Un esempio di applicazione dei DDPM
Laboratorio Computazionale

Matteo Scardala
m.scardala@studenti.unipi.it

Luglio 2024

Indice

1 Introduzione

I DDPM (Denoising Diffusion Probabilistic Models) sono uno degli strumenti che compongono la famiglia dei metodi generativi. Un metodo generativo è volto a generare dati nuovi che siano verosimili a dati reali come immagini di gatti che siano (per un occhio umano) indistinguibili da vere fotografie di gatti scattate "nel mondo reale".
L'idea che sta alla base di molti metodi generativi è che dati di un stesso tipo si possano modellare come variabili aleatorie IID con una distribuzione incognita: per generare dati nuovi e verosimili è sufficiente trovare questa distribuzione incognita e fare un campionamento.
I DDPM seguono questa logica. In particolare si compongono di due fasi:

1. **Processo di diffusione:** In questa fase un dato reale viene distorto gradualmente aggiungendo del rumore gaussiano passo dopo fino a quando il dato distorto segue (approssimativamente) una distribuzione guassiana
2. **Processo all'indietro:** In questa fase il dato iniziale è campionato da una distribuzione gaussiana isotropa e si ripercorrono all'indietro i passi del processo precedente per riottenere il dato iniziale

Nella sezione 3 si forniscono i dettagli di questo processo
In questo lavoro si presenta un esempio che mostra le fasi del processo tramite un'immagine che viene prima distorta e poi ricostruita a partire da sampling su una distribuzione gaussiana isotropa.

2 Reti Neurali

Una rete neurale è una funzione che si può modellare attraverso un grafo. In particolare noi considereremo reti neurali di tipo feedforward: il grafo è aciclico e i nodi si organizzano in layer (livelli) consecutivi. In questo contesto i nodi del grafo vengono chiamati unità e gli archi del grafo sono pesati. Ogni unità riceve in input un numero reale: la somma pesata (secondo i pesi dei relativi archi) degli output dei nodi della stella entrante. A questo input viene applicata una funzione di attivazione il cui output viene distribuito alle unità della stella uscente secondo i pesi dei realativi archi. Esempi di funzioni di attivazione usate di frequente sono la funzione sigmoidale $\sigma(x) = \frac{1}{1+\exp(-x)}$, la ReLU $ReLU(x) = max(0, x)$ e la tangente iperbolica. Queste funzioni garantiscono una buona espressività della rete neurale in quanto esistono dei teoremi che afferamano che le reti neurali con queste funzioni di attivazione sono dense in $C^0(\mathbb{R}^d, \mathbb{R}^m, ||\cdot||_\infty)$
Le reti neurali possono essere addestrate, cioè è possibile migliorarne le performance fornendo dei dati. Infatti una rete neurale è costruita con un obiettivo (ad esempio calcolare i parametri di una distribuzione). A questo obiettivo si può associare una loss function che misura in un opportuno senso quanto la performance della rete neurale si discosta da quella desiderata. È possibile migliorare i risultati delle rete neurale tramite un algortimo di discesa del gradiente cercando di trovare i valori dei pesi degli archi che minimizzano la loss function. Una proprietà interessante delle reti neurali è che per calcolare il gradiente della loss function è sufficiente calcolare le derivate parziali rispetto ai pesi del layer

di output. Infatti si può sfruttare la chain rule delle derivate per calcolare le derivate parziali rispetto agli altri pesi con semplici manipolazioni algebriche (somme e moltiplicazioni) di quelle che abbiamo già calcolato. In questo caso si parla di back-propagation, cioé il gradiente viene calcolato solo per l'ultimo layer e si ripercorre la rete all'indietro per calcolare il gradiente completo senza svolgere un effettivo calcolo di derivate.

