School of Computer Science and Statistics

Gesture Generation for Embodied Conversational Agents

Tobias Hallen

Supervisor: Dr Rachel Mc Donnell

A Dissertation submitted in partial fulfilment of the requirements for the degree of a Master’s in Computer Science

Submitted to the University of Dublin, Trinity College
August 9, 2021
Plagiarism Declaration:

I, Tobias Hallen, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not been previously submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend a copy of it or any part thereof on request.

Signed: ___________________________ Date: ___________________________
Acknowledgements

I stand on the shoulders of giants, and much of the work within this dissertation has been made possible by the prior research of other computer scientists.

A special thank you to my supervisor Rachel Mc Donnell, without whom this project would not have been possible. Rachel provided guidance, resources, and direction throughout the project’s development.

And a final thanks to my parents, who have supported me with boundless generosity and endless patience during these last 5 years of study.
Abstract:

Virtual agents will be defined in this paper as a software program which uses scripted rules and, increasingly, machine learning and artificial intelligence in order to provide automated service or guidance to humans [1]. Virtual agents may be more or less feature rich, including capabilities such as speech, animation, etc. The animation of virtual agents is an important field of research. Animation allows for more realistic mimicry of real-world behaviour. It allows users to interact with virtual agents more intuitively and naturally, as it more accurately portrays human-human interaction.

The simplest approach to generating animation in virtual agents is simple hand animation. A skilled worker crafts each animation from their own study of human behaviour. This method yields high-quality results but is limited by the availability of skilled workers and is very time intensive. It may also lead to a lack of variation in the agent’s movement capabilities, as only a set number of animations will be made.

A machine learning approach seeks to generate similarly high-quality expressive animation without the costly and time-intensive downsides of traditional animation.

Deep learning-based motion synthesis methods can probabilistically generate any number of different convincing output animations based on natural language audio inputs, which yields greater variation in gestures than traditional hand-animation.

A transformer-based architecture trained on a data set of aligned natural speech and motion capture data may be employed to interactively generate expressive gestures for virtual agents by supplying natural language text inputs.

In this project, we explore different methods of synthesising gestures by treating it as a sequence-to-sequence problem. We implement a Transformer-based motion synthesis engine which, supplied with motion capture data of a single actor’s performance, as well as a transcription of their speech, can generate new gestures when presented with novel text input data.
Chapter 1 - Introduction and Motivation .........................................................1

Chapter 2 - Background and Related Work ..................................................2

  2.1 - Sequence Transduction ........................................................................2
  2.2 - Recurrent Neural Networks .................................................................3
  2.3 - Long Short-Term Memory Networks (LSTM) .....................................4
  2.4 - Gated Recurrent Units (GRU) .............................................................5
  2.5 - Convolutional Neural Networks ..........................................................6

Chapter 3 - Transformer Neural Networks ..................................................7

  3.1 - Introduction ............................................................................................7
  3.2 - Parallelization .......................................................................................7
  3.3 - Architecture ..........................................................................................9
  3.4 - Attention ...............................................................................................10
  3.5 - Positional Encoding .............................................................................11
  3.6 - Layer Normalization ............................................................................12
  3.7 - Decoder Stack .......................................................................................13
  3.8 - Final Linear Layer ...............................................................................14

Chapter 4 - State of the Art ...........................................................................15

Chapter 5 - Data ............................................................................................17

  5.1 - BVH File Format ..................................................................................17
  5.2 - Original Input Data ...............................................................................20

Chapter 6 - Implementation .........................................................................22

  6.1 - Data Pre-Processing ............................................................................22
  6.2 - Motion Data Extraction .......................................................................24
Chapter 1 - Introduction and Motivation

Embodied conversational agents are often embodied by virtual avatars which interact with humans audio-visually. This interaction can take the form of a text-based chat or an audio-based conversation. In either case, the virtual agent is represented by such an avatar. This virtual avatar is often enhanced by animation to improve realism. User responses to fully animated agents are faster and more accurate when interacting with fully animated agents when compared to non-animated multi-image cues [2].

Creating realistic motion and gesturing for such a virtual avatar improves the user experience. The automatic synthesis of gestures in virtual agents is applicable in various fields such as interviewing [3] [4], online learning [5] [6] [7], and other social interactions [8] [9] [10]. Using traditional animation, one must manually align the motions with conversation. Oftentimes a lack of variation in animations means that the gestures do not match the text or audio correctly. Therefore, it is desirable to automate the synthesis of such gestures pre-aligned with the text of the conversation, in order to reduce costs both time and monetary.

