Generating Dance Moves

Cary Huang
carykh@stanford.edu

Cindy Jiang
cindyj@stanford.org

Connie Xiao
coxiao@stanford.org

Abstract

In this project, we try to construct a program that generates original dance videos using artificial intelligence. Our training data are low-resolution (96x54) images of grayscale silhouettes of a single dancer found on YouTube [2]. To get our model to successfully generate new video in the same dance style as this data, we modified code of a convolutional auto-encoder in TensorFlow [3] and a recurrent neural network in Torch [7] written by other people online. Our results were convincing enough to look like real humans, but the dancing itself was humorously repetitive. Increasing our dataset and refining the neural network architecture might improve the outputted dance further.

1. Introduction

In the ever-increasingly STEM-filled era of 2017, it has become harder and harder to find real human dancers (there are only about 50 dancing groups at Stanford.) If someone desperately needs a video of low-quality backup dancers, is it possible to produce such a video with artificial intelligence?

At the core of this project, we were interested in exploring the creative capabilities of artificial intelligence. We are also interested in dance. So, trying to find a way to generate dance moves was a natural extension of our interests.

Previous works have explored dance generation using human skeletal figures, where each dance move can be represented by a vector of joint angles. However, our project approaches dance generation with a new angle; our goal is to generate a sequence of images that combines into a dance video. The training data are images of silhouettes of a single dancer from a YouTube video [2]; Figure 1 displays screenshots of the video. These images are converted to grayscale and compressed into 128-dimensional vectors using an auto-encoder and fed into a neural network. The newly generated vectors are decoded back into images, which will be the resulting dance sequence.

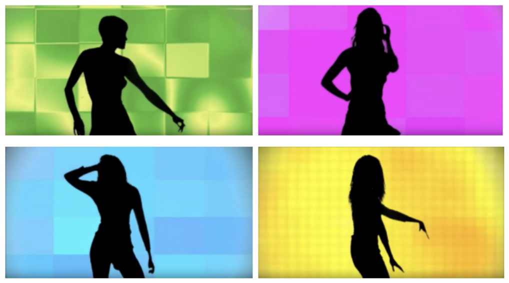


Figure 1. Dance video used as training data and dataset

2. Literature Review

There are some related works regarding the use of models to generate dance moves or activity. Holden *et al.* [6] uses a feedforward neural network on top of an auto-encoder to generate realistic-looking movements. Unlike our training set of images, Holden *et al* trained on an extensive dataset of motions. This project only generates a limited set of movements, such as walking, running, punching, and kicking. Actions are represented by a collection of joint angles on a 3D human skeletal structure.

The Lulu Art group, in collaboration with Peltarion, has developed a system called Chor-rnn [5] which uses deep recurrent neural networks to generate choreography. Similar to Holden *et al.*'s motion generator, the dances are represented with feature landmarks on a human skeletal structure, though Chor-rnn generates a 2D structure rather than 3D. The dances generated by Chor-rrn were evaluated by a choreographer, who also learned and performed the choreography to evaluate its feasibility.

Another project that generates dance movement created by Alemi *et al.* is GrooveNet, which creates real-time dance movements given an audio stream. GrooveNet extracts audio features from the music and feeds it into a Boltzmann machine (FCRBM), which generates dance moves for a 3D skeletal structure with each move represented by a root position and joint rotations. The paper mentioned that their initial experiments using an LSTM-RNN were not successful, probably because FCRBM works better on smaller training sets and is easier to train on real-valued, continuous data than LSTM.

Most of the related works address the goal of generating activity or dance, but there are no works tackling the exact same problem as our project. Rather than generating motion for a human skeletal structure, our project focuses on generating a sequence of images of a human silhouette dancing. To accomplish this, we tried to adapt some of the approaches used in related works like using an auto-encoder and a neural network.

3. Methods

In order to generate dance moves, we first represent moves in the video as a series of images. Then we transform our dataset of images into a sequence of dense vectors with an auto-encoder. So, a 30 minute video will no longer be thought of as a video, but rather a sequence of eighteen thousand 128-dimensional vectors. Next, we train a recurrent neural network (RNN) on that sequence of vectors and generate new vectors. We decode the generated vectors and string them together to get the final generated dance video.

