Jagiellonian University
Faculty of Mathematics and Computer Science
Department of Theoretical Computer Science

Unsupervised Learning of Disentangled Representations in Progressively Trained Generative Adversarial Networks

Master's Thesis
Computer Science – IT Analyst

Author:
Jonasz Pamuła

Supervisor:
Dr Piotr Micek

Kraków, September 2018

DRAFT

Abstract

This paper aims to investigate the effectiveness of the Mutual Information Penalty in conjunction with progressive training of Generative Adversarial Networks (GANs).

More specifically, we propose an architecture capable of generating high resolution images with isolated semantic features controlled by a set of real valued variables. This architecture is applied to the CelebA-HQ dataset. Selected isolated features of the experiment are summarized in youtu.be/U2okTa0JGZg.

Fig. 1: The direction of the look is a good example of a semantic feature. When generating images, we would like to independently control as many features as possible.

Table of Contents

Abstract **2**

1 Background & Introduction **5**
1.1 Generative Adversarial Networks 5
1.2 Progressive Training of GANs 6
1.3 InfoGAN 7
1.4 The objective of this paper 8

2 Metrics **8**
2.1 Sample quality 9
2.1.1 MSSSIM Diversity 9
2.1.2 Fréchet Inception Distance 9
2.1.3 Inception Score 9
2.2 Control of semantic features 10
2.2.1 InfoGAN Penalty 10
2.2.2 Code Reconstruction Error 10
2.3 Isolation of semantic features 11
2.3.1 MSSSIM Feature Isolation 11
2.3.2 MSE Feature Isolation 11

3 Architecture **11**
3.1 Loss 12
3.2 Generator 12
3.3 Discriminator 14
3.4 Hyperparameters 14
3.4.1 Initialization 14
3.4.2 Channels 14
3.4.3 Dynamic batch size 15
3.4.4 Optimizers 16
3.4.5 Parameters Exponential Moving Average 16

4 Injecting the latent code into Generator **16**
4.1 Variations on the classical InfoGAN approach 18
4.1.1 Vanilla Approach 18
4.1.2 Gradual Loss and Unmasking 21
4.1 Appending 22
4.2 Conditioning 23
4.5 No structured input 27
4.6 Metrics and summary 28

4.6.1 Sample Quality 28
4.6.2 Control of Semantic Features 29
4.6.3 Feature Isolation 29
4.6.4 Summary 30

5 Image drift across phases, consistency loss 30

6 Latent code size 33

7 Summary of the disentangled features 34

8 Technical considerations 34
8.1 TensorFlow and progressive training 34
8.2 tensorflow.contrib.gan 35

9 Potential applications and further work 35
9.1 Unsupervised extraction of high level features 35
9.2 Extending the Neural Photo Editor 35
9.3 Impact of InfoGAN penalty on training stability 36

10 Summary 36

11 Bibliography 37

12 Appendix 38
12.1 Feature summary 38
12.2 Non-curated set of samples 49

1 Background & Introduction

1.1 Generative Adversarial Networks

The framework of Generative Adversarial Networks (*GANs*), first proposed in 2014 by Ian Goodfellow et al. [1], is a technique of modeling the generating distribution of a given dataset. Using this technique one can obtain a *Generator* capable of producing *novel* datapoints that follow the distribution of the original dataset. Most remarkably, the method has proven to work well with extremely high-dimensional data, like images and audio waveforms, for which it is hard to manually define a metric to measure a generator's quality, and thus to guide its optimization process. The GAN framework circumvents this difficulty by simultaneously optimizing two deep neural networks: the Generator itself, and a *Discriminator*. The role of the latter is to distinguish between the images from the dataset and the samples coming from the Generator. By doing so the Discriminator provides a natural loss function that allows us to train the Generator: its objective is now to deceive the Discriminator. In other words, the Generator and the Discriminator are trained in alternating steps, each trying to pull the Discriminator's loss in a different direction.

More formally, the Discriminator's loss is the classical *cross entropy loss* (also called logloss) that is commonly used when training classifiers. We produce labeled datapoints for the Discriminator in the following fashion: first we draw variable y from a symmetric coin toss distribution, resulting in $y = 1$ or $y = 0$. If $y = 1$, we draw an image x from the original dataset, otherwise we draw it from the Generator. The Discriminator's role is to compute a conditional probability distribution $D(x) = P(y = 1 \mid x)$ that minimizes the *cross entropy* between y and $D(x)$. In order to adhere to the convention of representing the objective as a loss that is to be minimized, this cross entropy is usually given with a negative sign:

$$loss_D = E_{y \sim p(y),\, x \sim p(x \mid y)}[-y \cdot \log(D(x)) - (1-y) \cdot \log(1 - D(x))]$$

which is equivalent to:

$$loss_D = E_{x \sim dataset} - \log(D(x)) + E_{x \sim G} - \log(1 - D(x))$$

The Generator's objective, in turn, is to *maximize* the Discriminator's loss. Note that the Generator has no control over the value of the first part of the sum, and so its objective reduces to maximizing the second summand. As the author's note in [1], maximizing $-\log(1-D(x))$ leads to certain training problems related to vanishing gradients, and in practice it is more effective for the Generator to *minimize* $-\log(D(x))$. We thus arrive at what is called the *modified loss* of the Generator:

$$loss_G = E_{x \sim G} - \log(D(x))$$

As already mentioned, and as implied by the GAN's name, the Generator and the Discriminator are usually implemented as (deep) neural networks. The process of optimization is based on *Stochastic Gradient Descent (SGD)*: given a sample of datapoints, the network's weights are modified in a way that reduces the loss for these datapoints. This is achieved by pushing the weights in the direction of the (negative) *gradient*: the loss's derivative with respect to the weights. The gradient approach becomes problematic when we optimize the very network that is used for the sampling of the datapoints, as is the case with the Generator. To address this problem, the Generator is implemented as a differentiable transformation of a real-valued vector z (also called the *input noise)*. Imposing a probability distribution over z (usually isotropic Gaussian) simultaneously induces a distribution over Generator's outputs $G(z)$. This subtle modification, known as *reparameterization trick* [14], allows us to rewrite our loss functions in a way that can be directly optimized using SGD:

$$loss_D = E_{x \sim dataset} - \log(D(x)) + E_{z \sim p(z)} - \log(1 - D(G(z)))$$

$$loss_G = E_{z \sim p(z)} - \log(D(G(z)))$$

We use this formulation of the loss in all experiments throughout the paper.

1.2 Progressive Training of GANs

Since the introduction of GANs much research has been devoted to finding techniques that would make them more stable and reliable [2,5,7]. GANs are notorious for how difficult it is to train them. One common issue is that of *mode collapse*, which is said to happen when the Generator starts producing a single image on each draw. Once it happens, the training rarely recovers.

In the context of images a technique of *progressive training* has been proposed by researchers at Nvidia [2]. The main idea behind it is to train GANs in stages, increasing the resolution of generated images gradually. That is, given a dataset, all the images in it are downscaled to the resolution of 4×4 pixels, and a GAN is trained to model their distribution. Having done that, the original dataset is now downscaled to the resolution of 8×8, and a GAN is trained over them. But rather than doing it from scratch, the 8×8 GAN *builds on top of the* 4×4 *GAN.* The procedure continues in this fashion until the resolution of the original dataset is reached.

The researchers at Nvidia employed this technique on a newly assembled CelebA-HQ dataset to generate one-megapixel images of human faces with unprecedented quality and level of detail.

1.3 InfoGAN

In the usual setting, and when working with images, the Generator starts its computation from a vector of Gaussian noise and transforms it through a series of transposed convolutions (commonly called deconvolutions) into the generated image. Thanks to the continuous nature of all involved operations, a gradual change to the input noise results in a gradual change of the output image, changing the features of the picture. Therefore the input noise is sometimes referred to as *code*, which reflects the intuition that it *encodes* a description of high level features of the image to be obtained.

Characteristic to this encoding is high *entanglement*, which means that no single coordinate of the code is responsible for any given high level feature of the image. When generating faces, it would be desirable to control *isolated semantic features* of the image, like the length of the nose, head orientation and elevation, etc. – each by a single variable. In reality, though, changing the value of a single code coordinate results in an array of seemingly unrelated changes to the image.

A recent technique, called InfoGAN [3], shows how to impose a *disentangled* representation on a part of the input noise (called the the *latent code*, or the *structured input*) via the addition of an auxiliary loss, the *Mutual Information Penalty*. More specifically, the Generator is trained so that mutual information between the latent code and the resulting image is maximized. Note that conceptually the Generator now accepts two inputs: *code*, which represents disentangled semantic features; and *noise*, which serves as the source of randomness for features that are not structured. In order to maximize the mutual information $I(code;\ G(noise,\ code))$ the authors employ a technique known as *Variational Information Maximization* which requires the computation of an estimate Q of posterior probability distribution $P(code \mid G(noise,\ code))$. Interestingly, the Discriminator can be extended to perform this task through the addition of a fully connected layer, that approximates Q, onto one of the Discriminator's deep layers. Thus the Discriminator becomes what is known as a *double-headed* deep network; it outputs the already familiar classification result D, and the probability distribution Q. The loss functions are updated in the following fashion:

$$loss_D = E_{x\sim dataset} - \log(D(x)) + E_{z\sim p(z),c\sim p(c)}[-\log(1 - D(G(c,\ z)))\ -\ \gamma \cdot \log(Q(c \mid G(c,\ z)))]$$

$$loss_G = E_{z\sim p(z),c\sim p(c)}[-\log(D(G(c,\ z))) - \gamma \cdot \log(Q(c \mid G(c,\ z)))]$$

where:

- c stands for *code*
- z stands for *noise*
- γ is a scaling constant
- $Q(c \mid G(c,\ z))$ is the the estimate of the posterior probability

Interestingly, both losses have been extended by subtracting the following (scaled) term:

$$E_{z\sim p(z),c\sim p(x)}[\log(Q(c \mid G(c,\ z)))]\,,$$

which bounds the mutual information $I(c;\ G(c,\ z))$ from below [3]. Thus, while competing in the usual GAN game, the Discriminator and the Generator, at the same time *cooperate* to maximize the mutual information. Note, however, that they do so by controlling different parts of the lower bound: the Discriminator is responsible for improving Q, which makes the lower bound tighter, and the Generator controls the lower bound through adjusting its output image.

1.4 The objective of this paper

The original InfoGAN work [3] maximizes the "mutual information between a *fixed small subset of the GAN's noise variables* and the observations" (italics added). After training, these variables control *salient* features of the generated samples. The authors apply this technique to CelebA dataset [10] (consisting of 200 thousand human faces) resized to resolution of 32×32, and report successful isolation of four features: azimuth (i.e. left / right orientation), presence or absence of glasses, hairstyle, emotion.

In contrast, this paper sets out to find a way of incorporating a much higher number of variables, which would control more detailed features of the generated samples, including: placement of each lip, shape of the jaw, length of the nose, eyelids placement, eyebrows placement and shape, the size of pupils, etc. This is achieved by adopting the InfoGAN technique to the setting of progressive training.

