DATA AUGMENTATION

INTRODUCTION

In this Assignment the goal is to implement data augmentation for a net trained over 1000 samples.

Data augmentation is a technique used in situation where training dataset is limited. As explained during lesson, obviously the higher is the number of samples for the training, the better will be the training of our network. A good trained network will have good performance in the classification of new samples, which is our finally objective.

Unfortunately, it is often difficult to have at disposal many samples. What it is usually done in these situations is to digitally enlarge the training dataset. How? Applying some random transformations to the images of the dataset. In this way, new images different from the original ones will be produced and the training dataset is virtually containing way more samples! In fact, during training, a single training batch will contain both samples from the original dataset and samples that are generated thanks to these transformations. Of course, this augmentation becomes evident when the number of epochs increases: basing on the original training dataset, the network will become better and better until the very moment when overfitting occurs. With only 1000 samples, this can occur pretty soon, because even if in any epoch we are repeating the training with a random combination of the 1000 samples, these will be always the same and at the end overfitting will occur. With augmentation, instead, this overfitting moment will occur with a bigger number of epoch because samples will virtually be more than 1000 and so the final net should be better trained.

Depending on the application, it is fundamental to properly choose an appropriate transformation. As we will see from the results, in this particular MNIST case, some of the transformations can improve the performance of the network, while others can badly deprecate it.

Note also that since this is a didactic assignment, some of the applied transformations actually make no sense if the purpose was to optimize network performance, but they have been implemented to confirm how a bad transformation choice can affect the overall behavior.

Another technique for data augmentation is called Mixup, which basically “trains a neural network on convex combinations of pairs of examples and their labels. By doing so, (as explained in “mixup: BEYOND EMPIRICAL RISK MINIMIZATION” by Zhang, Cisse, Dauphin, Lopez-Paz), mixup regularizes the neural network to favor simple linear behavior in-between training examples.”

Mixup have been successfully implemented in this assignment and results will be analyzed.

Finally, the proposed transformations have been applied to two Neural Networks: a fully connected layers and a convolutional one in order to see if there are some relevant behaviors to be noticed.

TABLE OF RESULTS

NEURAL NETWORK – 3 fully connected layers (512, 512) with Batch Normalization and SGD optimizer

```
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)
  (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=10, bias=True)
)
```

	TRAINING LOSS	ACCURACY
ORIGINAL	0.003	89.10%
RandomAffine – Rotation	0.019	93.01%
RandomAffine - Translation	0.220	91.26%
RandomAffine – Scale	0.015	88.52%
RandomAffine – Shear	0.017	91.37%
RandomAffine - Combined	0.780	64.98%
ColorJitter	0.008	89.50%
RandomCrop	0.090	77.02%
RandomPerspective	0.300	91.37%
RandomResizedCrop	0.156	92.75%
RandomHorizontalFlip	0.007	86.77%
RandomChoice	0.352	93.83%
MIXUP	0.303	88.44%

CONVOLUTIONAL NEURAL NETWORK – 2 filters, 2 fully connected layers, SGD optimizer

