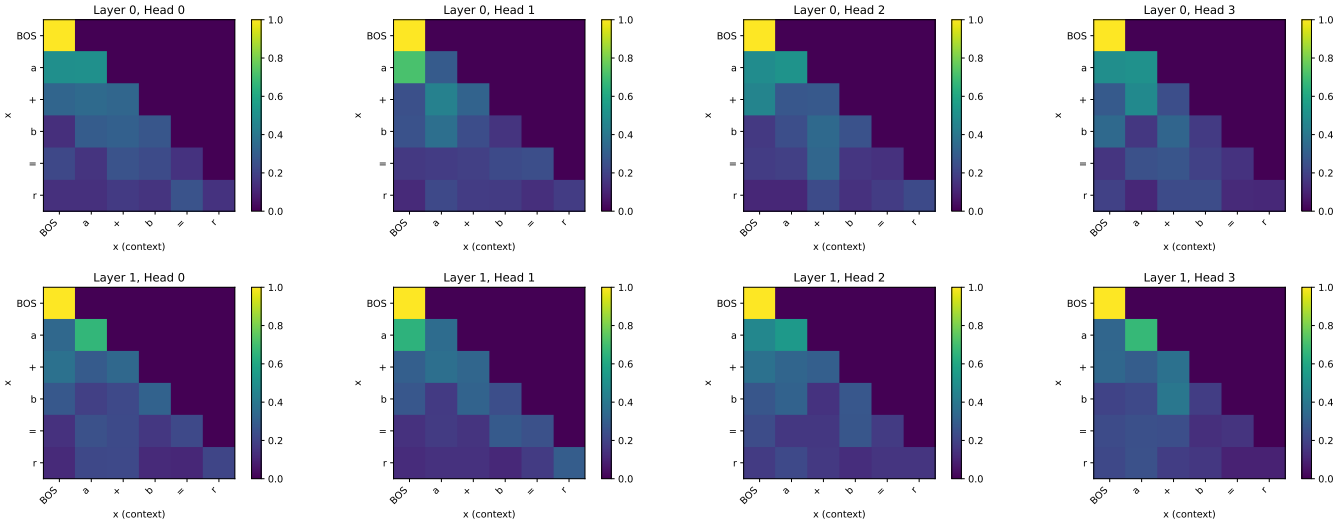



Figure 35: Attention weights of `GPT` for the input sequence "BOS a + b = r"

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 × 8 × 2 = 4pts) 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 35 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.