The original work from Nvidia [2] includes an impressive showcase of interpolations between the generated samples [15]. In this paper, in turn, we show how to produce similar interpolations, but with single semantic features isolated.

2 Metrics

When developing an image generator, the ultimate goal is often to be able to produce visually pleasing images. This objective, however, is difficult to quantify objectively. In the case of GANs, the value of the training loss does not reflect the generator's quality. Therefore, the research community has developed a number of heuristics for measuring various aspects of the generated samples, and maximizing these metrics can act as a proxy to our true objective.

For our purpose we will also need a set of metrics to measure the quality of the isolation of semantic features.

The availability of meaningful metrics is critical for an efficient, reliable development process. Below follows a description of a number of metrics used throughout this paper.

2.1 Sample quality

2.1.1 MSSSIM Diversity

Multi Structural Similarity (MSSSIM) [4] is a metric that was first introduced to measure the similarity of two given images. Its initial application was in assessing the quality of lossy compression. In a later work [5] it was adopted to measure the *diversity* of images produced through GANs. This is achieved by measuring the expected similarity of two sampled images. For better intuitiveness, we report the metric as one minus the expected similarity. Throughout this document, we will use the abbreviation *MSSSIM diversity* to refer to this metric. High diversity is a desirable trait of a generator, and it is often difficult to attain diversity on par with that of the original dataset.

2.1.2 Fréchet Inception Distance

Fréchet Inception Distance, introduced in [6], is used to measure statistical similarity of the generated samples, taken in aggregate, to the original dataset. It works by extracting a number of high level features from a number of generated images. This way we can estimate the overall distribution of such high level features for the generated images. Analogous computation is then performed on a sample of images from the original dataset. The final output is the Fréchet distance between the two estimated distributions. In order to calculate the aforementioned high level features, an image is fed into the Inception classifier [9], and the activations of a deep layer are extracted. In practice, the Fréchet Inception Distance seems to reflect well the subjective quality of the produced images.

2.1.3 Inception Score

Inception Score, introduced in [7] to measure the *overall quality* of a generator (including diversity and sample image quality), is a popular metric adopted by many researchers working with GANs. The nature of the CelebA-HQ dataset, however, disqualifies the use of Inception Score as a good measure of model's quality. In its original application, the metric was used for a model working with the ImageNet dataset [8], which is a collection of 1000 categories of images. In our case, when working with just a single class of images, the metric no longer seems to reflect the model's performance.

2.2 Control of semantic features

Both metrics in this subsection measure the success of the InfoGAN technique.

Metrics that are defined per-coordinate, when reported in aggregate (as in "model's MSSSIM feature isolation", "5th block's MSSSIM feature isolation"), are computed by taking the metric's average across all relevant coordinates.

2.2.1 InfoGAN Penalty

The *Mutual Information Penalty* introduced in [3] and used in this paper is based on the computation of an approximate posterior distribution $Q(c \mid G(c, z))$ of the true posterior probability distribution $P(c \mid G(c, z))$. This approximation is the basis for the computation of the lower bound of the mutual information $I(c;\ G(c, z))$:

$$I(c;\ G(c, z)) \geq E_{c \sim p(c),\ z \sim p(z)} \log(Q(c \mid G(c, z))$$

The value of this lower bound will be reported throughout the paper as *InfoGAN penalty*.

During training, the above-mentioned lower bound is maximized in two fashions: by changing the Generator to improve the true value of $I(c;\ G(c, z))$; and by improving the quality of the posterior approximation Q, which makes the estimate of $I(c;\ G(c, z))$ tighter. In this paper, we restrict Q to a family of parameterized isotropic multivariate Gaussian distributions:

$$Q(c \mid G(c, z)) \ = \ N(c;\ \mu = f(G(c, z)),\ \sigma = I)$$

The network that approximates Q (in our case, the Discriminator) does so by computing $\mu = f(G(c, z))$.

2.2.2 Code Reconstruction Error

With Q defined this way the parameter μ can be interpreted as a max-likelihood guess of the latent code that was used to generate the given image. Therefore, for any given coordinate i of the latent code, we will compute the expected absolute difference between the true latent code c_i, and the a posteriori estimated mean, μ_i: $|c_i - \mu_i|$ (expectation is taken across samples). This value will be reported as InfoGAN Code Reconstruction Error. In practice, this metric is easy to interpret: its value tells us how much we err, on average, when we reconstruct the latent code based on the generated image.

2.3 Isolation of semantic features

An important aspect of our model is the degree to which each coordinate of the latent code controls a single isolated semantic feature. High feature isolation is desired as it gives us more control over the produced image.

2.3.1 MSSSIM Feature Isolation

To quantify the degree of isolation we will once again employ the MSSSIM metric. More specifically, for a given latent code coordinate i, we will measure to what extent any given generated image changes when we vary the value of that coordinate and keep the rest of the generator's input unchanged. In our implementation we compare the image produced at the mean value of the coordinate of interest, μ_i, with images produced with coordinate set to $\mu_i + k \cdot \sigma_i$ for $k \in \{-3, -2, -1, 1, 2, 3\}$, where σ_i is the coordinate's standard deviation. We will refer to that metric as MSSSIM feature isolation.

It is important to be careful when interpreting this metrc. It quantifies to what extent the image stays unchanged when varying the value of a certain coordinate in the code. The highest possible value, 1., would mean that *nothing* changes, which is obviously undesirable. In general, high values of MSSSIM Feature Isolation are good as long as InfoGAN penalty and Code Prediction Error do not suffer - which means that the feature is highly isolated, but still salient enough to reconstruct the latent code.

2.3.2 MSE Feature Isolation

For a more complete picture, we will also report MSE feature isolation, which is computed in analogous fashion, but using Negative Mean Squared Error to compare two images.

3 Architecture

The general GAN architecture in our work follows the one described in *Progressive Growing of GANs for Improved Quality, Stability and Variation* [2], with several modifications. The modifications are motivated by: 1) the added functionality, and 2) computational resources constraints. The remainder of this section provides a general overview of the architecture. Further sections will describe certain important important parts of the architecture in more detail. We invite the reader interested in more details to read the source code at https://github.com/jonasz/progressive_infogan.

3.1 Loss

Nvidia uses a flavor of GANs known as WGAN-GP, i.e. the Wasserstein Loss [11] with gradient penalty [12] is used for training. We depart from this design choice and use the standard *modified loss* [1] instead. In our experiments, when used in combination with *mutual information penalty*, the modified loss did not cause any issues that WGAN-GP is known to alleviate (like saturating gradients, mode collapse). On the other hand, it allowed us to perform experiments more rapidly, as there is no need to calculate the gradient penalty; and using on the order of $400k$ real images per phase proved sufficient to obtain satisfactory image quality when experimenting (as compared to $1.6M$ images used in [2]). In our experiments described in Section 4, to ensure convergence, we use $384k$ images in all phases but last, which is trained for twice that amount.

3.2 Generator

The training proceeds in *phases*. In *phase d* the generator is extended with *block d* and is henceforth (technically) capable of producing images with resolution 2^d. For the sake of progressive training, the output of each block is twofold: the image itself, and an activation map that will serve as the input to *block d+1* (see Fig. 2).

Fig. 2: Conceptual overview of block architecture of the generator. Arrows represent flow of tensors, with tensor shapes given in parentheses.

Each block consists of a series of operations that transform the input tensor: see Fig. 3.

	Operation	Shape (if changed)	Description
1.	input	2^{d-1}, 2^{d-1}, $channels_{d-1}$	The output of the previous block.
2.	upsample	2^d, 2^d, $channels_{d-1}$	Increase the resolution of the activation map by duplicating each value across first two dimensions of the tensor.
3.	conv	2^d, 2^d, $channels_d$	A convolution with kernel size 3, stride 1, padding 1.
4.	batch norm		Classical conv-style batch norm [20],

			normalizing each channel across batch and across spatial locations within tensor.
5.	leaky ReLU		Pointwise activation function.
6.	conv		Same as 3.
7.	batch norm		Same as 4.
8.	leaky ReLU		Same as 5. The output of this operation serves as the input to the next block
9.	conv	2^d, 2^d, 3	A convolution with kernel size 1. It's objective is to reduce the number of channels to 3, which form the RGB image.
10.	tanh		Pointwise tanh. Its role is to ensure all output values lie in [-1. , 1.]. The output of this operation is returned as the RGB image produced by *block d*.

Fig. 3: The summary of operations inside of block d (for d >= 3). The specific number of channels per each block is reported in section 3.3 Hyperparameters.

Naturally, the operations within the initial block are distinct. The computation starts from the input noise, performs a fully connected transition into a vector of shape $4 \times 4 \times channels_d$, and a single convolution. We omit the details here for brevity, and refer the interested reader to the source code: https://github.com/jonasz/progressive_infogan.

Note that the architecture of a single block departs from that described in [2]. Our experiments supported the choice of batch normalization [11] over pixel normalization. This difference is likely to be caused by the different choice of the loss function.

As can be seen in Fig. 2, a generator of depth d will produce a series of $d-1$ images with gradually growing resolutions. At the end of the training, the last image is taken as the generator's output. However, during the training, at the outset of *phase d*, the newly produced image with resolution 2^d is not immediately used as output for the loss function. Instead, we use a linear combination of the 2^{d-1} image with a $2\times$ upscaled output of the previous block:

$$image = (1-\alpha) \cdot upscale_2(image_{d-1}) + \alpha \cdot image_d$$

This linear combination initially puts all weight on the old image, and as the phase proceeds, it gradually shifts α in order to fade in the higher resolution image: the α parameter grows linearly from 0 to 1. This smooth transitioning from generating lower-resolution images to generating higher-resolution images is the main principle behind the progressive training introduced in [2].

3.3 Discriminator

The structure of the Discriminator is, for the most part, a mirror image of the Generator. In *phase d* the discriminator is extended with *block d* and is now able to accept images with resolution $2^d \times 2^d$ as input. For the sake of progressive training, each block can also accept input from the successive block, once it is added (Fig 4.).

Fig. 4: Conceptual overview of the block architecture of the Discriminator.

The way these two inputs are combined is analogous to the mechanism used by Generator: a linear combination of the inputs gradually shifts the block's attention from the image to the output of the next block.

The architecture of a single block is very similar to that of the Generator. First, the input image is projected linearly to match the dimensions of the input activations. These two inputs, after being combined with the already familiar linear combination, are fed through two convolutional blocks, and then their spatial dimensionality is reduced for the next block by a mean pooling operation. There are two notable differences in the Discriminator's block when compared to that of the Generator: first, we use no batch normalization; and second, the convolution operations within the block may use a different number of channels (See Section 3.4.2.).

3.4 Hyperparameters

3.4.1 Initialization

We initialize the network's parameters using the *Equalized Learning Rate* technique from [2].

