# Creating a Transformer Assignment for Machine Translation

Logan Falzarano
lfalzar1@jhu.edu

**Abstract**

The current Neural Machine Translation course curriculum includes hands-on experience with recurrent neural networks but lacks practical components covering transformer architectures, which have become the de facto standard in modern machine translation systems [4][6]. This project aims to fill this gap by creating a comprehensive, hands-on assignment that guides students through implementing the transformer architecture as described in ”Attention is All You Need”[1]. The assignment bridges the gap between theoretical understanding and practical implementation by decomposing the transformer architecture into manageable components, each with clear learning objectives and implementation tasks. **All code including the reference implementation as well as a version to provide to students have been completed and are available at:** `https://github.com/loganfalzarano/MT_final_project`

## 1 Core Transformer Implementation

The assignment provides students with skeleton code for implementing a transformer architecture. The main Transformer class, shown below, provides an example of what will be provided to students.

```
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size, embedding_size,
                 n_heads, hidden_size, n_layers, dropout_p=0.1):
        super(Transformer, self).__init__()
        #TODO: initialize an input embedding layer for the source language
        #TODO: initialize a positional encoding layer for the source language
        #TODO: initialize an encoder

        #TODO: initialize an input embedding layer for the target language
        #TODO: initialize a positional encoding layer for the target language
        #NOTE (this could technically be shared with the encoder but we create
        #a separate instance to align with the architecture diagram in the paper)
        #TODO: initialize a decoder

        #TODO: initialize a linear layer to project the decoder outputs to the
        #target vocabulary size

    def encode(self, src_input, src_mask):
        #TODO: implement encoding
        return encoder_outputs, all_encoder_attention_scores

    def decoder(self, tgt_input, encoder_outputs, src_mask, tgt_mask):
        #TODO implement decoding
        return decoder_outputs, all_masked_attention_scores,
    all_cross_attention_scores

    def get_predictions(self, decoder_outputs):
        return F.log_softmax(self.projection_linear_layer(decoder_outputs), dim=-1)
```

The following subsections will detail each of the components that students will have to implement. All of the reference implementations can be seen in the **attention_is_all_you_need.py** file. The skeleton implementations will be provided in **attention_is_all_you_need-student.py**

## 1.1 Input Embedding

Students will implement the InputEmbedding class, which converts input tokens into dense vector representations. They will learn about embedding layers and the importance of scaling embeddings by $\sqrt{d_{model}}$ as described in [1]. Here student's will be able to use PyTorch's **nn.Embedding** class.

## 1.2 Positional Encoding

The PositionalEncoding class implementation teaches students how transformers maintain sequence order information without recurrence using sinusoidal position encodings. Since this is a bit more complicated and requires 1) modifying computation for numerical stability and 2) registering a buffer to ensure positional encodings are added to the model state, the full implementation of this class will be provided to students.

The PositionalEncoding class implements the equations:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

## 1.3 Multi-Head Attention

In the MultiHeadedAttention class, students implement the core attention mechanism, including:

- Query, key, and value transformations
- Scaled dot-product attention
- Parallel attention heads
- Output projection

Students will have to implement the **self_attention** function as well as splitting data among heads. One of the most complicated parts about implementing MultiHeadedAttention from scratch is concatenating the heads together after attention is computed. Thus, students will be provided:

```
#concatenate the heads
x = x.permute(0, 2, 1, 3).contiguous().view(x.size(0), -1, self.embedding_size)
```

This will help guide students towards a correct implementation of MultiHeadAttention [8].

## 1.4 Layer Normalization

The LayerNorm class implementation teaches students about stabilizing deep networks through normalization. Students implement the layer normalization equation:

$$LayerNorm(x) = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

Rather than using PyTorch's implementation of LayerNorm student's creating their own parameters will allow them to learn about how LayerNorm works and why it is applied in Transformers.

## 1.5 Feed-Forward Networks

Students will implement a simple feed forward neural network with ReLu activation.

