Note_2_FineTuneModel

April 19, 2022

Finetune the AlexNet Model Fine-tuning is a process that takes a model that has already been trained for one given task and then tunes or tweaks the model to make it perform a second similar task

[ ]:
```
from __future__ import print_function
from __future__ import division
from pyexpat import model
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import time
import os
import copy
```

Training Loop First step of the fine tunning process is to develop a training loop. We need to understand the following terminology before developing our training loop.

Gradient Descent: It is an optimization algorithm for finding a global minimum of a differentiable function. Gradient descent is simply used in machine learning to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible. We can compute the gradient descent using 4 steps and they are as follows. Step 1: Compute the loss. Step 2:Compute the gradients. Step 3: update the parameters. Step 4: Rinse and repeat.

Epoch: Refers to one cycle through the full training dataset. Usually, training a neural network takes more than a few epochs. In other words, if we feed a neural network the training data for more than one epoch in different patterns, i.e by shuffling data. we hope for a better generalization when given a new "unseen" input (test data).

For each epoch, there are four training steps: Compute model's prediction, compute the loss, compute the gradients for every parameter and update ther parameters.

Optimizer: An optimizer takes the parameters we want to update, the learning rate we want to use and performs the updates through its step() method. There are many optimizers available in pytorch and two of which are SGD and Adam.

Loss funtion: It's a method of evaluating how well specific algorithm models the given data. If predictions deviates too much from actual results, loss function would return a very large number. Various loss functions available in pytorch and can be referred here: https://pytorch.org/docs/stable/nn.html#loss-functions. In our code we are using nn.CrossEntropyLoss. This criterion computes the cross entropy loss between input and target.

[ ]:
```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Above code snippet checks for GPU in the system. If the system doesn't have any GPU, then it will perform its training in CPU.

In the below snippet, we are performing both training and validation for all the 50 epochs. When the model is in training phase, we are calculating the loss from its output and optimizing it. Also loss and accracy of is calculated for each epoch. In the validation mode, whenever epoch accuracy is greater than the best accuracy we are updating it with epoch accuracy and performing a deepcopy on model weights. And in the end we are returning the updated model with its accuracy history.

[ ]:
```
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25,
 ↪device="cpu"):
    since = time.time()
    val_acc_history = []
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)
        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode
            running_loss = 0.0
            running_corrects = 0
            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                # zero the parameter gradients
                optimizer.zero_grad()
                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    # Get model outputs and calculate loss
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    _, preds = torch.max(outputs, 1)
                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
```

```
                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].
↪dataset)
            print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss,␣
↪epoch_acc))
            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
            if phase == 'val':
                val_acc_history.append(epoch_acc)
        print('')
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60,␣
↪time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))
    # load best model weights
    model.load_state_dict(best_model_wts)
    return model, val_acc_history
```

Freezing and Replacing layers Freezing a layer prevents its weights from being modified and prevents gradients being computed. This is a trasnfer learning technique, where the base model(trained on some other dataset)is frozen.

This helper function in the below snippet sets the .requires_grad attribute of the parameters in the model to only want to compute gradients for the newly initialized layer

[ ]:
```
def set_parameter_requires_grad(model, only_tune_head):
    if only_tune_head:
        for param in model.parameters():
            param.requires_grad = False
```

Adding trainable layers on top of the frozen layers, can make the model learn to turn the old features into predictions on a new dataset. So, we replaced the 1000 neurons of the pre-trained AlexNet/ResNet model with 10 neurons where each neuron represents a single digit.

[ ]:
```
def initialize_model(model_name, num_classes, only_tune_head,␣
↪use_pretrained=True):
    # Replace the prediction head of the network to accomadate the new number␣
↪of classes
    model_ft = None
    input_size = 0
    if model_name == "resnet":
        """ Resnet18
        """
```

