
Assignment 2

Character-Level GPT: Training and Generation

Residual blocks • feed-forward network • vocabulary projection • cross-entropy • automatic differentiation • Adam • validation • sampling

implemented from scratch with fundamental tensor operations, in Python and in MATLAB

Ömer Kahyaoglu

Student No. 090240741 • kahyaoglu20@itu.edu.tr

25 October 2026

Summary

The Assignment 1 foundation (embeddings, LayerNorm, causal attention) is completed into a two-block character GPT: residual sums, a GELU feed-forward network, the vocabulary projection and a stable cross-entropy. It is implemented twice: in Python the gradients come from the reverse-mode automatic-differentiation engine written for these assignments, in MATLAB from the Deep Learning Toolbox (`dlgradient`). Every gradient is also derived by hand and checked against centred differences. Adam is written out from its definition.

Result: 67/67 verification checks pass in Python; 67/67 in MATLAB (24.1.0.2537033 (R2024a), gradients by `dlgradient`). $P = 112\,000 = 128V + 103\,680$ trainable parameters. After 24 500 Adam steps (50 epochs of $\lfloor N_{\text{tr}}/BT \rfloor = 490$ steps) on Tiny Shakespeare, the held-out cross-entropy falls from 4.1972 (the uniform level is $\log V = 4.174387$) to 1.6339 nats per character, against 1.3880 on training text: a generalization gap of +0.2459 nats, partly due to the held-out plays' new speakers and words. The MATLAB program, trained separately with its own random windows, reaches 1.6375 (training text 1.3906). The 200-character sample has speaker tags, line breaks and mostly real English words, but no coherent sentences.

Contents

Reproducibility at a glance	2
1 Implementation design	2
1.1 From the Assignment 1 foundation to a trainable GPT	2
1.2 What is reused from Assignment 1, and how compatibility was verified	3
1.3 Code organisation	3
1.4 Conventions	3
1.5 Automatic differentiation: three new local derivatives	4
1.6 How every derivative is verified	4
1.7 Departures from the specification and choices it leaves open	4
2 Question 1: residual connections and the feed-forward network	5
3 Question 2: vocabulary projection and cross-entropy	9
4 Question 3: automatic differentiation and Adam	12
5 Required Benchmark B: deterministic logits, loss, gradients and Adam	15
6 Question 4: training schedule, validation and gradient checks	16
Case B4: Tiny Shakespeare end to end	20
7 Question 5: autoregressive generation	23
8 Question 6: integration, verification and submission	27
8.1 The numbered checks of the assignment	27
8.2 Minimum acceptance tests	28
8.3 Case B5 and the summary of the validation suite	28
9 Table of numerical verification errors	29
10 Discussion	31

Reproducibility at a glance

Languages	Python 3.13.17 with NumPy 2.5.3 (matplotlib 3.11.2 for figures only). MATLAB: 24.1.0.2537033 (R2024a) with the Deep Learning Toolbox 24.1; training gradients by <code>dlgradient</code>
Platform	Python: Linux-6.18.44-fc-v153-x86_64-with-glibc2.39; MATLAB: PCWIN64; IEEE-754 double precision everywhere
Seeds	Initial weights: Assignment 1's saved <code>params_init.npz</code> (Python, seed 451) and <code>params_init.mat</code> (MATLAB, <code>rng(451, 'twister')</code>). Separate generators by purpose: training batches 451, validation batches 452, training previews 453, final sample 454 (<code>numpy.random.default_rng</code> ; MATLAB: <code>RandStream('mt19937ar', 'Seed', ...)</code> with the same numbers)
Corpus	<code>tiny_shakespeare.txt</code> , $N = 1\,115\,394$ characters, $V = 65$, SHA-256 <code>86c4e6aa9db7c042...</code>
Split	$N_{\text{tr}} = \lfloor 0.9N \rfloor = 1\,003\,854$, $N_{\text{va}} = 111\,540$, the same contiguous split and vocabulary as Assignment 1
Model	$B = 32$, $T = 64$, $d = 64$, $L = 2$, $d_{\text{ff}} = 4d = 256$, one attention head, $\sigma = 0.02$, $\epsilon = 10^{-5}$, mask -10^9 , untied head, tanh-GELU
Optimizer	Adam, $\eta = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon_A = 10^{-8}$; no clipping, no weight decay, no schedule
Schedule	$S_{\text{epoch}} = \lfloor N_{\text{tr}}/(BT) \rfloor = 490$, $S_{\text{total}} = 50S_{\text{epoch}} = 24\,500$ steps; validation on $\min(8, B) = 8$ fresh held-out windows at step 1 and every 50 steps (491 evaluations); final losses on 512 fixed validation and 512 fixed training windows
Parameters	$P = 112\,000 = 128V + 103\,680$
Verification	67/67 checks pass in Python (complete run 61.9 min, of which training 61.5 min); 67/67 in MATLAB (complete run 51.9 min, of which training 51.7 min); full list in Table 8
Run	<code>python python/run_assignment2.py</code> (from the submission root); <code>python -m unittest discover -s python/tests</code> ; <code>matlab -batch "run('matlab/run_assignment2.m')"</code> ; then <code>python report/build_assets.py</code> and <code>pdflatex</code>

1 Implementation design

1.1 From the Assignment 1 foundation to a trainable GPT

Assignment 1 produced, for every batch member, the embedded sequence $H_b^{(0)} = O_b W_E + W_P[1:T, :]$ and the causal attention output $F_b = \text{softmax}_{\text{row}}(Q_b K_b^\top / \sqrt{d} + M) V_b W_O$ of $U_b = \text{LN}_1(H_b^{(0)})$. This assignment adds everything that turns that stage into a language model:

$$\underbrace{X}_{B \times T} \xrightarrow{\text{embed}} H^{(0)} \xrightarrow{\text{block 1}} H^{(1)} \xrightarrow{\text{block 2}} H^{(2)} \xrightarrow{\text{LN}_f, \text{flatten}} \underbrace{\bar{H}}_{BT \times d} \xrightarrow{W_{\text{voc}}} \underbrace{Z}_{BT \times V} \xrightarrow{\text{cross-entropy}} \mathcal{L}, \quad (1.1)$$

$$\text{block } \ell : \quad R = H^{(\ell-1)} + \text{Attn}(\text{LN}_1(H^{(\ell-1)})), \quad H^{(\ell)} = R + \text{FFN}(\text{LN}_2(R)), \quad (1.2)$$

then differentiates $\mathcal{L}$, updates every parameter with Adam, measures held-out loss, and samples text.

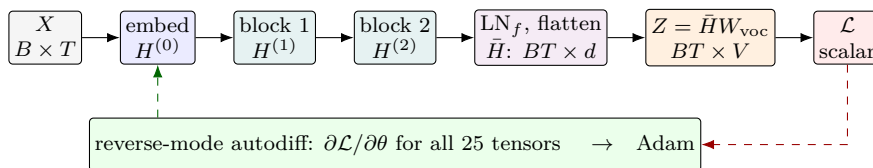


Figure 1: The complete computation. Solid arrows are the forward pass, dashed arrows the gradient and the update.

1.2 What is reused from Assignment 1, and how compatibility was verified

The Assignment 1 package `python/gpt_a1/` and its MATLAB functions are included *unchanged*, so this archive is self-contained. The new code imports them and never edits them.

Reused function	Role here	Compatibility evidence
Tensor (autodiff engine)	differentiates the whole GPT	every hand-derived gradient agrees with it to 1.4×10^{-16} (Q3.2)
load_corpus, build_vocab, encode, split_contiguous, get_batch	data, split and windows	identical vocabulary to Assignment 1's saved <code>vocab.json</code> (D.1)
init_params, count_params	the 25 tensors, $P = 112\,000$	the saved <code>params_init.npz</code> equals a re-run of the Assignment 1 initialization (D.2, D.3); fixed-input logits agree exactly (D.4)
embed, layer_norm, softmax_rows, causal_mask	inside every block	block-1 attention on a held-out batch equals the standalone Assignment 1 routine (B5.1: 0)
embedding_backward, layer_norm_backward, attention_backward	the hand-derived chain rule	term-by-term agreement with autodiff through all 25 tensors (Q3.2)

1.3 Code organisation

File	Content
<code>python/gpt_a2/ops.py</code>	the new differentiable primitives <code>tanh</code> , <code>log</code> , <code>pick</code> (target gathering) and GELU composed from them
<code>python/gpt_a2/model.py</code>	attention inside a block, feed-forward network, residual blocks, logits, cross-entropy; hand-derived backward maps
<code>python/gpt_a2/optim.py</code>	Adam from its definition
<code>python/gpt_a2/train.py</code>	step budget, training loop, held-out validation, fixed-window evaluation
<code>python/gpt_a2/sample.py</code>	context cropping and categorical (or greedy) sampling
<code>python/run_assignment2.py</code>	runs Q1–Q6, Benchmark B, Cases B1–B5, the acceptance tests, training and generation; writes <code>results/python/</code>
<code>python/tests/</code>	15 unit tests of the new package and 23 of the reused one (standard-library <code>unittest</code>)
<code>matlab/src/*.m</code> , <code>matlab/run_assignment2.m</code>	the MATLAB twin: one function per operation, the same check ids
<code>report/build_assets.py</code>	turns <code>results/*/results.json</code> into every table, number and listing of this report

1.4 Conventions

- **Indices.** Python labels are $0, \dots, V-1$; MATLAB labels are $1, \dots, V$. Worked examples print the mathematical one-based labels.
- **Right multiplication.** Every linear map is $Y = XW + \mathbf{1}b$ with $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, as in the assignment; no transposed `nn.Linear` convention.
- **Flattening order** (Section 3). Python flattens $B \times T \times d$ hidden states and $B \times T$ targets row-major, so row $r = bT + t$ (time-major within each batch member). MATLAB flattens both column-major, so $r = b + (t-1)B$. Each language uses one rule for *both* hidden states and labels.
- **Objective.** $\mathcal{L}$ is the mean over all BT predicted characters, so the $1/(BT)$ factor appears once, inside the loss, and never again in a gradient.

- **Precision.** IEEE double precision throughout.

1.5 Automatic differentiation: three new local derivatives

The engine of Assignment 1 records, for every operation, its parents and a closure that applies the local chain rule $\bar{x} += (\partial y / \partial x)^\top \bar{y}$; `backward()` runs the closures in reverse topological order. The GPT head and the GELU need three more primitives, added through the same hook:

$$d \tanh x = (1 - \tanh^2 x) dx, \quad d \log x = dx/x, \quad z_r = Z_{r,y_r} \Rightarrow \bar{Z}_{r,y_r} += \bar{z}_r. \quad (1.3)$$

GELU is *not* a primitive: it is composed from `tanh`, products and sums, so its derivative comes from the engine and can be compared with the hand-derived $g'(z)$ (Q1.3). In MATLAB the same role is played by `dlfeval/dlgradient` on `dlarray` copies of the parameters (`dl_gpt_loss.m`).

Code 1: Python: the new differentiable primitives (`gpt_a2/ops.py`).

```

1 def tanh(x):
2     """Elementwise tanh; the backward pass reuses the forward value y: g * (1 - y^2)."""
3     y = np.tanh(x.data)
4
5     def backward(g):
6         x._accumulate(g * (1.0 - y * y))
7     return x._make(y, (x,), "tanh", backward)
8
9 def log(x):
10    """Elementwise natural logarithm (inputs must be positive): g / x."""
11    def backward(g):
12        x._accumulate(g / x.data)
13    return x._make(np.log(x.data), (x,), "log", backward)
14
15 def pick(Z, y):
16    """z_r = Z[r, y_r] for a 2-D Z (R x V) and integer labels y (R): vectorized target gathering."""
17    rows = np.arange(Z.shape[0])
18    y = np.asarray(y)
19
20    def backward(g):
21        full = np.zeros_like(Z.data)
22        full[rows, y] = g                                     # each row receives its upstream value at its own label
23        Z._accumulate(full)
24    return Z._make(Z.data[rows, y], (Z,), "pick", backward)
25
26 def gelu(z):
27    """g(z) = z/2 * (1 + tanh(sqrt(2/pi) (z + 0.044715 z^3))), built from engine operations."""
28    u = (z + z * z * z * GELU_C) * SQRT_2_OVER_PI          # z*z*z: three cheap products instead of a float power
29    return z * 0.5 * (tanh(u) + 1.0)

```

1.6 How every derivative is verified

Each gradient is computed in three independent ways: (i) the hand-derived matrix calculus of this report, written on plain arrays (`*_backward`); (ii) automatic differentiation of the forward code; (iii) centred finite differences $[\mathcal{L}(\theta + \delta e_k) - \mathcal{L}(\theta - \delta e_k)]/(2\delta)$, with the relative discrepancy $\rho = |a - b|/\max(1, |a|, |b|)$ of the assignment. (i) against (ii) is checked term by term for every intermediate, and (ii) against (iii) for at least one scalar of every kind of tensor.

1.7 Departures from the specification and choices it leaves open

1. **Two implementations.** Both are complete and both were run. Python is the reference for the figures and the worked numbers of this report. MATLAB passes the same checks under the same ids (Table 8), trains its own model (Case B4) and generates its own sample (Section 7).
2. **Own autodiff in Python instead of PyTorch.** For automatic differentiation the assignment names `loss.backward()` in PyTorch or `dlfeval` and `dlgradient` in MATLAB. The MATLAB

program uses `dlgradient` exactly as named. The Python program uses the engine of Assignment 1, which provides the same `backward()` interface without a framework; both automatic gradients are checked against the hand-derived chain rule and against finite differences.

3. **Integer step budget.** $N_{\text{tr}}/(BT) = 1\,003\,854/2048$ is not an integer; $S_{\text{epoch}} = \lfloor N_{\text{tr}}/(BT) \rfloor = 490$.
4. **Starting point.** Training starts from the initialization Assignment 1 saved, not from a new draw (each language from its own file, `params_init.npz` or `params_init.mat`): the hand-off is then tested exactly (D.3, D.4, B5.1).
5. **Low-noise final estimate.** In addition to the periodic 8-window validation batches, the final losses are averaged over 512 fixed validation windows whose starts are spread evenly over the held-out part (and likewise 512 training windows), as the assignment recommends.

2 Question 1: residual connections and the feed-forward network

1. Purpose and assumptions

Attention lets each token read earlier tokens; the feed-forward network (FFN) then transforms each token's features on its own, with a nonlinearity. Residual sums keep an identity path around both sub-layers, so information and gradients can pass a block unchanged when the sub-layer contributes little. The block is pre-normalized: each sub-layer sees a LayerNorm of its input, the residual path does not.

2. Derivation

For batch member b and block ℓ , with F_b the attention output of Assignment 1,

$$R_b = H_b^{(\ell-1)} + F_b, \quad D_b = \text{LN}_2(R_b), \quad (2.1)$$

$$J_b = D_b W_1^{(\ell)} + \mathbf{1}_T b_1^{(\ell)} \in \mathbb{R}^{T \times 4d}, \quad G_b = g(J_b) W_2^{(\ell)} + \mathbf{1}_T b_2^{(\ell)} \in \mathbb{R}^{T \times d}, \quad H_b^{(\ell)} = R_b + G_b, \quad (2.2)$$

with $W_1 \in \mathbb{R}^{d \times 4d}$, $b_1 \in \mathbb{R}^{1 \times 4d}$, $W_2 \in \mathbb{R}^{4d \times d}$, $b_2 \in \mathbb{R}^{1 \times d}$. Entry by entry,

$$(J_b)_{ik} = \sum_{j=1}^d (D_b)_{ij} (W_1)_{jk} + (b_1)_k, \quad (G_b)_{ij} = \sum_{k=1}^{4d} g((J_b)_{ik}) (W_2)_{kj} + (b_2)_j. \quad (2.3)$$

Row i of G_b depends on row i of D_b only: the FFN mixes *features* of one token. Attention instead forms $A_b V_b$, whose row i is $\sum_{j \leq i} A_{ij} V_j$: it mixes *tokens*. Because no token interacts with another in the FFN, all BT rows are flattened into one $(BT) \times d$ matrix and multiplied at once.

