ACML-Assignment-2

p.arroelofs
yumao.gu

November 2020

1 Exercise 1

1.1 1.1

Divide your data set into training (80%), validation (10%) and test (10%). Normalize the data.
We randomly shuffle the train data set, use the first 1/8 of the data as the validation data, and use the remaining data as the training data. Since CIFAR-10 has it's own test data set, so we use it directly. We normalize the images to (0,1) using To-Tensor function in pytorch.

1.2 1.2

Implement the auto encoder network specified above. Run the training for at least 10 epochs, and plot the evolution of the error with epochs. Report also the test error.
We implement the model and get the result as below. The left image is the original picture and the middle is the output of the model. Also the errors are shown in the right.1

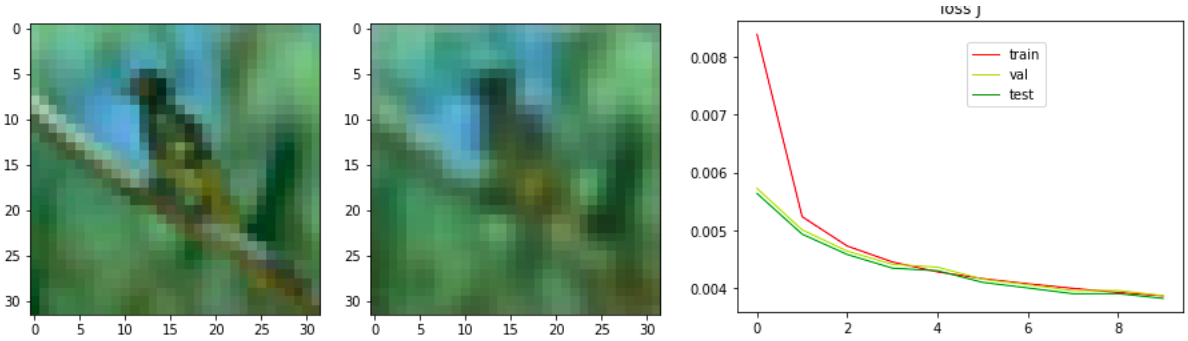

Figure 1: The results of exercise 1.2

2 Exercise 2

2.1 2.1

What is the size of the latent space representation of the above network?
According to the formula in the "Lab-CAE.pdf", the representation of latent layer is

$$W = 8, K = 3, P = 1, S = 1, C = 16$$

$$representaion = (\frac{W - K + 2P}{S} + 1)^2 \times C = (\frac{8 - 3 + 2}{1} + 1)^2 \times 16 = 1024$$

2.2 2.2

Try other architectures (e.g. fewer intermediate layers, different number of channels, filter sizes or stride and padding configurations) to answer questions such as: What is the impact of those in the reconstruction error after training? Is there a correlation between the size of the latent space representation and the error?

2.2.1 architecture factors

Generally speaking, we choose the size of the convolution **kernel** to be an odd number, because the size of the feature map before and after can be kept unchanged by adjusting the padding.
Smaller convolution kernels are more popular, because the receptive field of the multi-layer small convolution kernel is equivalent to the receptive field of the large convolution kernel, and it has more nonlinear expression capabilities and smaller parameter requirements.
There is no theoretical support for the number of **channel**s in each layer, but in the part that can be regarded as a feature extractor in the model, the deeper the layer often requires more channels. In order to make the expression ability of the front and back layers similar, generally choose to increase twice
The **padding** makes the input dimension of the convolution layer consistent with the output dimension; at the same time,
The max **pooling** layer can artificially increase non linearity, compress features, and provide spatial in-variance to a certain extent. But this leads to sparse gradients which is not conducive to information transmission.So it is more suitable for classification than for information extraction.
Stride convolution can also compress features, but it takes more time to train. Generally use *conv+stride conv* instead of *conv + max pool*, instead of directly using *conv(stride = 2)* instead.

