AIML425 ASSIGNMENT 1

Benjamin McEvoy

300579954

1. INTRODUCTION

This document provides experimentation results; training a neural network to project 3D zero-mean unit variance normal-distributed (Gaussian) data onto a unit-radius sphere (the surface of a ball) To access the code-base this, repository or this google drive link: https://colab.research.google.com/drive/1Hs0FgHBe3lpPAAUalFE75M8vLWbQwWuS?

2. THEORY

2.1. Normal Distribution

The 3D zero-mean unit variance normal-distributed (Gaussian) describes three random variables that follow a normal distribution, specified by a 3D mean vector

$$\boldsymbol{\mu} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

represents the expected values (means) of the three variables, indicating that the distribution is centered at the origin, and the 3x3 covariance matrix:

$$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

indicates that the three variables are uncorrelated and each has a variance of 1. This means each variable is independently normally distributed with unit variance.

With the probability density function (pdf) given by: $f_{\mathbf{X}}(x_1, x_2, x_3) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^3|\boldsymbol{\Sigma}|}}$

A unit sphere in 3D space is defined as the set of all points $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ such that: $\boldsymbol{x^2 + y^2 + z^2 = 1}$, where points on the unit sphere have a Euclidean normal of 1. To transform the data onto the unit-radius sphere involves normalising vectors to have a unit length.

To complete 3D Gaussian data transformation onto a unit-radius sphere; a neural network is trained to map input vectors from the multivariate normal distribution (MND) to output vectors on the sphere, while maintaining the direction of the original vectors. As per the assignment brief, to generate samples of a MND of a 3D vector:

$$X \sim N(\mu, \sum)$$

2.2. Network Architecture

The architecture to be designed follows four fully-connected layers that are 3 (input), 20, 20 and 3 (output) wide, and non-linear activation for all but the last layer. Where it can be represented as:

$$\begin{aligned} \mathbf{h}_1 &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2) \\ \mathbf{h}_3 &= \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3) \\ \mathbf{y} &= \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4 \end{aligned}$$

Where, $\boldsymbol{x}$ is the input vector; $\boldsymbol{W_i}$ are the weight matrices; $\boldsymbol{b_i}$ are the bias vectors, $\boldsymbol{h_i}$ are the hidden layer activations; $\boldsymbol{\sigma}$ is the activation function; and $\boldsymbol{y}$ is the output vector. With the use of the loss function, Mean Squared Error (MSE) for calculations between the predicted outputs of:

$$\theta^* = \arg\max_\theta LL(\theta|D) = \arg\min_\theta \sum_{n \in A} \|y^{(n)} - f(x^{(n)}; \theta)\|^2$$

where $\boldsymbol{y}$ is the actual output and $\boldsymbol{f(x^n; \theta)}$ is the predicted output given the inputs $\boldsymbol{x^n}$.

3. EXPERIMENTS

3.1. Setup

3.1.1. Data Generation

In this experiment, the samples generated is 30000 for the specified MND distribution, splitting the samples into training, validation and test sets with a ratio of 60:20:20 splits. The data generation method generates random samples (in this case, the 30000 provided) from a multivariate normal distribution with a mean of zero and an identity covariance matrix, it then normalises the samples and stores them in instance variables. To visualise this, a subset of data points is taken from the dataset and 3D plotted into an original and transformed subplot shown in Figure 1. To properly plot the data points, the random vector points are scaled to 1, concerning the radius of the unit-radius sphere; dividing each vector $\boldsymbol{x}$ by its length.

3.1.2. Network Configuration

The creation of the Neural Network reflected the requirements of the assignment brief. The architecture designed followed a four fully-connected layer format. The input layer being the first fully-connected layer takes the input of size 3 and outputs size 20, the middle two fully-connected layers (the hidden layers) have both input and output size of 20, and the final fully-connected layer (output layer) takes an input layer of 20 and outputs size 3; with ReLU activation functions applied at the first three layers but not the last.

3.1.3. Loss and Optimisation

The criterion is set to PyTorch's MSELoss() function [1], mimimising the average squared difference between the predicted and actual values, represented by the formula:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Measuring how well the predictions match the target values.

