Katholieke Universiteit Leuven

Master of A.I. - Erasmus Programme

Artificial Neural Networks and Deep Learning

Exercise Report

Student:
Federico Stella
r0766122

Professor:
Johan Suykens

May 2019

Notes

All the code used to produce results and images can be found at:
`https://github.com/ilceltico/ANN-Exercises`

Inside "Code", there is a subfolder for each exercise session with the code I wrote and the files required to run it.

Hoping that the print size is enough to understand the images, the repository contains anyway a directory called "Report_figures", in which one can find all the figures of this report in full resolution, with file names corresponding to the figure numbers in this report. I will also include the original MATLAB '.fig' figures.
Moreover, in the subfolders of "Code" one can find, if interested, all the images that I've generated and not included here.

1 Exercise session 1

To compare the characteristics of different algorithms, let's consider here the classic gradient descent (`traingd`), with adaptive learning rate (`traingda`), with adaptive LR and momentum term (`traingdx`), conjugate gradient (`trainscg` only, as it outperforms both `traincgf` and `traincgp`), quasi-Newton (`trainbfg`) and Levenberg-Marquardt (`trainlm`). Bayesian Regularization (`trainbr`) turned out to be part of the last exercise: it's included in the next figures but will be discussed later. In figure 1 we can see the MSE as it decreases with increasing training epoch, for all the mentioned training algorithms. In the training set we have 189 samples, and the network was left to the exercise default (1 hidden layer, 50 units). The number of parameters of the network (151) is very high for these few samples, but since there is no stochastic noise we don't expect overfitting. Overfitting could occur with deterministic noise too, given by the target complexity with respect to what the network can produce, but this is definitively not the case (for a 50-units NN). We can clearly notice that the standard gradient descent is outperformed by all the other algorithms. In particular, we see that the adaptive learning rate is, in general, a good improvement, but it can introduce oscillations. In the figure they are not too bad, but with more complex examples it can get much worse. To avoid this, a momentum term is a great choice: `traingdx`, in fact, performs better than both, while hugely attenuating the oscillations. The performances of `trainscg` and `trainbfg` are pretty much comparable, at least in this simple example, and offer a great improvement over gradient descent. `trainlm` is finally a huge improvement over all of them, since it converges much quicker and to much lower MSE (the y-axis is logarithmic). For higher NN dimensions and more complex problems, I expect the Levenberg-Marquardt method to become slower, and to be eventually overcome by the conjugate gradient algorithm.

In terms of performance, plotting the same MSE comparison over time, instead of epochs, will be much more interesting. In figure 2 we can clearly see that the Levenberg-Marquardt is still the best one, with gradient descent algorithms performing very similar to each other. Quasi-Newton shows that, while still performing good in simple NN, is the one that takes longer to execute. For comparison purposes, we can assess that the linear regression of the network outputs with the training set provides a perfect r-value of 1 for `trainlm`, while only around 0.65 for `traingd`. In the test set, the values remain almost always identical in both cases[1]. As expected, generalization is very good. Even changing the number of parameters of the NN by a great amount (in both directions), generalization doesn't change. The best performance is obtained starting form 28 hidden units, which is explained by knowing the target function, since that's the number of peaks that it has in the considered x interval. Finally, figure 3 shows the approximation of the target function by a 50-units NN with gradient descent and Levenberg-Marquardt: the approximation of the latter is almost perfect, the only error being represented by the imprefect representation of target peaks by the training set.

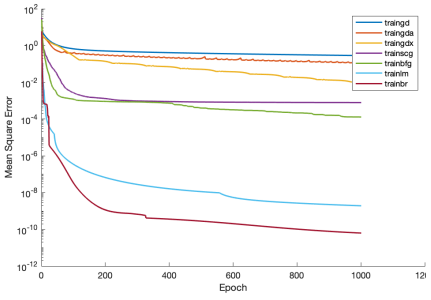

Figure 1: MSE over epochs.

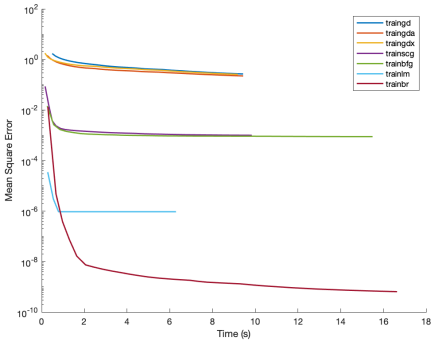

Figure 2: MSE over time.

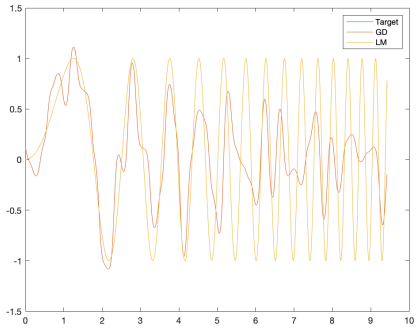

Figure 3: Output curves

Adding noise, things start behaving very differently in terms of error and generalization. Here we have stochastic noise, and the risk is that the NN optimizes the performance with the training set too much, ending up fitting the noise instead of the target function. To clearly show this, figure 4 shows the MSE over both a training and a (much bigger) validation set, with respect to the number of hidden units. The noise added to the target function is Gaussian with multiplication coeffient equal to 1 (quite high). The algorithm used is `trainlm`, since it was the best one in the previous exercise, and with 100 epochs. We can see that there is a good approximation of the out-of-sample error (that is, generalization) up to 20 hidden units. Above that value, things start behaving worse: the in-sample error decreases, while the out-of-sample increases, indicating overfitting. This is due to the fact that the number of parameters of the NN starts approaching the number of training examples. If we use, instead, a slower-converging algorithm such as gradient descent, shown again in figure 4, the situation gets a bit better[2]. The conceptual motivation of this is that

[1] "Test set" is used improperly here and in the noisy exercise. I simply mean that it is used to estimate with great accuracy the out-of-sample error, as opposed to the in-sample error which is given by the training set. Since here (and later) I use this information to estimate overfitting, it is more properly a validation set.

[2] I showed `traingdx` instead of the normal gradient descent because the latter showed divergence for high hidden units numbers. That is due to unoptimized (too high) learning rate, that overshoots the optimum and starts oscillating. The general observations remain valid. I also used 1000 epochs for this algorithm, because 100 were definitively not enough to show convergence.

the gradient descent explores a smaller space than `trainlm` with the same number of epochs, and this prevents the NN to fit too much (both the target and, in this case, the noise). This corresponds intuitively to a lower number of "effective" parameters (and VC-dimension). In fact we can see both a better approximation of the out-of-sample error by the in-sample error and also a better out-of-sample error! The overfitting threshold is therefore also higher, locating at around 40 units.
If we decrease the amount of noise and/or increase the number of examples, we can see much better generalization capabilities: reduced stochastic noise means that a fit on the training is more probable to also fit the validation/test; increased number of examples means both that we can, on average, get a better fit to the target, and also that we can afford a higher number of parameters without incurring in overfitting. A commonly used value for a good number-of-examples over VC-dimension ratio is 10, and in fact with around 1000 training samples we start having overfitting in `trainlm` at around 35 hidden units (that is a bit more than 100 parameters)[3]. Similarly to what I showed above, `traingdx` offers better generalization with higher number of parameters (figure 5), but this time the out-of-sample error is higher in absolute terms.

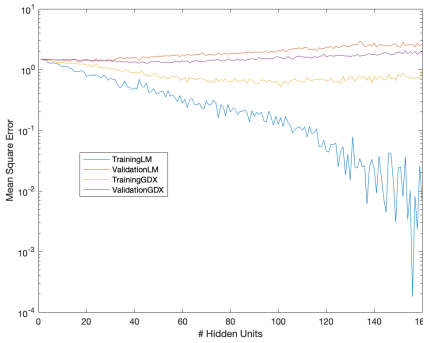

Figure 4: MSE over hidden units, high noise, small training set.

