Pose Prediction

LSTM :

LSTM (Long Short-Term Memory)

- **What**:
  LSTM is a type of **recurrent neural network (RNN)** that is designed to better capture long-range dependencies in sequential data. It addresses the vanishing gradient problem that traditional RNNs face, allowing it to remember information for longer periods.
- **Why**:
  LSTMs are useful when working with sequences where the context of previous elements affects the output, such as in time series, speech recognition, and language modeling. They can "remember" previous time steps or elements for longer periods, which is crucial for many sequence-based tasks.
- **When**:
  You would use LSTMs when you have data that follows a sequence, such as:
  - **Time series forecasting** (e.g., predicting stock prices, weather)
  - **Speech and text generation**
  - **Machine translation** (e.g., translating text from one language to another)
  - **Video processing** (e.g., action recognition in videos)

VAE :

VAE (Variational Autoencoder)

- **What**:
  VAE is a **generative model** that learns to represent high-dimensional data in a lower-dimensional latent space. It combines **autoencoders** (which learn to compress and reconstruct data) with **variational inference** to generate new samples similar to the input data. VAEs are often used to learn data distributions and generate new data points.
- **Why**:
  VAEs are useful when you want to generate new data samples that resemble the input data, like images, text, or audio. Unlike traditional autoencoders, VAEs learn probabilistic distributions, making them capable of generating new data by sampling from the learned distribution.
- **When**:
  You would use a VAE when working with:
  - **Unsupervised learning** tasks where you want to learn a good representation of the data (e.g., clustering, dimensionality reduction)
  - **Data generation** tasks, such as creating new images, music, or text that resembles a training dataset

    - **Anomaly detection**, since the model learns the distribution of normal data and can flag outliers

1. Introduction

This project focuses on predicting human poses using deep learning. We use the **3DPW dataset** for training, which includes video sequences with 3D human pose annotations. The goal is to predict future poses based on past data.

2. Dataset

The **3DPW dataset** was chosen because it provides real-world walking scenes and accurate 3D pose annotations, making it ideal for training models to predict human movement. Additionally, the dataset is easier to use compared to others due to its well-organized structure and convenient annotations.

The dataset is structured as **JSON files**, where each file corresponds to a specific video sequence. Each JSON file contains a list of frames, with each frame representing a human pose. The annotations for each frame include **keypoints** that represent the 3D coordinates of different body parts (e.g., head, shoulders, hips, knees, etc.). These keypoints are used to train the model to predict future poses based on previous frames.

3. Models

The project uses two main models:

- **Global Model (`net_g`)**: This model uses an **LSTM (Long Short-Term Memory)** network to predict the **global motion** of a person. Specifically, it predicts the person's speed and overall movement in the 3D space.
- **Local Model (`net_l`)**: This model is a **Variational Autoencoder (VAE)** that predicts **local motion** and speed, focusing on specific body part movements. It helps capture finer, detailed motions that are independent of the global movement.

These models work together to decouple the prediction of global and local motion, improving accuracy by focusing on different aspects of human motion.

4. Train and Test Sets

- **Training Set**: The training set contains a subset of the 3DPW dataset used to train the models. It provides the input-output pairs that allow the models to learn the relationships between previous and future poses.
- **Testing Set**: The testing set is a separate subset of the 3DPW dataset that is not used during training. It is used to evaluate the performance of the trained models and determine how well they generalize to unseen data.

5. Folder Structure and File Explanations

Here is the complete folder structure with explanations for each file and its role in the project:

```
pose-prediction/                     # Project repository
│
├── 3dpw/                            # Contains data, training, and model
files
│   ├── somof_data_3dpw/             # Dataset for 3DPW, including video
sequences and annotations
│   ├── train.py                     # Script for training the model on the
training set
│   ├── valid.py                     # Script for validating the model on the
validation set
│   ├── test.py                      # Script for testing the model on the
test set
│   ├── DataLoader.py                # Loads data for training and validation
datasets
│   ├── DataLoader_test.py           # Loads data for testing dataset
│   ├── model.py                     # Contains the implementation of the
neural network (Encoder, Decoder)
│   ├── utils.py                     # Contains various helper functions used
throughout the project
│   ├── viz.py                       # Script for visualizing predictions and
results
```

Explanation of Each File:

1. **somof_data_3dpw/**: This folder contains the dataset for the **3DPW dataset**. It includes video sequences and the corresponding 3D human pose annotations (in JSON format) that will be used for training and testing the models.
2. **train.py**: This script is used for **training the model** on the training dataset. It loads the training data, processes it, and trains the model (using `model.py`). It also saves the trained model weights for later use.
3. **valid.py**: This script is used for **validating the model** using the validation dataset. During training, it's important to evaluate the model's performance on unseen data to monitor overfitting. This script runs after training to check how well the model generalizes.
4. **test.py**: This script is used for **testing the model** on the test dataset. After training, the model's performance is tested using this dataset to evaluate its real-world application. The script generates predictions and compares them to the ground truth to calculate accuracy and other metrics.
5. **DataLoader.py**: This file contains the **data loader** function for the **training** and **validation datasets**. It reads the data from the 3DPW dataset, processes it, and prepares it

for input to the neural network. It also handles batching and shuffling of the data to improve training.
6. **DataLoader_test.py**: This file contains the **data loader** function for the **test dataset**. Like `DataLoader.py`, it reads the test data, processes it, and prepares it for evaluation. The key difference is that this loader is used exclusively for the test set, which is not used during training.
7. **model.py**: This file contains the **neural network architecture**. It defines the Encoder and Decoder networks, which are used to predict the global and local motion of the human body. It also includes the LSTM and VAE models for predicting future poses from past data.
8. **utils.py**: This file contains various **helper functions** needed throughout the project. These could include data processing functions, metrics calculation, model evaluation utilities, and other functions that are used in different parts of the project.
9. **viz.py**: This file is used for **visualizing predictions** and results. It can display graphs, plots, or visual representations of the predicted poses versus the actual poses. This helps in debugging and assessing the model's performance.

6. Encoder and Decoder

- **Encoder**: The encoder is a neural network that takes the input (previous poses) and encodes it into a latent space representation. This representation captures the essential information about the pose and motion of the person.
- **Decoder**: The decoder takes the encoded representation from the encoder and reconstructs the future poses. It outputs the predicted pose based on the information it learned from the training data.

7. Dataset Format

The 3DPW dataset is stored in **JSON files**, where each file contains a series of frames for a given video sequence. The key points are annotated in each frame and consist of 3D coordinates for specific body parts (e.g., head, shoulders, hips). These keypoints allow the model to learn how human bodies move in 3D space.

Each JSON file typically has the following structure:

```
{
  "frame_1": {
    "keypoints": [x1, y1, z1, x2, y2, z2, ..., xn, yn, zn]
  },
  "frame_2": {
    "keypoints": [x1, y1, z1, x2, y2, z2, ..., xn, yn, zn]
  },
  ...
}
```

Each frame contains keypoints with three values representing the 3D position of each body part.

8. Training Script Options (train.py)

1. **hidden_dim** (int):
   - Controls the "size" of the internal part of the model.
   - **Increase**: The model can capture more complex patterns.
   - **Decrease**: The model may not capture as much detail but will train faster.
2. **latent_dim** (int):
   - Determines how much information the model keeps in its internal "memory" (latent space).
   - **Increase**: More detailed information is captured, but requires more memory and training time.
   - **Decrease**: Less detailed information, which may lead to poorer performance.
3. **embedding_dim** (int):
   - The size of the "encoding" that represents input data.
   - **Increase**: The model may understand the input better but will need more time to process.
   - **Decrease**: Faster but might lose important details about the input.
4. **dropout** (float):
   - Randomly "turns off" parts of the model to prevent overfitting (learning patterns that are not useful).
   - **Increase**: More parts are turned off, which helps avoid overfitting but may slow down learning.
   - **Decrease**: Less dropout, which can speed up learning but may lead to overfitting.
5. **lr (learning rate)** (float):
   - Controls how much the model changes after each step of training.
   - **Increase**: The model learns faster but may overshoot the correct pattern.
   - **Decrease**: The model learns slower but more precisely.
6. **n_epochs** (int):
   - The number of times the model looks through the entire training dataset.
   - **Increase**: The model has more chances to learn, potentially improving accuracy.
   - **Decrease**: The model may not learn enough and perform poorly. (Too few epochs = underfitting)
7. **batch_size** (int):
   - The number of data points the model processes at once.
   - **Increase**: Faster training but requires more memory.
   - **Decrease**: Slower training but uses less memory.
8. **loader_shuffle** (bool):
   - Whether or not the training data is shuffled (mixed up) at the start of each round.
   - **True**: Helps the model learn better by preventing it from memorizing the order of data.
   - **False**: The model might learn the data order rather than the patterns.
9. **load_checkpoint** (bool):
   - If True, it continues training from where it left off instead of starting over.
   - **True**: Useful if training was interrupted.
   - **False**: Starts fresh training.
10. **dev** (str):

- Specifies whether to use a CPU or GPU for training.
- **cuda** (GPU): Faster training if you have a compatible graphics card.

