# Building an AI-Powered Urdu Poetry Generator with GRU and Streamlit

In this blog, I’ll walk you through my latest project: an \*\*Urdu Poetry Generator\*\* powered by a \*\*Gated Recurrent Unit (GRU)\*\* model. This application allows users to generate beautiful Urdu poetry by providing a seed text and adjusting the creativity and length of the poem. The app is deployed on \*\*Streamlit\*\*, making it accessible to anyone with an internet connection. I’ll explain both the \*\*.py\*\* and \*\*.ipynb\*\* files in detail, discuss the benefits of using GRU over RNN or LSTM, and leave space for screenshots to give you a visual understanding of the project.

---

## The Inspiration

Urdu poetry is known for its depth, emotion, and beauty. As someone who appreciates the art of poetry, I wanted to create a tool that could generate Urdu verses in a way that feels authentic and creative. With advancements in deep learning, particularly in sequence modeling, I decided to build a model that could learn the patterns of Urdu poetry and generate new verses based on a given seed text.

---

## The Tech Stack

The project is built using the following technologies:

1. \*\*Python\*\*: The primary programming language used for both model training and deployment.

2. \*\*PyTorch\*\*: A powerful deep learning framework used to build and train the GRU model.

3. \*\*Streamlit\*\*: A fantastic tool for building interactive web applications with minimal effort.

4. \*\*Pandas\*\*: Used for data manipulation and preprocessing.

5. \*\*TQDM\*\*: For displaying progress bars during model training.

---

## The Dataset

The dataset used for this project is a collection of \*\*Roman-Urdu poetry\*\*. Each poem is written in Roman script, making it easier to process and tokenize. The dataset contains over 1,300 poems, which were cleaned and preprocessed before being fed into the model.

---

## Code Explanation: Jupyter Notebook (`.ipynb`)

The Jupyter Notebook (`poemgenerator.ipynb`) is where the model was developed, trained, and tested. Let’s break it down step by step.

### 1. \*\*Data Loading and Preprocessing\*\*

The first step was to load the dataset and preprocess it. The dataset was loaded using Pandas, and the text was tokenized into individual characters. A vocabulary of unique characters was created, and each character was mapped to an integer for the model to process.

```python

import pandas as pd

# Load the dataset

data = pd.read\_csv('/kaggle/input/poetry-data-set/Roman-Urdu-Poetry.csv', usecols=[2], names=["text\_cleaned"])

```

### 2. \*\*Tokenization\*\*

The text was tokenized into individual characters, and a vocabulary was created using Python’s `Counter` class. The characters were then mapped to integers for the model to process.

```python

from collections import Counter

# Tokenize into characters

characters = list("".join(data['text\_cleaned']))

# Create vocabulary

vocab = Counter(characters)

vocab = sorted(vocab, key=vocab.get, reverse=True)

vocab\_to\_int = {char: i for i, char in enumerate(vocab)}

int\_to\_vocab = {i: char for char, i in vocab\_to\_int.items()}

```

### 3. \*\*Sequence Creation\*\*

The text was divided into input-output sequences, where the model predicts the next character based on the previous ones. This is a common approach in text generation tasks.

```python

seq\_length = 102

input\_sequences = []

output\_sequences = []

for i in range(len(encoded\_char) - seq\_length):

input\_seq = encoded\_char[i:i + seq\_length]

output\_seq = encoded\_char[i + seq\_length]

input\_sequences.append(input\_seq)

output\_sequences.append(output\_seq)

```

### 4. \*\*Model Architecture: GRU\*\*

The GRU model was defined using PyTorch. GRUs are a type of recurrent neural network (RNN) that are well-suited for sequence modeling tasks like text generation. They are computationally efficient and can capture long-term dependencies in sequential data.

```python

import torch.nn as nn

class GRUPoemGenerator(nn.Module):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, hidden\_size, num\_layers):

super(GRUPoemGenerator, self).\_\_init\_\_()

self.hidden\_size = hidden\_size

self.num\_layers = num\_layers

self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

self.gru = nn.GRU(embedding\_dim, hidden\_size, num\_layers, batch\_first=True)

self.fc = nn.Linear(hidden\_size, vocab\_size)

def forward(self, x):

x = self.embedding(x)

out, hidden = self.gru(x)

out = self.fc(out[:, -1, :])

return out, hidden

```

### 5. \*\*Training the Model\*\*

The model was trained for \*\*20 epochs\*\* using the \*\*Adam optimizer\*\* and \*\*CrossEntropyLoss\*\* as the loss function. The training process was monitored using \*\*TQDM\*\*, which provided a progress bar for each epoch.

```python

def train\_model(epochs):

for epoch in range(epochs):

total\_loss = 0.0

for inputs, targets in tqdm(train\_loader, desc=f'Epoch {epoch+1}/{epochs}', leave=False):

inputs, targets = inputs.to(device), targets.to(device)

optimizer.zero\_grad()

output, hidden = model(inputs)

loss = criterion(output, targets)

loss.backward()

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=1)

optimizer.step()

total\_loss += loss.item()

avg\_loss = total\_loss / len(train\_loader)

print(f"Epoch {epoch+1}/{epochs}, Loss: {avg\_loss:.4f}")

```

---

## Code Explanation: Streamlit App (`.py`)

The Streamlit app (`poemgen.py`) is where the trained model is deployed as an interactive web application. Let’s break it down step by step.

