```
!rm -rf /content/Landmark-classification-and-tagging
!git clone https://github.com/SwastikGorai/Landmark-classification-and-tagging
```

```
Cloning into 'Landmark-classification-and-tagging'...
remote: Enumerating objects: 6404, done.
remote: Counting objects: 100% (24/24), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 6404 (delta 10), reused 17 (delta 5), pack-reused 6380
Receiving objects: 100% (6404/6404), 715.32 MiB | 15.79 MiB/s, done.
Resolving deltas: 100% (13/13), done.
```

```
!mv -v Landmark-classification-and-tagging/* /content/
```

```
renamed 'Landmark-classification-and-tagging/app.ipynb' -> '/content/app.ipynb'
renamed 'Landmark-classification-and-tagging/cnn_from_scratch.ipynb' -> '/content/cnn_from_scratch.ipynb'
renamed 'Landmark-classification-and-tagging/mean_and_std.pt' -> '/content/mean_and_std.pt'
renamed 'Landmark-classification-and-tagging/README.md' -> '/content/README.md'
renamed 'Landmark-classification-and-tagging/requirements.txt' -> '/content/requirements.txt'
renamed 'Landmark-classification-and-tagging/src' -> '/content/src'
renamed 'Landmark-classification-and-tagging/static_images' -> '/content/static_images'
renamed 'Landmark-classification-and-tagging/transfer_learning.ipynb' -> '/content/transfer_learning.ipynb'
```

Convolutional Neural Networks

Project: Write an Algorithm for Landmark Classification

Introduction

The project folder has the following structure:

- In the main directory you have this notebook, `cnn_from_scratch.ipynb`, that contains the instruction and some questions you will have to answer. Follow this notebook and complete the required sections in order.
- In the `src/` directory you have several source files. As instructed in this notebook, you will open and complete those files, then come back to this notebook to execute some tests that will verify what you have done. While these tests don't guarantee that your work is bug-free, they will help you finding the most obvious problems so you will be able to proceed to the next step with confidence.
- Sometimes you will need to restart the notebook. If you do so, remember to execute also the cells containing the code you have already completed starting from the top, before you move on.

> Once you have completed all the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to HTML, all the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. If you are using Jupyter Lab, you can use **File -> Export Notebook as -> Export Notebook to HTML**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a '**Question**' header. Carefully read each question and provide thorough answers in the following text boxes that begin with '**Answer:**'. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Designing and training a CNN from scratch

In this notebook, you will create a CNN that classifies landmarks. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 50%.

Although 50% may seem low at first glance, it seems more reasonable after realizing how difficult of a problem this is. Many times, an image that is taken at a landmark captures a fairly mundane image of an animal or plant, like in the following picture.

Just by looking at that image alone, would you have been able to guess that it was taken at the Haleakalā National Park in Hawaii?

An accuracy of 50% is significantly better than random guessing, which would provide an accuracy of just 2% (100% / 50 classes). In Step 2 of this notebook, you will have the opportunity to greatly improve accuracy by using transfer learning to create a CNN.

Experiment with different architectures, hyperparameters, training strategies, and trust your intuition. And, of course, have fun!

