UFAZ

Master 2 DSAI
Autumn 2023

Advanced Deep Learning - Practical Work

Autoencoders and Variational Autoencoders

Goals

In this practical work, the first aim is to build an AutoEncoder for 1) denoising images and 2) detecting anomalies in time series. The second objective is to extend the architecture to build a Variational AutoEncoder in order to generate human motion sequences.

Ressources

For helping you, some parts of code are available here : Colab link

1 Autoencoder

1.1 Exercise 1 : Denoising Autoencoder

In this first exercise, we will see how an AutoEncoder can be employed for denoising data (images in our case). For that, an AutoEncoder is trained to reconstruct images without noise from noisy images.

1- The first thing to do is to load the MNIST dataset. To accelerate computation, you may want to use only a subset of the whole dataset (take at least 1000 images).

2- Apply the classical pre-processing steps on images, as we have seen during lectures and exercises.

3- Then, use the provided function `noise()` in order to generate a noisy version of the MNIST dataset. Try to plot some examples :

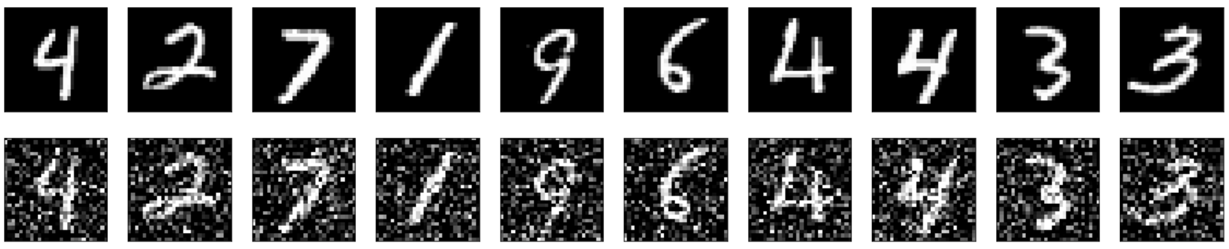

FIGURE 1 – Examples of some images from MNIST and their noisy version

4- Build an AutoEncoder based on a convolutional neural network. In that case, we won't need to analyze the latent space. Hence, adding Dense layers Dans ce cas, nous nous intéresserons pas à l'espace latent, l'ajout de couche `Dense` n'est donc pas obligatoire.

5- Train your model for reconstructing noisy images into orignal images during 100 epochs. For the loss function, you can use the `binary_crossentropy` which is more efficient in an autoencoder with MNIST (as images have a single channel).

6- Use you trained model for denoising test images, and then observe denoising results :

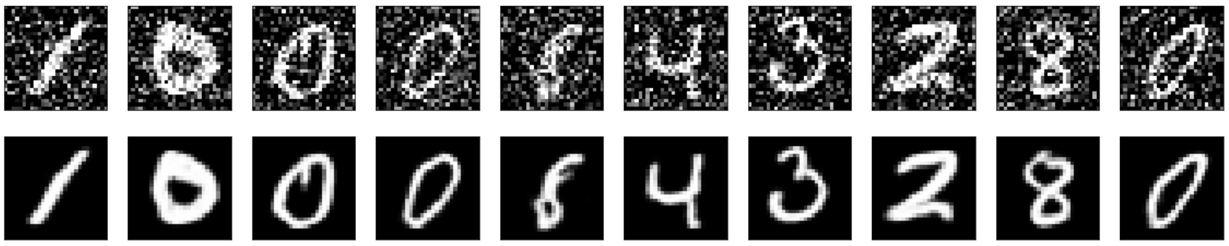

FIGURE 2 – Example of some denoised images with the autoencoder

1.2 Exercise 2 : anomaly detection

In this second exercise, we will see how an autoencoder can be used to detect anomalies (in time series in our case). For this, an autoencoder is trained with normal data. In the testing phase, the samples are passed through the network which reconstructs them. The detection error is thus used to decide whether a sample is normal or abnormal. The idea is that an autoencoder trained on normal data will reconstruct abnormal data less well.

1.2.1 Data

We will use time series datasets from the Numenta Anomaly Benchmark (NAB) database. In particular, we will use the following synthetic data :

— Normal data : `art_daily_small_noise.csv` (lien)

— Abnormal data : `art_daily_jumpsup.csv` (lien)

1- Use the code provided to load normal and abnormal data.

