Deep Learning - Neural Networks

Aristotle Universtity Thessaloniki - School of Informatics

Assignment 3: PCA with Hebbian Learning

Antoniou, Antonios - 9482

aantonii@ece.auth.gr

GitHub repository can be found here

Introduction

For the final assignment of the Semester for the Deep Learning course, we are given the option to implement a PCA model using Hebbian Learning. The results of that analysis can be then used for less memory and time consuming training of a prediction model, since it will be trained on the transformed dataset, consisting of a **linear combination** of the main components.

Suppose the number of those components is C, and the original number of components (also known as **features**) is F. We are then sure that $C < F$. This is the reason why the consequent training is executed on less data (but generally speaking no less *information*), therefore in shorter time.

As usual, before we dive into the specifics of the implementation, we will need to clear up the most significant parts of the terminology.

PCA

The idea behind PCA (**Principal Component Analysis**) is that not all features in a dataset have the same gravity, or "explain" the same amount of information. This information, that we also mentioned above, generally has to do with the amount of variance a specific feature is responsible for in the entirety of the matrix of samples.

This analysis is based on the concept of **eigenvalue decomposition**. A matrix is analysed into its eigenvalues and eigenvectors. The eigenvectors that correspond to the largest eigenvalues are the components that carry the largest percentage of information, which is measured by the variance of the result of the analysis. Ideally, we want to keep about **90% of the original variance**, whenever possible.

Hebbian Learning

The Hebbian rule is pretty simple and intuitive: The more two neurons agree on an output, the greater the weight between them should be. It can be expanded to the opposite as well: When two neurons output values of different sign, the connection between them should be weakened.

Mathematically speaking, for two neurons i and j, this can be expressed as:

- (1) $\Delta w_{ji}(n) = \eta \cdot y_j(n) \cdot x_i(n)$,

for a specific moment in time n, and a selected learning rate η. The problem arising from this rule is the lack of boundaries imposed to the weights. Below, we are going to lay out the way in which this challenged is tackled. Now that basics of Hebbian learning are cleared up, what remains to be specified is how PCA can be performed utilizing (1).

PCA with Hebbian Learning

For an ANN to perform PCA using the Hebbian rule, there needs to be a much stricter architecture than the usual Networks. More specifically, a Network is able to perform PCA, if it consists of **two layers**: The input layer, and the output layer, with as many nodes as the desired components. This means that the training boils down to training the weights connecting all input nodes with all output nodes.

The rule in **(1)** is the basis of the training, and to make sure that the solution doesn't diverge we use **Oja's rule**:

- (2) $\Delta w_{ji}(n) = \eta \cdot y_j(n) \cdot [x_i(n) - y_j(n) w_{ji}(n)]$

This way, we can ensure that $|w| = 1$.

However, this algorithm can only work properly when we need to filter out the *principal component* of the features (i.e. when the population of the output layer is 1). In any other case, where $C > 1$, the algorithm needs to be enhanced in order to guarantee both convergence and the results we expect. In practice, for the weight between output node j and input node i, the update rule will be transformed to:

- (3) $\Delta w_{ji}(n) = \eta \cdot y_j(n) \cdot [x_i(n) - \sum_{k=1}^{j} y_k(n) w_{ki}(n)]$

To get a better grasp of (3): The output neuron with index j represents the j-th component. So for an output neuron $j + 1$, to only represent one component, and not accumulate the components before it, the outputs and respective weights have to be subtracted. In this way, the neuron only processes the remainder of the information that was omitted by the previous nodes.

Implementation

The PCA model was built from scratch based on two classes

Layer

`Layer` contains all the nodes of a layer, their respective outputs and current weights, together with the arrays keeping the Δw values, as well as the d values. The latter is an array of size $(C + 1) \times F$. Each row contains F sums (one for every input neuron), that is the result of the sum component in (3).

Model

This class encapsulates the two layers of the predictor, and is responsible to fit around the input dataset `X`. Apart from the layers, and corresponding nodes, it contains information on the learning rate η, the total time of training, and the results of the dimensionality reduction. Last, but certainly not least, it contains the threshold `tolerance`. In short, during each epoch, we keep a metric of the

amount of change each weight was subject to, `mean_dw`. Essentially, it is an average of the total changes made to each weight, in proportion to η and the number of samples S. **Empirically**, it was found that a tolerance of 10^{-9} per output node was a low enough value to assume equilibium.

