Sport Detection And Video Processing In PyTorch

Achraf KHABAR

July 28, 2023

1 Introduction

This paper is explaining the process of training model of sport classification using the Torch framework. this model is trained on a huge image dataset with multiple classes by the name of sport which we want to detect.

GitHub's implementation: Source-code.

2 Data Cleaning

Our data is a bunch of images, our first problem is to transform the data from **pixels** to **tensors** in order to simplify the processing.

2.1 Filtering Images

The image is a 3D images (matrix of shape like $(x, y, 3)$) , first we need to extract the data from the dataset using **openCv** package, we need first to convert the image from BGR which is extracted by **openCv** to RGB , after we need to normalize all the shapes of images, from $(x, y, 3)$ to $(244, 244, 3)$.

Algorithm 1 Load and Preprocess Images

```
Initialize Sport_Labels as the set {boxing, swimming, table_tennis}
datapath ← '../data'
pathToImages ← ListPaths(datapath)
Initialize an empty list data
Initialize an empty list labels
for each image_path in pathToImages do
    label ← Split(image_path, os.path.sep)[-2]
    if label not in Sport_Labels then
        Continue to the next iteration
    end if
    image ← ReadImage(image_path)
    image ← ConvertToRGB(image)
    image ← ResizeImage(image, 244, 244)
    Append image to the list data
    Append label to the list labels
end for
End For
```

2.2 Converting The Result To Ndarray and Tensors

After getting the **images matrixes**, we need to convert them into **Ndarray** for example **NumPy** in python, for two main reasons :

- **NumPy** arrays are so soft, easy, and simple to manipulate.
- We will convert **NumPy** arrays into **Torch tensors**.

In the final result, we need to have two **data structures** : a list of images tensors and a list o labels of each bunch of images.

Algorithm 2 Data Processing

```
data ← np.array(data)
labels ← np.array(labels)
lb ← label_binarize(labels, classes=list(Sport_Labels))
class_labels ← list(Sport_Labels)                    ▷ Store the class labels separately
labels ← torch.tensor(lb, dtype=torch.float32)       ▷ Convert the labels to float
```

Now we have a **tensors variables**, now we can use **Torch** framework.

2.3 Splitting The Data For Training And Testing

In this step, we will split the data into **train set** and **test set**, we will use 25% of the **data** as **test set** and 75% of **data** as **train set**.

Algorithm 3 Train-Test Split

$X\_train, X\_test, Y\_train, Y\_test \leftarrow$ train_test_split(data, labels, test_size=0.25, stratify=labels, random_state=42)

3 Normalizing The Data

3.1 Data transformations

Data transformations are essential for improving the performance and robustness of machine learning models. By applying various random augmentations to the training data, the model learns to be more invariant to certain changes and generalizes better to unseen data. This reduces overfitting and helps the model achieve better accuracy on the test data. Using the **transforms** module in PyTorch makes it easy to create and apply these transformations efficiently during the data loading process. and we will use the same thing for the **test set**.

Algorithm 4 Train Transformation Pipeline

```
train_transform ← Compose([
    ToPILImage(),
    RandomRotation(30),
    RandomHorizontalFlip(),
    RandomVerticalFlip(),
    ToTensor(),
    Normalize(mean=[123.68/255.0, 116.779/255.0, 103.939/255.0], std=[1, 1, 1]),
    ])
```

3.2 Custom Dataset class

The Custom Dataset class is designed to create a custom dataset that can be used with PyTorch's Data Loader, which is an essential utility for efficient data loading during training or evaluation of machine learning models. By implementing the **len** and **getitem** methods, the class becomes iterable, allowing PyTorch's data loaders to access individual samples and process them on-the-fly as needed.

The class takes in two main inputs: images and labels. The images parameter represents the input data, typically a collection of images, while the labels parameter contains the corresponding target labels or classes for each image.

Additionally, the class has an optional transform parameter that allows you to pass a data transformation pipeline. This pipeline applies various image transformations or preprocessing steps to the

input images before returning them during data loading. These transformations can include resizing, data augmentation (e.g., rotation, flipping), normalization, and more.

- Implementing **len()** and **getitem()**:

The **len** method is implemented to return the length of the dataset, which corresponds to the total number of samples in the dataset. This enables the use of Python's built-in **len()** function to retrieve the number of samples.

The **getitem** method allows accessing individual samples in the dataset by providing an index idx. It retrieves the image and its corresponding label at the specified index. If a data transformation pipeline (self.transform) is provided, it applies the transformations to the image before returning it along with the label. This ensures that the data is processed on-the-fly during data loading, avoiding the need to preprocess the entire dataset before training.

- Integration with PyTorch DataLoader:

Once the CustomDataset class is defined, you can use it with PyTorch's DataLoader to efficiently load and process the data during training. The DataLoader handles batch creation, shuffling, and parallel data loading, which are crucial for training deep learning models effectively.

Algorithm 5 CustomDataset Class

```
class CustomDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images ← images
        self.labels ← labels
        self.transform ← transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image ← self.images[idx]
        label ← self.labels[idx]
        if self.transform:
            image ← self.transform(image)
        return image, label
```

4 Creation Of The Convolutional Neural Network

4.1 Creation Of Deep Learning Model

The **ResNetModel** is a custom neural network architecture designed for image classification tasks. It is built upon the popular ResNet-50 model, a deep convolutional neural network known for its effectiveness in computer vision tasks. The custom model leverages the pre-trained weights of the ResNet-50 to extract powerful image features and then adapts it for the specific classification task.