2.2.2 Generalizations

Generally speaking, the number of layers in a model is positively correlated with the model's performance **?**. We decided to follow this pattern by not only increasing the number of used channels, but to also increase the number of layers. This increase is done symmetrically for both the encode and decode part of the network. The final architecture can be seen in "TODO:architecture picture"

1. We directly double the channels in the latent layer. As we can see, it's much better than the base-line.

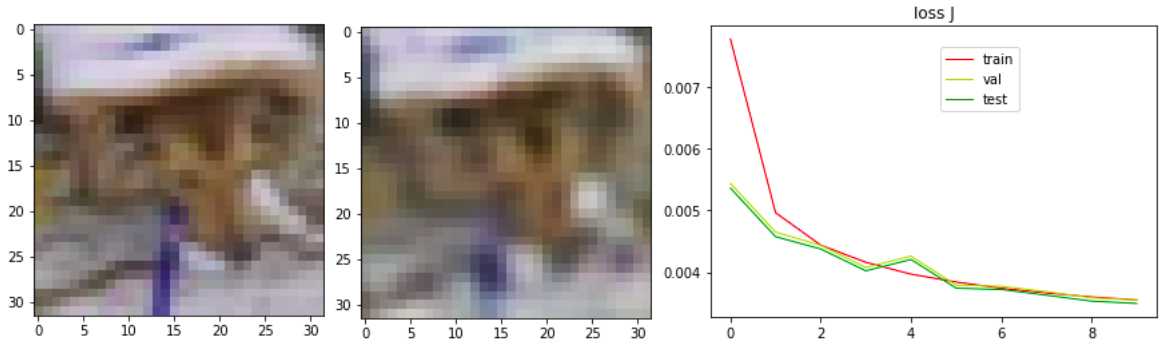

Figure 2: double the channels in the latent layer

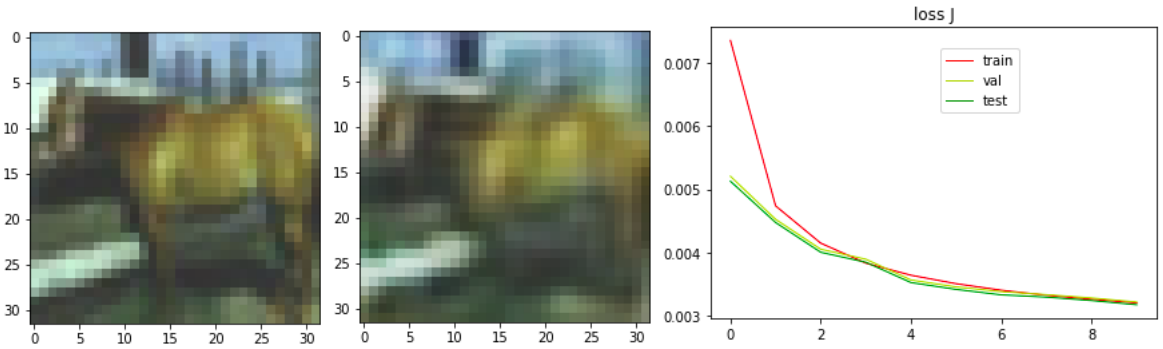

Figure 3: 4 times the channels in the latent layer

2. We directly increase the all channels. The latent layer representation is the same as above, but the performance is better. So the other layers representation ability can also affect the loss error.