3 Presentazione del modello

Sia x_0 la variabile aleatoria che modella un dato, siano $x_1, \cdots, x_T$ variabili latenti della stessa dimensione di x_0. Queste variabili modellano due processi stocastici a tempo discreto: il processo di diffusione da x_0 a x_T e il processo all'indietro da x_T a x_0. Nei DDPM questi processi sono catene di Markov. In particolare, siano $p_\theta(x_0, \cdots, x_T)$ la distribuzione congiunta tra le variabili e $q(x_1, \cdots, x_T|x_0)$ la distribuzione a posteriori che approssima $p_\theta(x_1, \cdots, x_T|x_0)$. Per la proprietà di Markov vale

$$p_\theta(x_0, \cdots, x_T) = p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \qquad q(x_1, \cdots, x_T|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}) \tag{1}$$

Per quanto detto nell'introduzione vorremmo che alla fine del processo di diffusione la variabile latente segua una distribuzione gaussiana isotropa, dunque poniamo $p(x_T) = \mathcal{N}(0, I)$. Inoltre completiamo la definizione del modello ponendo $p_\theta(x_{t-1}|x_t) = \mathcal{N}(\mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$ e $q(x_t|x_{t-1}) = \mathcal{N}(x_{t-1}\sqrt{1-\beta_t}, \beta_t I)$ dove $\beta_1, \cdots, \beta_T$ sono iperparametri del modello.
In questo ambito vorremmo addestrare una rete neurale a ricostruire i dati a partire da un rumore puramente gaussiano. I parametri della rete neurale sono "catturati" dal parametro θ. Per cercare il miglior valore di θ cerchiamo il MLE $\hat{\theta} = \underset{\theta}{argmin}\mathbb{E}[-\log p_\theta(x_0)]$
Tramite semplici conti:
$\mathbb{E}[-\log p_\theta(x_0)] \le \mathbb{E}_q[-\log \frac{p_\theta(x_0, \cdots, x_T)}{q}] = \mathbb{E}_q[-\log p(x_T) - \sum_{t\ge 1} \log \frac{p_{\theta|x_t}}{q(x_t|x_{t-1})}] =$

$= \mathbb{E}_q[D_{KL}(q(x_T|x_0)||p(x_T)) + \sum_{t>1} D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1)] \doteq L$
L viene chiamato variational upper bound e in questo caso per rendere il problema trattabile cerchiamo il valore di θ che minimizza il VUB invece che minimizzare direttamente la (meno)log-likelihood.
Siano $\alpha_t \doteq 1 - \beta_t$ e $\bar{\alpha}_t \doteq \prod_{s=1}^{t} \alpha_s$. Si può mostrare che valgono le seguenti:

$$q(x_t|x_0) = \mathcal{N}(x_0\sqrt{\bar{\alpha}_t}, (1-\bar{\alpha}_t)I) \tag{2}$$

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}(\tilde{\mu}_t, \tilde{\beta}_t I) \tag{3}$$

dove

$$\tilde{\mu}_t \doteq \frac{\beta_t\sqrt{\bar{\alpha}_t}}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}x_t \qquad \tilde{\beta}_t \doteq \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_{t-1}}\beta_t \tag{4}$$

Grazie a queste espressioni e al fatto che la divergenza di KL tra due distribuzioni gaussiane ha una forma chiusa in funzione di media e varianza è possibile calcolare esplicitamente il VUB. Infatti siano $p(x) = \mathcal{N}(\mu_1, \Sigma_1)$ e $q(x) = \mathcal{N}(\mu_2, \Sigma_2)$ distribuzioni gaussiane $d-$dimensionali, risulta:

$$D_{KL}(p||q) = \frac{1}{2}[\log \frac{|\Sigma_2|}{|\Sigma_1|} - d + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T \Sigma_2^{-1}(\mu_2 - \mu_1)] \tag{5}$$

Nell'esperimento supponiamo che $q(x_{t-1}|x_t, x_0)$ e $p_\theta(x_{t-1}|x_t)$ abbiano stesse matrice delle covarianze (diagonale).
Da ciò risulta che $D_{KL}(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) \propto ||\tilde{\mu}_t - \mu_\theta(t)||_2^2 + cost.$ e dunque ha senso considerare come loss function l'errore quadratico medio tra $\tilde{\mu}_t, \mu_\theta(t)$.