▶▶ Step 0: Setting up

The following cells make sure that your environment is setup correctly, download the data if you don't have it already, and also check that your GPU is available and ready to go. You have to execute them every time you restart your notebook.

```
# Install requirements
!pip install -r requirements.txt | grep -v "already satisfied"
```

```
Collecting livelossplot (from -r requirements.txt (line 10))
  Downloading livelossplot-0.5.5-py3-none-any.whl.metadata (8.7 kB)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch->-r requirements.txt (line 6))
```

```
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.2.106 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-nccl-cu12==2.20.5 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl.metadata (1.8 kB)
Collecting nvidia-nvtx-cu12==12.1.105 (from torch->-r requirements.txt (line 6))
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl.metadata (1.7 kB)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-cu12==11.4.5.107->torch->-r requirements.txt (line 6))
  Downloading nvidia_nvjitlink_cu12-12.5.82-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting jedi>=0.16 (from ipython>=4.0.0->ipywidgets->-r requirements.txt (line 9))
  Downloading jedi-0.19.1-py2.py3-none-any.whl.metadata (22 kB)
Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6 MB)
Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (14.1 MB)
Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (23.7 MB)
Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (823 kB)
Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7 MB)
Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6 MB)
Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5 MB)
Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl (124.2 MB)
Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl (196.0 MB)
Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl (176.2 MB)
Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
Downloading livelossplot-0.5.5-py3-none-any.whl (22 kB)
Downloading jedi-0.19.1-py2.py3-none-any.whl (1.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 69.2 MB/s eta 0:00:00
Downloading nvidia_nvjitlink_cu12-12.5.82-py3-none-manylinux2014_x86_64.whl (21.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.3/21.3 MB 13.9 MB/s eta 0:00:00
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12,
Successfully installed jedi-0.19.1 livelossplot-0.5.5 nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.1.105 nvidia-cuda-nvr
```

```python
from src.helpers import setup_env

# If running locally, this will download dataset (make sure you have at
# least 2 Gb of space on your hard drive)
setup_env()
```

```
GPU available
Downloading and unzipping https://udacity-dlnfd.s3-us-west-1.amazonaws.com/datasets/landmark_images.zip. This will take a while..
done
Reusing cached mean and std
```

▶▶ Step 1: Data

In this and the following steps we are going to complete some code, and then execute some tests to make sure the code works as intended.

Open the file `src/data.py`. It contains a function called `get_data_loaders`. Read the function and complete all the parts marked by `YOUR CODE HERE`. Once you have finished, test that your implementation is correct by executing the following cell (see below for what to do if a test fails):

```
!pytest -vv src/data.py -k data_loaders
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 4 items / 1 deselected / 3 selected

src/data.py::test_data_loaders_keys PASSED                                                   [ 33%]
```

```
src/data.py::test_data_loaders_output_type PASSED                                      [ 66%]
src/data.py::test_data_loaders_output_shape PASSED                                     [100%]

================================= 3 passed, 1 deselected in 2.93s ==================================
```

You should see something like:

```
src/data.py::test_data_loaders_keys PASSED                                  [ 33%]
src/data.py::test_data_loaders_output_type PASSED                           [ 66%]
src/data.py::test_data_loaders_output_shape PASSED                          [100%]

====================== 3 passed, 1 deselected in 1.81s =======================
```

If all the tests are `PASSED`, you can move to the next section.

What to do if tests fail When a test fails, `pytest` will mark it as `FAILED` as opposed to `PASSED`, and will print a lot of useful output, including a message that should tell you what the problem is. For example, this is the output of a failed test:

```
    def test_data_loaders_keys(data_loaders):

       assert set(data_loaders.keys()) == {"train", "valid", "test"}
E        AssertionError: assert {'tes', 'train', 'valid'} == {'test', 'train', 'valid'}
E          Extra items in the left set:
E          'tes'
E          Full diff:
E          - {'test', 'train', 'valid'}
E          + {'tes', 'train', 'valid'}
E          ?                          +++++++

src/data.py:171: AssertionError
------------- Captured stdout setup ----------------------------------------------
Reusing cached mean and std for landmark_images
Dataset mean: tensor([0.4638, 0.4725, 0.4687]), std: tensor([0.2699, 0.2706, 0.3018])
=========== short test summary info ===============================================
FAILED src/data.py::test_data_loaders_keys - AssertionError: The keys of the data_loaders dictionary should be train, valid and tes
```

In the `short test summary info` you can see a short description of the problem. In this case, the dictionary we are returning has the wrong keys. Going above a little, you can see that the test expects `{'test', 'train', 'valid'}` while we are returning `{'tes', 'train', 'valid'}` (there is a missing `t`). So we can go back to our function, fix that problem and test again.

