1.3 Deep Learning Interview Guide: Activation Functions

> This guide provides a comprehensive overview of activation functions in neural networks, a fundamental topic for any data science or machine learning interview. We begin by establishing the role of activation functions in introducing non-linearity and explore the critical issues of vanishing and exploding gradients. The guide then details a wide range of functions, from classic ones like Sigmoid and Tanh to the modern industry-standard ReLU and its variants (Leaky ReLU, ELU, PReLU). Finally, we delve into advanced activations like GELU, Swish, and the Gated Linear Unit (GLU) family, which are pivotal to the performance of state-of-the-art models like GPT, BERT, and LLaMA.

Knowledge Section

1. The Role of Activation Functions

At its core, a neural network is a series of linear transformations (matrix multiplications and additions). If we stack these linear layers without any intermittent non-linear function, the entire network would be equivalent to a single, larger linear transformation.

$$\text{Layer}_1(x) = W_1x + b_1$$
$$\text{Layer}_2(\text{Layer}_1(x)) = W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2) = W'x + b'$$

Such a model, regardless of its depth, could only learn linear relationships, making it incapable of modeling the complex, non-linear patterns found in real-world data like images, text, and sound.

Activation functions are mathematical functions applied to the output of each neuron (or layer) to introduce non-linearity into the network. This allows the model to learn arbitrarily complex functions and represent intricate data distributions.

2. The Vanishing and Exploding Gradient Problem

This is a critical challenge in training deep neural networks, primarily arising during backpropagation. Backpropagation computes gradients of the loss function with respect to

the network's weights using the chain rule.

Let L be the loss and W_i be the weight matrix of layer i. The gradient for a weight in an early layer is a product of many terms:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a_n} \frac{\partial a_n}{\partial z_n} \frac{\partial z_n}{\partial a_{n-1}} \cdots \frac{\partial a_2}{\partial z_2} \frac{\partial z_2}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial W_1}$$

Each term $\frac{\partial a_i}{\partial z_i}$ is the derivative of the activation function at layer i.

- **Vanishing Gradients:** If the derivatives of the activation function are consistently less than 1, their product will shrink exponentially as it propagates to earlier layers. The gradients for the initial layers become vanishingly small, causing their weights to update very slowly or not at all. This effectively "freezes" the early layers, preventing the network from learning. This is a notorious problem for `Sigmoid` and `Tanh` functions.
- **Exploding Gradients:** Conversely, if the gradients are consistently greater than 1, their product will grow exponentially. This results in massive, unstable weight updates, causing the model's loss to diverge (often to `NaN`). While less common with modern activations, it can still occur, especially in Recurrent Neural Networks (RNNs).

Solutions and Mitigation Strategies

1. **Choose a Better Activation Function:** Using `ReLU` and its variants largely solves the vanishing gradient problem for positive inputs, as their derivative is 1.
2. **Weight Initialization:** Proper initialization schemes like Xavier/Glorot or He initialization set the initial weights to have a variance that keeps the signal and gradients in a reasonable range, preventing them from immediately vanishing or exploding.
3. **Batch Normalization (and Layer Normalization):** These techniques normalize the activations within a layer to have a mean of 0 and a standard deviation of 1. This keeps the inputs to subsequent layers in a stable, well-behaved range, significantly mitigating the vanishing/exploding gradient problem and speeding up training.
4. **Residual Connections (ResNets):** By adding "skip connections" that allow the gradient to flow directly through an identity path, ResNets ensure that gradients

can propagate through very deep networks without diminishing.

5. **Gradient Clipping:** This is a direct solution for exploding gradients. If the L2-norm of the gradient vector exceeds a certain threshold, the vector is scaled down to have a norm equal to the threshold. This prevents excessively large weight updates.

3. Classic Activation Functions

These were foundational but are now less common in hidden layers of deep feed-forward networks, though they remain useful in specific contexts.

3.1. Sigmoid

The Sigmoid function maps any real-valued number into the range `(0, 1)`.

Formula:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Pros:

- **Probabilistic Interpretation:** Its `(0, 1)` output range makes it ideal for the final output layer in binary classification problems, where it can represent the probability of the positive class.
- **Smooth and Differentiable:** The function is smooth, which results in a smooth gradient for optimization.

