CS 182/282A: Designing, Visualizing and Understanding Deep Neural Networks **Spring 2019**

Section 8: Fooling Networks and Generative Models

Notes by: Daniel Seita

8.1 Course Logistics

- You have a midterm coming up on Wednesday 4/10. Good luck! It will include the topics we talked about on Monday's lecture (4/8).
- Homework 4 on deep reinforcement learning is out, but don't start that until after the midterm.
- Don't forget the final project! That is coming up as well. We are getting the room and location for the poster session finalized. The actual report will be due sometime later.
- Here is some additional advice for working in a team. Make sure you are using version control. For example, make a private GitHub repository for your team. Then, have each member contribute code in their own branches of the repository, and then keep merging the code into the master branch once it is sufficient for the project, while constantly reviewing and checking everyone's code and contributions. The master branch should be for polished code, while new features go in branches. For communication, Slack is often convenient.

8.2 Fooling Networks with Adversarial Examples

Two classic papers on adversarial examples in Deep Learning were [10] followed by [3], which were among the first to show that Deep Neural Networks reliably mis-classify appropriately perturbed images that look indistinguishable to "normal" images to the human eye.

More formally, image our deep neural network model f_θ is parameterized by θ. Given a data point x, we want to train θ so that our $f_\theta(x)$ produces some desirable value. For example, if we have a classification task, then we'll have a corresponding label y, and we would like $f_\theta(x)$ to have an output layer such that the maximal component corresponds to the class index from y. Adversarial examples, denoted as $\tilde{x}$, can be written as $\tilde{x} = x + \eta$, where η is a small perturbation that causes an imperceptible change to the image, as judged by the naked human eye. Yet, despite the small perturbation, $f_\theta(\tilde{x})$ may behave very different from $f_\theta(x)$, and this has important implications for safety and security reasons if Deep Neural Networks are to be deployed in the real world (e.g., in autonomous driving where x is from a camera sensor).

One of the contributions of [3] was to argue that the *linear* nature of neural networks, and many other machine learning models, makes them vulnerable to adversarial examples.[1] We can gain intuition from looking at a linear model. Consider a weight vector w and an adversarial example $\tilde{x} = x + \eta$, where we additionally enforce the constraint that $\|\eta\|_\infty < \epsilon$. A linear model f_w might be a simple dot product:

$$w^T \tilde{x} = w^T x + w^T \eta. \tag{8.1}$$

[1] This differed from the earlier claim of [10] which suspected that it was the complicated nature of Deep Neural Networks that was the source of their vulnerability. Also, while neural networks are not linear, many components behave in a linear fashion, and even advanced layers such as LSTMs and convolutional layers are designed to behave linear or to use linear arithmetic.

Now, how can we make $w^T\tilde{x}$ sufficiently different from w^Tx? Subject to the $\|\eta\|_\infty < \epsilon$ constraint, we can set $\eta = \epsilon \cdot \text{sign}(w)$. That way, all the components in the dot product of $w^T\eta$ are positive, which forcibly increases the difference between w^Tx and $w^T\tilde{x}$.

Equation 8.1 contains just a simple linear model, but notice that $\|\eta\|_\infty$ does not grow with the dimensionality of the problem, because the L_∞-norm takes the maximum over the absolute value of the components, and that must be ϵ by construction. The change caused by the perturbation, though, grows *linearly* with respect to the dimension n of the problem. Deep Neural Networks are applied on high dimensional problems, meaning that many small changes can add up to make a large change in the output.

Equation 8.1 then motivated the the *Fast Gradient Sign Method (FGSM)* as a cheap and reliable method for *generating* adversarial examples.[2] Letting $J(\theta, x, y)$ denote the cost (or loss) of training the neural network model, the optimal maximum-norm constrained perturbation η based on gradients is

$$\eta = \epsilon \cdot \text{sign}\left(\nabla_x J(\theta, x, y)\right) \tag{8.2}$$

to produce adversarial example $\tilde{x} = x + \eta$. You can alternatively write $\eta = \epsilon \cdot \text{sign}\left(\nabla_x \mathcal{L}(f_\theta(x), y)\right)$ where $\mathcal{L}$ is our loss function, if that notation is easier to digest.

