Generative Artificial Intelligence

Autoregressive Models

Seungjin Yang

Mon 3 – Wed 5, Aug 2026

Kyung Hee University

Outline

Learning Goals

After this lecture, students should be able to:

- factorize a joint distribution into conditionals with the chain rule;
- explain why the ordering is arbitrary in theory but not in practice;
- recognize the same autoregressive principle in PixelCNN and in a causal Transformer;
- describe self-attention, causal masking, and the role of positional information;
- distinguish teacher-forced parallel training from sequential generation;
- train and sample from a small character-level GPT.

Central question

How do we model a high-dimensional joint distribution without a latent bottleneck?

Roadmap

- Previously: a VAE routes all dependence through a low-dimensional latent z and optimizes a lower bound on the likelihood.
- Autoregressive models take the opposite route: the joint is built directly from conditionals, and the exact likelihood is maximized.
- PixelCNN imposes an ordering on pixels and masks the convolution kernels.
- Transformers replace the network inside the same factorization and mask the attention scores instead.
- Hands-on: PicoGPT, a character-level decoder-only Transformer trained by next-token prediction.

Autoregressive Models

Generative modeling

- A generative model assigns probability to data and samples new data
- For a sequence $\boldsymbol{x} = (x_1, x_2, \ldots, x_T)$, we want to model $p_\theta(\boldsymbol{x})$
- For text, each x_t is usually a word or a token
- For image, each x_t is usually a pixel

Why conditionals?

- In natural data the variables are strongly coupled: nearby pixels share intensity, adjacent words obey grammar
- Under independence, conditioning would have no effect:

$$p(x_1, x_2) = p(x_1)\,p(x_2) \quad \Longleftrightarrow \quad p(x_2 \mid x_1) = p(x_2). \tag{1}$$

- Real data violate this: observing x_1 changes which values of x_2 are plausible,

$$p(x_2 \mid x_1) \neq p(x_2). \tag{2}$$

- A generative model must capture how the observed variables constrain the remaining ones—this is the core difficulty

The latent-variable route and its limit

- A VAE explains all dependence through a low-dimensional latent:

$$z \sim \mathcal{N}(0, I), \qquad \boldsymbol{x} \sim p_\theta(\boldsymbol{x} \mid z). \tag{3}$$

- Every correlation among the $28 \times 28 = 784$ pixels of an MNIST image must be routed through z, which typically has only 2 to 20 coordinates
- This bottleneck is restrictive, but it is only one reason VAE samples look blurry. Other causes act at the same time:
  - the decoder is factorized, $p_\theta(\boldsymbol{x} \mid z) = \prod_i p_\theta(x_i \mid z)$, so pixels are conditionally independent given z and no dependence is left to model them jointly
  - a Gaussian likelihood with fixed variance makes the objective a squared error, which is minimized by an average over plausible images
  - the KL term pulls the posterior toward the prior, discarding information the decoder could have used
- The decoder is a useful component, but on its own too coarse for high-dimensional joints
- Alternative: build the joint directly from conditional distributions

Chain Rule of Probability

- Multiplying a marginal by a conditional recovers a joint:

$$p(x_1, x_2) = p(x_1)\, p(x_2 \mid x_1). \tag{4}$$

- Applying this recursively, any joint over T variables decomposes exactly:

$$p(\boldsymbol{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}). \tag{5}$$

- Each factor is a distribution over a single variable given its predecessors $\boldsymbol{x}_{<t} = (x_1, \ldots, x_{t-1})$
- The learning problem splits into T next-variable prediction tasks
- Each conditional $p_{\theta_t}(\cdot)$ is represented by a neural network:

$$p_\theta(\boldsymbol{x}) = \prod_{t=1}^{T} p_{\theta_t}(x_t \mid x_1, \ldots, x_{t-1}) \tag{6}$$

The factorization is not unique

- The chain rule applies to every ordering of the variables
- T variables admit $T!$ orderings, e.g.

$$\begin{aligned} p(x_1, x_2, x_3) &= p(x_1)\, p(x_2 \mid x_1)\, p(x_3 \mid x_1, x_2) \\ &= p(x_3)\, p(x_1 \mid x_3)\, p(x_2 \mid x_1, x_3). \end{aligned} \tag{7}$$

- Variables may also be generated in groups rather than one at a time:

$$p(x_1, x_2, x_3, x_4) = p(x_1, x_2)\, p(x_3, x_4 \mid x_1, x_2). \tag{8}$$

- A generation step may therefore emit a scalar, a vector, or a whole block, and the index t need not correspond to physical time [1]

[1] Generating blocks of variables per step underlies masked autoregressive (MAR) image models.

