Hong Kong Academy of Gifted Education

TECS3461 Independent Project on Artificial Intelligence and Application

Technical Report

[Haiku Generation]

Student: SONG Yiding
Supervisor: Prof. Tan Lee, Department of Electronic Engineering, CUHK

Task Description

The task in this project is to train an AI that can generate English haikus (a haiku is a short, three-line form of poetry) of unfixed length. The Haikus generated do not have to follow the 5-7-5 syllables convention, since this is currently beyond the scope of my capability.

My aim is that the generated Haikus should be indistinguishable from real ones by humans. Here, I approach the problem with a single character as a unit of generation, because I want to see if the AI can learn the relationships between characters, forming correct words and sentences with accurate grammar.

This task is interesting because it can provide a form of entertainment for the general public, demonstrating how poetry can be created by lifeless machines, seemingly incapable of creating language arts. However, it is also important because through it, we can evaluate the best approaches to take when handling text generation problems, and find out inherent problems in each.

There are two major approaches that one can take:

1) Tackling the task with a single RNN network: an RNN takes in a single character as input, and generates the most probable next character repeatedly
2) Treating the task as a GAN problem: a Discriminator learns the differences between real haikus and bad haikus, and the Generator tries to fool it with its own haikus

I've applied both approaches in Tensorflow, and the first performed evidently better. With the GAN approach, I've faced some major problems. These will all be detailed later.

Methods

I. The dataset

The dataset consists of 4828 English haikus in a text file, with erroneous punctuations (like useless tabs and spaces at the end of sentences) and overlong haikus removed, in case they interfere with the learning process.
Each haiku is separated by a token: '}\n\n' in the case of the GAN implementation, and '\n\n' for the RNN approach ('\n' = a new line). The file can be found here.

II. The system

1) An RNN Approach

a. Data Processing
The original dataset is loaded, and all of its characters converted into tokens. The tokens are non-negative integers ranging from 0 to char_size, where char_size is the number of unique characters in the dataset. Each unique character has its own token.
The whole dataset is converted into these tokens, split into input and output chunks of seq_length (an additional parameter determining the length of each individual sequence of tokenized characters fed into the RNN network during training; I chose this to be 100), and then shuffled and separated into batches for training.

b. The RNN Network [1]
During training, the RNN network is simply a network that takes in the first character token in a sequence as input in timestep 0, computes the most probable next character, maintains its 'memory' of that as a hidden state, then takes in the target character as input in the next timestep (timestep 1 here) and computes the most probable next character again, and so on.
To achieve that, the architecture of the RNN network starts with an embedding layer, which converts each character in the input sequence into a vector of arbitrary dimensions that has been defined by me as 128. These are passed on into the RNN, which is a GRU (Gated Recurrent Unit) layer. Lastly, the output layer is a dense layer with char_size neurons, which logits predict the log-likelihood of the next character.

c. The Losses and Optimizers
The loss is a cross entropy loss of the difference between the target sequence and the final output sequence (a sequence of logits) from the RNN. My optimizer implements the Adam optimization algorithm.

d. Training
In each training epoch, batches of input sequences of seq_length are repeatedly fed into the RNN until all of the batches in the dataset have been iterated over. When fed with each batch, the losses are computed

[1] The RNN approach is modified from the following tutorial on TensorFlow: https://www.tensorflow.org/tutorials/text/text_generation

after the RNN predicts the most-probable next characters throughout whole sequence, and its gradients are updated.

2) A GAN Approach

a. Data Processing
First, I load the dataset file from GitHub, a '.txt' file containing haikus separated by the token '}\n\n'. I separate the different haikus and convert all the characters in each haiku to tokens. The tokens are non-negative integers, just like in the RNN approach above. I further convert these tokens to one-hot vector encodings of each unique character. Each of these vectors have a dimension of char_size, and all of the values in it are 0, except for one value which is 1. The index of that value equals the original integer token. After that, I pad each haiku with a number of vectors consisting of only zeroes. These vectors for padding are of the same dimension as the one-hot vector encodings. After all the haikus have been tokenized and padded so that they are all of the same length as the longest one, the max_len, the processed dataset should have the following shape: [4812, max_len, char_size], where 4812 is the number of haikus. After shuffling the dataset and splitting it into batches of 128 haikus, it is now ready for training.