3.1. Auto-encoder

The auto-encoder compresses each grayscale 96x54 pixel frame to a 128-dimensional vector using four levels. Each level consists of convolution and pooling/upsampling. Convolution identifies the significant features of the image. Pooling compresses 2x2 squares of pixels into 1 pixel.

Symmetrically, in order to retrieve the image from the dense vector representation, decoding starts with a fully-connected layer that goes from 128 neurons to 4x6x32 neurons. After that, there are four levels to decoding; each level is an iteration of convolution and upsampling. Upsampling does the reverse of pooling and makes 1 pixel to a 2x2 square of pixels. The output of the final convolution is a single grayscale 96x54 pixel image, which should match the original input as closely as possible.

What is the point of computing, convoluting, and redistributing? Suppose the encoder and decoder work perfectly; that is, when the encoder compresses an image into a vector and the decoder expands that vector back into the image, the input and output images are exactly the same. In this case, we have two ”filetypes” of a dance image. We could either choose to store a dance image as a simple image file (.png), which will take 5,184 quadrabytes[1], or we could store it as a 128-dimensional numpy vector (.npy), which will take 128 quadrabytes. Although this saves space, we will have to run the vector through our decoder to get back an image our human eyes can understand.

Before we can encode and decode images, we need to train the auto-encoder. In the beginning, both the encoder and decoder are set up with random synapse weights, so they have garbage values. We then feed in true images through both the encoder and decoder, one at a time, hoping the output closely matches the input. If the two images do not match, we use stochastic gradient descent to nudge the synapse weights ever so slightly in the direction to make the input and output match more closely together. The batch size was two hundred, which means we randomly selected two hundred images out of our set of eighteen thousand images to train the network on. Also, the learning rate was 0.001 for the first ten thousand iterations and 0.0001 for the next one hundred forty thousand iterations.

3.2. Failed Method: Feedforward Neural Network

After implementing the auto-encoder, we had a decent way of converting between 96x54 pixel images and 128-dimensional vectors. However, we still did not have a neural network that could actually generate new dance poses in sequence: i.e., the ”meat” of our dance generation algorithm. Our first idea was a simple feedforward neural network (FNN); this acted as our baseline. This FNN would consist of three layers: input, hidden, and output, all with 128 neurons each. First, we wanted to train the FNN using a set of arbitrarily chosen frames represented as vectors from the original video. These frames were converted into vector-form by the auto-encoder. Ideally, the output layer should approximate the vector corresponding to the next frame of the video. The actual output would be compared to the ideal output in order to train the FNN. The difference between the two would be our loss function, which we tried to minimize with gradient descent.

Once we had sufficiently trained the FNN, we gave it a starting image from the video (a person in any kind of pose) and converted it into a vector with our encoder. Inputting this vector into the FNN would generate a successive vector (the next frame of the generated dance video) as output. We fed this output vector back as input into the FNN to generate the third vector. We kept repeating the process, using the third vector to generate the fourth vector, the fourth vector to generate the fifth vector, and so on. Once we had our sequence of generated vectors, we could decode them to images. That sequence of images would give us a video of generated dance moves. To make a more accurate FNN, we actually fed the previous two frames as input (256 neurons). This would give information about the velocities of body parts, which may make the generated dance more fluid and smooth.

Unfortunately, this neural architecture failed miserably, which is described in more detail in the Evaluation section.

3.3. Recurrent Neural Network

For our second attempt, we implemented a recurrent neural network based off of Andrej Karpathy’s famous char-rnn online article [7], which replicates styles of text in a simi-

[1]Quadrabyte = 4 bytes = the size of a floating point variable (not a real word, but we used it to make the explanation clearer)

lar way to how we wanted to replicate dance moves. To be specific, this type of RNN is a Long-short term memory RNN (LSTM), which means it has ”forget gates” that allow it to forget irrelevant information and remember relevant information for longer. In the same way you can imagine the auto-encoder as the ”file-converter” between the 96x54 pixel images and the dense 128-dimensional vectors, this recurrent neural network is the ”imitator” of the 128-dimensional vectors, generating new sequences of dance vectors in the same style as the training data.