- **cpu**: Slower but works on any computer.

9. Sequence of Events

1. **Data Loading**: The dataset is loaded and processed using `DataLoader.py` (for training/validation) and `DataLoader_test.py` (for testing).
2. **Model Training**: `train.py` is used to train both models (`net_g` for global motion and `net_l` for local motion).
3. **Model Validation**: `valid.py` is used to evaluate the model on the validation dataset.
4. **Model Testing**: `test.py` is used to test the final model on the test dataset and evaluate performance metrics.
5. **Model Saving**: The trained models are saved for future use.

Code parts :

Important!

`val.py`

- **Purpose**: This script is used for evaluating the performance of a trained model on validation data.
- **What it does**:
  - Loads the validation dataset.
  - Runs the model on the validation data.
  - Visualizes and saves the observed, predicted, and ground truth poses in the `visualizations/` folder.
  - Reports the model's performance based on its predictions.

`train.py`

- **Purpose**: This script is used to train the model.
- **What it does**:
  - Initializes the model, loss functions, optimizer, and dataset.
  - Loads the training data.
  - Loops through the dataset for a set number of epochs (typically 50) to train the model, adjusting the weights based on the loss function.
  - Saves the trained model for later use or evaluation.

`test.py`

- **Purpose**: This script is used to test the model on a test dataset (after training).
- **What it does**:
  - Loads the trained model and test data.
  - Evaluates the model's performance on the test dataset.
  - Typically outputs the final results of the model's predictions, such as accuracy or error metrics.

Model.py

- **LSTM_g Class (Global Model)**:

  - The `LSTM_g` model predicts future body positions based on past movement data. It uses two LSTM layers, one for encoding the input (past movement data) and one for decoding the predicted future positions.
  - The `embedding_fn` converts the input (which appears to be 3D pose data) into a more suitable form for the LSTM.
  - It processes the data and predicts future positions step by step.

- **Encoder Class**:

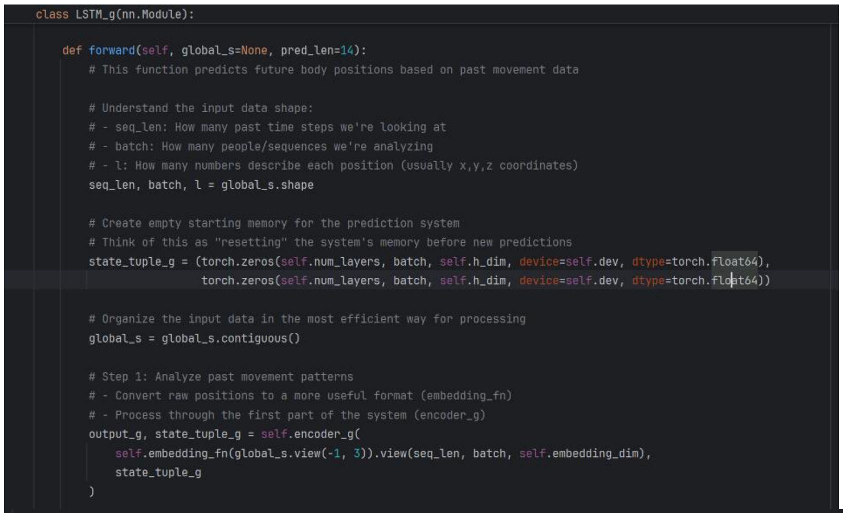
  - The `Encoder` is a standard LSTM encoder, which takes in sequential data (`obs_s`), processes it, and outputs a mean and variance representing a latent space (for the VAE).
  - The final hidden state of the LSTM is used to compute `mean` and `log_var`, which are then used for the reparameterization trick in the VAE.

- **Decoder Class**:

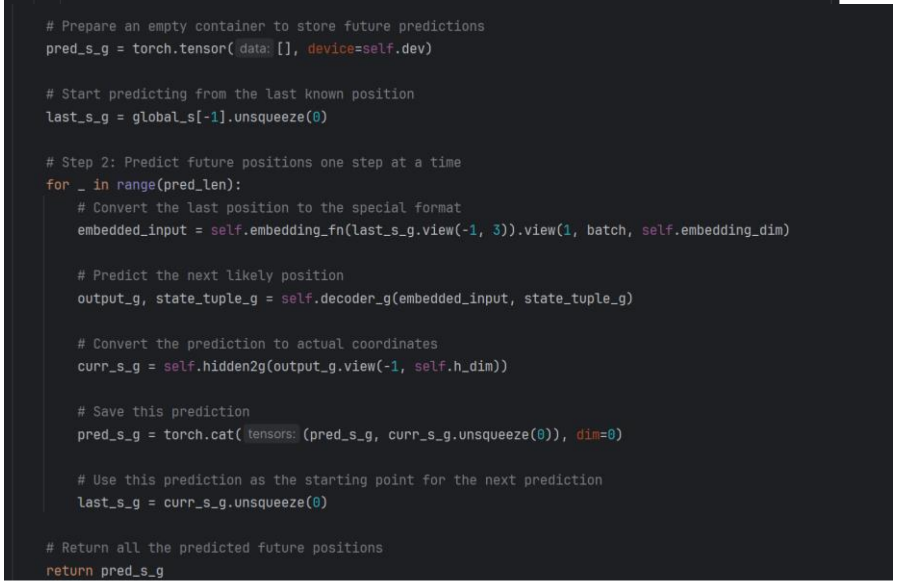
  - The `Decoder` takes the latent variable (from the encoder) and previous body positions (`obs_s`) to generate future pose data.
  - It uses an LSTM layer and a fully connected (FC) layer to map the latent representation back to pose space, iteratively predicting the next pose over a set length (`pred_len`).

- **VAE Class**:

  - The `VAE` class integrates the `Encoder` and `Decoder` into a full variational autoencoder model.
  - It implements the reparameterization trick to sample from the learned latent distribution (using the mean and variance from the encoder) and passes it through the decoder to reconstruct the sequence.



```python
class LSTM_g(nn.Module):

    def forward(self, global_s=None, pred_len=14):
        # This function predicts future body positions based on past movement data

        # Understand the input data shape:
        # - seq_len: How many past time steps we're looking at
        # - batch: How many people/sequences we're analyzing
        # - l: How many numbers describe each position (usually x,y,z coordinates)
        seq_len, batch, l = global_s.shape

        # Create empty starting memory for the prediction system
        # Think of this as "resetting" the system's memory before new predictions
        state_tuple_g = (torch.zeros(self.num_layers, batch, self.h_dim, device=self.dev, dtype=torch.float64),
                         torch.zeros(self.num_layers, batch, self.h_dim, device=self.dev, dtype=torch.float64))

        # Organize the input data in the most efficient way for processing
        global_s = global_s.contiguous()

        # Step 1: Analyze past movement patterns
        # - Convert raw positions to a more useful format (embedding_fn)
        # - Process through the first part of the system (encoder_g)
        output_g, state_tuple_g = self.encoder_g(
            self.embedding_fn(global_s.view(-1, 3)).view(seq_len, batch, self.embedding_dim),
            state_tuple_g
        )
```



