Generating Swipeable Tinder Profiles using AI: Adversarial & Recurrent Neural Networks in Realistic Content Generation

Adrian Yijie Xu

May 1 · 10 min read ★

Introduction

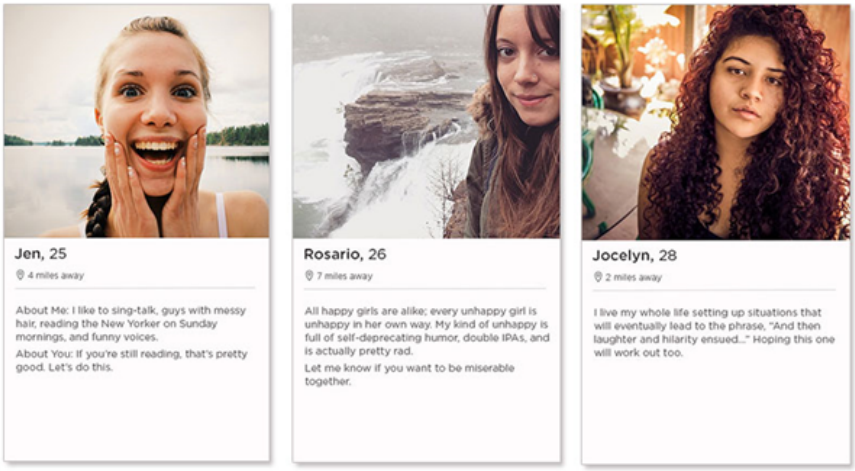
Over the past few articles, we've spent time covering two specialties of generative deep learning architectures covering image and text generation, utilizing Generative Adversarial Networks (GANs) and Recurrent Neural Networks (RNNs), respectively. We chose to introduce these separately, in order to explain their principles, architecture, and Python implementations in detail. With both networks familiarized, we've chosen to showcase a composite project with strong real-world applications, namely the generation of believable profiles for dating apps such as Tinder.

Calm down there Quagmire.

Fake profiles pose a significant issue in social networks—they can influence public discourse, indict celebrities, or topple institutions. **Facebook alone removed over 580 million profiles in the first quarter of 2018 alone**, while **Twitter removed 70 million accounts from May to June of 2018**.

On dating apps such as Tinder reliant on the desire to match with attractive members, **such profiles may lead to be serious financial ramifications on unsuspecting victims.** Thankfully, most of these can still be detected by visual inspection, as they often feature low-resolution images and poor or sparsely populated bios. Additionally, as most fake profile photos are stolen from legitimate accounts, there exists the chance of a real-world acquaintance recognizing the images, leading to faster fake account detection and deletion.

In the interests of science, let's play the devil's advocate here and ask ourselves: **could generate a swipeable fake Tinder profile**? **Can we generate a realistic representation and characterization of person that does not exist?** To better understand the challenge at hand, let's look at a few example female profiles:



From the profiles above, we can observe some shared commonalities—namely, the presence of a **clear facial image along** with a **text bio section consisting of multiple descriptive and relatively short phrases**. You'll notice that due to the artificial constraints of the bio length, these phrases are often entirely independent in terms of content from one another, meaning that an overarching theme may not exist in a single paragraph. **This is perfect for AI-based content generation.**

Fortunately, we already possess the components necessary to build the perfect profile—namely, StyleGANs and RNNs. We'll break down the

individual contributions from our components trained in Google's Colaboratory GPU environment, before piecing together a complete final profile. We'll be skipping through the theory behind both components as we've covered that in their respective tutorials, which we encourage you to skim over as a quick refresher.

Implementation

Image generation—StyleGAN

Briefly, StyleGANs are a subtype of Generative Adversarial Network created by an NVIDIA team designed to produce high-resolution and realistic images by generating different details at different resolutions to allow for the control of individual features while maintaining faster training speeds. We covered their use previously in generating artistic presidential portraits, which we encourage the reader to revisit.

For this tutorial, we'll be using a NVIDIA StyleGAN architecture pre-trained on the flicker FFHQ faces dataset, containing over 70,000 faces at a resolution of 1024^2, to generate realistic portraits for use in our profiles using Tensorflow.

In the interests of time, We'll use a pre-trained network to generate our images. **Our notebook is available here.** To summarize, we clone the NVIDIA StyleGAN repository, before loading the three core StyleGAN (karras2019stylegan-ffhq-1024x1024.pkl) network components, namely:

- An instantaneous memory snapshot of the generator
- An instantaneous memory snapshot of the discriminator
- A long term average of the generator, which tends to provide higher quality results than its instantaneous counterpart.

