Natural Language Processing (NLP)

Transformers 2

Residual Connections and Layer Normalization

By:

Dr. Zohair Ahmed

www.youtube.com/@ZohairAI Subscribe

www.begindiscovery.com

Transformer

- In **Natural Language Processing (NLP)**, a **Transformer** is a deep learning architecture introduced in the paper *"Attention Is All You Need"* (Vaswani et al., 2017).
- https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf
- It revolutionized NLP by replacing recurrent and convolutional models with a mechanism called **self-attention**, enabling models to process sequences in parallel rather than sequentially.

Transformers: Parallel Processing

- **Transformers**, introduced in *"Attention is All You Need"*, use **self-attention mechanisms** that allow them to process all tokens **simultaneously**.
- They **do not rely on previous hidden states**, so the entire sequence can be fed in at once.
- This enables **massive parallelization**, especially on GPUs/TPUs, making training much faster.
- Positional encodings are used to retain sequence order information.

Feature	RNNs	Transformers
Processing Style	Sequential	Parallel
Dependency Modeling	Temporal (via hidden states)	Global (via attention)
Training Speed	Slower	Faster
Long-Range Context	Harder to capture	Easier via attention

Positional Encoding (PE)

- Transformers don't have recurrence (like RNNs) or convolution (like CNNs), so they **don't inherently know the position** of each token in a sequence.
- **Why sin/cos?:** They create **unique patterns** for each position.
- They allow the model to **generalize to longer sequences** (since sin/cos are continuous and periodic).
- They help the model **learn relative positions** (like "next word", "previous word").

Token	Position	Positional Encoding (simplified)
I	0	[sin(0), cos(0), sin(0), cos(0)] = [0, 1, 0, 1]
like	1	[$\sin(x_1)$, $\cos(x_1)$, $\sin(x_2)$, $\cos(x_2)$]
pizza	2	[$\sin(x_3)$, $\cos(x_3)$, $\sin(x_4)$, $\cos(x_4)$]

Positional Encoding (PE): Absolute Position

- **Absolute Position:** This refers to the exact index of a token in the sequence.
- For example, in "I like pizza":
  - "I" is at position 0
  - "like" is at position 1
  - "pizza" is at position 2
- In absolute positional encoding, each token gets a unique vector based on its fixed position. This is what sinusoidal encoding does it maps position 0, 1, 2, etc., to unique vectors.

Positional Encoding (PE): Relative Position

- **Relative Position:** This refers to the **distance between tokens**, not their exact location.
- For example:
  - "I" is **1 step before** "like"
  - "like" is **1 step before** "pizza"
  - "pizza" is **2 steps after** "I"
- **Relative positional encoding** helps the model learn relationships like:
  - “This word is right after another.”
  - “This phrase is always 3 tokens apart.”
- Some advanced models (like Transformer-XL or T5) use **relative position embeddings** to better capture such patterns especially useful for long sequences.

Positional Encoding (PE): Generalization

- **Generalization:** This means the model can:
  - Handle **longer sequences** than it was trained on.
  - Understand **new patterns** it hasn't seen before.
- Sinusoidal positional encodings help with generalization because:
  - They are **continuous** and **mathematically defined**.
  - They don't depend on a fixed vocabulary or learned embeddings.
  - So even if the model sees position 1024 for the first time, it can still compute a meaningful encoding.
- **Training on diverse data**:
  - The more varied the training data, the better the model can generalize.
  - It sees many ways people express ideas, so it learns flexible patterns.
- **Why is generalization important?**
  - Without generalization, the model would only work on examples it has seen.
- With generalization, it can:
  - Translate new sentences
  - Answer new questions
  - Generate new text
  - Understand new contexts

PE: Not Labels Like "First", "Second", Etc.

- The values in **positional encoding** are **not labels** like "first", "second", etc.
- Instead, they are **unique patterns** generated by sine and cosine functions that vary with position.
- **[0, 1, 0, 1]** is the pattern for **position 0**
- The next token (position 1) might get something like **[0.0001, 0.9999, 0.0002, 0.9998]**
- And so on...
- These patterns are **distinct for each position**, and the model **learns to associate these patterns with position** during training.

PE: How Does Model Know the Position?

- The vector **[0, 1, 0, 1]** (for position 0) is **not a label**
- it's a **numerical pattern** generated by sine and cosine functions.
- These patterns are:
- **Unique for each position**
- **Consistent across sequences**
- **Smoothly varying**, so the model can learn **relative positions** too
- The model **learns during training** that this pattern corresponds to the **first position**, because it sees it repeatedly associated with the first token in many sequences.

