# A beginner's introduction to Neural Networks: four case studies on the MNIST dataset

Rodrigo Senofieni
email: rodrigo.senofieni@mail.polimi.it

February 2020

# Contents

# Preface

Neural networks are becoming more and more important since the past ten years. Their impact on many applications - where state of the art software or algorithms were used - has been huge. Especially, their impact was very important in the field of computer vision, where there have been thousands and thousand of researches carries on by many people all around the world.
However, having a complete and fully understanding view of how they work is yet not very simple. Aim of these four case studies is to give the reader the chance of understanding the working principle of neural networks and their mechanism (chapter one). Then, we will discuss about the importance of data and the impact that this aspect may have on the performances (chapter 2). After that we will have a quick look to a special -and yet very interesting- type of Neural Networks like GAN (chapter 3) and at the very end we will discuss about possible problems related to the *"safety"* aspects of NN (chapter 4).
As a final remark, these notes are the results of five month of research period at Tohoku university, where I attended with interest the course of *computer vision*. As a consequence, discussions and results may not be the most accurate ever, but they are meant to demonstrate the potential that neural networks can achieve. for any question or the code, feel free to contact at

# 1 Chapter 1
# Introduction to Neural Networks

In this first chapter we want to get in touch with the most important features of a NN, such as Activation function, layers, loss function and so on. The aim is to compare the performances of different neural networks. In NN design even the smallest difference in the structure can significantly change the performances of the overall net. In this analysis, we propose 19 nets, 7 of which are more "classical" neural networks with fully connected layers, while the remaining ones are more complex Convolutional neural network. In the following table there is the detailed description of all the nets. For simplicity, from now on we will refer to them by their identification number (e.g. NET 1, 2, 8.1, etc.).

| Net # | Type of net |
|---|---|
| 1 | SGD, minibatch, momentum = 0.9, ReLU act. fcn, 3 fcl(512-512-10) |
| 2 | ADAM, Relu, 3 fcl (512-512-10), batch normalization |
| 3 | SGD, minibatch, momentum = 0.9, Sigmoid act. fcn, 5 fcl (512-256-128-128-10), batch normalization |
| 4 | SGD, minibatch, mom = 0.9, ReLU, 5 fcl (512-512-512-512-10), batch norm, Node dropout |
| 5 | ADAM, Relu, 5 fcl (512-512-512-512-10) |
| 6 | ADAM, ReLU, 5 fcl (512-256-256-256-10), batch norm, nodes dropout |
| 7 | ADADELTA, **MISH** act. fcn., 5 fcl (512-256-256-256-10), batch norm, nodes dropout |
| 8.1 | CNN, 2 filters (32-64 channels), MAX_pooling, 2 fcl (1000-10), ADAM |
| 8.2 | CNN, 2 filters (32-64 channels), MAX_pooling, 2 fcl (1000-10), SGD |
| 9.1 | CNN, 2 filters (64-256 channels), AVG_pooling, 3 fcl (5000-1000-10), ADAM, Relu |
| 9.2 | CNN, 2 filters (64-256 channels), AVG_pooling, 3 fcl (5000-1000-10), SGD, Relu |
| 10.1 | CNN, 2 filters (8-16 channels), AVG_pooling, 3 fcl (500-250-10), batch_norm1d, Relu, ADAM |
| 10.2 | CNN, 2 filters (8-16 channels), AVG_pooling, 3 fcl (500-250-10), batch_norm1d, Relu, SGD |
| 11.1 | CNN, 2 filters (16-32 channels + batch_norm 2d), AVG_pool, 3 fcl (512-512-10), batch_norm_id, Relu, ADAM. **Filter 1 = 8x8, filter 2 = 6x6.** |
| 11.2 | CNN, 2 filters (16-32 channels + batch_norm 2d), AVG_pool, 3 fcl (512-512-10), batch_norm_id, Relu, SGD. **Filter 1 = 8x8, filter 2 = 6x6.** |
| 12.1 | CNN, 1 filter (16 channels + batch_norm2d, NO POOLING), 3 fcl, (1024-512-10), batch_norm1d, Relu, ADAM |
| 12.2 | CNN, 1 filter (16 channels + batch_norm2d, NO POOLING), 3 fcl, (1024-512-10), batch_norm1d, Relu, SGD |
| 13.1 | CNN, 2 filters (16-32 channels, batch_norm2d, AVG_pool, MISH act. fcn, batch_norm2d), 2fcl (512-10), batch_norm1d, ADADELTA |
| 13.2 | CNN, 2 filters (16-32 channels, batch_norm2d, AVG_pool, MISH act. fcn, batch_norm2d), 2fcl (512-10), batch_norm1d, SGD |

Figure 1: Structure of the net

## 1.1 General structure of the nets (activation fcn., loss fcn., optimizer, training and testing)

Being a multi-class classification problem, the aim of the net is to predict the probability of an example belonging to each known class. In our case, the problem is to classify each image from the MNIST dataset into a specific number (0 to 9). First, some assumptions that were considered:

- **Type of loss function used**. Considering the multi class classification problem, there are 3 main loss function available and widely used: cross entropy loss, sparse multi class cross entropy and Kullback-Leibler Divergence loss.

  - Sparse cross entropy loss is more suitable for high-dimensional classification problems, where the number of possible classes is very high, like more than tens or hundreds of thousands (in our case only 10).
  - Kullback-Leibler Divergence loss is a measure of how one probability distribution differs from a baseline distribution. For this reason, usually this loss function is used in mathematical learning model where the task can be, for example, learning to approximate a more complex function than simply multi-class classification.
  - Cross Entropy is the default loss function to use for multi-class classification problems. In this case, it is intended for use with multi-class classification where the target values are in the set 0, 1, 3, ..., n, where each class is assigned a unique integer value. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized, and a perfect cross-entropy value is 0.

  After these considerations, in all the nets we have adopted the Cross Entropy function for evaluating the loss.

- **The training procedure** has been conducted on 1000 samples ad 60000 samples for each net. The dataset used is the MNIST dataset, providing 60000 pictures of handwritten number. All the pictures are of dimension $28 \times 28$. The validation has been carried out on 10000 test samples. For the purpose of the exercise, the training on 1000 samples has been carried on by using 10 epochs, while with the 60000 training dataset only 4 epochs where used. The high number of epochs for the 1000 samples training dataset is mainly due to the fact that certain optimizer like SGD have a slow convergence towards the optimum minimum point, thus more iteration during the training procedure are required. Other optimizer, like Adam, will be initially much faster and bounce around the optimum as the epochs increases. For the testing, in order to obtain a more accurate mean accuracy, we have conducted the validation of the trained net on 10000 test samples multiple times. In a for cycle, by re-initializing the test dataset and by shuffling it, we were able to obtain random batch of 10000 samples each time. The mean accuracy has been evaluated on all the accuracies from each iteration of the for cycle (i.e. from each validation procedure). The results shown proved consistent results on the first and second net, thus we decided to extend this validation procedure on all of them.

- **Optimizer used.** Nowadays, there are several optimizers available in pytorch libraries. Each of them has its own properties and advantages. However, there are 2 optimizer which are most used in the community:
  - **SGD with momentum:** this optimizer is one of the oldest one and yet, one of the most used even in the most complex NN. The usage of momentum helps SGD to navigate along the relevant directions and softens the oscillations. It simply adds a fraction of the direction of the previous step to a current step. This achieves amplification of speed in the correct direction and softens oscillation in wrong directions. This fraction is usually in the (0, 1) range. There is one problem with momentum: when we are very close to the goal, our momentum in most of the cases is very high and it does not know that it should slow down. This can cause it to miss or oscillate around the minima.
  - **AdaDelta** resolves the problem of monotonically decreasing learning rate in AdaGrad: instead of summing all past square roots it uses sliding window which allows the sum to decrease.
  - **ADAM:** also called adaptive momentum is an algorithm similar to AdaDelta. But in addition to storing learning rates for each of the parameters it also stores momentum changes for each of them separately. It offers fast convergence towards the minima with great performances.

  In this analysis, we have used mainly ADAM and SGD with momentum to highlight the speed differences between them. In net 7 and 13.1 we have adopted instead AdaDelta.

- **Activation function used.**
  - The most used activation function in NN applications is ReLU. Combining its computation efficiency (allowing the net to converge very quickly) and the non-linearity (allowing back propagation), ReLU is definitely a good choice. It also solves (partially) the vanishing of the gradient (when, for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem and as result in the network refusing to learn further, or being too slow to reach an accurate prediction).
  - Being a multiclass classification problem, on each net we used in the last layer the SoftMax activation function: it normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class. Typically, Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories (in our case we need to classify the input image as a number, 0 to 9).
  - For the purpose of the exercise, net number 3 has the Sigmoid activation function in each of its layer. Even if it is not the best choice, the idea was to show how ReLU works much better for hidden layers. Results proved that net #3 was outperformed by any other net tested.
  - In net #7 and #13.$x$ we introduce a different activation function, not implemented in Pytorch (and as a consequence, it needs to be loaded in google Colabs, see README

file for further information). The activation function proposed is called MISH (evolution of the Swish activation function). It was first published in a paper in August, 2019 by the Google Brain research team (https://arxiv.org/ftp/arxiv/papers/1908/1908.08681.pdf). Its peculiarity is that it is very similar to ReLU and according to the paper, in offers an improvement over baseline accuracy between 3% to 5% (tested on CIFAR10 dataset). The results in net #7 and #13.$x$ proved an improvement with respect to baseline accuracy between 1% to 2%.

## 1.2 General results

In the following table there are the accuracy results of the different nets. First thing to notice is that there are two main blocks of nets: from #1 to #7 there are all NN with only fully connected layer. From the #8.$x$ we have more complex convolutional NN (with convolution layer plus different fully connected layer). As expected, the CNN performed better (from 2% to 3%) with respect to the other nets.

| # net | | TRAIN on 1000 samples<br>Validation: 10000 samples | TRAIN on 60000 samples<br>Validation: 10000 samples |
|---|---|---|---|
| 1 | | 84.59% | 93.38% |
| 2 | | 87.22% | 97.67% |
| 3 | | 86.00% | 92.33% |
| 4 | | 88.91% | 97.37% |
| 5 | | 86.96% | 97.52% |
| 6 | | 88.45% | 97.73% |
| 7 | | 89.05% | 98.07% |
| 8 | 8.1 | 94.14% | 99.07 |
| | 8.2 | 91.26% | 97.78 |
| 9 | 9.1 | 10.46% | 11.29 |
| | 9.2 | 84.44% | 95.95 |
| 10 | 10.1 | 96.08% | 98.74 |
| | 10.2 | 94.11% | 98.54 |
| 11 | 11.1 | 93.55% | 99.28 |
| | 11.2 | 94.31% | 98.92 |
| 12 | 12.1 | 88.73% | 98.73 |
| | 12.2 | 91.32% | 98.74 |
| 13 | 13.1 | 95.31% | 99.41 |
| | 13.2 | 94.62% | 99.36 |

Figure 2: Accuracy results. Highlighted in green there is the net with only fully connected layer. In Blue we have the best performing CNN. In red we have the net that failed the training.

## 1.3 In-depth analysis

- **Fitting problems.** Nets trained on 1000 samples and then tested on 10000 samples ALL showed overfitting phenomena. This is mainly due to the fact that, having a tiny amount of data means that the net can learn very well to fit it, adjusting the net parameters to perfectly fit the train samples. However, when validated, all the nets showed higher validation loss with respect to the training loss. Below there are some plot of learning curves on the 1000 samples training set. All the plots are available inside the .collab script. First 3 images are from nets 1, 2, 7. The remaining are from CNN 10.2, 11.1, 13.1

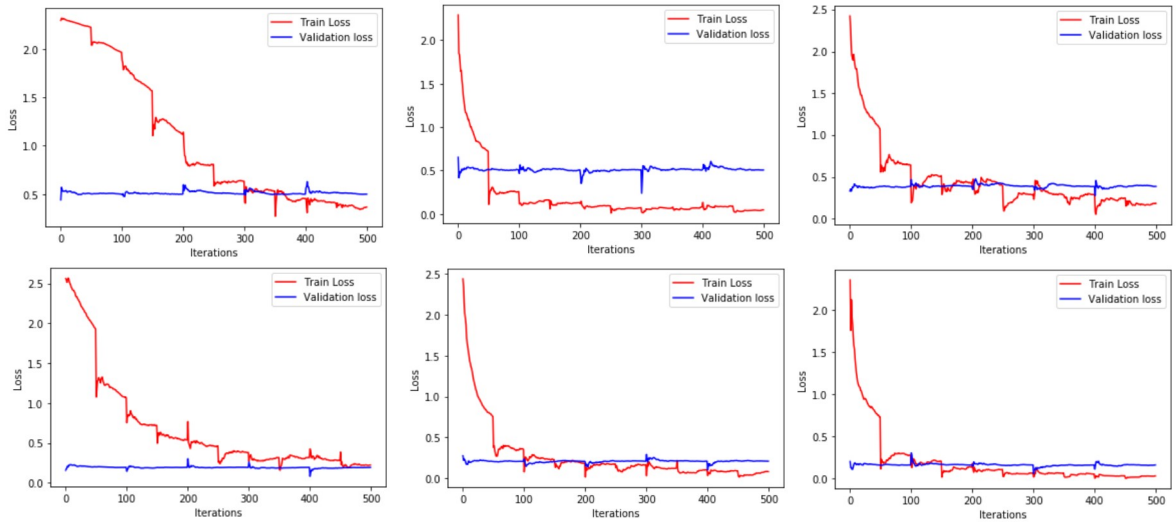



Figure 3: Training loss vs validation loss (1000 training samples) of net (left to righ, top to bottom):1, 2, 7, 10.2, 11.1, 13.1

It is worth noticing that the first 3 net (with only fully connected layers) have more fitting problems than the other 3. If we look for example at second picture (net 2, ADAM with 3 fcl and batch normalization), it is clear that the model is extremely well trained, but it has poor performances when validating it. On the other side, CNN looks to perform better even with a small amount of data and the overfitting problems are not too much big. For example, in the $4^{th}$ picture (net 10.2, CNN, 2 filters ($8-16$ channels), AVG_ pooling, 3 fcl ($500-250-10$), batch_ norm1d, Relu, SGD) the overall result is satisfactory, with not too many fitting problems. Overall, having a big dataset for training is always better, because it increases the amount of different data we can use to train our net (as results will demonstrate, nets trained on the 60000 samples dataset have no overfitting problems). With just a dataset of 1000 images, the overfitting phenomena is most likely to happen.

- **SGD vs ADAM convergence speed.** Having already explained the main differences in convergence speed between adam and sgd, the results confirm the theory. In all the nets, ADAM outperform SGD in terms of convergence speed. While SGD requires between 300 and 500 iterations to converge, ADAM only requires at maximum about 150 iterations.

Below the comparison between ADAM vs SGD speed in the net 8.1 and 8.2 (1000 and 60000 samples respectively).

