Colorful Image Colorization with TensorFlow

Sophia Schulze-Weddige Malin Spaniol
Maren Born
Implementing Artificial Neural Networks with Tensorflow
Universität Osnabrück

April 19, 2020

Contents

1 Introduction

Based on the paper *Colorful Image Colorization* (Zhang et al., 2016), this project aims to reimplement a similar artificial neural network which transforms grayscale images into colorful pictures. This task involves creating a dataset based on pictures that are converted into the CIELAB colorspace (*Lab*), such that the lightness channel L can be considered as the input whereas the a and b channels, which encode color information, form the target for the model. We closely rebuild the layers of the original model (they used "caffe" (Jia et al., 2014)) using tensorflow 2.0. Our project is divided into two steps, which are also discussed in the paper (Zhang et al., 2016). In the first step, the unaltered ab channels are utilized as the prediction target, applying the mean squared error loss. As predicted by Zhang et al. (2016) this approach favors desaturated colors predicting images to appear sepia or greyish. This approach can be thought of as the "classical" way of automated image colorization with convolutional neural networks.
The second step aims to rebuild the main contribution to the image colorization problem from Zhang et al. (2016), namely to translate the problem to a classification task. By doing so, Zhang et al. (2016) were able to predict more plausible colors for the grayscale images. Due to the lack of time, we were not able to train our models long enough to reproduce the good results provided. More training time will hopefully lead to vivid and realistic colorizations for gray scale images with our model as well.

2 Theoretical Background

Image colorization can be used to modernize pictures or movies. There is a wide range of methods, ranging from hand colorization with photoshop to automatic colorization with artificial intelligence. For examples on images colorization see https://www.reddit.com/r/Colorization/. The TV-series "Greatest Events of WWII in Colour" states a good example for the usage of colorized footage. They use the techniques such that events appear more contemporary thus the historic incidents turn out even more worrying (https://www.imdb.com/title/tt9103932/). Automating the colorization problem with deep learning seems to be achievable, as training data is easily available.

2.1 Related work

Automatic approaches solving the colorization problem mostly differ in acquisition and handling of the data in order to model the accurate correspondence (Zhang et al., 2016). One can differentiate between parametric and non-parametric approaches. Non-parametric methods predict colors based on one or more reference images. That means the color distribution of the refer-

ence images, which are either provided by the user (e.g. Scribble-based colorization by Levin et al. (2004)) or automatically, is transferred to the target image. Hence, performance depends heavily on the quality of the provided data (Cheng et al., 2015).
Parametric methods on the other hands, learn prediction functions from large datasets of color images. Different methods are available to achieve this, one of which are convolutional neural networks (CNNs) in which the problem can be posed as regression or classification of quantized color values (Zhang et al., 2016).

2.2 Convolutional neural networks for image colorization

Whenever dealing with image data, CNNs are frequently used as model structures for deep learning tasks. They are inspired by the visual cortex of the brain. The idea is that highly specialized components learn a very specific task, which is similar to the receptive fields of neurons in the visual cortex (Hubel and Wiesel, 1962). These components can be combined to high-level features, which again can be merged to classes or transformed to the desired output shape. In CNNs, this concept is implemented by several successive convolutional layers: a weight kernel moves over the input image and calculates the new pixel value for each pixel position by multiplying the weights of the kernel with the neighboring pixel values and summing them up (see fig. 1). Different kernels can generate different so-called feature maps. One feature map targets the same feature (e.g. edges) in different image sections.

In this manner, CNNs can store spatial information about pixels and features. In a subsequent pooling layer dimensions are reduced by summarizing over the image section (e.g. max pooling takes the highest value of a certain image section). This facilitates the computation and drops unnecessary information (Effenberger, 2019). In recent years, CNNs improved such that they outperform humans in many classification tasks (Russakovsky et al., 2014).

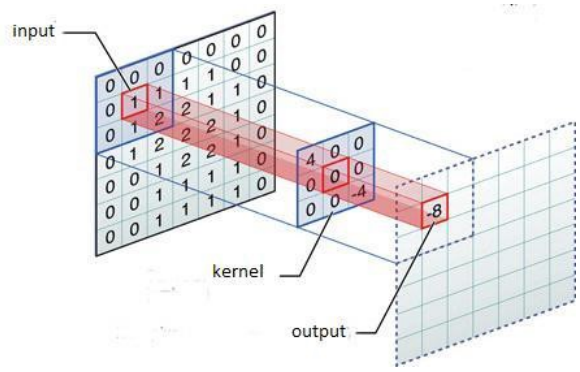

Figure 1: Convolutional operation: kernel slides over the input, multiplying it with its weight before summarizing the 3x3 neighborhood (Escontrela, 2018).

