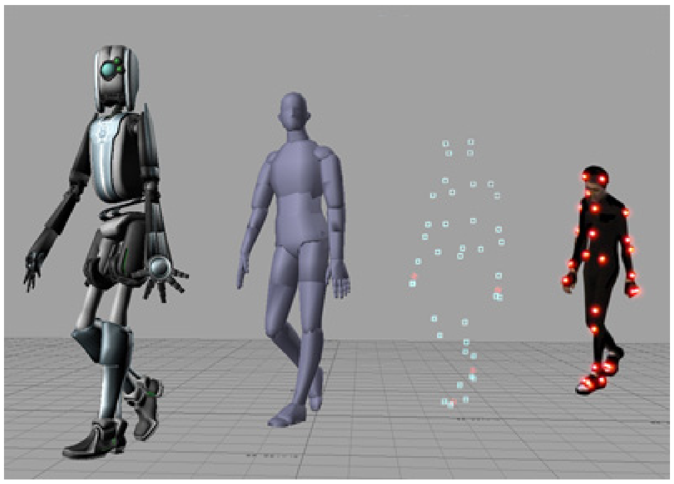
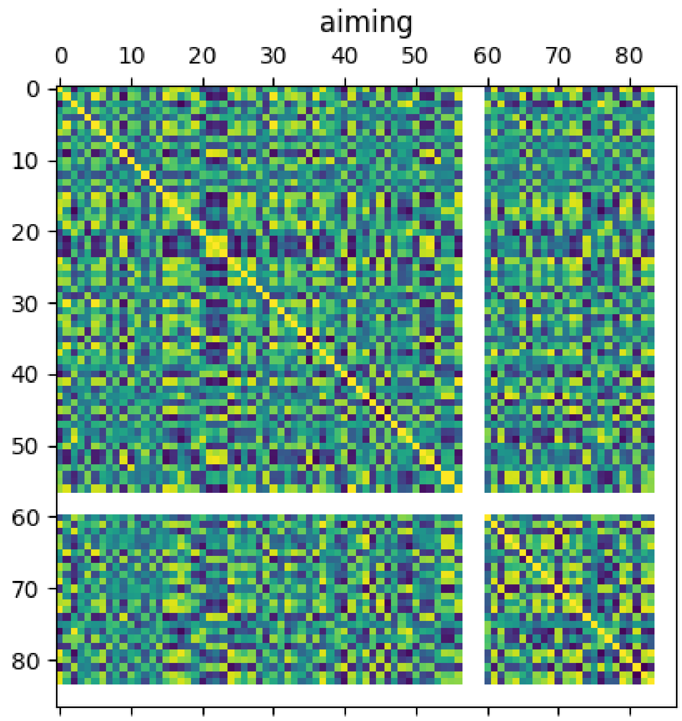
Exploratory Data Analysis for Stylized Motion Capture Classification

By Sohail Sayeed, Adnan Ali

Shreyas Shirsekar, and Pablo Bendiksen



(*Computer animation* 2022)



Observe your dataset

Our dataset comes in the form of visually inspected motion capture (Mocap) data, in bvh format (see here). *Each file captures a simple 'atomic' motion; currently we've gathered data with respect to the motions of 'walking', 'jumping', 'running' and 'waving' (Durupinar).* Furthermore, each motion sequence file was 'stylized' to generate 32 distinct 'drives' which are understood as the expression of three out of four 'efforts.' Thusly, efforts can be viewed as parameters that, according to the underlying theory of Laban Motion Analysis (LMA), are known to capture the perceivable dynamics of human motion, with expressions of combinations of three efforts (i.e., drives) being the most commonly employed by human beings. Please read here to learn more about this formal description of human motion and to check out our on-going project code base.

Each bvh file contains information describing a joint hierarchy for a bipedal model, with positional offset values of a given child joint from its parent, as well as the very motion data used for subsequent analyses. This motion data comes in the form of frames (rows), each of which contains features (columns) that encode the relative rotation of a given joint, in Z, X, Y global coordinates, relative to its parent (i.e., a total of three columns per joint listed in the hierarchy section), and, moreover, also contain the absolute position of the root joint, as global X, Y, Z coordinates. This information is sufficient for the construction of a pose per frame, with the collection of poses across frames representing a motion sequence.

In short, our dataset contains digital information that was extracted from a motion capture suit that performed various different common actions and is used to compile a sequence of digital images in order to create a digital animation.

Furthermore, our dataset comes from a sample size of about 200 different bvh files that contain positional and rotational data split between the four different actions performed: walking, jumping, running, and waving. Within each of these 200 files, there ranges between 50 and 400 different frames of animation depending on the complexity of the action. For instance, jumping on average has much more frames than walking. Each of these frames of animation are created by a variety of features that correspond to the positional data of the root body part—the hips in our case—followed by the rotational data of each child body part. In essence, each feature of our dataset is a body part of the Mocap suit, i.e., spine, legs, head, arms, hands, fingers, etc. Our main target of our dataset is to predict any combination of three of the four LMA effort values for any given movement.

Analyze your data

Our data contains no missing values or missing variables. The bvh file contains all the necessary information that we need to animate a motion sequence. The date contains both categorical and numerical features by nature. The root joint positions are real numbers to denote global coordinates across the Z, X, and Y axes, and all other joint values are real numbers that denote Euler angles, relative to the corresponding parent joint.

The Laban Movement Analysis is the governing theory for how each bvh file gets stylized. We are focusing only on its Effort description: the four effort parameters (whose expressions of three out of the four parameters we generate via an automated tool created by Dr. Funda Durupinar) are Space, Weight, Time and Flow (Durupinar).

