Dance Start TP - Report

Abner Astete

November 17, 2024

Introduction

The project introduces concepts in Machine Learning and Computer Vision, particularly Generative Adversarial Networks (GANs).

The objective is to synthesize a video of the target person mimicking the movements of the source person. The process involves extracting skeleton data from videos using a pre-trained Mediapipe model and leveraging it to guide image generation.

The implementation progresses through three stages:

1. A nearest-neighbor approach that matches skeletons to images in the dataset.

Figure 1: Nearest-neighbor approach for skeleton matching.

2. A direct neural network to generalize image synthesis from skeleton data.

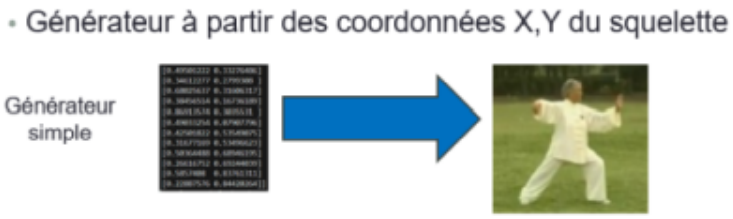

Figure 2: Neural network for image data.

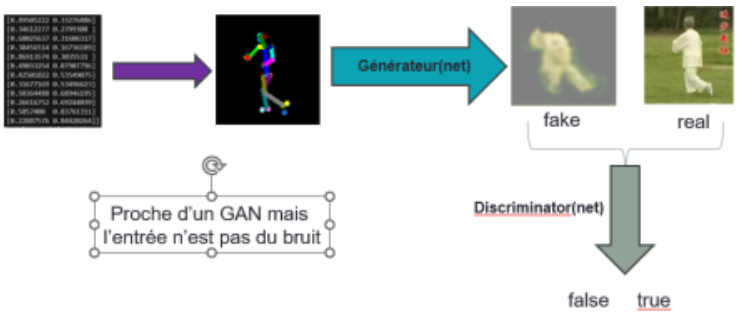

Figure 3: GAN model.

3. A GAN model to enhance the quality and realism of generated images.

How to Run the Code

To run the trained models, we focus on the 'DanceDemo.py' file. Each generator is associated with an assigned number, and to select the desired generator, the 'GEN_TYPE' variable must be modified within the 'main' function of the code. Additionally, the code provides a function to display a comparison between the original video and the one generated by the model.

Training the Neural Networks

To train the neural networks, we have three main scripts responsible for implementing each of the generators:

- **GenNearest.py**: This approach relies on calculating the distance between the *skeleton* and the *video_skeleton*. Since it does not utilize neural networks, this script does not require a training phase.
- **GenVanilla.py** and **GenGan.py**: Both scripts implement neural networks and include a main function (`main`) where training parameters, such as the number of epochs, can be configured. They also allow users to choose whether to train the model from scratch or use a previously trained model.

1 Details of Training in `GenVanilla.py`

The `GenVanilla.py` script uses an Adam optimizer and a Mean Squared Error (MSE) loss function. The training process in `GenVanilla.py` consists of the following steps:

1. **Setup**: The model is set to training mode (`self.netG.train()`), and the Adam optimizer and MSE loss function are initialized.

2. **Training Loop**: For each epoch:
    - The dataset is passed to the model in batches.
    - The loss is computed by comparing the model's predictions with the ground truth data.
    - Backpropagation is performed to adjust the model's parameters.
3. **Model Saving**: At the end of each epoch, the trained model is saved for future use.

2 Details of Training in `GenGan.py`

The `GenGan.py` script implements a training process that uses a Binary Cross-Entropy Loss (BCELoss) function along with two Adam optimizers, one for the generator (`netG`) and another for the discriminator (`netD`). The training process involves alternating updates between these two models to achieve an adversarial training setup.

1. **Setup**: The loss function (`BCELoss`) and Adam optimizers are initialized for both the generator and discriminator. Key parameters such as learning rates and `beta` values are set for stability in training.
2. **Training Loop**: For each epoch:
    - **Discriminator Training**:
        - The discriminator is trained to classify real images correctly. Real images are passed to the discriminator, and a real label is used to compute the loss (`errD_real`).
        - Fake images are generated using the generator, and the discriminator is tasked with classifying them as fake. A fake label is used to compute the loss (`errD_fake`).
        - Both losses are backpropagated, and the discriminator's optimizer (`optimizerD`) updates the discriminator's parameters.
    - **Generator Training**:
        - The generator is trained to create images that can fool the discriminator into classifying them as real.
        - Fake images are generated and passed to the discriminator, and a real label is used to compute the generator loss (`errG`).
        - The loss is backpropagated, and the generator's optimizer (`optimizerG`) updates the generator's parameters.
3. **Model Saving**: At the end of each epoch, a checkpoint containing the states of both the generator and discriminator is saved for future use.

3 BCELoss vs MSELoss

3.1 BCELoss (Binary Cross-Entropy Loss)

BCELoss is used for binary classification tasks, where the expected output is in the probability range $[0, 1]$. In the case of Generative Adversarial Networks (GANs), BCELoss is ideal for evaluating the probability that a sample is "real" or "fake". The discriminator uses this loss to classify images, and the generator aims to fool the discriminator into classifying its generated images as real.

