A Peek Into Autoencoder Neural Networks

Bibidh Subedi, Aayush Dhungana

Abstract

This paper presents the design, implementation, and training process of an Autoencoder Neural Network aimed at reconstructing a set of instances for each input instance. This work mostly explores the intuition behind the neural network rather than the rigorous mathematics behind the subject. This work is intended to be light in concept with a specific configuration of network being implemented and is for people getting into machine learning with neural networks.

Introduction

Artificial neural networks are non-linear mapping systems with a structure loosely based on principles observed in the biological nervous system [1]. It consists of several neurons organized in layers connected with some numerical ‘weights’. The major objective in training any neural network is to find the most optimal value of these ‘weights’.

In our implementation, we use the backpropagation algorithm in its gradient descent form to train the network to find these weights, enabling the neural network to reconstruct the input. The working of this back propagation is verified empirically. The network itself is implemented in C++ without the help of any external machine learning or data handling libraries. The final network is then trained on some data, and its functionality of data reconstruction is successfully implemented. Small changes to the configuration of the network allows the network to be used as a Denoising Autoencoder and Sparse Autoencoder.

Architecture of an Autoencoder Neural Network

In this section, we will see the components of an Autoencoder Neural Network and their role in the overall network architecture, with the details left out for a later section.

Neurons

Neurons are the most fundamental part of a neural network. A neuron holds a value that is a result of some previous calculation (except for the neurons in the first layer of the network). It processes input data, applies weights and biases, and passes the result through an activation function to produce an output.

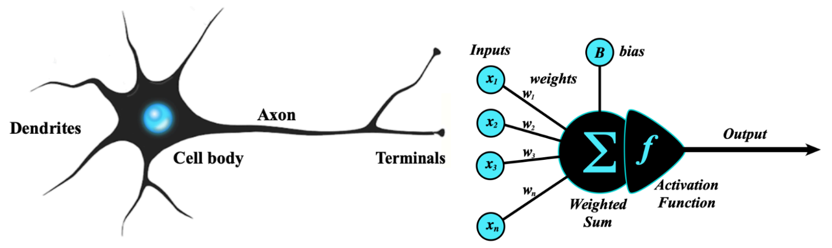

Fig: A biological and an artificial neuron [1]

Activation Functions

Activation functions are used to introduce nonlinearity to our model. Activation functions are tricky as they very heavily affect the network. The activation function is chosen on the basis of the requirement. There are a lot of different activation functions, but we will primarily focus on 2 activation functions that have been used in our model.

1. Sigmoid
   The sigmoid function basically outputs values between 0 and 1, which makes it useful for binary classification and binary encoding. And as we are going to deal with binary data in our model, we shall be using Sigmoid as an activation function for the output layer.
   $$\sigma(x) = \frac{1}{1+e^{-x}}$$

2. Leaky RELU
   RELU outputs the input if it’s positive and zero otherwise. But we use Leaky RELU instead of RELU in attempts to fix the "dying RELU" problem by allowing a small, non-zero gradient when the input is negative.
   $$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

Layers

Input Layer

The input layer consists of neurons that correspond to the input values the network receives from the real world. These values, known as features, represent the classification or label the network aims to identify. In the mathematical model y=f(x), the input x is represented as an n-dimensional vector.This input vector is commonly referred to as the feature vector.

In our case, we will be attempting to reconstruct a set of input vectors of n dimensions, so we will be feeding this vector into the input layer of our network.

Hidden Layers

From the input layer, information flows to a hidden layer containing neurons that process signals received from adjacent layers (either preceding or succeeding layers). A neural network may include multiple hidden layers.

In our case, we will be using 1 hidden layer. Please note that the architecture of a layer is a hyperparameter of the neural network. Determination and optimization of these hyperparameters, itself is a separate field in machine learning and artificial intelligence. We, however, have used the trusty hit-and-trial method as our network is fairly simple and we can bruteforce through a lot of combinations to search for optimal values of those hyperparameters.