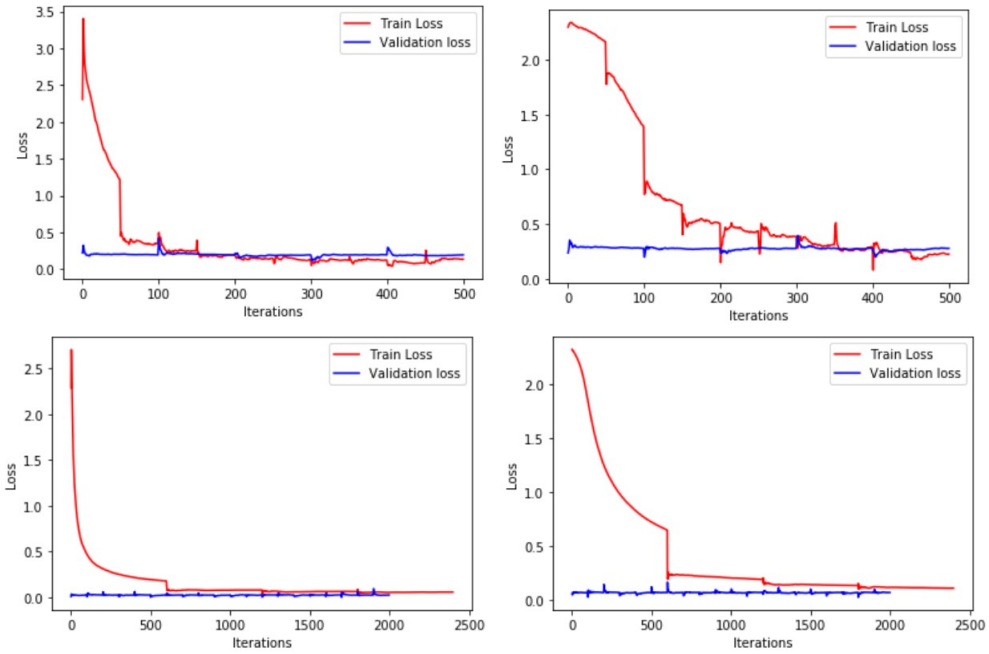



Figure 4: Performance of ADAM (left pictures) vs SGD (right pictures) on 1000 training samples (up) and 60000 training samples (down)

- **ReLU vs MISH vs Sigmoid.** As declared by the Google Brain team, MISH should perform slightly better than ReLu. If we consider a comparison between nets number 7 (with MISH) and number 6 (with ReLU), the increase in accuracy is around 1.6% for 60000 training samples. In the case of the 1000 sample training set, Relu seems to perform slightly better. However, there are too many factors that could have influenced the results. Not the results I was expecting (like the one declared in their paper), but overall the result might be influenced by the net structure, number of layers, etc.

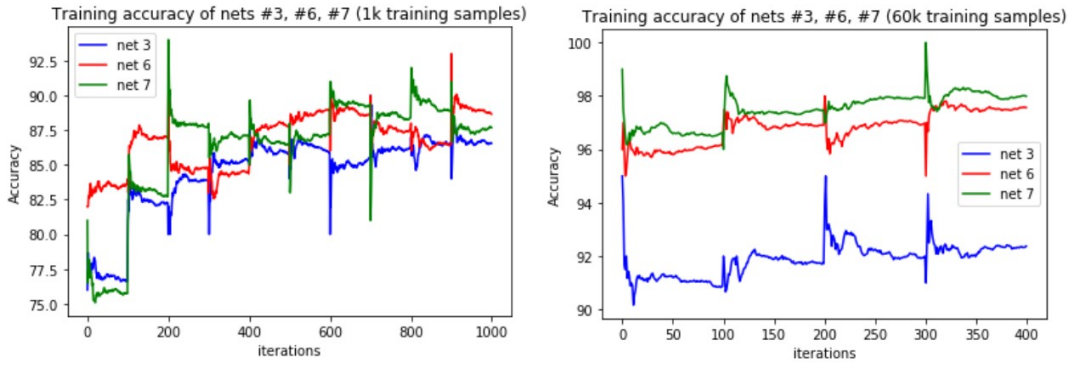



Figure 5: Performance on training accuracy of nets 3, 6, 7. 1000 training samples on the left, 60000 on the right.

For what concern the net number 3, the one using the sigmoid as activation function for all the layer, it is clear that it has been outperformed on the 60000 training samples by all the other nets. This is mainly due to the fact that the sigmoid output is an integer value 0/1, which is not very suitable for hidden layers of a multiclass classification problem.

- **Simple net VS complex net.** Not always a complex net with many layers is needed to obtain decent performances. If we look at the net number 2, we have only 3 fcl and batch normalization, but still, we manage to obtain an accuracy (on the 60000 samples training set) of 97.67%, comparable with the other nets with more layers, nodes dropout, etc. Same is for net number 5, with only 5 fcl and no other feature in the hidden layers, capable of reaching a decent 97.52% of accuracy.
  If we consider net 11.1 and we add 3 more convolution layer (5 filter in total, 16-32-32-32-32 channels) and adjust net parameters (computing the correct number of I/O), we can see that the final training accuracy has not improved with respect to the basic 2 convolution layer.

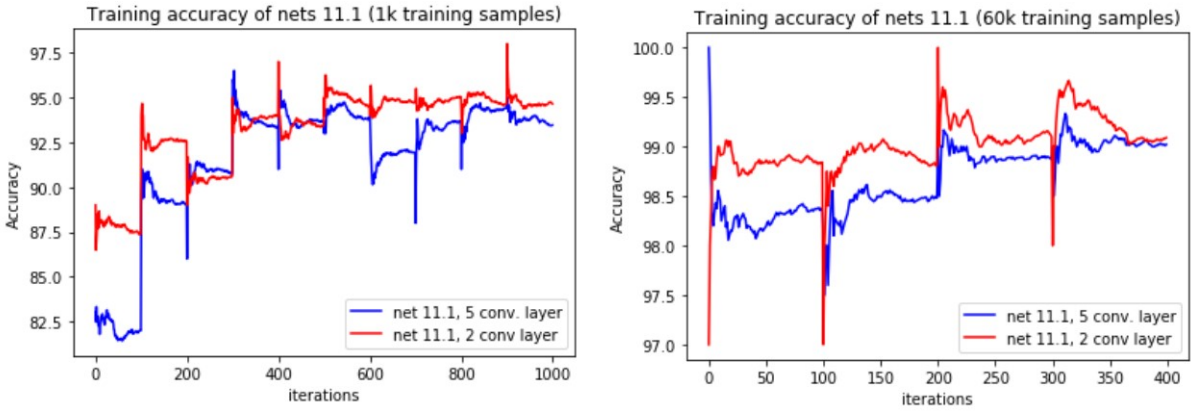



Figure 6: NET 11.1: 2 convolution layer vs 5 convolution layer. Left 1000 samples. Right 60000 training samples

In the case of the 60000 training samples, the net with 5 convolution layer is also slower in convergence, but it oscillate less around the minima. Net 11.1 with 5 convolution layer are not included in Figure 1 and 2 (results are available on the .colab file).

- **Number of channels and filter dimension**. The potential risk of designing our net overcomplicated is widely expressed by the net number 9.1. The idea here was to overcomplicate by purpose the net, making the 2 convolutions layer with respectively 64 and 256 filters. At a first, I observed the following results:
  - The one with ADAM completely failed the training, not succeeding in reaching a decent accuracy. Probably the optimizer fell in a saddle point and was not able to found the way out.
  - SGD instead managed to make the training and to achieve a decent 95.95% of accuracy (on the 60000 training samples). Still, this net has worst performances compared with net number 8.x and 10.x (where the number of channels is significantly low).

After a second training try, both nets trained well, and they didn't show any problem. But, still, results are in favour of "simpler" nets. The final results are similiar, but the net 9 with many channels is both slower in convergence and achieve less training accuracy.

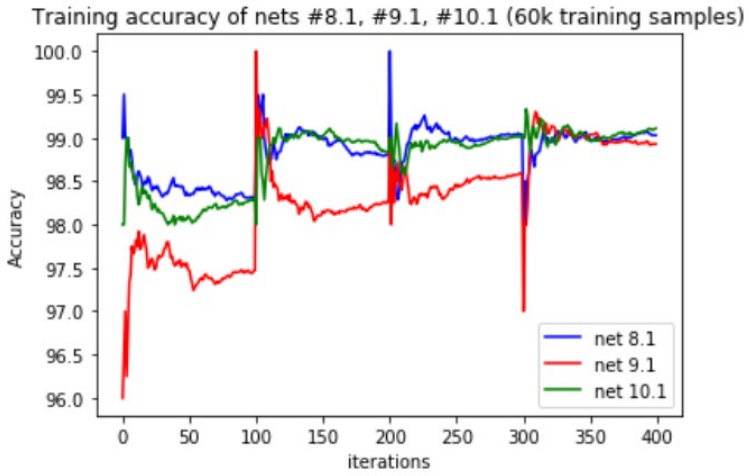



Figure 7: Performances of 3 CNN with different number of filters channels

If we look at net 11.x and net 10, we can see that the main difference is in the filters size. Net 11.x has got first filter dimension equal to 8 and the second equal to 6, while in net 10.x all filters are of dimension 5. Generally speaking, the selection of the correct filter dimension is very delicate and it strongly depends on the problem we are dealing with.

- Having a filter dimension $> 5$ means that we have a larger receptive field per layer and that we extract generic features spread across the image. Therefore, only the basic component of an image are extracted. Furthermore, fast reduction in the image dimension makes the network shallow and not working very well.
  With the size increasing, of course it increase also the number of weights in each layer. Having a too large filter means that the net becomes computationally inefficient.
- By keeping the filter dimension relatively small ($< 5$), it means that it has a smaller receptive field as it looks at very few pixels at once. As a consequence, highly local features are extracted without much image overview and therefore captures smaller, complex features in the image. Also, the amount of information extracted will be vast, maybe useful in later layers. Keeping the dimension of the filter small also mean less weights, making the net more computational efficient. However, the risk of having a net with very small filter size is to extract from an image also unnecessary features that might be resulting in slow convergence of the net.

After these considerations, assuming that in our problem we are using very simple images with a limited number of features (being handwritten number, the features extracted by the net will be very basic and simple) changing the filter size (like in net 10.x and 11.x) does not affect dramatically the results.

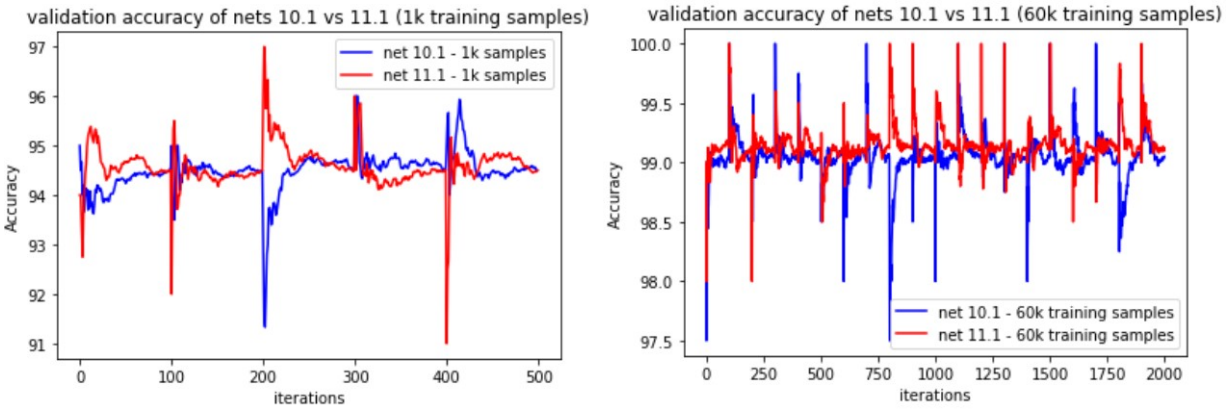



Figure 8: Validation accuracy on 10000 samples (1000 vs 60000 training samples. The two net are overall giving the same results

- **Effects on performance due to the presence of batch normalization.** The idea of batch normalization is to normalize the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. Consequently, batch normalization adds two trainable parameters to each layer, so the normalized output is multiplied by a "standard deviation" parameter (usually indicated with gamma) and add a "mean" parameter (usually indicated with beta). The main benefit of using batch normalization is a reduction of overfitting mainly because it regularizes the model. Regularization reduces overfitting which leads to better test performance through better generalization. In our analysis, by looking at the loss plots of nets without batch normalization, we can see that the phenomena of overfitting are more consistent. For example, comparing net number 5 (no batch normalization, left picture) and net number 6 (with batch normalization, right picture) we can clearly see that, on the 1000 samples training set, overfitting seems to affect less net 6 as shown below. In the left image we can see that the net is well trained with a very low loss, but when validating it, we do not obtain the results we were aiming for: the net learned to fit the training data very well, but is missing generalization capabilities when tested on the 10000 samples validation set. On the right picture we can see the opposite: the use of batch normalization has improved the generalization capabilities of the net, reducing fitting problems.
  In the case of the 60000 samples training set the results are not affected a lot by the presence of batch normalization, mainly due to the fact that, being the training dataset very large, the net learns to fit a large variety of feature. We can conclude that batch normalization is fundamental when working with a limited amount of data. Moving to CNN, presence of batch normalization seems not to affect too much the overfitting problem. This is mainly due to the fact that, in CNN, even if working with a low amount of data (e.g. 1000 samples), the presence itself of convolutional layers and filters increase (or for better saying, "augment") the data dimensionality (e.g. in net 8, we pass from an input picture of $28 \times 28$ to an output of the first convolution layer of dimension $28 \times 28 \times 16$). It is important to say that, a drawback of this "data augmentation" procedure (by implementing more filters) is that the complexity of the net gets higher and, as a consequence, training time increase by a lot. A possible solution to solve this problem is to adopt the nodes dropout technique.

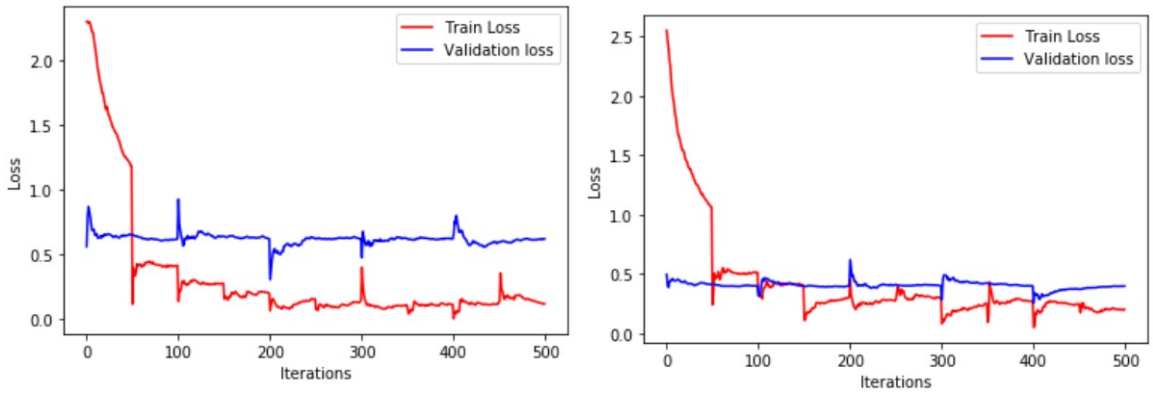



Figure 9: Left: net 5 (no batch norm.). Right: net 6 (batch norm)

- **Presence of nodes dropout.** In CNN, nodes dropout is a widely use technique to improve the performances of the net and to reduce overfitting problems. In our analysis, if we take as an example net 8 and we remove the nodes dropout, we can clearly see that the performances get worst for the net trained on 1000 samples. The reason is that the net is probably too much complex and during the training phase the data is perfectly fit, but we then we obtain worse generalization performance during validation. We can notice that, removing the nodes dropout decrease generalization capability of the net. Also, the accuracy has dropped down to 92.52% from 94.14%. Default dropout rate is set by Pytorch to $p = 0.5$ and it has been not changed.

