Generative Adversarial Networks

April 22, 2019

I wrote this note as I study papers about generative adversarial networks (GANs). The main paper that I read is the tutorial by Ian Goodfellow at NIPS 2016 [3].

1 Introduction

- A generative model is capable of generating samples of a probability distribution.
- Generative models are worth studyting because:
  - Making models that perform well tests for our ability to represent and manipulate high-dimensional probability distribution.
  - They can be used in reinforcement learning.
    - They can simulate possible futures while planning.
    - They enable learning in a simulated environment, where mistakes are not costly.
    - They can be used to guide exploration by keeping tracking of previous states and actions.
    - They can be used for inverse reinforcement learning (i.e., given a learned plan, recover the reward function). The paper doesn't say how this is possible.
  - Generative models can be trained with missing data to output predictions on missing data. This can be used in unsupervised learning.
  - Generative models can work with *multi-modal* outputs; i.e., where multiple outputs can be considered correct for a single input. Training models to minimize mean square loss tends to blur valid outputs together, but generative models can just sample them and produce sharp results.
  - A lot of tasks require generating samples for some distribution.
    - Single image super-resolution.
    - Creation of art.
    - Image-to-image translation.

2 A Glossary of Generative Models

- We focus only on models that uses the principle of *maximum likelihood*. GANs does not use this but can be made to do so.
- Let our model be parameterized by parameters θ. Let us also assume that training data $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}$ is given to us. The principle of maximum likelihood dictates that we find θ that maximize the **likelihood** $\prod_{i=1}^{m} p_{\text{model}}(\mathbf{x}^{(i)}; \theta)$. This is equivalent to solving the following problem:

$$\underset{\theta}{\text{argmax}} \sum_{i=1}^{n} \log p_{\text{model}}(\mathbf{x}^{(i)}; \theta).$$

- The principle of maximum likelihood can also be thought of as minimizing the KL-divergence between the data generating distribution and the model:

$$\operatorname*{argmax}_{\theta} D_{\mathrm{KL}}(p_{\mathrm{data}}(\mathbf{x}) \| p_{\mathrm{model}}(\mathbf{x}; \theta)).$$

  In practice, we don't have p_{data}. We only have samples from it. These are used to generate the **empirical distribution** $\hat{p}_{\mathrm{data}}(\mathbf{x})$.

- Generative models that use maximum likelihood can be classified as follows:
  - *Explicit density*
    - *Tractable density*
      - *Fully visible belief nets*: NADE, MADE, PixelRNN
      - *Change of variables models*: nonlinear ICA
    - *Approximate density*
      - *Variational*: variational autoencoders
      - *Markov chain:* Boltzmann machine
  - *Implicit density*
    - *Direct*: GANs
    - *Markov chain*: GSN

2.1 Explicit Density Models

- Explicit models define an explicit density function $p_{\mathrm{model}}(\mathbf{x}, \theta)$.
- It is easy to maximize the model once we have the definition: just do gradient descent. The difficulty is designing a tractable model that captures all the details in the data. You either (1) go ahead and construct such a model or (2) work with a tractable approximation of the likelihood.
- There are two popular approaches to tractable explicit density models.
  - Fully visible belief networks
  - Nonlinear independent component analysis
- **Fully visible belief networks** (FVBNs) decomposes the probability into a chain of conditional probabilities where a component of the data is conditioned on previous components:

$$p_{\mathrm{model}}(\mathbf{x}) = \prod_{i=1}^{n} p_{\mathrm{model}}(x_i | x_1, \ldots, x_{i-1}).$$

  - It takes $O(n)$ time to generate a sample. These $O(n)$ steps cannot be parallelized.
  - WaveNet is such a system [9]. It can generate realistic human speech, but it takes a long time to generate an example though.
- **Nonlinear independent component analysis** defines (1) a space of latent vectors and (2) a complicated nonlinear function that transforms the latent vectors to the real target vectors. The whole distribution is tractable if both the distribution over latent vectors and the nonlinear mapping are tractable.
  - Non-volume preserving transformation (NVP) is a member of this group. It can generate ImageNet images.

  - The main drawback of nonlinear ICA is that it restricts the choice of the mapping. It requires that the latent vectors must have the same dimensionality as the data vectors to make the mapping invertible.