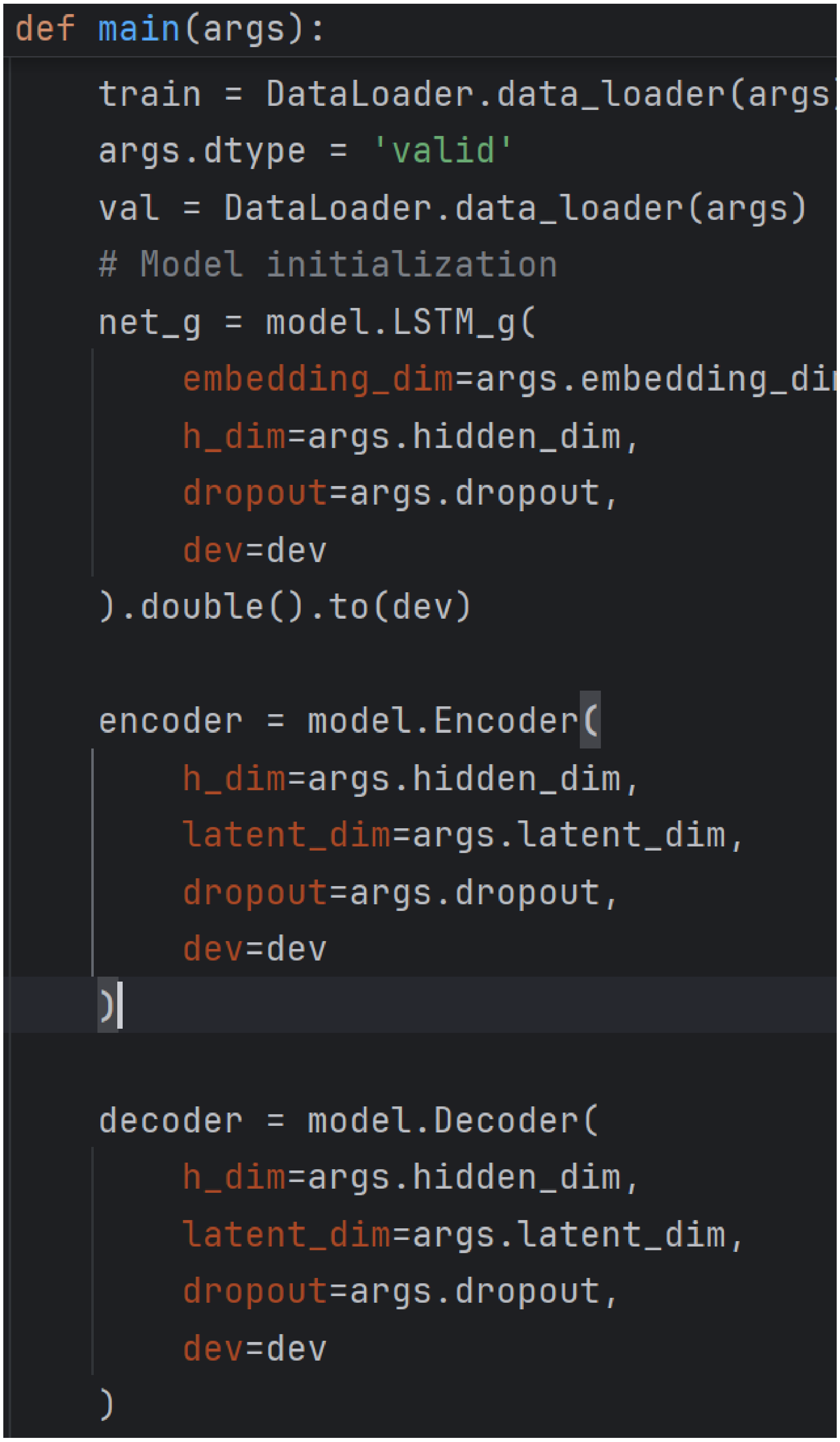
```python
        # Prepare an empty container to store future predictions
        pred_s_g = torch.tensor( data: [], device=self.dev)

        # Start predicting from the last known position
        last_s_g = global_s[-1].unsqueeze(0)

        # Step 2: Predict future positions one step at a time
        for _ in range(pred_len):
            # Convert the last position to the special format
            embedded_input = self.embedding_fn(last_s_g.view(-1, 3)).view(1, batch, self.embedding_dim)

            # Predict the next likely position
            output_g, state_tuple_g = self.decoder_g(embedded_input, state_tuple_g)

            # Convert the prediction to actual coordinates
            curr_s_g = self.hidden2g(output_g.view(-1, self.h_dim))

            # Save this prediction
            pred_s_g = torch.cat( tensors: (pred_s_g, curr_s_g.unsqueeze(0)), dim=0)

            # Use this prediction as the starting point for the next prediction
            last_s_g = curr_s_g.unsqueeze(0)

        # Return all the predicted future positions
        return pred_s_g
```

Train.py

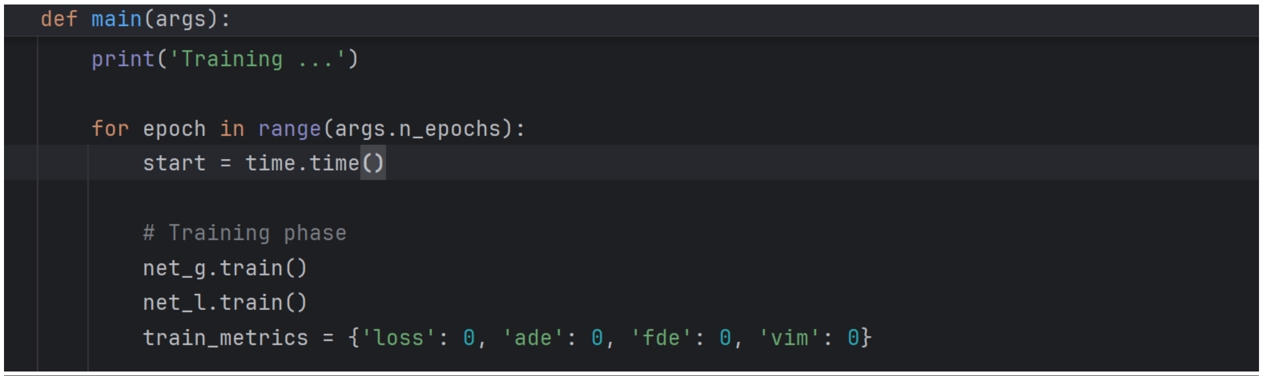
This script trains a motion prediction model using two networks: one for global motion and another for local motion. It loads training and validation data, calculates losses, and updates the model through backpropagation. Metrics like loss, ADE, FDE, and VIM are tracked to evaluate the model's performance. The trained models are saved periodically for later use.

These terms represent different evaluation metrics used to assess the model's performance in the context of motion prediction or trajectory forecasting:

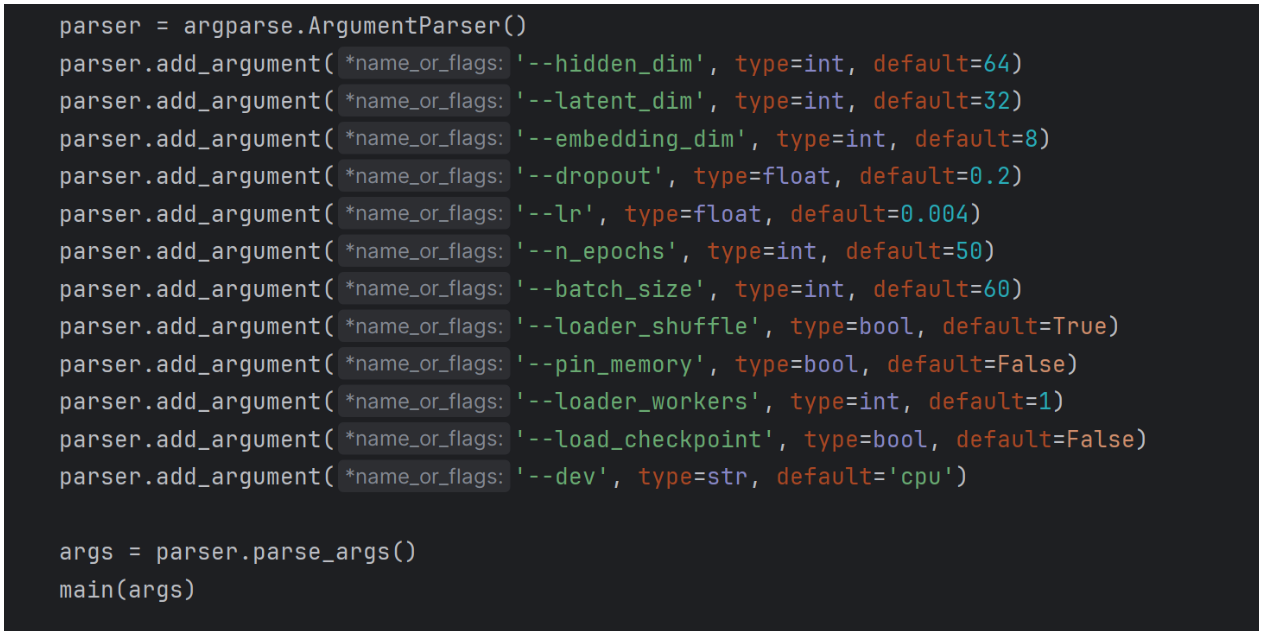
- **Loss**: A general measure of the error between the predicted and actual outputs. In this case, it combines the global and local motion predictions.
- **ADE (Average Displacement Error)**: Measures the average distance between the predicted and actual positions over the entire predicted trajectory. It gives an indication of how close the predicted path is to the true path.
- **FDE (Final Displacement Error)**: Measures the distance between the predicted final position and the actual final position. It's used to evaluate how well the model predicts the last point in the trajectory.
- **VIM (Velocity-Integrated Metric)**: A custom metric that likely combines velocity and displacement errors to provide a more detailed evaluation of motion prediction quality, though its exact definition can vary depending on the implementation.



```python
def main(args):
    train = DataLoader.data_loader(args
    args.dtype = 'valid'
    val = DataLoader.data_loader(args)
    # Model initialization
    net_g = model.LSTM_g(
        embedding_dim=args.embedding_di
        h_dim=args.hidden_dim,
        dropout=args.dropout,
        dev=dev
    ).double().to(dev)

    encoder = model.Encoder(
        h_dim=args.hidden_dim,
        latent_dim=args.latent_dim,
        dropout=args.dropout,
        dev=dev
    )

    decoder = model.Decoder(
        h_dim=args.hidden_dim,
        latent_dim=args.latent_dim,
        dropout=args.dropout,
        dev=dev
    )
```



```
def main(args):
    print('Training ...')

    for epoch in range(args.n_epochs):
        start = time.time()

        # Training phase
        net_g.train()
        net_l.train()
        train_metrics = {'loss': 0, 'ade': 0, 'fde': 0, 'vim': 0}
```



```
parser = argparse.ArgumentParser()
parser.add_argument( *name_or_flags: '--hidden_dim', type=int, default=64)
parser.add_argument( *name_or_flags: '--latent_dim', type=int, default=32)
parser.add_argument( *name_or_flags: '--embedding_dim', type=int, default=8)
parser.add_argument( *name_or_flags: '--dropout', type=float, default=0.2)
parser.add_argument( *name_or_flags: '--lr', type=float, default=0.004)
parser.add_argument( *name_or_flags: '--n_epochs', type=int, default=50)
parser.add_argument( *name_or_flags: '--batch_size', type=int, default=60)
parser.add_argument( *name_or_flags: '--loader_shuffle', type=bool, default=True)
parser.add_argument( *name_or_flags: '--pin_memory', type=bool, default=False)
parser.add_argument( *name_or_flags: '--loader_workers', type=int, default=1)
parser.add_argument( *name_or_flags: '--load_checkpoint', type=bool, default=False)
parser.add_argument( *name_or_flags: '--dev', type=str, default='cpu')

args = parser.parse_args()
main(args)
```

