1 Colorization as Regression (2pts): There are many ways to frame the problem of image colorization as a machine learning problem. A simple approach is to frame it as a regression problem, where we build a model to predict the RGB intensities at each pixel given the gray scale input. In this case, the outputs are continuous, and so squared error can be used to train the model.

A set of weights for such a model is included with the code. Read the code in `color_regression.py` and answer the following questions:

1. Describe the model RegressionCNN. How many convolution layers does it have? What are the filter sizes and number of filters at each layer? Construct a table or draw a diagram.

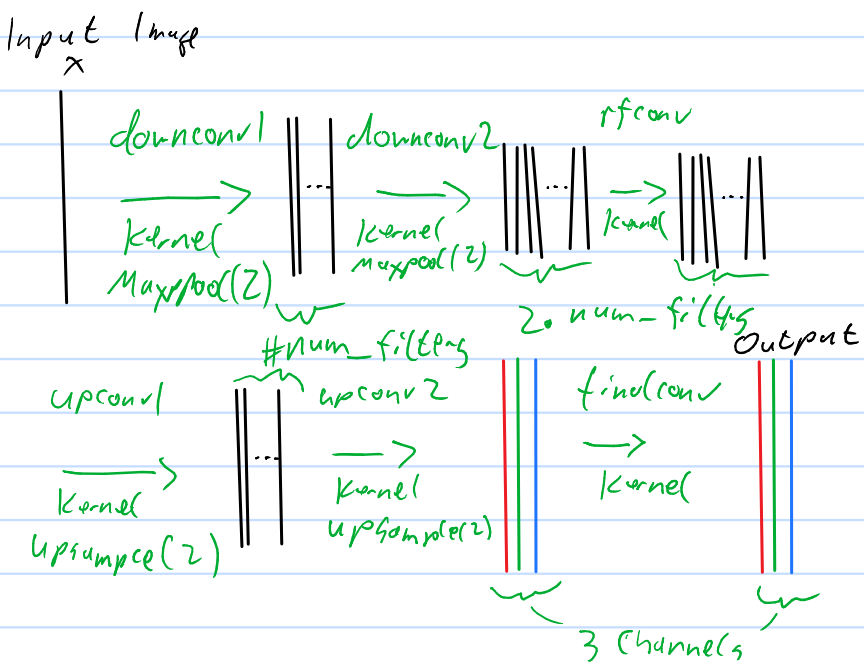
```
padding = kernel // 2

self.downconv1 = nn.Sequential(
    nn.Conv2d(1, num_filters, kernel_size=kernel, padding=padding),
    # MyConv2d(1, num_filters, kernel_size=kernel, padding=padding),
    nn.MaxPool2d(2),
    nn.BatchNorm2d(num_filters),
    nn.ReLU())
self.downconv2 = nn.Sequential(
    nn.Conv2d(num_filters, num_filters * 2, kernel_size=kernel, padding=padding),
    # MyConv2d(num_filters, num_filters * 2, kernel_size=kernel, padding=padding),
    nn.MaxPool2d(2),
    nn.BatchNorm2d(num_filters * 2),
    nn.ReLU())

self.rfconv = nn.Sequential(
    nn.Conv2d(num_filters * 2, num_filters*2, kernel_size=kernel, padding=padding),
    # MyConv2d(num_filters * 2, num_filters*2, kernel_size=kernel, padding=padding),
    nn.BatchNorm2d(num_filters * 2),
    nn.ReLU())

self.upconv1 = nn.Sequential(
    nn.Conv2d(num_filters * 2, num_filters, kernel_size=kernel, padding=padding),
    # MyConv2d(num_filters * 2, num_filters, kernel_size=kernel, padding=padding),
    nn.Upsample(scale_factor=2),
    nn.BatchNorm2d(num_filters),
    nn.ReLU())
self.upconv2 = nn.Sequential(
    nn.Conv2d(num_filters, 3, kernel_size=kernel, padding=padding),
    # MyConv2d(num_filters, 3, kernel_size=kernel, padding=padding),
    nn.Upsample(scale_factor=2),
    nn.BatchNorm2d(3),
    nn.ReLU())

self.finalconv = MyConv2d(3, 3, kernel_size=kernel)
```



I tried to draw it but it might be confusing, helped me to understand it though.

