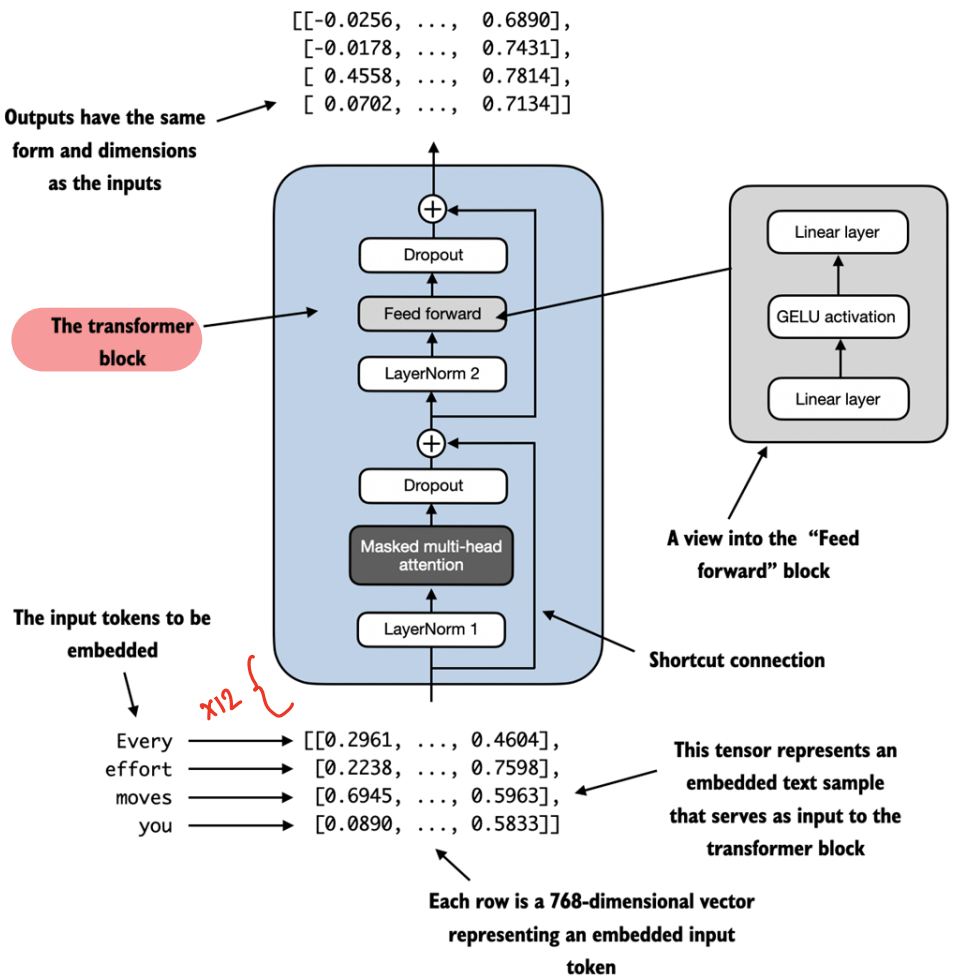
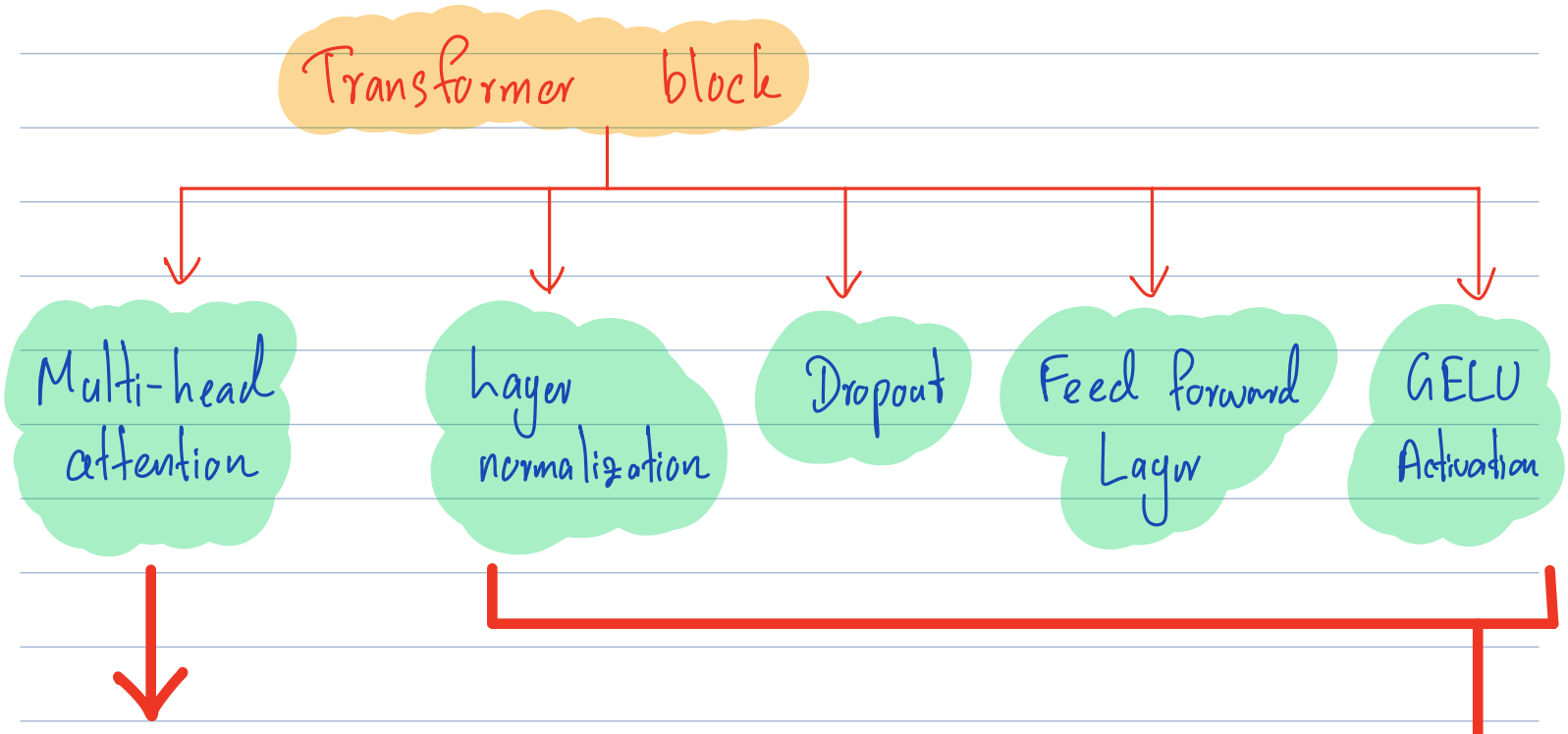
Transformer Block: The Fundamental Building Block of GPT and Other LLMs.