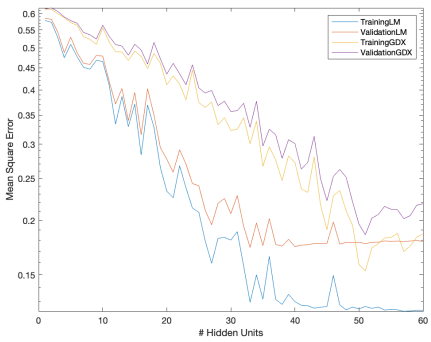

Figure 5: MSE over hidden units, lower noise, bigger training set.

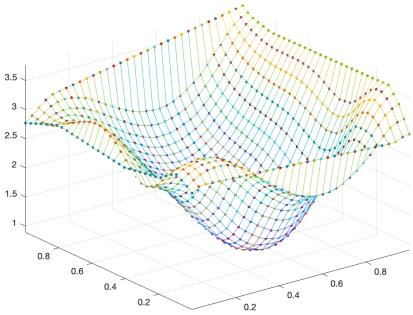

Figure 6: Training set surface.

1.1 Personal Regression

In this exercise, the 3 independent datasets are drawn with the `datasample` function: each of the elements is drawn independently from the others, thus with replacement. This means, of course, that the same element can appear in multiple sets, but reflects a uniform sample distribution. The training set will be used for NN training, which means that the goal of the training algorithms will be to minimize the in-sample error on the training set. The validation set can be used to set hyperparameters, it provides insights into the out-of-sample and generalization errors, and can be used, in this case, to choose a network structure (number of layers and of hidden units). These two sets are the only ones used during training: looking at real out-of-sample results (i.e. on the test set) would mean data snooping, and it is of course not allowed. The test set will only be used after having already chosen the structure and trained the NN. Figure 6 plots the training set surface, obtained by interpolation of the training set points. The points are also shown.
Learning algorithm: since it proved to be the fastest (at least for small regression problems), I will use `trainlm`. Transfer function: according to the most common usage of NN, I will use a linear transfer function in the output layer, while keeping nonlinearities in the hidden layer/s only. For regression problems, symmetry usually appears to have a positive impact on learning, thus I will use the tanh transfer function, which is implemented in MATLAB in a faster version called `tansig`. For the training I will use 1000 epochs, since they are generally not too computationally expensive and provide enough in-sample error minimization with `trainlm`. Because of this fact, we can consider a NN trained with 1000 epochs to make *approximately* full use of its number of parameters. In order to choose the right number of parameters I will then only modify the number of hidden units and layers, and test generalization with the validation set. Since we have 1000 examples in the training set, I would set the number of parameters to around 100, for the same VC-related reason as above, which means about 25 hidden units. I would then compare the generalization errors of NN from 1 to 25 hidden units. Since this comparison shows a clear decrease in both the training set and validation set errors, and since the validation set is quite big (as big as the training set), I will push it a bit further: figure 11 shows these results from 1 to 70 hidden units. We can see that both errors decrease, while reaching a plateau at around 40 units. Generalization seems to be good, even though for high unit numbers the gap between the errors slightly increases. I will then repeat this comparison from 25 to 45 units, but with 10 runs each (selecting the best one in validation error), as shown in figure 12. Above about 34 units, the validation error seems to stabilize.

[3]I very roughly consider here the VC dimension to be equal to the number of parameters of the NN. This is usually not the case, as the relationship can actually be quadratic or more, but for bounded inputs/outputs and with other restrictions it can become closer to linear.

I also tested networks with more than one layer, showing no significant improvements over what has already been shown: a shallow network is theoretically enough as a universal approximator, and for this *simple* example it seems to be enough even in practice. A further test with also the number of epochs as a parameter showed that `trainlm` quickly reaches the plateau with no overfitting, and 1000 epochs can be considered, in this case, a fair number. Both the tests with variable epoch numbers and with variable hidden units showed that overfitting is basically absent, with good generalization up to very high numbers of parameters, higher than expected with the first 25 units esteem. This suggests that the underlying function is not too complex.
The final choice of the network is then a 1-hidden-layer network, with 34 units and tanh transfer function: deeper networks do not offer significant improvements, neither do more units. It is randomly initialized and trained with `trainlm` 10 times for 2000 epochs, choosing the best one in terms of validation error. The resulting training set error is $1.53e-8$, while the validation set error is $2.81e-8$.

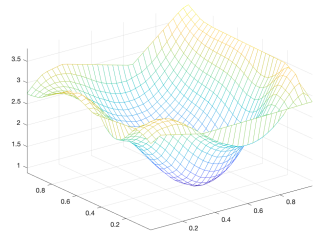

Figure 7: Test set surface.

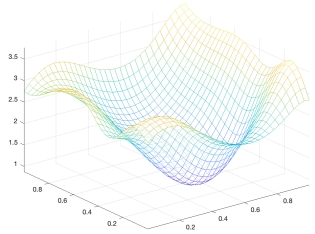

Figure 8: NN output surface.

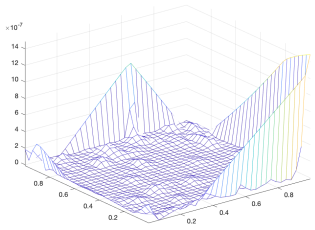

Figure 9: Squared error surface.

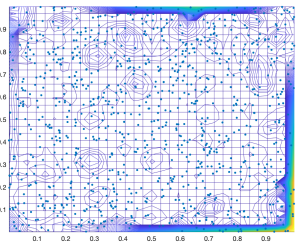

Figure 10: Error levels and training set.

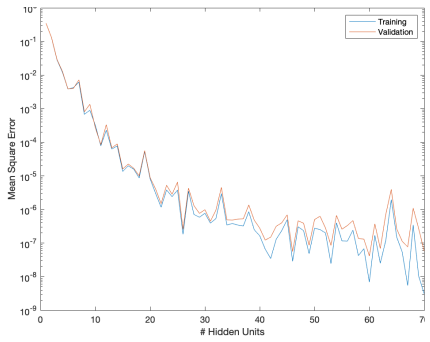

Figure 11: MSE over hidden units.

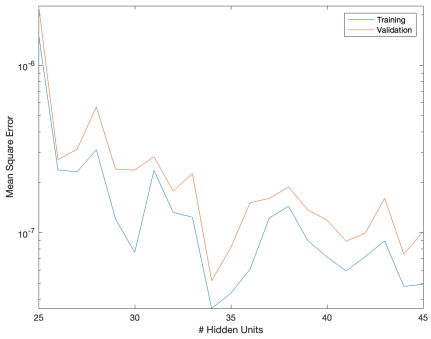

Figure 12: MSE over hidden units, 10 trainings.

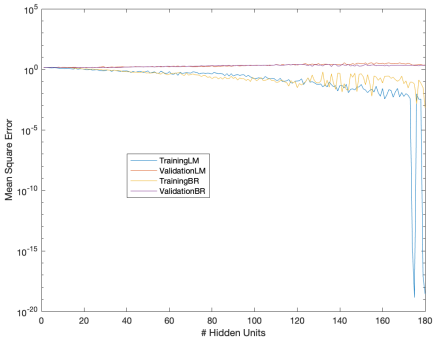

Figure 13: MSE over hidden units, high noise, small training set.

Finally, with all the parameters fixed, a test set evaluation shows a good performance, indicating good generalization of the network. The MSE on the test set is $2.27e-8$. Compared to the training set, we can see in general a good level of generalization, with error peaks accentuating near the borders in figure 9, that shows the error surface on the test set. We can obtain figure 10 by adding level curves on the error surface, plotting the training set points and viewing it from above the error surface. Here we can see, again, that the error is higher in borders, but we can also notice that the error tends to be bigger in areas were the training points density is lower, intuitively.

