02514 Deep Learning in Computer Vision

Project 3

Kim Reinhardt Jensen, s164038
Michael Rahbek, s164035
Ghassen Lassoued, s196609

Technical University of Denmark
June 24, 2020

1 Exercise 3: MNIST

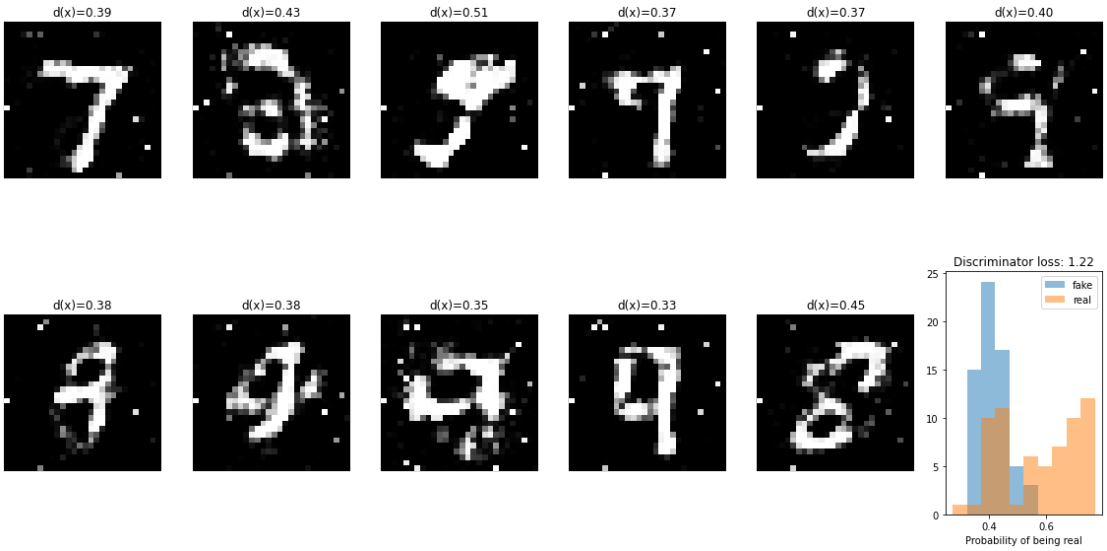

Figure 1: Generator images and histogram of the discriminator for the current minibatch.

Figure 1 shows images made by generating random numbers from a normal distribution and feeding then through the generator. The histogram is computed over the current mini-batch for both the real and the fake images when these are run through the discriminator network. It is seen that the discriminator performs reasonably well as should be the case, otherwise the networks will not be able learn. This is achieved by having a higher learning rate for the discriminator. We have used fully connected neural networks and tried using Vanilla, Wasserstein and LSGAN, which seemed to perform equally well though we did not perform any rigours comparison.

2 Dataset

The dataset is divided into a training set with 1067 images of horses and 1334 images of zebras, while the test set had 120 instances of horses and 140 instances of zebras. In our training this slight imbalance was handled by showing the network some of the horse images twice in each epoch such that all zebra images were used. Initially, we resized all the images to 128x128 and 3 color channels but for the final training, we used height and width 256 as given in the dataset.

3 CycleGAN implementation

For the CycleGAN we need to have generator and a discriminator. The code for the generator is handed out with the project and follows the architecture from the slides with 6 ResNet blocks, which all have $4 \cdot 64$ features with image dimesion $H/4, W/4$. We used this for the initial model development, but for the final training we increased the number of ResNet blocks to 9.

We constructed the discriminator using the architecture given in the slides with three 4x4 convolutional layers with padding 1 and stride 2 followed by two 4x4 convolutional layers with padding 1 and stride 1. Instance normalization and leaky ReLU with a negative slop of $\alpha = 0.2$ is used after all layers except the last one. Please refer to the code in Appendix A.1.

When using the CycleGAN framework, the input images need not to come in pairs. The network has two parts; one that takes a real horse image as input and one that takes a real

zebra image as input. Therefore, the network consists of two discriminators (the horse and the zebra discriminator) and two generators (the 'horse to zebra' and the 'zebra to horse' generator). The part of the network that takes the real horse image as input starts by running it through the horse discriminator. Then it runs the real horse through the 'horse to zebra' generator to generate a fake zebra. That fake zebra is run through the zebra discriminator and also through the 'zebra to horse' generator to generate a reconstructed horse. The same course of action is followed by the part of the network that takes the real zebra image as input. Additionally, each real image is run through the 'opposite' generator to get the identity.

