Human Action Recognition in Videos

Sapienza Universitá di Roma

Yusupha Juwara

yusuphajuwara@gmail.com

Abstract

Video analysis is an important research area in computer vision that has applications in many fields such as surveillance, healthcare, sports analysis, human-computer interaction, and much more. **Video action recognition**, a subset of video analysis, is vital too. It involves the classification of human actions in videos. In recent years, deep learning-based approaches have achieved remarkable success in this field. Deep learning models have been able to learn discriminative features automatically from the raw video frames, which has led to significant improvements in video action recognition performance. In this project, I aim to implement from scratch, demonstrate the effectiveness of, and explore three (plus one) different models for **human action recognition in videos**. Specifically, three of these models, **Resnet3D**, **R2Plus1D**, and **ConvLSTM**, are **trained from scratch**, while the (plus one) fourth, *R3D*, is a **pretrained model fine-tuned** for the task at hand.

The performance of each model is evaluated using accuracy and loss metrics.

My experiments show that R3D and R2Plus1D, which shares the same architecture (one trained from scratch, the other pretrained), outperforms the other models in terms of accuracy and training time. Finally, I demonstrate that fine-tuning the pre-trained R3D model leads to comparable performance to training from scratch.

Overall, my project contributes to the field of video action recognition by implementing from scratch, demonstrating the effec- tiveness of, and exploring several state-of-the-art models on a large and diverse dataset.

1 Introduction

1.1 ResNet3D

ResNet3D [1] is an extension of the ResNet architecture that incorporates the use of 3D convolution layers to capture **spatio-temporal features** from video frames. By considering both spatial and temporal information simultaneously, ResNet3D is able to extract more informative features from video data, which is essential for accurate action recognition.

Similar analogy to preventing **signal attenuation** [1], ResNet3D uses **skip connections** and **residual blocks** to help address the issue of **vanishing gradients** that often arise in very deep networks, allowing the network to bypass certain layers and better propagate the signal through the network. Each residual block contains two 3D convolution layers, followed by batch normalization and ReLU activation. The output of these layers is then combined with the input to the residual block through a skip connection. This enables the network to learn more complex features by allowing it to bypass certain layers in the network.

In contrast to traditional 2D convolution layers, 3D convolution layers are able to capture both spatial and temporal information simultaneously by considering multiple frames of a video at a time. This allows ResNet3D to capture motion information, which is a crucial aspect of video action recognition.

Overall, ResNet3D's ability to capture both spatial and temporal information simultaneously, coupled with its use of residual blocks and skip connections, make it an effective tool for modeling complex patterns in video data.

1.2 R2Plus1D = R(2+1)D

R(2+1)D [7] is an innovative extension of the ResNet architecture that leverages both 2D and 1D convolutions to capture spatio-temporal features from video frames. This variant is an improvement over the traditional ResNet architecture. It has 1D convolutions that can capture temporal information and 2D convolutions that can capture spatial information from video frames, making it more suitable for video action recognition tasks.

The (2+1)D convolution block in R(2+1)D factorizes the 3D convolution into two separate

[1] Signal attenuation refers to the weakening or loss of signal strength as it propagates through a medium. This phenomenon can occur in various domains, including signal processing, telecommunications, and optics. Attenuation can result in a decrease in signal quality, making it challenging to extract useful information from the signal.

and successive operations - a 2D spatial convolution and a 1D temporal convolution. This approach is advantageous because it allows for additional non-linear rectification between the two operations. By doing so, it effectively doubles the number of nonlinearities compared to a network using full 3D convolutions for the same number of parameters. As a result, the R(2+1)D model can represent more complex functions and is capable of capturing spatio-temporal features more efficiently.

Furthermore, the factorization of the 3D convolution facilitates optimization, which yields lower training and testing losses. This is because the factorization reduces the number of parameters in the model and simplifies the optimization process. Consequently, the R(2+1)D model trains faster and with less computational resources, making it more practical for large-scale video recognition tasks.

Overall, R(2+1)D is a significant advancement in the field of video action recognition, providing a more effective way to extract spatio-temporal features from video frames. Its innovative design, combining both 2D and 1D convolutions, allows for more efficient and accurate video analysis, opening new possibilities for applications such as surveillance, robotics, and autonomous driving.

1.3 Convolutional LSTM (ConvLSTM)

ConvLSTM [6] is a type of **recurrent neural network** that has been applied to video analysis tasks, such as video prediction and action recognition. Unlike traditional LSTMs, ConvLSTMs operate on 2D spatial and 1D temporal sequences by incorporating convolution layers in the LSTM cells.

The ConvLSTM architecture has a similar structure to the LSTM, but each cell contains convolution layers in addition to the gating units. These convolution layers allow ConvLSTMs to learn spatial and temporal features simultaneously, making them well-suited for video analysis tasks. The input to a ConvLSTM cell is a 3D tensor, consisting of the current input frame and the hidden state and cell state from the previous time step.

The convolution layers in ConvLSTM cells allow the network to learn spatial and temporal features jointly, which is especially useful for tasks like video action recognition, where both the spatial and temporal aspects of the video are important.

By incorporating convolution layers, ConvLSTMs can capture both **short-term** and **long-term dependencies** between frames, making them more effective for tasks where temporal context is critical.

1.4 R3D

R3D is a pre-trained model (imported from torchvision), which I fine-tuned end-to-end to see how it would compare to the other models that were not pre-trained on any data a priori. This pretrained model serves as a guide to tweak the hyper-parameters used for training the models in order to optimize them to have a very close loss and accuracy to this pretrained one. R3D and R2Plus1D share the same architecture, with the only difference being that R3D is already pre-trained, its classifier layer modified, and is fine-tuned end-to-end.

Memory Requirements

The **ConvLSTM** model has been found to have a **much higher memory footprint** compared to the other models, namely: **ResNet3D** and **R2Plus1D**. This implies that the **ConvLSTM requires a significantly larger amount of memory** to store and process data, which can limit its usability on devices with limited resources.