Human-human conversation relies on both the literal meaning of the words exchanged, as well as more abstract concepts in interaction such as body language, gesturing, and facial expression [11]. These subtle movements and poses lend meaning and imply intent, therefore making up an important part of social interaction. By adding synthesized gestures to virtual agents, we seek to add back the subtextual meaning of subtle motions and gestures between human parties in conversation to the interactions between humans and virtual agents.

Existing automatic synthesis solutions often leverage the richer information offered by the audio recording of speech, which more easily conveys intent, emotion, and excitement. However, these solutions rely on speeches pre-recorded by human or machine actors. This is costly in both time and storage. Ignoring this data, it is possible to generate coherent gestures directly from text transcripts, which significantly reduces the overhead of the production of such animation and makes it easier to supply a model with novel data to synthesise from.

Such a method would seek to eventually replace animators with writers, who would supply the model with a script of a given conversation and allow the model to generate a gesture for the given transcript.
Chapter 2 - Background and Related Work

2.1 - Sequence Transduction

Sequence transduction is the problem of solving any task which transforms an input sequence into a corresponding output sequence. More narrowly, transducers can be defined as a model which outputs a single datum per input timestep. This includes such common problems as speech recognition [12], text-to-speech translation [13], and text-to-gesture translation [14].

In the case of this project, it is the task of translating annotated transcriptions of natural language speech into human gesture data.

Sequence transduction solutions require the solving model to retain pertinent information across time steps. A common example lies within language translation:

“The vehicle pulled into the car-repair shop. The car later pulled back out onto the road, good as new.”

In this example, the word “car” refers to the vehicle introduced in the first sentence. A human intuitively knows that the two terms refer to the same object, but it may be challenging for a neural network to retain such meaning and connection between terms across time steps. This is particularly important where the considered data of interest is temporally related to other pieces of data, as is the case in Natural Language Processing (NLP) and text-to-gesture transduction.

The sequence transduction, or sequence-to-sequence model is typically composed of an encoder-decoder architecture. The encoder processes the input sequence into a context vector of fixed length. This vector represents a meaningful summary of the input sequence. A decoder is presented with the context vector and the history, processing it to emit the transformed output.

![Figure 1: Sequence Transduction](image)
2.2 - Recurrent Neural Networks

Regular Feedforward neural networks do not feature a mechanism for temporal context-specific information. Recurrent Neural Networks feature loops in order for information to persist throughout, as explored in [15] and [16]. At each time step in an RNN, information is passed to the next time step. One may view an RNN as a chain of identical copies of the same feedforward network, each passing its output to the next network as input. This allows information from previous layers to affect decision-making in the current layer. This is relevant to our problem of synthesizing gestures, as the data involved is sequential and temporally related.

![Figure 2: An unrolled Recurrent Neural Network](image)

However, vanilla RNN architecture becomes less effective as relevant information becomes increasingly distant in layers from the current layer, as information loss becomes likelier the further it travels along the chain. This is known as the vanishing gradient problem, where a term exponentially trends towards zero, making it difficult to learn long period dependencies [17]. The opposite problem is known as the exploding gradient problem, where a term trends towards infinity and becomes unstable [18].

\[
\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 < 1 \quad \left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 > 1
\]

![Figure 3: Equations for Vanishing and Exploding Gradient](image)
2.3 - Long Short-Term Memory Networks (LSTM)

Long Short-Term Memory Networks (LSTM) are a type of Recurrent Neural Network which use gates to regulate the flow of information throughout the layers [19]. Regular RNNs suffer from vanishing and exploding gradient problems. In order to solve the vanishing gradient problem, LSTMs introduce gates in order to control the gradient flow more precisely.

A standard RNN cell is composed of input from the previous step and current input. It uses an activation function $\tanh$ in this example, though other activation functions are possible [20].

Figure 4: Standard RNN Cell Architecture

Here $X_t$ and $h_t$ refer to the input and hidden state respectively at time step $t$. Comparatively, LSTM cell architecture is more complex:

Figure 5: LSTM Cell Architecture

The additional forget, input/update, and output gates allow for more precise control over the current cell state. The forget gate outputs a value between 0 and 1, a remembrance vector where 0 is to completely discard and 1 is to completely keep. The input/update gate decides what should be stored in the cell state, or whether the cell state should be updated with the current activation value. The output gate decides which parts of the cell are output to the hidden state.
Unfortunately, as LSTMs are still recursive in nature, they do not lend themselves to parallel computing, though studies show that certain aspects of the model may be parallelized to improve performance [21] [22]. Parallel computing is desirable in order to reduce training time dramatically by making efficient use of the Graphics Processing Unit (GPU), so therefore LSTMs are less suitable than a Transformer model, particularly when trained on consumer-grade hardware, as was the case in this project.

