CSE 253 Final Project Proposal

Sainan Liu
A13291871
sal131@eng.ucsd.edu

Hao-en Sung
A53204772
wrangle1005@gmail.com

Shiwei Song
A53206591
shs163@eng.ucsd.edu

Haifeng Huang
A53208823
hah086@eng.ucsd.edu

1 Main Idea

The main idea of this project is to explore techniques that can improve stochasticity in the output of the conditional generative adversarial networks(cGAN) based on top of the model used in a Image-to-Image Translation paper[1].

2 Data Source

To compare the result, we will use the same set of data that is provided by the authors[2]

Table 1: Datasets.

# Dataset Name	Task	Training No.	Validation No.	Test No.
Cityscapes[3]	Semantic labels ↔ photo	2975	500	-
CMP Facades[4]	Architectural labels ↔ photo	400	100	106
Google Maps	Map ↔ aerial photo	1096	1098	-
Edges to Shoes[5]	Edges ↔ photo	49825	200	-
Edges to Handbags[6]	Edges ↔ photo	138567	200	-

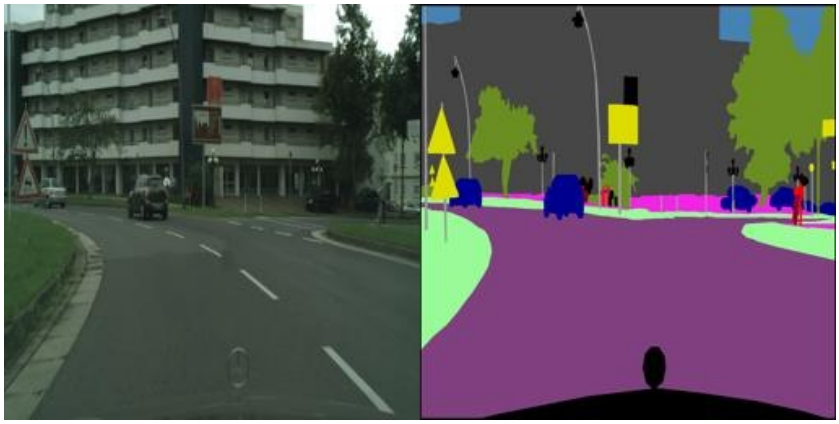


Figure 1: Sample image from Cityscapes

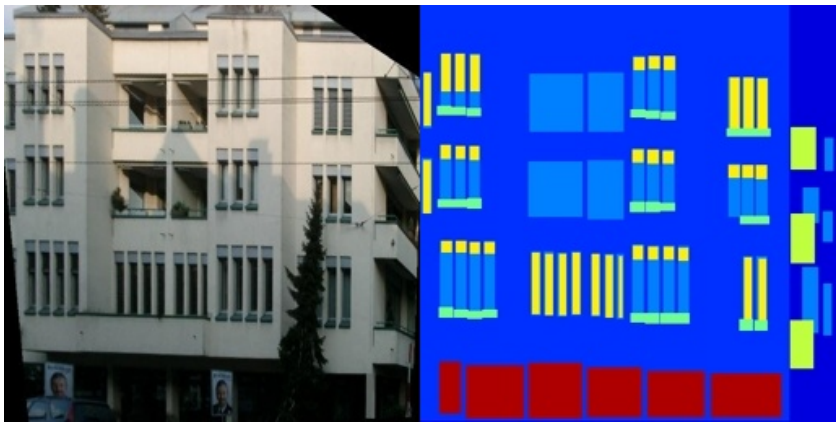


Figure 2: Sample image from CMP Facades

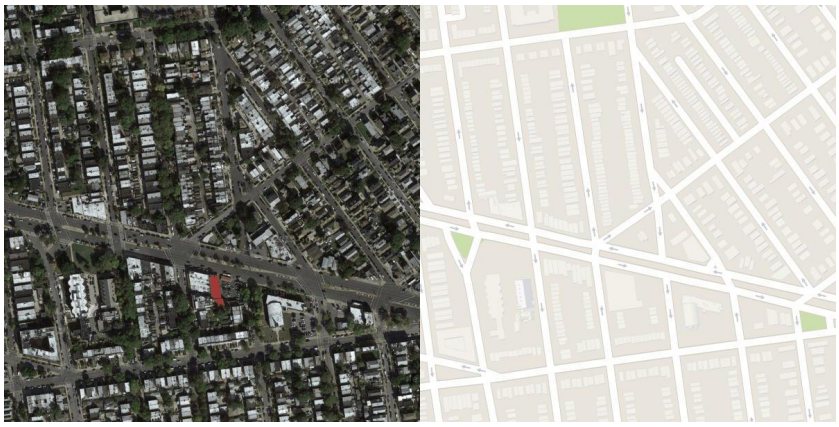


Figure 3: Sample image from Google Maps

Figure 4: Sample image from Edges to Shoes

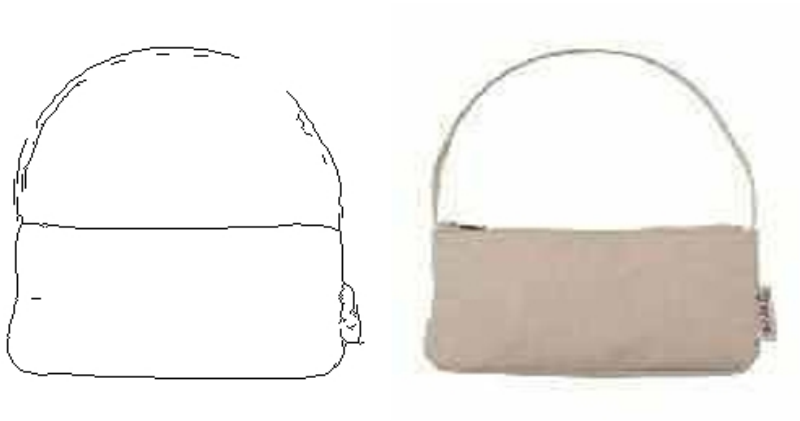


Figure 5: Sample image from Edges to Handbags

3 Network Architecture

3.1 Introduction to cGANs

The conditional Generative Adversarial Network consists two neural networks: a generator **G** and a discriminator **D**. Given a sample pair (X, T).

The generator will take a sample X and a random noise Z as input and generates output Y as close as T.

$$Y = G(X, Z)$$

The discriminator will take a pair (X, Y) and try to figure out whether Y is a real target or a fake target generated by the generator.

$$D(X, Y) = \begin{cases} 1 & D \text{ regards } Y \text{ as a real target} \\ 0 & D \text{ regards } Y \text{ as a fake target} \end{cases}$$

The loss function we will use to train the two networks are

$$L_D = \sum -t \log(D(x, y)) - (1 - t) \log(1 - D(x, G(x, z)))$$

$$L_G = \sum -log(D(x, G(x, z)))$$

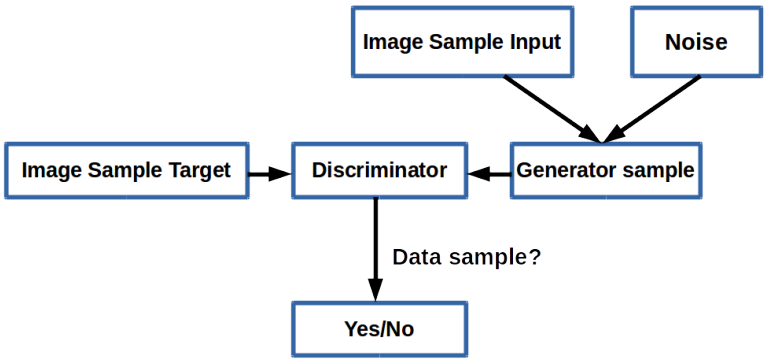


Figure 6: Generalized cGAN model

3.2 Detailed Structure

The two networks will all have a overall structure of convolution-BatchNorm-ReLu.

For the generator, it have a tanh activation function in the output layer. We will use the U-Net model described in [7]. So layer $n - i$ will have a connection with layer i.

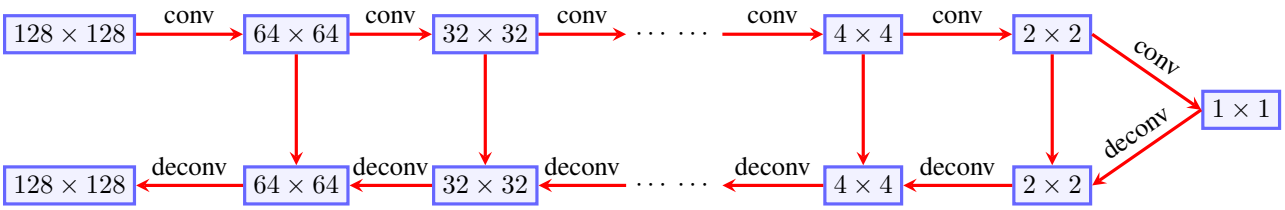


Figure 7: Structure of the generator network

For the discriminator, it will have a sigmoid activation function in the output layer. So the output would be the probability of Y as a real target.

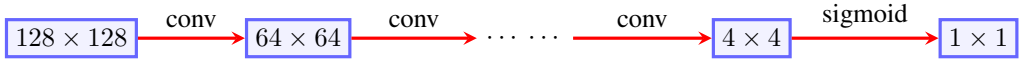


Figure 8: Structure of the discriminator network

4 Inspiration

In [1], they mentioned that in order to capture the full entropy of the conditional distributions that the cGAN models, we need to design a cGAN that produces stochastic output. [1] mentioned that, in previous research, [8] has tried to add Gaussian noise as an input to the generator in addition to x, but both [1] and [9] found that the generator would simply learn to ignore the noise. Therefore,

[1] proposed to provide noise only in the form of dropout on several layers of the generator at both training and test time. However, they observe very minor stochasticity in the output of their nets. Hence we decided to build on top of their network, and further explore ways to improve the stochasticity of the output for cGAN.

5 Proposed changes

5.1 Add temperature

We will try to add a layer of softmax with a temperature parameter to add stochasticity to the network outputs

5.2 Add weights to neurons

We can also try to randomly assign weights to neurons in order to increase stochasticity of the output.

5.3 Add another generator

We can add another generator that generates the sample from pure noise, and feed the result also into the discriminator, where the discriminator will try to differentiate between real data and conditional generator sample, as well as conditional generator sample and noise generator sample. As the three networks (two generator networks + one discriminator network) competes with each other, we hope that this will allow the conditional generator sample to gain more stochasticity in its output.

6 References

[1] Isola, P., Zhu, J., Zhou, T. & Efros, A.A. (2016) *Image-to-Image Translation with Conditional Adversarial Networks*,https://arxiv.org/pdf/1611.07004v1.pdf

[2] Isola, P. Zhu, J., Zhou, T. & Efros, A.A. (2016) *pix2pix datasets*, http://people.eecs.berkeley.edu/ tinghuiz/projects/pix2pix/datasets/

[3] Cordts,M., Omran,M., Ramos, S.,Rehfeld, T.,Enzweiler,M., Benenson,R. ,Franke, U., Roth, S. & Schiele, B. (2016). *The cityscapes dataset for semantic urban scene understanding.* CVPR

[4] Gatys,L. A., Ecker, A. S., & Bethge, M. (2016) *Image style transfer using convolutional neural networks.* CVPR

[5] Yu,A. & Grauman, K.(2014) *Fine-Grained Visual Comparisons with Local Learning* CVPR

[6]Zhu,J., Krhenbhl,P. , Shechtman, E. & Efros, A. A. (2016) *Generative visual manipulation on the natural image manifold* ECCV

[7]Ronneberger,O., Fischer, P. & Brox, T. (2015)*U-net: Convolutional networks for biomedical image segmentation.* MICCAI, pages 234241. Springer

[8]Wang, X. & Gupta,A.(2016) *Generative image modeling using style and structure adversarial networks* ECCV

[9]Mathieu,M., Couprie, C.,& LeCun, Y. (2016) *Deep multi-scale video prediction beyond mean square error* ICLR

CSE 253 Evaluate and Improve the Stochasticity of a cGAN Model for Image-to-Image Transformation

Sainan Liu
A13291871
sal131@eng.ucsd.edu

Hao-en Sung
A53204772
wrangle1005@gmail.com

Shiwei Song
A53206591
shs163@eng.ucsd.edu

Haifeng Huang
A53208823
hah086@eng.ucsd.edu

1 Main Idea

The main idea of this project is to explore techniques that can improve stochasticity in the output of the conditional generative adversarial networks(cGAN) based on top of the model used in a Image-to-Image Translation paper[1].