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (drop_out): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=3136, out_features=1000, bias=True)
  (fc2): Linear(in_features=1000, out_features=10, bias=True)
)
```

	TRAINING LOSS	ACCURACY
ORIGINAL	0.035	95.22%
RandomAffine – Rotation	0.115	95.94%
RandomAffine - Translation	0.202	95.46%
RandomAffine – Scale	0.089	94.48%
RandomAffine – Shear	0.073	95.53%
RandomAffine - Combined	0.603	78.89%
ColorJitter	0.060	94.44%
RandomCrop	0.151	92.10%
RandomPerspective	0.301	95.34%
RandomResizedCrop	0.192	96.41%
RandomHorizontalFlip	0.127	90.82%
RandomChoice	0.423	95.69%
MIXUP	0.69	95.86%

ANALYSIS OF RESULTS

First of all, some general considerations.

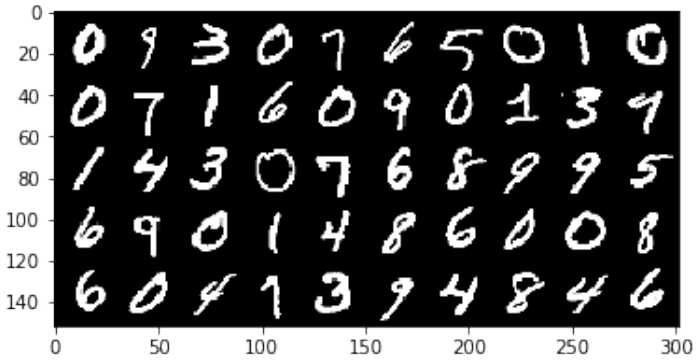
Of course, transformations have to be applied only to the training dataset! Testing dataset has to remain the same as the original one.

For the reason mentioned above, in general data augmentation is useful when we have a small dataset, but to notice its effect on performances it is usually used a sufficient high number of epochs. In this assignment, I decided to use 100 epochs, even if I run the code also with 20 epoch to see confirmation to what stated above: after a certain point, it is useless (and can also be dangerous) to increase furthermore the number of epoch if the samples are few. For example, this is the case of the original (without data augmentation) neural network, that with 20 epochs and 100 epochs returns in both cases a very close value of accuracy of 89%. However, with 20 epochs the loss is almost 10 times greater than the loss with 100 epochs (and in general loss is very small). This probably means that the network is overfitting the 1000 samples data. This overfitting, on the other hand, is more difficult to have with data augmentation

ORIGINAL

Both the fully connected layers and the convolutional neural networks have already been analyzed in the previous example. However, we can shortly say that their performances with SGD optimizer are good, especially considering the low number of samples in the training dataset, but since MNIST classification problem is relatively simple, some improvements can be achieved.

Sample of the original training dataset:

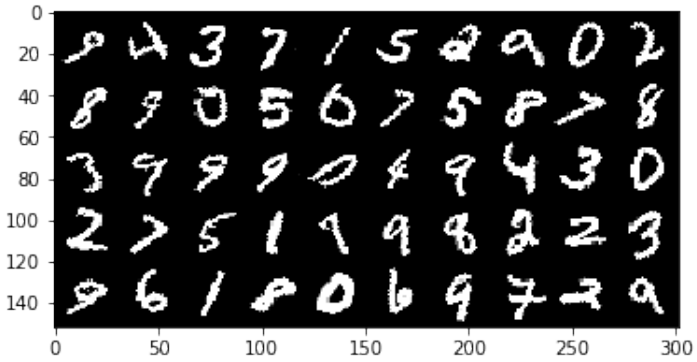


1) RANDOM AFFINE – ROTATION

class `torchvision.transforms.RandomAffine`(*degrees*, *translate=None*, *scale=None*, *shear=None*, *resample=False*, *fillcolor=0*)

With RandomAffine transformation, the samples can be transformed according to a combination of rotation, translation, scaling and shear. For the purpose of the assignment, firstly the single transformations effects are analyzed, and then also the effect of an overall RandomAffine transform.

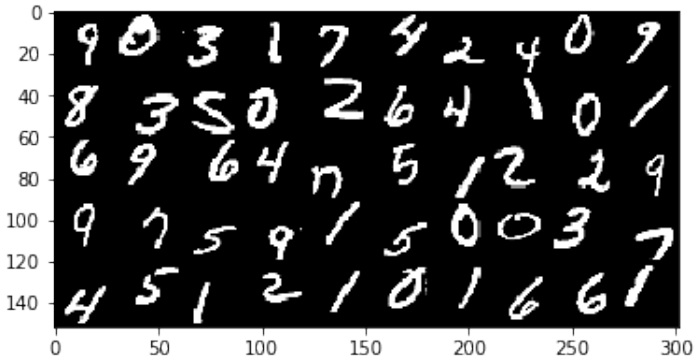
In this case, this is a simple rotation of [-30,30] degrees. Sample of training set is the following:



Results from both the networks are positive: especially for the fc layers net, there is a relevant improvement of the accuracy. It is reasonable: networks become more robust with respect to classification for rotating digits. Note that loss is bigger with respect to the loss of the net without data augmentation, however accuracy is better in this case. This confirms what stated above that with data augmentation, in general, overfitting occurs further.

2) RANDOM AFFINE – TRANSLATION

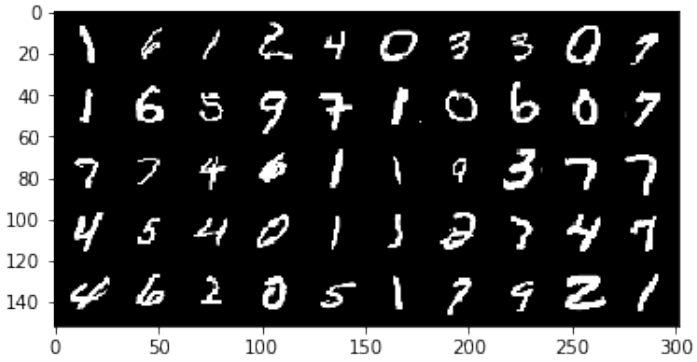
In this case, a random translation with values belonging to [0.2,0.2] interval is applied, obtaining the following dataset:



Also in this case, the significant improvement is for the fc layer, whose accuracy increases of almost 2%. CNN also slightly increases. This is also reasonable: nets are more robust with respect to slightly translated digits

3) RANDOM AFFINE – SCALE

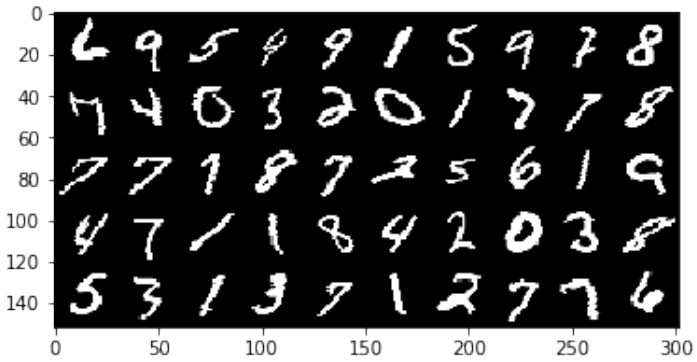
Scale reduces the dimension of the digit (in this case [0.6,1] is the range)and the resulting dataset is:



In this case, in both networks loss is quite small but accuracy very slightly decreases. Probably the network is making some slight overfitting of the training dataset and in some occasions, when in the training dataset a digit is in normal size, it is no more able to correctly classify it (but again, this is a very small error)

4) RANDOM AFFINE – SHEAR

This transformation applies a shear of an angle between [-30,30] to the dataset. Results:

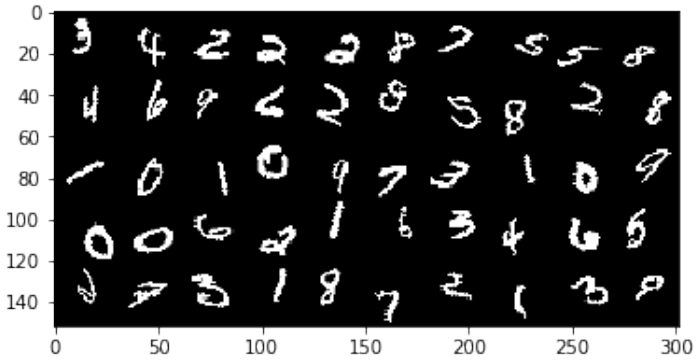


Again, accuracy slightly improves (improves more for the fcl network), making the networks more robust with respect to light shear of digits.

5) RANDOM AFFINE – COMBINED

