Rethinking Model Architecture and Means of Delivery to get Unique Video Interation with Generative Neural Networks Online

Josh Murr

November 12, 2020

Abstract

This paper is about exploring the challenges and limitations of getting an experience like that of *Learning to See* online; working on machines which do not have a powerful GPU and available to many more people.

1 Introduction

Machine Learning (ML) is a part of our everyday lives. For the technical among us it is currently easy enough to see the patterns in the outputs of these black-box systems; as the end of our sentence is predicted as we type, or an item we did not know we wanted is offered to us as we shop. There are however many non-technical users of ML systems who can only see the black-box, or perhaps nothing at all; as Arthur C. Clarke famously said 'Any sufficiently advanced technology is indistinguishable from magic.' It is often the case that exciting new technologies are taken on by artists as readily as industry as the gallery provides a unique environment to present back the to the end user one of these black-boxes, which they may or may not be familiar with, in a drastically different context. Artists such as Mario Klingemann, Terence Broad or Anna Ridler are all excellent examples of artists who have done just this prompting the gallery-goer to experience the medium *as* the message.

Learning to See by Memo Akten is another example, but unique in its own way in that it gives the user an interactive experience of the artwork; an immediate image translation of arbitrary objects on a tabletop to a rich image of waves crashing, the night sky or flowers. The user is able to manipulate the input data as they move the objects on the table and witness how the model interprets this data to produce an output image projected in front of them. Although the model still exists as a black box, being able to see how input-affects-output in real time is extremely powerful. Witnessing the relationship between input and output is in many ways more vital to understanding the function of a Neural Network (NN) than understanding what a neuron is and how backpropagation works. The role of the artist is important here as these artworks have done a huge amount to shed light on dataset bias[6], stimulate conversation about ownership and copyright[19] and show both the capabilities of ML systems *and* the limitations. It is therefore of great importance to make these stimulating

artworks accessible to as many people as possible, which is not as simple as it might seem.

Unfortunately the gallery in itself is a black box to many, entry restricted by a fee or simply by the location of the gallery. Moving such interactive experiences online is a clear next step as once something is online it is immediately available to anyone with an internet connection — this then poses numerous more challenges. Key advances in ML research have shown that deeper models and more data produces better results[*] which means a trained model will consists of many millions of parameters. Storing this amount of parameters can take up gigabytes of memory and so downloading the weights and biases of a pre-trained model can be a time consuming process detracting from the real-time focus. Even then, if a model is that big then it is a given that a significant amount of compute power will be required to run the model at a reasonable speed.

This paper aims to explore each of these challenges and to look at each through the perspective of the artworks previously described with the aim of getting a piece like *Learning to See* online and useable by most users.

2 Background Context

Getting NNs working online and on the clients machine is not a new thing and with modern CPUs it has certainly been shown possible to train and run a simple enough model in the browser on a modest computer[16, 2, 4, 17, 5] and many tools exist which aim to visualise the architecture of a model and show the training process[3, 22, 15]. Such tools are invaluable as learning resources for those dedicated to learning about machine learning. However they are often tailored to that use case expecting the user to have higher motivation for understanding what is going on than the average online passer-by.

There is an increasing amount of tools online which use an interactive machine learning approach allowing the user to retrain a model in real-time to create unique interactions of the users specification[10, 12, 11]. Again this is valuable work and useful to many, but often requires active engagement from the user even if the user-interaction is spoon-fed to them.

Learning to See provides an instinctive way for the user to interact with machine learning, avoiding technical terms or a prerequisite knowledge which is simply not seen online (at the time of writing). This is largely down to the issues described above, and these are especially applicable to image generative models which will be discussed further below.

Learning to See is an adaptation of the Pix2Pix model[14] which is a Conditional Generative Adversarial Network (cGAN)[8]. Unlike a conventional GAN[9], a cGAN takes in two data samples as input, in the case of the Pix2Pix model it is a *target* image and an *input* image. Pix2Pix also learns a structured loss which means it is able to penalize any possible structure that differs between output and target[14]. This model architecture is extremely flexible and has been shown to be useful for many use cases as detailed in the original paper. However neither the original Pix2Pix paper or *Learning to See* had an incentive to create a small model and a good result is simply a sharp image with a predictable output from a given input. Measuring the results of an image generating GAN is an open issue[21] and in an artistic context quatifiying the

outcome is largely pointless. A good result is one which delivers the message of the artists thus if a meaningful interaction can be maintained then perhaps some sacrifices can be made to improve the speed and reduce the size of the model.