A good improvement on the performance on this network would happen if we could use more training points, especially near the borders, but of course this requirement is out of the scientist's control. The target function was quite simple, but in more complex cases I would also add regularization to the network, for example with `trainbr`, as it helps avoiding overfitting and can lower the effective number of parameters of the NN. Moreover, it doesn't need a validation set, so I could use both the training and validation sets for training, doubling the number of available examples and thus potentially increasing performance and generalization capabilities.

Concerning Bayesian regularization, the figures 1 and 2 show that `trainbr` performs really good, better than all the others, as it combines the best one of them with regularization. It is definitively slower to run, but even the time plot shows that it's worth it (in the example). The algorithm adds regularization to the `trainlm` algorithm, so it is expected to have great performance if the number of parameters is not too high, being outperformed by `trainscg` at that point. In figure 13, that shows performance over hidden units for a very noisy target, we can see that `trainbr`, while still maintaining a comparable test/validation error, achieves better generalization (the training error doesn't drop as much as `trainlm`). This happens because the regularization hyperparameters tuning helps avoiding overfitting, lowering the number of effective parameters in the NN.

2 Exercise session 2

2.1 Hopfield Networks

After creating a Hopfield Network with attractors $T = [1\ 1; -1\ -1; 1\ -1]^T$ we can that, starting with different initial vectors, one can end in another, spurious, attractor. The attractors are thus actually 4, the last one being $[-1\ 1]$. It's also possible to notice an unstable equilibrium point in the origin, since it is equidistant from all the attractors. In figure 14 I proceed in a more systematic way to generate 250 random points and some more points from symmetry axes, such as diagonals, horizontal and vertical. The last 2 are plotted in blue, since their points land in other unstable points: the midpoints of the edges of the square, defined by the 4 attractors, since they are indeed equidistant from 2 attractors. The only way to land on the origin is then to start directly from the origin: any other point on the vertical or horizontal symmetry axes will land on the edges. On average, it takes slightly above 10 iterations to reach a stable point. This number varies, of course, with distance: the more distant the higher the number, the closer the faster the convergence.

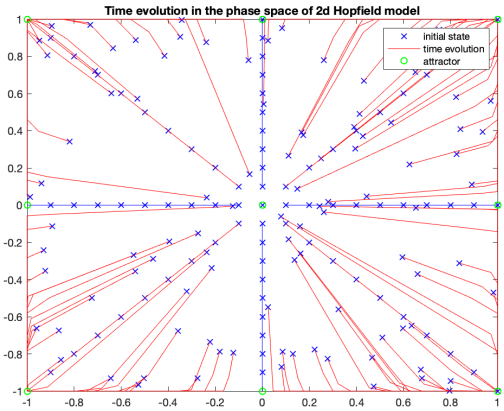

Figure 14: 2D time evolution.

With a 3D Hopfield Network things behave very similarly. By generating random points and letting them converge, we observe now that the network has been able to store correctly the 3 attractors that it was supposed to, without any additional attractor. This is due to the increased dimension of the network, which implies a higher probability of correctly storing a certain amount of attractors with respect to networks with less neurons: it seems in fact unlikely to correctly store 3 attractors in a 2-neuron Hopfield Network. The average number of iterations required for convergence is slightly higher than before, due to increased dimensionality: around 13.
If we proceed to test an equally spaced point grid as initial vectors, we can see the unstable equilibrium points. They are, of course, the circumcenter of the triangle defined by the 3 attractors, and the 3 midpoints of the edges of that triangle. We can see this in figure 15.

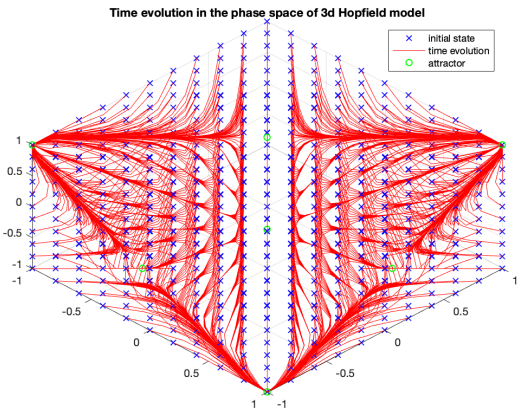

Figure 15: 3D time evolution.

Concerning noisy digit recognition with a Hopfield Network, we can notice that it does a quite nice job in reconstructing the noiseless digit, moving the noisy one to its closest attractor. In general, we can say that the higher the noise the higher the number of iterations required for reconstruction, because the noise pushes the digits farther away from attractors. But, if the noise is too high, the network can converge to a wrong attractor, and this is due to the fact that the noise can be so high that the digit resembles another digit more than its original noiseless version, resulting in a similar behavior to error correcting codes based on distance to the "attractor". It also sporadically happened for some inputs to reach unwanted spurious attractors, that do not represent any digit, which is common in Hopfield Networks. So, in general, the performance of the network is fairly good, since the only mistakes happen if the noise is way too high for the digit to be recognized as such (even by a human).
Speaking about numbers, a noise level of 3 yields noisy digits that are quite visually unrecognizable, but still yields almost always perfect results. Higher noise levels greatly increase the chance of ending with a different attractor. In general, the number of required iterations is quite low, for example for a noise level of 3, 10 iterations are enough to reach an attractor, on average. In figure 16 we can see that the network can reach spurious equilibrium points starting from a noise level of 10.

To conclude on Hopfield Newtorks, in general, we can say that they move inputs to the closest attractor in a finite number of iterations, and if taken as a black box they behave similarly to Nearest Neighbor algorithms. If we have an input dimensionality D and A attractors, the Nearest Neighbor has a complexity that depends linearly on both A and D (using the most naive method). The same problem on a Hopfield Network requires D^2 weights and multiple iterations, resulting in a potentially worse performance, with also the complications deriving from spurious attractors and unstable equilibrium points. Nearest Neighbor can also work with an arbitrary number of attractors, independently from input dimension. The importance of Hopfield Networks is, though, also in the parallelism with animal memory.

Figure 16: Digit recognition, noise=10.

2.2 Time Series with Neural Networks

In order to see how the performance of the network depends on the lag p and on the number of neurons, figure 17 shows an interesting plot. On the X axis there is the input lag, on the Y axis the number of hidden units and on Z the MSE (computed on the normalized dataset) in a logarithmic scale. The training is done using `trainscg` algorithm, that can easily cope with high numbers of parameters. The curves that appear lower in the plot are the training curves, while above them there are the test curves. As we can see, the training error quickly decreases both by increasing the number of hidden units and the lag, so it is easy to fit the training set and the results on it won't be shown in detail, but in order to see how it actually fits to the test we need to take a look to the upper curves. They appear to have a slight trend in getting better towards a higher input lag, while showing no particular improvements above a handful of hidden units. In order to asses the generalization capabilities without looking at the test performance, one should start by counting the number of parameters of the network. In this case, given N hidden units and p lag, this number is $N * (p + 2) + 1$, because p increases the number of inputs to the system. Thus, we expect not to be able (in terms of generalization) to use a sufficient amount of hidden units if we take p too high.

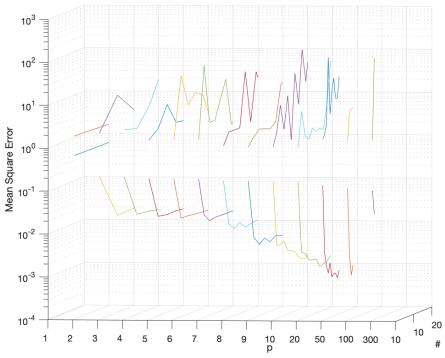

Figure 17: MSE vs Hidden Units vs lag.

