Transfer Learning, GANS, and Introduction to Autoencoders

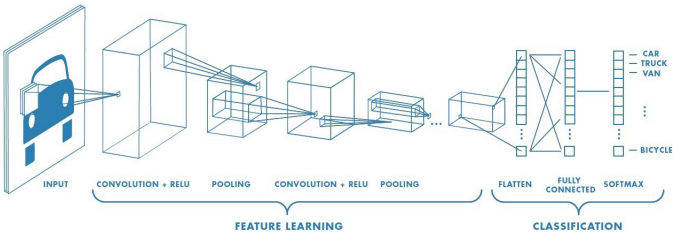
Saahith Janapati*

March 2019

1 Transfer Learning

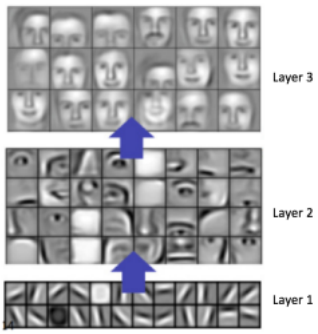
If you recall from a two weeks ago, we finished our discussion of Convolutional Neural Networks. In practice, most don't actually train entire CNNs. Instead we use pretrained models and make use of transfer learning.

Practicing transfer learning is a great way to alleviate some of the computational and time requirements necessary to train a CNN (or a basic neural network). However, prior to exploring transfer learning, it is essential that one understands what a Convolutional Neural Network is learning at each layer.



1.1 What are CNNs learning?

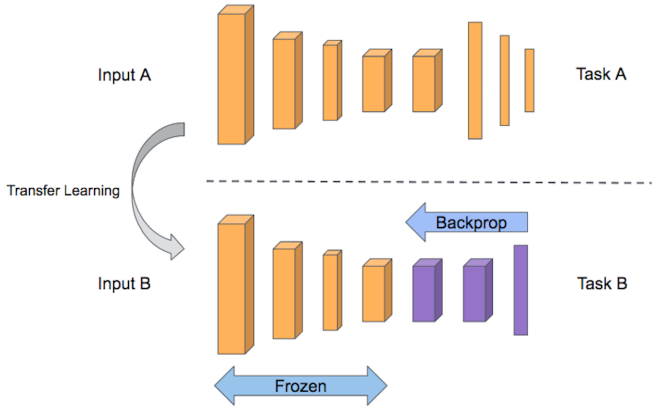
In the initial layers of a CNN, each output node in the layer corresponds to a small portion of the actual image. This can be seen in the first layer of the CNN above. The node depicted in the first hidden layer only "views" a small portion of the car, specifically the left-middle section. So, the information gained in these layers about the image corresponds to small, minute features such as edges. In the later layers, all this information is then compounded together to detect more complex features. This can be seen in the following image.



*The GANs and Autoencoders sections of these lectures are from Alan Zheng's and Mihir Patel's previous lectures on these topics respectively. The section titled Transfer Learning is from a previous lecture of mine.

1.2 Transfer Learning in Practice

Transfer learning takes advantage of the fact that small features, such as edges, are found in many different types of images. So, the initial layers of a CNN trained on one dataset can be useful for classifying many images (even images from a different dataset). Libraries such as Tensorflow and Keras already have CNNs pretrained on datasets such as ImageNet. With just a couple of lines of code, you can import such a CNN and use it out of the box. However, let's assume that you want to train this CNN for your own task. Instead of training the whole network again on your images, which can be extremely computationally intensive, we can choose to only train the later layers of the network and freeze the initial layers. In other words, we will not be conducting backpropagation on the weights corresponding to the initial layers of the network.



Why does this work? Remember that the initial layers of a network can be viewed as feature extractors. The entire function of these initial layers is only to detect small features such as edges. The initial layers do not depend very much on the context of the dataset that you are training on. By freezing the initial layers, you can save time and computation during training. And you'll still be allowing your model to learn to detect the larger features that are specific to your problem (i.e faces, cats, dogs). This works very well if the datasets that the model was trained on initially is similar to the dataset you're trying to train you model on.

2 Introduction to GANs

Generative Adversarial Networks (GANs) are a seminal type of generative model, introduced in 2014 by the University of Montreal. GANs have been heavily used in various generative tasks with impressive results. GANs are most actively used for image generation tasks: plain image generation, image inpainting, super-resolution image generation, and text-to-image. Over these few years, however, they have also been used in real-life applications for text/image/video generation, drug discovery and text-to-image synthesis. Since GAN's first release, there have been multiple iterations on different types of GANs; here, we'll cover the basics only.