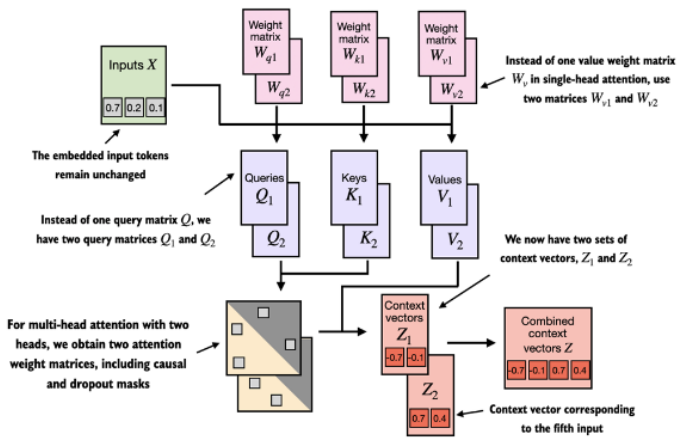
The Transformer block is the core unit powering GPT and modern large language models (LLMs). It combines multi-head attention to capture relationships between words, layer normalization for stable training, dropout to prevent overfitting, and a feedforward layer with GELU activation for powerful, non-linear transformations. Stacking these blocks enables models to understand and generate human-like text with remarkable accuracy.



① The transformer block is repeated 12 times in GPT-2 small (124M parameters)



Multi-head attention



Multi-head attention is a key mechanism in transformers that helps models like GPT and BERT understand language better. It works by letting the model focus on different parts of the input at the same time, capturing richer relationships between words.

Here's how it works in brief:

- The input is transformed into queries, keys, and values using learned weight matrices.
- These are split into multiple attention heads, where each head looks at the data from a different angle.
- The model calculates attention scores to decide which words should focus on which others.
- It applies softmax to turn these scores into attention weights, sometimes masking future words to prevent "peeking ahead".
- These weights are used to create context vectors that summarize important information.
- Finally, context vectors from all heads are combined to form a rich representation that the model uses to predict or understand the next steps.

Layer Normalization

① Training deep neural network with many layers can be challenging due to Vanishing / exploding gradients

② This leads to unstable training dynamics.

③ Layer normalization improves the stability & efficiency of neural network training

