Incorporating Convolutional Neural Networks and Bidirectional LSTM for Action Recognition

Data Science Institute, Brown University
Yuchen Hua
Yue Wang
Tianqi Tang
Kevin Le

Introduction

The field of image classification is well-defined in deep learning literature. Researchers worldwide have adopted and improved a general architecture tailored to image data: the convolutional neural network (CNN). On the other hand, making classifications on video data is much well-defined as image classification. Classification on video data, it turns out, can draw much inspiration from image classification techniques, but with additional subtleties.

The application of image classification methods on videos is only part of the recipe to extrapolate predictions from video data. Videos are sequences in a couple of senses. For one, they're sequences of frames, as previously mentioned. Second, they're sequential in a temporal sense as well, as they are viewed within a given time frame. Considering the image and sequential properties of video data, we must adopt methods used to process images and those used for sequential data as well. Our project intends to combine CNNs and modern recurrent neural networks to solve a problem within the domain of video classification: action recognition.

Action recognition seeks to classify what actions are being performed in a video. For example, if we had a video of someone kicking a soccer ball, we'd hope for a model to classify the action as something like "kick ball". Now, do we combine computer vision techniques with sequential data algorithms to make these predictions? The overall premise is to extract features from each frame separately. Using a special time distributed layer, which gives each input image its own "convolution flow", we are able to extract features from each frame in a computationally efficient way. After extracting pertinent image features, we then pass the sequence into a recurrent neural network, which handles the temporal properties of the video. After this, we pass the sequence of convoluted images to a fully connected layer, which makes an action classification.

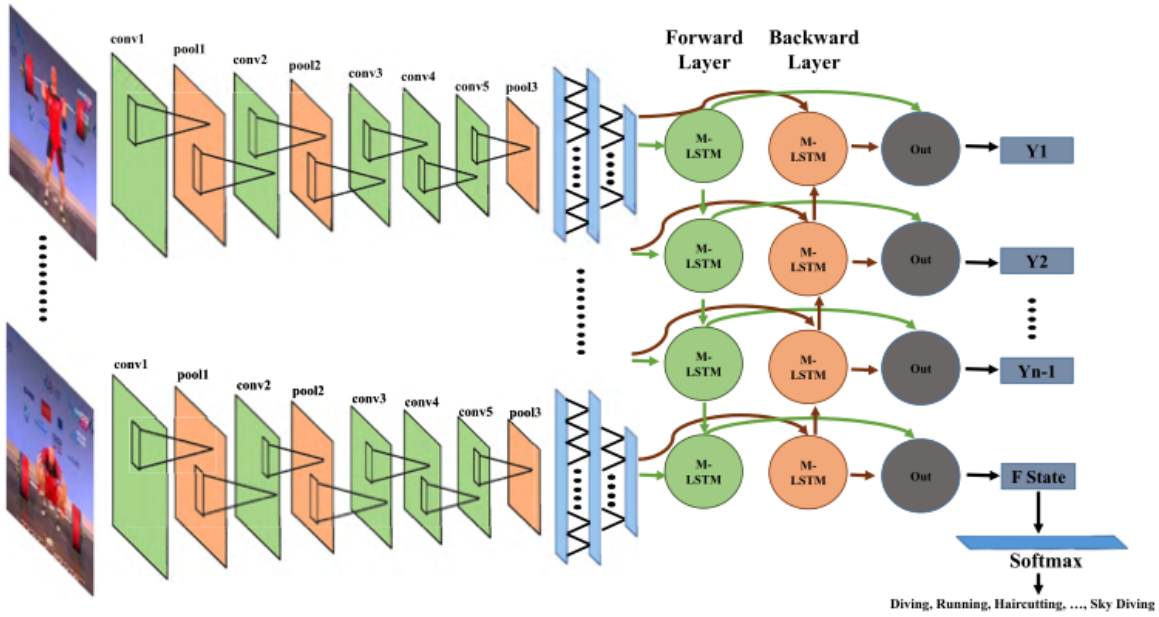
While the field of action recognition is relatively nascent, our project is inspired by earlier work done in the action recognition literature. Video datasets for this purpose are also prevalent online, and we use one especially common dataset, the HMDB dataset.

The HMDB-51 dataset, standing for Human Motion Database, is a benchmark dataset in action recognition. This dataset contains approximately seven thousand clips, scraped from various movies and YouTube, thus representing a diverse array of actions and video image qualities. The seven thousand clips represent one of 51 unique human motions, including walking, waving, clapping, and 47 others. Clips are relatively evenly distributed, with each action having little over 100 clips each. Although there are 400 clips for walking, data imbalance is not a concern, as there are thousands of other clips and 50 other categories. Many of the state-of-the-art models published in today's literature have used HMDB to test their action recognition models.

The past three years have seen an increase in activity in the action recognition literature. Numerous models for action recognition have been developed, showing comparable accuracy with one another. Despite the great diversity in start-of-the-art models, our project is largely inspired by the work of Ullah et. al (2018) from Sejong University. In their paper, they created a model that integrated AlexNet with deep bidirectional LSTM. They reported a test accuracy of 87% on the whole HMDB-51 dataset, which is one of the highest in the literature.