b. The Generator
The Generator takes in a random seed of the shape [batch_size, max_len, char_size]. The seed is fed through a GRU (Gated Recurrent Unit) layer, into several fully connected dense layers with a Leaky ReLU activation function. The output layer is a Softmax layer with char_size neurons. Ideally, the output should resemble the one-hot vector of a character in the dataset.
Since the first layer of the network is an RNN layer, the output of the last output vector it generated stays in its 'memory' as a hidden state. I hope this will make the Generator function better, because it generates text, and hence needs to learn some long-range dependencies between its outputs.
The final output of the RNN is of the shape [batch_size, max_len, char_size].

c. The Discriminator
The Discriminator's structure is very similar to the Generator, but it takes in batches of vectorised and padded haikus. Its output layer is a Sigmoid layer with only one neuron to limit the output value to between 0 and 1. In essence, it is just an RNN model aimed at training for regression problems.

d. The Losses and Optimizers [2]

[2] These are taken from the following tutorial on TensorFlow:
https://www.tensorflow.org/tutorials/generative/dcgan

I chose a cross entropy loss as the global loss function for both the Generator and Discriminator.
When the Discriminator is fed with data from the Generator, the target output is 0. The target output is 1 when the Discriminator is fed with real Haikus. The local Discriminator loss will be a sum of the two cross entropy losses of its deviations from the output targets after it has discriminated both generated Haiku and real Haiku inputs. That is:

```
real_haikus_loss = cross_entropy (1, real_haikus_discriminator_output)
fake_haikus_loss = cross_entropy(0, fake_haikus_discriminator_output)
total_loss = real_haikus_loss + fake_haikus loss
```

The Generator loss is again, a cross entropy loss, but it only considers how much the discriminated value of its generated Haikus diverges from 1, which is the value the Discriminator assigns to good haikus.
As for the optimizers, I chose an ADAM optimizer. Because I currently don't have time to explore how well other optimizers work, I don't know if this is the best choice.

e. The Training
In an attempt to avoid the Generator or Discriminator overpowering each other (e.g. Discriminator with a loss of 0.3 and Generator with a loss of 9.5), I trained the two in a dynamic process. Each epoch is separated into a number of training steps. In each step, the Generator first generates a number of haikus (this number equals to the number of haikus in each training batch in the dataset) from a random noise. The Discriminator then discriminates the generated haikus and a batch of haikus in the dataset. The losses of both the Discriminator and Generator in this step are calculated, but depending on which of the two networks we want to train, the gradients are updated. The exact mechanism for determining which of the two networks to train in each epoch are as follows:

Scenario 1: If it is the first training epoch, both models are trained
Scenario 2: Let D be the Discriminator loss in the last training epoch, and G be the Generator loss, then if $|D-G| < 0.5$, both models are trained
Scenario 3: If $D-G > 0.5$, then only the Discriminator is trained
Scenario 4: If $G-D > 0.5$, then only the Generator is trained

This proved to solve the problem of 'overpowering', which appeared during my early implementations.

Results and Findings

The RNN approach performed very well after around 50 training epochs, with each epoch costing me around 1 minute to run on Google's GPU. Here are some high-quality sample outputs from the trained model:

another spring
we talk about
the rain

snow shower -
the moon's reflection
in the stone buddha

long beach
in a silver midnight...
a snail crawls

The RNN model was not only capable of learning the relationship between characters, but also between words. This is a relatively successful implementation, but we can see some signs of overfitting. A lot of the generated haikus highly resemble those in the dataset. For instance, consider the following:

Original Data	RNN Output
brewing storm... how the clouds stir in my coffee	morning snow... how the clouds stir in my coffee

To solve this problem of haikus without 'originality', the GAN approach seems like the best solution. However, I encountered many problems during the training process. Most predominantly, the Generator seems to be incapable of 'fine-tuning' its gradients so as to produce better outputs that might fool the Discriminator.