The procedure described above is followed in the training and the code is included in Appendix A.2. We used a batch size of 6 without any memory issues. As recommended by the course material, we used the Adam optimizer with $\beta_1 = 0.5$ and a learning rate for the discriminators 0.0002 and a learning rate of the generators of 0.0001. We initially tried to train with a learning rate of the discriminator of 0.0004, but this leaning rate was overshooting. To update the discriminators and the generators, we need to calculate a loss. We used the l_1 loss for the identity and cycle loss and for the LSGAN loss we use the values $a = 0$, $b = 1$ and $c = 1$. The discriminator and generator loss when the input consists of horse images ($\mathbf{x}_h$) is given below. The losses related to zebra images ($\mathbf{x}_z$) are equivalent, but not shown here.

$$\mathcal{L}_G(\mathbf{x}_h) = (D_z(G_{h2z}(\mathbf{x}_h)) - c)^2 + \lambda_{cyc} l_1(G_{z2h}(G_{h2z}(\mathbf{x}_h)), \mathbf{x}_h) + \lambda_{id} l_1(G_{z2h}(\mathbf{x}_h), \mathbf{x}_h)$$
$$\mathcal{L}_D(\mathbf{x}_h) = (D_h(\mathbf{x}_h) - b)^2 + (D_z(G_{h2z}(\mathbf{x}_h)) - a)^2$$

Please note, that when we apply the discriminator to the transformed image in the discriminator loss, we use `.detach()` on the fake image such that this loss does not affect the generator. The parameters related to the regularisation were $\lambda_{cycle} = 10$ and $\lambda_{identity} = 5$. It is seen that the discriminator loss is low, when the discriminator is able to distinguish between original and the transformed images, while the generator loss is low when the generator is able to fool the discriminator. Therefore, these losses work in the opposite directions, which also makes it difficult to tell when the networks have converged as the discriminator and generator losses cannot be low at the same time. The added penalty terms to the generator loss ensures that the generator makes images that look similar to real images. If they were not present the generator could for instance make stripes all across the images in an effort to fool the discriminator.

The downside of the cycle loss is that it forces the generator to not remove characteristics from the original image as this makes it impossible to recover the original image. Additionally, the cycle loss forces the generators learn almost opposite transformations such that applying both results in the original image. The identity loss ensures that the generators only make minimal changes if the desired animal is already in the image. The zebra generator may, however, highlight lines in zebras and still get a low identity loss. Similarly, the horse generator may often patch brown on top of horse and get a low identity loss.

4 CycleGAN performance

To ensure our model is working we save the different losses to see that they improve over training. We store the different parts of the discriminator and generator losses as well as the Frechet Inception Distance (FID) computed on the test set. These are shown in figure 2. The FID measures the similarity between two distributions of images using the Inception V3 network where a score of zero means that the distributions are identical. It was implemented by removing the last, fully connected layer of Inception, then running all of the test images through the network and comparing the mean and covariances of the 2048 resulting features. This gave a score for both the zebra images and the horse images. As this is computed using the test set it may also be used to quantify the performance of the model.

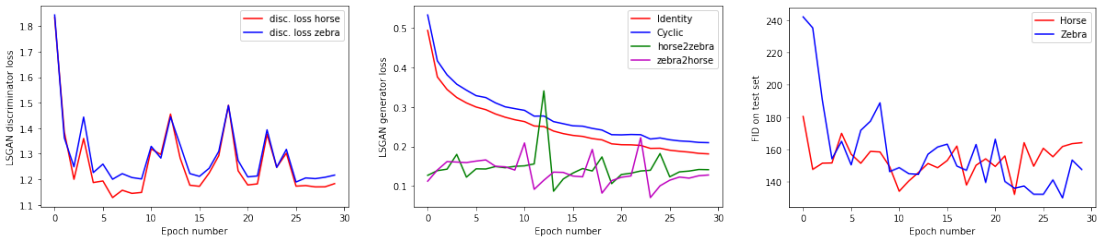

Figure 2: Discriminator and generator loss as well as the FID

Figure 3: The output after 7 epochs on the left and 18 epochs on the right.

