Analysis of a DBN model

Giorgia Rinaldi (2092226)

20 June 2023

1. Introduction

Visual concept learning is being explored using the Deep Belief Network (DBN), a fully connected deep neural network consisting of multiple layers of Restricted Boltzmann Machines (RBMs). This architecture enables the creation of a hierarchical generative model, capable of developing increasingly abstract and complex representations of features as the network goes deeper. DBNs are powerful tools employed in generative unsupervised learning, aiming to learn internal representations of the environment by observing available evidence and leveraging its inherent structure to replicate observed data. Moreover, the DBN architecture enables the simulation of the hierarchical structure found in the human brain and facilitates the generation of plausible models of human cognition.

2. Dataset

After the preprocessing phase (define a function to download DBN and RBM, import libraries, check the availability of the GPU), the dataset EMNIST has been imported. EMNIST is a dataset of handwritten characters derived from NIST Special Database 19 in which every character is converted to a 28x28 pixel image. There are different splits provided in EMNIST dataset (EMNIST Letters, EMNIST ByClass, EMNIST ByMerge, EMNIST Balanced, EMNIST Digits, EMNIST MNIST). EMNIST Letters is the split chosen for the analysis. It contains 145600 handwritten letters characters belonging to 26 different classes. The first step of the analysis involves downloading data to train models and training set necessary to evaluate data. Subsequently, in order to streamline computations, all elements undergo transformations. The images are converted into PyTorch tensors; then a normalization step takes place, ensuring that pixel values remain within the range of [0,1]. This is achieved by dividing each pixel by 255. These transformations simplify the overall computational process. Then, data and targets both of training and test set are moved to GPU.

```
[ ] %%capture
    emnist_train = tv.datasets.EMNIST('https://github.com/aurelienduarte/emnist', split = "letters", train = True, download = True,
                                      transform=tv.transforms.Compose(
                                                  [tv.transforms.ToTensor()]
                                            ))

    emnist_test = tv.datasets.EMNIST('https://github.com/aurelienduarte/emnist', split = "letters",
                                     train = False,
                                     download = True,
                                     transform=tv.transforms.Compose(
                                            [tv.transforms.ToTensor()]
                                     ))

    emnist_train.data = (emnist_train.data.type(torch.FloatTensor)/255)
    emnist_test.data = (emnist_test.data.type(torch.FloatTensor)/255)


    emnist_train.data = emnist_train.data.to(device)
    emnist_test.data = emnist_test.data.to(device)
    emnist_train.targets = emnist_train.targets.to(device)
    emnist_test.targets = emnist_test.targets.to(device)
```

3. DBN model

In the following study, the DBN model comprises a visible layer with 784 neurons, representing a 28×28-pixel image, and three RBM layers, consisting of 350, 450 and 600 hidden neurons. The processing flow follows a bottom-up approach, where the input evidence presented to the visible neurons is processed by the hidden neurons to construct credible explanations of phenomena. These hypotheses then serve as input for the next RBM layer, extracting higher-order levels of internal representation and continually building upon the existing hypotheses. Additionally, a top-down processing path exploits the learned statistical structure to generate new data, aiming to reconstruct the most probable distribution of instances based on the observed evidence. The core idea behind the RBMs algorithm lies in updating the connection weights to enhance the accuracy of reconstruction. Model parameters are adjusted proportionally to the distortion of data produced by the model in relation to the training patterns. This distortion is computed as the difference between the data-driven phase (positive phase) and the model-driven phase (negative phase). The alternating iterative phases continue until the Markov Chain process converges. The learning algorithm employed by each DBN core is contrastive divergence k, which allows the algorithm to terminate after k iterations.
The values in the script below are:

- 28x28 is the number of visible units;
- there are 3 RBM hidden layers with respectively 350, 450 and 600 hidden neurons;
- K, a hyperparameter that indicates the number of iterations that the algorithm has to do befor stopping. K = 1 means that it is necessary only a recostruction of data to reach a good model;
- Learning rate determines the step size of gradient descent. If the learning rate is set too high, the convergence could be too fast and it could overshoot the optimum. On the other hand, if the learning rate chosen is a too low value, the process could be too slow. Learning rate decay is set to FALSE: it will not decrease;
- Momentum range 0.5 – 0.95: momentum adds a fraction of the previous weight update to the new one. If the gradient is in the same direction, the size of step taken towards the minimum will be increased. Momentum will flatten the variation when the gradient changes direction;
- Weight decay is a regularization tecnique used to force weights to spontaneously decay;
- Xavier Initialization won't be used;
- k is constant over the whole training;
- if available, a GPU is used.

```
[ ]  dbn_emnist = DBN(visible_units=28*28,
                      hidden_units=[350, 450, 600],
                      k=1,
                      learning_rate=0.1,
                      learning_rate_decay=False,
                      initial_momentum=0.5,
                      final_momentum=0.95,
                      weight_decay=0.0001,
                      xavier_init=False,
                      increase_to_cd_k=False,
                      use_gpu=torch.cuda.is_available())
```

During training, the network divides the data into batches at each epoch. Every 10 epochs, the reconstruction error is computed and displayed for each layer. This error quantifies the disparity between the original

data and its reconstructed form. The reconstruction error diminishes as the number of epochs progresses, indicating improvement in the model's performance. The variables "mean_grad" and "std_grad" represent the magnitude and variability of the gradient, respectively. It is important for these values to strike a balance—neither excessively large nor overly small. This is because an optimal point must be reached, and while progress is necessary, an excessively substantial improvement might lead to overshooting the optimum. Therefore, maintaining an appropriate magnitude and variation in the gradient is crucial for achieving optimal results.
Here the output of layer 1, as an instance:

```
[ ]  num_epochs = 50
     batch_size = 125

     dbn_emnist.train_static(emnist_train.data, emnist_train.targets, num_epochs, batch_size)
```

```
--------------------
Training RBM layer 1
|Epoch |avg_rec_err |std_rec_err  |mean_grad |std_grad  |
|10    |1.3706      |0.0465       |316.4798  |5.0149    |
|20    |1.3167      |0.0441       |302.7906  |5.1818    |
|30    |1.3049      |0.0430       |299.9147  |4.6386    |
|40    |1.2973      |0.0411       |296.8663  |4.9892    |
|50    |1.2932      |0.0419       |295.5092  |4.3176    |
--------------------
```

