It Ain't Much but it's ONNX Work[1]

Matthew Ewer
[N/A]

Giles Edkins
[Independent researcher]

Joar Skalse, Lauro Langosco, Esben Kran, Fazl Barez

Abstract

We present a tool, ONNXplorer for visualizing the network topology of arbitrary Deep Neural Networks up to a given level of complexity. The tool makes use of Virtual Reality (VR) technology via the Unity gaming engine, and allows compatibility with different machine learning frameworks through the ONNX intermediate representation.

The tool shows the topology of the networks together with the neuron activations for a particular input.

We tested the tool with a real-world deep learning image classifier, MobileNet, as well as on toy models.

Keywords: Visualizations, Virtual Reality, ML safety, ONNX, Unity

1. Introduction

How can we visualize what's going on in a Deep Neural Network? With so many layers and so many connections between each neuron in each layer, things can get pretty confusing. We turn to Virtual Reality (VR), and some techniques for efficiently processing data and culling the less interesting parts of the structure.

We chose to standardize on the ONNX intermediate format for models, allowing support for models generated with different frameworks (e.g. PyTorch and Tensorflow, which are both compatible with ONNX). In addition, the ONNX runtime contains C# bindings which allowed compatibility with the Unity gaming/VR framework.

2. Methods

Replicating the results
GitHub: https://github.com/onnxplorer/ONNXplorer

[1] Research conducted at the Apart Research Alignment Jam #8 (Safety Verification), 2023 (see https://alignmentjam.com/jam/verification)

Follow the README (remember to run download_dependencies and download_models before starting Unity).

Methods

We used Unity 2022.1.14f1 as our development framework, and C# as our programming language.

The workflow was:

- Download a trained model in ONNX format
- Create a "modified" model, also in ONNX format, based on the original.
    - It has extra outputs corresponding to each of the original hidden layers
    - We also extract some useful model metadata at this time
    - We decide a "layer" for each tensor, based on the max of the previous layers plus 1.
- Run the modified model in inference mode, with a particular input tensor. For example, in the case of an image classification model, the input would be a (cropped and normalized) photograph.
- Extract the "outputs" of this model, which correspond to the original model's output plus the hidden layer activations, in the form of tensors.
- Feed this to a layout algorithm, which produces coordinate arrays corresponding to position and color, and separate coordinate arrays corresponding to the connections between neurons
    - The layout algorithm also culls the neurons that are less active (and the connections between them). This was to prevent an over-complex display. The culling is tuneable.
- Feed these arrays to the Unity engine for display in the VR engine.

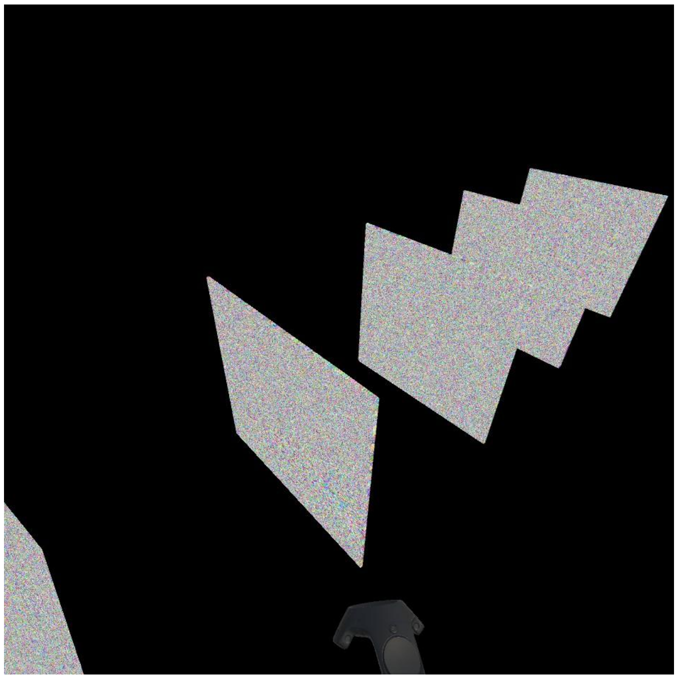
Development History

I'll spare you the stories of installation hassles regarding Unity and C# libraries from NuGet, or trying to make certain types of shader function in VR.