## 1.6 Encoder Layer

The EncoderLayer combines previously implemented components to create a single layer of the transformer encoder:

```
class EncoderLayer(nn.Module):
    def __init__(self, embedding_size, n_heads, hidden_size, dropout_p=0.1):
        super(EncoderLayer, self).__init__()

        #TODO create a MultiHeadedAttention block and layer norm
        #TODO create a FeedForward block and layer norm
        #TODO create dropout layers

    def forward(self, x, mask):
        #TODO implement a forward pass through an Encoder Layer as described in the paper

        return x, attention_scores
```

Students will:

- Combine their MultiHeadedAttention, LayerNorm, and FeedForward implementations
- Add residual connections around each sublayer
- Implement proper information flow: Input → Self-Attention → Normalization → Feed-Forward

## 1.7 Decoder Layer

The DecoderLayer extends the encoder pattern by adding cross-attention to the encoder's output:

```
class DecoderLayer(nn.Module):
    def __init__(self, embedding_size, n_heads, hidden_size, dropout_p=0.1):
        super(DecoderLayer, self).__init__()

        #TODO create MultiHeadedAttention (masked), dropout, and layer norm
        #TODO create a MultiHeadedAttention block (cross-attention), dropout and layer norm
        #TODO create a FeedForward block, dropout, and layer norm

    def forward(self, x, encoder_outputs, src_mask, tgt_mask):

        #TODO pass through masked attention (x as all three inputs)
        #TODO pass through cross attention (x as query, encoder_outputs as key and value)
        #TODO pass through feedforward network

        return x, masked_attention_scores, cross_attention_scores
```

Students will:

- Reuse their MultiHeadedAttention implementation twice:
  - Once for masked self-attention on decoder inputs

    - Once for cross-attention to encoder outputs

- Apply their LayerNorm implementation after each sublayer
- Chain the components: Input → Masked Self-Attention → Cross-Attention → Feed-Forward

The assignment emphasizes modular design - students see how their individual component implementations combine to create increasingly complex layers in the transformer architecture.

### 1.8 Encoder and Decoder

The Encoder and Decoder classes combine the above components with residual connections. Students learn about:

- Stacking multiple layers using **nn.ModuleList**

## 2 Data

The assignment has been configured to allow for use of any dataset in the format:

```
<french sentence>|||<english sentence>
```

thus, the data in the current HW4 can be used for this assignment as well.

- Training set: 8,701 sentence pairs
- Development set: 971 sentence pairs
- Test set: 971 sentence pairs
- French vocabulary size: 1,382 tokens
- English vocabulary size: 1,362 tokens

## 3 DataLoaders

Since Machine Translation is not a course about machine learning, we have provided the DataLoaders to students so that they can focus on understanding and implementing the transformer architecture. The data loading pipeline is implemented using PyTorch's Dataset and DataLoader classes. Key features include:

- Custom TranslationDataset class that converts sentence pairs to tensors
- Batch collation with dynamic padding
- Generation of attention masks for padded sequences
- Creation of shifted target sequences for teacher forcing

Each batch contains:

- encoder_inputs: Padded source sequences

- decoder_inputs: Padded target sequences (shifted right)
- labels: Target sequences for loss computation
- encoder_mask: Mask for padded positions in source
- decoder_mask: Combined causal and padding mask for target

# 4 Training

Students will implement a training loop that includes:

- Batch processing with gradient accumulation
- Label smoothing cross-entropy loss

To make the training loop a bit easier to follow, we have provided an annotated loop using the **tqdm** package, allowing students to see how training is progressing easily.

