CS230

Text Replacement and Generation in Images Using GANs

Andrew Sharp
Department of Computer Science
Stanford University
awsharp@stanford.edu

Christian Gabor
Department of Computer Science
Stanford University
gaborc@stanford.edu

Joseph Son
Department of Statistics
Stanford University
joeson@stanford.edu

Abstract

Currently, it is not easy to replace text in images while maintaining the same style and font. Replacing logos and text appearing in real world images requires manual labor. In this paper, we describe a GAN which automatically replaces a character in an image with another input character while maintaining the original image style. We also extend this model to work on longer strings by stitching together different input characters to create an image containing the desired string.

1 Introduction

In this project, we build upon existing GAN [1] research and explore conditional GAN (cGAN) [2] architectures to replace a character in an image with another input character. Specifically, the input to our generator is an image containing one of 62 characters (0-9, A-Z, a-z), as well as a label representing another one of these 62 characters. The goal of the model is to output a version of the original image in which the original character has been replaced by the desired input character while maintaining the style (e.g. background, font color) of the image.

We believe this work could serve as the foundation for wide-scale applications such as producing synthetic data for optical text recognition in order to automatically increase the size of existing datasets. This work could also be used for quick replacement of text in graphic design, such as for logos, animations, or video games. One potential real world application that we see for such synthetic data is modifying route numbers on images of buses, which could be used to produce training data so that visually impaired people can get an OCR recognizer app for bus routes without needing to ask other people.

2 Related work

Font generation: We looked at papers which use neural networks to produce new fonts. Atarsaikhan et al. [3] use a CNN architecture which takes a text image and adds style to it based on either another font or a simple style image involving a black pattern on a white background. However, this setting is much simpler than our intended setting of replacing text in real color images and also does not involve modifying the characters in an image, instead only changing their style. We also looked at work in style-consistent font generation using a GAN approach [4]. This paper used a GAN to produce new fonts (consisting of all 26 uppercase letters), which is similar to our goal of being able to produce any desired character in a given style. However, they were not attempting to match any existing fonts and were instead only generating completely new ones, and they were again limited to black and white images.

Conditional GANs: The most similar work to our intended model was that of Azadi et al. [5], who used a conditional GAN to produce a complete font given only a few examples of letters in each desired style. While they were still limited to more structured images of isolated characters on white backgrounds, this problem was fairly similar to ours, with the most significant difference being that their model was given several examples of each font while ours only sees a single character. This motivated us to use a conditional GAN [2] for our model as well. A conditional GAN (cGAN) involves providing both generator and discriminator a class label input so that when that label is provided, the discriminator ground truth probability is conditioned on a certain class image distribution. Conditioning the data this way can allow the generator to make images closer to the label distribution when it attempts to fool the discriminator. The effect of this is that we can provide the conditional class as input to the generator to get an image that looks like the particular class we want as output. Another paper which used a conditional GAN approach to font style transfer was Bhunia et al. [6], which again attempted to transfer certain text to match a new font from which only a few characters have been seen. This paper was again limited to black text on white backgrounds and required several character examples from the desired style, but it provided further evidence that conditional GANs are well-equipped to transfer a partially observed font style to new letters.

3 Dataset

We use the Chars74K dataset [7] which has images of 7,705 characters cropped from larger natural images. The images in the dataset have a wide variety of sizes because each character image's resolution is maintained from its larger original image. Each of these characters has one of 62 possible labels (because it could be an uppercase letter A-Z, a lowercase letter a-z, or an integer 0-9). The images consist of RBG channels , and for preprocessing, we normalize the images to have intensity values in the range $[-1, 1]$ and resize them to size 28×28. We used 400 of these images as our evaluation set for the style images we trained the generator to match and the rest were used for training the discriminator to recognize different characters. Figure 1 shows some example images.