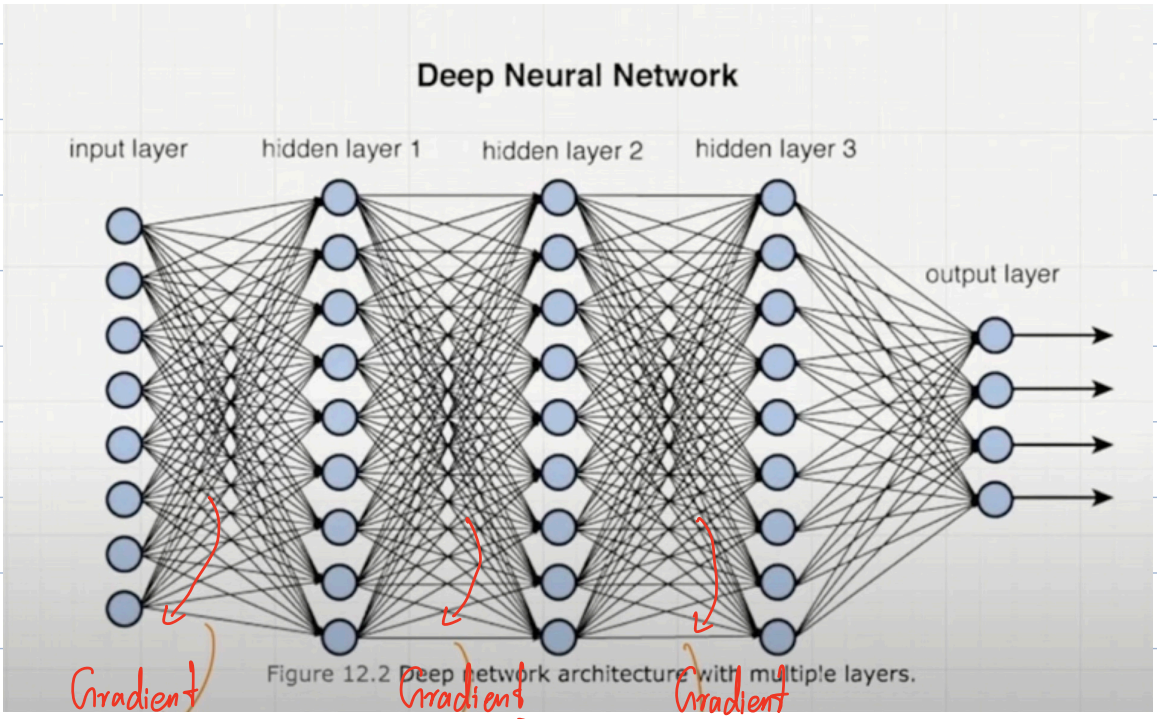


Figure 12.2 Deep network architecture with multiple layers.

Depends on layer output

As the training proceeds, inputs to each layer can change (internal covariate shift)

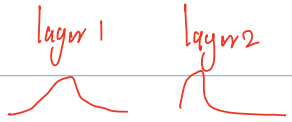
If layer output is too larger or small, gradient magnitude can become too large or small.

↓

This affects training

↓

Layer normalization keeps gradient stable



↓

This delays convergence

↓

Layer normalization prevents this

④ Main idea Adjust outputs of neural network to have mean zero & variance of one

x_1 x_2 x_3 x_4

$[1.1, 0.8, 2.3, 4.4]$

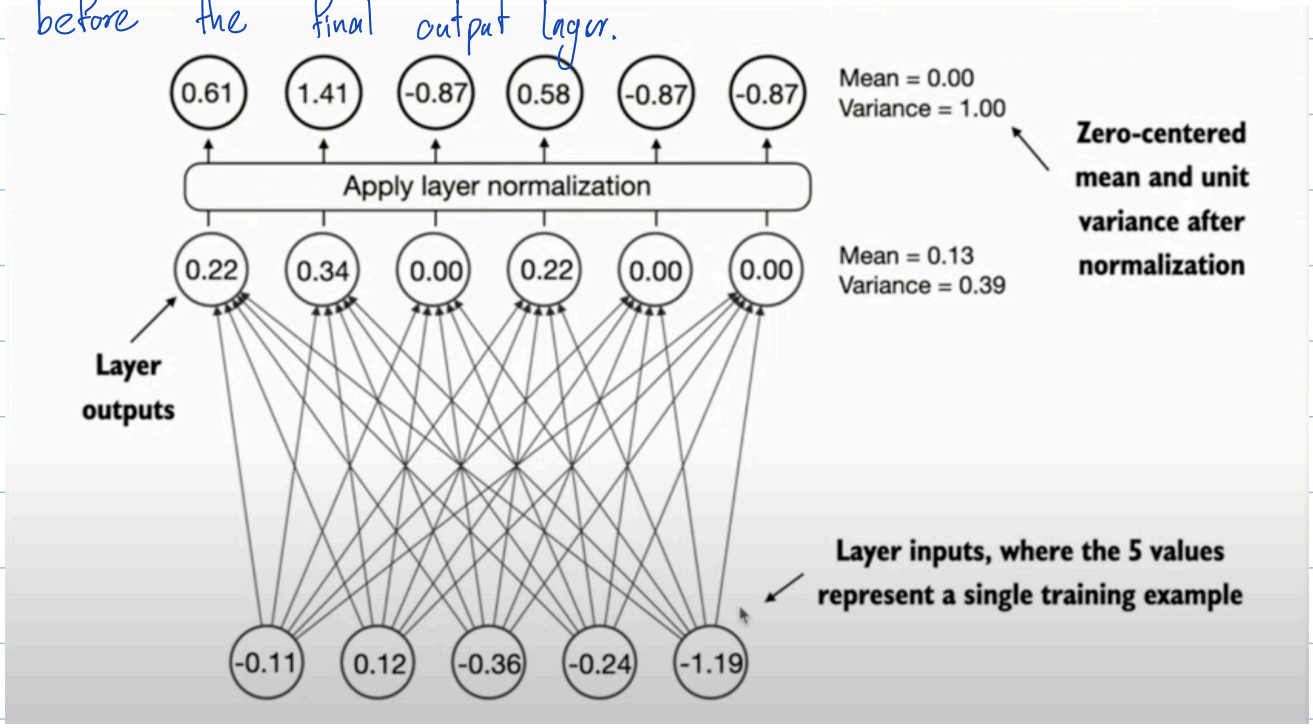
$$\text{mean} = \mu = \frac{x_1+x_2+x_3+x_4}{4} = 2.15$$