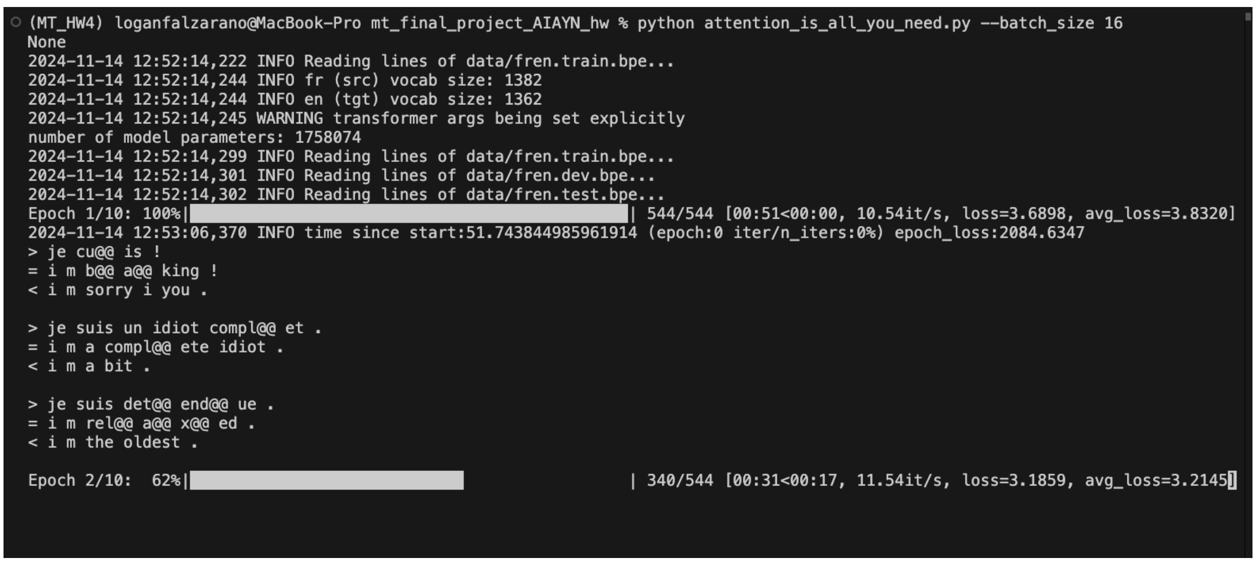


Figure 1: Training progress visualization using tqdm

# 5 Results

After some experimentation, we settled on the following default hyper-parameters for the model. These will be the default parameters provided to students. Note that all BLEU scores are presented in decimal format.

- Embedding size: 128
- Number of heads: 4
- Hidden size: 256
- Number of layers: 2

- Dropout: 0.2
- Batch size: 32
- Learning rate: 0.001

After 10 epochs of training, the model achieved:

- Development BLEU: 0.45
- Test BLEU: 0.45

In Tables 1 and 2 we have included the results of hyper-parameter optimization experiments.

Table 1: Machine Translation Hyperparameter Optimization Results - Part 1

| Parameter | Epoch Variations | | | | Embedding Size | | | Hidden Size | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Batch Size | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 |
| Embedding Size | 128 | 128 | 128 | 128 | 256 | 512 | 1024 | 128 | 128 | 128 | 128 |
| Num Heads | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Hidden Size | 256 | 256 | 256 | 256 | 256 | 256 | 256 | 128 | 512 | 1024 | 256 |
| Num Layers | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Learning Rate | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 |
| Epochs | 10 | 20 | 30 | 40 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Dev Score | 0.43 | 0.52 | 0.54 | 0.48 | 0.53 | 0.50 | 0.06 | 0.44 | 0.48 | 0.46 | 0.44 |
| Test Score | 0.44 | 0.51 | 0.55 | 0.47 | 0.53 | 0.54 | 0.07 | 0.44 | 0.48 | 0.47 | 0.44 |

