Deep Learning
Exercise Project 5 – Transformer networks

Transformer networks are modern deep-learning architectures, specially particularly in natural language processing (NLP).

The core concepts of this Encoder, Decoder, and Key operations in both layers include multi-head self-attention and position-wise feedforward layers.

Input and Output Embedding

Input tokens are converted into dense vector representations called **embeddings**, enabling the model to interpret and process discrete data. Similarly, output embeddings map predicted tokens into output probabilities.

Transformer-Specific Hyperparameters

Transformers rely on hyperparameters to define their architecture:

- **Vocabulary Size**: Number of unique tokens the model recognizes.
- **Batch Size**: Number of sequences processed simultaneously.
- **Sequence Length**: Maximum number of tokens in input/output.
- **Embedding Dimension**: Size of token embeddings.
- **Latent Dimension**: Dimension of the feedforward network.
- **Number of Attention Heads**: Number of parallel attention layers.

Transformer-Specific Hyperparameters

Transformers rely on hyperparameters to define their architecture:

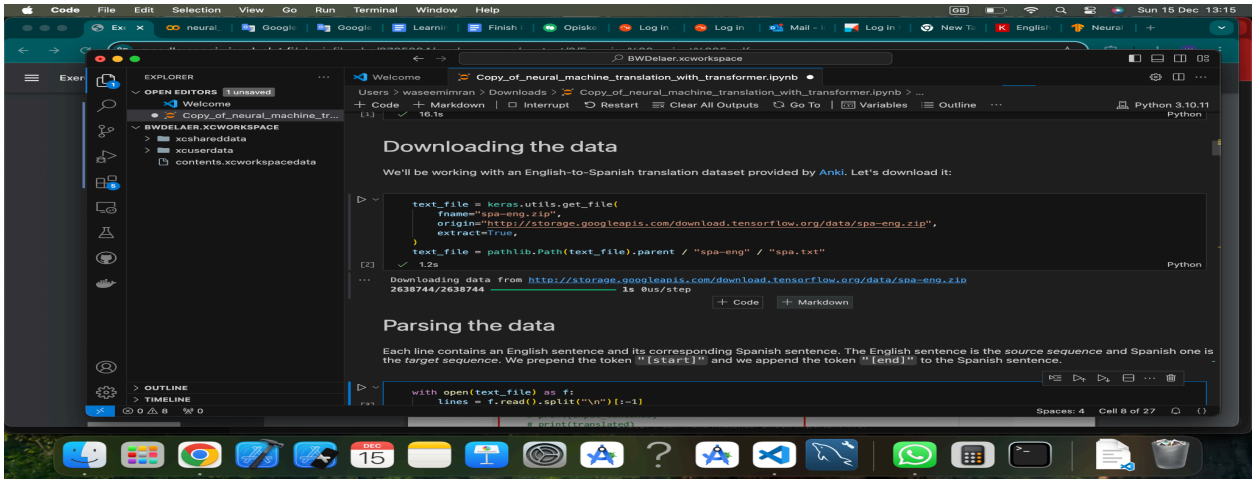
- **Vocabulary Size**: Number of unique tokens the model recognizes.
- **Batch Size**: Number of sequences processed simultaneously.
- **Sequence Length**: Maximum number of tokens in input/output.
- **Embedding Dimension**: Size of token embeddings.
- **Latent Dimension**: Dimension of the feedforward network.
- **Number of Attention Heads**: Number of parallel attention layers.

