INFOMCV Assignment 5 Report (Group 34)

Gianmarco Picarella

I. ARCHITECTURE CHOICES

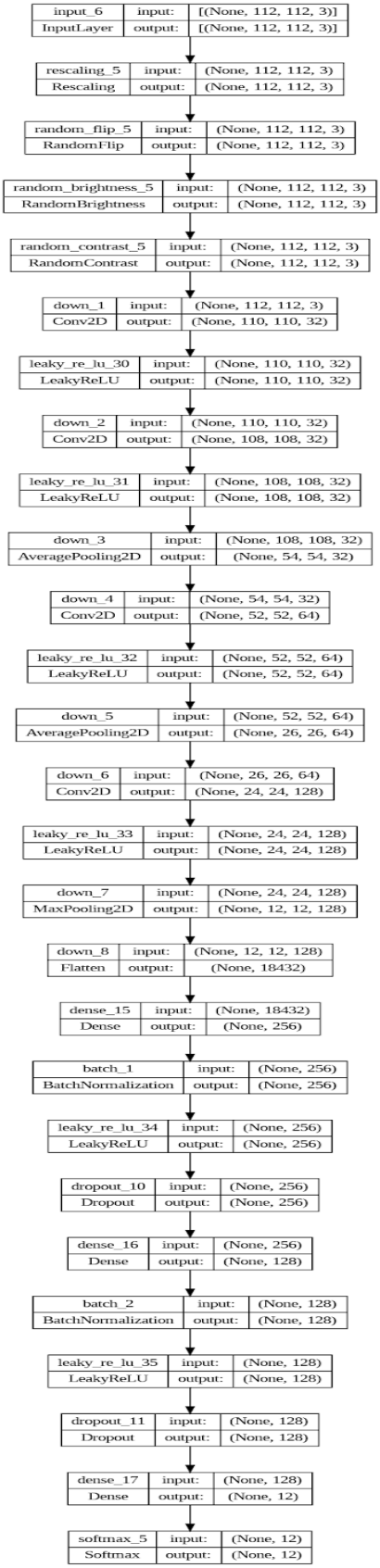
Every model employs a Rescaling layer (normalization) with factor 1/255 placed right after the input layer.

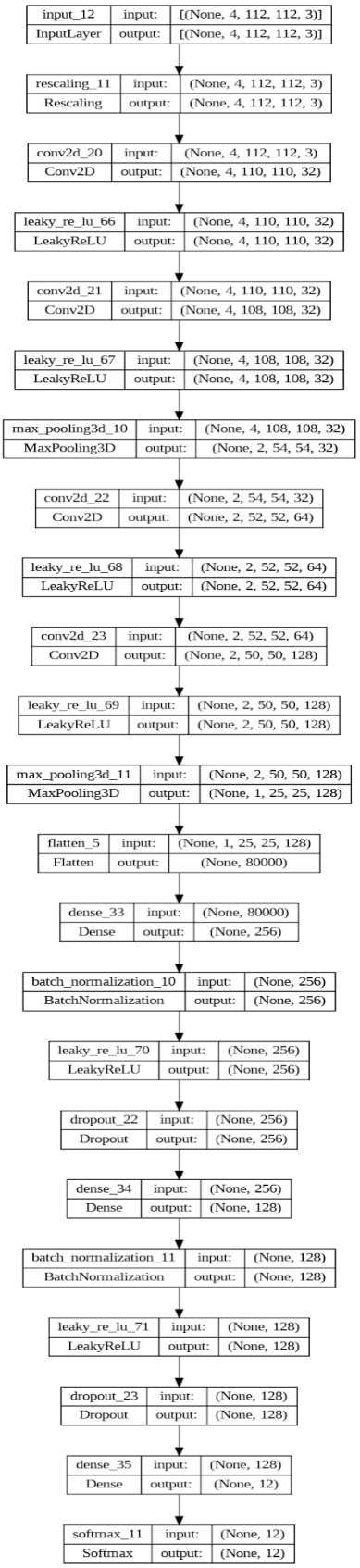
Frame CNN architecture: our architecture takes an RGB frame input with shape (112, 112, 3). The model applies a series of convolution layers with an increasing number of 3x3 kernels: the first 2 layers use 32 kernels, while the third and fourth layers use 64 and 128 kernels respectively. Then, we apply two average pooling and one last max pooling before flattening our features; this setup shows improvements when compared to other architectures we tried. After the flatten operation, we have 2 fully connected layers (with 256 and 128 neurons each) and one last fully connected output layer. For each layer, we use batch normalization and dropout (with dropout rate set to 0.5). Finally, we apply a softmax activation function to our last dense layer, so that our resulting vector contains the probability that our current sample is from a specific class. For every convolution and dense layer (except for the last one), we use LeakyReLU as activation function. The reason why we decided to use LeakyReLU is that it brings to the network all the improvements obtained with ReLU, while also solving some of the drawbacks that arise with it. In particular, large weight updates in the network may result in a summed input passed to the activation function below zero, producing an output value of 0. This behavior may result in large dead areas of the network where neurons do not add any contribution to the classification because their output is always 0 (also called dying ReLU state). LeakyReLU addresses this issue by allowing small negative values when the summed input is less than zero. We also have three data augmentation layers which will be discussed in the Choice Tasks section.