```
        model_ft = models.resnet18(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, only_tune_head)
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        input_size = 224
    elif model_name == "alexnet":
        """ Alexnet
        """
        model_ft = models.alexnet(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, only_tune_head)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224
    else:
        print("Invalid model name, exiting...")
        exit()
    return model_ft, input_size
```

Some common initalization before staring the training has been made in the below snippet.

[ ]:
```
# Models to choose from  are resnet and alexnet
MODEL_NAME = "alexnet"
# Number of classes in the dataset
NUM_CLASSES = 10
# Batch size for training (change depending on how much memory you have)
BATCH_SIZE = 1024
# Number of epochs to train for
NUM_EPOCHS = 50
# Modify finetune. When False, we finetune the whole model,
# when True we only update the reshaped layer params
ONLY_TUNE_HEAD = True
model_ft, input_size = initialize_model(MODEL_NAME, NUM_CLASSES,␣
 ↪ONLY_TUNE_HEAD, use_pretrained=True)
```

MNIST Data and Data loaders The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. Also MNIST data is a greyscal so we stack the image to create 3 channels.

Pytorch Data loader: Combines a dataset and a sampler, and provides an iterable over the given dataset. The DataLoader supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation).

[ ]:
```
transform = transforms.Compose([
 transforms.Resize(input_size),
 transforms.ToTensor(),
 transforms.Lambda(lambda x: x.repeat(3, 1, 1)),
 transforms.Normalize(
```

```
 mean=[0.485, 0.456, 0.406],
 std=[0.229, 0.224, 0.225]),
 ])

mnist_train_dataset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
mnist_test_dataset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)
dataloaders_dict = {
    "train": torch.utils.data.DataLoader(mnist_train_dataset,␣
 ↪batch_size=BATCH_SIZE, shuffle=True),
    "val": torch.utils.data.DataLoader(mnist_test_dataset,␣
 ↪batch_size=BATCH_SIZE, shuffle=True)
}
```

Before starting our training process. We need to send our model to GPU using the below code snippet. So that training can be done faster. It is also possible to train the model with CPU but the training time will be high.

[ ]:
```
# Send the model to GPU
model_ft = model_ft.to(DEVICE)
```

Next step is to collect the parameters which needs to be optimized/updated in this run. Can be gathered using .parameters() function like shown in the below snippet.

[ ]:
```
params_to_update = model_ft.parameters()
```

Setup the optimiser and Loss function: In this training, we are using one of the popular optimisers called Adam optimiser. It is easy to implement, will compute efficiently and requires less memory space. Here, we just need to pass the following parameters(parameters of the layers to update, learning rate, epsilon and weight_decay), Pytorch's built-in function will take care of the computation.

Similary, instead of we calculation the loss manually we can use the pytorch's buit-in function to compute the loss between input and target like shown in the below snippet.

[ ]:
```
# Setup the optimiser
optimiser_ft = optim.Adam(params_to_update, lr=0.001, eps=1e-08, weight_decay=0.
 ↪0)
# Setup the loss fxn
criterion = nn.CrossEntropyLoss()
```

Train and evaluate: We have created our own custom function to train and evaluate the model. model, dataloader dictionary, loss function, optimiser funtion, number of epochs and device typer are the parameters for our custom function. Once after training, we will have the updated model along with its history in the following model_ft and hist variables respectively.

[ ]:
```
model_ft, hist = train_model(model_ft, dataloaders_dict, criterion,␣
 ↪optimiser_ft, num_epochs=NUM_EPOCHS, device=DEVICE)
```

Save and export the model: After sucessfully performing our training with MNIST dataset, we need to save our model. So that it can be shared. To save the model, we are first creating a directly called model and performing the save operation using torch.save().

[ ]:
```
os.makedirs("models", exist_ok=True)
torch.save(model_ft.state_dict(), f"models/mnist_{MODEL_NAME}.pt")
print('Done')
```

On sucessful completion of the above code model with .pt extension can be found in the models directory. Later this can be share across team for further testing or developement.