After initializing our Tensorflow session, we begin by loading in our pre-trained model.

```
# Load pre-trained network.
 url = 'https://drive.google.com/uc?
id=1MEGjdvVpUsu1jB4zrXZN7Y4kBBOzizDQ' # karras2019stylegan-
ffhq-1024x1024.pkl

 with dnnlib.util.open_url(url, cache_dir=config.cache_dir)
as f:
   _G, _D, Gs = pickle.load(f)
```

```
 # _G = Instantaneous snapshot of the generator. Mainly
useful for resuming a previous training run.
 # _D = Instantaneous snapshot of the discriminator. Mainly
useful for resuming a previous training run.
 # Gs = Long-term average of the generator. Yields higher-
quality results than the instantaneous snapshot.
```

Next, we randomly seed a latent vector (latent), which you can think of as a compressed blueprint of an image, to use as our input for the SyleGAN generator. We then run the generator together with various quality improving arguments, and save the image for use:

```
for i in range(1,20):

 rnd = np.random.RandomState(5)
 latents = rnd.randn(i, Gs.input_shape[1])
```

```
# Generate image.
 fmt = dict(func=tflib.convert_images_to_uint8,
nchw_to_nhwc=True)
 images = Gs.run(latents, None, truncation_psi=0.7,
randomize_noise=True, output_transform=fmt)
```

```
# Save image.
 os.makedirs(config.result_dir, exist_ok=True)
 png_filename = os.path.join(config.result_dir, 'example'
+str(i)+'.png')
 PIL.Image.fromarray(images[0], 'RGB').save(png_filename)
```

You'll find the output images in your *results* folder. A collage of examples is displayed below:

Example output images generated using the FFHQ pre-trained network

Most impressive. While you generate more images, let's get to work on the bio!

Text generation—RNN

Briefly, RNNs are a type of neural network that are designed to handle sequences by propagating information about each previous element in a sequence to make a predictive decision concerning the next element of the sequence. We covered their use previously in text sequence sentiment analysis, which we also encourage the reader to revisit.

For this tutorial, we'll be creating a simple character sequence based RNN architecture in Keras, which we will train on the Kaggle Tinder Female Profiles dataset, containing the collected details of over 250,000 female profiles, including their first names, age, and biographical information. **Our notebook, based on the CharTrump implementation and Brownlee's excellent tutorial on RNNs, is available here.**

Let's start by importing all of our standard packages and downloading our dataset:

```
#Import all packages


import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import string
import warnings
import sklearn
import io
import scipy
import numpy
import json
import nltk
import sys
import csv
import os
import re
from keras.models import Sequential
from keras.layers import LSTM, Dropout, Activation, Dense
```

```
!gdown https://drive.google.com/uc?
id=1G5ud0F1DYut4Vcd6WSPrS-CtD8notTSd
```

With the dataset downloaded, let's access the bios of each profile, defined by the *'Unnamed: 2'* column, and define a basic vocabulary of characters for our network. These represent characters that our network will recognize and output.

```
#Text import and preprocessing
base_dir = 'tinder_data_final.xlsx'
```

```
tinder_profile_df = pd.read_excel(base_dir,
error_bad_lines=False, delimiter='\t')
```

```
# Print dataframe info
tinder_profile_df.info()
```

```
print(tinder_profile_df.head())
```

```
# This will print the number of rows and columns that will
be used to train
print("Shape of train set : ",tinder_profile_df.shape)
```

```
#tinder_profile_df.head()
```

```
#Convert dataframe to string already here
```

```
tinder_bios_df=tinder_profile_df['Unnamed: 2'].apply(str)
```

```
#Print only thhe bios
#print(tinder_bios_df)
```

```
#Define a generic vocabulary
# generic vocabulary
characters = list(string.printable)
characters.remove('\x0b')
characters.remove('\x0c')


VOCABULARY_SIZE = len(characters)
characters_to_ix = {c:i for i,c in enumerate(characters)}
print("vocabulary len = %d" % VOCABULARY_SIZE)
print(characters)
```