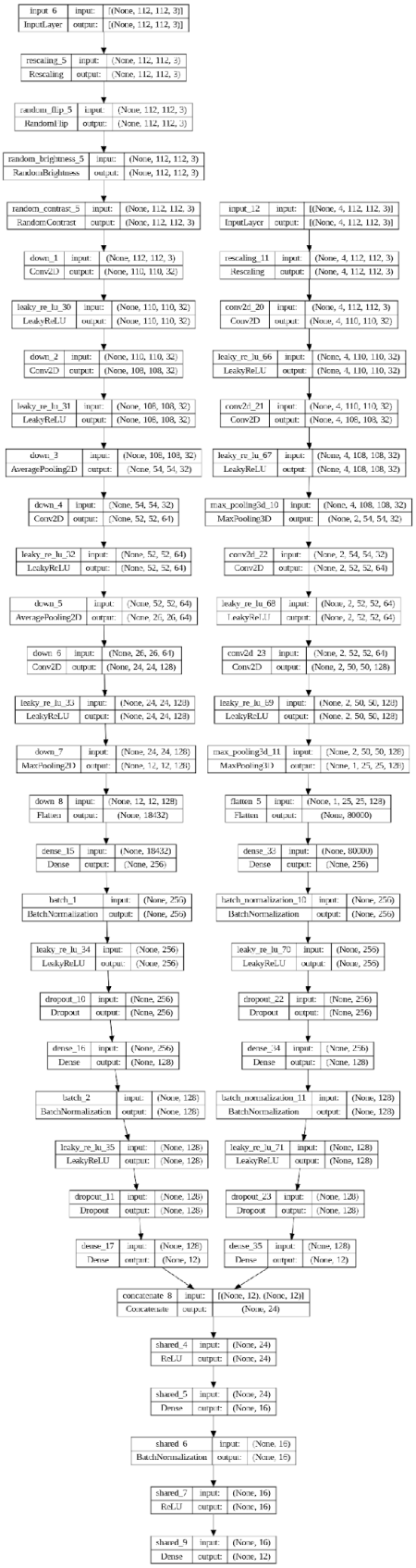
Optical flow CNN architecture: our architecture takes 4 RGB optical flow frames as input with an overall shape (4, 112, 112, 3). Each optical flow frame is captured around the video's mid frame: specifically, half the frames are taken before the mid frame and the remaining ones after the mid frame. We calculate a dense optical flow with the function cv2.calcOpticalFlowFarneback(…) and then compute the magnitudes and angles for each flow point with cv2.cartToPolar(…). Then, we transform our magnitudes and angles in an HSV image, where the hue represents the angle and the value represents the magnitude. We then convert the HSV frame to RGB and store our optical flow frame in the results stack. Our architecture for the optical flow model is similar to the one used for the first two tasks. We have 2 convolution layers with 32 3x3 kernels each, an average pooling which is 3D this time (the stack of frames is reduced also along its depth). Then again, we have 2 convolution layers with 64 and 128 3x3 kernels respectively and a fine Max pooling 3D operation. After the flatten operation, the network is the same as our first model. The idea behind this choice is that after the flatten operation we have abstracted the type of input and extracted only the set of features needed to classify our data sample. Again, for each convolution and dense layer (apart from the last dense output layer) we use LeakyReLU as our activation function. The reasons why we use LeakyReLU in this model are the same as explained in the previous model.

Combination into two-stream network: our two-stream network combines Model 2 (trained with transfer learning and fine tuning on the HMDB51 dataset) and Model 3. For each of these models, we skip the Softmax layer and use the concatenate layer as a merge operation to fuse the results from Model 2 and Model 3. Each model outputs a 12x1 vector that is fused in a 24x1 vector on which then a LeakyReLU activation function is applied. Because of this late merge, our model's architecture can be

classified as a late fusion architecture. We noticed massive overfitting while training our model. We tried using Dropout and Batch normalization, but in the end reducing the number of neurons in the dense layer right after the merging layer was the only change that reduced overfitting by some noticeable extent. Our shared architecture comprises two Dense layers followed by a Batch normalization operation (only for the first dense layer). Also here, we use LeakyReLU as our activation function (apart from our output layer which uses a Softmax activation).







II. DATASET MANIPULATION

Model 1 (Stanford40): For dataset Stanford40 (Model 1), we load each picture using *tf.keras.utils.load_img(...)*, transform the image from PIL to a numpy array with *tf.keras.utils.img_to_array(...)* and then resize the image to our target input shape of 112x112x3 with *resize_image(...)*. The last mentioned function resizes the image while maintaining the aspect ratio and centers the result in a 112x112 frame. All the remaining pixels are then set to black. For what concerns the classes, each label is converted into an integer-based format ranging from 0 to 11. Given a string label, its integer format is computed by finding the index at which the string is stored in a reference array (in our case, STAN40_CLASSES).

