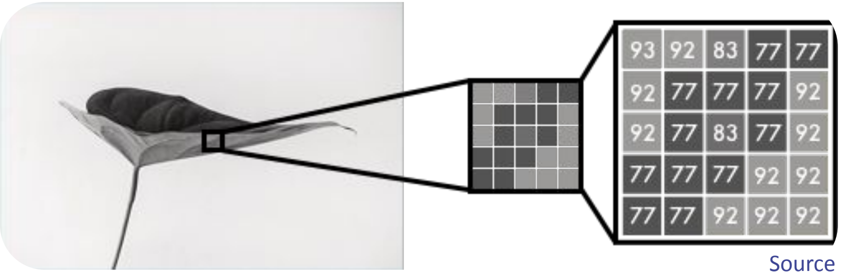
Colorizing Old Black & White Photos with CNNs

By Eli Bolker & Sabina Prochowski

DS-UA 301 - Final Project



93	92	83	77	77
92	77	77	77	92
92	77	83	77	92
77	77	77	92	92
77	77	92	92	92

Source

Agenda

Source

- Executive Summary
- Problem Motivation
- Background Work
- Technical Challenges
- Approach
- Solution Diagram / Architecture
- Implementation Details
- Demo / Experiment Design Flow
- Experimental Evaluation
- Conclusion
- GitHub Repository

Executive summary

Goal:

- To test which model architecture produces the best black and white image colorization results
  - Via lowest MSE and visual comparison

Solution:

- We will use the following architecture frameworks:
  - CNN (achieves lowest MSE)
  - Resnet18 (undoubtedly more visually accurate)

Source

Value / Benefit:

- For decades, photos were processed and colorized via photoshop or by hand, but now we have NNs that can do this in a fraction of the time it used to take!

Problem Motivation

- A NN may not perform as well on images with less predictable compositions / unconventional colors so we wanted to see how well we can realistically get our results to using a few automatic image colorizer techniques
- We wanted to see which model architecture can produce the best results
    - Also, we were interested in using the models on our own old family B&W photos and colorize them

Source

Source

Background Work

Dataset:

- **MIT Places:** Scene Recognition database of places, landscapes, and buildings (Link)
  - By MIT Computer Science and Artificial Intelligence Laboratory
  - We use a subset of 41,000 images for our ResNet18 implementation & 6000 images for our CNN implementation due to RAM / disk storage restrictions

Source

Background Work

CNN implementation:
https://emilwallner.medium.com/colorize-b-w-photos-with-a-100-line-neural-network-53d9b4449f8d

- We only use the CNN architecture directly from the source for our initial run (prior to our optimizations in the 2nd run)
- For optimization in the 2nd run, we standardize the train data, conduct different optimization measures (change the LR), implement different batch processing, use a different dataset, and also handle different processing of data

Resnet Implementation: https://lukemelas.github.io/image-colorization.html

- The implementation uses ResNet18, which we attempted to hand construct
- We keep most of the rest of the implementation the same - the optimizer and loss function remains consistent with our results comparison with the CNN implementation

