Reproducibility Report for Instance Aware Image Colorization

Osazee Ero
University of Waterloo
oero@uwaterloo.ca

Anshuman Gehlan
University of Waterloo
agehlan@uwaterloo.ca

Ruoxin Li
University of Waterloo
r444li@uwaterloo.ca

Reproducibility Summary

This paper highlights a modified implementation of the original paper "Instance-Aware Image Colorization [4]." The paper uses 3 sequential networks to colorize an image which are characterized by the following descriptions: full image colorization, instance level colorization and fusion of the full image with its instances. We elected to use their key insights of having 3 sequential networks but with different architectures, framework (Keras vs PyTorch), and a smaller data set for training. The paper used an RTX 2080 Ti on the ImageNet dataset with training time of 3 days to fine-tune a pre-existing model provided by the authors of the Real-Time User-Guided Image Colorization with Learned Deep Priors [3]. Our goal was to have training time be under 24 hours cumulatively for all 3 networks while still producing qualitatively similar results without any pre-trained weights.

Reproduction of the original architecture was not possible on our team's hardware within reasonable training times and storage constraints. Thus, we validated the papers claims of improved colorization using a system of networks on a smaller dataset. Experiments were conducted using an RTX 3070, RTX 2070 and GTX 970.

Furthermore, our experiments with different architectures and framework using the authors key insights without any pre-trained weights led to the same conclusions as the original paper. Our architectures performed well on both qualitative and quantitative (PSNR and SSIM) measures.

We found it easy and simple to run the completed model the authors provided. Furthermore, their code was easy to read, accompanied with a complete collection of helper code and scripts to replicate training on other data sets. However, we were not able to verify the original results produced from their methodology due to the time and resources at our disposal. Training of the networks is done in a sequential manner and because our GPU's had smaller memory sizes, the extended training time required was a key restriction.

We published our code on Github [6] for further experimentation with extensive descriptions of each model for other researchers wishing to extend this work. We proposed a few modifications to the pre-processing of the dataset which led to a 15% decrease in our training and validation loss.

The following paper is a reproduction of: Su, J., Chu, H., & Huang, J. (2020). Instance-Aware Image Colorization. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr42600.2020.00799

Throughout our experiments we did not communicate with the original authors.

1. Introduction

The original paper proposes a novel, end-to-end way of colorizing images that doesn't involve traditional approaches such as color scribbles. Furthermore, it achieves better results than similar research, most notably DeOldify [1]. The architecture proposed uses 3 neural networks, 2 of which have the same architecture. One network colorizes the entire image, another colorizes each object individually, and then the final network fuses the output of the first two networks weight maps (same architecture) to create the output image. This removes many previous limitations seen in similar research such as challenging foreground-background separation, multiple objects and most notably, different contexts in the same image (I.e., an orange on the road) [4].

The process begins with object detection using Mask R-CNN [2] where images that are not able to detect any object bounding boxes are removed from the data set. There are 13 convolutional layers for the colorization networks with the following number of channels: 64, 128, 256, 512, 512, 512, 512, 256, 256, 128, 128, 128, 128. This architecture is used from an earlier paper on colorization [3] with their pre-trained model fine-tuned in this implementation. Each network is trained sequentially starting with the full image colorization, instance and then fusion. Architectures are the same among the colorization networks to facilitate fusion, a key part in removing limitations surrounding blending objects back with the full image [4].

Related Background

Image colorization is an ill posed and challenging problem partly because instances can have multiple colors and be present in a variety of contexts. Color scribbles defined by the user were often used in conjunction with some form of segmentation and neural network to deal with this issue. The user defines the color of a few pixels that are then used to constrain and colorize the image. The paper alleviates some of this labor-intensive process by detecting and colorizing instances automatically. This works well when we have combinations of instances in different contexts that may not have been present in the training data but is still a difficult task in the cases where the instances themselves have a variety of colors they can be [4].

There also exist methods that use instance colorization. These current methods typically focus on single (FineGAN), non-overlapping (InstaGAN), and instance level boundaries (Pix2PixHD). The approach in this paper can handle multiple, overlapping instances due to their fusion module which alleviates these issues by blending the learned weight maps of the instance level images with the learned weight map of the full image [4].

The authors use qualitative, peak signal-to-noise ratio and structural similarity metrics to show their results outperforming similar work [4].

2. Scope of Reproducibility

The paper proposes a novel system of neural networks that leverage previously used architectures to color images producing results better than related research on metrics such as PSNR, SSIM and qualitative (visual).

- Claim 1: Using a system of full image, instance and fusion networks leads to qualitatively and quantitatively (PSNR, SSIM) better results than current research.
- Claim 2: Colorizing instances (foreground and background) separately will produce more visually appealing results and remove limitations posed by multiple contexts in the same image.

As will be described later, we were unable to replicate the papers original results using the author's code and dataset verbatim due to the training time and storage requirements. We do however report on results of running the author's code on a smaller dataset with minor tweaks to the hyperparameters more suited to our hardware. Furthermore, we took the authors key insights and intuition to build a similar model in another framework to try to achieve results that supported the authors claims.

3. METHODOLOGY

Our approach to replicating this paper was to implement based on the description provided in the paper. The authors provided the code to their work; however, the code was using the PyTorch framework and we found aspects difficult to understand. We were more comfortable with Keras and wanted to use this as a learning opportunity to experiment and build from ground up. Furthermore, the authors implementation was computationally intensive (~2 weeks of training time given our teams hardware and available hours a day for training) and required a lot of storage (ImageNet approximately 150 GB of data).

In this section, the architecture for the instance-aware image colorization is proposed. First, we briefly introduce the object objector for cropping out the image instances. Next, we talk about the pre-processing steps taken for the input image. Then, we introduce the colorization model backbone used for both full and instance image colorization tasks. Finally, we describe the fusion model for merging both models together to optimize both tasks in an end-to-end way.

Experiments were conducted using an RTX 3070, RTX 2070 and GTX 970. We constrained ourselves to have a training time for all 3 networks be under 24 hours while still producing reasonable results. Our code is available and can be found on GitHub [6].

Pretrained Mask-Region Convolutional Neural Network Detector

Like the paper approach, we first used an off-shelf pretrained network, Mask-RCNN [1] for detecting each object bounding box which we used for cropping the corresponding L (lightness) channel, A channel (from green (-) to red (+)) and B channel (from blue (-) to yellow (+)). The pretrained detector used is detectron2, a Facebook's open-source library for implementing state-of-the-art computer vision techniques. This is the "model_zoo" model which uses a residual networks architecture as its backbone and is trained using the COCO dataset [7].

Image Preprocessing

For training, we converted the RGB color image to the CIELAB color space. Next, we extracted the lightness channel as the input image to the colorization model, and the other two channels (a and b) as the target predictions of the model. We then normalize the L channel to lie within the

range 0 to 1, and the a,b channels to lie within the range -1 to 1. During preliminary testing, if we received a 1 channel grayscale image as an input, we repeated the grayscale image along its channel dimensions and carried out same process as described above.

Image Colorization backbone

We implemented a modified version of the image colorization backbone used by the authors. We tried several model variants ranging from using pretrained vgg16 architecture, to using residual networks. We finally decided on two model variants each based on the autoencoder architecture which consists of several convolution layers interleaved with batch normalization (BN) and leaky ReLU activations. This projects the input feature space into a smaller dimensional feature space, then reconstruct the original input feature space by using several up-sampling and deconvolution layers. We trained both the full and instance colorization model sequentially as described in the paper, using the weights from the trained full image model as the initial weights for the instance model. Figure 1 and 3 show the various variants for the image colorization backbone we experimented with.

