# Homework 4 — CompSci 389 — University of Massachusetts — Spring 2022

Assigned: April 1, 2024; Due: April 9, 2024 @ 11:59 PM EST

## Instructions!

In this assignment, you'll be using PyTorch again. If you need to install PyTorch again, you can find instructions to install it [here](). Some Windows users have issue using pip to install it so I recommend in that case to use [anaconda]().

This time, we'll be using PyTorch to implement two types of neural network -- these are pretty cool.

The first of these is going to be an *autoencoder*. An autoencoder is a neural network with a pretty unique structure, which will learn a function that can map an input to itself. This allows the network to extract important features from an input to effectively compress it, and then reconstruct those important features back into something that approximates the original input closely.

The second of these is going to be a *GAN (generative adversarial network)*, which is a framework where we train two neural networks - one learns to generate synthetic data based on training data (the generator), and the other learns to distinguish true data from generated data (the discriminator). This is where the word *adversarial* comes into the picture -- as the discriminator gets better at identifying fake data, the feedback is passed to the generator, which gets better at creating synthetic data.

One example of a cutting edge GAN that you can quickly check out is [This Person Does Not Exist](), which generates synthetic portraits of people!

```
# Step 1: Let's import some libraries!
import time
import torch
import torch.nn as nn
import torchvision
import numpy as np
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, random_split, Subset
from torchvision import datasets
from torchvision.datasets import ImageFolder
import torchvision.transforms as tt
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
from matplotlib.image import imread
import os
import random
from tqdm import tqdm


# This is a new dependency to load this dataset, so make sure you install this mc
from PIL import Image
```

## Our Dataset

For this project we're going to use **human faces**! Well, *images* of human faces, but it's still cool.

The particular dataset is calleb **CelebA** -- you can find the official website [here]() and you can find the torchvision documentation [here]()

```
def load_celebA(batch_size=32, train=True):
    '''
    Dataset loading will be handled for you automatically since it's a bit of a p
    to work with these large datasets and I'll just give you a subset
    '''

    dataset = []
    batch_counter = 0
    batch = []

    for file in tqdm(os.listdir('./celebA')):

        img = Image.open('./celebA/' + file)
        img = np.asarray(img).reshape(3, 109, 89)

        if batch_counter < batch_size:
            batch.append(img)
            batch_counter += 1
        else:
            dataset.append(np.array(batch))
            batch = []
            batch_counter = 0

    return np.array(dataset)
```

## Now let's see what our data looks like!

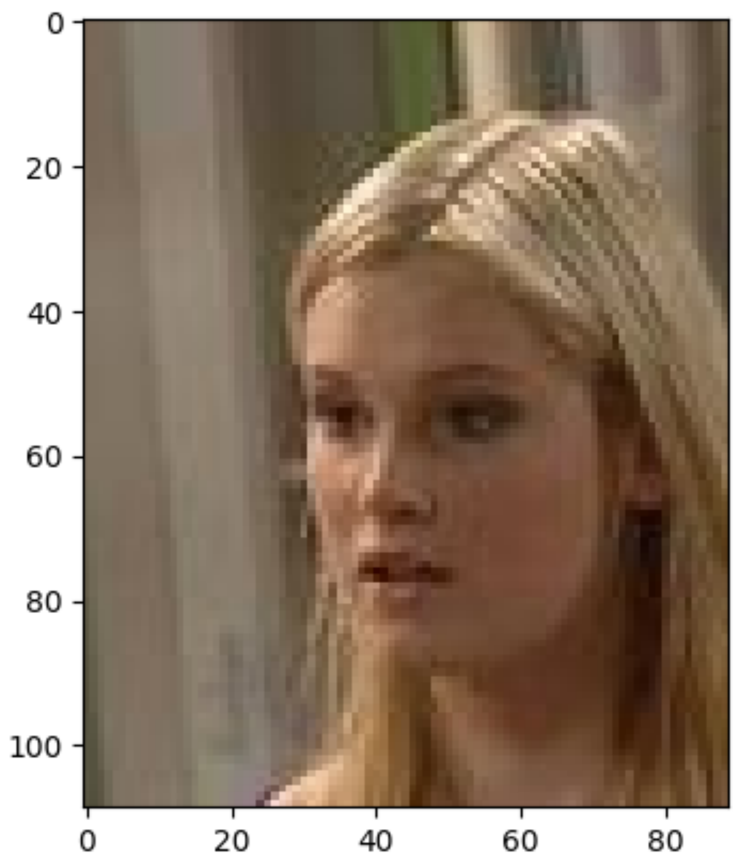
```
def plot_image(image):

    '''
    Takes in an image and shows it using matplotlib
    this is used to visualize the data and also the outputs of our network
    '''

    image = image.reshape(-1, 109,89,3)

    plt.imshow(image[0])
    return
```

```
# This will load the dataset and set the dataset variable to it
# Try to only run this code once since it takes a while (though you may again to

dataset = load_celebA(batch_size=128, train=True)
dataset = torch.from_numpy(dataset)
```

```
0%|          | 0/10000 [00:00<?, ?it/s]100%|██████████| 10000/10000 [00:02<
```

```
# This just displays a random image from the dataset
ex_image = dataset[random.randint(0,100)]
print("image shape:", ex_image.shape)

plot_image(ex_image)
```

```
image shape: torch.Size([128, 3, 109, 89])
```



## Autoencoder (40 points)

Just like we did in HW3, we're doing to build a PyTorch `nn.Module` for our autoencoder. Again, this consists of 2 parts: The initialization (defined in `__init__()` -- note that this the python convention for initalizing classes) and the forward pass (defined aptly as `forward()` )

Since we're using PyTorch, we can simply define this module and then the gradient can be found *automatically*.

Documentation for a pytorch module can be found here

Adam's little comment: guessing we'll want to put like a description of how to set up layers for the autoencoder here, depends on how much of a bottleneck we want. I put some starter code below for an autoencoder on MNIST from this tutorial (minor changes for code readability): https://medium.com/pytorch/implementing-an-autoencoder-in-pytorch-19baa22647d1

Leaving this description blank for now since we haven't covered autoencoder in class...

In our first model we will just be creating a perceptron which will use a single `nn.Linear()` module -- you can find documentation for that here

In later models we'll use nonlinearities (and that neat convolution thing) -- documentation `for nn.ReLU()` can be found here

## Autoencoder Model (10 points)

Here you are going to make the module for your encoder and decoder -- the encoder will take an image as an input and compress it to some size (its a hyperparameter) and then the decoder will take that compressed (latent) representation of the image and try to reconstruct the original

```
class Encoder(nn.Module):

    '''
    This will be the module for your encoder half of your autoencoder
    You may use Linear layers, conv2d layers and anything else you'd like (you ju

    One thing that might throw you is that input_shape is going to be a tuple whi
    entire shape of the input (i.e. an image is (3,109,89) in CelebA)
    ### TODO celebB change here
```

```
    '''

    def __init__(self, input_shape, compression_size):
        super().__init__()


        # TODO initialize the Encoder network
        # You must use nn.Conv2d, nn.Linear, and nn.ReLU but you can look for any

        #####################################################
        self.conv1 =nn.Conv2d(3,10,3)
        self.lin1 = nn.Linear((input_shape[1] - 2)*(input_shape[2]-2)*10 ,1000)
        self.lin2 = nn.Linear(1000,compression_size)
        self.rel = nn.ReLU()









        #####################################################

        self.flatten = nn.Flatten()


    def forward(self, features):

        features = features.view(-1, 3, 109, 89).float()

        #####################################################

        out = self.conv1(features)

        out = self.flatten(out)

        out = self.lin1(out)

        out = self.lin2(out)

        out = self.rel(out)
```

```
        ####################################################

        return out


encoder = Encoder((3,109,89), 100)
test_out = encoder(ex_image)

print(test_out.shape)
print("the shape of the output should be a vector of size batch_size,100, is it?"
```

```
torch.Size([128, 100])
the shape of the output should be a vector of size batch_size,100, is it?
```

```
class Decoder(nn.Module):
    '''
    This is the other bit of the autoencoder
    Likewise you can use whatever you'd like to get from the output of the encode
    (which should be a vector )
    '''

    def __init__(self, input_size, output_shape):
        super().__init__()



        self.output_shape = output_shape

        # TODO initialize your Decoder
        # You can use Linear or conv layers as you'd like, but since we are expar
        # You may want to look into deconvolution, which is called nn.Conv2Transp

        ###############################################
        print(input_size)
        self.lin1 = nn.Linear(input_size,1000)
        self.lin2= nn.Linear(1000,(output_shape[1] - 2)*(output_shape[2]-2)*10)
        self.shape = [-1,10,output_shape[1]-2,output_shape[2]-2]
        self.conv = nn.ConvTranspose2d(10,output_shape[0],3)
```

```
        ###############################################

    def forward(self, x):

        # TODO finish the forward pass of your Decoder
        # Input is the output of the encoder

        ###############################################
        out= self.lin1(x)
        out = self.lin2(out)
        out = out.reshape(self.shape)
        out = self.conv(out)



        ###############################################

        return out.to('cpu')


decoder = Decoder(100, (3,109,89))
test_out = decoder(torch.from_numpy(np.ones((32,100))).float())

print(test_out.shape)
print("the shape of the output should be shape (batch_size,3,109,89), is it?")
```

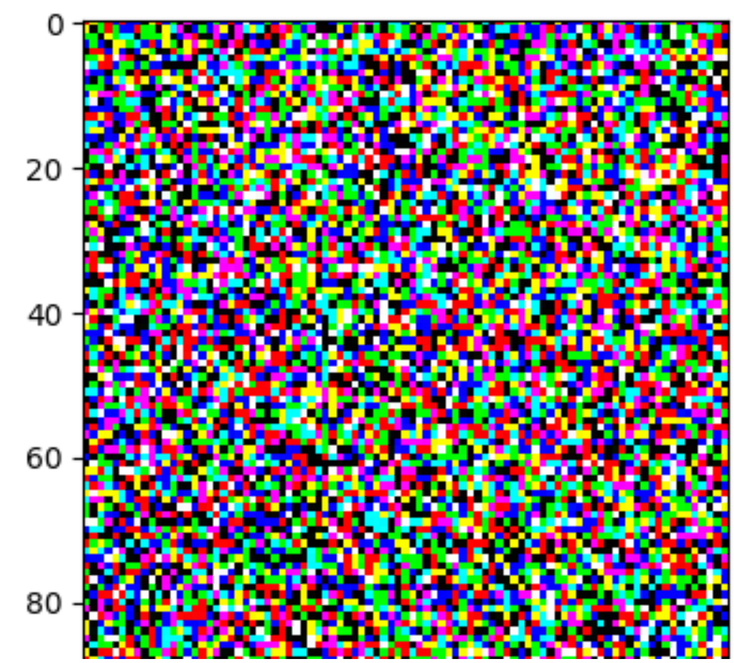
```
100
torch.Size([32, 3, 109, 89])
the shape of the output should be shape (batch_size,3,109,89), is it?
```

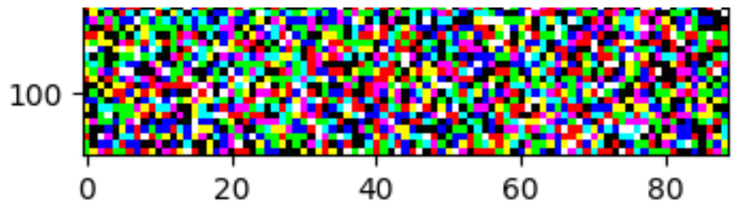
```
class Autoencoder(nn.Module):

    '''
    This will combine your encoder and decoder modules together
    with their powers combined they make an AUTOENCODER

    You may have issue with the shape of the input to the decoder
    remember that we pass in compression_size which will just be an int
    '''

    def __init__(self, input_shape, compression_size):
        super().__init__()

        self.input_size = input_shape

        self.encoder = Encoder(input_shape, compression_size)
        self.decoder = Decoder(compression_size, input_shape)
        self.relu = nn.ReLU()
        self.to('mps')



    def forward(self, features):
        features = features.to('mps')
        out = self.encoder(features)

        out = self.relu(out)

        out = self.decoder(out)


        return out.to('cpu')


# Shows the prediction of the autoencoder without training
# Not very good huh? (though theres a small chance it is lol)

input_shape = (3,109,89)

test_model = Autoencoder(input_shape, 100) # Takes input of celebA image size anc
test_output = test_model(ex_image)

print("The original image")
plot_image(ex_image.byte())
```

```
plt.show()
print("Your reconstruction")
plot_image(test_output.detach().byte())
```

```
100
The original image
```

```
Your reconstruction
```






## Loss and Optimizer for Autoencoder (5 points)

The loss for our autoencoder is nice and easy since we can just compare the input and output directly -- MSE, MAE or any other loss you'd like would work, though you can try multiple to see how they behave (or make your own if you're a nerd)

Likewise we can use

```
## Fill in the loss_function and optimizer below and run this cell to see if they

model = Autoencoder(input_shape, 100)
ex_image = ex_image.float().view(-1,3,109,89)

# TODO fill out the loss_function and optimizer

#############################################

loss_function = torch.nn.L1Loss()
optimizer =  torch.optim.SGD(model.parameters(),lr = 0.0001)

#############################################

# This checks that your model, loss and optimizer are valid
print("BEFORE GRADIENT STEP:")
ex_pred = model(ex_image)
ex_label = ex_image


optimizer.zero_grad() # Sets the gradient to 0 so that gradients don't stack toge

ex_loss1 = loss_function(ex_pred, ex_label)
print("loss",ex_loss1.item())

ex_loss1.backward() # This gets the gradient of the loss function w.r.t all of yo

print()
```

```
print("AFTER GRADIENT STEP:")
optimizer.step() # This takes the step to train

ex_pred = model(ex_image)
ex_label = ex_image

ex_loss2 = loss_function(ex_pred, ex_label)
print("loss",ex_loss2.item())

print()
print("Difference in loss:", (ex_loss1 - ex_loss2).item())
print("This should be some positive number to say we reduced loss")
```

```
100
BEFORE GRADIENT STEP:
loss 111.01061248779297

AFTER GRADIENT STEP:
loss 110.9880599975586

Difference in loss: 0.022552490234375
This should be some positive number to say we reduced loss
```

## Training Loop (10 points)

We're ready to train our autoencoder! Complete the `training()` function, just like in HW3. You can iterate over your data for 30 epochs to start.

Hint for reseting the optimizer

Hint for stepping with the optimizer (You'll have to use .backward() to get the gradient)

At this point you should record your training and validation *losses* and *accuracies* **(four lists in total)**. You'll need these values for the written section, where you will be plotting them.

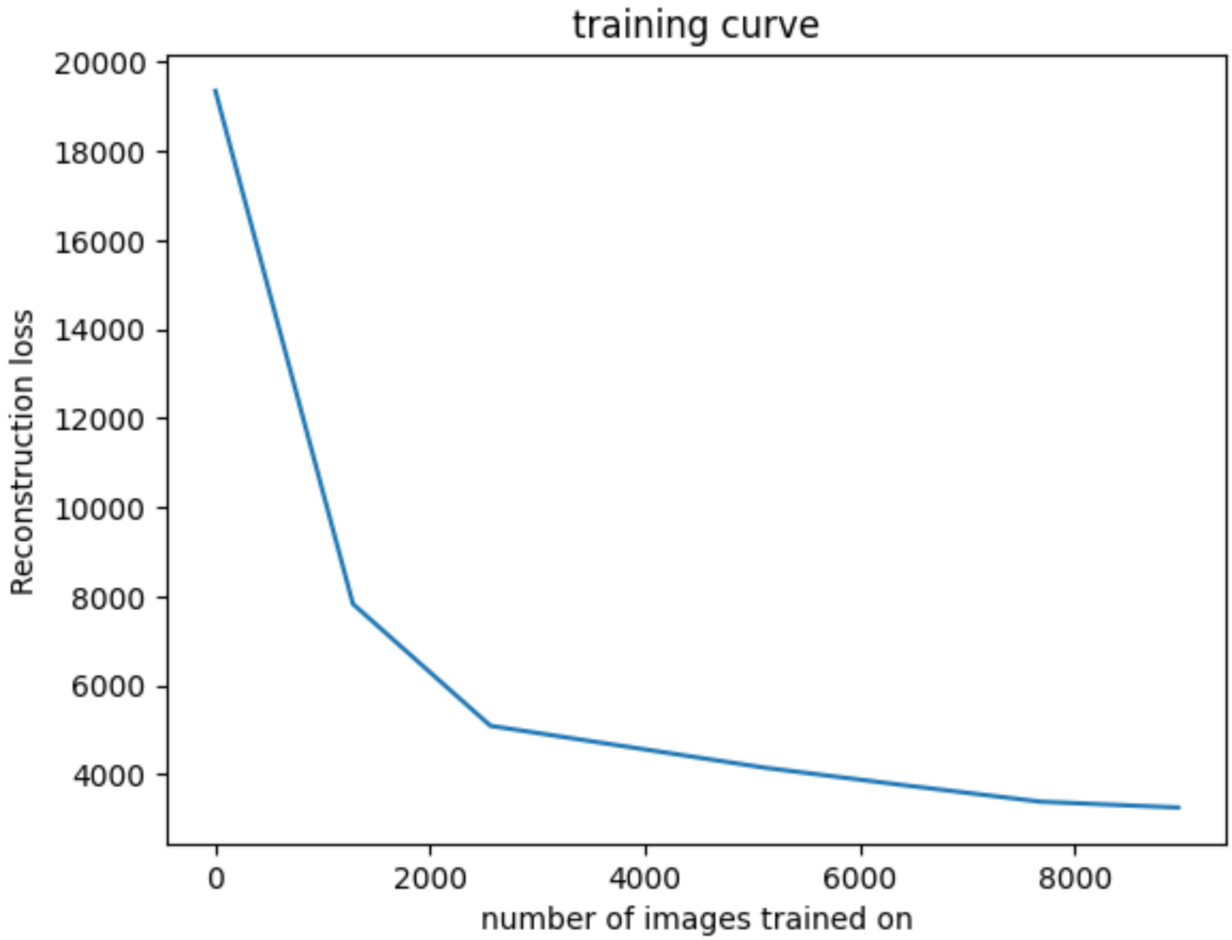
```
def autoencoder_training(model, loss_function, optimizer, train_data, n_epochs, u

    '''
    Updates the parameters of the given model using the optimizer of choice to
    reduce the given loss_function

    This will iterate over the dataloader 'n_epochs' times training on each batch
```

```
    To get the gradient (which is stored internally in the model) use .backward()
    and to apply it use .step() on the optimizer

    In between steps you need to zero the gradient so it can be recalculated -- u
    '''

    losses = []

    for n in range(n_epochs):
        for i, image in enumerate(tqdm(train_data)):

            image = image.float().view(-1,3,109,89)

            # TODO Complete the training loop using the instructions above
            # Hint: the above code essentially does one training step

            ##############################################################

            optimizer.zero_grad()

            output = model(image)

            loss = loss_function(output,image)

            loss.backward()

            optimizer.step()




            ##############################################################

            if i % update_interval == 0:
                losses.append(round(loss.item(), 2)) # This will append your loss

    return model, losses
```

```
# Plug in your model, loss function, and optimizer
# Try out different hyperparameters and different models to see how they perform
```