A couple of comments are in order:

1. In [3], the authors set $\epsilon = 0.007$ and were able to fool a GoogLeNet on ImageNet data. That ϵ is so small that it actually corresponds to the magnitude of the *smallest bit of an 8 bit image encoding* after GoogLeNet's conversion to real numbers.
2. In Equation 8.2, the gradient involved is taken with respect to x, and not the parameters θ. Therefore, this does not involve changing weights of a neural network, but adjusting the *data*, similar to what you did for portions of Homework 2 where you optimized an image to "represent" a certain class. Here, our goal is not to generate images that maximally activate the network's nodes, but to generate adversarial examples while keeping changes to the original image as small as possible.
3. It's also possible to derive a similar "fast gradient" adversarial method based on the L_2, rather than L_∞ norm.
4. It's possible to use adversarial networks as part of the training data, in order to make the neural network more robust, such as using the following objective:

   $$\tilde{J}(\theta, x, y) = \alpha J(\theta, x, y) + (1 - \alpha) J\left(\theta, x + \epsilon \cdot \left(\nabla_x J(\theta, x, y)\right)\right) \tag{8.3}$$

   Some promising results were shown in [3], which found that it was better than dropout.
5. The FGSM *requires knowledge of the model* f_θ, in order to get the gradient with respect to x.

The last point suggests a taxonomy of adversarial methods. Two ways of categorizing methods are:

1. **White-box attacks**. The attacker can "look inside" the learning algorithm he/she is trying to fool. The FGSM is a white-box attack method because "looking inside" implies that computing the gradient is possible. There is no precise threshold of information that one must have about the model in order for the attack to be characterized as a "white-box attack." Just view it as when the attacker has more information as compared to a black box attack.

[2] Research papers about adversarial examples are often about either *generating adversarial examples*, or *making neural networks more robust to adversarial examples*. As of mid-2019, it seems safe to say that any researcher who claims to have designed a neural network that is fully resistant to *all* adversarial examples is making an irresponsible claim.

2. **Black-box attacks**. The attacker has no (or little) information about the model he/she is trying to fool. In particular, "looking inside" the learning algorithm is not possible, and this disallows computation of gradients of a loss function. With black boxes, the attacker can still *query* the model, which might involve providing the model data and then seeing the result. From there the attacker might decide to compute *approximate* gradients, if gradients are part of the attack.

As suggested above, attack *methods* may involve *querying* some model. By "querying" we mean providing input x to the model f_θ and obtaining output $f_\theta(x)$. Attacks can be categorized as:

1. **Zero-query attacks**. Those that don't involve querying the model.
2. **Query attacks**. Those which *do* involve querying the model. From the perspective of the attacker, it is better to use few queries, with potentially more queries if their cost is cheap.

There are also two types of adversarial *examples* in the context of classification:

1. **Non-Targeted Adversarial Examples**: Here, the goal is to mislead a classifier to predict any labels *other than* the ground truth. Formally, given image x with ground truth[3] $f_\theta(x) = y$, a *non-targeted* adversarial example can be generated by finding $\tilde{x}$ such that

$$f_\theta(\tilde{x}) \neq y \quad \text{subject to} \quad d(x, \tilde{x}) \leq B. \tag{8.4}$$

   where $d(\cdot, \cdot)$ is an appropriate distance metric with a suitable bound B.

2. **Targeted Adversarial Examples**: Here, the goal is to mislead a classifier to predict a *specific target label*. These are harder to generate, because extra work must be done. Not only should the adversarial example reliably cause the classifier to misclassify, it must misclassify to a specific class. This might require finding common perturbations among many different classifiers. We can formalize it as

$$f_\theta(\tilde{x}) = \tilde{y} \quad \text{subject to} \quad d(x, \tilde{x}) \leq B. \tag{8.5}$$

   where the difference is that the label $\tilde{y} \neq y$ is specified by the attacker.

There is now an enormous amount of research on Deep Learning and adversarial methods. One interesting research direction is to understand how attack methods *transfer* among different machine learning models and datasets. For example, given adversarial examples generated for one model, do other models also misclassify those? If this were true, then an obvious strategy that an attacker could do would be to construct a substitute of the black-box model, and generate adversarial instances against the substitute to attack the black-box system.

