

[◀ Return to Classroom](#)[DISCUSS ON STUDENT HUB](#)

Landmark Classification & Tagging for Social Media

REVIEW

HISTORY

Requires Changes

2 specifications require changes

Hi, I am your new reviewer and this is the first time I am reviewing your submission.

You've done an excellent job at implementing the entire project! 🙌

However, there are some changes that you need to make before you can pass the project.

As a reviewer I am required to follow Udacity's guidelines when evaluating projects and hence I had to mark some requirements that need more work from your end.

Please go through the review to see what changes you need to make.

Once you make the desired changes and resubmit the project, you will pass the project and get a step closer to finishing your nanodegree.

Wishing you good luck! 😊

Some general suggestions

Use of assertions and Logging:

- Consider using [Python assertions](#) for sanity testing - assertions are great for catching bugs. This is especially true of a dynamically type-checked language like Python where a wrong variable type or shape can cause errors at runtime
- Logging is important for long-running applications. Logging done right produces a report that can be analyzed to debug errors and find crucial information. There could be different levels of logging or logging tags that can be used to filter messages most relevant to someone. Messages can be written to the terminal using `print()` or saved to file, for example using the [Logger module](#). Sometimes it's worthwhile to catch and log exceptions during a long-running operation so that the operation itself is not aborted.

Debugging:

- Check out this guide on [debugging in python](#)

Reproducibility:

- Reproducibility is perhaps the biggest issue in machine learning right now. With so many moving parts present in the code (data, hyperparameters, etc) it is imperative that the instructions and code make it easy for anyone to get exactly the same results (just imagine debugging an ML pipeline where the data changes every time and so you cannot get the same result twice).
- Also consider using random seeds to make your data more reproducible.

Optimization and Profiling:

- Monitoring progress and debugging with [Tensorboard](#): This tool can log detailed information about the model, data, hyperparameters, and more. Tensorboard can be used with Pytorch as well.
- Profiling with Pytorch: [Pytorch's profiler](#) can be used to break down profiling information by operations (convolution, pooling, batch norm) and identify performance bottlenecks. The performance traces can be viewed in the browser itself. The profiler is a great tool for quickly comparing GPU vs CPU speedups for example.

Files Submitted

The submission includes the required notebook file and HTML file. When the HTML file is created, all the code cells in the notebook need to have been run so that reviewers can see the final implementation and output.

All files are included in the submission zip

 Jupyter Notebook HTML Report Suggestion:

You can export your conda environment into `environment.yaml` file so that you can recreate your conda environment later while practicing on your own system. Use the following command -

```
conda env export -f environment.yaml
```


Step 1: Create a CNN to Classify Landmarks (from Scratch)

The submission randomly splits the images at `landmark_images/train` into train and validation sets. The submission then creates a data loader for the created train set, a data loader for the created validation set, and a data loader for the images at `landmark_images/test`.

```
# [-] data transform
data_transform = transforms.Compose([
    transforms.Resize(32),           # [.] resize
    transforms.CenterCrop(32),       # [.] center crop
    transforms.RandomRotation(10),   # [.] random rotate
    transforms.RandomHorizontalFlip(), # [.] random flip
    transforms.ToTensor(),           # [.] convert to tensor
    transforms.Normalize(            # [.] normalize RGB values
        (0.5, 0.5, 0.5),
        (0.5, 0.5, 0.5)
    ),
])

# [-] train and test data
train_data = datasets.ImageFolder( # [.] train data
    train_dir,
    transform=data_transform,
)

test_data = datasets.ImageFolder( # [.] test data
    test_dir,
    transform=data_transform,
)
```

 Please note that augmentations are only supposed to be applied to training

data.

For validation and testing data, some of the only allowable transformations are `CenterCrop` and `Resize` for cropping and resizing. Augmentations such as `RandomResizedCrop` are not supposed to be used. For more details on why random augmentations are not permitted on test data please checkout this [StackOverflow discussion](#)

✗ In your current implementation there are two issues:

- 1) You are using the same transform for training and testing data which causes augmentations to be applied to testing data. This is incorrect.
- 2) You are subsampling your validation data from the training data, which means that augmentations are also applied to validation data.