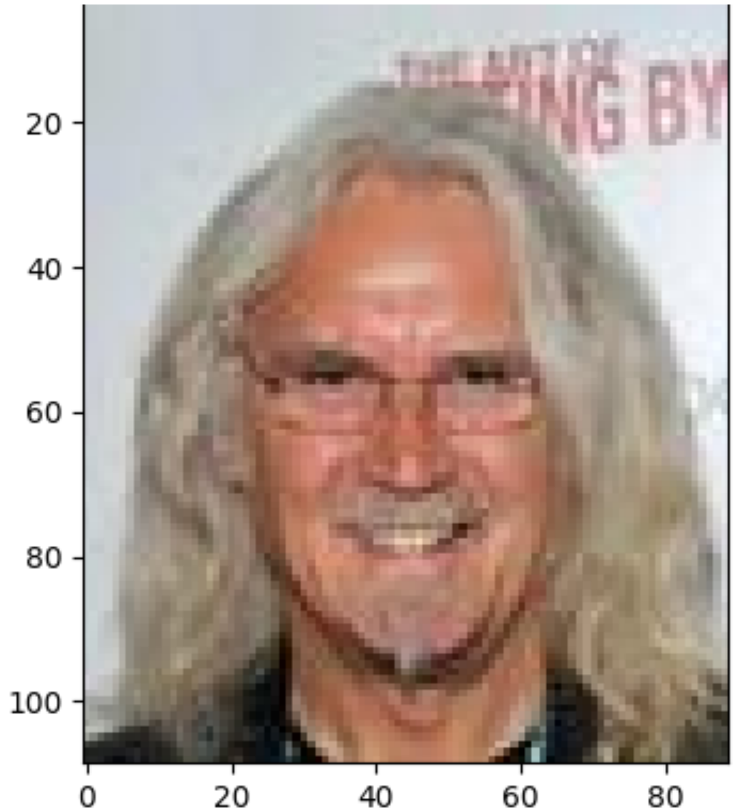
```
lr = 1e-4                 # The size of the step taken when doing gradient descent
batch_size = 128          # The number of images being trained on at once
update_interval = 10      # The number of batches trained on before recording loss
n_epochs = 1              # The number of times we train through the entire datase
compression_size = 100     # This is the size of the bottleneck (compression point

input_shape = (3,109,89)

dataset = dataset      # The dataset is a pain to load/unload so we want to keep

model = Autoencoder(input_shape, compression_size)
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

autoencoder_training_opt = torch.compile(autoencoder_training, mode="reduce-overh

trained_model, losses = autoencoder_training(model, loss_function, optimizer, dat

plt.plot(np.arange(len(losses)) * batch_size * update_interval, losses)
plt.title("training curve")
plt.xlabel("number of images trained on")
plt.ylabel("Reconstruction loss")
plt.show()


# NOTE: It will take a while for this to train (depending on your model)
# You can increase the batch size (way up top) or reduce the size of your model i
```

```
100
100%|██████████| 77/77 [00:25<00:00,  3.00it/s]
```

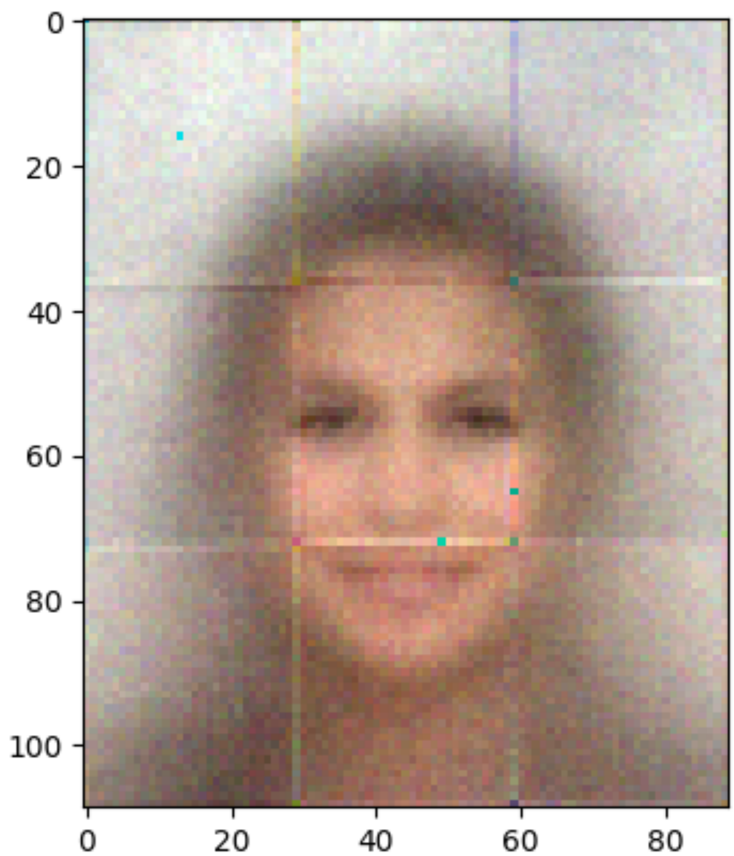



```
# Displays the reconstruction of the (now trained) autoencoder on the same exampl
# Notice that it worked and we have a better prediction (if your code works)
ex_image = dataset[random.randint(0,100)]
trained_output = trained_model(ex_image)

print("original image:")
plot_image(ex_image)
plt.show()
print("your (trained) reconstruction")
plot_image(trained_output.detach().byte())

# NOTE: It is very likely that this won't look that good at first
# It may even give the same output independent of the input -- rerun this a few t
# Play with the hyperparameters until you're happy with the output
```

```
original image:
```



your (trained) reconstruction



## Probably not great huh ^

Let's tune our hyperparameters so that we get something a bit cooler!

## Testing and HyperParameter Search (10 points)

Since the testing loop and training loop are so similar I'm going to go ahead and just give it to you -- but you gotta promise to at least look at the method to see how similar they are!

```
def testing(model, loss_function, test_data):

    '''
    This function will test the given model on the given test_data
    it will return the accuracy and the test loss (given by loss_function)
    '''

    sum_loss = 0

    for i, image in enumerate(tqdm(test_data)):

        # This is essentially exactly the same as the training loop
        # without the, well, training, part

        pred = model(image)
        loss = loss_function(pred, image)
        sum_loss += loss.item()

    avg_loss = round(sum_loss / len(test_data), 2)

    print("test loss:", avg_loss )

    return avg_loss

def train_and_test(model, loss_function, optimizer, batch_size, update_interval,

    '''
    This will use your/my methods to create a dataloader, train a gven model, anc

    Again, since I gave this to you for free you have to promise to look at it ar
    '''
```

```
    #training_opt = torch.compile(autoencoder_training, mode="reduce-overhead") #
    trained_model, losses = autoencoder_training(model, loss_function, optimizer,
    test_loss = testing(trained_model, loss_function, test_dataset)

    plt.plot(np.arange(len(losses)) * batch_size * update_interval, losses, color
    plt.hlines(test_loss, 0, len(losses) * batch_size * update_interval, color='r
    plt.legend()
    plt.title("training curve")
    plt.xlabel("number of images trained on")
    plt.ylabel("loss")
    plt.show()

    return trained_model, test_loss


avg_test_loss = testing(trained_model, loss_function, dataset[:1000]) # you'll ne
print(avg_test_loss)
```

```
100%|██████████| 77/77 [00:15<00:00,  4.94it/s]test loss: 1353.18
1353.18
```

```
#TODO Implement a hyperparameter search of your choice
# I'm not going to give any hand-holdy code cause I believe in you! (and NOT beca
import math
#######################################
#preforming random search because my computer wants to live to see another day (I
for i in range(10):
    lr = 1e-4*random.random() + 1e-10
    batch_size = math.floor(256*random.random())
    n_epochs = math.floor(10*random.random())
    compression_size = math.floor(500*random.random())+1
    print(f'({lr},{batch_size},{n_epochs},{compression_size})')
    model = Autoencoder(input_shape, compression_size)
    loss_function = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    autoencoder_training_opt = torch.compile(autoencoder_training, mode="reduce-o

    trained_model, losses = autoencoder_training(model, loss_function, optimizer,
    avg_test_loss = testing(trained_model, loss_function, dataset[:1000]) # you'l
    print(avg_test_loss)
```

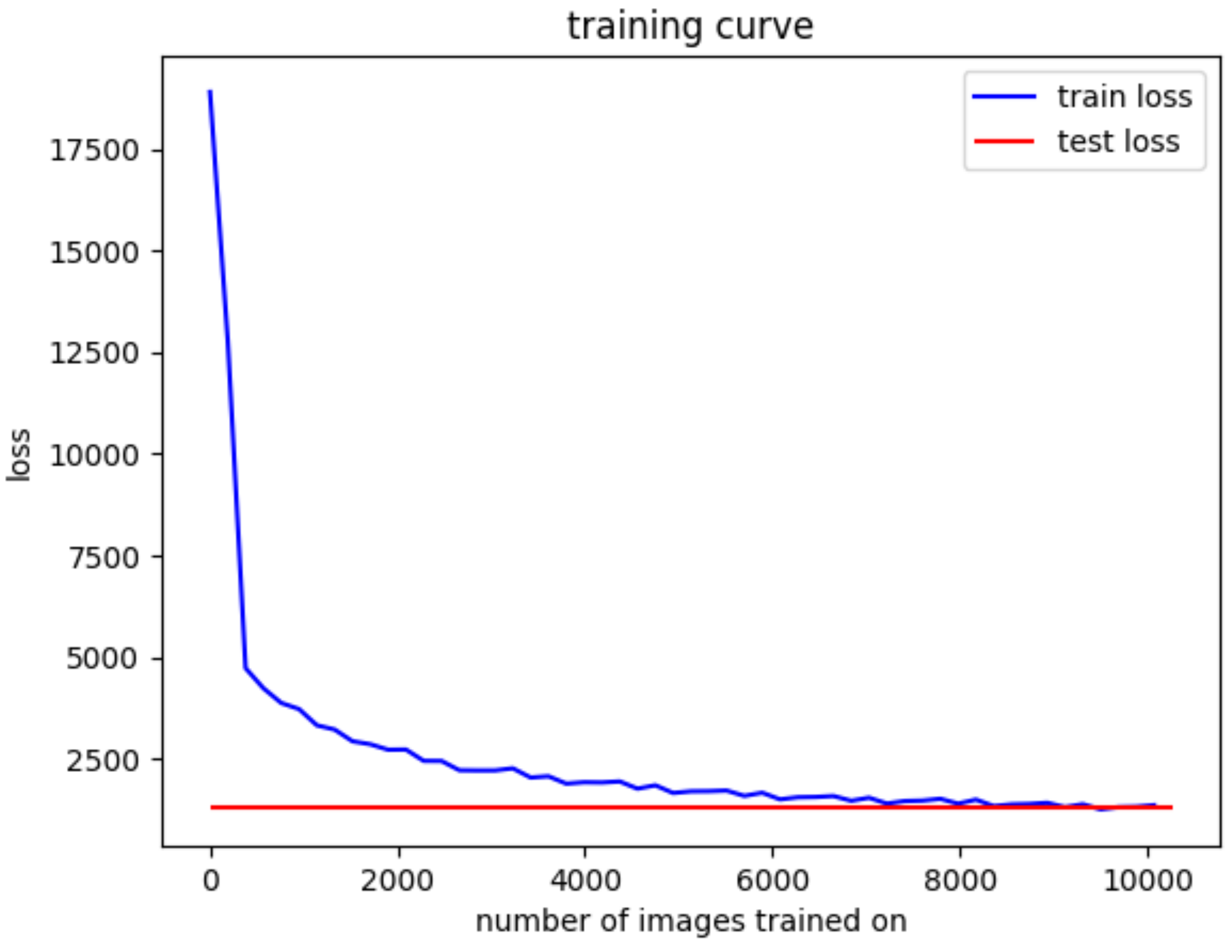
```
########################################

(7.683366581571608e-05,248,2,199)
199
100%|██████████| 77/77 [01:00<00:00,  1.27it/s]
100%|██████████| 77/77 [00:58<00:00,  1.32it/s]
100%|██████████| 77/77 [00:09<00:00,  8.36it/s]
test loss: 2420.84
2420.84
(2.212447510729385e-05,4,2,163)
163
100%|██████████| 77/77 [00:51<00:00,  1.49it/s]
100%|██████████| 77/77 [00:48<00:00,  1.58it/s]
100%|██████████| 77/77 [00:09<00:00,  8.54it/s]
test loss: 2969.62
2969.62
(7.873841655646074e-05,229,0,234)
234
100%|██████████| 77/77 [00:08<00:00,  8.60it/s]
test loss: 18421.35
18421.35
(5.439812263987071e-05,84,3,32)
32
100%|██████████| 77/77 [00:46<00:00,  1.67it/s]
100%|██████████| 77/77 [00:47<00:00,  1.62it/s]
100%|██████████| 77/77 [00:45<00:00,  1.70it/s]
100%|██████████| 77/77 [00:08<00:00,  8.61it/s]
test loss: 3953.45
3953.45
(7.56785324574722e-05,19,9,291)
291
100%|██████████| 77/77 [00:49<00:00,  1.57it/s]
100%|██████████| 77/77 [00:48<00:00,  1.60it/s]
100%|██████████| 77/77 [00:52<00:00,  1.46it/s]
100%|██████████| 77/77 [00:47<00:00,  1.62it/s]
100%|██████████| 77/77 [00:47<00:00,  1.60it/s]
100%|██████████| 77/77 [00:49<00:00,  1.57it/s]
100%|██████████| 77/77 [00:48<00:00,  1.59it/s]
100%|██████████| 77/77 [00:48<00:00,  1.59it/s]
100%|██████████| 77/77 [00:48<00:00,  1.58it/s]
100%|██████████| 77/77 [00:09<00:00,  8.49it/s]
test loss: 1274.05
1274.05
(6.191201171979885e-06,209,4,166)
166
100%|██████████| 77/77 [00:58<00:00,  1.31it/s]
100%|██████████| 77/77 [00:51<00:00,  1.48it/s]
100%|██████████| 77/77 [00:46<00:00,  1.64it/s]
```

```
100%|██████████| 77/77 [00:51<00:00,  1.49it/s]
100%|██████████| 77/77 [00:09<00:00,  8.43it/s]
test loss: 3517.64
3517.64
(5.513039895025278e-05,25,7,107)
107
100%|██████████| 77/77 [00:52<00:00,  1.48it/s]
100%|██████████| 77/77 [00:50<00:00,  1.54it/s]
100%|██████████| 77/77 [01:11<00:00,  1.07it/s]
100%|██████████| 77/77 [00:52<00:00,  1.46it/s]
100%|██████████| 77/77 [00:45<00:00,  1.68it/s]
100%|██████████| 77/77 [00:45<00:00,  1.69it/s]
100%|██████████| 77/77 [00:57<00:00,  1.34it/s]
100%|██████████| 77/77 [00:09<00:00,  8.44it/s]
```

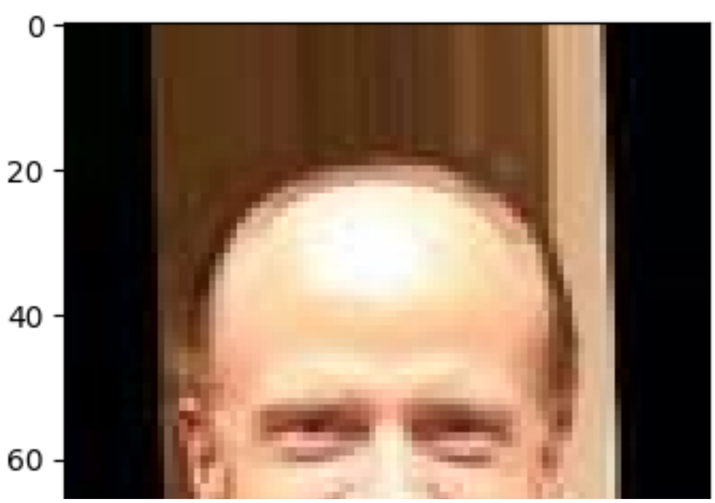
```
# Use your best hyperparameters -- your final test loss should be under 2000
lr = 7.56785324574722e-05              # The size of the step taken when doing gr
batch_size = 19       # The number of images being trained on at once
update_interval = 10   # The number of batches trained on before recording loss
n_epochs = 9          # The number of times we train through the entire dataset
compression_size = 291  # The output size of the encoder

model = Autoencoder(input_shape, compression_size)
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

train_dataset = dataset[len(dataset)//4:len(dataset)]
test_dataset = dataset[:len(dataset)//4]

best_model, _ = train_and_test(model, loss_function, optimizer, batch_size, updat

ex_image = dataset[random.randint(0,77)]
trained_output = best_model(ex_image)

print("original image:")
plot_image(ex_image)
plt.show()
print("your (BEST) reconstruction")
plot_image(trained_output.detach().byte())

# Try to get a reconstruction that you are happy with
# It is difficult though so try to set up a big search and go for a hike or somet
# be warned that google collab sometimes cuts off after some time so be careful!
```

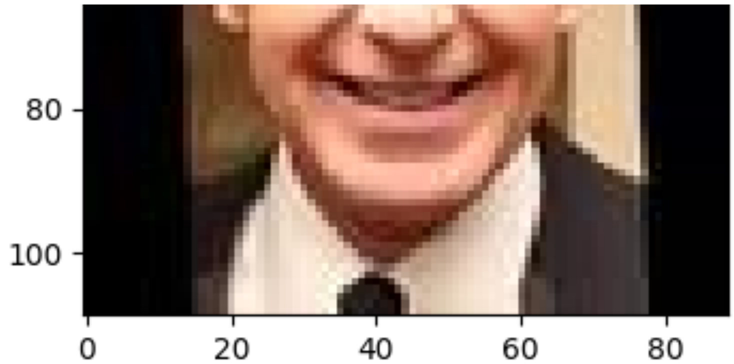
```
291
100%|██████████| 58/58 [00:42<00:00,  1.36it/s]
100%|██████████| 58/58 [00:43<00:00,  1.33it/s]
100%|██████████| 58/58 [00:42<00:00,  1.36it/s]
```

```
100%|██████████| 58/58 [00:42<00:00,  1.36it/s]
100%|██████████| 58/58 [00:43<00:00,  1.34it/s]
100%|██████████| 58/58 [00:38<00:00,  1.51it/s]
100%|██████████| 58/58 [00:40<00:00,  1.43it/s]
100%|██████████| 58/58 [00:40<00:00,  1.43it/s]
100%|██████████| 58/58 [00:37<00:00,  1.55it/s]
100%|██████████| 58/58 [00:38<00:00,  1.52it/s]
100%|██████████| 19/19 [00:02<00:00,  8.38it/s]
test loss: 1280.61
```

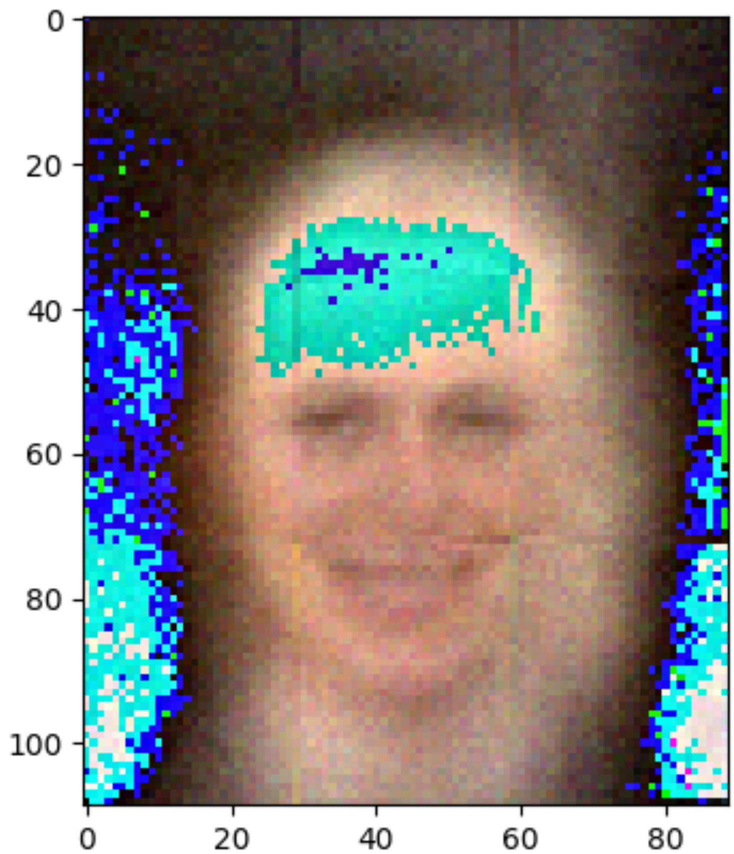



```
original image:
```






your (BEST) reconstruction



## Autoencoder Written Report (10 points)

Now, lets take a bit of break from implementing models and do some writing (I know you all love that right?) Fill out your answer to each question in the empty markdown cell below each question.