2- Visualize normal and abnormal data :

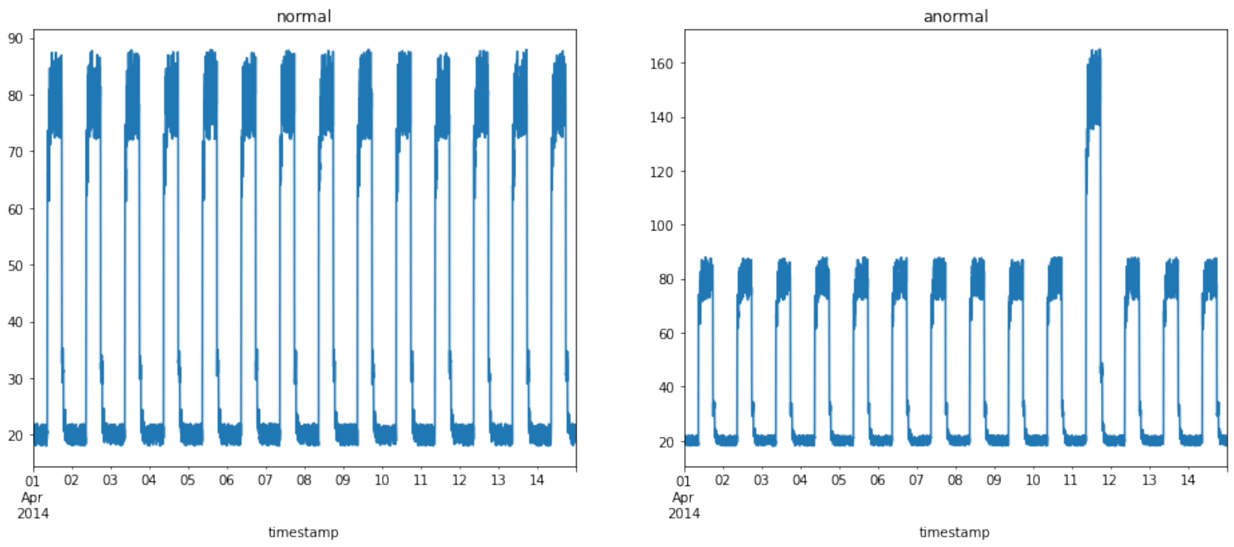

FIGURE 3 – Visualization of normal and abnormal time series

1.2.2 Training

3- Normalize normal data by retrieving the mean and standard deviation. Keep this mean and standard deviation to also normalize abnormal data. As a reminder, the normalization formula is as follows :

$$X_{normalized} = \frac{X - mean}{std} \tag{1}$$

4- To create the training dataset, divide the normal series into a set of subsequences of size $T = 300$ using a sliding window. To avoid being dependent on this window size, use a sliding step of 1. You should get 3733 subsequences of sizes $T = 300$.

5- Visualize some subsequences :

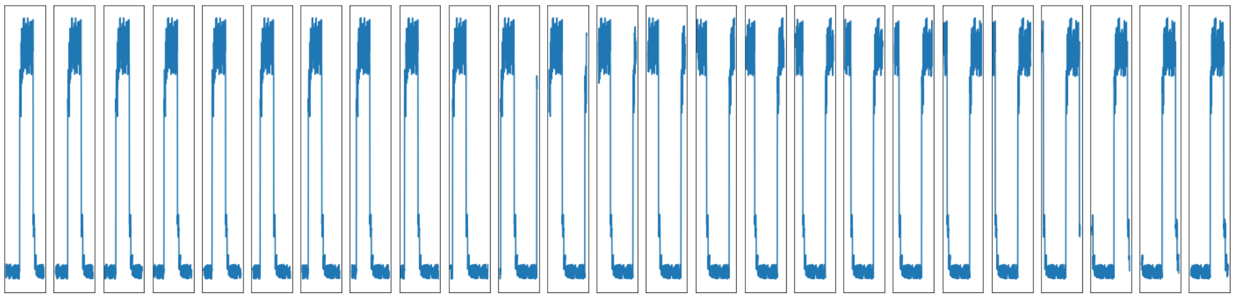

FIGURE 4 – Visualization of normal subsequences

6- Build an autoencoder based on 1D convolution layers, for example as in the figure below. Choose the Adam optimizer with a learning rate of 0.001. Use the "Mean Squared Error" (MSE) as loss.

