@IT
atmarkIT

Stable Diffusion 入門

かわさきしんじ , Deep Insider 編集部 [著]

誰もが知っておくべき画像生成 AI「Stable Diffusion」の仕組みと使い方

Stable Diffusion の概要と基本的な仕組み、それを簡単に使うための公式な Web サービスである「DreamStudio」を紹介し、Stable Diffusion で画像生成する際に行われていることについて駆け足で見ていきましょう。

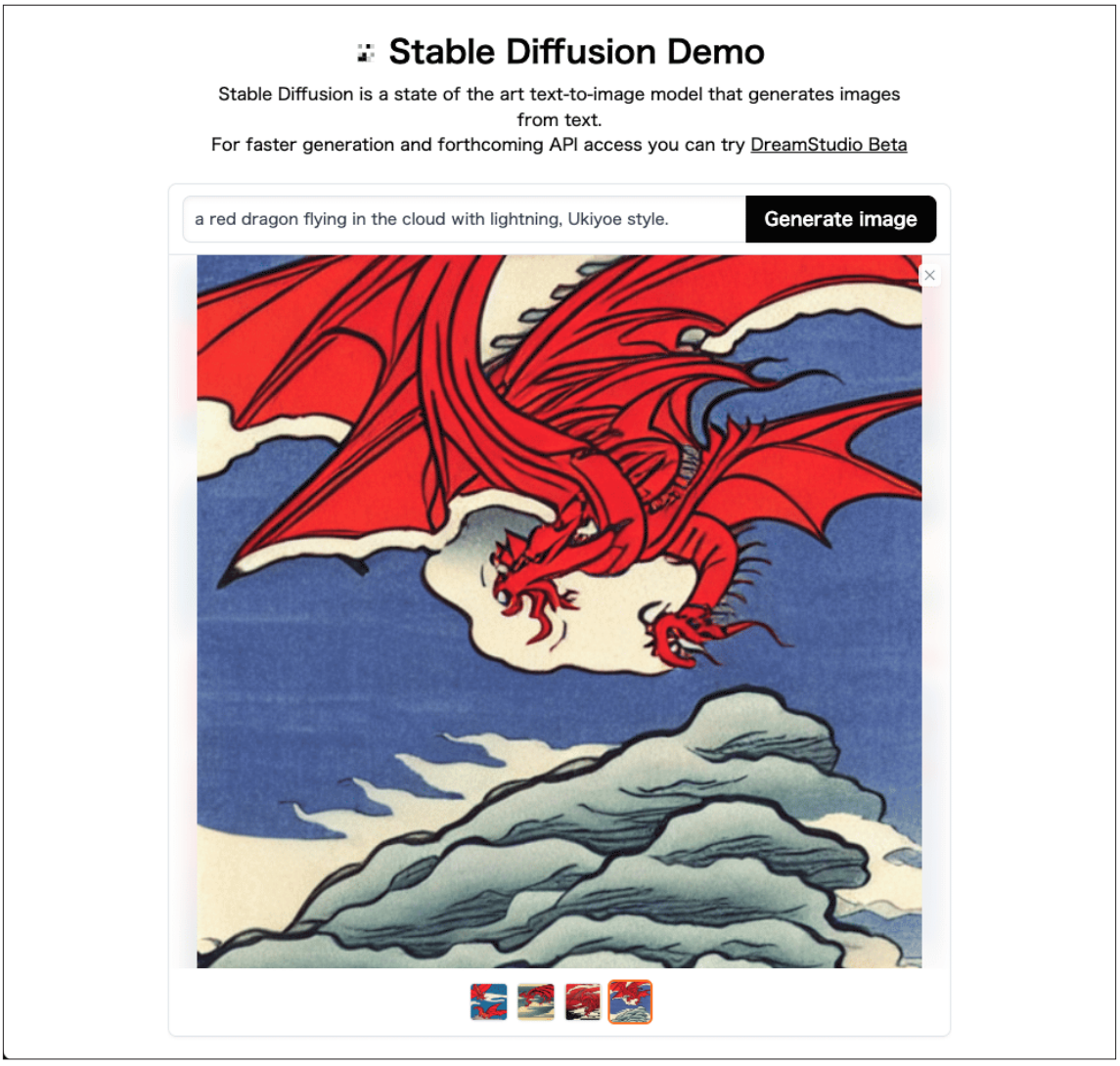
かわさきしんじ，Deep Insider 編集部（2022 年 09 月 16 日）

今、画像生成 AI が「革命」と言えるほど盛り上がっているのをご存じでしょうか。2022 年 8 月前後 から DALL・E 2（ダリ・ツー）や Imagen（イマジェン）、それからもちろん Midjourney（ミッドジャーニー）など、多数の画像生成系 AI が登場し、世の中を騒然とさせていました。が、それらを一足飛びで追い越して多くの人が熱中しているのが Stable Diffusion（ステーブルディフュージョン）です。最初は SNS で大きな話題となりましたが、2022 年 9 月ではテレビで紹介されるまでになっています。

特に Stable Diffusion は、「誰もが必ず知っておくべき」重要な AI だと、Deep Insider 編集部では考えており、この「Stable Diffusion 入門」連載を企画しました。本連載では、AI ／機械学習エンジニアやデータサイエンティストや、最新の話題が好きな人ではなく、「今、画像生成 AI がはやっているみたいだけど、何ができて、何がすごいのかよく分からない」という普通の人に向けて、Stable Diffusion の概要と基本的な仕組み、それを簡単に使うためのサービスなどをできるだけ分かりやすくコンパクトに紹介していきます。一部の説明では、Python というプログラミング言語を使ったコードによる説明も行っていますが、分からない部分はスキップして読んでも大丈夫です。

Stable Diffusion って?

これらの画像生成系 AI に共通するのは、プロンプトとか呪文とか呼ばれるテキストを入力することで、高解像度のグラフィックが生成されることです。このプロンプトをどう工夫すれば、生成される画像をよりよいもの（ユーザーが望む理想に近いもの）をできるかに興味を持つ人も多く、「プロンプトエンジニアリング」と呼ばれる分野があっという間に確立されてもいます。呪文の詠唱の仕方を変えることで、その効果をビジュアルに確認できることが人々のハック魂に火を付けたともいえるでしょう。



Stable Diffusion のデモページで生成した画像の例

その一方で、上に挙げたような高精度な画像生成 AI が多数登場したことで「イラストレーターの仕事はなくなってしまうのか」とか「AI が生成した画像の著作権はどうなってんの」とか「自分の絵を勝手に学習に使われたくない」などの観点から議論が巻き起こってもいます。特に Stable Diffusion はソースコードが公開されていることから画像生成 AI とユーザーとの距離がグッと近づいた結果、こうした問題も多くの人にとって身近なものになったといえるでしょう。実際、Stable Diffusion のソースコードが公開された後は、これをベースとした多くのサービスやアプリがこれでもかというほど、一気に登場していますよね。

が、こうした点については本連載では取り上げずに、あくまでも「Stable Diffusion とは?」「試してみるには?」「コードを書いてみたい」といった観点で話をしていくことにします（かわさき）。

私は「Stable Diffusion 登場前と後で世の中は変わってしまった」と考えています。それほど Stable Diffusion の登場は AI の歴史の中で重要な出来事で、だからこそ「誰もが Stable Diffusion のことを知っておくべきだ」と考えています。世の中には既に、Stable Diffusion の概要や、画像生成例、呪文の書き方などを書いたブログ記事が毎日のように多数出てきています。この連載は、そういったバラバラの情報が 1 本の連載の中でまとめられており、誰もが基礎から学べるもの、いわゆる教科書みたいなものになればよいなと思っています。そういうまとまったものが必要ですよね?

なお、「変わってしまった」とそこまで強く主張する理由は、Midjourney まではソースコードが非公開でしたが、Stable Diffusion はオープンソースだからです。しかも Stable Diffusion の方が高性能という話もあります。オープンソースということは、誰もが手元で使えるようになったということです。私の目には、Stable Diffusion を基点にさまざまな企業や人がこの技術を使って何かしらのサービスを作ったりしてさらに発展していくように映りました。覆水盆に返らず。始まった大きなうねりはもう止められない、というのが主張の理由です（一色）。

最初に述べておきたいことは、Stable Diffusion とは「画像を生成するための訓練済みのモデル」であるということです。極端なことをいえば、モデルを使う上ではその詳細な理解は（とてもありがたいことに）必要ないともいえます。Web サイトが提供しているサービスを利用したり、ローカル PC や Google Colab 上の環境にモデルをインストールしてスクリプトを実行したりプログラムコードを記述／実行したりすることで、多くの人が比較的簡単にその恩恵にあずかれるということです。

使う分には知っておく必要はありませんが、中の仕組み、アーキテクチャーについても簡単に触れておきます。その後で使い方について紹介します。

今も述べたように Stable Diffusion は訓練済みモデルなのですが、このモデルは「潜在拡散（latent diffusion）」と呼ばれるアルゴリズムを実装したものです。ちなみに「潜在拡散モデル」はその前身ともいえる「拡散モデル（diffusion model）」と呼ばれるモデルをより効率的にしたものと考えられます（ちなみに最初に述べた AI の多くも拡散モデル／潜在拡散モデルの系譜に連なるものです）。

「潜在拡散」など、仕組みの話は何やら難しそうな単語が出てきますね。「難しくて理解できない」と思ったらスキップして大丈夫だと思います。後で出てくる VAE というのは機械学習のモデルアーキテクチャーの一つで専門用語になります。

拡散モデルと潜在拡散モデルはどちらも基本的な考え方は同じです。つまり、「純粋なノイズから少しずつノイズを取り除いていくことで、最終的に何らかの画像を得る」というものです。

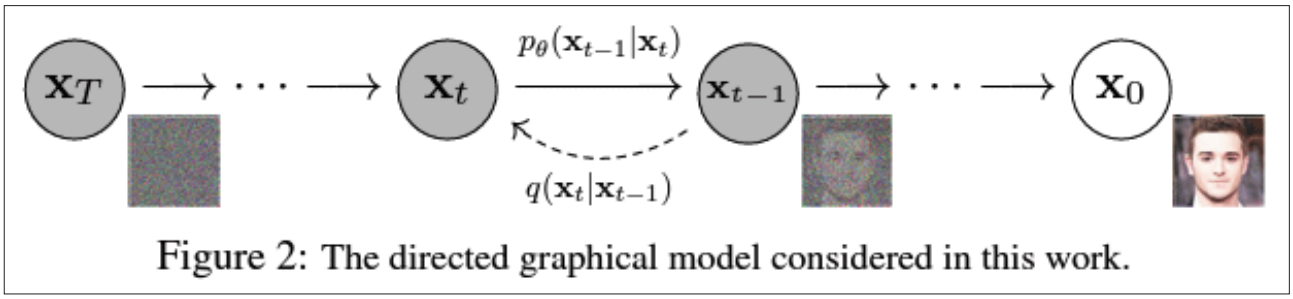


Figure 2: The directed graphical model considered in this work.

拡散モデル／潜在拡散モデルの基本的な考え方
Denoising Diffusion Probabilistic Models より引用。

これを行うには、モデルは実際の画像に少しずつノイズを付加しながら最終的に純粋なノイズ（ガウシアンノイズ）を得る過程（順方向の拡散プロセス）を基に、その逆に純粋なノイズから少しずつノイズを除去しながら最終的に元の画像を得る過程（逆方向の拡散プロセス）において、どんなノイズを純粋なノイズから除去していけば画像を得られるか（画像を生成できるか）を学習する必要があります。つまり、拡散モデル／潜在拡散モデルはこの除去されるノイズを推測し、初期値となるガウシアンノイズ（とは元の画像にノイズを適用して最後に得られたガウシアンノイズでもあります）から、逆方向の拡散プロセスの各ステップで推測したノイズを徐々に除去していき、最終的に何らかの画像を生成するものです。

拡散モデルではピクセル空間を直接操作して、この処理を実現していましたが、これは多くの計算資源を必要として、訓練や推測にも時間がかかるという弱点がありました。そこで、潜在拡散モデルでは VAE でおなじみの潜在空間を用いることでこの弱点を克服し、より効率的に（訓練と）推測を行えるようにしたものといえます。

なぜ「(訓練と)」がかっこ付きかといえば、Stable Diffusion は訓練済みのモデルなので、モデルを利用するだけであれば訓練の部分はあまり関係ないからです。

というのが、Stable Diffusion の概要となるのですが、先ほどもいったように使う分にはこんなことは知っている必要はないといえばないのです。そこで、次にこれを実際に試してみることにしましょう。

Stable Diffusion を試してみたい

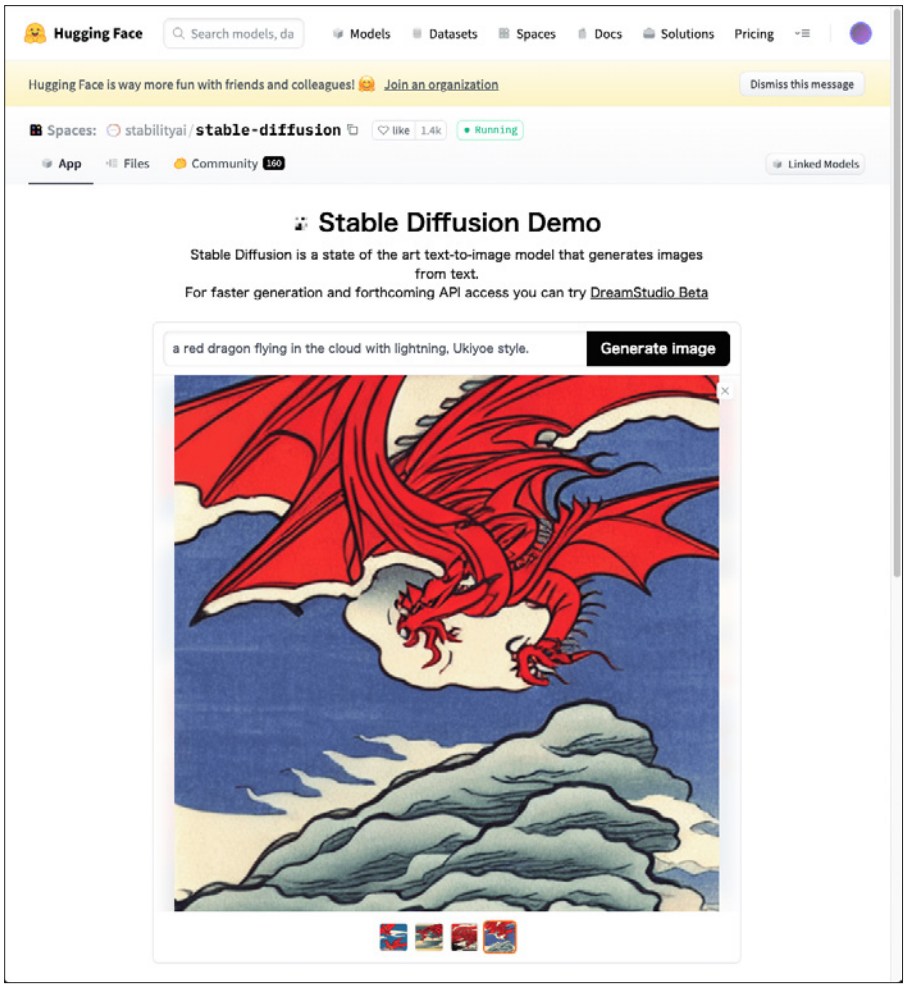
Stable Diffusion を実際に試してみたいという場合、大きく分けて以下の 2 つのやり方があります。

- 多数の Web サイトやアプリケーションがサービスを提供しているのでそれらを使用する
- 手元の PC やクラウド上のノートブック環境（Google Colab など）で Stable Diffusion をインストールしてコマンドラインでプロンプトを入力したり、プログラムコードを書いたりする

前者の方法としては以下のようなものがあります（後者については次回に試してみることにしましょう）。他にも同様なサービスやアプリは多数あるので、これらはあくまでも代表的なものでしかありません。

- Hugging Face という AI コミュニティーサイトで提供されている Stable Diffusion のデモページ
- DreamStudio

Hugging Face で提供されている Stable Diffusion のデモページは以下のようにとてもシンプルで、テキストボックスにプロンプトを入力して［Generate Image］ボタンをクリックするだけで画像が生成されます。