Proposed Framework

Similar to that of Ullah et. al (2018), our proposed framework consists of a three-step approach. First, a convolutional neural network extracts features of frames in a sequence. Using special TimeDistributed layers, we ensure the temporal relationships between each frame is maintained throughout this process. After feature extraction, we pass the output from each frame of the sequence to a recurrent neural network, specifically a bidirectional LSTM, that processes information at each time step. The results of the LSTM are then fed into a shallow MLP that then makes a decision using a softmax activation. A visual representation is shown below. Again, credit to Ullah et. al.



After surveying multiple CNN-LSTM architectures, which we discuss in a later section, we'd like to place our focus on the combination of MobileNet and bidirectional LSTM, as they are our best-performing model.

A. MobileNet

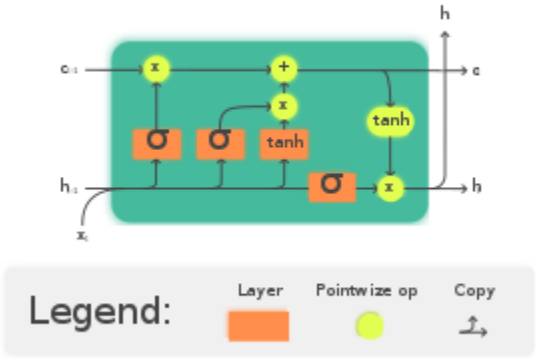
MobileNet distinguishes itself from other CNN architectures by virtue of "depthwise" convolution, which greatly speeds up training by reducing the model size and computation. In fact, depthwise separable MobileNet presented comparable accuracy to full convolutional MobileNet, while using 7x fewer parameters. It is often used to build light-weight deep neural networks for mobile and embedded vision applications, hence its name. Depthwise convolution consists of two parts, depthwise and pointwise. The depth dimension represents the number of channels. Images can have multiple channels, where each channel represents a unique interpretation. For

example, having three channels can correspond to RGB, where one channel interprets redness, another greenness, and a last one for blueness.

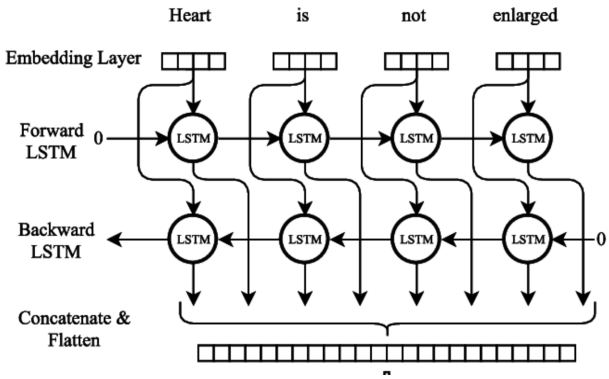
In a normal convolution layer, if we want to increase the number of channels in the output image, we can use N kernels to create N 8x8x1 images, and then stack them together to create an 8x8xN output. In depthwise convolution, each kernel iterates through 1 channel, producing an image and stacking these images together to create an output image. The distinction is that each individual channel has its own kernel. The next step is then to increase the number of channels of each image with a 1x1 kernel, which iterates through every single point in the image, hence “pointwise” convolution. This kernel has a depth equal to the number of channels and effectively merges the individual channels into one image.

There are two unique hyperparameters that can be tuned in MobileNet: width multiplier and resolution multiplier. The width multiplier controls channel depth in the model, whereas the resolution multiplier controls the resolution of the input image. In our case however, these hyperparameters are irrelevant because we adopt a pre-trained baseline MobileNet model on Tensorflow.

B. Bidirectional LSTM
LSTM units are very powerful memory cells where an LSTM cell has a forget *gate* to determine how much every previous information should be kept (and since was once called *keep gate*), and *input gate* to determine if the current candidate cell state should be added to the whole cell state of the sequence. It is designed to overcome the vanishing gradient problem so that the layer can carry information from the beginning of longer sequences to the end, compared to regular RNN units.



The purpose of a bidirectional RNN is to feed the sequence both forward and backward, so that the layer has information on this sequence in both ways. It is analogous to a speech-recognition scenario where someone hears a sentence “*I’m taking a deep … class this semester where we learn neural networks*”, the human, he/she/xe will then use information from later of the sentence, ‘*neural networks’* to complete the missing word in the sentence to be ‘*deep learning class*’, rather than a ‘*deep-diving class*’.



Bidirectional LSTM

(Modelling Radiological Language with Bidirectional Long Short-Term Memory Networks, Cornegruta et al)

We used a bidirectional wrapper from Keras to build bidirectional LSTM layers. In the context of videos, this bidirectional wrapper is useful for the same reason as explained above. For example, the model can watch a video of an athlete throwing the ball, but also catching the ball when the frames are processed in reverse.

Experimental Methods

A. Data loading and preprocessing

As previously mentioned, our data is from the HMDB dataset, which contains ~7k video files. Each file contains a video of a particular action. We used file names to split data on a ratio of 20%, where the test set contains 20% of the data. To get train and validation from the remaining 80%, we used VideoFrameGenerator from package *keras-video-generators* with a validation split rate as 0.33.