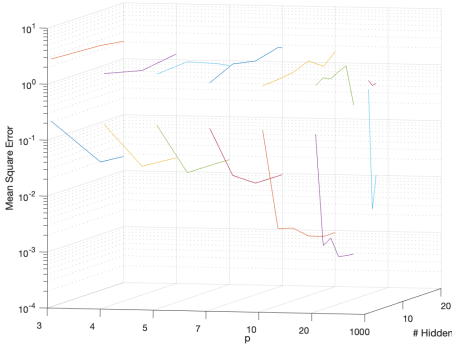

Figure 18: MSE vs Hidden Units vs lag - 10 trials.

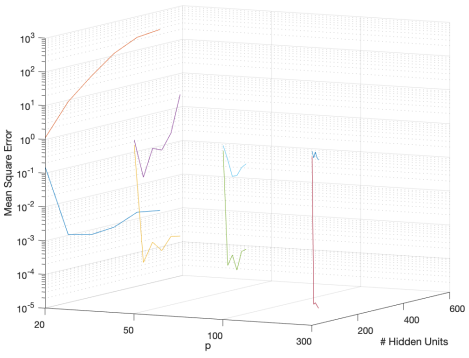

Figure 19: MSE vs Hidden Units vs lag - 10 trials.

In order to reach better conclusions, figure 18 shows a similar plot, but this time it is in a more restricted p range and each combination of p and hidden units has been trained 10 times and the best one was kept. The number of hidden units, as in the previous plot, has been limited for higher p values in order not to have too many parameters with respect to the $1000 - p$ training examples we can dispose of. In this plot we can see much smoother curves, indicating a clearer trend that higher values of p yield better results, since the network can access more past values of the time series. But in general there are not *big* differences between models, with the main problem being the difficulty in reproducing the amplitude drop at around the 60th test set sample (example in figure 20): in fact, to better reproduce that, the network should be able to access very old values, but this requires p to be too high for the network to actually generalize at all! LSTM Networks will provide a solution to this in the next subsection.

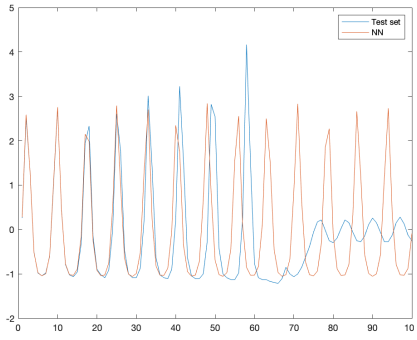

Figure 20: Typical result ($p = 10$, 11 units, $MSE = 1.9$).

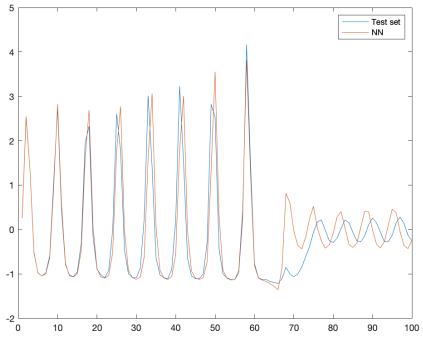

Figure 21: Good result ($p = 6$, 11 units, $MSE = 0.3$).

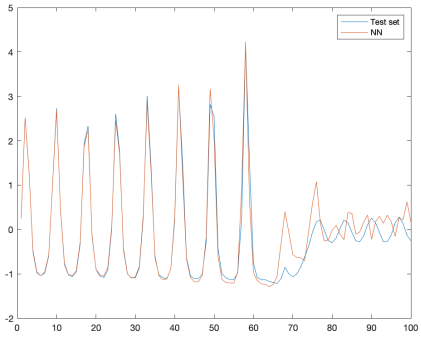

Figure 22: Very good result ($p = 50$, 41 units, $MSE = 0.1$).

Two good performances are plotted in figure 21 and figure 22. They provide a very small error on the test set, with the first one having much less parameters than the second one (and thus generalizes better), but the second one having anyway a smaller error (even with worse generalization, as the gap with the training error is higher). The problem with these two good performances is that they were produced after a lot of trials (10 trials for each parameter combination): the first one is the minimum test error of figure 18, the second one of figure 19. As a consequence, these results, while providing a good fit, wouldn't have been good models if we couldn't use the test set, which is the case of real applications. In figure 19, again, we can see that if one uses a very high p, correlated with a high number of

hidden units, one can have a really good fit (note the z-axis scale), since the network can process perfectly all the past values. In conclusion: good results *can happen*, but appear to be difficult to reach and with an unfairly high number of parameters in the network.

2.3 Time Series with LSTM Neural Networks

The LSTM model is much different from the previous one. First of all, we can see that it is part of the Deep Learning tools. Secondly, it is recurrent in training too, since it needs to know its current state in order to predict the next result, and in order to know its current state it has to have processed all the previous inputs. At high level, it works as follows: the input gate (together with the cell candidate) controls which input to add to the memory cell, that is inside the LSTM Network; the forget gate controls the amount of the previous state that is retained; the output gate controls the output of the network. All these gates have as inputs both the current input *and* the previous output. The output of the gates is then multiplied by the previous state (input gate) or by the state activation function of the current state (output gate). Thus, this particular structure can work as a memory that learns what to retain and what to forget in order to correctly predict a time series. It is particularly suited in our case, since there appear to exist "long" term correlations between values in the time series, but a high lag is unfeasible in terms of generalization and training set dimension $(1000 - lag)$.
The parameters to tune in this model are: the number of hidden units, that represents both the dimension of the output of the LSTM layer (which is then followed by a fully connected layer for single-valued regression) and of the internal cell state; the training algorithm; the epochs; the learning rate decrease in time; etc. The algorithm used is `adam`, which is suggested by literature to be more optimized than gradient descent for deep learning and is a recent very popular trend. Anyway, of course, it is much slower than the previous methods used for shallow networks, since the complexity of the network here is higher. Other algorithms can be `sgdm` and `rmsprop`, but they appear to be outperformed by `adam` in our case. The epochs were set to 1000, in order to have a good fit. The learning rate decrease over time can be tuned by 2 parameters: the drop period and the drop factor. It was found that, in this example, if one uses more parameters (hidden units) it's better to decrease the learning rate after a smaller amount of epochs, for example after 50 epochs (drop period) multiply by 0.5 (drop factor); using less parameters, even 200 epochs 0.5 can deliver good results.

Speaking about results, with LSTM it is much easier to have good results with respect to the previous model, starting from as low as 30 hidden units and with great performance at 50. Moreover, this model correctly reproduces the big drop in amplitude of the signal that happens at around 60 in the test set. This, in fact, requires knowledge of the previous drops, that are very far for the Time Series MLP model, while this model can account for them thanks to its special structure that has an internal status whose purpose is to learn dependencies between the next value and the past history, not being limited by the number of inputs as for the previous NN. Also the first predictions in the future appear to be on average more reliable than the previous model. For these reasons I personally prefer LSTM for this problem.

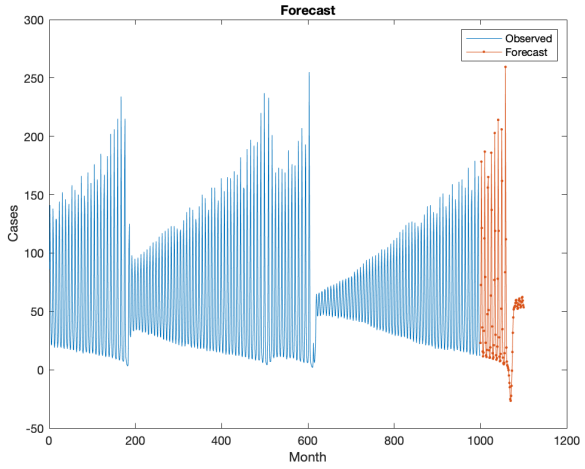

Figure 23: Training series followed by prediction.