Below are some helpful figures:

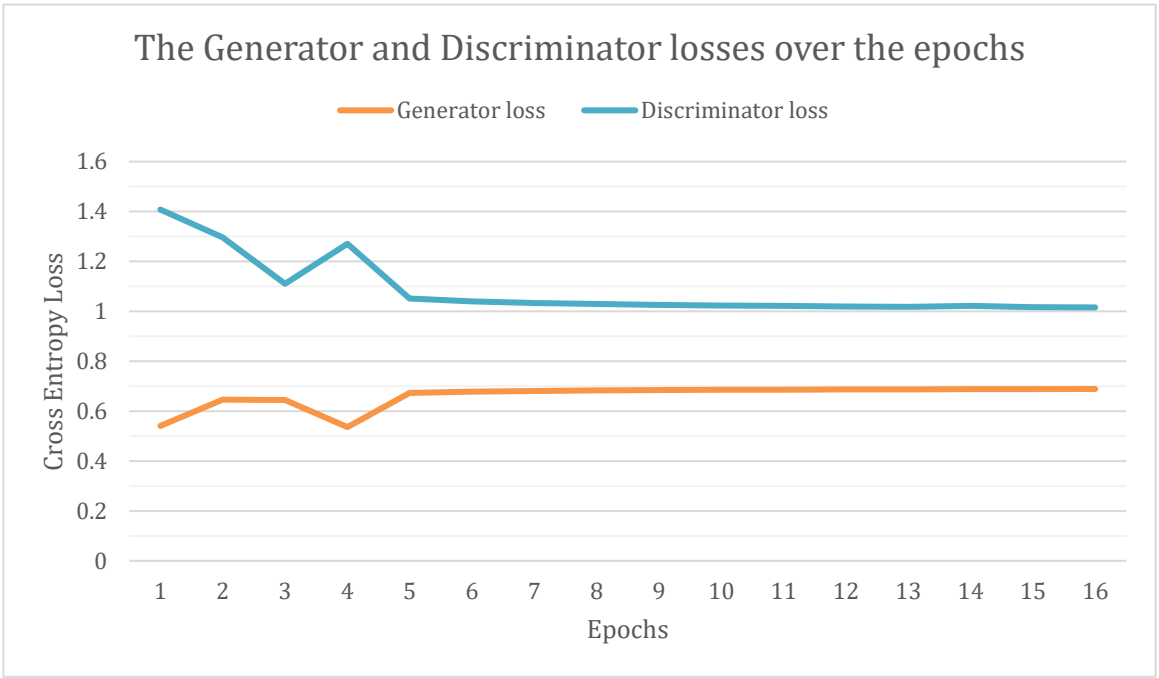

figure 1

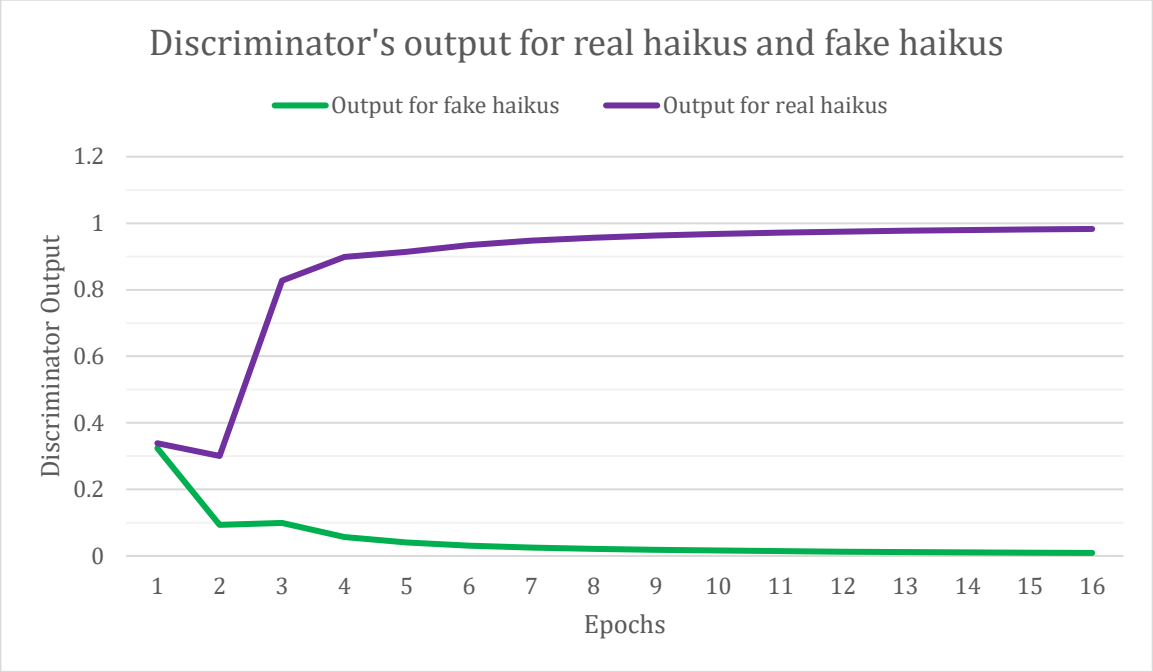

figure 2

As shown in *figure 1*, after brief fluctuations at first, the Generator loss and the Discriminator loss were almost stationary, with barely perceptible increases in that of the Generator and decreases for the Discriminator. The key problem here is that the Generator loss is unchanging, which results in an unchanging loss for the Discriminator. After each epoch, there is not much difference in the Generator's outputs, while according to *figure 2*, the Discriminator has almost completely learnt to differentiate between real haikus and generated haikus starting from the third epoch. After 16 epochs of training, this is the Generator output (converted to text) with a random seed as input: 'lllllllllllllllllllllllllllmlm llllrr lnmllllmllrlllllll mr llll lllmlllmllmllml '. The Discriminator gave this a core of 0.00906605, while in comparison, its score for the first haiku in the dataset is 0.9828997.

Evidently, the training has failed. The model of GAN I implemented cannot generate anything meaningful. It is not even capable of putting together characters into meaningful words, much less write poems. It seems that the RNN approach, although not perfect, enjoys a superiority in this task. Its problem of overfitting may be solved through other ways, like a larger dataset.

But what is the problem with the GAN approach? Surely there should be some other models of GAN that can do better in this task? The Discriminator is obviously competent in its job, and the that leaves us with only one possibility: the Generator cannot be trained efficiently in my approach. Based on several resources online, I believe this is due to the following reason:

GAN is designed for generating real-valued, continuous data, but has difficulties in directly generating sequences of discrete tokens, such as vectors in texts. [3]
Hence, using the gradient of the loss from the Discriminator with respect to the outputs of the Generator to guide its parameters makes little sense. [4] Now that we see why the Generator cannot learn properly, are there any solutions?

