Deep Learning - Project 1

Manuel Leone, Giorgio Savini Zangrandi
School of Computer and Communication Sciences - EPFL

***Abstract*—In this project we explored the problem of classifying pairs of MNIST images based on whether one number was larger than the other using deep neural network. In order to reach this goal, the baseline was set to a naive network with no training on the image classes. This network was then gradually improved by using mechanisms such as auxiliary losses and weight sharing, that led us to build a siamese network. The performance achieved by this network was satisfactory for the objective, with an overall accuracy of 94.5% over 25 epochs of training performed in 7.59 s.**

I. Introduction

As inputs, we were given a dataset of $[2, 14, 14]$ tensors, each containing 2 images of 14x14 pixels. Each image represented a number from the **MNIST** dataset. This is a commonly used dataset for Deep Learning (DL) tasks, providing 60.000 training images and 10.000 test images. Each image was reduced in size through an average-pooling operation.

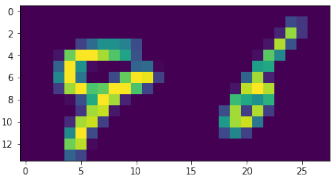

Fig. 1. Example input pair of reduced images from MNIST dataset.

For classification, we were provided with both *image classes* and *targets*. Image classes refers to the numbers represented by the images while the target is a boolean value representing if the number on the left is lesser or equal to the one on the right. The overall objective of the project was to correctly **classify** the pairs of images with the respect to the given targets.

The approach we used to reach this objective is commonly used in the field, going through a first phase of analysis of the input (Section II), followed by the implementation of a baseline network (Section III). We then proceeded to implement advanced networks, by using auxiliary losses and weight sharing to build a Siamese (Section IV) and Non Siamese network (Section V). The final results and their analysis is finally carried out in the Section VII.

II. Data Analysis

A. Distribution of Targets

Among all the dataset, 1000 pairs chosen at random were used for training and 1000 for testing. By sampling training and test targets multiple times and averaging the distributions, we found that the split between 1s and 0s on the targets was **55%-45%**. Thanks to this almost equal distribution, accuracy can be considered as a good metric to assess the performance of the networks, and 55% is the minimum accuracy to obtain a result better than random sampling.

B. Normalization

Before feeding the data to our nets, we normalized it across all dimensions, by subtracting the *mean* (`Tensor.mean()`) and dividing by the *standard deviation* (`Tensor.std()`). This passage is almost mandatory to achieve good performance in deep networks. As a matter of fact, we tried also to run without normalizing the images, and the accuracy of all the networks drops rapidly by around 15% each.

C. Validation and Hyper-parameter tuning

Due to the huge number of parameters in DL networks, such as number of convolution channels, number of hidden units in the fully connected layers, or optimizer learning rate, a phase of validation and hyper-parameter tuning is mandatory.

When the input data are scarce, the traditional way of doing this validation is performing a k-fold cross validation. However, the huge dimension of the MNIST dataset made this unnecessary. Instead, we based our choice of parameters on the average accuracy over **10 runs with a random input of size 1000**. At each run, a freshly new model was generated with random parameters and trained with a batch size of 100. This gives a good estimate of the selected parameters' performance, and ensures to avoid overfitting a dataset thanks to the random input.

To tune the optimizer, we tried both the **Adam** and **SGD** optimizers and explored the best values for the following;

- The learning rate, `eta` (Adam, SGD)
- The momentum factor, `momentum` (SGD)
- Nesterov momentum activation, `nesterov` (SGD)

We generally found that our nets performed better with **Adam** and `eta=0.005` or `0.0025`. Compared to SGD, Adam provided both faster performance and a slightly lower (on the order of 10%) number of errors. The results of this analysis can be found in Table I.

In order to optimize the number of channels in the convolution layers and the number of hidden units in the fully-connected layers, we performed a grid search. The number of channels were chosen among the set of powers of 2: 8, 16, 32, 64, 128 or 256. This search does not try every possible combination for the channels, but rather ensures that the number of channels is always increasing while growing in depth.

We also tried to perform some **data augmentation**, by generating new pairs and classifications at random from the input images. We made sure that the overall distribution of

the data remained uniform - however, this kind of manipulation surprisingly did not prove beneficial for our networks, resulting in slightly worse performance.

III. Baseline

As a baseline, we used a simple net with 2 blocks of Convolution-ReLU-MaxPool, a final Convolution-ReLU as feature extractor, and a classification phase with 3 Linear-ReLU layers and a final Linear. The number of channels was increased with depth, going from 2 to 32 to 64 to 128, before being flattened in the final phase. We used **Cross-Entropy Loss** for training, as it is more suited to a classification task.

