DS5983
PA2: The Transformer Architecture
Yunyu Guo

This report documents the implementation of two key components for a Transformer-based neural machine translation system: (1) a greedy decoding function for sentence translation (2) a comprehensive hyperparameter optimization framework. The work demonstrates practical application of the Transformer architecture for German-to-English translation with systematic performance evaluation across different model configurations.

1. Greedy Decoding Implementation for Sentence Translation
The translate_sentence function implements autoregressive text generation using greedy decoding, where the model selects the most probable next token at each step. This approach provides deterministic, fast inference suitable for real-time translation applications.

```
def translate_sentence(model, sentence, vocab_src, vocab_tgt, max_length=50):
```

1.2.2 Implementation Steps
Step 1: Model Preparation

```
model.eval()  # Set to evaluation mode
```

Disables dropout and batch normalization training behavior
Ensures consistent inference results
Step 2: Source Preprocessing

```
src_tokens = [vocab_src['<bos>']] + [vocab_src[token] for token in tokenizer_de(sentence)] +
[vocab_src['<eos>']]
src_tensor = torch.tensor(src_tokens, dtype=torch.long).unsqueeze(0).to(device)
```

Tokenizes German input using spaCy tokenizer
Adds beginning-of-sequence (<bos>) and end-of-sequence (<eos>) tokens
Converts to tensor format with batch dimension

Step 3: Encoder Processing

```
memory, src_mask = model.encode(src_tensor)
```

Processes source sequence through encoder stack
Returns encoded representations (memory) and attention mask

Step 4: Autoregressive Decoding

```
tgt_tokens = [vocab_tgt['<bos>']]
for _ in range(max_length):
  tgt_tensor = torch.tensor(tgt_tokens, dtype=torch.long).unsqueeze(0).to(device)
  output = model.decode(tgt_tensor, memory, src_mask)
  next_token = output[:, -1, :].argmax(-1).item()
  tgt_tokens.append(next_token)
  if next_token == vocab_tgt['<eos>']:
```

```
        break
```

Greedy Selection: Chooses token with highest probability at each step
Autoregressive: Uses previously generated tokens as input
Termination: Stops when <eos> generated or max length reached

Step 5: Post-processing

```
translated_tokens = [vocab_tgt.lookup_token(token) for token in tgt_tokens[1:-1]]
translated_sentence = ' '.join(translated_tokens)
```

Converts token IDs back to words
Removes special tokens (<bos>, <eos>)
Joins tokens into final sentence

1.3 Key Features
Deterministic Output: Greedy decoding ensures reproducible translations
Efficient Inference: O(n) time complexity for sequence length n
Memory Efficient: Reuses encoder output for all decoding steps
Robust Termination: Multiple stopping criteria prevent infinite loops

1.4 Limitations
Limited Diversity: May miss better translations due to greedy selection
Exposure Bias: Training uses teacher forcing, but inference is autoregressive
No Backtracking: Cannot correct early poor decisions

2. Hyperparameter Optimization Framework
2.1 Overview
A systematic experimental framework was developed to evaluate the impact of key Transformer hyperparameters on translation quality. The framework enables automated training, evaluation, and comparison across multiple model configurations.

2.2 Experimental Design
2.2.1 Hyperparameter Space
Selected Parameters:
Number of Attention Heads: [4, 8, 16]
Number of Layers: [3, 6, 9]
Learning Rate: [0.0001, 0.0005, 0.001]
Batch Size: [32, 64, 128]
Rationale:
Covers range from lightweight to computationally intensive models
Learning rates span typical optimization ranges for Transformers
Batch sizes accommodate different memory constraints

2.2.2 Experimental Configuration

```
hyperparameter_configs = {
    'num_heads': [4, 8, 16],
```

```
    'num_layers': [3, 6, 9],
    'learning_rate': [0.0001, 0.0005, 0.001],
    'batch_size': [32, 64, 128]
}

# Total combinations: 3 × 3 × 3 × 3 = 81
# Limited to 15 experiments for computational efficiency
experiments = generate_configs(hyperparameter_configs, max_experiments=15)
```