GELU. The implementation uses the tanh approximation, with $a = \sqrt{2/\pi}$, $c = 0.044715$, $u(z) = a(z + cz^3)$,

$$g(z) = \frac{z}{2} (1 + \tanh u(z)), \quad g'(z) = \frac{1}{2} (1 + \tanh u) + \frac{z}{2} (1 - \tanh^2 u) a (1 + 3cz^2). \quad (2.4)$$

The exact GELU is $z\Phi(z)$ with Φ the standard normal distribution function.

Residual Jacobian. For $R = H + F(H)$, the differential is $dR = dH + J_F(H) dH$, so

$$\frac{\partial R}{\partial H} = I + J_F(H), \quad \frac{\partial \mathcal{L}}{\partial H} = G_R + J_F(H)^\top G_R, \quad (2.5)$$

where $G_R = \partial \mathcal{L} / \partial R$. The first term reaches H unchanged, however small J_F is; stacking blocks multiplies factors $(I + J_{F_\ell})$, whose identity parts keep the product from vanishing. This is why residual connections make deeper stacks trainable.

FFN derivatives. With $\mathcal{R} = g(J)$ (rows flattened), $G = \mathcal{R}W_2 + \mathbf{1}b_2$ and upstream G_G :

$$\begin{aligned} G_{W_2} &= \mathcal{R}^\top G_G, & G_{b_2} &= \mathbf{1}^\top G_G, & G_{\mathcal{R}} &= G_G W_2^\top, & G_J &= G_{\mathcal{R}} \odot g'(J), \\ G_{W_1} &= D^\top G_J, & G_{b_1} &= \mathbf{1}^\top G_J, & G_D &= G_J W_1^\top. \end{aligned} \quad (2.6)$$

Each bias gradient is a sum over all BT rows because the same bias row is added to every row. Through the two residual sums, $G_R = G_{H'} + (\text{LN}_2\text{-backward of } G_D)$ and $G_H = G_R + (\text{LN}_1\text{-backward of the attention input gradient})$.

3. Code

Code 2: Python: the feed-forward network and the residual block (gpt_a2/model.py).

```
1 def feed_forward(D, W_1, b_1, W_2, b_2):
2     """G = g(D W1 + 1 b1) W2 + 1 b2: expand to 4d, GELU, contract to d. Mixes *features* of each token.
3
4     No token interacts with another here, so the B x T rows are flattened into one (BT) x d matrix."""
5     B, T, d = D.shape
6     J = D.reshape(B * T, d) @ W_1 + b_1 # (BT x d)(d x 4d) + 1 b1: bias row broadcast to every
7     Ga = ops.gelu(J) # elementwise tanh-approximation of GELU
8     G = (Ga @ W_2 + b_2).reshape(B, T, d) # (BT x 4d)(4d x d) + 1 b2, back to B x T x d
9     return G, {"J": J, "Ga": Ga}
10
11 def block(H, P, l, eps=1e-5, neg=-1e9):
12     """One pre-normalization Transformer block: R = H + Attn(LN1(H)), H' = R + FFN(LN2(R))."""
13     k = f"block[{l}]."
14     U, ln1 = A1.layer_norm(H, P[k + "ln1_gamma"], P[k + "ln1_beta"], eps)
15     F, att = attention(U, P[k + "W_QKV"], P[k + "W_0"], neg)
16     R = H + F # first residual: the identity path keeps H
17     D, ln2 = A1.layer_norm(R, P[k + "ln2_gamma"], P[k + "ln2_beta"], eps)
18     G, ffn = feed_forward(D, P[k + "W_1"], P[k + "b_1"], P[k + "W_2"], P[k + "b_2"])
19     Hn = R + G # second residual
20     return Hn, {"H": H, "U": U, "ln1": ln1, "F": F, "R": R, "D": D, "ln2": ln2, "G": G, **att, **ffn}
```

Code 3: Python: attention as it runs inside a block (batched over the leading axis).

```
1 def attention(U, W_QKV, W_0, neg=-1e9):
2     """Single-head causal self-attention of a (B, T, d) tensor, batched over the leading axis.
3
4     QKV = U W_QKV (B,T,3d); S = Q K^T / sqrt(d) + M (B,T,T); A = softmax_row(S); F = (A V) W_0 (B,T,d).
5     """
6     B, T, d = U.shape
7     QKV = (U.reshape(B * T, d) @ W_QKV).reshape(B, T, 3 * d) # one (BT x d)(d x 3d) product for Q, K and V
8     Q, K, Vm = QKV[:, :, :d], QKV[:, :, d:2 * d], QKV[:, :, 2 * d:]
9     S = (Q @ K.mT) * (1.0 / np.sqrt(d)) + A1.causal_mask(T, neg) # one T x T score page per sample
10    A = A1.softmax_rows(S) # rows are probability distributions over keys 0..i
11    Z = A @ Vm # (B,T,T) @ (B,T,d) -> (B,T,d): mixes *tokens*
12    F = (Z.reshape(B * T, d) @ W_0).reshape(B, T, d)
13    return F, {"Q": Q, "K": K, "V": Vm, "S": S, "A": A, "Z": Z}
```

Code 4: MATLAB: ffn_forward.m.

```
1 function [G, cache] = ffn_forward(D, W_1, b_1, W_2, b_2)
2 %FFN_FORWARD Feed-forward network of every token row: G = g(D W1 + 1 b1) W2 + 1 b2.
3 % D is B x T x d. No token interacts with another, so the B*T rows are
4 % flattened into one (BT) x d matrix; the 1 x 4d and 1 x d bias rows are
5 % added to every row by implicit expansion.
6 [B, T, d] = size(D);
7 D2 = reshape(D, B*T, d); % (BT) x d
8 J = D2 * W_1 + b_1; % (BT x d)(d x 4d) + 1 x 4d
9 Ga = gelu_tanh(J); % (BT) x 4d
10 G = reshape(Ga * W_2 + b_2, B, T, d); % (BT x 4d)(4d x d) + 1 x d -> B x T x d
11 cache = struct('D2', D2, 'J', J, 'Ga', Ga);
12 end
```

Code 5: MATLAB: block_forward.m.

```
1 function [Hn, c] = block_forward(H, blk, eps_ln)
2 %BLOCK_FORWARD One pre-normalization Transformer block on a B x T x d tensor:
3 % U = LN1(H), F = attention(U), R = H + F, D = LN2(R), G = FFN(D), H' = R + G.
```

```

4 % blk holds W_QKV, W_O, W_1, b_1, W_2, b_2 and the two LayerNorm gain/shift pairs.
5 [U, ln1] = layernorm_forward(H, blk.ln1_gamma, blk.ln1_beta, eps_ln);
6 [F, att] = attention_forward(U, blk.W_QKV, blk.W_O); % Assignment 1, unchanged
7 R = H + F; % first residual
8 [D, ln2] = layernorm_forward(R, blk.ln2_gamma, blk.ln2_beta, eps_ln);
9 [G, ffn] = ffn_forward(D, blk.W_1, blk.b_1, blk.W_2, blk.b_2);
10 Hn = R + G; % second residual
11 c = struct('H', H, 'U', U, 'ln1', ln1, 'att', att, 'F', F, 'R', R, 'D', D, 'ln2', ln2, ...
12          'ffn', ffn, 'G', G);
13 end

```

Code 6: Python: the hand-derived FFN backward map and g' .

```

1 def feed_forward_backward(G_G, D, J, Ga, W_1, W_2):
2     """G_W2 = g(J)^T G_G, G_b2 = 1^T G_G, G_g = G_G W2^T, G_J = G_g * g'(J), G_W1 = D^T G_J, G_b1 = 1^T G_J,
3     G_D = G_J W1^T (sums over every batch and time row)."""
4     d = D.shape[-1]
5     G2 = G_G.reshape(-1, d) # (BT) x d: the rows are independent
6     G_W2 = Ga.T @ G2 # (4d x BT)(BT x d)
7     G_b2 = G2.sum(axis=0, keepdims=True) # 1-T^T G_G, summed over every row
8     G_Ga = G2 @ W_2.T # (BT x d)(d x 4d)
9     G_J = G_Ga * ops.gelu_grad_np(J) # elementwise chain rule through g
10    G_W1 = D.reshape(-1, d).T @ G_J # (d x BT)(BT x 4d)
11    G_b1 = G_J.sum(axis=0, keepdims=True)
12    G_D = (G_J @ W_1.T).reshape(D.shape) # back to B x T x d
13    return {"W_2": G_W2, "b_2": G_b2, "Ga": G_Ga, "J": G_J, "W_1": G_W1, "b_1": G_b1, "D": G_D}

1 def gelu_grad_np(z):
2     """The assignment's derivative: g'(z) = (1 + tanh u)/2 + z/2 (1 - tanh^2 u) a (1 + 3 c z^2)."""
3     t = np.tanh(SQRT_2_OVER_PI * (z + GELU_C * z ** 3))
4     return 0.5 * (1.0 + t) + 0.5 * z * (1.0 - t * t) * SQRT_2_OVER_PI * (1.0 + 3.0 * GELU_C * z * z)

```

Code 7: MATLAB: ffn_backward.m and block_backward.m.

```

1 function g = ffn_backward(G_G, cache, W_1, W_2)
2 %FFN_BACKWARD Hand-derived gradients of the feed-forward network:
3 % G_W2 = g(J)' G_G, G_b2 = 1' G_G, G_g = G_G W2', G_J = G_g .* g'(J),
4 % G_W1 = D' G_J, G_b1 = 1' G_J, G_D = G_J W1' (sums over all BT rows).
5 [B, T, d] = size(G_G);
6 G2 = reshape(G_G, B*T, d); % (BT) x d
7 g.W_2 = cache.Ga' * G2; % 4d x d
8 g.b_2 = sum(G2, 1); % 1 x d
9 g.Ga = G2 * W_2'; % (BT) x 4d
10 g.J = g.Ga .* gelu_grad(cache.J); % elementwise chain rule through g
11 g.W_1 = cache.D2' * g.J; % d x 4d
12 g.b_1 = sum(g.J, 1); % 1 x 4d
13 g.D = reshape(g.J * W_1', B, T, d); % back to B x T x d
14 end

1 function [G_H, g] = block_backward(G_Hn, c, blk)
2 %BLOCK_BACKWARD Backward pass of one block. Each residual sum passes the upstream
3 % gradient along the identity path unchanged and adds the branch's share:
4 % dL/dX = G_Y + (dF/dX)' G_Y. The fields of g follow the order of init_params.
5 gf = ffn_backward(G_Hn, c.ffn, blk.W_1, blk.W_2);
6 [G_R_ln, g_ln2, b_ln2] = layernorm_backward(gf.D, c.ln2, blk.ln2_gamma);
7 G_R = G_Hn + G_R_ln; % H' = R + G
8 ga = attention_backward(G_R, c.att, blk.W_QKV, blk.W_O); % Assignment 1, unchanged
9 [G_H_ln, g_ln1, b_ln1] = layernorm_backward(ga.U, c.ln1, blk.ln1_gamma);
10 G_H = G_R + G_H_ln; % R = H + F
11 g = struct('W_QKV', ga.W_QKV, 'W_O', ga.W_O, 'W_1', gf.W_1, 'b_1', gf.b_1, 'W_2', gf.W_2, ...
12          'b_2', gf.b_2, 'ln1_gamma', g_ln1, 'ln1_beta', b_ln1, 'ln2_gamma', g_ln2, 'ln2_beta', b_ln2);
13 end

```

4. Operation-by-operation explanation

Python (own autodiff engine)	MATLAB	What it computes, shape
$D.\text{reshape}(B * T, d) @ W_1 + b_1$	$\text{reshape}(D, B*T, d) * W_1 + b_1$	$J: (BT \times d)(d \times 4d)$; the $1 \times 4d$ bias row is broadcast (implicit expansion) to every row

Python (own autodiff engine)	MATLAB	What it computes, shape
ops.gelu(J)	gelu_tanh(J)	elementwise g ; $(BT) \times 4d$
(Ga @ W_2 + b_2).reshape(B, T, d)	reshape(Ga * W_2 + b_2, B, T, d)	G : $(BT \times 4d)(4d \times d)$, back to $B \times T \times d$
R = H + F; Hn = R + G	R = H + F; Hn = R + G;	the two residual sums, $B \times T \times d$
Ga.T @ G2; G2.sum(axis=0)	cache.Ga' * G2; sum(G2, 1)	G_{W_2} ($4d \times d$), G_{b_2} ($1 \times d$)
G_Ga * ops.gelu_grad_np(J)	g.Ga .* gelu_grad(cache.J)	$G_J = G_{\mathcal{R}} \odot g'(J)$
G_R = G_Hn + G_R_ln	G_R = G_Hn + G_R_ln;	identity path plus branch: $\partial \mathcal{L} / \partial R$

5. Numerical verification and interpretation

The small case $d = 2$. With $W_1 \in \mathbb{R}^{2 \times 8}$, $b_1 \in \mathbb{R}^{1 \times 8}$, $W_2 \in \mathbb{R}^{8 \times 2}$, $b_2 \in \mathbb{R}^{1 \times 2}$, the self-chosen token row $D = (0.5, -1)$, $b_2 = (0.05, -0.05)$ and

$$W_1 = \begin{bmatrix} 0.4 & -0.3 & 0.2 & 0.1 & -0.5 & 0.6 & 0.0 & 0.3 \\ 0.1 & 0.2 & -0.4 & 0.5 & 0.3 & -0.1 & 0.7 & -0.2 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0.0 & 0.1 & -0.1 & 0.2 & 0.0 & -0.2 & 0.1 & 0.0 \end{bmatrix},$$

the elementwise sums give

$$J = \begin{bmatrix} 0.100 & -0.250 & 0.400 & -0.250 & -0.550 & 0.200 & -0.600 & 0.350 \end{bmatrix},$$

$$g(J) = \begin{bmatrix} 0.0540 & -0.1003 & 0.2622 & -0.1003 & -0.1602 & 0.1159 & -0.1646 & 0.2229 \end{bmatrix},$$

$$G = \begin{bmatrix} 0.01239 & -0.04819 \end{bmatrix}.$$

identical to the vectorized code within 2.8×10^{-17} (Q1.1). Negative entries of J are damped towards zero, not cut off as in a ReLU.

Approximation against the exact GELU (analytical comparison; the implementation uses the approximation):

z	tanh approximation $g(z)$	exact $z\Phi(z)$	difference
-1	-0.158808009	-0.158655254	1.53×10^{-4}
0	0	0	0
+1	+0.841191991	+0.841344746	1.53×10^{-4}

The largest difference on $[-6, 6]$ is 4.7×10^{-4} . The hand-derived g' agrees with the engine's derivative of the composed GELU within 2.2×10^{-16} on 2401 points (Q1.3) and with centred differences within 9.5×10^{-11} .