1. What would happen if the compression size of your autoencoder was as large as the input image? Try it and tell me what you found out!

I theorized that the image would be very accurate but since there are neuron layers in between it would not be perfect I was somewhat dissapointed with the results because there was still a lot of noise maybe more epochs, more convolutions, and especially would be more effective.

2. Was your model able to output any faces that were looking in different directions? Why do you think it would be hard for an autoencoder to learn to output faces with different orientations?

I think the autoencoder has diffuclty learning facial features facing a certain direction because the dataset is too small so the model probably just averages them out to facing forwards.

## Generative Adversarial Network (60 points)

Now it's time for us to make a Generative Adversarial Network (GAN)!

GANs contain a generator that generates an image based on a given dataset, and a discriminator (classifier) to distinguish whether an image is real or generated.

GANs are very similar to autoencoders in the sense that we will create two different models, but we will actually train them on different losses!

## The GAN model (20 points)

In this part you will create your model for both the Discriminator and Generator. The discriminator will take in an input the size of the images and output a bit which represents either real or fake (the discriminators geuss as to the input is real data or generated)

The generator will take in an input of some size (it will end up being noise but this wont come up in the model making part) and then output an "image" that is the same size as the real data.

```
class Discriminator(nn.Module):

    '''
    This will be your discriminator half of you GAN
    it will take in something of the shape of an image of a face

    it will then return either 0 or 1 depending on whether it
    believes the input is from the real distribution or not
    '''

    def __init__(self, input_shape):
        super(Discriminator, self).__init__()

        # TODO Initialize your discriminator
        # You can linear and conv layers -- as well as anything else you find (dc
        # HINT: if it trains too slow try reducing the dimensions for your linera
        #####################################

        self.conv1 =nn.Conv2d(3,10,3)
        self.flatten = nn.Flatten()
        self.lin1 = nn.Linear((input_shape[1] - 2)*(input_shape[2]-2)*10 ,1000)
        self.lin2 = nn.Linear(1000,291)
        self.lin3 = nn.Linear(291,1)
        self.rel = nn.ReLU()
        self.to('mps')




        #####################################

    def forward(self, x):
```

```
# TODO fill out the forward pass of your model
# Don't forget nonlinearities!
####################################
x= x.to('mps')

x = self.conv1(x)

x = self.flatten(x)

x = self.lin1(x)
x = self.rel(x)
x = self.lin2(x)
x = self.rel(x)
x = self.lin3(x)




####################################

x = nn.Sigmoid()(x) # This sigmoid will squish the outputs between 0 and

return x.to('cpu')
```

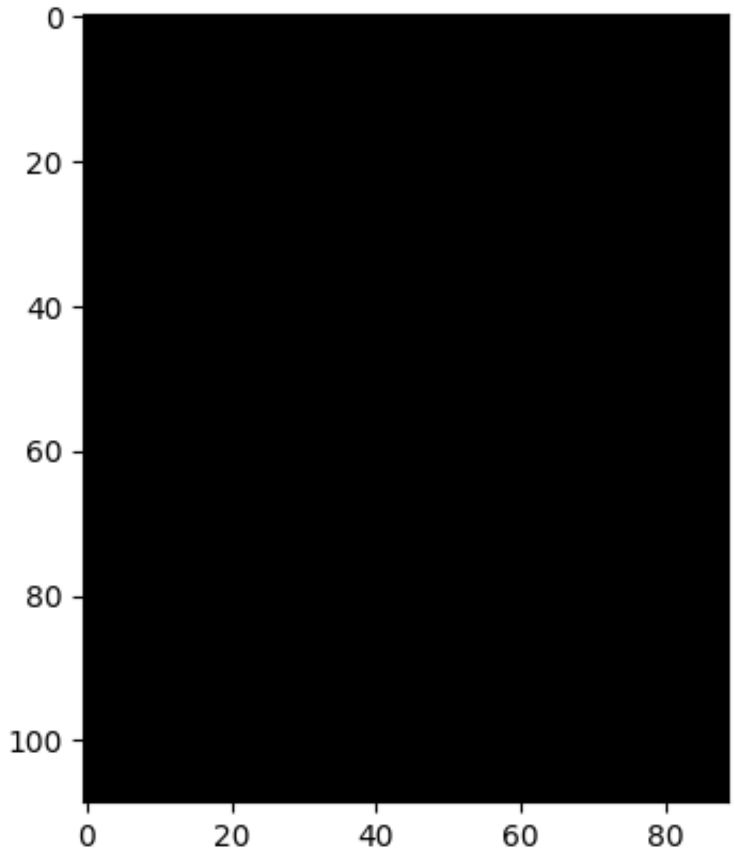
```
# This is the performance of the Discriminator (before training) on the example i
# If you rerun this it should change since we are randomly initializing the model
discriminator = Discriminator((3,109,89))
ex_output = discriminator(ex_image.float())

plot_image(ex_image)
print("Output of the discriminator given this input:", ex_output[0].detach().nump
```

```
Output of the discriminator given this input: 0.99999857
```

```
class Generator(nn.Module):
    def __init__(self, input_size, output_shape):
        super(Generator, self).__init__()

        # TODO
       ####################################

        self.lin1 = nn.Linear(input_size,1000)
        self.lin2= nn.Linear(1000,(output_shape[1] - 2)*(output_shape[2]-2)*10)
        self.shape = [-1,10,output_shape[1]-2,output_shape[2]-2]
        self.conv = nn.ConvTranspose2d(10,output_shape[0],3)
        self.sig = nn.Softmax()
        self.rel = nn.ReLU()
        self.to('mps')
        print(input_size)

        ####################################

    def forward(self, x):
        # TODO
       ####################################
        x= x.to('mps')

        out = self.lin1(x)
        out = self.rel(out)
        out = self.lin2(out)

        out = out.reshape(self.shape)

        out = self.conv(out)

        #out = self.sig(out)


        ####################################
        return out.to('cpu')
```

```
# This will show the output of our generator before training (it's fine if its al
test_gen = Generator(100, (3, 109, 89))
noise = (torch.rand(1, 100) - 0.5) / 0.5
test_output = test_gen(noise)

plot_image(test_output.detach().byte())
```

100



## This is our generator's attempt at making something before training ^

Let's train it to see how it can improve!

### Training Loop (20 points)

The training for a GAN is fundementally the same for all the other models we train with pytorch (zero grad, output, loss, loss backward, optimizer step). But we are going to do 2 seperate updates in each loop, with different losses!

For each loop you will calculate the loss for both the discriminator and generator and then update those models accordingly. Some code is provided to help you out, but not all of it!

You should record your *losses* for both the generator and the discriminator **(two lists in total)**. You'll need these values for the written section, where you will be discussing them.

Then, use your wisdom from the autoencoder hyperparameter search to find good settings to all the hyperparamers and try your best to get your model to produce a face!

[Hint for reseting the optimizer]()

[Hint for stepping with the optimizer]() (You'll have to use .backward() to get the gradient)

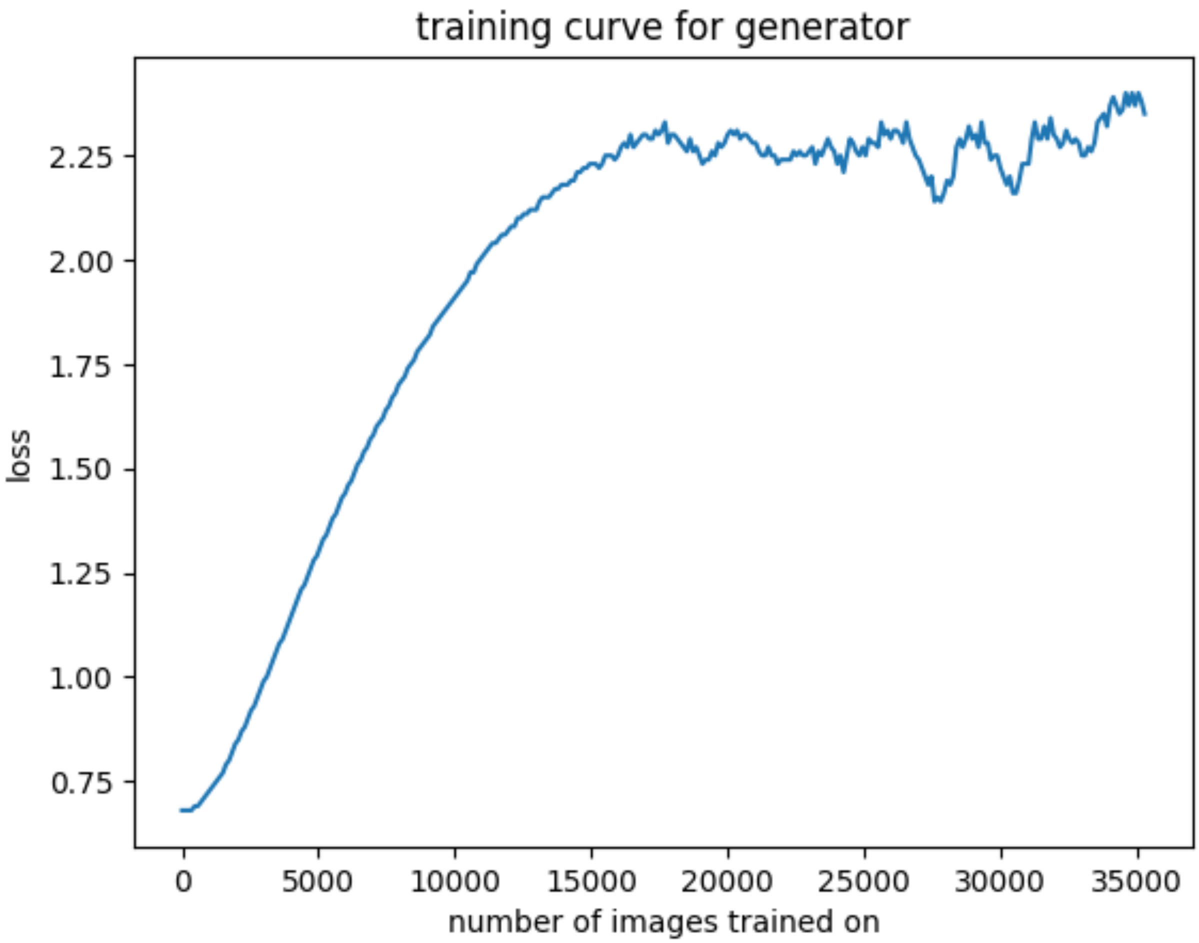
```
def training(generator, discriminator, loss, g_optimizer, d_optimizer, train_data

    g_losses = []
    d_losses = []

    for epoch in range(n_epochs):
        for i, image in enumerate(tqdm(train_dataloader)):

            # Training the discriminator
            # Real inputs are actual images from the CelebA dataset
            # Fake inputs are from the generator
            # Real inputs should be classified as 1 and fake as 0

            image = image.float()

            real_classifications = discriminator(image/255)
            real_labels = torch.ones(image.shape[0])

            noise = (torch.rand(image.shape[0], noise_samples) - 0.5) / 0.5
```

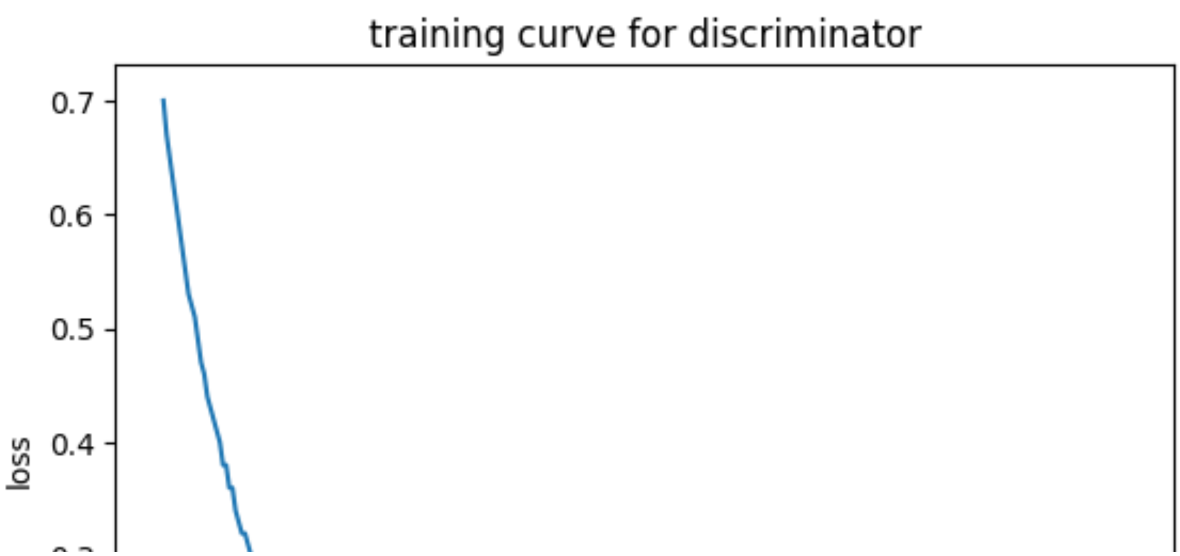
```
fake_inputs = generator(noise)
fake_classifications = discriminator(fake_inputs)
fake_labels = torch.zeros(image.shape[0])

classifications = torch.cat((real_classifications, fake_classificatic
targets = torch.cat((real_labels, fake_labels), 0)

# TODO Calculate the loss for the discriminator and apply the gradier
# This is the same as a normal training loop!
#######################################################

d_optimizer.zero_grad()



d_loss = loss(classifications,targets)

d_loss.backward()

d_optimizer.step()




#######################################################

if i % update_interval == 0:
    d_losses.append(round(d_loss.item(), 2))


# We do a seperate forward pass to update the gradient for the genera
# Pytorch doesnt like us reusing the same computation graph (it makes
noise = (torch.rand(image.shape[0], noise_samples) - 0.5) / 0.5
fake_inputs = generator(noise)
fake_classifications = discriminator(fake_inputs)
fake_labels = torch.zeros(image.shape[0], 1)

# TODO Calculate the loss for the generator and apply the gradient
# HINT: the loss for the generator is essentially the opposite of the
# discriminators loss but doesnt care about the real examples (they c
#######################################################
g_optimizer.zero_grad()
```

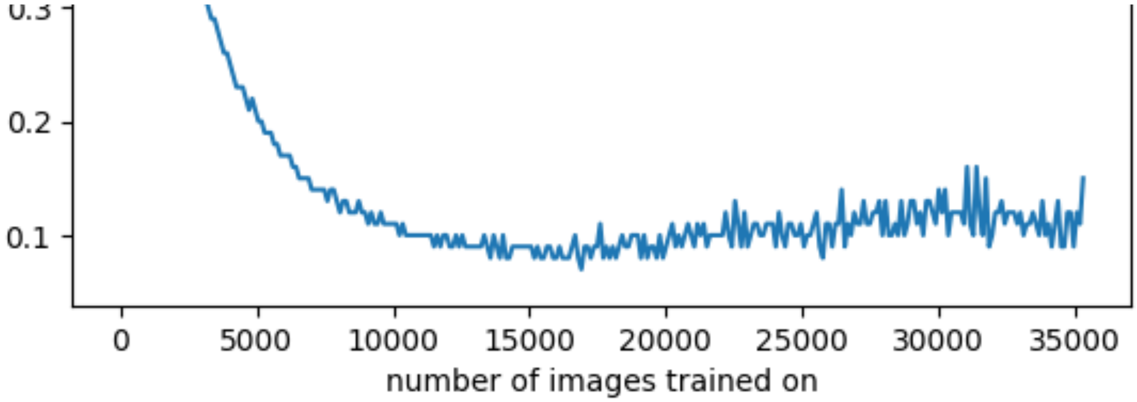
```
            g_loss = loss(1-fake_classifications,fake_labels)

            g_loss.backward()

            g_optimizer.step()



            ########################################################

            if i % update_interval == 0:
                g_losses.append(round(g_loss.item(), 2))

    return (generator, discriminator), (g_losses, d_losses)


lr = 1.9011708023003832e-06            # The size of the step taken when doing gr
batch_size = 115         # The number of images being trained on at once
update_interval = 1   # The number of batches trained on before recording loss
n_epochs = 4           # The number of times we train through the entire dataset
noise_samples = 310    # The size of the noise input to the Generator

loss_function = nn.BCELoss()

G_model = Generator(noise_samples, (3,109,89))
D_model = Discriminator((3,109,89))
G_optimizer = torch.optim.Adam(G_model.parameters(), lr=lr*2)     # This is an im
D_optimizer = torch.optim.Adam(D_model.parameters(), lr=lr)       # This is an im

train_dataset = dataset

models, losses = training(G_model, D_model, loss_function, G_optimizer, D_optimiz

G_model, D_model = models
g_losses, d_losses = losses

plt.plot(np.arange(len(g_losses)) * batch_size * update_interval, g_losses)
plt.title("training curve for generator")
plt.xlabel("number of images trained on")
plt.ylabel("loss")
plt.show()

plt.plot(np.arange(len(d_losses)) * batch_size * update_interval, d_losses)
plt.title("training curve for discriminator")
plt.xlabel("number of images trained on")
```

```
plt.ylabel("loss")
plt.show()
```

```
310
100%|██████████| 77/77 [01:01<00:00,  1.25it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
```



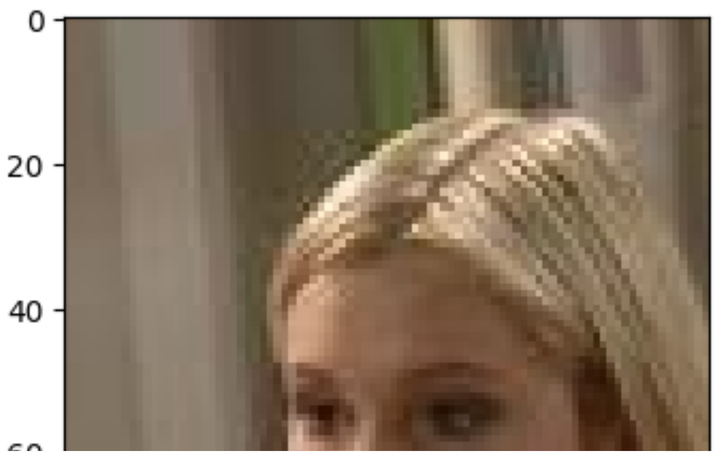





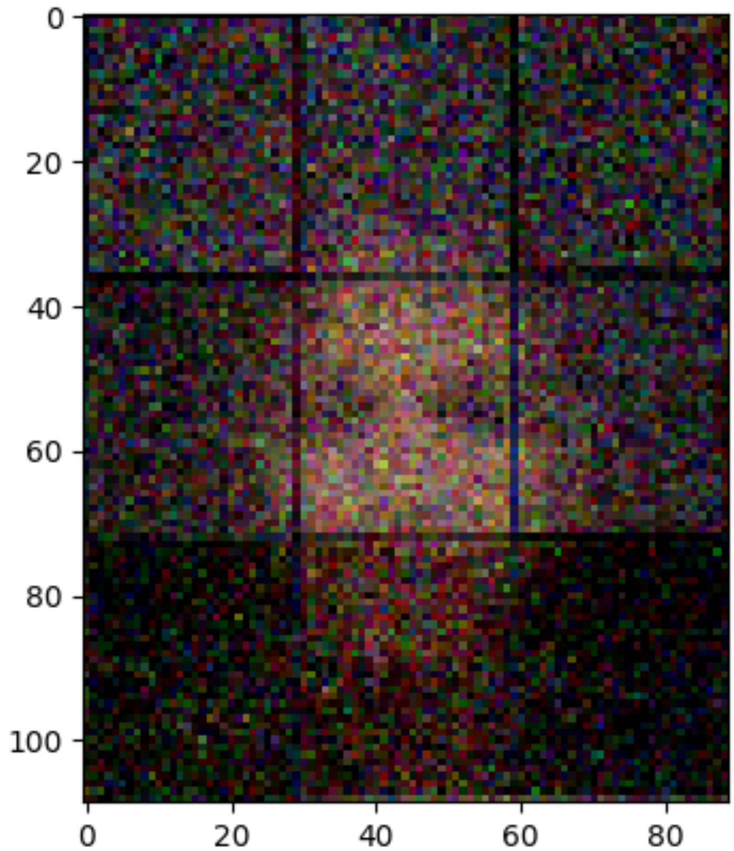

Now let's take a look at the generated images coming out of our trained GAN!

```
# This will show the same example as before with the discriminator's new score
# 0 is fake and 1 is real -- is it good at discriminating?

trained_output = D_model(ex_image.float())

plot_image(ex_image)
print("Output of the discriminator given this input:", trained_output[0].detach()
plt.show()

noise = (torch.rand(1, noise_samples) - 0.5) / 0.5
trained_gen = G_model(noise)

plot_image(trained_gen.detach())

trained_output = D_model(trained_gen.float())

print("Output of the discriminator given this generated input:", trained_output[0
```

```
Output of the discriminator given this input: 1.0
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.10205155
```



