First Person Action Recognition: different approaches exploiting Self-Supervised Task and two-in-one Flow Modulation

Paolo Alberto
Politecnico di Torino, Italy
s278098@studenti.polito.it

Lorenzo De Nisi
Politecnico di Torino, Italy
s276545@studenti.polito.it

Carmine De Stefano
Politecnico di Torino, Italy
s278176@studenti.polito.it

Abstract

In this paper we explore different methods for performing a first-person action recognition task on the GTEA61 dataset. We start by implementing a structure based on Ego-RNN: a two-stream architecture that works separately on the frames of the videos and on motion features extracted from optical flow. We then try to improve on top of this architecture by implementing a self-supervised task capable of working jointly on spatial and temporal features.
Finally, we implement a two-in-one stream architecture, embedding RGB and optical flow into a single stream.

1. Introduction

The classification of human actions from videos has been one of the most challenging and well studied tasks in computer vision, and it has the potential of creating a great impact across a large number of applications.
However, the majority of the studies on the subject concentrates on third person action recognition. While this classification task has seen a lot of progress over time, the more niche task of detecting first-person actions and activities can still be considered as a less explored field, and has seen some interesting developments in recent years.
One of the most important differences of this task is the presence of ego-motions, caused by the movement of the user, that is usually wearing the camera on the head or body.

Another challenging problem that arises when dealing with first person videos is the need to perform activity recognition: while classic action recognition consists of identifying a simple motion pattern (take, put, open...), in an activity recognition task we are working on a more fine-grained scale (take coffee, put cheese, open chocolate...).
When working with this kind of data the same object will be involved in multiple actions. Because of that, it becomes important to take into account both the objects and the motions characterizing the user performing the action.

2. Related Works

In a 2018 paper, Sudhakaran *et al*. [3] proposed to address the first person recognition task by combining an architecture for the analysis of the frames with a temporal network, capable of learning from image features with spatial attention. This architecture, called Ego-RNN, is used as the starting point for our experiments.

One of the problems of this model is that frames and optical flow features are learned separately, merging the two branches only at the end of the model. Taking this into account, we expand on this architecture by implementing the self-supervised task proposed by Planamente *et al*. [1].
We implement this step by creating a self-supervised motion segmentation task, and feeding the backbone of the Ego-RNN network to it. As a result of the added MS task, the backbone will try to learn features regarding the object movements, that are supposed to be beneficial to the classification task.

After implementing the MS task both as originally conceived in the paper and as a regression problem, we take it a step further, by applying a motion condition and a motion modulation layer to our model, following the schema suggested by Zhao and Snoek [5], with the goal of using the features from this layers to modulate the RGB network.

All of models presented in this paper were trained and validated on the GTEA61 dataset: a fine-grained collection of videos corresponding to 61 actions, performed by 4 different users. The code related to the various steps described above, together with some animated gifs and the downloadable weights of the models, is made available at `https://github.com/FPAR-NET/FPAR`.

3. Implementation of Ego-RNN

A ResNet-34 network pre-trained on ImageNet is implemented as the backbone for this model. This network is used together with a spatial attention layer in order to obtain image features with spatial attention. In this layer, we calculate the class activation map of the winning class, and we convert it to a probability map by applying softmax.

The features obtained allow us to encode both temporal and spatial dimensions at the same time, by using a convLSTM module. The output of the convLSTM is then fed to an average pooling layer and finally to a fully connected module for classification.

For the first stage we keep the weights of the pre-trained ResNet-34 locked, while training only the parts of the model without any trained weights yet (the convLSTM module and the final classifier). In this stage, as stated in [3], the network is trained for 200 epochs, with an initial learning rate of 10^{-3}, and a step down policy characterized by a decay factor of 0.1 after 25, 75 and 150 epochs. The best accuracy obtained with this parameters, with 16 frame videos, is 46.5%. For the second stage we train also the spatial attention layer, together with the last layer of the ResNet and its fully connected layer. With this changes in place, we expect to see an increase in the network's performance, since in this stage our model will do a better job at learning spatial and temporal features.
For this phase, as done in the first stage, the hyperparameters used are the same of the original paper: 150 epochs of training with a learning rate of 10^{-4}, decayed after 25 and 75 epochs by a factor of 0.1. The increase in accuracy is consistent with the expected results, and in line with the results obtained by Sudhakaran *et al*. [3].

For comparison, the RGB network was also trained without the spatial attention layer, obtaining 47.3% accuracy in the best of cases. In combination with the above convLSTM-attention model, another method often adopted in literature ([3][2][4]) for action recognition tasks, is the use of stacked optical flow images, in order to train the temporal stream to recognise actions from motion. The flow images are taken from the provided dataset, they are arranged in stacks of 5 and are then fed to a temporal network based on a ResNet-34 model pretrained on ImageNet. For the first run, the same parameters of the original paper were used, and the model was trained for 750 epochs, with a learning rate of 10^{-2}, decayed after 150, 300 and 500 epochs by a factor of 0.5. With this run, the best accuracy obtained on the validation set was slightly above 40%.

After training spatial and temporal features separately, the original paper proposes to concatenate the output of the two networks, adding a new fully connected layer to get the class scores, and performing a fine-tuning for 250 epochs. The best accuracy on the validation set, with 16 frame videos as inputs, was of about 63%.
However, when looking at the loss, we can see that a lot of overfitting is happening on the training set. We can speculate that, since the original model was trained on a larger quantity of data (25 frames instead of 7/16), the original Ego-RNN model had more features to learn with respect to ours. Because of that, our implementation of Ego-RNN takes less time before it begins to overfit.

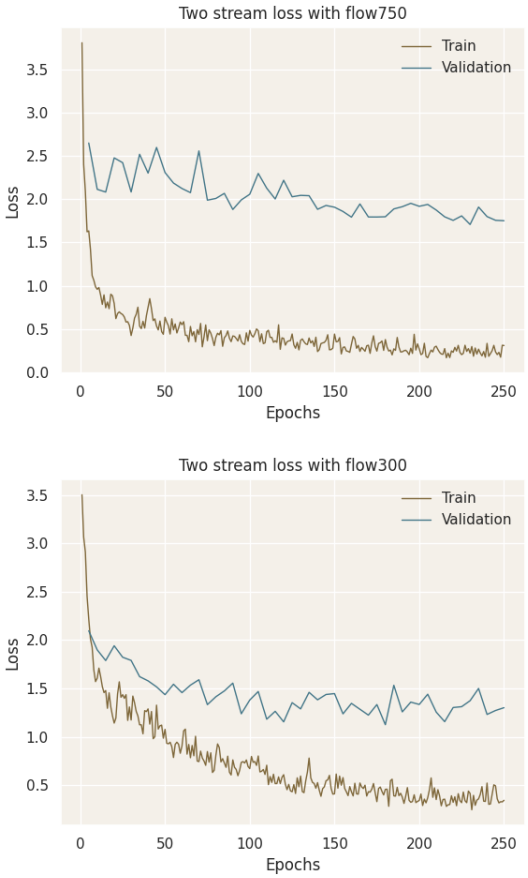

Figure 1. Loss of the two stream network before and after the reduction of epochs

