Grayscale Image Colorizing With User Hints
Project Idea

When I was looking for ideas for the final project, my father and I talked about old photos of my grandfather. I came up with the idea of coloring old black and white photos. But instead of creating a direct black and white photo coloring model, I thought that users could add their own colors to the photo. This way, users could make the color of the car in the background red or the sweater green in a family photo they had taken in the past.

Dataset Selection and Preparation

After this idea, we started looking for a dataset. First, we thought that old photos were usually family profile pictures or landscape pictures. Therefore, we decided to use the FFHQ dataset, which contains human face photos, and the Places365 dataset, which contains place photos.

First, we downloaded these two datasets from kaggle with the help of the python kagglehub library. Then, we created a data set containing 20,000 data in total, 10,000 from each of the two data sets, and we downloaded this data set to google drive because it allowed us to avoid downloading the same data set over and over. We did this by connecting the drive with the drive.mount('/content/drive') command.

The FFHQ dataset was taking up too much space due to its high quality images. Since the Places365 data set was 216x216, we decided to resize the FFHQ dataset to 512x512 and saved it after resizing it. The reason we didn't save it as 216x216 was because we might use 512x512 in model training; we didn't want to lose quality in the images in that case.

Model Architecture

First of all, we should state that we learned about many issues during the model training and made changes to our decisions.

Our model's

Input: Grayscale image + RGB user hints

Output: RGB colorized image

This was the initial idea in our minds and in this way, we planned to combine the hints we received from the user with RGB layers to obtain a 4-layer structure and use it as input and get 3-layer RGB outputs. Later, we learned that using the RGB color model in image coloring was a mistake and decided to switch to another color model. We will talk about this change later.

After obtaining the dataset and deciding on the input we would give to the model and the outputs we would request, we started researching how we could colorize black and white photos using the Fastai library. The most recommended models were U-Net based models. In addition, it was mentioned that we could use GANs; however, the biggest advantage of U-Net based models was that they were faster. In addition, GANs can give more effective results when used in combination with different models; however, we thought that this would be challenging for us since it requires more resources.

We planned to do all our training on Colab with A100 GPU. However, during the process, the connection drops we experienced during training caused problems even when using U-Net. Considering all these, we decided to train a U-Net based model.

In the encoder part of this U-Net based model, we used ResNet architectures to extract meaningful features from images. ResNet architectures are deep networks that have been previously trained with large datasets. While ResNet helps us understand "what" is in the image in general; U-Net uses this information to allow us to make color predictions at the pixel level. We thought these would be a good choice for us since we will be using the difference between the predicted pixel values and the actual values as losses in our project.

Before starting model training, we had to decide on the batch size and image_size values. The batch size was determined by changing it according to the size of the ResNet architecture we used experimentally and the GPU memory. As for image_size, we trained our models in the 128x128 size during the test stages to decide which ResNet model and which algorithm to use. Later, when training our final models, we increased the image_size value to 256x256. Because we experienced memory and resource problems with 512x512 sized images.

DataBlock and Dataloaders Creation

In the Data Preprocessing phase, we first converted the originally colored images we received into grayscale images in order to use them as input. Then, we created a hint rgb image from pixels we randomly took from the original image, approximately 2 percent of the pixel count, and we combined these two image layers. The resulting layer was as follows for a pixel (grayscale, red, green, blue).

```python
def get_model_input(fn):

    original_img = Image.open(fn).convert('RGB')

    original_img_resized = original_img.resize((IMG_SIZE, IMG_SIZE), Image.Resampling.LANCZOS)

    original_img_np = np.array(original_img_resized)

    gray_img_np = cv2.cvtColor(original_img_np, cv2.COLOR_RGB2GRAY)
    hint_mask_np = create_sparse_color_hints(original_img_np)

    combined = np.concatenate([
        np.expand_dims(gray_img_np, axis=2),
        hint_mask_np
    ], axis=2)

    return torch.from_numpy(combined).permute(2, 0, 1).float() / 255.0
```

```python
def create_sparse_color_hints(original_img_np, num_points=327, point_size=1):

    h, w, c = original_img_np.shape
    hint_mask = np.zeros_like(original_img_np)

    if h <= point_size*2 or w <= point_size*2:
        return hint_mask

    for _ in range(num_points):
        y = random.randint(point_size, h - point_size - 1)
        x = random.randint(point_size, w - point_size - 1)

        y_start, y_end = max(0, y - point_size), min(h, y + point_size)
        x_start, x_end = max(0, x - point_size), min(w, x + point_size)

        hint_mask[y_start:y_end, x_start:x_end] = original_img_np[y_start:y_end, x_start:x_end]

    return hint_mask
```

Next, we determined the augmentations we wanted to apply in the DataBlock. The critical augmentation for us in this step was max_lighting, because it caused changes in the colors of the photos. We added the other augmentations to give the images more variety. We set the p_affine value to 0.75, meaning that the probability of the images undergoing afine transformation was 75%.

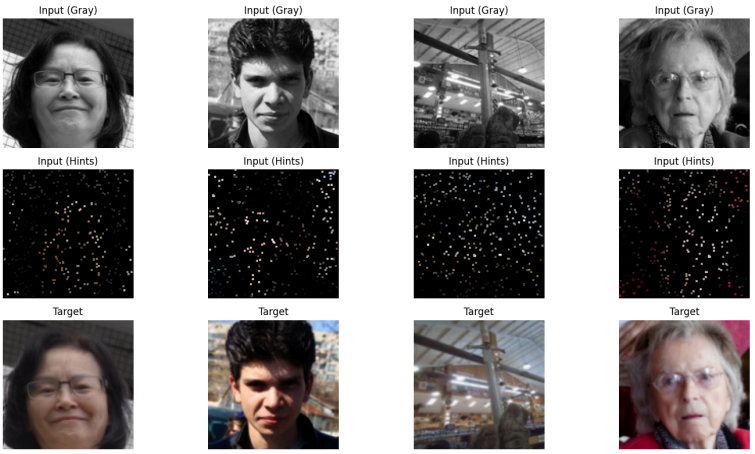
Finally, after normalization, which is one of the important steps of the data preprocessing, the average pixel data is compressed. Normalization prevents the data between RGB [0, 255] values from producing high loss values during model training and the weight changes in the backward pass of the model from being excessive. This process is performed by proportioning all data to the [0, 1] range.

Afterwards, we created the DataBlock that we will give to the network and the DataLoaders that will allow us to give this DataBlock to the network. Our batch size was determined as 64 due to the image size of 128x128 and the ResNet34 architecture we used in the model we created as a benchmark. This is a high batch number for our model training process.

