Selective Coloring

Francisco Noya
fnoya2@illinois.edu

Mesay Taye
mesayst2@illinois.edu

In this project we implemented a tool to selectively and automatically colorize the foreground of black and white photographs. The tool consists of two main algorithms. First, a ViT-LSGAN network colorizes the entire photograph. Second, a segmentation network segments the foreground from the background. Lastly, colorized foreground is blended with the original background to produce a distinctive image.

Approach for the Colorization Network

For the colorization task we worked on the **Lab** colorspace. In this space, the **L** channel contains the luminance or intensity information, while the **ab** channels contain the color information. Therefore, a neural network can be trained with the **L** channel of regular color images as input. Its predictions will be "fake" **ab** channels and its loss will be calculated with the "real" **ab** channels.

After a short bibliographic review we found that although traditional convolutional neural networks (CNNs) could produce results almost indistinguishable from real color photos (Zhang et al., 2016), *Generative Adversarial Networks* or GANs (Goodfellow et al., 2014) were the most proper approach for this kind of problem. This network architecture contains two modules, a Generator and a Discriminator. Both models are trained in paralell. The objective of the generator is to produce outputs similar enough to the ground truth that can fool the discriminator. The discriminator's objective is to properly tell the ground truth from the discriminator output.

Since training this kind of networks requires large datasets and computing time, we decided to use pretrained models that have been used for other tasks such as object classification. We followed a tutorial inspired by the *pix2pix* paper (Isola et al., 2016) but instead of training a naïve *UNet* as the generator, it used a *ResNet18* network as the generator (*Colorizing Black & White Images with u-Net and Conditional GAN — a Tutorial*, n.d.). Similar to *pix2pix* we used a patch discriminator that splits the image in 26 square patches (depending on the image size) and produces a *real* or *fake* prediction for each of them.

To try different approaches, we decided to use *Transformers* in place of the discriminator and the generator. *Transformers* are a special architecture of DNN that make extensive use of attention mechanisms (Vaswani et al., 2017). Because of their ability to have larger receptive fields compared to convolutional neural networks (CNNs), they can track long-range dependencies within an image. These attention based architectures have proven very effective in image processing tasks and gave rise to Visual Transformers or **ViT** (Dosovitskiy et al., 2020). We tried different architectures on our own using ViTs either as generators or discriminators and measured a range of metrics for each of them. Finally, we fine tune the loss function to get the best results.

Dataset

For training and validation purposes we used a subset of the COCO dataset of images (Lin et al., 2014) that is provided by the FastAI framework (*Fast.ai*, n.d.). We downloaded 10.000 images from this dataset and randomly splited them into two sets: a training set with 8.000 images and a validation set with 2.000 images. Then we resized the images so that they have

Final Project for CS445 Computational Photography - Spring 2022.

manageable dimensions that allow feeding into the different network architectures without requiring extremely high computational resources or long times. Similar to (Isola et al., 2016) data augmentation was achieved by flipping the images horizontally (this is only done for the training set). We used 16 images on each batch that goes through the network. Each image was converted to the **Lab** colorspace and the channels adjusted float values between -1 and 1.

Model improvement

Our initial model was a variation of the original *pix2pix* GAN in which the generator was replaced by a pre-trained *ResNet18 UNet*. We then experimented with different visual transfomers as discriminators and generators. Our final model, named **ViT-LSGAN**, uses a ViT as the generator and least squared errors instead of cross entropy as the loss function of the discriminator. The results of all the intermediate models that we trained and tested can be found in **"Supplemental Materials"**.

Loss functions

The loss function of the discriminator for each image is either the binary cross entropy or the least squared errors between the predictions and the ground truth. The ground truth is either *real* if the real **ab** channels were fed into the discriminator, or *fake* if the generated **ab** channels were used instead. The loss function of the generator was the combination of the L1 error and the loss function of the discriminator as if they were *real* **ab** channels. The intuition behind is that the generator "wins" every time it fools the discriminator into assigning *real* predictions to the generated outputs.

Model Training, Transfer and Validation

For implementing these models we worked in Python and Jupyter Notebooks making extensive use of PyTorch and NumPy libraries. All the training was done on a machine equipped with an NVIDIA K80 GPU, 4 vCPUs and 61 GB of RAM (*AWS EC2 p2.xlarge* instance). The transfer learning, validation and metrics calculation were done in an Intel Core i5 8th generation CPU with 8 GB of RAM. For validation we used the 2000 images from the **COCO** dataset that were not used for training, and a set of 70 photographs that were taken by the authors. To assess the different networks architectures we selected the following set of metrics for regression models:

- Correlation coeficient R squared
- Explained variance
- Mean absolute error
- Median absolute error
- Mean squared error

We calculated all these metrics for each one of the **ab** channels.

Approach for image segmentation

For foreground extraction, we first considered traditional approaches including grabcut, graphcut, and combinations thereof. However, when taking into account the fact that a) the categories of foreground objects (portraits) are well identified beforehand b) the accuracy of region segmentation should be optimized even in fairly complex backgrounds c) fully automated foreground segmentation, when possible, is likely to be preferable, we decided that deep neural network based segmentation methods were better candidates for this job.

After conducting literature review and experimentation with various architectures, we chose the Deeplabv3 network (Chen et al., 2017).

What is Deeplabv3?

Deeplabv3 is a deep convolutional network, ResNet-101-based in our case, which introduces a number of important concepts that distinguish it from conventional ConvNets and make it more suitable for semantic segmentation. First, Deeplabv3 employs atrous/dilated convolution for filters. This allows it to leverage wider field of view without increasing the number of parameters, and thus the amount of computation, involved. Second, Deeplabv3 uses atrous spatial pyramid pooling (ASPP) to capture objects at multiple scales. It does so by stacking

together filters of various sampling rates. Figure 1 shows filters with different sampling rates (Chen et al., 2018).