3. We use the 5*5 kernel and adjust the padding with 2. When the model's

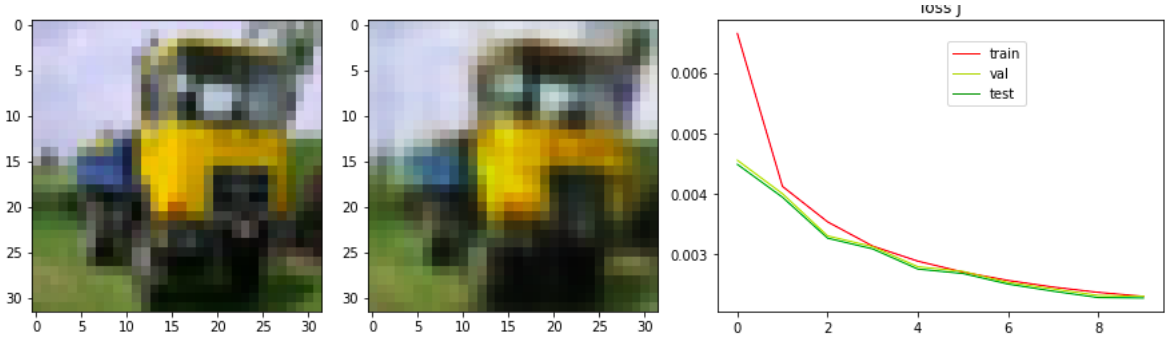

Figure 4: double all channels

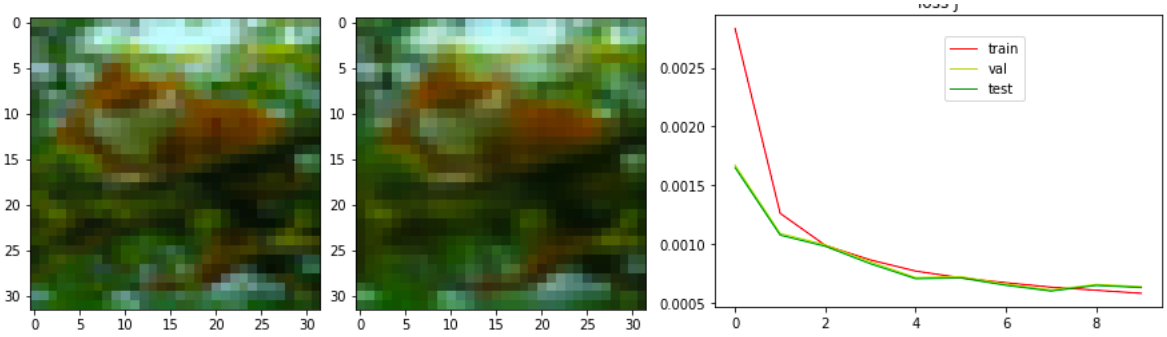

Figure 5: 20 times all channels

layers don't change, a bigger kernel size can get bigger receptive field, so it works a little better than the smaller one.

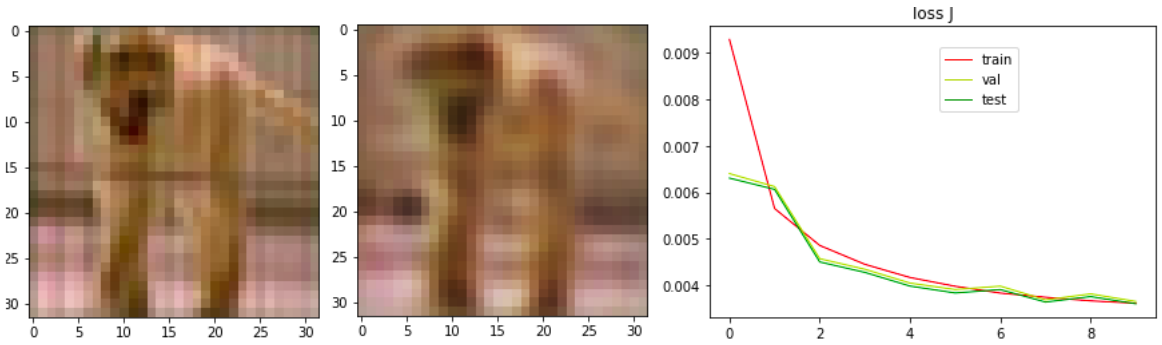

Figure 6: the 5*5 kernel

4. We use stride convolution layer to replace the max pooling layer. It works

much better.

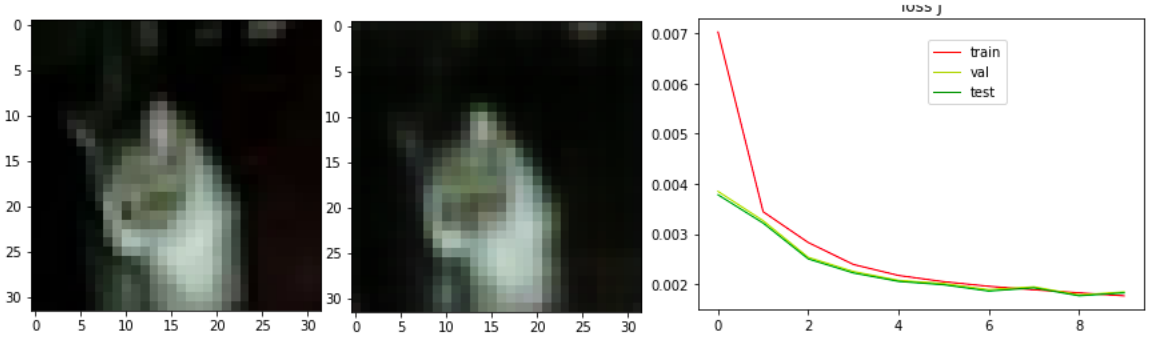

Figure 7: use stride convolution layer

5. We remove one pooling layer and one up-sampling layer and the performance is better than that of increasing channel, even though the both representation ability is same. Adding more layers will lead to a information loss, and performs bad.

