Assignment: Regularization techniques and Autoencoding over Multi-MNIST dataset

David Bertoldi – 735213
email: d.bertoldi@campus.unimib.it

Department of Informatics, Systems and Communication

University of Milano-Bicocca

1 Dataset

1.1 Inspecting the data

The data provided consists of a set of gray-scale images containing 2-digits handwritten numbers. The images are always treated as 2D matrices with values in the range $[0, 255]$. The source is a customization of the MNIST dataset: each image is composed by two digits coming from the original MNIST dataset that were overlapped with different intensities. This let the dataset to be expanded from numbers between 0 and 9 to numbers between 1 and 50.

Figure 1 shows the first 16 samples of the training dataset with different labels.

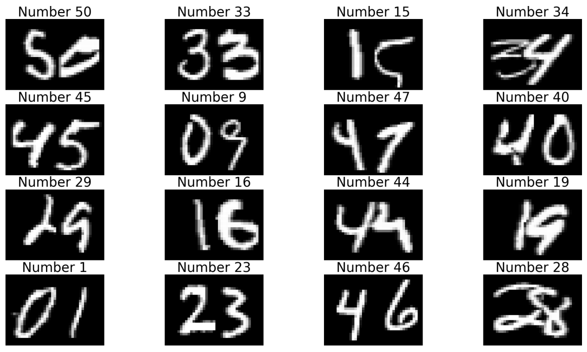

Figure 1: Sample of the first 16 unique samples

The data is already divided into training and test datasets. The first presents a non-uniform distribution of the classes, as shown in Figure 2. Many numbers have very low occurrences, like 43 with 117 samples or 17 with 130 samples, against other like 11 with 3112 samples or 19 with 2 890 samples. As a matter o fact the number of sample for each class in the training dataset has mean $\mu \simeq 1\,671$ and standard deviation $\sigma \simeq 1\,045$.

Along with the biases of the dataset, some number at first sight are difficult to interpret even to the human eye. For example, Figure 3 shows 6 difficult numbers to read: some have poorly defined lines and some have heavy overlaps. This may affect the model's performance if those irregularities are more present in some classes.

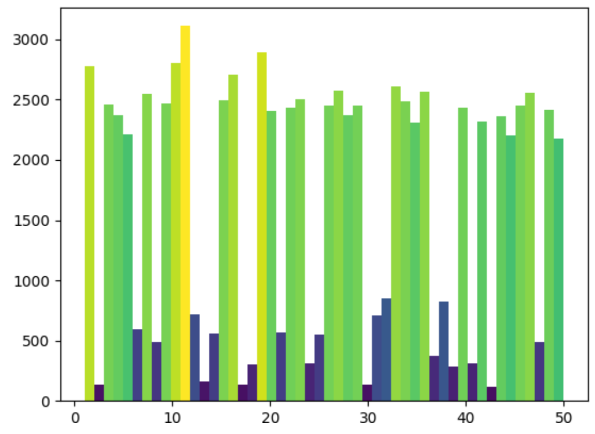

Figure 2: Histogram of the frequency of samples in the dataset

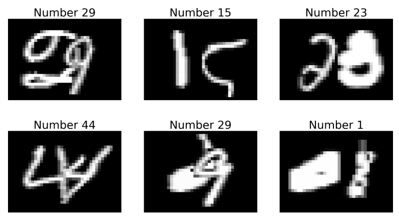

Figure 3: Numbers in the training set that are difficult to read even for a human being

Before training a FFNN using this images, encoded in 28×39 matrices with values from 0 to 255, we flattened them in arrays $1 \times 1\,092$ and rescaled each value in the continuous interval $[0, 1]$. This encoding will be used in every section of this work: a flat array better suits the input layer of a FFNN and small values increases the efficiency in the calculations [1].

1.2 Preparing the data

As noted in section 1.1, the dataset is divided into training and test samples. A validation subset is missing and thus

[1]Even if the final choices made for this work led to the usage of activation functions that were not affected by *exploding gradient*, rescaling the values of the matrices helped the experimentation with other activation functions that were affected.

is retrieved from the training set: 20% of the images are randomly used for validation instead of training (along with their labels).

About labels, we encoded them in one-hot vectors so that the 1s are set in the index representing the numerical class.

Another, technical, issue is that `np_utils.to_categorical`, used to create the one-hot vectors, generated 51 classes instead of 50. That's because the function created as many classes as the highest value inside the input plus one: it took for granted that we were using a zero-based index. In order to overcome this, without writing a similar function from scratch, we subtracted 1 to each element of each vector inside the training and test label sets. Obviously that operation had been reverted when trying to predict the values.

2 Unregularized FFNN

The aim of this section is to describe a FFNN with less than 100 000 parameters that is able to classify with high level of accuracy the numbers from the dataset without any regularization technique.