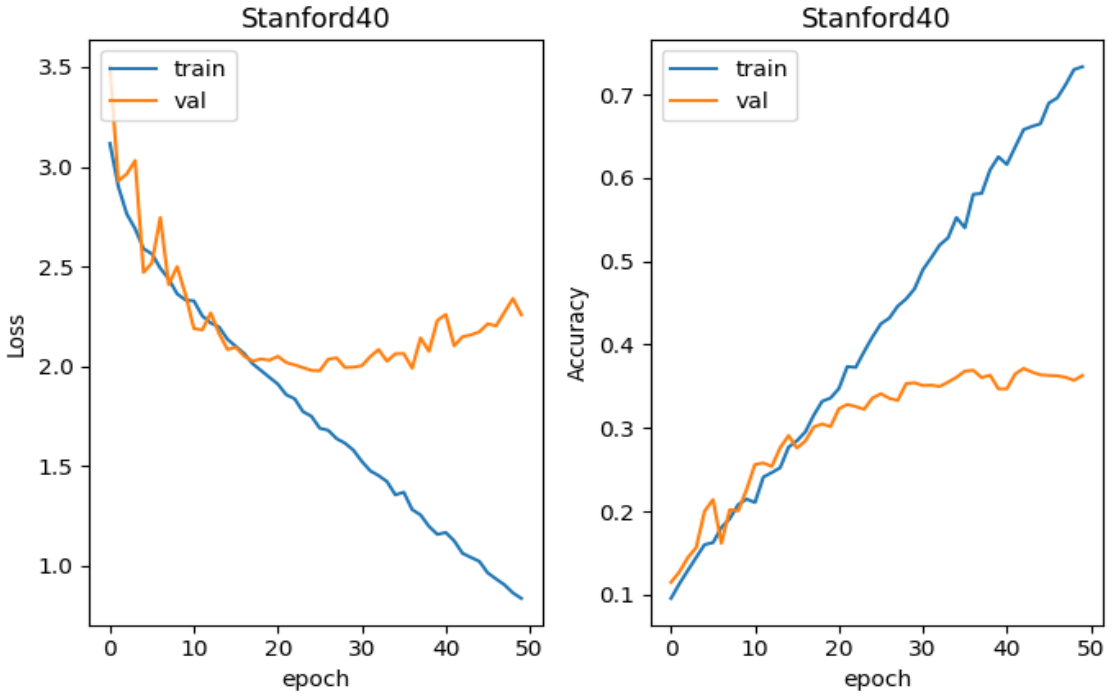
Model 2 (Hmdb51TL): For dataset HMDB51 (Model 2), the process is very similar to Stanford40: we load the mid frame picture for each video, convert it to RGB (because openCV uses the BGR format), resize the image to the target 112x112x3 shape with *resize_image(...)* and store the result. Again, each label is converted into an integer-based format ranging from 0 to 11. Given a string label, its integer format is computed by finding the index at which the string is stored in a reference array (in our case, HMDB51_CLASSES).

Model 3 (Hmdb51OF): For dataset HMDB51 (Model 3), we created a function called *compute_optical_flow(...)* which takes a video, the starting index used to collect samples and the number of optical flow frames to return. For each video in the dataset, we compute a stack of optical flow frames with shape 4x112x112x3 by selecting 5 subsequent frames with frame 3 corresponding to the mid frame of the video. We compute the dense optical flow (i.e. the optical flow is defined for each image pixel) with *cv2.calcOpticalFlowFarneback(...),* then convert the magnitudes and angles of the flow into an HSV representation where H = angle, S = 255 and V = magnitude. Finally, we convert this HSV image back to RGB, resize the image to our target shape 112x112x3 and then store the result into our stack of frames. Again, each label is converted into an integer-based format ranging from 0 to 11. Given a string label, its integer format is computed by finding the index at which the string is stored in a reference array (in our case, HMDB51_CLASSES).

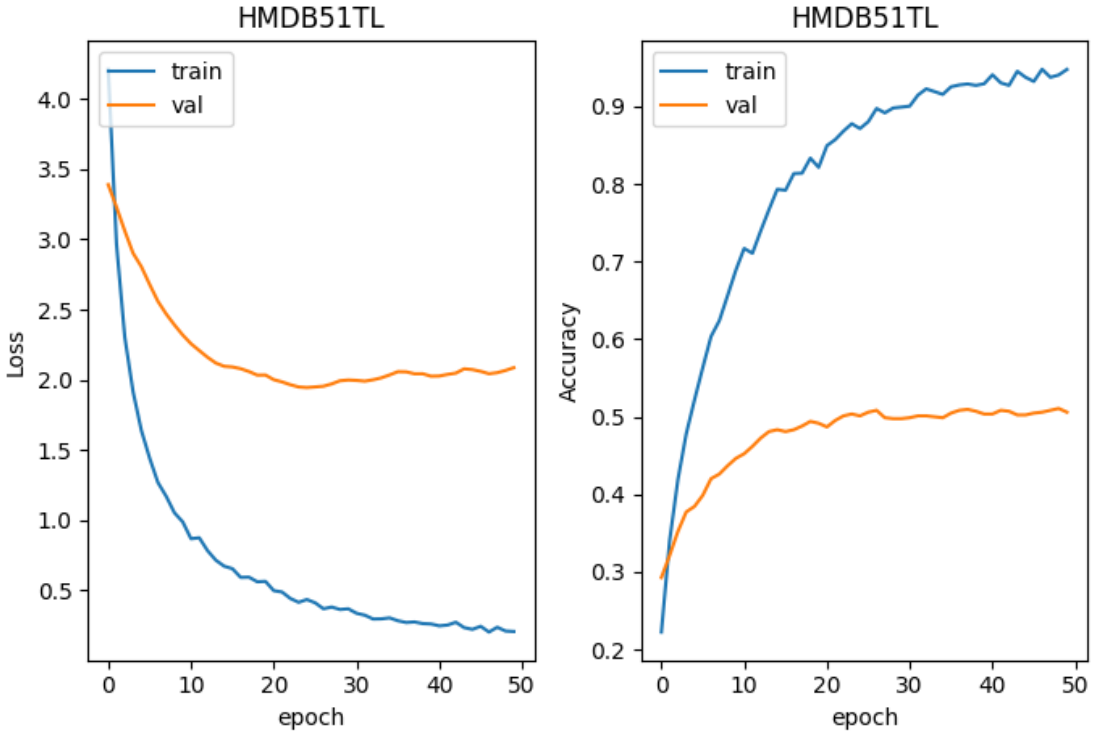
Model 4 (Hmdb51MXD): For dataset HMDB51 (Model 4), we merge the logic used for Models 2 and 3. For each mid frame, the corresponding optical flow stack is computed. The resulting input arrays have shapes *numSamplesx4x112x112x3* (optical flow stacks) and *numSamplesx112x112x3* (mid frames). Again, each label is converted into an integer-based format ranging from 0 to 11. Given a string label, its integer format is computed by finding the index at which the string is stored in a reference array (in our case, HMDB51_CLASSES).

