EE-559 Project 1 – Classification, weight sharing, auxiliary losses

Alexandre Reynaud (258402), Victor Taburet (261845), Vincent Rinaldi (239759)

I. Introduction

The objective of this project is to test different deep network architectures which compare 1000 pairs of two digits, each pair is visible in a two-channel image (a $2 \times 14 \times 14$ tensor, corresponding to a pair of 14×14 grayscale images), and predict for each pair if the first digit is lesser or equal to the second. The idea of using different architectures is to assess the performance improvement that can be achieved through ***weight sharing***, or using ***auxiliary losses***.

II. Implementation of the algorithm

The project processing was exclusively handled using the *PyTorch* library. The *Matplotlib* library is used to visualize our results. The file *dlc_practical_prologue* provided by the teaching team is used to generate our data sets. The file was slightly modified with the addition of a flag *trainall* to the parser in order to add the possibility of choosing between training all of our architectures, or only our best one. Those data sets are used to train our architectures, and evaluate their test error rate.

We decided to experiment on 4 different architectures and analyse their behavior over 20 instances (rounds). A seed was set to make our results reproducible. For each round, we generate new data sets for training and testing. A 2D tensor of dimension 4×20 (named *test_errors*) gathers every recorded test error rate on each model at each round. After setting our chosen hyperparameters, we launch the training. At each round, and for each model, we call the function *get_stats* which trains the given model on the training data sets at our disposal with the *train_model* function, and return the test error rate computed using the testing data set thanks to the *compute_nb_errors* function.

At the end of the 20 rounds, we build a graph to summarize all the recorded error rates for each architecture. We also display the mean and standard deviation of error rates estimated on 20 rounds for each model to visualize the improvement obtained from the addition of weight sharing and/or the use of auxiliary losses.

III. Deep network architectures

As mentioned in the previous part, we are comparing 4 different neural network architectures to assess the performance improvement when applying weight sharing and using auxiliary losses.

- ***NetBase:*** The first one is called *NetBase* and is our starting point, that we want to improve. The network is composed of two separate branches that receive each as input a $1 \times 14 \times 14$ tensor (representing the grayscale image of one of the two numbers of a pair) such that, when processing a pair of grayscale images, each number of this pair goes to a different branch. On both branches, we have a convolutional layer of size $32 \times 3 \times 3$ (32 filters with kernel dimension 3×3) followed by a pooling layer of type *Max Pooling* with a filter of kernel dimension 2×2, and stride argument set to 2. After that we again have a convolutional layer, but now of size $64 \times 3 \times 3$, also followed by a *Max Pooling* layer of kernel dimension 2×2, and stride set to 2. We then have two consecutive fully connected layers, the first one with 128 hidden units, and the second one with 10 of them (these are outputs in the case of auxiliary losses, but we will come back to it when describing the auxiliary loss process). The two branches are then merged (we simply concatenate the two 1D tensors of size 10 together in the same order as the images were, meaning that the 10 values obtained from the first image of the pair are the first 10 values of the resulting tensor of dimension 20, whereas the last 10 values of the resulting concatenation are those obtained from the second image of the pair). The 1D tensor of size 20 then goes through two fully connected layers, the first having 128 hidden units, and the second one outputting 2 neurons that will be passed to our chosen loss function. Note that we always apply the *ReLu* activation function on the output of each of our pooling and fully connected layers (except the very last fully connected layer outputting two neurons). Since we are using the *Cross Entropy* as our loss function, we don't need to add a *Softmax* layer at the very end.

- ***NetWS:*** Our second architecture is called *NetWS* and is exactly like *NetBase*, except that now the two branches share the same weights layer-wise. For example, the first convolutional layers of each branch have the same filters. Therefore, we obtain the structure of a *Siamese* network. The layers that come after the merge of the two branches stay unchanged.

- ***NetAL:*** The third architecture is *NetAL* and introduces *auxiliary losses*. Again, it is the same as *NetBase*, but we now treat the 10 output neurons of the last fully connected layer of each branch as auxiliary outputs, from which we compute auxiliary losses. The two auxiliary losses evaluate the digit recognition accuracy of the two images embedded in the input. Determining the value of a digit is useful to solve our main task since we are looking for the digit having the greatest value in a given pair. The two values of the auxiliary losses are then added to the one of the main loss in order to obtain the general loss of the network. Note that we don't apply the *ReLu* activation function before retrieving the auxiliary outputs, but after, just before merging the two branches.

- ***NetWSAL:*** The last architecture is *NetWSAL* (See Figure 1) and is simply *NetWS* and *NetAL* merged together : both improvements, *weight sharing* and *auxiliary losses*, are present, and implemented in the same way as in *NetWS* and *NetAL* respectively.

