Ch 3. Transfer Learning with ResNet50 Part I — from Dataloaders to Training

Seed of Thought : Just how much about the ML model do we know after looking at the confusion matrix?

Lucrece (Jahyun) Shin Follow 8 min read · Sep 18, 2021

57 2

Background: *I'm sharing my computer vision research project conducted during ML masters at University of Toronto. I was given Xray baggage scan images by an airport to develop a model that performs automatic detection of dangerous objects (gun and knife). Given only a small amount of Xray images, I performed Domain Adaptation by using a large number of normal (non-Xray) images of dangerous objects from Google to train a model and later adapting the model to perform well on Xray images.*

*This **Colab notebook** contains the code for all the transfer learning steps mentioned in this post.

In my previous post, I talked about iterative data collection process for web images of gun and knife to be used for domain adaptation. In this post, I will

discuss transfer learning with ResNet50 using the scraped web images. For now, we won't worry about the Xray images and only focus on training the model with the web images. **To read this post, it's recommended to have some knowledge about how to apply transfer learning using a model pre-trained on ImageNet in PyTorch.** I won't explain every step in detail, but will share some useful tips that can answer questions like :

- Why normalize images when using models pre-trained on ImageNet?
- What can we try to fix CUDA out-of-memory-error?
- Which built-in methods of PyTorch's pre-trained models can we use to print out the names of the model layers and parameters?

*** The Gems for critical thinking in this post are in the keyword *"Seed of Thought"*.** So if you are familiar with the basic steps for transfer learning in image recognition, I recommend skipping to sections with the keyword.

0. Seed of Thought 🌱

I've come across many "deep learning for beginners" online courses or articles that discuss transfer learning for image recognition. They cover topics from downloading a multi-class image dataset to fine-tuning a model pre-trained on ImageNet. It's a pretty simple yet powerful method that one can start to play around with deep learning libraries and appreciate the human-like visual recognition ability of neural networks.

0.0. Do Neural Networks "Think" Like We Do?

Is it simple though? **Do model results that we observe accurately reflect what the model is actually *"thinking"*?** Could we be prone to making **human-centric assumptions about the model**, overlooking the fact that this mathematical algorithm might perceive images far differently from how we

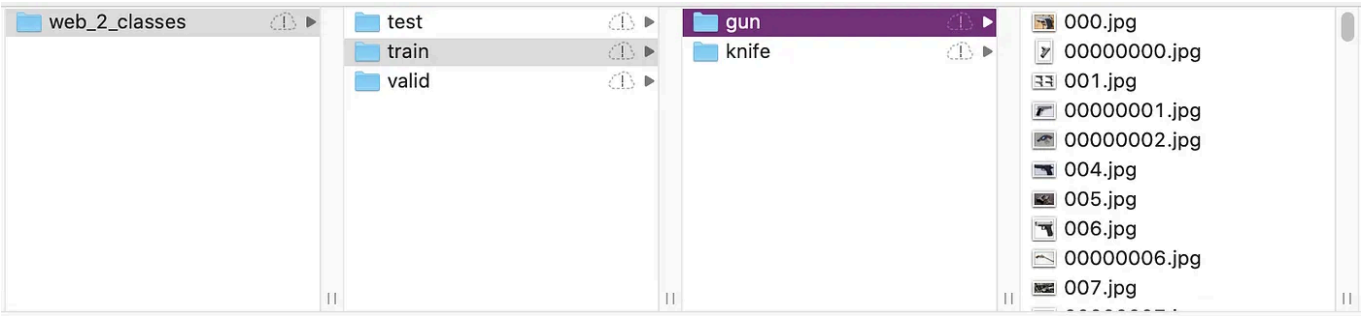
perceive them? These questions are what I've been trying to answer during my entire master's degree. Over the course of my year-long research since September 2020, I can say that I have gained a pool of intuitions to understand the hidden behind-the-scene workings of an image recognition model.

For now, I'll share the basic skeleton procedure for transfer learning in image recognition using ResNet50 and PyTorch. Again, I won't go into far details like the introductory articles but share **some tips I wished I knew when I was first learning this.**

1. Transfer Learning Basics (Tips) — DATA

1.1. Organize Folders

First, I organized the input images into folders using the following format:



Each train/valid/test folder must contain the same number of class folders containing the corresponding images. I used 80%, 10%, 10% split ratio for

train/valid/test partitions.