Cons:

- **Vanishing Gradients:** The derivative of the Sigmoid function has a maximum value of 0.25. In deep networks, multiplying these small values together during backpropagation causes gradients in early layers to become extremely small, leading to the vanishing gradient problem.
- **Not Zero-Centered:** The output is always positive (between 0 and 1). This means the gradients flowing to the weights of a subsequent layer will always have the

same sign (either all positive or all negative, depending on the upstream gradient). This can lead to inefficient, zig-zagging gradient updates.
- **Computationally Expensive:** The exponential function (`e^z`) is more computationally intensive than the simple operations used in functions like ReLU.

3.2. Tanh (Hyperbolic Tangent)

Tanh is a rescaled version of the Sigmoid function, mapping inputs to the range `(-1, 1)`.

Formula:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

Derivative:

$$\tanh'(z) = 1 - \tanh^2(z)$$

Pros:

- **Zero-Centered Output:** The `(-1, 1)` output range means the function's output is centered around 0. This alleviates the non-zero-centered issue of Sigmoid, often leading to faster convergence.
- **Steeper Gradient:** The derivative of Tanh ranges from `(0, 1]`, which is steeper than Sigmoid's `(0, 0.25]`. This provides slightly stronger gradients, though it doesn't fully solve the vanishing gradient problem.

Cons:

- **Vanishing Gradients:** Like Sigmoid, Tanh saturates for large positive or negative inputs, causing the gradient to become near-zero and still leading to the vanishing gradient problem in deep networks.
- **Computationally Expensive:** It also relies on the exponential function.

4. The ReLU Family: The Modern Standard

ReLU and its variants have become the default choice for hidden layers in most deep learning applications.

4.1. ReLU (Rectified Linear Unit)

ReLU is a simple yet powerful function that outputs the input directly if it's positive, and zero otherwise.

Formula:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

Derivative:

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

(Note: The derivative is technically undefined at z=0, but a value of 0 or 1 is assigned in practice).

Pros:

- **Solves Vanishing Gradients (for z>0):** For positive inputs, the gradient is a constant 1. This allows gradients to flow unimpeded during backpropagation, enabling the training of much deeper networks.
- **Computationally Efficient:** It involves a simple thresholding operation, which is much faster than the exponentials in Sigmoid/Tanh.
- **Induces Sparsity:** By setting negative inputs to zero, ReLU can cause many neurons to output zero for a given input. This creates sparse representations, which can be computationally efficient and may have representational benefits.
- **Faster Convergence:** In practice, networks with ReLU converge significantly faster than those with Sigmoid or Tanh.

Cons:

- **The "Dying ReLU" Problem:** If a neuron's weights are updated such that its input is always negative, it will always output zero. Consequently, the gradient flowing through it will also always be zero. The neuron becomes "stuck" and effectively "dies," as its weights will never be updated again. This can be caused

by a large learning rate or poor weight initialization.
- **Not Zero-Centered:** Similar to Sigmoid, the output is non-negative, which can be a minor drawback compared to Tanh or ELU.
- **Potential for Exploding Gradients:** Since the output is unbounded for positive values, it's theoretically possible for activations to grow very large, although this is often managed by other techniques like Batch Normalization.

4.2. Leaky ReLU

Leaky ReLU is a simple modification to fix the "Dying ReLU" problem.

Formula:

$$\text{Leaky ReLU}(z) = \max(\alpha z, z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}$$

Where α is a small, fixed constant, typically 0.01.

Pros:

- **Prevents Dying Neurons:** By providing a small, non-zero gradient for negative inputs, it allows "dead" neurons to potentially be revived.

Cons:

- **Inconsistent Performance:** The benefit of Leaky ReLU over standard ReLU is not always consistent across different tasks and architectures. The choice of the hyperparameter α can also be problematic.

4.3. PReLU (Parametric ReLU) & RReLU (Randomized ReLU)

These are variants of Leaky ReLU that treat the slope α differently.

- **PReLU:** Makes α a **learnable parameter** that is updated via backpropagation along with the model's weights. The network learns the best slope for its negative inputs.
- **RReLU:** In training, α is a **random number** sampled from a uniform distribution. During testing, it is fixed.

These aim to improve upon the fixed, arbitrary choice of α in Leaky ReLU, but they are less commonly used than standard ReLU or ELU.