2.1 What does it do

In short, they belong to the set of algorithms named **generative models**. These algorithms belong to the field of **unsupervised learning**, a sub-set of ML which aims to study algorithms that learn the underlying structure of the given data, without specifying a target value. Generative models learn the intrinsic distribution function of the input data ***p(x)*** (or ***p(x,y)*** if there are multiple targets/classes in the dataset), allowing them to generate both synthetic inputs x' and outputs/targets y', typically given some hidden parameters.

In contrast, supervised learning algorithms learn to map a function ***y'=f(x)***, given labeled data y. An example of this would be classification, where one could use customer purchase data (x) and the customer respective age (y) to classify new customers. Most of the supervised learning algorithms are inherently **discriminative**, which means they learn how to model the conditional probability distribution function

(p.d.f) $\boldsymbol{p(y|x)}$ instead, which is the probability of a target (age=35) given an input (purchase=milk). Despite the fact that one could make predictions with this p.d.f, one is not allowed to sample new instances (simulate customers with ages) from the input distribution directly.

2.2 Intuition

Although the GAN architecture can take on several possible forms, the vanilla GAN consists of two neural networks.

The first model is called a Generator and it aims to generate new data similar to the expected one. The Generator could be assimilated to a human art forger who creates fake works of art.

The second model is named the Discriminator. This model's goal is to recognize if an input data is 'real' — belongs to the original dataset - or if it is 'fake'—generated by a forger. In this scenario, a Discriminator is analogous to an art expert who tries to detect works as truthful or fraud.

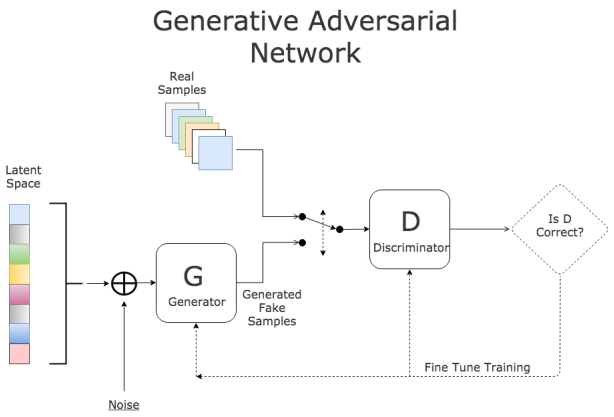


Figure 1: The GAN pipeline.

How do these models interact? Paraphrasing the original paper which proposed this framework, it can be thought of the Generator as having an adversary, the Discriminator. The Generator (forger) needs to learn how to create data in such a way that the Discriminator isn't able to distinguish it as fake anymore. The competition between these two teams is what improves their knowledge, until the Generator succeeds in creating realistic data.

2.3 Math

A neural network G(z, θ_1) is used to model the Generator mentioned above. It's role is mapping **random input noise variables z** to the **desired data space x** (say images). Conversely, a second neural network D(x, θ_2) models the discriminator and outputs the probability that the data came from the real dataset, in the range (0,1). In both cases, θ_i represents the weights or parameters that define each neural network.

As a result, the Discriminator is trained to correctly classify the input data as either real or fake. This means its weights are updated as to maximize the probability that any real data input x is classified as belonging to the real dataset, while minimizing the probability that any fake image is classified as belonging to the real dataset. In more technical terms, the loss/error function used **maximizes the function D(x), and it also minimizes D(G(z))**.

Furthermore, the Generator is trained to fool the Discriminator by generating data as realistic as possible, which means that the Generator's weights are optimized to maximize the probability that any fake image is classified as belonging to the real data. Formally this means that the loss/error function used for this network **maximizes D(G(z))**.

In practice, the logarithm of the probability (e.g. log D(...)) is used in the loss functions instead of the raw probabilities, since using a log loss heavily penalizes classifiers that are confident about an incorrect classification.

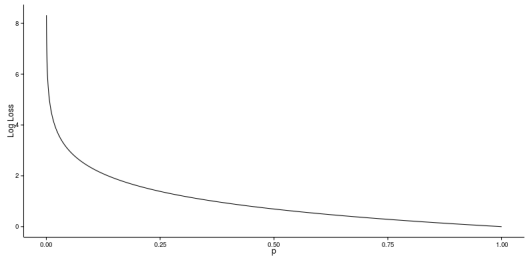


Figure 2: Log Loss Visualization: Low probability values are penalized significantly more.

