# ECE 405, Spring 2026
# CS336 Assignment 1 Basics

Kanta Saito

## Problem (unicode1): Understanding Unicode (1 point)

(a) What Unicode character does `chr(0)` return?

**Deliverable:** A one-sentence response.

`chr(0)` returns a null character, specifically U+0000.

(b) How does this character's string representation (`__repr__()`) differ from its printed representation?

**Deliverable:** A one-sentence response.

`__repr__()` shows exactly what the null character is (using hex code) so you can identify it, whereas `print()` outputs nothing as the character is invisible.

(c) What happens when this character occurs in text?

When the null character occurs in text, it is typically invisible when printed, appearing like the surrounding strings are concatenated together even though the character remains present in the string's data.

## Problem (unicode2): Unicode Encodings (3 points)

(a) What are some reasons to prefer training our tokenizer on UTF-8 encoded bytes, rather than UTF-16 or UTF-32?

One reason to prefer training our tokenizer on UTF-8 encoded bytes is space efficiency; processing less data leads to faster training compared to UTF-16 and UTF-32. Another reason is compatibility, as most data processing libraries utilize UTF-8 by default.

(b) Why is the function `decode_utf8_bytes_to_str_wrong` incorrect? Provide an example.

**Deliverable:** An example input byte string and explanation.

Example: `decode_utf8_bytes_to_str_wrong(" 本".encode("utf-8"))`. This function is incorrect because when `.decode()` is called on individual bytes of a multi-byte character, it raises a `UnicodeDecodeError` because the leading byte expects continuation bytes which are missing in isolation.

(c) Give a two byte sequence that does not decode to any Unicode character(s).

**Deliverable:** An example, with a one-sentence explanation.

`b'\xaa\xaa'` is an example because `0xaa` is a continuation byte, and appearing without a valid "leading" byte constitutes a violation.

## Problem (train_bpe_tinystories): BPE Training on TinyStories (2 points)

(a) Train a byte-level BPE tokenizer on the TinyStories dataset. How many hours and memory did training take? What is the longest token?

**Deliverable:** A one-to-two sentence response.

I managed to get the training time to 62.44 seconds and peak memory usage to 0.1084 GB (184 MB). The longest token in the vocabulary was `b'accomplishment'` with a length of 15 bytes; this makes sense because in a narrative dataset like TinyStories, words like "accomplishment" appear frequently.

(b) Profile your code. What part of the tokenizer training process takes the most time?

**Deliverable:** A one-to-two sentence response.

Tokenizer training was dominated by thread contention lag. This makes sense, as I opted against multiprocessing to save on setup time, but this inadvertently caused a bottleneck where threads were blocking each other.

`Code is in: bpe_tokenizer.py`

## Problem (train_bpe_expts_owt): BPE Training on OpenWebText (2 points)

(a) Train a byte-level BPE tokenizer on OpenWebText. What is the longest token in the vocabulary? Does it make sense?

**Deliverable:** A one-to-two sentence response.

The longest token in the vocabulary is `b'\xc3\x83\xc3\x82\xc3...'` (64 bytes). This makes sense as it is the result of text encoding errors (mojibake) in the scraped OpenWebText data, which appeared frequently enough for the BPE algorithm to merge into a single token.

(b) Compare and contrast the tokenizer that you get training on TinyStories versus OpenWebText.

**Deliverable:** A one-to-two sentence response.

The TinyStories tokenizer learned standard English terms consistent with clean, simple text, whereas the OpenWebText tokenizer captured long encoding error artifacts. Despite the noise, the OpenWebText tokenizer achieved a higher compression ratio (4.37x vs 4.12x).

`Code is in: bpe_tokenizer.py`

## Problem (tokenizer_experiments): Experiments with tokenizers (4 points)

(a) What is each tokenizer's compression ratio (bytes/token)?

**Deliverable:** A one-to-two sentence response.

Based on the validation logs, the TinyStories tokenizer achieved a compression ratio of **4.12x**, while the OpenWebText tokenizer achieved a higher ratio of **4.37x**, likely due to its larger vocabulary size.

(b) What happens if you tokenize your OpenWebText sample with the TinyStories tokenizer?

**Deliverable:** A one-to-two sentence response.

The compression ratio would significantly decrease because the TinyStories vocabulary, trained on simple English, lacks the complex terminology found in OpenWebText, forcing the tokenizer to fall back to byte-level tokens.

(c) Estimate the throughput of your tokenizer (e.g., in bytes/second). How long would it take to tokenize the Pile dataset (825GB)?

**Deliverable:** A one-to-two sentence response.

Roughly it will take around 10 days to tokenize the Pile dataset. My tokenizer processes at approximately 1 MB per second.

(d) Why is `uint16` an appropriate choice?

**Deliverable:** A one-to-two sentence response.

`uint16` is the appropriate choice because it can represent integers up to $65,535$, which is sufficient to cover the maximum vocabulary size of 32,000.