III. Results

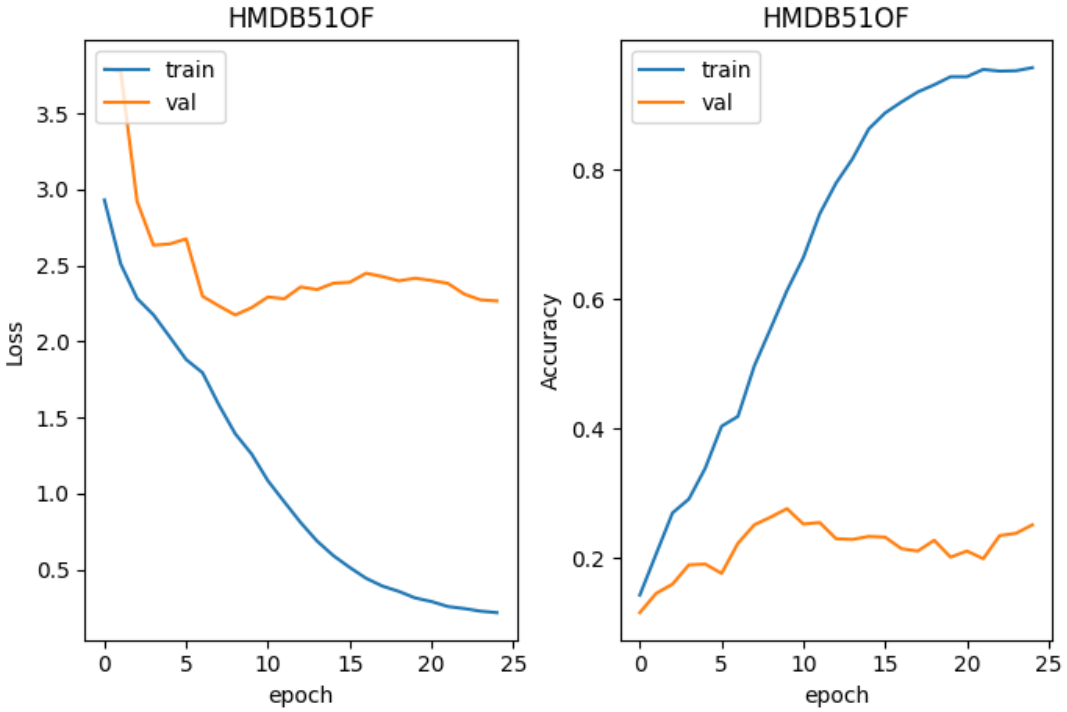
A. Stanford 40 Frames



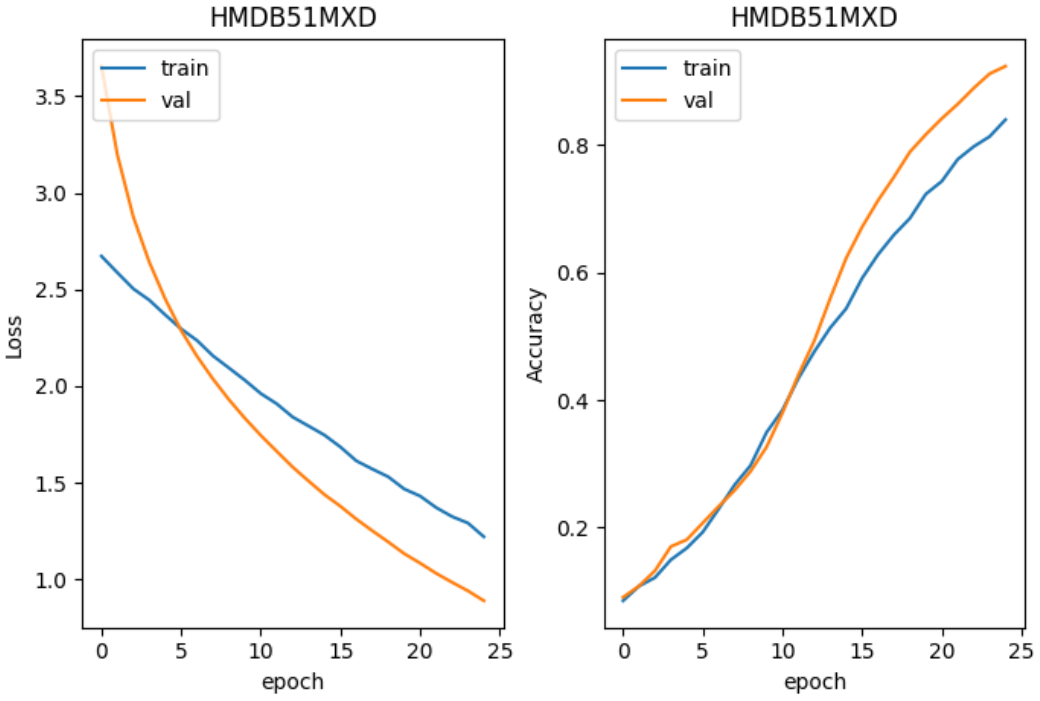
B. HMDB51 Frames



C. HMDB51 Optical flow



D. HMDB51 Two-stream



E. Result table

SUMMARY OF OUR RESULTS PER MODEL

Dataset	Model specifications				
	Model	Top-1 accuracy (train / valid)	Top-1 loss train	Model parameters	Test accuracy
Stanford 40	Stanford40 (Frames)	0,729 / 0,406	0,7993	4.857.324	0,345
HMDB51	HMDB51TL (Frames)	0,916 / 0,565	0,2	4.857.324	0,319
HMDB51	HMDB51OF (Optical Flow)	0,95 / 0,33	0,192	20.618.732	0,261
HMDB51	HMDB51MXD (Two-Streams)	0,824 / 0,916	1,228	25.476.724	0,272