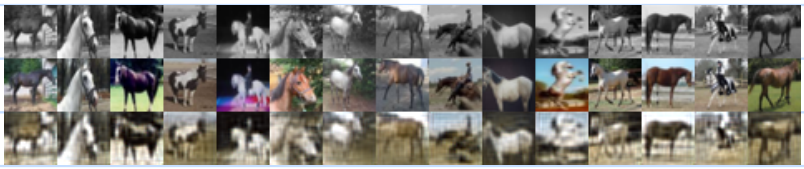
Here is a table for all convolutional layers, we have 6 in total.

Layer	Number of filters	Filtersize / Shape
downconv1	num_filters	[kernel, kernel]
downconv2	num_filters · 2	[num_filters, kernel, kernel]
rfconv	num_filters · 2	[2 · num_filters, kernel, kernel]
upconv1	num_filters	[2 · num_filters, kernel, kernel]
upconv2	3	[num_filters, kernel, kernel]
finalconv	3	[3, kernel, kernel]

It's important to understand, that per default each filter takes all input channels as input unless groups are specified. So the number of filters is equal to the number of output channels (unless groups are specified).

2. Run `color_regression.py`. This will load a set of trained weights and should generate some images showing validation outputs. Do the results look good to you? Why or why not?

Input
Truth
Output



The results look extremely "washed out" due to the models tendency to the means to minimize the loss (see next part).
Instead of using the full range of color, the colors in the output images only vary slightly around the mean color, which seems to be some shade of brown here.

3. A color space is a choice of mapping of colors into three-dimensional coordinates. Some colors could be close together in one color space, but further apart in others. The RGB color space is probably the most familiar to you, but most state of the art colorization models do not use RGB color space. The model used in `color_regression.py` computes squared error in RGB color space. How could using the RGB color space be problematic?

The problem is, that colors that are often close to each other in the images are also close to each other in RGB L2-distance, therefore it is harder for the model to distinguish between them and it tends to lean towards the mean to minimize the loss.

4. Most state of the art colorization models frame colorization as a classification problem instead of a regression problem. Why? (Hint: what does minimizing squared error encourage?)

Again: It encourages simply leaning towards the mean color to reduce the overall loss more significantly.

Framing the problem as a classification problem instead completely eliminates that issue, because each color, that is not predicted correctly, is equally "wrong", no matter how close it is to the correct color. So simply fitting the mean does not reduce the loss.

2 Colorization as Classification (2pts): We will select a subset of 24 colors and frame colorization as a pixel-wise classification problem, where we label each pixel with one of 24 colors. The 24 colors are selected using k-means clustering[a] over colors, and selecting cluster centers. This was already done for you, and cluster centers are provided in `color/color_kmeans*.npy` files. For simplicity we still measure distance in RGB space. This is not ideal but reduces the software dependencies for this assignment.
Read the code in colorization.py and answer the following questions:

1. Complete the model `CNN`. This model should have the same layers and convolutional filters as the `RegressionCNN`, with the exception of the output layer. Continue to use PyTorch layers like `nn.ReLU`, `nn.BatchNorm2d` and `nn.MaxPool2d`, however we will not use `nn.Conv2d`. We will use our own convolution layer `MyConv2d` included in the file to better understand its internals.

2. Run the following command:

   ```
   python colorization.py --model CNN --checkpoint weights/cnn_k3_f32.pkl --valid
   ```

   This will load a set of trained weights for your CNN model, and should generate some images showing the trained result. How do the result compare to the previous model?

Input
Truth
Output

The results seem much better to me, compared to the regression model. They span the whole color range. Clearly, the "washed out" problem due to mean tendency is resolved.

Of course, in practice we should use more than 24 colors, but that significantly increases the model size.

3 Skip Connections (3pts): A skip connection in a neural network is a connection which skips one or more layer and connects to a later layer. We will introduce skip connections.

1. Add a skip connection from the first layer to the last, second layer to the second last, etc. That is, the final convolution should have both the output of the previous layer and the initial gray scale input as input. This type of skip-connection is introduced by [2], and is called a "UNet". Following the `CNN` class that you have completed, complete the `__init__` and `forward` methods of the `UNet` class.