After training for a while, if the Generator and Discriminator have enough capacity (if the networks can approximate the objective functions), they will reach a point at which both cannot improve anymore. At this point, the Generator generates realistic synthetic data, and the Discriminator is unable to differentiate between the two types of input.

Since during training both the Discriminator and Generator are trying to optimize opposite loss functions, they can be thought of two agents playing a minimax game with value function V(G,D). In this minimax game, the generator is trying to maximize it's probability of having it's outputs recognized as real, while the discriminator is trying to minimize this same value.

$$min_G\, max_D V(D,G) = E_{x\ p_{data}(x)}[\log D(x)] + E_{z\ p_z(z)}[\log(1 - D(G(z))]$$

Thus, the gradient expression we train the discriminator on is as follows:

$$\nabla \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(z_i))]$$

where x is the real data in a given batch, z is the noise for the generator for the given batch, D is the discriminator function, and G is the generator. We want to maximize this expression. The first expression in the summation, $\log D(x_i)$, corresponds to the discriminator output on real data; we clearly want to maximize this probability. The second is a bit more complicated: the $D(G(z_i))$ corresponds to the discriminator's probability estimate that the generated data is real. We want to minimize this, so we make the term $\log(1 - D(G(z_i))$. If you haven't taken multivariable calculus yet, think about the ∇ as a derivative; we simply want to move in the direction that maximizes the summation.

The gradient expression we train the generator on is as follows:

$$\nabla \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z_i))$$

We want to minimize this expression (i.e. we want to maximize $D(G(z_i))$, the probability of the generated data being real, as determined by the discriminator).

Note that GANs are notoriously difficult to train. This is because GANs are highly unstable; in order to train correctly, we need the generator and discriminator to be roughly on equal levels throughout the training process. If the discriminator overpowers the generator, there will be little gradient for the generator to learn upon; vice-versa, and we run into *mode collapse*, where the generator produces outputs with extremely low variety.

2.4 Applications of GANs

GANs are not yet frequently used in applications at the high school level, so finding a decent (novel) application for them could be a great project idea.

Here are some examples:

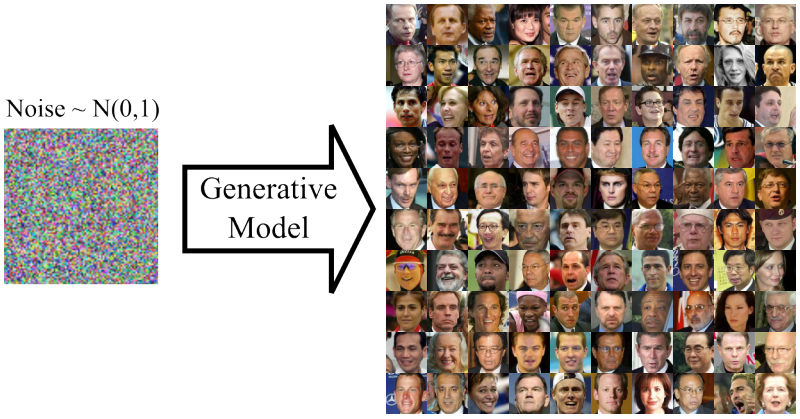


Figure 3: Image generation.

This has been done with a variety of GANs (e.g. Wasserstein GAN), and can be done (with limited success) with vanilla GANs. There are also architectures that convert a caption to an image, with remarkable success (e.g. StackGAN++).

Figure 4: Caption generation.

Conditional GANs (as well as RNNs) have been used, to much success, for this task.

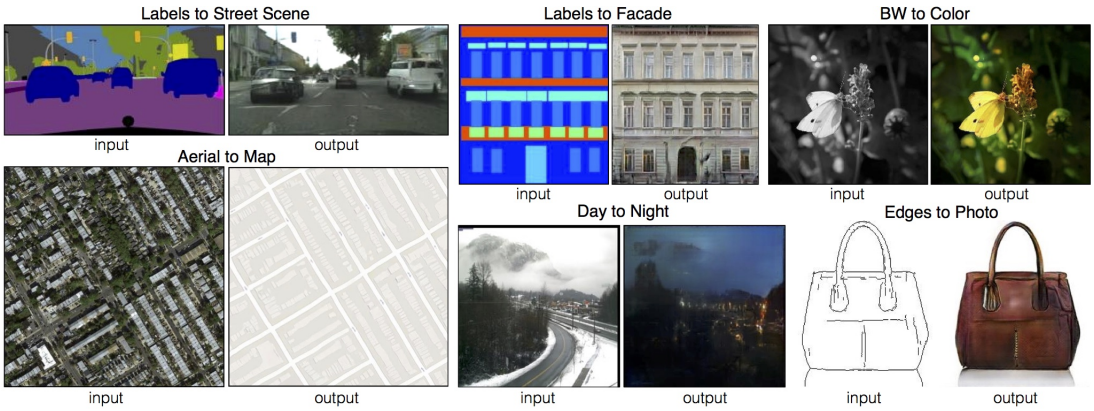


Figure 5: Image mapping.