In other cases, you might get an error like:

```
    def _conv_forward(self, input: Tensor, weight: Tensor, bias: Optional[Tensor]):
        if self.padding_mode != 'zeros':
            return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode=self.padding_mode),
                            weight, bias, self.stride,
                            _pair(0), self.dilation, self.groups)
        return F.conv2d(input, weight, bias, self.stride,
                        self.padding, self.dilation, self.groups)
E       RuntimeError: Input type (torch.cuda.FloatTensor) and weight type (torch.FloatTensor) should be the same

../../../../miniconda3/envs/udacity_starter/lib/python3.7/site-packages/torch/nn/modules/conv.py:440: RuntimeError
```

Looking at the stack trace you should be able to understand what it is going on. In this case, we forgot to add a `.cuda()` to some tensor. For example, the model is on the GPU, but the data aren't.

Question: Describe your chosen procedure for preprocessing the data.

- How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?
- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: My code first resizes the image to 256 and then crops to 224. I picked 224 as the input size because it is the recommended input size for using pytorch's pre-trained models. I did decide to augment the dataset via RandAugment, a typical set of augmentations for natural images. I added this augmentation with the goal of improving my model's robustness, thus improving test accuracy.

Visualize a Batch of Training Data

Go back to `src/data.py` and complete the function `visualize_one_batch` in all places with the `YOUR CODE HERE` marker. After you're done, execute the following cell and make sure the test `src/data.py::test_visualize_one_batch` is `PASSED`:

```
!pytest -vv src/data.py -k visualize_one_batch
```

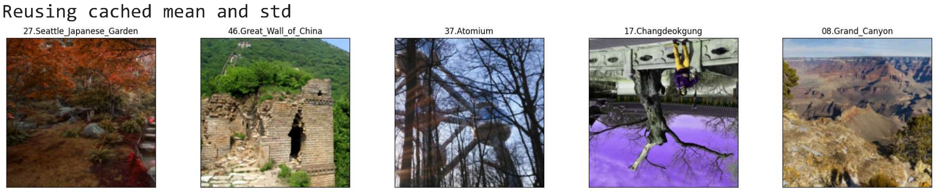
```
======================================== test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 4 items / 3 deselected / 1 selected

src/data.py::test_visualize_one_batch PASSED                                                  [100%]

================================= 1 passed, 3 deselected in 3.02s ==================================
```

We can now use the code we just completed to get a batch of images from your train data loader and look at them.

Visualizing the output of your data loader is a great way to ensure that your data loading and preprocessing (including transforms such as rotations, translations, color transforms...) are working as expected.

```
%matplotlib inline
from src.data import visualize_one_batch, get_data_loaders

# use get_data_loaders to get the data_loaders dictionary. Use a batch_size
# of 5, a validation size of 0.01 and num_workers=-1 (all CPUs)
data_loaders = get_data_loaders(batch_size=5, valid_size=0.01, num_workers=-1)

visualize_one_batch(data_loaders)
```

```
Reusing cached mean and std
Dataset mean: tensor([0.4638, 0.4725, 0.4687]), std: tensor([0.2697, 0.2706, 0.3017])
Reusing cached mean and std
```



Step 2: Define model

Open `src/model.py` and complete the `MyModel` class filling in all the `YOUR CODE HERE` sections. After you're done, execute the following test and make sure it passes:

```
!pytest -vv src/model.py
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 1 item

src/model.py::test_model_construction PASSED                                                  [100%]

======================================== 1 passed in 2.96s =========================================
```

Question: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer: I decided to use 5 convolutional layers so that my model could be sufficiently expressive. I used dropout layers to reduce my model's tendency to overfit the training data. I made my model output a 50-dimensional vector to match with the 50 available landmark classes.

