# From Transformer to GPT

## The Original Transformer

The original Transformer architecture (Vaswani et al., 2017) was designed for sequence-to-sequence tasks and uses an **Encoder-Decoder** framework.

- **Encoder**: Maps an input sequence to a contextualized representation.
- **Decoder**: Produces outputs token-by-token using the encoder output and previously generated tokens.
- **Self-Attention**: Mechanism for learning dependencies within a sequence.
- **Transformer**: Fully relies on self-attention in both encoder and decoder for translation tasks.

**Probabilistic Formulation:**

$$P(Y \mid X) = \prod_t P(Y_t \mid Y_{<t}, X)$$

## Evolving to a GPT

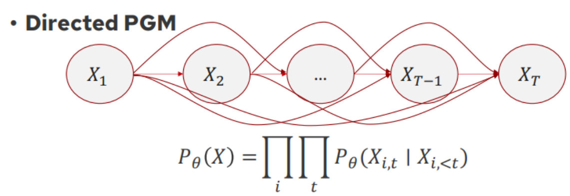
GPT simplifies the Transformer by dropping the encoder and using a decoder-only model to model input sequences:

$$P(X) = \prod_t P(X_t \mid X_{<t})$$

**Objective:**

$$\max_\theta \sum_i \sum_t \log P_\theta(X_{i,t} \mid X_{i,<t})$$

This enforces causal structure over the input, forming a directed probabilistic graphical model.




# Writing a GPT

## Model Configuration

```python
import torch
torch.manual_seed(1337)

# Training hyperparameters
batch_size = 16
max_iters = 5000
eval_interval = 100
learning_rate = 1e-3
eval_iters = 200

# Model hyperparameters
```

```python
from gpt_config import GPTConfig
config = GPTConfig(
    block_size = 8,
    device = 'cuda' if torch.cuda.is_available() else 'cpu',
    n_embd = 64,
    n_head = 4,
    n_layer = 4,
    dropout = 0.0
)
```

### Load and Encode Dataset

```python
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
chars = sorted(list(set(text)))
config.vocab_size = len(chars)

stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])
```

### Train/Test Split and Block Sampling

```python
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data))
train_data = data[:n]
val_data = data[n:]

x = train_data[:config.block_size]
y = train_data[1:config.block_size+1]
for t in range(config.block_size):
    context = x[:t+1]
    target = y[t]
    print(f"For input {context}, target is: {target}")
```

### Helper Functions

```python
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - config.block_size, (batch_size,))
    x = torch.stack([data[i:i+config.block_size] for i in ix])
    y = torch.stack([data[i+1:i+config.block_size+1] for i in ix])
    return x.to(config.device), y.to(config.device)

@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
```

```
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

### Training the Model

```
from gpt_zero import GPT
model = GPT(config)
m = model.to(config.device)
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")

    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

context = torch.zeros((1, 1), dtype=torch.long, device=config.device)
print(decode(m.generate(context, max_new_tokens=2000)[0].tolist()))
```

## Training Output

**Model Size:** `0.208961 M parameters`

**Sample Output (Shakespeare-like):**

