CENG 483

Introduction to Computer Vision

Fall 2021-2022

Take Home Exam 3

Image Colorization

Student ID: 2375574

Please fill in the sections below only with the requested information. If you have additional things to mention, you can use the last section. Please note that all of the results in this report should be given for the **validation set** by default, unless otherwise specified. Also, when you are expected to comment on the effect of a parameter, please make sure to **fix** other parameters. You may support your comments with visuals (i.e. loss plot).

1 Baseline Architecture (30 pts)

Based on your qualitative results (do not forget to give them),

- Discuss effect of the number of conv layers:
- Discuss effect of the kernel size(except the last conv layer):
- Discuss effect of the number of kernels(except the last conv layer):
- Discuss effect of the learning rate by choosing three values: a very large one, a very small one and a value of your choice:

As number of convolutional layers increases, the network capacity increases. Hence, model can fit to data more strictly. Learning the model with 4-convolution requires more time, since capacity is much greater than 2-conv and 1-conv networks and we can observe this phenomenon in Figure-1 where red curve decreases slower than green and orange curves. One interesting observation is that why 4-conv has more train-loss than lower capacity networks, since theoretically it has more capacity to overfit/memorize the train-data, the reason is that probably we haven't gave him enough epochs to memorize the train-data since max-epochs is restricted to be 100. However, my observation is that after 20-40 epochs, rate of change of 4-conv is greater than 2-conv and 1-conv meaning that it keeps decreasing more even if it's slow. Another interesting observation is that why train-loss and valid-losses are almost the same at every epoch, the reason is that problem is not very difficult, we are trying to colorize the facial images and also train data and valid data are very similar and balanced. So it's expected to get similar curves for both valid-loss and train-loss.
Common Hyperparameter Baseline in Exp:
learning-rate = 0.001, batch-size = 16, epoch = 100, kernel-num=8, kernel-size=3

I've experimented with different kernel sizes of 3 and 5 for both 2-conv and 4-conv networks. For both of these models 5x5 kernel yields better performance than 3x3 kernel as illustrated in Figure-2. Higher spatiality with larger kernels for convolution improves performance in this task. The optimal kernel size depends on the problem and the configuration, however in this problem higher spatiality over kernels allows us to get better estimations for the face pixels.
Common Hyperparameter Baseline in Exp:
learning-rate = 0.001, batch-size = 16, epoch = 100, kernel-num=8

Number of kernels or output channels of convolutions determine how many activation maps we are going to create as a result of convolution. Each convolution operation with a single kernel yields one activation map and we can obtain multiple activation maps by convolving img with different learned kernels. This allows our model to capture different features or patterns out of fed images. Thus, we have more powerful feature representation and more model complexity which is expected to increase the model performance or decrease the loss. In this experiment, observations confirm my expectations in the sense that models with higher number of activation maps yield better performance for both 2-conv and 4-conv: 8,4,2 respectively. When we analyze convs separately, in the order of increasing performance: 2-conv: 2-conv-2ks, 2-conv-4ks, 2-conv-8ks and 4-conv: 4-conv-2ks, 4-conv-4ks, 4-conv-8ks.
Common Hyperparameter Baseline in Exp:
learning-rate = 0.001, batch-size = 16, epoch = 100, kernel-size=5

Learning rate determines the scale of update in gradient descent. In this experiment I've tried several learning rates: 1, 0.1, 0.01, 0.001, 0.0001 respectively for both 2-conv and 4-conv. I've obtained similar observations for both 2-conv and 4-conv. Larger learning rates except for 1.0 has resulted in better accuracy. Learning rates in the order of performance: 0.1, 0.01, 0.001, 0.0001, 1.0. Learning rate 0.1 is the best, but lr=1.0 is too much and it diverges our nn resulting in NAN loss values. For this task, my choice would be lr=0.1 with its superior performance when compared to other learning rates. Theoretically, small learning rates should converge, but they usually require more training time since having small updates requires many steps to make a significant effect in the optimization. However, these experiments are bounded with max-epoch=100, hence we cannot observe smaller learning rates converging as low as higher learning rates. Higher learning rates like 0.1 (not as extreme as 1.0) would be a plausible lr choice for better performance, because our data is scaled and we backpropagate small loss values and we have max-epoch restriction.
Common Hyperparameter Baseline in Exp:
batch-size = 16, epoch = 100, kernel-num=8, kernel-size=5

