Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Studio di modelli generativi per gamification per beni culturali

Candidato
Andrea De Rosa

Relatore
Prof. Marco Bertini

Correlatore
Andrea Ferracani

Anno Accademico 2023/2024

Indice

Introduzione

Il sistema è progettato come Web app per dispositivi mobili e consiste in un gioco in cui l'utente può ricreare una o più opere presenti nella collezione del museo.
Il sistema è pensato per essere utilizzato dai visitatori una volta terminata la loro visita al museo, fornendo loro la possibilità di ricreare, inserendo una breve descrizione, una delle opere che hanno visto al museo. Aperta l'applicazione l'utente avrà la possibilità di scegliere quale opera ricostruire attraverso una descrizione da lui data, ed una volta scelta verrà reindirizzato alla pagina dove dovrà inserire la descrizione.

La descrizione inserita sarà inviata ad un algoritmo di Intelligenza Artificiale chiamato text to image, addestrato sulle opere disponibili; l'algoritmo restituirà quindi l'immagine generata, che verrà mostrata all'utente affianco all'originale insieme ad un'indice di similarità.

L'utente potrà leggere la descrizione dell'opera cliccando sull'immagine originale e avrà la possibilità di rigenerare nuove immagini finché non sarà soddisfatto del risultato.
Una volta soddisfatto potrà terminare il processo e sarà mandato ad una nuova pagina dove potrà vedere tutte le opere che ha generato fino a quel

momento; le immagini saranno mostrate all'interno di una canvas insieme all'opera originale su cui sono basate. L'immagine risultato dell'unione tra originale e generata sarà scaricabile tramite apposito pulsante.

L'interfaccia dell'applicazione è divisa in due parti:
Interfaccia Utente accessibili a tutti e *Interfaccia Admin* accessibile solo tramite login.
La *Interfaccia Utente* costituisce l'applicazione principale, ovvero l'effettivo gioco, dove l'utente può ricreare le immagini che desidera.
La *Interfaccia Admin* sarà accessibile agli operatori del museo tramite login e permetterà principalmente di aggiungere o rimuovere nuove opere nella collezione. Ogni museo avrà le proprie collezioni.

Il lato client, ovvero l'interfaccia, è stato creato utilizzando come linguaggi html e css e il design delle varie pagine pone il focus sull'essere usato su dispositivo mobile e avere tutti gli elementi necessari a schermo nello stesso momento. Il lato server è invece scritto in python utilizzando il framework flask.

L'algoritmo di intelligenza artificiale è scritto in python, utilizzando come modello di apprendimento Stable Diffusion

Struttura della tesi

Capitolo 1

Il primo capitolo descrive il funzionamento della WebApp, mostrando le funzionalità

Capitolo 2

Nel secondo capitolo sarà approfondito l'algoritmo di intelligenza artificiale che servirà per generare le immagini su descrizione dell'utente che verranno poi mostrate all'utente. Il capitolo sarà diviso in due parti, una per il text-to-image e l'altra per il DreamBooth

Capitolo 3

Il terzo capitolo descrive il codice dell'applicazione, mostrando l'interazione tra i vari moduli

Capitolo 1

WebApp

1.1 Funzionamento della WebApp

Come già descritto nell'introduzione, l'utente dopo una visita al museo, potrà accedere a un'applicazione dove gli verranno mostrate le varie opere che ha visionato.

Le immagini verranno mostrate con una sfocatura nello sfondo, e sarà ben visibile soltanto il soggetto principale dell'opera.
L'utente dovrà quindi dare una descrizione dello sfondo sulla base di ciò che ricorda.
Verrà generata un'immagine che l'utente potrà scaricare, e nel caso non sia soddisfatto del risultato potrà generarne una nuova.

Queste opere saranno inserite direttamente dal museo all'interno dell'applicazione. E' presente una sezione apposita per i gestori del museo dove è possibile inserire l'immagine delle opere presenti nel museo.

