Image-to-Image Translation in Tensorflow

Make discriminators do your work for you

Written by Christopher Hesse — January 25th, 2017

I thought that the results from pix2pix by Isola et al. looked pretty cool and wanted to implement an adversarial net, so I ported the Torch code to Tensorflow. The single-file implementation is available as pix2pix-tensorflow on github.

Here are some examples of what this thing does, from the original paper:

> *"The Sorcerer's Stone, a rock with enormous powers, such as: lead into gold, horses into gold, immortal life, giving ghosts restored bodies, frag*

> *trolls, trolls into gold, et cetera."*
>
> — Wizard People, Dear Readers

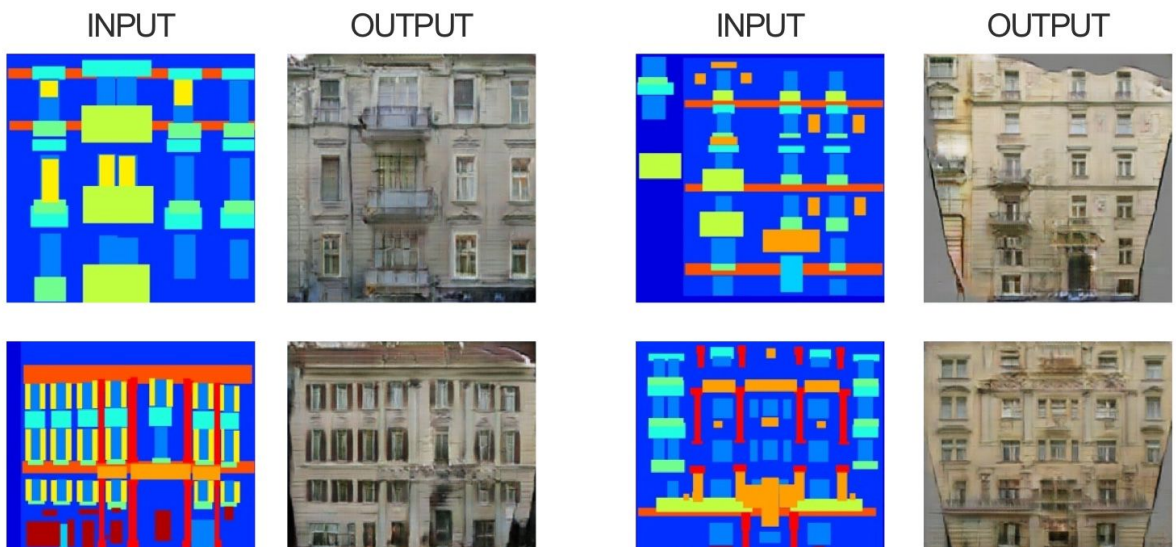
pix2pix probably will not give ghosts restored bodies, but if you have a dataset of normal horses and their corresponding golden forms, it may be able to do something with that.

Running the Code

Terminal

```
# make sure you have Tensorflow 0.12.1 installed first
python -c "import tensorflow; print(tensorflow.__version__)"

# clone the repo
git clone https://github.com/affinelayer/pix2pix-tensorflow.git
cd pix2pix-tensorflow

# download the CMP Facades dataset http://cmp.felk.cvut.cz/~tylecr1/facade/
python tools/download-dataset.py facades

# train the model
# this may take 1-9 hours depending on GPU, on CPU you will be waiting for a bit
python pix2pix.py \
  --mode train \
  --output_dir facades_train \
  --max_epochs 200 \
  --input_dir facades/train \
  --which_direction BtoA

# test the model
python pix2pix.py \
  --mode test \
  --output_dir facades_test \
  --input_dir facades/val \
  --checkpoint facades_train
```

After training this for a long while, you can expect output along the lines of:



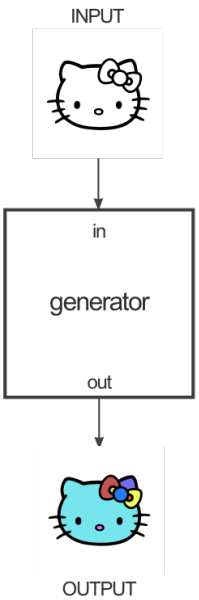
How pix2pix works

pix2pix uses a conditional generative adversarial network (cGAN) to learn a mapping from an input image to an output image.

