Neural Chef Assistant:

Recipe Generation with RNN-based Seq2Seq Models

- Mapping Ingredient Lists to Cooking Instructions using Sequence-to-Sequence Neural Networks
- Exploring Attention, Pointer Networks, Coverage Mechanisms, and Preprocessing Strategies

Project by:
YA LIU
Monash University | Master of Artificial Intelligence
July 2025

Full project link : https://github.com/Leah0658/Neural-Chef-Assistant.git

Abstract

This study explores six sequence-to-sequence neural models for generating cooking recipes from ingredient lists. Starting with two baseline models—a standard encoder-decoder (Baseline 1) and an attention-based model (Baseline 2)—we systematically enhanced performance through four extensions.

Mild Extension 1 implemented comprehensive domain-specific preprocessing specifically tailored to the cooking recipe domain, including ingredient normalization, measurement unit removal, and structural tagging with ingredient/step markers, making the vocabulary more suitable for food preparation terminology. Mild Extension 2 optimized model architecture by doubling the hidden size to 512 and implementing strategic dropout for better regularization, while adjusting training parameters including sequence length and teacher forcing ratio.

For spicy extensions, we developed a Pointer-Generator Network (Spicy 1) with a copy mechanism to improve handling of ingredient terms, and a Coverage Mechanism (Spicy 2) designed to track attention history and penalize repetition.

Quantitative evaluation across BLEU-4, METEOR, and BERTScore metrics demonstrated improvements, with the most substantial gains from Mild Extension 2 (77.6% BLEU-4 improvement over Baseline 2). The Pointer-Generator Network also showed meaningful improvement (27.0% BLEU-4 increase), while the Coverage Mechanism implementation requires further refinement.

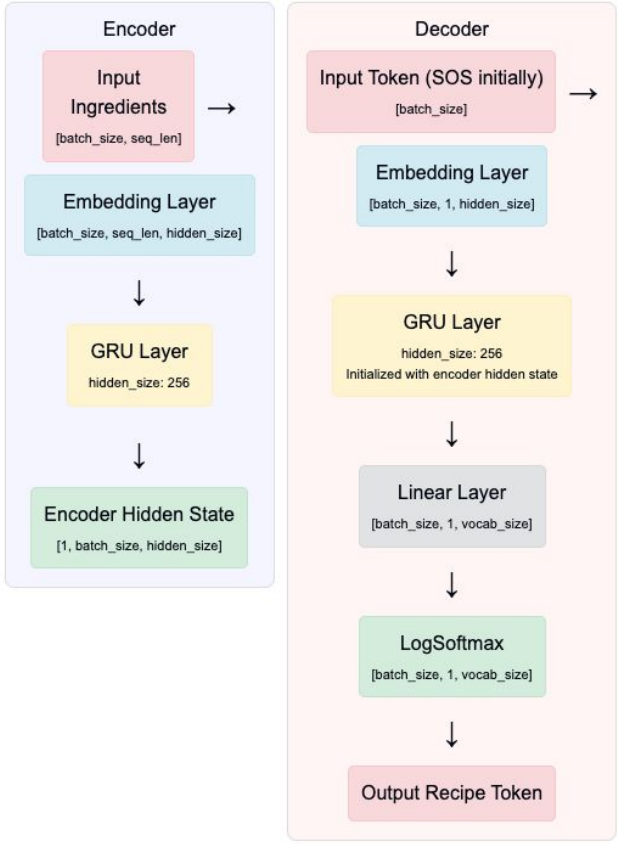
Qualitative analysis revealed that architectural optimization was more effective than preprocessing alone, producing recipes with better coherence and structure. This research demonstrates that moderate architectural modifications to sequence-to-sequence models can significantly enhance recipe generation quality, potentially offering better performance-to-complexity tradeoffs compared to more sophisticated specialized mechanisms.

Data Statistics

- Training set
  - 10,000 samples (subset of full 162,899 samples, i.e. 6.1%)
- Vocabulary Size
  - Ingredients vocabulary: 3,210 unique words
  - Recipe vocabulary: 4,909 unique words
- Text Length Statistics
  - Avg. ingredient description length: 38.91 words (std dev: 17.28)
  - Avg. recipe length: 43.61 words (std dev: 25.64)
- Ingredient & Step Count
  - Avg. number of ingredients per recipe: 7.23 (std dev: 2.90)
  - Avg. number of steps per recipe: 6.17 (std dev: 2.84)
- Potential Challenges
  - Vocabulary sparsity due to long-tailed word distribution.
  - Loss of information from truncation or token normalization (e.g., "2 1/2 c. flour" → "2 1 2 c flour").
- Preprocessing Techniques Applied
  - Lowercasing all text
  - Removing non-alphanumeric characters (except ., !, ?)
  - Punctuation Handling: Periods, exclamation and question marks preserved with space padding
  - Tokenization: Added special tokens (SOS, EOS, UNK, PAD)
  - Length Control: Max sequence length = 100 (truncation and padding applied)