Steps include:

- Step 1: Replicate the code from lua to keras.
- Step 2: Reproduce the result for different loss function comparisons in Table 1 of [1] shown in Table 10, Figure 4 in [1] shown in Fig. 11 and Figure 7 of [1] shown in Fig. 12. Use the result of this step as the baseline for step 3.
- Step 3: Improve stochasticity with various method (add temperature, weighted neurons or another generator) to the existing model.
- Step 4(optional): If time permits, we will continue to reproduce the result for using different patch sizes in Table 2 from [1] shown in 13, Figure 6 from [1] shown in Fig. 14.

2 Data Source

To compare the result, we will use the same set of data that is provided by the authors in [2].

Table 1: Existing data source.

# Dataset Name	Task	Training No.	Validation No.	Test No.
Cityscapes[3]	Semantic labels ↔ photo	2975	500	-
CMP Facades[4]	Architectural labels ↔ photo	400	100	106
Google Maps	Map ↔ aerial photo	1096	1098	-
Edges to Shoes[5]	Edges ↔ photo	49825	200	-
Edges to Handbags[6]	Edges ↔ photo	138567	200	-

Since it took us 3-4 hours to run 400 images on a 5G GPU, we will start with less images then gradually increase our dataset size. The main dataset we will be using for item 2 and 3 of our main idea would be based on Cityscapes dataset. Since it is the main dataset the paper used and can easily be used to calculated FCN-score using the structure from Fully Convolutional Networks for Semantic Segmentation in [10]. Some images are shown as follows.

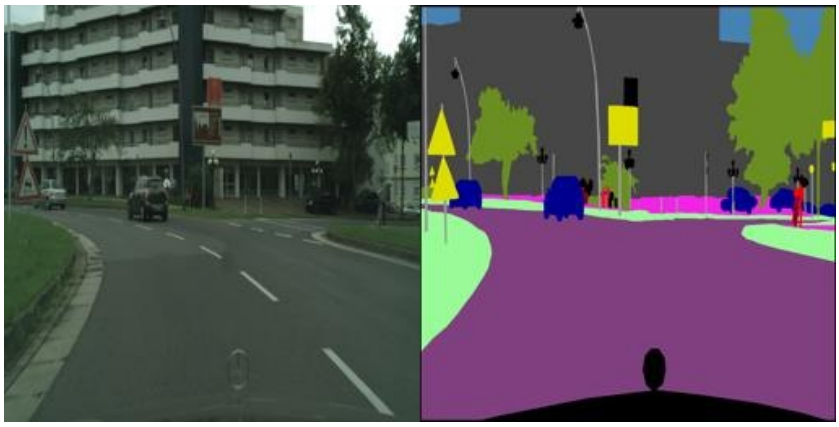


Figure 1: Sample image from Cityscapes

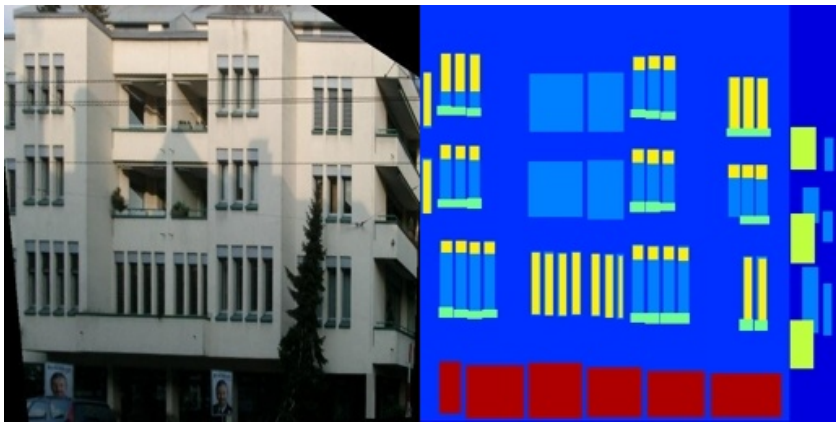


Figure 2: Sample image from CMP Facades

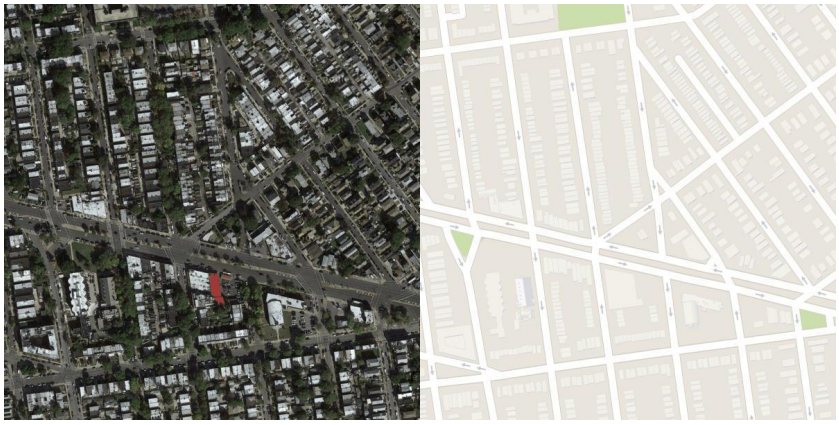


Figure 3: Sample image from Google Maps

Figure 4: Sample image from Edges to Shoes

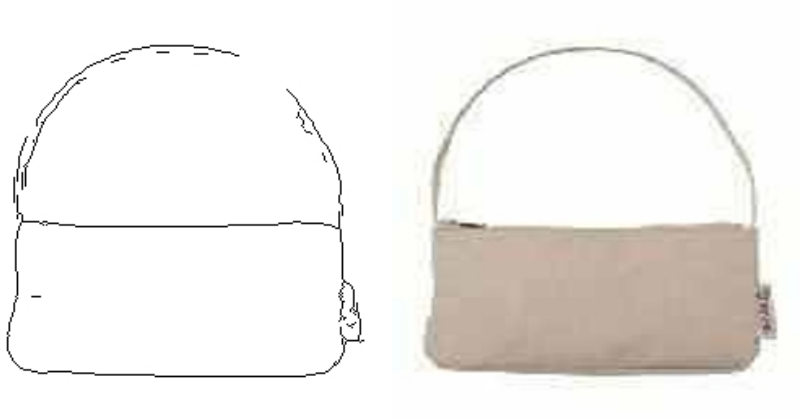


Figure 5: Sample image from Edges to Handbags

3 Network Architecture

3.1 Introduction to cGANs

The conditional Generative Adversarial Network consists two neural networks: a generator **G** and a discriminator **D**. Given a sample pair (X, T).

The generator will take a sample X and a random noise Z as input and generates output Y as close as T.

$$Y = G(X, Z)$$

The discriminator will take a pair (X, Y) and try to figure out whether Y is a real target or a fake target generated by the generator.

$$D(X, Y) = \begin{cases} 1 & D \text{ regards } Y \text{ as a real target} \\ 0 & D \text{ regards } Y \text{ as a fake target} \end{cases}$$

The loss function we will use to train the two networks are

$$L_D = \sum -t \log(D(x, y)) - (1 - t) \log(1 - D(x, G(x, z)))$$

$$L_G = \sum -log(D(x, G(x, z)))$$

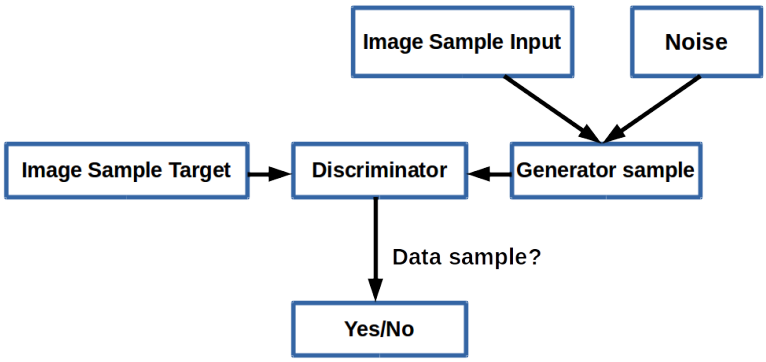


Figure 6: Generalized cGAN model

3.2 Structure Visualization

The two networks will all have a overall structure of convolution-BatchNorm-ReLu.

For the generator, it have a tanh activation function in the output layer. We will use the U-Net model described in [7]. So layer $n - i$ will have a connection with layer i.

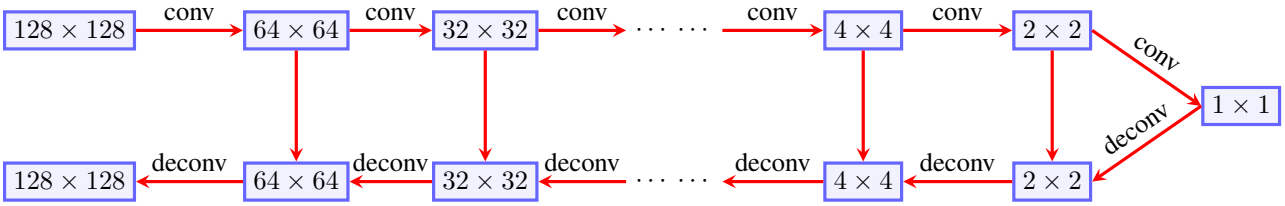


Figure 7: Structure of the generator network

For the discriminator, it will have a sigmoid activation function in the output layer. So the output would be the probability of Y as a real target.

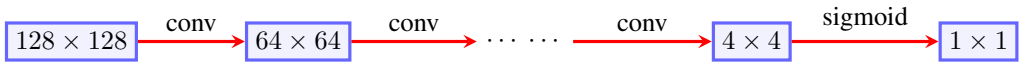


Figure 8: Structure of the discriminator network

4 Code Source and Progress

The code written in lua is available publicly at https://github.com/phillipi/pix2pix. We have duplicated the main generator and descriminator in keras so far, which has been attached to the end of this report.

