# NMT Experiment Report: Chinese-to-English Translation based on RNN and Transformer

- GitHub: https://github.com/YueLin301/NLP_final_project_NMT
- Checkpoints One Drive: SLAI_NLP_FinalProject_Checkpoints

## 1. Project Overview

The primary objective of this project is to develop and analyze Neural Machine Translation (NMT) systems capable of translating Chinese sentences into English. Specifically, we aim to implement two distinct architectures from scratch: a Recurrent Neural Network (RNN) based on Gated Recurrent Units (GRU) with Attention mechanisms, and a Transformer model based on the "Attention Is All You Need" paper.

Beyond implementation, a critical component of this project is to conduct a comparative analysis of these architectures. We evaluate their performance not only based on final translation quality (BLEU scores) but also on training stability, convergence speed, and the impact of various architectural decisions such as attention types and normalization strategies.

This report details the theoretical underpinnings, implementation specifics, experimental setup, and a comprehensive analysis of the results obtained from training these models on a 100k Chinese-English parallel corpus.

## 2. Model Architectures & Implementation Details

All models were implemented using PyTorch, emphasizing a modular design to facilitate ablation studies and component swapping.

### 2.1 RNN-based NMT Model (Seq2Seq with Attention)

Our RNN baseline adopts the Encoder-Decoder framework, enhanced with an attention mechanism to handle the variable-length nature of translation tasks and alleviate the information bottleneck.

#### 2.1.1 Encoder

The encoder is responsible for digesting the source Chinese sentence into a sequence of context-aware hidden states.

- **Embedding Layer**: Maps discrete token indices $(x_1, \ldots, x_T)$ to dense vectors of dimension $d_{emb} = 256$.
- **GRU Layers**: We utilize a 2-layer unidirectional GRU. The choice of GRU over LSTM was motivated by its simpler architecture (fewer gates), which often leads to faster training with comparable performance on smaller datasets.
  - Forward pass: $h_t = \mathrm{GRU}(e(x_t), h_{t-1})$.
  - The encoder outputs a sequence of hidden states $H = \{h_1, \ldots, h_T\}$, where each $h_t \in \mathbb{R}^{d_{hid}}$.
- **Dropout**: A dropout rate of 0.3 is applied to the embeddings and between GRU layers to mitigate overfitting.

### 2.1.2 Attention Mechanism

The core innovation in our RNN model is the Attention mechanism. Instead of relying on the final hidden state $h_T$ to capture the entire sentence meaning, the decoder attends to different parts of the source sentence at each step. We implemented and compared three specific scoring functions for calculating the alignment scores $e_{ij}$ between the decoder hidden state $s_{i-1}$ and encoder hidden state $h_j$:

1. **Dot-Product Attention**:
   $e_{ij} = s_{i-1}^T h_j$
   - *Pros*: Computationally efficient (matrix multiplication).
   - *Cons*: Requires encoder and decoder hidden dimensions to be identical; no learnable parameters to adapt the alignment space.
2. **General Attention**:
   $e_{ij} = s_{i-1}^T W_a h_j$
   - *Mechanism*: Introduces a learnable weight matrix $W_a \in \mathbb{R}^{d_{dec} \times d_{enc}}$.
   - *Pros*: Can handle different dimensions; learns a linear projection to align the spaces.
3. **Additive (Concat) Attention** (Bahdanau et al.):
   $e_{ij} = v_a^T \tanh(W_a[s_{i-1}; h_j])$
   - *Mechanism*: Concatenates states, passes them through a linear layer, a non-linear activation ( $\tanh$), and a final project vector $v_a$.
   - *Pros*: Highly expressive due to non-linearity; historically performs best for NMT.
   - *Cons*: Computationally more expensive.

The attention weights $\alpha_{ij}$ are obtained via Softmax: $\alpha_{ij} = \text{softmax}(e_{ij})$. The context vector $c_i$ is then the weighted sum: $c_i = \sum_j \alpha_{ij} h_j$.

### 2.1.3 Decoder

- **Input**: At step $i$, the decoder receives the embedding of the previous token $y_{i-1}$ concatenated with the context vector $c_i$.
  - Input dimension: $d_{emb} + d_{hid}$.
- **GRU**: Processes the concatenated input to update its hidden state $s_i$.
- **Output Projection**: A linear layer maps the concatenation of $[y_{i-1}, s_i, c_i]$ to the target vocabulary size ($|V_{tgt}| \approx 29,005$), producing logits for the next token prediction.

## 2.2 Transformer-based NMT Model

We implemented a Transformer model that relies entirely on self-attention mechanisms, discarding recurrence and convolutions.

### 2.2.1 Positional Encoding

Since the Transformer has no inherent sense of order, we inject positional information into the embeddings. We used the standard fixed sinusoidal encodings:
$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$
$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$
This allows the model to extrapolate to sequence lengths longer than those seen during training.