2.3 Guiding Paper

Zhang and colleagues (2016) propose a fully automatic approach to colorize grayscale images. They choose to solve this task with a feed-forward CNN as a classification task using a custom loss and class-rebalancing at training time in order to increase the diversity of the colours in

the final results. The model is trained over a million color images from ImageNet dataset (Russakovsky et al., 2014). Concerning the CNN architecture, they use a single stream, VGG-styled network with added depth and dilated convolutions.

The mapping, which Zhang and colleagues (2016) aim to learn, results out of the *Lab* color space: The input is the *L* lightness channel, the target channels are the *a* and *b* color channels. First, they used the Euclidean loss to train the model. This leads to overall grayish colors because the Euclidean loss favors the mean of all color pixel values. Therefore, Zhang et al. (2016) choose to treat the problem as a multinomial classification task. The *ab* output space is therefore divided into bins of grid size ten, and the 313 color values in-gamut span the 313 possible color combinations. Following, a mapping to a probability distribution over all 313 possible colors is learned for each input. The ground truth color is converted to a vector, using a soft-encoding scheme and a multinomial cross entropy loss, which is responsible for the class-rebalancing. Thereafter, they map the probability distribution to the color values. The class-rebalancing operates pixel-wise and the loss of each pixel is re-weighted at training time, based on how often the color occurs.

3 Networkstructure and Implementation

The re-implementation of the original paper (Zhang et al., 2016) was conducted in two steps and can be found on GitHub (https://github.com/marumse/colorize_images). This chapter will explain these two crucial steps in further detail. In both approaches, the ImageNet2012 dataset was used. Firstly, we implemented the model structure as described in the paper (see figure 2) and trained the model with the color layers of the input images as the target. Secondly, we translated the colorization task to a classification problem and trained the same model structure with the altered problem representation.

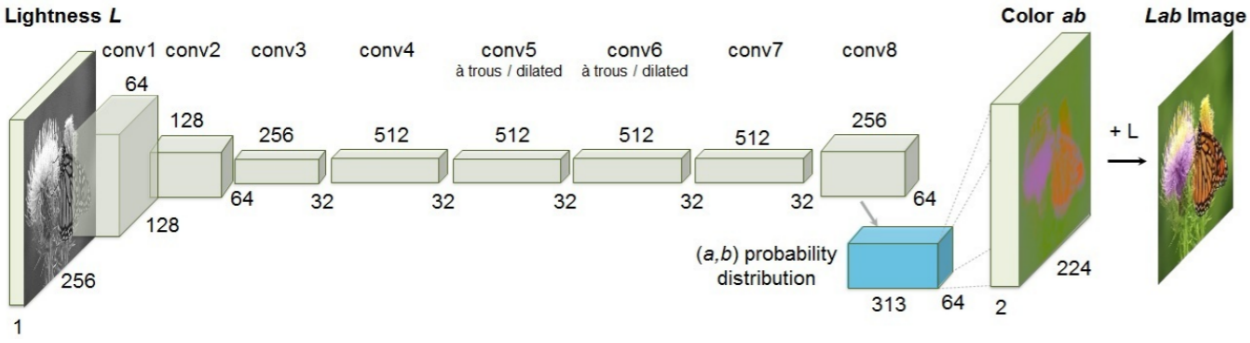

Figure 2: The network architecture of Zhang et al. (2016).

3.1 Data Generation

The ImageNet2012 dataset was available to us through the university server. As we were working with a large amount of data in each batch, it would have been impossible to load the whole dataset at once, hence we used a data generator to load the input and target images successively for each batch. Although there are inbuilt data generators available from keras that allow for some means of data augmentation, we built our costum generator to ensure the functionality we were aiming at. The generator takes the batch size and a list which contains the paths to the images that should be used in order to create the input and target arrays as described in the following.