$$\text{Var} = \text{var} = \frac{1}{4}\left[(x_1-\mu)^2+(x_2-\mu)^2+(x_3-\mu)^2+(x_4-\mu)^2\right] = 2.002$$

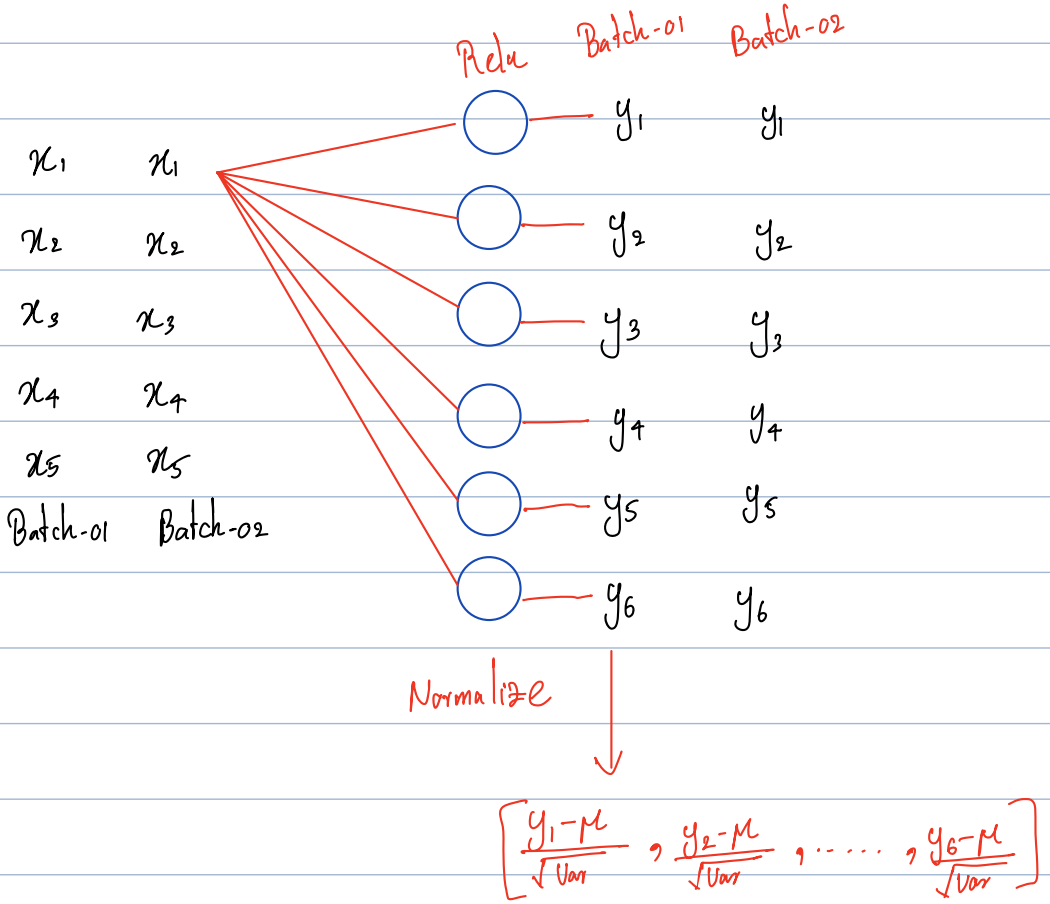
$$\text{Normalized}: \left[\frac{x_1-\mu}{\sqrt{\text{Var}}}, \frac{x_2-\mu}{\sqrt{\text{Var}}}, \frac{x_3-\mu}{\sqrt{\text{Var}}}, \frac{x_4-\mu}{\sqrt{\text{Var}}}\right] =$$

$$= [-0.74, -0.95, 0.1, 1.59]$$

⑤ In GPT-2 & modern transformers architectures, layer normalization is typically applied before & after the multi-head attention module & before the final output layer.



⑥



⑦ Layer vs Batch normalization

Normalized along with feature demention

independacy of batch size.