1.2 Interfaccia Utente

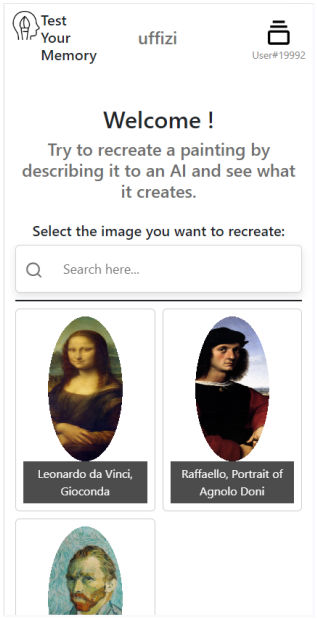

Figura 1.1: Homepage

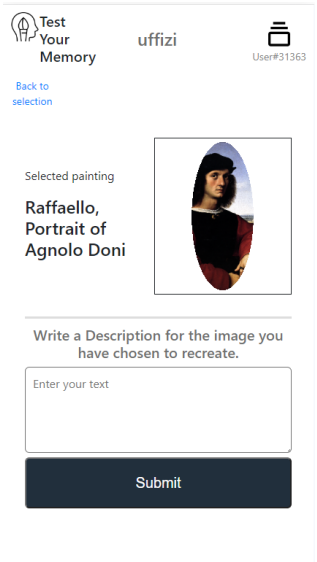

Figura 1.2: Dopo che l'utente avrà selezionato un'opera, sarà possibile inserire la descrizione dello sfondo, per poi generare l'immagine

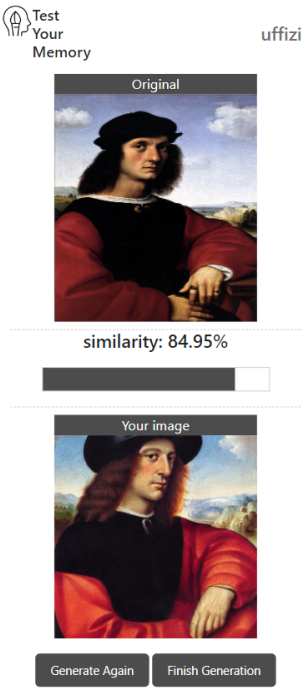

Figura 1.3: Dopo aver cliccato submit e aver generato l'immagine, all'utente viene mostrata l'originale e l'immagine che è stata generata

1.3 Interfaccia Museo

Il museo si occuperà inserire le immagini all'interno della WebApp, per poi renderle disponibili agli utenti.
In particolare, dovrà caricare l'immagine dell'opera nella sezione apposita, e insieme a questa inserire una descrizione accurata che servirà all'algoritmo di text-to-image per generare il modello, che verrà poi utilizzato per la generazione delle immagini da parte dell'utente.
Inoltre sarà possibile creare delle collezioni, ovvero raggruppare varie opere che sono già state inserite

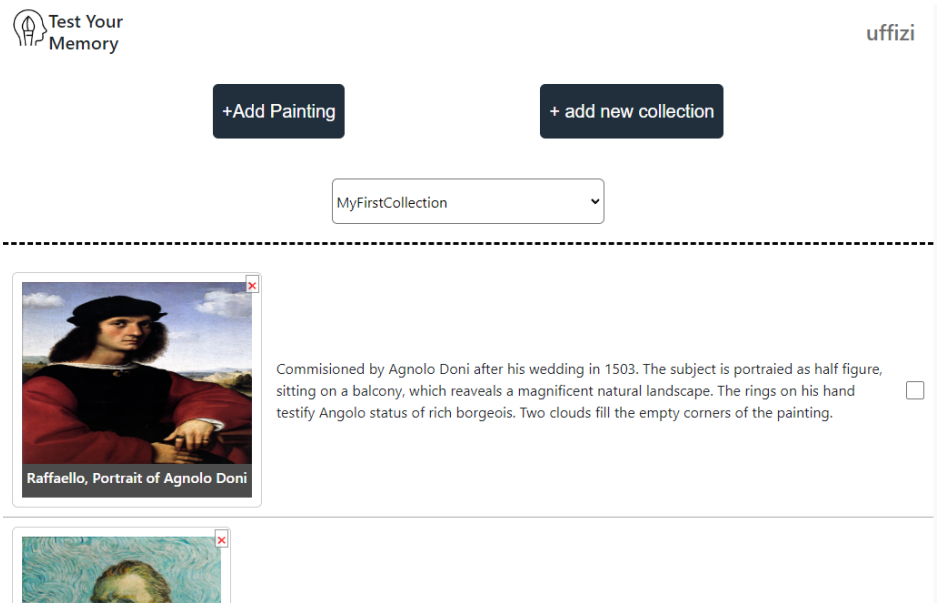

Figura 1.4:

Capitolo 2

Text-To-Image

2.1 Introduzione all'algoritmo text-to-image

Gli algoritmi di intelligenza artificiale text-to-image sono algoritmi in grado di generare immagini a partire da un testo descrittivo
In base a ciò che vuole essere ottenuto nel contesto in cui si lavora, è necessario applicare un processo chiamato fine tuning.

