Project #11: Increasing Network Size for Class Incremental Learning

Introduction

Requirements

```
numpy 1.18.5
matplotlib 3.2.2
torchvision 0.6.1+cu101
torch 1.5.1+cu101
google-colab 1.0.0
```

note: if the experiment is run directly into colab, all the requirements are automatically satisfied and there's no need of any installation.

Instructions

The whole experiment has been developed and run completed on the Google Colab cloud platform.

To re-run the experiment, upload the jupiter notebook provided together with this file on Colab, change the runtime type to GPU, and then launch the runtime execution ('run all') which will execute all the notebook cells sequentially.

The numpy files relative to the train and test images and labels have to be located inside the DIR folder, which by default is 'Bioinformatics'. It can be changed by changing accordingly the DIR global variable, which can be found and edited in the second notebook cell 'Set arguments'. By default, your Google Drive will be loaded, asking for a password for mounting your drive, and the working directory will be set to 'drive/My Drive/Bioinformatics' or, in case the default DIR was changed, it will be set to 'drive/My Drive/<DIR>'.

In case you wish to execute the experiment on a local runtime, just set the global variable LOCAL to True and launch the experiment from the Bioinformatics (or your DIR) directory.

Note: All the results presented in this report have been obtained using Colab's 'Tesla T4' GPU. Ensure to be using the same GPU when you initialize the Colab runtime in order to obtain our same results. This is due to the fact that floating point operations are handled differently depending on the processing hardware.

Task

Learning new classes by increasing the network size.

Table of Contents

Illustration Index

Data

The dataset used in this study was extracted from a public repository of H&E[1] stained whole-slide images (WSIs) of colorectal tissues. It consists of images split in 7 classes, among which 3 are the classes of interest:

- Healthy tissue (H)
- Cancer (AC)
- Adenoma (AD)

and the remaining 4 are of spurious images, associated with a fake class from AC, AD, H:

- Blood (BLOOD)
- Fat (FAT)
- Glass (GLASS)
- Stroma (STROMA)

The dataset is represented by images (32x32x3) in form of numpy arrays, already divided into a training and testing set:
X{train, test}.npy shape ({12336, 7308}, 32, 32, 3)
containing the training and test images

real_classes{train, test}.npy shape ({12336, 7308},)
containing the real class corresponding to {train, test} images

Below we can take a look of some dataset image samples belonging to the different classes:

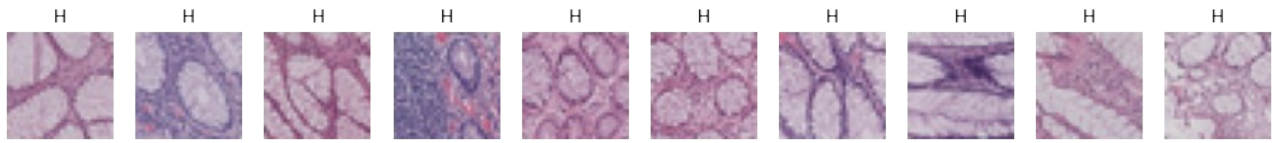

Fig. 1: Samples of class H

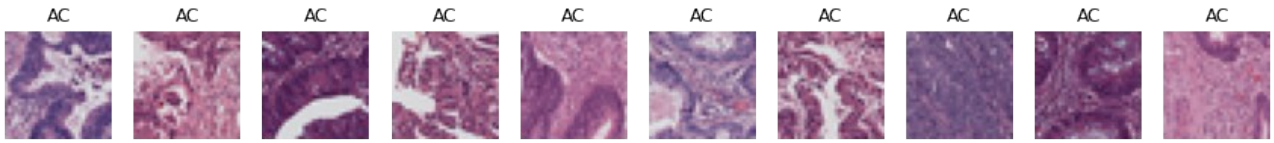

Fig. 2: Samples of class AC

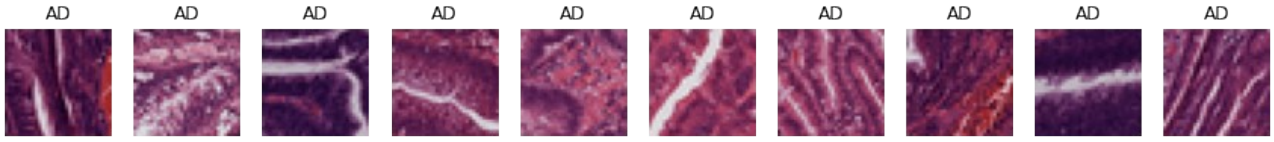

Fig. 3: Samples of class AD

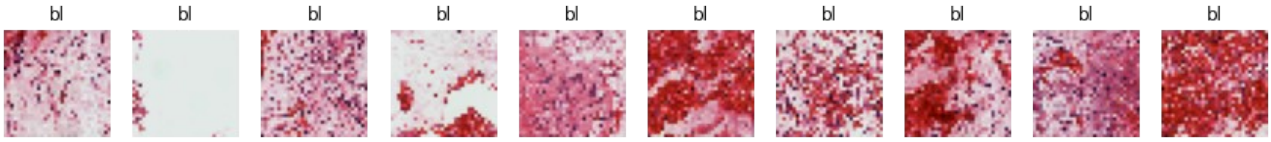

Fig. 4: Samples of class blood

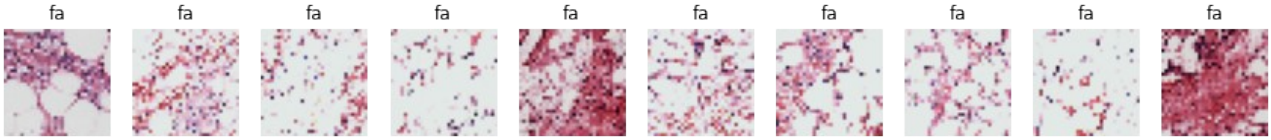

Fig. 5: Samples of class fat

Fig. 6: Samples of class glass

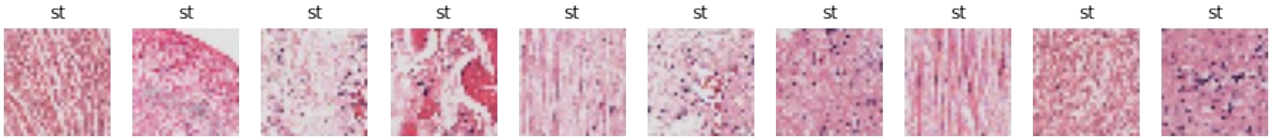

Fig. 7: Samples of class stroma

Class incremental learning

The scenario where the visual object classification system is able to incrementally learn new classes as training data for them becomes available is called class-incremental learning.