Choosing the order

- Every ordering represents the same joint, but the conditionals it creates differ:

$$p(x_1)\,p(x_2 \mid x_1)\,p(x_3 \mid x_1, x_2) \quad \text{vs.} \quad p(x_3)\,p(x_2 \mid x_3)\,p(x_1 \mid x_2, x_3) \tag{9}$$

- Some orderings produce conditionals with few modes that a network fits easily
- Others concentrate the difficulty in the early factors
- Every ordering is exact for the true joint, but a network of finite capacity cannot represent every conditional exactly
- The ordering therefore changes the approximate distribution that is actually learned, not only the cost of learning it
- Natural languages have a canonical order, left to right
- Images have no canonical order: the raster scan is a convention, and spiral, diagonal, or random-segment scans are equally admissible
- The chosen order is one way to inject prior knowledge or inductive bias into the model

Autoregression

- A model is autoregressive when it defines the joint by the chain-rule factorization

$$p_\theta(\boldsymbol{x}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}). \tag{10}$$

- **Auto** [2]: Earlier outputs of the model are fed back as inputs for later predictions
- **Regression** [3]: Predicting a variable from observed covariates
- The term names the factorization first; the sequential generation loop is a consequence of it
- Training may follow a different scheme
- Teacher forcing, introduced below, trains an autoregressive model without ever running the feedback loop

[2] "Auto" comes from the ancient Greek word "autos", which means "self".
[3] The term "regression" was coined by Francis Galton to describe regression toward the mean.

Text example

- Text has a natural order: left to right
- For a sentence

  “The sun rises every morning.”

  we model

$$
\begin{aligned}
p(\text{sentence}) = p(&\text{The}) \\
&\times p(\text{sun} \mid \text{The}) \\
&\times p(\text{rises} \mid \text{The sun}) \\
&\times p(\text{every} \mid \text{The sun rises}) \\
&\times p(\text{morning} \mid \text{The sun rises every}) \\
&\times p(. \mid \text{The sun rises every morning}).
\end{aligned}
\tag{11}
$$

- The model learns next-token prediction

From chain rule to a model

- The chain rule itself holds for any distribution, so nothing is lost in the decomposition
- Maintaining a separate network θ_t for each of the T factors would be possible but wasteful, and impossible for unbounded T
- Standard practice: a single network with weights θ (RNN, CNN, or Transformer) evaluates every factor:

$$p_\theta(\boldsymbol{x}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_1, \ldots, x_{t-1}) = p_\theta(x_1)\, p_\theta(x_2 \mid x_1) \cdots p_\theta(x_T \mid x_1, \ldots, x_{T-1}) \tag{12}$$

- Weight sharing across positions is a modeling assumption—the point where approximation error can enter

Representing one conditional

- Consider one factor

$$p_\theta(x_t \mid x_1, \ldots, x_{t-1}). \tag{13}$$

- The conditioning variables $x_1, \ldots, x_{t-1}$ enter the network as inputs
- The network outputs the parameters of a distribution over x_t
- If x_t is discrete, the output is a probability vector over its possible values—each step is a classification problem
- Continuous parameterizations (e.g., mixture densities) work equally well
- Discreteness is common practice, not a requirement

Neural autoregressive model

- A neural network maps context to next-token logits:

$$\boldsymbol{h}_t = f_\theta(x_{<t}), \qquad \boldsymbol{z}_t = W\boldsymbol{h}_t + \boldsymbol{b}. \tag{14}$$

