December 2022

Deep Learning

Mini Projects

Mini Framework, Classifications, Weight Sharing, Auxiliary Losses

Genis Skura

Joao Filipe Costa Da Quinta

Jan Dirkx

Contents

1 Introduction

This document will serve as the report of our work on the two mini projects given as part of the Deep Learning course in the University of Geneva, during the semester of Autumn 2022. It will include an overview and the results of the work done from the early stages to the finishing touches for both projects. We would like to emphasize the fact that the main sources used as references for this task were PyTorch's documentation site and the practicals given during our course.

Regarding the ***first task***, we will summarize our initial approach of trying out a Multi Layer Perceptron and a Convolutional Neural Network as our two NN models used for the classification task. Benchmarks to compare both models (which shared the same optimizer in addition to the same learning rate η to avoid a bias of learning techniques) were performed to compute the training and test errors with relation to different number of epochs along with different setups of dropout. Consequently and as expected, this resulted in the CNN model performing a bit better on average (the average of 10 runs), therefore that was chosen as the model to move forward to in our task to compute the effect of both the auxiliary loss and weight sharing, in addition to other methods.

As for the ***second task***, the tools and modules along with their forward, backward, parameters, gradient descents, constructed to build the structure of a simple deep learning framework will be explained. The efficiency and accuracy of these will be briefly explained using the same benchmarking approach as in the first task, however adapted to the other binary classification task related to the disk of radius $\frac{1}{\sqrt{2\pi}}$. This helped us understand how the model reacts to different number of hyper-parameters, epochs, sample sizes, batch sizes etc.

2 Project I - Classification

To begin with, we approached this task as a binary classification task (2 dimensional input $x \Rightarrow 1$ binary output y) related to our targets instead of a MNIST multi-class classification for both the *MLP* model and the CNN which was called *BinaryCNN*.

2.1 Shared General Properties

Both networks share the same optimizer for their parameter optimization, also as a kind of a similarity constraint to not influence the benchmarking, specifically the Adam Optimizer with a learning rate of $\eta = 0.001$.

Loss Function

Regarding the loss function, Binary Cross Entropy (BCE) was chosen as the best possible candidate for our binary classification task. Specifically, PyTorch's *nn.BCEWithLogitsLoss* that combines a Sigmoid layer with the BCE loss in one single class as it is a more stable version than applying them separately and removing the need to apply the Sigmoid function on the final output of the network.

$$BCE(t, p) = -(t * \log(p) + (1 - t) * \log(1 - p))^1$$

Computing Errors

To compute the number of errors in order to calculate the classification error for both the train and test sets, a function was constructed that takes the output of the model as parameter. After that, the sigmoid[2] function is applied to squish them in the interval between 0 and 1. The values are then rounded to the nearest binary value in order to be compared with our binary targets using the conditional function *torch.where(p != t)* that returns a list of the indices where they don't match. Thus, the length of this list is the total number of errors.

[1] $t \Rightarrow target$, $p \Rightarrow prediction$.
[2] $1/(1 + \exp(-x))$

Batch Normalization

In both networks, batch normalization was applied after each one of the layers except the final output fully connected one. By performing tests with it enabled and disabled, it was found out that the regularization it provides resulted in shorter training times and better results for both networks. The standardization that it provides also resulted in minimal standard deviation values with relation to performance over runs, pointing out that the same model with the same hyper-parameters will lead to the same (approximate) performance and to a faster network convergence no matter the number of runs.
As a side effect, applying batchnorm after each CNN/FC layer resulted in the decreasing of dropout probabilities p_D or the removal of them alltogether.