Output Layer

The output layer contains neurons that represent the network's final output or response after processing the input. Neurons in the output layer can be modeled as classifications or labels the network aims to recognize, with values ranging from 0 to 1. A value closer to 1 indicates stronger confidence that the input corresponds to that classification or label.

For example, if the task is to classify images into "apples," "oranges," or "others," the possible outputs can be represented as follows:

For "apple," the output may be:

$$[f(x) = \begin{bmatrix}1 & 0 & 0\end{bmatrix}]$$

For "orange," the output may be:

$$[f(x) = [0 \quad 1 \quad 0]]$$

For any other fruit, the output may be:

$$[f(x) = [0 \quad 0 \quad 1]]$$

The dimensions of the output layer depend on the purpose of the neural network. As we are trying to reconstruct the input, the dimensions of the output layers will be the same as those of the input layer.

Interconnection of Neurons

A neural network can contain many layers. These layers are connected sequentially, forming a network of connections among their neurons. These connections are mathematically realized by the weight and bias terms. The weight term is essentially a measure that tells how significant a neuron's output is as input to another neuron, and the bias acts as an offset independent of the input data. In a densely connected neural network (such as ours), each neuron in a layer links to all the neurons in adjacent layers.

Layers within a neural network can also be sparsely connected and can have different nuances depending on the design of the network. Autoencoders themselves can have variations where they employ sparse connections through the use of regularization techniques like dropout, where some neurons during the training process are randomly dropped (output set to 0) so that the network does not rely heavily on any specific neuron(s) for prediction. This helps in reducing overfitting and improving generalization so that the network does not pick up on the noise from the training data and can predict on untrained data.

How exactly does a neural network 'Learn'

Since the interconnection between neurons is merely the weight and bias term associated with the connection, training a neural network means obtaining the best values for these terms for all the connections in the network. This is obtained by iteratively adjusting these terms during subsequent data passes through the algorithm called epochs. During a forward pass, as the input data propagates through the network, the produced output is compared with the target value/label using a loss/error function. This function gives a measure of how well the network is performing. Some common loss functions include the mean squared error loss, generally used in regression tasks, and cross-entropy loss, which is more suited for classification tasks. In backpass, the loss at the output layer is propagated back to the hidden layers, and an optimization algorithm such as gradient descent is used to calculate the impact of each weight and bias term on this loss so that they can be updated to minimize the value of the loss function. Over multiple epochs, the network converges towards a configuration that enables it to make accurate predictions or perform the desired task, such as reconstructing input vectors.

Working of An Artificial Neural Network

Forward Propagation

It is the initial step in training a neural network. During forward propagation, each neuron receives inputs from the preceding layer, multiplies them by weights, adds a bias, and applies a non-linear activation function to the weighted sum before passing the output to the next layer. It can be mathematically represented as:

Forward pass to compute output:

$$z^{(l)} = W^{(l)}h^{(l-1)} + b^{(l)}$$

Activation of layer l:

$$h^{(l)} = f(z^{(l)})$$

Where f(z) is an activation function discussed previously.

Back Propagation

Back propagation is the part where the network uses some optimization algorithm like Gradient Descent to improve the accuracy of the network by updating its weights and biases. By taking the partial derivative of the error of the network with respect to each weight, we learn a little about the direction the error of the network is moving. [2] Back propagation is basically the neural network using the feedback (the error gradients) and figuring out which specific parameters to change in the network by how much in order to decrease the value of error in the next cycle.
Note that these gradients are merely the partial derivatives of the loss function (which we need to minimize).

In our network, back propagation occurs in 3 steps.