Moreover, if we analyse the behaviour of the two models that compose the two-stream model, we can see that the temporal network has the same problem. This model, in fact, starts to overfit when less than half of the training is completed, and after about 300 epochs the accuracy on the validation doesn't change too much anymore, with just some random fluctuations. Since the original model saves just the model with the highest accuracy on the validation set, it may happen that, due to random fluctuations, the saved model is found in the last epochs, even if at that

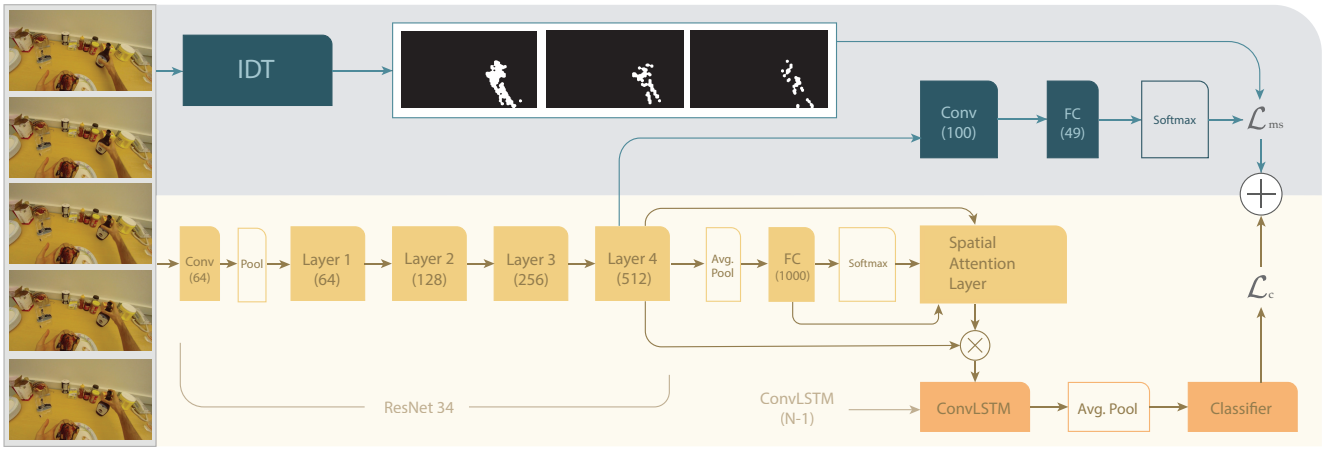

Figure 2. Self-supervised network architecture

point the model is overfitting, and there are no relevant improvements.

In figure 1 we plot the effect of this change on the two-stream loss before and after the reduction of the number of flow epochs from 750 to 300. We also make available the loss of the temporal branch in appendix (figure 9).
Assuming that this behaviour of the temporal-warp flow could have repercussions on the whole two-stream model, we decided to train it again for just 300 epochs. This approach proved to be beneficial, and was able to mitigate the overfitting effect on the two-stream model as well. After this changes were put in place, great improvements on the two-stream metrics were achieved, with the accuracy reaching 73.7% in the 16 frame case.
For the remaining part of this report, we will use this better performing version of the temporal network, and we will be referring to it as "flow300".
Complete results regarding the different training steps of the network are shown in the table below.

Configurations	7 Frames	16 frames
ConvLSTM	37.7%	46.5%
ConvLSTM-attention	55.2%	64.9%
Temporal-warp flow	42.2%	42.2%
Two-stream (joint train)	57%	73.7%

Table 1. Results of the experiments on the GTEA61 dataset

4. Adding a self-supervised task

Until now, the motion and temporal networks were trained separately. As suggested by Planamente *et al*. [1], we try to implement a motion segmentation self-supervised task on top of the second stage of the Ego-RNN network, in order to learn jointly information related to motion and frames. Another advantage that comes with this network is that, differently from Ego-RNN, it can be trained end-to-end in a single run.

As we can see from the schema above, the output of Ego-RNN's layer 4 is fed to the MS task: a simple branch composed of a convolutional block capable of reducing the channels from 512 to 100, and a fully connected layer of size 49, followed by a softmax. The output of the MS task is used to calculate the per-pixel cross-entropy between the computed 7x7 image and the ground truth, obtained from the motion maps already available in the original dataset.
These mmaps are down sampled to a 7x7 matrix, and each of the resulting 49 pixels is set to 0 or 1 depending on a given threshold. For the first run, the threshold was set to 0.5. However, setting it to 0 proved to be beneficial and increased the performance of the self-supervised task. For the calculation of the $\mathcal{L}_{ms}$ loss, we used the binary_cross_entropy function provided by PyTorch. In the beginning, the results obtained by training this model were not comparable to the ones of the two-stream network implemented in the previous section, and even doing some hyperparameter optimization didn't provide great improvements. However, a turning point was reached by reducing the kernel size of the convolutional layer in the MS task

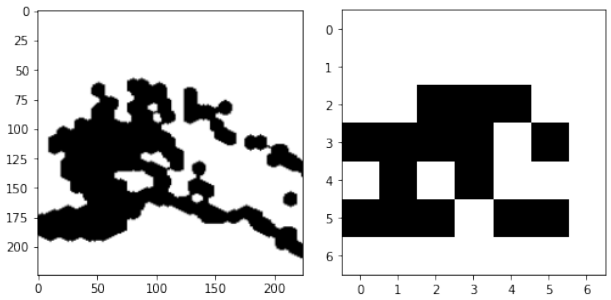

Figure 3. Mmaps after down sampling

from 7 to 1. This value is admittedly an unconventional choice, and the improvement obtained by using it can be explained by the fact that, with this value, the MS task is not losing information about the motion clues that we are trying to encode with this self-supervised task.

After this changes were put in place, the best accuracy obtained on the validation set was 67.5%, obtained with 150 epochs of training, a batch size of 32, an initial learning rate of 10^{-4} and a step down policy characterized by a decay factor of 0.1 after 25 and 75 epochs. For the various configuration of hyperparameters that were tried with this self-supervised model, we refer to table 2 in the dedicated section.
For this task we can consider ourselves satisfied with the results obtained. However, when looking at these metrics (and more generally at the other accuracy values reported in this paper), an important thing to remember is that the validation was done on a small number of samples, since the dataset used for this project is relatively small for a deep learning task of this kind, especially if compared with other similar datasets (*e.g*. Epic Kitchens).

