The Transformer and GPT: From Encoder–Decoder to Decoder-Only

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

*Prerequisites: the **ATTENTION** note; the **CNN** note (batch norm, residual blocks); the **RNN** note (language model, teacher forcing, greedy decoding, BLEU).*

The **Transformer** (Vaswani et al., 2017) drops recurrence: every layer is attention plus a per-word MLP. Keeping only a decoder, without cross-attention, gives **GPT** (§7).

1 The Big Picture: Parts You Know, Wired in a New Way

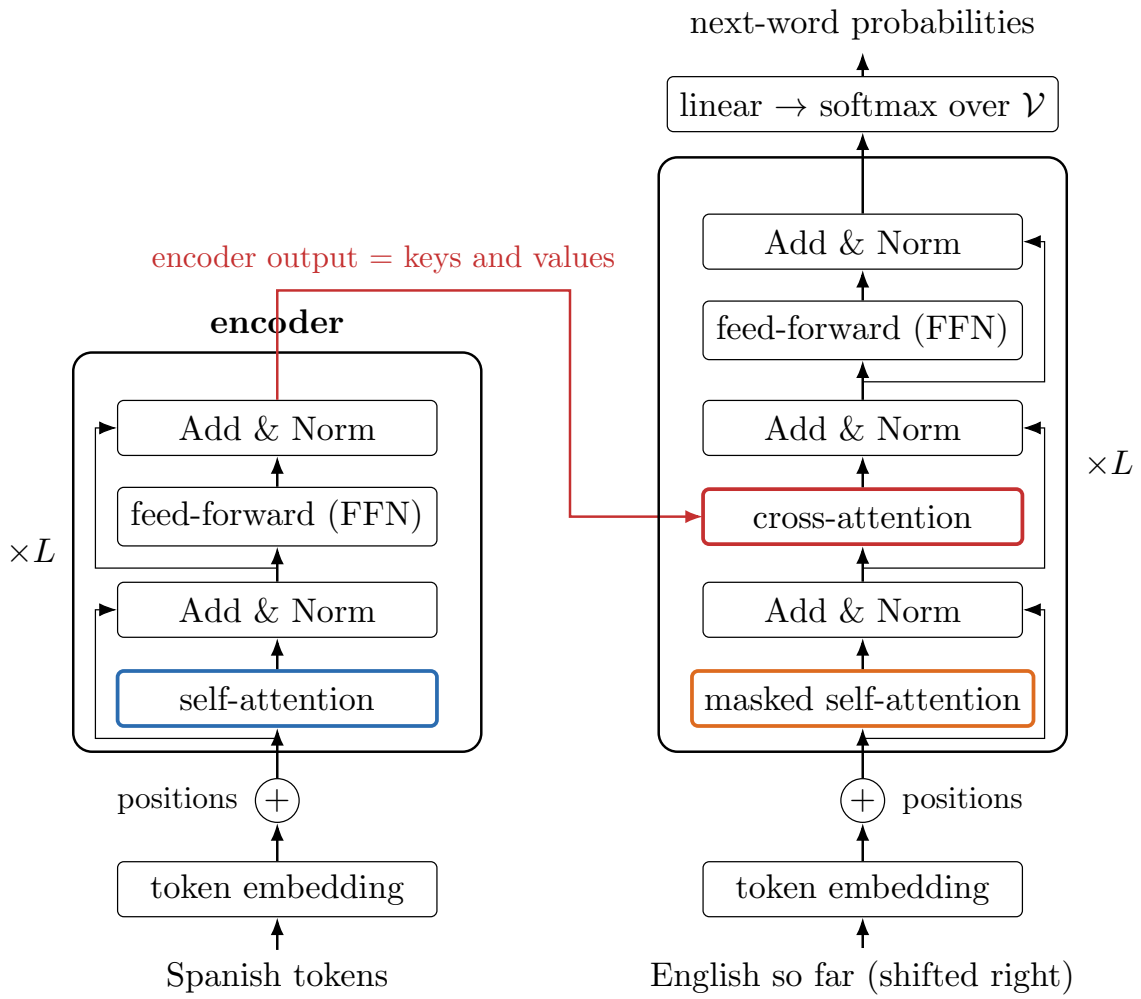


Figure 1: The Transformer (after the board, Lecture 8/8 p3). **Blue**: encoder self-attention. **Orange**: the decoder's *masked* self-attention. **Red**: cross-attention, the "old" attention of the attention translator, whose keys and values are the encoder's output. Thin arrows are the residual connections. Every block on each side is the same; the encoder's *last* output feeds the cross-attention of *every* decoder block.

Figure 1 is the whole model, redrawn from the lecture board. On the left, the **encoder** reads the Spanish sentence; on the right, the **decoder** writes the English one. Each side is a stack of L identical **blocks**. There are three kinds of attention, colour-coded as on the board.
Read it bottom to top, the way the data flows:

1. **Encoder.** Spanish tokens become vectors (embedding + positional encoding). Each encoder block lets every Spanish word look at every other Spanish word (self-attention), then processes each word's new vector with a small MLP. After L blocks, each word's vector describes that word *in the context of the whole sentence*.
2. **Decoder.** The English words written so far go in the same way. Each decoder block first lets every English word look at the English words *before* it (masked self-attention), then look at the Spanish sentence (cross-attention), then applies its MLP.
3. **Output.** A linear layer and a softmax turn each English position's final vector into probabilities for the next English word, exactly as in HW6's decoder.