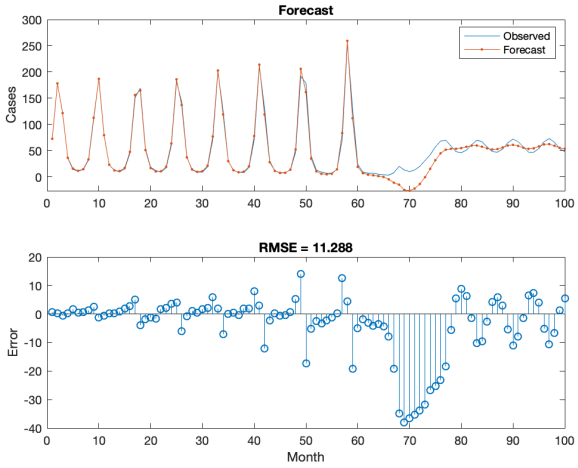

Figure 24: Prediction (50 hidden units, 1000 epochs, $MSE = 0.058$).

Results can be seen in figure 23 and figure 24. In addition to what has already been said, the MSE computed on the normalized test set is even lower than the best network of the previous exercise (0.06 vs 0.1), which has been selected after thousands of trials. The LSTM model can reach that performance with much more ease. The only way for the previous model to reach this performance with ease is to have huge p values, with the already mentioned problems.

3 Exercise session 3

3.1 Principal Component Analysis

The purpose of PCA is to reduce input dimension by minimizing the reconstruction error, i.e the difference between the original and the PCA reconstruction. A naive way to do this is by selecting the axes of the input that can be neglected because they contribute the least to the representation. Of course, a better way to handle this problem is by selecting a new basis that tries to identify the dimensions that contribute the most and the least to the input representation, without being constrained by the axes represented by the input variables. This corresponds to selecting the directions in the input hyperspace that have the maximum variance (hence a higher eigenvalue of the correlation matrix), while discarding the rest. The number of dimensions to keep can be selected.
If we apply this procedure to data which has been randomly generated with a Gaussian distribution, we don't expect the input dimensions to have any interesting correlation. In figure 25 we can see that, of course, the more dimensions (eigenvalues) we use for our representation, the smaller the error will be, finally approaching 0 if we use all of them. In orange, the sum of the unused eigenvalues decreases *close to linearly* with the number of used eigenvalues, indicating that input data is mostly uncorrelated. In blue, the reconstruction error (RMSE) decreases very similarly: there is a clear correlation, which will be explained later.

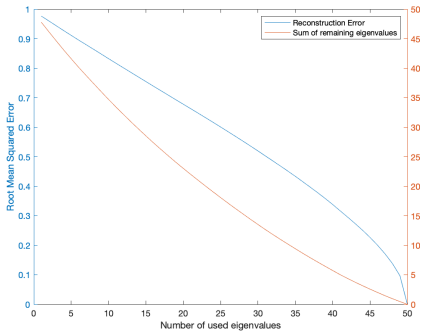

Figure 25: Gaussian input.

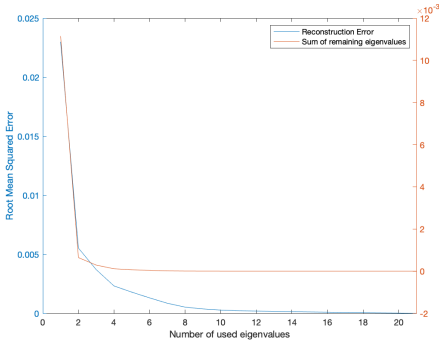

Figure 26: `choles` dataset.

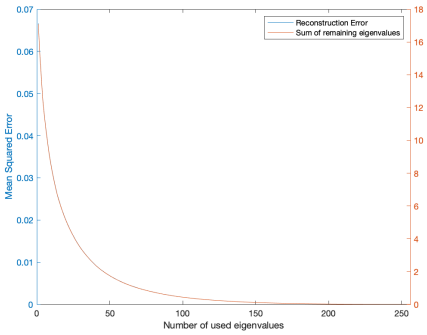

Figure 27: MSE in `threes` dataset.

If, instead, we apply PCA to some correlated data such as the `choles` datset, the results get more interesting. In figure 26 the error quickly decreases and the first few (2) eigenvalues are much higher than the rest of them and can thus be used to achieve a great reduction in dimensionality, much better than with uncorrelated data.

After further experiments with the MATLAB built-in PCA functions (`processpca`), it was found that the achievable results are very similar. The advantage of this function is that it allows to set a threshold to the reduction in error that you consider relevant (in form of a fraction). If adding a dimension reduces the error more than the set threshold, it is selected, otherwise it's not. The obtained results not identical, indicating that the underlying process is slightly different. In the subsequent exercises I will use my own implementation, since it can easily allow the setting of the number of used eigenvalues instead of the relevant error fraction.

3.1.1 PCA on Handwritten Digits (threes only)

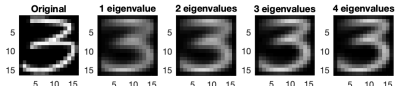

Figure 28: Reconstruction of a 3.

After computing the covariance matrix and taking a look at the eigenvalues we can notice that some eigenvalues are indeed much higher than the rest, and this can lead to a good image approximation with a low number of dimensions. Figure 28 shows the original image on the left, and the reconstructed images with 1 to 4 eigenvalues on the right: a single eigenvalue reconstructs a blurry image, but the '3' shape is already recognizable. It gets even better with more, of course.
Figure 27 is very similar to figure 26, in which the error is compared to the sum of the remaining eigenvalues. This time, however, the error is the MSE, no root. The strong correlation that was already visible before becomes a clearly linear correlation between the MSE and the sum of the remaining eigenvalues (quadratic if we consider RMSE), as the two curves get perfectly overlapped by adjusting the scale of the y-axis.
The error decreases very quickly with the used eigenvalues, until it seems to approach 0 when using them all. The error should actually be 0, because using all the eigenvalues means taking the N-dimensional input and representing it with N different orthonormal dimensions, resulting in a simple change of basis. In practice this is quite obviously not the case, since numerical variables have a finite precision. The error after full reconstruction is then $4.8070e-16$.

3.2 Stacked Autoencoders for Digit Recognition

In this exercise, with the goal being to correctly classify a test set of digits with randomized transformations, I try to investigate the effect of the network parameters on the final result.