1. Gradient Computation
   Here we calculate gradients for each layer of the network based on the error derivatives and activation derivatives.
   This is done through two steps:
   a. Output Layer Gradients
      $$\nabla = \delta \cdot \sigma'(z)$$
      δ: Error derivative
      $\sigma'(z)$: Derivative of the activation function at the output layer.

   b. Hidden Layer Gradients
      $$\nabla = (\nabla_{\text{next}} \cdot W^T) \cdot \sigma'(z)$$
      ∇_{next}: Gradient from the next layer.
      W^T: Transposed weight matrix from the next layer.
      $\sigma'(z)$: Derivative of the activation function at the current layer.

2. Weights and Biases Update
   Then the weights and bias are updated using the stochastic Gradient Descent function, which is explained below.

Gradient Descent

Gradient Descent is one of the most important optimization algorithms in the field of machine learning. Through continuous iteration, it obtains the gradient of the objective function, gradually approaches the optimal solution of the objective function, and finally obtains the minimum loss function and related parameters [3].

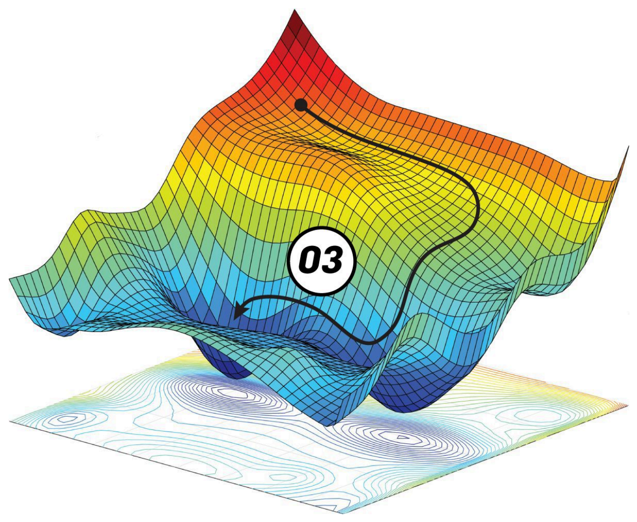

Fig: Gradient descent in a loss Landscape [4]

There are mainly 3 different types of Gradient Descent Algorithms.

1. Stochastic Gradient Descent

2. Batch Gradient Descent
3. Mini Batch Gradient Descent

Method	Update	Speed	Memory Usages	Convergence	Best Use Case
Stochastic Gradient Descent	Per Data Point	Fast Per Iteration	Low	Noisy, May Oscillate	Streaming data
Batch Gradient Descent	Entire Dataset	Slow Per Iteration	High	Precise But Slow	Small Datasets
Mini-batch Gradient Descent	Subset of the dataset	Balanced	Mid	Balanced, reduces noise	Large Data sets,

In our implementation, we have used the Stochastic Gradient Descent.

Knowing these gradient values, we can roughly determine how to adjust our weights to minimize the error. To simplify, since weight values influence the error, we aim to calculate the gradient values for each neuron (as weights are linked to neurons), starting from the output layer and propagating backward.

By using the gradients calculated in the previous step, we can now update the weights and biases as follows:.

$$W^{(l)} = W^{(l)} - \eta \nabla W^{(l)}$$
$$b^{(l)} = b^{(l)} - \eta \nabla b^{(l)}$$

Finally, one epoch is in the training process of our network.

The Network in Action

First we define the hyperparameters for the network

Architecture Hyperparameters
No. of Layers : 3
Topology of Layers : [3, 4, 3]
Activation Functions : Leaky RELU for Hidden Layers
Sigmoid for Output Layer

Optimization Hyperparameters
Learning Rate : 0.01
Loss Function : Binary Cross Entropy
Optimizer : Stochastic Gradient Descent
Batch size : 1
Number of Epochs : 2000

Regularization Hyperparameters
We do not use any kind of regularization hyperparameters, as our neural network is fairly trivial. However, we may need to add a certain dropout rate or L2 Regularization if we wish to use the neural network for other purposes, such as a denoising autoencoder or sparse autoencoder.