3.4.2 Channels

The primary architectural component of both Generator and Discriminator is a convolution operation. One of its key parameters is the number of channels it outputs. A high number of channels is desired in order to increase the model's capacity, but it also implies an increased cost in computation and memory. Here, we follow the conventional practice of using a number of channels inversely proportional to the spatial resolution of the activation map, but depart

from the exact numbers used in [2], see Fig. 5. More specifically, the number of channels in Generator has been capped to 480 to account for increased memory usage coming from InfoGAN components.

phase	spatial resolution (output for Generator, input for Discriminator)	output channels, Generator	output channels, Discriminator
2	4×4	dense: 480 conv: 480	conv: 512 dense: 512
3	8×8	conv1: 480 conv2: 480	conv1: 512 conv2: 512
4	16×16	conv1: 480 conv2: 480	conv1: 512 conv2: 512
5	32×32	conv1: 256 conv2: 256	conv1: 512 conv2: 512
6	64×64	conv1: 128 conv2: 128	conv1: 256 conv2: 512
7	128×128	conv1: 64 conv2: 64	conv1: 128 conv2: 256
8	256×256	conv1: 32 conv2: 32	conv1: 64 conv2: 128
9	512×512	conv1: 16 conv2: 16	conv1: 32 conv2: 64

Fig. 5: the number of channels after each operation.

3.4.3 Dynamic batch size

The choice of the batch size is driven by:

a) The desire to utilize GPU's parallel cores in full. We would like to achieve highest possible throughput as measured by images processed per second.
b) GPU's memory constraints.
c) Current model's depth. As we proceed into higher phases, the memory needed to process a fixed batch size grows – the network is deeper, and we are working with higher resolution images.

Therefore, we need to start with batch size big enough to satisfy a). We then gradually reduce the batch size so we do not violate b). See Fig. 6 for details.

Phase	Batch size
2	128
3	128
4	64
5	32
6	16
7	6
8	3
9	2

Fig. 6: Batch size adjusted to each phase of training.

3.4.4 Optimizers

We retain the optimizers used in [2]. Both Generator and Discriminator are optimized using Adam optimizer [16], with exponential decay rates set to 0 for mean estimate, and .99 for variance estimate. However, we chose to decrease the learning rate from 0.001 in [2] to 0.0005.

3.4.5 Parameters Exponential Moving Average

For the calculation of metrics and for visualisation purposes we use the parameters that are an exponential moving average across historical training steps. We use the momentum of 0.999. It is a standard technique, widely known to increase the performance of deep neural models, and is used in [2] as well.

4 Injecting the latent code into Generator

In a standard InfoGAN [3], the latent code (or structured input) is provided to the generator as input at the very beginning of the computation flow, alongside the noise. In our attempt to gain better control over isolated features of the output image we will leverage the progressive nature of our architecture. Namely, we would like to be able to designate the level of detail controlled by each variable in the latent code. To this end, we will divide the latent code into parts, each part corresponding to a certain training phase, and consequently its associated

block and resolution. We would like to ensure that each variable controls a feature only visible starting at the associated resolution (see Fig. 7).

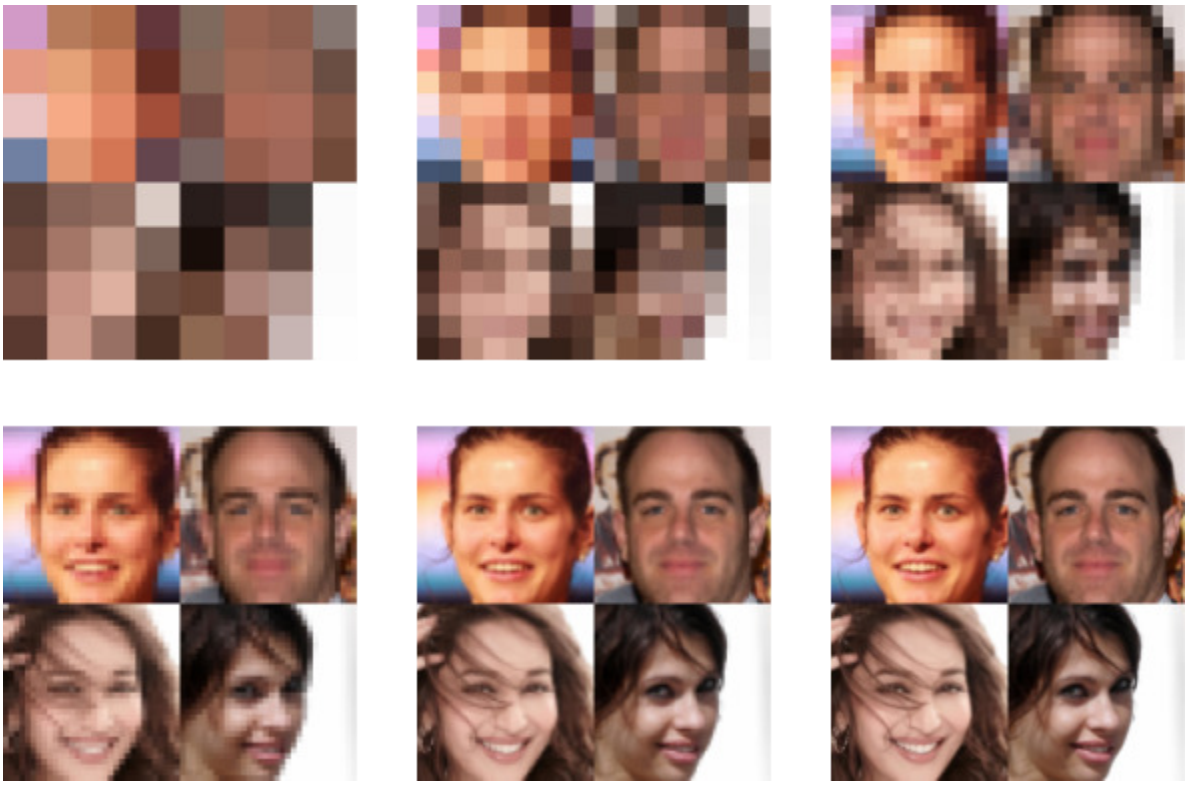


Fig. 7: Four faces from CelebA-HQ rendered at resolutions increasing from 4x4 to 128x128. Certain features, like skin color, are already visible at the lowest resolution. As the resolution increases, more features appear gradually: head orientation and elevation, hairstyle, presence of a smile, face oval, length of the nose, direction of the look, shape and placement of eyebrows, color of the eyes, hair texture, etc.

Let $code_d$ designate the part of latent code associated with *phase d*. We will experiment with a variety of different architectures that try to employ $code_d$ to control details only visible at resolutions 2^d and higher.

All our experiments share the same architecture and hyperparameters, unless stated otherwise.

We work with 16 structured variables per each block from block 2 to block 6, which yields 80 structured variables in total.

The following sections describe the tested architectures and their performance. Subsequently we present the obtained metric values, and summarize the experiments.

4.1 Variations on the classical InfoGAN approach

4.1.1 Vanilla Approach

We start off with applying the vanilla InfoGAN technique to a progressively trained GAN. That is, we ignore the division of the latent code into parts, and feed all 80 variables to the initial block of the model. We add the auxiliary Mutual Information Penalty to our loss. We will refer to this approach as *exp_vanilla*.

As can be seen from the metrics (Fig. 23), the feature isolation of this approach is low, and this is also visible in the images (Fig. 8, Fig. 9).

Fig. 8: Interpolating coordinate no. 57 of the generator in exp_vanilla experiment. In general, the variable controls smile. One can see, however, that other changes are present too: face oval, lightning, lip colour, facial hair, etc.

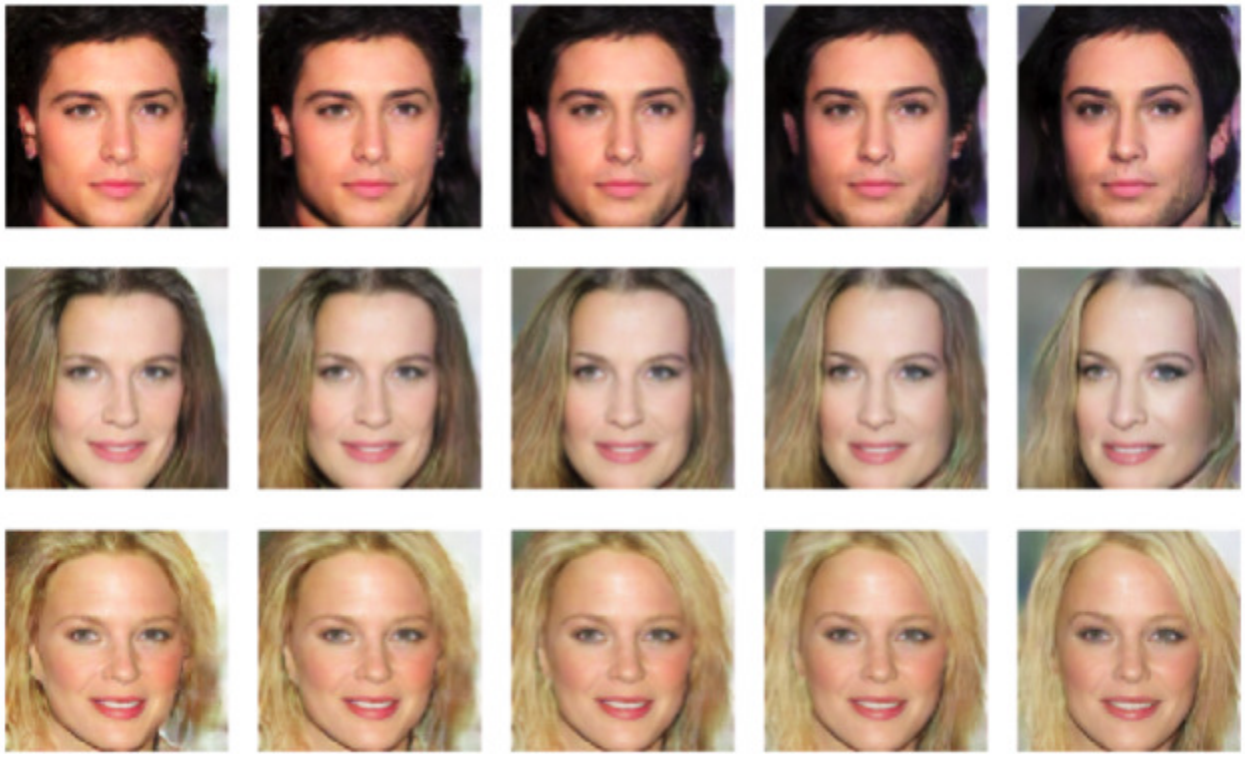


Fig. 9: coordinate 67 of exp_vanilla controls head rotation. One can see that other features of the face are changed as well when varying the coordinate's value: including hairstyle, facial hair, color saturation, and oval of the face.