Il fine tuning è il processo di perfezionamento dei modelli di base per la creazione di un nuovo modello più adatto a specifici compiti o domini.
E' possibile aggiungere dati per l'addestramento specifici per determinati scenari di utilizzo invece di affidarsi a un modello generico.

2.2 DreamBooth

Il DreamBooth è modello di deep learning sviluppato da Google che personalizza i modelli esistenti di text-to-image (**Stable Diffusion**) facendo un fine tuning

L'obiettivo principale di DreamBooth è quello di generare immagini personalizzate di uno specifico soggetto.

Per fare il fine tuning vengono utilizzate più immagini dello stesso soggetto, e per ogni immagine, viene data una descrizione

Figura 2.1: Esempio di generazione con DreamBooth

I parametri fondamentali per la generazione di un'immagine utilizzando un modello text-to-image sono i seguenti:

1: **Guidance**
Valore solitamente compreso tra 1 e 20.
Questo valore indica quanto peso verrà dato al testo per la generazione dell'immagine. Più questo valore è alto, più l'algoritmo tenderà a dare importanza al testo, viceversa, nel caso questo valore sia molto piccolo, verrà data poca importanza al testo dell'utente, e quindi verrà generata un'immagine quasi uguale a una di quelle utilizzata per l'addestramento del modello.

2: **Number of inference steps**

Valore che rappresenta le iterazioni utilizzate per generare l'immagine.

Ogni step rappresenta il progresso della generazione, e più questo numero è alto, più sarà migliorata la precisione dell'output.

Non c'è un numero preciso da utilizzare, perché il numero che si deciderà di utilizzare dipenderà principalmente da quali output sono ottenuti durante la fase di testing.

Generalmente si inizia sempre a fare dei testo con 30 steps, e si aumenta progressivamente di 5 steps per test.

E' importante notare che un numero alto di steps non vuol dire per forza qualità migliore

3: **Seed**

Valore utilizzato per inizializzare la generazione

Solitamente questo valore viene generato casualmente, quindi inserendo un input di testo, e generando 2 immagini, non si ottengono 2 immagini uguali

Mentre se si inserisce un input e si generano 2 o più immagini con lo stesso numero di seed, verranno generate immagini uguali.

2.3 Problemi di DreamBooth nei dipinti

Nel caso si abbia a che fare con dei dipinti la situazione diventa più complicata, perché non esistono più versioni dello stesso dipinto, quindi ci si ritrova a dover fare il fine tuning usando una sola foto, e questo non darà risultati soddisfacenti quando si andranno a generare delle immagini.

Provando a creare delle immagini che sono varianti dell'originale, per esempio lasciare solo lo sfondo del dipinto, oppure oscurare lo sfondo e lasciare il soggetto, e mettendo le opportune descrizioni alle immagini (che serviranno per il fine tuning) si riesce a dare una precisione maggiore.
Anche andando a modificare i 3 parametri citati prima, e prendendo i valori che danno risultati medio-buoni, si riescono a generare immagini che sono molto simili all'originale.

Il problema di questo approccio è che si cade nell'overfitting
L'overfitting si verifica quando il modello non può essere generalizzato e si adatta in modo eccessivo al set di dati di addestramento

In questo caso, il problema è la sovra ottimizzazione dei 3 parametri, e il modo di creare immagini alternative(immagine originale, immagine con solo sfondo, immagine con solo i soggetti principali del dipinto)

Di seguito è mostrato un esempio di overfitting.
E' stato fatto un lavoro di sovra ottimizzazione sulla Gioconda, ripetendo gli stessi procedimenti sull'autoritratto di Van Gogh si ottengono risultati non soddisfacenti