PE: Analogy

- Think of positional encoding like a **musical note**:
- The note itself doesn't say "this is the first beat."
- But if you always hear that note at the start of a song, you learn to associate it with the beginning.
- **Why sin/cos?**
- They create **continuous, non-repeating patterns** across dimensions.
- They allow the model to **generalize to longer sequences**.
- They help the model learn **relative distances** between tokens.

PE: Use of these Encodings

- They create **smooth, continuous signals** that help the model learn patterns like "next word", "previous word", "far away", etc.
- Know where each token is in the sentence.
- Understand relationships between tokens (e.g., subject–verb, modifier–noun).
- Maintain sequence structure even though it processes all tokens in parallel.

PE: Learned vs Fixed Positional Encoding

- Some models (like BERT) use **learned positional embeddings** where the position vectors are trained like word embeddings.
- Others (like the original Transformer) use **fixed sinusoidal encodings** which are mathematically generated and not learned.
- Both serve the same purpose: **inject position information** into the model.

PE: Learned vs Fixed Positional Encoding

- Some models (like BERT) use **learned positional embeddings** where the position vectors are trained like word embeddings.
- Others (like the original Transformer) use **fixed sinusoidal encodings** which are mathematically generated and not learned.
- Both serve the same purpose: **inject position information** into the model.

Residual Connections and Layer Normalization

An Example: Normalization

Salary	Bonus_percentage
1000000	15
1500000	12
2000000	10

- If we apply $(a + b)^2 =$ (Salary+ Bonus%), It skewed Calculations.
- **The Problem of Scale:**
- In many datasets, features are measured in different units and have vastly different ranges.
- (e.g., a person's salary in hundreds of thousands vs. a bonus percentage between 0-10%).
- **The "Dominance" Issue:** Without normalization, machine learning algorithms (especially those based on gradient descent) can become biased.
- The feature with the largest scale or variance will disproportionately influence the model's predictions, causing the contributions of smaller-scale features to be ignored.
- **The Solution:** Normalization rescales numeric features to a common, standard scale.
- This ensures that every feature contributes fairly to the model's learning process, leading to more accurate and reliable results.
- It makes the optimization landscape smoother, allowing the model to find the best solution more easily.

Why Normalize Data? A Deeper Look

- **Promotes Training Stability:** Unnormalized data can lead to extremely large or small weight updates during training, a problem known as "exploding" or "vanishing" gradients.
- Normalization constrains the data to a predictable range, which in turn keeps the gradient updates stable and prevents the training process from diverging.
- **Enables Faster Convergence:** Normalization helps the optimization algorithm (like Gradient Descent) find the optimal solution more directly.
- **Reduces Overfitting:** By scaling and centering data, normalization can act as a subtle form of regularization.
- It ensures the model learns the underlying patterns in the data rather than being overly influenced by the specific scale of the features in the training set.
- **Compatibility with Activation Functions:** Certain activation functions, like tanh and sigmoid, perform poorly with very large input values.
- They become "saturated" at their extremes (-1/1 or 0/1), causing their gradients to become near-zero.
- Normalizing the inputs keeps them in the "active" region of these functions, allowing gradients to flow properly.

Why Normalize Data? A Deeper Look

- **Promotes Training Stability:** Unnormalized data can lead to extremely large or small weight updates during training, a problem known as "exploding" or "vanishing" gradients.
- Normalization constrains the data to a predictable range, which in turn keeps the gradient updates stable and prevents the training process from diverging.
- **Enables Faster Convergence:** Normalization helps the optimization algorithm (like Gradient Descent) find the optimal solution more directly.
- **Reduces Overfitting:** By scaling and centering data, normalization can act as a subtle form of regularization.
- It ensures the model learns the underlying patterns in the data rather than being overly influenced by the specific scale of the features in the training set.
- **Compatibility with Activation Functions:** Certain activation functions, like tanh and sigmoid, perform poorly with very large input values.
- They become "saturated" at their extremes (-1/1 or 0/1), causing their gradients to become near-zero.
- Normalizing the inputs keeps them in the "active" region of these functions, allowing gradients to flow properly.