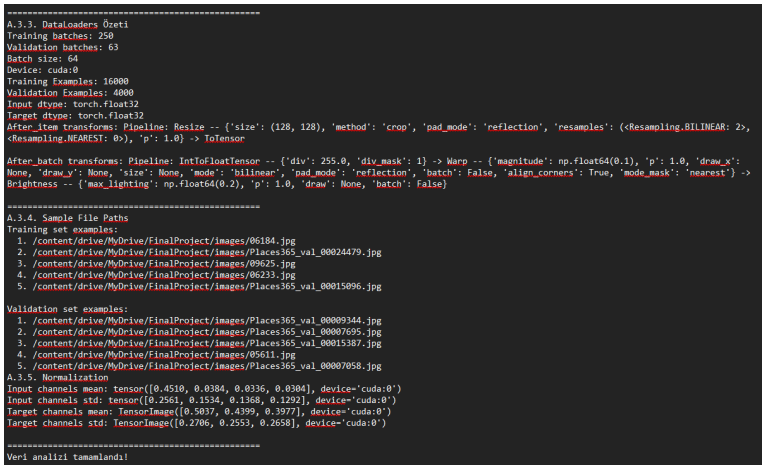
We wanted to see an example batch with show_batch. Here we used the term @typedispatch. The reason is that instead of the already existing show_batch function, it is a decorator that allows us to use our own show_batch method that we created to show our DataBlock with 4 layers of input.

```python
@typedispatch
def show_batch(x:Tensor, y:TensorImage, samples, ctxs=None, max_n=6, **kwargs):
    fig, axs = plt.subplots(3, max_n, figsize=(15, 8))
    for i in range(max_n):

        axs[0, i].imshow(x[i, 0].cpu(), cmap='gray')
        axs[0, i].set_title("Input (Gray)")
        axs[0, i].axis('off')

        axs[1, i].imshow(x[i, 1:].cpu().permute(1, 2, 0))
        axs[1, i].set_title("Input (Hints)")
        axs[1, i].axis('off')

        axs[2, i].imshow(y[i].cpu().permute(1, 2, 0))
        axs[2, i].set_title("Target")
        axs[2, i].axis('off')
    plt.tight_layout()
    plt.show()
    return ctxs

dls.show_batch(max_n=4)


x, y = dls.one_batch()

print(f"Input batch shape: {x.shape}")
print(f"Output batch shape: {y.shape}")
print(f"Input value range: [{x.min():.2f}, {x.max():.2f}]")
print(f"Output value range: [{y.min().item():.2f}, {y.max().item():.2f}]")
```

Here is a sample batch. In the following stages, we will realize that we followed the wrong way..



We got an error when we wanted to use the dls.summary() function. As a result of our research, we learned that this could be due to the 4-layer input. We tried to create a function that would print the information. Therefore, we asked the AI for help add more information that we can see in the dls.summary() method.



I said that we set the batch size as 64. When we divide our dataset containing 20k data into 64, there are 313 batches in total, and these are separated as 20% validation and 80% training. Afterwards, each image undergoes a transformation on its own — in our project, we only resize it so that the network does not receive images of different sizes — and then the augmentation transforms we mentioned earlier are applied batch by batch.

In the next line, the locations where the sample images were taken are shown.

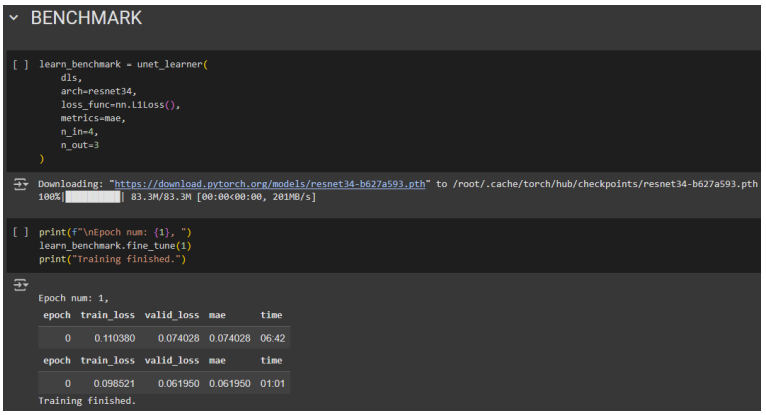
Start Training

Now we start the model training process.

We need to choose a loss_function to calculate the loss value that I mentioned in the previous paragraph. At this point, we preferred L1 Loss Function, namely Mean Absolute Error. Since we will be working with images and will be processing color data such as grayscale, red, green, blue in pixels, we thought that it would be sufficient to take the positive difference (absolute distance) between predicted and true value of pixels. We obtained the average color prediction error by taking the average of all pixel values. Mean Squared Error could also be used here; however, we did not prefer it because we did not want to further reduce the pixel values that we normalized.

First, we trained the model with the fine_tune method for only 1 epoch using the ResNet34 architecture to see if the model would start training correctly and at which level it would start if it did. In this process, we trained the model in a way that 1 epoch freezed network and 1 epoch unfreeze network.

In the freezed network, the model only trains the last layers that were added specifically to our model. In the Unfreeze case, as I mentioned at the beginning, the layers of ResNet that were previously trained with very large data sets are also included in the training. This process is called fine_tune; that is, a model that was previously trained with large data sets is re-optimized with fine-tuning according to our data and purpose.



The MAE (Mean Absolute Error) value we get here is 0.06 — meaning when pixels are normalized to the range [0, 1], the model makes an average error of 0.06. This corresponds to a value of about 15 units in the range [0, 255]. It seems like a good place to start, but we will explain the errors that come with the 4-layer input structure and the use of RGB later.

Resnet50, EarlyStopping and Learning Rate Finder

We did another test training using image size = 128 and batch size = 64. In this training, we used the ResNet50 architecture and the EarlyStopping method. The difference between ResNet50 and ResNet34 is the number of layers, that is, the network size. As the name suggests, ResNet50 has 50 layers, while ResNet34 has 34 layers. Larger model architectures can be preferred for more demanding tasks; however, it should be noted that large networks are more prone to overfitting — especially for small tasks and low data counts — However, we do not think that our process is simple or that 20K data is too small. Therefore, we plan to try the ResNet101 architecture, which is even larger than ResNet50, in the following stages.

Returning to the topic of ResNet50 and EarlyStopping: The metric followed for EarlyStopping is the val_loss. In other words, it is the minimum improvement rate that we will accept to be convinced that the model continues to learn without overfitting. We set this as min_delta=0.0001. We set this value according to the val_loss value we obtained in the benchmark. The patience value indicates how many epochs we will wait if the model does not show any progress in learning. We chose this value as 10.