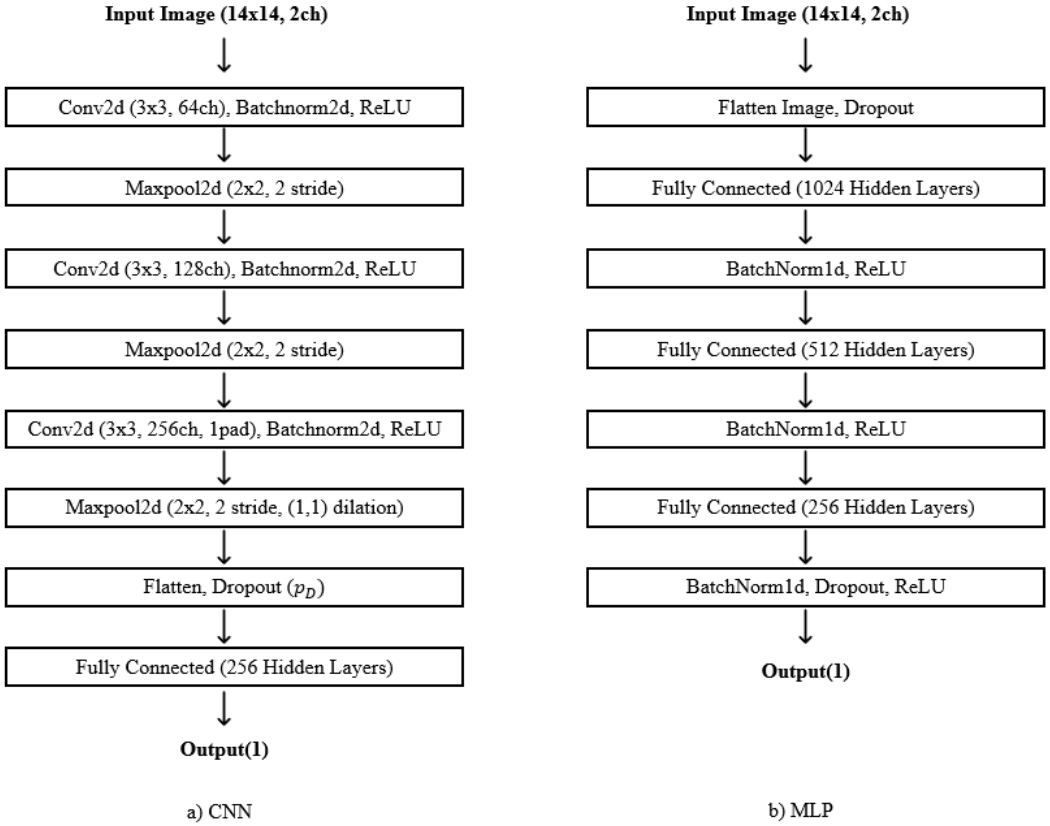
Dropout

Dropout layers were added to both models as the most simple and straightforward way to combat overfitting. The dropout rate p_D was passed as a hyper-parameter and different tests were made to conclude on the most adaptable one. Even though it was initially applied only on the fully connected layers as per usual, tests were made to apply dropout on the convolutional layers too, although with minimal values ($p_D \in [0.1, 0.2]$), given some of the research [3] that has suggested that it could achieve good results.
Having said that, it resulted in a worse performance in our case therefore we kept it limited to the fully connected layers. Even there, we stuck to low dropout rates given that higher values would interfere with the regularization already provided by the batchnorms everywhere and would lead to lower performance rates.

[3] Park S., Kwak N., Analysis on the Dropout Effect in Convolutional Neural Networks

2.2 The Networks

The schemas below will describe the architecture of both networks, as it was finally selected after running performance tests with different combinations of hidden layers for MLP, in addition to different combinations between number of channels, kernel sizes, and number of convolutional layers for CNN.



After arriving to this layout of the networks, we performed performance tests comparing the train and test error rates with relation to the number of epochs e ($1 \leq e \leq 100$) in order to decide which model performed best and which model to choose to move forward with our task of introducing the weight sharing and auxiliary loss.

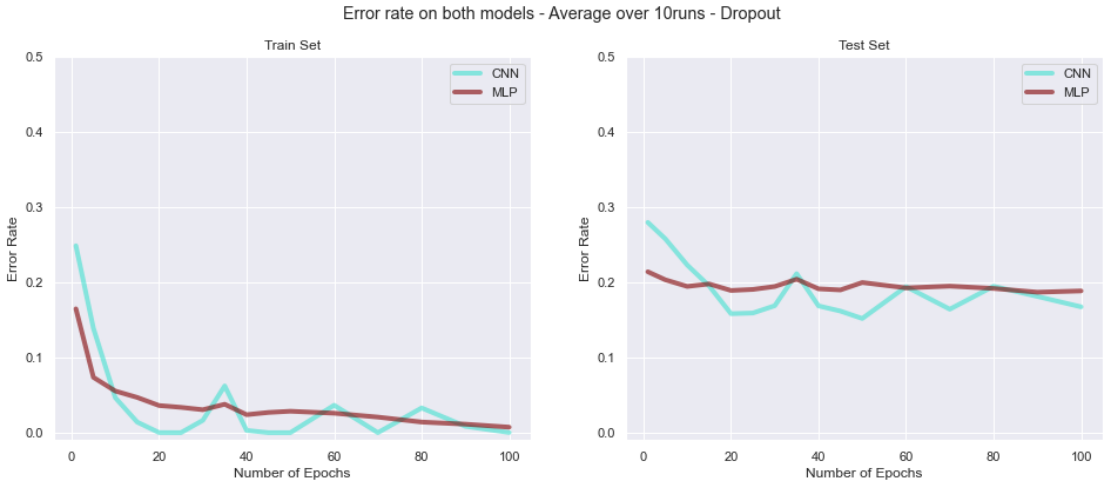
2.2.1 Performance Benchmark

Multiple separate runs to compute the average train and test errors were launched. The difference between them were the state of the dropout layers, one time enabled and the other time disabled, in order to compare their effect.

