Czech Technical University in Prague
Faculty of Electrical Engineering

Diploma Thesis

Colorization of black-and-white images using deep neural networks

David Futschik

Supervisor:
doc. Ing. Daniel Sýkora, Ph.D.

January 2018

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

DIPLOMA THESIS AGREEMENT

Student: Futschik David

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: Colorization of black-and-white images using deep neural networks

Guidelines:

Study state-of-the-art algorithms for colorization of black-and-white images based on the usage of deep neural networks [1-5]. Compare these method and evaluate visual quality they produce. Focus mainly on the method Colorful Image Colorization [5], implement it, train it, and verify its functionality on a data set of black-and-white photos as well as hand-drawn images which will be supplied by the thesis supervisor. Compare results of [5] with concurrent colorization techniques which do not utilize neural networks [6, 7].

Bibliography/Sources:

[1] Cheng et al.: Deep Colorization, Proceedings of IEEE International Conference on Computer Vision, pp. 415-423, 2015.

[2] Deshpande et al.: Learning Large-Scale Automatic Image Colorization. Proceedings of International Conference on Computer Vision, pp. 567-575, 2015.

[3] Iizuka et al.: Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous, ACM Transactions on Graphics 35(4):110, 2016.

[4] Larsson et al.: Learning Representations for Automatic Colorization, Proceedings of European Conference on Computer Vision, pp. 577-593, 2016.

[5] Zhang et al.: Colorful Image Colorization, Proceedings of European Conference on Computer Vision, pp. 649-666, 2016.

[6] Levin et al.: Colorization Using Optimization, ACM Transactions on Graphics 23(3):689-694 , 2004.

[7] Sýkora et al.: Colorization of Black-and-White Cartoons, Image and Vision Computing 23(9):767-852, 2005.

Diploma Thesis Supervisor: doc. Ing. Daniel Sýkora, Ph.D.

Valid until the end of the summer semester of academic year 2017/2018

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-

L.S.

prof. Dr. Michal Pěchouček, MSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 20, 2017

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Declaration

I hereby declare that I created the presented thesis independently and that I cited all used sources of information in accord with Methodical instructions about ethical principles for writing academic theses.

V Praze dne ....

Podpis autora práce

Acknowledgements

I would like to thank my supervisor, doc. Ing. Daniel Sýkora, Ph.D., for initial guidance, suggesting the used dataset and allowing me to work on this thesis. Furthermore, my family and friends deserve a big thanks for all the support that I received during my studies and during the work on this thesis.

Abstrakt

Automatické obarvovaní šedotónových obrázků se v posledních letech stalo více zkoumanou oblastí, zejména díky rozšířenému používání hlubokých konvolučních neuronových sítí. Cílem této práce je pokusit se aplikovat tuto metodu na automatické obarvování snímků kresleného seriálu. Předchozí výzkum v této oblasti se zaměřoval především na obarvovaní přirozených obrázků a fotografií. Kreslené seriály se tradičně obarvují metodami, které vyžadují lidskou asistenci. V této práci je navrženo plně automatické řešení, k dosažení kterého používáme dvě různé architektury konvoluční sítě trénované za použití různých chybových funkcí. Trénované varianty porovnáváme na základě získaných výsledků jako jednotlivé obrázky i videosekvence.

Abstract

Colorization of grayscale images has become a more researched area in the recent years, thanks to the advent of deep convolutional neural networks. We attempt to apply this concept to colorization of cartoon images obtained from video sequences. Previous similar research focused mainly colorization of natural images, while colorization of cartoons is traditionally done by leveraging manual scribble methods. Our proposed method is a fully automated process. To implement it, we propose and compare two distinct convolutional neural network architectures trained under various loss functions. We aim to compare each variant based on results obtained as individual images and videos.

Contents

Chapter 1

Introduction

In this thesis, we concern ourselves with the possibility of automated grayscale image colorization using deep convultional neural networks, trained on and applied to a unique cartoon dataset.

Generally, the idea of coloring a grayscale image is a task that is simple for the human mind, we learn from an early age to fill in missing colors in coloring books, by remembering that grass is green, the sky is blue with white clouds or that an apple can be red or green.

In some domains, automatic colorization can be very useful even without semantic understanding of the image, the simple act of adding color can increase the amount of information that we gather from an image. For example, it is commonly used in medical imaging to improve visual quality when viewed by human eye. Majority of equipment used for medical imaging captures grayscale images, and these images may contain parts that are difficult to interpret, due to inability of the eye of an average person to distinguish more than a few hues of gray [1].

As a computer vision task, several user-assisted methods have been proposed for colorization, be it for natural or hand-drawn images, and expect supplied localized color hints, or provided reference images that are semantically similar to the target image that we can transfer color from, or even just keywords describing the image, to search the web for the reference images automatically.

However, the high-level understanding of scene composition and object relations required for colorization of more complex images remains the reason developing new, fully automated solutions is problematic.

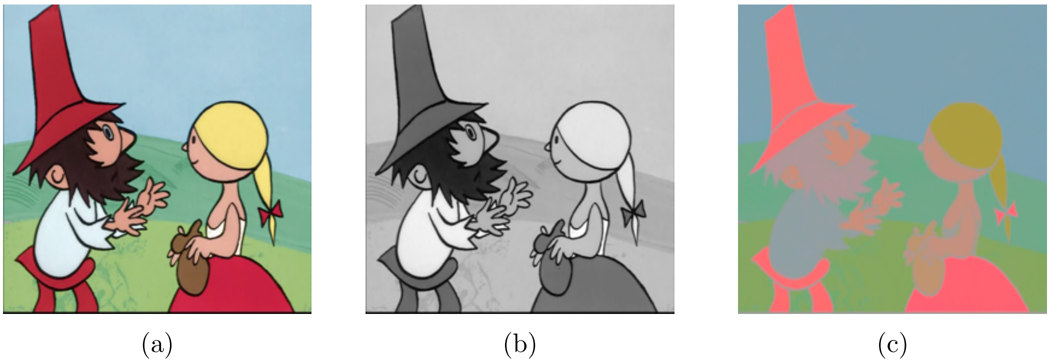
Recently, the approach to this task has shifted significantly from the human-assisted methods to fully automated solutions, implemented predominantly by convolutional neural networks. Research in this area seeks to make automated colorization cheaper and less time consuming, and by that, allowing its application on larger scale.

With the advent of deep convolutional neural networks, the task has been getting increased amounts of attention as a representative issue for complete visual understanding of artificial intelligence, similar to what many thought object recognition to be previously, since to truly convincingly colorize a target image, the method needs to be able to correctly solve a number of subtasks similar to segmentation, classification and localization.

1.1 Problem statement

Consider a color image I - we can decompose such image into two essential components, **luminance** and **chrominance**. With this knowledge, we can express the original image as $I = (I_l, I_c)$. The image can be fully reconstructed given the two components, but having either of the components alone is incomplete. Luminance images can still be viewed and appear naturally, since this component conveys most of the visual information, such as object edges and lighting effects.

Example of such decomposition can be observed in Figure 1.1, notice that the luminance component is what is commonly known as a grayscale image, and in some contexts, the two concepts overlap.



(a) (b) (c)

Figure 1.1: Color image (a) decomposed into (b) luminance and (c) chrominance in the Lab color space.

The problem this thesis attempts to explore is one where the grayscale I_l is available, but I_c is not known and must be derived automatically. For each pixel of the grayscale *target* image, we look for a color value to assign to this pixel. This estimation is commonly performed in the Lab color space, as it attempts to maximize the decorrelation of luminance and chrominance channels. In other words, we seek to learn a mapping

$$\hat{P} = \mathcal{F}(P) \tag{1.1}$$

from the pixels $P \in \mathbb{R}^{H \times W}$ of the grayscale image to the associated color channels $\hat{P} \in \mathbb{R}^{H \times W \times 2}$.

Notice that this problem is necessarily multimodal in its nature, as there are multiple *plausible* colorizations of many objects. We state that there is no one correct solution - in fact, almost any object can take on a wide range of colors. However, the colorization can be dependent on the context hidden within the scene which the object is a part of.

As such, even comparing two resulting colorizations can be difficult, since there are many methods to defining an image distance function, and traditional metrics penalize plausibly but incorrectly colorized images. For instance, when comparing an image of a car that is colored blue, but appears red in the ground truth image, most metrics will calculate high error rate. Thus, colorization results require subjective inspection, as deciding whether an image is plausibly colored via automated means is a problem nearly as difficult as colorization itself.

1.2 Thesis goals

In this thesis, we investigate colorization on a specific subset of images extracted from a cartoon movie. It is common for cartoons to be drawn without colors, but the presence of color greatly increases the visual appeal of the movie. Traditionally, the colors are added later on in the creative process by scribble based methods, which require considerable effort on the part of the user. Automatic colorization would lead to increase in the creation pipeline throughput, saving artists the time spent on colorization while requiring limited amounts of intervention.

The main goal of the thesis is to determine whether it is possible to use convolutional neural networks for fully automated colorization of black-and-white (grayscale) cartoon images plausibly, as standalone images and when put into video sequences. These images fundamentally differ from natural images by containing fewer textures, making the task of inferring semantic information harder.

To achieve this, we propose several variants of convolutional neural networks and compare their performance on the data, using two distinct neural network architectures; one more traditional, plain convolutional network, and one inspired by residual convolutional neural networks, which has not been used for colorization previously.

1.3 Outline

First, in Chapter 2, we provide an introduction into the domain of convolutional neural networks for the reader, focused on the concepts important for this thesis, along with brief historical context. Then, in Chapter 3, we present a comprehensive overview of work related to the task of colorization, including manual scribble methods, automated color transferring methods and research using convolutional neural networks. In Chapter 4, we describe the used dataset, give reasons for choosing the Rumcajs dataset and provide details about its extraction, composition and comparison to natural image datasets.

The method of our colorization is described in detail in Chapter 5. In Chapter 6, we define the two particular neural network architectures used in this thesis, complete with particular layers and hyperparameters and in Chapter 7 we provide details about all the trained variants and the process of their training. Chapter 8 contains overview and discussion of results, where we present side-by-side comparisons for all trained variants and compare our method to current automatic color transfer methods. In Chapter 9 the work is summarized, and we express thoughts about possible future extensions to it. Appendix A contains detailed description of supplementary material provided on the attached CD and Appendix B contains additional image results generated by our networks.

Chapter 2

Deep Convultional Neural Networks

This chapter explores previous work related to deep convolutional neural networks. We first describe brief history of the concept, mentioning the most important milestones and work relating to this thesis. Then, we outline and explain basic concepts surrounding convolutional neural networks we used for the task.

Convultional neural networks are a class of neural networks which contain convolutional layers. They are most commonly used in image related tasks, and though their use is not strictly limited to the image domain, it is commonly expected that they will operate on 2 or 3 dimensional data.

2.1 History

Since 2012, convolutional neural networks (often shortened to ConvNets, **CNNs**, dCNNs) have taken the world by storm, despite not being entirely new technology. In fact, the first documented commercial use of CNNs dates back to 1998, with LeNet-5. Designed by LeCun et al. [2] after spending years researching CNNs, it was used for text character recognition based on 32x32 pixel images. This area of computer vision has been practically dominated by CNNs since then, surpassing results obtainable by other machine learning methods.

However, large scale application of CNNs would not be possible until at least a decade later, just as computational power needed to train efficiently became more accessible and memory less restrictive, researchers started using CNNs more widely and found that CNNs could be leveraged even when applied to larger image datasets.

The CNN boom is famously attributed to 2012 ImageNet Large Scale Visual Recognition Challenge entry of Alex Krizhevsky et al. [3] and their AlexNet. When this CNN based model won that year's image classification challenge by a significant margin, it took most of the computer vision research community by surprise and sparked a large amount of interest.

Following this event, CNNs have quickly started becoming a staple in the computer vision field and many others. They have been successfully applied to countless problems, such as image recognition, video sequence tracking, automatic image segmentation, facial recognition, handwriting to text conversion [4], natural language processing or automated translations [5].

CNNs have proven to be models that are able to learn very complex mappings of inputs to outputs from large amounts of data, functioning as automatic data encoders. In the case of image recognition (classification) challenges, CNN models have shown dominance ever since 2012, pushing the state of the art every year in the process.

Year 2014 brought the 16 layers deep VGG-16 and the 22 layers deep GoogLeNet models, both of which surpass the projected human error in the image classification task on the ImageNet dataset (with later refinements). They are still frequently used today, especially as starting points to use for transfer learning or for building classifiers, using them as feature extractors.

A year later, in 2015, Microsoft Research Asia introduced an alternative architectural approach to traditional (plain) convolutional networks, called residual networks (shortened to ResNets), achieving state-of-the-art accuracy on ImageNet classification. This architecture allowed the team to significantly increase the depth of their networks, the best performing model consisted of 152 layers [6].

2.2 Basic concepts

Here, several basic concepts are introduced. This section is meant to serve as a reference to how these terms are used in this thesis or as a brief introduction into ideas behind CNNs.

2.2.1 Layer

CNNs are composed of smaller building blocks called layers. Each layer acts as a single step in the overall input to output transformation. These layers can take on many forms and may serve a multitude of purposes. Traditionally, a layer takes an input, such as an array representing an image, and produces an output, realizing a function of the form:

$$y = \sigma(Wx + b), \tag{2.1}$$

where $x \in R^n$ and $y \in R^m$ are the inputs and outputs of the layer, respectively, $W \in R^{m \times n}$ are the weights associated with the layer, $b \in R^m$ is the bias vector, which serves as an additive component to the transformation, and $\sigma : R \to R$ is a non-linear per-input *activation* function.

A layer can optionally take more than one input and produce more than one output, which can be represented as a concatenation of the inputs or outputs to produce a single input or output.

Weight matrix and bias vector are also called the *parameters* of the layer, and are learned through back-propagation of loss calculated as the error between the network's prediction and training data ground truth label[7].

The activation function provides the network with the ability to learn non-linear mappings, as the weight and bias components are only able to produce linear transformations. The most common examples of activation functions include the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

a hyperbolic function, such as hyperbolic tangent:

$$\sigma(x) = \tanh(x) \tag{2.3}$$

and in the case of CNNs especially, the Rectified Linear Unit (ReLU) function:

$$\sigma(x) = \max(0, x) \tag{2.4}$$

Each of these functions will result in non-linearity being introduced into the network. Note that it is also possible to represent the activation function as a separate layer with identity matrix W and a zero vector b, and that this is a common practice in many CNN frameworks.

2.2.2 Convolutional layer

Convolutional (conv) layers are the basic layers used in CNNs (hence the name). They represent the convolution operation performed on input with weights as parameters of the convolutional kernel, in the 2 dimensional case:

$$y_{i,j} = \sum_{m=i-\lfloor\frac{k}{2}\rfloor}^{i+\lfloor\frac{k}{2}\rfloor} \sum_{n=j-\lfloor\frac{k}{2}\rfloor}^{j+\lfloor\frac{k}{2}\rfloor} x_{m,n} W_{m-i+\lfloor\frac{k}{2}\rfloor,n-j+\lfloor\frac{k}{2}\rfloor} + b_{m-i+\lfloor\frac{k}{2}\rfloor,n-j+\lfloor\frac{k}{2}\rfloor} \tag{2.5}$$

where k is the size of the *convolution kernel* (also called filter), assuming it is odd (computation is ambiguous for even sizes and is therefore implementation dependent) and W and b simply run from the first index to the last. As we can see, the weights are stored per kernel and not per input, meaning that the layer does not need to have a fixed size input, acting as a sliding window. The weights are spatially identical for all parts of the input, which makes convolutional layers shift invariant.

The two dimensions are often called width and height, just like in an image.

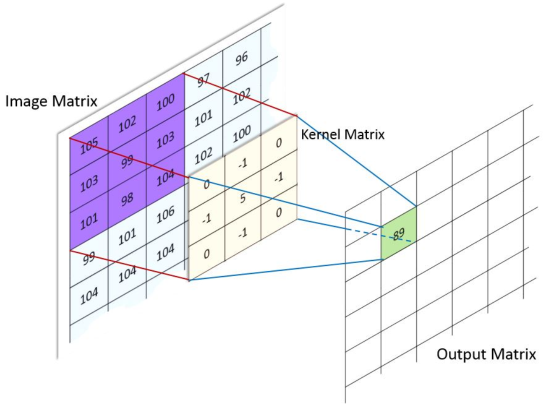


Figure 2.1: Visual explanation of convolutions

As we can see from Figure 2.1, a convolution's main hyperparameter is the **size of its kernel**. This size is usually given as a single number, or writen as a multiplication, e.g. 3 × 3 for a kernel of size 3. In theory, the size can be non-square, however, it is not practical for the purposes of this thesis to consider such cases.

As far as the output is concerned, there are several hyperparameters traditionally associated with a convolution layer. First, a layer generally does not consist of a single kernel, but several kernel filters are learned instead, and their outputs spatially concatenated (stacked) into a 3 dimensional volume. This hyperparameter is called **number of outputs**. If connected to another convolutional layer, the latter layer then treats the concatenated dimension as a fixed size dimension (also called depth) and learns its filters along the full length of depth. For example, if a convolution layer calculates its outputs on a 256 × 256 image and contains 64 filters, the output will be of shape 256 × 256 × 64. Should another convolutional layer "sit on top of" this one and have kernel size of 3, its convolution kernel filters will be of shape 3 × 3 × 64.

The convolution calculation does not need to be perfomed for every point of input, for example, we can skip every *nth* input point in either direction. This concept is called ***stride*** - in this example we are computing the output with stride of $n+1$. A convolution without any striding applied is said to have a stride of 1. Stride is an important concept, as it essentially allows us to downsample input resolution through convulution quickly.

Upsampling on the other hand can be implemented by transposed convolution, where the input matrix is spread out by zero padding, producing output resolution higher than the input resolution (can be thought of as having stride lower than 1).

We can also notice that convolutions cannot be calculated around the edges of the input, as there are no values to calculate with. As a result, the output will thus be $\lfloor \frac{k}{2} \rfloor$ smaller along each edge, for example, given a 256×256 input, performing a convultion with kernel size of 7 will result in an output size of 250×250. To counteract this effect, ***padding*** is added to the input, such that we can calculate the output even for edge points. This padding can be simply zero filled, as is most commonly used, or we can wrap the inputs around the other side.

The final formula for calculating the output size is:

$$w_O = \frac{w_I - k + 2p}{s} + 1, \tag{2.6}$$

where w_O is the width of the output, w_I is width of the input, k is the kernel filter size, p is the size of edge padding and s stands for stride. The calculation is analogous for height.

Initially, CNN models used wide ranges of k values, though generally not greater than 11×11. For example a common choice for low level feature layers was $k = 7$ as proposed in [3]. In recent years, filters of size 3×3 seem to be dominating in most models, some models choose to use 3×3 kernels exclusively [8]. One notable exception here is the GoogLeNet model, which builds on the idea of using filters of multiple sizes on the same input data and concatenating the results, which essentially lets the training process learn which values of k produce the most useful information [9].

A network in which many convultional layers are stacked on top of each other is called a **Deep convolutional neural network**, or a deep CNN, though it is not a precise term; there is no definition of how many layers need to be present in order for a network to be called deep.

Dilated convolution

More complex schemes of convolution have be devised, such as the recently introduced concept of dilated convolutions (also known as "atrous" convolution). Instead of using contiguous neighboring inputs to calculate the output value of a convolution operation, we insert gaps of size $d \in \mathbb{N}_0$ between the inputs. This hyperparameter is called the ***dilation*** of convolution. Normal convolutions have dilation of 0 (some sources define regular convolution dilation to be 1). In Figure 2.2 we can see a visual explanation of the operation. Dilation allows convolution layers to have *larger effective receptive field* while retaining a lower amount of parameters. Using this technique has been shown to help alleviate overfitting as well as improve performance [10].

Effective receptive field

A single convolutional layer with a kernel of size k can only encode relations between its inputs in $k \times k$ patches. The inputs are connected in a local region, in case of a kernel size

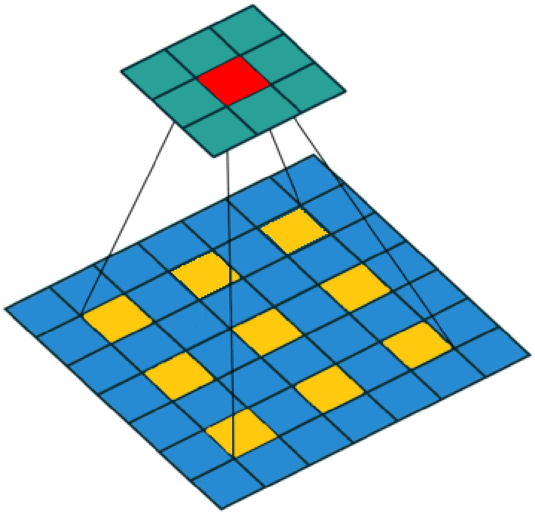


Figure 2.2: Visual explanation of dilated convolutions, yellow cells are inputs to be convolved with the kernel to produce the value of the red output cell

of 3 on an image, we can only hope to represent connections between pixels not further apart than 2 pixels. As this would be rather limiting, convolutional layers are put in a sequence.

When we stack convultional layers on top of one another, the effective reach of connections increases, in the example of 3×3 kernel, two stacked layers would yield a function of a 5×5 patch of the original input as the final output, since the second set of convolutions operates on inputs that are the outputs of the previous convolution. This concept is called the ***effective receptive field*** of the network and can be viewed as the size of the field of initial inputs that can influence a final output. Some sources also refer to this concept as field-of-view.