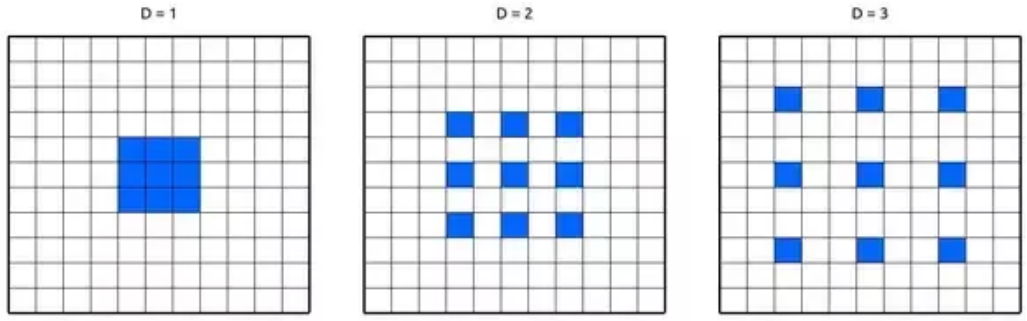

Figure 1: Atrous/dilated convolutions (from https://www.mdpi.com).

When sampling rate (represented by D in the image) is greater than one, the convolution is considered atrous/dilated.

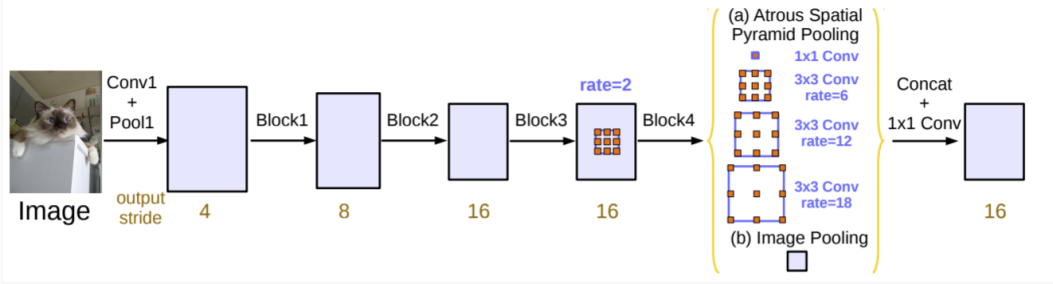

Figure 2: DeeplabV3 architecture (Chen et al., 2017).

The loss function used by the Deeplabv3 is the sum of cross-entropy terms for each spatial position for the output map. Using ImageNet-pretrained ResNet-101 as a backend, and augmenting it with Atrous convolutions followed by ASPP, DeeplabV3 achieves state-of-art segmentation results on PASCAL-Context, PASCAL-Person-Part, and Cityscapes (Chen et al., 2018) (Figure 2). It also performs similarly well on random images as evidenced by the experiments in this project.

Our choice to use Deeplabv3 has been primarily informed by the performance on PASCAL VOC 2012 in terms of pixel intersection-over-union (mIoU). Table 1 shows the relative performance of Deeplabv3 vis-a-vis other comparable models (Chen et al., 2018).

Binary mask generation and image blending

To produce the final blended image, we used the following procedure:

1. Generate a segment map using Deeplabv3
2. Convert the segment map into binary mask
3. Create reverse mask from the binary mask
4. Multiply the colorized input image by the binary mask to get the colorized foreground
5. Multiply the grayscale image by the reverse mask to get the grayscale background
6. Create a new image by adding the foreground and the background images

See details of the code in the **Supplementary Materials** section.

Results

Validation of colorization models

The results of the initial colorization model with a ResNet18 generator were acceptable (Figure 3). However, many times the produced images did not look natural because of an

Table 1: mIoU of segmentation models

Segmentation Method	mIoU
Adelaide VeryDeep FCN VOC	79.1
LRR 4x ResNet-CRF	79.3
DeepLabv2-CRF	79.7
CentraleSupelec Deep G-CRF	80.2
HikSeg COCO	81.4
SegModel	81.8
Deep Layer Cascade (LC)	82.7
TuSimple	83.1
Large Kernel Matters	83.6
Multipath-RefineNet	84.2
ResNet-38 MS COCO	84.9
PSPNet	85.4
IDW-CNN	86.3
CASIA IVA SDN	86.6
DIS	86.8
DeepLabv3	85.7