Our losses are computed using the *Cross Entropy* function. The main loss applies the loss function between the main output and the training targets (each target being 0 or 1 depending on whether the first digit is higher than the second one, or not), The auxiliary losses, on the other hand, perform the computation between the auxiliary outputs and the training class labels (each label being the corresponding digit).

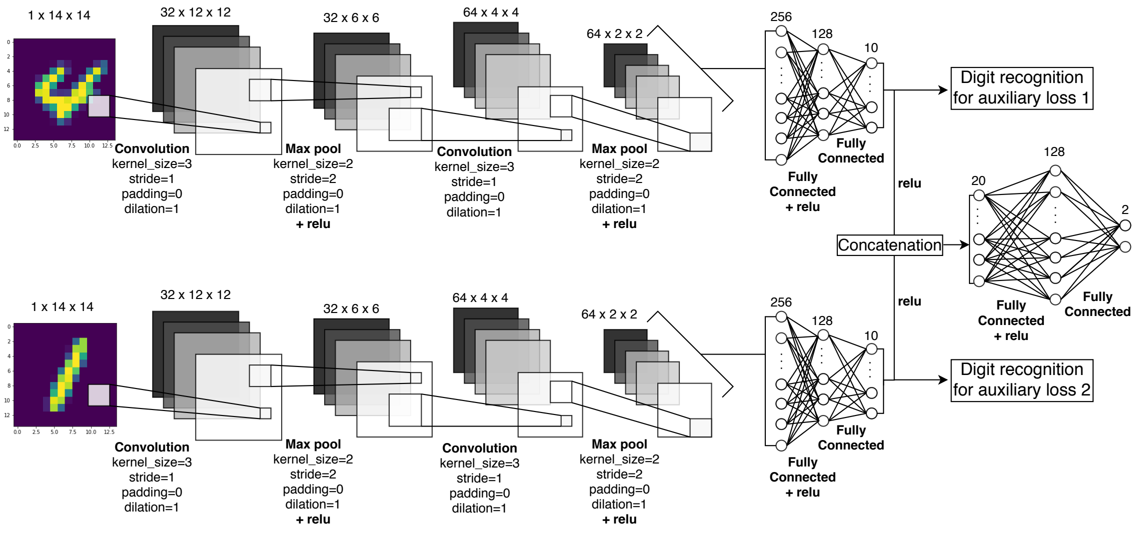

1 x 14 x 14
32 x 12 x 12
32 x 6 x 6
64 x 4 x 4
64 x 2 x 2
256
128
10
Digit recognition for auxiliary loss 1
Convolution kernel_size=3 stride=1 padding=0 dilation=1
Max pool kernel_size=2 stride=2 padding=0 dilation=1 + relu
Convolution kernel_size=3 stride=1 padding=0 dilation=1
Max pool kernel_size=2 stride=2 padding=0 dilation=1 + relu
Fully Connected + relu
Fully Connected
relu
Concatenation
20
128
2
Fully Connected + relu
Fully Connected
relu
Digit recognition for auxiliary loss 2

Figure 1: Architecture of NetWSAL, implementing both weight sharing and auxiliary losses, and achieving the best accuracy. The part with the two parallel branches is, thus, a siamese network, where the two branches share the same weights.

IV. Hyperparameters

Before starting computation, the choice of the hyperparameters remains. We distinguish five different hyperparameters : the *batch size*, the *number of training epochs*, the *optimizer*, the *learning rate*, and the *coefficient of the auxiliary losses*.

We chose to perform a *Grid Search* to select our hyperparameters. However, due to their high number and the range of values that should be tested to perform an efficient search, we decided to set a fixed value for the *batch size*, the *number of epochs*, and the *coefficient of the auxiliary losses*, since the goal of the project is not to obtain the best possible results, but only to assess performance improvement of the two concerned techniques.

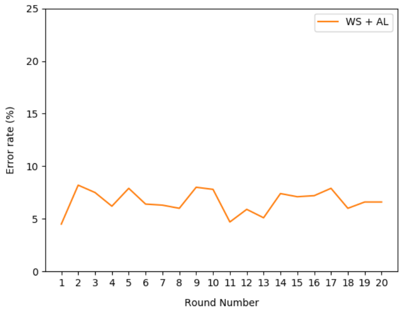
We decided to set the *batch size* to 25 as it was a good trade-off between running time and convergence rate per epoch. Considering this batch size, we chose to run 100 epochs to ensure that all models would have enough time to converge. The coefficients of the auxiliary losses were kept to 1 as this value already seemed to show the best results among tested values.