To create our training data, we'll concatenate all of our profile bio information into a two large strings made up of smaller individual phrases, representing our training and validation datasets (split at an 80:20 ratio). We'll also remove any empty profiles and special characters in the process.

```
#We define a filtering function first
holder = []
for i in range(0, len(tinder_bios_df)):
 #print (tinder_bios_df.iloc[i]);
 string = tinder_bios_df.iloc[i]
```

```
if(string== "nan"):
 print("nan removed")
 else:
 holder.append(string)
```

```
#print(holder)
```

```
alphaholder= []
regex = re.compile('[^A-Za-z0-9 -.,]')
#now that holder is ready we remove all non alphanumeric
entries
for i in range(0, len(holder)):
 string = holder[i]


 #First parameter is the replacement, second parameter is
your input string
 newstring = regex.sub('', string)
```

```
alphaholder.append(newstring)


#print(alphaholder)
```

```
#Split data into train and validation strings
trainholder=[]
validationholder=[]
```

```
print(len(alphaholder))
#Len alphaholder is 43660
for i in range(0, 34928):
 trainstring = alphaholder[i]
 trainholder.append(trainstring)
for i in range(34928, len(alphaholder)):
 valstring = alphaholder[i]
 validationholder.append(valstring)


# Data is now ready, append into giant string
finalstring = ''.join(trainholder)
validationstring = ''.join(validationholder)
```

```
print(finalstring)
print(validationstring)
```

With our pre-processing done, let's get to building our model. Let's begin by defining our hyperparameters. The SEQUENCE_LEN and LAYER_COUNT parameters represent the size of the input sequence and the layer count of the network, respectively, and have a direct effect on training time and prediction output legibility.

The choice of 20 characters and 4 layers were chosen as being a good compromise between training speed and prediction legibility. Fortunately , the short characteristic of our input bio phrases makes 20

characters an excellent choice, but feel free to try other lengths on your own.

In addition, let's define functions to describe and supply our input data batches to our network.

```
N_GPU = 1
SEQUENCE_LEN = 20;
#This is the char length of each individual sequence
#Essentially, we use 60 chars to predict the 61st char. This
needs to be shorter for us
BATCH_SIZE = 512 #Number of sequences fed at once in batch
EPOCHS = 13
HIDDEN_LAYERS_DIM = 512
LAYER_COUNT = 4
DROPOUT = 0.2
```

```
text_train_len = len(finalstring)
text_val_len = len(validationstring)
def describe_batch(X, y, samples=3):
 “””Describe in a human-readable format some samples from a
batch. Show the next char given previous char”””
 for i in range(samples):
   sentence = “”
   for s in range(SEQUENCE_LEN):
     sentence += characters[X[i,s,:].argmax()]
   next_char = characters[y[i,:].argmax()]

   print(“sample #%d: …%s -> ‘%s’” % (
      i,
      sentence[-20:],
      next_char
      ))

def batch_generator(text, count):
 “””Generate batches for training”””
 while True:
 for batch_ix in range(count):
   X = np.zeros((BATCH_SIZE, SEQUENCE_LEN, VOCABULARY_SIZE))
   y = np.zeros((BATCH_SIZE, VOCABULARY_SIZE))

   batch_offset = BATCH_SIZE * batch_ix
```

```
   for sample_ix in range(BATCH_SIZE):
    sample_start = batch_offset + sample_ix
    for s in range(SEQUENCE_LEN):
      X[sample_ix, s,
characters_to_ix[text[sample_start+s]]] = 1
    y[sample_ix, characters_to_ix[text[sample_start+s+1]]]=1
```

```
   yield X, y
```

```
for ix, (X,y) in enumerate(batch_generator(finalstring,
count=1)):
 # describe some samples from the first batch
```

```
    describe_batch(X, y, samples=5)
    break
```

Finally, let's define our architecture, consisting of multiple consecutive **Long-Short Term Memory (LSTM) and Dropout Layers** as defined by the LAYER_COUNT parameter. Stacking multiple LSTM layers helps the network to better grasp the complexities of language in the dataset by, as each layer can create a more complex feature representation of the output from the previous layer at each timestep. Dropout layers help prevent overfitting by removing a proportion of active nodes from each layer during training (but not during prediction).

```
 ###From trump char builder
def build_model(gpu_count=1):
 "”"Build a Keras sequential model for training the char-
rnn"”"
 model = Sequential()
 for i in range(LAYER_COUNT):
   model.add(
   LSTM(
   HIDDEN_LAYERS_DIM,
   return_sequences=True if (i!=(LAYER_COUNT-1)) else False,
   input_shape=(SEQUENCE_LEN, VOCABULARY_SIZE),
   )
   )
   model.add(Dropout(DROPOUT))

 model.add(Dense(VOCABULARY_SIZE))
 model.add(Activation('softmax'))

 #removed the multigpu line

 model.compile(loss='categorical_crossentropy',
optimizer="adam")
 return model


training_model = build_model(gpu_count=N_GPU)
```