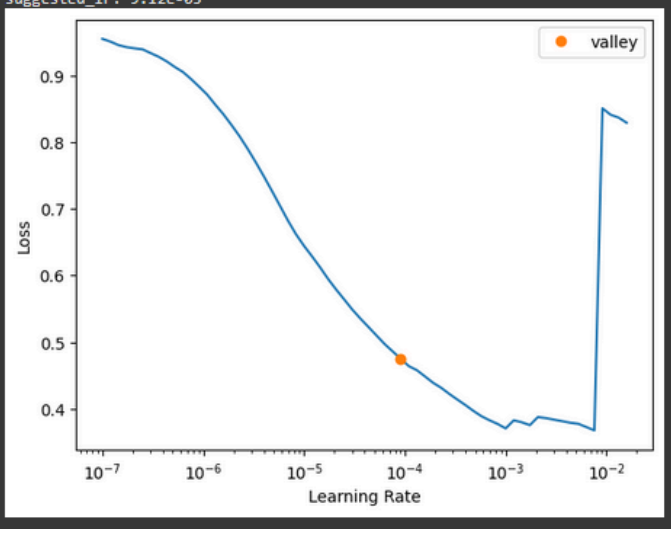
In this experiment, we used the fine_tune method because we wanted to determine the epoch numbers in the freeze and unfreeze stages ourselves.

In the next stage, we tried to determine the learning rate value with the lr_finder method. Then, we tried to ensure that the model was trained in the ideal number of steps using this learning rate. Because if the learning rate is high, the loss value may decrease quickly; however, this may cause us to skip the minimum loss value that can be reached. On the other hand, if the learning rate is too low, it may take more epochs for the model to reach this minimum value, which may increase the risk of overfitting.

Another helper function we use is SaveModelCallback. In this method, we enter the value we want to monitor, its type, and the name we want to save the model. The function follows the training according to the value type we give and saves the model of the epoch at the point where this value is lowest in the early stopping state.

```
[ ] # C.1. Learning Rate Finder
    lr_finder_result_head = learn_es_resnet50.lr_find()
    LEARNING_RATE = lr_finder_result_head.valley
    print(f"suggested_lr: {lr_finder_result_head.valley:.2e}")
```

```
suggested_lr: 9.12e-05
```



In this step, we trained the model for 20 epochs while the layers were in the freeze state. We did not unfreeze in this experiment; we were just curious about what we would get if we trained the model in the freeze state. Also, in the next step, we will already train the unfreezed network using the discriminative learning rate.

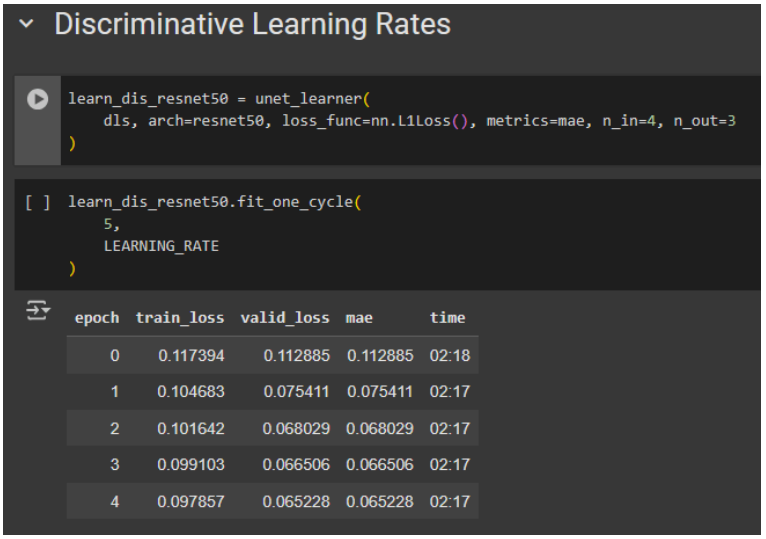
```
learn_es_resnet50.fine_tune(
    20,
    base_lr=LEARNING_RATE,
    cbs=[
        early_stop_cb,
        SaveModelCallback(monitor='valid_loss', fname='learn_es_resnet50'
    ]
)
```

epoch	train_loss	valid_loss	mae	time
0	0.104338	0.075097	0.075097	02:22
1	0.102554	0.069841	0.069841	02:23
2	0.103203	0.070591	0.070591	02:23
3	0.101785	0.073087	0.073087	02:19
4	0.101208	0.071069	0.071069	02:19
5	0.100198	0.066472	0.066472	02:20
6	0.098789	0.065327	0.065327	02:23
7	0.098326	0.066129	0.066129	02:23
8	0.097278	0.067166	0.067166	02:19
9	0.096815	0.065466	0.065466	02:19
10	0.096081	0.063799	0.063799	02:19
11	0.096612	0.062691	0.062691	02:23
12	0.095955	0.062803	0.062803	02:21
13	0.095154	0.062233	0.062233	02:19
14	0.095280	0.061829	0.061829	02:22
15	0.094971	0.061995	0.061995	02:23
16	0.094950	0.062261	0.062261	02:19
17	0.094302	0.061831	0.061831	02:19
18	0.094425	0.061903	0.061903	02:19
19	0.095025	0.062018	0.062018	02:19

```
Better model found at epoch 5 with valid_loss value: 0.06647178530693054.
Better model found at epoch 6 with valid_loss value: 0.06532692164182663.
Better model found at epoch 10 with valid_loss value: 0.06379938870668411.
Better model found at epoch 11 with valid_loss value: 0.0626906231045723.
Better model found at epoch 13 with valid_loss value: 0.06223326176404953.
Better model found at epoch 14 with valid_loss value: 0.061829227954149246.
```

Resnet50, Discriminative Learning Rates, Transfer Learning, Early stopping

Here, since we will use the new but same network structure and DataBlock as the previous network structure, we did not find a new learning_rate for the network in the frozen state. First, we trained the network in the frozen state for 5 epochs.



Here you can see that even though it starts with a higher loss compared to the previous training, but we get a better loss value with a more stable decrease. In other words, we can get different training results even with a new model created in the same structure and the same datablock.

