

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.

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).

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.

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.

```
# (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.

```
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.

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 (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 $a \cdot \text{batch_size} + b$ and max batch size.

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

Expression: $15.14 \cdot \text{batch_size} + 24.37$

Solving for 80GB: $\text{batch_size} \leq 3.67$. **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.

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.

Continued on next page...

Table 1 – continued from previous page

Experiment Name	Batch	Val Loss	Description
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.

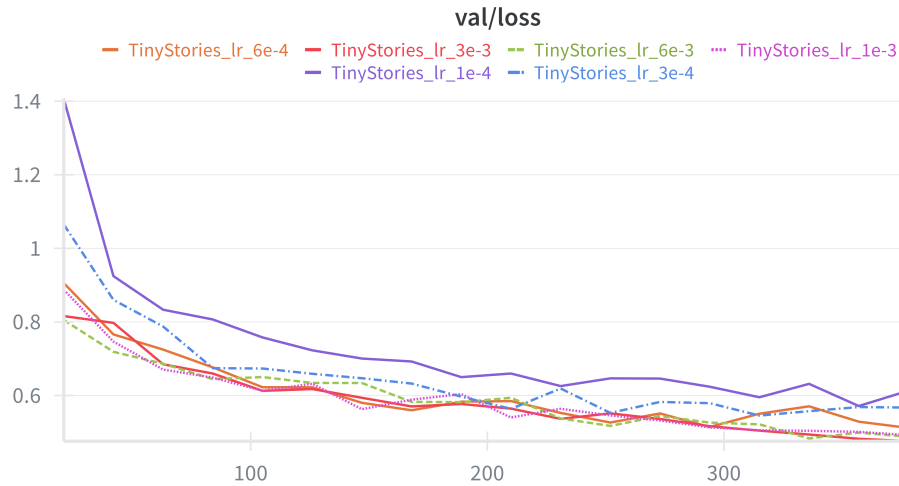


Figure 4: Learning Curve based on learning rate

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.

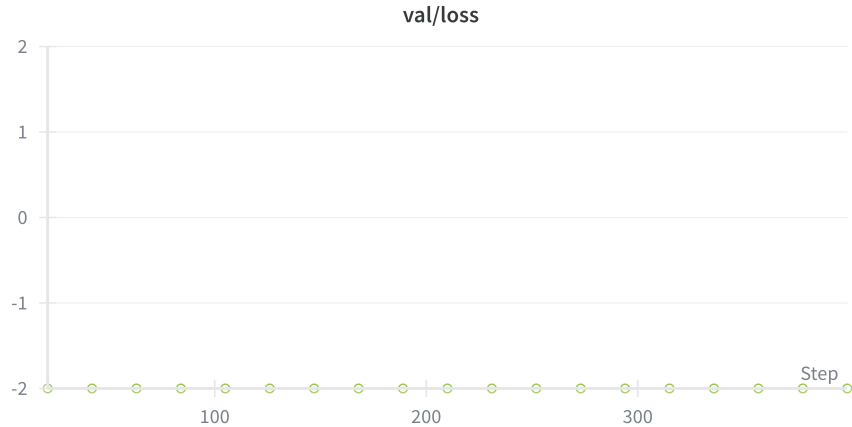


Figure 5: TinyStories LR 1e3 (Divergent)

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 (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 6: TinyStories generated output

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

Problem (layer__norm__ablation): Remove RMSNorm and train (1 point)

Deliverable: Learning curves comparing RMSNorm ablation.

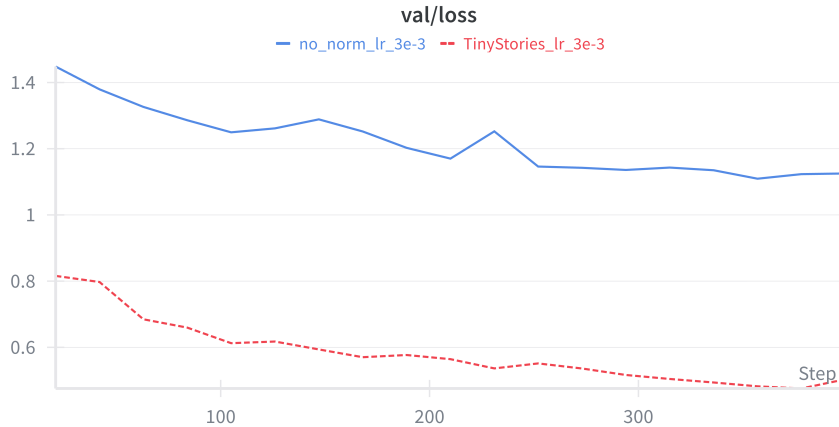


Figure 7: Effect of Removing RMSNorm

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.