### 2.2.2 Encoder Layer

The encoder consists of a stack of $N = 3$ identical layers. Each layer has two sub-layers:

1. **Multi-Head Self-Attention (MHA)**: Allows the model to jointly attend to information from different representation subspaces at different positions. We used $h = 4$ heads.
    - $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
2. **Position-wise Feed-Forward Network (FFN)**: A fully connected feed-forward network applied to each position separately and identically.
    - $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$

### 2.2.3 Decoder Layer

The decoder is also a stack of $N = 3$ layers. In addition to the two sub-layers in the encoder, it inserts a third sub-layer:

- **Masked Multi-Head Attention**: Prevents positions from attending to subsequent positions (i.e., ensuring predictions for position $i$ can depend only on known outputs at positions less than $i$).
- **Encoder-Decoder Attention**: Performs multi-head attention over the output of the encoder stack (Keys and Values) using the decoder's previous layer output as Queries.

### 2.2.4 Normalization Experiments

Normalization is crucial for training deep Transformers. We experimented with two variants:

- **LayerNorm (LN)**: $\text{LN}(x) = \frac{x-\mu}{\sigma} \cdot \gamma + \beta$. Standard in the original paper.
- **RMSNorm (Root Mean Square Norm)**: $\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{n}\|x\|_2^2 + \epsilon}} \cdot \gamma$. It simplifies LN by removing the mean subtraction, focusing only on re-scaling invariance. It is gaining popularity in recent LLMs (e.g., LLaMA) for its computational efficiency.

# 3. Experimental Setup

## 3.1 Dataset & Preprocessing

- **Source Data**: 100,000 sentence pairs from the provided `train_100k.jsonl`.
- **Validation**: 500 pairs (`valid.jsonl`).
- **Tokenization**:
    - **Chinese**: Processed using `jieba`, a statistical library for accurate Chinese word segmentation.
    - **English**: Processed using standard regular expressions to separate punctuation from words, followed by lowercasing.

- **Vocabulary Construction**: We built vocabularies for both languages, filtering out rare tokens (frequency < 2) to reduce noise.
    - Source Vocab Size: 30,000 (truncated max)
    - Target Vocab Size: ~29,005
    - Special Tokens: `<pad>` (0), `<sos>` (1), `<eos>` (2), `<unk>` (3).

## 3.2 Hyperparameters

To ensure a fair comparison within our computational constraints (MPS/GPU memory), we standardized dimensions where possible:

- **Embedding Dimension**: 256
- **Hidden Dimension**: 512
- **Batch Size**: 64
- **Epochs**: 3 (sufficient for comparative trend analysis)
- **Optimizer**: Adam ($\beta_1 = 0.9, \beta_2 = 0.999$, $\epsilon = 1e - 8$).
- **Learning Rate**:
    - RNN: $1e - 3$ (Standard for RNNs).
    - Transformer: $5e - 4$ (Transformers typically require smaller LRs or warmup schedules).

# 4. Results & Detailed Analysis

## 4.1 Comparison of Attention Mechanisms in RNN

We trained three RNN variants with identical hyperparameters, differing only in the attention scoring function.

| Attention Mechanism | Best Valid Loss | Best BLEU Score | Convergence Speed |
|---|---|---|---|
| **Dot-product** | 6.27 | 3.95 | Fast (Simple Ops) |
| **General** | 6.30 | 5.30 | Medium |
| **Concat (Additive)** | **6.12** | **6.41** | Slowest (tanh + linear) |

**Deep Dive Analysis**:

- **Dot-product Attention**: While computationally the most efficient, it performed the worst. This suggests that a simple dot product is insufficient to capture the complex semantic alignment between structurally distinct languages like Chinese and English. The lack of learnable weights in the scoring function limits its expressivity.
- **Concat Attention**: Achieved the highest BLEU score (6.41). The use of a multi-layer perceptron (Linear -> Tanh -> Linear) allows the model to learn highly non-linear alignment relationships. Despite being computationally heavier, the performance gain justifies the cost for this task.
- **Conclusion**: For Chinese-English NMT, the capacity to model complex alignments is critical. **Additive**

**attention is superior.**

## 4.2 Impact of Training Strategy: Teacher Forcing

We conducted a controlled experiment comparing a standard Teacher Forcing ratio (0.5) against a low ratio (0.1, essentially Free Running).

**Quantitative Results**:

- **High Teacher Forcing (0.5)**: Loss decreased monotonically from ~10.2 to ~6.0. The curve was smooth.
- **Free Running (0.1)**: Loss oscillated wildly between 9.0 and 11.0, failing to converge significantly even after 3 epochs.