3.1 Giustificazione del modello

In questa sezione viene fornita una breve giustificazione del modello, in particolare perché le distribuzioni definite come sopra dovrebbero definire un processo di diffusione e un processo all'indietro.
Il processo di diffusione può essere modellato dalla seguente

$$x_t = (1-\tau)x_{t-1} + \sqrt{2\tau}Z \qquad Z \sim \mathcal{N}(0, I) \qquad \tau << 1 \tag{6}$$

cioé ad ogni passo il dato viene moltiplicato per un numero leggermente più piccolo di uno e viene aggiunto del rumore gaussiano.
Si nota che questa è la discretizzazione di Eulero-Maruyama del processo stocastico continuo

$$dX_t = -X_t dt + \sqrt{2}dB_t \tag{7}$$

Si può mostrare che

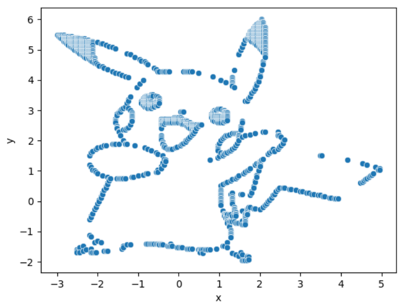
$$X_t \underset{legge}{=} e^{-t}X_0 + \sqrt{1 - e^{-2t}}Z \qquad Z \sim \mathcal{N}(0, I) \tag{8}$$

dunque ponendo $\bar{\alpha}_t = e^{-2t}$ vale $p(x_t|x_0) \sim (\mathcal{N}(x_0\sqrt{\bar{\alpha}_t}, (1-\bar{\alpha}_t)I)$
Similmente si possono giustificare le altre distribuzioni

4 L'esperimento

L'esperimento consiste nel mostrare come funziona un processo di diffusione tramite un programma in python. Di seguito si farà riferimento al codice che viene riportato nella sezione seguente
Si parte dall'immagine di un noto personaggio dei cartoni animati (riportata in seguito)

Questa immagine viene semplificata, cioé ridotta a un immagine in bianco e blu, tramite la funzione `scatter_pixels` (Listing 5). Il risultato è il seguente



In seguito definiamo gli iperparametri del modello:

- `beta_start` è β_0
- `beta_end` è β_T
- `num_diffusion_timesteps` è il numero di stati dei due processi stocastici
- `betas` array che contiene i valori di β_t che abbiamo scelto
- `alphas` array che contiene i valori di α_t
- `list_bar_alphas` lista che contiene i valori di $\bar{\alpha}_t$ calcolati come spiegato nella sezione precedente
- `training_steps_per_epoch` numero di iterazioni per addestrare la rete neurale per epoch (nota: quando si addestra una rete neurale i dati del dataset vengono forniti alla macchina diverse volte in loop. Un'epoch corrisponde a un'iterazione di questo loop)
- `pbar= tqdm(range(n))` numero di epoch: n

A questo punto viene creata e addestrata una rete neurale per cercare di ricostruire l'immagine di partenza a partire da un rumore puramente gaussiano. In questo esperimento è stata utilizzata la libreria `pytorch` che permette di creare facilmente reti neurali. Queste vengono definite come classi python i cui attributi sono le funzioni di attivazione e i layer e che hanno un metodo `forward` che definisce l'applicazione della rete a un dato. In Listing 1 viene riportata la

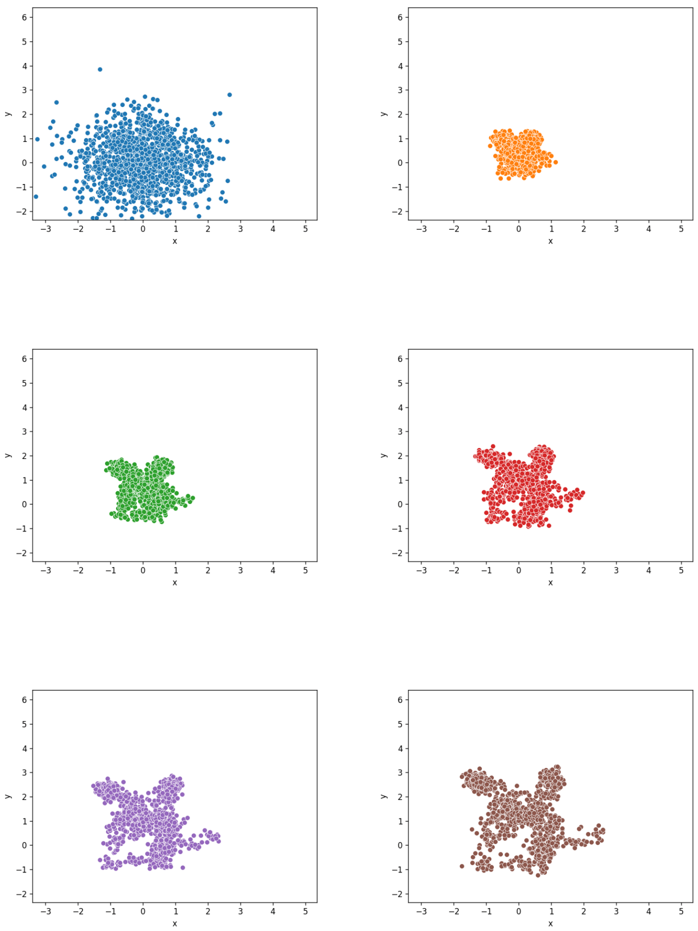
rete neurale `Denoising`, il modello di quella addestrata nell'esperimento. Questa rete prende in input un dato distorto e un tempo t (che rappresenta il tempo in cui si trova il processo) e resituisce la media della distribuzione $p_\theta(x_{t-1}|x_t)$.
Per l'addestramento abbiamo dovuto