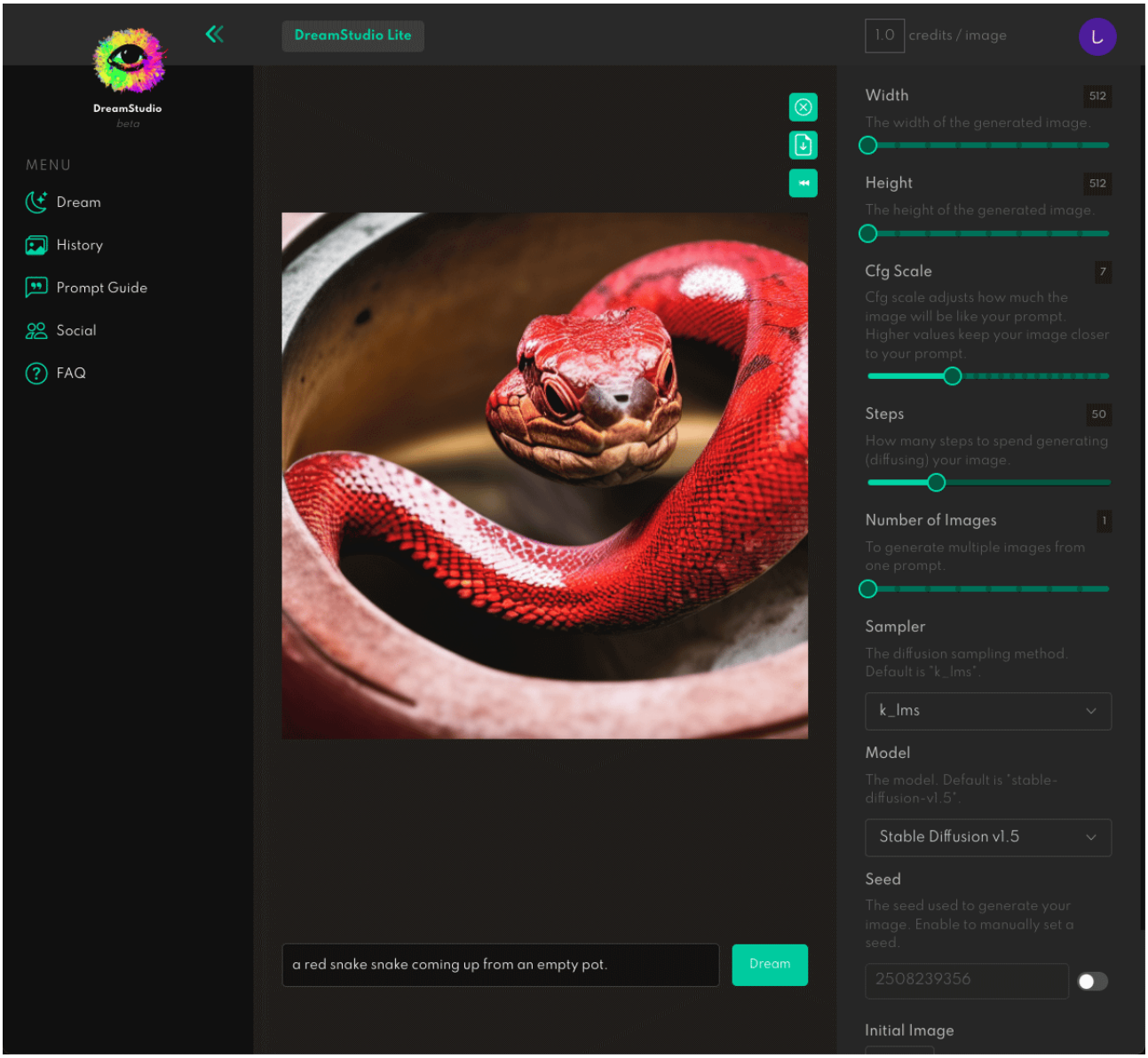


Hugging Face で提供されている Stable Diffusion のデモページ

→目次に戻る

ただし、画像が生成されるまでの待ち時間が長くなることもありますし、画像の大きさや最終的な画像を生成するまでに上で見た逆方向の拡散処理を行うステップ数の指定などのオプションを できないなど、使い勝手についてはあまりよいとはいえません。

画像生成にかかる時間を短くしたい、さまざまなパラメーターを変更してみたいといった場合には、公式な Web サービスである「DreamStudio」がよいでしょう。2022 年 9 月 14 日の時点ではこのサイトはあくまでもベータ版の状態であることには注意してください。また、Stable Diffusion と DreamStudio で生成された画像の著作権は CC0 1.0 Universal Public Domain Dedication とのことです。

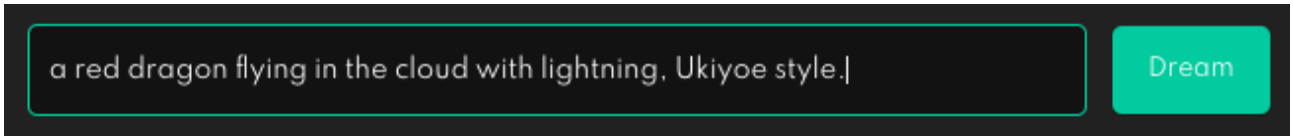


DreamStudio でポットからヘビが出てくる画像を生成したところ

こちらは登録した時点で 200 単位（単位は「generation」）のポイントが割り当てられています。そのポイントを使い果たすと、有償でメンバーシップを購入可能です。1 枚の画像を生成するのに必要なコストは、その大きさとステップ数で決まっていて、512×512 ピクセルで 50 ステップを指定した場合のコストが 1 となっています（つまり、200 generations のポイントがあれば今述べたサイズとステップ数の画像を 200 枚生成できるということでしょう）。

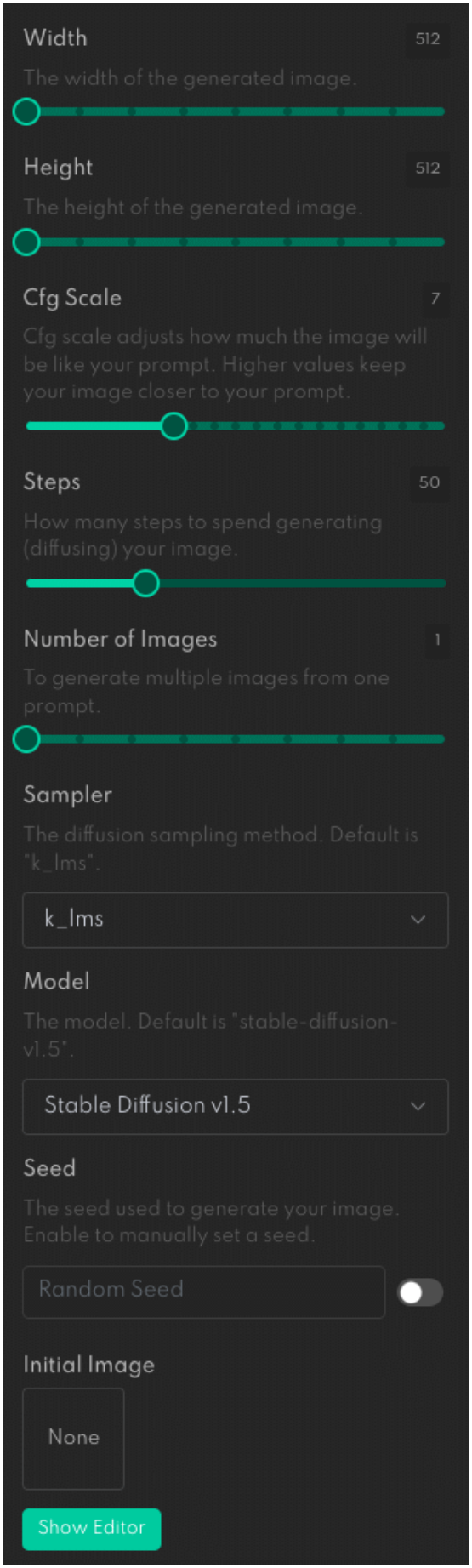
コストについては DreamStudio の FAQ ページで確認できます。

基本的な使い方としては画面下部にあるテキストボックスにプロンプト（どんな絵を生成したいかを記述したテキスト）を入力して、［Dream］ボタンをクリックするだけです。



プロンプトを入力するテキストボックスと［Dream］ボタン

ここに入力するテキストが生成される画像のだいたいの方向性を条件付けます。それをカスタマイズしていくのが、ウィンドウ右側の部分です。プロンプトと各種の項目を変更しながら［Dream］ボタンをクリックすることで、生成される画像がどんどん変わっていくのはとても楽しいです。



オプションを指定するフィールド

→目次に戻る

ここでは以下に示すような項目を指定できます。

- ［Width］：画像の幅
- ［Height］：画像の高さ
- ［Cfg Scale］：プロンプトに対する忠実度の指定
- ［Steps］：画像生成に何ステップを費やすか
- ［Number of Images］：1 つのプロンプトから何枚の画像を生成するか
- ［Sampler］：使用するサンプラー（後述）
- ［Model］：使用する Stable Diffusion のモデルのバージョン
- ［Seed］：画像生成に使用する乱数の初期値
- ［Initial Image］：image-to-image（画像から画像を生成するモード）で画像を生成する際の初期画像

［Width］と［Height］はその名の通り、生成される画像の幅と高さを指定するものです。［Cfg Scale］については後で少し詳しく説明しましょう。［Steps］は画像を生成する際に、冒頭で述べたノイズ除去の過程を何回繰り返すかを指定します。これが多いほど高精細な画像になりそうですが、多くの場合はデフォルト値の「50」程度で十分でしょう。［Number of Images］は一度に何枚の絵を生成するかの指定です。［Sampler］についても後で取り上げます。［Model］では使用する Stable Diffusion のバージョンを指定できます。［Seed］は乱数の初期値です。デフォルトでは毎回ランダムな値が初期値を使って画像が生成されるのですが、この値を固定すると、（他のパラメーターも同じであれば）常に同じ絵が生成されるようになります。これらのオプションの意味は連載の中でいろいろと調べていくことにしましょう。

［Cfg Scale］の説明としてプロンプトに対する忠実度の指定と書きましたが、この値を大きくするとプロンプトに忠実な絵が生成され、この値を低めに設定すると生成される画像の多様性が大きくなります（忠実度と多様性がトレードオフの関係にあるといえます）。

例えば、以下は［Cfg Scale］として 2 と 7 と 20 を指定したときに「a red dragon flying in the cloud with lightning, Ukiyoe style.」というプロンプトから生成された画像です（乱数のシードは「3361770942」で固定）。

「a red dragon flying in the cloud with lightning, Ukiyoe style.」（稲光のする雲海を飛ぶレッドドラゴン、浮世絵スタイル）というプロンプトから生成された画像

上：［Cfg Scale］＝ 2、中：［Cfg Scale］＝ 7（デフォルト）、下：［Cfg Scale］＝ 20（最大）

→目次に戻る

稲光が全てどこかに消えてしまってはいますが、どれも浮世絵っぽい画像にはなっています。[Cfg Scale] = 2 のときには雲海とレッドドラゴン以外にもいろいろな要素が絵に含まれているようにも見えます。[Cfg Scale] = 7 では雲海なのか別のブラックドラゴンなのかは分かりませんがプロンプトにある程度は忠実な絵ができたように感じられます。[Cfg Scale] = 20 になると忠実度も上がっていそうです。とはいえ、試した限りでは[Cfg Scale]を大きくすればよいというわけでもなく、ある程度の多様性を含めるのがよさそうでした（そういうこともあって、デフォルト値は 7 になっているのでしょう）。

困ったことに上の画像を作成するために「あーでもない。こーでもない」とやっているだけで、初期の 200generations がアッという間になくなってしまって、お金を払うことになってしまったのでした（笑）。少し変更して、結果がパッと出てくると［Dream］ボタンをバンバンクリックしちゃいますね。ガチャとかやらない派なんですが、ハマったら課金が大変なことになりそうです。はやいとこローカル環境を作らないと（汗）。

今の画像生成AIってプロンプトを少しずつ書き換えながら絵を調整するものだと思うから、確かにローカル環境で実行しないとお金が湯水のように流れていきそうですね……。

[Sampler] はどんなアルゴリズムを用いてノイズ除去を行うかを指定するものです。アルゴリズムによっては少ないステップ数でも十分な質の画像を生成できるでしょう（これについては後続の回でアルゴリズムを入れ替えながら画像の生成ができればいいなと考えています）。

どんなプロンプトが効果的 についてかは、Web を検索すれば幾らでも解説記事が見つかるはずです（例えば、Gigazine の「画像生成 AI『Stable Diffusion』『Midjourney』で使える呪文のような文字列にパラメーターを簡単に追加できる『promptoMANIA』の使い方まとめ」や、生成される画像のスタイルの指定に使える語を集めた「100+ Stable Diffusion styles & mediums」などがあります）。でも、DreamStudio の「Prompt Guide」ページにも基本的な要素については説明があるので、最初のうちは参考になるかもしれません。

Stable Diffusion で画像生成を行う際に行われていること

最後に Stable Diffusion で画像生成を行う際にはどんなことが行われているのか、その概要をざっくりと紹介します。ただし、ここでは Hugging Face で公開されている「Stable Diffusion with Diffusers」というドキュメントを基にします（この後で出てくる Diffusers が提供する StableDiffusionPipeline クラスは、Stable Diffusion の txt2img.py ファイルで行われている処理と似たことをラップしているのだと筆者は考えていますが、双方の実装を細かく読み下したわけではないので、そうではない可能性もあります）。

このドキュメントでは、Stable Diffusion を構成する 3 つの主要要素として以下が挙げられています。

- VAE
- U-Net
- テキストエンコーダー

やっぱり中の仕組みの話は難しい単語ばかりですね。いずれも専門用語なので、「分からない」と思ったら、流し読みで大丈夫だと思います。このセクションは Python コードを読み書きする知識が必要になっています。

VAE のうち、エンコーダーは訓練にのみ使われ、次に述べるように、U-Net からの出力を基に画像生成をする際にはデコーダーだけが使われます。

この中で画像（の潜在表現）を生成するのが U-Net です。U-Net は潜在表現を受け取り、そこから逆方向の拡散プロセス（ノイズを除去して画像を生成するプロセス）の何らかの段階で使われるノイズ（の残差の潜在表現）を推測します。これを使って作成した（少しノイズが除去された）画像の潜在表現が次の U-Net への入力になります。逆方向の拡散プロセスが終わって、最終的に手に入れた潜在表現は上記 VAE のデコーダーを使った画像生成に使われます。

U-Net は潜在表現を受け取るだけではなく、テキストエンコーダーを使ってユーザーが入力したテキストを埋め込み表現に変換したものも受け取ります。これにより、先ほどの「a red dragon flying in the cloud with lightning, Ukiyoe style.」のようなテキストにより生成される画像に対する条件付けを行えます。

ここでは例として Hugging Face が提供する Diffusers に含まれている Stable Diffusion を使うための StableDiffusionPipeline クラスのコードを見てみます。このクラスの __call__ メソッドには今述べたような処理を実際に行っている部分があります。抜粋して以下に示します。

```
for i, t in enumerate(self.progress_bar(self.scheduler.timesteps)):

    # ……省略……

    noise_pred = self.unet(latent_model_input, t,
                           encoder_hidden_states=text_embeddings).sample

    # ……省略……

    latents = self.scheduler.step(noise_pred, t, latents,
                                  **extra_step_kwargs).prev_sample


image = self.vae.decode(latents).sample
```

U-Net へ潜在表現を入力し、その出力から次のステップでの U-Net への入力を計算する

このコードには latents と latent_model_input という 2 つの変数が見られますが、これらは基本的には同一のもので潜在表現を表しています。latent_model_input は、U-Net（self.unet）への適切な入力となるように latents に手を加えたものだと考えてください（「classifier free guidance」による手法で画像生成を行うかどうかや、サンプリングによるノイズ除去をどのように行うかを決定する「スケジューラ」と呼ばれるオブジェクトとして何を選択したかなどによって、latents の内容を基に latent_model_input の内容が決まります）。

この for ループでは逆方向の拡散プロセスを行います。その中で、「noise_pred = self.unet(latent_model_input, t, encoder_hidden_states=text_embeddings).sample」のようにして推測したノイズの残差を取得しています。そして、「latents = self.scheduler.step(nlatents = self.scheduler.step(……)」行ではそのノイズを除去した画像（の潜在表現）を計算して、これを次の U-Net への入力としています（上のコードでは 1 行にしていますが、実際には先ほど述べたスケジューラに何を選択したかで呼び出し方が変わっています）。