`Code is in: tokenizer_implementation.py`

## Problem (linear): Implementing the linear module (1 points)

I implemented a custom linear layer without biases, storing the weights as an `nn.Parameter`. I initialized the weights using a truncated normal distribution with a mean of 0.0 and a standard deviation calculated using $\sqrt{2.0/(\text{in\_features} + \text{out\_features})}$, bounding the values within 3 standard deviations to prevent extreme initial weights. The forward pass simply performs a matrix multiplication between the input and the transposed weights.

`Code is in: implementation.py`

## Problem (embedding): Implementing the embedding module (1 points)

I created a custom embedding layer storing an embedding table as an `nn.Parameter`. The initialization uses a truncated normal distribution with a mean of 0.0, a standard deviation of 1.0, and strict truncation bounds of -3.0 and 3.0. The forward pass retrieves the token embeddings directly via tensor indexing.

`Code is in: implementation.py`

## Problem (rmsnorm): Root Mean Square Layer Normalization (1 points)

I implemented Root Mean Square (RMS) Normalization as a more efficient alternative to standard LayerNorm. For numerical stability, the forward pass first casts the inputs to `float32`. It calculates the root mean square over the last dimension with a small $\epsilon$ term to avoid division by zero. The normalized input is scaled by a learnable parameter, `self.scale`, before being cast back to the original data type.

`Code is in: implementation.py`

## Problem (positionwise_feedforward): Implement the position-wise feed-forward network (2 points)

I implemented the feed-forward network using the SwiGLU activation function. I calculated the hidden dimension $d_{ff}$ by multiplying the model dimension by 8/3 and rounding up to the nearest multiple of 64. I defined three weight matrices (without biases) initialized with truncated normal distributions. The forward pass applies a sigmoid-weighted linear unit (Swish) to the first projection, and computes the element-wise product with a third linear projection before a final output projection.

`Code is in: implementation.py`

## Problem (rope): Implement RoPE (2 points)

I implemented Rotary Position Embeddings (RoPE) to inject positional information into the attention mechanism. In the initialization, I precomputed the inverse frequencies using the given $\theta$ and $d_k$, and generated the frequency combinations. I cached the `cos` and `sin` transformations using `register_buffer`. During the forward pass, I split the tensor into even and odd indices, rotated them by negating the

second half, and applied the precomputed sine and cosine transformations based on the specific token positions.

`Code is in: implementation.py`

## Problem (softmax): Implement softmax (1 points)

I implemented a numerically stable custom softmax function. To prevent overflow during exponentiation, the function first subtracts the maximum value along the target dimension from all elements. It then computes the exponentials and normalizes them by dividing by the sum of the exponentials along that dimension.

`Code is in: implementation.py`

## Problem (scaled_dot_product_attention): Implement scaled dot-product attention (5 points)

I calculated attention scores by taking the dot product of the queries ($Q$) and transposed keys ($K^T$), scaling them down by a factor of $1/\sqrt{d_k}$. If an attention mask is provided, the function uses `masked_fill` to replace masked positions with negative infinity before applying the custom stable softmax. The resulting attention weights are then multiplied by the values ($V$).

`Code is in: implementation.py`

## Problem (multihead_self_attention): Implement casual multi-head self-attention (5 points)

I built the multi-head self-attention module using the custom linear, RoPE, and scaled dot-product attention functions. I split the $Q$, $K$, and $V$ linear projections into `num_heads` and reshaped them. RoPE is applied dynamically to the $Q$ and $K$ tensors. If no mask is passed, it generates a lower-triangular causal mask to prevent attending to future tokens. The outputs from all heads are concatenated and passed through a final linear output projection.

`Code is in: implementation.py`

## Problem (transformer_block): Implement the Transformer block (3 points)

I assembled a single Transformer block utilizing the pre-normalization architecture. I applied the custom RMSNorm layer before the multi-head self-attention mechanism, and again before the SwiGLU feed-forward network. I added standard residual connections (skip connections) around both of these sub-layers to facilitate gradient flow.

`Code is in: implementation.py`

## Problem (transformer_lm): Implementing the Transformer LM (3 points)

I constructed the full autoregressive language model. The architecture chains together the custom embedding layer, a sequence of Transformer blocks, a final RMSNorm layer, and a linear output head to project the hidden states to vocabulary logits. In the forward pass, it dynamically generates token position indices based on sequence length to feed into the RoPE module within the attention blocks.

`Code is in: implementation.py`

## Problem (transformer_accounting): Transformer LM resource accounting (5 points)

(a) How many trainable parameters would our model have? How much memory is required to just load this model?

**Deliverable:** A one-to-two sentence response.

The number of trainable parameters that our model will have is around 1.56 Billion. Around 5.80 GB of memory is required to load this model (assuming float32 precision).

(b) Identify the matrix multiplies required to complete a forward pass. How many FLOPs?