Image Pre-processing for Model Training

Validation Data

Should *never* be augmented!

- Just like how we didn't create an artificial balance of positive and negative cases in our validation set...
- ...We also *never* want to augment our validation data
- We should still normalize so that intensity values are close to zero
- **But we want our validation data to reflect the real world and only be comprised of real data**

⚠ To fix this, you should first specify the transforms for the datasets and then apply the sampler. I am sharing a sample code snippet below but you should modify it as per your own needs.

```
train_transform = transforms.Compose([transforms.Resize(224),
                                     transforms.CenterCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

```
valid_test_transform = transforms.Compose([transforms.Resize(224),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])


train_data = datasets.ImageFolder('/data/landmark_images/train', transform=train_transform)
valid_data = datasets.ImageFolder('/data/landmark_images/train', transform=valid_test_transform)
test_data = datasets.ImageFolder('/data/landmark_images/test', transform=valid_test_transform)

num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, sampler=valid_sampler)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)

loaders_scratch = {'train': train_loader, 'valid': valid_loader, 'test': test_loader}
```

 Suggestion: A better way to generate validation data is to use the `splitfolders` module to split the training dataset into training and validation sets.

```
import splitfolders

output = splitfolders.ratio("/data/landmark_images/train", output="output", seed=1337, ratio=(.8, .2), group_prefix=None)
```

Answer describes each step of the image preprocessing and augmentation. Augmentation (cropping, rotating, etc.) is not a requirement.

✖ You may want to update your answer based on the changes suggested above.

Question 1: 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:

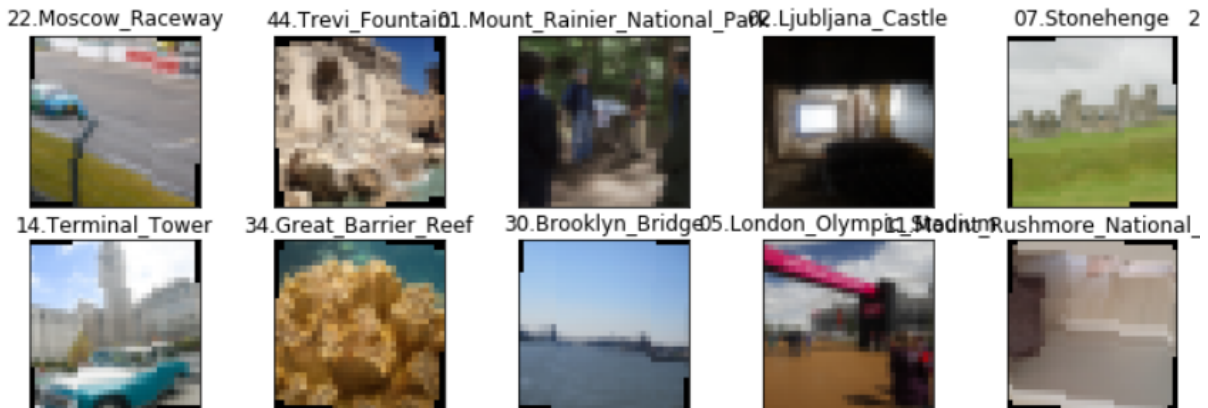
- For resize, used `transforms.Resize(32)` and `transforms.CenterCrop(32)`
- For augmentation, used `transforms.RandomRotation(10)` and `transforms.RandomHorizontalFlip()`

The submission displays at least 5 images from the train data loader, and labels each image with its class name (e.g., "Golden Gate Bridge").

```
fig = plt.figure(figsize=(25, 4))