We could instantly deduce that the CNN performs a better feature extraction when taken both images as input at the same time therefore its results are better on average and the model is more stable overall. The MLP model on the other hand still maintains a good performance but suffers from a lack of stability which can be seen through the sudden peaks in error rate even though a higher number of epochs was provided to train. Having said that, we could also see that the CNN model suffers from overfitting more than the MLP model.

For the second part of this benchmark with dropout enabled, a dropout rate $p_D = 0.4$ was set for each one of the dropout layers of MLP while for CNN the same rate was enabled to be applied to the flattened input layer before going in through the final fully connected layer which acts as a classifier.

We should emphasize again that no dropout was added after convolutions as it worsened the performance therefore we decided to drop them and just add it before the final classifier.



Some runs from the results above were selected to be displayed on a table, to allow for a more close view on the differences between epochs and the run with the global minimum.[4]

Model	Epochs	Train Error (%)	Test Error (%)	Minimum Test Error (%)
MLP	5	7.36	20.34	19.2
CNN	5	13.89	25.79	17
MLP	25	3.37	19.06	16.9
CNN	25	0	15.92	14.8
MLP	50	2.85	19.98	18.4
CNN	50	0	**15.19**	14.5
MLP	80	1.41	19.19	17.9
CNN	80	3.29	19.48	**13.8**

We could see that the introduction of dropout had a larger positive effect on the MLP network rather than the CNN. Comparing these MLP results with the previous ones, we could see that the performance curve is more stable in addition to the obvious better results. Having said that, CNN still performed better so it was the model selected to move forward with the task from this point forward. Regarding the number of epochs, an $e = 25$ was selected as the default value and the compromise between runtime and stability in performance.

[4]Train and Test error rates are averages over 10 runs

Note: Arriving to these results even after having added dropout layers, we could see that the model is still suffering a bit of overfitting, especially CNN. Hence since that was chosen as the model to move forward with the task, a couple of modifications were made to the train function to try to combat this phenomena and improve results.

Weight Decay

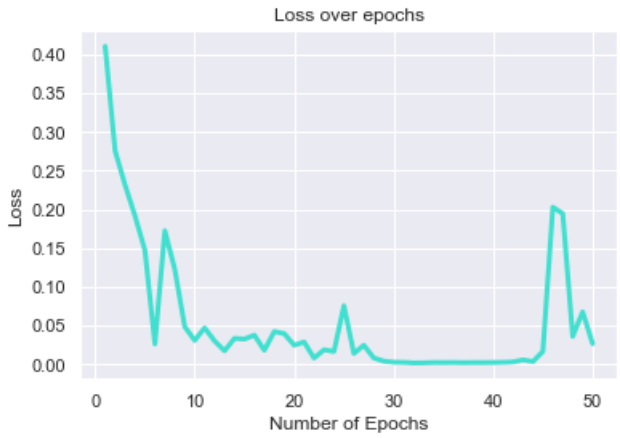
Even though we already have the numerous batchnorm layers as our main regularization technique, the L2 regularization known as weight decay was also added as another technique to penalize any potential large weights. The PyTorch implementation of this method by putting it as a parameter inside the *optimizer* function applies weight decay to the weights and introduced the regularization term to the loss function as:

$$Loss = Error(y, \hat{y}) + \lambda \sum_1^N w_i^2$$

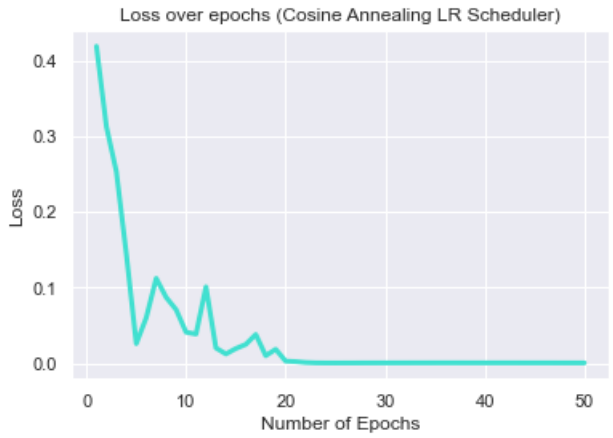
After testing the network with different values, a $\lambda = 0.01$ was selected as the parameter which kept the results stable across runs and had a small but positive effect on the test error rate. ($\in [-1\%, -0.5\%]$)

LR Scheduler

While running the benchmarks displayed above, we noticed bizarre fluctuations in the loss value during the later stages of training, when the number of epochs went above $e \geq 30$. For example, the spike in loss value during $e = 45$ visualized below:



Given that, we decided to make the learning rate η dynamically change over runs by implementing a scheduler, instead of leaving it constant at $\eta = 0.001$. This was a decision made with the hope to improve results and allow the optimizer a better exploration of the search space. Knowing that *Adam* was our optimizer, and inspired by the paper from *Loshchilov et al.*[5] on the topic, we decided on two schedulers, Cosine Annealing and Cosine Annealing with Warm Restarts, both implemented in PyTorch through *torch.optim.lr_scheduler*.
The implementation of Cosine Annealing resulted in the majority of the runs having the following type of decrease on the loss:



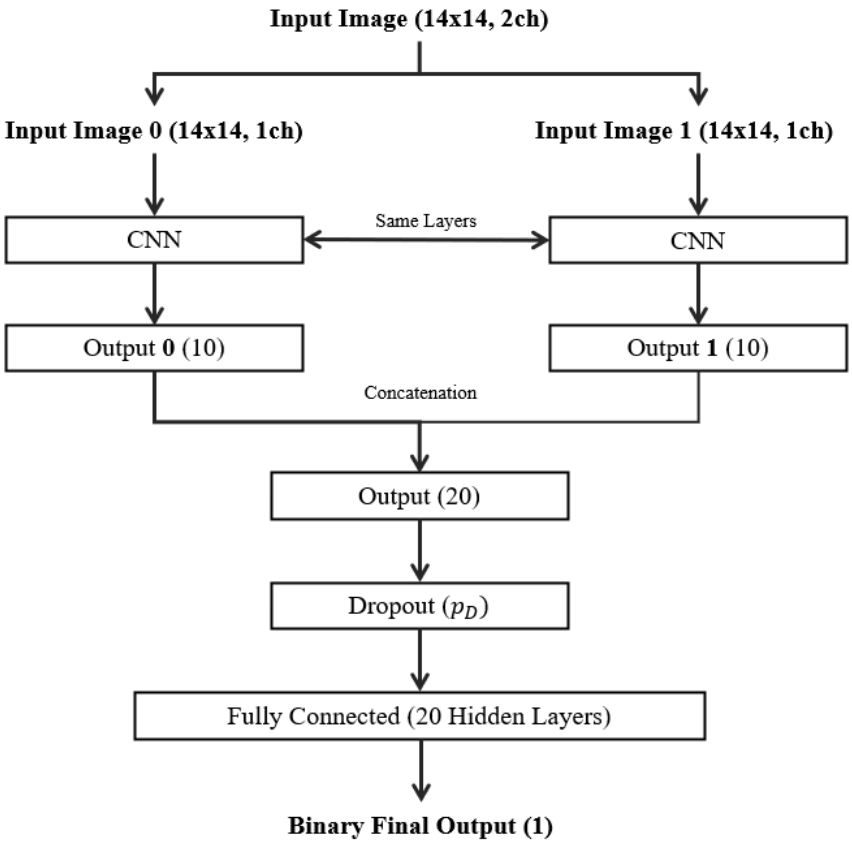
The continuous decrease of the learning rate that the method provides through a cosine reduction, removed the fluctuations over multiple runs, however the faster convergence worried us since it became more probable to get stuck in a local minimum. Therefore, the scheduler chosen was *Cosine Annealing with Warm Restarts*, with following parameters:

- Minimum learning rate: $eta_{min} = 0.0001$
- Number of iterations till restart: $T_0 = 5$, (every 5 epochs)
- Factor by which to increase the LR after restart: $T_i = 1$ (set it back to initial η)

[5]Loshchilov I., Hutter F., Decoupled Weight Decay Regularization, 2019

2.3 Weight Sharing

After having decided to use and already tested the CNN network, the next step is to introduce weight sharing to study the effect it would have on performance. Given the fact that our input is made of 2 channel images, we instinctively decided to split the two images an pass them one by one through the same CNN model, similar to a Siamese architecture, which served as inspiration. Hence, the model took the following shape:



c) Weight Sharing CNN

After updating our model to the structure visualized above, we *expected* an improvement in performance in comparison to the first CNN model before performing the actual benchmarks. The reasons behind this were purely instinctual.

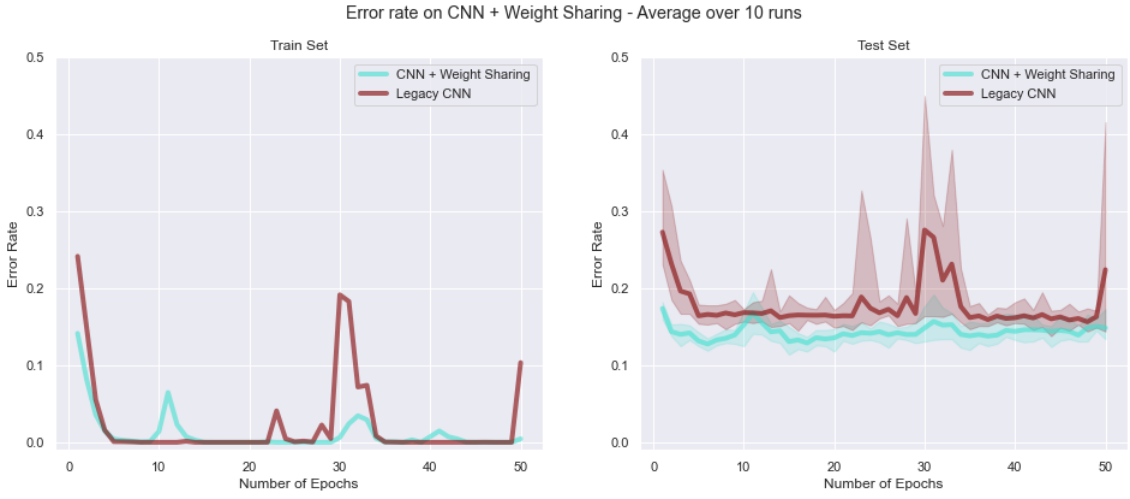
Specifically, splitting both channels and passing the images through the convolutional layers one by one resulting in a 1d vector of 10 elements as output, is equivalent to a traditional MNIST classification task. Therefore our network would have a better understanding of the digits contained in the image themselves rather than a feature map computed over both channels at the same time.

