WIDS - UID85 - Implementing Generative AI Transformer

23b0986
Tanishq Mandhane

2024 - December

Worked Under Aditya Neeraje and Yash Sable
I want you to check my week 3 for wids project

Feedforward Neural Network

- First and simplest type of neural network.
- Information moves in only one direction.
- No cycles or loops.

If we include a loop, it becomes a **Recurrent Neural Network**.

$$a_{t+1} = \sigma(w_1 a_t + b_1)$$

Types of Neural Networks

- **Convolutional Neural Network** $\rightarrow$ Good for image recognition.
- **Long Short-Term Memory Network** $\rightarrow$ Good for speech recognition.

Neuron

A neuron is a thing that holds numbers.

Activation

Activation in one layer brings activation in the second layer.

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

Activation should be in range.

Learning

Learning $\Rightarrow$ Finding weights and biases.

Learning Model and Propagation

A learning model finds parameters (weights and biases) with the help of forward and backpropagation.

Forward Propagation

Forward Propagation $\rightarrow$ Input data is fed in the forward direction through the network.

$$\text{Input Data} \rightarrow \text{Activation Function} \rightarrow \text{Successive Layers}$$

Input data moves only in the forward direction. Such networks are called **Feedforward Networks**.

At each neuron in a hidden or output layer, processing happens in two steps:

1. **Preactivation** $\rightarrow$ The weighted sum of inputs is available. Based on this aggregated sum and activation function, we decide whether to pass this information further or not.

2. **Activation** $\rightarrow$ The calculated weighted sum of inputs is passed to an activation function such as sigmoid, hyperbolic tangent, ReLU, or softmax.

Activation Function

Instead of the sigmoid function, we can also use other activation functions.

Hyperbolic Tangent Activation

For hidden layers, the **tanh** function works better than sigmoid in many cases:

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

Activation Functions

Because the hyperbolic tangent (**tanh**) function makes the mean activation close to 0, it helps in making learning easier.

Choosing Different Activation Functions for Different Layers

The activation function can be different for each layer.

Issue: The slope of some activation functions (like sigmoid) is very small, which makes gradient descent slow. So, one more popular function is:

ReLU (Rectified Linear Unit)

$$a = \max(0, z)$$

Rules of Thumb

- If your output is binary (0/1), then **sigmoid** is the natural choice for the output layer.
- Otherwise, use **ReLU** or maybe **tanh**.

Since the slope of ReLU is 1 for half the time, it makes learning faster compared to other functions.

Pros and Cons of Activation Functions

- **Sigmoid** → Never use, except in specific cases.
- **tanh** → Always superior to sigmoid.
- **ReLU** → The default and most commonly used activation function.
- **Leaky ReLU** → Defined as:

$$a = \max(0.01z, z)$$

Conclusion

We need a **non-linear activation function** to enable complex learning in neural networks.

Optimization Problem

A neural network is trained using the **gradient descent** optimization algorithm. The weights are updated using the **backpropagation of error** algorithm. The optimization algorithm is navigated down the gradient of error.

Loss Function

With neural networks, we seek to **minimize error**. Such an objective function is called the **cost function**, and the value calculated by the cost/loss function is called **loss**.

Maximum Likelihood and Cross Entropy

$$\hat{y} = \sigma(w \cdot x)$$

$$J = \text{loss function} \quad ?$$

$$x \longrightarrow \hat{y} \qquad \begin{cases} y - \text{Ground truth} \\ \text{Cost} \end{cases}$$

What about Least Squares?

$$J^{LS} = \frac{1}{2}(y - \hat{y})^2$$

- Not a good cost function.
- The cost incurred for misclassification is low.

Binary Cross-Entropy Cost Function

$$J = -\left[y \ln \hat{y} + (1 - y) \ln(1 - \hat{y})\right]$$

- $\hat{y}$ follows the **sigmoid** function.
- $y \in \{0, 1\}$, $\hat{y} \in (0, 1]$.

Desirable Properties of a Loss Function

1. $J = 0$ for $y = \hat{y}$.
2. J should be **really high** for misclassification.
3. $J \geq 0$.

Loss Functions for Regression and Classification

For regression tasks, we use the **least squares** cost function, whereas for classification tasks, we use the **binary cross-entropy** cost function.

Backpropagation

We use the loss calculated from the loss function to make corrections to our model. For this, we use **gradient descent** and then update our parameters.

Calculus Behind Backpropagation

$$a^{(L-1)} \quad \longrightarrow \quad a^{(L)}$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

$$C_o(...) = (a^{(L)} - y)^2$$

Gradient Computation

$$\frac{\partial C_o}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_o}{\partial a^{(L)}}$$

$$\frac{\partial C_o}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

$$\frac{\partial C_o}{\partial w^{(L)}} = a^{(L-1)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

where n is the number of training examples. This represents the average over all training examples.