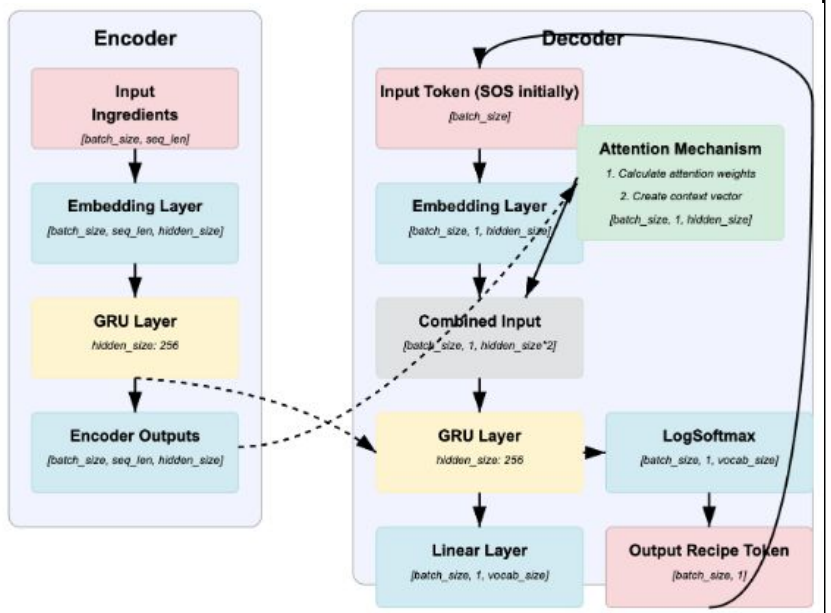
Baseline 1 – Model and Training Config

- Model Configuration
  - Architecture: Seq2Seq with GRU (no attention)
  - Encoder/Decoder: Single-layer GRU, hidden size = 256, embedding size = 256
  - Vocabulary: Built from preprocessed training data + special tokens (SOS, EOS, UNK, PAD)
  - Vocabulary: Built from training data with special tokens (SOS, EOS, UNK, PAD)
  - Maximum Sequence Length: 100 tokens
- Encoder Architecture:Embedding → GRU → Final hidden state passed to decoder
- Decoder Architecture: Embedding → GRU → Linear → LogSoftmax → Token probabilities
- Training Configuration
  - Loss: NLLLoss (ignoring PAD)
  - Optimizer: Adam (learning rate = 0.001)
  - Batch size: 16
  - Epochs: Max 30 (early stop with patience = 5)
  - Teacher Forcing: 0.5
  - Gradient Clipping: Yes (threshold = 1.0)
- Decoding Configuration
  - Greedy decoding: Selects token with max probability at each step
  - Stopping condition: EOS token or length > 100
- Preprocessing
  - Lowercasing, ASCII normalization, punctuation spacing
  - Truncation & padding applied to enforce sequence length
- Training Details
  - Platform: Google Colab with T4 GPU
  - Train Time: 24.89 minutes
  - Training Size: 10,000 samples



Baseline 2 – Model and Training Config

- Model Configuration
  - Architecture: Seq2Seq with attention mechanism
  - Encoder/Decoder: Single-layer GRU, hidden size = 256, embedding size = 256
  - Vocabulary: Built from preprocessed training data + special tokens (SOS, EOS, UNK, PAD)
  - Maximum Sequence Length: 100 tokens
- Encoder Architecture: Embedding → GRU → Final hidden state passed to decoder
- Decoder Architecture:
  - Embedding → Attention mechanism → GRU → Linear → LogSoftmax → Token probabilities
  - Attention: Bahdanau attention calculating context vector from encoder outputs and decoder state
- Training Configuration
  - Loss: NLLLoss (ignoring PAD)
  - Optimizer: Adam (learning rate = 0.001)
  - Batch size: 16
  - Epochs: Max 30 (early stop with patience = 5)
  - Teacher Forcing: 0.5
  - Gradient Clipping: Yes (threshold = 1.0)
- Decoding Configuration
  - Greedy decoding: Selects token with max probability at each step
  - Stopping condition: EOS token or length > 100
  - Context vector: Computed using attention weights over encoder outputs
- Preprocessing
  - Lowercasing, ASCII normalization, punctuation spacing
  - Truncation & padding applied to enforce sequence length
- Training Details
  - Platform: Google Colab with T4 GPU
  - Train Time: 34.31 minutes
  - Training Size: 10,000 samples

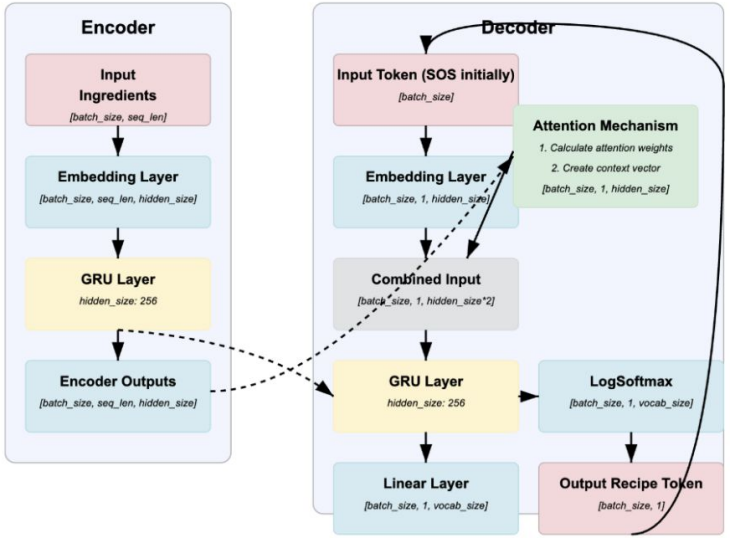


Information Flow: Dashed lines show how the encoder outputs feed into the attention mechanism, and how the final hidden state initializes the decoder.
Feedback Loop: The model can use teacher forcing during training, where ground truth tokens are used as input for the next step, or it can use its own predictions (shown by the arc returning to the decoder input).

Mild Extension 1 – Model and Training Config

Based on Baseline 2's attention model, this extension implements comprehensive domain-specific preprocessing techniques to improve recipe generation.