Table 1 lists the parts. Only two are new.

Part	**Where you have seen it**	**What changes**
token embeddings	RNN note (`nn.Embedding`, HW6 P3–P4)	scaled by $\sqrt{d}$; the starting size matters (§6)
positional encoding	ATTENTION note (sin/cos, $X + P$)	added once, at the bottom of each stack
multi-head attention	ATTENTION note (HW7 P2(C))	used three times, wired differently (Table 2)
causal mask	ATTENTION note (HW7 P2(D))	in the decoder's first attention
cross-attention	ATTENTION note: attention in the translator (HW7 P3)	now multi-head; it reads the encoder's *last* output
residual connection	CNN note: residual blocks (HW6 P2)	around every sub-layer
feed-forward network	HW5's MLP	applied to every position separately (§3)
output layer, softmax, cross-entropy	RNN note (HW6 P3–P4)	nothing
teacher forcing, greedy decoding, BLEU	RNN note (HW6 P4)	training runs all positions in parallel (§6)
layer normalization	batch norm (CNN note, HW6 P1(D))	**NEW**: the same formula over the other axis (§2)
the wiring of the three attentions	—	**NEW**: Table 2

Table 1: The parts of a Transformer. Nine are reused; LayerNorm is a small variation of batch norm; the wiring is the real new idea.

The wiring: who asks, who answers. Every attention layer takes three inputs: queries Q, keys K and values V (ATTENTION note). The three kinds of attention differ only in *where those come from* and *which mask* they use. Table 2 is the one table to remember.

Attention	Queries from	Keys = values from	Mask	Each position can see
encoder self-attention	the Spanish positions (previous encoder block)	the same Spanish positions	Spanish padding	the whole Spanish sentence, both directions
decoder masked self-attention	the English positions (previous decoder block)	the same English positions	causal	the English words up to and including itself
cross-attention	the English positions (output of the masked sub-layer)	the encoder's **final** output	Spanish padding	the whole Spanish sentence

Table 2: The three attentions of Figure 1. Shapes with batch B, Spanish length S, English length T, width d, h heads: encoder states (B, S, d), decoder states (B, T, d); the weights are (B, h, S, S), (B, h, T, T) and (B, h, T, S) respectively.

Key point: In cross-attention the **decoder asks** (queries) and the **encoder answers** (keys and values). This is exactly the attention translator of HW7 Problem 3: the English side looks up the Spanish words it needs *now*. Self-attention is the same operation with Q, K and V all taken from one sequence.

Compared with the attention translator (ATTENTION note), the Transformer changes three things: (1) no recurrence anywhere, so nothing is computed one word at a time during training; (2) the encoder's own words talk to each other through self-attention, so each Spanish vector already knows the whole sentence; (3) every attention is multi-head, and positions come from positional encoding instead of from reading order.

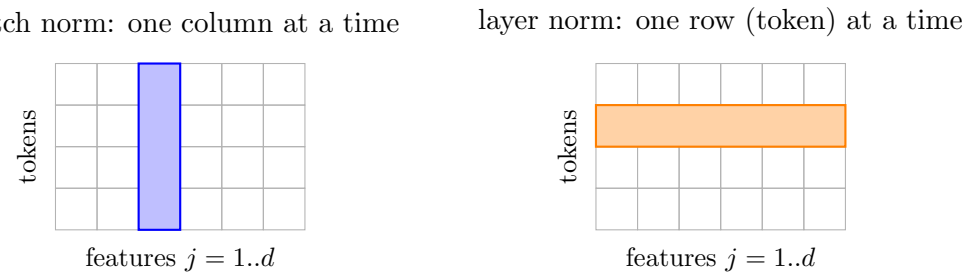
2 Layer Normalization: Batch Norm Over the Other Axis

Batch norm (CNN note) standardizes each *feature* using the statistics of the batch. Layer norm standardizes each *token*: it takes one token's d numbers, subtracts their mean and divides by their standard deviation, then scales and shifts with learned γ, β:

$$\mathrm{LN}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta, \qquad \mu = \frac{1}{d}\sum_{j=1}^{d} x_j, \quad \sigma^2 = \frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2 \tag{1}$$

Shapes: X $(B, T, d) \to (B, T, d)$; μ, σ^2 are $(B, T, 1)$, one number per token.

In words: every token's vector is brought to mean 0 and variance 1 *on its own*, then each of the d features gets back a learned scale γ_j and offset β_j (so $\gamma, \beta \in \mathbb{R}^d$, one pair per feature, like batch norm's one pair per channel). The variance divides by d, the "biased" version, as batch norm does with its batch statistics.



(For batch norm the "rows" are all tokens of all sentences in the batch; for layer norm each row is normalized by itself, whatever else is in the batch.)

Why layer norm for sequences. Three reasons, all from things that go wrong with batch norm:

- **Padding and different lengths.** Batch statistics would mix real words with padding positions, and a sentence's numbers would depend on how long its batch-mates are.
- **No dependence on other examples.** Batch norm at test time uses running averages collected during training (HW6); that works, but a token's output then depends on statistics of other sentences. Layer norm needs nothing but the token itself.
- **Same at training and test time.** No running averages, no train/eval switch (one bug fewer than in HW6).

