THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

Alternative Architectures for Image Generation and Residual Dilated Convolutions for Image Colorization with Adversarial Networks

by

Christopher Curro

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

September 14, 2016

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE
ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dean, School of Engineering   Date

Prof. Sam Keene, Thesis Advisor   Date

Acknowledgments

To Sam Keene for all he's done as an advisor, a professor, and a friend.

To my peers under Sam Keene: Ethan Lusterman, Daniel Gitzel, and David Li, we've finished it together.

To the rest of the electrical engineering faculty, full-time and adjunct, for the countless hours.

To the remaining staff and faculty of The Cooper Union; your friendships have been invaluable these years.

To my friends among the students and alumni; I've needed you all this time. I thank every one of you, especially those who have made me smile in the peaks of despair. From folk on the 9th floor of 29 3rd Avenue, to the 6th floor of 41 Cooper Square, from my radio pals on the staircase, to the poetry gang in the park, from the tunnels beneath the Great Hall, to the rooftop gardens, from the nonstop space, to the Peter Cooper Suite, I thank you.

To Sarah Cooper, Abram S. Hewitt, Edward Cooper, and all those today that act as steward to the plans laid out by last my acknowledgment.

To Peter Cooper for the institution of "grander, higher and more durable character" that has brought us all together.

Abstract

Generative adversarial networks have shown state of the art performance in natural image generation, however they are difficult to train and cannot always generate convincing samples for a given data-set. We introduce a modified discriminator loss function utilizing the properties of hinge loss for generative adversarial networks, and integrate work on moment matching and style-transfer to introduce a new generator loss function for generative adversarial networks. Both of these new loss functions are used for full image generation. Further, in our study of generative adversarial networks we introduce a new architecture for coloring gray-scale photographs using residual networks consisting of dilated convolution operations.

Contents

List of Figures

1 Introduction

1.1 Machine learning

Machine learning in general encompasses all tasks where we learn from data to make some kind of decision or estimate [1]. Classical examples include handwritten digit recognition, and flower species classification based on sepal width and petal length. As a rough mathematical example, consider a sample of N pairs of examples drawn from a larger population $\mathfrak{P}$:

$$\mathfrak{D} = \{(x_i, y_i) \colon i = 1, 2, \dots, N\} \tag{1}$$

$$\mathfrak{D} \subset \mathfrak{P} \tag{2}$$

Our machine learning task may be to find some function $f(x)$ that returns an estimate $\hat{y}$ for any (x, y) sample drawn from the population at large, where x is an input observation and y is the target or label emission. To learn the function we might setup a loss function $l(y, \hat{y})$ to quantify how well our model performs on our sample $\mathfrak{D}$. As a general rule, the loss function satisfies the following limit:

$$\lim_{\hat{y_i} \to y_i} l(y_i, \hat{y_i}) = 0 \tag{3}$$

In other words our goal is to find some $f(x)$ that minimizes the loss function $l(y, \hat{y})$ for all (x_i, y_i) pairs in $\mathfrak{D}$. This type of task is referred to as supervised learning, because we have a domain of inputs and a specified codomain of known or labeled targets; there are other applications with unknown targets, these tasks are referred to as unsupervised learning tasks. With a few extra constraints we are able use a number of techniques to accomplish this supervised learning goal.

1.1.1 Differentiable parametric modeling

Let us restrict our study of functions f specifically to functions that are differentiable and parametric in form, so from now on we refer to $f(x \mid \theta)$ where θ refers to the function's set of parameters. Using these new terms our goal is to find some $\hat{\theta}$ that satisfies the following:

$$\hat{\theta} = \arg\min_{\theta} \left(l(y, f(x \mid \theta)) \right), \ \forall x, y \tag{4}$$

For any problem of this type we would like to calculate the gradient of l with respect to each θ_i, set them all to zero and solve for each θ_i:

$$\frac{\partial l(x, f(x \mid \theta))}{\partial \theta_i} = 0, \ \forall i \tag{5}$$

However it is not always possible or even desirable to solve these analytically, due to the computational complexity of finding such solutions for systems with a very large number of free parameters. As an example, consider a solution that requires a matrix inverse, which has $O(N^3)$ memory complexity; for large number of training points N this can take an intractable amount of memory. In lieu of an analytic solution we can use a numerical optimization approach, and jointly optimize the θ_i's via a gradient descent algorithm. For example we could use this update rule, which we refer to as steepest descent:

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial l(x, f(x \mid \theta))}{\partial \theta_i} \tag{6}$$

where η is the learning rate parameter, which controls how large our steps are towards the minimum [2]. The magnitude of the learning rate represents a trade-off between the rate of convergence and the closeness to the nearest minima at convergence. Therefore it is common to schedule decreases in the learning rate as learning continues so that early in the learning process we have large steps but then as we the near a minima we take small steps to hone in on it [3].

1.1.2 A simple example with basis expansions

Let us consider our first supervised learning example. Consider a noisy sine-wave consisting of k samples:

$$\mathbf{y} = \sin \mathbf{x} + \mathbf{n},\ \ \mathbf{y}, \mathbf{x}, \mathbf{n} \in \mathbb{R}^k \tag{7}$$

We would like to find some $f(\mathbf{x} \mid \theta)$ that produces an estimate $\hat{\mathbf{y}}$. Let us consider the following functional form for f:

$$f(\mathbf{x}) = m\mathbf{x} + b,\ \ m, b \in \mathbb{R} \tag{8}$$

Or we can rewrite this as a matrix multiplication by packing $\mathbf{x}$ with a ones vector to create a $2 \times k$ matrix we refer to as Φ, and packing m and b into a vector $\mathbf{w}$:

$$\hat{\mathbf{y}} = \mathbf{w}^T \Phi \tag{9}$$

If we a select a mean squared error loss function, we are able to compute $\hat{m}$ and $\hat{b}$ analytically:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2} \|\hat{\mathbf{y}} - \mathbf{y}\|^2 \tag{10}$$

Now we substitute our model in for $\hat{y}$ and set the gradient to zero:

$$\frac{\partial l(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}} \left(\frac{1}{2} \left\| \mathbf{w}^T \Phi - \mathbf{y} \right\|^2 \right) = 0 \tag{11}$$

Carrying out the derivative we see:

$$\frac{1}{2} \frac{(\mathbf{w}^T \Phi - \mathbf{y})^T}{\|\mathbf{w}^T \Phi - \mathbf{y}\|} \Phi^T = 0 \tag{12}$$

$$\rightsquigarrow \mathbf{w}^T \Phi - \mathbf{y} = 0 \tag{13}$$

$$\rightsquigarrow \mathbf{w}^T = \mathbf{y} \Phi^+ \tag{14}$$

where Φ^+ denotes the right multiplicative inverse form of the Moore-Penrose pseudoinverse: $\Phi^T \left(\Phi \Phi^T \right)^{-1}$. An example of a fit using this method of regression is in Figure 1. Looking at the figure it is clear that working with a model that is linear

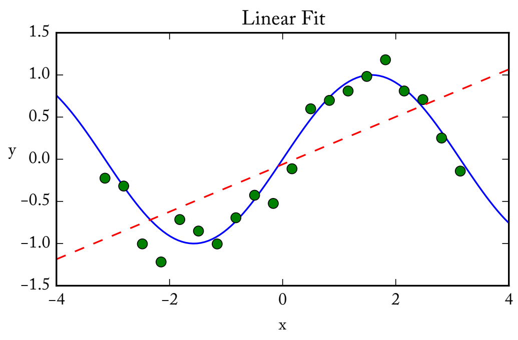


Figure 1: Linear regression of a noisy sine-wave. Data points are in green circles, the fit is a dashed red line, and the underlying function is a blue line.

with respect to the input data is limited to being able to represent underlying data is that is linear.

To overcome this limitation we modify the linear regression technique by introducing the concept of basis expansion. While in the last example we constructed a matrix Φ implicitly of the form:

$$\Phi = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{pmatrix} \tag{15}$$

Now we consider a matrix Φ of the form:

$$\Phi = \begin{pmatrix} \phi_0(x_1) & \phi_0(x_2) & \cdots & \phi_0(x_n) \\ \phi_1(x_1) & \phi_1(x_2) & \cdots & \phi_1(x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{M-1}(x_1) & \phi_{M-1}(x_2) & \cdots & \phi_{M-1}(x_n) \\ 1 & 1 & \cdots & 1 \end{pmatrix} \tag{16}$$

Where the $\phi_i(\cdot)$'s corresponds to some family of parametric basis functions. Some examples include the gaussian family of functions:

$$\phi_i(x \mid \mu_i, \sigma_i) = \exp\left(-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right) \tag{17}$$

and the sigmoidal family:

$$\phi_i(x \mid \mu_i, \sigma_i) = \frac{1}{1 + \exp\left(\frac{-(x-\mu_i)}{\sigma_i}\right)} \tag{18}$$

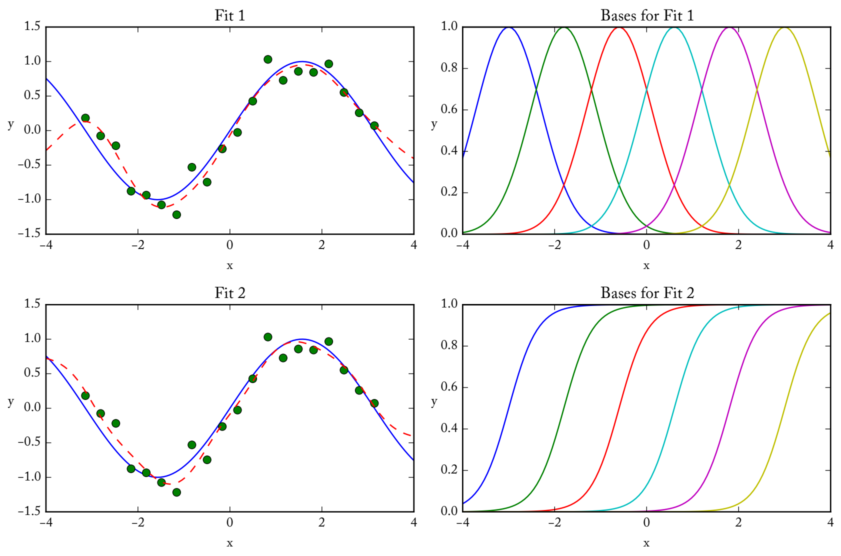


Figure 2: Linear regression of a noisy sine-wave with basis expansion. Data points are in green circles, the fit is a dashed red line, and the underlying function is a blue line.

Keeping the mean squared error loss function from the previous example, the derivation of our estimate for $\mathbf{w}$ remains the same, therefore our solution of $\mathbf{w}^T = \mathbf{y}\Phi^+$ still holds. Following this method we can compute the fits shown in Figure 2 of a noisy sine-wave with gaussian and sigmoidal basis functions.

1.1.3 Kernels

Often we find ourselves needing a method to measure the similarity between two examples. One way to do this is with a function with signature:

$$k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R} \tag{19}$$

that measures the similarity between any pair of inputs $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. We call this function $k(\cdot, \cdot)$ a kernel if it can be used to construct a positive semi-definite matrix $\mathbf{K}$ with elements $k(\mathbf{x}_n, \mathbf{x}_m)$ for all possible $(\mathbf{x}_n, \mathbf{x}_m)$ pairs. We call this matrix $\mathbf{K}$ the Gramm matrix. It turns out that with this requirement, there exists a dual representation for every function $k(\mathbf{x}_n, \mathbf{x}_m)$ that can construct a Gramm matrix. This alternative representation is an inner product of basis functions:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') \tag{20}$$

It follows from this statement then, that in any situation where an inner product is computed in a feature space we can replace that inner product with another kernel function. We call this kernel substitution or an instance of the kernel trick. The kernel trick allows us to substitute inner products with other kernel functions which are themselves inner products of nonlinear basis functions, therefore we can take linear methods and transform them into nonlinear ones with respect to the input data. Furthermore this allows us to implicitly compute high (or even infinite) dimensional inner products implicitly, allowing for more complex representations of data.

1.1.4 Two sample test statistic

Consider two samples $\mathfrak{X} = \{x_1, x_2, \dots, x_N\}$ and $\mathfrak{Y} = \{y_1, y_2, \dots, y_M\}$, where the underlying x_i and y_j's are draw from two distributions. We can compute a two sample test statistic to measure how similar these two samples are [4]. Consider following statistic:

$$\begin{aligned}\psi(\mathfrak{X}, \mathfrak{Y}) = \frac{1}{N^2} \sum_i \sum_{i'} \phi(x_i)^T \phi(x_{i'}) + \\ + \frac{1}{M^2} \sum_j \sum_{j'} \phi(y_j)^T \phi(y_{j'}) + \\ - \frac{1}{NM} \sum_i \sum_j \phi(x_i)^T \phi(y_j)\end{aligned} \tag{21}$$

where $\phi \colon \mathbb{R} \to \mathbb{R}^n$ for any $n \in \mathbb{Z}^+$. If $\phi(\cdot)$ is the identity function this statistic tends to zero as the means of the distributions tend to each other. As the argument of these sums are inner products we can replace them with a kernel function $k(\cdot, \cdot)$ to measure against other moments, and for certain choices of kernel function all moments of the distributions.

1.1.5 Curse of dimensionality

We've introduced two main techniques to allow for complex feature space representations of data: basis expansions and kernel methods. We've shown that both of these methods can take low dimensional data and project them onto higher dimensioned feature spaces for making a prediction. While high dimensioned feature space representations can allow for more powerful models, it can also hinder them.

Take for example N samples $\mathbf{x}_i \in \mathbb{R}^n$ which exist in the n-dimensioned unit ball in $\mathbb{R}^n$. Then using basis expansions we project the $\mathbf{x}_i$'s onto the m-dimensioned unit ball in $\mathbb{R}^m$ where $m > n$.

We can compute the average density of the $\mathbf{x}_i$'s within the unit ball in $\mathbb{R}^n$:

$$\rho_n(\mathbf{x}_i) = \frac{\Gamma\left(\frac{n}{2}+1\right)}{\pi^{n/2}} N \tag{22}$$

Where $\Gamma(\cdot)$ is Euler's Gamma function (extension of factorial to real and complex numbers). While the average density of the transformed $\mathbf{x}_i$'s within in the unit ball in $\mathbb{R}^m$ is:

$$\rho_m(\mathbf{x}_i) = \frac{\Gamma\left(\frac{m}{2}+1\right)}{\pi^{m/2}} N \tag{23}$$

Dividing the two we can calculate the change in the density between the representations:

$$\begin{align} \frac{\rho_n(\mathbf{x}_i)}{\rho_m(\mathbf{x}_i)} &= \frac{\Gamma\left(\frac{n}{2}+1\right)\pi^{m/2}}{\Gamma\left(\frac{m}{2}+1\right)\pi^{n/2}} \tag{24} \\ &= \frac{\Gamma\left(\frac{n}{2}+1\right)\pi^{(m-n)/2}}{\Gamma\left(\frac{m}{2}+1\right)} \tag{25} \end{align}$$

which will approach zero as m increases by nature of the Gamma function's rapidly increasing value (far outstripping the $\pi^{(m-n)/2}$) and the fact that $m > n$. This indicates that as the dimensionality of the feature representation increases the density of examples in that space rapidly decreases, at a rate proportional to the growth of the Gamma function. As the feature space becomes more sparsely populated it can be difficult to make decisions because in some sense the model needs to interpolate between the data points. Therefore we like to work in the least dimensioned feature space that can satisfy our optimization goals, or otherwise get more data to populate the higher dimensioned space if possible.

1.1.6 Over-fitting

Another complication in machine learning that motivates working in the smallest possible feature space is the concept of over-fitting. Generally over-fitting refers to the idea that with increased model complexity we may end up modeling the noise in the data-set rather than just the underlying trends. Therefore when we introduce new data at inference time, the model performs below expectations because it did not generalize.

To work with an example let us consider $\mathbf{x} \in \mathbb{R}^n$:

$$x_i = 0.1t_i^3 + t_i^2 + 0.3t_i - 1 + \epsilon_i \text{ for } i = 0, 1, \ldots, N-1 \tag{26}$$

where the t_i's are linearly spaced over the range $[-2, 2]$, and the ϵ_i's are independently identically distributed gaussian random variables with a variance of $1/4$. Given this noisy polynomial we attempt to find some linear fit using polynomial bases. With this in consideration we have our set of M basis functions:

$$\phi_j(t) = t^j \text{ for } m = 0, 1, \ldots, M-1 \tag{27}$$

Setting up the rest of the model as before, with a mean squared error loss function:

$$\hat{\mathbf{y}} = \mathbf{w}^T\Phi \tag{28}$$

$$\frac{\partial l(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}}\left(\frac{1}{2}\|\mathbf{w}^T\Phi - \mathbf{y}\|^2\right) = 0 \tag{29}$$

$$\frac{1}{2}\frac{(\mathbf{w}^T\Phi - \mathbf{y})^T}{\|\mathbf{w}^T\Phi - \mathbf{y}\|}\Phi^T = 0 \tag{30}$$

$$\rightsquigarrow \mathbf{w}^T\Phi - \mathbf{y} = 0 \tag{31}$$

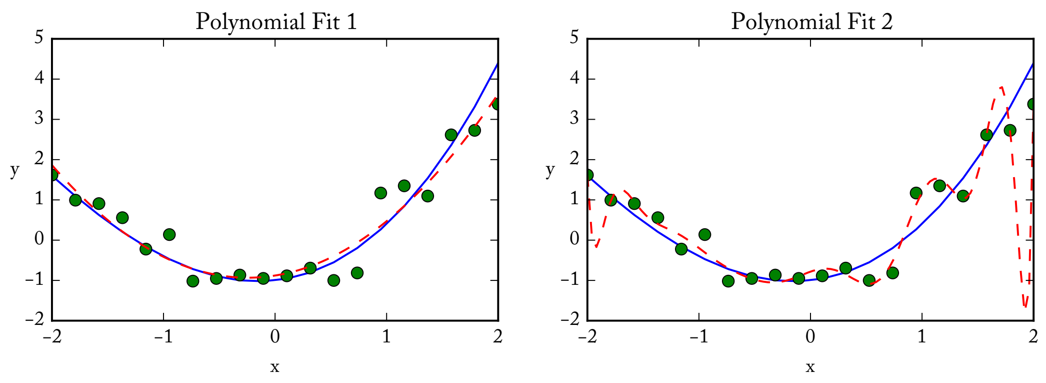


Figure 3: On the left a polynomial fit for $M = 3$ and on the right a fit for $M = 15$. Data points are in green circles, the fit is a dashed red line, and the underlying function is a blue line.

$$\rightsquigarrow \mathbf{w}^T = \mathbf{y}\Phi^+ \tag{32}$$

Using this analytic solution we compute two fits, one for $M = 3$ and another for $M = 15$, we can see the results of these fits in Figure 3. Looking at the $M = 3$ fit we can see the fit straddles nicely in-between the sample points remaining close to the underlying function, while the $M = 15$ fit closely follows the examples creating a high variance fit that at some points is very far from the underlying function.

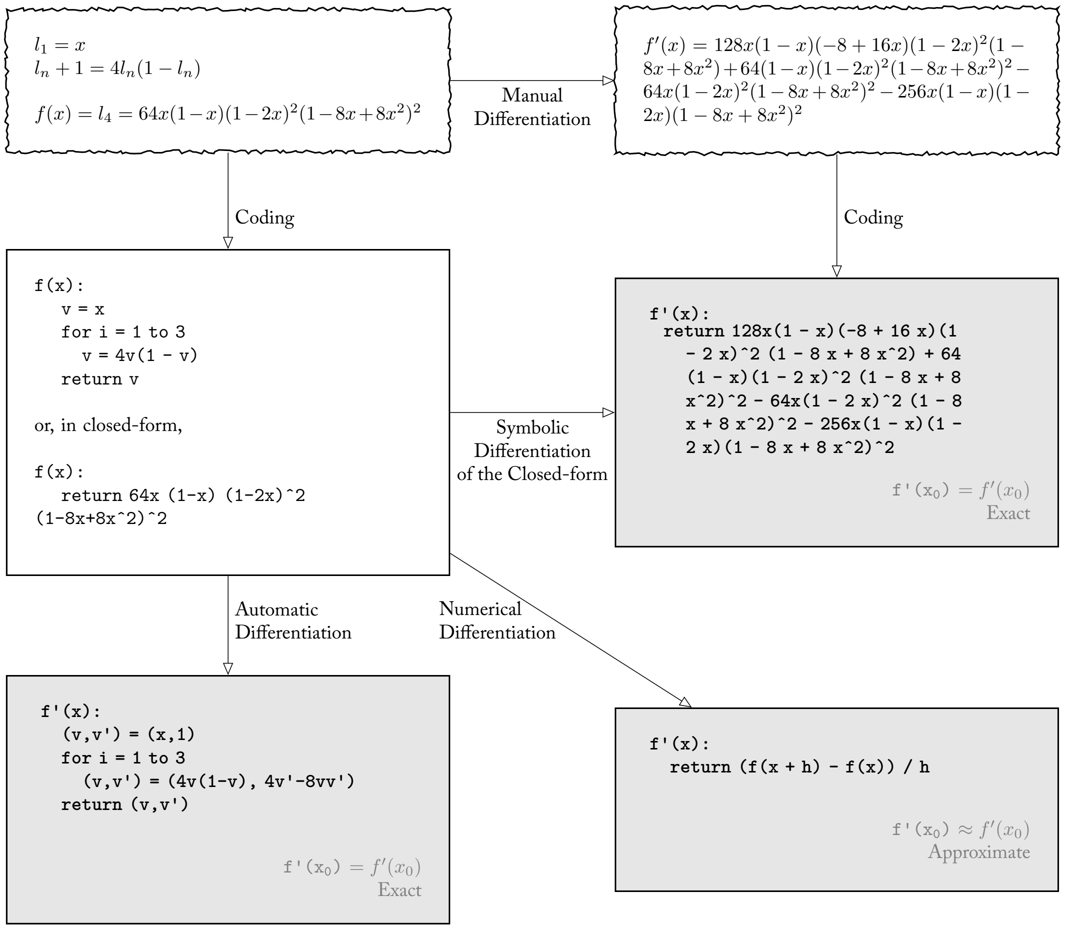
1.2 Neural networks

Neural networks have shown tremendous capabilities in the task of computer vision, routinely winning major contests and displaying new states of the art in a variety of research categories [5–8]. Neural networks have been used for object detection, semantic segmentation, and artistic style transfer, all of which are difficult perceptual tasks, therefore it is natural to use this type of model for the fully perceptual task of natural image generation.