Development was divided between both team members, with one primarily handling VR rendering and interface, with the other primarily handling data processing and model digestion, and both collaborating on particularly troublesome problems. Entertainingly, a fair fraction of the code - not to mention the name of the project! - was generated by ChatGPT and Github Copilot: several shaders, some rendering code, a PriorityQueue, and so forth. The code sometimes needed some tweaks, and sometimes was simply wrong, but often provided the key insights the developer had been missing.

One problem we faced was extracting the shape of the hidden layers. These are not encoded explicitly in the ONNX format, and instead are calculated by the ONNX Runtime (and in fact can depend on the shape of the input data, e.g. batch size). We wrote some code to calculate shapes and propagate constants for some of the common ONNX operators. This was sufficient to compute the shapes of the tensors for the MobileNet model.

Initially we created a separate C# object per neuron and per connection. We positioned each tensor according to its layer number in the X direction, and each neuron randomly in the Y and Z directions.



3. Results

Negative results

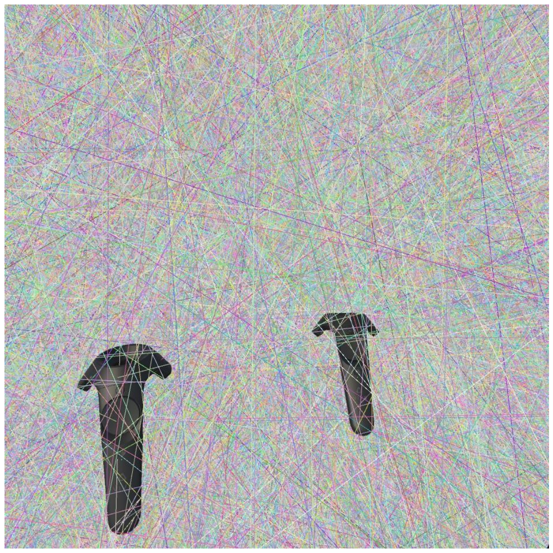
We initially had some serious performance problems. While the system could run fine up to a few hundred thousand elements, once it got into the millions it began to suffer, and we along with it. Considering that machine learning models routinely contain tens or hundreds of millions of elements, this was problematic, and threatened to limit the usability of the system to overly simple examples. While we don't expect our system to load the largest models, we wanted to have a decent range, at least.

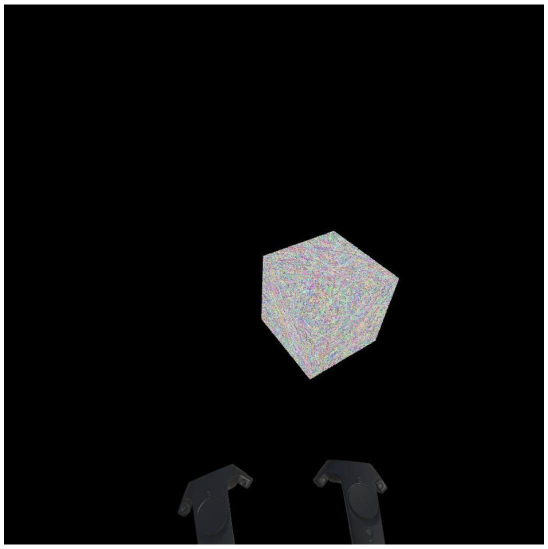
Unity Hang

Constructing C# objects for every neuron and connection turned out to be too computationally expensive, locking up the Unity runtime while it ran, and consuming far too much memory, limiting the size of the model we could display. We could work around the lockups by introducing threading, but it still ran too slow to be able to iterate.