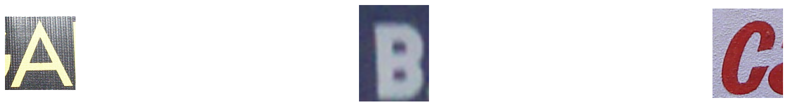


Figure 1: Example character images from Chars74K

4 Methods

Our initial model is a traditional cGAN, which uses a target character class which is input into both the discriminator and the generator of a DCGAN architecture and turned into a class embedding. This embedding allows us to use the target class label to select the desired character output. The generator then attempts to produce an image which the discriminator will believe comes from the target class. The overall structure of this model can be seen in Figure 2, and the details of each layer are found in Figure 3.

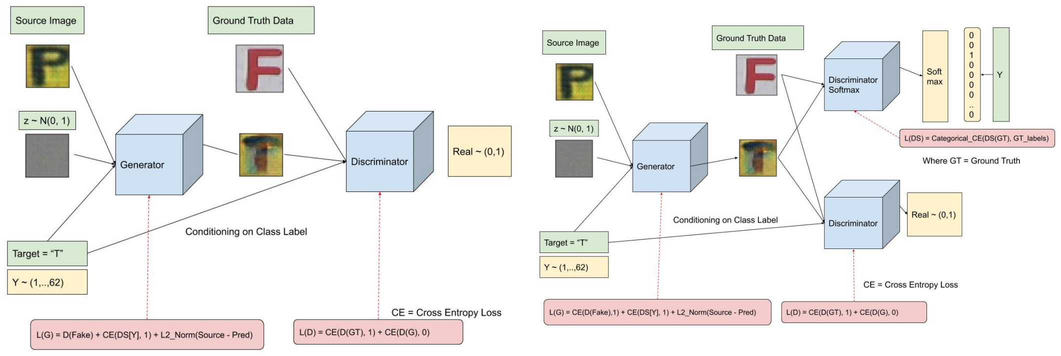
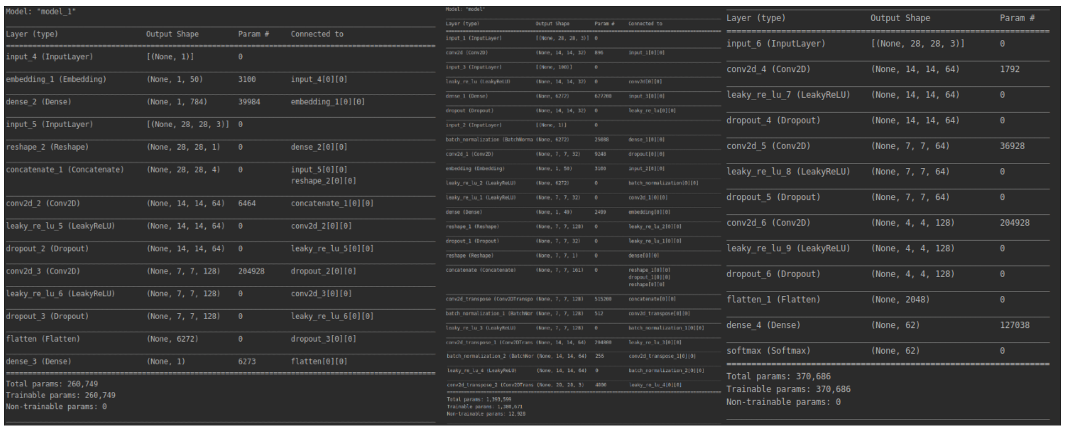