It is important to realize that not every input, even if it falls into the receptive field of an output, will be able to influence the output in the same way, as shown by Luo et al. [11]. Intuitively, the input units (pixels or voxels) spatially closer to the output units can influence it more significantly, as they are used in more convolution calculations when computing the output's value. When the effective receptive field is mentioned in regards to a network, it is understood as the size of the field of an arbitrary output in relation to the network's inputs.

Effective receptive field is a critical concept when considering CNN architectures, as it provides a way of reasoning about individual outputs.

It follows naturally that the receptive field of a CNN can be increased by employing different methods. Every method tends to have advantages and disadvantages, some more severe than others. The simplest means of increasing the receptive field is to stack more layers. This comes with the huge downside of significantly impacting training times and increasing memory requirements (though this is framework dependent), as well as, generally, increasing the number of epochs needed for training to converge. However, on the flip side, this method increases the number of spatially independent parameters in the network as a side-effect, which can be beneficial to increasing accuracy if overfitting is accounted for.

Another technique is using subsampling, either through increasing the stride of some convolutional layers or by pooling. Through reducing the size of one layer's output, we are effectively "packing" or "compressing" those pixels into a smaller spatial area, which means that successive layer's receptive field will increase to a larger patch of the original input. Subsampling has the advantage of spatially decreasing the size of inputs, which translates into improved training times, and can help with overfitting, as it is more likely to force the network to generalize, even though it does not necessarily reduce the parameter space. On the other hand, subsampling can make certain features harder to extract due to lowered resolution - typically subtle details in input images and smaller objects become lost in the process.

A relatively recent and popular method of enlarging the receptive fiend is using dilated convolutions. Most commonly the dilation is set to a small number, as the beneficial effect seems to diminish quickly with increased dilation. The biggest advantage to using dilated convolutions is the ability to expand the receptive field without losing resolution, unlike increasing stride or pooling. Dilated convolutional layers allow us to quickly grow the effective receptive field, much more aggressively than non-dilated convolutions.

Note that the receptive field of a single layer is equal to the size of its filters, as that is the extent of the inputs that contribute to individual output's value.

2.2.3 Pooling layer

Pooling layers represent one method of resolution reduction in CNNs, which serves as a way of increasing the effective receptive field of the network. Their functionality can be intuitively described as applying a function to a local field, producing a single value. Traditionally, pooling layers have predefined filters and no trainable parameters. Common examples include the maxpooling layer, which calculates its value as:

$$y_{i,j} = \max_{m=ik}^{ik+k-1} \max_{n=jk}^{jk-1} x_{m,n} \tag{2.7}$$

Maxpooling layer is taken as the default pooling layer and in most contexts it is referred to simply as the pooling layer.

Notice that the pooling is very "destructive" toward its input data, plainly dropping information in the process of producing output. In the image classification tasks, this is not considered harmful, because the final output of the network is not directly proportional to the size of the input (sans receptive field size). In fact, using pooling layers often helps prevent overfitting. However, in tasks like colorization or segmentation, using pooling layers can result in negative influence on final accuracy, as the resolution of the output is directly proportional to the size of input.

2.2.4 Element-wise layer

Element-wise layers are basic operations which take n inputs of the same shape and perform a function in an element-wise fashion. These functions are most commonly multiplications or additions, but can also include max or min function. The output for 2 dimensional case is calculated as:

$$y_{i,j} = f(x_{1_{i,j}}, x_{2_{i,j}}, \ldots, x_{n_{i,j}}) \tag{2.8}$$

where f stands for the function peformed.

2.2.5 Training

Collectively, parameters of all layers in the network combined are called network's parameters or network's weights. These parameters need to be learned from user provided data, and the process of acquiring the weights is called training.

Weight adjusting in CNNs is done through a process of **backpropagation**. Backpropagation can be summarized in four steps: the forward pass, the loss function, the backward pass and the weight update.

During the forward pass we take a training example x and pass it through the network's layers to obtain the prediction. Based on this prediction, we calculate the error between the prediction and the ground truth label of the training example, measured as the value of a chosen loss function. We call this value the *Loss* of the network, given example x.

Since our goal is to minimize *Loss*, we need to update the weights in all layers of the network, which is why we perform the backward pass. We calculate the discrete partial derivatives between the network's weights the loss function output as $\frac{dLoss}{dW}$, and in the weight update step, we update the weights as follows:

$$w = w_i - \eta \frac{dLoss}{dW} \tag{2.9}$$

where η is the ***learning rate*** hyperparameter and w_i stands for the current weights. This concept is applied to every layer in the backward direction of the forward pass, replacing *dLoss* for partial derivatives of the previous layer in every step. In practice, update Formula 2.9 will be more complicated, as it has been gradually improved as more research has been carried out and is influenced by the optimization algorithm's hyperparameters [12].

This implies the requirement for a known label for every point of training data, which means that CNNs are a method of supervised learning. For example, if we are training the network to predict an object's type from an image, for every image in the dataset we must know its correct classification in advance.

2.2.6 Loss function

A network's loss (cost, objective) function is defined as a function of its prediction and the ground truth. The value of this function measures the error of the prediction in relation to the predefined label and is essential to training the network.

The function can be custom-tailored to the task the network is supposed to learn, but in practice, there are several commonly used functions, such as Euclidean L_2 norm, or cross-entropy loss.

A loss function must conform to two assumptions in order to be applicable in any learning algorithm used today. First, we must be able to decompose the loss of the entire training set into an average of loss functions of individual training examples:

$$F = \frac{1}{n} \sum_x F_x \tag{2.10}$$

this assumption is necessary in order to be able to compute partial derivatives of the loss function for a single training example to use in the backpropagation algorithm and still be able to make the assumption that averaging over those partial derivatives will yield the differences of the whole data set. This data-iterative optimization is called ***stochastic gradient descent*** (SGD).

The second assumption is that it must be a function of the networks output, which is also implied by the definition.

Note that iterating over all examples too many times tends to result in overfitting - the weights learn to encode the training dataset rather than generalize well on unseen examples, especially if the dataset is small and parameter space is large.

2.2.7 Dropout layer

A technique developed to help prevent training data overfitting is including **dropout layers**. These layers do not have any parameters, but instead randomly set previous layer's activations to zero with a probability equal to their dropout ratio hyperparameter setting.

This forces the training process to incorporate a degree redundancy into the learned model, which prevents too much co-adaption. After training, dropout layers are removed to take advantage of the full predictive power of the network. Using these layers has been shown to improve performance on tasks where overfitting is a problem.

Common dropout ratios for convolutional layers range between 0.1 and 0.3. [13]

2.2.8 Hyperparameters

Hyperparameters of a network are any parameters which affect the whole network, whereas parameters of the network refer to layer weights. Individual layers may also have their hyperparameters, such as the size of convulutional kernels in the case of convolutional layers.

Some hyperparameters are closely linked to network's ***architecture***, which describes the overall layout and connectivity between layers, such as number of layers, chosen non-linearity functions or input to output size ratio. Others influence the training phase - these include optimization algorithm choice, weight initialization method, weight decay multiplier, learning rate or momentum.

Two important training hyperparameters here are the number of epochs, which regulate how many times the network will use each example to update its weights (one epoch translates into passing each example once) and the batch size, which controls the number of training examples for which weight updates are accumulated before being applied. Batching is performed to take advantage of hardware parallelism during training phase, but larger batch sizes have been shown to slow down convergence rate [14].

Tuning these hyperparameters can prove to be quite a difficult task, and may appear to be more of an art rather than exact science.

2.2.9 Normalization

When training a CNN, there are two types of normalization that are commonly performed. The first type deals with input normalization, making sure that any input that enters the network is clamped or rescaled to the same ranges. This is necessary because of the global learning rate hyperparameter; if some inputs operate on a different scale of values, it is significantly harder to train evenly.

In addition, it is beneficial to scale all inputs into a 0-1 range. There are several reasons for this, normalization leads to improved convergence speeds, as the absolute size of inputs will have to be adjusted for by tuning the learning rate of the network, and higher learning rate values can lead to weight oscillation. Another reason is purely technical - considering that weights are commonly represented as double-precision floating-point numbers, making use of the denormalized interval can improve accuracy, as it allows parameters to learn more finely.

Optionally, we can subtract the mean of the training data to center the values around 0, which has also been shown to improve convergence rates.

The second type of normalization deals with internal covariate shift. During the training of a network, the distribution of each of network's input's absolute values are likely to change as the parameters of previous layers change. This slows down the training by requiring frequent modifications of the learning rate and the effect is especially pronounced when using the ReLU activation function, as it does not naturally normalize its outputs. The mechanism that has shown to be effective at combating these "exploding" (or vanishing analogously) outputs is called Batch Normalization [15].

We can make the batch normalization step a part of the architecture itself, and perform the normalization on a per-layer basis, or between every block of layers. The normalization is normally done by subtracting the expected value and dividing by the standard deviation. These values are calculated and stored as a statistic across each batch of the training.

2.2.10 Initialization

In order to begin training the network, initial weights are required as a starting point to update from. Weight initialization has severe effects on the convergence rate and improper initialization can lead to vanishing or exploding gradients. Common initializations include zero-weight initialization, however, this turns out to be detrimental to the convergence rate (and in some cases it may cause the network to never converge), because performing forward pass can completely negate any effect that input has on the outputs.

In practice, a good initialization is using random weights, normalized for the size of input and output, as shown by Glorot and Bengio [16], called Xavier initialization:

$$W = rand(\mathcal{N}(0, \frac{2}{n_{in} + n_{out}})) \tag{2.11}$$

where n stands for the size of the layer's inputs and outputs and function $rand$ generates random samples from the normal distribution. This initialization produces numbers of small absolute value, both positive and negative, which introduces the asymmetry lacked by zero-weight initialization.

Recently, there has also been development in training-data-dependent initialization by Philipp Krähenbühl et al. [17], based on unsupervised learning algorithms. This approach has also been shown to significantly improve convergence rates and to produce initializations under which majority of units in the network train at roughly the same rate, which helps avoid vanishing or exploding gradient problems and prevents huge variances in per-parameter learning rates in optimization algorithms which support it.

2.2.11 Optimization algorithm

Optimization algorithm, also called the solver, is the algorithm used to actually update weights in accordance with Formula 2.9. Updating the weights without any modifications to the formula results in regular SGD algorithm, which has been observed to have several problems.

To deal with those, varying concepts are employed. One is that of a momentum; SGD optimization may be very slow when encountered with ravines such as those around local minima. Adding a momentum term helps keep SGD optimizing in roughly the same direction, if possible, and prevent oscillations. With the momentum term, Formula 2.9 changes to:

$$w = w_i - u_i \tag{2.12}$$

$$u_i = \gamma u_{i-1} + \eta \frac{dLoss}{dW} \tag{2.13}$$

where γ is called the momentum coefficient and u_i is the current *velocity vector*, which is of the same size as the updated weights - there is a separate parameter for each weight, allowing SGD to update individual weights using steps of varying magnitudes.[18]

Numerous other algorithms that deal with the different problems SGD suffers from have been devised, often building on top of the previous improvements.

Adam (Adaptive Moment Estimation) is undoubtedly the most popular algorithm for updating weights when training CNNs currently. Adam computes an adaptive learning rate for each parameter separately, effectively expanding the idea of a global learning rate to weights (though a global learning rate is still used as a multiplier) and greatly reducing the benefits of manual adjustments of the learning rate during the training process. In addition, Adam stores an exponentially decaying average of previous gradients, an idea very similar to momentum:

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1) \frac{dLoss}{dW} \tag{2.14}$$

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) (\frac{dLoss}{dW})^2 \tag{2.15}$$

m_i and v_i are thus moving averages that estimate the first two moments of the per-parameter gradients (mean and uncentered variance). Hyperparameters β_1, β_2 control the exponential decay rate of these averages. However, since these averages are initialized to vectors of zero, the estimates are biased towards zero, especially during initial steps. To counteract this, the estimates are bias-corrected to:

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i} \tag{2.16}$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_2^i} \tag{2.17}$$

These bias-corrected estimates are then used to normalize the gradients and multiplied by the global learning rate to update the weights [19]:

$$W_{i+1} = W_i - \frac{\eta}{\sqrt{\hat{v}_i} + \epsilon} \hat{m}_i \tag{2.18}$$

Since the Adam algorithm is the only one used in our work, we limit ourselves to describing only this optimizer.

2.2.12 Transfer learning

It is not always required to train the network from random initialization values, many newer networks use a pretrained model as its initialization instead. When training a model that deals with a similar problem as an already trained network, this can be extremely beneficial, increasing convergence rate and improving performance.

One disadvantage of doing so is that it imposes constraints on used network architecture, as the weights that are fine-tuned from are generally locked into a specific structure.

It is, however, possible to use only some layers (typically the initial ones) of a reference model and build new layers on top of them.

The already-learned layers might be frozen during training (receiving no weight updates), or have reduced learning rates (receiving damped weight updates) to preserve their properties. This technique is called transfer learning or **fine-tuning**.

Chapter 3

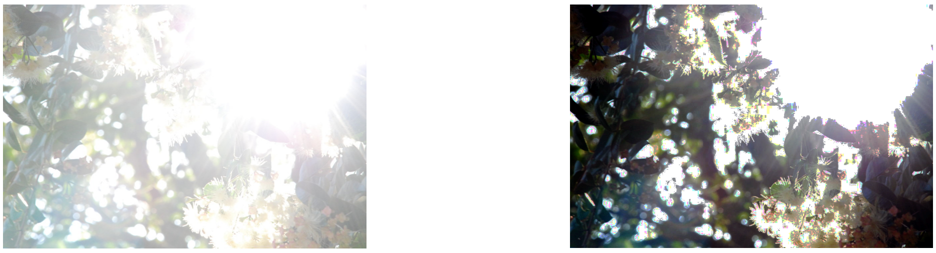
Related Work

This chapter is dedicated to creating an overview of previous work in the field of automatic colorization, with particular focus on works that have inspired, influenced and helped shape this thesis.

In general, working with color channels in images is a rather well studied and surveyed area. However, compared to other similar problems, the idea of generating some or all of the color information given other data is one that has seen relatively limited amount of widespread application and, conversely, research. However, it is quickly becoming one of the more popular image-to-image tasks in the computer vision field, with several works published in the recent years using various approaches.

Countless algorithms that are not considered colorization methods, but rather mere color enhancements which aim to improve existing poor color information or modify the color palette of an image are worth mentioning in this section, as they serve as a sort of a precursor to full colorization techniques. The goal of these methods is to take images with color data as their inputs and transform them into images with better visual properties. Often they serve to remedy certain camera defects, such as overexposure or underexposure contrast adjustment through histogram equalization[20], as shown in Figure 3.1.

The simple transformations performed by these non-parametric methods are frequently used to describe behaviors of other, more complex algorithms, even in the domain of CNNs. It is for example possible to say that one of the transformations performed by a CNN resembles histogram equalization.



(a) (b)

Figure 3.1: (a) Overexposed image. (b) Image after color enhancement using histogram normalization.

When considering full image colorization, there are generally three major types of approaches that have been used. Firstly, a non-parametric approach, in which the user provides hints to an algorithm as to what the final colorization should look like. These hints come in the form of ***scribbles*** - small patches of color in specific areas of the image.

More automated methods developed still rely on additional user input, but instead of providing direct color data, the user is expected to provide one or more reference images from which to perform ***color transfer*** onto the target image using statistical data or texture matching.

Recently, with the advent CNNs, the approach has shifted more towards a fully automated solution, where the only input provided by the user is the target grayscale image. However, this enormous advatage can also turn into a disadvantage - in case the CNN result turns out to be unsatisfactory, there are few to no options to easily remedy it trivially, unless the model has been specifically designed with this requirement in mind.

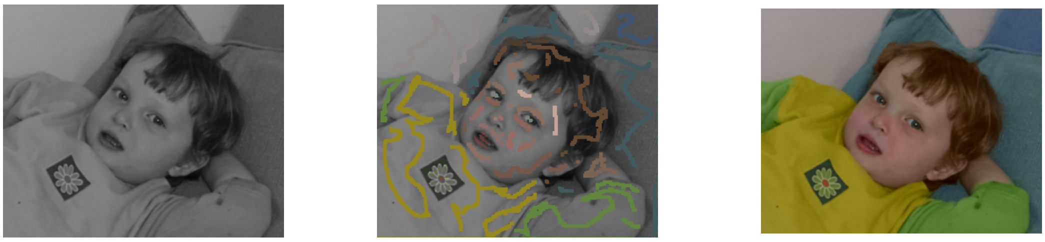
3.1 User provided color hints

Colorization methods that depend on color scribbles generally use an optimization framework without explicit parameter learning to propagate the color from the color patches onto the whole image. The scribbles are usually provided as a separate image in the form of a color-transparency mask, and the segments of the image that have no explicit color defined in this mask should have color information propagated to them. The basic assumption behind most of these methods is that nearby pixels of similar intensities should also have similar colors.

In the method proposed by Levin et al. [21], the colorization is achieved by solving a convex quadratic cost function obtained as differences of intensities between neighboring pixels. With further improvements by Huang et al. [22] to exploit edge detection in order to reduce common problems with color bleeding over object boundaries, this has become a relatively popular technique to interactively colorize natural images.

Luan et al. [23] presented a method extending the use of scribbles to texture similarity, automatically labeling pixels that should share roughly similar colors and grouping them into coherent regions. They extend the color locality assumption, seeking remote pixels with similar textures to color alike to effectively propagate the colorization, further improving the technique. A similar approach to transferring the color from scribbles, introduced by Qu et al. [24], is extracting statistical pattern features of local neighborhoods to measure texture continuity, resulting in fewer scribbles required.

A completely different method of optimization was introduced by Sýkora et al. with LazyBrush [25], along with relaxing the requirement of complete spatial accuracy of scribbles for the purposes of cartoon image colorization, by solving a multiway cut problem



(a) Target image

(b) User created scribbles

(c) Resulting image

Figure 3.2: Example of Levin's method

on a graph defined over image's pixels with edge weights calculated based on neighboring pixels' relative intensity levels, which, unlike the other algorithms, works well on images with large homogeneous regions.

As apparent from Figure 3.2, these methods can require significant amounts of user input. The advantage that they provide is in the ease of result refinement, as changing or adding more scribbles can effectively propagate the desired colorization.

3.2 Automatic color transfer

Much like scribble-based methods, algorithms which perform image-to-image color transferring expect the user to provide extra inputs. Simpler methods only transfer coloring onto the target image from a single image, though it is more common to define a set of images which serve as references for color extraction based on statistical properties. Some algorithms choose to process the target image with color enhancement algorithms discussed previously, to remove effects such as varying illuminance [26] or enforce global properties of the result, such as a desired or known color histogram [27].

Most algorithms extract various image features from the set, such as SURF, Gabor, patch or Daisy descriptors, and learn a mapping of these features to color channel data. These descriptors are then also extracted from the target image and mapped color distributions are transferred onto the regions that the obtained descriptors represent, such as in the work of Welsh et al. [28], who propose a method in which the features selected are the luminance value and statistical properties of 5×5 local neighborhood. Each pixel in the target image is matched to a set of these features extracted from the source image.

The set is produced by jittered sampling or using manually defined rectangular samples. After the best matching features are found, the color information is transferred onto the target pixel. The luminance channel remains unchanged, as is common to most colorization methods.

Gupta et al. [29] improve this approach by using a number of more advanced features which have rotational invariance and are extracted at multiple scale levels of the image. They attempt to make their method close to fully automated by running a web search on popular image sharing websites, based on keywords provided by the user instead of requiring reference images, acquiring semantically relevant results. The retrieved images are scored based on their colorfulness and non fitting images, such as grayscale or images with filtering effects applied, are discarded, creating reference image sets of up to 2000 images.

Similarly, Chia et al. [30] choose to perform an automated web search in conjunction with user provided foreground-background segmentation cues. This provides the user with more control over the resulting colorization, while retaining the automated nature of image-to-image color transfer. Liu et al. [26] automatically generate scribbles from the reference images obtained from the web and propagate them by using Levin's method, combined with automatic segmentation.

In Deep Colorization by Cheng et al. [31], a large dataset is divided up into smaller clusters based on global descriptors such as intensity histograms. For each cluster, a neural network consisting of 3 fully connected layers is trained, using multiple local feature descriptors computed at random pixel locations from images in the reference set as training data. The result is obtained as per-pixel colorization prediction of the network which has been trained on the reference set that most closely matches the target image based on the global descriptors, instead of explicitly defining the color transferring method.

Deshpande et al. [27] minimize an objective function automatically learned from example sets and subsequently train a forest of regression trees for color prediction, using multiple image filters to handle scale invariance. To choose good reference images and choose the used trees, bag-of-features retrieval on the training set is used.

While these methods generally require less from the user compared to scribble-based methods, they make it more difficult to influence the colorization output, due to reliance on data that are not straight-forward to interpret by visually inspecting the reference image set - such as feature descriptors.

3.3 Using CNNs

All methods described previously, with the exception of Deep Colorization, require some form of user assistance, which reduces their theoretical throughput for large scale colorization and can make them inconvenient to use, possibly resulting in having to resort to trial-and-error in order to obtain a satisfactory result. Given that their running times are generally in orders of minutes, that makes them not suitable for colorizing too many images.