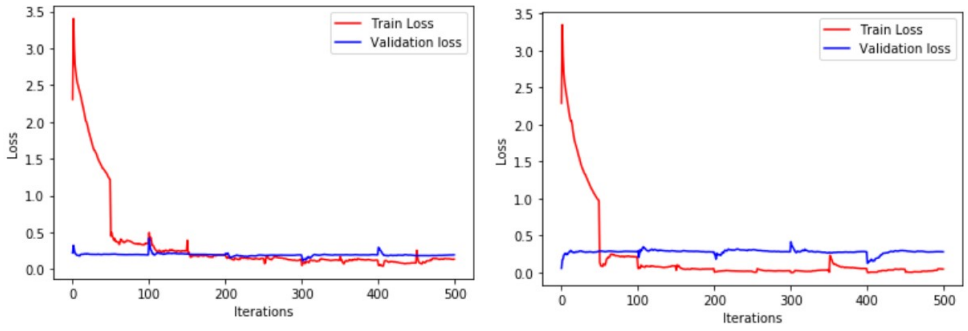



Figure 10: Left: net 8.1 (drop-out). Right: net 8.1 (no drop-out). Both are referring to 1000 training samples case

## 1.4 Summary

Neural network architecture can be very tricky, and as it was shown in this first chapter, even the smallest change can make the network working much better or much worse. There are still several improvements that can be done to all the nets. For sure, one further approach to improve the validation accuracy is to use a more suitable *Cross validation approach*. In the next chapter we are going to talk about another important aspect of Neural Networks, which is of course *data*.

# 2 Chapter 2
# Importance of data in Neural Networks

Aim of this chapter is to underline the effects that different data augmentation techniques can have in neural networks.
In an ideal case, the best situation when dealing with the training of neural network is definitely having a very big amount of data. The net can train over many epochs and can understand how to fit the data. However, in many other real-life case (like for example in medial applications) the amount of data is very limited and does not offers a big variety of samples. Thus, data augmentation is a fundamental technique that helps a lot the training procedure of neural networks.
The task is to analyse the effect of different data augmentation techniques on a limited dataset, in our case the MNIST data set, limited to 1000 images. The comparison will be made with the old nets designed for assignment 1.

## 2.1 Data augmentation explained

What is exactly data augmentation? Data augmentation is a strategy that enables to significantly increase the diversity of data available for training models, without actually collecting new data. Data augmentation techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks. However, most approaches used in training neural networks only use basic types of augmentation.

Figure 11: Basic example of data augmentation. Pictures are just examples, not taken from MNIST dataset.

As already said, the most effective way of augmenting data is to apply very basic *Affine transformations*. An *affine transformation* is a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation. The affine transformation technique is typically used to correct geometric distortions or deformations that occur with non-ideal camera angles. For example, satellite imagery uses affine transformations to correct for wide angle lens distortion, panorama stitching, and image registration. Transforming and fusing the images to a large, flat coordinate system is desirable to eliminate distortion. In our case, the aim is the exact opposite: instead of correcting the images, we want to manipulate them and modify them in order to enlarge the dataset and obtain a bigger variety of samples.
Examples of affine transformation can be (Figure 11): (1.1) Original image, (1.2) Horizontal flip, (1.3) Pad and crop, (1.4) Rotate.

## 2.2 Types of nets and number of epochs used

Two different nets have been used in order to compare the results of different transformations. Both nets were designed in chapter 1 and have been modified with the transformation in this case:

- Net 1: NN with 5 layer, SGD for gradient computation, cross entropy and ReLu activation function. Below the structure of the net:

```
simple_network(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (bn1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout1): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (bn2): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc3): Linear(in_features=512, out_features=512, bias=True)
  (bn3): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc4): Linear(in_features=512, out_features=512, bias=True)
  (bn4): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc5): Linear(in_features=512, out_features=10, bias=True)
)
```

Figure 12: Net number one

- Net 2: CNN with 2 convolution layer, 3 fcl, ADAM for gradient computation, cross entropy and ReLu activation function. Below the structure of the net:

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 8, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (layer2): Sequential(
    (0): Conv2d(8, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (drop_out): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (bn2): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=500, out_features=250, bias=True)
  (bn3): BatchNorm1d(250, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc3): Linear(in_features=250, out_features=10, bias=True)
)
```

Figure 13: Net number two

Both the net were already tested in the previous chapter and gave the following accuracy results:

Net 1: validation accuracy on 1000 training samples = 88.91%
Net 2: validation accuracy on 1000 training samples = 94.73%

It is worth saying that only 20 epochs were used for both nets for training on 1000 samples (**not augmented**) of the MNIST dataset. Here, in all the nets 100 epochs have been used for training procedure. This is mainly due to the fact that, thanks to data augmentation, in each

epoch we will have a different batch of transformed pictures. Thus, enlarging the number of epochs is the same of training the net on a bigger variety of images. Without data augmentation instead, increasing the number of epochs after a certain threshold do not affect the accuracy results, due to the fact that the net is continuously training on the same 1000 samples.

## 2.3 Types of transformations used

Eight different type of transformations have been tested on the two different nets, plus the bonus transformation ***"MixUp"*** has been taken from the provided script and implemented in the two nets. From now on, for simplicity, we will refer to each transformation as $T_i$, $i = 1 \dots 8$. Below all the types of transformations:

- T1: transform.ColorJitter(brightness=[0.4, 1.6], contrast=[0.4, 1.6], saturation=None, hue=None)

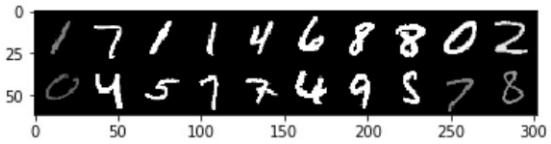



Figure 14: T1 samples

- T2: transform.RandomAffine(degrees=(-20, 20), translate=(0.1, 0.2), scale=(0.8, 1), shear=[-30, 30, -30, 30])

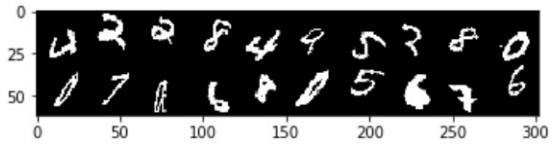



Figure 15: T2 samples

- T3: transform.RandomAffine(degrees=(-20, 20)) (only rotation)

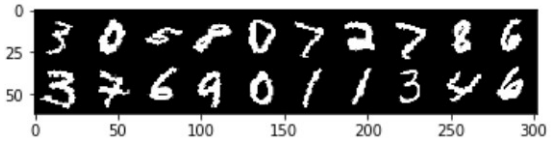



Figure 16: T3 samples

- T4: transform.RandomErasing(p=0.6, scale=(0.02, 0.1), ratio=(0.3, 0.5), value=0, inplace=False)

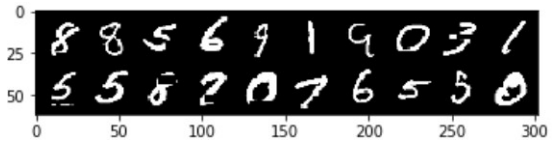


Figure 17: T4 samples

- T5: transform.RandomPerspective(p=0.5)

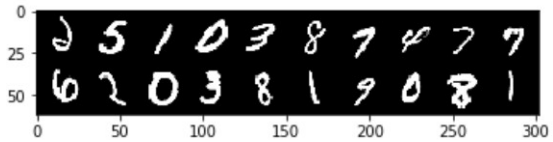


Figure 18: T5 samples

- T6: transform.RandomVerticalFlip(p=0.5)

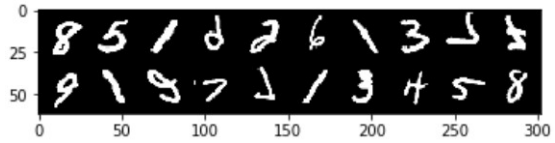


Figure 19: T6 samples

- T7: transform.RandomAffine(degrees=(0, 0), shear=[-30, 30, -5, 5]) (Only shear)

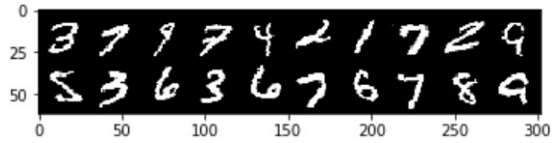


Figure 20: T7 samples

- T8: transform.RandomPerspective(p=0.4), transform.RandomAffine(degrees=(-20, 20), shear=[-20, 20, -5, 5]) (shear-Rotation-perspective)

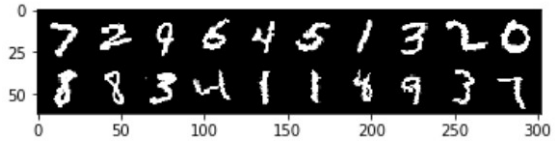


Figure 21: T8 samples

| | Accuracy | | Improvement (%) | |
|---|---|---|---|---|
| | Net 1 | Net 2 | Net 1 | Net 2 |
| T1 | 90.63% | 95.99% | 1.72% | 1.26% |
| T2 | 88.23% | 96.62% | -0.68% | 1.89% |
| T3 | 93.08% | 96.68% | 4.17% | 1.95% |
| T4 | 89.92% | 95.44% | 1.01% | 0.71% |
| T5 | 93.65% | 96.19% | 4.74% | 1.46% |
| T6 | 87.73% | 93.49% | -1.18% | -1.24% |
| T7 | 92.81% | 96.39% | 3.9% | 1.66% |
| T8 | 94.40% | 96.77% | 5.49% | 2.04% |
| T9 | 91.47% | 91.40% | 2.56% | -3.33% |

Table 1: Accuracy results over tested nets

## 2.4 Results

Above there are the accuracy results of the 2 nets tested on the different transformations:
The best transformation that gives the better accuracy and the biggest improvement over the old net is the T8. Accuracy results were obtained via multiple validation over a 10000 samples test set. To obtain better mean accuracy, validation has been carried many times (50 epochs). The worst performing transformation method is definitely T6, which decrease the performances of about 1.2%.

## 2.5 Analysis of results: performances over different transformations

### 2.5.1 T1

Using only color jitter on B&W picture is definitely not the most useful data augmentation technique (because only brightness and contrast are going to be changed, Figure 14), but still both nets performed reasonably well, with a small increase on reference accuracy of about 1% each. However, since the basic feature of the images are untouched (like rotation, translation, etc.) but only a change in brightness and contrast is applied, the nets still show an overfitting behaviour over 100 epochs. This is probably due to the fact that this data augmentation technique does not change or augment in any useful way the 1000 sample training set. Thus, during the training the variety of the samples is not increased a lot. Below the performances of the two net with T1.

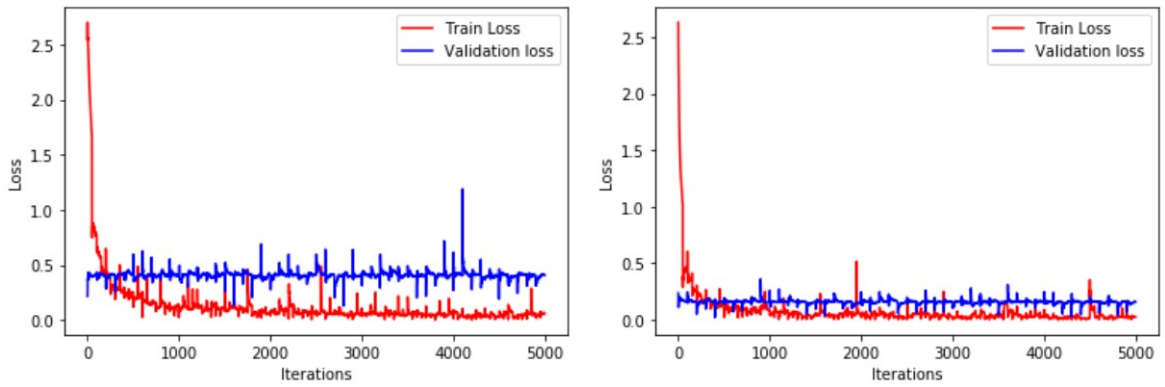



Figure 22: T1: Net 1 on the left, Net 2 on the right. CNN on the right shows less overfitting behaviour with respect to the classic NN on the left.

### 2.5.2 T2

This transformation was supposed to show the effect of trying to manipulate and augment the data too much. By setting the parameters of the *transform.RandomAffine* quite high and *extreme* (in Figure 15 the effects of increasing too much the parameters of affine transformations are definitely clear, with some digits being unrecognisable even by human).
As expected, net 1 is not capable of training well on this augmented dataset and the performance decreases by $-0.68\%$. Net 2 still manages to train well, with a decent increase on performances of 1.89%. This is probably thanks to the fact that the convolution layers still manages to extract useful features from the images, even if they are highly modified (and some unrecognisable). Furthermore, net 1 shows an highly varying loss through all iterations (due to the fact that the Random.Affine apply randomly different transformations defined by the ranges declared in the brackets and thus generating a highly varying dataset) and an underfitting behaviour: This is due to the fact that the net performs much better on the validation set (not augmented neither modified with 10000 validation samples) with respect to the training one. Net 2 shows similar results of net 1, but with less underfitting. Below the performances of the two nets with T2.

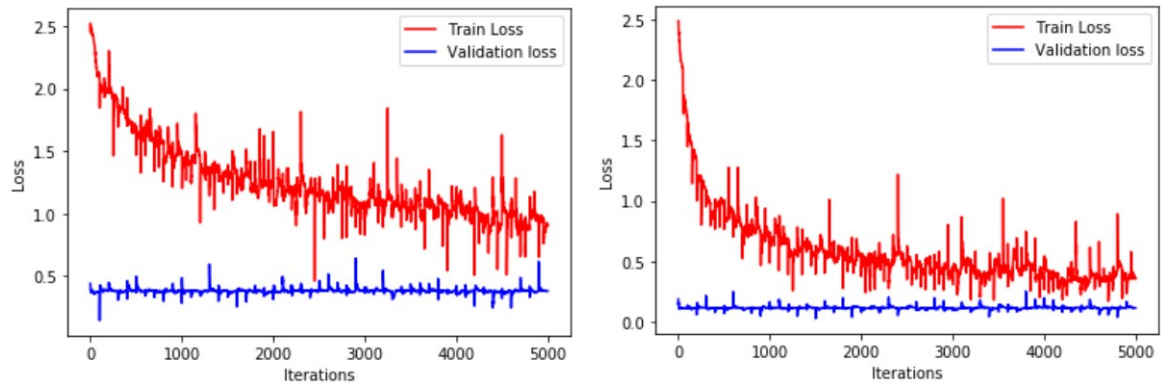



Figure 23: T2: Net 1 on the left, Net 2 on the right. NN on the left shows underfitting behaviour and highly varying loss. Same for the CNN on the right, with less overfitting.

### 2.5.3 T3

In contrary to what founded with T2, T3 is definitely performing much better. By applying only rotation with Random.Affine, both net performed very well with an accuracy of 93.08% (+4.17% w.r.t. reference) for net 1 and 96.68% (+1.95%) for net 2.
In this case the rotation was limited by $[-20°, +20°]$: this choice is quite reasonable in order to not make confusion with numbers like 6 and 9 (T6 will show the danger of confusing them). If we consider the fact that the MNIST dataset contains hand-written digits, by a first analysis it seems reasonable to apply a random rotation to the samples, in order to rotate them and to correct eventually bad-written digits.

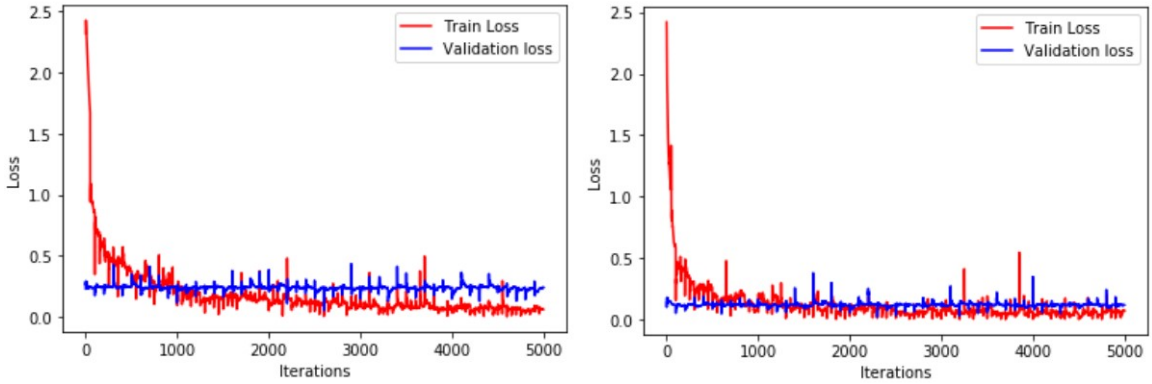



Figure 24: T3: Net 1 on the left, Net 2 on the right. Both net behave very well, without too much overfitting and loss spikes.

### 2.5.4 T4

The concept of T4 is very simple: erase a portion of the sample with a probability p (in our case $p = 0.6$). The portion erased is defined in the range $[0.02, 0.1]$. The scale values are small due to the fact that the images from the MNIST dataset are $28 \times 28$ picture, B&W, with only numbers and any other background noise. Increasing too much the dimension of the erased portion means loosing too many valuable informations and features from a single sample. After that, both nets performed well with an accuracy of 89.92% and 95.44% respectively (improvement around 1% for both of them).

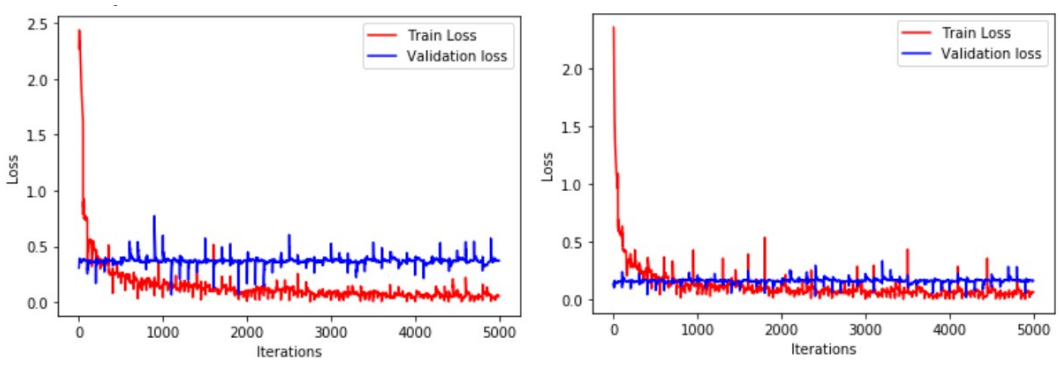



Figure 25: T4: Net 1 on the left, Net 2 on the right. Both showed some underfitting

If we look at the above Figure 25, we can clearly see that both the net showed some overfitting. In this case, it probably means that the nets have trained very well on the augmented dataset, but performed not as expected during validation. Thus, the scale factor can probably be increased a little bit more, in order to augment even more the data and to increase the variability of the data.