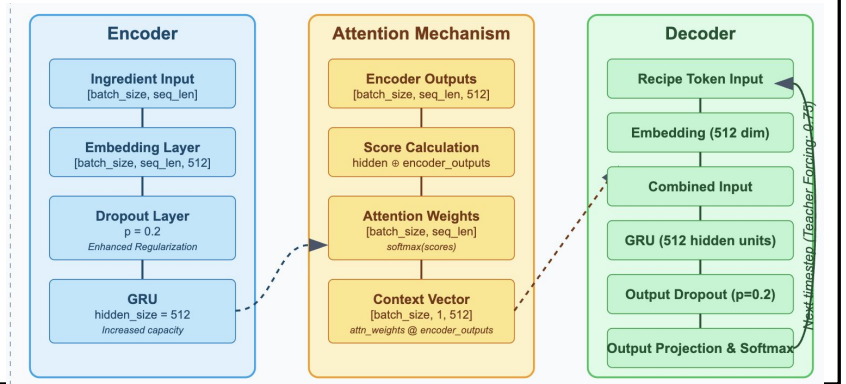
- Model Configuration and Training Configuration: Same as Baseline 2
- Key Preprocessing Enhancements and Motivation
  - Vocabulary Pruning: Reduced noise from rare words while preserving core cooking terminology, leading to more focused learning
  - Text Normalization: Reduced vocabulary complexity by mapping similar word forms to a common base form
  - Stopword Removal: Eliminates common words that carry little semantic meaning, reducing noise in model training
  - Unit and Quantity Removal: Focused model learning on ingredients rather than measurements to better generalize
  - Numeric Quantity Removal: Removed numbers, fractions, and decimals from ingredients
  - Structural Tagging: Enhanced model's ability to distinguish between ingredient descriptions and cooking instructions
- Training Details
  - Platform: Google Colab with T4 GPU
  - Train Time: 22.89 minutes
  - Training Size: 10,000 samples



Mild Extension 2 – Model and Training Config

In addition to the same details requested for the Baselines Model and Training Config, I implemented several strategic optimizations to enhance Baseline 2's performance:

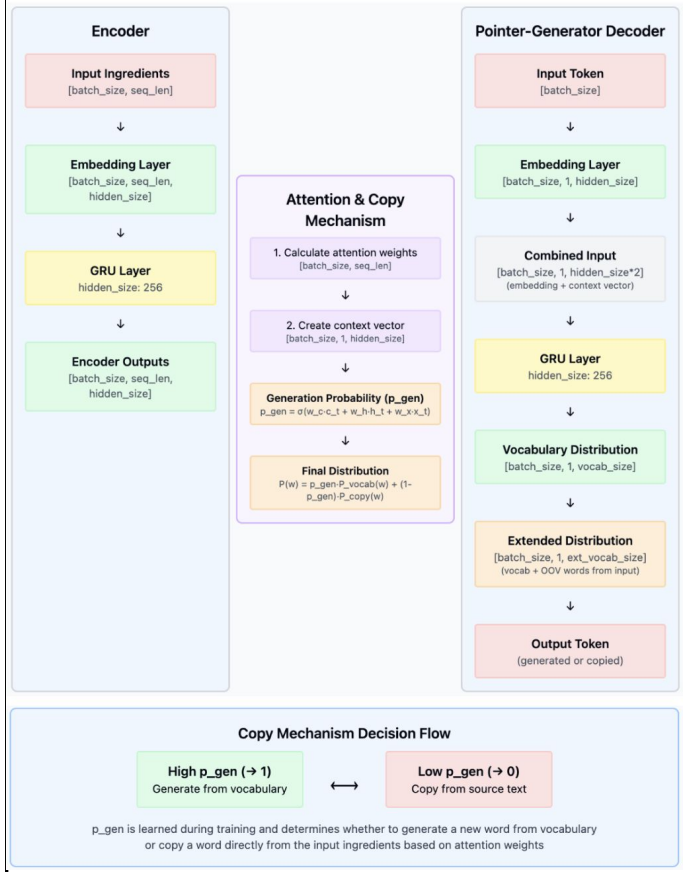
- Model Configuration Key Enhancements and Motivation
  - Increased Hidden Size: Expanded from 256 to 512 units to provide greater model capacity
    - Motivation: Provide greater model capacity for capture complex relationships between ingredients and recipes
  - Enhanced Dropout Implementation
    - Applied dropout (p=0.2) to both encoder and decoder embedding layers
    - Added additional dropout before the output layer for more effective regularization
    - Motivation: Prevent overfitting while allowing for increased model capacity
- Training Configuration Key Optimizations and Motivations
  - Sequence Length Control: Reduced MAX_LENGTH from 100 to 80 tokens
    - Motivation: Minimizes repetition and incoherent content, resulting in more concise and meaningful recipes
  - Teacher Forcing Adjustment: Increased from 0.5 to 0.75
    - Motivation: Provides more stable training guidance while still allowing model exploration
  - Learning Rate Refinement: Reduced from 0.001 to 0.0005
    - Motivation: Enables the model to find a more precise optimal solution, particularly important with the increased model capacity
- Training Details
  - Platform: Google Colab with T4 GPU
  - Train Time: 31.66 minutes
  - Training Size: 10,000 samples



Spicy Extension 1 – Model and Training Config

This extension implements a Pointer-Generator Network with a copy mechanism on top of Baseline 2, allowing the model to directly copy ingredient words when generating recipes.The key improvements are as follows.