1.2.1 Neural networks as computational graphs

A neural network generally is a computational graph $G = (V, E)$ where the vertices V correspond to nodes of computation, and the edges E correspond to data flow paths connecting the computational nodes. We limit our discussion to feed-forward or directed acyclic graphs, in other words graphs where given a starting vertex v we are unable to follow the directed edges away from it and return to v. With this limitation in place we are able to focus on stateless graphs, which treat input data examples independently of one another.

With the additional limitation that all computational nodes in the graph perform differentiable operations, and that some nodes are parametric, we are able to use automatic differentiation and a gradient descent like optimization algorithm to optimize the parameters of the graph against a differentiable loss function [9]. Automatic differentiation is an alternative to numeric and symbolic differentiation, that is efficient, accurate to machine precision, and scalable to arbitrary graphs consisting of elementary operations. The core tenet of automatic differentiation is that since graph vertices are differentiable we can perform the chain rule at each node to build up gradients of the full graph. Furthermore automatic differentiation is the generalization of the backpropagation algorithm [10] designed for training neural networks. Several machine learning frameworks, such as Theano and TensorFlow implement automatic differentiation, allowing us to specify the functional form of the graph and get gradients at minimal cost [11, 12]. Figure 4 shows a very basic example of automatic differentiation and compares it to symbolic and numerical differentiation.


$l_1 = x$
$l_n + 1 = 4l_n(1 - l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

```
f(x):
   v = x
   for i = 1 to 3
     v = 4v(1 - v)
   return v
```

or, in closed-form,

```
f(x):
   return 64x (1-x) (1-2x)^2
(1-8x+8x^2)^2
```

```
f'(x):
  return 128x(1 - x)(-8 + 16 x)(1
    - 2 x)^2 (1 - 8 x + 8 x^2) + 64
    (1 - x)(1 - 2 x)^2 (1 - 8 x + 8
    x^2)^2 - 64x(1 - 2 x)^2 (1 - 8
    x + 8 x^2)^2 - 256x(1 - x)(1 -
    2 x)(1 - 8 x + 8 x^2)^2
```

f'(x_0) $= f'(x_0)$
Exact

```
f'(x):
   (v,v') = (x,1)
   for i = 1 to 3
     (v,v') = (4v(1-v), 4v'-8vv')
   return (v,v')
```

f'(x_0) $= f'(x_0)$
Exact

```
f'(x):
  return (f(x + h) - f(x)) / h
```

f'(x_0) $\approx f'(x_0)$
Approximate

Manual Differentiation
Coding
Coding
Symbolic Differentiation of the Closed-form
Automatic Differentiation
Numerical Differentiation

Figure 4: The range of approaches for differentiating mathematical expressions and computer code. Symbolic differentiation (center right) gives exact results but suffers from unbounded expression swell; numeric differentiation (lower right) has problems of accuracy due to round-off and truncation errors; automatic differentiation (lower left) is as accurate as symbolic differentiation with only a constant factor of overhead. Figure and caption reproduced from [9].

1.2.2 Neural networks operations

1.2.2.1 Fully connected neural networks

While we have introduced the idea that neural networks can be made up of arbitrary computational nodes, there are several common types of nodes that are used to build up a toolkit of functional forms at our disposal. First, let us a consider a simple inner product with signature:

$$\langle \cdot, \cdot \rangle \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \tag{33}$$

And functional form:

$$\langle \mathbf{w}, \mathbf{x} \rangle = \sum_{i=1}^{n} w_i x_i \tag{34}$$

We can interpret this inner product as a single node in our graph, with parameters $\mathbf{w}$ and input vector $\mathbf{x}$. While this functional form is nice, in that it is linear, and easy to define, it is limited by the fact that it brings the rich space $\mathbb{R}^n \times \mathbb{R}^n$ down to $\mathbb{R}$ which has limited representational power for problems of interest in machine learning. Instead we will follow a common design pattern, and create an array of inner product nodes at a constant number of hops from the source node in the graph, or as we will refer to from here on, at a constant depth. In this case the signature would be:

$$g \colon \mathbb{R}^{m \times n} \times \mathbb{R}^n \to \mathbb{R}^m \tag{35}$$

And functional form:

$$g\left(\mathbf{W}, \mathbf{x}\right)_j = \sum_{i=1}^{n} w_{ji} x_i \tag{36}$$

which we recognize as ordinary matrix multiplication $\mathbf{W}\mathbf{x}$, where $\mathbf{W} \in \mathbb{R}^{m \times n}$. We call this collection of nodes at a common depth a layer. This type of matrix multiplication layer is generally referred to as a linear, dense, or fully connected layer.

Layers can be connected together with the goal of creating a more powerful model. If we feed one dense layer into another directly we have have a functional form:

$$\mathbf{y} = \mathbf{W}_n \cdots \mathbf{W}_3 \mathbf{W}_2 \mathbf{W}_1 \mathbf{x} \tag{37}$$

where the $\mathbf{W}_i$'s are all multiplicatively compatible. However this approach is actually equivalent to a single matrix multiply $\mathbf{Wx}$ where $\mathbf{W}$ is the matrix product of the $\mathbf{W}_i$'s, so we did not actually achieve our goal of being able to express complex relationships between our output and input variables, our model is still merely linear.

To address this issue we introduce the addition of a nonlinear transformation $f(\cdot)$ (known as an activation function) at the output of the matrix multiply. So instead our functional form would be:

$$\mathbf{y} = f(\mathbf{W}_n \cdots f(\mathbf{W}_3 f(\mathbf{W}_2 f(\mathbf{W}_1 \mathbf{x}))) \cdots) \tag{38}$$

As long as $f(\cdot)$ is differentiable we are able to train our now highly nonlinear model with our methods of automatic differentiation and gradient based optimization.

Finally we introduce a bias $\mathbf{b}$ so that each node in a layer is not constrained to intercept zero. Therefore our full functional form for a single fully connected layer with a non-linearity is:

$$\mathbf{y} = f\left(\mathbf{Wx} + \mathbf{b}\right) \tag{39}$$

with $\mathbf{W} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$.

A traditional activation function is the sigmoid function:

$$S(t) = \frac{1}{1 + \exp(-t)} \tag{40}$$

which maps $t \in \mathbb{R}$ (i.e the interval $[-\infty, \infty]$) to $S(t)$ on the interval $[0, 1]$. This can allow the interpretation of $S(t)$ as a probability. Using this activation function Cybenko proved the universal approximation theorem [13] which shows that neural

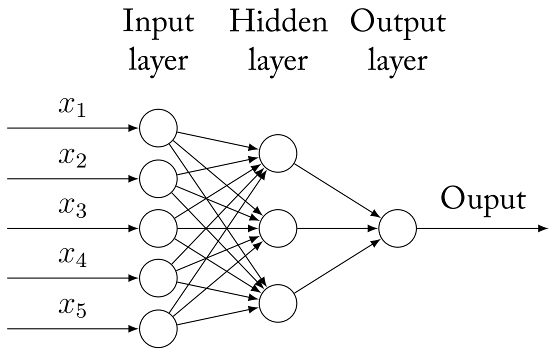


Figure 5: A single hidden layer feed-forward neural network. Each directed connection has a scaling factor w_{ij} associated with it; these entries make up the **W** matrix. Each circle, now referred to as a neuron, performs the sum and activation functions corresponding to the matrix multiply, and mapping by $f(\cdot)$ in Equation 39. Layers in the middle of the network are referred to as hidden layers because they are not directly observable — instead these are latent variables.

networks of the form in Figure 5 can approximate any function with appropriate domain and codomains given a large of enough number nodes in the middle layer, generally referred to as a hidden layer.

Figure 5 shows a feed-forward network with one hidden layer. The outputs of the hidden layer are referred to as latent variables and are a new type of representation for the input data **x**. With each hidden-layer added to the network increasingly complex data representations may be available, but the networks will also tend to over-fit their training data as the number of parameters increase. In order to combat over-fitting, and increase the generalization performance of the network new types of architectures have been proposed.

1.2.2.2 Convolutional neural networks

The first convolutional neural network was proposed by Fukushima in 1980 [14]. The idea of the network is that instead of creating a weight matrix **W** with parameters that correspond to every entry in **x**, that there would be a kernel of some small size that would be swept across $\mathbf{x}$. Processes of this type are generally referred to as convolutions. In the coming example we will be using tensors, which are multi-dimensioned arrays; the order of a tensor refers to how many dimensions it has. For our example we will consider an image represented by a tensor X with shape $d_i \times h \times w$. Let us consider a convolution kernel K of shape $d_i \times d_o \times k_h \times k_w$. With this kernel we consider the following operation:

$$\mathsf{Z}_{i,j,k} = \sum_l \sum_m \sum_n \mathsf{X}_{l,j+m-1,k+n-1} \mathsf{K}_{i,l,m,n} \tag{41}$$

over all indices l, m, n where the indexing for the summation is valid [15]. The output map Z has shape $d_o \times (h - k_h + 1) \times (w - k_w + 1)$. Effectively each element $\mathsf{Z}_{i,j,k}$ is the sum of l many $m \times n$ matrix multiplications of a d_i length vector from the map X with a $d_i \times d_o$ matrix slice of the tensor K across the spatial extent of the kernel.

Now we may consider strided convolutions, where we skip over some number of input features, resulting in a down-sampled output and decreased computational cost:

$$\mathsf{Z}_{i,j,k} = c(\mathsf{K}, \mathsf{X}, s)_{i,j,k} = \sum_l \sum_m \sum_n \mathsf{X}_{l,(j-1)\times s+m,(k-1)\times s+n} \mathsf{K}_{i,l,m,n} \tag{42}$$

This effectively moves the kernel over by s units for every product it computes over its spatial extent, therefore for say $s = 2$ the output map will have roughly half the spatial extent of the input. If we were to write expressions for the the output shape we would see:

$$w_o = \lfloor (w_i - k_w)/s + 1 \rfloor \tag{43}$$

$$h_o = \lfloor (h_i - k_h)/s + 1 \rfloor \tag{44}$$

Allow us to denote $c(\mathsf{K}, \mathsf{X}, s)_{i,j,k}$ across its entire range as $C(\mathsf{K}, \mathsf{X})$. Given this convention, we can write a multilayered network with the following recurrence relation:

$$\mathsf{Z}^l = f(C(\mathsf{K}^l, \mathsf{Z}^{l-1}) + \mathsf{B}^l) \tag{45}$$

where $f(\cdot)$ is an activation function, B is a tensor of appropriate shape for element-wise addition (with or without broadcasting [16]), the super-script denotes the layer, and $\mathsf{Z}^0 = \mathsf{X}$. Given a network of this type each successive layer will have a decreased spatial extent compared to the previous layer, but this partially represents an increase in the size of the receptive field of that layer compared to the previous one. We define the receptive field to the refer the the number of elements in the input tensor X that influence a given element in Z^l. With networks of this type we see a general trade-off between the spatial extent of the feature map and the size of the receptive field. By the definition of strided convolution it should be the clear that for strides of size one that the size of the receptive field grows at a rate directly proportional to the size of the kernel, while the spatial extent of the feature map can remain constant or shrink slightly depending on the padding scheme. Meanwhile if the stride is larger than one, then the size of the receptive field is directly proportional to the product of the size of the kernel and the stride, however the spatial extent of the feature map decreases according to Equations 43 and 44.

1.2.2.3 Convolutions with holes

In some instances we are interested in computing dense convolutional feature maps, and want to keep the spatial extent of the original input tensor, but still get the increased receptive field that comes with successive strided convolutional layers. For these situations we introduce a convolution with holes (or "à trous") algorithm. Also known as a dilated convolution, conceptually we modify the formulation so as to up-sample the kernel by introducing interspersed zeros, according to an up-sampling rate factor r:

$$\mathsf{Z}_{i,j,k} = \sum_l \sum_m \sum_n \mathsf{X}_{l,(j-1)+m\times r,(k-1)+n\times r} \mathsf{K}_{i,l,m,n} \tag{46}$$

Introduced for semantic segmentation purposes, convolutions with holes allow for output features maps with the same spatial extent as the input and with a receptive field across the entire input [17, 18]. This is possible in a reasonable number of layers because the size of the receptive field is a square of exponentially increasing size. For a point $\mathsf{Z}^l_{i,j,k}$ the receptive field is $\left(2^{l+1}-1\right) \times \left(2^{l+1}-1\right)$ in shape. With large receptive fields the networks are able to reason globally and assign semantic functions to high-level features.

1.2.2.4 Transposed convolutions

Consider our earlier discussion of convolutional neural networks; by the definition of the operation the spatial extent of the output feature map either remained constant or decreased depending on the padding and stride parameters. However sometimes it is desirable to have convolutional feature maps but with the goal of increasing the spatial extent of the output feature map compared to the input. In these instances we use an operation referred to as a transposed convolution, a fractionally strided convolution, or, although not technically correct, deconvolution. This convolutional operation is referred to as transposed convolution because if we were to take an ordinary convolution operation, flatten (reshape an arbitrary tensor into a first order tensor or vector) its input feature map and reshape the kernel into a block circulant matrix $\mathbf{C}$, then the transposed convolution would be performed by applying the affine map defined by $\mathbf{C}^T$ to the input feature map.

1.2.3 Layer operations

1.2.3.1 Pooling

As an alternative to strided convolutions we show a variety of pooling techniques for sub-sampling feature maps for the purpose of enhancing the invariance properties of convolutional neural networks. Generally pooling operations are non-parametric or hyper-parametric operations that operate on successive windows of the input and yield a single element per window. A common pooling operation is max-pooling.

Given a neighborhood the max-pooling operation yields the maximum element. While this can be performed over any set of axis for images it is common to pool over windows spanning several elements in the height and width dimensions. Other common pooling operations include mean-pooling and L^p-pooling which returns the L^p norm of the neighborhood.

1.2.3.2 Dropout

Various teams have demonstrated that in deep networks neurons can learn complex co-adaptation schemes; that is deep neurons respond to "mistakes" in shallower neurons. In order to prevent co-adaptation, so that each neuron learns a meaningful representation, the dropout scheme has been proposed [8, 19, 20]. Dropout is a masking process. In order to apply dropout to a layer during training, a binary mask is applied across neurons. The mask is determined by sampling a Bernoulli distribution with a parameter p corresponding to the probability that a neuron will be masked. When a neuron is masked its output is considered fixed at zero. When the network is used for evaluation no masks are applied and all neurons are connected.

1.2.3.3 Batch normalization

Proposed to combat internal covariate shift (activation distributions drifting away from zero mean, unit variance), and help the gradients flow through a network, batch normalization is a simple method that normalizes features at inference time with accumulated statistics from the training period [21]. During the training period we describe the normalization process of a batch $\mathfrak{B} = \{x_1, x_2, \dots, x_m\}$ with the following steps:

$$\mu_{\mathfrak{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{47}$$

$$\sigma_{\mathfrak{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathfrak{B}})^2 \tag{48}$$

$$y_i \leftarrow g \frac{x_i - \mu_{\mathfrak{B}}}{\sqrt{\sigma_{\mathfrak{B}}^2 + \epsilon}} + b \tag{49}$$

where g and b are learnable parameters, and ϵ is some small constant value for numerical stability. At inference time we slightly modify the procedure to have statistics accumulated across the training period. Rather than normalizing against the batch mean and variance we use a average of the mean and variance across all batches from the data-set. Take note that from this formulation we see that batch normalization causes the outputs of the network to have an inter-example dependency graph, in other words the output for an example during training time is influenced by the other examples in the batch.

1.2.3.4 Layer normalization

An alternative to batch normalization, proposed for recurrent networks, is layer normalization which is fully defined to work on batches of size one, and is computed with same procedure at training and inference time [22]. We formulate layer normalization for a fully connected layer with the following steps:

$$\mu \leftarrow \frac{1}{H} \sum_{i=1}^{H} a_i \tag{50}$$

$$\sigma^2 \leftarrow \frac{1}{H} \sum_{i=1}^{H} (a_i - \mu)^2 \tag{51}$$

$$\mathbf{y} \leftarrow \frac{\mathbf{g}}{\sigma} \odot (\mathbf{a} - \mu) + \mathbf{b} \tag{52}$$

where a_i is the i^{th} neuron output, $\mathbf{g}$ and $\mathbf{b}$ are learned gain and bias vectors respectively, and $\odot$ is element-wise multiplication.

1.2.4 Training techniques

1.2.4.1 Variance scaling initializer

Kaiming et al. have demonstrated an improved parameter initialization technique specifically designed for neurons with ReLU activations [7]. This approach derives a method that enables extremely deep models comprised of ReLU neurons to converge rather than stall, as they would with other initialization schemes. The initial parameters for the convolutional layers are drawn from a zero mean normal distribution with a standard deviation of $\sqrt{2/n_l}$. Where $n_l = k^2c$. This corresponds to the number of connections in the response for $k \times k$ kernels processing c input channels.

1.2.4.2 Minibatch training

Traditionally there were two different approaches to optimizing neural network parameters, the online approach and the batch approach. In the online approach, the parameters of the network are updated after each exposure to a training example and gradient calculation. In the batch approach, the network is exposed to all of the training examples, the gradients are accumulated and then the network's parameters are updated. This was long believed to be the better approach, as the accumulated and averaged gradient was more likely to be an estimate of the "true" gradient of the network towards an optimized solution. It was later shown that, while the batch mode may give a better estimate of the gradient, due to the noise and its stochastic nature, the online approach actually leads to a faster convergence time, in terms of number of examples [23]. This is because in the stochastic approach the optimizer is less likely to get stuck in local minima. An alternative approach that recently has become popular is the mini-batch approach. In this approach some number of examples, say N, are exposed to the network, the gradients are accumulated, and the parameters are updated. In this case N is much less than the total number of training examples available. This approach, with serial computational resources really only represents a decrease in training speed, because it is fairly similar to the batch approach. The reason this approach has become popular lately is that with the parallel resources afforded by modern high performance graphics processing units

(GPUs) the increase in example-wise training time becomes a decrease in wall-time to train.

1.2.4.3 Adam optimizer

In order to train neural networks, classically ordinary minibatch gradient descent was sufficient but with more complicated networks with millions of parameters, other optimization algorithms have become necessary [24]. Adam, whose name is derived from adaptive moment estimation is a first order optimization algorithm that uses bias-corrected estimates to approximate the first two moments of the gradient. After initializing m_0, v_0, and t to zero, and selecting hyper-parameters α, β_1, β_2, the algorithm proceeds according to the following steps, and is repeated until convergence:

$$t \leftarrow t + 1 \tag{53}$$

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \tag{54}$$

$$m_t \leftarrow \beta_1 m_{t-1} + g_t(1 - \beta_1) \tag{55}$$

$$v_t \leftarrow \beta_2 v_{t-1} + g_t^2(1 - \beta_2) \tag{56}$$

$$\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t} \tag{57}$$

$$\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t} \tag{58}$$

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \tag{59}$$

1.2.5 Activation functions

Here we present several common activation functions, each of which is pertinent to the present work. A comparison of each of them can be seen in Figure 6.

1.2.5.1 Sigmoid

While we introduced it earlier, allow us to return to the sigmoid activation function:

$$f(x) = \frac{1}{1 + \exp(-x)} \tag{60}$$

and gradient:

$$f'(x) = \frac{\exp(x)}{(\exp(x) + 1)^2} \tag{61}$$

which has codomain $(0, 1/4]$ and is symmetric around zero. Because the gradients are guaranteed to be less than one, sigmoidal networks suffer from the vanishing gradient problem. If we recall that we compute gradients for the network using the chain rule, and therefore are taking products of gradients at each node, it becomes clear that having gradients with values less than one eventually lead to incredibly small gradients which can slow and stall learning.

1.2.5.2 Rectified linear units

Various activations have been proposed on the basis of similarity to the potential activations in actual biological neurons in human eyes. Recently the rectified linear unit (ReLU) has demonstrated significant performance increases for network generalization and increased training speed [8,25]. The ReLU activation is defined as $\max(0, x)$. In other words a ReLU activation forwards along any positive inputs and sets any negative inputs to zero. The aspect of increased training speed is attributed to the way ReLU's deal with the exploding and vanishing gradient problems. Following from the piece-wise linear definition of the ReLU it is easy to see that the only possible values for the gradient of the ReLU are zero and one, leading not just to fast gradient calculations but also stable gradients.