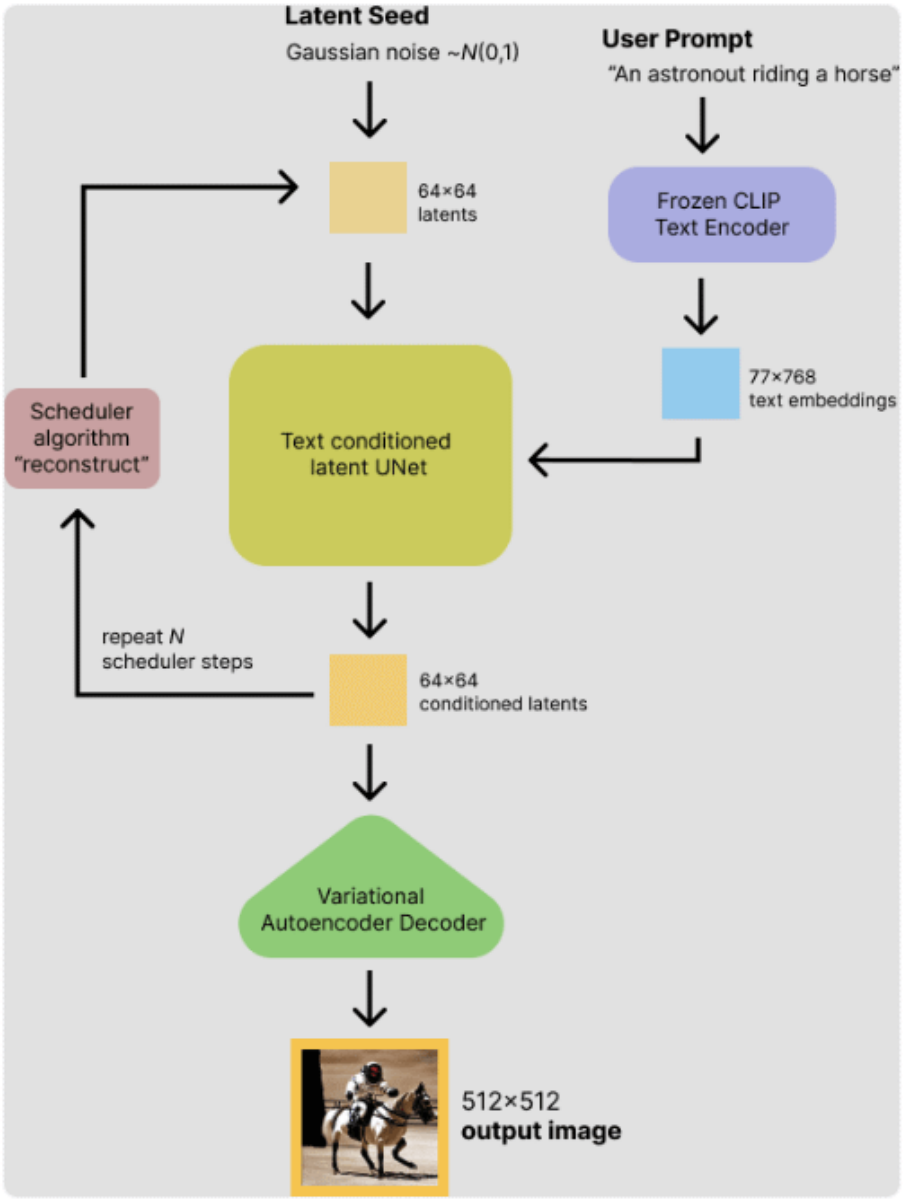
ループが終わった後は、既に述べた通り、VAE のデコーダーを使って潜在表現から実際の画像を生成しています。

U-Net の呼び出しは「noise_pred = self.unet(latent_model_input, t, encoder_hidden_states=text_embeddings).sample」のようになっていますが、このときキーワード引数 encoder_hidden_states に指定しているのが、埋め込み表現に変換されたテキストです。Stable Diffusion では事前に訓練済みの CLIP ViT-L/14 text encoder を使用して入力されたテキストをトークン化し、さらにこれを埋め込み表現に変換したものを U-Net に入力しています。以下は上記と同じソースコード（__call__ メソッド）からこの処理を抜き出したものです。テキストエンコーダーの効果については、本連載の中で試してみる予定です。

```
text_input = self.tokenizer(……)
text_embeddings = self.text_encoder(text_input.input_ids.to(self.device))
[0]

# ……省略……

latents = …… # latents の初期化

# ……省略……

for i, t in enumerate(self.progress_bar(self.scheduler.timesteps)):

    # ……省略……

    noise_pred = self.unet(latent_model_input, t,
                           encoder_hidden_states=text_embeddings).sample

    # ……省略……
```

ノイズを推測するループ

Stable Diffusion が提供する訓練済みモデルを使って画像を生成する際には、今述べたように VAE（のデコーダー）や U-Net、テキストエンコーダー（トークナイザーとエンコーダー）が使われます。これらを図にまとめたものが以下です。



Stable Diffusion で画像を生成する過程
Stable Diffusion with Diffusers より引用。

ユーザーがプロンプトを入力すると、今述べたような処理を経て、画像が生成されるわけです。

ここまでは「Stable Diffusion とは訓練済みのモデル」「DreamStudio を使ってみる」「Stable Diffusion のモデルは VAE、U-Net、テキストエンコーダー、スケジューラなど、さまざまなパーツで構成されている」「それぞれのパーツが関連して画像が生成される」といったことを見てきました。

次章ではローカル PC または Google Colab 上のノートブック環境に Stable Diffusion（と Diffusers）をインストールして、最初の一歩となるコードを書いてみることにします。

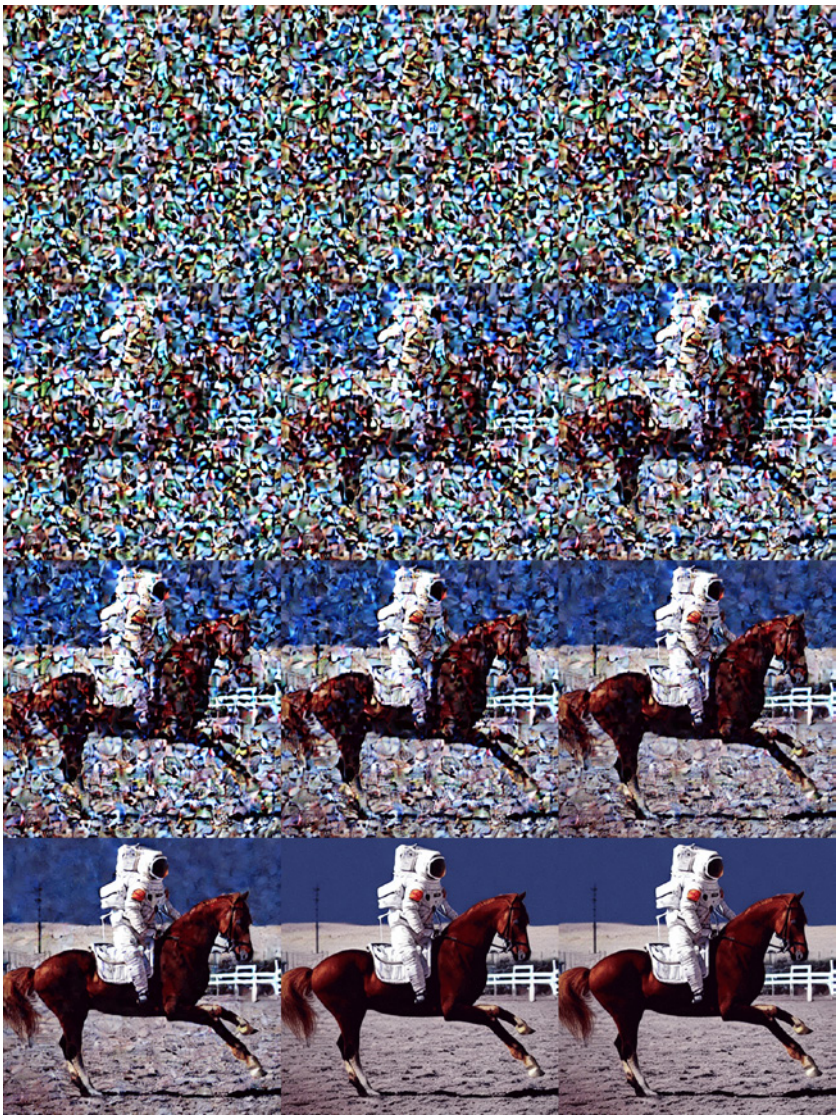
「Stable Diffusion」で
ノイズから画像が生成される過程を確認しよう

ホントにノイズからノイズを除去していくとキレイな画像が生成されるのか。これを今回は自分の目で確認してみましょう。

かわさきしんじ，Deep Insider 編集部（2022 年 09 月 30 日）

前章では Stable Diffusion とは何かと、Web ページでその動作を確認しました。おおざっぱにいえば、Stable Diffusion とは「画像を生成するための訓練済みモデル」で、その動作原理は「純粋なノイズから、徐々にノイズを除去していき、最終的なきれいな画像を得る」というものでした。

というわけで、今回はこれがホントかどうかを目に見える形で確認することにしましょう。最初に答えを見せてしまうと、以下のような画像を得るのが目的です。その過程で Diffusers から Stable Diffusion を使って、画像を生成する基本的なやり方を紹介していきます。



左上から右下に向かってノイズ除去が行われている

→目次に戻る

Windows に Stable Diffusion 環境を構築するか、Google Colab を使用するかは悩みましたが、Google Colab のノートブックを公開しておけば、多くの方が実際にコードを動かして試してみることができるだろうということで、今回は Google Colab を使用しています。なお、無償版の Google Colab では「ノートブックにはハイメモリが必要です」のようなメッセージが表示されたり、実際にメモリ不足で実行できなかったりするかもしれません。その場合は当たりの環境が割り当てられるまでがんばってみてください（かわさき）。

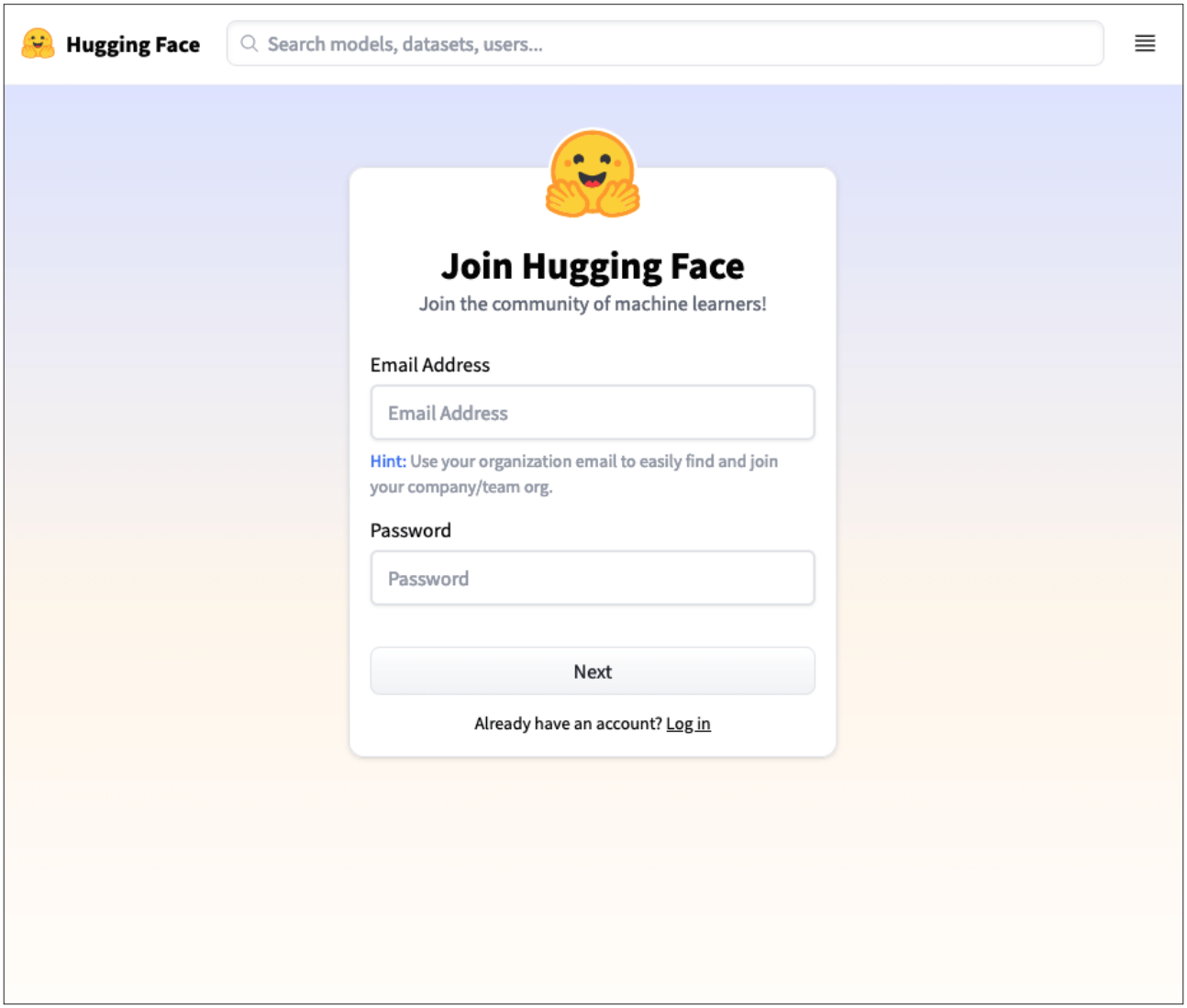
ノートブック内で Python というプログラミング言語が使われていますが、単に実行するだけなので誰にでもできそうですね。コードの意味は以下の本文で分かりやすく説明されているので Python が初めてでも理解できると思います。自分でコードを書けるようになるための記事ではなく単に動かして挙動を確認するための記事なので、難しいと感じたら分からない部分はスキップして概要だけつかんでみるようにしてください。私の場合、手元でコードから Stable Diffusion を実行できるだけでも感動しました（一色）。

Diffusers

Diffusers とは Stable Diffusion をはじめとする「拡散モデル」による画像生成を数行のコードで行えるようにするフレームワークです。例えば、Diffusers には Stable Diffusion を簡単に使えるようにするための StableDiffusionPipeline クラスが含まれていて、これを利用することで確かにほんの数行で画像を生成できるようになります。

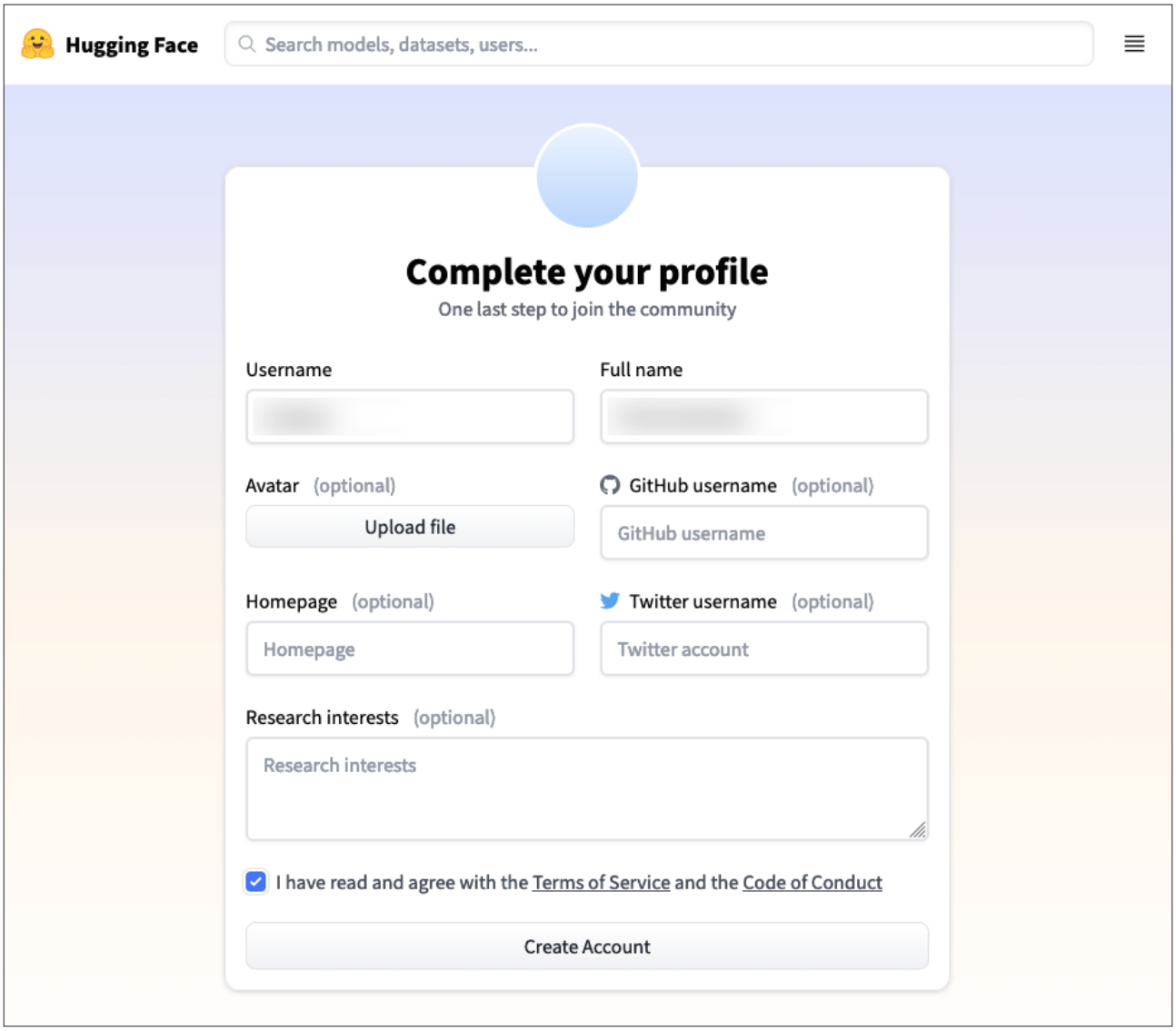
Hugging Face へのサインアップとアクセストークンの取得

とはいえ、その前に Stable Diffusion のモデルを利用するために Hugging Face と呼ばれる AI コミュニティーサイトに登録をして、Stable Diffusion のモデルにアクセスするための「トークン」と呼ばれる文字列を取得する必要があります。上記の URL にアクセスすると、以下のような画面が表示されるので、メールアドレスとパスワードを入力しましょう。



