PROJECT 1 : CLASSIFICATION, WEIGHT SHARING, AUXILIARY LOSSES

EE-559 Deep Learning

Authors
OLIVEIRA Gabriel
JEANFAVRE Sébastien
RUIZ DE VELASCO Pablo

Course given by
Prof. François Fleuret

Spring semester 2021

1 Introduction

The first goal of this project was to compare different deep learning architectures on a specific task using the PyTorch library. The task consisted in taking as input a two-channel image representing two hand-written digits from the MNIST dataset (although they were resized from 28x28 to 14x14 to work with smaller images) and predicting if the first digit is lesser or equal to the second.
The second goal was to show the improvement in performance that can be achieved for a given architecture by using weight sharing and/or an auxiliary loss. Considering the usual case where an image (single channel) goes through several layers of a network, the idea behind weight sharing is that for a multi-channel image, if the processing of the different channels should be similar, then it makes sense to use the same layers (and therefore the same weights) for all the channels. Auxiliary losses are used to train a network while considering an additional loss function. This loss should allow the network to reach a certain intermediate result that will help the training task in the layers coming after. In the case of the given task, an auxiliary loss that accounts for the digit recognition can be used.

2 Architectures

We designed two architectures : a convolutional neural network named ConvNet, and a multi-layer perceptron named MLP.
For each image of the input, ConvNet has two blocks which are as follows : convolution 3x3, max-pool 2x2 and the ReLU activation function. The first convolution increases the number of channels from 1 to 32 and the second one from 32 to 64. The outputs of these blocks for each image are then concatenated and given to a series of two fully connected layers with 10 hidden units, which was found to provide better results than the other candidates tested, i.e. 100 or 200 hidden units.
The MLP has only fully connected layers, therefore the images were flattened to give them to the network as input vectors, which we could say is similar to considering every pixel in the image as a feature. Each image of the input goes through three fully connected layers. Then, similarly to ConvNet, the outputs of these three layers for each image are concatenated and given as input to the last two layers. The number of hidden units for the first and fourth layers were respectively set to 256 and 100, since they provided better results than the other values tested, i.e. 128 or 512 for the first layer, and 10 or 200 for the fourth layer.
To train both architectures, the Adam optmizer and the cross-entropy loss were selected.

3 10-Fold Cross Validation

Once we fixed the architectures, in order to optimize the training of the models certain hyperparameters such as the batch size, the number of epochs and the learning rate had to be determined. As the number of training samples was limited, to select the batch size and the number of epochs, we implemented the K-fold cross validation method with K=10 folds, a value that has empirically shown to provide good performance estimates. The learning rate however was not optimized and was left at the default of 0.001. Several rounds of cross validation were performed and all performance results were averaged. This procedure was implemented considering only the two original architectures, since we wanted to keep the same hyperparameters for the variants with weight sharing or an auxiliary loss so that the change in performance would only stem from these additional techniques. Table 1 shows the obtained results.

		Batch size		
Architecture	Epochs	25	50	100
ConvNet	25	2.12% ± 7.86% 2.29% ± 0.75%	3.63% ± 6.74% 2.53% ± 0.81%	5.96% ± 6.63% 2.45% ± 0.71%
	50	1.12% ± 6.28% 2.45% ± 0.85%	1.49% ± 6.57% 2.37% ± 0.8%	0.59% ± 4.66% 2.24% ± 0.65%
MLP	25	0.31% ± 0.73% 2.05% ± 0.46%	0.72% ± 2.08% 2.06% ± 0.48%	0.01% ± 0.03% 2.04% ± 0.40%
	50	0.02% ± 0.13% 2.05% ± 0.4%	0.01% ± 0.12% 2.03% ± 0.43%	0.0% ± 0.0% 2.08% ± 0.42%

Table 1 – Performance results after 10 rounds of 10-fold cross validation. In each cell, the first and second results correspond respectively to the average of the train and test errors with their standard deviations.

Looking at this table for ConvNet, we notice that the performance is better with 50 epochs than with 25 epochs, the latter not being enough to sufficiently train the model. Among the performances on 50 epochs, the best was obtained with a batch size of 100 samples, i.e. 0.59% ± 4.66% and 2.24% ± 0.65% respectively for the train and test errors. For the MLP, the train and test errors are very close for any of the studied batch sizes and numbers of epochs, which in terms of number of epochs means that the additional 25 epochs don't help improving the model much more. Among the performances on 25 epochs, the best was obtained with a batch size of 100 samples, i.e. 0.01% ± 0.03% and 2.04% ± 0.4% respectively for the train and test errors. From this, we concluded that we could set the number of epochs and batch size for ConvNet to respectively 50 and 100, and for the MLP to 25 and 100.

4 Final results

With the hyperparameters set, we finally moved on to evaluate the performance of each architecture and its variants. The performance results are shown in Table 2. They were averaged through 20 rounds for each model using the virtual machine provided in the course. At each round, the data and the initial weights of each model were randomized (by default in PyTorch the initial weights of the different layers are sampled from a uniform distribution). Other than the train and test errors, we also considered the number of parameters of the model and the training time.
Since the results of the 10-fold cross validation for the MLP were very close for different batch sizes with 25 epochs, we also tested a batch size of 25 samples and achieved better performance, which is why this is the batch size used for the final results with the MLP.