The main concept we are going to address in this process is how can our model learn continuously and prevent the catastrophic forgetting that convolution networks tend to do after adding new training data.

In each step by feeding our model with new training data, it should be able to remember the previous knowledge and expand it to the new data as well (new class).

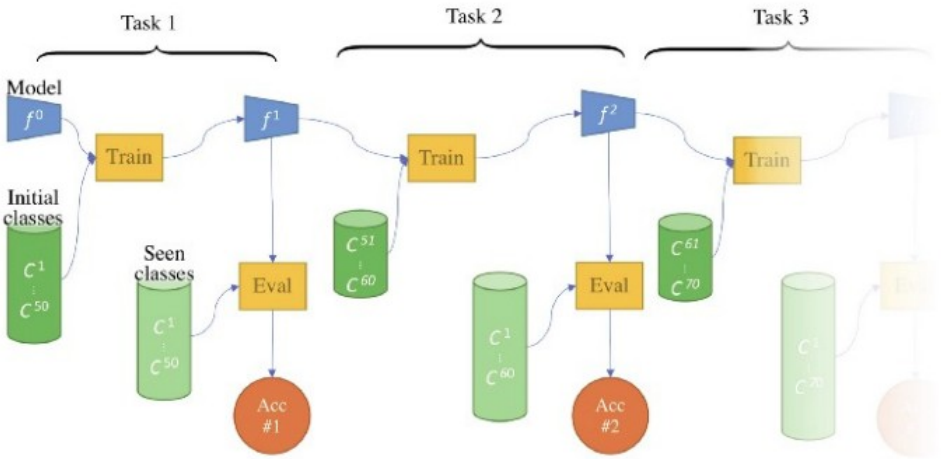

Fig. 8: Incremental learning

As it can be seen on Fig.8, we start by training the model on the initial classes and we evaluate its performance. In the next phase we feed the network with the samples from the new classes (in our case we only add 1 new class per time) and we want to perform the training just on that part and keep our previously learned knowledge.

The main challenge in this approach is how to find a trade-off between rigidity (being good on old tasks) and plasticity (being good on new tasks).

Exemplar sets

A common technique used in many incremental learning rehearsal approaches in literature is to use sets of *exemplar* images that are selected dynamically from the available dataset. For each observed class there is such an exemplar set and the total number of exemplar images shouldn't exceed a fixed parameter *K*. The whole set including all single exemplar sets of each class is called 'buffer'.

Depending of how many classes are available we split this parameter on *n_classes* parts so that each class has the same number of exemplar images *m=(K/n_classes)* present in the buffer at each step. *n_classes* refers to the number of classes seen so far up to the current step.

After each training step we rebuild the new buffer by randomly choosing *m* samples from the current new class and the *m* first samples from each single previous exemplar set.

In the first step where we have only samples belonging to the first class and only one output neuron, the whole exemplar set is constructed from the same class. In all the other steps, the exemplars for the previous classes are picked from the exemplar set available at the beginning of the step, for each class, and then they are expanded with exemplars from the new class. In this way at each step we get *m=(K/n_classes)* samples per class in the buffer.

Regarding the choice of *K*, also called 'Memory', we did choose a size of 2000, as it is the standard in most of the rehearsal methods among those which use a buffer that can be found in literature.

Training process

The class samples are processed in batches for a given number of training epochs.

The way we implement the incremental technique is the following:

Initial step (number of classes = 1):

1. Record the network's output predictions
2. Compute the classification loss and back-propagate the error
3. Repeat steps 1-2 for all batches of the samples

Successive steps (number of classes > 1):

1. Save a copy of the old network
2. Update the network by adding one more output neuron
3. Record the network's output predictions
4. Compute total loss (described later) and back-propagate the error
5. Repeat steps 3-4 for all batches of the samples

After each of these steps the new exemplars for the current new class are chosen randomly among all the training samples available of that class, and therefore the number of exemplar images for each class is reduced to *m=(K/n_classes)*, so depending on the number of classes available for the current step.

We also test our current model on the test sets for the available classes seen so far (*n_classes*) to assess the model performance on the held-out samples which were totally unseen during training.

The update of the network by adding one more output neuron on the classification layer is done in such a way that first we save a copy of all the parameters of the classification layer learned so far. After that we re-initialize the classification layer as a new fully connected layer with output dimension increased by one and we copy the previously learned available parameters back into the newly initialized classification layer.

Loss function

After each epoch the loss is computed given a loss function that depends on the number of classes available. For the initial step when we have only 1 class the loss is computed based on the current outputs that the network predicts for the samples and their true labels.

For all the successive steps we take into consideration also the success of the previous network on the input samples of the current class plus the old class samples within the buffer, and how different it is from what the current network predicts.

The total loss of each epoch (after the initial phase with 1 class only) actually consists of two terms: *classification* and *distillation* terms.

The classification term is basically estimating how far is the output from the target, done by a chosen criterion.

We have chosen the Binary Cross Entropy loss criterion (*BCEWithLogitsLoss*), which combines a *Sigmoid* layer and the *Cross entropy loss*, as it can be seen below:

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

$$CE = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

$$f(s_i) = \frac{1}{1 + e^{-s_i}} \tag{1}$$

Note: In order to properly use the Binary Cross Entropy Loss, we firstly transformed our ground truth labels to one hot labels.

For the distillation term instead, we take into consideration the old outputs (the old networks outputs for the samples) and the current outputs (the current networks outputs for the samples restricted to the old *n_classes-1* neurons) on those samples. We decided to use the Mean Squared Error loss (2) criterion for this term, and before adding it to the sum for the final loss we multiply it by the parameter *lambda.* This parameter represents how much we want to penalize our network for the difference between the old and new outputs on the same samples (distillation term).

$$\ell(x, y) = L = l_1, \ldots, l_N^{\top}, l_n = (x_n - y_n)^2 \tag{2}$$

Actually, before passing to the Mean Squared Loss function both the old and the new network's outputs, we apply the sigmoid function to both these output values. This is done because, otherwise, if the old and the new network's outputs differ much, the huge difference would be squared and we would obtain a big error value. Such big error could potentially affect negatively the training procedure, hence we apply the sigmoid to limit the values into a [0,1] range in a way the maximum difference is more contained.