Too many lines

Constructing a "full" graph via coordinate arrays was feasible but there were simply too many lines, reducing the framerate to 2fps and rendering as an impenetrable solid block.



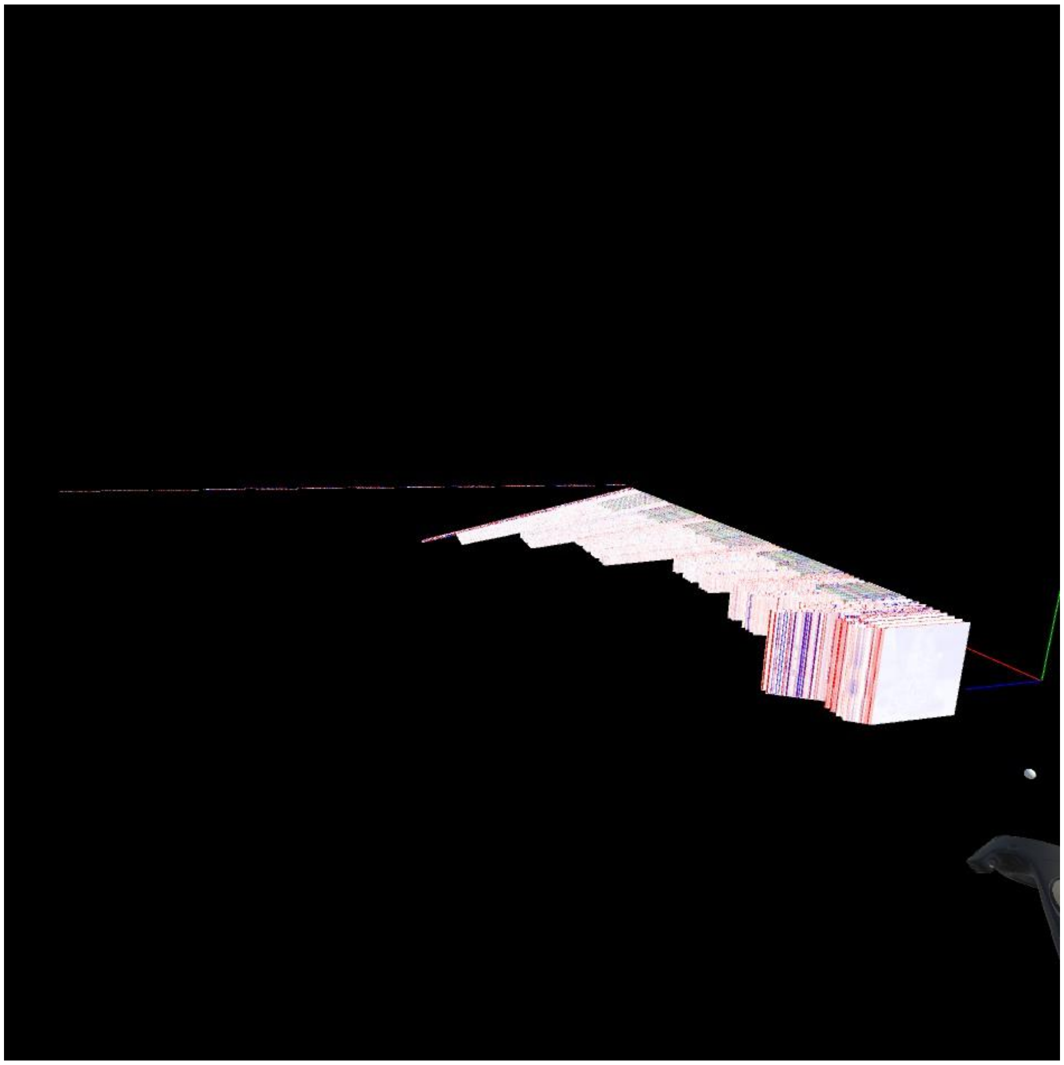


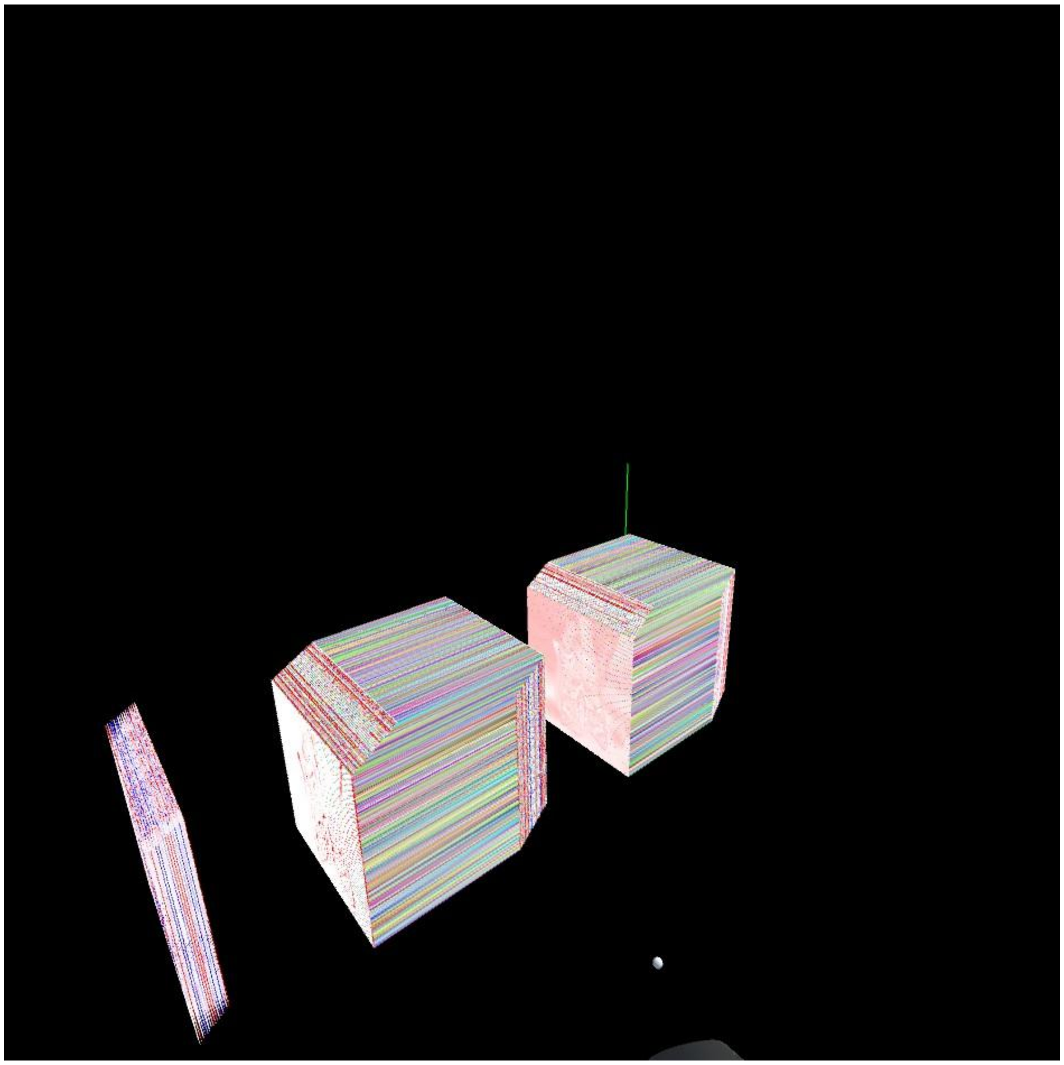