3.2 MSELoss (Mean Squared Error Loss)

MSELoss is used for regression tasks, where the output is a continuous value or a reconstruction that should approximate an expected value. In `GenVanilla.py`, the goal is to synthesize images from input data, such as skeletons. The model is not performing classification but is instead trying to generate an output (image) that is as close as possible to the target image. The MSELoss computes the squared difference between the generated and the target images, penalizing large differences.

Implementation of generators

Before talking about how neural networks were created, I would like to explain several preprocessing steps are necessary both in **GenVanilla** and **GenGan**:

- **Resizing:** The images are resized to a standard size of 64x64 pixels. This ensures that all images have the same size, which is essential for the network to process them efficiently.
- **Center Cropping:** The images are cropped from the center, which helps focus on the most important part of the image, usually the area where the human figure is located, eliminating irrelevant areas.
- **Conversion to Tensors:** The images are converted to tensors, which are data structures that can be processed by the neural network. Tensors are essential for training and evaluation in neural networks.
- **Normalization:** The pixel values of the images are normalized to be in the range of -1 to 1. This is done by subtracting 0.5 from the pixel values and dividing by 0.5, which helps make the training more stable and efficient.
- **DataLoader Preparation:** A DataLoader is created, responsible for feeding the images into the model in batches of 32, which improves efficiency during training and evaluation. Additionally, it ensures that the data is shuffled randomly in each iteration, which helps prevent overfitting and improves model generalization.

1 GenNearest

The process starts with a video containing several postures of a person, each represented as a skeleton, which is basically a collection of key points on the body, such as the joints. When you provide the system with a new skeleton, the goal of the program is to search the video to find which posture (skeleton) is most similar to the skeleton you provided.

The system iterates through all the skeletons in the video, one by one, comparing them with the new skeleton. To make this comparison, it calculates a "distance" between the skeleton you provided and each skeleton in the video. This distance measures how similar the positions of the joints are between the two skeletons. The smaller this distance is, the more similar the skeletons are.

As the program goes through the skeletons, it keeps track of the image that contains the skeleton most similar to the input. Each time it finds a skeleton with a smaller distance than the previous one, it updates the best match.

At the end of the process, the system selects the image of the frame whose skeleton is the most similar to the new skeleton and returns it as the result. If it does not find any similar posture, it would return a blank or error image.

2 GenVanillaNN

GenVanillaNN is a simple neural network designed to generate an image from a skeleton, which is a digital representation of a human posture.

The `GenNNSkeToImage` class is a neural network model designed to generate an image from a skeleton. Function of this network is to transform a skeleton's data (a set of joint positions) into an image that visually represents the human posture.

- **Initialization:** The class is initialized by setting the input dimension based on the skeleton's reduced dimensionality. The model architecture consists of several fully connected layers (linear layers) followed by activation functions (ReLU). The layers progressively increase in size and eventually output a tensor that represents the pixel values of an image.
- **Model Architecture:**
  - The first layer takes the input skeleton and maps it to a 256-dimensional space.
  - A ReLU activation function is applied to introduce non-linearity and allow the network to learn more complex patterns.
  - The subsequent layers progressively expand the feature map (512, then 1024), with ReLU activation applied after each linear layer.
  - The final linear layer outputs a tensor of size $64 \times 64 \times 3$, which corresponds to an image with dimensions 64x64 pixels and 3 color channels (RGB).
  - The `Tanh` activation function is used in the final layer to scale the output image pixels to a range between -1 and 1.
- **Forward Pass:** During the forward pass, the input skeleton (a tensor of size `[batch_size, input_dim]`) is passed through the network. The network generates an image, which is reshaped into the dimensions of $[\texttt{batch\_size}, 3, 64, 64]$ (i.e., a batch of images with 3 color channels and 64x64 pixels).

3 GenGAN

In a Generative Adversarial Networks (GANs), Its primary function is to distinguish between real and generated (fake) images. Determining whether each image is real (from the dataset) or fake (generated by the network).

- **Initialization:** The model architecture consists of several convolutional layers (Conv2D), followed by activation functions, batch normalization layers, and finally, a sigmoid function to output a probability that indicates whether the image is real or fake.

- **Model Architecture:**
  - The first convolutional layer takes in an image with 3 channels (RGB) and applies a kernel of size 4, with a stride of 2 and padding of 1. This reduces the spatial dimensions while increasing the number of feature maps to 64.
  - A Leaky ReLU activation function is used with a negative slope of 0.2 to introduce non-linearity, allowing the network to learn complex patterns even from negative inputs.
  - Each subsequent convolutional layer further reduces the spatial size (by using a stride of 2) while increasing the number of feature maps to 128, 256, and 512 in the following layers. Batch normalization is applied after each of these layers to stabilize training and speed up convergence.
  - The final convolutional layer reduces the feature map to a single value, outputting a 1D tensor representing the probability that the input image is real.
  - The `Flatten` layer converts the output of the final convolutional layer (which has multiple dimensions) into a 1D tensor, making it suitable for classification.
  - The final activation function is `Sigmoid`, which squashes the output to a value between 0 and 1, representing the probability that the input image is real.
- **Forward Pass:** During the forward pass, the input image (a tensor with shape `[batch_size, 3, 64, 64]`) is passed through the layers of the model. After applying all the convolutional layers and activations, the output is a single value between 0 and 1 for each image, indicating whether the image is real (close to 1) or fake (close to 0).