III. DISCUSSION OF RESULTS

Every model was trained with the Adam optimizer and a learning rate set to 0.001. We use "categorical_crossentropy" as loss function for all the trainings.

Stanford40 (Model 1) is trained with 50 epochs and a batch size set to 128. The model achieves 0.345 accuracy on the testing set. As we can see in the loss graph, our model is overfitting the dataset and the validation loss starts increasing after epoch 25. At the same time, we notice that the training and validation accuracies have a huge gap, which is another confirmation that our model is overfitting the dataset. As a possible improvement, we could surely use an early stopping procedure that checks whether the validation loss is rising and in that case stop the training.
For what concerns overfitting, we reduced the gap between accuracy and validation accuracy by almost 0.4 with data augmentation, i.e. by increasing the number of samples in the training set. We also reduced the number of kernels for each convolution layer and decreased the number of neurons in each dense layer. We used Dropout and BatchNormalization to partially mitigate overfitting and increase the model robustness over the testing set.

HMDB51TL (Model 2) is trained with 50 epochs and a batch size set to 128. The model achieves a 0.319 accuracy on the testing set. Also in this case, it can be easily noticed that our model is overfitting the training set and from epoch 25 the validation loss starts rising. Our transfer learning approach is quite straightforward: we load the Stanford40 model (Model 1), freeze all the feature extraction layers (i.e. every layer from the input till the flatten operation), run a first training with the new dataset, unfreeze all the layers, run a fine-tuning training with a learning rate set to 1e-6.
For what concerns the validation accuracy, we have our average value around 0.5. We were expecting a testing accuracy somewhat aligned with the validation accuracy that we get during training, but unfortunately - as previously stated - our testing accuracy is way lower (around 0.32). We checked the distribution of each class in the training, validation and testing set and they seem to be balanced. We supposed that the testing set is more difficult than the training set, but then after trying to swap training and testing sets we still got this discrepancy between the two values. We debugged our dataset loading logic and there seem to be no issues, so we suppose that, for what concerns plain RGB images, the training set shares some easier features to spot that the testing set doesn't have, thus making the validation score much higher than the actual testing score.

HMDB51OF (Model 3) is trained with 25 epochs and a batch size set to 128. The model achieves a 0.261 accuracy on the testing set. As we can see in the loss graph our model is overfitting the dataset and the validation loss starts increasing after epoch 8. The accuracy graph also shows a decrease for the validation accuracy after epoch 8. The overfitting in this case is much more noticeable in both graphs. We tried to reduce overfitting with Dropout and BatchNormalization. Both methods helped us achieve 0.07% more testing accuracy.

HMDB51OF (Model 4) is trained with 25 epochs and a batch size set to 128. The model achieves a 0.272 accuracy on the testing set. Even here, we notice the same validation score issue we have with Model 2. We set Model 2 and Model 3 branches to non-trainable. At some point, we can notice that tje validation accuracy is greater than the training accuracy: this is actually because of both Dropout and the fact that at test time all the features are used, thus leading to more robust predictions.
Our two-stream model is actually pretty simple and it does not involve early fusion layers. By using different types of fusion operations in more areas of the networks, we could have surely got better results. Also, we managed to improve our testing accuracy up to almost 31% by experimenting with different fusion operations for the network. A detailed description can be found in the Choice Tasks section.

Among all, we select Model 2 as the most robust model for action classification on the HMDB51 dataset. The main advantages (apart from its testing accuracy) include more contained overfitting (when compared with Model 4), which makes it surely more accurate on new unseen samples. We can still get similar accuracies with different merge operation for our two stream network (Add operation) but the very high overfitting makes the model less reliable on new data.