Step 3: define loss and optimizer

Open `src/optimization.py` and complete the `get_loss` function, then execute the test and make sure it passes:

```
!pytest -vv src/optimization.py -k get_loss
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 7 items / 6 deselected / 1 selected

src/optimization.py::test_get_loss PASSED                                                     [100%]

================================= 1 passed, 6 deselected in 1.40s ==================================
```

Then, in the same file, complete the `get_optimizer` function then execute its tests, and make sure they all pass:

```
!pytest -vv src/optimization.py -k get_optimizer
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 7 items / 1 deselected / 6 selected

src/optimization.py::test_get_optimizer_type PASSED                                         [ 16%]
src/optimization.py::test_get_optimizer_is_linked_with_model PASSED                         [ 33%]
src/optimization.py::test_get_optimizer_returns_adam PASSED                                 [ 50%]
src/optimization.py::test_get_optimizer_sets_learning_rate PASSED                           [ 66%]
src/optimization.py::test_get_optimizer_sets_momentum PASSED                                [ 83%]
src/optimization.py::test_get_optimizer_sets_weight_decat PASSED                            [100%]

================================= 6 passed, 1 deselected in 2.12s ==================================
```

Step 4: Train and Validate the Model

> Testing ML code is notoriously difficult. The tests in this section merely exercise the functions you are completing, so it will help you catching glaring problems but it won't guarantee that your training code is bug-free. If you see that your loss is not decreasing, for example, that's a sign of a bug or of a flawed model design. Use your judgement.

Open `src/train.py` and complete the `train_one_epoch` function, then run the tests:

```
!pytest -vv src/train.py -k train_one_epoch
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 4 items / 3 deselected / 1 selected

src/train.py::test_train_one_epoch PASSED                                                   [100%]

================================= 1 passed, 3 deselected in 5.75s ==================================
```

Now complete the `valid` function, then run the tests:

```
!pytest -vv src/train.py -k valid_one_epoch
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 4 items / 3 deselected / 1 selected

src/train.py::test_valid_one_epoch PASSED                                                   [100%]

================================= 1 passed, 3 deselected in 5.09s ==================================
```

Now complete the `optimize` function, then run the tests:

```
!pytest -vv src/train.py -k optimize
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 4 items / 3 deselected / 1 selected

src/train.py::test_optimize PASSED                                                           [100%]

================================= 1 passed, 3 deselected in 7.30s ==================================
```

Finally, complete the `test` function then run the tests:

```
!pytest -vv src/train.py -k one_epoch_test
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 4 items / 3 deselected / 1 selected

src/train.py::test_one_epoch_test PASSED                                                     [100%]

================================= 1 passed, 3 deselected in 5.69s ==================================
```

Step 5: Putting everything together

Allright, good job getting here! Now it's time to see if all our hard work pays off. In the following cell we will train your model and validate it against the validation set.