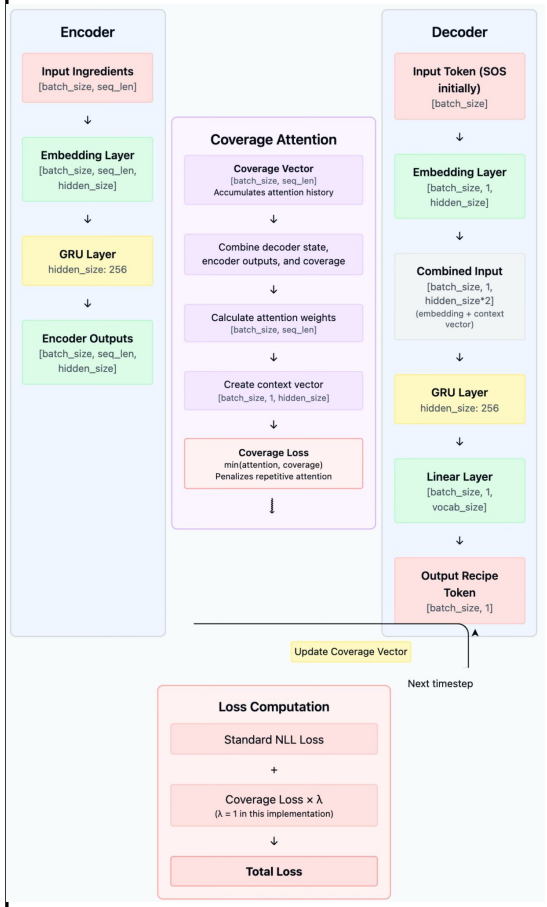
- Model Configuration Key Enhancements and Motivation
  - Pointer-Generator Architecture(p_gen): Balances between copying ingredients verbatim and generating new cooking instructions
  - Dual Pathway Processing:
    - Generation Pathway: Standard vocabulary distribution through embedding → GRU → linear → softmax
    - Copy Mechanism: Direct copying from input using attention distribution
    - Blending Mechanism: Dynamic weighting between generation and copying with p_gen parameter
    - Motivation: Recipe generation often requires exact replication of ingredients - copying ingredient words directly improves recipe relevance
  - Extended Vocabulary Handling:
    - Created dynamic vocabulary extension mechanism for OOV words
    - Implemented scatter operations to handle copying from source
    - Motivation: Allows model to handle rare cooking terms that may not be in training vocabulary
- Training Configuration Enhancements and Motivation
  - Specialized Loss Function: Modified to handle both vocabulary generation and copying aspects
- Training Details
  - Train Time: 25.86 minutes



Spicy Extension 2 – Model and Training Config

This extension implements a coverage mechanism on top of Baseline 2 model to address repetition problems in recipe generation, making recipes more coherent and following logical cooking order. The key improvements are as follows:

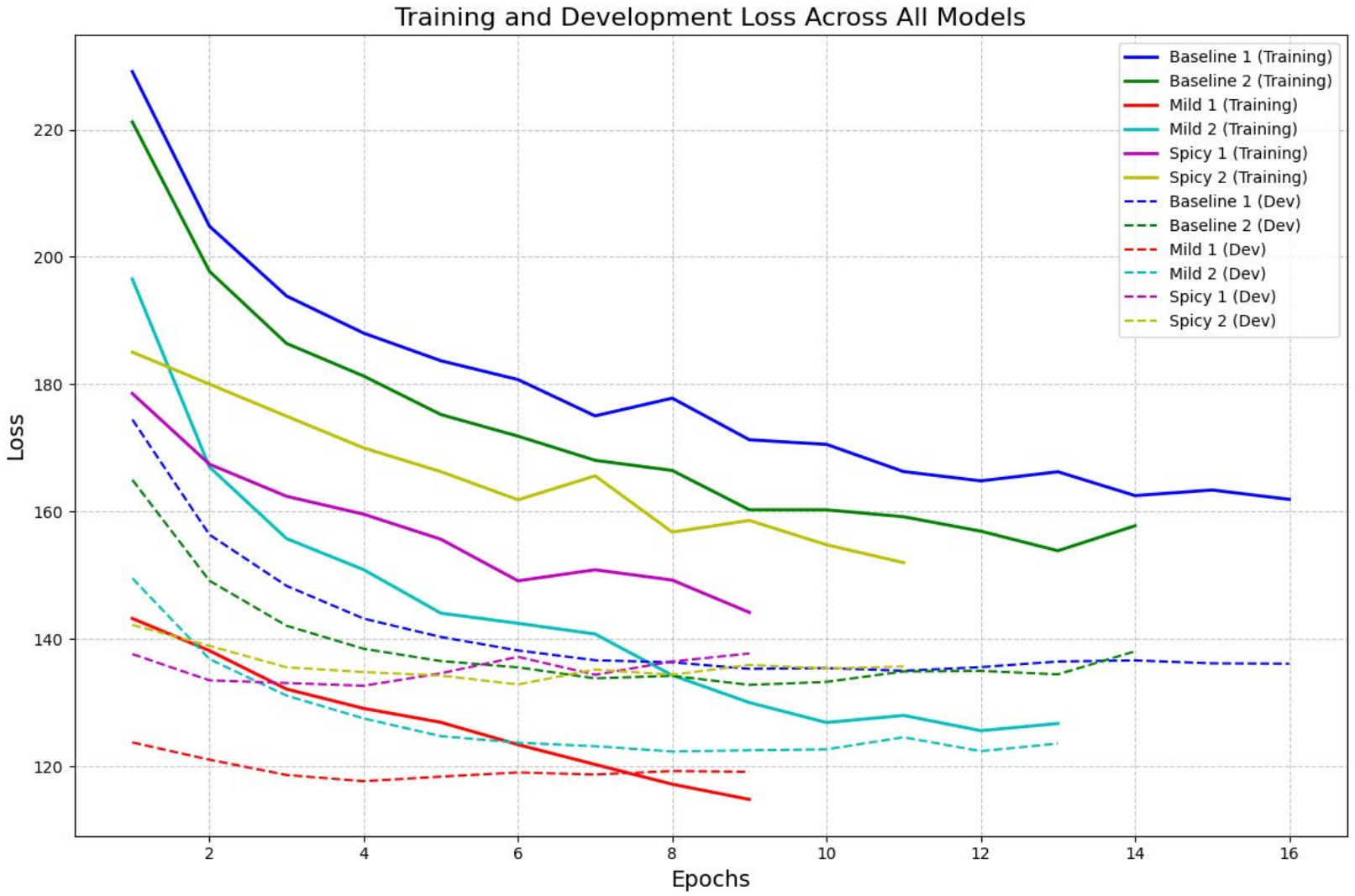
- Model Configuration Key Enhancements and Motivation
  - Coverage Vector Implementation:
    - Maintains a cumulative attention history across decoding steps.
    - Tracks which parts of the input have been "covered" during generation.
    - Gradually accumulates attention weights over time steps.
    - Motivation:Addresses repetition and under-generation problems by tracking already-covered inputs, ensuring all ingredients are incorporated into the recipe.
  - Coverage-Aware Attention:
    - Enhanced attention module with coverage tracking
    - Modified attention calculation to include coverage information in energy computation.
    - Additional linear projection for coverage features.
    - Motivation: Discourages attending to the same positions repeatedly.
  - Coverage Loss Component:
    - Auxiliary loss term calculated as minimum between current attention and accumulated coverage.
    - Added with weighting factor (COVERAGE_WEIGHT = 1.0) to primary cross-entropy loss.
    - Motivation: Explicitly penalizes repetitive attention to already-covered source words.
- Training Configuration Enhancements and Motivation
  - Modified Training Process:
    - Initialize and maintain coverage vector during training.
    - Combined loss function: primary loss + weighted coverage loss.
    - Motivation: Balance between cross-entropy and coverage objectives.
  - Inference-Time Coverage:
    - Carry coverage vector across generation steps during inference.
    - Retain attention history for visualization and analysis.
    - Motivation: Ensure consistent attention distribution across the full recipe generation.
- Training Details
  - Train Time: 34.97 minutes