Technical Challenges

Source

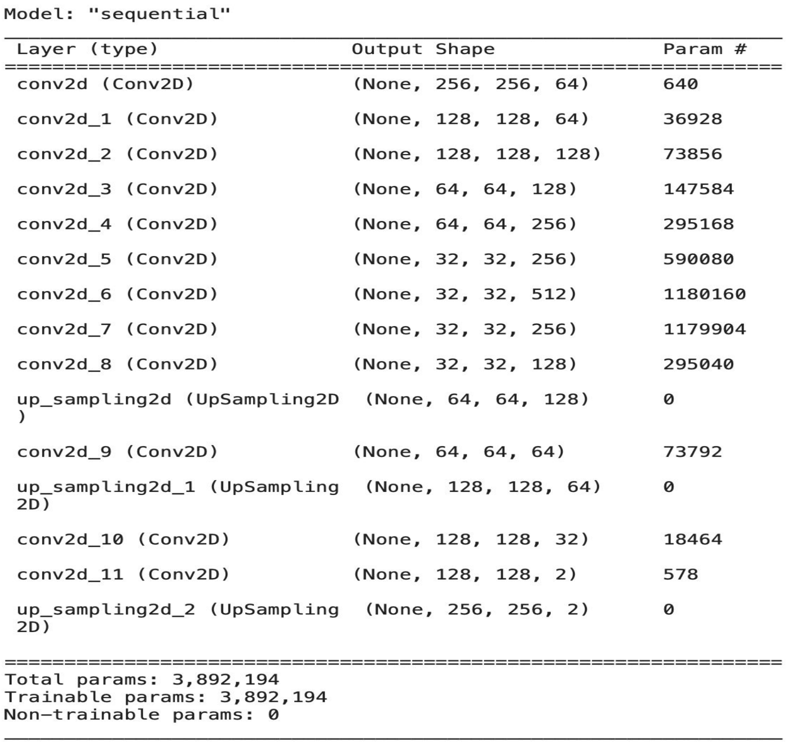
- No optimal way to measure performance, it is quite subjective
    - Since we are using regression, we use the MSE loss function: minimizing the squared distance between the color value we try to predict, and the ground truth color value.
        - However, this loss function is still problematic due to multi-modality issue (extensive research here)
            - For ex., if the true color is actually blue and our NN chooses the wrong color, it will be harshly penalized, causing the model to choose desaturated colors that are less likely to be "very wrong" than bright, vibrant colors (like we see in our first CNN implementation results)
- Though image data is plentiful, we ran into issues with finding a scalable large dataset of face / people data. We tried to use the Coco dataset (330K+ images), but its file size was too large and segmenting to only people specifically was filling up our disk usage. However, after finding that the face data on the original CNN had poor results, we decided to not use face data altogether.

Technical Challenges

Source

- Resnet18 had to run overnight (taking approximately ~9 hours)
- We tried using HPC & GCP to run our models, but it did not speed up the run-time by much
  - We first each purchased Google Colab Pro, but our sessions would crash when converting images to vectorized format in our Tensorflow CNN implementation, filling up our RAM (25GB)
    - We would incrementally decrease 41,000 of MIT Places down until 6000, but even this filled up the RAM
    - Hence, we used Google Colab Pro+ which had a total of 51GB
- We were restricted by computer power - We were considering also using inception for more architecture comparison variety, but it was estimated to take too long (20+ hours), filling up our RAM

Approach



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 256, 256, 64)      640

 conv2d_1 (Conv2D)           (None, 128, 128, 64)      36928

 conv2d_2 (Conv2D)           (None, 128, 128, 128)     73856

 conv2d_3 (Conv2D)           (None, 64, 64, 128)       147584

 conv2d_4 (Conv2D)           (None, 64, 64, 256)       295168

 conv2d_5 (Conv2D)           (None, 32, 32, 256)       590080

 conv2d_6 (Conv2D)           (None, 32, 32, 512)       1180160

 conv2d_7 (Conv2D)           (None, 32, 32, 256)       1179904

 conv2d_8 (Conv2D)           (None, 32, 32, 128)       295040

 up_sampling2d (UpSampling2D  (None, 64, 64, 128)      0
 )

 conv2d_9 (Conv2D)           (None, 64, 64, 64)        73792

 up_sampling2d_1 (UpSampling  (None, 128, 128, 64)     0
 2D)

 conv2d_10 (Conv2D)          (None, 128, 128, 32)      18464

 conv2d_11 (Conv2D)          (None, 128, 128, 2)       578

 up_sampling2d_2 (UpSampling  (None, 256, 256, 2)      0
 2D)