Positive results

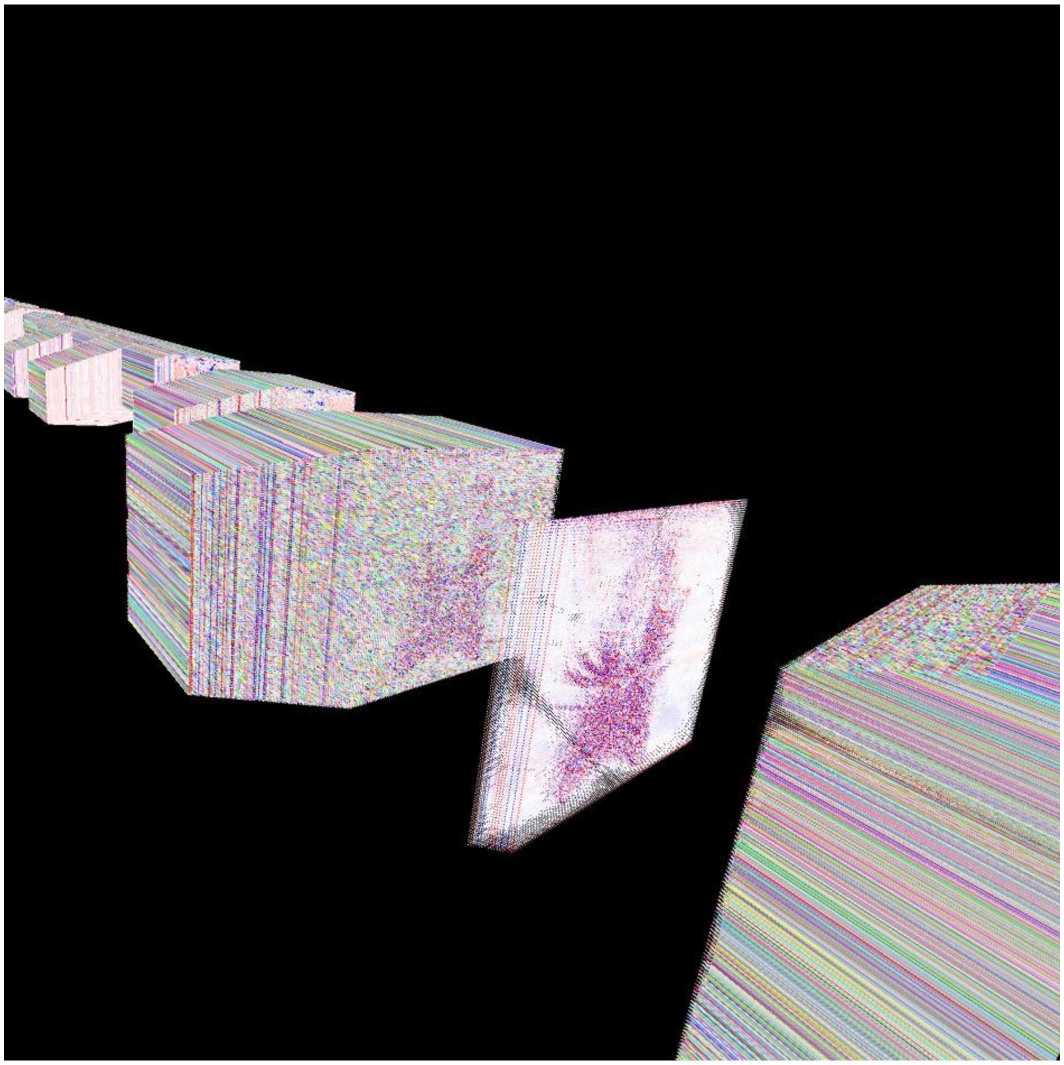
Eventual success

While the object-oriented approach was good for clarity and for development, eventually we transitioned to storing the data in large arrays, formatted to be handed directly to Unity for rendering. This reduced load times from minutes down to ~13 seconds, increased frame rates from 2-30 fps up to 90 fps with room to spare, and increased the capacity of the system up to over 25 million elements easily, at which point our main test model was fully parsed.

