DREU Report 2018

London Lowmanstone IV

7/23/2018

Introduction

This is my final report for my research during DREU in 2018. During this ten week period, I worked on two main projects. The first is Sierra, a framework for automating large-scale simulated robotics experiments on a supercomputer. The second project attempted to use a capsule network as an autoencoder to determine if capsule networks might help computers to understand the world more similarly to humans. This report covers the work I did on both projects.

Sierra

Background

One of the graduate students working in the lab, John Harwell, studies swarm robotics. To run experiments on different algorithms or environments, Mr. Harwell uses a simulation environment called ARGoS [1], which allows for the simulation of extremely large swarms of robots. However, the process of setting up multiple different random simulation environments, running all the simulations, and collecting all the data takes a lot of time and work. So, Sierra was created to streamline that process. In practice, it's been used to run simulations containing at least 500 robots in multiple different foraging scenarios, and was utilized in a paper, on which I am listed as second author, which was submitted to the International Conference on Robotics and Automation.

Goal

The overall goal of Sierra was to create a command-line program that would be able to run entire experiments from a single command. It would generate all of the configuration files to create the simulations, run the simulations in parallel on a supercomputer, place the output data into organized folders, and then generate labeled graphs from the data.

While this was the overall goal of Sierra, my smaller goal for the summer was to write a program that did the following:

- Edit XML Files
  - Search for elements by their ID attribute
  - Change attributes
  - Change tags
  - Delete elements
- Average CSV files
- Run multiple ARGoS simulations in parallel on a supercomputer

Using this simpler framework, Mr. Harwell could then expand it into the full pipeline by adding onto the main core modules I created.

Main Modules

xml_helper

The `xml_helper` module contains the class `XML_Helper`, which takes a path to an XML file as input and creates an object representing that XML file. The `XML_Helper` class contains functions for doing the following operations:

- Write the XML data to a file
- Change the value of an attribute in the XML
- Change the name of a tag in the XML

- Remove an element from the XML

All of the attributes and tags and elements are specified by paths that trace through the XML object to find their target. Behind the scenes, the class relied on the ElementTree XML API which comes built-in with Python.

csv_class

This module contains the class for representing CSVs. I overloaded Python's magic methods in order to allow the `CSV` objects to be summed via the "`+`" operator, and divided by constants using the "`/`" operator. This made it simple to create a function that averaged all the CSV objects, since they could be averaged like any other set of numbers.

One major check the `CSV` class also performs ensures that each CSV that is being averaged has the same dimensions (rows and columns) as the other CSVs. An exception is raised whenever a CSV that has an inconsistent amount of columns in each row is created or CSV objects without the same dimensions are summed. (This became useful later on for discovering bugs or determining which simulations were not working properly because Mr. Harwell could immediately see that the CSVs were invalid.)

experiment_runner

This is the main heart of Sierra. It is a command line program which uses Python's `argparse` module. It takes in a configuration file describing a simulation environment, copies the configuration file, removes the visualization elements (since the supercomputer does not have visualization capabilities), and changes the random seed on each new copied configuration file. This creates a set of environments with different random seeds, which can then be run on a supercomputer for faster results.

Since I am doing my work at the University of Minnesota, we are using a cluster from the Minnesota Supercomputing Institute (MSI) [2] to run ARGoS. As part of creating Sierra, I created PBS batch scripts which defined how to run the project in parallel on the supercomputers at MSI. Each PBS script sets up the environment and then gives control to the experiment runner to run the experiment and collect the data.

Internally, the experiment runner uses GNU Parallel [3] to run the ARGoS simulations in parallel on the supercomputer. MSI provides a nice framework for using GNU Parallel, and by using the `subprocess` module in Python, we could run each simulation in parallel, making use of MSI's computing resources.

Background

World Models Architecture

As part of my research on artificial intelligence, I became interested in learning how to program computers to play video games. As part of my research on that topic, I came across the paper "World Models" [4], which describes a very high-level architecture for programming a computer to play a video game, assuming that the available data is a video screen capture combined with the user's input to the game during gameplay.

The World Models Architecture consists of three main components: a Variational Autoencoder (VAE), a Mixture Density Network that is a Recurrent Neural Network (MDN-RNN), and a Controller. The paper gives different names to each of the components, but I think they are best understood as a Modeler, a Simulator, and a Controller.

