Unsupervised pretraining with Autoencoders

Barletta Valentina and De Benedetti Sara

June 2024

1 Abstract

This project aims to evaluate the performance of the encoder part of a Convolutional Autoencoder, pretrained on the MNIST training set, in classifying digits once a Softmax layer is attached and trained. The results are compared with those obtained using a Convolutional Neural Network (CNN) and an encoder with random weights. Additionally, the Autoencoder is trained on a different dataset (FashionMNIST) to determine if learning more complex features impacts the classification capability of the Encoder with Softmax net.

The reasoning behind wanting to study if unsupervised pretrainig is a possible alternative is that performances of a CNN trained with a low number of images can be poor

As expected, the CNN achieves the best performance even with a small dataset. However, various unsupervised pretraining methods also perform quite well. Among them, the sparse Autoencoder trained on MNIST shows the best results, achieving an accuracy of over 80% for a range of dataset sizes between 160 and 1000.

The study demonstrates that while the CNN remains the superior approach for this task, unsupervised pretraining, particularly with a sparse Autoencoder, provides a valuable alternative, especially when labeled data is limited.

2 Task

The focus of the project is to study the behavior of unsupervised pretraining on a small dataset. The objective is to determine whether a more meaningful data representation, obtained through an Autoencoder, could compensate for the lack of available labeled data. An Autoencoder is used to learn a compressed representation of the data and then the encoder is transferred to a network with a Softmax layer for classification. The supervised task involves classifying digits in the MNIST dataset. The goal is to compare the performance of this pretrained network with that of a Convolutional Neural Network trained from scratch on the same dataset. In that case, the advantage of pretraining is that one can pretrain once on a huge unlabeled set, learn a good representation and then use this representation or fine-tune it for a supervised task for which the training set contains substantially fewer examples.

3 Unsupervised pretraining

Unsupervised pretraining is a common process in machine learning based on the idea that discovering a new representation in the unsupervised phase can improve the performance and efficiency of subsequent supervised learning.

It combines two key concepts: selecting initial parameters for a deep neural network can significantly regularize the model, and understanding the input distribution can enhance the learning of the mapping from inputs to outputs.

The first idea suggests that pretraining places the model in a state that would otherwise be inaccessible. To avoid the complexities of understanding how supervised learning preserves information from unsupervised learning, one can simply freeze the feature extractor's parameters and use supervised learning only to add a classifier on top of the learned features.

The other idea is based on the concept that some features that are useful for the unsupervised task may also be useful for the supervised learning task. However, how and why this work is not yet understood so it is not always possible to predict which tasks will benefit from unsupervised learning. Many aspects of this approach heavily depend on the specific models used.

Since unsupervised pretraining acts as a regularizer, it is especially helpful when there are few labeled examples but a large amount of unlabeled data. In such cases, the hypothesis is that pretraining encourages the learning algorithm to discover important features that lead to better representations of the input.

For study this problem, we have implemented an Autoencoder which learns effective representations of unlabeled input data. Then we freezed all the coder layers' weights and we replaced the decoder by a fully connected layer whit a Softmax at the end to classify the digits of MNIST dataset.

The Autoencoder was trained on 59000 examples from MNIST and FashionMNIST dataset and 1000 samples were used to form a validation set, just to qualitatively study if the Autoencoder learns well.

4 Methods

In the following sections, each supervised net was trained for 100 epochs with 50 different sizes of datasets increasing at a constant rate from 160 samples to 8000. The test size was fixed at 350 images and the batch sizes of test and training sets were respectively 32 and 16. Each training was repeated three times in order to obtain an average accuracy.

4.1 Convolutional Autoencoder

Since we needed to solve an image classification task, we chose to implement a Convolutional Autoencoder.

Like a traditional Autoencoder, a Convolutional Autoencoder has two main components: an encoder and a decoder. The encoder processes the input image with convolutional layers and pooling operations to produce a lower-dimensional feature representation of the image. The decoder then takes this lower-dimensional feature representation and upsamples it back to the original input image size using deconvolutional layers. The network's final output is a reconstructed image that is as close as possible to the original input image.