Model description

Version 1

The architecture for the initial model we implemented is shown in figure 1. we took an inspiration from the work of Kamyar Nazeri, Eric Ng and Mehran Ebrahimi [5] in their paper titled “Image colorization and generative adversarial networks” U-Net Network Architecture. From figure 1, we can observe the model is symmetric with same numbers of encoding sections and decoding sections. The input encoding path consists of 4x4 convolution layers with strides of 2 for down sampling, then a batch normalization layer, and a leaky-ReLU activation with a slope of 0.2. After each encoding layer in the network, the number of channels is doubled, and image size is halved. The decoding path consists of 4x4 transposed convolution layers presided by batch normalization layers, and ReLU activation functions. Each mirror layer of the decoding path relative to the encoding path was concatenated together. The last layer is a 4x4 convolution layer with ‘tanh’ activation function since the target prediction lies within -1 to 1. The model summary of parameters is shown in figure 2.

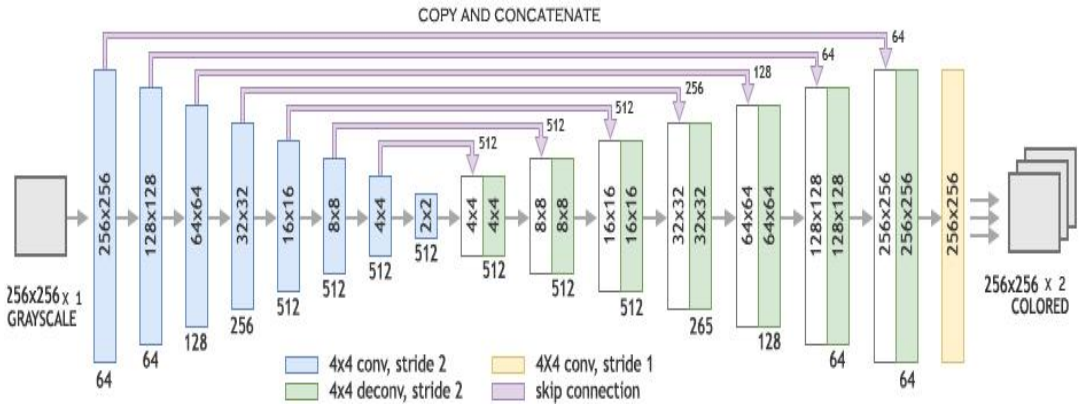
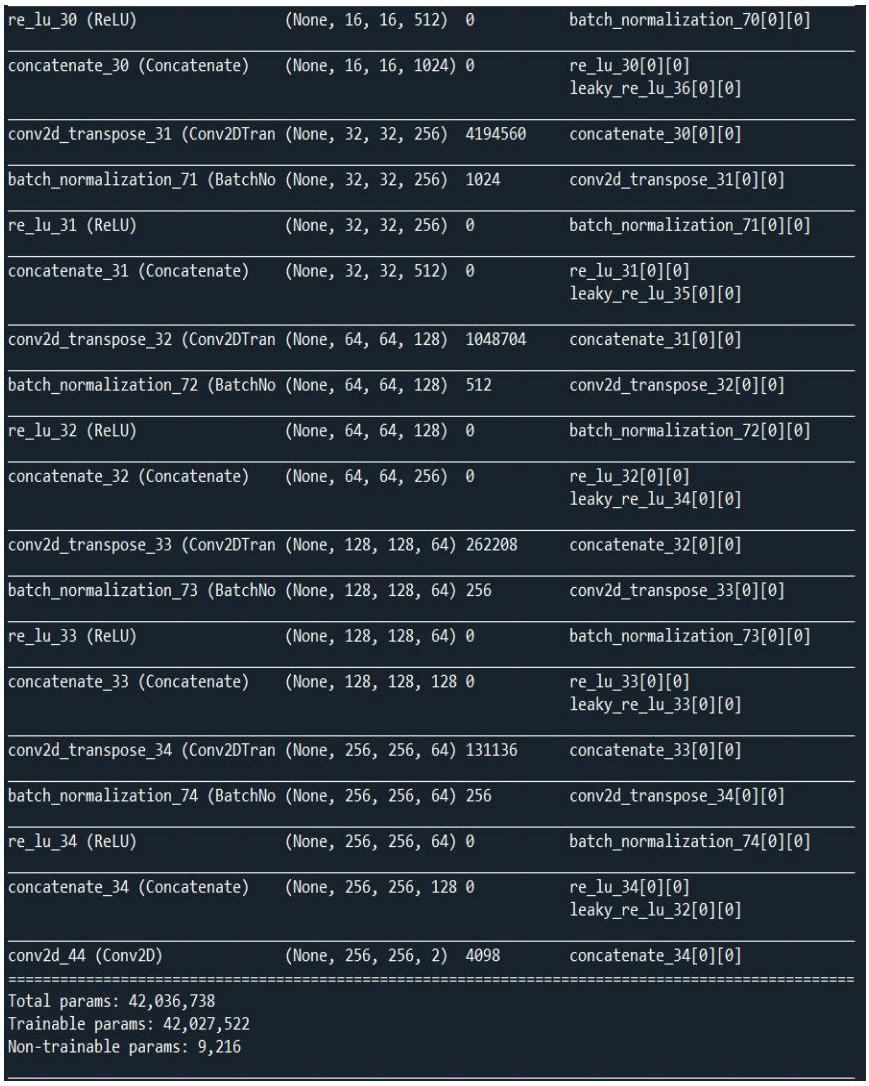


Figure 1: Image colorization backbone. Source (Modified) Image colorization and generative adversarial networks [5]



re_lu_30 (ReLU)	(None, 16, 16, 512)	0	batch_normalization_70[0][0]
concatenate_30 (Concatenate)	(None, 16, 16, 1024)	0	re_lu_30[0][0] leaky_re_lu_36[0][0]
conv2d_transpose_31 (Conv2DTran	(None, 32, 32, 256)	4194560	concatenate_30[0][0]
batch_normalization_71 (BatchNo	(None, 32, 32, 256)	1024	conv2d_transpose_31[0][0]
re_lu_31 (ReLU)	(None, 32, 32, 256)	0	batch_normalization_71[0][0]
concatenate_31 (Concatenate)	(None, 32, 32, 512)	0	re_lu_31[0][0] leaky_re_lu_35[0][0]
conv2d_transpose_32 (Conv2DTran	(None, 64, 64, 128)	1048704	concatenate_31[0][0]
batch_normalization_72 (BatchNo	(None, 64, 64, 128)	512	conv2d_transpose_32[0][0]
re_lu_32 (ReLU)	(None, 64, 64, 128)	0	batch_normalization_72[0][0]
concatenate_32 (Concatenate)	(None, 64, 64, 256)	0	re_lu_32[0][0] leaky_re_lu_34[0][0]
conv2d_transpose_33 (Conv2DTran	(None, 128, 128, 64)	262208	concatenate_32[0][0]
batch_normalization_73 (BatchNo	(None, 128, 128, 64)	256	conv2d_transpose_33[0][0]
re_lu_33 (ReLU)	(None, 128, 128, 64)	0	batch_normalization_73[0][0]
concatenate_33 (Concatenate)	(None, 128, 128, 128	0	re_lu_33[0][0] leaky_re_lu_33[0][0]
conv2d_transpose_34 (Conv2DTran	(None, 256, 256, 64)	131136	concatenate_33[0][0]
batch_normalization_74 (BatchNo	(None, 256, 256, 64)	256	conv2d_transpose_34[0][0]
re_lu_34 (ReLU)	(None, 256, 256, 64)	0	batch_normalization_74[0][0]
concatenate_34 (Concatenate)	(None, 256, 256, 128	0	re_lu_34[0][0] leaky_re_lu_32[0][0]
conv2d_44 (Conv2D)	(None, 256, 256, 2)	4098	concatenate_34[0][0]

Total params: 42,036,738
Trainable params: 42,027,522
Non-trainable params: 9,216

Figure 2: Model summary (version 1). Note, we used more parameters than the original papers implementation by approximately 8 million.