2.1 The network

Because the number of parameters are partially determined by the size of the input and output, we tested a FFNN with 2, 3, 4 and 5 hidden layers, but we found that 2-layers model generalized better over a deeper (and less wide) model. Without entering in the details, the deepest model had 5 layers with 70 to 50 neurons each.

We found a good spot with 80 neurons in the first layer and 50 in the second one, for a total of 96 660 parameters. Figure 4 shows the architecture of the network used in this section.

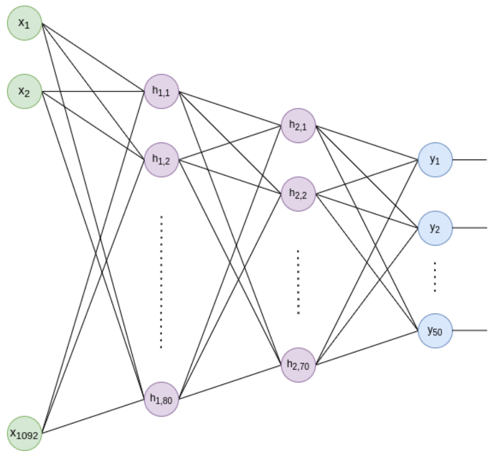

Figure 4: Architecture of the unregularized network

Each hidden neuron used the *Sigmoid* as activation function. We tried *LeakyReLU* with $\alpha = 0.01$ as well but resulting in a divergence in the model (the validation loss kept to slightly grow after 15 epochs). The source was probably the explosion of the gradient caused by the function.

The output layer computes a *Softmax*, which converts the input vector of real values to an output vector that can be interpreted as categorical probabilities.

2.2 Training

The choice of the optimizer was among *SGD*, *RMSProp* and *Adam*. This selection was influenced also by the initialization of the weights: *SGD* performed well with `GlorotNormal` (also known as *Xavier*) initializer but not as good as *RMSProp* and *Adam* with `HeNormal`. *SGD* with `HeNormal` resulted in a model that couldn't converge at all.

We tried all of them and chose *Adam* with learning rate of 10^{-3} , $\beta_1 = 0.9$ and $\beta_2 = 0.999$ because it seemed to escape better from local minima, converging faster and giving better accuracy.

Because we are trying to find which images best suits in one of the 50 classes, the best loss function is the *Categorical Cross-Entropy*.

The batch size of 256 gave the best results: 512 was another good choice but the convergence was slower and lower values of 256 performed worse; that might be due to the fact that the model needed a good amount of variety of information before every update. But even using mini-batches of 8 the validation accuracy reached 89%. When experimenting different batch sizes we always used powers of 2 in order to take advantage of the hardware parallelization.

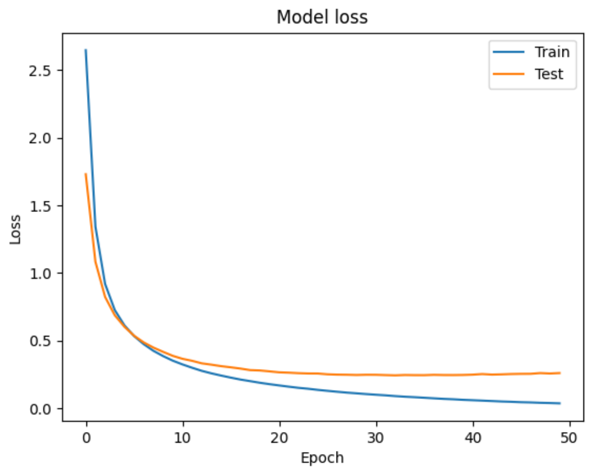

Figure 5: Loss (unregularized)

We choose 50 as number of epochs in order to see the effects of missing regularization like *early stopping*, even if the model reached an optimal state after 10 epochs.

As metric we used the *categorical accuracy* because calculates the percentage of predicted values that match with true values for one-hot labels. As shown in Figure 5 and 6 it is possible to see that the training validation reached $\sim$ 100% and the validation accuracy reached 91.5% after 14 epochs and never got better. This means the model overfitted too much and that would perform worse with samples it had never seen.

2.3 Evaluation

The categorical accuracy over the test set reached 89% and can be analyzed with the help of the confusion matrix (Figure 7).