The visualized graphs for our keras model code are shown as follows:

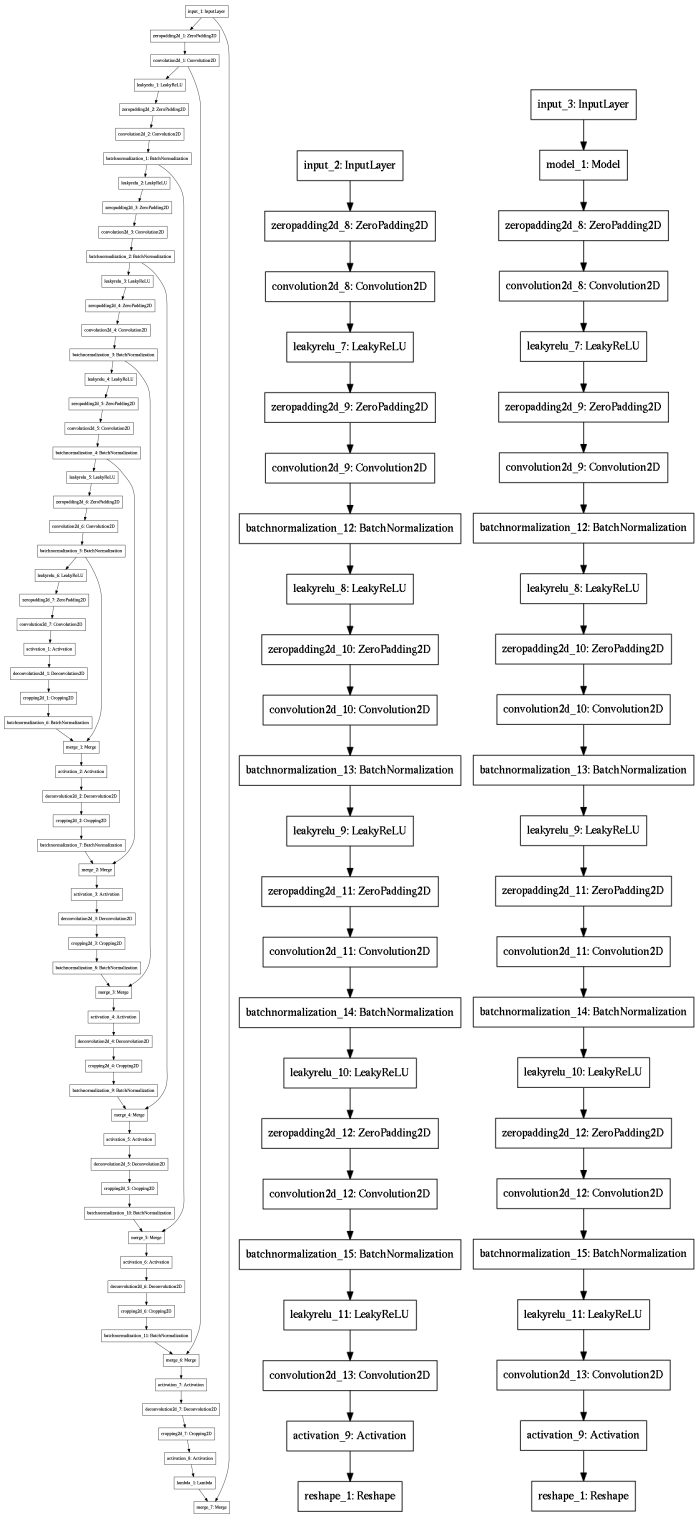


Figure 9: keras model for generator(left) - model 1, descriminator(middle) - model 2, cGAN(right) - model 3

5 Goals and Evaluation Methods

5.1 Replicate the Results

5.1.1 Replication Target

We will first try to replicate part of the results from [1] in both lua and our own keras code. The experiments from original papers are snapshot as follows, and we are expected to reproduce similar results as theirs in [1].

- Replicate results for different objective functions on Cityscapes in terms of FCN-scores, as shown in Fig. 10, images shown in Fig. 11, and color distributions with lab colors in Fig. 12.

Loss	Per-pixel acc.	Per-class acc.	Class IOU
L1	0.44	0.14	0.10
GAN	0.22	0.05	0.01
cGAN	0.61	**0.21**	**0.16**
L1+GAN	**0.64**	0.19	0.15
L1+cGAN	0.63	**0.21**	**0.16**
Ground truth	0.80	0.26	0.21

Figure 10: FCN-scores for the result of different objective functions. Taken from Table 1 from [1].

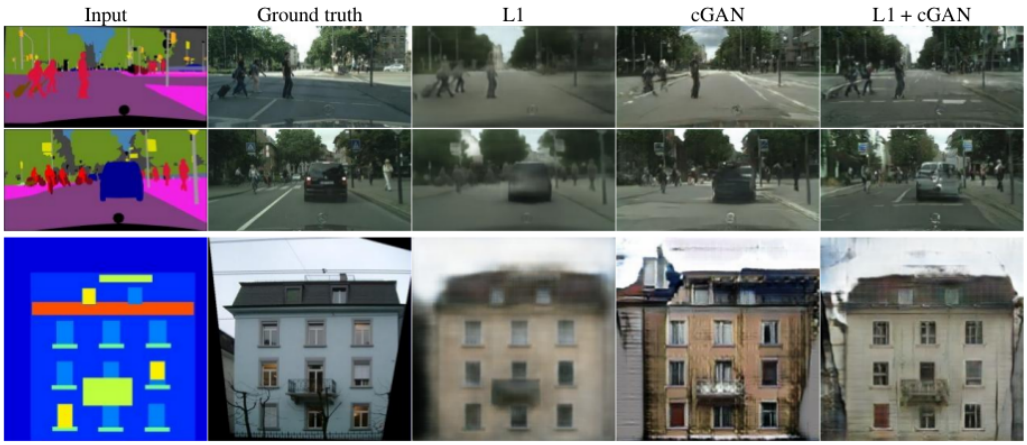
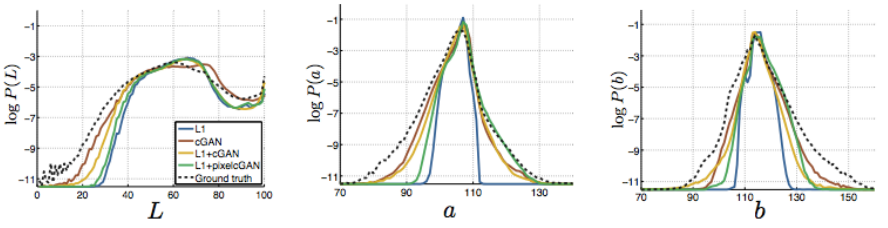


Figure 11: Image results for different objective functions. Taken from Figure 4 from [1].



Loss	Histogram intersection against ground truth L	a	b
L1	0.81	0.69	0.70
cGAN	**0.87**	0.74	**0.84**
L1+cGAN	0.86	**0.84**	0.82
PixelGAN	0.83	0.68	0.78

(d)

Figure 12: Color distributions from results of different objective Functions. Taken from Figure 7 from [1].

- If time permits, we will also attempt to replicate results from using different patch sizes in the descriminator using the Cityscapes dataset in terms of FCN-scores, as shown in Fig. 13 and images in Fig. 14.

Discriminator receptive field	Per-pixel acc.	Per-class acc.	Class IOU
1×1	0.44	0.14	0.10
16×16	0.62	0.20	**0.16**
70×70	**0.63**	**0.21**	**0.16**
256×256	0.47	0.18	0.13

Figure 13: FCN-score for different receptive field sizes of the discriminator evaluated on Cityscapes labels → photos dataset. Taken from Table 2 from [1].

Figure 14: Image results for different patch size variations. Taken from Figure 6 from [1].

5.1.2 Evaluation method

FCN-score is proposed in Fully Convolutional Networks for Semantic Segmentation [10], whose implementation can be found in github; while lab color space in a certain color space consists of L, a, and b. More details of lab color space can be found in wiki.

5.2 Improve stochasticity

5.2.1 Inspiration

In [1], they mentioned that in order to capture the full entropy of the conditional distributions that the cGAN models, we need to design a cGAN that produces stochastic output. [1] mentioned that, in previous research, [8] has tried to add Gaussian noise as an input to the generator in addition to x, but both [1] and [9] found that the generator would simply learn to ignore the noise. Therefore, [1] proposed to provide noise only in the form of dropout on several layers of the generator at both training and test time. However, they observe very minor stochasticity in the output of their nets. Hence we decided to build on top of their network, and further explore ways to improve the stochasticity of the output for cGAN.

5.3 Potential changes

5.3.1 Add temperature

We will try to add a layer of softmax with a temperature parameter to add stochasticity to the network outputs

5.3.2 Add weights to neurons

We can also try to randomly assign weights to neurons in order to increase stochasticity of the output.

5.3.3 Add another generator

We can add another generator that generates the sample from pure noise, and feed the result also into the discriminator, where the discriminator will try to differentiate between real data and conditional generator sample, as well as conditional generator sample and noise generator sample. As the three

networks (two generator networks + one discriminator network) competes with each other, we hope that this will allow the conditional generator sample to gain more stochasticity in its output.

5.4 Measurement

We found no literature mentioning how to evaluate how stochastic a GAN image output is. Therefore, we will compare the GAN and the final cGAN model from [1] and try to come up with a set of measurement of our own. Our current thought is that we will repeatedly produce the generated image from the same image source from GAN, cGAN and our cGAN models. Then use a clustering method to evaluate how closely they are related to each other and how closely they are related within each group. For example, we may use t-SNE [11] as a clustering method to visualize the distribution of these generated images.

6 Results

6.1 Replicate the results

We have ran a small experiment with the lua data we were given, and got some baseline images from the facades dataset. With 400 images, our training took 3.5 hours. Here are our training result from 5000(start) and 80000(end) iterations in Fig 15.

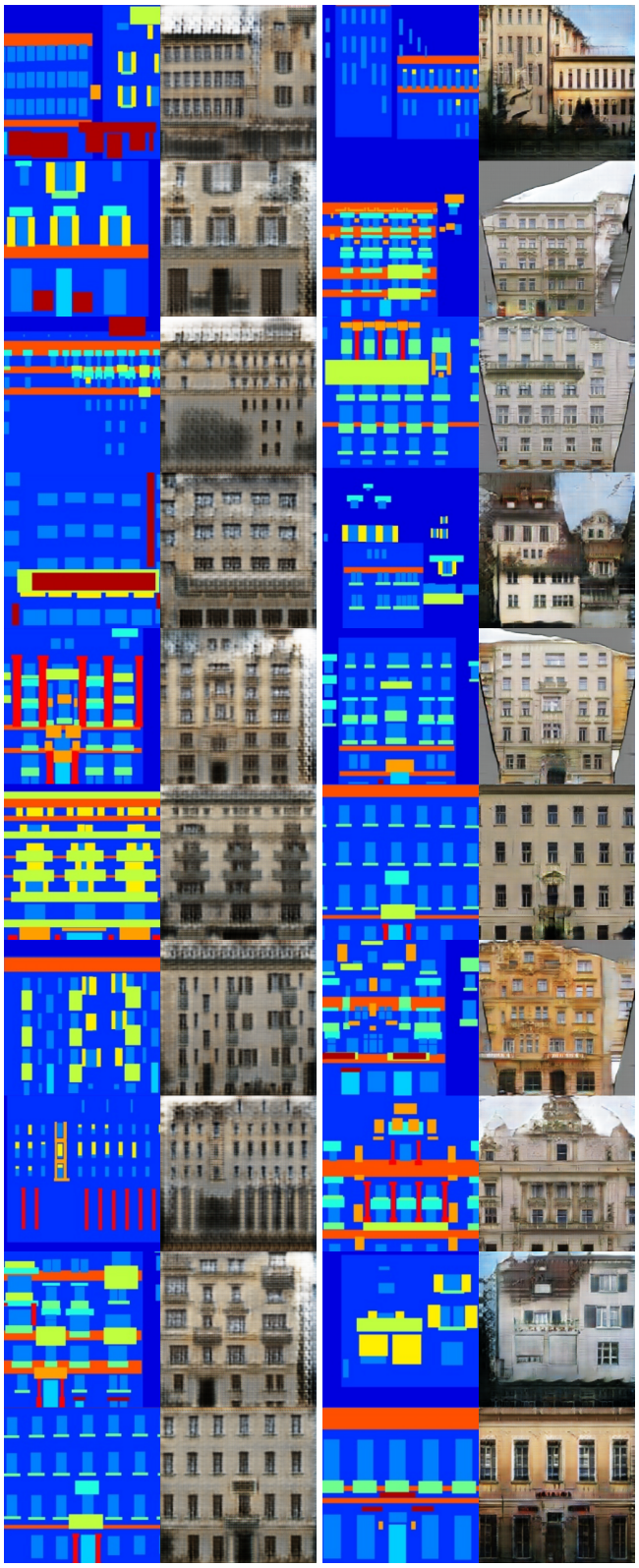


Figure 15: Training result images from 400 facades examples at 5000 iterations and 8000 iterations.