**Theoretical Explanation**:
This phenomenon is a classic example of the **"Exposure Bias"** problem combined with the difficulty of "Cold Start" in RL-like generation.

1. In the early stages (Epoch 1), the model's predictions are essentially random noise.
2. Under **Free Running**, the decoder consumes this noise as input for the next step.
3. This leads to a "compounding error" effect: once the model generates one wrong token, the state trajectory diverges entirely from the valid manifold. The model generates a sequence of nonsense, and the gradients derived from this nonsense are high-variance and uninformative.
4. **Teacher Forcing** acts as "training wheels," correcting the trajectory at every step, ensuring the model learns the correct conditional probability $P(y_t|y_{<t}, x)$ given the *true* history.

## 4.3 Transformer Ablation: LayerNorm vs. RMSNorm

| Normalization | Best Valid Loss | Best BLEU Score |
| --- | --- | --- |
| **LayerNorm** | **5.01** | 4.11 |
| **RMSNorm** | 5.51 | **4.77** |

**Analysis**:

- **Loss vs. BLEU Discrepancy**: A fascinating finding is that LayerNorm achieved better cross-entropy loss (better probability estimation), while RMSNorm achieved better BLEU (better discrete generation).
- **RMSNorm Efficacy**: RMSNorm simplifies the normalization by enforcing scale invariance without shifting the mean. Our results suggest this inductive bias might be beneficial for the Transformer's optimization landscape in NMT, allowing it to focus on relative token relationships rather than absolute activation magnitudes.
- **Conclusion**: **RMSNorm** is a highly competitive alternative to LayerNorm, offering slight generation quality improvements with reduced computational overhead.

## 4.4 Architecture War: RNN vs. Transformer

**Comparative Metrics (Epoch 3)**:

- **RNN (Concat)**: BLEU **6.41**, Loss 6.12
- **Transformer (RMSNorm)**: BLEU 4.77, Loss **5.51**

**Synthesized Analysis**:

1. **The "RNN Wins Early" Phenomenon**: Contrary to the general consensus that Transformers dominate, our RNN outperformed the Transformer in BLEU after 3 epochs. This is attributable to **Inductive Bias**. RNNs process data sequentially, which inherently aligns with the sequential nature of language. They learn local dependencies (like n-grams) extremely quickly. Transformers, lacking this bias (relying on positional encodings), require more data and time to "learn how to be sequential."
2. **The Transformer's Potential**: The Transformer achieved a significantly lower loss (5.51 vs 6.12). Lower loss indicates the model is less "surprised" by the test data and assigns higher probability to the true targets. The lower BLEU suggests that while its probability distribution is better, its greedy decoding (taking the max prob) hasn't yet sharpened enough to produce contiguous correct n-grams.
3. **Verdict**: For rapid prototyping with limited compute/time, RNNs are robust. For scaling up (more epochs, larger data), the Transformer's lower loss trajectory indicates a much higher performance ceiling.

# 5. Case Studies & Error Analysis

We generated translations for the test set using our best models.

| Source (中文) | Model | Translation | Error Analysis |
|---|---|---|---|
| 由于经济危机，很多人失去了工作。 | **RNN (Concat)** | `In the crisis, many people, many jobs.` | **Partial Success**: The model correctly identified "crisis", "many people", and "jobs". However, the syntax is broken ("many jobs" instead of "lost jobs"). This reflects the RNN's ability to capture keywords via attention but struggle with complex grammar in early training. |
| | **Transformer** | `The economic crisis has been a lot of economic crisis.` | **Repetition Loop**: The model generated fluent English phrases but got stuck in a loop. This is a common failure mode in Transformers when the attention mechanism hasn't fully converged to distinct positions. |
| 历史总是惊人的相似。 | **RNN (Dot)** | `Historical history is history.` | **Tautology**: The model recognized the topic "history" but failed to translate the predicate "similar", resorting to repeating the subject. |
| | **Transformer** | `The situation is not a mistake.` | **Hallucination**: The generated sentence is fluent and grammatical but semantically unrelated to the source. This indicates the decoder is functioning as a language model but ignoring the encoder's context. |

# 6. Development Challenges & Solutions

Developing a high-performance NMT system from scratch involves navigating numerous technical pitfalls. Here, we document the key challenges encountered and our solutions, providing insights for future implementation.

## 6.1 MPS (Apple Silicon) Device Compatibility

- **Challenge**: While training on Mac M2/M3 chips using the `mps` backend, we frequently encountered `RuntimeError: Placeholder storage has not been allocated on MPS device!`. This error often occurs when tensors involved in an operation are scattered across CPU and MPS devices, or when specific operations (like embedding lookup) receive input indices that haven't been explicitly moved to the device.
- **Solution**: We enforced strict device management in the training loop and model forward passes. Specifically, we added explicit `.to(device)` calls for all inputs generated dynamically during decoding (e.g., `input = tgt[:, t].to(self.device)` in RNN decoder loop). We also ensured that the `Vocabulary` and `Dataset` classes yielded tensors that were device-agnostic until the `DataLoader` handed them to the training loop.