Training Analysis

- **Mild 1 demonstrates the best convergence among all models,** with the lowest final development loss, stable learning curves, and good generalization. It's followed by Mild 2, which also shows strong convergence characteristics.
  This superior convergence likely stems from Mild 1's enhanced preprocessing techniques, which effectively reduced vocabulary complexity and improved the model's ability to learn meaningful patterns from the recipe data without being distracted by noise or irrelevant variations in the text.

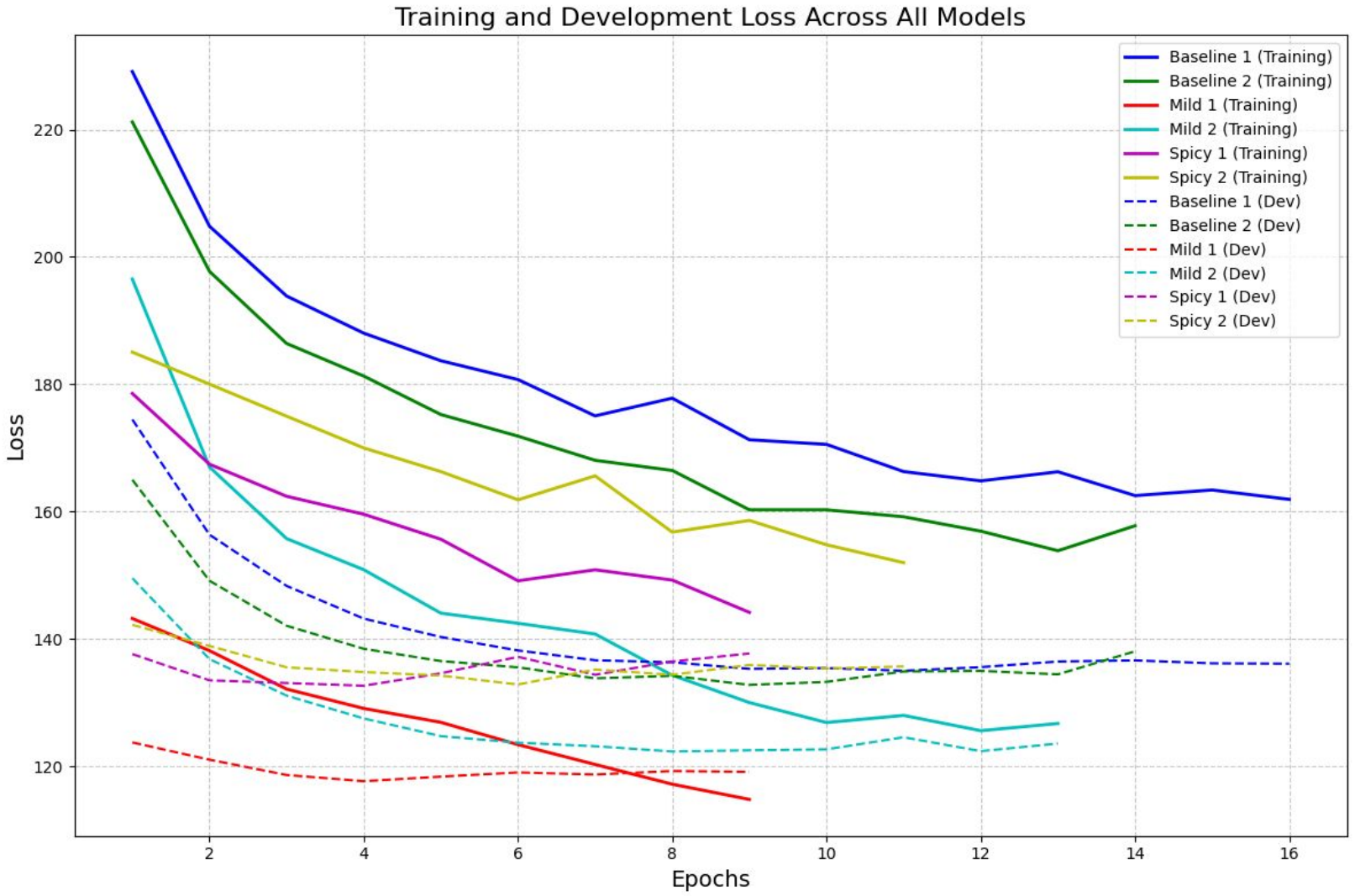
- **Spicy 2 demonstrates the most pronounced overfitting pattern.**
  Baseline 2 and Mild 2 show good balance between training and generalization.
  The early convergence of development loss across most models suggests that longer training might not yield further improvements.