Effort tells us the characteristics of movement based on human inner attitudes across the elements of Space, Weight, Time and Flow. Shape tells us what posture the body is in and is expressed based on the Body and Effort value. Space tells us about the environment and surroundings of the body and could be expressed as the state that the body is in in regard to how it interacts with the world (i.e., what direction it is facing and where it can move to next). Again, we are only focusing on Effort values when adding style to the base motion sequences.

The dataset can be considered representative as long as we continue gathering an exemplary set of 'atomic' motions to train on, while maintaining equivalent class representations across training / testing splits. The dataset may be considered imbalanced in the sense that classes (i.e., Drives) vary in frequencies relative to one another. This resulted from the fact that some Drives (e.g., Drives expressing *Sudden* Time) modify motions to the end of greatly reducing their frame count, in effect eliminating themselves from the prepped data given the use of a time series sliding window of a fixed size. Training takes the form of a neural network that can predict the Drive associated with the stylized motion sequence. We curbed the disproportionate class representation by selecting base motion sequences with frame counts ranging from around 300 to 700 (fixing frame rate at .0333, that is, 30 frames per second). Of course, this selection criteria did not apply to motions already having been selected prior to the project being handed to us (see here: https://github.com/pablobendiksen/motion-similarity/tree/main/data/effort)

The ranges for our numerical data are expressed in Euler angles that vary from 0 to 2pi across the X, Y, Z axes given that we are working with three dimensional space. Since we used Euler angles, there are no outlier values since they only represent angles about a reference axis which are entirely necessary for bvh parsers to construct poses.

Transform your dataset

- *This section is devoted to a deep dive within our dataset.*

Absolute Position –

Hips_Xposition', 'Hips_Yposition', 'Hips_Zposition',

Rotations –

'Hips_Xposition', 'Hips_Yposition', 'Hips_Zposition', 'Hips_Zrotation', 'Hips_Xrotation', 'Hips_Yrotation', 'LeftUpLeg_Zrotation', 'LeftUpLeg_Xrotation', 'LeftUpLeg_Yrotation', 'LeftLeg_Zrotation', 'LeftLeg_Xrotation', 'LeftLeg_Yrotation', 'LeftFoot_Zrotation', 'LeftFoot_Xrotation', 'LeftFoot_Yrotation', 'LeftToeBase_Zrotation', 'LeftToeBase_Xrotation', LeftToeBase_Yrotation', 'RightUpLeg_Zrotation', 'RightUpLeg_Xrotation', 'RightUpLeg_Yrotation', 'RightLeg_Zrotation', 'RightLeg_Xrotation', 'RightLeg_Yrotation', 'RightFoot_Zrotation', RightFoot_Xrotation', 'RightFoot_Yrotation', 'RightToeBase_Zrotation', 'RightToeBase_Xrotation', 'RightToeBase_Yrotation', 'Spine_Zrotation', 'Spine_Xrotation', 'Spine_Yrotation', 'Spine1_Zrotation', 'Spine1_Xrotation', 'Spine1_Yrotation', 'Spine2_Zrotation', 'Spine2_Xrotation', 'Spine2_Yrotation', 'LeftShoulder_Zrotation', 'LeftShoulder_Xrotation', 'LeftShoulder_Yrotation', 'LeftArm_Zrotation', 'LeftArm_Xrotation', 'LeftArm_Yrotation', 'LeftForeArm_Zrotation', 'LeftForeArm_Xrotation', 'LeftForeArm_Yrotation', 'LeftHand_Zrotation', 'LeftHand_Xrotation', 'LeftHand_Yrotation', 'LeftHandIndex1_Zrotation', 'LeftHandIndex1_Xrotation', 'LeftHandIndex1_Yrotation', 'LeftHandIndex2_Zrotation', 'LeftHandIndex2_Xrotation', 'LeftHandIndex2_Yrotation', 'LeftHandIndex3_Zrotation', 'LeftHandIndex3_Xrotation', 'LeftHandIndex3_Yrotation', 'Neck_Zrotation', 'Neck_Xrotation', 'Neck_Yrotation', 'Head_Zrotation', 'Head_Xrotation', 'Head_Yrotation', 'RightShoulder_Zrotation', 'RightShoulder_Xrotation', 'RightShoulder_Yrotation', 'RightArm_Zrotation', 'RightArm_Xrotation', 'RightArm_Yrotation', 'RightForeArm_Zrotation', 'RightForeArm_Xrotation', 'RightForeArm_Yrotation', 'RightHand_Zrotation', 'RightHand_Xrotation', 'RightHand_Yrotation', 'RightHandIndex1_Zrotation', 'RightHandIndex1_Xrotation', 'RightHandIndex1_Yrotation', 'RightHandIndex2_Zrotation', 'RightHandIndex2_Xrotation', 'RightHandIndex2_Yrotation', 'RightHandIndex3_Zrotation', 'RightHandIndex3_Xrotation', 'RightHandIndex3_Yrotation'

All BVH files were first parsed via the pyMO library (https://github.com/omimo/PyMO). The features for each sample are mostly the relative expmap rotation values of each body joint in 3-dimension plane i.e., the X, Y and Z coordinate. In other words, rotation values for a given joint are relative to the position of the corresponding parent joint. Moreover, the 'root' joint (i.e., Hips) absolute position coordinates are included. This is essential as the determination of all joint positions results from a recursive matrix transformation process that begins with the position coordinates of the root joint. In all we have 87 features that comprise our input vector for any neural network model use. We additionally tested the inclusion of joint velocities (see here: link) to the feature set—extending feature count to 186—on model performance.

