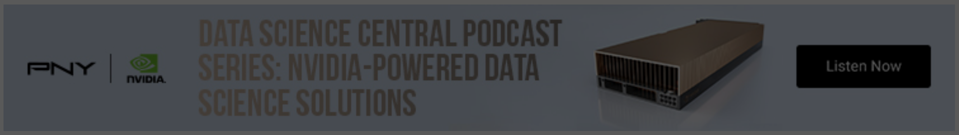

PNY | NVIDIA
DATA SCIENCE CENTRAL PODCAST SERIES: NVIDIA-POWERED DATA SCIENCE SOLUTIONS
Listen Now

Learn how NVIDIA RTX GPUs bring productivity and faster insight to data science and big data analytics.

Submit a blog to KDnuggets -- Top Blogs Win A Reward

Build an app to generate photorealistic faces using TensorFlow and Streamlit

<= Previous post Next post =>

Like 6 Share 6 Tweet Share Share 20

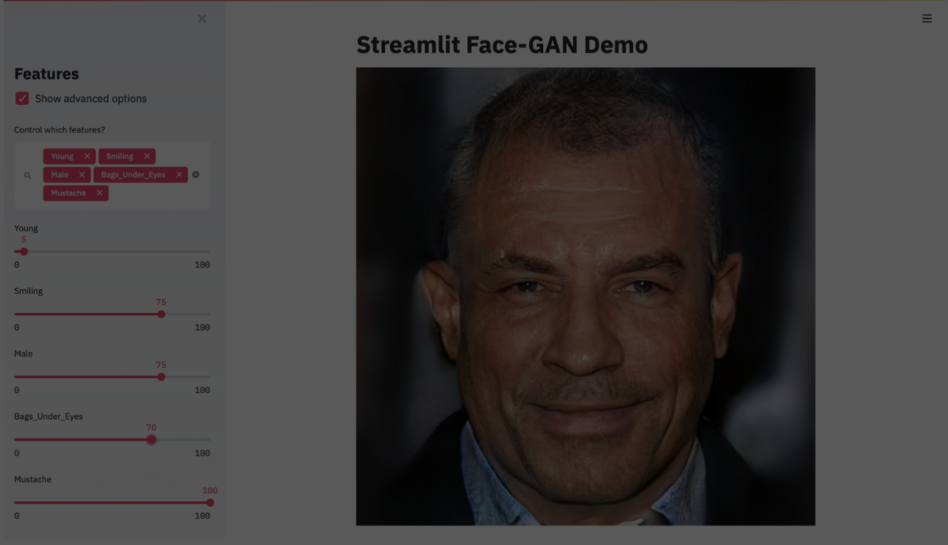
Tags: App, GANs, Generative Adversarial Network, Python, Streamlit, TensorFlow

We'll show you how to quickly build a Streamlit app to synthesize celebrity faces using GANs, Tensorflow, and st.cache.

KNIME Fall Data Talks
Bringing Business and Data Science Together

comments

By Adrien Treuille, Co-founder at Streamlit



[GAN-synthesized face]

Machine learning models are black boxes. Yes, you can run them on test sets and plot fancy performance curves, but it's *still* often hard to answer basic questions about how they perform. A surprisingly powerful source of insight is simply to **play with your models**! Tweak inputs. Watch outputs. Let your coworkers and managers play with them too. This interactive approach is not only a powerful way to gain intuition, but also a great way to get people excited about your work.

Making interactive models is one of the use-cases that inspired Streamlit, a Python framework that makes writing apps as easy as writing Python scripts. This overview will walk you through creating a Streamlit app to play with one of the hairiest and black-box-iest models out there: a deep *Generative Adversarial Network* (GAN). In this case, we'll visualize Nvidia's PG-GAN [1] using TensorFlow to synthesize photorealistic human faces from thin air. Then, using Shaobo Guan's amazing TL-GAN model [2], we'll create an app that gives us the ability to tweak GAN-synthesized celebrity faces by attributes like age, smileyness, male likeness, and hair color. By the end of the tutorial, you'll have a fully parametric model of humans! (Note we didn't create the attributes. They came from the CelebA dataset [3], and some of them can get a bit weird...)