4. Visualization of receptive fields

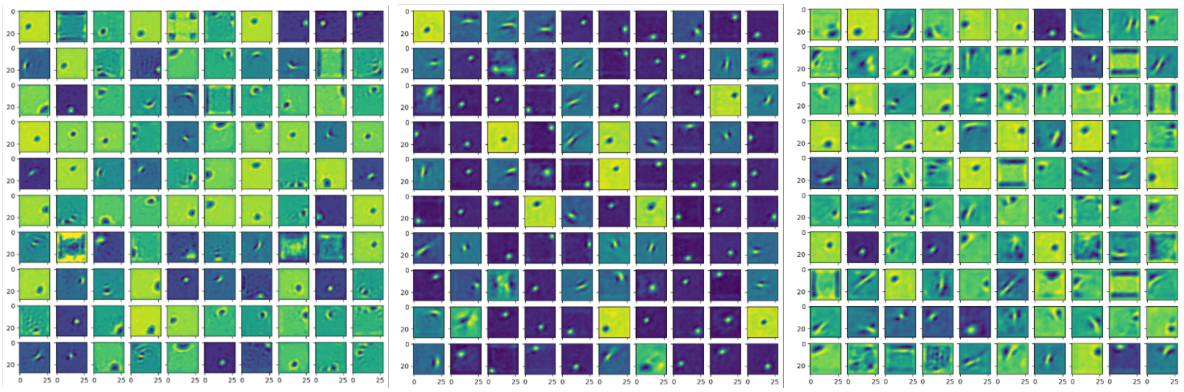
Upon completing the network training, the next step is to visualize the learned weights. This visualization allows to gain insights into the patterns and features that trigger specific units within the network. To achieve this, we will plot a selection of weights in an image with the same dimensions as the EMNIST images used during training. In order to reduce noise in the plots, a threshold to the weights is applied. Additionally, we scale all weight values uniformly to facilitate easier comparisons between different filters. We will represent the first 100 filters learned in each layer, providing a glimpse into the specific regions of an image or patterns that these individual nodes are capable of recognizing. Since the dimensions of the second and third layers differ from the original 28x28 size, we need to project each layer onto a 28x28 dimensional space. This projection ensures that the visualized weights align properly with the original image dimensions, allowing for a more meaningful interpretation of the learned features.

```
[ ]  def get_weights(dbn, layer):
       return dbn.rbm_layers[layer].W.cpu().numpy()

     def apply_threshold(weights, threshold=0):
       return weights * (abs(weights) > threshold)
```

```
[ ]  def plot_layer_receptive_fields(weights):
       num_subplots = 100
       n_rows_cols = int(math.sqrt(num_subplots))
       fig, axes = plt.subplots(n_rows_cols, n_rows_cols, sharex=True, sharey=True, figsize=(10, 10))
       for i in range(num_subplots):
         row = i % n_rows_cols
         col = i // n_rows_cols
         axes[row, col].imshow(weights[i,:].reshape((28,28)))
```

```
[ ]  def apply_min_max_scaler(learned_weights):
       original_shape = learned_weights.shape
       min_max_scaler = sklearn.preprocessing.MinMaxScaler()
       min_max_scaled_learned_weights = min_max_scaler.fit_transform(learned_weights.ravel().reshape(-1,1))
       min_max_scaled_learned_weights = min_max_scaled_learned_weights.reshape(original_shape)
       return min_max_scaled_learned_weights
```

The result is a matrix composed by 100 receptive fields. Weights for layer 2 and layer 3 are obtained as dot product among weight matrices and new level matrix.



5. Clustering internal representations

Next, we proceed to calculate the centroids of the learned representations for each class and assess their proximity to one another using a standard hierarchical clustering algorithm. The implemented Deep Belief Network (DBN) consists of multiple Restricted Boltzmann Machine (RBM) objects internally. Hence, to compute the hidden representations, we must utilize the weights of each RBM layer within the DBN. Specifically, we calculate the representations of the second layer by utilizing the representations from the previous layer, and continue this process for subsequent layers. This stepwise computation ensures that each layer's hidden representation builds upon the information learned in the preceding layer. By leveraging these representations, we can analyze the proximity between different classes and gain insights into their relationships.

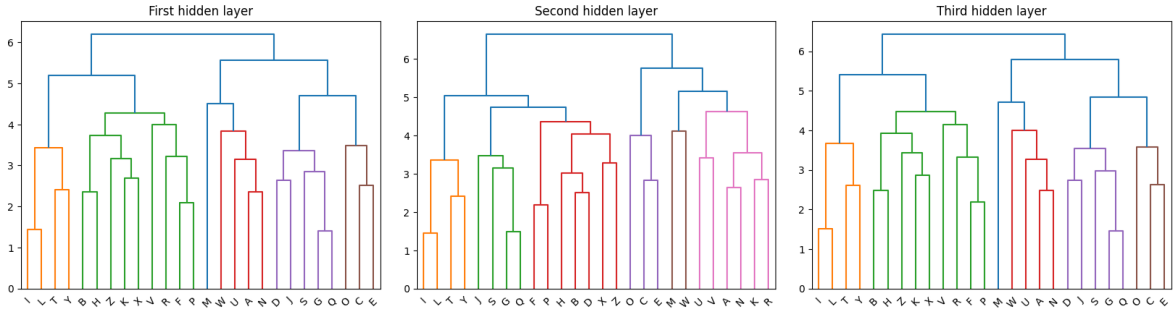
```
[ ]  def get_kth_layer_repr(input, k, device):
       flattened_input = input.view((input.shape[0], -1)).type(torch.FloatTensor).to(device)
       hidden_repr, __ = dbn_emnist.rbm_layers[k].to_hidden(flattened_input)
       return hidden_repr

[ ]  hidden_repr_layer_1 = get_kth_layer_repr(emnist_train.data, 0, device)
     hidden_repr_layer_2 = get_kth_layer_repr(hidden_repr_layer_1, 1, device)
     hidden_repr_layer_3 = get_kth_layer_repr(hidden_repr_layer_2, 2, device)
```