2.4 - Gated Recurrent Units (GRU)

The flow of information within a GRU is very similar to that of an RNN. The difference between the 2 architectures lies within the individual work units [23]. The RNN cell, as pictured in Figure 4, is very simple in design, featuring only a single activation function between the input and output and hidden state of the cell.

The design of a Gated Recurrent Unit is more complex. It features 2 logic gates - a reset gate and an update gate to control the flow of information in the cell.

![Figure 6: Gated Recurrent Unit](image)

The Update gate controls whether or not the cell state will be updated with the candidate current activation value state. The higher the value of this gate, the more weight is placed on the current activation value. The Reset gate is optional in GRUs but serves the purpose of deciding the importance of the previous cell’s output value. This allows the model to discard irrelevant information that will not be useful further along the chain.

GRUs feature fewer tensor operations than LSTMs, which can lead to a slightly faster training process. However, they suffer from the same sequential order of operation, and therefore do not lend themselves to parallelization.
2.5 - Convolutional Neural Networks

Convolutional Neural Networks also attempt to solve the problems of vanishing and exploding gradient [24].

In convolutional models, the distances between positions are logarithmic rather than linear, as it is in Recurrent Neural Networks, which minimizes the vanishing gradient problem. The distance between the output and any input for a CNN is in the order of log(N), whereas in RNNs this distance is on the order of N.

![Convolutional Neural Network Structure](image)

*Figure 7: Convolutional Neural Network Structure*

Convolutional Neural Networks also solve the issue of parallelization, as each sample can be processed simultaneously and independently of previous samples. However, Convolutional Neural Networks do not inherently find the solution to dependencies between samples, which make them less suitable to the task than the Transformer architecture used in this project. They are better suited to tasks such as image recognition, which do not feature sequential data.

Attempts have been made to utilize CNN architecture for transduction tasks such as Natural Language Translation in [25] [26] [27], but the standard in sequence-to-sequence tasks remains RNN – and Transformer-based models for the reasons described above.
Chapter 3 - Transformer Neural Networks

3.1 - Introduction

Recurrent Neural Networks, along with LSTM and GRU neural networks have been established as state-of-the-art approaches to sequence modelling and transduction problems. Transformers are a novel neural network architecture which aims to solve sequence transduction while handling long-range dependencies and allowing for inherent parallelization. It was published in the paper “Attention is all you Need” [28] and has gained in popularity since its proposal in 2017.

3.2 - Parallelization

As described in 2.2, Recurrent Neural Networks typically operate on an input sequence in order, generating a sequence of hidden states for each input token at position \( t \) in the sequence by factoring in the output from the previous position \( t-1 \). This practice of sequential operation precludes the possibility for inherent parallelization, which is a cornerstone of efficient training. This becomes particularly problematic with long sequence lengths. Work has been performed to improve the performance of RNN architecture [22] [21], but the constraint of sequential operation on individual input tokens remains.

Figure 8: Parallel Nature of Transformer Encoder Stack
Conversely to RNN approaches, the transformer model does away with sequential, recurrent behaviour, instead focusing entirely on attention mechanisms to produce the required dependencies between input tokens. This allows for significantly greater parallelization than RNN approaches and is therefore better suited to training on consumer-grade GPU hardware, as was used in this project.
3.3 - Architecture

The transformer model still follows the general encoder-decoder architecture but makes use of stacked self-attention and pointwise, fully connected layers for both the encoder and the decoder.

![Encoder-Decoder Structure of a Transformer Network](image)

The encoder is made up of a set of $N$ identical layers, each of which is composed of 2 sub-layers. Inputs into the encoder first flow through a self-attention layer before being fed into a fully connected feed-forward neural network. The self-attention layer allows the model to analyse other input tokens in order to gain a better understanding of the token currently being processed. Using an NLP problem as an example, in the sentence “The chicken crossed the road because it needed to reach the other side.”, it is important to the meaning of the sentence to understand which term the word “it” is referring to. In the self-attention layer, as the word “it” is being processed, the encoder will look for meaningful links to other words in the sentence to answer that question. Succinctly, self-attention relates different tokens from an input sequence to one another to gain a better understanding of the entire sequence and the meaningful links between individual input tokens.

