Colorization of video games' images using convolutional autoencoders

Capstone Project for Udacity's Machine Learning Engineer Nanodegree

Eryk Lewinson, April 2020

1. Definition

1.1 Project Overview

For the capstone project, I wanted to work on a project that would be genuinely interesting to me. For a long time already, I was intending to familiarize myself with a domain of computer vision, namely image colorization. From as long as I can remember, I was a big fan of video games and that is why in this project I wanted to work with something close to my heart.

Recently, I saw a few posts on the Internet [4] showing that by using Deep Learning it is possible to enhance the quality of emulated video games. By emulation, I mean running a game using dedicated software on a system different than the one the game was originally created for. An example could be playing Nintendo 64 games on a PC. By embedding pre-trained Neural Networks in the emulator software, it is possible to upscale the resolution to 4K or to increase the quality/sharpness of the textures. What is really amazing is that these solutions work out of the box for all games, not only for one or two games on which they were directly trained.

Of course, doing a project on such a scale is much more complex than what I intend to do for the capstone. When I was starting my adventure with video games, I was playing on Game Boy Color, which aside from new games in color also worked with the previous generation - the grayscale Game Boy. Taking inspiration from those times, I will try to use Deep Learning to colorize grayscale images, so an emulator (in combination with a network trained on many similar games) could potentially colorize the output close to real-time.

1.2 Problem Statement

The problem I would like to tackle in this project is quite simple – to colorize grayscale images coming from old-school video games to a satisfactory degree. To do so, I use Deep Learning, in particular, autoencoders based on Convolutional Neural Networks (also known as convolutional autoencoders). All the models are developed in PyTorch and trained as *training jobs* in AWS's SageMaker.

I use a self-gathered dataset of images coming from a Game Boy Color video game called *Wario Land 3* (more on the process of gathering the data will follow in a later part). The dataset can easily be expanded to include more than one game.

1.3 Metrics

It is quite hard to select a good evaluation metric, as the coloring is, to a great extent, subjective and different versions can be considered acceptable. That is why often human observation is required. In the literature ([1], [3]), a popular solution for training colorizing Neural Networks is to use the Mean Squared Error (MSE) as a loss function. In this project, I follow the same approach.

However, investigating different evaluation functions is definitely an area worth exploring. Some papers [5] also explore the idea of framing the image colorization as a classification problem (instead of the suggested regression) and using metrics such as accuracy as the loss function.

In this report, I present the MSE of selected network architectures at different epochs (15th, 30th and the best one, in case it was not the last one). Additionally, I present the colorized images for visual inspection.

2. Analysis

2.1 Data Exploration

For the colorization project, I used one of my favorite games from my childhood - *Wario Land 3*. To obtain the dataset, I captured a video from YouTube (https://www.youtube.com/watch?v=btnFUbexxEE). I found a “longplay” (a complete playthrough of the entire game, only showing the actual game screen), so after downloading it, I can easily extract every *x*-th frame from the video to have a full dataset of color images.

The video plays 24 frames per second and I extracted every 58th frame. Doing so resulted in a total of 7640 images.

A potential problem with the presented approach is that the dataset contains:

- all in-game menus
- introductory screen
- credits
- screen transitions - when switching screens/stages in-game, there is often a short phase-out screen when the next screen/stage loads

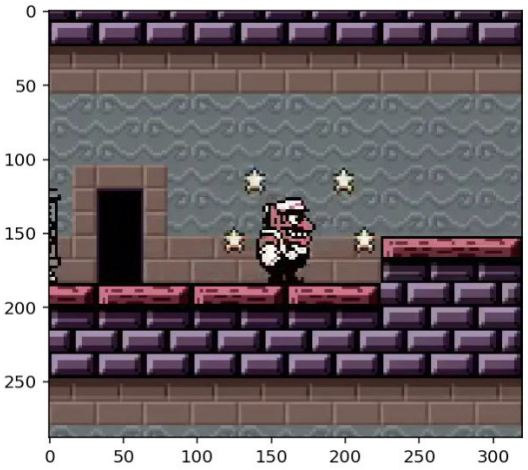
Altogether, this can introduce some noise to the dataset. However, the only solution that I can come up with is to either manually remove such screens or, for example,

cut out the first/last *X* seconds of the video to remove the intro/credits. For simplicity, I do not do it and leave all the frames in the dataset.