In fact, most of the time, it is hard to pinpoint the exact single feature controlled by any given variable. Many coordinates in the structured code control features that are accompanied by additional changes to head rotation, oval shape, hairstyle. To aid the detection of the *primary* feature controlled by each variable, we use *sample averaging*: we draw a number of random samples from the generator, with one restriction – the chosen coordinate in the latent code is set to a certain value. This set of images is then averaged in a pixel-wise arithmetic fashion into a single image. When done for two extreme values of a chosen coordinate, the primary features controlled by the coordinate emerge, as features not systematically controlled by the variable average out (see Fig. 10).

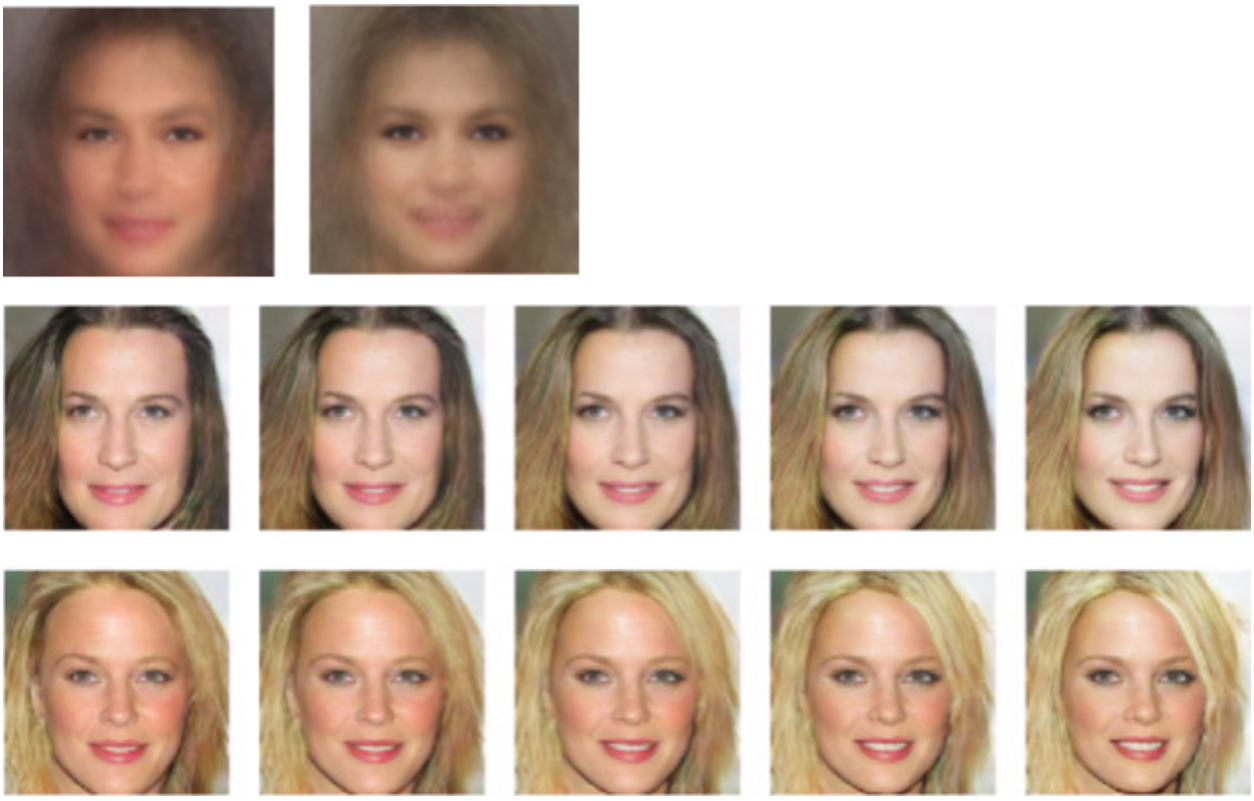


Fig. 10: Using the image-averaging technique can aid the identification of the features controlled by a single coordinate. Here we can see its result for coordinate 65 of exp_vanilla (upper row). In two bottom rows we can confirm that the eye contour intensity does change, along with other features: smile and color balance among others.

Overall, the vanilla approach succeeds at controlling lowly isolated features, like head inclination, horizontal angle, emotion, sex. This result is consistent with the features reported in the original InfoGAN paper. On the other hand, it fails to control subtle, highly isolated features, like shape of the nose, eyebrows position, texture of the hair. It is common for a single coordinate to control multiple features: change to the shape of the nose is accompanied by a change to head rotation, face oval, hairstyle, color balance (see Fig. 11).

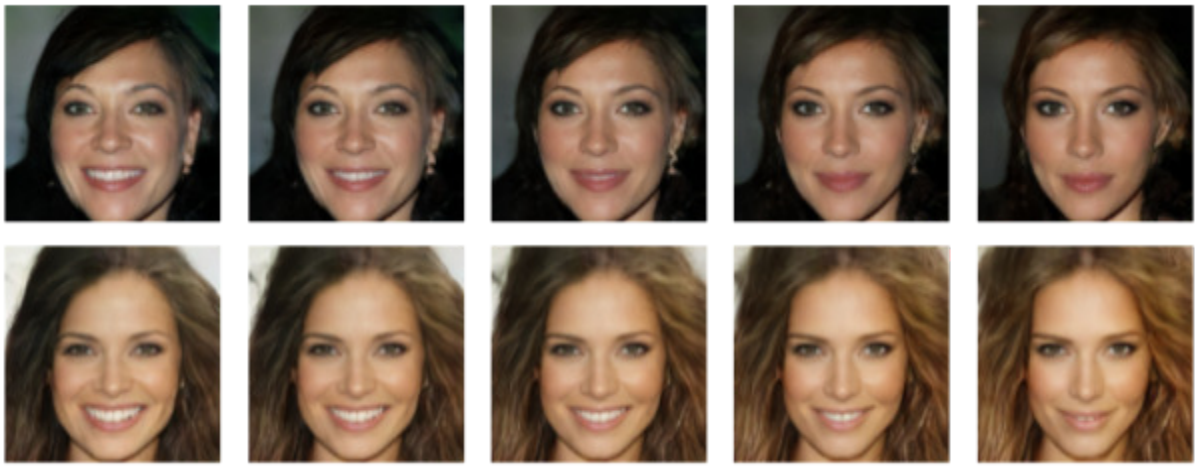


Fig. 11: Coordinate 48 in exp_vanilla seems to control the shape of the tip of the nose. At the same time, however, it also controls the hairstyle, face oval, and smile. It is typical of variables in the structured code in exp_vanilla to exhibit such grouping of features, and therefore low isolation.

4.1.2 Gradual Loss and Unmasking

We extend the *exp_vanilla* experiment with two techniques in order to encourage the partitioning of the latent code into parts responsible for increasing level of detail.

In the first approach, *exp_gradual_loss*, we activate the mutual information penalty for variables in $code_d$ only in phase d and onwards. Up until that phase, $code_d$ is technically indistinguishable from the noise that is also fed to the network.

In the second approach, *exp_unmask*, we also hide the value of $code_d$ from the network in all phases prior to phase d. This is achieved by multiplying the code by a *mask* consisting of zeros, that are gradually replaced with ones at the beginning of subsequent phases. This way, unlike in previous experiments, at the outset of phase d, $code_d$ is not associated with any (potentially entangled) representation of the image. Therefore, the network can start afresh in imposing representation on it. As measured by our metrics, *exp_unmask and exp_gradual_loss* are roughly on par, and superior to *exp_vanilla* in terms of feature isolation. Upon manual inspection, one can see that higher-block coordinates of the latent code succeed in isolating certain salient features (see Fig. 12 for an example).

Fig. 12: Coordinate 79 of exp_unmask has successfully isolated the size of the jaw.

4.1 Appending

In all our further experiments we move on to feeding the latent code into the network in a progressive fashion. That is, $code_d$ becomes the input of the network only in $phase_d$. We also preserve this dynamic approach in the architecture. In Generator, we feed $code_d$ directly to block d, completely omitting previous blocks. To preserve the symmetry of the architecture, now it is block d of the Discriminator that is extended to approximate the posterior probability distribution $Q(code_d \mid G(noise, code))$.

The question that remains is how exactly to feed $code_d$ to Generator's *block d*. We start with the most straightforward approach - appending it to the block's input.

In the simplest version of this technique, we expand the code from shape $[len(code_d)]$ to $[len(code_d), 2^{d-1}, 2^{d-1}]$ by tiling (that is, concatenating $2^{2(d-1)}$ copies of $code_d$ along two dimensions). We refer to this approach as *exp_append_simple.*

A slightly more advanced approach is to first project $code_d$ into a higher dimension through a linear transformation, and only then to append it to the input. We refer to this approach as *exp_append_channels*.

The subjective assessment of techniques based on appending is that higher-phase parts of the latent code focus on color-related characteristics of the image, while failing to control any structural features (see Fig. 13).

Fig. 13: Features isolated by exp_append_channels in phases 5 and 6 are almost entirely color-based, and not structural. This explains why exp_append_channels attains such high MSSSIM Feature Isolation.

4.2 Conditioning

Conditioning is a technique introduced in the WaveNet paper [13], where it was used to control the identity of the speaker of the generated audio wave. In order to *condition* a tensor *t* with another tensor *h*, the authors first perform a trainable linear transformation of *h*, and then add it to *t* in a pointwise fashion. A single tensor can be used to condition activations in the entire network.

We employ this technique by conditioning the input and all activations within *block d* in the Generator with $code_d$. We will refer to this approach as *exp_condition*.

As evinced by the metrics, this approach is superior to all previous ones (except for appending-based techniques, which we will explain later) in terms of isolation it achieves, especially for latent variables associated with blocks 5 and 6. The isolation is impressive upon visual inspection. The model has controlled to control multiple features, some of which are summarized in Fig. 14.

Phase	Isolated Feature
6	Size and color of the irises.
	Shape of the eyes and eyelids.
	Amount of wrinkles on the face.
5	The direction of the look.
	Shape / length of the tip of the nose.
	Upper lip placement.
	Placement and shape of the eyebrows.
4	Head forward / backward inclination.

	Smile with teeth, with no teeth.
	Size of the lower jaw.
3	Various aspects of face oval.
	Various aspects of hairstyle.
2	Left / right rotation of the head.
	Hair color.

Fig. 14: Selected features isolated by exp_condition.

We include a selection of isolated features in the figures below.

Fig. 15: One of the coordinates in exp_condition controls how wide the eyes are open. One can notice the accompanying side effects relating to color saturation. These side effects, however, being color-based and not structural, are much less significant than those seen in exp_vanilla and exp_unmask.

The improvement in feature isolation, as also reflected in the metrics, is the bigger the higher the phase corresponding to the given variable. Thus, in features controlled by variables corresponding to lower blocks, we can still see side effects that we have seen in previous experiments. Examples of such features include head rotation (Fig 16) and gender (Fig 17).

Fig. 16: Coordinates corresponding to lower blocks still exhibit behavior akin to that in exp_vanilla experiment. Here we can see that the variable responsible for head rotation (corresponding to block 3), also influences the hairstyle.