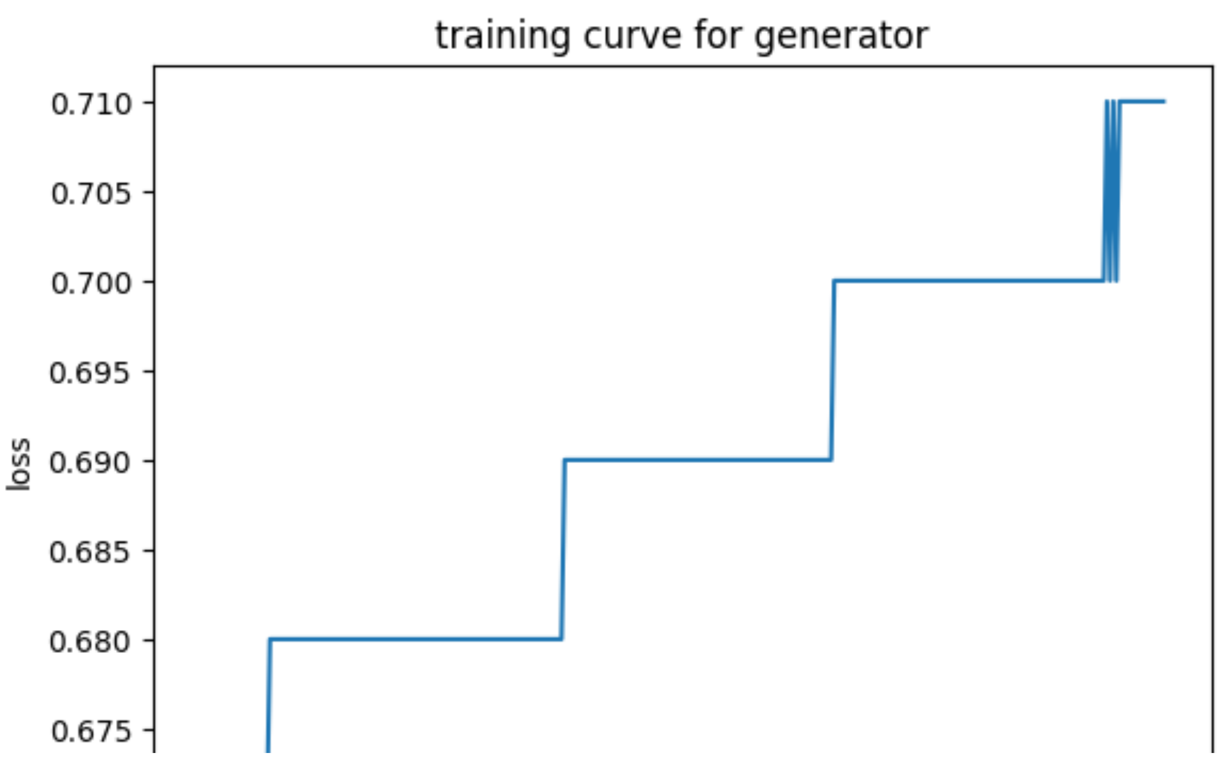
## GAN hyperparameter Search (10 points)

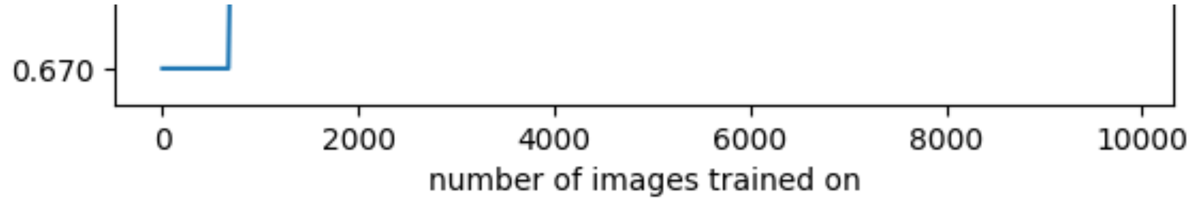
GANs are notoriously hard to train -- generally you have to do a lot of hyper parameter searching to find good settings. Try it out until you get results above that you're happy with it! You're going to have to write a good amount of code for this, but you can base it off of what is above if you want

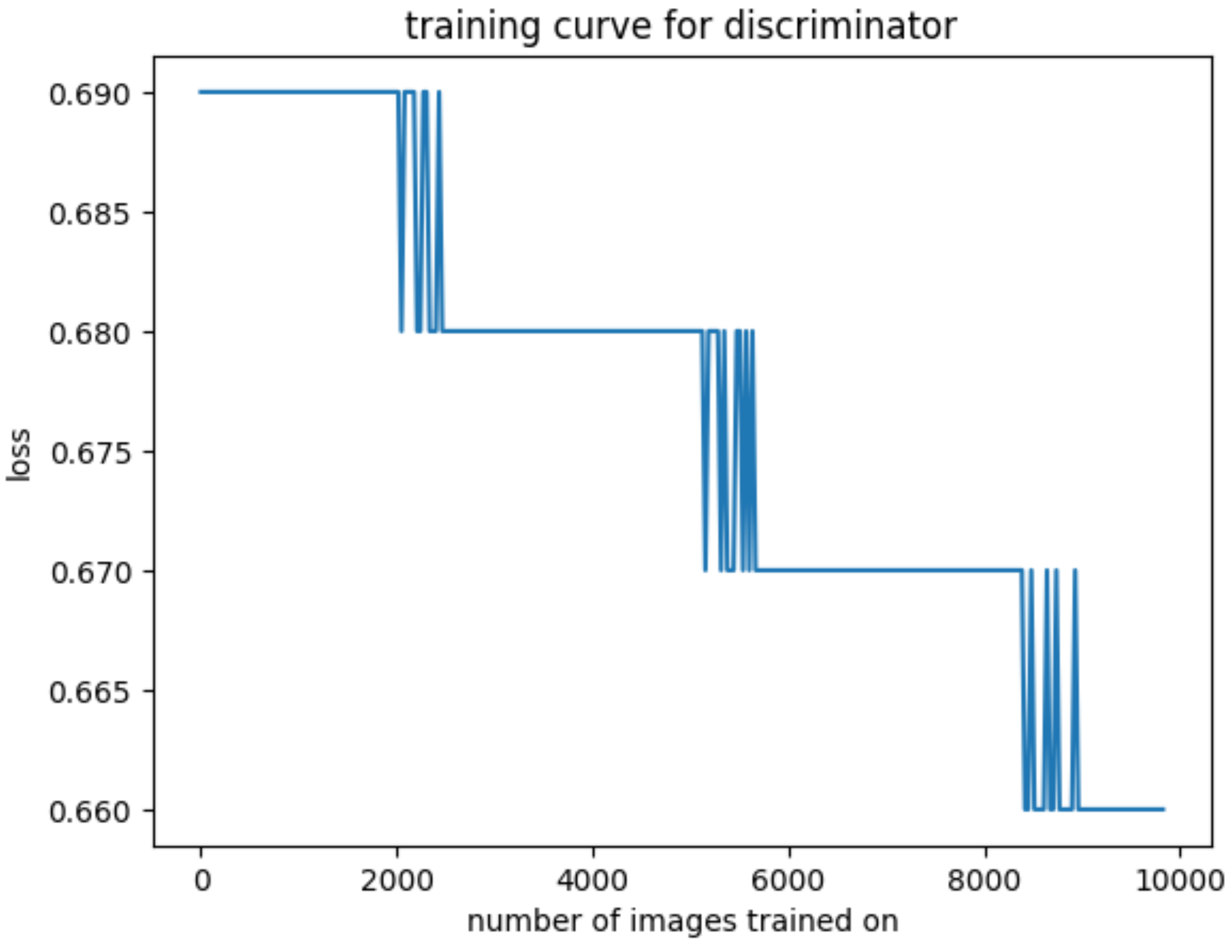
```
# TODO Do a hyper parameter search and find the best settings for your model
# Again I leave this up to you as to what to do, but I'm gonna warn you that tryi
# is probably going to be too slow to work
#######################################################
import math
for i in range(10):
    lr = 2*(10**(-(9*random.random())))              # The size of the step taker
    batch_size = math.floor(256*random.random())        # The number of images be
    update_interval = 1   # The number of batches trained on before recording los
    n_epochs = 4            # The number of times we train through the entire dat
    noise_samples = math.floor(1000*random.random())+100    # The size of the noi

    loss_function = nn.BCELoss()

    G_model = Generator(noise_samples, (3,109,89))
    D_model = Discriminator((3,109,89))
    G_optimizer = torch.optim.Adam(G_model.parameters(), lr=lr)     # This is an
    D_optimizer = torch.optim.Adam(D_model.parameters(), lr=lr)       # This is a

    train_dataset = dataset

    models, losses = training(G_model, D_model, loss_function, G_optimizer, D_opt

    G_model, D_model = models
    g_losses, d_losses = losses

    plt.plot(np.arange(len(g_losses)) * batch_size * update_interval, g_losses)
    plt.title("training curve for generator")
    plt.xlabel("number of images trained on")
    plt.ylabel("loss")
    plt.show()

    plt.plot(np.arange(len(d_losses)) * batch_size * update_interval, d_losses)
    plt.title("training curve for discriminator")
    plt.xlabel("number of images trained on")
```

```
    plt.ylabel("loss")
    plt.show()
    for i in range(3):
        trained_output = D_model(ex_image.float())

        plot_image(ex_image)
        print("Output of the discriminator given this input:", trained_output[0].
        plt.show()

        noise = (torch.rand(1, noise_samples) - 0.5) / 0.5
        trained_gen = G_model(noise)

        plot_image(trained_gen.detach())

        trained_output = D_model(trained_gen.float())

        print("Output of the discriminator given this generated input:", trained_

        print(f'lr:f{lr} batch_size:f{batch_size} noise_samples:f{noise_samples}'

#######################################################
```

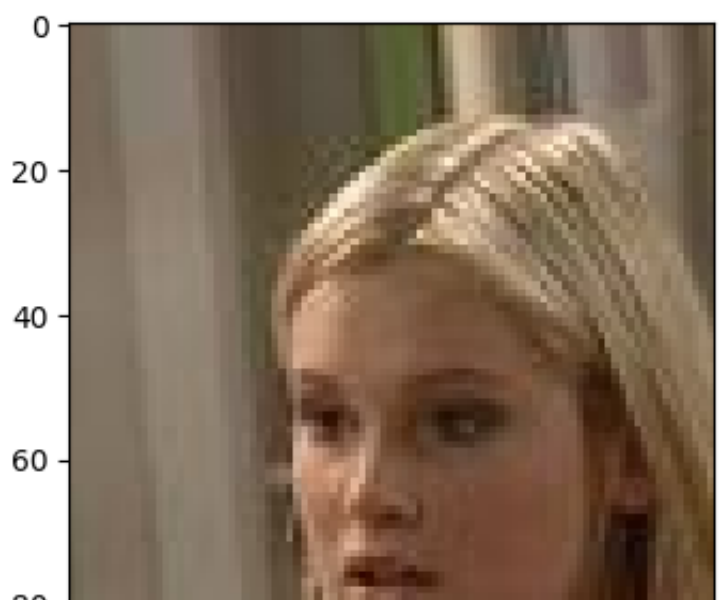
```
248
100%|██████████| 77/77 [01:00<00:00,  1.28it/s]
100%|██████████| 77/77 [00:59<00:00,  1.29it/s]
100%|██████████| 77/77 [00:59<00:00,  1.30it/s]
100%|██████████| 77/77 [00:59<00:00,  1.30it/s]
```

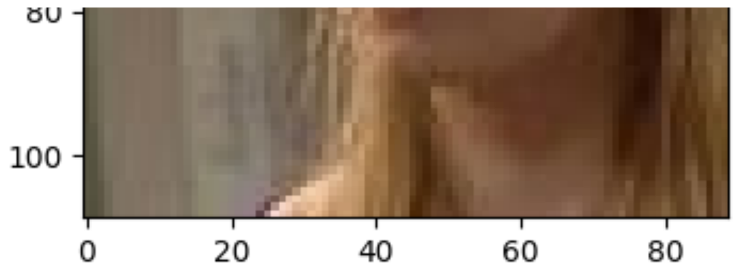




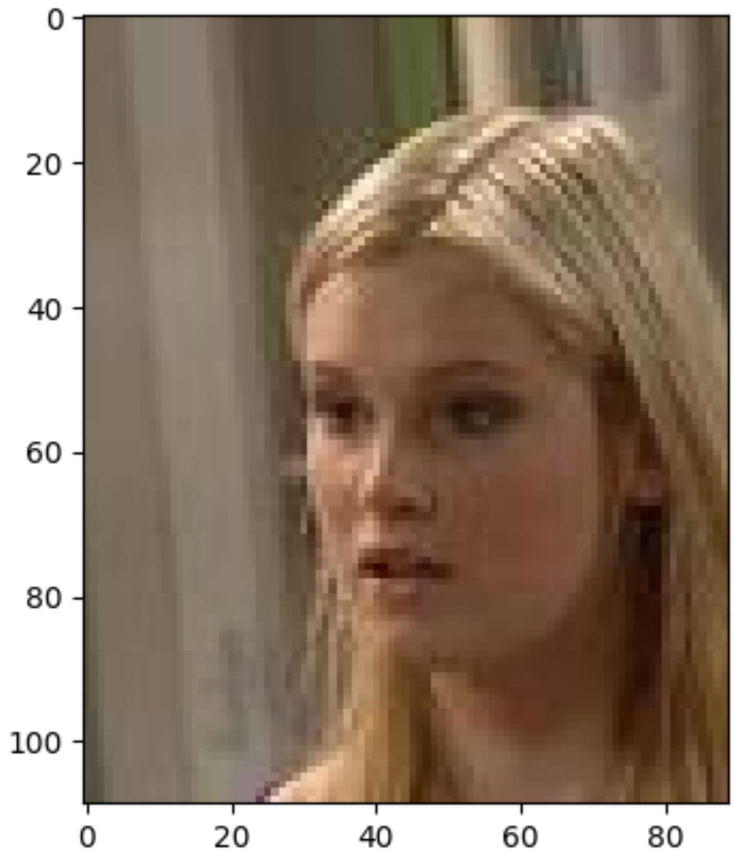




```
Output of the discriminator given this input: 1.0
```



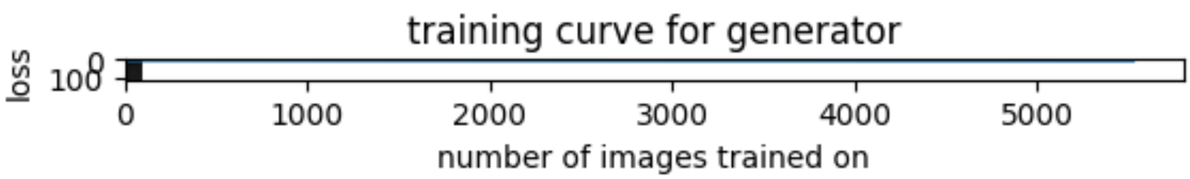


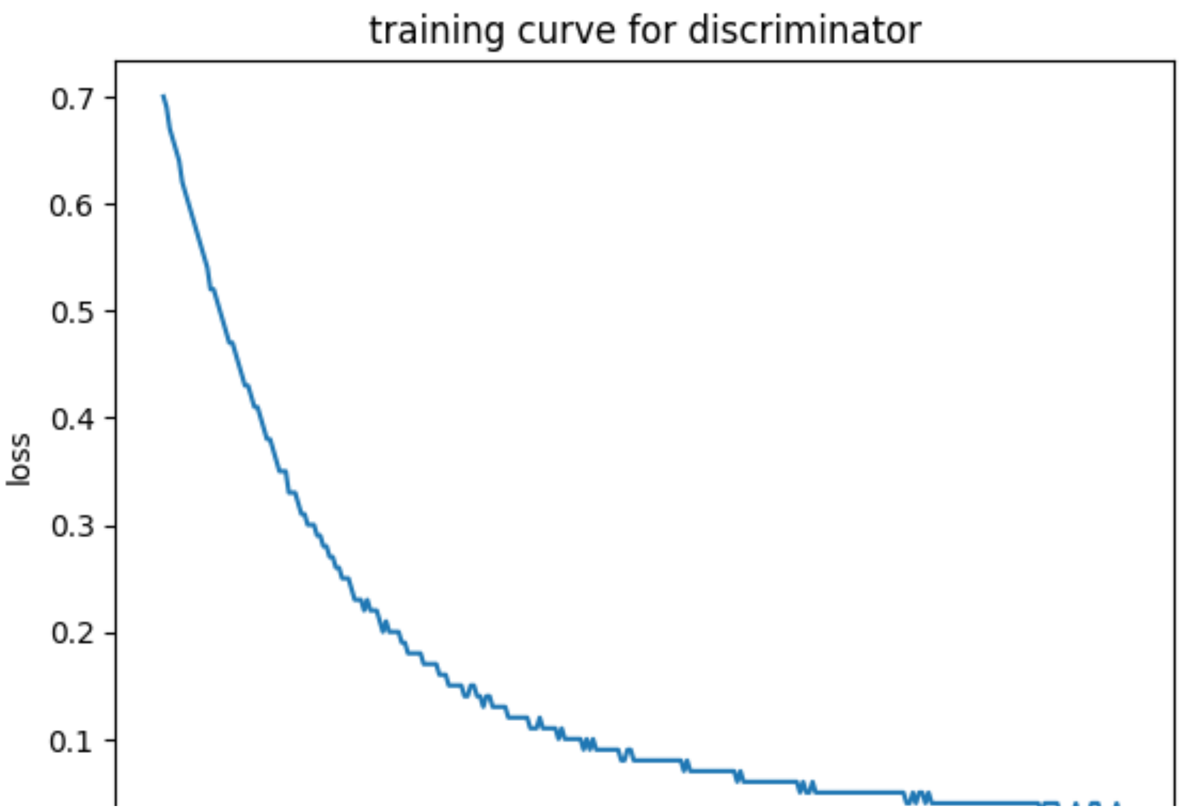
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.493941
lr:f1.0203720979709525e-08 batch_size:f32 noise_samples:f248
Output of the discriminator given this input: 1.0
```

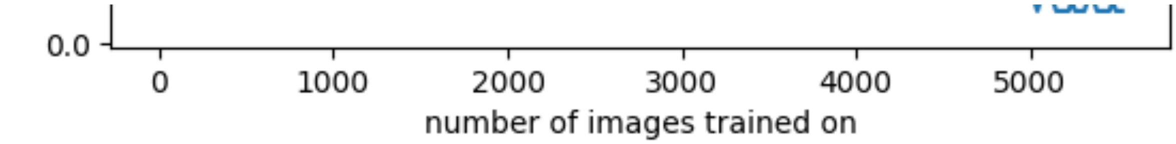


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.49568194
lr:f1.0203720979709525e-08 batch_size:f32 noise_samples:f248
Output of the discriminator given this input: 1.0
```


