Auto-Encoder with Optimised Architecture for Unsupervised Classification

Aymene Mohammed Bouayed
Department of Electronics & Computer Science
University of Sciences and Technology Houari Boumediene
Bab-Ezzouar, Algiers, Algeria
`bouayedaymene@gmail.com`

Hoceine Kennouche
Department of Electronics & Computer Science
University of Sciences and Technology Houari Boumediene
Bab-Ezzouar, Algiers, Algeria
`hocine199902@hotmail.fr`

Abstract

This project [1] deals with the development of an Auto-Encoder with an optimised architecture for the task of unsupervised classification. More specifically, this project focussed on two principal aspects. Firstly, we harnessed the power of auto-encoders to compress and extract the most important information from the input data. The encoding produced by the encoder prove to be very informative, resulting in clusters that are inline with the classes of the input data. Secondly, we have used the Genetic Algorithm to search for the optimal architecture of the auto-encoder that maximises the unsupervised classification accuracy. This is very helpful as the task of hyper-parameter tuning is a laborious task requiring a lot of intuition and testing of numerous combinations of parameters. The produced model combined with a simple linear classifier achieved an accuracy of 96.32% on the MNIST dataset.

Keywords: Auto-Encoder, Unsupervised Classification, Neural Architecture Search, Genetic Algorithm, MNIST.

1 Introduction

Since the 60s deep learning and neural networks have invaded many aspects of our lives and they achieved great success in many tasks. Classification is one of the most important tasks in deep learning. It consists of assigning a label to an input. The input could be an image, an audio file, a video, a simple vector of values . . . etc. The task of classification has numerous applications such as emails classification like spam detection, disease identification, celestial object classification, and so on. Current deep learning technics are able to achieve great performance on this task when labeled data is available. Unlucky, most of the world's data is unlabelled. As a result, researchers have thought of many algorithms and methods to do unsupervised classification like using the K-Means algorithm (4), Auto-Encoders (3), Generative Adversarial Networks . . . etc. In our project, we will be using Auto-Encoders to do unsupervised classification. This is achieved through the creation of an encoding of the elements of the dataset that will be used in the classification. Moreover, the Genetic

[1]This work was done as a semester projet for the course "Neural networks and machine learning".
September 9, 2020

Preprint. Under review.

Algorithm will be used to optimise the architecture of the Auto-Encoder in order to find the optimal hyper-parameters and maximise the classification accuracy.

2 Auto-Encoders

2.1 Definition

An auto-encoder is a special kind of neural network architecture. It is used mainly to learn an abstract representation (an encoding) of the most important features of the input in an unsupervised manner. The obtained encoding is then used to reconstruct the input (2).

Auto-encoders are composed of two neural networks an encoder and a decoder in the shape of a funnel and a reverse funnel respectively (See figure 1). They are stacked one on top of the other and trained at the same time. The objective of the encoder is to compress the input features from a large dimensional space to a more compact one while keeping all the important information. The decoder takes as input the encoder's output and tries to reconstruct the input the best way possible (2).

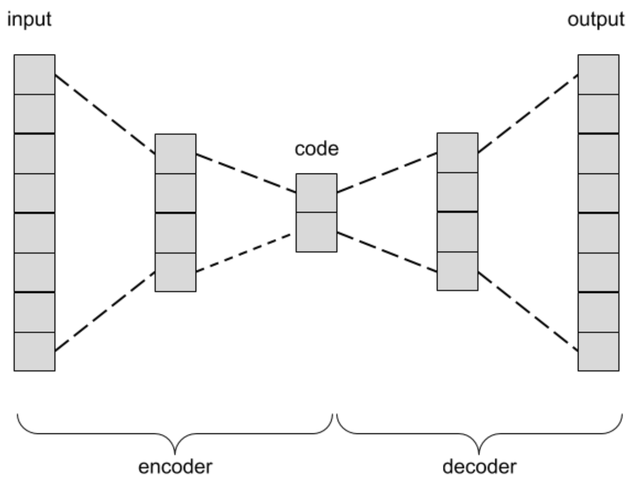

Figure 1: Auto-encoder

2.2 Training an Auto-Encoder

Auto-encoders are trained using the back-propagation algorithm or one of its variants. First, the data is passed through the encoder to obtain an encoding.

$$e \leftarrow encoder(X);$$

Then, the encoding e is passed through the decoder so that it tries to reconstruct the input using only the encoding e.

$$X' \leftarrow decoder(e);$$

After that, the reconstructed input X' is compared to the true input X and a loss is calculated using a loss function such as the log-loss or the MSE.

$$loss \leftarrow LossFunction(X, X');$$

Using the value of the loss, the weights of the encoder and the decoder are updated as if they were one network following the same steps as in a simple multi-layer perceptron using the back propagation algorithm.