Finally, the total loss looks like this:

$$loss = loss_{classification} + loss_{distill} * lambda \tag{3}$$

Actually, we found out that, in our case, keeping *lambda* as a constant as the incremental learning proceeds class by class is not enough to sufficiently penalize the network on the distillation part. Hence, to encourage the network at 'remembering' from the old one as the number of classes increases, we added a 'delta' value to *lambda* at each step:

$$loss = loss_{classification} + loss_{distill} * \left(lambda + n_{classes} * delta \right) \tag{4}$$

Architecture: ResNet

A residual neural network (a.k.a. Resnet) is a particular type of artificial neural network characterized by the addition of skip connections, or shortcuts to jump over some layer.

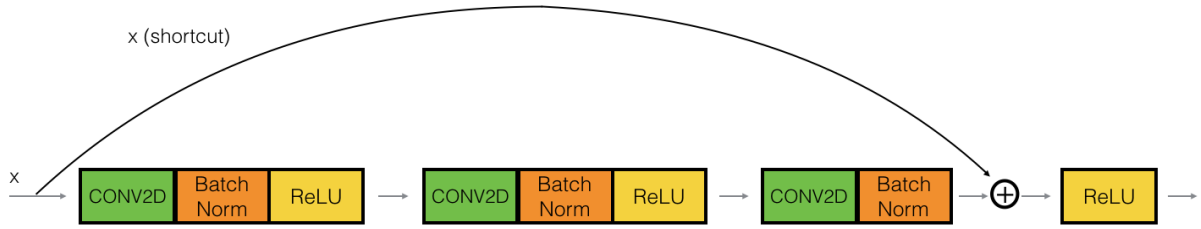

Fig. 9: ResNet : Skip connection

ResNet aims to solve the vanishing gradient problem which arises when training deep neural networks with many layers: as the gradient of the loss is back-propagated to previous layers, repeated multiplication might make the gradient infinitively small. Therefore, the weights of the first layers will either update very slowly or remain the same, such that the initial layers won't learn effectively. Hence, as we keep adding layers, the performance of the network gets saturated and eventually starts degrading.

What the ResNet architecture actually introduces for preserving the gradient are the skip connections, also called identity shortcut connections: connections which allow the flow of information from earlier layers in the network to later layers, actually bypassing two or three layers of the network.

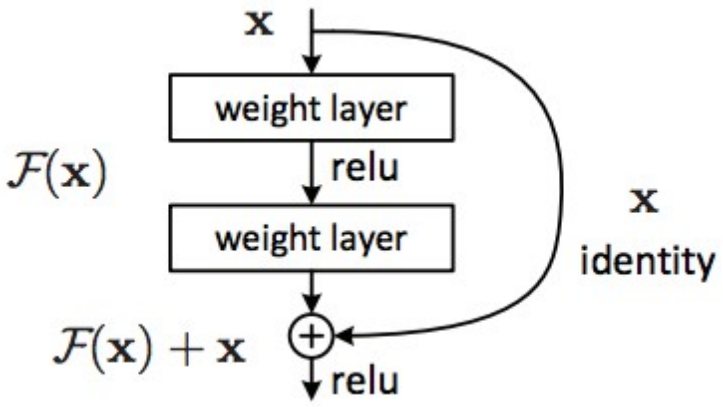

Fig. 10: ResNet : Residual block

The convolutional layers are stacked together like in a traditional CNN, but they are now added as residual blocks, which contain what is normally found within two (or three) layers. Each residual block has an identity mapping, so the original input *x* is added to the output of the last layer. This serves to help the network at calculating the optimal mapping function for each block. In fact, if an identity mapping was optimal, the weights of the layers can be trivially driven toward zero to approach an identity mapping. In such a case the network would then behave like a shallower

counterpart without that block, as an identity mapping would simply transmit forward the same previous input data. In this way stacking layers shouldn't degrade performances anymore, as by stacking identity mappings (so layers that essentially do nothing), the resulting architecture would perform the same, and so a deeper model would not produce a higher training error.

About the vanishing gradient problem, skip connections allow an alternate shortcut path for the gradient to flow through during back-propagation. If the gradient is back-propagated through the identity function, it would be simply multiplied by one so it will not be changed.

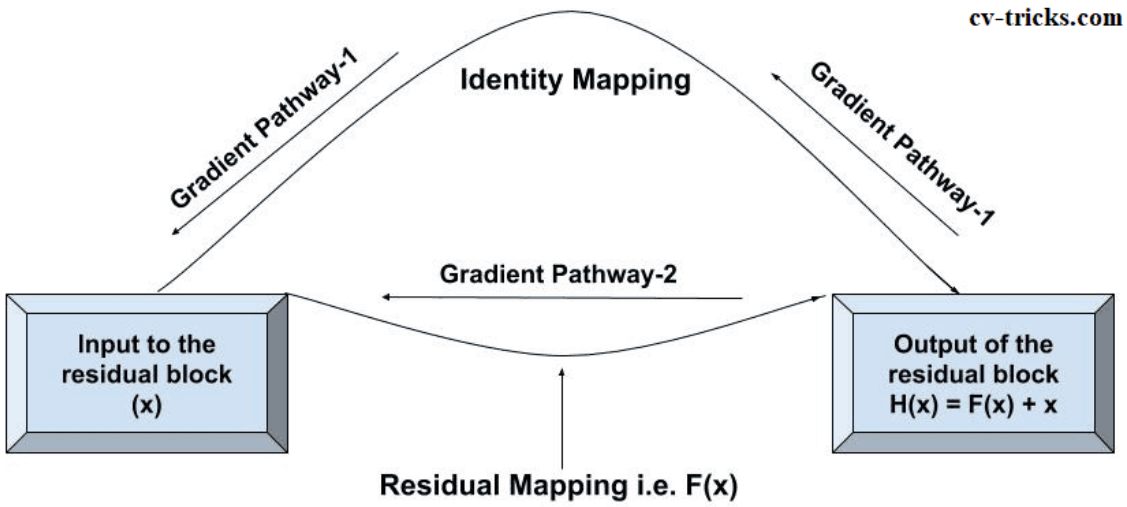

Fig. 11: ResNet : Alternative path for the gradient

To save the gradient from vanishing in the initial layers, it can directly pass through the skip connection. The residual block(s) will be skipped at once and the gradient can reach the first layers such that they can learn the correct weights.

Network choice: ResNet32

Among the ResNet family, we chose to use the 32-layer version, as it already proved to be good in literature when used in the incremental learning context, in particular in the class-incremental learning challenges on the CIFAR10 and CIFAR100 datasets. Hence, we use one pre-trained on CIFAR100, as the lower-level features should be general enough to be able to guarantee better performances than starting from scratch.