5. CAM visualization

Inside the spatial attention layer of Ego-RNN we take advantage of the average pooling of the ResNet34 model to compute class activation maps as proposed by Zhou *et al*. [6], in order to identify the regions of the image that were used to predict the winning class. To do it, we just need to map the predicted class score back to the previous convolutional layer, thus highlighting the class-specific regions that are being activated the most. While these maps have a useful purpose in the inner workings of the network, they can also be visualized, and used to find out the importance that the classificator gives to the different regions in the image.

We can visualize the effect of the self-supervised task by plotting the class activation maps before and after the implementation of the MS task. As we can see from the image below, before training the self-supervised task the model is focusing on the plate, while afterwards the network is capable of better identifying the jar. In the second example, the focus shifts from the container to the lid when identifying the opening of mayonnaise.

Figure 4. Class activation maps before and after the introduction of the MS task.

The complete animations related to the displayed images, together with some other examples on self-made videos, are made available in the provided GitHub repository, together with the code.

6. Self supervised task as a regression problem

Since we are using a threshold to arbitrarily decide if a pixel of the motion maps is switched on or off, we could argue that some useful information is lost during this process. A better idea would be to use a regression technique instead.
To achieve this goal, we add a sigmoid to the head of the MS task. We also need to remove the threshold-based conversion of the mmaps that we implemented in the original self-supervised task, and we have to change the loss function used for the calculation of $\mathcal{L}_{ms}$.

In order to find an efficient loss function we used an empirical approach, performing experiments with various functions. After some runs, the best performing parameters used for this task were obtained when training for

150 epochs, a batch size of 16, with a step down policy implemented after 50 and 100 epochs. All of the various loss functions were implemented with this hyperparameters in place. The best validation accuracy (72.8%) was found with the use of the Mean Squared Error loss:

$$MSE = \frac{\sum_{i=1}^{n}(x_i - y_i)^2}{n} \tag{1}$$

We mention also another interesting loss function that we implemented with good results: the Kullback–Leibler divergence. This formula is defined, for two discrete probability distributions P and Q defined on the same probability space $\mathcal{X}$, as:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right) \tag{2}$$

While not providing the best accuracy value in absolute terms, the run performed with this loss provided an acceptable accuracy nonetheless, and it proved to be helpful in lowering the overfitting effect when used in the experiments described further on (two-in-one stream action detection).

At the end of the self-supervised task training, we need to merge the obtained loss together with the one coming from the RGB branch. However, when looking at the value of these two losses, we notice that they are often characterized by different orders of magnitude. For example, in the case of the Kullback–Leibler loss, we notice that its value is about one order of magnitude smaller than the $\mathcal{L}_c$ value. There is a discrepancy also with the use of MSE, with a value that is four times higher with respect to $\mathcal{L}_c$.
An idea in this case would be to weight one of them, in order to have comparable values when merging the losses together. However, repeating the training with this change in place didn't prove to bring improvements, so we decided to move on to the hyperparameter optimization phase, keeping the original implementation with the unweighted losses.

The best accuracy on the validation set obtained when applying the MSE loss was 72.8%. For the results obtained with the other loss functions we refer to table 3 in the next section.

To compare this model with the original self-supervised task, we can plot the loss obtained with the regression task and compare it to the one obtained before. We can also visualize the 7x7 maps and compare it with the ones obtained.

7. Hyperparameter optimization

Some hyperparameter tweaking was performed after the implementation of the self-supervised task. We changed

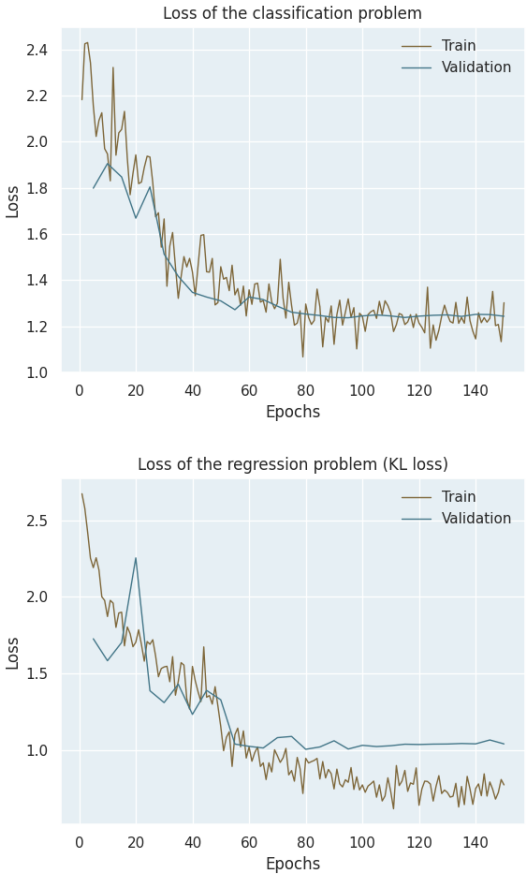

Figure 5. Loss before and after the regression task

epochs, batch size, step size (which is the number of epochs before the application of a step down policy on learning rate) and type of optimizer. The results are shown in the table below.

Epochs	Batch size	Step size	Optimizer	Accuracy
150	32	[25, 75]	Adam	67.5%
200	32	[50, 100]	Adam	64.9%
150	32	[25, 75]	SGD	53.5%
150	16	[50, 100]	Adam	67.5%
150	16	[25, 75]	Adam	51.7%

Table 2. Hyperparameter optimization on self-supervised task

For the implementation of the regression problem, the experiments were mainly focused on experimenting different loss functions for the $\mathcal{L}_{ms}$ loss of the self-supervised task. The details of these experiments, trained for 150 epochs, a batch size of 16 and a step size of [50, 100], are shown in the table below.