Hugging Face へのサインアップ画面

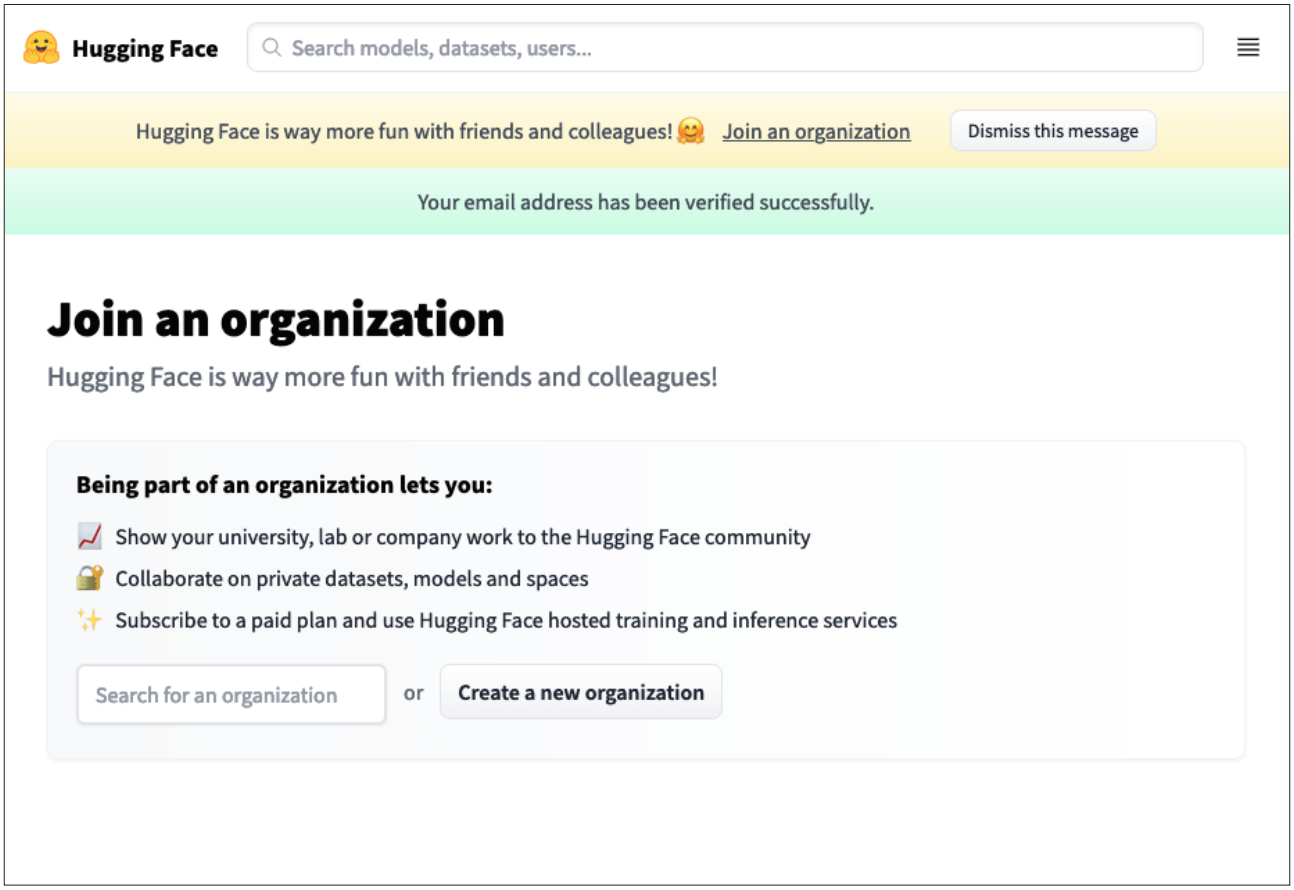
すると、次にユーザー名やフルネームなどのプロフィールを入力する画面が表示されます。



プロフィール入力画面

→目次に戻る

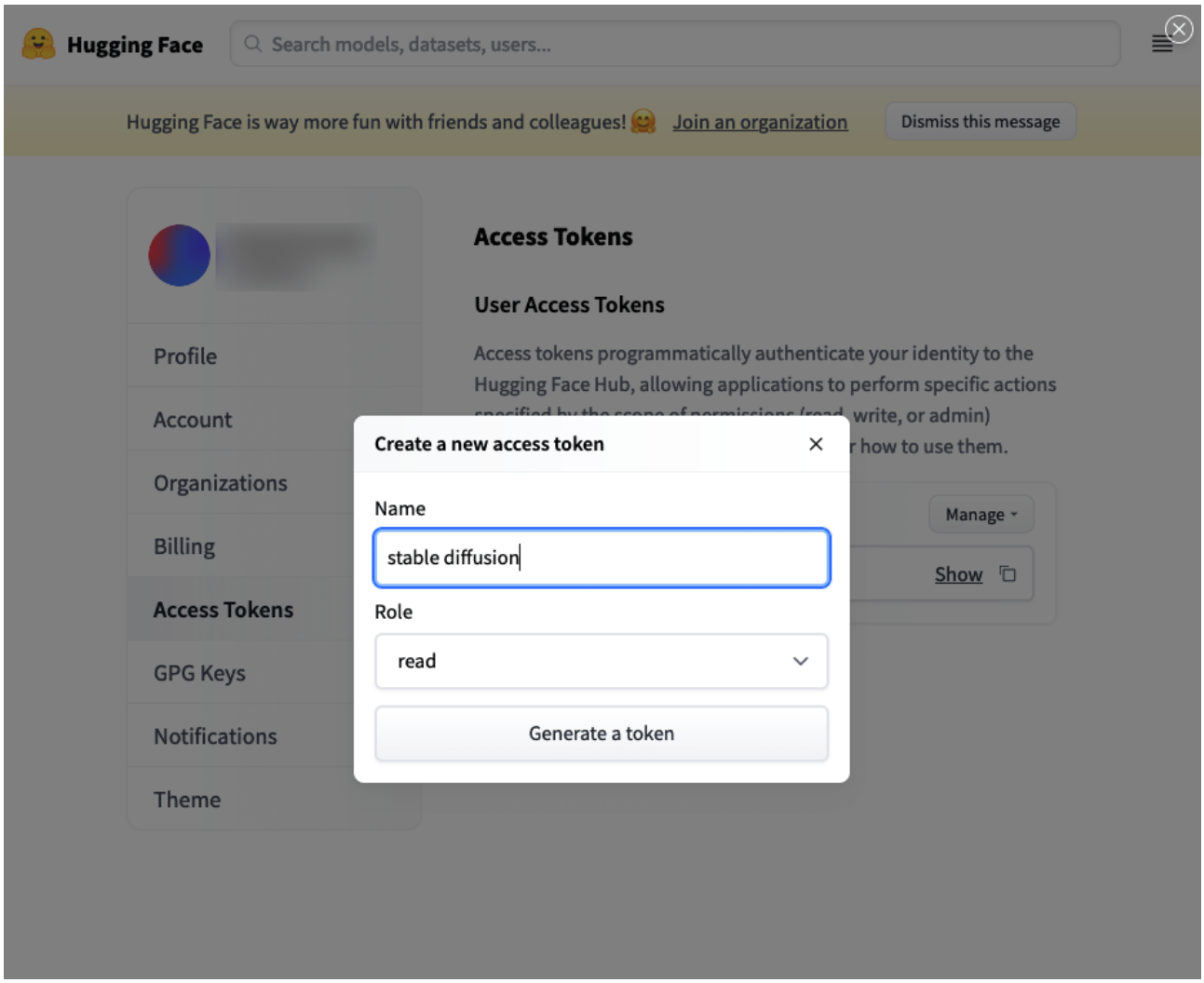
利用規約と行動規範をよく読んで［I have read and agree with the Terms of Service and the Code of Conduct］にチェックをして、［Create Account］ボタンをクリックすると、入力したメールアドレスに確認メールが届くのでリンクをクリックすれば登録は完了です。



登録完了画面

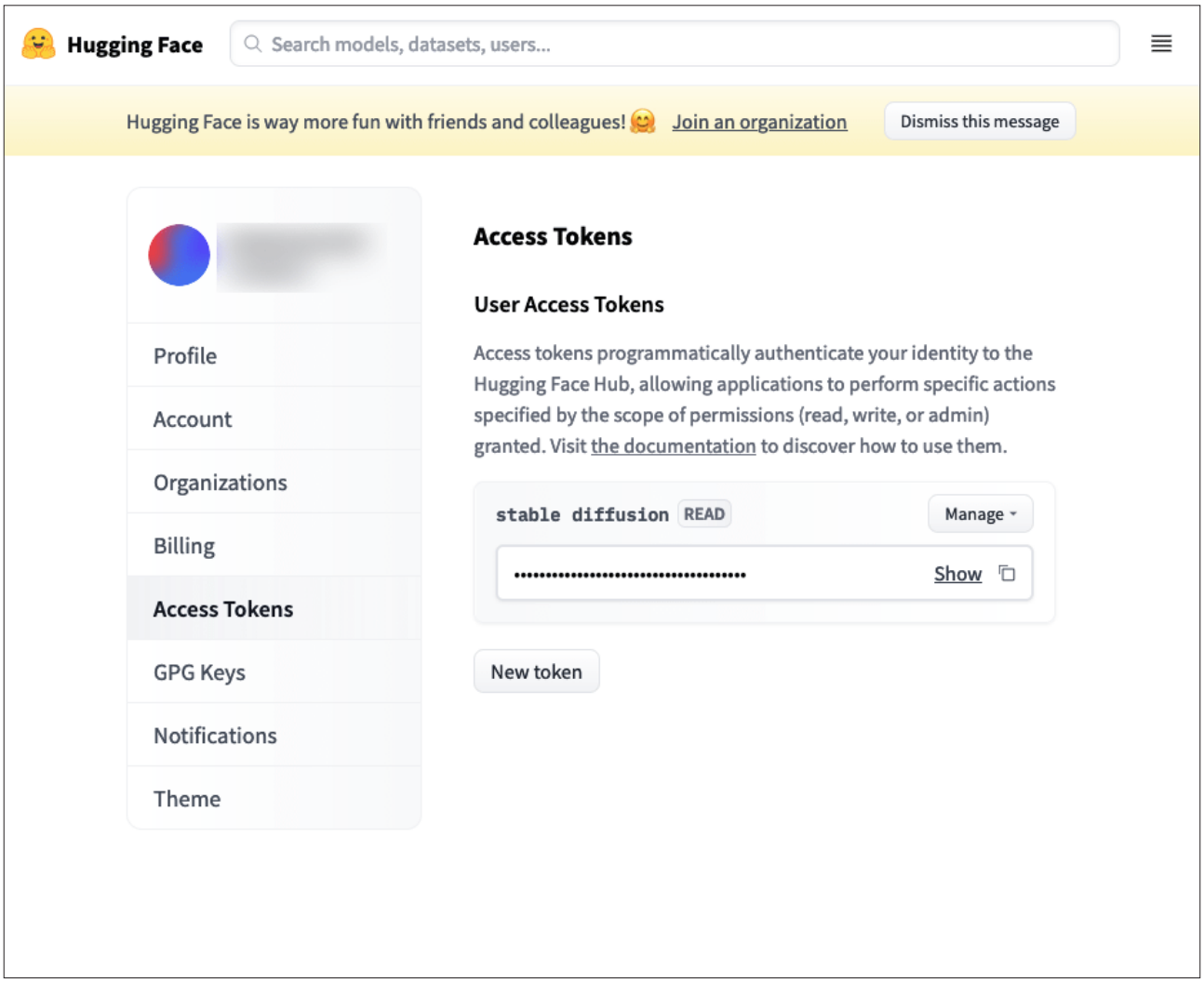
この画面の上にある検索ボックスに「stable diffusion」などと入力すると、その下に「CompVis/stable-diffusion-v1-4」という選択肢が表示されるので、これをクリックします。すると、Stable Diffusion について説明をしたページが表示されます。画面の上部には「このモデルにアクセスするにはコンタクト情報を共有する必要があります」というメッセージが表示されています。この欄を読み進めて、[I have read the License and agree with its terms］にチェックを入れて、［Access Repository］ボタンをクリックします。

次に自身の設定ページに移動して、［New Token］ボタンをクリックしてください。すると、以下のような画面が表示されるので、［Name］欄に「stable diffusion」などと入力して、［Generate a token］ボタンをクリックします。



Stable Diffusion のモデルにアクセスするためのトークンを生成

これにより以下のように生成されたトークンが表示されます。



トークンの一覧

●文字で表示されたトークンの右端にある（[Show] の右隣にある）[Copy token to clipboard] ボタンを押せば、トークンをコピーできます。これを使って、後で Stable Diffusion のモデルにアクセスするようにコードを記述します。

ここまでくれば、Colab のノートブックにコードを記述する準備が完了です。といっても、今回は既に記述済みのノートブックのコードを見ていくことにしましょう。

必要なライブラリのインストール

Diffusers を使うには、通常「pip install diffusers」コマンドを実行するだけです（Google Colab 環境で OS コマンドを実行する際には基本的に先頭に「!」が必要です）。ですが、ここでは GitHub で公開されている最新版（2022 年 9 月時点）を利用することにしました。他にも transformers、scipy、ftfy の各パッケージもインストールしています。

必要なライブラリをインストールするコマンド

Diffusers を使って画像を生成するコードを書いてみよう

ライブラリのインストールができたら、さっそくコードを書いてみます。といっても、以降の内容は「Stable Diffusion with Diffusers」のままです。やることはほんのわずか。

- torch パッケージと StableDiffusionPipeline クラスのインポート
- トークンの設定
- パイプラインの作成
- プロンプトの指定
- 画像生成

以上です。これをコードにしたものが以下です。

```
import torch
from diffusers import StableDiffusionPipeline

YOUR_TOKEN = 'hf_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'

pipe = StableDiffusionPipeline.from_pretrained('CompVis/stable-diffusion-v1-4',
                                               use_auth_token=YOUR_TOKEN)
pipe.to('cuda')

prompt = 'a photograph of an astronaut riding a horse'

with torch.autocast('cuda'):
    image = pipe(prompt).images[0]

image
```

画像を生成するコード

最初の 2 行が torch パッケージと StableDiffusionPipeline クラスのインポートです（実際には後の「with torch.autocast('cuda'):」がなければ torch パッケージのインポートは必要ないかもしれませんね。この with ブロックは計算で使用する浮動小数点数値の精度が異なっている場合に、精度を自動的にそろえてくれるものです）。

変数 YOUR_TOKEN には上で取得したトークンを代入しておきます。

その次の「pipe = StableDiffusionPipeline.from_pretrained('CompVis/stable-diffusion-v1-4', use_auth_token=YOUR_TOKEN)」行でパイプラインと呼ばれる仕組みを作っています。この第 1 引数に指定しているのが、Stable Diffusion のモデル本体です。そして、第 2 引数にはその上で設定しているトークンを指定します。その後の「pipe.to('cuda')」行は、作成したパイプラインを GPU に移すためのものです。

