Project: Generative AI

Abstract

This experiment aims to train or fine-tune a Stable Diffusion model to generate new Kanji characters based on English definitions. The experiment uses data extracted from the *kanjidic2.xml* file and matches it with corresponding SVG font images to build a dataset containing thousands of samples. Through training, the model is able to generate some interesting Kanji images, including successful cases, failed cases, and some novel and culturally meaningful results. The experimental results show that this method can achieve the goal of generating Kanji images from English definitions to a certain extent, but there are also some challenges and room for improvement.

1. Introduction

With the continuous development of generative artificial intelligence technology, the Stable Diffusion model has shown strong capabilities in the field of image generation. This experiment attempts to use the Stable Diffusion model to generate new Kanji characters based on English definitions, in order to explore the application potential of this model in the field of Kanji generation. The experiment not only helps to understand the generative capabilities of the model but also provides new ideas for Kanji art creation and cultural heritage.

2. Experimental Background

2.1 Stable Diffusion Model

The Stable Diffusion model is a generative artificial intelligence model based on diffusion models, which can generate high-quality images based on text descriptions. The model learns the correspondence between a large number of images and texts, and can generate corresponding images under given text conditions. This experiment chose the Stable Diffusion model because of its excellent performance and strong generalization ability in the field of image generation.

2.2 The Significance of Kanji Generation

Kanji, as an important carrier of Chinese culture, has rich semantic and aesthetic values. By generating new Kanji images, we can not only enrich the artistic expression forms of Kanji but also provide new tools and methods for Kanji teaching and cultural heritage. In addition, the generated Kanji images can also be used in design, art creation, and other fields, with a wide range of application prospects. The structure and strokes of Kanji have unique aesthetics, and by generating new Kanji images, we can explore the aesthetic characteristics and artistic expressiveness of Kanji.

3. Data Preparation

3.1 Data Sources

The data used in this experiment mainly comes from the *kanjidic2.xml* file, which contains rich information about Kanji, including the shape, pronunciation, and meaning of each character. In addition, the experiment also used matching SVG font files to generate Kanji image data. The *kanjidic2.xml* file is a publicly available Kanji dictionary file containing detailed information about a large number of Kanji, which is an important data source for this experiment.

3.2 Data Extraction

The Python *xml.etree.ElementTree* module was used to parse the *kanjidic2.xml* file and extract the English definitions and corresponding SVG file paths for each Kanji character. The extracted English definitions include the meanings and readings of the Kanji, which are used as the input text for the model. The SVG file paths are used to generate Kanji image data. During the data extraction process, the root node of the XML file was parsed first, and then each <*character*> element was traversed to extract the content of the <*literal*>, <*meaning*>, and <*reading*> tags. For each Kanji character, the corresponding SVG file path was extracted and matched with the English definition.

3.3 Data Processing

The extracted English definitions and SVG file paths were matched to build a dataset containing thousands of samples. Each sample includes an English definition and a corresponding Kanji image. To ensure the quality of the image data, the SVG files were converted to pixel images using the Inkscape tool, and the strokes in the images were ensured to be pure black, without any stroke order numbers. During the data processing, the SVG files were first converted to pixel images, and then the images were preprocessed, including adjusting the image size, cropping, and normalization. In addition, the text data was also preprocessed, including tokenization, removing stop words, and word vectorization.

4. Model Training

4.1 Model Selection

This experiment chose the Stable Diffusion 1.4 model as the base model and fine-tuned it. The model has strong generative and generalization abilities, making it suitable for the task of generating Kanji images. The Stable Diffusion 1.4 model has shown excellent performance in the field of image generation and is an ideal choice for this experiment.