## 6.2 PyTorch Serialization Security

- **Challenge**: During inference, loading checkpoints failed with `_pickle.UnpicklingError: Weights only load failed`. This was due to a recent PyTorch security update (version 2.6+) that defaults `weights_only=True` in `torch.load()`. Since our checkpoints included the entire `Vocabulary` object (a custom class), the loader rejected it as potentially unsafe code execution.
- **Solution**: We modified the inference script to explicitly set `weights_only=False` when loading the vocabulary files (`src_vocab.pt`, `tgt_vocab.pt`), as these are trusted local files generated by our own training script. For model weights, we kept `weights_only=True` for best practices.

### 6.3 Inference Configuration Mismatch

- **Challenge**: We trained multiple model variants (e.g., `rnn_dot.pt`, `rnn_concat.pt`). A naive inference script failed to load `rnn_concat.pt` because the model definition defaults to `dot` attention, resulting in a state dictionary key mismatch (missing weights for the concat attention MLP).
- **Solution**: We implemented an intelligent model loader in `inference.py`. It inspects the model filename (e.g., detecting "concat" in the string) to automatically instantiate the correct model architecture before loading weights. We also added command-line overrides (`--attention`, `--norm_type`) for manual control.

## 7. Implementation Source Code

To ensure reproducibility and transparency, we include the core implementation of our models and training logic below.

### 7.1 RNN Model (`src/models/rnn.py`)

This module implements the Encoder, Attention, Decoder, and the overall Seq2Seq container. Note the modular `Attention` class supporting three scoring methods.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import random

class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()
        self.hid_dim = hid_dim
        self.n_layers = n_layers

        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim, hid_dim, n_layers, dropout=dropout,
batch_first=True)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        # src = [batch size, src len]
        embedded = self.dropout(self.embedding(src))
        outputs, hidden = self.rnn(embedded)
        return outputs, hidden
```

```python

class Attention(nn.Module):
    def __init__(self, hid_dim, method='dot'):
        super().__init__()
        self.method = method # 'dot', 'general', 'concat'
        self.hid_dim = hid_dim

        if method == 'general':
            self.W = nn.Linear(hid_dim, hid_dim)
        elif method == 'concat': # Additive
            self.W = nn.Linear(hid_dim * 2, hid_dim)
            self.v = nn.Linear(hid_dim, 1, bias=False)

    def forward(self, hidden, encoder_outputs):
        # hidden = [1, batch size, hid dim] (last layer hidden state)
        # encoder_outputs = [batch size, src len, hid dim]

        batch_size = encoder_outputs.shape[0]
        src_len = encoder_outputs.shape[1]

        # Squeeze the hidden state to [batch size, hid dim]
        hidden = hidden.squeeze(0)

        if self.method == 'dot':
            # score = hidden * encoder_output
            score = torch.bmm(hidden.unsqueeze(1), encoder_outputs.permute(0, 2, 1))

        elif self.method == 'general':
            # score = hidden * W * encoder_output
            x = self.W(encoder_outputs)
            score = torch.bmm(hidden.unsqueeze(1), x.permute(0, 2, 1))

        elif self.method == 'concat':
            # score = v * tanh(W * [hidden; encoder_output])
            hidden_expanded = hidden.unsqueeze(1).repeat(1, src_len, 1)
            combined = torch.cat((hidden_expanded, encoder_outputs), dim=2)
            energy = torch.tanh(self.W(combined))
            score = self.v(energy).permute(0, 2, 1)

        return F.softmax(score, dim=2)

class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout, attention):
        super().__init__()
        self.output_dim = output_dim
        self.attention = attention
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim + hid_dim, hid_dim, n_layers, dropout=dropout,
batch_first=True)
        self.fc_out = nn.Linear(emb_dim + hid_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

```

```python
    def forward(self, input, hidden, encoder_outputs):
        input = input.unsqueeze(1)
        embedded = self.dropout(self.embedding(input))