4.4. ELU (Exponential Linear Unit)

ELU is another alternative to ReLU that aims to produce outputs closer to zero-mean and to avoid the dying neuron problem.

Formula:

$$\text{ELU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases}$$

Where α is a positive hyperparameter, typically set to 1.0.

Pros:

- **Alleviates Dying ReLU Problem:** Has a non-zero gradient for negative inputs.
- **Outputs Closer to Zero-Mean:** The negative part of the function pushes the mean of the activations closer to zero, which can speed up learning (similar to Tanh).
- **Robustness to Noise:** The saturation for negative inputs can make the model more robust to noise.

Cons:

- **Computationally Expensive:** Involves the exponential function, making it slower than ReLU.

5. Advanced Activations for Transformers and LLMs

Modern large-scale models, particularly in the Transformer architecture, have favored smoother and more complex activation functions. Information up-to-date as of early 2024.

5.1. GELU (Gaussian Error Linear Unit)

GELU is the standard activation function used in models like BERT, GPT-2, GPT-3, and other foundational Transformers. It weights an input by its value, but this weighting is stochastically

modulated by the probability of it being greater than other inputs under a standard Gaussian distribution.

Intuition: The function combines properties of dropout, zoneout, and ReLUs. It stochastically determines whether to output the input (identity mapping) or zero (zero mapping). This choice is not binary but probabilistic, based on the input's magnitude. A larger positive input has a higher probability of being preserved, while a larger negative input has a higher probability of being zeroed out.

Formula:
The exact formula involves the Gaussian Cumulative Distribution Function (CDF), $\Phi(x)$.

$$\text{GELU}(x) = x \cdot \Phi(x) = x \cdot P(X \leq x), \quad \text{where } X \sim \mathcal{N}(0, 1)$$

Since the CDF is expensive to compute, a fast approximation is often used:

$$\text{GELU}(x) \approx 0.5x \left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right)$$

Pros:

- **Smooth and Non-Convex:** Unlike ReLU, GELU is a smooth curve, which can be beneficial for gradient-based optimization.
- **Probabilistic Motivation:** Its stochastic formulation provides a nice theoretical grounding.
- **High Performance:** Empirically shown to outperform ReLU in many NLP tasks, especially within the Transformer architecture.

Cons:

- **Computationally Intensive:** Even the approximation is more complex than ReLU.

5.2. Swish

Swish was discovered through automated architecture search and is remarkably simple yet effective.

Formula:

$$\text{Swish}(x) = x \cdot \sigma(\beta x)$$

Where σ is the Sigmoid function and β is a constant or a learnable parameter. When $\beta = 1$, it's often called SiLU (Sigmoid-weighted Linear Unit).

Pros:

- **Smooth and Non-Monotonic:** The function dips slightly for negative values before rising back towards zero. This non-monotonic "bump" may allow the model to learn more complex patterns.
- **Unbounded Above, Bounded Below:** Avoids saturation for positive values (like ReLU) while providing regularization and preventing large negative activations.
- **Outperforms ReLU:** Often shows slight but consistent performance improvements over ReLU on deep models.

5.3. GLU (Gated Linear Unit) and its Variants

GLU introduces a gating mechanism that allows the network to dynamically control the flow of information. This has proven extremely effective in the Feed-Forward Network (FFN) blocks of modern LLMs.

Original GLU Formula:

$$\text{GLU}(X) = (XW + b) \otimes \sigma(XV + c)$$

Where X is the input, W, b, V, c are learnable parameters, and $\otimes$ denotes element-wise multiplication. One part of the input $(XW + b)$ is the content, while the other part $\sigma(XV + c)$ acts as a "gate" that controls how much of the content is passed through.

SwiGLU (Used in PaLM, LLaMA):
A popular and powerful variant replaces the Sigmoid gate with a Swish activation. The FFN block is typically modified as follows:

$$\text{FFN}_{\text{SwiGLU}}(x) = (\text{Swish}_1(xW_1) \otimes (xW_2))W_3$$

This structure splits the input projection into two parts, one of which becomes a Swish-based

gate for the other. It has been shown to improve performance and training stability in very large language models.

Interview Questions

Theoretical Questions

Q1: Compare and contrast Sigmoid, ReLU, and GELU. Discuss their mathematical properties, computational costs, gradient behavior, and typical use cases.