Fine tuning

Instructions note: to execute the experiment with the fine tuning setting instead of the default incremental learning setting, change the global variable FINETUNE to True before re-executing the notebook by pressing 'Restart and run all'.

In order to verify the effectiveness of the incremental learning approach in comparison to fine tuning, we will use the complete dataset with all the classes present in it. We do that by concatenating all the training class samples in one dataset and we train the Resnet32 model using it.

The difference in the training process is that we won't use exemplar sets since we need all the class' samples present in one dataset. Therefore the training process finishes after all the epochs have been iterated.

The loss function to be used consists of a simple classification term which shows us how close is the network's output to the target. The criterion that is used here is the same as in the previous step, *BCEWithLogitsLoss*, to have a fair comparison between the two methods.

The obtained results as well as the comparison between the incremental learning and fine tuning approach can be seen in the following section.

Results and comparison

After an extensive grid search of the different hyper-parameters that are used in the scope of our project, we have come up with the decision to use the following parameters:

BATCH_SIZE = 128
LR = 1
MOMENTUM = 0.9
WEIGHT_DECAY = 1e-4
EPOCHS = 70
MEMORY = 2000 (this is the capacity of the buffer we are using)
LAMBDA = 0.4 (hyper-parameter controlling the weight of the distillation)
DELTA = 0.5 (value added to lambda at each incremental step)

We start with a rather large learning rate (LR) as at the beginning, after the weights are initialized, we are far from the optimal solution. Then, we divide the initial LR by a factor of 5 twice, respectively at 49 and 63 epochs (7/10 and 9/10 of the training process). This is done as, towards the end of the training, we want to fine-grain the search of the optimal weights.

For the class criterion we decided to use the *BCEWithLogitsLoss* which showed as the most successful for our task.

For the distillation criterion on the other hand we will be using the *MSELoss* criterion.

The optimizer implements the *Stochastic Gradient Descent* (SGD), to which we pass our network's parameters as well as the other necessary parameters (learning rate, momentum, weight decay).

Next we will report some of the results we got for the different scenarios:

1. ResNet32 pre-trained model on CIFAR100, using exemplars
2. Resnet32 model with funetuning

ResNet32 pre-trained on CIFAR100

In order to guarantee better results, as mentioned before we used a pre-trained ResNet32 model on CIFAR100 dataset. Given the fact that the low level features obtained by training the model on this dataset should be useful also in our scenario, it is expected that starting with a pre-trained model will result in better performance.

After loading the weights of the ResNet32 model pre-trained on the available dataset (CIFAR100), we replace the fully connected (FC) layer with a new Linear layer which has as input dimension the same dimension of the previous FC layer, and as output dimension the number of current classes, starting from 1.

Below we can take a look of the training loss during epochs, as the number of classes increases.