for idx in np.arange(num_img):
    ax = fig.add_subplot(
        num_rows,
        num_img/num_rows,
        idx+1,
        xticks=[],
        yticks=[],
    )
    # [.] unnormalize image data
    img = images[idx] / 2 + 0.5
    # [.] convert from PyTorch Tensor image type (0=C, 1=H, 2=W) to Numpy image type
    plt.imshow(np.transpose(img, (1, 2, 0)))
    # [.] print text label for each image
    ax.set_title(classes[labels[idx]])
```

num of classes: 50



Good job performing Exploratory Visualization to develop a better understanding of the dataset.

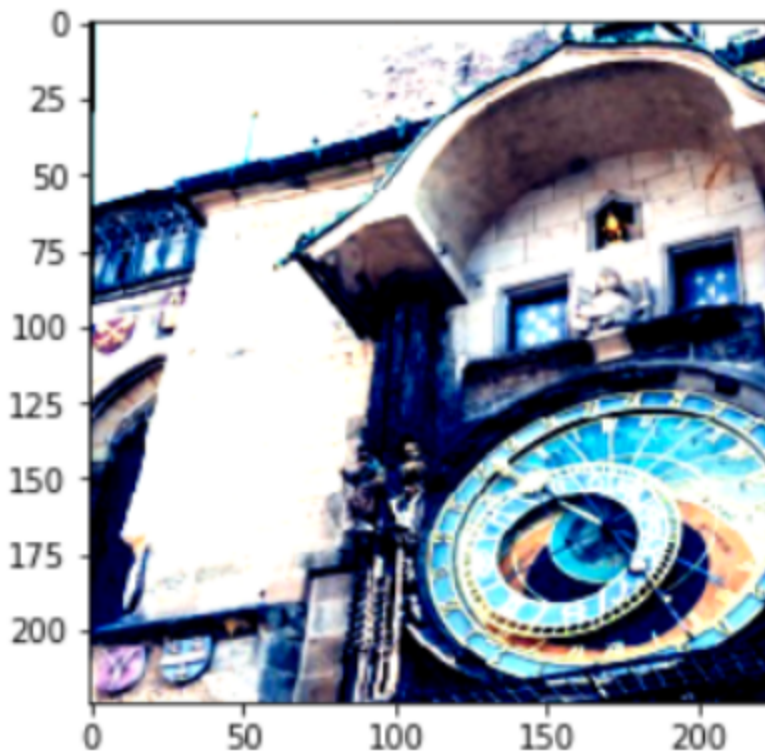
This step is more generally referred to as EDA (shot for Exploratory Data Analysis). Performing EDA on the dataset helps us develop a complete understanding of the data especially when working on projects where we are not aware of the characteristics of training data. In case of image classification, it's usually a good idea to visualize atleast a small subset of the images.

💡 You could've processed the class labels and then used them in the plotting functions

```
classes = [item[3:].replace("_", " ") for item in loaders_scratch['train'].dataset.classes]
```

Then your labels would look as follows:

Prague Astronomical Clock



The submission chooses appropriate loss and optimization functions for this classification task.

Loss function:

- `CrossEntropy`

Optimizer:

- `Adam`

💡 Suggestion: You can also implement a [Learning Rate Scheduler](#) to increase performance and reduce training time.

```
scheduler = optim.lr_scheduler.StepLR(optimizer_scratch, step_size=100, gamma=0.9)
```

You can also check your GPU memory usage during the training sessions as follows

```
print("use_cuda: ", use_cuda, " -> ", torch.cuda.get_device_name(0))
print('Memory Usage:')
print('\tAllocated:', round(torch.cuda.memory_allocated(0)/1024**3, 1), 'GB')
```



```
print('\tCached:  ', round(torch.cuda.memory_reserved(0)/1024**3,1), 'GB')
'
```

💡 Additional Reading

- Check out this excellent blog post to understand the difference between different loss functions:
https://gombru.github.io/2018/05/23/cross_entropy_loss/
- This blog post by Sebastian Ruder on different optimization algorithms is also a good read:
<https://ruder.io/optimizing-gradient-descent/>

The submission specifies a CNN architecture.