Fig. 17: As we progress to variables in higher and higher blocks, we see a decreasing amount of side effects, and thus increased isolation. Here, a block 5 variable is shown to control head inclination.

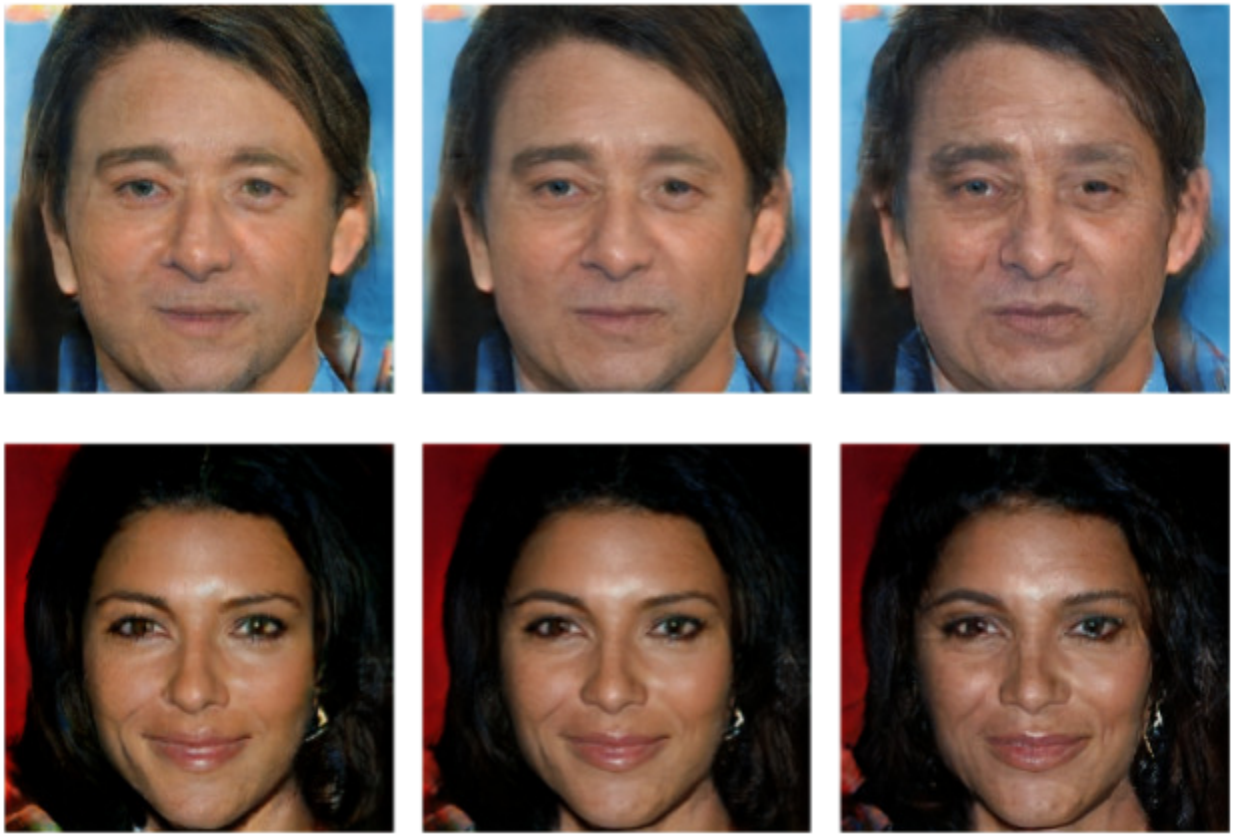


Fig. 18: A block-6 coordinate of exp_condition: when trained to higher resolutions, exp_condition can be seen to control the amount of wrinkles on the face, thus controlling the perceived age. The feature is very well isolated.

As much as the isolation within higher blocks is improved when compared the previous experiments, it is still not perfect. For a human observer, a certain group of features, when varied simultaneously, represent a single high level concept (e.g. changing the wrinkles around eyes, shape of cheeks and lips, and amount of teeth visible can be interpreted as "adding a smile"). At other times it is not so, and changing the shape and the color of the eyes together with the hue of the hair and the lightning conditions doesn't seem to be an "isolated feature". Our model still seems to exhibit such side effects accompanying the control of certain highly-detailed features. Most often these side effects include changes to skin texture and to overall color balance (See Fig. 19). We will try to address by adding a *consistency loss* in section 5.

Fig. 19: coordinate 72 (block 6) of exp_condition controls lower eyelid, thus creating the effect of smiling eyes. It also exhibits particularly strong side effect of changed color saturation.

4.5 No structured input

For comparison we also ran an experiment, *exp_noinfo*, with no structured input fed to the model – that is, no part of Generator's input is subject to Mutual Information Penalty. Interestingly, *exp_noinfo* reliably runs into a mode collapse in early phases of training, from which it does not recover. As a quick remedy, we also try two additional *exp_noinfo* experiments with lower learning rates (0.00025 and 0.000125, down from 0.0005) - but this does not alleviate the issue.

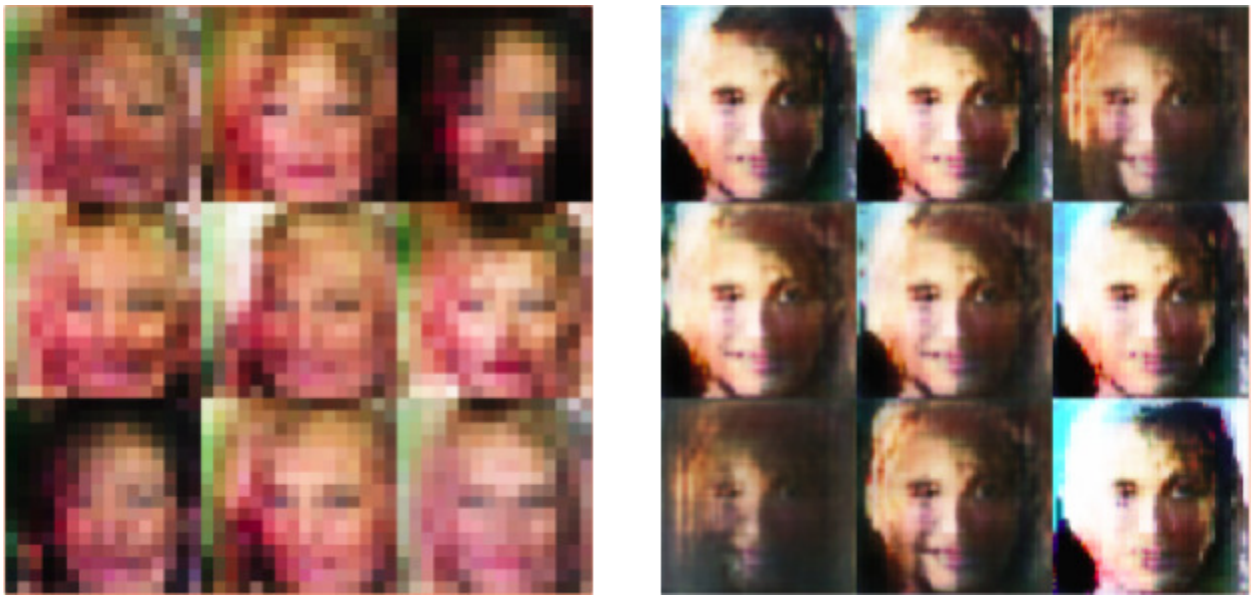


Fig. 20: A sample of nine pictures from exp_noinfo: during phase 4 of training (left), and during phase 6 (right). It is easy to notice poor diversity and quality of the samples.

Intuitively, it is easy to surmise why the inclusion of InfoGAN loss would act against mode collapse, thus stabilising training. Namely, it forces the generated images to be diverse enough to allow the Discriminator to reconstruct the initial latent code. Or, in other words, for InfoGAN loss to be low, two distinct latent codes cannot be translated into images too similar to each other.

Investigating this proposition further is interesting in itself, as it could potentially contribute a method of stabilising GAN training without resorting to batch-level techniques (see e.g. Minibatch Discriminator technique in [7]). This, however, lies outside of the scope of this paper. See section Further Work.

4.6 Metrics and summary

4.6.1 Sample Quality

Experiment	Frechet Inception Distance	MSSSIM Diversity
exp_vanilla	16.9881	0.6615
exp_gradual_loss	17.5554	0.6673
exp_unmask	17.1377	0.6854
exp_append_simple	17.6121	0.6757
exp_append_channels	17.8219	0.6719
exp_condition	22.6782	0.6725

Fig. 21: Quality metrics for all experiments after phase 7 of training.

Fréchet Inception Distance indicates that *exp_condition* attains lower image quality when compared to the rest of the experiments, which are roughly on par with each other. Upon visual inspection, all experiments' images are high quality and visually pleasing. See the Appendix for a visual comparison of a random sample of images from the best performing *exp_vanilla*, and from *exp_condition.*

In terms of diversity, there is little variance across experiments described so far.

4.6.2 Control of Semantic Features

Experiment	Code Reconstruction Error	Code Reconstruction Error Block 5	Code Reconstruction Error Block 6	InfoGAN Penalty Overall	InfoGAN Penalty Block 5	InfoGAN Penalty Block 6
exp_vanilla	0.4327	N/A	N/A	1.0678	N/A	N/A
exp_gradual_loss	0.4296	0.4497	0.4822	1.0681	1.0809	1.1043
exp_unmask	0.4351	0.4514	0.4653	1.07	1.0813	1.0913
exp_append_simple	0.4363	0.4653	0.4456	1.0749	1.0952	1.0811
exp_append_channels	0.4613	0.5014	0.4842	1.0919	1.1217	1.1108
exp_condition	0.3035	0.2266	0.1432	1.0027	0.9604	0.9357

Fig. 22: Metrics related to the control of semantic features for all experiments after phase 7 of training.

Both InfoGAN Penalty and InfoGAN Code Prediction Error clearly indicate that *exp_condition* is the most successful in translating the latent code into features of the image, and then reconstructing it back. As the latter metric reports, the model is able to recover the values of the latent code with error slightly above 0.3. For the parts of the latent code corresponding to the highest phase, the error is below 0.15.

4.6.3 Feature Isolation

Experiment	MSSSIM Isolation Overall	MSSSIM Isolation Block 5	MSSSIM Isolation Block 6	MSE Isolation Overall	MSE Isolation Block 5	MSE Isolation Block 6
exp_vanilla	0.8771	N/A	N/A	-0.0361	N/A	N/A
exp_gradual_loss	0.8897	0.9312	0.9553	-0.034	-0.0174	-0.0101
exp_unmask	0.8849	0.9276	0.9506	-0.0343	-0.0188	-0.0112
exp_append_simple	0.9284	0.9985	0.9996	-0.0276	-0.0004	-0.0001
exp_append_channels	0.9138	0.9956	0.9985	-0.0345	-0.002	-0.0006
exp_condition	0.9303	0.9529	0.9665	-0.0259	-0.0074	-0.0046

Fig. 23: Metrics related to the feature isolation for all experiments after phase 7 of training.