We then performed a *Grid Search* to choose the optimizer and its corresponding learning rate that is the most adapted for **all of our 4 models** in order to carry out an effective analysis and be able to make accurate comparisons on each model performance, so that we can bring a quality conclusion on the level of improvement of each technique.

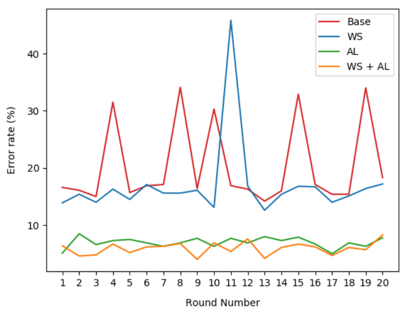
We ran our *Grid Search* on 5 different optimizers: *SGD*, *Adam*, *AdaDelta*, *AdaGrad*, *RmsProp*, with 3 different learning rates, being 0.001, 0.005 and 0.01, during 10 consecutive rounds for all 4 models. The output of the four *Grid Searches* can be seen in the notebook *Proj1_main_code_and_grid_searches* we provided.

According to our *Grid Search* results, even if *Adam*, with a learning rate of 0.001, does not present the best performances on *NetBase* and *NetWS* (recording a mean error rate of 18-20% due to the fact that it sometimes produces an unexpected high error rate of 35-45%), they are still acceptable. Furthermore, it records an amazing mean error rate of 5-6% on *NetAL* and *NetWSAL*, which is among the best results for those two models. We then decided to keep this optimizer and learning rate for our experiments, since they seem to be the best choice to fit efficiently all 4 models.

V. Results



(a) Test error rates on 20 rounds for the NetWSAL architecture.



(b) Test error rates on 20 rounds for all architectures.

Figure 2: Results obtained when running the algorithm on the **provided Virtual Machine (VM) of the course**. They may actually be slightly different on other machines, like ours, even after setting the same seed.

After running our algorithm **on the provided Virtual Machine** for 20 rounds only on the architecture *NetWSAL*, we obtain a mean error rate of 6.6649% which is a great performance for only 1000 training samples, and a standard deviation of 1.1022 (See Figure (a)). If we run every architectures on 20 rounds **on the provided VM**, we obtain, for respectively *NetBase*, *NetWS*, *NetAL* and *NetWSAL*, a mean error rate of 20.3099%, 16.9200%, 6.9800% and 5.9449%, and a standard deviation of 7.3464, 6.9292, 0.9023 and 1.1236 (See Figure (b)).

If we focus on the results obtained **on the VM**, we can see that the use of *auxiliary losses* is hugely efficient in our case, decreasing the mean error rate by about 10%. The implementation of *weight sharing* also brought a positive effect on the training process, since the mean error rate has also slightly decreased. Both techniques act as regularizers, since they restrict the solution space. We then expect the standard deviation of the estimator to be reduced when incorporating those techniques. This actually happens since the estimated standard deviation of *NetWS*, *NetAL* and *NetWSAL* are lower than the one for *NetBase*. One would expect *NetWSAL* to have the lowest recorded standard deviation, since it implements both techniques at the same time, but we have to keep in mind that our estimations are done over 20 rounds only and have therefore high variances. For example, another run (which output can be found in the provided notebook *Proj1_main_code_and_grid_searches*) performed **on our own machine** gets a mean error rate of 20.4999%, 16.9500%, 7.0099% and 5.9600%, and a standard deviation of 7.7486, 6.8812, 1.1125 and 0.9670 for each respective architecture.

Something intriguing in our results is the fact that the error rate sometimes suddenly reaches a very high value for a few number of rounds when training *NetBase* and *NetWS*. This is probably due to the chosen optimizer *Adam*, which is known for its instability. The problem seems to be mitigated by the introduction of auxiliary losses, as can be seen on Figure 2.

VI. Conclusion

The implementation of *weight sharing* and the use of *auxiliary losses* are regularization techniques. In this project, we were constrained to use a small data set, so regularization makes sense. This being said, we wanted to reduce the variance of the model as much as possible without increasing its bias too much. As first intuition, we thought that forcing a shared use of weights between our branches would do just that, as we could envision the model extracting similar information from both input digits and using it afterwards to predict which number is bigger. Similarly, adding auxiliary losses at the end of each branch looked like it would encourage the model to also recognize their respective input digit, making it maintain crucial information about them which would then facilitate their comparison. In addition to a large reduction of its variance, the model is indeed able to greatly improve, even with a relatively small number of samples at its disposal, supporting our reasoning. Nevertheless, one should not forget that the structure of the model, and the selected hyperparameters, also play an important role on the performance improvement brought by the regularization, since an unsuited choice may discard any viable configuration to begin with.