1.2. Define Dataloaders

Next, I defined PyTorch dataloaders for my dataset using `torch.utils.data.Dataset` and `torch.utils.data.Dataloader`:

```
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms

root_dir = "web_2_classes/"  # 2 classes: gun, knife
transform = transforms.Compose([transforms.Resize((224, 224)),
                                transforms.RandomHorizontalFlip(),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)
train_data = datasets.ImageFolder(root_dir + 'train', transform)
valid_data = datasets.ImageFolder(root_dir + 'valid', transform)
test_data = datasets.ImageFolder(root_dir + 'test', transform)
batch_size = 16
dataloaders = {}
dataloaders['train'] = DataLoader(train_data, batch_size=batch_size, shuffle=True)
dataloaders['valid'] = DataLoader(valid_data, batch_size=batch_size, shuffle=True)
dataloaders['test'] = DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

1.3. Transform Images

There are two necessary transforms that must be applied to the images: **resizing** and **tensorization**. All input images must be resized into the same dimensions using `Resize`, usually a square shape as required by ResNet50 (224 by 224). All images must also be made into tensors using `ToTensor` in order to be processed in PyTorch. There are many other transform functions in `torchvision.transforms` library to augment input images. Popular ones include `RandomRotation`, `RandomResizedCrop`, and `RandomHorizontalFlip`.

1.4 Set Batch Size

You can set your own batch size, preferably as a power of 2. If you start training with GPU and get `RuntimeError:CUDA error: out of memory` error, you can try decreasing the batch size.

```
for i, batch in enumerate(dataloaders['train']):
  print(type(batch), "length:", len(batch))
  print(batch[0].shape)
  print( batch[1])
  if i==0:
    break
```

```
<class 'list'> length: 2
torch.Size([16, 3, 224, 224])
tensor([0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1])
```

This code shows that each batch of a dataloader is a list containing :

- image batch: a 4D tensor of dimension [16 (batch size), 3 (RGB channels), 224 (image width), 224 (image height)]
- target batch : a 1D tensor of length 16 (corresponding labels for the 16 images)

1.5 Normalize Images (Why?)

When using models pre-trained on ImageNet, many articles recommend normalizing the input images using `Noramlize(mean=(0.485,0.456,0.406), std=(0.229,0.224,0.225))`, which are mean and standard deviation of the ImageNet images. What does this normalization do to the images? Since our brains are good at perceiving things visually than mathematically, let's visualize the result of normalization:

Before and after normalizing images

Can you spot the difference? Normalizing seems to increase the images' contrast and light exposure. So why do we need to normalize images? This answer on stack exchange says it serves to *"centre the data for each feature to have a similar range so that the gradients during training don't go out of control"*. Although this sounded right, it felt a little too abstract for me. So I did a practical experiment to check how normalizing input images affect model performance. This table summarizes the results:

	Trained with Normalized images	Trained with Original images
Accuracy for Normalized Test images	99.1%	53.5%
Accuracy for Original Test images	97.2%	99.1%

Using original training images results in a very low accuracy for normalized images

While normalizing training images (left column) gave high accuracies for both normalized and original (not-normalized) test images, not normalizing

them (right column) gave a very low accuracy (nearly a random guess of 50% since there were only 2 classes) for normalized test images. This shows that :

- Normalizing images make the model **more robust**.
- Neural networks can become **very sensitive to the incoming data distribution**. To human eyes, although the normalized images look a bit unclear due to high light exposure, our classification ability won't drop down to a random guess level like this model did.

2. Transfer Learning Basics (Tips) — MODEL

2.1. Download a Pre-trained Model

Next, I downloaded ResNet50 with pre-trained weights on ImageNet: `model = torchvision.models.resnet50(pretrained=True)`. The `torchvision.models` library contains other pre-trained model such as VGG or AlexNet, but I chose Resnet50 for its relatively few number of parameters compared to other architectures.

```
1 for w_name, w in model.named_parameters():
2   print(w_name, w.requires_grad)
```

```
conv1.weight True
bn1.weight True
bn1.bias True
layer1.0.conv1.weight True
layer1.0.bn1.weight True
layer1.0.bn1.bias True
layer1.0.conv2.weight True
layer1.0.bn2.weight True
layer1.0.bn2.bias True
layer1.0.conv3.weight True
```

This code shows that using `named_parameters()` method of the model, you can print out each model parameter's `requires_grad` property (a Boolean value) to see if it is "trainable", i.e. training the model will update its value at each

iteration. By default, this property is set to True for all parameters, which I kept as is since I wanted to **fine-tune all parameters** of ResNet50. Fine-tuning a subset of parameters resulted in lower accuracies.

2.2. Adjust the Final Fully Connected Layer

Using `named_children()` method of the model, you can print the names of the model layers. The last layer of ResNet50 is called "fc", as shown below, which stands for "fully connected". If you print out `model.fc`, you can see that it is a linear layer with 2048 input features and 1000 output features :