For a thorough description of the pre-processing steps and a clear definition of inputs and outputs of the model, please see section *3.1 Data Preprocessing*.

2.2 Exploratory Visualization

The raw downloaded data is stored as JPG images, each one of the size 288x320 pixels. The following image presents a sample image:



I believe this is a good place to mention some details about Game Boy Color's technical side. The screen's resolution is 144x160 (so the captured video is effectively upscaled by a factor of 2). Moving on to the color palette, the Game Boy Color systems use a 15-bit RGB palette (up to 32,768 possible colors). In practice, it was possible to present up to 56 colors without using any special programming techniques, while the full 32,768 colors required some tricks.

The current standard is the 24-bit RGB color space, in which each color channel is expressed as a number between 0 and 255. Such representation results in a total of ~16.7 million color combinations.

The GBC's color somewhat "limited" palette is one of the reasons I picked it as the target platform, as it should be easier to colorize such images rather than the ones from the newer generations of consoles or real photos.

2.3 Algorithms and Techniques

After a literature review, I believe that the two most popular approaches to image colorization are based on:

- Autoencoders based on Convolutional Neural Networks
- Generative Adversarial Networks (GANs)

I considered both approaches and decided to use the first one. The reasons for that are:

1. On average, GANs take longer to train and I was limited by the AWS credits and time to submitting the capstone
2. GANs require much more expertise in applying changes to the network architecture and I wanted to play around with the number of layers, filters, etc.
3. I wanted to do a comparative study and try out at least a few architectures
4. There are many variants of autoencoders: plain, denoising, variational, etc.

Next, I provide a very brief high-level overview of the techniques used in the project.

Autoencoder
Autoencoders are a type of Neural Networks that are similar to techniques such as the Principal Component Analysis - they both try to reduce the dimensionality of the data. However, autoencoders can do much more. They are built from two components:

- encoder - transforms the input data into a lower-dimensional representation (also known as the latent vector/representation). To achieve this goal, the encoder must learn only the most important features of the data.
- decoder - recovers the input from the low-dimensional representation.

I train the autoencoders to minimize the loss (in this case the MSE), which is a metric of the difference between the input data and the decoded output. Often, the latent representation can be used as the extracted features, such as the principal components from the PCA. In our case, however, we are interested in the decoded output.

An important thing to be aware of while training autoencoders is that they have a tendency to memorize the input when the latent representation is larger than the input data.

Convolutional Neural Networks
The goal of the convolutional layers is to extract potential features by applying convolutional filtering. The convolutional layers read the input (such as a 2D image) and drag a kernel (of a specified shape) over the image. The kernel represents the

features we want to locate in the image. For each step, the input is multiplied by the values of the kernel and then a non-linear activation function is applied to the result. This way, the original input image is transformed into a filter map.

Convolutional and pooling (aggregating) layers can be stacked on top of each other to provide multiple layers of abstraction. Many popular image classification architectures are built in a similar way, such as AlexNet, VGG-16, or ResNet.

In this project, I combine the CNNs with the autoencoders and effectively use a class of architectures known as convolutional autoencoders. The input of the network is a grayscale image (1 channel), while the outputs are the 2 layers representing the color. For more information, please read about the *Lab* encoding in 3.1 Data Preprocessing.

Next, I describe the three architectures (named V0, V1, V2), which I decided to include in the project. During the project, I have also experimented with many different ones.

Model V0

The model's architecture is based on the one presented in [1]. The author used it to make a point about colorizing one image as an example. I decided to use that model as my benchmark, as it was the simplest colorization autoencoder I managed to find on the Internet. I believe it would work as a good benchmark, analogically to evaluating the performance of advanced classifiers as compared to basic ones, such as the Logistic Regression or a Decision Tree.