Now the aim is to calculate the average hidden representation for each class within the EMNIST dataset. This process involves constructing a matrix that contains all the necessary centroids. The resulting matrix will have dimensions of 26xD, where D represents the number of nodes in that particular layer. Then we execute the clustering algorithm and visualize its output in a dendrogram plot.

```
[ ]  def get_mask(label):
       labels = emnist_train.targets.cpu().numpy()
       return labels == label

     def get_label_to_mean_hidd_repr(hidden_representation):
       hidden_representation_np = hidden_representation.cpu().numpy()
       return {
         label: hidden_representation_np[get_mask(label)].mean(axis=0)
         for label in range(1, 27)
       }

     def get_hidden_reprs_matrix(hidden_representation):
       label_to_mean_hidd_repr = get_label_to_mean_hidd_repr(hidden_representation)
       return np.concatenate(
         [np.expand_dims(label_to_mean_hidd_repr[label], axis=0)
         for label in range(1, 27)])

     def plot_dendrogram(mean_repr_matrix, title=""):
       fig, ax = plt.subplots()
       linkage = cluster.hierarchy.linkage(mean_repr_matrix, method="complete")
       dendrogram = cluster.hierarchy.dendrogram(linkage, labels = [emnist_dict[i+1] for i in range(26)])
       ax.set_title(title)
```

The dendrogram provides a comprehensive representation of how the data points are grouped into clusters at different levels. It is highly unlikely for one class to be mistaken for another if they are located far apart in the dendrogram. We generate dendrograms for each of the three layers, showcasing the unique clusterings

created by each layer based on the considered features. While the differences may not be immediately apparent due to the layout, a closer examination of the branches reveals distinctions. For instance, similar letters such as G and Q (or I and L, or F and P) consistently appear in close proximity to each other. However, there may be slight variations in the branching patterns around letters like W and M.



6. Linear read-out: linear classifier model

A linear readout of the computed representations from each layer of the Deep Belief Network (DBN) is now performed. These representations are utilized for classifying the original images using a simple linear classifier, allowing us to evaluate the information content within each hidden representation. To accomplish this, a class is defined that includes a function responsible for linearly transforming the input tensor using a single fully connected layer. The linear classifier is constructed to take the previously computed hidden representations from each layer as input, with the labels of the EMNIST dataset serving as the targets. Stochastic gradient descent is employed as the optimizer for this model, with calculations performed on a per-batch basis. Each linear layer is trained using this approach, and it becomes apparent that the loss (e.g., cross-entropy) decreases over epochs for each of the three layers. If the decision is made to train the model for more epochs, the final loss would be smaller. However, to prevent overfitting and considering that the rate of improvement in loss decreases over time, it was decided to halt training after 1500 epochs. This decision balances the desire for better performance with the need to avoid overfitting the model to the training data.

```
class LinearModel(torch.nn.Module):
  def __init__(self, rbm_layer_size):
    super().__init__()
    self.linear = torch.nn.Linear(rbm_layer_size, 27)

  def forward(self, x):
    return self.linear(x)
```