Algorithm 1: Training an auto-encoder

```
Pre-process the training data (normalization, data augmentation, deal with missing values ...etc).
  epoch ← 0;
while epoch < maxEpochs do
    for X in trainingData do
        e ← encoder(X); // e is the encoding of the input X
        X′ ← decoder(e); // X′ is the reconstruction of the input X produced by the decoder using
          the encoding e.
        loss ← LossFunction(X,X′);
        updateWeights(encoder, decoder, loss);
    end
    epoch ← epoch + 1;
end
```

3 Proposition

To concretise the idea of unsupervised classification using encodings of the data, an auto-encoder network can be used. The output of the encoder represents the encoding of the input and the decoder reconstructs the input starting from this encoding. It can be noticed when plotting these encodings onto a 2D space that the encodings of input data of the same class are closer to each other forming clusters for each class. This allows to distinguish between the classes and hence perform unsupervised classification.

In addition to unsupervised classification this approche has many advantages like :

- The clustering will make the task of labelling the data much easier. One has to label only one element per cluster.
- This approche could be implemented into a commercial business where the encodings can be stored in the database instead of the actual data which provides security and storage space saving while the classification of the data in embedded in the encoding.

So in order to actualise the proposed idea, we used an auto-encoder with fully connected layers in both the encoder and the decoder in conjunction with the ReLU activation function that has proven to be very effective in helping neural network converge faster and achieve high accuracies (see figure 2) (3). Also, to find the architecture that yields the best performance we use the Genetic Algorithm to explore the search space of the different hyper-parameters of the architecture (6).

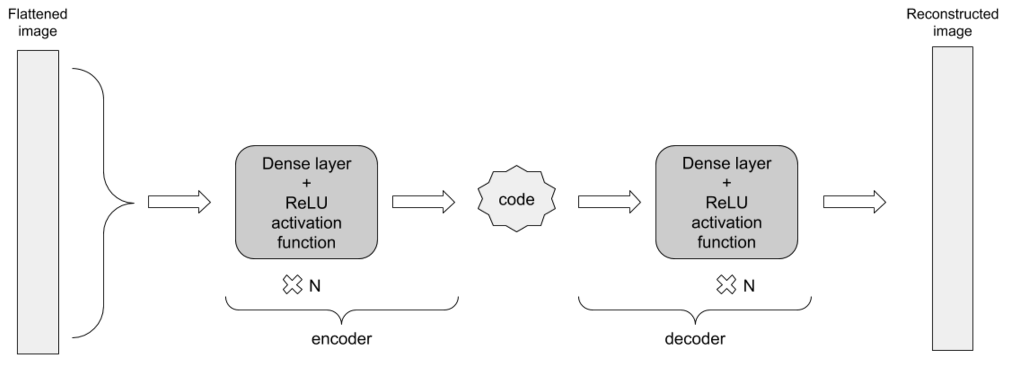
Flattened image
Dense layer + ReLU activation function
N
encoder
code
Dense layer + ReLU activation function
N
decoder
Reconstructed image

Figure 2: Proposed Auto-Encoder Architecture

4 Neural Architecture Search

Hyper-parameter tuning is a laborious task and arguably a complex one requiring a lot of domain knowledge, intuition and testing of numerous combinations of parameters. The hyper-parameters that need to be fixed are the number of dense layers in the encoder and the decoder and the number of neurones per layer. To counter the problem of the huge search space, the Genetic Algorithm (GA) have been used to explore it (see algorithm 2). The choice of the GA is motivated by the fact that it doesn't take many epochs to converge and it is relatively simple to implement (6).