The identity path, measured. For one block with $T = 4$, $d = 3$, the full 12×12 Jacobians of R and of F with respect to H were formed by centred differences; $\partial R / \partial H - J_F - I$ is at most 1.7×10^{-10} (Q1.4). The token-by-token maxima show the two kinds of mixing:

$$\left[\max \left| \frac{\partial F_i}{\partial H_j} \right| \right]_{ij} = \begin{bmatrix} 2.90 & 0.00 & 0.00 & 0.00 \\ 2.53 & 0.06 & 0.00 & 0.00 \\ 1.32 & 0.05 & 0.04 & 0.00 \\ 0.31 & 0.07 & 0.03 & 0.05 \end{bmatrix},$$

$$\left[\max |\Delta G_i| \text{ after changing } D_j \right]_{ij} = \begin{bmatrix} 0.15 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.26 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.28 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.22 \end{bmatrix}.$$

Attention is lower triangular in tokens (rows i , columns j ; exactly 0 above the diagonal, Q1.6); the FFN is diagonal, exactly 0 off the diagonal (Q1.5). Every term of the FFN backward map matches automatic differentiation within 6.7×10^{-16} (Q1.7).

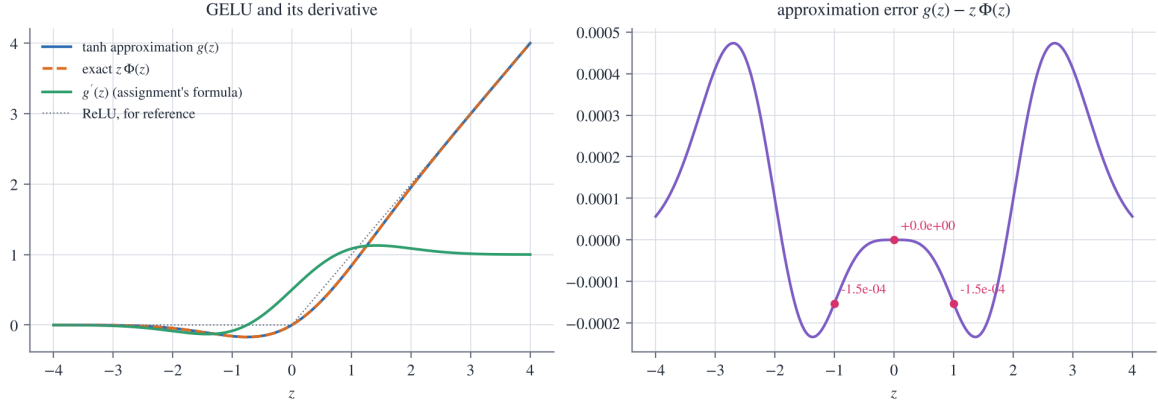


Figure 2: Left: the tanh-GELU, the exact GELU and g' . Right: the approximation error, $\approx 1.5 \times 10^{-4}$ at $z = \pm 1$.

Case B2: random feed-forward and residual (seed 451, $B = 2$, $T = 5$, $d = 4$, $d_{\text{ff}} = 16$)

H , W_1 , W_2 have $\mathcal{N}(0, 0.02^2)$ entries, biases are 0, gains are 1. Per sample, $J \in \mathbb{R}^{5 \times 16}$, $G \in \mathbb{R}^{5 \times 4}$, $H_{\text{next}} = H + G \in \mathbb{R}^{5 \times 4}$ (B2.1, B2.2). For $\mathcal{J} = \sum_{b,i,j} (H_{\text{next}})_{bij}^2$, centred differences on five random entries of W_1 and five of W_2 give:

entry	autodiff	$h = 10^{-3}$	10^{-4}	10^{-5}	10^{-6}
$(W_1)_{4,14}$	-2.491×10^{-3}	4×10^{-10}	4×10^{-12}	1×10^{-14}	2×10^{-13}
$(W_1)_{2,14}$	8.069×10^{-5}	7×10^{-10}	7×10^{-12}	1×10^{-13}	4×10^{-13}
$(W_1)_{4,3}$	2.818×10^{-4}	8×10^{-10}	8×10^{-12}	3×10^{-14}	2×10^{-13}
$(W_1)_{3,16}$	3.049×10^{-3}	2×10^{-9}	2×10^{-11}	1×10^{-13}	1×10^{-13}
$(W_1)_{3,16}$	3.049×10^{-3}	2×10^{-9}	2×10^{-11}	1×10^{-13}	1×10^{-13}
$(W_2)_{15,4}$	4.477×10^{-3}	8×10^{-16}	3×10^{-15}	8×10^{-14}	7×10^{-13}
$(W_2)_{14,2}$	1.377×10^{-3}	5×10^{-16}	4×10^{-15}	7×10^{-14}	7×10^{-14}
$(W_2)_{6,2}$	-2.112×10^{-3}	2×10^{-16}	6×10^{-15}	6×10^{-14}	3×10^{-14}
$(W_2)_{6,3}$	5.347×10^{-3}	4×10^{-16}	5×10^{-15}	1×10^{-14}	7×10^{-14}
$(W_2)_{7,3}$	-3.296×10^{-3}	4×10^{-16}	4×10^{-15}	3×10^{-14}	1×10^{-13}

Because the objective is quadratic in W_2 and smooth in W_1 , truncation error is negligible; with weights of size 0.02 the derivatives are $\sim 10^{-4}$, so the step $h = 10^{-5}$ keeps the perturbation small relative to the weights while the round-off error $\approx \varepsilon_{\text{mach}} \mathcal{J}/h$ stays near 10^{-12} . Worst error at $h = 10^{-5}$: 1.4×10^{-13} (assignment measure, B2.3); scale-free relative error 1.4×10^{-9} (B2.4).

3 Question 2: vocabulary projection and cross-entropy

1. Purpose and assumptions

The last hidden state of each position must become a probability distribution over the V characters, and training needs one scalar that rewards probability on the actual next character. Logits are unrestricted scores; probabilities are their softmax; the loss is the mean negative log-probability of the targets, computed without ever forming a probability that could underflow.

2. Derivation

After the final normalization, the B matrices $\text{LN}_f(H_b^{(L)}) \in \mathbb{R}^{T \times d}$ are flattened into $\bar{H} \in \mathbb{R}^{BT \times d}$ and projected:

$$\underbrace{\bar{H}}_{BT \times d} \underbrace{W_{\text{voc}}}_{d \times V} = \underbrace{Z}_{BT \times V}. \quad (3.1)$$

The targets Y are flattened into $(y_1, \dots, y_{BT})$ by the same rule, so row r of Z is scored against y_r . With $a_r = \max_j Z_{rj}$,

$$\text{LSE}_r = a_r + \log \sum_{j=1}^V e^{Z_{rj} - a_r}, \quad \mathcal{L} = -\frac{1}{BT} \sum_{r=1}^{BT} (Z_{r,y_r} - \text{LSE}_r), \quad p_{rj} = e^{Z_{rj} - \text{LSE}_r}. \quad (3.2)$$

Why the shift changes nothing. For any constant c_r , $\text{softmax}(Z_{r,:} + c_r) = \text{softmax}(Z_{r,:})$ and $\text{LSE}(Z_{r,:} + c_r) - (Z_{r,y_r} + c_r) = \text{LSE}(Z_{r,:}) - Z_{r,y_r}$. Choosing $c_r = -a_r$ makes every exponent ≤ 0 (no overflow) and one exponent equal to 0 (the sum is ≥ 1 , so its logarithm is finite).

Gradient. The softmax Jacobian of one row is $\partial p_i / \partial z_j = p_i(\delta_{ij} - p_j)$. For one row, $\ell = \text{LSE}(z) - z_y$, so $\partial \ell / \partial z_j = p_j - \mathbb{1}[j = y]$. Averaging over the BT rows, with E the one-hot targets,

$$\frac{\partial \mathcal{L}}{\partial Z} = \frac{P - E}{BT} \in \mathbb{R}^{BT \times V}, \quad \frac{\partial \mathcal{L}}{\partial W_{\text{voc}}} = \bar{H}^\top \frac{P - E}{BT} \in \mathbb{R}^{d \times V}, \quad \frac{\partial \mathcal{L}}{\partial \bar{H}} = \frac{P - E}{BT} W_{\text{voc}}^\top \in \mathbb{R}^{BT \times d}. \quad (3.3)$$

Each row of $P - E$ sums to $\sum_j p_{rj} - 1 = 0$. **Uniform logits.** If $Z = 0$, then $p_{rj} = 1/V$ and every term is $-\log(1/V)$, so $\mathcal{L} = \log V = 4.174387$ for $V = 65$: the loss of a model that knows nothing. A target probability of 0.7 contributes $-\log 0.7 \approx 0.357$.

3. Code

Code 8: Python: the complete forward pass, the cross-entropy and the training objective (`gpt_a2/model.py`).

```
1 def gpt_forward(X, P, L=2, eps=1e-5, neg=-1e9):
2     """Indices X (B x T) -> logits Z ((BT) x V) through embeddings, L blocks, the final LN and W_voc."""
3     B, T = np.asarray(X).shape
4     H, O = A1.embed(X, P["W_E"], P["W_P"]) # (B,T,d): token rows plus position rows
5     blocks = []
6     for l in range(1, L + 1):
7         H, c = block(H, P, l, eps, neg)
8         blocks.append(c)
9     Hf, lnf = A1.layer_norm(H, P["lnf_gamma"], P["lnf_beta"], eps)
10    d = Hf.shape[-1]
11    Hbar = Hf.reshape(B * T, d) # row r = b*T + t (row-major flattening)
12    Z = Hbar @ P["W_voc"] # ((BT) x d) @ (d x V) -> (BT) x V logits
13    return Z, {"O": O, "blocks": blocks, "HL": H, "lnf": lnf, "Hbar": Hbar, "B": B, "T": T}
14
15 def cross_entropy(Z, y):
16     """Mean next-token negative log-likelihood with a stable log-sum-exp and vectorized gathering.
17
18     LSE_r = a_r + log sum_j exp(Z_rj - a_r), a_r = max_j Z_rj (a constant shift),
19     L = -(1/R) sum_r (Z_{r,y_r} - LSE_r).
20     """
21     R = Z.shape[0]
22     a = Z.max(axis=1, keepdims=True) # constant: softmax(Z - a) = softmax(Z), so no gradient
23     lse = a + ops.log((Z - a).exp().sum(axis=1, keepdims=True)) # (R,1); every exponent is <= 0
24     return (lse.reshape(R) - ops.pick(Z, y)).mean()
25
26 def loss_fn(X, Y, P, L=2, eps=1e-5):
27     """The training objective on one batch: L(theta) = cross_entropy(gpt_forward(X, flatten(Y)))."""
28     Z, cache = gpt_forward(X, P, L, eps)
29     return cross_entropy(Z, np.asarray(Y).reshape(-1)), Z, cache
```

Code 9: Python: $\partial \mathcal{L} / \partial Z = (P - E) / R$.

```
1 def cross_entropy_backward(Z, y):
2     """dL/dZ = (P - E) / R, E the one-hot targets: every row sums to zero."""
3     R = Z.shape[0]
4     G = softmax_np(Z)
5     G[np.arange(R), y] -= 1.0
6     return G / R
```

Code 10: MATLAB: `gpt_forward.m` and `cross_entropy.m`.

```
1 function [Z, c] = gpt_forward(X, P, eps_ln)
2 %GPT_FORWARD Indices X (B x T, 1-based) -> logits Z ((BT) x V).
3 % Embeddings, numel(P.blocks) Transformer blocks, the final LayerNorm and W_voc.
4 % The B x T x d hidden states are flattened column-major, so row r = b + (t-1) B;
```

```

5 % the targets must be flattened the same way, as Y(:).
6 [H, O] = embed_forward(X, P.W_E, P.W_P); % B x T x d (Assignment 1)
7 nb = numel(P.blocks);
8 blocks = cell(1, nb);
9 for l = 1:nb
10     [H, blocks{l}] = block_forward(H, P.blocks(l), eps_ln);
11 end
12 [Hf, lnf] = layernorm_forward(H, P.lnf_gamma, P.lnf_beta, eps_ln);
13 [B, T, d] = size(Hf);
14 Hbar = reshape(Hf, B*T, d); % (BT) x d, column-major rows
15 Z = Hbar * P.W_voc; % (BT x d)(d x V)
16 c = struct('O', O, 'HL', H, 'lnf', lnf, 'Hbar', Hbar, 'B', B, 'T', T);
17 c.blocks = blocks; % a cell inside struct() would build a struct array
18 end

1 function [L, P] = cross_entropy(Z, y)
2 %CROSS_ENTROPY Mean next-token negative log-likelihood with a stable log-sum-exp:
3 % LSE_r = a_r + log sum_j exp(Z_rj - a_r), a_r = max_j Z_rj,
4 % L = -(1/R) sum_r (Z_{r,y_r} - LSE_r). y holds 1-based labels, one per row of Z.
5 % Vectorized gathering with sub2ind (no loop over tokens). Works on dlarray Z.
6 R = size(Z, 1);
7 a = max(Z, [], 2); % R x 1 row maxima: every exponent is <= 0
8 lse = a + log(sum(exp(Z - a), 2)); % R x 1
9 idx = sub2ind(size(Z), (1:R)', y(:)); % linear index of Z(r, y_r)
10 L = mean(lse - Z(idx));
11 if nargin > 1
12     P = exp(Z - lse); % softmax rows
13 end
14 end

```

4. Operation-by-operation explanation

Python (own autodiff engine)	MATLAB	What it computes, shape
Hf.reshape(B * T, d)	reshape(Hf, B*T, d)	$\tilde{H}$; Python row $r = bT + t$, MATLAB row $r = b + (t - 1)B$
Hbar @ P["W_voc"]	Hbar * P.W_voc	$Z: (BT \times d)(d \times V)$
Z.max(axis=1, keepdims=True)	max(Z, [], 2)	$a_r, BT \times 1$; a constant shift
a + ops.log((Z - a).exp().sum(axis=1, keepdims=True))	a + log(sum(exp(Z - a), 2))	LSE_r ; every exponent ≤ 0
ops.pick(Z, y)	Z(sub2ind(size(Z), (1:R)', y(:)))	Z_{r,y_r} for all rows at once (no loop)
(lse - picked).mean()	mean(lse - Z(idx))	$\mathcal{L}$, the mean over BT rows
G[np.arange(R), y] -= 1; G / R	G(idx) = G(idx) - 1; G / R	$(P - E)/(BT)$

5. Numerical verification and interpretation

The 2×3 example. For $Z = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$ and one-based labels $y = (1, 2)$: the row maxima are $a = (2, 1)$, $LSE_1 = \log(e^2 + 1 + e^{-1}) = 2.169846$, $LSE_2 = \log(2 + e) = 1.551445$, the negative log-likelihoods are $LSE_1 - 2 = 0.169846$ and $LSE_2 - 1 = 0.551445$, and

$$\mathcal{L} = \frac{1}{2} [\log(e^2 + 1 + e^{-1}) - 2 + \log(2 + e) - 1] = 0.360645366744 \quad (\text{closed form } 0.360645366744),$$

$$\frac{\partial \mathcal{L}}{\partial Z} = \frac{P - E}{2} = \begin{bmatrix} -0.078103 & 0.057098 & 0.021005 \\ 0.105971 & -0.211942 & 0.105971 \end{bmatrix}.$$

Both rows of the gradient sum to zero and agree with autodiff (Q2.2). Adding $(+1000, -1000)$ to the two rows leaves the stable loss at 0.360645366744; without the shift the computation returns nan (Q2.3).