```
# define the CNN architecture
class Net(nn.Module):
    ## TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(      # [.] Conv1, in=3x32x32, out=16x16x16
            in_channels = 3,
            out_channels = 16,
            kernel_size = 3,
            padding      = 1,
        )
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)      # [.] conv2, out:
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)      # [.] conv3, out:
        self.maxpool = nn.MaxPool2d(2, 2)                # [.] maxpool for downs
        self.fc1 = nn.Linear(64*4*4, 500)                 # [.] fc1, flatten the
        self.fc2 = nn.Linear(500, 50)                    # [.] fc2, in: 500, out
        self.dropout = nn.Dropout(0.25)                  # [.] dropout by 25%
```

Your network contains **3** convolution layers and **2** dense layers. Good choice!



The advantage of using a Deep Learning library such as PyTorch is that you only need to define the `forward` function and the `backward` function (i.e. Backpropagation step) is defined automatically by the built-in `autograd` module. Things would be quite hard if we had to worry about all those gradients and calculus and implement chain rule manually!

💡 Suggestion: You can make use of `nn.Sequential` module to design your architecture more cleanly. This also makes it much easier to keep track of layer sizes as compared to the mental gymnastics required in the traditional approach.

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.fc1 = nn.Linear(56*56*32, 50)

    def forward(self, x):

        x = self.layer1(x)
        x = self.layer2(x)

        x = x.view(x.size(0), -1)
        x = self.fc1(x)

        return x
```

Answer describes the reasoning behind the selection of layer types.

Question 2: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:

- Use a few conv2d layers to increase the number of feature maps incrementally, but reduce their size gradually.
- Apply fully connected layer to reduce the number of output to that of the number of classes.
- Apply dropout to reduce overfitting.

Note - When working on a personal project or any other project in corporate scenario, it's a good idea to spend some time thinking about the utilities of using different functions and layers and see if they fit your requirements.

💡 Suggestion - You can use `torchsummary` library to generate a summary of your model architecture as shown below

```
from torchsummary import summary

summary(model_scratch, (3, 224, 224))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 224, 224]	448
MaxPool2d-2	[-1, 16, 112, 112]	0
BatchNorm2d-3	[-1, 16, 112, 112]	32
Conv2d-4	[-1, 32, 112, 112]	4,640
MaxPool2d-5	[-1, 32, 56, 56]	0
BatchNorm2d-6	[-1, 32, 56, 56]	64
Conv2d-7	[-1, 64, 56, 56]	18,496
MaxPool2d-8	[-1, 64, 28, 28]	0
BatchNorm2d-9	[-1, 64, 28, 28]	128
Conv2d-10	[-1, 128, 28, 28]	73,856
MaxPool2d-11	[-1, 128, 14, 14]	0
BatchNorm2d-12	[-1, 128, 14, 14]	256
Conv2d-13	[-1, 256, 14, 14]	295,168
MaxPool2d-14	[-1, 256, 7, 7]	0
BatchNorm2d-15	[-1, 256, 7, 7]	512
Linear-16	[-1, 4096]	51,384,320
Dropout-17	[-1, 4096]	0
Linear-18	[-1, 1024]	4,195,328
Dropout-19	[-1, 1024]	0
Linear-20	[-1, 133]	136,325

Total params: 56,109,573

Trainable params: 56,109,573

Non-trainable params: 0

Input size (MB): 0.57

Forward/backward pass size (MB): 17.88

Params size (MB): 214.04

Estimated Total Size (MB): 232.50

The submission implements an algorithm to train a model for a number of epochs and save the "best" result.

The `train()` function only saves checkpoints if the validation loss drops below the previous lowest value



```
if valid_loss <= valid_loss_min:
    print(
        'Validation loss decreased ({:.6f} -> {:.6f}). Saving mode
l ....'
        .format(valid_loss, valid_loss_min)
    )
    torch.save(model.state_dict(), save_path)
    valid_loss_min = valid_loss
```

The submission implements a custom weight initialization function that modifies all the weights of the model. The submission does not cause the training loss or validation loss to explode to `nan`.

```
def custom_weight_init(m):
    ## TODO: implement a weight initialization strategy