First of all, the default settings of the exercise already provide a great performance, reaching a 3-runs average of 99.76% accuracy[4] with 100 units in the first autoencoder and 50 units in the second. I tried training MLPs tenths of times, averaging the performances and/or taking the best ones. The performances were fairly good, but comfortably under 98% for any network configuration and number of layers. With one hidden layer with 50 units, the best performances are around 97.6%; with 100 units they can also be less. 2 hidden layers do sometimes provide slightly better results, but also worse from time to time. 3 hidden layers seem to perform a bit worse, independently from the number of units.
The problem with MLPs is that they are initialized with random weights, and if they have a high number of parameters (which is easily the case when the input representation is high dimensional), the hypothesis space is so huge that it becomes impossible to actually find a good minimum. For this reason, MLPs with so many parameters do not even tend to overfit, because their actual VC-dimension is lower than what it seems, since navigating the huge hypothesis space is unfeasible. Another problem that arises when adding multiple hidden layers is that the backpropagation algorithm can quickly have problems like the vanishing gradient, since to get weight updates to the first layers one should multiply the partial derivatives by the weight values of the subsequent layers. Also, using more complex and reliable techniques such as Levenberg–Marquardt becomes quickly impossible due to the high complexity of the network.
Back to the training of autoencoders, playing with `MaxEpochs` can easily produce bad results. It has been best to set it to a value that guarantees a good fit, already in the pre-training phases. This value depends on the specific configuration, but 400 for the first and 100 for the second have always provided decent results. In terms of number of layers, it was found that using more than 2 autoencoders doesn't seem to produce great results, probably because the problem is not so complex and the input size is not big enough. Using only one, instead, while being slightly worse than 2, still yields interesting results: it can reach a 99.6% accuracy after fine tuning, and also already provides a great accuracy with pre-training only. Depending on the configurations and on the number of layers and units, a good pre-training performance can reach 90% or above, while other configurations can go much worse. The more layers one uses, the worse the pre-training performance gets; with one single autoencoder it is possible to reach the best pre-training performances. This is due to the fact that using more autoencoders reduces multiple times the dimensionality of the input[5], ending up by providing to the final classifier a representation that is not close enough for it to work well without fine tuning. Using a single autoencoder limits this process, since the dimensionality-reduction layer is only one and is explicitly trained on the real input. Anyway, the importance of fine tuning is crucial, especially with multiple stacked autoencoders, because the features learned by the autoencoders are good to reconstruct the input, but not yet perfect for classification purposes. Moreover, multiple autoencoders will also deviate more and more from the perfect input, requiring fine tuning in an even more explicit way.
Comparing these concepts to the MLP training argument above, it is immediately clear that pre-training the autoencoders gives a big advantage over random initialization of MLPs: on the one hand it provides a good starting point in the huge hypothesis space, reduces the input dimensions and the vanishing gradient problem; on the other hand it provides this with unsupervised learning, and thus it has a great positive effect on generalization. In fact, while the number of parameters can seem high, the biggest part of training is done in an unsupervised way, thus constraining the hypothesis space into a much smaller area that can be explored in a supervised way.
To answer the question about the number of layers, it seems that 1 single autoencoder can already perform better than any number of layers in an MLP, in this specific task.

Back to parameter optimization, the best results were obtained by setting up a network with 2 autoencoders with 100 and 50 units, and greatly increasing the sparsity proportion, that otherwise constrains too much the activation of neurons, up to 0.4. This configuration easily delivers more than 99% accuracy, going up to 99.86% (and with 85.82% before fine tuning). In figure 29 we can see the weights of the first autoencoder: by looking closely, it is possible to recognize some curly lines, typical of numbers.

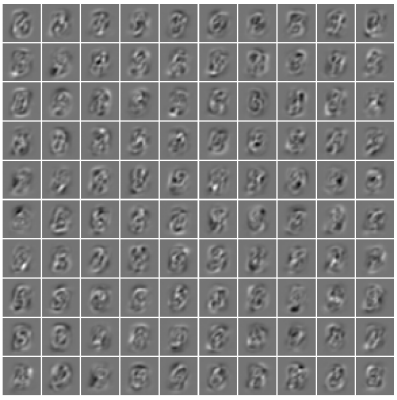

Figure 29: Neurons in the first autoencoder.

3.3 Convolutional Neural Networks

Referring to the exercise `CNNex.m`, what follows is the answer to the questions.

The size of weights of the second layer (thus the first Convolutional Layer) are [11 11 3 96]. This means that the second layer can dispose of $11 * 11 * 3 * 96 = 34848$ different weights, and that each filter is 3-Dimensional (11x11x3), since it has to be applied to the 3 input channels. 96 is the number of different filters, and thus feature maps.

The input images are 227x227x3, and the first filters are 11x11x3, with stride 4 in both directions and no padding. This means that each filter is shifted by 4 pixels (not 1) before each new output computation. In each direction, the filter is applied starting from pixel 1, 5, 9, 13, and so on. The total number of horizontal applications is $\lfloor (227-11)/4 \rfloor + 1 =$

[4] This and all the subsequent values are computed on the test set.
[5] And the trainings are chained, with the subsequents not influencing the precedents, thus increasing the reconstruction error.

55, the same number being also true for vertical, and thus leading to a 55x55 output for each filter, so 55x55x96. The Max Pooling Layer applies the max function to 3x3 patches with stride 2 in both directions and no padding, thus, with the same formula, we end up with $\lfloor (55-3)/2 \rfloor + 1 = 27$. So the input to layer 6 is 27x27x96, values confirmed with a simple check with `activations()` function.

The final layer, before Softmax, is a fully connected layer with 1000 units, which is the number of classes of the original problem. The input dimension was 227x227x3, which is 154587 inputs, so it was a huge problem. The actual input used for classification is the output of the last fully connected layer before classification itself, which has 4096 hidden units. Another fully connected layer precedes it, with again 4096 units. Thus, after convolutions, poolings and dropouts, it ends up with 2 layers with 4096 neurons each and an output layer with 1000 neurons, so the total is 9192. This is of course much lower to what it would have been if we used fully connected layers directly at the input images!

3.3.1 CNNs for Digit Recognition

For this task, several architectures and parameter combinations have been tried. First of all, the training algorithms used were both the default `sgdm` and the already mentioned `adam`.
In terms of architecture in general, using 2 convolutional layers proved to be better than 1; more than 2 proved to be too many. In fact, adding more convolutional layers implies the necessity of high dimensional input, which we don't have in this exercise. Moreover, digits are small and simple structures, that do not need a deep architecture to be recognized, such as for images of objects. If the first layer learns to detect edges and blobs, the second layer is applied to an area which is already large enough to detect features directly related to digits. Concerning the fully connected layers at the end, using more than one still provided good results, but using only one proved to be the best choice.

Another choice we can make is the number and dimension of filters. If in the first layer we decrease the number of filters the performance drops visibly (but not dramatically). If we increase them to 40, for example, the performance gets a bit better. The same reasoning is true for filter dimension: a bigger filter (10x10) performs better. If the bigger filter is accompanied by a higher number of filters, performance can quickly drop. Moreover, filter dimension plays also an intuitive role for visual feature detection in the input images, so it's important not to take them too large nor too small. Since the input is only 28x28, 10x10 filters seem to be already quite big for a 2-layer CNN.

Learning rate is also important (with also the drop rate over epochs), but usually *reasonable* parameters work fine. If they're not fine-tuned, they can lead to a bit more epochs required for good convergence, but still (usually) not too bad results. What actually played a bigger role is the `MiniBatchSize` parameter. A whole epoch has to scan 7500 examples, the batch size determines how many of them are to be considered for each descending step. The default batch size is 128. A higher batch size takes more *time* to converge with the same learning rate, since it requires less steps but each step takes much longer. The convergence is fairly stable, though, and this can be a positive factor. A smaller batch size, such as 32, can lead to a very quick convergence, after a smaller number of epochs, but of course it takes more time for it to reach the same number of *epochs* because the number of steps is much higher. By doing so, though, it *can* also reach better performances. Another pro of using smaller batch sizes is that they introduce some randomness in error minimization process of the network, and this *can* sometimes lead to better minima. Of course if one approaches too much the online learning paradigm, convergence can get slower and sometimes impossible.

Playing with these parameters has led me to a network with 2 convolutional layers, both followed by ReLUs. The first convolution applies 12 5x5 filters and is followed by a 2x2 Max Pooling with stride 2 and no padding. The second one applies 24 5x5 filters. Finally, a fully connected layer with 10 units performs classification. Training algorithm: `sgdm` (similar results with `adam`); learning rate: 0.0001; epochs: 15. The batch size, quite incredibly, has been set to 4. This network provided the best performance: **99.92%** accuracy on the test set, outperforming also the autoencoders of the previous exercise. In figure 30 and 31 we can see the feature maps of the first and the second convolutional layer, obtained through the `deepDreamImage()` function. The features of the first layer seem quite random, but the ones of the second layer are clearly dealing with lines, directions and curls. Finally, in figure 32, we can see the weights of the fully connected layer divided by neuron (class): some of them are recognizable as digits.

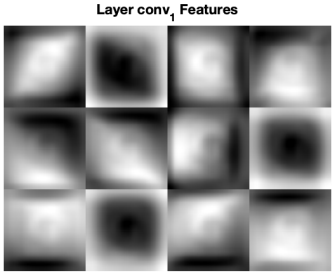

Figure 30: 1st Conv Layer.