How does this work? To start, we converted the eighteen thousand dancing-pose-images from our training data into a sequence of eighteen thousand 128-dimensional vectors using the auto-encoder. Then, the RNN moved through these vectors one timestep at a time, trying to predict what the next 128-dimensional vector would be (i.e., the next pose). By setting our cost function as the mean-squared-error between the RNN's prediction and the actual next pose, we could train our RNN to accurately predict the next pose by minimizing this cost. Soon, the RNN could output realistic poses following any sequence of input frames.

3.4. Modifying the RNN to suit our needs

We had to make major modifications to Karpathy's char-rnn code to make it work for our project. For one, the original LSTM inputted and outputted discrete data: text characters. Since the neurons in LSTMs are typically continuously defined, the way Karpathy converted continuous output into discrete data was to use one-hot vectors. A one-hot vector has as many dimensions as there are types of output: for example, if there are 64 types of characters that appear in a body of text, the corresponding vectors for each character would be 64-dimensional. Then, a one-hot vector simply assigns a 1 to the dimension of that character's type, and a 0 to all the others. So, encoding ”D”, the fourth letter of the alphabet, might look something like (0, 0, 0, 1, 0, 0, 0, ... 0, 0, 0).

Unfortunately, our dataset does not have discrete elements: rather, each dance pose exists in a continuous, 128-dimensional space. There is no way to encode that information in single characters of text. So, we replaced every part of code that initially required a one-hot vector with code that could interpret the 128-dimensional vectors read in from our .npy file of dancing poses. This also required a change of the original loss function from softmax cross-entropy with logits, which is a probability distribution, to mean-squared-error. It makes sense for the original algorithm to use cross-entropy for a probability distribution of what text character will come next. However, you cannot really assign probabilities to what the next dance pose will be, when there are technically infinitely many possible dance poses, since it's continuous. So, when in doubt, it is easiest to go with the most basic type of loss: mean-squared-error.

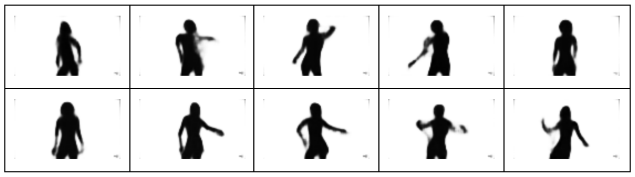


Figure 2. Dance moves produced by the RNN

3.5. Dance Generation

This process of training our RNN makes it great for prediction. But how do we use the RNN to generate a completely new dance? We start by feeding it a random 128-dimensional vector (which corresponds to a dance pose from the dataset). The RNN will predict the second frame. The trick is to now feed that second frame back into the RNN as if it is a concrete piece of training data. The RNN will now look at those first two frames, and predict a third frame. The process repeats, with the third frame being fed back as input, the RNN producing a fourth frame, and so on. After this is all said and done, we now have an arbitrarily long sequence of these 128-dimensional vectors. (In our final generation test, we generated 600 frames of a dance, but we could just as easily create 60,000 frames if we had the patience.)

However, to make a ”dance sequence” watchable by humans, we need to convert each of these 128-dimensional vectors back into 96x54 pixel images. Fortunately, the ”decoder” portion of the auto-encoder's purpose is to reconstruct a 96x54 pixel image from a 128-dimensional vector. So, we feed the 600 vectors through this decoder, one-by-one, and we get out a sequence of images. Now, it is simply a matter of exporting those images as a video file using video-editing-software like Sony Vegas 12, and we've got our desired dance video!

3.6. Training Algorithm

The algorithm used to train both the auto-encoder and the neural networks is the Adam optimizer, an extension of stochastic gradient descent. The Adam optimization algorithm adds momentum to the standard SGD to speed up the learning rate and dampen oscillations.

