Introduction

This project focuses on developing a video classification system using a combination of Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks. The objective is to effectively classify videos into predefined categories from the UCF-101 dataset, leveraging the spatial feature extraction capabilities of CNNs and the temporal processing strengths of LSTMs. The following report is provided with python implementation. Please refer to /code/README.md for more.

System Design

Model Architecture

My model integrates a CNN with an LSTM network. For the CNN component, both a custom-defined architecture and a pre-trained model (e.g., ResNet18) are employed to compare their performance. The CNN acts as a feature extractor from individual video frames, which are then passed to the LSTM network to capture temporal dependencies between frames.

CNN Backbone

Pre-trained ResNet18:

When use_pretrained is set to True, I use ResNet18, a widely recognized and powerful CNN architecture known for its effectiveness in image classification tasks. I have replaced the final fully connected layer of ResNet18 with an identity layer (nn.Identity()), utilizing it solely for feature extraction.

Custom CNN:

If use_pretrained is False, I define a simpler custom CNN architecture. This option is designed to be less computationally intensive while still being effective for feature extraction.

The custom CNN consists of sequential convolutional layers with increasing filter sizes (from 64 to 128, from 128 to 256, from 256 to 512), each followed by batch normalization and ReLU activation for stability and non-linearity.

The use of nn.MaxPool2d and nn.AdaptiveAvgPool2d aids in reducing the spatial dimensions of the output feature maps, distilling essential features into a compact form.

In the custom CNN, the configuration of stride, padding, and kernel size is carefully selected for each convolutional layer:

Stride: A stride of 2 is used in most convolutional layers, effectively reducing the spatial dimensions of the feature maps by half with each layer. This stride value helps in quickly decreasing the size of the output feature maps, enabling the network

to focus on higher-level features and reducing computational load.

Padding: Padding of 1 is used in the 3x3 convolutional layers to maintain a balance between reducing feature map sizes and retaining spatial information. This padding ensures that important features near the edges of the frames are not lost during convolution.

Kernel Size: The initial layer uses a larger kernel size of 7x7 to capture broader features in the input frames, followed by 3x3 kernels in subsequent layers for finer and more specific feature extraction. The combination of larger initial kernels and smaller subsequent kernels allows the network to capture a wide range of features, from general patterns to more detailed aspects.

These parameters work in tandem to ensure that the custom CNN efficiently processes input frames, capturing essential spatial information while progressively reducing data dimensionality, making it suitable for subsequent temporal analysis by the LSTM layers.

```
self.backbone = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1),

    nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(inplace=True),

    nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),

    nn.Conv2d(256, 512, kernel_size=3, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),

    nn.AdaptiveAvgPool2d((1, 1))
```

LSTM Network

Regardless of the chosen CNN backbone, the extracted features are then passed to an LSTM network.

The LSTM is designed with 512 or 256 hidden units (to match the output dimension of ResNet18 or custom CNN) and processes the temporal sequence of features extracted from the frames of the video.

Why use LSTM? While the CNN is responsible of feature extraction, the LSTM

captures temporal dependencies and dynamics, which are crucial for understanding actions and movements in videos.

Final Classifier

The last hidden state of the LSTM, which encapsulates both spatial and temporal information, is passed through a fully connected layer (nn.Linear) to produce the final classification output.

The number of output units in this layer corresponds to the number of classes in the classification task (10 in my case).

Code Base

My code-base is modular, with functionality separated into distinct files for clarity and ease of maintenance. This structure not only enhances readability but also facilitates easy updates and modifications.
Here's an overview of each module:

model.py: Contains the definition of the CNNLSTM class. This module defines the architecture of the neural network, incorporating both CNN and LSTM layers.

train.py: Responsible for the training process of the model. It includes functions to execute the training loop, handle the optimization process, and save the model's state.

evaluate.py: Used for evaluating the model's performance on validation or test data. It includes functions for calculating metrics such as accuracy and loss, and for visualizing the model's performance.

main.py: Serves as the entry point of the program. It integrates all other modules and orchestrates the process of data loading, model training/evaluation, and handling of the results.

dataloader.py: Handles data loading and preprocessing. This module includes the definition of a custom DataLoader that feeds data into the model in the correct format, ensuring efficient and effective training and evaluation.

demo.py: contains code for demonstrating the model's capabilities, typically on sample data. It's useful for showcasing how the model performs in real-world scenarios or with specific examples.

utils.py: Includes utility functions that are used across the project. This includes helper functions for data manipulation and visualization.

params.json: A configuration file that contains hyperparameters and other settings in a JSON format. The use of params.json allows for easy adjustment of model parameters without the need to alter the core code. This is especially useful for hyperparameter tuning and for adapting the model to different datasets or requirements.