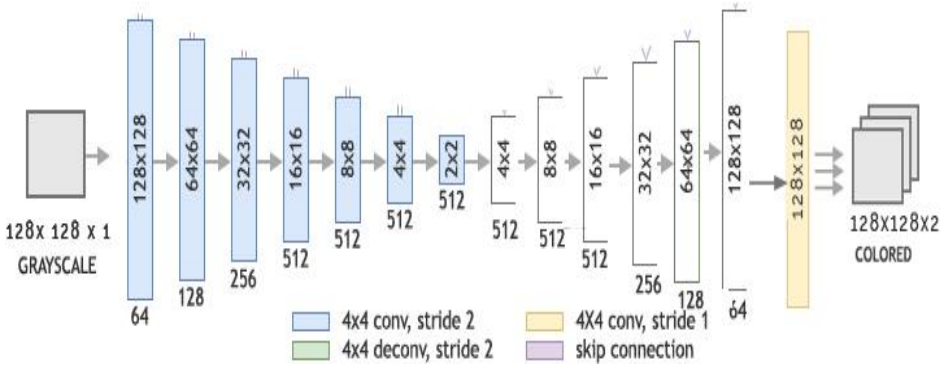


Figure 3: Image colorization backbone (version 2). Source (Modified) Image colorization and generative adversarial networks [5]

batch_normalization_94 (Batc	(None, 2, 2, 512)	2048
leaky_re_lu_53 (LeakyReLU)	(None, 2, 2, 512)	0
conv2d_transpose_41 (Conv2DT	(None, 4, 4, 512)	4194816
batch_normalization_95 (Batc	(None, 4, 4, 512)	2048
re_lu_41 (ReLU)	(None, 4, 4, 512)	0
conv2d_transpose_42 (Conv2DT	(None, 8, 8, 512)	4194816
batch_normalization_96 (Batc	(None, 8, 8, 512)	2048
re_lu_42 (ReLU)	(None, 8, 8, 512)	0
conv2d_transpose_43 (Conv2DT	(None, 16, 16, 512)	4194816
batch_normalization_97 (Batc	(None, 16, 16, 512)	2048
re_lu_43 (ReLU)	(None, 16, 16, 512)	0
conv2d_transpose_44 (Conv2DT	(None, 32, 32, 256)	2097408
batch_normalization_98 (Batc	(None, 32, 32, 256)	1024
re_lu_44 (ReLU)	(None, 32, 32, 256)	0
conv2d_transpose_45 (Conv2DT	(None, 64, 64, 128)	524416
batch_normalization_99 (Batc	(None, 64, 64, 128)	512
re_lu_45 (ReLU)	(None, 64, 64, 128)	0
conv2d_transpose_46 (Conv2DT	(None, 128, 128, 64)	131136
batch_normalization_100 (Bat	(None, 128, 128, 64)	256
re_lu_46 (ReLU)	(None, 128, 128, 64)	0
conv2d_60 (Conv2D)	(None, 128, 128, 2)	2050

Total params: 26,565,314
Trainable params: 26,557,250
Non-trainable params: 8,064

Figure 4: Model summary (version 2) of our final implementation. This has less parameters than the original paper's implementation.

Version 2

The number of parameters for model version 1 was over 42M parameters, which when used for both full and instance image colorization combined with the fusion model was about 95M parameters, so we decided to reduce the number of parameters by removing the concatenation layers in the version 1 model and reduce the input image size to 128. The total number of parameters for the model was then 26M as shown in figure 4.

Fusion Network

For achieving better colorization, we fused the feature maps at each layer of both the full image and instance colorization model. We implemented a modified version of the approach used by the authors where the fusion take place at the "jth" layer as shown in figure 5.

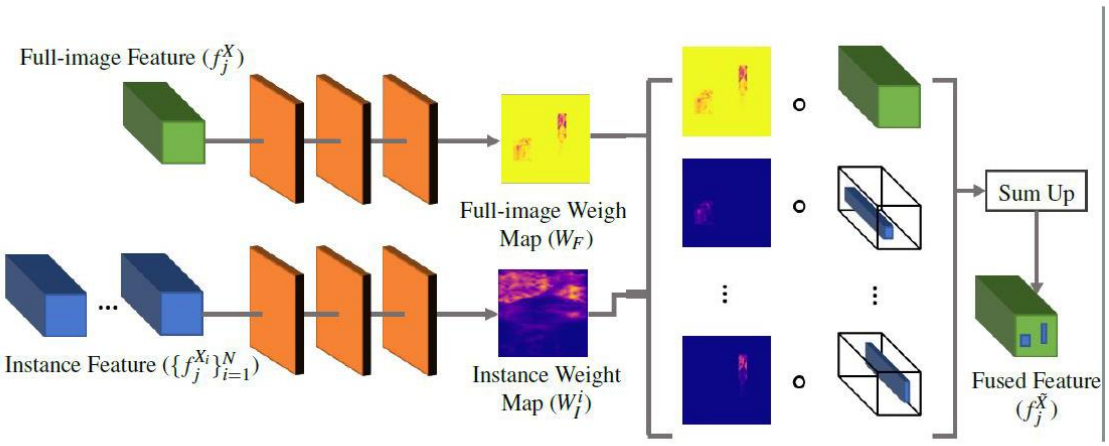


Figure 5: Fusion Network at a given layer. Source (Modified): Instance-aware Image colorization (Jheng-Wei Su, Hung-Kuo Chu, Jia-Bin Huang) [4]

The fusion takes place as follows: Given a full-image feature map, and multiple object instances, we predict two weight maps using two small neural networks of 3 convolutional layers. Then equation (1) was implemented for fusing both feature maps together.

$$F_j^X = F_j^X * W_f + F_j^{X_i} * W_I^i \ldots\ldots\ldots (1)$$

We tried a weighted implementation of equation 1, but the results were poor. We omitted the bounding box resizing and localizing in our implementation because we were getting out-of-memory errors while training the model. We determined that the resizing of the individual feature maps to the original size with the bounding box is extremely computational expensive and would require at least 8GB of graphics memory to implement successfully, which our hardware did not have. Figure 6 and 7 shows the model summary for both model variants we implemented.

```
batch_normalization_238 (BatchN (None, 256, 256, 64) 256
conv2d_transpose_32[1][0]
__________
batch_normalization_253 (BatchN (None, 256, 256, 64) 256
conv2d_transpose_39[1][0]
__________
re_lu_32 (ReLU)                 (None, 256, 256, 64) 0
batch_normalization_238[1][0]
__________
re_lu_39 (ReLU)                 (None, 256, 256, 64) 0
batch_normalization_253[1][0]
__________
simple_weight_model_42 (SimpleW (None, 256, 256, 64) 265482      re_lu_32[1][0]
                                                                 re_lu_39[1][0]
                                                                 input_12[0][0]
                                                                 input_11[0][0]
__________
concatenate_53 (Concatenate)    (None, 256, 256, 128 0
simple_weight_model_42[0][0]
                                                                 simple_weight_model_2
8[0][0]
__________
conv2d_312 (Conv2D)             (None, 256, 256, 2)  4098        concatenate_53[0][0]
==================================================================================
============
Total params: 95,522,712
Trainable params: 11,449,692
Non-trainable params: 84,073,020
```