- Explicit density models can be effective, but they impose a lot of restriction on the models.
- Working with approximations of the likelihood can reduce the restrictions on the models. There are two popular approaches:
  - Variational approximation.
  - Markov chain approximation.
- The **variational approximation** approach defines a lower bound

$$\mathcal{L}(x;\theta) \leq \log p_{\text{model}}(x;\theta)$$

  and optimize for $\mathcal{L}(x;\theta)$ instead of the log likelihood. It is possible to define $\mathcal{L}$ that is tractable.
- The **variational autoencoder** (VAE) is currently the most popular variational method to date [4].
  - One problem with VAE is that, when weak models are used to approximate prior or posterior distributions, the gap between $\mathcal{L}$ and the true likelihood would result in p_{model} learning something other than p_{data}.
  - VAE can produce good likelihood but is generally regarded as producing lower quality samples than GAN. However, there is no good objective measure for sample quality.
- The **Markov chain method** generates sample by starting from an initial value x_0 and repeatedly sampling the next value from the conditional propbability distribution $x_{i+1} \sim q(x_{i+1}|x_i)$. The premise is that the later elements of the sequence x_0, x_1, x_2, ... would be distributed according to p_{model}, so an element x_i where i is large would be a sample that we want.
- **Botlzman machines** are a family of Markov chain generative machines that have been widely researched for a long time. They are not popular now because it takes a long time to generate samples because it hard to hell how long we have to sample before we reach a good one. Moreover, they do not scale well to problems such as ImageNet generation.

2.2 Implicit Density Models

- These models do not compute or approximate p_{model} directly.
- The **generative stochastic network** (GSN) uses Markov chain to generate samples [1]. As a result, it doesn't scale well to higher dimentionss and takes a long time generate one sample.
- **Generative Adversarial Networks** (GANs), the subject of this document, can be classified into this category.
- When comparing GANs against other models:
  - GANs can generate all components of a sample in parallel unlike FVBNs.
  - GANs have very few restriction on the model used.
  - GANs does not use Markov chains, so a sample can be generated in one shot.
  - GANs can be shown to be asymptotically consistent. That is, given enough data and model complexity, it approximates the data distribution correctly without bias.
  - Training GANs requires finding a Nash equilibrium of a game, which is harder than minimizing an objective function.

3 GANs

3.1 The Framework

- We set up a game with two players:
  - The **generator** creates samples that are intended to come from the same distribution as the training data.
  - The **discriminator** examines samples to determine whether they actually come from the data distribution or not.
- We evolve the two players together:
  - The discriminator learns to do its job with labeled training data. The data labeled *real* come from the training data. The data labeled *fake* are generated by the generator.
  - The generator is trained to fool the discriminator.

  In the end, the generator should be able to fool the discriminator by generating samples that are very similar to the real samples, and the discriminator should be totally confused.
- Notationally:
  - The discriminator is a function D, with parameter $\theta^{(D)}$ that takes x (i.e., something with the same dimensionality as a training example) as input. It typically outputs the probability that x comes from the real distribution.
  - The generator is a function G, with parameter $\theta^{(G)}$ that takes a latent vector z (sampled from a prior distribution, which is usually a Gaussian) and produces a sample x.
- The discriminator has a cost function $J^{(D)}(\theta^{(D)}, \theta^{(G)})$ that it wants to minimize. However, it can only do so by changing $\theta^{(D)}$. On the other hand, the generator wants to minimize another cost funtion $J^{(G)}(\theta^{(D)}, \theta^{(G)})$, and it can only do so by changing $\theta^{(G)}$.
- The two optimization problems above, when taken together, is a *game* rather than a simple optimization problem.
- The solution to a game is a **Nash equilibrium**: a tuple $(\theta^{(D)}, \theta^{(G)})$ that is a local minimum of $J^{(D)}$ with respect to $\theta^{(D)}$ and also a local minimum of $J^{(G)}$ with respect to $\theta^{(G)}$. (In the original tutorial, it is not mentioned explicitly whether we want a local minimum when the other variable is heled fixed, but it should be so.)
- There are some restrictions on the generator.
  - $G(z; \theta^{(G)})$ should be differentiable with respect to $\theta^{(G)}$. This is because the output will be fed to the discriminator, whose output will be consumed by the cost function. This means that GANs cannot output discrete values.
  - If we want p_{model} to have the full support on x, we need the dimension of z to be as large as the dimension of x.
- Typically, G and D are deep neural networks. (Duh!)
- As said earlier, z is a random vector. It does not have to be vanilla input to the network G. It can be incorporated in the form of additive or multiplicative noise to hidden layers (e.g., dropout).
- The training process.