Training Process

Training Setup

The model is trained using PyTorch on a GPU-enabled setup. Key configurations and hyper parameters are: (see params.json)

Loss Function: Cross-Entropy Loss
Optimizer: Adam
Learning Rate: 0.0001 (initial)
Batch Size: 4
Epochs: 50
Number of Image Frames: 20
Initial Image Size: [128, 128]
Hidden Size of the LSTM layer: 128
Number of LSTM layers: 2

Training method

My training methodology is designed to efficiently teach the model to classify videos accurately:

Epochs and Batch Processing: I train the model in batches over multiple epochs, balancing learning effectiveness with computational efficiency.

Gradient Reset and Optimization: Each training step begins by resetting gradients to ensure independence between batches. The forward pass computes predictions, and the backward pass optimizes the model using these predictions.

Loss Function: I use cross-entropy loss, ideal for classification tasks, to guide the model in aligning its predictions with the actual labels.

Validation and Model Saving: Post each training epoch, I validate the model to gauge its performance on unseen data. I save the model state if it shows improved performance on the validation set, indicating better generalization.

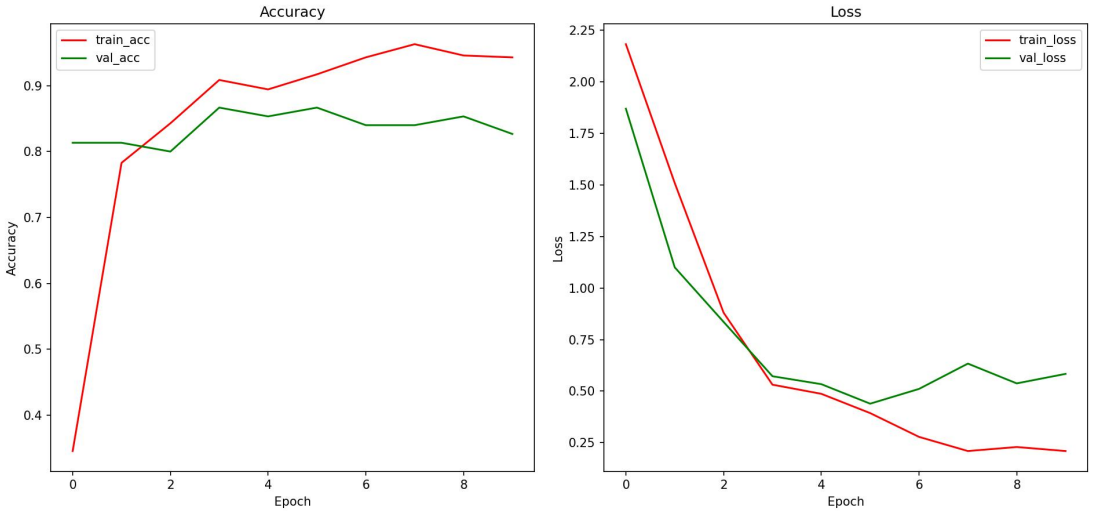
```python
for epoch in range(epochs):
    model.train()
    total_loss, total_correct = 0.0, 0
    # Training loop
    for inputs, labels in tqdm(train_data, desc=f'Epoch {epoch+1}/{epochs}', leave=False):
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = loss_fn(outputs, labels.long())

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_correct += (predicted == labels).sum().item()
```

train.py

Parameter adjustments and results

PreTrained resnet18 model

```
self.backbone = models.resnet18(pretrained=True)
self.backbone.fc = nn.Identity()
lstm_input_size = 512  # ResNet18's feature size b
```




```
Weights loaded successfully from path: cache\best_weights.pt
Evaluate: 100%|                                                  | 19/19 [00:06<00:00,  2.97it/s]
Loss: 0.367, Acc: 0.893
```

This model gives pretty good results,(can improve more by fine-tuning on our data set) so we will concentrate on improving our custom CNN model.