The VideoFrameGenerator is used for extracting the sequence from videos. Unlike the image recognition, which accepts a single image input, we want to extract a sequence of images, representing a video. ImageDataGenerator generates *(N, W, H, C)* data, where *N* is batch size, *W* and *H* are width and height, and *C* is number of channels which for RGB is 3, for grayscale images is 1. However, in the case of videos, there is one more dimension representing how many frames we want to produce for a datum. The shape then changes to *(N, F, W, H, C),* where F is the number of frames.

As the videos can be different lengths, to retrieve the same number of images from each data, the generators take distributed images from the entire video instead of getting the first several images, because the action may not happen in the beginning or the end of the videos.

Essentially, what VideoFrameGenerator does is read the video, read N frames and return batches. It is also compatible with ImageDataGenerator to create data augmentation. There are other possible generators from the packages, but for our project, we mainly use the VideoFrameGenerator due to the limited computing resources.

For data augmentation, we implemented zoom with range 0.1, horizontal flip, rotation with range 20, width shift with range 0.2, and height shift range with 0.2. We did not include vertical flip as the camera is always holding horizontally to film people and people's actions in the video seldom are horizontal, unlike static objects.

To build up a baseline model and improve on it gradually, we start with 4 classes data then move to 9 classes, 16 classes, 24/25 classes, and then finally use the whole dataset. Video classification takes much longer time than image classification, as each video can be treated as multiple image frames. It is worth noting that training the entire dataset with thousands of videos takes anywhere from 4-10 hours. To solve this issue and speed up training, we pick the first feature extractor pre-trained model as MobileNet as it has depth-wise separable convolutions.

B. MobileNet

We first design the architecture for training 4 classes then make changes from it. By using transfer learning, we only unfreeze the last 15 layers for training to prevent overfitting for 4 classes training. Combine the feature extractor with bidirectional LSTM layers we came up with the structure as below. The model consists of 3 bidirectional LSTM layers taking the output from the TimeDistributed layer.

```
def build_mobilenet(shape=(SIZE, SIZE, 3), nbout=nbout):
    model = keras.applications.mobilenet.MobileNet(
        include_top=False,
        input_shape=shape,
        weights='imagenet')
    trainable = 15
    for layer in model.layers[:-trainable]:
        layer.trainable = False
    for layer in model.layers[-trainable:]:
        layer.trainable = True
    output = GlobalMaxPool2D()
    return keras.Sequential([model, output])

    def action_model(shape=(5, SIZE, SIZE, 3), nbout=nbout):
    head = build_mobilenet(shape[1:])
    model = keras.Sequential()
    model.add(TimeDistributed(head, input_shape=shape))
    model.add(tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(256, return_sequences=True)))
    model.add(tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(128, return_sequences=True)))
    model.add(tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)))
    model.add(Dense(1024, kernel_initializer='he_normal', activation='relu'))
    model.add(Dropout(.5))
    model.add(Dense(nbout, activation='softmax'))
    return model
```

Figure1: MobileNet architecture using on 4 classes

Because the results from 4 classes to 9 are similar, we present the results for 9 classes. Using a pre-trained MobileNet model delivered the best training results, regardless of the number of classes we train on. The training result for 9 classes shown below.

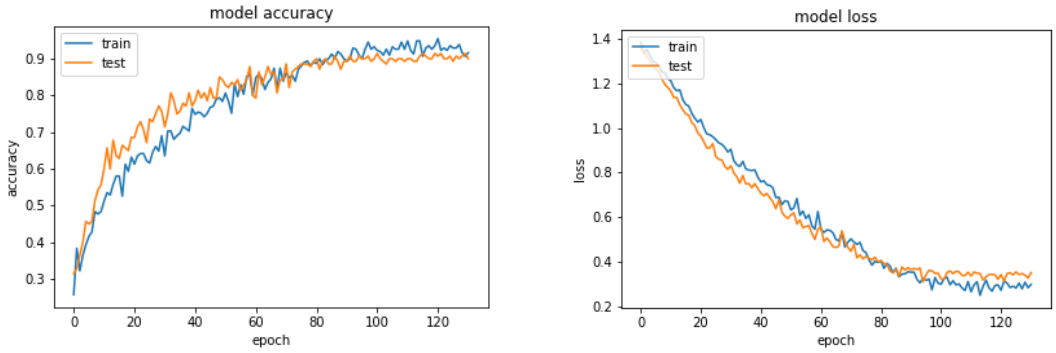


Figure 2: training process for 9 classes MobileNet

To fit the whole dataset with 51 classes, we unfreeze all the layers in MobileNet to increase model complexity and predicting power. Also we have reduced bidirectional LSTM layers to two to overcome the overfitting issues. Also, we include l1 and l2 regularization with rate from 0.0001 to 0.01 and find the best one is l2 = 0.001. The training results for one run shows as below:

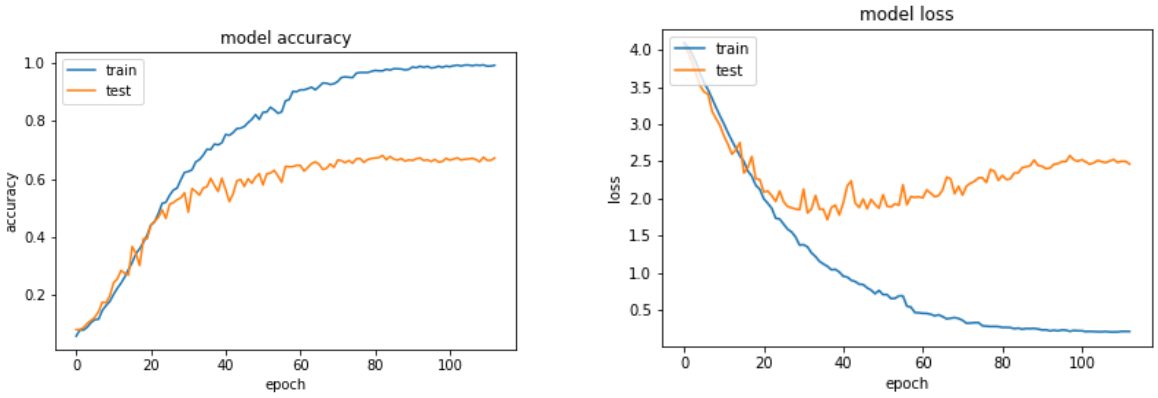


Figure 3: training on 51 classes, early stopping = 30

We have implemented early stopping with patience to 30 although from the loss curve, we may want to stop it earlier. After changing the patience to 10, it performs better as shown below.

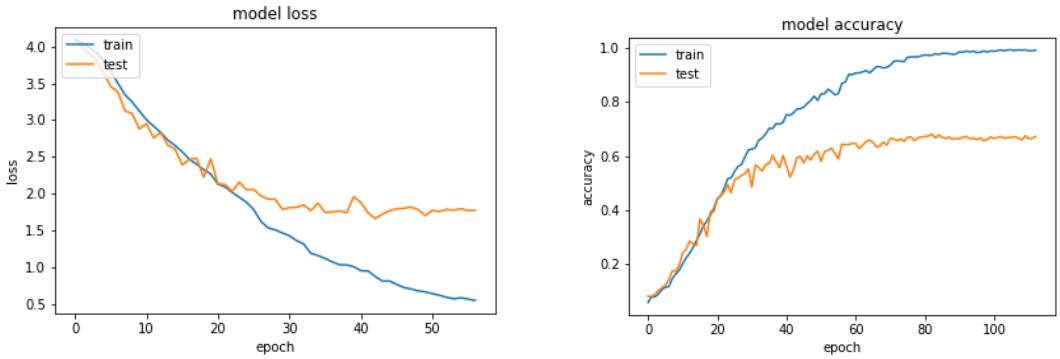


Figure 4: training on 51 classes, early stopping = 10

Although the model generally still has the overfitting issue. One possible reason is that we are extracting only 5 frames from the video due to limited computing resources. Video classification takes longer time but our time is limited. So in the future work, we will increase the number of frames or try to use other methods to generate video frames to improve the model performance. To be mentioned, we have also tried the other framework to generate the videos using "keras_video.SlidingFrameGenerator" but it takes much longer time than the VideoFrameGenerator.

We plan to overcome the overfitting issue with ensemble methods with current available models we have trained.

C. AlexNet

We also tried incorporating AlexNet with our bidirectional LSTM. While this replicates the work of Ullah et. al (2018), this is also a necessary comparison with our other CNN architectures. Unlike for MobileNet and DenseNet, Tensorflow does not provide pre-trained weights for AlexNet, so we implemented it manually and trained the layers from scratch. The overall structure of our AlexNet head is shown below.

```
def build_Alexnet(shape=(SIZE, SIZE, 3), nbout = nbout):
    model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu'
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu'
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu'
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu'
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    ])
    return model
```

AlexNet architecture hosted with ♥ by GitHub view raw

Figure 5: AlexNet architecture

We utilize the same action model as the other CNN models. The difference between Ullah's work and our work is that we include three bidirectional LSTMs, whereas they only include two.

The AlexNet-LSTM model shows great promise when training with 4 classes. Below we see that validation loss and training loss are rather stable. Although accuracy and loss are rather unstable, there is little to no overfitting, and the validation accuracy and loss values show reasonable values, as shown in Figure 6.