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.49452174
lr:f1.0203720979709525e-08 batch_size:f32 noise_samples:f248
257
100%|██████████| 77/77 [00:59<00:00,  1.30it/s]
100%|██████████| 77/77 [00:56<00:00,  1.37it/s]
100%|██████████| 77/77 [00:56<00:00,  1.37it/s]
100%|██████████| 77/77 [00:56<00:00,  1.37it/s]
```

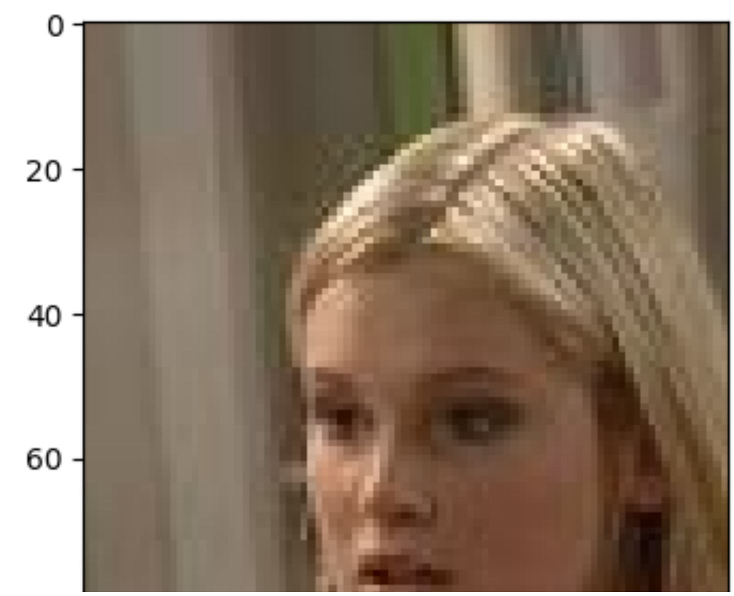


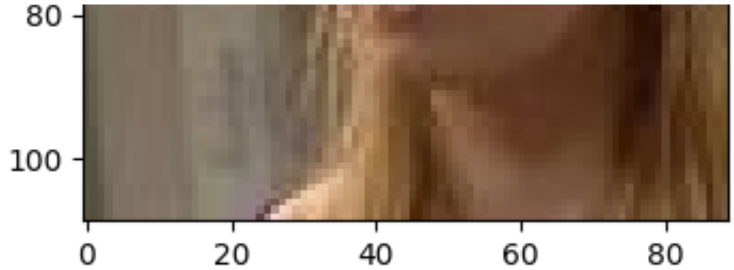






```
Output of the discriminator given this input: 1.0
```

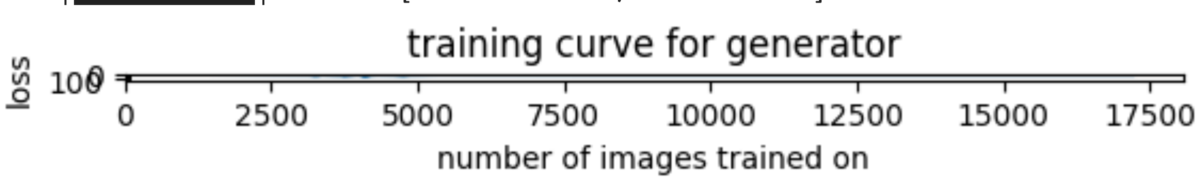

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0496094
lr:f8.022610829104374e-07 batch_size:f18 noise_samples:f257
Output of the discriminator given this input: 1.0
```

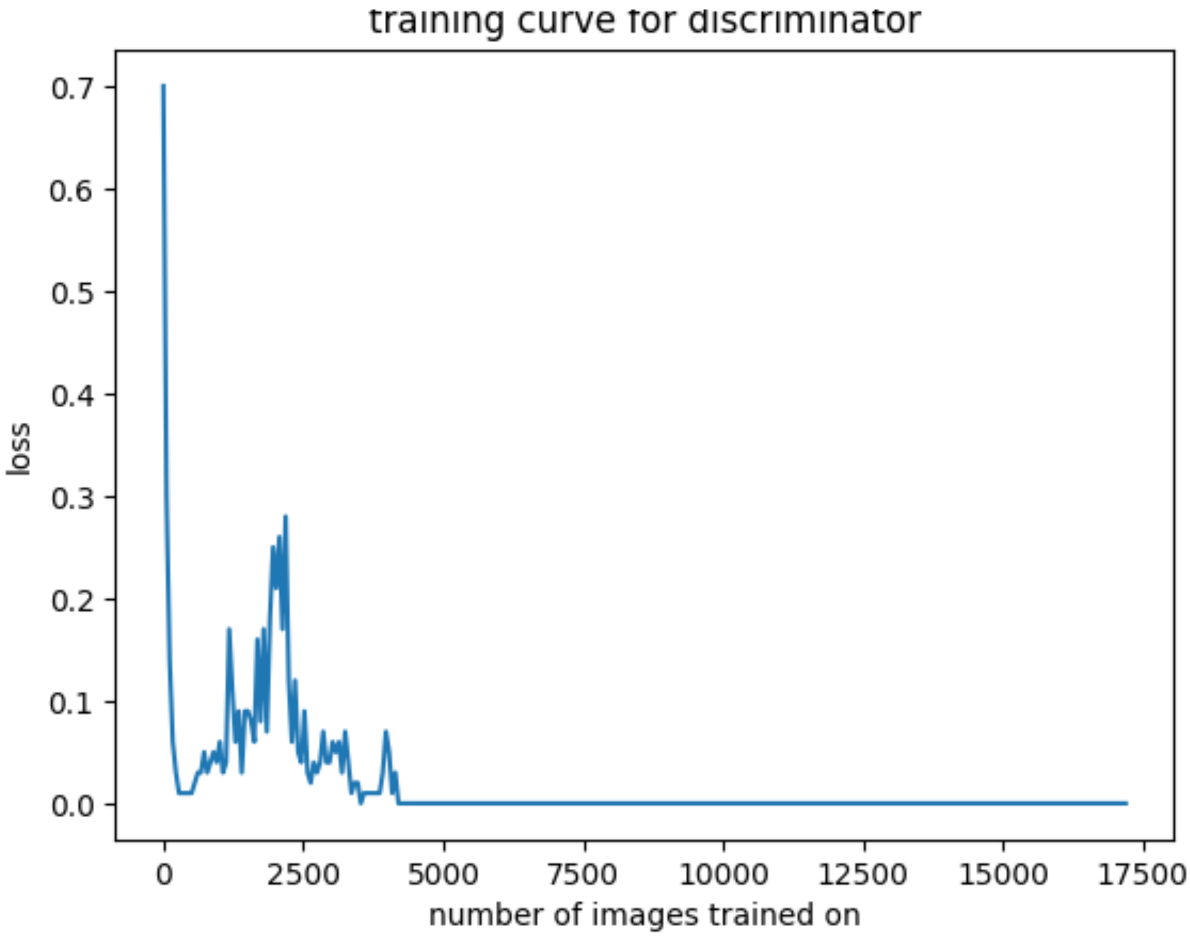




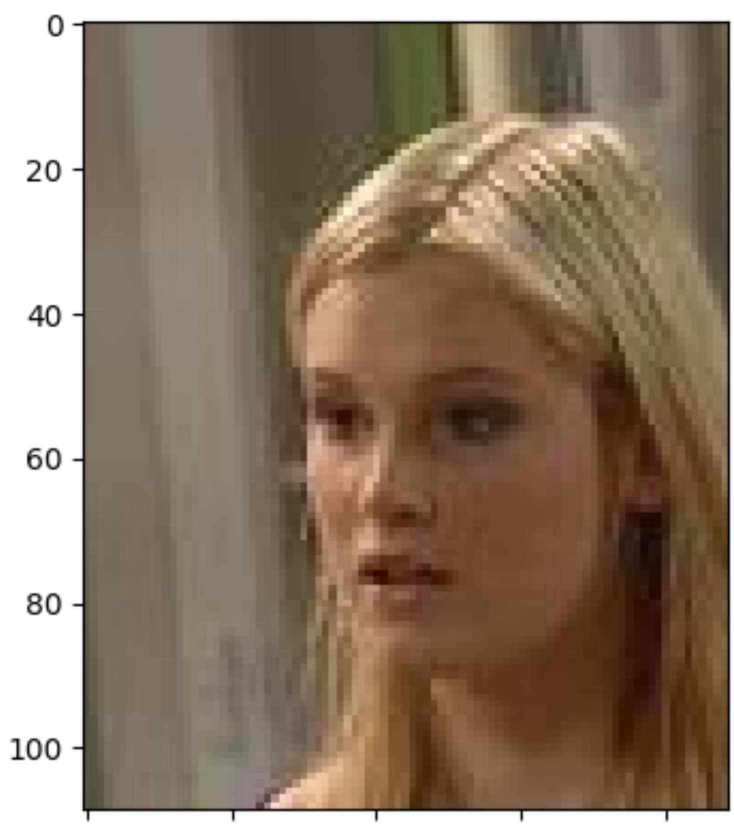

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.04756458
lr:f8.022610829104374e-07 batch_size:f18 noise_samples:f257
Output of the discriminator given this input: 1.0
```


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.050070975
lr:f8.022610829104374e-07 batch_size:f18 noise_samples:f257
724
100%|██████████| 77/77 [00:59<00:00,  1.28it/s]
100%|██████████| 77/77 [00:58<00:00,  1.31it/s]
100%|██████████| 77/77 [00:58<00:00,  1.31it/s]
100%|██████████| 77/77 [00:58<00:00,  1.31it/s]
```

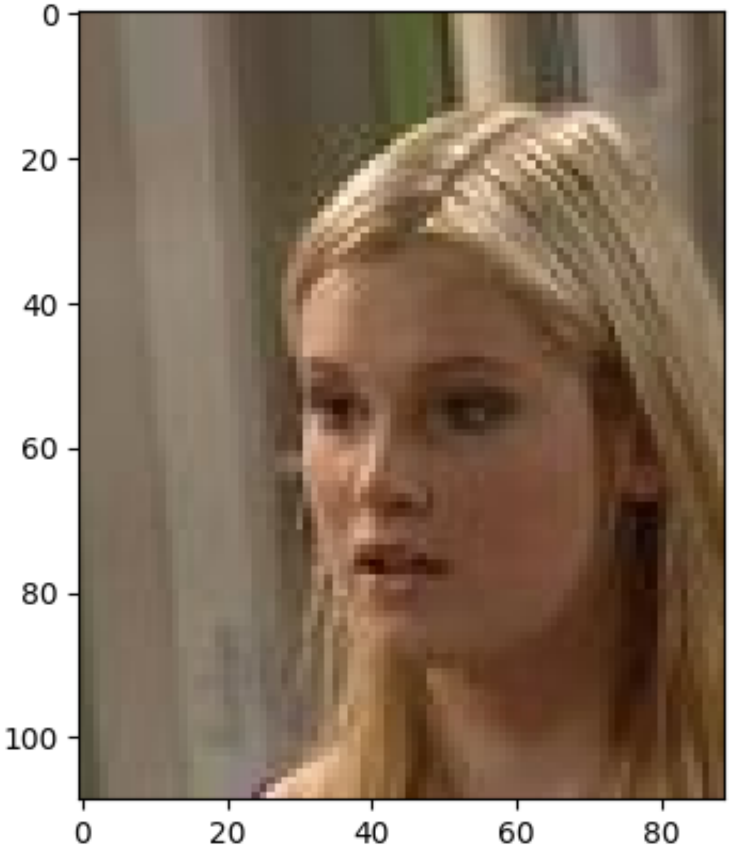




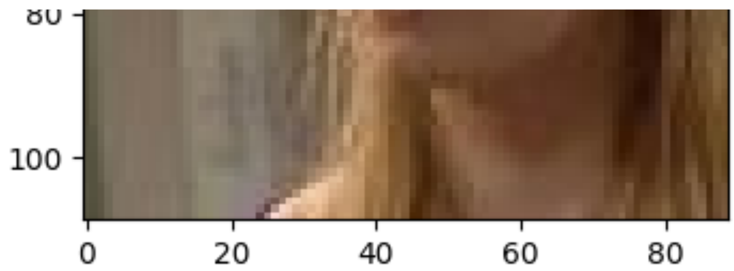

```
Output of the discriminator given this input: 1.0
```



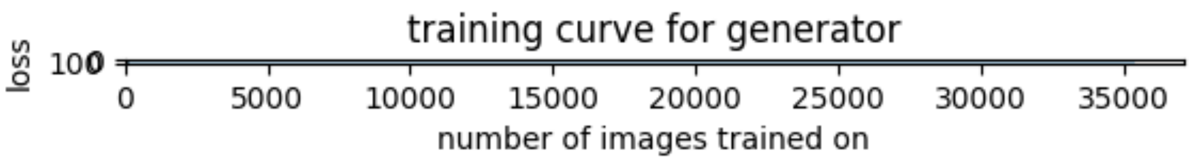
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 2.7984556e-06
lr:f0.00010266910332654086 batch_size:f56 noise_samples:f724
Output of the discriminator given this input: 1.0
```

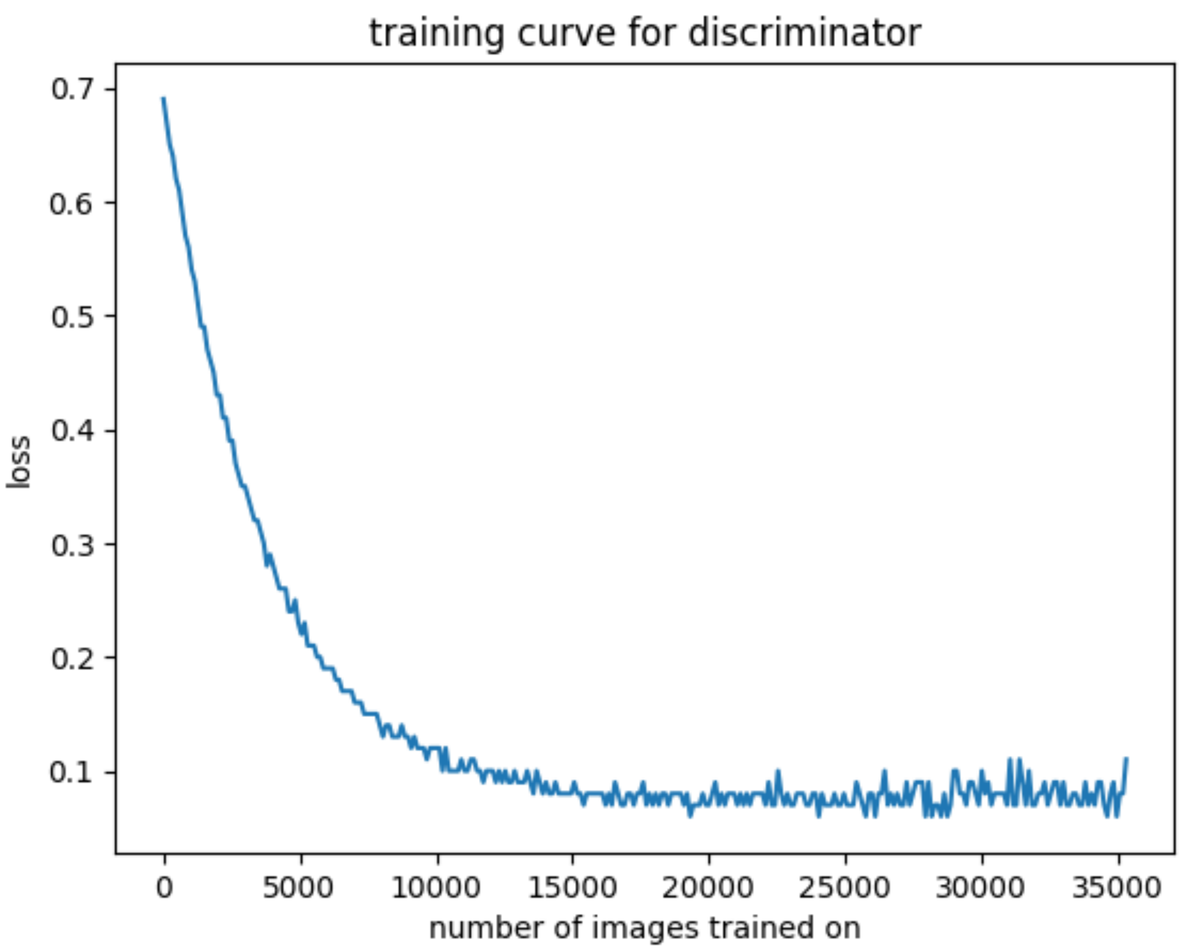



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 9.2530166e-07
lr:f0.00010266910332654086 batch_size:f56 noise_samples:f724
Output of the discriminator given this input: 1.0
```

