Comparing Neural Approaches for Tiny Story Generation

DATS 6312: Natural Language Processing

Individual Report

Abhilasha Singh

Instructor: Prof. Amir Jafari

Table of Contents:

1. Introduction

The automatic generation of coherent and engaging short stories remains a central challenge within the field of Natural Language Processing (NLP). Story generation requires more than language fluency; effective models must understand narrative structure, maintain thematic and character consistency, incorporate given constraints, and generate contextually appropriate continuations. Unlike tasks such as summarization or translation, narrative generation is inherently open-ended and demands creativity, long-range dependency modeling, and the ability to integrate user-provided cues such as keywords or story beginnings.

This project investigates a range of neural approaches for children's story generation, leveraging the Tiny Stories dataset from Hugging Face, a collection of synthetically generated, child-friendly narratives created using GPT-3.5 and GPT-4. The dataset is characterized by its simple vocabulary, concise narrative style, and clear story progression, making it an ideal testbed for studying how different model architectures capture narrative flow, sentence-level coherence, and adherence to constraints such as required keywords. Given an initial story prompt and a set of target vocabulary words, the goal is to produce a complete story continuation that is coherent, grammatically sound, age-appropriate, and semantically aligned with the prompt.

2. Description of My Individual Work

Throughout the project, I contributed to multiple core components spanning proposal development, model implementation, and evaluation. My work supported both the design and execution phases of the study, ensuring that our system could be trained, tested, and compared across several neural architectures. The key areas of my contribution are summarized below.

2.1 Contribution to Group Proposal

I actively participated in writing the group proposal, helping define the problem statement, scope, and study objectives. My input included describing the motivation for automated story generation, outlining the significance of comparing multiple model types, and structuring the overall research plan. I also contributed to identifying relevant datasets, proposing evaluation strategies, and defining milestones for model development.

2.2 Development of Evaluation Metrics

I designed and implemented the evaluation pipeline used to assess the performance of all story-generation models. This involved studying each evaluation metric in depth and preparing the complete evaluation script. The project applies to a combination of lexical, semantic, structural, and constraint-based metrics to provide a comprehensive assessment of generated stories. The metrics integrated into the pipeline include:

- **BLEU**, which measures n-gram overlaps between generated and reference stories
- **ROUGE**, which evaluates recall-oriented text similarity
- **BERTScore**, which captures semantic similarity using contextual embeddings
- **Perplexity**, which quantifies model fluency and predictive confidence
- **Keyword Coverage Metrics**, including strict match and percentage coverage, to assess adherence to required keywords

```
summary: Dict[str, float] = {
    "corpus_bleu": corpus_bleu_score,
    "avg_perplexity": mean_ppl,
    "bert_precision": P_mean,
    "bert_recall": R_mean,
    "bert_f1": F1_mean,
    "rouge1": rouge1_mean,
    "rouge2": rouge2_mean,
    "rougeL": rougeL_mean,
    "keyword_strict_accuracy": keyword_strict_acc,
    "keyword_avg_percentage": keyword_avg_pct,
}

return df, summary
```

By implementing and validating these metrics, we ensured that all models in the study could be compared consistently and fairly using standardized evaluation criteria.

2.3 Building the Baseline Model: GPT-2 Fine-Tuning

First, I constructed structured training prompts that combine natural-language instructions, the list of required keywords, and the beginning of each story. These components are concatenated into a single conditioning prefix (e.g., "You are a children's story writer… Use these words: … Story prompt: … Write a complete story:"), followed by the gold story, so that GPT-2 can learn to continue the story given the prompt and constraints.

```
def format_example(row):  2 usages
    if isinstance(row["words"], str) and row["words"].startswith("["):
        words_list = ast.literal_eval(row["words"])
    else:
        words_list = row["words"]

    # Fall back to empty list if something went wrong
    if not isinstance(words_list, list):
        words_list = []

    keywords_str = ", ".join(str(w) for w in words_list)
    prompt_str = str(row["story_beginning_prompt"]).strip()
    story_str = str(row["story"]).strip()

    input_text = (
        "You are a children's story writer.\n"
        f"Use these words: {keywords_str}\n"
        f"Story prompt: {prompt_str}\n"
        "Write a complete story:\n"
    )
    return input_text + story_str


print("Formatting text for TRAIN split...")
train_df["text"] = train_df.apply(format_example, axis=1)
```

All inputs are prepared using the GPT-2 Byte Pair Encoding (BPE) tokenizer from Hugging Face. I configured the tokenizer and data pipeline to support truncation to a fixed maximum length and to use an appropriate padding token compatible with GPT-2's causal language modeling objective. Hugging Face's Data Collator For Language Modeling is used to automatically create shifted labels and mask out padding positions, ensuring that loss is only computed over valid tokens.

```
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False,
)
```

I implemented the full training and validation workflow using a custom loop: mini-batches are loaded with PyTorch Data Loader, passed through the GPT-2 model, and optimized with Adam using a low learning rate for stable fine-tuning. For each epoch, I compute the loss over non-ignored tokens, accumulate total loss per token, and track validation loss. The best-performing checkpoint (based on validation loss) is saved separately, while a final model is also stored at the end of training for reproducibility.