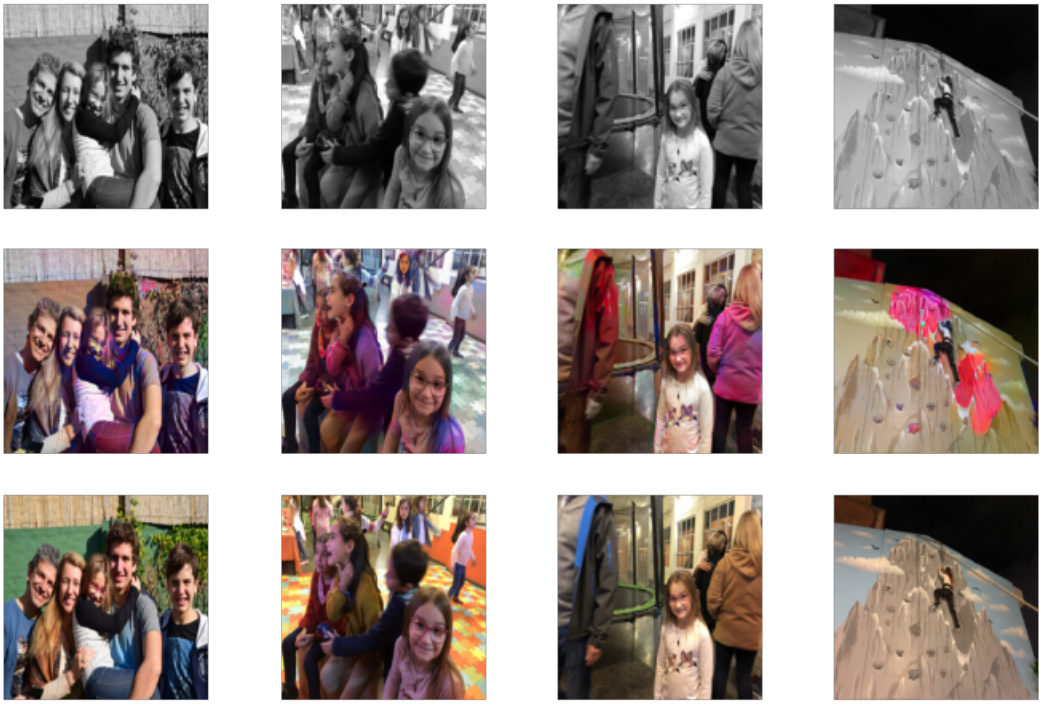

Figure 3: Samples of results from using a ResNet18 network as the generator. Top row: inputs, Middle row: predictions, Bottom row: ground truth.

Table 2: ResNet18 metrics on validation dataset

Metric	a-channel	b-channel
R-square	0.9762	0.7944
Explained variance	0.9763	0.8047
Mean absolute error	0.0221	0.0813
Median absolute error	0.0115	0.0533
Mean squared error	0.0016	0.0144
Sample size	2000	2000

Table 3: GAN architectures trained and validated in this work.

GAN architecture	Summary of results
ResNet18 UNet as generator	Patches of colors, modest b-channel predictions.
ViT as discriminator	Gray images.
ViT as generator	Gray images.
ViT-LSGAN	Good natural colorization, best metrics for ab-channels.

excessive use of colors that resulted in blotches in the pictures. In agreement with the visual inspection, the resulting metrics on the validation dataset (Table 2) showed that the network did a pretty good job at predicting the **a**-channel with over 97% of the variance of the channel predicted by the model with a very low mean squared error. However, the prediction on the **b**-channel was not as good with the model predicting only 80% of its variance.

As explained before, to improve this results we tried variations of the GAN using *Transformers*. We experimented with the architectures summarized in Table 3. The description of each one and their results can be found in the **Supplementary Materials** section.

In particular, the results from ViT-LSGAN model were very encouraging. We got good results on our own photographies (Figure 4) as well as in the COCO dataset ones (Figure 5). By using visual transformers we got an improvement over the initial UNet based model on every metric in particular in the **b**-channel which was the most difficult to predict (Table 4). For example, this model was able to explain 98% of the variance of the **a**-channel and 86% of the variance of the **b**-channel, against 97% and 80%, respectively, when using the UNet model.

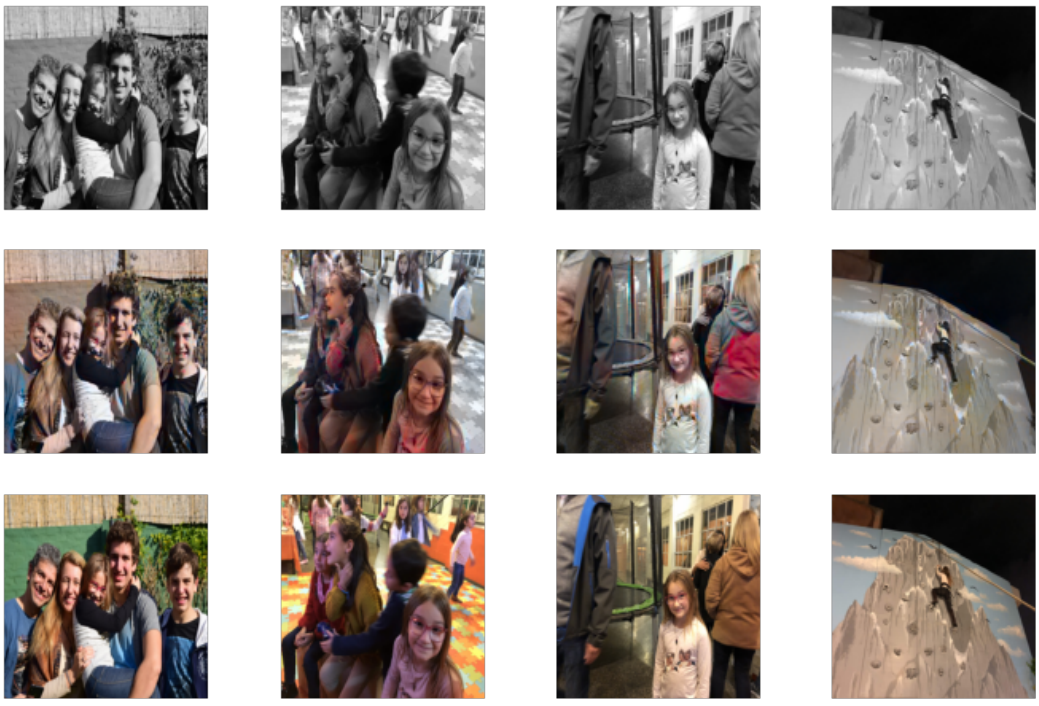

Figure 4: Samples of results of the ViT-LSGAN model. Top row: inputs, Middle row: predictions, Bottom row: ground truth.

Final colorization results

We tested our ViT-LSGAN with historic black and white urban pictures as well as with historic portraits (Figures 6 and 7). We noticed that sometimes old photographs are saturated, particularly, in the sky area. This causes the LSGAN to interpret them as cloudy skies and

Figure 5: COCO images from the validation set colorized with the final ViT-LSGAN model. Top row: inputs, Middle row: predictions, Bottom row: ground truth.