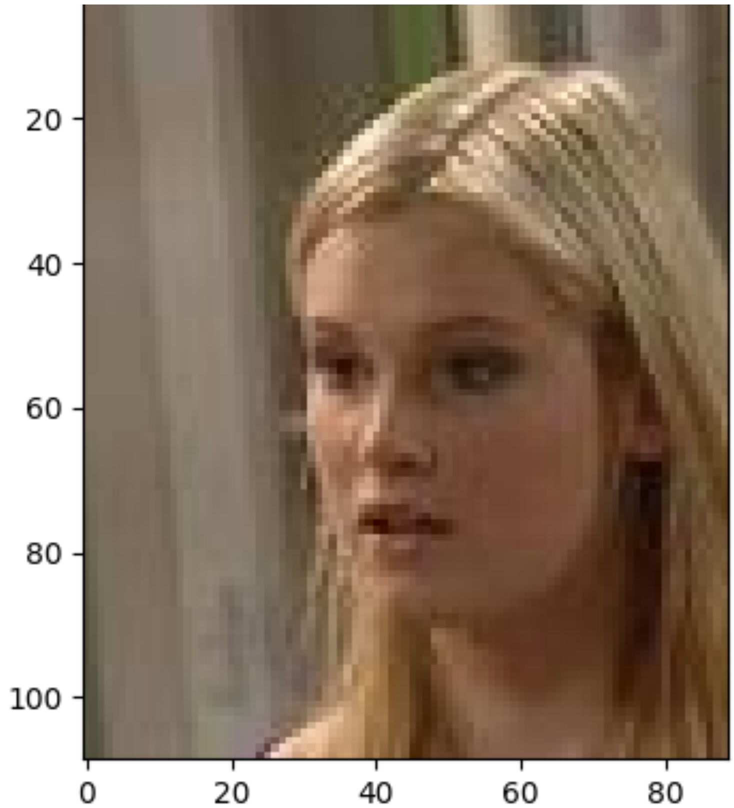


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.2776206e-05
lr:f0.00010266910332654086 batch_size:f56 noise_samples:f724
310
100%|██████████| 77/77 [01:06<00:00,  1.16it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
```

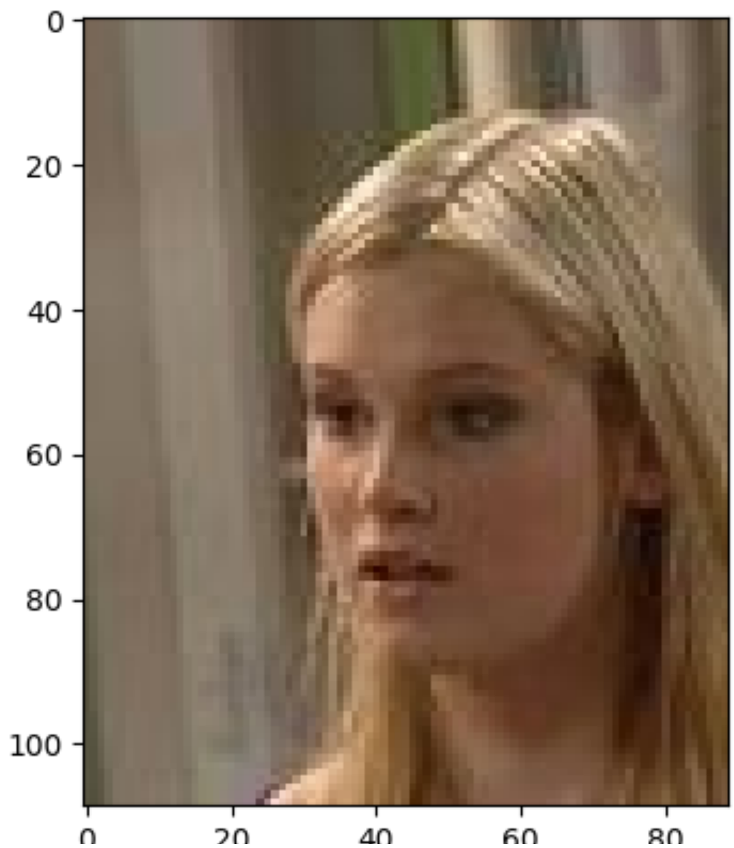


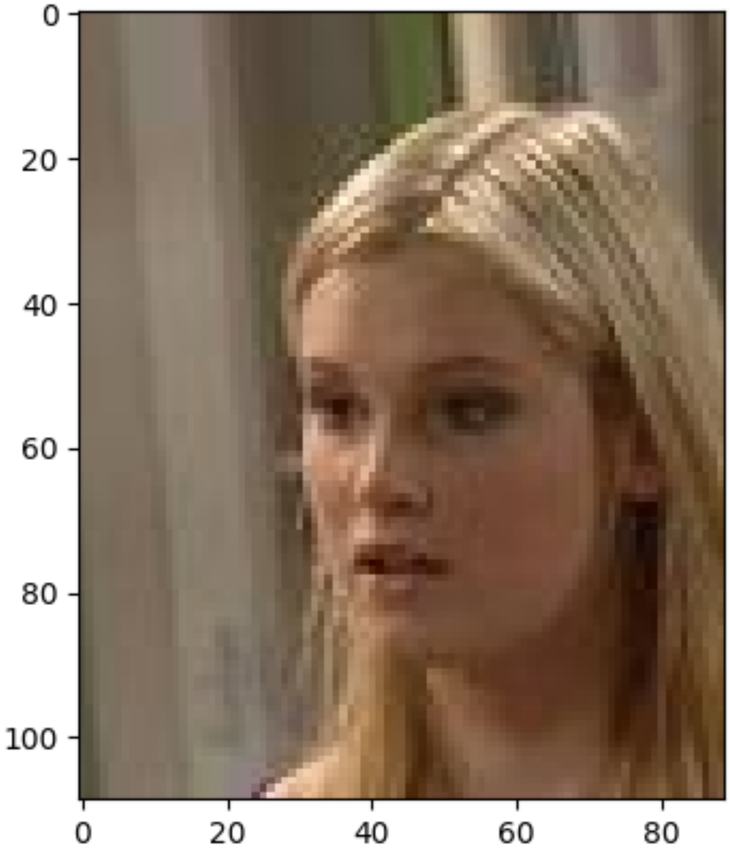

```
Output of the discriminator given this input: 1.0
```



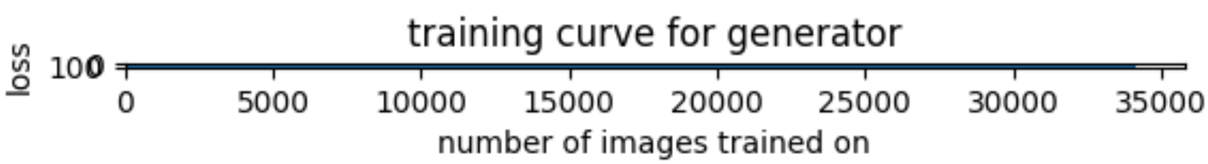

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.06739414
lr:f1.9011708023003832e-06 batch_size:f115 noise_samples:f310
Output of the discriminator given this input: 1.0
```

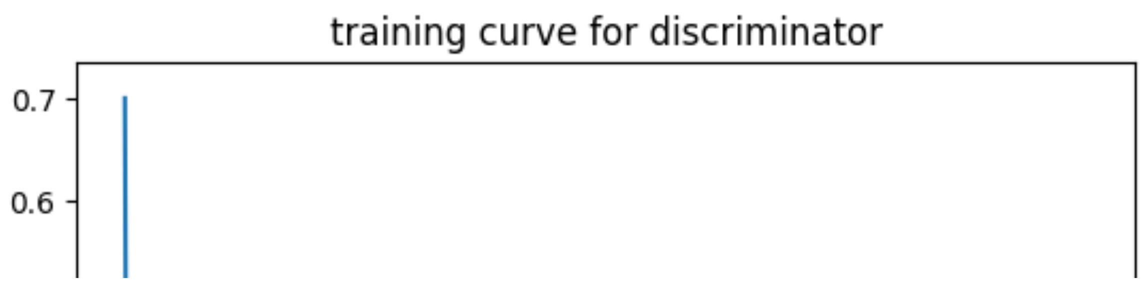


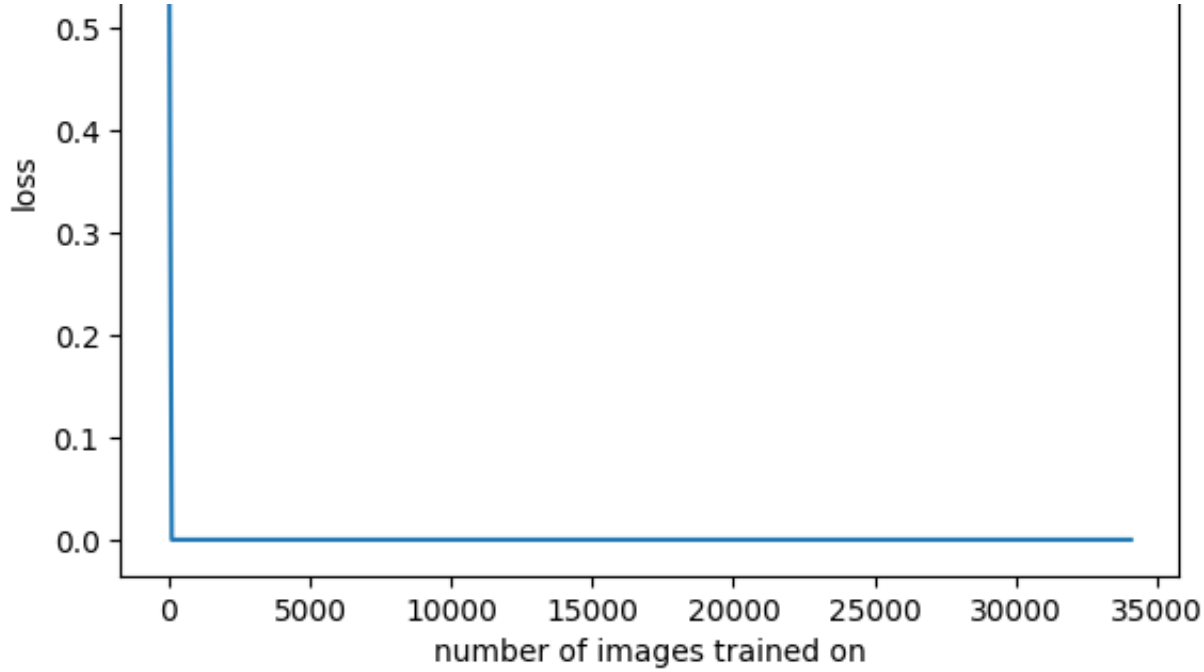
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.05888891
lr:f1.9011708023003832e-06 batch_size:f115 noise_samples:f310
Output of the discriminator given this input: 1.0
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.055518005
lr:f1.9011708023003832e-06 batch_size:f115 noise_samples:f310
110
100%|██████████| 77/77 [03:42<00:00,  2.89s/it]
100%|██████████| 77/77 [03:26<00:00,  2.68s/it]
100%|██████████| 77/77 [00:54<00:00,  1.40it/s]
100%|██████████| 77/77 [00:55<00:00,  1.39it/s]
```

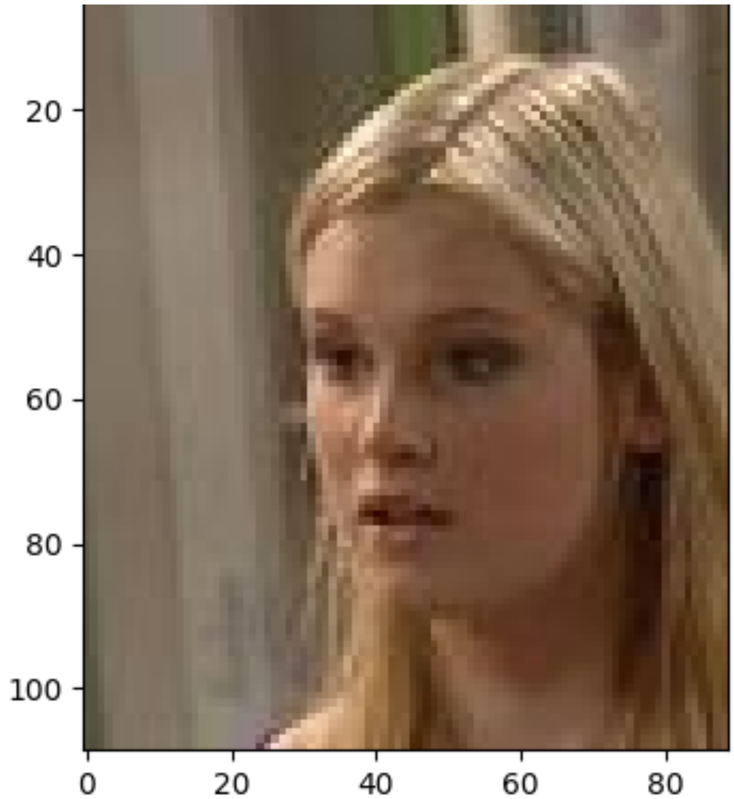


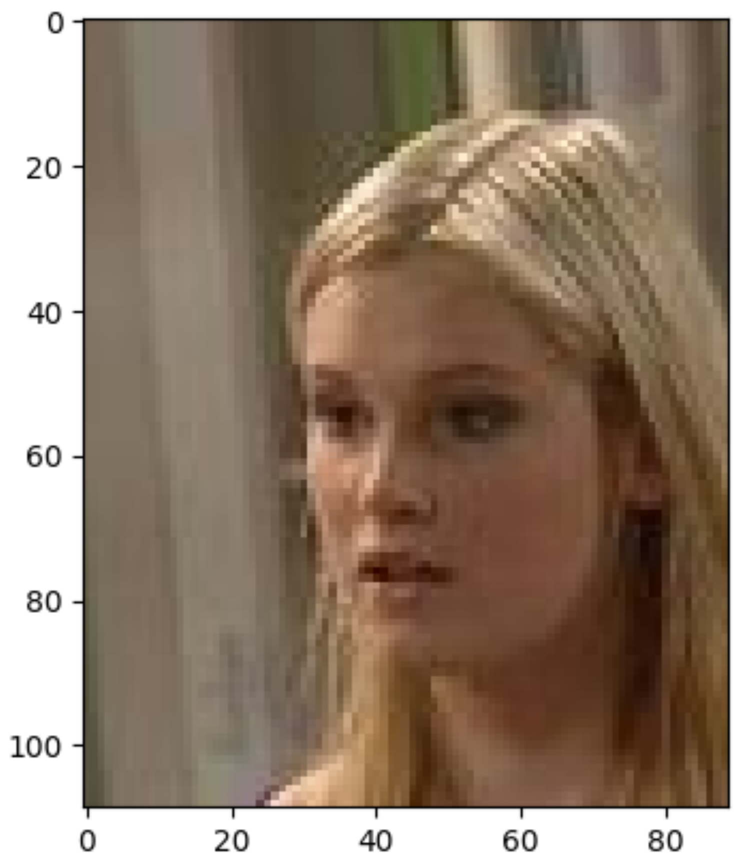



```
Output of the discriminator given this input: 1.0
```

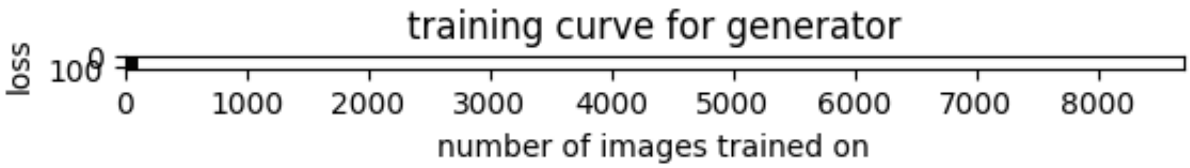

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0
lr:f0.21293085706478135 batch_size:f111 noise_samples:f110
Output of the discriminator given this input: 1.0
```

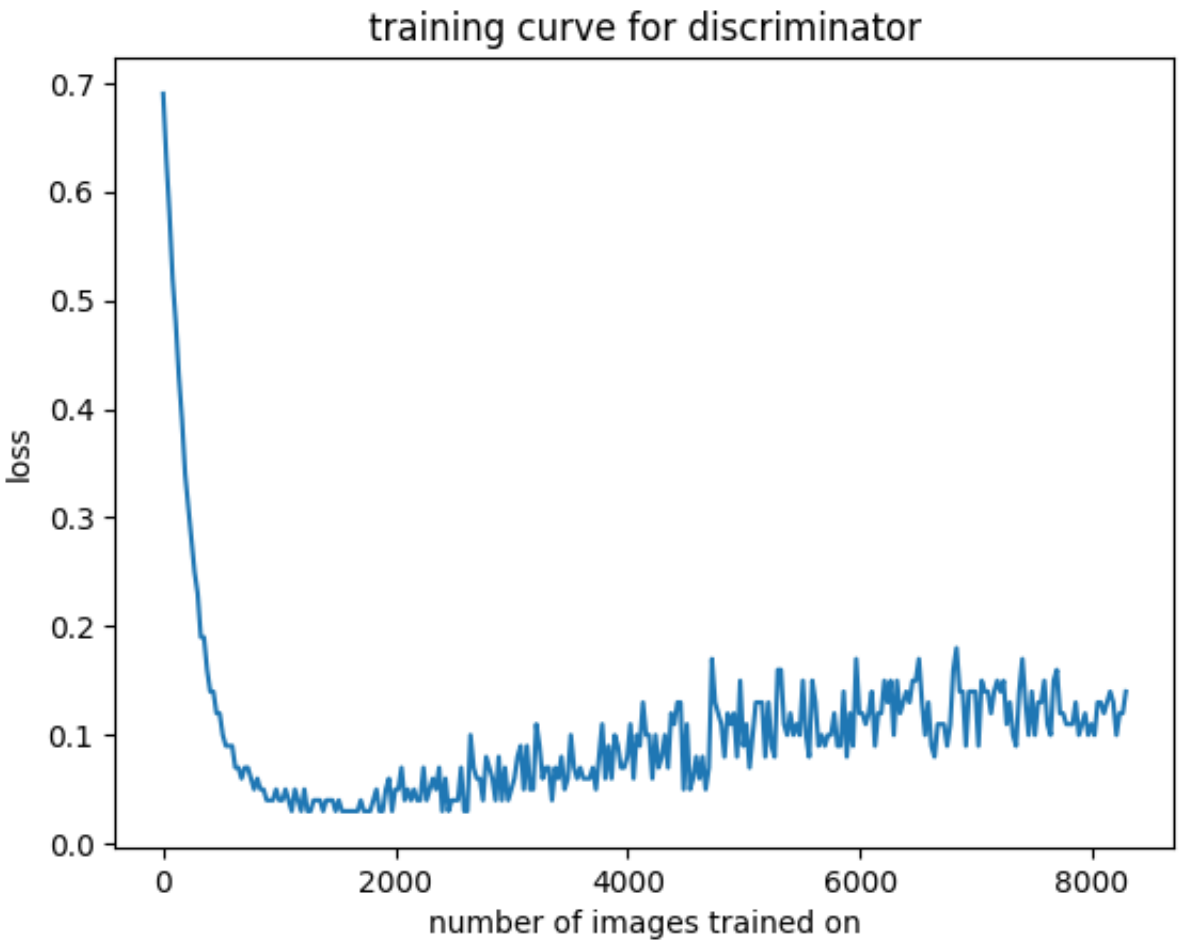


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0
lr:f0.21293085706478135 batch_size:f111 noise_samples:f110
Output of the discriminator given this input: 1.0
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0
lr:f0.21293085706478135 batch_size:f111 noise_samples:f110
1031
100%|██████████| 77/77 [00:59<00:00,  1.29it/s]
100%|██████████| 77/77 [00:54<00:00,  1.41it/s]
100%|██████████| 77/77 [00:54<00:00,  1.41it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
```

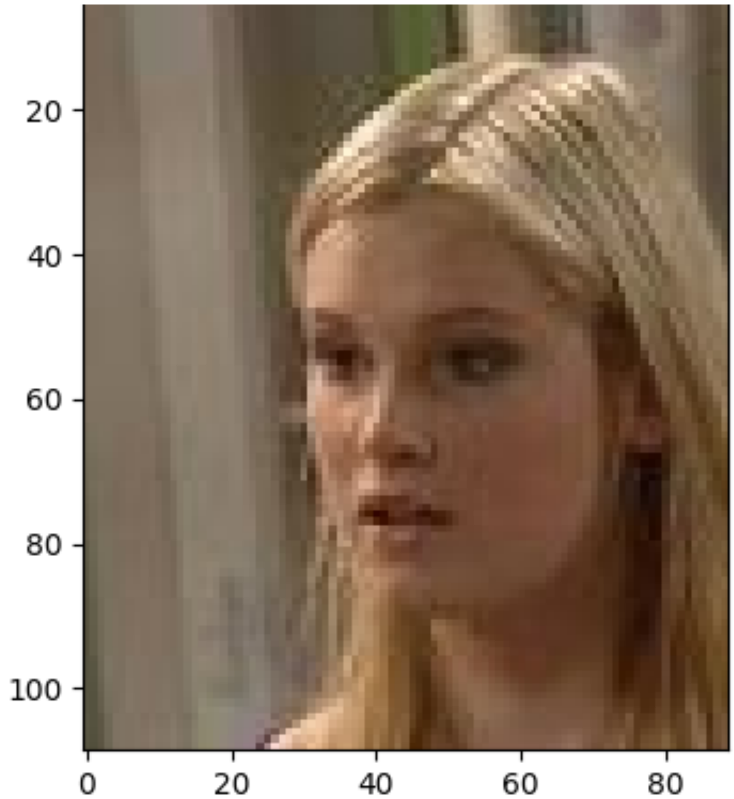


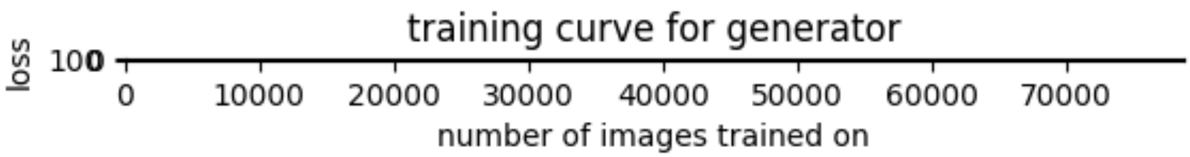