Modeler Component

The Modeler component (consisting of a VAE) learns how to encode the game into a smaller vector. The video screen captures of gameplay consist of large amounts of data since each individual pixel is represented in the data of each frame of the game, and each video can consist of hundreds or thousands of frames. By creating a representation of the game that consists of less data, the computer can run computations orders of magnitude more quickly. In the World Models paper, each frame of the game is resized to 64x64 pixels, [4, Appendix A] which is 4096 dimensions. To lower this dimensionality, the VAE trains to take in an image and output a smaller vector. This vector is its representation of the game for all of its future computations and training, so it must learn to create a representation of the game that keeps data necessary to understanding the game and discards data that is not needed.

Simulator Component

The Simulator component creates a virtual simulator of the game based on given images and user inputs during training. This part of the model is an MDN-RNN.

The RNN portion, implemented as a long short-term memory (LSTM) unit, takes an encoded vector of the current game frame, concatenated with the current input/action from the user at that time step,

concatenated with its own internal state for computation. The RNN then outputs two vectors: one with parameters that define a probability distribution as to what the next encoded game vector could look like one time step into the future, and another vector which will be its new internal state one time step into the future. (The internal state of the RNN is also called the "hidden" state. It is unclear from the paper what is used as the initial internal state. From looking at a paper the authors referenced [5], it may be guessed that it is the hyperbolic tangent function applied to the initial encoded vector of the first frame of the game.)

The MDN portion takes the RNN's vector of parameters which define the probability distribution of the next (encoded) frame in the game, and a single number $0 < \tau < 1$ called the "temperature". The MDN then returns a single encoded vector representing the next frame in the simulation; the higher the temperature, the more likely the MDN is to sample from the edges of the distribution, giving more wild results.

In this way, we have a simulator for the game. The RNN will give the probability distribution of the next future frame, and the MDN will select a future frame. The user (or controller) can then pick an action, and the process can be redone to generate the next frame. Thus, we can generate a sequence of frames that are similar to the game: a simulator.

Controller Component

The Controller component plays the game. This is done with a genetic algorithm, starting with completely random neural networks playing, and then keeping the best players and combining them together to create a new generation of better players (while adding back in some more random players to help with exploration of different strategies). All of the training for these controllers is done within the Simulator component using a high temperature. This allows for very quick playthroughs of games and rapid learning from the AI, without requiring an API for the original game. The high temperature helps the controllers to not specialize too much to one particular method of playing the game; by introducing random changes to the game, the controllers must learn to adapt to unexpected changes very quickly, which helps their performance when applied to the real game, since the real game is quite different from the simulator.

The final output of this algorithm is the best controller from the Controller component, which can then be applied to the original game and have its performance evaluated accordingly.

Motivation

The Modeler component performs a similar function to humans describing situations while playing a video game. While the screen holds all of the available data of what's occurring in the game, we might describe the video game by saying "The road suddenly takes a sharp left here" which is a smaller encoding that still is able to accurately describe the state of the game at that point.

If we consider the encoding to be an analog for how the computer "understands" what is occurring in the game, it would make sense if each dimension of the encoded vector corresponded to a particular property or object in the game. However, that is not how the computer understands the game; the dimensions created by the variational autoencoder don't appear to correlate to any human understanding of the game. This can be tested by varying the encoding to explore the latent space and testing to see if certain dimensions seem to change the image in a meaningful fashion. When doing so on the online demo of the world models architecture, none of the dimensions seem to transform the image in any meaningful way.

This is a worrying issue, since there is no meaningful way for the computer to describe its actions if there is no correlation between its understanding of the game and how humans understand the game.

Capsule networks provide a potential solution to this problem. Unlike variational autoencoders, the output of a capsule network is a single output capsule which represents an object and its orientation and placement, also known as its "instantiation parameters." In a multi-layered capsule network, each new layer of capsules represents a higher class of objects comprised of smaller components represented by the earlier layers. If the variational autoencoder in the World Models architecture were replaced with a capsule network, the output of the network could be a high-level capsule representing the entire game, with capsules in previous layers and their orientation representing objects within the game and their orientation.

Thus, if we used the final capsule layer as the encoding, each component of that encoding would hopefully map to a more human-understandable notion, describing what objects exist on the screen and where.