Table 4: ViT metrics on validation dataset

Metric	a-channel	b-channel
R-square	0.9856	0.8596
Explained variance	0.9859	0.8620
Mean absolute error	0.0157	0.0625
Median absolute error	0.0075	0.0374
Mean squared error	0.0010	0.0095
Sample size	2000	2000

producing a white sky. This effect can be partially overcome by adjusting the gain of the original photo before feeding it into the network.

Segmentation and blending

After colorizing the images, we segmented the foreground from the background using the DeepLabv3 network. To obtain the final images we blended the colorized foreground with the original black and white background (Figures 8-14).

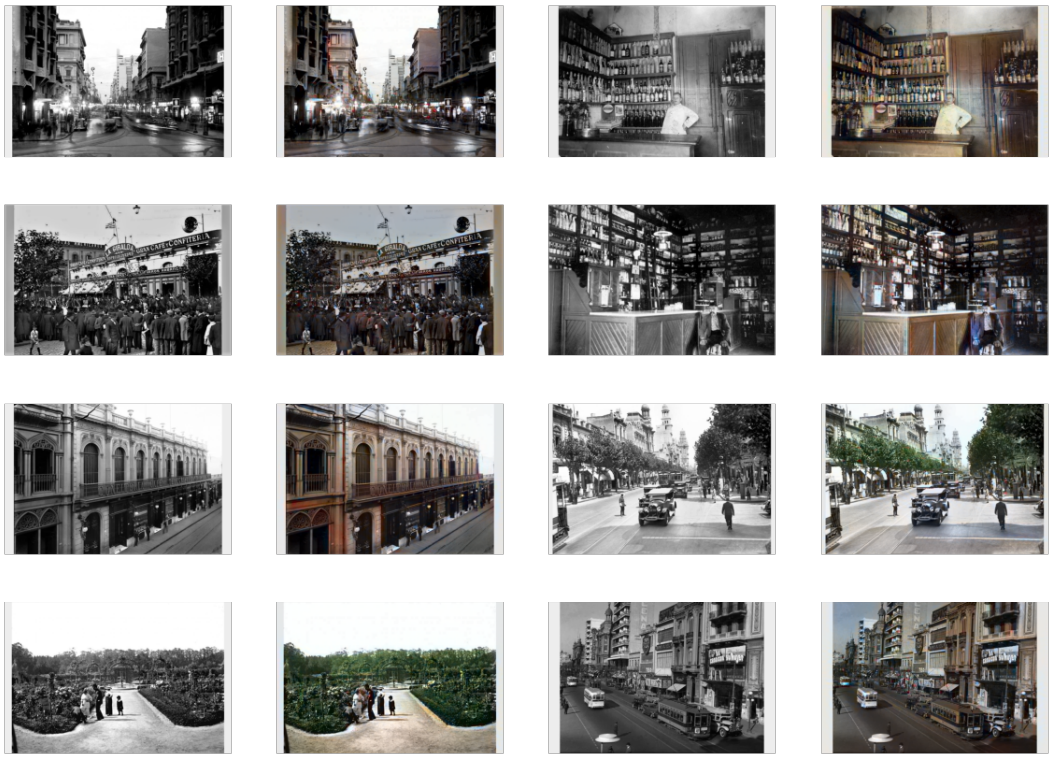

Figure 6: Historic photos of Montevideo.

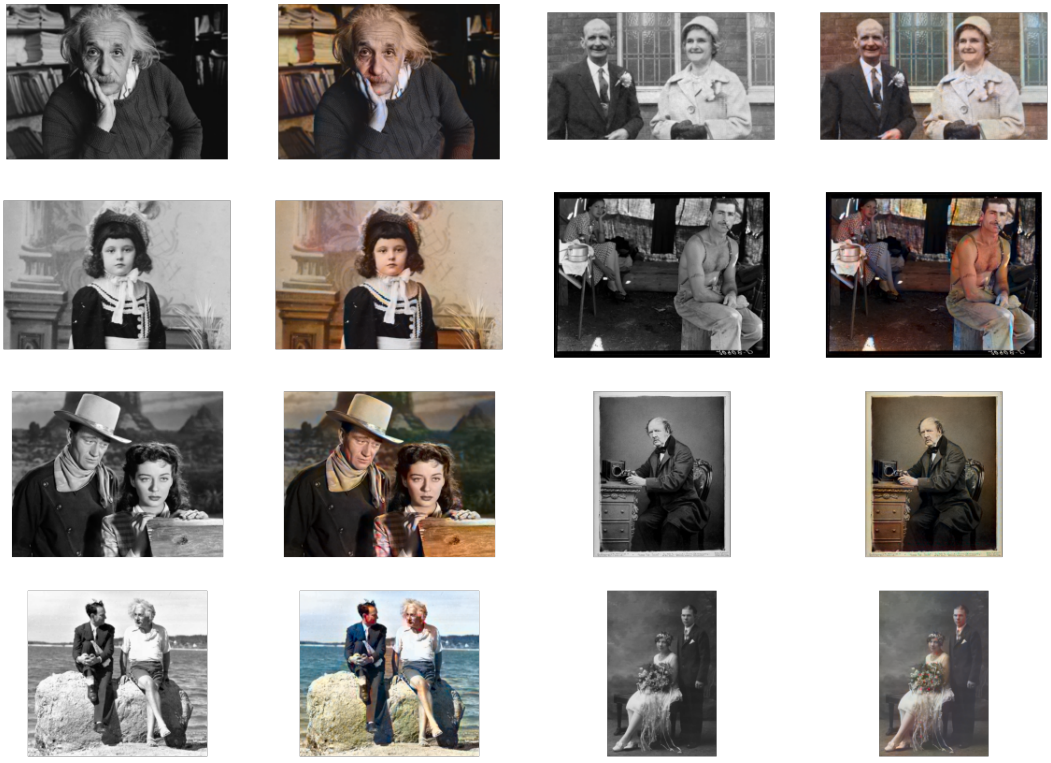

Figure 7: Historic portraits.