Even so, in order to reduce the computing time, few restrictions to shrink the search space have been taken and they are :

1. Considering the number of layers of the encoder to be equal to the ones of the decoder (The decoder is composed of the same number of layers and neurones as the encoder but the reverse order). Since it takes as many steps to encode an images as it takes to decode it. Also, we limited our selves to a number of layers in {1,2,3,4,5}.

2. When building an architecture we try to create one that resembles a funnel. Thus, this can change during the crossover operation in the GA.

3. We fixe the batch size at the value of 100.

4. We use the Adam optimiser which adapts its parameters automatically according to the loss.

5. For each configuration (or architecture) we build a network and train it for 10 iterations, then test it on the test data using a linear classifier and the accuracy achieved is the value of the fitness. The number of iterations is fixed to 10 as it is sufficient to asses the effectiveness of an architecture without taking too much training time.

Algorithm 2: Genetic Algorithm For Hyper-parameter Tunning

```
population_array ← create random initial population and calculate the fitness of its individuals.
Sort(population_array, fitness) // sort the population_array according to the fitness of the
  individuals.
epoch ← 0;
while epoch < maxEpochs do
    it_crossover_rate = random(0,1)
    if (it_crossover_rate > crossover_rate) then
        child1, child2 = crossover(population_array[0], population_array[1]) // crossover and
          calculate fitness.
        population_array.append(child1, child2)
    else
        child1, child2 = population_array[0], population_array[1]
    end
    it_mutation_rate = random(0,1)
    if (it_mutation_rate > mutation_rate) then
        child3 = mutate(child1) // mutate and calculate fitness.
        child4 = mutate(child2) // mutate and calculate fitness.
        population_array.append(child3, child4)
    end
    Sort(population_array, fitness) // sort the population_array according to the fitness.
    epoch ← epoch + 1;
end
```

4.1 Training parameters

In training our models, we have used the Adam optimiser due to the fact that it converges quickly and it adapts its parameters according to the current loss (see appendix A). The loss is calculated using the Mean Squared Error function (MSE) between the reconstructed image and the original image which are both flattened out to be in the form of a vector of 784 dimensions. The MSE loss function is usually used in the task of regression and the task of reconstruction of an image can be viewed as a regression of each pixel of the image.

4.2 Model evaluation

To asses the effectiveness of the model's architecture we have used a linear classifier because when the clusters are well formed, a line should be more than enough to distinguish between them. More specifically we have used Support Vector Machines (SVMs; see appendix B) for this task.

After training the auto-encoder we use the training dataset without any data augmentation to extract the encoding of the images. Then, we use these encoding in conjunction with the labels of the images to train the SVM. Finally, we use the network to extract the encodings of the images of test dataset and evaluate the accuracy of the cluster formations using the SVM (see algorithm 3) (1). This value of the accuracy represents the fitness of the architecture in the GA.

Algorithm 3: Calculating the fitness

```
train_encodings ← encoder(train_data)
test_encodings ← encoder(test_data)
svm ← SVM()
fit(svm,train_encodings,train_target)
fitness ← score(svm,test_encodings,test_target)
return fitness
```

5 Data preprocessing

For this project, we have chosen to illustrate the task of unsupervised classification using auto-encoders on the MNIST dataset (9). The MNIST dataset comes in the form of images of gray scale hand-written digits with a shape of 28x28x1. Before training any of the neural networks, input images have been normalised so that the values of the intensity of the pixels are in the range [0,1]. This helps the neural network converge faster by avoiding gradient explosion. Also in an effort to make the network robust to small changes in the input images, data augmentation of the dataset have been introduced. This was done by randomly rotating either clock-wise or counter clock-wise the images, by 30°or 45°. Or doing a random affine transformation on the images which consists of a maximum rotation of 45°, a scaling down of the image by down to 0.5 times or scaling up by up to 1.5 times and a shear of up to 5 pixels. Finally before the images are fed to the neural network they get flattened out to be a vector of 784 dimensions. This is done so that the image can go through the dense layers of the network which do not accept matrices as input.

6 Development Environment

6.1 Software Environment

For the implementation of the proposed auto-encoder and for data preprocessing we have used the following libraries :

- **Pytorch** an open source machine learning library, used for deep learning applications such as computer vision and natural language processing. It provides an implementation of the most used neural network layer, activation function, loss function, variations of the back propagation algorithm . . . etc (10).
- **Torchvision** a Pytorch package that contains of popular datasets such as the MNIST dataset and common data augmentation transformations for computer vision (10).
- **Sci-kit learn** an open source machine learning library. It features various classification, regression and clustering algorithms including linear classifiers and k-nearest neighbours.
- **Matplotlib** a general purpose plotting library for the python programming language.