Figure 6: Model summary (version 1).

```
simple_weight_model_69[0][0]
__________
conv2d_transpose_51 (Conv2DTran (None, 128, 128, 64) 131136      re_lu_50[1][0]
__________
batch_normalization_446 (BatchN (None, 128, 128, 64) 256
conv2d_transpose_45[1][0]
__________
batch_normalization_459 (BatchN (None, 128, 128, 64) 256
conv2d_transpose_51[1][0]
__________
re_lu_45 (ReLU)                 (None, 128, 128, 64) 0
batch_normalization_446[1][0]
__________
re_lu_51 (ReLU)                 (None, 128, 128, 64) 0
batch_normalization_459[1][0]
__________
simple_weight_model_70 (SimpleW (None, 128, 128, 64) 265482      re_lu_45[1][0]
                                                                 re_lu_51[1][0]
                                                                 input_18[0][0]
                                                                 input_17[0][0]
__________
conv2d_498 (Conv2D)             (None, 128, 128, 2)  2050
simple_weight_model_70[0][0]
==================================================================================
============
Total params: 63,133,444
Trainable params: 10,000,208
Non-trainable params: 53,133,236
```

Figure 7: Model summary (version 2) of our final implementation. This has less parameters than the original paper's implementation.

Datasets

We used the COCO-Stuff data set which can be downloaded from the following reference [7]. This is the same dataset the paper used for their fine-tuning and validation. There are 118287 images in the dataset, including some grayscale images. We first pre-process our data by detecting bounding boxes in the images and removing images of which no bounding boxes were detected. In this step, 2375 images were deleted from the original data set with 115912 images remaining. The process of acquiring all the bounding boxes took 8 hours on an RTX 2070, so we saved all the files generated to save time from running this operation again. A helper program in our implementation was used to delete images with no bounding boxes when setting up the environment again [6]. A link to download the bounding box data calculated by our team on COCO-dataset can be found from the following reference [8].

To save computational power and reduce the running time, we did not use the full dataset to train in our initial experiments. A random sample of 5000 images was chosen to test our program.

Before we started training on a larger dataset, we split all the images into two sets, training and validation. We trialed three different split ratios, 99/1, 40/60, 60/40, training and validation respectively. We found that the ratio 60/40 had the best result; however, decided to use 90/10 as it is a common ratio in literature and also achieved similar results to that of a 60/40 split.

From the COCO Explorer [7], there are 80 labels for all the images in the dataset. It shows as the result for 123,287 images and 886,284 instances, so on the average, there are 7.189 instances in each image.

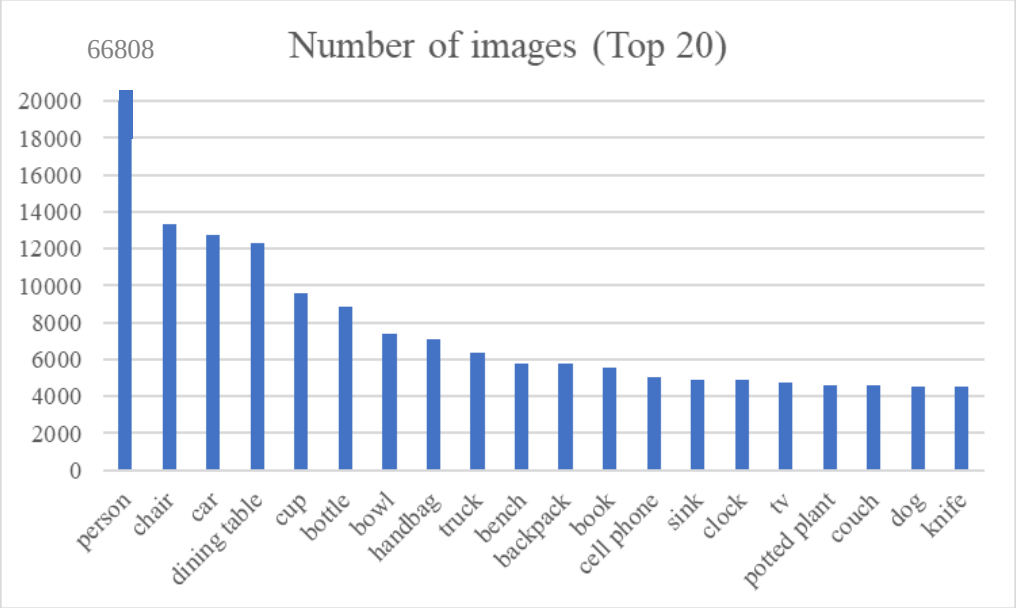


Figure 8: Distribution of the top 20 instance labels in COCO-stuff dataset. The class person is significantly higher than other object types.

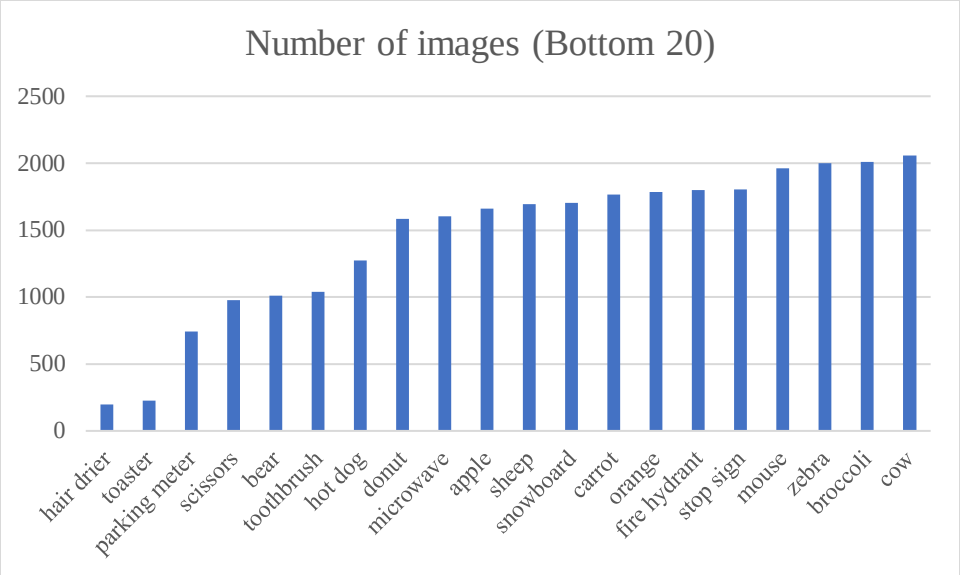


Figure 9: Distribution of the bottom 20 instance labels in COCO-stuff dataset.

From the data set with 115912 images left, there are total 642190 bounding boxes detected, and each bounding box has an associated accuracy score.