Figure 3 shows a few outputs during training. As we use patchGAN, we feed the networks with some patches that only contain the background, which also causes the models to learn the typical differences in background between the horse and zebra images. The horse generator seems to learn the typical dark brown colour of the horses and the green colour of the background quite quickly. Similarly, the zebra generator learns the more barren background though it seems to have a harder time learning the stripes of the zebras. This could be because the stripes have different sizes depending on the amount of zoom in the image while the solid colours are more invariant to scaling. Additionally, we enforce identity and cycle consistency using the l_1-loss which is sensitive to exact pixel values. For the cycle consistency, one could imagine that the horse to zebra generator creates the exact same stripes as in the original zebra but with black and white interchanged, which would result in a large loss.

The zebra generator also seems to start by highlighting the stripes on the horses that are already there and then later on it adds new stripes by itself. Because of the cycle regularisation, the stripes are never completely removed from the zebras but a brown color is patched on top of the zebra. This way, the generator may recover the original location of the stripes. This also causes the recovered horses to have a pattern of stripes that was not present in the original image as seen in figure 4. In contrast the recovered zebras seem to carry fewer artifacts from the transformation as the colouration is more easily accounted for. It does, however, never generate impressive horses as the cycle loss force it to retain the location of the stripes.

In figure 5 some of the less successful transformations from the trained networks are shown. It is seen that the 'horse to zebra' generator reacts to the colours of the horses as stripes has been added the brown horse in the leftmost image while the black horse has been left untouched. Also, the black and white image of the horses is almost unchanged after the transformation though the sharp gradients in the photo has been highlighted which corresponds to emphasising the stripes of a zebra. If a better performance on grey images is desired, the network could be supplied with a black and white version of every image, which would be a form of data augmentation.

Figure 4: The original, transformed, recovered and identity image after training for 30 epochs.

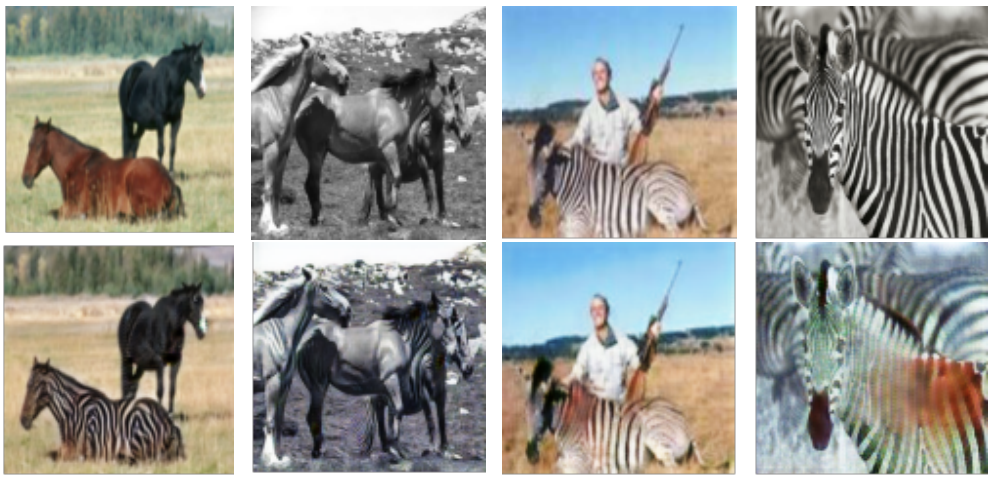

Figure 5: Less successful transformation after the networks have been trained for 30 epochs.

The 'zebra to horse' generator is seen to perform better on the grey images though it sometimes patches the zebra with green instead of brown as it has learned the background of the horses which typically contain grass. In the second image from the right, it is also seen to handle the presence of humans quite well, though it does relatively little to transform the zebra. In general, we have observed both generators to handle the presence of other objects quite well though we occasionally get a zebra-striped rock or bush.

We have not applied data augmentation though it would be most natural to use it immediately before the images are fed to the discriminator as this is more likely to overfit than the generator. Then we would need to apply it to both the real and the fake images.