After these experiments on different loss functions,

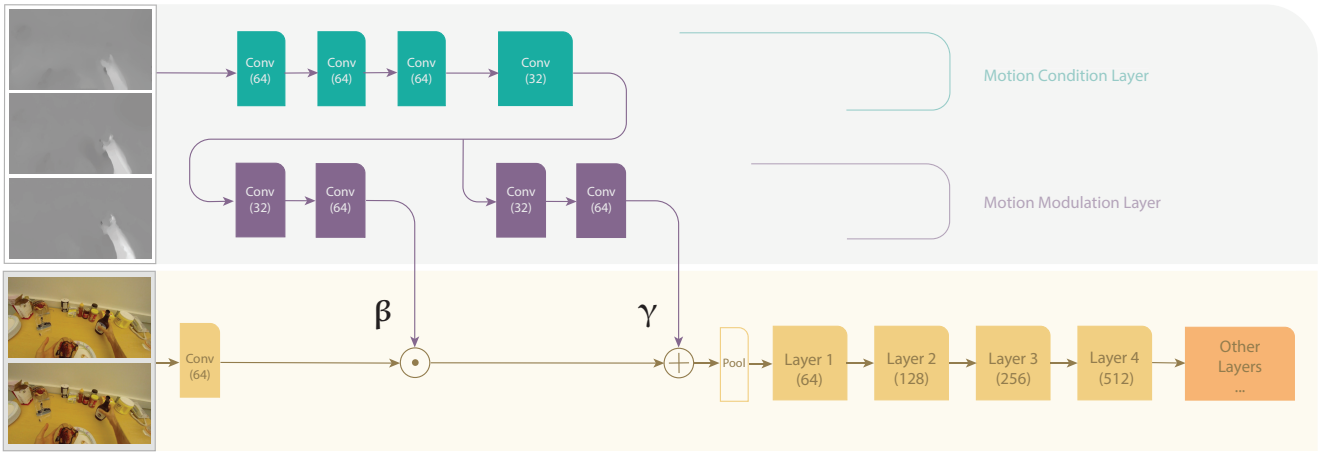

Figure 6. Two-in-one network architecture

Loss function	Accuracy
L1	67.5%
Smooth L1	70.2%
MSE	72.8%
Soft-margin	66.7%
Kullback–Leibler	70.2%

Table 3. Accuracy with different loss functions

we tried to re-train the two-stream model by using the best weights obtained with the regressive task, in combination with the usual flow300 weights obtained from the original temporal network. While increasing the overfitting effect by a lot, there was actually a slight improvement on the validation accuracy, reaching a value of 74.5% with the KL loss.

8. Two-in-one stream action detection

In 2019, Zhao and Snoek [5] proposed a method to embed RGB and optical-flow into a single two-in-one stream, with the goal of performing spatio-temporal action detection with a model that can be trained end-to-end in a single run and can be easily implemented on top of a two-stream action detection model.
This allows us do conduct some more experiments with the optical flow images (that we stopped using in favour of mmaps computation), while reducing the higher computational costs required by the two separate streams that come as a disadvantage with the implementation of Ego-RNN.

This is achieved with the addition of two new layers: a motion condition and a motion modulation layer. The first layer applies a series of convolutions to the initial flow images, with the goal of generating some simple features from them. After this, the motion modulation layer learns a pair of affine transformations parameters β and γ through the use of two parallel series of convolutional layers. These parameters are then used in order to modulate the information coming from the first convolutional layer of the RGB network, by applying to the RGB features F^{rgb} the following transformation:

$$\mathcal{M}^2(F^{rgb}) = \beta \odot F^{rgb} + \gamma \tag{3}$$

Where $\odot$ is an element-wise multiplication operation.
Due to the limitations given by the Colab hardware, we were forced to reduce the batch size to 4, increasing by a lot the training time. With the first implementation of this model we obtained an accuracy of 66.7%, reaching a metric similar to the one obtained with the first self-supervised task that we implemented before, and showing the effectiveness of this kind of approach.
In order to visualize better the mechanics involved in the implemented motion layers, we can plot the output of the first convolution before and after the application of the two motion layers, and also the inputs coming from β and γ.

As a last experiment, we tried to train the motion layers together with the first convolution of the RGB network. This is done since this layer can be considered as a part of the first block of our network, together with motion layers. Because of that, it makes sense to try training this first stage all together. The experiment was performed multiple times,

Figure 7. Gray-scale visualizations of pre-motion layers, beta, gamma and post-motion layers

with MSE, KL, and also without the MS task at all. A first training was done starting from the first stage (convLSTM), to compare the results with the other approaches.
Training also the first convolution proved to be beneficial in some cases, with an accuracy of 71% obtained with the KL loss. However, as we can see from the figure 8, there was also an increase in overfitting. This overfitting was even greater with the MSE loss, and that is probably the reason why the MS task with KL loss was the best performing after the training of Motion Layers + conv1.

In the following table we can see all the details regarding the changes in accuracy obtained after the training of motion layers together with conv1.

Network	Motion Layers	ML+conv1
ConvLSTM	66.7%	62.3%
ConvLSTM attention	62.3%	65.7%
MS task with KL	67.5%	71.0%
MS task with MSE	70.2%	67.5%

Table 4. Hyperparameter optimization on self-supervised task

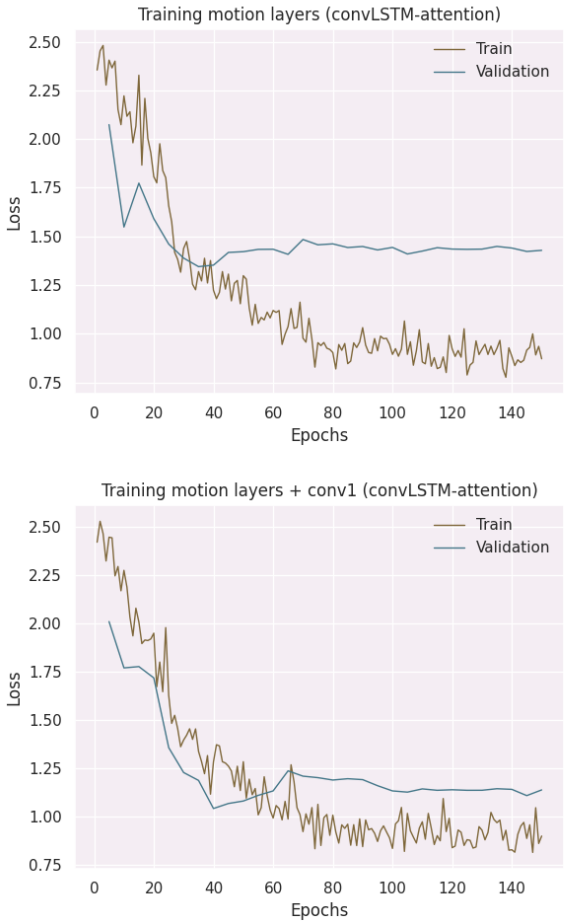

Figure 8. Loss of the two-in-one model before and after the joint training on motion layers + conv1

9. Conclusion

To summarize the steps followed during this project, we provided an implementation of Ego-RNN with 7 and 16 frames, obtaining acceptable results. We implemented a Motion Segmentation task on top of the Ego-RNN backbone, obtaining good results, especially with the implementation of a regressive version of said task. We observed how a lighter, end-to-end trainable two-in-one model is capable of providing some good accuracy values, that are comparable with the results obtained before.
Finally, we plotted a confusion and a correlation matrix for each of the three networks analysed in the project. When looking at the correlation matrices (13, 15, 17), we have an interesting insight: the division between the different actions characterizing the activities(open, close, take...) is clearly visible in the plots. A possible idea to expand on the topic in the future would be to create specialized branches for the classification of action and objects separately.

References

[1] M. Planamente, A. Bottino, and B. Caputo. Joint encoding of appearance and motion features with self-supervision for first person action recognition, 2020.

