Generating Facial Expressions

Jonathan Suit
Georgia Tech

Abstract

In this report, I use CAS-PEAL dataset [1] to try and generate facial expressions. That is, I take in, say, a smiling face, and try to generate a face that is neutral. I build off Ghodrati's, et al. *[2] to see if I generate facial expressions. Finally, I use a variant of Deep Convolutional Adversarial Networks, hereafer DCGAN's, [7] to generate a facial expressions.*

1. Introduction

Realistic facial images are hard to generate. Put differently, it is hard to learn a distribution to generate realistic faces. There are lots of reasons it is hard: images are typically high-dimensional, highly non-linear relations are involved in generating an image of a face, and capturing facial expressions requires capturing intricate minutia of the face. The latter can be difficult for CNs.

The CAS-PEAL dataset gives you about 300 pairs of users where each user gives 6 different facial expressions: neutral, laughing, frowning, surprised, eyes closed, and eyes open. The images are around 380 x 480 before being resized to 128 x 128. The images include just the faces, however, the alignment and pose of the faces is very rough.

The format of this paper is as follows: introduce the two-step process, DCGAN's, go over the my variations of these two networks. Finally, I give visualizations to show what the network has and has not learned.

In the end, I show that DCGANs seem promising. Using them I can reconstruct the original image with great accuracy. However, the network has not learned to modify the image.

2. Related Work

There are two major approaches I will focus on: the "two-step" CN and DCGAN.

2.1. Two Step

The two-step process comes from [2]. The network they employ can best be seen by the following figure from [2]

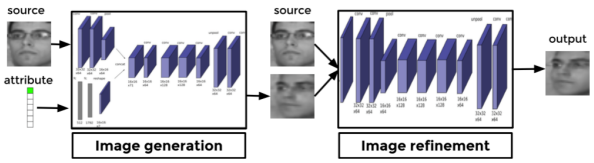

Figure 1. The two-step process.

The network takes in an image, a one-hot vector and then produces a new image. This output image should be an image with the pose corresponding to the which index was turned on in the one-hot vector. The next step, a.k.a., the second step in the two-step process, is there to refine the image, to make it sharper.

It's important to see that this network uses a encoder-decoder framework within each CN, the image-generation and image-refinement net. For a 32 x 32 image, the network condenses its representation down to 16 x 16 before decoding that representation back to 32 x 32 in pixel space.

The network is trained in a two-step process. First, you learn to generate the image and then you use a second network to refine the image generated by the first network.

Both networks use MSE as the loss function.They also use SGD with a batch size equal to 32, momentum = .95, a learning rate = 1e-5. All weights initalized according to [4], which in torch7 is implemented as $math.sqrt(4/fan_{in} + fan_{out})$ for each layer.

2.2. DCGAN

*DCGAN*s are generative convolutional networks based off of GoodFellows's Generative Adversarial Networks, *GAN*s, in [3]. See section 2.2.1 for a brief intro into *GAN*s. DCGAN's are *GAN*s with specific architectural guidelines:

1. No pooling layes. Use either convolutions with stride > 1 and use fractional-strided convolutions, such as in [6].

2. Use Spatial Batch Normalization from [5]

3. For deeper architectures, remove the fully connected hidden layers.

4. Use ReLU in all stages at of the generator except the last unit, which uses a Tanh activation layer.

5. Use Leaky ReLU in all layers of discriminator.[1]

These guidelines make it easier to train *GAN*s as *GAN*s are notoriously hard to train.

2.2.1 GANs

Briefly put, *GAN*s work by pitting a generator vs. a discriminator. The generator should try to generate images to fool a discriminator. The discriminator's goal is to be able to detect th "fakes", i.e., the output of the generator. They show in the paper that the best outcome occurs when the generator's output matches the true data distribution.[2]

*GAN*s are notoriously difficult to train as it involves keeping the discriminator and the generator competitive with one another. The discriminator cannot be too good at detecting fakes relative to the generator or else the generator does not receive a strong enough error signal to update its parameters so it can become better at fooling the discriminator. If the generator is too good at fooling the discriminator, the generator is probably not being forced to explore the distribution. Thus, it is probably not actually learning a good approximation of a distribution.

2.3. My Model: the Two-Step process

2.3.1 Model

The model is roughly the same as in Figure 1. As I go along, I describe the changes to this base model and the effects it causes.

2.3.2 Training Process

I use an image size of 32x32. I quickly dropped this as the images were too small to see whether any errors were due to the shrinking of the image (the original images were in the 380 x 400 range) or due to the CN. Originally, I also kept the images in the [0,1] range. But (on 32 x 32 sized images) I had a harder time learning than with the image in the [0,255] range. More accurately, when using SGD, minimal batch normalization, and images in the [0,1] range, the network outputs images of random noise. By minimal batch normalization, I mean batch normalization after the first convolutional layer in the first network and batch normalization after the second convolutional layer in the second network.