First, the images are loaded and resized to a uniform shape of (224, 224, 3), which corresponds

```python
def generate_data(batch_size, file_list):
    """ Replaces Keras' native ImageDataGenerator.
        This function is a data generator that loads a costumized version of some data. More precisely, it loads images,
        tranforms them into LAB color space and returns the first layer as the input and the other two layers as the target for the model
        Args:       batch_size
                    file_list containing all image paths
        Return:     a tuple containing a numpy array with the inputs and a second numpy array with the targets
    """
    i = 0
    while True:
        image_batch = []
        label_batch = []

        for b in range(batch_size):
            # shuffle data when all files have been seen
            if i == len(file_list):
                i = 0
                np.random.shuffle(file_list)
            sample = file_list[i]
            i += 1
            image = cv2.resize(cv2.imread(sample), (224,224))
            image = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
            # split the image into the L layer for the input and ab layers for the target
            L = image[:,:,0][:,:,np.newaxis]
            ab = image[:,:,1:]
            # reduce the number of colors to 121 (11 different a and b values respectively)
            ab = (ab//25)*25
            # use cantor pairing to combine layer a and b
            cantor = cantor_pairing(ab)
            # make a one-hot-encoding out of the cantor labels
            hot = one_hot_encoding(cantor)
            # append both to the corresponding lists
            image_batch.append(L)
            label_batch.append(hot)
        yield (np.array(image_batch), np.array(label_batch))
```

Figure 3: The data generator creates batches of input and target tupels.

to the height, the width and the number of the color channels, respectively. Then, the images are transformed from BGR to *Lab* color space. The lightness channel (*L*-layer), which displays most of the structure, is separated from the other two channels and used as the input for the model. The remaining two channels (*ab*-layers) encode the color information of the images and are used as the models' target in the first approach. In the second approach, these processing steps remain the same. Additionally the target arrays are transformed to become a classification task.

The classification task is done in three main steps. The first step is to discretize the continuous

color space and reduce the number of possible colors by quantizing the a and b color ranges into eleven bins each. This yields a total of 121 possible colors by combining the a and b layers. In the second step, the a and b layers are combined into a single layer, which keeps the same height and width dimensions as before. This means that the color information of each pixel is now encoded in a single number rather than two, which leads to a single target layer. Cantor pairing is used to generate a unique and deterministic number from the two a and b values of each pixel. The Cantor pairing formula is shown in (1) and its implementation in figure 4. As cantor pairing is reversible, one can easily translate the pairing result back to the original color values with no loss of information (Meri, 2007).

$$z = \pi(x, y) = \frac{(x + y + 1)(x + y)}{2} + y \tag{1}$$

Now that there is a single value encoding each pixels' color, the third and last step is to translate

```
73  def cantor_pairing(ab):
74      """ Cantor pairing calculates a deterministic and unique number for a pair of positive integers.
75      Args:       two arrays of integers in our case the a and b color channels
76      Return:     the corresponding array with unique values
77      """
78      a = ab[:,:,0].astype(np.uint32)
79      b = ab[:,:,1].astype(np.uint32)
80      c = ((a + b) * (a + b + 1)) / 2 + b
81      return c
```

Figure 4: Implementation of the Cantor Pairing Function.

this value into a one-hot encoding. With the help of a dictionary, the cantor values are translated to numbers from 0 to 120. These numbers serve as the index in the one-hot encoding. Hence, color values that were previously represented in two values (a and b color channels) are now encoded by their index in the one-hot encoding. The target array has a shape of (224, 224, 121) and a one-hot vector is located at each pixel position.

```
51  def one_hot_encoding(cantor):
52      """ Get one-hot-encoding for cantor target image.
53      Args:       cantor transformed ab layers from input images
54      Return:     one-hot-encoding with shape width x height x number of colors (224 x 224 x 121)
55      """
56      one_hot = np.zeros((cantor.shape[0], cantor.shape[1], 121))
57      for i, unique_value in enumerate(np.unique(cantor)):
58          index = pair_to_index[unique_value]
59          one_hot[:, :, index][cantor == unique_value] = 1
60      return one_hot
```

Figure 5: Implementation of the one-hot encoding.

3.2 Modelstructure

The model structure is equivalent to the original implementation by Zhang et al. (2016). Only minor changes occur, mostly due to the translation from caffe to keras. The model consists of eight blocks comprising two or three repeated convolution and ReLU activation layers followed by a batch normalization. There are no pooling layers in the model, as changes in resolution are achieved through spatial down- or upsampling between the convolution blocks. A transposed convolutional layer is used to inverse the convolution and upsample the output back to the correct size.

