Introduction

Image classification is one of the most thrilling subjects in the deep learning community. Not only it is used for practical applications and as a benchmark to compare the performance of competing learning architectures, but it is also helpful to investigate the strategies that the networks figure out to solve the tasks that are given to them. Here, using PyTorch, we dig into different architectures and training procedures to learn a specific task: tell whether an image contains a hand-written digit that is larger or smaller than the one contained in another image.

Methods

In the present report, we analyse the gain associated to the presence (or not) of weight-sharing and alternative losses (understand losses that are not computed by comparing the final output of the network to the sample labels). The sample sets consist in pairs of images, each of one containing a hand-written digit. For each pair, the digits are always different. The labels correspond to whether the first digit is larger or smaller than the second one. To compute alternative losses, each sample also provides the identity of both digits. In Fig. 1, we present the different implemented networks that we compared based on their ability to solve the comparison task.

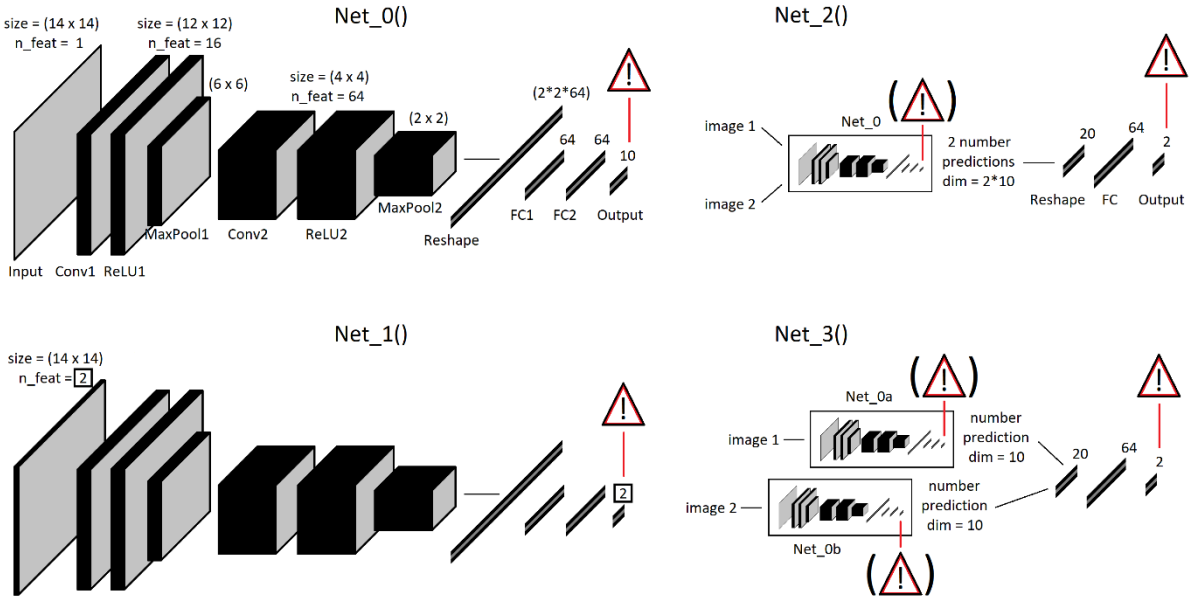

Fig. 1. Summary of the different architectures whose performances are compared in the present report. In each graph, the "danger" sign indicates the basis for the network's loss. **Net_0:** This is a basic neural network doing number classification. It consists in two subsequent [convolution, non-linearity, pooling] sets, followed by a final classifier with two hidden layers for the hand-written digit's identity (10 output nodes). This network does not compare number images, but its architecture was used as a parent class for all other networks and as a baseline for their performance. The basic parameters of the network (convolution size, stride, number of features, etc.) were optimized by hand and were kept the same for all sibling networks. **Net_1:** This network's architecture is similar to the one of Net_0, but it is now naively adapted for number comparison: the input layer now hosts 2 features (both number images) and the final output layer has only 2 nodes (bigger / smaller). Numbers that were modified from Net_0's architecture are framed. **Net_2:** This network contains one instance of Net_0 that analyses both number images, one after the other. This corresponds to using two different number classifiers that share their weights. The output of both number classifications is then fed to a final comparative classifier. The loss is either based only on the final output of the network or on both the final number comparison and the number classification. **Net_3:** For comparison, this network does not instantiate weight sharing. The number images are fed to 2 independent instances of Net_0, whose output are fed to a final classifier. Again, loss is either based on the final output only or on both final and number classifications.