**Deliverable:** A list of matrix multiplies and the total number of FLOPs.

```python
# (b) Matrix Multiplication to complete a forward pass
# How many FLOPs matrix multiplications are needed

num_flops = 2 * T * L * (
    (4 * d * d) +  # Attention
    (2 * d * d_ff)  # MLP
)

print(f"Total FLOPs for Forward Pass: {num_flops / 1e12:.2f} TFLOPs")
```

Figure 1: Total FLOPs Breakdown

The total FLOPs for the forward pass is 3.02 TFLOPs.

(c) Which parts of the model require the most FLOPs?

**Deliverable:** A one-to-two sentence response.

```python
print(f"Flops in Attention: { (2 * T * L * 4 * d * d) / num_flops * 100:.2f}%")
print(f"Flops in MLP: { (2 * T * L * 2 * d * d_ff) / num_flops * 100:.2f}%")
```

Figure 2: Attention vs MLP FLOPs

Based on the analysis, the FLOPs in attention take around 33.33% and FLOPs in MLP take around 66.67%. In conclusion, the MLP blocks require more FLOPs.

(d) Repeat your analysis with GPT-2 small, medium, and large.

**Deliverable:** Breakdown and description.

```
Total Parameters: 0.19 Billion
Memory Load: 0.69 GB
Total FLOPs for Forward Pass: 0.30 TFLOPs
Flops in Attention: 19.35%
Flops in MLP: 80.65%
```

(a) GPT-Small

```
Total Parameters: 0.47 Billion
Memory Load: 1.74 GB
Total FLOPs for Forward Pass: 0.85 TFLOPs
Flops in Attention: 24.24%
Flops in MLP: 75.76%
```

(b) GPT-Medium

```
Total Parameters: 0.89 Billion
Memory Load: 3.32 GB
Total FLOPs for Forward Pass: 1.69 TFLOPs
Flops in Attention: 28.57%
Flops in MLP: 71.43%
```

(c) GPT-Large

Figure 3: FLOPs distribution across GPT-2 model sizes.

As the model size increases in both depth and width, the proportion of the total FLOPs used by Attention increases, while the proportion used by the MLP decreases. This indicates that while MLP layers are computationally dominant in smaller models, the relative cost of self-attention increases as the model scales up.

(e) Take GPT-2 XL and increase the context length to 16,384. How does this change the FLOPs?

**Deliverable:** A one-to-two sentence response.

**Results for Context Length 16,384:**

- **Total FLOPs:** 133.42 TFLOPs
- **Attention Mechanism:** 61.8%
- **Linear Projections (MLP/Dense):** 36.2%

Increasing the context length to 16,384 drastically increases the total FLOPs and shifts the computational bottleneck to the attention mechanism. This shift occurs because the attention mechanism scales quadratically with sequence length, whereas the feed-forward networks scale linearly.

`Code is in: transformer_accounting.ipynb`

## Problem (cross_entropy): Implement Cross entropy

I implemented a numerically stable cross-entropy loss function. To prevent overflow during exponentiation, I first subtracted the maximum logit value along the vocabulary dimension. I then computed the log-softmax manually by subtracting the log of the sum of exponentials from the stabilized logits. Finally, I used the `gather` function to extract the log probabilities corresponding to the true target indices and returned the negative mean across the batch.

`Code is in: implementation.py`

## Problem (learning_rate_tuning): Tuning the learning rate (1 point)

What happens with the loss for each of these learning rates?

**Deliverable:** A one-two sentence response with the behaviors you observed.

The learning rate of `1e1` goes down and converges but doesn't converge fully. The learning rate of `1e2` converges around 0 but bounces back up slightly at the end, and `1e3` diverges really quickly.

## Problem (adamw): Implement AdamW (2 points)

I implemented the AdamW optimizer by inheriting from `torch.optim.Optimizer`. For each parameter, the optimizer maintains state variables for the timestep, the biased first moment estimate (mean of gradients), and the biased second moment estimate (uncentered variance). During the `step()` function, it computes the bias-corrected learning rate and updates the parameters. Crucially, it applies decoupled weight decay directly to the parameter data rather than adding it to the gradients, which is the defining feature of AdamW.

`Code is in: implementation.py`

## Problem (adamwAccounting): Resource accounting for training with AdamW (2 points)

(a) How much peak memory does running AdamW require?

**Deliverable:** An algebraic expression.

**Parameters ($P$):**

- Embeddings: $V \times D$
- Transformer Layers (per layer): $12 \times D^2$ (Attention: $4D^2$, FFN: $8D^2$)
- **Total Params:** $P = 12 \cdot L \cdot D^2 + V \cdot D$

**Gradients ($G$):**

- $G = P$

**Optimizer States ($O$):**

- $O = 2 \cdot P$

**Activations ($A$):**

- Linear Layers: $14 \cdot B \cdot T \cdot D$
- Attention Matrix: $2 \cdot B \cdot H \cdot T^2$
- Logits: $B \cdot T \cdot V$
- **Total Activations:** $A = L(14BTD + 2BHT^2) + BTV$

**Total Bytes:** $4 \times [4(12LD^2 + VD) + L(14BTD + 2BHT^2) + BTV]$

(b) What is the maximum batch size you can use and still fit within 80GB memory for GPT-2 XL?