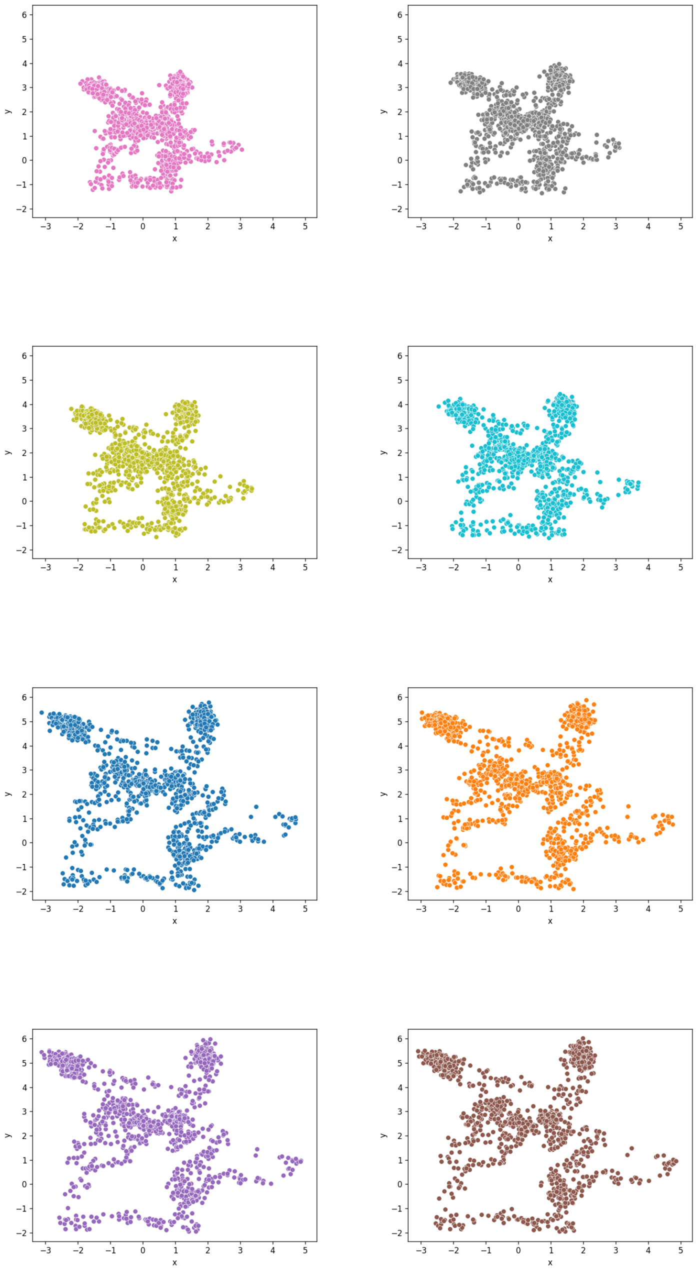
- specificare la loss fuction tramite `criterion = nn.MSELoss()`. In questo caso è l'errore quadratico medio come anticipato nella sezione 3.
- definire la rete neurale tramite `denoising_model = Denoising(DATA_SIZE, num_diffusion_timesteps).to(device)`. Da questo momento in poi la rete neurale sarà identificata da `denoising_model`
- il metodo di discesa del gradiente tramite `optimizer = optim.AdamW(denoising_model.parameters())`