We then train the model on similar data as we wish to predict with the network. Training of the model is done by setting an input, forward propagating it, calculating loss, using loss to back propagate, and updating the weights and biases. We train the network on some combinations of 0 and 1 like [1, 1, 0], [0, 1, 0], etc., as that type of data is what we are trying to reconstruct.

After training the network for 2000 epochs, we get the following graph for the loss function.

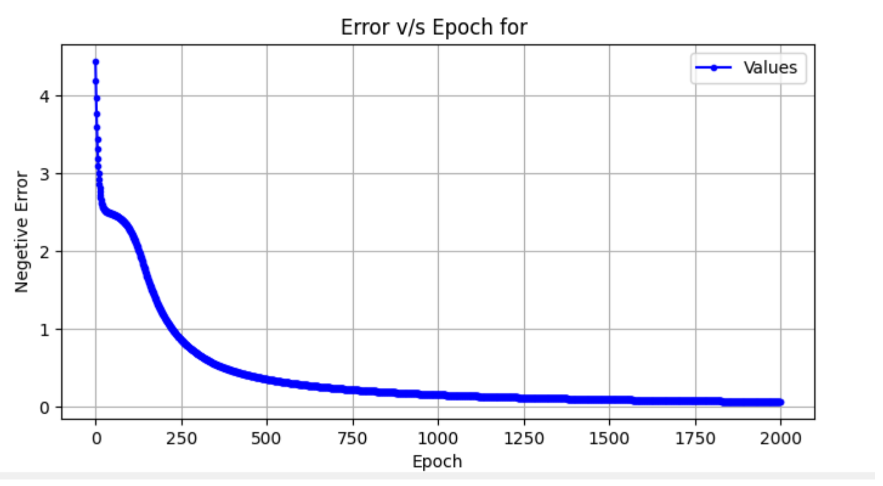

Fig: Network's Loss over epoch during training

Let's take a look at the final network visually.

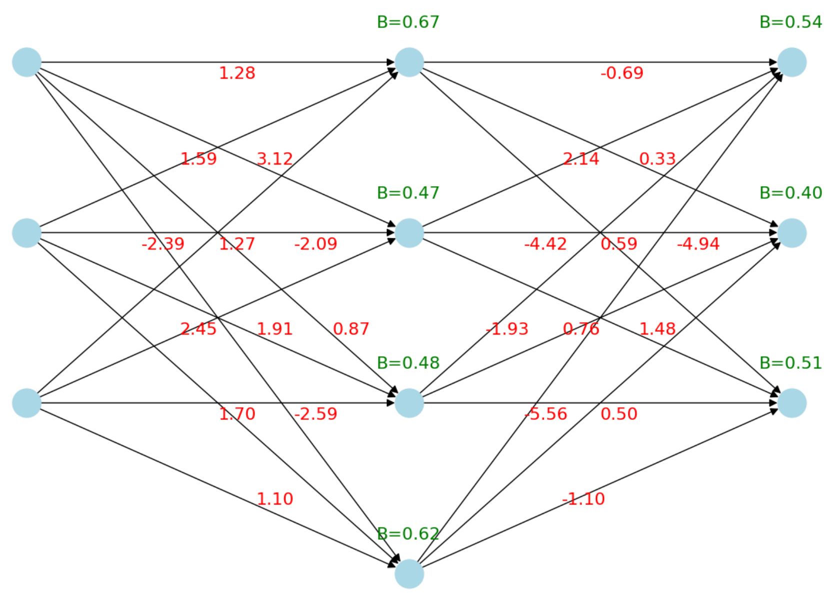

Fig: Visual Representation of the Final Trained Network

The above network is the actual representation of our final network. i.e., the weights and biases are true weights and biases which we achieved after training our network. Now we can use these weights and biases to give an input to the network and get the output.

Let's take an example of input [1, 0, 1] and forward run through the network.