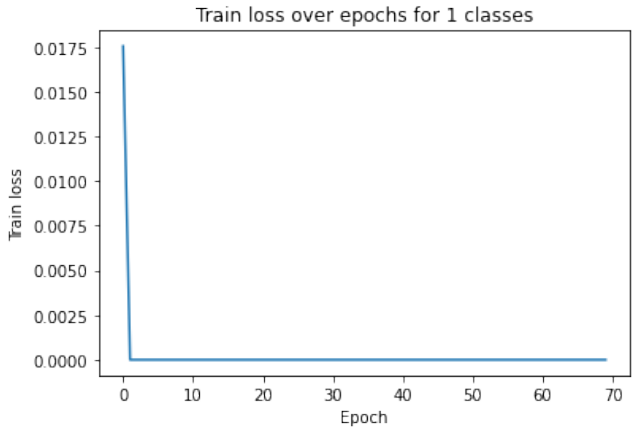

Fig. 12: ResNet32 pre-trained, loss for n_classes=1

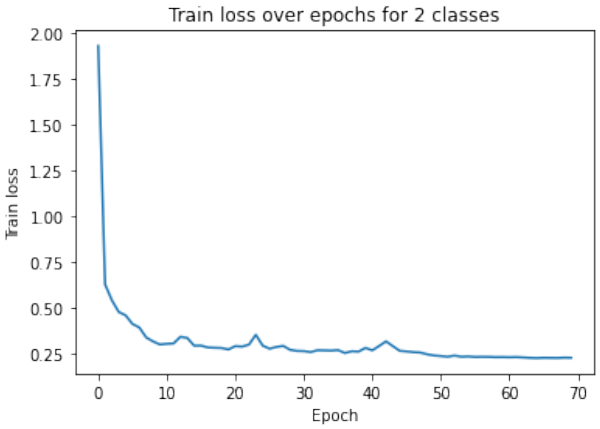

Fig. 13: ResNet32 pre-trained, loss for n_classes=2

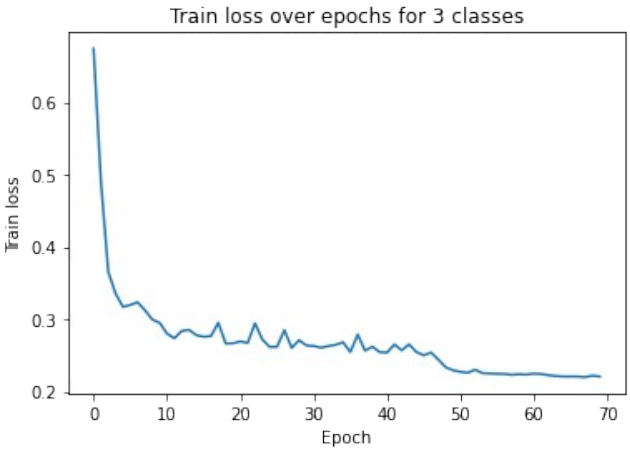

Fig. 15: ResNet32 pre-trained, loss for n_classes=3

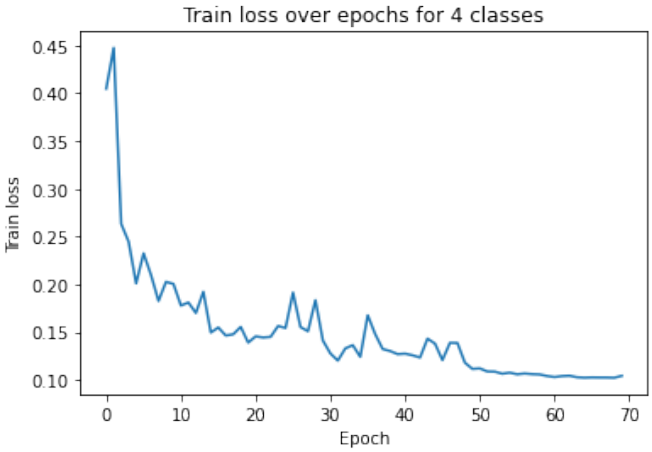

Fig. 14: ResNet32 pre-trained, loss for n_classes=4

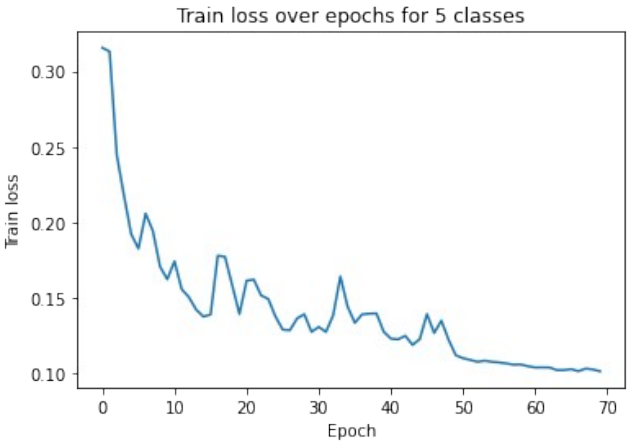

Fig. 16: ResNet32 pre-trained, loss for n_classes=5

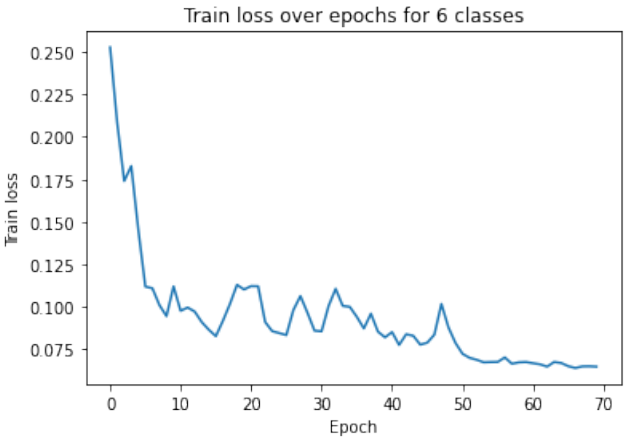

Fig. 17: ResNet32 pre-trained, loss for n_classes=6

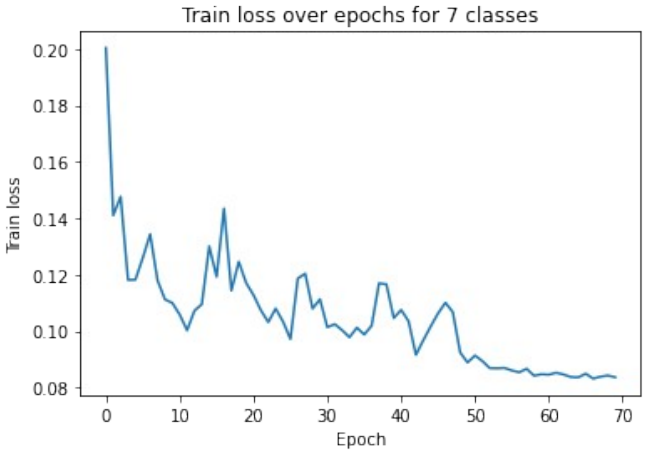

Fig. 18: ResNet32 pre-trained, loss for n_classes=7

From the previous plots it can be seen that in the first steps the loss convergence happens right after the start of the training process, while in the later steps it happens only after lowering the learning rate (epoch 49). In case more classes were present, the training process would likely need more epochs to reach this loss convergence and the lowering of the learning rate should be postponed to a later epoch.

After the training for the batch of classes has been completed, a testing is done for all the current classes that are present, taking as samples the samples which are in the test datasets. After a test accuracy is obtained for each of these classes, the average of these accuracies is computed and saved.

The change of the average test accuracy during the incremental steps of the training can be seen below:

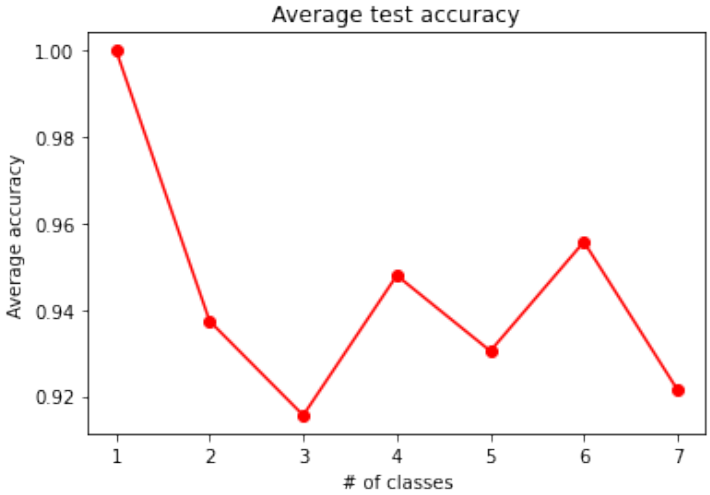

Fig. 19: Average test accuracy during incremental learning

There are several thing to note from Fig.19. Firstly as always in the presence of only one class (H) the model only predicts that class so the accuracy is always 100%.

In the next two steps when there are 2 and 3 classes present the accuracy decreases to about 92%. In the next steps the accuracy reaches a maximum of ~96% when 6 classes are present.

The reason why we have peaks in the accuracy again in the step 4 and 6 is because, at that moment the samples from the classes ‘blood’ and ‘fat’ are added, they are much more easier to be differentiated from the others. For this reason the network is less likely prone to confuse those samples with the samples from other classes, resulting in a higher average accuracy during those steps.