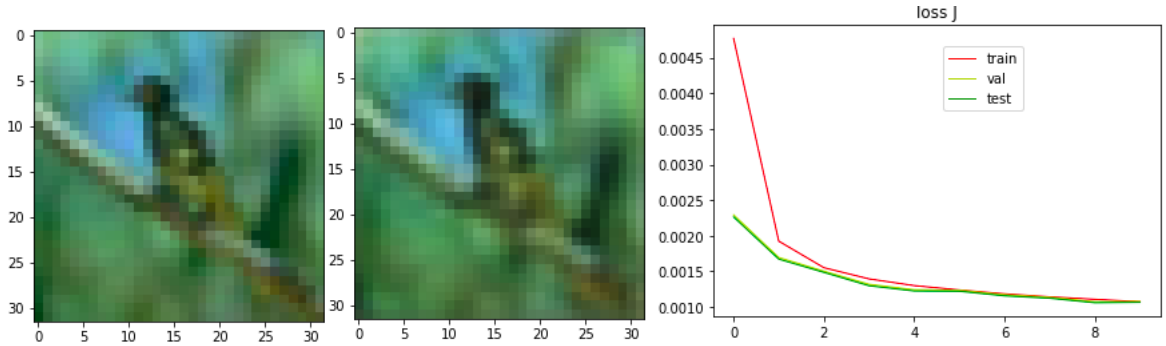

Figure 8: remove one pooling layer and one up-sampling layer

2.2.3 other factors

We don't give the experiments about non-architecture factors in this section, but give the simple reasons about non-architecture factors choice.

1. Optimizer: We use the Adam because it considers both the first and second moments

2. Learning rate: We just set a fixed value of 0.001

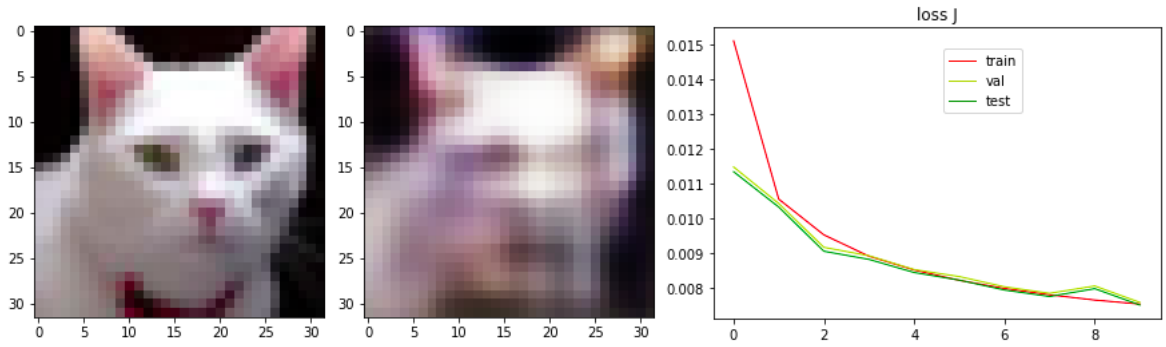

Figure 9: Adding more layers

3. Relu activation function make the gradient sparse because the negative part is always 0 and some neurons cannot be activated, so we use LeakRelu here.

4. We use the Sigmoid function in the last layer because it provides more info than LeakRelu but won't lead to a gradient disappearance.

5. We use 16 as the batch size to get a balance between training speed(one epoch on GPU),stability(gradient decent more smoothly) and generalization ability within the same training time(different batches have same gradient decent direction, leading to a local minimum within finite time).