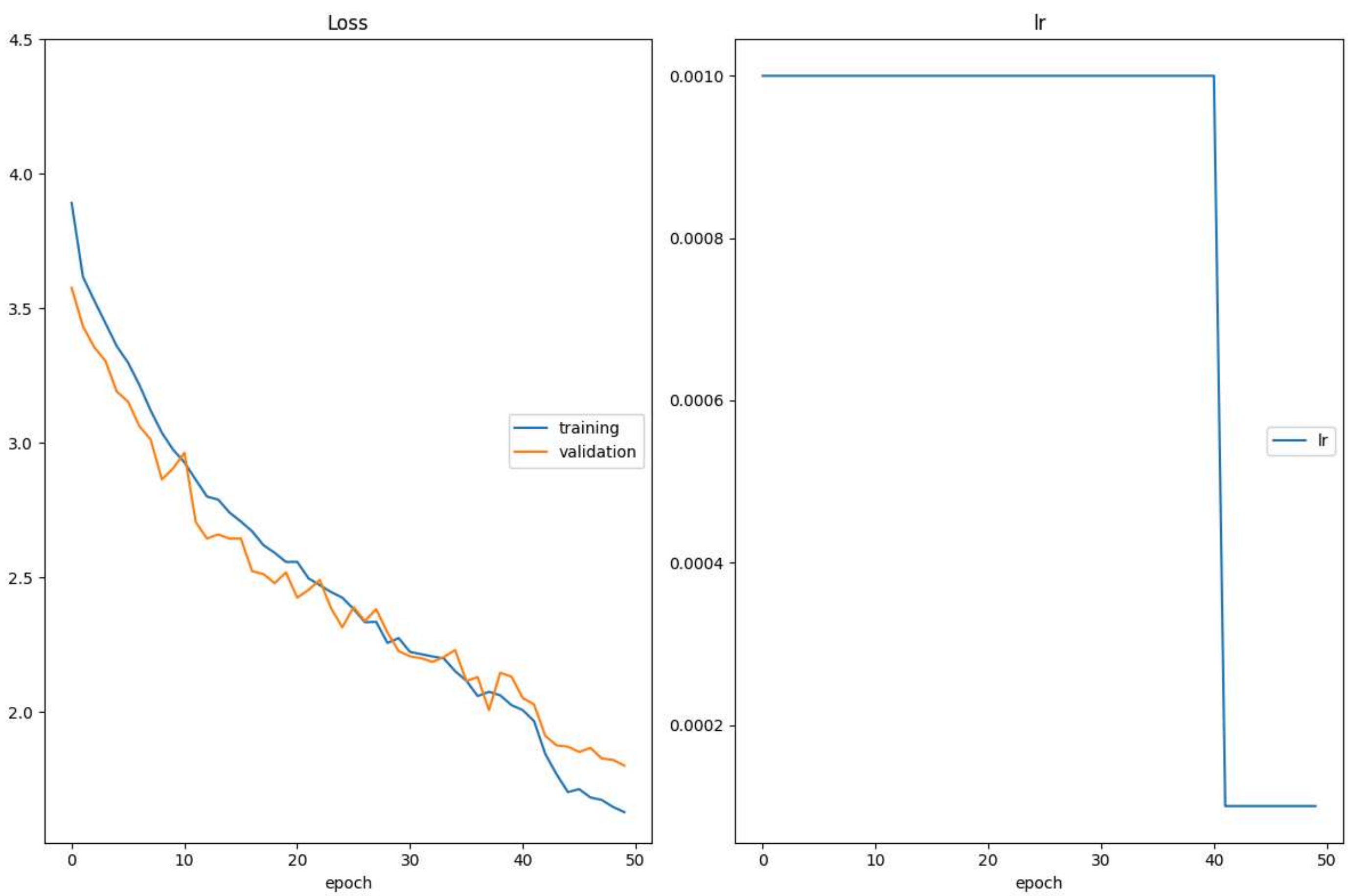
Let's start by defining a few hyperparameters. Feel free to experiment with different values and try to optimize your model:

```
batch_size = 64        # size of the minibatch for stochastic gradient descent (or Adam)
valid_size = 0.2       # fraction of the training data to reserve for validation
num_epochs = 50       # number of epochs for training
num_classes = 50       # number of classes. Do not change this
dropout = 0.4          # dropout for our model
learning_rate = 0.001  # Learning rate for SGD (or Adam)
opt = 'adam' #sgd            # optimizer. 'sgd' or 'adam'
weight_decay = 0.001     # regularization. Increase this to combat overfitting
```

```
import torch
import gc
torch.cuda.empty_cache()
gc.collect()
```

```
13763
```

```
from src.data import get_data_loaders
from src.train import optimize
from src.optimization import get_optimizer, get_loss
from src.model import MyModel

# get the data loaders using batch_size and valid_size defined in the previous
# cell
# HINT: do NOT copy/paste the values. Use the variables instead
data_loaders = get_data_loaders(batch_size=batch_size, valid_size=valid_size)

# instance model MyModel with num_classes and drouput defined in the previous
# cell
model = MyModel(num_classes=num_classes, dropout=dropout)

# Get the optimizer using get_optimizer and the model you just created, the learning rate,
# the optimizer and the weight decay specified in the previous cell
optimizer = get_optimizer(
    model=model,
    learning_rate=learning_rate,
    optimizer=opt,
    weight_decay=weight_decay
)

# Get the loss using get_loss
loss = get_loss()

optimize(
    data_loaders,
    model,
    optimizer,
    loss,
    n_epochs=num_epochs,
    save_path="checkpoints/best_val_loss.pt",
    interactive_tracking=True
)
```