```
Output of the discriminator given this input: 1.0
```

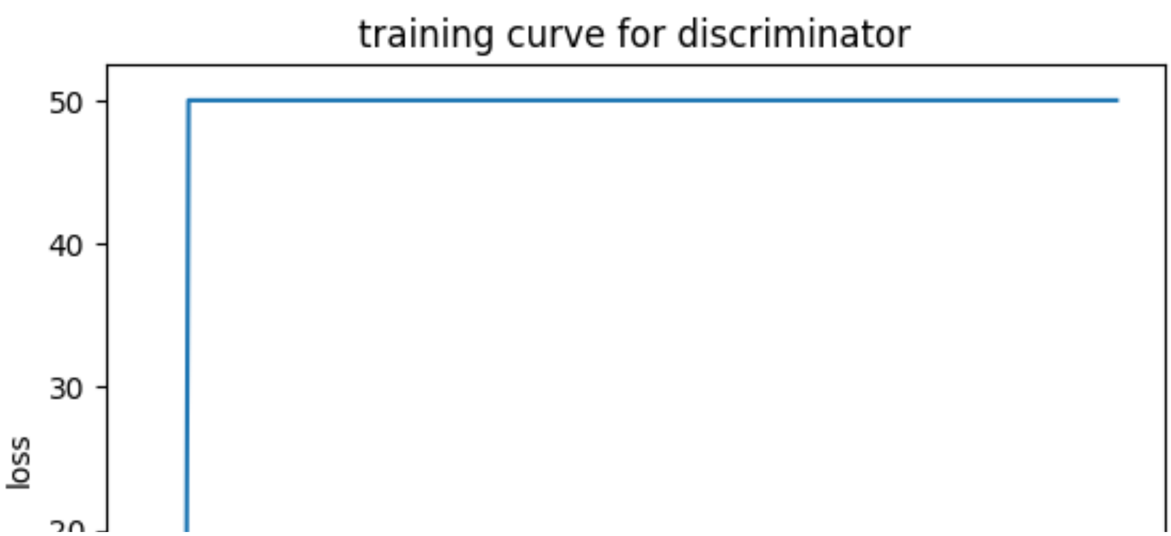
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.08589364
lr:f7.213036669096926e-06 batch_size:f27 noise_samples:f1031
Output of the discriminator given this input: 1.0
```

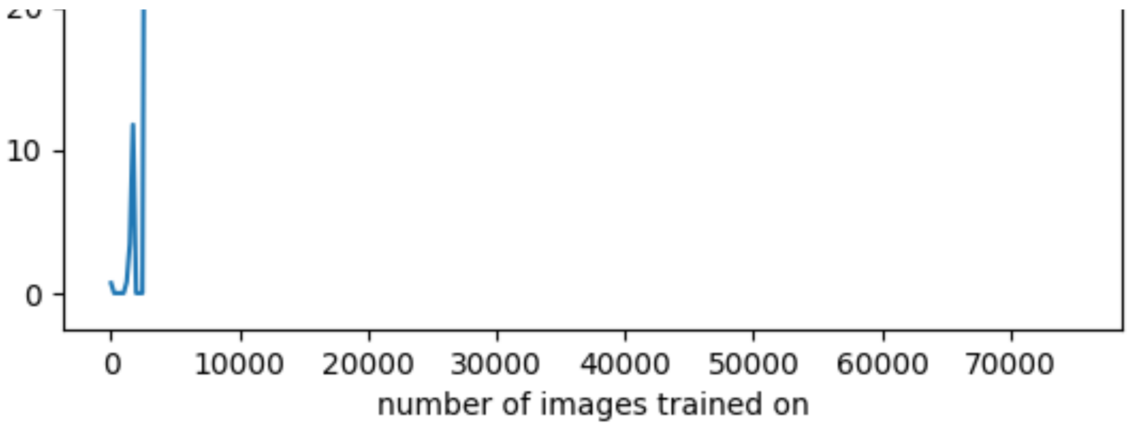
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.069439776
lr:f7.213036669096926e-06 batch_size:f27 noise_samples:f1031
Output of the discriminator given this input: 1.0
```



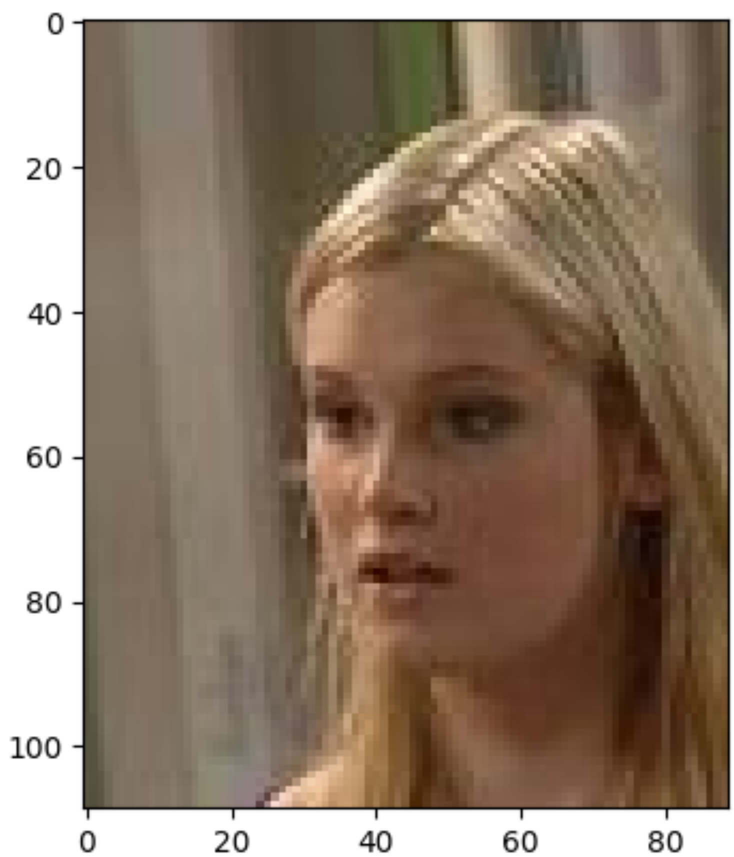

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.069014676
lr:f7.213036669096926e-06 batch_size:f27 noise_samples:f1031
566
100%|██████████| 77/77 [00:57<00:00,  1.35it/s]
100%|██████████| 77/77 [00:54<00:00,  1.41it/s]
100%|██████████| 77/77 [00:55<00:00,  1.39it/s]
100%|██████████| 77/77 [00:54<00:00,  1.40it/s]
```



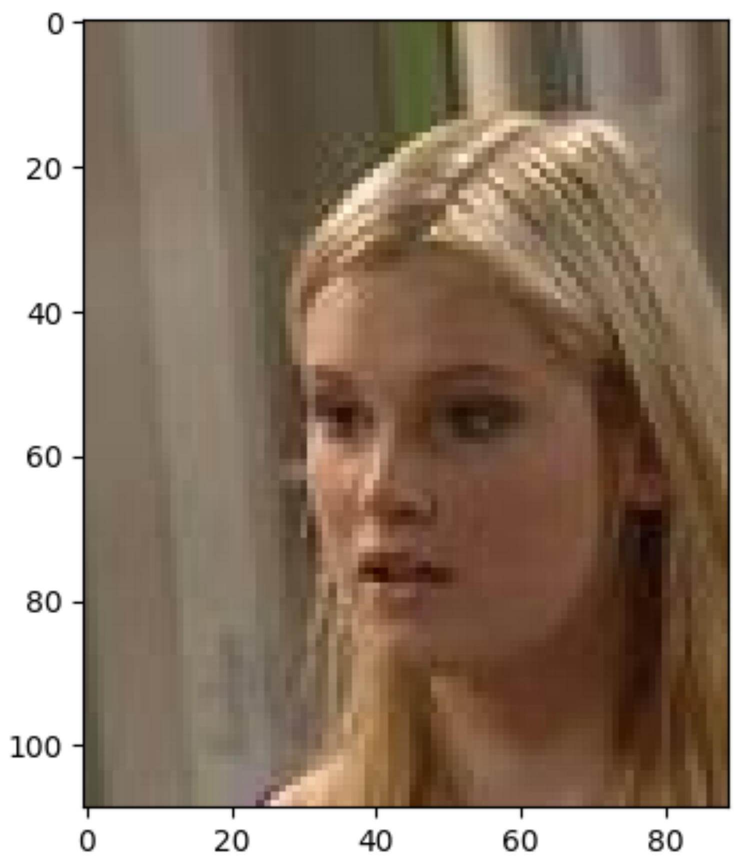



```
Output of the discriminator given this input: 1.0
```



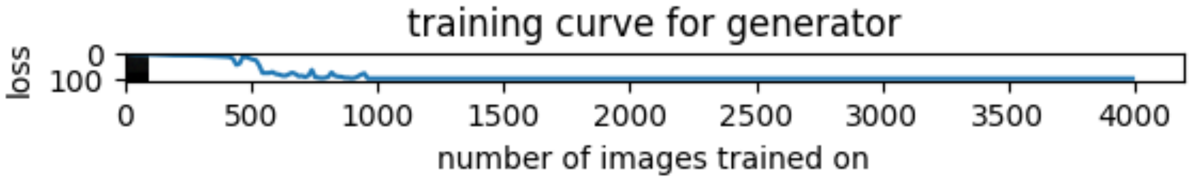
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.0
lr:f0.008836714719693516 batch_size:f244 noise_samples:f566
Output of the discriminator given this input: 1.0
```

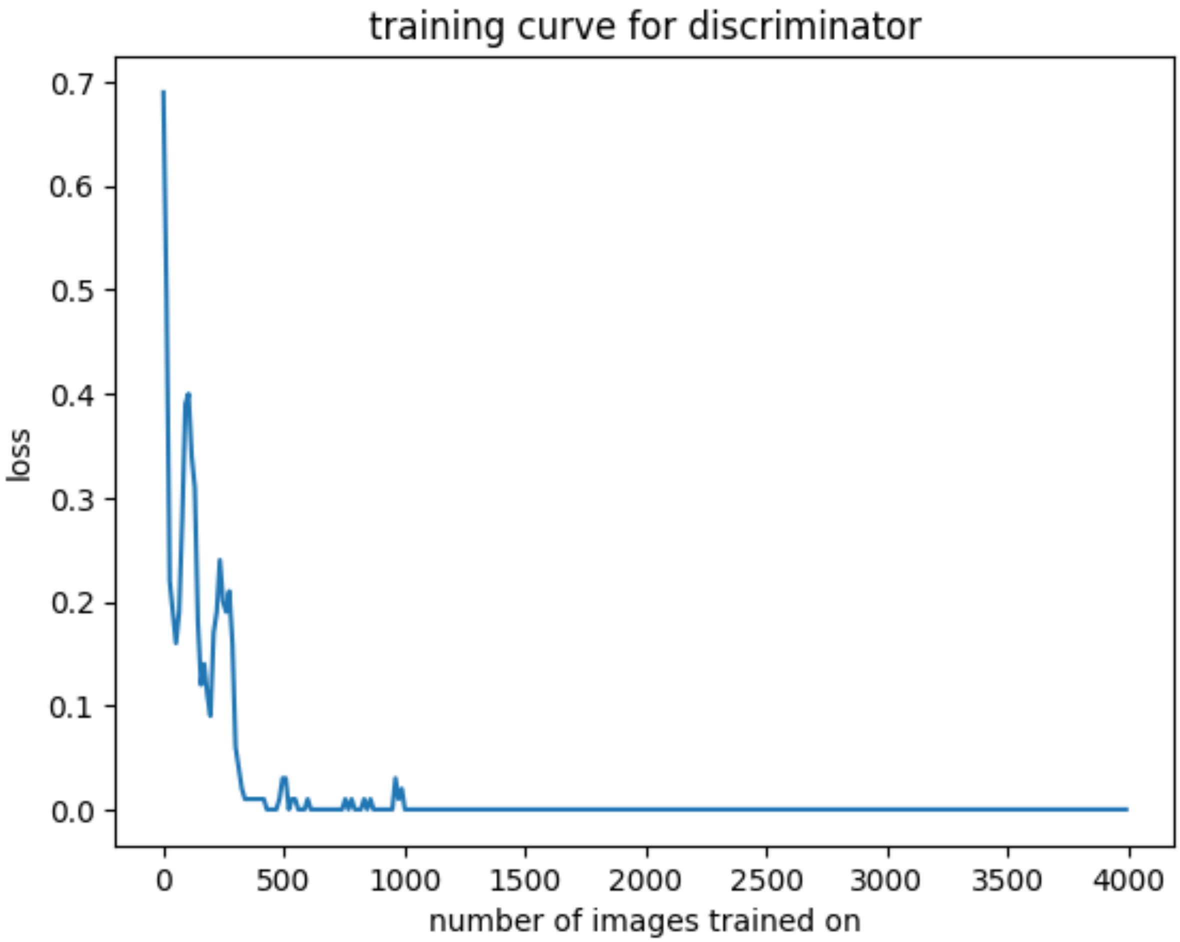

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.0
lr:f0.008836714719693516 batch_size:f244 noise_samples:f566
Output of the discriminator given this input: 1.0
```



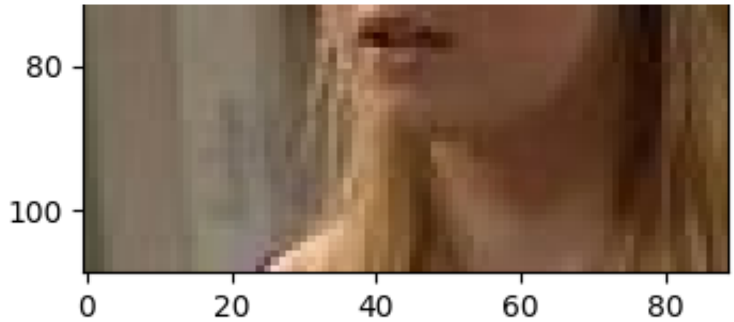

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.0
lr:f0.008836714719693516 batch_size:f244 noise_samples:f566
629
100%|██████████| 77/77 [00:56<00:00,  1.37it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
```

```
100%|██████████| 77/77 [00:54<00:00,  1.41it/s]
100%|██████████| 77/77 [00:54<00:00,  1.40it/s]
```

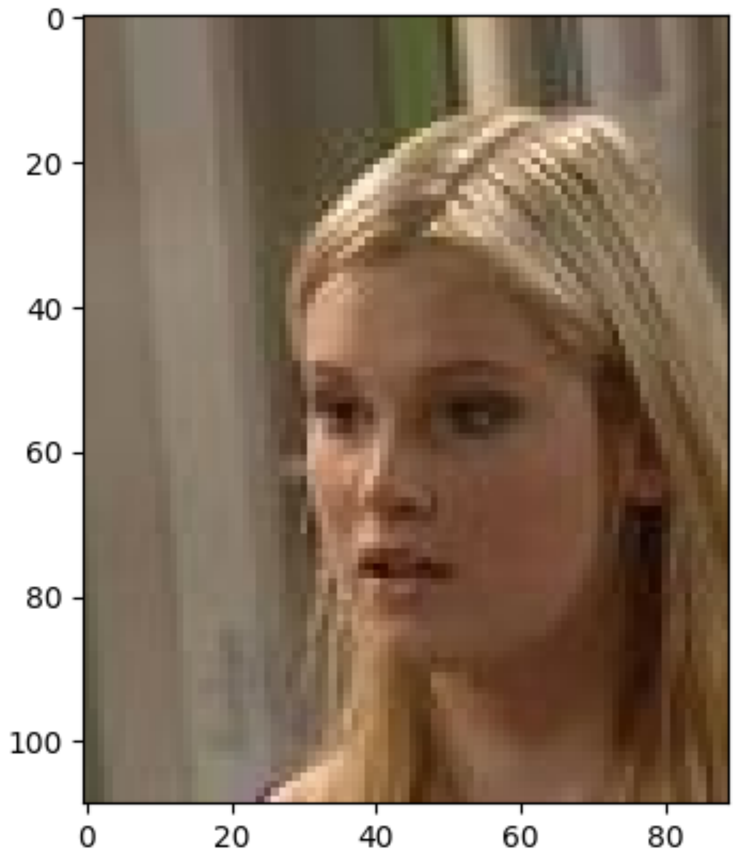



```
Output of the discriminator given this input: 1.0
```

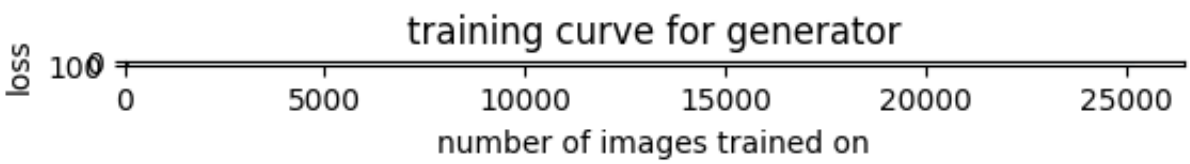


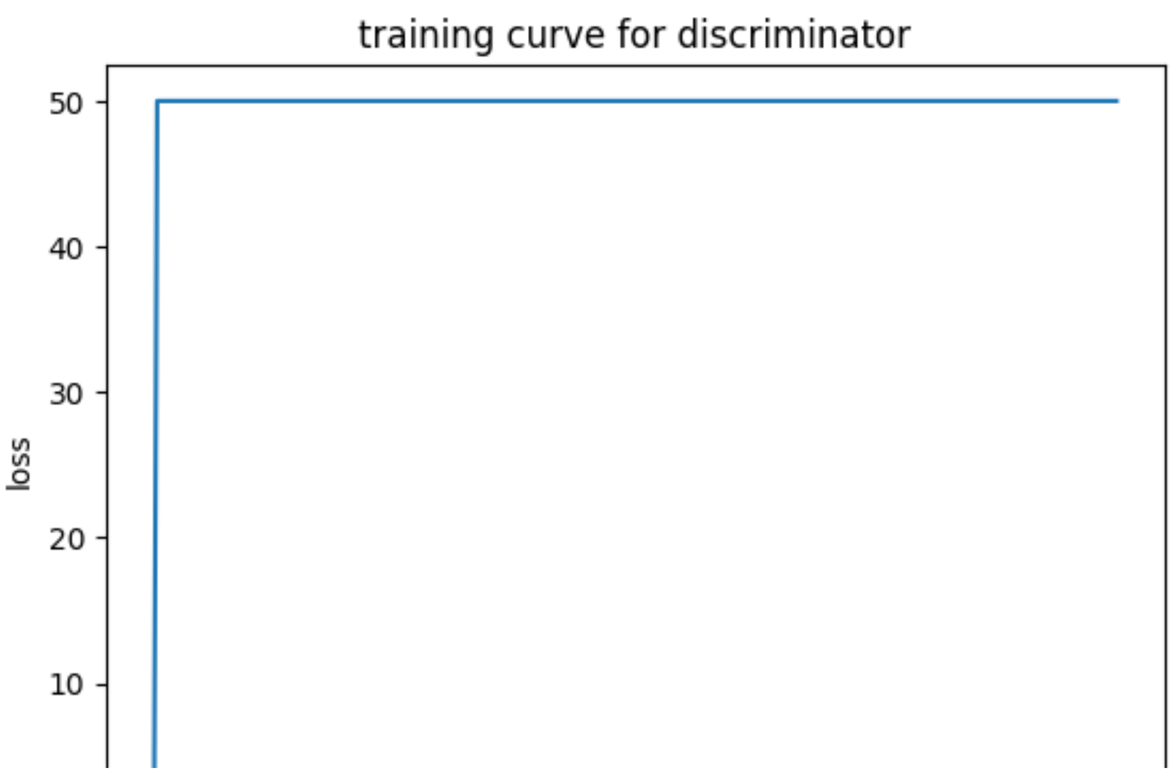
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0
lr:f0.00020013817249032946 batch_size:f13 noise_samples:f629
Output of the discriminator given this input: 1.0
```

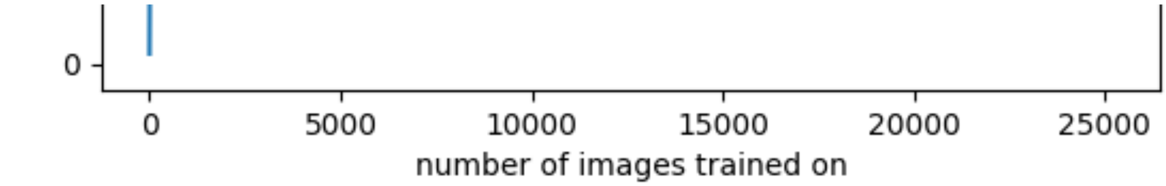


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0
lr:f0.00020013817249032946 batch_size:f13 noise_samples:f629
Output of the discriminator given this input: 1.0
```