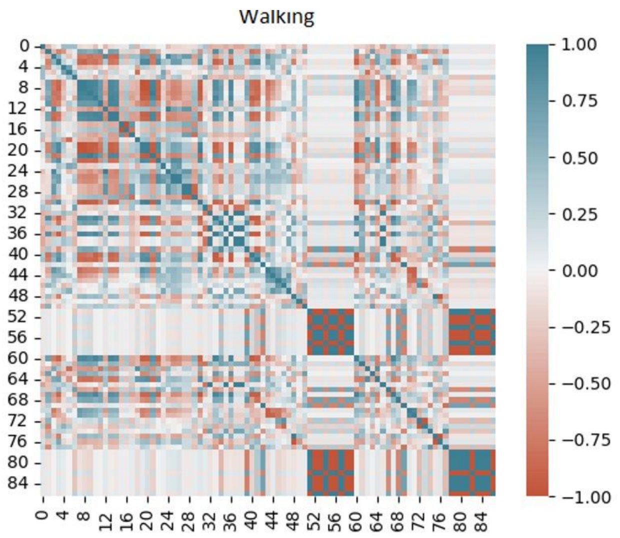
All features were scaled to the standard normal distribution. A sliding window of size 100 frames, with a step size of 1, was applied to our final concatenated dataset of motion numpy arrays in order to ensure uniform exemplars. Our final dataset was randomly partitioned across training and testing sets with a 0.8 train/test split and a seed value of 0. We have no need to one-hot encode our labels since we will be using sparse_categorical_crossentropy instead of categorical_crossentropy; besides, doing so makes for more performant training.

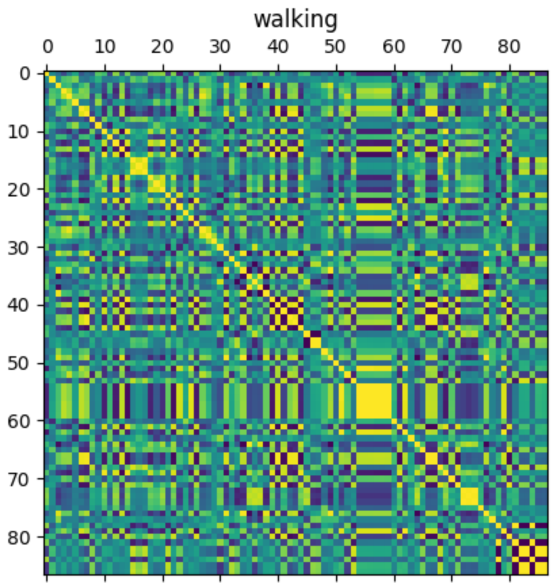
Graphical representation of Correlation Matrix for each type of motion –

A correlation matrix is said to be "square", with the same features shown in the rows and columns. The line going from the top left to the bottom right is the main *diagonal*, which shows that each feature always perfectly correlates with itself. We generated correlation matrices, with respect to the original 87 features, for one bvh file, across our five original action types. Our final dataset included a sixth action type - jumping - not visualized here, in order to make our data even more representative of 'atomic' motions.

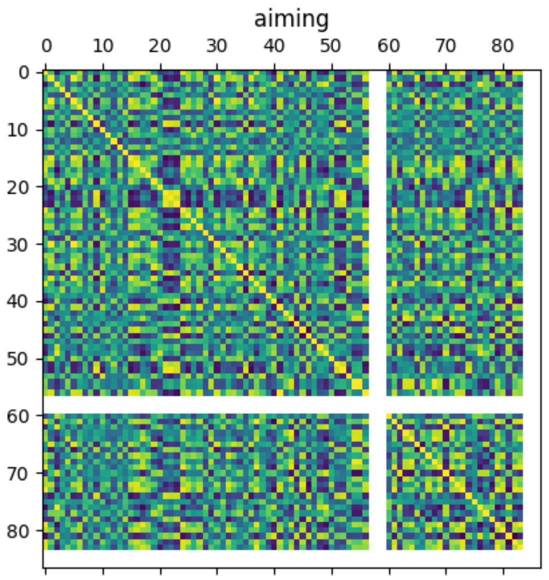
For all of the below graphs, the x-axis and y-axis are mirrored and each number represents a different feature (i.e. 'LeftLeg_Yrotation' or 'LeftFoot_Zrotation'). The scale of the features are also dictated by a color spectrum. For all the graphs except the Walking graph below, yellow denotes a strong expression and blue denotest no expression between the other features being compared. A perfect 50/50 mixture of the colors represents a neutral expression. Some graphs are also missing information about the features and overall, are considered bad training sets.

