

project work generative models

paolo marino

February 2024

Contents

1	Introduzione	2
2	Strumenti utilizzati	2
3	Struttura progetto	2
4	Implementazione modello	3
4.1	Backbone	3
4.1.1	Generatore	3
4.1.2	Discriminatore	4
4.1.3	Critico	4
4.2	Training	5
5	Risultati	5
6	Conclusione	7

1 Introduzione

L'obiettivo di questo lavoro è cercare di re-implementare una versione dell'HP-GAN: una fusione tra una gan standard e una wgan-gp dove la prima si occupa esclusivamente di valutare la qualità del modello e la seconda oltre valutare la qualità, ha il compito di allenare il generatore. La versione dell'hp gan è una versione semplificata in cui si fa a meno di due loss implementate nel paper (la consistency loss e la bone loss) che ho provato a sostituire con una nuova loss data dal ground truth









2 Strumenti utilizzati

Per implementare questa rete ho usato:

- Pytorch
- Dataset nturgbd
- Librerie prese dall'implementazione originale del paper (<https://github.com/ebarsoum/hpgan>) per leggere le sequenze
- File cameras.h5 che serve contiene i parametri della camera per proiettare le pose dal 3d al 2d

3 Struttura progetto

La struttura della cartella è la seguente:

 braniac	14/04/2023 22:33	Cartella di file	
 skeleton	14/04/2023 18:57	Cartella di file	
 skeleton_images	21/02/2024 17:38	Cartella di file	
 splitted_skeleton	14/04/2023 22:43	Cartella di file	
 cameras.h5	09/05/2023 10:33	File H5	96 KB
 HP_Gan	21/02/2024 17:38	File di origine Jupy...	97 KB
 split_data	14/04/2023 16:29	File di origine Pyth...	7 KB
 utils	20/02/2024 22:08	File di origine Pyth...	1 KB

- **Braniac:** contiene le librerie importate dal progetto originale
- **Skeleton:** contiene i dati riguardo le pose umane
- **Skeleton images:** dove salvo le immagini generate

- **Splitted skeleton:** contiene i file di training e di test
- **Cameras.h5**
- **HP Gan:** Foglio di lavoro principale che contiene l'implementazione della rete e il training di questa
- **Split data:** File python che divide i dati in train e test
- **Utils:** File python che contiene l'implementazione del gradient penalty

4 Implementazione modello

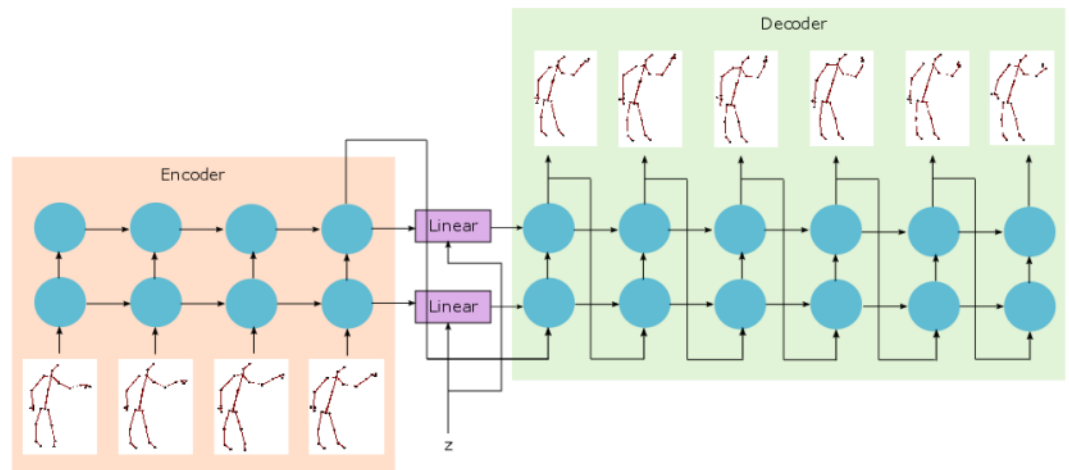
4.1 Backbone

Il backbone è formato da 3 parti:

- Generatore
- Discriminatore
- Critico

4.1.1 Generatore

Il generatore è una rete sequence to sequence che prende in input una serie di pose in input e deve generare pose successive



Le pose in input sono 10 per l'esattezza e deve generare 20 pose che corrispondono ai 20 movimenti successivi.

L'encoder è formato da una semplice GRU a 2 layers a cui passo la sequenza di input.

Il decoder prende l'output del generatore e gli strati nascosti che vengono concatenati con un vettore generato da una distribuzione uniforme, denominato z sempre ad una gru a due layer dove ogni cella del primo layer prende in input la sequenza precedente

```
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers=2):
        super(Encoder, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size, num_layers=num_layers, batch_first=True)

    def forward(self, input_seq):
        output, hidden = self.gru(input_seq)
        # ultimo hidden state dell'ultimo layer
        output = output[:, -1, :]
        return output, hidden

class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.linear_hidden = nn.Linear(2*hidden_size, output_size)
        self.linear_out = nn.Linear(hidden_size, output_size)
        self.gru1 = nn.GRUCell(output_size, output_size)
        self.gru2 = nn.GRUCell(output_size, output_size)

    def forward(self, x, hidden, z):
        hidden_1 = torch.cat([hidden[0, :], z], dim=1)
        hidden_2 = torch.cat([hidden[1, :], z], dim=1)
        hidden_1 = self.linear_hidden(hidden_1)
        hidden_2 = self.linear_hidden(hidden_2)
        out = self.linear_out(x)
        decoded_sequence = []
        for i in range(20):
            hidden_1 = self.gru1(out, hidden_1)
            hidden_2 = self.gru2(hidden_1, hidden_2)
            out = hidden_2
            decoded_sequence.append(out)

        decoded_sequence = torch.stack(decoded_sequence, dim=1)
        return decoded_sequence
```