Custom CNN

Several experiments were conducted to optimize model performance

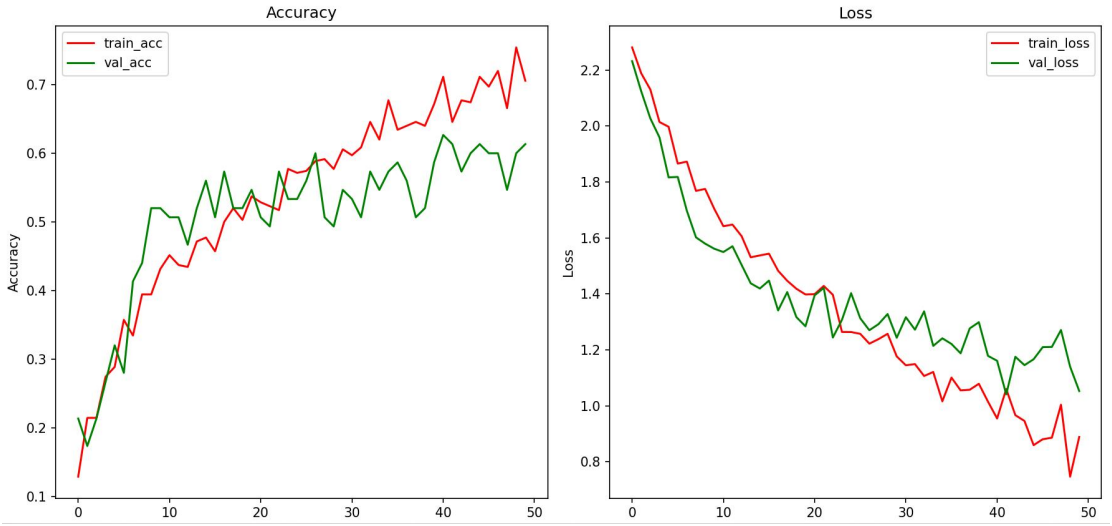
Model Structure Changes

1. **Add More Convolutional Layers**

   Adding more convolutional layers helped the network learn more complex features.

1. First try (2 Convolution layers)

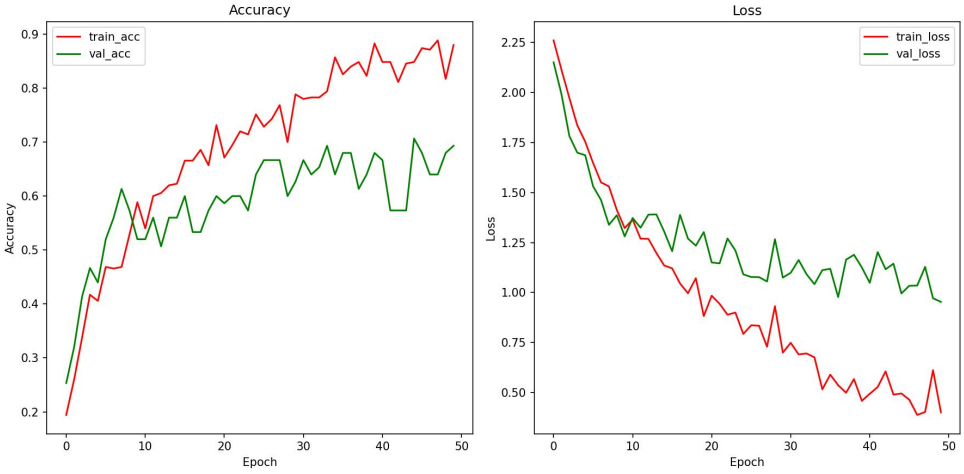
```
else:
    # Define a custom CNN
    self.backbone = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
        nn.BatchNorm2d(64),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=
        nn.BatchNorm2d(128),
        nn.ReLU(inplace=True),
        nn.AdaptiveAvgPool2d((1, 1))
    )
    lstm_input_size = 128  # Custom CNN's output size
```




```
Epoch 50: Train Loss: 0.8885, Train Acc: 0.7057, Val Loss: 1.0527, Val Acc: 0.6133
2024-01-11 17:29:35.614 | INFO     | __main__:main:122 - Saved last model state to: cache\last_weights.pt
Plot saved to results.png
2024-01-11 17:30:33.396 | INFO     | __main__:main:129 - Evaluating on test dataset
Weights loaded successfully from path: cache\last_weights.pt
Evaluate: 100%|████████████████████████████████████████| 19/19 [00:06<00:00,  2.73it/s]
Loss: 0.971, Acc: 0.667
Weights loaded successfully from path: cache\best_weights.pt
Evaluate: 100%|████████████████████████████████████████| 19/19 [00:06<00:00,  2.76it/s]
Loss: 0.949, Acc: 0.693
```