The network is composed of two main pieces, the `Generator` and the `Discriminator`. The `Generator` applies some transform to the input image to get the output image. The `Discriminator` compares the input image to an unknown image (either a target image from the dataset or an output image from the generator) and tries to guess if this was produced by the generator.

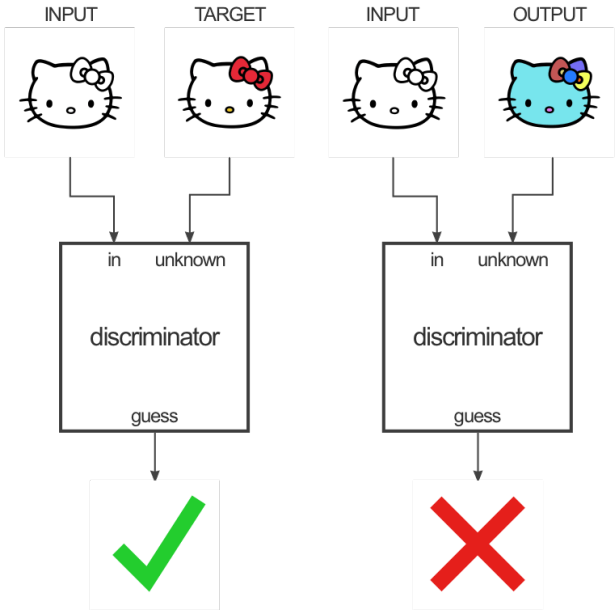
An example of a dataset would be that the input image is a black and white picture and the target image is the color version of the picture:

The generator in this case is trying to learn how to colorize a black and white image:



The discriminator is looking at the generator's colorization attempts and trying to learn to tell the

difference between the colorizations the generator provides and the true colorized target image provided in the dataset.