![Transformer Encoder Architecture](image)

Each standard decoder layer contains the same self-attention and feed-forward layers as the encoder and adds a multi-head attention layer between them.
3.4 - Attention

The concept of Attention in the field of Neural Networks refers to a model’s ability to selectively concentrate on a few relevant points of data out of many. Attention in deep learning is a vector of importance weights used to predict or infer one element by estimating, using the attention vector, how strongly this element is correlated with, or “attends to”, other elements. One may then use the sum of these element values weighted by the attention vector to approximate the target element.

The attention model was developed as an alternative to the LSTM and GRU solutions in attempting to solve the vanishing gradient problem. The performance of the encoder-decoder networks degrade rapidly as the length of the input sequence increases. In traditional RNN-based approaches, only the final state of the encoder is considered in the context vector. The attention model solves this problem by giving the context vector access to the entire sequence of hidden states, which solves the issue of information degradation across long chains.

Calculating simple self-attention is a multi-step process. For each input token, we create 3 matrices from the embedding of the token – the Query, Key, and Value matrices. These are calculated by packing the embedding of each token into a matrix and multiplying it by 3 weight matrices calculated during training.

Next, a score is calculated for each input token by taking the dot product of the query and key matrices. These score values are used to compare the input tokens to one another and determine how much attention to pay to other tokens in the input sequence as we encode a certain token. The scores are then divided by the square root of the key matrix dimension and passed through a softmax operation to normalize the scores. The scores are now positive and sum to 1. This score represents how much each word is expressed at this position. A high score signifies that a token is related to the token at the current position. The final step is to multiply the Value matrix by its respective softmax scores to produce the output of the self-attention layer at a certain position. Therefore, the entire calculation may be represented as so:

\[ \text{Att}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{k} \right) V, \]

*Figure 11: Attention Calculation*
One can expand the self-attention mechanism by way of “multi-head” attention [29]. One performs the same self-attention calculation as above, but several times with several different weight matrices, resulting in multiple outputs. These matrices are then concatenated and multiplied by an additional weight matrix to produce the final output of the multi-headed self-attention layer.

3.5 - Positional Encoding

It is vital in any sequence-to-sequence neural network architecture to account for the order of tokens in the input sequence. Transformers address this by way of positional encoding [30], adding a vector to each input token’s embedding vector. The vectors follow a certain pattern which the model learns as it is trained, effectively tracking each token’s position in the input sequence by tracking its distance relative to other tokens.

**Figure 12: Enhancing Embeddings with Positional Encoding**

In the original paper [28], the positional encoding functions used are sine and cosine functions of different frequencies, though many kinds could be used:

\[ PE_{pos,2i} = \sin(pos/10000^{2i/d_{max}}) \]
\[ PE_{pos,2i+1} = \cos(pos/10000^{2i/d_{max}}) \]

This sinusoidal method carries the benefit of being capable of extrapolating to sequence lengths of unseen proportions, i.e., input sequences longer than the longest training input sequence.
3.6 - Layer Normalization

Between each sub-layer of the encoder and decoder in a transformer, there is an additional layer-normalization step where the input of the sublayer is added to the output of the sublayer and has the layer-normalization function [31] applied to it, which has been shown to improve the pace of training of neural networks. The approach is based on batch normalization and improves upon it by removing the dependency on batches and making it easier to apply to RNN models. Layer Normalization normalizes each feature of the activations to zero mean and unit variance.

In layer normalization, one first calculates the mean and the variance of each sample.

\[
\mu_i = \frac{1}{K} \sum_{k=1}^{K} x_{i,k} \\
\sigma_i^2 = \frac{1}{K} \sum_{k=1}^{K} (x_{i,k} - \mu_i)^2
\]

Following this, one normalizes each sample such that the elements within have a mean and unit variance of 0.

\[
\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}
\]

The final step is to scale and shift:

\[
y_i = \gamma \hat{x}_i + \beta \equiv \text{LN}_{\gamma,\beta}(x_i)
\]

The layer normalization operation does not involve any of the other samples within the batch, operating on each sample individually. Normalization techniques such as this stabilize the learning process, which can dramatically reduce the number of epochs required to train a deep neural network.
3.7 - Decoder Stack

The output of the top encoder of the stack is transformed into a set of attention vectors K and V. These vectors are used by each decoder in an encoder-decoder attention layer. After the encoding phase finishes, the decoding phase begins to output elements.

