EditGAN: Using Generative Adversarial Networks for Image Editing

Category: Generative Modeling

Logan Bruns
Stanford University
lbruns@stanford.edu

Abstract

This project is to explore using conditional Generative Adversial Networks for Image Editing. Primarily around changing, moving, or removing objects in images

1 Overview

The goal of this project is to explore using conditional Generative Adversial Networks (GANs) for Image Editing. Primarily around changing, moving, or removing objects from images and changing the dimensions of existing objects. GANs have been demonstrated to be able to generate high quality images. It would be interesting to see a GAN could be trained to understand an existing image and make a change to that image. So, generate a replacement portion of the image using knowledge based both of previous training and on the context of the specific image. In theory, perhaps it should be to extrapolate like a human and continue a fence behind a person or form a car in the correct orientation for a shape.

This seems to be most likely to be successful if we start with images that are labelled to a pixel level and use this pixel level semantic mapping both to train the model and to tell the model how to manipulate the image.

In order to train a model based on pixel level semantic labels and allow pixel level manipulation while still retaining consistency across the existing image parts we are considering using concepts from Energy-based Generative Adversial Networks[2] (ebGAN) as building blocks. There are several useful concepts developed in this architectural building block. Most notably though 1) the ability to compute the training example loss in a more flexibly way including approaches such as using cross entropy loss against pixel level labels 2) the use of an auto-encoder as an intermediate representation.

When trying to converge to the modified image two approaches were considered.

1. **Conditional mask to Generative Model**
   Where the nearest latent vector for the image would be found, the predicted mask modified, and the forward pass of generative model run to obtain the resulting image
2. **Clipped conditional mask to Generative Model**
   Same as above but modified to only change the portions of the image where the mask are changed.

2 Dataset

The City-Scapes[1] dataset consists of street view images of various cities in Europe captured via a dashcam of car as well as pixel level semantic classification labelling for a subset of the data.

The fine labelled portion of the dataset consists of images are 2048x1024 with pixel level labelling. 2975 images in the training set and 500 in the validation set. Of the original labels only 21 of them are kept for this model: road, sidewalk, building, wall, fence, pole, traffic light, traffic sign, vegetation, terrain, sky, person, rider, car, truck, bus, train, motorcycle, bicycle, license plate, and unknown.

3 Methods

3.1 Frameworks

All the training and exploration was done in python using tensorflow primarily on a GPU with 16gb of memory on linux box. Some work experimentation was done with multi-GPUs and support was added to the

code with the consideration of scaling to out to larger many GPU AWS instances but the most useful work was found to be in error analysis and iteration on loss functions and architecture which could be done easier by carefully babysitting a single linux box.

3.2 Data Loading and Augmentation

The images and labels are added to the training graph as a dataset. Originally preprocessing was done with a python generator but it was found to be slow and switching to dataset graph components speed up epoch training time by approximately a factor of eight.

Dataset pipeline steps:

- Shuffle
- (One time) Create and cache label PNG
- Load image and label PNGs
- Rescale to 4x current training size
- Subsample a cropped window of current training size
- Batch

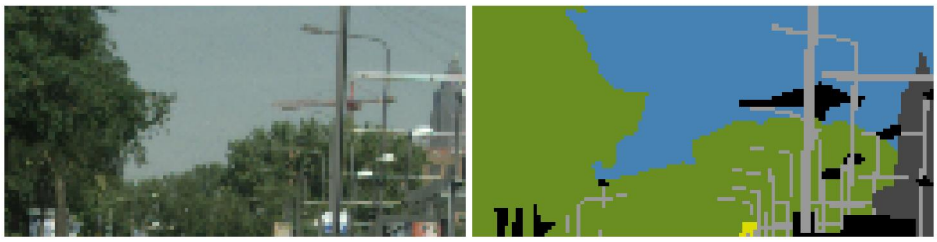

Figure 1: Training Example after Subsampling

Figure 1 shows an example of what is a single training example looks like.