To address this issue, I had to make some adjustments in the ConvLSTM model's architecture. For example, the resolution of the input data was reduced, which means that the size of the video frames was made smaller. Additionally, the number of frames per video and layers were also decreased compared to the other models. This was necessary to reduce the memory requirements of the ConvLSTM model, which otherwise would have exceeded the available memory and led to a **Memory Out Of Bounds error**.

It is worth noting that the memory requirements of deep learning models can vary significantly depending on their architecture, the size of the input data, and the number of parameters they contain. As a result, I carefully consider these factors when designing the architectures and training the models to ensure that they can run efficiently.

2 Model Architectures

2.1 ResNet 3D with 18 Layers

ResNet 3D with 18 Layers (ResNet3D-18) is a variant of the ResNet3D architecture that has 18 layers. The architecture consists of **four residual groups**, each of which contains **two residual blocks** [2]. Each **residual block** in turn contains one 3D convolution layer with a

[2]Notice that in some articles the **residual groups** are called the **residual blocks**. However, here in this document, for a better documentation, I have distinguished between the two

kernel size of (3, 3, 3), followed by a batch normalization, and a ReLU activation. A **skip connection** bypasses these two layers/blocks and adds the input to the output of the **residual groups**. The skip connections downsample the input through them, if necessary, to have the same shapes as the output of the residual groups. This allows the network to learn residual features and address the vanishing gradient problem. The first residual block does not downsample the input that passes through it, while the other three blocks downsample the spatial dimensions but they all increase the temperal dimensions as seen in the figure 1.

The input to the network is a 3D tensor of video frames. The first layer is a 3D convolution layer with a kernel size of (3, 7, 7), which applies 3D convolution on the input video frames to downsample and extract spatio-temporal features.

There are four residual groups as explained above. The output of the last one of which is passed through a global average pooling layer, which computes the average of each feature map output and returns a single feature vector. This is then passed through a fully connected layer to produce a vector of class scores, indicating the probability of each class for the input video. For a better view of the **ResNet3D-18** architecture, including the kernel sizes, input/output shapes at each layer, residual blocks and skip connections, click this external link here

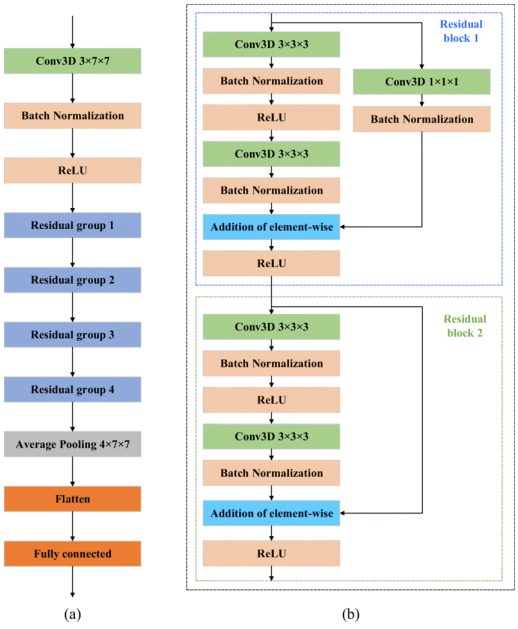

Figure 1: Resnet3D Architecture of 18 layers. Each residual group consists of two residual blocks. The first residual group does not have any convolution layer in the skip connection because it did not downsample, while the other three have a convolution layer in their first residual blocks to downsmaple the spatial dimensions and upsample the channel dimensions

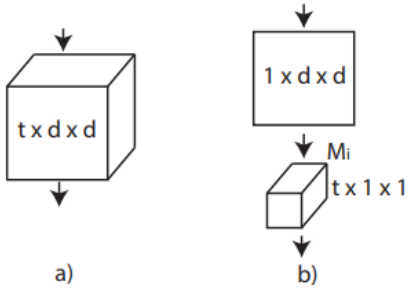

Figure 2: **(2+1)D vs 3D convolution**. The illustration is given for the simplified setting where the input consists of a spatiotemporal volume with a single feature channel. (a) Full 3D convolution is carried out using a filter of size t × d × d where t denotes the temporal extent and d is the spatial width and height. (b) A (2+1)D convolution block splits the computation into a spatial 2D convolution followed by a temporal 1D convolution. The numbers of 2D filters (M_i) are chosen so that the number of parameters in the (2+1)D block matches that of the full 3D convolution block.

2.2 ResNet 2Plus1D (R2Plus1D)

R2Plus1D is just an improvement over the Resnet3D as already explained. They share similar architecture, with the main difference being that R2Plus1D factorizes the 3D Convolutions in ResNet3D to a 1D, for temporal, and a 2D, for spatial features as shown in figure 2. To have a better idea about how the 3D in ResNet3D factorizes to a 2D, for a spatial, and 1D, for a temporal, in the R2Plus1D, click the external link here

2.3 Convolutional LSTM (ConvLSTM)

ConvLSTM, as shown in figure 4 and 3 as two variants, is a type of recurrent neural network that integrates convolutional layers within the LSTM units. It takes an input with shape (b, t, c, h, w) and outputs, as shown in the one in figure 4, the last sequence of the last layer. It consists of **convolutional memory cells**, each of which consists of four main parts: the **input gate**, the **forget gate**, the **output gate**, and the **memory cell**, as shown in figure 5.

The **input gate** controls the flow of information from the input to the **cell state**. It is a sigmoid layer that decides which parts of the input to update and which parts to discard.

The **forget gate** controls the flow of information from the previous state to the current state. It decides which parts of the previous state to keep and which parts to forget. The forget gate is also a sigmoid layer.

The **output gate** controls the flow of information from the cell state to the output. It decides which parts of the cell state to output. The output gate is a sigmoid layer followed by a tanh layer.

the **memory cell** in an LSTM network is responsible for maintaining a long-term memory of

previous inputs, which can be used to make predictions about the current input. The memory cell takes in both the previous cell state and the current input and computes a candidate value for the current cell state, which is combined with the previous cell state to produce the updated cell state.

