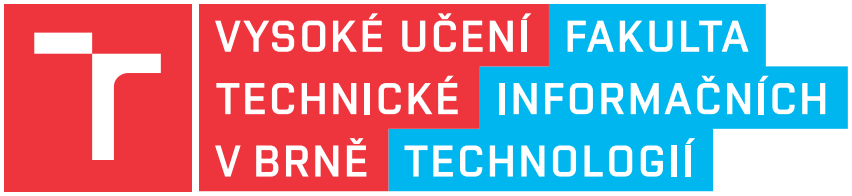

Demonstration of Autoencoder learning process

Soft computing

November 23, 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 in accordance with objective function and model weights changes. 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 *backpropagation 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].

2.1 Architecture

Although autoencoder is general autoassociative model in rest of this document, only neural based autoencoder[2] will be considered. Model will be also constrainted 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, $|L_i| = 2 \wedge i \neq 1 \wedge i \neq N$. These constraints allow simple visualization of feature vectors through time.

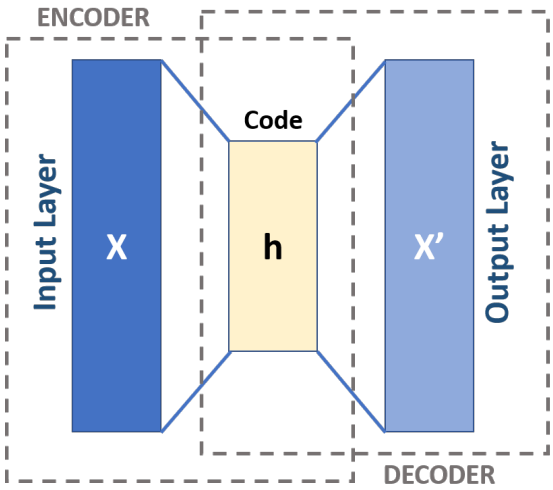

Figure 1: Schema of a basic Autoencoder architecture [8].

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 a lose of generalization we may normalize all those vectors in a following way $\mathbf{x}, \mathbf{y}, \mathbf{f} \in \langle -1; 1 \rangle^N$ for visualization purposes. This also allows us to model first layer neurons as identity function and 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

$$\mathbf{z_l} = \mathbf{W_l a_{l-1}} + \mathbf{b_l} \tag{1}$$

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. Obviously $\mathbf{a_1} = \mathbf{x}$ and $\mathbf{a_N}$ should gets as close as possible to the desired output $\mathbf{y}$, which is equal to $\mathbf{x}$ as specified above.

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

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

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

2.2 Objective function

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

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y_n - a_{N,n})^2}. \quad (3)$$

2.3 Activation functions

To allow model to learn more complex functions activation functions are used to add non linearity to weighted input of neuron. 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

$$\sigma(\mathbf{x}) = \mathbf{x}. \quad (4)$$

2.3.2 Sigmoid

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

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{x}}}. \quad (5)$$

2.3.3 Hyperbolic tangent

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

$$tanh(\mathbf{x}) = \frac{sinh(\mathbf{x})}{cosh(\mathbf{x})} = \frac{e^{2\mathbf{x}} - 1}{e^{2\mathbf{x}} + 1}. \quad (6)$$

2.4 Optimizers

As described above the backpropagation algorithm computes the gradient of the cost function for a single training example. Updating weights by this way would be very slow, hence 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. Algorithm is also well described in already mentioned book Neural Networks and Deep Learning [6]. To obtain better results in faster time following optimizers can be used.

Let's define Stochastic Gradient Descent update rule

$$\boldsymbol{\theta_i} = \boldsymbol{\theta_{i-1}} - \alpha \nabla_{\boldsymbol{\theta_{i-1}}} J_{minibatch}(\boldsymbol{\theta_{i-1}}), \quad (7)$$

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. Following optimizers descriptions are based by the ML Glossary article [1].