Model	Train error	Test error	Parameters	Train time (s/epoch)
ConvNet	0.25% ± 0.86%	18% ± 4.99%	144684	0.57
ConvNet_WS	0.22% ± 0.71%	15.04% ± 0.67%	72458	0.54
ConvNet_AL	0.12% ± 0.54%	12.02% ± 1.19%	144684	0.55
ConvNet_WS_AL	0.0% ± 0.0%	13.27% ± 1.16%	72458	0.56
MLP	0.22% ± 0.64%	18.32% ± 3.33%	171538	0.38
MLP_WS	0.11% ± 0.28%	14.86% ± 0.91%	86920	0.29
MLP_AL	0.15% ± 0.37%	10.01% ± 1.45%	171538	0.4
MLP_WS_AL	0.18% ± 0.71%	9.89% ± 1.36%	86920	0.31

Table 2 – Performance results for each architecture and its variants with weight sharing (WS) and/or an auxiliary loss (AL).

ConvNet alone presents a train error of 0.25% ± 0.86%, a test error of 18% ± 4.99%, with approximately 145k parameters and a training time of 0.57 s/epoch. Using weight sharing made the train error improve by a small amount, but the test error went down by almost 3%, the number of parameters was reduced by almost a factor 2 and the train time was also reduced. Using an auxiliary loss to help with the digit recognition task, the train error was reduced and the test error went down by almost 6% compared to ConvNet. The number of parameters stayed the same as this only changes when using weight sharing, and the train time was reduced. The combination of weight sharing with the auxiliary loss however resulted in a worse test error than when only using the auxiliary loss. This might be because of overfitting, since the obtained train error in this case is 0%.
MLP alone presents similar results as ConvNet, i.e. a train error of 0.22% ± 0.64%, a test error of 18.32% ± 3.33%, with approximately 170k parameters and a training time of 0.38 s/epoch. Again, using weight sharing decreased the train error, the test error went down by almost 3%, the number of parameters was reduced by almost a factor 2 and the train time was also reduced. With the auxiliary loss, the train error decreased, the test error went down by approximately 8% but the train time increased. Finally, the combination of both methods yielded an even better test error of 9.89% ± 1.36%, which is the best performance result we have obtained across all models. The train time results are more consistent with our expectations for the MLP than for the ConvNet : what we expected was, for a given model, to have a decrease when adding weight sharing and an increase when adding the auxiliary loss (since weight sharing requires less parameters and using the auxiliary loss needs more computations). This is exactly what we get for MLP, but for ConvNet the auxiliary loss variant presents a smaller train time than the original model, which we can't explain.
We also notice for both architectures that when using one of the additional techniques or both, the standard deviation of the test error is decreased considerably, meaning that the results across multiple rounds are more stable.
The plots of the average of the loss per epoch are presented in Figure 1 and 2 respectively for ConvNet and MLP. Looking at the plots for ConvNet and its variants revealed that when using one of the additional techniques, the loss is almost constant after 25 epochs or less, meaning that 25 epochs would suffice to obtain similar performance. For both architectures, what is more noticeable in these plots is how faster the convergence is when using the auxiliary loss.

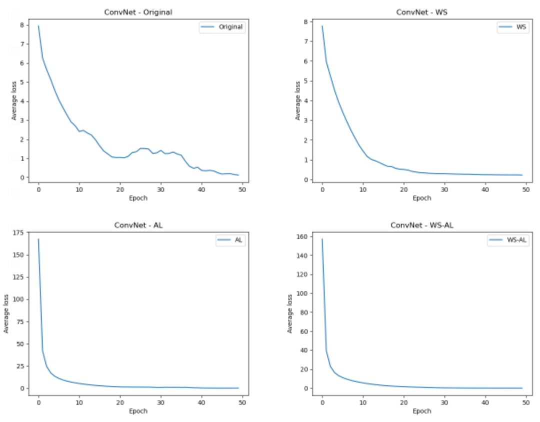

Figure 1 – Average loss per epoch of ConvNet and its variants.

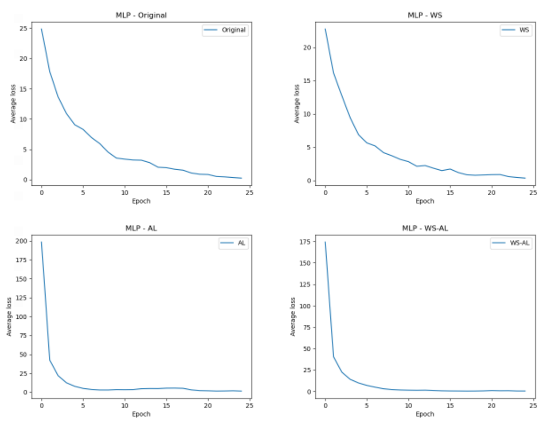

FIGURE 2 – Average loss per epoch of MLP and its variants.

5 Conclusion

To conclude, the two studied architectures present similar results but the variants of MLP achieve slightly better results than those of ConvNet. The train time however is always smaller for MLP than for ConvNet. We showed that the additional methods didn't affect the train error by much but had a great impact on the test errors. This implies that the use of these methods helps the models generalize better. The impact on the test error was always greater for the auxiliary loss than for weight sharing. The latter reduces the number of parameters of the model and thus the train time and results in a simpler model, i.e. less prone to overfitting. The auxiliary loss allows for a faster convergence although it increases the train time by a small amount. When comparing the two methods alone, the auxiliary loss seems better than weight sharing but since they are not mutually-exclusive, the best option seems to combine both of them as shown with MLP. However, it can result in worse performance than the auxiliary loss alone, as shown with ConvNet.

Usually considering every pixel as an input feature is not the best approach for image classification but here the performance results of the MLP are quite good, and the variant with weight sharing and the auxiliary loss gives the best performance obtained. This can be explained by the fact that we are working with small images. Otherwise convolutional layers start to be necessary to achieve good performance.

It should also be noted that we could have further optimized the architectures with for instance a decaying learning rate and by varying the coefficients associated to the auxiliary loss to find better ones.