3 Exercise 3

3.1 Exercise 3.1

In section 2 we described using the same input with different gamma, contrast and saturation to achieve better reconstruction. Unfortunately, the input of the autoencoder is an image with only one (luminance) channel. Therefore, contrast and saturation of the image cannot be shifted as they both need three RGB channels to work properly. Instead, the input of the coloring autoencoder are four grayscale images with shifted gamma. Empirically it was found that the gamma shifts of $[0.1, 0.5, 1.5, 3.0]$ tend to work the best when reconstructing images.
The output of the coloring autoencoder are the Cb and Cr channels of YCbCr encoding. The Y part does not need to be predicted as it stands for luminance, or greyscale image itself which we already have. The output is then converted to RGB for better visualisations.

3.2 Experiments and Results

To get an accurate read on the effects different hyperparameters have on the performance of the model we have run a total of fifteen (15) tests, starting from the baseautoencoder and slowly improving it. The changes which showed incremental improvement are left in the model for further testing, while changes which caused the model to perform worse are left out.

3.2.1 Base autoencoder

The performance of the standard model can be found in Figure 10. This model has used Adam as it's optimizer with a learning rate of $\alpha = 0.001$. "TODO: describe results"

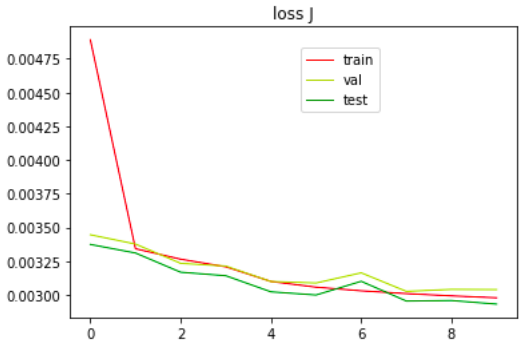

Figure 10: Performance of the base autoencoder presented in the assignment when coloring images.

3.2.2 Base autoencoder using RMSprop

Figure 11 shows the impact that the RMSprop optimizer had on the results. As can be clearly seen from Figures 10, 11, the Adam optimizer not only converges faster, but also to a lower value after the same number of iterations.

3.3 Decreasing α value by a factor of 10

In this experiment, the learning rate used for Adam autoencoder was increased from $\alpha = 0.001$ to $\alpha = 0.0001$. The results are shown in Figure 12. This change allowed for a smooth transition during every epoch to eventually a lower loss.

3.4 Increasing the number of channels by a factor of two (2)

The increased number of channels would give the model more parameters to work with, allowing it to express the data better. The results are shown in

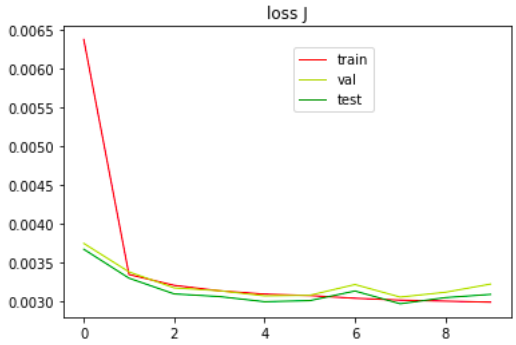

Figure 11: Base autoencoder using RMSprop as it's optimizer

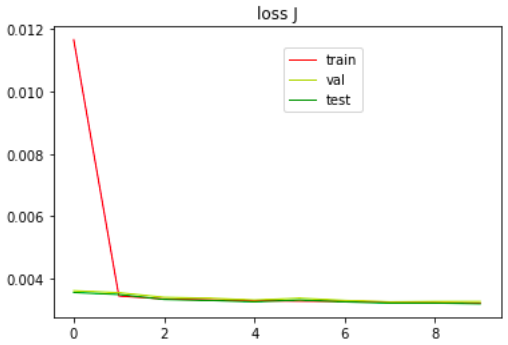

Figure 12: Decreasing the learning rate by a factor of 10

Figure 13. Increasing the number of channels has indeed produced better results and decreased all three loss functions.

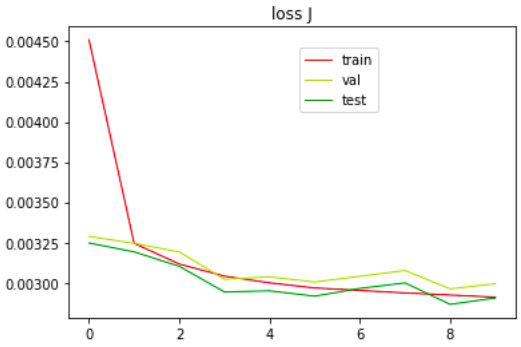

Figure 13: Base autoencoder double the convolutional channels for every layers

3.5 Increasing batch size from 16 to 64

Increasing the batch size significantly decreases the training time but at the same time might hurt the performance of the model. The results are shown in Figure 22. As expected, the overall loss has increased even when compared to an unaided base autoencoder.