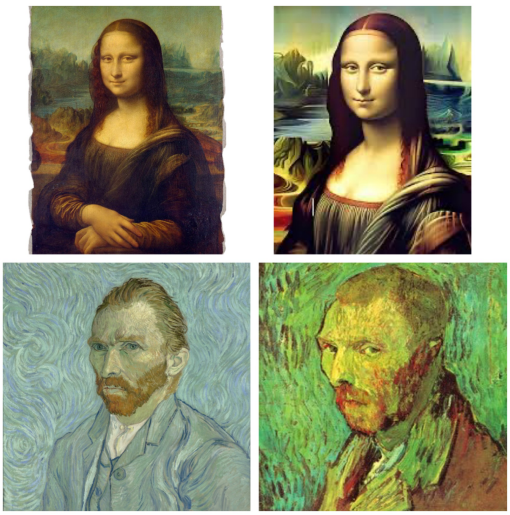

Figura 2.2: Esempio di overfitting, a sinistra l'originale, a destra l'immagine generata con un testo descrittivo

Si utilzza quindi una variante di DreamBooth chiamata Imagic, che ha bisogno di una sola immagine per fare un fine tuning, e per la generazione dell'immagine, andrà a manipolare quest'ultima per restituire quello che è stato descritto nell'input.

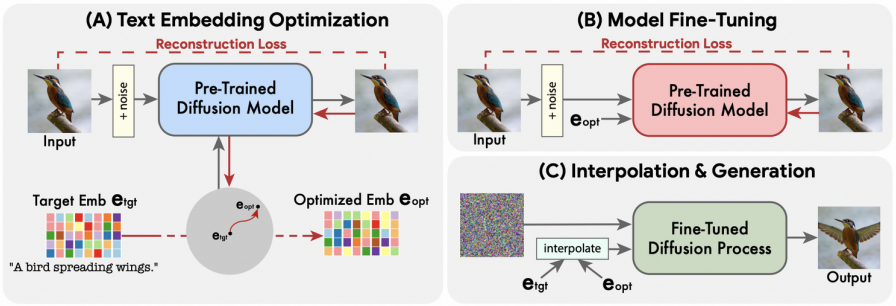

Figura 2.3: Schema del funzionamento del fine tuning e della generazione dell'immagine

Capitolo 3

Struttura del codice

3.1 Librerie e framework utilizzati

Sono stati utilizzati:

Pytorch: è un framework di apprendimento automatico basato sulla libreria Torch, utilizzato per applicazioni come la visione artificiale e l'elaborazione del linguaggio naturale

Request: è una libreria per poter effettuare chiamate rest.
Il funzionamento di una chiamata rest è che il client invia una chiamata ad un URL specificando un metodo (GET, POST, PUT, DELETE) e passando, se necessario, dei dati in JSON nel corpo della richiesta

Redis: è una libreria per la gestione delle code, in particolare si occupa di eseguire i task in background.

L'utilizzo di una coda dove vengono messi i vari task da eseguire è indi-

spensabile, perché la generazione di un modello richiede molto utilizzo della gpu, quindi in caso di molte richieste concorrenti, si arriva a saturare la gpu e non portare a termine le operazioni.

Può anche capitare che ci siano molti utenti che inviino la richiesta di generazione dell'immagine, e anche questo evento potrebbe saturare la gpu, rendendo l'applicazione non utilizzabile

3.2 Casi d'uso e Documentazione del codice

Gli attori principali sono: Utente e Gestore del museo.