As we can see from the metrics (Fig. 23), *exp_condition* attains the highest overall feature isolation, as supported by both MSSSIM Feature Isolation and MSE Feature Isolation.

The feature isolation for higher-phase parts of the latent code is the highest for appending-based experiments, which attain near-perfect isolation. Of course, as described in

prior sections, this result is explained by the trivial nature of the isolated features, which are almost entirely color based.

4.6.4 Summary

Overall, *exp_condition* is a clear leader in controlling and isolating semantic features of the image. This comes at the cost of higher Fréchet Inception Distance. However, the quality of the produced images is still pleasing. Further sections will focus on improving various aspects of this experiment.

5 Image drift across phases, consistency loss

By injecting $code_d$ directly into $block_d$ we hope to ensure variables in $code_d$ only affect features that start to emerge at resolution 2^d. However, this criterion is not enforced in any way. Once the training progresses well into $phase_d$, there is no reason why the model should not choose to completely remodel the samples it generates based on $code_d$. In fact, during the training of a progressive GAN, one can observe that the images produced by blocks $d-1$ and d are exactly the same at the beginning of phase *d* (due to α being close to 0, see section 3.2), but then start to diverge from each other in terms of colors. Images produced by blocks $d-2$ and earlier are now completely defunct – they don't take part in the computation of the loss – and, in effect, they diverge from the current-block image even further (see Fig. 24).

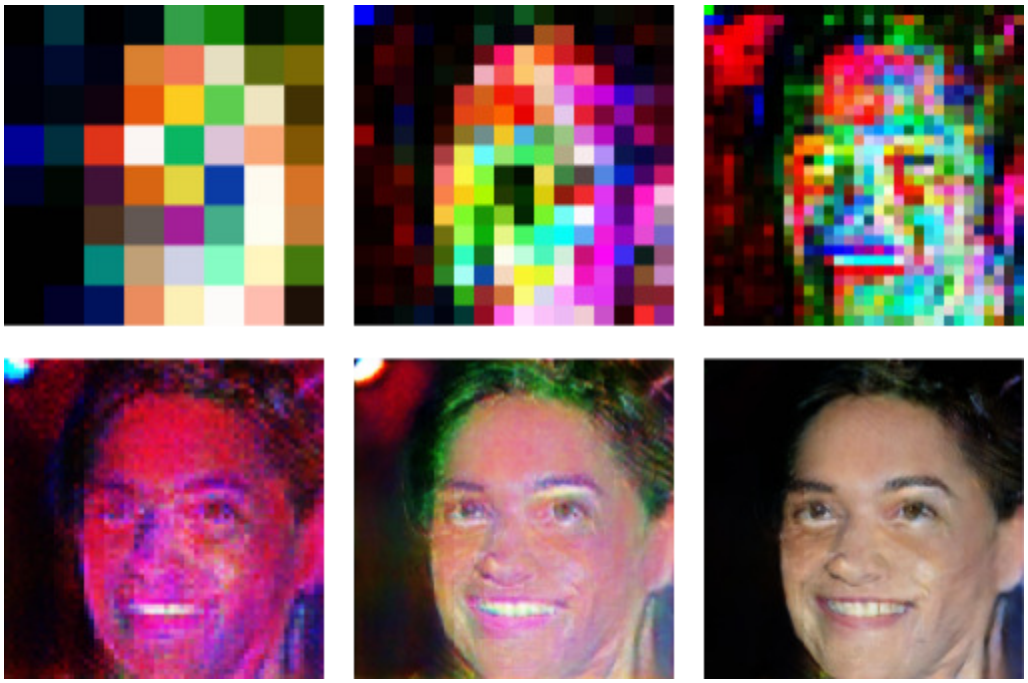


Fig. 24: As training progresses to higher and higher phases, the lower-resolution outputs of previous blocks, no longer contributing to the final loss, drift away from the highest-resolution image.

When training a progressive GAN, this drifting away of the no-longer-active image outputs of the network is irrelevant - it is only the final image output that matters. For the purpose of ensuring high feature isolation, however, the availability of the lower-resolution versions of the generated image would be useful. More specifically, by ensuring that the downscaling of $image_d$ by factor of 2 results in $image_{d-1}$ notwithstanding the value of $code_d$, we could enforce our initial design of $code_d$ being responsible for features only visible at resolution 2^d and higher.

In practice, both objectives (eliminating the low-resolution image drift, and isolating the impact of $code_d$) can be achieved by the introduction of an auxiliary *consistency loss* for the Generator:

$$loss_{consistency} = \sum_{d=3}^{max\ res} MSE(downscale_2(image_d),\ image_{d-1})$$

The *Mean Squared Error (MSE)* used in the auxiliary loss is the average of squared differences of corresponding pixel values.

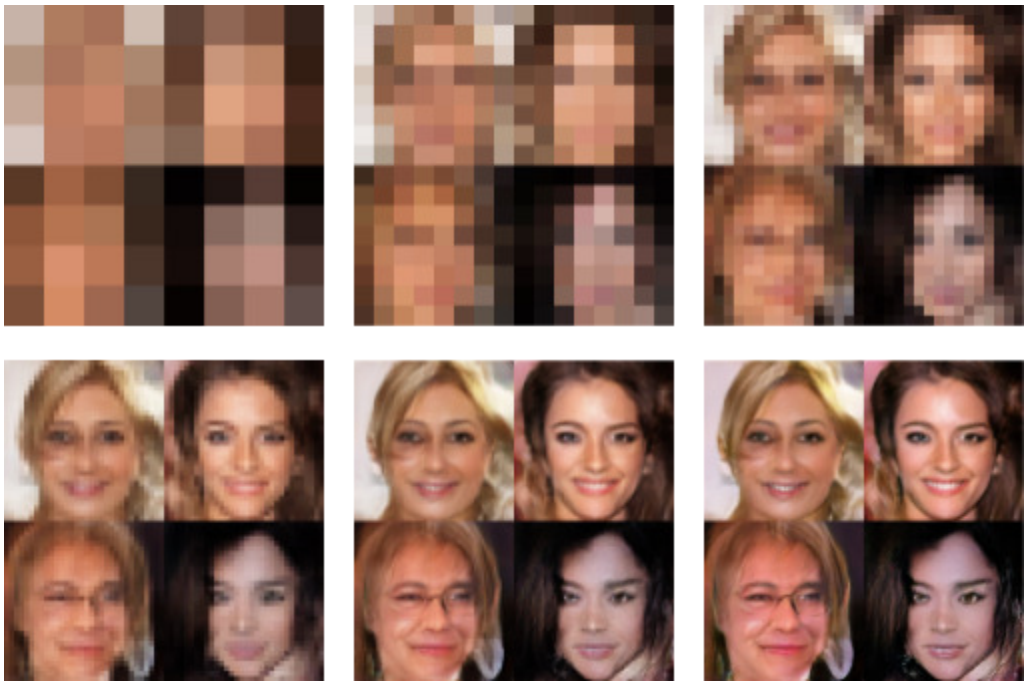


Fig. 25: Introduction of the consistency loss makes the outputs of all blocks consistent throughout the entire training. Here: outputs of blocks 2 to 7 at a single step during phase 7.

We run two experiments to investigate the effectiveness of consistency loss: *exp_consistency_30*, and *exp_consistency_300.* Both of them are based on *exp_condition*, and differ in the weight given to consistency loss, which is equal to 30 in the former, and to 300 in the latter. The numbers are chosen so that the weighted consistency loss constitutes a fraction of the *modified loss* in the first experiment, and is roughly the same order of magnitude in the second one. See the Figures below for the results of the experiments.

Experiment	Frechet Inception Distance	MSSSIM Diversity
exp_condition	22.6782	0.6725
exp_consistency_30	24.2859	0.6526
exp_consistency_300	24.4284	0.6507

Fig. 26: Quality metrics for consistency loss experiments.

Experiment	MSSSIM Isolation Overall	MSSSIM Isolation Block 5	MSSSIM Isolation Block 6	MSE Isolation Overall	MSE Isolation Block 5	MSE Isolation Block 6
exp_condition	0.9303	0.9529	0.9665	-0.0259	-0.0074	-0.0046
exp_consistency_30	0.9407	0.9686	0.9827	-0.0214	-0.0033	-0.0014
exp_consistency_300	0.9415	0.976	0.9839	-0.0214	-0.0026	-0.0013

Fig. 27: Feature isolation metrics for consistency loss experiments.

Experiment	Code Reconstruction Error	Code Reconstruction Error Block 5	Code Reconstruction Error Block 6	InfoGAN Penalty Overall	InfoGAN Penalty Block 5	InfoGAN Penalty Block 6
exp_condition	0.3035	0.2266	0.1432	1.0027	0.9604	0.9357
exp_consistency_30	0.25	0.2086	0.1369	0.9736	0.9542	0.9346
exp_consistency_300	0.2516	0.2126	0.1391	0.9746	0.9559	0.935

Fig. 28: Metrics related to the control of semantic features for consistency loss experiments.

The metrics suggest that image quality remains only slightly affected. The diversity is slightly reduced or on par, while feature isolation is improved in terms of both MSSSIM and MSE. Therefore, the consistency loss achieves its objective of enforcing better feature isolation. Interestingly, Code Reconstruction Error and InfoGAN Penalty are also visibly improved.

Most importantly, the consistency loss greatly reduces the side effects related to color balance of the image. When varying a single coordinate responsible for a particular feature, we no longer see the accompanying side effect of the face drifting from pale white to saturated red (as in Fig. 19).

6 Latent code size

The latent code size in previous experiments was fixed to 80 variables, 16 per each phase from 2 to 6. We verify this design choice here. To this end, we run two experiments, with number of variables reduced to 8 in the first one, and to 4 in the second one. All other parameters are same as in *exp_condition*.

Experiment	Frechet Inception Distance	MSSSIM Diversity
exp_condition	22.6782	0.6725
exp_condition_8vars	18.9611	0.673
exp_condition_4vars	19.099	0.6655

Fig. 29: Quality metrics for latent code size experiments.

Experiment	MSSSIM Isolation Overall	MSSSIM Isolation Block 5	MSSSIM Isolation Block 6	MSE Isolation Overall	MSE Isolation Block 5	MSE Isolation Block 6
exp_condition	0.9303	0.9529	0.9665	-0.0259	-0.0074	-0.0046
exp_condition_8vars	0.878	0.9116	0.941	-0.0515	-0.0165	-0.0084
exp_condition_4vars	0.7901	0.8661	0.9039	-0.1009	-0.0282	-0.0253

Fig. 30: Feature isolation metrics for latent code size experiments

Experiment	Code Reconstruction Error	Code Reconstruction Error Block 5	Code Reconstruction Error Block 6	InfoGAN Penalty Overall	InfoGAN Penalty Block 5	InfoGAN Penalty Block 6
exp_condition	0.3035	0.2266	0.1432	1.0027	0.9604	0.9357
exp_condition_8vars	0.2349	0.1817	0.1269	0.9682	0.9459	0.9323
exp_condition_4vars	0.1864	0.1734	0.1106	0.9506	0.9434	0.9294

Fig. 31: Metrics related to the control of semantic features for latent code size experiments.