Table 2: Machine Translation Hyperparameter Optimization Results - Part 2

| Parameter | Num Layers | | Num Heads | | | Learning Rate | | | Best Results |
|---|---|---|---|---|---|---|---|---|---|
| Batch Size | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 |
| Embedding Size | 128 | 128 | 128 | 128 | 128 | 128 | 128 | 128 | 256 |
| Num Heads | 4 | 4 | 2 | 1 | 8 | 4 | 4 | 4 | 4 |
| Hidden Size | 256 | 256 | 256 | 256 | 256 | 256 | 256 | 256 | 512 |
| Num Layers | 4 | 8 | 2 | 2 | 2 | 2 | 2 | 2 | 4 |
| Learning Rate | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.01 | 0.0001 | 0.001 |
| Epochs | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 50 |
| Dev Score | 0.45 | 0.45 | 0.44 | 0.46 | 0.47 | 0.43 | 0.00 | 0.13 | 0.59 |
| Test Score | 0.47 | 0.45 | 0.44 | 0.47 | 0.46 | 0.44 | 0.00 | 0.13 | 0.58 |

Based on these results, we propose:

- Beginner threshold: BLEU score of 0.30
- Advanced threshold: BLEU score of 0.45

## 6 Evaluation Script

The course staff will be provided with an evaluation script that evaluates student's reference translations called **grade.py**:

The script takes two arguments

- - -translations, which is a path to the translations to be evaluated.
- - -references, which is a path to the references to be used for evaluation.

## 7 Extension - Diagonal Encoder Attention

When training the transformer we noticed that BLEU scores were high and the network was learning to translate but the attention weights didn't match with human intuition. For example, in one training run that achieved a BLEU score of 0.45, the encoder attention weights did not seem to follow a diagonal attention pattern as shown in Figure 2. To address this, we implemented a custom encoder mask which forced the encoder to look at diagonal entries similar to [2]. After implementing this attention mask, we observed the attention weights shown in Figure 3 and an improved BLEU score of 0.46 (with identical model hyperparameters). Over several training runs, we observed higher performance in the diagonally masked network as shown in table 3. This suggests that a diagonally masked encoder can be a powerful tool, especially with very small parallel datasets.

Table 3: Machine Translation Model Size and Masking Results (after 10 epochs)

| **Parameter** | **Small Model** | | **Medium Model** | | **Large Model** | |
|---|---|---|---|---|---|---|
| Embedding Size | 64 | 64 | 128 | 128 | 256 | 256 |
| Hidden Size | 128 | 128 | 256 | 256 | 512 | 512 |
| Num Heads | 2 | 2 | 2 | 2 | 2 | 2 |
| Num Layers | 2 | 2 | 2 | 2 | 2 | 2 |
| Encoder Mask | None | Diagonal | None | Diagonal | None | Diagonal |
| Epochs | 10 | 10 | 20 | 20 | 20 | 20 |
| Batch Size | 32 | 32 | 32 | 32 | 32 | 32 |
| Train BLEU | 0.36 | 0.42 | 0.70 | 0.84 | 0.76 | 0.83 |
| Dev BLEU | 0.29 | 0.33 | 0.45 | 0.56 | 0.51 | 0.54 |
| Test BLEU | 0.30 | 0.34 | 0.49 | 0.56 | 0.53 | 0.53 |

## 8 Provided Files Outline

1. **attention_is_all_you_need.py** - base implementation of the "Attention is All You Need" Paper
2. **attention_is_all_you_need-student.py** - base implementation of the "Attention is All You Need" Paper with code replaced by TODOs for students to implement
3. **attention_is_all_you_need-ext.py** - extended implementation of the "Attention is All You Need" Paper with diagonal attention mask
4. **data** - data directory
   (a) **fren.train.bpe** - byte-pair encoded data to be used for training
   (b) **fren.dev.bpe** - byte-pair encoded data to be used for evaluation by students
   (c) **fren.test.bpe** - byte-pair encoded data to be used for evaluation by course staff
5. **grade.py** - file to be used by students to evaluate translation on dev set and course staff to perform automated grading.
6. **README.md** - brief description of the task and the provided code.

## 9 Conclusion

This assignment has been carefully designed to provide students with hands-on experience implementing the transformer architecture. By breaking down the implementation into modular components, students can better understand how each part contributes to the overall model. The assignment balances theoretical understanding with practical implementation skills, preparing students for working with modern machine translation systems.