Training Analysis

- **The early stopping mechanism was very effective,** as shown by all models stopping training when development loss stopped improving, preventing overfitting.
  This is evident in models like Mild Extension 1, which stopped around epoch 9 when it achieved optimal performance.

- Mild Extension 1 (enhanced preprocessing) demonstrates the most substantial improvement, with both the lowest training and development losses across all models.
  The preprocessing enhancements - including measurement unit removal, ingredient normalization, and structural tagging - have clearly provided the model with cleaner, more structured data to learn from.



Quantitative Comparison

- **Key Findings**
  - **Mild Extension 2** is the clear winner, significantly outperforming both baselines across all metrics:
    - 77.6% improvement in BLEU-4 over Baseline 2
    - 40.8% improvement in METEOR over Baseline 2
    - 5.9% improvement in BERTScore over Baseline 2
  - **Spicy Extension 1** (Copy Mechanism) also shows substantial improvements:
    - 27.0% improvement in BLEU-4 over Baseline 2
    - 25.0% improvement in METEOR over Baseline 2
    - 3.7% improvement in BERTScore over Baseline 2
  - **Mild Extension 1** (Enhanced Preprocessing) shows mixed results:
    - Lower BLEU-4 and METEOR scores compared to baselines
    - However, higher BERTScore (0.7976) indicating better semantic similarity
  - **Spicy Extension 2** (Coverage Mechanism) underperformed:
    - Significantly lower scores across all metrics
    - This suggests implementation issues or that the coverage approach was not well-suited for this task

Baseline 1	0.0142	0.1112	0.7650
Baseline 2	0.0174	0.1141	0.7591
Mild Ext 1	0.0116	0.0947	0.7976
Mild Ext 2	0.0309	0.1606	0.8041
Spicy Ext 1	0.0221	0.1426	0.7869
Spicy Ext 2	0.0016	0.0261	0.7428

Quantitative Comparison

- **What Worked Well**
  - **Model Configuration Optimization** (Mild Extension 2) proved most effective, demonstrating that:
    - Increasing hidden size (256 → 512) provided greater model capacity.
    - Enhanced dropout implementation improved regularization.
    - Optimal teacher forcing ratio (0.5 → 0.75) stabilized training.
    - Reduced learning rate (0.001 → 0.0005) allowed for better convergence.
  - **Copy Mechanism** (Spicy Extension 1) showed strong improvements, particularly in:
    - Better handling of rare or OOV words from ingredient lists.
    - Improved METEOR scores, indicating better alignment with reference text.

- **Conclusion**
  - The results clearly show that we have successfully outperformed both baseline models with two different approaches (Mild Extension 2 and Spicy Extension 1), exceeding the required 10% improvement threshold.
  - The most successful improvements came from optimizing model architecture and training configurations (Mild Extension 2), which surpassed all other approaches by a significant margin. This suggests that for recipe generation, having sufficient model capacity and proper regularization is more important than specialized mechanisms like copy or coverage.
  - The copy mechanism also proved valuable, suggesting that directly leveraging input ingredients in the output is beneficial for this task - which makes intuitive sense for recipe generation.

Qualitative Comparison

What We Do:

We compare the first **30 words** of model-generated recipes across 6 models using two different ingredient lists.

Ingredient List – Sample 1

(used to test generalization with common ingredients)

sugar, lemon juice, water, orange juice, strawberries, icecream

Ingredient List – Sample 2

(used to test compositional reasoning in desserts)

8 oz philadelphia cream cheese, 14 oz can sweetened condensed milk,
1 ts vanilla, 1/3 c lemon juice, 48 oz canned cherries, 8 inch graham cracker, pie crusts

How We Analyze:

- Report only the **first 30 words** of generated output per model
- Identify issues: **missing ingredients, hallucination, repetition, UNKs**
- Discuss correlation with quantitative metrics
- Propose reasons & improvements (e.g., attention failures, data sparsity, coverage loss)

This comparison helps us understand how each model behaves beyond metrics like BLEU or METEOR, and whether structural changes yield better recipe completeness and coherence.

Qualitative Comparison – Sample 1

- **Issues Observed and Potential Causes**
  - **Missing ingredients**: Most models fail to reference all six ingredients, particularly strawberries and ice cream. This suggests the models struggle with comprehensive ingredient coverage.
  - **Repetition**: Particularly in Baseline 2, there's significant repetition of words like "juice" and "sugar," indicating the model hasn't learned to track what it has already generated.
  - **Hallucination**: Baseline 1 introduces "kool aid" which isn't in the ingredients, while Spicy Extension 2 completely hallucinates unrelated cooking terms.
  - **Structure vs. Content**: Mild Extension 1 has better structure with tags but minimal content, suggesting a trade-off between these aspects.