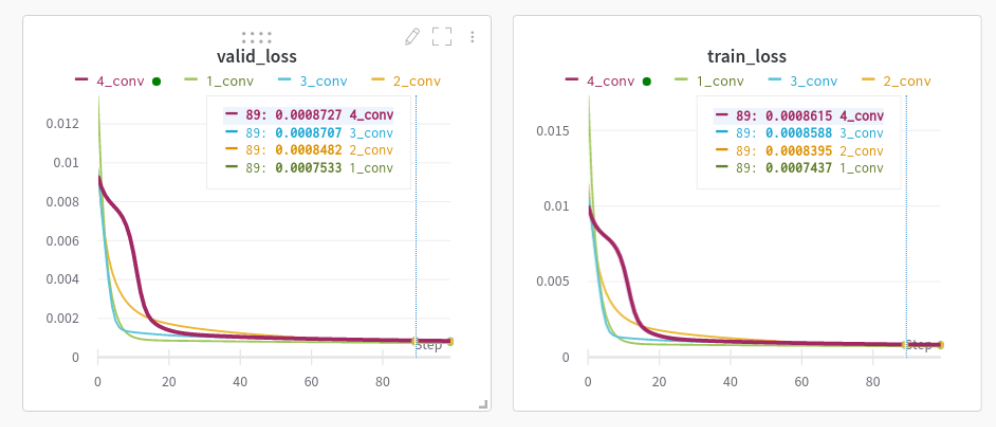

Figure 1: Effect of Different Conv Layers

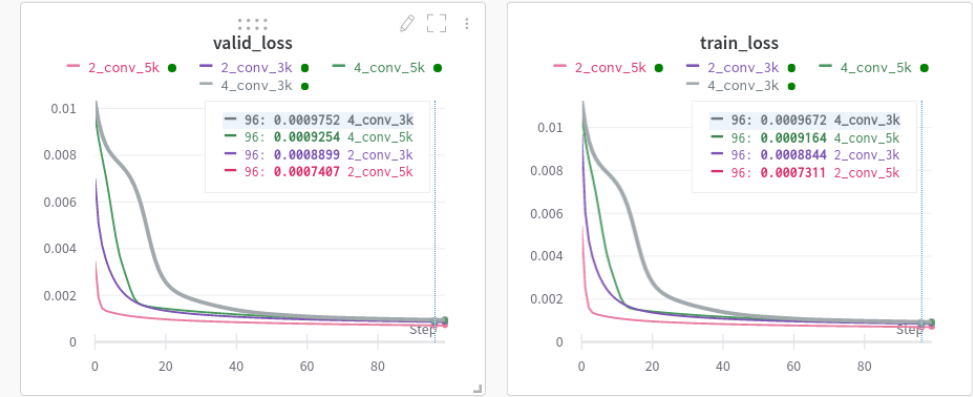

Figure 2: Effect of Different Kernel Sizes

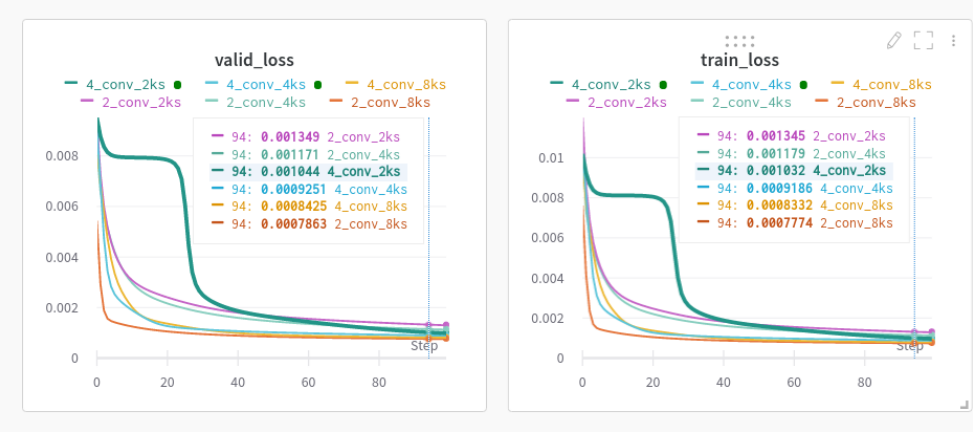

Figure 3: Effect of Different Number of Kernels

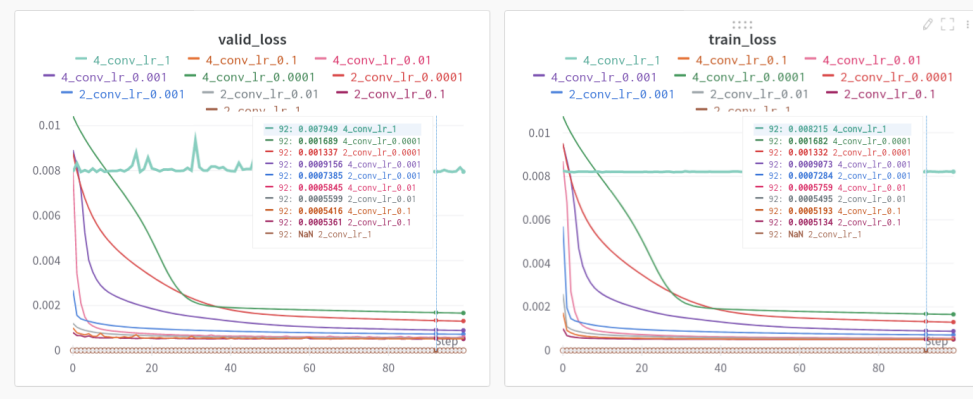

Figure 4: Effect of Different Learning Rates

2 Further Experiments (20 pts)

Based on your qualitative results (do not forget to give them),

- Try adding a batch-norm layer (torch.nn.BatchNorm2d) into each convolutional layer. How does it affect the results, and, why? Keep it if it is beneficial.
- Try adding a tanh activation function after the very last convolutional layer. How does it affect the results, and, why? Keep it if it is beneficial.

- Try setting the number of channels parameter to 16. How does it affect the results, and, why? Keep it if it is beneficial.

I compared my results with respect to two baselines for different number of convolutional layers, 4-conv and 2-conv. Overall, my observations are aligned for both 2-conv and 4-conv. All of the experiments in this part are summarized in Figure-5 with following abbreviations; bn: batch-norm experiment, tanh: tanh experiment, c16: 16 channel experiment. 2-conv and 4-conv stands for 2 and 4 conv layer NN respectively.
Common Hyperparameter Baseline in Exp:
learning-rate = 0.1, batch-size = 16, epoch = 100, kernel-size=5

Usually, batch-norm increases the performance of NN since scaling eases optimization during learning. However, BatchNorm2d increased mse loss for both 2-conv and 4-conv, its loss is the highest among all observations in Figure-5. The reason might be because, the input and ouput data are already scaled to the interval of [-1,1], so model doesn't have any benefit in this configuration. However, it has some regularization effect on the weights and as a result it requires more data for training and increases the loss. Therefore, I don't see much of a benefit for batch-norm and I'm not planning to keep it under this configuration.