Figure A1 shows the net in detail: this is very similar to the canonical neural net used on the MNIST dataset, with 3 key differences:

1) The input is over **2 channels** instead of 1, as 2 images are fed into the networks to perform a classification.
2) The output is of dimension **2** rather than 10. This is because the targets used for training are boolean values.
3) The *classes* of the images (i.e. the actual numbers they represent) are not used for training.

The third point above is especially relevant. In principle, a network could learn by itself how to order the images, given the inputs and the boolean targets we provided. As we will show, however, this kind of network was not capable of doing so in a satisfactory manner.

For the hyper-parameters shown in Figure A1 (`padding`, `groups`, `kernel_size`, etc...), we used a trial-and-error approach, keeping in mind that our options were very limited due to the small size of the images. We considered both how much a layer would shrink our image (in the convolution and max-pooling cases) and whether the kernel would be able to "traverse" all of the pixels (for max-pooling).

IV. Siamese Net

A. Motivation

Given that not using the image classes (i.e. the actual numbers represented by the images) resulted in poor performance, we decided to make a network that first understood **what numbers are present** in each pair and then classified the pair in a subsequent stage. The image class classification is really important, as this gives two **auxiliary losses** that can be used in training. Given that the same kind of training applies to both the left and right images, a **siamese** network with weight sharing seemed ideal for the task.

B. Topology

We split our network into 3 sections:

1) **Feature extractor**: extracts features for classification. This is siamese, meaning that both input images are sent through this component and share weights.
2) **First classifier**: classifies the input image according to the provided *classes*, by assigning it a value from 0 to 9. This is also a siamese component.
3) **Second classifier**: classifies the image pair with a boolean as per the project requirements. This component is not siamese: it receives as an input both outputs of the first classifier and produces a 1/0 value at the end.

The feature extractor is built as a canonical MNIST network, with 2 phases of Convolution-ReLU-MaxPool, followed by a convolution and a ReLU. The 2 classifiers are also a fairly standard succession of fully-connected layers and ReLU, with an addition of Dropout and Batch-Normalization layers at various stages. Figure A2 shows this structure in detail.

C. Training

For training, we used the losses resulting from the 2 instances of the first classifier as auxiliary losses, and summed them to the loss from the second classifier. As before, we used **Cross-Entropy** loss as our criterion. The algorithm is:

1) Split input into left and right image.
2) Send both images through siamese portion of the network and record both losses as auxiliary losses.
3) Send the concatenated output of the siamese portion through the second classifier and record the loss.
4) Sum the auxiliary losses to the loss above.
5) Perform backward pass and optimizer step.

D. Design Choices

We had limited choice on the hyper-parameters that shrunk the images, such as the kernel sizes. We then proceeded as the baseline, by trial-and-error and making sure that the kernel would always traverse all pixels.

For the auxiliary losses, we tried to give different weights to the losses, but in the end we found that **equal weights** performed best. One important point was also choosing how to concatenate the outputs from the first classifier. We tried different approaches, such as taking their difference, the pointwise maximum or **concatenating** them, and in the end we found that the concatenation over-performed the other choices (and this is why the input of the second classifier has dimension 20).

One significant difference between this net and the baseline one is the introduction of **Dropout** and **Batch-Normalization** layers. Introducing them improved performances, most likely due to a reduction in overfitting and in the vanishing gradient effect from Dropout and Batch-Normalization. Due to the small size of the fully connected layers, we used 0.25 as the drop probability for the dropout, assessing a good regularization without losing performance.

V. Non-Siamese Net

To assess the effects of weight sharing, we also built a **non-siamese** version of our previous network, with almost the same structure except that the siamese portions were duplicated in order to prevent the input images from sharing weights. Also, the number of channels was slightly different (but kept the same along each duplicate), as reported by the results of the validation algorithm. Specifically, the channels of each feature extractor in the non-siamese net goes from 32 to 64 to 128. The fully connected layers are instead the same, with Batch Normalization and Dropout. Overall, the siamese network proved to be superior, as will be presented in section VI.

	eta	Accuracy (train)	Std (train)	Accuracy (test)	Std (test)	Number of Params	Train Time
Baseline	0.0025	0.999	0.001	0.827	0.013	81239	4.05 s
Non-siamese	0.0025	0.987	0.009	0.918	0.016	117516	8.84 s
Siamese	0.005	0.991	0.005	**0.948**	**0.010**	192601	**7.59** s

TABLE I
FINAL RESULTS, INCLUDING MEAN AND STANDARD DEVIATION OF ACCURACY VALUES

VI. RESULTS

Our final results for each network are shown in Table I. The siamese net performed best, with a mean test accuracy of **94.5%** and a standard deviation of about **1%**. The non-siamese net performed slightly worse and, finally, the baseline net settled around 80% test accuracy. All of our results were achieved with **25** epochs of training and the estimations are computed over **15 runs**, where at each run the network is freshly generated.