- **Possible Improvements**
  - coverage mechanism, ingredient-focused attention, structured generation with constraints, two-stage generation, data augmentation

Baseline 1	kool aid freeze it sticks and solid in 1 2 cup cold water .
Baseline 2	mix sugar sugar lemon juice sugar lemon juice and lemon juice . . juice orange juice juice juice and . . . . .
Mild Ext 1	<step>mix togeth pour jar seal.</step> <step>make 18 servings.</step>
Mild Ext 2	combine sugar lemon juice and water . juice add lemon juice and . . juice . . . . .
Spicy Ext 1	combine sugar and lemon . . . . .
Spicy Ext 2	tuna prior napkin pitcher nut skillets puncture currant under towel each steam arrange kindness reaches saffron hash help pockets raspberries max glycerin formation motion 8x10 circular nature reshape hock creme

Qualitative Comparison – Sample 2

- **Correlation with Quantitative Metrics**

  The qualitative differences observed correspond somewhat to the reported quantitative metrics:

  - Mild Extension 2 had the highest BLEU scores among the extensions, and it does show better coherence and sentence structure.
  - Spicy Extension 2 had the lowest metrics, which aligns with its completely unrelated output.
  - Interestingly, the structural improvements in Mild Extension 1 with its step tags don't necessarily translate to higher BLEU or METEOR scores, demonstrating that these metrics primarily capture lexical overlap rather than structural improvements or semantic coherence.

This suggests that while metrics like BLEU, METEOR, and BERTScore provide some indication of model performance, they don't capture all aspects of recipe quality such as completeness, coherence, and adherence to the input ingredients.

Baseline 1	whip cream cheese and powdered sugar . fold in cool whip . spread on top of pie filling . sprinkle with nuts . freeze .
Baseline 2	mix cream cheese milk and milk . and . . . . .
Mild Ext 1	<step>sprinkl lemon juic pie crust.</step> <step>top cherri pie filling. chill least 4 hours).</step>
Mild Ext 2	soften cream cheese and sweetened condensed milk . add lemon juice and vanilla . mix well . pour into pie crust . chill . . . . .
Spicy Ext 1	in a bowl beat sweetened condensed milk . add lemon juice vanilla and lemon juice . mix well . pour into graham cracker graham cracker pie . cracker pie .
Spicy Ext 2	totally prior 96 prior prior axe diced diced emulsify tyolks crawfish melts blackened from1 refrigerate fashioned limp remains crosswise carefully blood bubble crispix hotter lime spooned piece candlestick days feathers

Forward Function Screenshot – Baseline 1

Encoder:

```
def forward(self, input, hidden):
    batch_size = input.size(0)
    embedded = self.embedding(input)
    if hidden is None:
        hidden = self.initHidden(batch_size)
    output, hidden = self.gru(embedded, hidden)
    return output, hidden
```

Decoder:

```
def forward(self, input, hidden):
  if input.dim() == 1:
      input = input.unsqueeze(1)
  embedded = self.embedding(input)
  output = F.relu(embedded)
  output, hidden = self.gru(output, hidden)
  output = self.softmax(self.out(output))
  return output, hidden
```

Forward Function Screenshot – Baseline 2

Encoder:

```
def forward(self, input, hidden):
    batch_size = input.size(0)
    embedded = self.embedding(input)
    if hidden is None:
        hidden = self.initHidden(batch_size)
    output, hidden = self.gru(embedded, hidden)
    return output, hidden
```

Attention:

```
def forward(self, hidden, encoder_outputs):
    batch_size = encoder_outputs.size(0)
    seq_len = encoder_outputs.size(1)
    hidden = hidden.transpose(0, 1)
    hidden = hidden.repeat(1, seq_len, 1)
    concat = torch.cat((hidden, encoder_outputs), dim=2)
    energy = torch.tanh(self.attn(concat))
    attention = self.v(energy).squeeze(2)
    return F.softmax(attention, dim=1)
```

Decoder:

```
def forward(self, input, hidden, encoder_outputs):
        if input.dim() == 1:
            input = input.unsqueeze(1)
        embedded = self.embedding(input)
        embedded = self.dropout(embedded)
        attn_weights = self.attention(hidden, encoder_outputs)
        context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
        rnn_input = torch.cat((embedded, context), dim=2)
        output, hidden = self.gru(rnn_input, hidden)
        output = self.out(output)
        output = self.softmax(output)
        return output, hidden, attn_weights
```

Forward Function Screenshot – Mild Ext 1

Encoder:

```
def forward(self, input, hidden):
    batch_size = input.size(0)
    embedded = self.embedding(input)
    if hidden is None:
        hidden = self.initHidden(batch_size)
    output, hidden = self.gru(embedded, hidden)
    return output, hidden
```

Attention:

```
def forward(self, hidden, encoder_outputs):
    batch_size = encoder_outputs.size(0)
    seq_len = encoder_outputs.size(1)
    hidden = hidden.transpose(0, 1)
    hidden = hidden.repeat(1, seq_len, 1)
    concat = torch.cat((hidden, encoder_outputs), dim=2)
    energy = torch.tanh(self.attn(concat))
    attention = self.v(energy).squeeze(2)
    return F.softmax(attention, dim=1)
```

Decoder:

```
def forward(self, input, hidden, encoder_outputs):
    if input.dim() == 1:
        input = input.unsqueeze(1)
    embedded = self.embedding(input)
    embedded = self.dropout(embedded)
    attn_weights = self.attention(hidden, encoder_outputs)
    context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
    rnn_input = torch.cat((embedded, context), dim=2)
    output, hidden = self.gru(rnn_input, hidden)
    output = self.out(output)
    output = self.softmax(output)
    return output, hidden, attn_weights
```