The ordering requirement. For the label matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, Python’s row-major flattening gives (1, 2, 3, 4, 5, 6), MATLAB’s $Y(:)$ gives (1, 4, 2, 5, 3, 6). On a random $2 \times 3 \times 4$ batch, flattening hidden states and labels with the same rule gives the same mean loss in either language (row-major 1.950721077065, column-major 1.950721077065), and a joint random permutation of rows and labels does too (1.950721077065; Q2.4). Pairing row-major hidden states with column-major labels instead gives 2.907759547899: wrong targets for four of six rows (Q2.5). The rule used everywhere in this submission is therefore fixed per language and applied to both arrays in the same line of code.

Case B1: fixed logit matrix and exact cross-entropy

For $Z = \begin{bmatrix} 0 & 0 & 0 \\ \log 2 & 0 & 0 \\ 0 & \log 3 & 0 \end{bmatrix}$ and $y = (1, 1, 2)$,

$$P = \begin{bmatrix} 0.333333 & 0.333333 & 0.333333 \\ 0.500000 & 0.250000 & 0.250000 \\ 0.200000 & 0.600000 & 0.200000 \end{bmatrix},$$

$$\frac{\partial \mathcal{L}}{\partial Z} = \frac{P - E}{3} = \begin{bmatrix} -0.222222 & 0.111111 & 0.111111 \\ -0.166667 & 0.083333 & 0.083333 \\ 0.066667 & -0.133333 & 0.066667 \end{bmatrix}.$$

The correct-class probabilities are $1/3$, $1/2$ and $3/5$ (B1.3), so $\mathcal{L} = \frac{1}{3}(\log 3 + \log 2 + \log \frac{5}{3}) = 0.767528364331$; the program measures 0.767528364331 (B1.1, error 0) and the gradient error is 2.8×10^{-17} (B1.2). Adding $(5, -3, 100)$ to the three rows changes the loss by 1.8×10^{-15} : a constant shift of a row changes neither its probabilities nor its loss.

4 Question 3: automatic differentiation and Adam

1. Purpose and assumptions

Training needs $\nabla_{\theta} \mathcal{L}$ for all 25 tensors at every step. Automatic differentiation evaluates the chain rule through the computational graph the forward code actually builds. Adam then turns each gradient into an update whose size is normalized per coordinate. Computing the gradient and applying the update are separate steps: the first only reads the parameters, the second only changes them.

2. Derivation

The gradient flow.

$$\mathcal{L} \longrightarrow Z \longrightarrow \bar{H} \longrightarrow (H^{(L)}, \dots, H^{(0)}) \longrightarrow (W_E, W_P). \quad (4.1)$$

The edge $\mathcal{L} \rightarrow Z$ carries $(P - E)/(BT)$; $Z \rightarrow \bar{H}$ meets W_{voc} (gradient $\bar{H}^{\top} \partial \mathcal{L} / \partial Z$, $d \times V$); $\bar{H} \rightarrow H^{(L)}$ meets γ_f, β_f ; each block ℓ meets $W_1, b_1, W_2, b_2, \gamma_2, \beta_2$ on its FFN branch and $W_{QKV}, W_O, \gamma_1, \beta_1$ on its attention branch, while the residual identity carries the gradient past both; $H^{(0)} \rightarrow (W_E, W_P)$ gives $\partial \mathcal{L} / \partial W_E = \sum_b O_b^{\top} G_{H_b}$ and $\partial \mathcal{L} / \partial W_P[t, :] = \sum_b G_{H_b}[t, :]$. Only the rows of W_E whose characters occur in the batch receive a gradient; on the real batch below, 57 of the $V = 65$ rows do.

Why the embeddings must stay in the graph. If W_E entered the forward pass as a numerical copy without history, the chain would stop at $H^{(0)}$: no gradient would reach W_E , Adam would never move it, and the model could not learn which character is which (Q3.6).

Adam. For the gradient g_t at step t ,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t, \quad (4.2)$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \quad \theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon_A}}. \quad (4.3)$$

Because $m_0 = v_0 = 0$, the raw moments are biased towards zero in early steps; dividing by $1 - \beta^t$ removes that bias. At $t = 1$, $\hat{m}_1 = g_1$ and $\hat{v}_1 = g_1^2$, so $\theta_1 - \theta_0 = -\eta g_1 / (|g_1| + \epsilon_A) \approx -\eta \text{sign}(g_1)$.

Numerical exercise ($g_1 = 0.2$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.001$, $\epsilon_A = 10^{-8}$): $m_1 = 0.1 \cdot 0.2 = 0.02$, $v_1 = 0.001 \cdot 0.04 = 4.00 \times 10^{-5}$, $\hat{m}_1 = 0.2$, $\hat{v}_1 = 0.04$, and the first update is $-0.001 \cdot 0.2 / (0.2 + 10^{-8}) = -0.000999999950$ (Q3.1).

3. Code

Code 11: Python: the hand-derived backward pass through a block and through the whole GPT (`gpt_a2/model.py`).

```

1 def block_backward(G_Hn, c, P, l):
2     """Backward through one block. Each residual sum sends the upstream gradient along the identity path
3     unchanged and adds the branch's contribution: dL/dX = G_Y + (dF/dX)^T G_Y."""
4     k = f"block{l}."
5     val = lambda t: t.data if isinstance(t, Tensor) else t
6     g = {}
7     # H' = R + G -> G_R = G_H' (identity path) + contribution through the feed-forward branch
8     ffn = feed_forward_backward(G_Hn, val(c["D"]), val(c["J"]), val(c["Ga"]), val(P[k + "W_1"]), val(P[k + "W_2"]))
9     G_R_ln, g_ln2, b_ln2 = A1.layer_norm_backward(ffn["D"], val(c["ln2"]["xhat"]), val(c["ln2"]["s"]), val(P[k + "ln2_gamma"]))
10    G_R = G_Hn + G_R_ln
11    # R = H + F -> G_H = G_R (identity path) + contribution through attention and LN1
12    att = A1.attention_backward(G_R, val(c["U"]), val(c["Q"]), val(c["K"]), val(c["V"]), val(c["A"]), val(c["Z"]),
13                               val(P[k + "W_QKV"]), val(P[k + "W_0"]))
14    G_H_ln, g_ln1, b_ln1 = A1.layer_norm_backward(att["U"], val(c["ln1"]["xhat"]), val(c["ln1"]["s"]), val(P[k + "ln1_gamma"]))
15    G_H = G_R + G_H_ln
16    g.update({k + "W_1": ffn["W_1"], k + "b_1": ffn["b_1"], k + "W_2": ffn["W_2"], k + "b_2": ffn["b_2"],
17             k + "ln2_gamma": g_ln2, k + "ln2_beta": b_ln2, k + "W_QKV": att["W_QKV"], k + "W_0": att["W_0"],
18             k + "ln1_gamma": g_ln1, k + "ln1_beta": b_ln1})
19    return G_H, g, {"R": G_R, "F": G_R, "D": ffn["D"], "J": ffn["J"], "U": att["U"]}
20
21 def gpt_backward(G_Z, cache, P, L=2):
22     """Every parameter gradient from dL/dZ: head -> final LN -> blocks L..1 -> embeddings."""
23     val = lambda t: t.data if isinstance(t, Tensor) else t
24     B, T = cache["B"], cache["T"]
25     Hbar = val(cache["Hbar"])
26     d = Hbar.shape[1]
27     g = {"W_voc": Hbar.T @ G_Z}
28     G_Hf = (G_Z @ val(P["W_voc"])).T.reshape(B, T, d)
29     G_H, g["lnf_gamma"], g["lnf_beta"] = A1.layer_norm_backward(G_Hf, val(cache["lnf"]["xhat"]), val(cache["lnf"]["s"]),
30                                                                val(P["lnf_gamma"]))
31     inner = {}
32     for l in range(L, 0, -1):
33         G_H, gb, inner[l] = block_backward(G_H, cache["blocks"][l - 1], P, l)
34         g.update(gb)
35     g["W_E"], g["W_P"] = A1.embedding_backward(G_H, cache["O"], val(P["W_P"]).shape[0])
36     g["H0"] = G_H
37     return g, inner

```

Code 12: Python: Adam (`gpt_a2/optim.py`).

```

1 class Adam:
2     def __init__(self, params, lr=1e-3, beta1=0.9, beta2=0.999, eps=1e-8):
3         self.lr, self.beta1, self.beta2, self.eps = lr, beta1, beta2, eps
4         self.t = 0
5         self.m = {k: np.zeros_like(v) for k, v in params.items()} # moments start at zero
6         self.v = {k: np.zeros_like(v) for k, v in params.items()}
7
8     def step(self, params, grads):

```

```

9      """Update every array of `params` in place from `grads` (same keys); returns the applied steps."""
10     self.t += 1
11     b1, b2 = self.beta1, self.beta2
12     c1, c2 = 1.0 - b1 ** self.t, 1.0 - b2 ** self.t
13     applied = {}
14     for k, theta in params.items():
15         g = grads[k]
16         self.m[k] = b1 * self.m[k] + (1.0 - b1) * g
17         self.v[k] = b2 * self.v[k] + (1.0 - b2) * (g * g)
18         m_hat, v_hat = self.m[k] / c1, self.v[k] / c2
19         delta = -self.lr * m_hat / (np.sqrt(v_hat) + self.eps)
20         theta += delta # in place: the model sees the new value
21         applied[k] = delta
22     return applied
23
24     def fingerprint(self):
25         """A cheap digest of the optimizer state (used to prove that validation never changes it)."""
26         return (self.t, float(sum(np.abs(m).sum() for m in self.m.values())), float(sum(v.sum() for v in self.v.values())))

```

Code 13: MATLAB: gpt_backward.m, the traced objective dl_gpt_loss.m, and adam_step.m.

```

1 function g = gpt_backward(G_Z, c, P)
2 %GPT_BACKWARD Every parameter gradient from dL/dZ: head -> final LayerNorm ->
3 % blocks L..1 -> embeddings. g has the same layout as P.
4 d = size(c.Hbar, 2);
5 G_Wvoc = c.Hbar' * G_Z; % (d x BT)(BT x V) = d x V
6 G_Hf = reshape(G_Z * P.W_voc', c.B, c.T, d); % undo the flattening with the same rule
7 [G_H, G_gf, G_bf] = layernorm_backward(G_Hf, c.lnf, P.lnf_gamma);
8 nb = numel(P.blocks);
9 gb = cell(1, nb);
10 for l = nb:-1:1
11     [G_H, gb{l}] = block_backward(G_H, c.blocks{l}, P.blocks{l});
12 end
13 [G_WE, G_WP] = embed_backward(G_H, c.O, size(P.W_P, 1));
14 g = struct('W_E', G_WE, 'W_P', G_WP, 'blocks', [gb{:}], 'lnf_gamma', G_gf, ...
15         'lnf_beta', G_bf, 'W_voc', G_Wvoc);
16 end

```

```

1 function [J, G] = dl_gpt_loss(F, X, y, L, eps_ln)
2 %DL_GPT_LOSS Traced training objective for dlfeval: the flat parameter struct F holds
3 % darray copies of every tensor, so dgradient differentiates the whole GPT
4 % (embedding gathers, both LayerNorms, masked softmax, GELU, residual sums, head).
5 % [J, G] = dlfeval(@dl_gpt_loss, F, X, y, L, eps_ln)
6 P = unpack_params(F, L);
7 Z = gpt_forward(X, P, eps_ln);
8 J = cross_entropy(Z, y);
9 G = dgradient(J, F);
10 end

```

```

1 function [F, state, delta] = adam_step(F, G, state, lr, beta1, beta2, eps_a)
2 %ADAM_STEP One Adam update of every field of the flat parameter struct F:
3 % m_t = b1 m + (1-b1) g, v_t = b2 v + (1-b2) g.*g,
4 % m_hat = m_t / (1 - b1^t), v_hat = v_t / (1 - b2^t), theta = theta - lr m_hat ./ (sqrt(v_hat) + eps).
5 % No clipping, no weight decay. delta returns the applied steps.
6 state.t = state.t + 1;
7 c1 = 1 - beta1^state.t; % bias corrections (moments start at zero)
8 c2 = 1 - beta2^state.t;
9 fn = fieldnames(F);
10 delta = struct();
11 for k = 1:numel(fn)
12     n = fn{k};
13     g = G.(n);
14     state.m.(n) = beta1 * state.m.(n) + (1 - beta1) * g;
15     state.v.(n) = beta2 * state.v.(n) + (1 - beta2) * (g .* g);
16     step = -lr * (state.m.(n) / c1) ./ (sqrt(state.v.(n) / c2) + eps_a);
17     F.(n) = F.(n) + step;
18     delta.(n) = step;
19 end
20 end

```

4. Operation-by-operation explanation

Python (own autodiff engine)	MATLAB	What it computes, shape
<code>L_t.backward()</code>	<code>dlfeval(@dl_gpt_loss, ...)</code>	every $\partial\mathcal{L}/\partial\theta$ by reverse-mode autodiff
<code>Hbar.T @ G_Z</code>	<code>c.Hbar' * G_Z</code>	$\partial\mathcal{L}/\partial W_{\text{voc}}$, $(d \times BT)(BT \times V) = d \times V$
<code>(G_Z @ W_voc.T).reshape(B, T, d)</code>	<code>reshape(G_Z * P.W_voc', c.B, c.T, d)</code>	undo the flattening with the same rule
<code>for l in range(L, 0, -1)</code>	<code>for l = nb:-1:1</code>	blocks in reverse order
<code>A1.embedding_backward(G_H, 0, T_max)</code>	<code>embed_backward(G_H, c.0, ...)</code>	$\sum_b O_b^\top G_{H_b}$ and $\sum_b G_{H_b}$
<code>self.m[k] = b1 * self.m[k] + (1 - b1) * g</code>	<code>state.m.(n) = beta1 * state.m.(n) + ...</code>	first moment
<code>m_hat / (np.sqrt(v_hat) + self.eps)</code>	<code>(m / c1) ./ (sqrt(v / c2) + eps_a)</code>	bias-corrected, per-coordinate step
<code>theta += delta</code>	<code>F.(n) = F.(n) + step</code>	the update (in place in Python)

5. Numerical verification and interpretation

On a real batch ($B = 32$, $T = 64$) at the Assignment 1 initialization (loss 4.196727), the hand-derived chain rule and the autodiff engine agree for every one of the 25 tensors to 1.4×10^{-16} (Q3.2); none of the tensors has an all-zero gradient, so every parameter lies on a path from $\mathcal{L}$ (Q3.3); $\partial\mathcal{L}/\partial W_{\text{voc}} = \bar{H}^\top \partial\mathcal{L}/\partial Z$ has shape $d \times V = 64 \times 65$ and matches (Q3.4); the gradients of W_E and W_P are finite and non-zero (Q3.5). With W_E replaced by a detached copy, no gradient reaches W_E while W_P still receives one (Q3.6).

Case B3, part 1: one Adam step on a random 2×3 matrix (seed 451)

$$\theta_0 = \begin{bmatrix} 0.547402 & -0.970805 & 0.202290 \\ 0.250126 & -0.517967 & -0.561857 \end{bmatrix}, \quad g = \begin{bmatrix} 0.50 & -0.20 & 0.01 \\ -1.50 & 0.30 & 2.00 \end{bmatrix},$$

$$m_1 = \begin{bmatrix} 0.0500 & -0.0200 & 0.0010 \\ -0.1500 & 0.0300 & 0.2000 \end{bmatrix}, \quad v_1 = \begin{bmatrix} 0.0002500 & 0.0000400 & 0.0000001 \\ 0.0022500 & 0.0000900 & 0.0040000 \end{bmatrix},$$

$$\theta_1 = \begin{bmatrix} 0.546402 & -0.969805 & 0.201290 \\ 0.251126 & -0.518967 & -0.562857 \end{bmatrix}.$$

The optimizer and the matrix equations agree to 1.3×10^{-23} (B3.1). Every entry moved by almost exactly $-\eta \text{sign}(g)$: at $t = 1$ Adam ignores the size of the gradient and keeps only its direction.