Linear classifier training phase on EMNIST letters:

```
[ ]  def train(model, input, epochs=1500):
       print_stride = 100 if epochs >= 1500 else 10
       optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
       loss_fn = torch.nn.CrossEntropyLoss()
       targets = emnist_train.targets

       for epoch in range(epochs):
         optimizer.zero_grad()
         predictions = model(input)
         loss = loss_fn(predictions, targets)
         loss.backward()
         optimizer.step()

         if epoch % 100 == 0:
           print("epoch: {:3d}/{} | loss: {:.4f}".format(epoch + 1, epochs, loss))
```

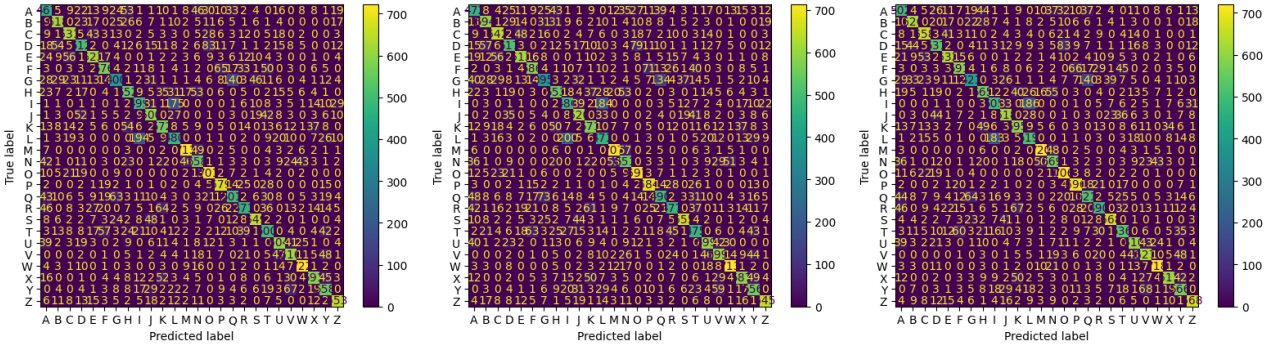
The result is that the loss decreases as epochs increase: loss 1 goes from 3,3126 to 1,0101; loss 2 decreases from 3,3190 to 1,0125 and loss 3 passes from 3,3369 to 0,9336.

Linear classifier testing phase: Next, the test set: the hidden representations of the three layers are computed for this test set. These representations capture the learned features and patterns within the DBN. Subsequently, predictions are made using the previously trained linear model. To evaluate the performance of the linear model and determine if it effectively classifies the test set, an accuracy score is computed. Accuracy represents the number of correct predictions divided by the total number of predictions. It is used to understand how well neurons encode hidden data. In order to compute the accuracy of the model it is necessary to obtain the hidden representation of RBM hidden layers, such that they can be used for predictions and accuracy estimation. The obtained accuracy scores for all three layers are approximately 0.7. This value indicates that a significant proportion of the predictions made by the linear model were correct. As it can be seen, the last layer gives a better accuracy. It means that increasing the complexity of the model allows the model to achieve better performance.

```
[ ]  def compute_accuracy(predictions_test, targets):
       predictions_indices = predictions_test.max(axis=1).indices
       accuracy = (predictions_indices == targets).sum() / len(targets)
       return accuracy.item()

[ ]  compute_accuracy(predictions_test1, emnist_test.targets)

     0.7233653664588928

[ ]  compute_accuracy(predictions_test2, emnist_test.targets)

     0.7189903855323792

[ ]  compute_accuracy(predictions_test3, emnist_test.targets)

     0.7439903616905212
```

7. Confusion matrices

Confusion matrices allows to visualize the performance of the model. Diagonal elements represent the sum of well-predicted labels related to a specific class. The third confusion matrix has the highest well-predicted amount of elements. According with the results reached computing accuracies, the confusion matrix of the third layer shows that the third layer is more accurate and it can recognize a higher amount of instances than the first and second layers.