The confusion matrix uses a custom color-map (*viridis* but with the first 10 values set to 0) so that was easier to hide

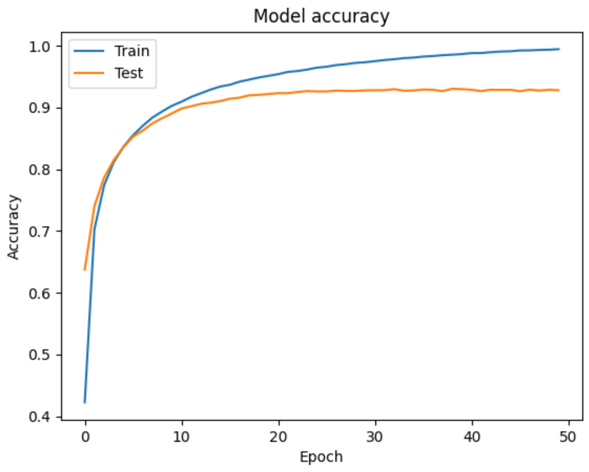

Figure 6: Categorical accuracy (unregularized)

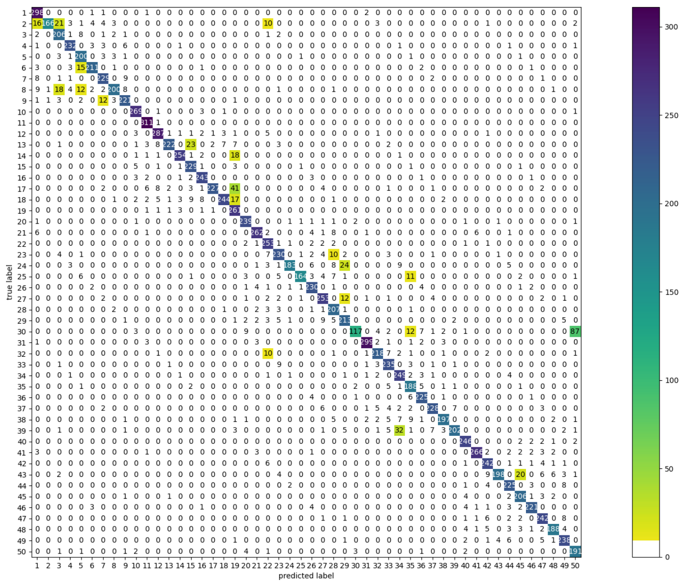

Figure 7: Confusion matrix of the evaluation of the test set (unregularized network)

negligible errors (in this case less than 10 mismatches) in such a large matrix. It is noticeable that the model confuses number 50 with 30, 19 with 17, 34 with 39 and others. These errors are not only due to the similarity between their digits, but because these are the classes with less elements in the training set. So with no surprise the unbalances inside the training set played an important role.

3 Regularized FFNN

The aim of this section is to describe a FFNN with less than 100 000 parameters that is able to classify with high level of accuracy the digits from the dataset with one or more regularization technique.

3.1 The network

We wanted to keep the network as similar as possible to the one described in Section 2 in order to better measure the effects of regularization. For this reason we used the very same architecture as base (2 layers, 80 and 70 neurons). The only thing that changed was the activation functions: we switched *Sigmoid* with *LeakyReLUs* because the first outperformed the second. Probably the effect of *exploding gradient* was restrained by the regularization and a *Sigmoid*-based model suffered too much the underfitting.

3.2 Regularization techniques

In this section we describe the techniques tested, used or discarded to achieve a lower level of underfitting and secondary to reach a better level of accuracy.

3.2.1 *Data augmentation*

The first attempt of indirect regularization involved filling the gap between the classes by generating new samples for the less populated classes. The procedure, described in Algorithm 1, generates for each class as many samples as there are in the percentage p of the difference with the most populated class. By choosing $p = 0.1$ each class diminished the gap with the major class by 10%.

Three techniques were taken in account: *adding noise*, *image shifting* and *image rotation*. The first added a Gaussian noise $\mathcal{N}(0, \frac{1}{2})$, the second randomly shifted the image long the 2 axis by 2 pixels (2 positions in the matrix) in both directions and the third rotated the image by a random angle between $-20°$ and $20°$.

None of the above helped the network: the level of accuracy dropped to 86% and the loss was very high when applying noise ($\geqslant 0.6$). Image rotation alone is the only one that didn't make the accuracy worse. For these reasons data augmentation was discarded even if theoretically speaking filling the gap between samples made sense. Probably the model already recognized the single features of those samples so that adding similar samples just increased redundancy.