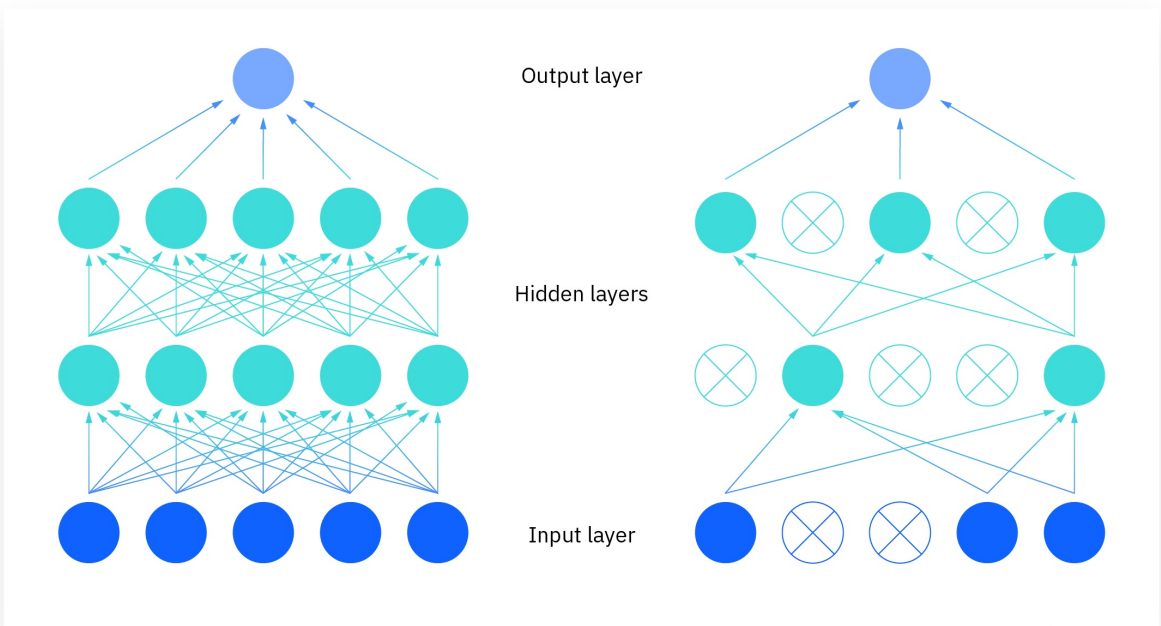
* doesn't depend on the batch size along

depend on the batch size

leads to more flexibility & stability for distributed training environment which lack of resources.

Dropout

Dropout is a regularization technique used to prevent overfitting. In the Transformer block, it randomly "drops" (sets to zero) some of the output values after the multi-head attention layer and before the feedforward network during training. This helps the model learn more robust patterns by not relying too heavily on any single part of the data.



Before Dropout

After Dropout

Feed Forward Layer + GELU activation

Feedforward Layer:

This layer adds extra “thinking power” to the model by processing each position separately after attention. In GPT-2, it first expands the input size (from 768 to 3072 dimensions — 4x wider), applies activation, and then compresses it back to 768 dimensions. This two-step expansion and contraction lets the model capture richer and more complex patterns, beyond just sequence relationships.

🔎 Tip: Even though attention links words together, the feedforward layer deepens the understanding at each word position.

GELU Activation:

GELU (Gaussian Error Linear Unit) is the brain behind the feedforward layer’s non-linearity. Instead of just cutting off negative values like ReLU, GELU softly weights inputs — small values are partly passed, large values go through fully. This smooth curve helps the model learn subtle patterns better, making it especially useful in large language models.

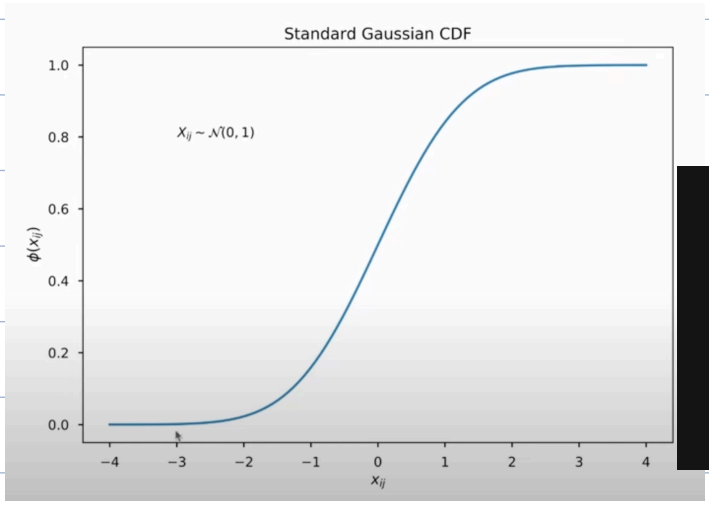
🔎 Fun fact: GELU is used in GPT and BERT because it boosts performance over simpler functions in deep networks.

Implement a small neural network sub-module that is a part of the LLM transformer block.