The progressive nature of the assignment, from basic components to the full architecture, allows students to debug and understand each part independently. The provided evaluation metrics and thresholds ensure that students can verify their implementations while encouraging optimization and experimentation.

## References


[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.

[2] Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., Ahmed, A. (2020). Big Bird: Transformers for Longer Sequences.

[3] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32.

[4] Lin, T., Wang, Y., Liu, X., Qiu, X. (2022). A survey of transformers. *AI Open*, 3, 111-132.

[5] Haddow, B., Bawden, R., Barone, A. V. M., Helcl, J., Birch, A. (2022). Survey of Low-Resource Machine Translation. *Computational Linguistics*, 48(3), 673-724.

[6] Kocmi, T., Avramidis, E., Bawden, R., Bojar, O., Dvorkovich, A., Federmann, C., ... & Suzuki, J. (2023). Findings of the 2023 Conference on Machine Translation (WMT23): LLMs Are Here But Not Quite There Yet. *Proceedings of the Eighth Conference on Machine Translation (WMT)*, 1-42.

[7] Vig, J. (2019). Visualizing Attention in Transformer-Based Language Representation Models. *Technical Report*, Palo Alto Research Center.

[8] Bloem, P. (2019). Transformers from scratch. *Retrieved from* `https://peterbloem.nl/blog/transformers`.