8. Feed Forward

To compare the performance of a non-linear, end-to-end trained model and a simple linear classifier that uses unsupervised learned representations, a feed-forward neural network with the same structure as the Deep Belief Network (DBN) was trained. The objective was to solve a classification task. The feed-forward network consists of three layers. The first layer takes an input size of 784, which corresponds to the number of pixels in an EMNIST image. The training procedure and the number of epochs used were kept consistent, and the number of nodes in each hidden layer is 350, 450 and 600 as before (in order to have a fair comparison between the two models and the three RBMs). The output layer is composed by 27 neurons. To prevent the issue of gradient vanishing, each hidden unit in the network employs the rectified linear unit (ReLU) activation function. ReLU is a non-linear function that operates on the global input of each hidden unit. It helps mitigate the problem of vanishing gradients that can occur during training. By using ReLU, the network aims to preserve the gradient flow and enable effective learning and optimization.

```
[ ]  class Feedforward(torch.nn.Module):
       def __init__(self, first_hidden_layer_size, second_hidden_layer_size, third_hidden_layer_size):
         super().__init__()
         self.first_hidden = torch.nn.Linear(784, first_hidden_layer_size)
         self.second_hidden = torch.nn.Linear(first_hidden_layer_size, second_hidden_layer_size)
         self.third_hidden = torch.nn.Linear(second_hidden_layer_size, third_hidden_layer_size)
         self.output = torch.nn.Linear(third_hidden_layer_size, 27)

       def forward(self, input):
         relu = torch.nn.ReLU()
         first_hidden_repr = relu(self.first_hidden(input))
         second_hidden_repr = relu(self.second_hidden(first_hidden_repr))
         third_hidden_repr = relu(self.third_hidden(second_hidden_repr))
         output = self.output(third_hidden_repr)
         return output

[ ]  ffnn = Feedforward(350, 450, 600).to(device)
```

As it can be seen in the .ipynb file, loss decreases from 3.2952 to 0,8384 as epochs increase and accuracy is a bit higher than the accuracy level of the third hidden layer classifier (but it is a very small difference: +0,0227).

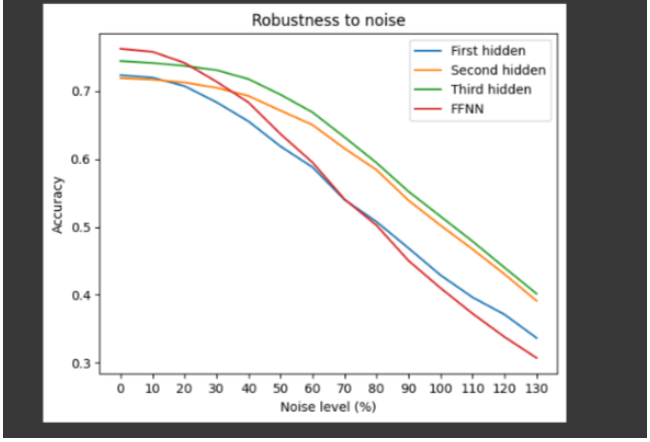
9. Psychometric curves: robustness to noise

It has been analyzed the level of the model accuracy in prediction of distorted data. First of all, it has been injected a noise on the test sample. The noise has been firstly introducted to each instance randomly following a gaussian distribution. The computed value is multiplied by the noise level, which serves as an indicator of the variance of the Gaussian distribution. The expected result is that accuracy should decrease as noise increases.

```
[ ]  def inject_noise(emnist_data, noise_level):
       random_gaussian_tensor = torch.randn(emnist_data.shape, device = device)*noise_level
       return emnist_data + random_gaussian_tensor
```