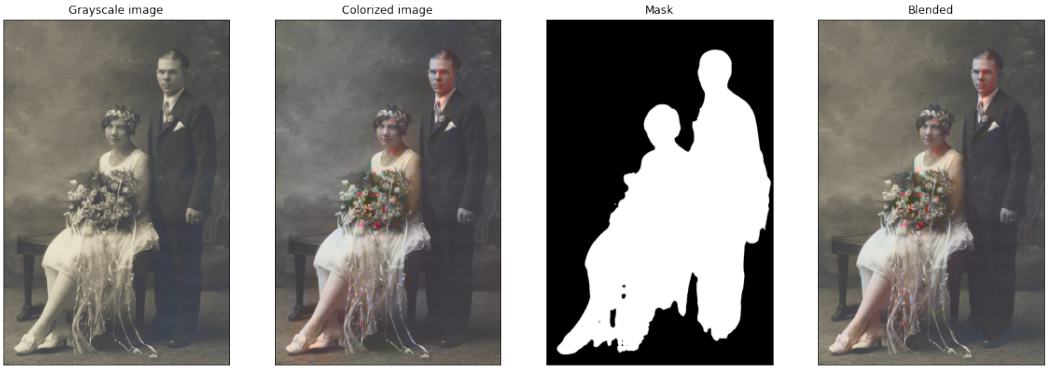

Figure 8: Recolorization and segmentation example 1.

Figure 9: Recolorization and segmentation example 2.

Figure 10: Recolorization and segmentation example 3.

Figure 11: Recolorization and segmentation example 4.

Figure 12: Recolorization and segmentation example 5.

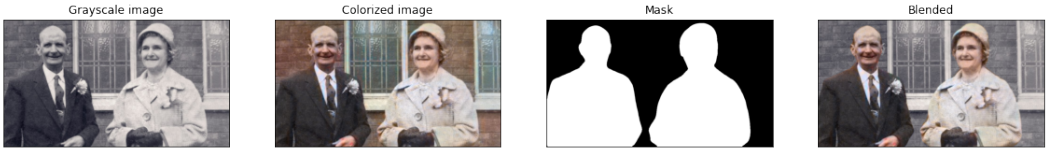

Figure 13: Recolorization and segmentation example 6.

Figure 14: Recolorization and segmentation example 7.

Supplemental Materials

Colorization models trained and tested

First approach: *ResNet18* generator

On our first approach we employed a generator based on the ResNet18 network. One of the challenges of GANs is that, at the beginning of the training, the task of the discriminator is much easier than that of the generator because the generated outputs are very different from the real ones. In this situation, the discriminator learns so much faster and gives no time to the generator to adapt. To avoid this, we gave the generator a *head start* by training it alone (without the generator) for 20 epochs with a L1 loss function and saving its weights. After that we started the parallel training of the generator and the patch discriminator for another 20 epochs (see Figure 3 for results).

Second approach: ViT as discriminator

Since the results obtained from the UNet generator did not look quite natural, our first intention was to improve the discriminator so that it will be better at telling apart the *real* from the *fake* images. To do that we decided to replace the CNN based Patch Discriminator with a Visual Transformer.

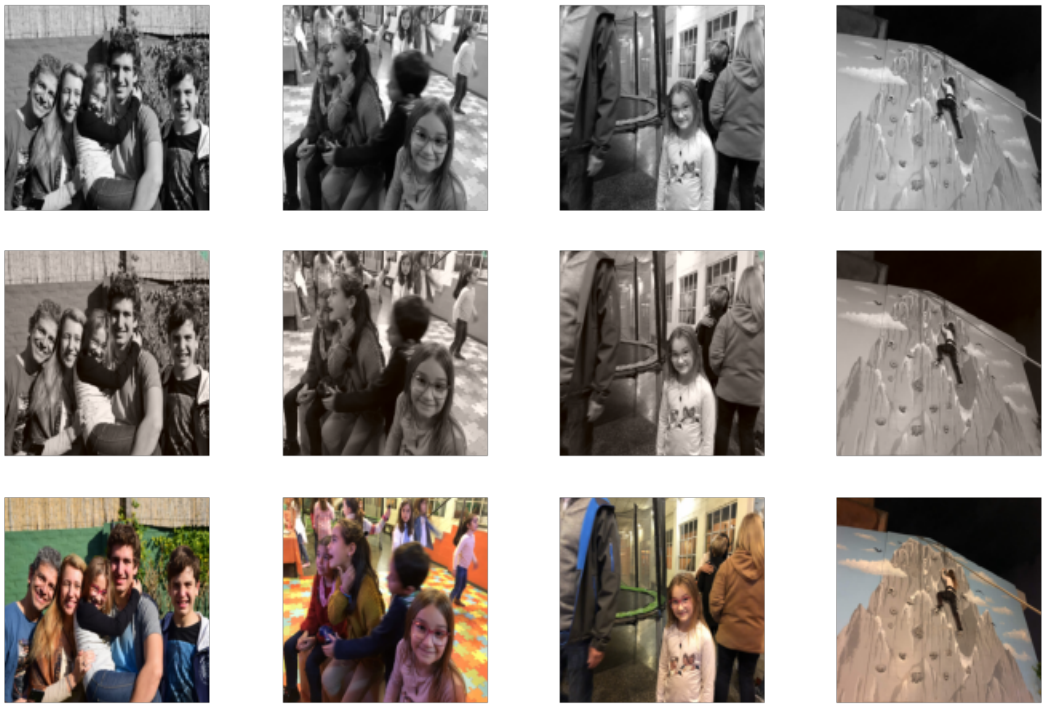

Figure 15: Samples of results from using a ViT as the discriminator network. Top row: inputs, Middle row: predictions, Bottom row: ground truth.

