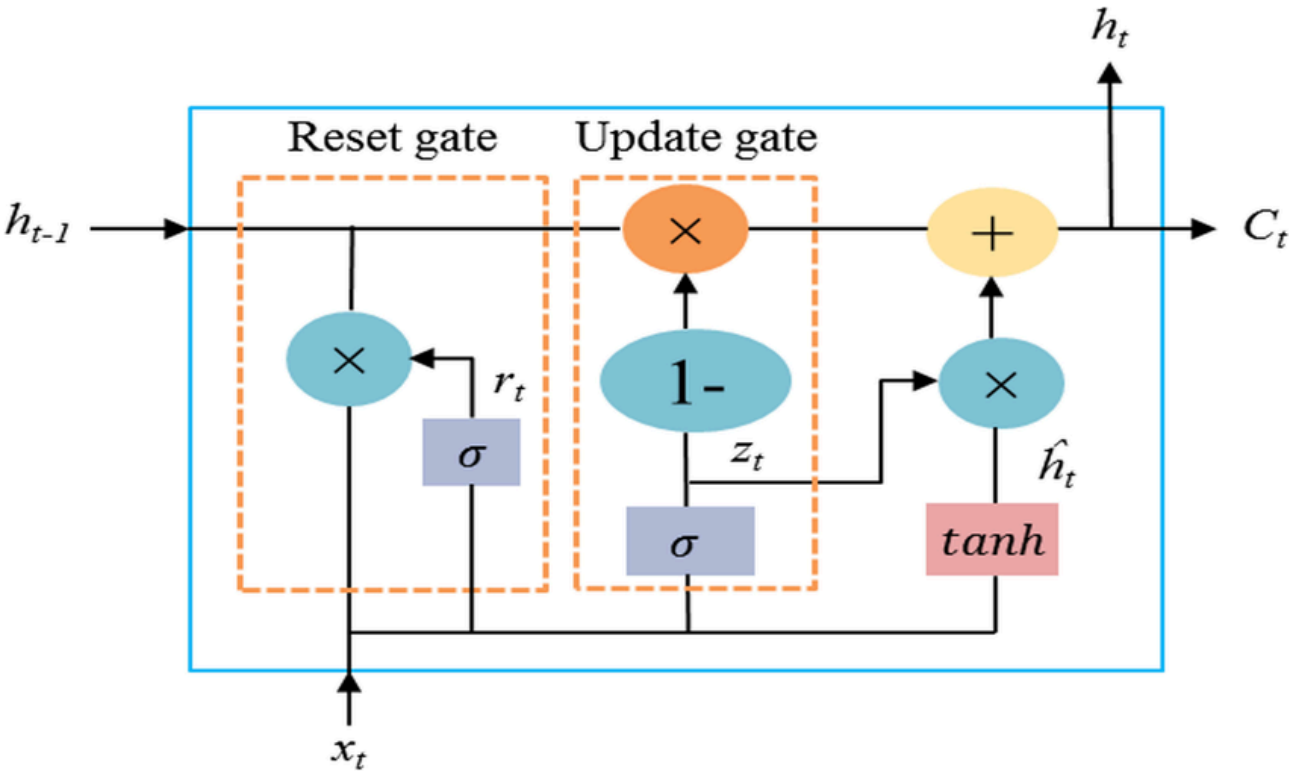
Understanding the Gated Recurrent Unit (GRU) Forward Pass

Introduction

The Gated Recurrent Unit (GRU) is a type of Recurrent Neural Network (RNN) that, like the Long Short-Term Memory (LSTM) network, is designed to solve the vanishing gradient problem that is common to traditional RNNs. This article provides a detailed walkthrough of the forward pass of a single GRU cell, explained using the Feynman technique to build a strong intuition for its inner workings.

Visualizing the GRU Cell

Below is a diagram illustrating the architecture of a GRU cell, showing the flow of information through the reset and update gates.



The GRU Equations

A GRU cell operates through a set of equations that manage the flow of information. These equations involve two main gates: the **update gate** and the **reset gate**.

Update Gate (z_t)

The update gate determines how much of the previous hidden state to keep and how much of the new candidate hidden state to incorporate. It is calculated as follows:

```
z_t = σ(x_t * W_z + h_{t-1} * U_z + b_z)
```

- **Intuition**: Think of the update gate as a controller for memory. If `z_t` is close to 1, the old memory is retained. If it is close to 0, the old memory is forgotten and replaced with new information.

Reset Gate (r_t)

The reset gate decides how much of the past information to forget. It is calculated as:

```
r_t = σ(x_t * W_r + h_{t-1} * U_r + b_r)
```

- **Intuition**: The reset gate allows the model to decide how much of the previous hidden state is relevant for computing the new candidate hidden state. If `r_t` is close to 0, the model effectively "resets" its memory, ignoring the previous state.