- The logits define a categorical distribution:

$$p_\theta(x_t = v \mid x_{<t}) = \mathrm{softmax}(\boldsymbol{z}_t)_v \tag{15}$$

- Technically, each word is labelled by a integer and then embedded in a vector space using a learned embedding matrix
  - “cat” $\mapsto$ 42 $\mapsto$ cat_emb = torch.nn.Embedding(num_embeddings=NUM_WORDS)(42)

Generation unrolls the chain rule

- Sampling proceeds through the factorization one variable at a time:

$$\text{context: } x_{<t} \longrightarrow p_\theta(x_t \mid x_{<t}) \longrightarrow \text{sample } x_t. \tag{16}$$

- Each new sample is appended to the context, $t \leftarrow t+1$, so the input length grows by one per step
- The loop is sequential by nature, but the network inside it need not be recurrent—an MLP, CNN, or Transformer serves equally well

Why not train on generated sequences?

- Naive idea: run the sampling loop, score the result against data, and backpropagate through the whole loop
- This is not how the likelihood is defined, and it is impractical for two further reasons:
  - The gradient at step t must traverse all earlier steps, so the backprop path grows as (sequence length) $\times$ (network depth)
  - Each step contains a sampling operation, which is not differentiable
- With a deep network per step—e.g. a full Transformer—direct optimization of the generation loop is impractical

Teacher forcing

- During training, take the conditioning variables from the data set rather than from the model:

$$\text{inputs } x_{<t} \text{ from the data set}, \qquad \text{target } x_t. \tag{17}$$

- Purpose: every factor $p_\theta(x_t \mid x_{<t})$ is then evaluated at the ground-truth prefix, so the exact log-likelihood of the data is computable and differentiable—this is what makes maximum-likelihood training efficient
- Mechanism: removing the feedback edges decouples the T predictions, so given the data all of them can be computed in a single parallel pass
  - Each prediction backpropagates through one network only
  - No sampling operation appears in the training graph
- Price
  - Train/test mismatch: at generation time the model conditions on its own outputs, which it never saw during training
  - Errors can therefore compound over the generated sequence (exposure bias)

Training objective

- Under teacher forcing the whole log-likelihood is one sum over positions:

$$\log p_\theta(\boldsymbol{x}) = \sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}). \tag{18}$$

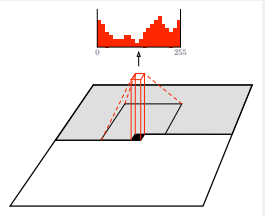
- Training minimizes negative log-likelihood:

$$\mathcal{L}_{\mathrm{NLL}} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}). \tag{19}$$

- For discrete tokens, this is the usual cross-entropy loss
- The objective is the exact likelihood: no bound, no surrogate

PixelCNN: AR model for images

PixelCNN



1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

Figure 1: (Left) A visualization of the PixelCNN that maps a neighborhood of pixels to prediction for the next pixel. (Right) n example matrix that is used to mask the 5×5 filters to make sure the model cannot read pixels below (or strictly to the right) of the current pixel to make its predictions. Source: Ref. [1]

- PixelCNN is an autoregressive model for images
- An image can be treated as an ordered sequence of pixels

$$p(\boldsymbol{x}) = \prod_{i=1}^{HWC} p(x_i \mid x_{<i}) \tag{20}$$

- PixelCNN uses masked convolutions so that each pixel sees only previous pixels

Various Autoregressive Prediction Schemes

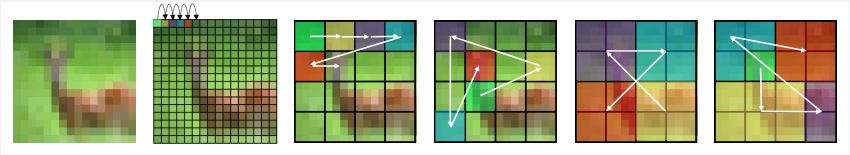


Figure 2: Autoregressive Prediction Schemes. Left-to-right: (a) original image from CIFAR 10; (b) raster-order pixel prediction; (c) raster-order patch prediction; (d) stochastic patch prediction; (e) stochastic square segment prediction ($M = 2$); (f) stochastic blob segment prediction ($K = 5$). Source: Ref. [2]