=================================================================
Total params: 3,892,194
Trainable params: 3,892,194
Non-trainable params: 0
_________________________________________________________________
```

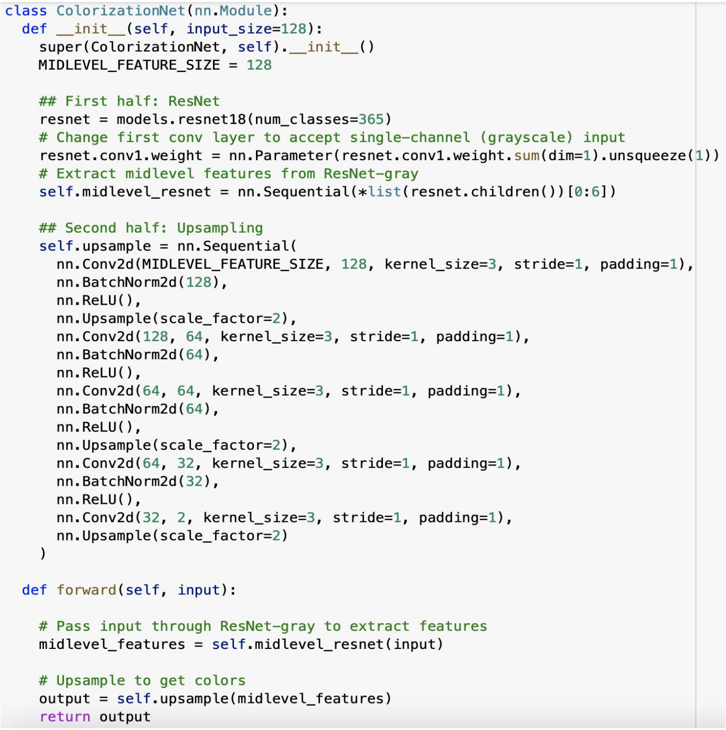
Source

Here, we designed a basic Conv Neural Net with all of the basic layers present (i.e conv, pooling etc). However what is interesting to note, is that because we break apart the input to extract patterns at all scales, we need to then bring back the output to the original size.

We train this model on altered training data until we get convergence, and then check its final accuracy on validation data that we had loaded in separately.

Finally we export out the predictions of our validation set and observe them for final analysis.

Approach

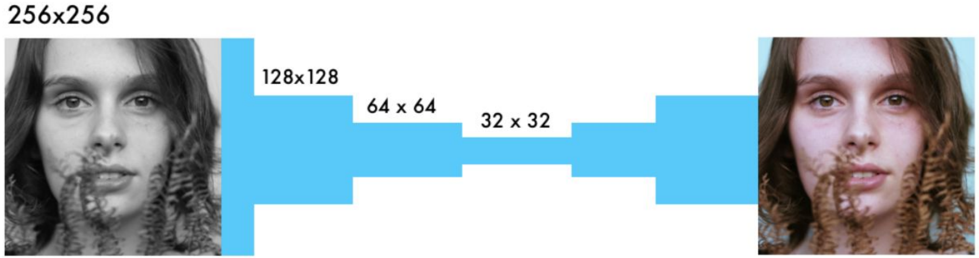


```
class ColorizationNet(nn.Module):
  def __init__(self, input_size=128):
    super(ColorizationNet, self).__init__()
    MIDLEVEL_FEATURE_SIZE = 128

    ## First half: ResNet
    resnet = models.resnet18(num_classes=365)
    # Change first conv layer to accept single-channel (grayscale) input
    resnet.conv1.weight = nn.Parameter(resnet.conv1.weight.sum(dim=1).unsqueeze(1))
    # Extract midlevel features from ResNet-gray
    self.midlevel_resnet = nn.Sequential(*list(resnet.children())[0:6])

    ## Second half: Upsampling
    self.upsample = nn.Sequential(
      nn.Conv2d(MIDLEVEL_FEATURE_SIZE, 128, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(128),
      nn.ReLU(),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(128, 64, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(64, 32, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(32),
      nn.ReLU(),
      nn.Conv2d(32, 2, kernel_size=3, stride=1, padding=1),
      nn.Upsample(scale_factor=2)
    )

  def forward(self, input):