Algorithm 1: Data augmentation algorithm

```
Data: $X$ training samples, $Y$ training labels, $p \in [0,1]$
Result: $X$ augmented, $Y$ augmented
1  $l = |Y|$;
2  $C \leftarrow \{\}$;
3  for $i \leftarrow 1$ to $l$ do
4      if $Y_i \in C$ then
5          $C_i \leftarrow C_i + 1$;
6      else
7          $C_i \leftarrow 1$;
8      end
9  end
10 $m \leftarrow \max_k C_k$;
11 for $(k, v) \in C$ do
12     $S = \{e \in X : e = k\}$;
13     $g \leftarrow \lfloor (m - v) \cdot p \rfloor$;
14     for $i \leftarrow 1$ to $g$ do
15         $x \leftarrow augment(S_{random})$;
16         append $x$ to $X$;
17         append $k$ to $Y$;
18     end
19 end
20 return $shuffle(X, y)$;
```

3.2.2 L1 and L2

L1 and *L2* are typical techniques used to reduce the overfitting of the model by settings penalties in the coefficients. We tried first *L1*, but even with low regularization factors (from 10^{-3} to 10^{-5}) the model underfitted too much. *L2* with a regularization factor of 10^{-5} performed better, similar to a a regularization factor of 10^{-4} and a learning rate 10 times higher (10^{-2}), but was outperformed in accuracy by a model not implementing it.

3.2.3 Dropout

Dropout turned out to be a good choice: it decreased the overfitting without losing too much accuracy. We chose a drop out probability for each layer of 10% because the network wasn't too big and higher probabilities gave worse results. This strategy helped the network to ignore certain features of the images or the 0*s* of the matrices (black spaces). A representation of the network can be found in Figure 8.

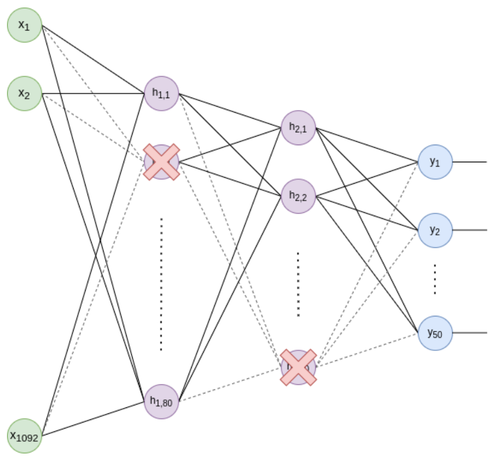

Figure 8: Architecture of the network with dropout

3.2.4 Early stopping

Another game changer was the application of *early stopping* over the validation process, with a patience factor of 10 epochs and a minimum $\delta = 0.05$. This greatly reduced the overfitting and because the model converged after 15-20 epochs we restored the weights to the last best snapshot (20th epoch).

3.3 Training

The optimizer used was *Adam* with a learning rate of 10^{-3} in conjunction with `HeNormal` initializer. We used 50 epochs and batches of 256 elements not only to replicate the methodologies used with the unregularized model, but because outperformed other configurations. This means that the addition of dropout hasn't changed the quantity of information needed by the model before an update.

3.4 Evaluation

The validation accuracy reached 93% (+1.5% over the unregularized model) but with a training accuracy of 94%: this means the model had less memorized the dataset in favor of a better generalization. The test accuracy reached 92% (+3%) and this proved that the model implementing regularization techniques had a higher capabilities to generalize the problem.

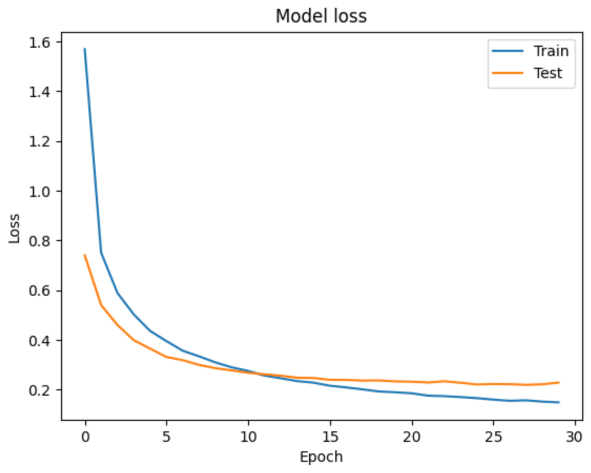

Figure 9: Loss (unregularized)

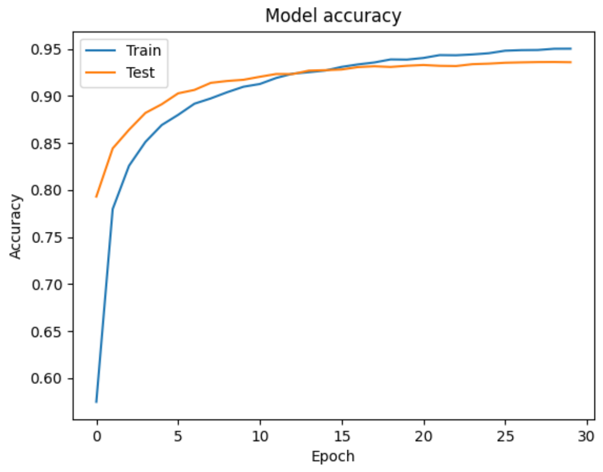

Figure 10: Categorical accuracy (unregularized)

The confusion matrix in Figure 11 shows the improvements made from the previous model: there are still some mismatches between predicted and true labels, but they have lower intensity *i.e.* there are less cases where the 50 is confused with 30 (-28%) or 19 with 17 (-70%).