5 Required Benchmark B: deterministic logits, loss, gradients and Adam

Setup and prediction

$B = 1$, $T = 2$, $d = 2$, $V = 3$. Residual test: $H = I_2$, attention output $F = 0$, $W_1 = W_2 = 0$, $b_1 = b_2 = 0$, so $R = H$, the FFN output is 0 and $H' = H$. Head test: $\bar{H} = I_2$ supplied directly, $W_{\text{voc}} = 0_{2 \times 3}$, one-based targets $(y_1, y_2) = (2, 3)$.

Measured: $R = H$, $G = 0$, $H' = H$ exactly (BB.1). The logits are $Z = 0$, every probability is $1/3$, and

$$P = \begin{bmatrix} 0.33333333 & 0.33333333 & 0.33333333 \\ 0.33333333 & 0.33333333 & 0.33333333 \end{bmatrix}, \quad \mathcal{L} = 1.098612288668 = \log 3,$$

$$\frac{\partial\mathcal{L}}{\partial Z} = \frac{P - E}{2} = \begin{bmatrix} \frac{1}{6} & -\frac{1}{3} & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{3} \end{bmatrix}, \quad \frac{\partial\mathcal{L}}{\partial W_{\text{voc}}} = \bar{H}^\top \frac{P - E}{2} = \begin{bmatrix} \frac{1}{6} & -\frac{1}{3} & \frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{3} \end{bmatrix},$$

all within 5.6×10^{-17} of the displayed values (BB.2, BB.3). The first Adam step gives

$$W_{\text{voc}}^{\text{new}} = \begin{bmatrix} -0.001000000 & +0.001000000 & -0.001000000 \\ -0.001000000 & -0.001000000 & +0.001000000 \end{bmatrix},$$

equal to $-\eta g/(|g| + \epsilon_A)$ within 0 and to the rounded display ± 0.001 within 6.0×10^{-11} (BB.4, BB.5).

Why zero weights give uniform probabilities but a non-zero gradient. $W_{\text{voc}} = 0$ makes every logit 0, hence $P = \frac{1}{3}\mathbf{11}^\top$ whatever $\bar{H}$ is. The gradient $\bar{H}^\top(P - E)/2$, however, depends on $\bar{H}$ and on the targets: $P - E$ is non-zero because the targets are one-hot and the prediction is not. The gradient points from the uniform guess towards the targets, which is exactly what the first update starts to do.

6 Question 4: training schedule, validation and gradient checks

1. Purpose and assumptions

Training repeats: draw fresh windows from the training part, compute the loss and its gradient, update. Validation measures the same loss on held-out text, never updating anything. The step budget is defined by the assignment as 50 “epochs” of S_{epoch} steps.

2. Derivation

One step consumes B windows of T predictions, i.e. BT predicted characters. The step budget is

$$S_{\text{epoch}} = \left\lfloor \frac{N_{\text{tr}}}{BT} \right\rfloor = \left\lfloor \frac{1\,003\,854}{2048} \right\rfloor = 490, \quad S_{\text{total}} = 50 S_{\text{epoch}} = 24\,500. \quad (6.1)$$

It is a *budget*, not a pass over the data: the $S_{\text{epoch}}B$ starts of one epoch are drawn independently and uniformly from $M = N_{\text{tr}} - T$ possibilities, so some characters are predicted several times and others not at all. A given character is a target of the T windows that start in the T positions before it, so it is never predicted in an epoch with probability $(1 - T/M)^{S_{\text{epoch}}B} \approx \exp(-S_{\text{epoch}}BT/M) \approx e^{-1} \approx 37\%$. Over 50 epochs that falls to e^{-50} : every character is seen many times, in many different windows.

Training loss vs validation loss. The training loss of step t is computed *before* its update, on the batch that produces the gradient; it is the quantity being minimized. The validation loss at step t is computed *after* the update, on 8 fresh windows from the held-out part, with constant parameters: no graph, no gradient, no change to the Adam moments. Eight windows are only 512 correlated predictions from a random place in the held-out text, so each validation value is a noisy estimate: over the last 100 evaluations of the run, the per-evaluation noise, estimated from the differences of consecutive values (the model barely changes in 50 steps), has a standard deviation of 0.135 nats. The final numbers therefore average over 512 fixed windows.

Gradient check. For one scalar θ_k of each kind of tensor and an identical fixed batch,

$$g_k^{\text{FD}} = \frac{\mathcal{L}(\theta + \delta e_k) - \mathcal{L}(\theta - \delta e_k)}{2\delta}, \quad \rho_k = \frac{|g_k^{\text{AD}} - g_k^{\text{FD}}|}{\max(1, |g_k^{\text{AD}}|, |g_k^{\text{FD}}|)}. \quad (6.2)$$

The truncation error is $\frac{\delta^2}{6}|\mathcal{L}'''|$; the round-off error is $\approx \varepsilon_{\text{mach}}|\mathcal{L}|/\delta$ with $\varepsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$ and $|\mathcal{L}| \approx 4$. They balance near $\delta \approx (3\varepsilon_{\text{mach}}|\mathcal{L}|/|\mathcal{L}'''|)^{1/3} \approx 10^{-5}$, which is the step used.

3. Code

Code 14: Python: the step budget and the evaluation helpers (gpt_a2/train.py).

```
1 def schedule(n_train, B, T, epochs=50, s_max=0):
2     s_epoch = n_train // (B * T)
3     s_total = epochs * s_epoch if s_max <= 0 else min(epochs * s_epoch, s_max)
4     return s_epoch, s_total
```

```

5
6 def evaluate(P, X, Y, L=2):
7     """Loss on one batch with constant parameters (no graph is recorded)."""
8     Z, _ = gpt_forward(X, as_constants(P), L)
9     return float(cross_entropy(Z, np.asarray(Y).reshape(-1)).data)
10
11 def fixed_windows(n, T, count):
12     """A documented, deterministic set of window starts spread evenly over a split of length n."""
13     return np.unique(np.linspace(0, n - T - 1, count).round().astype(np.int64))
14
15 def evaluate_fixed(P, ids, T, starts, L=2, chunk=64):
16     """Mean loss over fixed windows (equal weight per predicted character): a low-noise estimate."""
17     total, count = 0.0, 0
18     for i in range(0, len(starts), chunk):
19         r = starts[i:i + chunk]
20         X = ids[r[:, None] + np.arange(T)]
21         Y = ids[r[:, None] + np.arange(T) + 1]
22         total += evaluate(P, X, Y, L) * X.size
23         count += X.size
24     return total / count

```

Code 15: Python: the training loop.

```

1 def train(P, train_ids, val_ids, cfg, rng_train, rng_val, rng_sample=None, vocab=None, log=print,
2           preview_every=0, preview_len=80, prompt="ROMEO: "):
3     """Optimize the Tensors of P in place. Returns the full record of the run."""
4     B, T, L = cfg["B"], cfg["T"], cfg["L"]
5     s_epoch, s_total = schedule(len(train_ids), B, T, cfg.get("epochs", 50), cfg.get("s_max", 0))
6     arrays = {k: p.data for k, p in P.items()} # Adam updates these arrays in place
7     opt = Adam(arrays, cfg["lr"], cfg["beta1"], cfg["beta2"], cfg["eps_adam"])
8     B_val = min(8, B)
9     train_curve = np.empty(s_total)
10    evals, previews, norms = [], [], []
11    val_ok = True
12    t0 = time.perf_counter()
13    for s in range(1, s_total + 1):
14        X, Y, _ = get_batch(train_ids, B, T, rng_train) # fresh random windows, training part only
15        for p in P.values():
16            p.grad = None
17        L_t, _, _ = loss_fn(X, Y, P, L)
18        L_t.backward() # reverse-mode autodiff through the whole
19        GPT
20        g = gradients(P)
21        if not all(np.isfinite(v)) for v in g.values():
22            raise FloatingPointError(f"non-finite gradient at step {s}")
23        train_curve[s - 1] = float(L_t.data) # training loss, measured before the update
24        opt.step(arrays, g)
25        if s == 1 or s % cfg["eval_every"] == 0:
26            before = opt.fingerprint()
27            Xv, Yv, rv = get_batch(val_ids, B_val, T, rng_val) # held-out part only
28            L_v = evaluate(P, Xv, Yv, L) # after the update; no backward, no step
29            val_ok &= opt.fingerprint() == before and int(rv.min()) >= 0 and int(rv.max()) <= len(val_ids) -
30            T - 1
31            gn = {k: float(np.linalg.norm(v)) for k, v in g.items()}
32            evals.append((s, train_curve[s - 1], L_v))
33            norms.append((s, gn))
34            if s == 1 or s % (cfg["eval_every"] * 20) == 0 or s == s_total:
35                rate = s / (time.perf_counter() - t0)
36                log(f"step {s:6d}/{s_total} epoch {s / s_epoch:5.2f} train {train_curve[s - 1]:.4f} "
37                    f"val {L_v:.4f} ({rate:.1f} steps/s)")
38            if preview_every and vocab and (s % preview_every == 0 or s == s_total):
39                text = generate(P, prompt, vocab["stoi"], vocab["itos"], preview_len, T, rng_sample, L)
40                previews.append((s, text))
41                log(f" preview @ {s}: {text!r}")
42    return {"s_epoch": s_epoch, "s_total": s_total, "train_curve": train_curve, "evals": evals,
43            "previews": previews, "norms": norms, "val_only_heldout_and_no_update": bool(val_ok),
44            "seconds": time.perf_counter() - t0, "adam_steps": opt.t}

```

Code 16: MATLAB: train_gpt.m and sample_batch.m.

```

1 function [F, rec] = train_gpt(F, train, val, cfg, streams, vocab)
2 %TRAIN_GPT The optimization loop: fresh random training windows, cross-entropy,
3 % gradients (dlgradient or the hand-derived chain rule, cfg.grad), one Adam step.
4 % At step 1 and every cfg.eval_every steps a fresh batch of min(8, B) windows is
5 % drawn from the validation part and evaluated AFTER the update, without any

```

```

6 % gradient or optimizer call; the Adam state is checked to be unchanged.
7 %
8 % S_epoch = floor(N_train / (B T)); S_total = cfg.epochs S_epoch, capped by cfg.s_max > 0.
9 s_epoch = floor(numel(train) / (cfg.B * cfg.T));
10 s_total = cfg.epochs * s_epoch;
11 if cfg.s_max > 0
12     s_total = min(s_total, cfg.s_max);
13 end
14 state = adam_init(F);
15 train_curve = zeros(s_total, 1);
16 evals = zeros(0, 3);
17 norms = zeros(0, 5);
18 previews = struct('step', {}, 'text', {});
19 val_ok = true;
20 B_val = min(8, cfg.B);
21 t0 = tic;
22 for s = 1:s_total
23     [X, Y] = sample_batch(train, cfg.B, cfg.T, streams.train);
24     [Ls, G] = loss_and_grad(F, X, Y(:), cfg.L, cfg.eps, cfg.grad);
25     gvals = struct2cell(G);
26     if ~all(cellfun(@(x) all(isfinite(x), 'all'), gvals))
27         error('train_gpt:nonfinite', 'non-finite gradient at step %d', s);
28     end
29     train_curve(s) = Ls; % training loss, measured before the update
30     [F, state] = adam_step(F, G, state, cfg.lr, cfg.beta1, cfg.beta2, cfg.eps_adam);
31     if s == 1 || mod(s, cfg.eval_every) == 0
32         before = adam_fingerprint(state);
33         [Xv, Yv, rv] = sample_batch(val, B_val, cfg.T, streams.val); % held-out part only
34         Lv = cross_entropy(gpt_forward(Xv, unpack_params(F, cfg.L), cfg.eps), Yv(:));
35         val_ok = val_ok && isequal(before, adam_fingerprint(state)) && min(rv) >= 1 && max(rv) <= numel(val)
36         - cfg.T;
37         evals(end + 1, :) = [s, Ls, Lv]; %ok<AGROW>
38         norms(end + 1, :) = [s, group_norms(G)]; %ok<AGROW>
39         if s == 1 || mod(s, 20 * cfg.eval_every) == 0 || s == s_total
40             fprintf('step %6d/%d epoch %5.2f train %.4f val %.4f (%.1f steps/s)\n', ...
41                     s, s_total, s / s_epoch, Ls, Lv, s / toc(t0));
42         end
43     end
44     if cfg.preview_every > 0 && (mod(s, cfg.preview_every) == 0 || s == s_total)
45         txt = generate_text(unpack_params(F, cfg.L), cfg.prompt, vocab, cfg.preview_len, cfg.T, cfg.eps,
46                             streams.preview);
47         previews(end + 1) = struct('step', s, 'text', txt); %ok<AGROW>
48         fprintf(' preview @ %d: %s\n', s, strrep(txt, newline, '\n'));
49     end
50 end
51 rec = struct('s_epoch', s_epoch, 's_total', s_total, 'train_curve', train_curve, 'evals', evals, ...
52             'norms', norms, 'previews', previews, 'val_ok', val_ok, 'seconds', toc(t0), 'adam_steps', state.
53             t);
54 end
55
56 function fp = adam_fingerprint(state)
57 % a cheap digest of the optimizer state: step count and the sums of both moments
58 m = struct2cell(state.m);
59 v = struct2cell(state.v);
60 fp = [state.t, sum(cellfun(@(x) sum(abs(x), 'all'), m)), sum(cellfun(@(x) sum(x, 'all'), v))];
61 end

```

```

1 function [X, Y, r] = sample_batch(data, B, T, stream)
2 %SAMPLE_BATCH B shifted windows wholly inside one split, drawn from a given RandStream
3 % (training, validation and sampling use separate streams, so the evaluation
4 % schedule never changes which training windows are drawn).
5 % r_b ~ Uniform{1, ..., M-T}, X(b,t) = x(r_b+t-1), Y(b,t) = x(r_b+t).
6 M = numel(data);
7 if M <= T
8     error('sample_batch:short', 'split of length %d too short for T = %d', M, T);
9 end
10 r = randi(stream, [1, M - T], B, 1); % B x 1 legal starts
11 idx = r + (0:T-1); % B x T positions (implicit expansion)
12 X = reshape(data(idx), B, T);
13 Y = reshape(data(idx + 1), B, T); % shifted one step: the next characters
14 end

```

4. Operation-by-operation explanation

Python (own autodiff engine)	MATLAB	What it computes, shape
get_batch(train_ids, B, T, rng_train)	sample_batch(train, B, T, streams.train)	fresh windows from the training part only
train_curve[s - 1] = float(L_t.data)	train_curve(s) = Ls;	loss recorded before the update
opt.step(arrays, g)	adam_step(F, G, state, ...)	one Adam update of all 25 tensors
get_batch(val_ids, B_val, T, rng_val)	sample_batch(val, B_val, T, streams.val)	8 held-out windows, separate generator
evaluate(P, Xv, Yv, L)	cross_entropy(gpt_forward(...), Yv(:))	validation loss on constants: no graph, no update
opt.fingerprint() == before	isequal(before, adam_fingerprint(state))	proves the Adam state is unchanged by validation