The architecture of our Autoencoder is:

- **Encoder**
  - *Convolutional layer*: Input ($28 \times 28 \times 1$) is convolved with 8 filters of size 3×3, resulting in output ($28 \times 28 \times 8$). ReLU was used as the activation function;
  - *Max-pooling layer*: Reduces the spatial dimensions using a 2×2 kernel and a stride of 2, resulting in output ($14 \times 14 \times 8$);
  - *Convolutional layer*: 4 filters of size 3×3 are applied resulting in output ($14 \times 14 \times 4$). Again ReLU was used as the activation function;
  - *Max-pooling layer*: Further reduces the dimensions to produce a lower-dimensional feature representation ($7 \times 7 \times 4$).
- **Decoder**
  - *Convolutional layer*: Encoded features ($7 \times 7 \times 4$) are processed with 4 filters of size 3×3, resulting in output ($7 \times 7 \times 4$);
  - *Upsample layer*: increases the spatial dimensions using nearest-neighbor interpolation, resulting in output ($14 \times 14 \times 4$);
  - *Convolutional layer*: 8 filters of size 3×3 are applied resulting in output ($14 \times 14 \times 8$);
  - *Upsampling layer*: Further increases the spatial dimensions, resulting in output ($28 \times 28 \times 8$);
  - *Convolutional layer*: Single filter of size 3×3 reconstructs the original image, resulting in output ($28 \times 28 \times 1$).

The network is trained to minimize the difference between the original input image and the reconstructed output image and for this project we used the *mean squared error* (MSE) as the loss function:

$$J(w) = \frac{1}{2m}||y - \hat{y}||^2 = \frac{1}{2m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

where y is the real image and $\hat{y}$ is the reconstructed image by the Autoencoder. m is the dataset size.

We chose, also, to use the *Adam optimizer* with a learning rate of 0.001.

4.1.1 Sparse Autoencoder

We investigated also the performance of a Sparse Autoencoder which is simply an Autoencoder whose training criterion involves a sparsity penalty. The *L1 regularization*, which tends to shrink the penalty coefficient to zero, was chosen in this work. The Loss function is:

$$J(w) = \frac{1}{2m}||y - \hat{y}||^2 + \lambda||w||$$

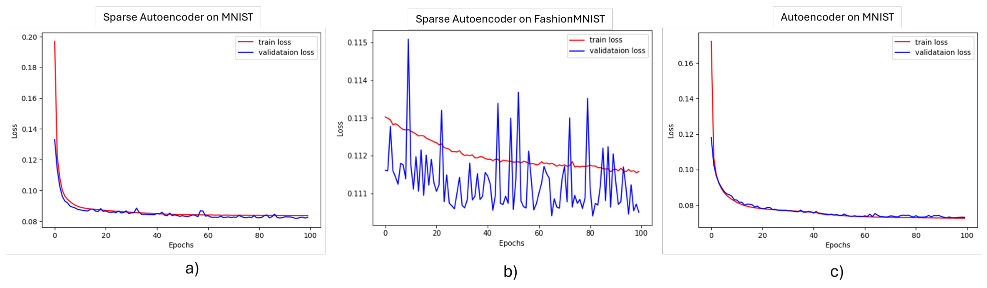

Figure 1: Loss curves in function of epochs for a) Sparse Autoencoder trained on MNIST, b) Sparse Autoencoder trained on FashionMNIST and c) Autoencoder trained on MNIST

whit $\lambda = 0.001$ as the regularization parameter.

In Fig.1 are shown the behave of the loss functions for the different kinds of Autoencodere which were implemented.

4.2 Encoder with Softmax

Once we have trained the Autoencoder and freezed the weights, we removed the decoder and attached a fully conneted layer with Softmax at the end.

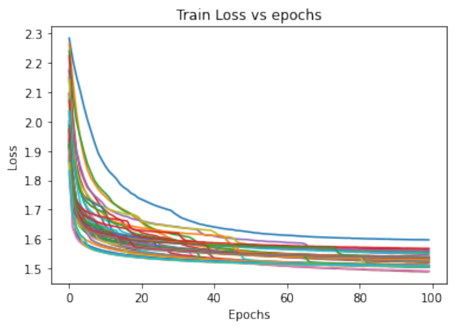

Figure 2: How the different Train Loss for the different dataset size decrease during the training phase. The unsupervised pretraining was made by the Autoencoder trained on MNIST.