Getting started with Streamlit

Latest News

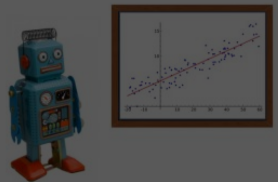
- How to Train a BERT Model From Scratch
- Querying the Most Granular Demographics Dataset
- Introduction to Statistical Learning Second Edition
- MLOPs And Machine Learning RoadMap
- 3 mindset changes to become a better analyst
- How to Detect and Overcome Model Drift in MLOps

20 SHARES

Top Stories Last Week

Most Popular

1. 3 Reasons Why You Should Use Linear Regression Models Instead of Neural Networks



2. Most Common Data Science Interview Questions and Answers
3. How Visualization is Transforming Exploratory Data Analysis
4. GitHub Copilot Open Source Alternatives
5. How To Become A Freelance Data Scientist – 4 Practical Tips

Most Shared

1. Bootstrap a Modern Data Stack in 5 minutes with Terraform
2. GPU-Powered Data Science (NOT Deep Learning) with RAPIDS
3. 3 Reasons Why You Should Use Linear Regression Models Instead of Neural Networks
4. How Visualization is Transforming Exploratory Data Analysis
5. Top Stories, Jul 26 – Aug 1: GitHub Copilot Open Source Alternatives; Why and how should you learn "Productive Data Science"?

More Recent Stories

And if you're a seasoned Streamlit-er, you'll need to be on 0.57.1 or later, so make sure to upgrade!

```
pip install --upgrade streamlit
```

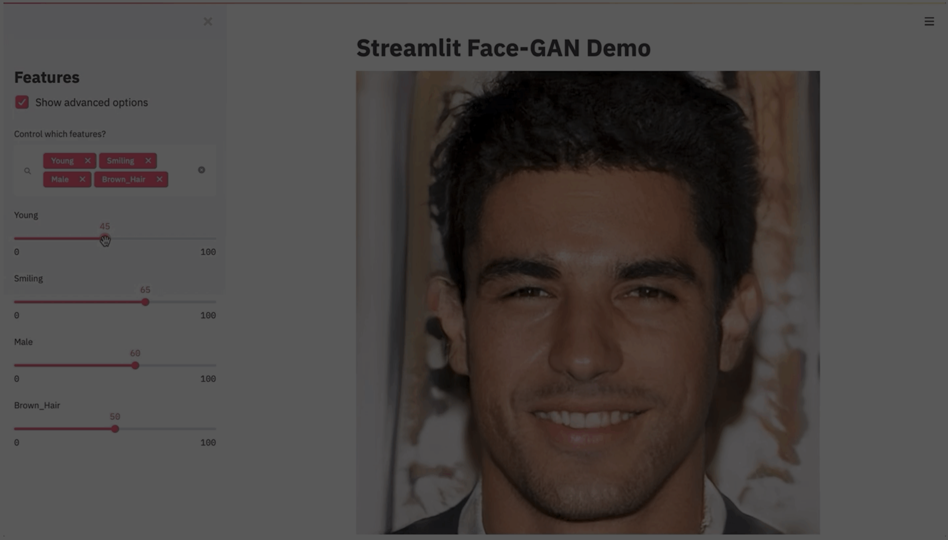
Setting up your environment

Before we begin, use the commands below to check out the project's GitHub repo and running the Face GAN demo for yourself. This demo depends on Tensorflow 1, which does not support Python 3.7 or 3.8, so you'll need Python 3.6. On Mac and Linux, we recommend using pyenv to install Python 3.6 alongside your current version, then setting up a new virtual environment using venv or virtualenv. On Windows, the Anaconda Navigator allows you to pick your Python version with a point-and-click interface.