Step 1: Hidden Layer 1

$$z^{(1)} = W^{(1)} \cdot \text{input} + b^{(1)}$$

$$z^{(1)} = \begin{bmatrix} 1.28098 & 3.1223 & -2.08707 & 0.870437 \\ 1.58878 & 1.26969 & 1.91459 & -2.58573 \\ -2.38938 & 2.4454 & 1.70066 & 1.09686 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.665083 \\ 0.468071 \\ 0.475733 \\ 0.618163 \end{bmatrix}$$

$$z^{(1)} = \begin{bmatrix} -0.443317 \\ 6.035771 \\ 0.089323 \\ 2.58546 \end{bmatrix}$$

Applying Leaky ReLU:

$$a_i^{(1)} = \begin{cases} z_i^{(1)} & \text{if } z_i^{(1)} > 0 \\ 0.01 \cdot z_i^{(1)} & \text{otherwise} \end{cases}$$

$$a^{(1)} = \begin{bmatrix} -0.00443317 \\ 6.035771 \\ 0.089323 \\ 2.58546 \end{bmatrix}$$

Step 2: Hidden Layer 2

$$z^{(2)} = W^{(2)} \cdot a^{(1)} + b^{(2)}$$

$$z^{(2)} = \begin{bmatrix} -0.691801 & 0.332733 & -4.93794 \\ 2.13842 & 0.592468 & 1.48174 \\ -4.41755 & 0.758111 & 0.5045 \\ -1.93134 & -5.56105 & -1.09996 \end{bmatrix} \cdot \begin{bmatrix} -0.00443317 \\ 6.035771 \\ 0.089323 \\ 2.58546 \end{bmatrix} + \begin{bmatrix} 0.53651 \\ 0.397774 \\ 0.511934 \end{bmatrix}$$

$$z^{(2)} = \begin{bmatrix} 8.05859916 \\ -10.33785547 \\ 6.67842892 \end{bmatrix}$$

Applying Sigmoid:

$$a_i^{(2)} = \frac{1}{1 + e^{-z_i^{(2)}}}$$

$$a^{(2)} = \begin{bmatrix} 0.99968373 \\ 0.0000323826 \\ 0.998743828 \end{bmatrix}$$

Final Output

$$\text{Output} = \begin{bmatrix} 0.99968373 \\ 0.0000323826 \\ 0.998743828 \end{bmatrix}$$

Conclusion

This work provides a practical explanation of the design, implementation, and training of an Autoencoder Neural Network, with a focus on reconstruction tasks. Through their ability to learn a compressed representation of higher dimensional data, autoencoders can be used to spot anomalies in large datasets. This work aims to serve as an accessible introduction for those new to machine learning, emphasizing practical intuition over mathematical complexity.

References

[1] MRIquestions. (n.d.). *What is a neural network?* Retrieved January4,2025,from https://s.mriquestions.com/what-is-a-neural-network.html#/

[2] Neural network applications to vehicle’s vibration analysis - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/a-A-biological-nervous-systems-and-b-an-artificial-neuron-model_fig1_256934640 [accessed 4 Jan 2025]

[3] Abraham, A. (2005). *Artificial Neural Networks*. In P. H. Sydenham & R. Thorn (Eds.), *Handbook of Measuring System Design* (pp. 901-908). John Wiley & Sons. Retrieved from https://wsc10.softcomputing.net/ann_chapter.pdf

[4] X. Wang, L. Yan and Q. Zhang, "Research on the Application of Gradient Descent Algorithm in Machine Learning," 2021 International Conference on Computer Network, Electronic and Automation (ICCNEA), Xi'an, China, 2021, pp. 11-15, doi: 10.1109/ICCNEA53019.2021.00014. keywords: {Machine learning algorithms;Stochastic processes;Training data;Machine learning;Linear programming;Classification algorithms;Task analysis;Gradient Descent;Logistic Regression;Machine Learning},

[5]https://blog.paperspace.com/part-3-generic-python-implementation-of-gradient-descent-for-nn-optimization/