1. *Walking –*

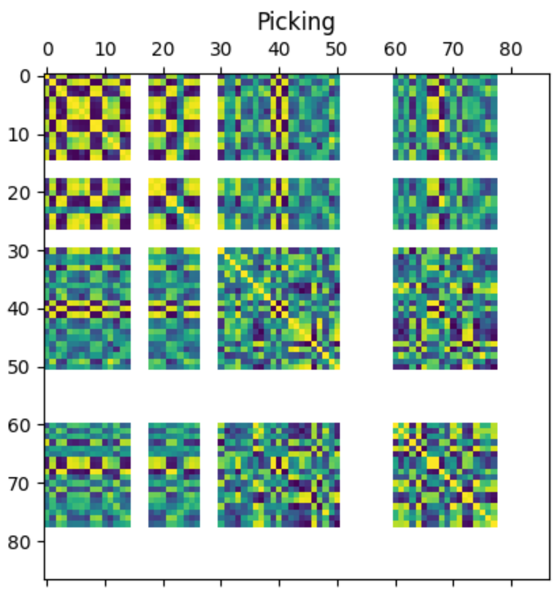




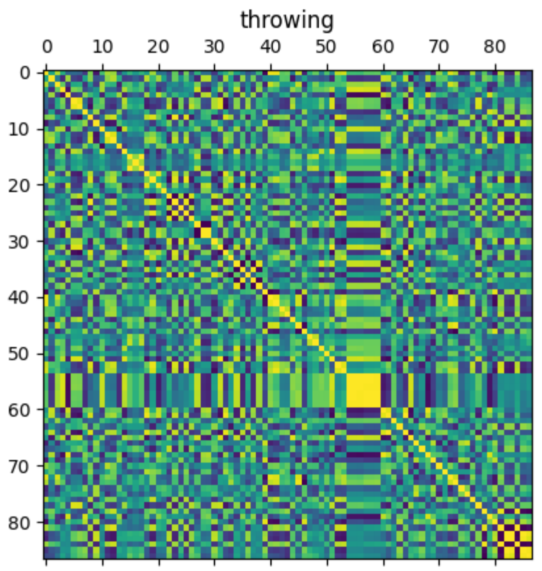
2. *Aiming –*



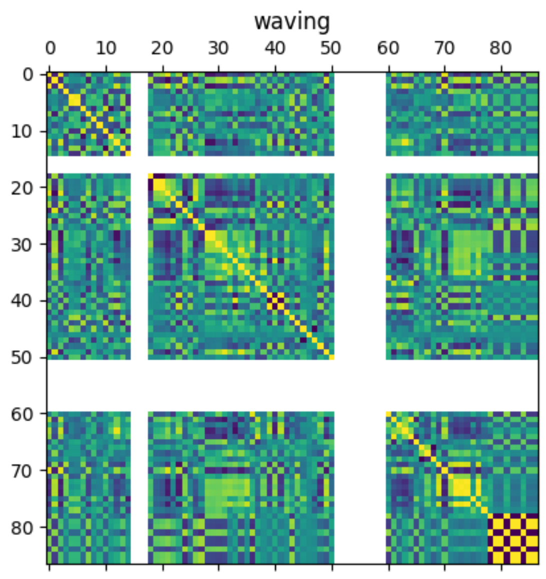
3. *Picking –*



4. *Throwing –*



5. *Waving –*



Each motion is associated with all the 87 features as mentioned above. We have plotted the correlation Matrix of those motions with respect to those features. A correlation matrix plot shows correlation coefficients between features. Each cell in the graph shows the correlation between two features. We have used a correlation matrix to summarize our input data, as an input into a more advanced analysis, and how we would use this feature to analyze the motion.

Formulate your ML task

Our plan for our Neural Network is to make use of Tensorflow and Keras to assist with model training and prediction. Depending on the model used, our task can be viewed as either a regression that seeks to predict a four-valued outcome variable (representing each of four effort values) or, alternatively, a classification task of 33 classes (all Drives plus the neutral, or un-stylized, motion). Lastly, as previously mentioned, we plan on implementing velocities to increase the available input features to a given model.

Think about your potential NN architecture

Of models experimented with we emphasize a 2D Convolutional Neural Network, into our project. This is because we believe the filters of a 2D convolution can best capture correlations across frames of a given joint, as well as across joints of a given frame, in the features they respectively represent. We plan to start with a filter size of either 87 or 183, as these values correspond to the respective input feature sizes. We will also include a filter size of 256 to test a high number of filters relative to input feature number. Moreover, we set kernel size to (3,3), with strides of (1,1,) as these are very commonly employed. We want to experiment with the Loss function: "MSE" when regressing, and "Sparse Categorical Cross-Entropy" when classifying, as both are widely used in the field of Neural Networks. We plan on using the Adaptive Movement Estimation (Adam) optimizer to reduce errors in our predictions and specifically use a method in Adam to compute the adaptive learning rate for each parameter. The method uses both the decaying average of past gradient and decaying average of past squared gradient "RMSProp" algorithm. We fix epoch size to 50 thinking this sufficient for a model to update weights to the best of its capacity. We will explore ReLU activation functions for hidden layers as these are very popular given their robustness to vanishing or exploding gradients. For our output layer we will explore the tanh activation function with respect to our regressors, and the softmax activation function for our classifiers.

Think what metrics will you use for your model evaluation

We will use accuracy as our training metric because our goal is to achieve as close to 100 percent accuracy as possible.

Part 2: Implementation with Tensorflow and Keras

Create custom class for your dataset, then create DataLoader

As opposed to a class for our custom dataset we stored our hand selected bvh files in a directory (see here:). Initial files included in the original project base not withstanding, we selected our data from the openly available CMU bvh motion capture dataset (see here: http://mocap.cs.cmu.edu/motcat.php?maincat=3). However, since this database does not include files in the bvh format, we were given access to all corresponding files of the database in bvh format by Professor Funda (see here: https://drive.google.com/drive/folders/1mdNvXL7tNiCneh69bpaQTyA84IDX14D6?usp=sharing). We visually inspected each motion file using the Blender software prior to their incorporation into the data directory. As opposed to the use of DataLoader, we pre-processed our data using the pyMO library and our own code for data preparation (see here: https://github.com/pablobendiksen/motion-similarity/blob/main/organize_synthetic_data.py#:~:text=def%20load_data(rotations%20%3D%20True%2C%20velocities%3DFalse)%3A).