As shown in Figure 2 and 3 we were also able to confirm that the siamese network was not overfitting by plotting training and test accuracy and the total loss over the number of epochs. As the test accuracy never decreases within this time frame and stabilizes around 25 epochs - and the loss stays the same around the same number of epochs - we concluded that this configuration was **optimal**. The error bars in the figures show the maximum displacement with respect to the mean toward higher and lower values.

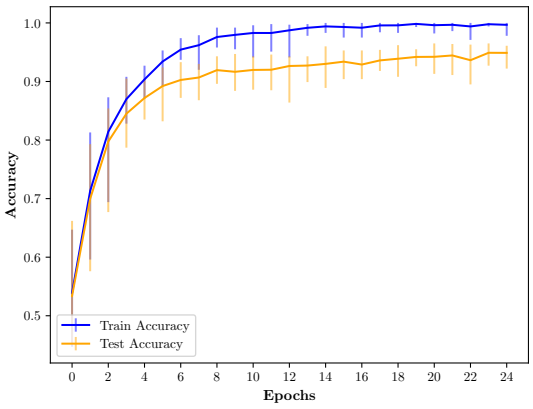

Fig. 2. Accuracy of siamese network over epochs (average over 15 runs).

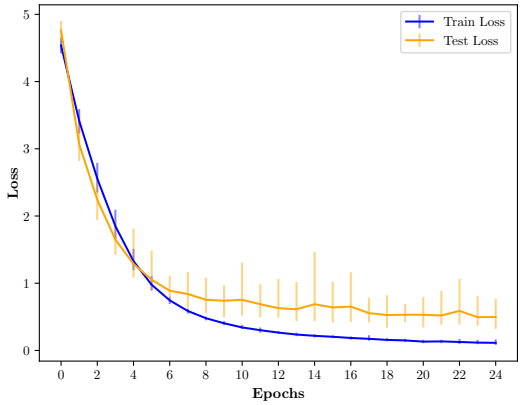

Fig. 3. Loss of siamese network over epochs (average over 15 runs).

Similar plots for the other nets and their relative loss values can be found in Appendix B. Analyzing these plots allows us to reach some conclusion on why the siamese network outperforms the other networks. What is clear is that the other two networks have **high variance** among the runs: this can also be seen by observing their standard deviation values, which are greater compared to the siamese network. Moreover, the baseline network tends to overfit the training set (as shown by the increasing loss after 10 epochs), even though its accuracy doesn't decrease.

A final consideration can be done considering the training times. Of course the baseline is the fastest to train, as it is naive and simple. Between siamese and non-siamese instead, the first requires less time to train. This means that weight sharing improves performance while **reducing training time**, which is a big improvement for the network.

VII. CONCLUSION

In this project we explored 3 different networks:

1) A **naive baseline**, with no training on image classes, adapted from the canonical MNIST classification network.
2) A **siamese network**, with weight-sharing, auxiliary losses, batch-normalization and dropout, that performed training on both classes and targets.
3) A **non-siamese network** equivalent to the one above, without shared weights.

The gradual increase in performance from the baseline to the non-siamese network and from the non-siamese to the siamese one suggests that weight-sharing played an important role, as well as the introduction of image classes into the training phase to compute auxiliary losses. The best result was obtained with the siamese network, which achieved an accuracy of 94.5% with a low training time.

Further improvements are obviously possible, such as using already pretrained networks from the literature, introducing optimizations such as residual networks, changing the optimizers or the losses, or tuning the networks parameters with different values or initializations. Despite this, the results achieved in this project are really **satisfactory**, as we reached the objective of understanding the benefits of weight sharing and auxiliary losses and achieved a really high accuracy.

APPENDIX A
NETWORK STRUCTURES

Baseline Net
2x14x14
nn.Conv2d(2, 32, kernel_size=3, groups=2)
32x12x12
F.relu
32x12x12
nn.MaxPool2d(kernel_size=2, stride=2)
32x6x6
nn.Conv2d(32, 64, kernel_size=3, padding=1)
64x6x6
F.relu
64x6x6
nn.MaxPool2d(kernel_size=2)
64x3x3
nn.Conv2d(64, 128, kernel_size=2)
128x2x2
F.relu
128x2x2
x.view(-1, 128*2*2)
512x1
nn.Linear(512, 50)
50x1
F.relu
50x1
nn.Linear(50, 50)
50x1
F.relu
50x1
nn.Linear(50, 25)
25x1
F.relu
25x1
nn.Linear(25, 2)
2x1

Fig. A1. Layers of one of our first baseline networks, along with the data shape at each stage.

