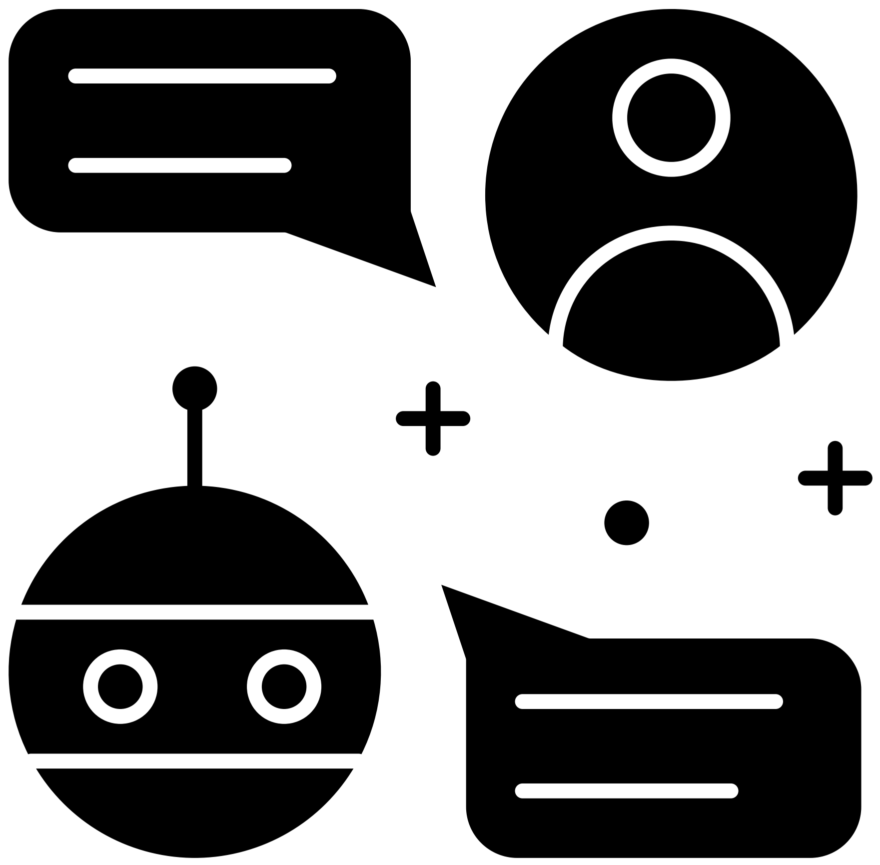
Course: Build a Small Language Model from scratch

Module 4: Model Components

Instructor: Nusrat Jahan Lia

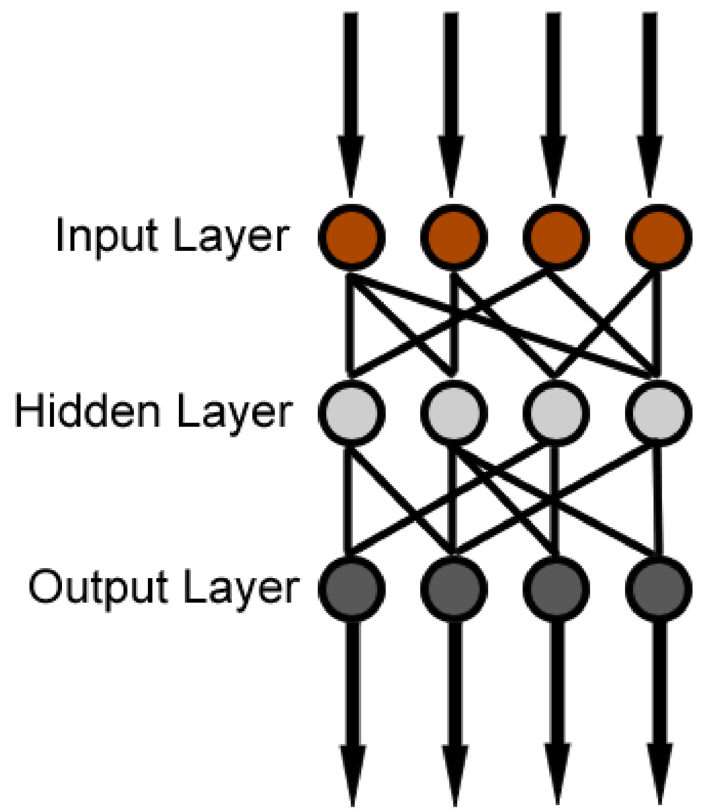


Learning Objectives

- Implement and understand feed-forward networks (MLP) with GELU activation
- Build complete transformer blocks with residual connections and layer normalization
- Assemble the full GPT model architecture with proper forward pass implementation
- Calculate and analyze the 58M parameter distribution across model components
- Understand the data flow from input tokens to output logits