I compare it to the more advanced architectures by:

- comparing the MSE on the validation set
- visually inspecting the colorization done by the networks

The image below presents the network's architecture:

```
(color): Sequential(
  (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): ReLU()
  (2): Conv2d(8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU()
  (4): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): ReLU()
  (6): Conv2d(16, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (7): ReLU()
  (8): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (9): ReLU()
  (10): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (11): ReLU()
  (12): Upsample()
  (13): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (14): ReLU()
  (15): Upsample()
  (16): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU()
  (18): Upsample()
  (19): Conv2d(16, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): Tanh()
)
```

In the network, I do not use pooling layers for reducing the dimensionality of the input. Instead, I use a stride of 2 in some of the convolutional layers. The upsampling is done using a non-learnable interpolation. In this case, this is an upsampling by a factor of 2 (effectively doubling the size of the image) and uses the nearest neighbor interpolation algorithm.

Model V1

This model is based on the "beta" architecture presented in [1]. Before settling for this architecture, I tried recreating the "beta" one verbatim, however, the model was not learning at all. I suspect that the latent representation was simply too big and the model was not learning any useful features. That is why I effectively scaled it down, in terms of the number of channels, by a factor of 2. The following image presents the details of the architecture:

```
(color): Sequential(
  (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (3): ReLU()
  (4): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): ReLU()
  (6): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (7): ReLU()
  (8): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (9): ReLU()
  (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (11): ReLU()
  (12): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU()
  (14): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU()
  (16): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU()
  (18): Upsample()
  (19): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU()
  (21): Upsample()
  (22): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (23): ReLU()
  (24): Conv2d(16, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): Tanh()
  (26): Upsample()
)
```

I will not describe the details of that network, as they are analogous to the ones in the V0 model.

Model V2

The last model is a slight modification of the model presented in [3]. The author combined the first few layers of ResNet-18 as an encoder extracting the features from a grayscale image (the input of the network also had to be modified to accept 1 channeled images). For brevity, I do not include the detailed specification, as the image would be quite big.

Then, the author used a decoder similar to what we already saw in [1] and [2]. What is different is the addition of the Batch Normalization layers after the convolutions.

```
(upsample): Sequential(
  (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): Upsample()
  (4): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): ReLU()
  (7): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (9): ReLU()
  (10): Upsample()
  (11): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (12): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (13): Tanh()
  (14): Conv2d(32, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): Upsample()
)
```

3. Methodology

3.1 Data Preprocessing

I have already described the data gathering process in *2.1 Data Exploration*. In this part, I focus on the transformations applied to the data and what the inputs and outputs are. The downloaded data is encoded as RGB (Red, Green, Blue) images. So each image can be considered an [3, 288, 320] array, where numbers on the scale of 0-255 describe the intensity of a given channel. If I left it like it was, the neural network would accept a grayscale image of size [288, 320] and return the three RGB channels, so again an [3, 288, 320] array.

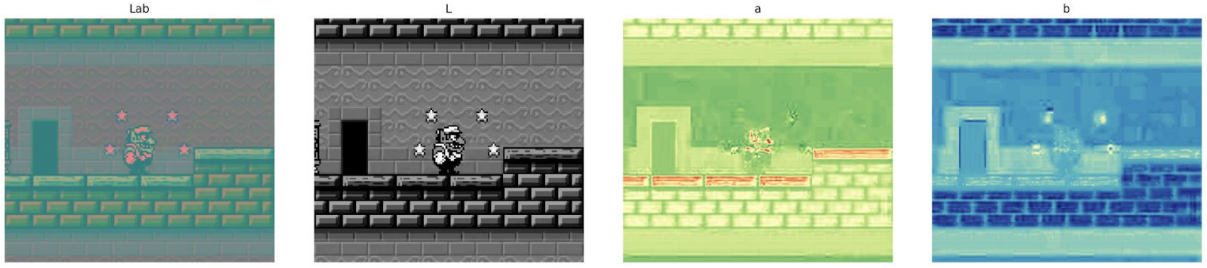
However, there is a way to reduce the complexity of the problem at hand. To do so, we can use another color space, namely the *Lab* color space (also known as CIELAB).