In the first approach, a stochastic gradient descent optimizer (SGD) with a learning rate of 0.001 and a momentum of 0.9 is used, which is inspired by the original paper. SGD is a stochastic approximation to gradient descent, which uses an estimation for the gradients and thereby decreases the computational complexity. In each optimization step, a random data point is selected from which the gradients are calculated rather than using the whole dataset. The downside to SGD is that it does not guarantee to converge to a solution as the gradients might vary heavily from sample to sample (Effenberger, 2019). Further, mean squared error (MSE) is implemented as the loss function, which uses the average squared difference between the estimated and the target values (2). In the case of image data, that means that each estimated pixel value is compared to the target pixel value and the average over the squared differences is used to optimize the estimation process.

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2 \tag{2}$$

In the second approach, the softmax activation function is used in the last layer to prepare the model's output for the categorical cross entropy loss (3), which compares the output vector ($\tilde{y}$) to the one-hot encoded target vector (y).

$$L(y, \tilde{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N} y_{ij} * log(\tilde{y}_{ij}) \tag{3}$$

Both, SGD and Adam optimizer are tested. In the two approaches, kernel weights are initialized with the *Glorot uniform initializer* which draws samples from a uniform distribution depending on the number of input and output units of the corresponding weight tensor. Biases are initialized with zeros.

3.3 Training

The models were trained on the grid of the Institute of Cognitive Science at Osnabrueck University. A helper script was written that distributed several grid jobs on different computers in order to experiment with the hyperparameters such as the learning rate and the batch size. Through this script, one can easily change these parameters as well as select the data set in addition to the environment and switch between the training mode and the prediction mode. The classical model was trained with 5000 images, a batch size of 10 and a learning rate of 0.001. The classification model was trained with 2000 images, batch sizes of 10 or 20 and learning rated from 0.1 to 0.001. Unfortunately, the grid jobs terminated after three to six epochs without giving a hint for errors in the code. It appears to be a grid internal memory problem (see appendix figure 9). Therefore, we saved the model in checkpoints after each epoch and used those weights for the testing.

3.4 Testing

To evaluate the predictive power of our two approaches, colors for unseen test images are predicted and evaluated via visual inspection. When the script is started in prediction mode, the model is not trained, but the weights from the corresponding training process are loaded. The model then predicts the most plausible colors for the L-layer of each input image. In the first approach, the model's output can be interpreted as the ab-layers. That means the output can be combined with the input (L-layer) and displayed as an image straight away. For the second approach, three decoding steps are necessary before generating human-readable images. Firstly, the output of the softmax layer is decoded to find the index of the most likely color value. Secondly, this index is translated to the cantor pairing value it represents with the help of the created dictionary. And thirdly, the cantor pairing value is transformed back to the a and b values it is originally constituted of. These a and b layers can finally be combined with the L-layer to display the predicted image in Lab color space.