When you're all set, open a terminal window and type:

```
git clone https://github.com/streamlit/demo-face-gan.git
cd demo-face-gan
pip install -r requirements.txt
streamlit run app.py
```

Give it a minute to finish downloading the trained GAN and then try playing with the sliders to explore the different faces the GAN can synthesize. Pretty cool, right?



The full app code is a file that has ~190 lines of code, out of which only 13 are Streamlit calls. **That's right, the entire UI above is drawn from just those 13 lines!**

Let's take a look at how the app is structured:

```
def main():
    st.title("Streamlit Face-GAN Demo")

    # Step 1. Download models and data files.
    for filename in EXTERNAL_DEPENDENCIES.keys():
        download_file(filename)

    # Step 2. Read in models from the data files.
    tl_gan_model, feature_names = load_tl_gan_model()
    session, pg_gan_model = load_pg_gan_model()

    # Step 3. Draw the sidebar UI.
    ...
    features = ...  # Internally, this uses st.sidebar.slider(), etc.

    # Step 4. Synthesize the image.
    with session.as_default():
        image_out = generate_image(session, pg_gan_model, tl_gan_model,
                features, feature_names)

    # Step 5. Draw the synthesized image.
    st.image(image_out, use_column_width=True)
```

face-gan-main.py hosted with ❤ by GitHub view raw

Now that you have a sense of how it is structured, let's dive into each of the 5 steps above to see how they work.

Learning Survey
The Difference Between Data Scientists and ML Engineers
For SQL, or why I'm so over-protective of my data people
DeepMind's New Super Model: Perceiver IO is a Transformer th...
KDnuggets 21:n30, Aug 11: Most Common Data Science Intervie...
AI in Real Life
How My Learning Path Changed After Becoming a Data Scientist
Practising SQL without your own database
Visualizing Bias-Variance
5 Tips for Writing Clean R Code
Top Stories, Aug 2-8: 3 Reasons Why You Should Use Linear Regr...
Including ModelOps in your AI strategy
How to Query Your Pandas Dataframe
Using Twitter to Understand Pizza Delivery Apprehension During...
Bootstrap a Modern Data Stack in 5 minutes with Terraform **[Gold Blog]**
Essential Math for Data Science: Introduction to Systems of Li...
Be Wary of Automated Feature Selection — Chi Square Test of ...
Most Common Data Science Interview Questions and Answers **[Silver Blog]**
Artificial Intelligence vs Machine Learning in Cybersecurity

20 SHARES

The `download_file` utility function is a little smarter than a pure downloader:

- It checks if the file is already present in the local directory, so it only downloads it if needed. It also checks if the downloaded file's size is what we expected it to be, so it's able to fix interrupted downloads. This is a great pattern to follow!

```python
#  If the file exists and has the expected size, return.
if os.path.exists(file_path):
    if "size" not in EXTERNAL_DEPENDENCIES[file_path]:
        return
    elif os.path.getsize(file_path) == EXTERNAL_DEPENDENCIES[file_path]["size"]:
        return
```

face-gan-download_file.py hosted with ❤ by GitHub   view raw

- It uses `st.progress()` and `st.warning()` to show a nice UI to the user while the file downloads. Then it calls `.empty()` on those UI elements to hide them when done.

```python
# Draw UI elements.
weights_warning = st.warning("Downloading %s..." % file_path)
progress_bar = st.progress(0)

with open(file_path, "wb") as output_file:
    with urllib.request.urlopen(...) as response:

        ...

        while True:

            ...  # Save downloaded bytes to file here.

            # Update UI elements.
            weights_warning.warning(
                "Downloading %s... (%6.2f/%6.2f MB)" %
                (file_path, downloaded_size))
            progress_bar.progress(downloaded_ratio)

...

# Clear UI elements when done.
weights_warning.empty()
progress_bar.empty()
```