I've expected tanh to improve the model's accuracy and decreasing the loss. I've observed in Figure-5 that for 4-conv tanh decreases the loss as expected, however for conv-2 the behaviour is opposite, the loss is higher than 2-conv-base. This might be because of the fact that, when we change the network, even 1 single additional activation function yields another different optimization problem, hence our baseline hyperparameters eg. learning-rate might not be the optimal for that configuration. In addition, these loss curves correspond to avg loss per pixel and also all of these individual pixel values are scaled to the interval of [-1,1] which is relatively small. Hence all of these models result in a very low mse loss value and loss differences between the models are marginal, so we need to make use of accuracy checking mechanisms like 12-margin and also manual image inspection. For example, I observe that applying tanh in both 2-conv and 4-conv has removed exploding pixel values eg. small blue segments in the image which is an improvement, but not directly visible in mse loss. Tanh works because we are squeezing our network's output to [-1,1] and our labels are also in this range. However, if we didn't apply tanh to conv output, it would be theoretically unbounded which could be more difficult to learn, in a way we are guiding the network to learn better since we have prior knowledge about our ground truth interval. Consequently, I'll keep tanh activation for the last layer.

Number of channels or number of kernels that I've experimented with so far were 2, 4, and 8 as mentioned in the HW description. In this experiment, I've set channel-out=16 for all conv layers except for the last one which had to be 3 (r,g,b). I've expected a higher channel number to decrease the loss and this phenomenon is observable in Figure-5 where losses are the lowest among all of the models in this part for both 4-conv and 2-conv. Having higher number of channels/kernels gives our model more capacity to extract more different features from images and the model makes use of more of these features to fit the problem/data in a better way as a result decreasing the loss. Therefore, I'll keep channels as 16 since it improved our model by considerable amount.

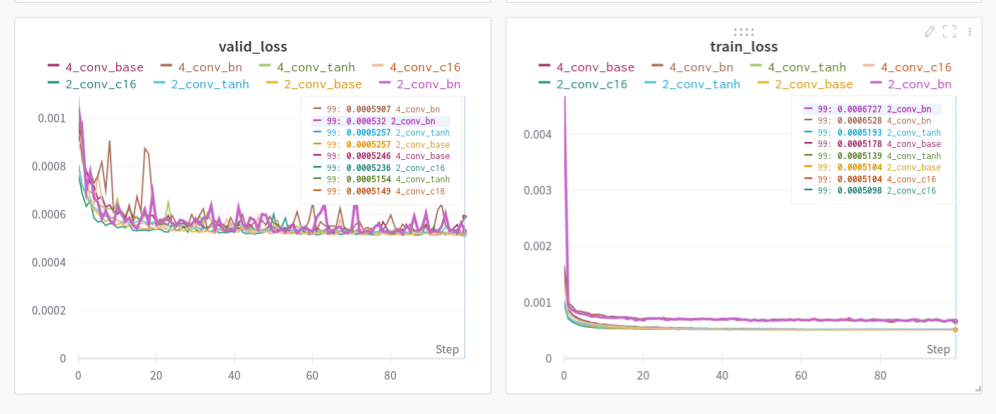

Figure 5: Effect of batch-norm, tanh, channels

3 Your Best Configuration (20 pts)

Using the best model that you obtain, report the following:

- The automatically chosen number of epochs(what was your strategy?):
- The plot of the training mean-squared error loss over epochs:
- The plot of the validation 12-margin error over epochs (see the3 text for details):
- At least 5 qualitative results on the validation set, showing the prediction and the target colored image:
- Discuss the advantages and disadvantages of the model, based on your qualitative results, and, briefly discuss potential ways to improve the model:

Best model I've obtained so far has the following hyperparameter configuration:
learning-rate=0.1, kernel-size=5, kernel-num=16, out-activ=tanh, conv-layers=2

For automatically choosing the number of epochs, I've monitored training with validation loss values. Namely, I've compared previous epoch's validation loss to the current epoch's validation loss and if we don't have sufficient improvement more than epsilon or if our model's loss increases more than epsilon for more than patience number of times then we stop learning and take our model's configuration at the epoch of (current epoch - patience) where patience determines the amount of toleration and epsilon is the small difference value that determines whether change should be considered as success or not. I've set epsilon = 5e-6 and patience = 5. The learning has stopped at 47 th epoch which is determined by our mechanism automatically.