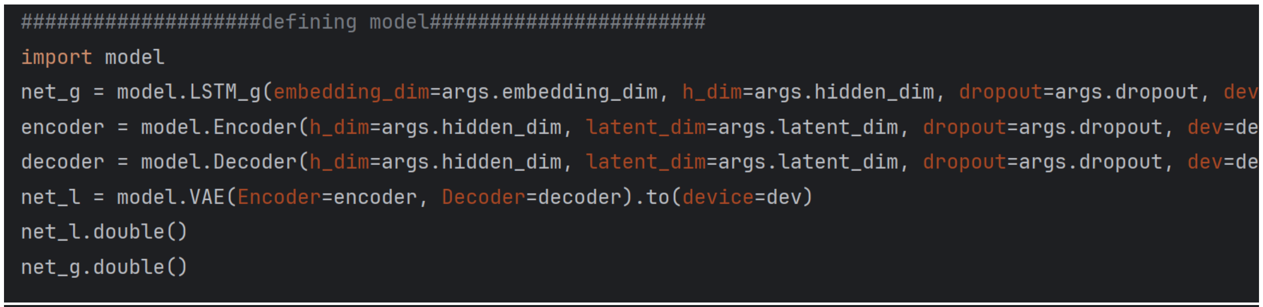
Test.py:

Global Speed:
It refers to the **movement of the whole body in space** — like how fast and in what direction a person is walking or running overall. For example, if someone moves 1 meter forward in 1 second, their global speed is 1 m/s forward.

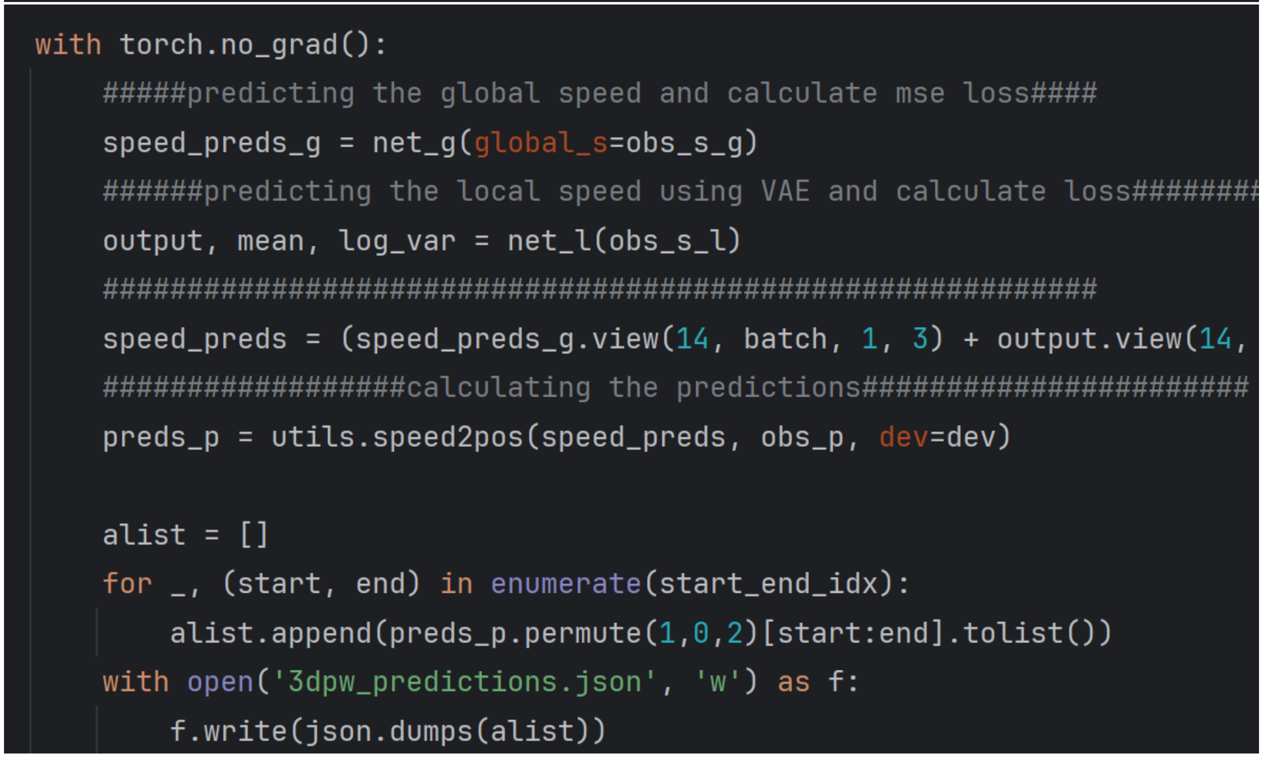
Local Speed:
It refers to the **motion of each body part relative to the body itself** — like how arms or legs move while walking. Even if a person stands still (global speed = 0), their local speed can be high if they wave their arms.

Checkpoint:
A checkpoint is a saved file that stores the **learned parameters (weights)** of a trained model. Instead of training the model from zero, we load the checkpoint to reuse what the model already learned.

the code initializes the models. `net_g` is an LSTM that predicts the global speed of the human body. `net_l` is a VAE made of an encoder and decoder, which is responsible for learning and predicting local motion patterns (the movements of individual joints relative to the body). The models are moved to the selected device and set to use double precision. Then, the code loads the trained parameters for these models from saved checkpoint files (`checkpoint_g.pkl` and `checkpoint_l.pkl`) and sets them to evaluation mode.



```
###################defining model#####################
import model
net_g = model.LSTM_g(embedding_dim=args.embedding_dim, h_dim=args.hidden_dim, dropout=args.dropout, dev
encoder = model.Encoder(h_dim=args.hidden_dim, latent_dim=args.latent_dim, dropout=args.dropout, dev=de
decoder = model.Decoder(h_dim=args.hidden_dim, latent_dim=args.latent_dim, dropout=args.dropout, dev=de
net_l = model.VAE(Encoder=encoder, Decoder=decoder).to(device=dev)
net_l.double()
net_g.double()
```



```
with torch.no_grad():
    #####predicting the global speed and calculate mse loss####
    speed_preds_g = net_g(global_s=obs_s_g)
    ######predicting the local speed using VAE and calculate loss#######
    output, mean, log_var = net_l(obs_s_l)
    ###########################################################
    speed_preds = (speed_preds_g.view(14, batch, 1, 3) + output.view(14,
    ##################calculating the predictions######################
    preds_p = utils.speed2pos(speed_preds, obs_p, dev=dev)

    alist = []
    for _, (start, end) in enumerate(start_end_idx):
        alist.append(preds_p.permute(1,0,2)[start:end].tolist())
    with open('3dpw_predictions.json', 'w') as f:
        f.write(json.dumps(alist))
```

Val.py

This script is for **testing** a deep learning model that predicts human body poses based on motion (like predicting how a person will move).