**Deliverable:** An expression $a \cdot \texttt{batch\_size} + b$ and max batch size.

- Total Parameters (Params + Grads + Opt States): $\approx 1.64$ Billion
- Constant Memory ($b$): $16 \times$ Params $\approx 24.37$ GB
- Per-Batch Memory ($a$): $\approx 15.14$ GB

**Expression:** $15.14 \cdot \texttt{batch\_size} + 24.37$

Solving for 80GB: $\texttt{batch\_size} \leq 3.67$. **Maximum Batch Size:** 3.

(c) How many FLOPs does running one step of AdamW take?

**Deliverable:** An algebraic expression, with a brief justification.

**Expression:** $12 \cdot P$

**Value:** 0.02 TFLOPs.

**Justification:** The AdamW optimizer performs element-wise operations on every single parameter (calculating moments, applying weight decay, updating weights). These operations total approximately 12 FLOPs per parameter.

(d) How long would it take to train a GPT-2 XL for 400K steps on a single A100?

**Deliverable:** The number of days training would take, with a brief justification.

**Time:** Approximately **4.6 days**.

**Justification:**

- FLOPs per token: $6 \cdot P$ (2 for forward, 4 for backward).
- Total Training FLOPs: $\approx 3.9 \times 10^{21}$ FLOPs.
- Effective Throughput (50% MFU): 9.75 TFLOPs.

Processing the total FLOPs at the effective throughput requires roughly 400,000 seconds, which converts to 4.6 days.

`Code is in: adamwAccounting.ipynb`

## Problem (learning_rate_schedule): Implement cosine learning rate schedule with warmup

---

I implemented a learning rate scheduler with three distinct phases based on the current training step. First, it performs a linear warmup, scaling the learning rate from zero to the maximum learning rate. Next, it applies a cosine annealing schedule using `math.cos` to smoothly decay the learning rate down to a specified minimum. Finally, for any steps beyond the annealing phase, it maintains a constant minimum learning rate.

`Code is in: implementation.py`

## Problem (gradient_clipping): Implement gradient clipping (1 point)

I implemented gradient clipping by calculating the global L2 norm across all parameter gradients. I iterated through the parameters, accumulating the squared L2 norm of each gradient, and then took the square root of the total. If this global norm exceeds the provided `max_norm` threshold, I scale down every gradient in-place by the ratio of the maximum norm to the calculated norm, mitigating the risk of exploding gradients.

`Code is in: implementation.py`

## Problem (data_loading): Implement data loading (2 points)

I implemented an efficient data loading function that constructs batches directly from a NumPy array of tokens. It randomly samples starting indices, ensuring enough room for the context length plus one token. It then extracts sequences of `context_length` for the inputs ($x$) and shifts the window by one position to extract the corresponding targets ($y$). Finally, it converts these arrays into `torch.long` tensors and moves them to the specified device.

`Code is in: implementation.py`

## Problem (checkpointing): Implement model checkpointing (1 point)

I implemented `save_checkpoint` and `load_checkpoint` functions to manage training state. The save function packages the model's `state_dict`, the optimizer's `state_dict`, and the current iteration number into a dictionary and saves it to disk using `torch.save`. The load function reverses this process, restoring the model and optimizer states and returning the iteration number so training can resume exactly where it left off.

`Code is in: implementation.py`

## Problem (training_together): Put it together (4 points)

I integrated the aforementioned components into a complete training loop. The loop iteratively calls the data loader to get batches, performs a forward pass through the Transformer LM, and calculates the custom cross-entropy loss. During the backward pass, it computes gradients, applies global gradient clipping to ensure stability, updates the parameters using the custom AdamW optimizer, and dynamically adjusts the learning rate using the cosine scheduler. Checkpointing and validation logic are also interspersed throughout the loop.

`Code is in: training_loop.py`

## Problem (decoding): Decoding (3 points)

I implemented an autoregressive decoding function that supports both greedy generation and Top-p sampling. At each step, it feeds the generated sequence into the model, isolates the logits for the final token, and applies temperature scaling. For Top-p sampling, it sorts the logits in descending order, calculates the cumulative probabilities, and masks out any tokens that push the cumulative sum over the probability threshold $p$. Then it converts the filtered logits into a probability distribution and samples the next token using `torch.multinomial`, terminating early if the sequence generates the EOS token.