Answer:

This is a classic question assessing breadth and depth. A good answer should be structured around these four points:

1. **Mathematical Properties & Shape:**
    - **Sigmoid:** Maps input to $(0, 1)$. It's a smooth, monotonic function that saturates at both ends.
    - **ReLU:** Maps input to $[0, \infty)$. It's a piece-wise linear function, non-smooth at the origin (kink), and monotonic. It is unbounded for positive inputs.
    - **GELU:** Maps input to approximately $(-0.17, \infty)$. It's a smooth, non-monotonic approximation of ReLU. It dips slightly for negative values before approaching zero.
2. **Gradient Behavior & Training Stability:**
    - **Sigmoid:** Its derivative is $\sigma(x)(1-\sigma(x))$, with a maximum value of 0.25. This leads to the **vanishing gradient problem** in deep networks, severely slowing down or halting training for early layers.
    - **ReLU:** Its derivative is 1 for positive inputs and 0 for negative inputs. The constant gradient of 1 for positive inputs effectively **solves the vanishing gradient problem**. However, the zero gradient for negative inputs leads to the **"Dying ReLU" problem**, where neurons can get stuck and stop learning.
    - **GELU:** Its gradient is smooth and non-zero everywhere. It doesn't

suffer from the dying neuron problem as severely as ReLU. Its smoothness can also lead to more stable optimization landscapes. It provides a good balance between the benefits of ReLU (no saturation for positive values) and avoiding its pitfalls.

3. **Computational Cost:**
    - **Sigmoid:** High cost due to the exponential function `e^-x`.
    - **ReLU:** Extremely low cost. It's just a `max(0, x)` operation, which is very fast on modern hardware.
    - **GELU:** High cost. The exact form requires computing the Gaussian CDF, and even the fast approximation involves `tanh` and polynomial terms, making it significantly slower than ReLU.
4. **Typical Use Cases:**
    - **Sigmoid:** Rarely used in hidden layers of modern deep networks. Its primary use case is in the **output layer for binary classification** to produce a probability. It is also used in the gates of legacy RNNs like LSTMs.
    - **ReLU:** The **default and most common activation function for hidden layers** in Convolutional Neural Networks (CNNs) and many other feed-forward architectures due to its efficiency and effectiveness.
    - **GELU:** The **standard activation for hidden layers in Transformer-based models** like BERT, GPT, and Vision Transformers (ViT). Its superior performance on massive NLP and vision tasks outweighs its computational cost in these large-scale models.

Q2: Derive the derivative of the Sigmoid function and mathematically explain why it leads to the vanishing gradient problem.

Answer:

1. Derivation:

Let the Sigmoid function be $\sigma(z) = \frac{1}{1+e^{-z}} = (1+e^{-z})^{-1}$.

We use the chain rule to find the derivative $\frac{d\sigma}{dz}$:

$$\frac{d\sigma}{dz} = -1 \cdot (1 + e^{-z})^{-2} \cdot \frac{d}{dz}(1 + e^{-z})$$

The derivative of the inner part is $\frac{d}{dz}(1 + e^{-z}) = -e^{-z}$.

Substituting this back:

$$\frac{d\sigma}{dz} = -1 \cdot (1 + e^{-z})^{-2} \cdot (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Now, we can rearrange this expression to show it in terms of $\sigma(z)$:

$$\frac{d\sigma}{dz} = \frac{1 + e^{-z} - 1}{(1 + e^{-z})^2} = \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \frac{1}{(1 + e^{-z})^2}$$

$$\frac{d\sigma}{dz} = \frac{1}{1 + e^{-z}} - \left(\frac{1}{1 + e^{-z}}\right)^2$$

Since $\sigma(z) = \frac{1}{1+e^{-z}}$, we have:

$$\frac{d\sigma}{dz} = \sigma(z) - \sigma(z)^2 = \sigma(z)(1 - \sigma(z))$$

2. Explanation of Vanishing Gradient:

To find the maximum value of the derivative, we can take its second derivative and set it to zero, or simply analyze the expression $\sigma(z)(1 - \sigma(z))$. This is a quadratic in terms of $\sigma(z)$, which is a parabola opening downwards. Its maximum value occurs when $\sigma(z) = 0.5$.