[2] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos, 2014.

[3] S. Sudhakaran and O. Lanz. Attention is all we need: Nailing down object-centric attention for egocentric activity recognition, 2018.

[4] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. V. Gool. Temporal segment networks: Towards good practices for deep action recognition, 2016.

[5] J. Zhao and C. G. M. Snoek. Dance with flow: Two-in-one stream action detection. 2019.

[6] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization, 2015.

Additional material

Figure 9. Loss of the temporal network before and after the reduction of epochs

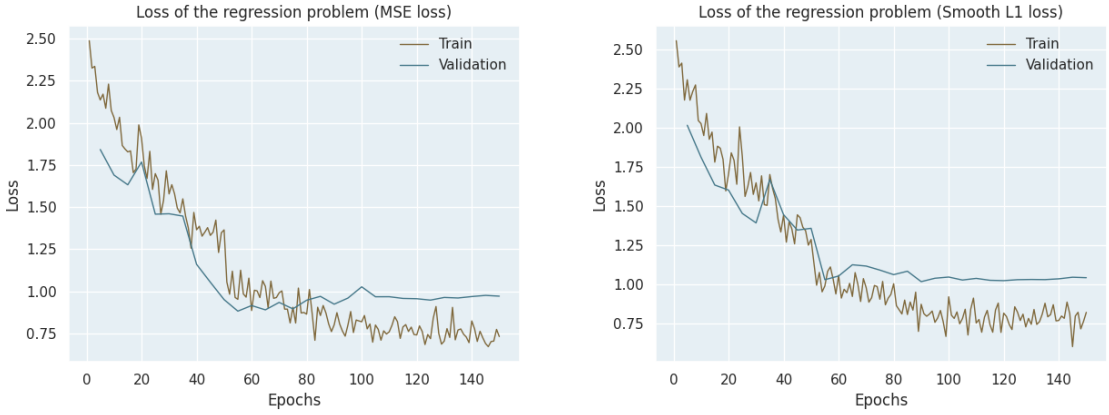

Figure 10. Loss of the regression task with two other functions

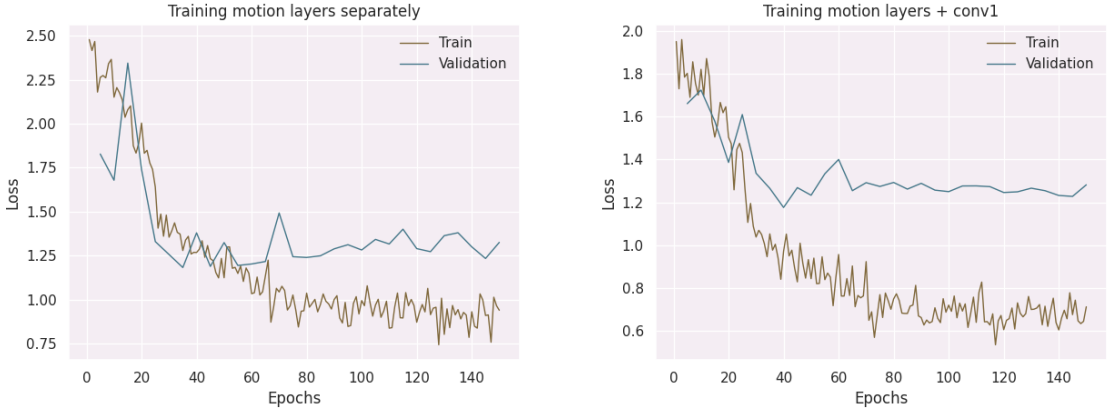

Figure 11. Loss of the two-in-one model before and after the joint training on motion layers + conv1, with KL loss