Feed-Forward Networks

A feedforward neural network (FNN) is a type of artificial neural network where information flows in one direction, from input to output, without any loops or feedback connections



Multi-Layer Perceptrons (MLPs) in transformer models:

A type of feedforward artificial neural network characterized by multiple layers of interconnected nodes, including an input layer, one or more hidden layers, and an output layer.

MLP Role in Transformers — “Beyond Attention”

What happens after attention?

💬 Imagine self-attention as a gossip session — each token listens to others and exchanges information.

But after gossiping, each token needs to do **internal reflection**:

"Now that I know what others think... how do I combine this into something meaningful?"

That’s what the MLP does:

It transforms the enriched token into a new form, not by listening to others, but by thinking deeply about itself. The MLP allows each token to process intra-token features: internal representation transformation.

- fc1 is the input projection: It takes the input embedding vector of each token and expands it.
- The first linear transformation of the MLP.
- Expands the embedding size by a factor of 4.

```
class MLP(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.fc1 = nn.Linear(cfg.n_embd, 4*cfg.n_embd)
        self.fc2 = nn.Linear(4*cfg.n_embd, cfg.n_embd)
        self.act = nn.GELU()
        self.drop= nn.Dropout(cfg.dropout)
```

- fc2 is the output projection: It takes the expanded, non-linear representation and compresses it back to the original size.
- Projects the 4× expanded representation back to the original embedding dimension.

- Applies the Gaussian Error Linear Unit non-linearity.
- Introduces non-linear behavior so the model can learn complex functions.

Remember “Dropout” from earlier module?

```python
def forward(self, x):
    return self.drop(self.fc2(self.act(self.fc1(x))))
```

What is forward()?

- This is the method that defines what happens to an input tensor x when it's passed through the MLP.
- Think of it as a recipe: input goes in, transforms happen, output comes out.

Formula (Compact View)

$$MLP(x)=Dropout(fc2(GELU(fc1(x))))$$

Activation Function

A mathematical function that determines the output of a neuron, deciding whether and how strongly it should be activated based on its input. It introduces non-linearity.

Non-Linear Transformations — “Let Me Think Differently!”

Imagine a painter with only rulers (linear tools) vs. one who can bend, blend, splash, distort colors. Which one creates better art?

Without non-linearity, MLP = a matrix multiplication ➜ just a fancy linear equation.

Function	Formula
Sigmoid	$\frac{1}{1+e^{-x}}$
Tanh	$\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
ReLU	$\max(0, x)$
Leaky ReLU	$\max(0.01x, x)$
GELU	$x * \frac{1}{2}\left[1+\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$ where $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$

GELU Activation Function — “Smooth Brain Waves”