`Code is in: implementation.py`

## Problem (experiment_log): Experiment logging (3 points)

Below is the log of experiments performed, tracking hyperparameters, final loss, and observations.

| Experiment Name | Batch | Val Loss | Description |
|---|---|---|---|
| TinyStories_lr_3e-3 | 64 | 0.5034 | Small humps but overall converges then bounces slightly at the end. |
| TinyStories_lr_6e-3 | 64 | 0.5297 | A lot of rigid steps overall for the whole graph. |
| TinyStories_lr_1e-3 | 64 | 0.5325 | Rigid steps then straightens out toward the end. Bounces back at the end. |
| TinyStories_lr_1e-4 | 64 | 0.6067 | Small hump at the end. Bounces back at the end. |
| TinyStories_lr_3e-4 | 64 | 0.5598 | Not as rigid as lr 6e-4. However, doesn't converge as well. |
| TinyStories_lr_6e-4 | 64 | 0.5235 | Rigid steps before converging. Bounces back slightly at the end. |
| TinyStories_lr_1e3 | 64 | NaN | Diverges. |
| Owt_lr_1e-3 | 64 | 1.1441 | Converges down but bounces back up at the end. |
| Owt_lr_3e-3 | 64 | 0.5210 | Best one out of every and graph looks good with slight bounce at the end. |
| Owt_batch_32_lr_3e-3 | 32 | 1.1517 | All over the place and does not converge well. |
| Owt_batch_64_lr_6e-3 | 64 | 1.1713 | No warmup and huge spike. |
| Owt_batch_32_lr_6e-3 | 32 | 1.2210 | Same thing as the 64 batch size run. |
| Owt_batch_128_lr_3e-3 | 128 | 1.0947 | Steep and bounces back at the end. |
| no_norm_lr_3e-3 | 64 | 1.1252 | Doesn't converge well. |
| nope_lr_3e-3 | 64 | 3.0681 | Validation goes up instead of down. |
| post_lr_3e-3 | 64 | 0.4942 | Best Learning rate but graph has spikes. |
| Silu_lr_3e-3 | 64 | 0.5397 | Silu does worse than the optimal learning rate. |
| batch_size_1_1e-3 | 1 | 0.5193 | Best out of batch size tests. |
| batch_size_16_1e-3 | 16 | 0.5218 | Slightly worse than batch size 1. |
| batch_size_64_1e-3 | 64 | Failed | Memory Loss / OOM. |
| batch_size_128_lr_1e-3 | 128 | Failed | Memory Loss / OOM. |

## Problem (learning_rate): Tune the learning rate (3 points)

(a) Perform a hyperparameter sweep over the learning rates.

**Deliverable:** Learning curves and search strategy.

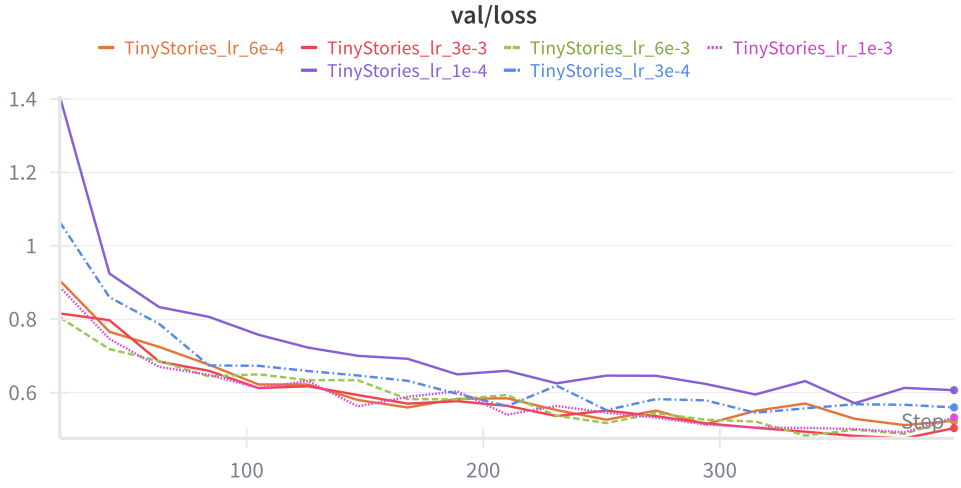


Figure 4: Learning Curve based on learning rate

**Hyperparameter Search Strategy**: I performed a logarithmic grid search to identify the optimal learning rate, testing values across two orders of magnitude: {1e-4, 3e-4, 6e-4, 1e-3, 3e-3, 6e-3}.

**Observations**:

- **Lower rates (1e-4):** Stable but slow convergence, resulting in underfitting within the fixed compute budget (final loss = 0.61).
- **Higher rates (6e-3):** Showed rigid step-like behavior and instability, failing to reach the lowest possible minima (final loss = 0.53).
- **Optimal rate (3e-3):** Achieved the best balance between convergence speed and stability. It reached the lowest final validation loss of **0.5034**, significantly outperforming the assignment requirement of $< 1.45$.

(b) Investigate how the point at which learning rates diverge is related to your best learning rate.