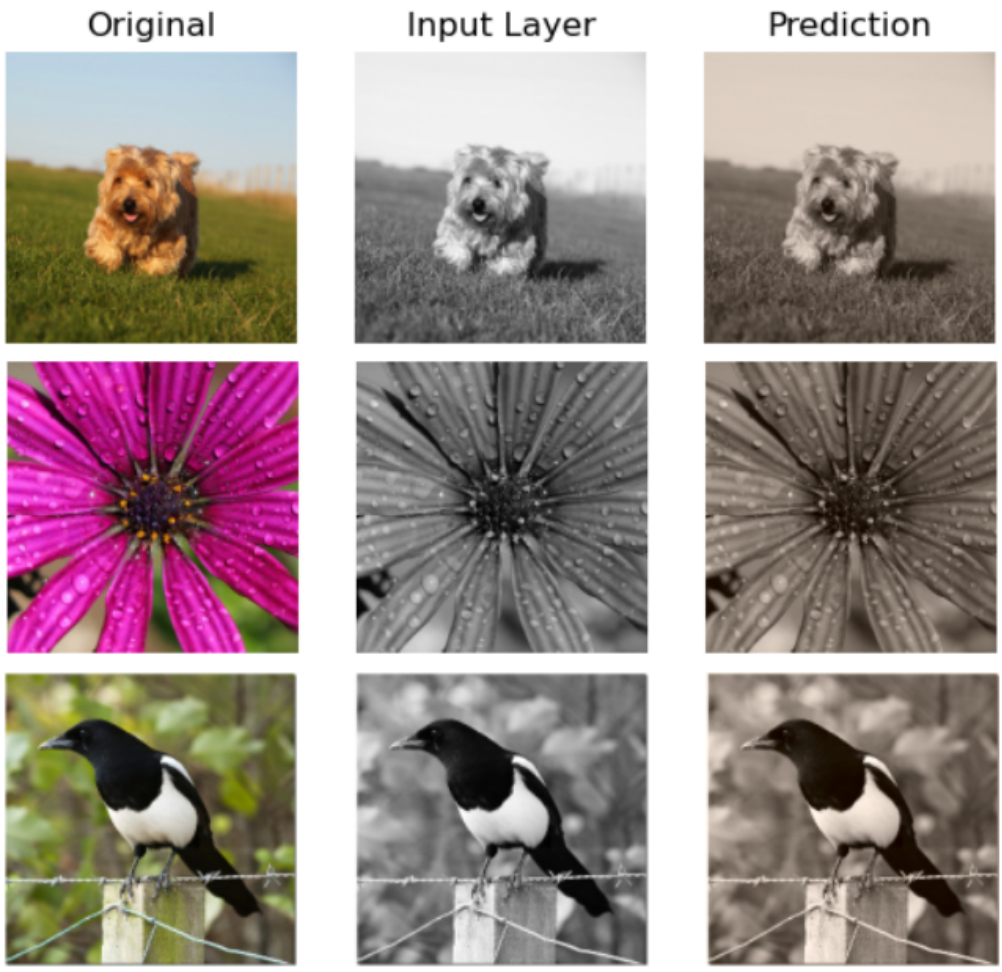

Figure 6

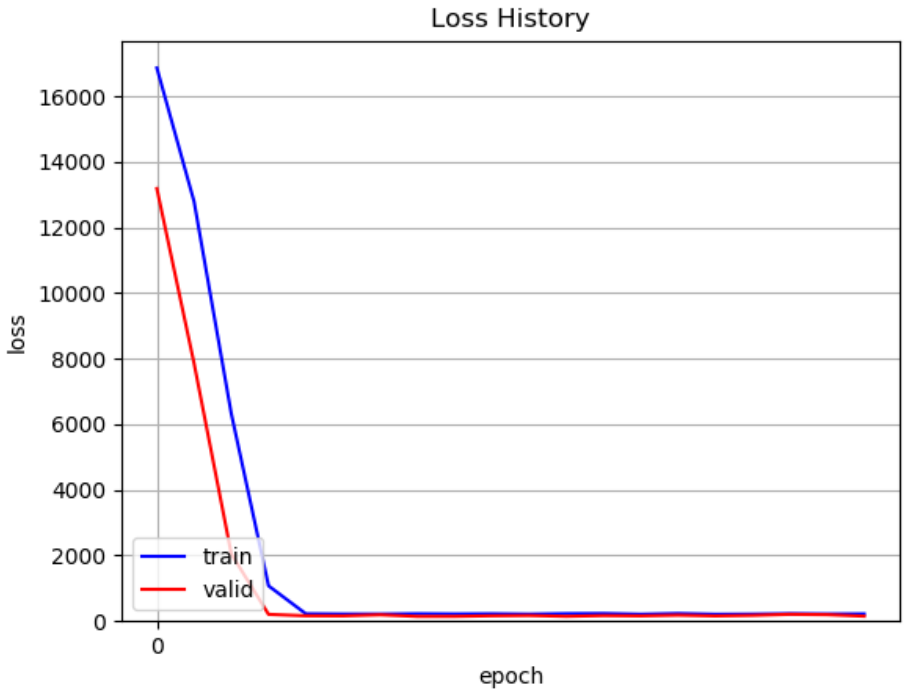

Figure 7

4 Results

4.1 Classical Approach

4.2 Classification Approach

5 Discussion

5.1 Conclusion and Future Work

- vergleich ziehen zu Zhang

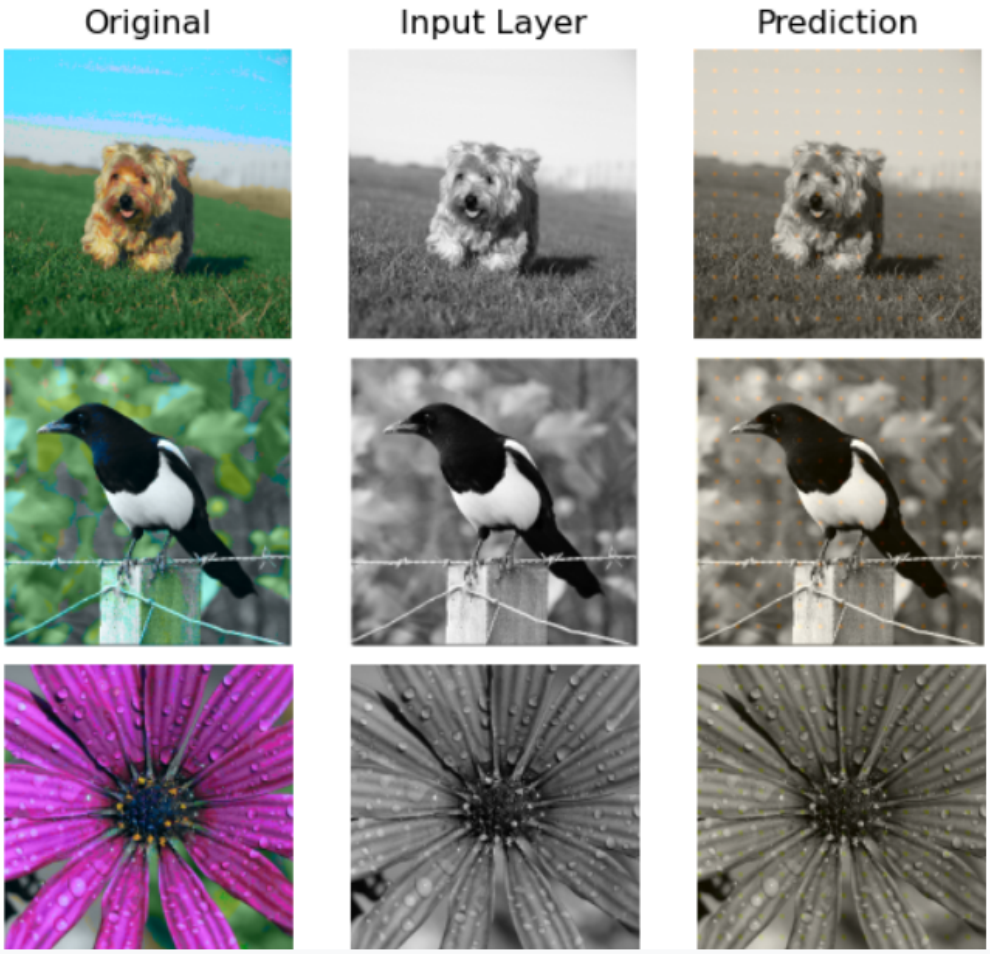

Figure 8

6 Literature

Cheng, Z., Yang, Q., and Sheng, B. (2015). Deep colorization. *2015 IEEE International Conference on Computer Vision (ICCV)*.

Effenberger, L. (2019). Implementing anns with tensorflow.

Escontrela, A. (2018). Towards data science: Convolutional neural networks from the ground up. https://towardsdatascience.com/convolutional-neural-networks-from-the-ground-up-c67bb41454e1.

Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106–154.