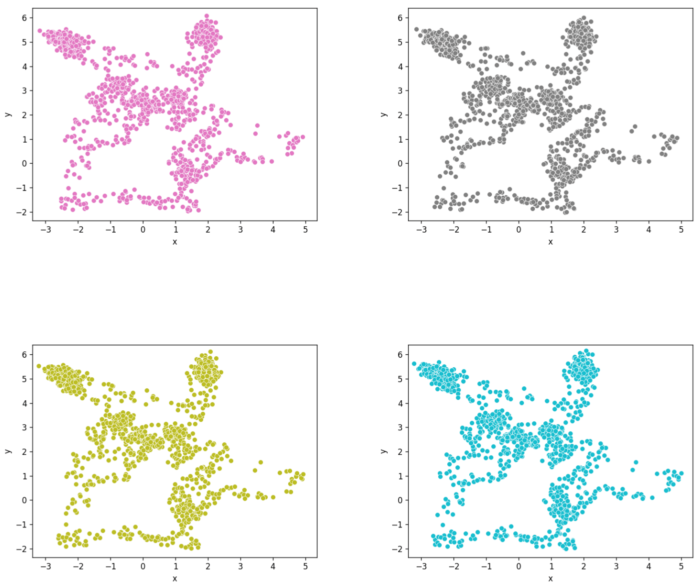
Nel ciclo contraddistinto da `for epoch in pbar` viene addestrata la rete neurale. Si spiega in modo dettagliato cosa accade nel ciclo:

- `running_loss = 0.0` : viene azzerata la loss function
- `Ts = np.random.randint(1, num_diffusion_timesteps, size=training_steps_per_epoch)` : vengono scelti in modo casuale un numero pari a `training_steps_per_epoch` di passi temporali del processo e su questi si itera il processo di addestramento tramite il ciclo `for _, t in enumerate(Ts)`
- `q_t = q_sample(x_init, t, list_bar_alphas, device)` : campiona un dato dalla distribuzione $q(x_t|x_0)$. La funzipne `q_sample` è riportata in Listing 2: per fare il campionamento usa la formula (2)
- `mu_t, cov_t = posterior_q(x_init, q_t, t, alphas, list_bar_alphas, device)`: calcola $\tilde{\mu}_t$ e $\tilde{\beta}_t I$ usando la formula (4). Il codice della funzione `posterior_q` è riportato in Listing 3
- `sigma_t`: salva il valore di $\tilde{\beta}_t$
- `optimizer.zero_grad()` azzera il gradiente. In pytorch il gradiente è una variabile che non viene dichiarata, ma è il programma che lo memorizza autonomamente
- `mu_theta= denoising_model(q_t, t)`: viene applicata la rete neurale al dato distorto al tempo t. Viene restituita dunque $\mu_\theta(t)$
- `loss = criterion(mu_theta, mu_t)`. Calcola l'errore quadratico medio tra $\mu_\theta(t)$ e $\tilde{\mu}_t$.
- `loss.backward()` calcola il gradiente della loss function come spiegato nella sezione sulle reti neurali. Quando viene applicato il metodo `forward` di una rete neurale il dato resituito oltre al valore numerico ha anche un attributo `grad_fn`. Tramite questo attributo quanto viene invocata `loss.backward()` l'interpete riesce a tracciare da dove viene il dato fino a recuperare la rete neurale che l'ha generato e quindi riesce a calcolare il gradiente in funzione dei parametri di questa rete neurale.
- `optimmizer.step()` viene fatto un passo di discesa del gradiente aggioranando i parametri della rete neurale.