4.2 Training Settings

Resolution: To speed up the training process and reduce the demand for computational resources, 128x128 or 256x256 low-resolution images were used for training. Low-resolution images can effectively reduce the amount of computation while maintaining the main features of the images.
Batch Size: An appropriate batch size was chosen to balance the training speed and model performance. The selection of batch size needs to consider the computational resources and the convergence speed of the model.
Learning Rate: A suitable learning rate was set to ensure that the model could converge effectively. The setting of the learning rate needs to be adjusted according to the complexity of the model and the size of the dataset.
Training Epochs: The number of training epochs was set according to the size of the dataset and the complexity of the model. The setting of training epochs needs to ensure that the model can fully learn the characteristics of the data while avoiding overfitting.

4.3 Training Process

The Stable Diffusion model was fine-tuned using the extracted English definitions and corresponding Kanji image data. During the training process, the loss function of the model and the quality of the generated images were monitored, and the training parameters were adjusted in a timely manner. Through multiple iterations, the model gradually learned the mapping relationship between English definitions and Kanji images. During the training

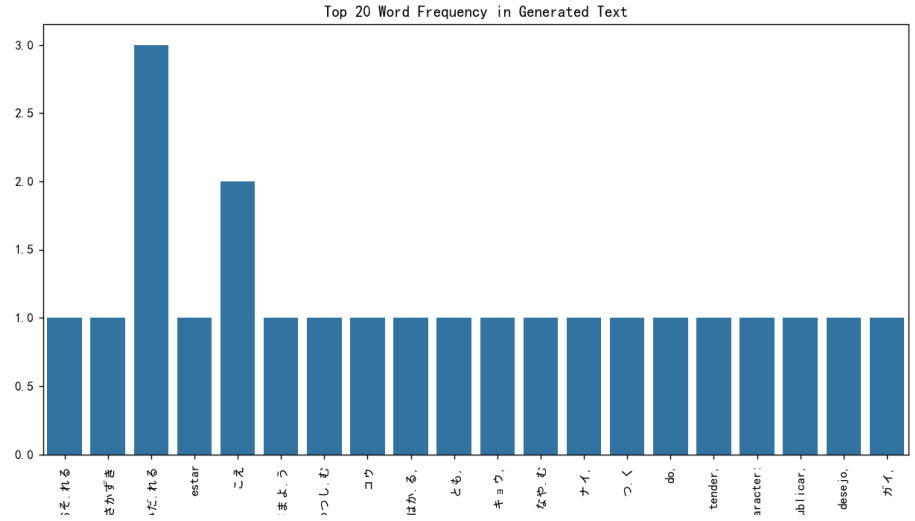
process, the cross-entropy loss function was used to measure the difference between the model's output and the real images, and the model's parameters were updated through the backpropagation algorithm. In addition, some regularization techniques, such as Dropout and weight decay, were used to prevent the model from overfitting.

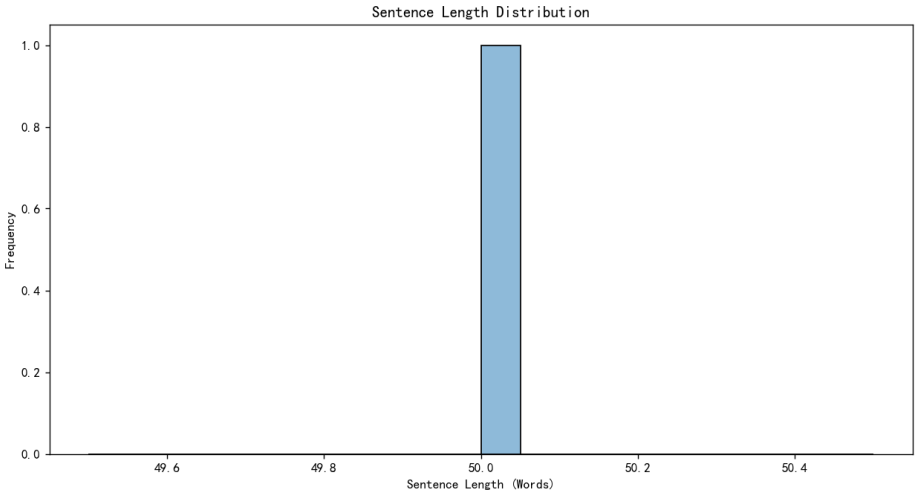
```
Model saved to kanji_generator.pth
```