For generating stories, I implemented a decoding function that builds prompts without appending the gold story and then calls GPT-2's generate method. The function uses controlled sampling strategies, including temperature scaling, nucleus sampling (top-p), and a repetition penalty, to produce fluent yet diverse continuations. After generation, the original prompt is stripped from the output so that only the model-generated story is passed to the evaluation pipeline.

```
def generate_story(prompt, max_new_tokens=256, temperature=0.6, top_p=0.95):  1 usage
    model.eval()
    with torch.no_grad():
        inputs = tokenizer(prompt, return_tensors="pt").to(device)
        input_ids = inputs["input_ids"]
```

Finally, I integrated this baseline with the shared evaluation framework by formatting outputs into a standardized DataFrame containing reference stories, generated stories, and cleaned keyword lists. This allowed direct computation of BLEU, ROUGE, BERTScore, perplexity, and keyword-coverage metrics, and ensured that GPT-2 results could be fairly compared with other architectures.

The evaluation outcomes indicate that GPT-2 performs moderately well in capturing the semantic meaning of the reference stories, achieving a BERTScore F1 of approximately **0.61**, and demonstrates reasonable lexical overlap with ROUGE-1 at **0.43**. The model also shows strong adherence to required keywords, reaching **80.9% keyword coverage** and **53% strict keyword accuracy**. However, the low BLEU score (**0.0829**) and comparatively high perplexity (**13.55**) reveal its limitations in producing fluent, precise, and structurally aligned narrative continuations.

```
Validation Evaluation Summary:
corpus_bleu: 0.0829
avg_perplexity: 13.5553
bert_precision: 0.5843
bert_recall: 0.6464
bert_f1: 0.6132
rouge1: 0.4313
rouge2: 0.1386
rougeL: 0.2395
keyword_strict_accuracy: 0.5300
keyword_avg_percentage: 80.9038
```

Overall, GPT-2 serves as a meaningful and informative baseline, capable of generating coherent content yet still exhibiting noticeable weaknesses in fluency and narrative precision. These results highlight the necessity and value of exploring more advanced or specialized architectures, to achieve higher-quality story generation.

2.4 Building the LSTM + Attention Model

To complement the GPT-2 baseline with a classical neural benchmark, I implemented a sequence-to-sequence LSTM encoder–decoder model with Bahdanau additive attention. Each example uses the same structured prompt format as GPT-2 plus the gold story; both are tokenized with the GPT-2 BPE tokenizer. Decoder sequences include explicit BOS/EOS tokens and are padded to fixed lengths for efficient batching.

```
class BahdanauAttention(nn.Module):  1 usage
    def __init__(self, hidden_dim):
        super().__init__()
        self.W1 = nn.Linear(hidden_dim, hidden_dim)
        self.W2 = nn.Linear(hidden_dim, hidden_dim)
        self.V = nn.Linear(hidden_dim, 1)

    def forward(self, decoder_hidden, encoder_outputs, mask=None):

        dec_hidden_exp = decoder_hidden.unsqueeze(1).expand_as(encoder_outputs)
        score = self.V(torch.tanh(self.W1(encoder_outputs) + self.W2(dec_hidden_exp))).squeeze(-1)
        if mask is not None:
            score = score.masked_fill(mask == 0, -1e9)
        attn_weights = torch.softmax(score, dim=-1)  # (B, T)
        context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)  # (B, 1, H)
        context = context.squeeze(1)  # (B, H)
        return context, attn_weights
```

The encoder is a single-layer unidirectional LSTM that converts the tokenized prompt into hidden states. During decoding, each token is predicted using the previous decoder state, the current embedded input token, and an attention-based context vector over encoder states, allowing the model to focus on important parts of the prompt (e.g., keyword lists and long descriptions).

```python
# ------------------------------------------------------
# 6. LSTM + Attention Model
# ------------------------------------------------------
class Encoder(nn.Module):  1 usage
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=PAD_ID)
        self.lstm = nn.LSTM(
            embed_dim,
            hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0.0,
            bidirectional=False,
        )

    def forward(self, src_ids, src_mask):
        embedded = self.embedding(src_ids)  # (B, T, E)
        outputs, (h, c) = self.lstm(embedded)  # outputs: (B, T, H)
        return outputs, (h, c)
```

```python
class Decoder(nn.Module):  1 usage
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=PAD_ID)
        self.lstm = nn.LSTM(
            embed_dim + hidden_dim,  # concat [embed, context]
            hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0.0,
        )
        self.attention = BahdanauAttention(hidden_dim)
        self.fc_out = nn.Linear(hidden_dim, vocab_size)

    def forward(self, dec_input_ids, initial_hidden, initial_cell, encoder_outputs, src_mask):
        embedded = self.embedding(dec_input_ids)  # (B, T_dec, E)
        B, T_dec, _ = embedded.size()
        h, c = initial_hidden, initial_cell

        logits = []
        for t in range(T_dec):
            emb_t = embedded[:, t, :]  # (B, E)
            context, _ = self.attention(h[-1], encoder_outputs, src_mask)  # (B, H)
            lstm_input = torch.cat([emb_t, context], dim=-1).unsqueeze(1)  # (B, 1, E+H)
            output, (h, c) = self.lstm(lstm_input, (h, c))  # (B, 1, H)
            step_logits = self.fc_out(output.squeeze(1))  # (B, vocab)
            logits.append(step_logits.unsqueeze(1))

        logits = torch.cat(logits, dim=1)  # (B, T_dec, vocab)
        return logits
```

```python
class Seq2SeqAttnModel(nn.Module):  1 usage
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.encoder = Encoder(vocab_size, embed_dim, hidden_dim, num_layers, dropout)
        self.decoder = Decoder(vocab_size, embed_dim, hidden_dim, num_layers, dropout)

    def forward(self, src_ids, src_mask, dec_input_ids, dec_target_ids=None):
        enc_outputs, (h, c) = self.encoder(src_ids, src_mask)
        logits = self.decoder(dec_input_ids, h, c, enc_outputs, src_mask)
        return logits


model = Seq2SeqAttnModel(
    vocab_size=VOCAB_SIZE,
    embed_dim=EMBED_DIM,
    hidden_dim=HIDDEN_DIM,
    num_layers=NUM_LAYERS,
    dropout=DROPOUT,
).to(device)

print(model)
```