References

[1] Isola, P., Zhu, J., Zhou, T. & Efros, A.A. (2016) *Image-to-Image Translation with Conditional Adversarial Networks*,https://arxiv.org/pdf/1611.07004v1.pdf

[2] Isola, P. Zhu, J., Zhou, T. & Efros, A.A. (2016) *pix2pix datasets*, http://people.eecs.berkeley.edu/ tinghuiz/projects/pix2pix/datasets/

[3] Cordts,M., Omran,M., Ramos, S.,Rehfeld, T.,Enzweiler,M., Benenson,R. ,Franke, U., Roth, S. & Schiele, B. (2016). *The cityscapes dataset for semantic urban scene understanding.* CVPR

[4] Gatys,L. A., Ecker, A. S., & Bethge, M. (2016) *Image style transfer using convolutional neural networks.* CVPR

[5] Yu,A. & Grauman, K.(2014) *Fine-Grained Visual Comparisons with Local Learning* CVPR

[6]Zhu,J., Krhenbhl,P. , Shechtman, E. & Efros, A. A. (2016) *Generative visual manipulation on the natural image manifold* ECCV

[7]Ronneberger,O., Fischer, P. & Brox, T. (2015)*U-net: Convolutional networks for biomedical image segmentation.* MICCAI, pages 234241. Springer

[8]Wang, X. & Gupta,A.(2016) *Generative image modeling using style and structure adversarial networks* ECCV

[9]Mathieu,M., Couprie, C.,& LeCun, Y. (2016) *Deep multi-scale video prediction beyond mean square error* ICLR

[10]Long, J, Shelhamer, E. & Darrell, T. (2015) *Fully convolutional networks for semantic segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*CVPR, pages 3431 3440.

[11]Laurens, M. & Hinton, G. (2008) *Visualizing Data using t-SNE*

Codes

Codes developed so far

Listing 1: model.py for generator descriminator and combined GAN model

```
from keras.models import Model, Sequential
from keras.layers import Activation, Input, Reshape, merge, Lambda
from keras.layers.convolutional import Convolution2D, Deconvolution2D, ZeroPadding2D, Cropping
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.normalization import BatchNormalization
from keras.utils.visualize_util import plot
import numpy as np


def defineGAN(input_nc=3, nf=32, alpha=0.2):
# ----------------Generator Model----------------------
    g_inputs = Input(shape=(128, 128, input_nc))
    # 128 * 128 * nc
    e1 = ZeroPadding2D(padding=(1,1))(g_inputs)
    e1 = Convolution2D(nf, 4, 4, subsample=(2, 2))(e1)
    # 64 * 64 * nf
    e2 = LeakyReLU(alpha)(e1)
    e2 = ZeroPadding2D(padding=(1, 1))(e2)
    e2 = Convolution2D(nf * 2, 4, 4, subsample=(2, 2))(e2)
    e2 = BatchNormalization()(e2)
    # 32 * 32 * (nf * 2)
    e3 = LeakyReLU(alpha)(e2)
    e3 = ZeroPadding2D(padding=(1, 1))(e3)
    e3 = Convolution2D(nf * 4, 4, 4, subsample=(2, 2))(e3)
    e3 = BatchNormalization()(e3)
    # 16 * 16 * (nf * 4)
    e4 = LeakyReLU(alpha)(e3)
    e4 = ZeroPadding2D(padding=(1, 1))(e4)
    e4 = Convolution2D(nf * 8, 4, 4, subsample=(2, 2))(e4)
    e4 = BatchNormalization()(e4)
    # 8 * 8 * (nf * 8)
    e5 = LeakyReLU(alpha)(e4)
    e5 = ZeroPadding2D(padding=(1, 1))(e5)
    e5 = Convolution2D(nf * 8, 4, 4, subsample=(2, 2))(e5)
    e5 = BatchNormalization()(e5)
    # 4 * 4 * (nf * 8)
    e6 = LeakyReLU(alpha)(e5)
    e6 = ZeroPadding2D(padding=(1, 1))(e6)
    e6 = Convolution2D(nf * 8, 4, 4, subsample=(2, 2))(e6)
    e6 = BatchNormalization()(e6)
    # 2 * 2 * (nf * 8)
    e7 = LeakyReLU(alpha)(e6)
    e7 = ZeroPadding2D(padding=(1, 1))(e7)
    e7 = Convolution2D(nf * 8, 4, 4, subsample=(2, 2))(e7)
    # 1 * 1 * (nf * 8)
    g1 = Activation('relu')(e7)
    g1 = Deconvolution2D(nf * 8, 4, 4, output_shape=(None, 4, 4, nf * 8), subsample=(2, 2))(g1
    g1 = Cropping2D(cropping=((1, 1), (1, 1)))(g1)
    g1 = BatchNormalization()(g1)
    # 2 * 2 * (nf * 8)
    g1 = merge([g1, e6], mode='concat', concat_axis=3)
    # 2 * 2 * (nf * 8 * 2)
    g2 = Activation('relu')(g1)
    g2 = Deconvolution2D(nf * 8, 4, 4, output_shape=(None, 6, 6, nf * 8), subsample=(2, 2))(g2
    g2 = Cropping2D(cropping=((1, 1), (1, 1)))(g2)
    g2 = BatchNormalization()(g2)
    # 4 * 4 * (nf * 8)
    g2 = merge([g2, e5], mode='concat', concat_axis=3)
    # 4 * 4 * (nf * 8 * 2)
```

```
60      g3 = Activation('relu')(g2)
61      g3 = Deconvolution2D(nf * 8, 4, 4, output_shape=(None, 10, 10, nf * 8), subsample=(2, 2))(
62      g3 = Cropping2D(cropping=((1, 1), (1, 1)))(g3)
63      g3 = BatchNormalization()(g3)
64      # 8 * 8 * (nf * 8)
65      g3 = merge([g3, e4], mode='concat', concat_axis=3)
66      # 8 * 8 * (nf * 8 * 2)
67      g4 = Activation('relu')(g3)
68      g4 = Deconvolution2D(nf * 4, 4, 4, output_shape=(None, 18, 18, nf * 4), subsample=(2, 2))(
69      g4 = Cropping2D(cropping=((1, 1), (1, 1)))(g4)
70      g4 = BatchNormalization()(g4)
71      # 16 * 16 * (nf * 4)
72      g4 = merge([g4, e3], mode='concat', concat_axis=3)
73      # 16 * 16 * (nf * 4 * 2)
74      g5 = Activation('relu')(g4)
75      g5 = Deconvolution2D(nf * 2, 4, 4, output_shape=(None, 34, 34, nf * 2), subsample=(2, 2))(
76      g5 = Cropping2D(cropping=((1, 1), (1, 1)))(g5)
77      g5 = BatchNormalization()(g5)
78      # 32 * 32 * (nf * 2)
79      g5 = merge([g5, e2], mode='concat', concat_axis=3)
80      # 32 * 32 * (nf * 2 * 2)
81      g6 = Activation('relu')(g5)
82      g6 = Deconvolution2D(nf, 4, 4, output_shape=(None, 66, 66, nf), subsample=(2, 2))(g6)
83      g6 = Cropping2D(cropping=((1, 1), (1, 1)))(g6)
84      g6 = BatchNormalization()(g6)
85      # 64 * 64 * (nf)
86      g6 = merge([g6, e1], mode='concat', concat_axis=3)
87      # 64 * 64 * (nf * 2)
88      g7 = Activation('relu')(g6)
89      g7 = Deconvolution2D(input_nc, 4, 4, output_shape=(None, 130, 130, input_nc), subsample=(2
90      g7 = Cropping2D(cropping=((1, 1), (1, 1)))(g7)
91      # 128 * 128 * input_nc
92      g7 = Activation('tanh')(g7)
93      def mapTo255(x):
94          return (x + 1) * 127.5
95      g7 = Lambda(mapTo255, output_shape=(128, 128, input_nc))(g7)
96
97      g_outputs = merge([g_inputs, g7], mode='concat', concat_axis=3)
98      G = Model(input=g_inputs, output=g_outputs)
99
100 # ----------Discriminator Model----------
101     d_inputs = Input(shape=(128, 128, input_nc * 2))
102     # 128 * 128 * (input_nc * 2)
103     d_l11 = ZeroPadding2D(padding=(1, 1))
104     d1 = d_l11(d_inputs)
105     d_l12 = Convolution2D(nf, 4, 4, subsample=(2, 2))
106     d1 = d_l12(d1)
107     d_l13 = LeakyReLU(alpha)
108     d1 = d_l13(d1)
109     # 64 * 64 * nf
110     d_l21 = ZeroPadding2D(padding=(1, 1))
111     d2 = d_l21(d1)
112     d_l22 = Convolution2D(nf * 2, 4, 4, subsample=(2, 2))
113     d2 = d_l22(d2)
114     d_l23 = BatchNormalization()
115     d2 = d_l23(d2)
116     d_l24 = LeakyReLU(alpha)
117     d2 = d_l24(d2)
118     # 32 * 32 * (nf * 2)
119     d_l31 = ZeroPadding2D(padding=(1, 1))
120     d3 = d_l31(d2)
121     d_l32 = Convolution2D(nf * 4, 4, 4, subsample=(2, 2))
122     d3 = d_l32(d3)
123     d_l33 = BatchNormalization()
124     d3 = d_l33(d3)
```

```
125         d_l34 = LeakyReLU(alpha)
126         d3 = d_l34(d3)
127         # 16 * 16 * (nf * 4)
128         d_l41 = ZeroPadding2D(padding=(1, 1))
129         d4 = d_l41(d3)
130         d_l42 = Convolution2D(nf * 8, 4, 4, subsample=(2, 2))
131         d4 = d_l42(d4)
132         d_l43 = BatchNormalization()
133         d4 = d_l43(d4)
134         d_l44 = LeakyReLU(alpha)
135         d4 = d_l44(d4)
136         # 8 * 8 * (nf * 8)
137         d_l51 = ZeroPadding2D(padding=(1, 1))
138         d5 = d_l51(d4)
139         d_l52 = Convolution2D(nf * 8, 4, 4, subsample=(2, 2))
140         d5 = d_l52(d5)
141         d_l53 = BatchNormalization()
142         d5 = d_l53(d5)
143         d_l54 = LeakyReLU(alpha)
144         d5 = d_l54(d5)
145         # 4 * 4 * (nf * 8)
146         d_l61 = Convolution2D(1, 4, 4)
147         d6 = d_l61(d5)
148         d_l62 = Activation('sigmoid')
149         d6 = d_l62(d6)
150         d_l63 = Reshape((1,))
151         d_outputs = d_l63(d6)
152
153         D = Model(input=d_inputs, output=d_outputs)
154
155     # ----------GAN Model----------
156         gan_inputs = Input(shape=(128, 128, input_nc))
157         gan_hidden = G(gan_inputs)
158         gan_hidden = d_l11(gan_hidden)
159         gan_hidden = d_l12(gan_hidden)
160         gan_hidden = d_l13(gan_hidden)
161         gan_hidden = d_l21(gan_hidden)
162         gan_hidden = d_l22(gan_hidden)
163         gan_hidden = d_l23(gan_hidden)
164         gan_hidden = d_l24(gan_hidden)
165         gan_hidden = d_l31(gan_hidden)
166         gan_hidden = d_l32(gan_hidden)
167         gan_hidden = d_l33(gan_hidden)
168         gan_hidden = d_l34(gan_hidden)
169         gan_hidden = d_l41(gan_hidden)
170         gan_hidden = d_l42(gan_hidden)
171         gan_hidden = d_l43(gan_hidden)
172         gan_hidden = d_l44(gan_hidden)
173         gan_hidden = d_l51(gan_hidden)
174         gan_hidden = d_l52(gan_hidden)
175         gan_hidden = d_l53(gan_hidden)
176         gan_hidden = d_l54(gan_hidden)
177         gan_hidden = d_l61(gan_hidden)
178         gan_hidden = d_l62(gan_hidden)
179         gan_outputs = d_l63(gan_hidden)
180
181         GAN = Model(input=gan_inputs, output=gan_outputs)
182
183         return G, D, GAN
184
185     if __name__ == "__main__":
186             G, D, GAN = defineGAN()
187             plot(G, to_file='model_G.png')
188             plot(D, to_file='model_D.png')
189             plot(GAN, to_file='model_GAN.png')
```