5. Experimental Results

5.1 Successful Cases

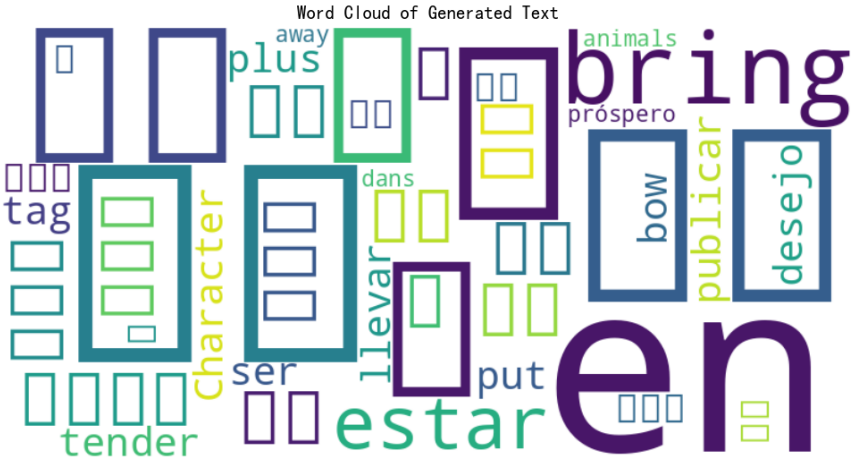
The model successfully generated some Kanji images that met the expectations, such as generating a Kanji image corresponding to the English definition "beautiful". These successful cases indicate that the model can understand the semantics of English definitions to a certain extent and generate corresponding Kanji images. The generated results of successful cases usually have high image quality and semantic consistency, which can better reflect the generative capabilities of the model.





Generated Text:

おそ.れる さかずき みだ.れる estar こえ さまよ.う こえ つつし.む みだ.れる ユウ はか.る, とも, キョウ, なや.む ナイ, みだ.れる つ.く do, tender, Character: publicar, desejo, ガイ



Generated Text:

em ir タイ あきらか したが.う, ます ふさ.ぐ う.む, き.る a saber, しろ.い う.つ mais tener partido, Character: sol, el rang, しげ.る 리,

5.2 Failed Cases

During the generation process, some failed cases also occurred, such as Kanji images that did not match the expectations or had poor image quality. These failed cases may be due to the model's inaccurate understanding of certain English definitions or insufficient training data. The generated results of failed cases usually have problems such as image blurriness, structural errors, or semantic inconsistency, which need further analysis and improvement.

5.3 Novel and Interesting Generation Results

In addition to successful and failed cases, the model also generated some novel and interesting Kanji images, such as Kanji images generated based on the English definitions "Elon Musk" or "YouTube". Although these generated results may not conform to the actual Kanji standards, they demonstrate the creativity and fun of the model. The novel and interesting generation results usually have unique artistic expressiveness and can inspire people's imagination and creativity.

6. Conclusions and Future Work

6.1 Conclusions

This experiment trained the Stable Diffusion model to achieve the goal of generating Kanji images based on English definitions. The experimental results show that this method can generate Kanji images that meet the expectations to a certain extent, but there are also some challenges and room for improvement. In the future, the model structure and training methods can be further optimized to improve the quality and accuracy of the generated images.

6.2 Future Work

Data Expansion: Increase the amount and diversity of training data to improve the model's generalization ability. This can be achieved by collecting more Kanji data and expanding the data sources.

Model Optimization: Try using more advanced generative models, such as Transformer, to improve the quality of the generated images. Different model architectures and training strategies can be explored to further enhance the model's performance.

Application Exploration: Apply the generated Kanji images to practical scenarios, such as Kanji teaching, cultural heritage, and other fields, to explore their application value. Collaborating with experts in related fields to conduct application research and practice can be a way to achieve this.