Finito l'addestramento della rete neurale viene preso un dato campionato da una gaussiana isotropa tramite
`data = torch.distributions.MultivariateNormal(loc=torch.zeros(DATA_SIZE), covariance_matrix=torch.eye(DATA_SIZE)).sample().to(device)`
Nel successivo ciclo `for d in range(1, num_diffusion_timesteps)` il dato viene gradualmente ricostruito tramite la funzione `denoise_with_mu` il cui codice è riportato in Listing 4. Questa funzione applica la rete neurale a un dato x_t, calcola la distribuzione $p_\theta(x_{t-1}|x_t)$ e resituisce un dato meno distorto x_{t-1} tramite campionamento sulla distribuzione
A ogni iterazione viene fatto un plot del dato che viene memorizzato per costruire una gif che rappresenti questo processo. Si presenta la gif scomposta in ordine temporale:







5 Codice

Listing 1: Paradigma della rete neurale

```
def position_encoding_init(n_position, d_pos_vec):

    position_enc = np.array([
        [pos / np.power(10000, 2 * i / d_pos_vec) for i in
            range(d_pos_vec)]
        if pos != 0 else np.zeros(d_pos_vec) for pos in
            range(n_position)])

    position_enc[1:, 0::2] = np.sin(position_enc[1:, 0::2]) # dim 2i
    position_enc[1:, 1::2] = np.cos(position_enc[1:, 1::2]) # dim 2i+1
    return torch.from_numpy(position_enc).to(torch.float32)

class Denoising(torch.nn.Module):

    def __init__(self, x_dim, num_diffusion_timesteps):
        super(Denoising, self).__init__()

        self.linear1 = torch.nn.Linear(x_dim, x_dim)
        self.emb = position_encoding_init(num_diffusion_timesteps, x_dim)
        self.linear2 = torch.nn.Linear(x_dim, x_dim)
        self.linear3 = torch.nn.Linear(x_dim, x_dim)
        self.relu = torch.nn.ReLU()

    def forward(self, x_input, t):
        emb_t = self.emb[t]
        x = self.linear1(np.add(x_input,emb_t))
        x = self.relu(x)
```

```
        x = self.linear2(x)
        x = self.relu(x)
        x = self.linear3(x)
        return x
```

Listing 2: funzione q sample

```
    def q_sample(x_start, t, list_bar_alphas, device):

      alpha_bar_t = list_bar_alphas[t]

      mean = (alpha_bar_t ** 0.5 )* x_start
      cov = torch.eye(x_start.shape[0]).to(device)
      cov = cov * (1 - alpha_bar_t)
      return torch.distributions.MultivariateNormal(loc=mean,
          covariance_matrix=cov).sample().to(device)
```

Listing 3: funzione posterior q

```
def posterior_q(x_start, x_t, t, list_alpha, list_alpha_bar, device):

    beta_t = 1 - list_alpha[t]
    alpha_t = list_alpha[t]
    alpha_bar_t = list_alpha_bar[t]
    # alpha_bar_{t-1}
    alpha_bar_t_before = list_alpha_bar[t - 1]

    # calcola mu_tilde
    first_term = x_start * torch.sqrt(alpha_bar_t_before) * beta_t / (1
        - alpha_bar_t)
    second_term = x_t * torch.sqrt(alpha_t) * (1 - alpha_bar_t_before) /
        (1 - alpha_bar_t)
    mu_tilde = first_term + second_term

    # beta_t_tilde
    beta_t_tilde = beta_t * (1 - alpha_bar_t_before) / (1 - alpha_bar_t)

    cov = torch.eye(x_start.shape[0]).to(device) * beta_t_tilde

    return mu_tilde, cov
```

Listing 4: funzione denoise with mu

```
def denoise_with_mu(denoise_model, x_t, t, list_alpha, DATA_SIZE,
    device):

    alpha_t = list_alpha[t]
    beta_t = 1 - alpha_t
    mu_theta = denoise_model(x_t, t)
    x_t_before = torch.distributions.MultivariateNormal(loc=mu_theta,
        covariance_matrix=torch.diag(
       beta_t.repeat(DATA_SIZE))).sample().to(device)
```

```
    return x_t_before
```

Listing 5: funzioni utili

```
def scatter_pixels(img_file):
    w = IMG_SIZE
    img = Image.open(img_file).resize((w, w)).convert("L")
    pels = img.load()
    black_pels = [(x, y) for x in range(w) for y in range(w)
                    if pels[x, y] <= 50]
    return [t[0] for t in black_pels], [w - t[1] for t in black_pels]

def pack_data(x, y):
    """
    pack 2d data to 1d vector
    """
    one_d_data = []
    for i in range(len(x)):
        one_d_data.append(x[i])
        one_d_data.append(y[i])

    return one_d_data

def unpack_1d_data(one_d_data):
    """
    unpack 1d data to 2d vector
    """
    x = []
    y = []
    for i in range(len(one_d_data)):
        if i % 2 == 0:
            x.append(one_d_data[i])
        else:
            y.append(one_d_data[i])
    return x, y
```