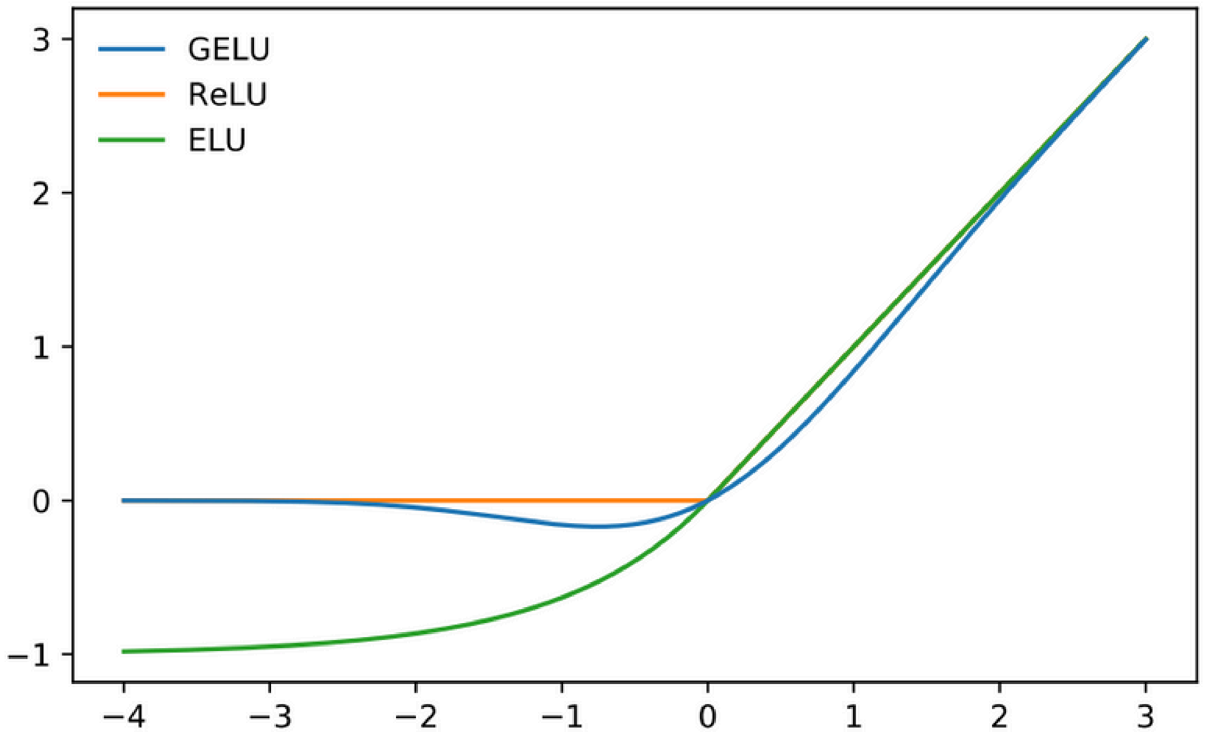


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

- ReLU: “If I don’t like you, I ignore you (zero).”
- GELU: “Hmm... I’ll consider how much I like you.”

If ReLU is a staircase, GELU is a slide. Tokens slide into the next layer gently.

Meet the Transformer Block:

This is like a brain module in your GPT model.

```
class Block(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.ln1 = nn.LayerNorm(cfg.n_embd)
        self.ln2 = nn.LayerNorm(cfg.n_embd)
        self.attn = CausalSelfAttention(cfg)
        self.mlp = MLP(cfg)

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x
```

self.ln1 and self.ln2: LayerNorms

Normalize across the embedding dimension.Keeps the activations stable across tokens.

LayerNorm(x)=
$$
\begin{aligned}
y &= \frac{x - E[x]}{\sqrt{Var[x]}} * \gamma + \beta \\
&= \frac{x - \mu}{\sigma} * \gamma + \beta \\
&= \hat{x} * \gamma + \beta
\end{aligned}
$$

$$\mu = \frac{1}{N} \sum_{j=1}^{N} x_j$$

$$\sigma = \left(\frac{1}{N} \sum_{j=1}^{N} (x_j - \mu)^2 \right)^{\frac{1}{2}}$$

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Where:

- μ, σ are per-token mean and std,
- γ, β are learnable scale and shift.

self.attn = CausalSelfAttention(cfg)

- Each token looks back at earlier tokens to decide what's relevant.
- Implements masked attention so tokens can't peek into the future.

self.mlp = MLP(cfg)

- Expands → activates → projects → regularizes.
- Introduces non-linearity and richer representations.

Forward Pass – The Transformer Mini Drama

```
x = x + self.attn(self.ln1(x))
```

Scene 1: Attention Time!

1.Normalize: ln1(x)
Each token gets its input normalized. It's like making sure all tokens are equally prepped before the team meeting.

2. Attend: attn(...)
Each token decides what to focus on from earlier tokens — using query-key-value attention!

3.Residual Add: Add it back to original x
This allows the model to: a. Keep original token info; b.Combine it with the attended context

x = x + self.mlp(self.ln2(x))

Scene 2: MLP Time!

1. Normalize again: ln2(x)
Let's normalize the outputs from the attention.

2. Inside the MLP:

- Linear: d→4d
- GELU: Smooth non-linearity
- Linear: 4d→d
- Dropout: Regularization

3.Residual Add again:
Combine MLP output with x.