4 Architecture, Model Training, and Hyper-parameter tuning

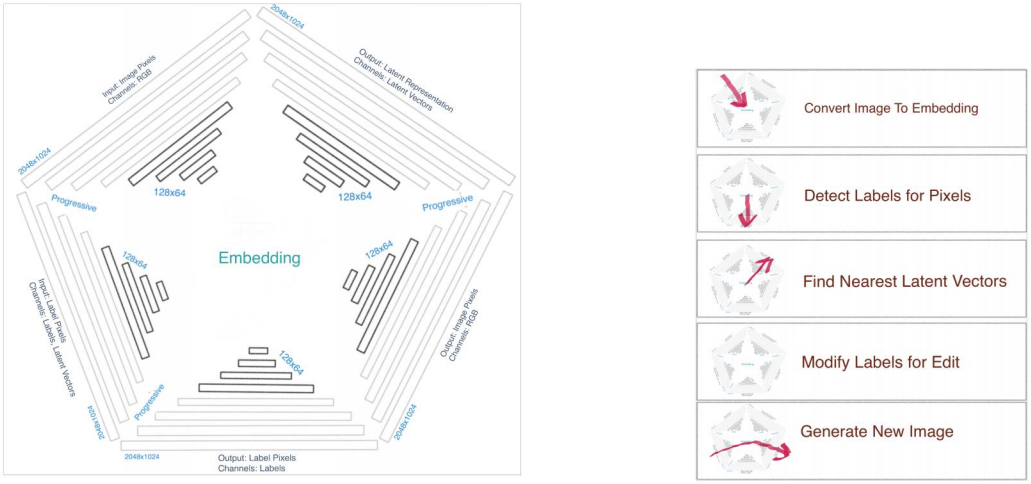

Figure 2: Model Architecture

Figure 2 shows an overview of the model architecture and how it relates to the intended edit flow. The CNNs and DeCNNs are Inception[3] CNN inspired blocks which have 1x1, 3x3, 5x5, and 7x7 (instead of max pooling) filters as well as batch normalization and a skip connection. They all have the same architecture which through testing seems to be complex enough to model the images. Testing the autoencoder block has been useful in evaluating this component and in this current architecture works pretty well as shown in Figure 3. The images on the left are encoded and decoded versions of the originals on the right. Initially, with simpler CNNs the results were not nearly as good.

In general, it has been proven useful to leverage symmetry in the model. Many iterations were tried with a mixture of CNNs and DNNs but this provided problematic in both training, output quality, and scaling training to larger sizes. Although it takes a long time in general it seems to work to scale up an existing model by adding a set of layers as shown in the figure. Training only the new layers at first, then fine tuning across all layers, and then repeating.

Also, note as in ebGAN incorporating an autoencoder losses seems to help with training even of reusable components even if not necessary for final output. For example, half of the pixel autoencoder is used for the discriminator and the other half is used for the generator. Likewise, half of the latent autoencoder is used for generator and the other half for finding the nearest latent vector.

Figure 3: Autoencoder Examples for Evaluating CNNs/DeCNNs (original on the right)

The training is done in three phases: autoencoder training, discriminator training, and generator training. Each has their own loss and many share components but many layers are held constant or frozen while training for a particular objective.

The loss functions for each training phase are listed below:

$$\mathscr{L}_{autoencoder} = \alpha_{autoencoder} * \frac{1}{3 * n_{pixels} * m_{batch}} * \sum_i (ScaledPixel_i - RebuiltPixel_i)^2 \tag{1}$$

$$\mathscr{L}_{real\_labels\_l2} = \frac{1}{n_{pixels} * n_{labels} * m_{batch}} * \sum_i (OneHotLabel_i - RealLabelSoftMax_i)^2 \tag{2}$$

$$\mathscr{L}_{fake\_labels\_l2} = \frac{1}{n_{pixels} * n_{labels} * m_{batch}} * \sum_i (OneHotLabel_i - FakeLabelSoftMax_i)^2 \tag{3}$$

$$\mathscr{L}_{real\_labels\_cross} = \frac{1}{n_{pixels} * m_{batch}} * \sum_i -OneHotLabel_i * \log RealLabelSoftMax_i \tag{4}$$

$$\mathscr{L}_{fake\_labels\_cross} = \frac{1}{n_{pixels} * m_{batch}} * \sum_i -OneHotLabel_i * \log FakeLabelSoftMax_i \tag{5}$$

$$\begin{aligned} \mathscr{L}_{discriminator} = \alpha_{discriminator} * [& \\ & \mathscr{L}_{real\_labels\_l2} + \\ & \mathscr{L}_{real\_labels\_cross} + \\ & \max_{\mathscr{L}_{real\_labels\_l2} - \mathscr{L}_{fake\_labels\_l2} \leq \alpha_{slack}} -\mathscr{L}_{fake\_labels\_l2} + \\ & \max_{\mathscr{L}_{real\_labels\_cross} - \mathscr{L}_{fake\_labels\_cross} \leq \alpha_{slack}} -\mathscr{L}_{fake\_labels\_cross} \\ &] \end{aligned} \tag{6}$$

$$\mathscr{L}_{latent} = \frac{1}{n_{pixels} * n_{labels} * m_{batch}} * [\sum_i (OneHotLabel_i - PredictedLabelSoftMax_i)^2 + \sum_i (Latent_i - PredictedLatent_i)^2] \tag{7}$$

$$\begin{aligned} \mathscr{L}_{generator} = \alpha_{discriminator} * [& \\ & \mathscr{L}_{fake\_labels\_l2} + \\ & \mathscr{L}_{fake\_labels\_cross} + \\ &] + \alpha_{autoencoder} * \mathscr{L}_{latent} \end{aligned} \tag{8}$$

Note that there were many iterations across different loss functions and architectures. Including using boolean discriminators based on DNNs which did not appear to work as well for the diversity of images in this dataset. Using the label based loss with a maximum slack seems to work best both since it makes the architecture

symmetrical and balanced and because with the slack separate terms cannot overpower each other and stay in balance better. This seems to work better than trying to automatically adjust the scale of the loss components or change the relative frequency of update steps. That said having the loss scale hyperparameters was very useful when babysitting it to adjust the relative strength of terms. There are a lot of combined objectives and while they don't have to be perfectly balanced due to separation of training steps they do need to relatively balanced.