5 CycleGAN with image buffer

To improve the performance, we tried to train the discriminator with images generated from previous versions of the generators in addition to the current ones. Hence one generated image from each of the last 50 mini batches is kept in a buffer. In accordance with the paper Ashish Shrivastava et al. 2017, an equal number of current images and old images is put into the discriminator. We then draw 6 random images from the buffer at put through the discriminator along with 6 newly generated images. The idea is that instead of the discriminator only looking at the transformations of the current stage of the generator, it also looks at previous transformations that it should also take into account. The results are shown below in figure 6 and 7. The visual results are very similar to the ones for the model without buffered images. The FID value is also similar, but the discriminator loss now is smaller than for the model without buffering. This is in accordance with the idea, that the discriminator is forced remember all features. Because the discriminator performs better, we see that the 'horse to zebra' and 'zebra to horse' generators loss is greater in this case, as it gets harder to fool the discriminator. From figure 6, the discriminator and generator losses are seen to be interconnected though the identity and cyclic loss may be improved without increasing the discriminator loss and is thus seem to decrease continuously.

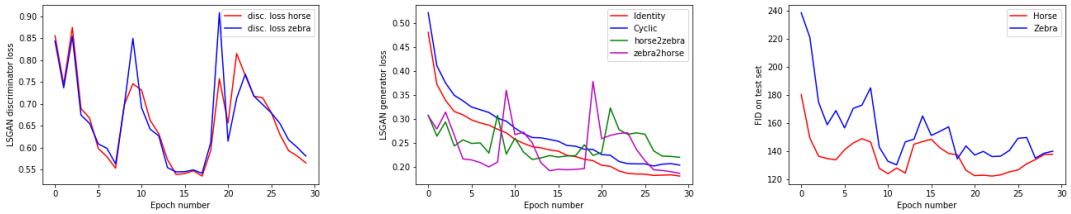

Figure 6: Discriminator and generator loss and FID with buffered images

Figure 7: Original, transformed, recovered, identity image after 30 epochs with buffered images.

References

Ashish Shrivastava Tomas Pfister, Oncel Tuzel et al. (2017). "Learning from Simulated and Unsupervised Images through Adversarial Training". In: *Apple Inc.*

Diary

We have all been present from morning to evening every day from Monday the 22th to Wednesday the 24th and worked together on all parts on the report.

A Appendix

A.1 Discriminator

```
class Discriminator(nn.Module):
  def __init__(self):
    super(Discriminator, self).__init__()
    self.conv = nn.Sequential(
      # three 4x4 convs, stride 2, padding 1, HW reduction
      nn.Conv2d(3,   64,4,   stride=2, padding=1), norm_layer(64),
                nn.LeakyReLU(negative_slope=0.2,inplace=True),
      nn.Conv2d(64, 128,4, stride=2, padding=1), norm_layer(128),
                nn.LeakyReLU(negative_slope=0.2,inplace=True),
      nn.Conv2d(128,256,4,stride=2, padding=1), norm_layer(256),
                nn.LeakyReLU(negative_slope=0.2,inplace=True),

      # Two 4x4 convs, stide 1, padding 1, reduce by 1 pixel
      nn.Conv2d(256,512,4,stride=1, padding=1), norm_layer(512),
                nn.LeakyReLU(negative_slope=0.2,inplace=True),
      nn.Conv2d(512,1,4,stride=1, padding=1)
    )

  def forward(self, x):

    x = self.conv(x)

    return x
```