We believed that this connection of these convolutional layers through the concatenation of the results output by the shared CNN would lead to a better weights update. If one of the layers results in the mapping of a specific feature, it would be reflected on the update in the parameters, which could be seen as "informing" the other layer. Hence, we expected a better generalization.

As for the loss function, *Binary Cross Entropy* was still kept since the models still outputs a binary result.

2.3.1 Performance Benchmark

This shared network was compared with *first CNN network*[6] we described in the section 2.2. Average error rates over 10 runs were computed for each epoch e in the interval $e \in [1, 50]$.



[6]Labeled as LegacyCNN in the results

Model	Epochs	Train Error (%)	Test Error (%)	Minimum	SD
Legacy CNN	6	0.08	16.61	15.3	0.0148
Weight Sharing	6	0.27	12.77	**11.9**	0.0049
Legacy CNN	10	0	16.85	15.5	0.0201
Weight Sharing	10	1.42	15.25	12.5	0.0129
Legacy CNN	15	0	16.45	15.5	0.0111
Weight Sharing	15	0.01	13.08	**11.4**	0.0097
Legacy CNN	35	0.04	16,1	14.7	0.0107
Weight Sharing	35	0.05	13.8	12	0.0122
Legacy CNN	45	0.01	15.88	14.5	0.0135
Weight Sharing	45	0	14.41	12.4	0.0103

Some of the runs from the visualized graph, including the run with the global minimum

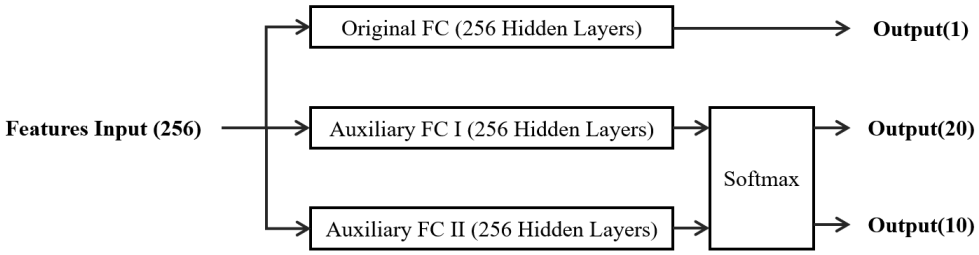
Remarks:

As expected, introducing weight sharing to the convolutional network resulted in better performances on average in addition to reaching a better global minimum. The network with weight sharing is more stable through the total interval of all epochs as we can notice by the lack of spikes in the graph, a phenomena that happened to the Legacy CNN model, especially between epochs 30 to 35. The minimum values of standard deviation enforce this statement of consistency too, meaning that for each of the 10 runs, the model's performances were very close to each other.

Another interesting remark that we would like to mention is the faster convergence (e.g. results at epoch $e = 6$) that occurred in this benchmark in comparison to the previous one computed at section *2.2.1*. This should not come as a surprise as it is an effect of the addition of the learning rate scheduler, given the restarts happening every $e = 5$ epoch, creating the ground for the optimizer to take more aggressive steps.

2.4 Auxiliary Loss

In this part of the task, we introduce an auxiliary loss as the additional loss supplied to our loss function in order to achieve a better learning process and potentially optimize the results. In order to introduce and calculate this new loss, the true target classes of our MNIST digits are available to use. By implementing this, the same neural network model will go through the process of auxiliary learning too, thus a traditional MNIST classification task in parallel with the current binary classification task. Therefore, another fully connected layer used as this auxiliary classifier was added to the already existing convolutional network, taking as input the same feature maps extracted from the convolutional layers. Given the nature of our two existing CNNs, two auxiliary classifier output layers were implemented:



Auxiliary fully connected layer's output dimensions were set to 10 and 20 with one hot encoded target classes in mind, one time to classify one single image and the other time to classify the two images at once. Therefore, *Cross Entropy* was selected as the auxiliary loss criterion. It could also work with the target classes as the original integers, a case in which we would need 1 and 2 outputs respectively, however through testing it was found that one hot encoding leads to better results.
Therefore, for a number of classes M:

$$aux\_loss = -\sum_1^M y_{p,c} log(p_{p,c})^7$$

Leading to the final loss function:

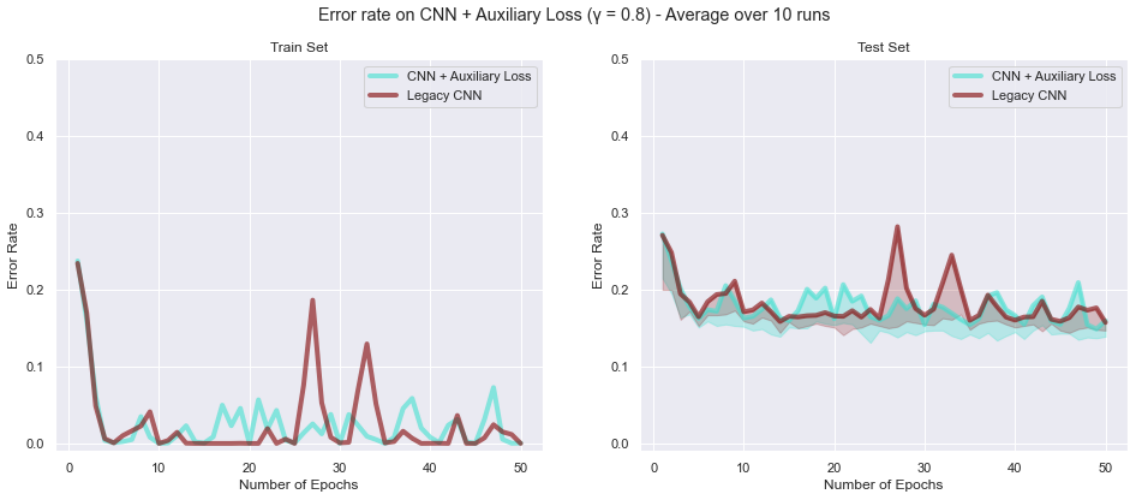
$$total\_loss = BCELoss + (\gamma * aux\_loss)^8$$

[7] $y \Rightarrow$ Binary indicator if the class label c is the correct one for prediction p
[8] $\gamma \Rightarrow$ Multiplier of the impact that auxiliary loss should have, hyper-parameter

2.4.1 Performance Benchmark

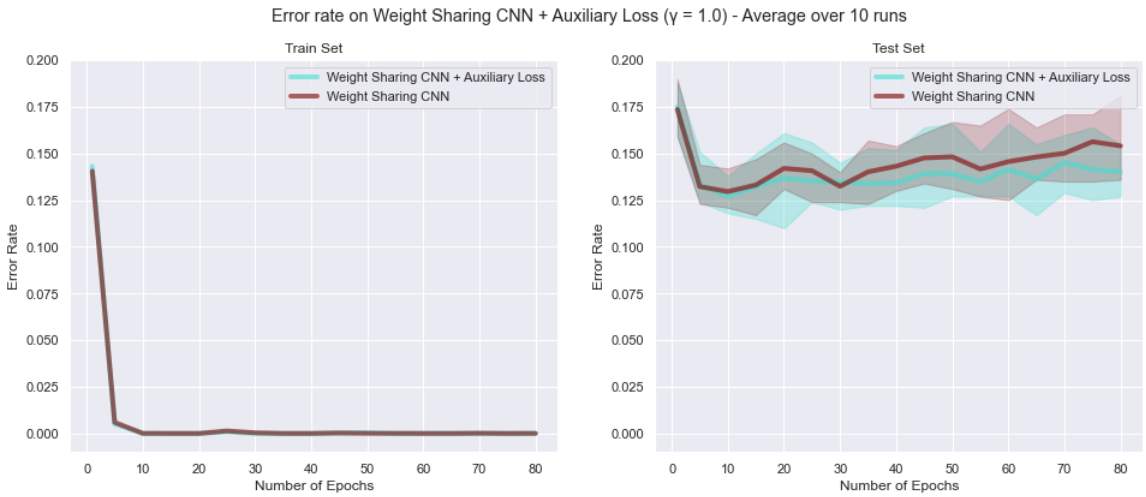
After modifying both convolutional models in order for them two return two outputs, the main one and the other one used for the auxiliary loss, in addition to modifying the training function itself to calculate the newly defined loss, benchmarks were run.
The first *LegacyCNN* model was compared with the modified version of itself, modified to include the 20 output auxiliary loss knowing that the model takes both image channels at once as input.
While the other better performing CNN model with added weight sharing was also compared with the modified version of itself, that included the auxiliary loss computed with regard to the prediction of the class for one image at a time, essentially a MNIST classification task.