ここまでが画像生成の準備です。

そして、プロンプトとしてよく見られる「a photograph of an astronaut riding a horse」（馬に乗っている宇宙飛行士の写真）を設定しています。最後に「image = pipe(prompt).images[0]」行で画像を生成しています。この行のうち「pipe(prompt)」という作成したパイプラインを関数のように呼び出している部分で画像が生成されます。その次にある「.images[0]」は「pipe(prompt)」呼び出しの戻り値のうち、images 属性にある先頭要素を取り出すものです。つまり、これが Stabled Diffusion により生成された画像となります（with 文は既に述べた通り、浮動小数点数値の精度をそろえるための機構です）。

コード自体は確かにかなり少ない量ですが、ホントにこんな簡単に画像が生成できるのでしょうか。というわけで、以下に実行結果を示します。

生成された画像の表示

→目次に戻る

それらしい画像が確かに生成されました。すばらしいですね。ちなみに上のコードは毎回別々の画像を生成しますが、設定を変えながら、同じ画像を生成するのであれば、前回と同様に乱数の初期値を固定します。コードでこれを行う場合には、torch.Generator クラスの manual_seed メソッドを使用します。

以下はその例です。

```
generator = torch.Generator('cuda').manual_seed(2)
image = pipe(prompt, guidance_scale=7.5, num_inference_steps=50,
             generator=generator).images[0]
image
```

乱数のシードや生成される画像のプロンプトに対する忠実度、ノイズ除去を行う回数などを指定

乱数のシードは generator キーワード引数に指定します。prompt は上と同様、生成される画像の指示で、guidance_scale キーワード引数には前回にも出てきたプロンプトに対する忠実度で（大きいほどプロンプトに忠実になり、小さいほど生成される画像の多様性が大きくなる）、num_inference_steps キーワード引数にはノイズ除去を行う回数を指定します。この他にも画像のサイズも指定できます。

以下はこのコードの実行結果です。

生成された画像

ここまで Diffusers を使うことで、数行のコードで画像生成が行えることを見てきました。StableDiffusionPipeline クラスの __call__ メソッドには、画像生成を行うための処理がまとめられていて、このクラスのインスタンスを関数のように呼び出すことで、このメソッドのコードが実行されるようになっています。興味のある方はソースコードを見てみると、結構な量のコードが書かれていることが分かるはずです。

こうしたコードを事前に定義することで、定型的で煩雑なコードを記述する必要をなくすのが Diffusers のよいところですね。

次に、純粋なノイズからノイズ除去を繰り返していくことで、上の画像がホントに生成されるのかを確認してみましょう。

ノイズが除去されてちゃんとした画像になっていく過程の確認

上では StableDiffusionPipeline クラスを使って画像を生成しました。また、そこではパイプラインを関数のように呼び出すことで、画像生成が行われると述べました。ここでは、StableDiffusionPipeline クラスを継承する MyStableDiffusionPipeline クラスを定義して、そこで画像生成過程の特定の時点で生成された画像を保持しておいて、それを最終的に生成された画像と一緒に返すようなクラスを作ります。

以下コードはかなり長いのですが、その内容を詳細に知る必要はありません。ありませんよ。ありませんからね！

ここでは主に __call__ メソッドをオーバーライド（上書き）していますが、その内容は StableDiffusionPipeline クラスの __call__ メソッドとほぼ同様です。やったことはだいたい次のようなことです。

- 画像生成を行うループ処理中に、特定の時点でそのときに生成された画像を保持するコードを追加
- このときに画像生成を行うコードをインスタンスメソッドとして切り出す

実際のコードを以下に示します。かなりの部分を省略して、変更または切り出した部分を強調書体で表示しています。

```
class MyStableDiffusionPipeline(StableDiffusionPipeline):
    def __init__(
        self, vae=None, text_encoder=None, tokenizer=None, unet=None,
        scheduler=None, safety_checker=None, feature_extractor=None):
           super().__init__(vae, text_encoder, tokenizer, unet, scheduler,
safety_checker, feature_extractor)

    @torch.no_grad()
    def __call__(
        self, prompt, height=512, width=512, num_inference_steps=50,
        guidance_scale=7.5, eta=0.0, generator=None, latents=None,
        output_type='pil', return_dict=True, step=5, **kwargs):

        # ……省略……

        images = []
        for i, t in enumerate(self.progress_bar(self.scheduler.timesteps)):

            # ……省略……

            if i % step == 0:
                image, _ = self.make_image(latents, output_type)
                images.append(image[0])

        image, has_nsfw_concept = self.make_image(latents, output_type)

        if not return_dict:
            return (image, has_nsfw_concept, images)

        return (StableDiffusionPipelineOutput(images=image,
                    nsfw_content_detected=has_nsfw_concept), images)
```

```
    @torch.no_grad()
    def make_image(self, latents, output_type):
        latents = 1 / 0.18215 * latents
        image = self.vae.decode(latents).sample

        image = (image / 2 + 0.5).clamp(0, 1)
        image = image.cpu().permute(0, 2, 3, 1).numpy()

           safety_checker_input = self.feature_extractor(self.numpy_to_
pil(image),
                        return_tensors="pt").to(self.device)
        image, has_nsfw_concept = self.safety_checker(images=image,
                        clip_input=safety_checker_input.pixel_values)

        if output_type == 'pil':
            image = self.numpy_to_pil(image)

        return image, has_nsfw_concept
```

MyStableDiffusionPipeline クラス

make_image メソッドの内容は StableDiffusionPipeline クラスの __call__ メソッドに含まれていたコードを切り出しただけなのであまり説明することはありません。要するにここで、Stable Diffusion が生成した画像の潜在空間に置ける表現を VAE のデコーダーに入力して画像として、やばい画像かどうかのチェックをしているだけだと考えてください。

StableDiffusionPipeline クラスと比べて、__call__ メソッドでは steps というパラメーターが増えています（デフォルト値は 5）。これは、ノイズ除去を行うループで何回ごとに生成された画像を保存しておくかを指定するものです。num_inference_steps の値が 50 であればノイズ除去がおおよそ 50 回行われますが、このときに steps の値が 5 ならループが 5 回実行されるたびにその時点での画像が保持されるようになるということです（実際にはスケジューラーと呼ばれる機構の仕組みによっては num_inference_steps の値が 50 だからといってループも 50 回とはならないこともあるので、これはあくまでもおおよその値だと考えてください）。

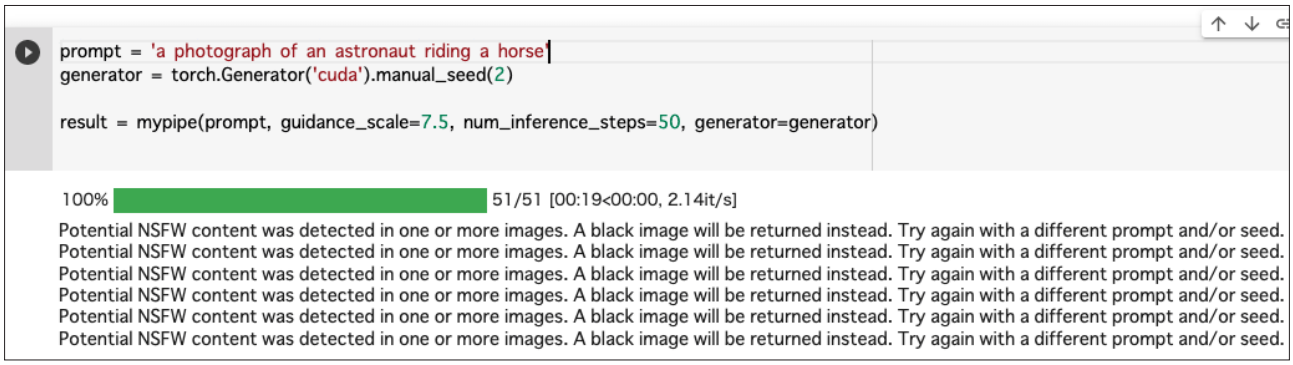
保持された画像はリストにまとめられていて、StableDiffusionPipeline クラスの __call__ メソッドが返していた画像と後述する NSFW フィルターに引っかかったかを知らせる値に加えて、このリストを返送するようになっています。

このクラスを使って画像生成を行うコードは例えば次のようになります。

```
mypipe = MyStableDiffusionPipeline.from_pretrained(
    'CompVis/stable-diffusion-v1-4', use_auth_token=YOUR_TOKEN)
mypipe.to('cuda')

prompt = 'a photograph of an astronaut riding a horse'
generator = torch.Generator('cuda').manual_seed(2)

result = mypipe(prompt, guidance_scale=7.5, num_inference_steps=50,
generator=generator)
```

MyStableDiffusionPipeline クラスを使って画像を生成するコード

クラスが StableDiffusionPipeline から MyStableDiffusionPipeline クラスに変わっただけで、後は上で見たコードと同様ですね。実行結果は以下の通りです。



何やらヘンなメッセージが表示されている

表示されているメッセージは要するに「やばげな画像が生成されたので、代わりに真っ黒な画像を返すよ。今度はプロンプトを変えるか、シードを変えるかして試してみてね」という意味です。Stable Diffusion では NSFW（Not Safe For Work）フィルター機能が標準で組み込まれていて、職場などで閲覧するには問題がある画像が自動的に真っ黒な画像に差し替えられるようになっています。

エッチな画像を生成しているわけではないのですが、これについては後で対処することにしましょう。

エッチな画像を生成する方法というのが話題になっていたけど、これで制限を解除できるのか。ふむふむ。

重要なのは、このパイプラインから返された値の第 1 要素（0 始まり）には画像を含んだリストが格納されているということです。ここでは Stable Diffusion with Diffusers で紹介されている image_grid 関数をそのまま借用させてもらって、画像をグリッド状に並べることにしました。

```
from PIL import Image

def image_grid(imgs, rows, cols):
    assert len(imgs) == rows*cols

    w, h = imgs[0].size
    grid = Image.new('RGB', size=(cols*w, rows*h))
    grid_w, grid_h = grid.size

    for i, img in enumerate(imgs):
        grid.paste(img, box=(i%cols*w, i//cols*h))
    return grid
```

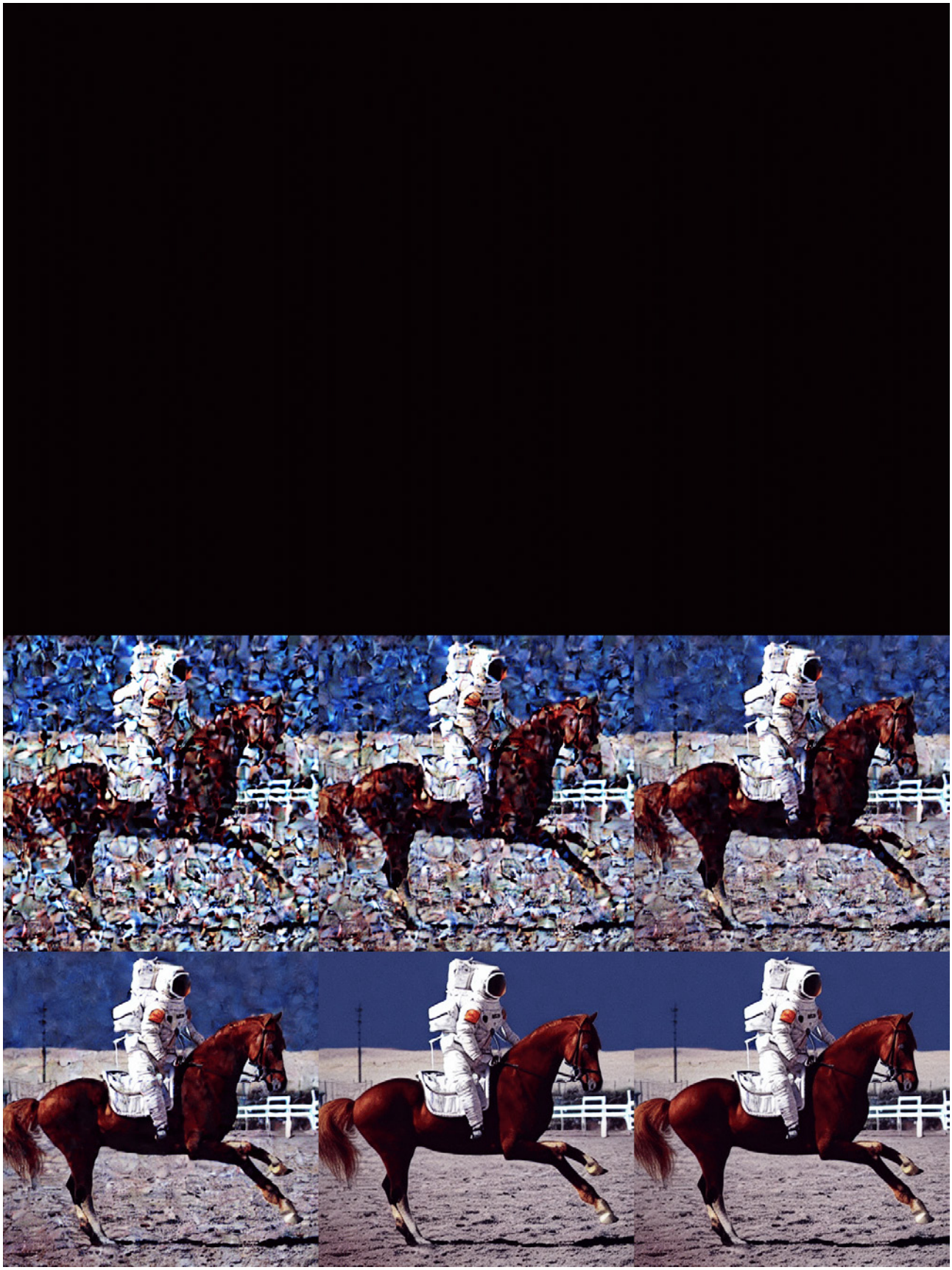
image_grid 関数

image_grid 関数を呼び出す際には、第 1 引数に画像を含むリストを、第 2 引数に画像を並べるグリッドの行数を、第 3 引数に列数を指定します。実はパイプラインが返す画像の数はここでは 11 個だったので、ここでは最後に生成された画像をリストに追加して 12 個の画像を含むリストとして 4 行 3 列のグリッドに画像を並べることにしました。

```
result[1].append(result[0].images[0])
grid = image_grid(result[1], 4, 3)
grid
```

画像をグリッド状に並べて表示

実行結果を見てみましょう。



上の方は真っ黒

→目次に戻る

NSFW フィルターのおかげで最初の 6 枚が真っ黒になっていることが分かりました。が、後半の 6 枚は徐々にノイズが除去されて、だんだんとキレイになっていますね。

というわけで、最後に NSFW フィルターを無効化することにします。ここでは、何もチェックしない関数を定義しました。画像がそのまま帰ってくればよいので、画像と同時に返送する値は True で固定です。

仕様を調べていなかったのですが、False でもいいのかな？

```
def my_checker(self=mypipe, images=None, clip_input=None):
    return images, True
```

何もチェックしない my_checker 関数

これをパイプラインの safety_checker 属性（safety_checker メソッド）にセットすれば OK です。メソッドとするので第 1 パラメーターはパイプラインを参照する self になります。通常、self の値はメソッド呼び出し時に自動的にパイプラインを参照する値がセットされるのですが、筆者が試したところではなぜかこれがうまくいかないときがあったので、上のコードではデフォルト引数値として、MyStableDiffusionPipeline クラスのインスタンスである mypipe を指定しています。

setattr 関数を使って、この関数をパイプラインの safety_checker メソッドにしたら、先ほどと同様に画像を生成してみましょう。

```
setattr(mypipe, 'safety_checker', my_checker)

prompt = 'a photograph of an astronaut riding a horse'
generator = torch.Generator('cuda').manual_seed(2)

result = mypipe(prompt, guidance_scale=7.5, num_inference_steps=50,
                generator=generator)
```