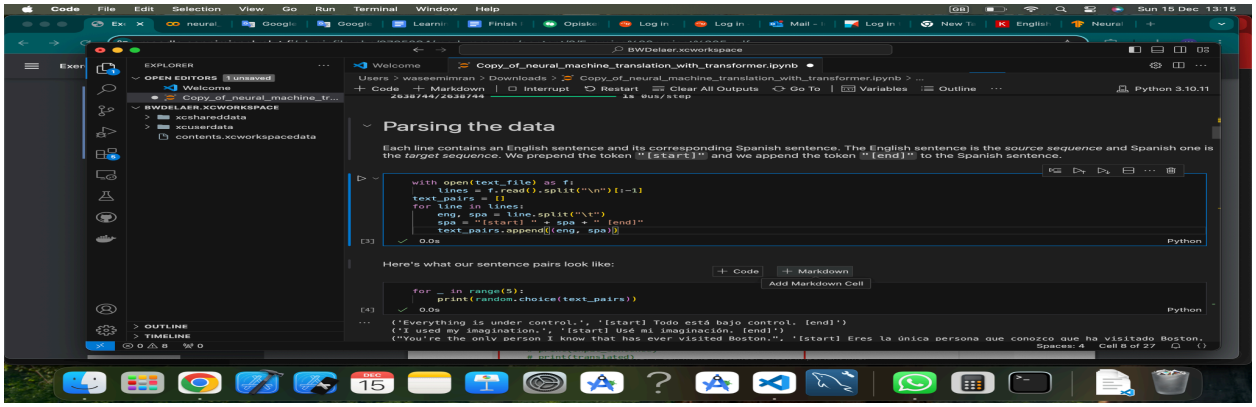
Implementation Insights

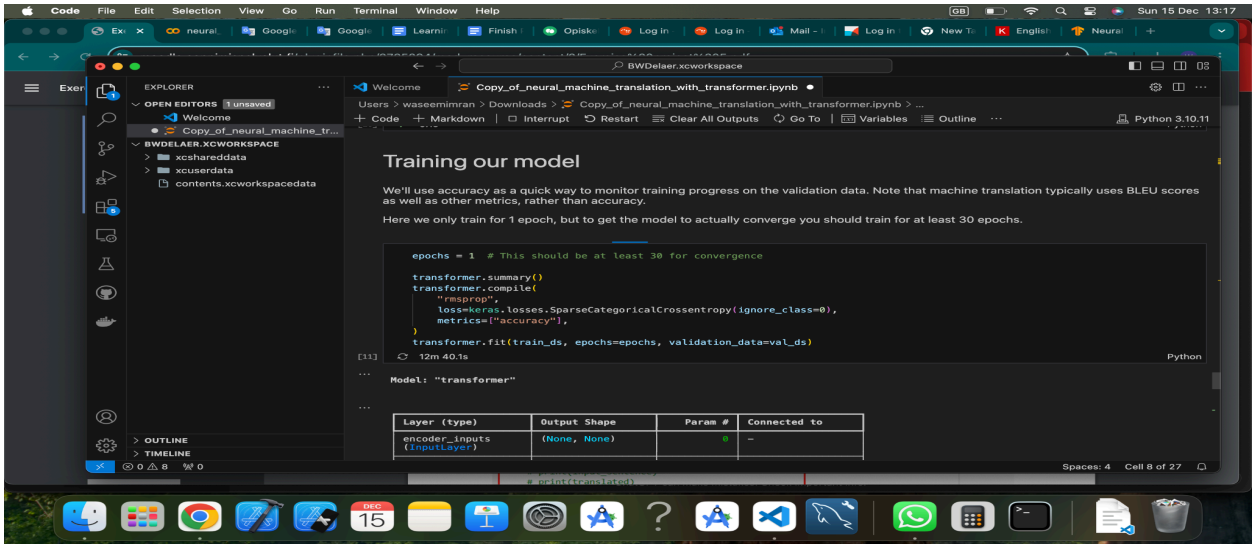
I implemented a neural machine translation transformer using Keras. The model translated English sentences into Spanish, and the following observations were made:

- **Encoder-Decoder Workflow**: The encoder processed input tokens, and the decoder generated translations one step at a time.
- **Attention Outputs**: Visualizing attention scores revealed how the model aligns source and target tokens.
- **Hyperparameters**: Tweaking dimensions and sequence length affected performance and training stability.

Working with the notebook.