Common Normalization Techniques

- **Min-Max Normalization:**
- **Formula:** $x\_norm = (x - \min(x)) / (\max(x) - \min(x))$
- **Function:** This technique linearly rescales all data points to a fixed range, typically between 0 and 1.
- **Pros & Cons:** It's simple and guarantees all features will have the exact same scale.
- However, it's very sensitive to outliers; a single extreme value can compress the rest of the data into a very small range.

Salary	Bonus_percentage	Salary_Normalized	Bonus_Percentage_Normalized
1000000	15	0.0	1.0
1500000	12	0.5	0.4
2000000	10	1.0	0.0

Common Normalization Techniques

- **Z-score (Standard) Normalization:**
- **Formula:** $z = (x - \mu) / \sigma$
- Where mu (μ) is the mean and Sigma (σ) is the standard deviation
- **Function:** This method recenters the data to have a mean of 0 and a standard deviation of 1. It doesn't bind values to a specific range.
- **Pros & Cons:** It handles outliers better than Min-Max normalization and preserves the shape of the original distribution. It's the most common normalization technique for general-purpose machine learning.
- Original data: [10, 12, 14, 16]
- Mean = 13, Std ≈ 2.236
- For 16: So, 16 is z=**1.34 standard deviations above the mean**.
- **Think of it like this:**
- If you're **1 standard deviation above the mean**, you're a little above average.
- If you're **2 standard deviations above**, you're much higher than most values.
- If you're **0**, you're exactly at the average.

Layer normalization

- **For Example:**
  - S1: Hi Hello
  - S2: My Name is Zohair
- So, Maximum Sequence Length is = 4
  - Less than Max Token use placeholder <pad>
- Verticals are Features
- Horizontal are Tokens
- Vertical/Feature wise norm = Batch normalization
- In transformers we use layer normalization
- Horizontal/Token wise norm = Layer normalization

	P1	P2	P3	P4
Hi	0.1	0.2	0.3	0.4
Hello	0.11	0.22	0.33	0.44
<pad>	0	0	0	0
<pad>	0	0	0	0
My	0.13	0.78	-0.7	-0.15
Name	0.14	0.8	-0.75	-0.15
Is	0.11	0.28	-0.17	-0.15
Aman	0.13	0.21	-0.12	-0.11

Normalization: Batches and Layers

- Data in Deep Learning: In deep learning, data is processed in large, multi-dimensional tensors.
- For NLP, a typical tensor might have the shape (batch_size, sequence_length, embedding_dim).
- **Two Directions of Normalization:**
- **Batch Normalization (Vertical):** Normalizes values *feature-wise*.
- For a given feature (e.g., the 10th dimension of a word embedding), it calculates the mean and standard deviation across all the tokens in a *batch*.
- It asks, "What's the distribution of this specific feature across different sentences?"
- **Layer Normalization (Horizontal):** Normalizes values *token-wise*.
- For a given token (a single word's vector representation), it calculates the mean and standard deviation across all of its *features*.
- It asks, "What's the distribution of features within this single token?"

Why Transformers Use Layer Normalization

- **The Padding Problem:** Transformer models process sentences of varying lengths.
- To create uniform batches, shorter sentences are "padded" with zero-value tokens.
- In Batch Normalization, these zeros would be included in the feature-wise calculations, distorting the true mean and variance and destabilizing the training.
- Layer Normalization avoids this by normalizing each token independently.
- **Independence from Batch Size:** The statistics for Batch Normalization depend heavily on having a sufficiently large and representative batch.
- In NLP, it's common to use very small batch sizes (even a batch size of 1).
- In such cases, batch statistics are noisy and unreliable.
- Layer Normalization's statistics are calculated per-token, making it completely independent of the batch size and more stable.
- **Suitability for Sequential Data:** Layer Normalization was designed to work well with sequential data (like in RNNs and Transformers) where each element (a token) is processed with its own context.
- It treats each token's feature vector as a layer to be normalized, which aligns perfectly with the Transformer's architecture.

Transformers: Key Architectural Components

- **Embeddings:** The initial layer that converts discrete input tokens (words or sub-words) into dense, continuous numerical vectors. This is the first step in turning language into math.
- **Positional Encoding:** Since the Transformer architecture processes all tokens at once (unlike RNNs), it has no inherent sense of sequence order.
- Positional encodings are vectors that are added to the embeddings to give the model information about the position of each token in the sequence.
- **Multi-head Attention:** The core engine of the Transformer. It allows the model to weigh the importance of all other tokens in the sequence when processing a single token, learning the complex relationships and dependencies between words.
- **Add & Norm (Residual Connection + Layer Normalization):**
- The "glue" that holds the architecture together.
- After each major sub-layer (like attention or the feed-forward network), this component adds the input of the sub-layer to its output and then applies layer normalization.