6.2 Hardware Environment

Our different neural networks have been trained on GPU using the Google Colaboratory website.

7 Experiments

To find the optimal parameters of the GA and the number of layer of the auto-encoder. We start with the following configuration:

- Number of auto-encoder layers : 1
- Mutation rate : 0.1
- Crossover rate : 0.1
- Epochs : 5
- Population size : 5

Then we change parameter per parameter and when we move from one to another we keep the best value. This configuration allows to explore up to 25 different architectures in 5 epochs in an "intelligent way" then it outputs the best 5.

7.1 Number of layers of the Auto-Encoder

In this part, the GA searches for the optimal number of layers in the network. We first start by exploring the number of layers of the auto-encoder with the current configuration of the GA because the mutation and crossover rates are very low allowing much more diversity in the population. Hence exploring the impact of each number of layers on the accuracy. We try a number of layers in {1, 2, 3, 4, 5}. The results found are shown in the Table 1 hereafter. The architecture column shows the number of neurones in each layer of the encoder and the decoder. As it can appreciated from the table 1 an architecture with 5 layers achieves the best accuracy with almost 95%.

N°of Layers	1 Layer		2 Layers		3 Layers	
	Architecture	Accuracy	Architecture	Accuracy	Architecture	Accuracy
1st solution	**631**	**91.85%**	**457-470**	**92.43%**	**483-398-292**	**93.69%**
2nd solution	391	91.52%	358-291	91.5%	440-156-292	93.62%
3rd solution	468	91.52%	662-470	91.5%	440-350-292	93.54%
4th solution	677	91.51%	489-367	91.49%	483-454-292	93.54%
5th solution	391	91.49%	457-291	91.33%	483-454-292	93.42%

N°of Layers	4 Layers		5 Layers	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**537-490-241-173**	**94.47%**	**486-454-234-180-173**	**94.98%**
2nd solution	537-490-241-80	92.80%	486-454-211-180-174	94.69%
3rd solution	537-490-241-80	92.52%	486-454-234-180-174	94.62%
4th solution	314-181-101-98	92.23%	486-454-234-180-174	94.24%
5th solution	574-444-335-72	92.16%	486-454-211-180-107	93.75%

Table 1: Table showcasing the top 5 accuracies obtained using the GA for different number of layers

7.2 Mutation rate

This section is dedicated to the search for the optimal mutation rate in the GA. We try the following rates {0.1, 0.3, 0.6, 0.9}. The results found are shown in Table 2 hereafter. It can be appreciated that the best accuracy of 94.98% was obtained with a mutation rate of 0.1.

Mutation Rate	0.1		0.3	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**486-454-234-180-173**	**94.98%**	**108-310-227-193-163**	**94.22%**
2nd solution	486-454-211-180-174	94.69%	524-310-227-193-163	93.98%
3rd solution	486-454-234-180-174	94.62%	524-310-227-54-163	93.76%
4th solution	486-454-234-180-174	94.24%	524-310-227-193-163	93.71%
5th solution	486-454-211-180-107	93.75%	524-310-227-193-163	93.47%

Mutation Rate	0.6		0.9	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**696-237-49-45-29**	**90.08%**	**610, 365, 332, 74, 86**	**92.96%**
2nd solution	635-122-49-45-29	89.63%	610, 365, 332, 74, 40	92.78%
3rd solution	658-237-49-45-29	89.05%	610, 365, 332, 74, 40	92.36%
4th solution	635-122-74-45-29	88.54%	281, 187, 118, 90, 86	91.77%
5th solution	696-122-49-45-29	87.90%	610, 365, 332, 74, 40	91.42%

Table 2: Table showcasing the top 5 accuracies obtained using the GA for different mutation rates

7.3 Crossover rate

To find the optimal crossover rate in the GA, we try the following rates $\{0.1, 0.3, 0.6, 0.9\}$. The results obtained are shown in Table 3 hereafter. It can be appreciated that the best accuracy of 94.98% was obtained with a crossover rate of 0.1.

