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

Josh Murr

November 14, 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."*[34] It is often the case that exciting new technologies are adopted by artists as readily as industry because the gallery provides a unique environment to present back the to the user one of these black-boxes in a drastically different context. Artists such as Mario Klingemann[20], Terence Broad[4] or Anna Ridler[23] are all excellent examples of artists who have done just this; delivering the *medium as the message*.

Learning to See by Memo Akten[1] 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 everchanging 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

Figure 1: A frame from Learning to See using a model trained on ocean waves. — ©Memo Akten,2017

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[9], stimulate conversation about ownership and copyright[24] 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. Even a modestly sized NN can contain many millions of parameters. Modern computers are able to perform millions of floating point operations per second (FLOPS) but advances in ML research have shown that *deeper* models and more data produces better results[6]; as such, without expensive hardware accelleration from discrete graphics processing units (GPU) or even more modern tensor processing units (TPU), an average modern computer cannot crunch the numbers needed to run a large, deep NN. Even if one *could* perform the necessary FLOPS, storing this number 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.

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[19, 3, 7, 21, 8] and many tools exist which aim to visualise the architecture of a model and show the training process[5, 27, 18]. 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 number of tools available 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[13, 15, 14]. 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[17] which is a Conditional Generative Adversarial Network (cGAN)[11]. Unlike a conventional GAN[12], 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[17]. 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; a good result is simply a sharp image with a predictable output from a given input. As stated by the Pix2Pix authors measuring the results of an image generating GAN is an open issue[26, 17] and in an artistic context quatifiying the outcome is largely pointless. A good result is one which delivers the message of the artist, thus if a meaningful interaction can be acheived 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[30] or PyTorch[22], 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[32]. 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[33] 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[10].

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. The model for this work was based on the Tensorflow implementation[31] and adjusted according to the *Learning to See* paper: 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[2].

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

Figure 2: A caption.

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[25] 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[16] apart from the first, and layers 8–10 have dropout[29] 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*[28]. 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 each layer by up to a factor 8, very usable results were obtained whilst drastically reducing the model size.

Generator Layer	Pix2Pix Filters	Our Filters
1	64	8
2	128	16
3	256	32
4	512	64
5	512	128
6	512	256
7	512	512
8	512	512
9	512	512
10	512	256
11	512	128
12	512	64
13	512	32
14	256	16
15	128	8

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:

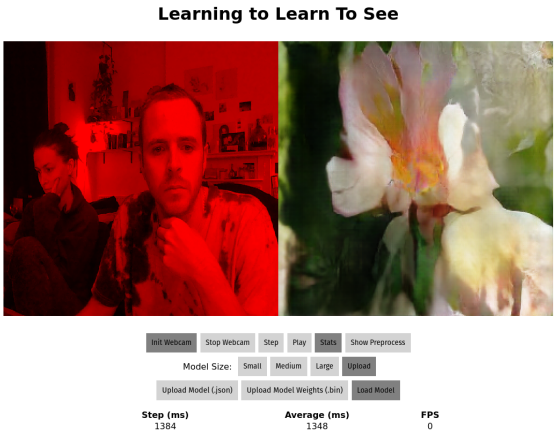
1. Does a great deal to reduce the size (in memory) of the model.
2. Does very little to improve performance in inference.
3. Does very little to affect the output of the model.

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 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 created with the aforementioned processes to test three sizes of model. The project is currently online at `https://learning-to-learn-to-see.netlify.app/`.



	HDF5 Size (mb)	Uint8 Compressed Size (mb)	Compression	Total Parameters	Upload Time (s)
Small	25.3	6.3	25.2%	6,297,491	8.3
Medium	67.3	16.9	25.0%	16,786,211	9.9
Large	217.8	54.5	25.0%	54,423,811	27.5

As already stated, the focus was on finding a sufficiently responsive interaction for the user while maintaining enough variety in the output to maintain engagement. The target audience is someone with a relatively modern computer 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 or upload their own, and different statistics are displayed.

In the table below *D-GPU* means *Discrete-GPU* and *I-GPU* means *Integrated-GPU*. *FPS* is rounded to the nearest number.

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. However a good result in our case in more dependent on the quality of the interaction.

It can be observed that models trained with fewer parameters struggle to generalise. The *102 Category Flower Dataset*[35] 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

Figure 3: A caption.

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.

In training the model is attempting to learn a mapping from a blurry greyscale image to a 3-channel, well defined image. As a user interacts using the online platform one can quickly see the effects of bright or dark areas, as in such an abstract input, the model appears to associate these bright spots with key areas on interest in the output.

With this and more knowledge of the models limits in mind, one can train a model with a less varied dataset such as daytime clouds which still produces compelling results even in the smallest model as the model needs to generalise less. A dataset produced from a timelapse of the Aurora Borealis was used for the smallest model. The smallest model does not produce particularly exciting results, but still does a good job of illustrating the relationship between input and output, and has even been shown to work on mobile devices.

5 Discussion

6 Conclusion

References

[1] Memo Akten. *Learning to See*. 2017. URL: http://www.memo.tv/works/learning-to-see/.

[2] 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.

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

[4] Terence Broad. *Terence Broad*. 2020. URL: https://terencebroad.com/.

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

[6] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. eprint: arXiv:2005.14165.

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] Mario Klingemann. *Mario Klingemann*. 2020. URL: http://quasimondo.com/.

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

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

[23] Anna Ridler. *Anna Ridler*. 2020. URL: http://annaridler.com/.

[24] 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.

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

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

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

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

[29] 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.

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

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

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

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

[34] Alvin comp. Toffler. *The futurists. Edited with an introd. by Alvin Toffler*. Random House, 1972.

[35] 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.