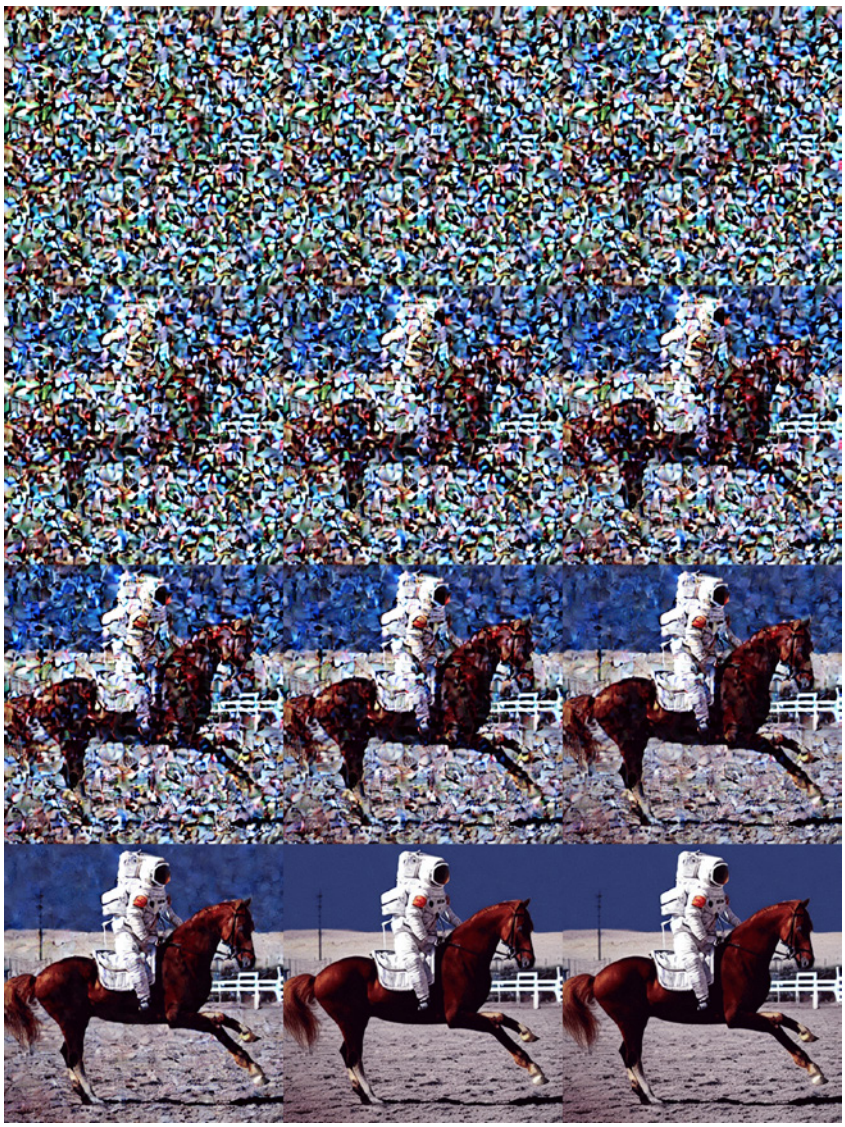
チェックを行わないようにして画像を生成

最後にこれを image_grid 関数でグリッド状に並べます。

```
result[1].append(result[0].images[0])

grid = image_grid(result[1], 4, 3)
grid
```

image_grid 関数で画像を並べる

以下は実行結果です。



ノイズ除去の過程が分かるようになった

どうでしょう。ノイズの中から宇宙飛行士と馬が徐々に現れてくる過程がよく分かるようになりました。

確かに Stable Diffusion では純粋なノイズから徐々にノイズが除去されていき、最終的にキレイな画像が生成されることが確認できました。

「Stable Diffusion」で生成された画像とプロンプトがどのくらい似ているのかを確認してみよう

Stable Diffusion は内部で CLIP と呼ばれるモデルを使用しています。CLIP を使うと何ができるようになるのかを見てみましょう。

かわさきしんじ，Deep Insider 編集部（2022 年 10 月 14 日）

前章では Stable Diffusion でノイズから画像が生成される過程を確認しました。今回は少し方向性を変えて、CLIP（Contrastive Language-Image Pre-Training）と呼ばれ、Stable Diffusion の内部でも使われている「テキストとイメージ（画像）の組を基に学習を行ったモデル」について見てみます。

今回もコードは Google Colab のノートブックとして公開することにしました（かわさき）。

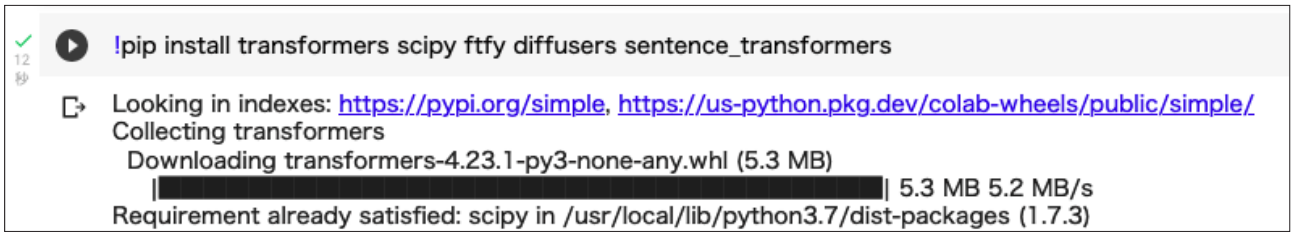
CLIP は、OpenAI という組織が 2021 年 1 月 5 日に「DALL・E: テキストから画像を生成」と同時に発表した技術で、AI エンジニアやデータサイエンティストの間では有名ですね。OpenAI 公式ページでは「CLIP: テキストと画像をつなぐ」というタイトルの記事が公開されています（一色）。

Stable Diffusion（をラップした Diffusers の StableDiffusionPipeline クラス）では、CLIP（CLIP ViT-L/14）を利用してプロンプトを「埋め込み表現」と呼ばれる「単語や文の意味を n 次元空間に埋め込んだベクトル」に変換し、それを潜在表現（latent representation）と一緒に画像生成用のモデル（U-Net）へと入力するようになっています。

txt2img.py ファイルのコードと等価な処理かどうかは、両方のコードの見た目がずいぶんと異なっているのでちょっと確認できていません。ざっくりと見分けるのであれば、同じ設定で txt2img.py ファイルと StableDiffusionPipeline クラスに画像を生成させて、その結果を確認するという方法があるかもしれませんね。

まず CLIP でどんなことができるのかを簡単に確認してみましょう。なお、ここではベクトルのコサイン類似度を簡単に確認できる sentence_transformers が提供している clip-ViT-L-14 モデルを使用することにします。これを利用して生成される埋め込み表現が、StableDiffusionPipeline クラスで実際に生成される埋め込み表現と異なる可能性があることには注意してください。それでも、CLIP にどんなことができるのかは十分に理解できるはずです。

今回も Google Colab のノートブックを使っているので、以下のように先頭に「!」付けて「pip install transformers scipy ftfy diffusers sentence_transformers」コマンドを実行しておきましょう。



必要なライブラリのインストール

埋め込み表現とコサイン類似度

次に CLIP ViT-L/14 モデル（これは恐らく「Vision Transformer を使用した巨大サイズの CLIP、パッチレベル 14」という意味です）を利用したモデルをインスタンス生成します。

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('clip-ViT-L-14')
```

CLIP ViT-L/14 を使用したモデルのインスタンス生成

なお、先頭行では util モジュールをインポートしていますが、このモジュールにはコサイン類似度を計算する cos_sim 関数が含まれています。

モデルを生成したら、さっそく簡単な単語から埋め込み表現を作ってみましょう。

```
dog = 'a dog'

dog_emb = model.encode(dog)

print(dog_emb)
print(dog_emb.shape)
```

埋め込み表現の作成

埋め込み表現を作成するにはモデルの encode メソッドに文字列（テキスト、プロンプト）を渡すだけです。ここでは「a dog」という 2 語を渡しています。最後の 2 行では返された埋め込み表現とそのサイズを表示しています。

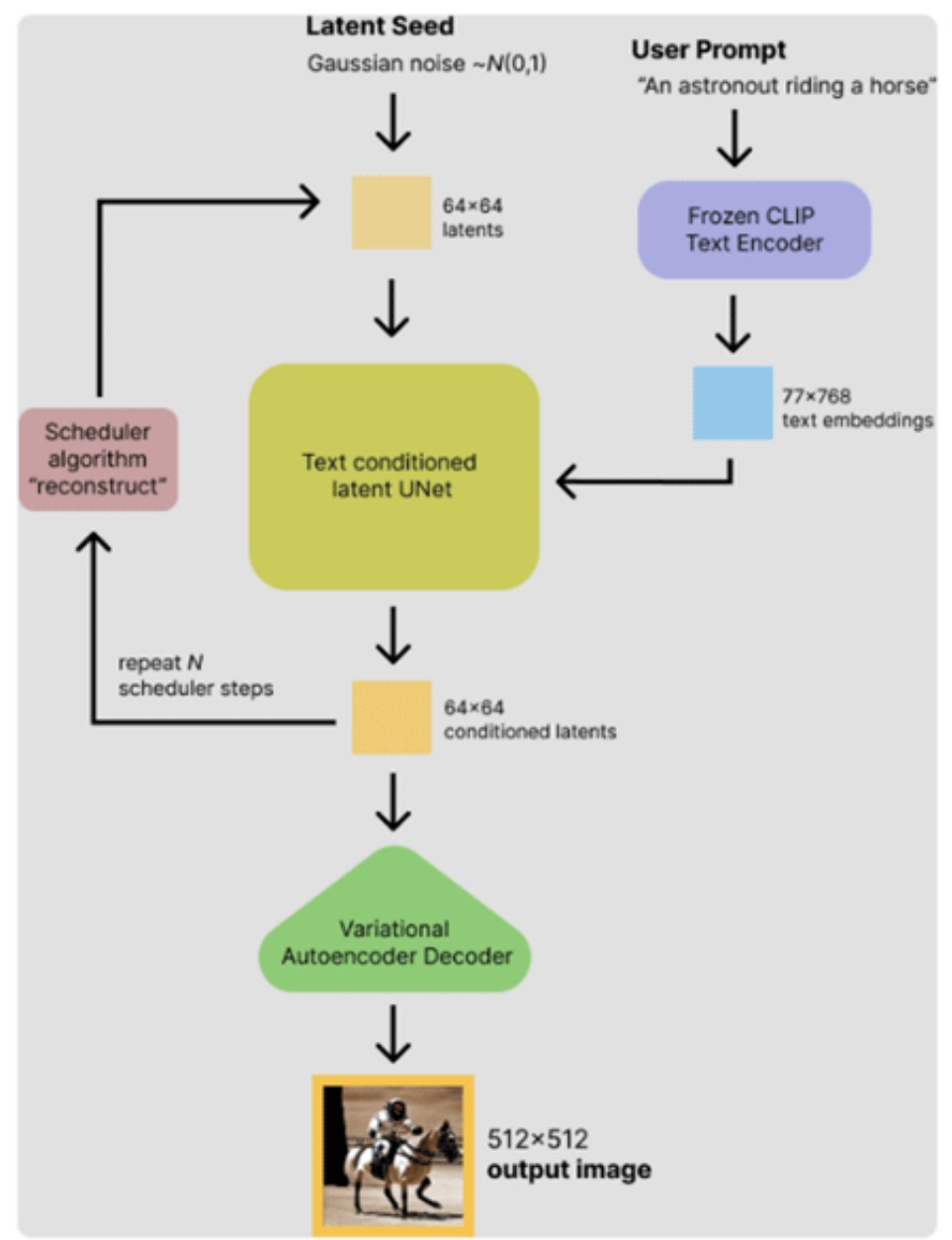
```
[3]  model = SentenceTransformer('clip-ViT-L-14')

     dog = 'a dog'

     dog_emb = model.encode(dog)

     print(dog_emb)
     print(dog_emb.shape)
```

```
 4.67852205e-01 -1.16709568e-01  2.68041581e-01  2.04794630e-02
 8.89909387e-01  9.89129543e-02 -2.62628198e-01 -2.96048462e-01
-2.23013207e-01 -1.81432471e-01  1.64904729e-01  7.15323448e-01
-1.02122426e-02  6.98529184e-03 -7.11440593e-02 -4.83009815e-02

                                 ⋮

 1.27893239e-01  6.50521517e-01  3.58045310e-01 -8.14030111e-01
 2.23832920e-01  3.39320421e-01 -5.91827869e-01 -1.66155398e-03
-1.21581994e-01 -2.58612216e-01  5.83052874e-01  3.06190968e-01
-1.40532136e-01 -1.32945552e-02 -1.39656365e-01 -2.23660290e-01]
(768,)
```

埋め込み表現とそのサイズの表示

数多くの浮動小数点数値が表示され、768 次元（1 行 768 列）であることが分かりました。

「Stable Diffusion with Diffusers」で示されている以下の図を覚えている方がいれば、上の形状（1 行 768 列）が Stable Diffusion が内部で使用している埋め込み表現とは異なっていることに気付くかもしれませんね。



Stable Diffusion の動作原理
Stable Diffusion with Diffusers より引用。

この動作原理ではプロンプトからは 77 行 768 列の埋め込み表現が作られることになっています。しかし、ここでは CLIP によりどんなことができるのかを見るだけなので、このまま話を続けることにしましょう。

ここで作成した埋め込み表現は、「a dog」というテキストがどのような意味を持つかが 768 次元空間のいずれかの座標にマッピングすることで表現されているものと考えられます。また、これと似た意味を持つテキストは 768 次元空間の中で近い位置に存在します。それがどのくらい似ているかを示すのが先ほども出てきた「コサイン類似度」です。

コサイン類似度は 2 つのベクトル（ここでは 768 次元ベクトル）がどれくらい似ているかを示すもので「1 に近いほど 2 つのベクトルはよく似ていて、0 に近いほど 2 つのベクトルは無関係であり、-1 に近いほど 2 つのベクトルはよく似ていない」といったことがいえます。

というわけで、犬の「反対」っぽい猫（a cat）の埋め込み表現を作って、2 つの埋め込み表現がどのくらい似ていないか（または似ているか）を確認してみましょう。

```
cat = 'a cat'
cat_emb = model.encode(cat)
```

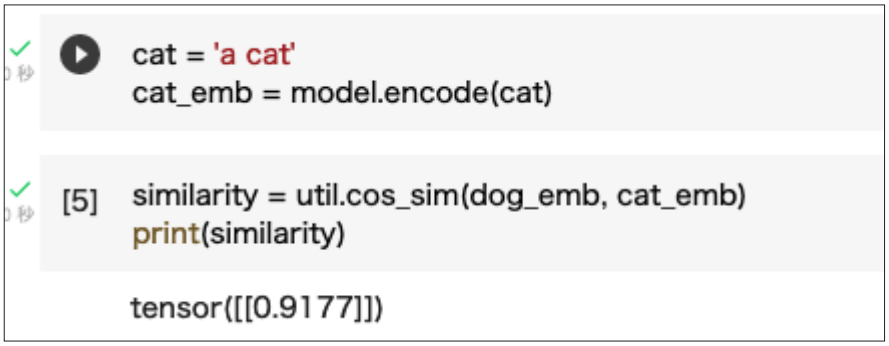
「a cat」の埋め込み表現の作成

後はこれを util モジュールの cos_sim 関数に渡すだけでコサイン類似度を計算できます。