However, with the CNN boom described in Section 2.1 some researchers started tackling this to a higher degree of success as a fully automated process by training CNNs on large datasets such as SUN or ImageNet. Currently, these methods are the state-of-the-art for natural image colorization.

It is worth noting that even the application of CNNs to this task can be viewed as a form of automatic color or style transfer - with the references automatically pre-selected by the choice of the original CNN training set - using a complex method learned and realized by the network. However, by using training set which contains a large variety of semantically different scenes and commonly occurring objects (such as ImageNet), it is expected that the chosen color transfer should be the best matching one.

Zhang et al. [32] propose a plain CNN with 22 convolutional layers on a subset of the ImageNet dataset, employing a custom tailored multinomial cross entropy loss with class rebalancing based on prior color distribution obtained from the training set to predict a color histogram for each output pixel to handle the multi-modal nature of the task.

Similarly, Larsson et al. [33] also predict a color histogram, however, they choose to use a 16-layer convolutional model attached to a fully connected *hypercolumn* layer to predict pixels' chromatic values, pretrained on image classification task and fine-tuned for colorization. Rather than train densely and predict the colorization of the whole image in one pass, the CNN is trained on spatially sparse samples of grayscale patches of size equal to the receptive field of the network, predicting the color value of the central pixel. Larsson et al. also explore the possibility of transferring a known ground truth color histogram (as a global descriptor) to improve the colorization.

Iizuka et al. [8] propose a network which combines two paths of computation, one to predict the global features of the target image and the other to specialize in local features. To achieve this, the global features are trained for image classification rather than colorization and are subsequently concatenated to the local features that are trained directly for colorization using L_2 Euclidean loss function. This technique allows their model to gain a higher semantic understanding of the image, producing very consistent colorizations.

Recent developments in the field of conditional *generative adversarial networks* (GAN), a model in which two distinct networks are trained - a generator and a discriminator - have lead researchers to attempt to use them for colorization. In the domain of colorization, the generator is used to produce a colorized image of the target grayscale image, and the discriminator is then trained to decide whether the generated image looks more convincing than the ground truth coloring. If that is not the case, the weights of the generator are updated in the direction of making the image more convincing for the discriminator, essentially using the discriminator as an adaptive loss function [34].

Cao et al. [35] show application of GAN to colorization of natural images while producing highly convincing colorizations on the SUN dataset. Along with the target grayscale image, a random noise vector is given to the generator as input (known as latent space sample), reintroducing some user-defined influence over the colorization result exhibited by methods in Section 3.2, albeit one that may be difficult to reason about.

Fu et al. [36] also use a cartoon movie dataset and choose to train a GAN model for automatic colorization, though their data source is of larger magnitude (over 15 hours of raw footage compared to less than 1.5 hours of our data). Their image extraction method is also different - sampling every 50 frames in the original footage - and, most importantly, their testing and validation sets are sampled randomly, which, due to the nature of the data source, skews the results, as it is reasonable to assume that randomly chosen frames may be mere translations of frames included in the training set, or that background information will easily be learned by recognizing objects in the image. Therefore, it is hard to compare the results of our work to results of Fu et al.

Chapter 4

Dataset

This chapter is focused on describing and discussing the chosen dataset and outlining its uniqueness compared to natural images as well as detailing the method of its extraction.

4.1 Rumcajs dataset

Since working with smaller datasets of natural images may yield visually unappealing results due to insufficient information included, and learning on larger datasets (such as ImageNet) makes for extremely unwieldy experimentation and hyperparameter tuning and also poses a significant challenge due their scope related to increased training times, we choose to limit ourselves to a smaller set containing cartoon images obtained from a Czech cartoon movie called ***O loupežníku Rumcajsovi***.

This dataset has been hand-colorized by scribble methods, which makes attempting to learn to colorize it automatically unique and interesting at the same time. Compared to learning natural image colorization, it forces the network to behave differently, as objects in scenes usually appear very sharply, without overly smoothed edges and with fewer textural cues, with large homogeneous regions. Rather than learning inherent colorization, it should aim to learn a perception of the images that is closer to how a human brain views it.

4.1.1 Format of the data

Every image that is in either training or testing set shares the same format. Image size is limited to 256 × 256 pixels, resized from the original 704 × 528 frame resolution. We decrease the resolution in order to increase the speed of training and reduce the amount of memory required to train. Since blurring (during potential upsampling back to original dimensions) the chrominance channel generally does not result in big visual differences, this is acceptable. 256 × 256 or 224 × 224 are sizes conventionally used for most image processing CNNs.

4.1.2 Method of extraction

Since the origin of the data set is a cartoon movie, there is a couple of problems surrounding the extraction of images. First, the movie is running at 25 frames per second, which means that many of the frames will be near-identical images. Therefore, a method of extracting images must be able to detect duplicate images and filter them out.

Secondly, there are some parts of the cartoon that we might not want to include in the data set, namely the opening and closing credits. The reason for removing the images depicting the credits is their uniformness and their general prevalence in the set. If we were to leave the credits frames in the data set, it may have resulted in increased tendency of the network to, for example, use shades of blue (since the credits are mostly a uniformly colored blue screen) even in images that were not credit frames.

To ensure that only reasonably differing images were extracted, we used simple squared mean thresholding, acting on consecutive frames extracted from the movie:

$$\frac{\sum_{p \in P} (frame_n(p) - frame_{n-k}(p))^2}{|P|} > t \tag{4.1}$$

where $frame_n$ is the currently processed frame, $frame_{n-k}$ is the last frame accepted as unique, P is the set of pixel locations, and t is an empirically obtained threshold, $t = 20$.

This effectively means that any noticeable difference in the two consecutive frames would result in the newer frame being accepted as a unique frame. Comparing against the last accepted frame rather than the last frame allows small changes to accumulate between accepted images.

This approach can be disadvantageous due to random noise present in the images, but working under the assumption that the amount of noise present in frames is roughly constant and similar in most frames, we can disregard it by accounting for it in the threshold value.

This simple method accomplishes the desired effect of filtering out duplicate frames and only preserving unique images. There are several instances where images are repeated due to the nature of the movie rather than frame difference (scenes that repeat the same sequences several times for example) but those are not common enough to warrant handling such cases specifically and those duplicates are allowed into the final data set.

Following the duplicate filtration, every frame was resized to 256 × 256 image size and saved into a lossless PNG file format, as using a compressed file format such as JPEG would result in loss of precision, which is particularly observable around object edges on this dataset and negatively impairs performance of the models.

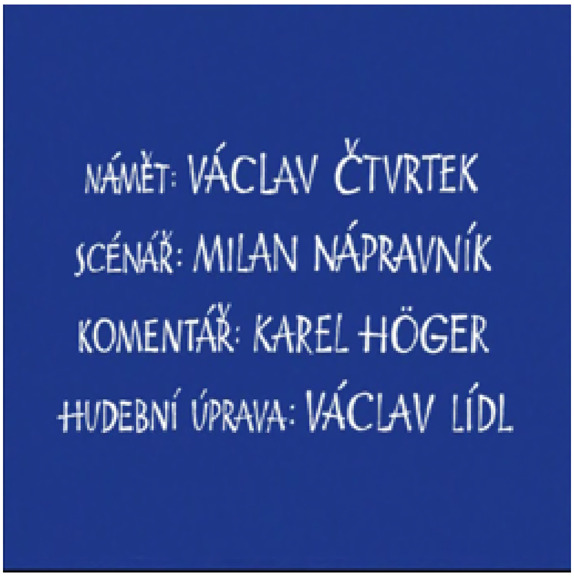
4.1.3 Filtering credits

The extracted images contained both the opening and closing credits of the cartoon, and since they made up quite a large portion of the overall number of images (roughly 4%), we decided that it would be beneficial to filter those images out and not use them for CNN training to minimize their impact on the final colorization.

The credits appear with fade in and fade out to black effects near their start and end, respectivelly. Since the credits are mostly static, the duplicate frame filtering algorithm already removes majority of them, usually leaving just a single image, with the exception of the first and last ones due to the fading effect.

Whenever the credits appear or disappear, the cartoon's images also fade out and in while staying constant, for that reason, we decided to also remove some of those faded frames as they generally bring very little new information into the set. Such step is not absolutely necessary, and in fact, these frames could act as a form of data augmentation, which is the reason we leave a small number of them in the dataset.

As can be seen in Figure 4.1, credit frames have a very specific format of blue background with centered white text in the middle. There are several issues that the filtering algorithm needs to deal with in order to successfully classify a frame as credits. What is perhaps even more important, however, is to reduce false positive rate to a minimum in order not to lose any additional images.



NÁMĚT: VÁCLAV ČTVRTEK

SCÉNÁŘ: MILAN NÁPRAVNÍK

KOMENTÁŘ: KAREL HÖGER

HUDEBNÍ ÚPRAVA: VÁCLAV LÍDL

Figure 4.1: Representative image of opening credits

First, the algorithm needs to be able to detect credits regardless of the fading effect, or at least correctly classify for the majority of the duration of the fading to limit residual images. Second, to reduce false positives, it must be able to provide a measure of confidence.

For this task, we elect to use a form of unsupervised machine learning. An image is converted into a collection of its pixels, yielding 65536 data points. Then we use the k-means algorithm to cluster these pixels into exactly two clusters based on their RGB values. The resulting centroids are then taken as objects of interest.

These centroids give us two RGB values which we examine further. What we are looking for is a large blue cluster representing the background, and a smaller white cluster for the text.

In order to make classification invariant to fading, we convert the RBG values to HSV color space and ignore the Value component of resulting colors. This technique seems to work rather well, as the fading is applied as a grayscale filter only.

Next, we check the hue and saturation values of the centroids as the main step of the classification. The blue centroid must have hue value between 190 and 260 and saturation of at least 250, while the white cluster is required to have a hue value between 190 and 250 and its saturation must not be greater than 120.

Via empirical testing, we concluded that credit images generally contain over 80% of pixels that can be considered blue. Thus, we also test sizes of the clusters, with the hypothesis that the blue cluster's magnitude is greater than 0.8.

These last two necessary conditions are strong enough to ensure that no legitimate frames were classified as credit frames. In practice, this algorithm was able to filter out the majority of the credit frames, which also provided offsets into the dataset, so that any remaining images could be filtered out by hand.

4.1.4 Training and testing sets

The cartoon movie itself is naturally split into 13 distinct episodes of roughly the same length. Despite that, there are several options of splitting the data into training and testing sets, so the choice warrants a more detailed explanation.

The first option would be to disregard episodes altogether and simply choose images at random to leave for the testing set, until a desired training/testing split is obtained. On less correlated data, this would be a good way to ensure generality, as the chosen testing data would span the entire data set and contain images independently on their position inside the set.

However, since the origin of the images is a movie, chances are that we would pick images that are very close to, or near identical to images contained in the training set (as identical as the initial filtration allowed), which would result in poor reflection of generalization - our testing data could be highly correlated with the training data.

Another way of splitting the data would have been to extract a portion of frames from every episode to use as testing data. The obvious disadvantage of this approach is that there is generally a limited number of objects that appear in an episode and therefore tend to appear at multiple points of it.

That could result in accidental overlap of training and testing data and making parts of testing data very representative of the training set, which would also poorly reflect the ability of the learned network to generalize.

Perhaps the most obvious one, and the one that we choose to use, is to leave some of the episodes as testing data. This has the advantage of conceptually making certain that most of the image sequences in testing data are previously unseen data during the training of the network.

Thus, the data is representative of the network's ability to generalize colorization onto new images. However, it also poses a challenge - if we choose an episode that is fundamentally different from the rest of the set, we may find that the resulting network will not perform well at all. We aim to choose such segment which contains some elements that were previously seen, but also many elements that are not present in the rest of the training data. We choose to leave the first of the 13 episodes (as they appear on the DVD) for testing.

In total, the training dataset contains 42715 images, while testing set contains 4434 images. However, these figures are overly generous, and the actual *meaningful* magnitude of the training set is much smaller, as most images contain simple modifications of previously seen frames such as subtle character movements or camera translations. Even though they introduce limited amount of new information into the training set, they act as a sort of data *augmentation* that is very natural to the origin of the images, video sequences.

Other common image data augmentation techniques include mirroring, randomized cropping and scaling, additive luminosity changes, randomized rotations, shearing, blurring or noise adding; none of these simulate video sequences particularly well. This is the reason why we allow the dataset images to have data differences smaller than is perhaps common, as well as the fact that there are at most several thousands of truly unique images included in the original video source. For instance, if we were to sample images every 50 frames of the original source, we would have a dataset of around 2620 images in total.

4.1.5 Additional remarks

The images were originally extracted from *O Loupežníku Rumcajsovi 1*, re-released on DVD in 2005 with updated colorization applied by manual scribble methods. 26 additional episodes exist, though those have not been recolored and therefore it would be unreasonable to include them in the training set, as their color properties are vastly different from the training images. We use one of these additional episodes to demonstrate the generalization properties of our learned models in Chapter 8.

4.2 Difficulty of the set

Compared to natural image datasets used by Larsson et al. [33] and Zhang et al. [32], the Rumcajs dataset has several specific features. While in natural images, many objects have distinct textures which help used methods to correctly select a color, the images in this dataset generally contain fewer texture types, consisting of sets of homogeneous regions of constant grayscale levels and so the trained model has to rely mostly on learning shapes (e.g. hats are shaped in a specific way and are red, boots are brown, ..) and spatial information (green grass tends to be in the lower part of the image, blue sky in the upper part) only.

Since texture often carries a large amount of information about the intended color, under missing texture information, objects are truly multimodal - it is significantly easier to colorize a wall with brick texture than a uniform grayscale region. Images 4.2a and 4.2b in Figure 4.2 perfectly illustrate this point, having differently colored backgrounds with only a few irregularities in the grayscale images.

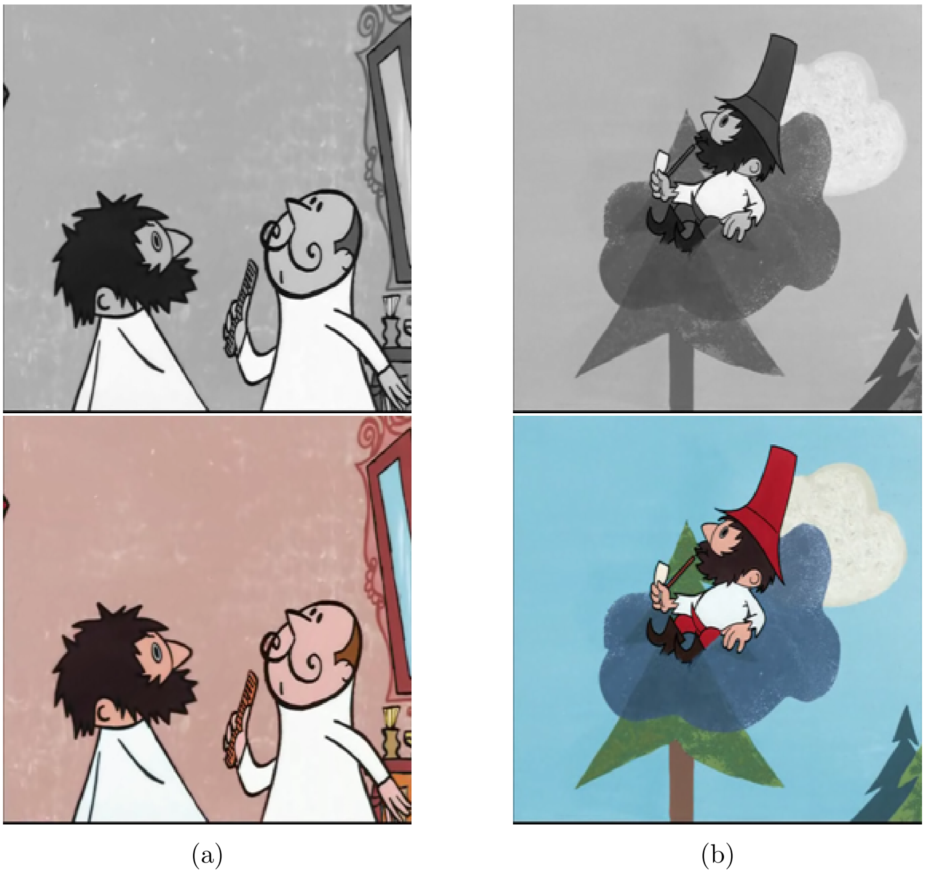

(a) (b)

Figure 4.2: Example of input data with ground truth colorizations. Notice the large regions of uniform intensity with little texture.

On top of that, the total number of different objects in the training set is relatively low (orders of magnitude lower than natural image datasets), which makes learning generalization to unseen objects more difficult. We hypothesize that these two point make the dataset significantly harder to automatically plausibly colorize overall.

4.3 Acknowledgement

We acknowledge that the used dataset is copyrighted material owned by Česká televize.

Chapter 5

Method

In this chapter, we look at several challenges that this task presents and how we choose to solve them. First, we discuss the various possible approaches to the problem in terms of overall design. Then, we take a more detailed look at individual components.

When considering the usage of CNN for this task, a natural question that may arise is whether it would be possible to use transfer learning as described in Section 2.2.12. Other networks that deal with related problems exist, such as ResNet mentioned in Section 2.1 designed for image classification. It is indeed true that many traits of the networks are shared, particularly the low level image feature extraction layers, and thus it may be possible to successfully fine-tune using at least the initial layers of a network trained for a task like image classification.

However, the first problem appears from the fact that all publicly available classification networks are trained on colored images and their performance on grayscale images is generally much lower. This is not a fundamental architectural problem, the network can still produce features from grayscale images, but the features are likely to turn out to be inefficient for the colorization task.

Another problem comes from the inner representation learned by the other networks. Since the color data is expected to be present in the input image, it is unlikely that networks trained for classification would be able to generate it based on grayscale image, if we were to train e.g. an extension on top of the hypercolumn image representation of the VGG network architecture.

Larsson et al. [33] train a fully connected model while using modified VGG-16 convolution layers, fine-tuned on grayscale image classification, but only choose to predict a single pixel's color at a time. During experiments, we attempted training a network on top of up to 9 initial layers of VGG-16 with frozen weights (disabled updates) that performed dense prediction, however, we conclude that training networks for colorization from scratch to be the best course of action, as it resulted in better outcomes.

5.1 Overall approach

In this task, our goal is to take a grayscale image, a single channel of image data, and transform it into a standard RGB image, an image with three channels of data.

We pose the problem as learning a mapping C such that $R = C(G)$, where $R \in \mathbb{R}^{H \times W \times 3}$ represents an image with RGB channels and $G \in \mathbb{R}^{H \times W}$ represents an image with grayscale values only. To learn this mapping, we use CNN based models. To simplify training and reduce memory requirements to tractable amounts, we downsample the input images to 224×224 resolution and output a 56×56 resolution colorization, which is scaled up to match the grayscale input dimensions. Despite the radical decrease in resolution, it should be possible to learn a good looking colorization, as evident from the work of Zhang et al. [32], even if some finer detail is lost.

The input's total number of pixels is 50176 and output's total number is 3136. To upsample the colorized result, we use a conventional spline interpolation algorithm.

We can observe several properties of the formulation that influence how we approach the task.

5.1.1 Color space

First, we notice that as much information in the grayscale image ought to remain unchanged in the final, colored image, and that it should act as a passthrough channel. Therefore, color information would preferably be separate channels added to the grayscale image to obtain the result.

This effect is easily achievable by working in the CIE Lab color space. The Lab color space was deliberately designed to more closely match human perception of an image, compared to more standard RGB color space, and to contain a channel with a measure of luminance maximally decorrelated from chrominance. An image in the Lab color space therefore consists of one channel for achromatic luminance (L) and two color channels (ab, sometimes noted as $\alpha\beta$). The a channel controls hues between green and red, while the b channel has control over hues between blue and yellow. See Figure 5.1 for channel-wise decomposition of the Lab color space.

By convention, the values range from 0 to 100 for the L channel, and from -128 to 127 for a and b channels, though not every combination of these values maps into the visible sRGB color gamut, especially the marginal values of a and b, and for that reason, we only consider values in the range of -110 to 110.

Still, in this range there are combinations which do not map back into the sRGB space naturally, in which case the conversion values are clamped into the conventional RBG space of $\langle$ 0, 255 $\rangle$ [37].

In this color space, our grayscale input image is seamlessly represented by the luminance channel, leaving us with the task of estimating the chrominance channels a and b. HSL and HCL could alternatively be used, as they implement a similar concept of separating the lightness channel.

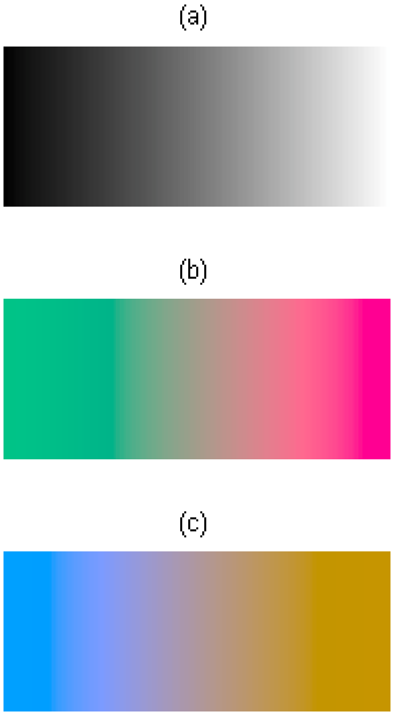


Figure 5.1: Lab color space decomposition. **(a)** L channel with a, b = 0. **(b)** a channel with L = 65, b = 0. **(c)** b channel with L = 65, a = 0.