        # Calculate attention weights
        attn_weights = self.attention(hidden[-1:], encoder_outputs)

        # Calculate context vector
        context = torch.bmm(attn_weights, encoder_outputs)

        # Concatenate embedding and context vector
        rnn_input = torch.cat((embedded, context), dim=2)

        output, hidden = self.rnn(rnn_input, hidden)

        prediction = torch.cat((output, context, embedded), dim=2).squeeze(1)
        prediction = self.fc_out(prediction)

        return prediction, hidden

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, tgt, teacher_forcing_ratio=0.5):
        batch_size = src.shape[0]
        tgt_len = tgt.shape[1]
        vocab_size = self.decoder.output_dim

        outputs = torch.zeros(batch_size, tgt_len, vocab_size).to(self.device)
        encoder_outputs, hidden = self.encoder(src)
        input = tgt[:, 0] # Start token

        for t in range(1, tgt_len):
            output, hidden = self.decoder(input, hidden, encoder_outputs)
            outputs[:, t] = output
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            input = tgt[:, t] if teacher_force else top1
            input = input.to(self.device)

        return outputs
```

## 7.2 Transformer Model (`src/models/transformer.py`)

This module implements the full Transformer architecture, including Positional Encoding, Multi-Head Attention, and Layer/RMS Normalization.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class RMSNorm(nn.Module):
    def __init__(self, d_model, eps=1e-8):
        super().__init__()
        self.eps = eps
        self.scale = nn.Parameter(torch.ones(d_model))

    def forward(self, x):
        norm = x.norm(keepdim=True, dim=-1)
        return x / (norm + self.eps) * self.scale

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_head, dropout=0.1):
        super().__init__()
        self.d_k = d_model // n_head
        self.n_head = n_head
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.fc = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.scale = torch.sqrt(torch.FloatTensor([self.d_k]))

    def forward(self, query, key, value, mask=None):
        batch_size = query.shape[0]
        Q = self.w_q(query).view(batch_size, -1, self.n_head, self.d_k).permute(0, 2, 1, 3)
        K = self.w_k(key).view(batch_size, -1, self.n_head, self.d_k).permute(0, 2, 1, 3)
        V = self.w_v(value).view(batch_size, -1, self.n_head, self.d_k).permute(0, 2, 1, 3)

        energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.scale.to(query.device)
        if mask is not None:
            energy = energy.masked_fill(mask == 0, -1e10)

        attention = torch.softmax(energy, dim=-1)
        x = torch.matmul(self.dropout(attention), V)
        x = x.permute(0, 2, 1, 3).contiguous().view(batch_size, -1, self.n_head * self.d_k)
        return self.fc(x), attention

class Transformer(nn.Module):
    # ... (Initialization code omitted for brevity, see source file) ...

    def make_src_mask(self, src, pad_idx):
        return (src != pad_idx).unsqueeze(1).unsqueeze(2)

```

```python
    def make_tgt_mask(self, tgt, pad_idx):
        tgt_pad_mask = (tgt != pad_idx).unsqueeze(1).unsqueeze(2)
        tgt_len = tgt.shape[1]
        tril = torch.tril(torch.ones((tgt_len, tgt_len), device=self.device)).bool()
        return tgt_pad_mask & tril.unsqueeze(0).unsqueeze(0)

    def forward(self, src, tgt):
        src_mask = self.make_src_mask(src, 0)
        tgt_mask = self.make_tgt_mask(tgt, 0)

        # Positional Encoding + Embedding
        batch_size, src_len = src.shape
        batch_size, tgt_len = tgt.shape
        pos_src = torch.arange(0, src_len).unsqueeze(0).repeat(batch_size, 1).to(self.device)
        src = self.dropout((self.src_embedding(src) * self.scale) + self.pos_embedding(pos_src))

        pos_tgt = torch.arange(0, tgt_len).unsqueeze(0).repeat(batch_size, 1).to(self.device)
        tgt = self.dropout((self.tgt_embedding(tgt) * self.scale) + self.pos_embedding(pos_tgt))

        for layer in self.encoder_layers:
            src = layer(src, src_mask)

        for layer in self.decoder_layers:
            tgt = layer(tgt, src, src_mask, tgt_mask)

        return self.fc_out(tgt)
```

## 7.3 Training Loop (`src/train.py`)

This script manages the training epoch, validation, and history logging.

```python
class Trainer:
    # ... (Init omitted) ...

    def train_epoch(self, clip, update_history_callback=None):
        self.model.train()
        epoch_loss = 0

        for i, (src, tgt) in enumerate(tqdm(self.train_loader, desc="Training")):
            src = src.to(self.device)
            tgt = tgt.to(self.device)

            self.optimizer.zero_grad()

            if isinstance(self.model, nn.Transformer) or hasattr(self.model, 'encoder_layers'):
                # Transformer Training: Target shifting
```

```
                tgt_input = tgt[:, :-1]
                output = self.model(src, tgt_input)
                output_dim = output.shape[-1]
                output = output.contiguous().view(-1, output_dim)
                tgt_output = tgt[:, 1:].contiguous().view(-1)
                loss = self.criterion(output, tgt_output)
            else:
                # RNN Training
                output = self.model(src, tgt, teacher_forcing_ratio=self.config.TEACHER_FORCING_RATIO)
                output_dim = output.shape[-1]
                output = output[:, 1:].contiguous().view(-1, output_dim)
                tgt_output = tgt[:, 1:].contiguous().view(-1)
                loss = self.criterion(output, tgt_output)

            loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), clip)
            self.optimizer.step()

            epoch_loss += loss.item()
            if update_history_callback:
                update_history_callback(loss.item())

        return epoch_loss / len(self.train_loader)
```

# 9. Comprehensive Comparison and Theoretical Reflections

In this section, we synthesize our experimental findings with the theoretical concepts discussed in Section 2, addressing the specific comparison dimensions outlined in the course requirements.