[1]Leaky ReLU is defined as $max(0, x) + a * min(0, x)$. In [7], they use $a = .2$

[2]More accurately, the generator's output should look exactly like it was sampled from the true data distribution. The generator does not explicity output a distribution.

By changing one of these (SGD, more batch normalization, or images within [0-255] range) my network(s) learned something. Later, I found that a lack of batch normalization was able to fix these problems. I still stuck with adam over SGD as SGD was more fickle about learning rates and even loss functions (e.g., SGD with a variant of the L1 loss did not learning anything). Visually, we can SGD's difficulties

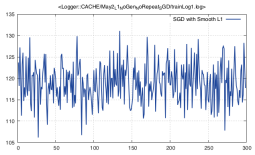

Figure 2. SGD issues with SmoothL1 Loss.

with different loss functions. Figure 1 shows the issue that SGD has smooth L1 loss. Figure 3 shows that the SGD has trouble with the mean squared error (MSE) loss without batch normalization. Finally, the benefits of using batch normalization can be seen in Figure 4. Lastly, I would like

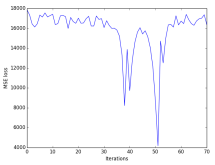

Figure 3. (Left.) SGD troubles with MSE with no batch normalization

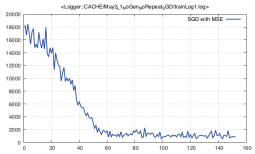

Figure 4. (Right.) SGD with MSE with batch normalization

to point out that using adam let me experiment with a number of higher learning rates, whereas SGD would often diverge.

2.3.3 Visualization

The visualization reveals one of the biggest issues with using CNs in the two-step process to generate facial expressions: blurriness.

Figure 5. 32x32 blurry Image, result of image generation model. The person should be frowning.

This is the result of 4 epochs. This means that the network has seen this pair of images 4 times. While this does not seem like a lot, the loss function shows otherwise. The model converges well before the end of the first epoch. And, as can be seen in Fig. 6, the image refinement step does not seem to make much of a difference.

Figure 6. 128x128 blurry Image from image refinement model. Input is smiling expression. Output should be frowning. Generated image is far left, Input is the middle, and the output is the far right.

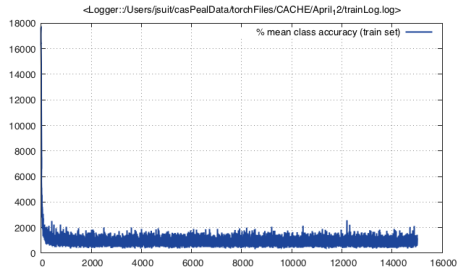

Figure 7. MSE Loss plot for the image refinement step. Well before the end of the first epoch, the model converges, and then it stays there. The loss acts the same for the image generation step. With L1 Smooth loss, you get the same sort of plot, though the error's magnitude is not as big. For MSE, the loss converges to around 800 and for the L1 Smooth loss it converges to about 20.

The visualizations reveal another issue: the model is biased towards the identity function as can be seen in Fig. 8.

Figure 8. Generated Image = left, input = middle, output = right. Even though the image is blurry, we can still see from the location and pose of the head that the generated image is looks more like the input than the output.

I will talk more about this issue in section 3

2.3.4 Refinements

Following much of [7], I got rid of maxpooling layers, which were replaced with convolutions with a stride > 1, I replaced any upsampling layers (which do not have any parameters) with deconvolution layers from [6], batch normalization was used after every layer except after the last layer (in both the image generation step and image refinement step), a ReLU was used after every layer in the image generation network, and a leaky ReLU (with hyper-parameter .2) was used in the image refinement network.[3]

However, this did not result in any improvements. The loss functions has a similar kind of shape and images generated still look very blurry. Note that the Smooth L1 loss is less blurry than MSE, but it is still very blurry. Experimentation showed that any gains in sharpness are the result from the change in loss function, not any refinements made to the model.

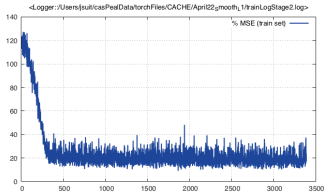

Figure 9. Loss Smooth L1 loss

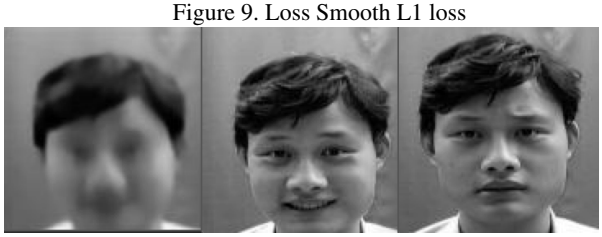

Figure 10. Generated Image with Smooth L1 loss. Slighly sharper than generated image from Fig. 6.