We trained every network of Fig. 1 over 1000 training samples (1 sample corresponds to a set of two number images and their labels) and for 100 epochs. After each training epoch, we ran the network

over 1000 testing samples (requiring the test runs not to compute the gradient nor to update the parameters of the network). Using this scheme, both the training and testing errors were monitored during the whole training phase. This procedure was repeated 10 times for each model, shuffling the training samples order for each run, so that we could compute error intervals for the training and testing errors. This whole set of runs was repeated once using the classic PyTorch stochastic gradient descent (SGD) optimizer algorithm and another time using the Adam optimizer algorithm, using a single learning rate per optimizer (1e-1 and 5e-3, respectively), which was optimized by hand.

Results

Using the method described in the previous section, we could produce the results depicted in Fig. 2.

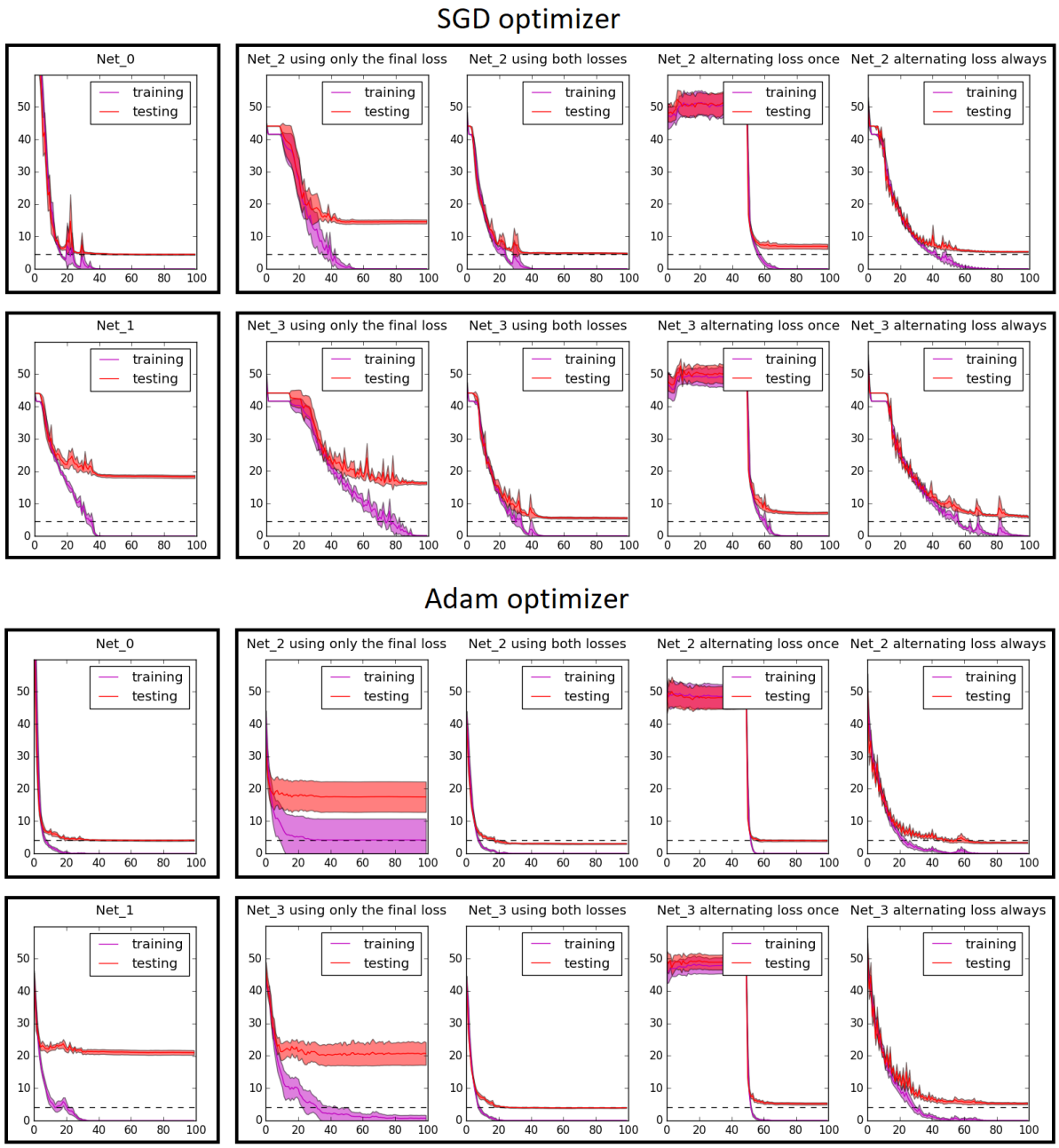

Fig. 2. Results of the simulation of all networks described in Fig. 1. In each graph, the x-axis is the number of elapsed epochs and the y-axis is the percent of errors done by the network. Training and testing errors are shown respectively in purple and red. The shaded areas correspond to the standard deviations and the darker lines in the center of the areas are the means of the errors along the trials. For Net_0, the plotted error was computed on the digit labels. For all other networks, the plotted

error was computed on the final labels. Given an optimizer, the dashed line is the same for all graphs and is computed from the Net_0 results. It corresponds to the amount of error done by this model on the testing set after 100 training epochs (averaged over 10 runs). It is used as a baseline for all other models. **Top.** Testing and training errors, using the standard PyTorch SGD optimizer. **Bottom.** Testing and training errors, using the Adam optimizer.

Discussion

First, training error reached 0 percent in almost all cases. Because the learning rate was kept the same in all training sessions of a given optimizer (for fair comparisons), it was sometimes too large (see "Net_2 using only the final loss" in the Adam optimizer part of Fig. 2), where the gradient descent algorithm got stuck in a sub-optimal configuration of parameters (training error did not converge to 0 percent). In general, the Adam optimizer algorithm produced more stable results (the shaded areas are less fuzzy), took less epochs to converge and led, in the best cases, to smaller testing error rates.