Now each token has:

- Seen others (attention)
- Transformed itself (MLP)
- Stayed stable (residual + norm)

Consequences of Removing ln2:

```
x = x + self.attn(self.ln1(x))
x = x + self.mlp(x)  # ⚠️ No LayerNorm before MLP
```

A Toy Example

Assume:

- attn output has values in range [−10, 10]
- MLP expects values centered around 0, say [−1, 1]

→ Without ln2, MLP gets "hot" or skewed inputs → activations (like GELU) go into extreme non-linear regions → gradients go haywire.

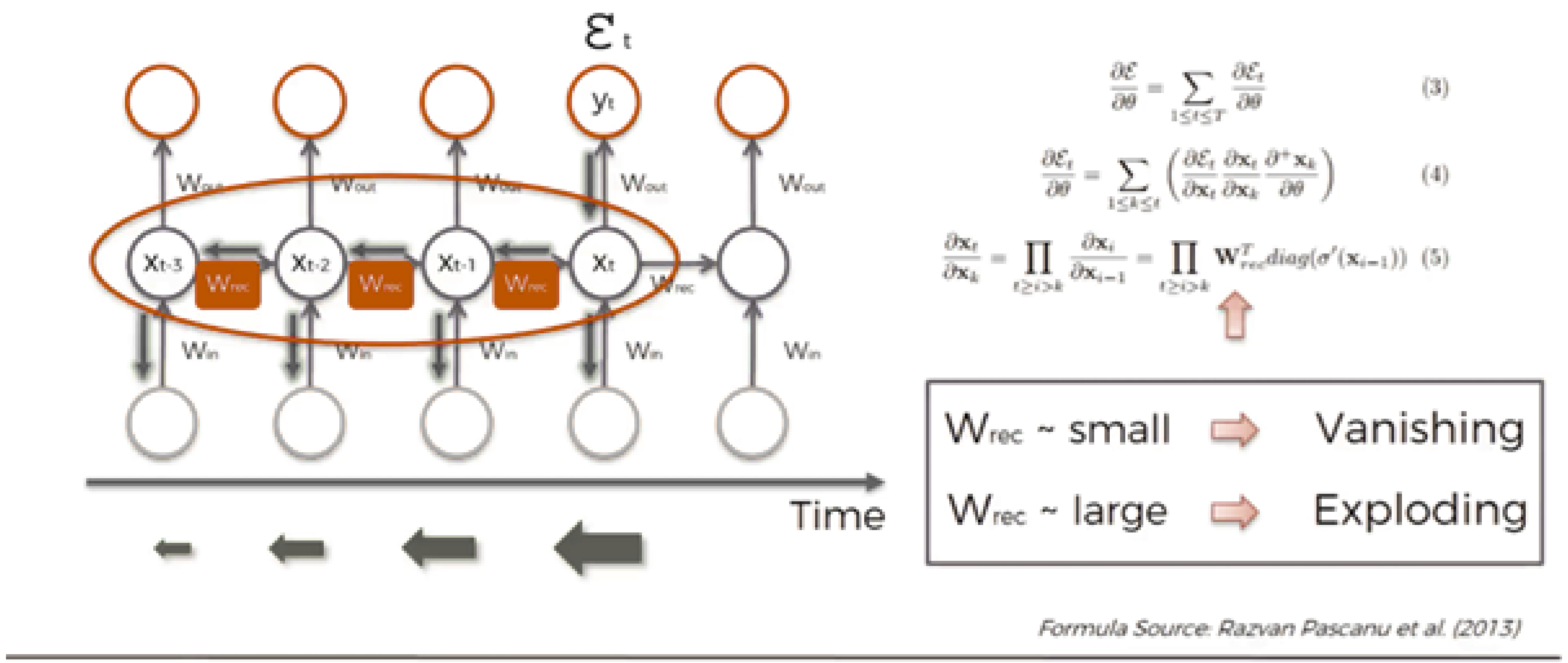
Without LayerNorm, input to the MLP could have a wildly varying scale, especially after several blocks.

This can lead to:

- ⚠️ **Gradient explosion/vanishing**
- ⚠️ **Unstable training**
- ⚠️ **Poor generalization**

⚠️ Gradient Explosion / Vanishing

- Vanishing: Gradients become tiny → weights don't update → model doesn't learn.
- Explosion: Gradients become huge → weights update too much → model blows up (loss becomes NaN).