face-gan-UI.py hosted with ❤ by GitHub   view raw

Step 2. Load the models into memory

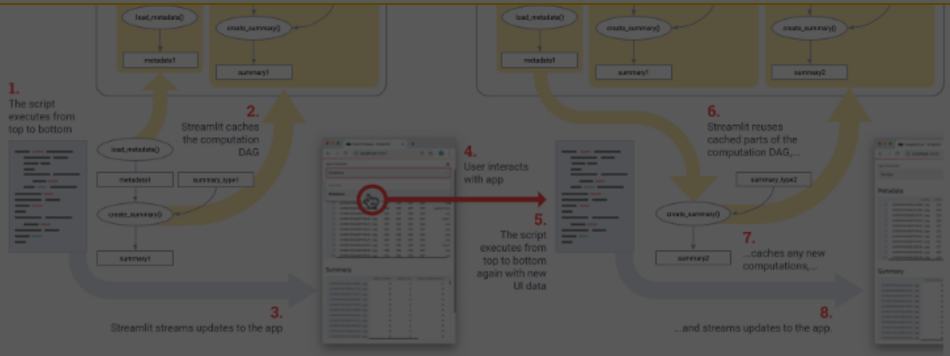
The next step is to load these models into memory. Here is the code for loading the PG-GAN model:

```python
@st.cache(allow_output_mutation=True, hash_funcs=TL_GAN_HASH_FUNCS)
def load_pg_gan_model():
    """
    Create the tensorflow session.
    """
    config = tf.ConfigProto(allow_soft_placement=True)
    session = tf.Session(config=config)

    with session.as_default():
        with open(MODEL_FILE_GPU if USE_GPU else MODEL_FILE_CPU, 'rb') as f:
            G = pickle.load(f)
    return session, G
```

face-gan-load-pg.py hosted with ❤ by GitHub   view raw

Notice the `@st.cache` decorator at the start of `load_pg_gan_model()`. Normally in Python, you could just run `load_pg_gan_model()` and reuse that variable over and over. Streamlit's execution model is unique, however, in that every time a user interacts with a UI widget your script executes again *in its entirety,* from top to bottom. By adding `@st.cache` to the costly model-loading functions, we are telling Streamlit to only run those functions the first time the script executes — and just reuse the cache output for every execution after that. That is one of Streamlit's most fundamental features, as it lets you run scripts efficiently by caching the results of function calls. This way, the large fitted GAN models will be loaded into memory exactly once; and by the same token, our TensorFlow session will be created exactly once, as well. (See our launch article for a refresher on Streamlit's execution model.)


load_metadata()
create_summary()
metadata1
summary1
1. The script executes from top to bottom
2. Streamlit caches the computation DAG
summary_type1
3. Streamlit streams updates to the app
4. User interacts with app
5. The script executes from top to bottom again with new UI data
6. Streamlit reuses cached parts of the computation DAG,...
summary_type2
summary2
7. ...caches any new computations,...
8. ...and streams updates to the app.

[Figure 1. How caching works in Streamlit's execution model]

There's one wrinkle, though: the TensorFlow session object can mutate internally as we use it to run different computations. Ordinarily, we don't want cached objects to mutate, as that can lead to unexpected results. So when Streamlit detects such mutations, it issues a warning to the user. However, in this case we happen to know that it's OK if the TensorFlow session object mutates, so we bypass the warning by setting `allow_output_mutation=True`.

Step 3. Draw the sidebar UI

If this is the first time you're seeing Streamlit's API for drawing widgets, here's the 30-second crash course:

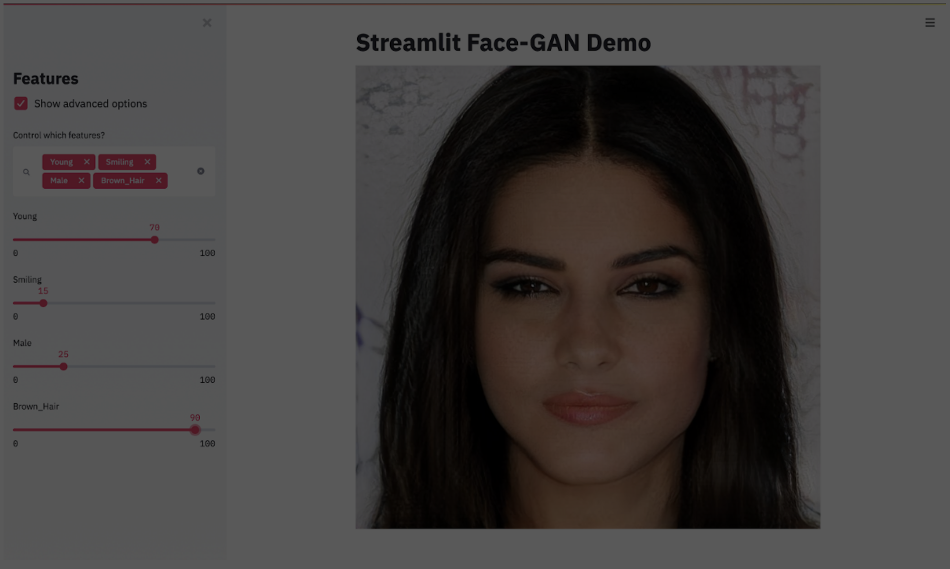
- You add widgets by calling API methods like `st.slider()` and `st.checkbox()`.
- The return value of those methods is the value shown in the UI. For example, when the user moves a slider to position 42, your script will be re-executed and in that execution the return value of that `st.slider()` will be 42.
- You can put anything in a sidebar by prepending it with `st.sidebar`. For example, `st.sidebar.checkbox()`.

So to add a slider in the sidebar, for example — a slider to allow the user to tune the `brown_hair` parameter, you would just add:

```python
brown_hair = st.sidebar.slider("Brown Hair", 0, 100, 50, step=5)
# Translation: Draw a slider from 0 to 100 with steps of size 5.
# Then set the default value to 50.
```

face-gan-sidebar.py hosted with ❤ by GitHub view raw

In our app we want to get a little fancy to show off just how easy it is to make the UI itself modifiable in Streamlit! We want to allows users to first use a multiselect widget to pick a set of features they want to control in the generated image, which means our UI needs to be drawn programmatically:



Streamlit Face-GAN Demo

Features

☑ Show advanced options

Control which features?

Young × Smiling × Male × Brown_Hair ×

Young 70 (0–100)

Smiling 15 (0–100)

Male 25 (0–100)

Brown_Hair 99 (0–100)

With Streamlit, the code for that is actually quite simple:

```python
st.sidebar.title('Features')
```

20 SHARES

```
if st.sidebar.checkbox('Show advanced options'):
    # Randomly initialize feature values.
    features = get_random_features(feature_names, seed)

    # Let the user pick which features to control with sliders.
    control_features = st.sidebar.multiselect(
        'Control which features?',
        sorted(features), default_control_features)

else:
    features = get_random_features(feature_names, seed)

    # Don't let the user pick feature values to control.
    control_features = default_control_features

# Insert user-controlled values from sliders into the feature vector.
for feature in control_features:
    features[feature] = st.sidebar.slider(feature, 0, 100, 50, 5)
```

face-gan-main2-alt.py hosted with ❤ by GitHub view raw

Step 4. Synthesize the image

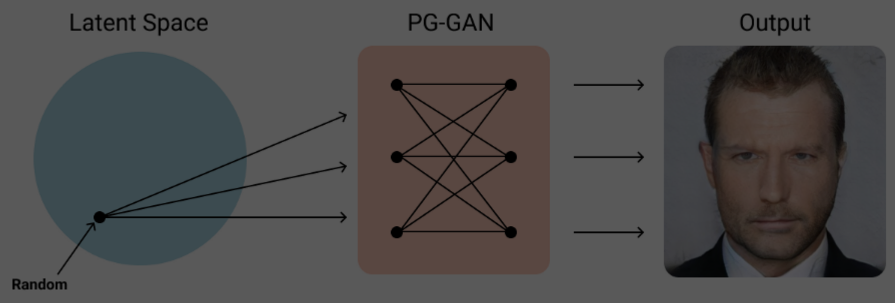
Now that we have a set of features telling us what kind of face to synthesize, we need to do the heavy lifting of synthesizing the face. The way we'll do that is by passing the features into the TL-GAN to generate a vector in the PG-GAN's latent space, then feed that vector to PG-GAN. If that sentence made no sense to you, let's take a detour and talk about how our two neural nets work.