    classname = m.__class__.__name__

    if classname.find('Linear') != -1:
        # apply a uniform distribution to weights, set bias = 0
        m.weight.data.uniform_(0.0, 1.0)
        m.bias.data.fill_(0)
```

The trained model attains at least 20% accuracy on the test set.

✗ Since model's accuracy relies on its architecture and training methodology, this requirement will be evaluated once you remove the augmentations in your validation dataset and re-run the training/testing loops.

Please don't worry about it. Once you fix all the issues, the next reviewer will evaluate and pass this requirement.

Step 2: Create a CNN to Classify Landmarks (using Transfer Learning)

The submission specifies a model architecture that uses part of a pre-trained model.

You chose `vgg16` for transfer learning. That's a good choice!

✗ However, please note that you've repeated the augmentation mistake as pointed out earlier. Augmentations should only be applied to training data.

As an experiment, you could also try some other pre-trained models available in `torchvision.models` to see which one performs the best (<https://pytorch.org/vision/stable/models.html>)

The submission details why the chosen architecture is suitable for this classification task.

Question 3: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

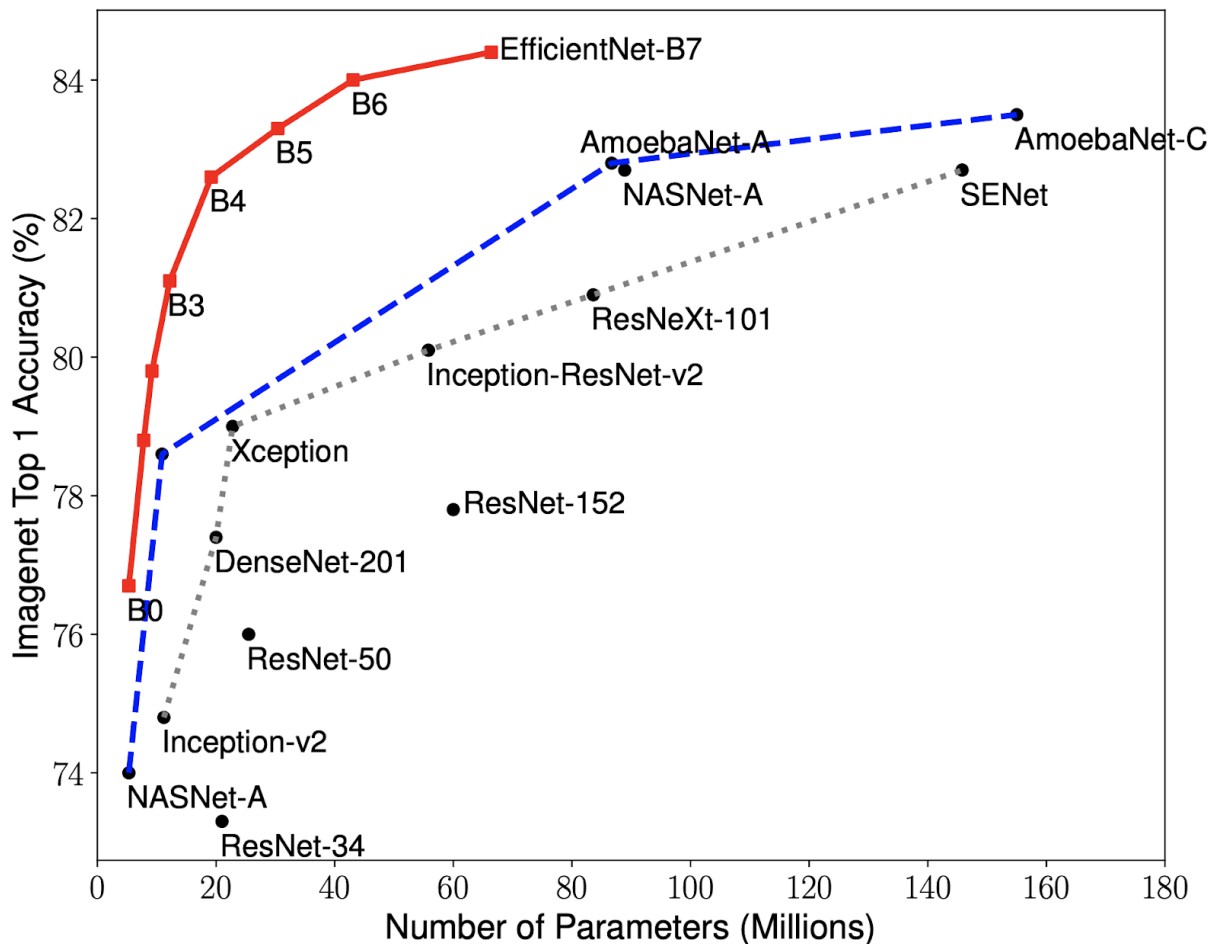
Answer:

- use pretrained vgg16 network
- first, freeze feature parameters
- modify the last layer, keep `num_inputs`, change num of classes to match the current one

You've presented a succinct summary of the transfer learning architecture you implemented. 🙌