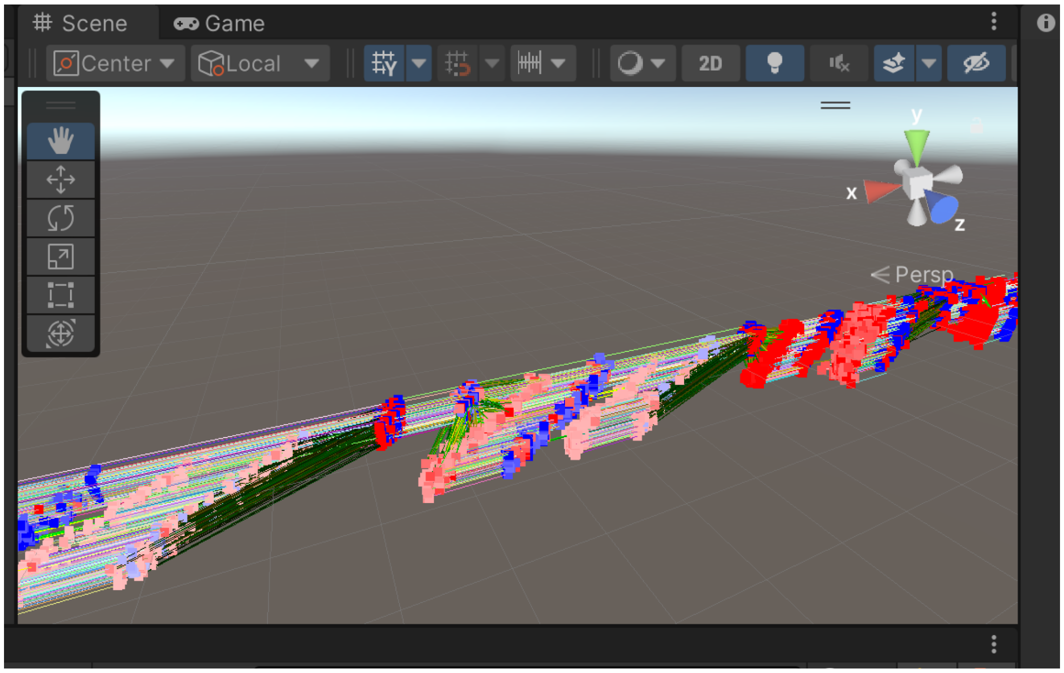
In the end, we'd built a tool with partial support for ONNX models, able to load and inspect them in fine detail, during inference. The tool is open source on github, forming a potential platform for others to improve upon. We also got some cool pics (at a relatively high framerate).