A detour into GANs

To understand how the above app generates faces from slider values, you first have to understand something about how PG-GAN and TL-GAN work — but don't worry, **you can skip this section and still understand how the app works at a higher level!**

PG-GAN, like any GAN, is fundamentally *a pair* of neural networks, one generative and one discriminative, which are trained against each other, forever locked in mortal combat. The generative network is in charge of synthesizing images it believes look like faces, and the discriminative network is in charge of deciding whether or not the images are indeed faces. The two networks are iteratively trained against each other's output, so each one does its best to learn to fool the other network. The end result is the final generative network is able to synthesize realistic-looking faces even though at the start of training all it could synthesize was random noise. Its really quite amazing! In this case, the face-generating GAN we use was trained on celebrity faces by Karras *et al* using their Progressive Growing of GANs algorithm (PG-GAN), which trains GANs using progressively higher-resolution images. [1]

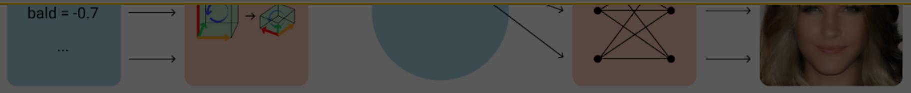
The input to PG-GAN is a high-dimensional vector belonging to its so-called *latent-space*. The latent-space is basically the space of all possible faces the network can generate, so each random vector in that space corresponds to a unique face (or at least it should! Sometimes you get weird results…) The way you typically use a GAN is to give it a random vector and then check out what face gets synthesized (Figure 2.a).



[Figure 2.a]

However, that sounds a bit dull and we'd rather have some more control over the output. We'd like to tell PG-GAN "generate an image of a man with a beard", or "generate an image of a brown-haired woman". That's where the TL-GAN comes in.

The TL-GAN is yet another neural network, this one trained by entering random vectors into PG-GAN, taking the generated faces, and running them through classifiers for attributes like "is young-looking", "is bearded", "is brown-haired", etc. In the training phase, TL-GAN labels thousands of faces from PG-GAN with those classifiers and identifies directions in the latent space that correspond to changes in the labels we care about. As a result, the TL-GAN learns how to map those classes (i.e. "young-looking", "bearded", "brown-haired") into the appropriate random-looking vector that should be input into PG-GAN to generate a face with those characteristics (Figure 2.b).



[Figure 2.b]

Going back to our app, at this point we've already downloaded the pre-trained GAN models and loaded them into memory, and we've also grabbed a feature vector from the UI. So now we just have to feed those features into TL-GAN and then PG-GAN to get an image out:

```python
@st.cache(show_spinner=False, hash_funcs={tf.Session: id})
def generate_image(session, pg_gan_model, tl_gan_model, features, feature_names):

    # Create rescaled feature vector.
    feature_values = np.array([features[name] for name in feature_names])
    feature_values = (feature_values - 50) / 250

    # Multiply by Shaobo's matrix to get the latent variables.
    latents = np.dot(tl_gan_model, feature_values)
    latents = latents.reshape(1, -1)
    dummies = np.zeros([1] + pg_gan_model.input_shapes[1][1:])

    # Feed the latent vector to the GAN in TensorFlow.
    with session.as_default():
        images = pg_gan_model.run(latents, dummies)

    # Rescale and reorient the GAN's output to make an image.
    images = np.clip(np.rint((images + 1.0) / 2.0 * 255.0),
                          0.0, 255.0).astype(np.uint8)  # [-1,1] => [0,255]

    if USE_GPU:
        images = images.transpose(0, 2, 3, 1)  # NCHW => NHWC

    return images[0]
```

face-gan-gen-def.py hosted with ❤ by GitHub view raw

Optimize performance

The `generate_image()` function above can take some time to execute, especially when running on a CPU. To improve our app's performance it would be great if we could cache the output of that function, so we don't have to re-synthesize faces we've already seen as we move the slider back and forth.