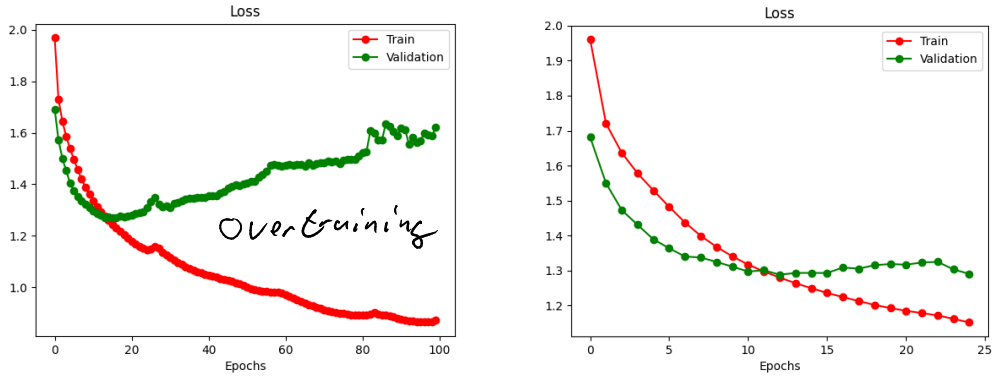
   Hint: You will need to use the function `torch.cat`.

2. Train the model for at least 5 epochs and plot the training curve using a batch size of 10:

   ```
   python colorization.py --model UNet --checkpoint unet_k3_f32.pkl -b 10 -e 5
   ```

   This should take around 25 minutes on a CPU based desktop computer. If you train for 25 epochs, the results will be a lot better (but not necessary for this question). If you have an NVIDIA GPU (and associated drivers) your code will automatically use the GPU make training significantly faster.

flag has to be set!



I wanted to make sure to see overfitting to find the ideal number of epochs to train for (hyper parameter).

3. How does the result compare to the previous model? Did skip connections improve the validation loss and accuracy? Did the skip connections improve the output qualitatively? How? Give at least two reasons why skip connections might improve the performance of our CNN models.

   Note: We recommend that you answer this question with the pre-trained weights provided, especially if you did not train for 25 epochs in the previous question. Clearly state if you choose to do so in your writeup. You can load the weights using the following command:

   ```
   python colorization.py --model UNet --checkpoint weights/unet_k3_f32.pkl --valid
   ```

My trained network:

```
Evaluating Model UNet: unet_k3_f32.pkl
Val Loss: 1.2724, Val Acc: 52.6%
Sample output available at: outputs/eval_UNet.png
```

Input
Truth
Output

Input passed through

Provided weights:

```
Evaluating Model UNet: weights/unet_k3_f32.pkl
Val Loss: 1.3659, Val Acc: 48.0%
Sample output available at: outputs/eval_UNet.png
```

some here

Before:

```
Evaluating Model CNN: weights/cnn_k3_f32.pkl
Val Loss: 1.5881, Val Acc: 41.1%
Sample output available at: outputs/eval_CNN.png
```

better

better

Skip connections significantly improve loss and accuracy. I personally don't feel like the output improved subjectively. Clearly, the model tends to pass through the grey scale input, when it is unsure.

Two reasons why it might improve performance:

1. The model is larger. It has more parameters (the skip connection weights). It's not really fair to compare them directly for that reason. Overall, it's a tiny model, so simply increasing its size in whatever way helps a lot.

2. Idea behind skip connections: It might help the later layers to "remember" the initial input to combine the high level features.

4 Dilated Convolution (1pt): In class, we discussed convolutional filters that are contiguous – that is, they act on input pixels that are next to one another. However, we can dilated the filters so that they have spaces between input pixels [3], as in Figure 1.

A dilated convolution is a way to increase the size of the receptive field without increasing the number of weights.

1. We have been using 3×3 filters for most of the assignment. Compare between:
   (a) 3×3 convolution
   (b) 5×5 convolution
   (c) 3×3 convolution with dilation 1

   How many weights (excluding biases) do they have? What is the size of the receptive field of each?

	Filter shape	# weights	receptive field	
(a)	C × 3 × 3	9	3 × 3	9
(b)	C × 5 × 5	25	5 × 5	25
(c)	C × 3 × 3	9	5 × 5	25

input channel(s) — for C = 1

2. The `DilatedUNet` class replaces the middle convolution with a dilated convolution with dilation 1. Why we might choose to add dilation here, instead of another convolution? (Hint: think about impact on receptive field.)

It will significantly impact the receptive field for combining high level features at the beginning of the coloring process. For example: The model might be confident that the top right corner is sky and should be blue. But the top left corner is less obvious, increasing the receptive field might help to identify such long range correlations.