### 2.5.5 T5

*transform.RandomPerspective()* is another transformation that performed very well. Again, if we think to the nature of the dataset, it is reasonable that the hand written numbers are not perfectly symmetric and all shaped in the same way. Furthermore, this makes even more sense if we think to the fact that the hand written number were probably written by both right handed and left handed persons, changing inevitably the shape. Everyone has its own writing style and every number was probably written in a different way (e.g. the number 6 can both written in a clockwise or counter-clockwise way). Thus, by applying a transformation that randomly change the perspective of the sample might be affecting in a good way the net performances. Accuracy results actually confirmed the suppositions stated above. Net 1 performed incredibly well with an accuracy of 93.65% (improvement of 4.74%) while net 2 achieve 96.19% of accuracy (improvement of 1.46%).

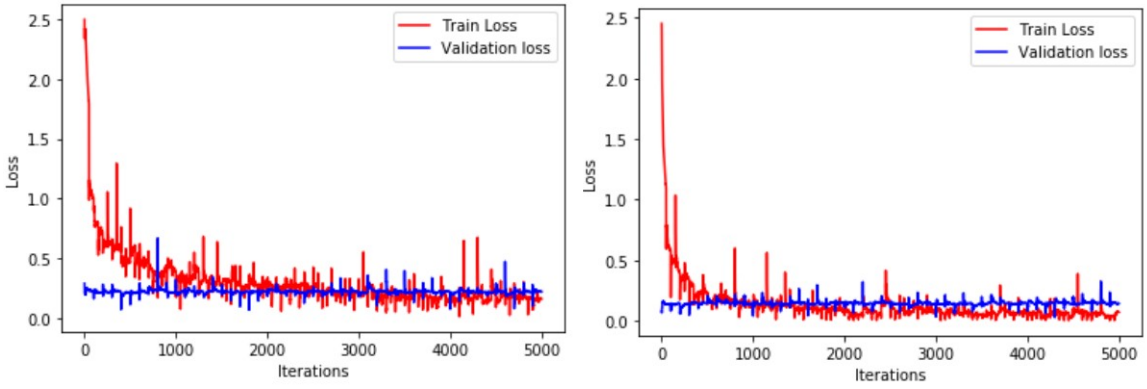



Figure 26: T5: Net 1 on the left, Net 2 on the right. Both net trained incredibly well and achieve good accuracy results.

Furthermore, from Figure 26 it can be seen that neither the first or second net showed overfittig phenomena and the loss is not oscillating too much.

### 2.5.6 T6

T6 is a clear example of the danger of not applying wisely transformations to a given dataset. The vertical flip might be very a useful transformation in many other classification problem, but a very important condition is that for any reason the samples in the dataset must not have any symmetry. By applying a random vertical flip (see Figure 19 for examples) to the MNIST dataset all the numbers 6 and 9 in the dataset are confused (or for example 7 and 1). The net will systematically get the wrong prediction for both the 6 and the 9. As results, Net 1 decreased its performances to 87.73% (-1.18%) and Net 2 decreased to 93.49% (-1.24%).
Both show overfitting, meaning that during training the nets try to fit in the best way the flipped number (like 6 or 9), but then during validation performs much worse.

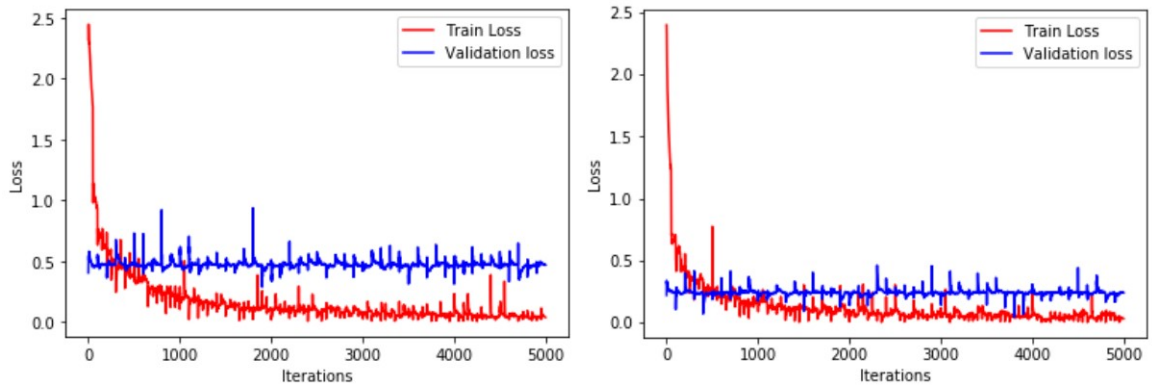



Figure 27: T6: Net 1 on the left, Net 2 on the right.

### 2.5.7 T7

Recalling the idea and the benefits of manipulating the shape of the picture expressed in section 6.5, applying only a random shear transformations to numbers may help improving the capability to recognize distorted numbers, (like a 1 which is not perfectly straight or a 3 written very quickly). As expected, both nets performed well even with this transformation. Net 1 reached 92.81% of accuracy (+3.9%) and Net 2 reached 96.39% (+1.66%).

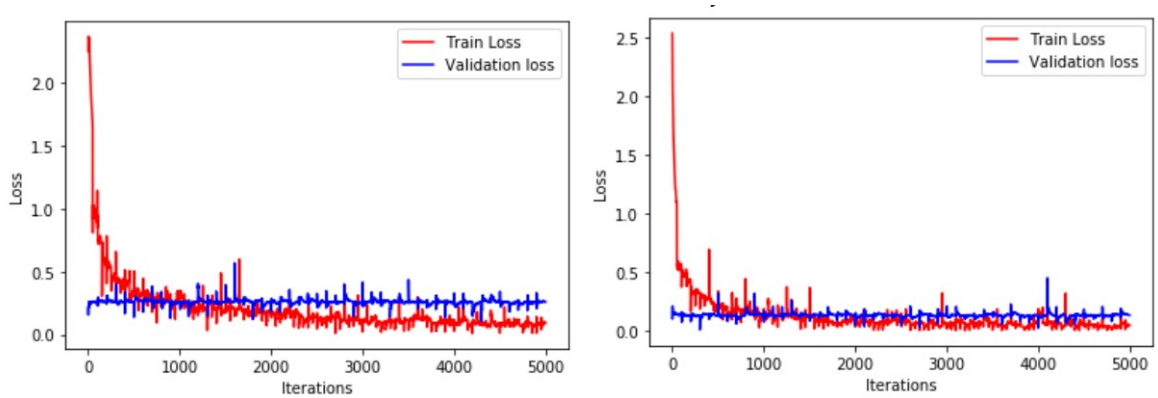



Figure 28: T7: Net 1 on the left, Net 2 on the right.

Again, with this transformation there is no significant overfitting phenomena in both nets.

### 2.5.8 T8

It is quite clear at this point that transformations that manipulate the shape and symmetry of numbers (like only rotation - T3, randomPerspective - T5 and Shear - T7) increase by a lot the performances of the nets. With this concept in mind, transformations 8 include all of the previous mentioned T3,T5,T7. As results, with this set of transformations I obtained by far the best result (in term of accuracy and improvement) between al the transformations tested. Net 1 reached an incredible 94.40% of accuracy (which, for a net of only fully connected layers and trained with only 1000 base samples, is a lot) with an improvement of +5.49%. Net 2 achieved an accuracy of 96.77%, with an improvement of +2.04%. Both Net 1 and Net 2 have not shown any overfitting. However, applying 3 transformations at once means that the variety of the

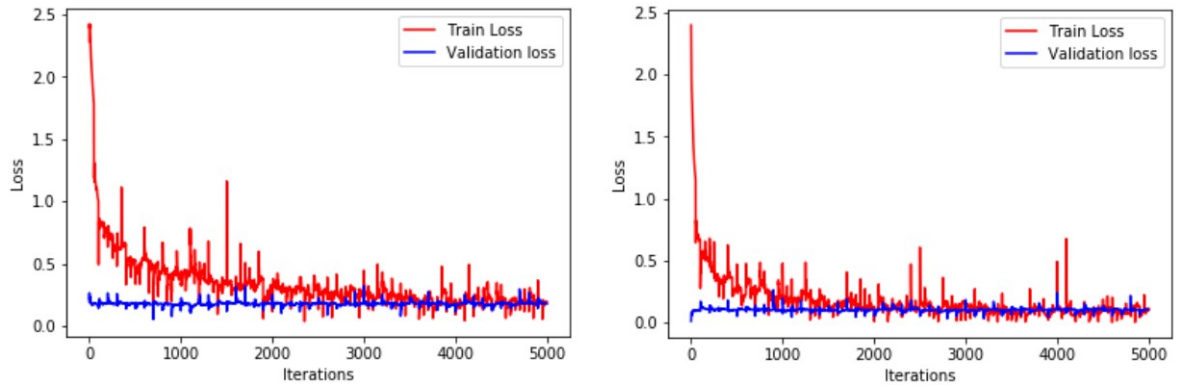



Figure 29: T8: Net 1 on the left, Net 2 on the right.

samples are increased a lot. As results, along the 100 epochs, the loss seems to be oscillating more with respect to T3, T5 and T7.

### 2.5.9 T9

Mixup is a neural network training method that generates new samples by linear interpolation of multiple samples and their labels. Below a practical example of how MixUp augment the dataset.

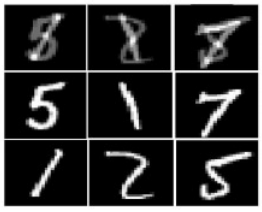


Figure 30: Example of MixUp: the numbers are taken from the MNIST dataset. First row shows the augmented data generated by this augmentation procedure.

The idea is that, by merging with a factor $\epsilon$ two samples, the net during training learn to recognize some confused digits (like the first one which has a 5 and a 7 mixed together). Then, during validation it should perform much better with clear digits and no ambiguities made on purpose.
Even if the MixUp algorithm proposed in the script was not the proper implementation of the real MixUp technique presented by Hongyi Zhang in *mixup: Beyond Empirical Risk Minimization* [ICLR 2018], it still performed reasonably well, with an accuracy of 91.47% for the first net (improvement of 2.56%). However, for what concern the second net, the performances decreases by a lot: accuracy of the second net is around 91.40% (decrease of -3.33%).
First thing that can be noticed is the presence of very high oscillations of the loss during training. This is probably due to the fact that the net struggles to train and requires many parameters update (due to the intentional ambiguity introduced by the MixUp), making the loss varying and oscillating a lot. However, on the other side, there is no sign of overfitting.
It is still not very clear from the mathematical point of view why MixUp should help and improve the net. However, from a practical point of view, there are two fundamental advantages:

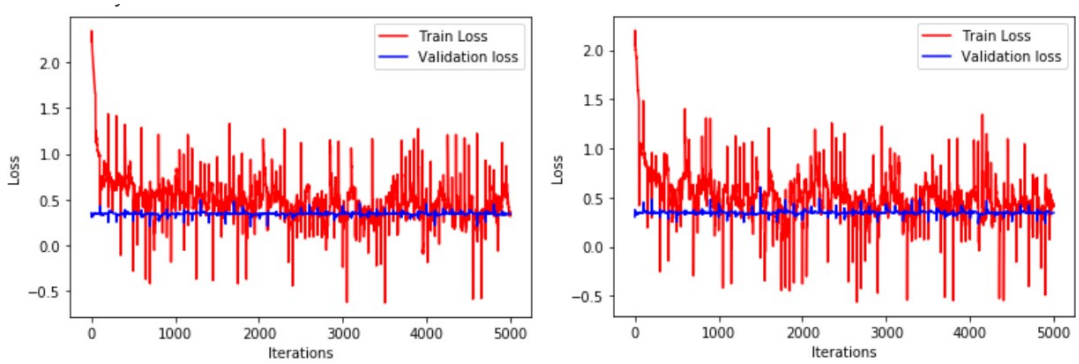



Figure 31: T9: Net 1 on the left, Net 2 on the right.

- It makes decision boundaries transit linearly from class to class, providing a smoother estimate of uncertainty.
- It reduces the memorization of corrupt labels.

## 2.6 Final considerations

Overall, T8 was the best performing set of transformations. If we compare the results of the two net with the old results from chapter 1 we can give some final considerations.

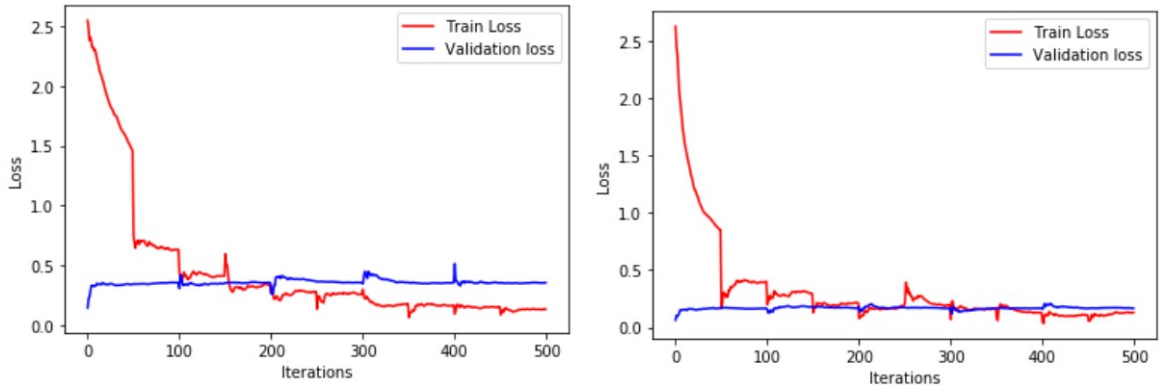



Figure 32: Performance of Net 1 (left) and Net 2 (right) from assignment 1. 1000 training samples, no data augmentation

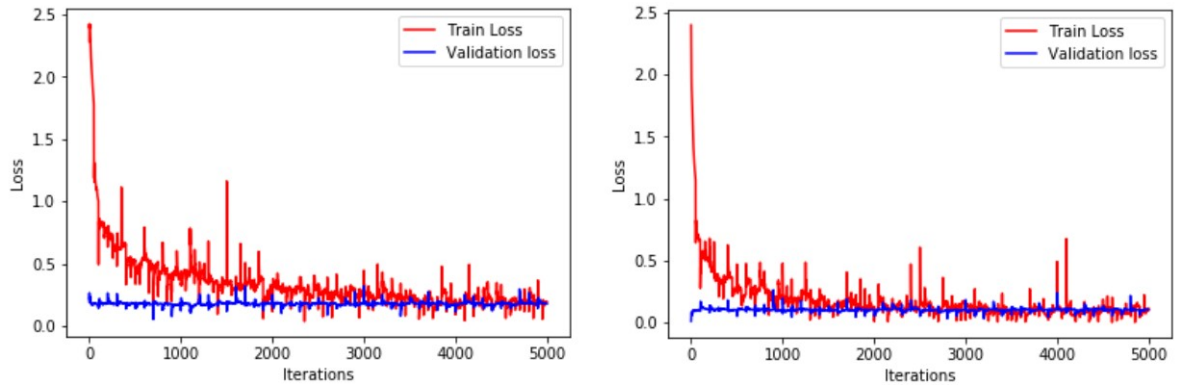



Figure 33: Performance of Net 1 (left) and Net 2 (right) with data augmentation (1000 samples training set)

First thing to notice is that in net 1 overfitting has disappeared. If we consider the fact that net 1 uses SGD, this actually makes lot of sense: the slower convergence of SGD combined with a big augmented dataset and many epochs gives the possibility to the net to train well and to perform as expected during validation. On the other hand, net 2 (CNN with 2 convolution layer and 3 fcl) seems to be overall less affected by the data augmentation, but still improved of 2.04%.
As a final considerations, we need to have in mind the overall structure of the MNIST dataset: the handwritten digits have been already pre-processed and the number itself is noise-free (no background noise coming from paper, other words or numbers, etc.). Thus, in our academic analysis data transformations can be applied with the only intent of augmenting the dataset and increasing the variety of data.
In the next chapter we are going to give an introduction to GAN (generative adversarial NN) and deduct some interesting features.

# 3 Chapter 3
# Generative Adversarial Neural Networks

Generative adversarial networks (or GAN) are a typology of NN that use the principle of unsupervised learning in order to achieve several possible different goals, like for example detection of anomalies or generation of new data. In this chapter, we are focusing on the second aspect. The goal is to study the effects of different noise vectors z (also called *latent variable*, white noise with Gaussian distribution) and how the interpolation of two of them can generate a new digit.

## 3.1 How a GAN works

The main idea is to have two nets working and training in an Adversarial way: Net G (usually called Generator) learn how to generate from a random noise vector z an output that most resemble a digit. On the other side, net D (usually called Discriminator) get as input the output of net G or a real image and it is capable of detecting wheter the image is fake or not. Net G will try to fool as much as possible net D, while Net D will try to understand as much as possible if the input image is fake or not. Net G will improve its output, such that net D is fooled.
In our case, after training the two nets (task that for GAN requires more time than usual due to the complexity), we are able of generating random number by simply giving to net G as input a random noise vector z (Gaussian distributed).

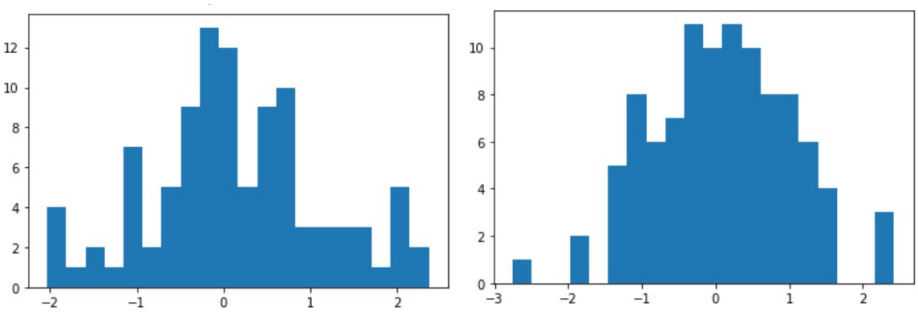



Figure 34: Histograms of the Gaussian distribution of the two white noise vectors. Z1 on the left, Z2 on the right

But what happen if we interpolate two different z vectors ($z_1$ and $z_2$)?
The interpolating functions that generates the new vector $z_3$ is defined as follow:

$$z_3 = az_1 + (1 - a)z_2 \tag{1}$$

where a is a defined parameter between 0 and 1. By iterating in a for cycle the different values of $a = 0, 0.1, \ldots 1$ we obtained ten different digits. Below the explanation and results of the code.