Results I:



The introduction of auxiliary loss (with a set $\gamma = 0.8$ after tests with different values) didn't have a noticeable effect on performance like in the case of weight sharing however made the model more stable. It also had better global minimums than the first CNN model.
Having said that, average error across runs ranged in $\in [0.145, 0.21]$ while the global minimum was at $error = 0.131$ at epoch $e = 35$.
Knowing that weight sharing resulted as a better addition to our model than auxiliary loss, we decided to combine them both for the next results.

Results II:



Note: Y-axis limits have been shkrinked in comparison to other figures

2.5 Remarks

Here we will list some of the most important general remarks we concluded by working with the task. We should also mention that we found this project in particular the most interesting and challenging since it allowed us to be more creative with different type of models, losses, schedulers, optimizers, hyper-parameters.

- ***Networks*** $\Rightarrow$ Since the first part of this report, it was shown that the CNN model performed better than NLP. Therefore we can conclude that for a computer vision binary classification task as this one, convolutional layers are more performant, faster, and easier to train.
- ***Regularization*** $\Rightarrow$ One of the earliest modifications to the networks which resulted in a better and more stable performance, were the dropout and batchnorm layers in particular. The regularization they provided created the ground for stability, faster training, avoiding problems related to the gradients.
- ***LR Scheduler*** $\Rightarrow$ Even though not consistently, introducing the *Cosine Annealing with Warm Restarts* scheduler to the optimizer resulted in a better and faster convergence to the global minimum. Restarting frequently every 5 epochs made the models a bit more unstable on later epochs though.
- ***Auxiliary Loss*** $\Rightarrow$ Introducing cross-entropy loss as an additional loss to our model, resulted in more stable models with lower variance. However, in terms of error-rate performance, it didn't have a very distinguishable positive effect like the addition of weight sharing did. Having said that, it should be seen as a good stabilizer of the training and regularization mechanism, so we should implement it when possible.
- ***Weight Sharing*** $\Rightarrow$ Taking the 2-channel images one by one through the model with shared weights, resulted in the best performance booster, improving average error rates by $[\sim 2\%, \sim 5\%]$ and lowering the SD to minimal values that showed the stability that it introduced to the model through the runs. Global error test minimums in the interval $\in [10\%, 11.5\%]$ resulted only in runs of a model with weight sharing enabled.

Potential Improvements

In the list below are some aspects we would've liked to explore during this task, if we had more time to explore it.

- ***Larger Dataset*** $\Rightarrow$ Testing how the model would perform in a larger dataset, instead of only $n = 1000$ samples, potentially even the full MNIST dataset.
- ***Data Augmentation*** $\Rightarrow$ An idea discussed as a way to reduce the overfitting that occurred in all models, augmentation of the input images through tactics such as random rotation. Most likely, this would lead to a better performance.
- ***Deeper CNN Networks*** $\Rightarrow$ Since the networks we proposed and built are still considered shallow networks, we would've liked to have in hand a deeper model with a larger number of parameters, or even comparing our models with an established known network.

3 Project II - Mini Deep Learning Framework

The objective of this second mini-project was to create a deep-learning framework using only tensor operations in PyTorch, the framework given during the course, the standard Python's built-in library *math*, and most importantly without using autograd. In order to achieve the construction of this framework, we had to create each one of the modules, which we will define as *tools*. Specifically, *Linear*, *Tanh*, *ReLU* functions and the *Sequential* to connect all the layers. As for the calculation of the error, we had to create MSE function as a loss function.

3.1 Data generation

We generated the data from as a set of 1000 points in a 2D Space, by first sampling from a random distribution in the interval $[0, 1]$. We then trained the model with the classification of the points by giving the a label, 1 of they are in a circle of radius $\frac{1}{\sqrt{(2\pi)}}$, -1 the other way. This type of classification is the reason why the *tanh* activation function before the final output was needed, since it achieves the redistribution of values to the problem's exact interval, $y \in [-1, 1]$.

3.2 Tools

Each one of the tools is implement with the 2 main methods, in which we move through a neural network, in mind: the *forward* and *backward* passes, which achieve the forward and backward propagation steps. As for the implementation process, and with the theory behind each one of the functions in mind, we first started by implementing the activation functions of $ReLU$ and *Tanh* respectively, and after that we moved to the implementation of the fully connected *Linear* layers.
After the functionality of these implementations was tested and after making sure that they're stable, the *sequential* module was built and the tools mentioned above were modified to adapt them to the newly sequential one.
Once all of these modules were implemented and functional, we moved through with the problem itself, which was in need of a loss function, therefore we finished with the MSE Module.

3.2.1 ReLU

ReLU is a linear function which for forward propagation will output the input directly if it is positive, otherwise, it will output zero. It's the easiest activation function to implement and use, and the improvement that it has on results easily makes its discovery a milestone in deep learning.

$$f(x) = max\{0, x\}$$

For the back-propagation it gives 0 if the input is negative or 1 if it is positive, as for the respective derivatives.