Figure 2: Initial cGAN architecture (left) and updated architecture with softmax discriminator (right)



```
Model: "model_1"
_________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
=================================================================================================
input_4 (InputLayer)            [(None, 1)]          0
_________________________________________________________________________________________________
embedding_1 (Embedding)         (None, 1, 50)        3100        input_4[0][0]
_________________________________________________________________________________________________
dense_2 (Dense)                 (None, 1, 784)       39984       embedding_1[0][0]
_________________________________________________________________________________________________
input_5 (InputLayer)            [(None, 28, 28, 3)]  0
_________________________________________________________________________________________________
reshape_2 (Reshape)             (None, 28, 28, 1)    0           dense_2[0][0]
_________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 28, 28, 4)    0           input_5[0][0]
                                                                 reshape_2[0][0]
_________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 14, 14, 64)   6464        concatenate_1[0][0]
_________________________________________________________________________________________________
leaky_re_lu_5 (LeakyReLU)       (None, 14, 14, 64)   0           conv2d_2[0][0]
_________________________________________________________________________________________________
dropout_2 (Dropout)             (None, 14, 14, 64)   0           leaky_re_lu_5[0][0]
_________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 7, 7, 128)    204928      dropout_2[0][0]
_________________________________________________________________________________________________
leaky_re_lu_6 (LeakyReLU)       (None, 7, 7, 128)    0           conv2d_3[0][0]
_________________________________________________________________________________________________
dropout_3 (Dropout)             (None, 7, 7, 128)    0           leaky_re_lu_6[0][0]
_________________________________________________________________________________________________
flatten (Flatten)               (None, 6272)         0           dropout_3[0][0]
_________________________________________________________________________________________________
dense_3 (Dense)                 (None, 1)            6273        flatten[0][0]
=================================================================================================
Total params: 260,749
Trainable params: 260,749
Non-trainable params: 0
_________________________________________________________________________________________________
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 28, 28, 3)]  0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 14, 14, 32)   896         input_1[0][0]
__________________________________________________________________________________________________
input_3 (InputLayer)            [(None, 100)]        0
__________________________________________________________________________________________________
leaky_re_lu (LeakyReLU)         (None, 14, 14, 32)   0           conv2d[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 6272)         627200      input_3[0][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 14, 14, 32)   0           leaky_re_lu[0][0]
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, 1)]          0
__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 6272)         25088       dense_1[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 7, 7, 32)     9248        dropout[0][0]
__________________________________________________________________________________________________
embedding (Embedding)           (None, 1, 50)        3100        input_2[0][0]
__________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)       (None, 6272)         0           batch_normalization[0][0]
__________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 7, 7, 32)     0           conv2d_1[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 1, 49)        2499        embedding[0][0]
__________________________________________________________________________________________________
reshape_1 (Reshape)             (None, 7, 7, 128)    0           leaky_re_lu_2[0][0]
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 7, 7, 32)     0           leaky_re_lu_1[0][0]
__________________________________________________________________________________________________
reshape (Reshape)               (None, 7, 7, 1)      0           dense[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 7, 7, 161)    0           reshape_1[0][0]
                                                                 dropout_1[0][0]
                                                                 reshape[0][0]
__________________________________________________________________________________________________
conv2d_transpose (Conv2DTranspo (None, 7, 7, 128)    515200      concatenate[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 7, 7, 128)    512         conv2d_transpose[0][0]
__________________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)       (None, 7, 7, 128)    0           batch_normalization_1[0][0]
__________________________________________________________________________________________________
conv2d_transpose_1 (Conv2DTrans (None, 14, 14, 64)   204800      leaky_re_lu_3[0][0]
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 14, 14, 64)   256         conv2d_transpose_1[0][0]
__________________________________________________________________________________________________
leaky_re_lu_4 (LeakyReLU)       (None, 14, 14, 64)   0           batch_normalization_2[0][0]
__________________________________________________________________________________________________
conv2d_transpose_2 (Conv2DTrans (None, 28, 28, 3)    4800        leaky_re_lu_4[0][0]
==================================================================================================
Total params: 1,393,599
Trainable params: 1,380,671
Non-trainable params: 12,928
__________________________________________________________________________________________________
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 28, 28, 3)]       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 64)        1792
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
dropout_5 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 128)         204928
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_6 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 62)                127038
_________________________________________________________________
softmax (Softmax)            (None, 62)                0
=================================================================
Total params: 370,686
Trainable params: 370,686
Non-trainable params: 0
_________________________________________________________________
```

Figure 3: Keras architectures: Discriminator (left), Generator (middle), Softmax Discriminator (right)

Our improvement upon the basic cGAN architecture is an embedding in the generator that takes both the replacement target character and the target style image to which it should maintain similarity. We use several CNN layers to down sample the source image before up sampling with the input label and noise. The discriminator takes the target class label and will decide if the output image in the new style is realistic conditioned on the target label. We later added a second softmax discriminator to force the generator to produce the desired output class.