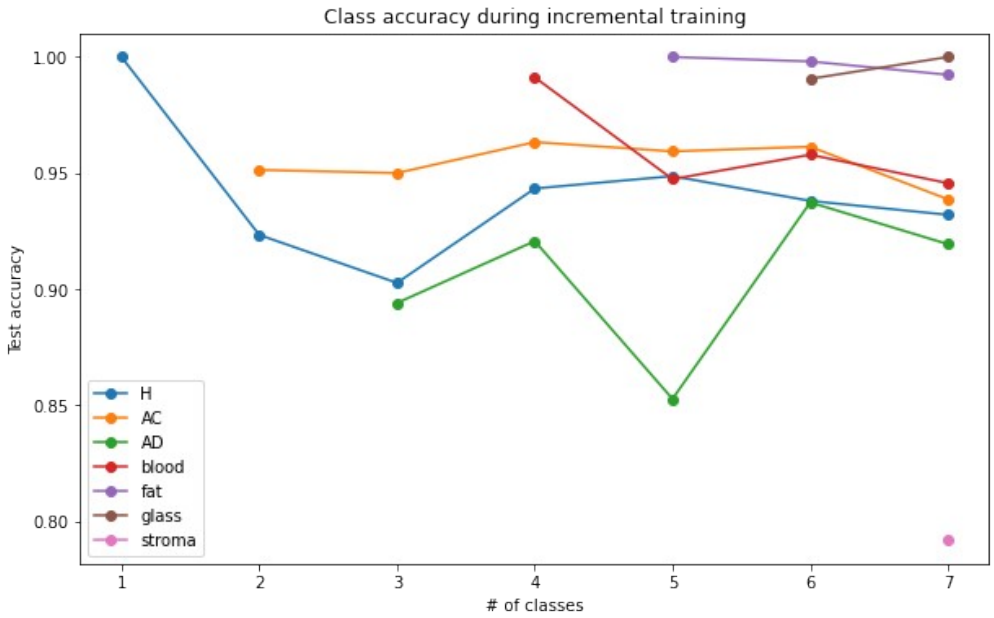

Fig. 20: Class accuracy during incremental learning

Fig.20 on the other hand gives us a better overview of how the accuracy of each class varies during the process of incremental learning.

In order to show that this approach works even when the order of the classes changes, we ran the same experiment for different class orders. Some of them presented an oscillating behavior of the average accuracy when adding new classes, while others had a smoother decrease in accuracy.

An example of such a smoother behavior is the following alternative class order:
H, stroma, fat, AC, glass, blood, AD

Instructions note: to execute this alternative order of class instead of the default class order, change the global variable DEFAULT to False and then press ‘restart and run all’.

Below we can take a look of the smoother decrease of the average test accuracy as new classes are being added:

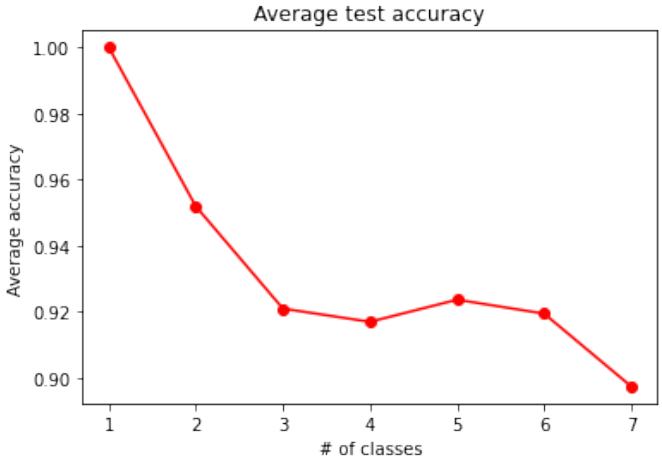

Fig. 21: Alternative order: average test accuracy

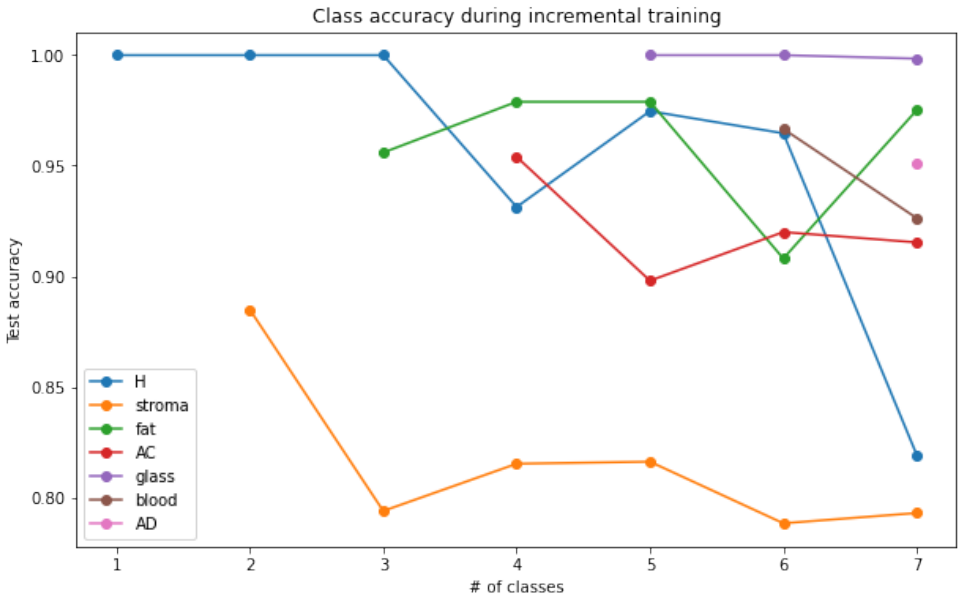

Fig. 22: Alternative order: class accuracy during incremental learning

And on Fig.22 is presented the change of each class' accuracy during the incremental training with this new alternative class order.

We can notice that, as the class order changes, also the class' accuracy is changed accordingly. Therefore the class order in the incremental learning approach plays an important role. In particular, if we introduce the non-spurious classes (H, AC, AD) later on during the incremental learning process, they are more likely to be confused by the network with the other classes.

That's both because the real classes are more visually complex and because, as we proceed in the incremental learning process, the saved samples in the buffer are less, thus the encouraging factor provided by the rehearsal is much weaker in the last steps.

ResNet32 model with fine tuning

Lastly, we take a look of the obtained results during the fine tuning technique. As mentioned previously, we train a ResNet32 model on the full dataset.

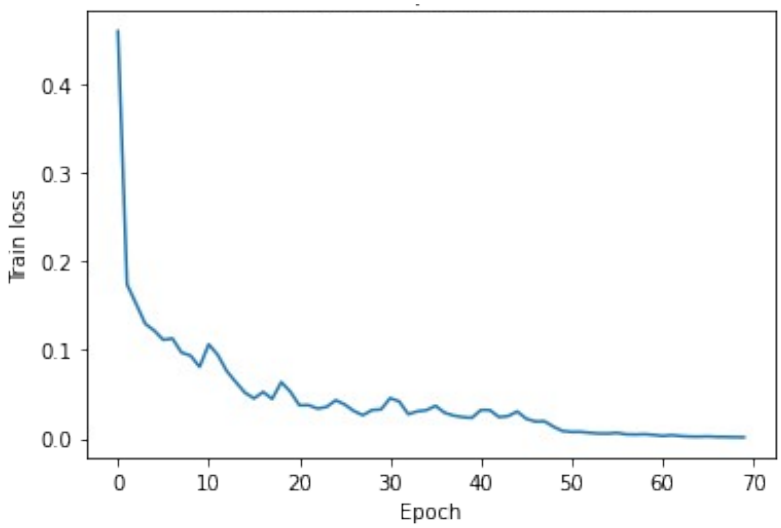

Fig. 23: ResNet with fine tuning, loss over epochs

As it is shown on Fig.21, the loss function decreases steadily over the epochs, with a final value of 0.0016 reached at the last epoch.