2.3.5 Extensions

I take the second network in the two-step process (i.e., the image refinement network in Fig. 1) and a discriminative network to it in order to see if any of the features are being learned are good at disciminating they type of facial expression. Specifically, I take the output of first network (the image generation network in Fig. 1), run it through the first three layers of the second network, and use the output of these features to both generate an image and predict what facial expression we are seeing. The part of the network that tries to generate a facial expression has not changed; however, to predict the facial expression I attach a CN with 3 convolutional layers and a fully connected one: Convolution(128,32,5,5,1,1), Convolution(32,16,5,5,5,5), Convolution(16,1,2,2,2,2), FullyConnected(36,6). Each convolution is followed by a batch normalization and Leaky ReLU. The fully connected layer also has a ReLU as its activation. [4]

This network using Smooth L1 loss gets 58.97% of training examples correct and 48% of test cases correct (the split between the training and test cases was randomly chosen to be 50-50). Using MSE, the accuracy drops to 40% and 20% accuracy in the training and test cases, respectively. It's

[3] Actually, ReLU and Leaky ReLu were both used and the difference was indiscernible.

[4] Convolution(a,b,c,d,e,f) is a convolutional layer that takes in an input of depth a, outputs b filter, kernel of size c x d, and has stride e, f.

surprising that the Smooth L1 loss function does so much better. It's not surprising that the the accuracy is not terribly high for training or testing. The split between train-test should probably be closer to 80-20 and the classifier in the second network should have (a) more parameters and (b) should not have such a huge stride in the second convolution. (I did it because: first, the classification accuracy is not the most important thing we are interested in, and second, I wanted to quickly get the input down to an output of size 6, the number of facial expressions possible.) What is even more interesting is that using the Smooth L1 loss, if you train the everything together as one big network (that is, the image generation network and the image refinement network, where the image refinement network both outputs an image and a classification) you can get 95% correct on the training cases.[5] However, the images generated by this network are terrible and barely resemble a human face.

3. DCGAN Model

Unable to get past the blurriness issue, I moved to using a modified version of DCGAN model as in [7]. The results, in terms, of sharpness of the image were *much* better. However, it puts into focus (no pun intended) the other problem: the network acts like an autoencoder that is trying to reproduce its input.

3.1. DCGAN

I don't use the exact same DCGAN as mentioned in [7], but it's close. First, I use all the refinements from the two-step process. Put differently, I use all architetural guidelines from section 2.2 except I keep the fully connected layers. And I also use adam as my optimizer. My discriminative model is the exact same as the one at https://github.com/soumith/dcgan.torch/blob/master/main.lua[6], except I have an additional convolutional layer at the top of the network since my imagesize is two times as large as the input expected in the original discriminator code.

3.1.1 Generator

My generative model is different from the one in [7]. First, the generative model is the has the same structure from the image generatrion network from section 2.3. It's output, as you might recall, are images of size 128x128. I do not reuse the weights, but it has a different structure from the DCGAN in [7]. It's input are images of size 12 x 128 and a one-hot vector of size 6, which tells the network how to modify the image. Secondly, I do not feed random noise to the generator. Instead, I feed in an image of the person's face that we wish to modify. Lastly, my generator has two loss functions: (1) reconstruct the output image and (2) fool the generator.

3.1.2 Disciminator

I try two different things with the disciminator. First, the discriminator tries to predict whether this is an image from the true data distribution or whether it is an image generated by the generator.[7] I also attempt to see how well it can classify the facial expression of an image, regardless of whether it is from the generator or is an image from the dataset.

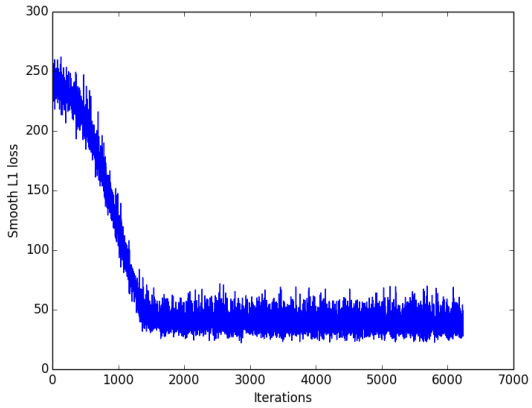

Figure 11. The Generator loss

3.2. Visualization

The following visualizations reveal that the DCGAN model gets around the blurriness issue, but is not learning how to modify the image. In fact, the one-hot vector plays no meaningful role in generating the image. Of course, this affected the previous model, but it is striking when looking at the generatred images.