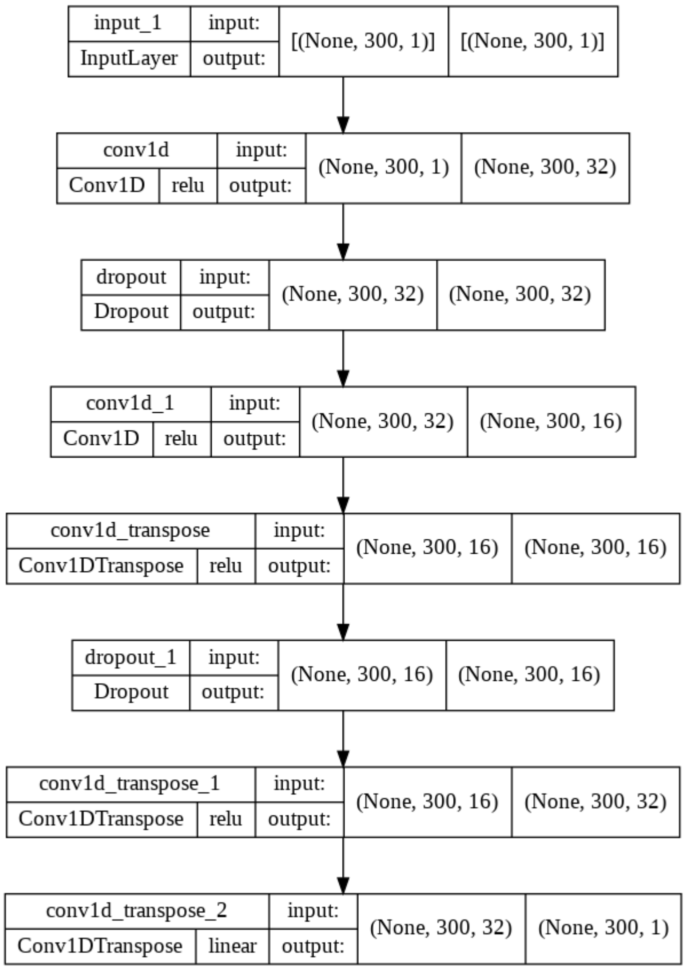
input_1 | InputLayer | input: [(None, 300, 1)] | output: [(None, 300, 1)]
conv1d | Conv1D | relu | input: (None, 300, 1) | output: (None, 300, 32)
dropout | Dropout | input: (None, 300, 32) | output: (None, 300, 32)
conv1d_1 | Conv1D | relu | input: (None, 300, 32) | output: (None, 300, 16)
conv1d_transpose | Conv1DTranspose | relu | input: (None, 300, 16) | output: (None, 300, 16)
dropout_1 | Dropout | input: (None, 300, 16) | output: (None, 300, 16)
conv1d_transpose_1 | Conv1DTranspose | relu | input: (None, 300, 16) | output: (None, 300, 32)
conv1d_transpose_2 | Conv1DTranspose | linear | input: (None, 300, 32) | output: (None, 300, 1)

FIGURE 5 – Autoencoder architecture based on 1D convolutions

7- Train your model from normal subsequences on 50 epochs with a batch size of 128. You can use a validation set of 10% (`validation_split=0.1`), to check that the model generalizes well. You can plot the loss curves to check this :

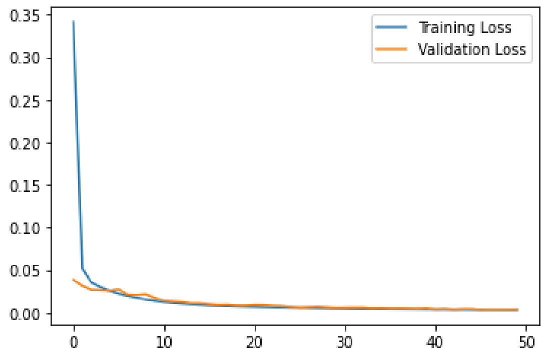

FIGURE 6 – Loss curves during autoencoder training

8- Visualize a reconstruction against an original series to see how the autoencoder is performing :