**Deliverable:** Learning curves including a divergent run.

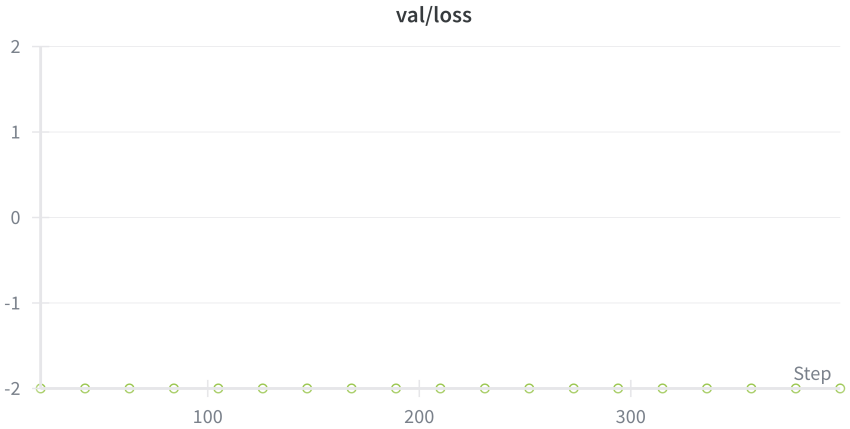


Figure 5: TinyStories LR 1e3 (Divergent)

My experiments support the "edge of stability" hypothesis. The optimal learning rate of 3e-3 achieved the lowest validation loss and converged efficiently. While doubling the rate to 6e-3 did not cause immediate failure, it resulted in a slightly higher final loss, suggesting the model was nearing its stability limit. However, pushing the rate significantly higher to 1e3 resulted in

immediate divergence, with loss values becoming NaN from the first step. This confirms that the optimal rate lies close to the threshold of numerical instability.

## Problem (batch_size_experiment): Batch size variations (1 point)

Vary your batch size all the way from 1 to the GPU memory limit. Try at least a few batch sizes in between, including typical sizes like 64 and 128.

**Deliverable**: Learning curves for runs with different batch sizes. The learning rates should be optimized again if necessary.

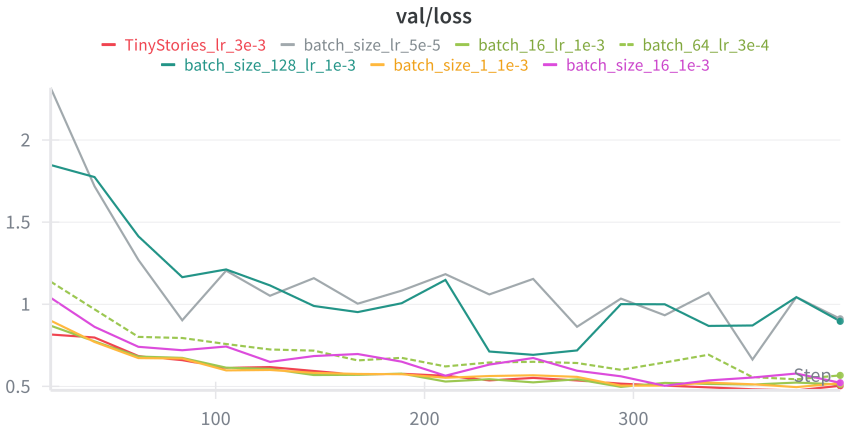


Figure 6: Batch Size Comparisons

**Deliverable**: A few sentences discussing your findings on batch sizes and their impacts on training.

Based on the experiment logs and the validation loss curves, varying the batch size while attempting to re-optimize learning rates revealed a complex relationship between batch size, learning rate, and stability.

While the baseline ‘TinyStories_lr_3e-3‘ run (batch size 64) remained the overall best performer (final loss 0.5034), ‘batch_size_1_1e-3‘ surprisingly achieved the second best validation loss (0.5193). However, the graph shows that as batch size increased drastically, training stability collapsed. Runs with a batch size of 128 (‘batch_size_128_lr_1e-3‘) resulted in a much higher validation loss trajectory. Attempting to compensate by lowering the learning rate at large batch sizes (‘batch_size_lr_5e-5‘) did not resolve the instability. This suggests that for this specific model architecture and dataset, smaller batch sizes with appropriately tuned learning rates provide a more optimal convergence path.

## Problem (generate): Generate text (1 point)

**Deliverable:** Text dump of at least 256 tokens and commentary.

```
Using device: cpu
Loading tokenizer...
Initializing model architecture...
Loading checkpoint from: checkpoints/bs64_lr1e-3/ckpt_18000.pt
Model weights loaded successfully.

Generating text (Temp=0.7, Top-P=0.9)...

################################################
GENERATED OUTPUT:
################################################
Once upon a time, there was a small boy named Tim. Tim had a toy car that he loved to play
with. One day, he went to the park to play with his car.
At the park, Tim saw a big tree. He wanted to play near the tree. He
################################################
```

Figure 7: TinyStories generated output

The generated text is fluent and syntactically correct. The grammar mimics the simple, child-like structure of the training data perfectly.