The regularization definitively changed the network performances without changing the architecture per se.

4 Autoencoding

In this section we show how to build an autoencoder that compresses the knowledge of the original input and that reconstructs the input starting from the compressed (encoded) version.

4.1 The network

Before deciding the compression rate, we describe here the fundamentals of the autoencoder's architecture. The input, like in the previous cases, is represented by a flatten version of the images, that is a $1 \times 1\,092$ array. Because it is an autoencoder, the same goes for the output. We chose 2 hidden layers for the encoder and 3 for decoder; we used this "unbalance"

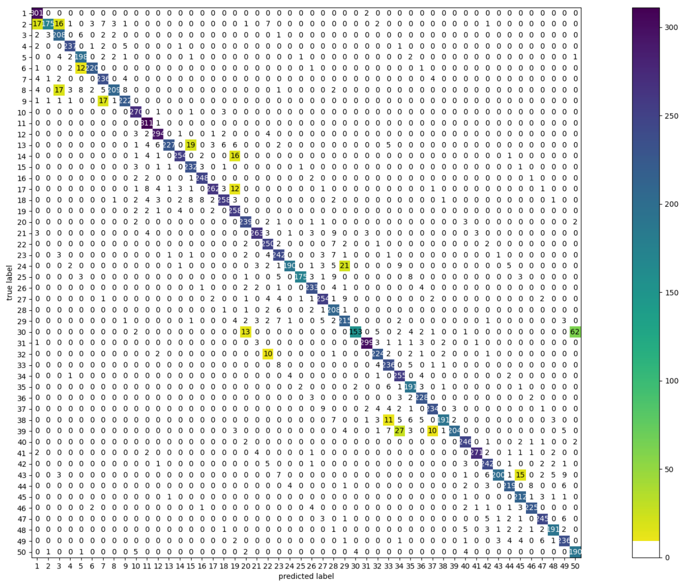

Figure 11: Confusion matrix of the evaluation of the test set (regularized network)

because it easy for the network to compress information but harder to decompress them. So the decoding part uses a more complex model.

The compression factor that defines the length of the encoding had been decided with a benchmark: we trained 10 networks with different compression factors (from 20 to 30) and analyzed their mean squared errors (*MSE*) when trying to reproduce the input. Figure 12 and 13 demonstrates that the best compression factor was 28, that is having an encoded layer formed by 39 neurons. This result depends more on the architecture of the network rather than the problem itself, but it is noticeable and not so expected that the plot in Figure 13 does not represent a monotonic function nor an increasing function.

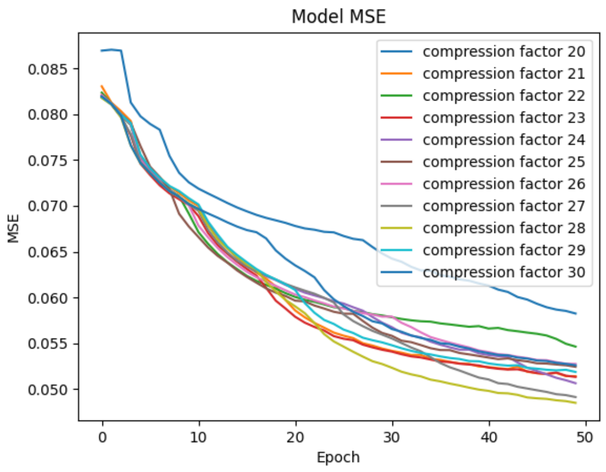

Figure 12: *MSE* vs. number of epochs with multiple compression factors

Figure 14 shows the final architecture of the autoencoder: 1 092 neurons for input, 256 and 128 for encoding, 39 for storing the compressed knowledge, 128, 256 and 512 for decompression and 1 092 for output.

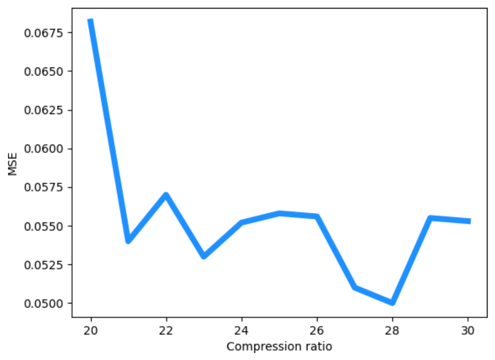

Figure 13: *MSE* with different compression factors

All the layers used `LeakyReLU` with $\alpha = 0.01$ as activation function. The output layer used a *Sigmoid* that better represents values between 0 and 1.