5.2 Color channels estimation

When estimating the a and b channels, there are several possible approaches to choose from. The choice can significantly affect the resulting visual colorization, training time or final CNN performance.

A straightforward option would be to estimate the values of a and b for each pixel directly. The advantage of this approach is the simplicity, both in terms of reasoning and implementation. It provides the option of using a simple loss functions, such as L_2 Euclidean norm, which speeds up training.

Disadvantages are more subtle. One disadvantage is that this approach leaves no room for the notion of probability distribution over possible results, possibly resulting in jarring transitions between colors. For example, when a CNN predicts a pixel to have red color, there is no method of retrieving other possible colorizations or the level of confidence in the prediction.

Additionally, this approach does not handle multimodality well. If an object can take on a range of different ab values, the optimal solution to the Euclidean loss will tend to average out towards zero, producing desaturated colors. This approach is used by Iizuka et al. [8], in conjunction with Euclidean norm in ab space as loss function.

A completely different approach is to pose the task as a classification problem, which is a well-studied area in the domain of CNNs, essentially turning the problem into estimating a color histogram for every pixel in the target image. We use the notion of color histogram in this context, but it is equivalent to confidence probability distribution predicted by the network for every pixel, since the final layer uses a softmax activation function, which normalizes the outputs to form a proper distribution.

First, we define a set of canonical colors as classes to predict. Larsson et al. [33] propose sampling the ab space (or in their case, the hue/chroma space) based on evenly spaced Gaussian quantiles and selects 1024 values around the origin, but due to this sampling, this method may heavily favor more desaturated colors thanks to over-representation. Similarly to Zhang et al. [32], we quantize the ab grid space into evenly spaced bins, taking a point every 10 units of the grid and remove any points that do not map to colors inside of the sRGB gamut. This leaves us with 313 proper ab pairs which can be used.

To convert a given value from ab to a color histogram, we find k-nearest-neighbor ($k = 5$) values in this quantized set and create a convex combination that closely matches the original value (proportionate to the distances from the neighbors). Furthermore, this "sharp" histogram is smoothed with a Gaussian kernel ($\sigma = 5$) to speed up CNN training convergence. This process is very similar to color quantization. This method works better than e.g. 1-hot encoding of the closest point on the grid, because it simulates the computation performed by CNN (it is harder to learn sharp peaks than smoothed distributions).

The CNN's function is then to estimate color histogram for every pixel of the output image, essentially classifying each pixel's color. In our case, that means estimating a matrix of the shape $56 \times 56 \times 313$. The cost function then calculates loss based on the this prediction and colors of the ground truth image encoded by the scheme described above.

Afterwards, there are multiple methods of converting the predicted color histogram back into an ab pair in order to obtain the result.

1. **Mode** - treating the histogram as 1-hot encoding and assigning the pixel the color of the bin with highest prediction confidence.
2. **Expectation** - summing over the bins' ab values, weighted by the histogram value.
3. **Sampling** - drawing a random sample from the estimated probability distribution.

Since in our framework this conversion has to be done per-pixel, using mode tends to create areas with frequent and jarring color shifts where the prediction peaks change. When an artifact is produced, the area is very crisply defined. Sampling tends to create color jittering in locations where the prediction of the network creates color histograms with multiple peaks. We find that using expectation achieves the best visual quality of the result, which is consistent with findings of [32], [33]. Instead of creating artifacts that have crisp boundaries, expectation "blends" them in.

The prediction layer is activated by the softmax function, as is common when predicting probability distributions. To extract colorization results in Lab color space, an additional 1x1x313 conv layer with 2 output channels can be inserted after the prediction layer. This produces ab values for every pixel by implementing the expectation calculation of the form:

$$ab = \sum_{i=1}^{313} p_i \cdot q_i \tag{5.1}$$

where p are the predicted probabilities and q are the canonical 2-dimensional ab points we selected on the ab grid.

To obtain the colorization of the input image, the ab values are treated as a 2 channel image, upsampled through conventional spline interpolation to match the original image dimensions (256×256) and concatenated to the input image L channel, producing a Lab image that can be converted back into RGB space.

5.3 Loss function

Regardless of the output color representation, we require a loss function to measure prediction errors and obtain meaningful gradients to update the network's weights.

Since the color histograms are closely related (by construction) to probability distributions, we use the Kullback–Leibler divergence function as a natural distance function of two probability distributions.

Additionally, we enhance the loss function by using a prior-boosting reweighting factor, a technique called class rebalancing. The distribution of ab values in most images is strongly biased towards certain ab values, due to the presence of backgrounds such as blue sky, or green grass which take up significant portions of images. Figure 5.2 shows the measured distribution of pixels in ab space, gathered from all images in the training set. Notice that the number of pixels using more desaturated shades of blue and green is significantly higher than more vibrant colors.

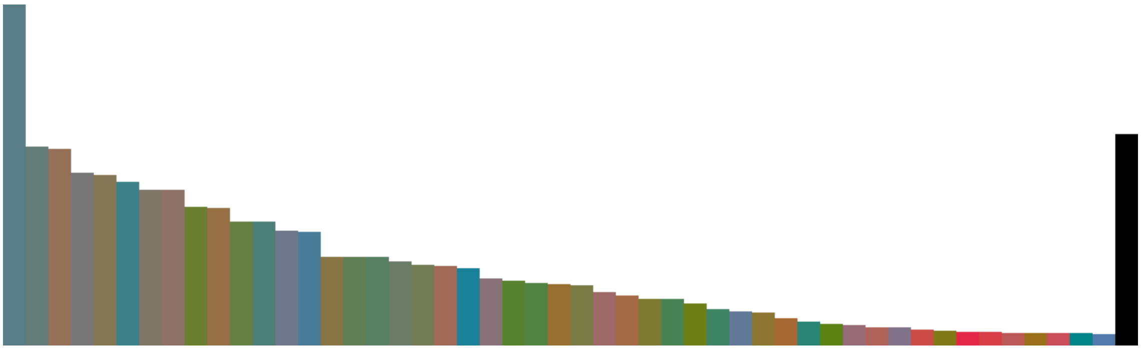


Figure 5.2: Measured prior color distribution on the training set, showing 49 most common colors with column heights proportionate to relative probability of occurrence. The last column shows the mass belonging to all remaining colors combined.

With that in mind, the loss function becomes:

$$L(\hat{P}, P) = -\sum_{h,w} v(P_{h,w}) \sum_{q=1}^{313} (\log(\hat{P}_{h,w,q}) - \log(P_{h,w,q})) \tag{5.2}$$

where P are the ground truth image's colors encoded into color histograms, $\hat{P}$ is the network prediction of color histograms for pixels, h, w represent pixel indices and v is the weighting term (a function of the ground truth's histogram) that we use to rebalance the loss to emphasize usage of rare colors defined by Zhang et al. [32] as

$$v(P_{h,w}) = w_{q^\star}, q^\star = \arg\max_q P_{h,w,q} \tag{5.3}$$

$$w \approx \frac{1}{(1-\lambda)r + \frac{\lambda}{313}} \tag{5.4}$$

where r is the empirical prior distribution of ab pairs calculated over the training set and λ is a term used to create a mixture of the training set prior probability and a uniform distribution. During all training, we set $\lambda = 0.5$. The mixture of uniform distribution with prior probability ensures that the loss function does not favor one color over another too disproportionately. Weighing term w is normalized such that $\sum_q r_q w_q = 1$ holds.

Effectively, loss defined in this manner rebalances each pixels' contribution to the final loss inversely proportionate to how rare the color of the ground truth pixel is according to the prior distribution. Zhang provides hints that compared to non-rebalanced or L_2 loss function, this cost function may result in lower accuracy, but generally produces more plausible colorizations thanks to the strong preference of rare colors.

Chapter 6

Network architecture

In this chapter, we will descripe the used CNN architectures, including individual layers and their hyperparameters. We choose to train two different architectures.

The first CNN architecture is a classical CNN with convolutional layers stacked on top of one another in a straight-forward way. The second architecture draws inspiration from the ResNet architecture, providing shortcut connections in between layers that are not directly on top of each other in the convolutional layer "pipeline".

Both networks are capable of processing images of arbitrary size, downsampling the output by a factor of 4, producing same output form for easy comparison, but work best on the image size they were trained on - 224 × 224.

In Chapter 8, we compare the results produced by both networks, and attempt to point out their strengths and weaknesses on two different data sets.

6.1 Pooling layers

An important feature that both network architectures have in common is that they employ no pooling layers. Part of the reasoning for this decision was given in Section 2.2.3, pointing out that pooling layers are a form of downsampling. Even though some downsampling is required to effectively expand the network's receptive field, we found pooling to have adverse effects on the results in both architectures, which we attribute to the fact that pooling layers drop too much information about the inputs.

Instead, downsampling is achieved through increasing the stride of some of the convolutional layers. This is a more natural fit for the task of colorization, as more information is preserved and can reach the later layers of the network.

6.2 Plain CNN model

The first model that we train variants of is identical to the model used by Zhang et al. [32] for natural image colorization on ImageNet. It is a model composed of stacked convolutional layers only (hence plain). A general overview of network architecture can be seen in Figure 6.1, and a detailed listing of layers and their hyperparameters is shown in Table 6.1.

Each block of **conv** layers refers to a groupping of 2 to 3 convolutional layers followed by rectified linear unit activation for each individual conv layer. Between blocks, a batch normalization layer is inserted to help prevent exploding or vanishing gradient problems and speed up convergence. All convolutional layers learn filters of size 3 × 3, with the

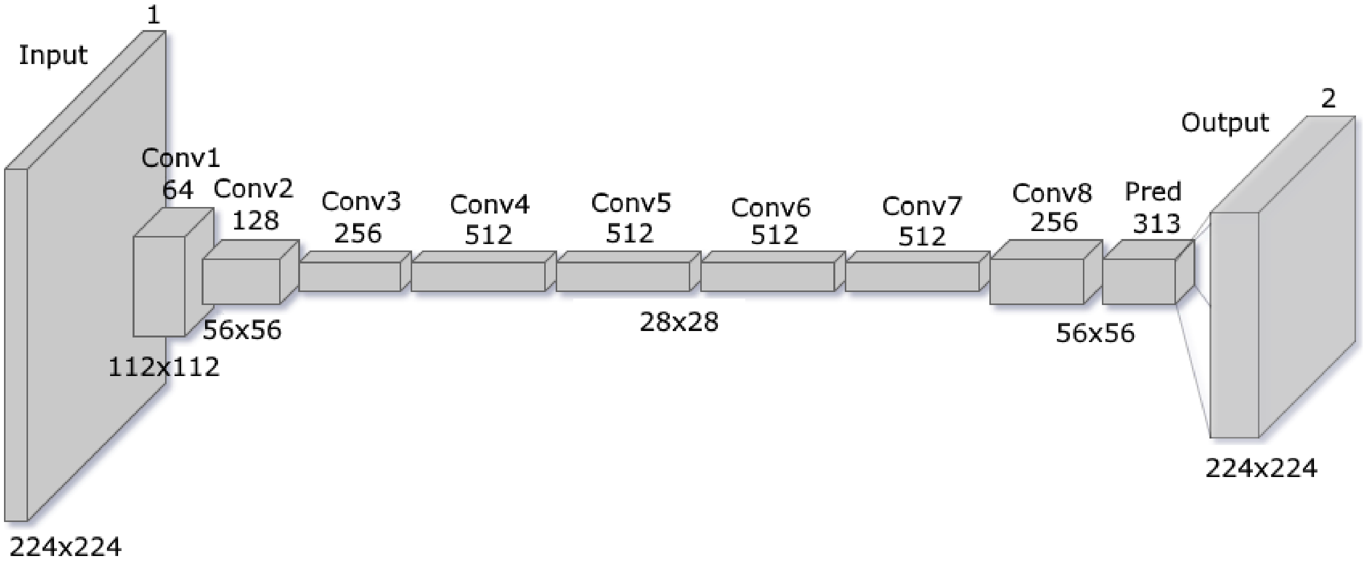


Figure 6.1: Plain CNN network architecture

exception of the upsampling transposed convolution (”deconvolutional” in Caffe) **conv8_1**, which uses 4 × 4 filters instead.

In total, there are 22 convolutional layers split into 8 blocks, plus the prediction layer at the end. Initial layers contain a lower number of output channels, to simulate the use of a moderate number of low level features, usually very similar to filters that are used by many corner or edge detection algorithms. This can be more difficult to recognize and have a less noticeable effect when using small filter sizes, but encodes information equally well.

The initial layers function as *feature extractors*, while the later layers have the role of information encoding.

The resolution is quickly reduced to half of input size and quarter of input size after 2 and 4 layers, respectively. It is further reduced to one eighth of input size in the **conv3** block and later upsampled back to one fourth by block **conv8** via learned upsampling filters. One fourth of input size is the total prediction output resolution.

Using lower resolution for the densest layers helps prevent overfitting and requires the model to learn effective encoding of the input, allowing denser concentration of information in the encoding layers, which are in blocks **conv4** through **conv7**, as well as rapid growth of effective receptive field, shown in the **ERF** column of Table 6.1. This is where most of the network's parameters are located, over 27.13 million of the total 32.23 million parameters learned by the network.

Blocks **conv5** and **conv6** have dilated convolutions with dilation set to 1, further increasing the growth of receptive field. Every conv layer has input zero padding set such that its convolutions have valid ranges to operate over all of the previous layer's outputs, which translates to 2 pixels of padding in layers with dilation set to 1 and 1 pixel of padding in the remaining layers.

Block	**Layer id**	**R**	**O**	**S**	**D**	**ERF**	**P**
-	data	224	1	-	-	-	-
conv1	conv1_1	224	64	1	0	3	0.5
	conv1_2	112	64	2	0	5	36.8
conv2	conv2_1	112	128	1	0	9	73.7
	conv2_2	56	128	2	0	13	147.4
conv3	conv3_1	56	256	1	0	21	294.9
	conv3_2	56	256	1	0	29	589.8
	conv3_3	28	256	2	0	37	589.8
conv4	conv4_1	28	512	1	0	53	1179.6
	conv4_2	28	512	1	0	69	2359.2
	conv4_3	28	512	1	0	85	2359.2
conv5	conv5_1	28	512	1	1	117	2359.2
	conv5_2	28	512	1	1	149	2359.2
	conv5_3	28	512	1	1	181	2359.2
conv6	conv6_1	28	512	1	1	213	2359.2
	conv6_2	28	512	1	1	245	2359.2
	conv6_3	28	512	1	1	277	2359.2
conv7	conv7_1	28	256	1	0	293	2359.2
	conv7_2	28	256	1	0	309	2359.2
	conv7_3	28	256	1	0	325	2359.2
conv8	conv8_1	56	128	0.5	0	341	2097.1
	conv8_2	56	128	1	0	349	589.8
	conv8_3	56	128	1	0	357	589.8
pred	pred_313	56	313	1	0	357	80.1

Table 6.1: Plain CNN network architecture. **R** spatial resolution of output, **O** number of channels in output, **S** layer stride, **D** layer dilation, **ERF** effective receptive field with regards to data layer, **P** number of learned parameters in thousands rounded off

6.3 Residual CNN model

The second model that we introduce and train is one that is inspired by the success of the ResNet models in area of image classification, mentioned in Section 2.1. These models show much faster convergence rates as well as improved performance.

As more research was carried out in the field of training various deep CNNs architectures, it became apparent that the problems vanishing or exploding gradients, which can mostly be resolved by applying normalization, were not the only factors hampering convergence and accuracy of networks.

In particular, a problem of *accuracy degradation* was uncovered [6], [38]; during the training stage, accuracy gets saturated, as is expected when training converges. However, if training continues past the saturation point, accuracy starts degrading rapidly. A similar phenomenon is known as *overfitting*, but this is not the cause of degradation, since adding more layers to the model results in increased training error as well as reduced accuracy, the opposite of what would be expected in the case of overfitting. With increased model

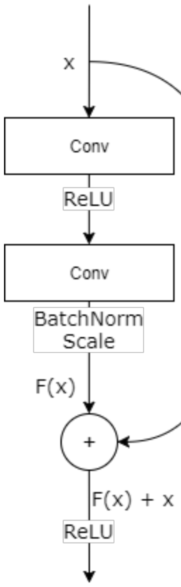


Figure 6.2: Building block of residual learning, the number of conv layers between shortcut connections generally does not exceed 3-4 layers

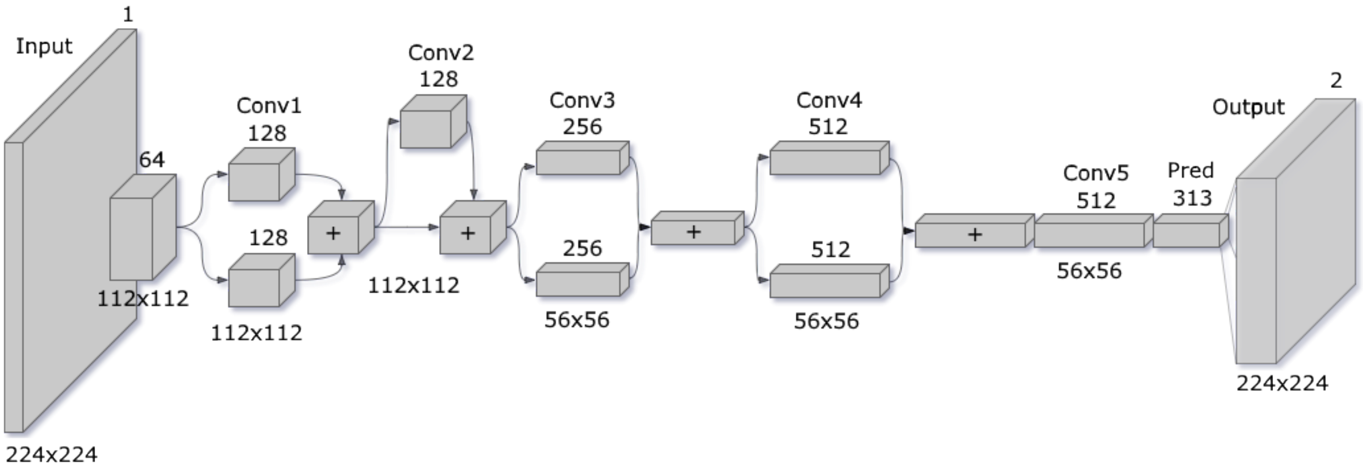


Figure 6.3: Residual CNN network architecture

variance, overfitting would result in further reduced training error.

This indicates that deeper models are more difficult to train, even when vanishing/exploding gradient problem is accounted for by normalization, with the likely culprit being the number of stacked non-linearity functions [39]. In plain convolutional networks, the expectation is that each few stacked layers (blocks in the Plain CNN architecture) learn the underlying mapping produced by the following block of layers, denoted as $\mathcal{H}(x)$, which gets progressively more difficult to learn effectively as we add more layers.

Several techniques to alleviate this problem have been proposed, such as using better optimizing algorithms like Adam (described in Section 2.2.11)) or improved initialization techniques (Section 2.2.10).

ResNet models choose to solve this problem by using a different approach. Instead of expecting every block to learn the mapping $\mathcal{H}(x)$, the stacked layers are allowed to fit mapping $\mathcal{F}(x) = \mathcal{H}(x) - x$. In [6], the authors hypothesize that it is inherently easier to optimize this *residual* mapping.

In the CNN, the modification of the mapping is realized with ”shortcut connections” between blocks of layers, as shown in Figure 6.2. After a block of layers has computed its outputs, the original inputs are added to it via element-wise summing.

Block	**Branch**	**Layer id**	**R**	**O**	**S**	**D**	**ERF**	**P**	**BN&S**
-	-	data	224	1	-	-	-	-	-
-	-	conv0	224	64	1	0	3	0.5	-
conv1	top	conv1_branch2a	112	128	1	0	5	73.7	-
	top	conv1_branch2b	112	256	2	0	9	294.9	Yes
	bottom	conv1_branch1	112	256	1	0	5	147.5	Yes
sum1	-	res1	112	256	-	-	13	0	-
conv2	top	conv2_branch2a	112	256	1	0	13	589.8	-
	top	conv2_branch2a	112	256	1	0	17	589.8	Yes
sum2	-	res2	112	256	-	-	21	0	-
conv3	top	conv3_branch2a	56	512	2	0	21	1179.6	-
	top	conv3_branch2b	56	512	1	1	37	2359.2	-
	top	conv3_branch2c	56	256	1	1	53	1179.6	Yes
	bottom	conv3_branch1	56	256	2	0	29	589.8	Yes
sum3	-	res3	56	256	-	-	69	0	-
conv4	top	conv4_branch2a	56	512	1	1	69	1179.6	-
	top	conv4_branch2b	56	512	1	1	85	2359.2	Yes
	bottom	conv4_branch1	56	512	1	0	77	1179.6	Yes
sum4	-	res4	56	512	1	0	93	0	-
conv5	-	conv5_1	56	256	1	0	93	1179.6	-
	-	conv5_2	56	512	1	1	109	2359.2	-
	-	conv5_3	56	512	1	0	117	1179.6	Yes
pred	-	pred_313	56	313	1	0	117	160.2	-

Table 6.2: Residual network architecture. **R** spatial resolution of output, **O** number of channels in output, **S** layer stride, **D** layer dilation, **ERF** effective receptive field with regards to data layer, **P** number of learnt parameters in thousands rounded off, **BN&S** is followed by a sequence of batch normalization and scaling layers

The overall network architecture that we propose can be seen in Figure 6.3, and a detailed overview of layers is found in Table 6.2.