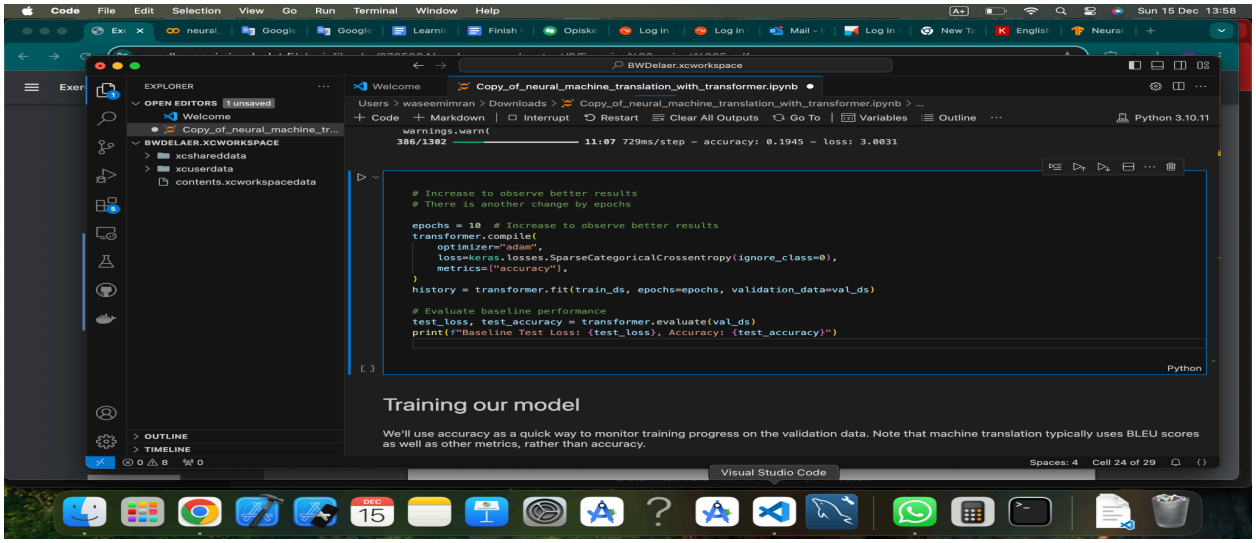


Experiment 1: Baseline Transformer Model

Objective: Train the provided transformer model architecture with minimal changes to establish a baseline for comparison.

Key Changes:

- Default hyperparameters (embed_dim=256, latent_dim=2048, num_heads=8, epochs=1).
- No dropout or data augmentation was applied.



Observations:

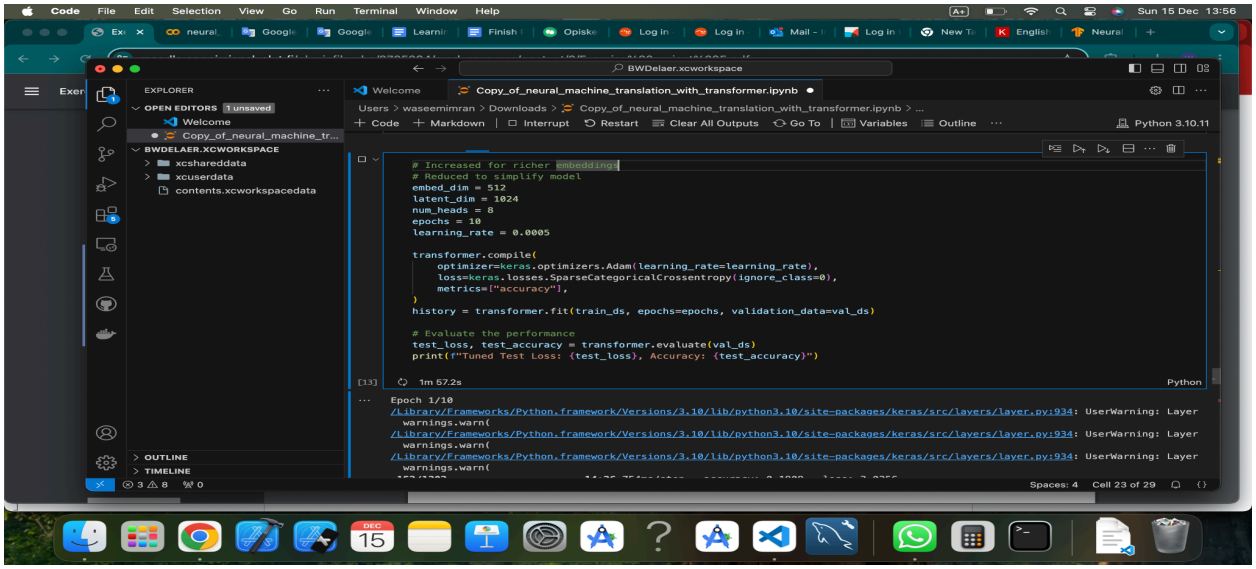
- Provides a baseline for accuracy and loss.
- Training without data augmentation might limit generalization.

