VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Demonstration of Autoencoder learning process

Soft computing

November 29, 2021 Alexander Polok (xpolok03)

1 Introduction

This project aims to demonstrate how Autoencoder neural network learns. As it was not hardly specified, I have focused on feature representation in 2D space following objective function and model weights changes. The learning process is demonstrated in a GUI application, which contains changes of an objective function, model weights and features representation over time. The purpose of this document is to clarify used theoretical knowledge introduced in section 2 with the model implementation described in section 3. Section 4 presents program manual. The project is available at GitHub[1] and it's licensed under Apache License 2.0.

2 Autoencoder

An autoencoder is a simple learning circuit that aims to transform inputs into outputs with the least possible amount of distortion. In other words, a model is discovering structure within data to develop a compressed representation of the input. Autoencoders were first introduced in the 1980s to address the problem of *back-propagation without a teacher*, by using the input data as the teacher [7]. While conceptually simple, autoencoders play an important role in machine learning. Figure 1 shows sample autoencoder architecture, which contains:

- an encoder that maps the input into the code
- a decoder that maps the code to a reconstruction of the input

.

Hence the model could only copy input signal to the output, autoencoders are typically forced to reconstruct the input approximately, preserving only the most relevant aspects of the data. This may lead to intuitive comparison with dimensionality reduction techniques like Principal component analysis [3] or Linear discriminant analysis [2]. However, autoencoders are used in many other fields. Such as image compression, image denoising, feature extraction, image generation, or sequence to sequence prediction [5].

[1]https://github.com/Lakoc/SFC_project

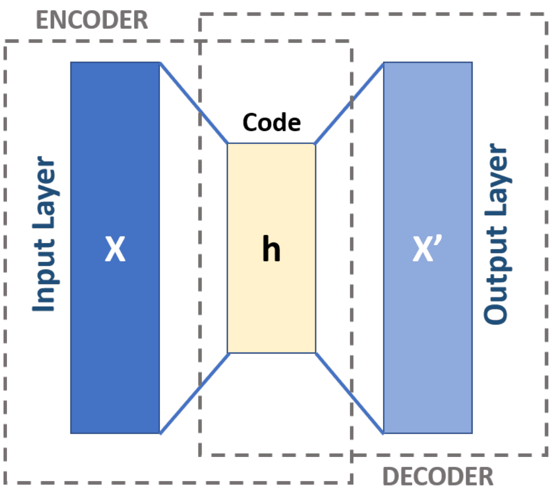


Figure 1: Schema of a basic Autoencoder architecture [8]. An autoencoder has two main parts, left side of the graph is called an encoder. It is responsible of mapping the input into the code, that code later referred are feature vector is visualized in the GUI application. Part on the right side is a decoder. It maps the code to a reconstruction of the input.

2.1 Architecture

Only neural-based autoencoder[2] is considered in the rest of this document, although autoencoder is a general auto-associative model. Model is also constrained with at least 3 layers $L_1, \ldots, L_N$, where $N \geq 3$, with the code layer L_i containing exactly two neurons $\sim$ features in 2D space, $|L_i| = 2 \land i \neq 1 \land i \neq N$. These constraints allow the simple visualization of feature vectors through time.

Let's note $\mathbf{x} \in \mathbb{R}^N$ as input vector, $\mathbf{y} \in \mathbb{R}^N$ as output vector and $\mathbf{f} \in \mathbb{R}^2$ as feature vector. Thus goal of model is to learn way to encode vector $\mathbf{x}$ into vector $\mathbf{f}$, in such a way, that vector $\mathbf{y}$ can be decoded from $\mathbf{f}$. Without loss of generality we may normalize all those vectors in a following way $\mathbf{x}, \mathbf{y}, \mathbf{f} \in \langle -1; 1\rangle^N$. This normalization of input data is realized for visualization purposes. This normalization also allows us to use activation functions described in the next subsection 2.3 without limitation.