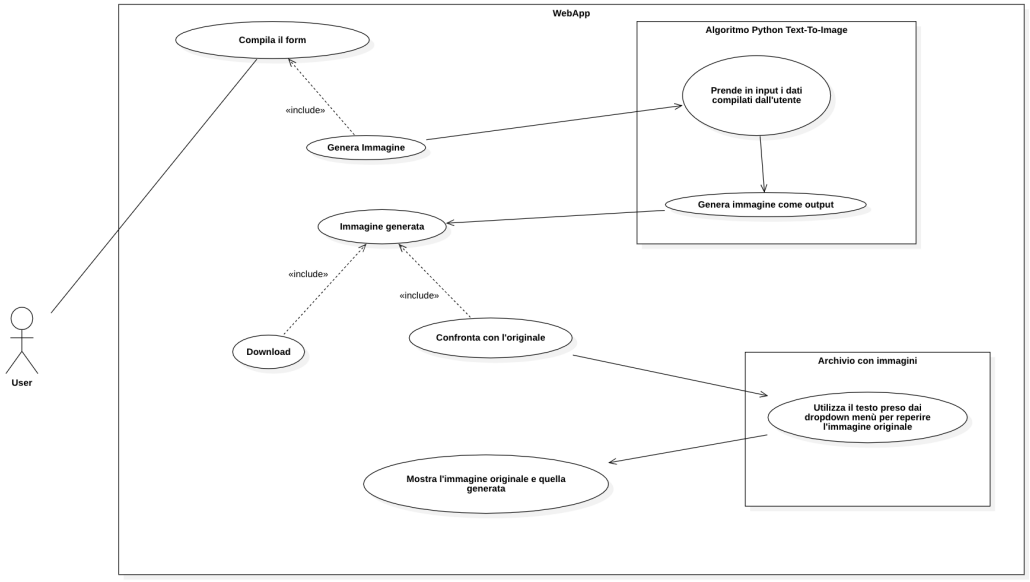

Figura 3.1: Caso d'uso per utente che genera l'immagine

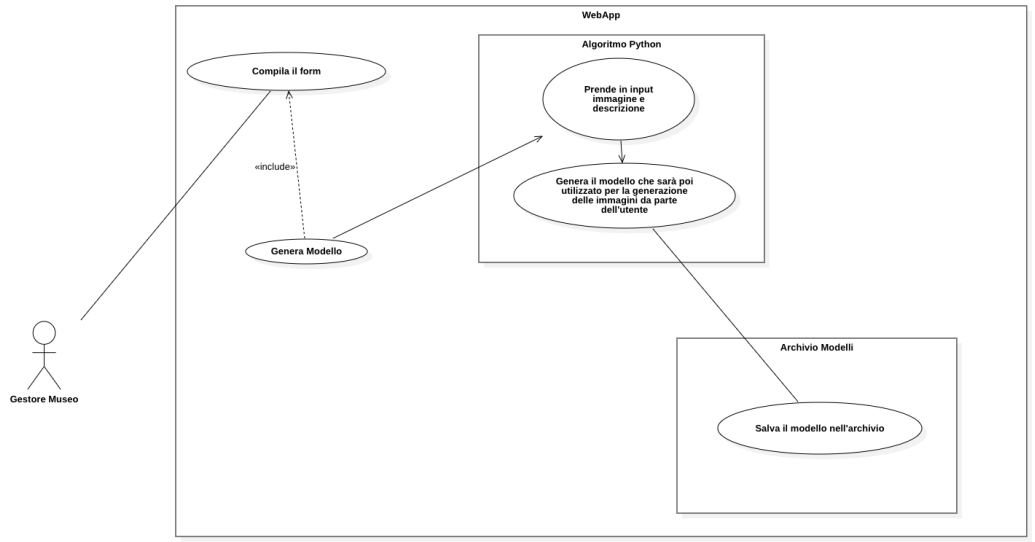

Figura 3.2: Caso d'uso per gestore museo che genera il modello

Documentazione del codice:

handle_request_admin(): Funzione per la generazione del modello da parte del gestore del museo.
Viene eseguita attraverso l'app routing di flask, ovvero che una volta effettualo l'accesso a una pagina specifica di un sito, viene eseguita la rispettiva funzione.
Dopodiché viene aggiunta la richiesta nella coda.
I valori: *portrait_name, description, image* vengono presi in input attraverso la libreria request

background_task_admin(portrait_name, description, image): task eseguito all'interno della coda, il codice si occupa di generare il modello

handle_request_user(): Funzione per la generazione dell'immagine da parte dell'utente.
Anche questa viene eseguita attraverso l'app routing di flask e viene aggiunta la richiesta nella coda una volta eseguita la funzione.
I valori: *prompt, directory, target_text* vengono presi in input attraverso la libreria request

background_task_user(prompt, directory, target_text): task eseguito all'interno della coda, il codice si occupa di generare l'immagine da restituire all'utente

3.3 Codice per la generazione del modello

Quando il museo inserisce immagine e descrizione che serviranno per creare il modello nell'apposita sezione del sito, e invierà la propria richiesta, verrà effettuata una chiamata rest

Una cosa importante da tenere in considerazione è la descrizione dell'immagine che viene data all'algoritmo per generare il modello, che verrà poi usato per generare le immagini con gli input degli utenti.
Essa deve essere una descrizione completa dell'opera, seguito dal nome dell'autore dell'opera.
Mettere l'autore in seguito alla descrizione dell'opera aiuta il modello a generare un'immagine che si avvicina molto allo stile dell'artista.