4. Results

The final video product of our auto-encoder and RNN, along with our failed attempts, can be accessed at https://www.youtube.com/watch?v=cfRoj7HUnqo. Figure 2 is every fourth image of our generated dance.

5. Evaluation

5.1. Auto-encoder

To evaluate the success of the auto-encoder, each initial image is encoded into a vector and then decoded back into a 96x54 pixel image, which should be the same as the initial image. We put the original image on top, and the attempted reconstruction from our auto-encoder on the bottom, as shown in Figure 4. Ideally, we would see two of the exact same image. As more iterations of gradient descent occur, the auto-encoder gets more accurate.

If this process can work successfully, that means weve captured all the necessary information to replicate a 5,184 pixel image in just a 128-dimensional space, with each dimension hopefully capturing a higher-level trait of the image, such as ”How raised are the arms?” or ”Is the body to the left or right of the image?”

There is a chance that our auto-encoder is memorizing the images in a nonsensical way. Even if the original image matches up to the image that comes out of our auto-encoder, there is a possibility that small changes in the vector representation do not decode to a similar image and that our auto-encoder is not depicting features we want. To check if this is the case, we can look at vector-image pairs and see if similar images have similar vector representations. If they are similar, then we know our auto-encoder is not just memorizing pixels. In Figure 3, we can see that similar images capturing the gradual arm movements indeed have similar vector representations. From these observations, we can conclude that our auto-encoder is functioning correctly.

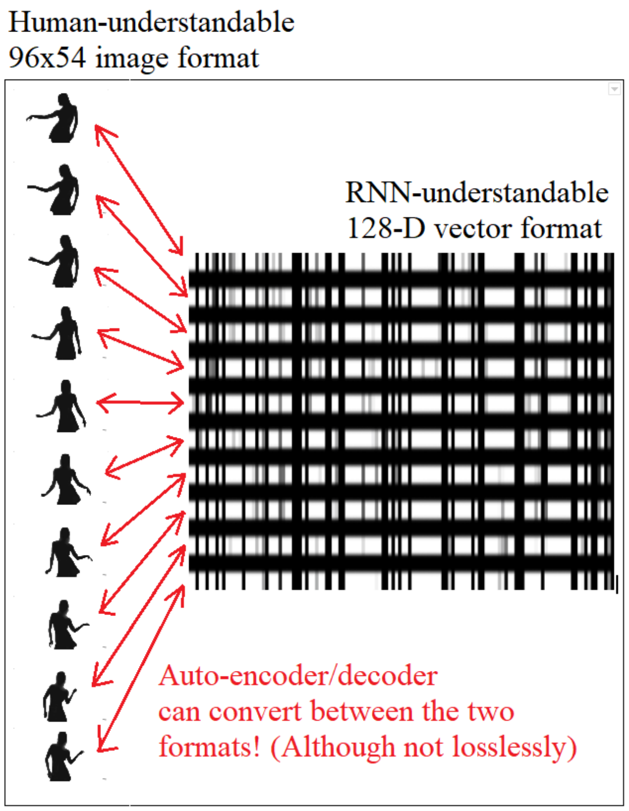


Figure 3. Images and vectors

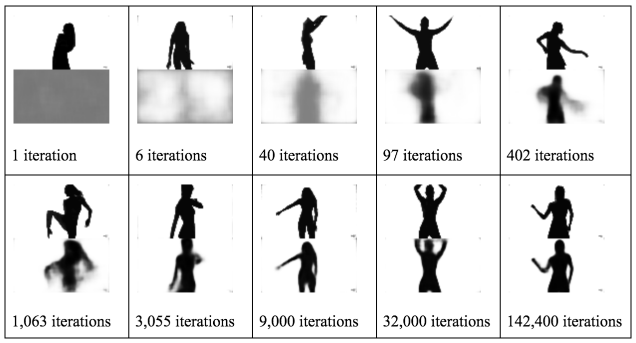


Figure 4. Autoencoder training results

5.2. Feedforward Neural Network

As stated before, our FNN failed to produce any image of a believable dance pose. The main challenge was that, despite being fed specifically different inputs, the FNN kept generating the average of all the training data. You would expect a working FNN to produce a drastically different dance pose for differing inputs. For example, if you just jumped upward, its pretty likely youll fall down afterward and not float in the air. If you just swayed right, you should probably sway left next. This is all to say, an FNNs output should clearly change depending on the type of input.