1.2.5.3 Exponential linear units

Exponential linear units (ELU) are an alternative activation to ReLU's which tend to have mean activations closer to zero and also deal well with the exploding and vanishing gradients problems [26]. The ELU with hyperparameter $\alpha > 0$ is

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \tag{62}$$

and gradient:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ f(x) + \alpha & \text{if } x \leq 0 \end{cases} \tag{63}$$

1.2.5.4 Leaky rectified linear units

LeakyReLUs are another alternative to the ReLU designed to have some non-zero (albeit small) gradient when the neuron is deactivated [27]. The LeakyReLU with hyperparameter $\alpha > 0$ is

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{64}$$

with gradient:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{65}$$

1.2.6 Higher level architectures

1.2.6.1 Residual networks

Residual networks are a particular class of neural network architectures first developed for image classification as an entry to the ILSVRC 2015 classification task

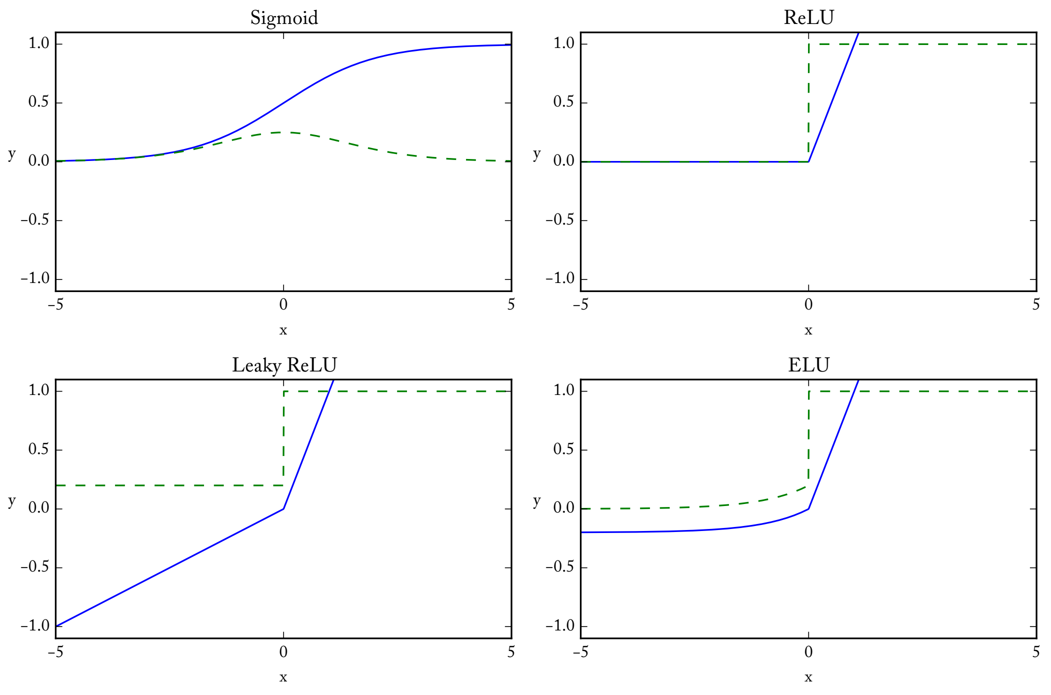


Figure 6: Comparison of various activation functions. Forward pass in blue, gradient in dashed green.

[28]. Consider a desired mapping $\mathcal{H}(\mathbf{x})$, and then we define some other mapping:

$$\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x} \tag{66}$$

With this mapping we can recast the original mapping:

$$\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x} \tag{67}$$

We can readily construct a graph consisting of blocks of mappings of this type. We can refer to each $\mathcal{F}(\cdot)$ as a residual as it is the difference between the desired mapping $\mathcal{H}(\cdot)$ and the input $\mathbf{x}$. A diagram of a simple residual block can be seen in Figure 7.

Residual networks have been shown to train more quickly than ordinary networks and can allow networks to have incredible depth (more than 1000 layers) however Zagoruyko et al. have actually shown that it is more beneficial to have shallower albeit wider residual networks than deep, thin networks [29].

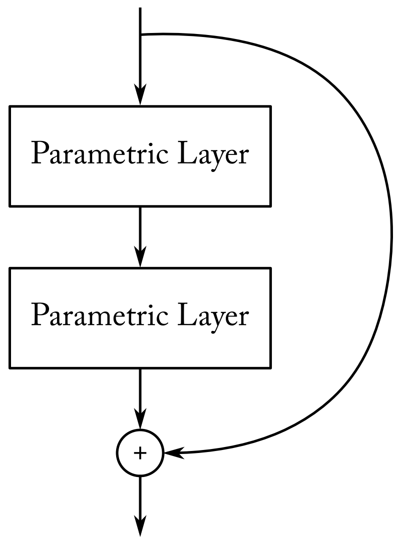


Figure 7: Residual block

1.3 Color representations

Considering a portion of the present work deals with coloring gray-scale photographs, we now present some material on the mathematical representations of color according the ITU-R BT.601-7 3 standard [30].

1.3.1 RGB

Consider $E_R, E_B, E_G \in [0, 1]$ which represent some scaling factors for the primary colors red, green, and blue. By mixing these primary colors (which correspond to specific physical colors, i.e. electromagnetic waves with specific wavelengths) we are able to represent a wide variety of colors. The set of colors we can represent with this 3 dimensioned vector space is referred to as the gamut of the space. A mixture of these colors emitted by three different approximately collocated light sources is processed by human brains and is perceived as a single physical color. This type of color scheme is referred to as an additive color scheme because it is physically renderable and when rendered if each E_R, E_B, E_G is at maximum intensity then the perceived color will be white.

1.3.2 YUV

An alternative to the RGB color space presented above, is the YUV color space which was developed to remove redundant information in an RGB encoded signal, so that it could transmitted over a channel with less bandwidth. In the YUV color space, Y refers to the luminance information (how bright a color is), the U refers to blue differential information, and the V refers to red differential information. To make this more clear, let us take a look at the equations to convert from RGB to YUV:

$$W_R := 0.299 \tag{68}$$

$$W_B := 0.114 \tag{69}$$

$$W_G := 1 - W_R - W_B \tag{70}$$

$$U_{\text{Max}} := 0.436 \tag{71}$$

$$V_{\text{Max}} := 0.615 \tag{72}$$

$$E_Y = W_R E_R + W_G E_G + W_B B \tag{73}$$

$$E_U = U_{\text{Max}} \frac{E_B - E_Y}{1 - W_B} \tag{74}$$

$$E_V = V_{\text{Max}} \frac{E_R - E_Y}{1 - W_R} \tag{75}$$

We can see that the luminance is a scaled sum of the red, green, and blue channels, and according to the standard these scale factors are based on human perceptual sensitivity to each of the primary colors. Meanwhile the U and V channels encode a scaled difference of the red and blue weights from the luminance. Therefore this allows us to separate the core details of an image from its chrominance. Figure 8 shows this clearly; the majority of the semantic information is contained in the luminance component.

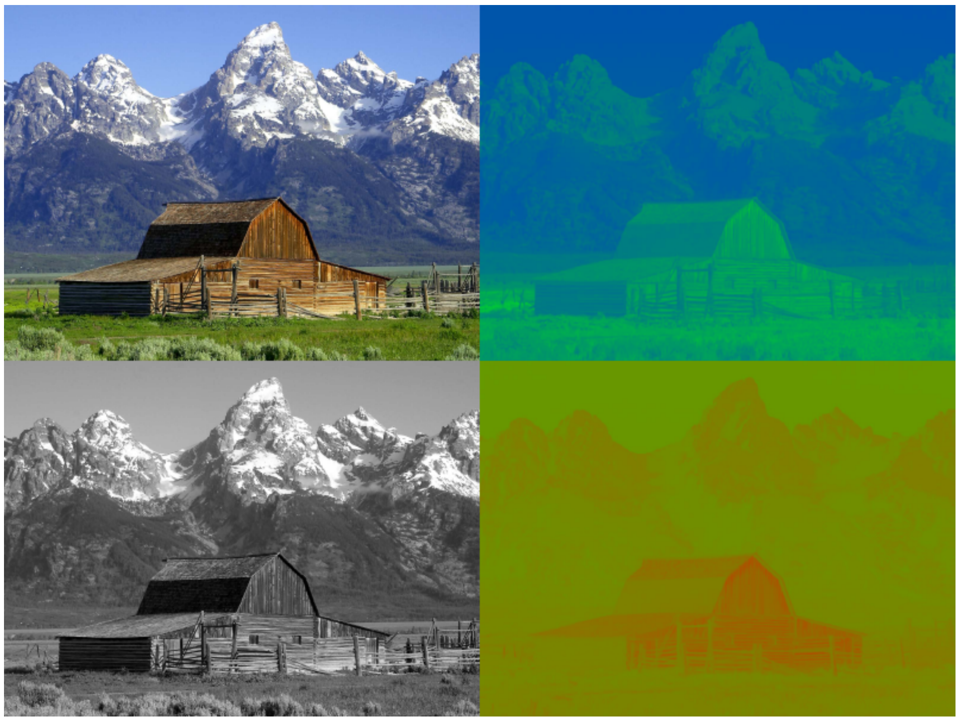


Figure 8: The photograph in the upper left is broken down into its luminance, blue differential, and red differential components. Counter clockwise from upper left. Source photo is in the public domain, photographed by Jon Sullivan, PD Photo.

2 Problem Statement and Related Works

2.1 Problem statement

2.1.1 Generative modeling

Generative models either explicitly or implicitly model the distribution of values for some data-set. We can sample from generative models to create synthetic data because they model the distribution of the true data. A good generative model is able to create synthetic data that is indistinguishable from true data, and as such could be used to create a larger data-set for supervised or semi-supervised learning tasks. In the present work we have taken steps towards high-quality generative models for natural images.

2.2 Deep generative models

2.2.1 Generative adversarial networks

Goodfellow et al. proposed the framework of generative adversarial networks for training deep neural networks as generative models, with particular focus on natural images [31]. Consider learning a generator's distribution p_g over data $\mathbf{x}$, we define a prior $p_{\mathbf{z}}(\mathbf{z})$, a mapping $\mathcal{G}(\mathbf{z} \mid \theta_{\mathcal{G}})$, where is $\mathcal{G}(\cdot)$ is a differentiable function with parameters $\theta_{\mathcal{G}}$. We define a second mapping $\mathcal{D}(\mathbf{x} \mid \theta_{\mathcal{D}})$ that outputs a single scalar which represents the probability that $\mathbf{x}$ is true data rather synthetic data. We train $\mathcal{D}(\cdot)$ to tell the difference between synthetic samples from $\mathcal{G}(\cdot)$ and true samples, while training $\mathcal{G}(\cdot)$ to fool $\mathcal{D}(\cdot)$. Formally they play the following two-player minimax game with value function $V(\mathcal{G}, \mathcal{D})$:

$$\min_{\mathcal{G}} \max_{\mathcal{D}} V(\mathcal{G}, \mathcal{D}) = \mathbb{E}_{x \sim p_{\text{data}}(\mathbf{x})} [\log \mathcal{D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - \mathcal{D}(\mathcal{G}(\mathbf{z})))] \tag{76}$$

Practically, we approximate this solution with an iterative algorithm by taking alternating turns optimizing $\mathcal{G}(\cdot)$ and $\mathcal{D}(\cdot)$ in step, as adversaries.

While generative adversarial networks have shown state of the art results results on MNIST digits [32] and the TFD faces [33], they could not extend to CIFAR 10 [34] or any other more complicated data-sets. The success of generative adversarial models was limited to small, low complexity images, and even then they could not produce examples fully indistinguishable from true data.

2.2.2 Deep convolutional generative adversarial networks

In the original formulation $\mathcal{D}(\cdot)$ and $\mathcal{G}(\cdot)$ took the form of fully-connected networks. Radford et al. proposed an alternative architecture for $\mathcal{G}(\cdot)$ that was able to produce much higher quality samples than the original implementation, they called this class of models deep convolutional generative adversarial networks [35]. They came up

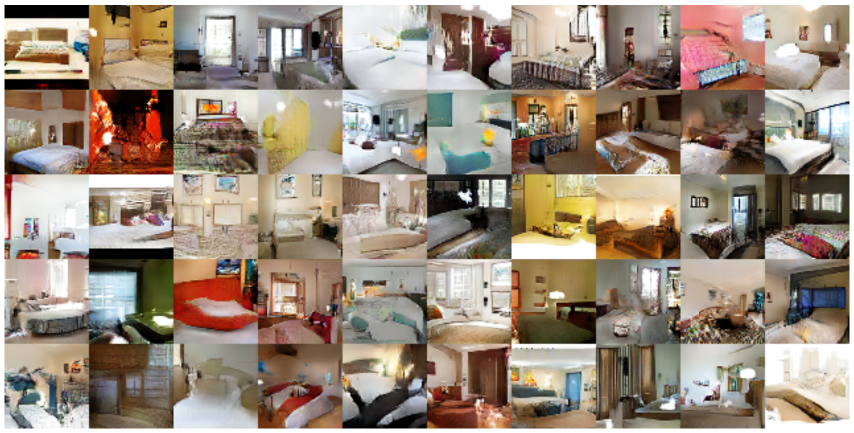


Figure 9: Sampled bedrooms from a deep convolutional generative adversarial network trained on LSUN bedrooms [35].

with a set guidelines for constructing architectures which showed good practical results for the LSUN data-set [36], and a purpose-collected face data-set, namely:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator)
- Use batch normalization in both the generator and the discriminator
- Remove fully connected hidden layers for deeper architectures
- Use ReLU activation in generator for all layers except for the output, which uses sigmoid
- Use LeakyReLU activation in the discriminator for all layers

2.2.3 Additional techniques for training generative adversarial networks

Salimans et al. furthered the art of generative adversarial networks by introducing several new techniques at once that created samples of the highest quality for MNIST and TFD, and new state of the art samples for ImageNet-1k [37, 38].

2.2.3.1 Feature matching

Feature matching adds an additional term to the loss function for the generator, which attempts to cause the synthetic data samples to have similar features to the true data samples. Taking some set of activations $\mathcal{F}^j(\cdot)$ from an intermediate layer of $\mathcal{D}(\cdot)$, we define this new loss term as:

$$l_{\text{feats}}(\mathbf{x}, \mathbf{z}) = \left\| \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \mathcal{F}^j(\mathbf{x}) - \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} \mathcal{F}^j(\mathcal{G}(\mathbf{z})) \right\|_2^2 \tag{77}$$

where $\mathbf{x}$ and $\mathbf{z}$ are minibatches of data and noise respectively, the expectations are approximated as means over those minibatches, and $\|\cdot\|_2$ is the L^2 norm. The core assumption here is that if the expected value for the features over the true data is the same as the synthetic data that those synthetic data represent high quality samples. This is purely a first order term, and does not take into account higher order moments.

2.2.3.2 Minibatch discrimination

One failure mode for generative adversarial networks is for the generator to collapse to produce only one output for any given input. In order to encourage varied samples we can allow the discriminator to make predictions utilizing side information, that is we can allow the discriminator to look at other examples within a batch and use them to determine whether or not the example under inspection is a true sample or a forgery. Consider the following: let $\mathcal{F}(\mathbf{x}_\text{i}) \in \mathbb{R}^A$ denote a vector of features for some input $\mathbf{x}_i$ in some minibatch produced by some layer of the discriminator. We multiply $\mathcal{F}(\mathbf{x}_i)$ by some tensor $\mathsf{T} \in \mathbb{R}^{A \times B \times C}$ (consisting of learned parameters) resulting in a matrix $\mathbf{M}_i \in \mathbb{R}^{B \times C}$. We define $c_b(\mathbf{x}_i, \mathbf{x}_\text{j})$:

$$c_b(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\|m_{ib} - m_{jb}\|_1\right) \tag{78}$$

where $\|\cdot\|_1$ is the L^1 norm. Using this definition we construct our minibatch information $o(\mathrm{X})$:

$$o(\mathbf{x}_i)_b = \sum_{j=1}^{n} c_b(\mathbf{x}_i, \mathbf{j}) \tag{79}$$

$$o(\mathbf{x}_i) = [o(\mathbf{x}_i)_1, o(\mathbf{x}_i)_2, \ldots, o(\mathbf{x}_i)_B] \tag{80}$$

$$o(\mathrm{X}) = \begin{bmatrix} o(\mathbf{x}_1) \\ o(\mathbf{x}_2) \\ \vdots \\ o(\mathbf{x}_n) \end{bmatrix} \tag{81}$$

We then concatenate $o(\mathrm{X})$ to $\mathcal{F}(\mathrm{X})$ and feed the resulting feature map down the rest of the network.

2.2.3.3 Historical averaging

An additional term can be added to the loss function of either the generator or discriminator to penalize large steps and oscillations that don't have large effects on the terms of the loss functions. We define:

$$l_{\text{hist}} = \left\| \boldsymbol{\theta} - \frac{1}{t} \sum_{i=1}^{t} \boldsymbol{\theta}[i] \right\|_2^2 \tag{82}$$

where $\boldsymbol{\theta}$ is a flattened vector of parameters at the present time step, $\boldsymbol{\theta}[i]$ is a flattened vector of parameters at time step i, t is the present time step, and $\|\cdot\|_2$ is the L^2 norm.

2.2.3.4 One-sided label smoothing

Label smoothing or softening represents an adjustment in a target value, in order to change the equilibrium point for optimal decision making [39]. For generative adversarial networks we smooth only the positive labels to some α and leave the negative labels set to zero, leading to an optimal discriminator being:

$$D_{\text{opt}}(\mathbf{x}) = \frac{\alpha\, p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_{\text{synth}}(\mathbf{x})} \tag{83}$$

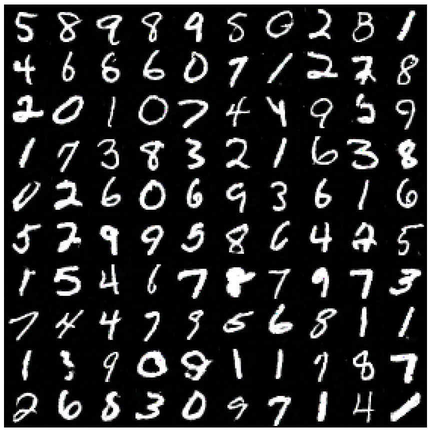


Figure 10: Sample MNIST digits trained with minibatch discrimination, virtual batch normalization, and one-sided label smoothing [37].

The net effect of this is that the loss function allows some uncertainty in the predictions for true examples, essentially slowing down how quickly true examples retreat from the decision boundary in the feature space.

2.2.3.5 Virtual batch normalization

While batch normalization greatly improves the performance of neural networks, it causes the output of the network to be highly dependent of the other examples within the batch, which can cause unpredictable steps during training time, and is problematic for relatively unstable algorithms like generative adversarial networks. Therefore virtual batch normalization offers an alternative, where instead of normalizing a batch against other examples within the batch, we preselect and fix a reference batch and always use that batch for normalization, at every layer in the network. This comes at a high computational cost so it must be used intelligently. Salimans et al. tended to only use it for the generator network and not for the discriminator.

2.3 Style transfer models

Consider the task of the artistic style transfer: given a photograph, manipulate it such that its semantic content remains constant but it appears in the style of a specific artist. Gatys et al. developed a neural algorithm for style transfer that was a major breakthrough for this task [40]. We see several examples of style transfer from their work in Figure 11.

Consider an arbitrary photograph $\mathsf{P} \in \mathbb{R}^{H_\mathsf{P} \times W_\mathsf{P} \times 3}$, a photograph of a piece of art $\mathsf{A} \in \mathbb{R}^{H_\mathsf{A} \times W_\mathsf{A} \times 3}$, a noise tensor $\mathsf{Z} \in \mathbb{R}^{H_\mathsf{P} \times W_\mathsf{P} \times 3}$, and finally the convolutional layers of a neural network $\mathcal{F}(\cdot)$ pretrained on some other vision task such as the VGG-Net [41]. Given these pieces, Gatys et al. use the network to measure the content similarity of Z and P, and to measure the style similarity of Z and A. They update Z such as to jointly maximize these too similarity measures. If we formulate the similarity as a loss (identical content or style results in zero loss), then mathematically this process equates to:

$$\arg\min_{\mathsf{Z}} \; l_{\text{style}}(\mathsf{A}, \mathsf{Z}) + l_{\text{content}}(\mathsf{P}, \mathsf{Z}) \tag{84}$$

Of the two terms the content loss is simpler, so let us consider that first. Assuming that feature map values directly correspond to semantic information we can measure content loss at a depth j by computing the mean squared difference between the pair of activations. If we consider an intermediate layer of the network $\mathcal{F}^j(\cdot) \in \mathbb{R}^{H_j \times W_j \times C_j}$, we can write the content loss at a layer j:

$$l^j_{\text{content}}(\mathsf{P}, \mathsf{Z}) = \left\| \mathcal{F}^j(\mathsf{P}) - \mathcal{F}^j(\mathsf{Z}) \right\|_F^2 \tag{85}$$

where $\|\cdot\|_F$ is the Frobenius norm. Then as their whole content loss they take a linear combination of the content losses at some selection of layers:

$$l_{\text{content}}(\mathsf{P}, \mathsf{Z}) = \sum_j w_j l^j_{\text{content}}(\mathsf{P}, \mathsf{Z}) \tag{86}$$

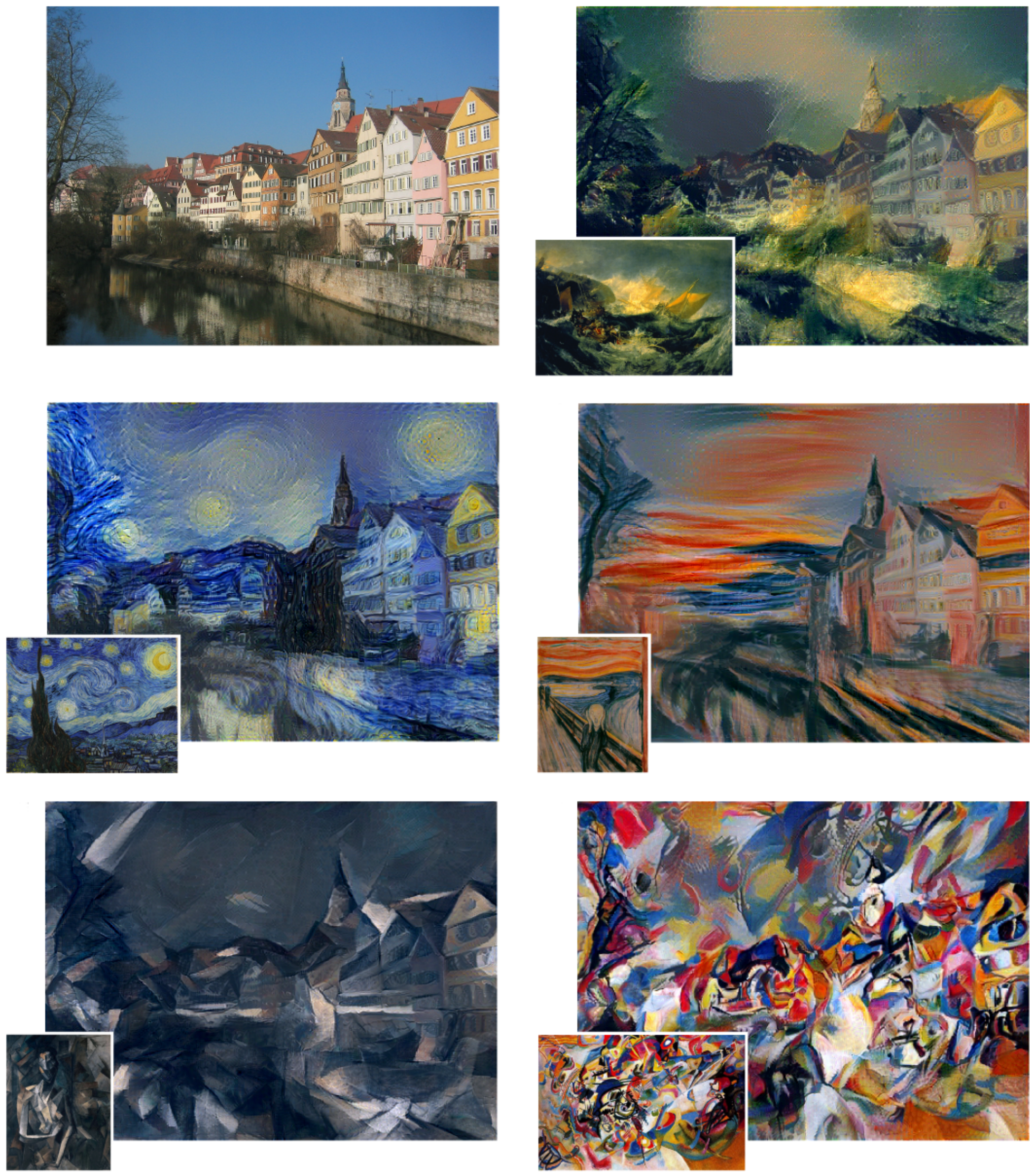


Figure 11: Several examples of artistic style transfer [40]. Paintings include *The Shipwreck of the Minotaur* by J.M.W. Turner, 1805; *The Starry Night* by Vincent van Gogh, 1889; *Der Schrei* by Edvard Munch, 1893; *Femme nue assise* by Pablo Picasso, 1910; *Composition VII* by Wassily Kandinsky, 1913. Original photograph by Andreas Praefcke.