Listing 6: Codice

```
import torch
import numpy as np
from diffusion import q_sample, posterior_q, Denoising, denoise_with_mu
from utils import pack_data, unpack_1d_data, scatter_pixels
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from operator import mul
from functools import reduce
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm

#se si dispone di un'opportuna scehda grafica i dati verrano mandati a
    questa per risultati piu veloci
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# converto l'immagine in puntini con scatter pixels e la plotto
x, y = scatter_pixels('homer.png')
x = [x/25 - 3 for x in x]
y = [y/25 - 2 for y in y]
df = pd.DataFrame({'x': x,
                  'y':y
                  })
ax = sns.scatterplot(data=df,x='x',y='y')
plt.show()
## Salvo gli assi per plottare dopo
y_ax = ax.get_ylim()
x_ax = ax.get_xlim()
axes = (x_ax,y_ax)

# mando i dati al device
one_d_data = pack_data(x,y)
x_init = torch.tensor(one_d_data).to(torch.float32).to(device)

DATA_SIZE = len(x_init)


#Parametri di diffusione

beta_start = .0004
beta_end = .02
num_diffusion_timesteps = 30
betas = np.linspace(beta_start ** 0.5, beta_end ** 0.5,
    num_diffusion_timesteps) ** 2
print(betas)
alphas = 1 - betas
# mando parametri al device
betas = torch.tensor(betas).to(torch.float32).to(device)
alphas = torch.tensor(alphas).to(torch.float32).to(device)
list_bar_alphas = [alphas[0]]
for t in range(1, num_diffusion_timesteps):
    list_bar_alphas.append(reduce(mul, alphas[:t]))

list_bar_alphas = torch.cumprod(alphas,
    axis=0).to(torch.float32).to(device)
training_steps_per_epoch = 40

#definisco loss function e metodo di discesa del gradiente e la rete
    neurale
criterion = nn.MSELoss()
denoising_model = Denoising(DATA_SIZE,
    num_diffusion_timesteps).to(device)
denoising_model.emb = denoising_model.emb.to(device)
optimizer = optim.AdamW(denoising_model.parameters())

#training della rete neurale
pbar = tqdm(range(20))
for epoch in pbar:
```

```
    running_loss = 0.0
    Ts = np.random.randint(1, num_diffusion_timesteps,
         size=training_steps_per_epoch)
    for _, t in enumerate(Ts):
        q_t = q_sample(x_init, t, list_bar_alphas, device)
        mu_t, cov_t = posterior_q(x_init, q_t, t, alphas,
             list_bar_alphas, device)
        optimizer.zero_grad()
        mu_theta = denoising_model(q_t, t)
        loss = criterion(mu_theta, mu_t)
        loss.backward()
        optimizer.step()
        running_loss += loss.detach()
    pbar.set_description('Epoch: {} Loss: {}'.format(epoch, running_loss
         / training_steps_per_epoch))
print('Finished Training')

#genera dato casuale
data =
     torch.distributions.MultivariateNormal(loc=torch.zeros(DATA_SIZE),covariance_matrix=torch.eye(DA

#quest'ultima parte serve a creare la gif che mostra il procedimento
from celluloid import Camera
import pandas as pd
import matplotlib.pyplot as plt

fig = plt.figure()
camera = Camera(fig)

for d in range(1, num_diffusion_timesteps):
    data =
         denoise_with_mu(denoising_model,data,num_diffusion_timesteps-d,
         alphas, DATA_SIZE, device)
    data_plot = data.detach().cpu().numpy()
    x_new, y_new = unpack_1d_data(data_plot)
    df_new = pd.DataFrame({'x': x_new,
                        'y': y_new
                        })

    graph = sns.scatterplot(data=df_new, x='x', y='y')
    plt.pause(0.1)
    graph.set_xlim(axes[0])
    graph.set_ylim(axes[1])

    camera.snap()

anim = camera.animate(blit=False)
anim.save('output.gif',fps=24, dpi=120)
```