2.4.1 Momentum

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

$$\mathbf{m_i} = \beta_1 \mathbf{m_{i-1}} + (1 - \beta_1) \nabla_{\boldsymbol{\theta_{i-1}}} J_{minibatch}(\boldsymbol{\theta_{i-1}}), \quad (8)$$

$$\boldsymbol{\theta_i} = \boldsymbol{\theta_{i-1}} - \alpha \mathbf{m_i}, \quad (9)$$

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

2.4.2 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:

$$\mathbf{U} = \nabla_{\boldsymbol{\theta_{i-1}}} J_{minibatch}(\boldsymbol{\theta_{i-1}}), \quad (10)$$

$$\mathbf{v_i} = \beta_2 \mathbf{v_{i-1}} + (1 - \beta_2)(\mathbf{U} \odot \mathbf{U}), \quad (11)$$

$$\boldsymbol{\theta_i} = \boldsymbol{\theta_{i-1}} - \alpha \frac{\nabla_{\boldsymbol{\theta_{i-1}}} J_{minibatch}(\boldsymbol{\theta_{i-1}})}{\sqrt{\mathbf{v_i}} + \epsilon}, \quad (12)$$

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).

2.4.3 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:

$$\mathbf{m_i^{corrected}} = \frac{\mathbf{m_i}}{1-(\beta_1)^t}, \tag{13}$$

$$\mathbf{v_i^{corrected}} = \frac{\mathbf{v_i}}{1-(\beta_2)^t}, \tag{14}$$

$$\boldsymbol{\theta_i} = \boldsymbol{\theta_{i-1}} - \alpha \frac{\mathbf{m_i^{corrected}}}{\sqrt{\mathbf{v_i^{corrected}}}+\epsilon}. \tag{15}$$

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 start the program loads default config file and simple form is shown to user. Form may be seen on figure 2. User is allowed to change model hyperparameters (such as network architecture and optimizer parameters). User may generate data from several multivariate normal distributions.

When is form submitted and config is valid, neural network model, trainer instance and figures are initialized and the user is moved to the second window (may move over windows at any time). Second window is shown on figure 3, it consist 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) and several buttons, which are self explaining.

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

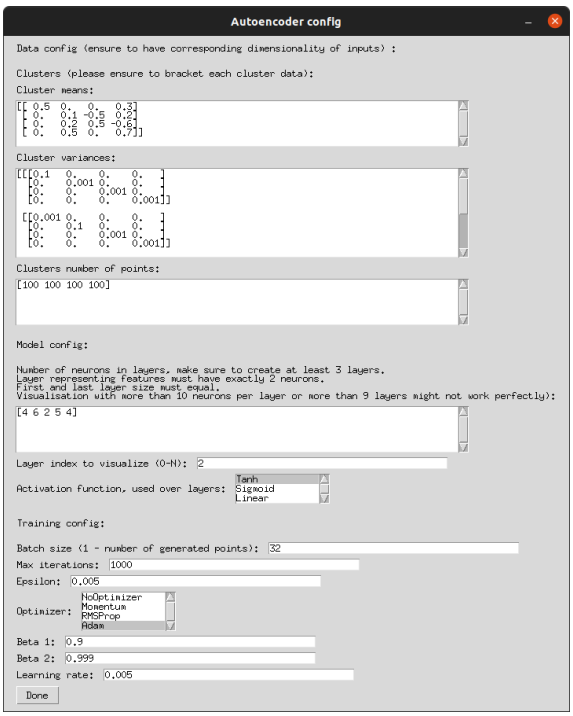

Figure 2: Hyperparameters form.

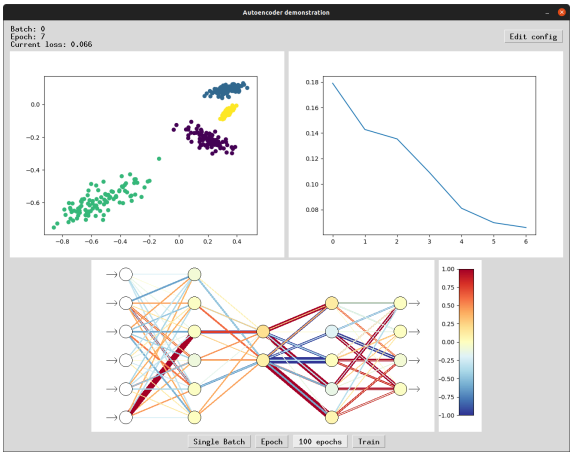

Figure 3: Training window.

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

[4] https://numpy.org/

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

[6] https://matplotlib.org/

4 Manual

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

Simple autoencoder neural network was implemented and gui application which demonstrates learning of that network. In the feature space several phases of learning may might be distinguished, space rotations could be associated with weights changes. Provided program satisfy requirements and may be in the future expanded by normalization, dropout or other techniques for 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].

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