So, a single iteration in an epoch during training is comprised of simple steps:

- The outputs `y` are calculated using the product of weights and inputs `x`,
- Based on both of the above, we calculate the `d` array, in order to then calculate the array of Δw values,
- We calculate `mean_dw` and see if the stability criterion is met. We require:
  - $$mean\_dw \leq (C+1) \cdot 10^{-9}$$
  - $$\#epochs > C+1$$

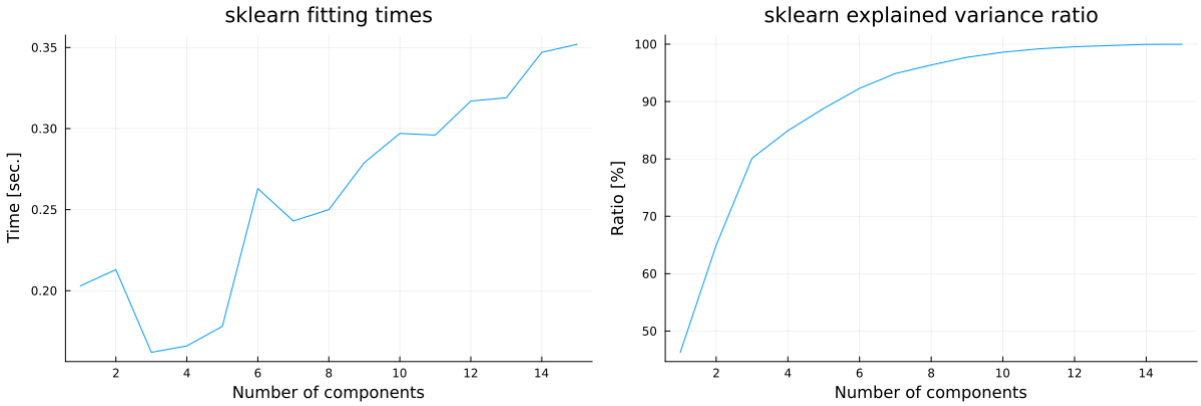
Now, why do we use $C+1$ output neurons instead of C? It is a solution that was introduced to the "ghost component" that appears in the place of the first principal component (no idea why this is happening). So, when we need 3 components, we train a model for 4, and start from the second one. This means that the calculation of the results of the PCA looks like:

```
def pca(self, x):
    nsamples = len(x)
    for s in range(nsamples):
        self._comps[s, :] = np.dot(self.layers[1].w[1:,:], x[s])
```

Testing

The Body Signals of Smoking dataset was used. We first conduct some test with PCA from the `sklearn.decomposition` module. The data is **immediately normalized**, so we can avoid overflows during the training of the Hebbian Network. The reason why we also use the normalized data for the direct solution is because we would like to measure the discrepancies between the models with as much common ground as possible.

We test for number of components C, in the range $[1, 10]$. This is more than enough testing, since we reach a ratio of 98.602 explained variance:



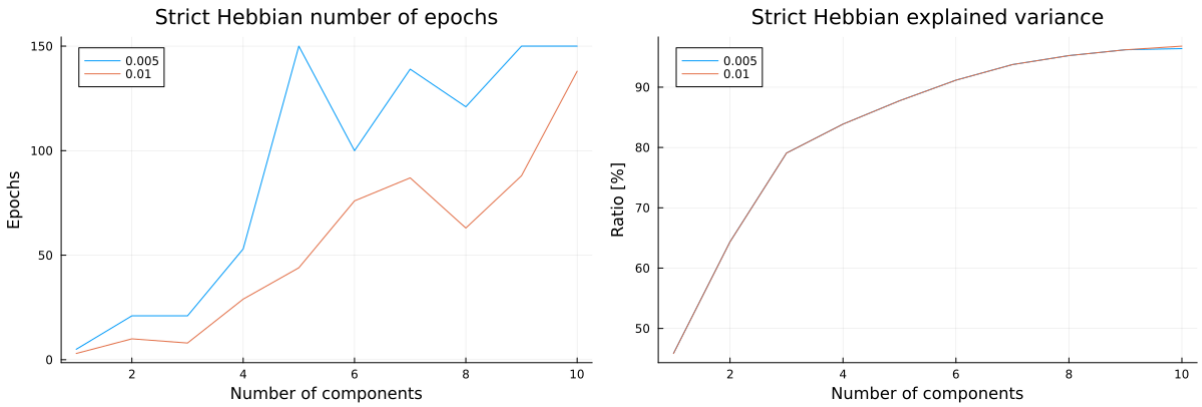
The dataset has a column named `ID`. It has no meaningful value, and should be totally discarded by the analysis. For one component, we take normalized data again, but don't drop the column. The result:

```
** Total percentage of variance explained: 90.579
```