Gradient Computation

$$\frac{\partial C_o}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_o}{\partial a^{(L)}}$$

$$= 1 \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

$$\frac{\partial C_o}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_o}{\partial a^{(L)}}$$

$$= w^{(L)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

This is the backpropagation formula.

Case of Multiple Neurons in Each Layer

$$Z_j^{(L)} = \sum_k w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

$$a_j^{(L)} = \sigma(Z_j^{(L)})$$

$$C_o = \sum_{j=0}^{n_L - 1} (a_j^{(L)} - y_j)^2$$

Gradient Computation for Multiple Neurons

$$\frac{\partial C_o}{\partial w_{jk}^{(L)}} = \frac{\partial Z_j^{(L)}}{\partial w_{jk}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial Z_j^{(L)}} \cdot \frac{\partial C_o}{\partial a_j^{(L)}}$$

$$= a_k^{(L-1)} \cdot \sigma'(Z_j^{(L)}) \cdot 2(a_j^{(L)} - y_j)$$

$$\frac{\partial C_o}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L - 1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_o}{\partial a_j^{(L)}}$$

$$= \sum_{j=0}^{n_L - 1} w_{jk}^{(L)} \sigma'(z_j^{(L)}) \cdot 2(a_j^{(L)} - y_j)$$

Implementing a Complete Neural Network

Loss Function

$$J = \frac{1}{2} \sum (y - \hat{y})^2$$

Gradient of Loss with Respect to Weights

$$\frac{\partial J}{\partial w^{(3)}} = \sum -(y - \hat{y}) \frac{\partial \hat{y}}{\partial z^{(3)}}$$

$$\frac{\partial J}{\partial w^{(2)}} = \sum -(y - \hat{y}) f'(z^{(3)}) \frac{\partial z^{(3)}}{\partial w^{(2)}}$$

Matrix Form Representation

$$z^{(3)} = a^{(2)} w^{(2)}$$

$$\delta^{(3)} = \begin{bmatrix} -(y_1 - \hat{y}_1) f'(z_1^{(3)}) \\ -(y_2 - \hat{y}_2) f'(z_2^{(3)}) \\ -(y_3 - \hat{y}_3) f'(z_3^{(3)}) \end{bmatrix}$$

$$\frac{\partial J}{\partial w^{(2)}} = (a^{(2)})^T \delta^{(3)}$$

$$\frac{\partial J}{\partial w^{(1)}} = 2 \sum \frac{1}{2} (y - \hat{y})^2 = -(y - \hat{y}) \frac{\partial \hat{y}}{\partial w^{(1)}}$$

$$= -(y - \hat{y}) \frac{\partial a^{(2)}}{\partial w^{(1)}}$$

$$= -(y - \hat{y}) f'(z^{(3)}) \frac{\partial z^{(2)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial w^{(1)}}$$

$$\frac{e^x}{(1 + e^x)^2}$$

$$X^T \delta^{(3)} (w^{(1)})^T f'(z^{(2)})$$

$$\frac{e^x}{1 + e^x} \left(\frac{e^x}{1 + e^x} \right) \left(1 + e^{-x}\right) e^{-x}$$

Limitations of Neural Networks

Thank you for choosing neural nets, but it doesn't warrant:

1. Finding a good solution
2. Finding a solution in X iterations
3. Finding a solution at all

Relationship Between Neural Nets and Mathematical Optimization

$$\textit{training} = \text{Neural Nets} \cap \text{Mathematical Optimization}$$

Introduction

Transformers are a class of deep learning models that leverage self-attention mechanisms to process sequential data efficiently. They are widely used in natural language processing (NLP), text generation, and image synthesis.

Key Components

1. Encoder-Decoder Architecture

Transformers consist of an encoder and a decoder. The encoder processes input sequences, and the decoder generates outputs step-by-step using previous context.

2. Multi-Head Self-Attention

The self-attention mechanism allows the model to focus on different parts of the input sequence, improving contextual understanding.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

3. Positional Encoding

Since transformers do not process inputs sequentially like RNNs, positional encodings are added to maintain the order of tokens.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

4. Feedforward Network

Each transformer block contains a fully connected feedforward network (FFN) with activation functions.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

5. Layer Normalization and Residual Connections

Residual connections help gradient flow, and layer normalization stabilizes training.

6. Output Generation with Softmax

The final layer applies a softmax function to generate probabilities over the vocabulary.

$$P(y) = \text{softmax}(W_o h_T)$$

Applications

- Text Generation (GPT)
- Machine Translation (T5, BART)
- Code Generation (Codex)
- Image Synthesis (Stable Diffusion, DALL·E)