To maintain style, we use an L2 distance metric between the generated and source images in the generator loss function. This forces the GAN to output similar background and font colors. The loss contribution is weighted with γ which must be carefully tuned to not overpower the discriminator loss.

We then stitch these stylized generated characters into a text string and use Gaussian smoothing on the borders to get a functional text replacement. This is a first attempt at producing a full output string, while a later improvement could involve a larger model to take in full string images.

5 Experiments/Results/Discussion

We evaluated two different model architectures: first, cGAN with just the source embedding; second, cGAN with an additional softmax target class discriminator. The hyperparameters that we tuned include mini-batch size, generator learning rate, discriminator learning rate, γ to scale L2 norm between input and output image, and β to balance the softmax contribution to the generator loss.

cGAN with Source Embedding: Specifically, we run the cGAN with source embedding using the following hyperparameters: **mini-batch size**: 64; **generator learning rate**: 6e-4 (Adam); **discriminator learning rate**: 1e-4 (Adam); and **gamma**: 5e-3.

We use a mini-batch size of 16 so that we are able to make more finely tuned gradient adjustments. We find that tuning the γ hyperparameter has the largest effect on the results of the GAN. As such, we looked at values larger and smaller than 5e-3 but generally found the best results with 5e-3. Looking at the actual results of the GAN, we can see that even in the best hyperparameter setting, most of the generated images struggled to match the target label. We evaluated a set of 400 images produced by the generator and found that only 1.25 percent had successfully been changed to the desired character. However, if we look at the sample in Figure 4, we have a few instances where there is a clear matching.

In addition to visually analyzing the results of the model, we also can look at the log loss values and accuracy of the generator and discriminator. We see that the accuracy of the discriminator on the fake images stays relatively high while the accuracy on the real images is lower. Ideally, we would want to see these accuracies converge around 0.5, indicating the discriminator cannot tell the real images from the generated ones.

cGAN with Softmax Discriminator: Adding a softmax discriminator improved our initial model performance, but we found that it was more challenging to train the generator by selecting hyperparameters. Our generator loss was the following (where $\beta = 0$ in our initial model):

$$Loss(G) = BinaryCrossEntropy(D(G), 1) + \beta BinaryCrossEntropy(D_{softmax}[targetlabel], 1) + \gamma||Source - G||_2$$

where $D_{softmax}[targetlabel]$ represents value at the index of target label in the softmax output vector. The generator will minimize this part of the loss when the softmax is confident the image matches the desired target label. The loss functions for the discriminators are given below:

$$Loss(D) = BinaryCrossEntropy(D(GroundTruthImg), 1) + BinaryCrossEntropy(D(G), 0)$$

$$Loss(D_{softmax}) = CategoricalCrossEntropy(D_{softmax}(GroundTruthImg), GroundTruthLabels)$$

The generator model does not contribute to the softmax discriminator loss, as the softmax discriminator only trains on the groundtruth data. But this discriminator still contributes to the generator loss, while both are training from initialization.