A way out?

SeqGAN treats the task as a Reinforcement Learning problem and seems to be quite capable of sequence generation, like generating poetry. There are many such tutorials online, and I won't explain it in detail here.

I have my own way out as well, though the results aren't as ideal, with more training epochs and a bigger dataset I'm confident it'll perform better. My solution is a combined implementation of the RNN approach and GAN approach. I replaced the Generator model with the RNN network, and thus it takes in sequences of integer tokens of individual haikus, all padded to max_len. It then generates sequences of log likelihoods. The Discriminator then takes in these log likelihoods, uses the Softmax function to convert them to vectors similar to the one-hot encodings in the processed GAN dataset, and discriminates the generated haikus. The Generator loss is the original loss in the GAN implementation plus the loss in the RNN approach.

This solution seems to solve some of the problem, at least now. In the end, after 403 epochs of training (each epoch typically taking about 18 seconds), this is a sample output:

inmer's end
my hand'lingers
in the dry grass

Although not ideal, it's already a huge improvement.

Future plan

If I am allowed to continue with the project, I would first try to find a larger dataset to prevent overfitting and facilitate training. Then, I will study SeqGAN in detail and understand how it manages to tackle the problems in simple GAN models aimed at sequence generation (namely, the discrete-token problem as mentioned above). I may even try implementing it on TensorFlow with my dataset. But most importantly, I will try drawing conclusions and insights from it and use these to perfect my own RNN and GAN combined approach to create an AI capable of generating original and beautiful haikus.

---

[3] 'SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient', by Lantao Yu, Weinan Zhang, Jun Wang, Yong Yu, published on arXiv (arXiv:1609.05473), last updated on 25 Aug 2017
[4] 'SeqGAN: GANs for sequence generation', by Pratheeksha Nair, published on Medium, last updated on Sep 27, 2018