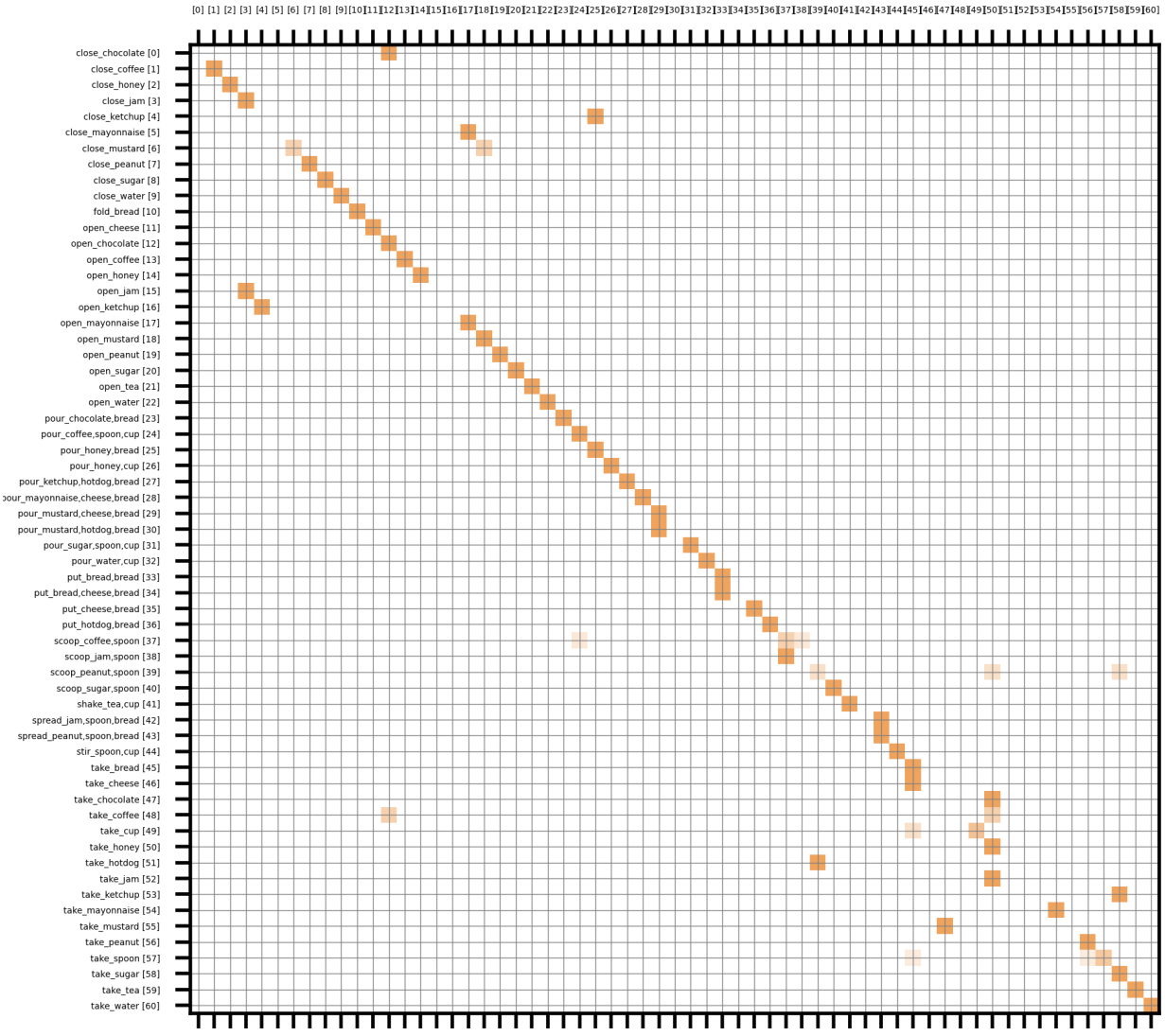

Figure 12. Confusion matrix for the two stream network

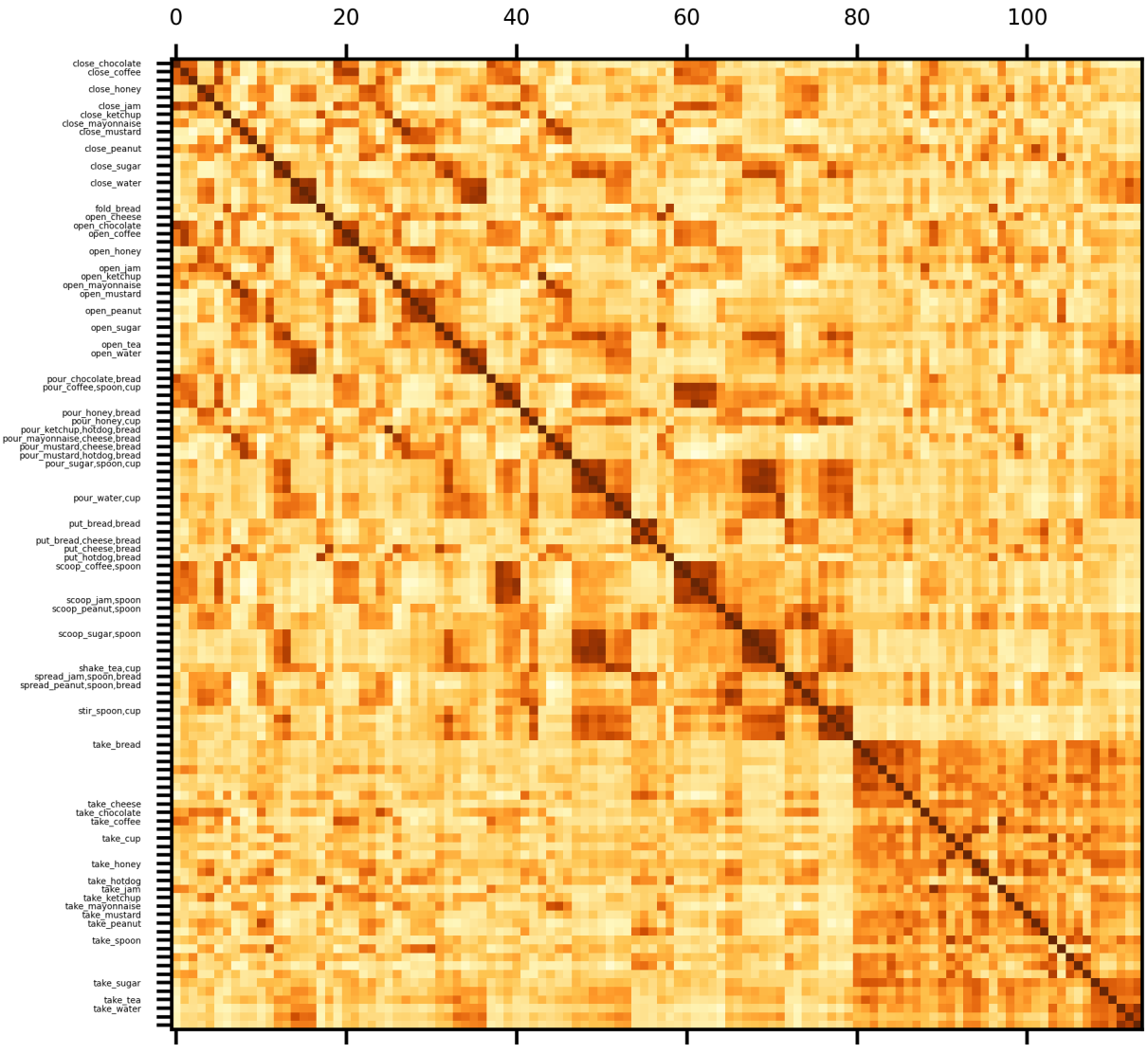

Figure 13. Correlation matrix for the two stream network features before classification