```
similarity = util.cos_sim(dog_emb, cat_emb)
print(similarity)
```

コサイン類似度の計算

結果はどんなものでしょうか。



「a dog」と「a cat」のコサイン類似度は「0.9177」！

何と「a dog」と「a cat」のコサイン類似度は「0.9177」になりました。先ほどもいいましたが、コサイン類似度は 1 に近いほど、2 つのベクトルが似ていることを意味します。CLIP は単に「犬←→猫」という概念だけではなく、哺乳類であるとか、生物であるとか、そんなことまでを 768 次元空間の中で表せるようになっているのかもしれません。そうなると、無関係なもの、あるいは全く似ていないものを考えつくのもなかなか難しいものです。

画像の埋め込み表現

上で見た通り、CLIP はテキストから埋め込み表現を作成することもできますが、画像を基にその意味を表す埋め込み表現も作成できます。そして、2 種類の埋め込み表現を cos_sim 関数に渡して、それらのコサイン類似度を計算することも可能です。

なるほど、CLIP を使えば「テキストの埋め込み表現」と「画像の埋め込み表現」を同じように作成できるのですね。同様の埋め込み表現を使えることこそが OpenAI の公式ページのタイトルにあった「テキストと画像をつなぐ」ための鍵になっている、という理解で合っていますかね。そこで今回の記事では、テキストの埋め込み表現と画像の埋め込みが実際にどれくらい似ているかを幾つかの例で調べてみたというわけですね。

そこで、画像から埋め込み表現を作成してみます。これには PIL モジュールの Image クラスを使うのが簡単です。筆者は紅葉の写真と桜の写真を選んで、Colab にアップロードしています。アップロードするには Colab ノートブックの左側に並んでいる［ファイル］アイコンをクリックして、［セッション ストレージにアップロード］ボタンをクリックした後に、アップロードするファイルを選択するだけです。ただし、セッションストレージにアップロードした画像ファイルはノートブック環境が回収されるときに削除されてしまうので、ノートブックのこの部分を試すときにはご自分で適当な画像をアップロードして名前を修正し、比較対象のテキストも適宜修正する必要があります。

本稿のノートブックには紅葉と桜の 2 枚の画像が含まれているので、それらをダウンロードしてから、［セッション ストレージにアップロード］ボタンを使ってアップロードし直すのが簡単かもしれません。

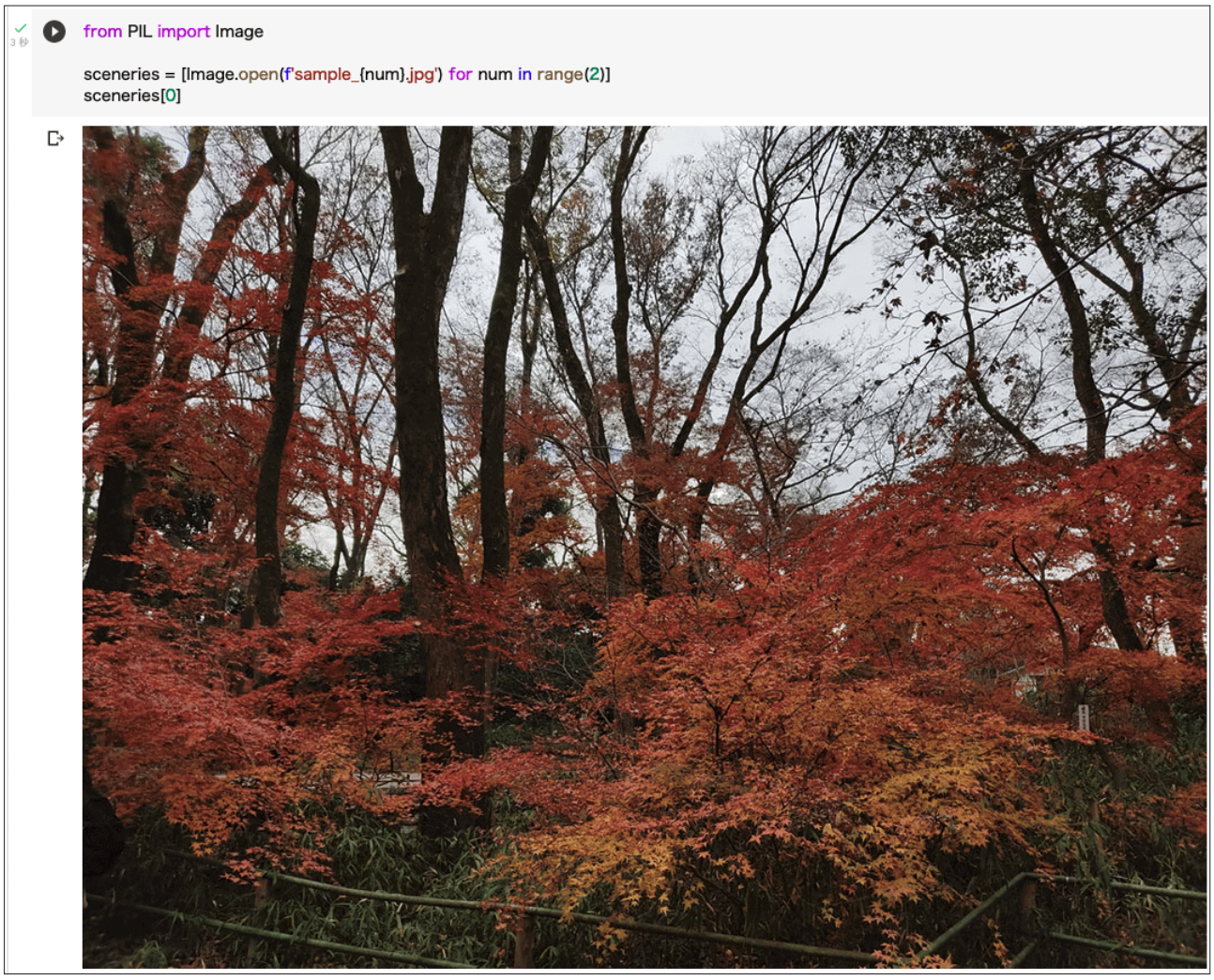
ファイルのアップロード

アップロードができたら、まずは写真ファイルを読み込みます。

```
from PIL import Image

sceneries = [Image.open(f'sample_{num}.jpg') for num in range(2)]
sceneries[0]

sceneries[1]
```

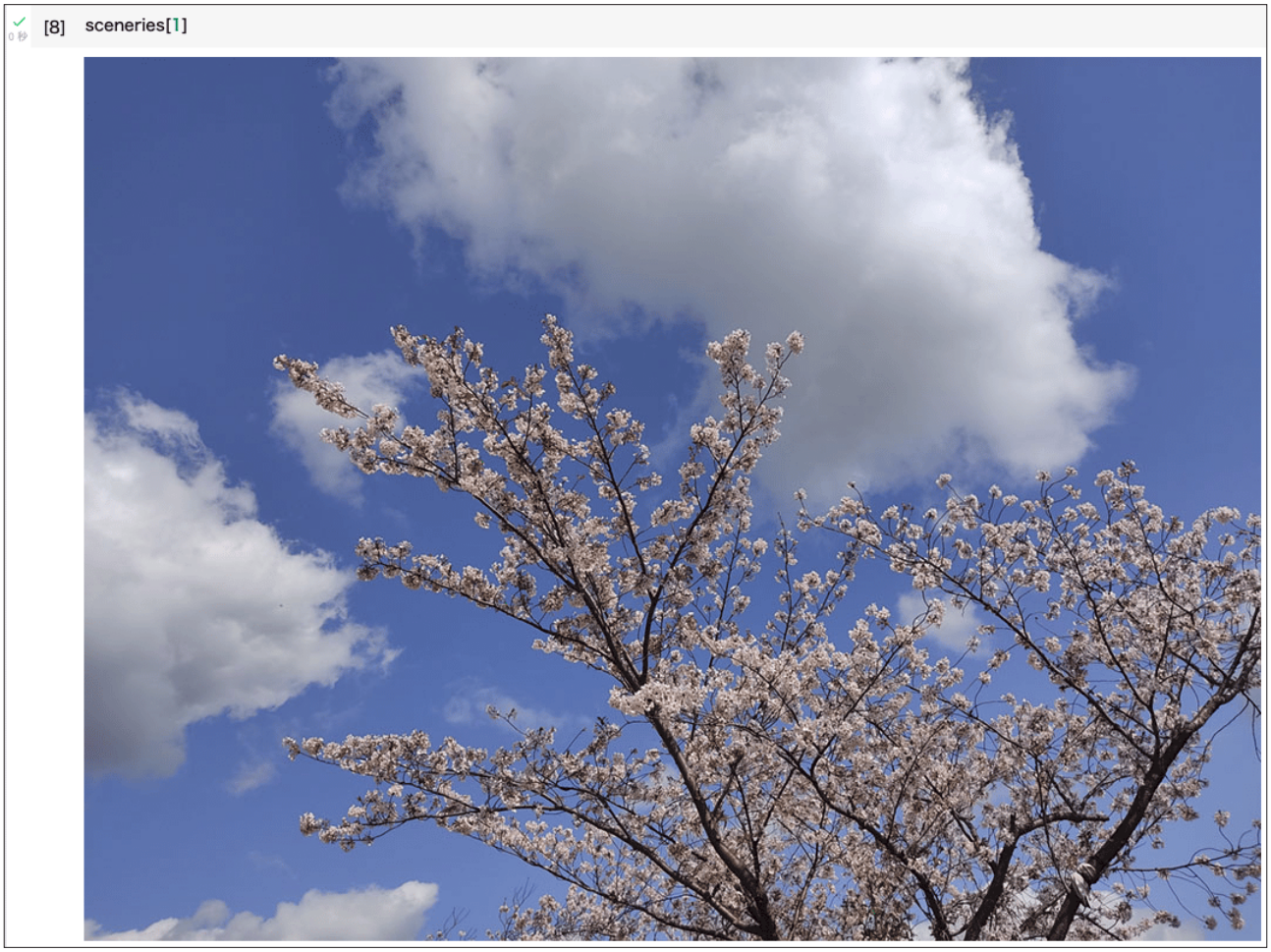
写真を読み込む

実行結果を以下に示します。紅葉の写真は以下の通りです。



紅葉の写真

そして、以下が桜の写真です。



桜の写真

このモデルでは画像を埋め込み表現に変換するのも encode メソッドを呼び出して、読み込んだ画像を渡すだけです。ここでは 2 つの写真があるのでリスト内包表記を使っています。

```
sceneries_embs = [model.encode(sceneries[num]) for num in range(2)]
print(sceneries_embs[0].shape)
```

写真の埋め込み表現の作成

後はこれらをテキストの埋め込み表現とのコサイン類似度を計算するだけですが、先ほどの「a dog」「a cat」ではなく、ここでは以下のようにそれっぽい表現のテキストを用意しました。

```
text0 = 'red leaves in a park'
text1 = 'cherry blossoms in blue sky'
text2 = 'a stray cat on a street'
sentence_list = [text0, text1, text2]
sentence_embs = [model.encode(item) for item in sentence_list]
```

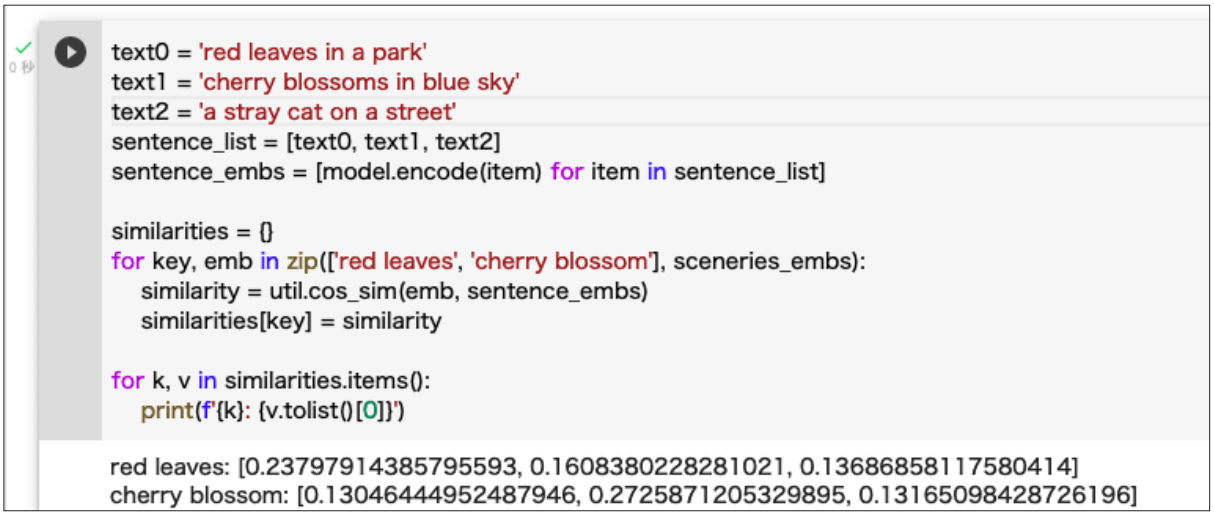
写真とコサイン類似度を比較するためのテキストとその埋め込み表現

最後に 2 枚の写真（の埋め込み表現）それぞれについて、上のテキスト（の埋め込み表現）とコサイン類似度を計算するだけです。

```
similarities = {}
for key, emb in zip(['red leaves', 'cherry blossom'], sceneries_embs):
    similarity = util.cos_sim(emb, sentence_embs)
    similarities[key] = similarity

for k, v in similarities.items():
    print(f'{k}: {v.tolist()[0]}')
```

コサイン類似度の計算

ここでは 3 つのテキストの埋め込み表現をまとめたリストを util.cos_sim 関数に渡していますが、こうすると一度にまとめてコサイン類似度を計算してくれます。その結果を辞書に格納して（キーは紅葉を表す「red leaves」と桜の花を表す「cherry blossom」としました）、それぞれの計算結果を表示しています。



2 枚の写真と 3 つのテキストとのコサイン類似度

「a dog」と「a cat」のコサイン類似度と比べると、ずいぶんその値は低くなっていますが、紅葉の写真については「red leaves in a park」というテキストが、桜については「cherry blossoms in blue sky」というテキストが一番よく似ている結果になっています。これはまずまずの結果といえるのかもしれませんね。

Stable Diffusion で作成した画像とそのプロンプトを比較してみる

最後にちょっと長めのプロンプトを Stable Diffusion に渡して画像を生成してもらい、画像とその生成に使用したプロンプトのコサイン類似度を計算してみます。

ここではプロンプトは「a photo of a knight armed with a long sword, sitting on a rock, in a forest」（ロングソードを持っている騎士が森の中で岩に座っている写真）としました。また、前回と同様に、ここでは Diffusers の StableDiffusionPipeline クラスを用いています。これを使って画像を生成する手順については前回の記事を参照してください。

以下に画像を生成するコードを示します。

```
import torch
from diffusers import StableDiffusionPipeline

YOUR_TOKEN = 'hf_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'  # 自分のトークンを設定

pipe = StableDiffusionPipeline.from_pretrained('CompVis/stable-diffusion-v1-4',
                                               use_auth_token=YOUR_TOKEN)
pipe.to('cuda')

prompt = 'a photo of a knight armed with a long sword, sitting on a rock, in a forest'

generator = torch.Generator('cuda').manual_seed(2)
image = pipe(prompt, guidance_scale=7.5, num_inference_steps=50,
             generator=generator).images[0]
image
```

「ロングソードを持っている騎士が森の中で岩に座っている写真」を生成

実行結果を以下に示します。

```
[13] generator = torch.Generator('cuda').manual_seed(2)
     image = pipe(prompt, guidance_scale=7.5, num_inference_steps=50,
                  generator=generator).images[0]
     image
```

```
100%|██████████| 51/51 [00:28<00:00, 1.75it/s]
```

実行結果

筆者的にはまずまずの画像だと思いますが、コサイン類似度はどんな感じになるでしょう。

```
def calc_cos_score(image, prompt):
    img_emb = model.encode(image)
    text_emb = model.encode(prompt)
    cos_scores = util.cos_sim(img_emb, text_emb)
    return cos_scores

calc_cos_score(image, prompt)
```

コサイン類似度を計算する関数の定義とその呼び出し

ここではコサイン類似度を計算する関数を定義して、それを呼び出すようにしました。関数内で行っているのは既に見た通りの処理です。実行結果を以下に示します。

```
[14]  def calc_cos_score(image, prompt):
          img_emb = model.encode(image)
          text_emb = model.encode(prompt)
          cos_scores = util.cos_sim(img_emb, text_emb)
          return cos_scores

[15]  calc_cos_score(image, prompt)

      tensor([[0.3036]])
```

思ったより低かったコサイン類似度

あれ？ 「0.3036」と思ったよりもコサイン類似度が低いような気がします。0.9 とはいわないまでも 0.6 くらいは出そうな期待感を持って実行してみたのですが。

そこで、guidance_scale に強めの値を指定してみることにしました。これは指定したプロンプトに対する忠実度というか、画像生成時にプロンプトの影響力を強める値です。これに「20」を指定してみましょう。

```
generator = torch.Generator('cuda').manual_seed(2)
image = pipe(prompt, guidance_scale=20, num_inference_steps=50,
             generator=generator).images[0]
image
```

プロンプトの影響力を強くする

実行結果を以下に示します。

```
[16]  generator = torch.Generator('cuda').manual_seed(2)
      image = pipe(prompt, guidance_scale=20, num_inference_steps=50,
                   generator=generator).images[0]
      image
```

```
100% 51/51 [00:29<00:00, 1.72it/s]
```

実行結果

変わったといえば変わりましたが、コサイン類似度にはそれほど差は出そうもない気がします。先ほど定義した関数を呼び出してみましょう。

```
[ ]  calc_cos_score(image, prompt)

     tensor([[0.3121]])
```

実行結果

うーむ。少しは高くなりましたが、やはりコサイン類似度が 1 に大きく近づくことはありませんでした。guidance_scale を強めにするというのは、StableDiffusionPipeline クラスの実装では次の部分に関係をしています。