```
WARWICK:
Yeart, their to you's my 'tcknow your turrothose...
ROMEO:
Wwell-Bethal,
Be lords!
```

## Scaling from "GPT" to GPT-4

### Overview

This section summarizes the evolution of the GPT family, from a simple, scratch-built model to GPT-4. Each stage reflects major increases in model size, dataset quality, training techniques, and inference strategies.

As the architecture scaled, so did the model's capabilities—enabling GPT to move from character-level toy outputs to world-class language generation and multimodal reasoning.

## Glossary of Key Terms

- **Tokenizer:** Breaks input text into smaller units. GPT evolved from character-level to Byte-Pair Encoding (BPE), and later to multimodal tokenization.
- **ReLU / GELU:** Activation functions that determine how neurons in the model fire. GELU is smoother and generally performs better in transformers.
- **Embedding Dimension:** Size of the vector used to represent each token.
- **Attention Heads:** Parallel attention mechanisms in each transformer block that allow the model to focus on different parts of the input simultaneously.
- **Context Length:** Maximum number of tokens the model can consider at once.
- **Parameters:** Total number of learnable weights in the model. Higher parameter counts generally allow for more expressive power.
- **Training Dataset:** Text corpus used to train the model. Increased in size and diversity across versions.
- **Optimizer:** Algorithm used to adjust the model weights during training. Adam is widely used, often with modifications like learning rate warmup and weight decay.
- **Sampling Strategy:** Method for generating output text. Greedy selects the highest-probability token each time, while top-$k$ introduces diversity.
- **Mixture-of-Experts (MoE):** A sparsely activated architecture where only subsets of the model are used per input, improving scalability.
- **RLHF:** Reinforcement Learning from Human Feedback—a training method that aligns model outputs with human preferences.

## From "GPT" to GPT-1

- **Architecture**
  - *Tokenizer*: Characters → Byte-Pair Encoding (BPE)
  - *Activation*: ReLU → GELU
  - *Weight Sharing*: Tied input/output embeddings
  - *Scale* (117M params):
    - Layers: 4 → 12
    - Attention heads: 4 → 12
    - Context length: 32 → 512
    - Vocabulary: 65 → 40,000 tokens
    - Embedding dim: 64 → 768

- **Training**
  - Dataset: TinyShakespeare (1MB) → BookCorpus (5GB)
  - Initialization/normalization: Default → Tuned
  - Optimizer: Adam → Adam + warmup + weight decay
- **Inference**
  - Sampling: Greedy → Top-$k$

## From GPT-1 to GPT-2

- **Architecture (1.5B params max):**
  - Layers: 12 → 48
  - Heads: 12 → 25
  - Embedding dim: 768 → 1600
  - Context length: 512 → 1024
  - Vocab size: 40k → 50k tokens
- **Training:** BookCorpus (5GB) → WebText (40GB)

## From GPT-2 to GPT-3

- **Architecture (1.5B → 175B params):**
  - Layers: 48 → 96
  - Heads: 25 → 96
  - Embedding dim: 1600 → 12,288
  - Context length: 1024 → 2048
- **Training:** WebText (40GB) → Common Crawl + books, code, etc. (∼570GB)

## From GPT-3 to GPT-4

- **Architecture:**
  - Likely incorporates Mixture-of-Experts (MoE)
  - Tokenizer expanded for multimodal inputs (e.g., images)
  - Scale:
    - Parameters: 175B → estimated >1T
    - Context length: 2048 → 128,000 tokens
- **Training:**
  - *Data sources:* WebText + books, Wikipedia, code, etc. (∼570GB) → much larger, proprietary dataset (details undisclosed)

- *Reported training data:* ∼13 trillion tokens (∼50TB)
- *Alignment:* Reinforcement learning from human feedback (RLHF) + system-level safety mechanisms

## Mixture of Experts (MoE)

**Core Idea:** Instead of using a single feedforward layer (FFN) at each transformer block, MoE introduces multiple "expert" FFNs and uses a routing mechanism to dynamically select a subset for each input.

**Diagram 1: MoE Routing Mechanism**

- A **router** analyzes each input token and assigns it to the most relevant expert(s).
- Only a small subset of experts (e.g., 1 or 2 out of 4) is activated per token.
- This allows for *sparse computation*, reducing the cost while increasing model capacity.

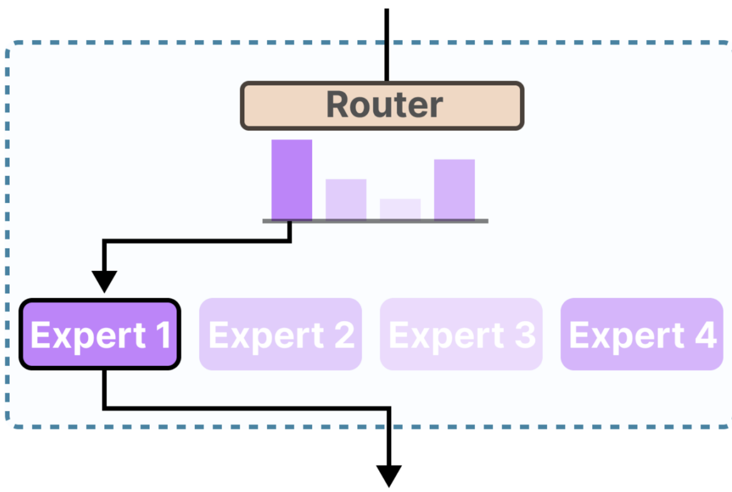
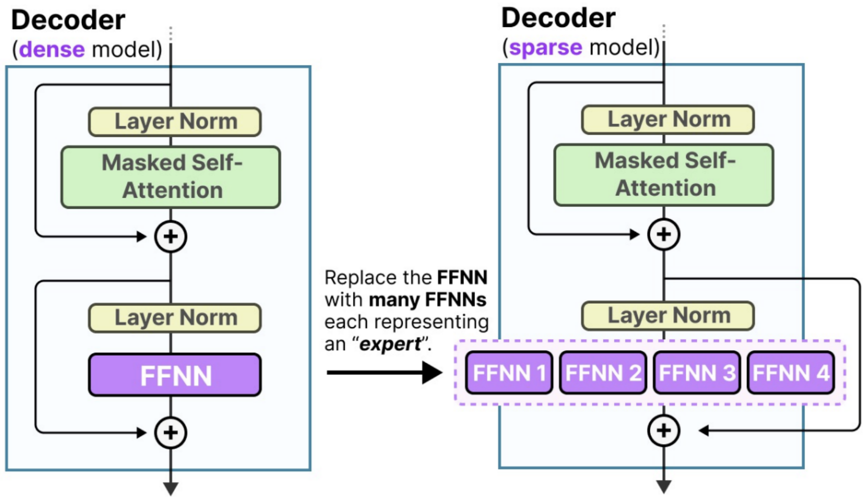



**Diagram 2: MoE in Transformer Decoders**

- In a standard transformer (left), each token passes through a single FFN after self-attention.
- In an MoE transformer (right), this FFN is replaced with **many parallel FFNs** (the experts).
- The router selects a few experts to process the token, and the outputs are combined.
- Result: Higher model capacity with the same or lower computational cost at inference time.




## Mixture of Experts: Probabilistic View

**Key Idea:** Model the output as a weighted sum of predictions from multiple expert networks.

- Let
$$P(Y \mid X) = \sum_m g_m(X) \cdot P_m(Y \mid X)$$
where:
  - $P_m(Y \mid X)$: prediction from expert $m$
  - $g_m(X)$: gating function output (i.e., weight) for expert $m$
- Subject to constraints:
$$\sum_m g_m(X) = 1, \quad \text{and} \quad g_m(X) \geq 0 \quad \forall m, X$$
ensuring a valid probability distribution over experts.
- A **gating network** computes $g_m(X)$, deciding how much each expert contributes based on the input.
- The **stochastic selector** uses these weights to sample or activate experts probabilistically.
- This framework can be trained via the **EM algorithm**, where:
  - E-step estimates expert responsibilities $g_m(X)$
  - M-step updates expert and gating parameters

## MoE: A Unifying Framework for Ensembles

Let the predictive distribution be modeled as:

$$P(Y \mid X) = \sum_m g_m(X) \cdot P_m(Y \mid X)$$

- **Mixture of Experts:** $g_m(X)$ is a *learned gating function.*
- **Bagging:** $g_m(X) = \frac{1}{M}$ is a *uniform weight* across experts.
- **Boosting:** $g_m(X) = \alpha_m$ is a *fixed expert-specific weight*, constant across inputs.

## MoE: Error Analysis

Let:

$$P(Y \mid X) = \sum_m g_m(X) \cdot P_m(Y \mid X)$$

Define the expected prediction (mean function) of the ensemble:

$$\bar{f}(x) := \mathbb{E}[Y \mid X = x] = \sum_m g_m(x) f_m(x)$$

Compare two types of errors:

- **Ensemble error:**

$$\epsilon(x) := \left(Y - \bar{f}(x)\right)^2$$

- **Average expert error:**

$$\bar{\epsilon}(x) := \frac{1}{M} \sum_m (Y - f_m(x))^2$$

**Key question:** Will minimizing ensemble error $\epsilon(x)$ also minimize average expert error $\bar{\epsilon}(x)$?

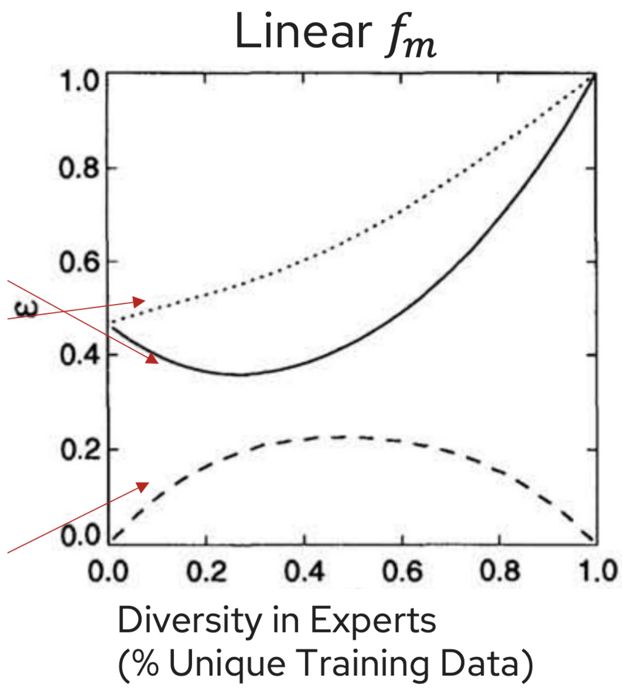
## MoE: Diversity vs. Error

**Recall:**

- Ensemble error: $\epsilon(x) = (Y - \bar{f}(x))^2$