Worked example: one formula, two axes (the lecture's matrix) $X = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, $\epsilon = 0$, $\gamma = 1$, $\beta = 0$. Layer norm, per row: row 1 has $\mu = 2$, $\sigma^2 = \frac{2}{3}$; row 2 has $\mu = 4$, $\sigma^2 = \frac{8}{3}$. Both rows become $(-1.2247,\ 0,\ 1.2247)$. Batch norm, per column: every column becomes $(-1,\ 1)$.

Layer norm erased the fact that row 2 is row 1 times two: it keeps each token's *pattern* across features and removes its overall scale. Batch norm kept the patterns across rows and removed each feature's scale. The numbers $(-1.2247, 0, 1.2247)$ are HW7 P4(A)'s first checkpoint.

3 Add & Norm and the Feed-Forward Network

Every sub-layer (an attention, or the FFN) is wrapped the same way, called **Add & Norm**:

$$\text{AddNorm}(X, Y) = \text{LN}\big(X + \text{Dropout}(Y)\big), \qquad Y = \text{Sublayer}(X) \quad \texttt{X, Y: (B,T,d)} \tag{2}$$

The "Add" is the residual connection of the CNN note's ResNet: the sub-layer only has to compute a *correction* Y to its input, and the input has a direct path around it, for the signal forward and the gradient backward. The "Norm" keeps the sum at a stable scale, block after block. This is the *post-norm* arrangement of the original Transformer and of the d2l book; §8 moves the norm.

The position-wise feed-forward network. After attention has mixed information *between* positions, each position's vector is processed *on its own* by the same two-layer MLP:

$$\text{FFN}(x) = \text{Dropout}\big(\max(0,\ xW_1 + b_1)\big)\, W_2 + b_2, \qquad W_1 \in \mathbb{R}^{d \times d_{ff}},\ W_2 \in \mathbb{R}^{d_{ff} \times d} \tag{3}$$

Shapes: $(B, T, d) \to (B, T, d_{ff}) \to (B, T, d)$.

with $d_{ff} = 4d$. It is HW5's MLP applied to each of the T rows with the same weights (in CNN language: two 1×1 convolutions over the sequence). A useful slogan: *attention communicates, the FFN computes.* About two thirds of a block's parameters are in the FFN (§4).

Dropout, and why it must sit in the same places as PyTorch's. A Transformer drops out in four places: after the embeddings, on the attention weights, on each sub-layer's output before the add (Eq. 2), and inside the FFN between its two layers (Eq. 3). The first belongs to the stack (the given wrapper that adds embeddings and positions); the other three are inside the blocks, exactly where PyTorch's `nn.TransformerEncoderLayer` and `DecoderLayer` put them. With dropout off, your block and PyTorch's give the same output once the weights are copied; with dropout on, they are still the *same model*, so after training they reach about the same quality.

4 The Encoder Block

An encoder block is self-attention and an FFN, each with Add & Norm:

$$Y = \text{AddNorm}\big(X,\ \text{MHA}(X, X, X;\ \text{padding mask})\big) \quad \texttt{(B,S,d)} \tag{4}$$

$$X' = \text{AddNorm}\big(Y,\ \text{FFN}(Y)\big) \quad \texttt{(B,S,d)} \tag{5}$$

The output has the input's shape, so blocks stack: L of them, each with its own weights. There is no causal mask: every Spanish word sees the whole sentence, left *and* right. (HW6's GRU encoder read left to right only.) The padding mask stops real words from attending to `<pad>`.

Worked example: counting a block's parameters (a unit test) Multi-head attention has four $d \times d$ projections with biases: $4(d^2 + d)$. The FFN has $d \cdot d_{ff} + d_{ff}$ and $d_{ff} \cdot d + d$. Two layer norms have $2d$ each. With HW7 P4(B)'s checkpoint sizes $d = 16$, $h = 4$, $d_{ff} = 64$: 1,088+2,128+64 = **3,280**. At the full size $d = 256$, $d_{ff} = 1024$: $263{,}168 + 525{,}568 + 1{,}024 =$ **789,760**, the same as `nn.TransformerEncoderLayer(256, 4, 1024)`. The number of heads does not change the count.

Checking against PyTorch. `nn.TransformerEncoderLayer` with `batch_first=True` (otherwise it expects (S, B, d)!), `norm_first=False` (post-norm), ReLU and dropout 0 is exactly this block. Its mask argument uses the opposite convention from HW7's (`True` = blocked); the starter's helpers convert (mask table in the ATTENTION note).

5 The Decoder Block

A decoder block has three sub-layers, each with Add & Norm:

$$Y = \text{AddNorm}\big(X,\ \text{MHA}(X, X, X;\ \text{causal mask})\big) \qquad \text{masked self-attention} \quad \texttt{(B,T,d)} \tag{6}$$

$$Z = \text{AddNorm}\big(Y,\ \text{MHA}(Y,\ E,\ E;\ \text{Spanish padding mask})\big) \qquad \text{cross-attention} \quad \texttt{(B,T,d)} \tag{7}$$

$$X' = \text{AddNorm}\big(Z,\ \text{FFN}(Z)\big) \quad \texttt{(B,T,d)} \tag{8}$$

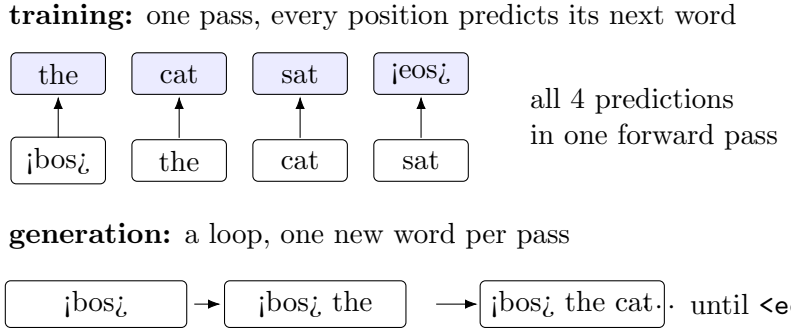
Here X is the decoder's input (the English words so far, or the previous decoder block's output) and E (B, S, d) is the encoder's final output. Read the middle line slowly: **the first argument of** MHA **is the queries**, so the queries are English (Y) and the keys and values are Spanish (E). Swapping them is the classic bug: the shapes may still work when $S = T$, and the model trains badly.