Step 6: testing against the Test Set

> only run this *after* you have completed hyperpameter optimization. Do not optimize hyperparameters by looking at the results on the test set, or you might overfit on the test set (bad, bad, bad)

Run the code cell below to try out your model on the test dataset of landmark images. Ensure that your test accuracy is greater than 50%.

```
from src.data import get_data_loaders
from src.train import optimize
from src.optimization import get_optimizer, get_loss
from src.model import MyModel

data_loaders = get_data_loaders(batch_size=batch_size, valid_size=valid_size)
loss = get_loss()
```

```
Reusing cached mean and std
Dataset mean: tensor([0.4638, 0.4725, 0.4687]), std: tensor([0.2697, 0.2706, 0.3017])
```

```
# load the model that got the best validation accuracy
from src.train import one_epoch_test
from src.model import MyModel
import torch

model = MyModel(num_classes=num_classes, dropout=dropout)

# YOUR CODE HERE: load the weights in 'checkpoints/best_val_loss.pt'

checkpoint_path = "checkpoints/best_val_loss.pt"

model.load_state_dict(torch.load(checkpoint_path))
model.eval()


# Run test
one_epoch_test(data_loaders['test'], model, loss)
```

```
Testing: 100%|██████████████████████████████| 20/20 [00:09<00:00,  2.10it/s]Test Loss: 1.695569


Test Accuracy: 57% (716/1250)

1.6955690890550612
```

Step 7: Export using torchscript

Great job creating your CNN models! Now that you have put in all the hard work of creating accurate classifiers, let's export it so we can use it in our app.

But first, as usual, we need to complete some code!

Open `src/predictor.py` and fill up the missing code, then run the tests:

```
!pytest -vv src/predictor.py
```

```
======================================= test session starts ========================================
platform linux -- Python 3.10.12, pytest-7.4.4, pluggy-1.5.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: typeguard-4.3.0, anyio-3.7.1
collected 1 item

src/predictor.py::test_model_construction PASSED                                             [100%]

======================================== 1 passed in 3.45s =========================================
```

Allright, now we are ready to export our model using our Predictor class:

```
# NOTE: you might need to restart the notebook before running this step
# If you get an error about RuntimeError: Can't redefine method: forward on class
# restart your notebook then execute only this cell
from src.predictor import Predictor
from src.helpers import compute_mean_and_std
from src.model import MyModel
from src.data import get_data_loaders
import torch

data_loaders = get_data_loaders(batch_size=1)

# First let's get the class names from our data loaders
class_names = data_loaders["train"].dataset.classes

# Then let's move the model_transfer to the CPU
# (we don't need GPU for inference)
model = MyModel(num_classes=50, dropout=0.5).cpu()

# Let's make sure we use the right weights by loading the
# best weights we have found during training
# NOTE: remember to use map_location='cpu' so the weights
# are loaded on the CPU (and not the GPU)
model.load_state_dict(torch.load("checkpoints/best_val_loss.pt", map_location='cpu'))

# Let's wrap our model using the predictor class
mean, std = compute_mean_and_std()
predictor = Predictor(model, class_names, mean, std).cpu()

# Export using torch.jit.script
scripted_predictor = torch.jit.script(predictor)

scripted_predictor.save("checkpoints/original_exported.pt")
```

```
Reusing cached mean and std
Dataset mean: tensor([0.4638, 0.4725, 0.4687]), std: tensor([0.2697, 0.2706, 0.3017])
Reusing cached mean and std
```