Designing and training a NN is of course not limited to the big name ML frameworks like Tensorflow[25] or PyTorch[18], but given the modular nature of a NN and the want in most cases to utilise the GPU for faster training and inference, it is no surprise that a limited number of ML frameworks have become the goto to create and iterate on a novel ML model and to take care of interfacing with a highly optimized backend and the GPU at a lower level. Tensorflow offers a relatively full end-to-end pipeline from designing and training a model in Python, to saving and converting a model into something can be run in the browser using TensorflowJS[27]. A trained model is simply a collection of weights and biases and to use this model on a different device or context, these weights and biases need to be loaded into a model of the same architecture as was used to train the model. The pipeline from training-to-browser is largely about keeping data structures in order and consistent such that the next process knows how to work with it. As part of the process Tensorflow as developed the TensorflowJS Converter[28] which creates a `model.json` file which is the blueprints of the model, and a series of binary files which hold the weights. The pipeline developed by the team at Google behind Tensorflow is great acheivement, as ML model design as a broad field and needs to be flexible; catering for this flexibility is no easy task. The framework needs to interpret and reimplement a model consisting of many layers and potentially millions of parameters in an entirely new context for it work in the browser. This felxibility comes at a performance cost however which can be observed in a naive implementation of *Learning to See* which will be covered in more detail later. In order to recreate a model based on the provided `model.json` file, seperate programs in the chosen backend (WebGL, WebAssembly, WebGPU, etc.) are created to take care of specific tasks such as 2D convolutions, batchnorm, max pooling etc. Moving data through the computational graph is now the difficult part as each of these stages exists as a discrete routine which performs efficient calculations, but getting the data out of the GPU to pass on to the next program comes at a great performance cost and increases latency of the overall system, particularly with web technologies[7].

There exists nothing even similar to *Learning to See* in the catalog of examples provided by TensrflowJS — ie a model which takes video as input and produces video as output. Thus the uniqueness of this use case has yielded a number of hurdles to overcome to acheive something usable on the web for *most users* (users with a reasonably modern laptop, produced in the past 5 years).

3 Method

TensorflowJS is currently the most developed framework for deploying models in the browser so working with the Tensorflow pipeline made the most sense. Fortunately the popularity of the Pix2Pix model has meant that Tensorflow has their own implementation of the model and an accompanying write up on their website[26]. The model was first adjusted according to the *Learning to See* paper[1]: the input and target images are the same, only the input goes

through a series of augmentations details of which can be found in the original paper[1].

The data augmentation requires the generator to take a single channel input. Augmenting the dataset at runtime during training makes forming a dataset very easy as in reality a full dataset is simply a directory of many similar images. As stated earlier, evaluating the model quantitatively by some distance metric has been shown to be of little use, and particularly in this case the quality of the outcome can only really be measured by how well it performs when interacted with. Thus the common process of splitting a dataset into *training* and *testing* data at a 4:1 split was not as important; however approximately 5% of the dataset was reserved for testing after training for visual inspection.

It was found that for a model that takes an input image size of 256×256 pixels and produces an output of the same size, a dataset of 800–1000 images uniformly shuffled with a batch size of 4, trained for 200–250 epochs was sufficient to produce *interesting* results — in this case *interesting* means: varied enough to provide unique experiences whilst producing a output which is representative of the input data. The models were trained on a NVIDIA Quadro RTX 6000 with 24GB memory locally, or a Tesla P100-PCIE with 16GB memory using Google Colab. In either case a usable model will train in a few hours allowing for a relatively quick development cycle. More details on the training process and some tools developed to make the process easy and fast in the appendix.

The Pix2Pix generator is an Autoencoder similar to the "U-Net" architecture[20] but differs in some key ways, and we offer some further changes to drastically reduce the size. The generator consists of 8 2D convolution layers downsampling the input data in width and height, and 7 transpose convolutions upsampling the data back to original size. Each layer is batchnormalized[13] apart from the first, and layers 8–10 have dropout[24] applied at a rate of 50%. The Pix2Pix architecture includes no avg- or max-pooling as all filter kernels are 4×4 and a stride of 2 is used. This has the effect of downsampling the layer input by half in width and height, thus circumventing the need for any kind of pooling, otherwise known as an *all convolutional net*[23]. The difference proposed in this work is to greatly reduce the number of filters used in each layer. The majority of trainable parameters are found in the middle of the generator as a result of the structure described. By reducing the number of filters in the first layer to 8, but following the same pattern of doubling the number of filters each layer to a limit of 512 and then reversing the process to upscale to the output size, very usable results were obtained whilst drastically reducing the model size.