	Siamese Net
FEATURE EXTRACTION (SIAMESE)	1x14x14
	nn.Conv2d(1, 16, kernel_size=3)
	16x12x12
	F.relu
	16x12x12
	nn.MaxPool2d(kernel_size=2)
	16x6x6
	nn.Conv2d(16, 32, kernel_size=2)
	32x5x5
	F.relu
	32x5x5
	nn.MaxPool2d(kernel_size=2, dilation=1)
	32x2x2
	nn.Conv2d(32, 64, kernel_size=2)
	64x1x1
	F.relu
	64x1x1
FIRST CLASSIFIER (SIAMESE)	nn.Linear(64, 75)
	75x1
	nn.BatchNorm1d(75)
	75x1
	F.relu
	75x1
	nn.Dropout(0.25)
	75x1
	nn.Linear(75, 75)
	75x1
	nn.BatchNorm1d(75)
	75x1
	F.relu
	75x1
	nn.Dropout(0.25)
	75x1
	nn.Linear(75, 10)
	10x1
SECOND CLASSIFIER	nn.Linear(20, 50)
	50x1
	nn.BatchNorm1d(50)
	50x1
	F.relu
	50x1
	nn.Dropout(0.25)
	50x1
	nn.Linear(50, 10)
	10x1
	nn.BatchNorm1d(10)
	10x1
	F.relu
	10x1
	nn.Dropout(0.25)
	10x1
	nn.Linear(10, 2)
	2x1

Fig. A2. Layers of our siamese network along with data dimensions. Different stages are labeled on the left.

APPENDIX B
OTHER NETWORKS PLOTS

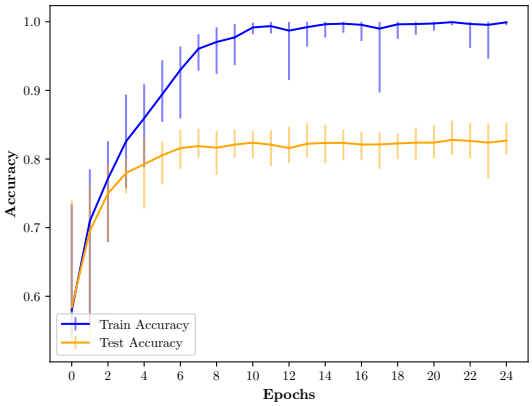

Fig. B1. Accuracy of baseline network over epochs (average over 15 runs).

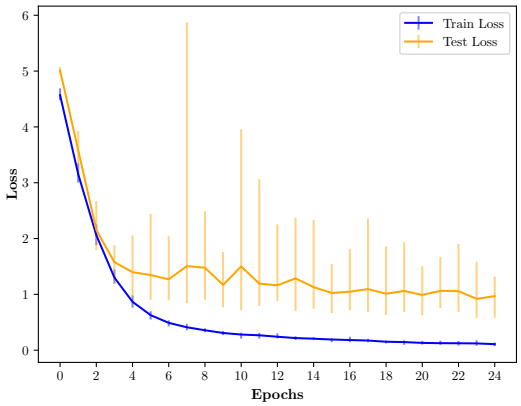

Fig. B4. Loss of non-siamese network over epochs (average over 15 runs).

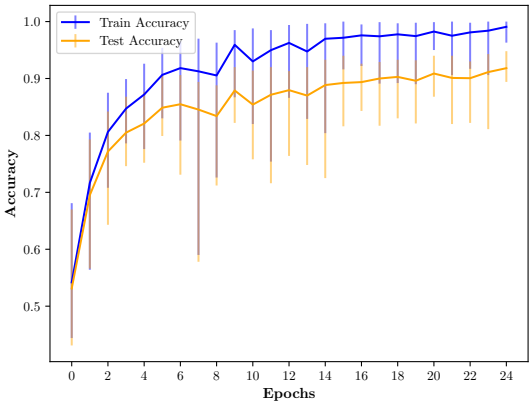

Fig. B2. Accuracy of non-siamese network over epochs (average over 15 runs).

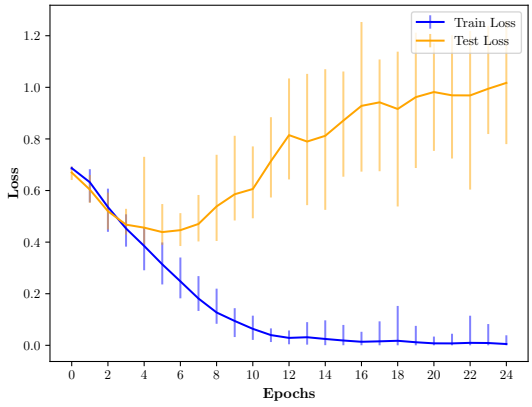

Fig. B3. Loss of baseline network over epochs (average over 15 runs).