Here's a simple explanation:

◆ 1. Load Data

It uses a `DataLoader` to load **validation data** (pre-recorded human motion samples).
Each data sample has:

- `obs_p`: observed past poses (positions)
- `obs_s`: observed past speeds
- `target_p`: future poses (ground truth)
- `target_s`: future speeds (ground truth)

◆ 2. Load Trained Models

It loads 2 saved models from `.pkl` files:

- `net_g`: predicts **global speed** (how the body moves in space)
- `net_l`: VAE that predicts **local speed** (how body parts move relative to the body)

◆ 3. Split Motion: Global vs Local

Motion is split into:

- **Global**: average of 2 body joints (usually hips)
- **Local**: difference from the global (how limbs move)

◆ 4. Make Predictions

- `net_g` predicts global speed.
- `net_l` predicts local speed.
- These are combined to get **full speed prediction**.
- Then `utils.speed2pos` converts predicted speed into predicted positions (poses).

◆ 5. Evaluate Performance

It calculates how good the predictions are using:

- **MSE Loss**: difference between predicted & real speed.
- **ADE (Average Displacement Error)**: average distance error between predicted and real positions.
- **FDE (Final Displacement Error)**: error at the final frame.
- **VIM**: another metric to evaluate motion quality.

IMPORTANT (down)

- `net_g` uses LSTM to predict **global speed**.

- `net_l` uses VAE to predict **local speed**.

`net_g` = the **global speed predictor**

- Predicts the **overall body motion**, like how a person is moving in space.

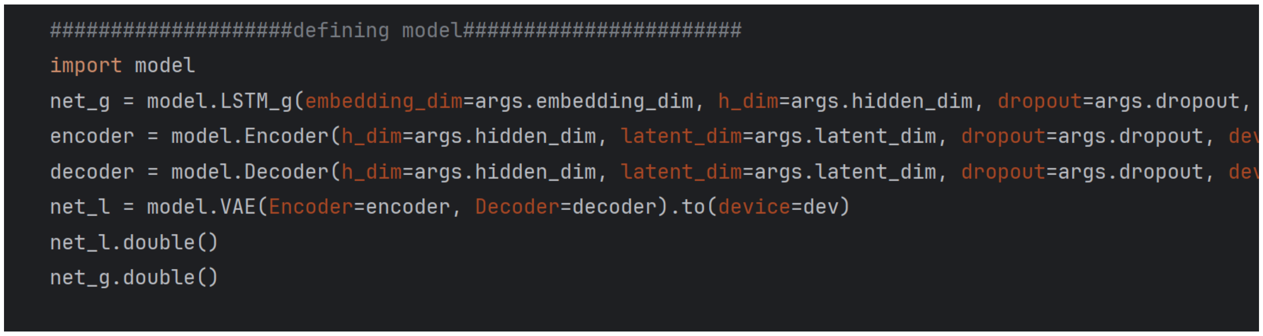
`net_l` = the **local speed predictor** (VAE model)

- Predicts **body part motion**, like how arms and legs move **relative to the body**.

Ground truth means the **real, correct data**.
In this script, it's the **real future speeds and positions** of the human body that the model tries to predict.

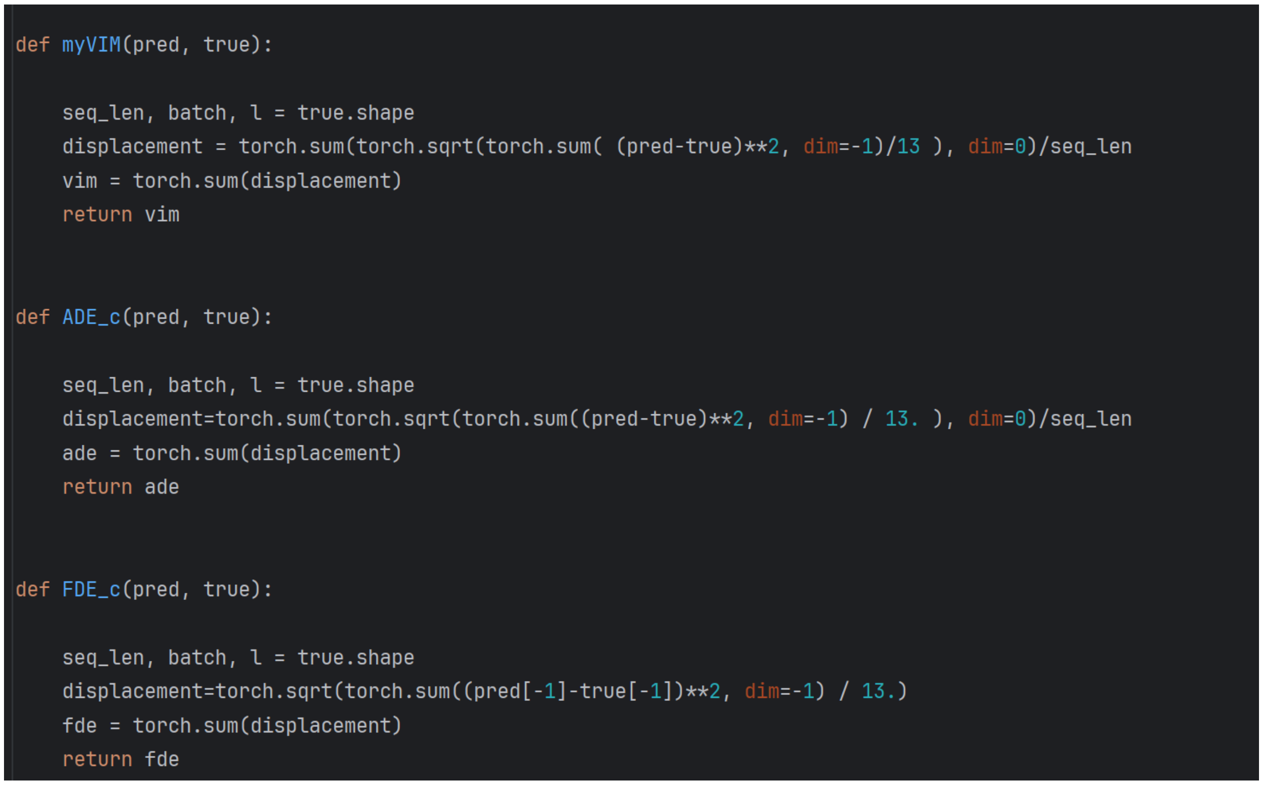
- `target_s`: real future **speeds** (ground truth for speed)
- `target_p`: real future **positions** (ground truth for position)

The model output is compared to these to calculate errors (loss).



```
####################defining model#######################
import model
net_g = model.LSTM_g(embedding_dim=args.embedding_dim, h_dim=args.hidden_dim, dropout=args.dropout,
encoder = model.Encoder(h_dim=args.hidden_dim, latent_dim=args.latent_dim, dropout=args.dropout, dev
decoder = model.Decoder(h_dim=args.hidden_dim, latent_dim=args.latent_dim, dropout=args.dropout, dev
net_l = model.VAE(Encoder=encoder, Decoder=decoder).to(device=dev)
net_l.double()
net_g.double()
```

Utils.py:



```python
def myVIM(pred, true):

    seq_len, batch, l = true.shape
    displacement = torch.sum(torch.sqrt(torch.sum( (pred-true)**2, dim=-1)/13 ), dim=0)/seq_len
    vim = torch.sum(displacement)
    return vim


def ADE_c(pred, true):

    seq_len, batch, l = true.shape
    displacement=torch.sum(torch.sqrt(torch.sum((pred-true)**2, dim=-1) / 13. ), dim=0)/seq_len
    ade = torch.sum(displacement)
    return ade


def FDE_c(pred, true):

    seq_len, batch, l = true.shape
    displacement=torch.sqrt(torch.sum((pred[-1]-true[-1])**2, dim=-1) / 13.)
    fde = torch.sum(displacement)
    return fde
```

This file has helper functions for **evaluating** and **visualizing** pose predictions.

- `myVIM`, `ADE_c`, `FDE_c`: measure prediction error.