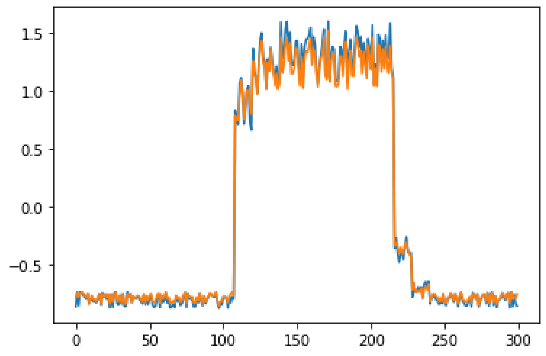

FIGURE 7 – Reconstruction of a series

9- To detect potential anomalies, we will evaluate the reconstruction error for each subseries, then make the decision in relation to a threshold. To define this threshold, we can for example use the reconstruction errors of normal subsequences. Calculate the reconstruction error (e.g. mean absolute error MAE)) for the set of normal subsequences. Retrieve the maximum value that will constitute the anomaly detection threshold. Visualize the distribution of reconstruction errors for all subseries using a histogram (bins=50 in the example) :

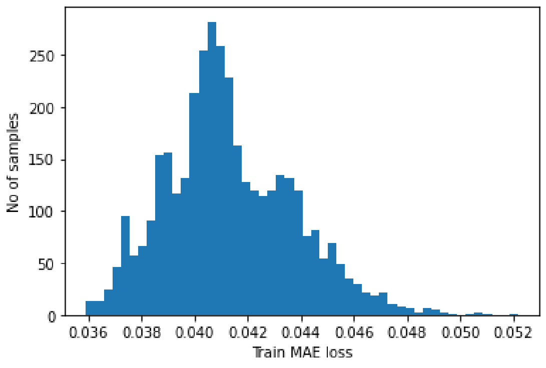

FIGURE 8 – Distribution of reconstruction errors for normal subsequences

1.2.3 Anomaly detection

10- We will now move on to evaluating anomalous data to detect anomalies. As with training, start by normalizing the data, **but using the mean and standard deviation variables** found during training.

11- Create subsequences of size $T = 300$ from the anomalous data.

12- Use the trained autoencoder to predict the reconstruction of anomalous subsequences and calculate the overall reconstruction errors. You can visualize the distribution of errors using a histogram. The majority of errors are small, except for a few subsequences : the anomalies.

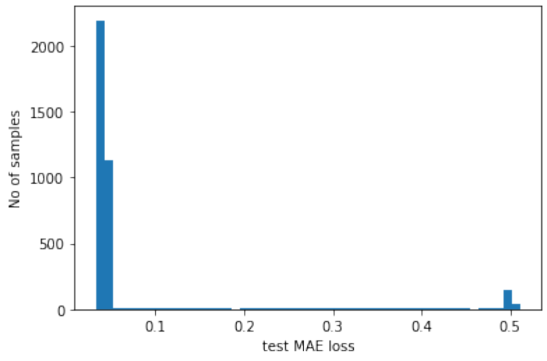

FIGURE 9 – Distribution of reconstruction errors for anomalous subsequences

13- Retrieve the indices of the subsequences for which the error is greater than the threshold defined above. You can display the corresponding parts in red to view the errors :

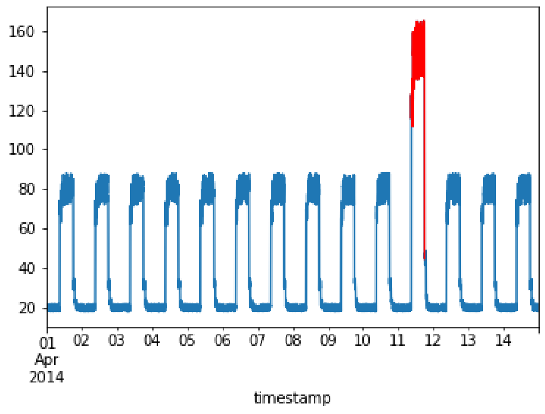

FIGURE 10 – Visualization of abnormal subsequences

2 Variational AutoEncoder

2.1 Exercise 3 : Movement generation

3D skeleton data

We will use the MSR Action 3D dataset (https://sites.google.com/view/wanqingli /data-sets/msr-action3d). This dataset contains sequences of 3D skeletons extracted from a Kinect depth camera observing people performing different actions. Each skeleton contains a set of 20 3D joints corresponding to different parts of the body.
The data can be found here :

— Skeleton data (X.npy) : https://maxime-devanne.com/datasets/MSRAction3D/X.npy
— Action labels (Y.npy) : https://maxime-devanne.com/datasets/MSRAction3D/Y.npy

2.1.1 Loading and observing data

1- The first thing to do is load the datasets. Try to understand how each file is organized to find 3D skeleton information.