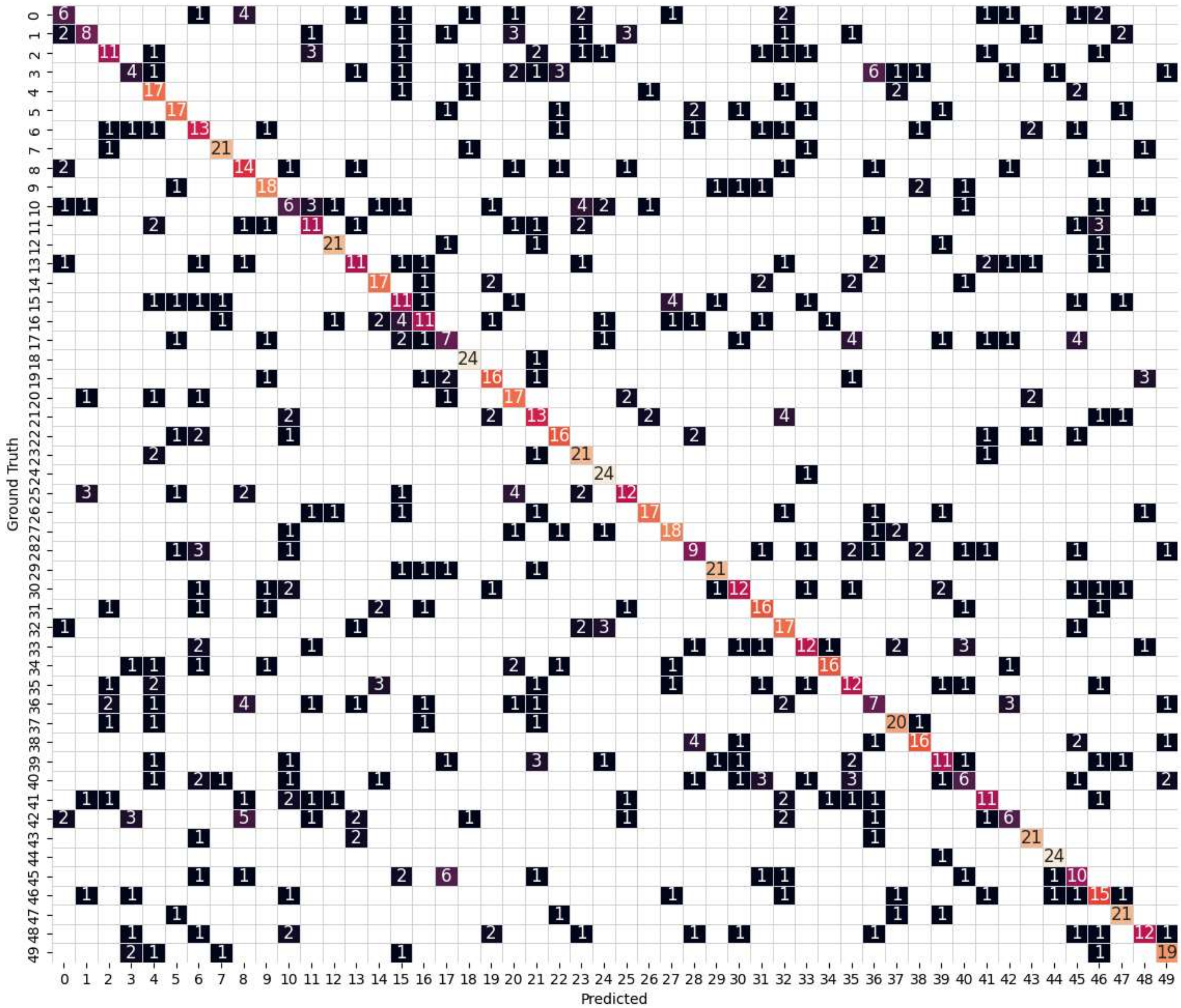
Now let's make sure the exported model has the same performance as the original one, by reloading it and testing it. The Predictor class takes different inputs than the non-wrapped model, so we have to use a specific test loop:

```
import torch

# Load using torch.jit.load
model_reloaded =   torch.jit.load("checkpoints/original_exported.pt")


from src.predictor import predictor_test

pred, truth = predictor_test(data_loaders['test'], model_reloaded)
```

```
100%|██████████████████████████████| 1250/1250 [00:39<00:00, 32.04it/s]Accuracy: 0.5728
```

Finally, let's have a look at the confusion matrix of the model we are going to use in production:

```
from src.helpers import plot_confusion_matrix

plot_confusion_matrix(pred, truth)
```




```
Start coding or generate with AI.
```