In the original ResNet model, the shortcut connections are simply identity operations with no modification of the per-block input x. In our case, due to the necessity of downsampling the input early on, the shortcut connections perform a single conv operation where necessary, downsampling by increasing stride. These layers are denoted in Table 6.2 as being in the bottom branch.

Compared to the original ResNet, we also include a scale layer after any conv layer that connects its outputs directly to the element-wise sum layers **res1**, **res2**, **res3** and **res4** to allow the netwok to rebalance residual weights on a per-parameter basis.

The network does not contain a shortcut connection between the last conv block (**conv5**) and the prediction layer, as we found that this connection caused obvious artifacts on images with low overall luminance. This can be seen as an analog to a fully connected layer at the end of a classification network, which also usually does not receive shortcuts.

Most other hyperparameters are shared with the plain convolutional model in Section 6.2. All used conv layers learn 3×3 filter sizes. Input and output sizes are not fixed, but output resolution will be one fourth of input resolution. Spatial downsampling in both branches is implemented with strides only, zero padding is added to retain resolution where needed.

The first conv layer also produces 64 output channels to simulate low level image features. In total, the longest path from input to output contains 13 conv layers (always top branch), 5 of which are dilated convolutions. This is a rather low depth, but since ResNet models generally require much higher amounts of memory to train, we found this to be the best tradeoff with limited hardware options. In principle, the network should exhibit even better results with added depth.

Compared to the plain CNN architecture, this model contains about half the number of total parameters (16.6 million vs 32.2 million) and has roughly one third of the effective receptive field size. Considering we use 224×224 images for training, this is likely to cause the network to colorize small and moderately sized objects more evenly, but produce more artifacts in images that are mostly homogeneous.

Chapter 7

Training details

We use the Caffe [40] framework to train all models, but fundamentally, any deep learning framework (Theano, TensorFlow, MXNet, ..) could be used, as the differences are mostly syntactic rather than semantic. Caffe has the advantage of providing generally good computational performance, but perhaps with a slightly higher barrier to entry with prototxt files usage, which define a domain specific language, compared to the other, mostly Python based, frameworks.

The models are not quite trained end-to-end, the final step of concatenating predicted ab values with the original L channel values is done outside of the network, by an external Python script, though it could be implemented with a special concatenate layer (not packed with Caffe by default, but used by e.g. Iizuka et al. [8] for concatenation of low level and high level features).

To achieve the color to color-histogram conversions and for calculation of the rebalancing term described in Section 5.3, we use custom Python layers.

7.1 Trained variants

For comparison and to demonstrate strengths of individual setups, we train several different variants of our method. Firstly, we train both networks as described in Chapters 5 and 6 without any modifications, to establish a baseline. Afterwards, we also train both networks without the color class rebalancing term of the loss function defined in Formula 5.3 to confirm its effects and uncover possible advantages and disadvantages. Because validation of the models is unclear (no precise metric for plausible colorization), we train a model that uses dropout layers after each **conv** block in the plain convolutional model to show that the visual failures of the method are unlikely to be caused by overfitting and appear even with further regularization besides only batch normalization and input normalization applied during training, as suggested by Srivastava et al. [13].

We include a dropout layer after each **conv** block, and progressively increase the dropout ratio, starting with 0.1 after **conv1** block and ending with 0.3 after conv8 **block**.

Additionally, to verify the differences between classification based colorization and ab value prediction colorization, we train a variant of the plain convolutional network which replaces its **pred** layer with a 2 channel output ab layer, which encodes ab values predicted by the network directly, using the same output resolution as the classification networks of $\frac{input}{4}$.

In this network, the final softmax activation function is replaced by a tanh activation layer (which has the convenient range of $[-1, 1]$), and its output is multiplied by a constant coefficient to span the whole standardized ab space. The loss function used for calculating gradients is replaced by the Euclidean (L_2) loss function without rebalancing.

Therefore, all trained variants are as follows:

1. Plain CNN
2. Plain CNN without rebalancing
3. Plain CNN with dropout
4. L_2 ab CNN (plain CNN)
5. Residual CNN
6. Residual CNN without rebalancing

7.2 Initializations

The plain CNN model variants are initialized with weights derived by the MagicInit tool developed Philipp Krähenbühl et al. [17] from 10 iterations over the whole training set. This increases the convergence speed and helps avoid the vanishing/exploding gradient problem.

However, the tool does not currently support scale or element-wise layers used by the Residual model and since the calculation of initialization requires all following layers to have been initialized, it would thus only produce a good initialization for at most the last 3 layers. We instead use the Xavier initialization described in Section 2.2.10 for initialization of the residual CNN variants. Perhaps surprisingly, we found that compared to the data-dependent initialization of the CNN, it did not result in noticeably worse initialization as far as convergence speed was concerned, although initialization may not be the biggest factor in this case - considering the architectures are different.

7.3 Optimizer

There are several options when choosing algorithm to use for weight update optimization. Generally speaking, for training CNNs, SGD has been historically used. With later advancements, Adagrad or Adadelta became the standard, and since its introduction in 2015, Adam has been used for image CNN training almost predominantly.

Since Adam seems to be the go-to optimizing algorithm among researchers who use CNNs, we decide to follow suit in this matter. There are certain disadvantages to using Adam, such as increased memory usage, as it keeps at least two state variables for each weight - generally increasing memory requirements by a factor of 3, which can cause some issues if hardware is a limiting factor, bringing the need to reduce batch size in order to fit the network into the GPU VRAM, but the convergence speed gains are significant when compared to plain SGD. Note that most other algorithms also have this disadvantage.

In all training runs, we use Adam with global learning rate η initially set to 0.001, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. η is automatically multiplied by 0.3 after 15 epochs (when training loss usually stagnated), to fine-train the models.

7.4 Training

All models have been trained on at least 30 epochs of the whole training dataset. Due to different memory requirements, the batch sizes are not consistent, though Caffe allows setting the *iter_size* hyperparameter, which accumulates gradients for several batches before updating the network's parameters.

The plain CNN model was trained with batch size of 24 and iter_size of 2, effectivelly performing weight updates after every 48 data samples. The residual CNN model, as it requires more memory, has the batch size set to 4 and iter_size to 12, to retain the same number of examples per weight update.

Additionally, the images are randomly mirrored by the Caffe framework when used as training samples, and randomly cropped to size of 224 × 224. These augmentations complement the dataset well. Prior to entering **conv1** blocks of the networks, the images are converted to Lab space by a special ColorConversion layer, the ab dimensions are sliced off and used to generate ground truth color histograms. The L channel is multiplied by constant 0.01 to normalize the 0-100 luminance to the 0-1 range and used as the input of the network.

The training requires ~ 890 iterations per epoch of data, and around 26700 iterations to complete the training of 30 epochs. Normally, this could be considered a low number of epochs to run (it is common to pass upwards of 100 epochs when training from scratch), especially for an image-to-image CNN, but considering the points made in Section 4.1.4, specifically the ones about many images being very similar, this can be viewed as running many more epochs with specially augmented data.

While we observed small improvements when training past 30 epochs in some areas, it generally came with degradation of other cases, most often unnatural color shifts observed on large backgrounds, particularly in the variants which used color rebalancing. Note that since input dependent rebalancing is used, the absolute value of loss attained during training provides little information other than comparison between models. This applies to testing loss too; plausible colorization is disproportionately penalized by any loss function, which means that testing error has little correlation with the final visual result of the colorization.

A single epoch takes from 10 minutes (Plain L_2 ab CNN variant) to 5.25 hours (Residual CNN variant). All training and timing tests were performed on a single Nvidia GTX 970 with 4 GB VRAM GPU with cuDNN capabilities, CUDA version 9.0, cuDNN version 7.0, Intel i7-4790K CPU with 16 GB of RAM.

Chapter 8

Results and experiments

In this chapter, we first compare the trained variants quantifiably, with image error metrics compared to the ground truth colorization. Then, we present colorized results of the different variants of the network architectures and discuss these results. In Section 8.3, we set our results side by side with automatic color transfer methods. Finally, we propose some possible further improvements of the obtained results with post processing algorithms.

To obtain the colorized images with the same resolution as input images, we use the spline interpolation method implemented in the SciPy Python library [41] in all variants. The input size used for the network prediction used are 224×224 (56×56 output). The L channel is passed through from the input image to the output image unchanged and this transfer is done with the highest resolution available (i.e. after upsampling of the color channels - 256×256) to preserve as much quality of the grayscale input as possible.

8.1 Testing set results

To compare the variants, we offer several metrics calculated between the ground truth testing episode frames and colorized results. In Table 8.1, we provide the results of three metrics: Root-mean-square error (RMSE), which measures direct color differences between ground truth and colorized images, calculated as:

$$\mathrm{RMSE}(P_{\alpha\beta}, \hat{P}_{\alpha\beta}) = \frac{1}{I \cdot N} \sum_{i=1}^{I} \sum_{n=1}^{N} \sqrt{\|P_{\alpha\beta}^{i,n} - \hat{P}_{\alpha\beta}^{i,n}\|^2} \tag{8.1}$$

where I is the number of images and N is the number of pixels in an image. The error is calculated in de-correlated 2-channel $\alpha\beta$ color space, used by Larsson et al. [33] and Deshpande et al. [27], defined as

$$\alpha = \frac{B - \frac{1}{2}(R + G)}{L + \epsilon} \beta = \frac{R - G}{L + \epsilon} \tag{8.2}$$

where $L = \frac{R+G+B}{3}$, $\epsilon = 0.0001$ and R,G,B channels are normalized $\in [0, 1]$.

Peak signal-to-noise ratio (PSNR) metric, designed to approximate human perception of image differences, commonly used as a quality measurement of image compression algorithms (higher is better). While not directly related, learning colorization methods can be seen as means of compression - by automatically inferring the missing data channels. It is calculated in the RGB space as:

Model/Metric	RMSE	PSNR	PA
Plain	0.3909	18.5312	0.1292
Plain no rebalance	**0.3629**	18.5768	0.1536
Plain L_2 ab	0.3638	**18.8571**	**0.0465**
Plain with dropout	0.3823	18.4803	0.1344
Residual	0.3989	18.0632	0.1745
Residual no rebalance	0.3935	18.2205	0.1624

Table 8.1: Testing performance of trained variants under RMSE, PSNR and PA (thresh = 2) metrics.

$$\mathrm{PSNR}(P_{RGB}, \hat{P}_{RGB}) = \frac{1}{I}\sum_{m=1}^{I} -10 \cdot \log_{10}\Big(\frac{\sum_{n=1}^{N}\|P^{i,n}_{RGB} - \hat{P}^{i,n}_{RGB}\|^2}{3N}\Big) \tag{8.3}$$

where I is the number of images and N is the number of pixels in an image.

And lastly, the custom perceivable-artifacts (PA) metric, which aims to capture the model's ability to produce consistent colorizations, without artifacts that were not present in the ground truth, calculated in (L)ab space as:

$$\mathrm{PA}(P_{ab}, \hat{P}_{ab}) = \frac{1}{I}\sum_{i=1}^{I} \frac{\sum_{n=1}^{N}(|P^{i,n}_{ab} - P^{i,n,neigh}_{ab}| < thresh) \cdot (|\hat{P}^{i,n}_{ab} - \hat{P}^{i,n,neigh}_{ab}| < thresh)}{\sum_{n=1}^{N}(|P^{i,n}_{ab} - P^{i,n,neigh}_{ab}| < thresh)} \tag{8.4}$$

where I is the number of images and N is the number of pixels in an image, $neigh$ contains pixels in valid 4-neighborhood of a pixel and $thresh$ is a user-defined value controlling the leniency of the metric. This metric calculates the ratio of pixels in the colorized image that have a consistently colored neighborhood (all neighboring pixels have ab values within $thresh$ of the investigated pixel) compared to the number of consistently colored pixels in the ground truth image. A pixel is only counted towards the error value if the same pixel position has consistently colored neighborhood in the ground truth image, but does not in the colorized image.

We include this metric because when comparing to ground truth images, the RMSE and PSNR metrics will penalize reasonable, but incorrect color choices for many objects (e.g. blue skirt instead of red skirt) more than obvious colorization artifacts. PA decouples the color values of ground truth and the result, instead focusing on color differences between neighboring pixels only.

Further, in Figure 8.1, we provide cumulative histograms of per-pixel RMS errors for all model variants, as well as cumulative histograms of per-image RMS errors for all model variants. These figures better demonstrate the distribution of error magnitudes for each variant, showing which models achieve higher percentages of pixels (images) with low errors. Higher per-image errors generally mean either bad spacial artifacts or incorrectly (but perhaps plausibly) colored large sections of the image, such as uniformly colored backgrounds.

Interpreting these results is not straight-forward, perhaps with the exception of the PA metric. In the PA column of Table 8.1, we can clearly see that all classification variants produce significantly more artifacts compared to simple ab prediction. Rather than the output format, we attribute this to the loss function used during training, which, incidentally, is why the L_2 ab CNN variant performs quite well in the other two metrics as well, as it is trained to predict values more akin to the metric's calculation - in fact RMSE is closely related to the Euclidean distance loss function used during training.

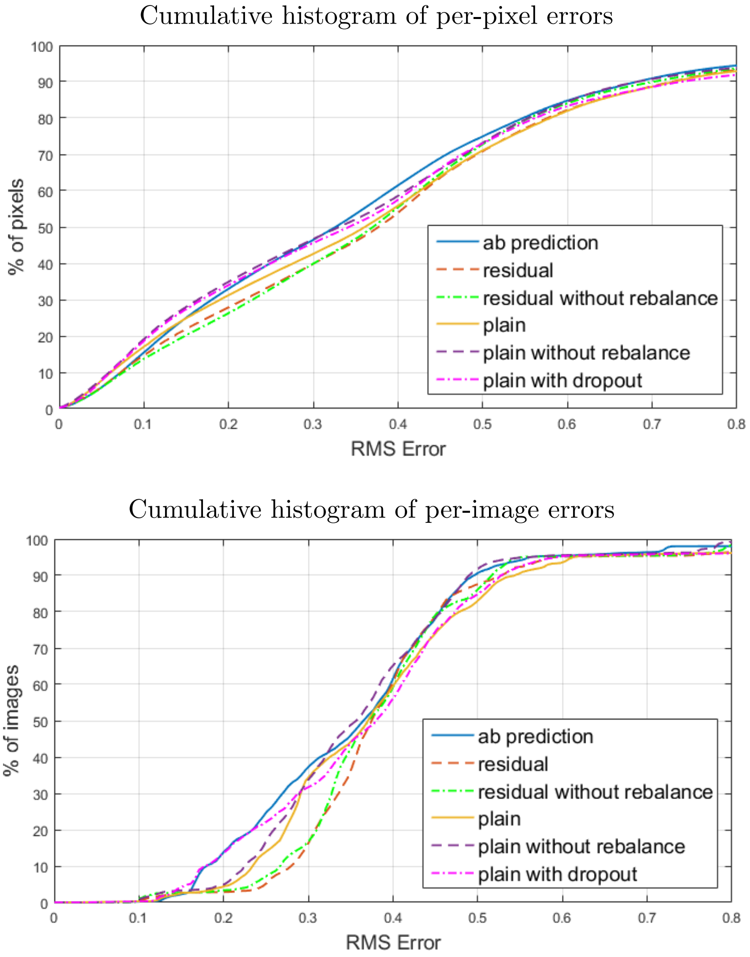


Figure 8.1: Cumulative histograms of per-pixel and per-image RMS errors. Variants based on the plain CNN tend to perform better, with L_2 ab CNN model performing best.

8.2 Visual comparison of variants

It is worth noting that while the above metrics provide a quantifiable result, they fail to capture the semantic intricacies of particular colorizations, and therefore, it is necessary to analyze the results quantitatively too. This is rather impossible to do rigorously, and conclusions can be considered subjective or influenced by personal tastes. When comparing the models, we try to provide this analysis using as many objectively observable features as possible, but it is inevitable that some perceptions will be affected by opinions. Some researchers choose to offload this problem by carrying out a two-alternative survey of real vs. fake test [32], showing each image for a brief moment to the participants, however, it would be highly impractical when comparing multiple variants.

For the purposes of comparison, we processed the testing episode described in Chapter 4 and an extra episode of the same cartoon, which has not been newly hand-recolored (specifically, we choose the 25th episode, called *"O vodnickém kolovrátku"*), in hopes of observing generalization properties of each variant. We refer to this extra episode as *secondary* dataset.

Here we present several examples of colorized images to demonstrate the basic properties of the colorization implemented by the network variants, as well as multiple examples which show commonly seen shortcomings of the colorization method. Full results, including episode sequences turned into video files, are available on the attached CD described in Appendix A. More hand-picked results can be found in Appendix B.

In some cases, our models create colorizations that are semantically plausible and while it would be far-fetched to say that it is hard to distinguish the results from hand-made colorization, they are far from trivial, exhibiting signs of semantic understanding of the scene. These images usually feature characters or objects that are uniformly colored in the training dataset (such as the bandit's hat in the last image of Figure 8.4).

Not all variants perform consistently on the whole testing set, in fact, it is common for each variant to perform well in some areas, while being less successful in others, even though there are examples where none of the variants produce particularly good looking results. Observing these differences can be quite interesting, see Figures 8.3, 8.4 and 8.5 for comparisons of all variants on selected examples.

8.2.1 Forms of failure

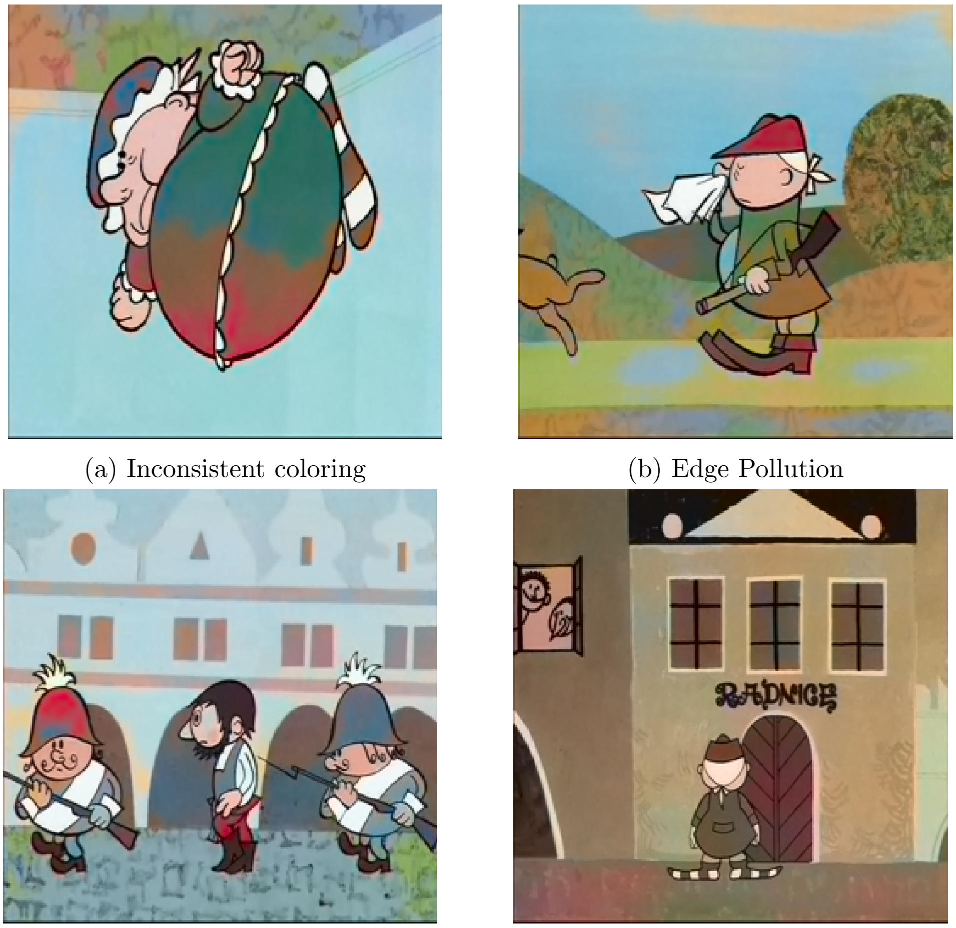
To properly asses strengths and weaknesses of each variant, we first describe commonly seen forms of failure. Some failures only cause some parts of the image to become implausible and others can, in some cases, cause the whole image to be perceived as badly colorized. While some objects may still be colored naturally, other parts of the image contain problematic areas.

The failures are most commonly observable on backgrounds, as they tend to have limited (often none at all) texture information, span large portions of the image and appear colored with a number of different colors in the training set, with limited cues for correct colorization (e.g. a background appears pink or blue for wall or sky respectively, with no indication of whether the current scene happens indoors).

Observed forms of failure appearing in most variant's results, include the following cases, examples are shown in Figure 8.2:

- **inconsistent coloring** - objects contain spatially inconsistent colors, such as color shifts or artifacts
- **edge pollution** - inconsistent color around the edges of the image
- **color bleeding** - object's color boundary spills over its intended edges
- **color deficiency** - colors appear too desaturated or too uniform across the entire image

Some of the variants also show specific failures, such as sensitivity to luma fading, producing extremely different colorizations when the same scene is fading in or out. Many images combine more than one of these failures, but usually, one is most noticeable.