2- Then you can, for example, use the provided visualization functions to view some sequences.

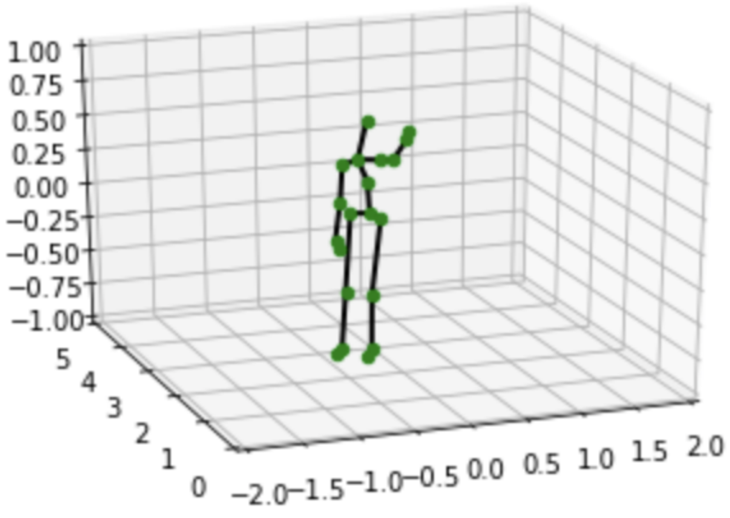

FIGURE 11 – Visualization of a sequence of skeletons

2.1.2 Variational AutoEncoder

3- Implement a Variational AutoEncoder to generate a random skeleton sequence. An idea for an architecture is below (you can of course choose your own architecture) :

Encoder :