5. Numerical verification and interpretation

Gradient check of the full model (fixed batch $B = 4$, $T = 64$, Assignment 1 initialization; the scalar with the largest gradient in each tensor):

tensor	autodiff	$\delta = 10^{-2}$	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
W_E	-0.09433	1×10^{-4}	1×10^{-6}	1×10^{-8}	6×10^{-11}	2×10^{-10}	2×10^{-9}	3×10^{-8}
W_P	-0.04708	4×10^{-5}	4×10^{-7}	4×10^{-9}	5×10^{-11}	2×10^{-10}	3×10^{-9}	3×10^{-8}
$W_{QKV}^{(1)}$	-0.02962	1×10^{-6}	1×10^{-8}	1×10^{-10}	2×10^{-11}	1×10^{-10}	2×10^{-9}	1×10^{-8}
$W_O^{(1)}$	0.04058	7×10^{-7}	7×10^{-9}	7×10^{-11}	2×10^{-11}	2×10^{-10}	3×10^{-9}	2×10^{-8}
$W_1^{(1)}$	0.02028	8×10^{-7}	8×10^{-9}	8×10^{-11}	3×10^{-12}	3×10^{-10}	3×10^{-9}	2×10^{-8}
$b_1^{(1)}$	0.03007	2×10^{-7}	2×10^{-9}	2×10^{-11}	4×10^{-11}	1×10^{-10}	2×10^{-9}	2×10^{-8}
$W_2^{(1)}$	-0.02657	1×10^{-6}	1×10^{-8}	1×10^{-10}	1×10^{-11}	5×10^{-10}	3×10^{-9}	2×10^{-8}
$b_2^{(2)}$	0.2098	7×10^{-5}	7×10^{-7}	7×10^{-9}	8×10^{-11}	1×10^{-10}	2×10^{-9}	4×10^{-8}
$\gamma_1^{(1)}$	-1.37×10^{-3}	2×10^{-10}	3×10^{-12}	4×10^{-13}	8×10^{-12}	3×10^{-10}	2×10^{-9}	7×10^{-9}
$\beta_2^{(2)}$	6.79×10^{-3}	2×10^{-9}	2×10^{-11}	1×10^{-12}	2×10^{-11}	4×10^{-10}	2×10^{-9}	2×10^{-8}
γ_f	8.51×10^{-3}	2×10^{-10}	2×10^{-12}	2×10^{-12}	3×10^{-11}	8×10^{-11}	2×10^{-9}	2×10^{-8}
W_{voc}	0.1195	2×10^{-7}	2×10^{-9}	3×10^{-11}	3×10^{-11}	2×10^{-10}	7×10^{-10}	4×10^{-8}
worst		1×10^{-4}	1×10^{-6}	1×10^{-8}	8×10^{-11}	5×10^{-10}	3×10^{-9}	4×10^{-8}

Table 5: ρ_k for one scalar of twelve tensors and seven steps δ .

The error falls by 10^2 per decade of δ down to $\delta = 10^{-5}$ (truncation, slope 2) and rises again below it (round-off): the best step is $\delta = 1 \times 10^{-5}$ with worst $\rho = 8.1 \times 10^{-11}$ (Q4.2, Q4.3). Every parameter and gradient is finite (Q4.4).

Case B3, part 2: the complete model on ABAB...

The full architecture ($L = 2$, $d = 32$, $T = 32$, $B = 16$, $V = 2$) was trained for 300 steps on the repeated sequence **ABAB**.... The training loss falls from 0.6417 ($\log 2 = 0.6931$) to 1.8×10^{-3} (B3.2); the held-out loss is 1.8×10^{-3} . Every gradient stayed finite. Samples from the prompt **A**:

categorical: ABABABABABABABABABABABABABAABABABABABABAB
greedy: ABABABABABABABABABABABABABABABABABABABABABABA

The greedy continuation never breaks the alternation (B3.3). This shows that the optimizer and the gradients work end to end, but not that the model generalizes: the validation text here is the same two-letter pattern, so a falling training loss proves fitting, not understanding of unseen structure. Generalization is measured on Tiny Shakespeare, below.

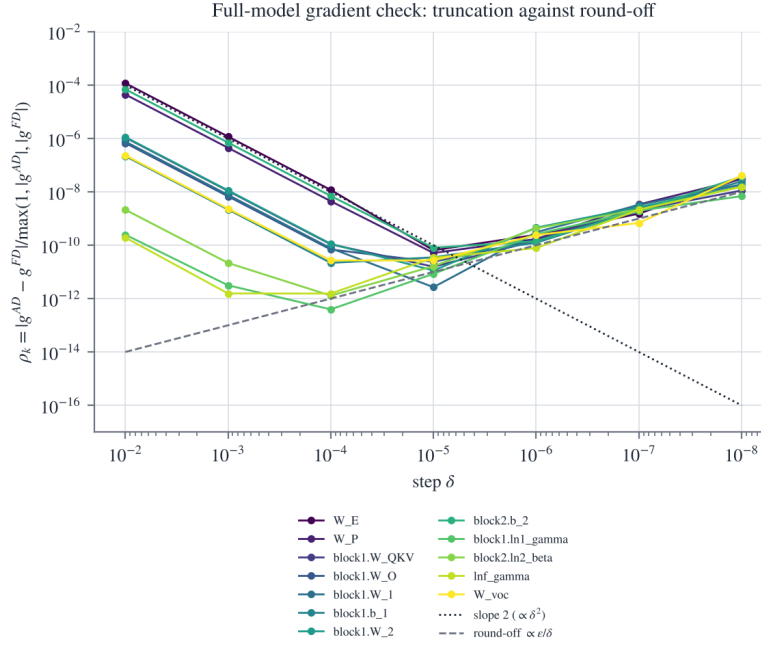


Figure 3: Truncation against round-off in the full-model gradient check.

Case B4: Tiny Shakespeare end to end

Corpus `tiny_shakespeare.txt`, SHA-256 `86c4e6aa9db7c042ec79f339dcb96d42b0075e16b8fc2e86bf0ca57e2dc565ed`, $N = 1\,115\,394$, $V = 65$; the first $\lfloor 0.9N \rfloor = 1\,003\,854$ characters train, the remaining 111 540 validate; Assignment 1’s vocabulary and split are reused, with windows drawn separately inside each part. Configuration: $B = 32$, $T = 64$, $d = 64$, $L = 2$, $\eta = 10^{-3}$, 24 500 steps (50 epochs), evaluation every 50 steps, training seed 451. Training took 61.5 minutes on a CPU (NumPy, double precision; the platform is listed in the reproducibility table at the start).

	cross-entropy (nats/char)	perplexity $e^{\mathcal{L}}$
uniform prediction, $\log V$	4.174387	65
initial model, 512 fixed validation windows	4.1972	66.50
trained, 512 fixed validation windows	1.6339	5.12
trained, 512 fixed training windows	1.3880	
generalization gap (validation – training)	+0.2459	

The first evaluation (step 1) reads 4.1967 / 4.0313 (train / validation batch) and the last 1.4138 / 1.7188. The trained model needs 2.357 bits per character on held-out text, against $\log_2 V = 6.02$ for a uniform guess. Every validation batch came from the held-out part only and no evaluation changed the optimizer state (B4.2); exactly S_{total} Adam steps were taken (B4.1).

Interpretation. The steep drop of the first $\sim 1\,000$ steps (two epochs) captures the short-range statistics: letter frequencies, common pairs and triples of letters, spaces and line breaks. The slow decline afterwards is the model learning words, speaker tags and punctuation inside its 64-character window.

The generalization gap is not small. On the fixed windows the trained model scores 1.6339 on held-out text and 1.3880 on training text, a gap of +0.2459 nats, and Figure 4 shows how it builds up. The two curves separate within the first few epochs, when every training character has been a target only a few times and a model with 112 000 parameters cannot have memorized a million of them. That early part of the gap is a property of the split: the held-out tenth of the file is the end of *The Taming of the Shrew* and the opening of *The Tempest*, whose speakers (Prospero, Miranda,

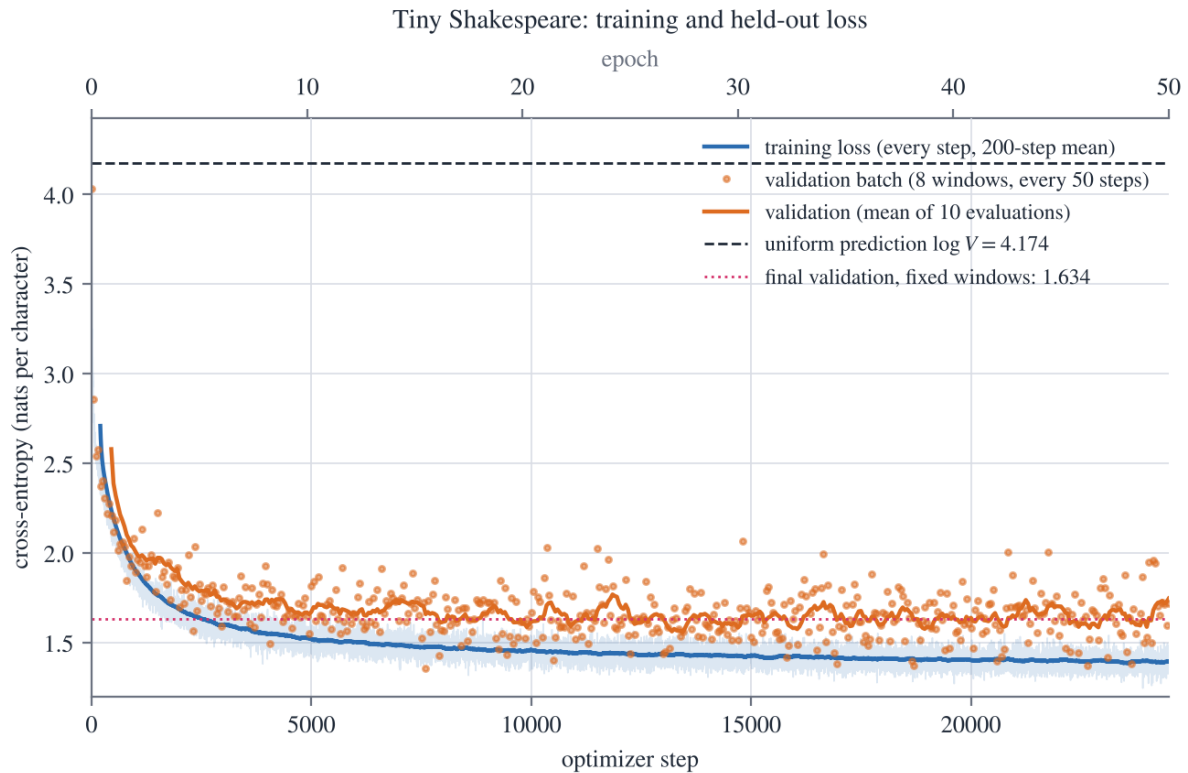


Figure 4: Cross-entropy against optimizer step. Blue: training loss of every step (faint) and its 200-step mean; orange: the 8-window validation batches and their running mean; dashed: the uniform level $\log V$; dotted: the final fixed-window validation loss.

Ariel, ...) never speak in the training part, so it is harder text for any model trained on the rest (Assignment 1, Section 3). From about epoch 20 on, the validation curve is flat within the noise of its 8-window batches while the training loss keeps falling slowly, so the gap widens towards the end: the late epochs fit the training plays a little better without transferring to new ones. A run aiming at the lowest held-out loss would stop earlier or add regularization (dropout, weight decay); this experiment keeps the prescribed 50-epoch budget and reports the final model.

The same experiment in MATLAB. `run_assignment2.m` trains its own model with the same configuration, the same schedule ($S_{\text{total}} = 24\,500$ steps), the same evaluation protocol (8-window validation batches at step 1 and every 50 steps, 512 fixed windows per split chosen by the same rule) and gradients from `dlfeval/dlgradient`. It starts from MATLAB’s own saved Assignment 1 initialization (`params_init.mat`, drawn with MATLAB’s generator, so its numbers differ from Python’s) and draws its windows from MATLAB’s Mersenne-Twister streams (seeds 451, 452, 453). It is therefore a second, independent run of the experiment, not a copy of the Python run:

fixed-window cross-entropy (nats/char)	Python (own autodiff)	MATLAB (<code>dlgradient</code>)
held-out text, initial model	4.1972	4.1945
held-out text, trained model	1.6339	1.6375
training text, trained model	1.3880	1.3906
generalization gap	+0.2459	+0.2469
held-out perplexity $e^{\mathcal{L}}$	5.12	5.14
training time (minutes)	61.5	51.7

Although it starts from different initial weights and sees different random windows, the MATLAB run ends within 0.0036 nats of the Python run on held-out text and with practically the same generalization gap, and its curves (Figure 5) have the same shape: a steep drop in the first two epochs, then a slowly falling training loss while the noisy validation batches stay level. The final numbers

are therefore a property of the model and the protocol, not of one language, one initialization or one sequence of batches.

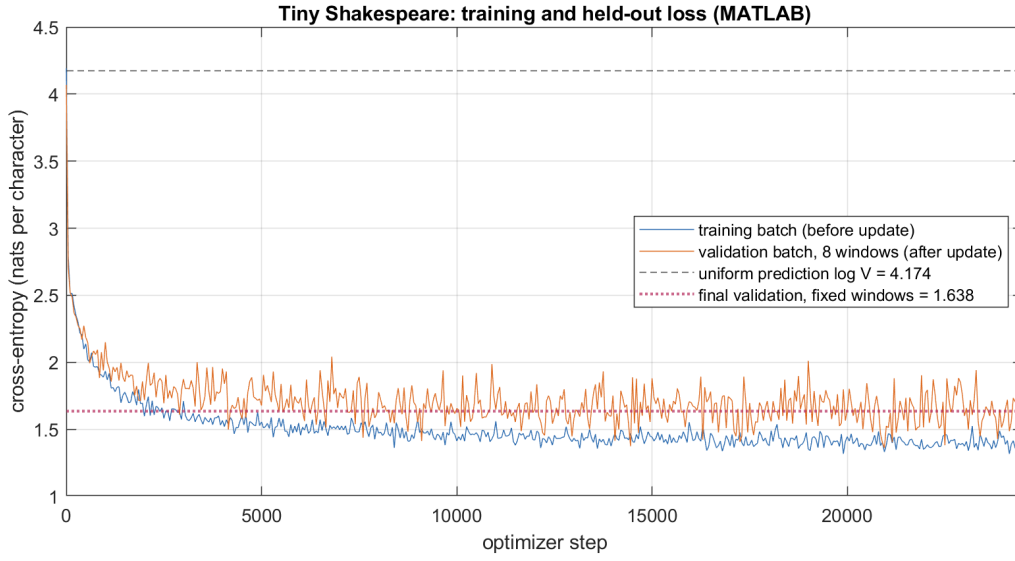


Figure 5: The MATLAB run: training-batch loss (before each update) and validation-batch loss (after it) at the 491 evaluation steps; dashed: the uniform level $\log V$; dotted: the final fixed-window validation loss.