The **memory gate** is a sigmoid layer that acts as a switch, controlling the flow of information from the previous cell state to the current cell state. It takes the previous cell state and the current input as inputs and decides which parts of the previous state to keep and which parts to discard. The gate output is between 0 and 1, and any information with an output close to 0 will be discarded, while information with an output close to 1 will be retained.

Together with the forget gate, which determines which parts of the previous cell state to forget, the memory gate ensures that only relevant information from the previous state is carried forward to the current state. This mechanism helps prevent the vanishing gradient problem and allows LSTMs to maintain long-term dependencies in sequential data.

Furthermore, ConvLSTM can stack layers to form a multi-layer ConvLSTM, as shown in both figures 3 and 4.

Overall, the ConvLSTM architecture allows for the efficient processing of spatio-temporal data by integrating convolutional layers with LSTM units and incorporating memory gating mechanisms to control the flow of information.

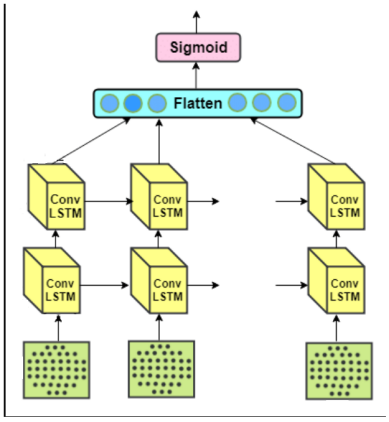

Figure 3: Convolutional LSTM with outputs of all the sequences of the last layer. This variant can be used in tasks similar to sequence-to-sequence, where all the sequences of the last layer are needed

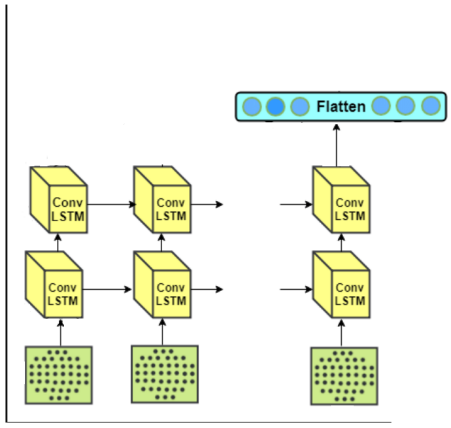

Figure 4: Convolutional LSTM with the output of only the last sequence of the last layer. This is the variant used in this project. It is less memory consuming w.r.t to the one in figure 3

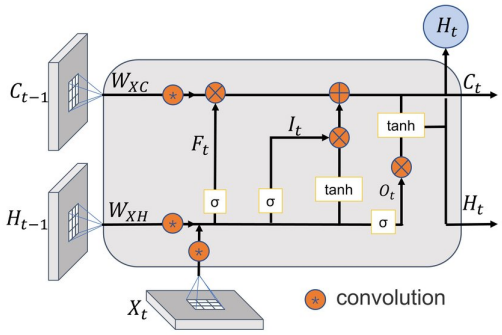

Figure 5: A convolutional LSTM with convolutional operations

3 Dataset

In this project, I trained and evaluated the models for the task at hand on the **UCF101 dataset**. On **Kaggle**, the dataset can be added by just clicking on **Add Data** and then **type Ucf101 in the search field**. The UCF101 dataset consists of **13,320 videos** of human actions of over **27 hours** of video data, which are divided into **101 action categories**. The videos have different lengths and resolutions. So, I resize them and do common data augmentation techniques such as random cropping, rotatation, flipping, etc., as discussed in Data Augmentation

To train the models, I split the dataset into **8:2 ratio for train and test**, and then further split the train dataset in the ratio **8:2 for train and evaluation**. So, the training set consists of $0.8 \times 0.8 \times 13,320 \approx 8,525$ videos, the validation set is $0.8 \times 0.2 \times 13,320 \approx 2,131$ - used to tune the hyperparameters of the models and prevent overfitting (more on this later), and finally the test set is $0.2 \times 13,320 \approx 2,664$ videos - used to evaluate the performance of the models.

Training and evaluating the models on the UCF101 dataset allowed me to assess their performance on a diverse set of human actions in various scenarios. The dataset was split into training, validation, and test sets to ensure that the models were not overfitting to the training set and could generalize well to new data.

But why not a smaller dataset like ucf50 for faster training, or a larger one with many action categories for a more robust model? The UCF101 dataset is used because it is considered to be an appropriate dataset size to train a **very deep model from scratch** while still being able to generalize well to unseen data. The decision not to use a smaller dataset like UCF50 was based on the fact that human action recognition involves many actions beyond just 50, and a smaller dataset may not provide enough variety to train a robust model. On the other hand, larger datasets like Sports-1M require a considerable amount of computational resources, which is not available to me. In this case, I used **Kaggle GPU**s to train the models but had limited **GPU time**, which led to the need for a dataset of modest size that is sufficcient enough to train a robust model.

Training deep learning models on large datasets with many categories requires a lot of computational resources and time. However, it is necessary to achieve high accuracy and generalization in tasks such as human action recognition. Using a smaller dataset may not provide enough variety to capture the nuances of different human actions, while using a larger dataset may require an unreasonable amount of time and compute. Therefore, using a dataset that strikes a balance between size and computational requirements is essential for achieving a robust and generalizable for the task at hand.

Now, imagine using just 20 frames per video of size (h=112, w=112, c=3) for 13,320 videos. That is above three billion and thus huge for the limited compute at my disposal. So, using a larger dataset for this task is out of reach for me.

Overall, training and evaluating the models on the UCF101 dataset provided insights into their effectiveness for action recognition in real-world scenarios.