Once a model is trained, it is saved using the `HDF5` format (`.h5`) and converted using the TensorflowJS Converter. The Converter provides options for compressing the model of varying levels of severity. It was found that the compression of the model:

1. Does a great deal to reduce the size (in memory) of the model.
2. Does very little to improve performance in inference.
3. Does very little to affect the output of the model.

Therefore the highest compression of `uint8_affine_quantize` can be used to reduce the size by an order of magnitude, while maintaining only a very slightly compromised output.

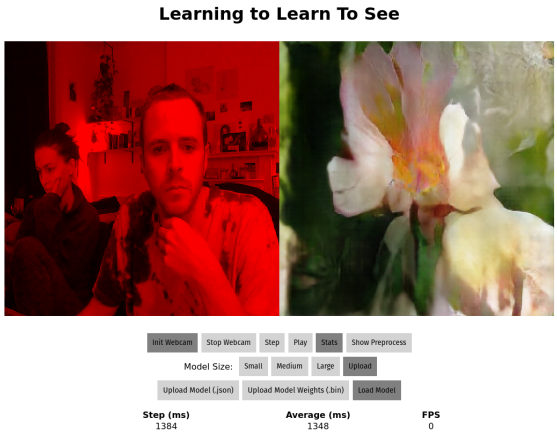
Finally, a bespoke WebGL wrapper consisting of a number of shader programs was created to handle the input data (pre-augmentation to match that in training) and the output data to render the image to the screen as video. It was found that the image handling capabilities of TensorflowJS seem to be tailored to static images. In inference, the data is of the *tensor* data structure and is transformed and manipulated by the model, but at each iteration of the draw cycle when dealing with video input data, a frame of video must be preprocessed and then converted into a format which can be digested by the model, and then the output of the model must be converted back into a data structure which the browser can handle to display as an image, or in our case, video. The internal computation at inference is handled by WebGL shader programs compiled by TensorflowJS, but the input/output was a rather slow process of converting the tensor data back to pixels value-by-value. The input shader programs take data from the webcam as a 16×16 image and scales it up to 256×256 while applying three gaussian blurs — slightly different to the training process but the effect is the same. The output of the model is passed into a very simple shader program as a 32 bit floating point RGB texture and is rendered to a canvas element in the browser.

4 Results

A test-bed was set up with the aforementioned processes in place to test three sizes of model. The project is currently online at `https://learning-to-learn-to-see.netlify.app/`. In the table below *Avg Step Time* is the time taken to create a new frame to output which includes pre-processing and post-processing via the custom WebGL wrapper.

	HDF5 Size in Bytes	Uint8 Compressed Size	# Parameters	Upload Time	Avg Frame Time	FPS
Small	25323104	50	6297491	3	0.1	50
Medium	67280248	16852241	16786211	3	0.1	50
Large	210008464	50	100	3	0.1	50

As already stated, the focus was on finding a sufficiently responsive interaction for the user while maintaining enough variety in the output to maintain



interest and engagement. The target audience is someone with a relatively modern computer (one made in the past 5 years perhaps) and not necessarily someone with a computer with a high specification GPU. For testing the user is able to download a model size at their choosing and different statistics are displayed.

D-GPU means *Discrete-GPU* and *I-GPU* means *Integrated-GPU*. *FPS* is rounded to the nearest number, so 0 FPS means less than 0.5, but in general it will always run at at least produce images.

Performance (FPS) on Various Computers			
Machine	Small	Medium	Large
System76 Laptop 2020, D-GPU	TODO	TODO	TODO
System76 Laptop 2020, I-GPU	26	12	0
Macbook Pro 2017, D-GPU	40	27	0
Macbook Pro 2017, I-GPU	26	14	0
Macbook Pro 2020, I-GPU	25	15	1
Macbook Pro 2015, I-GPU	30	18	0
MSI Laptop 2016, I-GPU	18	12	1
Pixel 3A Phone	3	-	-
Samsung Phone	4	-	-

In general the results are very promising. The average frame rate on an integrated GPU is ∼14FPS which suggests that the average 'everyday' computer will be able to run the *medium* sized model at a useable speed.