```
if do_classifier_free_guidance:
    noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)
     noise_pred = noise_pred_uncond + guidance_scale * (noise_pred_text -
noise_pred_uncond)
```

guidance_scale が使われる箇所（StableDiffusionPipeline クラスのソースコードより）

簡単に説明をすると、guidance_scale が 1 より大きいときには「classifier free guidance」と呼ばれる処理が行われます。これはプロンプトで条件付けをした上で除去されるノイズを予測した結果と、プロンプトなしで除去されるノイズを予測した結果から最終的なノイズを予測するといった処理を行うようです。上のコードを見ると、これは実際には guidance_scale の値が「プロンプトを基に予測されたノイズとプロンプトなしで予測されたノイズの差」の値に乗算されるようなコードになっています。

こうすることで先ほども述べたようにプロンプトの影響力が強まる（忠実度が上がる）結果になるようです。ですがだからといって、影響力が高まるとコサイン類似度が大きく 1 に近づくというわけでもないようです。これは、画像の埋め込み表現を得る際に、人の目には見えないような多数の情報が 768 次元の空間に織り込まれるからかもしれません。

ノートブックには guidance_scale の値を変化させながら、画像を生成し、それらとプロンプトのコサイン類似度を計算するコードもあるので興味のある方はご覧ください。

今回の記事で何が重要かといえば、CLIP はテキストと画像の両者から互換性のある埋め込み表現を得ることができることと、それ故に画像生成に利用するプロンプトから埋め込み表現を得れば、それを画像生成時にどんな画像にするかの条件付けに利用できるということです。埋め込み表現がテキストと画像の橋渡しとなったことで、Stable Diffusion に代表される画像生成 AI が大きく進化することになったといえるでしょう。

Stable Diffusion 2.0 で追加された機能を試してみよう

Stable Diffusion 2.0 で可能になった 768×768 ピクセルの画像生成、画像のアップスケール、画像の一部の書き換えとはどんなものかを見ていきます。

かわさきしんじ，Deep Insider 編集部（2022 年 12 月 02 日）

Stable Diffusion 2.0

その発表と共に画像生成 AI の世界に大きな変化を巻き起こした Stable Diffusion のバージョン 2.0 が 2022 年 11 月にリリースされました。リリースによれば、このモデルは、Stability AI のサポートを受けて、LAION が開発した新しいテキストエンコーダーを使用して一から訓練が行われたもので、バージョン 1 と比べると生成される画像の品質が大幅に向上したと述べられています。

Stable Diffusion 2.0 のリリースのアナウンス

そこで、本稿ではその新機能の幾つかを Diffusers（Stable Diffusion などの拡散モデルをラップするライブラリ）を用いて実際に試してみることにします。

今回もサンプルコードは Google Colab のノートブックに記述しています。が、個別のコードはそれほど長いものではありません（というか、とても短いです。かわさき）。

リリースによれば、Stable Diffusion 2.0 では以下のような点が強化されています。

- 生成される画像のデフォルトの解像度が 512×512 ピクセルとなるモデルと、768×768 ピクセルとなるモデルを提供
- 画像を入力すると 4 倍の大きさにアップスケール可能なモデル
- 入力された画像の深度を推測し、深度とプロンプトから新たな画像を生成可能なモデル
- 画像（および変更したい部分を示すマスク画像）とプロンプトを入力すると、画像の一部だけをプロンプトに応じて書き換えるモデル

これらの強化点は以下のモデルによって提供されます。

- 512×512 ピクセルの画像生成（text-to-image）：stable-diffusion2-base モデル
- 768×768 ピクセルの画像生成：stable-diffusion-2 モデル
- 画像を 4 倍のサイズにアップスケール：stable-diffusion-x4-upscaler モデル
- 画像の深度とプロンプトからの画像生成：stable-diffusion-2-depth モデル
- 画像の一部の書き換え：stable-diffusion-2-inpainting モデル

最初の 2 つのモデルの違いは stable-diffusion-2-base モデルは 256×256 ピクセルと 512×512 ピクセルの画像を使って訓練が行われているのに対して、stable-diffusion-2 モデルはこれを基にさらに 768×768 ピクセルの画像を使って訓練が行われている点です。より解像度の高い画像を使って訓練されているので、stable-diffusion-2 モデルは stable-diffusion-2-base モデルよりも生成される画像がより高精細なものになることが期待できます。

本稿執筆時点（2022 年 11 月 29 日）では、これらのモデルのうち stable-diffusion-2-depth モデルが Diffusers でまだサポートされていません。そのため、以下では 768×768 ピクセルの画像生成、アップスケール、画像の一部の書き換えの 3 つについて簡単なコードを書いて、その機能を体験してみましょう。

なお、上記のモデル名は Diffusers が提供するパイプラインクラスのインスタンスを生成する際に ID として指定する際に使用します。また、本稿執筆時点ではモデルの使用に際して Hugging Face への登録などは必要ありませんが、今後、変更される可能性はあります。

Google Colab ノートブックで以下の例を試すには、最初に以下のコードで Diffusers（とその他のライブラリ）をインストールしておく必要があります。

```
!pip install --upgrade git+https://github.com/huggingface/diffusers.git transformers accelerate scipy
```

Diffusers とその他のライブラリのインストール

768×768 ピクセルの画像生成

Stable Diffusion バージョン 1 では生成できる画像の最大サイズは 512×512 ピクセルでしたが、stable-diffusion-2-base モデルを基とした stable-diffusion-2 モデルでは 768×768 ピクセルの画像を生成できるようになっています。

以下はこれを試すコードの例です。

```
import torch
from diffusers import StableDiffusionPipeline

model_id = 'stabilityai/stable-diffusion-2'

pipe = StableDiffusionPipeline.from_pretrained(model_id, revision='fp16', torch_dtype=torch.float16)
pipe = pipe.to('cuda')

prompt = 'a photo of an astronaut riding a horse on mars'
image = pipe(prompt).images[0]
image
```

768×768 ピクセルの画像生成

diffusers パッケージから StableDiffusionPipeline クラスをインポートするのはバージョン 1 と同様ですね。その後で変数 model_id に指定しているのが、どのモデルを使用するかの指定です。ここでは「stabilityai/」に続けて上で紹介したモデル名を付加した「stabilityai/stable-diffusion-2」がモデルの ID となっています。これをパイプラインクラスの from_pretrained メソッドに渡すことでモデルを手に入れています。

後はプロンプトを設定して、モデルを呼び出すだけです。このようなモデル ID を指定してパイプラインクラスのインスタンスを生成して、それを関数のように呼び出すというのが Stable Diffusion 2.0 を Diffusers から使用する場合の定型パターンとなっているようです。

実行結果は次のようになりました（馬に乗っていないような……笑）。

```
[2]  model_id = 'stabilityai/stable-diffusion-2'

     pipe = StableDiffusionPipeline.from_pretrained(model_id, revision='fp16', torch_dtype=torch.float16)
     pipe = pipe.to('cuda')

     generator = torch.Generator('cuda').manual_seed(2)
     prompt = 'a photo of an astronaut riding a horse on mars'
     image = pipe(prompt, generator=generator).images[0]
     image
```

```
Fetching 12 files: 100% 12/12 [00:00<00:00, 323.21it/s]
ftfy or spacy is not installed using BERT BasicTokenizer instead of ftfy.
100% 50/50 [00:28<00:00, 1.99it/s]
```

生成された 768×768 ピクセルの画像

では、stable-diffusion-2-base モデルではどうなるでしょう。モデル ID を「stabilityai/stable-diffusion-2-base」に変更した以外は上と同じコードを実行した結果を以下に示します。

```
[3]  model_id = 'stabilityai/stable-diffusion-2-base'

     pipe = StableDiffusionPipeline.from_pretrained(model_id, revision='fp16', torch_dtype=torch.float16)
     pipe = pipe.to('cuda')

     prompt = 'a photo of an astronaut riding a horse on mars'
     image = pipe(prompt).images[0]
     image
```

```
Fetching 12 files: 100% 12/12 [00:00<00:00, 172.60it/s]
ftfy or spacy is not installed using BERT BasicTokenizer instead of ftfy.
100% 50/50 [00:08<00:00, 5.77it/s]
```

生成された 512×512 ピクセルの画像

なお、stable-diffusion-2-base モデルでも 768×768 ピクセルの画像は生成できます。Google Colab ノートブックには試した結果もあるので、興味のある方はご覧ください。筆者には stable-diffusion-2 モデルとどちらが高精細な画像かの区別は付きませんでした（笑）。

横幅 768px（ピクセル）で作成できることで利用できる場面も増えそうですね。欲を言えば、横幅 1280px の画像が作成できると、もっとうれしいな。

4 倍のサイズにアップスケール

stable-diffusion-x4-upscaler モデルを使って、アップスケールする場合もその使い方は上と同様です。ただし、使用するパイプラインクラスは StableDiffusionUpscalePipeline クラスに、モデル ID は「stabilityai/stable-diffusion-x4-upscaler」になります。

以下に簡単な例を示します。

```
from diffusers import StableDiffusionUpscalePipeline
from PIL import Image

model_id = 'stabilityai/stable-diffusion-x4-upscaler'
pipe = StableDiffusionUpscalePipeline.from_pretrained(model_id,
revision='fp16', torch_dtype=torch.float16)
pipe = pipe.to('cuda')
```

パイプラインクラスのインスタンスを生成

上で述べた通り、「stabilityai/stable-diffusion-x4-upscaler」をモデル ID として、StableDiffusionUpscalePipeline クラスのインスタンスを生成しています。そして、ここでは以下に示すネコちゃんの写真をアップスケールしてみることにしました。

ネコちゃん

なお、上の画像は 512×512 ピクセルの画像となっているので、ご自分でコードを試してみるのであれば画像を保存して、Google Colab にアップロードしてください（ファイル形式に合わせてコード中の拡張子を修正してください）。

この画像をまずは縦横 4 分の 1 のサイズつまり 128×128 ピクセルに縮小しておきます。その後、アップスケールを行って、元の画像と遜色ないものになっていれば万々歳です。

```
img = Image.open('cat.jpg') # 512×512 ピクセルのネコ画像を読み込み
low_res_img = img.resize((128, 128)) # それを 4 分の 1 に縮小
```

ネコちゃん画像を読み込んでそれを縦横 4 分の 1 に縮小

最後にプロンプトとこの画像をモデルに入力するだけです。

```
prompt = 'a white dog'

upscaled_image = pipe(prompt=prompt, image=low_res_img,).images[0]
upscaled_image
```

縦横 4 分の 1 に縮小した画像をアップスケール

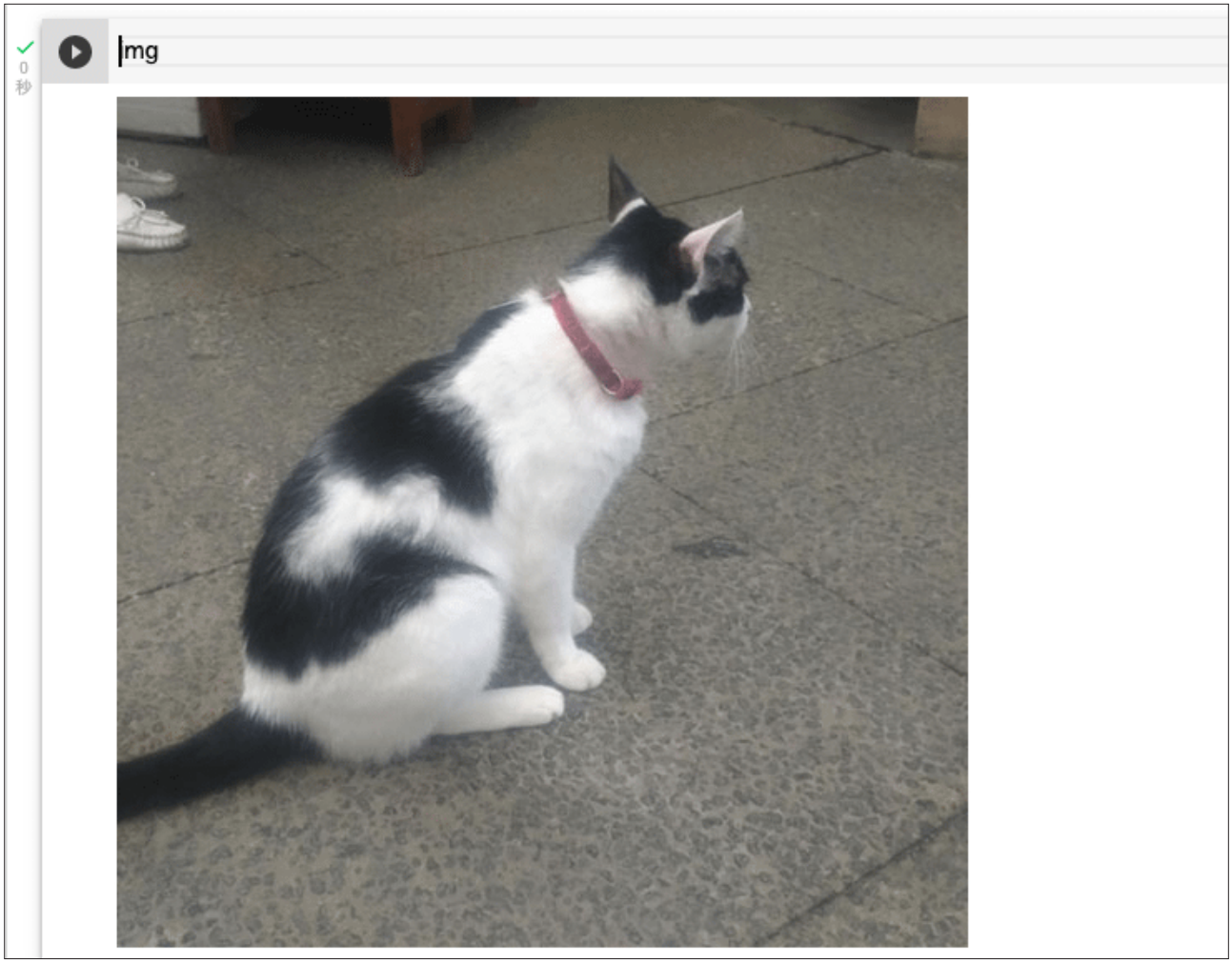
プロンプトに「a white dog」にしている点に注目してください。果たして、ネコちゃんはワンちゃんになった上で 512×512 ピクセルになるのでしょうか。

実行結果

ワンちゃんには全くなりませんでしたね。

ソースコードを見る限りはある程度の影響はありそうなのですが、どうしてこうなった（何度やってもこんな感じなので、画像のアップスケールという目的を考えると、プロンプトによる影響が出ないようになっているのかもしれません）。

ネコちゃんの元画像を再掲しておきます。



ネコちゃんの元画像（再掲）

首輪の周辺や背中の毛並みなどをよく見ると、アップスケールされた画像と元画像には相違点があることが分かります。これは 128×128 ピクセルの画像をアップスケールしたからかもしれません（元画像のサイズがもっと大きければ、もっとよい結果が出そうな気はします）。

画像の一部の書き換え

ここまでくればお分かりでしょうが、画像の一部を書き換えるコードもほぼほぼ同じです。特にモデルに画像を入力する点は同じです。ですが、画像の一部を書き換えるときには、書き換えたい部分とそうではない部分を指定する必要があります。stable-diffusion-inpainting モデルでは、書き換えたい部分は白で、そのままにしておきたい部分は黒で塗りつぶした「マスク画像」も入力することになります。

ここではマスク画像として以下を用意しました（書き換えを行う元画像は先ほどのネコちゃんです）。

マスク画像

マスク画像は筆者が Photoshop で適当に作ったものですが、まあ、こんなものでしょう。

マスク画像が必要なのは手間ですね……。正直、使いづらいなと思います。候補マスク画像も自動生成してくれるともっと良いのだけど。

元の画像と上のマスク画像、それからプロンプトをモデルに入力すると、マスク画像で白くなっている箇所に対応する元画像の位置がプロンプトに従って書き換えられるというわけです。

まずはパイプラインクラスのインスタンスを取得するコードから。

```
from diffusers import StableDiffusionInpaintPipeline
from PIL import Image

model_id = 'stabilityai/stable-diffusion-2-inpainting'
pipe = StableDiffusionInpaintPipeline.from_pretrained(model_id,
revision='fp16', torch_dtype=torch.float16)
pipe = pipe.to('cuda')
```

StableDiffusionInpaintingPipeline オブジェクトの生成

そして、2 つの画像を読み込みます。

```
img = Image.open('cat.jpg')
img_mask = Image.open('cat_mask.jpg')
```

画像の読み込み

最後にプロンプトを指定して、モデルを呼び出しましょう。

```
prompt = 'silver fox'
image = pipe(prompt=prompt, image=img, mask_image=img_mask, num_inference_
steps=25).images[0]
image
```

画像の一部を書き換える

ここではプロンプトはシンプルに「a silver fox」にしてあります。結果は次のようになりました。

```
[13] prompt = 'silver fox'
     image = pipe(prompt=prompt, image=img, mask_image=img_mask).images[0]
     image
```

```
100% 50/50 [00:09<00:00, 5.45it/s]
```

実行結果

元画像ではネコちゃんがいたところに銀狐（silver fox）っぽいものが見事に描画されています（わーい）。

マスク部分に無理やり当てはめているため、若干、無理やり感ありますね……。

それはいわない約束でしょっ。

このように Stable Diffusion 2.0 では少量のコードで高度な機能を簡単に扱えるようになっています。

編集：@IT 編集部
発行：アイティメディア株式会社