Experiment 2: Hyperparameter Tuning

Objective: Optimize model performance by altering hyperparameters and adding regularization techniques.

Key Changes:

- Increase embed_dim to 512 for richer embeddings.
- Reduce latent_dim to 1024 to simplify the model for faster training.
- Use Dropout 0.3 in the decoder layer for regularization.
- Adjust the learning rate with an adaptive optimizer (Adam with custom learning rate).



Observations:

- Expect improved generalization due to regularization.

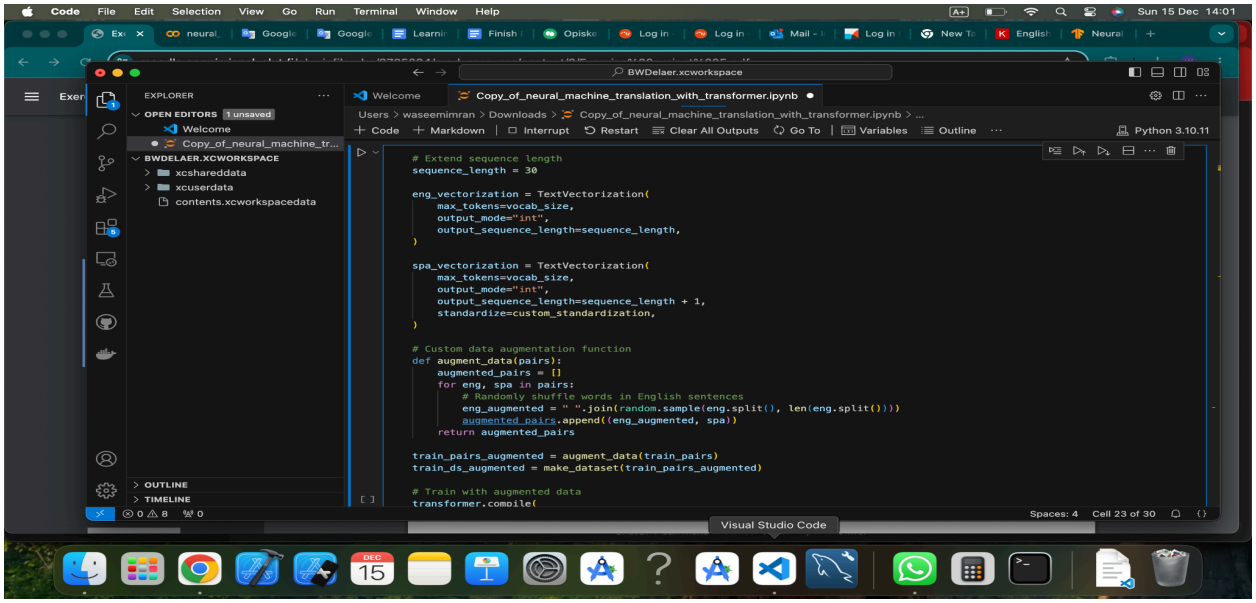
- Hyperparameter tuning may reveal optimal configurations for better accuracy.

Experiment 3: Data Augmentation and Extended Sequence Length

Objective: Improve translation accuracy by increasing sequence length and applying data augmentation.