Two activation function commonly implemented in LLMs

→ ① GELU

$x * \phi(x)$ → Cumulative Function of standard Gaussian distribution



very high values of $x \to 1$

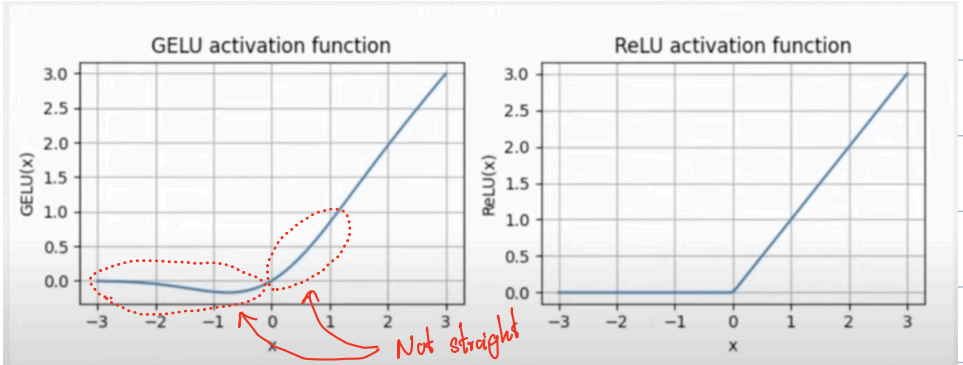
```python
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

Here is an approximation for GELU activation function used for training GPT-2

$$GELU(x) \simeq 0.5 \cdot x \cdot (1 + \tanh[\sqrt{(2/\pi)} \cdot (x + 0.044715 \cdot x^3)])$$

Difference between GELU & ReLU




```python
import matplotlib.pyplot as plt

gelu, relu = GELU(), nn.ReLU()

# Some sample data
x = torch.linspace(-3, 3, 100)
y_gelu, y_relu = gelu(x), relu(x)

plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)

plt.tight_layout()
plt.show()
```

Advantages of GELU

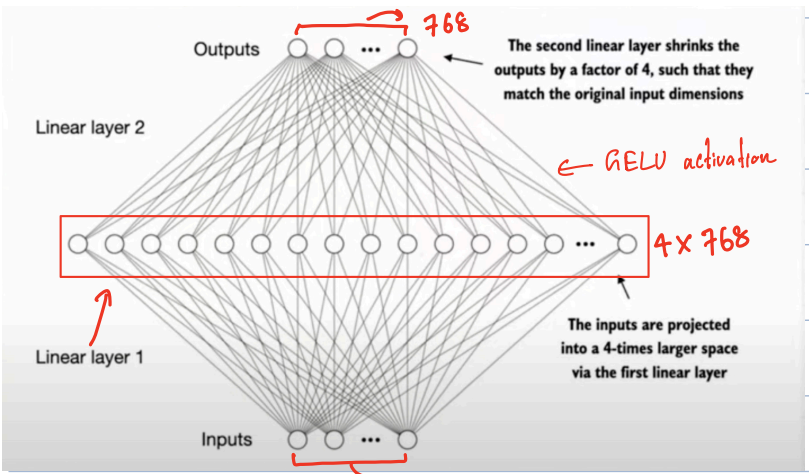
① Smooth & Differentiable

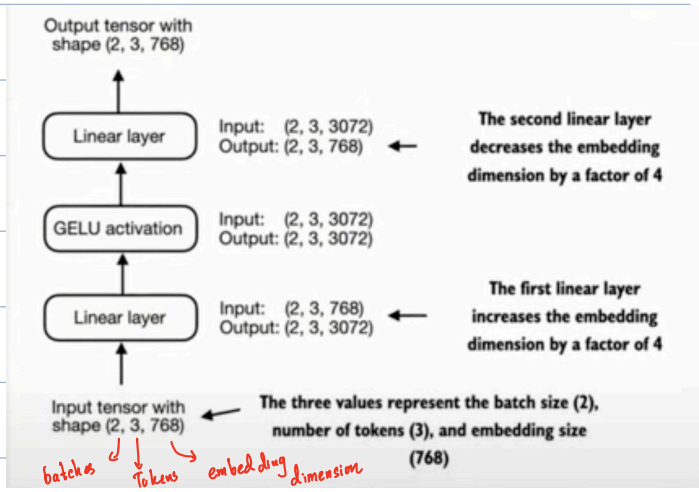
② Not zero for (-x), solve dead neurone problem

③ Work better than ReLU

Feed Forward Network

By projecting to large dimension (4 times):-
Can capture more propatives between input






```python
GPT_CONFIG_124M = {
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 1024, # Context length
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.1,       # Dropout rate
    "qkv_bias": False       # Query-Key-Value bias
}
```

```python
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

```python
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768) #A
out = ffn(x)
print(out.shape)
```

Shortcut Connection

A shortcut connection skips over layers and adds the input directly to the output. In the Transformer block, it helps the model train better and faster by:

- Preventing important information from getting lost as it flows through layers
- Helping avoid the vanishing gradient problem, so gradients stay strong during backpropagation
- Making it easier to stack many layers without hurting performance

🔎 Think of it like a fast lane that keeps the original information alive while new features are added.