- First, select two random $z_1$ and $z_2$. Save the two tensors in two separated files, in order to handle them easily

- **Important remark:** executing the script multiple times means that the files *z1.py* and *z2.py* are overwritten with new digits. Thus, results in the script might be different from the ones below.
- Show the two images generated by the two random noise vector

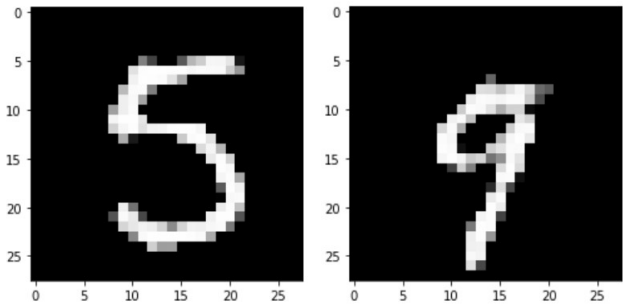


Figure 35: On the left the digit generated by input $z_1$, on the right the one generated by $z_2$

- In a for cycle, update at each iteration the value of the interpolation parameter.
- Plot the results.

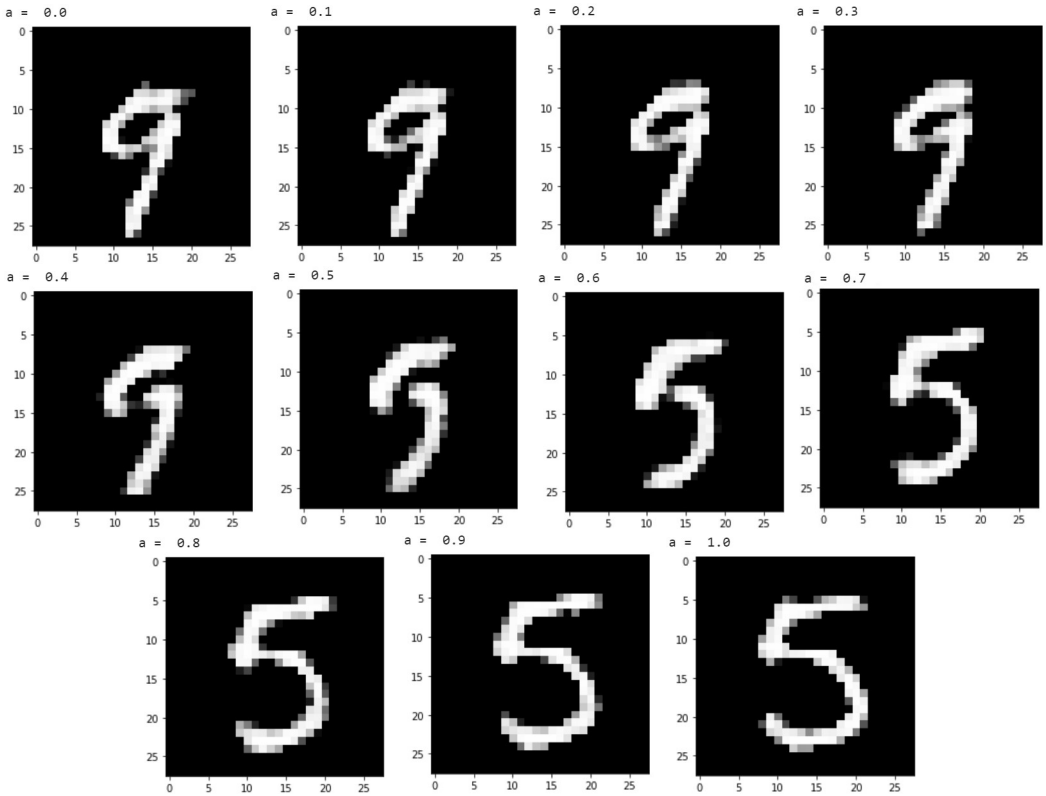



Figure 36: From left to right, top to bottom. The evolution of the interpolation parameter during the ten iterations. At a = 0.5 (second picture, second row), the digit is barely recognizable.

If we try to reload new random vectors z1 and z2, we obtain, for example the random digits in Figure 4. Vector z1 generated digit number 2 and vector z2 generated digit 0. By iterating as previously done, we obtained the following results:

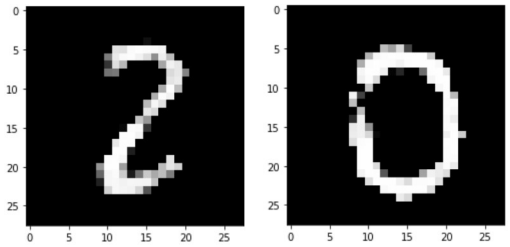


Figure 37: On the left the digit generated by input $z_1$, on the right the one generated by $z_2$

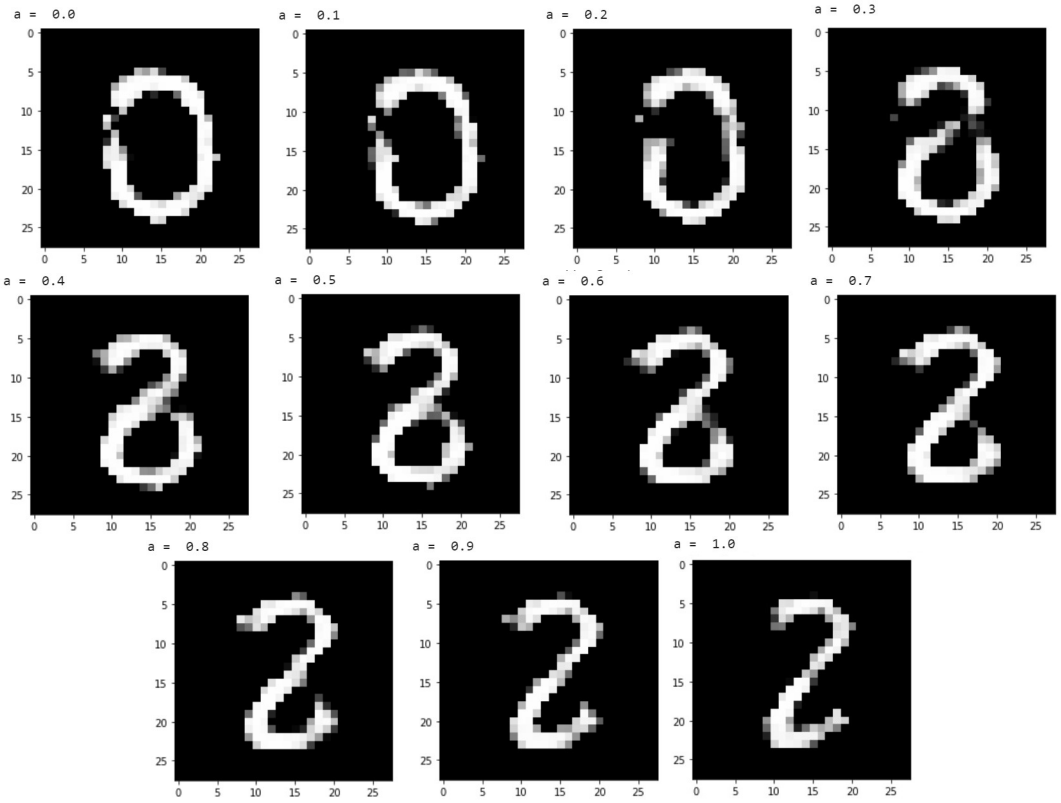



Figure 38: From left to right, top to bottom. The evolution of the interpolation parameter during the ten iterations.

At iteration a = 0.3 the digit has inside the feature of the zeros and the two. The number from a = 0.3 to a = 0.7 is more likely to be recognized as an 8, due to the fact that is the mix of both digits.

## 3.2 Understanding GAN: visualization of the *latent space*

The aim of this second analysis is to understand the effects of generating 1000 samples and 10000 samples each time from a different random noise vector of dimension $z = 100$. In order to carry out this analysis we need to introduce the concept of *dimensionality reduction*.
Usually, when dealing with neural networks, it is very useful to visualize what my network is doing and how it is performing. However, when dealing with high dimensionality data, visualizing all the features my net is learning is very difficult - and almost impossible-. If we consider for example our case, the dimensionality of the feature space is

$$28 \times 28 \times (number\ of\ samples) = 784N$$

Each dimension of this space can be addressed to each single pixel of the picture, multiplied by the number of samples we are using. It is clear that, trying to visualize a space of dimensionality $784 \times N$ is impossible.
Here's why, in order to understand better how our net is performing and to check the results we are obtaining, it is very useful to apply some *dimensionality reduction technique* on the feature space. The target dimensionality of course is usually $\mathbb{R}^2$ or $\mathbb{R}^3$ in order to be visualized in Cartesian space.
There are several ways of how to perform dimensionality reduction techniques:

- Feature Elimination: by simply eliminating some features, the dimensionality can be reduced. However, precious informations might be lost
- Feature Selection: apply some statistical test on the feature space and select the ones most suitable for the situations. However also in this case valuable feature might be discarded.
- Feature Extraction: new features are created as a combination of the input ones. These techniques can further be divided into linear and non-linear dimensionality reduction techniques.

There are several dimensionality reduction techniques, each one of them suitable for the situation we are dealing with, like *Principal Component Analysis* and *t-Distributed Stochastic Neighbour Embedding (t-SNE)*. Further informations about the working principle of tSNE is available at the original paper by *Maaten, Hinton: Visualizing Data using t-SNE* (http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf).
In this analysis we have used the t-SNE algorithm implemented in the *sklearn* libraries.
Two main analysis of latent space has been carried out:

- Visualization of the latent space of the G(z). The idea was to visualize how our pre trained GAN was performing, trying to visualize all the clusters of the generated outputs.
- Visualization of the latent space of the noise vector input z.

Both analysis, as requested, have been carried out in two cases: 1000 random z vectors ad 10000 random z vectors.

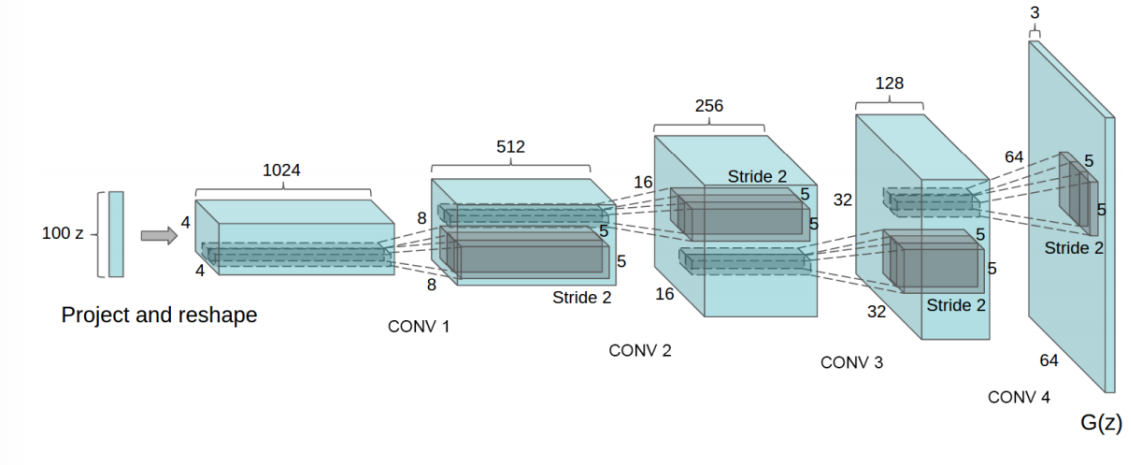



Figure 39: Structure of generic GAN. We will focus on plotting the latent space of z and G(z)

### 3.2.1 Main steps followed

1. Load and train a good network for classifying the generated images from the GAN. In our case we used net 11.1 (from chapter 1) with the following characteristics:

   CNN, 2 filters (16-32 channels + batch_norm 2d), AVG_pool, 3 fcl (512-512-10), batch_norm_1d, Relu, ADAM. Filter 1 = $8 \times 8$, filter 2 = $6 \times 6$.
   Average validation accuracy: **99.28%**

2. Load the pre-trained GAN with dim(z) = 100. Check for file integrity.

3. Generate in a for cycle 1000 random vector z. At each iteration, classify the produced image G(z) with the validation net.

4. Store the obtained classification labels for final plotting.

5. Store both G($z_i$) and $z_i$ in two separate vectors for further analysis.

6. Iterate until maximum number of iterations is reached.

7. Reshape and re-arrange vectors containing the images and the z vectors in order to be compatible with TSNE function built-in sklearn libraries.

8. Perform dimensionality reduction. Store the obtained results.

9. Plot the obtained vectors in 2d cartesian space. Notice that I used the *zip()* function instead of 3 nested for loops (1 for scrolling output vector of tSNE algorithm, 1 for selecting the correct classification label and 1 for plotting the legend).

### 3.2.2 Results: 1000 z samples

### 3.2.3 Latent space of generated images

The main idea here is to trying to visualize how our GAN has performed in generating digits from random z vectors as inputs. One effective way of studying the performances of the net is to visualize the clusters produced by the GAN. TSNE will reduce the feature space of the images from $784 \times (N = 1000)$ to a vector of dimension $2 \times 1000$, which can be plotted in Cartesian space. The resulting plot can be analysed as follows:

- Given the input vector of images, TSNE will extract out of the 784 feature only 2. Each (x,y) point in the output vector can be seen as a classified digits.
- The great thing of TSNE is that each region of the (x,y) space is occupied by a specific cluster. This is obtained thanks to the working principle of TSNE.
- The idea is to obtain visible clusters (or conglomerates) of points, depending on the corresponding label.
- The *distance* between two point can be seen as the transition from one cluster to another one, and *how much* a digit is differing from another one.

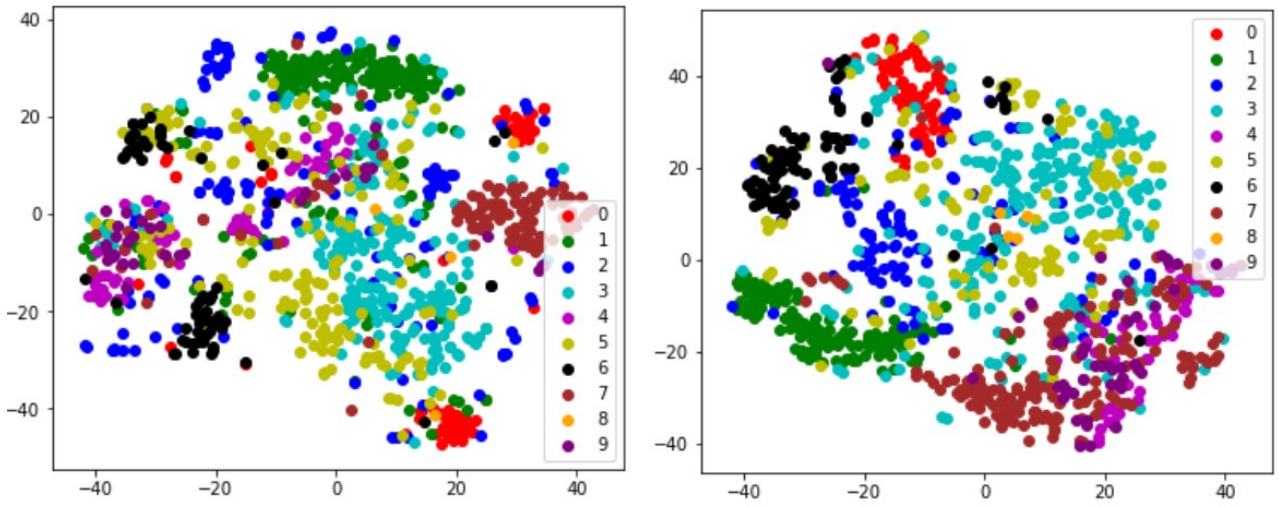



Figure 40: Two latent space of G(z) generated with 2 epochs. dim(z) = 100 and number of samples = 1000.

By looking at Figure 40 we can give further explanation of the results.

- It is now clear the concept of clusters and distance between clusters. If we look for example at digit 1 (green color), all the samples classified as ones occupies a very compact region of the latent space, meaning that the GAN has generated very good and recognizable digits 1. Same for digit zero (red color) and digit 7 (brown color).
- Worst results were given by digits 3,4,6,9. The cluster region is more spread, meaning that the difference between these digits and others is less marked, meaning that the GAN has struggled a little bit generating consistently these digits.

- Digit 2 is spread all over the region in left Figure 40, meaning that most probably the digit number 2 was generated wrongly by the GAN. On the right side however, with different 1000 samples, digits 2 seems to have been generated much better. The blue cluster is far more compact than the left figure.

Next step is now to visualize the latent space of the vector z and to analyse the different clusters depending on the random vector z.

### 3.2.4 Latent space of z

The idea is to obtain something similar to the latent space of validation outputs. In the other hand, the space actually represents the distributions of different random vectors z with respect to corresponding produced images. The distance between two point of the plot represents how much a specific vector differs from another one. Theoretically, we are expecting to obtain some clusters for different vectors z depending on which digit the vector generates.
The vector z is defined as a random variable normally distributed ($z \sim N(0, 1)$) of dimension 100. Thus, all the values in the vector z will be generated according to this distribution. By applying tSNE on all the 1000 random vectors generated and trying to cluster them gave the following results:

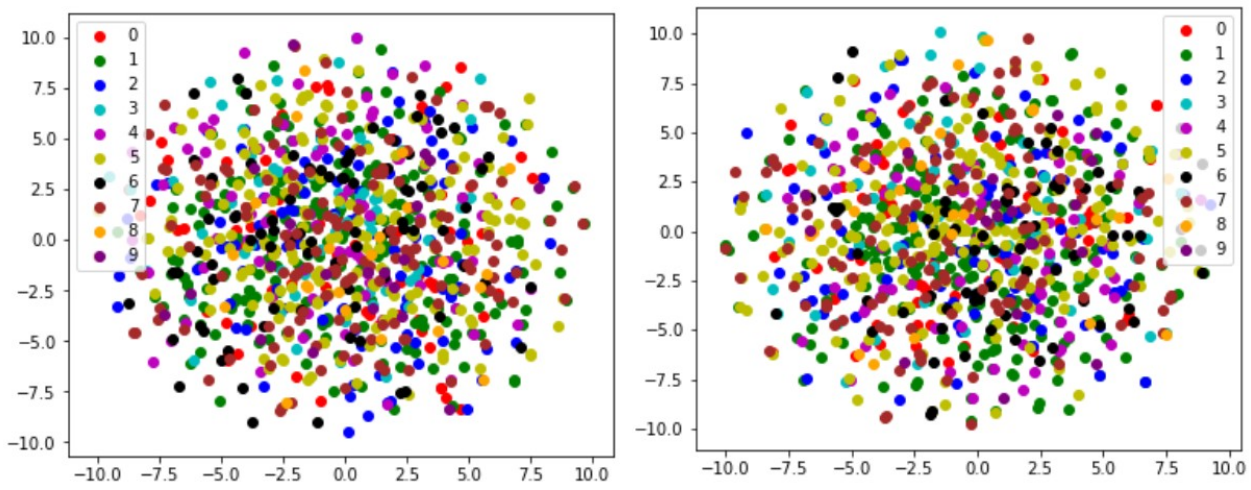