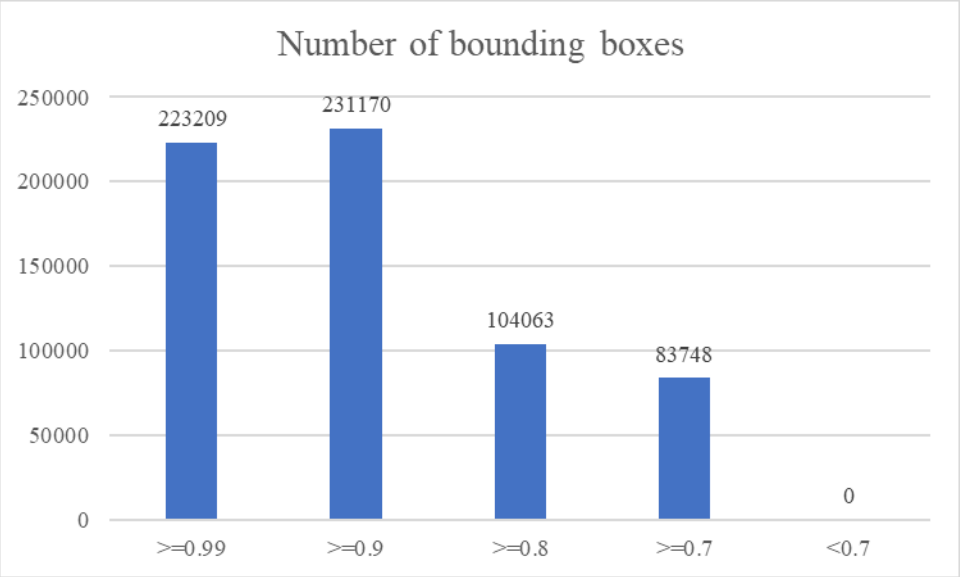


Figure 10: Distribution of bounding boxes based on their accuracy scores. Most bounding boxes have a score over 0.9, and over one third boxes have the score over 0.99.

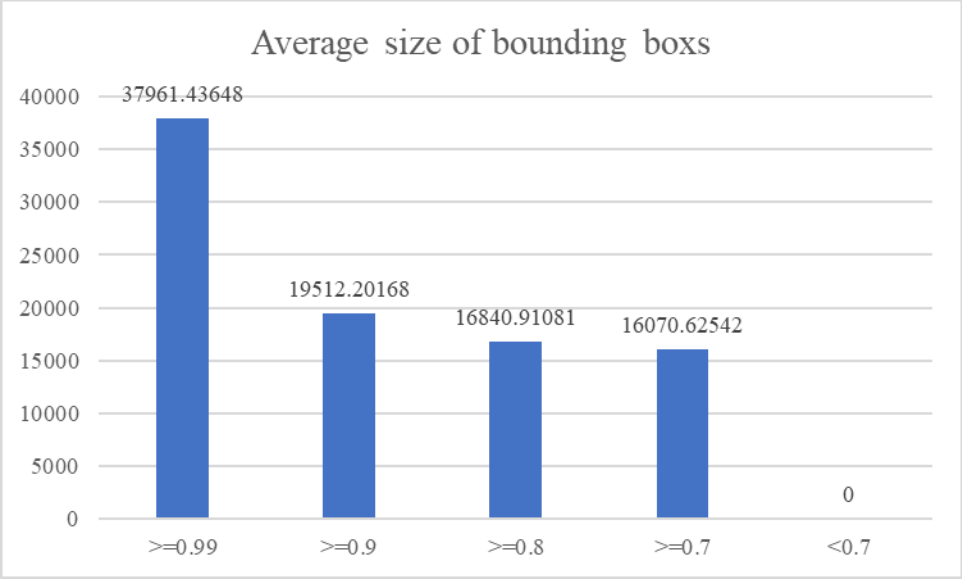


Figure 11: Average size of bounding boxes grouped by accuracy scores. The boxes with higher scores usually have a larger area.

Hyperparameters

When selecting the hyperparameters for model training, we took into consideration several factors such as the training time, the number of model parameters, and the input preprocessing steps on the images. The hyperparameter settings we used in our final implementation is shown in Table 1.

Table 1: Summary of hyper parameters used in our model

S/N	PARAMETER NAME	VALUE
1	Activation function	LeakyReLU, ReLU and TanH
2	Learning rate	1e-5
3	Loss function	Mean square error and Huber loss
4	Batch size	16

Activation Function:

We used the Leaky-ReLU with slope of 0.2, and ReLU, as the activation function for the model encoder and decoder sections respectively, and a TanH activation at the model output. We reasoned that during the backpropagation process, layers at the encoder section may suffer from the dying ReLUs problem, thus, we made use of the LeakyReLU activation to mitigate against this. We used the ReLU function for the decoder section as it is less likely to suffer from the vanishing gradients problem. When we converted from RGB to LAB, the L channel was normalized to the range 0 – 1, and color channels to the range –1 to 1. Therefore, tanh was used as

the activation function as a result of the preprocessing done on the input since its values lies between –1 to 1.,

Learning Rate, Loss function, Batch size:

The image colorization task is complex, and the tendency for the model to overfit is quite high, thus we opted to use a low learning rate of 1e-5, which helped prevent the model from overfitting. We used the mean square error loss function and the smooth-L1 loss used by the authors for training, and a batch size of 16 considering the computational resources.

Experimental Setup and Code

Various architectures and experiments were conducted in a manner that achieved visually reasonable results first. Afterwards, the models were evaluated on metrics the authors of the original paper used: PSNR and SSIM.

Our process followed the papers initial steps of first calculating bounding boxes of each image in our data set. This script for calculating the bounding box is the same as what the authors used and mentioned earlier in the report. This script removed any image in which bounding boxes were not found from the dataset as well as any grayscale images (1 channel only).

When training the instance network, additional pre-processing steps were undertaken to avoid passing images for training that resembled those in figure 12. These textures typically had a low certainty associated with them and in general the image had lower variance. For reference, the images in figure 12 typically had a variance of less than 1500 whereas the images in figure 15 had variances exceeding 5000. Thus, images that had a bounding box score of less than 99% and variance less than 2000 were not included in the training and validation process. From figures 10 and 11 we can see that this means we passed approximately 150,000 images into our network. These minor changes had an impact on our training loss and validation loss of approximately 0.001 units (approximately 15% difference).

We also reduced the image size to a 128-pixel square to speed up training time compared to the 256-pixel square used in the original paper's implementation.

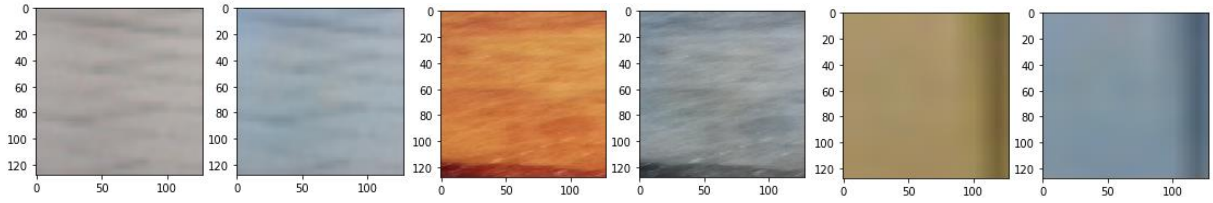


Figure 12. Each pair of images represents an instance detected by Detectron2 (the detector used in the paper) coming from different images and resized to a 256-pixel square. From left to right the images are from the beach, a pie, and bathroom.

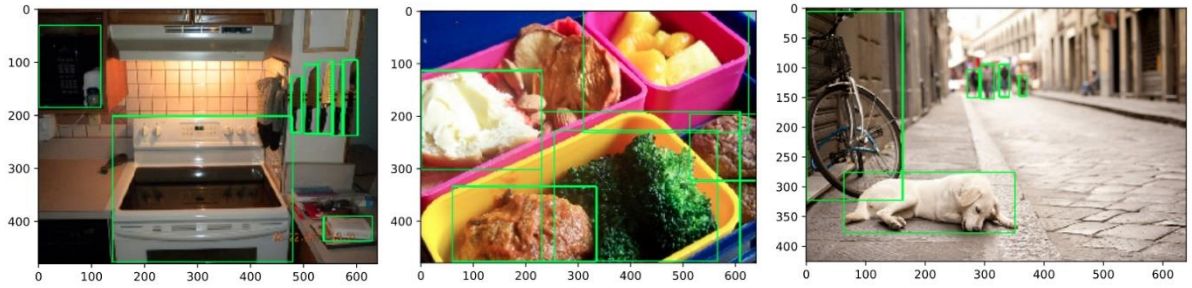


Figure 13. Sample images with associated bounding boxes. The left image has confidences all exceeding 98%; however, half the microwave is not in the image. This microwave instance image when resized and isolated is hard to classify when not in context of a kitchen. The middle image presents a similar challenge with some items such as what appears to be an occluded cookie on the right side of the image with a confidence of 78%. Finally, for the image on the right, some of the pedestrians in the background appear as smears resembling the pictures in figure 6 when cropped and resized.