- RandSAC applies GPT-style training to effective visual representation learning by grouping image patches into spatial segments and predicting them autoregressively in a random hierarchical order
- Each scheme defines a different but equally valid autoregressive factorization of the same joint: they differ in the order and the grouping of the variables, hence in efficiency and inductive bias

PixelCNN: the key lesson

- PixelCNN is useful here because it shows that AR modeling is not only for text
- AR factorization defines the probabilistic structure.
- Masking enforces the causal dependency constraint.
- The architecture can be convolutional, recurrent, or attention-based.
-

PixelCNN: mask convolution kernels

Transformer: mask attention scores

Transformers

Why Transformers?

- Text has long-range dependencies

Example

The animal near the window was a cat. It was sleeping.

- To predict sleeping, the model must still hold cat from several tokens back
- An RNN compresses the whole history into one fixed-size state, where distant and recent information compete for the same coordinates
- Self-attention gives each token direct access to every previous token: the path length between two positions is 1 instead of their distance

Transformer language model

- A Transformer language model [3, 4] is an autoregressive model: it represents the same conditionals as before,

$$p_\theta(x_t \mid x_{<t}). \tag{21}$$

- Pipeline:

$$\text{tokens} \;\to\; \text{embeddings} \;\to\; \text{causal Transformer blocks} \;\to\; \text{logits} \;\to\; \mathrm{softmax}\,.$$

- Only the network f_θ changes. The factorization, the teacher-forced training, and the sequential generation loop are unchanged

Caveat: this is not the original Transformer

The Transformer was introduced as an encoder–decoder model for neural machine translation [3]. Its decoder block also has a cross-attention layer that attends to the encoder output. Here we use only causal self-attention—a decoder-only Transformer with no cross-attention—because it maps directly onto the autoregressive factorization.

Tokens and embeddings

- Tokens are discrete indices:

$$x_t \in \{1, \ldots, |\mathcal{V}|\}, \tag{22}$$

  where $\mathcal{V}$ is the vocabulary

- A learned embedding table maps each token to a vector:

$$\boldsymbol{e}_t = E[x_t] \in \mathbb{R}^{d_{\text{model}}}. \tag{23}$$

- Attention itself is order-agnostic, so position information is added explicitly:

$$\boldsymbol{y}_t = \boldsymbol{e}_t + \boldsymbol{p}_t, \tag{24}$$

  where $\boldsymbol{p}_t$ encodes the position t

Self-attention

- Stack the input vectors into a matrix $X \in \mathbb{R}^{T \times d_{\text{model}}}$
- Each position builds query, key, and value vectors by linear maps:

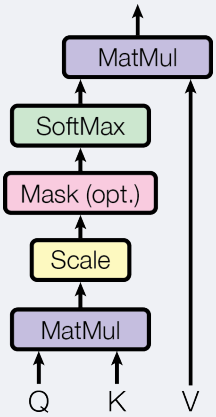
$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V. \tag{25}$$

- Attention returns, at each position, a weighted sum of the value vectors:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V. \tag{26}$$

- The weights are large where query and key match, so each token selects the information it needs from the other tokens

Scaled dot-product attention



- $QK^\top$: match every query with every key
- Scale by $1/\sqrt{d_k}$, then softmax over keys
- Multiply by V: weighted sum of values

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.$$

Scaled dot-product attention [3]

Attention step by step

- Scores: $E = QK^\top \in \mathbb{R}^{T\times T}$, where $E_{ij} = \boldsymbol{q}_i \cdot \boldsymbol{k}_j$ measures how relevant position j is to position i:

$$E = \begin{bmatrix} \boldsymbol{q}_1 \cdot \boldsymbol{k}_1 & \boldsymbol{q}_1 \cdot \boldsymbol{k}_2 & \boldsymbol{q}_1 \cdot \boldsymbol{k}_3 \\ \boldsymbol{q}_2 \cdot \boldsymbol{k}_1 & \boldsymbol{q}_2 \cdot \boldsymbol{k}_2 & \boldsymbol{q}_2 \cdot \boldsymbol{k}_3 \\ \boldsymbol{q}_3 \cdot \boldsymbol{k}_1 & \boldsymbol{q}_3 \cdot \boldsymbol{k}_2 & \boldsymbol{q}_3 \cdot \boldsymbol{k}_3 \end{bmatrix}. \tag{27}$$