4.2 Training

For the training phase we used *Adam* but with *binary cross-entropy* loss function. *Binary cross-entropy*, based on the logarithmic operation, better avoids the effects present in the *Sigmoid* function, that is based on the exponential function.

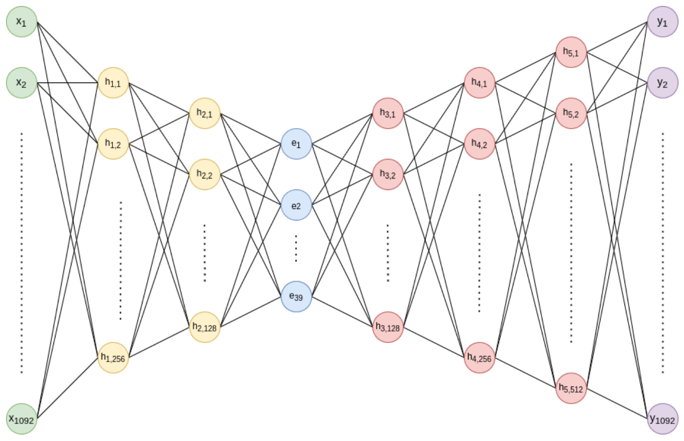

Figure 14: Final architecture of the autoencoder: in green the input, in yellow the encoders, in blue the latent layer, in red the decoders and in purple the output

4.3 Evaluation

The best way to evaluate the autoencoder, other than plotting the loss and the *MSE*, is to visualize the results (Figure 15). In the first row there are 10 samples from the test set while in the second row the output of the autoencoder. The similarity between the two sequences is quite high, but we can notice that the second row presented a certain level of blurriness. This is caused by the data lost during the encoding and the fact that the model didn't converged to a global minimum (lossless compression).

4.4 Generation

In order to generate random new samples, we trained the autoencoder as previously done in section 4.2 and used only

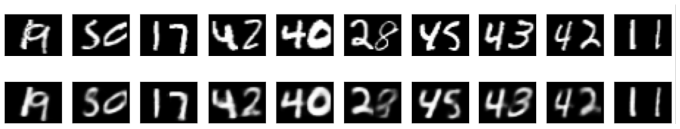

Figure 15: First row: original test data. Second row: the same input but reproduced by the autoencoder.

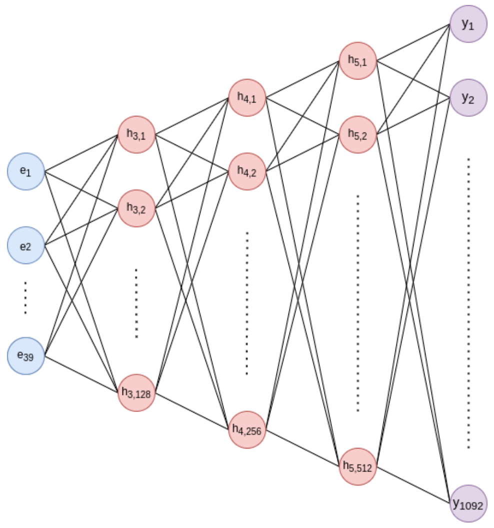

Figure 16: Architecture of the generator: the input and encoding layers are removed from the Autoencoder

the second part of the architecture, that is from the encoded layer to the output (Figure 16)

We fed the model with random numbers with two different distributions: the first is a uniform distribution $U(0,1)$ and the second a custom one we describe later.

The random input with uniform distribution gave the results shown in Figure 17

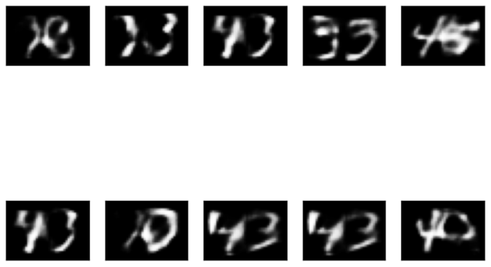

Figure 17: New samples generated from a uniform distribution

Some digits are distinguishable (like 38, 33, 40, 43 and 45) but they are blurry, lowly defined and redundant (many 33 and 43).

Because results were not exhaustive, we analyzed the distribution in the encoded layer after having fed the network with the entire test set. Figure 18 demonstrate that the distribution is far from being uniform.

So we implemented a custom distribution that tried to emulate the real distribution present in the encoded layer after feeding the model with entire test set. Algorithm 2 describes the procedure.