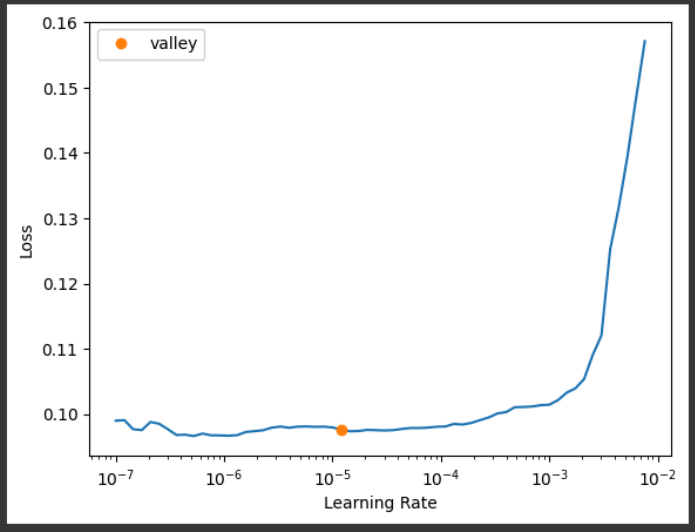
Afterwards, we unfreezed the model, that is, we opened the previously trained layers and started to fine-tune them according to our own dataset. Before doing this, we determined the safe learning rate limits that we will use for the discriminative learning rate with the lr_finder method.

Discriminative learning rate allows us to determine different learning rates for different layers. The reason for use, as I have mentioned many times before, is that pre-trained layers should change less, while new layers added specifically to our model are should updated more with a higher learning rate.

We do this by giving a start and end value in the form of slice(lr_start, lr_end) to the lr parameter we give to the fit_one_cycle method.

```
[ ]  learn_dis_resnet50.unfreeze()
```

```
lr_finder_results = learn_dis_resnet50.lr_find()
```




```
[2]  lr_start = 1e-6  # 10^(-6)
     lr_end = 1e-4    # 10^(-4)

     print(f"Suggested LR range: {lr_start:.2e} to {lr_end:.2e}")
```

```
Suggested LR range: 1.00e-06 to 1.00e-04
```

In the graph, you can see that the most reliable interval starts at a point between 10^(-7) and 10^(-6) and ends at 10^(-4). Using this reliable interval, we will train the unfreezed network with the discriminative learning rate.

```
[ ] print("Training the full model with discriminative learning rates...")
    learn_dis_resnet50.fit_one_cycle(
        15,
        lr_max=slice(lr_start, lr_end),
        cbs=[
            EarlyStoppingCallback(monitor='valid_loss', min_delta=0.0001, patience=10),
            SaveModelCallback(monitor='valid_loss', fname='learn_dis_resnet50')
        ]
    )
```

Training the full model with discriminative learning rates...

epoch	train_loss	valid_loss	mae	time
0	0.098056	0.064548	0.064548	02:19
1	0.098956	0.068202	0.068202	02:23
2	0.098824	0.063709	0.063709	02:20
3	0.097078	0.063762	0.063762	02:23
4	0.095895	0.062087	0.062087	02:20
5	0.096198	0.063528	0.063528	02:23
6	0.094489	0.061869	0.061869	02:20
7	0.094425	0.062018	0.062018	02:23
8	0.094091	0.060447	0.060447	02:19
9	0.093784	0.063083	0.063083	02:23
10	0.093676	0.061923	0.061923	02:19
11	0.093981	0.060544	0.060544	02:20
12	0.093067	0.060353	0.060353	02:19
13	0.092584	0.060046	0.060046	02:23
14	0.092941	0.060241	0.060241	02:23

```
Better model found at epoch 0 with valid_loss value: 0.06454789638519287.
Better model found at epoch 2 with valid_loss value: 0.06370875984430313.
Better model found at epoch 4 with valid_loss value: 0.06208747252821922.
Better model found at epoch 6 with valid_loss value: 0.061868928372859955.
Better model found at epoch 8 with valid_loss value: 0.06044693663716316.
Better model found at epoch 12 with valid_loss value: 0.06035275384783745.
Better model found at epoch 13 with valid_loss value: 0.0600464753806591.
```

Although there was an increase in the loss value in some steps during the model training period, epochs with better loss values were obtained within a certain pattern. In other words, training can be continued. Since it is a model trained for testing purposes, we decided that this training system is usable and by setting the image_size to 256, we start training the model we really want to use.

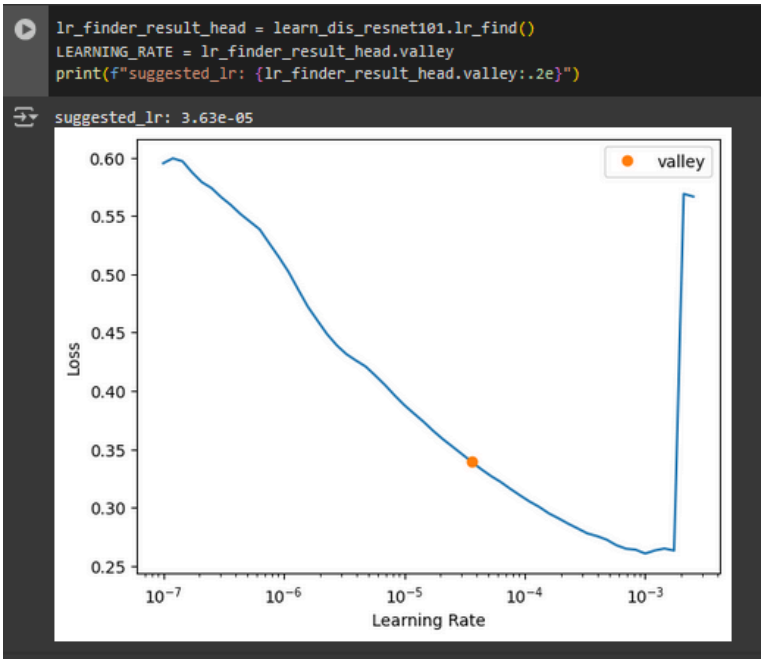
Same Approach with resnet101 Image Size increase to 256x256

At this stage, since the image_size has changed, we re-run the helper functions such as create_sparse_color_hints, get_model_input, get_target_image to work properly for 256x256 images. And as it should be, we create the DataBlock and DataLoader with all parameters except image_size being the same.

We decided to use the discriminative learning rate, freeze/unfreeze training and early stopping training as the main training type, which we think is successful. The only difference was that we preferred a larger architecture such as ResNet101.

Due to this large architecture and the 40 GB capacity of the GPU we used in Colab, we had to reduce the batch size to 32. This means that the weights will be updated more frequently; but it does not mean that the model will learn faster because the amount of data used for each update decreases.

After creating the model, we found the learning rate to train it in the freeze state.