So, the fact that our FNN outputted the same, middle-of-the-road pose regardless of the input, means it failed. Essentially, it produced a fuzzy still image that looks like all dance images blurred together, as you can see in Figure 5.

This led us to wonder, why would an FNN do this? Consider that we used mean-squared-error as our loss function. So, whenever the FNN outputs some value, we assess how well it did by taking the difference between its output for index 1, and the true value of index 1, and square it. Then we do the same for index 2, and 3, and 4, and so on. Then we add all those squares together, and thats our loss! If the FNN wants a loss near 0, it has to replicate the true values closely.

However, suppose that for whatever reason the FNN does not have enough information to guess the next frame closely. If it has to take a shot in the dark, guessing the midpoint is actually the best strategy. If the range of possible values is $[0, 1]$, then if you guess 0, your loss could be anywhere from $(0 - 0)^2 = 0$ to $(0 - 1)^2 = 1$. To be honest, 1 is a pretty nasty loss. If you guess 1, same story: your loss could be high as 1. However, if you guess 0.5, youll never be more than 0.5 away from the true value, and 0.5^2

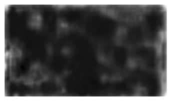
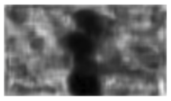
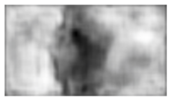
Frame 1	Frame 2	Frame 3
		
Frame 4	Frame 5-100 (essentially)	
		

Figure 5. The FNNs not-so lively ”dance video”, getting stuck on a blurry average.

is only 0.25. This loss is so low, it actually means guessing the midpoint (a.k.a. average) is the FNNs best bet for a low loss. So, it is no surprise that our FNN in this case did the same thing, and outputs the average of all possible dance poses.

It turns out that changing the loss function from mean-squared-error to cross-entropy did not fix this bug, even though we thought cross-entropy would be more likely to pick riskier poses away from the average. In the end, we were forced to make a more drastic change: upgrading from a feedforward neural network, to a recurrent one.

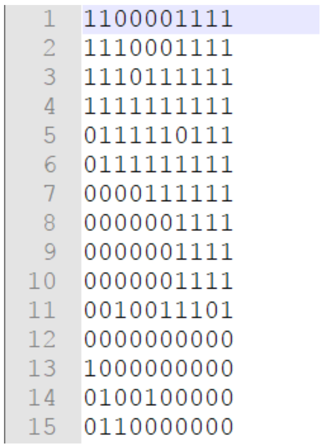
5.3. Recurrent Neural Network

The first way to evaluate our RNN's performance is simply looking at whether the loss function is going down. Low loss usually indicates high quality output, although we learned that is not always the case.

We first used mean-squared-error (MSE). This means, we looked at each element of the 128-dimensional output of the RNN, compared it with the corresponding element of the actual training data, and took the difference. Then we squared all those differences and added them together. We were a little apprehensive of MSE as a loss function because it favors picking the midpoint to lower worst-possible-loss, which is similar to the problem of the aforementioned FNN.

However, we were pleasantly surprised by the RNN's performance with MSE. As a baseline test, we first created a test training set of 10-dimensional vectors that simply trace out a 2D map of perlin noise over time (Figure 6). The RNN was almost instantly able to replicate this style of output, with the loss dropping 100-fold in just a few minutes. To get this RNN to work on the dancing data, it seemed we'd only need to expand our output dimensionality from 10 to 128. Indeed, that shift appeared to work, indicative by how the loss function went down.

