DREU Report 2019

London Lowmanstone IV

7/29/2019

Introduction

This is my final report for my research during DREU in 2019. I worked on two projects during this ten-week period. The first project was creating an artificial intelligence algorithm for generating covers of songs in a MIDI format. The second project was working with John Harwell on a mathematical representation of decisions made by robots in multi-thousand robot swarms. Since the second project is still in progress and results will likely be released in a future paper, I will not be including discussion of that project in this report.

Implementing Time-Contrastive Networks in Music

Background

In 2017, Google researchers released a paper on a new artificial intelligence model called a time-contrastive network (TCN) [1]. This network took in pairs of videos of people pouring liquid into cups and learned an encoding of video frames that allowed it to determine where in the motion of pouring a particular frame was. After the encoding was determined, it could be used to train a robot to pour into a cup by giving the robot a video feed of its own hand movement, and giving it rewards for having the encodings of its own movement mimic the encodings of a video of human pouring (using reinforcement learning).

My overarching goal is to use the model to complete an analogous task in music. I would like the network to be able to take in different versions of the same song, and learn an encoding that allows the network to determine which part of the song is playing. Then, hopefully, this encoding would allow another

component to generate MIDI and gain rewards for creating MIDI files that sounded like the original song. This would allow for quick music transcription with many different instrumentations. For example, given multiple examples of someone humming along to orchestral songs, the model may be able to generate a new orchestral song after hearing someone hum a new tune.

Along the path to this end goal of pouring liquid or generating music, there is an intermediate step used to verify that the TCN worked. In the case of pouring liquid, the intermediate step was to give the computer multiple videos of people pouring liquid into cups that were not synced in time, and ask the computer to sync all of the videos to match the timing of one of the videos. If it did this well, this would show that the encodings did a good job of determining where frames were in the motion of pouring. The analogous task in music would be to be able to take a song and different versions of that song, and align all of the songs to play at the same time. This intermediate task would at least show that, given a suitable reinforcement learning algorithm, it might be possible to create music based off of the encodings created by the network.

Goal

For the purposes of this ten-week period, my goal was to get as close as I could to completing the intermediate task of aligning different versions of songs to the original song.

Methods

There were two potential methods for accomplishing this goal. The first method was a top-down method. I would first use the code released by the Google team and test to ensure that I could achieve their results. Then, I would inspect the code and determine how to modify it to work for music data. This would require changing the input format and data pipeline built into the code, as well as modifying the neural network to an architecture suitable for audio data rather than image and video data. I would then need to find or create a dataset to train and test on, run the network, and then fine-tune the architecture until I obtained reasonable results.

The second method was a bottom-up method. I would look at the paper, understand the formulas, and implement it myself, finding or creating my own audio dataset and modifying the architectures as needed.

The first method had the drawback of potentially consuming a lot of time without making any progress. Understanding programmers' code takes a while, especially when it's work done at the cutting edge of a

field by a team of researchers who have been working in the environment for a long time. However, it carried the security of a baseline - if I got the original network up and running and could replicate their results, I would know that my network was fundamentally correct, and could make modifications from there. In addition, any progress I did make using the original code would likely quickly lead to my goal, since the general pipeline for doing so would be immediately available. The second approach was much less certain in that if I implemented something incorrectly, I would not know what I had done wrong or why it wasn't working unless I was able to compare differences to a working version or was able to find the bug myself. This could be extremely time-consuming as well depending on how difficult the bug was to track down. However, assuming that I did implement the paper correctly, building up the network from scratch would likely allow me to make *some* progress towards a music-creating network more quickly. Overall, the first method might take a long time to understand, but would likely lead to rapid jumps in progress. The second method would likely move more quickly, but only lead to small amounts of progress.

Top-down Method

At first, I attempted to work on the top-down method. I was successfully able to get the baseline code running and passing the given tests on resources available through the Minnesota Supercomputing Institute [2].

However, when I began to inspect the code, I quickly realized that the assumption of image data was baked fairly deep into their code. It didn't look like they separated their loss function from their network, and their network also baked in regularization that I wasn't certain I wanted to use. Thus, it didn't seem like I was going to be able to understand how to modify their code and continue working on my research with John Harwell within the remaining six-week period. So, I switched to the bottom-up method.