The *Lab* color space encodes colors as three channels:

- *L*: the lightness on a 0-100 scale (from black to white), which as a matter of fact is the grayscale image
- *a*: a layer representing the green-red color spectrum (values in [-128, 127])
- *b*: a layer representing the blue-yellow color spectrum (values in the same range as *a*)

What is handy about using the *Lab* color space is that the color is expressed only using two layers, so I can reduce the output size to [2, 288, 320].

In the following image, I present the *Lab* representation of the image used before:



Having discussed the *Lab* color space, I move on to the two steps of preparing the data.

The Training/Validation Split
For this project, I do a 90-10 training-validation split. The validation data will not be used for training, rather for evaluating the network's performance after each epoch.

Pre-processing Steps
All the pre-processing steps are done within a PyTorch DataLoader, to make the process as simple and efficient as possible. The steps I take are:

- Cropping the images to 224x224 - in one of the architectures I would like to follow, the first few layers of ResNet-16 are used as the encoder. To keep the changes to a bare minimum, I crop all the images to 224x224 (as this was the size of the images used in that architecture). However, the input of the network must be changed to accept a one-channel image. For training, I take a random crop of the images. For validation, a center crop, which guarantees that the validation sample is always the same.
- Additionally, the training images have a 50% probability of being horizontally flipped.
- I encode the RGB input image into two images - the grayscale input of size 224x224 and the *a/b* layers of the *Lab* color space as the output (shape [2, 224, 224])
- After encoding the images as *Lab*, I normalize the layers, similarly to what was done in [1] and [2]. I divide the *L* channel by 100 so the values are in the range of [0,1]. Then, I divide the values in the a/b channels by 128, and the results are in the range of [-1, 1], which is appropriate for the *tanh* activation function (last layer). I also experimented with a different transformation, normalizing the *a/b* layers to the range of [0,1] by first adding 128 and then dividing by 255. In such a case, a *ReLU* or *Sigmoid* activation function would be appropriate for the last layer. However, empirically the first approach worked better.

All the data-splitting and pre-processing in the abovementioned steps is done with prespecified seeds, to ensure the project's reproducibility.

3.2 Implementation

I implemented all the models in PyTorch. On the way, I also experimented with some of the models in Tensorflow/Keras, as many of the models I was basing my project on were actually coded in Keras.

The workflow could be summarized in the following steps:

1. Experimentation within Jupyter Notebooks
2. Modularization of the code - exporting working functions/classes to separate utility scripts, which can be imported to the Notebooks/other scripts

3. Preparing the scripts required to train the Neural Networks as *training jobs* in AWS
4. Training the models using GPU and evaluating the performance from a dedicated Notebook

Some issues I encountered during the project concerned porting Keras code to PyTorch. Keras does a lot automatically for the user, while that does not happen in PyTorch. That is why, for example, I had to calculate the values of the padding parameter in order to match the “same” padding from Keras.

3.3 Refinement

I started the project by finding and implementing a few colorizing autoencoders *as is*, however, I later had to create one framework to which I would stick. The reason for that is that I effectively combined a few different approaches, each one of them used a slightly different way of pre-processing the images, applying transformations, etc.

To make the results comparable, I settled for one variant of the framework and trained all the models in a similar fashion.

Some of the refinements I introduced to the models I worked with:

- implemented and trained all the networks using a unified approach - same loss (MSE), optimizer (RMSprop), learning rate (0.001), the maximum number of epochs (30), etc.
- decided to use an additional normalization step on top of the *Lab* conversion, that is why I had to modify some final activation functions in the networks to *tanh*
- used a set of transformations (random cropping, horizontal flipping, center cropping) to augment the dataset. Some of the examples I based the project on did not include any transformations at all.