However, we soon learned the hard way that even though



```
1100001111
1110001111
1110111111
1111111111
0111110111
0111111111
0000111111
0000001111
0000001111
0000001111
0010011101
0000000000
1000000000
0100100000
0110000000
```

Figure 6. The baseline training data we tested our RNN on. The generated output looks essentially the same.

the loss value was technically at a minimum, we would need a new way to evaluate output quality. Loss function gave us a numerical measure, but the only true way to tell if this RNN was generating good dances was to judge the dances with our own human eyes. This would require using the auto-encoder's decoder to convert the 128-dimensional vectors back into images before we could judge them. It turned out that our supposedly successful RNN settled on one pose and stayed frozen there forever, as you can see in Figure 7.

We next tried decreasing the batch size from 50 to 5 and increasing the number of neurons per layer from 400 to 600. Somehow, the combination of these changes allowed the RNN to no longer be frozen on a single pose, and it produced a believable dance. Since increasing the batch size does not really change the math happening behind the scenes, we believe this improvement happened as a result of increasing the number of neurons. With enough neurons, the network can finally have enough information to make an atypical pose more confidently.

At this point, the RNN was producing results we could safely call a ”dance”. It was ”successful” in the sense that the silhouette looked like a real human and seemed to obey the physics of a real human. (Limbs moved at the right pace, with the right amount of momentum to sway believably.) However, it was not ”successful” in the sense that the dances were not so enjoyable to watch. The dancing primarily consisted of shimmying back and forth, with very few extraordinary dance moves. This lack of memorable actions is perhaps due to the training data itself having mediocre dancing, or due to the auto-encoder blurring everything. Either way, this minor problem can hopefully be solved with more data or more neurons.

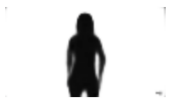
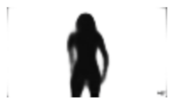
Frame 1	Frame 3	Frame 10
Frame 30	Frame 31-100 (essentially)	
		

Figure 7. The LSTMs tendency to get frozen

6. Conclusion

This project demonstrates that AI-based "video generation" is possible, despite a few limitations. These limitations include relatively short temporal memory (dance moves barely last a second), low-resolution, low-quality images being required to fit in the dense representation, potential overfitting by the RNN's architecture being significantly larger than the training set, the lack of color (although this can easily be fixed by adding multiple input channels into the auto-encoder), and more.

Although it would be quite a leap to claim this project proves AI can possess creativity, it at least shows that AI can learn nuanced details about how the human body moves. It also demonstrates that video, which is one of the most information-dense forms of media, can still be understood by AI if you're willing to compress it enough. Perhaps, with enough intelligent compression, artificial intelligence will be able to perform even the most mind-bogglingly information-dense tasks humans can imagine, with ease.

6.1. Future Work

Two easy ways to improve this project further is getting better training data by filming actually skilled dancers dancing, and using all three color-channels so that the output is in full RGB color.

In terms of make the generated dance more interesting, we could try generating dance moves at a higher level and look at combinations of moves, rather than images. This would pose the additional challenge of finding a way to represent dance moves discretely. We could also further investigate different types of RNNs to see if one can create riskier dance moves.

7. Code and Supplements

The code for this project can be found at https://gitlab.com/conniexiao/221-dance-generator.

Again, a video of our results can be found at https://www.youtube.com/watch?v=cfRoj7HUnqo.

References

[1] O. Alemi, J. Franoise, and P. Pasquier. GrooveNet: Real-Time Music-Driven Dance Movement Generation using Artificial Neural Networks. 2017.

[2] H. Arslan. Techno/Tech House Mix by DJ Haluk Arslan with Shadow Dancers. https://youtu.be/NdSqAAT28v0, 2015. 1

[3] M. Chablani. Autoencoders Introduction and Implementation in TF. 2016. 1

[4] F. Chollet. Building Autoencoders in Keras. 2016.

[5] L. Crnkovic-Friis and L. Crnkovic-Friis. Generative choreography using deep learning. In *ICCC*, 2016. 1

[6] D. Holden, J. Saito, and T. Komura. A deep learning framework for character motion synthesis and editing. 35:1–11, 07 2016. 1

[7] A. Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks. 2015. 1, 2