(a) Inconsistent coloring

(b) Edge Pollution

(c) Color Bleeding

(d) Color Deficiency

Figure 8.2: Forms of failure commonly seen in colorized results.

8.2.2 Variant comparison

Here we list each variant's general colorization properties, based on the observed colorization of the testing set and secondary data. These observations are therefore hopefully representative of their generalization abilities, given how the testing set is selected.

Plain CNN - generally produces good colorization, with fewer artifacts in backgrounds, but more in smaller objects like character faces or clothing. It tends to use more vibrant colors, but suffers from color bleeding and inconsistent coloring in certain cases.

Plain CNN without rebalancing - compared to the version with rebalancing, creates bad artifacts in backgrounds, in some cases almost splits backgrounds into several sections, all colored differently. It seems to also have problems with small objects, but generally produces more consistent colorizations of mid-sized objects, such as fences or doors.

However, it produces noticeable edge pollution effects in most cases. Somewhat surprisingly, it uses colors qualitatively only a little dimmer than the version with rebalancing, rather than more desaturated colors that could be expected.

Plain CNN with dropout - qualitatively, compared to the variant trained without the dropout layers, the colorization is slightly worse in almost all aspects, with more inconsistent coloring of the backgrounds and smaller objects. Some edge pollution is also apparent. Since the problems are generally worse counterparts of the failures produced by the plain CNN, we conclude that the plain CNN does not suffer from overfitting on the training data too much.

Plain $\mathbf{L_2}$ ab CNN - the simpler L_2 ab prediction produces mostly consistently colored images (as indicated by the PA metric result), with fewer images with noticeable edge pollution or color bleeding, but uses much less vibrant colors (color deficiency), which is to be expected, given the Euclidean loss function. The images appear plausibly colored, but dull or feint in color, except in the most prevalent objects (bandit's hat, bandit's face), which retain their lively colors.

Residual CNN - compared to the plain CNN variants, the residual CNN produces visibly more color inconsistencies in the backgrounds of images, but succeeds at colorizing smaller objects, such as character hands (which are problematic for all plain CNN variants, for example) or character clothing (unless too large).

This could be explained by having smaller ERF than the plain model, and therefore encoding more information about smaller objects, even with less overall parameters available and in fewer convolutional layers.

In terms of color usage, it is similar to the plain model, since rebalancing is used in both variants. While the plain model is more prone to using dimmer colors for smaller objects and more vibrant ones for larger objects or backgrounds, the residual model seems works the other way around.

Residual CNN without rebalancing - Removing the rebalancing term of the loss function has similar effects as with the plain CNN model, except the color shift towards dimmer colors is more pronounced. It also suffers more from color bleeding than residual CNN with rebalancing and from more edge pollution than residual CNN.

These traits can be largely observed on the selected examples shown in Figures 8.3, 8.4 and 8.5 as well as on the additional selected examples in Appendix B. The inconsistently colored background problem is the most common one, but edge pollution can be seen on the images produced by residual CNN without rebalancing in Figure 8.4. Interestingly, compared to plain CNNs, residual variants learn different background colorization properties regardless of the usage of rebalancing, as can be observed on both cases shown in Figure 8.5 (blue taint on the rocks, green inconsistency on the wall).

As a summary, none of the variants work perfectly. They each produce different colorizations, with different strong and weak points. It would highly subjective to determine which colorization is the most aesthetically pleasing - though plain CNN with dropout and residual CNN without rebalancing produce visually weaker colorization when the entire testing and secondary datasets are considered.

All of the models tend to overuse shades of blue, possibly due to overrepresentation of sky-like backgrounds in the training set. For the most part, the number of different colors that the models predict is somewhat limited to shades of green, light blue, varying hues of brown, dark red and beige. We assume that even when rebalancing is used, the rarer colors seen in the training set (light red, pink, dark blue or light yellow) are not represented enough to be predicted unless the object is very specific.

In images where texture information is present (e.g. some foliage for grass, clouds in the sky or darker strokes on tree trunks), the colorization tends to be much better across all variants. This is consistent with our hypothesis made in Section 4.2.

Concerning the secondary data, there are some points that are common to all the variants; compared to the outcomes on the testing set, colorizing the secondary data results are less vibrant and generally have more faded colors, but objects show tendency to be consistently colored more often. Common objects retain their color even on this set however, implying well learned insensitivity to noise and luminance shifts. Surprisingly, residual CNN model seems to produce better looking results than the other variants, perhaps due to the fact that one of its weakest points (coloring backgrounds) becomes the weak point of all variants.

Generally, the models trained with rebalancing produce more consistent colorizations with better-looking colors than the models trained without rebalancing or the model using Euclidean loss, which tends to use desaturated colors.

In Section 8.4, we explore the possibilities of improving the visual result through the usage of additional algorithms.

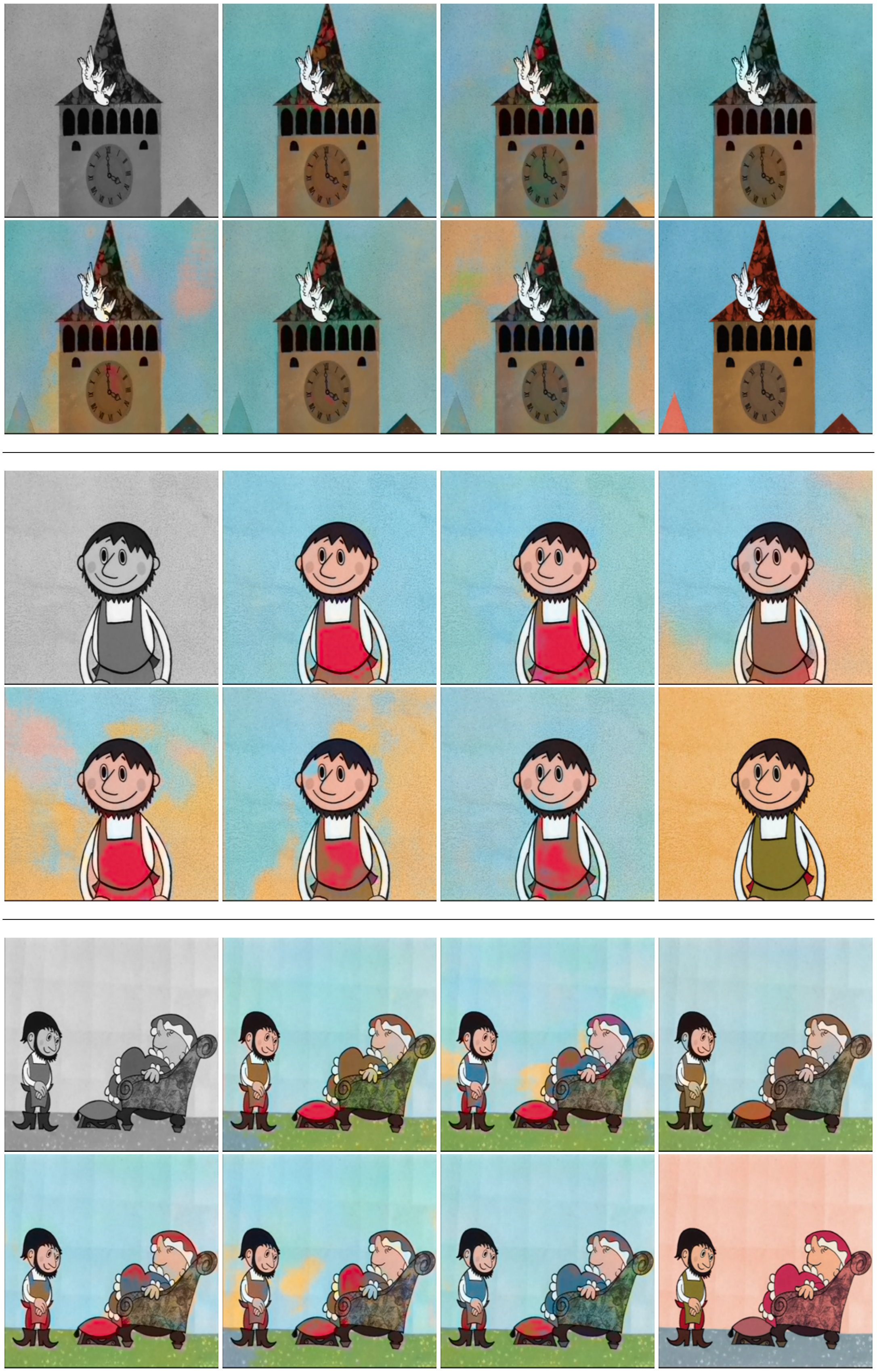


Figure 8.3: Colorization example results on testing set, from upper left to lower right: input, plain CNN, residual CNN, L_2 ab CNN, plain CNN with dropout, plain CNN without rebalancing, residual CNN without rebalancing, ground truth

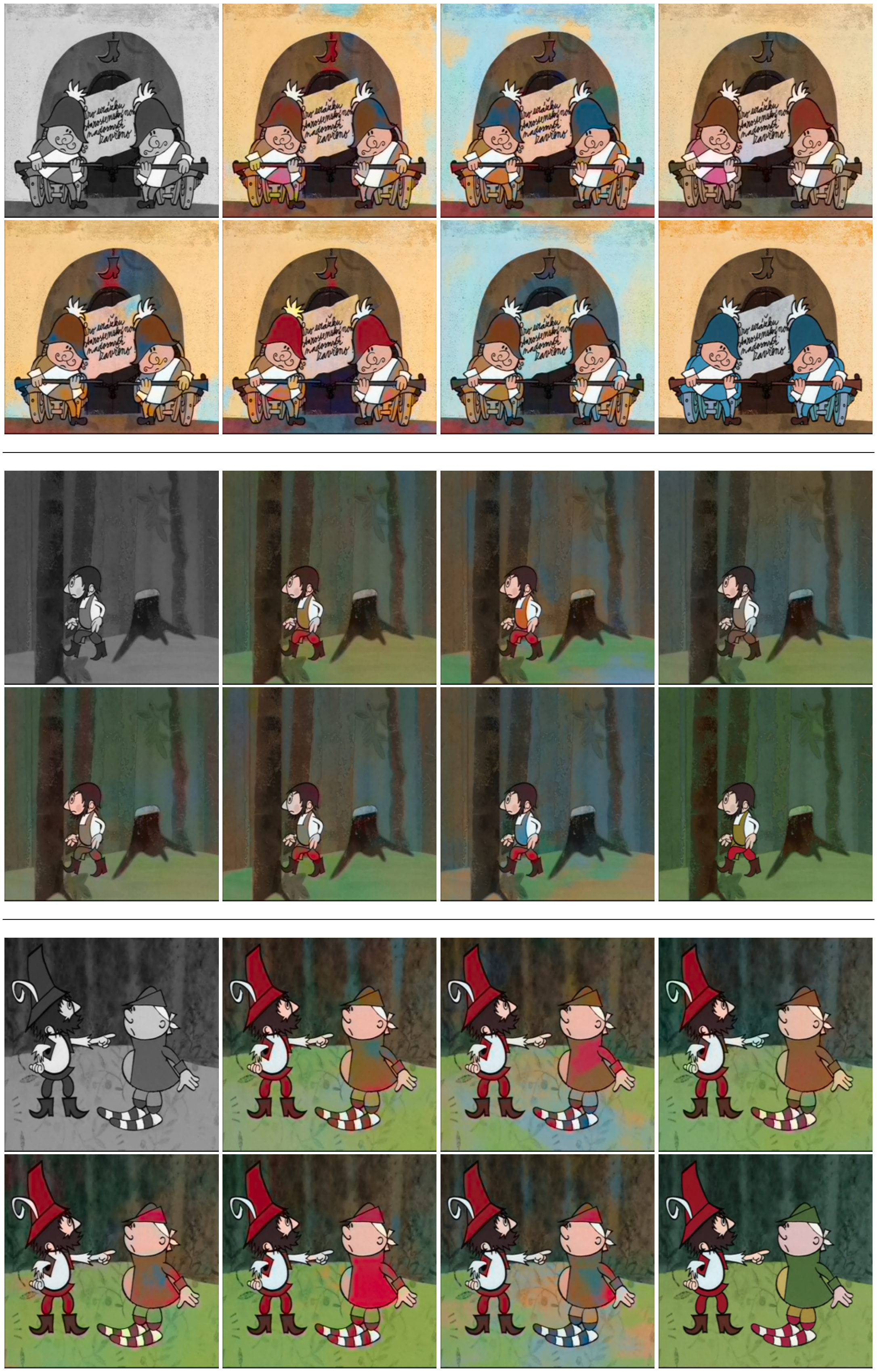


Figure 8.4: Colorization example results on testing set, from upper left to lower right: input, plain CNN, residual CNN, L_2 ab CNN, plain CNN with dropout, plain CNN without rebalancing, residual CNN without rebalancing, ground truth

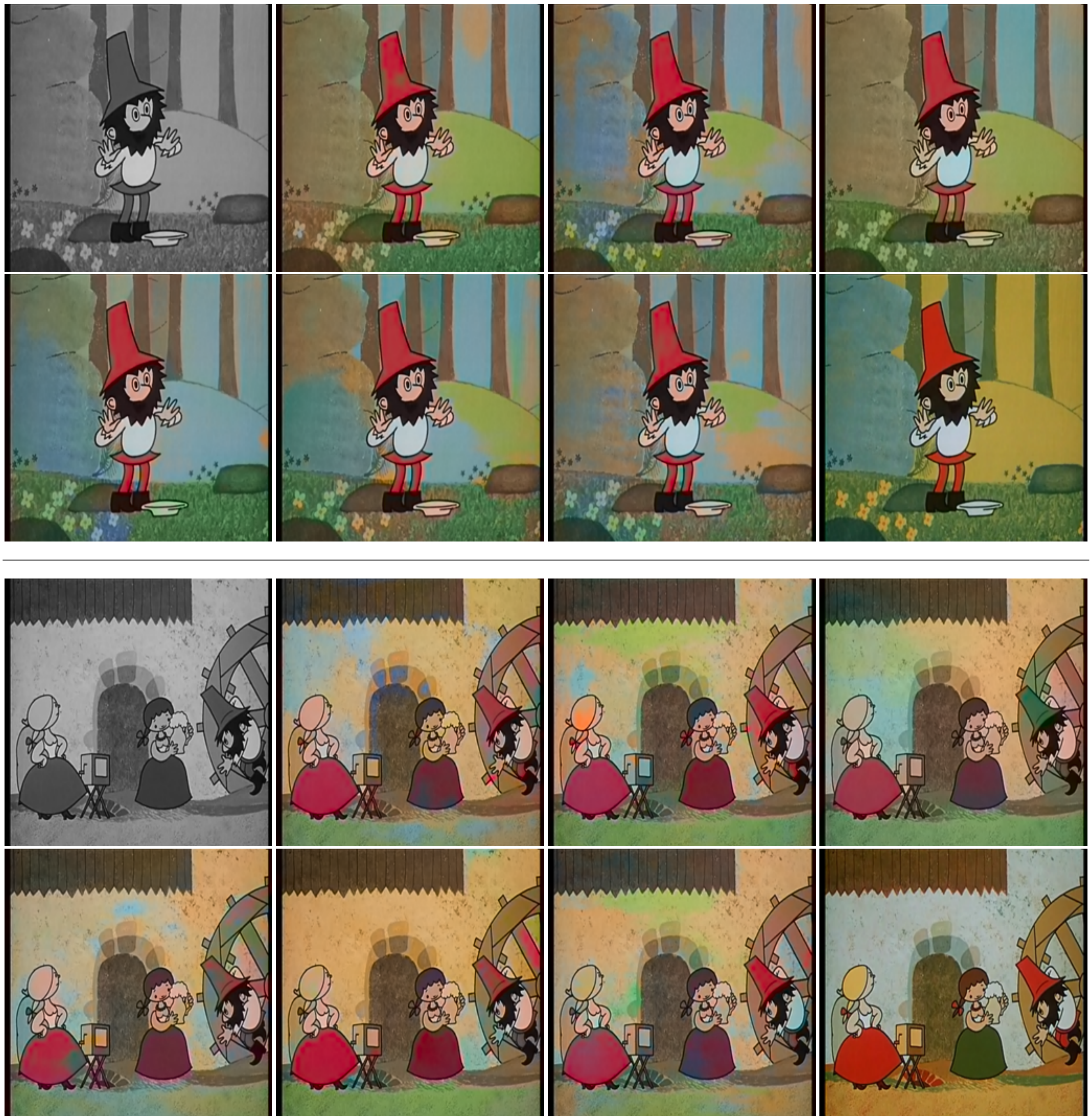


Figure 8.5: Colorization example results on secondary data, from upper left to lower right: input, plain CNN, residual CNN, L_2 ab CNN, plain CNN with dropout, plain CNN without rebalancing, residual CNN without rebalancing, ground truth

8.3 Comparison to color transfer methods

To fully understand the performance of our models, we attempt to compare our models with current color transfer methods.

Comparing algorithm performance can be quite unfair however, since most color transfer methods were designed to work on natural images and do not specifically deal with the difficulties outlined in Section 4.2. Nevertheless, we offer comparison of our fully automated method to two selected algorithms of Welsh et al. [28] and Deshpande et al. [27], which are also used for comparison by Zhang et al. and Larsson et al. [32], [33] and their source codes are available online.

See Figure 8.6 for three image examples colored by the different methods. On all tested images, our method visually outperforms color transferring algorithms, but the plausibility of the colorizations created by the color transferring methods is highly dependent on the chosen references. Every method produces results with artifacts, but the color transferring methods create much less visually consistent colorizations.

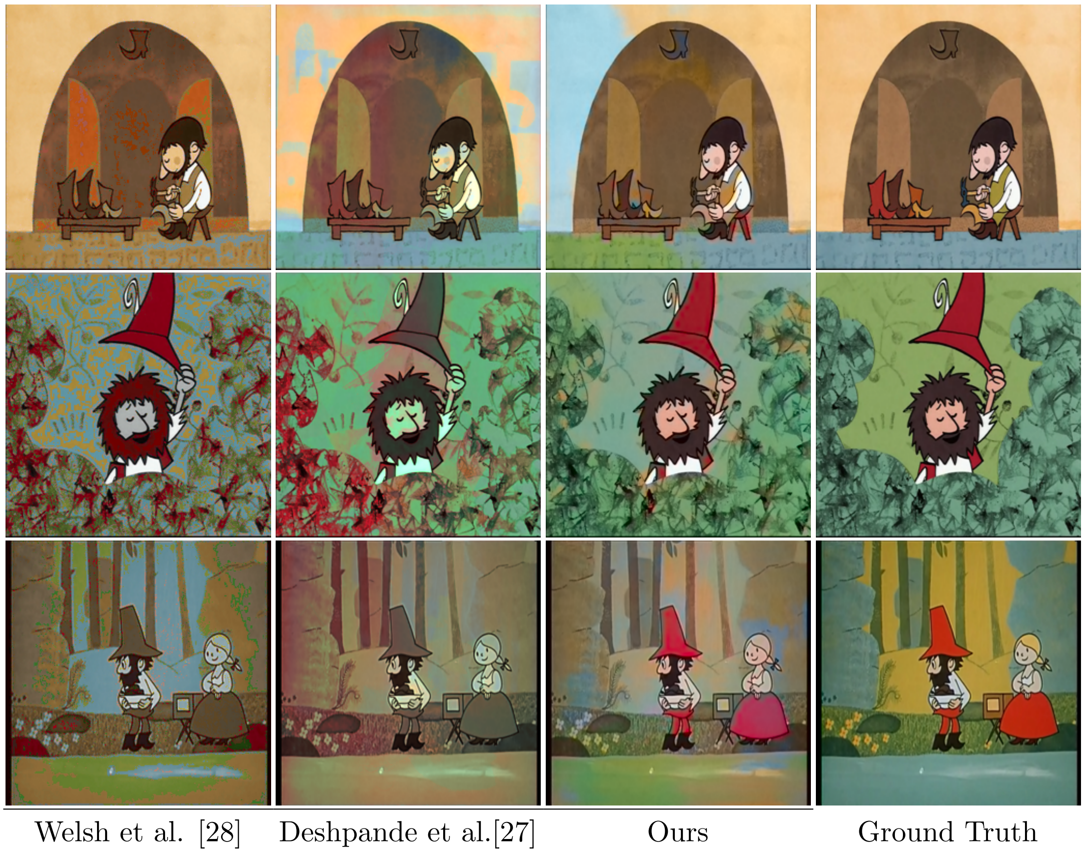


Figure 8.6: Side-by-side comparison of color transfer colorization methods to our results. In case of Deshpande's algorithm, best looking results is chosen (either using GT histogram or not), in our column, the visually best looking result is chosen - Plain CNN without rebalance, Plain CNN and Residual CNN for the three images respectively.

Reference images for Welsh's algorithm were hand-selected from the training set to match the colorized image as closely as possible (semantically and color-wise), Deshpande's algorithm was trained on 11 hand-selected example-with-5-matches sets (Deshpande et al. demonstrate reasonable performance with as few as 5 training sets) and colorized images were supplied with 5 matching images chosen from the training set. The particular selection of reference and training images can be found in the supplementary material described in Appendix A.

Additionally, on the input size of 256×256, running Welsh's color transfer took 35.3 seconds on average and Deshpande's transfer required 174 seconds per image, which is quite prohibitive, considering our method needed only 74 ms on average to compute full colorization of an image on Intel Core i7-4790k CPU and NVIDIA GeForce GTX 970 GPU, turning making colorization results into an afterthought. This is particularly important for applying the colorization to video sequences.