IV. Link to Model Weights

*Our models can be retrieved **here***

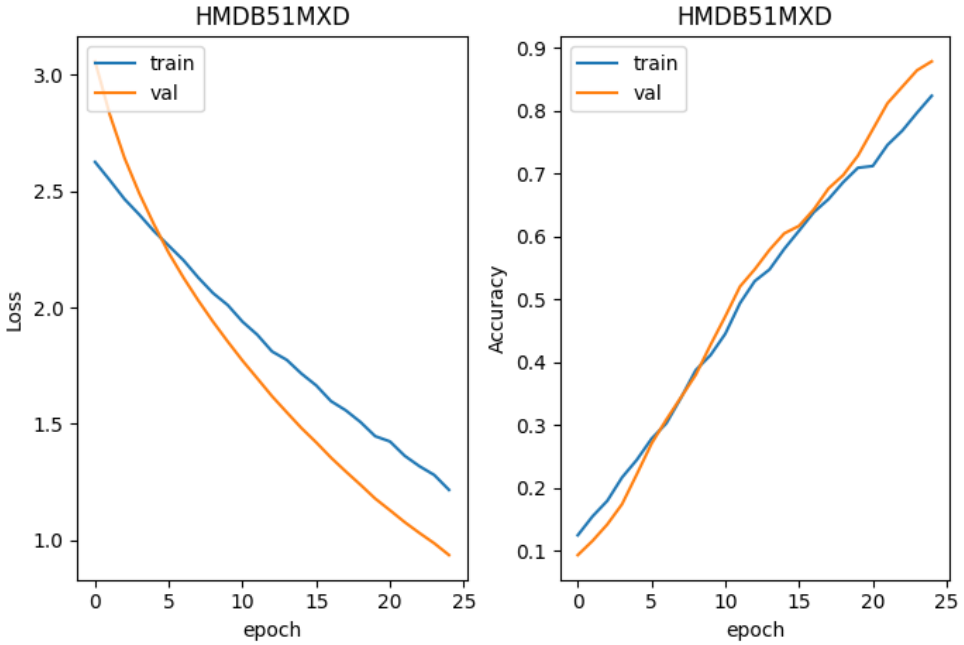
V. Choice Tasks

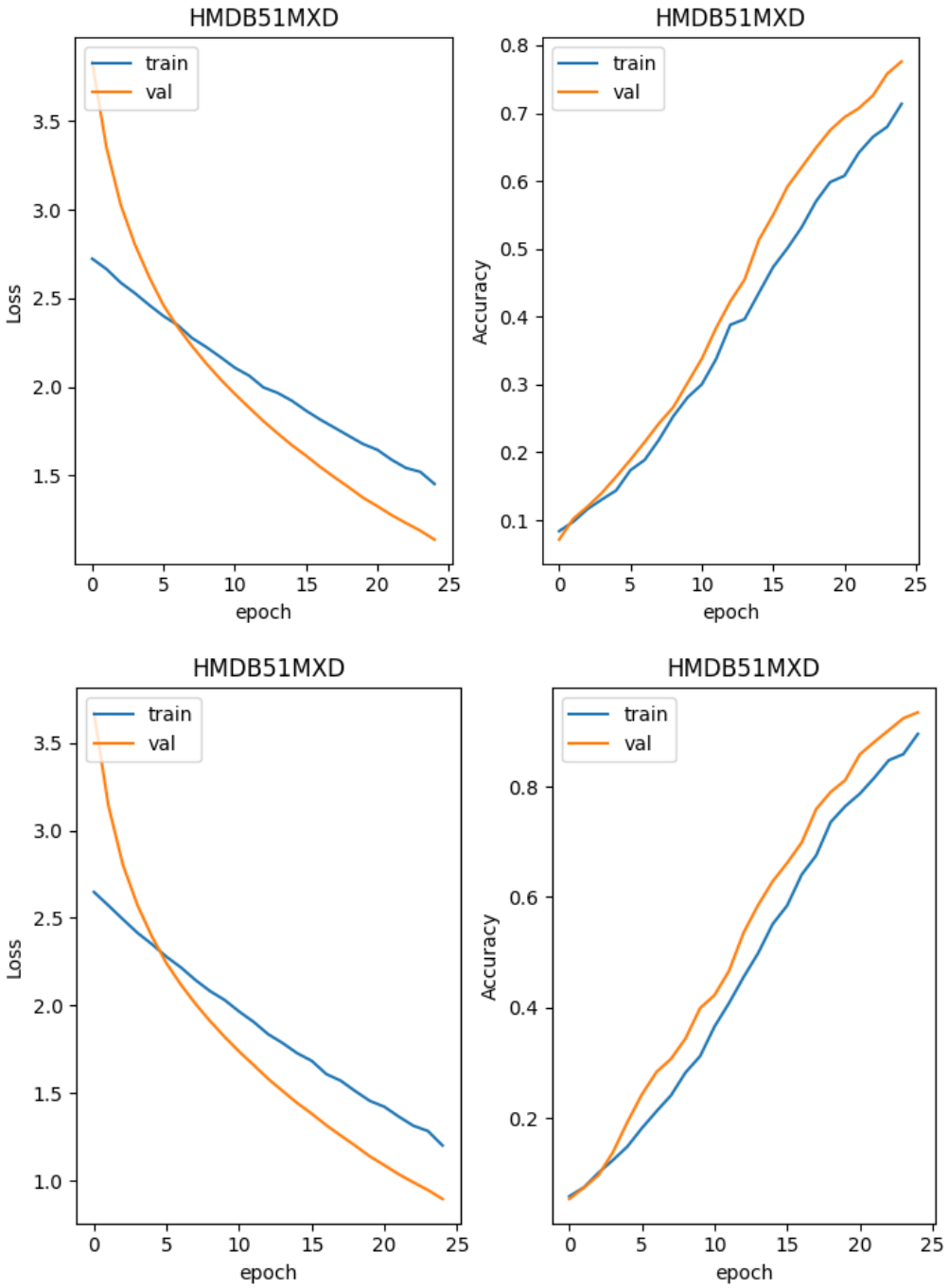
1) Do data augmentation for Stanford-40: We integrated data augmentation into our model by exploiting Keras data augmentation layers. In particular, we used 3 data augmentation techniques: horizontal flipping, random brightness and random contrast. Random contrast and brightness vary the contrast and brightness within a range specified with a factor parameter. In our case, we found that factor (-0.6, 0.6) for brightness and 0.6 for contrast worked reasonably well. In particular, we reduced overfitting and increased the model robustness on the testing set. We also tried using random translation and rotation, but the results were almost the same (< ~1% increase on validation accuracy). We got the best results with the 3 aforementioned augmentation layers.