    # Pass input through ResNet-gray to extract features
    midlevel_features = self.midlevel_resnet(input)

    # Upsample to get colors
    output = self.upsample(midlevel_features)
    return output
```

This architecture also uses a CNN, which extracts features from our image, and then applies deconvolutional layers to upscale our features, thus increasing the spatial resolution of the images.

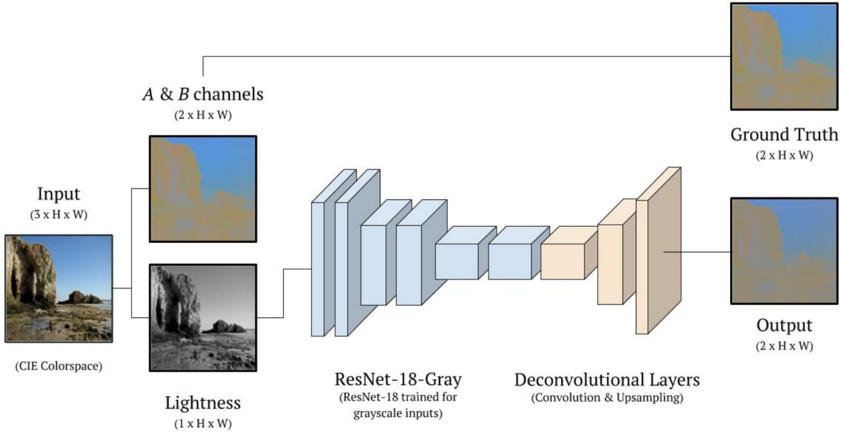
The beginning of the colorization model is the the first half of ResNet-18, of which the first layer of the network is altered so that it accepts grayscale input rather than colored input, and it is cut after the 6th set of layers and then given to the upsampling network.

Solution diagram / architecture



Start with low-level features, such as an edge. Layers closer to the output are combined into patterns. Then, they are combined into details, and eventually transformed into a face.

For the ResNet18, we first ran the architecture structure previously shared. Then, we defined a custom data loader to convert the images in the LAB space. As we train, we track the training loss and convert images back to RGB for prediction.