Crossover Rate	0.1		0.3	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**486-454-234-180-173**	**94.98%**	**437, 334, 107, 83, 58**	**92.29%**
2nd solution	486-454-211-180-174	94.69%	437, 164, 107, 83, 58	91.30%
3rd solution	486-454-234-180-174	94.62%	316, 334, 107, 83, 58	90.61%
4th solution	486-454-234-180-174	94.24%	212, 164, 107, 83, 58	90.07%
5th solution	486-454-211-180-107	93.75%	437, 334, 107, 83, 36	89.78%

Crossover Rate	0.6		0.9	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**535-250-175-118-55**	**92.51%**	**315-158-52-22-19**	**87.45%**
2nd solution	390-250-175-118-55	91.83%	206-158-52-22-19	85.96%
3rd solution	535-250-175-85-55	91.72%	206-114-52-22-19	85.47%
4th solution	390-250-175-144-55	91.66%	199-66-59-36-34	84.96%
5th solution	390-250-175-85-55	91.41%	199-66-59-36-21	84.35%

Table 3: Table showcasing the top 5 accuracies obtained using the GA for different crossover rates

7.4 Epochs

In order to study the effect of increasing the number of epochs on the same initial population, we ran a code for 10 epochs and recorded the results at 5, 7 and 10 epochs. They are show cased in the columns "5 new", 7 and 10 of Table 4 respectively. The column "5 old" represents the best result obtained with 5 epochs.

We can appreciate from Table 4 that the optimal number of epochs of the GA is 10. It can be seen that the top solution didn't change but the rest of the individuals evolved. If we increase the number of epochs further the population will evolve even further and achieve a higher accuracy.

GA epochs	5 old		5 new	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**486-454-234-180-173**	**94.98%**	**351-416-158-97-87**	**93.43%**
2nd solution	390-250-175-118-55	91.83%	576-416-158-97-87	92.75%
3rd solution	535-250-175-85-55	91.72%	92-416-158-97-87	91.59%
4th solution	390-250-175-144-55	91.66%	465-230-213-55-42	90.72%
5th solution	390-250-175-85-55	91.41%	214-36-213-55-42	88.94%

GA epochs	7		10	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**576-416-278-97-87**	**93.52%**	**576-416-278-97-87**	**93.52%**
2nd solution	351-416-158-97-87	93.43%	351-416-158-97-87	93.43%
3rd solution	576-416-158-97-87	92.75%	576-416-158-97-87	92.75%
4th solution	576-416-158-97-75	91.82%	351-416-158-97-83	92.65%
5th solution	92-416-158-97-87	91.59%	351-202-158-97-87	92.09%

Table 4: Table showcasing the top 5 accuracies obtained using the GA for different epochs

7.5 Population size

To study the effect of the population size of the GA, we try the following population sizes {5, 7, 10} . The results obtained are shown in the Table 5. We can see that the optimal population size of the GA is 10. As it allowed the whole population to grow not only the top individuals. But, the run with a population size of 5 surpassed it in termes of accuracy. This is mainly due to the initialisation of the population which was better which was better.

Population size	5		7	
	Architecture	Accuracy	Architecture	Accuracy
1st solution	**486-454-234-180-173**	**94.98%**	**642-241-109-99-99**	**93.27%**
2nd solution	390-250-175-118-55	91.83%	642-492-109-84-51	92.76%
3rd solution	.576-416-158-97-87	92.75%	642-241-109-84-51	92.57%
4th solution	576-416-158-97-75	91.82%	642-407-109-84-51	91.84%
5th solution	92-416-158-97-87	91.59%	481-329-313-203-61	90.97%
6th solution	-	-	642-241-109-99-51	90.85%
7th solution	-	-	481-329-313-203-48	90.69%

Population size	10	
	Architecture	Accuracy
1st solution	**459-455-402-190-115**	**94.55%**
2nd solution	667-455-402-190-124	94.40%
3rd solution	459-204-402-190-124	94.36%
4th solution	459-455-402-190-163	94.06%
5th solution	667-455-402-402-124	94.06%
6th solution	667-455-402-190-115	93.86%
7th solution	459-455-402-190-124	93.77%
8th solution	459-204-402-222-124	93.76%
9th solution	459-204-402-146-124	93.7%
10th solution	667-455-154-144-99	93.61%

Table 5: Table showcasing the top 5 accuracies obtained using the GA for different population sizes

7.6 Discussion

From Table 1 we can see that the best accuracy (94.98%) was achieved using 5 dense layers, which is the highest accuracy we have found. Subsequently, we can deduce that increasing the number of layers in the Auto-Encoder results in a better performance. This is because the input goes through many layers of feature extraction which results in a very informative encoding w.r.t the class of the input.