Transformers: Key Architectural Components

3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Mathematical Formulas Behind Normalization

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu_B^{(k)}}{\sqrt{(\sigma_B^{(k)})^2 + \epsilon}}$$

Batch Norm(Vertical/feature wise)

where $\mu_B^{(k)}$ and $\sigma_B^{(k)}$ are the mean and variance **over the batch** for feature k.

Formula:

$$\hat{x} = \frac{x - \mu_{\text{features}}}{\sqrt{\sigma^2_{\text{features}} + \epsilon}}$$

Layer Norm/horizontal/Token wise

Residual Connection

- The **embedding** goes through the first sublayer (attention).
- Then you add the **original embedding (input)** to the **attention output** (residual connection).
- Then apply **LayerNorm**.

Formula for the first sublayer:

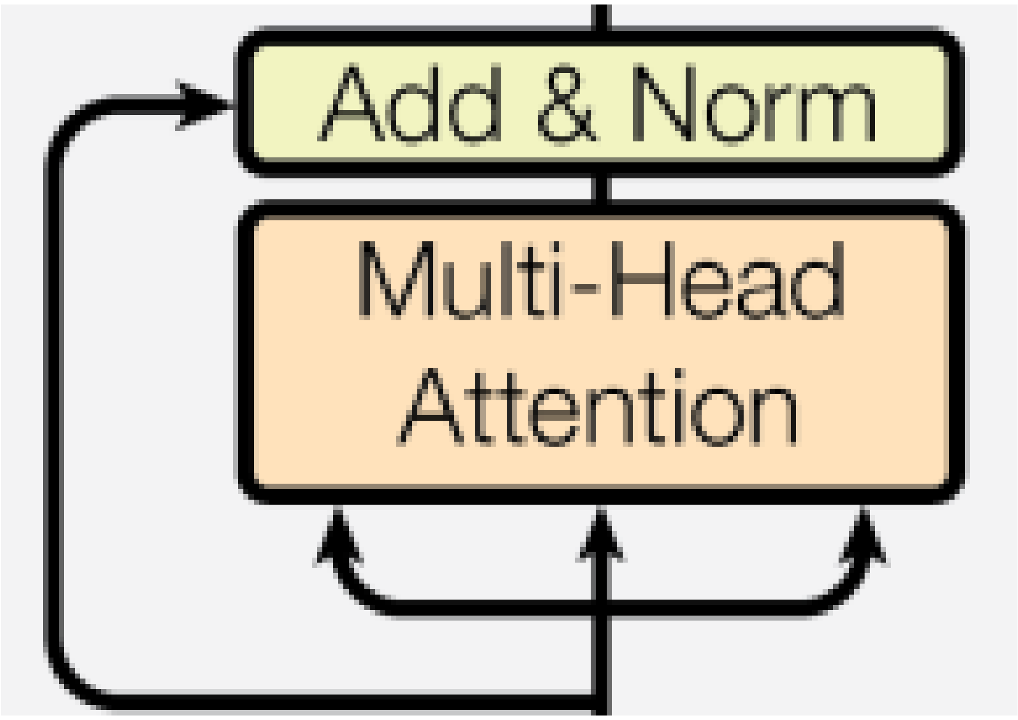
$$\text{AttentionOutput} = MultiHeadAttention(X)$$

$$\text{Add\&Norm} = LayerNorm(X + AttentionOutput)$$

The same pattern applies to the **Feed Forward Network** sublayer.

- **Why Keep residual connection?**
- 1. Store/keep original info
- 2. Make Learning Faster

```
x = embedding(input_ids)
z = sublayer(x)  # e.g., self-attention or feed-forward
output = LayerNorm(z + x)  # residual connection + LayerNorm
```



Residual Connection

- In a **Transformer**, a **Residual Connection** means adding the original input of a layer back to its output before applying normalization.
- **Why is it used?**
- To **help gradients flow** during backpropagation (avoids vanishing gradients).
- To **preserve original information** while adding new transformations.
- To **stabilize training** of deep networks.
- In Transformers:
- **Base** $(\boldsymbol{x})$ = input embeddings or previous layer output.
- **Toppings** $F(x)$ = result of attention or feed-forward network.
- **Final pizza** = $x + F(x)$, then normalized.