Implementation details

1st Attempt of CNN → Less deeper network, slower convergence, converts each color scheme by batching one at a time.

2nd Attempt of CNN → Change the LR from 0.01 to 0.001 to converge quicker. Also, we do not convert each batch one at a time, but instead convert all of them at once, and then feed the resized and properly formatted train data in. We also create a deeper network, more convolutional layers

Resnet18 → Use first half of Resnet and then upsampling to bring output to desired size

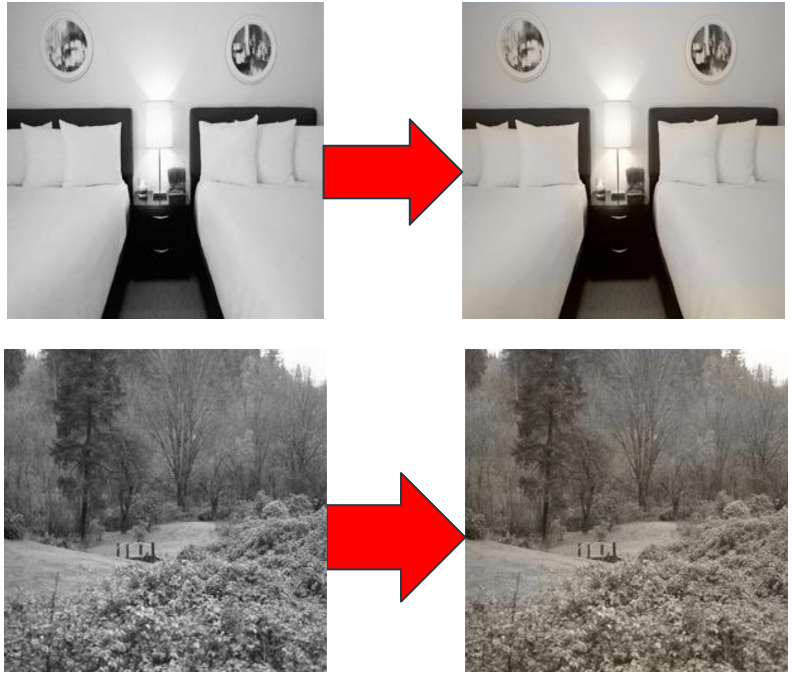
Demo / Experiment Design Flow

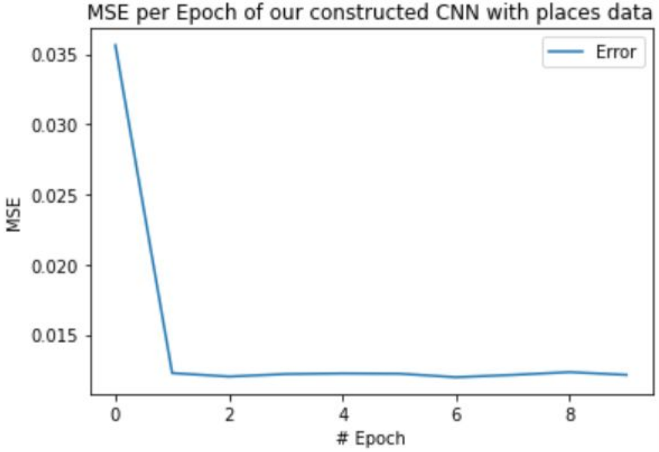
In the following slides of the experimental evaluation, we will look at the model performance for each architecture via observing the differences of

- MSE over epochs
- B&W, predicted images, and real images
- Old images of our families shown through the original CNN implementation as well as a realistic colored model through our Resnet implementation

Experimental evaluation

CNN Places Performance Before Optimization



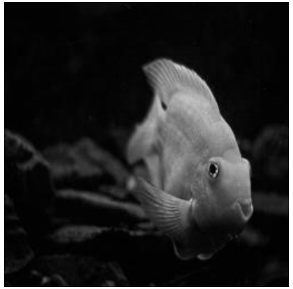


MSE per Epoch of our constructed CNN with places data

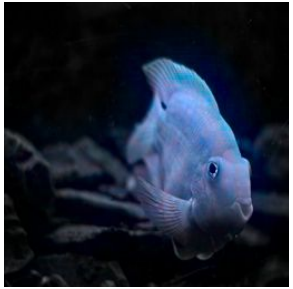