Bottom-up Method

My work using the bottom-up method began with understanding the formulas and architecture behind the TCN in more detail. During my research, I discovered the authors were using a triplet loss function, which was originally developed in a paper on face recognition [3]:

$$\mathcal{L}(A, P, N) = \max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0)$$

,

where A is the vector representing the anchor image, P is the vector representing the positive image, N is the vector representing the negative image, α is the margin, and $f(I)$ is the output of the network with input I. Essentially, this loss incentivizes minimizing the difference between $f(A)$ and $f(P)$ and maximizing the difference between $f(A)$ and $f(N)$.

Since the loss function was originally used for images, I realized that the best way to ensure that I was using the loss correctly was to test it on images. Even though the original paper used faces, I decided I would first test the loss on the MNIST dataset [4], since it was the dataset I was the most familiar with.

Triplet Loss Network

Code

The code used to generate the results of the triplet loss network can be found at https://github.com/London-Lowmanstone/triplet-loss-network.

Environment

In order to obtain the following results, I used a mid-2010 MacBook Pro running TensorFlow v1.5 with only a CPU. The training took slightly under half an hour. 100,000 triplets (with replacement) were randomly selected for training from the Keras MNIST training set, and 10,000 triplets (with replacement) were randomly selected for testing from the Keras MNIST test set. I used an α margin of 0.6 for the loss function, a learning rate of 0.0001 with the Adam optimizer, and trained for 40 epochs with a batch size of 32.

The internal network has an input layer with 784 neurons, an inner layer with 200 neurons, and a final output layer of 100 neurons. The final output layer uses a sigmoid activation function. (The previous two layers merely use the identity function.)

Results

During training, the accuracy on the test set reached over 93%. (Accuracy is defined as the percentage of triplets in the test set that achieved a loss of 0 with an α margin of 0.)

Figures 1, 2, and 3 detail one triplet in the training set. The first figure displays the images in the triplet from the MNIST dataset, and the next two images demonstrate the encodings of the images before

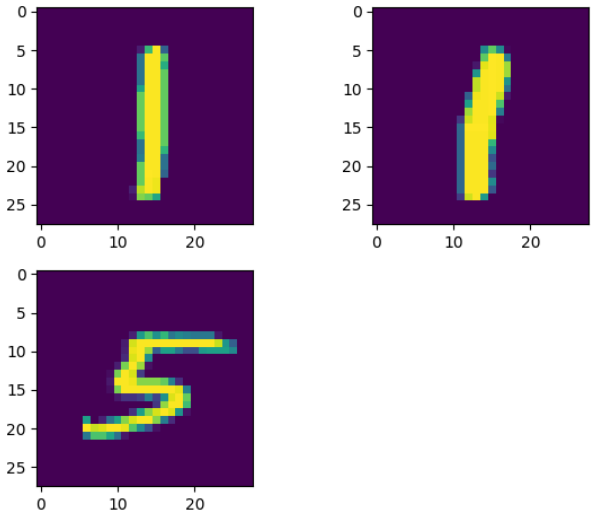

Figure 1: Images from the MNIST dataset: anchor image (top left), positive example (top right), and negative example (bottom left).

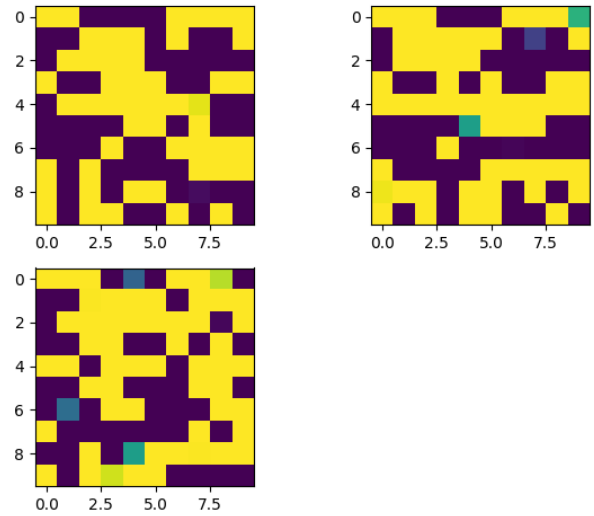

Figure 2: Encodings of the MNIST images before training: anchor encoding (top left), positive example encoding (top right), and negative example encoding (bottom left).

and after training. Note how in figure 3 the images of the anchor and positive example look exactly alike, despite the differences in the original MNIST images in figure 1. This, along with how different the encodings of the negative example are from the encoding of the anchor, demonstrate how well the network trained.