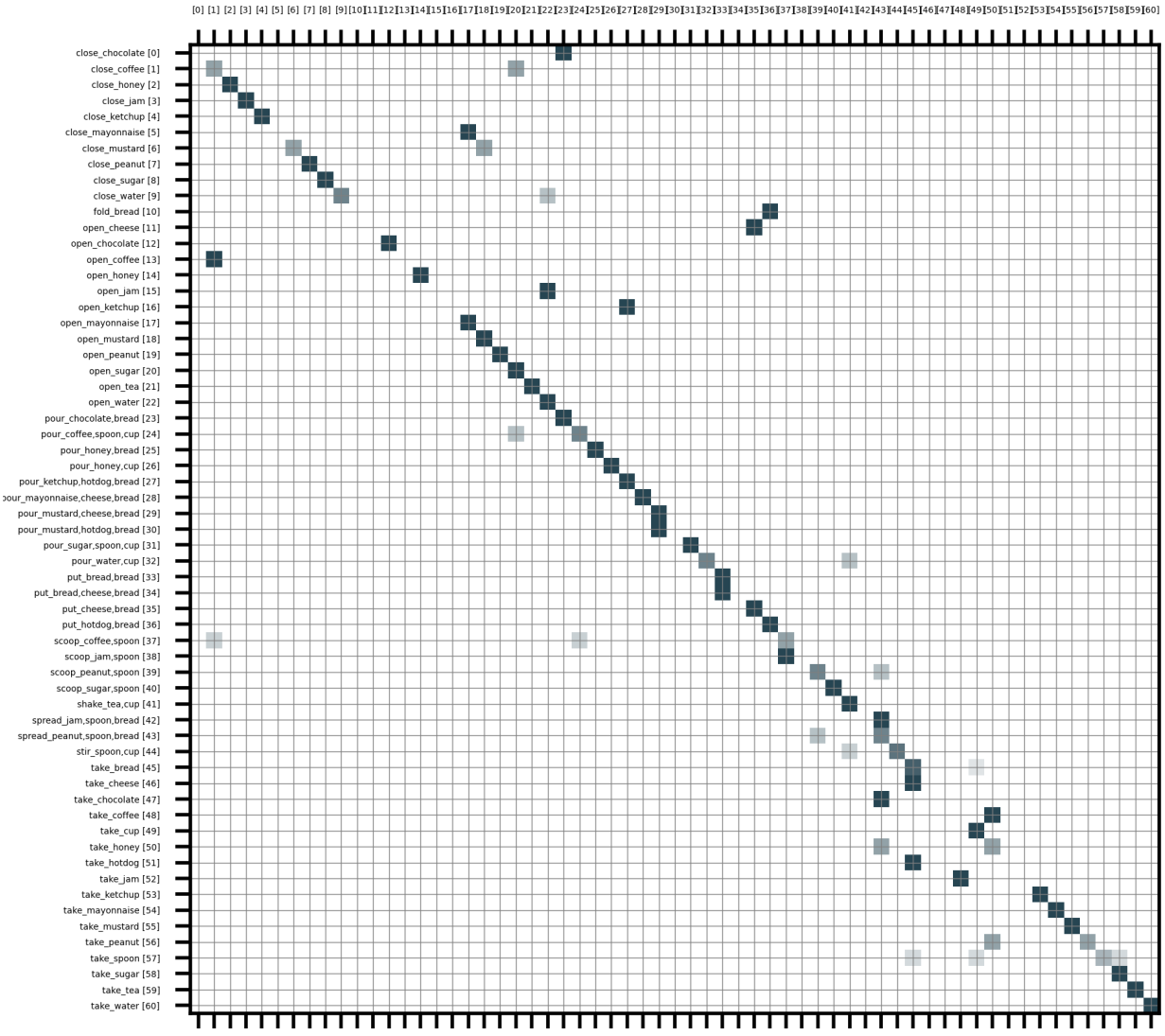

Figure 14. Confusion matrix for the MS task model

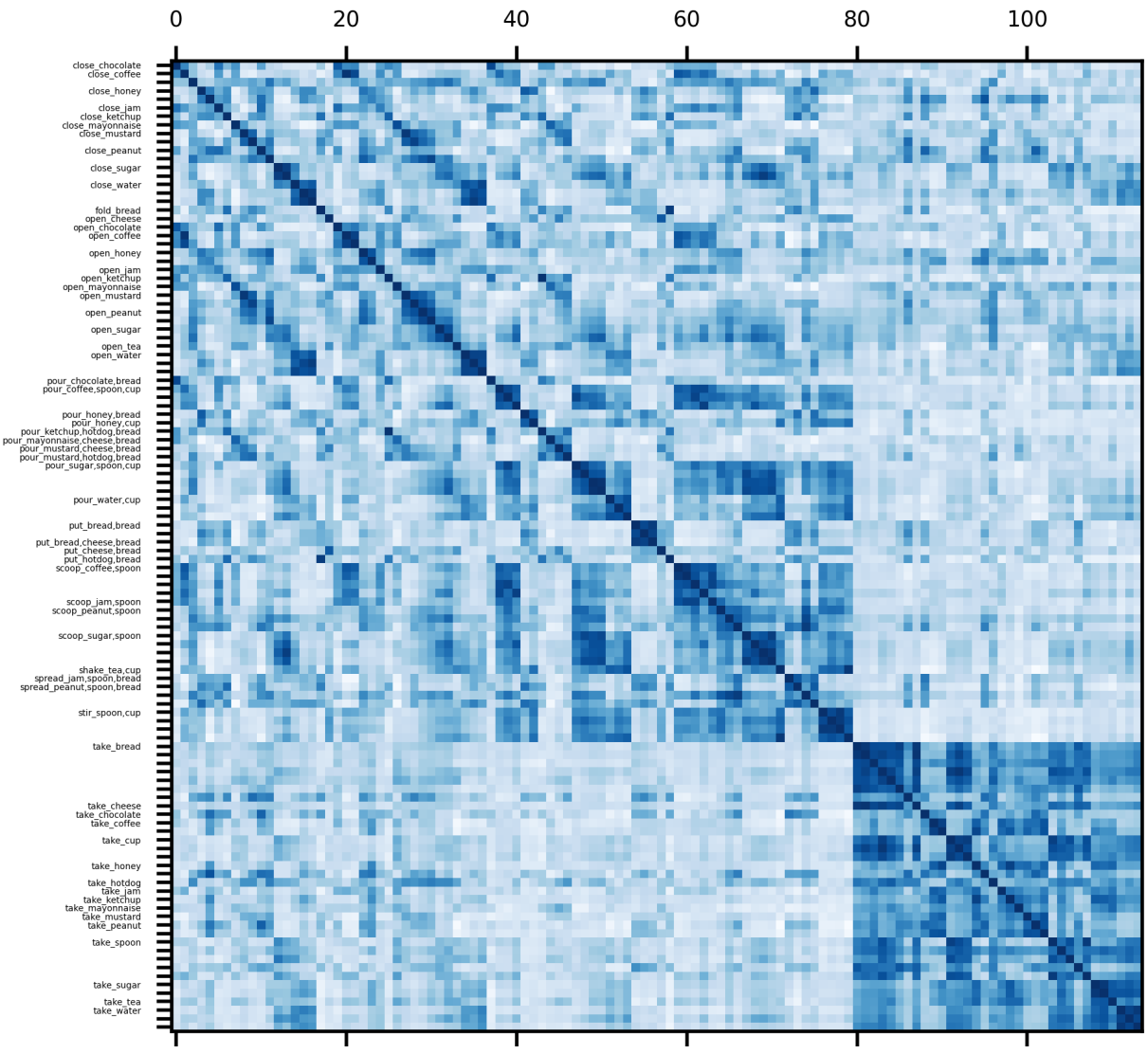

Figure 15. Correlation matrix for the MS task model features before classification