Now consider the style loss term which incorporates information about features that activate together; that is by measuring which features tend to fluctuate together we can gain insight into the "style" of a feature map. Let $\mathsf{F}^j \in \mathbb{R}^{H_j \times W_j \times C_j}$ be the activations at the j^{th} layer of the network. We define the Gramm matrix $\mathbf{G}^j$ to be the $C_j \times C_j$ matrix with elements:

$$\mathbf{G}^j_{cc'} = \frac{1}{C_j H_J W_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \mathsf{F}^j_{hwc} \mathsf{F}^j_{hwc'} \tag{87}$$

The Gramm matrix can be easily computed by reshaping F^j into a matrix ψ of shape $C_j \times H_j W_j$; then $\mathbf{G}^j = \psi\psi^T / C_j H_j W_j$ [42]. Then we define the style loss term to be the following:

$$l^j_{\text{style}} = \left\| \hat{\mathbf{G}}^j - \mathbf{G}^j \right\|_F^2 \tag{88}$$

where $\mathbf{G}^j$ denotes the target style, $\hat{\mathbf{G}}^j$ is the inferred style, and $\|\cdot\|_F$ is the Frobenius norm. Take note that the style loss is well defined for feature maps of different sizes so long as they they have the same depth. Therefore we can work with style examples of different sizes than the inferred feature maps. Furthermore as with the content loss, we can compute the style loss for multiples layers $j \in \mathfrak{J}$, and take a linear combination of the terms to have style computed against features at different levels of abstraction.

3 Full Image Generation

3.1 Generator loss terms

3.1.1 Kernel based moment matching

In Section 1.1.3 we introduce kernel methods as a way of measuring the similarity between two vectors $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. Here we revisit kernels as a way to formulate a term in the loss function for generative adversarial networks. First though let us consider the usual generative adversarial loss formulation:

$$l_{\mathcal{G}}(\mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} \log \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)) \tag{89}$$

$$l_{\mathcal{D}}(\mathsf{X}, \mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} [\log \mathcal{D}(\mathsf{X}_i) + \log(1 - \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)))] \tag{90}$$

where X is a minibatch of true data examples, X_i is the i^{th} example, Z is a minibatch of noise vectors, and N is the number of examples in a minibatch.

We introduce here another term for the generator loss function $l_{\mathcal{G}}(\cdot)$ utilizing kernel functions. Consider the a^{th} layer of $\mathcal{D}(\cdot)$, which we denote as $\mathcal{D}^a(\cdot)$, $\mathcal{D}^a_k(\cdot)$ denoting the k^{th} activation of that layer. We can measure the similarity between two minibatches of data, one true and one synthetic with the following two sample test statistic:

$$\begin{aligned}\psi_a(\mathsf{X}, \mathcal{G}(\mathsf{Z})) = \frac{1}{N^2}\sum_i\sum_j\sum_k \mathcal{D}^a_k(\mathsf{X}_j)\mathcal{D}^a_k(\mathsf{X}_i) + \\ + \frac{1}{N^2}\sum_i\sum_j\sum_k \mathcal{D}^a_k(\mathcal{G}(\mathsf{Z}_j))\mathcal{D}^a_k(\mathcal{G}(\mathsf{Z}_i)) + \\ - \frac{2}{N^2}\sum_i\sum_j\sum_k \mathcal{D}^a_k(\mathsf{X}_j)\mathcal{D}^a_k(\mathcal{G}(\mathsf{Z}_i))\end{aligned} \tag{91}$$

This statistic is strictly linear. We notice that the argument of the two outer sums, for each term in the statistic, is an inner product over the pair of feature maps. We therefore can introduce a nonlinear variant utilizing the kernel trick:

$$\begin{aligned}\psi_a(\mathsf{X}, \mathcal{G}(\mathsf{Z}) \mid \sigma) = \frac{1}{N^2}\sum_i\sum_j \exp\left(-\frac{1}{2\sigma}\|\mathcal{D}^a_k(\mathsf{X}_j) - \mathcal{D}^a_k(\mathsf{X}_i)\|^2\right) + \\ + \frac{1}{N^2}\sum_i\sum_j \exp\left(-\frac{1}{2\sigma}\|\mathcal{D}^a_k(\mathcal{G}(\mathsf{Z}_j)) - \mathcal{D}^a_k(\mathcal{G}(\mathsf{Z}_i))\|^2\right) + \\ - \frac{2}{N^2}\sum_i\sum_j \exp\left(-\frac{1}{2\sigma}\|\mathcal{D}^a_k(\mathsf{X}_j) - \mathcal{D}^a_k(\mathcal{G}(\mathsf{Z}_i))\|^2\right)\end{aligned} \tag{92}$$

where σ is a bandwidth hyper-parameter. By looking at this equation together with the series expansion for e^X:

$$e^X = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \cdots + \frac{X^n}{n!} \tag{93}$$

and recalling that $\mathbb{E}(X^n)$ is the n^{th} moment of a random variable X, we can see that Equation 92 calculates a sampled difference between *all* of the moments of the respective feature maps. The only condition under which this statistic approaches zero is if the distribution of $\mathcal{D}^a(\mathcal{G}(\mathsf{Z}))$ approaches the distribution of $\mathcal{D}^a(\mathsf{X})$.

This statistic is relatively sensitive to the value of the bandwidth hyper-parameter, and can render the statistic useless if selected poorly. While it may be possible to search for an optimal value we instead take to utilizing a linear combination of the statistics with varying bandwidths:

$$l_{\text{moments}}(\mathsf{X}, \mathsf{Z}) = \sum_a \sum_j w_{aj} \psi_a(\mathsf{X}, \mathcal{G}(\mathsf{Z}) \mid \sigma_j) \tag{94}$$

With this we can rewrite our new adversarial loss functions:

$$l_{\mathcal{G}}(\mathsf{X}, \mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} \log \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)) + \lambda \sum_a \sum_j w_{aj} \psi_a(\mathsf{X}, \mathcal{G}(\mathsf{Z}) \mid \sigma_j) \tag{95}$$

$$l_{\mathcal{D}}(\mathsf{X}, \mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} [\log \mathcal{D}(\mathsf{X}_i) + \log(1 - \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)))] \tag{96}$$

where λ is a hyper-parameter.

3.1.2 Style loss term

Let us recall our formulation for style loss, and reformulate it to work for a batch of true data examples and a batch of forgeries. Consider two tensors $\mathcal{D}^j(\mathsf{X}_i), \mathcal{D}^j(\mathcal{G}(\mathsf{Z}_i)) \in \mathbb{R}^{H \times W \times C_j}$ each consisting of a batch of feature maps from some intermediate layer at j depth in the discriminator $\mathcal{D}(\cdot)$. Substituting these tensors in as appropriate we reach the following loss function:

$$l^j_{\text{style}}(\mathsf{X}, \mathcal{G}(\mathsf{Z})) =$$
$$\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C_j}\sum_{c'=1}^{C_j}\sum_{h=1}^{H_j}\sum_{w=1}^{W_j} \left|\mathcal{D}^j_{hwc}(\mathsf{X}_i)\mathcal{D}^j_{hwc'}(\mathsf{X}_i) - \mathcal{D}^j_{hwc}(\mathcal{G}(\mathsf{Z}_i))\mathcal{D}^j_{hwc'}(\mathcal{G}(\mathsf{Z}_i))\right|^2 \tag{97}$$

Then we take a linear combination of these style losses at different levels of abstraction to reach:

$$l_{\text{style}}(\mathsf{X}, \mathcal{G}(\mathsf{Z})) = \sum_j w_j l^j_{\text{style}}(\mathsf{X}, \mathcal{G}(\mathsf{Z})) \tag{98}$$

which we can incorporate into the ordinary generative adversarial loss formulation:

$$l_{\mathcal{G}}(\mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} \log \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)) + \lambda \sum_j w_j l^j_{\text{style}}(\mathsf{X}, \mathcal{G}(\mathsf{Z}) \tag{99}$$

$$l_{\mathcal{D}}(\mathsf{X}, \mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} [\log \mathcal{D}(\mathsf{X}_i) + \log(1 - \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)))] \tag{100}$$

where λ is a hyper-parameter.

3.2 Modified discriminator with hinge loss

We again visit the familiar adversarial loss formulation:

$$l_{\mathcal{G}}(\mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} \log \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)) \tag{101}$$

$$l_{\mathcal{D}}(\mathsf{X}, \mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} [\log \mathcal{D}(\mathsf{X}_i) + \log(1 - \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)))] \tag{102}$$

In contrast to our previous techniques, let us focus explicitly on the discriminator loss function. Each term in the discriminator loss function penalizes probability estimates that are not precisely the true label which is either zero or one. Therefore the discriminator has no notion of sufficiency, and it will continue learning to separate true examples from forgeries as much as possible, regardless of whether or not the generator is keeping up with its learning rate. As such we introduce a modified version of the discriminator loss function such that when examples cross a particular threshold it drops their cost to zero. This way correctly classified examples will retreat more slowly from the threshold, as they will contribute only the zero vector to the accumulated and averaged gradients. This gives the generator the opportunity to keep up with the discriminator. This new hinge loss based function follows:

$$l_{\mathcal{D}}(\mathsf{X}, \mathsf{Z}) = -\frac{1}{N}\sum_{i=1}^{N} [\max(0, \alpha - \mathcal{D}(\mathsf{X}_i)) + \max(0, \alpha + \mathcal{D}(\mathcal{G}(\mathsf{Z}_i)))] \tag{103}$$

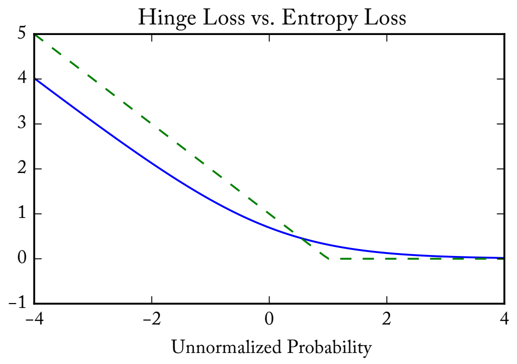


Figure 12: A comparison of the hinge loss function ($\max(0, 1-p)$) to binary cross entropy loss ($\log(1/(1+\exp(-p)))$) with unnormalized probabilities as the input. We can see that hinge loss penalizes up to 0.73 on a normalized probability scale.

where $\alpha > 0$ is an unnormalized probability threshold parameter determining what the positive or negative label is; for ordinary hinge loss $\alpha = 1$.

3.3 Results

Here we present our results for full image generation, as well as the details for training the models.

3.3.1 MNIST

We trained on the full MNIST data-set, and did not include any class conditional information, so our approach was fully unsupervised. For all MNIST models the discriminator networks were composed of six layer deep fully connected networks, with two instances of minibatch discrimination layers preceding the final two dense layers. All layers except the output were normalized using layer normalization from §1.2.3.4. ELU activations were used for all layers except for the output, where a sigmoid was used. White gaussian noise was added at each layer with a standard deviation of 0.3 for regularization. The generator consisted of a single fully connected layer followed by a series of transposed convolution layers. Layer normalization with ELUs was used for each layer except the output layer which used a sigmoid. The

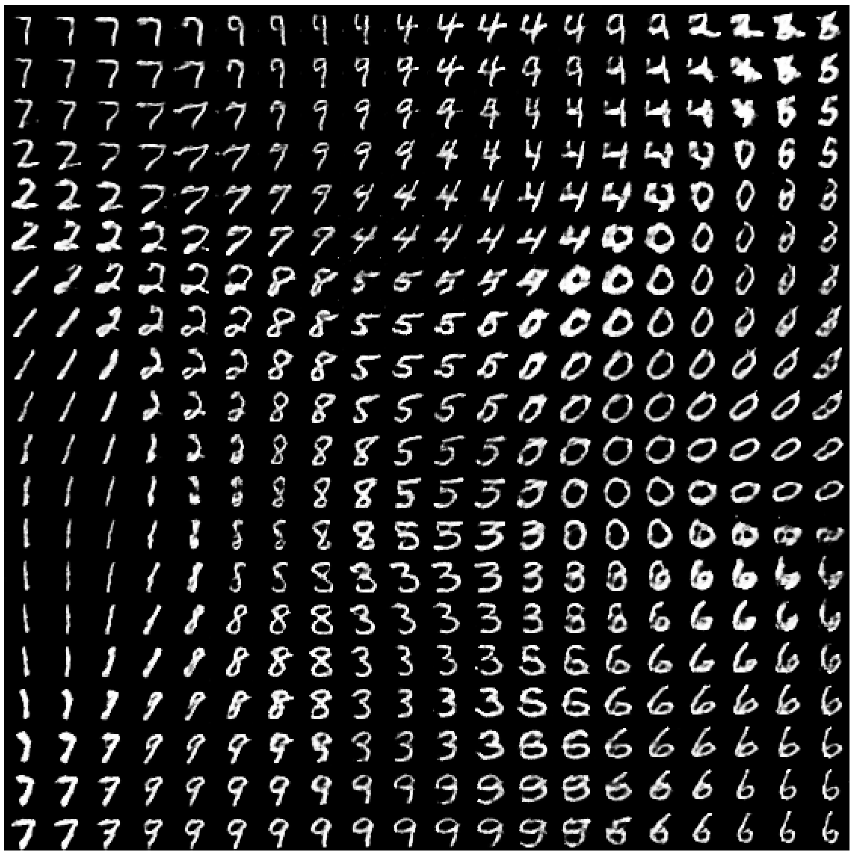


Figure 13: Example MNIST manifold sampled from $\mathcal{G}(\mathsf{Z})$ with Z consisting of values uniformly spaced in $\mathbb{R}^2$ over the range $[-1, -1]$. This generator was trained with $p_{\mathbf{z}}(\mathbf{z})$ as a 2 dimensional uniform distribution over $[-1, 1]$.

Adam optimizer was used to minimize both loss functions. We performed two updates on the generator network for every one discriminator update. The noise prior $p_{\mathbf{z}}(\mathbf{z})$ a was always a uniform distribution over $[-1, 1]$, however for our experiments demonstrating manifolds we used a two dimensional uniform distribution, for all other experiments we used a one hundred dimensional uniform distribution.

3.3.2 MIT Places

We trained on the building facade category of the MIT Places data-set resized to 64×64 pixels. For all MIT Places models the discriminator network consisted of several convolution layers followed by two fully connected layers, with one instance of minibatch discrimination. Batch normalization was used in all convolution layers,

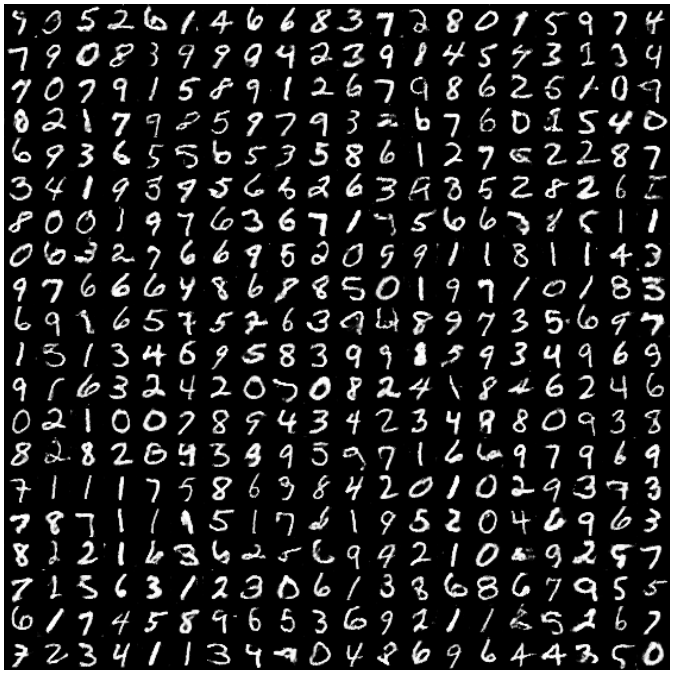


Figure 14: Sample of generated images using kernel based moment matching, style loss on the generator, and hinge loss on the discriminator, trained on the MNIST digits. $p_{\mathbf{z}}(\mathbf{z})$ was a 100 dimensional uniform distribution over $[-1, -1]$.

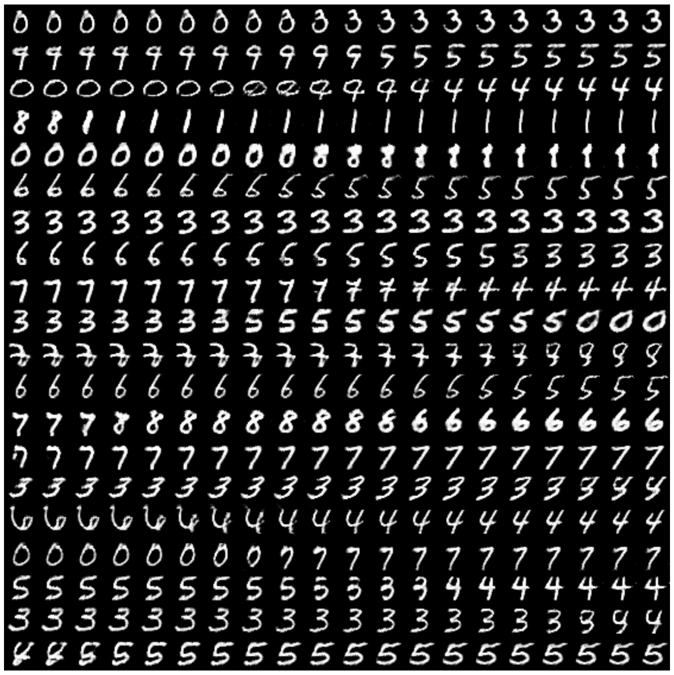


Figure 15: Sample of generated images using kernel based moment matching, style loss on the generator, and hinge loss on the discriminator, trained on the MNIST digits. $p_{\mathbf{z}}(\mathbf{z})$ was a 100 dimensional uniform distribution over $[-1, -1]$. Digits on the end columns are random samples from the generated distribution, all other columns are selected to linearly interpolate between the two end samples.

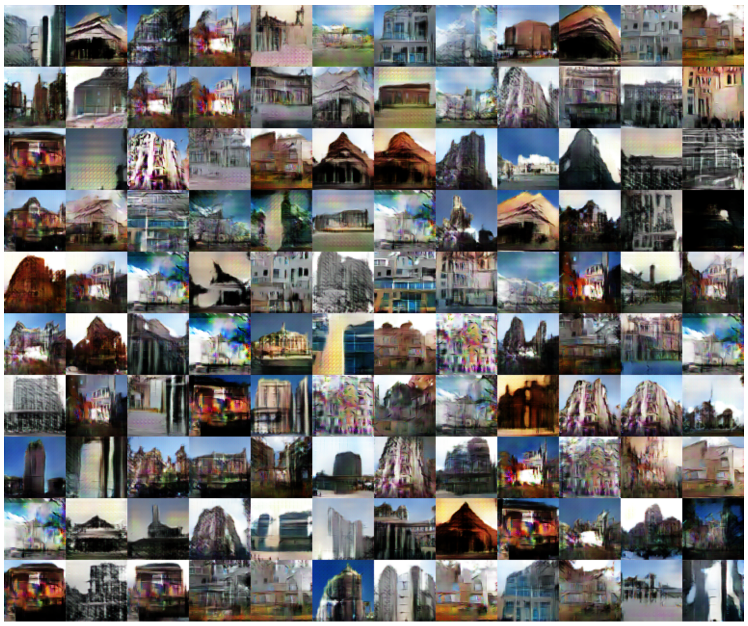


Figure 16: Sample of generated images using kernel based moment matching and style loss, trained on the MIT Place data-set using the "building facade" category.

while dropout was used in the fully connected layers for some experiments. ELUs and ReLUs were both used for various experiments, except on the output layer where sigmoids were used. The generator consisted of one fully connected layer followed by several transposed convolution layers each with batch-normalization and ReLUs, however sigmoids were used on the output layer. The Adam optimizer was used to minimize both loss functions. We performed two updates on the generator network for every one discriminator update. The noise prior $p_{\mathbf{z}}(\mathbf{z})$ a was always a one hundred dimensional uniform distribution over $[-1, 1]$.