Figure 41: Latent space of z (2 re-iterations). Clusters are not visible and data is spread all around

The clusters are not as visible as for the latent space of the G(z) and the points seems to be randomly spread all over the space. On one hand, these results are unexpected, but also a lot meaningful, as we will now explain.
We need to consider and think about the essence of tSNE: this algorithm is capable of finding correlations between input samples and it is able of reducing the dimensionality of a feature space. In the first case, by giving as input the generated images of the generative network, tSNE was capable of identifying similarities between different samples and thus was capable of generating different clusters. All of this thanks to the fact that the input data are actually vectors containing pictures, i.e. arranged pixels in the space.
Trying to apply tSNE to normally distributed data means that the algorithm is not much capable of determining similarities between different vectors z and thus, all points are spread

over the space. Therefore, the resulting latent space is nothing more than randomly distributed points on the space, due to the fact that tSNE was not capable of collecting them (rightly) in clusters. This analysis was made with the only purpose of showing that the informations and the learned features are not contained in the random vectors z, but instead the GAN will apply on them some kind of transformations in all the hidden layers and as output will give some random digit. We knew a-priori that applying tSNE to random vectors would be meaningless, and in fact results showed exactly that. Since also the case with 10000 samples gave similar results, no further examples of this aspect will be given.

### 3.2.5 Results: 10000 z samples

Increasing the number of samples (and thus of random vectors z) to 10000 has given slightly better results in terms of variety of digits generated. If we look at Figure 42 below the clusters appear to be much more defined and all digits that were previously spread over (like 5 or 3) are now more compact.

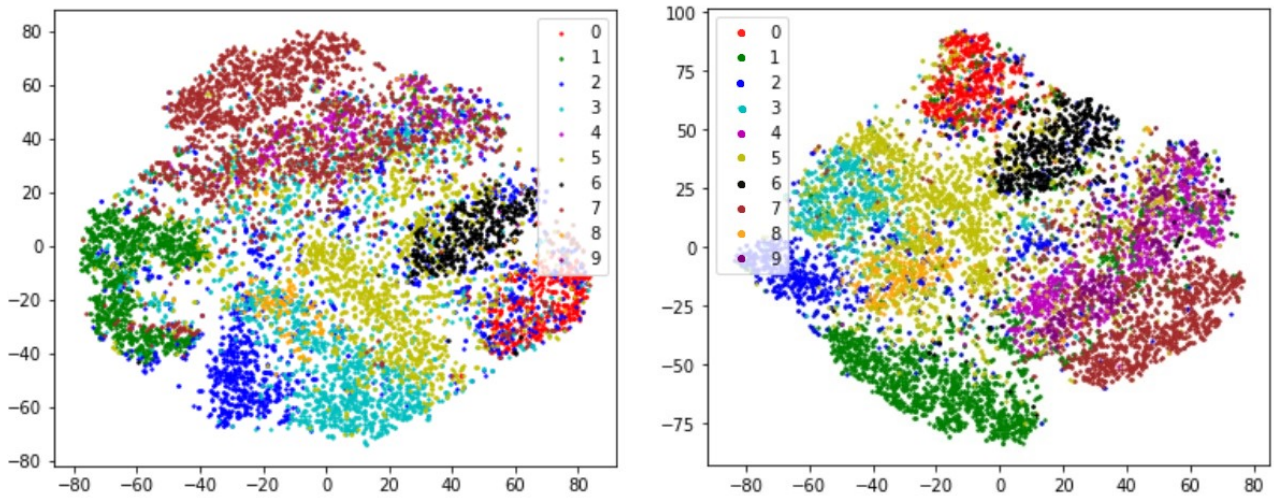



Figure 42: Two latent spaces generated with 2 different epochs.

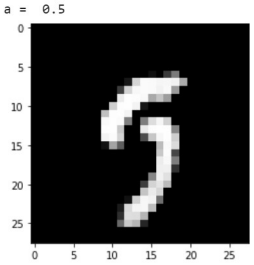
In this second case all digits seems to have their own well-defined cluster a part of:

- Digits 4 and digits 9 seems to be completely confused with no defined cluster. That means that the associated random vectors generates very similar digits.
- Number 8 is still struggling to have its own cluster, with a very small amount of 8 digits generated by the net G.
- All other digits have relatively well defined clusters.

## 3.3 Final considerations

If we look at the clusters we have obtained in Figure 42, we can conclude that the random z vectors that generates a specific cluster are somehow similar each other. What if we try to explain the linear interpolation property seen in section 1 by looking at the clusters configuration

of Figure 9? What we expect to see is a clear transition from 1 cluster to another one, with maybe some intermediate digits.
By looking at example 1 of Figure 36 (step a = 0.5), the digit generated when linearly interpolating number 5 and 9 is actually not classifiable as a proper digit.




As results, if we look to the latent space plot of Figure 42 we can notice that in both plots there is no clear transition between digits 5 and 9 with any intermediate cluster. If we try to plot the digits (from left plot of Figure 42) corresponding to the intermediate positions (x range: [0,20], y range: [-20,40]), we obtain the following generated digits:

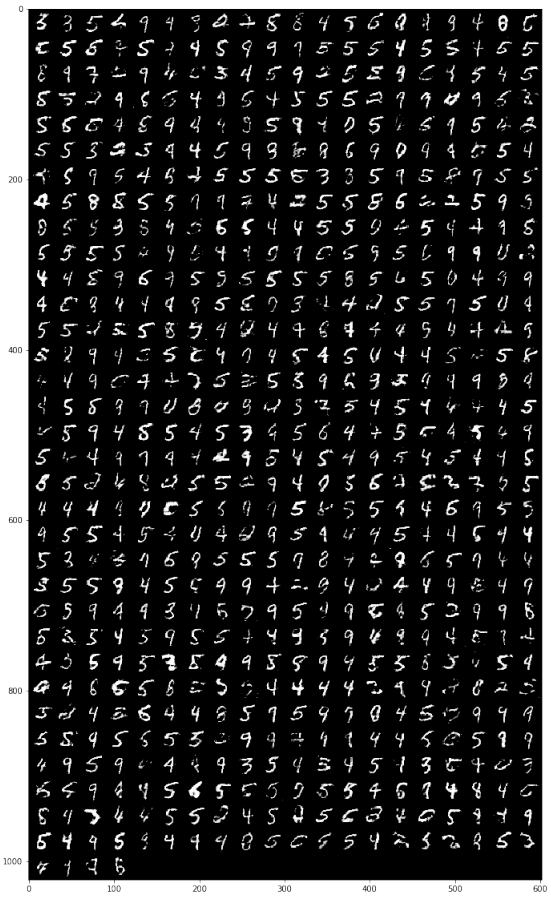



Figure 43: Transition from cluster 5 to 9. Digits in the middle are from different clusters and a lot spread around.

It is quite clear that there is no clear transition between the two clusters and as a consequence, the digits in the middle are highly random. Therefore, in terms of the random vector z,

in this region of the latent space digits are not well defined and tSNE will locate them sparsely in the space instead of a more compact way.
If we look instead at the transition from cluster 3 to 2 (x range: [-10, -30], y range: [-50,-70]) we have:

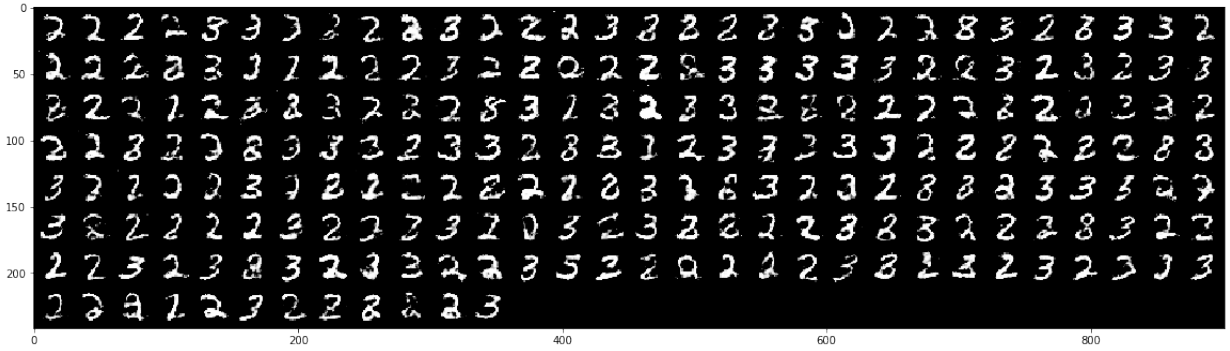



Figure 44: Transition from cluster 3 to cluster 2 (right to left).

In this second case the transition is much more clear and there are not too many outliers from other clusters in this region of the latent space.
It is worth noticing also the evolution of digits in the same cluster. Let's consider for example the digits 1 and let's see how it is changing in the latent space.

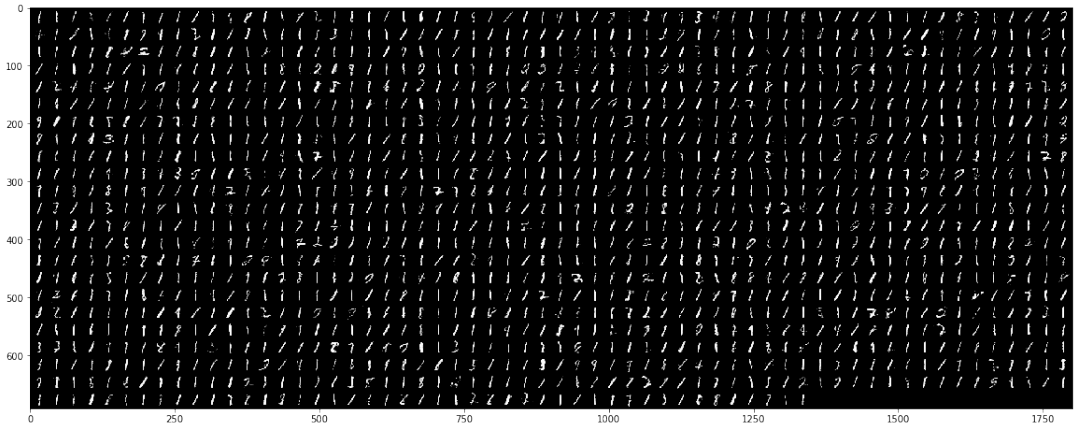



Figure 45: Visualization of a portion of the cluster of digit 1

As a final general consideration, by thinking at the nature of the GAN (generative network that tries to fool the discriminator network), the easiest way to fool the discriminator would be to generate the "easiest" digits to synthesize (like zeros, ones, sevens, sixs, etc.) By looking at the plots in Figure 9 we can actually notice that digits 1,7,0,6 are definitely the most numerous and the ones with the best defined clusters. It is difficult to say exactly ***why*** these digits are the best generated ones, but from a human perspective these are the ones that are most different from each other and do not generate too many ambiguities. Instead, if we think to numbers like 9 and 4, it would be much more difficult to generate them in a sharp and correct way: as results, most of digits 9 and 4 are confused and do not have their own cluster.

## 3.4 Final remarks

There are several factors that may change the performance and the outcome of the whole analysis, like for example the structure of the GAN (which is the one provided in the class notes and has not been modified) or the validation net. GAN networks offers a wide field of applications and many interesting aspects and features that were not taken into account in this brief discussion.
In the next final chapter we are going to discuss about possible problems related to the *"safety"* aspect of a neural network, giving attention, for example, to *FGSM attacks*.

# 4 Chapter 4
# Robustness of Neural Networks

Aim of this last chapter is to study the effect of an adversarial attack on a Neural network. Neural networks are a very powerful tool capable of reaching outstanding performances in many fields (classification, recognition, generation of new data and many others). However, they are still far of being applied in some fields in which, in addition to high accuracy, also robustness and "safety" are required. The biggest problem is that NN can be easily fooled, as we are now going to explain.
Let's take as first (and very effective) example the one below:

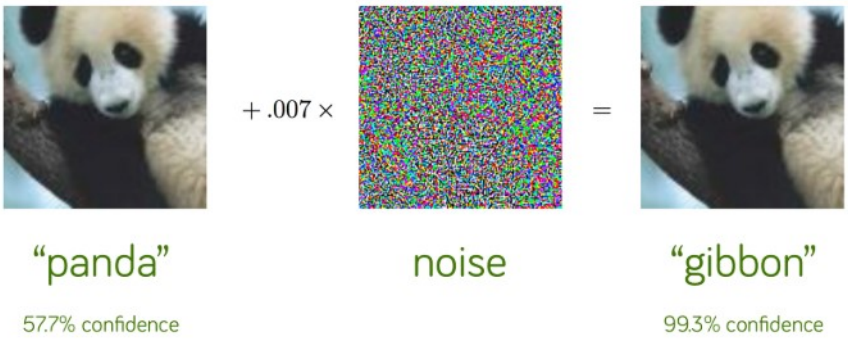



Figure 46: Example of fooling a NN.
Source:*Explaining and Harnessing Adversarial Examples, Goodfellow et al, ICLR 2015* (https://arxiv.org/abs/1412.6572)

after adding a noise to the image which cannot be even recognize by human, the NN goes from classifying a panda with 57% of accuracy to recognizing it as a Gibbon with 99.3% of accuracy. This can be clearly seen as a "mathematical illusion" for the net, which will systematically classify all the pictures wrong. If we consider now the fact that fooling a net is so much easy, by taking into account the *safety* aspect there is one big problem emerging. In the field of autonomous driving, for example, this weak aspect of NN can results in dramatical accident due to the presence of some disturbances (accidental or volunteer) on, for example, stop signs or other road signs. The misinterpretation of road signs or other object on the road (like cars, pedestrians) can lead to bad consequences.
Aim of this brief discussion is to show how an adversarial attack works and to analyse the effects on 6 different nets.

## 4.1 Nets used for the analysis

In order to study the effects of an adversarial attack, 3 different nets have been used. Net 1 and net 2 are the same with the only difference of the optimizer (respectively ADAM and SGD). Net 3 is is different from both 1 and 2, having only 1 convolution layer. Notice that for each net, there is a duplicate (e.g. N11), meaning that it's the same net but initialized with different random parameters. Thus, in total there are 6 nets for the analysis.

- Net N1: 2 convolution layer, 3 fcl, ReLu, crossentropy loss, ADAM

- Net N2: same as net 1, SGD
- Net N3: 1 convolution layer, 3 fcl, Relu, crossentropy loss, ADAM.

Below the structure of the 3 nets.

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(8, 8), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (layer2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(6, 6), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (drop_out): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=800, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
)
```