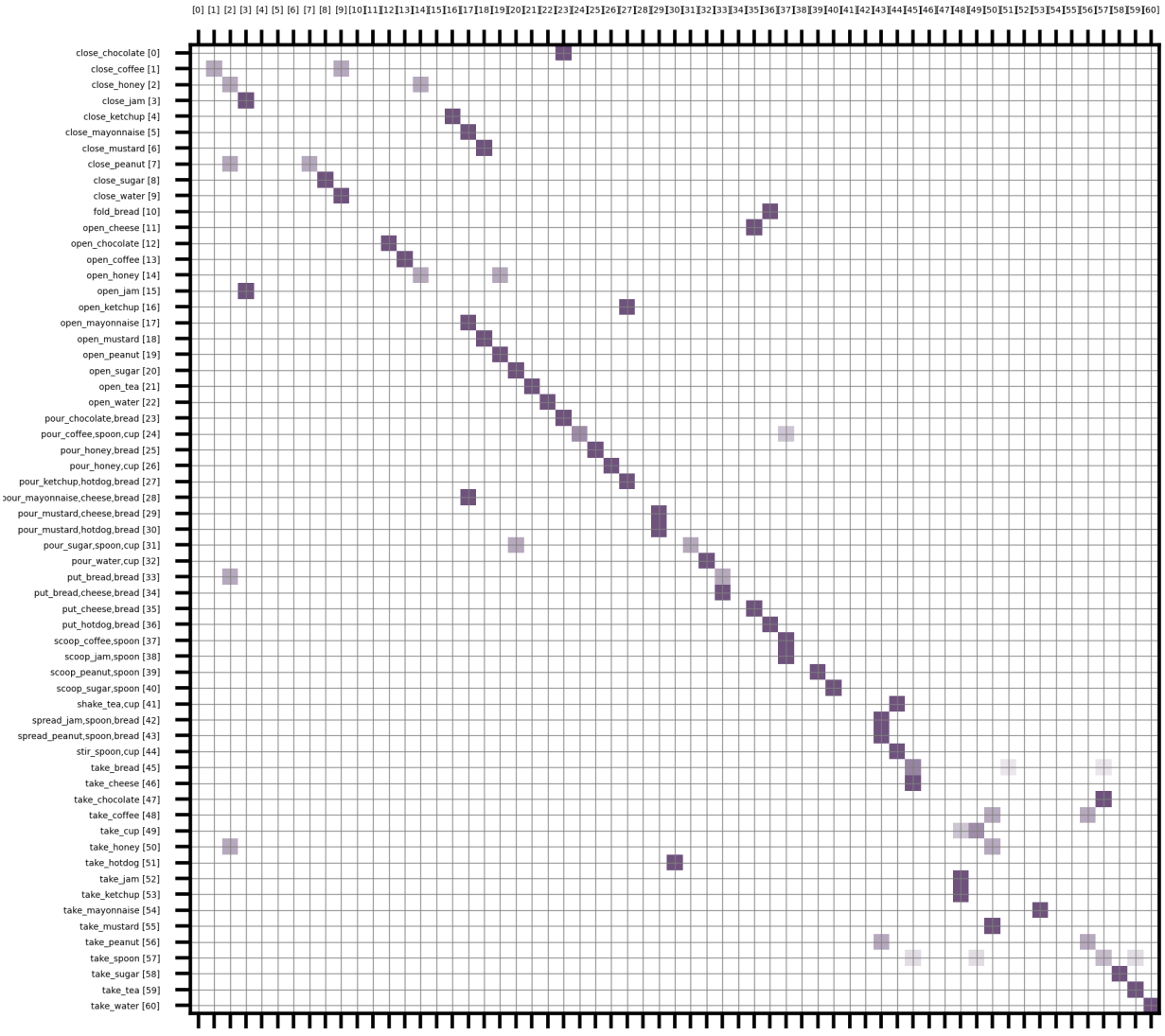

Figure 16. Confusion matrix for the two-in-one model

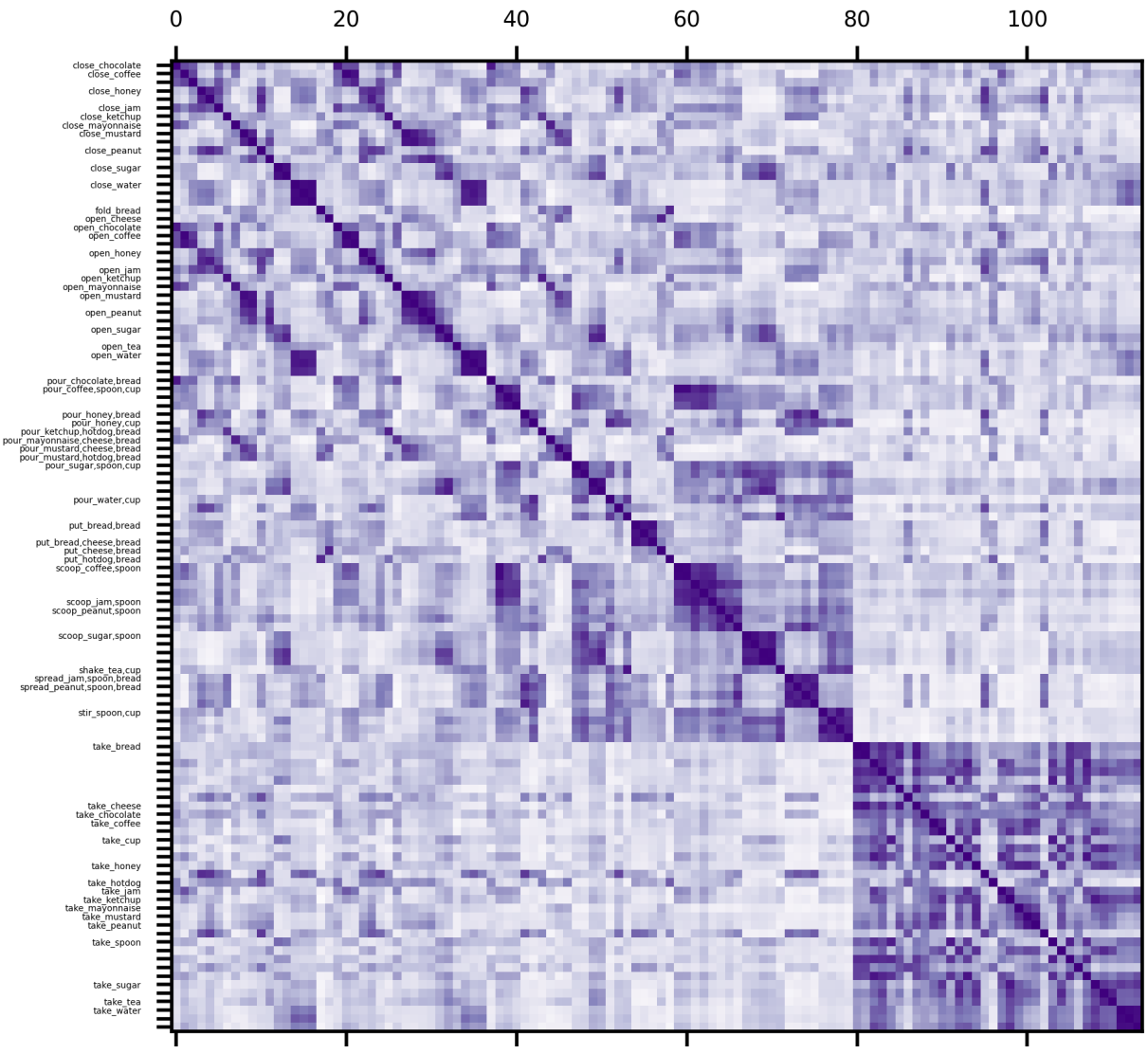

Figure 17. Correlation matrix for the two-in-one model features before classification