3.1 Data Augmentation

Here, I applied a series of **transformations to the input image to create a variation thereof**, which can help to increase the diversity of the data and prevent overfitting. The transformations include:

- **Resizing and cropping** the image to a random size and aspect ratio using the **RandomResizedCrop transformation**. This allows the model to learn to recognize objects at different scales and positions in the image.
- **Rotating** the image by a random angle between -15 and 15 degrees using the **RandomRotation transformation**. This can help the model to learn to recognize objects from different viewpoints. Notice that the range (-15,15) is just a small tilting of the image.
- **Center cropping** the image to the desired size using the **CenterCrop transformation**. This ensures that the image has the same size and aspect ratio as the input to the model.
- **Converting the image to a tensor** and **normalizing** it using the **ToTensor** and **Normalize transformation**s. This converts the image to a format that can be processed by the model and scales the pixel values to a range between 0 and 1.

This defines **two Compose transformation**s: one for the training data and another for the test data. The training data transformation includes all the transformations, while the test data transformation only includes resizing, center cropping, converting to a tensor, and normalization.

3.2 Frame Extraction

Here, I extracted a fixed number of frames (SEQUENCE_LENGTH) from each video file. The extracted frames are then preprocessed using a data augmentation transform technique as explained above base on whether the data is used for training or testing/validation. Note that for the pretrained model, **R3D**, I only resize the data and then convert from **PIL Image** to a tensor of the same type - this does not scale values. This is so because I used the **preprocess method of the model itself** that comes with its weights

I first initialized a VideoCapture object from OpenCV, which reads the video file at the specified path. The total number of frames in the video is then obtained using the **get** method of the VideoCapture object.

Then I used a step value to determine which frames to extract from the video file. The step value is calculated as the maximum of 1 and the total number of frames divided by the desired sequence length (SEQUENCE_LENGTH). This extracts frames at SEQUENCE_LENGTH equally spaced intervals, starting from the first frame [3].

For each extracted frame, I applied the given data augmentation transform if **use_transform** is True, or applies a default transform if **use_transform** is False - used for the training data of the pretrained model only. The transformed frame is then appended to a list of frames.

[3]There are other techniques of extracting frames based on the information contained in the frames, though.

If the number of extracted frames is less than SEQUENCE_LENGTH, I add more frames to the list by duplicating the last extracted frame until the desired sequence length is reached [4].

3.3 Efficient Dataloading Pipeline

Efficient data loading is a critical aspect of machine learning projects, particularly in avoiding the bottlenecking of the training process. To address this issue, PyTorch provides the **DataLoader** class, which efficiently loads data **batch-by-batch**. To further enhance this process, I created a new dataset class called **LoadDataset**, which extends the Dataset class and takes in a few parameters, one of which is the transform object defined above 3.1 for data augmentation in order to have a robust model that generalizes well to new unseen data.

The LoadDataset class overrides the __len__() and __getitem__() methods to provide the number of samples in the dataset and the process to load and preprocess data for a specific sample, respectively. These methods can be customized based on the requirements of the project.

The DataLoader class and LoadDataset work together to load and preprocess data in parallel, significantly reducing memory requirements and maximizing the use of computing resources. Additionally, the DataLoader class can **shuffle data**, **split it into batches**, and **load it in parallel across multiple workers**, further improving the efficiency of the data loading pipeline.

By optimizing the data loading process using the DataLoader and LoadDataset classes, it is ensured that my models train faster (compared to not using it) and more efficiently.

3.4 Class Imbalance

In any classification task, it is common to encounter **class imbalance**, where some classes have significantly more samples than others. This can lead to poor performance on the minority classes since the model is biased towards the majority classes. To solve this problem, I adjusted the loss function to give more weight to the minority class. I calculated the class weights based on the number of samples in each class.

By using these class weights in the loss function, the model can give more weight to the minority classes and prevent being biased towards the majority classes. This can improve the overall performance of the model, especially in cases of severe class imbalance.

[4] I also tried adding a random noise to make the model more robust rather than duplicating the last frame, but that did not work well.

4 Hyper-parameters

The success of deep learning models depends largely on selecting appropriate hyperparameters, such as the **batch size**, **learning rate**, and **number of epochs**. These hyperparameters determine the speed of convergence, overfitting, and model performance. In this project, I defined these key hyperparameters carefully and conducted experiments to fine-tune them for optimal performance.

Other hyper-parameters in this project include, but not only:

- **patience**: the number of epochs to wait before stopping the training if the validation loss has not improved for at least **epsilon_increment**
- **epsilon_increment**: the minimum percentage improvement in the validation metric (accuracy or loss) required to continue training the model

,

5 Optimizers

I used the **Adam optimizer** [3], which is a widely used optimization algorithm for stochastic gradient descent [5] in deep learning. The optimizer updates the weights of the neural network during training, aiming to minimize the loss function. However, selecting the right learning rate is crucial for the optimizer to achieve convergence to the optimal solution. Therefore, I used a **learning rate scheduler** to dynamically adjust the learning rate during training. This strategy is known as **learning rate annealing**, which helps to prevent overfitting and improve model generalization. The learning rate scheduler used reduces the learning rate by a certain factor when the validation loss did not improve for a certain number of epochs. By using this approach, the optimizer can converge more efficiently to the optimal solution, which results in improved model performance and better generalization.

6 Loss

The choice of a suitable loss function is crucial since it determines the model's ability to learn and make accurate predictions. The loss function measures the difference between the predicted output and the actual target output, and the goal of the training process is to minimize this difference.

In this project, I used the **Cross Entropy Loss**, which is a commonly used loss function in

deep learning for classification tasks. It is particularly useful when the output of the model is a probability distribution over multiple classes. The loss function computes the negative log-likelihood of the predicted class probabilities given the true class labels.

It is worth mentioning two important parameters to the loss function:

- The **weight** (3.4) parameter in the **CrossEntropyLoss** function is used to assign different weights to each class. This is useful when dealing with Class Imbalance datasets where some classes have much fewer samples than others. By assigning higher weights to the minority class, the model is encouraged to pay more attention to these samples and learn to classify them accurately.
- The **reduction** parameter determines how the loss is aggregated over the batches. The default value, "mean", computes the mean loss over all the samples in the batch [5].