During the training phase we have noticed that the learning did not work properly and this was caused by the *Exploding Gradients*. This is a common issue during the train of neural network and occurs when the gradients of the network's loss with respect to the parameters (weights) become excessively large.

To solve this problem we used the *Gradient Clipping technique*. This strategy involves setting a threshold value, and if the gradient exceeds this threshold, it is scaled down to keep it within a manageable range. This prevents any single update from being too large.

In addition to this, the optimizer we chose is Adam with learning rate 0.001. We also have tried to use Stochastic Gradient Descent (SGD) with momentum, but got worse performances.

4.3 Random encoder

This section wants to prove that the Autoencoder, despite being a tool that does unsupervised learning, has actually grasped some hidden features in the MNIST dataset.

If the classification of the images were random the accuracy would be around 10% since there are ten classes. However, in some cases, the weights initialization can be quite impactful and untrained neural networks can achieve unexpected results.

In order to show the Autoencoder's ability to learn, the results obtained in section 4.2 have been compared with the ones derived by initializing the encoder part of the Autoencoder with random weights.

The structure of encoder used to do unsupervised pretraining is preserved and the weights are initialized the same way PyTorch, the Python library used, would have: with a uniform distribution in the interval $\left[-\sqrt{6/n}, \sqrt{6/n}\right]$ where n is the number of input from the previous layer. This initialization is a heuristic thought out to prevent exploding gradients. Finally a Softmax is attached to the end of the encoder, which weights are frozen, and it's trained.

The loss is calculated with *cross-entropy* (eq. (1)) and its minima are searched with Adam algorithm with learning rate $\eta = 10^{-3}$.

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{C} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{1}$$

In the *cross-entropy* y is the label and $\hat{y}$ is the predicted label, m is the dataset size and C is the number of classes ($C = 10$).

4.4 Convolutional Neural Network

Convolutional Neural Networks are known to be a very good instrument to classify images. However, as every supervised learning algorithm, CNNs need to be trained on a large number of labeled data which often don't exist or are expensive to obtain. The loss used to train the algorithm is *cross-entropy* (equation 1) which minima are searched using *Stochastic Gradient Descent* with learning rate $\eta = 0.1$.

The architecture of the neural network is the following:

- *Convolutional layer*: Input $(28 \times 28 \times 1)$ is convolved with 5 filters of size 3×3 with padding and stride equal to 1. ReLu was used as an activation function.
- *Max-pooling layer*: reduces the spatial dimensionality using a 2×2 kernel and stride 2, resulting in output $(14 \times 14 \times 5)$.
- *Convolutional layer*: 5 filters of size 3×3 with padding and stride equal to 1 are applied resulting in output $(7 \times 7 \times 5)$. Again ReLu was used as an activation function.
- The output of the previous layer is flattened to prepare it for the fully connected layer.
- *Dropout layer*: dropout probability of 0.2 is applied to prevent overfitting.
- *Linear layer*
- *Softmax*: It produces a probability distribution over the 10 classes.

As one might expect the CNN presents excellent results especially once the number of data used for training increases. In picture 3 are represented the accuracies for training and test sets for different training dataset sizes. The gap between the two curves tents to decrease with the increase size of the training set, which reminds that another problem of training CNN with small datasets is that it can do overfitting, despite an attempt at regularization is done by introducing dropout.

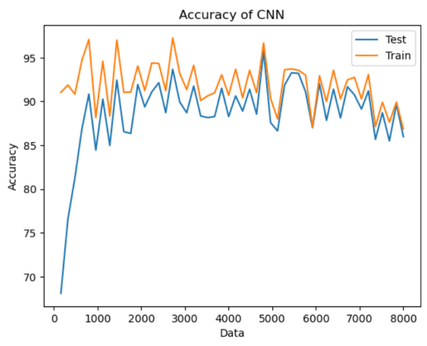

Figure 3: Accuracy of test and train set in function of dataset size for CNN

5 Results

In this section, we discuss the main result of our study. Fig.4 shows the test accuracy curves for different training sizes obtained using three different kinds of unsupervised pretraining (Autoencoder trained with the MNIST dataset, sparse Autoencoder trained with MNIST, and sparse Autoencoder trained with FashionMNIST), the performance of unsupervised pretraining by freezing random weights, and the accuracy curve of a simple CNN.