3.2.2 Tanh

The other activation function implemented was the hyperbolic tangent one. The advantage of this one is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh sigmoidal graph. This function is very useful in binary classification tasks. It usually results in high values of the gradient during training leading to higher updates on the weights, therefore if we want strong gradient and aggressive learning steps, we should use this activation function.

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

3.2.3 Linear

At initialization, the module takes both dimensions of the input and output as parameters, as well as a bias optionally. The weights and biases are randomly initialized. During the forward method we modify the weights using the feed forward formula, taking as inputs the weight matrix W, the input tensor x, and the bias b, as given in the course:

$$y = W * x + b$$

The derivative for the backward propagation. We also implemented and use stochastic gradient descent to improve the results and control the learning process a bit more through the introduction of a learning parameter η.

A *zerograd()* method was added in order to be able to recompute the module using new parameters (zeroing them), so that the gradient doesn't become a combination of the old gradient.

The objective of this module is to change the dimensions with the output/input of the layer and to combines the input nodes with weights and biases, finding the perfect ones through the optimization of them using SGD (in our simple case), would lead us to be able to make the perfect prediction with the given output, for a large number of tasks.
This help in the change of the dimensionality that they provide, in addition to the discovery of relationships between the values in the data through a simple function such as the one defined above, makes linear layers one of the most important types of layers in deep learning.

3.2.4 Sequential

Once every module was implemented and functional, another one was necessary in order to provide an easier way to run then together. This is where the sequential module comes in. This module links the classes together as well by creating a sequence of the layers and activation functions in both directions, forward and backwards.
The forward() call saves the sequence in the given order, so it is able to propagate as specified in addition to keeping the correct dimensions after each layer. This is achieved by running the forward functions of each tool, in a for loop iterating through the given order. While the *backward()* method does the same to achieve back propagation, however just in *reversed()* mode.

Once implemented, we can create our models by input the functions in the order and depth we want, and with the parameters that we need. The only thing left in order to solve our task involving a radius, is a missing loss function.

3.2.5 MSE Loss

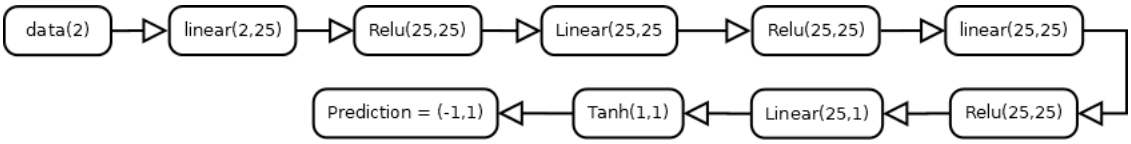
To evaluate the loss we use the Mean Squared Error after finishing the forward pass. The result are then use in the back propagation to improve results. This process is the equivalent of the *loss.backward()* method, already implemented in PyTorch.

$$MSE = (x - y)^2$$

The criterion to measure this between each input element x and target y.

3.3 Results

We tried several models, both shallow and deep versions, and settled on one that was the most performing on average (10 runs) after benchmarking. As can be seen below, all the tools we mentioned above are included in the model, all wrapped up in a final sequential.

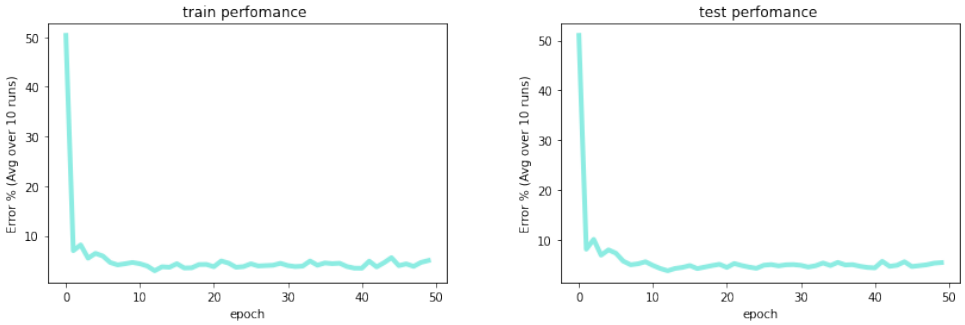


With a bias $\beta = 0.5$, number of epochs $e = 50$, a batch size of 1 and after running it 10 times, we arrived to these results:

Metric	Train	Test
mean	2.41	2.51
min	1.5	1.7
max	3.5	3.6
std	0.6	0.65

Mean of 10 rounds

To be able to see the evolution of the model loss, we tested the model with several values of epochs up to 50 which resulted in :



The graphs above show how fast our model improves with the number of epochs increasing.

3.4 Conclusion

We can see that the error rate is very for the training data as for the test data both the error rate and the standard deviation are not very high. In the graph we notice that the error decreases logarithmically which is expected, as the goal of this network is to reduce the quadratic error between the predicted result and the target.