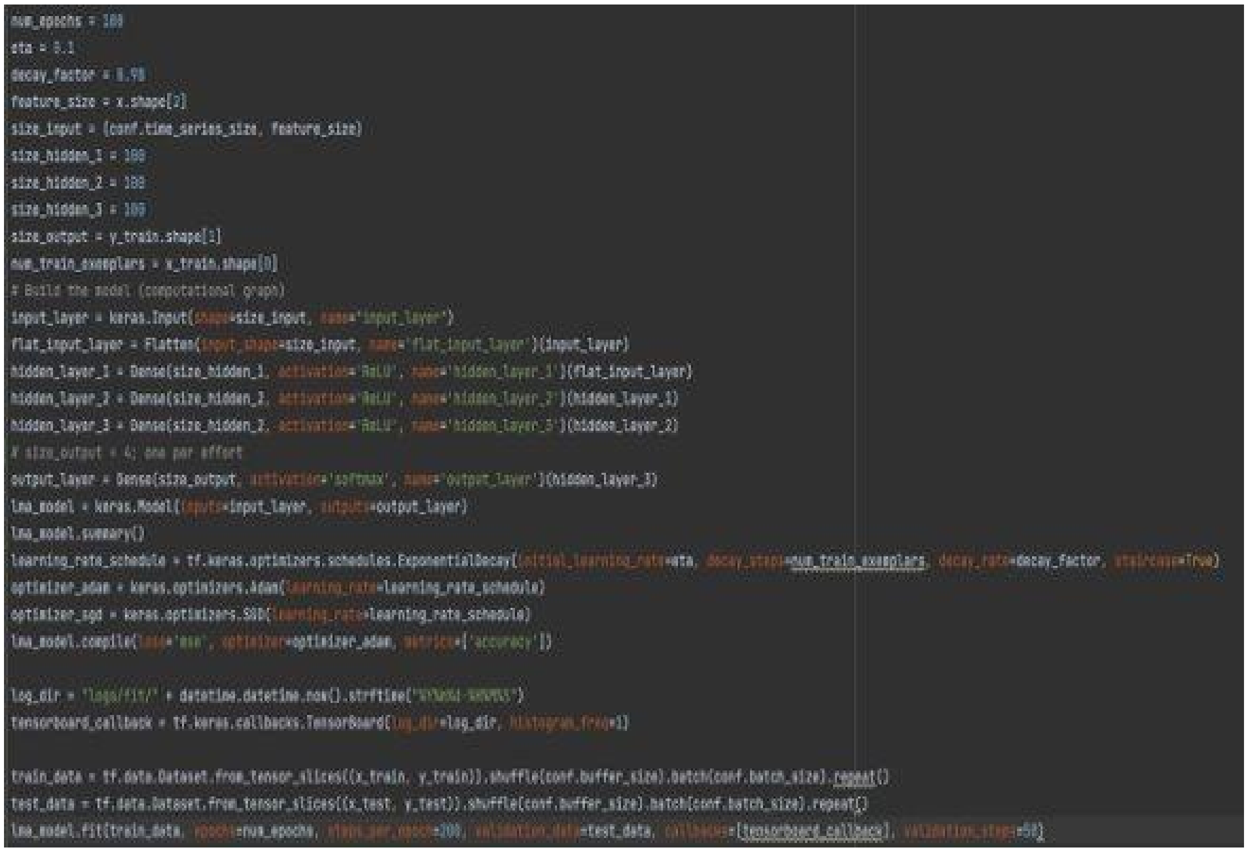
Specify loss, optimizer and write training and test steps in separate functions

Our plan for our use of Neural Networks is to make use of Tensorflow to help with model training and prediction. We want to implement various models to the end of maximizing Drive predictions. We want to create the Loss function: “MSE.” We plan on using the Adaptive Movement Estimation (Adam) optimizer, given its popularity, to reduce errors in our predictions and specifically use a method in Adam to compute the adaptive learning rate for each parameter.

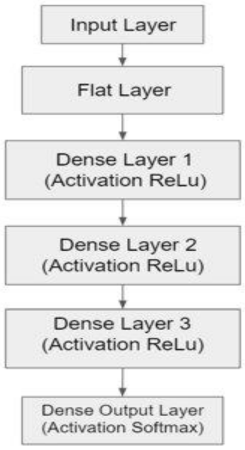
Make your model initialization, training and test

1st. Attempt: *Dense Network*

Our first attempt was to create a Dense Network. This was effective since it yielded a 100% accuracy, however the system could be improved upon since it was slow and it required 183 epochs in order to get that accuracy.