8.4 Possible refinements

After obtaining the colorization results, given the unique properties of the dataset, there are improvements that can be made in order to make the colorization look more akin to ground truth's style. One option would be to produce scribble images by treating the colorization produced by the CNNs as the manual input required by other, traditionally human assisted algorithms, described in Section 3.1. This, in turn, would make the algorithms fully automatic, all the while preserving the various desirable properties that they have otherwise been designed to have. We briefly experimented with this idea, for example, by sampling single-pixel scribbles in areas of locally maximal classification confidence (spatial peaks in predicted color histograms), but found that no commonly used scribble algorithms work well on such input. Instead, we propose the following methods.

8.4.1 Segmentation with flood fill

The first refinement that we experiment with is one that attempts to make the colorization more uniform, given that the ground truth images are colored very uniformly (e.g. no blurring around edges or soft transitions of colors) and contain limited textures and large regions that should be colored evenly.

After observing that edges are very clearly defined, we can do this by automatically segmenting the grayscale input image into disjoint areas using a seeded 4-neighborhood region growing mean-shift segmentation algorithm with empirically selected threshold value (in particular it was $\frac{3}{51}$ for the normalized L range of $[0, 1]$, which worked consistently well across the whole set, though an inferred threshold value could improve the result further, as in some images smaller or larger values provide more sensible segmentations). For thresholding to be more effective on images with varying luminance (mostly fading in/out of scenes), the L channel is normalized into a constant range on every image.

Seed locations are chosen iteratively at pixel positions with largest values in distance matrix of a binary image consisting of already found regions, with first seed located at (0, 0). The region growing segmentation continues until there are no pixels with distance greater than 4 from any segmented region (chosen empirically).

This gives us areas which should be evenly colored. In each of these areas, we take the ab values of all pixels and perform cluster analysis using k-means with $k = 2$, based on the observation that artifacts and color shifts produced by our models rarely exceed single color deviations. Then, the cluster with largest magnitude is selected, and its centroid is used as the ab value for each pixel located in the area.

For a more rigorous analysis, k could be automatically selected by the elbow method, taking the point with largest absolute second derivative value, but doing so would result in significantly increased computation time with little benefit, making the process less viable for video sequences. While the choice of $k = 2$ might be arbitrary, the method still creates results that behave much better than when modus (changes across images are too sharp) or mean (colors are too desaturated and tend to blue) are used. We can see example results in Figure 8.7. This refinement mostly removes the effects of color bleeding and inconsistent coloring.

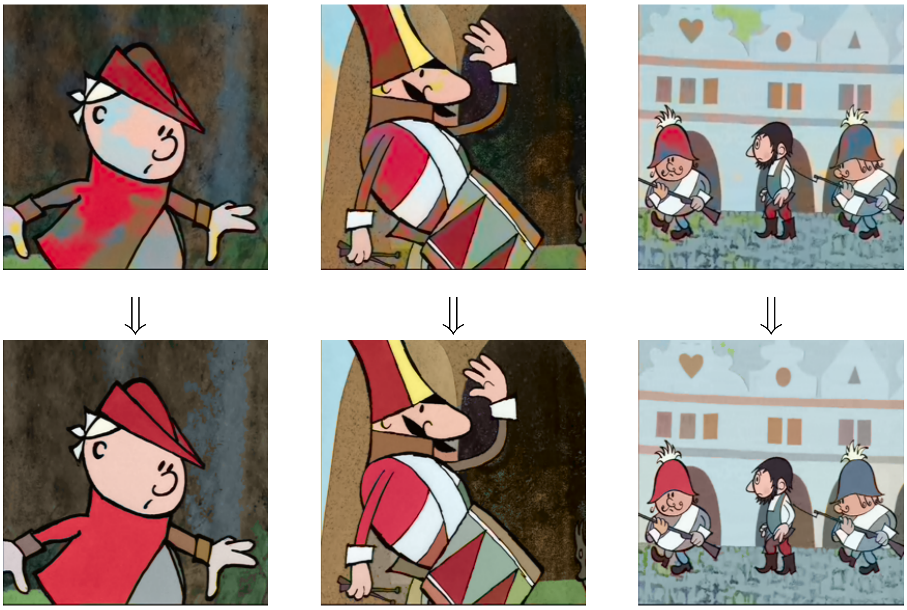


Figure 8.7: Example results of the automatic segmentation with flood fill algorithm. Upper images are the colorizations produced by CNN models, lower row contains images after running the algorithm.

Using unsupervised graph-based image segmentation, such as the algorithm by Felzenszwalb and Huttenlocher [42] instead of region growing is also an option, and we find it to produce better results on the secondary data - as the object boundary edges can be more subtle, region growing tends to spill over intended object edges or produces too many small areas when the threshold value is decreased due to random noise and fine textures.

Full results applied to all variants on both datasets can be found on the attached CD described in Appendix A.

8.4.2 Ensemble as mean

By having results from multiple variants of our models, we can employ ensemble learning - combining the different results into one prediction. One way to do that is to take the mean value of pixel colors across all variants (either in RGB or Lab, but since the luminance is the same in all of the images, the choice is mostly irrelevant here). We generate the results of this operation to demonstrate the properties of such combination. The fundamental idea behind this method is that inconsistently colored regions will cancel out, while consistently colored areas will stay mostly unaffected, even if a small number of models fails to produce stable colorization of them.

Of course, this method inherently suffers from the same problems as using L_2 loss function - averaged multimodal predictions will drive the chrominance values towards zero, producing more desaturated images. This effect can be observed on the example shown in Figure 8.8, however, with the small number of variants that predict similar colors, the effect is not as pronounced, and actual results are more satisfactory than individual model's results.

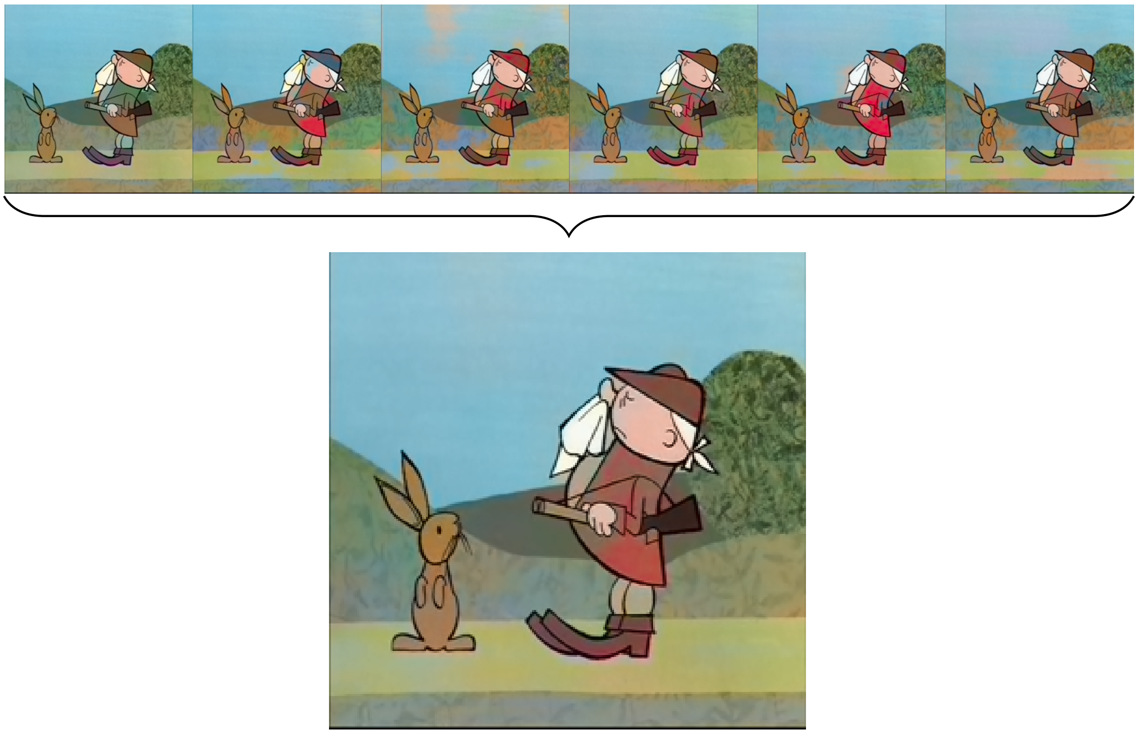


Figure 8.8: Example of mean image of all trained variants. Failures are averaged out at the cost of faded colors, while consistently colored regions remain well colored.

Full results can again be found on the attached CD described in Appendix A.

Another way to accomplish ensemble learning would be to combine the color histograms predicted by the networks directly, for example by choosing the value with highest confidence across all variants (essentially choosing max instead of mean), however, this invalidates the results of the L_2 ab CNN variant, as it does not output a color histogram for each pixel.

8.5 Applicability in video

So far, we have only described results concerning colorization on single images. However, since the original data comes from video sequences, it comes naturally that the ultimate goal should be to colorize a full image sequence plausibly.

Unfortunately, many problems only manifest when the images are put into a sequence - such as the fact that most objects tend to change color with changing scale or object rotation (translation is affected to a lesser degree, due to translational invariance of convolutional layers combined with using small filter sizes). This effect can also be observed with occlusion, although it is not as severe. These problems imply that the models have not been able to fully learn rotation or scale invariance.

Another problem that emerges is that one object's proximity can change the coloring of an unrelated object. This effect is shown in Figure 8.9, where two subsequent frames are shown. Notice that the only thing changing between the frames is the position of the doves, but this inadvertently affects the colorization of the clock face, making it inconsistent.

This is not limited to cases where occlusion happens, and it is most commonly seen on backgrounds.

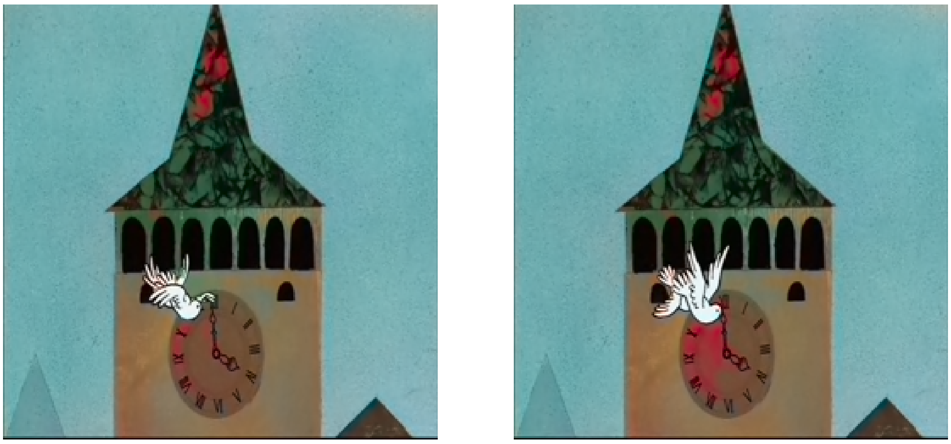


Figure 8.9: Two subsequent frames showing the effect of the object proximity failure.

These problems are mostly prevalent among all model variants, with no clear improvements or deteriorations between the variants. It leads to strange looking sequences, with color of objects changing as they get closer or further away from the camera or are rotated slightly.

The sequences look better when generated from images with refinements applied, but while the color fill augmentation produces much better looking colorizations on single images, in sequences it creates unpleasant color flickering between images in cases where the cluster sizes change their ordering (where inconsistent coloring is too severe). Using ensemble mean images, the results become more consistent, with no flickering and evenly colored objects, but the colors look dimmer and are generally not as vibrant overall except for objects that are very prevalent in the dataset (hats, faces, grass, ..). Despite this disadvantage, we think that videos produced from mean images of the segmented and color filled images compose into the best looking video, compared to all the to the other options.

In summary, this is where the method falls short - perhaps due to the difficulty of the training set (or rather its small size), moderate changes in object scale or rotation or even object's position relative to another can result in sweeping changes of the colorization, producing noticeable temporal color shifts. Though the problems could be alleviated with temporal filtering, selection of keyframes would require manual input and alterations of the colorizations to make them more consistent with one another.

The full resulting video sequences are available on the attached CD, as described in Appendix A.

Chapter 9

Conclusion and future work

We presented a method of fully automatic colorization of unique grayscale cartoon images combining state-of-the-art CNN techniques. Using the right loss function and color representation, we have shown that the method is capable of producing a plausible and vibrant colorization of certain parts of individual images even when applied to a moderately sized dataset that has properties which make it harder to colorize than natural images, but does not perform as well when applied to video sequences.

In doing so, we visually and quantifiably compared several variants of CNN design, which differed in loss functions, architectures and regularization methods. It is clear that the models we used have a hard time learning colorization of large uniform regions such as background sky or walls but fare better when smaller objects and characters are present.

We also proposed two methods of improving the generated results which greatly increase the visual resemblance of generated colorization to the ground truth images.

One novel contribution is using and comparing a model inspired by residual CNNs for the task of colorization and showing that despite the smaller ERF and fewer parameters, it can generate results that are comparable or even surpass plain convolutional neural networks in generalization to unseen data.

9.1 Future work

In order to be applicable for video, the method would currently require further refinement, performed manually by an artist. If trained on a larger dataset, the predictive power of the model would increase and is likely to produce more consistent colorization.

For future work, it would be interesting to compare colorization produced by ResNet models with significantly more depth (which require more computational resources to train) and models based on conditional generative adversarial networks, as the results they have been able to put together when applied to natural images are quite impressive and allow the user to have more control over the result by adjusting the latent space variable.

Additionally, the CNN model could be adjusted to generate scribbles to use in conjunction with the algorithms we mentioned in Chapter 3, instead of full colorization. This could lead to results that more closely match the currently used colorization methods that apply color to cartoon movies.

Bibliography

[1] V. Bochko, P. Välisuc, T. Alho, S. Sutlnen, J. Parkkinen, and J. Alander, "Medical image colorization using learning", pp. 70–74, Jan. 2010.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[4] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing", in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Université de Rennes 1, La Baule (France): Suvisoft, Oct. 2006.

[5] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning", *CoRR*, vol. abs/1705.03122, 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", *CoRR*, vol. abs/1512.03385, 2015.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1", in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., Cambridge, MA, USA: MIT Press, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362, ISBN: 0-262-68053-X.

[8] S. Iizuka, E. Simo-Serra, and H. Ishikawa, "Let there be color!: Joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification", *ACM Trans. Graph.*, vol. 35, no. 4, 110:1–110:11, Jul. 2016, ISSN: 0730-0301.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", *CoRR*, vol. abs/1409.4842, 2014.

[10] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions", *CoRR*, vol. abs/1511.07122, 2015.

[11] W. Luo, Y. Li, R. Urtasun, and R. S. Zemel, "Understanding the effective receptive field in deep convolutional neural networks", *CoRR*, vol. abs/1701.04128, 2017.

[12] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, ISSN: 0028-0836.

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting”, *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[14] M. Li, T. Zhang, Y. Chen, and A. J. Smola, “Efficient mini-batch training for stochastic optimization”, in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’14, New York, New York, USA: ACM, 2014, pp. 661–670, ISBN: 978-1-4503-2956-9.

[15] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, *CoRR*, vol. abs/1502.03167, 2015.

[16] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks”, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[17] P. Krähenbühl, C. Doersch, J. Donahue, and T. Darrell, “Data-dependent initializations of convolutional neural networks”, *CoRR*, vol. abs/1511.06856, 2015.

[18] R. S. Sutton, “Two problems with backpropagation and other steepest-descent learning procedures for networks”, in *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Erlbaum, 1986.

[19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization”, *CoRR*, vol. abs/1412.6980, 2014.

[20] K. Kapoor and S. Arora, “Colour image enhancement based on histogram equalization”, *Electrical & Computer Engineering: An International Journal*, vol. 4, pp. 73–82, Sep. 2015.

[21] A. Levin, D. Lischinski, and Y. Weiss, “Colorization using optimization”, in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH ’04, Los Angeles, California: ACM, 2004, pp. 689–694.

[22] Y.-C. Huang, Y.-S. Tung, J.-C. Chen, S.-W. Wang, and J.-L. Wu, “An adaptive edge detection based colorization algorithm and its applications”, in *Proceedings of the 13th Annual ACM International Conference on Multimedia*, ser. MULTIMEDIA ’05, Hilton, Singapore: ACM, 2005, pp. 351–354, ISBN: 1-59593-044-2.

[23] Q. Luan, F. Wen, D. Cohen-Or, L. Liang, Y.-Q. Xu, and H.-Y. Shum, “Natural image colorization”, in *Eurographics Conference on Rendering Techniques*, Jan. 2007, pp. 309–320.

[24] Y. Qu, T.-T. Wong, and P.-A. Heng, “Manga colorization”, in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH ’06, Boston, Massachusetts: ACM, 2006, pp. 1214–1220, ISBN: 1-59593-364-6.

[25] D. Sýkora, J. Dingliana, and S. Collins, “LazyBrush: Flexible painting tool for hand-drawn cartoons”, *Computer Graphics Forum*, vol. 28, no. 2, pp. 599–608, 2009.

[26] X. Liu, L. Wan, Y. Qu, T.-T. Wong, S. Lin, C.-S. Leung, and P.-A. Heng, “Intrinsic colorization”, *ACM Trans. Graph.*, vol. 27, no. 5, 152:1–152:9, Dec. 2008, ISSN: 0730-0301.

[27] A. Deshpande, J. Rock, and D. Forsyth, “Learning large-scale automatic image colorization”, in *Conference: Conference: 2015 IEEE International Conference on Computer Vision*, Dec. 2015, pp. 567–575.

[28] T. Welsh, M. Ashikhmin, and K. Mueller, “Transferring color to greyscale images”, *ACM Trans. Graph.*, vol. 21, no. 3, pp. 277–280, Jul. 2002, ISSN: 0730-0301.

[29] R. K. Gupta, A. Y.-S. Chia, D. Rajan, E. S. Ng, and H. Zhiyong, “Image colorization using similar images”, in *Proceedings of the 20th ACM International Conference on Multimedia*, ser. MM ’12, Nara, Japan: ACM, 2012, pp. 369–378, ISBN: 978-1-4503-1089-5.

[30] A. Y.-S. Chia, S. Zhuo, R. K. Gupta, Y.-W. Tai, S.-Y. Cho, P. Tan, and S. Lin, “Semantic colorization with internet images”, in *Proceedings of the 2011 SIGGRAPH Asia Conference*, ser. SA ’11, Hong Kong, China: ACM, 2011, 156:1–156:8, ISBN: 978-1-4503-0807-6.

[31] Z. Cheng, Q. Yang, and B. Sheng, “Deep colorization”, *CoRR*, vol. abs/1605.00075, 2016.

[32] R. Zhang, P. Isola, and A. A. Efros, “Colorful image colorization”, *CoRR*, vol. abs/1603.08511, 2016.

[33] G. Larsson, M. Maire, and G. Shakhnarovich, “Learning representations for automatic colorization”, *CoRR*, vol. abs/1603.06668, 2016.

[34] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training gans”, *CoRR*, vol. abs/1606.03498, 2016.

[35] Y. Cao, Z. Zhou, W. Zhang, and Y. Yu, “Unsupervised diverse colorization via generative adversarial networks”, *CoRR*, vol. abs/1702.06674, 2017.

[36] Q. Fu, W.-T. Hsu, and M.-H. Yang, “Colorization using convnet and gan”, 2017.

[37] D. Ruderman, T. W. Cronin, and C.-C. Chiao, “Statistics of cone responses to natural images: Implications for visual coding”, vol. 15, pp. 2036–2045, Aug. 1998.

[38] R. K. Srivastava, K. Greff, and J. Schmidhuber, “Highway networks”, *CoRR*, vol. abs/1505.00387, 2015.

[39] ——, “Training very deep networks”, *CoRR*, vol. abs/1507.06228, 2015.

[40] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding”, *arXiv preprint arXiv:1408.5093*, 2014.

[41] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy: Open source scientific tools for Python*, Online at `http://www.scipy.org/`; accessed 29th December 2017, 2001–.

[42] P. F. Felzenszwalb and D. P. Huttenlocher, “Efficient graph-based image segmentation”, *Int. J. Comput. Vision*, vol. 59, no. 2, pp. 167–181, Sep. 2004, ISSN: 0920-5691.

Appendix A

Contents of the attached CD

The attached CD contains:

1. All network variants' definitions in prototxt files
2. All scripts used to create and process the dataset, all Matlab scripts used to generate videos from the images
3. All images used for training and testing, outputs of the networks, as well as the versions augmented by the algorithms described in Section 8.4. The images have been converted to JPEG format with 75% quality setting in order to reduce their overall size on the disk
4. All videos generated from these images, converted to MPEG (H.264) format to reduce file sizes
5. Caffe distribution compiled for Linux with custom layers for fast color space conversion and the loss function described in Section 5.3, including the custom Python layers mentioned in Chapter 7
6. Electronic version of this thesis

The model definitions can be found in the "networks" subdirectory, each model consists of three prototxt files, one for model definition used for training ("train.prototxt"), one for model definition used for generating results ("deploy.prototxt") and one containing the optimizer's settings used during training ("solver.prototxt"). Additionally, "weights.caffemodel" file contains the caffe's representation of a given network's learned weights, and Python script "process_images" is used to colorize images.