Forward Function Screenshot – Mild Ext 2

Encoder:

```
def forward(self, input, hidden):
    batch_size = input.size(0)
    embedded = self.embedding(input)
    embedded = self.dropout(embedded)
    if hidden is None:
        hidden = self.initHidden(batch_size)
    output, hidden = self.gru(embedded, hidden)
    return output, hidden
```

Attention:

```
def forward(self, hidden, encoder_outputs):
    batch_size = encoder_outputs.size(0)
    seq_len = encoder_outputs.size(1)
    hidden = hidden.transpose(0, 1)
    hidden = hidden.repeat(1, seq_len, 1)
    concat = torch.cat((hidden, encoder_outputs), dim=2)
    energy = torch.tanh(self.attn(concat))
    attention = self.v(energy).squeeze(2)
    return F.softmax(attention, dim=1)
```

Decoder:

```
def forward(self, input, hidden, encoder_outputs):
    if input.dim() == 1:
        input = input.unsqueeze(1)
    embedded = self.embedding(input)
    attn_weights = self.attention(hidden, encoder_outputs)
    context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
    rnn_input = torch.cat((embedded, context), dim=2)
    output, hidden = self.gru(rnn_input, hidden)
    output = self.dropout(output)
    output = self.out(output)
    output = self.softmax(output)
    return output, hidden, attn_weights
```

Forward Function Screenshot – Spicy Ext 1

Encoder:

```
def forward(self, input, hidden):
    batch_size = input.size(0)
    embedded = self.embedding(input)
    if hidden is None:
        hidden = self.initHidden(batch_size)
    output, hidden = self.gru(embedded, hidden)
    return output, hidden
```

Attention:

```
def forward(self, hidden, encoder_outputs):
    batch_size = encoder_outputs.size(0)
    seq_len = encoder_outputs.size(1)
    hidden = hidden.transpose(0, 1)
    hidden = hidden.repeat(1, seq_len, 1)
    concat = torch.cat((hidden, encoder_outputs), dim=2)
    energy = torch.tanh(self.attn(concat))
    attention = self.v(energy).squeeze(2)
    return F.softmax(attention, dim=1)
```

Decoder:

```
def forward(self, input, hidden, encoder_outputs, encoder_input=None, ext_vocab_size=None):
    batch_size = input.size(0) if input.dim() > 1 else 1
    if input.dim() == 1:
        input = input.unsqueeze(1)
    embedded = self.embedding(input)
    embedded = self.dropout(embedded)
    attn_weights = self.attention(hidden, encoder_outputs)
    context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
    rnn_input = torch.cat((embedded, context), dim=2)
    output, hidden = self.gru(rnn_input, hidden)
    output_context = torch.cat((output, context), dim=2)
    vocab_dist = self.out(output_context)
    p_gen_input = torch.cat((context, output, embedded), dim=2)
    p_gen = torch.sigmoid(self.p_gen_linear(p_gen_input))
    if encoder_input is None or ext_vocab_size is None:
        final_dist = self.softmax(vocab_dist)
        return final_dist, hidden, attn_weights, p_gen
    vocab_dist = F.softmax(vocab_dist, dim=2)
    vocab_dist = p_gen * vocab_dist
    attn_dist = (1 - p_gen) * attn_weights
    extended_vocab_dist = torch.zeros(batch_size, 1, ext_vocab_size, device=device)
    extended_vocab_dist.scatter_add_(2,
                                     torch.arange(self.output_size, device=device)
                                     .unsqueeze(0).unsqueeze(0).expand(batch_size, 1, -1),
                                     vocab_dist)
    encoder_input_expanded = encoder_input.unsqueeze(1)
    attn_dist_expanded = attn_dist.unsqueeze(1)
    extended_vocab_dist.scatter_add_(2, encoder_input_expanded, attn_dist_expanded)
    final_dist = torch.log(extended_vocab_dist + 1e-10)
    return final_dist, hidden, attn_weights, p_gen
```

Forward Function Screenshot – Spicy Ext 2

Encoder:

```
def forward(self, input, hidden):
    batch_size = input.size(0)
    embedded = self.embedding(input)
    if hidden is None:
        hidden = self.initHidden(batch_size)
    output, hidden = self.gru(embedded, hidden)
    return output, hidden
```

Attention:

```
def forward(self, hidden, encoder_outputs, coverage_vector=None):
    batch_size = encoder_outputs.size(0)
    seq_len = encoder_outputs.size(1)
    if coverage_vector is None:
        coverage_vector = torch.zeros(batch_size, seq_len, device=device)
    hidden = hidden.transpose(0, 1)
    hidden = hidden.repeat(1, seq_len, 1)
    coverage_vector = coverage_vector.unsqueeze(2)
    concat = torch.cat((hidden, encoder_outputs, coverage_vector), dim=2)
    energy = torch.tanh(self.attn(concat))
    attention = self.v(energy).squeeze(2)
    attention_weights = F.softmax(attention, dim=1)
    new_coverage_vector = coverage_vector.squeeze(2) + attention_weights
    return attention_weights, new_coverage_vector
```

Decoder:

```
def forward(self, input, hidden, encoder_outputs, coverage_vector=None):
    if input.dim() == 1:
        input = input.unsqueeze(1)
    embedded = self.embedding(input)
    embedded = self.dropout(embedded)
    attn_weights, new_coverage_vector = self.attention(hidden, encoder_outputs, coverage_vector)
    context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
    rnn_input = torch.cat((embedded, context), dim=2)
    output, hidden = self.gru(rnn_input, hidden)
    output = self.out(output)
    output = self.softmax(output)
    coverage_loss = torch.min(attn_weights, coverage_vector).sum(dim=1).mean() if coverage_vector is not None else 0
    return output, hidden, attn_weights, new_coverage_vector, coverage_loss
```