Models pre-trained on ImageNet database are very good feature extractors already. So, we mostly need to change the dense layers and retrain the network while freezing the convolution layer parameters and only training the fully-connected layers.

Here's a representation of the no. of parameters involved in models vs their ImageNet performance



The submission uses model checkpointing to train the model and saves the model weights with the best validation loss.

Accuracy on the test set is 60% or greater.

✗ Since model's accuracy relies on its architecture and training methodology, this requirement will be evaluated once you remove the augmentations in your validation dataset and re-run the training/testing loops.

Please don't worry about it. Once you fix all the issues, the next reviewer will evaluate and pass this requirement.

Step 3: Write Your Landmark Prediction Algorithm

The submission implements functionality to use the transfer learned CNN from Step 2 to predict top k landmarks. The returned predictions are the names of the landmarks (e.g., "Golden Gate Bridge").

✗ As mentioned earlier, we should never apply augmentations during inference. You've used `data_transform` which contains augmentations and therefore its use is incorrect here.

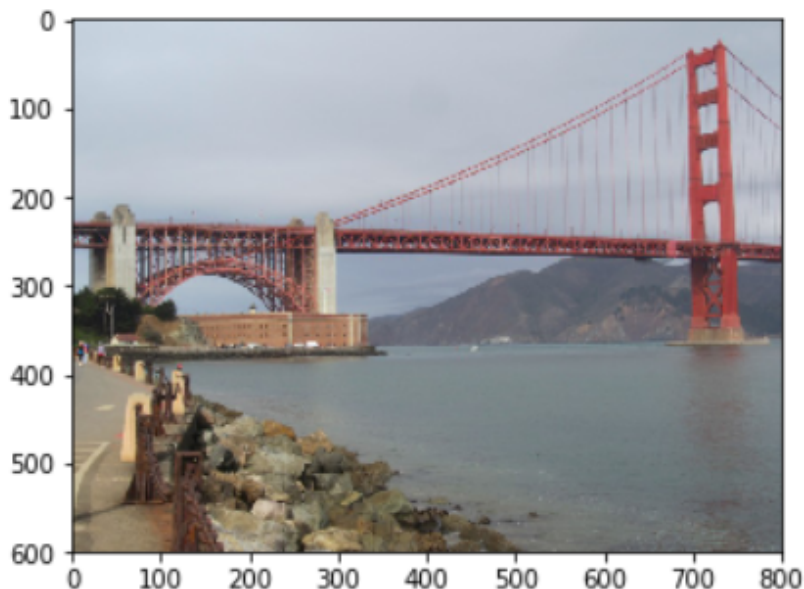
(Note - There are some occasions when augmentations may be applied to test data but that is rare and the methodology is quite different. It is a bit beyond the scope of our discussion here)

⚠ It's a good idea to set the model in evaluation mode before performing inference.