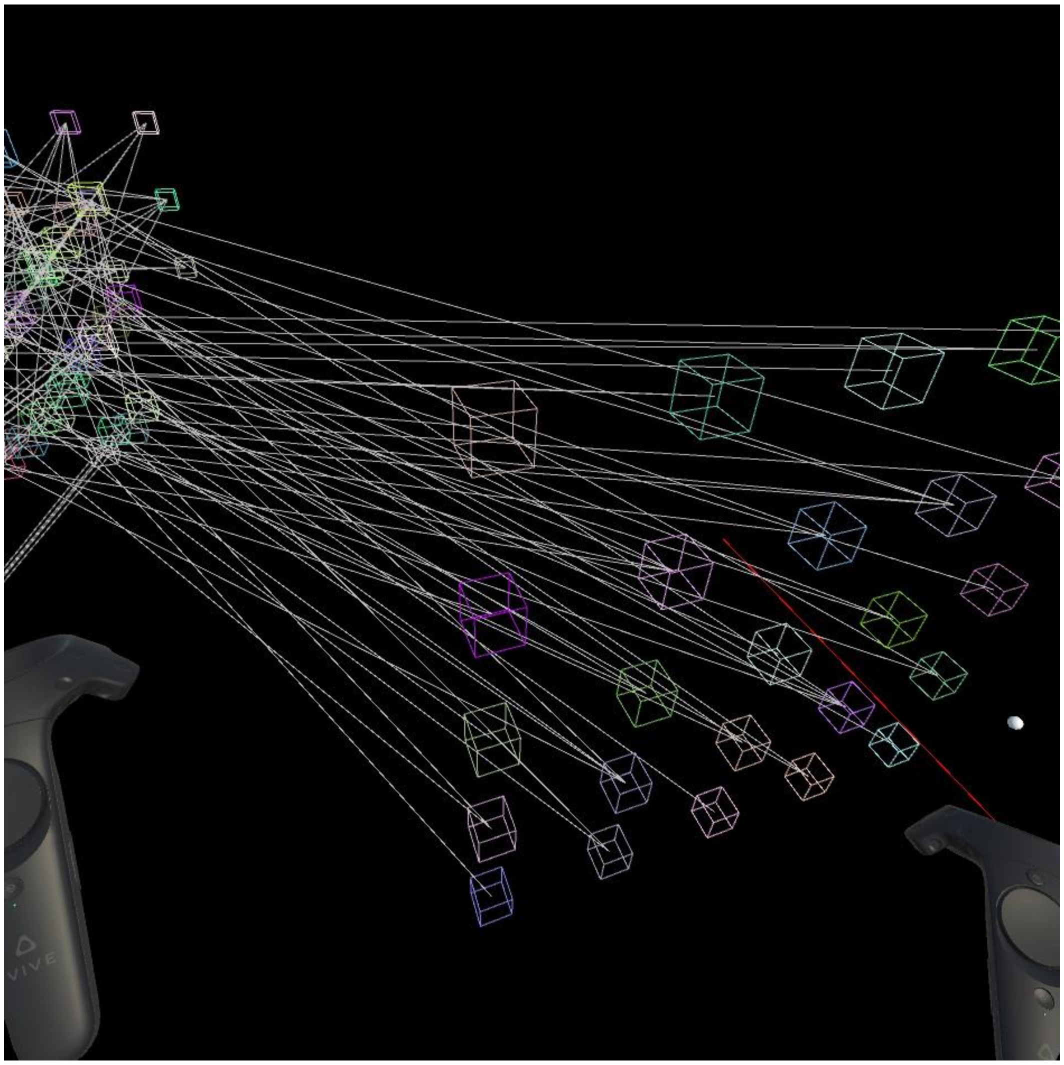




Green showing weights, red and blue showing activation



Toy models showing convolution



Demonstration video:
https://youtu.be/6wkNMwZ_VAU

4. Discussion and Conclusion

Neural nets are massive, complex mechanisms. Their structures can be difficult to reduce to the two dimensions of a computer screen - even reduced to three dimensions, you're still missing several, in most cases! In an attempt to make use of all the dimensions we've got, ONNXplore is an initial stab at a VR tool for inspecting and analyzing ML models, providing a starting point for more tools of this kind, and we hope it's suitable for the task.

Directions for Future Research

- Add support for more ONNX operators and attributes - there are *quite* a few
- Repair some functionality lost on transition from objects to number arrays

- Cleanup of the code detritus left in the rush of the jam
- Allow comparison between two datapoints
- Add realtime control of rendering parameters, such as tensor projection bases, hiding neurons by filter, etc.
- Further optimizations may be possible, such as storing all the data directly in Unity's rendering buffers, rather than copying it in, and updating it in place

5. References

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, (arXiv: 1704.04861) https://arxiv.org/abs/1704.04861

ONNX MobileNet, retrieved May 2023. https://github.com/onnx/models/tree/main/vision/classification/mobilenet

ONNX. Open Neural Network Exchange. 2017, https://github.com/onnx/onnx. Retrieved May 2023.

ONNXRuntime. 2018. https://onnxruntime.ai/. Retrieved May 2023.

Unity, 2005, https://unity.com/. Retrieved May 2023 (version 2022.1.14f1)