Unfortunately, the results were not satisfactory (Figure 15). The discriminator got very good a discriminating real from fake very early on and gave not chance for the generator to adapt. The final results are just gray images or sepia looking images with almost no color. This is because the best loss the generator could achieve was by producing an average value on the **ab** channels disregarding of the inputs.

Third approach: ViT as generator

Once we understood that to have a truly creative network we should put our efforts on the generator instead the discriminator. We decided to include a ViT as the generator. After exploring different options we selected a ViT trained in the task of completing masked images (Xie et al., 2021). Since this model expects a 3-channels input, we mimicked it just by copying the **L** channel into each of the input channels. We replaced the decoder

block of this model by 3 convolution layers and activation functions. These layers converted the 14x14x768 inputs of the encoder into 14x14x512 outputs. A pixel shuffle layer with an upscale factor of 16 reshaped those outputs into the final 224x224x2 output. To avoid the problem of the discriminator learning too fast, we kept the pretraining step with the generator alone to give a *head start* to it. We trained for 20 epochs.

```
# Build transformer based generator
# https://huggingface.co/docs/transformers/model_doc/vit

from transformers import ViTForMaskedImageModeling, ViTConfig


def build_VTi_generator():

    ## Pretrained generator ViT, 3-input channels, multilayer decoder.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = ViTForMaskedImageModeling.from_pretrained("google/vit-base-patch16-224-in21k")

    model.decoder = nn.Sequential(nn.Conv2d(768, 768, kernel_size=3, stride=1, padding=1), \
                                  nn.ReLU(inplace=True),
                                  nn.Conv2d(768, 768, kernel_size=3, stride=1, padding=1), \
                                  nn.ReLU(inplace=True),
                                  nn.Conv2d(768, 512, kernel_size=3, stride=1, padding=1), \
                                  nn.PixelShuffle(upscale_factor=16))
    model = model.to(device)
    return model
```

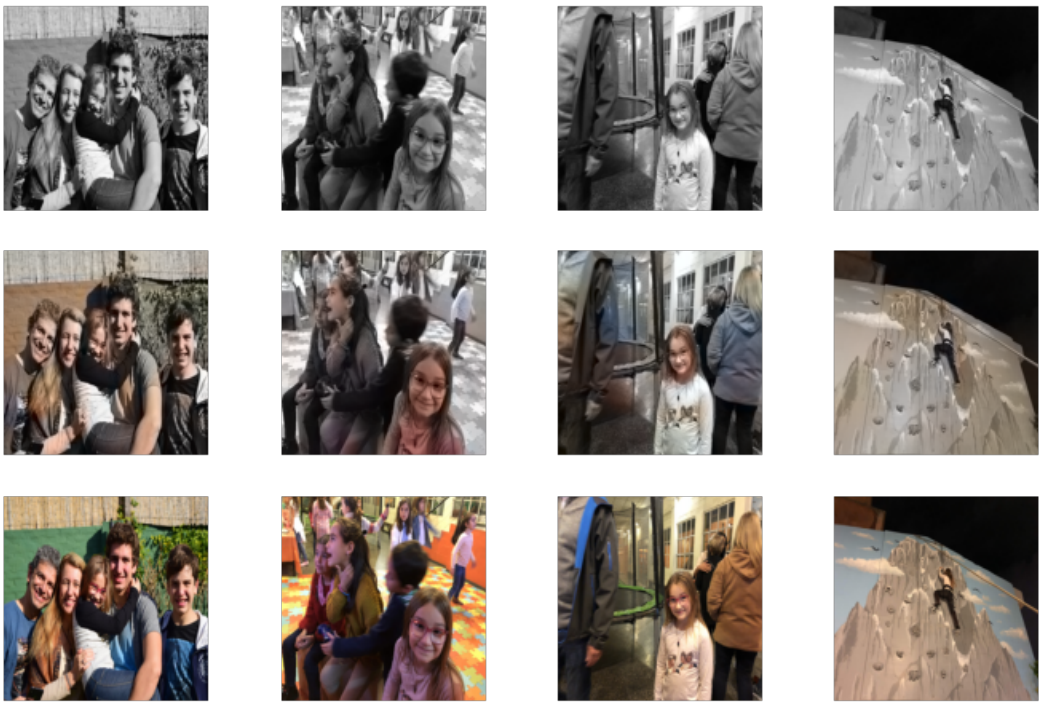

Figure 16: Samples of results from pretraining the ViT generator alone. Top row: inputs, Middle row: predictions, Bottom row: ground truth.

The results were not very encouraging. After the pretraining when the generator trained alone, the results were acceptable although the colors were not very bright or varied (Figure 16). However, when we trained with the discriminator, the discriminator won the game and the generator produced just gray images, as before (Figure 17).

Fourth approach: LSGAN with ViT

The other main challenge of training a GAN is choosing the right loss function. During training of regular networks a convergence of the loss function to a small enough value signals that the network has achieved an equilibrium and cannot learn more from the training data. However, training of a GAN is a two players game in which each one tries to minimize its