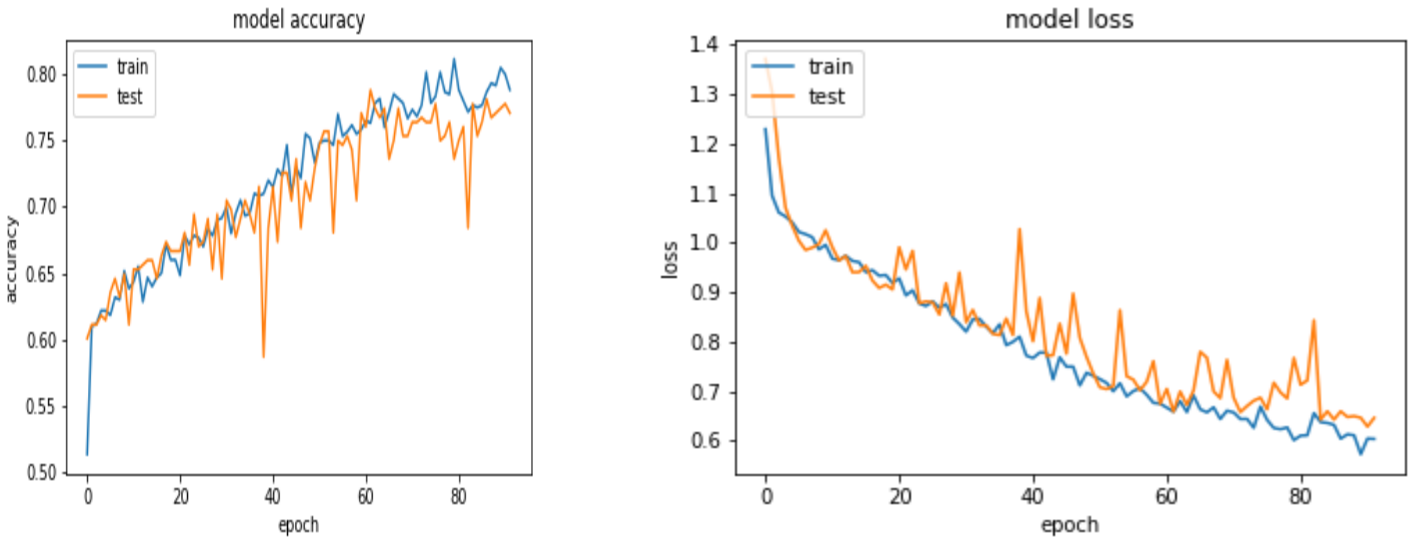


Figure 6: Training AlexNet on 4 classes

However, as we increase the number of classes, the model severely overfits the data. When training AlexNet-LSTM on 51 classes, training is relatively stable until the 60th epoch, where training accuracy continues to increase to 0.99, whereas the validation accuracy converges to 0.55. This overfitting could be due to the fact that we're training all layers from scratch. The convolutional layers could have perhaps overanalyzed the features of each image, reducing the generalizability of the model in validation.

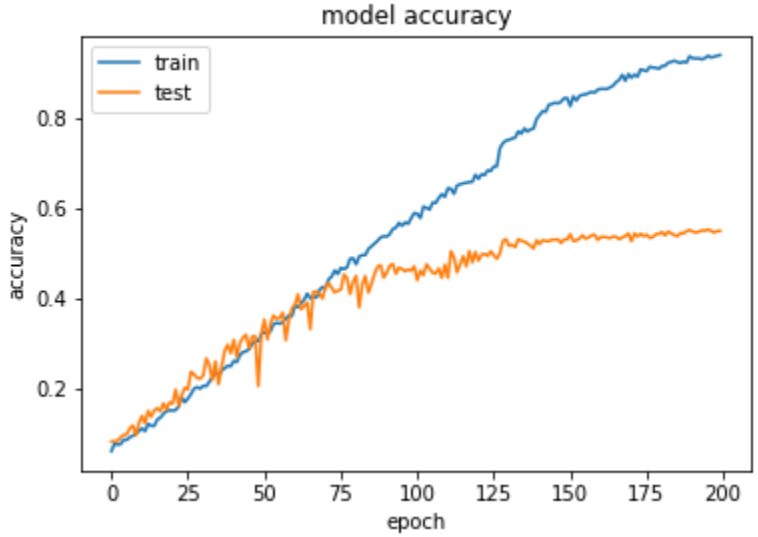


Figure 7: AlexNet on 51 classes

D. DenseNet

We also adopted densely connected convolutional networks, or DenseNet for short, as our base model. DenseNet consists of a series of dense blocks with convolution and pooling layers in between.

Each dense block consists of several layers. Each layer consists of a 1x1 and a 3x3 convolution layer. The 1x1 convolution layers enable transitioning of outputs directly to the latter layers. Because the latter layers receive outputs directly from the previous layers, the gradients in the backprop procedures can propagate more easily. Therefore, DenseNet is robust against vanishing/exploding gradient problems. Furthermore, because the latter layers contain raw outputs from all its previous layers in a block, the features in the latter layers are diverse.

Our densenet model adopted pretrained weights from imagenet, whilst excluding the top layers. The base model is entirely trainable. It is then followed by a global max pooling layer, a time distributed layer, 2 bidirectional LSTM layers, a dense layer, a dropout layer and the output layer. The architecture shows as below,

```
def build_densenet(shape=(SIZE, SIZE, 3), nbout=nbout):
    model = tf.keras.applications.DenseNet121(weights='imagenet', input_shape = shape, include_top=False)
    model.trainable = True
    output = GlobalMaxPool2D()
    return keras.Sequential([model, output])
nbout = len(classes)
def action_model(shape=(5, SIZE, SIZE, 3), nbout=nbout):
    head = build_densenet(shape[1:])
    model = keras.Sequential()
    model.add(TimeDistributed(head, input_shape=shape))
    model.add(tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(256, return_sequences=True)))
    model.add(tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(128)))
    model.add(Dense(1024, kernel_initializer='he_normal', activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.001)))
    model.add(Dropout(.5))
    model.add(Dense(nbout, activation='softmax'))
    return model
```

Figure 8: DenseNet model structure.