2) Do data augmentation for HMDB51: We employ the same data augmentation techniques used for Stanford-40. In this case, we only use data augmentation for model 2. The reason behind this choice is that integrating data augmentation in the optical flow models required to decouple our data augmentation layers logic into a custom Python function (executed before the model's training is performed) and given the limited amount of time, we ended up opting for a quicker path.

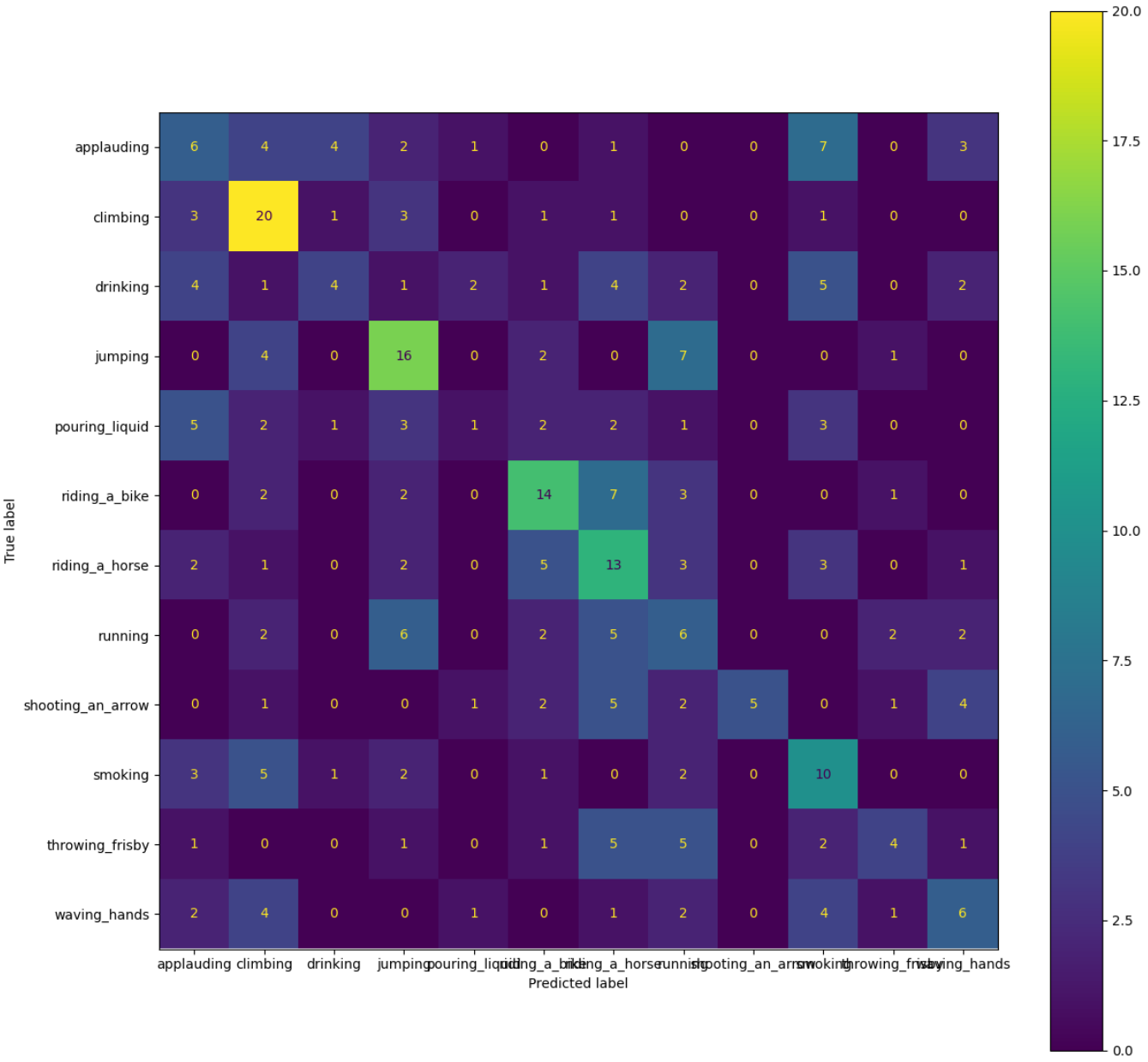
3) Experiment with at least three types of cooperation for the two networks (average, concatenate, maximum): We tested 3 different fusion operations: average, max, add and the one used in our two-streams model that is concatenation. Among all, we managed to improve our model from 0.27 to 0.302 testing accuracy with the Add operation. Again, because of the issue we have with validation accuracy, we think it makes much more sense to look at the testing accuracy and ignore the validation accuracies in this specific case. The plots are ordered as in the table.