4.1.2 Discriminatore

Il discriminatore è formato da tre semplici layer lineari intervallati da due leaky relu termina con una sigmoide che serve a intervallare l'output tra 0 e 1 in modo da avere una probabilità come output

4.1.3 Critico

Il critico ha esattamente la medesima struttura del discriminatore senza la sigmoide

4.2 Training

Il training avviene prendendo un batch alla volta che è un tensore che contiene 16 batch e 30 pose umane. Questo tensore viene diviso in due tensori che ho denominato prior poses, ovvero le prime 10 e future poses, ovvero le ultime 20. Queste prime 10 pose sono quelle che passo per input al generatore e le altre sono quelle che uso come ground truth.

Una volta creati i tensori e il rumore z vado a generare le pose ed allenare il critico

```
#Train critic
for _ in range(CRITIC_ITERATIONS):
    #genero pose
    fake_poses = generator(prior_poses.view(prior_poses.shape[0], prior_poses.shape[1], -1), z_data)
    fake_poses = fake_poses.view(fake_poses.shape[0], 20, 25, 3)
    fake_poses_generated = fake_poses.detach()
    fake_poses = torch.cat((prior_poses, fake_poses), axis=1)
    critic_real = critic(real_poses.view(real_poses.shape[0], -1)).view(-1)
    critic_fake = critic(fake_poses.view(fake_poses.shape[0], -1)).view(-1)
    gp = gradient_penalty(critic, real_poses, fake_poses)
    lossC = -(torch.mean(critic_real) - torch.mean(critic_fake)) + LAMBDA_GP*gp
    critic.zero_grad()
    lossC.backward(retain_graph=True)
    opt_critic.step()
```

Poi c'è il discriminatore il cui obiettivo è semplicemente quello di valutare le pose generate e infine uso la loss del ground truth per provare a generare pose più realistiche

5 Risultati

Esempi di pose generate nel tempo

- Pose di input passate al generatore :



- Epoca 50:



- Epoca 80:



- Epoca 200:



- Epoca 500:



- Epoca 900:



- Differenza tra pose generate e ground truth

- Pose di input:



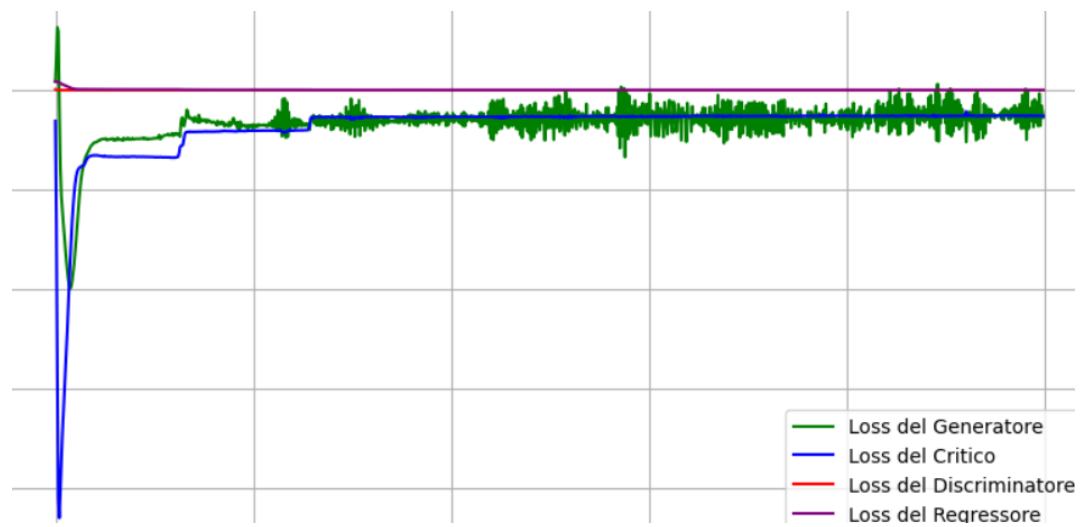
- Pose ground truth:



- Pose generate:



Andamento delle Loss:



6 Conclusione

Da come si può vedere dai risultati le pose generate seguono una sorta di logica in base a quelle precedenti ma non sono molto accurate e spesso si vedono arti che non sono regolari e prendono una loro direzione.

Nel paper originale si usavano altre due loss:

- Consistency loss: una loss che viene usata per creare pose simili a quelle precedenti in modo che la sequenza non presenti pose troppo dissimili tra di loro
- Bone loss: loss usata per mantenere le giuste porporzioni tra le "ossa" degli scheletrini generati

Ho provato a sostituire queste due loss con una loss data dal **ground truth**: le pose, come ci si poteva aspettare, seguono il ground truth e si nota un miglioramento però le proporzioni tra gli arti rimangono irregolari. In conclusione questa trovata, forse, può essere una buona idea per sostituire la consistency loss ma non può farlo con la bone loss perché le proporzioni tra arti migliorano leggermente ma comunque rimane un margine di errore troppo ampio