2. Going deep

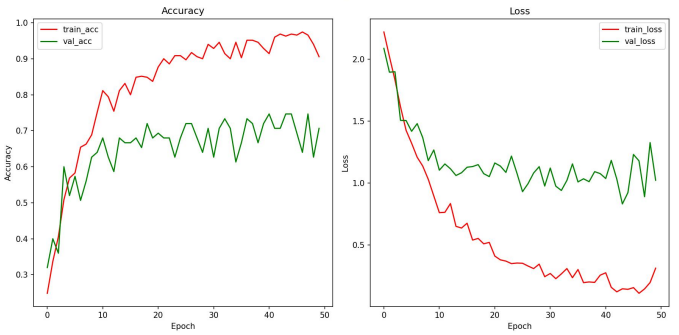
```
nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1, bias=False),
nn.BatchNorm2d(128),
nn.ReLU(inplace=True),
```




```
2024-01-11 18:58:41.553 | INFO     | __main__:main:129 - Evaluating on test dataset
Weights loaded successfully from path: cache\last_weights.pt
Evaluate: 100%|████████████████████████████████████████| 19/19 [00:06<00:00,  2.80it/s]
Loss: 0.991, Acc: 0.693
Weights loaded successfully from path: cache\best_weights.pt
Evaluate: 100%|████████████████████████████████████████| 19/19 [00:06<00:00,  3.10it/s]
Loss: 0.991, Acc: 0.693
```

3. Deeper !

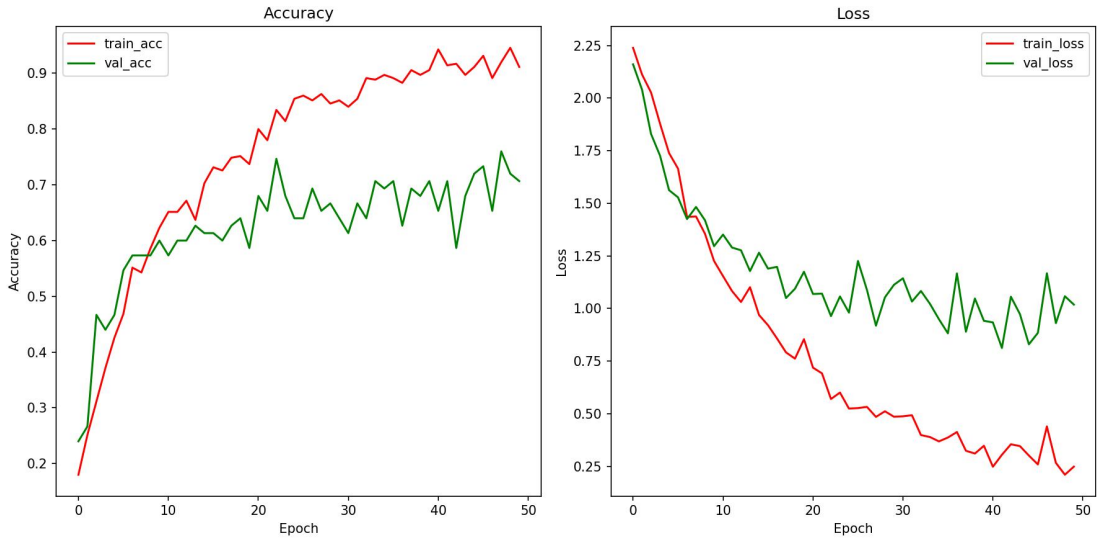
```
    nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(inplace=True),

    nn.Conv2d(256, 512, kernel_size=3, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),

    nn.AdaptiveAvgPool2d((1, 1))
)
```




```
Weights loaded successfully from path: cache\last_weights.pt
Evaluate: 100%|| 19/19 [00:06<00:00,  2.76it/s]
Loss: 1.187, Acc: 0.707
Weights loaded successfully from path: cache\best_weights.pt
Evaluate: 100%|| 19/19 [00:06<00:00,  3.03it/s]
Loss: 0.884, Acc: 0.800
```