① Shortcut connections first propose in the field of Computer Vision to solve the vanishing gradient problem.

gradient became progresively smaller as they propagrate backwards → making it difficult to train earlier layer.

$$w^{*} = w^{old} - \alpha \frac{\partial L}{\partial w}$$

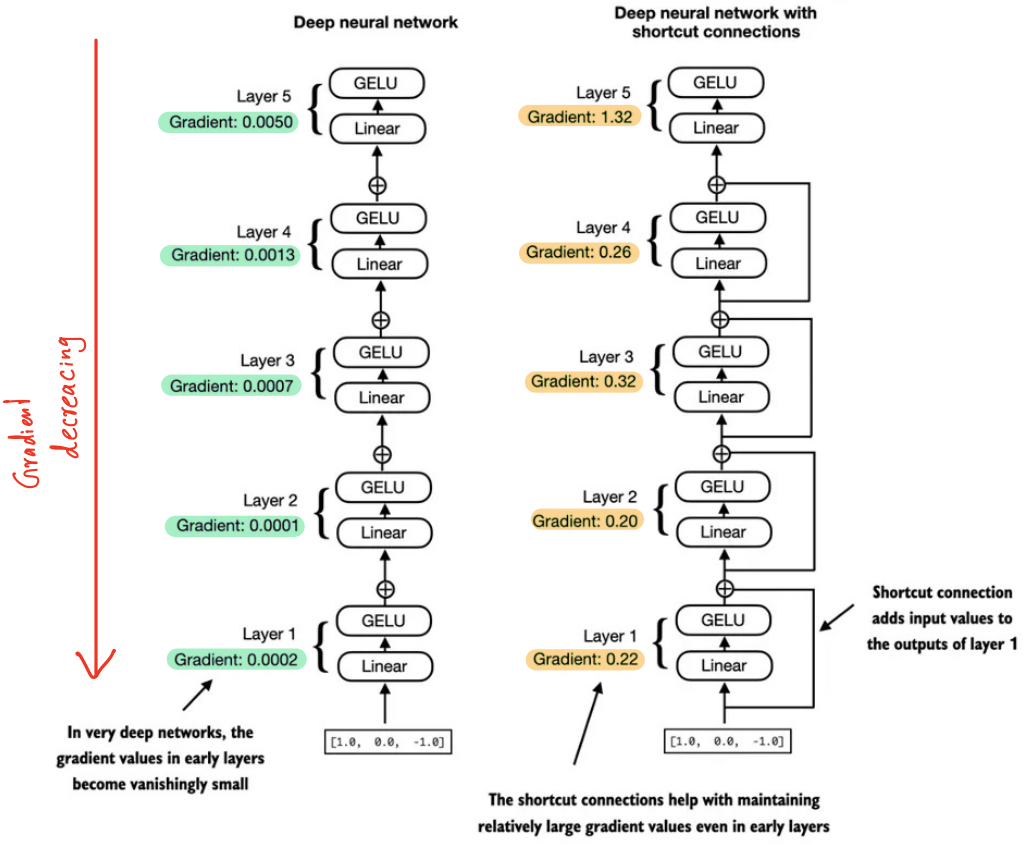
$\frac{\partial L}{\partial w} \approx 0$

weights are not updating

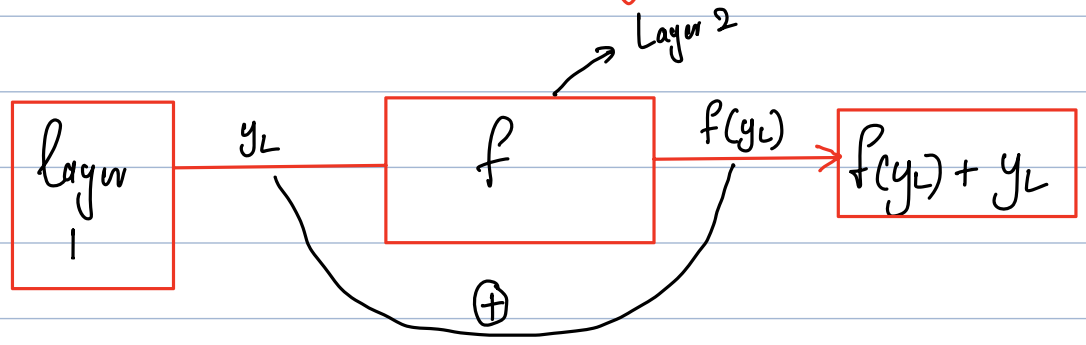
∴ not learning.

② Shortcut connections create an alternative path for the gradient to flow, by skipping one or more layers.