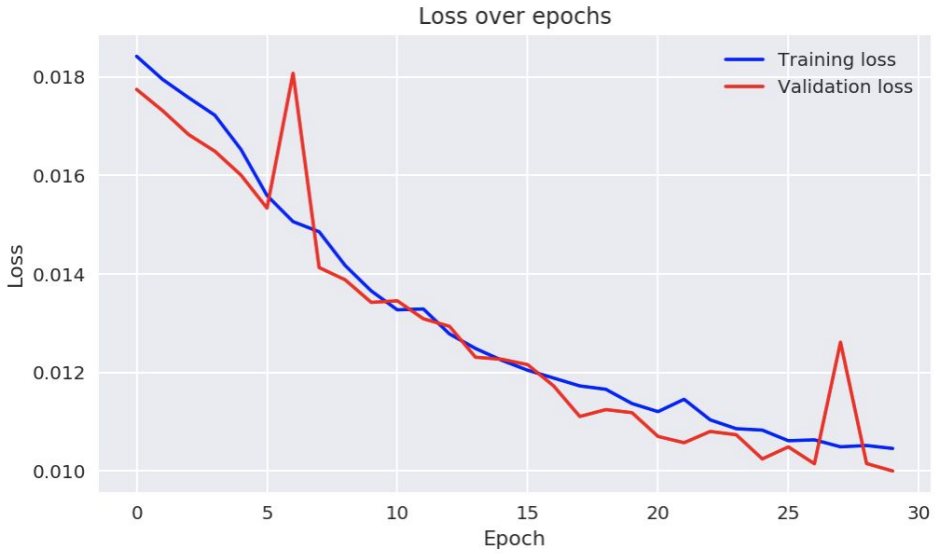
I tried to make the code as flexible as possible for easy experimentation. That is why some of the refinements such as the normalization of the *Lab* images are parametrized for convenience.

4. Results

4.1 Model Evaluation and Validation

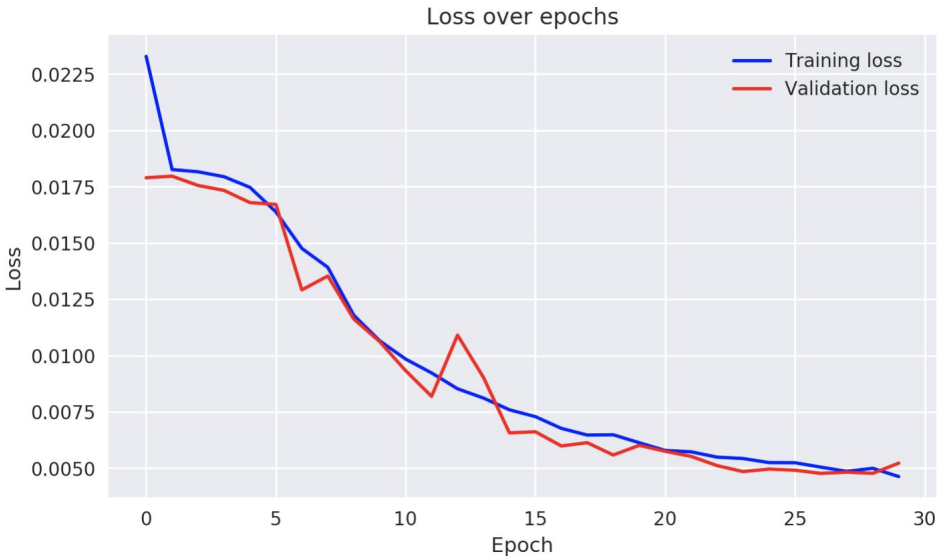
To evaluate the performance of the colorization autoencoders, I start by inspecting the evolution of the training/validation losses over the epochs, for each of the 3 models separately.