Table 2 and Table 3 show that the best performance was achieved using a mutation rate of 0.1 and a crossover rate of 0.1. These low values yield to a better exploration of the search space through the generation of more solutions thus approaching the best one.

The tests that have been run on the number of epochs (Table 4) and population size (Table 5) show that increasing the population size allows more diversity thus the individuals evolve much faster. Also, increasing the number of epochs gives more time to the population to mature and approche the optimal solution.

From our testing we found that the optimal parameters of the GA are:

- Mutation rate : 0.1
- Crossover rate : 0.1
- Epochs : 10
- Population size : 10

Through the numerous architectures explored by the GA, it can be seen that the auto-encoder with the following architecture 486, 454, 234, 180, 173 (5 layers) achieved the most promising accuracy. This architecture will be explored further in the upcoming section.

8 Evaluation and Discussion

In this section, the most promising architecture found by the GA is explored. This is done by training an auto-encoder with the best architecture found for up to 50 epochs to obtain the highest accuracy possible. The results regarding the training loss, test loss and accuracy in each epoch are summarised in the graphs of figure 3.

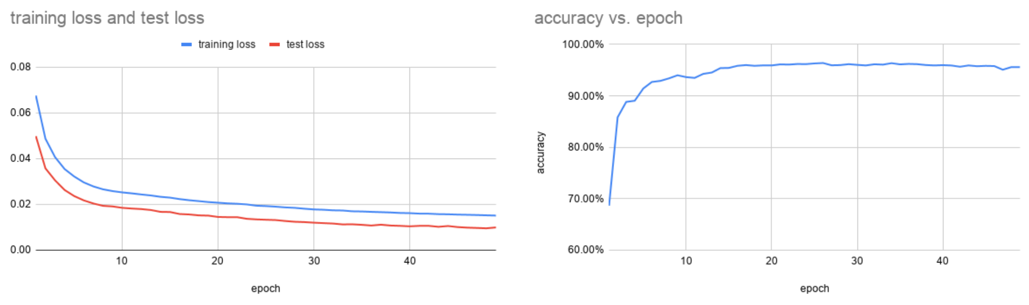

Figure 3: Evolution of the accuracy, training and test losses w.r.t the number of epochs

From the graphs in Figure 3, it can be observed that after around 25 epochs even tough the training and test loss decrease the accuracy no longer improves. This happened mainly because the encoding vector now contains the maximum information that can be extracted and fit into it w.r.t the task of classification.

The evolution of the clusters constructed from the encodings can be seen in Figure 4. These embedding were obtained after casting the encoding vector from $\mathbb{R}^{173}$ to $\mathbb{R}^2$ using the t-SNE algorithm (see appendix C). It can be seen that through-out the training iterations clusters form and become more defined for each semantic class. As a result, they can be well separated using a linear classifier like an SVM with low misclassifications rate.

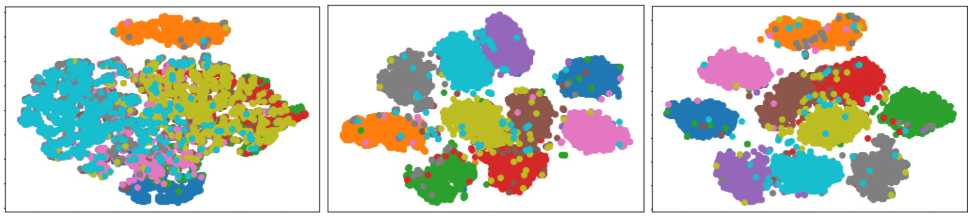
Figure 4: Evolution of cluster formation; *The picture on the left, middle and right represent the clusters after 1, 25 and 50 epochs respectively. Each colour represents a semantic class of the input images.*

Comparing the results obtained to the ones tackling the task of unsupervised classification found on the literature (see Table 6), It can be appreciated that on the MNIST dataset the accuracy obtained with a simple auto-encoder is remarkable. Improving the layers of the auto-encoder to convolutional ones and searching for the optimal architecture using them will further improve the accuracy.