Let consider following notations, $\mathbf{W}_l$ stands for matrix of weights from layer $l-1$ to l layer, $\mathbf{b}_l$ as bias vector and σ as activation function. Thus we can represent

$$\mathbf{z}_l = \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l \tag{1}$$

[2]The one that is modeled by neural network [6].

as vector of weighted inputs to the neurons in the layer l, and

$$\mathbf{a}_l = \sigma(\mathbf{z}_l) \tag{2}$$

as vector of activations of the neurons in layer l.

$$\mathbf{a}_1 = \mathbf{x} \tag{3}$$

Goal is to get $\mathbf{a}_N$ as close as possible to the desired output $\mathbf{y}$, which is equal to $\mathbf{x}$ as specified above.

2.1.1 Backpropagation algorithm

Model weights updates are calculated by the backpropagation algorithm. Details of these algorithms are well described in the second chapter of Neural Networks and Deep Learning by Michael Nielsen [6].

2.2 Objective function

As objective function in next sections is considered root mean square error [4], which measures, how good model is in terms of transforming inputs into outputs

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y_n - a_{N,n})^2}. \tag{4}$$

2.3 Activation functions

To allow a model to learn more complex functions, activation functions are used to add nonlinearity to the weighted input of neurons. Following activation functions are used.

2.3.1 Identity

Although identity does not add any non linearity, identity function is available in the implemented program for demonstration purposes. Identity is defined as

$$\sigma(\mathbf{x}) = \mathbf{x}.by \tag{5}$$

2.3.2 Sigmoid

Activation function that projects input value to the range $(0, 1)$ by following equation:

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{x}}}. \tag{6}$$

2.3.3 Hyperbolic tangent

Activation function that projects input value to the range $(-1, 1)$ by following equation:

$$tanh(\mathbf{x}) = \frac{sinh(\mathbf{x})}{cosh(\mathbf{x})} = \frac{e^{2\mathbf{x}} - 1}{e^{2\mathbf{x}} + 1}. \tag{7}$$

2.4 Optimizers

As described above the backpropagation algorithm computes the gradient of the cost function for a single training example. Updating weights this way would be very slow, hence a mini-batch stochastic descent algorithm is used. In particular, given a mini-batch of m training examples, the algorithm applies a gradient descent learning step based on that mini-batch. The algorithm is also well described in the already mentioned book Neural Networks and Deep Learning [6]. To obtain better results in a faster time the following optimizers can be used. Their descriptions are based by the ML Glossary article [1].

2.4.1 Stochastic Gradient Descent

Let's define Stochastic Gradient Descent update rule

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \alpha\nabla_{\boldsymbol{\theta}_{i-1}} J_{minibatch}(\boldsymbol{\theta}_{i-1}), \tag{8}$$

where $\boldsymbol{\theta}$ is a vector containing all of the model parameters, J is the loss function, $\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta})$, is the gradient of the loss function with respect to the parameters on a minibatch of data, and α is the learning rate.

2.4.2 Momentum

Momentum is technique of optimizing learning of model by keeping track of $\mathbf{m}$, a rolling average of the gradients:

$$\mathbf{m}_i = \beta_1 \mathbf{m}_{i-1} + (1 - \beta_1)\nabla_{\boldsymbol{\theta}_{i-1}} J_{minibatch}(\boldsymbol{\theta}_{i-1}), \tag{9}$$
$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \alpha\mathbf{m}_i, \tag{10}$$

where β_1 is a hyperparameter between 0 and 1 (often set to 0.9).

2.4.3 RMSProp

Root Mean Square Prop (RMSProp) works by keeping track of $\mathbf{v}$, an exponentially weighted average of the squares of past gradients. RMSProp then divides the learning rate by this average to speed up convergence:

$$\mathbf{U} = \nabla_{\boldsymbol{\theta}_{i-1}} J_{minibatch}(\boldsymbol{\theta}_{i-1}), \tag{11}$$

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1-\beta_2)(\mathbf{U} \odot \mathbf{U}), \tag{12}$$

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \alpha \frac{\nabla_{\boldsymbol{\theta}_{i-1}} J_{minibatch}(\boldsymbol{\theta}_{i-1})}{\sqrt{\mathbf{v}_i} + \epsilon}, \tag{13}$$

where $\odot$ denote elementwise multiplication (Hadamard product), ϵ is very small value to avoid dividing by zero and β_2 is a hyperparameter between 0 and 1 (often set to 0.99).