NB: Le variabili *PORTRAIT_NAME, DESCRIPTION, IMAGE* nel codice della chiamata rest lato admin e *TARGET_TEXT, USER_TEXT, PORTRAIT_NAME* nel caso della chiamata rest lato utente, vengono assegnate dai parametri presi in input direttamente dalla WebApp.
Le variabili sono state dichiarate come costanti solo a scopo illustrativo

```
BASE_URL = 'http://solaris.micc.unifi.it/andreaderosa'

PORTRAIT_NAME="gioconda"
DESCRIPTION="woman with long hair and crossed hand by
                              leonardo da vinci"
IMAGE=cv2.imread('/directory/gioconda.png')

payload = {'portrait_name':PORTRAIT_NAME, 'description':
                              DESCRIPTION, 'image':IMAGE}
response = rq.get(BASE_URL, params=payload)
```

```
img_name=response.text
```

funzione che viene chiamata quando il museo carica l'opera e fa richiesta per la generazione.

Verrà restituito in output una stringa che dirà l'esito dell'operazione, e nel caso di esito positivo, verrà detto in che posizione della coda si è posizionati. In questo modo chi gestisce le opere del museo, potrà anche abbandonare la pagina, senza dover aspettare che sia finito il processo di generazione del modello

```
app = Flask(__name__)
r=redis.Redis()
q=Queue(connection=r)

@app.route('/generateModel', methods=['GET', 'POST'])
def handle_request_admin():

portrait_name = str(request.args.get('portrait_name'))
description = str(request.args.get('description'))
image = str(request.args.get('image'))

if request.args.get("portrait_name"):
#aggiunta del task alla coda q
job = q.enqueue(background_task,request.args.get("
                                portrait_name"),
      request.args.get("description"),request.args.get("
                                image"))

return "Richiesta messa in coda, lunghezza coda: "+len(q)
```

```
else: return "Errore, riprovare"
```

Di seguito è mostrato il task che viene eseguito rispettando la coda

```
def background_task_admin(portrait_name,description,image):

OUTPUT_DIR="directory"

filename=portrait_name+".jpg"
im=cv2.imwrite(filename, image)

INPUT_IMAGE="directory"+im

command = 'python3 accelerate launch train_imagic.py --
                            pretrained_model_name_or_path=
                            $MODEL_NAME --output_dir=
                            $OUTPUT_DIR --input_image=
                            $INPUT_IMAGE --target_text="{
                            TARGET_TEXT}" --seed=3434554
                            --resolution=512 --
                            mixed_precision="fp16" --
                            use_8bit_adam --
                            gradient_accumulation_steps=1
                            --emb_learning_rate=1e-3 --
                            learning_rate=1e-6 --
                            emb_train_steps=500 --
                            max_train_steps=1000'
command = command.replace('MODEL_NAME', MODEL_NAME)
command = command.replace('OUTPUT_DIR', OUTPUT_DIR)
command = command.replace('INPUT_IMAGE', INPUT_IMAGE)
command = command.replace('TARGET_TEXT', description)
```

```
    os.system(command)

    return
```

3.4 Codice per la generazione delle immagini

Siccome l'utente deve inserire una descrizione del background, viene aggiunto come prefisso dell'input utente la descrizione completa dell'opera, che è stata definita dal museo quando ha generato il modello.
In questo modo si riesce ad avere una manipolazione dell'immagine originale.

```
BASE_URL = 'http://solaris.micc.unifi.it/andreaderosa'
SUBFOLDER='/image/'

TARGET_TEXT="woman with long hair and crossed hand by
                              leonardo da vinci" #
                              descrizione immagine
USER_TEXT="with a mountain as background" #input utente
PORTRAIT_NAME="gioconda" #nome opera, che  serve per aprire
                               la cartella del finetuning

payload = {'prompt':USER_TEXT, 'portrait_name':
                              PORTRAIT_NAME, 'target_text':
                              TARGET_TEXT}
response = rq.get(BASE_URL, params=payload)

img_name=response.text

url=BASE_URL+SUBFOLDER+img_name
response = rq.get(url)
if response.status_code == 200:
with open(img_name, 'wb') as f:
f.write(response.content)
```

Funzione che viene eseguita dopo aver effettuato la chiamata rest.