The metrics for *exp_condition_8vars* and *exp_condition_4vars* experiments show a clear trend. Namely, a lower number of structured variables entails an improvement to InfoGAN Penalty and Code Reconstruction Error at the cost of reduced feature isolation. Intuitively, this can be explained by the fact that with reduced size of the latent code, an increased number of semantic features can now be controlled by each variable. Understandably, such redundancy

in encoding of the latent code into the image results in the reconstruction of latent code being more reliable, and naturally reduces isolation.

As we can see, a higher size of the latent code yields better isolation. Experimenting with even bigger latent codes was not possible due to GPU memory limitations.

7 Summary of the disentangled features

We choose *exp_consistency_300* for further detailed inspection, as it reports the most satisfying feature isolation metrics, while still having satisfying image quality. The model is trained up until phase 9, and thus produces images in resolution of 512×512. We include an exhaustive list of features learned by *exp_consistency_300* in the Appendix. We also refer the reader to the animated version of the list: youtu.be/mOckeVkM1jU. Some of the subtler features are easier to notice in an animation, rather than in a static series of images.

In the second subsection of the Appendix we include a non-curated, random set of samples.

8 Technical considerations

For a reader interested in replicating experiments in this paper, we report our technical experience relevant to the techniques of InfoGAN and progressive training of GANs.

8.1 TensorFlow and progressive training

The experiments reported in this paper were implemented in TensorFlow [18], one of the most popular deep learning frameworks of today. Training in Tensorflow consists of two stages. First, the user defines a *static* computational *graph*, which defines how the model transforms inputs into its outputs, and how the loss is computed. The graph is automatically traversed and extended with operations for the computation of gradients of the loss with respect to the model's parameters. The actual training happens when this static graph is repeatedly *executed* on the chosen computational device, most frequently the GPU, each time updating the model's parameters.

This computational paradigm does not align well with the progressive training of GANs. The difficulty stems from the fact that our model grows across training phases, which is not easily expressed in a static computational graph. Secondly, certain hyperparameters, like batch size and input image resolution, change throughout training, thus changing the dimensions of tensors within the graph.

We found it easiest to completely rebuild the computational graph when entering a higher phase of training. This way we can optimize computation within each phase separately, for example through the usage of a dedicated data input stream, preprocessed for the resolution and batch size desired in the current phase. This approach, however, required some custom code to handle the importing of variables from the smaller model to the bigger one.

8.2 tensorflow.contrib.gan

The contrib branch of tensorflow codebase includes a dedicated tensorflow.contrib.gan library. We found certain parts of the library useful, for example the implementation of the most popular loss functions, and certain debugging utilities.

Unfortunately, the drawback of the library is that it is not easily customizable. More specifically, high-level functions of the library accept the model in a form of a python function that builds and returns the TensorFlow graph. If the user wants to build a graph more complex than it is expected by the library, it is necessary to use error-prone tricks like argument-dependent function signature, global collections, or to duplicate and modify parts of the library.

9 Potential applications and further work

9.1 Unsupervised extraction of high level features

The InfoGAN technique falls under the umbrella of *unsupervised* learning algorithms. In other words, it can be applied to extract high level information from *unlabelled* datapoints (in our case, images). By improving overall quality and isolation of the isolated features, we hope to increase the general applicability and effectiveness of this technique.

9.2 Extending the Neural Photo Editor

The Neural Photo Editor [17] is an application that makes use of Generative Adversarial Networks to aid editing of photographs. A user can upload a real-life photo and modify it by applying a chosen color to any place in the image. However, rather than simply replacing the pixel under the cursor with the chosen color in a paintbrush fashion, the editor changes the photo by introducing realistic high level changes to the picture that match the color and the location of the cursor. For example, by applying white color to the area between lips we can change the image so that a smile appears. In a similar fashion, the hairstyle, eye color, and other features of the image can be changed.

The basic principle behind Neural Photo Editor is to first find a *shadow image* - that is, a sample from a pre-trained Generator that matches the user-provided real-life photo as closely as possible. In its simplest form, the shadow image can be obtained by performing gradient descent on Generator's input z to minimize $MSE(G(z), photo)$. From that point on, it is the

shadow photo that is really edited. This is done, again, through gradient descent on the Generator's input, that optimizes the similarity of Generator's output and the original shadow photo with the paintbrush applied. Having done that, the areas of the shadow photo that changed the most are overlaid on the user-supplied photo.

This second stage of the Neural Photo Editor can be substituted with the direct manipulation of the Generator's input. If the Generator is trained to support disentangled representations, the user will be able to change semantic features of the supplied image with a set of sliders, each controlling a separate feature. If successful, this kind of high-level editing could give the user an easy way of applying changes that would be otherwise hard to achieve with vanilla Neural Photo Editor, like the control of the shape of the nose.

9.3 Impact of InfoGAN penalty on training stability

Our experiments support the claim that the inclusion of the Mutual Information Penalty in the model's loss functions improves training stability. We were able to train high quality, deep GAN models with the basic *modified loss* (note, that Nvidia [2] used more sophisticated loss functions in their experiments: WGAN-GP [12] and LSGAN [19]).

This paper provides only anecdotal evidence for InfoGAN's impact on training stability. We propose a more thorough research in this area as further work.

10 Summary

In this paper we explored the effectiveness of the InfoGAN technique in conjunction with progressive training of GANs. Firstly, we defined the aspects of interest: image quality, control over semantic features, and the level of isolation of the controlled features. We picked metrics to measure success in each of those areas, proposing novel metrics for the assessing of feature isolation.

We started from a naive application of InfoGAN loss. Later we experimented with a number of other approaches for employing the InfoGAN loss in a progressive fashion. We arrived at a solution that significantly improves feature isolation at the cost of lower, though still satisfactory quality, and with no significant impact on diversity.

We explored the phenomenon of lower-resolution outputs of the network drifting away from the highest-resolution output. By eliminating it, we simultaneously improved the feature isolation and InfoGAN metrics, at the cost of slightly impacted image quality and diversity. Most importantly, visual inspection of the images suggests that consistency loss removes the color-related side effects that previously accompanied the change of a single latent code coordinate.

Finally, we proposed an application for our architecture: it can be used to extend the Neural Editor introduced in [17] to automatically change semantic properties of real-world images.

We also proposed an area of further research supported by anecdotal evidence – investigating the adoption of the InfoGAN technique for the purpose of stabilising GAN training.

11 Bibliography

1. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio (2014). Generative adversarial nets. *arXiv preprint arXiv:1406.2661*.
2. Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen (2017). Progressive Growing of GANs for Improved Quality, Stability, and Variation. arXiv preprint arXiv:1710.10196.
3. Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, Pieter Abbeel (2016). InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. arXiv preprint arXiv:1606.03657.
4. Zhou Wang , Eero P. Simoncelli, Alan C. Bovik (2003). Multi-scale structural similarity for image quality assessment. In proceedings of the 37th IEEE Asilomar Conference on Signals, Systems, and Computers.
5. Augustus Odena, Christopher Olah, Jonathon Shlens (2016). Conditional Image Synthesis With Auxiliary Classifier GANs. arXiv preprint arXiv:1610.09585.
6. Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Sepp Hochreiter (2017). GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. arXiv preprint arXiv:1706.08500.
7. Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen (2016). Improved Techniques for Training GANs. arXiv preprint arXiv:1606.03498.
8. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei (2014). ImageNet Large Scale Visual Recognition Challenge. arXiv preprint arXiv:1409.0575.
9. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich (2014). Going Deeper with Convolutions. arxiv preprint arXiv:1409.4842
10. Ziwei Liu, Ping Luo, Xiaogang Wang, Xiaoou Tang (2014). Deep Learning Face Attributes in the Wild. arXiv preprint arXiv:1411.7766.
11. Martin Arjovsky, Soumith Chintala, Léon Bottou (2017). Wasserstein GAN. arXiv preprint arXiv:1701.07875.
12. Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, Aaron Courville (2017). Improved Training of Wasserstein GANs. arXiv preprint arXiv:1704.00028.

13. Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu (2016). WaveNet: A Generative Model for Raw Audio. arXiv preprint arXiv:1609.03499.
14. Deep Learning Book, chapter 20.9: Back-Propagation through Random Operations
15. Tero Karras (2017). One hour of imaginary celebrities. https://youtu.be/36lE9tV9vm0
16. Diederik P. Kingma, Jimmy Ba (2017). Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.
17. Andrew Brock, Theodore Lim, J.M. Ritchie, Nick Weston (2016). Neural Photo Editing with Introspective Adversarial Networks. arXiv preprint arXiv:1609.07093.
18. tensorflow.org
19. Xudong Mao, Qing Li, Haoran Xie, Raymond Y.K. Lau, Zhen Wang, Stephen Paul Smolley (2016). Least Squares Generative Adversarial Networks. arXiv preprint arXiv:1611.04076.
20. Sergey Ioffe, Christian Szegedy, 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv preprint arXiv:1502.03167.

12 Appendix

12.1 Feature summary

What follows is a complete summary of features isolated by *exp_consistency_300*. We advise the reader to also see the animated version of the summary at youtu.be/mOckeVkM1jU, which is easier to analyse than a static series of images. The summary was created using *exp_consistency_300*, trained up to phase 9.

coord 79: Size of the irises, face redness.

coord 78: Eyelids, face redness.

coord 77: Nostrils & eyelids.

coord 76: Shape of the nostrils, wrinkles.

coord 75: Wrinkles.

coord 74: Eyelids (subtle), wrinkles.

coord 73: Wrinkles, skin redness.

coord 72: Lower eyelid.

coord 71: Shape of the eyes.

coord 70: Face redness.

coord 69: Wrinkles.

coord 68: Hair texture.

coord 67: Shape of the nostrils.

coord 66: Hair texture.

coord 65: Size of the irises.

coord 64: Color of the irises.

coord 63: Skin color, face geometry.

coord 62: Look direction.

coord 61: Nose tip: left / right, eyebrows.

coord 60: Vertical face stretch.

coord 59: Nose geometry (subtle).

coord 58: Upper lip + face geometry.

coord 57: Look direction.

coord 56: Nose: vertical position.

coord 55: Eyebrows shape.

coord 54: Nose position, look direction.

coord 53: Age.

coord 52: Nose: upturned tip.

coord 51: Cheeckbones and nose, subtle.

coord 50: Nose tip shape, hair texture.

coord 49: Nose length.

coord 48: Eyebrows up / down.

coord 47: Face oval: chin.

coord 46: Subtle head inclination.

coord 45: Hair: curly / straight.

coord 44: Mouth open / closed, man / woman.

coord 43: Forward / backward inclination, hair texture.

coord 42: Face oval, smile, chin size.

coord 41: Man / woman, slight nod.

coord 40: Smile, chin size.

coord 39: Lower jaw width.

coord 38: Mouth position: up / down.