Figure 47: Structure of N1 and N2

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
  )
  (drop_out): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=12544, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
)
```

Figure 48: Structure of N3

For each net, the performances has been evaluated on a different set of $\varepsilon$ defined as follow:

$$\varepsilon = \begin{bmatrix} 0.2 & 0.4 & 0.5 & 0.7 & 0.8 & 1 \end{bmatrix}$$

## 4.2 FGSM attack explained

Fast Gradient Sign Method is a well established method for generating effective adversarial samples. The idea is to find the smallest perturbation which will make the prediction wrong. Given any NN, by taking its loss function, evaluating the sign of the gradient of the loss and compute a perturbation $\eta$ as follow:

$$\eta = \varepsilon \, \mathsf{sign}(\nabla_x L(\theta, x, y))$$

we are capable of generating an adversarial sample that is perturbed in the direction of the gradient of the loss. In other words, the generated adversarial image is the "worst case scenario" for the net. The parameter $\varepsilon$ can be used to decide the magnitude of the perturbation in the direction of the gradient of the loss.
Since the loss function $L(\theta, x, y)$ depends on the net parameters, the input and the output label, in this analysis we have duplicated each net (e.g. N1 and N11) in order to attack one net with FGSM (e.g. N1) and to check the effects of the generated adversarial samples on the same net (N11), but initialized with different random parameters.

## 4.3 Code and results

On the *.colab* file there are 3 main sections:

- Section 1: Initialization and training of the 3( 6 in total) net N1, N11, N2, N22, N3, N33. Average validation accuracy results from assignment 1 are:
  - N1: 99.19%
  - N2: 94.97%
  - N3: 98.70%
- Section 2: Initialize the functions for FGSM attack, attack function and validation function (for testing the other nets on the generated adversarial samples).
- Section 3: there are 6 subsection, each for one net. In each subsection (e.g. 3.N1) there is:
  - Attack to the current net (i.e. N1). Get accuracy.
  - Test generated adversarial samples for N1 on the other nets (N11, N2, N22, etc.).
  - Iterate for different $\varepsilon$
  - Show accuracy/epsilon plot and some adversarial samples for the current net.

Below the obtained results:

| | | Networks for evaluation of accuracy | | | | | |
|---|---|---|---|---|---|---|---|
| | $\varepsilon = 0.2$ | N1 | N11 | N2 | N22 | N3 | N33 |
| | N1 | 96.16% | 99.01% | 94.84% | 94.94% | 98.31% | 98.70% |
| Target | N11 | 99.08% | 97.70% | 94.52% | 94.73% | 98.22% | 98.60% |
| networks of | N2 | 99.49% | 99.56% | 89.59% | 96.89% | 98.96% | 99.34% |
| adversarial | N22 | 99.48% | 99.49% | 96.75% | 89.17% | 98.89% | 99.14% |
| attacks | N3 | 99.14% | 99.04% | 95.01% | 95.12% | 95.85% | 98.59% |
| | N33 | 99.25% | 99.04% | 94.99% | 95.16% | 98.38% | 94.77% |

| | | Networks for evaluation of accuracy | | | | | |
|---|---|---|---|---|---|---|---|
| | $\varepsilon = 0.4$ | N1 | N11 | N2 | N22 | N3 | N33 |
| | N1 | 94.86% | 98.32% | 93.70% | 93.84% | 97.67% | 98.12% |
| Target | N11 | 98.27% | 96.42% | 93.12% | 93.33% | 97.38% | 97.89% |
| networks of | N2 | 99.28% | 99.29% | 85.75% | 92.11% | 98.61% | 98.98% |
| adversarial | N22 | 99.19% | 99.20% | 92.37% | 84.84% | 98.35% | 98.78% |
| attacks | N3 | 98.63% | 98.63% | 94.12% | 94.23% | 92.77% | 97.73% |
| | N33 | 98.74% | 98.55% | 94.15% | 94.42% | 97.37% | 91.24% |

| | | Networks for evaluation of accuracy | | | | | |
|---|---|---|---|---|---|---|---|
| | $\varepsilon = 0.5$ | N1 | N11 | N2 | N22 | N3 | N33 |
| | N1 | 87.27% | 94.20% | 90.75% | 90.58% | 96.14% | 96.22% |
| Target | N11 | 92.30% | 86.05% | 89.45% | 89.39% | 95.52% | 95.52% |
| networks of | N2 | 97.36% | 97.35% | 60.81% | 78.78% | 97.87% | 98.19% |
| adversarial | N22 | 97.32% | 97.32% | 80.13% | 56.49% | 97.94% | 98.08% |
| attacks | N3 | 97.30% | 97.56% | 92.68% | 92.82% | 81.8% | 95.14% |
| | N33 | 97.27% | 97.39% | 92.74% | 92.84% | 94.93% | 78.82% |

| | | Networks for evaluation of accuracy | | | | | |
|---|---|---|---|---|---|---|---|
| | $\varepsilon = 0.7$ | N1 | N11 | N2 | N22 | N3 | N33 |
| | N1 | 39.24% | 51.73% | 61.62% | 59.31% | 82.38% | 78.16% |
| Target | N11 | 44.11% | 22.76% | 52.40% | 51.24% | 78.06% | 75.40 |
| networks of | N2 | 65.79% | 65.15% | 12.65% | 17.71% | 90.11% | 85.50% |
| adversarial | N22 | 65.52% | 69.07% | 19.00% | 10.8% | 91.30% | 88.69% |
| attacks | N3 | 85.45% | 88.19% | 82.38% | 82.46% | 28.7% | 72.21% |
| | N33 | 82.62% | 85.35% | 81.91% | 81.69% | 72.52% | 33.15% |

| | | Networks for evaluation of accuracy | | | | | |
|---|---|---|---|---|---|---|---|
| | $\varepsilon = 0.8$ | N1 | N11 | N2 | N22 | N3 | N33 |
| | N1 | 22.70% | 32.40% | 43.39% | 41.65% | 61.61% | 56.75% |
| Target | N11 | 22.36% | 12.31% | 33.43% | 33.08% | 52.68% | 52.02% |
| networks of | N2 | 41.99% | 36.37% | 5.00% | 6.19% | 76.12% | 71.49% |
| adversarial | N22 | 39.79% | 41.36% | 6.78% | 3.25% | 80.61% | 73.40% |
| attacks | N3 | 70.83% | 76.10% | 69.13% | 68.17% | 14.75% | 50.59% |
| | N33 | 65.99% | 69.91% | 68.25% | 67.05% | 48.38% | 18.55% |

| | | Networks for evaluation of accuracy | | | | | |
|---|---|---|---|---|---|---|---|
| | $\varepsilon = 1$ | N1 | N11 | N2 | N22 | N3 | N33 |
| | N1 | 11.5% | 14.60% | 22.75% | 21.27% | 26.27% | 25.28% |
| Target | N11 | 7.85% | 5.65% | 13.75% | 13.53% | 22.34% | 21.25% |
| networks of | N2 | 13.15% | 8.05% | 0.34% | 0.56% | 36.49% | 27.78% |
| adversarial | N22 | 10.70% | 10.73% | 0.57% | 0.21% | 41.78% | 31.47% |
| attacks | N3 | 41.97% | 46.48% | 46.37% | 45.92% | 6.58% | 24.42% |
| | N33 | 38.70% | 41.34% | 41.44% | 40.01% | 23.94% | 7.31% |

Table 2: Top to bottom, accuracies for different epsilons. On the columns the are the attacked NN, on the rows the other nets tested on the adversarial samples of the current net.

Below we can observe some different adversarial samples generated for two different nets (N1 and N3)

Figure 49: Adversarial samples for net N1 and N3

We can clearly see that first the two noise patterns are different (because FGSM is computing the noise depending on the gradient of the loss function of the net) and second that up to $\varepsilon = 0.5$ the changes are not too much relevant (need to zoom in to see the pattern). After 0.5 the noise become visible even at human eye (and as expected the performances of the nets dropped after this value).

## 4.4 Considerations on the obtained results

1. There is a clear behaviour before and after $\varepsilon = 0.5$. After this value, from the tables we can observe that all the nets have a decrease in performances from 10% to 30%. Before this value instead, the accuracy is for sure lower than the baseline one (Section 4.3 - point 1 for the values)

2. In each table there is a common trend: on the main diagonal there are always the lowest values of each single row. This is obvious because on the diagonal there are the accuracy values of the attacked net. Therefore, accuracy will be lower with respect to any other value of the same row due to the fact that the adversarial samples where generated in

order to fool the net in consideration. Only for $\varepsilon = 0.2, 0.4$ this is not true: by looking for example at the first row of first table, we can clearly see that even if the attacked net is N1, net N2 and N22 performed worse with the adversarial samples of N1. This is probably due to the fact that the net 2 is less robust to noised images and thus, it is more affected by adversarial samples. Another cause might be that net N2 and N22, along the same direction of the gradient of the loss of N1, are more much more effected. By re-executing the script another time (and thus by re-initializing new parameters for the nets) we are probably going to get different results.

3. If we consider the performances of N1, N11, N2, N22 and remembering that they differ only on the optimizer (respectively ADAM vs SGD), we can clearly see that in all the tables Net 1 has always outperformed net 2 in terms of robustness. If we look for example at tables of $\varepsilon = 0.7, 0.8, 1$, when testing adversarial samples of N1, N11 on N2, N22 and viceversa, N1 and N11 showed higher robustness with respect of N2, N22. It is hard and not trivial at all to explain why a net trained with ADAM should be more robust with respect to another one trained with SGD. Some other studies have been conducted on the understanding of the effects that the optimizer have. For example, an interesting analysis can be found in *Assessing Optimizer Impact on DNN Model Sensitivity to Adversarial Examples* (https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8878095). In section 4.A, when discussing about the performances of different optimizers with the MNIST dataset, SGD resulted to be the worst of them all and the slower in convergence. Thus, in our analysis we obtain compatible results with the one provided above.

4. N3 and N33 were definitely the most robust. If we look to the last 2 columns of all the tables of accuracies, N3 and N33 are the ones that manages to keep the most constant and consistent value of accuracy. Up to $\varepsilon = 0.8$ N3 and N33 managed to keep the accuracy higher with respect to the other net. The drop of performances occurs at $\varepsilon = 1$. For N1 and N11 the drop of performance occur at $\varepsilon = 0.8$, while or N2 and N22 we are experiencing significant drops of performances even at $\varepsilon = 0.7$. Therefore, in our analysis N3 and N33 (the simplest NN with only 1 convolution layer) are the ones more robust with respect to adversarial attacks and testing on adversarial samples (generated on other nets).
   To understand better the robustness, let's visualize the trend of accuracy for each net when attacked, depending on the different values of $\varepsilon$:

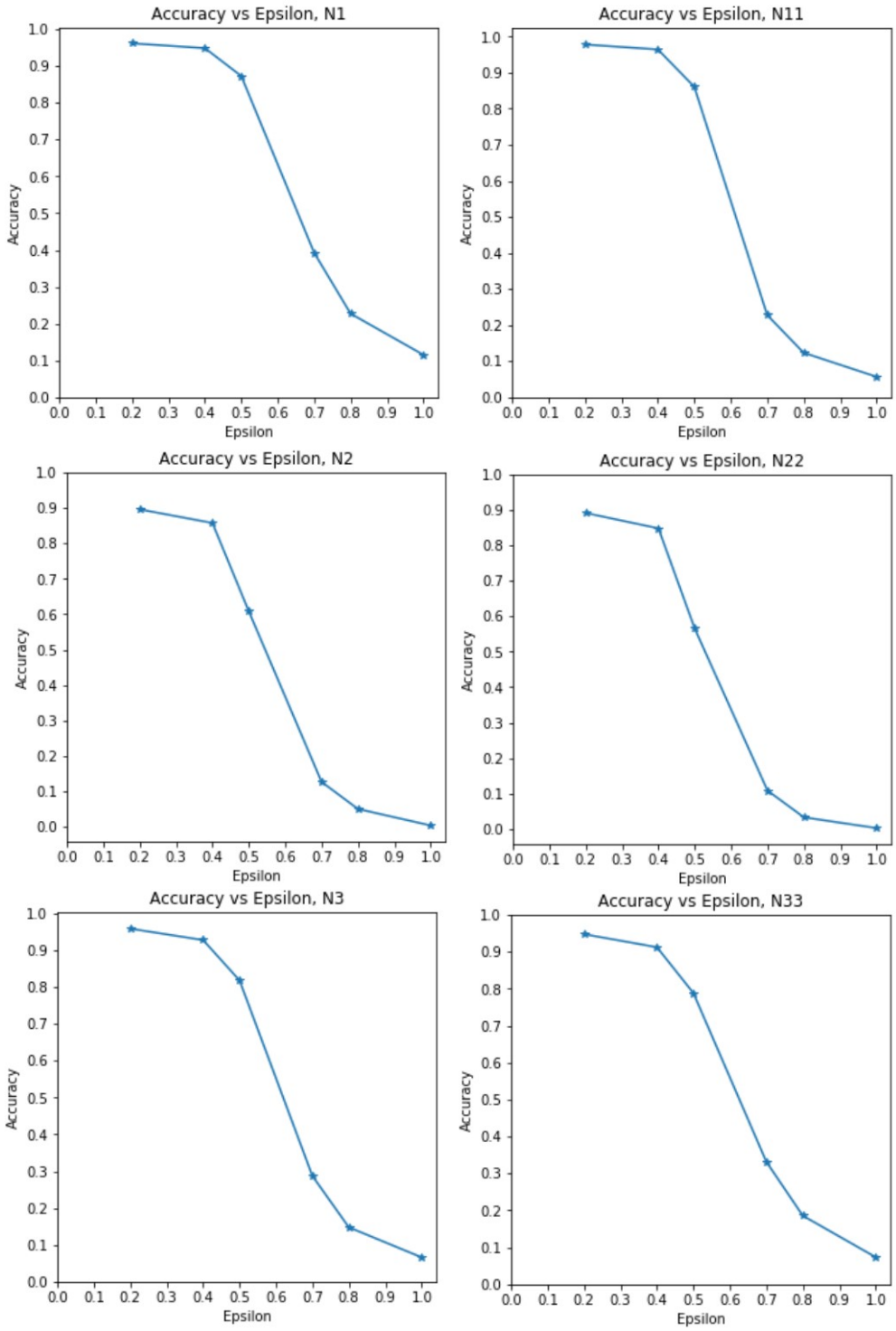



Figure 50: Accuracy/epsilon plot for attacked nets

As expected, N2 and N22 are the ones less robust to an adversarial attack. N1 and N11 are more consistent and the accuracy drops with a lower ratio. N3 and N33 obtained similar performances to N1 and N11, with a slight advantage overall of N1 and N33.

5. Another important observation is about the size of $\varepsilon$: in general, all nets were capable of keeping up with the attack up to $\varepsilon = 0.5$. After this value, all of them starts struggling more and more. Understanding why this happen is not easy, but from a "human" point of view the reason can be found in the fact that in the overall sample, after $\varepsilon = 0.5$ there is more noise information than actual digit information. Thus, when the net tries to extract the feature from the input samples, there are less valuable informations to be extracted and as a consequence, the net will struggle to learn.

6. If we want to visualize what is happening inside the net as the epsilon increase, we can have a look at the latent space of classified images (of N1) for $\varepsilon = 0.2, 0.6, 1$. What we are expecting is a clear and well subdivided latent spaces with visible clusters for $\varepsilon = 0.2$, more confused clusters for $\varepsilon = 0.6$ and literally no clusters for $\varepsilon = 1$. By following the same steps used for assignment 3, we were capable of plotting and visualizing the latent space of classified adversarial images for N1.

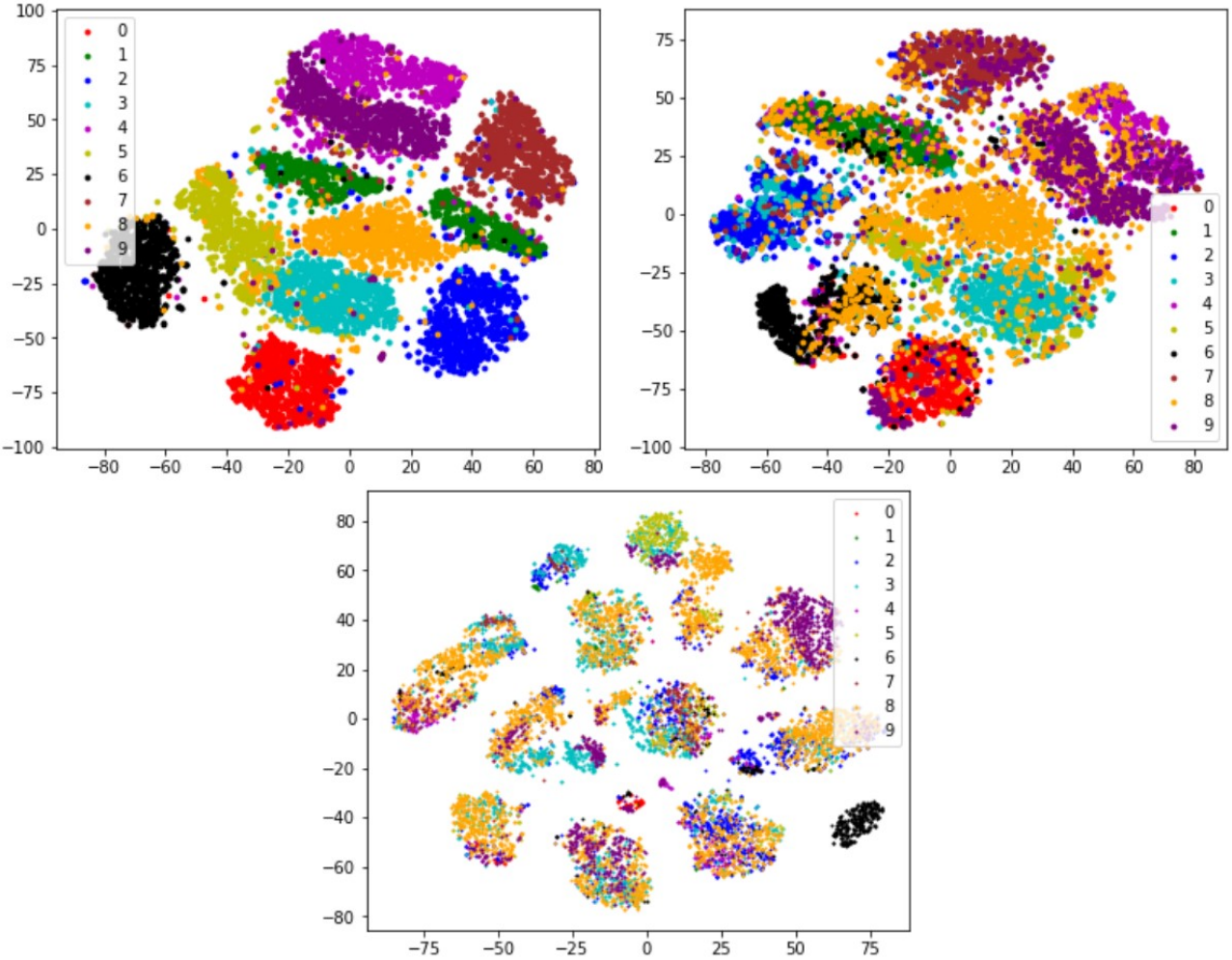



Figure 51: Left to right, top to bottom: latent space of classified adversarial images for $\varepsilon = 0.2, 0.6, 1$

As expected, the adversarial images for $\varepsilon = 0.2$ did not confuse too much N1 and the clusters are still well visible. For $\varepsilon = 0.6$ we can notice that the clusters are more

confused, with in particular digit number 8 being spread all over the space. But still, N1 still achieves decent performance with an accuracy of 63.72%. With $\varepsilon = 1$ the net is totally fooled and that can be clearly seen by the missing and completely random clusters. Only digit 6 seems to be still perfectly recognized.
This analysis was performed only on N1, but can be easily extended also to other nets. This was not performed in this brief discussion. Please also notice that the *.colab* script was ran multiple times, thus values in the saved workspace might be different from the ones showed above.

## 4.5 Final remarks

It is now clear about the danger of an adversarial attack on a NN. There is not yet a *state of the art method* for training NN to not be fooled by adversarial samples. One effective way (already exploited in chapter 3) is the usage of Generative Adversarial Networks (GAN). In this way, the generative net will try to fool with adversarial samples the discriminator net (my actual net), and D will learn to recognize better and better adversarial samples. However, even if this technique might appear to be very effective, it will result in a game of *"who is the best"* where attackers and defenders are just trying to one-up each other. This improves the generalization of the model but hasn't been able to provide a meaningful level of robustness. The biggest question (and probably the biggest unsolved problem) is definitely why is so hard to "protect" a NN from adversarial attack? If we think to the essence of a NN, all the hidden layers will try to learn from a high dimensional space all the useful features necessary, for example, to classify a digit. Changing the values of the pixels appositely and systematically (e.g. adding a noise with FGSM method) will effect in cascade all the I/O relationship of each neuron, and eventually, will produce bad results.
In conclusion, by quoting Ian Goodfellow, one of the pioneers of this field it can be said that, "*many of the most important problems still remain open, both in terms of theory and in terms of applications. We do not yet know whether defending against adversarial examples is a theoretically hopeless endeavour or if an optimal strategy would give the defender an upper ground. On the applied side, no one has yet designed a truly powerful defence algorithm that can resist a wide variety of adversarial example attack algorithms.*"