In summary, choosing an appropriate loss function is crucial for training deep learning models. The 'Cross Entropy Loss' function provided by pytorch allows for assigning weights to classes and choosing different reduction options to aggregate the loss over batches.

7 Overfitting and Underfitting

Overfitting and underfitting are common issues when training deep learning models.

Overfitting:

- Overfitting can occur when the model learns to fit the training data too closely, resulting in poor generalization to new, unseen data. To address this issue, regularization techniques such as dropout, weight decay, and early stopping [6] can be used.
- Another effective approach is to adjust the learning rate of the optimizer during training, which can prevent the model from learning too quickly and overfitting to the training data. By reducing the learning rate, the optimizer can take smaller steps towards the minimum of the loss function and prevent the model from getting stuck in local optima or oscillating around the optimal solution. This approach is known as **learning rate annealing** [4], which helps to prevent overfitting and improve the model's generalization performance on new, unseen data.
- Furthermore, **Batch normalization** [2] can also act as a form of regularization by stabilizing and regularizing the training process. It reduces the internal covariate shift, which is the phenomenon of the distribution of the input to a layer changing during training due to the changing weights of the previous layers. By normalizing the activations, batch normalization can help to reduce the effect of this phenomenon and make the training process more stable. Moreover, batch normalization can reduce the dependence of the model on specific weight initializations, which in turn can improve the model's generalization performance. Thus, using a combination of regularization techniques and batch normalization can help to prevent overfitting and improve the generalization performance of the models.

Underfitting:

- Underfitting occurs when the model is too simple to capture the complexity of the data, resulting in poor performance on both the training and test data. Techniques to address underfitting include increasing the model's capacity - which in my case is an 18 layer nerwork that is enough for the aforementioned dataset as shown in figure 1, adjusting the hyperparameters - as explained in 4, and improving the quality (and quantity) of the training data - as explained in 3.1.

Addressing both overfitting and underfitting requires a combination of regularization techniques, hyperparameter tuning, and data augmentation. By effectively using these techniques, the generalization performance of the used models were improved further, resulting in better performance on new, unseen data.

8 Training and Validation

The process of training and validating deep learning models is crucial for their success. The training loop is a commonly used procedure that

[5] Other options for the **reduction** parameter in **CrossEntropyLoss** include "sum", which sums the loss over all the samples in the batch, and "none", which returns the individual loss values for each sample in the batch.

[6] For the early stopping, I used a trick to save the best model during training based on metrics on the validation dataset. This means that subsequent models (that were overfitted to the training dataset) that do not outperform this saved-best model on the metrics on the validation dataset are not used during the final testing on an unseen dataset. The training stops if the model does not improve for a certain number of epochs

involves various inputs such as the model architecture, optimizer, learning rate, number of epochs, and several hyperparameters. The procedure initializes several variables to monitor the training progress and iterates over a specified number of epochs.

During each epoch, the model is set to training mode, and training data is fed to the model in batches. The procedure performs a forward pass through the model, calculates the Loss using a loss function that measures the difference between predicted outputs and the ground truth labels, and updates the model parameters using the Optimizers. The procedure tracks the training loss and accuracy for each epoch.

After the completion of the training loop, the model is switched to evaluation mode, and the validation data is fed to the model in batches. The procedure computes the loss between the predicted outputs and the ground truth labels and calculates the validation loss and accuracy for each epoch.

The procedure saves the model's best weights and optimizer state based on the monitored metric, which is typically the validation accuracy (as in this case). It also includes a specified **patience** parameter, which determines how many epochs the training process will wait for an improvement in the monitored metric before terminating the training early.

This procedure is a highly flexible and adaptable mechanism that allows to explore different hyperparameters, optimizer settings, and model architectures to improve the model's performance. Note that in classification tasks like the one at hand, it is common to choose the best model based on the best validation accuracy rather than the best validation loss, even though it is the loss that is being minimized.

9 Results

The performance of the best saved models are measured based on training loss and accuracy, validation loss and accuracy, test accuracy, and the average of train, validation, and test accuracy. These are video-level metrics instead of clip-level, and for the accuracy, only the top-1 are reported instead of, say, top-5. This, in some published papers, is called Video@1.

Note: When I say lower training loss, I mean the minimum loss among all the model's training losses accumulated in all the epochs as shown in, for example, figure 8. Similar language applies when I say higher training, validation, or testing accuracy (also applies for highest, lowest and similar)

Losses

For the presentation of the results, the pre-trained R3D, fine-tuned end-to-end, get the least training and validation losses. This is because it was pre-trained on a diverse dataset called kinetics and with end-to-end fine-tuning, and of course carefully hyper-parameter tweaking, it was able to optimize its weights for the task at hand. As shown in the figure 6, it has a training loss of **0.1918** and evaluation loss of **0.3018**.

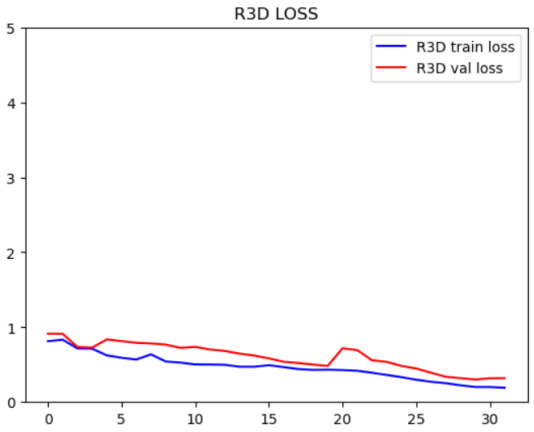

Figure 6: Training and validation losses of **R3D** over all the epochs. It has training loss of 0.1918 and evaluation loss of **0.3018**

After the pre-trained model R3D, R2Plus1D gets a lower loss for the validation than both the ResNet3D and ConvLSTM, but a negligible higher loss for the training than ResNet3D. This means that decoupling the spatial and the temporal dimensions was a key factor for R2Plus1D to fit well the validation dataset (and the test dataset, as will be discussed below) than ResNet3D. For R2Plus1D to have a lower validation loss than all but R3D, this is not surprising since it shares the same architectural design with R3D, with the only difference being that R3D was already pre-trained and then fine-tuned. Even with that, R2Plus1D was very close to the losses of R3D. It has a training loss of 0.2589 and evaluation loss of **0.4195**. As shown in figure 7, even after terminating its training, its loss continues to go down a little further, which suggests that with further training, it may continue to decrease its loss on par with R3D (6).