And we trained the network in the frozen state for 10 epochs. As the model grows, we need to increase the number of epochs because as the number of weights increases, more tuning is required.

```
[ ] learn_dis_resnet101.fit_one_cycle(
        10,
        LEARNING_RATE
    )
```

20.00% [2/10 27:40<1:50:40]

epoch	train_loss	valid_loss	mae	time
0	0.123831	0.089093	0.089093	13:48
1	0.111443	0.076524	0.076524	13:51

78.40% [98/125 00:33<00:09 0.1098]

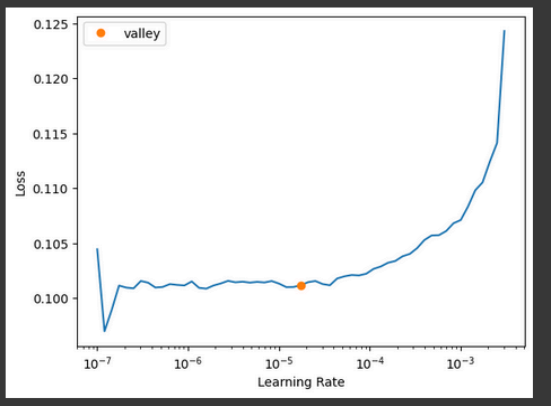
epoch	train_loss	valid_loss	mae	time
0	0.123831	0.089093	0.089093	13:48
1	0.111443	0.076524	0.076524	13:51
2	0.109763	0.085667	0.085667	13:56
3	0.107997	0.071111	0.071111	13:56
4	0.106045	0.071622	0.071622	13:57
5	0.104336	0.072696	0.072696	13:56
6	0.103192	0.069318	0.069318	13:56
7	0.103035	0.069697	0.069697	13:56
8	0.102251	0.068076	0.068076	13:56
9	0.102584	0.068364	0.068364	13:56

Despite having twice as many epochs, we obtained a higher loss value than the model trained with ResNet50 for 5 epochs. There may be two reasons for this: the first is the increased image_size, and the second is the network which has larger layer. As I mentioned, large networks require more training. Also, since the image_size increased, the number of estimated pixels increased, and therefore the error may have increased.

We unfreezed the network and created the graph with lr_finder.

```
[ ] learn_dis_resnet101.unfreeze()
```

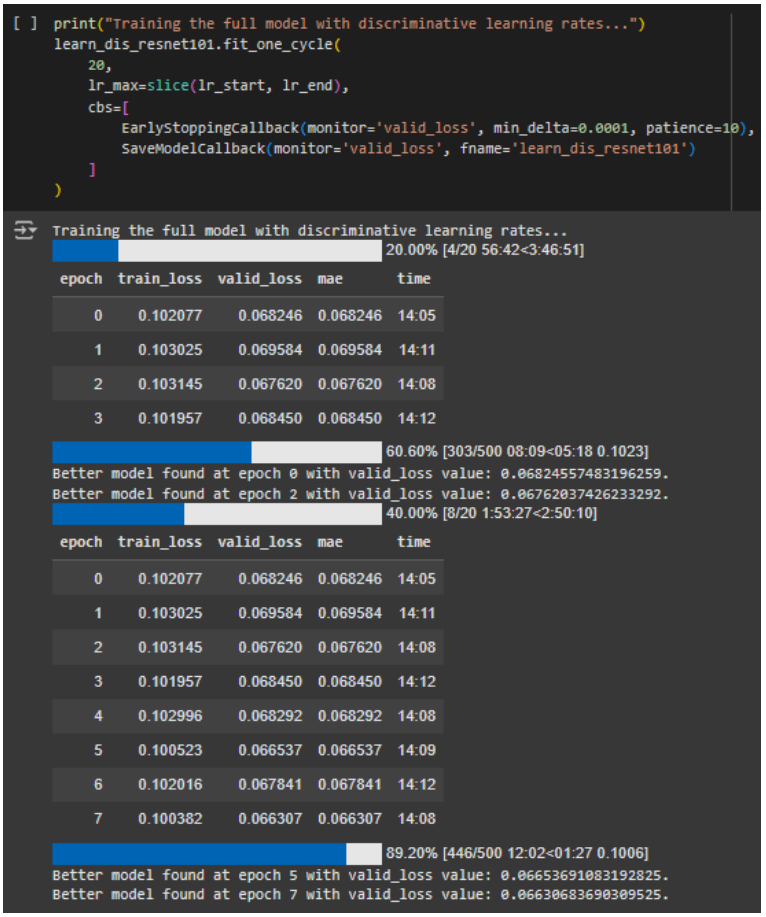
```
[ ] lr_finder_results = learn_dis_resnet101.lr_find()
```




```
[ ] lr_start = 1e-6  # 10^(-6)
    lr_end = 5e-5   # 0.00005

    print(f"Suggested LR range: {lr_start:.2e} to {lr_end:.2e}")
```

```
Suggested LR range: 1.00e-06 to 5.00e-05
```

The graph seems to be stable just before 10^(-6); however, it starts to rise earlier than the previous unfreezed lr_finder graph. Therefore, 10^(-4) is not a safe zone. It makes sense to end our learning rate range earlier, closer to 10^(-5). We take the midpoint between 10^(-5) and 10^(-4) as the end of our learning rate range and try to train our unfreeze network using early stopping for 20 epochs; however, Colab stopped training at this point. And training did not finish.



```
[ ] print("Training the full model with discriminative learning rates...")
    learn_dis_resnet101.fit_one_cycle(
        20,
        lr_max=slice(lr_start, lr_end),
        cbs=[
            EarlyStoppingCallback(monitor='valid_loss', min_delta=0.0001, patience=10),
            SaveModelCallback(monitor='valid_loss', fname='learn_dis_resnet101')
        ]
    )
```

```
Training the full model with discriminative learning rates...
20.00% [4/20 56:42<3:46:51]
```

epoch	train_loss	valid_loss	mae	time
0	0.102077	0.068246	0.068246	14:05
1	0.103025	0.069584	0.069584	14:11
2	0.103145	0.067620	0.067620	14:08
3	0.101957	0.068450	0.068450	14:12

```
60.60% [303/500 08:09<05:18 0.1023]
Better model found at epoch 0 with valid_loss value: 0.06824557483196259.
Better model found at epoch 2 with valid_loss value: 0.06762037426233292.
40.00% [8/20 1:53:27<2:50:10]
```

epoch	train_loss	valid_loss	mae	time
0	0.102077	0.068246	0.068246	14:05
1	0.103025	0.069584	0.069584	14:11
2	0.103145	0.067620	0.067620	14:08
3	0.101957	0.068450	0.068450	14:12
4	0.102996	0.068292	0.068292	14:08
5	0.100523	0.066537	0.066537	14:09
6	0.102016	0.067841	0.067841	14:12
7	0.100382	0.066307	0.066307	14:08

```
89.20% [446/500 12:02<01:27 0.1006]
Better model found at epoch 5 with valid_loss value: 0.06653691083192825.
Better model found at epoch 7 with valid_loss value: 0.06630683690309525.
```

Using LAB as Color Model

The next day we decided to start the training again, but we started to question how accurate the RGB color model was for our project. Because when we used the RGB color model, we were almost trying to recreate the photo. There is no separate layer in the RGB color model that only adjusts black and white; [0, 0, 0] represents black, [255, 255, 255] represents white. But in a color model like LAB, which I saw in previous Computer Graphics courses, the L (Lightness) value determines the black-white ratio of the color, A determines the green-red balance, and B determines the blue-yellow balance. This is actually a color model that we can use exactly. It is enough to train the model only the A and B values; because in grayscale images, we already have the L layer. So instead of recreating the image as in RGB, we are just trying to predict the green-red and blue-yellow ratios.