Better! But over-fitting a little

Include Dropout: To prevent overfitting, especially when I increase the model's complexity, I added dropout layer after activation functions.




```
Weights loaded successfully from path: cache\last_weights.pt
Evaluate: 100%|| 19/19 [00:07<00:00,  2.42it/s]
Loss: 0.909, Acc: 0.747
Weights loaded successfully from path: cache\best_weights.pt
Evaluate: 100%|| 19/19 [00:08<00:00,  2.34it/s]
Loss: 0.756, Acc: 0.813
```

Improved with 0.1 point!

Change Kernel Sizes or Strides: I have experimented with different kernel sizes and strides in my convolutional layers. I understood that, smaller kernels (e.g., 3x3) can capture finer details in the image.

Optimize Learning Rate: Initially, I used a learning rate of 0.01, but found that it led to unstable and erratic updates during training. To address this, I later adjusted the learning rate to 0.0001, which resulted in more stable and gradual learning, allowing the model to converge more effectively and improve its performance on the validation set. This change underscores the importance of fine-tuning hyperparameters to achieve optimal training results.

Future ideas to improve accuracy of the model

Regularization : Besides dropout, I might consider other regularization techniques like L2 regularization (weight decay in the optimizer).

Advanced Optimizers: Experiment with different optimizers like SGD with momentum, RMSprop, or AdamW.

Pooling Layers: Experiment with different pooling strategies (max pooling, average pooling) or different pooling sizes to see how they affect performance.

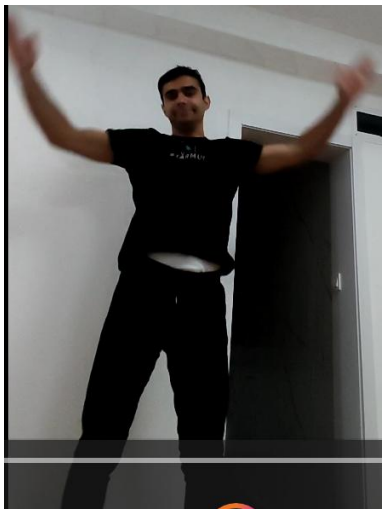
Results Analysis

The experiments revealed that:

- The pre-trained ResNet18 provided better initial accuracy.
- Lower learning rates yielded more stable training progress.
- Making CNN model more complex by adding conv and maxpool layers increased accurcy.

Demo

You can find my own Jumping Jack video on resources/demo.mp4 or test with your motion from 10 classes. The model correctly classifies my motion as jumping jack. Pretty cool!



```python
# Load the video
video_path = 'resources/demo.mp4'
video_tensor = read_video(video_path, params['num_frames'], transform, params['img_size'])

# Perform inference
loss_fn = torch.nn.CrossEntropyLoss()
predicted_label, loss = infer_single_video(model, video_tensor, loss_fn, device)
print(f"Predicted Label: {activity_labels[int(predicted_label)]}")
```

demo. py

```
PS D:\edu\SJTU\3-1\ML\hmw\lab3\521030990006_VAHAGN\code> & C:/U
d:/edu/SJTU/3-1/ML/hmw/lab3/521030990006_VAHAGN/code/demo.py
Predicted Label: JumpingJack
```

Conclusion

This project successfully demonstrates the use of a CNN-LSTM architecture for video classification. The experiments provide valuable insights into the impact of different architectures and hyperparameters on model performance.

The combination of CNN for spatial feature extraction and LSTM for temporal analysis allows the model to effectively understand and classify complex video content. This dual approach ensures that both the individual content of each frame and the movement and actions across frames are considered, leading to more accurate video classification.