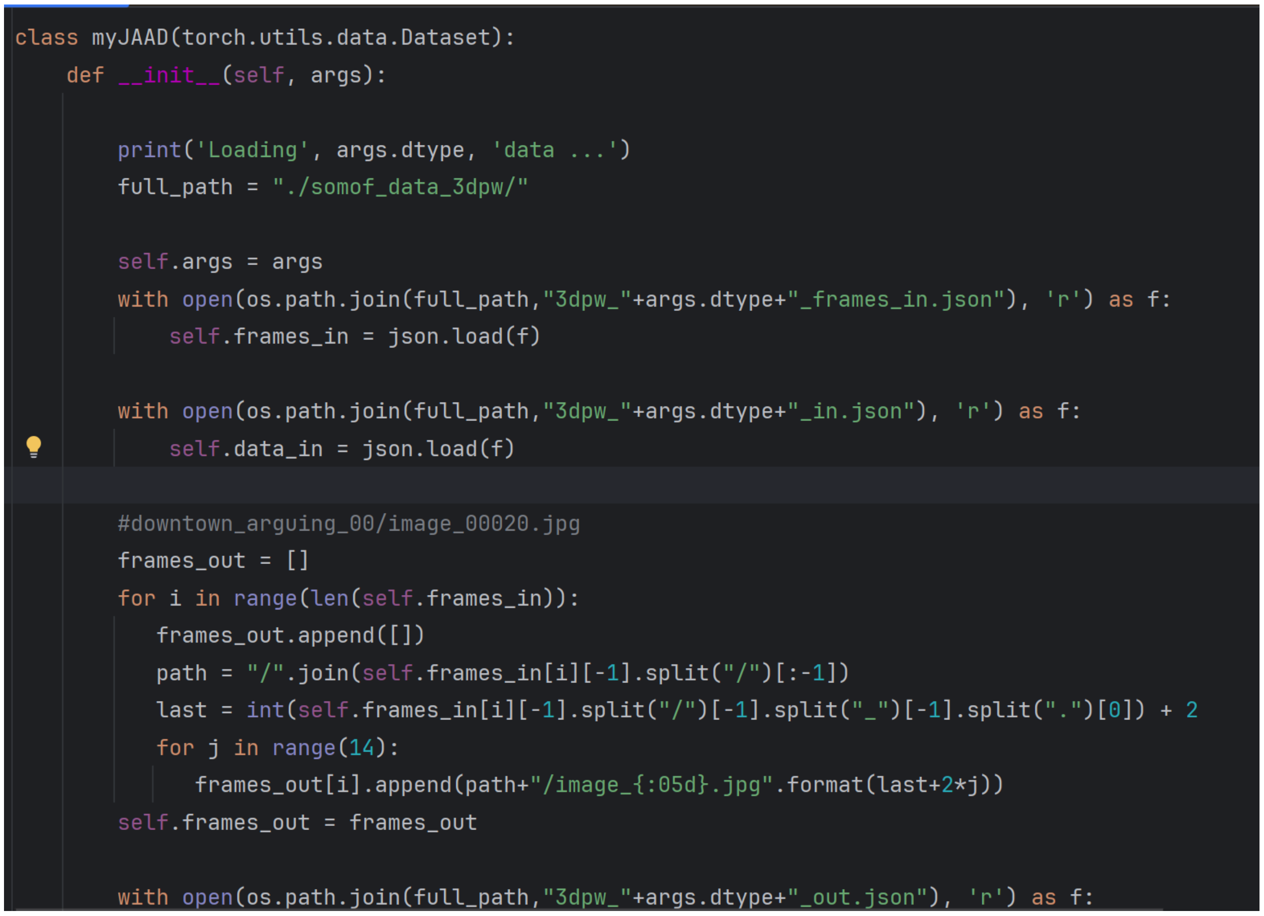
Used to check how accurate the model is.

Dataloader.py :

This file loads and prepares pose data for training/testing.

- `myJAAD`: custom dataset class that loads 3DPW pose sequences (input/output) from JSON files.
- `__getitem__`: returns observed positions, speeds, future positions, and their image paths.
- `my_collate`: merges multiple samples into a batch for training.
- `data_loader`: returns a PyTorch `DataLoader` using the dataset and batch settings from `args`.

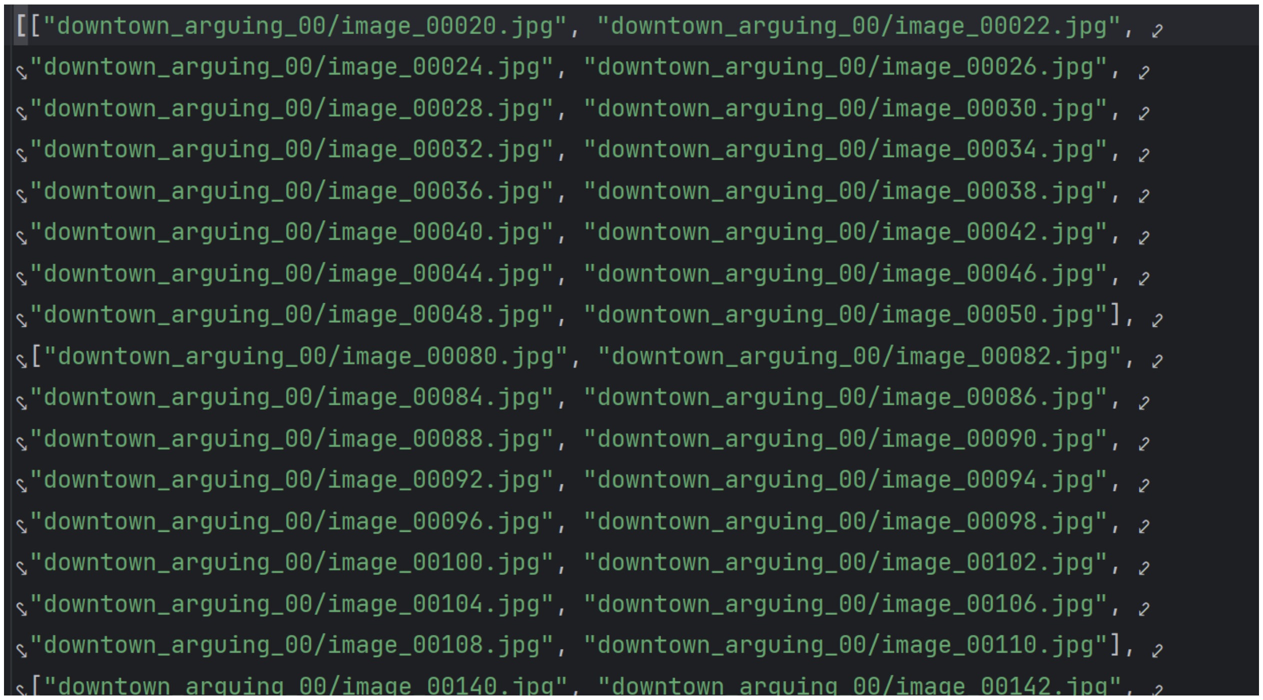
Used to feed pose data into a model.



```python
class myJAAD(torch.utils.data.Dataset):
    def __init__(self, args):

        print('Loading', args.dtype, 'data ...')
        full_path = "./somof_data_3dpw/"

        self.args = args
        with open(os.path.join(full_path,"3dpw_"+args.dtype+"_frames_in.json"), 'r') as f:
            self.frames_in = json.load(f)

        with open(os.path.join(full_path,"3dpw_"+args.dtype+"_in.json"), 'r') as f:
            self.data_in = json.load(f)

        #downtown_arguing_00/image_00020.jpg
        frames_out = []
        for i in range(len(self.frames_in)):
            frames_out.append([])
            path = "/".join(self.frames_in[i][-1].split("/")[:-1])
            last = int(self.frames_in[i][-1].split("/")[-1].split("_")[-1].split(".")[0]) + 2
            for j in range(14):
                frames_out[i].append(path+"/image_{:05d}.jpg".format(last+2*j))
        self.frames_out = frames_out

        with open(os.path.join(full_path,"3dpw_"+args.dtype+"_out.json"), 'r') as f:
```

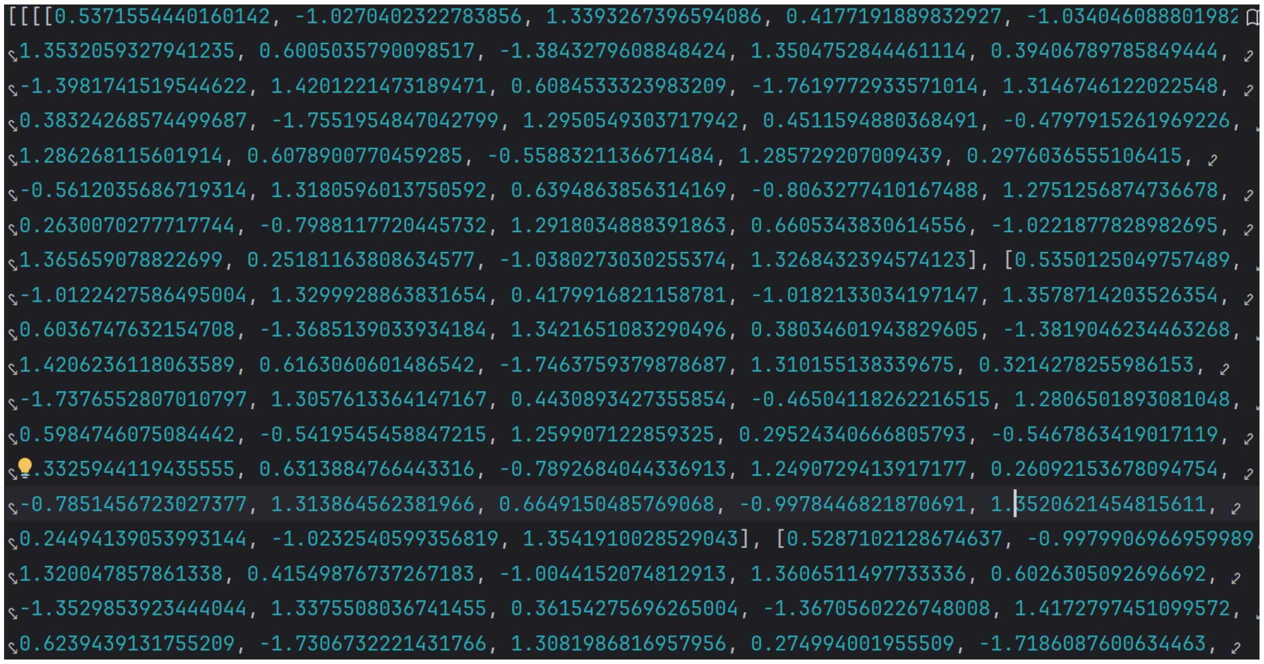
Checkpoints are weights as a pickle file so we cant show