- Weights: a row-wise softmax turns each row of $E/\sqrt{d_k}$ [4] into a probability distribution over positions,

$$A_{ij} = \frac{\exp\left(E_{ij}/\sqrt{d_k}\right)}{\sum_{j'} \exp\left(E_{ij'}/\sqrt{d_k}\right)}, \qquad \sum_j A_{ij} = 1. \tag{28}$$

- Output: each position gets a weighted sum of the value vectors,

$$\boldsymbol{h}_i = \sum_j A_{ij} \boldsymbol{v}_j \quad \Longleftrightarrow \quad H = A\,V. \tag{29}$$

[4] Dividing the scores by $\sqrt{d_k}$ restores unit variance, keeps the softmax in its sensitive range, and stabilizes training

Multi-head attention

- One attention operation gives one weight pattern per position. A token may need several (e.g., syntax and meaning)
- Multi-head attention [3] runs h attention operations (heads) in parallel, each with its own projections:

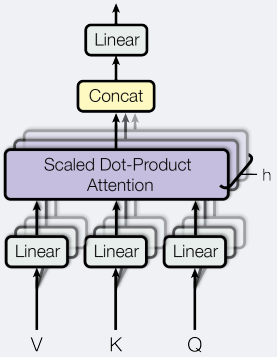
$$H^{(i)} = \mathrm{Attention}\left(XW_Q^{(i)},\ XW_K^{(i)},\ XW_V^{(i)}\right), \qquad i = 1, \ldots, h. \tag{30}$$

- Each head works in a smaller subspace, $d_k = d_{\mathrm{model}}/h$, so the total cost matches one full-width attention
- The head outputs are concatenated and mixed by a final linear map:

$$\mathrm{MultiHead}(X) = \mathrm{concat}\left(H^{(1)}, \ldots, H^{(h)}\right) W_O. \tag{31}$$

- Different heads learn different relations between tokens

Multi-head attention



- Project Q, K, V into h lower-dimensional subspaces
- Run scaled dot-product attention in each head in parallel
- Concatenate the head outputs, then apply a final linear map

$$\mathrm{MultiHead}(X) = \mathrm{concat}\left(H^{(1)}, \ldots, H^{(h)}\right) W_O.$$

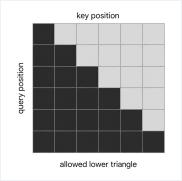
Multi-head attention: several attention layers in parallel [3]

Causal mask

- Full self-attention is not a valid autoregressive model: position t would see the future
- An additive mask removes the forbidden connections before the softmax:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V, \qquad M_{ij} = \begin{cases} 0, & j \le i, \\ -\infty, & j > i. \end{cases} \tag{32}$$

- A score of $-\infty$ becomes a weight of exactly 0 after the softmax
- Same role as the masked convolution in PixelCNN: the architecture changes, the causality constraint does not



Output: logits over the vocabulary

- At position t the last block outputs a hidden vector $\boldsymbol{h}_t$, which depends on $x_{<t}$ only
- A linear head maps it to logits over the vocabulary:

$$\boldsymbol{z}_t = W_{\text{out}}\boldsymbol{h}_t + \boldsymbol{b}_{\text{out}}, \qquad \boldsymbol{z}_t \in \mathbb{R}^{|\mathcal{V}|}. \tag{33}$$

- The next-token distribution is the softmax of the logits:

$$p_\theta(x_t = v \mid x_{<t}) = \text{softmax}(\boldsymbol{z}_t)_v. \tag{34}$$

- This is exactly the neural autoregressive model of the first section, with f_θ a causal Transformer

Training with shifted sequences

- Teacher forcing for a Transformer: feed the true tokens, predict the same sequence shifted by one position

Input	$\langle \mathrm{BOS} \rangle$	The	cat	is	sleeping
Target	The	cat	is	sleeping	.