5 Visualizing Intermediate Activations (1pt): We will visualize the intermediate activations for several inputs. The python script `activation.py` has already been written for you, and outputs.

1. Visualize the activations of the CNN for a few test examples. How are the activation in the first few layers different from the later layers? You do not need to attach the output images to your writeup, only descriptions of what you see.

   `python activations.py <num> --model CNN --checkpoint weights/cnn_k3_f32.pkl`

The filters in the first layers are low level features, that are focused on small individual portions in the image

The later layers combine those lower level features into high level features. The activations are all over the image and not just focused one some specific region (where the low level features are located).

2. Visualize the activations of the UNet for a few test examples. How do the activations differ from the CNN activations?

   `python activations.py <num> --model UNet --checkpoint weights/unet_k3_f32.pkl`

The low level features are very similar, but the high level features differ, taking more information from the skip connections into account.

6 Conceptual Questions (1pt):

1. Data augmentation can be helpful when the training set size is small. Which of these data augmentation methods do you think would have been helpful for our CNN models, and why?
   (a) Augmenting via flipping each image upside down
   (b) Augmenting via flipping each image left to right
   (c) Augmenting via shifting each image one pixel left / right
   (d) Augmenting via shifting each image one pixel up / down
   (e) Augmenting via using other of the CIFAR-10 classes

(a) Probably not helpful because horses are not symmetric in the horizontal plane and we have no need for coloring upside-down horses.

(b) This will most likely be helpful, because the horses and landscapes are symmetrical in the vertical plane, so the same filters can be trained this way.

(c) and (d) Could both potentially be helpful to avoid positional bias in the data. For example I've noticed two filters just scan the border pixels (especially the lower border).

(e) If we really didn't have enough horse pictures, some other classes might be helpful. But only those that are similar, like dancers maybe. Or just landscapes (for the background). It would not be helpful to train on entirely different subjects/objects, like cats or cars (Unless of course we want a more general model, not just for horses).

2. We also did not tune any hyperparameters for this assignment. What are some hyperparameters that could be tuned? List five.

The optimizer and its settings, like learning rate and decay. Also batch size. Training time (number of epochs). And of course the model itself, especially its size (number of filters, kernel size, filter groups, number of layers, etc.).

7 Extra Points: Implementing Dilated Convolution (1pt) This is an optional portion of the assignment where we will implement the Dilated UNet.

1. Dilations are included as a parameter in PyTorch `nn.Conv2d` module and `F.conv2d` function. For the purpose of this assignment we will not use the native implementation. Instead, we will re-implement dilated convolution by directly manipulating the filters using PyTorch tensor manipulation, and ensuring that the implemented dilated convolution is consistent with Fig.1. The purpose is to (a) better understand PyTorch and (b) better understand what the filters look like. Implemented the `forward()` method of `MyDilatedConv2d` function, without using the dilation parameter of `F.conv2d`.

   Hint: You can do a lot with PyTorch tensors that you can do with NumPy arrays, like slicing `tensor[2:4]`, and assigning values to slices `tensor[2:4] = 0`. Bear in mind that PyTorch will need to backpropagate through whatever operations that you use to manipulate the tensors. You may also find the function `F.upsample` to be helpful.

   Note: If you cannot complete this problem, use the dilation parameter of `F.conv2d`. You will not receive credit for this section, but it will allow you to continue to the next section.

2. Using the pre-trained weights provided, use the following command to load the weights and run validation step:

   ```
   python colorization.py --model DUNet --checkpoint weights/dunet_k3_f32.pkl --valid
   ```

   How does the result compare to the previous model, quantitatively (loss and accuracy) and qualitatively? You may or may not see an improvement in this case. In what circumstances might dilation be more helpful?

no time

```
Evaluating Model DUNet: weights/dunet_k3_f32.pkl
Val Loss: 1.4554 Val Acc: 45.4%
Sample output available at: outputs/eval_DUNet.png
```

It's worse than the UNet but better than the vanilla CNN when it comes to loss and accuracy. Subjectively I can't really tell any significant difference in the output.

– Optional: What part of this homework was the most illuminating, and the part that was difficult?

I found this one generally a lot easier than the previous one.

More illuminating: Working with PyTorch, I'm used to TensorFlow.

Hardest: The resolution of the images is so small that it's hard for me to see differences even when the loss/accuracy is drastically different. I think you could update it for next year with higher resolution. Even training for 100 epochs only took a few minutes for me on the HPCC K80.