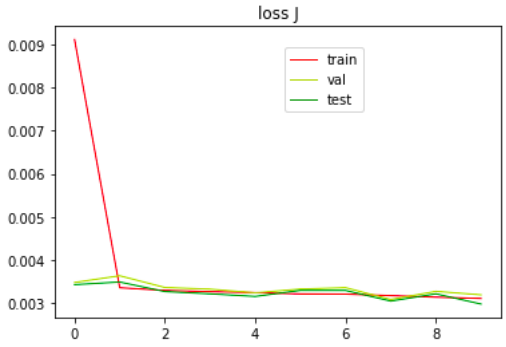

Figure 14: Base autoencoder with an increased batch size. From 16 to 64.

3.6 A custom model

During this experiment, we decided to create our own model for color autoencoder based on the state of the art papers that we had found on the internet. As can be clearly seen in Figure 23, the results have drastically improved. This can be attributed to a number of factors, including the decreased learning rate and batch size and the increased expressiveness of the model. Images produced by this result are shown in Figures 15, 16, 17, 18, 19, 20.

The architecture is as follows:
Learning rate: 0.0001
Batch size: 1
Convolutional layer: 4 input, 16 output, 3 kernel
Convolutional layer: 16 input, 16 output, 3 kernel
Convolutional layer: 16 input, 16 output, 3 kernel, stride 2
Convolutional layer: 16 input, 32 output, 3 kernel
Convolutional layer: 32 input, 32 output, 3 kernel
Upsamlping: 2,2
Reverse convolutional layer: 32 input, 32 output, 3 kernel
Reverse convolutional layer: 32 input, 16 output, 3 kernel
Upsamlping: 2,2
Reverse convolutional layer: 16 input, 8 output, 3 kernel
Reverse convolutional layer: 8 input, 2 output, 3 kernel

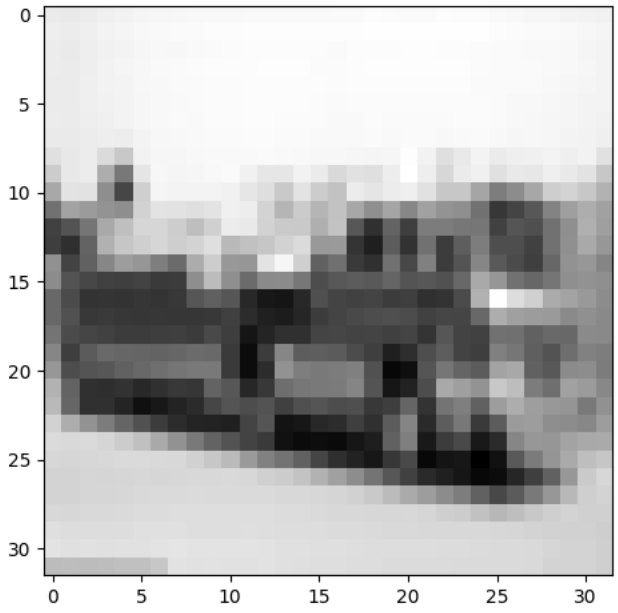

Figure 15: Grayscale motorscycle

3.7 Exercise 3.2

One obvious yet notable shortcoming of our network is the concept of asking the autoencoder to color a set of mostly color-independent images. A metaphorical enterpretation of our model could be a human trying to color in a painting in a single brush stoke, as opposed to focusing on coloring each object individually. A shortcoming which was noted by all successful image-coloring algorithms **?** "TODO: cite more?". The solution to this problem follows directly from the problem definition - coloring the image in "one stroke". Instead, state-of-the-art algorithms universally tend to employ some form of either custom or pre-built object detect. The position and class of those objects is then fed into the network, providing far more information about the grayscale image than the image itself. In a way, this process allows for the autoencoder to mimic a human painter, focusing on each object, one at a time. Unfortunately using a pre-trained classifier to color the images is slightly out of the scope of this project.