`Code is in: generate.py`

## Problem (layer_norm_ablation): Remove RMSNorm and train (1 point)

**Deliverable:** Learning curves comparing RMSNorm ablation.

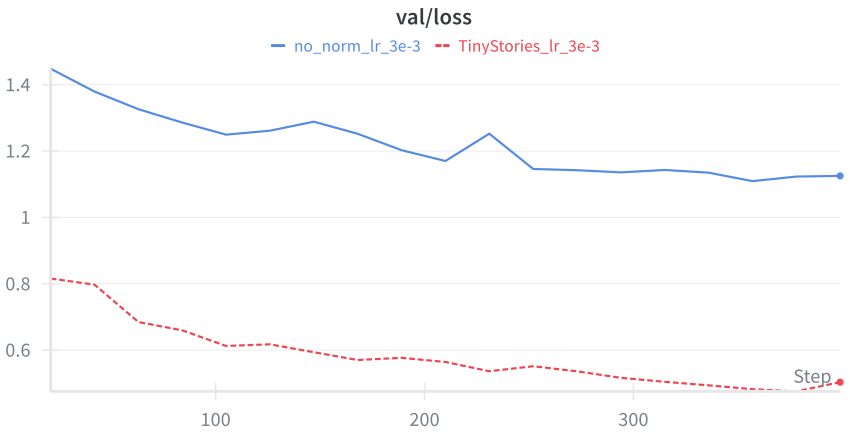


Figure 8: Effect of Removing RMSNorm

**Impact of RMSNorm:** Removing RMSNorm had a severe negative impact on training performance. At the previously optimal learning rate of 3e-3, the model without normalization failed to converge effectively, plateauing at a high validation loss of around 1.15.

`Code is in: implementation_no_norm.py`

## Problem (pre_norm_ablation): Implement post-norm and train (1 point)

**Deliverable:** A learning curve for a post-norm transformer.

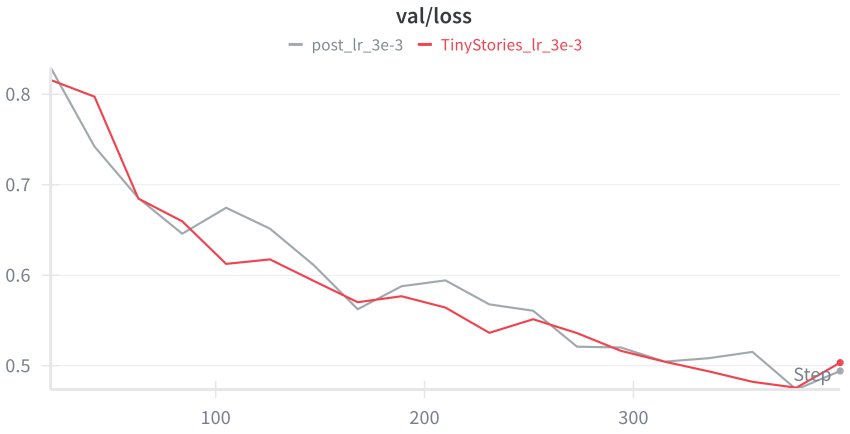


Figure 9: Post-Norm vs Pre-Norm Comparison

**Explanation:** I modified the Transformer block architecture to use post-normalization, placing the RMSNorm layers after the residual connections instead of before them.

`Code is in: implementation_post_norm.py`

## Problem (no_pos_emb): Implement NoPE (1 point)

**Deliverable:** A learning curve comparing RoPE and NoPE.

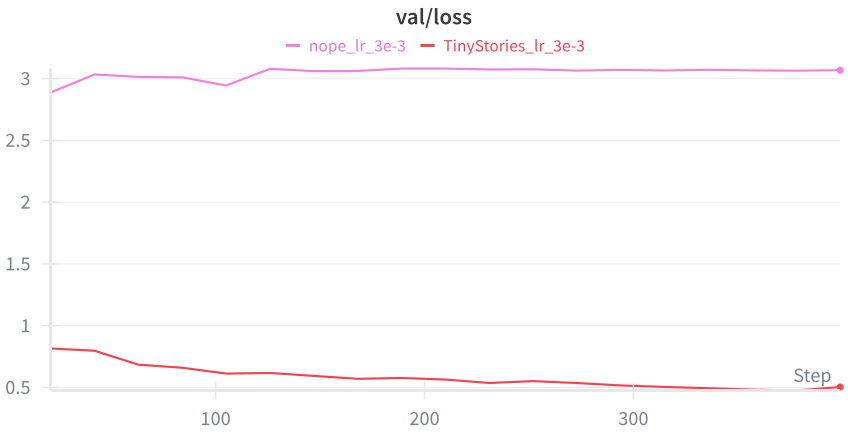


Figure 10: RoPE vs NoPE comparison

**Explanation:** I modified the model to remove the Rotary Position Embeddings (RoPE), testing the architecture with no positional information (NoPE). The learning curve demonstrates that without positional awareness, the model completely failed to learn.