Similar to the R2Plus1D, ResNet3D also got a fairly smooth and stable loss curve as shown in figure 8. It has a training loss of 0.2418 and evaluation loss of **0.5218**. It has a lower training loss than R2Plus1D and ConvLSTM, and a lower validation loss than only ConvLSTM. This means that R2Plus1D was able to generalize a little well than ResNet3D, and this difference did in fact reflect on the test accuracy were R2Plus1D beats it by few percentages as shown in figure 15.

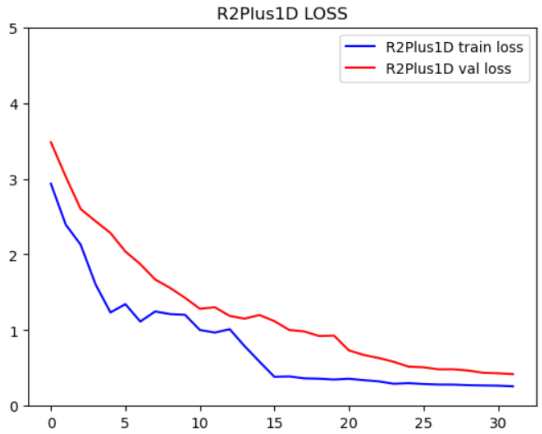

Figure 7: Training and validation losses of **R2Plus1D** over all the epochs. It has training loss of **0.2589** and evaluation loss of **0.4195**

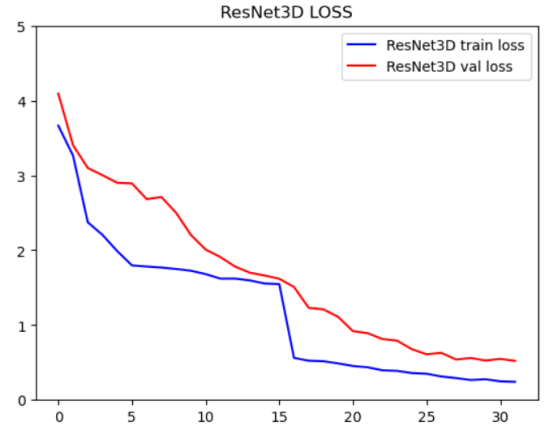

Figure 8: **ResNet3D** has a training loss of **0.2418** and evaluation loss of **0.5218**

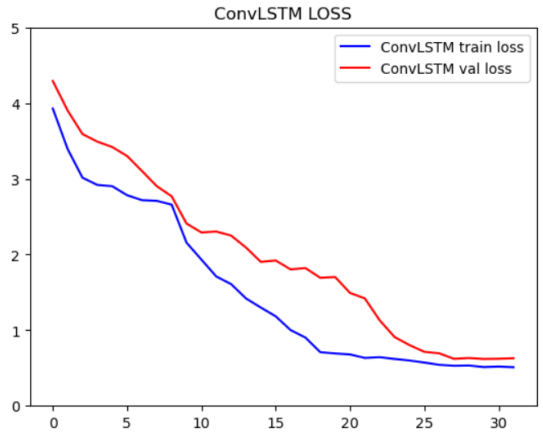

Figure 9: **ConvLSTM** has a training loss of **0.5103** and evaluation loss of **0.6192**

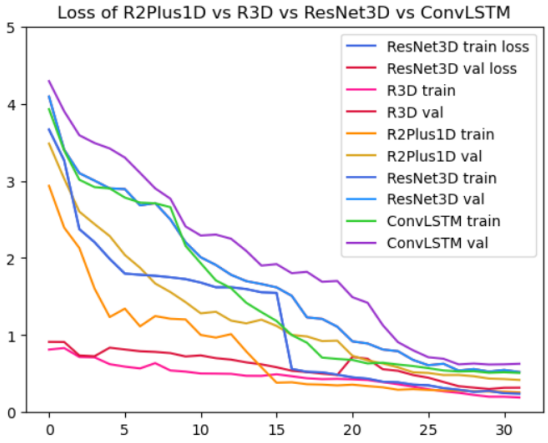

Figure 10: This compares the losses of all the models accumulated during training across all the epochs

ConvLSTM got a higher loss compared to the rest. This means that it did not fit the data well compared to the others. As explained earlier, there are a few reasons for that, two of which are the smaller number of layers and resolution used. It has a training loss of 0.5103 and evaluation loss of **0.6192**, which are closer to the other models'. However, this little difference reflected on the test accuracy, where ConvLSTM got the lowest again as shown in figure 10.

Overall, in terms of loss as shown in the figure 10, the pre-trained R3D model achieved the best result - training loss of **0.1918** and evaluation loss of **0.3018**, followed by the R2Plus1D model - training loss of 0.2589 and evaluation loss of **0.4195**, then ResNet3D - training loss of 0.2418 and evaluation loss of **0.5218**, and lastly ConvLSTM - training loss of 0.5103 and evaluation loss of **0.6192**.

Accuracy

In this section, the accuracy of the four models is evaluated on the action recognition task using the top-1 accuracy metric. The reported accuracies are video-level metrics, indicating the percentage of videos that the model correctly classified out of all the videos in the test set.

Note: the accuracies are x out of 1 (x/1). E.g., for R3D, the accuracy for the testing dataset is 0.8652

9.0.1 R3D

The pre-trained R3D model achieved the lowest training and validation losses 6, indicating that it captured the underlying patterns in the training data better than the other models. This led to better generalization performance and higher accuracy on the test set. As shown in Figure 11, R3D model achieved the highest accuracy among the four models, with a test accuracy of **0.8652**. It outperforms the other models consistently throughout the training process, but only slightly against the R2Plus1D.