```
transforms.RandomAffine(30, translate=(0.2,0.2), scale=(0.8,0.8), shear=30)
```

In this transformation, a combination of all the previous ones is performed



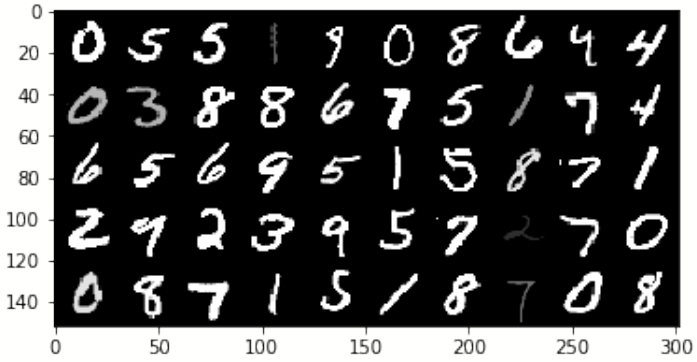
As you can see, resulting digits can be very deformed with respect to the original ones. This explains why for both nets this is the transformation for which the loss has the highest value, and accuracy is badly deprecated (a decrement of 30 to 20 %). Substantially, even during training, networks are confused in classifying this digits. During training, probably parameters are updated many times because if at the iteration before a digit was classified with a label and at the current iteration a very different digit has the same label, this generates confusion. Also, during testing, it is evident that correct classification fails.

This is an example showing that it is necessary to be careful with the choice of transformations for data augmentation.

6) COLOR JITTER