The models trained with fewer parameters certainly have a harder time training in that they struggle to generalise. The *102 Category Flower Dataset*[29] has proven to be a useful dataset for testing as it is quite varied; the subject is not always centered in the image, the colours vary drastically and there is great variation in tone and texture throughout. A full sized *large* model is capable of capturing a lot of this texture, however with fewer filters in the smaller models, artefacts of the kernels themselves often appear in the output, as seen below. However this is not commonly seen at runtime as it seems to be the more mottled input image which seem to 'confuse' the model which rarely happens as there is normally a users head in the middle of the frame giving reasonable contrast to the background.

At any rate with this in mind one can train a model with a less varied dataset such as daytime clouds which still produces compelling results in the smaller model as the model needs to generalise less. The clouds dataset was created from time-lapse videos found online of clouds forming.

5 Discussion

6 Conclusion

References

[1] Memo Akten, Rebecca Fiebrink, and Mick Grierson. "Learning to See: You Are What You See". In: (2020). DOI: 10.1145/3306211.3320143. eprint: arXiv:2003.00902.

[2] *Brain JS*. 2018. URL: http://brain.js.org/.

[3] Terence Broad. *Topological Visualisation of a Convolutional Neural Network*. 2016. URL: https://blog.terencebroad.com/archive/convnetvis/vis.html.

[4] Juan Cazala. *Synaptic*. 2017. URL: http://caza.la/synaptic/.

[5] Leon Chen. *Keras JS*. 2017. URL: https://transcranial.github.io/keras-js/.

[6] Kate Crawford and Trevor Paglen. *Excavating AI: The Politics of Images in Machine Learning Training Sets*. Sept. 2019. URL: https://www.excavating.ai/.

[7] Emscripten. *Optimizing WebGL*. 2015. URL: https://emscripten.org/docs/optimizing/Optimizing-WebGL.html#avoid-redundant-calls.

[8] Ian Goodfellow. *NIPS 2016 Tutorial: Generative Adversarial Networks*. 2016. eprint: arXiv:1701.00160.

[9] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. eprint: arXiv:1406.2661.

[10] Google. *Magenta*. 2019. URL: https://magenta.tensorflow.org/.

[11] Google. *Teachable Machines*. 2019. URL: https://teachablemachine.withgoogle.com/.

[12] Mick Grierson. *Mimic*. 2019. URL: https://mimicproject.com/.

[13] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: arXiv:1502.03167.

[14] Phillip Isola et al. "Image-to-Image Translation with Conditional Adversarial Networks". In: (2016). eprint: arXiv:1611.07004.

[15] Minsuk Kahng et al. *ActiVis: Visual Exploration of Industry-Scale Deep Neural Network Models*. 2017. eprint: arXiv:1704.01942.

[16] Andrej Karpathy. *ConvNetJS*. 2016. URL: https://cs.stanford.edu/people/karpathy/convnetjs/.

[17] Steven Miller. *Mind*. 2016. URL: http://caza.la/synaptic/.

[18] PyTorch. *PyTorch*. 2019. URL: https://pytorch.org/.

[19] Aja Romano. *A guy trained a machine to "watch" Blade Runner. Then things got seriously sci-fi*. 2016. URL: https://www.vox.com/2016/6/1/11787262/blade-runner-neural-network-encoding.

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. eprint: arXiv:1505.04597.

[21] Tim Salimans et al. *Improved Techniques for Training GANs*. 2016. eprint: arXiv:1606.03498.

[22] Daniel Smilkov and Shan Carter. *Tensorflow Playground*. 2016. URL: http://playground.tensorflow.org/.

[23] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2014. eprint: arXiv:1412.6806.

[24] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[25] Tensorflow. *Tensorflow*. 2019. URL: https://www.tensorflow.org/.

[26] Tensorflow. *Tensorflow Pix2Pix*. 2019. URL: https://www.tensorflow.org/tutorials/generative/pix2pix.

[27] TensorflowJS. *TensorflowJS*. 2019. URL: https://www.tensorflow.org/js.

[28] TensorflowJS. *TensorflowJS Converter*. 2019. URL: https://github.com/tensorflow/tfjs/tree/master/tfjs-converter.

[29] Oxford University. *102 Category Flower Dataset*. 2008. URL: https://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html.

A Training Process

Some text that will be the appendix.