Method

In order to test if capsule networks could be used as encoders, I modified an implementation of capsule networks in Keras [6]. Initially, the capsule network was intended to be used on the MNIST dataset. However, since I wanted to test the encoding of a more simple dataset that was more game-like, I developed

a very basic game where the player dodges balls falling from the top of the screen. The game has the capability to record gameplay, thus providing the screen data and input data needed for the autoencoder and later stages of the World Models algorithm.

However, after modifying the capsule network to function as an autoencoder, I encountered an issue. The capsule network output almost the exact same output image for each input image. When trying to debug this issue, I switched to using a feedforward neural network encoder and found that this exact same issue occurred when the encoding dimension was too small, making it so that the computer had to compress too much information. What occurs in both situations is that the capsule network outputs an image that is very close to the *average* of all of input images. Thus, the network's error between the output image and the input image is relatively low for all of the input images because each output image looks somewhat like each input image. The issue is that if all of the output images look the same, the rest of the World Models algorithm will not be able to function, because the simulator will essentially only be one static image, from which the controller component cannot learn how to play the game. This defeats the purpose of the autoencoder.

Possible explanations

After researching this issue, I have come across three possible explanations.

Option 1: Capsule Networks Require Labeled Data

The first possible explanation is that capsule networks work better when using labeled datasets. When used for the MNIST dataset, each capsule in the last layer of the network represents a number. The magnitude of the output vector from those capsules represents how likely it was that the input image was that number, and the direction of the vector represents how to draw that particular number so that it matches the input image. This is a large difference from the architecture I used, which has a single final output capsule to encode any input image; in the MNIST implementation, there are multiple final capsules, each one only encoding a single number.

However, this architecture is only possible when there is labeled data, such as the MNIST dataset. During training, the capsule network was only trained on the correct responses. For example, if the capsule network was given an image of a 3, and the output of the capsule representing the number 8 had the greatest magnitude (the network predicted that the image was an 8), then instead of training the

capsule representing the number 8 to encode the input image, the system used the fact that the original image was a 3 and trained the capsule representing the number 3 to encode the input image instead.

With video games or screen captures, there is no labeled data, so I could not use this technique and instead tried to train a single final capsule to encode every input image. It may be that labeled data is required in order for capsule networks to work well as encoders.

Option 2: Encoding Vector Too Small or Not Enough Training

I discovered that when feedforward networks exhibited the same behavior of always outputting an average of all the input images, the issue could be solved by increasing the encoding dimension and by training for longer periods of time [7].

It may be that I did not train the capsule network for long enough or that or that my encoding dimension was too small. However, I did try training for longer periods of time and raising the encoding dimension, neither of which had a noticeable impact on the results.

Option 3: Incorrect Programming

It also may be that I programmed the network incorrectly by not removing portions of the program that were exclusively for use with the MNIST dataset or by modifying code incorrectly in some other way.

Future Research

A further investigation into why the capsule network failed to create unique outputs for unique inputs would be helpful in determining whether capsule networks can be used on unlabeled data.

Even if capsule networks can be used as autoencoders in this fashion, more work needs to be done in order to allow computer programs to explain to humans why they are functioning in a particular manner. This is also known as the "semantic gap problem." Any general solution to this issue will be necessary if we want humans to be able to quickly understand why a program is acting in a particular way and correct fundamental misunderstandings the program holds.

Acknowledgements

I would like to thank Professor Maria Gini for mentoring me throughout this process and providing me with so many wonderful opportunities and experiences. I would also like to thank Mr. Harwell for allowing me to work on his project and for helping teaching me many useful programming ideas and practices. Finally, I would like to thank the DREU program for organizing and funding this work.

References

[1] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

[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] O. Tange, *GNU Parallel 2018*. Ole Tange, Mar. 2018.

[4] D. Ha and J. Schmidhuber, "World models," *CoRR*, vol. abs/1803.10122, 2018.

[5] D. Ha and D. Eck, "A neural representation of sketch drawings," *CoRR*, vol. abs/1704.03477, 2017.

[6] XifengGuo, "Capsnet-keras." `https://github.com/XifengGuo/CapsNet-Keras`, 2018.

[7] P. Q. (https://stackoverflow.com/users/5049813), "How to get an autoencoder to work on a small image dataset." Stack Overflow Stack Exchange. URL: https://stackoverflow.com/q/51253611 (version: 2018-07-13).