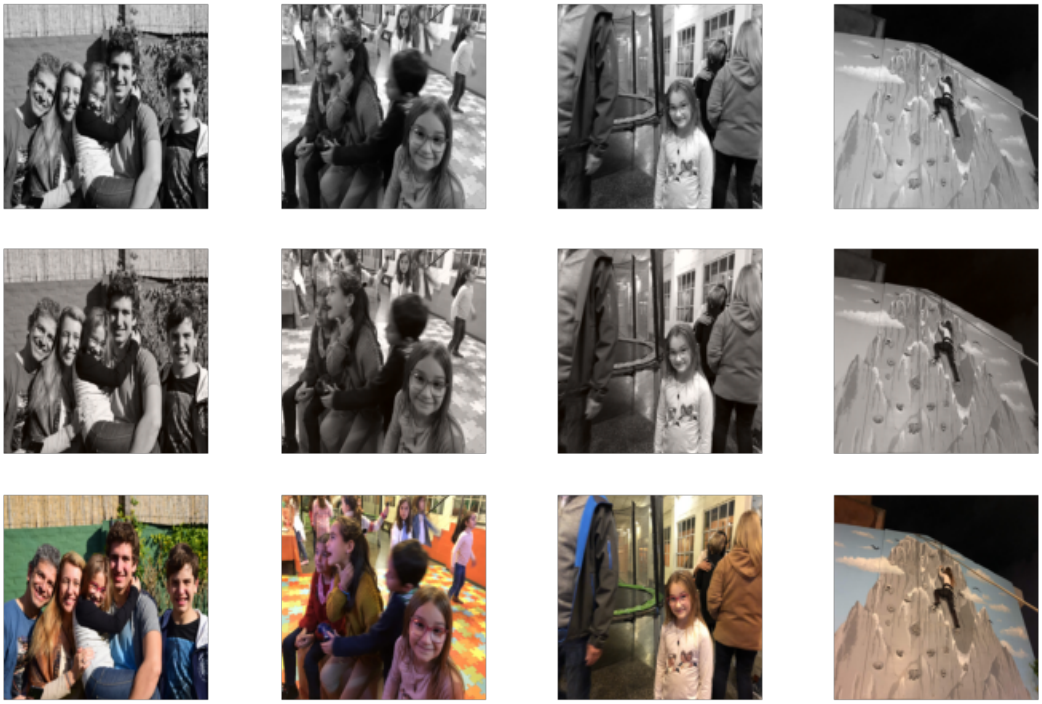

Figure 17: Samples of results after training the ViT generator with the discriminator. Top row: inputs, Middle row: predictions, Bottom row: ground truth.

own loss by maximizing the other player loss. The generator and discriminator losses should not converge but stay in a permanent unstable equilibrium which signals that the game is still being played.

The loss function is what gives the gradient the generator needs to learn to fool the discriminator and not all loss functions are equal for this task. As already mentioned, at the beginning of the training it is very easy for the discriminator to tell fake from real. When Cross Entropy is used, it can provide very low or vanishing gradients at the start of the training that do not help the improvement of the generator. To overcome this problem, it has been suggested the use of least squared errors loss functions (Mao et al., 2016). Therefore, we replaced the BCE loss with least square errors loss to construct a **ViT-LSGAN**. This architecture gave the best results. The results and metrics obtained with this model were presented before (Figures 4 and 5, Table 4).

Image Upscaling

The ViT-LSGAN was trained with 224 by 224 images, so it is better to use downsampled images for transfering color. To colorize the full size image, we upscaled the outputs of the network by resizing the predicted **ab** channels to the size of the original **L** channel, and combined the results with the **L** channel to produce a color image in the **Lab** colorspace.

Selected Python code sections

Colorization Model class

```
class MainModel(nn.Module):
    def __init__(self, net_G=None, net_D=None, use_ViT_gen = False, lr_G=2e-4, lr_D=2e-4,
                 beta1=0.5, beta2=0.999, lambda_L1=100.):
        super().__init__()

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.lambda_L1 = lambda_L1
        self.use_ViT_gen = use_ViT_gen

        if net_G is None:
            raise NotImplementedError
        else:
            self.net_G = net_G.to(self.device)

        if net_D is None:
            self.net_D = init_model(PatchDiscriminator(input_c=3, n_down=3, num_filters=64), self.device)
        else:
            self.net_D = net_D.to(self.device)

        #self.GANcriterion = GANLoss(gan_mode='vanilla').to(self.device)  # Original BCE Loss
        self.GANcriterion = GANLoss(gan_mode='lsgan').to(self.device)  # Final improvement with Least
        ↪  Square Error loss
        self.L1criterion = nn.L1Loss()
        self.opt_G = optim.Adam(self.net_G.parameters(), lr=lr_G, betas=(beta1, beta2))
        self.opt_D = optim.Adam(self.net_D.parameters(), lr=lr_D, betas=(beta1, beta2))

    def set_requires_grad(self, model, requires_grad=True):
        for p in model.parameters():
            p.requires_grad = requires_grad

    def setup_input(self, data):
        self.L = data['L'].to(self.device)
        self.ab = data['ab'].to(self.device)

    def forward(self):
        if (self.use_ViT_gen == True):
            outputs = self.net_G(self.L.repeat(1,3,1,1))  # Copy the L channel 3 times to mimick the
↪   3-channels input that the pretrained network requires.
            self.fake_color = outputs.logits
        else:
            self.fake_color = self.net_G(self.L)

    def backward_D(self):
        fake_image = torch.cat([self.L, self.fake_color], dim=1)
        fake_preds = self.net_D(fake_image.detach())
        self.loss_D_fake = self.GANcriterion(fake_preds, False)
        real_image = torch.cat([self.L, self.ab], dim=1)
        real_preds = self.net_D(real_image)
        self.loss_D_real = self.GANcriterion(real_preds, True)
        self.loss_D = (self.loss_D_fake + self.loss_D_real) * 0.5
        self.loss_D.backward()

    def backward_G(self):
        fake_image = torch.cat([self.L, self.fake_color], dim=1)
        fake_preds = self.net_D(fake_image)
        self.loss_G_GAN = self.GANcriterion(fake_preds, True)
        self.loss_G_L1 = self.L1criterion(self.fake_color, self.ab) * self.lambda_L1
        self.loss_G = self.loss_G_GAN + self.loss_G_L1
        self.loss_G.backward()

    def optimize(self):
        self.forward()
        self.net_D.train()
        self.set_requires_grad(self.net_D, True)
        self.opt_D.zero_grad()
        self.backward_D()
        self.opt_D.step()

        self.net_G.train()
        self.set_requires_grad(self.net_D, False)
        self.opt_G.zero_grad()
        self.backward_G()
        self.opt_G.step()
```

