

Create Your Own Image Classifier

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Great work!
Congratulations on completing your project!
I certainly enjoyed walking through your code. It's very clean and very well commented. I can clearly see the effort that has been put into this.

You have perfectly learnt the course content and incorporated them flawlessly in this attempt.
You have met all the specifications, but don't stop here, keep experimenting. Experimenting is the only way you understand DL. Recommend you to experiment with other architectures.

Through this project, we were able to see the basics of using PyTorch as well as the concept of transfer learning, an effective method for object recognition. Instead of training a model from scratch, we can use existing architectures that have been trained on a large dataset and then tune them for our task. This reduces the time to train and often results in better overall performance. The outcome of this project is some knowledge of transfer learning and PyTorch that we can build on to build more complex applications.

As far as what you can do next - You can try building a flask application out of the desktop app that you created now.

We truly live in an incredible age for deep learning, where anyone can build deep learning models with easily available resources! Now get out there and take advantage of these resources by building your own project.
All the very best for your future and happy learning!!

```
from flask import Flask, request, jsonify
import flower_classifier

model = flower_classifier.load_checkpoint('checkpoints/model_egg18_0048.pth', False)
cat_to_name = flower_classifier.load_cat_to_name('cat_to_name.json')

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def process():
    if request.method == 'POST':
        context = request.get_json()
        print(context)
        img = context['data'].split(',')[1]
        img = flower_classifier.open_image_base64(img)
        img = flower_classifier.process_image(img)

        print('Predicting...')
        probs, labels = flower_classifier.predict(img, model, 5, False)
        label_names = [cat_to_name[id] for id in labels]
        print('Got result')

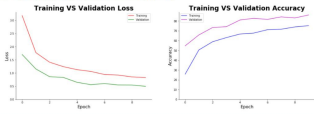
        result = []
        for i in range(len(label_names)):
            result.append({'name': label_names[i], 'prob': probs[i] })
        print(result)
        return jsonify(result)
    else:
        return 'Hi'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Files Submitted

✓	The submission includes all required files. (Model checkpoints not required.)
	The project notebook and helper files are included.

Part 1 - Development Notebook

✓	All the necessary packages and modules are imported in the first cell of the notebook
	All good here! Moving all the imports to the top is a good practice as it helps in looking at the dependencies and the import requirements of the project in one go. Please make sure to import the json module in the first cell of the notebook. I understand that this might seem like a trivial thing, but it really helps in understanding the dependency of a program, and is also considered a good practice in programming in general.
✓	torchvision transforms are used to augment the training data with random scaling, rotations, mirroring, and/or cropping
	Good work augmenting the data using torchvision transforms. Having a diverse and large dataset is crucial for the performance of the deep learning model and sometimes it is hard to accumulate such a diverse variety. The idea of data augmentation is to artificially increase the number of training images our model sees by applying random transformations to the images. For example, we can randomly rotate or crop the images or flip them horizontally. We want our model to distinguish the objects regardless of orientation. Data augmentation can also make a model invariant to transformations of the input data. A flower is still a flower no matter which way it's facing! Extra: Deep learning frameworks usually have built-in data augmentation utilities which you have used in this project, but those can be inefficient or lacking some required functionality. Recommend you to check out imgaug , it is a powerful package for image augmentation with over 60 image augmenters and augmentation techniques. This is extremely helpful!
✓	The training, validation, and testing data is appropriately cropped and normalized
	Well done appropriately cropping and normalizing the data! This is the most important step of working with image data. During image preprocessing, we simultaneously prepare the images for our network and apply data augmentation to the training set. Each model will have different input requirements, but if we read through what ImageNet requires, we figure out that our images need to be 224x224 and normalized to a range. To process an image in PyTorch, we use transforms which are simple operations applied to arrays. The validation (and testing) transforms are as follows: <ol style="list-style-type: none">1. Resize2. Center crop to 224 x 2243. Convert to a tensor4. Normalize with mean and standard deviation The end result of passing through these transforms are tensors that can go into our network. The training transformations are similar but with the addition of random augmentations.
✓	The data for each set is loaded with torchvision's DataLoader
	Dataloader helps load the data in batches, shuffle it and also allow usage of multiple workers to load large amount of data quickly. Great work on implementing it. Each set is correctly loaded and you have chosen an appropriate batch size.
✓	The data for each set (train, validation, test) is loaded with torchvision's ImageFolder
	By using <code>dataset.ImageFolder</code> to make a dataset, PyTorch will automatically associate images with the correct labels provided our directory is set up as above. The datasets are then passed to a <code>Dataloader</code> , an iterator that yield batches of images and labels. The shape of a batch is <code>(batch_size, color_channels, height, width)</code> . During training, validation, and eventually testing, we'll iterate through the Dataloaders, with one pass through the complete dataset comprising one epoch. Every epoch, the training <code>Dataloader</code> will apply a slightly different random transformation to the images for training data augmentation.
✓	A pretrained network such as VGG16 is loaded from torchvision.models and the parameters are frozen
	Good work loading the pretrained networks and using transfer learning. PyTorch has a number of models that have already been trained on millions of images from 1000 classes in ImageNet . The complete list of models can be seen here . The process to use a pre-trained model is well-established: <ol style="list-style-type: none">1. Load in pre-trained weights from a network trained on a large dataset2. Freeze all the weights in the lower (convolutional) layers: the layers to freeze are adjusted depending on similarity of new task to original dataset3. Replace the upper layers of the network with a custom classifier4. Train only the custom classifier layers for the task thereby optimizing the model for smaller dataset I saw that you have perfectly freedzed the parameters of the pretrained layers so that there is no gradient computation on back propagation call using: <pre>for param in model.parameters(): param.requires_grad = False</pre>