After evaluating the model on the test datasets, the average test accuracy of all the classes is about 87%. Here's a snapshot of such result:

```
### Test accuracy for H: 0.94
    Test accuracy for AC: 0.88
    Test accuracy for AD: 0.76
    Test accuracy for blood: 0.92
    Test accuracy for fat: 0.98
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.79
# Total Accuracy: 0.87
```

```
### Test accuracy for H: 0.93
    Test accuracy for AC: 0.94
    Test accuracy for AD: 0.92
    Test accuracy for blood: 0.95
    Test accuracy for fat: 0.99
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.79
# Total Accuracy: 0.92
```

Fine tuning (on the left) compared to Incremental learning default order (on the right)

As it can be seen from the comparison of the last training step of incremental learning method and fine tuning one, we managed to have an increase (or at least perform the same) in the accuracies of the first 4 classes (H, AC, AD, blood), while the remaining 3 performed almost the same.

That was what we were aiming for: the first classes are learned better as we enforce the network to remember the previous classes learned so far, thus avoiding, on average, a decrease in accuracy as much as possible as new classes are introduced. Also, occasionally, we have an increase of the old class' accuracies in the next incremental steps, thanks to the rehearsal effect of the buffer.

As a final remark, in this case we had a remarkable increase of 5% in the performance, but on average, when trying out different class orders or GPUs, the increase of the incremental learning approach was about 2%.

```
### Test accuracy for H: 0.94
    Test accuracy for AC: 0.88
    Test accuracy for AD: 0.76
    Test accuracy for blood: 0.92
    Test accuracy for fat: 0.98
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.79
# Total Accuracy: 0.87
```

```
### Test accuracy for H: 0.82
    Test accuracy for stroma: 0.79
    Test accuracy for fat: 0.98
    Test accuracy for AC: 0.92
    Test accuracy for glass: 1.0
    Test accuracy for blood: 0.93
    Test accuracy for AD: 0.95
# Total Accuracy: 0.9
```

Fine tuning (on the left) compared to Incremental learning alternative order (on the right)

Also for the alternative class order, apart from the class ‘H’, all the other classes have better scores in the Incremental learning scenario in comparison to the fine tuning one.

The most significant drop in the accuracy of the class ‘H’ happens in the last step of the process when the class ‘AD’ is introduced. This is due to the fact that we have fewer samples of each class (only 285 for class H) in the buffer at this step, and these two classes share some same visual patterns. In this case the distillation factor is not able alone to preserve the previous knowledge of the class ‘H’, but by examining the trend of our incremental approach, also in the other tries and class order combinations, we would potentially expect an increase of ‘H’ again if further incremental steps are added.

However, we could not avoid this drop on the ‘H’ class, as if we tailored the *lambda* value to a bigger value for covering this drop, the overall performance would have decreased. In fact, as we started in the beginning, our global aim is finding the best performance for both the old (rigidity) and the new (plasticity) classes.

Knowledge distillation

Knowledge distillation is a concept that is based on the idea that a small network (student model) can be taught step by step what to do using a bigger already trained network (teacher model).

This technique trains the student network to replicate the behavior of the more complex teacher network.

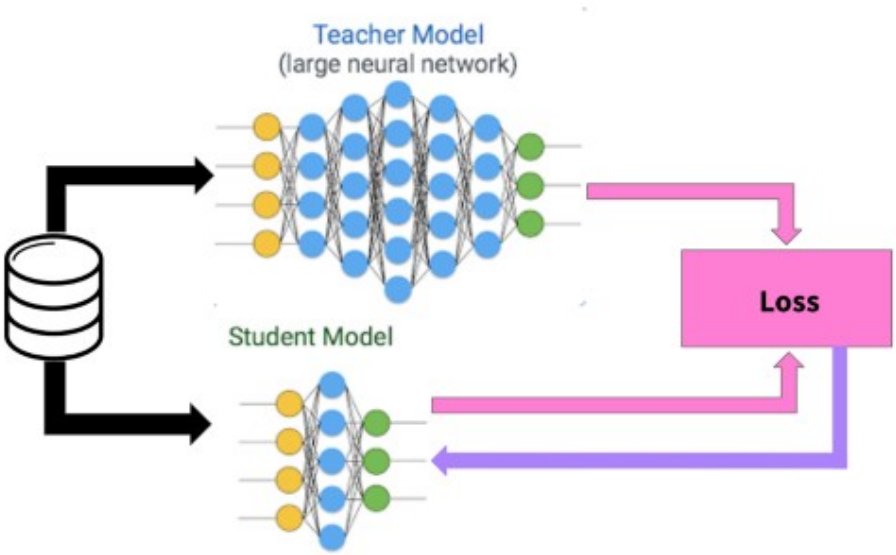

Fig. 24: Knowledge distillation process

The way it's done is that the complex teacher network is firstly trained on the complete dataset and learns the more complex features, which is why this step can take longer to execute or require higher computational performance.

After that, we want to transfer this knowledge to a smaller, simpler model, so that the amount of computation is minimized and so is the memory which is used.

KD training steps

Firstly we load the previously trained class-incremental model from the previous steps. We will refer to this model as the teacher model. The teacher model has a complex Resnet32 architecture with 7 output classes.

For the student network we will be using a simple 5 layer architecture, with 2 convolution layers and 3 fully connected layers. The output dimension of the last FC layer is 7.

The training set to be used by the student network is the full train set (containing all the samples).

The training process is the following:

1. Compute the output of the both teacher and student networks for the current image samples
2. Compute the loss and back-propagate it
3. Repeat steps 1-2 for all batches of the samples

This process is repeated a given number *NUM_EPOCHS*.

Loss function

The loss term that has been mentioned in the training process, takes into consideration two terms: the classification and knowledge distillation term.