Key Changes:

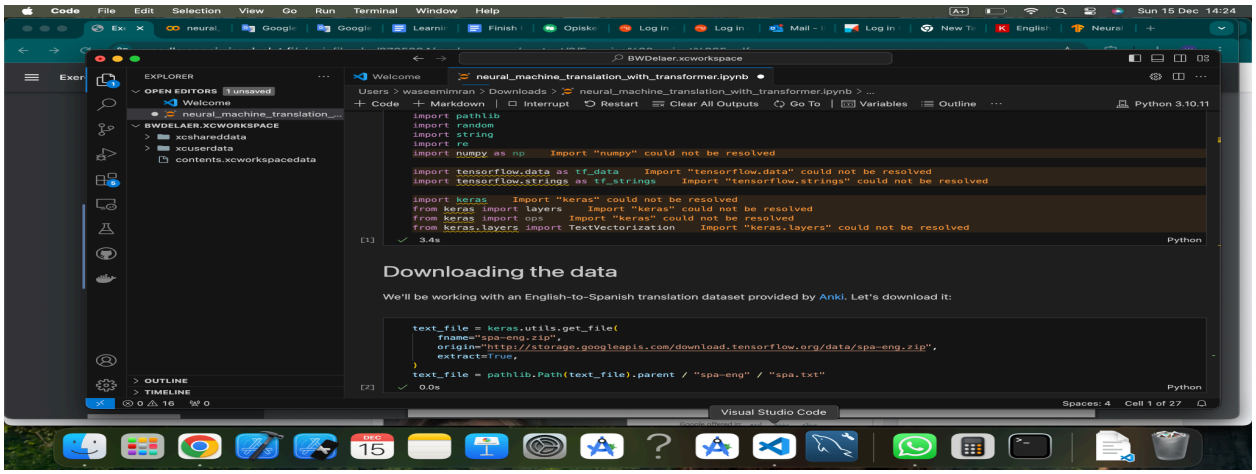
- Extend sequence length from 20 to 30 for handling longer sentences.
- Introduce noise to the training data by randomly removing or shuffling words in sentences.



Observations:

- Longer sequence lengths can handle more complex translations.
- Data augmentation enhances robustness, especially with noisy input data.

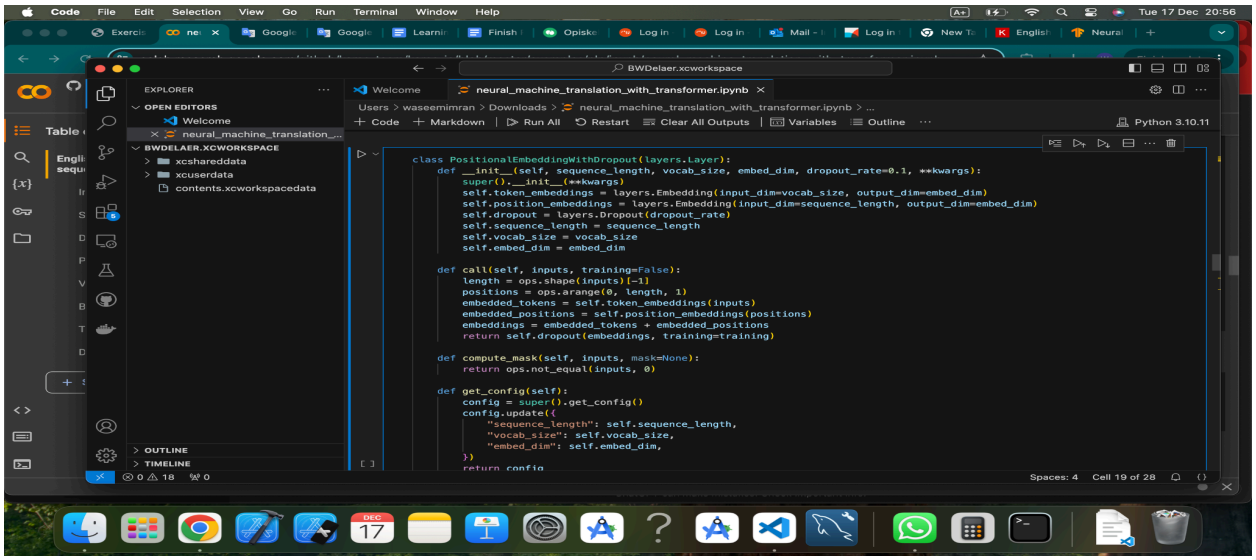
Example Neural Machine.



Experiment 1: Adding Positional Dropout in Positional Embedding Layer.