This is a clear indicator that **we need to study the data we are dealing with**. While sophisticated algorithms can help with the results of any analysis, the first tool we have to use is the knowledge of the dataset, so that we know the measures we will have to take.

When it comes to the implemented model, once again we test for $C = [1, 10]$. We also test for various values of η, to see if the same results can be achieved through a lesser number of epochs. All tests for $\eta = 0.001$ reached the maximum number of epochs allowed by the model, which is **150**. So we narrow down the testing range to $\eta = \{0.005, 0.01\}$. The results for $\eta = 0.001$ and $\eta = 0.005$ are almost identical, so we don't worry about their integrity.

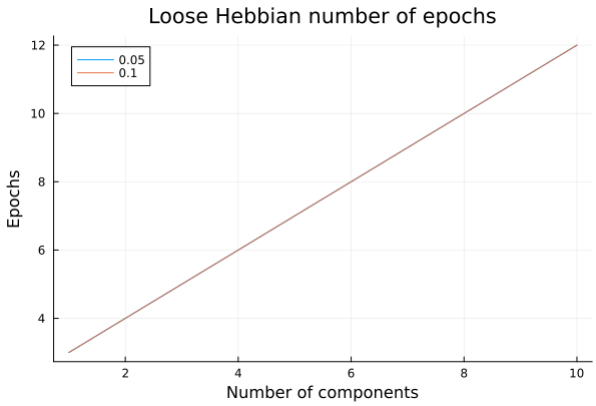
First we see the number of epochs it took the model to converge, then, for $\eta = 0.01$, we take a look at the execution time per epoch, so we can have a measure of the added complexity of an extra output node, and the weight adjustments that come with it.

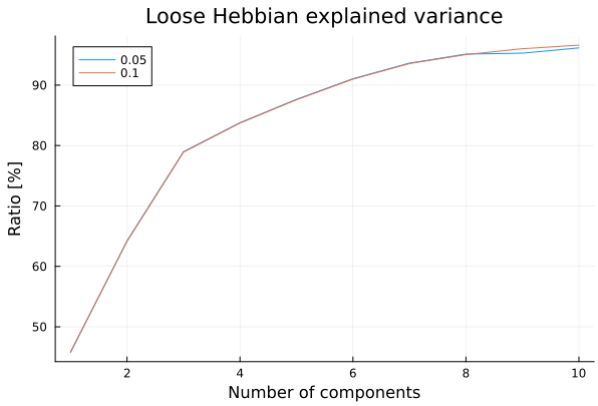


Components	time/epoch	Components	time/epoch
1	1,808	6	4,972
2	2,305	7	6,917
3	2,856	8	10,423
4	3,895	9	7,155
5	4,026	10	7,621

As expected, as complexity increases, the time per epoch also does, almost linearly.

For $C = \{0.05, 0.1\}$, we make the equilibrium criterion more lenient (tolerance per output node equal to 10^{-7}, from 10^{-9}) and compare the results:





Conclusion

For the dataset in question, we achieve comparable -if not identical- results, but at the expense of fitting time. While the standard PCA module consistently takes less than a second to fit, the Hebbian model needs orders of magnitude of that time. The more lenient criterion may cut that time down drastically, but the difference is still vast.