DeepLabv3 based segmentation

```
from torchvision import models
import torchvision.transforms as T
import torch
import cv2
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

dlv3 = models.segmentation.deeplabv3_resnet101(pretrained=1).eval()
n_classes = 21
palette = torch.tensor([2 ** 25 - 1, 2 ** 15 - 1, 2 ** 21 - 1])
colors = torch.as_tensor([i for i in range(n_classes)])[:, None] * palette

grayscale_images = (
    imgs_grayscale_1,
    imgs_grayscale_2,
    imgs_grayscale_3,
    imgs_grayscale_4,
    imgs_grayscale_5,
    imgs_grayscale_6,
    imgs_grayscale_7,
    imgs_grayscale_8,
)


colorized_images = (
    imgs_color_1,
    imgs_color_2,
    imgs_color_3,
    imgs_color_4,
    imgs_color_5,
    imgs_color_6,
    imgs_color_7,
    imgs_color_8,
)

def infer_segment_color(img):
  [r,g,b] = 3 * [np.zeros_like(img).astype(np.uint8)]
  for i in range(n_classes):
    idx = (img == i)
    r[idx],  g[idx],  b[idx] =  colors[i, 0],  colors[i, 1], colors[i, 2]
  rgb = np.stack([r, g, b], axis=2)
  return rgb

def generate_mask(img):
  transform = T.Compose([
                    T.ToTensor(),
                    T.Normalize(mean = [0.485, 0.456, 0.406],  std = [0.229, 0.224, 0.225])])
  output = dlv3(transform(img).unsqueeze(0))['out']
  mask = torch.argmax(output.squeeze(), dim=0).detach().cpu().numpy()
  mask = infer_segment_color(mask)
  print("Mask shape", mask.shape)
  return mask

masks = {}
masks[0] = generate_mask(imgs_grayscale_1)
masks[1] = generate_mask(imgs_grayscale_2)
masks[2] = generate_mask(imgs_grayscale_3)
masks[3] = generate_mask(imgs_grayscale_4)
masks[4] = generate_mask(imgs_grayscale_5)
masks[5] = generate_mask(imgs_grayscale_6)
masks[6] = generate_mask(imgs_grayscale_7)
masks[7] = generate_mask(imgs_grayscale_8)

_masks = []
_reverse_masks = []
blended_images = []

for i in range(len(masks)):
    _mask = ((masks[i] != 0).astype(int))
    _masks.append(_mask)
    _rev_mask =  ( (masks[i] == 0).astype(int))
    _reverse_masks.append(_rev_mask)
    blended_images.append(_mask * colorized_images[i] + _rev_mask * grayscale_images[i])

def show_images(im_grayscale, im_colorized, im_mask, im_blended):
    fig, axes = plt.subplots(1, 4, figsize=(20,20))
    axes[0].imshow(im_grayscale, cmap='gray')
    axes[0].set_title('Grayscale image'), axes[0].set_xticks([]), axes[0].set_yticks([])
```

```
    axes[1].imshow(im_colorized,cmap='gray')
    axes[1].set_title('Colorized image'), axes[1].set_xticks([]), axes[1].set_yticks([]);
    axes[2].imshow(im_mask * 255,cmap='gray')
    axes[2].set_title('Mask'), axes[2].set_xticks([]), axes[2].set_yticks([]);
    axes[3].imshow(im_blended ,cmap='gray')
    axes[3].set_title('Blended'), axes[3].set_xticks([]), axes[3].set_yticks([]);

for i in range(len(grayscale_images)):
    show_images(grayscale_images[i], colorized_images[i], _masks[i], blended_images[i])
```

References

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2018). DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *40*(4), 834–848. https://doi.org/10.1109/TPAMI.2017.2699184

Chen, L.-C., Papandreou, G., Schroff, F., & Adam, H. (2017). Rethinking atrous convolution for semantic image segmentation. *arXiv Preprint arXiv:1706.05587*.

Colorizing black & white images with u-net and conditional GAN — a tutorial. (n.d.). https://towardsdatascience.com/colorizing-black-white-images-with-u-net-and-conditional-gan-a-tutorial-81b2df111cd8.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, *abs/2010.11929*. https://arxiv.org/abs/2010.11929

Fast.ai. (n.d.). https://www.fast.ai/.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). *Generative adversarial networks*. https://doi.org/10.48550/ARXIV.1406.2661

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2016). Image-to-image translation with conditional adversarial networks. *CoRR*, *abs/1611.07004*. http://arxiv.org/abs/1611.07004

Lin, T.-Y., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft COCO: Common objects in context. *CoRR*, *abs/1405.0312*. http://arxiv.org/abs/1405.0312

Mao, X., Li, Q., Xie, H., Lau, R. Y. K., & Wang, Z. (2016). Multi-class generative adversarial networks with the L2 loss function. *CoRR*, *abs/1611.04076*. http://arxiv.org/abs/1611.04076

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *CoRR*, *abs/1706.03762*. http://arxiv.org/abs/1706.03762

Xie, Z., Zhang, Z., Cao, Y., Lin, Y., Bao, J., Yao, Z., Dai, Q., & Hu, H. (2021). SimMIM: A simple framework for masked image modeling. *CoRR*, *abs/2111.09886*. https://arxiv.org/abs/2111.09886

Zhang, R., Isola, P., & Efros, A. A. (2016). Colorful image colorization. *CoRR*, *abs/1603.08511*. http://arxiv.org/abs/1603.08511