2.3 Implementation Framework

2.3.1 Model Factory Pattern

```
def create_model_with_config(config, src_vocab_size, tgt_vocab_size, pad_idx):
    model = Transformer(
        src_vocab_size=src_vocab_size,
        tgt_vocab_size=tgt_vocab_size,
        d_model=512,                    # Fixed
        N=config['num_layers'],         # Variable
        n_heads=config['num_heads'],    # Variable
        d_ff=2048,                      # Fixed
        max_seq_length=5000,            # Fixed
        dropout=0.1,                    # Fixed
        pad_idx=pad_idx
    )
    return model.to(device)
```

2.3.2 Training Protocol

Training Configuration:

Epochs per Experiment: 10 (reduced to 5 for efficiency)

Optimization: Adam optimizer with β_1=0.9, β_2=0.98

Loss Function: CrossEntropyLoss with padding token masking

Gradient Clipping: Max norm = 1.0

Evaluation: Validation loss monitoring

2.3.3 Metrics Collection

```
def train_with_metrics(model, train_iterator, valid_iterator, optimizer, criterion, config,
num_epochs=10):
    return {
        'config': config,
        'train_losses': train_losses,
        'val_losses': val_losses,
        'best_val_loss': best_val_loss,
        'final_train_loss': train_losses[-1],
        'final_val_loss': val_losses[-1]
    }
```

2.4 Experimental Results Analysis
2.4.1 Performance Metrics
Primary Metric: Validation loss (cross-entropy)Secondary Metrics:
Training loss convergence
Training stability
Computational efficiency

2.4.2 Analysis Framework
Statistical Analysis:
```
def analyze_results(results):
    # Convert to DataFrame for analysis
    # Sort by validation performance
    # Compute correlations between hyperparameters and performance
    # Identify optimal configurations
```
Visualization Components:
Bar Charts: Average performance by hyperparameter value
Training Curves: Loss progression for top configurations
Correlation Analysis: Hyperparameter impact quantification

2.5 Key Findings
2.5.1 Expected Hyperparameter Effects
Number of Attention Heads:
More heads generally improve performance up to a point
Diminishing returns beyond 8-16 heads
Computational cost increases linearly
Number of Layers:
Deeper models can capture more complex patterns
Risk of overfitting with limited training data
Training instability in very deep networks
Learning Rate:
Critical for convergence speed and final performance
Too high: unstable training, poor convergence
Too low: slow convergence, potential underfitting
Optimal range: 0.0001-0.001 for Transformers
Batch Size:
Larger batches: more stable gradients, better hardware utilization
Smaller batches: more gradient updates, potential regularization effect
Memory constraints limit upper bound

2.5.2 Implementation Challenges
Technical Issues Resolved:
Parameter Name Mismatch: Fixed num_heads vs n_heads inconsistency
Memory Management: Optimized for available GPU memory

Training Stability: Implemented gradient clipping and learning rate scheduling

3. System Integration and Validation
3.1 End-to-End Pipeline
The complete system integrates:
Data Preprocessing: Multi30k German-English dataset
Model Training: Hyperparameter-specific configurations
Model Evaluation: Validation loss and translation quality
Translation Inference: Greedy decoding implementation

3.2 Validation Approach
Translation Quality Assessment:

```
src_sentence = "Ein kleiner Junge spielt draußen mit einem Ball."
translated_sentence = translate_sentence(model, src_sentence, vocab_src, vocab_tgt)
# Expected: "A little boy playing outside with a ball."
```

Performance Benchmarking:
Baseline model training verification
Random data inference testing
Translation quality spot-checking

4. Conclusions
4.1 Achievements
Successful Implementation: Both greedy decoding and hyperparameter optimization frameworks function correctly
Systematic Evaluation: Structured approach to model comparison
Practical Translation System: End-to-end German-to-English translation capability