9.0.2 R2Plus1D

R2Plus1D achieved the second-highest accuracy, with a test accuracy of **0.8256**. As shown in Fig-

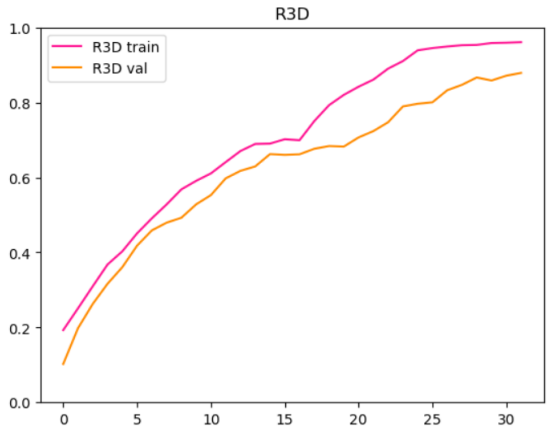

Figure 11: Training and validation accuracies of **R3D** over all the epochs. It has training accuracy of **0.9608**, evaluation accuracy of **0.8788**, and a testing accuracy of **0.8652**

ure 12, R2Plus1D closely follows R3D throughout the training process, and the difference in performance between the two models is small. Similar to the loss, R2Plus1D's accuracy keeps improving slightly even after the training termination. This entails that if training continues for a few more epochs, it may catch up on R3D.

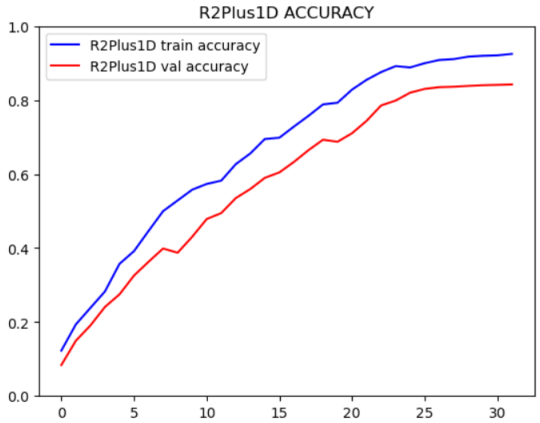

Figure 12: **R(2+1)D** has training accuracy of **0.9248**, evaluation accuracy of **0.8422**, and a testing accuracy of **0.8256**

9.0.3 ResNet3D

ResNet3D achieved a test accuracy of **0.7621**, which is, as similar to the case of R2Plus1D and R3D, slightly lower than R2Plus1D. This means that factorizing the spatial and temporal components of the R2Plus1D actually simplifies the optimization process for the R2Plus1D model to converge faster than ResNet3D. However, similar to the accuracy of R2Plus1D, ResNet3D's accuracy also seems to be slightly increasing as shown in Figure 13, and if the training process continues, it may catch up as well because it is also being optimized but does not eases the optimization process as much as R2Plus1D does. These results are not surprising because all the three above shares the same architecture with just a small difference.

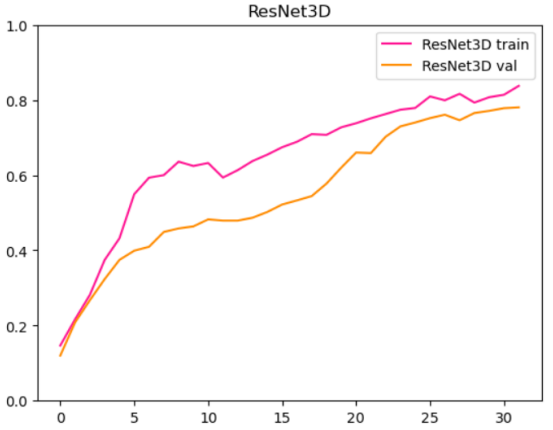

Figure 13: **ResNet3D** has training accuracy of **0.8377**, evaluation accuracy of **0.7805**, and a testing accuracy of **0.7621**

9.0.4 ConvLSTM

ConvLSTM achieved the lowest accuracy among the four models, with a test accuracy of 0.6482 as shown in figure 14 . Figure 15 shows that it consistently underperforms the other models throughout the training process. It is not the best fit for the dataset. This is because the model **doesn't have enough capacity to fit the data well**. It uses fewer layers compared to the other three models and has a larger memory footprint. Therefore, there needs to be caution when choosing models for video data and consider their capacity, memory footprint, and other hyperparameters.

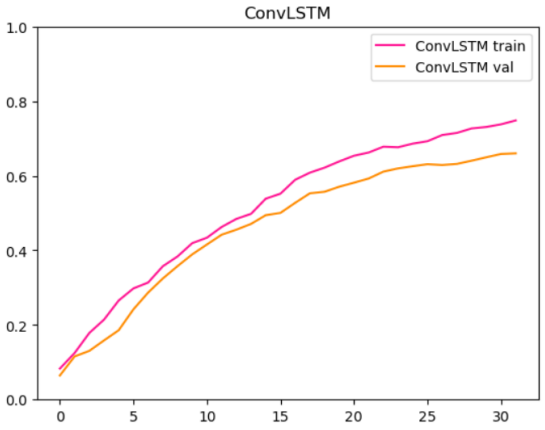

Figure 14: **ConvLSTM** has training accuracy of **0.7483**, evaluation accuracy of **0.6601**, and a testing accuracy of **0.6482**

Overall, in terms of accuracy as shown in the figure 15, the results are consistent with the loss results discussed earlier in figure 10. The pre-trained R3D model achieved the best result - testing accuracy of **0.8652**, followed by the R2Plus1D model - testing accuracy of **0.8256**, then ResNet3D - testing accuracy of

Model	Train Loss	Train Acc	Val Loss	Val Acc	Test Acc	Avg Acc
R3D	0.1918	0.9608	0.3018	0.8788	0.8652	0.9016
R2Plus1D	0.2589	0.9248	0.4195	0.8422	0.8256	0.8642
Resnet3d	0.2418	0.8377	0.5218	0.7805	0.7621	0.7934
ConvLstm	0.5103	0.7483	0.6192	0.6601	0.6482	0.6855

Table 1: Summary of the performances of the models based on the train loss and accuracy, validation loss and accuracy, test accuracy, and the average of train, validation, and test accuracy.