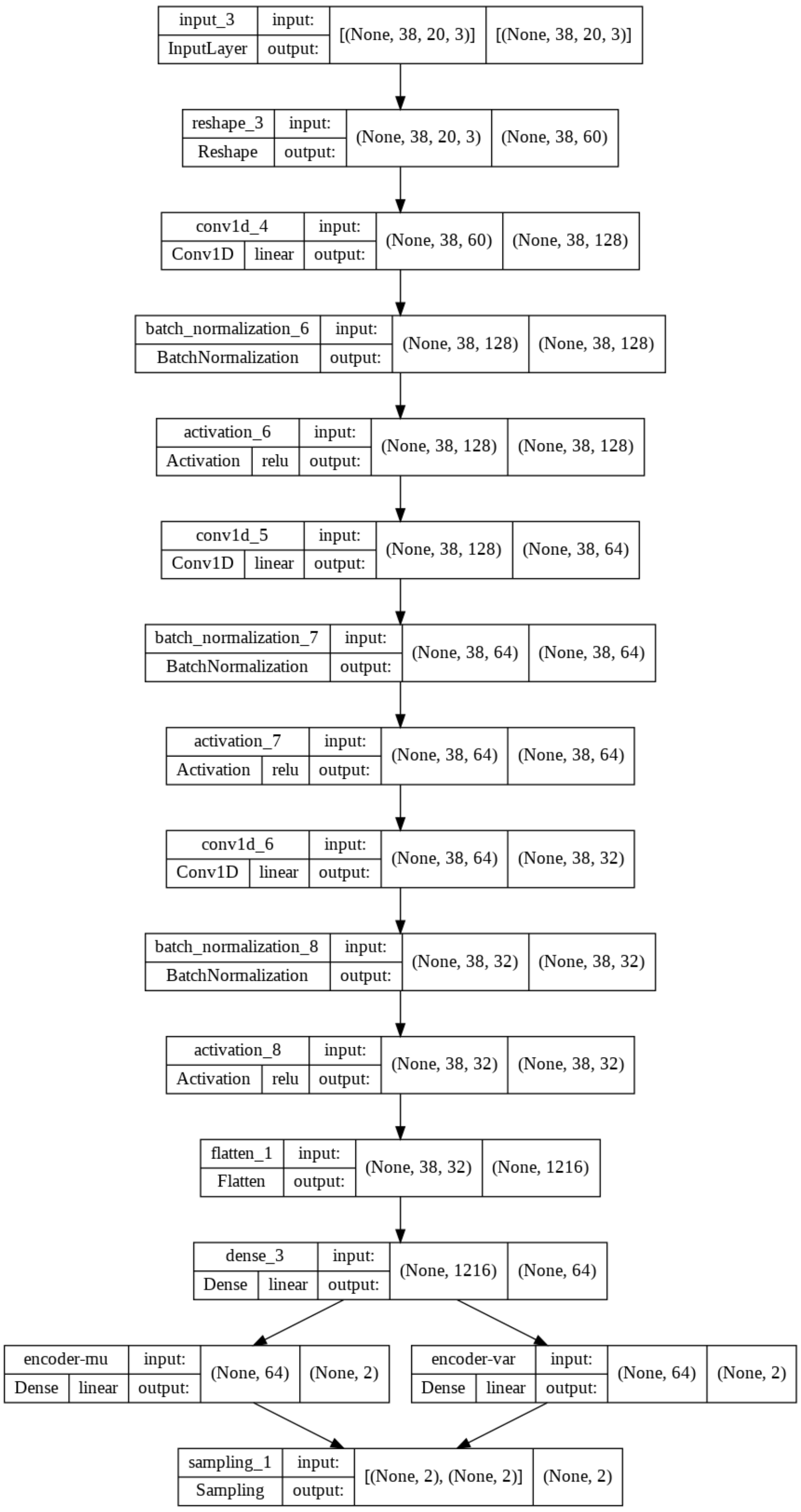
input_3 | InputLayer | input: | output: | [(None, 38, 20, 3)] | [(None, 38, 20, 3)]
reshape_3 | Reshape | input: | output: | (None, 38, 20, 3) | (None, 38, 60)
conv1d_4 | Conv1D | linear | input: | output: | (None, 38, 60) | (None, 38, 128)
batch_normalization_6 | BatchNormalization | input: | output: | (None, 38, 128) | (None, 38, 128)
activation_6 | Activation | relu | input: | output: | (None, 38, 128) | (None, 38, 128)
conv1d_5 | Conv1D | linear | input: | output: | (None, 38, 128) | (None, 38, 64)
batch_normalization_7 | BatchNormalization | input: | output: | (None, 38, 64) | (None, 38, 64)
activation_7 | Activation | relu | input: | output: | (None, 38, 64) | (None, 38, 64)
conv1d_6 | Conv1D | linear | input: | output: | (None, 38, 64) | (None, 38, 32)
batch_normalization_8 | BatchNormalization | input: | output: | (None, 38, 32) | (None, 38, 32)
activation_8 | Activation | relu | input: | output: | (None, 38, 32) | (None, 38, 32)
flatten_1 | Flatten | input: | output: | (None, 38, 32) | (None, 1216)
dense_3 | Dense | linear | input: | output: | (None, 1216) | (None, 64)
encoder-mu | Dense | linear | input: | output: | (None, 64) | (None, 2)
encoder-var | Dense | linear | input: | output: | (None, 64) | (None, 2)
sampling_1 | Sampling | input: | output: | [(None, 2), (None, 2)] | (None, 2)

FIGURE 12 – Encoder architecture

Decoder :