This is achieved by adding the out of one layer to the output of latter layer. → also called skip connections



Mathematical Understanding.



$$y_{L+1} = f(y_L) + y_L$$

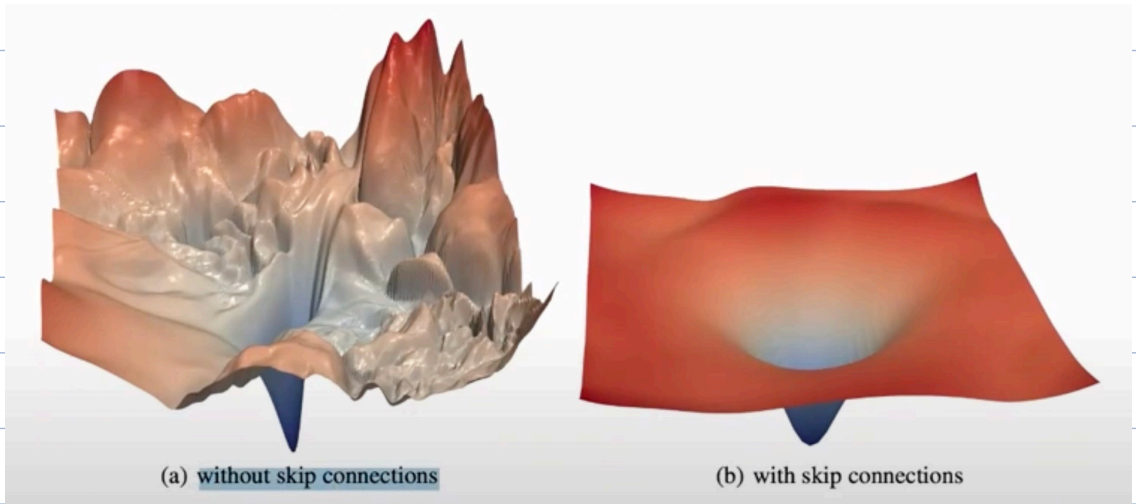
$$\frac{\partial L}{\partial y_L} = \frac{\partial L}{\partial y_{L+1}} \left(\frac{\partial y_{L+1}}{\partial y_L} \right) \rightarrow \frac{\partial [f(y_L) + y_L]}{\partial y_L}$$

$$= \frac{\partial L}{\partial y_{L+1}} \left(\frac{\partial f(y_L)}{\partial y_L} + 1 \right)$$

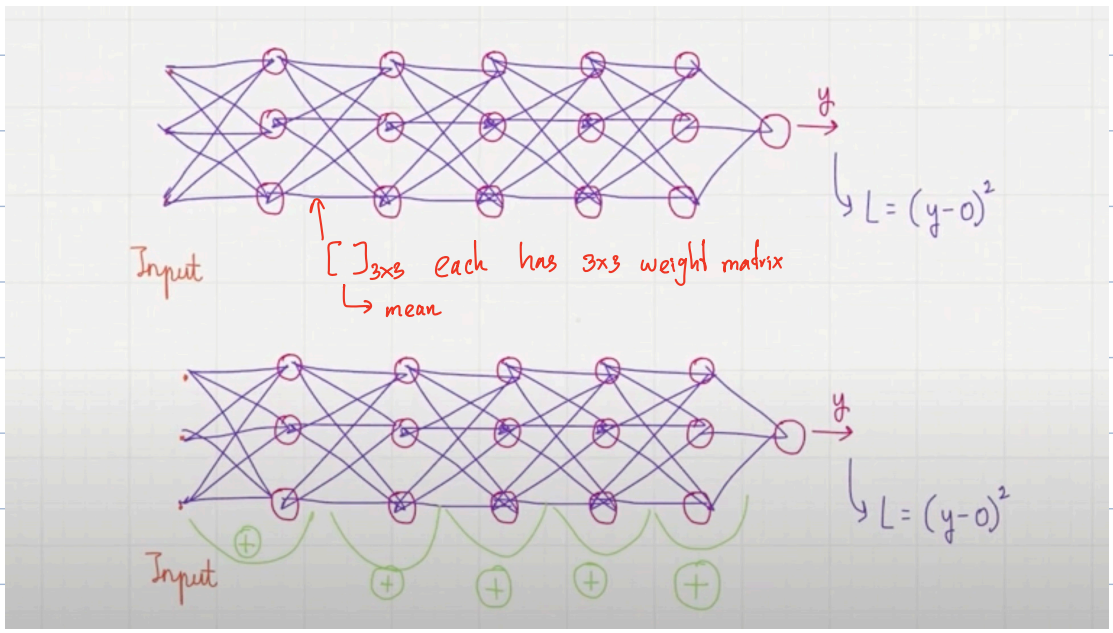
keeps the gradient flowing through the network.

if ≈ 0

$(+1)$ remains



(a) without skip connections (b) with skip connections



When transformer block process an input sequence, each element is represented by a fixed size vector.

The operation within the transformer block such as multihead attention & feed forward layer are design to transform the input vector such that their dimensionality is preserved.

Self attention ⟶ Analyzed relationship between input elements

Feed Forward Network ⟶ modifies data individually at each position

Full Code

```python
GPT_CONFIG_124M = {
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 1024, # Context length
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.1,       # Dropout rate
    "qkv_bias": False       # Query-Key-Value bias
}
```

```python
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        # Shortcut connection for attention block
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)  # Shape [batch_size, num_tokens, emb_size]
        x = self.drop_shortcut(x)
        x = x + shortcut  # Add the original input back

        # Shortcut connection for feed forward block
        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut  # Add the original input back

        return x
```