The output of each time step will be fed into the next time step’s bottom decoder, until an End of Sequence (EOS) symbol is encountered, marking the end of a sequence of tokens. The opposite symbol denotes the Start of Sequence (SOS). Similar to the encoder stack, the decoder inputs are positionally encoded to denote each input token’s relative position in the sequence.

The self-attention layers in the decoder stack differ to those in the encoder stack by only computing the attention of tokens previous to the current token in the sequence. Others will be ‘masked’ with -inf before the softmax operation described in section 3.4.
3.8 - Final Linear Layer

The decoder stack outputs a vector of floats which needs to be transformed into usable output. This is performed by the final Linear Layer. This is a fully connected neural network which maps the decoder output into a vector of logits. A logits vector contains a cell for each possible output option, each containing the score of an individual token. A softmax operation is applied to this logits vector to normalize the values above 0, summing to 1. The option with the highest probability is chosen as the output token.

In order to compute the loss function for this final output, one compares the probability distribution function of the desired, correct output with that returned by the final layer. This loss function will ideally be minimized over time.
Chapter 4 - State of the Art

This section will review some of the literature in the field of gesture synthesis from secondary input data. It will also include a review of study into speech to text and video to motion translation, which is important in the field of gesture synthesis to provide sample data by which to train our models.

Human motion synthesis as a field of study has gained interest in recent years in a wide range of different intended applications.

An example of using RNN architecture for motion synthesis can be found in [32]. In this paper, the authors explore human motion synthesis using modified RNN architecture dubbed auto-conditioned RNN (acRNN). The acRNN is able to synthesize arbitrary complex motion of up to 18,000 continuous frames by explicitly accommodating for autoregressive noise accumulation during training. The acRNN makes use of LSTM architecture to store historical sequence information as described in section 2.3.

Described in [33], Normalising Flows are another deep-learning architecture which may be used to produce high-fidelity animation. This approach provides a generalisable approach to gesture generation and has the capability of generating a wide variety of different motion types, including both human and non-human locomotion. It is also controllable – generated gestures can be conditioned on an arbitrary control input, such as direction of travel. Additionally, the results in probabilistic output, which allows one to generate multiple plausible output samples from the same input sequence, as the model seeks to describe all possible motions given certain input.

The method is built upon normalising flows, which are a method for constructing complex distributions by transforming a probability density through a series of invertible mappings. Repeatedly applying the rule for change of variables, the initial density ‘flows’ through the sequence of invertible mappings. At the end of the process a valid probability distribution is output, and hence this type of flow is referred to as a normalising flow. This concept was introduced by Rezende et al. in “Variational Inference with Normalising Flows”.

Previous work [34] explores flow-based generative models using an invertible 1x1 convolution. The Glow implementation was adapted by Henter et al. to better solve the task of gesture generation in [33].
This approach yields promising results when presented with aligned motion capture and audio data. However, they are not suited to our desired approach of text-aligned gesture generation, unless the text data were transformed into natural human speech, which has been explored in [35], where a model is trained to synthetically replicate an actor’s voice, allowing one to generate audio samples of arbitrary natural speech. This approach, however, requires a high volume of sample data of the actor’s voice, which significantly increases overhead.

Text-based motion synthesis methods have been explored in [14], [36]. In [36], the authors explore the use of a trimodal context of text, audio, and speaker identity in tandem. [14] explores the use of a transformer model along with tertiary emotive data to synthesise motion.
Chapter 5 - Data

One of the most important aspects of any machine learning project is the input data used. High-quality data sources and proper data processing are half the battle, and there are a number of factors that go into decisions regarding sample data format, length, etc. This section will explore the nature of the input data. It will touch upon the format of the data, how the data was collected and sourced, as well as how it was processed for the purposes of this project.

5.1 - BVH File Format

The motion input data is represented by the BioVision Hierarchy (BVH) data format [37]. The format was developed by the motion capture services company BioVision to provide motion capture data. It provides skeletal hierarchy data as well as motion data in order to fully describe the motion captured.

![Figure 14: Skeletal Structure of a BVH File](38)
This file format is simple and human readable. The only drawback of BVH files is that the root pose is described only in each bone’s translational offset from its parent – no rotational data is provided, though it may be inferred from the translational data present.

The BVH file is made up of two main parts, a header section which contains a description of the skeletal hierarchy and a data body which contains the frame-by-frame motion data.

The header section begins with a keyword – “HIERARCHY” which marks the beginning of the skeleton’s hierarchy definition. It is followed by a definition of the skeleton “ROOT” joint, which will be parent to all other joints in the skeleton. In the case of this project, the “Hips” are defined as the “ROOT” joint.

