UDACITY

‹ Return to "AI Programming with Python Nanodegree" in the classroom

DISCUSS ON STUDENT HUB

Create Your Own Image Classifier

REVIEW

CODE REVIEW 5

HISTORY

▸ train.py 2

▾ functions.py 2

```
__author__ = "Chris"

import numpy as np
import torch
from torch import nn, optim
import torch.nn.functional as F
from torchvision import datasets, transforms, models


def build_classifier(model, input_units, hidden_units, classes, dropout):
    """
    Function to build a new classifier

    :param model: type of model
    :param input_units: number of input units to the NN
    :param hidden_units: number of hidden units of the NN
    :param classes: number of classes to categorize
    :param dropout: probability of dropout

    :return: classified but untrained model
    """
    # Weights of the pretrained model should be frozen so we don't backprop thi
    # Weights of pretrained model are frozen so we don't backprop through/updat
```

```
    for param in model.parameters():
        param.requires_grad = False

    from collections import OrderedDict
    classifier = nn.Sequential(OrderedDict([
                              ('fc1', nn.Linear(input_units, hidden_units)),
                              ('relu', nn.ReLU()),
                              ('dropout1', nn.Dropout(dropout)),
                              ('fc2', nn.Linear(hidden_units, classes)),
                              ('output', nn.LogSoftmax(dim=1))
                              ]))

    # Replacing the pretrained classifier with the one above
    model.classifier = classifier
    return model


def validation(model, valid_loader, criterion, gpu):
    """
    Function to validate the trained model

    :param model: type of model
    :param valid_loader: transformed validation data
    :param criterion: loss function
    :param gpu: gpu mode (T/F)
    :return: loss value and accuracy
    """
    valid_loss = 0
    accuracy = 0

    if gpu == True:
        images, labels = images.to('cuda'), labels.to('cuda')
    else:
        pass

    for ii, (images, labels) in enumerate(valid_loader):
        if gpu == True:
            images, labels = images.to('cuda'), labels.to('cuda')
        else:
            pass

        outputs = model.forward(images)
        valid_loss += criterion(outputs, labels).item()
        probs = torch.exp(outputs)
        equality = (labels.data == probs.max(dim=1)[1])
        accuracy += equality.type(torch.FloatTensor).mean()

    return valid_loss, accuracy


def train_model(model, epochs, train_loader, valid_loader, criterion, optimizer
    """
    Function to train neural network

    :param model: type of model
    :param epochs: number of epochs
    :param train_loader: transformed training data
    :param valid_loader: transformed validation data
    :param criterion: loss function
    :param optimizer: optimization method
    :param gpu: gpu mode (T/F)
```

```
    :return: trained model and optimizer
    """
    steps = 0
    print_every = 10

    if gpu == True:
```

SUGGESTION

- Here you also need to check if we have the GPU available on the system

```
        model.to('cuda')
    else:
        pass

    for epoch in range(epochs):
        running_loss = 0

        for ii, (inputs, labels) in enumerate(train_loader):
            steps += 1

            if gpu == True:
                inputs, labels = inputs.to('cuda'), labels.to('cuda')
            else:
                pass

            # zero out gradients
            optimizer.zero_grad()

            outputs = model.forward(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Validate
            if steps % print_every == 0:
                # set model to evaluation mode
                model.eval()
                # Turn off gradients (not training)
                with torch.no_grad():
                    valid_loss, accuracy = validation(model, valid_loader, crit
                print(f"Epoch {epoch+1}/{epochs}.. "
                      f"Train loss: {running_loss/print_every:.3f}.. "
                      f"Validation loss: {valid_loss/len(valid_loader):.3f}.. "
                      f"Validation accuracy: {accuracy/len(valid_loader):.3f}"
```

AWESOME

- Great job printing the logs here again

```

                running_loss = 0
                # Turn training mode back on
```

```
                model.train()
    return model, optimizer


def test_model(model, test_loader, gpu):
    """
    Function to test NN

    :param model: type of model
    :param test_loader: transformed test data
    :param gpu: gpu mode (T/F)
    """
    correct = 0
    total = 0

    if gpu == True:
        model.to('cuda')
    else:
        pass

    with torch.no_grad():
        for ii, (images, labels) in enumerate(test_loader):

            if gpu == True:
                images, labels = images.to('cuda'), labels.to('cuda')
            else:
                pass

            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f"Test accuracy of model for {total} images : {round(100* correct / t


def save_model(model, train_data, optimizer, save_dir, epochs):
    """
    Function to save the information/checkpoint of the model

    :param model: trained model
    :param train_data: data trained upon
    :param optimizer: optimization method
    :param save_dir: directory to save to
    :param epochs: number of epochs in training
    :return: checkpoint
    """
    checkpoint = {'state_dict': model.state_dict(),
                  'classifier': model.classifier,
                  'class_to_idx': train_data.class_to_idx,
                  'opt_state': optimizer.state_dict,
                  'num_epochs': epochs}

    return torch.save(checkpoint, save_dir)


def load_checkpoint(model, save_dir, gpu):
    """
    Function to load the saved state of a trained model

    :param model: trained model
```

```
    :param save_dir: directory of saved state
    :param gpu: gpu mode (T/F)
    :return: model with previously trained values
    """
    if gpu == True:
        checkpoint = torch.load(save_dir)
    else:
        pass

    model.classifier = checkpoint['classifier']
    model.load_state_dict(checkpoint['state_dict'])
    model.class_to_idx = checkpoint['class_to_idx']

    return model


def predict(processed_image, loaded_model, topk, gpu):
    """
    Function to predict the class of an image using a trained NN

    :param processed_image: image that has been transformed
    :param loaded_model: trained model
    :param topk: highest probability of classification
    :param gpu: gpu mode (T/F)
    :return: lists of the top probabilities and classes
    """
    loaded_model.eval()

    if gpu == True:
        loaded_model.to('cuda')
    else:
        loaded_model.cpu()

    with torch.no_grad():
        outputs = loaded_model.forward(processed_image)

    probs = torch.exp(outputs)
    probs_top = probs.topk(topk)[0]
    index_top = probs.topk(topk)[1]

    probs_top_list = np.array(probs_top)[0]
    index_top_list = np.array(index_top[0])

    # Load index and class mapping
    class_to_idx = loaded_model.class_to_idx
    # Invert index-class dictionary: y is a class and x is an index
    indx_to_class = {x: y for y, x in class_to_idx.items()}

    # Convert index list to class list
    classes_top_list = []
    for index in index_top_list:
        classes_top_list += [indx_to_class[index]]

    return probs_top_list, classes_top_list
```

▸ predict.py 1

utilities.py

RETURN TO PATH