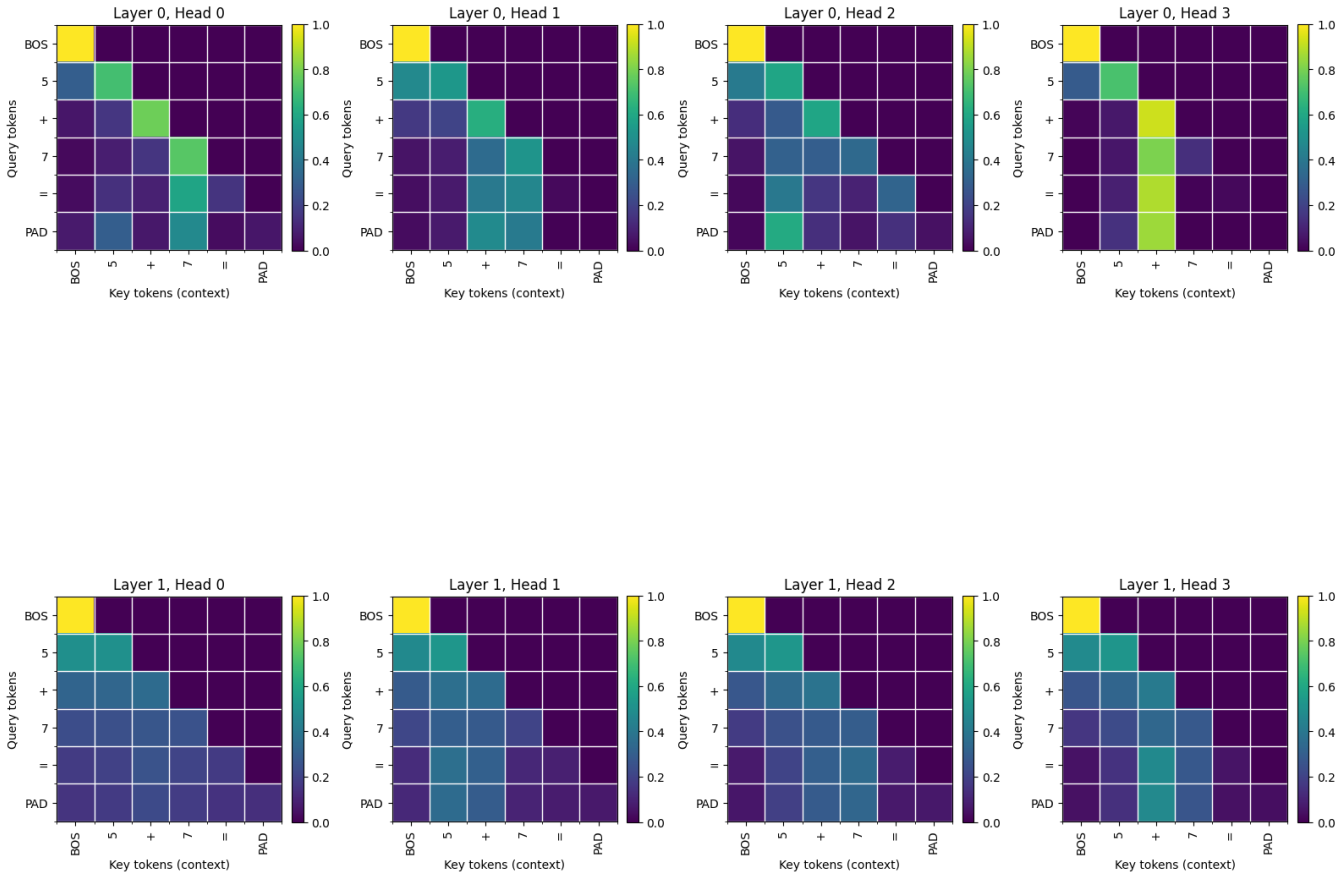



Figure 36: Attention maps for the input `5 + 7 =`. Each square shows how much one token pays attention to another. Brighter colors mean stronger attention. Early heads focus mostly on the start token (`BOS`) and nearby numbers. Deeper heads spread attention more evenly across the sequence.

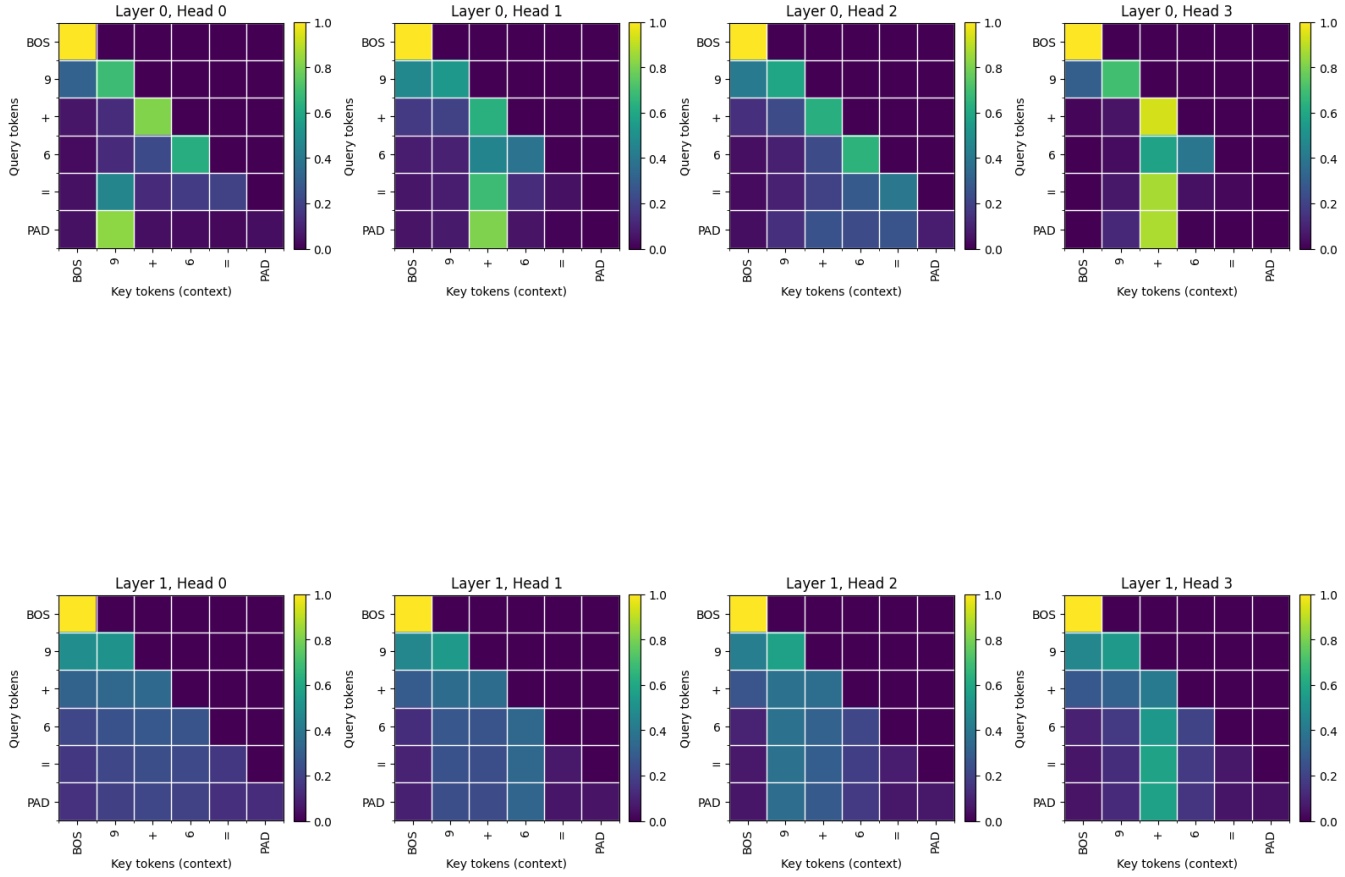



Figure 37: Attention maps for the input `9 + 6 =`. Brighter colors show stronger attention between tokens. Heads in the first layer look mostly at the beginning of the sequence and recent tokens. Heads in the second layer show more balanced attention across the input.

(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.

**Answer:** Many heads in Layer 0 focus most attention on the `BOS` token, especially for early query positions. This behavior is strongest in Layer 0 across all heads, but becomes less dominant in Layer 1, where attention is more spread out across earlier tokens. This likely reflects the model's use of `BOS` as an anchor or global reference during early processing, while deeper layers begin integrating more contextual information.