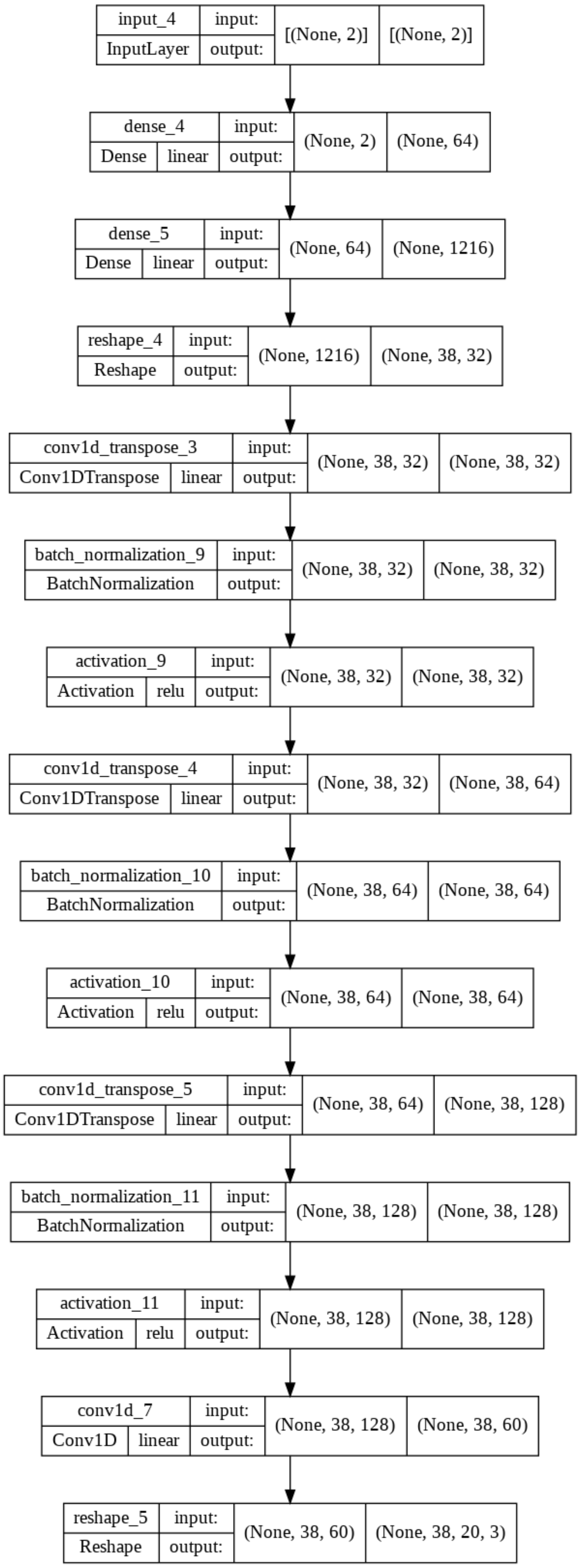
input_4 | InputLayer | input: | output: | [(None, 2)] | [(None, 2)]
dense_4 | Dense | linear | input: | output: | (None, 2) | (None, 64)
dense_5 | Dense | linear | input: | output: | (None, 64) | (None, 1216)
reshape_4 | Reshape | input: | output: | (None, 1216) | (None, 38, 32)
conv1d_transpose_3 | Conv1DTranspose | linear | input: | output: | (None, 38, 32) | (None, 38, 32)
batch_normalization_9 | BatchNormalization | input: | output: | (None, 38, 32) | (None, 38, 32)
activation_9 | Activation | relu | input: | output: | (None, 38, 32) | (None, 38, 32)
conv1d_transpose_4 | Conv1DTranspose | linear | input: | output: | (None, 38, 32) | (None, 38, 64)
batch_normalization_10 | BatchNormalization | input: | output: | (None, 38, 64) | (None, 38, 64)
activation_10 | Activation | relu | input: | output: | (None, 38, 64) | (None, 38, 64)
conv1d_transpose_5 | Conv1DTranspose | linear | input: | output: | (None, 38, 64) | (None, 38, 128)
batch_normalization_11 | BatchNormalization | input: | output: | (None, 38, 128) | (None, 38, 128)
activation_11 | Activation | relu | input: | output: | (None, 38, 128) | (None, 38, 128)
conv1d_7 | Conv1D | linear | input: | output: | (None, 38, 128) | (None, 38, 60)
reshape_5 | Reshape | input: | output: | (None, 38, 60) | (None, 38, 20, 3)

FIGURE 13 – Decoder architecture

4- Train your VAE from the MSR dataset. Remember to normalize the data beforehand (the function is available).

5- Use your trained VAE to generate a random sequence (you can use the `np.random.normal(...)` method on the latent space.

6- Visualize your generated sequence. Remember to do the “denormalization” before (the function is available).

bonus- Transform your VAE into conditional VAE to generate a specific class of movement.