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

Levin, A., Lischinski, D., and Weiss, Y. (2004). Colorization using optimization. *ACM Trans. Graph.*, 23(3):689–694.

Meri, L. (2007). Some remarks on the cantor pairing function some remarks on the cantor pairing function. *Le Matematiche*, 62.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C., and Li, F. (2014). Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575.

Zhang, R., Isola, P., and Efros, A. A. (2016). Colorful image colorization. *CoRR*, abs/1603.08511.

A Error Message

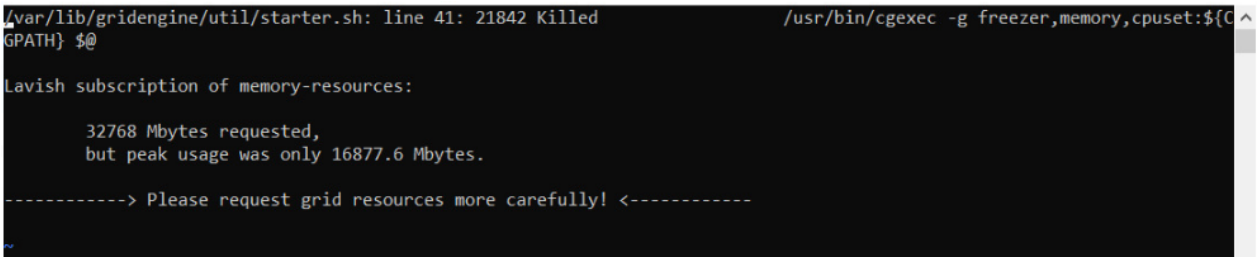

Figure 9: Error message that occurs during running the classification model on the university server.