The first large-scale study was reported in [6], which studied black-box attacks and transferrability of non-targeted and targeted adversarial examples.[4] The highlight of the results were that (a) non-targeted examples are easier to transfer, which we commented earlier above, and (b) it is possible to make even targeted examples more transferrable by using a novel ensemble-based method (sketched in Figure 8.1). In Lecture 16, we saw a bunch of slides about Clarifai.com. This is a black-box image classification system, which we can continually feed images and get outputs. The system in [6] demonstrated the ability to transfer targeted adversarial examples generated for models trained on ImageNet to the Clarifai.com system, despite an unknown model, training data, and label set.

[3]The formalism here comes from [6], and assumes that $f_\theta(x) = y$ (i.e., that the classifier is correct) so as to make the problem non-trivial. Otherwise, there's less motivation for generating an adversarial example if the network is already wrong.

[4]Part of the motivation of [6] was that, at that time, it was an open question as to how to efficiently find adversarial examples for a black box model.

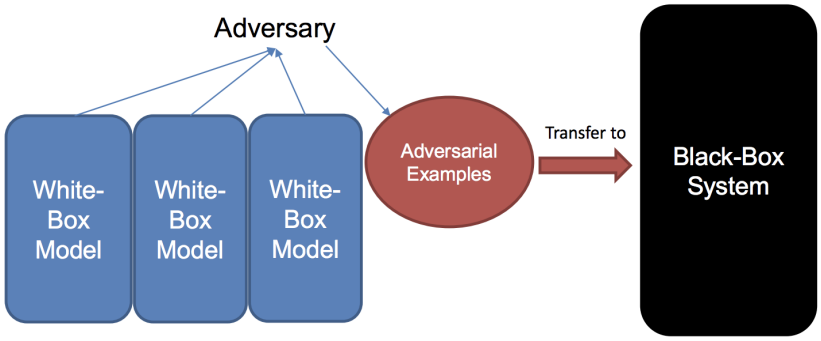

Figure 8.1: A method proposed in [6] for generating adversarial examples using ensembles. (From our Lecture 16 slides.)

8.3 (Deep) Generative Models

As discussed in Lecture 17, Generative models "represent" the full joint distribution $p(\mathbf{x}, \mathbf{y})$ where $\mathbf{x}$ is an input sample and $\mathbf{y}$ is an output or label. While we can use these to compute a joint probability of a sample $(\mathbf{x}, \mathbf{y})$, in Deep Learning we are often interested in *generating new samples* that generalize a given dataset. This may not require labeled data at all (though labels can help), as the goal would be to understand something fundamental about the data distribution.

Three broad classes of deep generative models are PixelRNN/PixelCNN (autoregressive models), Variational Autoencoders (VAEe) and Generative Adversarial Networks (GANs). We will discuss these with a focus on their applications to *images*, though of course these models can be applied on non-image data.[5]

8.3.1 Autoregressive Models

An *autoregressive model* is one that generates one pixel at a time *conditioned* on the pixel values from its prior predictions. For an $n \times n$ image with n^2 pixels in some order $\mathbf{x} = (x_1, \ldots, x_{n^2})$, we define the probability for the image $\mathbf{x}$ as:

$$p(\mathbf{x}) = p(x_1, \ldots, x_{n^2}) = \prod_{i=1}^{n^2} p(x_i \mid x_1, \ldots, x_{i-1}). \tag{8.6}$$

Since the distribution $p(x_i \mid x_1, \ldots, x_{i-1})$ from Equation 8.6 is complex, it makes sense to model it using a neural network f_θ. (We could also have just written Equation 8.6 using p_θ instead of p.) Naively, if we wanted to create a new image, we could define an ordering over the pixels,[6] use our f_θ to sample pixels one at a time, each time conditioning on previously generated pixels. Since this is prohibitively expensive, the work of [7] proposed to use RNNs or CNNs which only examine a few nearby pixels at each step.

More precisely, they propose two model classes, both of which involve starting with a blank image and then generating pixels to it by sampling from a 256-way softmax each time.[7]. The two model classes are:

[5] In this section, we use $\mathbf{x}$ to represent full vectors, i.e., $\mathbf{x} = (x_1, \ldots, x_k)$ for some k.

[6] Normally we would start at one corner and then proceed row-wise (or "diagonally") throughout the 2D spatial map.