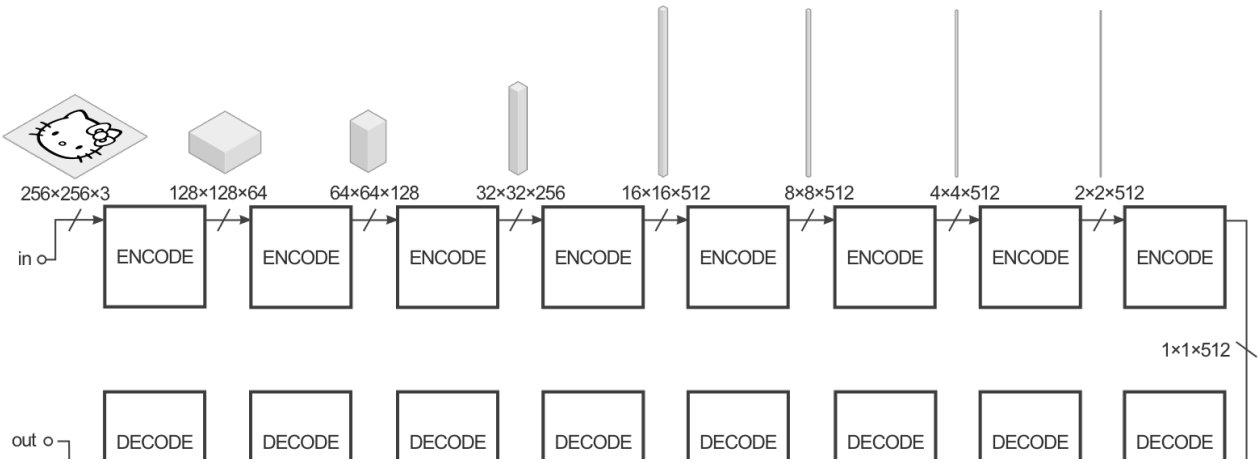


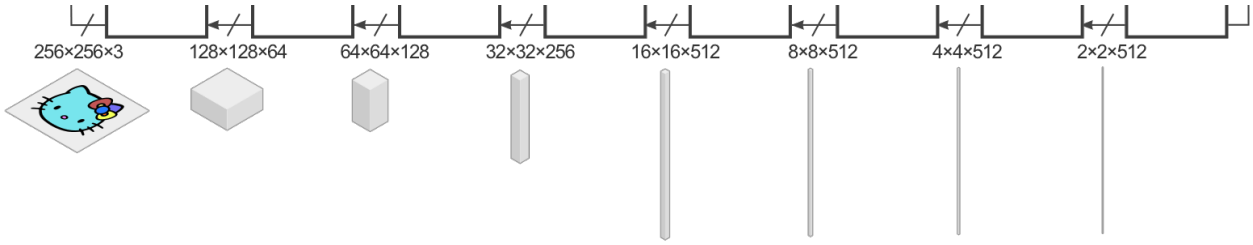
Why go through all this trouble? One of the main points of the paper is that the discriminator provides a loss function for training your generator and you didn't have to manually specify it, which is really neat. Hand-engineered transformation code has been replaced with training neural nets, so why not replace the hand-engineered loss calculations as well? If this works, you can let the computer do the work while you relax in comfort and fear that computers will replace your job.

Let's look at the two parts of the adversarial network: the `Generator` and the `Discriminator`.

The Generator

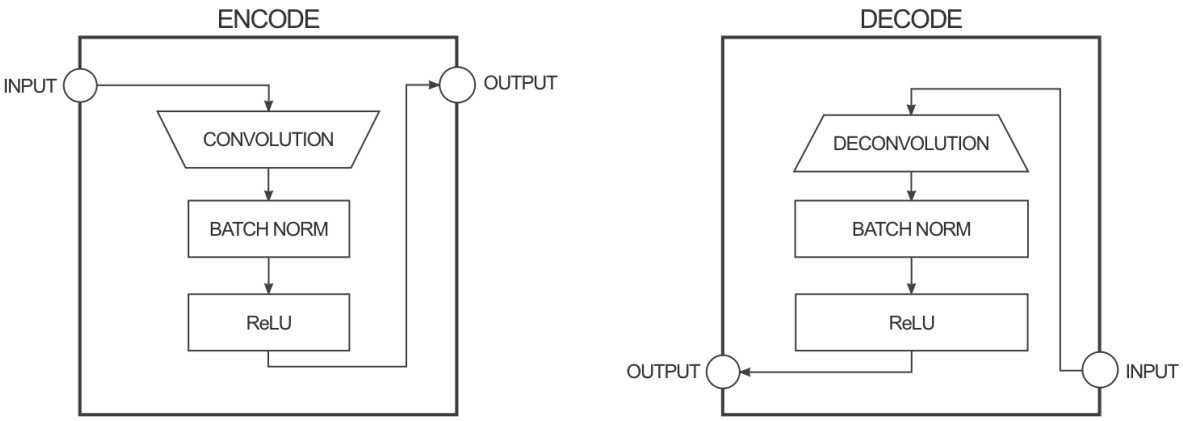
The `Generator` has the job of taking an input image and performing the transform we want in order to produce the target image. An example input would be a black and white image, and we want the output to be a colorized version of that image. The structure of the generator is called an "encoder-decoder" and in `pix2pix` the encoder-decoder looks more or less like this:



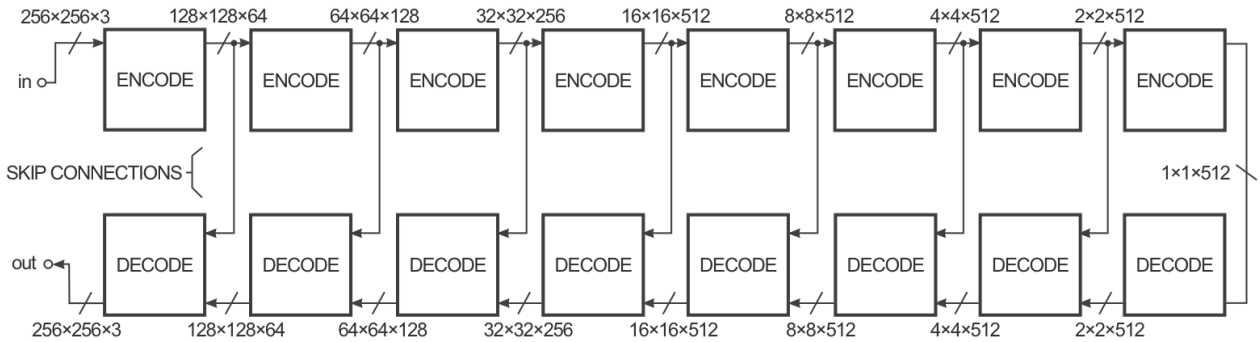


The volumes are there to give you a sense of the shape of the tensor dimensions next to them. The input in this example is a 256x256 image with 3 color channels (red, green, and blue, all equal for a black and white image), and the output is the same.

The generator takes some input and tries to reduce it with a series of encoders (convolution + activation function) into a much smaller representation. The idea is that by compressing it this way we hopefully have a higher level representation of the data after the final encode layer. The decode layers do the opposite (deconvolution + activation function) and reverse the action of the encoder layers.



In order to improve the performance of the image-to-image transform in the paper, the authors used a "U-Net" instead of an encoder-decoder. This is the same thing, but with "skip connections" directly connecting encoder layers to decoder layers:

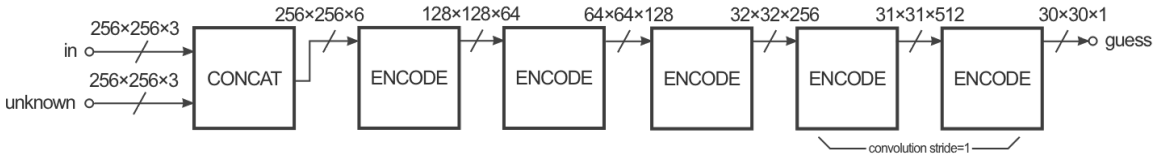


The skip connections give the network the option of bypassing the encoding/decoding part if it doesn't have a use for it.

These diagrams are a slight simplification. For instance, the first and last layers of the network have no batch norm layer and a few layers in the middle have dropout units. The colorization mode used in the paper also has a different number of channels for the input and output layers.

The Discriminator

The `Discriminator` has the job of taking two images, an input image and an unknown image (which will be either a target or output image from the generator), and deciding if the second image was produced by the generator or not.