```
transforms.ColorJitter(brightness=0.8, contrast=0.4, saturation=0.2, hue=0.2
)
```

with this transformation, you can modify brightness, contrast, saturation and hue from a range specified by the given values.

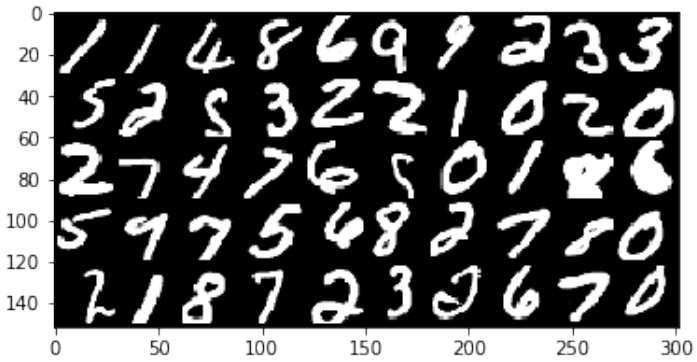


In this case, loss is very small and values for accuracy are totally comparable to the ones obtained from original training dataset. I believe this kind of transformation is useful when the net must be trained to classify digits in different light conditions, but in this case, even if we train on this enlarged dataset, the testing is always done on the samples with standard light conditions and then accuracy is unchanged.

7) RANDOM CROP

```
transforms.RandomCrop(22, padding=None, pad_if_needed=False, fill=0, padding
_mode='constant'),
transforms.Resize(28, interpolation=2),
```

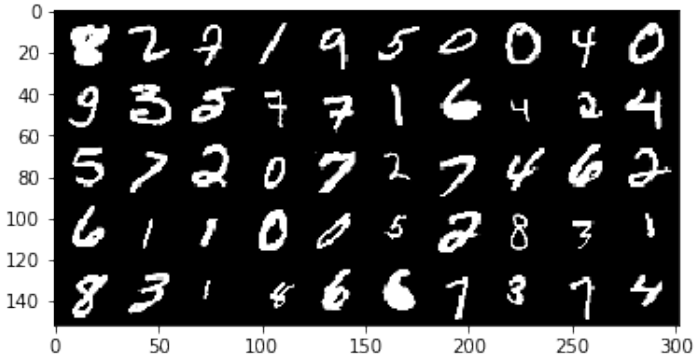
Randomly crop the image reducing its size to a square(22x22), so that some small particular features of an image digit are cut off. To use this function, also transform.Resize is used to have the image with original dimension 28x28 even after the crop.



In both networks, accuracy decreases, in particular for the fcl net. Since CNN performs filtering of the image, it is less sensible to this type of transformation with respect to the fclayers one. Probably, when a particular is cut off (for example, the inferior part of a “2” digit), it becomes more difficult to classify it (continuing with the example, it can be mistaken with a “7”).

8) RANDOM PERSPECTIVE

`transforms.RandomPerspective(distortion_scale=0.6, p=0.5, interpolation=3).` Image is distorted according to a distorsion scale factor (0.6), and result is the following:



You can notice how some digits become smaller, others thicker, others sheared etc..

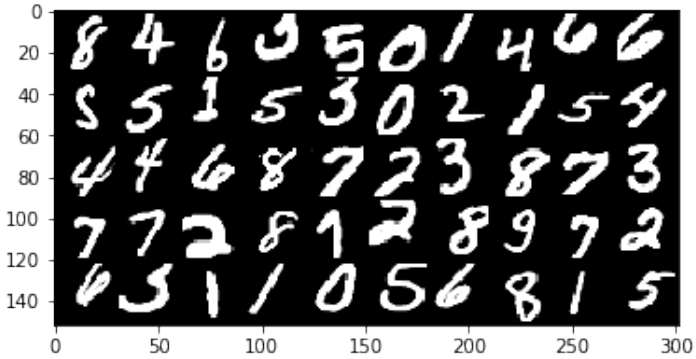
Here, loss is quite relevant with respect to the others obtained from other transformations, but accuracy improves in both nets (2% increase for fc layers)! This is an interesting situation and will be even more evident with Random Choice transformation, but basically what is happening is that the networks are trained over a very heterogeneous dataset and have some difficulties in setting the parameters. However, since they do their best to obtain a good model even with a difficult dataset, at the very moment when the testing is performed over a simpler dataset, it is easier for them to correctly classify the digits!

To make a comparison, it is like if a student is training over a difficult set of exercises and when it comes the moment of the exam, the exercise in it are actually simpler than the exercises that he has done!

9) RANDOM RESIZED CROP