Why each mask. During training the decoder sees the whole English sentence at once (teacher forcing, §6). Position t must predict word $t+1$ without seeing it, so the self-attention is **causal**: position t attends to positions $1..t$ only. The cross-attention has *no* causal mask: the Spanish sentence is fully known before the first English word is written. It only needs the Spanish padding mask.
A test you can always run: change the last English input word; the outputs at all earlier positions must not change (HW7 P4(C) checks this). `nn.TransformerDecoderLayer` calls the two attentions `self_attn` and `multihead_attn` (the cross-attention), and its norms `norm1..3`.

6 Train in Parallel, Generate One Word at a Time

The whole translator. Encoder stack (embedding $\times\sqrt{d}$, + positional encoding, dropout, L encoder blocks) $\rightarrow$ decoder stack (the same input layer for English, L decoder blocks) $\rightarrow$ a linear layer over the English vocabulary. HW7 uses $d = 256$, $h = 4$, $d_{ff} = 1024$, $L = 3 + 3$, dropout 0.1, 14.6M parameters.

Training is one parallel pass. Teacher forcing is the same as in HW6: the decoder input is `<bos>` + the English sentence shifted right, the target is the sentence + `<eos>`, and the loss is the masked cross-entropy over real (non-pad) positions. The difference: HW6's GRU had to loop over the T positions, because step t needs state $t-1$. Here the causal mask already guarantees that position t uses only words $1..t$, so all T positions are computed *at once*, as a few large matrix products. That is what GPUs are good at, and it is why the board says "run in parallel".



Generation is still a loop. At test time the English words don't exist yet, so the decoder writes one word per pass: run the decoder on the words so far, take the last position's distribution, pick the next word, append it, repeat until `<eos>`.

What the HW's checks test. Two kinds of checkpoint appear around the training cells. *Whole sequence = one word at a time:* the same supplied English tokens are fed once all together and once position by position (dropout off, evaluation mode), and the scores must agree. That tests the causal mask and the state handling, not translation quality: it does *not* say that greedy output equals the teacher-forced prediction. *Overfit one batch:* a correct model must be able to drive the loss on one small batch close to 0 in a few dozen steps. If it can't, something in the wiring is broken (a mask, a skipped block, shifted targets); if it can, that says nothing yet about how well it generalizes, which only the full run shows.

Algorithm: greedy translation with a Transformer

1. Encode the Spanish sentence once: $E = \text{Encoder}(\text{Spanish})$ $(1, S, d)$
2. $y \leftarrow$ [<bos>]
3. Repeat up to a length cap: run the decoder on y with E (causal + Spanish padding masks); take the scores at the *last* position; append arg max; stop if it is <eos>.

HW7's decoder recomputes all of y at every step: simple and correct, but it repeats work. Real systems cache each block's keys and values (the *KV cache*, advanced note).

Key point: The starting size of the embeddings matters. The stacks multiply embeddings by $\sqrt{d} = 16$ so that they are as large as the positional encoding (entries about ± 1). That only works if the embeddings *start* with standard deviation $1/\sqrt{d}$, as in the original Transformer. With PyTorch's default `nn.Embedding` start, $\mathcal{N}(0, 1)$, times $\sqrt{d}$, the words are 16 times louder than their positions. In the HW7 build that model reached only about 31 BLEU instead of about 45, with the same architecture and budget. The starter sets this for you; remember it if you ever write your own stack.

What to expect (HW7 P4(D), reference run, same data and number of steps).

model	parameters	seconds	BLEU	1–4	5–6	7–8	9–12
HW6 GRU translator	18.0M	227	27.9	32.4	37.0	30.1	18.6
GRU + attention (P3)	19.0M	238	37.6	36.0	45.6	39.6	30.3
your Transformer	14.6M	201	45.4	48.7	52.2	47.6	37.9
`nn.Transformer`	14.6M	164	44.4	49.1	50.4	47.0	37.3

(Columns 1–4 ... 9–12: BLEU by Spanish sentence length in tokens; times on an M1 Max.) The Transformers translate clearly best, in every length bucket, with fewer parameters and less time. Two reasons: every word reaches every other word in *one* step (in the GRU, information from the first Spanish word travels through every recurrent step), and nothing is sequential in training. Yours and PyTorch's are the same model; the 1-BLEU difference comes from different random starts and dropout draws. PyTorch's is about 20% faster per step because it runs attention and layer norm as fused kernels. (One Mac detail: `nn.Transformer`'s "nested tensor" shortcut crashes on Apple GPUs in evaluation mode; the starter turns it off.)