From figure 4, we can see that the network trained quite well, smoothly decreasing loss and increasing accuracy over time. Also note the rapid convergence for the test set accuracy which occurs within less than 15 epochs.

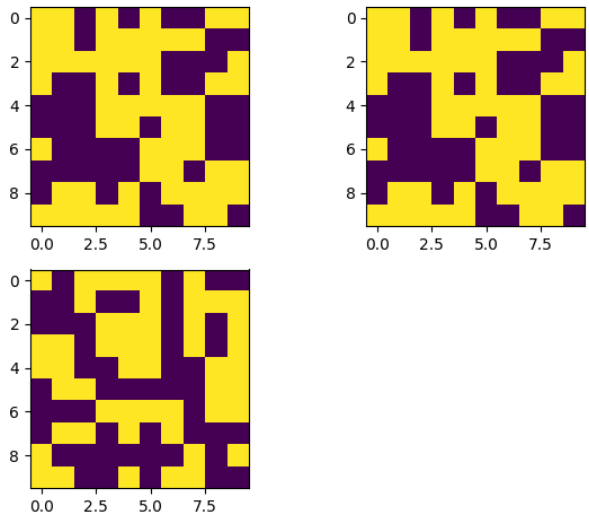

Figure 3: Encodings of the MNIST images after training: anchor encoding (top left), positive example encoding (top right), and negative example encoding (bottom left).

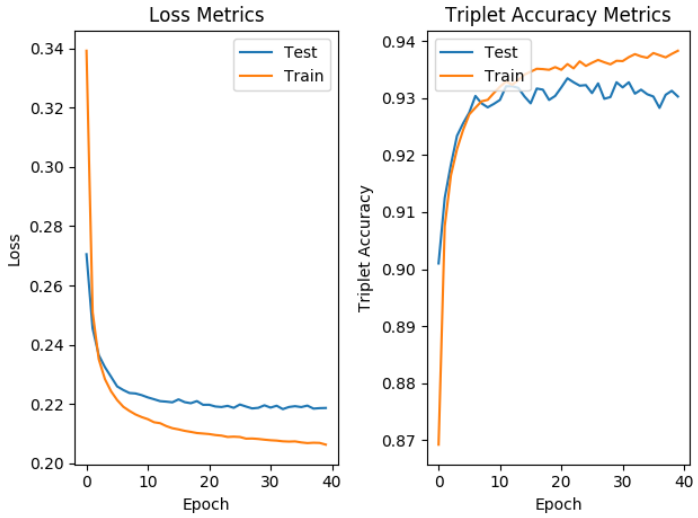

Figure 4: Train and test metrics for both loss and accuracy throughout training.

Future Research

The next goal is to develop a loss function that can be used to train a time-contrastive network such that it learns encodings of songs that represent the same portion of that song.

One way forward is to use the original loss function and create a dataset where the different versions of the original song have the exact same timing as the original song, but with different instrumentation. This would allow training to work exactly as the training for pouring liquid into cups, but will require specific data that may be difficult to obtain, since different versions of a song usually don't retain the exact same

timing as the original.

Another potential option is to create a new loss function that still uses time as a signal for training, but does not assume that time is a perfect signal. That is, it won't assume that a note 1/3 of the way through the original song is the same note as a note 1/3 of the way through the cover song. Note that the current TCN model does make this assumption, and this assumption is primarily what drives training. Thus, creating such a loss function will weaken the time signal, but will allow training with songs and covers of those songs without any special data generator.

Acknowledgements

I would like to thank Professor Maria Gini for being willing to mentor me this year and for providing me with helpful guidance throughout the program, as well as many additional experiences and opportunities. I would also like to thank the DREU program for organizing and funding this work. I am very grateful I was able to participate in the DREU program this year.

References

[1] P. Sermanet, C. Lynch, J. Hsu, and S. Levine, "Time-contrastive networks: Self-supervised learning from multi-view observation," *CoRR*, vol. abs/1704.06888, 2017.

[2] "Minnesota Computing Institute." https://www.msi.umn.edu/. The authors acknowledge the Minnesota Supercomputing Institute (MSI) at the University of Minnesota for providing resources that contributed to the research reported within this paper. URL: http://www.msi.umn.edu.

[3] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," *CoRR*, vol. abs/1503.03832, 2015.

[4] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.