```
1 for child in model.named_children():
2   print(child[0])
```

```
conv1
bn1
relu
maxpool
layer1
layer2
layer3
layer4
avgpool
fc
```

```
1 model.fc
```

```
Linear(in_features=2048, out_features=1000, bias=True)
```

2048 comes from the number of convolutional filters for ResNet50's final convolutional layer just before the fc layer and 1000 is the number of classes of ImagNet dataset. To train ResNet50 on my own data which has two classes (gun and knife), I changed this number 1000 to 2 by setting : `model.fc = nn.Linear(in_features=2048, out_features=2, bias=True)`

3. Transfer Learning Basics (Tips) — LOSS & OPTIMIZER

3.1. Define Loss Criterion

I used **cross entropy loss** since I have discrete categories of labels : `criterion = nn.CrossEntropyLoss()` This answer to a post in PyTorch forums explains that we can use either binary cross entropy (BCE) loss or cross entropy loss for a binary classification problem.

3.2. Define Optimizer for Training

Next I defined Adam optimizer to use for training the model parameters : `optimizer = nn.optim.Adam(model.parameters(), lr=0.00002)`. Usually it's recommended to start with a smaller learning rate (lr) for a bigger, deeper architecture. If the learning rate is too big, the model can overfit immediately.

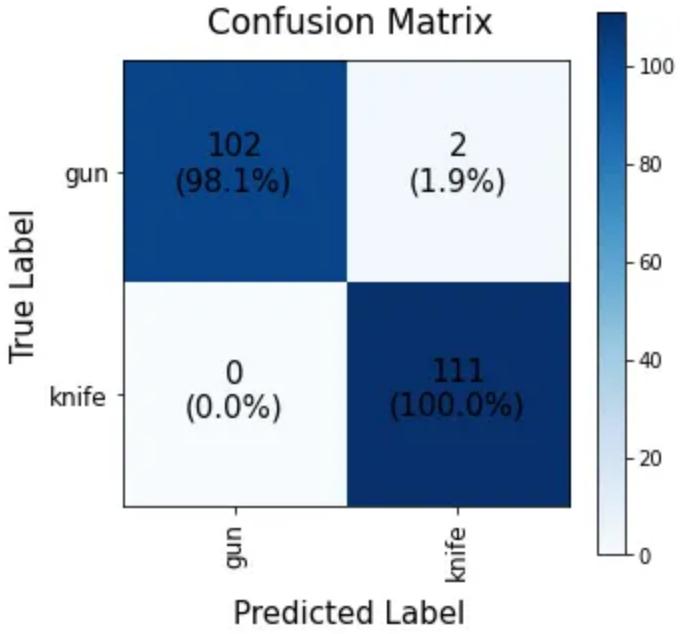
4. Transfer Learning Basics (Tips) — TRAINING & TESTING

4.1. Train the model

Finally, I trained the model for 10 epochs using a training function: `train(model, dataloaders, criterion, optimizer, n_epochs=10)`. I will not explain the training code but this PyTorch tutorial explains it well and also here is my colab notebook containing the function. Since this was a pretty simple fine-tuning task with a relatively small amount of images (~1000 images per class for two classes), the training took only about 3 to 4 minutes for 10 epochs to get to 99% validation accuracy.

4.2. Preliminary Result — Confusion Matrix (All good?)

Here's the preliminary result of ResNet50 fine-tuned on the gun vs. knife binary classification using only web images :



Looks pretty good, right? With 98% and 100% recall for gun and knife, the model seems to be classifying between them quite well. Looking at this confusion matrix, I thought: *"Oh wow! 🎉 The model must have learned the* ***shapes of gun and knife*** *pretty well."* (Would you think that too?)

5. Going Back to the Seed of Thought

Hmm… 🤔 I will ask again the questions I asked in the beginning of this post :

- **Do model results that *we* observe accurately reflect what the model is actually "thinking"?** *(Does the simple confusion matrix showing 100% recall for knife reflect that the model accurately picked up the shape of a knife? Does the model acknowledge a knife' sharp tip, thin blade, and blunt handle while classifying each knife image into the knife class?)*
- **Could we be prone to making human-centric assumptions about the model, overlooking the fact that this mathematical algorithm might perceive images far differently from how we perceive them?** *(If the model learned the shape of a knife, does it know that a knife is "sharp"? Does a model*

know what it means to be "sharp"? Maybe it looks at the angle between the two lines composing the sharp tip? A witch hat also has a sharp tip but not quite in the same sense as a knife. If a model did learn that a sharp tip is a prominent feature of knife, would it classify an image of a witch hat as a knife?)

In the next post, I will try to address these questions by introducing an unexpected problem I faced while analyzing the fine-tuned ResNet50 model's performance. This problem ended up becoming a year-long riddle I tried to find the answer to!!

Again, this colab notebook contains the code for all the transfer learning steps mentioned in this post. Thanks for reading 😊!

- L ℂ₊˚.

Transfer Learning | Computer Vision | Deep Learning | Pytorch | Machine Learning

Written by Lucrece (Jahyun) Shin

173 followers · 17 following

DL Enthusiast. https://www.linkedin.com/in/lucrece-shin/

Follow

Responses (2)