- The context at every position comes from the data set, never from the model
- $\langle \mathrm{BOS} \rangle$ is a special begin-of-sequence token, so the first prediction is also a conditional

Parallel training

- The causal mask guarantees that output t depends on $x_{<t}$ only, so all positions are valid predictions of one forward pass
- One pass therefore evaluates every factor of the likelihood:

$$\mathcal{L}_{\mathrm{NLL}} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}). \tag{35}$$

- Each term is a next-token cross-entropy loss

Important distinction

The probability factorization is sequential, but Transformer training is parallel over positions.

Generation loop

- Generation is sequential because future tokens do not exist yet
- Starting from a prompt, repeat:
  1. run the Transformer on the current context,
  2. read the next-token distribution at the last position,
  3. choose or sample one token,

$$x_t \sim p_\theta(\cdot \mid x_{<t}), \tag{36}$$

  4. append it to the context and advance $t \leftarrow t + 1$
- The loop stops at a special end-of-sequence token or at a length limit

Generation example

```
prompt = "The cat"

while not finished:
    logits = Transformer(prompt)
    probs = softmax(logits[-1])
    next_token = sample(probs)
    prompt = prompt + next_token
    finished = (next_token == "<EOS>") or (len(prompt) > max_length)
```

- Example trajectory:

The cat $\rightarrow$ The cat is $\rightarrow$ The cat is sleeping $\rightarrow$ The cat is sleeping.

Decoding choices

- The network outputs a distribution. Decoding turns it into a token
- **Greedy decoding**: take the most probable token,

$$x_t = \arg\max_v \; p_\theta(v \mid x_{<t}). \tag{37}$$

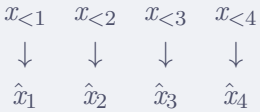
- **Sampling**: draw from the full distribution,

$$x_t \sim p_\theta(\cdot \mid x_{<t}). \tag{38}$$

- **Top-k / top-p**: sample from a truncated set of likely tokens—a compromise between diversity and quality

Training is parallel, generation is sequential

Training



- One forward pass, T predictions
- $x_{<1}$ is the empty context, supplied as $\langle \mathrm{BOS} \rangle$

Generation

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow \cdots$$

- T forward passes, one prediction each
- Each new token becomes part of the next input

Same AR principle, different masks

Model	Data	How causality is enforced
PixelCNN	pixels	masked convolution kernels
Transformer LM	tokens	masked attention scores

- The architecture changes, but the factorization remains:

$$p_\theta(\boldsymbol{x}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}). \tag{39}$$

- GPT-style large language models are this model at scale: more layers, more data, the same next-token objective [4]

Common misconceptions

- A Transformer does not output a full sentence. It outputs logits for the next-token distribution
- Softmax converts logits into probabilities. Decoding converts probabilities into tokens
- Training conditions on true previous tokens. Generation conditions on generated previous tokens
- "Autoregressive" describes the factorization $p_\theta(\boldsymbol{x}) = \prod_t p_\theta(x_t \mid x_{<t})$, and hence the generation loop, not the architecture: the Transformer itself has no recurrence

Hands-on: PicoGPT

Hands-on: PicoGPT

- A character-level, decoder-only GPT trained on TinyShakespeare (a 1.1 MB corpus)
- Everything from the slides, in $\sim$300 lines of PyTorch: token and learned position embeddings, L causal blocks, a softmax head
- Scale: block_size $= 256$, $d_{\text{model}} = 384$, 6 heads, 6 layers $\Rightarrow$ about 10.7 M parameters, 5000 training steps
- Goal: validation loss falls from $\ln 65 \approx 4.17$ (the uniform guess) to ≈ 1.5 nats/char, and samples turn from noise into Shakespeare-shaped dialogue