As one might expect, the CNN's performance is the best even when only a small dataset is available. However, the three different kinds of unsupervised pretraining also perform quite well. If we have to choose a method, the

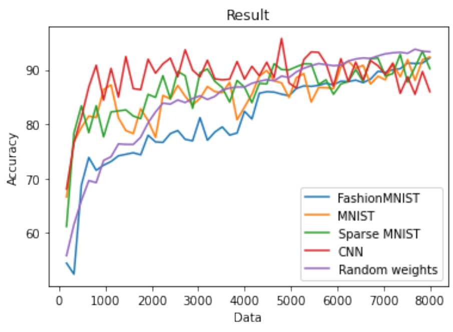

Figure 4: How the different size of the training dataset affect the test accuracy

sparse Autoencoder trained on the MNIST dataset is undoubtedly the best choice. It seems to work decently with small datasets (range from 160 to 1000), reaching an accuracy of 80%.

It is interesting to notice that pretraining on FashionMNIST also works quite well, even though the representation learned by the Autoencoder is associated with clothes and not digits. This indicates that more complex inputs can lead to rich representations even if the final task, in this project classifying digits, is different.

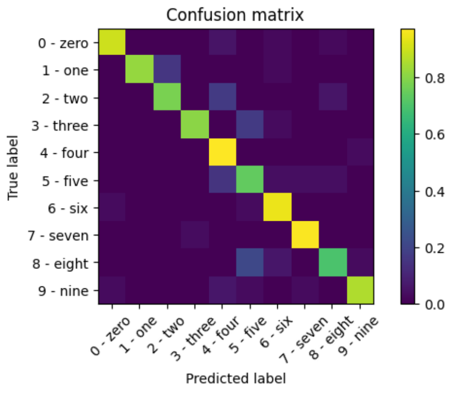

Figure 5: Confusion Matrix of the unsupervised pretrained Sparse Autoencoder on a subset of the MNIST dataset, consisting of 480 samples. The model achieved an accuracy of approximately 80% on this dataset. The diagonal elements show the number of correctly classified samples for each digit, while the off-diagonal elements indicate misclassifications.

The "Random"Autoencoder behaves surprisingly well. One might think that freezing random weights would be the worst method for unsupervised pretraining, but random initialization allows the initial position to be set in regions near a minimum of the cost function, enabling the algorithm to learn effectively. Sometimes, however, the random initialization sets the initial weights far from minima or in flat regions of the loss function. The problem is that we cannot predict when this random version might be beneficial.

The CNN accuracy fluctuates more compared to the other curves. This can be attributed to the fact that during each run, the weights are initialized randomly. In contrast, the curves derived from unsupervised pretraining have fixed parameters from the Autoencoder, with only a small set of parameters being initialized randomly.

6 Conclusions

In this work, we explored different kinds of unsupervised pretraining to understand if there are any benefits when only a small dataset is available. We compared these three methods against a standard CNN and pretraining by freezing the weights of an autoencoder randomly.

The results indicate that for this specific problem, the classification of MNIST digits, the CNN strategy is the winning one. Unsupervised pretraining performs well, especially with the sparse Autoencoder trained on MNIST digits, but the CNN remains the best approach.

This can be explained by considering the main problems of unsupervised pretraining. As we discussed in the previous section, pretraining acts as a regularizer but is not like lasso or ridge regularization. There isn't a hyperparameter that can be chosen to express the strength of regularization. Another disadvantage is having two separate training phases, each with its own hyperparameters. The performance of the second phase usually cannot be predicted in the first one, leading to a long delay between setting the hyperparameters for the first phase and being able to update them using feedback from the second phase.