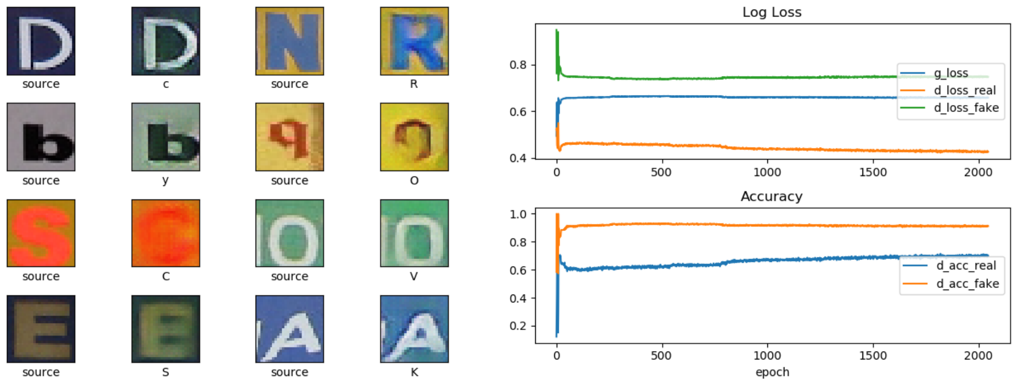


Figure 4: Results, loss, and accuracy for initial cGAN model

We run the cGAN with additional softmax discriminator using the following hyperparameters: **minibatch size**: 16; **generator learning rate**: 6e-4 (RMSProp); **discriminator learning rate**: 1e-4 (Adam); **softmax discriminator learning rate**: 1e-5 (Adam); **gamma**: 5e-2; and **beta**: 0.2.

Here we adjust the gamma parameter to be smaller so that we penalize the distance from the original image less. We also reduce the learning rate of the discriminator so that the generator has more time to improve and produce more realistic images. We found that this model produced good images much faster (around 100 epochs) than the initial model (which had to train for at least 1000 epochs to produce good results).

Looking at the results output from this model in Figure 6, we see that there are also instances where the image is not able to match the target label. However, we find that by adding the additional softmax discriminator, we are able to get more examples of successful images generated with convincing matching of the target label. Our evaluated accuracy on the 400 source/result pairs generated by this new model was 17 percent, a significant improvement over the non-softmax model. Some of the more convincing examples are shown in figure 5.

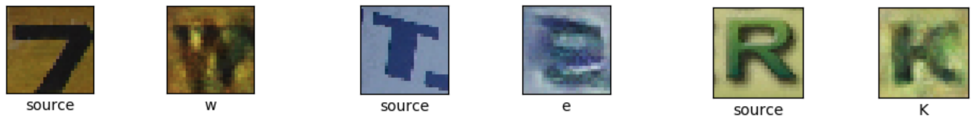


Figure 5: Example source/result pairs output by the softmax model

Again, we can look at the log loss values and accuracy of the generator and discriminator for the softmax model. We see from the graph of the losses that the discriminator quickly overpowers the generator, which is also reinforced by the graph of the accuracies (the accuracy of the discriminator on both real and fake images almost reaches 1.0).

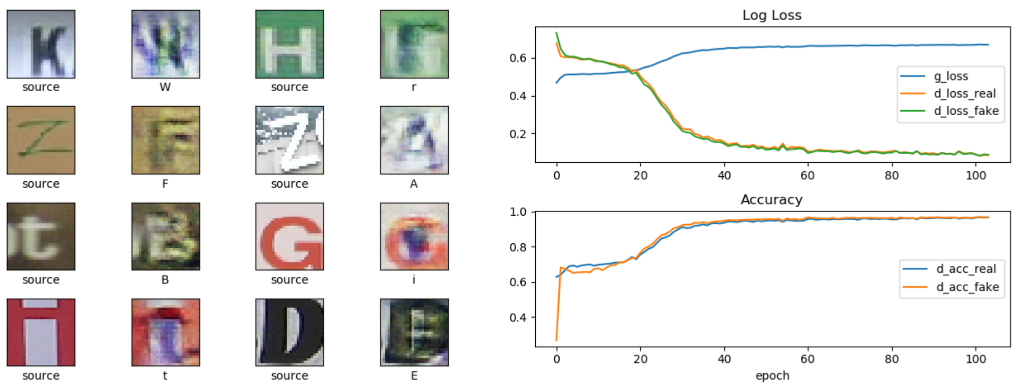


Figure 6: Results, loss, and accuracy for cGAN model with softmax discriminator

6 Conclusion/Future Work

We found that incorporating the softmax discriminator into our model allowed it to perform noticeably better, no longer producing a significant number of images which looked almost identical to the style image. This is because the generator was forced to produce an image which resembled the desired character more than the original character in order to fool the softmax discriminator. We found, however, that the generator was still able to fool the discriminator by reshaping the background of the image to resemble the desired character instead of reshaping the character itself. This led to many results in which the character and background colors were flipped from the style image to the output. A more complex discriminator loss function might be useful for determining whether the recognized character is actually a part of the background shape.