Listing 2: preprocess.py for data preprocessing

```
from PIL import Image
import numpy as np

SIZE = 128
N_SAMPLE = 1024

def readData():
    print("Reading Data...")
    x_train = np.empty([N_SAMPLE, SIZE, SIZE, 3])
    y_train = np.empty([N_SAMPLE, SIZE, SIZE, 3])

    for i in range(1, N_SAMPLE+1):
        s = "maps/train/" + str(i) + ".jpg"
        img = Image.open(s)
        limg = img.crop((0, 0, 600, 600)).resize((SIZE, SIZE))
        rimg = img.crop((600, 0, 1200, 600)).resize((SIZE, SIZE))

        x_train[i-1] = np.array(rimg.getdata()).reshape(SIZE, SIZE, 3)
        y_train[i-1] = np.array(limg.getdata()).reshape(SIZE, SIZE, 3)

    x_test = np.empty([N_SAMPLE, SIZE, SIZE, 3])
    y_test = np.empty([N_SAMPLE, SIZE, SIZE, 3])

    for i in range(1, N_SAMPLE+1):
        s = "maps/val/" + str(i) + ".jpg"
        img = Image.open(s)
        limg = img.crop((0, 0, 600, 600)).resize((SIZE, SIZE))
        rimg = img.crop((600, 0, 1200, 600)).resize((SIZE, SIZE))

        x_test[i-1] = np.array(rimg.getdata()).reshape(SIZE, SIZE, 3)
        y_test[i-1] = np.array(limg.getdata()).reshape(SIZE, SIZE, 3)

    print("Finish")
    print("Training samples: %d\nTesting samples: %d"%(len(x_train), len(x_test)))

    return x_train, y_train, x_test, y_test


def showImage(x, y, gx, title):
    img = Image.new("RGB", (SIZE*3, SIZE))
    im = np.append(x, y, axis=1)
    im = np.append(im, gx, axis=1)
    im = list(im.reshape(SIZE * SIZE * 3, 3))
    im = [map(int, z) for z in im]
    im = map(tuple, im)
    img.putdata(im)
    img.show(title)

def saveImage(x, y, gx, title):
    img = Image.new("RGB", (SIZE*3, SIZE))
    im = np.append(x, y, axis=1)
    im = np.append(im, gx, axis=1)
    im = list(im.reshape(SIZE * SIZE * 3, 3))
    im = [map(int, z) for z in im]
    im = map(tuple, im)
    img.putdata(im)
    img.save('result/pic' + str(title))
```

Listing 3: train.py for training sample code

```
from tqdm import *
from model import defineGAN
from preprocess import readData, showImage
from keras.models import Model
```

```
5  from keras.optimizers import Adam
6  import numpy as np
7  from PIL import Image
8  import math
9
10 EPOCH = 50
11 BATCH_SIZE = 32
12
13 x_train, y_train, x_test, y_test = readData()
14
15 G, D, GAN = defineGAN()
16
17 G_optim = Adam(lr=0.0002)
18 G.compile(optimizer=G_optim, loss='binary_crossentropy')
19 G.summary()
20
21 D_optim = Adam(lr=0.0002)
22 D.compile(optimizer=D_optim, loss='binary_crossentropy')
23 D.summary()
24
25 GAN.compile(optimizer=G_optim, loss='binary_crossentropy')
26
27
28 real_label = np.array([1] * BATCH_SIZE).reshape(BATCH_SIZE, 1)
29 fake_label = np.array([0] * BATCH_SIZE).reshape(BATCH_SIZE, 1)
30
31 x_valid = x_test[:BATCH_SIZE]
32 y_valid = y_test[:BATCH_SIZE]
33
34 for epo in range(EPOCH):
35     print("Epoch %d/%d"%(epo+1, EPOCH))
36     train_size = len(x_train)
37     valid_size = len(x_valid)
38     n_batch = train_size / BATCH_SIZE
39
40     # train on batch
41     for i in tqdm(range(n_batch)):
42         x_batch = x_train[i*BATCH_SIZE:(i+1)*BATCH_SIZE]
43         y_batch = y_train[i*BATCH_SIZE:(i+1)*BATCH_SIZE]
44
45         fake = G.predict_on_batch(x_batch)
46         real = np.append(x_batch, y_batch, axis=3)
47
48         D.trainable = True
49         D.train_on_batch(real, real_label)
50         D.train_on_batch(fake, fake_label)
51
52         D.trainable = False
53         GAN.train_on_batch(x_batch, real_label)
54
55     # test on validation set
56     fake = G.predict_on_batch(x_valid)
57     real = np.append(x_valid, y_valid, axis=3)
58
59     errD_real = D.test_on_batch(real, real_label)
60     errD_fake = D.test_on_batch(fake, fake_label)
61     errG = GAN.test_on_batch(x_valid, real_label)
62     errD = (errD_fake + errD_real) / 2
63     print("On validation set, errD = %f, errG = %f"%(errD, errG))
64     for i in range(1):
65         showImage(x_valid[i], y_valid[i], fake[i,:,:,3:], str(epo+1))
66
67 G.save_weights("result/G_weight.h5")
68 D.save_weights("result/D_weight.h5")
```

CSE 253 Evaluate and Improve the Stochasticity of a cGAN Model for Image-to-Image Transformation

Sainan Liu
A13291871
sal131@eng.ucsd.edu

Hao-en Sung
A53204772
wrangle1005@gmail.com

Shiwei Song
A53206591
shs163@eng.ucsd.edu

Haifeng Huang
A53208823
hah086@eng.ucsd.edu

1 Introduction

The Image-to-Image Translation paper[1] proposed an interesting cGAN(conditional adversarial networks) as a general-purpose solution to image-to-image translation problems. It is interesting to us because it was able to use a generalized neural network to tackle many types of problems in image processing and computer vision, such as translate GPS maps to google maps, turn black and white image to a colorful image, and even turn segmented image to a photo-realistic image, etc. These problems are difficult because they normally require different loss functions, and the translation is not one-to-one as is mentioned in [1]. Hence it is very important to learn why this network works, and how can we improve on its performance.

By trying to replicate their model in keras, we can learn about how the network works, and what we can do to improve the results. The main achievement the paper has mentioned is that the network automatically learns a loss function for their generative part of the model, which allows the model to be versatile enough to all type of image-to-image translations. Therefore, the images (Figure 2), FCN-scores (Figure 1) and color distributions (Figure 3) of the ouput we have tried to replicate is based on the 5 types of general loss functions they have compared in the paper: generator + L1 loss alone, cGAN, GAN(Generative adversarial networks)[8], GAN + L1 and cGAN + L1. The paper mentioned that the best result can be obtained using cGAN + L1. The data we will be using is called cityscapes, and the task is to translate segmented cityscapes images to photo-realistic images as shown below in figure 4.

Loss	Per-pixel acc.	Per-class acc.	Class IOU
L1	0.44	0.14	0.10
GAN	0.22	0.05	0.01
cGAN	0.61	**0.21**	**0.16**
L1+GAN	**0.64**	0.19	0.15
L1+cGAN	0.63	**0.21**	**0.16**
Ground truth	0.80	0.26	0.21

Figure 1: FCN-scores for the result of different objective functions. Taken from Table 1 from [1].

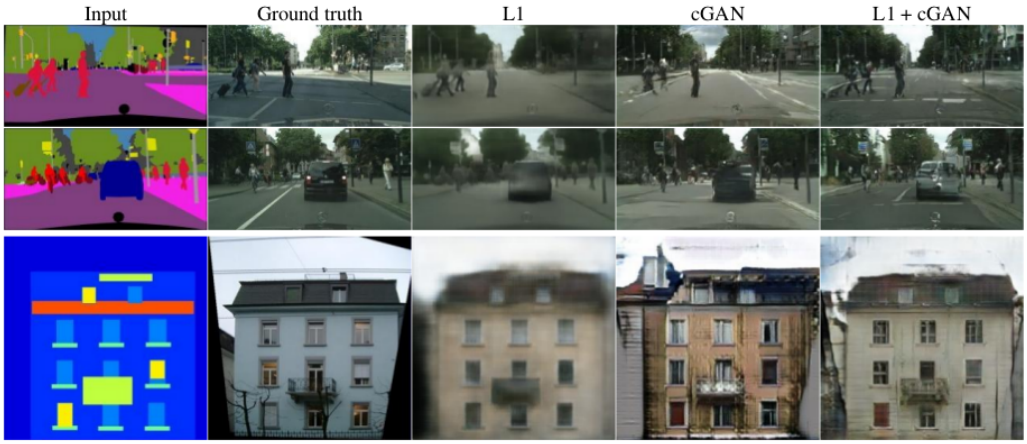
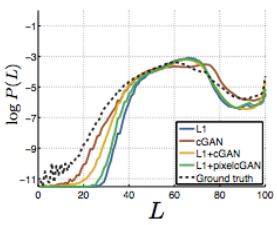
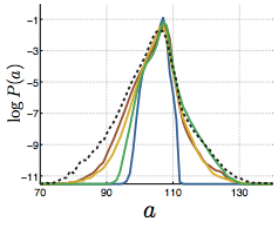


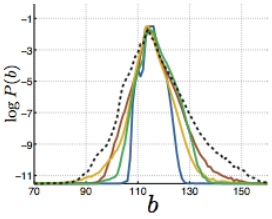
Figure 2: Image results for different objective functions. Taken from Figure 4 from [1].



(a)



(b)



(c)

Loss	Histogram intersection against ground truth L	a	b
L1	0.81	0.69	0.70
cGAN	**0.87**	0.74	**0.84**
L1+cGAN	0.86	**0.84**	0.82
PixelGAN	0.83	0.68	0.78

(d)

Figure 3: Color distributions from results of different objective Functions. Taken from Figure 7 from [1].

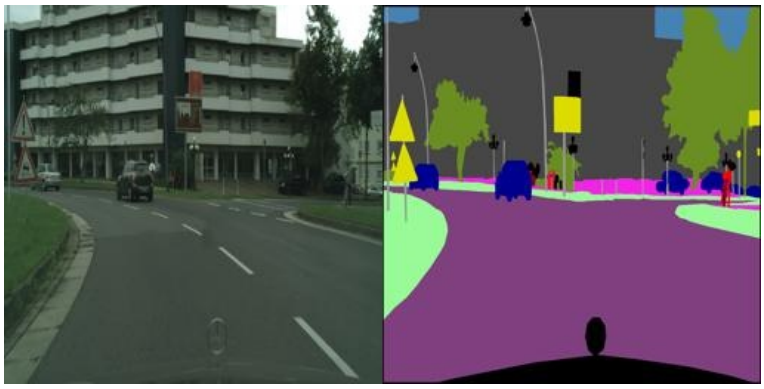


Figure 4: Sample image from Cityscapes. The mapping we are replicating is from right to left.