It must be kept in mind that the task of this project is very simple. The work can be extended by studying if CNN is still the winning choice once a more complex datasets is used.

In future work, we suggest using validation set error in the supervised phase to select the hyperparameters of the pretraining phase. It might be a good idea to use early stopping in the pretraining phase, which is not ideal but computationally much cheaper.

As we have shown in the last section, the unsupervised pretraining done with the FashionMNIST dataset seems to perform quite well. Therefore, further investigation in this direction, pretraining with more complex datasets, could be interesting.

7 Code

The code of this work can be found on GitHub at this link: https://github.com/Awalilly/Unsupervised-pre-training/tree/main

8 Appendix

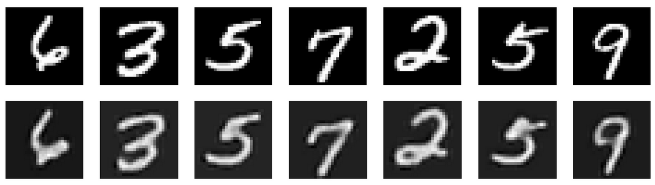

Figure 6: MNIST images reconstructed with sparse Autoencoder

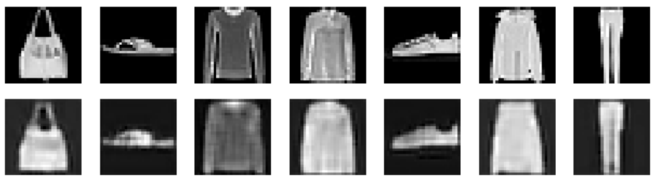

Figure 7: FashionMNIST images reconstructed with sparse Autoencoder

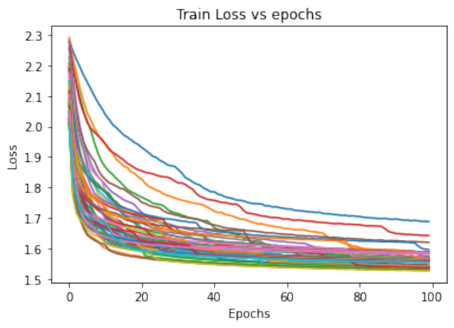

Figure 8: Loss in function of epochs for all datasets used. The unsupervised pretraining was done on MNIST with sparse Autoencoder

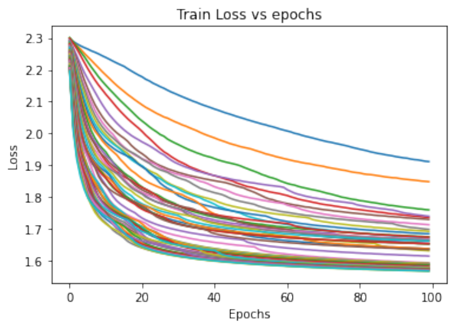

Figure 9: Loss in function of epochs for all datasets used. The unsupervised pretraining was done on FashionM-NIST

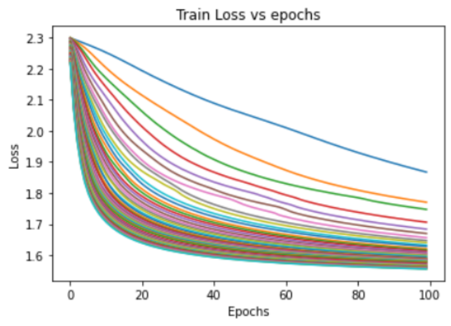

Figure 10: Loss in function of epochs for all datasets used. The encoder was initialized with random weights

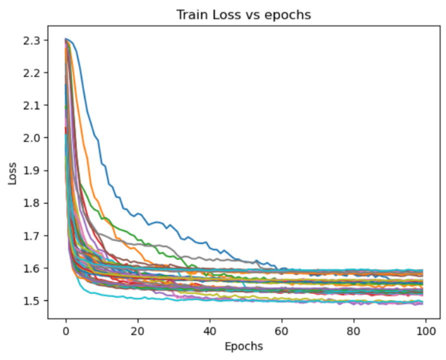

Figure 11: CNN loss in function of epochs for all datasets used