4 Color Generation

We now divert our attention from focusing on full image generation to present work on image enhancement. Consider a single-channel gray-scale image $\mathbf{Y} \in \mathbb{R}^{H \times W}$ containing only luminance characteristics and no chrominance information. Given this image $\mathbf{Y}$ we attempt to infer the YUV color-space chrominance components

$\mathbf{U} \in \mathbb{R}^{H \times W}$ and $\mathbf{V} \in \mathbb{R}^{H \times W}$ packed into a single tensor $\mathsf{C} \in \mathbb{R}^{H \times W \times 2}$:

$$\mathsf{C} = f(\mathbf{Y}) \tag{104}$$

Packing $\mathbf{Y}$ and C results in the final YUV image $\mathsf{A} \in \mathbb{R}^{H \times W \times 3}$ which can be converted to RGB for viewing. To frame this in terms of generative adversarial networks, let us consider a generator network $\mathcal{G}(\cdot)$ and a discriminator network $\mathcal{D}(\cdot)$, and adversarial loss functions:

$$l_{\mathcal{G}}(\mathsf{Y}) = -\frac{1}{N} \sum_{i=1}^{N} \log \mathcal{D}(\mathcal{G}(\mathsf{Y}_i)) \tag{105}$$

$$l_{\mathcal{D}}(\mathsf{A}, \mathsf{Y}) = -\frac{1}{N} \sum_{i=1}^{N} [\log \mathcal{D}(\mathsf{A}_i) + \log(1 - \mathcal{D}(\mathcal{G}(\mathsf{Y}_i)))] \tag{106}$$

where $\mathsf{A} \in \mathbb{R}^{N \times H \times W \times 3}$ is drawn from some set of images $\mathfrak{D}_1$, and $\mathsf{Y} \in \mathbb{R}^{N \times H \times W}$ is drawn from some other fully distinct set of images $\mathfrak{D}_2$, in other words this is a fully unsupervised approach.

The generator network $\mathcal{G}(\cdot)$ consists of residual dilated convolution blocks and is defined by the following recurrence relation:

$$\mathsf{G}^l = \begin{cases} f(l(C(\mathsf{K}^l, \mathsf{G}^{l-1}, r = 2^l) + \mathsf{B}^l + \mathsf{G}^{l-2})) & \text{if } (l \geq 2) \wedge (l \bmod 2 = 0) \\ f(l(C(\mathsf{K}^l, \mathsf{G}^{l-1}, r = 2^l) + \mathsf{B}^l)) & \text{otherwise} \end{cases} \tag{107}$$

where G^l is a tensor from $\mathcal{G}(\cdot)$ at an intermediate layer of depth l, G^0 is the input $\mathsf{Y} \in \mathbb{R}^{N \times H \times W}$, $C(\cdot, \cdot, \cdot)$ is an instance of dilated convolution, r is the dilation rate factor, $f(\cdot)$ is an activation function, and $l(\cdot)$ is an instance of layer normalization. Let us note us that the residual formulation with constant spatial extent is a natural approach because the chrominance characteristics of an image are closely related to the luminance characteristics in terms of their visual distribution, as seen in the earlier Figure 8.

4.0.1 Results

We trained on a subset of the CelebA faces data-set [43]. For this task the discriminator network was composed of a series of convolutional layers followed by two dense layers. All layers except the output were normalized using layer normalization from §1.2.3.4. ELU activations were used for all layers except for the output, where a sigmoid was used. White gaussian noise was added at each layer with a standard deviation of 0.3 for regularization. The generator consisted entirely of residual dilated convolutional blocks followed by one regular convolutional layer. Layer normalization with ELUs was used for each layer except the output layer which used a Tanh. The Adam optimizer was used to minimize both loss functions. We performed two updates on the generator network for every one discriminator update.

Looking at Figure 17 we can see a sample of colored photographs from the CelebA faces data-set. It performs very well on the skin portions of the faces for a majority of the photographs, with a few main failure cases. We notice that for the most notable skin failure cases the result is a completely desaturated image, which may result from the use of a residual dilated convolutional network, which can learn to forward its input to its output. We also notice that the network has difficultly assigning colors to objects where their semantic function is independent of their color, for example articles of clothing, with the exception of men's suit jackets and similar articles where it is able to assign a dark color. Likewise the network has difficultly assigning realistic background colors, where we see significant splotching and bleeding.

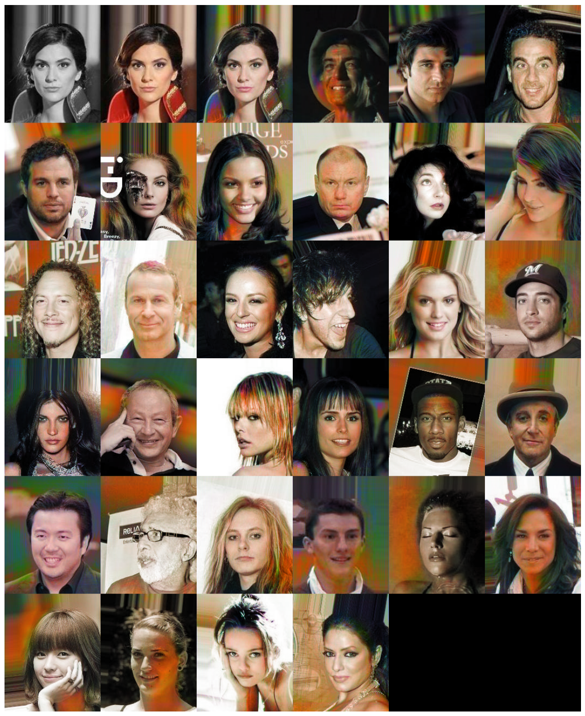


Figure 17: Sample of generated coloring for gray-scale photographs, upper left shows gray-scale and ground truth image for first sample.

5 Conclusion

We have presented a variety of techniques taking steps towards generative models capable of producing high quality samples for natural images. We have shown strong results in full image generation similar to the state of the art methods for MNIST, and presented the first instance of generative experimentation on the MIT Places data-set. Further we have demonstrated the first steps towards high quality end-to-end neural network image colorization.

References

[1] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[2] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[3] W. Xu, “Towards optimal one pass large scale learning with averaged stochastic gradient descent,” *CoRR*, vol. abs/1107.2490, 2011. [Online]. Available: http://arxiv.org/abs/1107.2490

[4] Y. Li, K. Swersky, and R. Zemel, “Generative moment matching networks,” in *International Conference on Machine Learning*, 2015, pp. 1718–1727.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[7] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[8] G. E. Dahl, T. N. Sainath, and G. E. Hinton, “Improving deep neural networks for lvcsr using rectified linear units and dropout,” in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8609–8613.

[9] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, “Automatic differentiation in machine learning: a survey,” *CoRR*, vol. abs/1502.05767, 2015. [Online]. Available: http://arxiv.org/abs/1502.05767

[10] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*. Springer, 1998, pp. 9–48.

[11] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: A cpu and gpu math

compiler in python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 3 – 10.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[13] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[14] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980. [Online]. Available: http://dx.doi.org/10.1007/BF00344251

[15] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[16] "Broadcasting — numpy v1.11 manual," http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html, (Accessed on 08/30/2016).

[17] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected crfs," *CoRR*, vol. abs/1412.7062, 2014. [Online]. Available: http://arxiv.org/abs/1412.7062

[18] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *CoRR*, vol. abs/1511.07122, 2015. [Online]. Available: http://arxiv.org/abs/1511.07122

[19] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[20] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation

of feature detectors," *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[22] J. Lei Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *ArXiv e-prints*, Jul. 2016.

[23] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural Netw.*, vol. 16, no. 10, pp. 1429–1451, Dec. 2003. [Online]. Available: http://dx.doi.org/10.1016/S0893-6080(03)00138-2

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[25] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, vol. 15, 2011, pp. 315–323.

[26] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *CoRR*, vol. abs/1511.07289, 2015. [Online]. Available: http://arxiv.org/abs/1511.07289

[27] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML*, vol. 30, no. 1, 2013.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[29] S. Zagoruyko and N. Komodakis, "Wide residual networks," *CoRR*, vol. abs/1605.07146, 2016. [Online]. Available: http://arxiv.org/abs/1605.07146

[30] "Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios," International Telecommunication Union, Tech. Rep. BT.601-7, February 2011.

[31] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial networks," *CoRR*, vol. abs/1406.2661, 2014. [Online]. Available: http://arxiv.org/abs/1406.2661

[32] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard *et al.*, "Learning algorithms for classification: A comparison on handwritten digit recognition," *Neural networks: the statistical mechanics perspective*, vol. 261, p. 276, 1995.

[33] J. Susskind, A. Anderson, and G. Hinton, "The toronto face dataset," 2010.

[34] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[35] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015. [Online]. Available: http://arxiv.org/abs/1511.06434

[36] F. Yu, Y. Zhang, S. Song, A. Seff, and J. Xiao, "LSUN: construction of a large-scale image dataset using deep learning with humans in the loop," *CoRR*, vol. abs/1506.03365, 2015. [Online]. Available: http://arxiv.org/abs/1506.03365

[37] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *CoRR*, vol. abs/1606.03498, 2016. [Online]. Available: http://arxiv.org/abs/1606.03498

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575

[39] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567

[40] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015. [Online]. Available: http://arxiv.org/abs/1508.06576

[41] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[42] J. Johnson, A. Alahi, and F. Li, "Perceptual losses for real-time style transfer and super-resolution," *CoRR*, vol. abs/1603.08155, 2016. [Online]. Available: http://arxiv.org/abs/1603.08155

[43] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[44] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 1139–1147.

[45] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. [Online]. Available: http://arxiv.org/abs/1311.2901

[46] S. Geman, E. Bienenstock, and R. Doursat, "Neural networks and the bias/variance dilemma," *Neural computation*, vol. 4, no. 1, pp. 1–58, 1992.

[47] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[48] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[49] Y. Nesterov *et al.*, "Gradient methods for minimizing composite objective function," 2007.

[50] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*. Morgan Kaufmann, 1992, pp. 950–957.

[51] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[52] E. L. Denton, S. Chintala, A. Szlam, and R. Fergus, "Deep generative image models using a laplacian pyramid of adversarial networks," *CoRR*, vol. abs/1506.05751, 2015. [Online]. Available: http://arxiv.org/abs/1506.05751

[53] A. Makhzani, J. Shlens, N. Jaitly, and I. J. Goodfellow, "Adversarial autoencoders," *CoRR*, vol. abs/1511.05644, 2015. [Online]. Available: http://arxiv.org/abs/1511.05644