Additionally, the paper has mentioned in order to capture the full entropy of the conditional distributions that the cGAN models, we need to design a cGAN that produces stochastic output. This allows the same conditional input to produce different image content instead of one deterministic output. This area has not been explored much, so we wanted to come up with some method to visualize this problem better, and potentially improve the stochasticity of the output.

In summary our project took the following steps:

- Step 1: Replicate the code from lua to keras.
- Step 2: Reproduce the result for different loss function comparisons in Table 1 of [1] shown in Table 1, Figure 4 in[1] shown in Fig. 2 and Figure 7 of [1] shown in Fig. 3. Use the result of this step as the baseline for step 3.
- Step 3: Research on how to evaluate stochasticity.
- Step 4: Try to improve stochasticity by reducing dropout values, add random weights in the network layers or add additional network structures.

2 Background

Generative Adversarial Network (GAN) is a kind of system consisting of two neural networks invented by Ian Goodfellow [15]. One is a generator and another is a discriminator. In learning procedures, they will compete against each other and improve their results. GAN are widely used in image generation.

The paper we based our project on is "Image-to-Image Translation Using Conditional Adversarial Networks" [1]. In this paper, they used conditional GAN to fulfill the task of image-to-image translation, e.g. label image to street scenes, BW image to color image and edges to photos. For task image A to image B, generator will take A as input and try to generate an image as close as B. The discriminator will take an image as input and decide whether it's real or generated. They implemented the GAN using lua with torch. They improved the generator with a u-net structure where high layer will have direct connection with low layer. They also improved the discriminator to have different sizes of receptive fields.

[1] mentioned that, in previous research, [8] has tried to add Gaussian noise as an input to the generator in addition to x, but both [1] and [9] found that the generator would simply learn to ignore the noise. Therefore, [1] proposed to provide noise only in the form of dropout on several layers of the generator at both training and test time. However, they observe very minor stochasticity in the output of their nets. Hence we decided to build on top of their network, and further explore ways to improve the stochasticity of the output for cGAN.

3 Model

3.1 Introduction to cGANs

The conditional Generative Adversarial Network consists two neural networks: a generator **G** and a discriminator **D**. In this image-to-image translation problem, given a sample pair (X, T), where we call them input X and target T for example. The generator will take a sample X and a random noise Z as input and try to generate an output Y that is as close to T as possible. This process can be simplified as Euqation 1. The goal is to train a good generator **G** that can do translations such as from segmented images to photos.

$$Y = G(X, Z) \tag{1}$$

The discriminator will take a pair (X, Y) and try to figure out whether Y is a real target or a fake target generated by the generator as is shown in Euqation 2

$$D(X, Y) = \begin{cases} 1 & D \text{ regards } Y \text{ as a real target} \\ 0 & D \text{ regards } Y \text{ as a fake target} \end{cases} \tag{2}$$

The loss function we will use to train the two networks are:

$$L_D = \sum -t \log(D(x, y)) - (1 - t) \log(1 - D(x, G(x, z))) \tag{3}$$

$$L_G = \sum -log(D(x, G(x, z))) \tag{4}$$

We will try to find $G^* = argmin_G max_D L_{cGAN}(G, D)$, where

$$L_{cGAN} = E_{x,y\sim p_{data}(x,y)}[logD(x, y)] + E_{x\sim p_{data}(x),z\sim p_z(z)}[log(1 - D(x, G(x, z)))] \quad (5)$$

The general network that is used in [1] can be simplified as shown in Figure 5

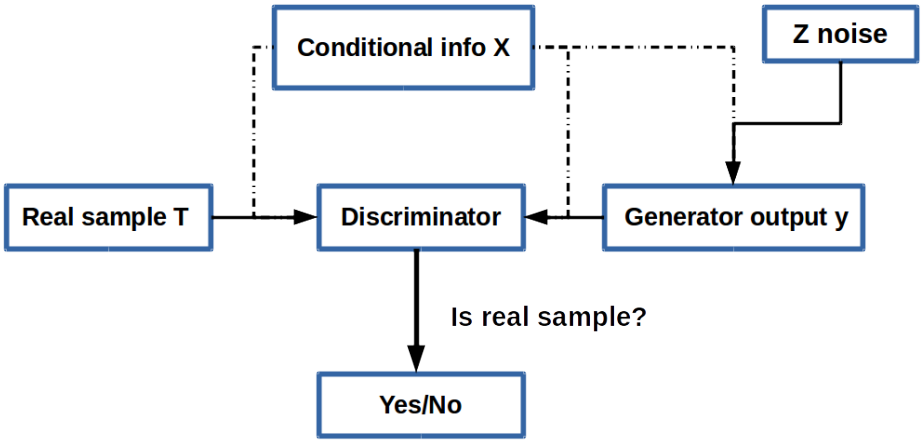


Figure 5: Generalized cGAN model

The loss functions we use for G and D are binary cross entropy, which is consistent with [1]. When adding L1 loss on top of G part of the cGAN model, we use mean absolute error as the loss function.

3.2 Structure Visualization

The two networks both are consist with a general structure: convolution-BatchNorm-ReLu.

For the generator **G**, it have a tanh activation function in the output layer. We will use the U-Net model described in [7]. So layer $n - i$ will have a connection with layer i as shown below. In the network, both of the input and output dimensions are $128 \times 128 \times 3$ scaled images with pixel values scaled to zero-mean between -1 to 1. The detailed model graph is shown in Appendix A and in the code section.

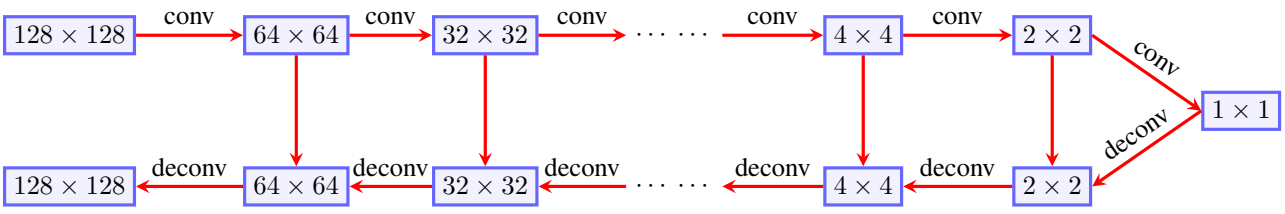


Figure 6: Structure of the generator network

For the discriminator, **D**, it will have a sigmoid activation function in the output layer. So the output would be the probability of Y as a real target. The detailed model graph is shown in Appendix B and in the code section.

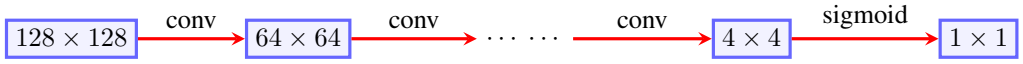


Figure 7: Structure of the discriminator network

The entire cGAN model is shown in Appendix C.

3.3 Data Set

To compare the result, we will use the same set of data that is provided by the authors in [2], which is the cityscapes dataset [3] for most of the result replication and baseline testing. CMP Facades [4] is also used for testing due to its manageable sample size. The original image size for cityscapes are $256 \times 256 \times 3$, we scaled it to $128 \times 128 \times 3$ and preprocessed the pixel values to zero mean between -1 and 1 as mentioned previously.

An image sample from the cityscapes dataset has been shown in 4. The image on the left is the photo of the city that we are trying to generate, the segmented image on the right will be our input information. An image sample from facades can be seen here in 8

Table 1: Existing data source.

# Dataset Name	Task	Training No.	Validation No.	Test No.
Cityscapes[3]	Semantic labels ↔ photo	2975	500	-
CMP Facades[4]	Architectural labels ↔ photo	400	100	106
Google Maps	Map ↔ aerial photo	1096	1098	-
Edges to Shoes[5]	Edges ↔ photo	49825	200	-
Edges to Handbags[6]	Edges ↔ photo	138567	200	-

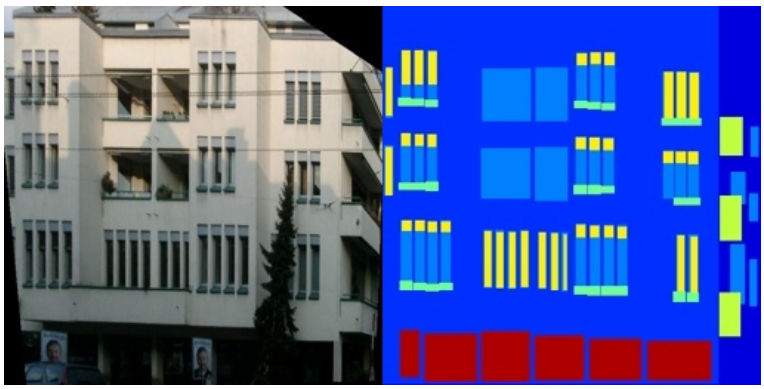


Figure 8: Sample image from Facades. The mapping we are replicating is from right to left here.

3.4 Software

The original code was written in lua [12] with the Torch library [13], which is available publicly at pix2pix github[2]. We duplicated their structure in keras [14] shown in the code section.

4 Experiments & Results

4.1 Replication the cGAN Model in Keras

4.1.1 Experiments

As mentioned previously in the introduction, we will try to replicate the output images (Figure 2), FCN-scores (Figure 1) and color distributions (Figure 3) of [1] to verify our replication code.

The 5 type of loss function we will run on are: generator + L1 loss ('mean_absolute_error') alone, cGAN, GAN(Generative adversarial networks)[8], GAN + L1 and cGAN + L1.

- We first run the original lua code provided by [1] with the 5 loss functions on cityscapes[3] dataset. The images are shown here in Figure 9
- We then run the the same set of loss functions on our own keras code to try to acquire the same set of results to compare with the original results.

4.1.2 Evaluation Methods for Result Replication

FCN-score is proposed in Fully Convolutional Networks for Semantic Segmentation [10], whose implementation can be found in github; while lab color space in a certain color space consists of L, a, and b. More details of lab color space can be found in wiki.

4.1.3 Results

We have first tested their original code for different losses shown in Fig 9.

Figure 9: Test result for different loss functions. Each column shows results trained under a different loss. The models are used were trained over 50 epochs

4.2 Reconstruction through networks

To gain some deeper understanding of the generator, we want to visualize the internal representations through the network. The method we chose is to reconstruct an image that will produce close outputs at the layer.

Say we want to reconstruct the output at layer 5. We selected one sample image and fed it to the trained network and got the output at layer 5. Then we started with a white noise image. We fixed the weights of the networks and ran backpropagation to update the image. Finally we gained the reconstructed image at that layer. Fig 10 and Fig 11 shows some of the reconstructed images through the generator.

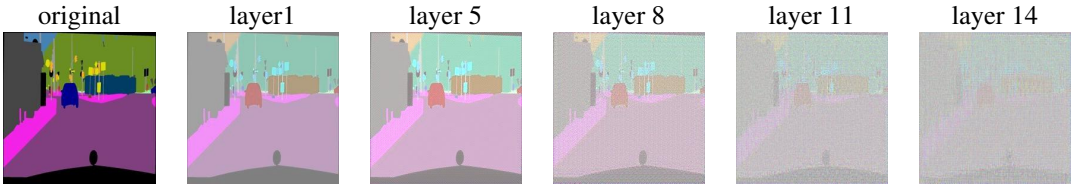


Figure 10: Reconstruction results from different layers for label as input

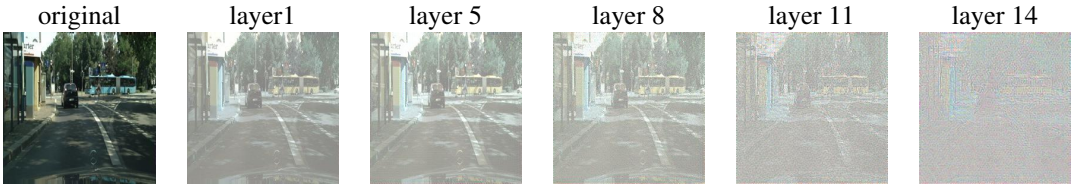


Figure 11: Reconstruction results from different layers for image as ainput

4.3 Improve stochasticity

4.3.1 Experiment

In order the improve the stochasticity, we have come up with the following experiments.