Our CNN implementation achieves low loss, but as we can see from our predicted colorized results, they still lack color and contain a brown-gray tone with some hints of coloring.
Brown is the safest color the model chooses.

Experimental Evaluation

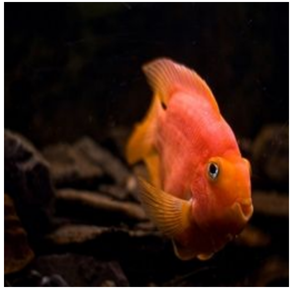
After CNN optimization: decreasing the LR, changing optimization as well as altering our code in order to convert all of our train and test images outside of the batch loop.



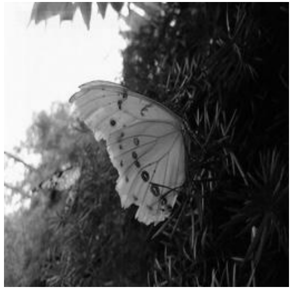
B&W



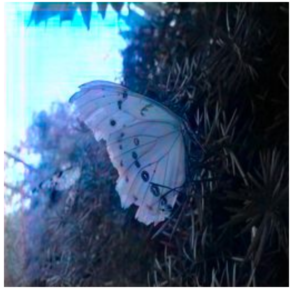
Predicted



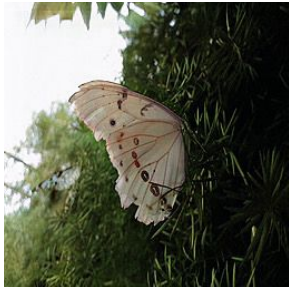
Real



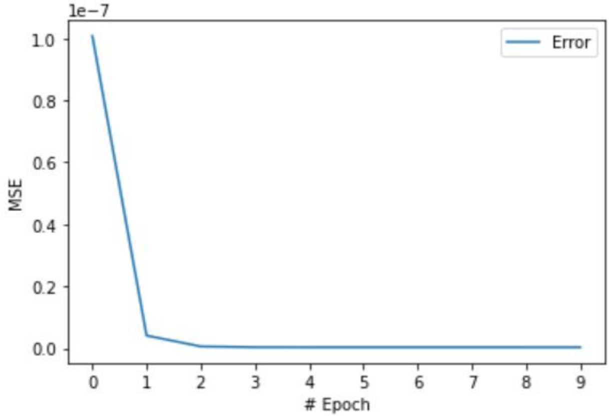
B&W



Predicted

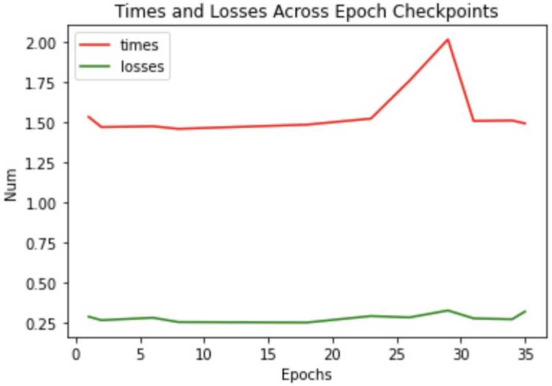


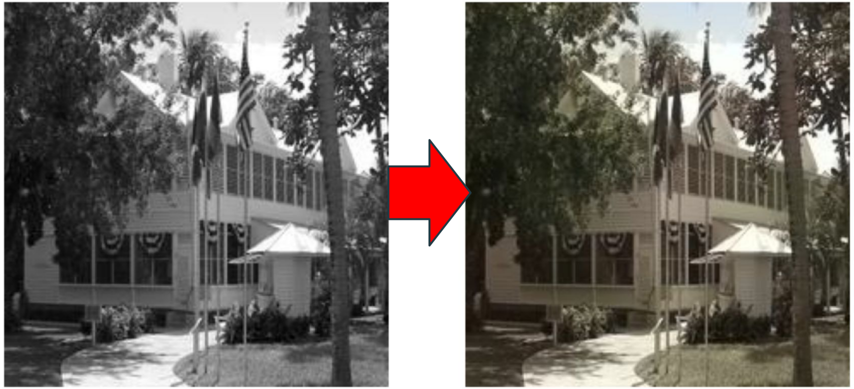
Real

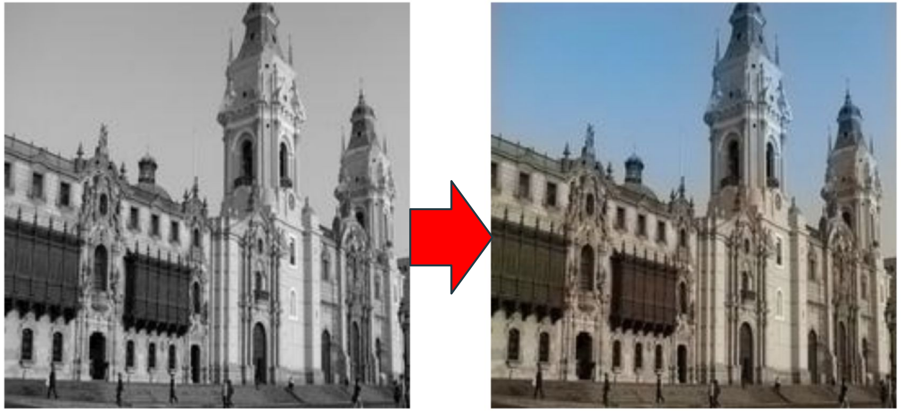


Notice that our MSE drops significantly!

Experimental Evaluation

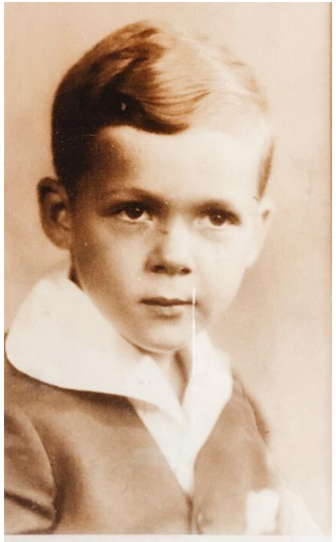




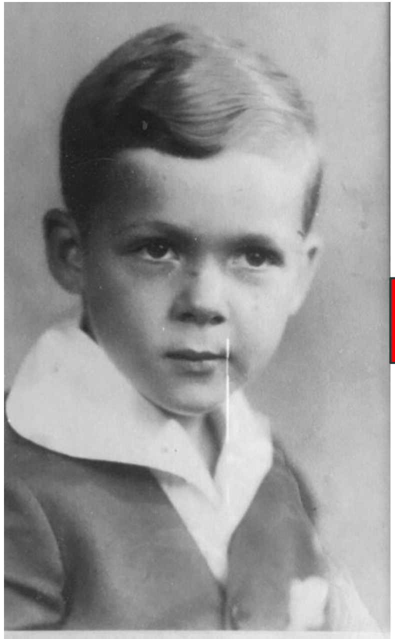


ResNet18 produces the best results after 10 epochs (loss: 0.28).
Even though the loss gets even lower (as low as 0.2497), the loss is not the best performance metric, and thus we found that of all checkpointed models, the pictures produced here were subjectively the most realistic