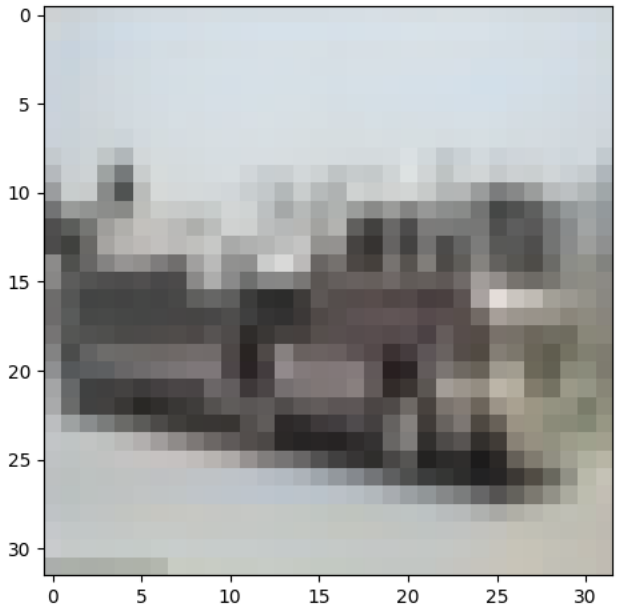

Figure 16: Colored motorscycle

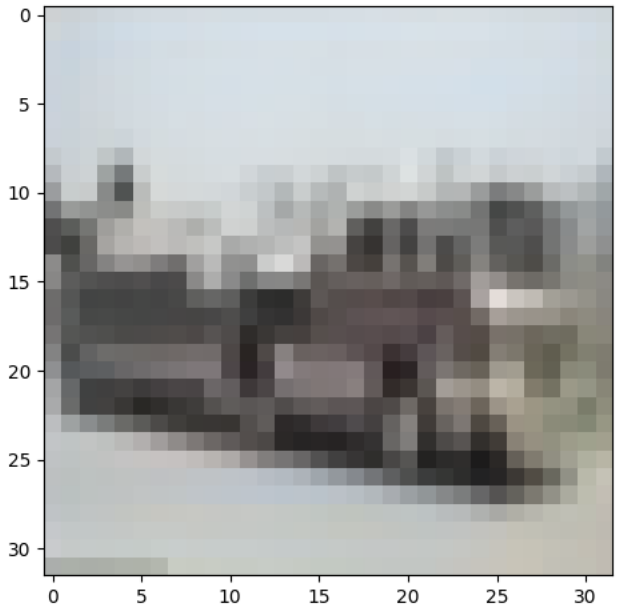

Figure 17: Original motorscycle

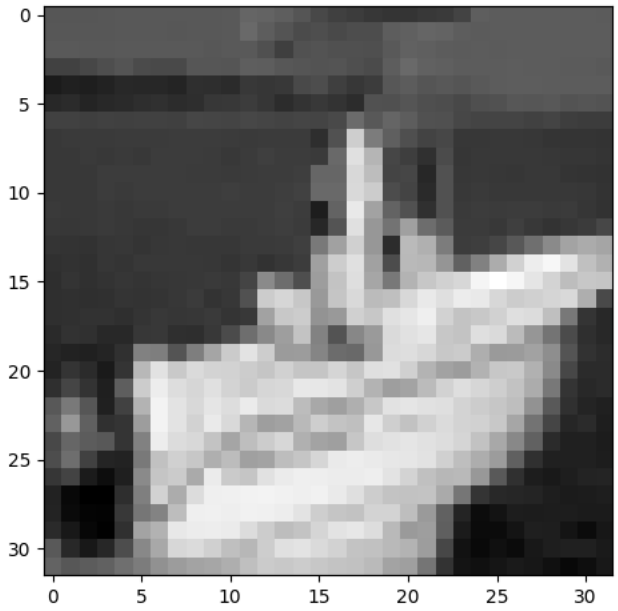

Figure 18: Grayscale boat

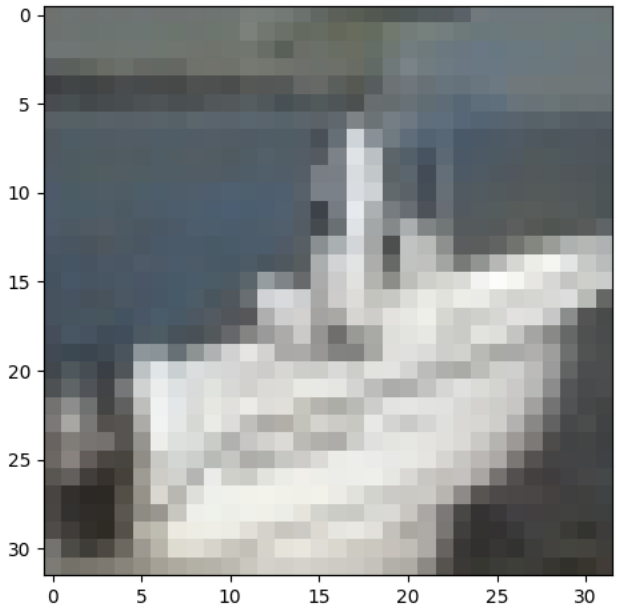

Figure 19: Colored boat

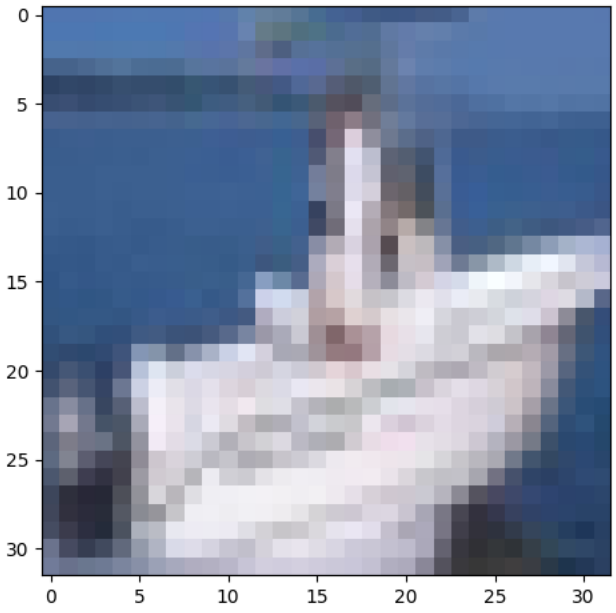

Figure 20: Original boat