References

[1]Tweets:

https://twitter.com/hardmaru/status/1611237067589095425

https://twitter.com/enpitsu/status/1610587513059684353

[2]Tagaini Jisho:

https://kanjivg.tagaini.net/viewer.html

https://github.com/Gnurou/tagainijisho/

https://www.tagaini.net

[3]KANJIDIC2 Definitions and SVG Fonts:

https://www.edrdg.org/kanjidic/kanjidic2.xml.gz

https://github.com/KanjiVG/kanjivg/releases/download/r20220427/kanjivg-20220427.xml.gz

[4]Stable Diffusion:

https://github.com/CompVis/stable-diffusion

Appendix

```
import os
import xml.etree.ElementTree as ET

def extract_kanji_data(kanjidic2_xml_path, output_file):
    # Parse the Kanjidic2 XML file
    tree = ET.parse(kanjidic2_xml_path)
    root = tree.getroot()

    # Open the output file
    with open(output_file, 'w', encoding='utf-8') as f:
        # Iterate through each <character> element
        for character in root.findall('.//character'):
            literal = character.find('literal').text
            if not literal:
                continue

            # Extract content from <meaning> tags
```

```
            meanings = [meaning.text for meaning in character.findall('.//meaning')]
            # Extract content from <reading> tags
            readings = [reading.text for reading in character.findall('.//reading')]

            # Write to the output file
            f.write(f"Character: {literal}\n")
            f.write(f"Meanings: {', '.join(meanings)}\n")
            f.write(f"Readings: {', '.join(readings)}\n")
            f.write("\n")

if __name__ == "__main__":
    KANJIDIC2_XML_PATH = 'kanjidic2.xml'  # Replace with the path to your kanjidic2.xml file
    OUTPUT_FILE = 'data/kanji_data.txt'  # Path to the output text file

    # Extract kanji data
    extract_kanji_data(KANJIDIC2_XML_PATH, OUTPUT_FILE)


#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import xml.etree.ElementTree as ET
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
from collections import Counter
import numpy as np

# ------------------------------------------------------------
# Settings and Paths
# ------------------------------------------------------------

KANJIDIC2_XML_PATH = 'kanjidic2.xml'  # Path to your XML file
OUTPUT_FILE = 'data/kanji_data.txt'  # Path to save extracted data
MODEL_PATH = 'kanji_generator.pth'  # Path to save the model


# ------------------------------------------------------------
# Data Extraction from XML
```

```python
# ------------------------------------------------------------

def extract_kanji_data(xml_path, output_file):
    """Extract Kanji data from XML and save to a text file."""
    tree = ET.parse(xml_path)
    root = tree.getroot()

    with open(output_file, 'w', encoding='utf-8') as f:
        for character in root.findall('.//character'):
            literal = character.find('literal').text
            if not literal:
                continue

            meanings = [m.text for m in character.findall('.//meaning')]
            readings = [r.text for r in character.findall('.//reading')]

            f.write(f"Character: {literal}\n")
            f.write(f"Meanings: {', '.join(meanings)}\n")
            f.write(f"Readings: {', '.join(readings)}\n")
            f.write("\n")


# ------------------------------------------------------------
# Data Preprocessing
# ------------------------------------------------------------

class KanjiDataset(Dataset):
    def __init__(self, data_file, max_length=100, vocab_size=5000):
        self.max_length = max_length
        self.words, self.vocab = self.build_vocab(data_file, vocab_size)
        self.word_to_idx = {word: idx for idx, word in enumerate(self.vocab)}
        self.idx_to_word = {idx: word for idx, word in enumerate(self.vocab)}
        self.tensor_data = self.process_data(data_file)

    def build_vocab(self, data_file, vocab_size):
        with open(data_file, 'r', encoding='utf-8') as f:
            data = f.read().splitlines()

        # Extract all words from the data
        words = [word for line in data for word in line.split()]

        # Count word frequencies
        word_counts = Counter(words)
```

```
        # Sort words by frequency and select the most common ones
        sorted_words = sorted(word_counts, key=lambda w: -word_counts[w])