2.4.4 Adam

Adaptive Moment Estimation (Adam) combines ideas from both RMSProp and Momentum (same computation of $\mathbf{v}$ and $\mathbf{m}$, followed by correction depended by current iteration of learning process t). It computes adaptive learning rates for each parameter and works as follows:

$$\mathbf{m}_i^{corrected} = \frac{\mathbf{m}_i}{1-(\beta_1)^t}, \tag{14}$$

$$\mathbf{v}_i^{corrected} = \frac{\mathbf{v}_i}{1-(\beta_2)^t}, \tag{15}$$

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \alpha \frac{\mathbf{m}_i^{corrected}}{\sqrt{\mathbf{v}_i^{corrected}} + \epsilon}. \tag{16}$$

3 Implementation

The project was implemented in the python[3] language. Neural network representation and algorithms described above are written in pure numpy[4]. Graphical user interface was designed in PySimpleGUI[5]. Plots are processed in matplotlib[6]. Source files are divided into following modules:

- gui - create layouts, handle events, updates window
- graphs - represents neural network state, follows weights updates
- plots - plot feature representation and objective function over time
- training - model logic, keeps representation of model and learning state

After starting the program loads the default config file and a simple form is shown to the user. The form may be seen in figure 2. The user is allowed to change model hyperparameters (such as network architecture and optimizer parameters). Users may generate data from several multivariate normal distributions.

When the form is submitted and the config is valid, the neural network model, trainer instance, and figures are initialized and the user is moved to the second window (may move between windows at any time). Data from every cluster are generated from specified multivariate normal distributions then concatenated to the single matrix and normalized to the interval $\langle -1; 1 \rangle^N$ as mentioned above, normalization is processed by dividing all data by the highest value generated. Data are then initially randomly split into mini-batches.

The second window is shown in figure 3, it consists of three main figures (input data features in 2D space, objective function, and model weights - biases are displayed as neuron color and weights colors and widths are proportional to magnitudes of weights) and several buttons, which are self-explanatory.

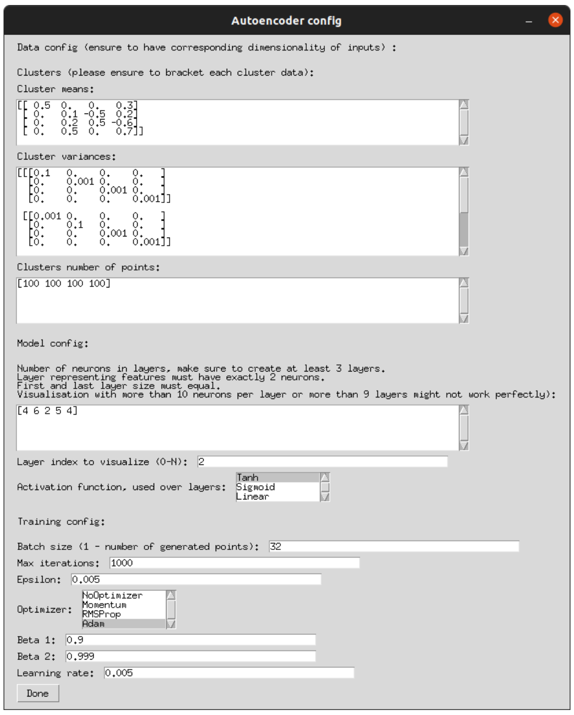
The model features colors that distinguish the clusters from which the data come. The figure shows well how is model trying to do self-associate input and how those clusters preserve in reduced dimension space. Model training might be stopped when convergence chosen by the ϵ parameter is reached.