✓	A new feedforward network is defined for use as a classifier using the features as input
	A new feedforward network has been defined to be used as a classifier using input features. Dropouts are essential here to reduce overfitting. The final outputs from the network are log probabilities for each of the 102 classes in our dataset.
✓	The parameters of the feedforward classifier are appropriately trained, while the parameters of the feature network are left static
	This is perfect! You are only training the classifier layers and not the layers from the pretrained models. When the extra layers are added to the model, they are set to trainable by default (<code>requires_grad=True</code>), this is the reason there you don't explicitly mention that in your code. Just to give you a scale, vgg-16 model has a total of 135 million parameters, of which just over 1 million will be trained!
✓	During training, the validation loss and accuracy are displayed
	Well done clearly logging the validation loss and accuracy at each step! The training loss (the error or difference between predictions and true values) is the negative log likelihood (NLL) . The NLL loss in PyTorch expects log probabilities, so we pass in the raw output from the model's final layer. The optimizer is Adam , an efficient variant of gradient descent that generally does not require hand-tuning the learning rate. During training, the optimizer uses the gradients of the loss to try and reduce the error ("optimize") of the model output by adjusting the parameters. Only the parameters we added in the custom classifier will be optimized. Extra: Recommend you to check out early stopping , where we stop training when the validation loss has not decreased for a number of epochs. As we continue training, the training loss will only decrease, but the validation loss will eventually reach a minimum and plateau or start to increase. We ideally want to stop training when the validation loss is at a minimum in the hope that this model will generalize best to the testing data. When using early stopping, every epoch in which the validation loss decreases, we save the parameters so we can later retrieve those with the best validation performance. It will be good if you could bring in a plot like this to showcase your training.
	
✓	The network's accuracy is measured on the test data
	Great job on getting 92% accuracy for resnet and 95% for densenet on the testing set. Convolutional neural networks for object recognition are generally measured in terms of top-k accuracy . This refers to the whether or not the real class was in the k most likely predicted classes. For example, top 5 accuracy is the % the right class was in the 5 highest probability predictions. Recommend you to calculate top5 along with top1. Extra: Check out time augmentation
✓	There is a function that successfully loads a checkpoint and rebuilds the model
	Nicely done writing the <code>load_checkpoint</code> method to successfully load the checkpoint and rebuild the model.
✓	The trained model is saved as a checkpoint along with associated hyperparameters and the class_to_idx dictionary
	Great work here. You have stored all the right parameters required to rebuild the model from the checkpoint. Along with the classifier architecture, please make sure to save the major hyperparameters, such as learning rate and the batch size, in the checkpoint as well. <pre>checkpoint = { 'architecture': arch, 'classifier': model.classifier, 'class_to_idx': model.class_to_idx, 'state_dict': model.state_dict(), 'criterion_state': criterion.state_dict(), 'epochs': epochs, 'learning_rate': learning_rate, 'training_batch': training_batch, 'validation_testing_batch': validation_testing_batch }</pre>
✓	The predict function successfully takes the path to an image and a checkpoint, then returns the top K most probable classes for that image
	Awesome job finding the top K classes along with the associated probabilities!
✓	The process_image function successfully converts a PIL image into an object that can be used as input to a trained model
	You just need to make the image have a smaller size sized at 256 and maintain the aspect ratio, which you have taken care. Good job!
✓	A matplotlib figure is created displaying an image and its associated top 5 most probable classes with actual flower names
	Beautifully done. You have done well with the inference and its display.

Part 2 - Command Line Application

✓	train.py successfully trains a new network on a dataset of images and saves the model to a checkpoint
	Well done configuring the application to use.
✓	The training loss, validation loss, and validation accuracy are printed out as a network trains
	The training loss, validation loss, and validation accuracy are printed out as a network trains.
✓	The training script allows users to choose from at least two different architectures available from torchvision.models
	The training script allows users to choose from at least two different architectures available from torchvision.models.
✓	The training script allows users to set hyperparameters for learning rate, number of hidden units, and training epochs
	The training script allows users to set hyperparameters for learning rate, number of hidden units, and training epochs.
✓	The training script allows users to choose training the model on a GPU
	The training script allows users to choose training the model on a GPU. Good job using an additional safety check for GPU availability through <code>torch.cuda.is_available()</code> method!
✓	The predict.py script successfully reads in an image and a checkpoint then prints the most likely image class and it's associated probability
	The predict.py script successfully reads in an image and a checkpoint then prints the most likely image class and it's associated probability.
✓	The predict.py script allows users to print out the top K classes along with associated probabilities
	The predict.py script allows users to print out the top K classes along with associated probabilities.
✓	The predict.py script allows users to load a JSON file that maps the class values to other category names
	The predict.py script allows users to load a JSON file that maps the class values to other category names.
✓	The predict.py script allows users to use the GPU to calculate the predictions
	The predict.py script allows users to use the GPU to calculate the predictions. Good job using an additional safety check for GPU availability through <code>torch.cuda.is_available()</code> method!