A Code Sample

The remaining pages contain a representative sample of the code used for the presented experiments on full image generation using hinge loss, moment matching, and style loss.

```
import os
import numpy as np
import scipy.misc as sp
import tensorflow as tf
import random
from time import gmtime, strftime

from glob import glob

import sys
sys.path.insert(0,'utils/')
sys.path.insert(0,'models/')
sys.path.insert(0,'dataset_loading/')

print sys.path

from input_chain import build_input_queue
from model import DCGAN

name = 'b64'
images, batch_size = build_input_queue(name)

gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.9)
sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options,allow_soft_p
lacement=True))

with tf.device('/gpu:1'):
   dcgan = DCGAN(sess, image_shape=images.get_shape().as_list()[1:],
            batch_size=batch_size, queue=images, dataset_name=name)
   dcgan.train_init()

coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)
try:
      step = 0
      while not coord.should_stop():
            dcgan.train_iter()

except tf.errors.OutOfRangeError:
      print('Done training for %d epochs, %d steps.' % (1000, step))
finally:
      # When done, ask the threads to stop.
      coord.request_stop()

        # Wait for threads to finish.
      coord.join(threads)
      sess.close()
```

```
import os
import time
from glob import glob
import tensorflow as tf
import datetime
import random

import sys
sys.path.insert(0,'util/')
sys.path.insert(0,'vgg/')

from generator import Generator
from discriminator import Discriminator

from ops import *
from utils import *
#from vgg import Vgg16

class DCGAN(object):
    def __init__(self, sess, queue, batch_size=128, image_shape=[28, 28, 1],
                 z_dim=100, dataset_name='mnist'):
        """

   Args:
     sess: TensorFlow session
     batch_size: The size of batch. Should be specified before training.
     z_dim: (optional) Dimension of dim for Z. [100]
   """
        self.sess = sess
        self.batch_size = batch_size
        self.image_shape = image_shape
        self.images = queue

        self.z_dim = z_dim

        self.dataset_name = dataset_name
        self.checkpoint_dir = 'saved_models/'

        self.generator = Generator(self.dataset_name, self.batch_size)
        self.discriminator = Discriminator(self.dataset_name, self.batch_siz
e)

        self.build_model()

    def build_model(self):
        # self.images = tf.placeholder(tf.float32, [self.batch_size] + self.
image_shape,
                                    # name='real_images')
        self.z = tf.random_uniform([self.batch_size, self.z_dim], minval=-1,
 maxval=1,
                                    name='z')

        self.G = self.generator(self.z, dev='/gpu:0')
        # tf.image_summary('samples', self.G, max_images=4)

        # return un-normalized probits
        self.p_example, self.example_layers = self.discriminator(self.images
, dev='/gpu:0')
```

```
        self.p_forgery, self.forgery_layers = self.discriminator(self.G, reu
se=True, dev='/gpu:1')


        #x = np.linspace(-0.9, 0.9, 20)
        #a, b = np.meshgrid(x,x)
        #a = a.flatten()
        #b = b.flatten()
        #self.z_samp = tf.constant(np.vstack([a, b]).transpose(), tf.float32
)

        #start = np.random.uniform(low=-1, high=1, size=self.z_dim)

        #positions = [start]

        #for i in xrange(1, 20**2):
        #    next = positions[-1] + np.random.uniform(low=-0.3, high=0.3, si
ze=self.z_dim)
        #    next = np.minimum(np.ones_like(next), np.maximum(-np.ones_like(
next), next))
        #    positions.append(next)

        #self.z_samp = tf.constant(np.vstack(positions), tf.float32)

        def interp(z_dim):
            r = random.random()
            if r < 0.25:
                start = np.random.uniform(low=-1, high=0, size=z_dim)
                end = np.random.uniform(low=-1, high=1, size=z_dim)
            elif (r > 0.25) and (r < 0.5):
                start = np.random.uniform(low=-1, high=1, size=z_dim)
                end = np.random.uniform(low=0, high=1, size=z_dim)
            elif (r > 0.5) and (r < 0.75):
                start = np.random.uniform(low=0, high=1, size=z_dim)
                end = np.random.uniform(low=-1, high=1, size=z_dim)
            else:
                start = np.random.uniform(low=-1, high=1, size=z_dim)
                end = np.random.uniform(low=0, high=1, size=z_dim)

            diff = end - start

            positions = [start]

            for i in xrange(1, 10):
                next = positions[-1] + diff/10
                positions.append(next)

            return np.vstack(positions)

        sample = []

        for i in xrange(0,10):
            sample.append(interp(self.z_dim))

        self.z_samp = tf.constant(np.vstack(sample), tf.float32)
        self.z_samp_rand = tf.random_uniform([10**2, self.z_dim], minval=-1,
 maxval=1,
                                   name='z')
        self.sampler = self.generator(self.z_samp, sample=True)
        self.sampler_rand = self.generator(self.z_samp_rand, sample=True)


#        self.sampler = tf.concat(0, [self.sampler, tf.slice(self.images,[0,
0,0,0],[1,-1,-1,-1])])
        s = self.p_example.get_shape().as_list()
        print s
        if s[1] == 1:
            self.nClasses = 1
        else:
            self.nClasses = s[1] - 1

        if self.nClasses == 1:
            with tf.variable_scope('loss_calc'):
                #alpha = 0.0 # softening parameter

                #self.example_loss = binary_cross_entropy_with_logits(tf.one
s_like(self.p_example) - alpha, tf.nn.sigmoid(self.p_example))
                #self.forgery_loss = binary_cross_entropy_with_logits(tf.zer
os_like(self.p_forgery), tf.nn.sigmoid(self.p_forgery))
                #self.d_loss = self.example_loss + self.forgery_loss

                self.example_loss = tf.reduce_mean(tf.maximum(tf.zeros_like(
self.p_example),1-self.p_example ))
                self.forgery_loss = tf.reduce_mean(tf.maximum(tf.zeros_like(
self.p_forgery),1+self.p_forgery ))


                self.d_loss = self.example_loss + self.forgery_loss

                self.ad_loss = binary_cross_entropy_with_logits(tf.ones_like
(self.p_forgery), tf.nn.sigmoid(self.p_forgery))

                self.mmd_loss = tf.reduce_mean(map(lambda x,y : mmd(flatten(
x),flatten(y)), self.example_layers, self.forgery_layers))
                self.style_loss = tf.reduce_mean(map(lambda x,y : tf.reduce_
mean(tf.pow(style(x)- style(y),2)), self.example_layers, self.forgery_layers
))
                self.g_loss = self.ad_loss + 0.2*self.mmd_loss + 0.1*self.st
yle_loss
        else:
            exit(-1)

        tf.scalar_summary('loss/example', self.example_loss)
        tf.scalar_summary('loss/forgery', self.forgery_loss)
        tf.scalar_summary('loss/ad_generator', self.ad_loss)
        tf.scalar_summary('loss/mmd_generator', self.mmd_loss)
        tf.scalar_summary('loss/style_generator', self.style_loss)
        tf.scalar_summary('loss/generator', self.g_loss)
        #tf.scalar_summary('g_loss/entropy', self.forgery_entropy_loss)
        #tf.scalar_summary('g_loss/hinge', self.hinge_loss)
        #tf.scalar_summary('g_loss/uniformity', self.dist_loss)

        t_vars = tf.trainable_variables()
        param_count = 0
        for var in t_vars:
            param_count += reduce(lambda x, y: y*x, var.get_shape().as_list(
))

        print "There are", format(param_count, ",d"), "parameters"

        self.d_vars = [var for var in t_vars if 'd_' in var.name]
        self.g_vars = [var for var in t_vars if 'g_' in var.name]

        self.saver = tf.train.Saver()

    def train_init(self):
        """Init Train DCGAN"""
        self.d_optim = tf.train.AdamOptimizer(0.00005, beta1=0.5) \
                          .minimize(self.d_loss, var_list=self.d_vars)
        self.g_optim1 = tf.train.AdamOptimizer(0.0005, beta1=0.5) \
                          .minimize(self.g_loss, var_list=self.g_vars)
        with tf.control_dependencies([self.g_optim1]):
            self.g_optim2 = tf.train.AdamOptimizer(0.0005, beta1=0.5) \
                          .minimize(self.g_loss, var_list=self.g_vars)

        self.counter = 1
        self.sess.run(tf.initialize_all_variables())

        logdir = 'logs/' + str(int(time.time()))
        self.writer = tf.python.training.summary_io.SummaryWriter(logdir)
        self.writer.add_graph(self.sess.graph)
        self.writer.flush()

        self.summ = tf.merge_all_summaries()

    def train_iter(self):
        """Train DCGAN
      Args:
        batch: numpy array of shape [batch_size, image_shape]
    """
        start_time = time.time()

        # Update D network
        _, _, _, summ, err_d, err_e, err_f, err_h = self.sess.run([self.d_op
tim, self.g_optim1, self.g_optim2, self.summ, self.d_loss, self.example_loss
, self.forgery_loss, self.g_loss])
        #_, summ, err_d, err_e, err_f, err_h = self.sess.run([self.g_optim1,
 self.summ, self.d_loss, self.example_loss, self.forgery_loss, self.g_loss])

        # Update G network
        # _, summ, err_d, err_e, err_f = self.sess.run([self.g_optim, self.s
umm, self.d_loss, self.example_loss, self.forgery_loss])
        self.writer.add_summary(summ, self.counter)
        self.writer.flush()
        self.counter += 1
        print("%d, time: %4.3f, d: %.5f, e: %.4f, f: %.4f, h: %.4f" \
            % (self.counter, time.time() - start_time, err_d, err_e, err_f,
err_h ))

        if np.mod(self.counter, 100) == 0:
            samples, samples_rand = self.sess.run(
                [self.sampler, self.sampler_rand],)
            w = int(np.ceil(np.sqrt(samples.shape[0])))
            self.save_images(samples, [w, w],
                        './samples_' + self.dataset_name + '/train_interp_%s.png' %
(self.counter))
            self.save_images(samples_rand, [w, w],
                        './samples_' + self.dataset_name + '/train_rand_%s.png' % (
self.counter))

        #     self.save_images(samples, [2, 2],
        #                 './samples_' + self.dataset_name + '/large_%s.png'
 % (self.counter))

        if np.mod(self.counter, 10000) == 0:
            self.save(self.checkpoint_dir, self.counter)

    def save(self, checkpoint_dir, step):
        model_name = "DCGAN.model"
        model_dir = "%s_%s" % (self.dataset_name, self.batch_size)
        checkpoint_dir = os.path.join(checkpoint_dir, model_dir)

        if not os.path.exists(checkpoint_dir):
            os.makedirs(checkpoint_dir)

        self.saver.save(self.sess,
                        os.path.join(checkpoint_dir, model_name),
                        global_step=step)

    def load(self, checkpoint_dir):
        print(" [*] Reading checkpoints...")

        model_dir = "%s_%s" % (self.dataset_name, self.batch_size)
        checkpoint_dir = os.path.join(checkpoint_dir, model_dir)

        ckpt = tf.train.get_checkpoint_state(checkpoint_dir)
        if ckpt and ckpt.model_checkpoint_path:
            ckpt_name = os.path.basename(ckpt.model_checkpoint_path)
            self.saver.restore(self.sess, os.path.join(checkpoint_dir, ckpt_
name))
        else:
            raise Exception(" [!] Testing, but %s not found" % checkpoint_dir)


    def save_images(self, images, size, path):
        h, w = images.shape[1], images.shape[2]
        img = np.zeros((1, h * size[0], w * size[1], 3))

        for idx, image in enumerate(images):
            i = idx % size[1]
            j = idx / size[1]
            img[0, j*h:j*h+h, i*w:i*w+w, :] = image
        return scipy.misc.imsave(path, img[0])
```

```
import os
import time
from glob import glob
import tensorflow as tf

import sys
sys.path.insert(0,'util/')

from ops import *
from utils import *

class Generator(object):
    """docstring for Generator"""
    def __init__(self, dataset_name, batch_size):
        super(Generator, self).__init__()
        self.dataset_name = dataset_name
        self.batch_size = batch_size
        generator_funcs = {
            'mnist' : 'mnist_generator_ln',
            'buildings' : 'buildings_generator',
            'faces' : 'faces_generator',
            'b64' : 'b64_g_p'
        }

        self.callee = getattr(self, generator_funcs[self.dataset_name])

    def __call__(self, z, sample=False, dev='/gpu:0'):
        return self.callee(z, sample=sample, dev=dev)

    def mnist_generator(self, z, sample=False, dev='/gpu:0'):
        with tf.variable_scope('generator'):
            if sample:
                tf.get_variable_scope().reuse_variables()
                reuse = True
            else:
                reuse = False

            # batch_size = tf.shape(z)[0]
            batch_size = self.batch_size
            print z.get_shape()

            gf_dim = 64

            # project `z` and reshape
            h0 = tf.reshape(linear(z, gf_dim*8*4*4, 'g_h0_lin'),
                                [-1, 4, 4, gf_dim * 8])
            h0 = tf.nn.relu(batch_norm(h0, train=True, reuse=reuse, name='g_
bn0'))
            #h0 = add_feats(h0, gf_dim*4)
            print h0.get_shape()

            h1 = deconv2d(h0, [batch_size, 7, 7, gf_dim*3], name='g_h1')
            h1 = tf.nn.relu(batch_norm(h1, train=True, reuse=reuse, name='g_
bn1'))
            #h1 = add_feats(h1, gf_dim)

            h2 = deconv2d(h1, [batch_size, 14, 14, 2*gf_dim], name='g_h2')
            h2 = tf.nn.relu(batch_norm(h2, train=True, reuse=reuse, name='g_
b2'))
            #h2 = add_feats(h2, gf_dim/2)
            print h2.get_shape()

            h3 = deconv2d(h2, [batch_size, 28, 28, 1], name='g_h3')
            print h3.get_shape()

            return tf.nn.sigmoid(h3)

    def mnist_generator_ln(self, z, sample=False, dev='/gpu:0'):
        with tf.variable_scope('generator'):
            if sample:
                tf.get_variable_scope().reuse_variables()
                reuse = True
            else:
                reuse = False

            batch_size = z.get_shape().as_list()[0]
            #batch_size = 1024 #self.batch_size
            print z.get_shape()

            gf_dim = 32

            # project 'z' and reshape
            h0 = tf.reshape(linear(z, gf_dim*8*4*4, 'g_h0_lin'),
                            [batch_size, 4, 4, gf_dim * 8])
            h0 = tf.nn.relu(layer_norm(h0, 'g_ln0'))
            #h0 = add_feats(h0, gf_dim*4)
            print h0.get_shape()

            h1 = deconv2d(h0, [batch_size, 7, 7, gf_dim*4], name='g_h1')
            h1 = tf.nn.relu(layer_norm(h1, 'g_ln1'))
            #h1 = add_feats(h1, gf_dim)

            h2 = deconv2d(h1, [batch_size, 14, 14, gf_dim*2], name='g_h2')
            h2 = tf.nn.relu(layer_norm(h2, 'g_ln2'))
            #h2 = add_feats(h2, gf_dim/2)
            print h2.get_shape()

            h3 = deconv2d(h2, [batch_size, 28, 28, 1], name='g_h3')
            print h3.get_shape()

            return tf.nn.sigmoid(h3)

    def b64_g_p(self, z, sample=False, dev='/gpu:0'):
        with tf.variable_scope('generator'):
            if sample:
                tf.get_variable_scope().reuse_variables()

            batch_size = z.get_shape().as_list()[0]
            print z.get_shape()

            gf_dim = 64
            with tf.device(dev):
                # project 'z' and reshape
                h0 = tf.reshape(linear(z, gf_dim*8*4*4, 'g_h0_lin'),
                                [-1, 4, 4, gf_dim * 8])
                h0 = tf.nn.relu(layer_norm(h0, 'g_ln0'))
                h1 = deconv2d(h0, [batch_size, 8, 8, gf_dim*4], name='g_h1')
                h1 = tf.nn.relu(layer_norm(h1, 'g_ln1'))

                h2 = deconv2d(h1, [batch_size, 16, 16, 2*gf_dim], name='g_h2
')
                h2 = tf.nn.relu(layer_norm(h2, 'g_ln2'))

                h3 = deconv2d(h2, [batch_size, 32, 32, gf_dim], name='g_h3')
                h3 = tf.nn.relu(layer_norm(h3, 'g_ln3'))

                h4 = deconv2d(h3, [batch_size, 64, 64, 3], name='g_h4')

                return tf.nn.sigmoid(h4)

    def mnist_generator_fc(self, z, sample=False, dev='/gpu:0'):
        with tf.variable_scope('generator'):
            if sample:
                tf.get_variable_scope().reuse_variables()
                reuse = True
            else:
                reuse = False

            batch_size = z.get_shape().as_list()[0]
            print z.get_shape()

            # project `z` and reshape
            h0 = linear(z, 1000, 'g_h0_lin')
            h0 = tf.nn.elu(h0)
            print h0.get_shape()

            h1 = linear(h0, 1000, 'g_h1_lin')
            h1 = tf.nn.elu(h1, 'g_ln1')
            print h1.get_shape()

            #h2 = linear(h1, 512, 'g_h2_lin')
            #h2 = tf.nn.elu(layer_norm(h2, 'g_ln2'))
            #print h2.get_shape()

            h3 = linear(h1, 784, 'g_h3_lin')
            h3 = tf.nn.sigmoid(h3)
            print h3.get_shape()

            h3 = tf.reshape(h3, [batch_size, 28, 28, 1])
            print h3.get_shape()

            return h3

    def mnist_generator_vbn(self, z, sample=False, dev='/gpu:0'):
        with tf.variable_scope('generator'):
            if sample:
                tf.get_variable_scope().reuse_variables()
                reuse = True
            else:
                reuse = False

            # batch_size = tf.shape(z)[0]
            batch_size = self.batch_size
            print z.get_shape()

            gf_dim = 16

            z_ref = tf.constant(np.random.random(z.get_shape().as_list())*2
- 1, dtype=tf.float32)
            z = tf.concat(0, [z, z_ref])

            def vbn(x_, name):
                with tf.variable_scope(name):
                    bias = tf.get_variable('gain', x_.get_shape().as_list()[-
1],
                               initializer=tf.constant_initializer(value=0.0
))
                    gain = tf.get_variable('bias', x_.get_shape().as_list()[-
1],
                            initializer=tf.constant_initializer(value=1.0))

                    x_ref = tf.slice(x_, [batch_size, 0, 0, 0], [batch_size,
 -1, -1, -1])
                    m, s = tf.nn.moments(x_ref, axes=[0], keep_dims=True)
                    return gain*(x_ - m)/tf.sqrt(s) + bias

            # project `z` and reshape
            h0 = tf.reshape(linear(z, gf_dim*2*4*4, 'g_h0_lin'),
                            [-1, 4, 4, gf_dim * 2])
            h0 = tf.nn.relu(vbn(h0, 'g_vbn0'))
            print h0.get_shape()

            h1 = deconv2d(h0, [2*batch_size, 7, 7, gf_dim*3], name='g_h1')
            h1 = tf.nn.relu(vbn(h1, 'g_vbn1'))

            h2 = deconv2d(h1, [2*batch_size, 14, 14, 2*gf_dim], name='g_h2')
            h2 = tf.nn.relu(vbn(h2, 'g_vbn2'))
            print h2.get_shape()

            h2 = tf.slice(h2, [0, 0, 0, 0], [batch_size, -1, -1, -1])
            h3 = deconv2d(h2, [batch_size, 28, 28, 1], name='g_h3')
            print h3.get_shape()

            return tf.nn.sigmoid(h3)

    def b64_g(self, z, sample=False, large_sample=False, dev='/gpu:0'):
        with tf.variable_scope('generator'):
            if sample or large_sample:
                tf.get_variable_scope().reuse_variables()

            # batch_size = tf.shape(z)[0]
            batch_size = self.batch_size
            if large_sample:
                batch_size=4
            print z.get_shape()

            gf_dim = 64
            with tf.device(dev):
                # project `z` and reshape
                h0 = tf.reshape(linear(z, gf_dim*8*4*4, 'g_h0_lin'),
                                [-1, 4, 4, gf_dim * 8])
                h0 = tf.nn.relu(layer_norm(h0))
                # h0 = add_feats(h0, gf_dim*4)

                h1 = deconv2d(h0, [batch_size, 8, 8, gf_dim*3], name='g_h1')
                h1 = tf.nn.relu(layer_norm(h1))
                # h1 = add_feats(h1, gf_dim*2)

                if large_sample:
                    h2 = deconv2d(h1, [batch_size, 32, 32, 2*gf_dim], d_h=4,
 d_w=4, name='g_h2')
                    h2 = tf.nn.relu(layer_norm(h2))

                    h3 = deconv2d(h2, [batch_size, 128, 128, gf_dim], d_h=4,
 d_w=4, name='g_h3')
                    h3 = tf.nn.relu(layer_norm(h3))

                    h4 = deconv2d(h3, [batch_size, 512, 512, 3], d_h=4, d_w=
4, name='g_h4')

                    return tf.nn.sigmoid(h4)

                h2 = deconv2d(h1, [batch_size, 16, 16, 2*gf_dim], name='g_h2
')
                h2 = tf.nn.relu(layer_norm(h2))
                # h2 = add_feats(h2, gf_dim)

                h3 = deconv2d(h2, [batch_size, 32, 32, gf_dim], name='g_h3')
                h3 = tf.nn.relu(layer_norm(h3))

                h4 = deconv2d(h3, [batch_size, 64, 64, 3], name='g_h4')

                return tf.nn.sigmoid(h4)

    def mnist_generator_wide(self, z, sample=False):
        if sample:
            tf.get_variable_scope().reuse_variables()
            reuse = True
        else:
            reuse = False

        # batch_size = tf.shape(z)[0]
        batch_size = self.batch_size
        print z.get_shape()

        gf_dim = 16

        # project `z` and reshape
        h0 = tf.reshape(linear(z, gf_dim*8*4*4, 'g_h0_lin'),
                        [-1, 4, 4, gf_dim * 8])
        h0 = add_feats(h0, gf_dim*4)

        h1 = wide_basic_deconv(h0, (batch_size, 7, 7, 4*gf_dim), 2, 'g_h1',
reuse=reuse)
        h1 = add_feats(h1, gf_dim)

        h2 = wide_basic_deconv(h1, (batch_size, 14, 14, 2*gf_dim), 2, 'g_h2'
, reuse=reuse)
        h2 = add_feats(h2, gf_dim/2)
        h3 = deconv2d(h2, [batch_size, 28, 28, 1], name='g_h3')

        return tf.nn.sigmoid(h3)

    def mnist_generator_atrous(self, z, sample=False, dev='/gpu:0'):
        if sample:
            tf.get_variable_scope().reuse_variables()
            reuse = True
        else:
            reuse = False

        batch_size = self.batch_size
        print z.get_shape()

        gf_dim = 4

        # project `z` and reshape
        # h0 = tf.reshape(linear(z, gf_dim*8*4*4, 'g_h0_lin'),
        #                  [-1, 4, 4, gf_dim * 8])
        # h0 = tf.nn.relu(batch_norm(h0, train=True, reuse=reuse, name='g_bn
0'))
        # # h0 = add_feats(h0, gf_dim*4)
        # print h0.get_shape()

        # h1 = deconv2d(h0, [batch_size, 7, 7, gf_dim*3], name='g_h1')
        # h1 = tf.nn.relu(batch_norm(h1, train=True, reuse=reuse, name='g_bn
1'))
        # # h1 = add_feats(h1, gf_dim)

        # h2 = deconv2d(h1, [batch_size, 14, 14, 2*gf_dim], name='g_h2')
        # h2 = tf.nn.relu(batch_norm(h2, train=True, reuse=reuse, name='g_b2
'))
        # # h2 = add_feats(h2, gf_dim/2)
        # print h2.get_shape()

        # h2 = deconv2d(h2, [batch_size, 28, 28, 16], name='g_h2_1')
        # h2 = tf.nn.relu(batch_norm(h2, train=True, reuse=reuse, name='g_b2
_1'))

        h2 = tf.reshape(linear(z, 8*28*28, 'g_h0_lin'),
                        [-1, 28, 28, 8])

        h3 = atrous_conv2d(h2, 16, rate=1, name='g_h3')
        h3 = tf.nn.relu(batch_norm(h3, train=True, reuse=reuse, name='g_b3')
)
        h3 = add_feats(h3, 16)
        print h3.get_shape()

        h4 = atrous_conv2d(h3, 32, rate=2, name='g_h4')
        h4 = tf.nn.relu(batch_norm(h4, train=True, reuse=reuse, name='g_b4')
)
        h4 = add_feats(h4, 8)
        print h4.get_shape()

        h5 = atrous_conv2d(h4, 32, rate=4, name='g_h5')
        h5 = tf.nn.relu(batch_norm(h5, train=True, reuse=reuse, name='g_b5')
)
```

```
        h5 = add_feats(h5, 4)
        print h5.get_shape()

        h6 = atrous_conv2d(h5, 1, rate=8, name='g_h6')
        # h6 = tf.nn.relu(batch_norm(h6, train=True, reuse=reuse, name='g_b6
'))
        # print h6.get_shape()

        return tf.nn.sigmoid(h6)

    def buildings_generator(self, z, sample=False):
        with tf.device("/gpu:0"):
            if sample:
                tf.get_variable_scope().reuse_variables()
                reuse = True
            else:
                reuse = False

            batch_size = self.batch_size
            print z.get_shape()

            low_dim = 32
            gf_dim = 32

            # project `z' and reshape
            h0 = tf.reshape(linear(z, low_dim*8*4*4, 'g_h0_lin'),
                            [-1, 4, 4, low_dim*8])
            h0 = tf.nn.relu(batch_norm(h0, train=True, reuse=reuse, name='g_
bn0'))
            h0 = add_feats(h0, gf_dim*4)

            h1 = deconv2d(h0, [batch_size, 8, 8, 4*low_dim], name='g_h1')
            h1 = tf.nn.relu(batch_norm(h1, train=True, reuse=reuse, name='g_
bn1'))
            h1 = add_feats(h1, gf_dim)

            h2 = deconv2d(h1, [batch_size, 16, 16, 2*low_dim], name='g_h2')
            h2 = tf.nn.relu(batch_norm(h2, train=True, reuse=reuse, name='g_
b2'))
            h2 = add_feats(h2, gf_dim/2)

            h3 = deconv2d(h2, [batch_size, 32, 32, low_dim], name='g_h3')
            h3 = tf.nn.relu(batch_norm(h3, train=True, reuse=reuse, name='g_
b3'))
            h3 = add_feats(h3, gf_dim/4)

            h4 = deconv2d(h3, [batch_size, 64, 64, low_dim], name='g_h4')
            h4 = tf.nn.relu(batch_norm(h4, train=True, reuse=reuse, name='g_
b4'))
            h4 = add_feats(h4, gf_dim/8)

            h5 = deconv2d(h4, [batch_size, 128, 128, low_dim], name='g_h5')
            h5 = tf.nn.relu(batch_norm(h5, train=True, reuse=reuse, name='g_
b5'))
            h5 = add_feats(h5, gf_dim/16)

            h6 = deconv2d(h5, [batch_size, 256, 256, 3], name='g_h6')
            # h6 = tf.nn.relu(batch_norm(h6, train=True, reuse=reuse, name='
g_b6'))
            # # h6 = tf.random_uniform([batch_size, 256, 256, 1], minval=-1,
 maxval=1)
            # h6 = tf.reshape(linear(z, 2*256*256, 'g_h0_lin'),
            #                  [-1, 256, 256, 2])
            # h7 = atrous_conv2d(h6, 2*gf_dim, rate=1, name='g_h7')
            # h7 = tf.nn.relu(batch_norm(h7, train=True, reuse=reuse, name='
g_b7'))
            # h7 = add_feats(h7, 2*gf_dim)

            # h8 = atrous_conv2d(h7, gf_dim, rate=2, name='g_h8')
            # h8 = tf.nn.relu(batch_norm(h8, train=True, reuse=reuse, name='
g_b8'))
            # h8 = add_feats(h8, gf_dim)

            # h9 = atrous_conv2d(h8, gf_dim, rate=4, name='g_h9')
            # h9 = tf.nn.relu(batch_norm(h9, train=True, reuse=reuse, name='
g_b9'))
            # h9 = add_feats(h9, gf_dim/2)

            # h10 = atrous_conv2d(h9, 3, rate=8, name='g_h10')

            return tf.nn.sigmoid(h6)

    def faces_generator(self, z, sample=False):
        with tf.variable_scope('generator'):
            with tf.device("/gpu:0"):
                if sample:
                    tf.get_variable_scope().reuse_variables()
                    reuse = True
                else:
                    reuse = False

                batch_size = self.batch_size
                print z.get_shape()

                low_dim = 32
                gf_dim = 32

                # project `z` and reshape
                h0 = tf.reshape(linear(z, low_dim*8*6*7, 'g_h0_lin'),
                                [-1, 6, 7, low_dim*8])
                h0 = tf.nn.relu(batch_norm(h0, train=True, reuse=reuse, name
='g_bn0'))
                h0 = add_feats(h0, gf_dim*4)

                h1 = deconv2d(h0, [batch_size, 6, 7, 4*low_dim], d_h=1, d_w=
1, name='g_h1')
                h1 = tf.nn.relu(batch_norm(h1, train=True, reuse=reuse, name
='g_bn1'))
                h1 = add_feats(h1, gf_dim)

                h2 = deconv2d(h1, [batch_size, 12, 14, 2*low_dim], name='g_h
2')
                h2 = tf.nn.relu(batch_norm(h2, train=True, reuse=reuse, name
='g_b2'))
                h2 = add_feats(h2, gf_dim/2)
                h3 = deconv2d(h2, [batch_size, 23, 28, low_dim], name='g_h3'
)
                h3 = tf.nn.relu(batch_norm(h3, train=True, reuse=reuse, name
='g_b3'))
                h3 = add_feats(h3, gf_dim/4)

                h4 = deconv2d(h3, [batch_size, 45, 55, low_dim], name='g_h4'
)
                h4 = tf.nn.relu(batch_norm(h4, train=True, reuse=reuse, name
='g_b4'))
                h4 = add_feats(h4, gf_dim/8)

                h5 = deconv2d(h4, [batch_size, 89, 109, low_dim], name='g_h5
')
                h5 = tf.nn.relu(batch_norm(h5, train=True, reuse=reuse, name
='g_b5'))
                h5 = add_feats(h5, gf_dim/16)

                h6 = deconv2d(h5, [batch_size, 178, 218, 3], name='g_h6')

                return tf.nn.sigmoid(h6)

def add_feats(x, nFeats):
    s = x.get_shape().as_list()
    # import ipdb; ipdb.set_trace()
    s[3] = nFeats
    h = tf.random_normal(s, stddev=0.3)
    return tf.concat(3, [x,h])
```

```
import os
import time
from glob import glob
import tensorflow as tf

import sys
sys.path.insert(0,'util/')

from ops import *
from utils import *

class Discriminator(object):
    """docstring for Discriminator"""
    def __init__(self, dataset_name, batch_size):
        super(Discriminator, self).__init__()
        self.dataset_name = dataset_name
        self.batch_size = batch_size
        discriminator_funcs = {
            'mnist'  :  'mnist_discriminator_ln_noise',
            'buildings'  :  'buildings_discriminator_atrous',
            'faces'  :  'face_discriminator_atrous',
            'b64'  :  'mnist_discriminator_ln_noise'
        }

        self.callee = getattr(self, discriminator_funcs[self.dataset_name])

    def __call__(self, image, reuse=False, dev='/gpu:0'):
        return self.callee(image, reuse, dev)

    def mnist_discriminator(self, image, reuse=False, dev='/gpu:0'):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        batch_size = self.batch_size
        df_dim = 32

        h0 = lrelu(
            batch_norm(
                conv2d(image, df_dim, name='d_h0_conv'),
                train=True, reuse=reuse, name='d_bn0')
            )
        h1 = lrelu(
            batch_norm(
                conv2d(h0, df_dim*2, name='d_h1_conv'),
                train=True, reuse=reuse, name='d_bn1')
            )
        h2 = lrelu(
            batch_norm(
                conv2d(h1, df_dim*4, name='d_h2_conv'),
                train=True, reuse=reuse, name='d_bn2')
            )
        h3 = lrelu(
            batch_norm(
                conv2d(h2, df_dim*8, name='d_h3_conv'),
                train=True, reuse=reuse, name='d_bn3')
            )
        h3 = lrelu(linear(flatten(h3), 256, 'd_h3_lin'))
        h3 = minibatch_features(flatten(h3))
        h4 = linear(h3,10, 'd_h4_lin')

        return h4

    def mnist_discriminator_ln(self, image, reuse=False, dev='/gpu:0'):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        #batch_size = 1024 #self.batch_size
        df_dim = 8

        h0 = lrelu(
            layer_norm(
                conv2d(image, df_dim, name='d_h0_conv'), 'd_ln0')
            )
        h1 = lrelu(
            layer_norm(
                conv2d(h0, df_dim*2, name='d_h1_conv'), 'd_ln1')
            )
        h2 = lrelu(
            layer_norm(
                conv2d(h1, df_dim*4, name='d_h2_conv'), 'd_ln2')
            )
        h3 = lrelu(
            layer_norm(
                conv2d(h2, df_dim*8, name='d_h3_conv'), 'd_ln3')
            )

        h4 = lrelu(linear(flatten(h3), 256, 'd_h3_lin'))
        h4 = minibatch_features(flatten(h4))

        h4 = linear(h4, 1, 'd_h4_lin')

        return h4, [h0, h1, h2, h3]

    def mnist_discriminator_ln_noise(self, image, reuse=False, dev='/gpu:0'):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        def noise_layer(x_, stddev=0.5):
            s = x_.get_shape().as_list()
            return x_ + tf.random_normal(s, stddev=stddev)

        #batch_size = 1024 #self.batch_size
        #df_dim = 8 #mnist
        df_dim = 64 # b64

        h0 = lrelu(
            layer_norm(
                noise_layer(conv2d(image, df_dim, name='d_h0_conv')), 'd_ln0')
            )
        h1 = lrelu(
            layer_norm(
                noise_layer(conv2d(h0, df_dim*2, name='d_h1_conv')), 'd_ln1')
            )
        h2 = lrelu(
            layer_norm(
                noise_layer(conv2d(h1, df_dim*4, name='d_h2_conv')), 'd_ln2')
            )
        h3 = lrelu(
            layer_norm(
                noise_layer(conv2d(h2, df_dim*8, name='d_h3_conv')), 'd_ln3')
            )

        h4 = lrelu(linear(flatten(h3), 1000, 'd_h3_lin')) # 500 for mnist
        h4 = minibatch_features(flatten(h4), 'd_mb1')

        h4 = lrelu(linear(flatten(h4), 500, 'd_h31_lin'))
        h4 = minibatch_features(flatten(h4), 'd_mb2')

        h4 = linear(h4, 1, 'd_h4_lin')

        return h4, [h0, h1, h2, h3]

    def mnist_discriminator_fc(self, image, reuse=False, dev='/gpu:0'):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        def noise_layer(x_, stddev=0.5):
            s = x_.get_shape().as_list()
            return x_ + tf.random_normal(s, stddev=stddev)

        image = flatten(image)

        h0 = linear(noise_layer(image, stddev=0.3), 1000, 'd_lin0')
        h0 = tf.nn.elu(layer_norm(h0, 'd_ln0'))

        h1 = linear(noise_layer(h0), 500, 'd_lin1')
        h1 = tf.nn.elu(layer_norm(h1, 'd_ln1'))

        h1_ = linear(noise_layer(h1), 500, 'd_lin11')
        h1_ = tf.nn.elu(layer_norm(h1_, 'd_ln11'))

        h2 = linear(noise_layer(h1_), 250, 'd_lin2')
        h2 = tf.nn.elu(layer_norm(h2, 'd_ln2'))

        h2_ = minibatch_features(h2, 'd_mb1')
        h3 = linear(noise_layer(h2_), 250, 'd_lin3')
        h3 = tf.nn.elu(layer_norm(h3, 'd_ln3'))

        h3_ = minibatch_features(h3, 'd_mb2')
        h4 = linear(noise_layer(h3_), 1, 'd_lin4')

        return h4, [h0, h1, h2, h3]

    def mnist_discriminator_style(self, image, reuse=False, dev='/gpu:0'):
        def style(x):
            shape = x.get_shape().as_list()
            c = shape[3]
            h = shape[1]
            w = shape[2]
            batch_size = shape[0]

            phi = tf.reshape(x, [batch_size, -1, c])
            print phi.get_shape().as_list()
            print tf.transpose(phi, [0, 2, 1]).get_shape().as_list()
            g = tf.batch_matmul(tf.transpose(phi, [0, 2, 1]), phi) / (c*h*w)

            assert g.get_shape().as_list()[1] == c

            return g

        if reuse:
            tf.get_variable_scope().reuse_variables()
            self.forgery_styles = []
            styles = self.forgery_styles
        else:
            self.true_styles = []
            styles = self.true_styles
            print '*********WOMP'

        batch_size = self.batch_size
        df_dim = 32

        h0 = lrelu(
            batch_norm(
                conv2d(image, df_dim, name='d_h0_conv'),
                train=True, reuse=reuse, name='d_bn0')
            )
        styles.append(style(h0))

        h1 = lrelu(
            batch_norm(
                conv2d(h0, df_dim*2, name='d_h1_conv'),
                train=True, reuse=reuse, name='d_bn1')
            )
        styles.append(style(h1))

        h2 = lrelu(
            batch_norm(
                conv2d(h1, df_dim*4, name='d_h2_conv'),
                train=True, reuse=reuse, name='d_bn2')
            )
        styles.append(style(h2))

        h3 = lrelu(
            batch_norm(
                conv2d(h2, df_dim*8, name='d_h3_conv'),
                train=True, reuse=reuse, name='d_bn3')
            )
        styles.append(style(h3))

        h3 = lrelu(linear(flatten(h3), 256, 'd_h3_lin'))
        h3 = minibatch_features(flatten(h3))

        h4 = tf.nn.sigmoid(linear(h3,1, 'd_h4_lin'))

        return h4

    def mnist_discriminator_atrous(self, image, reuse=False, dev='/gpu:0'):
        with tf.variable_scope('discriminator'):
            if reuse:
                tf.get_variable_scope().reuse_variables()
            batch_size = self.batch_size
            df_dim = 8

            h0 = lrelu(
                batch_norm(
                    atrous_conv2d(image, df_dim, rate=1, name='d_h0_conv'),
                    train=True, reuse=reuse, name='d_bn0')
                )
            h1 = lrelu(
                batch_norm(
                    atrous_conv2d(h0, df_dim*2, rate=2, name='d_h1_conv'),
                    train=True, reuse=reuse, name='d_bn1')
                )
            h2 = lrelu(
                batch_norm(
                    atrous_conv2d(h1, df_dim*4, rate=4, name='d_h2_conv'),
                    train=True, reuse=reuse, name='d_bn2')
                )
            h3 = lrelu(
                batch_norm(
                    atrous_conv2d(h2, df_dim*8, rate=8, name='d_h3_conv'),
                    train=True, reuse=reuse, name='d_bn3')
                )

            h4 = tf.nn.sigmoid(batch_norm(
                    atrous_conv2d(h3, 1, rate=1, name='d_h4_conv'),
                    train=True, reuse=reuse, name='d_bn4')
                )

            if reuse:
                tf.histogram_summary('logits/example', h4)
            else:
                tf.histogram_summary('logits/forgery', h4)

            # h4 = tf.reduce_mean(h4, 0)

            h4 = tf.nn.sigmoid(linear(flatten(h3),1, 'd_h4_lin'))

            return h4

    def b64_d_atrous2(self, image, reuse=False, dev='/gpu:0'):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        batch_size = self.batch_size
        df_dim = 32

        with tf.device(dev):
            h0 = lrelu(
                batch_norm(
                    conv2d(image, df_dim, name='d_h0_conv'),
                    train=True, reuse=reuse, name='d_bn0')
                )
            # 32x32
            h1 = lrelu(
                batch_norm(
                    conv2d(h0, df_dim*2, name='d_h1_conv'),
                    train=True, reuse=reuse, name='d_bn1')
                )
            # 16x16
            h2 = lrelu(
                batch_norm(
                    atrous_conv2d(h1, df_dim*4, rate=1, name='d_h2_conv'),
                    train=True, reuse=reuse, name='d_bn2')
                )
             # 16x16
            h3 = lrelu(
                    batch_norm(
                        atrous_conv2d(h2, df_dim*4, rate=2, name='d_h3_conv')
,
                        train=True, reuse=reuse, name='d_bn3')
                    )
            # # 16x16
            h4 = lrelu(
                batch_norm(
                    atrous_conv2d(h3, df_dim*8, rate=4, name='d_h4_conv'),
                    train=True, reuse=reuse, name='d_bn4')
                )
            # # 16x16 * 256

            h4_lin = lrelu(linear(flatten(h4), 512, 'd_h4_lin'))
            h4_mb = minibatch_features(flatten(h4_lin))

            h5 = tf.nn.sigmoid(linear(h4_mb,1, 'd_h5_lin'))

            if reuse:
                tf.histogram_summary('logits/example', h5)
                tf.histogram_summary('d_h0_conv/act/example', h0)
                tf.histogram_summary('d_h1_conv/act/example', h1)
                tf.histogram_summary('d_h2_conv/act/example', h2)
                tf.histogram_summary('d_h3_conv/act/example', h3)
                tf.histogram_summary('d_h4_conv/act/example', h4)
            else:
                tf.histogram_summary('logits/forgery', h5)
                tf.histogram_summary('d_h0_conv/act/forgery', h0)
                tf.histogram_summary('d_h1_conv/act/forgery', h1)
                tf.histogram_summary('d_h2_conv/act/forgery', h2)
                tf.histogram_summary('d_h3_conv/act/forgery', h3)
                tf.histogram_summary('d_h4_conv/act/forgery', h4)

            return h5

    def b64_d_wide_atrous2(self, image, reuse=False, dev='/gpu:0'):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        batch_size = self.batch_size
        df_dim = 16

        with tf.device(dev):
            h0 = conv2d(image, df_dim/2, d_h=1, d_w=1, name='d_h0_conv')
            # 64x64
            h1 = wide_basic_conv(h0, df_dim, 2, name='d_h1_conv', reuse=reuse
)
            # 32x32
            h2 = wide_basic_conv(h1, df_dim, 2, name='d_h2_conv', reuse=reuse
)
            h2 = lrelu(batch_norm(h2, train=True, reuse=reuse, name='d_bn2')
)
            #16x16
            h3 = lrelu(
                    batch_norm(
                        atrous_conv2d(h2, df_dim*2, rate=2, name='d_h3_conv')
,
                        train=True, reuse=reuse, name='d_bn3')
                    )
            # # 16x16
            h4 = lrelu(
                batch_norm(
                    atrous_conv2d(h3, df_dim*8, rate=4, name='d_h4_conv'),
                    train=True, reuse=reuse, name='d_bn4')
                )
            # # 16x16 * 256

            h4_lin = lrelu(linear(flatten(h4), 512, 'd_h4_lin'))
            h4_mb = minibatch_features(flatten(h4_lin))

            h5 = tf.nn.sigmoid(linear(h4_mb,1, 'd_h5_lin'))

            if reuse:
                tf.histogram_summary('logits/example', h5)
                tf.histogram_summary('d_h0_conv/act/example', h0)
                tf.histogram_summary('d_h1_conv/act/example', h1)
                tf.histogram_summary('d_h2_conv/act/example', h2)
                tf.histogram_summary('d_h3_conv/act/example', h3)
                tf.histogram_summary('d_h4_conv/act/example', h4)
            else:
                tf.histogram_summary('logits/forgery', h5)
                tf.histogram_summary('d_h0_conv/act/forgery', h0)
                tf.histogram_summary('d_h1_conv/act/forgery', h1)
                tf.histogram_summary('d_h2_conv/act/forgery', h2)
                tf.histogram_summary('d_h3_conv/act/forgery', h3)
                tf.histogram_summary('d_h4_conv/act/forgery', h4)

            return h5

    def b64_d_atrous(self, image, reuse=False, dev='/gpu:0'):
        with tf.variable_scope('discriminator'):
            if reuse:
                tf.get_variable_scope().reuse_variables()

            batch_size = self.batch_size
            df_dim = 32

            with tf.device(dev):
                h0 = lrelu(
                    batch_norm(
                        atrous_conv2d(image, df_dim, rate=1, name='d_h0_conv'
),
                        train=True, reuse=reuse, name='d_bn0')
                    )

                h1 = lrelu(
                    batch_norm(
                        atrous_conv2d(h0, df_dim*2, rate=2, name='d_h1_conv')
,
                        train=True, reuse=reuse, name='d_bn1')
                    )

                h2 = lrelu(
                    batch_norm(
                        atrous_conv2d(h1, df_dim*2, rate=4, name='d_h2_conv')
,
                        train=True, reuse=reuse, name='d_bn2')
                    )

                h3 = lrelu(
                    batch_norm(
                        atrous_conv2d(h2, df_dim*2, rate=8, name='d_h3_conv')
,
                        train=True, reuse=reuse, name='d_bn3')
                    )

                h4 = lrelu(
                    batch_norm(
                        atrous_conv2d(h3, df_dim*2, rate=16, name='d_h4_conv'
),
                        train=True, reuse=reuse, name='d_bn4')
                    )

                h5 = tf.nn.sigmoid(batch_norm(
                        atrous_conv2d(h4, 1, rate=1, name='d_h5_conv'),
                        train=True, reuse=reuse, name='d_bn5')
                    )

                h = flatten(tf.nn.avg_pool(h5, [1, 6, 6, 1], [1, 3, 3, 1], p
adding='SAME'))

            if reuse:
                tf.histogram_summary('logits/example', h5)
                tf.histogram_summary('d_h0_conv/act/example', h0)
                tf.histogram_summary('d_h1_conv/act/example', h1)
                tf.histogram_summary('d_h2_conv/act/example', h2)
                tf.histogram_summary('d_h3_conv/act/example', h3)
                tf.histogram_summary('d_h4_conv/act/example', h4)
            else:
                tf.histogram_summary('logits/forgery', h5)
                tf.histogram_summary('d_h0_conv/act/forgery', h0)
                tf.histogram_summary('d_h1_conv/act/forgery', h1)
                tf.histogram_summary('d_h2_conv/act/forgery', h2)
                tf.histogram_summary('d_h3_conv/act/forgery', h3)
                tf.histogram_summary('d_h4_conv/act/forgery', h4)

            # h4 = tf.nn.sigmoid(linear(h3,1, 'd_h4_lin'))

            return h



    def mnist_discriminator_wide(self, image, reuse=False):
        if reuse:
            tf.get_variable_scope().reuse_variables()

        batch_size = self.batch_size
        df_dim = 32
        print 'init_disc:'

        h0 = conv2d(image, df_dim, d_h=1, d_w=1, name='d_h0_conv')
        print h0.get_shape()

        h1 = wide_basic_conv(h0, df_dim*2, 2, name='d_h1_conv', reuse=reuse)
        print h1.get_shape()

        h2 = wide_basic_conv(h1, df_dim*4, 2, name='d_h2_conv', reuse=reuse)
        print h2.get_shape()

        h3 = wide_basic_conv(h2, df_dim*8, 2, name='d_h3_conv', reuse=reuse)
        h3 = lrelu(batch_norm(h3, train=True, reuse=reuse, name='d_h3_bn'))
        print h3.get_shape()

        h3 = lrelu(linear(flatten(h3), 256, 'd_h3_lin'))
        print h3.get_shape()
        h3 = minibatch_features(h3)

        print h3.get_shape()
        h4 = tf.nn.sigmoid(linear(h3,1, 'd_h4_lin'))
        # exit()
        return h4

    def buildings_discriminator_atrous(self, image, reuse=False):
        with tf.device("/gpu:1"):
            with tf.variable_scope('discriminator'):
                if reuse:
                    tf.get_variable_scope().reuse_variables()

                batch_size = self.batch_size
                df_dim = 4

                h0 = lrelu(
                    batch_norm(
                        atrous_conv2d(image, df_dim, rate=1, name='d_h0_conv'
),
                        train=True, reuse=reuse, name='d_bn0')
                    )
                h1 = lrelu(
                    batch_norm(
                        atrous_conv2d(h0, df_dim*2, rate=2, name='d_h1_conv')
,
                        train=True, reuse=reuse, name='d_bn1')
                    )
                h2 = lrelu(
                    batch_norm(
                        atrous_conv2d(h1, df_dim*2, rate=4, name='d_h2_conv')
,
                        train=True, reuse=reuse, name='d_bn2')
                    )
                h3 = lrelu(
                    batch_norm(
                        atrous_conv2d(h2, df_dim*2, rate=8, name='d_h3_conv')
,
                        train=True, reuse=reuse, name='d_bn3')
                    )

                h4 = lrelu(batch_norm(
                        atrous_conv2d(h3, df_dim*2, rate=16, name='d_h4_conv'
),
                        train=True, reuse=reuse, name='d_bn4')
                    )

                h5 = lrelu(batch_norm(
                        atrous_conv2d(h4, df_dim*4, rate=32, name='d_h5_conv'
),
                        train=True, reuse=reuse, name='d_bn5')
                    )

                h6 = lrelu(batch_norm(
                        atrous_conv2d(h5, df_dim*4, rate=64, name='d_h6_conv'
),
                        train=True, reuse=reuse, name='d_bn6')
                    )

                h7 = tf.nn.sigmoid(batch_norm(
                        atrous_conv2d(h6, 1, rate=1, name='d_h7_conv'),
                        train=True, reuse=reuse, name='d_bn7')
                    )

                if reuse:
                    tf.histogram_summary('logits/example', h7)
                else:
                    tf.histogram_summary('logits/forgery', h7)

                h7 = tf.reduce_mean(h7, 0)

                # h4 = tf.nn.sigmoid(linear(h3,1, 'd_h4_lin'))

                return h7


    def buildings_discriminator(self, image, reuse=False):
        with tf.device("/gpu:1"):
            # 256x256
            if reuse:
                tf.get_variable_scope().reuse_variables()

            batch_size = self.batch_size
            df_dim = 32

            h0 = lrelu(
                batch_norm(
                    conv2d(image, df_dim, d_h=2, d_w=2, name='d_h0_conv'),