Two prominent networks for this task are the pix2pix network (Conditional GAN) and the CycleGAN. The results, as you can see, are quite impressive, and can be extended to a variety of image-based tasks.

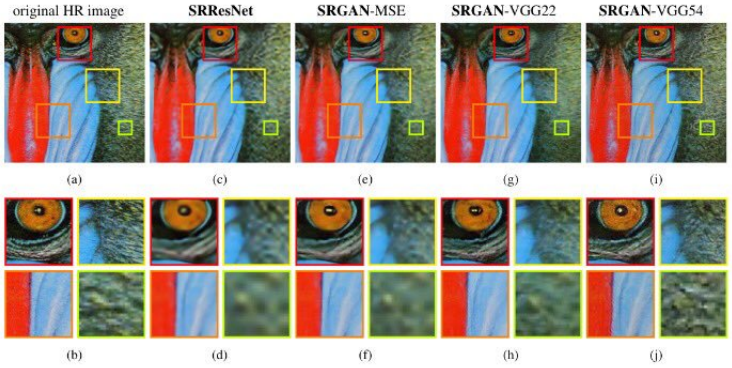


Figure 5: Reference HR image (left: a,b) with corresponding SRResNet (middle left: c,d), SRGAN-MSE (middle: e,f), SRGAN-VGG2.2 (middle right: g,h) and SRGAN-VGG54 (right: i,j) reconstruction results.

Figure 6: Super-resolution.

GANs have been used in the task of super-resolution, interpolating finer texture details that are lost in a low-res image. The most prominent architecture for this task is the SRGAN. The SRGAN has been used to moderate success (though MOS scores are subjective and difficult to validate).

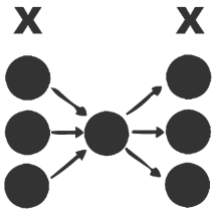
2.5 History of GANs

Although generative models have been a key part of machine learning for quite some time, GANs were first formally introduced by Ian Goodfellow and colleagues from the University of Montreal in 2014. Since then, research in GANs and other generative models has exploded and there has tremendous progress in the field since then, as visualized in the following picture.

Ian Goodfellow, at the time of this writing, is the Director of Machine Learning at Apple.

3 Introduction to Autoencoders

One of the issues with machine learning when going from raw data is too many input variables. The sheer number of inputs makes it hard to train and often can lead to overfitting (see curse of dimensionality). Autoencoders were originally proposed as a method of reducing dimensions and extracting higher level features that we know for sure contain most if not all of the information. It can be thought of as storing the information in a more efficient and meaningful way.



3.1 Definition

Autoencoders, instead of narrowing down like most networks, instead are shaped like an hourglass. When trained, the output should be exactly the same as the input. So if we feed the network a picture of a cat, it will give the exact same picture back. This forces the network to try to maintain all of the information through each layer and not lose anything. However, in the middle it gets narrower! This means it must find a more effective way of storing the same amount of information. The first half that converts the information into the narrow region is called the encoding portion and the second half that converts it back into the original information is called the decoding portion.

3.2 Example of Autoencoders: Number Systems

This can be a bit tricky to wrap our head around, so lets start with an example. Say we have a network that takes in 8 inputs. Each input node represents a number. So if we want to pass the number 1, we make the first neuron a 1 and the rest 0s. If we want to pass the number 2, we make the second neuron a 1 and the rest 0s. And so on. This process is actually called one-hot encoding. We then create a network with a very simple structure. Besides the input layer with 8 nuerons, we will add a hidden layer with 3 neurons and another output layer with 8 neurons. Now we train the network to spit back out the same exact input. This means it MUST learn to take the input and store it in only 3 neurons if it wants to learn. In this case, it can just learn binary! One of the three hidden ones represents +1, another represents +2, and another represents +4. This way, we can represent 8 different numbers (2^3).