Gradient norms. The four groups the assignment names, at the initialization (fixed batch) and at the first and last evaluations of training:

group	initialization	step 1	last evaluation
embeddings W_E, W_P	0.7291	0.6057	0.4389
attention W_{QKV}, W_O	0.6852	0.6468	0.4925
feed-forward W_1, W_2	0.9108	0.7366	0.5122
vocabulary W_{voc}	0.7438	0.5957	0.2042

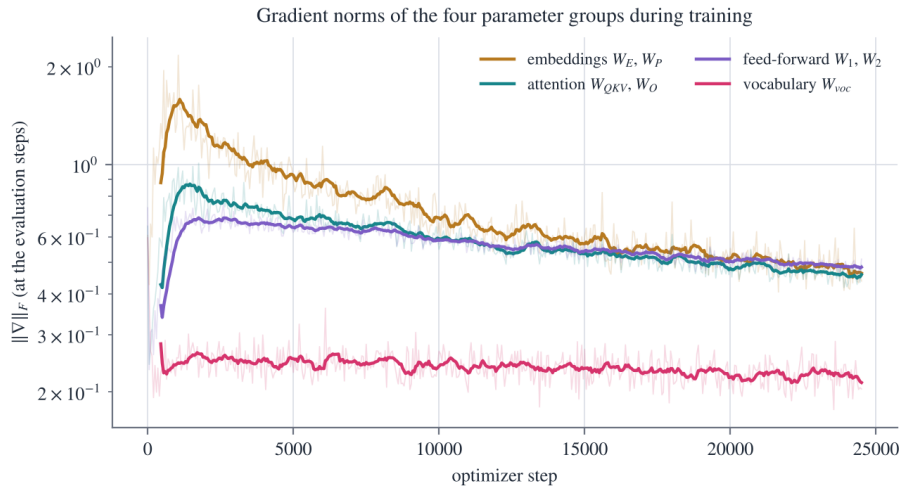


Figure 6: Gradient norms of the four parameter groups at the evaluation steps (faint) and their running mean.

Previews during training (80 characters from ROMEO: , every 5 epochs, categorical sampling):

```
[step 2450] ROMEO: and I made make I wish'd.
           DUKE VINCENTIO:
```

```

                I venfuce Tent oxe the.
                Kis had I'll

[step  4900] ROMEO: crue his recomposed jaments in;
                Like him to tell to could noted
                blesion-there:

[step  7350] ROMEO: go's pake, dust of him.

                LORTHUMBERLAND:
                We a trues poders, for
                you ever I'll
                Wh

[step  9800] ROMEO: where show ere accoun we hands up of the world; but Caes never
↪ virtue and ran wi

[step 12250] ROMEO: there fould dies.

                KING RICHARD III:
                Now, she sing your cushiod traitor of heari

[step 14700] ROMEO: you have the war,
                We me of soldiers, weeps: I will to Wave of King Ely Away, ear

[step 17150] ROMEO: O, else they dream,
                And where I wast, is so bokle your call,
                Show my prinession

[step 19600] ROMEO: go, he nake abrough from doth,
                And shrive proud I neithence.

                WARWICK:
                A Pelant

[step 22050] ROMEO: be grief,---

                AURTUCH:

                A woundier.
                A fightly brother's each varige, sir, and to

[step 24500] ROMEO: but they vanish'd will well think his the duke
                Do met! brough! well.

                ISABELLA:

```

7 Question 5: autoregressive generation

1. Purpose and assumptions

A trained model gives, for any context, a distribution over the next character. Generation repeatedly samples one character, appends it and asks again. The model only ever sees the last T characters, because its position table has T rows.

2. Derivation

For a prefix $x_{1:n}$ and continuation $x_{n+1:n+K}$,

$$p_\theta(x_{n+1:n+K} \mid x_{1:n}) = \prod_{k=1}^K p_\theta(x_{n+k} \mid x_{\max(1, n+k-T):n+k-1}). \quad (7.1)$$

At each step the retained context $x_{m:n}$, $m = \max(1, n - T + 1)$, is embedded with positions $1, \dots, n - m + 1$ (re-indexed from the start of the window, because W_P only has rows $1..T$ and the model was trained on windows starting at position 1). After the blocks and LN_f , only the last row matters:

$$z_{\text{last}} = h_{\text{last}} W_{\text{voc}} \in \mathbb{R}^{1 \times V}, \quad p_j = \frac{\exp(z_j/\tau - \max_k z_k/\tau)}{\sum_q \exp(z_q/\tau - \max_k z_k/\tau)}. \quad (7.2)$$

Row $t < n$ predicts a character that is already known; causality guarantees those rows ignore the newest token (Q5.4). **Sampling** draws $u \sim \mathcal{U}(0, 1)$ and returns the first index j with $\sum_{q \leq j} p_q > u$, so index j is chosen with probability exactly p_j . **Greedy decoding** returns $\arg \max_j p_j$ instead: deterministic, and prone to loops. The required sample uses categorical sampling at $\tau = 1$.

Probability calculation for $z = (2, 1, 0)$:

temperature	(p_1, p_2, p_3)	entropy (nats)
$\tau = 1$ (baseline)	(0.665241, 0.244728, 0.090031)	0.8324
$\tau = 0.5$	(0.866813, 0.117310, 0.015876)	0.4411
$\tau = 2$	(0.506480, 0.307196, 0.186324)	1.0202

At $\tau = 1$, $p = (e^2, e, 1)/(e^2 + e + 1)$, summing to one (Q5.1). Dividing logits by $\tau < 1$ stretches their differences, so the largest logit dominates and the entropy falls; $\tau > 1$ compresses them towards uniform. In 100 000 draws the sampler's frequencies were (0.6635, 0.2456, 0.0910) (Q5.2).

Context window. For a prompt of length $T+5 = 69$, the next prediction keeps characters 6, 7, $\dots$, 69 (one-based) and places them at positions 1, $\dots$, 64 of W_P ; the program's crop equals an explicit forward pass on those T characters exactly (Q5.3).

3. Code

Code 17: Python: cropping, the categorical draw and the generation loop (gpt_a2/sample.py).

```

1 def next_distribution(context, P, T, L=2, temperature=1.0):
2     """p over the vocabulary for the next character, given the full history `context` (1-D indices)."""
3     window = np.asarray(context)[-T:] # crop: the newest T tokens, positions re-indexed
4     Z, _ = gpt_forward(window[None, :], P, L) # (len(window) x V) logits for a batch of one
5     z_last = Z.data[-1] / temperature # only the final row predicts the next character
6     return softmax_np(z_last[None, :])[0]
7
8 def sample_index(p, rng):
9     """Inverse-CDF categorical draw: the first index whose cumulative probability exceeds u ~ U(0,1)."""
10    c = np.cumsum(p)
11    return int(min(np.searchsorted(c, rng.random() * c[-1], side="right"), len(p) - 1))
12
13 def generate(P, prompt, stoi, itos, n_new, T, rng, L=2, temperature=1.0, greedy=False):
14    """Encode `prompt`, append `n_new` sampled (or greedy) characters, decode the whole sequence."""
15    Pc = as_constants(P)
16    ids = list(encode(prompt, stoi))
17    for _ in range(n_new):
18        p = next_distribution(ids, Pc, T, L, temperature)
19        ids.append(int(np.argmax(p)) if greedy else sample_index(p, rng))
20    return decode(np.array(ids), itos)

```

Code 18: MATLAB: generate_text.m.

```

1 function txt = generate_text(P, prompt, vocab, n_new, T, eps_ln, stream, temperature, greedy)
2 %GENERATE_TEXT Autoregressive sampling from a prompt, one character at a time.

```

```

3 % Keep at most the last T indices (positions re-indexed 1..n), run the model,
4 % use only the last row of logits, softmax with temperature, then draw one index
5 % by inverse CDF (or take the argmax when greedy is true).
6 if nargin < 8, temperature = 1; end
7 if nargin < 9, greedy = false; end
8 ids = encode_text(prompt, vocab);
9 for k = 1:n_new
10     ctx = ids(max(1, end - T + 1):end); % the newest T tokens
11     Z = gpt_forward(ctx, P, eps_ln); % B = 1, so row t of Z is position t
12     z = Z(end, :) / temperature; % only the last row predicts the next character
13     p = exp(z - max(z));
14     p = p / sum(p);
15     if greedy
16         [~, nxt] = max(p);
17     else
18         nxt = find(cumsum(p) >= rand(stream), 1);
19         if isempty(nxt), nxt = numel(p); end % guards against round-off in the last bin
20     end
21     ids(end + 1) = nxt; %#ok<AGROW>
22 end
23 txt = decode_idx(ids, vocab);
24 end

```

4. Operation-by-operation explanation

Python (own autodiff engine)	MATLAB	What it computes, shape
np.asarray(context)[-T:]	ids(max(1, end - T + 1):end)	the newest $\leq T$ indices; positions restart at 1
gpt_forward(window[None, :], P, L)	gpt_forward(ctx, P, eps_ln)	a batch of one; Z has one row per position
Z.data[-1] / temperature	Z(end, :) / temperature	only the last row predicts the next character
np.searchsorted(np.cumsum(p), u)	find(cumsum(p) >= rand(stream), 1)	inverse-CDF categorical draw
ids.append(...), decode(...)	ids(end + 1) = nxt; decode_idx	append, repeat, map back to characters

5. The generated text

200 characters from the prompt ROMEO: with the trained model, categorical sampling, $\tau = 1$, seed 454:

```

ROMEO: I dead.

EDWARD:
To God by
Mlooy'd that will prithee, the heieves.

LEONTES:
Did not, as nothing prevail.

DUCHESS OF YORK:
That a did asseless'd, I cannot thee honest though upon of the man
Cause me,

```

Every generated character belongs to the corpus vocabulary (Q5.5); the output has the requested length (Q5.6). Greedy decoding from the same prompt, for contrast:

```

ROMEO: the sea, that we have the seen that will not be the world to the sea,
And the seat of the sea, and the sea,

```

```
And the seater of the stand of the stand of the stand of the stand of the
↪ stand
```

Optional temperatures (120 characters each):

```
[tau = 0.5]
ROMEO: O, for he did your hand.

ANGELO:
So be the strong that the way with one:
And he she did shame and him to means,
When th

[tau = 2.0]
ROMEO: Go,
EXEN EDWARL:
Use
I, Ww
Livew'st; I crrk tornous: Rome ent
haviorfecty Thomas it; I togets. 'Zound
Most crast will un
```

The MATLAB model (its own training run, Case B4), the same prompt and decoding rule, MATLAB's generator with seed 454:

```
ROMEO: power to Prince?
Why should my mistress deight sir,
And yet put myself this packlings
Return music that I had speak.
What more than an well make: thou queen,
I cannot you keep'd him does, no, break-no
```

Interpretation. The Python sample has the local form of the corpus: capitalized speaker tags followed by a colon and a line break (real speakers, though from different plays), short verse lines, mostly real English words with a few invented ones, and plausible punctuation. The MATLAB sample has the same local form; its 200 characters stay inside one speech, so no speaker tag appears. Neither has long-range coherence: no sentence carries a meaning and the speakers do not answer each other. The model conditions on at most 64 characters, has one head, two blocks and 112 000 parameters, so it learns which characters follow which within words and lines, not what a sentence should mean. Greedy decoding shows the other failure mode of a peaked distribution: it falls into a loop of high-probability phrases (“the stand of the stand of ...”). At $\tau = 0.5$ the text is built from common words in well-formed lines; at $\tau = 2$ it fills with invented spellings and broken speaker tags.

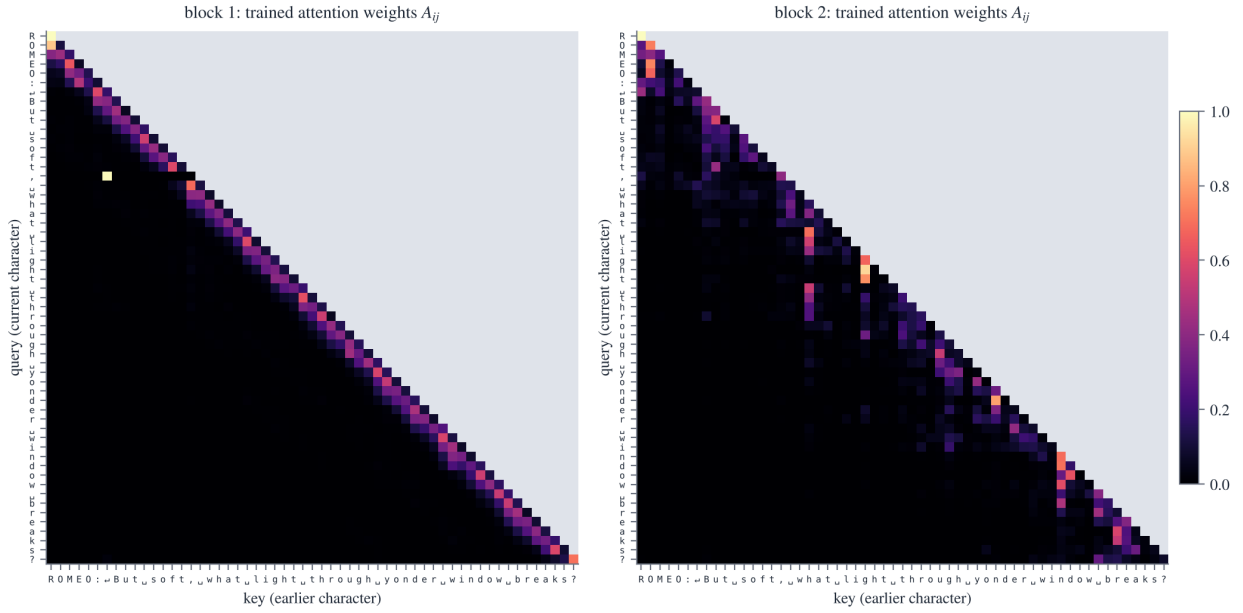


Figure 7: Trained attention weights of both blocks on a 56-character context (rows: the current character; columns: earlier characters; grey: exactly zero by the causal mask).

8 Question 6: integration, verification and submission

8.1 The numbered checks of the assignment

#	requirement	evidence (check ids, Table 8)	Python
1	every trainable shape printed and $P = 128V + 103680$	Q6.1, D.2	pass
2	embedding output and residual block input dimensions	Q6.2	pass
3	causal attention output feeds the first residual sum	Q6.3	pass
4	masked weights ~ 0 , attention rows sum to one	Q6.4, Q6.5	pass
5	non-zero finite gradients for W_E and W_P	Q3.5	pass
6	logits $BT \times V$, finite loss, initial loss vs log V	Q6.6, Q6.7	pass
7	loss plot, 200-character sample, parameter count, run instructions	B4.1, Q5.6, Q6.1	pass

The parameter table (check 1):

tensor	name	shape	entries
W_E	W_E	65×64	4160
W_P	W_P	64×64	4096
$W_{QKV}^{(1)}$	block1.W_QKV	64×192	12288
$W_O^{(1)}$	block1.W_O	64×64	4096
$W_1^{(1)}$	block1.W_1	64×256	16384
$b_1^{(1)}$	block1.b_1	1×256	256
$W_2^{(1)}$	block1.W_2	256×64	16384
$b_2^{(1)}$	block1.b_2	1×64	64
$\gamma_1^{(1)}$	block1.ln1_gamma	1×64	64
$\beta_1^{(1)}$	block1.ln1_beta	1×64	64
$\gamma_2^{(1)}$	block1.ln2_gamma	1×64	64

tensor	name	shape	entries
$\beta_2^{(1)}$	block1.ln2_beta	1×64	64
$W_{QKV}^{(2)}$	block2.W_QKV	64×192	12 288
$W_O^{(2)}$	block2.W_0	64×64	4 096
$W_1^{(2)}$	block2.W_1	64×256	16 384
$b_1^{(2)}$	block2.b_1	1×256	256
$W_2^{(2)}$	block2.W_2	256×64	16 384
$b_2^{(2)}$	block2.b_2	1×64	64
$\gamma_1^{(2)}$	block2.ln1_gamma	1×64	64
$\beta_1^{(2)}$	block2.ln1_beta	1×64	64
$\gamma_2^{(2)}$	block2.ln2_gamma	1×64	64
$\beta_2^{(2)}$	block2.ln2_beta	1×64	64
γ_f	lnf_gamma	1×64	64
β_f	lnf_beta	1×64	64
W_{voc}	W_voc	64×65	4 160
total			112 000

At initialization the logits of a real batch are 2048×65 , the loss is 4.196727 and $\log V = 4.174387$ (Q6.7): small random weights give nearly uniform predictions. The embedding output and the block input are $32 \times 64 \times 64$ (Q6.2); the first residual sum is exactly $H^{(0)} + F$ (Q6.3); in both blocks every masked weight is exactly 0 and every attention row sums to one within 4.4×10^{-16} (Q6.4, Q6.5).