Using the above architecture on the DenseNet model, we achieved around 70% accuracy on the validation set for 51 classes after 4 hours of training. However, it suffers from overfitting severely, like the other models. A possible reason could be that there is not enough validation data, as each category only has 100 files in total and for validation is around 30 files. To improve this, we came up with the ensemble method and combined it with our MobileNet model.

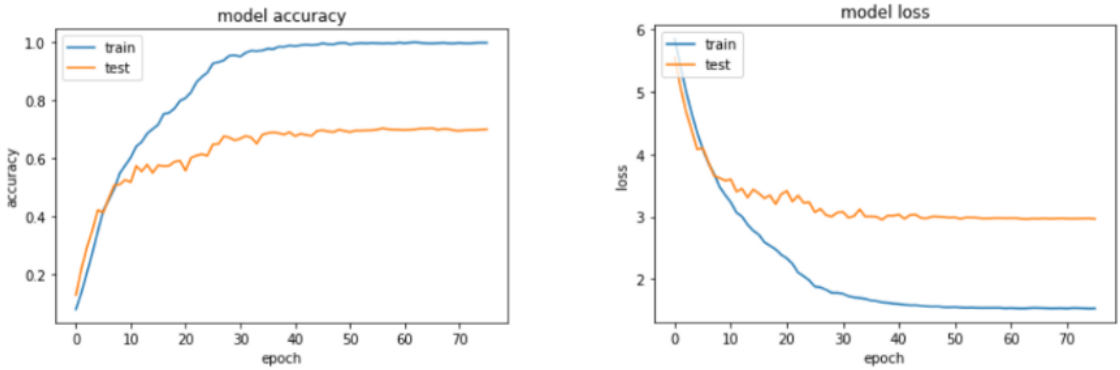


Figure 9: DenseNet results.

E. Ensemble

To benefit from the independently-trained, various CNN architectures mentioned above as heads, trained models with MobileNet, DenseNet and AlexNet were ensembled with an average layer for them to vote on the final prediction. Ensemble modeling has been used in different areas such as facial recognition, financial decision making, medical diagnosis to overcome the overfitting on single models and reduce the generalization error, as well as increase the robustness to different input data with the variety of sub-models.

As shown in the visualizations of 10 instances from the test set below, correct classifications of a single model remained to be correct, while there is one sequence that was misclassified by the MobileNet model. The model predicted ‘clap’ instead of the ground truth label ‘wave’. This error was corrected when ensembling it with our DenseNet-head model.

