Create Your Own Image Classifier

REVIEW | HISTORY

Meets Specifications

Congratulation!!, I think you've done a perfect job of implementing an image classifier. It's very clear that you have a good understanding of the basics. Keep improving and keep learning. 👍

Here are some addition links that might help you further your understanding:

Pytorch note

Data Augmentation

Transfer learning 1

Transfer learning 2

Files Submitted

✓ **The submission includes all required files. (Model checkpoints not required.)**

The required files are submitted !!

Part 1 - Development Notebook

✓ **All the necessary packages and modules are imported in the first cell of the notebook**

Moving all the imports to the top is just a good practice as it helps in looking at the dependencies and the import requirements of the project in one go.

✓ **torchvision transforms are used to augment the training data with random scaling, rotations, mirroring, and/or cropping**

- To get better generalization in your model you need more data and as much variation possible in the data. Sometimes, dataset is not big enough to capture enough variation, in such cases we need to generate more data from given dataset. That is were Data Augmentation comes into the show.
- Great job applying agumentations on the training set to enlarge the data, have a look at this repository for what is further possible in regards to augmentations: https://github.com/aleju/imgaug

✓ **The training, validation, and testing data is appropriately cropped and normalized**

Well done. You were able to resize and normalize the data according to the requirements of the pretrained models. This is the most important process in machine learning / deep learning.

✓ **The data for each set (train, validation, test) is loaded with torchvision's ImageFolder**

Correctly implemented `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.

✓ **The data for each set is loaded with torchvision's DataLoader**

Correctly implemented `ImageFolder`.
The code looks good here as well. Each set is correctly loaded with torchvision's DataLoader and a reasonable batch size.
If you'd like to take a closer look at how the batch size affects training you can check out this post.

✓ **A pretrained network such as VGG16 is loaded from torchvision.models and the parameters are frozen**

The model is properly loaded, and its parameters are correctly frozen. The following link is a good introduction to Transfer Learning.
https://machinelearningmastery.com/transfer-learning-for-deep-learning/

✓ **A new feedforward network is defined for use as a classifier using the features as input**

Correctly load and freeze parameters of pre-trained model.
This method can be applied to most of the pre-trained networks.

✓ **The parameters of the feedforward classifier are appropriately trained, while the parameters of the feature network are left static**

You rightly trained only the parameters of your feed forward network and left those of the feature network static.

✓ **During training, the validation loss and accuracy are displayed**

Perfect! Training loss, validation loss and accuracy are all displayed. You definitely want to keep an eye on these metrics to catch any bugs or inconsistencies with your model quickly.

✓ **The network's accuracy is measured on the test data**

Reasonable test loss and test accuracy.
To further improve the accuracy, consider using a different model architecture and/or tuning the hyperparameters, such as number of epochs, learning rate, number of hidden units and so on.

✓ **There is a function that successfully loads a checkpoint and rebuilds the model**

✓ **The trained model is saved as a checkpoint along with associated hyperparameters and the class_to_idx dictionary**

Successfully save and load a checkpoint.

✓ **The process_image function successfully converts a PIL image into an object that can be used as input to a trained model**

`process_image` function works fine.
This function aims to walk you through the most important stage in working with CNN.

✓ **The predict function successfully takes the path to an image and a checkpoint, then returns the top K most probably classes for that image**

This gives you an insight of how the trained model reacts to each image, you can check whether the model is confused between some labels.

✓ **A matplotlib figure is created displaying an image and its associated top 5 most probable classes with actual flower names**

This is awesome. 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**

This is amazing ! and well-structured code. Great job with the command line utility. This work smoothly.

However, it's better to create a function for argument parser, for example:

```
def init_argparse():
    parser = argparse.ArgumentParser()
    parser.add_argument("data_dir", action="store")
    parser.add_argument('--arch', action="store", dest="arch", type=str)
    parser.add_argument('--learning_rate', action="store", dest="learning_rate", type=float)
    parser.add_argument('--epochs', action="store", dest="epochs", type=int)
    parser.add_argument('--hidden_units', action='append', dest='hidden_units', type=int)
    parser.add_argument('--gpu', action="store_true", default=False, dest='gpu')
    args = parser.parse_args()
    return args
```

Moreover, we normally create several functions instead of writing the whole long script.
For example, you might create functions for `load_data`, `init_model`, `train`, `save_model`, then call these functions inside the `main` function, so that your code will be more beautiful and easier to understand.

✓ **The training loss, validation loss, and validation accuracy are printed out as a network trains**

Training loss, validation loss etc. are being printed correctly during the training.
By validating the model at each training epoch on the validation data helps us to determine how the training process is progressing. We could determine whether the model is able to make the necessary distinctions and learn at each step or if the model in unable to learn after reaching a certain threshold.

✓ **The training script allows users to choose from at least two different architectures available from torchvision.models**

There are definitely several pre-trained architectures that you should try !
If you're interested in state-of-the-art CNN, visit this blog

✓ **The training script allows users to set hyperparameters for learning rate, number of hidden units, and training epochs**

Correctly implemented again and again.
We also usually add `batch_size` to argument parser.

✓ **The training script allows users to choose training the model on a GPU**

An important step when working with torch.
Although you allow user to choose between GPU and CPU, don't forget to always check GPU availability, `torch.cuda.is_available()` is a very useful command.

✓ **The predict.py script successfully reads in an image and a checkpoint then prints the most likely image class and it's associated probability**

Nicely done as required.

✓ **The predict.py script allows users to print out the top K classes along with associated probabilities**

Correctly implemented.

✓ **The predict.py script allows users to load a JSON file that maps the class values to other category names**

Good job as expected.

✓ **The predict.py script allows users to use the GPU to calculate the predictions**

Clean and concise.

DOWNLOAD PROJECT

RETURN TO PATH