[7] This assuming that there are 256 options for a pixel value at the end; there might be some subtleties with multiple channels to form colors but let's not worry about that now.

1. **PixelCNN**: Uses a CNN to model[8] the probability of a pixel given "previous" pixels, which are located in the appropriate nearby spatial region. PixelCNN is slow at *generating* images, because there is a pass through the entire network for each pixel. But it is fast to *train* because there is no recurrence (only a single pass for the image) since the spatial maps are known in advance. During training the convolutions must be causally masked to ignore pixels at the same or later position in the pixel generation order.
2. **PixelRNN**: Uses an RNN (well, actually an LSTM...) instead of a CNN from earlier. The RNN models "remember" the state from more distant pixels via their recurrent state. The PixelRNN uses recurrence instead of the 3x3 convolutions to allow long-range dependencies, and can generate a full row of pixels in one pass. Training is generally longer, though, due to the recurrence involved; each row has its own hidden state in the LSTM layer(s).

See the Lecture 17 slides for visuals on how PixelCNN and PixelRNN generate images.

For training, both models use labels that are automatically supplied by the image, because the "next pixel" (given the prior ones) are part of the data. This is a key technique that is often seen in unsupervised learning tasks: *figuring out a way to turn it into a supervised learning task*.

A more recent autoregressive model would be the *image transformer* [8], which is similar to PixelCNN and PixelRNN, except (you guessed it) it uses a multi-headed attention network.

8.3.2 Variational Autoencoders

Variational Autoencoders (VAEs) [5] are one of the most popular class of generative models. The math can be tricky, though the ideas are easy to explain at a high level.

First, recall what an *autoencoder* means.[9] These are models that, given data $\mathbf{x}$, are trained to predict $\mathbf{x}$ again! That by itself isn't very interesting, but autoencoders that are implemented as neural networks can be designed so that they compress the input into a smaller hidden state represented by $\mathbf{z}$, and then "decompress" it back to the original output size. Then, after training is done, we can *discard* the second part of the network and just use $\mathbf{z}$ as useful features for the original data; it might perhaps then be used as initialization for a different, supervised learning problem. *This is a trick commonly used in unsupervised learning for obtaining concise feature representations which capture factors of variation in the data.*

These two parts of an autoencoder are the *encoder* and *decoder*. (Recall that we used similar terminology in NLP.) A natural question that arises from this is whether we can use the autoencoder framework to *generate new images*. That's what VAEs do. In VAEs, the encoder and decoder do the following:

1. An **encoder** maps a high-dimensional input $\mathbf{x}$ and then outputs the *parameters* of a Gaussian distribution that specify the hidden variable $\mathbf{z}$, i.e., it outputs $\mu_{\mathbf{z}|\mathbf{x}}$ and $\Sigma_{\mathbf{z}|\mathbf{x}}$. We can implement this as a deep neural network, parameterized by ϕ, which computes the probability $q_\phi(\mathbf{z} \mid \mathbf{x})$.
2. A **decoder** maps the latent representation back to a high dimensional reconstruction, which we'll denote as $\hat{\mathbf{x}}$ to distinguish it from the original input $\mathbf{x}$ (to the encoder). Though, as with the encoder, we want to sample, so the decoder will actually output Gaussian parameters $\mu_{\mathbf{x}|\mathbf{z}}$ and $\Sigma_{\mathbf{x}|\mathbf{z}}$, which we can *then* sample from to get $\hat{\mathbf{x}}$. We can implement this as another deep neural network, parameterized by θ, which computes the probability $p_\theta(\mathbf{x} \mid \mathbf{z})$. To generate new images, VAEs sample the latent

[8] The paper [7] "advertises" PixelCNN as a simplified version of PixelRNN which shares the same core components.

[9] The extension to *variational* autoencoders involves using variational approximations and lower bounds when designing and training VAEs, respectively.