Adapted from Andrej Karpathy's nanoGPT: `https://github.com/karpathy/nanoGPT/`

Beyond the slides: architecture

- **Character-level tokens**: the vocabulary is just the 65 distinct characters of the corpus—no subword tokenizer. Spelling and line breaks are learned from scratch
- **Pre-norm blocks**: normalization sits inside each branch,

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \mathrm{attn}(\mathrm{LN}(\boldsymbol{x})), \qquad \boldsymbol{x} \leftarrow \boldsymbol{x} + \mathrm{MLP}(\mathrm{LN}(\boldsymbol{x})),$$

  leaving a clean residual path—this is what lets a deep stack train without a learning-rate warm-up
- **Feed-forward network**: position-wise MLP that widens by $4\times$, applies GELU, then projects back
- **Weight tying**: the token embedding matrix is reused as the output head—one matrix for input and output
- **Dropout**: regularizes the attention weights and outputs (0 by default, so the run is free to overfit; raising it reduces overfitting and narrows the train–validation gap)

Beyond the slides: training

- **Data**: random fixed-length windows; the target is the input shifted left by one character (teacher forcing at the character level)
- **Loss**: cross-entropy in nats per character; the baseline $\ln 65 \approx 4.17$ is the uniform guess over the vocabulary
- **Optimizer**: `AdamW` with $\beta = (0.9,\ 0.99)$—a shorter second-moment memory than the usual 0.999, suited to a run of only a few thousand steps
- **Gradient clipping**: rescale the global gradient norm after `backward` and before `step`, so one unlucky batch cannot move the weights too far

Sampling and things to try

- **Temperature**: divide the logits before the softmax—below 1 sharpens the distribution, above 1 flattens it. Combined with the top-k truncation from the slides
- Things to try:
  - sweep `temperature`: 0.2 is clean but repetitive, 1.5 is inventive nonsense
  - raise `dropout` to 0.2 and watch the train–validation gap narrow—5000 steps over 1 MB is enough to start memorizing
  - drop `block_size` to 32: the loss floor rises, because long-range structure is no longer predictable
  - remove weight tying, or swap the hand-written attention for `F.scaled_dot_product_attention`

Backup

Why divide by $\sqrt{d_k}$?

- For random query and key vectors with unit-variance entries, the dot product $\boldsymbol{q} \cdot \boldsymbol{k}$ is a sum of d_k terms, so its variance grows like d_k
- Large scores push the softmax into its saturated regime: one weight near 1, the rest near 0, and gradients near zero
- Dividing the scores by $\sqrt{d_k}$ restores unit variance, keeps the softmax in its sensitive range, and stabilizes training

Attention is permutation-equivariant—without position or mask

- The statement below holds for unmasked attention on inputs carrying no positional information
- Permute the key–value pairs with a permutation P: each row of $QK^\top V$ is a sum over pairs, so the sum is unchanged,

$$\Big[Q(PK)^\top (PV)\Big]_i = \sum_j (\boldsymbol{q}_i \cdot \boldsymbol{k}_{\pi(j)})\, \boldsymbol{v}_{\pi(j)} = \sum_j (\boldsymbol{q}_i \cdot \boldsymbol{k}_j)\, \boldsymbol{v}_j = \Big[QK^\top V\Big]_i . \tag{40}$$

- Permuting the queries just permutes the output rows in the same way
- All other Transformer operations (embedding, feed-forward, layer normalization) act on each position independently
- Under these hypotheses the whole model is permutation-equivariant: it cannot see token order unless order is injected explicitly
- Both standard ingredients break the equivariance, by design:
  - positional encodings make the input depend on t
  - the causal mask M_{ij} depends on the indices themselves, so a general permutation changes which pairs are masked
- A causal Transformer is therefore not permutation-equivariant under arbitrary permutations—which is exactly what an autoregressive model needs

Sinusoidal positional encoding

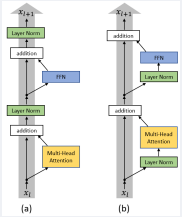
- The original Transformer encodes position t with fixed sinusoids of different frequencies:

$$p_{t,2i} = \sin\left(\frac{t}{10000^{2i/d_{\text{model}}}}\right), \qquad p_{t,2i+1} = \cos\left(\frac{t}{10000^{2i/d_{\text{model}}}}\right). \tag{41}$$