This happens when $z = 0$, because $\sigma(0) = \frac{1}{1+e^0} = \frac{1}{2}$.
The maximum value of the derivative is therefore:

$$\sigma'(0) = 0.5 \cdot (1 - 0.5) = 0.25$$

The derivative is always in the range $(0, 0.25]$.

In a deep network with n layers using the Sigmoid activation, the gradient for the first layer's weights will involve a product of $n-1$ such derivatives due to the chain rule:

$$\frac{\partial L}{\partial W_1} \propto \prod_{i=1}^{n-1} \sigma'(z_i)$$

Since each $\sigma'(z_i) \leq 0.25$, this product will shrink exponentially:

$$\text{Gradient} \propto (0.25)^{n-1}$$

For a deep network (e.g., $n = 10$), this product becomes vanishingly small ($0.25^9 \approx 3.8 \times 10^{-6}$). The gradients for the initial layers become so tiny that their weights are barely updated, effectively halting the learning process for those layers. This is the essence of the vanishing gradient problem caused by Sigmoid.

Q3: Explain the gating mechanism in SwiGLU. Why is this beneficial in modern Transformers like LLaMA?

Answer:

The FFN (Feed-Forward Network) block in a standard Transformer projects the input embedding x to a higher dimension, applies a non-linearity (like ReLU or GELU), and then projects it back down:

$$\text{FFN}_{\text{ReLU}}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

The SwiGLU Gating Mechanism:

SwiGLU fundamentally changes this structure by introducing a data-dependent gating mechanism. Instead of one large projection, it uses multiple smaller projections. A common formulation is:

$$\text{FFN}_{\text{SwiGLU}}(x) = (\text{Swish}(xW_1) \otimes (xW_2))W_3$$

Let's break this down:

1. **Two Projections:** The input x is projected twice in parallel using different weight matrices, W_1 and W_2.
2. **The "Gate":** The first projection, xW_1, is passed through a Swish activation function. The output, $\text{Swish}(xW_1)$, acts as the **gate**.
3. **The "Value":** The second projection, xW_2, is the **value** or content to be filtered.
4. **Element-wise Multiplication ($\otimes$):** The gate is multiplied element-wise with the value. For each dimension, the gate's value (which is near 0 for large negative inputs and large for large positive inputs) determines how much of the corresponding value's information is allowed to pass through. If a value in the gate is close to 0, it effectively "closes the gate" for that dimension, zeroing out the information. If it's large, it lets the information flow.
5. **Final Projection:** The resulting gated vector is then projected back down to the original dimension by W_3.

Benefits in Modern Transformers:

1. **Dynamic and Expressive Control of Information Flow:** Unlike ReLU, which applies a fixed threshold, the SwiGLU gate is **dynamic**. The gate's values are computed based on the input x itself. This allows the network to learn complex, context-dependent routing of information. It can decide, for each token and each dimension, which information from the FFN's hidden state is important and which should be suppressed. This is far more expressive than a static non-linearity.
2. **Improved Performance and Parameter Efficiency:** Papers like the PaLM paper and subsequent findings have shown that replacing the standard FFN with a SwiGLU variant leads to significant performance improvements for the same parameter count or computational budget. The gating mechanism seems to allow the model to utilize its parameters more effectively.
3. **Reduced Training Instability:** While requiring more matrix multiplications, the gating structure has been observed to contribute to more stable training for very large models. By selectively filtering information, it can prevent problematic activations from propagating and causing issues with gradients.

In summary, SwiGLU gives the FFN block the ability to act like a smart filter rather than just a simple non-linear transformer. This added expressiveness and control is a key reason for its adoption in state-of-the-art LLMs like LLaMA and PaLM.

Practical & Coding Questions

Q1: Implement and visualize common activation functions (Sigmoid, Tanh, ReLU, Leaky ReLU) and their derivatives using NumPy and Matplotlib.