We would have liked to experiment more with the hyperparameters balancing both the various components of the loss functions and the generator and discriminator learning rates. We found that it was very difficult to produce a model that consistently maintained similarities with the desired style images but also consistently changed the style images enough to match the desired characters and fool the discriminator. With more resources, we could perform a more exhaustive search over the hyperparameter space to find a combination that balances these objectives.

Another modification we would like to make in the existing model is the evaluation of similarity to the style image. We currently simply use an L2-norm of the per pixel difference between the style image and the generated one to evaluate this similarity, but this could be replaced by a more sophisticated method such as that used in neural style transfer where an image is put through a convolutional neural network and one of the intermediate hidden layers is used to represent its style. This could help to make up for the fact that some pairs of letters (such as C and G) are more likely to have similar shapes than other pairs and that these pairs will be penalized less by the similarity term of the generator loss as well as the tendency of the generator to produce the new character in the background color instead of the color of the style image's character (as can be seen by the "3" in Figure 7).

Given more time, we would have liked to build upon this model further by training it to replace entire words instead of just a single character at a time. This would require a more complex architecture, as we would likely need to incorporate an RNN into the embedding-generating process to account for words of varying lengths. Instead, we had the model output a single image for each character and then stitch these images together, as seen in Figure 7.

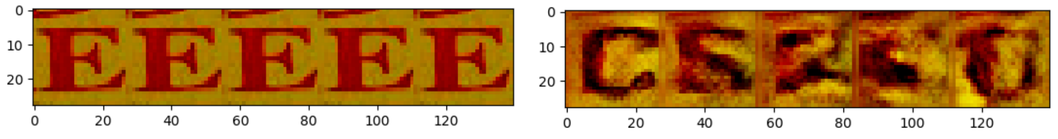


Figure 7: The desired style image (left) and the GAN output for the input string "CS230" (right)

7 Contributions

Andrew contributed to the literature review, AWS setup, image processing code, model evaluation, and producing the final report.

Chris contributed to writing the code for the model architectures, creating the loss functions, collecting data for the input data pipeline, training the models, evaluating inference and testing hyperparameters.

Joe contributed to literature review, model evaluation, performance evaluation, and producing the final report and poster.

8 Code

Our code can be found at the following link: https://github.com/gaborchris/DeepReplace

It was implemented in Python 3.6 using Tensorflow 2 [8] and Keras [9]. It is in part based on Jason Brownlee's conditional GAN implementation [10] and the Tensorflow DCGAN tutorial [11].

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, June 2014.

[2] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, November 2014.

[3] Gantugs Atarsaikhan, Brian Iwana, Atsushi Narusawa, Keiji Yanai, and Seiichi Uchida. Neural font style transfer. In *14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, pages 51–56. IEEE, November 2017.

[4] Hideaki Hayashi, Kohtaro Abe, and Seiichi Uchida. Glyphgan: Style-consistent font generation based on generative adversarial networks, May 2019.

[5] Samaneh Azadi, Matthew Fisher, Vladimir Kim, Zhaowen Wang, Eli Shechtman, and Trevor Darrell. Multi-content gan for few-shot font style transfer. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, June 2018.

[6] Ankan Kumar Bhunia, Ayan Kumar Bhunia, Prithaj Banerjee, Aishik Konwer, Abir Bhowmick, Partha Pratim Roy, and Umapada Pal. Word level font-to-font image translation using convolutional recurrent generative adversarial networks. In *International Conference on Pattern Recognition*, January 2018.

[7] T. E. de Campos, B. R. Babu, and M. Varma. Character recognition in natural images. In *Proceedings of the International Conference on Computer Vision Theory and Applications*, February 2009.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[10] Jason Brownlee. How to develop a conditional gan (cgan) from scratch. `https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/`, July 2019.

[11] Google. Deep convolutional generative adversarial network. `https://www.tensorflow.org/tutorials/generative/dcgan`.