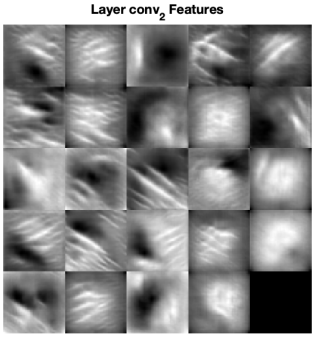

Figure 31: 2nd Conv Layer.

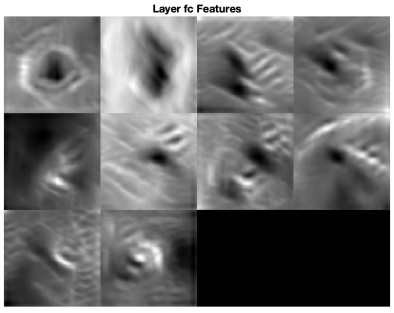

Figure 32: Output neurons.

4 Exercise session 4

4.1 Restricted Boltzmann Machines

About the training parameters, the first thing I notice is that the learning rate works similarly to previous models: a low learning rate makes the fitting much slower, but also prevents overshooting; a high learning rate means that learning can be theoretically faster, but with more chances of overshooting. Practically, keeping 10 as the number of components, a learning rate of 0.01 seems to be too slow. A learning rate comprised in the [0.1 0.5] range proved to be fast, and able to yield good results in only 10 iterations. Some lower learning rate values can also reach similar results, but with longer training times and a higher number of iterations. As a consequence, the number of iterations seems to be better to be high in general, provided that the learning rate doesn't cause divergence (i.e. the generation of "noise", instead of reconstructions of the input). With reasonable parameters, even with similar pseudo-likelihood scores, a longer training tends to provide better visual results.

Figure 33: RBM samples with 1024 units. Top: 10 Gibbs steps; bottom: 1000.

Figure 34: DBM samples after 10000 Gibbs steps.

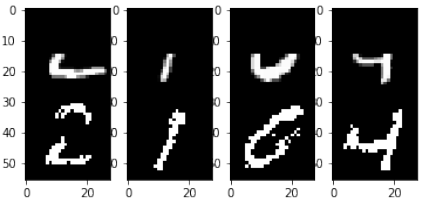

Figure 35: RBM with 1024 units, successful reconstruction of 15 missing lines with 10 Gibbs steps. Original numbers: 2104.

Anyway, the results obtained with only 10 components are not very encouraging. Reducing this number produces outputs that get very similar one to another, and it doesn't seem to be a great idea indeed to have less components than the number of different digits. Using 20 components, for example, already gives much better results. By quoting Le Roux and Bengio[6], it can be shown that unless the RBM already perfectly models the training distribution, adding a hidden unit (and properly choosing its weights and offset) can always improve the log-likelihood. Thus, I will try with more hidden units, and this will increase the training time per iteration. 50 of them with 0.3 LR and 10 iterations give a much better result, but still struggle to provide a good representation of certain numbers (such as 5, which can easily become a 2 or a 3 after a few reconstruction steps). They get better with a diminished learning rate (0.05) and more epochs (100). So, in general, to have good results with more units it seems better to have lower learning rates (and potentially more iterations). With 100 units, even 10 epochs with a 0.1 LR provide interesting results: with a low number of Gibbs sampling steps the result is similar to the input; with a higher number most of the outputs are still recognizable as digits, but they often become a different one, with '1', '2', '3', '5' and '7' dominating the distribution. The number of Gibbs sampling steps applied to test images is important to assess results: a low number, such as 1, will provide an output that closely resembles the input. A single pass through the model will reconstruct the input based on what the model has learned, and will thus be closer to the input if the model better fits the training distribution. A single pass is also what is used for the CD1 training algorithm, and thus is explicitly optimized (in the training set). More steps will gradually converge to the model distribution, and will then show what the model has learned to produce. A good model will be able to produce an output distribution that closely matches the training distribution, thus generating new samples with similar characteristics to the training ones (but hopefully not identical, otherwise it's not learning anymore, it's memorization). The best results in these terms were obtained with 1024 hidden units, 0.05 LR and 120 iterations[7], and can be seen in figure 33. Increasing too much the number of hidden units can eventually (and theoretically) lead to the mentioned memorization, since RBMs learn a representation of the input in a similar fashion to autoencoders. The aforementioned model is able to reliably reproduce the input from 1 to 10-50 (depending on the digit) Gibbs steps. More steps will more likely result in a digit becoming another one, eventually approaching '0', '1', '2', '3', '4' or '7', which appear to be the digits "preferred" by the network, thus showing that the model hasn't learned the training distribution too well. We still expect digits, though, even after lots of steps, because the model

[6] Le Roux, N., & Bengio, Y. (2008). Representational power of restricted boltzmann machines and deep belief networks. Neural Computation, 20(6), 1631–1649.

[7] Idea obtained from the code in `https://github.com/yell/boltzmann-machines`. My own best model was 100 units, 0.01 LR and 30 it (or 0.005 and 100 it), but proved to be worse.

should have learned to produce digits from the training distribution. A quality-wise note: the output images of the network are nice but I expect deeper networks to produce more realistic results, with less corruption.

Concerning reconstruction, in general deleting less than 8-12 lines will result in a good output, deleting more, not always: in figure 35, 15 lines were deleted and reconstruction was still possible. Of course this depends on the position of the deleted lines, which digit is being displayed and how it was written. Some digits, like 0, are more recognizable than others. A 9 can easily become a 7 if the too many rows are deleted, and so on. Moreover, deleting central rows often results in a worse output. In order to reconstruct the image, the RBM training parameters that were found good for the sampling exercise are still good. The better the fit of the model to the training distribution, the easier it is for the model to produce valid digits in general, and thus to reconstruct missing inputs in particular: a higher number of units is then better (without exaggerating), and so is true for the number of iterations. The learning rate, as before, has to optimize a trade-off, and can thus be played with. The Gibbs sampling parameter here appears to play a crucial role: if it's too low the digit *may* not be fully reproduced; if it's too high it may switch to a different digit. 5-10 usually proved to be fine.

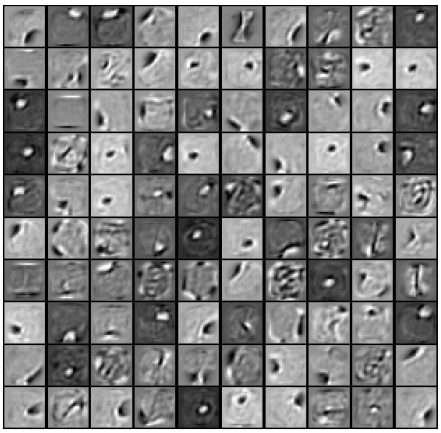

Figure 36: RBM with 100 units.

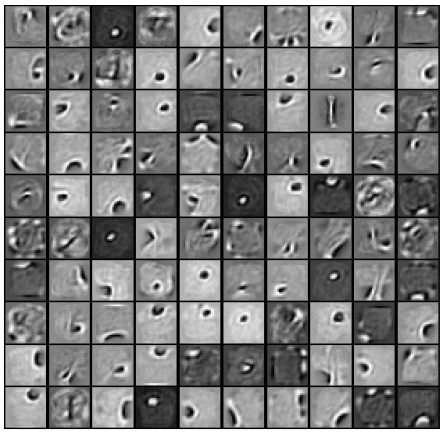

Figure 37: First layer of DBM.