```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.0
lr:f0.00020013817249032946 batch_size:f13 noise_samples:f629
488
100%|██████████| 77/77 [02:17<00:00,  1.79s/it]
100%|██████████| 77/77 [05:53<00:00,  4.59s/it]
100%|██████████| 77/77 [02:15<00:00,  1.76s/it]
100%|██████████| 77/77 [04:59<00:00,  3.88s/it]
```

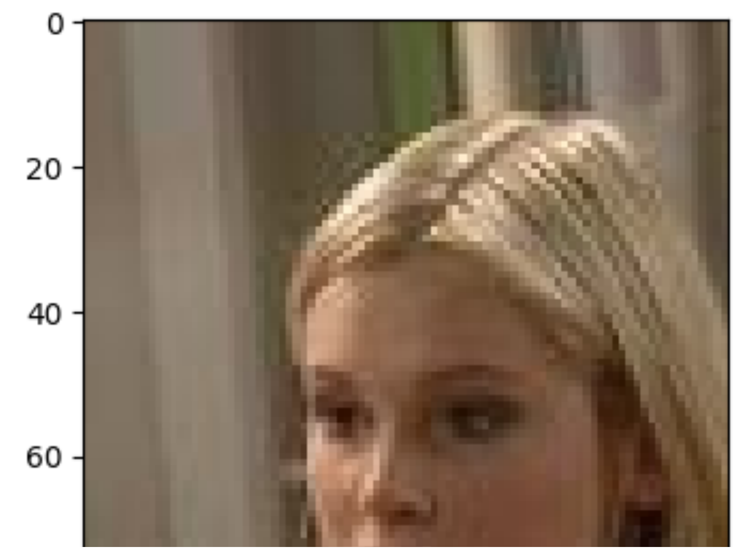


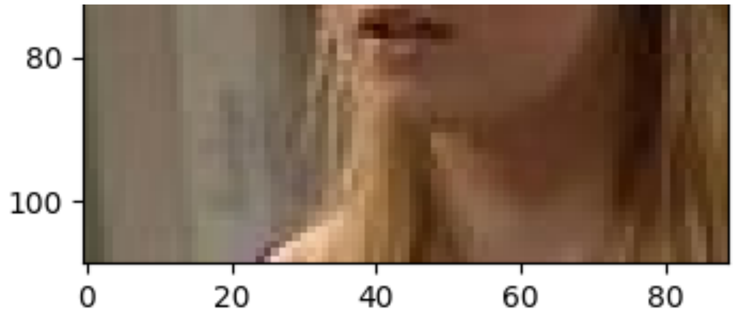






```
Output of the discriminator given this input: 1.0
```

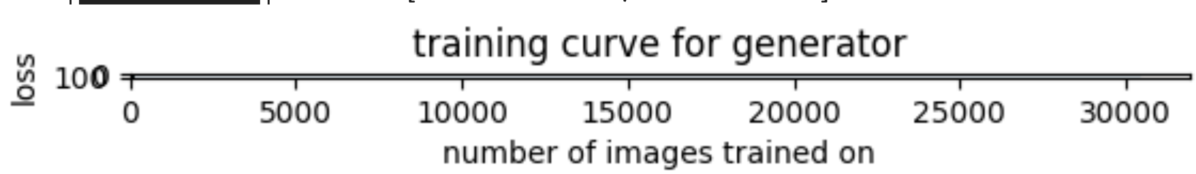

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.0
lr:f1.3183225320228673 batch_size:f82 noise_samples:f488
Output of the discriminator given this input: 1.0
```

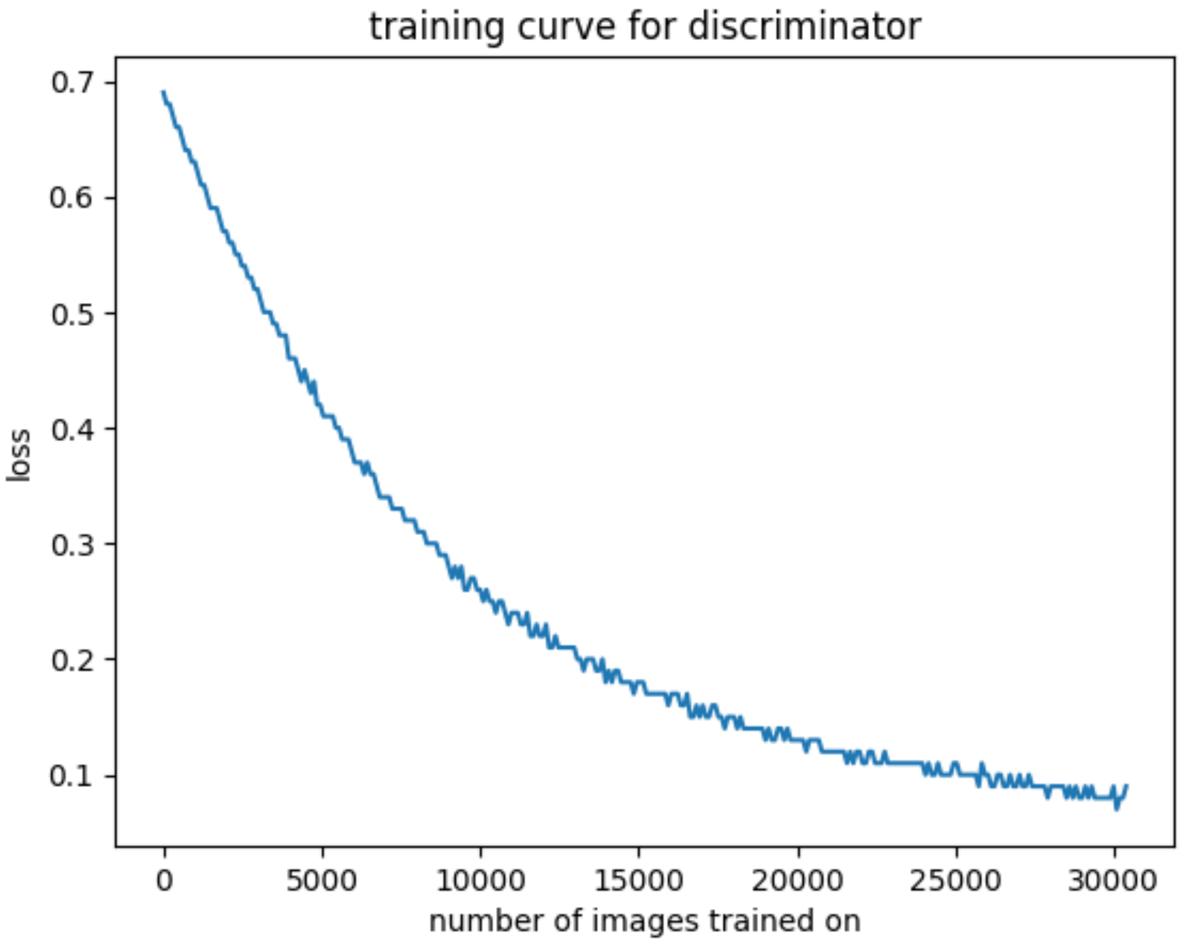




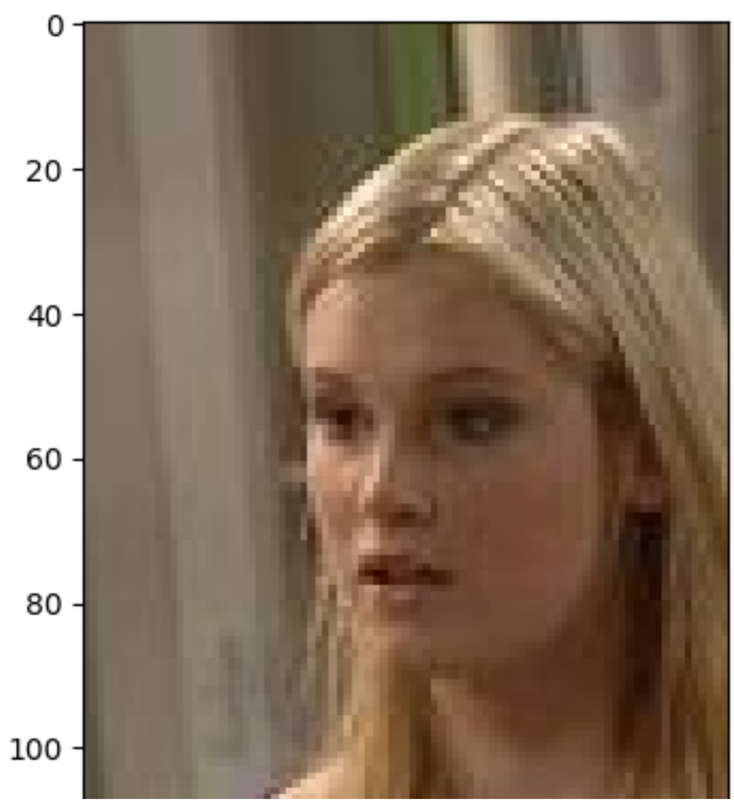
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.0
lr:f1.318322532022867З batch_size:f82 noise_samples:f488
Output of the discriminator given this input: 1.0
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 1.0
lr:f1.3183225320228673 batch_size:f82 noise_samples:f488
398
100%|██████████| 77/77 [06:02<00:00,  4.71s/it]
100%|██████████| 77/77 [00:55<00:00,  1.39it/s]
100%|██████████| 77/77 [00:54<00:00,  1.41it/s]
100%|██████████| 77/77 [00:54<00:00,  1.42it/s]
```

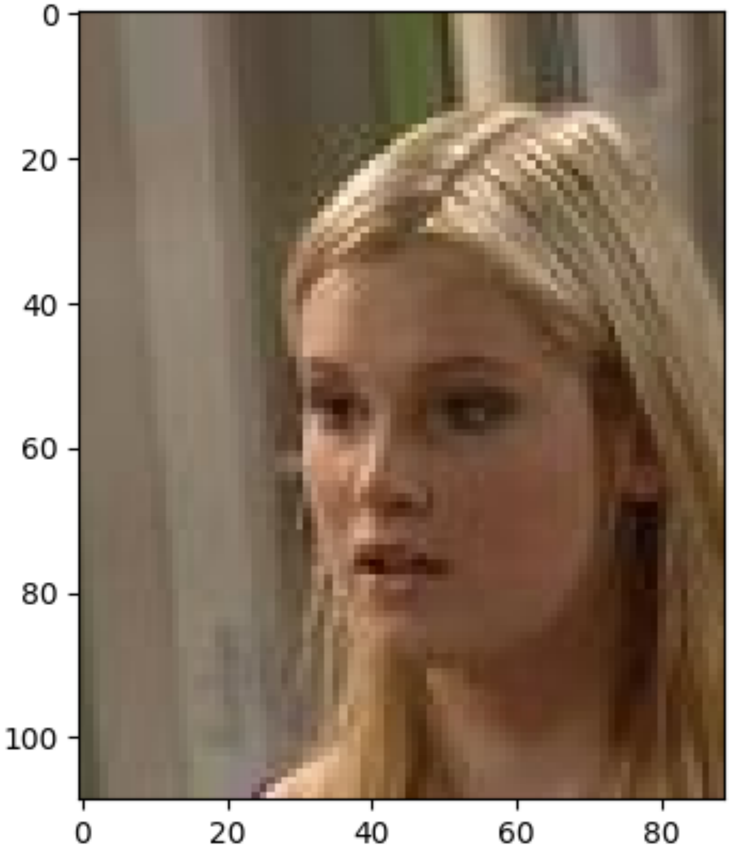




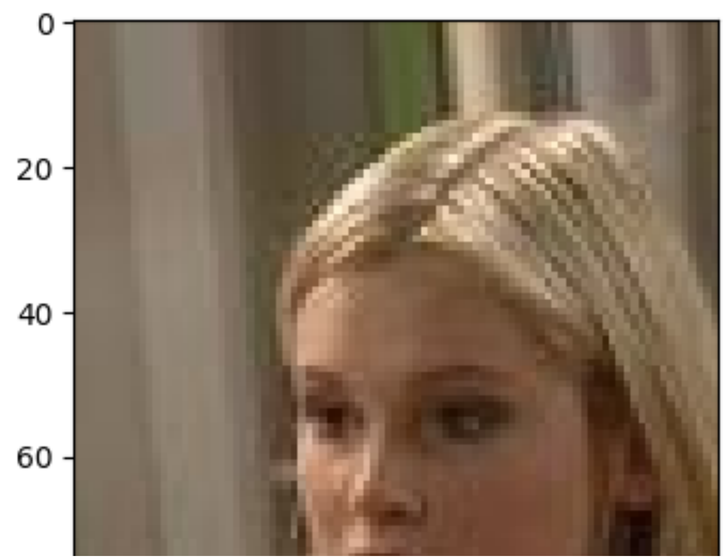

```
Output of the discriminator given this input: 1.0
```

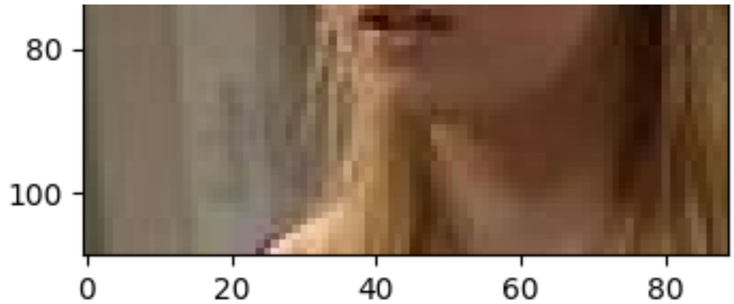



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.10773177
lr:f4.797232847044623e-07 batch_size:f99 noise_samples:f398
Output of the discriminator given this input: 1.0
```

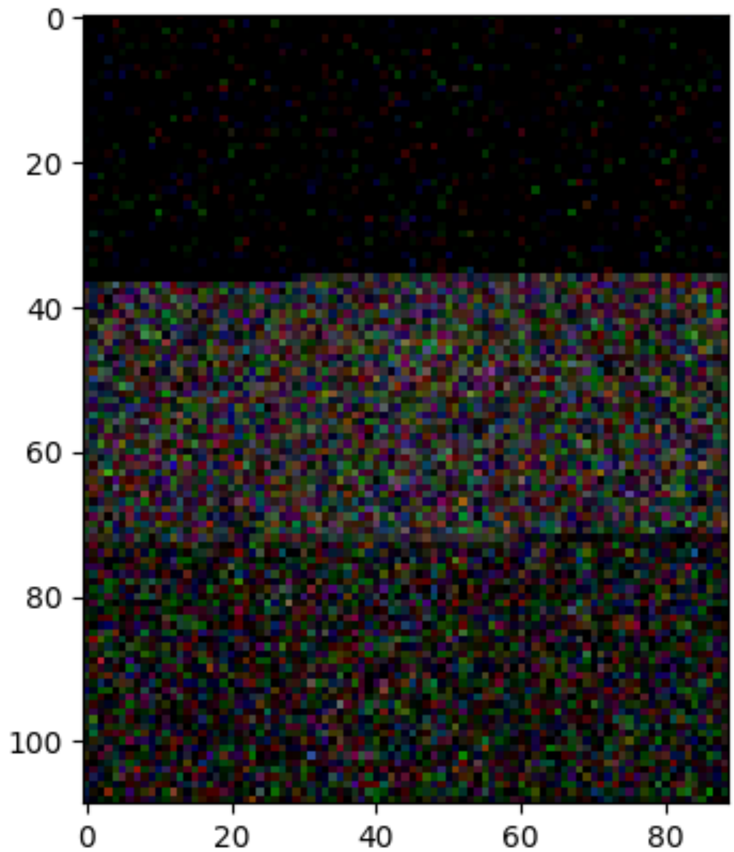



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.10643379
lr:f4.797232847044623e-07 batch_size:f99 noise_samples:f398
Output of the discriminator given this input: 1.0
```



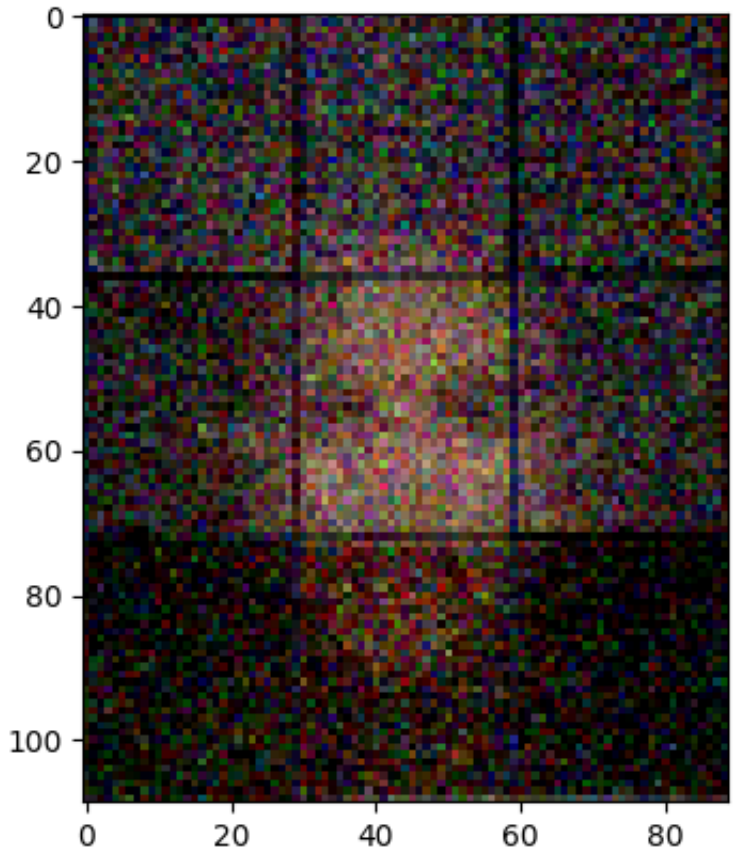



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
Output of the discriminator given this generated input: 0.106940754
lr:f4.797232847044623e-07 batch_size:f99 noise_samples:f398
```



```
# This will show the output of our *best* generator after training
noise = (torch.rand(1, 310) - 0.5) / 0.5
trained_output = G_model(noise)

plot_image(trained_output.detach())
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
```



## GAN Written Report (10 points)

More writing, yay! Hopefully these questions will make you think!

1. Does your trained discriminator learn to correctly discriminate generated examples -- how does yours perform above? What would you guess is the ideal discriminator performance of a trained GAN? Why?

My discriminator preforms better than my generator I think the ideal would be equal preformance where one model doesnt outpace the other in which case both models will be able to reap some reward from learning. too much power on the discriminator and the generator wont have enough reward for taking a step in the right direction because it will still get discriminated as a 0 and too little power to the discriminator and the generator wont learn.

2. Sometimes our generator can produce images that dont look at all like faces (to us) but still fools our discriminator. We can these exmaples *adverserial examples* for our disriminator. Why might our generator produce images like this instead of faces?

Double-click (or enter) to edit

The discrminator looks for certain features and does not look for face the same way a human does so it might get fooled by different things than what a human does, because we too get fooled by things and find them to be faces.

## BONUSES (5 points each)

These are some extra questions that require more code or are just downright hard -- if you're interested in this stuff it could be fun though!

1. Write some code to augment your input to your autoencoder to either be black and white or occluded (part of the image missing) and train it to output the original unaugmented image. Show the results of your model after training below (you can edit whatever code you like to make this work)

2. After getting your generator able to generate faces (if you are able), show what it generates for some particular noise and then iteratively change the noise by a little bit. Does the generator produce faces that are similar for similar inputs?