Classification loss is the calculated estimation of how far the student network's output is from the ground truth labels for the given image. This is done by a chosen criterion, in our case a *BCEWithLogitsLos*s (chosen again for having a more fair comparison with respect to the teacher network),which has been described in point (1).

Knowledge distillation loss on the other hand computes the difference between the student network's output and the recorded output from the trained teacher network, using as criterion the Mean Squared Error Loss. This term is multiplied by a defined parameter *kd_lambda*.

The final loss that will be back-propagated is the sum of the two losses:

$$loss = loss_{classification} + loss_{kd\ distill} * lambda_{kd} \quad (5)$$

Instructions note: to later re-execute this part of the experiment without the distillation term, set the global variable DISTILL to False and press 'run after' when the 'Knowledge Distillation' cell is selected. In this way the total loss will be equal to the classification loss only.

Results

The parameters to be used for the process of knowledge distillation are the following:

KD_EPOCHS = 50
KD_LR = 0.1
KD_LAMBDA = 0.03

KD starting from ResNet32 pre-trained model

The loss function during the training process of the student network over epochs is shown below:

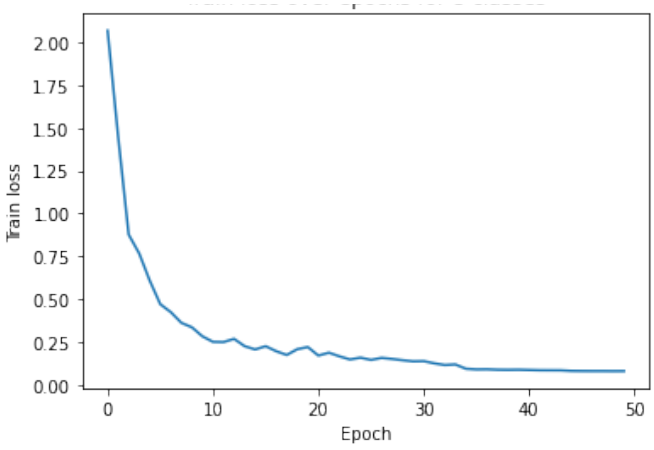

Fig. 25: Knowledge distillation loss over epochs

After the training process of the student network is completed, the average test accuracy is 86%.

If we run the same experiment, starting from the same teacher model but setting the DISTILL variable to False, the loss will consist only of the classification term, as previously mentioned. The

average test accuracy after such an experiment is about 77%, which is much lower than the previous result where the loss also included the distillation term.

```
### Test accuracy for H: 0.9
    Test accuracy for AC: 0.71
    Test accuracy for AD: 0.88
    Test accuracy for blood: 0.95
    Test accuracy for fat: 0.95
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.78
# Total Accuracy: 0.86
```

```
### Test accuracy for H: 0.49
    Test accuracy for AC: 0.74
    Test accuracy for AD: 0.85
    Test accuracy for blood: 0.95
    Test accuracy for fat: 0.92
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.75
# Total Accuracy: 0.77
```

Incremental learning comparison: DISTILL = True (on the left) and DISTILL = False (on the right)

With this we demonstrate that, in this scenario, the knowledge distillation from the teacher network helps out the smaller student network to output the right predictions.

KD starting from fine tuning

In case we perform knowledge distillation after having performed fine tuning of our previous network, the behavior of the loss looks like this:

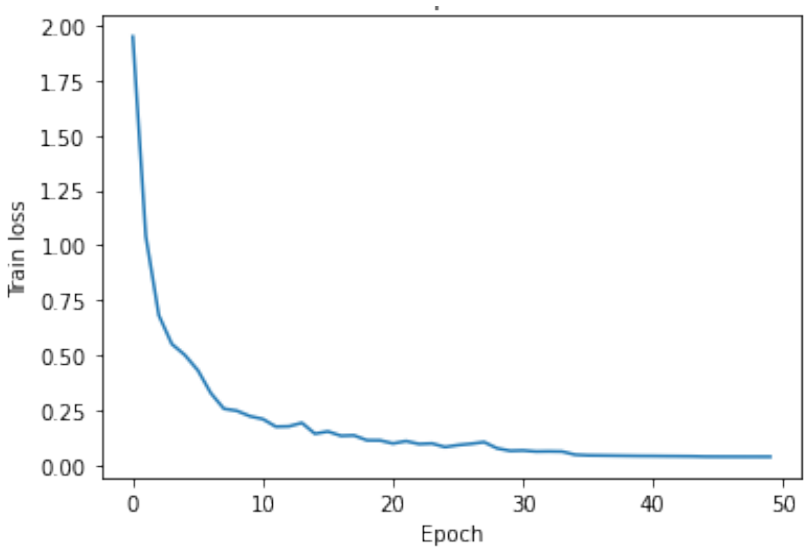

Fig. 26: Knowledge distillation loss after fine tuning

```
### Test accuracy for H: 0.77
    Test accuracy for AC: 0.79
    Test accuracy for AD: 0.88
    Test accuracy for blood: 0.94
    Test accuracy for fat: 0.95
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.84
# Total Accuracy: 0.85
```

```
### Test accuracy for H: 0.49
    Test accuracy for AC: 0.74
    Test accuracy for AD: 0.85
    Test accuracy for blood: 0.95
    Test accuracy for fat: 0.92
    Test accuracy for glass: 1.0
    Test accuracy for stroma: 0.75
# Total Accuracy: 0.77
```

Fine tuning comparison : DISTILL = True (on the left) and DISTILL = False (on the right)

The total test accuracy after this process is about 85%. If instead we omit the distillation term of the loss, the final accuracy drops to 77%. So, again, we have an increase of 8% when adding the distillation term to the total loss.

Hence, knowledge distillation proved to positively affect a smaller network performance both in the incremental learning and fine tuning scenario.

Conclusion

From this experiment we verified that our incremental learning technique performed slightly better than fine tuning. This was thanks to the fact that we both used a ‘buffer’ to encourage the network to remember the old classes, and an additional loss term, which pushes the network at not forgetting the knowledge embedded in the previous steps.

Achieving consistent and good results in the incremental learning scenario is not an easy task. Doing it only with replay or only with distillation doesn’t achieve the best results. But, by combining these two, we were able to preserve the knowledge of the network from the previous steps and expand it to the new classes.

By applying the same ‘distillation’ reasoning we can enforce a small (student) network to mimic the behavior of a larger (teacher) trained network. Through penalizing the student network when its soft outputs differentiate from the teacher’s, we incentivize the student network to perform similarly (and, so, better) to the teacher network.

We demonstrated that our approach works under different settings, and we achieved good results.