7 Delete the Encoder: GPT, the Decoder-Only Transformer

Take Figure 1 and delete the encoder. The cross-attention has nothing left to read, so delete it too. What remains (Figure 2) is a stack of blocks with *causal* self-attention and an FFN, reading a sequence and predicting, at every position, the next token. That is **GPT** (Radford et al., 2018, 2019), the architecture behind today's large language models.

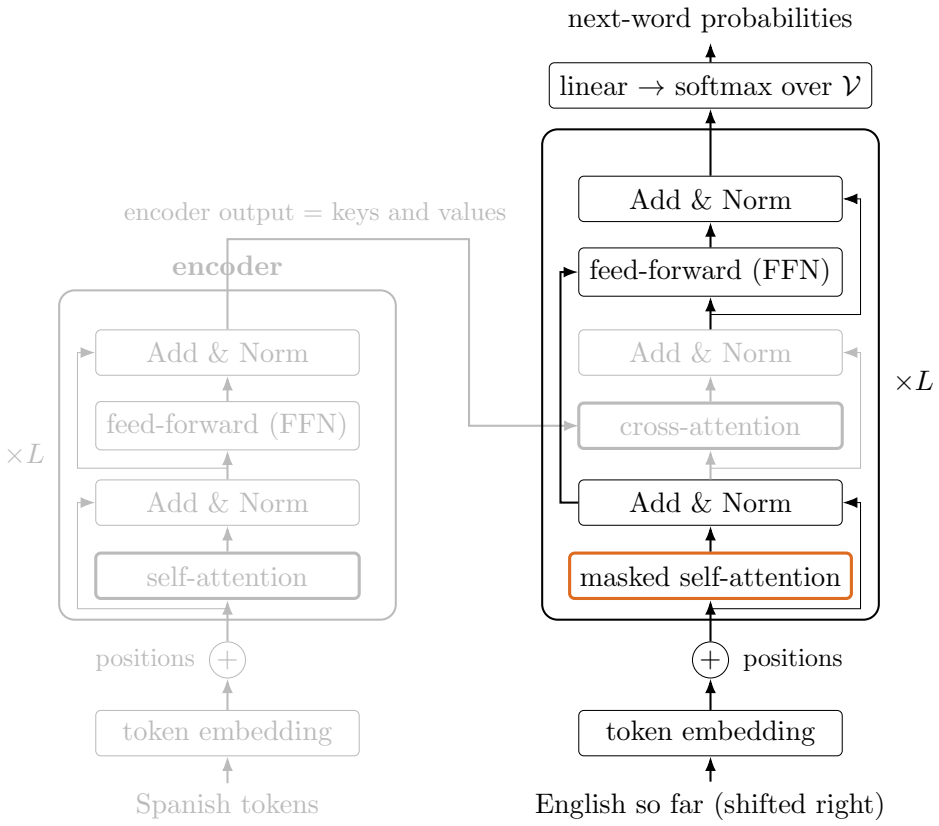


Figure 2: GPT = Figure 1 with the encoder and the cross-attention removed (grey). What is left is the encoder block of §4 with the causal mask.

Key point: A GPT block is your *encoder* block with the *causal* mask. It is called "decoder-only" because it decodes (generates) left to right, but structurally it has no cross-attention. Its training objective is HW6 Problem 3's: next-token cross-entropy, at every position, with teacher forcing.

The rest of the model (GPT-2 style).

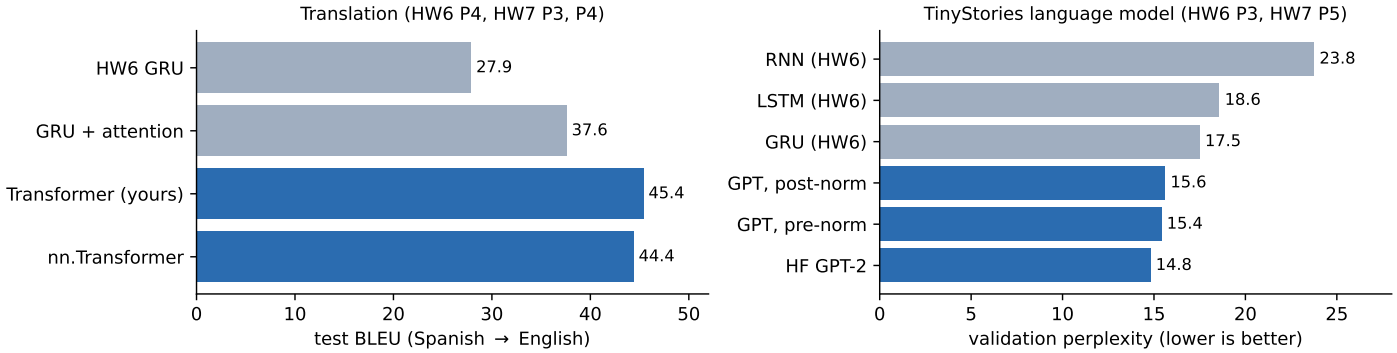
- **Learned positions.** One trainable vector per position 0..63 instead of sin/cos; added to the token embedding.
- **Weight tying.** The output layer reuses the token embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d}$: the score of word w at a position is $h \cdot e_w + b_w$, the dot product of the final hidden vector with w's own embedding. Meaning: the model predicts word w when its hidden vector "looks like" w. Without tying, the output layer would be a second $5{,}274 \times 256$ matrix: 1.35M more weights, 30% of the model.
- **Small start:** all weights $\mathcal{N}(0, 0.02^2)$, as GPT-2.