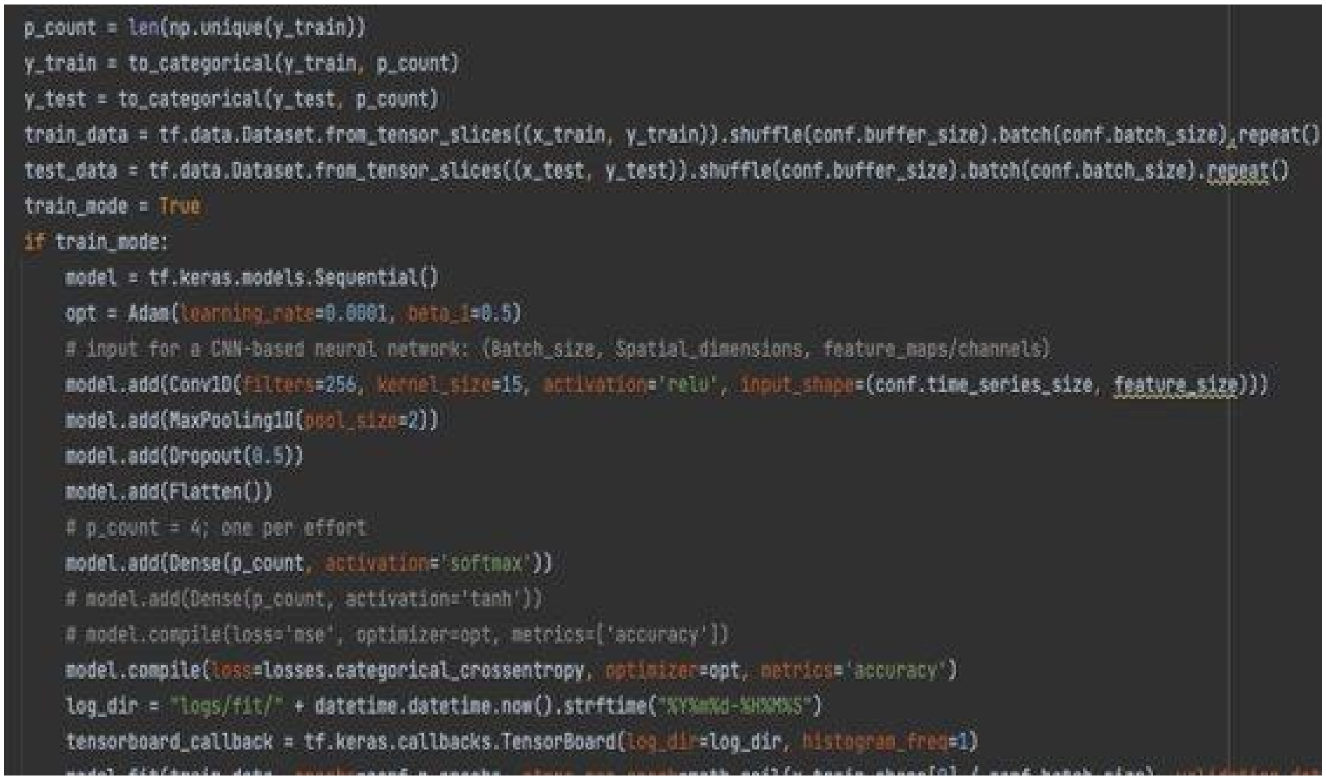


```
num_epochs = 180
eta = 0.1
decay_factor = 0.98
feature_size = x.shape[2]
size_input = (conf.time_series_size, feature_size)
size_hidden_1 = 100
size_hidden_2 = 100
size_hidden_3 = 100
size_output = y_train.shape[1]
num_train_exemplars = x_train.shape[0]
# Build the model (computational graph)
input_layer = keras.Input(shape=size_input, name='input_layer')
flat_input_layer = Flatten(input_shape=size_input, name='flat_input_layer')(input_layer)
hidden_layer_1 = Dense(size_hidden_1, activation='ReLU', name='hidden_layer_1')(flat_input_layer)
hidden_layer_2 = Dense(size_hidden_2, activation='ReLU', name='hidden_layer_2')(hidden_layer_1)
hidden_layer_3 = Dense(size_hidden_2, activation='ReLU', name='hidden_layer_3')(hidden_layer_2)
# size_output = 4; one per effort
output_layer = Dense(size_output, activation='softmax', name='output_layer')(hidden_layer_3)
lma_model = keras.Model(inputs=input_layer, outputs=output_layer)
lma_model.summary()
learning_rate_schedule = tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=eta, decay_steps=num_train_exemplars, decay_rate=decay_factor, staircase=True)
optimizer_adam = keras.optimizers.Adam(learning_rate=learning_rate_schedule)
optimizer_sgd = keras.optimizers.SGD(learning_rate=learning_rate_schedule)
lma_model.compile(loss='mse', optimizer=optimizer_adam, metrics=['accuracy'])

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

train_data = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(conf.buffer_size).batch(conf.batch_size).repeat()
test_data = tf.data.Dataset.from_tensor_slices((x_test, y_test)).shuffle(conf.buffer_size).batch(conf.batch_size).repeat()
lma_model.fit(train_data, epochs=num_epochs, steps_per_epoch=200, validation_data=test_data, callbacks=[tensorboard_callback], validation_steps=50)
```

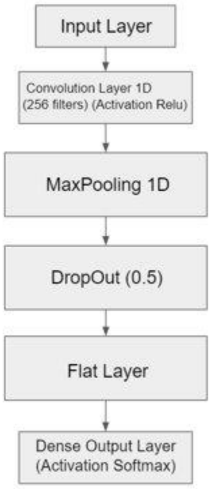

Input Layer
Flat Layer
Dense Layer 1 (Activation ReLu)
Dense Layer 2 (Activation ReLu)
Dense Layer 3 (Activation ReLu)
Dense Output Layer (Activation Softmax)

2nd. Attempt: *Convolutional Neural Network 1D*

The second attempt yielded both positive and negative results. We managed to reduce the time and epochs to reach a steady accuracy result, but we were unable to achieve 100% accuracy. So some progress was made while other was lost.