- **Model V0**



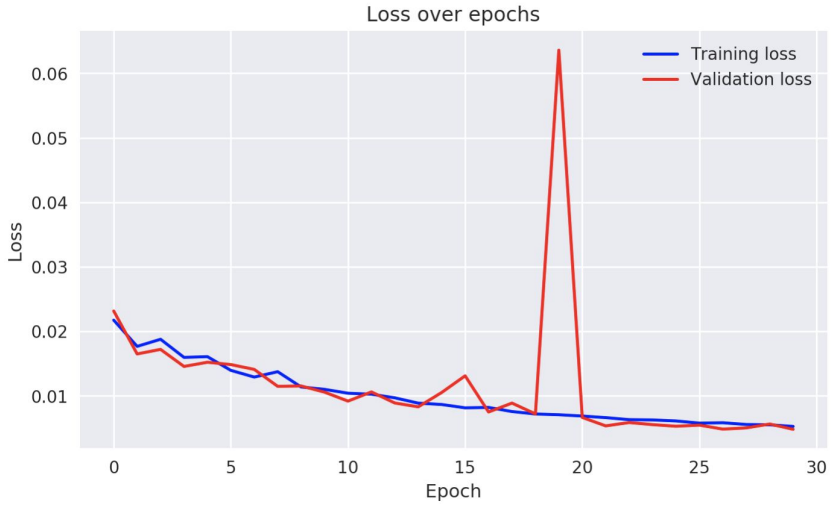
In the image we can see that the loss started to plateau, however, it would probably still decrease if trained for more epochs. Also, there are two weird spikes in the validation loss, which is something the author of [1] also noticed when using the ADAM optimizer. For him using RMSprop solved the issue.

- **Model V1**



The models started at a higher loss than the V0 model, probably due to the much larger number of learnable parameters. However, in the end, it achieved a better performance within the 30 epochs. Also, we can observe one smaller spike in the validation loss over the training time.

- **Model V2**



For the most complex model (also in terms of the number of learnable parameters), the only thing that stands out in the plot above is the huge spike in the validation loss. I do not pay much attention to it as it only happened once and the rest of the training was stable.

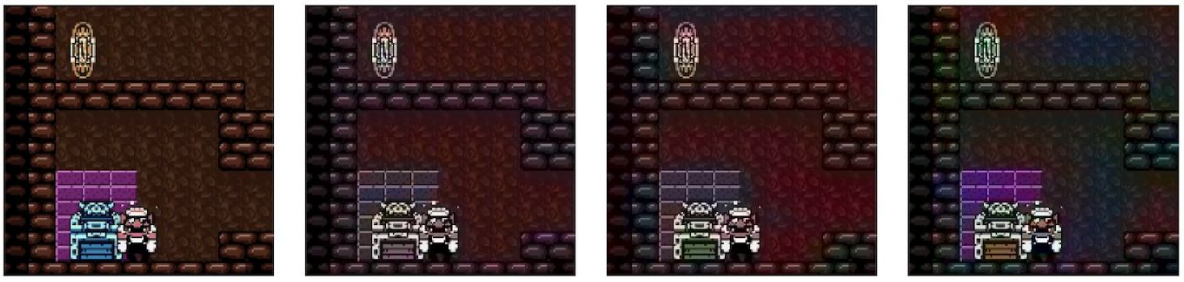
As the next step, I compare the validation set MSE of all the models at the 15th, 30th epochs, and the best epoch (in case it was not the last one).

Model / Epoch	15	30	best
V0	0,01227	0,00999	0,00999
V1	0,00657	0,00523	0,00477
V2	0,01052	0,00481	0,00481

Overall, the V1 model scored the lowest validation loss among the considered models. Also, both V1 and V2 models outperformed the benchmark, which makes sense given its simple architecture. I can conclude that I achieved the goal of outperforming the benchmark model.

Lastly, I present the results for visual inspection. The images come from the validation set (were not seen during the training) and are obtained from the respective models' best epoch.

RGB V0 V1 V2



The V2 model manages to capture the color of the blocks around the treasure chest, however, incorrectly colored the chest itself.