Answer:

Here is a complete Python script to implement and plot these functions and their derivatives.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the input range
z = np.linspace(-10, 10, 200)

# --- 1. Activation Function Implementations ---

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x)**2

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

def leaky_relu(x, alpha=0.1):
    return np.maximum(alpha * x, x)

def leaky_relu_derivative(x, alpha=0.1):
    return np.where(x > 0, 1, alpha)

# --- 2. Plotting ---

# Create a figure and a set of subplots
fig, axes = plt.subplots(2, 4, figsize=(20, 10))
fig.suptitle('Common Activation Functions and Their Derivatives', fontsize=16)
```

```
# Plot titles
functions = ['Sigmoid', 'Tanh', 'ReLU', 'Leaky ReLU']
implementations = [
    (sigmoid, sigmoid_derivative),
    (tanh, tanh_derivative),
    (relu, relu_derivative),
    (leaky_relu, leaky_relu_derivative)
]

for i, (func_name, (func, deriv)) in enumerate(zip(functions, implementations)):
    # Plot the activation function
    ax1 = axes[0, i]
    ax1.plot(z, func(z), label='Function', color='blue')
    ax1.set_title(f'{func_name} Function')
    ax1.grid(True)
    ax1.spines['left'].set_position('center')
    ax1.spines['bottom'].set_position('center')
    ax1.spines['right'].set_color('none')
    ax1.spines['top'].set_color('none')
    ax1.xaxis.set_ticks_position('bottom')
    ax1.yaxis.set_ticks_position('left')
    ax1.set_ylim(-2, 2)

    # Plot the derivative
    ax2 = axes[1, i]
    ax2.plot(z, deriv(z), label='Derivative', color='red')
    ax2.set_title(f'{func_name} Derivative')
    ax2.grid(True)
    ax2.spines['left'].set_position('center')
    ax2.spines['bottom'].set_position('center')
    ax2.spines['right'].set_color('none')
    ax2.spines['top'].set_color('none')
    ax2.xaxis.set_ticks_position('bottom')
    ax2.yaxis.set_ticks_position('left')
    ax2.set_ylim(-0.5, 1.5)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

Q2: Implement the GELU activation function from scratch in PyTorch, including both the exact (using `erf`) and approximate versions.

Answer:

This question tests knowledge of the GELU formula and proficiency with PyTorch.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import math

# --- GELU Implementations ---

class GELU(nn.Module):
    """
    An implementation of the GELU activation function.
    This module allows for choosing between the 'exact' and 'approximate' versions.
    """
    def __init__(self, use_approximate=False):
        """
        Args:
            use_approximate (bool): If True, uses the tanh-based approximation.
                                    If False, uses the exact formula with the error functi
        """
        super().__init__()
        self.use_approximate = use_approximate

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        if self.use_approximate:
            # Tanh-based approximation from the paper
            # GELU(x) ≈ 0.5x * (1 + tanh(sqrt(2/π) * (x + 0.044715 * x^3)))
            return 0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x + 0.044715 *
        else:
            # Exact GELU using the error function (erf)
            # GELU(x) = x * Φ(x) = x * 0.5 * (1 + erf(x / sqrt(2)))
            return x * 0.5 * (1.0 + torch.erf(x / math.sqrt(2.0)))

# --- Verification and Comparison ---

# Create some input data
x = torch.linspace(-4, 4, 100)

# Instantiate our GELU modules
gelu_exact = GELU(use_approximate=False)
```

```
gelu_approx = GELU(use_approximate=True)

# Get the official PyTorch GELU for comparison
pytorch_gelu = nn.GELU()

# Calculate outputs
y_exact = gelu_exact(x)
y_approx = gelu_approx(x)
y_pytorch = pytorch_gelu(x) # PyTorch's default is the exact version

# --- Plotting for Visual Comparison ---
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(x.numpy(), y_exact.numpy(), label="My Exact GELU (erf)", linewidth=4, linestyle=
plt.plot(x.numpy(), y_approx.numpy(), label="My Approximate GELU (tanh)", linewidth=2, li
plt.plot(x.numpy(), y_pytorch.numpy(), label="Official PyTorch GELU", linewidth=2, linesty
plt.title("Comparison of GELU Implementations")
plt.xlabel("Input x")
plt.ylabel("GELU(x)")
plt.grid(True)
plt.legend()
plt.show()

# --- Check numerical difference ---
# The official pytorch GELU uses the exact method by default
diff = torch.max(torch.abs(y_exact - y_pytorch))
print(f"Maximum absolute difference between my exact GELU and PyTorch's GELU: {diff.item(

diff_approx = torch.max(torch.abs(y_exact - y_approx))
print(f"Maximum absolute difference between exact and approximate GELU: {diff_approx.item
```

This code not only provides the implementations but also verifies them against the official PyTorch `nn.GELU` and visually demonstrates the close correspondence between the exact and approximate formulas.