The model is trained end-to-end in PyTorch with teacher forcing, cross-entropy loss that ignores padding tokens, the Adam optimizer (learning rate 3×10^{-4}), and gradient clipping. Training and validation loss are tracked each epoch, and the best checkpoint by validation loss is saved for reproducibility. For inference, I reuse the GPT-2 prompt format without the gold story and generate continuations via greedy decoding from the BOS token until EOS or a maximum length. Generated stories are decoded back to text and passed into the shared evaluation pipeline, enabling direct comparison with GPT-2.

```
Validation Evaluation Summary:
corpus_bleu: 0.1219
avg_perplexity: 9.5241
bert_precision: 0.6721
bert_recall: 0.6222
bert_f1: 0.6456
rouge1: 0.4509
rouge2: 0.1952
rougeL: 0.3054
keyword_strict_accuracy: 0.0672
keyword_avg_percentage: 42.9269
```

The LSTM-Attention model shows clear improvements over the GPT-2 baseline in several core metrics. It achieves stronger lexical and structural alignment, as evidenced by higher BLEU (0.1219) and ROUGE scores (ROUGE-2 = 0.1952, ROUGE-L = 0.3054). Its semantic quality also improves, with a higher BERTScore F1 (0.6456) and a substantially lower perplexity (9.52), indicating more fluent and confident story generation.

Despite these gains, the model performs poorly in keyword-controlled generation, reaching only 6.7% strict keyword accuracy and 42.9% keyword coverage, showing that it often fails to incorporate required terms. Overall, the LSTM-Attention architecture produces more coherent and well-structured stories than GPT-2 but struggles with constraint satisfaction, limiting its effectiveness in tasks that depend on reliable keyword adherence.

2.5 Additional Contributions

Beyond implementing and evaluating the core models, I also contributed to several supporting components that were essential to the overall progress and clarity of the project.

• Model Testing and Debugging:
I assisted in validating generated outputs across different model iterations, systematically identifying issues such as incorrect token handling, attention instability, and decoding errors. I helped troubleshoot these problems by refining data preprocessing steps, adjusting training configurations, and verifying evaluation pipelines to ensure reliable and consistent model performance.

• Presentation and Report Development:
I contributed extensively to the project's documentation and presentation materials by writing the introductory sections, summarizing the exploratory data analysis (EDA), and explaining the evaluation metrics used across all models. In addition, I prepared clear descriptions of the GPT-2 baseline and the LSTM-Attention architecture, including their design, training workflows, and experimental results, ensuring that each model's behavior, strengths, and limitations were accurately and effectively presented in both the written report and the final project presentation.

3. Summary and Conclusions

Through building and evaluating both the GPT-2 baseline and the LSTM-Attention model, I learned that strong performance depends as much on computational resources as on model architecture and tuning. Even with optimized token lengths, batch sizes, and learning rates, training these models remained resource intensive. A single full training and evaluation cycle took nearly 12 hours, demonstrating how sensitive text-generation systems are to hardware limitations. These constraints directly influenced the types of experiments that were feasible and the extent to which hyperparameters could be explored.

The practical challenges underscored that scaling larger datasets, longer sequences, or more advanced architectures would require significantly greater computational power. Real-world deployment of such models demands hardware capable of supporting faster training iterations, larger memory capacities, and more efficient experimentation. This experience highlighted the importance of balancing model complexity, training cost, and available resources, illustrating that high-quality text-generation performance ultimately relies on both sound model design and adequate computational infrastructure.

4. Reflection on My Coding Process

I am still developing my coding skills, so I sought guidance from ChatGPT throughout the project. I requested support in generating the code incrementally, focusing on small sections aligned with the requirements of my models. Although I received assistance, I made sure to review each part of the code in detail to understand the purpose and functionality of every step. This process allowed me to learn the underlying logic of the implementation rather than simply copying the code without comprehension.