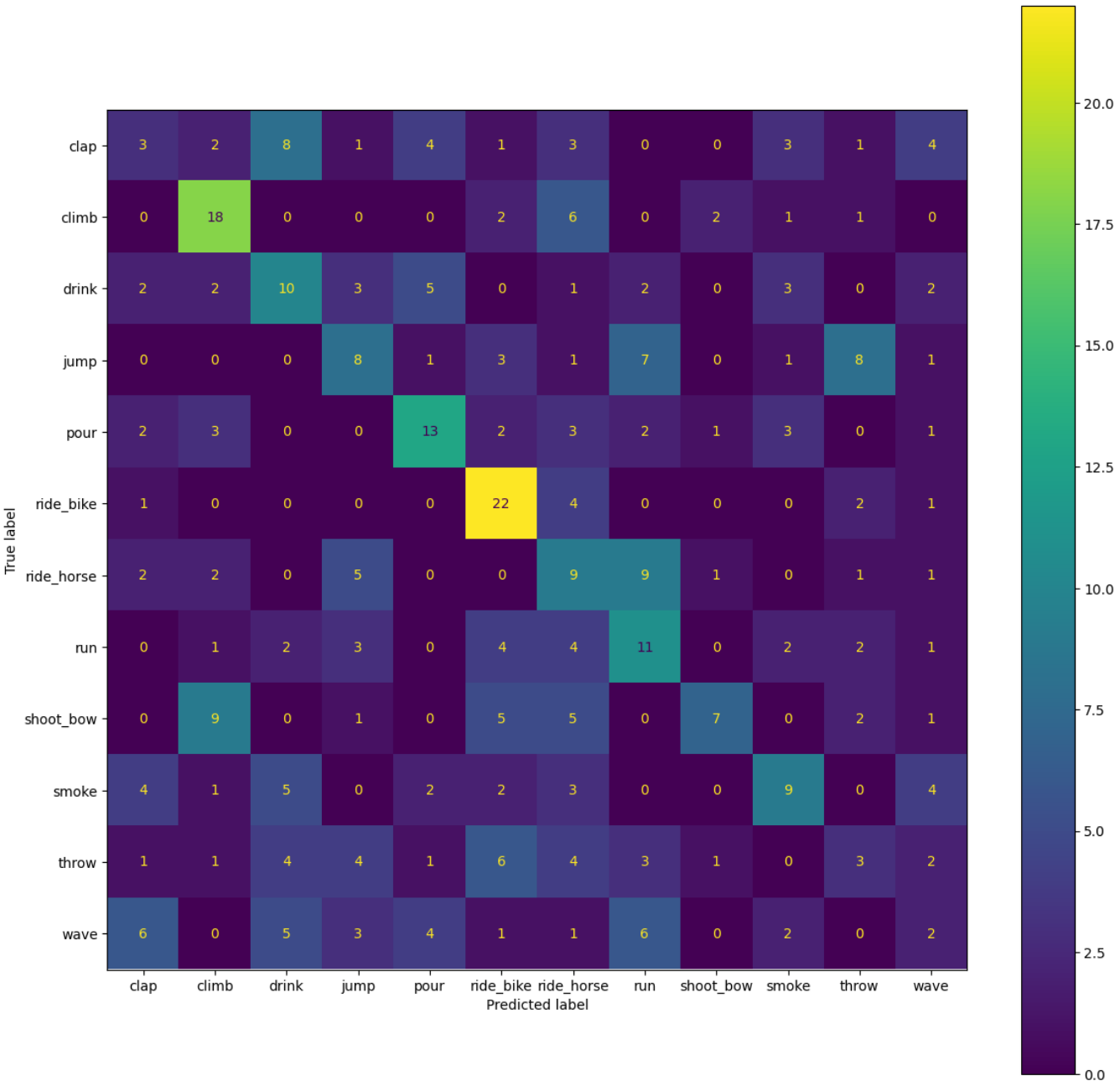
Merge layer	**Top-1 accuracy (train / valid)**	**Top-1 loss train**	**Test accuracy**
Average	0.886 / 0.928	0.97	0.297
Max	0.684 / 0.761	1.484	0.300
Add	0.913 / 0.958	1.123	0.302





4) Present and discuss a confusion matrix for your (1) Stanford 50 Frames and (2) HMDB51 Optical flow models: Both confusion matrices show an overall better ability of the models to classify the first 8 classes (counting from applauding/clap class). Among all misclassifications, the one happening the most for Model 1 involves the classes “riding_an_horse” and “riding_a_bike”. Both classes share substantial features, such as the human pose while riding both a horse and a bike. Model 1 seems to be a little bit biased towards the “riding_a_horse” class: it can be noticed that the number of samples misclassified for the “riding_a_bike” class is higher than the ones misclassified for the class “riding_a_horse”. On the other hand, Model 2 reaches much better results at classifying “riding_a_bike” samples, but introduces an overall misclassification rate increase among all classes. We think this is because of the difficulty gap between Stanford40 and HMDB51 datasets as well as the lack of a sufficient amount of data samples during training. We believe that both models can be improved by using more detailed samples (i.e. higher resolution), different learning settings (i.e. optimizers, learning rate) or a different model architecture.


applauding
climbing
drinking
jumping
pouring_liquid
riding_a_bike
riding_a_horse
running
shooting_an_arrow
smoking
throwing_frisby
waving_hands
True label
Predicted label
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0


clap 3 2 8 1 4 1 3 0 0 3 1 4
climb 0 18 0 0 0 2 6 0 2 1 1 0
drink 2 2 10 3 5 0 1 2 0 3 0 2
jump 0 0 0 8 1 3 1 7 0 1 8 1
pour 2 3 0 0 13 2 3 2 1 3 0 1
ride_bike 1 0 0 0 0 22 4 0 0 0 2 1
ride_horse 2 2 0 5 0 0 9 9 1 0 1 1
run 0 1 2 3 0 4 4 11 0 2 2 1
shoot_bow 0 9 0 1 0 5 5 0 7 0 2 1
smoke 4 1 5 0 2 2 3 0 0 9 0 4
throw 1 1 4 4 1 6 4 3 1 0 3 2
wave 6 0 5 3 4 1 1 6 0 2 0 2
True label
clap climb drink jump pour ride_bike ride_horse run shoot_bow smoke throw wave
Predicted label
20.0 17.5 15.0 12.5 10.0 7.5 5.0 2.5 0.0