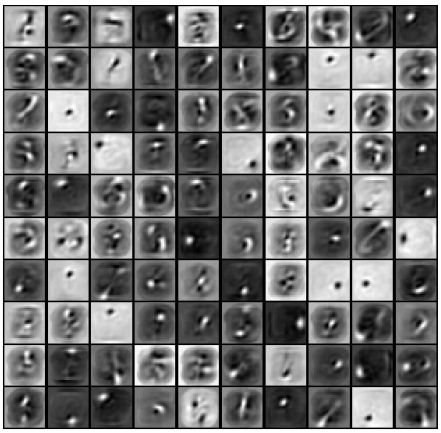

Figure 38: Second layer of DBM.

4.2 Deep Boltzmann Machines

The first layer of the pre-trained DBM shows features that appear quite similar to those extracted in the previous exercise (figures 36 and 37), in the sense that they're extracting simple patterns like circles, curls, angles and small lines. The key visible difference here is that the feature extracted from the DBM seem to be cleaner: considering them images, they have less "noise" than the ones from my RBM. This can indicate that they have learned to better recognize those patterns. The features of the second layer (figure 38), instead, appear quite different, and generally more blurred. Some of them seem to comprehend multiple features from the first layer, and they appear more complex anyway, because they are extracting higher level features from the data, with some of them being basically digits. If the first level features are extracted from the input, these are extracted from the first level, and can thus provide higher level information, similarly to what happens with autoencoders.

The sample quality here is incomparably higher (figure 34). The images resemble real digits, with little-to-no noise and very few errors (images that do not resemble any digit). Also, when sampling with much more Gibbs steps, the results are still of the same quality (of course the digits themselves change), and the distribution of the generated examples seem to be really random, with no digit dominating the rest, as opposed to what happened in my previous model. This indicates that the fit to the training distribution is better, and is due to the fact that DBMs are much more capable machines than RBMs. Boltzmann machines are in general capable of learning probability distributions, but of course the restricted version of them imposes some limitations: no relationships between hidden units are allowed. This limits their capabilities but makes the learning process feasible. DBMs essentially maintain the learning efficiency of RBMs, while providing relationships between the hidden units (of different layers), and thus being more capable than RBMs. Just as MLPs, RBMs can theoretically represent any discrete distribution if enough units are used[8], but using deeper networks makes this much easier.

4.3 Generative Adversarial Networks

After training the GAN for a long time (50k batches) on different classes, I noticed that it is able to produce quite realistic results. They can appear blurry or sometimes with weird proportions, but I would overall say that they belong to the class they're supposed to. The loss and accuracy of both the networks start in a quite random situation,

[8]Freund, Y., & Haussler, D. (1994). Unsupervised learning of distributions on binary vectors using two layer networks. Tech. rep. UCSC-CRL-94-25, University of California, Santa Cruz.

with possible wide disparities between the generator and the discriminator, but after a small amount of batches the situations starts stabilizing around a smaller interval. The oscillations remain till the end of the training, though, indicating that training the GAN is difficult, because of its minimax structure. In some iterations the generator gets much better, and the discriminator can't really keep up, in some others it's the contrary, depending on the gradient of the objective (reward) function[9]. As a consequence, the overall time evolution of both loss and accuracy is not very stable, with the discriminator loss ranging from 0.5 to 0.75; accuracy usually above 0.5 (of course); generator loss from 0.9 and 1.3 (higher than discriminator); accuracy from 0.1 to 0.45. These oscillations don't prevent the GAN from reaching interesting results, showing that it is anyway learning to generate something. The model is quite stable in generating good samples, meaning that when it starts doing so it keeps up in the following iterations.

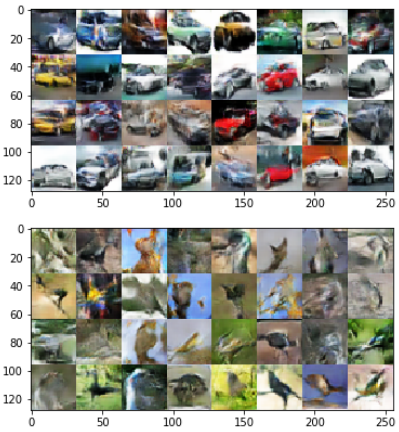

Figure 39: Cars and birds generated after 50k batches.

4.4 Optimal Transport

Using OT to transfer color between images can produce interesting results, but it also has some limitations: since it optimizes the transportation map, it can lead to a map that assigns very different colors to pixels that, in the original coloring, were actually quite similar. This produces irregular images that feel badly colored, for example the central column in figure 40. Using a regularization term, instead, can help fixing this problem: using the entropy (of the map) as a regularizer means that we try to find a balance between the best mapping and an equal mapping, resulting in a much more realistic coloring, with less color spikes. The higher the weight on the regularization term, the smoother the image coloring (and the worse the optimality of the mapping).

4.4.1 Regarding WGANs and the elaboration of the acquired knowledge

The first thing to notice in using WGANs is the performance drop, requiring almost double the time per iteration. Set this aside, it's possible to notice that in quite a large number of situations the discriminator has a natural advantage over the generator: the discrimination task is easier. When the discriminator has a good performance, and thus the generator a bad one, it is very difficult for the generator to recover following the gradient of the reward function for vanilla GANs, because it can produce exploding and vanishing gradients (depending on the reward function). Wasserstein GANs have a much more linear gradient because they don't output probabilities with a log function, they output scores, and *can* thus produce cleaner results[10]. The Wasserstein distance deriving from the OT is used here as a metric to compare the distribution of the training data with the distribution of the generated samples. The objective of the WGAN is then to minimize this distance, exploiting better gradients than normal GANs.
Comparing the two provided models, I can notice that the GAN stably produces quite good results after lots of iterations, but it isn't very stable, of course, in terms of loss function and accuracy. The WGAN appears to be more stable and produces cleaner images, with much less random white pixels that can be defined as noise, thus producing more distinct shapes even with a low number of epochs, even though these shapes do not resemble digits or are very biased towards 0's and 1's. Training the models for more time, the GAN greatly reduces its noise and the WGAN starts converging to produce actual digits, both models getting better. Overall, I'm more impressed with the performance of GAN in terms of how real the numbers feel, set aside the noise, because while the WGAN's images are better with "few" iterations, GAN's images get much better over time, eventually outperforming WGAN in this example.

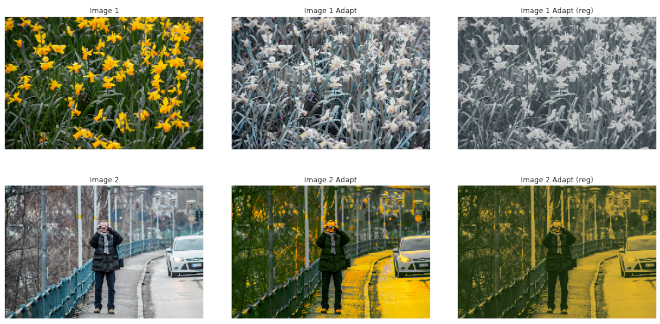

Figure 40: Color transfer with OT. Right column: with regularization.

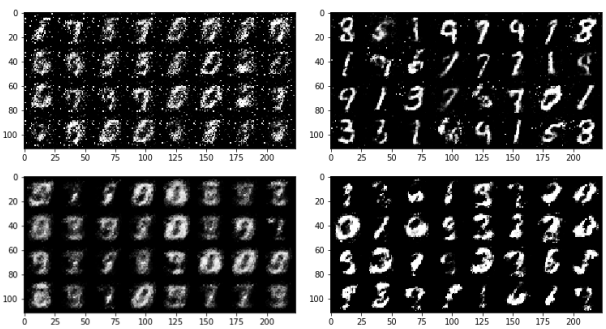

Figure 41: Top left: GAN after 5k iterations; top right: after 50k iterations. Bottom: WGAN.

[9] A regularization would be interesting to try here, and WGANs could actually be seen as a form of gradient smoothing, especially when using the Sinkhorn distance.
[10] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In ICML, pp. 214–223, 2017.