  - On each step, two minibatches are sampled:
    * A minibatch of x values from the training data.
    * A minibatch of z values from the prior distribution.
  - We feed z value to $G(\cdot;\theta^{(G)})$ to generate a batch of fake x's.
  - Both the real batch and fake batch are used to do an SGD step on D to minimize $J^{(D)}$.
  - The fake batch and its output from D are then used to perform an SGD step to minimize $J^{(G)}$.
- We can use any gradient-based optimization algorithm to perform the two gradient steps above. Goodfellow recommends Adam.
- Some researchers recommend running more steps of one player than the other. However, around the end of 2016, Goodfellow thought doing one step for each player at a time works the best.

3.2 Cost Functions

3.2.1 The Discriminator's Cost

- The cost function for the discriminator is usually:

$$J^{(D)}(\theta^{(D)},\theta^{(G)}) = -\frac{1}{2}E_{x\sim p_{\text{data}}}[\log D(x)] - \frac{1}{2}E_z[\log(1-D(G(z)))]$$

- As mentioned earlier, we train the discriminator with two batches.
  - When training the the real batch, the cost function is reduced to:

$$-\frac{1}{2n}\sum_{i=1}^{n}\log D(x_i).$$

  - When training with the fake batch, the cost function is reduced to:

$$-\frac{1}{2n}\sum_{i=1}^{n}\log(1-D(G(z_i))).$$

- Given enough training data and model capacity, the optimal D is given by:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)}$$

  where $p_{\text{model}}(x) = \Pr(G(z)=x)$.
- When we finish training, the value $1/(1-1/D(x))$ gives an estimate of the ratio $p_{\text{data}}(x)/p_{\text{model}}(x)$. This is, in effect, the approximation that GANs make in order not to deal with p_{model} directly. The estimate is also useful because it enables us to compute a wide variety of divergences and their gradients.

3.2.2 Minimax

- We can require the game to be a *zero-sum game*. That is, the discriminator's gain is the generator's loss. In other words:

$$J^{(G)} = -J^{(D)}.$$

- In this way, the entire game can be summarized with a **value function**. In our case, this is just the discriminator's pay of:

$$V(\theta^{(D)}, \theta^{(G)}) = -J^{(D)}(\theta^{(D)}, \theta^{(G)}).$$

- Finding the optimal parameter for the generator is equivalent to computing:

$$\theta^{(G)*} = \underset{\theta^{(G)}}{\operatorname{argmin}} \max_{\theta^{(D)}} V(\theta^{(D)}, \theta^{(G)}).$$

  This is why this game is called the **minimax game**.

- The minimax game enables theoretical analysis, but the loss function does not perform well in practice.

  The problem is that the discriminator loss function is a cross entropy loss on the correct real/fake class of the sample. The gradient of this function becomes very small when the sample is already classified correctly. Because the generator's cost function is the negative of the discriminator's, there will be no update to the generator if the discriminator has learned to reject the generator's output with high probability.

3.2.3 Heuristic, Non-Saturating Game

- To solve the saturating gradient problem of the minimax game, we let the cost of the generator be:

$$J^G = -\frac{1}{2} E_z[\log D(G(z))].$$

  $\log D(G(z))$ is proportional to the probability that the discriminator thinks a fake example is real. $-logD(G(z))$ is thus proportional to the probability that the discriminator is "right" on a fake example. In effect, through this cost function, the generator tries to minimize the probability that the discriminator is right.

- The advantage of this cost function is that, when the discriminator is very right, the generator can still get non-saturated gradient.

3.2.4 Maximum Likelihood Game

- Using

$$J^{(G)} = -\frac{1}{2} E_z[\exp(\sigma^{-1}(D(G(z))))],$$

  where σ is the logistic sigmoid function, is equivalent to minimizing the KL divergence $D_{KL}(p_{\text{data}}|p_{\text{model}})$, given that the discriminator is optimal.