Plotting psychometric curves a comparison can be done.
It can be seen that DNN is the worst model in terms of robustness. It still is the best model until noise level gets greater or equal to 0.2. Second and third hidden layers show better robustness, in fact with a noise level equal to 1.3, their accuracy value is around 40%. As shown in the chart, the third hidden layer is more accurate than the second hidden layer for higher level of noise. The reason could be the risk of overfitting of the third layer.

```
[ ] noise_levels = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3]
    plot_noise_robustness_curves(noise_levels)
```



10. Adversarial attacks

Adversarial attacks aim to manipulate the input in a way that the model becomes unable to correctly classify it, resulting in an increased loss for that specific input. The loss function typically takes into account the input, the model's parameters, and the model's outputs. During model training, the weights are adjusted based on the gradient of the loss function, moving in the opposite direction to decrease the loss. However, when creating an adversarial sample, instead of modifying the model's weights, it is altered the input itself. In this case, it is followed the same direction as the gradient, intentionally increasing the loss function. The objective of such modifications is to deceive the model and exploit its vulnerabilities, causing misclassification or inducing false outputs. By understanding the gradient's direction, adversaries can manipulate inputs strategically to trigger specific responses from the model, leading to compromised performance. The first step is to modify DBN model to simulate back propagation and to show how the DBN would react to an adversarial attack.

```
[ ] def fgsm_attack(image, epsilon, data_grad):
        sign_data_grad = data_grad.sign()
        perturbed_image = image + epsilon*sign_data_grad
        perturbed_image = torch.clamp(perturbed_image, 0, 1)
        return perturbed_image
```

A read-out classifier is introduced, utilizing the previously defined DBN classifier. The pre-trained linear classifiers within the DBN are employed to generate the final prediction by leveraging the DBN as a feature extractor. In this specific scenario, the third layer of the DBN functions as the read-out layer during the inference process. With the assistance of backpropagation, this model can be conveniently fine-tuned to cater to specific requirements or optimize performance based on desired outcomes.
Next, the robustness of each of the three models is evaluated by subjecting them to increasingly stronger attacks.
The FFN consistently performs worse than the DBN in both reconstructed and unaltered states. These results clearly demonstrate the DBN's superior accuracy in tackling such challenges.
The chart below shows that even few value of the strength of avversarial attack could create a high loss of model accuracy. While the DNN accuracy decreases for small values of epsilon, DBN keeps a quite good level of accuracy (around 0.4) for epsilon values smaller than 0.15, than the accuracy loss for the DBN bacomes higher. DBN top-down iterated more times could reduce the presence of the noise. So, DBN model can remove injected attacks, infact with only 1 step the performance increases.

```
[47] class DBNWithReadOut(torch.nn.Module):
        def __init__(self, dbn_emnist, readouts, readout_level=0):
            super().__init__()
            self.readouts = readouts
            self.dbn_emnist = dbn_emnist
            self._require_grad()
            self.readout_level = readout_level

        def _require_grad(self):
          for rbm in self.dbn_emnist.rbm_layers:
            rbm.W.requires_grad_()
            rbm.h_bias.requires_grad_()

        def forward(self, image):
          p_v = image
          hidden_states = []
          for rbm in self.dbn_emnist.rbm_layers:
            p_v = p_v.view((p_v.shape[0], -1))
            p_v, v = rbm(p_v)
            hidden_states.append(p_v)
          return self.readouts[self.readout_level].forward(hidden_states[self.readout_level])
```

```
[55] epsilon_values = [0, 0.05, 0.10, 0.15, 0.20, 0.25]
```

In comparison to an FFN (Feed-Forward Neural Network), the DBN consistently outperformed in all the tested scenarios. This emphasizes the DBN's exceptional performance when attempting to simulate human perception. Despite applying small amounts of noise and subjecting the images to attacks, these challenges posed no significant obstacle for a human observer, who would effortlessly distinguish them. However, for a neural network, such tasks are not as trivial. The experiments demonstrated that the DBN stands out as a promising choice when aiming to develop an algorithm that can "simulate" our perception to a certain extent.