Link to original papers GitHub [4] and our implementation [6] can be found in the references.

Computational Requirements

The hardware used on our team consisted of 3 GPUs, all by Nvidia: RTX 3070, RTX 2070 and GTX 970. We made attempts to use Google Collab but found that our hardware in general was faster and not constrained to time limits or cloud storage constraints.

Table 2. This table shows the amount of time it took to train our networks with an RTX 2070 as a reference. It was found that RTX 3070 operates 30-40% faster than RTX 2070 during training.

Network	**Instance Images**	**Batch Size**	**Step Time**	**Time / Epoch (110k / 10k)**
Full Image	N/A	16	~600 ms	~ 1 hour
Instance	10 max / img	16	~600 ms	~ 1 hour
Fusion	N/A	1	~205 ms	~ 4 hours

In general, we ran cumulatively, approximately 40 experiments totaling 300+ hours of GPU time. A snapshot of some our different trainings can be found in figure 15 and the appendix. We saw that implementations that led to low training and validation losses were usually in the order of 1e-2 to 1e-4. The average time taken to colorize an image was approximately 0.5 seconds. Training the full image and instance colorization network for approximately 8 epochs is about 8 hours. Training the fusion network for 2 epochs is also approximately 8 hours as summarized in Table 2.

4. Results

Overall, we can train 3 colorization networks without any pre-trained weights, in under 24 hours, with a dataset 1/10th the size of the original paper and produce reasonable results. We were also able to loosely validate the papers original claim that having an additional instance network will

lead to better results visually, and quantitatively using SSIM and PSNR. We experienced difficulties in implementing the fusion network but did see improved results when colorizing with the instance network instead of the full image colorization network. We also validated the authors original work on a reduced data set and computational load validating their claims.

Results Reproducing Original Paper

We chose to run the code provided by the authors with the following changes. The image size was decreased to 128x128 because the size 256x256 will run out of GPU memory. Their model was trained on an RTX 2080ti which has 11 GB of memory, but our GPUs have a maximum of 8 GB of memory. Next, we trained their model from ground up using the COCO-stuff data set whereas they used pre-trained weights and ImageNet.

We trained full and instance network for 3 epochs, and only 1 epoch for the fusion network.

When training, the authors code calculates the number of epochs to train. The default settings led to 150 epochs for each of full and instance network, and 30 epochs for the fusion network. This was not feasible in our time frame as the fusion network alone would be 3 weeks of training time on our fastest GPU (RTX 3070).

Table 3. This table shows the amount of time it took to train networks in the paper with an RTX 3070

Network	**Batch Size**	**Time / Epoch (110k / 10k)**
Full Image	16	~30 mins
Instance	16	~30 mins
Fusion	1	~ 5 hours

Figure 14. The left column is the result from the trained model provided by the paper. The right column is the result from our reproduced model using a smaller dataset with the pre-trained weights.

Compare the results in figure 14, when trained with less epochs, the smaller area in the images will be colorized. Due to the fact that only after a few epochs the results returned are reasonable, we chose to train the model for less than 10 epochs to save a significant amount of training time while still being able to validate their claims.

Colorizing an image takes approximately 0.11 seconds on an RTX 3070. This was trialed by colorizing 300 images and taking the average time for an image. 300 images took 32 seconds using the full colorization network.

Results Beyond Original Paper

Most of our time was done experimenting with architectures beyond the original paper. Due to the fact that we ran our experiments in Keras, compared to Pytorch used in the paper, we were not able to take advantage of the pre-trained weights. Many unsuccessful attempts were made to convert the pre-trained weights and load it into a Keras architecture. We found this a good opportunity to experiment with different architecture models and train without any pre-trained weights. We experimented with different input image sizes, learning rate, and architectures that led to different results as shown in figure 15.

Figure 15 shows images of the various architectures that were trained on the COCO stuff dataset. Each architecture ranged from 10 to 50 million parameters and was trained between 5-8 epochs. We can visually see the differences and improvements made to the architectures trialed as time went on starting from the top left and moving towards bottom right by each row.

Figure 15. Each image is from a different architecture trained from ground-up using the COCO-stuff dataset. For each set of images, we have the real image on the left and the predicted image using our network on the right. Only the full scene network was used for colorizing images in this figure.

The progression of our experimentation and result highlighted in figure 9 in our opinion was qualitatively good. We go from almost no colorization to visually more appealing colorization of the zebras in grassy field. At this stage we moved forward to training our instance network based off our full image colorization weights and calculated other metrics such as PSNR and SSIM in table 3 below.

Figure 16. During our experimentation, we came across an interesting result for one of our images. Here we can see that the original image on the left does not have a picture of the sun but when we input a grayscale image to our model, it colored in the sun on the horizon.

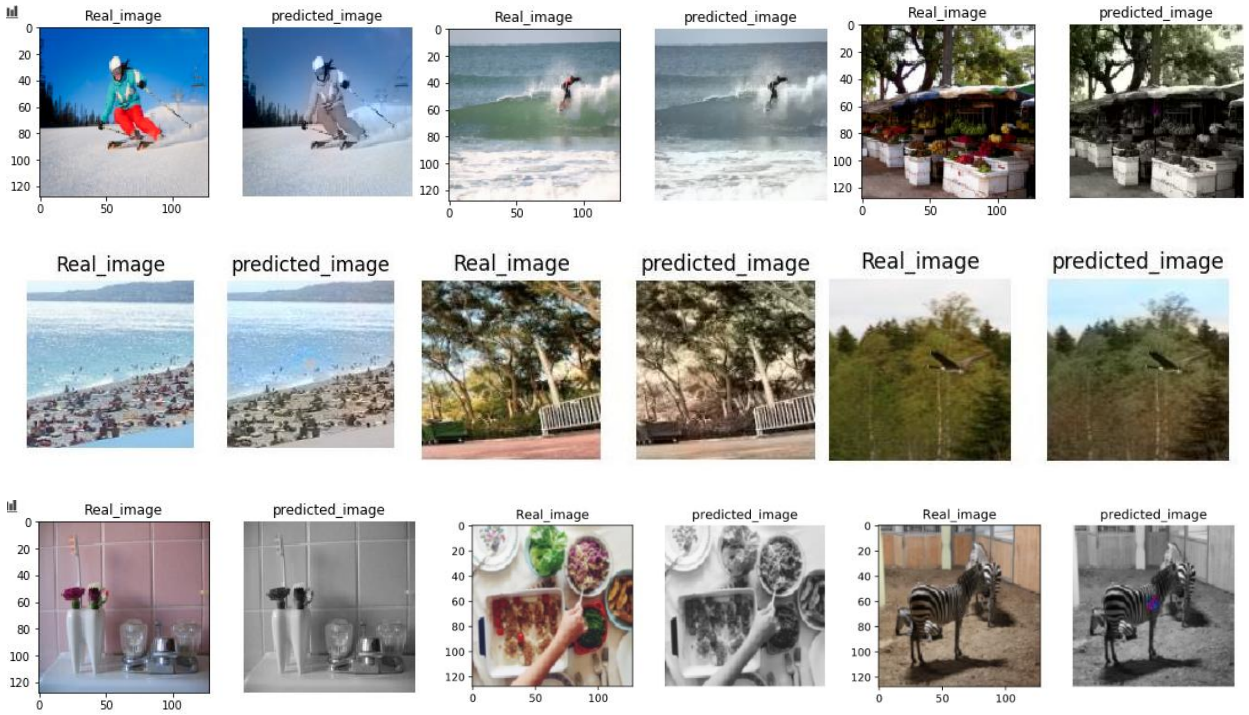


Figure 17. Top row represents colorization from the full image network, middle row represents colorization from the instance network and finally the bottom row represents colorization with the final fusion network.