As the reader can see, the generated image is practically identical to the input. And it does not matter what index is turned on in the one-hot vector. All the generated images look virtually identical to the input regardless of what the one-hot vector is. Nor were these images cherry-picked. All the images generated are as clean and sharp as the input image. In short, I accidentally learned the identity function. It should be pointed out that this could be useful, for perhaps the features learned might be useful for some discrimination task.

3.3. Trying not to learn the identity function

The next attempts were seeing as to how I could avoid my network learning to perfectly (or almost perfectly) re-

[5] I have not had time to look at the test accuracy, but more than likely, it is much higher.

[6] The disriminator starts on line 84.

[7] An image from the true data distribution are the images with the facial expression that we wish to generate.

construct the input.

I tried several avenues to figth against this, but none were really successful.

The "best" way to avoid reconstructing the input image was train on an image for several iterations before getting the next batch. Around 8-16 iterations per batch seemed sufficient. 2 iterations was not sufficient. This strategy worked in the sense that the generated images looked more like the output faces than the input faces. However, the drawback was that the faces looked fairly grotesque.

I would insert generated images, but the images I am allowed to present via the license agreement are not ready in time. However, looking at the faces in [7] gives a good idea of what they look like.

To show that the network is actually learning the identity function and is not just memorizing the images, I took a picture of myself. The network has never seen this image.

Figure 12. The generated image is on the left; the input is on the right. Regardless, of the one-hot vector, the output always looks like the input.

3.4. Why is it learning the identity function?

It's reasonable to ask why the network is learning the identity network. While it is not clear to me, I do have some ideas. First, once the network learns the input, it will be nearly impossible for it to leave this local optima. That's because the subtle changes account for the difference in facial expressions. Thus, the network must make extremely subtle changes to account for these changes in expressions. (Perhaps one can argue that the net has not learned to disentangle the factors that create different facial expressions. Maybe we should make the network even deeper?) Moving away from the input image, once it has learned that, from the point of the view of the CN, will only increase the error.

But why does it move to learn the input in the first place? This is where we need to more experimentation. It could be that all the batch normalizations regularize the network too much. In fact, if you demean the data and divide by the standard deviation of all the images and use the batch normalizations at every layer you reach a much,much worse optima. This could suggest that the network is at the tipping point when it comes to normalizing the data, i.e., over-regularized.

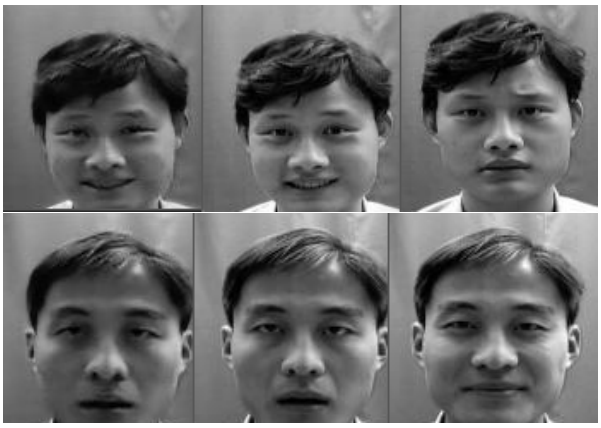

Figure 13. For both rows, the leftmost image is the generated image, the center is input, and right is desired output.

4. Summary

it is hard to generate facial expression because the facial images are high dimensional inputs and modifying them requires learning highly non-linear transformations. Furthermore, CN's have a hard time generating non-blurry images. And once the CN can figure out how to generate an image, it can recreate the original image, but my networks have not figured out how modify the image so that only small parts of the image are modified (e.g., adding a smile).

References

[1] W. Gao, B. Cao, S. Shan, X. Chen, D. Zhou, X. Zhang, and D. Zhao. The cas-peal large-scale chinese face database and baseline evaluations. *Trans. Sys. Man Cyber. Part A*, 38(1):149–161, Jan. 2008.

[2] A. Ghodrati, X. Jia, M. Pedersoli, and T. Tuytelaars. Towards automatic image editing: Learning to see another you. *CoRR*, abs/1511.08446, 2015.

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[5] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[6] H. Noh, S. Hong, and B. Han. Learning deconvolution network for semantic segmentation. *CoRR*, abs/1505.04366, 2015.

[7] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.