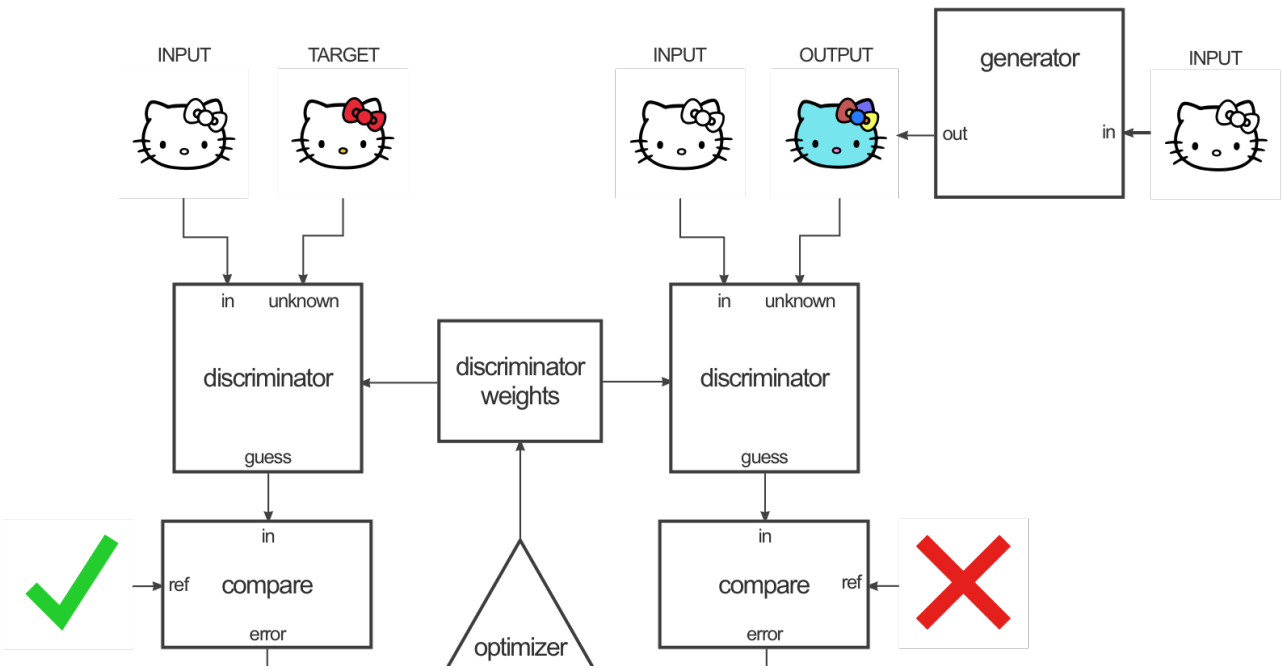


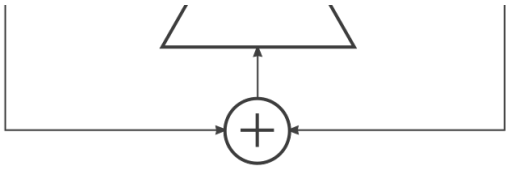
The structure looks a lot like the encoder section of the generator, but works a little differently. The output is a 30x30 image where each pixel value (0 to 1) represents how believable the corresponding section of the unknown image is. In the `pix2pix` implementation, each pixel from this 30x30 image corresponds to the believability of a 70x70 patch of the input image (the patches overlap a lot since the input images are 256x256). The architecture is called a "PatchGAN".

Training

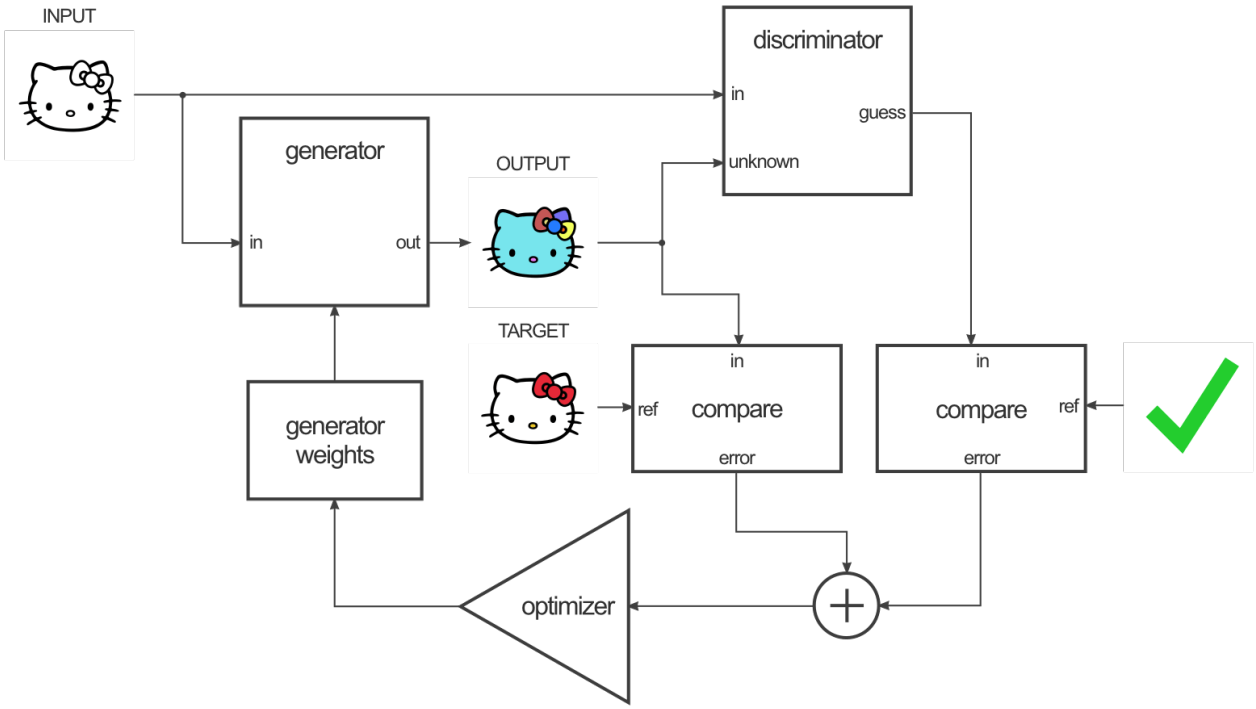
To train this network, there are two steps: training the discriminator and training the generator.