0.7621, and lastly ConvLSTM - testing accuracy of **0.6482**.

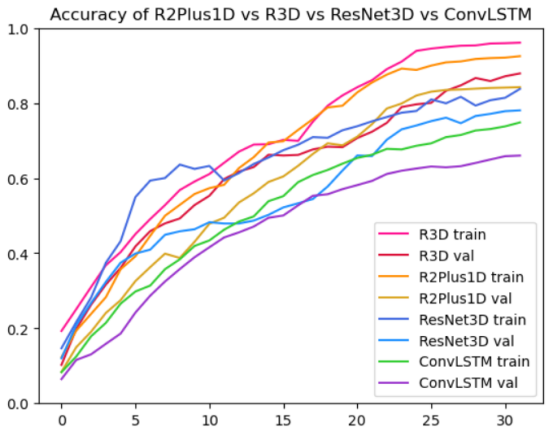

Figure 15: Accuracy of the four models during training and validation.

Table 1 summarizes and measures the performance of these models based on the train loss and accuracy, validation loss and accuracy, test accuracy, and the average of train, validation, and test accuracy. The pre-trained R3D model achieved the lowest training and validation losses, indicating that it captured the underlying patterns in the training data slightly better than the other models. This led to better generalization performance and higher accuracy on the test set.

Even though the pretrain model got the highest accuracies and lowest losses, it is still comparable with those trained from scratch (R2Plus1D and ResNet3D). We can observe these in the figures that compare the losses 10 and accuracies 15 of all the models on the training and validation datasets.

Similarly, R2Plus1D and ResNet3D achieved lower validation losses than ConvLSTM, and this was reflected in their higher accuracy on the test set. ConvLSTM, with the highest validation loss, had the lowest accuracy on the test set.

10 Conclusion

The results indicate that pre-training on a diverse dataset like **Kinetics**, fine-tuning the model on the target task, and carefully tuning the hyperparameters are critical for achieving high performance on action recognition tasks. Among the four models evaluated in this project, R3D achieved the highest accuracy, followed closely by R2Plus1D and ResNet3D, while ConvLSTM had the lowest accuracy, indicating that it is not the best fit for the dataset. The results indicate that the pretrained model performs slightly better on the dataset, which might be attributed to the fact that it was trained on a large-scale dataset and have learned richer features compared to the non-pretrained models.

Overall, this project highlights the importance of exploring different model architectures and training strategies when working with video data. By comparing the performance of 4 different models, we were able to identify which models were most effective and gain insights into how their performance can be improved in future work.

Appendix

Figure 16 illustrates the decomposed filters of the R(2+1)D model at the conv1 layer. It is worth to remind that, instead of using 64 3D filters of size $3 \times 7 \times 7$ at conv1 as in R3D, R(2+1)D decomposes conv1 into 45 2D filters of size $1 \times 7 \times 7$ and 64 1D filters of size $3 \times 1 \times 1$ with non-linear ReLUs in between. This (2+1)D convolutional block has the same number of parameters as that of R3D. Figure 16(a) shows the 45 spatial filters of size 7×7 upscaled by 4x for better visualization. Figure 16(b) presents the 64 temporal filters from left to right. Each temporal filter is visualized as a 45×3 matrix. Each matrix shows how the temporal filter combines the 45 channels from the spatial filters across time (3 frames). Some interesting temporal patterns can be seen by looking at the filter weights along the time dimension

References

[1] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. *CoRR*, abs/1708.07632, 2017.

[2] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network

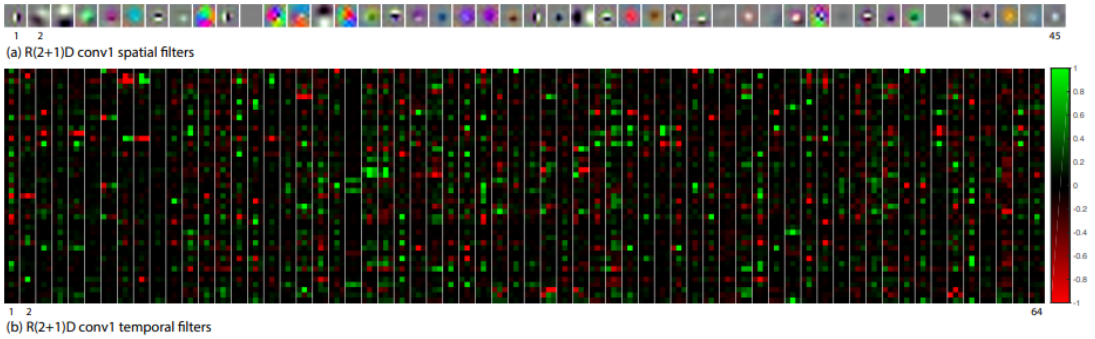

Figure 16: **R(2+1)D learned filters**. (a) The 45 spatial 2D filters learned by R(2+1)D at the conv1 layer. Each filter has size $1x7x7$ (upscaled by $4x$ for better visualization). (b) The 64 temporal 1D filters learned by R(2+1)D at the conv1 layer. Each temporal filter is visualized as a $4x3$ matrix (one column in (b)). The filter weight magnitudes are scaled from [-0.3, 0.3] to [-1, 1] for better presentation. Best viewed in color

training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[4] Preetum Nakkiran. Learning rate annealing can provably help generalization, even for convex problems. *CoRR*, abs/2005.07360, 2020.

[5] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[6] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wangchun Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *CoRR*, abs/1506.04214, 2015.

[7] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, and Manohar Paluri. A closer look at spatiotemporal convolutions for action recognition. *CoRR*, abs/1711.11248, 2017.