The procedure calculates a new matrix of encodings, generated by a normal distribution $\mathcal{N}(\mu,\sigma)$ on each column, where μ and σ are calculated on the same column of the matrix of encodings. Since the first dimension describes different sequences, what is important is to maintain the distribution of each column. That's because the weights in the encoding layer (like in any other) cannot be permutated without affecting the final result of the model. That's why we emulated the distribution by column, assuming that the values are normally distributed (which is probably not, but this should require more time into investigation).

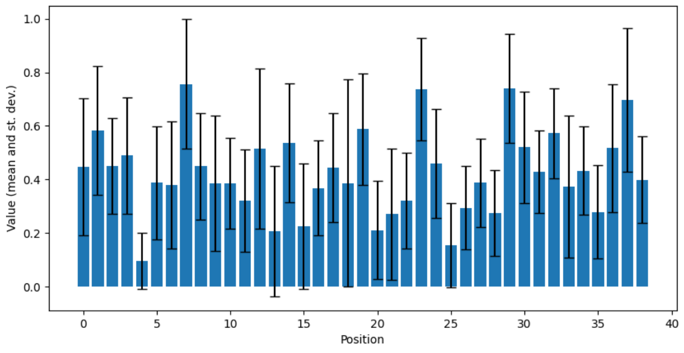

Figure 18: Means and deviations standard of each column of the encodings generated over the training set

Algorithm 2: Custom distribution

```
   Data: $P$ matrix of predictions over $X_{test}$,
         $n \in \mathbb{N}$ number of encodings
   Result: $E$ matrix of random $n$ encodings
1  $E \leftarrow \mathcal{M}(n \times 39)$;
2  for $i \leftarrow 1$ to $n$ do
3      for $j \leftarrow 1$ to 39 do
4          $column \leftarrow P_{*,j}$;
5          $m \leftarrow \mu(column)$;
6          $s \leftarrow \sigma(column)$;
7          $E_{i,j} \leftarrow random(\mathcal{N}(m,s))$
8      end
9  end
10 return $E$;
```

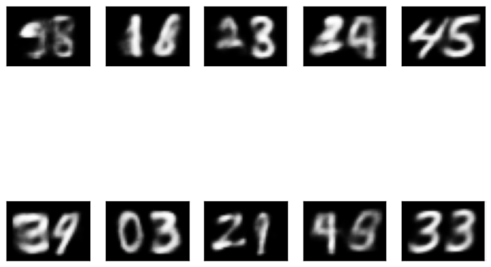

Figure 19

Figure 19 shows the results, which are way better than the ones generated with a uniform distribution. We can find sharper shapes, a little bit of more diversity in the digits (although there are many 3s). What is very interesting is the appearance of a 98, even if the network has been trained on numbers between 1 and 50. This suggests that the order of the weights matters but it doesn't describe the order of the digits; that is quite predictable since the layers are densely-connected and every neuron can affect any other neuron of the next layer.

5 Combination of the latent representation with SVMs

In this section we built an hybrid architecture with FFNN and a non-linear SVM. The input flows into the encoder described in section 4.1 in order to generate a latent representation. This representation is used to feed a SVM that solves the classification task.

5.1 Architecture

The architecture is shown in Figure 20. We used the same encoder with the same activation functions and number of neurons per layers. On the other side, we had to calculate the optimal cost parameter of the SVM. This parameter decided the size of the margin hyperplane (the higher the cost the smaller the margin) and thus it made the model penalize less or more the samples inside the margin.

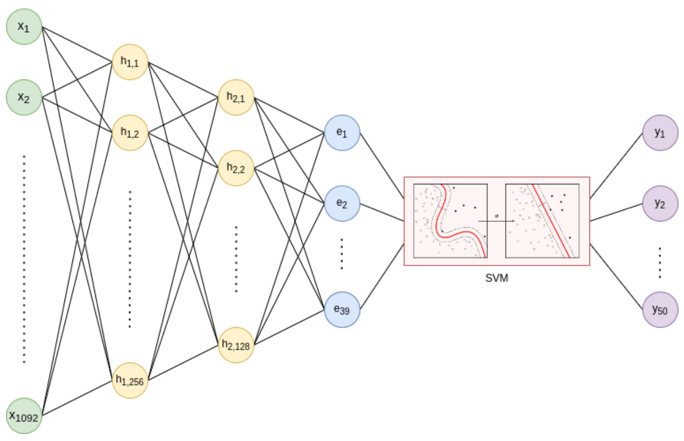

Figure 20: Architecture of the encoder that feeds the SVM

In order to decide the cost parameter, we trained the SVM with multiple costs in a logarithmic scale: from 10^{-4}, to 10^4. We measured the accuracy both with training and test set to evaluate the overfitting of the SVM. To speedup the process we used only 1000 samples per training since SVM training time grows as $O(f(C, d \cdot |X^{train}|^2))$ where f is some increasing function, C is the cost parameter and d is the number of features in the training set X^{train}.