Method	Accuracy
K-means (4)	53.49%
AE (3)	81.2%
GAN	82.8%
IMSAT	98.4% (0.4)
IIC	98.4% (0.6)
ADC	98.7% (0.6)
SCAE (LIN-MATCH) (1)	98.7% (0.35)
SCAE (LIN-PRED) (1)	99.0% (0.07)
Our AE	**96.32% (0.001)**

Table 6: Table showcasing the top accuracies obtained on the MNIST dataset using different methods(1)

9 Conclusion

In this work we have explored the task of unsupervised classification using neural networks. To do so we have used auto-encoders because the output of the encoder forms clusters for each semantic class. We have proposed an auto-encoder architecture composed of dense layers that achieved an accuracy of 96.32% on the MNIST dataset. The search for the optimal hyper-parameters of this architecture (number of layer and the number of neurones per layer) have been done using the Genetic Algorithm (GA). The choice the GA have been motivated by its quick convergence time. As a follow up to our work, a more advanced encoder and decoder architecture could be used such as using convolutional layers and/or a different heuristic to find the best hyper-parameters such as Bee Swarm Optimisation heuristic.

References

[1] Kosiorek Adam Roman, Sabour Sara , Teh Yee Whye, Geoffrey Hinton (2019). *"Stacked Capsule Autoencoders"*. In: *Advances in Neural Information Processing Systems*. `https://arxiv.org/abs/1906.06818`

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville (2016). *"Deep Learning"*, In: MIT Press. `http://www.deeplearningbook.org`

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle (2007). *"Greedy Layer-wise Training of Deep Networks"*. In: *Advances in Neural Information Processing Systems*. `https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf`

[4] P. Haeusser, J. Plapp, V. Golkov, E. Aljalbout, D. Cremers (2018). *?Associative Deep Clustering: Training a Classification Network with No Labels?*. In: German Conference on Pattern Recognition.

[5] Laurens van der Maaten, Geoffrey Hinton (2008). *"Visualizing Data using t-SNE"*. In: *Journal of Machine Learning Research.* `https://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf`

[6] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, Amos Storkey (2020). *"Meta-Learning in Neural Networks: A Survey"*. In: `https://arxiv.org/pdf/2004.05439`

[7] Sebastian Ruder (2016). *"An overview of gradient descent optimization algorithms"*. In: `https://arxiv.org/pdf/1609.04747`

[8] A.M. Bouayed, K. Atif (2019). *"Un modele de reseaux de neurones efficace pour la classification de donnees simulees pour l?identification du signal du Higgs au collisionneur hadronique du CERN"*. In: [link]

[9] Yann LeCun, Corinna Cortes, CJ Burges (2010) *"MNIST handwritten digit database"*, In: *ATT Labs [Online]. Available:* `http://yann.lecun.com/exdb/mnist`

[10] *"Pytorch Documentation" [Online]. Available:* `https://pytorch.org/docs/stable/index.html`

[11] An Introduction to t-SNE with Python Example.*[Online]. Available:* `https://towardsdatascience.com/an-introduction-to-t-sne-with-python-example-5a3a293108d1`

[12] Tips and Tricks for Multi-Class Classification.*[Online]. Available:* `https://medium.com/@b.terryjack/tips-and-tricks-for-multi-class-classification-c184ae1c8ffc`

A Adam optimiser

This optimiser is a combination of the RMSprop optimiser and mini-batch SGD with momentum optimiser. It has on one hand, a learning rate s_t that varies and adapts to each weight, and on the other hand, a momentum parameter υ_t to accelerate the learning and allow the network to converge faster :

$$\upsilon_t = \beta_1 \upsilon_{t-1} + (1-\beta_1)\frac{\partial E}{\partial W_{i,j}}$$

$$s_t = \beta_2 s_{t-1} + (1-\beta_2)(\frac{\partial E}{\partial W_{i,j}})^2$$

s_t and υ_t are not used directly, they have to be normalised first for the algorithm to give good results from the first iterations. The normalisation is done as follows:

$$\hat{\upsilon_t} = \frac{\upsilon_t}{(1-(\beta_1)^t)}$$

$$\hat{s_t} = \frac{s_t}{(1-(\beta_2)^t)}$$

Then, we update the weights as follows:

$$W_{i,j} = W_{i,j} - \frac{\eta \hat{\upsilon}_t}{\sqrt{\hat{s_t}} + \epsilon}$$

ϵ is of the order of $1e-8$. It is used mainly to avoid division by 0. As for β_1 and β_2 they are decay rates. The recommended values for β_1 and β_2 are 0.9 and 0.999 respectively (7).