How Dataset looks like closely :



```
[["downtown_arguing_00/image_00020.jpg", "downtown_arguing_00/image_00022.jpg",
"downtown_arguing_00/image_00024.jpg", "downtown_arguing_00/image_00026.jpg",
"downtown_arguing_00/image_00028.jpg", "downtown_arguing_00/image_00030.jpg",
"downtown_arguing_00/image_00032.jpg", "downtown_arguing_00/image_00034.jpg",
"downtown_arguing_00/image_00036.jpg", "downtown_arguing_00/image_00038.jpg",
"downtown_arguing_00/image_00040.jpg", "downtown_arguing_00/image_00042.jpg",
"downtown_arguing_00/image_00044.jpg", "downtown_arguing_00/image_00046.jpg",
"downtown_arguing_00/image_00048.jpg", "downtown_arguing_00/image_00050.jpg"],
["downtown_arguing_00/image_00080.jpg", "downtown_arguing_00/image_00082.jpg",
"downtown_arguing_00/image_00084.jpg", "downtown_arguing_00/image_00086.jpg",
"downtown_arguing_00/image_00088.jpg", "downtown_arguing_00/image_00090.jpg",
"downtown_arguing_00/image_00092.jpg", "downtown_arguing_00/image_00094.jpg",
"downtown_arguing_00/image_00096.jpg", "downtown_arguing_00/image_00098.jpg",
"downtown_arguing_00/image_00100.jpg", "downtown_arguing_00/image_00102.jpg",
"downtown_arguing_00/image_00104.jpg", "downtown_arguing_00/image_00106.jpg",
"downtown_arguing_00/image_00108.jpg", "downtown_arguing_00/image_00110.jpg"],
["downtown_arguing_00/image_00140.jpg", "downtown_arguing_00/image_00142.jpg",
```

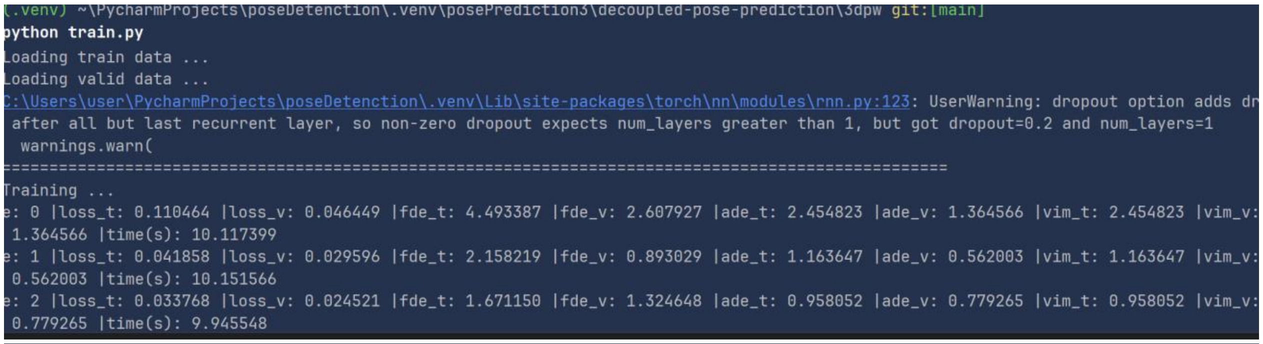
We pass a sequence of images (frames) not a video



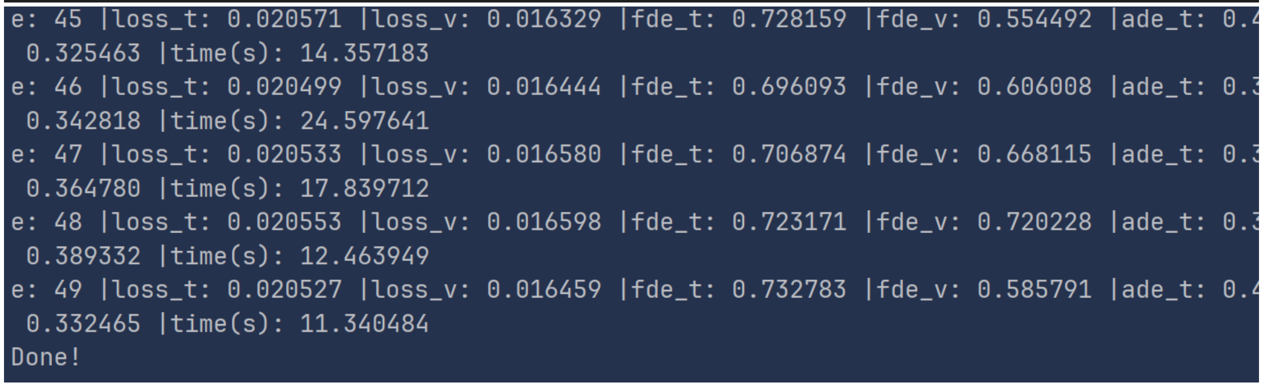
```
[[[[0.5371554440160142, -1.0270402322783856, 1.3393267396594086, 0.4177191889832927, -1.034046088801982
1.3532059327941235, 0.6005035790098517, -1.3843279608848424, 1.3504752844461114, 0.39406789785849444,
-1.3981741519544622, 1.4201221473189471, 0.6084533323983209, -1.7619772933571014, 1.3146746122022548,
0.38324268574499687, -1.7551954847042799, 1.2950549303717942, 0.4511594880368491, -0.4797915261969226,
1.286268115601914, 0.6078900770459285, -0.5588321136671484, 1.285729207009439, 0.2976036555106415,
-0.5612035686719314, 1.3180596013750592, 0.6394863856314169, -0.8063277410167488, 1.2751256874736678,
0.2630070277717744, -0.7988117720445732, 1.2918034888391863, 0.6605343830614556, -1.0221877828982695,
1.365659078822699, 0.25181163808634577, -1.0380273030255374, 1.326843239457412 3], [0.5350125049757489,
-1.0122427586495004, 1.3299928863831654, 0.4179916821158781, -1.0182133034197147, 1.3578714203526354,
0.6036747632154708, -1.3685139033934184, 1.3421651083290496, 0.38034601943829605, -1.3819046234463268,
1.4206236118063589, 0.616306060148654 2, -1.7463759379878687, 1.310155138339675, 0.3214278255986153,
-1.7376552807010797, 1.3057613364147167, 0.4430893427355854, -0.46504118262216515, 1.2806501893081048,
0.5984746075084442, -0.5419545458847215, 1.259907122859325, 0.29524340666805793, -0.5467863419017119,
.3325944119435555, 0.6313884766443316, -0.7892684044336913, 1.2490729413917177, 0.26092153678094754,
-0.7851456723027377, 1.313864562381966, 0.6649150485769068, -0.9978446821870691, 1.352062145481561 1,
0.24494139053993144, -1.0232540599356819, 1.3541910028529043], [0.5287102128674637, -0.9979906966959989
1.320047857861338, 0.41549876737267183, -1.0044152074812913, 1.3606511497733336, 0.6026305092696692,
-1.3529853923444044, 1.3375508036741455, 0.36154275696265004, -1.3670560226748008, 1.4172797451099572,
0.6239439131755209, -1.7306732221431766, 1.3081986816957956, 0.274994001955509, -1.7186087600634463,
```

This is what thee program needs and reads , they are coordinates and keypoints of a sequence or image