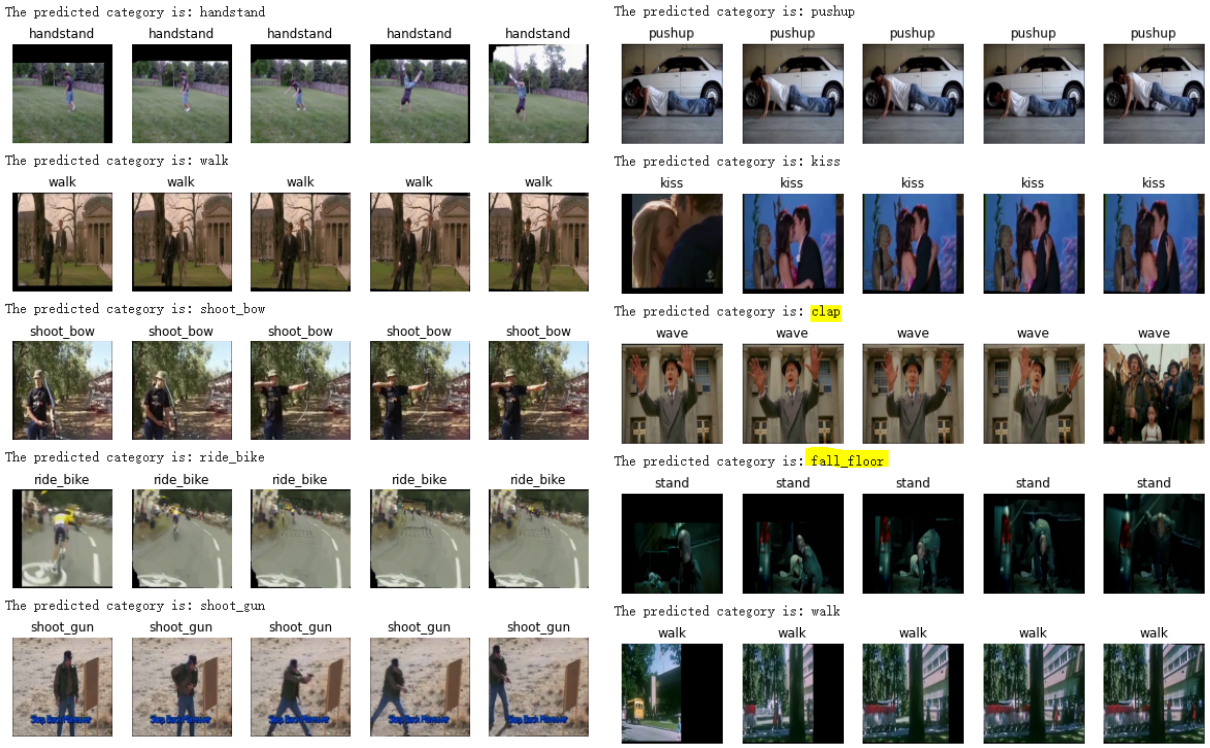


Figure 10: Predictions without ensembling

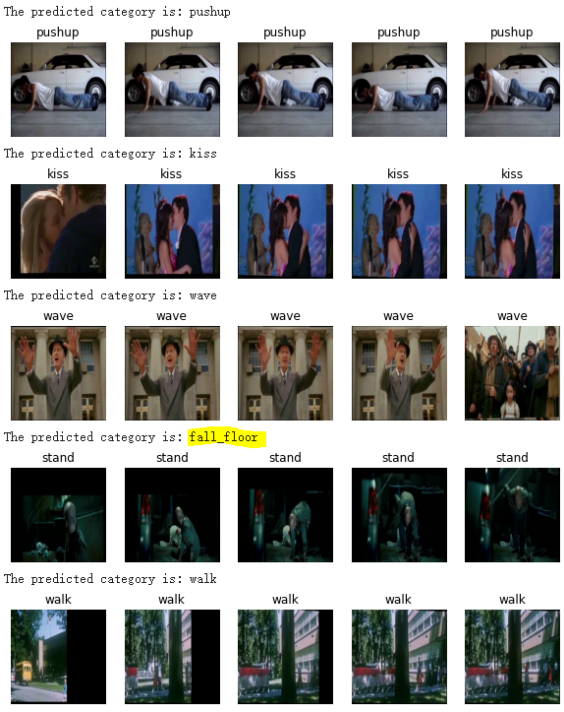


Figure 11: Predictions after ensembling

The ensembled model was evaluated with our test set. The accuracy increased more than 4% compared to single models and ended up at 74% on the test dataset. This accuracy would be boosted further if we have other model structures such as 3DCNN or BERT because of their

different structure compared to all three of our CNN-LSTM structures, and generalization error will be further reduced.

Conclusion

A. Summary

We have achieved reasonable results from fewer classes such as 4, 9, and 16. Even for 25 classes, MobileNet achieved 85% validation accuracy with slightly overfitting issues. For 51 classes, MobileNet and DenseNet with 2 layers of bidirectional LSTM both achieved around 70% validation accuracy which is not bad with our computing resources and limited time. We have tried using different numbers of fully-connected layers in our model with different combinations of regularization for kernel, bias and activation. Although a strong regularization causes underfitting problems, so finally we have used l1/l2 with a regularization rate as 0.001. Also we have tried freezing a few layers to compare with the fully unfreezed MobileNet model, although it does not perform better than unfreeze all.

We used MobileNet and DenseNet models to do ensemble and generalize the errors. Both models individually have validation accuracy lower than 70%, however, with the ensemble method, we have got a model with 72% test score, which means ensemble method boosting out results. So, in the future, we may consider training more models and doing the ensemble method to improve the performance and address the overfitting issues.

B. Challenges

The most prominent challenge in our project was training time. One model takes anywhere from 6–10 hours to train in completion, and that's without stumbling across RAM overflow issues on the way. Speaking of RAM, there were many instances where our RAM would overflow during the first epoch of training. This would happen most often as we increased the number of action classes in our data. Luckily, we prepared a useful function for saving and pickling our models, which helped alleviate many of the RAM issues we experienced.

We believe these challenges stem from the fact that we're uploading the videos directly onto a shared Google Drive. There are other data storage methods that store and upload videos more efficiently, such as TFRecord. We'd like to use TFRecord in the future, as we can create structured TFDatasets from TFRecord. Utilizing TFRecord also greatly clarifies our image frame preprocessing procedure, which was something we experienced uncertainty with.

In addition, some of our models were overfitting, namely our AlexNet + LSTM and DenseNet + LSTM models. Considering the long training time, we'd like to do more hyperparameter tuning given more time.

C.Future improvements

There are a variety of possible improvements for our project, from increasing the size of the training data to changing to a completely different model.

Due to time and memory constraints, we are only able to handle 5 frames per video clip, which may be insufficient to make very good predictions. This is because the transitions between consecutive frames are too large, and layers made with LSTM cells are capable of memorizing longer sequences of data, and it was not taken advantage of by using only 5 frames per video. If time permits, we may consider more frames for a smoother transition of temporal information. We may also try different video generators to generate more training samples. One alternative generator we have tried is SlidingFrameGenerator. It uses a window method to extract more training data from the video than VideoFrameGenerator which we mainly focused on, but it also requires a lot more computing power

Our group has been heavily utilizing LSTM for Action Recognition (AR), which performs fairly well. However, it does have some shortcomings. Because the model has to be trained sequentially, parallel computing does not work. Even though our LSTM is bidirectional, the forward and backward training were not taken care of simultaneously. The lack of communication between forward and backward training creates inconsistencies for prediction. Furthermore, because LSTM is RNN based, the training can only be processed sequentially, meaning that it does not benefit from parallel computing.