## 9.1 Model Architecture Analysis: Serial vs. Parallel (Architecture)

- **Sequential vs. Parallel Computation**:
  - **RNN**: Processes tokens one by one ($t = 1, 2, \ldots$). To compute state $h_t$, it strictly depends on $h_{t-1}$. This inherent sequentiality limits GPU parallelization, as computation cannot proceed to the next step until the previous one is finished.
  - **Transformer**: Processes all tokens in the sequence simultaneously using matrix operations. The Self-Attention mechanism allows for massive parallelization ($O(1)$ sequential operations), significantly speeding up training on modern hardware.
- **Recurrence vs. Self-Attention**:
  - **RNN (Recurrence)**: Relies on compressing history into a hidden state vector. While effective for short sequences, this compression becomes a bottleneck ("forgetting") for long sequences.
  - **Transformer (Self-Attention)**: Allows each token to directly attend to any other token in the sequence, modeling global dependencies without compression loss.

## 9.2 Training Efficiency: Convergence & Speed (Training Efficiency)

- **Convergence Speed**:
  - Our experiments showed RNNs converging faster in terms of BLEU score in early epochs (Epoch 1-3). This is due to their **Inductive Bias** for sequential data—they "know" word order matters without needing to learn it.
  - Transformers, lacking this bias (relying on Positional Encodings), take longer to learn the concept of order but eventually reach a higher performance ceiling.
- **Hardware Requirements**: Transformer training is more memory-intensive due to the $O(N^2)$ complexity of the attention matrix (vs $O(N)$ for RNNs), requiring more VRAM for long sequences.

## 9.3 Translation Performance: BLEU & Accuracy (Translation Performance)

- **BLEU Score**: In our limited training (3 epochs), the RNN with Concat Attention achieved the highest BLEU (6.41), outperforming the Transformer (4.77). This highlights that for low-resource or short-training scenarios, optimized RNNs are surprisingly competitive.
- **Accuracy & Fluency**: Qualitative analysis reveals that Transformers produce more fluent English grammar (better Language Modeling) but suffer from hallucinations in early stages. RNNs are more faithful to the source keywords but often produce broken grammar or repetitive phrases.

## 9.4 Scalability & Generalization (Scalability & Generalization)

- **Long Sentence Handling**: Transformers theoretically handle long sentences better due to the direct attention path ($O(1)$ distance). RNNs suffer from vanishing gradients over long paths ($O(N)$ distance).
- **Low-Resource Scenarios**: Our results suggest RNNs generalize better when data or training time is scarce (Low-Resource), whereas Transformers are data-hungry and shine in high-resource settings.

## 9.5 Practical Trade-offs (Real-world Trade-offs)

- **Model Size**: Our Transformer implementation ($d_{model} = 256$) is parameter-efficient but memory-heavy during training. RNNs are memory-efficient but slow in inference.
- **Inference Latency**: RNN generation is sequential and cannot be parallelized, leading to higher latency for long outputs. Transformers can cache Key/Value pairs (KV-Cache) to speed up decoding, though they are still fundamentally autoregressive during generation.
- **Implementation Difficulty**: RNNs are conceptually simpler to implement. Transformers require careful handling of masks, positional encodings, and numerical stability (e.g., Warmup schedulers, Norm placement), making them harder to tune.

# 10. Conclusion and Future Directions and Future Directions

This project provided a rigorous, hands-on comparison of two paradigms in NMT.

**Summary of Achievements**:

1. Successfully implemented functional RNN and Transformer NMT systems from scratch.

2. Demonstrated the superiority of **Concat Attention** for RNNs.
3. Validated the necessity of **Teacher Forcing** for stable training.
4. Highlighted **RMSNorm** as an effective optimization technique for Transformers.
5. Observed the trade-off between RNNs' fast convergence and Transformers' high capacity.

**Limitations**:

- **Vocab Size**: A 30k vocabulary with word-level tokenization leads to many `<unk>` tokens, limiting translation quality for rare words.
- **Training Time**: 3 epochs are insufficient for the Transformer to fully converge.