⚠️ Unstable Training

- Model loss jumps around instead of going down smoothly.
- Training may fail to converge or suddenly diverge (go off track).

⚠️ Poor Generalization

- Model memorizes training data but fails to perform well on new, unseen data.
- Often caused by overfitting or inconsistent internal representations (e.g., due to missing LayerNorm).

Big Picture: What is this GPT class?

This is the full Generative Pre-trained Transformer : the neural net architecture that learns to predict the next word/token in a sequence.

```
self.token_emb = nn.Embedding(cfg.vocab_size, cfg.n_embd)
```

Token Embedding:
Each token (word, subword, punctuation) is assigned a learned vector of size n_embd (e.g., 512). Think of this as converting token IDs into dense, meaningful representations.

```
self.pos_emb = nn.Embedding(cfg.block_size, cfg.n_embd)
```

Positional Embedding:
Because transformers don't inherently understand order, this adds a unique position vector for each token's place in the sentence. Like giving each token a GPS coordinate

Dropout: self.drop = nn.Dropout(cfg.dropout)
Regularization method to avoid overfitting. Like randomly dropping tokens from memory to make the model more robust

Transformer Blocks: self.blocks = nn.ModuleList([Block(cfg) for _ in range(cfg.n_layer)])
A list of stacked blocks (say, 8). Each block = LayerNorm + Attention + MLP (as we saw before). These build deeper understanding.

Final Layer Normalization: self.ln_f = nn.LayerNorm(cfg.n_embd)
Smooths things out after all the blocks — ensures stable scale/variance before prediction.

Output Head: self.head = nn.Linear(cfg.n_embd, cfg.vocab_size, bias=False)
Maps the final embedding back into vocabulary space — each position will now score how likely each vocab word is as the next token.

forward(self, idx, targets=None) — The Factory in Action

Step 1 — B, T = idx.size()

You give it a batch of sequences (idx).
This gets:
B: batch size
T: number of tokens in each sample (sequence length)

Step 2 — pos = torch.arange(0, T, device=idx.device)

This makes a tensor [0, 1, 2, ..., T-1] representing each token's position in the sequence.

Step 3 — x = self.drop(self.token_emb(idx) + self.pos_emb(pos))

Let's say:

- idx = [[4, 17, 99]] → tokens like "The cat sat"
- token_emb(idx) → shape [B, T, 512]
- pos_emb(pos) → shape [T, 512] → broadcast to [B, T, 512]

We add token meaning + position, then apply dropout.

Step 4 — Pass through Transformer Blocks

```
for blk in self.blocks:
        x = blk(x)
```

This is where the magic happens :

- Each block refines the representation of every token.
- Tokens attend to each other causally (left-only).
- MLPs add depth, normalization keeps things stable.

After n layers, each token now has a deep contextual understanding.

Step 5 — Final LayerNorm

```
x = self.ln_f(x)
```

Smooths out final token representations.
Shape still [B, T, 512].

Step 6 — Output Projection: `logits = self.head(x)`

Mathematically:

For each token embedding vector $x\_i \in R^{\mathbf{d}}$

$logits\_i = x\_i \times W^{\top}$; where $W \in R^{\mathbf{vocab\_size \times d}}$

Step 7 — Loss Computation

```
if targets is not None:
    loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))
```

Compute cross-entropy between predicted logits and ground truth tokens.

- logits.view(-1, vocab_size) flattens to [B*T, vocab_size]
- targets.view(-1) flattens to [B*T]
- Calculates how wrong the model is — backprop will fix it!

Tokens → TokenEmb + PosEmb → Dropout

↓

[Transformer Block 1]

↓

[Transformer Block 2]

↓

...

↓

[Transformer Block n]

↓

Final LayerNorm → Linear (head) → Logits → (Loss if targets)

Assessment Question

1 . Explain the role of each of the following components in the GPT model architecture:

- nn.Embedding
- nn.LayerNorm
- nn.ModuleList([Block(cfg) for _ in range(cfg.n_layer)])
- nn.Linear(cfg.n_embd, cfg.vocab_size)

How do these work together in the forward pass?

2 . Imagine you're building GPT for a language with a massive vocabulary (e.g., Chinese).

- What changes would you consider in token_emb, head, and model capacity?
- What would be the trade-offs between model size and performance?

3 . What is the mathematical purpose of the MLP after attention?