- Another method for approximating maximum likelihood is given by Nowozin et al. [5].

3.2.5 Comparison of Cost Functions

- The generator receives feedback from the value $D(G(z))$. The cost to the generator is monotonically decreasing in $D(G(z))$. (That is, the more real the discriminator things the fake sample is, the less the cost to the generator.) Different generator cost functions have different derivatives on different parts of the $[0, 1]$ domain of $D(G(z))$.

- For the minimax and the maximum likelihood games, the loss plateaus near $D(G(z)) = 0$, meaning that the gradient is low when the generator performs poorly. Moreover, near $D(G(z)) = 1$, the gradient's absolute value is high. This is bad when we want to improve the generator when it is wrong.

- The non-saturating heuristic is the opposite of the above. It has high gradient near $D(G(z)) = 0$ and low gradient near $D(G(z)) = 1$. The gradient also has lower varience than the other two functions.

3.3 DCGAN

- **Deep convolutional GAN** (DCGAN) is a popular architecture for image-generating GANs [6]. It has the following characteristics (taken from the paper):
  - Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator). That is, use the all convolution network approach [8].
  - Use batch norm in both generator and discriminator; except for the output layer of the generator and the input layer of the discriminator in order to avoid sample oscillation. Two minibatches for the discriminator are normalized separately.
  - Remove all fully connected layers.
  - Use RELU activation in generator for all layers except for the output, which uses tanh.
  - Use LeakyRELU activiation in all discriminator layers.
  - Train with Adam.
- DCGAN is the first architecture to learn to generate high resolution images in a single shot. It also demonstrates that arithmetic on the latent vector z can be interpreted as arithmetic on semantic attributes of the samples.

4 Tips and Tricks

4.1 Train with Labels

- Using labels almost always result in dramatic improvements.
- Denton et al. built class-conditional GANs that generated better samples than GANs that are free to generate from any class [2].
- Salimans et al. found that sample quality improved by training the discrimator to recognize specific classes. Here, the class information is not fed to the generator at all. [7].

4.2 One-Sided Label Smoothing

- When the whole GAN works, the discriminator should estimate the ratio $p_{\text{data}}(x)/(p_{\text{data}} + p_{\text{model}})$. However, deep neural networks are prone to producing highly confident outputs that have too extreme probability. This can be a result of having too large of a logit for the correct class. It is thus better to produce more moderate probabilities in order to encourage smaller logits.
- One way to do this is **one-sided label smoothing** [7]: have the discriminator learns to output the probability of 0.9 instead of 1 when it is given the *real* examples. (Nothing is changed on the *fake* examples.)
- Goodfellow recommends that no smoothing be done to the fake examples [3] because it changes the shape of the optimal discriminator function. This will in turn reinforce incorrect behavior in the generator.

4.3 Virtual Batch Normalization

- When using batch normalization, different minibatches result in fluctuation of normalizing constants. When minibatches are small, these fluctuations can become large enough that they have more effect on the output than the latent vector z.
- Salimans et al. [7] introduced two techniques to combat the above problem.

- **Reference batch normalization** samples a **reference minibatch** once before training and uses the mean and standard deviation of this minibatch to normalize all other minibatches during training. This can, however, cause the networks to overfit to the reference minibatch.
- **Virtual batch normalization** computes the normalization constants from the union of the reference minibatch and the minibatch currently being processed.

4.4 Should one balance G and D?

- Goodfollow believes that the discriminator should be optimal, and there's no need to balance between the generator and the discriminator.
- If the discriminator is too confident, this should be solved by one-sided label smoothing rather than dumbing it down.
- However, training the discriminator for $k > 1$ iterations for every generator iteration has not been shown to result in clear improvement.

References

[1] Guillaume Alain, Yoshua Bengio, Li Yao, Jason Yosinski, Eric Thibodeau-Laufer, Saizheng Zhang, and Pascal Vincent. Gsns : Generative stochastic networks, 2015.

[2] Emily Denton, Soumith Chintala, Arthur Szlam, and Rob Fergus. Deep generative image models using a laplacian pyramid of adversarial networks, 2015.

[3] Ian J. Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *CoRR*, abs/1701.00160, 2016.

[4] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[5] Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization, 2016.

[6] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[7] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[8] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2014.

[9] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.