EE-559 Deep Learing Project 1

Damien Gengler, Vincent Yuan

May 27, 2021

1 Introduction

The objective of this project is to test different architectures to compare two digits visible in a two-channel image. It aims at showing in particular the impact of weight sharing, and of the use of an auxiliary loss to help the training of the main objective.

Our goal was to first have a correct implementation of a network that solved the task with similar parameters as the one given in the information sheet. Namely we wanted to have a convnet with rougly 70000 parameters, trainable within 25 epochs in the VM in less than 2s per epochs and achieving around 15% error rate.

We decided to begin with a simple model, and to optimize it using various techniques relevant to our problems until we were satisfied with the result.

2 Implementations

2.1 Baseline Model

Our first iteration aimed at having the simplest working implementation possible first, a simple but modular implementation, that we will be able to improve later.

The input is the concatenation of 2 MNIST-like image, downscaled from 28x28 to 14*14. First we split the two digits and feed them to the two CNNs. The two 2 CNN's goal is to recognize the digits on the image. They have the same structure, which is described below.

We first have a convolutionnal layer with kernel size of 3, which increases the number of channels from 1 to 32 and reduces the input's size by 2. It is then aggregated with a max pool of a kernel size 2 and ReLU as activation function.

The second layer increases again the number of channel from 32 to 64, this time with the same parameters (a convolution kernel size of 3, a max pool with a kernel size of 2, and a ReLU activation function)

We unwrap the 64 channels of size 2*2 in a vector of length 256. We then pass this vector through a Linear and a ReLU activation function to get 200 nodes. Then, we decreases again with Linear layer and ReLU to 10 nodes, each one corresponding to a MNIST digits.

We now have 2 CNN which have 10 outputs, each representing one digit of one of the two inputs. We concatenate these 20 outputs as inputs of a MLP whose goal is to decide whether or not the first digit is greater than the second one.

This MLP also uses two layers of perceptron with ReLu activation function, going from 20 nodes to 300 nodes, then 300 to 300 and finally 300 to 2. The two final output are the probability that one digit is greater than the other.

We evaluate our model using Cross Entropy Loss. Indeed, we have tested the results using CEL versus Mean Squared Error, and the results with CEL outperformed the one with MSE. We have also tested two different optimizer, SGD and Adam, and Adam outperformed SGD, which make sense as Adam is the state of the art optimizer for MNIST. An important remark is that we don't want to pass the output throught a softmax activation layer as the pytorch function CrossEntropyLoss does a combination of LogSoftmax and Negative Log Likelihood Loss.

In order to train the function we assess the CrossEntropyLoss of the final output w.r.t the target given, we use Adam as an optimizer and a learning rate of 0.001. Over 20 trials with 25 epochs each, we find an average precision of **18.42%** with standard deviation of **2.88**.

We obtain similar scores to the one in the instructions. Given the relatively simple architecture of the network, the performances are quite correct. Although the accuracy is quite good, there are techniques we can use to get even better results.

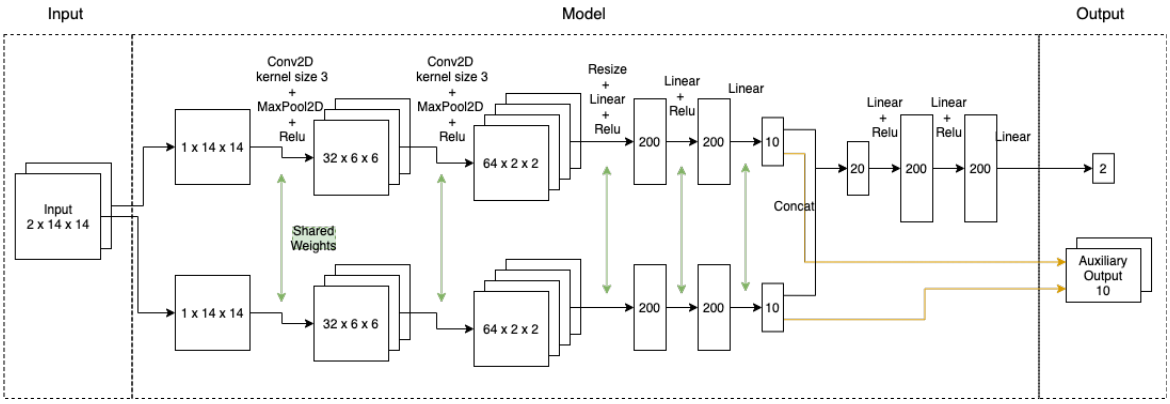

Figure 1: Network's Architecture, the green arrows show where the weights are shared, the orange arrows shows where the auxiliary output used for auxiliary loss comes from.