In questo caso l'utente che genera l'immagine verrà messo in coda e dovrà aspettare per ricevere l'immagine generata prima di avere risposta, ma siccome la generazione di un'immagine richiede molte meno risorse rispetto alla generazione di un modello, l'attesa sarà di qualche decina di secondi, anche se la coda ha un buon numero di richieste.

```
app = Flask(__name__)
r=redis.Redis()
q=Queue(connection=r)

@app.route('/generateImage', methods=['GET', 'POST'])
def handle_request_user():

prompt=str(request.args.get('prompt'))
#testo descrittivo dell'utente

directory=str(request.args.get('directory'))
#directory dove reperire il modello

target_text=str(request.args.get('target_text'))
#testo descrittivo dell'opera, ovvero quello inserito dal
                            gestore del museo durante la
                            creazione del modello

if request.args.get("prompt"):
job = q.enqueue(background_task, request.args.get("prompt")
                            , request.args.get("directory"
                            ),request.args.get("
                            target_text"))

while (image_with_time_string==""):
time.sleep(3)
```

```
return image_with_time_string
```

Di seguito è mostrato il task che viene eseguito per la generazione dell'immagine.

I valori di guidance e il numero di steps sono stati scelti sulla base dei test effettuati su diversi dipinti.

```
image_with_time_string=""

def background_task_user(prompt,directory,target_text):

model_path='path'+directory

#setup per algoritmo di dreambooth
scheduler = DDIMScheduler(beta_start=0.00085, beta_end=0.
                          012, beta_schedule="
                          scaled_linear", clip_sample=
                          False,
set_alpha_to_one=False)
pipe = StableDiffusionPipeline.from_pretrained(model_path,
                          scheduler=scheduler,
                          torch_dtype=torch.float16).to(
                          "cuda")
target_embeddings = torch.load(os.path.join(model_path, "
                          target_embeddings.pt")).to("
                          cuda")
optimized_embeddings = torch.load(os.path.join(model_path,
                          "optimized_embeddings.pt")).to
                          ("cuda")
g_cuda = None

g_cuda = torch.Generator(device='cuda')
```

```
seed = -1
g_cuda.manual_seed(seed)

num_samples = 1
guidance_scale = 5
num_inference_steps = 100
height = 512
width = 512

pipe.safety_checker = None
pipe.requires_safety_checker = False

with autocast("cuda"), torch.inference_mode():
images = pipe(
target_text + " " + prompt,
height=height,
width=width,
num_images_per_prompt=num_samples,
num_inference_steps=num_inference_steps,
guidance_scale=guidance_scale,
generator=g_cuda
).images

image_with_time_string=datetime.now().strftime('%Y-%m-%d %H
                                :%M:%S')+".jpg"
#aggiungo la data con i secondi in modo che il nome dell'
                                immagine generata sia univoco

plt.imshow(images[0])
plt.axis("off")
plt.savefig(image_with_time_string)
return
```

Capitolo 4

Conclusioni

Questo studio si è posto come obiettivo quello di utilizzare l'algoritmo di DreamBooth in un contesto di opere d'arte.
Sebbene gli algoritmi standard di DreamBooth diano buoni risultati su foto di persone o personaggi di cartoni animati, sulle opere d'arte danno spesso problemi in termini di qualità del risultato finale, infatti è molto frequente che lo stile dell'autore non venga rispettato, perché un quadro dipinto a mano avrà una visione diversa da quella di una comune fotografia, oppure un cartone animato costruito a computer.
In alcuni casi applicare uno style-transfer su alcuni dipinti può essere la soluzione, in altri no.

Usare un algoritmo che utilizza una sola immagine, e sulla base di quella va a manipolare i singoli pezzi, ha dato risultati nettamente migliori, tenendo conto che per ogni opera sono stati dati valori di setup uguali sia per la generazione del modello, sia per la generazione dell'immagine, quindi non è stato necessario fare uno studio di parametri ottimali per ogni modello.

C'è sempre da tenere in considerazione che questi algoritmi sono moderni (rilasciati a partire dal 2022), quindi per il momento è normale non si ottengano risultati molto precisi in determinati contesti.