What follows is the rest of the hierarchy, with each following joint being described by its translational offset to its parent. The “CHANNELS” describe the format of the motion data given every frame, in our case the X, Y, and Z rotation of each joint. The child of a joint is optionally indented to aid human readability, and each joint is surrounded by curly braces. Each joint may have many children, but only 1 parent. In this way each joint’s positional information is stored recursively by observing the information of its parent.

![Figure 15: Hierarchical Skeleton Definition of a BVH file](image)
The “MOTION” section first defines the total number of frames of motion data stored in the file, as well as the frame time, which is used to play the file back at the correct speed. Our input data is given with a frame time of 0.0166667, which translates to 60fps (frame per second) playback. The rest of the “MOTION” section defines the motion data each frame.

The floating-point values given represent the Euler angles for each joint. The first three values denote the root joint’s position, and each following grouping of three floats represents the relative rotation of that joint to its parent, concatenated in the order they are defined in the hierarchy above.

Figure 16: Motion Data Section of a BVH File
5.2 - Original Input Data

The data used in this project was provided courtesy of Trinity College Dublin. The Trinity Speech-Gesture data repository is comprised of a number of different collections of data, the one of which being used is the GENEVA Challenge 2020 data release.

The data captures the performance of a single actor speaking and gesturing naturally. The actor’s performance was motion captured and has been provided in the aforementioned BVH format. The BVH files contain motion information on 56 bones each. The actor’s performance is split into 33 segments, 23 for training and 10 for testing purposes.

The training and testing data is made up of 3 different types – Audio, Motion, and Transcript data of the same performance, each split into the aforementioned 33 segments. The audio captures the actor’s spoken performance, which is given in .wav file format. The motion data is given in BVH file format.

Figure 17: Example Frame of BVH Input Data, Captured in Blender
The audio is transcribed to JSON format using Google Cloud automatic speech recognition (ASR). Each word is given a start and an end time in seconds. Names are replaced by tokens in the transcription to preserve the actor’s anonymity.

```
{
    "start_time": "0.700s",
    "end_time": "1.400s",
    "word": "warm"
}
```

*Figure 18: Sample JSON Element Representation of a word transcribed by ASR*

The audio and motion data were manually aligned in order to correct dropped frames. Although some corresponding audio and motion files are of differing lengths, they are aligned correctly and will remain aligned until the end of the shorter file.

For this project the audio data was not considered in order to reduce the production overhead incurred by recording such performances in the future, as above.
Chapter 6 - Implementation

This section will provide an overview of the implementation of the project. It will begin with an explanation of the data pre-processing that was performed on our source motion and text data, followed by an analysis of the network and training of the model. Finally, we will also examine the performance of the training process and the resulting model. Much of this transformer implementation is based on the transformer model described in [14], adapted for an entirely new data set.

6.1 - Data Pre-Processing

The aforementioned input data must first be pre-processed in order to be suitable for this project. As it is given, the data is only split into a total of 33 segments. Without further splitting the data, each sample will be too large for our purposes. The transformer-based architecture of the project demands smaller samples in order to be run on the current consumer-grade GPU hardware used throughout this project. Such large samples resulted in memory overflows.

Additionally, it is desirable to train a model using samples similar in length to the desired output of the application in question. In our application of embodied virtual agents, the use case will likely often demand gestures generated for small, sentence-wise segments of text only.

Multiple sample sizes were explored, ranging from one to ten sentences. The most consistent gestures were achieved with a one-sentence sample size, maintaining the most consistently low mean loss value across epochs, and producing the most realistic gestures.

As shown above, the text data was split into individual word elements in a JSON file. Each word is denoted with a start and an end time in seconds, and the words may end in a full-stop if the current sentence has come to an end.

As the motion capture BVH files deal in frames rather than seconds, the time data from the transcriptions had to be translated into frames using the frame time given in the sample BVH files. As this frame time was 0.16667, or 60 frames per second, each word’s time value was simply multiplied by the inverse of the frame time $1/0.16667$. 


Having converted the word timing data to frames, words were now concatenated into a sentence structure, which similarly carries a start and an end frame. Iterating over the sentences, the motion capture data was then split using the same frame data.

The transcriptions were converted from JSON to simple text format for easier human readability.

This resulted in transforming 33 input motion capture BVH and transcription JSON files into 2004 individual sentence segments.
6.2 - Motion Data Extraction

The motion data, having been split into sentence-wise segments, is read in using a standard BVH loading function which constructs and returns an animation based on the contained motion capture data.