After this idea, the input and output shapes of the model also changed:

Input L (grayscale image) + A and B (Hint Mask) as 3 layers where we obtained the LAB color palette,

Output became just A and B channels.

Because there is no need to guess L for us, just A and B are enough.

When the image size is 256:

Input shape: [3, 256, 256]

Output shape: [2, 256, 256]

After this change, we also had to update the helper functions.

Functions were added to normalize and denormalize the color values of the images according to the new color palette.

In the LAB color model, although the A and B layers have a range of 256 as in RGB, they take values in the range of [-127, 128] instead of [0, 255]. The L (Lightness) layer takes values in the range of [0, 100]. Our normalization and denormalization functions were updated according to these ranges.

In the model training, in order to simulate the input we will receive from the user, we updated the hints we randomly received from the original image to only take the A and B layers. Because we will already have the L layer. We will obtain the LAB color model by concat the A and B layers containing color clues to this L layer.

However, we will not use this hint mask for every input. Because in this case, our model may predict colors poorly in cases where the user does not provide any hints. Or if the user provides a hint for only a small area, the rest may remain uncolored. To prevent this, we set the probability of adding a hint to 50% (probability 0.5) in the get_input_image function.

In the DataBlock creation phase for the models we will train using LAB, we applied the same augmentation transforms to the batches as in the version where we used the RGB color model. The

only difference in this part was that we reduced the validation set ratio from 0.2 to 0.15 to slightly increase the number of data in the training set.

Here is a show_batch output for the LAB color model:

In the model training process with the LAB color model, we decided to use the Discriminative Learning Rate and EarlyStopping functions, which we thought were a more accurate approach when using the RGB color model, and the algorithm we determined as our main model training algorithm. We started our model trainings for the LAB color model with this understanding.

Due to the longer duration of model training with Resnet101 and the interruptions we experienced in colab, two of our Resnet101 trainings were interrupted, which caused almost 8 hours of model training to be wasted. Therefore, we decided to train using Resnet34 and Resnet 50 architectures as model training architectures.

Resnet34, Discriminative Learning Rates, Transfer Learning, Early stopping

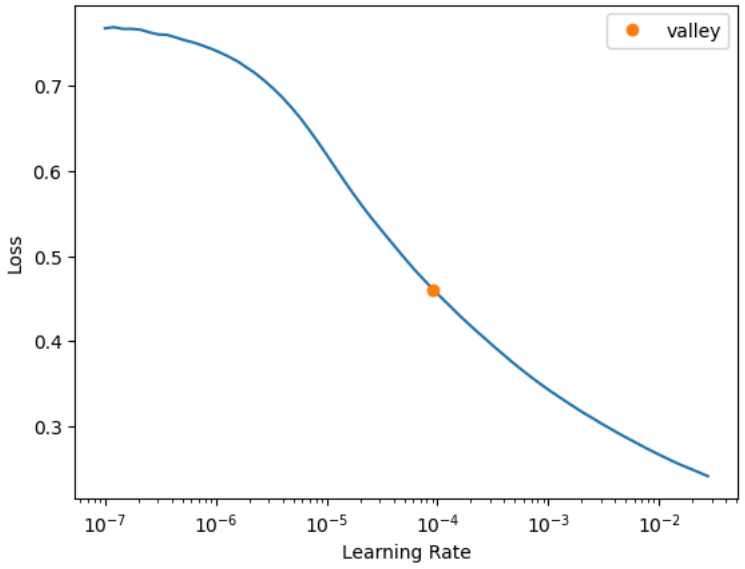
We performed the training of both models with the same stages and different parameters.

First, we trained the model using the resnet34 architecture;

We found the learning rate for the network in the frozen state and trained it for 10 epochs according to this learning rate. However, since there is a 2-stage training here (freeze and unfreeze network training), we thought that using EarlyStopping in the first stage would be a bit absurd and removed it.

```
learn = unet_learner(
    dls,
    arch=resnet34,
    loss_func=nn.L1Loss(),
    metrics=mae,
    n_in=3,
    n_out=2,
    self_attention=True,
)
```

```
[ ]  lr_finder_result_head = learn.lr_find()
     LEARNING_RATE = lr_finder_result_head.valley
     print(f"suggested_lr: {lr_finder_result_head.valley:.2e}")
```

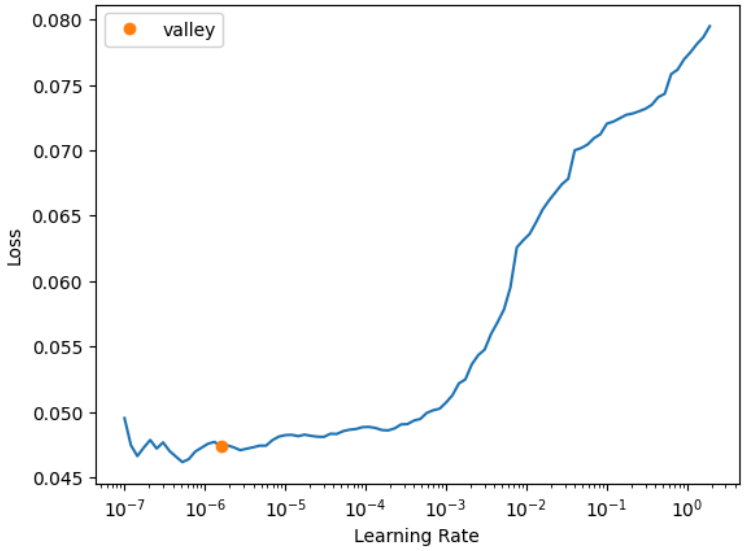


We obtained a different loss value because we used a different color model. I can say that we experienced a stable and high loss decrease during the model training period.

```
learn.fit_one_cycle(
    10,
    LEARNING_RATE
)
```

epoch	train_loss	valid_loss	mae	time
0	0.092471	0.080172	0.080172	01:57
1	0.064631	0.061752	0.061752	01:57
2	0.057780	0.058339	0.058339	01:57
3	0.054335	0.056049	0.056049	01:57
4	0.053137	0.052964	0.052964	01:57
5	0.051753	0.052108	0.052108	01:57
6	0.050402	0.051051	0.051051	01:57
7	0.049615	0.051035	0.051035	01:57
8	0.049241	0.050672	0.050672	01:57
9	0.049106	0.050886	0.050886	01:57

Then we unfreezed the network and created the learning rate finder graph for the unfreezed network.