Worked example: where every parameter is (a unit test) $|\mathcal{V}| = 5{,}274$, $d = 256$, $h = 4$, $d_{ff} = 1024$, $L = 4$, context 64. Token embedding $5{,}274 \cdot 256 = 1{,}350{,}144$; positions $64 \cdot 256 = 16{,}384$; 4 blocks $4 \cdot 789{,}760 = 3{,}159{,}040$; output bias 5,274 (the output *weights* are the tied embedding). Total: **4,530,842**. Hugging Face's `GPT2LMHeadModel` with the same sizes has **4,526,080**: no output bias, but a final layer norm ($2d = 512$). The difference is $5{,}274 - 512 = \mathbf{4{,}762}$, which HW7 P5(A) asserts.

What to expect (HW7 P5(B), reference run). Same TinyStories data, vocabulary, number of training words and evaluation as HW6 Problem 3:

model	parameters	validation perplexity
RNN (HW6)	4.45M	23.8
LSTM (HW6)	5.63M	18.6
GRU (HW6)	5.24M	17.5
your GPT, post-norm	4.53M	15.6
your GPT, pre-norm (§8)	4.53M	15.4
Hugging Face GPT-2, same size	4.53M	14.8

With about the same number of parameters, the GPT is clearly better than every recurrent model: on average it hesitates between about 15 words instead of 17–24. Every position looks *directly* at all 63 words before it instead of relying on what a recurrent state managed to carry along. Hugging Face's GPT-2 lands within a point of yours (it differs in pre-norm, GELU instead of ReLU, and initialization details), so yours is a real GPT.



Encoder–decoder vs decoder-only.

	encoder–decoder (P4)	**decoder-only (P5)**
input	a source and a target sequence	one sequence
stacks, attentions	two stacks; three kinds of attention	one stack; causal self-attention only
what is predicted	the target words, given the source	every next token of the sequence
labels per sequence	the target's words	every token (all T positions)
a new task is posed as	a new (source, target) dataset	a prompt: "`Spanish <sep>`" → continue
translation in HW7	45.4 BLEU (yours)	43.9 BLEU (optional P8: `es <sep> en`, loss on the English part only)

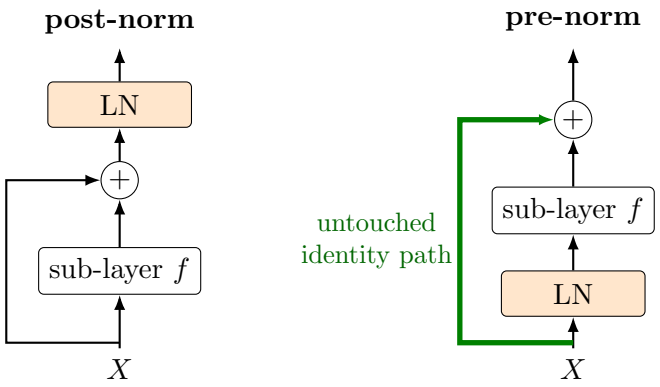
At this budget the two translate about equally well: a single causal stack can learn translation when the source is simply the beginning of the sequence. Section 10 explains why the decoder-only version took over.

8 Pre-Norm vs Post-Norm: Where the Layer Norm Goes

There are two ways to combine a sub-layer f with its residual and layer norm:

$$\text{post-norm (Vaswani 2017, d2l, BERT):} \quad X \leftarrow \mathrm{LN}\big(X + f(X)\big) \tag{9}$$

$$\text{pre-norm (GPT-2 and almost every large model since):} \quad X \leftarrow X + f\big(\mathrm{LN}(X)\big) \tag{10}$$



Why pre-norm wins at depth. Unroll L pre-norm blocks:

$$X_L = X_0 + f_1(\cdot) + f_2(\cdot) + \cdots + f_L(\cdot).$$

The input X_0 reaches the top *unchanged*, plus a sum of corrections: the CNN note's ResNet idea, kept clean through the whole stack, so the gradient reaches the first blocks directly. In post-norm, every block puts a layer norm *on* that path, which rescales the signal and the gradient $2L$ times; deep post-norm stacks are hard to train without a careful learning-rate warm-up. Since the residual stream of a pre-norm model is never normalized, it needs *one final* layer norm before the output layer.
What you should see in HW7: essentially no difference at 4 blocks (15.4 vs 15.6 in the reference run). Both train fine at this depth. The reason for pre-norm is depth: GPT-2's largest model has 48 blocks, today's models 100 or more. In PyTorch, pre-norm is `nn.TransformerEncoderLayer(..., norm_first=True)`.