The rotations of joints relative to their parent joint in the skeleton are stored as quaternions in order to circumvent the potential issue of gimbal lock. Specifically, the angles are stored as vectors of unit quaternions.

The start and end of each animation is capped with a predefined custom pose. This pose serves as the SOS and EOS tokens described in the section 3. As the motion data captures a single actor standing, the pose simply shows the skeleton in a neutral standing position, arms at its sides.

The source data was also down sampled by a factor of 2 in order to reduce training time. As the original data was presented in 60 frames per second, the down sample results in 30 frames per second, which is more than enough for most applications. Down sampling is such a way reduced the training time from ~7 minutes per epoch to ~5 minutes due to the reduction in frame count.

6.3 - Transcription Data Extraction

The goal of this project is to produce a sequence of corresponding motion gestures given an input of natural language text sentences. The gestures will be represented as sequences of joint rotations relative to their parent joint on the skeleton.

As is standard practice in such sequence-to-sequence problems, the input sequence sentences will be represented as word embeddings. The transformation is performed using the tried and tested Global Vector (GloVe) model [39], pre-trained on the common crawl corpus. The common crawl corpus is one of the largest publicly accessible web datasets [40]. GloVe was used as it performs similarly to models of higher-dimensionality, while out-performing most similarly dimensional models. The start and end of each input token (sentence) is marked with special Start of Sequence (SoS) and End of Sequence (EoS) tokens, predefined by the GloVe model. Each token also has its corresponding VAD values extracted using the NRC-VAD Lexicon.
6.4 - Training

Having formatted our data as described above, the problem of gesture synthesis from the transcribed text of an actor’s performance can be viewed as a sequence transduction problem, which we will solve using a transformer architecture based neural network.

Our model roughly follows the design of a standard transformer as described section 3, but we will revisit the concept here in explaining its application to this project.

As stated, the transcribed sentence data is transformed using GloVe into word embeddings. To these word embeddings we add positional encoding to represent each token’s relative position to other tokens in the sequence. Finally, we append secondary positional data pertaining to the position of the entire sentence in the larger samples of the actor’s performance initially recorded. This combination of transformed word embeddings and secondary positional data are taken as input by our fully connected layer in order to be converted into latent representations, to be passed to the decoder stack.
Within the encoder unit, information first passes through a Self-Attention layer, which carries information on the importance of other words in the sentence when considering the current word. After this self-attention layer, information flows into a Multi-Headed Attention Layer which allows the model to detect different sets of connections between words in the same sentence. Otherwise, only a single, averaged set of connections would be considered for each word, whereas there may be different types of connection between different words in the input sequence, such as semantic and grammatical similarities. The model makes use of 4 heads in each Multi-Headed Attention layer used throughout. After the Multi-Headed Attention Layer, the information flows into 2 fully connected layers. These layers map to 250-dimensional outputs. This process is repeated N=4 times, once per encoder unit.

Our tertiary data (start frame, end frame) is appended to the resultant latent representations and the combination data is processed through 2 fully connected layers into encoded features.

The decoder stack takes in the encoders’ output along with past gesture history in order to predict future output. It operates very similarly to the encoder stack, with several additions, including a Masked Multi-Headed Self Attention Layer. As described above, the masked layer will consider only samples previous in the sequence to the current sample. This is done by masking the future positions before the softmax step in the self-attention calculation. The output of this masked layer is then passed through an additional unmasked Self-Attention Layer before flowing through a final Multi-Headed Self Attention Layer and into two fully connected layers to complete the decoder unit. This entire process is repeated N=4 times, once per decoder unit.

A dropout factor of 0.2 is applied across both the encoder and decoder units in order to reduce and prevent overfitting as described in [41].
6.5 - Model Evaluation

The model’s effectiveness was evaluated on several different factors. The first and simplest metric used is the square difference of the converted Euler angle rotation of each output joint. Euler angles are used rather than quaternions as the Euclidean distances between Euler angles are simpler to calculate.

In addition to this, we also calculate the squared norm difference between the two values at all time steps, to ensure that the general movement of the skeleton across all joints is also respected, as the joint loss function alone does not account for the skeleton entire.

Finally, to improve the expressiveness of the synthesized gestures, we calculate a loss function from the affective features of the skeleton. Affective features computed from human gestures and poses have been shown to denote emotion and intent [42] [43] [44]. One can isolate emotions such as fear, surprise, anger, sadness, happiness, and disgust based entirely on the positional data of a human skeleton in pose.