This time, we thought that 10^(-6) was not within the confidence interval in the graph and that it would be more logical to choose a value between 10^(-5) and 10^(-6) and determined a learning rate range for Discriminative Learning Rate by taking the starting point as 5.25e-05 and the ending point as 10^(-4). Then, we trained our model for 20 epochs using this range and the EarylStopping logic. At this stage, we increased the min_delta value of EarylStopping compared to the one in the RGB color model trainings because the loss value are completely different. We determined the new min_delta value as min_delta = 0.0005 by taking into account the loss values we obtained in the previous step.

epoch	train_loss	valid_loss	mae	time
0	0.049303	0.050406	0.050406	02:01
1	0.048806	0.049887	0.049887	02:01
2	0.048292	0.048894	0.048894	02:01
3	0.046904	0.050040	0.050040	02:02
4	0.045705	0.047234	0.047234	02:01
5	0.043612	0.047581	0.047581	02:01
6	0.042615	0.046177	0.046177	02:01
7	0.040853	0.045637	0.045637	02:02
8	0.039759	0.045457	0.045457	02:02
9	0.038430	0.044752	0.044752	02:02
10	0.037115	0.044637	0.044637	02:02
11	0.036221	0.044168	0.044168	02:02
12	0.035819	0.044456	0.044456	02:02
13	0.035066	0.043035	0.043035	02:01
14	0.034172	0.043677	0.043677	02:02
15	0.033949	0.043345	0.043345	02:01
16	0.033003	0.043024	0.043024	02:01
17	0.032984	0.043300	0.043300	02:02
18	0.032118	0.043216	0.043216	02:02
19	0.032574	0.043045	0.043045	02:01

In this training process, while there were high decreases in the first 10 epochs, the decrease continued in the following epochs, even less. Due to this decrease in the acceleration of the model's loss decrease, we did not consider training more. It was a model that made us happy with the outputs, but we still wanted to finish the project by training with the resnet50 architecture.

Resnet50, Discriminative Learning Rates, Transfer Learning, Early stopping

The training stages of the resnet50 architecture were the same as the training stages of this model, only the epoch number was different from the parameters. Since we used a larger architecture, the epoch time increased significantly, so we updated the epoch numbers from 10 and 20 in the previous step to 5 and 10 in this algorithm.

```
[ ] learn_renset50 = unet_learner(
        dls,
        arch=resnet50,
        loss_func=nn.L1Loss(),
        metrics=mae,
        n_in=3,
        n_out=2,
        self_attention=True,
    )
```

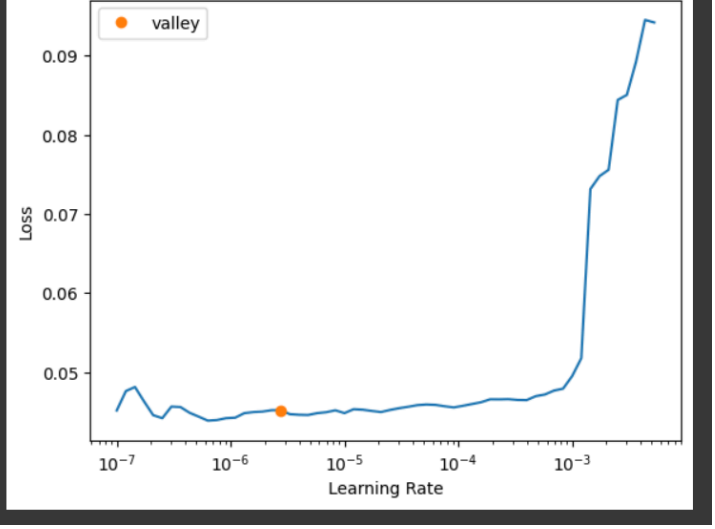
```
[ ]
    lr_finder_result_head = learn_renset50.lr_find()
    LEARNING_RATE = lr_finder_result_head.valley
    print(f"suggested_lr: {lr_finder_result_head.valley:.2e}")
```

```
[ ] learn_renset50.fit_one_cycle(
        5,
        LEARNING_RATE
    )
```

epoch	train_loss	valid_loss	mae	time
0	0.060288	0.064047	0.064047	14:18
1	0.050630	0.049668	0.049668	14:17
2	0.047846	0.047508	0.047508	14:17
3	0.046115	0.046145	0.046145	14:17
4	0.045850	0.045267	0.045267	14:17

```
[ ] learn_renset50.unfreeze()
```

```
[ ] lr_finder_results = learn_renset50.lr_find()
```



At this stage, it was easier to make a choice to create a Discriminative Learning Rate range. The graph is in a safe range from 10^(-6) to almost 10^(-3). However, for a more secure approach, we determined a learning rate range as [10^(-6), 10^(-4)] and continued training.

```
lr_start = 1e-6  # 10^(-6)
lr_end = 1e-4   # 10^(-4)

print(f"Suggested LR range: {lr_start:.2e} to {lr_end:.2e}")
```

```
Suggested LR range: 1.00e-06 to 1.00e-04
```

```
print("Training the full model with discriminative learning rates...")
learn_renset50.fit_one_cycle(
    10,
    lr_max=slice(lr_start, lr_end),
    cbs=[
        EarlyStoppingCallback(monitor='valid_loss', min_delta=0.0005, patience=10),
        SaveModelCallback(monitor='valid_loss', fname='learn')
    ]
)
```

```
Training the full model with discriminative learning rates...
```

epoch	train_loss	valid_loss	mae	time
0	0.047169	0.046882	0.046882	14:23
1	0.047463	0.046987	0.046987	14:26
2	0.045423	0.044920	0.044920	14:22
3	0.044079	0.044505	0.044505	14:25
4	0.043195	0.045213	0.045213	14:25
5	0.042143	0.043567	0.043567	14:22
6	0.040721	0.042484	0.042484	14:25
7	0.039742	0.042252	0.042252	14:23
8	0.039543	0.042248	0.042248	14:25
9	0.039684	0.041674	0.041674	14:25

```
Better model found at epoch 0 with valid_loss value: 0.04688176140189171.
Better model found at epoch 2 with valid_loss value: 0.0449199378490448.
Better model found at epoch 3 with valid_loss value: 0.044504713267087936.
Better model found at epoch 5 with valid_loss value: 0.0435674674808979.
Better model found at epoch 6 with valid_loss value: 0.042484499514102936.
Better model found at epoch 7 with valid_loss value: 0.04225235432386398.
Better model found at epoch 8 with valid_loss value: 0.04224840924143791.
Better model found at epoch 9 with valid_loss value: 0.04167449474334717.
```