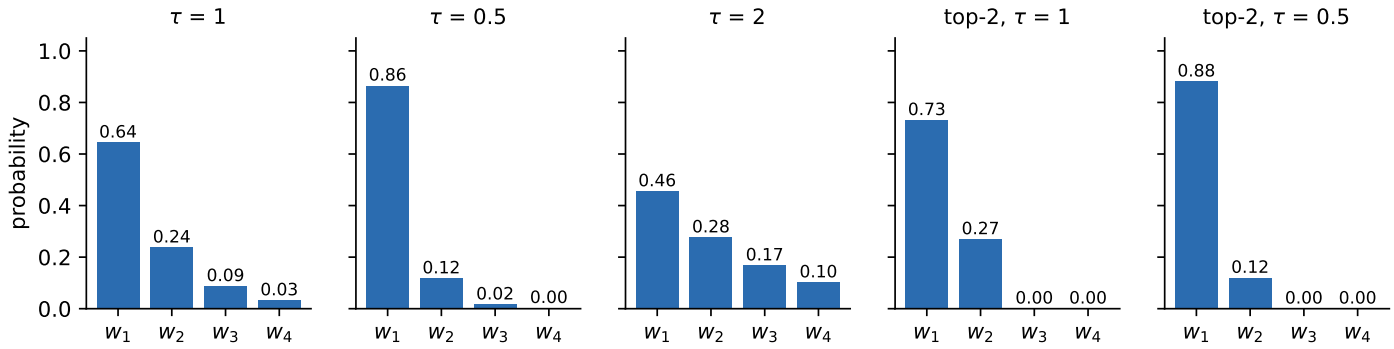
9 Sampling: Turning Probabilities Into Text

A language model gives a probability for every next token; *how* you pick from it changes what gets written. With the last position's logits $z \in \mathbb{R}^{|\mathcal{V}|}$:

- **Greedy:** always $\arg\max_w z_w$. Safe, but repetitive: once a phrase is the most likely continuation of itself, greedy repeats it.
- **Temperature τ:** sample from $p = \text{softmax}(z/\tau)$. $\tau < 1$ sharpens the distribution (safer, closer to greedy; $\tau \to 0$ *is* greedy); $\tau > 1$ flattens it (more variety, more mistakes).
- **Top-k:** keep only the k largest logits, set the rest to $-\infty$, then softmax (which renormalizes). This cuts the long tail of unlikely words that temperature alone keeps.
- (**Top-p**, "nucleus": keep the smallest set of words whose probabilities sum to p; advanced note.)

Worked example: one distribution, five settings (a unit test) Logits $z = (2, 1, 0, -1)$ for four words:

setting	w_1	w_2	w_3	w_4
$\tau = 1$	0.6439	0.2369	0.0871	0.0321
$\tau = 0.5$	0.8650	0.1171	0.0158	0.0021
$\tau = 2$	0.4551	0.2760	0.1674	0.1015
top-2, $\tau = 1$	0.7311	0.2689	0	0
top-2, $\tau = 0.5$	0.8808	0.1192	0	0



Two small facts: dividing by τ and keeping the top k commute (both preserve the order of the logits), and temperature acts on *logits*: dividing *probabilities* by τ and renormalizing changes nothing at all, a silent bug.

Algorithm: generate a story

---

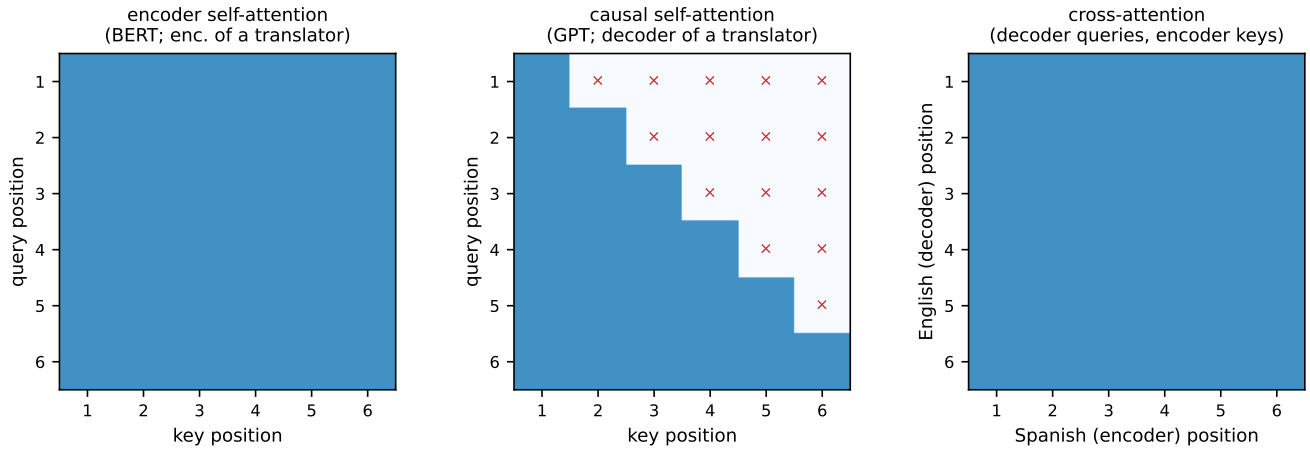
1. Start from the prompt's token IDs.
2. Repeat: crop to the last 64 tokens (the context length); run the model; take the *last* position's logits z; compute p (temperature, top-k); draw one token from p (or take arg max for greedy); append it; stop at `<eos>` or a length cap.

Key point: Check probabilities, not text. Two correct samplers with different random numbers write different stories, so comparing sampled text with a library's tells you nothing. HW7 P5(C) compares your distribution p with Hugging Face's temperature and top-k processors on the trained model's real logits; they agree exactly.

What the stories show (reference run). Greedy is fluent but circles back to the same phrases (9 repeated 4-word sequences in one story). $\tau = 0.5$ is similar: safe, common phrases. $\tau = 1$ is more varied, with odd slips. Top-10 at $\tau = 1$ is varied and mostly coherent. $\tau = 1.5$ picks rare words and falls apart. One sample per setting is a small sample; read it for tendencies.