To train the discriminator, first the generator generates an output image. The discriminator looks at the input/target pair and the input/output pair and produces its guess about how realistic they look. The weights of the discriminator are then adjusted based on the classification error of the input/output pair and the input/target pair.





The generator's weights are then adjusted based on the output of the discriminator as well as the difference between the output and target image.



The clever trick here is that when you train the generator on the output of the discriminator, you're actually calculating the gradients through the discriminator, which means that while the discriminator improves, you're training the generator to beat the discriminator.

The theory is that as the discriminator gets better, so does the generator. If the discriminator is good at its job and the generator is capable of learning the correct mapping function through gradient descent, you should get generated outputs that could fool a human.

Validation

Validation of the code was performed on a Linux machine with a ~1.3 TFLOPS Nvidia GTX 750 Ti GPU. Due to a lack of compute power, validation is not extensive and only the `facades` dataset at 200 epochs was tested.

Terminal

```
git clone https://github.com/affinelayer/pix2pix-tensorflow.git
cd pix2pix-tensorflow
python tools/download-dataset.py facades
```

```
sudo nvidia-docker run \
  --volume $PWD:/prj \
  --workdir /prj \
  --env PYTHONUNBUFFERED=x \
  affinelayer/pix2pix-tensorflow \
  python pix2pix.py \
    --mode train \
    --output_dir facades_train \
    --max_epochs 200 \
    --input_dir facades/train \
    --which_direction BtoA

sudo nvidia-docker run \
  --volume $PWD:/prj \
  --workdir /prj \
  --env PYTHONUNBUFFERED=x \
  affinelayer/pix2pix-tensorflow \
  python pix2pix.py \
    --mode test \
    --output_dir facades_test \
    --input_dir facades/val \
    --checkpoint facades_train
```

For comparison, the first image in the validation set looks like this:

If you wish you can download the full results on the validation set.

Implementation

The implementation is a single file, `pix2pix.py`, that does as much as possible inside the Tensorflow graph.

The porting process was mostly a matter of looking at the existing Torch implementation as well as the Torch source code to figure out what sorts of layers and settings were being used in order to make the Tensorflow version as true to the original as possible. Debugging a broken implementation can be time consuming, so I attempted to be careful about the conversion to avoid having to do extensive debugging.

The implementation started with the creation of the generator graph, then the discriminator graph, then the training system. The generator and discriminator graph were printed at runtime using the Torch `pix2pix` code. I looked in the Torch framework source for the different layer types and found what settings and operations were present and implemented those in

Tensorflow.

Ideally I would have been able to export the `pix2pix` trained network weights into Tensorflow to verify the graph construction, but that was annoying enough, or I am bad enough at Torch, that I did not do it.

Most of the bugs in my code were related to the build-graph-then-execute model of Tensorflow which can be a little surprising when you are used to imperative code.