Note that while most of the pixel level losses are currently L2 plus cross entropy there were also experiments with other losses such as L1, losses in the frequency domain (via FFT), and cosine similiarity. All of these were done to try to increase detail but seem less effective then just training longer with the simpler losses. Frequency domain might deserve more experimentation. Other losses in some cases appeared to some benefit but eventually some deficit. For example, cosine similarity was helpful in early training before introducing the slack inequality terms but had to be removed since it inevitably created bright spots like lens flares later in training.

5 Model Evaluation

While during training looking at specific loss contributions is useful they ultimately are not reliable for determining the final quality. One example is when the discriminator is far behind the generator the generator is essentially successful in adversarial attacks against the discriminator. So, the loss looks good but the images look like noise. This case can be detected by comparing the visual output with the discriminator of the generator output. If the detected labels look good but the image looks horrible than the generator has overpowered the discriminator. This is just one example. Ultimately many problems can be diagnosed by examining intermediate outputs. For example, a grid pattern might reflect a stride problem in the architecture. Or a fine persistent noise in generated images turned out to be due to adding the latent vectors next to the embeddings instead of next to the labels.

In short, while the graphs are very useful ultimately looking at the outputs of submodels has been just as important. For example, see Figure 3 for autoencoder output, Figure 4 for labeling, and Figure 5 for generator output.

Figure 4: Labeling Examples for Evaluating Discriminator (original in the middle)

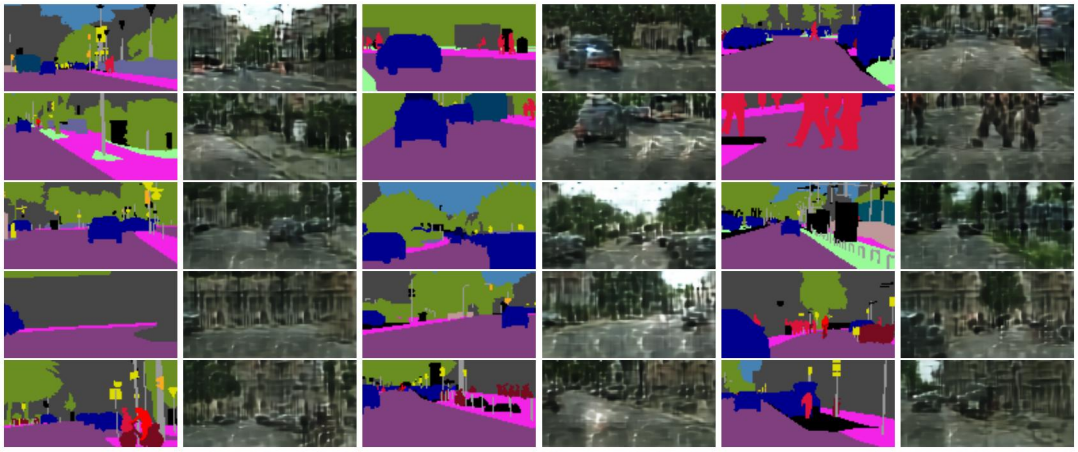

Figure 5: Generated Examples for Evaluating Generative Model (requested labels on the left)

6 Conclusion and Final Thoughts

There was not enough time to properly explore the ideas in this project. A lot was learned about the loss functions and architecture. Most notably the use a slack term for the adversarial portion of the loss along with symmetry in the architecture resolved many training stability issues albeit at the cost of slower training.

Also, it appears that at least for this domain using label based discriminator loss terms handles the diversity of the image better than a boolean discriminator which labels as an additional input. Or at least it is easier to train.

In terms of partial results, Figures 5 and 4 show some success in terms of the generative model and discriminator. It has been found that the training process can scale up the model but is very slow overall. Even though the quality of generated images still could use some improving it does seem that it was successful in being able to learn an embedding representation that can be shared between the submodels and so it should be possible to perform the editing if quality of all the submodels can be improved. Figure 3 shows promise that the model has enough complexity for higher quality images. Recent training has shown that the loss continues to decrease on the latent decoder model and while it currently does little more than correct the colors of the generated image to the colors of the original image it is promising that with further training it might further correct the generated image to the source image to be edited.

Further, although originally implemented to improve a problem with noise we believe that splitting the latent vector by label allows for the interesting possibility of potentially transfering latent vectors for a given class from one image to another. For example, introduce a car with car latent vectors from a donor image or swap out car latent vectors.

The next step would be to restart training from a very small image size such as 8x4 and be patient to not increase until the quality is really high on each progressive scale jump. This will take a long time but so far all loss curves are always still going down so it may be the only way to determine what quality can ultimately be achieved and complete the experiment.

Code

Complete source can be found at `https://github.com/loganbruns/EditGAN/`

References

[1] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. *arXiv* arXiv:1604.01685

[2] Junbo Zhao, Michael Mathieu, Yann LeCun, "Energy-based Generative Adversial Network" in ICLR 2017. *arXiv* arXiv:1609.03126

[3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, "Going Deeper with Convolutions" in CoRR abs/1409.4842, 2014. *arXiv* arXiv:1409.4842