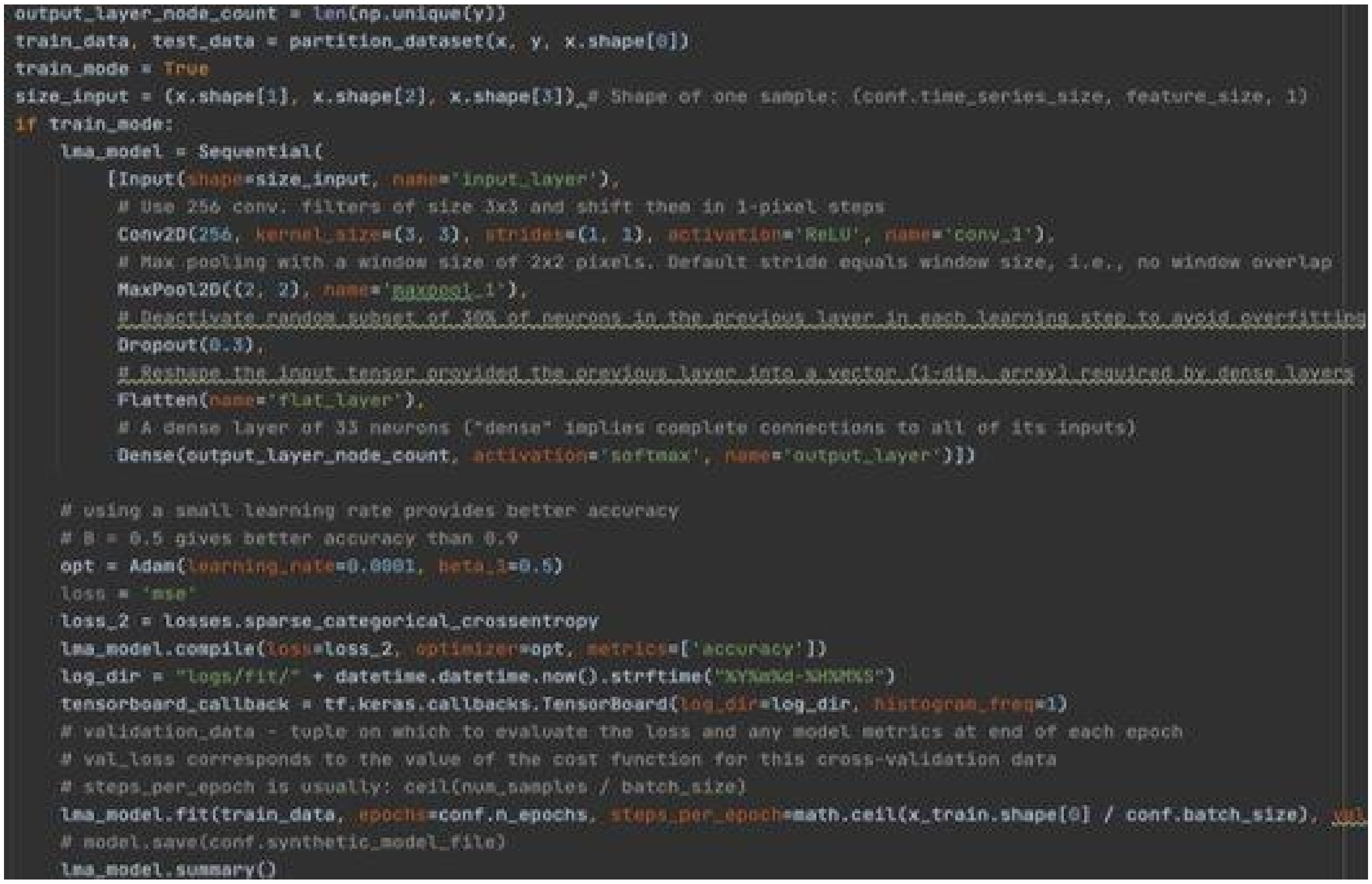


```
p_count = len(np.unique(y_train))
y_train = to_categorical(y_train, p_count)
y_test = to_categorical(y_test, p_count)
train_data = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(conf.buffer_size).batch(conf.batch_size).repeat()
test_data = tf.data.Dataset.from_tensor_slices((x_test, y_test)).shuffle(conf.buffer_size).batch(conf.batch_size).repeat()
train_mode = True
if train_mode:
    model = tf.keras.models.Sequential()
    opt = Adam(learning_rate=0.0001, beta_1=0.5)
    # input for a CNN-based neural network: (Batch_size, Spatial_dimensions, feature_maps/channels)
    model.add(Conv1D(filters=256, kernel_size=15, activation='relu', input_shape=(conf.time_series_size, feature_size)))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Dropout(0.5))
    model.add(Flatten())
    # p_count = 4; one per effort
    model.add(Dense(p_count, activation='softmax'))
    # model.add(Dense(p_count, activation='tanh'))
    # model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])
    model.compile(loss=losses.categorical_crossentropy, optimizer=opt, metrics='accuracy')
    log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
```

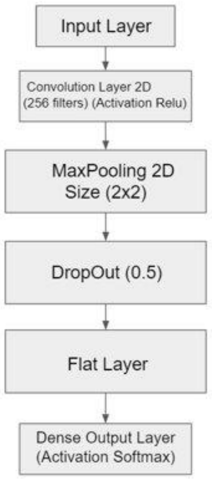

Input Layer
Convolution Layer 1D (256 filters) (Activation Relu)
MaxPooling 1D
DropOut (0.5)
Flat Layer
Dense Output Layer (Activation Softmax)

3rd. Attempt: *Convolutional Neural Network 2D*

On the third attempt we achieved the results that we wanted. We were able to achieve 100% accuracy while reducing the amount of epochs and cutting the time in half compared to the Dense Network approach.



```
output_layer_node_count = len(np.unique(y))
train_data, test_data = partition_dataset(x, y, x.shape[0])
train_mode = True
size_input = (x.shape[1], x.shape[2], x.shape[3])_# Shape of one sample: (conf.time_series_size, feature_size, 1)
if train_mode:
    lma_model = Sequential(
        [Input(shape=size_input, name='input_layer'),
         # Use 256 conv. filters of size 3x3 and shift them in 1-pixel steps
         Conv2D(256, kernel_size=(3, 3), strides=(1, 1), activation='ReLU', name='conv_1'),
         # Max pooling with a window size of 2x2 pixels. Default stride equals window size, i.e., no window overlap
         MaxPool2D((2, 2), name='maxpool_1'),
         # Deactivate random subset of 30% of neurons in the previous layer in each learning step to avoid overfitting
         Dropout(0.3),
         # Reshape the input tensor provided the previous layer into a vector (1-dim. array) required by dense layers
         Flatten(name='flat_layer'),
         # A dense layer of 33 neurons ("dense" implies complete connections to all of its inputs)
         Dense(output_layer_node_count, activation='softmax', name='output_layer')])

    # using a small learning rate provides better accuracy
    # B = 0.5 gives better accuracy than 0.9
    opt = Adam(learning_rate=0.0001, beta_1=0.5)
    loss = 'mse'
    loss_2 = losses.sparse_categorical_crossentropy
    lma_model.compile(loss=loss_2, optimizer=opt, metrics=['accuracy'])
    log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
    # validation_data - tuple on which to evaluate the loss and any model metrics at end of each epoch
    # val_loss corresponds to the value of the cost function for this cross-validation data
    # steps_per_epoch is usually: ceil(num_samples / batch_size)
    lma_model.fit(train_data, epochs=conf.n_epochs, steps_per_epoch=math.ceil(x_train.shape[0] / conf.batch_size), val
    # model.save(conf.synthetic_model_file)
    lma_model.summary()
```


Input Layer
Convolution Layer 2D (256 filters) (Activation Relu)
MaxPooling 2D Size (2x2)
DropOut (0.5)
Flat Layer
Dense Output Layer (Activation Softmax)

Split your training set into train and validation sets. As a result, you will have 3 sets: train, validation and test.