The output of this procedure is shown in Figure 21. We noticed that the model starts to overfit with cost $C \geqslant 10$, so we tested the interval $1 \leqslant C \leqslant 10$ and the model scored a maximum accuracy of 84,8% with $C = 10$ over the entire test set.

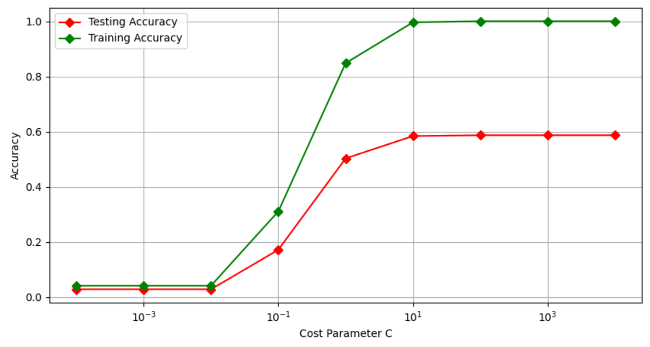

Figure 21: Accuracy vs. cost parameter C (logarithmic scale)

This test produced the confusion matrix in Figure 22. We noticed that the model is less accurate than the unregularized and regularized models and it still maintains their same weaknesses: the less populated classes suffered of higher level of confusion *i.e.* it confused numbers 50 with 30, 19 with 17 and 34 with 39.

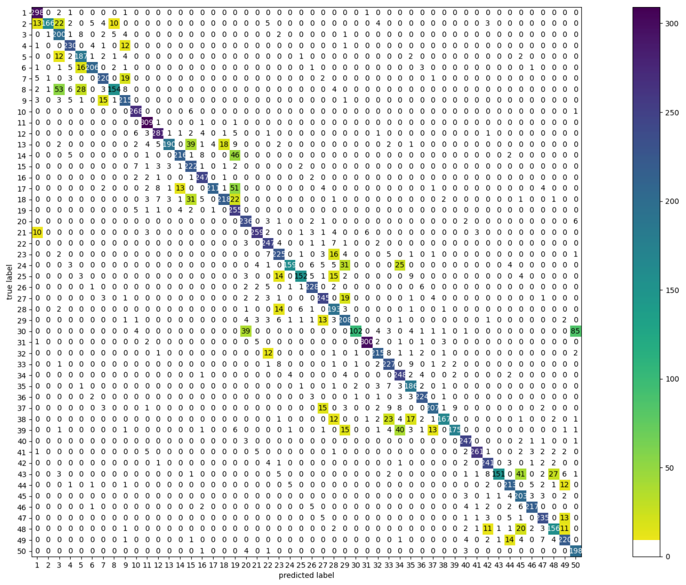

Figure 22: Confusion matrix of the hybrid model with $C = 10$

Even if the results is not as good as the regularized model's ones, the experiment was not a complete failure. We demonstrate that is possible to train hybrid ML models with good results; also it doesn't mean that a better integration between a FFNN and other classical ML models doesn't exist. This case study covered only non-linear SVMs.

6 Implementations

The training of the unregularized model can be started with `noreg-training.py`. It generates the needed images for the project in `./images/noreg/` a model checkpoint (hdf5 file) in in `./noreg`. The validation can be performed with `noreg-eval.py`.

The training of the regularized model can be started with `reg-training.py`. It generates the needed images for the project in `./images/reg/` a model checkpoint (hdf5 file) in in `./reg`. The validation can be performed with `reg-eval.py`.

Training, reconstruction and generation of new samples via autoencoder can be started with `generate.py`

Training and validation of the hybrid model can be started with `svm.py`

7 Conclusions

In this work we trained a FFNN on a custom MNIST dataset without any regularization technique and compared the same FFNN with dropout and early stopping techniques. We found out that the second one performed better during both training and test phase; regularization techniques are a powerful tool that prevents overfitting even without changing the overall architecture. We tried also data augmentation and $L1/L2$ penalization but with worse results.

The constraint of 100 000 parameters played an important role in the decisions made; without it other form of regularization would have shined and outperformed the unregularized model with a larger gap.

We build an autoencoder and used its parts to generate new random samples and attach it to a SVM to classify the samples. In the first case we discovered that the generation of new samples cannot be accomplished by using any random distribution and that the model was able to produce meaningful images whose class was not part of the training dataset; in the second case we demonstrated that hybrid architectures can give good results, although the FFNN with regularization gave the best results.