### 1. \*\*Page Configuration\*\*

The app’s page configuration is set using Streamlit’s `set\_page\_config` function. This includes the title, icon, layout, and initial state of the sidebar.

```python

st.set\_page\_config(

page\_title="Urdu Poetry Generator",

page\_icon="📝",

layout="wide",

initial\_sidebar\_state="collapsed"

)

```

### 2. \*\*Custom Styling\*\*

Custom CSS is used to style the app, including the title, input fields, and poem container.

```python

st.markdown("""

<style>

.main {

max-width: 1200px;

margin: 0 auto;

padding: 2rem;

}

.stTitle {

color: #1E88E5;

text-align: center;

margin-bottom: 2rem;

font-size: 2.5rem !important;

}

.poem-container {

background-color: transparent;

color: #ffffff;

padding: 40px;

border-radius: 15px;

font-size: 24px;

direction: rtl;

text-align: right;

box-shadow: 0 4px 15px rgba(0, 0, 0, 0.2);

margin: 30px auto;

max-width: 800px;

border: 1px solid rgba(255, 255, 255, 0.1);

}

</style>

""", unsafe\_allow\_html=True)

```

### 3. \*\*Model Loading\*\*

The trained GRU model is loaded using PyTorch’s `torch.load` function. The model is then set to evaluation mode for inference.

```python

@st.cache\_resource

def load\_model():

model = GRUPoemGenerator(vocab\_size, embedding\_dim, hidden\_dim, n\_layer).to(device)

model\_path = Path(\_\_file\_\_).parent / 'gru\_model.pth'

checkpoint = torch.load(model\_path, map\_location=device)

model.load\_state\_dict(checkpoint['model\_state\_dict'])

model.eval()

return model

```

### 4. \*\*Poem Generation\*\*

The `generate\_poem` function takes a seed text and generates a poem based on the model’s predictions. The temperature parameter controls the creativity of the generated text.

```python

def generate\_poem(model, seed\_text, vocab\_to\_int, int\_to\_vocab, seq\_length, length, temperature):

characters = list(seed\_text)

device = next(model.parameters()).device

with torch.no\_grad():

for \_ in range(length):

input\_seq = torch.tensor([[vocab\_to\_int[char] for char in characters[-seq\_length:]]], dtype=torch.long).to(device)

output, \_ = model(input\_seq)

probabilities = F.softmax(output / temperature, dim=-1)

next\_char\_idx = torch.multinomial(probabilities, 1).item()

next\_char = int\_to\_vocab[next\_char\_idx]

characters.append(next\_char)

return ''.join(characters)

```

### 5. \*\*User Interface\*\*

The app’s user interface is built using Streamlit’s interactive components, including sliders for adjusting creativity and poem length, and a text input for the seed text.

```python

def main():

st.title("✨ Urdu Poetry Generator ✨")

st.markdown("<p style='text-align: center; font-size: 1.2rem; color: #888;'><em>Create beautiful Urdu poetry using artificial intelligence</em></p>", unsafe\_allow\_html=True)

with st.sidebar:

st.markdown("<h3 style='text-align: center; color: #1E88E5;'>🎮 Generation Settings</h3>", unsafe\_allow\_html=True)

temperature = st.slider("🌡️ Creativity Level", 0.1, 1.0, 0.5, 0.1)

length = st.slider("📏 Poem Length", 50, 500, 250, 50)

seed\_text = st.text\_input("✍️ Enter your starting words:", value="ishq ", help="Start your poem with a few words")

generate\_button = st.button("✨ Generate Poem ✨")

if generate\_button:

model = load\_model()

with st.spinner("🎨 Crafting your masterpiece..."):

generated\_text = generate\_poem(model, seed\_text, vocab\_to\_int, int\_to\_vocab, seq\_length, length, temperature)

st.markdown("<h3 style='text-align: center; color: #1E88E5;'>📜 Your Generated Poem</h3>", unsafe\_allow\_html=True)

st.markdown(f"<div class='poem-container'>{generated\_text}</div>", unsafe\_allow\_html=True)

```

---

## Benefits of Using GRU Over RNN or LSTM

1. \*\*Efficiency\*\*: GRUs are computationally more efficient than LSTMs because they have fewer parameters. This makes them faster to train and less resource-intensive.

2. \*\*Simplicity\*\*: GRUs have a simpler architecture compared to LSTMs, with only two gates (reset and update) instead of three (input, output, and forget).

3. \*\*Performance\*\*: GRUs often perform as well as LSTMs in many sequence modeling tasks, including text generation, while being faster to train.

4. \*\*Memory\*\*: GRUs are better at capturing long-term dependencies in sequences compared to standard RNNs, making them more suitable for tasks like text generation.

---

## Screenshots

![Streamlit App Screenshot](#)

\*Caption: The Streamlit app interface for generating Urdu poetry.\*

![Training Progress](#)

\*Caption: Training progress of the GRU model over 20 epochs.\*

---

## Conclusion

Building this Urdu Poetry Generator was a rewarding experience that combined my love for poetry with my passion for AI. The GRU model, combined with Streamlit’s ease of use, resulted in a powerful and interactive application that can generate beautiful Urdu verses with just a few clicks. I hope this project inspires others to explore the creative possibilities of AI and NLP.

Feel free to check out the app and generate your own Urdu poetry! You can find the code and deployment details on my [GitHub repository](#).

---

\*\*Attachments\*\*:

- [Streamlit App Screenshot](#)

- [Jupyter Notebook (poemgenerator.ipynb)](#)

- [Streamlit Python File (poemgen.py)](#)

Let me know your thoughts in the comments below! Have you ever built a text generation model? What challenges did you face? I’d love to hear your experiences!