8.2 Minimum acceptance tests

Each case builds a random two-block model ($d = 8$, weights $\mathcal{N}(0, 0.5^2)$, perturbed gains and biases), computes the loss and all gradients, changes the *last* input token and compares the logits of every earlier position, and compares the batched attention with a per-sample loop:

case	loss	finite	earlier logits change	last logits change	vectorized vs loop
one-token context T=1 (B=3)	1.6130	yes	—	—	4.4×10^{-16}
batch of size one B=1 (T=6)	2.5329	yes	0	0.487	4.4×10^{-16}
repeated character 'aaaaaa'	1.1355	yes	0	1.36	8.9×10^{-16}
tiny vocabulary V=2 'abba' x2	1.7665	yes	0	0.483	2.2×10^{-16}
short batch B=2, T=3, d=8	2.2963	yes	0	0.945	2.2×10^{-16}

All losses and gradients are finite (ACC.1); changing a future token leaves every earlier position's logits bit-for-bit unchanged (ACC.2) while the last position does change (ACC.3); batched and per-sample attention agree within 8.9×10^{-16} (ACC.4).

8.3 Case B5 and the summary of the validation suite

On the held-out batch of Assignment 1's protocol (32 validation windows, seed 451) and the saved Assignment 1 weights, the standalone attention of Assignment 1 (`causal_self_attention` in `gpt_a1/model.py`) and the attention inside the Assignment 2 block (`attention` in `gpt_a2/model.py`, and the output that `gpt_forward` stores) differ by at most 0 (B5.1).

case	status	what	dimensions	expected property	observed
B1	pass	fixed exact cross-entropy	logits, 3 x 3	$L = (\log 3 + \log 2 + \log 5/3)/3$; $dL/dZ = (P-E)/3$	B1.1 0, B1.2 2.8×10^{-17}
B2	pass	random and residual	FFN B=2, T=5, d=4, d_ff=16	$H_{\text{next}} = H + G$; FD = autodiff	B2.1 0, B2.2 0, B2.3 1.4×10^{-13}
B3	pass	Adam step and synthetic training	2 x 3; V = 2	matrix equations = optimizer; loss $\rightarrow 0$	B3.1 1.3×10^{-23} , B3.2 2.8×10^{-3} , B3.3 0
B4	pass	Tiny speare end end	Shake- end to d=64, L=2	val loss well below log V; held-out only	B4.1 0, B4.2 1, B4.3 2.54, B4.4 0
B5	pass	A1 vs A2 attention	32 x 64 x 64	$\max F1 - F2 < 1e-5$	B5.1 0

No case failed.

9 Table of numerical verification errors

Every recorded check, with its pass rule and the measured value. MATLAB uses the same ids.

Table 8: All 67 verification checks.

id	what is checked	pass rule	Python	MATLAB	
D.1	vocabulary differs from Assignment 1's vocab.json (characters)	= 0	0	0	pass
D.2	P(loaded) - (128V + 103680)	= 0	0	0	pass
D.3	saved Assignment 1 weights vs re-running its init (seed 451)	= 0	0	0	pass
D.4	fixed-input logits: saved vs re-created weights	= 0	0	0	pass
Q1.1	d=2 example: vectorized FFN vs the element-wise sums	$\leq 1 \times 10^{-15}$	2.8×10^{-17}	2.8×10^{-17}	pass
Q1.2	$\max \tanh\text{-GELU} - z \Phi(z) $ at $z = -1, 0, 1$	$\leq 1 \times 10^{-3}$	1.5×10^{-4}	1.5×10^{-4}	pass
Q1.3	$g'(z)$ formula vs autodiff of the composed GELU (2401 points)	$\leq 1 \times 10^{-14}$	2.2×10^{-16}	2.2×10^{-16}	pass
Q1.4	residual Jacobian: $\max dR/dH - J_F - I $	$\leq 1 \times 10^{-8}$	1.7×10^{-10}	2.2×10^{-10}	pass
Q1.5	FFN mixes features only: change of other tokens' outputs	= 0	0	0	pass
Q1.6	attention mixes tokens causally: $\max dF_i/dH_j $ for $j > i$	$\leq 1 \times 10^{-8}$	0	0	pass
Q1.7	feed-forward backward: every term vs autodiff	$\leq 1 \times 10^{-13}$	6.7×10^{-16}	8.9×10^{-16}	pass
B2.1	shape mismatches (J 5x16, G 5x4, H_next 5x4)	= 0	0	0	pass
B2.2	$\max H_{\text{next}} - (H + G) $	= 0	0	0	pass
B2.3	FD vs autodiff on 10 entries of W1, W2 (assignment measure, h=1e-5)	$\leq 1 \times 10^{-9}$	1.4×10^{-13}	8.8×10^{-13}	pass
B2.4	FD vs autodiff, scale-free relative error at h=1e-5	$\leq 1 \times 10^{-5}$	1.4×10^{-9}	3.1×10^{-10}	pass
Q2.1	2x3 example: $ L - \text{closed form} $ (engine and by hand)	$\leq 1 \times 10^{-15}$	1.1×10^{-16}	1.1×10^{-16}	pass
Q2.2	dL/dZ : $ \text{formula} - \text{autodiff} $ and $\max \text{row sum} $	$\leq 1 \times 10^{-16}$	1.4×10^{-17}	6.9×10^{-18}	pass
Q2.3	row shift by +1000: $ L(Z + c) - L(Z) $ (stable LSE)	$\leq 1 \times 10^{-12}$	2.7×10^{-15}	2.7×10^{-15}	pass
Q2.4	same flattening rule (row- or column-major) or a joint permutation: $\max \delta L $	$\leq 1 \times 10^{-14}$	2.2×10^{-16}	0	pass
Q2.5	mismatched flattening changes the loss: $ L_{\text{mix}} - L $	$\geq 1 \times 10^{-3}$	0.957	0.397	pass

id	what is checked	pass rule	Python	MATLAB	
Q2.6	all-zero logits: $ L - \log V $ ($V = 65$)	$\leq 1 \times 10^{-15}$	0	0	pass
Q2.7	target probability 0.7: $ NLL + \log 0.7 $	$\leq 1 \times 10^{-15}$	1.1×10^{-16}	1.1×10^{-16}	pass
Q2.8	head gradients $\bar{H}^T G$ and $G W_{\text{voc}}^T$ vs autodiff	$\leq 1 \times 10^{-15}$	4.2×10^{-17}	5.6×10^{-17}	pass
B1.1	$ L - L_{\text{ref}} $	$\leq 1 \times 10^{-6}$	0	0	pass
B1.2	$\max dL/dZ - (P - E)/3 $ (autodiff)	$\leq 1 \times 10^{-6}$	2.8×10^{-17}	0	pass
B1.3	correct-class probabilities vs $(1/3, 1/2, 3/5)$	$\leq 1 \times 10^{-15}$	1.1×10^{-16}	1.1×10^{-16}	pass
B1.4	shifting each row by a constant: $ \delta L $	$\leq 1 \times 10^{-13}$	1.8×10^{-15}	1.4×10^{-15}	pass
Q3.1	scalar Adam exercise: $ \text{optimizer update} - \text{hand computation} $	$\leq 1 \times 10^{-18}$	0	0	pass
Q3.2	every parameter gradient: hand-derived chain rule vs autodiff (max abs)	$\leq 1 \times 10^{-15}$	1.4×10^{-16}	1.1×10^{-16}	pass
Q3.3	tensors without any gradient (every parameter is on a path from L)	$= 0$	0	0	pass
Q3.4	$dL/dW_{\text{voc}} = \bar{H}^T dL/dZ$: shape $d \times V$ and value vs autodiff	$\leq 1 \times 10^{-16}$	1.4×10^{-17}	0	pass
Q3.5	W_E and W_P gradients: zero or non-finite (count of tensors)	$= 0$	0	0	pass
Q3.6	gradient reaching W_E through a detached copy ($1 = \text{present}$)	$= 0$	0	0	pass
B3.1	Adam step: $\max \text{optimizer} - \text{matrix equations} $ ($m1, v1, \theta1$)	$\leq 1 \times 10^{-6}$	1.3×10^{-23}	5.6×10^{-17}	pass
BB.1	$\max R - H + G + H' - H $	$= 0$	0	0	pass
BB.2	logits, probabilities, loss: max abs error	$\leq 1 \times 10^{-6}$	0	5.6×10^{-17}	pass
BB.3	dL/dZ and dL/dW_{voc} vs the displayed matrices	$\leq 1 \times 10^{-6}$	5.6×10^{-17}	2.8×10^{-17}	pass
BB.4	first Adam update vs $-\eta g/(g + \text{eps})$	$\leq 1 \times 10^{-7}$	0	0	pass
BB.5	first Adam update vs the rounded display ($+0.001$)	$\leq 1 \times 10^{-7}$	6.0×10^{-11}	6.0×10^{-11}	pass
Q4.1	$S_{\text{epoch}} - \text{floor}(N_{\text{train}} / (B T))$	$= 0$	0	0	pass
Q4.2	full-model ρ_k at $\delta = 1e-5$ (12 tensors, fixed batch)	$\leq 1 \times 10^{-8}$	8.1×10^{-11}	9.8×10^{-11}	pass
Q4.3	full-model ρ_k at the best δ ($1e-05$)	$\leq 1 \times 10^{-9}$	8.1×10^{-11}	9.8×10^{-11}	pass
Q4.4	non-finite parameters or gradients at the fixed batch ($0 = \text{none}$)	$= 0$	0	0	pass
B3.2	ABAB: final training loss / initial	≤ 0.05	2.8×10^{-3}	2.8×10^{-3}	pass
B3.3	ABAB greedy continuation: characters that break the alternation	$= 0$	0	0	pass
Q6.1	$P - (128V + 103680)$ and P - formula	$= 0$	0	0	pass
Q6.2	embedding output and block input are $B \times T \times d$ (mismatch count)	$= 0$	0	0	pass
Q6.3	first residual: $\max R - (H0 + F) $ (attention output feeds the sum)	$= 0$	0	0	pass
Q6.4	both blocks: largest masked future weight	$\leq 1 \times 10^{-7}$	0	0	pass
Q6.5	both blocks: $\max_i \sum_j A_{ij} - 1 $	$\leq 1 \times 10^{-12}$	4.4×10^{-16}	8.9×10^{-16}	pass
Q6.6	logits shape ($BT \times V$) mismatch + non-finite loss	$= 0$	0	0	pass
Q6.7	initial loss vs $\log V$: $ L0 - \log V $	≤ 0.05	0.0223	0.0179	pass
ACC.1	non-finite losses or gradients in the five cases (count)	$= 0$	0	0	pass
ACC.2	change the last token: max change of earlier positions' logits	$= 0$	0	0	pass
ACC.3	the change reaches the last position (min over cases with $T > 1$)	$\geq 1 \times 10^{-12}$	0.483	0.305	pass
ACC.4	vectorized vs per-sample attention (max over cases)	$\leq 1 \times 10^{-14}$	8.9×10^{-16}	1.3×10^{-15}	pass
B5.1	standalone (A1) vs embedded (A2) attention, saved weights	$\leq 1 \times 10^{-5}$	0	3.5×10^{-17}	pass

id	what is checked	pass rule	Python	MATLAB	
B4.1	training steps taken - S_{total}	$= 0$	0	0	pass
B4.2	validation batches only from the held-out part and no update during validation (1 = ok)	$= 1$	1	1	pass
B4.3	final fixed-window validation loss below $\log V$ (nats)	≥ 0.5	2.54	2.54	pass
B4.4	non-finite trained parameters (0 = none)	$= 0$	0	0	pass
Q5.1	$z=(2,1,0)$: $ p - (e^2, e, 1)/(e^2+e+1) $ and $ \sum p - 1 $	$\leq 1 \times 10^{-15}$	1.1×10^{-16}	1.1×10^{-16}	pass
Q5.2	sampler: max empirical frequency - p over 100000 draws	≤ 0.01	1.8×10^{-3}	2.4×10^{-3}	pass
Q5.3	cropped context: next_distribution vs explicit last-T forward	$= 0$	0	0	pass
Q5.4	changing the newest token: change of all earlier rows' logits	$= 0$	0	0	pass
Q5.5	generated characters outside the corpus vocabulary	$= 0$	0	0	pass
Q5.6	sample length - (prompt + 200)	$= 0$	0	0	pass

10 Discussion

What the implementation establishes. The forward pass, written once and differentiated automatically (by the Assignment 1 engine in Python, by `dlgradient` in MATLAB), is the computation the assignment specifies: every intermediate shape is checked, the hand-derived gradient of every tensor agrees with automatic differentiation to rounding error, and both agree with centred differences at the rate the theory predicts. The deterministic cases (Benchmark B, Case B1, the $d = 2$ example) have exact answers, and both programs reproduce them.

What training shows. On the synthetic sequence the model reaches near-zero loss within a few hundred steps, so optimization works end to end. On Tiny Shakespeare the held-out cross-entropy falls from $\log V$ to 1.6339 nats per character (2.357 bits), and the samples show that the model has learned the corpus's local structure. The generalization gap of +0.2459 nats is moderate and has two sources: the split (the held-out plays bring new speakers and words) and late-epoch fitting of the training text, since the validation loss stops improving well before the 50-epoch budget ends. The independent MATLAB run (other initial weights, other batches) reaches 1.6375 with a gap of +0.2469 nats, so these conclusions do not hinge on one particular run.

Limitations. One run per language: the two final losses agree within 0.0036 nats, which suggests that the spread due to initialization and batch order is small, but two runs are not a mean with a confidence interval, and the curve depends on the random windows drawn. Two small blocks and a 64-character context limit what can be learned; there is no dropout, weight decay, learning-rate warm-up or gradient clipping. The 8-window validation batches are noisy by design; the fixed-window estimate reduces that noise but its windows overlap one another. Pure NumPy in double precision makes each step slower than a framework on a GPU would be, which is why the experiment stops at the prescribed budget.

Where the remaining error lives. Larger context, more heads and wider blocks would let the model use longer-range structure; weight tying ($W_{\text{voc}} = W_E^T$) would save dV parameters; mini-batch noise could be reduced with larger batches or a decaying learning rate.

References

- [1] MAT 451E lecture slides and the Assignment 2 specification (B. Uzunoğlu), ITU, 2026.

- [2] A. Vaswani et al., *Attention is all you need*, NeurIPS 2017.
- [3] K. He, X. Zhang, S. Ren, J. Sun, *Deep residual learning for image recognition*, CVPR 2016.
- [4] J. L. Ba, J. R. Kiros, G. E. Hinton, *Layer normalization*, arXiv:1607.06450, 2016.
- [5] D. Hendrycks, K. Gimpel, *Gaussian error linear units (GELUs)*, arXiv:1606.08415, 2016.
- [6] D. P. Kingma, J. Ba, *Adam: a method for stochastic optimization*, ICLR 2015.
- [7] A. Radford et al., *Language models are unsupervised multitask learners*, OpenAI, 2019.
- [8] A. Karpathy, *micrograd* and *nanoGPT*, <https://github.com/karpathy>, and the Tiny Shakespeare corpus of *char-rnn*.
- [9] S. Raschka, *Build a Large Language Model (From Scratch)*, Manning, 2024.