From figure 17 we can see that our final architectures for full image and instance images are able to colorize reasonably well. In the case of the full image colorization, we can clearly see the difficulty in colorizing individual instances, however scenery and background are done quite well. The instance image colorization takes the weights of the full image colorization and then trains only on the instances of the dataset images. The images produced by this network seem to be colorized better than just the full image colorization as expected. This is also quantitatively shown in our PSNR and SSIM metrics that we calculated. Finally, we were unable to have a successful implementation of our fusion network which is the bottom row of figure 17. Although we tried different implementations and methods of fusing weight maps, our results would end up being gray scale images. In some of our implementations, we would get colorization of the images but in a manner that made no sense as shown in figure 18.

Figure 18: Fusion network producing weird results but colorizing the image.

Colorizing an image takes approximately 0.53 seconds on an RTX 2070. This was trialed by colorizing 100 images and then taking the average time for an image. 100 images took 53 seconds using the full image colorization network.

Table 4: Quantitative metrics of our full image colorization network

Implementation	**PSNR**	**SSIM**
DeOldify	23.923	0.904
Original Paper	29.522	0.929
Reproduced	24.066	0.894
Ours Full Image	20.176	0.827
Our Inst. Image	21.049	0.888
Our Fusion	21.216	0.902

5. Discussions

Our experimental results both reproducing the paper using a smaller data set and our own networks without any pre-trained weights support the claims of the paper. We believe our implementation performed reasonably well compared to the original paper and DeOldify on both quantitative and qualitative metrics as highlighted in table 4. Given a larger dataset with pre-trained weights, we believe our implementation can be improved greatly.

During our experimentation, we tried varying the following parameters:

- Rate scheduling
- Pre-trained VGG weights
- Residual network architecture
- Several activation functions for output of the model (sigmoid, tanh and then none for final steps)
- MSE and Huber Loss
- Input size / batch size / number of instances

The conclusions drawn from these various experiments ultimately were not meaningful (thus not mentioned) because we realized a few bugs in our data generation code that was the root cause of the poor results we saw early on (top row of figure 15). As mentioned earlier, we also noticed that many images like that of figure 12 were being passed into training of our instance network which was leading to poor losses. To correct this issue, we only let instances with a high bounding box score and variability be passed into our network leading to approximately 15% decrease in both our training and validation loss. This highlights the importance of training the networks on quality data.

Next, we also see that some types of scenes were colorized better than others. In particular any scene that was outdoors (grass, water, trees or sky) was able to produce results that in some cases were visually more appealing than the original image like the bottom right image of figure 15. We believe this was the case because outdoor scenes do not have much variability in their colors

than indoor scenes do. For example, grass, water, trees any the sky typically have an obvious color associated with them (low color variance and probability), but objects in indoor scenes or of people have a much higher variability with respect to their color probability.

Finally, when comparing quantitative metrics such as PSNR and SSIM in table 4, we can see that our implementation is relatively good when considering the lack of pre-trained weights and dataset size. Furthermore, implementation of our method and the original authors method on the smaller data set also validated the claim that a system of neural nets specialized in full scene, object and fusion outperforms a single neural net for colorization. We saw a difference of 1 unit in our PSNR value going from full image colorization to instance image colorization.

What was easy

In general, we found the authors code very well written and clear. They provided many additional scripts that allowed for training on different datasets and helper code that we used in our implementation for pre-processing.

What was difficult

We initially had difficulty in getting the model to stop overfitting, we later realized it was the implementation of the custom data generator for loading the full and multiple image instances. After fixing the bug, our model performed fairly well on the colorization of natural scene environments, but not so good at colorizing object instances. We concluded it may be as a result of the diversity of our training data. We also had difficulty in understanding the authors implementation of the fusion network, and the implementation of our understanding of the authors approach was extremely computational expensive as the resizing of each feature maps for each image instances lead to out of memory errors.

Recommendations and Next Steps

In this work we have demonstrated the use of deep learning techniques for automatic image colorization, we have attempted to reproduce the authors approach to this task, which involves sub-dividing the process into three sections: full image colorization, instances image colorization and a combination of the two previously mentioned sections for the final automatic colorization of an image. Moving forward, we have identified several areas the proposed approach which could be further improved. First, the fusion network proposed by the authors is computationally expensive as the resizing of individual feature maps for each image instance would require high system memory, thus further research is needed to seamlessly blend both feature maps together that would require less computational resource. Next, the authors were including images in training the instance network that did not represent any object, like the textures seen in figure 12. A more specialized or pruned dataset and additional pre-processing can be trialed to see if colorization improves for specific image scenes than others, particularly at the boundaries of different objects. Finally, we believe other metrics rather than PSNR and SSIM should be experimented with when measuring success of colorization. PSNR and SSIM did not convey the relative qualitative differences seen between the different colorization networks trialed.

Our team did not have any communication with authors of the original paper.

References

1. Jason Antic.jantic/deoldify:A deep learning basedproject for colorizing and restoring old images (and video!).https://github.com/jantic/DeOldify,2019.
2. Kaiming He, Georgia Gkioxari, Piotr Doll´ar, and Ross B.Girshick. Mask r-cnn. InICCV, 2017.
3. Richard Zhang, Jun-Yan Zhu, Phillip Isola, Xinyang Geng,Angela S. Lin, Tianhe Yu, and Alexei A. Efros.Real-time user-guided image colorization with learned deep pri-ors.ACM TOG (Proc. SIGGRAPH), 36(4):119:1–119:11,2017.
4. Su, J., Chu, H., & Huang, J. (2020). Instance-Aware Image Colorization. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr42600.2020.00799
5. Nazeri, K., Ng, E., & Ebrahimi, M. (2018, July). Image colorization using generative adversarial networks. In *International conference on articulated motion and deformable objects* (pp. 85-94). Springer, Cham.
6. Ero, O., Gehlan, A., Li, R. (2020, Dec). Replication of Instance Aware Image Colorization. https://github.com/osazee-ero/image_colorization
7. Lin, Maire. "Microsoft COCO: Common Objects in Context." In Computer Vision – ECCV 2014, 740–755. Cham: Springer International Publishing, n.d. https://cocodataset.org/#download
8. Bounding Box dataset for COCO-Stuff: https://uofwaterloo-my.sharepoint.com/:u:/g/personal/r444li_uwaterloo_ca/EVvHa-Nq9tVPghjlZ4JHfvkBu-KhIpV6uPZjiESHdiLcFw?e=FOhTeL