**Future Work**:

1. **Subword Tokenization (BPE)**: Replacing `jieba`/regex with Byte-Pair Encoding (BPE) would eliminate `<unk>` tokens and significantly improve the translation of rare words and names.
2. **Beam Search**: Implementing Beam Search (e.g., width 5) during inference would help models recover from greedy errors and reduce repetition.
3. **Extended Training**: Training the Transformer for 20+ epochs with a learning rate scheduler (Warmup + Decay) would likely allow it to surpass the RNN significantly.

---

*Appendices: Full Inference Logs*

```
(basic) yue@Yues-Mac-mini NMT_ly % python run_all_inference.py
==================================================
🚀 Batch Inference on All Trained Models
==================================================


🔍 Testing Model: rnn_concat.pt (rnn)
----------------------------------------
Loading vocabs from checkpoints/src_vocab.pt and checkpoints/tgt_vocab.pt
Loading model from checkpoints/rnn_concat.pt
Loading RNN with attention: concat

==============================
Running Inference Examples (Model: rnn)
==============================

Building prefix dict from the default dictionary ...
Loading model from cache /var/folders/z2/4sp579091154mcqmms0fk76c0000gn/T/jieba.cache
Loading model cost 0.269 seconds.
Prefix dict has been built successfully.
Source: 今天天气很好。
Translation: <unk> is is.
------------------------------
Source: 我喜欢学习自然语言处理。
Translation: I learn to learn to the to the.
------------------------------
```

```
Source: 这本书很有趣。
Translation: That is interesting interesting.
------------------------------
Source: 由于经济危机，很多人失去了工作。
Translation: In the crisis, many people, many jobs.
------------------------------
Source: 我们必须采取行动保护环境。
Translation: We must ensure that we must ensure.
------------------------------
Source: 人工智能正在改变世界。
Translation: AI is changing world changing world.
------------------------------
Source: 你会说英语吗?
Translation: You can be????
------------------------------
Source: 这是一个非常复杂的问题。
Translation: It is a complicated problem.
------------------------------
Source: 我们需要更多的时间来完成这个项目。
Translation: We need more ambitious program.
------------------------------
Source: 历史总是惊人的相似。
Translation: History is often examples of history.
------------------------------



🔍 Testing Model: rnn_dot.pt (rnn)
----------------------------------------
Loading vocabs from checkpoints/src_vocab.pt and checkpoints/tgt_vocab.pt
Loading model from checkpoints/rnn_dot.pt
Loading RNN with attention: dot

==============================
Running Inference Examples (Model: rnn)
==============================

Building prefix dict from the default dictionary ...
Loading model from cache
/var/folders/z2/4sp579091154mcqmms0fk76c0000gn/T/jieba.cache
Loading model cost 0.266 seconds.
Prefix dict has been built successfully.
Source: 今天天气很好。
Translation: The is is a.
------------------------------
Source: 我喜欢学习自然语言处理。
Translation: I my own to the.
------------------------------
Source: 这本书很有趣。
Translation: The is a.
------------------------------
Source: 由于经济危机，很多人失去了工作。
```

```
Translation: For many many many many people many people are not.
------------------------------
Source: 我们必须采取行动保护环境。
Translation: We must must be to to.
------------------------------
Source: 人工智能正在改变世界。
Translation: Artificial learning is AI.
------------------------------
Source: 你会说英语吗?
Translation: Can you you you?
------------------------------
Source: 这是一个非常复杂的问题。
Translation: This is a.
------------------------------
Source: 我们需要更多的时间来完成这个项目。
Translation: We need more more more than the.
------------------------------
Source: 历史总是惊人的相似。
Translation: Historical history is history.
------------------------------



🔍 Testing Model: rnn_free.pt (rnn)
----------------------------------------
Loading vocabs from checkpoints/src_vocab.pt and checkpoints/tgt_vocab.pt
Loading model from checkpoints/rnn_free.pt
Loading RNN with attention: dot

==============================
Running Inference Examples (Model: rnn)
==============================

Building prefix dict from the default dictionary ...
Loading model from cache
/var/folders/z2/4sp579091154mcqmms0fk76c0000gn/T/jieba.cache
Loading model cost 0.268 seconds.
Prefix dict has been built successfully.
Source: 今天天气很好。
Translation: The.
------------------------------
Source: 我喜欢学习自然语言处理。
Translation: I have to.
------------------------------
Source: 这本书很有趣。
Translation: That is
------------------------------
Source: 由于经济危机，很多人失去了工作。
Translation: Since the crisis crisis.
------------------------------
Source: 我们必须采取行动保护环境。
Translation: We must.
```

```
------------------------------
Source: 人工智能正在改变世界。
Translation: The AI ’ s
------------------------------
Source: 你会说英语吗?
Translation: Who!