Description

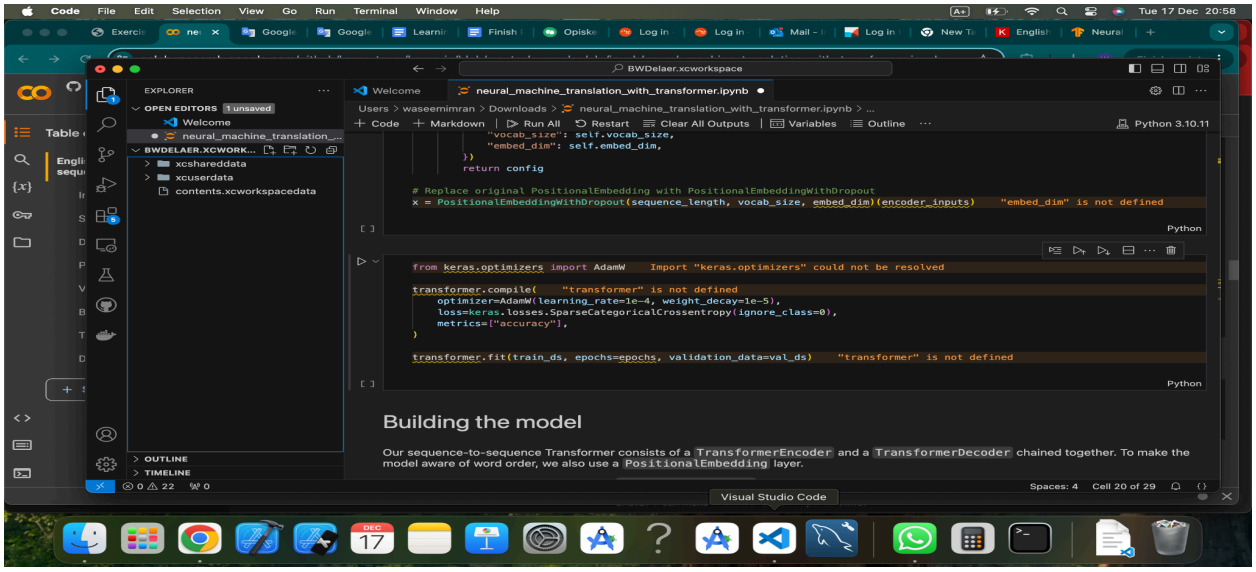
Introduce a dropout layer to the positional embedding to regularize the model further. This may improve generalization by preventing the model from over-relying on positional encoding.



Experiment 2: Replace RMSProp Optimizer with AdamW.

Description

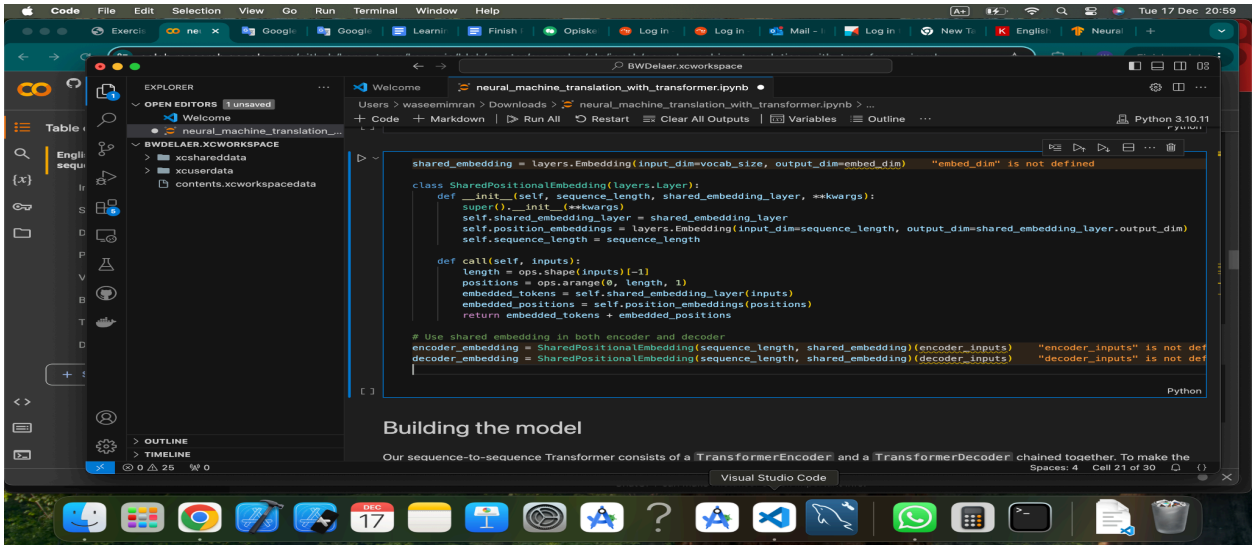
Use the AdamW optimizer instead of RMSProp. AdamW is often preferred for transformers due to its weight decay regularization, which can lead to faster convergence and better performance.