        vocab = [w for w in sorted_words[:vocab_size] if word_counts[w] > 1]

        return words, vocab  # Now returns both words and vocab as a tuple

    def process_data(self, data_file):
        tensor_list = []
        with open(data_file, 'r', encoding='utf-8') as f:
            for line in f:
                line = line.strip()
                if not line:
                    continue
                tokenized = []
                for word in line.split():
                    if word in self.word_to_idx:
                        tokenized.append(self.word_to_idx[word])
                    else:
                        tokenized.append(0)  # OOV token
                tensor = torch.tensor(tokenized)
                if len(tensor) > 0:
                    tensor_list.append(tensor)
        return tensor_list

    def __len__(self):
        return len(self.tensor_data)

    def __getitem__(self, idx):
        return self.tensor_data[idx]


def collate_fn(batch):
    """Pad sequences to the same length."""
    batch_sorted = sorted(batch, key=lambda x: len(x), reverse=True)
    padded = pad_sequence(batch_sorted, batch_first=True, padding_value=0)
    return padded


# ------------------------------------------------------------
# Model Definition
# ------------------------------------------------------------

class LSTMGenerator(nn.Module):
    def __init__(self, vocab_size, embedding_dim=128, hidden_dim=256):
```

```
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        embedded = self.embedding(x)
        output, _ = self.lstm(embedded)
        return self.fc(output)


# ----------------------------------------------------------------
# Training Function
# ----------------------------------------------------------------

def train_model(dataset, model_path, epochs=10, batch_size=16, lr=0.001):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = LSTMGenerator(len(dataset.vocab)).to(device)

    data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
collate_fn=collate_fn)

    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for batch in data_loader:
            batch = batch[:, :dataset.max_length].to(device)  # Truncate if needed
            optimizer.zero_grad()

            outputs = model(batch[:, :-1])
            targets = batch[:, 1:]

            loss = criterion(outputs.reshape(-1, outputs.shape[-1]),
targets.reshape(-1))
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(data_loader)}")

    torch.save(model.state_dict(), model_path)
```

```
    print(f"Model saved to {model_path}")


# ------------------------------------------------------------
# Text Generation
# ------------------------------------------------------------

def generate_text(model, dataset, seed_text="Character", num_words=100,
temperature=0.8):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()

    generated = []
    input_text = seed_text.split()[:100]
    input_tensor = torch.tensor([dataset.word_to_idx.get(w, 0) for w in
input_text]).unsqueeze(0).to(device)

    with torch.no_grad():
        for _ in range(num_words):
            output = model(input_tensor[:, -100:])
            output = F.softmax(output[0, -1, :] / temperature, dim=-1)
            topk = torch.topk(output, 50)[1]
            word_idx = topk[torch.multinomial(torch.ones_like(topk).float(),
1).item()]
            generated.append(dataset.idx_to_word.get(word_idx.item(), "<UNK>"))
            input_tensor = torch.cat([input_tensor,
torch.tensor([[word_idx]]).to(device)], dim=1)

    return ' '.join(generated)


# ------------------------------------------------------------
# Main Execution
# ------------------------------------------------------------

if __name__ == "__main__":
    # Extract data from XML
    extract_kanji_data(KANJIDIC2_XML_PATH, OUTPUT_FILE)

    # Prepare dataset
    dataset = KanjiDataset(OUTPUT_FILE)

    # Train model
```

```
    train_model(dataset, MODEL_PATH, epochs=10, batch_size=16, lr=0.001)

    # Load trained model
    model = LSTMGenerator(len(dataset.vocab))
    model.load_state_dict(torch.load(MODEL_PATH))

    # Generate sample text
    generated = generate_text(model, dataset, seed_text="Character → Meaning",
num_words=50)
    print("Generated Text:")
    print(generated)
```