Second, a naïve comparative classifier (see "Net_1" panels) was very prone to overfitting, as its testing error converged to around 20 percent after learning. Adding the basic structure of the task to the network architecture (having two number classifiers and a final comparative classifier, as in Net_3) did not really help, as long as the structure of the task was not implemented in the loss (see "Net_2/3 using only the final loss"), leading only to a slight performance improvement. The error of those network even needed more epochs to converge, because a larger set of parameters were optimized.

However, once the loss of those networks also depended of the digit labels, which corresponded to seal the network's architecture around the true structure of the task, the testing error converged to significantly lower values. It even went below the final testing error rate of Net_0 (see "Net_2/3 using both losses" in the Adam optimizer part), which was a bit surprising, since it seemed reasonable to us that the final classifier could not guess the final label from misclassified numbers. We explain this improvement by the fact that, during the training process, the latter networks have access to new information, digit labels, whose probability distribution is not completely uncorrelated from the distribution of the final labels. Indeed, if for example the first image contains a low digit, the probability that the final label is "image 1 contains a smaller number than image 2" is higher. In the other way around, for this same final label, the first image will most probably contain a low digit and less probably a high digit, and vice-versa. Those additional pieces of information help the network to make more clever guesses on the final label, after the training has converged, when there is too much uncertainty about the identity of one of the digits.

Weight sharing (see Net_2 vs Net_3, for almost all cases) gave a slightly lower testing error after convergence, and also led to a slightly faster convergence. We attribute these small gains to the fact that, during the training phase, the shared backbone of Net_2 is confronted to more digits (twice as many, compared to both backbones of Net_3), avoiding overfitting a bit better and converging faster.

Finally, we tried to schedule the learning process of Net_2 and Net_3, when using both losses, by forcing the networks to only care about digits identities for the first half of the epochs, and only about the final labels for the second half (see "Net_2/3 alternating loss once"). We also tried to instantiate this switch at every epoch (see "Net_2/3 alternating loss always"). In both cases, the schedule actually led to a slower convergence and a slightly poorer final testing error. This confirms that being able to correlate the digit identities and the final labels is useful to the network.