```
model_transfer.eval()
output = model_transfer(img_tensor)
```

The submission displays a given image and uses the functionality in "Write Your Algorithm, Part 1" to predict the top 3 landmarks.

`suggest_locations()` function is implemented such that it calls the `predict_landmarks()` function defined earlier to fetch the `top-3` predicted class labels and then print them alongside the image.



I think this picture is:

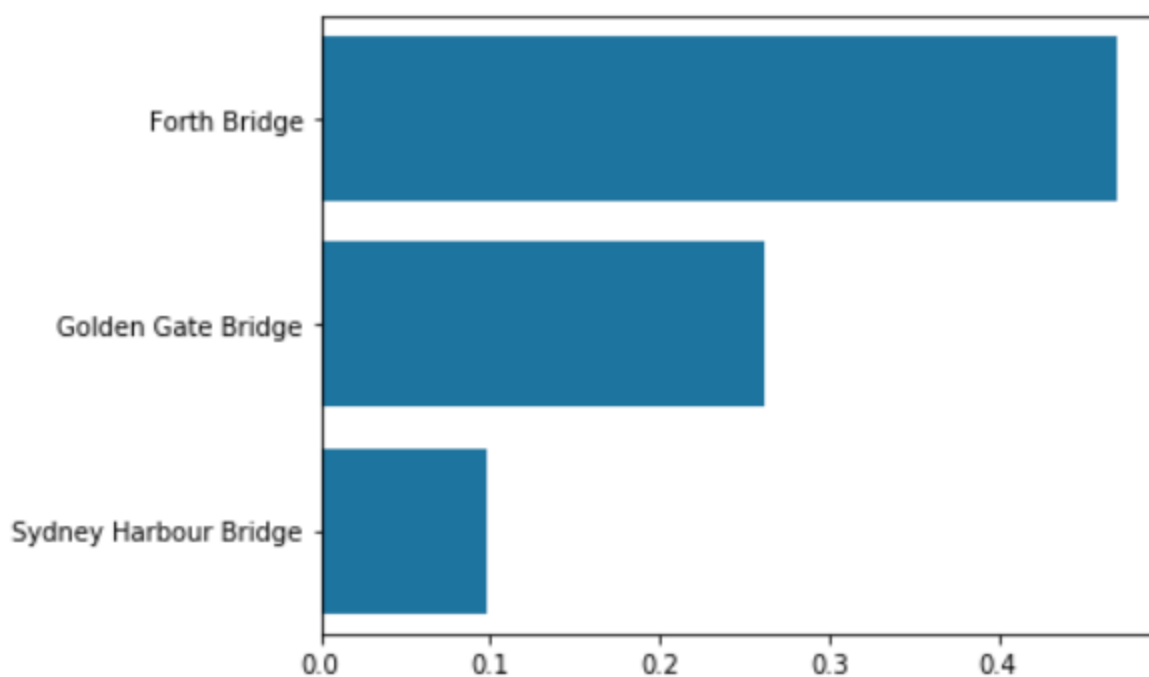
38.Forth_Bridge 09.Golden_Gate_Bridge 28.Sydney_Harbour_Bridge

💡 Suggestion: You could also display a plot of the predicted topk probabilities as follows