The plot of both training and validation MSE Loss over epochs is illustrated in Figure-6 below.

The plot of both training and validation 12-Margin Loss over epochs is illustrated in Figure-7 below.

In valid_imgs directory there are 7 pairs of (pred, label) images.

The model is very simple and easy to train, it takes around 10 minutes at max for training with gpu. It gives very close results to ground truth images, it doesn't capture every pixel perfectly, however it's still good approximation. Its 12-margin accuracy is around 0.75 (equivalently 0.25 12-margin loss) which is quiet amazing result from such a trivial neural network. However, train set and validation set include

similar images and it's observable from our loss values such that both train and valid losses are almost similar which diminishes the generalizability of our model. Hence, if the domain of application for this model may contain different data (different images with different face shapes, skin colors etc), then we would need to have a more generic validation set to properly measure the performance of the model. Additionally, this model or NN structure wouldn't work with images of different size, since there's no pooling layer in NN. This also limits the usability and practicality of the model. In order to increase the performance of the model, we need to increase the capacity of the model as well as introduce more training data. As it's clear from learning curves, both train and validation losses converge around same values. In that case higher network capacity would also increase the gap between validation and training which needs to be reduced through regularization. Hence, I would try & apply weight-decay and dropout. In order to increase model capacity, I would try increasing the number of kernels for each conv layer as motivated by our previous empirical results. Furthermore, I would try stacking more conv layers with pooling layers in between. In this regard, another disadvantage of the model is that it doesn't sufficiently decrease the spatiality of the propagated tensors throughout convolutions due to padding, limited number of convolutional layers, kernel size, and stride. Higher spatiality would be helpful to focus on the details of the image sections eg. hair, nose, lips, ears etc.

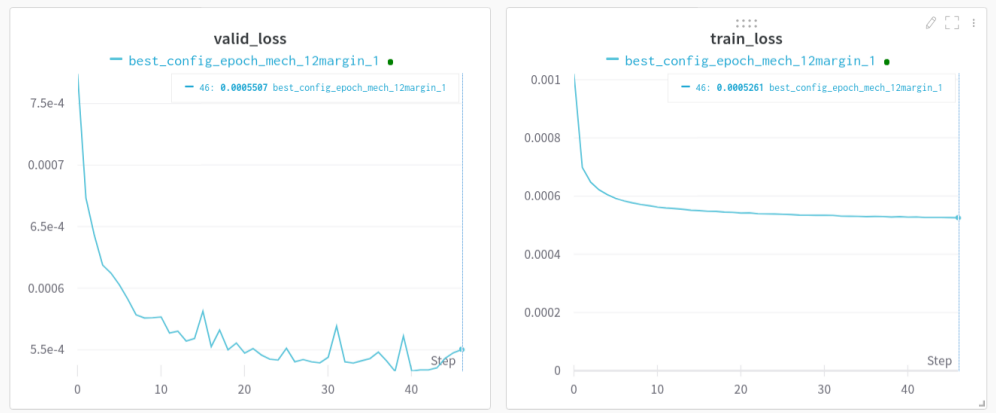

Figure 6: MSE Loss Curves for best configuration

Figure 7: 12-Margin Loss Curves for best configuration

4 Your Results on the Test Set(30 pts)

This part will be obtained by us using the estimations you will provide. Please tell us how should we run your code in case of a problem:

My implementation is a .ipynb python notebook. For hyperparameter tuning and extracting loss-curves I've made use of wandb (weights and biases). in order to tune hyperparameters and monitor plots more compactly. It just requires an API-key to initiate the terminal, then just need to set wandb.init as corresponding project name and user. I supposed that this is a common platform in research, but I can also provide my additional account's API-key in case you don't have any wandb:
api-key = 8bcc0e905b907134d50839c4ecd0a65d7ac754b6
replace wandb.init
wandb.init(project="vision", entity="aoao")
If this doesn't work out, you can open your wandb, create a project and replace the corresponding wandb parameters.
Also, I've written my model's estimations in estimations_test.npy which basically includes first 100 jpg images in test dataset. img_names.txt corresponds to [0.jpg,99.jpg] and it has a newline at the end each line which might be a bit different than the given line reading method in eval.py, because I couldn't fully understand how lines should be split.

In my test function, I had an issue while converting torch tensors to numpy arrays. My code successfully saves colorized test_inputs imgs to png from torch tensors. However, when I've read .npy and tried to cv2.imshow, the extracted picture from the numpy array was totally non-sense (like smaller and duplicated x9 times and grayed etc.), I've checked some forums and identified:
https://stackoverflow.com/questions/33725237/image-fromarray-changes-size.
I think there is a compatibility issue between numpy and image reading, that's why I tried transposing and reshaping to get the same images that are saved from torch tensors, and it did, but there might be a small bug that I'm not aware of. However, from the 12-margin error standpoint, it shouldn't matter much, if we've extracted or saved numpy arrays in the same manner, every test pixel should correspond to its label pixel. In addition, I have calculated 12-margin error in training and validation loops by using torch tensors which was much easier than numpy arrays in terms of saving and extracting.

Moreover, this notebook can also be run through google collab, at first I've worked with colab however google has banned me for using their servers too much that's why I had to run on my local gpu. That's why I've left some of the colab codes as commented, but they can be used easily in case needed, just need to put corresponding files and folders in your drive.
Also, I've started the terminal from src/wandb directory that's why my data-root is a bit different, than the given one.

In get_loaders function, Test_loader is not actually a data-loader, it's torch tensor, since we don't have any labels for test-set, I couldn't use the given dataloader, but instead created a 200x1x80x80 tensor and converted it to 200x1x1x80x80 since model runs on batches and it's handled that way. Also, I've assumed that there are 2000 images in test_inputs and hard-coded img extraction in this part. Test function takes this test_loader and for each image it calculates the output, saves test-img (this is optional not mentioned in requirements) with save_image which expects [0,1] scaled images, but for numpy estimations they're scaled to [0,255] interval.

Furthermore, my code calculates estimations for all 2000 images in test-set and saves as .npy, for first 100 I've just extracted first 100 of them and saved in terminator. So, my .npy submission includes first 100 jpgs and you will get 100x80x80x3, but if you run the code you'll get 2000x80x80x3.

5 Additional Comments and References

MSE losses are per datapoint not per batch which has been assured by counting number of examples in loops and scaling accumulated batch losses.

Throughout this assignment and experiments, I've made use of wandb for data visualization and logging. All of my experiments are logged in my wandb vision repository and some of the results for each part can be monitored. In case any further inspection is needed check out the following wandb experiment reports at urls:

https://wandb.ai/alpayozkan/vision/reports/Part-3–VmlldzoxNTAyNTQ3?accessToken=x7gwi5ni3z7sy6hfa5jioj
https://wandb.ai/alpayozkan/vision/reports/Part-2–VmlldzoxNTAyNTUw?accessToken=wy4vv5uzq2uif1b4jq6q
https://wandb.ai/alpayozkan/vision/reports/Part-1–VmlldzoxNTAyNTUz?accessToken=s4076hl3ue1agne51jubk

Additionally, I've included my model's predictions for test_inputs imgs as zip file.