For these disadvantages, our group also attempted to create a new model based on Bidirectional Encoder Representations from Transformers (BERT). Because BERT trains on the context of words, it is able to simultaneously consider contextual information in both directions. Also, because BERT is based on transformers, which do not need to train the data sequentially, it is able to utilize parallel computing, significantly improving the training speed.

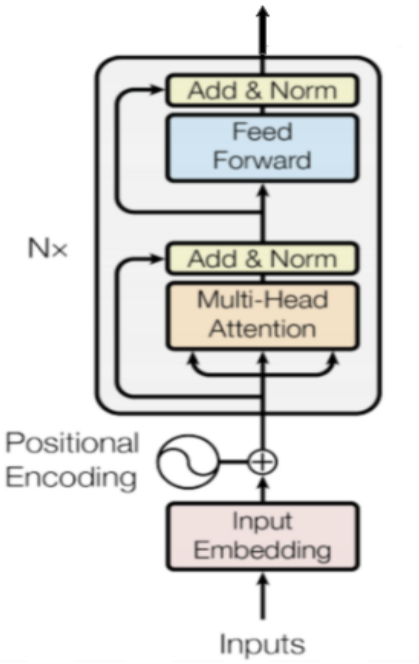


Figure 7: Transformer diagram

Kalfaoglu et. al. proposed an AR algorithm using BERT. It utilizes 3D CNN for feature extractions and the Late-fusion method to combine the information from different frames.

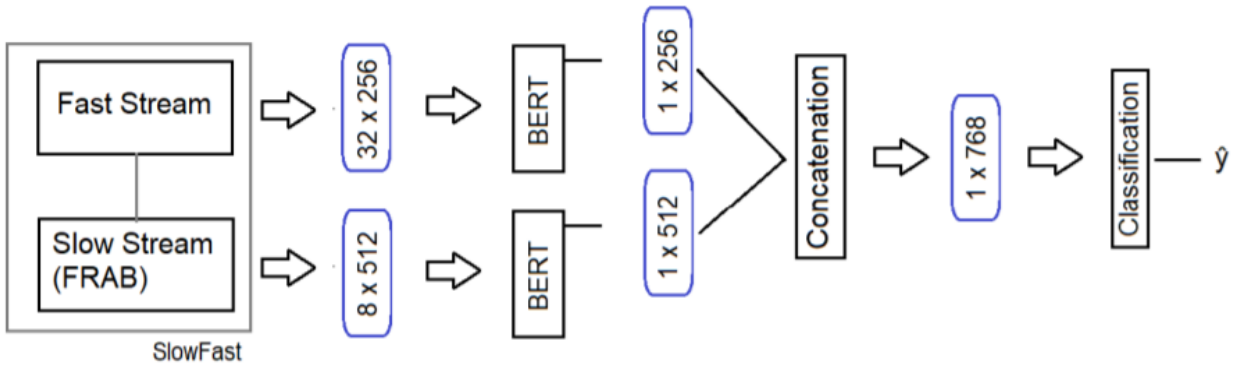


Figure 8: Late-fusion diagram.

The Late-fusion method uses two different BERT models corresponding to fast stream and slow stream, or a SlowFast architecture. The slow stream has a better spatial capability whereas the fast stream has a better temporal capability. Each BERT model hence specializes in either spatial or temporal information. Their outputs are then concatenated and the result is then used for determining the final label. We hope to pursue these future directions to further improve the performance of action recognition models.

These different models will also boost the final accuracy with ensembling because of the difference between them and our CNN-LSTM structure.

References

[1] Ullah, A., Ahmad, J., Muhammad, K., Sajjad, M., & Baik, S. W. (2017). Action recognition in video sequences using deep bi-directional LSTM with CNN features. *IEEE access*, *6*, 1155–1166.

[2] Xin, M., Zhang, H., Wang, H., Sun, M., & Yuan, D. (2016). Arch: Adaptive recurrent-convolutional hybrid networks for long-term action recognition. *Neurocomputing*, *178*, 87–102.

[3] Wang, P., Cao, Y., Shen, C., Liu, L., & Shen, H. T. (2016). Temporal pyramid pooling-based convolutional neural network for action recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, *27*(12), 2613–2622.

[4] Kalfaoglu, M. Esat, et al. “Late Temporal Modeling in 3D CNN Architectures with BERT for Action Recognition.” *Computer Vision — ECCV 2020 Workshops*, 2020, pp. 731–747., doi:10.1007/978–3–030–68238–5_48.

[5] “Dive into Deep Learning¶.” *Dive into Deep Learning — Dive into Deep Learning 0.16.2 Documentation*, d2l.ai/.

[6] Huang, G., Liu, Z., der Maaten, L., Weinberger, K. Q. (2018). Densely Connected Convolutional Networks. *CVPR.*

[7] Tsang, S., Review: DenseNet — Dense Convolutional Network (Image Classification), https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803