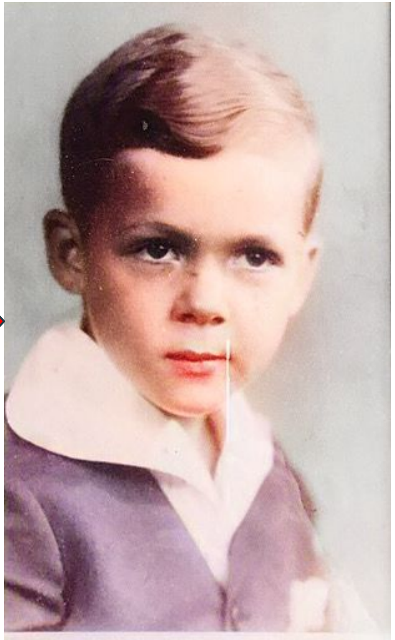
Experimental Evaluation



Using our first CNN

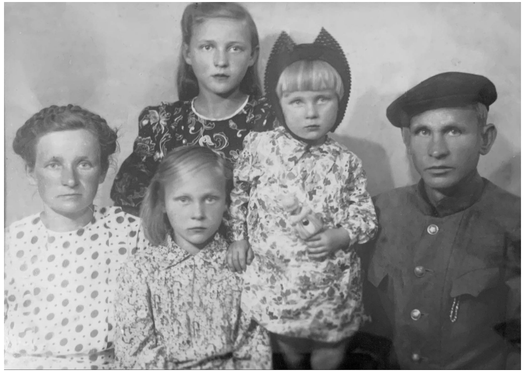


B&W of Eli’s grandfather

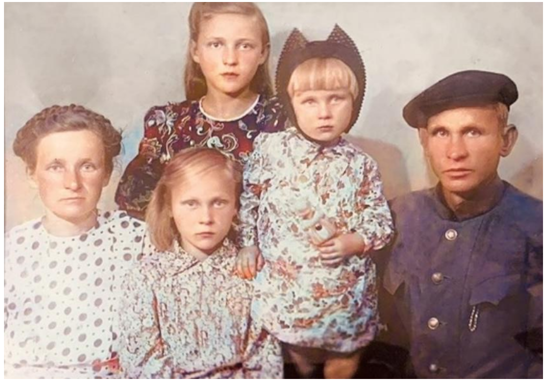


ResNet18 result

Experimental Evaluation

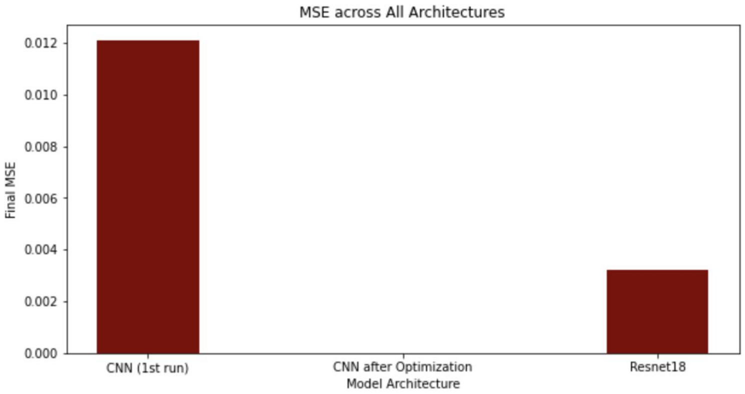


Sabina’s old
B&W photo!



Using ResNet18

Conclusion



Notice that CNN after optimization has the lowest MSE, yet is not the most subjectively realistic.

MSE and all other error measurements for CNN are imperfect, and do not necessarily represent ground truth in the way that they do in other NNs.

- After more research, we would have liked to experiment with GANs (with encoders and decoders) and U-nets as they generate better and more realistic images (more info on this here)
- We would also consider using ensemble multiple models stacking, such as by combining GANs with ResNet and U-net (as you can see here: link)
- If we had more compute power, we would have liked to implement this with more training data
- Using this project, a next step could also be to colorize videos

GitHub Repository

Thank you! (:

[link]