4.2 Limitations and Improvements
Current Limitations:
Greedy decoding may miss optimal translations
Limited hyperparameter exploration due to computational constraints
No advanced evaluation metrics (BLEU, METEOR)

```
==================================================
Experiment 1/15
Config: {'num_heads': 4, 'num_layers': 6, 'learning_rate': 0.0005, 'batch_size': 128}
==================================================
Epoch 1/10 - Train Loss: 5.7427, Val Loss: 6.8937
Epoch 2/10 - Train Loss: 5.2915, Val Loss: 11.8970
Epoch 3/10 - Train Loss: 5.2373, Val Loss: 10.9806
Epoch 4/10 - Train Loss: 5.1910, Val Loss: 11.9907
Epoch 5/10 - Train Loss: 5.1753, Val Loss: 11.1682
Epoch 6/10 - Train Loss: 5.1530, Val Loss: 11.6485
Epoch 7/10 - Train Loss: 5.1501, Val Loss: 11.4985
Epoch 8/10 - Train Loss: 5.1334, Val Loss: 11.4764
Epoch 9/10 - Train Loss: 5.1243, Val Loss: 11.5945
Epoch 10/10 - Train Loss: 5.1085, Val Loss: 12.2402

==================================================
Experiment 2/15
Config: {'num_heads': 4, 'num_layers': 3, 'learning_rate': 0.0005, 'batch_size': 32}
==================================================
Epoch 1/10 - Train Loss: 4.8733, Val Loss: 4.3700
Epoch 2/10 - Train Loss: 4.1209, Val Loss: 4.0380
Epoch 3/10 - Train Loss: 3.8970, Val Loss: 3.8139
Epoch 4/10 - Train Loss: 3.7588, Val Loss: 3.7099
Epoch 5/10 - Train Loss: 3.6813, Val Loss: 3.6829
Epoch 6/10 - Train Loss: 3.6150, Val Loss: 3.6415
Epoch 7/10 - Train Loss: 3.5662, Val Loss: 3.6067
Epoch 8/10 - Train Loss: 3.5192, Val Loss: 3.5755
Epoch 9/10 - Train Loss: 3.4726, Val Loss: 3.5741
Epoch 10/10 - Train Loss: 3.4242, Val Loss: 3.5208
```

```
==================================================
Experiment 3/15
Config: {'num_heads': 8, 'num_layers': 3, 'learning_rate': 0.001, 'batch_size': 128}
==================================================
Epoch 1/10 - Train Loss: 5.7630, Val Loss: 5.6792
Epoch 2/10 - Train Loss: 5.5046, Val Loss: 11.5638
Epoch 3/10 - Train Loss: 5.2794, Val Loss: 12.6781
Epoch 4/10 - Train Loss: 5.2329, Val Loss: 12.3284
Epoch 5/10 - Train Loss: 5.1877, Val Loss: 5.7439
Epoch 6/10 - Train Loss: 4.9255, Val Loss: 5.3508
Epoch 7/10 - Train Loss: 4.6947, Val Loss: 5.3394
Epoch 8/10 - Train Loss: 4.6441, Val Loss: 5.2400
Epoch 9/10 - Train Loss: 4.6306, Val Loss: 5.4277
Epoch 10/10 - Train Loss: 4.5485, Val Loss: 4.9637

==================================================
Experiment 4/15
Config: {'num_heads': 8, 'num_layers': 3, 'learning_rate': 0.0005, 'batch_size': 64}
==================================================
Epoch 1/10 - Train Loss: 5.0353, Val Loss: 4.6216
Epoch 2/10 - Train Loss: 4.3090, Val Loss: 4.1364
Epoch 3/10 - Train Loss: 3.9351, Val Loss: 3.8649
Epoch 4/10 - Train Loss: 3.7157, Val Loss: 3.7105
Epoch 5/10 - Train Loss: 3.5845, Val Loss: 3.6769
Epoch 6/10 - Train Loss: 3.4812, Val Loss: 3.5582
Epoch 7/10 - Train Loss: 3.4006, Val Loss: 3.4847
Epoch 8/10 - Train Loss: 3.3247, Val Loss: 3.4388
Epoch 9/10 - Train Loss: 3.2608, Val Loss: 3.4493
Epoch 10/10 - Train Loss: 3.2101, Val Loss: 3.3976
```

Overfitting Indicators:
Training loss decreases, validation loss increases/stagnates
Experiment 1

Unstable Training:
Wild validation loss fluctuations
Experiment 3 (epochs 1-4)

Healthy Training:
Both losses decrease together
Validation loss doesn't diverge from training
Experiments 2 and 4

Optimal Configuration (so far):
```
optimal_config = {
    'num_heads': 4,          # Sufficient complexity
    'num_layers': 3,         # Prevents overfitting
    'learning_rate': 0.0005, # Stable convergence
    'batch_size': 32         # Best generalization
}
```