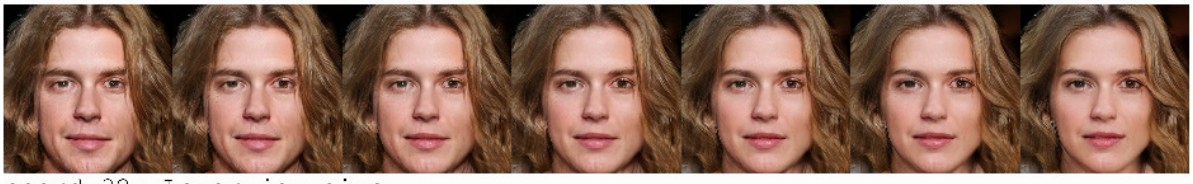
coord 37: Face oval: width.

coord 36: Lower jaw length.

coord 35: Head size.

coord 34: Jaw size.

coord 33: Smile: upper lip.



coord 32: Lower jaw size.

coord 31: Face oval: width.

coord 30: Hairstyle.

coord 29: Hairstyle: left side.

coord 28: Hairstyle.

coord 27: Hairstyle.

coord 26: Hairstyle: right side.

coord 25: Neck width.

coord 24: Hairstyle: background size.

coord 23: Hairstyle.

coord 22: Hairstyle.

coord 21: Skin: pale / red.

coord 20: Hairstyle.

coord 19: Hairstyle: forhead right size, overall size.

coord 18: Hairstyle.

coord 17: Hairstyle.

coord 16: Face oval: size.

coord 15: Rotation + hair color.

coord 14: Rotation + hairstyle.

coord 13: Skin paleness + hair color.

coord 12: Rotation + hairstyle.

coord 11: Rotation + hair color.

coord 10: Blond / black hair, slight rotation.

coord 9: Hairstyle: color, bangs.

coord 8: Hair color.

coord 7: Hairstyle: color, bangs.

coord 6: Lighting, hair color.

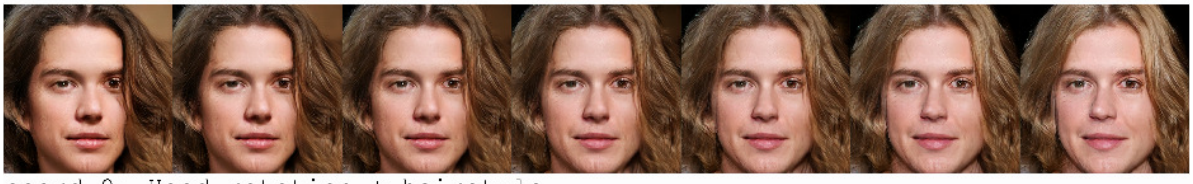
coord 5: Hairstyle.

coord 4: Hair length & color.

coord 3: Left / right rotation.

coord 2: Hairstyle, light.

coord 1: Light bright / dark.

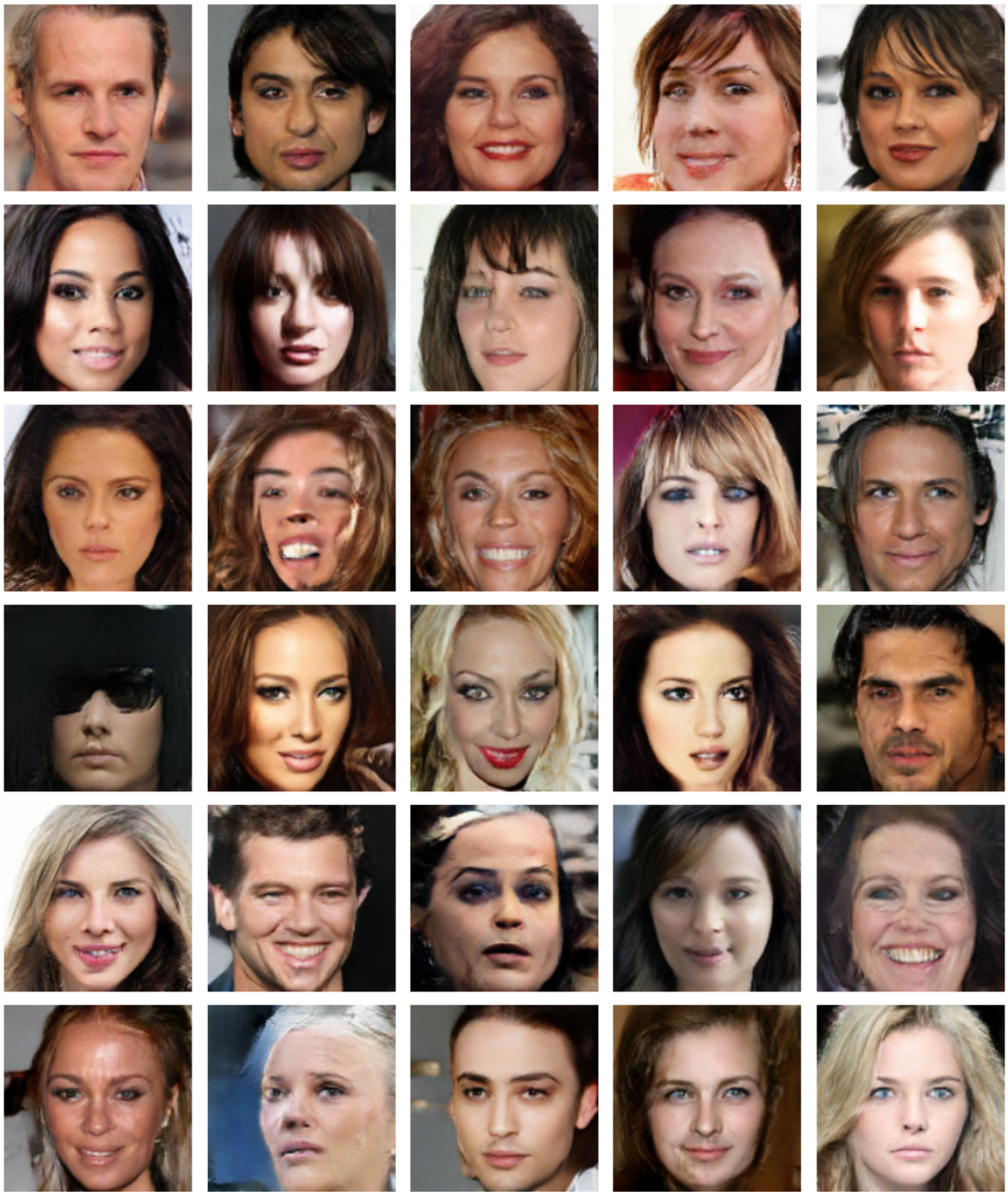


coord 0: Head rotation + hairstyle.

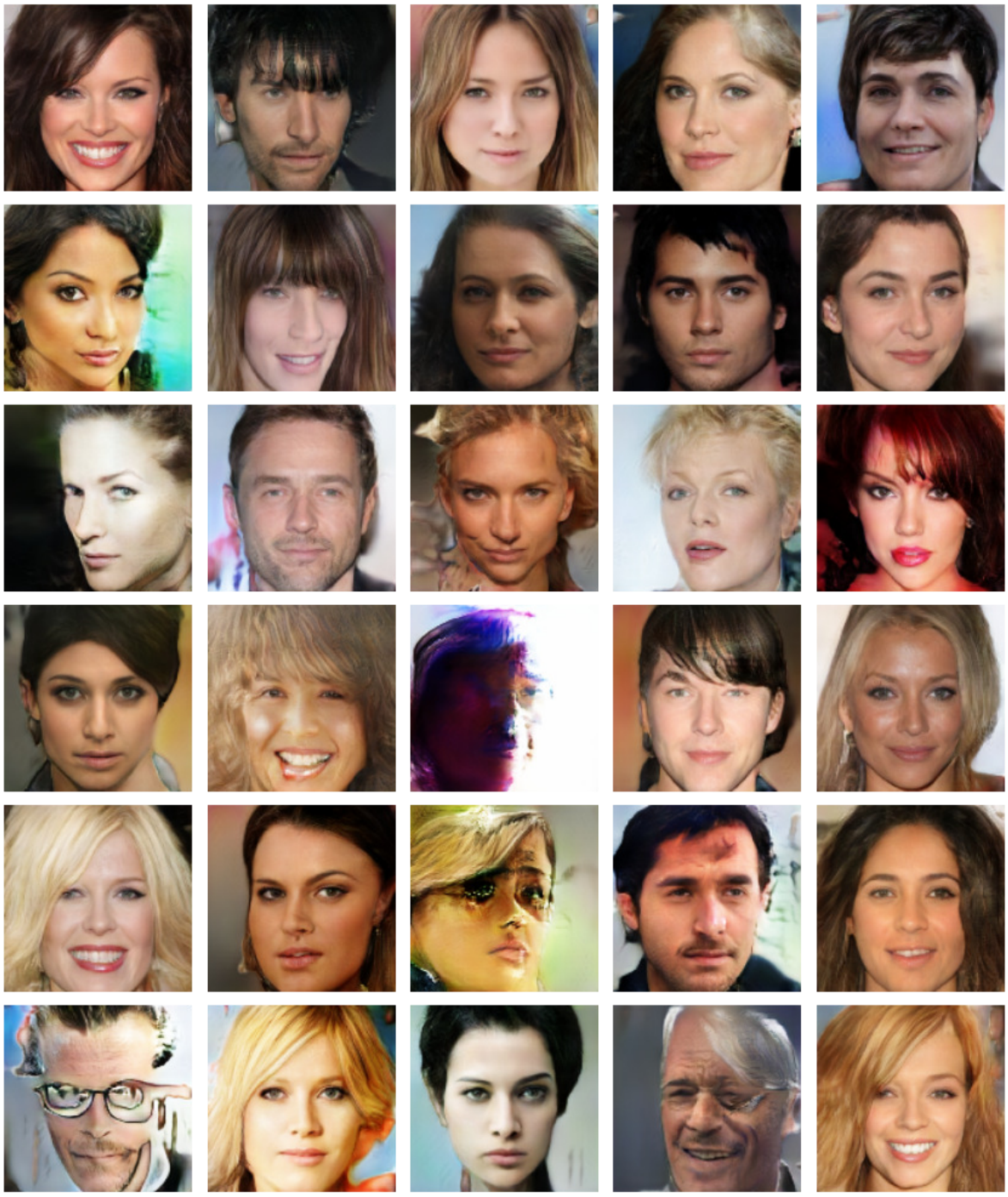
12.2 Non-curated set of samples

Each of the following subsections contains a random sample of images from the experiment mentioned in the subsection heading.

12.2.1 exp_condition



12.2.2 exp_vanilla



12.2.3 exp_consistency_300

Samples from exp_consistency_300, trained up until phase 9.