RGB V0 V1 V2

The V1 model did the best job in coloring the map, the V2 image is a bit unstable with various bright spots in the image.

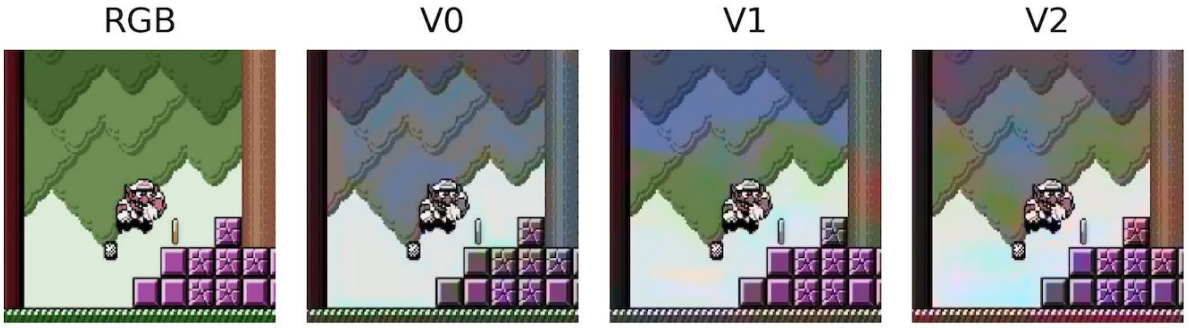
RGB V0 V1 V2

The V1 model did a good job of capturing the background, while the V2 model is struggling to decide which color to use.

RGB V0 V1 V2

All the models managed to capture the purple-ish broken blocks.

The last one was tricky, as the fragment of the map was actually close to grayscale. Model V2 tried to colorize the map, while V1 mostly kept it intact and correctly tried to colorize the frame only.



V1 and V2 are quite close to capturing the green color of the forest while remembering that the broken blocks are purple. The models also correctly color the trunk of the tree as brown.

Lastly, the models tried to colorize the image above, but the results are not so “perfect”. A potential reason is that it is a kind of a one-in-the-game kind of image, so it was hard to use any latent features to derive the correct colors.

Summing up, the models are doing quite a decent job of colorizing the grayscale images. The neural networks seem to have picked up some patterns such as purple broken blocks, purple tiles around a treasure chest, yellowish coins, greenish forest, etc. Also, the more advanced models (V1/V2) clearly outperform the benchmark. However, the V2 model is quite unstable and often results in bright and mismatched colors.

In the GitHub repo, I also tried to apply the models trained on *Wario Land 3* to images coming from *Wario Land 2*, however, the performance was rather poor. The reason for that might be a bit different style of the game and the fact that the developers were not reusing a lot of sprites/models.

4.3 Potential extensions

By no means is the project complete and exhaustive. Therefore, I list some ideas for further expansion:

- Apply data cleaning, for example, filter out images with a high percentage of white/black pixels. This might indicate that these are some screen transition screens and coloring them is not that important and might introduce unnecessary noise to the model.
- The project could be expanded to capture more than 1 video game, potentially from the same genre (for example, platformer), to increase the performance and cover for potential overfitting.
- Use trainable upscaling layers (Transposed Convolution Layers) instead of the non-trainable one.
- Use transfer learning from an already pre-trained network.

5. References

[1] https://blog.floydhub.com/colorizing-b-w-photos-with-neural-networks/
[2] Baldassarre, F., Morín, D. G., & Rodés-Guirao, L. (2017). Deep koalarization: Image colorization using cnns and inception-resnet-v2. arXiv preprint arXiv:1712.03400.
[3] https://lukemelas.github.io/image-colorization.html
[4]https://www.theverge.com/2019/4/18/18311287/ai-upscaling-algorithms-video-games-mods-modding-esrgan-gigapixel
[5]Hwang, J., & Zhou, Y. (2016). Image colorization with deep convolutional neural networks. In Stanford University, Tech. Rep.