- Average expert error: $\bar{\epsilon}(x) = \frac{1}{M} \sum_m (Y - f_m(x))^2$

**Graphical Insight:**

- As **diversity among experts** increases (x-axis = % unique training data per expert):
  - $\epsilon(x)$ (solid line): Ensemble error is minimized at moderate diversity.
  - $\bar{\epsilon}(x)$ (dotted line): Average expert error increases with diversity.
  - Dashed line: Represents disagreement (variance) between experts' outputs.



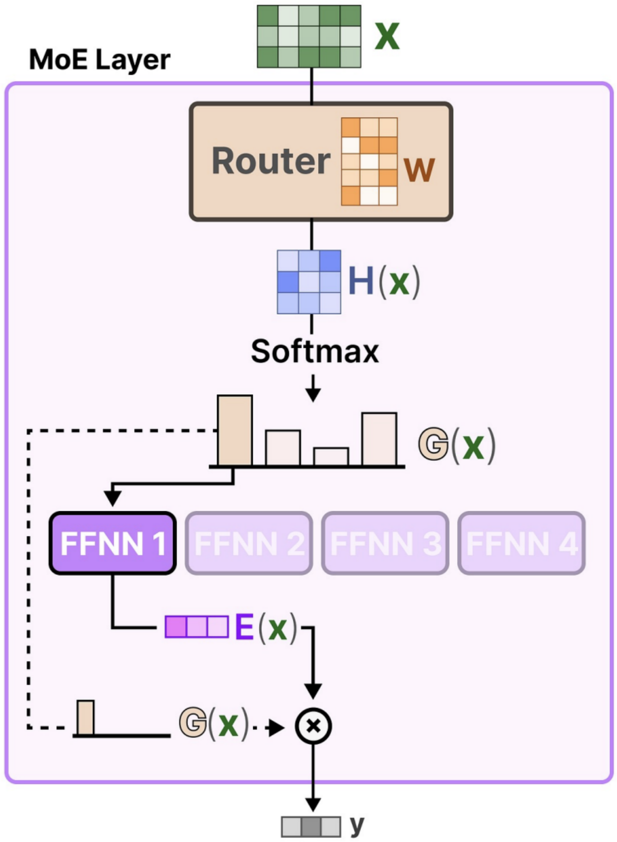

**Interpretation:**

- Expert predictions can individually overfit, but if their errors are uncorrelated, the ensemble prediction can be robust and accurate.
- **Key intuition:** Diverse (even slightly overfitted) experts cancel out each other's errors when averaged.
- *Mild overfitting* or purposeful variation among experts improves ensemble generalization.

## MoE in Large Language Models (LLMs)

**MoE Layer Structure:**

- Input $\mathbf{X}$ is passed through a **router**, which outputs activations $\mathbf{H}(x)$.
- The router computes **softmax scores** $G(x)$, representing how much to weight each expert.
- A small number of experts (e.g., top-1 or top-2) are activated based on $G(x)$.
- Only the selected FFNNs are evaluated; their outputs $E(x)$ are weighted by $G(x)$ and aggregated into the final output $y$.




**Implications for Serving:**

- *Efficiency:* Sparse activation means only a few experts need to be loaded and executed per token, reducing compute and memory.
- *Scalability:* Enables use of massive models without needing to evaluate all parameters at once.

**Implications for Training:**

- *Over-specialization:* Without proper regularization, some experts dominate while others are underused.
- *Trade-off with scale:* As shown in the plot, more experts (e.g., 128) can reduce training loss but may hurt validation performance due to overfitting or imbalance.

**Empirical Insight:**

- Validation loss increases for larger expert counts, indicating a generalization gap.
- Solution approaches may include load balancing, expert dropout, or routing noise.

# Summary Tables

## From Transformer to GPT

| **Component** | **Transformer** | **GPT** |
|---|---|---|
| Architecture | Encoder-decoder (full) | Decoder-only |
| Attention | Full self-attention | Masked (causal) self-attention |
| Positional encoding | Sinusoidal (original) | Learned positional embeddings |
| Output | Task-specific | Next-token prediction |
| Training objective | Flexible (e.g., translation) | Language modeling (autoregressive) |
| Inference | Depends on task | Greedy / sampling for text gen |

## From GPT-1 to GPT-4

**Architecture:**

- **Scale:** Broad range; largest grew from 1.5B $\rightarrow$ >1T parameters
  - Context length: 512 $\rightarrow$ 128,000
  - Layers: 12 $\rightarrow$ >96
  - Attention heads: 12 $\rightarrow$ >96
  - Embedding dimension: 768 $\rightarrow$ >12,288
  - Vocabulary size: 40k $\rightarrow$ >50k tokens
- Tokenizer: Supports multimodal inputs (e.g., images)
- Architecture includes: **Mixture-of-Experts (MoE)**

**Training:**

- Dataset: BookCorpus (5GB) $\rightarrow$ Private corpus of 13T tokens ($\sim$50TB)
- Alignment: Reinforcement learning from human feedback (RLHF)