B Support Vector Machines

B.1 Definition

A support vector machine (SVM) is a supervised machine learning model, it is used for classification, regression and outliers detection. It helps categorise training data in an optimal way and assigns new examples to its most fitting category. SVMs do so by creating hyper-planes and maximising the margin (distance) that separates the data points of each class the best way possible (12).

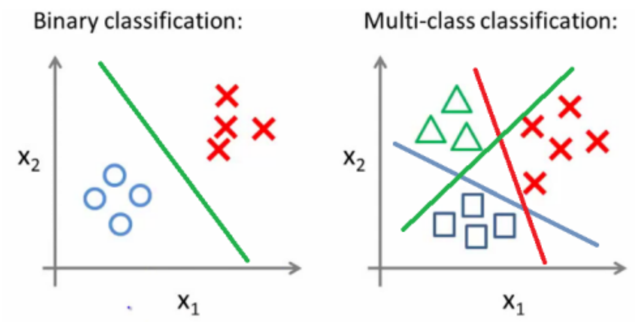

Figure 5: SVM classification (12)

C T-distributed Stochastic Neighbor Embedding

t-Distributed Stochastic Neighbour Embedding (t-SNE) is an unsupervised, non-linear neighbour preserving embedding technique. It was developed by Laurens van der Maatens and Geoffrey Hinton in 2008 (5). t-SNE is primarily used to cast data from a high dimensional space a lower dimensional one while preserving the distance between the data points in the same neighbourhood (11).

Algorithm 4: t-SNE Algorithm

```
for data point $x_i$ in all data points in the higher dimensional space do
    // Calculating the similarity between $x_i$ and all the others data points.
    Calculate the distances between $x_i$ and all the others data points.
    Create a normal distribution $\mathcal{N}(\mu = x_i, \sigma = parameter\ of\ the\ algorithm)$.
    Plot the distances on the normal distribution.
    Mesure the density for all the data points then normalise them so they sum up to 1.
end
Take the data points from the higher dimensional space and randomly put them on the lower
  dimensional space.
while epoch < max_epochs do
    for data point $x_i$ in all data points in the lower dimensional space do
        // Calculating the similarity between $x'_i$ and all the others data points.
        Calculate the distances between $x'_i$ and all the others data points.
        Create a t-distribution centered around $x'_i$.
        Plot the distances on the t-distribution.
        Mesure the density for all the data points then normalise them so they sum up to 1.
        Calculate the difference between the t-distribution and the normal distribution using the
          KL divergence loss.
        Using gradient descent optimise the parameters of the t-distribution so that it matches the
          normal distribution.
    end
    epoch ← epoch + 1
end
```

D Code files

- `AE-interface.py`: python code containing the code for the app interface. It can be run using the command "`python AE-interface.py`"
- `AE_model.py`: python code containing the code for the auto-encoder model used in the app. This file is required to run the app interface.
- `NAS_GA.py:` python code containing the code for the Genetic Algorithm for the Neural Architecture Search. It can be run using the command "`python NAS_GA.py`"
- `final_eval.py`: python code containing the code for running the best architecture found for 50 iterations. It can be run using the command "`python final_eval.py`. It outputs the accuracy, training loss, and test loss on each accuracy. Also, it can be configured to output also the figures of the clusters by setting the `draw` parameter on line 207 to `True`.
- `model.model` **and** `svm.pkl`: are the files containing the weights of the auto-encoder and the SVM that achieve the vest accuracy.

E Tasks affiliations

- Aymene Mohammed Bouayed
  - Coming up with the project and app idea.
  - Writing of the report and the code.
  - Doing 60% of the hyper parameter tuning.
  - Coding the app.
- Hoceine Kennouche
  - Doing 40% of the hyper parameter tuning.