Candidate Hidden State (h_tilde_t)

The candidate hidden state is a new memory that is calculated based on the current input and the previous hidden state, with the influence of the reset gate:

```
h_tilde_t = tanh(x_t * W_h + (r_t ⊙ h_{t-1}) * U_h + b_h)
```

- **Intuition**: This is where the new information is generated. The reset gate `r_t` controls how much of the previous hidden state `h_{t-1}` is used to create this new candidate. The `tanh` activation function ensures that the output is between -1 and 1.

New Hidden State (h_t)

Finally, the new hidden state is a linear interpolation between the previous hidden state and the candidate hidden state, controlled by the update gate:

```
h_t = (1 - z_t) ⊙ h_{t-1} + z_t ⊙ h_tilde_t
```

- **Intuition**: This is the final output of the GRU cell. The update gate `z_t` acts as a blending factor, mixing the old memory with the new candidate memory.

Python Implementation

Here is a Python implementation of the GRU cell forward pass, with detailed explanations for each line of code.

```python
import numpy as np

def _sigmoid(x):
    """Numerically stable sigmoid function"""
    return np.where(x >= 0, 1.0/(1.0+np.exp(-x)), np.exp(x)/(1.0+np.exp(x)))

def _as2d(a, feat):
    """Convert 1D array to 2D and track if conversion happened"""
    a = np.asarray(a, dtype=float)
    if a.ndim == 1:
        return a.reshape(1, feat), True
    return a, False

def gru_cell_forward(x, h_prev, params):
    """
    Implement the GRU forward pass for one time step.
    Supports shapes (D,) & (H,) or (N,D) & (N,H).
    """

    # Convert 1D inputs to 2D for uniform processing
    x, was_1d_x = _as2d(x, x.shape[-1] if x.ndim > 0 else 1)
    h_prev, was_1d_h = _as2d(h_prev, h_prev.shape[-1] if h_prev.ndim > 0
else 1)

    # Extract dimensions
    N, H = h_prev.shape

    # Unpack parameters
    Wz, Uz, bz = params["Wz"], params["Uz"], params["bz"]
    Wr, Ur, br = params["Wr"], params["Ur"], params["br"]
    Wh, Uh, bh = params["Wh"], params["Uh"], params["bh"]

    # Update Gate: Decides how much of the old hidden state to keep.
    z_t = _sigmoid(x @ Wz + h_prev @ Uz + bz)

    # Reset Gate: Decides how much of the old hidden state to use for the
new candidate.
    r_t = _sigmoid(x @ Wr + h_prev @ Ur + br)

    # Candidate Hidden State: Computes the new information.
    h_tilde_t = np.tanh(x @ Wh + (r_t * h_prev) @ Uh + bh)

    # New Hidden State: Blends the old and new hidden states.
    h_t = (1 - z_t) * h_prev + z_t * h_tilde_t
```

```
    # Reshape output to match input format
    if was_1d_x or was_1d_h:
        h_t = h_t.reshape(-1)

    return h_t
```

How to Run the Code

You can test the implementation with the following code:

```
if __name__ == "__main__":
    # Test Case 1: All zeros with batch input
    x = np.zeros((2, 3))
    h_prev = np.array([[1.0, -1.0], [2.0, 0.0]])
    params = {
        "Wz": np.zeros((3, 2)), "Uz": np.zeros((2, 2)), "bz": np.zeros(2),
        "Wr": np.zeros((3, 2)), "Ur": np.zeros((2, 2)), "br": np.zeros(2),
        "Wh": np.zeros((3, 2)), "Uh": np.zeros((2, 2)), "bh": np.zeros(2),
    }
    output = gru_cell_forward(x, h_prev, params)
    print(f"Test 1 Output: {output}")

    # Test Case 2: 1D input with non-zero parameters
    x = np.array([0.5, -1.0, 0.0, 0.25, 0.75])
    h_prev = np.array([0.0, 0.1, -0.1, 0.2])
    np.random.seed(42)
    D, H = 5, 4
    params = {
        "Wz": np.random.randn(D, H) * 0.1, "Uz": np.random.randn(H, H) *
0.1, "bz": np.zeros(H),
        "Wr": np.random.randn(D, H) * 0.1, "Ur": np.random.randn(H, H) *
0.1, "br": np.zeros(H),
        "Wh": np.random.randn(D, H) * 0.1, "Uh": np.random.randn(H, H) *
0.1, "bh": np.zeros(H),
    }
    output = gru_cell_forward(x, h_prev, params)
    print(f"Test 2 Output: {output}")
```

Conclusion

The GRU is a powerful and efficient alternative to LSTMs for many sequence modeling tasks. By understanding the forward pass, you gain insight into how GRUs selectively update their memory, allowing them to capture long-range dependencies and avoid the vanishing gradient problem. This implementation provides a clear and concise example of how to build a GRU cell from scratch.