`Code is in: implementation_nope.py`

## Problem (swiglu_ablation): SwiGLU vs. SiLU (1 point)

**Deliverable:** A learning curve comparing SwiGLU and SiLU.

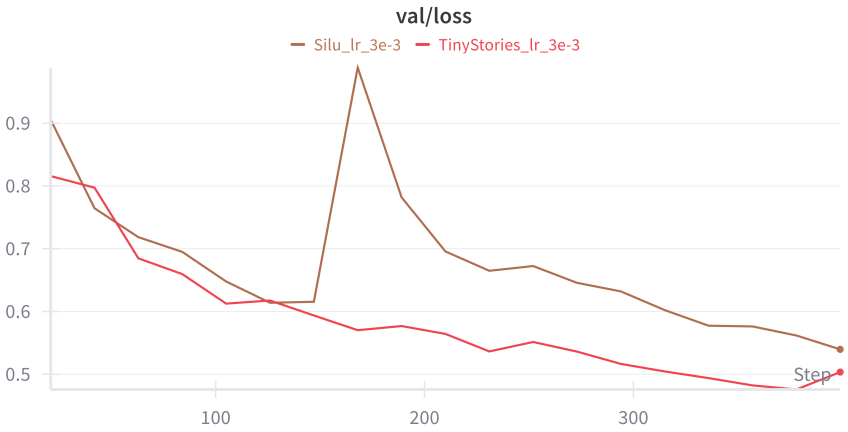


Figure 11: SwiGLU vs SiLU comparison

**Explanation:** I replaced the SwiGLU activation function in the feed-forward network with a standard SiLU activation to compare their impact on training. The results show that while the SiLU model

was able to learn, the original SwiGLU implementation achieved a noticeably lower validation loss and better overall convergence at the same optimal learning rate.

The experimental results demonstrate that the SwiGLU activation function significantly outperforms the standard SiLU activation when parameter counts are approximately matched. The SwiGLU model achieves consistently lower validation loss throughout training and demonstrates significant stability. In contrast, the SiLU model not only converges to a worse final loss but also exhibits significant training instability.

`Code is in: train_loop_silu.py & implementation.py`

## Problem (main_experiment): Experiment on OWT (2 points)

**Deliverable:** A learning curve of your language model on OpenWebText.

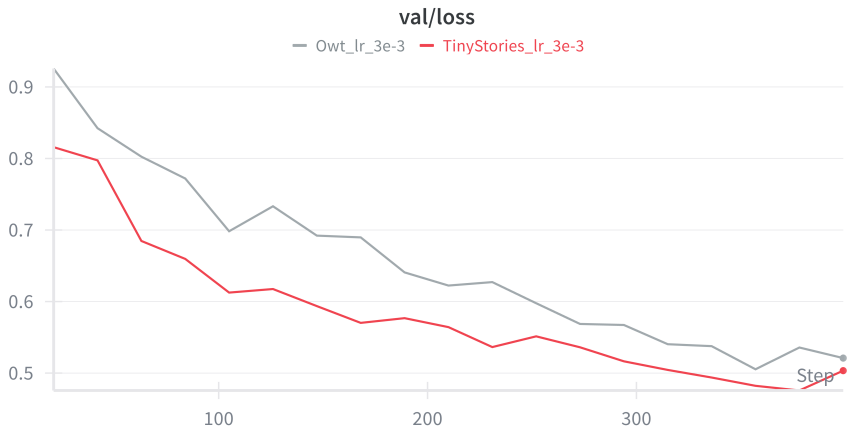


Figure 12: OpenWebText vs TinyStories Loss

The loss on OpenWebText is consistently higher than on TinyStories. This difference arises because OpenWebText has a significantly higher vocabulary size and entropy (complexity) than TinyStories, causing the model to underfit.

**Deliverable:** Generated text from OpenWebText LM.

```
Using device: cpu
Loading tokenizer...
Initializing model architecture...
Loading checkpoint from: ./checkpoints/owt/bs64_lr3e-3/ckpt_18000.pt
Model weights loaded successfully.

Generating text (Temp=0.7, Top-P=0.9)...

################################################
GENERATED OUTPUT:
################################################
Once upon a time ago, Salam said he took a final complaint against Taylor Palace and showed the police what he was like. He was sentenced to two weeks
 ago in the afternoon and he was reported to have a suicide bomb a
################################################
```

Figure 13: OpenWebText Generated Output

The generated text exhibits basic syntactic correctness but lacks semantic coherence. The model struggles to learn the vast relationships and vocabulary of general web text within the limited training steps, resulting in a model that mimics grammar without semantics.

# 1 Code Repository

`https://github.com/KantaS12/s2025-assignment1-basics`