- [1] mentioned that their random noise is introduced in the form of dropout layers. Therefore, in order to increase the stochasticity, we wanted to first reduce the dropout rate further to see if that helps with the performance.
- We want to add random weights to the output neurons.

Stochasticity is hard to measure. We found no literature mentioning how to evaluate how stochastic a GAN image output is so far. Therefore, we compared the results from GAN and the final cGAN model from [1] and tried to come up with a set of measurement of our own. We will repeatedly produce the generated image from the same image source from GAN, cGAN and our cGAN models for 10 times. Then use t-SNE [11] to reduce the output dimension from $128 \times 128 \times 3$ to 2 in order to visualize the distribution of these generated images in 2D space. By looking at the clustering of different models, we can see if the spread of the data has increased or not. If the spread is small, we think this means that the pixel values are closer to each other in the higher dimensions, if the pread is large, and they visually look realistic, then we think this means that we have generated more stochastic results. The other visual representation we used is histogram. We can also draw histogram of all values from all images drawn from the same iamge and see how much overlapping do we get in the histogram figure.

5 Discussion and Conclusions

5.1 Replicate models in keras

[TODO: 8 points
Does the discussion of the results provide insights into them, rather than simply repeating them?
Are all of the results discussed, including successes and failures?
Is the discussion of the import of your results well done?
If the model didn't "work", are potential reasons for its failure clearly discussed?]

5.2 Replicate figures and tables

5.2.1 Images Output

5.2.2 FCN-score

5.2.3 Color Distribution

6 Concept andor Innovation

[TODO: 4 points
Here, you should describe the ”valued added” of your project.
Did you come up with a novel architecture?
Did you apply the architecture to a new dataset?
Did you find a better way to set the metaparameters, or some other innovation?
Did you provide a novel analysis of a previous result?
If all you did was run someone else’s code and replicate what they did, your score here will be 0/4.]

7 Individual Contributions

Hao-en and Haifeng are in charge of FCN-score and Color Distribution measurement. Shiwei and Sainan are in charge of duplicating keras code and test out the stochasticity improvement.

References

[1] Isola, P., Zhu, J., Zhou, T. & Efros, A.A., (2016). *Image-to-Image Translation with Conditional Adversarial Networks*.
https://arxiv.org/pdf/1611.07004v1.pdf
https://github.com/shelhamer/fcn.berkeleyvision.org

[2] Isola, P. Zhu, J., Zhou, T. & Efros, A.A. (2016). *pix2pix datasets*,
http://people.eecs.berkeley.edu/ tinghuiz/projects/pix2pix/datasets/

[3] Cordts,M., Omran,M., Ramos, S.,Rehfeld, T.,Enzweiler,M., Benenson,R. ,Franke, U., Roth, S. & Schiele, B. (2016). *The cityscapes dataset for semantic urban scene understanding.* CVPR

[4] Gatys,L. A., Ecker, A. S., & Bethge, M. (2016) *Image style transfer using convolutional neural networks.* CVPR

[5] Yu,A. & Grauman, K.(2014) *Fine-Grained Visual Comparisons with Local Learning* CVPR

[6] Zhu,J., Krhenbhl,P. , Shechtman, E. & Efros, A. A. (2016) *Generative visual manipulation on the natural image manifold* ECCV

[7] Ronneberger,O., Fischer, P. & Brox, T. (2015)*U-net: Convolutional networks for biomedical image segmentation.* MICCAI, pages 234241. Springer

[8] Wang, X. & Gupta,A.(2016) *Generative image modeling using style and structure adversarial networks* ECCV

[9] Mathieu,M., Couprie, C.,& LeCun, Y. (2016) *Deep multi-scale video prediction beyond mean square error* ICLR

[10] Long, J, Shelhamer, E. & Darrell, T. (2015) *Fully convolutional networks for semantic segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*CVPR, pages 3431 3440.

[11] Laurens, M. & Hinton, G. (2008) *Visualizing Data using t-SNE*

[12] Lua, an embeddable scripting language.
http://www.lua.org/

[13] Torch, a scientific computing framework for LUAJIT
http://torch.ch/

[14] Keras, Deep learning library for Theano and Tensorflow
https://keras.io/

[15] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2014). *Generative Adversarial Networks*

Appendices

A Generator Model Detail

Model graph for Generator:

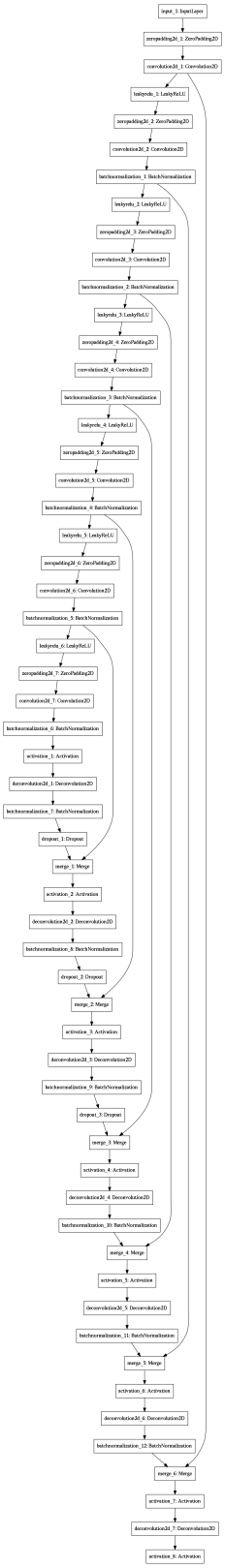


Figure 12: keras model for generator **G** - model1

B Descriminator Detail

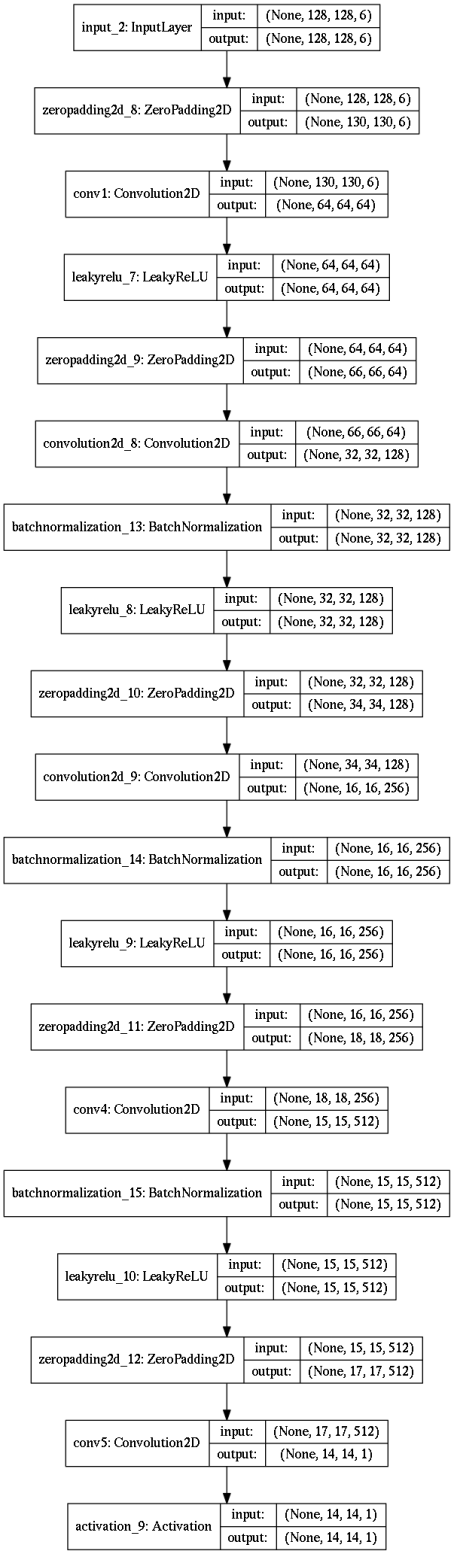


Figure 13: keras model for descriminator **D** - model 2

C cGAN Detail

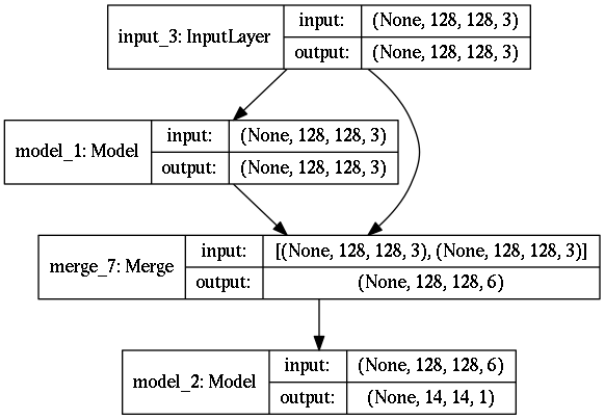


Figure 14: keras model for cGAN

Codes

Codes developed so far

Listing 1: model.py for generator descriminator and combined GAN model

```
from keras.models import Model, Sequential
from keras.layers import Activation, Input, Reshape, merge, Lambda, Dropout, Flatten, Dense
from keras.layers.convolutional import Convolution2D, Deconvolution2D, ZeroPadding2D, Cropping
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.normalization import BatchNormalization
from keras.utils.visualize_util import plot

import keras.backend as K
import numpy as np

def conv(x, nf, norm=True):
    y = LeakyReLU(0.2)(x)
    y = ZeroPadding2D(padding=(1, 1))(y)
    y = Convolution2D(nf, 4, 4, subsample=(2, 2))(y)
    if norm:
        y = BatchNormalization(mode=2)(y)
    return y

def deconv(x, nf, output_size, norm=True):
    y = Activation('relu')(x)
    y = Deconvolution2D(nf, 4, 4, output_shape=(None, output_size+2, output_size+2, nf), subsa
    y = Cropping2D(cropping=((1, 1), (1, 1)))(y)
    if norm:
        y = BatchNormalization(mode=2)(y)
    return y

def generator(input_size=128, nf=64, p=0.5):
    g_inputs = Input(shape=(input_size, input_size, 3))

    # 128 * 128 * 3
    e1 = ZeroPadding2D(padding=(1, 1))(g_inputs)
    e1 = Convolution2D(nf, 4, 4, subsample=(2, 2))(e1)

    # 64 * 64 * nf
    e2 = conv(e1, nf * 2)
    # 32 * 32 * (nf * 2)
    e3 = conv(e2, nf * 4)
    # 16 * 16 * (nf * 4)
    e4 = conv(e3, nf * 8)
    e5 = conv(e4, nf * 8)
    # 8 * 8 * (nf * 8)
    # 4 * 4 * (nf * 8)
    e6 = conv(e5, nf * 8)
    # 2 * 2 * (nf * 8)
    e7 = conv(e6, nf * 8)
    # 1 * 1 * (nf * 8)
    g1 = deconv(e7, nf * 8, output_size=2)
    g1 = Dropout(p)(g1)
    g1 = merge([g1, e6], mode='concat', concat_axis=-1)
    # 2 * 2 * (nf * 8 * 2)
    g2 = deconv(g1, nf * 8, output_size=4)
    g2 = Dropout(p)(g2)
    g2 = merge([g2, e5], mode='concat', concat_axis=-1)
    # 4 * 4 * (nf * 8 * 2)
    g3 = deconv(g2, nf * 8, output_size=8)
    g3 = Dropout(p)(g3)
    g3 = merge([g3, e4], mode='concat', concat_axis=-1)
    # 8 * 8 * (nf * 8 * 2)
    g4 = deconv(g3, nf * 4, output_size=16)
```

```
60      g4 = merge([g4, e3], mode='concat', concat_axis=-1)
61      # 16 * 16 * (nf * 4 * 2)
62      g5 = deconv(g4, nf * 2, output_size=32)
63      g5 = merge([g5, e2], mode='concat', concat_axis=-1)
64      # 32 * 32 * (nf * 2 * 2)
65      g6 = deconv(g5, nf, output_size=64)
66      g6 = merge([g6, e1], mode='concat', concat_axis=-1)
67      # 64 * 64 * (nf * 2)
68      g7 = deconv(g6, 3, output_size=128, norm=False)
69      g_outputs = Activation('tanh')(g7)
70
71      G = Model(input=g_inputs, output=g_outputs)
72      plot(G, to_file='../model/model_G.png', show_shapes=True)
73      return G
74
75  def discriminator(input_size=128, nf=64, n_layers=3):
76      d_inputs = Input(shape=(input_size, input_size, 6))
77      # 128 * 128 * 6
78      d_hidden = ZeroPadding2D(padding=(1, 1))(d_inputs)
79      d_hidden = Convolution2D(nf, 4, 4, subsample=(2, 2))(d_hidden)
80      # 64 * 64 * nf
81      ndf = nf
82      for i in range(n_layers-1):
83          ndf = min(nf * 8, ndf * 2)
84          d_hidden = conv(d_hidden, ndf)
85      ndf = min(nf * 8, ndf * 2)
86      d_hidden = LeakyReLU(0.2)(d_hidden)
87      d_hidden = ZeroPadding2D(padding=(1, 1))(d_hidden)
88      d_hidden = Convolution2D(ndf, 4, 4, subsample=(1, 1))(d_hidden)
89      d_hidden = BatchNormalization(mode=2)(d_hidden)
90      d_hidden = LeakyReLU(0.2)(d_hidden)
91      d_hidden = ZeroPadding2D(padding=(1, 1))(d_hidden)
92      d_hidden = Convolution2D(1, 4, 4, subsample=(1, 1))(d_hidden)
93      d_outputs = Activation('sigmoid')(d_hidden)
94      # d2 = conv(d1, nf * 2)
95      # # 32 * 32 * (nf * 2)
96      # d3 = conv(d2, nf * 4)
97      # # 16 * 16 * (nf * 4)
98      # d4 = conv(d3, nf * 8)
99      # # 8 * 8 * (nf * 8)
100     # d5 = conv(d4, nf * 8)
101     # # 4 * 4 * (nf * 8)
102     # d6 = conv(d5, nf * 8)
103     # # 2 * 2 * (nf * 8)
104     # d7 = conv(d6, nf * 8)
105     # # 1 * 1 * (nf * 8)
106     # d8 = Flatten()(d7)
107     # d8 = Dense(1)(d8)
108     # d_outputs = Activation('sigmoid')(d8)
109
110     D = Model(input=d_inputs, output=d_outputs)
111     plot(D, to_file='../model/model_D.png', show_shapes=True)
112     return D
113
114 def cGAN(G, D, input_size=128):
115     gan_inputs = Input(shape=(input_size, input_size, 3))
116     g_outputs = G(gan_inputs)
117     d_inputs = merge([gan_inputs, g_outputs], mode='concat', concat_axis=-1)
118     d_hidden = D.layers[1](d_inputs)
119     for i in range(2, len(D.layers)):
120         d_hidden = D.layers[i](d_hidden)
121
122     GAN = Model(input=gan_inputs, output=[g_outputs, d_hidden])
123     return GAN
124
```

```
125 if __name__ == "__main__":
126     G = generator()
127     D = discriminator()
128     cGAN = cGAN(G, D)
```

Listing 2: preprocess.py for data preprocessing

```
 1 from PIL import Image
 2 import numpy as np
 3 from os.path import isdir
 4 from os import makedirs
 5
 6 def preprocess(x):
 7     # Scale input 0 to 255 to -1 to 1.
 8     func = np.vectorize(lambda x: x/127.5-1)
 9     return func(x)
10
11 def deprocess(x):
12     # Scale output -1 to 1 back to 0 to 255.
13     func = np.vectorize(lambda x: (x+1)*127.5)
14     return func(x)
15
16 def readData(dataset='../data/cityscapes/', size=128, n_train=1024, n_valid=128):
17     print("Reading Data...")
18     x_train = np.empty([n_train, size, size, 3])
19     y_train = np.empty([n_train, size, size, 3])
20
21     img = Image.open(dataset + 'train/1.jpg')
22     w, h = img.size
23     for i in range(1, n_train+1):
24         s = dataset + "train/" + str(i) + ".jpg"
25         img = Image.open(s)
26         limg = img.crop((0, 0, w/2, h)).resize((size, size))
27         rimg = img.crop((w/2, 0, w, h)).resize((size, size))
28
29         x_train[i-1] = np.array(rimg.getdata()).reshape(size, size, 3)
30         y_train[i-1] = np.array(limg.getdata()).reshape(size, size, 3)
31
32     x_test = np.empty([n_valid, size, size, 3])
33     y_test = np.empty([n_valid, size, size, 3])
34
35     for i in range(1, n_valid+1):
36         s = dataset + "val/" + str(i) + ".jpg"
37         img = Image.open(s)
38         limg = img.crop((0, 0, w/2, h)).resize((size, size))
39         rimg = img.crop((w/2, 0, w, h)).resize((size, size))
40
41         x_test[i-1] = np.array(rimg.getdata()).reshape(size, size, 3)
42         y_test[i-1] = np.array(limg.getdata()).reshape(size, size, 3)
43
44     x_train = preprocess(x_train)
45     y_train = preprocess(y_train)
46     x_test = preprocess(x_test)
47     y_test = preprocess(y_test)
48
49     print("Finish")
50     print("Training samples: %d\nTesting samples: %d"%(len(x_train), len(x_test)))
51
52     return x_train, y_train, x_test, y_test
53
54
55 def showImage(x, y, gx, title='image', size=128):
56     img = Image.new("RGB", (size*3, size))
57     im = np.append(x, y, axis=1)
58     im = np.append(im, gx, axis=1)
```

```
59      im = list(im.reshape(size * size * 3, 3))
60      im = [map(int, z) for z in im]
61      im = map(tuple, im)
62      img.putdata(im)
63      img.show(title)
64
65  def saveImage(x, y, gx, title='', saveDir='', size=128):
66      img = Image.new("RGB", (size*3, size))
67      im = np.append(x, y, axis=1)
68      im = np.append(im, gx, axis=1)
69      im = list(im.reshape(size * size * 3, 3))
70      im = [map(int, z) for z in im]
71      im = map(tuple, im)
72      img.putdata(im)
73      if not isdir(saveDir):
74          makedirs(saveDir)
75      img.save(saveDir + str(title) + '.png')
76
77  if __name__ == '__main__':
78      readData()
```

Listing 3: train.py for training sample code

```
1  from tqdm import *
2  from model import generator, discriminator, cGAN
3  from preprocess import readData, showImage, saveImage, preprocess, deprocess
4  from keras.models import Model
5  from keras.optimizers import Adam
6  import numpy as np
7  from PIL import Image
8  import math
9  import keras.backend as K
10
11
12 EPOCH = 50
13 BATCH_SIZE = 8
14 N_FILTER = 64
15 N_LAYER = 4
16 L1_WEIGHT = 100
17 N_TRAIN = 2968
18 N_VALID = 256
19 INPUT_SIZE = 128
20 DATASET = '../datasets/cityscapes/'
21
22 G = generator(nf=N_FILTER)
23 D = discriminator(nf=N_FILTER, n_layers=N_LAYER)
24 GAN = cGAN(G, D)
25
26 G_optim = Adam(lr=0.0002, beta_1=0.5)
27 G.compile(optimizer=G_optim, loss='binary_crossentropy')
28 G.summary()
29
30 D_optim = Adam(lr=0.0002, beta_1=0.5)
31 D.compile(optimizer=D_optim, loss='binary_crossentropy')
32 D.summary()
33
34 GAN.compile(optimizer=G_optim, loss=['mean_absolute_error', 'binary_crossentropy'], loss_weigh
35
36 x_train, y_train, x_valid, y_valid = readData(dataset=DATASET, n_train=N_TRAIN, n_valid=N_VALI
37
38 patch_size = D.output_shape[1]
39
40 real_label = np.array([1] * BATCH_SIZE * patch_size * patch_size).reshape(BATCH_SIZE, patch_si
41 fake_label = np.array([0] * BATCH_SIZE * patch_size * patch_size).reshape(BATCH_SIZE, patch_si
42
```

```
43 real_vlabel = np.array([1] * len(x_valid) * patch_size * patch_size).reshape(len(x_valid), pat
44 fake_vlabel = np.array([0] * len(x_valid) * patch_size * patch_size).reshape(len(x_valid), pat
45
46 print("Start training")
47 print("Batch size = %d"%(BATCH_SIZE))
48 print("# of filter for first layer = %d"%(N_FILTER))
49 print("# of layer for discriminator = %d"%(N_LAYER))
50 print("lambda for L1 = %d"%(L1_WEIGHT))
51
52 for epo in range(1, EPOCH+1):
53     print("Epoch %d/%d"%(epo, EPOCH))
54     train_size = len(x_train)
55     valid_size = len(x_valid)
56     n_batch = train_size / BATCH_SIZE
57
58     # train on batch
59     for i in tqdm(range(n_batch)):
60         x_batch = x_train[i*BATCH_SIZE:(i+1)*BATCH_SIZE]
61         y_batch = y_train[i*BATCH_SIZE:(i+1)*BATCH_SIZE]
62
63         gen = G.predict_on_batch(x_batch)
64         fake = np.append(x_batch, gen, axis=3)
65         real = np.append(x_batch, y_batch, axis=3)
66
67         D.trainable = True
68         D.train_on_batch(real, real_label)
69         D.train_on_batch(fake, fake_label)
70
71         D.trainable = False
72         GAN.train_on_batch(x_batch, [y_batch, real_label])
73
74     # test on validation set
75     gen = G.predict_on_batch(x_valid)
76     fake = np.append(x_valid, gen, axis=3)
77     real = np.append(x_valid, y_valid, axis=3)
78
79     lossD_real = D.test_on_batch(real, real_vlabel)
80     lossD_fake = D.test_on_batch(fake, fake_vlabel)
81     errG = GAN.test_on_batch(x_valid, [y_valid, real_vlabel])
82     lossD = (lossD_fake + lossD_real) / 2
83     print("On validation set, lossD = %f, L1_G = %f, lossG = %f"%(lossD, errG[0], errG[1]))
84     for i in range(5):
85         #showImage(deprocess(x_valid[i]), deprocess(y_valid[i]), deprocess(fake[i,:,:,3:]), st
86         saveImage(deprocess(x_valid[i]), deprocess(y_valid[i]), deprocess(fake[i,:,:,3:]), str
87     # save weights for every 10 epoch
88     if epo % 10 == 0:
89         G.save_weights("../model/G_weight.h5")
90         D.save_weights("../model/D_weight.h5")
91
92 G.save_weights("../model/G_weight.h5")
93 D.save_weights("../model/D_weight.h5")
```