4.2 Model Architecture

- **Forward Pass :** During the forward pass, an input image is fed into the **ResNetModel**. The image then passes through the base **ResNet-50** model to extract image features. The output of the base model contains high-level feature representations of the input image.
- **Custom Classification Head:** The extracted image features are then fed into the custom classification head, consisting of three layers:
  - A **fully connected layer** reduces the feature dimensions to 512, which acts as a bottleneck layer.

  - A **ReLU activation function** introduces non-linearity, allowing the model to learn complex relationships between features.
  - A **dropout layer** with a dropout rate of 0.5 randomly sets half of the neuron outputs to zero during training. This technique helps prevent overfitting by introducing robustness in the model's predictions.

- **Final Output:** The output of the custom classification head is a 1D tensor representing the predicted class probabilities for each input image. The tensor has a size of **numClasses**, where **numClasses** corresponds to the number of output classes required for the specific image classification task.

Algorithm 6 ResNetModel: Custom ResNet-Based Neural Network

```
class ResNetModel(nn.Module):
    def __init__(self, num_classes) :        super(ResNetModel, self).__init__()
        self.base_model ← resnet50(pretrained=True)
        in_features ← self.base_model.fc.in_features
        self.base_model.fc ← nn.Identity()
        self.fc ← nn.Sequential(
            nn.Linear(in_features, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
            )

    def forward(self, x):
        features ← self.base_model(x)
        out ← self.fc(features)
        return out
```

4.3 Approach Of Training

4.4 Optimizer

In this step, we will proceed to the function that will help us to train our neural network model. in our case, we will use **stochastic gradient descend**. The **SGD** optimizer is a widely used optimization algorithm for training machine learning models. The update step for each parameter is computed using the following formula:

$$\theta_{new} = \theta_{old} - (\alpha \times \nabla\mathcal{L}(x, C) - (\text{momentum} \times \text{previousUpdate}) - (\text{weightDecay} \times \theta_{old})) \quad (1)$$

with :

- θ_{new} : is the updated value of the parameter.
- θ_{old} : is the current value of the parameter.
- α : is the learning rate hyperparameter, controlling the step size during updates.
- $\nabla\mathcal{L}(x, C)$: is the gradient of the loss function with respect to the parameter.
- momentum : is the momentum hyperparameter, which smooths the update process and helps to accelerate convergence in certain cases.
- previousUpdate : is the accumulated previous update for the parameter.
- weightDecay : is the weight decay hyperparameter, which applies L2 regularization to the parameter to prevent overfitting.

4.5 Loss Function

have somehow generated a model that predicts the probability of y given x. We denote this model by $f(x, \theta)$, where θ represents the parameters of the model. Let us again use the idea behind maximum likelihood, which is to find a θ that maximizes $P(D|\theta)$. Assuming a multinomial distribution, and given that each example $\{(x1, y1), (x2, y2), \ldots (xn, yn)\}$ is independent, we have the following expression:

$$P(\theta) = -\sum_{i=1}^{n} y_i \cdot log(x_i, \theta) \tag{2}$$

4.6 Training Loop

In PyTorch we need to define the Training loop, in order to train our neural network.

Algorithm 7 Training the Model

```
function TRAIN(model, train_loader, criterion, optimizer, num_epochs)
    MODEL.TRAIN
    for epoch in [1, 2, ..., num_epochs] do
        running_loss ← 0.0
        for inputs, labels in train_loader do
            OPTIMIZER.ZERO_GRAD
            outputs ← MODEL(inputs)
            loss ← criterion outputs, torch.argmaxlabels, dim=1
            LOSS.BACKWARD
            OPTIMIZER.STEP
            running_loss ← running_loss + LOSS.ITEM
        end for
        print"Epoch [" + (epoch + 1) + "/" + num_epochs + "] Loss: " + (running_loss /
lentrain_loader)
    end for
end function
```

the train loop function implements the training process for a neural network model using the provided data loader, loss function, and optimizer. The function iterates over the training dataset multiple times (epochs) and updates the model's parameters to minimize the loss, ultimately improving the model's performance on the task at hand.

4.7 Conclusion

This approach is found to be highly effective in achieving accurate and robust image classification results. By leveraging the power of the ResNet-50 model and combining it with a custom classification head, we have successfully developed a deep learning architecture tailored for our specific task. The use of transfer learning and pretrained weights from the **ResNet-50** model significantly reduced the training time and data requirements, making it feasible to achieve exceptional performance even with limited labeled samples.

The training process, as implemented in the **train** function, demonstrates the capability of the model to learn from the training data and adapt to the target classes efficiently. The nested loops for epochs and batches ensure that the model iteratively updates its parameters to minimize the loss function and improve its predictive capabilities.

Moreover, the integration of PyTorch's DataLoader and CustomDataset classes allows seamless data loading, processing, and transformation, further streamlining the training pipeline. The ability to implement custom data transformations using **transforms.Compose** provides flexibility and convenience in data preprocessing and augmentation, contributing to the overall success of the project.

In conclusion, the presented approach not only achieves remarkable accuracy in image classification but also offers flexibility, efficiency, and scalability, making it a valuable solution for a wide range of computer vision tasks. As deep learning continues to advance, this architecture can serve as a foundation for building more complex and sophisticated models for various real-world applications.