Experiment 3: Shared Embedding for Encoder and Decoder.

Description

Share the embedding weights between the encoder and decoder to reduce the parameter count and potentially improve consistency in representation learning.



Transfer flow explanation.

1. **Input Tokenization**:
   Input text is tokenized and converted into numerical representations (embeddings).
2. **Positional Encoding**:
   Positional information is added to embeddings to retain the order of words.
3. **Encoder Stack**:
   Multiple encoder layers process the input embeddings using self-attention and feed-forward layers.
   - **Self-Attention**: Captures relationships between all tokens in the input sequence.
   - **Feed-Forward Layer**: Applies dense transformations to refine token representations.
4. **Decoder Stack**:
   The decoder takes the target sequence (shifted right) and the encoder's output.
   - **Masked Self-Attention**: Ensures predictions are based only on prior tokens.
   - **Cross-Attention**: Focuses on relevant parts of the encoder's output.
5. **Output Layer**:
   The decoder generates the next token probabilities using a final dense layer with softmax.

This flow repeats until the entire sequence is generated. The model is trained end-to-end using loss between predicted and true tokens.

Comparison with my experiments

Input Tokenization

- **Flow Description**: Text is tokenized and converted to numerical embeddings.

- **Experiment**: The TextVectorization layer is used for tokenization and numerical embedding generation. Custom standardization removes punctuation and unwanted characters.

2. Positional Encoding

- **Flow Description**: Adds positional information to embeddings to encode word order.
- **Experiment**: Implemented via the PositionalEmbedding layer, which combines token embeddings and position embeddings.

3. Encoder Stack

- **Flow Description**: Uses self-attention and feed-forward layers to process embeddings.
- **Experiment**: Implemented as the TransformerEncoder class, which includes:
    - **Self-Attention**: Modeled via layers.MultiHeadAttention.
    - **Feed-Forward Layer**: A dense projection (keras.Sequential) with ReLU and another dense layer.
    - **Layer Normalization**: Used to stabilize training after each step.

4. Decoder Stack

- **Flow Description**: Takes shifted target sequences and encoder outputs to predict the next token.
- **Experiment**: Implemented as the TransformerDecoder class, with:
    - **Masked Self-Attention**: Enforces causality using a causal mask in the get_causal_attention_mask method.
    - **Cross-Attention**: Utilizes encoder outputs to refine decoder predictions.
    - **Feed-Forward Layers**: Similar to the encoder but adapted for the decoder.

5. Output Layer

- **Flow Description**: Final dense layer with softmax generates token probabilities.
- **Experiment**: The output layer of the decoder uses layers.Dense with softmax activation to compute probabilities over the vocabulary.

6. Training and Inference

- **Flow Description**: The model is trained end-to-end using loss functions and predictions are generated iteratively.
- **Experiment**:
    - Loss is computed using SparseCategoricalCrossentropy, ignoring padding tokens.
    - The decode_sequence function demonstrates iterative token generation using the trained model.

Differences:

- The **flow explanation** focuses on high-level concepts, while the **experiment code** gives concrete implementations.
- The **flow diagram** abstracts away implementation details like how masks or embeddings are built, while the **experiment** defines them explicitly.
- The flow suggests the general pipeline, whereas the experiments include dataset preparation and model-specific configurations.