```
transforms.RandomResizedCrop(28, scale=(0.5, 1.0), ratio=(0.75, 1.3333), int
erpolation=2),
```

This case is very similar to the one of Random Crop, however here the cutting is performed with a random aspect ratio (like 3/4) according to passed values.



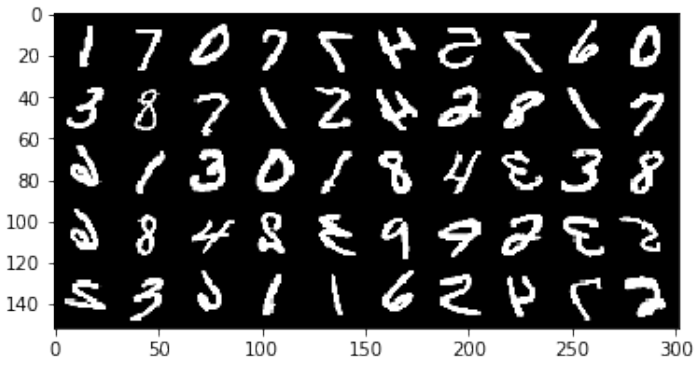
However, in this case there is an interesting improvement of both accuracies of the two networks! This is probably related to the fact that now the cutting is performed with an aspects ratio and a scaling range that tends to magnify the most particular feature of a digit, making the classification more easy to perform! This is interesting because it highlights how a very similar type of transformation can lead to different results (improving or not improving the accuracy) depending on the given parameters! So not only depending on the application we must carefully choose which transformation to perform, but also it is necessary to carefully choose its parameters.

10) RANDOM HORIZONTAL FLIP

This is one of that transformations that for our application does not make sense. As a matter of fact, some digits can mistakenly be interpreted: for example a “2” can be easily be mistaken with a “5” if it is horizontally flipped. However, I think it is useful to see if results confirm theory, which in this case suggest that accuracy should

decrease, but probably not so badly deprecated because many digits, even if flipped, are easily distinguishable (like “1”, “8”, etc).

Dataset example:



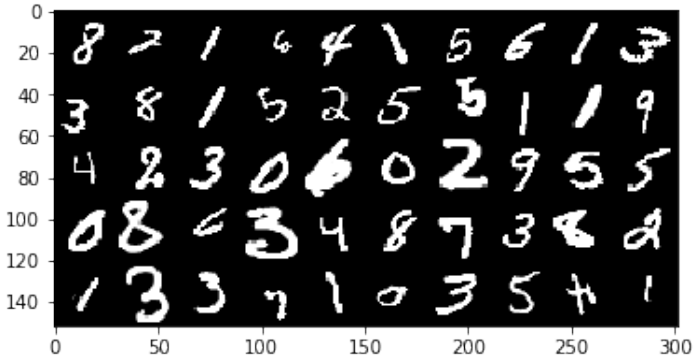
Results perfectly confirm the theory: in both cases, accuracy is decreased by 5% ca., leading to an absolute value of 86% for the connected layers network and 90% for the CNN. This decreasing is probably related to what stated above: few digits can be easily mislabeled.

11) RANDOM CHOICE