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

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

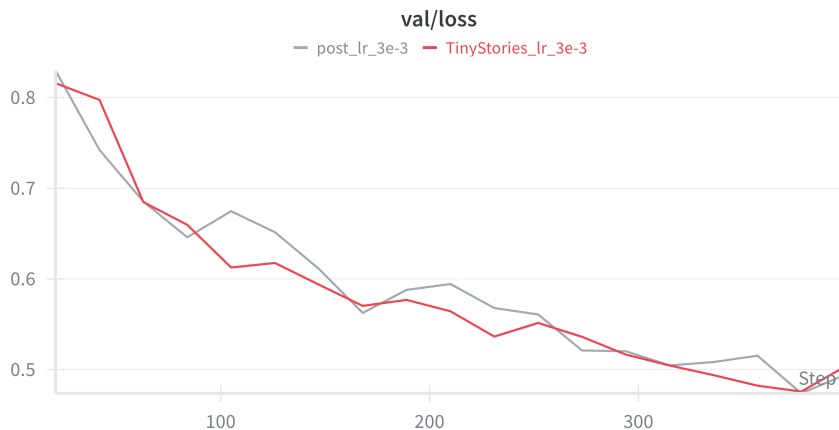


Figure 8: Post-Norm vs Pre-Norm Comparison

Problem (no_pos_emb): Implement NoPE (1 point)

Deliverable: A learning curve comparing RoPE and NoPE.

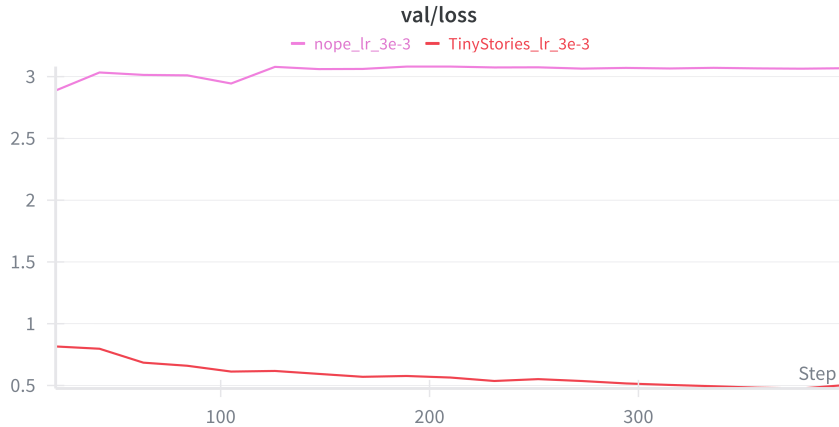


Figure 9: RoPE vs NoPE comparison

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

Deliverable: A learning curve comparing SwiGLU and SiLU.

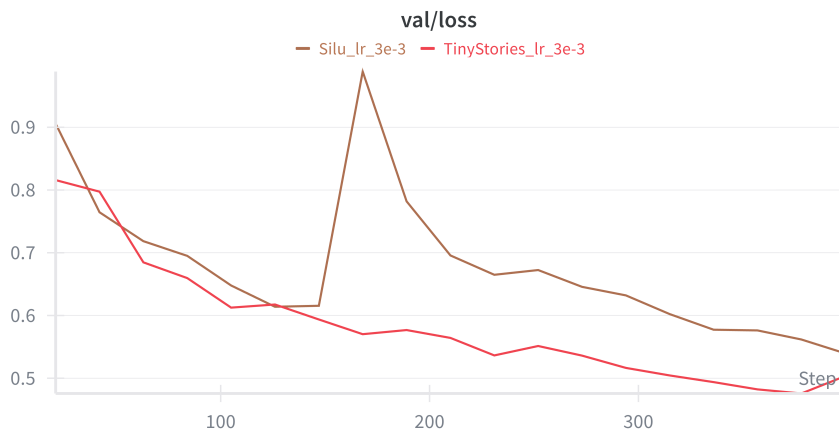


Figure 10: SwiGLU vs SiLU comparison

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.

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

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

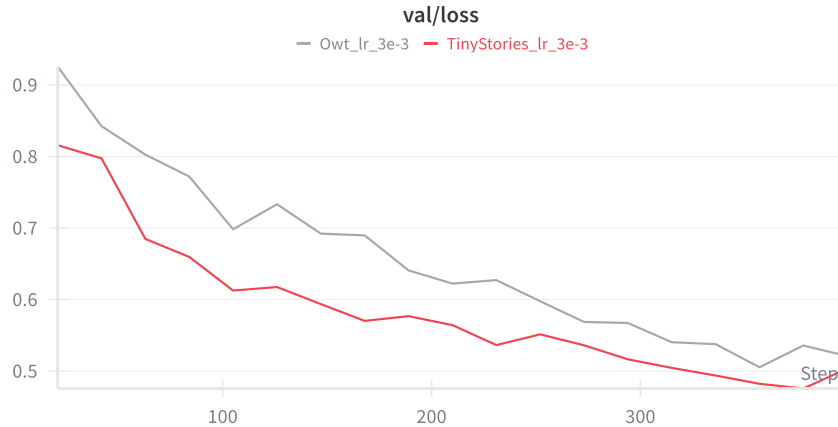


Figure 11: OpenWebText vs TinyStories Loss

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 12: 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.