We split our dataset into training and testing sets only, along a 80/20 divide. We chose not to create a validation set because we wanted to maximize the number of training exemplars in order to best achieve perfect accuracy, using our test set as our validation set within each model's training workflow. The process of manually selecting and inspecting candidate exemplars is a time-consuming process but, as we add more data to our training set we will consider the need for a validation set.

Train your model

We did three trials to finalize the model for training.

1. Dense Network -

   Input layer - > Flat Layer -> Dense Layer 1 (Activation ReLu) - > Dense Layer 2 (Activation ReLu) - > Dense 4 Node Layer (Activation Softmax)

2. 1D Convolution Network

   Input Layer -> Convolutional Layer 1D (256 filters) (Activation Relu) -> Max Pooling 1D -> Dropout (0.5) -> Flat Layer -> Dense Output Layer (Activation Softmax)

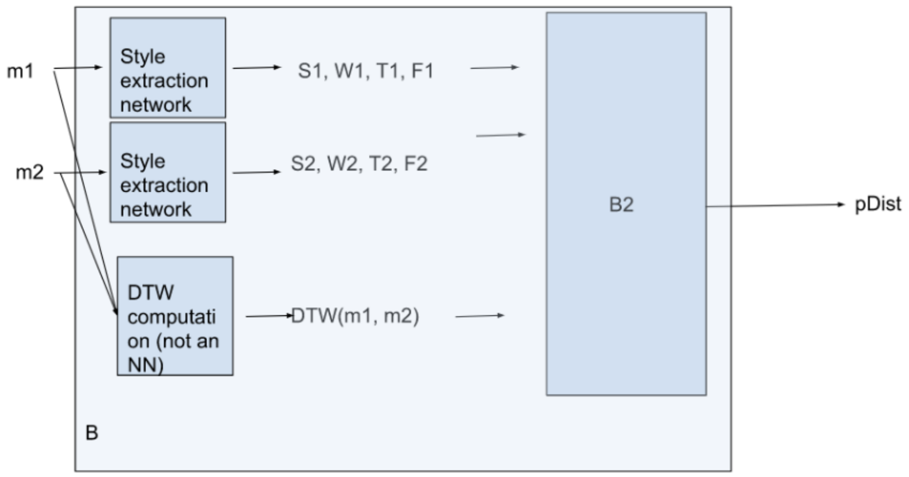
3. 2D Convolution Layer

   Input Layer -> Convolution Layer 2D (183 filters) (Activation Relu) -> Max Pooling 2D (Size 2*2) -> Dropout (0.5) -> Flat Layer -> Dense 33-Node Output Layer (Activation Softmax)

The final model was able to achieve high Training and validation accuracy which was able to serve our use case in predicting the Effort label for a given frame of motion.

Evaluate your model on the test set

We title the resultant model our ***Style Extraction Network***. This network will be positioned as one part of a greater system, geared towards devising a perceived similarity metric between any two motions. Such a metric, if implemented correctly, will have direct effects on the open problem of character motion synthesis.



This image depicts the predicted drives (e.g., [S1, W1, T1, F1]) from our neural network (Style extraction network(s)) that will then be used in a future model (B2) that has yet to be implemented.

Fine-Tune/improve your model

Our 'model tuning' took the form of an iterative approach wherein we tested more than one model. We experimented with different filter sizes for Convolutional Neural Networks (CNNs) as well as two forms of regularization - dropout and Max Pooling. What made a few interesting observations:

1) Having started with a dense network, we found that perfect accuracy could not be achieved with either 87 or 183 input features.
2) A 1D CNN was unable to classify all 33 classes perfectly.
3) By contrast, our 2D CNN achieved perfect accuracy across the three filter sizes when considering all 183 input features.
4) Our 2D CNN achieved perfect accuracy for 2 of the filter sizes when inputting the original 87 features.

We summarize our observations regarding CNN model performance in the following table:

CNN	256 Filters	183 Filters	87 Filters
2D (87 features)	x	36 (21 mins)	95 (26 mins)
2D (183 features)	23 (45 mins)	21 (27 mins)	24 (14.5 mins)
1D (87 features)	x	x	x
1D (183 features)	x	x	x

Table 1: Epochs (and training time) to perfect classification for 2 CNNs of varied input feature and filter sizes.

Based on these results we settle for our 2D Convolutional Neural Network, trained on input vectors of 183 features, and with a filter size of 87, as our Style Extraction Network to be relied upon for next stages of research.

Works Cited

Durupinar, F. (2021). Perception of Human Motion Similarity Based on Laban Movement Analysis. *ACM Symposium on Applied Perception 2021*, 1–7. https://doi.org/10.1145/3474451.3476241

Durupinar, F., Kapadia, M., Deutsch, S., Neff, M., & Badler, N. I. (2017). PERFORM: Perceptual Approach for Adding OCEAN Personality to Human Motion Using Laban Movement Analysis. *ACM Transactions on Graphics*, *36*(1), 1–16. https://doi.org/10.1145/2983620

Wikipedia The Free Encyclopedia. (2022). *An example of computer animation using motion capture*. Computer animation. Retrieved December 2, 2022, from https://upload.wikimedia.org/wikipedia/commons/6/6d/Activemarker2.PNG.