Coming back to the VAD model, emotions with high dominance values are often expressed with an expanded upper body and shoulders, wide arms, and an upward posture, whereas emotions such as sadness are conversely associated with rounded shoulders, a more collapsed upper body, with limbs held close.

For each pair of ground truth and synthesized output, we compare certain angles and areas between joints representing specific motions and poses in humans. As the lower body is less relevant to the application of embodied virtual agents, only affective features of the torso, head, and arms are considered. By examining the hand, elbow, shoulder, neck, head, and hip joints, we can compute affective features by observing their relative positions to one another.

For each ground truth and output sample, we calculate the respective affective features and determine the loss by calculating the absolute squared difference between each set of values.

Combining these 3 types of loss – Joint Angle, Pose, and Affective Feature – we produce our total training loss function.
6.6 - Training Parameters

The model is trained on the Adam optimizer, as described in [34]. The initial learning rate applied is 0.001, and we apply a learning rate decay of 0.999 every epoch. The stochastic batch size used throughout is 8. The data is split 80% - 10% - 10% between training, test, and evaluation data respectively. The training was performed on an Nvidia GTX 1070, with an average epoch length of ~5 minutes. The model tends to reach respectable mean loss values of ~0.02 very quickly, then oscillating around this value indefinitely. Further iterations improve the smoothness of generated gestures and reduce jitter.
Chapter 7 - Results

Through the process detailed in the sections above, the Transformer model can effectively synthesize human gesture samples using training data consisting of the motion-capture data of a single actor’s performance, along with the time-aligned transcription of their speech. The generation of these new poses and gestures is possible in interactive frame-times on a consumer-grade Graphical Processing Unit such as the Nvidia GTX 1070, and so may be suited to interactive applications such as video games.

The synthesized BVH files were found to be working with Blender 2.80. In addition to this, results were visualized using the GENE A 2020 BVH Visualizer [45], which strips the output of its lower extremities, which were not the focus of the project.

![Figure 23: Output of GENE BVH Visualizer](image)

The synthesized gestures appear to suffer similar issues to those described in other papers in this field. Over time, the model regresses towards the mean pose [46] [47]. The generated gestures lack some of the emotive quality and expressive gesticulation of real human motion.

This is in part a limitation of this transformer-based approach. To generate very high-quality models will well-defined gestures, one needs a significantly long history. In the case of this project, the history refers to the number of gesture frames which make up each input sample in our model. An alternative approach to solving the issue would involve restructuring the Transformer architecture to better suit learning long-term predictions from short histories. Both options are quite computationally expensive, prohibitively so on the consumer-grade hardware currently available. Attempts have been made to improve long-term predictions from short histories in RNNs, such as in [36], though a similar approach has yet to be applied to Transformer architecture to the best of our knowledge.
Chapter 8 - Conclusion

We present a transformer-based approach to the synthesis of human motion data based on input data consisting of transcribed human speech and temporally aligned motion capture data. The produced gestures can be generated at interactive speeds, with frame times as low as 3ms. Synthesized gestures returned an accurate average pose of the data presented to the model.

8.1 - Limitations and Future Work

The synthesized motion produced by the model is imperfect. Ideally, the gestures produced would be as emotive and expressive as the input data presented to the model. It is a limitation of this approach to fail at producing high-fidelity, expressive gestures when trained under the limitations of consumer-grade hardware.

However, this approach still shows promise considering this limitation, and would benefit from more powerful hardware. It would allow one to use a longer, richer history to produce more varied output gestures, rather than regressing towards a mean pose.

Further study must be conducted into the effects of hyperparameter tuning and architectural design changes of the Transformer model applied to this project. This work was limited by the availability of powerful CPU and GPU hardware, and would benefit from being explored further without these restrictions.

Additionally, other datasets should be used for training to determine the generalizability of this approach.

Further study is being conducted into the use of RNN and GRU architecture for the purposes of motion synthesis in [36] and [48], so it may be of benefit to apply this dataset to other sequence to sequence models in order to contrast the performance of each approach.
Chapter 9 - Bibliography


[27] M. Elbayad, L. Besacier and J. Verbeek, “Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction”.

[29] J.-B. Cordonnier, A. Loukas and M. Jaggi, “Multi-Head Attention:”.


[37] M. Meredith and S. Maddock, “Motion Capture File Formats Explained”.


[40] V. Kolias, I. Anagnostopoulos and E. Kayafas, “Exploratory Analysis of a Terabyte Scale Web Corpus”.