Appendix

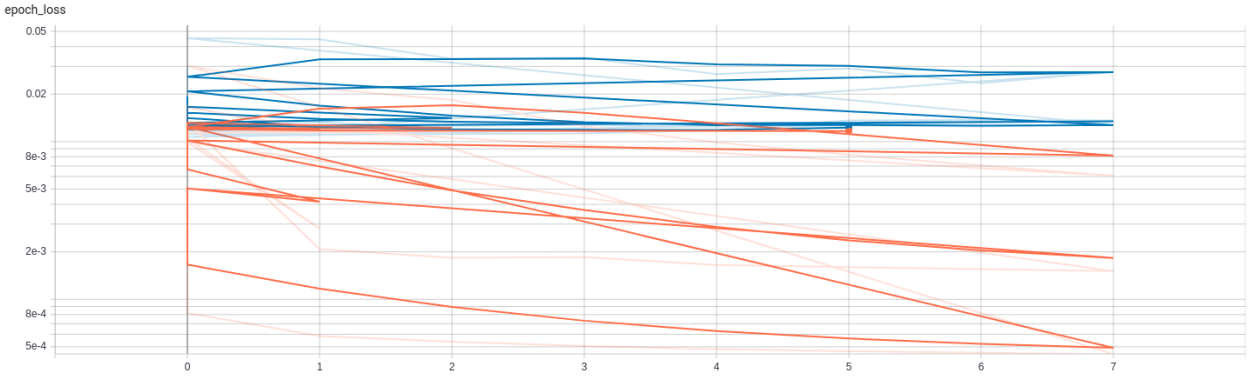
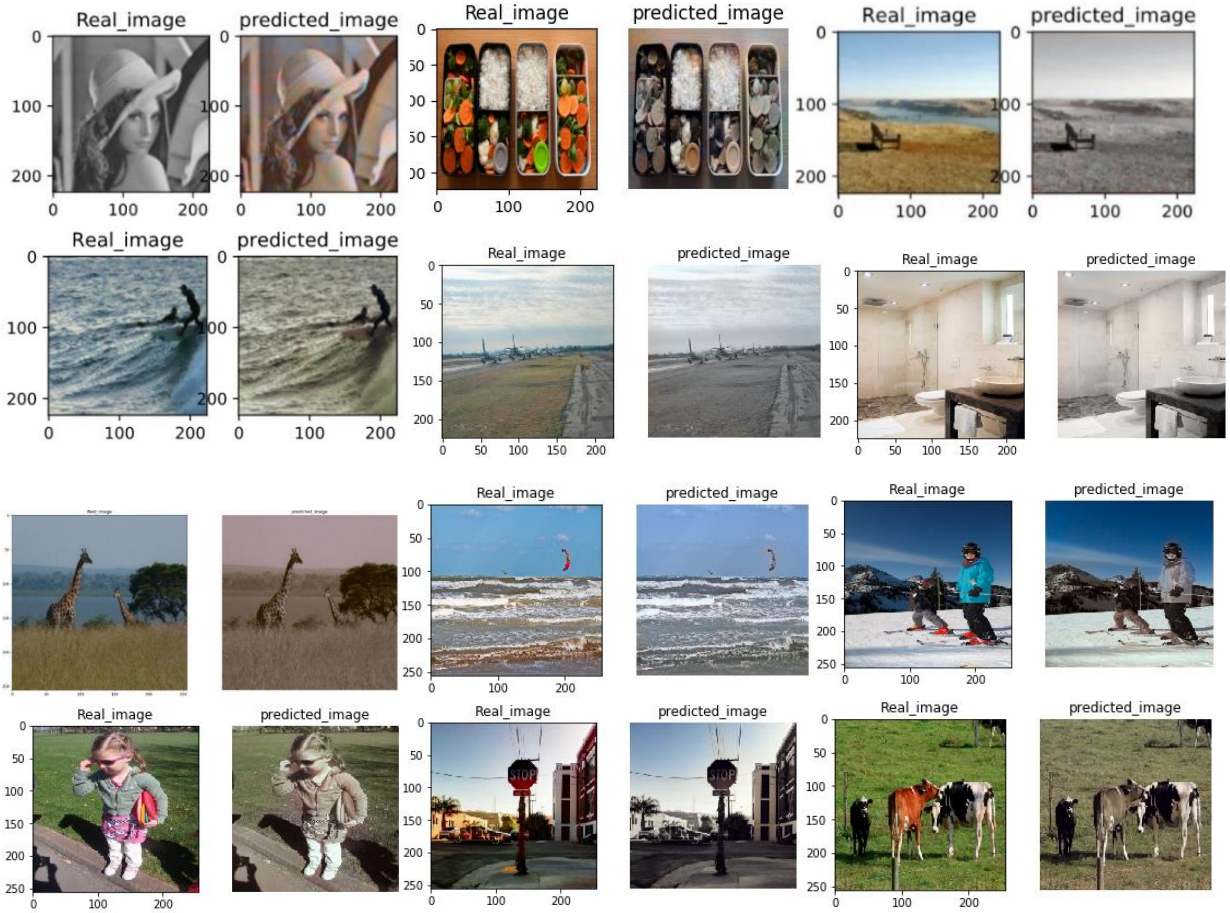


Figure 19: Tensorboard visualization of our loss as a function of epochs for a few early experiments trialed. Orange is training loss whereas blue is validation loss.



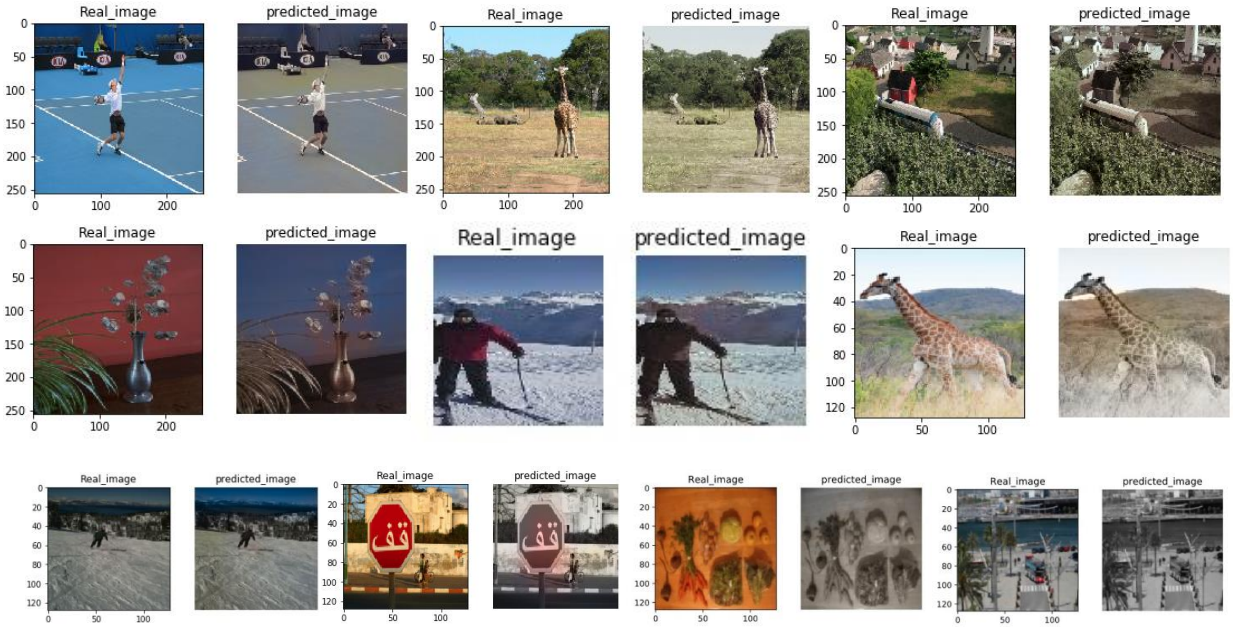


Figure 20: Collection of image colorizations for the various architectures trialed.