# Chapter 30 – Generative Neural Networks

## Deep fakes and van Gogh

### 30.1 - Introduction

The primary purpose of neural networks has long been the prediction or classification of an output from a set of relevant inputs. The class of algorithms is also referred to as discriminative. However, more recently, advances in both algorithms and necessary hardware to train them, paved the way for neural networks to start creating the very inputs they once were tasked to classify. Today, neural networks are used human sounding speech from text, compose music, and even artificial human genomes. But where it has been most prolific is, logically, in the space of images and video. Some of these neural networks are now capable of painting a perfectly acceptable van Gogh, together with the more sinister ability to create footage seemingly so real, it will have everyone belief they are watching Tom Cruise play the guitar, with all his unique mannerisms and body language using a method called Deepfake. This is an exciting technological development, but one that is worrisome for society at large. However, deepfakes can have a beneficial application as well, perhaps starting with recreating what was taught to be lost forever, like the presence of a lost loved one, or generating artificially what that is needed but does not exist, like artificial human genomes for research. This chapter will discuss two such generative methods, one which we have already seen. The demos a relatively simple examples, in terms that they use simple images. This is because this latest generation of neural networks is still pushing the boundaries of what we can teach in terms of the hardware needed to accomplish that. Simple CPU run algorithms fall short, or will take too long to train, and new technologies such as GPUs and TPUs will need to step in.

### 30.2 - Creating something from nothing

Before we start looking at two generative algorithms, it is perhaps good to reiterate what most deep belief neural networks are accomplishing within the depts of their hidden layers. We have done that before in some sense, but now we can apply that idea to not only capturing complex patterns, but also on how to recreate them, or generate new ones with similar properties. So, let’s imagine we want to design a neural network to design a car for us that is a unique product of its own imagination. Humans have already designed countless of different cars; how do we transfer that creative process to a machine? The answer is to understand the concept of a car. Let’s step away from images for a second and let’s assume we can just talk directly to the neural network, as a thought experiment. The first obvious question we get after asking a neural network to design a car: what is a car? But rather then telling it, we just give it a large random subsampling of objects to inspect, including cars, but also a lot of other random things, like apples, boat, airplanes, monkeys, etc. A successful network will eventually start to realize something that is somewhat unique to cars (and not other objects) and all cars (and not other objects) have in common. A car typically has a chassis, on average 4 wheels, seats, steering wheel, engine, battery, etc. Although cars can vary from each other in an infinite way, they have the same things in common. Otherwise, they wouldn’t be cars, right? Look more closely, and you might recognize how we came across this idea earlier, when we discussed dimensionality reduction. However, in dimensionality reduction our aim was to represent something of a large dimensionality (e.g., the pixels in an image) into something with a much lower dimensionality, like a set of ‘eigenfaces’. With generative neural networks, the aim is similar, but typically serves a different purpose (although technically our eigenfaces demo was already an example of a generative model, just not a neural network) Conventional predictive algorithm are done when they figure out this relationship: all cars are some mixtures of different parts. With generative neural networks we can go a step further. Since it knows the recipe of cars, so to speak, we can ask to randomly recombine mixing those parts back together. Perhaps it puts the engine of a fiat in a Ferrari chassis. Probably not the best, but still a car. Perhaps another time it while make a hatchback with giant spoiler, but still a car. And perhaps by randomly recombing all the bits and pieces it knows about what constitutes a car it will come up that is perfectly acceptable, and perhaps even an innovation on the design of cars altogether. At the core, generative neural networks operate on that principle (amongst others). What differs between algorithms is 1) how they figure out the parts and how 2) they put them back together.

### 30.3 – Auto-encoders

Consider, once again, our face image data. With this data, how do we train a generative model that can create completely artificial faces that nevertheless could look perfectly human (with enough training and training data)? Again, we did this using PCA in an earlier demo, but perhaps we can do better? If we want a more advanced and more capable version of that, auto-encoders are good place to start with generative neural nets. Auto-encoders have many useful applications, including compression, reconstruction, denoising, dimensionality reduction as well as the generative use case we describe here. But all these applications rely on the same structurally feature that describes auto-encoders: they are bottlenecks.

### 30.4 – Bottlenecks as manifold learning

An autoencoder works in the following way. We give it inputs to train on (e.g., images, soundwaves, PHI). Like a feed forward network, the data is passed through a series of consecutive layers that at first will have fewer and fewer nodes. However, midway through the hidden layers, we reverse this process, having ever more nodes in each layer. How many nodes the final layer will have depends on our use case. But in most applications, including generative auto-encoders, we will end up with the same nodes in the output layer as the input layer. So, for our face data, the number of input and output nodes is equal to the number of pixels in our face images. Training can use all the common techniques used in deep belief neural network, with exception of our error and loss function. Instead of defining error as the difference between network output and some objective ground truth, we define it as the difference between output and input. During training, the neural network is thus penalized for not producing the same output as its input. In other words, by passing it through the layers and the bottleneck, it must figure out how to reconstruct the same output from its input. The conceptual trick is that because it must pass it through the bottleneck, it needs to find a way to compress the input to a smaller dimensionality (the number of nodes) before reconstructing it again. In our example, it will have to compress the face images number of pixels into a smaller number of so-called latent dimensions. These are the smallest number of nodes the data must pass through. This need for compression informs the need for finding the smaller set of base elements that constitute a face (or a car).