A.2 Training

```
#Initialize networks
d_horse = Discriminator().to(device)
d_zebra = Discriminator().to(device)
g_horse2zebra = Generator().to(device)
g_zebra2horse = Generator().to(device)

# Get parameters from the models
d_params = list(d_horse.parameters()) + list(d_zebra.parameters())
g_params = list(g_horse2zebra.parameters()) + list(g_zebra2horse.parameters())

# Initialise optimiser
d_opt = torch.optim.Adam(d_params, 0.0002, (0.5, 0.999))
g_opt = torch.optim.Adam(g_params, 0.0001, (0.5, 0.999))

discriminator_final_layer = torch.sigmoid

# Weights of the cycle and identity loss
lambda_cycle = 10
lambda_identity = 5
```

```
im_criterion = nn.L1Loss()

# LSGAN loss, a = 0, b = 1, c = 1
def disc_true_loss(x_real, b=1):
  return torch.mean((x_real - b)**2)

def disc_false_loss(x_fake, a=0):
  return torch.mean((x_fake - a)**2)

def gen_loss(x_fake, c=1):
  return torch.mean((x_fake - c)**2)

# Dictionary with losses
loss_dict = {'d_horse': [], 'd_zebra': [], 'ident': [], 'cyc': [], 'g_h2z': [],
             'g_z2h': [], 'FID_h':[], 'FID_z':[], 'd_h_real_acc': [],
             'd_h_fake_acc': [], 'd_z_real_acc': [], 'd_z_fake_acc': []}

### Training using buffer of old images ###
# Buffer tensor
buf_horse = torch.rand(50,3,size,size)*2-1
buf_zebra = torch.rand(50,3,size,size)*2-1
buf_horse = buf_horse.to(device)
buf_zebra = buf_zebra.to(device)
buf_index = 0

for epoch in range(num_epochs):
print('Epoch: ', epoch, ' of ', num_epochs)
# iter() creates a new instance of the data loader for horses
horse_iter = iter(train_loader_horse)

# Losses to be stored in dict
d_horse_lst, d_zebra_lst, ident_lst, cyc_lst, g_h2z_lst, g_z2h_lst = [],
    [], [], [], [], []
d_h_real_acc, d_h_fake_acc, d_z_real_acc, d_z_fake_acc = [], [], [], []

for minibatch_no, zebra in enumerate(train_loader_zebra):
  # Try to fetch the next batch of horses. If it fails that means we have
  # run through all of the training examples so we create a new iterator
  try:
    horse = next(horse_iter)
  except StopIteration:
    horse_iter = iter(train_loader_horse)
    horse = next(horse_iter)
    horse = next(iter(train_loader_horse))

  zebra_real = zebra.to(device)*2-1 #scale to (-1, 1) range
  horse_real = horse.to(device)*2-1 #scale to (-1, 1) range

  # Setting gradients to zero
  d_opt.zero_grad()
  g_opt.zero_grad()
  d_loss = 0
  g_loss = 0

  ''' Network with real horse input '''

  # Calculating discriminator of real horse image
  out_horse_real = d_horse(horse_real)

  # Generating fake zebra image
  zebra_fake = g_horse2zebra(horse_real)
```

```
# Calculating fake zebra image contribution generator loss
out_zebra_fake = d_zebra(zebra_fake)
g_h2z_loss_tmp = gen_loss(out_zebra_fake)
g_loss += g_h2z_loss_tmp

# Calculating discriminator of fake zebra image
out_zebra_fake = d_zebra(zebra_fake.detach())
out_zebra_buff = d_zebra(buf_zebra[[randint(0,49),randint(0,49),randint(0,49),
                 randint(0,49),randint(0,49),randint(0,49)]])

# Discriminator loss related to the real horse image and fake zebra image
d_horse_loss_tmp = (disc_true_loss(out_horse_real) +
  disc_false_loss(out_zebra_fake) + disc_false_loss(out_zebra_buff))
d_loss += d_horse_loss_tmp

# Generating recunstructed horse image
horse_rec = g_zebra2horse(zebra_fake)

# Identity of the horse
horse_id  = g_zebra2horse(horse_real)

# ImLoss using the L1 loss
horse_id_loss   = im_criterion(horse_id, horse_real)
horse_cyc_loss  = im_criterion(horse_rec, horse_real)

''' Network with real zebra input '''

# Calculating distributer of real zebra image
out_zebra_real = d_zebra(zebra_real)

# Generating fake horse image
horse_fake = g_zebra2horse(zebra_real)

# Calculating fake horse image contribution to the generator loss
out_horse_fake = d_horse(horse_fake)
g_z2h_loss_tmp = gen_loss(out_horse_fake)
g_loss += g_z2h_loss_tmp

# Calculating discriminator of fake horse image
out_horse_fake = d_horse(horse_fake.detach())