Well, as you may have noticed in the snippet above already, the solution here is to once again use the `@st.cache` decorator.

But notice the two arguments we passed to `@st.cache` in this case: `show_spinner=False` and `hash_funcs={tf.Session: id}`. What are those there for?

The first one is easy to explain: by default, `@st.cache` shows a status box in the UI letting you know that a slow-running function is currently executing. We call that a "spinner". However, in this case, we'd like to avoid showing it, so the UI doesn't jump around unexpectedly. So we set `show_spinner` to False.

The next one solves a more involved problem: the TensorFlow session object, which is passed as an argument to `generate_image()`, is usually mutated by TensorFlow's internals in between runs of this cached function. This means the input arguments to `generate_image()` will always be different and we'll never actually get a cache hit. In other words, the `@st.cache` decorator won't actually do anything! How can we solve this?

Hash_funcs to the rescue

The hash_funcs option allows us to specify custom hash functions that tell `@st.cache` how it should interpret different objects when checking whether this is a cache hit or a cache miss. In this case, we're going to use that option to tell Streamlit to hash a TensorFlow session by calling Python's `id()` function rather than by examining its contents:

```python
@st.cache(..., hash_funcs={tf.Session: id})
def generate_image(session, ...):
   ...
```

face-gan-hash-funcs.py hosted with ❤ by GitHub view raw

This works for us because the session object in our case is actually a singleton across all executions of the underlying code since it comes from the @st.cache'd `load_pg_gan_model()` function.

For more information about `hash_funcs`, check out our documentation about advanced caching techniques.

20 SHARES

```
st.image(image_out, use_column_width=True)
```

And we're done!

Wrapping up

So there you have it: interactive face synthesis with TensorFlow in a 190-line Streamlit app and only 13 Streamlit function calls! Have fun exploring the space of faces these two GANs can draw, and many thanks to Nvidia and Shaobo Guan for letting us build off their super cool demos. We hope that you have as much fun building apps and playing with models as we do. 🎈

For more Streamlit app examples, you can check out our gallery at https://www.streamlit.io/gallery.

Thanks to Ash Blum, TC Ricks, Amanda Kelly, Thiago Teixeira, Jonathan Rhone and Tim Conkling for their helpful input on this article.

References:
[1] T. Karras, T. Aila, S. Laine, J. Lehtinen. *Progressive Growing of GANs for Improved Quality, Stability, and Variation.* International Conference on Learning Representations (ICLR 2018)
[2] S. Guan. *Controlled image synthesis and editing using a novel TL-GAN model.* Insight Data Science Blog (2018)
[3] Z. Liu, P. Luo, X. Wang, X. Tang. *Deep Learning Face Attributes in the Wild.* International Conference on Computer Vision (ICCV 2015)

Bio: Adrien Treuille is co-founder of Streamlit, the ML tooling framework. Adrien was a computer science prof at Carnegie Mellon, lead a Google X project, and was VP at Zoox.

Original. Reposted with permission.

Related:

- Generate Realistic Human Face using GAN
- 12-Hour Machine Learning Challenge: Build & deploy an app with Streamlit and DevOps tools
- The Forgotten Algorithm

<= Previous post Next post =>

Top Stories Past 30 Days

Most Popular
1. Top 6 Data Science Online Courses in 2021
2. Data Scientists and ML Engineers Are Luxury Employees
3. Advice for Learning Data Science from Google's Director of Research
4. GitHub Copilot Open Source Alternatives
5. Geometric foundations of Deep Learning

Most Shared
1. Why and how should you learn "Productive Data Science"?
2. Not Only for Deep Learning: How GPUs Accelerate Data Science & Data Analytics
3. Bootstrap a Modern Data Stack in 5 minutes with Terraform
4. GPU-Powered Data Science (NOT Deep Learning) with RAPIDS
5. Become an Analytics Engineer in 90 Days