                    train=True, reuse=reuse, name='d_bn0')
                )
            # 128x128

            h1 = lrelu(
                batch_norm(
                    conv2d(h0, df_dim*2, d_h=2, d_w=2, name='d_h1_conv'),
                    train=True, reuse=reuse, name='d_bn1')
                )
            # 64x64

            h2 = lrelu(
                batch_norm(
                    conv2d(h1, df_dim*4, d_h=1, d_w=1, name='d_h2_conv'),
                    train=True, reuse=reuse, name='d_bn2')
                )
            # 64x64

            h3 = lrelu(
                batch_norm(
                    conv2d(h2, df_dim*8, d_h=2, d_w=2, name='d_h3_conv'),
                    train=True, reuse=reuse, name='d_bn3')
                )
            # 32x32

            h4 = lrelu(
                batch_norm(
                    conv2d(h3, df_dim*8, d_h=1, d_w=1, name='d_h4_conv'),
                    train=True, reuse=reuse, name='d_bn4')
                )
            # 32x32

            h5 = lrelu(
                batch_norm(
                    conv2d(h4, df_dim*8, d_h=2, d_w=2, name='d_h5_conv'),
                    train=True, reuse=reuse, name='d_bn5')
                )
            # 16x16

            h6 = lrelu(
                batch_norm(
                    conv2d(h4, df_dim*8, d_h=2, d_w=2, name='d_h6_conv'),
                    train=True, reuse=reuse, name='d_bn6')
                )
            # 8x8

            # a0 = atrous_conv2d(h2, 2*df_dim, rate=1, name='d_a0')
            # a0 = lrelu(batch_norm(a0, train=True, reuse=reuse, name='d_a0_
bn'))
            # a0 = conv2d(a0, 2*df_dim, d_h=2, d_w=2, name='d_a0_1')
            # a0 = lrelu(batch_norm(a0, train=True, reuse=reuse, name='d_a0_
bn1'))

            # a1 = atrous_conv2d(a0, 2*df_dim, rate=2, name='d_a1')
            # a1 = lrelu(batch_norm(a0, train=True, reuse=reuse, name='d_a1_
bn'))
            # a1 = conv2d(a1, 2*df_dim, d_h=2, d_w=2, name='d_a1_1')
            # a1 = lrelu(batch_norm(a1, train=True, reuse=reuse, name='d_a1_
bn1'))

            # a2 = atrous_conv2d(a1, 2*df_dim, rate=4, name='d_a2')
            # a2 = lrelu(batch_norm(a2, train=True, reuse=reuse, name='d_a2_
bn'))
            # a2 = conv2d(a2, 2*df_dim, d_h=2, d_w=2, name='d_a2_1')
            # a2 = lrelu(batch_norm(a2, train=True, reuse=reuse, name='d_a2_
bn1'))

            # a2 = tf.nn.avg_pool(a2, [1, 8, 8, 1], [1,8,8,1], padding='SAME
')
            h6 = tf.nn.avg_pool(h6, [1, 8, 8, 1], [1,8,8,1], padding='SAME'
)
            h6 = flatten(h6)
            # h6 = tf.concat(1, [flatten(h6), flatten(a2)])


            h6 = lrelu(linear(h6, 512, 'd_h6_lin'))
            h6 = minibatch_features(h6)

            h7 = tf.nn.sigmoid(linear(h6,1, 'd_h7_lin'))

            return h7

    def face_discriminator_atrous(self, image, reuse=False):
        with tf.device("/gpu:1"):
            with tf.variable_scope('discriminator'):
                if reuse:
                    tf.get_variable_scope().reuse_variables()

                batch_size = self.batch_size
                df_dim = 4

                h0 = lrelu(
                    batch_norm(
                        atrous_conv2d(image, df_dim, rate=1, name='d_h0_conv'
),
                        train=True, reuse=reuse, name='d_bn0')
                    )
                h1 = lrelu(
                    batch_norm(
                        atrous_conv2d(h0, df_dim*2, rate=2, name='d_h1_conv')
,
                        train=True, reuse=reuse, name='d_bn1')
                    )
                h2 = lrelu(
                    batch_norm(
                        atrous_conv2d(h1, df_dim*2, rate=4, name='d_h2_conv')
,
                        train=True, reuse=reuse, name='d_bn2')
                    )
                h3 = lrelu(
                    batch_norm(
                        atrous_conv2d(h2, df_dim*2, rate=8, name='d_h3_conv')
,
                        train=True, reuse=reuse, name='d_bn3')
                    )

                h4 = lrelu(batch_norm(
                        atrous_conv2d(h3, df_dim*2, rate=16, name='d_h4_conv'
),
                        train=True, reuse=reuse, name='d_bn4')
                    )
                h5 = lrelu(batch_norm(
                        atrous_conv2d(h4, df_dim*4, rate=32, name='d_h5_conv'
),
                        train=True, reuse=reuse, name='d_bn5')
                    )

                # h6 = lrelu(batch_norm(
                #         atrous_conv2d(h5, df_dim*4, rate=64, name='d_h6_co
nv'),
                #         train=True, reuse=reuse, name='d_bn6')
                #     )

                h7 = lrelu(batch_norm(
                        atrous_conv2d(h5, 1, rate=1, name='d_h7_conv'),
                        train=True, reuse=reuse, name='d_bn7')
                    )

                if reuse:
                    tf.histogram_summary('logits/example', h7)
                else:
                    tf.histogram_summary('logits/forgery', h7)

                # h7 = tf.reduce_mean(h7, 0)

                h7 = tf.nn.sigmoid(linear(flatten(h7),1, 'd_h4_lin'))

                return h7

def minibatch_features(h, name='d_mb'):
    with tf.variable_scope(name):
        batch_size = h.get_shape().as_list()[0]
        n_kernels = 32 #64
        dim_per_kernel =  8 #12
        x = linear(h, n_kernels * dim_per_kernel, "d_minibatch_disc")
        activation = tf.reshape(x, (batch_size, n_kernels, dim_per_kernel))

        big = np.zeros((batch_size, batch_size), dtype='float32')
        big += np.eye(batch_size)
        big = tf.expand_dims(big, 1)
        mask = 1. - big

        abs_dif = tf.reduce_sum(tf.abs(tf.expand_dims(activation, 3) - tf.ex
pand_dims(tf.transpose(activation, [1, 2, 0]), 0)), 2)

        masked = tf.exp(-abs_dif) * mask
        abs_dif = tf.reduce_sum(masked, 2) / tf.reduce_sum(mask)

        return tf.concat(1, [h, abs_dif])
```

```
import math
import numpy as np
import tensorflow as tf

from tensorflow.python.framework import ops
from tensorflow.python.ops import array_ops
from tensorflow.contrib.layers.python.layers import utils as util

from utils import *

def style(x):
    shape = x.get_shape().as_list()
    c = shape[-1]
    #h = shape[1]
    #w = shape[2]
    batch_size = shape[0]

    phi = tf.reshape(x, [batch_size, -1, c])
    print phi.get_shape().as_list()
    print tf.transpose(phi, [0, 2, 1]).get_shape().as_list()
    g = tf.batch_matmul(tf.transpose(phi, [0, 2, 1]), phi) / (c)

    assert g.get_shape().as_list()[1] == c

    return g

def l2_norm(input_,dim):
        return tf.reduce_mean(tf.square(input_),reduction_indices=dim,keep_d
ims=True)

def combinations(X, Y):
        batch_size = int(X.get_shape()[0])
        num_latents = int(X.get_shape()[1])

        X = tf.reshape(X,[batch_size,1,num_latents])
        Y = tf.reshape(Y,[batch_size,1,num_latents])

        X = tf.reshape(tf.tile(X, [1, batch_size, 1]),[batch_size**2,num_lat
ents])
        Y = tf.reshape(tf.transpose(tf.tile(Y, [1, batch_size, 1]),[1, 0, 2]
),[batch_size**2,num_latents])

        return X, Y

def mmd(x,y):
        def kernel(x,y,sigma=1):
                return tf.exp(-l2_norm(x-y,1)/2/sigma)
        def kw(x,y,k=1):
                return kernel(x,y,sigma=1*k) + \
                kernel(x,y,sigma=5*k) +\
                kernel(x,y,sigma=10*k) +\
                kernel(x,y,sigma=20*k)

        batch_size = int(x.get_shape()[0])

        def mask(x):
                m = []
                ind = 0
                for i in xrange(0,batch_size**2):
                        if i == ind:
                                m.append(False)
                                ind += batch_size + 1
                        else:
                                m.append(True)

                return tf.boolean_mask(x, m)

        # X, Y = map(mask,combinations(x,y))
        # Y_, X_ = map(mask,combinations(y,x))

        X, Y = combinations(x,y)
        Y_, X_ = combinations(y,x)

        # X_, X__ = combinations(tf.slice(x,[0,0],[batch_size/2,-1]),tf.slic
e(x,[batch_size/2,0],[-1,-1]))
        # Y_, Y__ = combinations(tf.slice(y,[0,0],[batch_size/2,-1]),tf.slic
e(y,[batch_size/2,0],[-1,-1]))

        XY = tf.reduce_mean(kw(X,Y))

        XX = tf.reduce_mean(kw(X,X_))
        YY = tf.reduce_mean(kw(Y,Y_))

        # XX = tf.reduce_mean(kw(X__,X_))
        # YY = tf.reduce_mean(kw(Y__,Y_))

        H = tf.sqrt(-2*XY + XX + YY)
        # H = tf.maximum(H,0)

        return H



def layer_norm(x_, name):
    with tf.variable_scope(name):
        r = len(x_.get_shape().as_list())
        if r == 4:
            bias = tf.get_variable('gain', x_.get_shape().as_list()[-1],
                                   initializer=tf.constant_initializer(value=0.0
))
            gain = tf.get_variable('bias', x_.get_shape().as_list()[-1],
                                initializer=tf.constant_initializer(value=1.0))
            if tf.get_variable_scope().reuse == False:
                variable_summaries(name + '/bias', bias)
                variable_summaries(name + '/gain', gain)
            m, s = tf.nn.moments(x_, axes=[1,2,3], keep_dims=True)
            return gain*(x_ - m)/tf.sqrt(s) + bias
        elif r ==2:
            bias = tf.get_variable('gain', x_.get_shape().as_list()[-1],
                                   initializer=tf.constant_initializer(value=0.0
))
            gain = tf.get_variable('bias', x_.get_shape().as_list()[-1],
                                initializer=tf.constant_initializer(value=1.0))
            if tf.get_variable_scope().reuse == False:
                variable_summaries(name + '/bias', bias)
                variable_summaries(name + '/gain', gain)
            m, s = tf.nn.moments(x_, axes=[1], keep_dims=True)
            return gain*(x_ - m)/tf.sqrt(s) + bias
        else:
            print('rip')
            exit(-1)




#def layer_norm(x_):
#       m, s = tf.nn.moments(x_, axes=[1,2,3], keep_dims=True)
#       return (x_ - m)/tf.sqrt(s)

def variable_summaries(name, var):
  """Attach a lot of summaries to a Tensor."""
  with tf.name_scope('summaries'):
    mean = tf.reduce_mean(var)
    tf.scalar_summary('mean/' + name, mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_sum(tf.square(var - mean)))
    tf.scalar_summary('sttdev/' + name, stddev)
    tf.scalar_summary('max/' + name, tf.reduce_max(var))
    tf.scalar_summary('min/' + name, tf.reduce_min(var))
    tf.histogram_summary(name, var)

def flatten(inputs,
            outputs_collections=None,
            scope=None):
  """Flattens the input while maintaining the batch_size.
  Assumes that the first dimension represents the batch.
 Args:
  inputs: a tensor of size [batch_size, ...].
  outputs_collections: collection to add the outputs.
  scope: Optional scope for op_scope.
 Returns:
  a flattened tensor with shape [batch_size, k].
 Raises:
  ValueError: if inputs.shape is wrong.
 """
  if len(inputs.get_shape()) < 2:
    raise ValueError('Inputs must be have a least 2 dimensions')
  dims = inputs.get_shape()[1:]
  k = dims.num_elements()
  with ops.op_scope([inputs], scope, 'Flatten') as sc:
    outputs = array_ops.reshape(inputs, [-1, k])
    return util.collect_named_outputs(outputs_collections, sc, outputs)

def batch_norm(input_, train=True, reuse=False, name='batch_norm'):
    return tf.contrib.layers.batch_norm(input_, center=True, scale=True, reu
se=reuse, is_training=train, scope=name)

def binary_cross_entropy_with_logits(logits, targets, name=None):
    """Computes binary cross entropy given `logits`.

  For brevity, let `x = logits`, `z = targets`.  The logistic loss is

      loss(x, z) = - sum_i (x[i] * log(z[i]) + (1 - x[i]) * log(1 - z[i]))
```

```
   Args:
      logits: A ‘Tensor‘ of type ‘float32‘ or ‘float64‘.
      targets: A ‘Tensor‘ of the same type and shape as ‘logits‘.
   """
    eps = 1e-12
    with ops.op_scope([logits, targets], name, "bce_loss") as name:
        logits = ops.convert_to_tensor(logits, name="logits")
        targets = ops.convert_to_tensor(targets, name="targets")
        return tf.reduce_mean(-(logits * tf.log(targets + eps) +
                              (1. - logits) * tf.log(1. - targets + eps)))

def conv_cond_concat(x, y):
    """Concatenate conditioning vector on feature map axis."""
    x_shapes = x.get_shape()
    y_shapes = y.get_shape()
    return tf.concat(3, [x, y*tf.ones([x_shapes[0], x_shapes[1], x_shapes[2]
, y_shapes[3]])])

def conv2d(input_, output_dim,
           k_h=3, k_w=3, d_h=2, d_w=2, stddev=0.02,
           name="conv2d"):
    with tf.variable_scope(name):
        w = tf.get_variable('w', [k_h, k_w, input_.get_shape()[-1], output_
dim],
                            initializer=tf.contrib.layers.variance_scaling_i
nitializer())
        conv = tf.nn.conv2d(input_, w, strides=[1, d_h, d_w, 1], padding='S
AME')

        if tf.get_variable_scope().reuse == False:
            variable_summaries(name + '/w', w)
            tf.histogram_summary(name + '/preact', conv)

        return conv

def atrous_conv2d(input_, output_dim, k_h=3, k_w=3, rate=2, stddev=0.02, nam
e='atrous_conv2d'):
    with tf.variable_scope(name):
        w = tf.get_variable('w', [k_h, k_w, input_.get_shape()[-1], output_
dim],
                            initializer=tf.contrib.layers.variance_scaling_i
nitializer())
        conv = tf.nn.atrous_conv2d(input_, w, rate, padding='SAME')
        if tf.get_variable_scope().reuse == False:
            variable_summaries(name + '/w', w)
            tf.histogram_summary(name + '/preact', conv)

        return conv

# def wide_basic_conv(input_, output_dim, stride, name, reuse=False):
#     path1 = tf.nn.relu(batch_norm(
#         input_, train=True, reuse=reuse, name=name+'_bn0'))
#     path1 = conv2d(path1, output_dim, d_h=stride, d_w=stride, name=name+'_
conv1')
#     path1 = tf.nn.relu(batch_norm(
#         path1, train=True, reuse=reuse, name=name+'_bn1'))
#     path1 = conv2d(path1, output_dim, d_h=1, d_w=1, name=name+'_conv2')
#     if input_.get_shape()[-1] == output_dim and stride==1:
#         path2 = input_
#     else:
#         path2 = conv2d(input_, output_dim, k_h=1, k_w=1, d_h=stride, d_w=s
tride, name=name+'_conv_sc')

#     return path1 + path2

def wide_basic_conv(input_, output_dim, stride, name, reuse=False):
    with tf.name_scope(name):
        path1 = input_
        if input_.get_shape()[-1] == output_dim and stride==1:
            path2 = path1
            path1 = tf.nn.relu(batch_norm(
                path1, train=True, reuse=reuse, name=name+'_bn0'))
        else:
            path1 = tf.nn.relu(batch_norm(
                path1, train=True, reuse=reuse, name=name+'_bn0'))
            path2 = path1
            path2 = conv2d(path2, output_dim, k_h=1, k_w=1, d_h=stride, d_w=
stride, name=name+'_conv_sc')

        path1 = conv2d(path1, output_dim, d_h=stride, d_w=stride, name=name+
'_conv1')
        path1 = tf.nn.relu(batch_norm(
            path1, train=True, reuse=reuse, name=name+'_bn1'))
        path1 = conv2d(path1, output_dim, d_h=1, d_w=1, name=name+'_conv2')

        return path1 + path2

def wide_basic_deconv(input_, output_shape, stride, name, reuse=False):
    path1 = tf.nn.relu(batch_norm(
        input_, train=True, reuse=reuse, name=name+'_bn0'))
    path1 = deconv2d(path1, output_shape, d_h=stride, d_w=stride, name=name+
'_conv1')
    path1 = tf.nn.relu(batch_norm(
        path1, train=True, reuse=reuse, name=name+'_bn1'))
    path1 = deconv2d(path1, output_shape, d_h=1, d_w=1, name=name+'_conv2')

    if input_ == output_shape:
        path2 = input_
    else:
        path2 = tf.image.resize_images(input_, output_shape[1], output_shape
[2])
        # output_shape_ = list(output_shape)
        # output_shape_[3] = output_shape[3] // 2
        # path2 = deconv2d(input_, output_shape_, k_h=3, k_w=3, d_h=stride,
d_w=stride, name=name+'_conv_sc1')
        # path2 = add_feats(path2, output_shape_[3])
        path2 = deconv2d(path2, output_shape, k_h=1, k_w=1, d_h=1, d_w=1, na
me=name+'_conv_sc2')

    return path1 + path2

def add_feats(x, nFeats):
    s = x.get_shape().as_list()
    # import ipdb; ipdb.set_trace()
    s[3] = nFeats
    h = tf.random_uniform(s, minval=-1, maxval=1)
    return tf.concat(3, [x,h])

def artifact_mask(x):
    eps = 1e-11
    return tf.abs(tf.round(tf.div(x,x+eps)) - 1)

# def deconv2d(input_, output_shape,
#              k_h=6, k_w=6, d_h=2, d_w=2, stddev=0.02,
#              name="deconv2d"):
#     with tf.variable_scope(name):
#         # filter : [height, width, output_channels, in_channels]
#         w = tf.get_variable('w', [k_h, k_h, output_shape[-1], input_.get_s
hape()[-1]],
#                             initializer=tf.random_normal_initializer(stdde
v=stddev))
#         return tf.nn.conv2d_transpose(input_, w, output_shape=output_shape
,
#                                strides=[1, d_h, d_w, 1])

def deconv2d(input_, output_shape,
             k_h=6, k_w=6, d_h=2, d_w=2, stddev=0.02,
             name="deconv2d", with_w=False,
             init_bias=0.):
    # designed to reduce padding and stride artifacts in the generator

    # If the following fail, it is hard to avoid grid pattern artifacts
    # assert k_h % d_h == 0
    # assert k_w % d_w == 0

    with tf.variable_scope(name):
        # filter : [height, width, output_channels, in_channels]
        w = tf.get_variable('w', [k_h, k_w, output_shape[-1], input_.get_sh
ape()[-1]],
                            initializer=tf.contrib.layers.variance_scaling_i
nitializer())

        def check_shape(h_size, im_size, stride):
            if h_size != (im_size + stride - 1) // stride:
                print "Need h_size == (im_size + stride - 1) // stride"
                print "h_size: ", h_size
                print "im_size: ", im_size
                print "stride: ", stride
                print "(im_size + stride - 1) / float(stride): ", (im_size + stride - 1) /
float(stride)
                raise ValueError()

        check_shape(int(input_.get_shape()[1]), output_shape[1], d_h)
        check_shape(int(input_.get_shape()[2]), output_shape[2], d_w)

        deconv = tf.nn.conv2d_transpose(input_, w, output_shape=[output_shap
e[0],
            output_shape[1], output_shape[2], output_shape[3]],
                                strides=[1, d_h, d_w, 1])
        # deconv = tf.slice(deconv, [0, k_h // 2, k_w // 2, 0], output_shape
)
```

```
        biases = tf.get_variable('biases', [output_shape[-1]], initializer=tf
.constant_initializer(init_bias))
        deconv = tf.reshape(tf.nn.bias_add(deconv, biases), deconv.get_shape
())

        if tf.get_variable_scope().reuse == False:
            variable_summaries(name + '/w', w)
            variable_summaries(name + '/b', biases)
            tf.histogram_summary(name + '/preact', deconv)

        if with_w:
            return deconv, w, biases
        else:
            return deconv

def atrous_deconv2d(input_, output_shape, k_h=6, k_w=6, d_h=2, d_w=2, rate=2
,name='atrous_deconv'):
    input_shape = input_.get_shape().as_list()
    h = input_shape[1]
    w = input_shape[2]
    def pad_comp(l, rate):
        if np.mod(h, rate) == 0:
            pad1 = 0
            pad2 = 0
        else:
            pad = np.mod(h, rate)
            if np.mod(pad, 2) == 0:
                pad1 == pad // 2
                pad2 == pad // 2
            else:
                pad1 = pad
                pad2 = 0

        return pad1, pad2
    padt, padb = pad_comp(h, rate)
    padl, padr = pad_comp(w, rate)
    pads = [[padt, padb],[padl, padr]]

    y = tf.space_to_batch(input_, pads, rate)
    print y.get_shape()
    batch_size_ = y.get_shape().as_list()[0]
    y = deconv2d(y, (batch_size_, output_shape[1]/2, output_shape[2]/2, outp
ut_shape[3]), k_w=k_w, k_h=k_h, name=name)
    print y.get_shape()
    y = tf.batch_to_space(y, crops=pads, block_size=rate)
    return y

def lrelu(x, leak=0.2, name="lrelu"):
    with tf.variable_scope(name):
        f1 = 0.5 * (1 + leak)
        f2 = 0.5 * (1 - leak)
        return f1 * x + f2 * abs(x)

def linear(input_, output_size, scope=None, stddev=0.02):
    shape = input_.get_shape().as_list()

    with tf.variable_scope(scope or "Linear"):
        matrix = tf.get_variable("Matrix", [shape[1], output_size], tf.float3
2,
                                    tf.contrib.layers.variance_scaling_initiali
zer())
        biases = tf.get_variable('biases', [output_size], initializer=tf.cons
tant_initializer(0))
        return tf.matmul(input_, matrix) + biases
```

```
import tensorflow as tf

def read_and_decode_places(filename_queue):
  reader = tf.TFRecordReader()
  _, serialized_example = reader.read(filename_queue)
  features = tf.parse_single_example(
      serialized_example,
      # Defaults are not specified since both keys are required.
      features={
          'image': tf.FixedLenFeature([], tf.string),
      })

  image = tf.image.decode_jpeg(features['image'], channels=3)
  image.set_shape([256,256,3])
  return tf.image.convert_image_dtype(image, tf.float32)

def read_and_decode_faces(filename_queue):
  reader = tf.TFRecordReader()
  _, serialized_example = reader.read(filename_queue)
  features = tf.parse_single_example(
      serialized_example,
      # Defaults are not specified since both keys are required.
      features={
          'image': tf.FixedLenFeature([], tf.string),
      })

  image = tf.image.decode_jpeg(features['image'], channels=3)
  image.set_shape([218,178,3])
  image = tf.image.transpose_image(image)
  return tf.image.convert_image_dtype(image, tf.float32)

def read_and_decode_mnist(filename_queue):
  reader = tf.TFRecordReader()
  _, serialized_example = reader.read(filename_queue)
  features = tf.parse_single_example(
      serialized_example,
      # Defaults are not specified since both keys are required.
      features={
          'image': tf.FixedLenFeature([], tf.string),
      })

  image = tf.image.decode_png(features['image'], channels=1)
  image.set_shape([28,28,1])
  return tf.image.convert_image_dtype(image, tf.float32)

def read_and_decode_b64(filename_queue):
  reader = tf.TFRecordReader()
  _, serialized_example = reader.read(filename_queue)
  features = tf.parse_single_example(
      serialized_example,
      # Defaults are not specified since both keys are required.
      features={
          'image': tf.FixedLenFeature([], tf.string),
      })

  image = tf.image.decode_png(features['image'], channels=3)
  image.set_shape([64,64,3])
  return tf.image.convert_image_dtype(image, tf.float32)
```

```
def build_input_queue(name):
    if name == 'mnist':
        path = "../mnist_preproc/mnist_png/mnist.tfrecords"
        batch_size = 128
        read_and_decode = read_and_decode_mnist
    elif name == 'buildings':
        path = "/mnt/d4/places/data/vision/torralba/deeplearning/images256/b/building_facade/buildings.
tfrecords"
        batch_size = 16
        read_and_decode = read_and_decode_places
    elif name == 'faces':
        path = "/mnt/d2/celeba/img_align_celeba/celeba.tfrecords"
        batch_size = 16
        read_and_decode = read_and_decode_faces
    elif name == 'b64':
        path = "/mnt/d4/places/data/vision/torralba/deeplearning/images256/b/building_facade/buildings6
4.tfrecords"
        batch_size = 32
        read_and_decode = read_and_decode_b64
    else:
        print 'RIP'
        exit(-1)

    filename_queue = tf.train.string_input_producer([path], num_epochs=1000)
    image = read_and_decode(filename_queue)

    images = tf.train.shuffle_batch(
            [image], batch_size=batch_size, num_threads=4,
            capacity=1000 + 3 * batch_size,
            # Ensures a minimum amount of shuffling of examples.
            min_after_dequeue=1000)

    return images, batch_size
```