CSE 527 HW5 Report

Name: Shreyash Paratkar

SBU-ID: 112673930

Feature Extraction:

The UCF101 training data consists of videos of 101 different actions.

For the first 25 classes of the dataset, the number of videos available is 3360. Out of these 3360 videos, 2409 comprise of the training dataset while 951 comprise the testing dataset.

The respective datasets are constructed after writing custom datasets using the Dataset class of pytorch which is an abstract class representing a dataset. The '__init__' and '__getitem__' methods of the Dataset parent class are overridden according to our problem statement.

Initially, the file 'videos_labels_subsets.txt' in Annos folder is parsed line by line to taken into consideration 5 classes of video frame folders each time. A pandas dataframe is constructed to store the video id and class of each video in separate training and testing dataframes.

These dataframes are then passed into a pickle file to be read by the __init__ function in the custom dataset UCF101Dataset class. Different instances of the UCF101Dataset class are used for training and testing respectively.

In the __getitem__ function of UCF101Dataset, each row of the dataframe is read and parsed to find out the video folder name and the corresponding action of the video.

The action extracted, will become our label and the 25 images, the data.

From the 25 frames for every video, each image is read using open CV and transformations are carried out on each of the images. The transformation for each image is a sequence of normalization followed by conversion of the image to a torch tensor.

But the torch tensor of the image obtained is of dimensions 256 X 340. We need to pass 224 X 224-dimension images to the VGG16 net in order to get the 4096 image feature vectors. But since we must do this without losing out on key data, our work around for the task is to crop the 256 X 240 image into 5 images of 224 X 224 each. The 5 different cropped sections are the top-left, top-right, center, bottom-left and bottom-right respectively. This gives us 125 224X224 images for a single video.

VGG16 Net

The first 2 layers of VGG16Net classifier are kept for feature extraction. Keeping a batch size of 1 (1 video per batch), in the Dataloader, the Dataloader was enumerated to parse the UCF101Dataset created above. Each enumeration of the dataloader gives us a single batch i.e. 125 images for 1 video. These 125 images are passed to the VGG net as a stack of tensors to find 125, 4096 feature vectors for every video in the dataset.

We group the 125, 4096 feature vectors into groups of 5 each, giving us 25 groups of 5. In each group of 4096 sized vectors, we then compute mean of the vectors obtained for 5 cropped variants of the video frame. This mean is used as the feature vector for the i'th image in the video folder.

The below table shows the number of images processed along with the time taken in seconds to extract the features from the images for each of the classes.

Classes	Train size	Train extraction time	Test size	Test extraction time
1-5	11525	987 s	4575	374 s
6-10	13100	1496 s	5150	368 s
11-15	11425	1232 s	4475	364 s
16-20	12275	1190 s	4825	334 s
21-25	11900	902 s	4750	326 s

This computed data is then saved in pickle files.

Here are some stats to check the number of images for which we have computed features.

Number of images in training dataset of 25 classes = 60225

Number of training videos seen for 25 classes = 2409

Number of images in testing dataset of 25 classes = 23775

Number of testing videos seen for 25 classes = 951

Total number of images for 25 classes = 84000

LSTM

LSTM requires us to input data in a 3D format. To conform our data to this format, we load the data from pickle files stored in the previous step and then reshape the data using torch.split and torch.stack. torch.split converts a tensor into a tuple and this tuple is then stacked using torch.stack to give a higher dimension tensor.

After reshaping the data, we get the below tensors

Training data: torch.Size([2409, 25, 4096])

Training labels: torch.Size([2409])

Testing data: torch.Size([951, 25, 4096])

Testing labels: torch.Size([951])

This data is passed to the Long Short-Term Memory Model.

LSTM Architecture

The gating mechanism is what sets LSTMs apart from RNNs and enables it to preserve long term memory [1]. In LSTMs, information from much earlier times is used in the predictions are much later time steps. LSTMs have a complex structure. At each time step, the LSTM take 3 different inputs, namely the current input data, short term memory from the preceding cell and the long-term memory. The cell state is what enables an LSTM to preserve long term memory. Gates (Input, Forget and Output) tell the network whether to preserve or discard information at each time step.

The Pytorch LSTM neural network takes in input dimensions, hidden dimensions and number of layers, where the input dimensions signify the size of the input at each time-step. This size is 4096 for our problem statement, since we have 4096 features for every video frame. Hidden dimensions represent the size of the hidden state and cell state at each time step. As the size of the cell state and hidden state in an LSTM increases, we have more long term and short-term memory preserved in each step. It was found that increasing this value of hidden dimensions, increases the training and testing accuracies. But the hidden dimensions can only be increases up to a certain size for a given input, owing to overfitting, memory utilization and time taken for training the neural network.

Accuracy obtained for hidden_dim:

1.) 256 = 72.3%
2.) 512 = 74.5%
3.) 1024 = 80%

Another important factor in LSTM parameters in the number of layers in the architecture. As we increase the number of layers, we increase the time-steps in the neural network. This causes an increase of 100% in the time taken for the training of the neural network using back propagation.

2 layers were chosen for our network.

The architecture used for the LSTM is inspired by Jessica Yung's LSTM network in [2].

Since we process data of 25 classes on the neural network, the number of output dimensions = 25. We compare this output, i.e. the output obtained from the linear layer with the labels of the training dataset after the reshaping and subtracting 1 from the class numbers, since the loss function needs the labels to start at 0.

The learning rate of 0.0001 does give us an increase in the epoch wise training accuracy over 5 epochs. But the final accuracy in training still doesn't reach 100%. Whereas, a learning rate of 0.00001 gives us the below epoch wise training accuracies.

Training accuracies per epoch:

Epoch: 1 = 74.72 %

Epoch: 2 = 99.045 %

Epoch: 3 = 99.751 %

Epoch: 4 = 99.917 %

Epoch: 5 = 100.0 %

Below are the training and testing stats observed with the final set of model parameters.

Training	Accuracy: 100%	Time taken: 228 seconds
Testing	Accuracy: 80.967%	Time taken: 4 seconds

Linear SVC

Below accuracies are obtained with LinearSVC model with a regularization factor of 0.0001 after passing it a stacked 4096x25 feature matrix as a long vector.

Training	Accuracy: 100%	Total time taken for training and testing: 388 seconds
Testing	Accuracy: 86.014 %	Total time taken for training and testing: 195 seconds

Conclusion

We obtain a better accuracy with Linear SVC (86.014%) as compared to LSTM (80.967%)

References

1] https://blog.floydhub.com/long-short-term-memory-from-zero-to-hero-with-pytorch/

2] https://www.jessicayung.com/lstms-for-time-series-in-pytorch/