Program usage example :



```
(.venv) ~\PycharmProjects\poseDetenction\.venv\posePrediction3\decoupled-pose-prediction\3dpw git:[main]
python train.py
Loading train data ...
Loading valid data ...
C:\Users\user\PycharmProjects\poseDetenction\.venv\Lib\site-packages\torch\nn\modules\rnn.py:123: UserWarning: dropout option adds dr
 after all but last recurrent layer, so non-zero dropout expects num_layers greater than 1, but got dropout=0.2 and num_layers=1
  warnings.warn(
================================================================================================
Training ...
e: 0 |loss_t: 0.110464 |loss_v: 0.046449 |fde_t: 4.493387 |fde_v: 2.607927 |ade_t: 2.454823 |ade_v: 1.364566 |vim_t: 2.454823 |vim_v:
 1.364566 |time(s): 10.117399
e: 1 |loss_t: 0.041858 |loss_v: 0.029596 |fde_t: 2.158219 |fde_v: 0.893029 |ade_t: 1.163647 |ade_v: 0.562003 |vim_t: 1.163647 |vim_v:
 0.562003 |time(s): 10.151566
e: 2 |loss_t: 0.033768 |loss_v: 0.024521 |fde_t: 1.671150 |fde_v: 1.324648 |ade_t: 0.958052 |ade_v: 0.779265 |vim_t: 0.958052 |vim_v:
 0.779265 |time(s): 9.945548
```

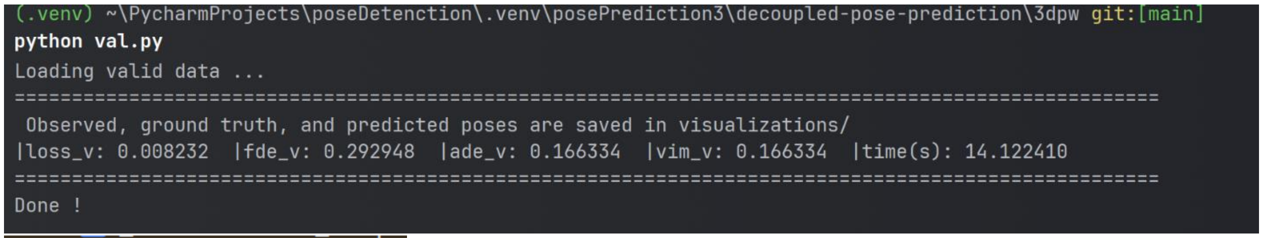


```
e: 45 |loss_t: 0.020571 |loss_v: 0.016329 |fde_t: 0.728159 |fde_v: 0.554492 |ade_t: 0.4
 0.325463 |time(s): 14.357183
e: 46 |loss_t: 0.020499 |loss_v: 0.016444 |fde_t: 0.696093 |fde_v: 0.606008 |ade_t: 0.3
 0.342818 |time(s): 24.597641
e: 47 |loss_t: 0.020533 |loss_v: 0.016580 |fde_t: 0.706874 |fde_v: 0.668115 |ade_t: 0.3
 0.364780 |time(s): 17.839712
e: 48 |loss_t: 0.020553 |loss_v: 0.016598 |fde_t: 0.723171 |fde_v: 0.720228 |ade_t: 0.3
 0.389332 |time(s): 12.463949
e: 49 |loss_t: 0.020527 |loss_v: 0.016459 |fde_t: 0.732783 |fde_v: 0.585791 |ade_t: 0.4
 0.332465 |time(s): 11.340484
Done!
```

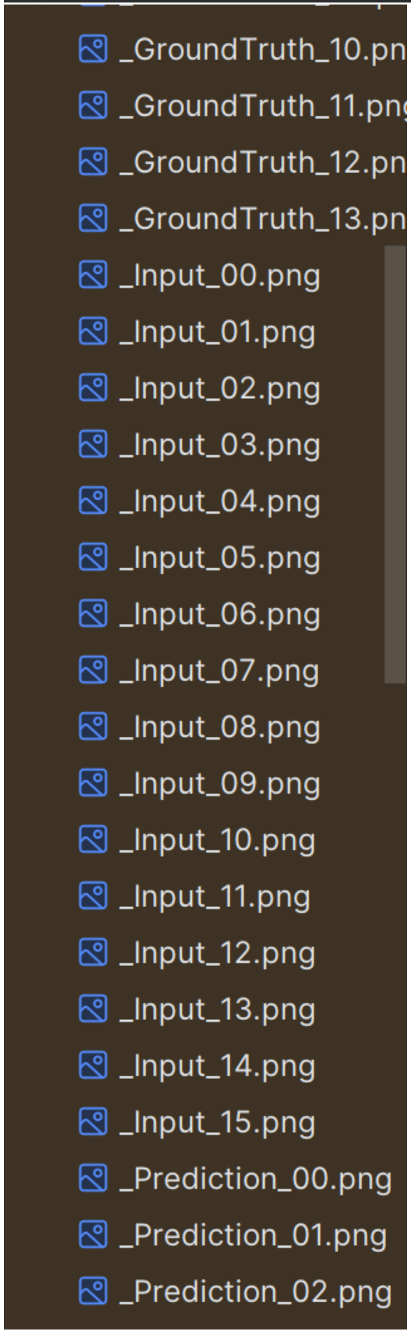
Here, we start the program using `train.py` without specifying any values, so it defaults to the parameters we've pre-defined.

Why 50 Epochs?

- **Epoch**: An **epoch** refers to one full pass through the entire dataset during the training process. In other words, after the model processes each data point once, one epoch is completed.
- **Why 50 Epochs**: The number of epochs is a hyperparameter that you choose to determine how many times the model will learn from the entire dataset. **50 epochs** is a commonly used starting point for training, as it often provides a good balance between model convergence and computational cost. You may adjust this based on the model's performance.
    - **Too few epochs** might lead to underfitting (the model hasn't learned enough).
    - **Too many epochs** might cause overfitting (the model memorizes the data instead of generalizing)

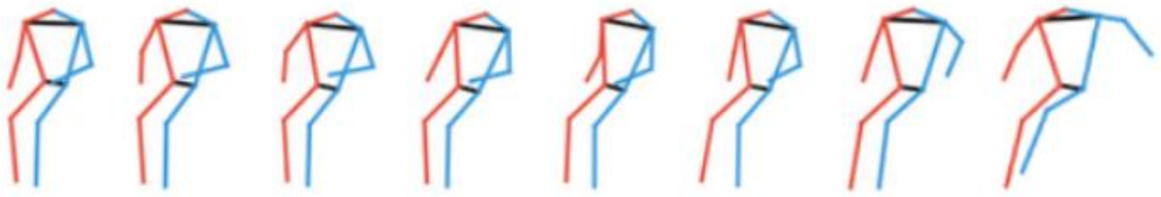


```
(.venv) ~\PycharmProjects\poseDetenction\.venv\posePrediction3\decoupled-pose-prediction\3dpw git:[main]
python val.py
Loading valid data ...
====================================================================================================
 Observed, ground truth, and predicted poses are saved in visualizations/
|loss_v: 0.008232  |fde_v: 0.292948  |ade_v: 0.166334  |vim_v: 0.166334  |time(s): 14.122410
====================================================================================================
Done !
```



```
_GroundTruth_10.pn
_GroundTruth_11.png
_GroundTruth_12.pn
_GroundTruth_13.pn
_Input_00.png
_Input_01.png
_Input_02.png
_Input_03.png
_Input_04.png
_Input_05.png
_Input_06.png
_Input_07.png
_Input_08.png
_Input_09.png
_Input_10.png
_Input_11.png
_Input_12.png
_Input_13.png
_Input_14.png
_Input_15.png
_Prediction_00.png
_Prediction_01.png
_Prediction_02.png
```

Image :



Predicted:

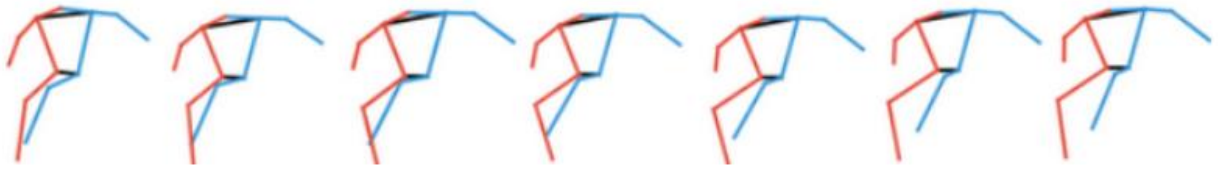
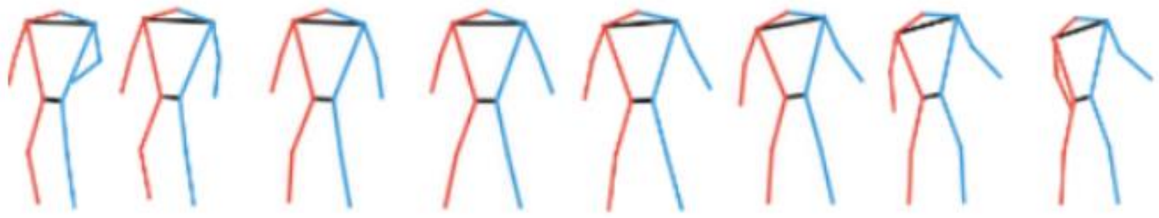


Image :



Predicted :