[3] https://www.python.org/downloads/release/python-380/

[4] https://numpy.org/

[5] https://pysimplegui.readthedocs.io/en/latest/

[6] https://matplotlib.org/



```
Data config (ensure to have corresponding dimensionality of inputs) :

Clusters (please ensure to bracket each cluster data):
Cluster means:
[[ 0.5  0.   0.   0.3]
 [ 0.   0.1 -0.5  0.2]
 [ 0.   0.2  0.5 -0.6]
 [ 0.   0.5  0.   0.7]]
Cluster variances:
[[[0.1   0.    0.    0.   ]
  [0.    0.001 0.    0.   ]
  [0.    0.    0.001 0.   ]
  [0.    0.    0.    0.001]]

 [[0.001 0.    0.    0.   ]
  [0.    0.1   0.    0.   ]
  [0.    0.    0.001 0.   ]
  [0.    0.    0.    0.001]]
Clusters number of points:
[100 100 100 100]

Model config:

Number of neurons in layers, make sure to create at least 3 layers.
Layer representing features must have exactly 2 neurons.
First and last layer size must equal.
Visualisation with more than 10 neurons per layer or more than 9 layers might not work perfectly):
[4 6 2 5 4]
Layer index to visualize (0-N): 2
Activation function, used over layers: [✓ Tanh] Sigmoid Linear

Training config:

Batch size (1 - number of generated points): 32
Max iterations: 1000
Epsilon: 0.005
Optimizer: NoOptimizer Momentum RMSProp [✓ Adam]
Beta 1: 0.9
Beta 2: 0.999
Learning rate: 0.005
```

Figure 2: Hyperparameters form. User can define clusters from which data will be generated. User can also experiment with the different architecture of the model and training process. The form cannot be submitted until all inputs are valid.

4 Manual

Implementation presupposes availability of the anaconda tool[7]. Once is anaconda present a new environment could be created by the following command.

```
$ conda env create -f environment.yml
```

Activate new enviroment.

```
$ conda activate SFC
```

Export python path.

```
$ export PYTHONPATH=$PWD:$PYTHONPATH
```

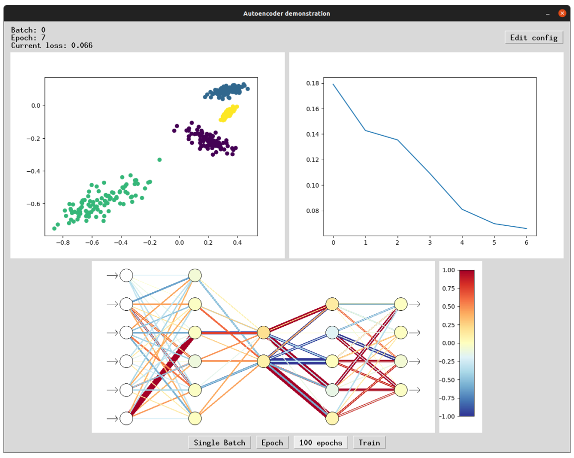
And finally run gui application.

```
$ python3 main.py
```

5 Conclusion

A simple autoencoder neural network was implemented and a GUI application demonstrates learning of that

[7] https://www.anaconda.com/



```
Batch: 0
Epoch: 7
Current loss: 0.066
```

Figure 3: Training window. A Current iteration, batch, and loss can be seen at the top of the window. User may train the model with buttons at the bottom of the window. Plot at the top left visualize current data feature representations, at the right side one can see objective function values over time. At the bottom of these plots model weights and biases are visualized.

network. In the feature space, several phases of learning may be distinguished, space rotations could be associated with weights changes. Provided program might be expanded in the future by implementing additional features such as weight normalization normalization, dropout, or other techniques for an even faster and better learning process.

References

[1] Optimizers. URL `https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html`.

[2] S. Balakrishnama and A. Ganapathiraju. Linear discriminant analysis-a brief tutorial. *Institute for Signal and information Processing*, 18(1998):1–8, 1998.

[3] C. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 10 2007. ISBN 0387310738.

[4] T. Chai and R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? *Geosci. Model Dev.*, 7, 01 2014. doi: 10.5194/gmdd-7-1525-2014.

[5] N. Mantri. Applications of autoencoders, Jul 2019. URL https://iq.opengenus.org/applications-of-autoencoders/.

[6] M. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL https://books.google.cz/books?id=STDBswEACAAJ.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 026268053X.

[8] Wikipedia. Autoencoder — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Autoencoder&oldid=1055131436, 2021. [Online; accessed 23-November-2021].