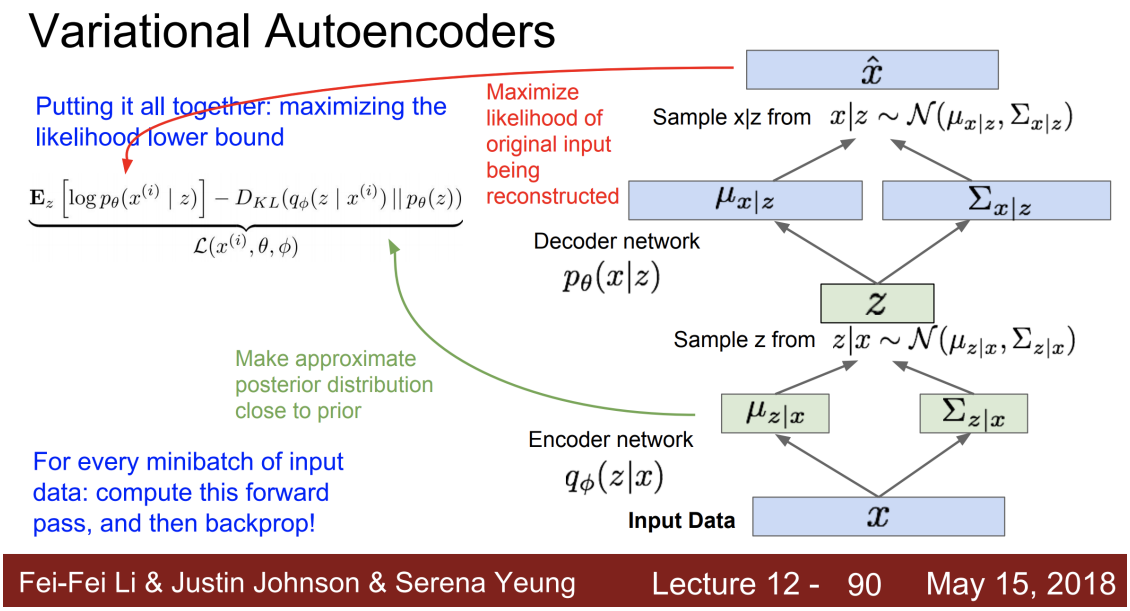

Figure 8.2: The training process for Variational Autoencoders (image from CS 231n slides).

variable **z** and then sample **x** given **z** (using the decoder!). The other sampling process (that of sampling **z**) is easy because VAEs assume it is Gaussian.

This specifies the design and usage of VAEs. But how do we *train* them? Well, we can simply find parameters that maximize the likelihood of the data. However, the likelihood of a single data point **x** from our model assumption is:

$$p_\theta(\mathbf{x}) = p_\theta(x_1, \ldots, x_{n^2}) = \int p_\theta(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z}) d\mathbf{z}. \tag{8.7}$$

Notice the key difference between Equation 8.7 and Equation 8.6. Here, we have an additional *latent* variable **z**, which unfortunately makes the integral intractable. The good news is that with some math (see Lecture 17 slides) you can optimize a tractable lower bound on the data likelihood. See Figure 8.2 for a high-level overview of the training process for VAEs. The forward pass relies on the *reparameterization trick* to make it differentiable. For more about reparameterization tricks, check this blog post.[10]

Overall, vanilla VAE models train efficiently but have some issues with blurriness. For a nice (albeit a bit old in Deep Learning land) overview of VAEs, see [1].

8.3.3 Generative Adversarial Networks

Generative Adversarial Networks (GANs) [4] have been one of the hottest topics in machine learning over the last five years. They are generally regarded as being able to produce higher-quality, less blurry images as compared to VAEs. See Figure 8.3 for an overview of how GANs work. Unlike PixelCNN and PixelRNN, which model an explicit, tractable density (Equation 8.6), and unlike VAEs, which model an explicit but *intractable* density (Equation 8.7), GANs do not explicitly model the density. They model it "implicitly" through a two-player "game" between a generator and a discriminator.