10 Three Kinds of Transformer

Every large language model you have heard of is one of three arrangements of the same blocks. They differ in the blocks they keep, the masks, and what they are trained to predict.



	blocks	masks	trained to	typical use
encoder-only (BERT, 2018)	encoder blocks (§4)	padding only: every token sees the whole text	fill in hidden words: 15% of the tokens are masked out	understanding: classify a text, tag words, find an answer span
decoder-only (GPT)	encoder blocks with the causal mask (§7)	causal	predict the next token, at every position	generate text; any task as prompt → continuation
encoder–decoder (original Transformer, T5, BART)	both stacks with cross-attention (§1)	encoder: padding; decoder: causal; cross: source padding	produce the output text given the input text (T5: every task as "text to text", pretrained by span corruption, below)	translation, summarization

All three cut text into subword tokens (TOKENS note): BERT with WordPiece (30,522 tokens), GPT-2 with byte-level BPE (50,257), T5 with a SentencePiece unigram model (32,100).

How T5 is pretrained. T5 is pretrained on raw text with *span corruption*: random spans of the input are replaced by sentinel tokens, and the decoder must write the missing spans. Input `the dog <X> the ball <Y> park`, target `<X> chased <Y> in the <Z>`. Like BERT's hidden words and GPT's next word, the targets come from the text itself, so no labels are needed; afterwards every task (translate, summarize, answer) is phrased as input text → output text.

Adaptations, not subsets. Real models are not exactly your blocks: BERT adds segment embeddings, learned positions, GELU and a word-filling output head; GPT-2 is pre-norm with learned positions and a tied output layer. The arrangement of the blocks and masks is what defines the family.

What cross-attention gives a translator. The encoder reads the whole source in both directions, once, and keeps the result as a fixed memory. At every step every decoder position queries that memory with several heads and fetches exactly the source words it needs now, instead of relying on one summary vector (HW6). Source and target keep separate representations, lengths and vocabularies.

Why decoder-only became the common choice.

1. **One simple objective on raw text.** Next-token prediction needs no labels and no pairs, so it can use all the text there is (BERT and T5 also pretrain on raw text, but with a corruption task that differs from how they are later used).
2. **Every token is a training example.** One sequence of 1,000 tokens gives 1,000 predictions in one parallel pass (BERT learns only from its 15% hidden tokens).
3. **One interface for every task.** Translation, question answering and summarization all become "prompt → continuation"; no task-specific heads, not even for translation (optional P8).
4. **Simplicity and scale.** One stack of identical blocks is easy to make bigger, and its quality kept improving with more data and parameters.

Encoder–decoder models remain a good choice when input and output are clearly separate (translation, speech recognition); encoder-only models when only understanding is needed (search, classification).

11 HW7 Map (Problems 4, 5, 7) and Common Bugs

- **P4(A)** LayerNorm (Add & Norm and the FFN are given; read them): §2–§3. The lecture's matrix is the first checkpoint.
- **P4(B)** encoder block: §4. **P4(C)** decoder block: §5 and Table 2: which tensor is the query in cross-attention.
- **P4(D)** train and compare: §6. Expect the Transformers clearly ahead in every length bucket, and yours ≈ PyTorch's (the same model).
- **P5(A)** the GPT block and model: §7 (your encoder block + the causal mask; token + learned positions; tied output). **P5(B)** perplexity and pre-norm: §7, §8. For the "why", write the output of L blocks as "input + . . . ".
- **P5(C)** sampling: §9; compare probabilities, not text.
- **P7** (theory): §10. **Optional P8, P9:** the advanced note.

Common bugs.

- Layer norm over the wrong axis (`dim=0`: that is batch norm), or the unbiased variance.
- Forgetting the final layer norm in a pre-norm model, or comparing a pre-norm block with `nn` layers built with `norm_first=False`.
- PyTorch's defaults: `batch_first=False` (time-major!) and masks where `True` means *blocked*.
- Queries and keys swapped in cross-attention; a causal mask on the encoder; no causal mask on the decoder (the training loss drops suspiciously fast: the model reads the answer).
- Embeddings started with PyTorch's default $\mathcal{N}(0, 1)$ and multiplied by $\sqrt{d}$ (§6).
- Dropout left on in a checkpoint comparison; tied vs untied output layers when counting parameters.
- Temperature applied to probabilities instead of logits; top-k that zeroes probabilities without renormalizing.

Take-home points

- A Transformer is attention + a per-position MLP, each wrapped in a residual connection and a layer norm, stacked. Layer norm is batch norm over the features of one token.
- Three attentions: encoder self (both directions), decoder masked self (causal), and cross (decoder queries, encoder keys and values).
- Training is one parallel pass under the causal mask; generation is a loop.
- GPT = the encoder block + the causal mask + the next-token objective. Pre-norm keeps a clean identity path, which matters at depth.
- Sample with temperature and top-k; check samplers by their probabilities.

References

- Vaswani et al. (2017), *Attention Is All You Need*, NeurIPS (arXiv:1706.03762).
- Radford et al. (2019), *Language Models are Unsupervised Multitask Learners* (GPT-2), OpenAI report.
- d2l book, §11.7 (the Transformer) and §11.9 (large-scale pretraining with Transformers).
- A. Karpathy, *nanoGPT* (github.com/karpathy/nanoGPT): a complete GPT in about 300 lines.
- Optional: **TRANSFORMER_ADVANCED** (BERT and T5, RoPE, the KV cache, beam search, compute and scaling, training stability).