------------------------------
Source: 这是一个非常复杂的问题。
Translation: This is a..
------------------------------
Source: 我们需要更多的时间来完成这个项目。
Translation: We more more..
------------------------------
Source: 历史总是惊人的相似。
Translation: Historical.
------------------------------



🔍 Testing Model: rnn_general.pt (rnn)
----------------------------------------
Loading vocabs from checkpoints/src_vocab.pt and checkpoints/tgt_vocab.pt
Loading model from checkpoints/rnn_general.pt
Loading RNN with attention: general

==============================
Running Inference Examples (Model: rnn)
==============================

Building prefix dict from the default dictionary ...
Loading model from cache
/var/folders/z2/4sp579091154mcqmms0fk76c0000gn/T/jieba.cache
Loading model cost 0.264 seconds.
Prefix dict has been built successfully.
Source: 今天天气很好。
Translation: <unk> is..
------------------------------
Source: 我喜欢学习自然语言处理。
Translation: I am to the the.
------------------------------
Source: 这本书很有趣。
Translation: This is a..
------------------------------
Source: 由于经济危机，很多人失去了工作。
Translation: Since the,, people are working to
------------------------------
Source: 我们必须采取行动保护环境。
Translation: We must must be to.
------------------------------
Source: 人工智能正在改变世界。
Translation: AI is is the world.
------------------------------
```

```
Source: 你会说英语吗?
Translation: You you say that?
------------------------------
Source: 这是一个非常复杂的问题。
Translation: This is a a problem.
------------------------------
Source: 我们需要更多的时间来完成这个项目。
Translation: We need to be to.
------------------------------
Source: 历史总是惊人的相似。
Translation: History was a.
------------------------------



🔍 Testing Model: trans_layernorm.pt (transformer)
----------------------------------------
Loading vocabs from checkpoints/src_vocab.pt and checkpoints/tgt_vocab.pt
Loading model from checkpoints/trans_layernorm.pt
Loading Transformer with norm_type: layernorm

==============================
Running Inference Examples (Model: transformer)
==============================

Building prefix dict from the default dictionary ...
Loading model from cache /var/folders/z2/4sp579091154mcqmms0fk76c0000gn/T/jieba.cache
Loading model cost 0.270 seconds.
Prefix dict has been built successfully.
Source: 今天天气很好。
Translation: The first is not a good thing.
------------------------------
Source: 我喜欢学习自然语言处理。
Translation: I am not just my friends.
------------------------------
Source: 这本书很有趣。
Translation: The first thing is the first.
------------------------------
Source: 由于经济危机，很多人失去了工作。
Translation: The economic crisis has been a lot of economic crisis.
------------------------------
Source: 我们必须采取行动保护环境。
Translation: We need to ensure that we must need to ensure that we must be able to achieve.
------------------------------
Source: 人工智能正在改变世界。
Translation: The world ’ s biggest challenge is not the world.
------------------------------
Source: 你会说英语吗?
Translation: So what is you?
------------------------------
```

```
Source: 这是一个非常复杂的问题。
Translation: The problem is that the problem is.
------------------------------
Source: 我们需要更多的时间来完成这个项目。
Translation: The goal should be to achieve this goal.
------------------------------
Source: 历史总是惊人的相似。
Translation: The situation is not a mistake.
------------------------------



🔍 Testing Model: trans_rmsnorm.pt (transformer)
----------------------------------------
Loading vocabs from checkpoints/src_vocab.pt and checkpoints/tgt_vocab.pt
Loading model from checkpoints/trans_rmsnorm.pt
Loading Transformer with norm_type: rmsnorm

==============================
Running Inference Examples (Model: transformer)
==============================

Building prefix dict from the default dictionary ...
Loading model from cache /var/folders/z2/4sp579091154mcqmms0fk76c0000gn/T/jieba.cache
Loading model cost 0.263 seconds.
Prefix dict has been built successfully.
Source: 今天天气很好。
Translation: The same is not.
------------------------------
Source: 我喜欢学习自然语言处理。
Translation: The <unk> of the <unk> <unk> <unk> <unk>?
------------------------------
Source: 这本书很有趣。
Translation: The same is not.
------------------------------
Source: 由于经济危机,很多人失去了工作。
Translation: The world is not a new role.
------------------------------
Source: 我们必须采取行动保护环境。
Translation: The same is not.
------------------------------
Source: 人工智能正在改变世界。
Translation: The world is not a new.
------------------------------
Source: 你会说英语吗?
Translation: <unk> <unk> <unk>?
------------------------------
Source: 这是一个非常复杂的问题。
Translation: The world is not.
------------------------------
Source: 我们需要更多的时间来完成这个项目。
```

```
Translation: The same is not a result.
------------------------------
Source: 历史总是惊人的相似。
Translation: The same is not.
------------------------------
```