out_horse_buff = d_horse(buf_horse[[randint(0,49),randint(0,49),randint(0,49),
                 randint(0,49),randint(0,49),randint(0,49)]])

# Discriminator loss related to the real zebra image and fake horse image
d_zebra_loss_tmp = (disc_true_loss(out_zebra_real) +
  disc_false_loss(out_horse_fake) + disc_false_loss(out_horse_buff))
d_loss += d_zebra_loss_tmp

# Generating recunstructed zebra image
zebra_rec = g_horse2zebra(horse_fake)

# Identity of the zebra
zebra_id  = g_horse2zebra(zebra_real)

# ImLoss using the L1 loss
zebra_id_loss   = im_criterion(zebra_id, zebra_real)
zebra_cyc_loss  = im_criterion(zebra_rec, zebra_real)

# Add the L1 losses to the generator loss
g_loss += (lambda_identity*(zebra_id_loss + horse_id_loss))
g_loss += (lambda_cycle*(zebra_cyc_loss + horse_cyc_loss))
```

```
# Losses for the dictionary
d_horse_lst.append(d_horse_loss_tmp.item())
d_zebra_lst.append(d_zebra_loss_tmp.item())
ident_lst.append((zebra_id_loss + horse_id_loss).item())
cyc_lst.append((zebra_cyc_loss + horse_cyc_loss).item())
g_h2z_lst.append(g_h2z_loss_tmp.item())
g_z2h_lst.append(g_z2h_loss_tmp.item())

# Calculate the discriminator accuracy
d_h_fake_numel = 1/out_zebra_fake.numel()
d_h_real_numel = 1/out_horse_real.numel()
d_h_fake_acc.append(d_h_fake_numel*(discriminator_final_layer(out_zebra_fake)
  <0.5).sum().cpu())
d_h_real_acc.append(d_h_real_numel*(discriminator_final_layer(out_horse_real)
  >0.5).sum().cpu())
d_z_fake_numel = 1/out_horse_fake.numel()
d_z_real_numel = 1/out_zebra_real.numel()
d_z_fake_acc.append(d_z_fake_numel*(discriminator_final_layer(out_horse_fake)
  <0.5).sum().cpu())
d_z_real_acc.append(d_z_real_numel*(discriminator_final_layer(out_zebra_real)
  >0.5).sum().cpu())

''' Update discriminators and generators '''
d_loss.backward()
g_loss.backward()

d_opt.step()
g_opt.step()

# Updating buffer tensor
buf_index = buf_index % 50
buf_horse[buf_index] = horse_fake[0].detach()
buf_zebra[buf_index] = zebra_fake[0].detach()
buf_index += 1

#Plot results every 100 minibatches
if minibatch_no % 100 == 0:
  real_horses = horse_real.cpu().detach().numpy()/2+.5
  fake_zebras = zebra_fake.cpu().detach().numpy()/2+.5
  real_zebras = zebra_real.cpu().detach().numpy()/2+.5
  fake_horses = horse_fake.cpu().detach().numpy()/2+.5

  display.clear_output(wait=True)
  plt.figure(figsize=(15,15))
  for k in range(batch_size):
    plt.subplot(4,batch_size,k+1)
    plt.imshow(np.swapaxes(np.swapaxes(real_horses[k], 0, 2), 0, 1))
    plt.axis('off')
    plt.subplot(4,batch_size,k+1+batch_size)
    plt.imshow(np.swapaxes(np.swapaxes(fake_zebras[k], 0, 2), 0, 1))
    plt.axis('off')
    plt.subplot(4,batch_size,k+1+2*batch_size)
    plt.imshow(np.swapaxes(np.swapaxes(real_zebras[k], 0, 2), 0, 1))
    plt.axis('off')
    plt.subplot(4,batch_size,k+1+3*batch_size)
    plt.imshow(np.swapaxes(np.swapaxes(fake_horses[k], 0, 2), 0, 1))
    plt.axis('off')
  title = 'Epoch {e} - minibatch {n}/{d}'.format(e=epoch+1, n=minibatch_no,
      d=len(train_loader_zebra))
  plt.suptitle(title, fontsize=20)
  plt.show()
```

```
# Compute FID on the test set
FID_horse, FID_zebra = FID(g_horse2zebra, g_zebra2horse, test_loader_horse,
    test_loader_zebra)

# Update the dictionary at the end of every epoch
loss_dict['d_horse'].append(np.mean(d_horse_lst))
loss_dict['d_zebra'].append(np.mean(d_zebra_lst))
loss_dict['ident'].append(np.mean(ident_lst))
loss_dict['cyc'].append(np.mean(cyc_lst))
loss_dict['g_h2z'].append(np.mean(g_h2z_lst))
loss_dict['g_z2h'].append(np.mean(g_z2h_lst))
loss_dict['FID_h'].append(FID_horse)
loss_dict['FID_z'].append(FID_zebra)
loss_dict['d_h_real_acc'].append(np.mean(d_h_real_acc))
loss_dict['d_h_fake_acc'].append(np.mean(d_h_fake_acc))
loss_dict['d_z_real_acc'].append(np.mean(d_z_real_acc))
loss_dict['d_z_fake_acc'].append(np.mean(d_z_fake_acc))
```


Make a zebra
Make a coarse grained image
GENERATOR
imgflip.com
JAKE-CLARK.TUMBLR