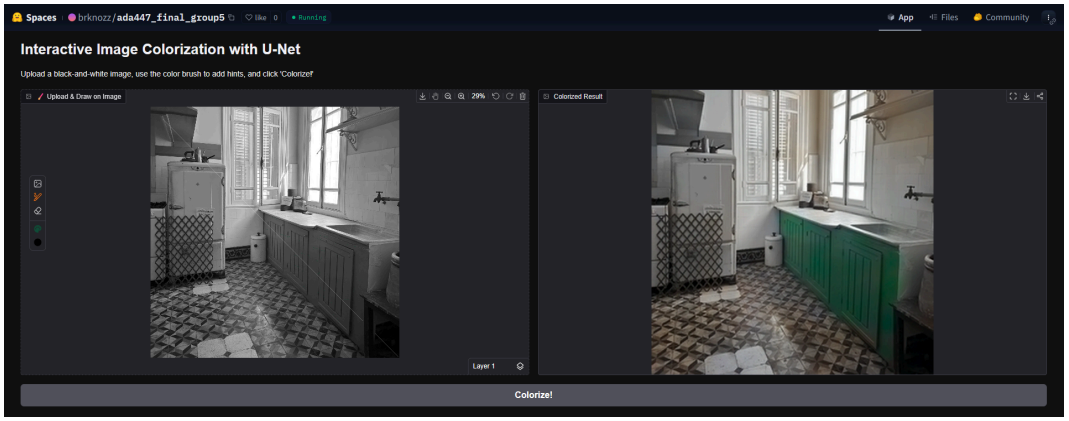
This time, we obtained a better loss by training the larger architecture in fewer epochs. The possible reasons for this may be that the learning rate range in the unfreeze case was smoother and we were able to choose a wide and stable range. Another reason is that our project (input-output shapes) is more suitable for this network.

Throughout this process, we experimented using the gradio interface to test the models, and the model we were most satisfied with was the LAB Color Model, which we trained for a total of 15 epochs using Discriminative Learning Rate and Transfer Learning strategies.

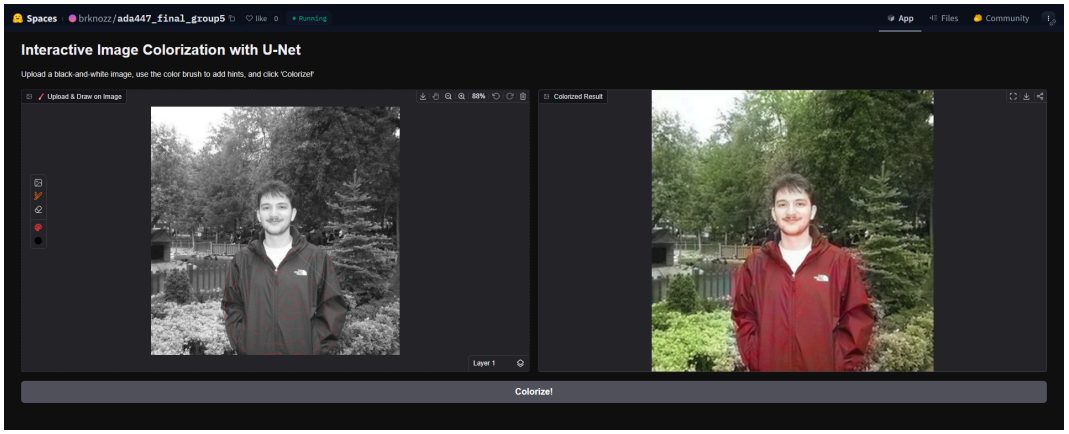
Model Test with in Real Life Images

In our Gradio interface, we don’t need to take random hint points because the photo is not colored. It takes the A and B channels of the hint given by the user and combines them with the grayscale image and gives them as input to our model.

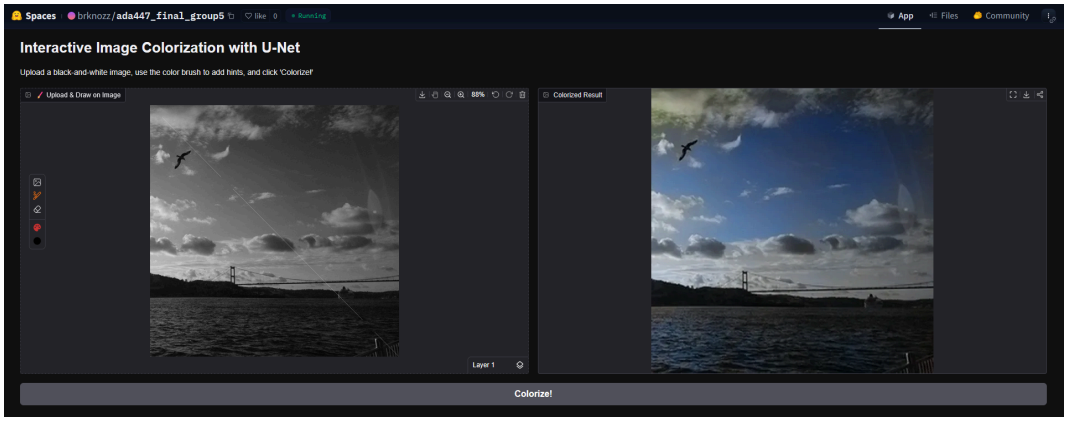
Here are some sample outputs.



(We drew green dots on the cabinet as a hint. It's originally similar to this green.)



(I just placed red hint dots on the raincoat I was wearing. It's originally black.)



(Here I didn't give any hint.)

You can visit our HuggingFace Space, Medium Blog and Colab File:

https://huggingface.co/spaces/brknozz/ada447_final_group5

https://medium.com/@brknozz/grayscale-image-colorizing-with-user-hints-176abc858350

ADA447_FinalProject_Group_5.ipynb