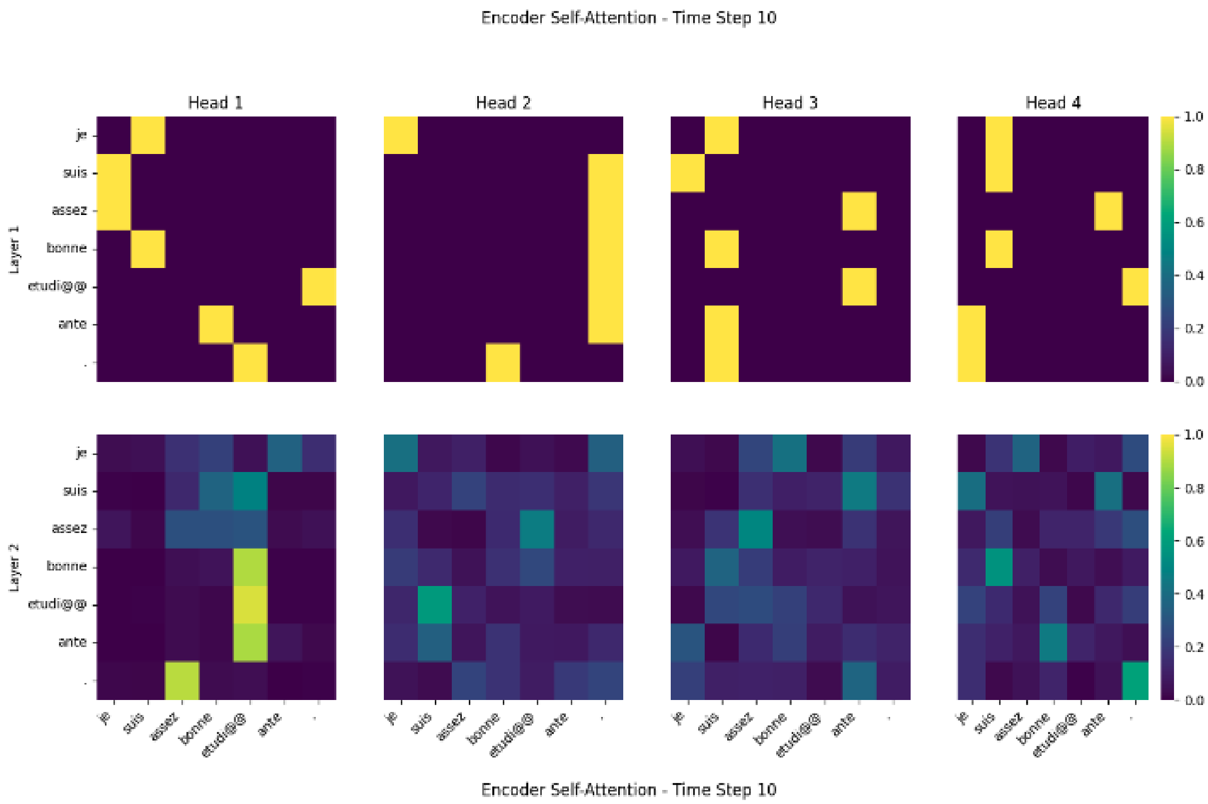


Figure 2: Attention weights for: "je suis assez bonne etudi@@ ante ." after 10 epochs

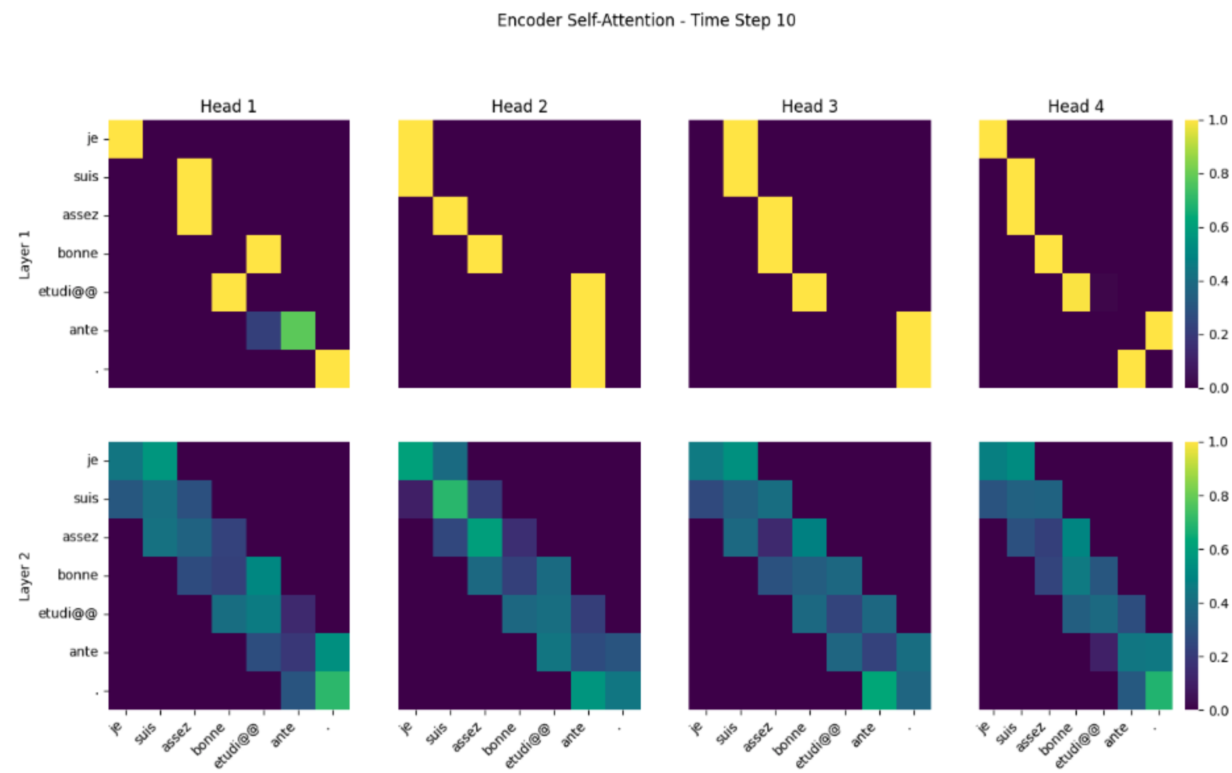


Figure 3: Attention weights for: "je suis assez bonne etudi@@ ante ." after 10 epochs (diagonal masking)