30.5 – Compression, reconstruction and denoising

Before we discuss how such compression is subsequently used to generate artificial stimuli it is perhaps useful for what functionality auto-encoders provide through this process. The first we already discussed: compression. Depending on the quality of the reconstruction our auto-encoder is capable off, we might use it to seriously cut down on data needs and improve speeds down other AI pipelines. Second, it can use to reconstruct or denoise inputs. If the compression the auto-encoder isn’t perfect, it will never be able to reconstruct inputs back to the same output. But this in some cases is more a blessing than a curse. Because anything that is random noise can never be captured structurally by one of our latent dimensions. It is after all noise; the neural network cannot learn to predict it. Therefore, in passing a new input containing such noise, the output is without such noise.

30.6 – Segmentation - UNet

30.7 – Synthesis

Generation of completely new and artificial outputs with an auto-encoder is done as described above in our fictional car example. Technically, this means we will not be passing any inputs through our entire network. Instead, we can assign random inputs to the bottleneck nodes themselves. Properly trained, an auto-encoder can take these input as random weights on its latent dimensions and combine them into by passing them through the widening second part of the network, generating a completely artificial output that nonetheless captures the essentials bits of what define the stimuli we trained it, if such defining characteristics do exist.

30.8 – Variational Convolutional Auto-encoders

Auto-encoders can be extended in several ways. While the overall architecture of the bottle neck is the defining feature, the structure of its layers doesn’t have to be. Convolutional auto-encoders, for example, are what conventional neural CNNs are to conventional feed forward deep belief neural networks. Instead of traditional fully connected layers, a convolutional auto-encoder has convolutional layers. And like CNNs, this makes them ideal for working with image data.

30.9 – Generative Adversarial Neural Networks

More recently, a class of networks called generative adversarial neural networks or GANs was introduced that could produce artificially generated content and data with far better fidelity than the previous generation of auto-encoders. Not only that, but it also introduced a new and clever approach to learning a particular task, generative or otherwise. In GANs, we have not one, but two separate networks. One, the generator network is in charge of generating examples out realistic outputs (image or otherwise), while the other network, called the discriminator, is in charge of learning to spot a real example of the input, or one that was generated by the generator. In essence, it is a game they play. The generator tries to fool the discriminator, while the discriminator attempts to spot the fraud. Training the entire network is a zero-sum game, meaning that the game (i.e., training) ends when two players are in equilibrium. In its original formulation, the generator is not given the true outputs it is trying to generate itself. Instead, it is initialized with random weights, and it receives random noise with no relationship to the input. This is called the seed. Passing it through its architecture, the generator creates and output that serves as the subsequent input to the discriminator. The discriminator passes that input through its own architecture with the aim of rendering a decision: the real thing or generated by the generator. However, the discriminator is also given actual inputs, inputs the generator has never seen. But like the generator, the discriminator’s network is also initialized with random weights. Back to our face data as an example. First, we give generator some random input and generate an output image. With random weights, this output obviously will like nothing, and certainly not a face. We flip a coin to decide whether to give that output to the discriminator or use a face image from our actual data instead. The discriminator will then classify that input as real or fake. But the discriminator is also clueless about what constitutes an image of a real face, it has also random weights. So, at first it will guess randomly. When it gets the prediction wrong, we penalize it by updating its weights. If it gets it right, we instead penalize the generator and update its weights. Repeating this process iteratively, we thus push each of them to be better at their job: fooling or not getting fooled. This process is stopped when the discriminator has an accepted false positive rate: it is being misled often enough in thinking the generator’s inputs are real. Like auto-encoders, GANs aren’t tied to a specific type of internal layer architecture. Thus, like auto-encoders, GANs with generators and discriminators that have convolutional layers are ideal for working with image data.

30.10 – Diffusion Models

30.11 – Neural Style Transfer

Another interesting example of a generative neural network approach is Neural Style Transfer (NST). Since it perhaps not quite as relevant to MSK as other networks, I’ll only discuss it briefly, but the web has many cool demos and examples of the technique. Put simply, it aims to take a content image (like a photograph) and transform it such that it appears to be drawn or painting a particular style, whether this is an individual artist, art period, or some other recognizable style, called the style image. We start with a standard CNN used to classify objects. First, we feed the photograph image through it and remember the activations of all weights in all layers. We do the same for the style image. The idea is that the ‘style’ of an image is encoded in early layers of the network we image detail is encoded. In contract, the ‘content’ of an image is encoded in the later layers, where nodes start to encode high level objects, like cars, cats, trees, etc. Therefore, the NST attempts finds a nice mixture of the activation in the lower layers corresponding to the style image, and activation in the higher layers corresponding to the content image. This generates, or more aptly reverse engineers, the image evoked that response, allowing you to draw a Picasso like painting of your cat if you had 1) an image of your cat and 2) an image of a Picasso painting.

30.12 – Deep dreaming

30.13 – Deep fakes

### 30.14 - Conclusion

The way that these newer networks can generate realistic looking data, especially in the image and speech domain is truly impressive. It’s likely to become harder to distinguish them from the real deal. This is of course worrisome because how it can be misused. But it offers opportunities as well. Not only can their generated output be useful, but we can also dive into their inner workings and see what they have learned. Like deep belief neural networks, it offers the ability reveal the underlying pattern in complex processes and patterns that are difficult to find otherwise.

### 30.15 - Demos

There are two demos that show how to implement an autoencoder and GANs in TensorFlow and one how to build a diffusion model in PyTorch. You can train them on any dataset you find interesting (I included (MNIST and the CelebA face data). Keep in mind that on standard GPU machines and with the data provided, convergence will be slow, especially for the CelebA data.

### 30.16 - References

https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee

https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

https://arxiv.org/abs/1701.00160

https://blog.paperspace.com/nvidia-gaugan-introduction/