```
transform_set = ([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomAffine(30, translate=None, scale=None, shear=0),
    transforms.RandomAffine(0, translate=(0.2,0.2), scale=None, shear=0),
    transforms.RandomAffine(0, translate=None, scale=(0.8,0.8), shear=None),
    transforms.RandomAffine(0, translate=None, scale=None, shear=30),
    transforms.RandomAffine(30, translate=(0.2,0.2), scale=(0.8,0.8), shear=
30),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue
=0.5),
    transforms.RandomResizedCrop(28, scale=(0.5, 1.0), ratio=(0.75, 1.3333),
interpolation=2),
    transforms.RandomPerspective(distortion_scale=0.6, p=0.5, interpolation=
3),
])

trans_train = transforms.Compose([
    transforms.RandomChoice(transform_set),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

The transformation “transforms.RandomChoice” randomly selects a transformation from the set above and applies it to the training images dataset. In this case resulting samples can be very different and we will see that this have positive effect on the training of the net:

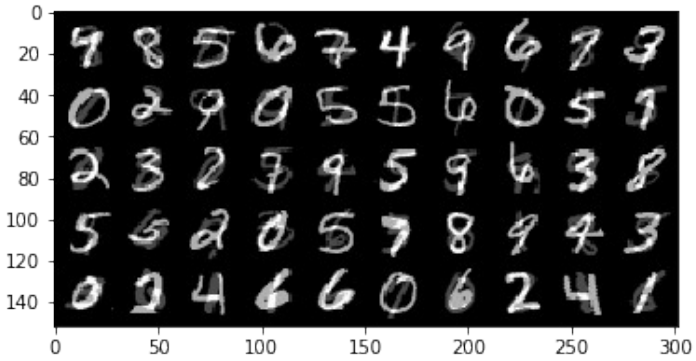


So during training the network is fed up with many different samples. From results, we see that for the CNN accuracy slightly increases, while for the fully connected layers the accuracy rises almost of 4%!! Note also that this is a very similar situation to the one described for Random Perspective: loss value over the training data is not small, but accuracy increases with respect to the network trained over the original dataset without augmentation and the reason is the same as the one stated above.

OPTIONAL – MIXUP

As you can see, a version of data augmentation through Mixup is also implemented. Based on the provided sample code, Mixup have been applied to the same two networks described above. Mixup basically expands the dataset creating a virtual new sample which is identified by an average Input value between the two and an average desired Output. In this way, the net is trained over a set where the edge separation between the labels is, in a certain sense, smaller with respect to the normal training dataset. In this way, when the net will have to classify the inputs, it will be able to return the correct output with more precision because edges between the labels are “wider” and more distinguishable.

Here an example of the obtained dataset augmentation:



You can see that there is a superimposition between 2 digits for a single digit! What the networks do during training is to set the parameters in order to classify these digits with the correct label, which in this case is no more “1””2””3” etc, but a linear and weighted combination between the two used digits for the single sample.

Results in this case are a bit in conflict because accuracy of the simple neural network slightly decreases, while the one of the convolutional network increases. This might be related to the different number of parameters of the two nets: the second one, which is basically more accurate, can exploit better this average generated numbers setting more parameters.

CONCLUSION

In conclusion, with this assignment I was able to understand how helpful the technique of data augmentation can be when the number of available training samples is limited. In many tested cases, the augmentation actually contributed in achieving better accuracy of the net, without changing its structure. I also noticed how important is the choice of the transformation to be applied and also the importance of the parameters of the transformation itself. Results can vary a lot depending on the application and a proper pre-analysis of the task should always be done.

Finally, for this MNIST classification, improvements (when present) may not be so big, but the networks have been tested many times and every time results where coherent with the ones reported in the table. Improvements are limited probably due to the relative simplicity of the MNIST classification problem, but if the transformations would have been applied to a different classification problem, resulting accuracy could improve even more.