[10] http://blog.shakirm.com/2015/10/machine-learning-trick-of-the-day-4-reparameterisation-tricks/

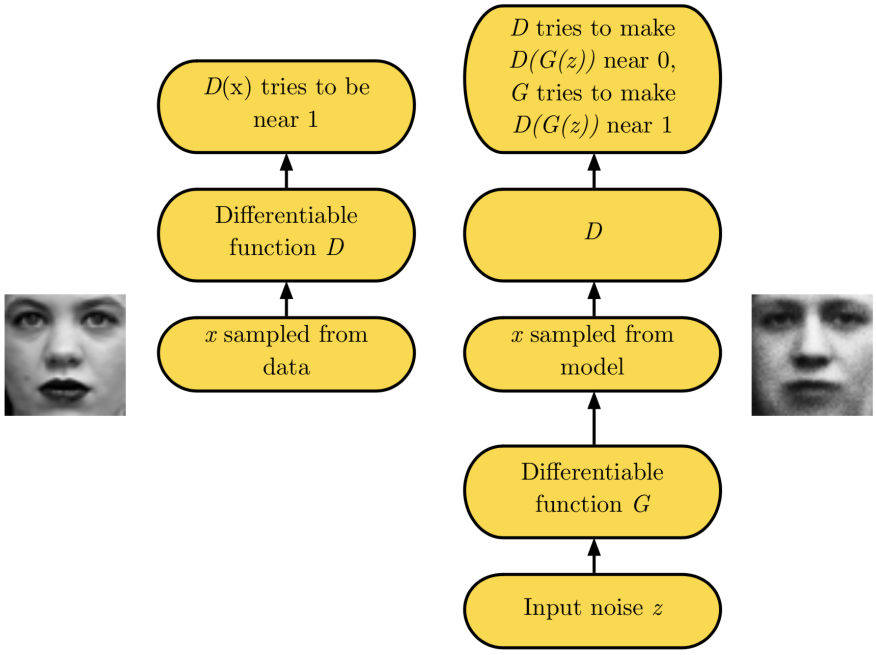

Figure 8.3: The Generative Adversarial Network process [2].

1. A **generator** G tries to fool the discriminator by generating "real-looking" images $\mathbf{x}$ from the target class, given input noise $\mathbf{z}$ (usually Gaussian or from a uniform distribution).
2. A **discriminator** D is trained to classify "fake" samples from the Generator versus samples from the true data. The output of D is in the range $[0, 1]$, representing the probability that input image $\mathbf{x}$ is a *real* image, and not a fake one.

GANs correspond to the following minimax "game:"

$$\min_{\theta_G} \max_{\theta_D} \; V(D,G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} \Big[\log(D(x)) \Big] + \mathbb{E}_{z \sim p_z(z)} \Big[\log(1 - D(G(z))) \Big] \tag{8.8}$$

where we denote θ_G and θ_D as parameters of separate neural networks to be optimized for the generator G and discriminator D functions. The goal of training is to come up with a local Nash equilibrium where the Generator has been trained to fool the Discriminator. When that happens, it means the Generator can now generate fake images that are indistinguishable from images in the training data. Training involves alternating between gradient *ascent* on Equation 8.8 with respect to θ_D, and then gradient *descent* on Equation 8.8 with respect to θ_G. Though, in practice the generator often maximizes $\mathbb{E}_{z \sim p_z(z)} \Big[\log(D(G(z))) \Big]$ instead of minimizing $\mathbb{E}_{z \sim p_z(z)} \Big[\log(1 - D(G(z))) \Big]$ for better gradient signals.

Since the original GAN paper, there have been many improvements to the architecture. One of the most important upgrades was to DCGANs [9], which were the first GANs to generate high-resolution images in a single forward pass. For a nice tutorial, see [2].

References

[1] Carl Doersch. "Tutorial on Variational Autoencoders". In: *arXiv* (2016).

[2] Ian Goodfellow. "NIPS 2016 Tutorial: Generative Adversarial Networks". In: *arXiv* (2016).

[3] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *International Conference on Learning Representations (ICLR)*. 2015.

[4] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Neural Information Processing Systems (NIPS)*. 2014.

[5] Diederik Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *International Conference on Learning Representations (ICLR)*. 2014.

[6] Yanpei Liu et al. "Delving into Transferable Adversarial Examples and Black-box Attacks". In: *International Conference on Learning Representations (ICLR)*. 2017.

[7] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel Recurrent Neural Networks". In: *International Conference on Machine Learning (ICML)*. 2016.

[8] Niki Parmar et al. "Image Transformer". In: *International Conference on Machine Learning (ICML)*. 2018.

[9] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *International Conference on Learning Representations (ICLR)*. 2016.

[10] Christian Szegedy et al. "Intriguing Properties of Neural Networks". In: *International Conference on Learning Representations (ICLR)*. 2014.