```
def suggest_locations(img_path):  
  
    path = img_path.split('/')  
    print(f"Actual Label: {img_path.split('/')[2][3:].replace('_', ' ')}")  
  
    # get landmark predictions  
    confidence, landmarks = predict_landmarks(img_path, 3)  
  
    print(f"Predicted Label: {landmarks[0]}")  
  
    img = Image.open(img_path).convert('RGB')  
  
    plt.figure(figsize = (6,10))  
  
    ax = plt.subplot(2,1,1)  
    ax.imshow(img)  
  
    plt.subplot(2,1,2)  
    sns.barplot(x=confidence, y=landmarks, color=sns.color_palette()[0]);  
    plt.show()
```


Actual Label: Golden Gate Bridge

Predicted Label: Forth Bridge



The submission tests at least 4 images.

⚠ You'll need to retest these images after removing the augmentations from the pipeline

Submission provides at least three possible points of improvement for the classification algorithm.

✗ Oops! Looks like you forgot to answer this question. Please answer it in your next submission

Question 4: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

💡 Suggestion: You can use the [extractor library](#) to generate a visualization of the convolutional feature maps as shown below

```
from extractor import Extractor

extractor = Extractor(list(model_transfer.children()))
extractor.activate()
extractor.info()

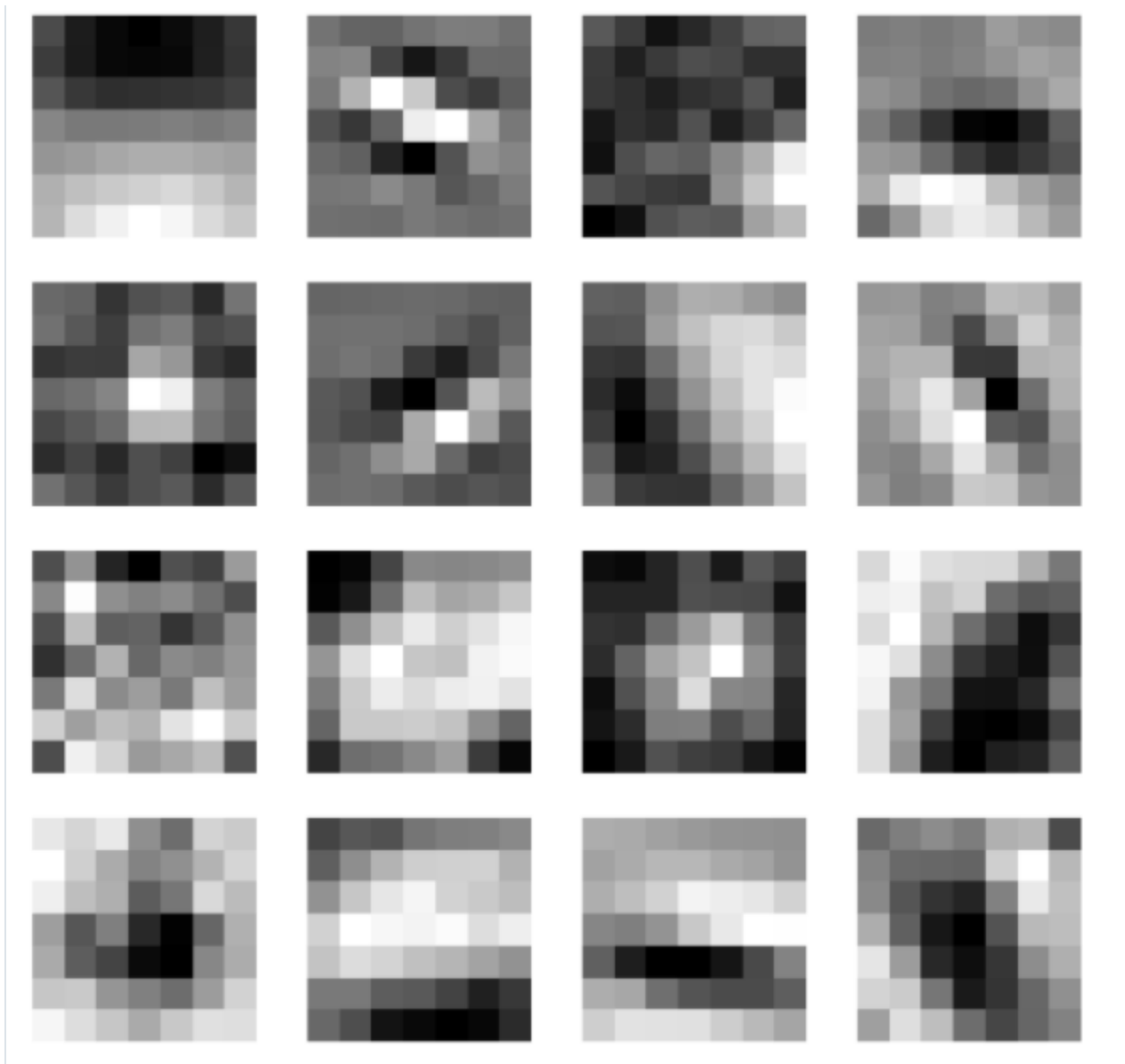
# Visualising the filters
import cv2
import torchvision.transforms as transforms

plt.figure(figsize=(25, 25))
for index, filter in enumerate(extractor.CNN_weights[0]):
    if index == 64:
        break
    plt.subplot(8, 8, index + 1)
    plt.imshow(filter[0, :, :].detach(), cmap='gray')
    plt.axis('off')
plt.show()
img = cv2.cvtColor(cv2.imread('./images/Curly-coated_retriever_03896.jpg'), cv2.COLOR_BGR2GRAY)
```

```
plt.imshow(img, cmap='gray')  
plt.show()
```



For the image shown above, this is what the feature maps would look like (I am only showing some of the maps due to lack of space):



 RESUBMIT

 DOWNLOAD PROJECT





Best practices for your project resubmission

Ben shares 5 helpful tips to get you through revising and resubmitting your project.

[Watch Video](#) (3:01)

RETURN TO PATH

Rate this review

START