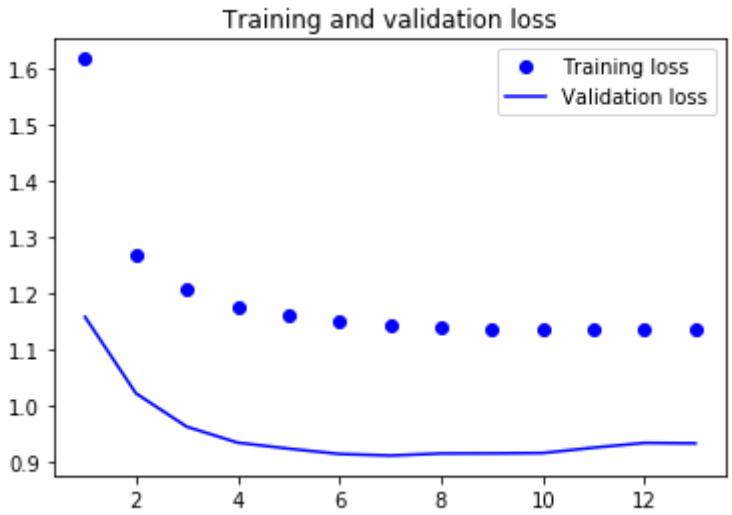
```
train_batch_count = (text_train_len — SEQUENCE_LEN) //
BATCH_SIZE
val_batch_count = (text_val_len — SEQUENCE_LEN) //
BATCH_SIZE
print("training batch count: %d" % train_batch_count)
print("validation batch count: %d" % val_batch_count)
```

With that finished, let's train our network for across 13 epochs and save our network for future use. As our dataset is relatively inconsistent owing to the high number of different users, traditional parameters for measuring progress such as accuracy or loss are only indicative for us,

but a plot of loss over epochs is shown below for the sake of completeness.

```
history = training_model.fit_generator(
 batch_generator(finalstring, count=train_batch_count),
 train_batch_count,
 max_queue_size=1, # no more than one queued batch in RAM
 epochs=EPOCHS,
 validation_data=batch_generator(validationstring,
count=val_batch_count),
 validation_steps=val_batch_count,
 initial_epoch=0
)
```

```
training_model.save('basic_LSTM_tindergenv1_4layers_30char.h
5')
```



Training and validation loss of our RNN over 13 epochs

With our network trained, let's generate some fake bios using different seed words.

Using a seed phrase of "Im" yields excerpts such as:

- [Im] just chilling. Strictly here for good vibes nothing better. I dont have a driver license.Shoot me your best pick up line 5'1 of the Beach. Dog mom.
- [Im] 5'8 but I love to laugh. I love dogs and other peoples babies. Im not a snacc, Im the full meal. I have a big personality, and an even bigger ass. Honestly just looking for someone interesting intellectual conversations.

Using a seed phrase of "I like" yields excerpts such as:

- [I like] makeup and showing my artistic side. Im funny and goofy I love cooking Im not into the hook up culture as much as people my age.
- [I like] to travel and go on adventures! Im a mom. I love going to concerts and trying new things.

Not bad at all, although you can tell that as the predictions continue they start getting weaker, which can be attributed to the network drawing from a multitude of users. Feel free to try other seed phrases in the notebook.

Complete Profiles

Finally, let's wrap up building a couple of complete fake profiles. In the interests of time, we'll paste our results directly onto the three example profiles shown previously. You could easily build a random seed generator to generate a range of ages and names.

Our results look remarkably believable, or should I saw *swipeable*, but some weaknesses remain:

- All of our images are close-range headshots of our subjects. This is due to the nature of our dataset and the requirements of the StyleGAN: training examples to occupy a shared feature space in order to generate realistic images in the same style. In other words, having images of a person jet-skiing alongside portrait photos would lead to unrealistic results.

- Only one image of a specific individual exists. As each seed and style-mix are specific to that particular instance of image generation, it's extremely difficult to guide the StyleGAN output to generate multiple images of the same individual i.e. slightly different angles.
- Similarly, we currently cannot selectively control the age and sex of our profiles. This could be remedied by retraining a StyleGAN from scratch using a custom dataset, which would take a significant amount of resources.
- For our RNN, the elements making up our bios originate from thousands of users, meaning that certain phrases may end up contradicting one another, leading to a loss in profile authenticity.

All in all, this has been a successful showcase on the capabilities of AI in generating believable human representations using freely available resources. A future study could include comparing the differences between real and generated profiles, and whether a neural network would be able to tell the difference. But that's a battle for another day.

If you enjoyed this article, please consider following GradientCrescent for more! Next up, we determine if a neural network could have foreseen the financial crisis.

Sources

Karras et. al, NVIDIA StyleGAN respository

Brownlee, *Text Generation With LSTM Recurrent Neural Networks in Python with Keras*

Testud, *Yet Another Text Generation Project*