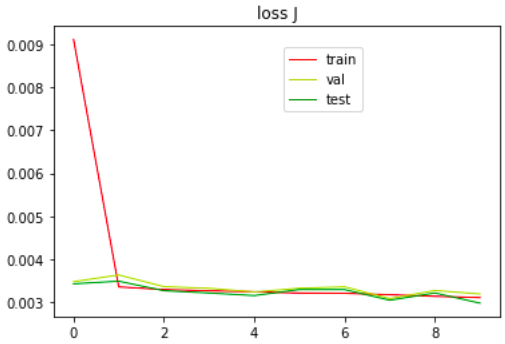

Figure 21: Base autoencoder with an increased batch size. From 16 to 64.

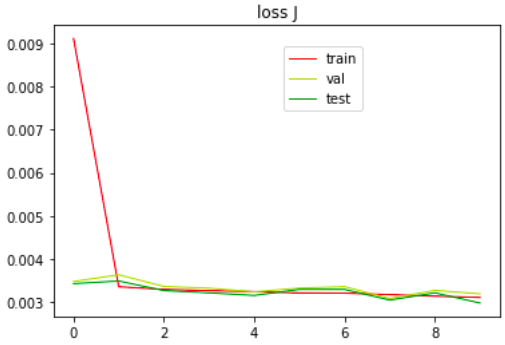

Figure 22: Base autoencoder with an increased batch size. From 16 to 64.

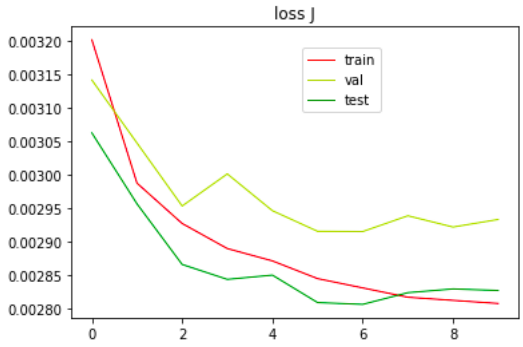

Figure 23: Custom autoencoder with significantly increased number of layers and channels.