2.2 Weight Sharing Model

One main flaw of the naive model is the differenciation between the first digit and the second digit in the dataset input. Indeed, we train 2 different networks which have exactly the same goal: extract the digit from a 14*14 MNIST image. This is not optimal as we do the work twice.

Thus, we decided to use weight sharing between these 2 CNN to share the same CNN for the two digits. This CNN will thus twice more input data. However, the MLP comparing the digits stays the same.

We benchmark it again with CEL and Adam optimizer. Over 20 trials with 25 epochs each, we find an average precision of **15.91%** with standard deviation of **2.675**.

The accuracy increases by around 2.5%, which is a pretty nice increase but given the standard deviation of 2 for each results, the increase is not extremely significant and in certain cases, and depending on the initialization of the network, both architectures can give similar results. In our case using weight sharing reduced the number of parameters of our model from 240k to 170k which is quite a reduction.

2.3 Auxiliary Loss Model

The fact that we can "split" our network between a part that will classify the image and a part that will compare them is quite useful for us. Indeed, these auxiliary result allows us to compute an auxiliary loss, which is basically evaluating how well the CNN part of the model classified the digits.

The idea here is that our model will output the final result that interest us, but also the result of the digits classifications. Rather than using a single Loss Function we also evaluate the Loss of each digit by comparing it to the classes target. This regularization technique helps us getting better results by avoiding over-fitting. Given the very few datapoints we have at our disposal, it makes quite a lot of sense to use such techniques.

This is quite interesting because it help training the CNN part by giving it feedback on how well it performed on the final input but the but also direct feedback on digit classification. It intuitively will improve the results because the parameters will be updated several times. This also makes it so lose less information during training.

The training is done in a similar manner but this time we use 3 Loss functions, two for the digits output (The red arrows on Figure 1) and one for the final output. We chose to use the Cross Entropy Loss for the three losses as well as Adam optimizer.

Then we call backward on the three loss function consecutively and once this is done we do our optimizer step.

On 20 trials with this architecture and 25 epochs at each trial, we achieve an average error rate of **11.06%** with a standard deviation of **2.230**.

We can see that using this auxiliary loss gives us much better performances. This does make sense because we are able to push the CNN in the right direction much more precisely because we evaluate its output directly, rather than a "proxy" of its output in the final output.

2.4 Adding Dropout layers

As we don't want the model to overfit on the training data, we decided to use dropout, which is randomly dropping nodes during the training. After benchmarking, we decided to only dropout the layers of the CNN and not the ones of the MLP, even though the difference was ¡ 0.5%

We benchmarked it with different dropout rates, namely from 0.1 to 0.9 with a step of 0.1.

On 20 trials with this architecture and 50 epochs each, we achieve and average error rate of **8.515%** with a standard deviation of **1.957**.

An important note about this Model is that it requires more epochs to achieve this precision. This makes sense because we discard some data so we need to go through it several times to train correctly. The error rate with 25 epochs was worse than the model without dropout.

2.5 Optimization to meet the requirements

As our model including weight sharing, auxiliary losses and dropout layer performed well, we decided to keep it as our final model.

While the models are performing well, they have 170 000 parameters when using weight sharing. We decided to reduce it to follow the project specifications, namely roughly 70k parameters and a run with 25 epochs and 1000 input in 2 seconds per epoch.

To achieve 70k parameters, we decreased the numbers of nodes in our perceptron from the CNN and in the MLP. Namely, we decreased the size of the vector from 200 to 128, and the size of layers from 300 to 128, which enables us to arrive to 70k parameters.

On 20 trials with the Auxiliary Loss Architecture, we achieve an average error rate of **11.81%** with a standard deviation of **2.305**.

While the accuracy is still pretty good, it is still a bit worse than with bigger layers. Although our initial number of parameters is bigger than the one require it is still relatively close to 70k (only twice as many), so we decided to keep our model for the previous benchmark.

3 Conclusion

Starting from a naive yet working model, we enhanced it by adding iteratively various deep learning techniques relevant to our problem. Through the use of various regularization techniques we managed to improve the performances of our model.

The use of Weight Sharing allowed us to get better performance as well as a smaller model. Adding Auxiliary Loss added regulation to the model that allowed us to avoid overfitting which is quite useful when dealing with deep nets on such a small training set.

While the use of weight sharing was not an extremely significant increase in performances, adding the auxiliary Loss allowed us to get a 5% increase which is pretty good for our model.

Adding Dropout as an additional Regularization technique allowed us to get even better results, at the cost of more computation time.

While these regularization techniques show good results on our Data, we still have to start from a pretty good model, that risk Overfitting on the data. Indeed, they are not magic and adding such techniques in a context where we underfit would not show such good results. In those case, one should focus on increasing the model performances first.