Directory "shared_resources" contains files which are common to all the models, most notably the used ab pairs definitions ("ab_pairs.npy"), prior probabilities of the pairs calculated on the training set ("rumcajs_train_priors.npy") and Python implementation of layers handling calculation of the rebalancing terms and conversion from ab value to color histogram in "layers.py".

All of the images are stored in the "images.7z" archive, which has a self-explanatory hierarchy. For each variant, there are at minimum directories containing all colorized images of the testing data, secondary data and at least one directory where the images have been refined by segmentation with flood fill, the full path follows structure of "images/<architecture>/<variant>/<secondary, testing>_<algorithm used for segmentation>", where algorithm can be either none, region growing (regiongrow) or Felzenszwalb segmentation (fs). Mean images and source data are placed on the architecture level.

Generated videos can be found in the "videos" directory, and they follow a similar naming structure. Additionally, a more comprehensive comparison video is included for both datasets, which shows the episode played with several colorizations side by side, "CompareVid_{Secondary, Testing}".

In the directory "scripts", all source code can be found, including code used to generate figures used in this thesis, Matlab functions for computing the error metrics, including mex implementation of the PA metric, Matlab functions for generating the refined results, including mex implementation of the region growing segmentation algorithm and scripts we used to filter the initial dataset as described in Sections 4.1.2 and 4.1.3.

Directory "scripts/3rd_party" contains source codes and reference image sets used to generate results shown in Section 8.3, as well as an implementation of Felzenszwalb segmentation.

Appendix B

Additional results

In this Appendix, additional colorization results are included. These results are generated both from the training set and the secondary dataset.

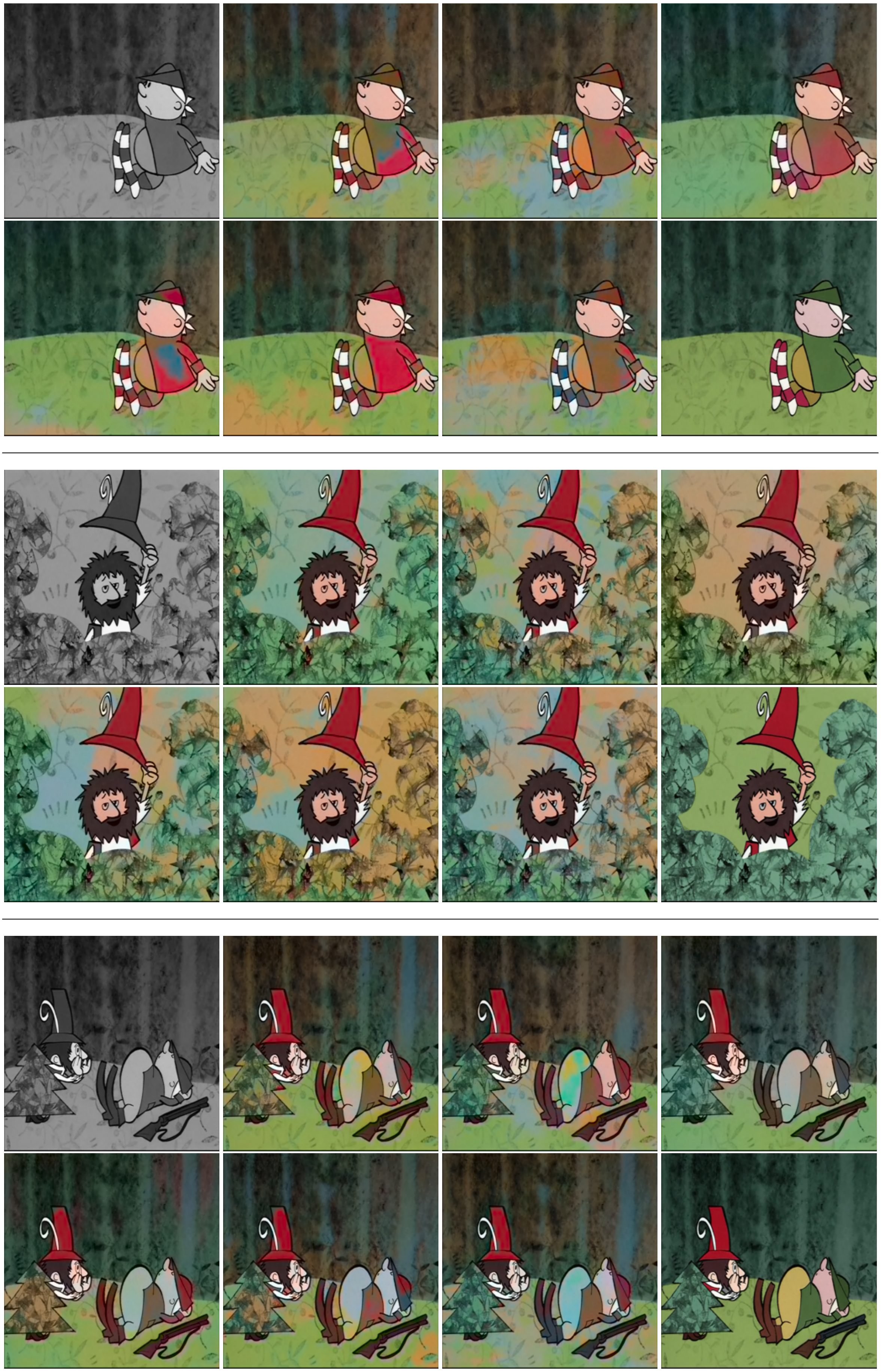


Figure B.1: Colorization results, from upper left to lower right: input, plain CNN, residual CNN, L_2 ab prediction, plain CNN with dropout, plain CNN without rebalancing, residual CNN without rebalancing, ground truth

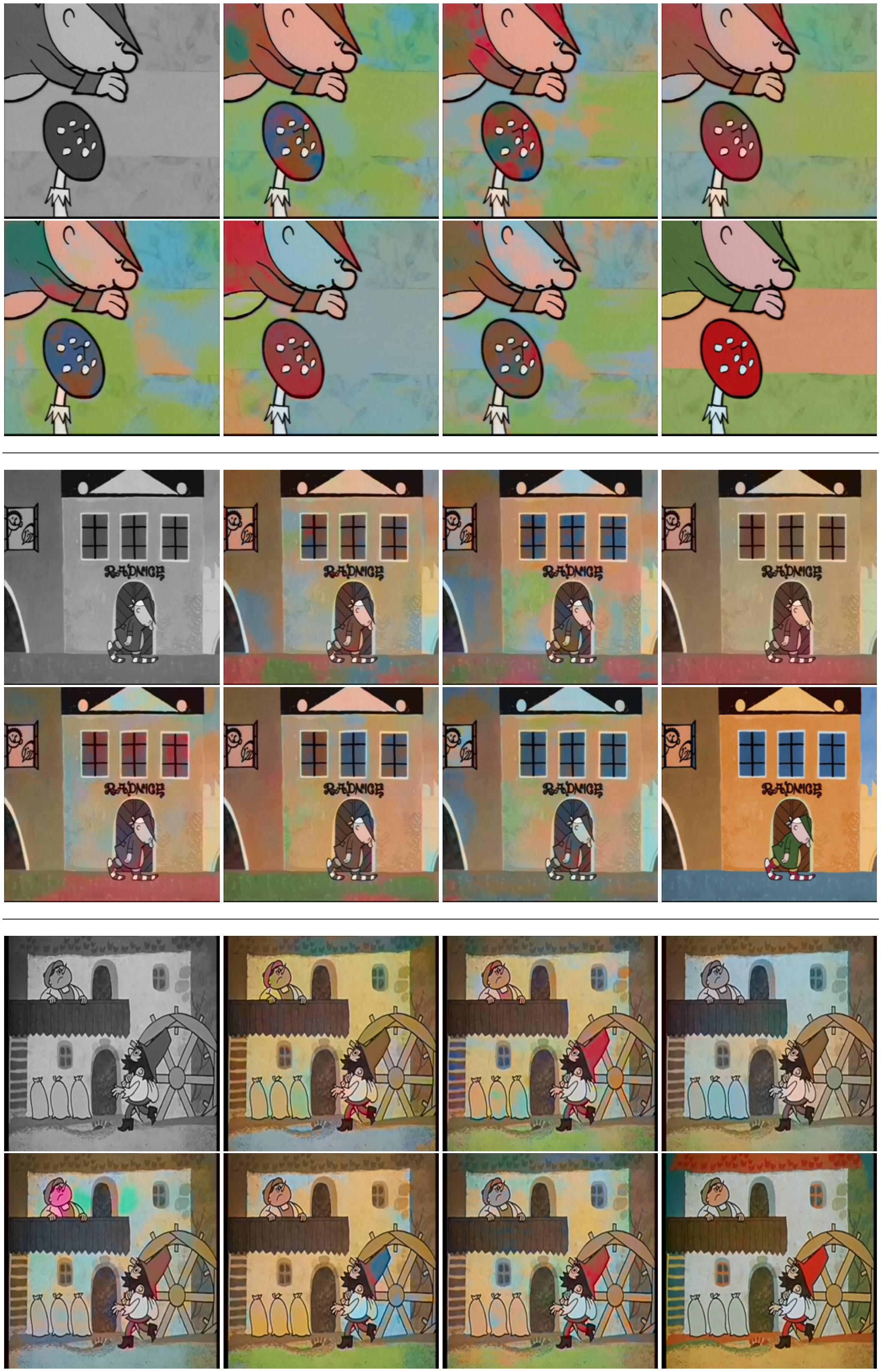


Figure B.2: Colorization results, from upper left to lower right: input, plain CNN, residual CNN, L_2 ab prediction, plain CNN with dropout, plain CNN without rebalancing, residual CNN without rebalancing, ground truth. Last image comes from the secondary dataset

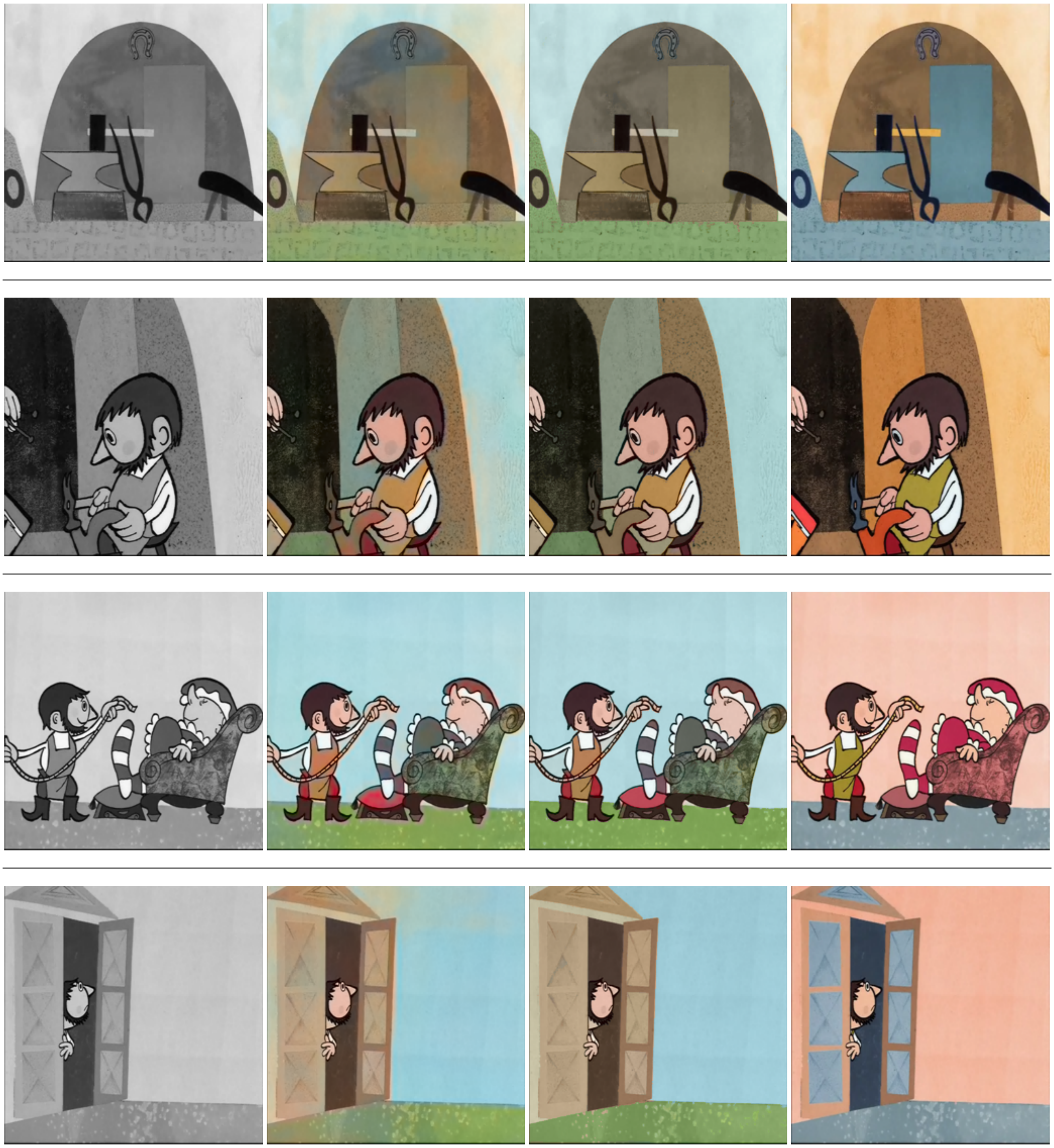


Figure B.3: Colorization results on testing data, from left to right: input, mean image of all variants, mean image produced from outputs of segmentation + flood fill for all variants, ground truth

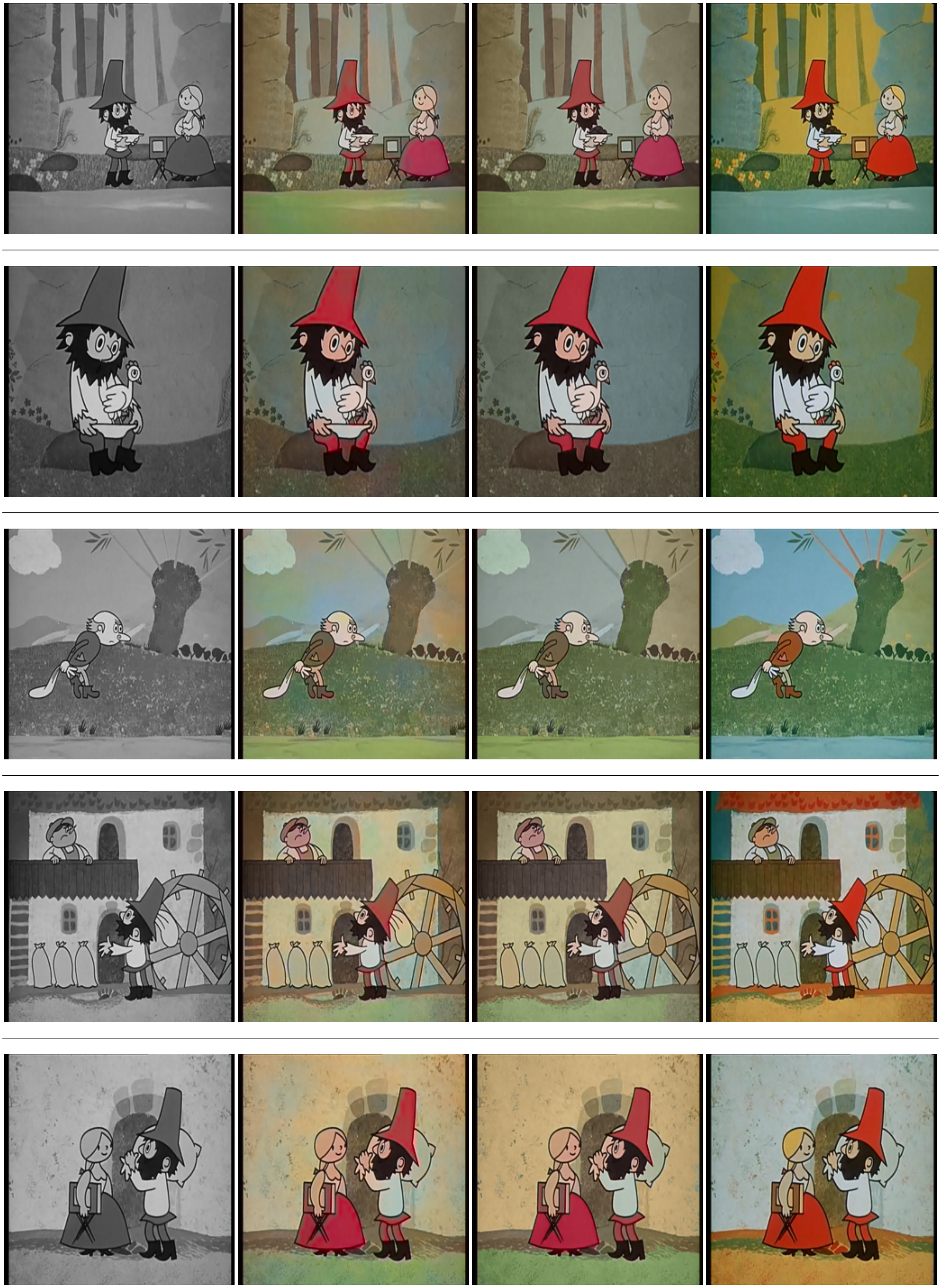


Figure B.4: Colorization results on secondary data, from left to right: input, mean image of all variants, mean image produced from outputs of Felzenszwalb segmentation + color fill for all variants, ground truth

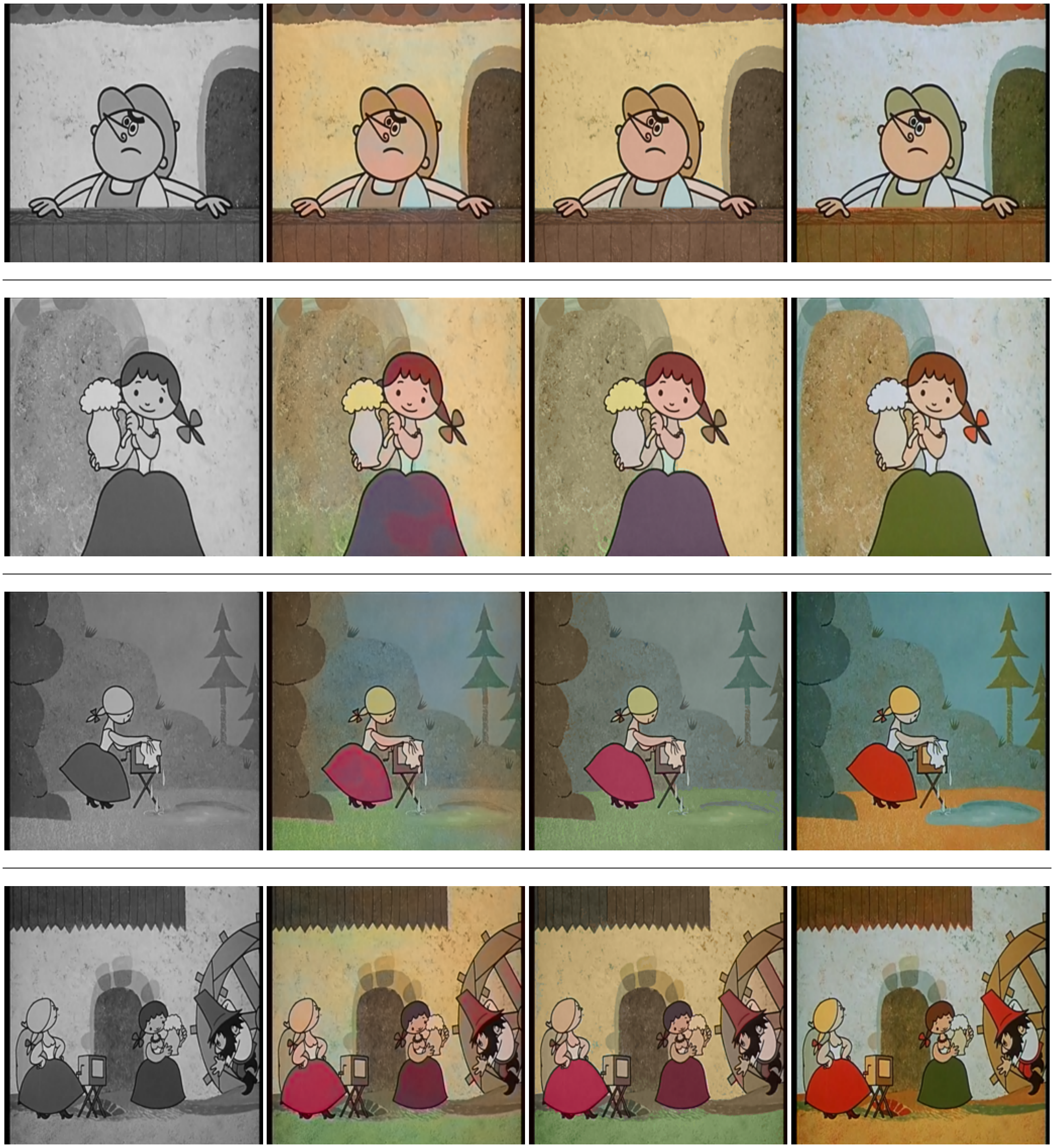


Figure B.5: Colorization results on secondary data, from left to right: input, mean image of all variants, mean image produced from outputs of Felzenszwalb segmentation + color fill for all variants, ground truth