- Nearby positions get similar codes; each dimension oscillates at its own wavelength, from 2π up to $10000 \cdot 2\pi$
- The code is added to the token embedding: $\boldsymbol{y}_t = \boldsymbol{e}_t + \boldsymbol{p}_t$
- Many later models instead learn the position vectors $\boldsymbol{p}_t$ as parameters (e.g., GPT)
- Current large language models mostly use relative schemes applied inside attention rather than added to the embedding:
  - **RoPE**: rotate the query and key vectors by an angle proportional to t, so the score depends on $i - j$
  - **ALiBi**: add a linear penalty $-m\,(i - j)$ to the attention score, which extrapolates to longer contexts

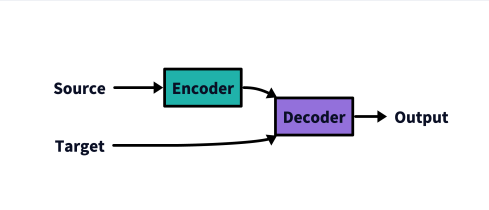
Post-LN vs. Pre-LN

- The original Transformer applies layer normalization after each residual addition (Post-LN)
- At initialization this gives large gradients near the output layers, so training needs a learning-rate warm-up stage
- Pre-LN moves the normalization before attention and the feed-forward network, inside the residual branch: gradients are well-behaved from the start [5]
- Pre-norm is the default in GPT, LLaMA, and ViT
- The normalizer itself may differ: LLaMA uses pre-RMSNorm, which rescales by the root mean square, without mean subtraction or bias



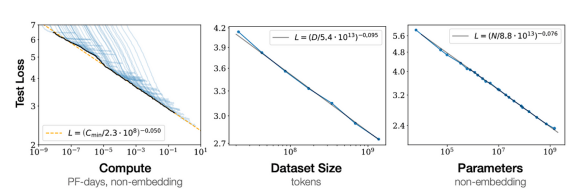
The original Transformer: encoder-decoder

- The 2017 Transformer [3] was built for translation, with two stacks:
  - an encoder reads the source sentence with full (unmasked) self-attention,
  - a decoder generates the target sentence with causal self-attention
- Each decoder block adds cross-attention: queries come from the target, keys and values from the encoder output
- The decoder is still autoregressive; the encoder is not, because the source sentence is fully known
- GPT-style language models keep only the causal decoder stack



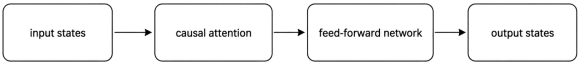
Scaling laws

- For Transformer language models, the test loss falls as a power law in compute, data set size, and model size [6]
- The trend spans many orders of magnitude, so performance somewhat beyond the measured range is predictable before training
- This is an empirical fit, not a law: it holds for a fixed architecture, data, and optimizer, over a finite range, and an irreducible-loss term bounds the gain—far extrapolation is unjustified
- The exponents were later revised: at fixed compute, model and data set size should grow together (Chinchilla)
- Even so, predictable returns motivated GPT-class models



Transformer decoder block

- A causal Transformer block contains
  - causal multi-head self-attention,
  - a position-wise feed-forward network,
  - residual connections,
  - layer normalization
- Blocks are stacked L times. A composition of causal maps is causal, so the stack remains a valid autoregressive model



References i

- Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu.
  Conditional image generation with pixelcnn decoders.
  CoRR, abs/1606.05328, 2016.
- Tianyu Hua, Yonglong Tian, Sucheng Ren, Michalis Raptis, Hang Zhao, and Leonid Sigal.
  Self-supervision through random segments with autoregressive coding (randsac).
  In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin.
Attention is all you need.
In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever.
Language models are unsupervised multitask learners.
OpenAI Technical Report, 2019.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu.
On layer normalization in the transformer architecture.
CoRR, abs/2002.04745, 2020.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei.
Scaling laws for neural language models.
CoRR, abs/2001.08361, 2020.