The Adam optimiser is an adaptive learning rate optimisation algorithm with its choice of use due; to its efficiency and effectiveness [2]. The model.parameters() function calls for an iterator over the parameters of the model, which the optimiser will adjust during training to minimise the loss function. For this experiment, it is left in the default settings.

3.2. Results

Experimentation was conducted between a mini-batch and a half-batch approach, the direct numbers are in the code base; however, the calculations to obtain the batches are $(dataset.size * 0.01)$ and $(dataset.size * 0.5)$.

3.2.1. Performance

Both batches were set to 200 epochs. This choice was made based on the idea that efficiency and performance can be better evaluated over a smaller epoch window. The mini-batch loss function stabilizes rapidly within 30 epochs, as shown in Figure 3. According to the literature, mini-batches typically lead to faster convergence and quicker accuracy, as depicted in Figure 2. As the epochs progress, the graphs reveal distinct trends: the mini-batch experiences a sharp decline in loss before the 25th epoch, while the half-batch continues to decrease in loss until the final epoch, as illustrated in Figures 2 and 3. The data indicates that the half-batch—and potentially a full batch—requires significantly more epochs to achieve a similar level of performance compared to the mini-batch. Despite this, the only difference in the learning network is the batch size. By the 200th epoch, the half-batch approximation resembles a unit-radius sphere, although it is not as spherical as the mini-batch.

3.2.2. Error Vector Assessment

To further review the performance of the model, focus on point vector length was prioritised. This projection aims to have a perfect distribution at 1 to prove whether the model is efficient and robust enough to perform well; as seen in Figure 4. The narrow distribution at 1 indicates the desired outputs of the model, however, I neglected to evaluate the amount of angular errors in these vectors, only assessing the the point vector intersection with the unit-radius sphere.

4. TOOLS

This assignment used the tools of Pytorch [1], Numpy [3], Matplotlib [4], and Adam Optimiser [2] to carry out the theory mentioned. Pytorch was responsible for the functions to train, and calculate, where Numpy was also used for calculations and normalisation. Matplotlib was used to plot the data in a visual format, while the optimiser was used for its learning function and rate.

Google Colab [5] is where the code base is hosted and is provided with the runtime allocation and specifications of the following:

- **Model name:** Intel(R) Xeon(R) CPU @ 2.20GHz
- **Core(s) per socket:** 1
- **Thread(s) per core:** 2
- **RAM:** 12G
- **Available RAM:** 81G
- **Socket(s):** 1
- **L3 cache:** 55 MiB (1 instance)

5. CONCLUSION

In conclusion, the results and figures demonstrated that the mini-batch approach outperforms the half-batch method in terms of convergence speed. The mini-batch loss function stabilised quickly within 30 epochs, whereas the half-batch continued to decrease in loss throughout the entire training period, indicating slower convergence. The mini-batch method also achieved a more spherical approximation of the unit-radius sphere by the 200th epoch compared to the half-batch.

The four-layer neural network using ReLU activation functions, PyTorch's MSELoss function [1] and Adam optimiser [2], proved effective for this task. The Adam optimiser's adaptive learning rate contributed to performance and minimised the average squared difference between predicted and actual values.

While the model demonstrated a narrow distribution in Figure 4, indicating desired calculations, performances and evaluations; different methods in evaluating the proficiency of the model should be explored further such as Angular Error or extended dataset comparisons to provide a more comprehensive assessment of the model.

6. REFERENCES

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al., "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

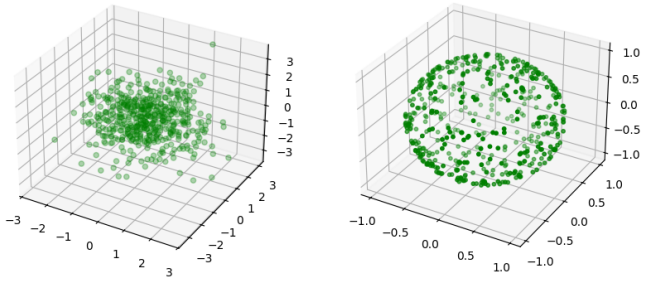
[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

[3] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al., "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[4] John D Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[5] Google, "Google colaboratory," Online, 2023, Available at: https://colab.research.google.com.

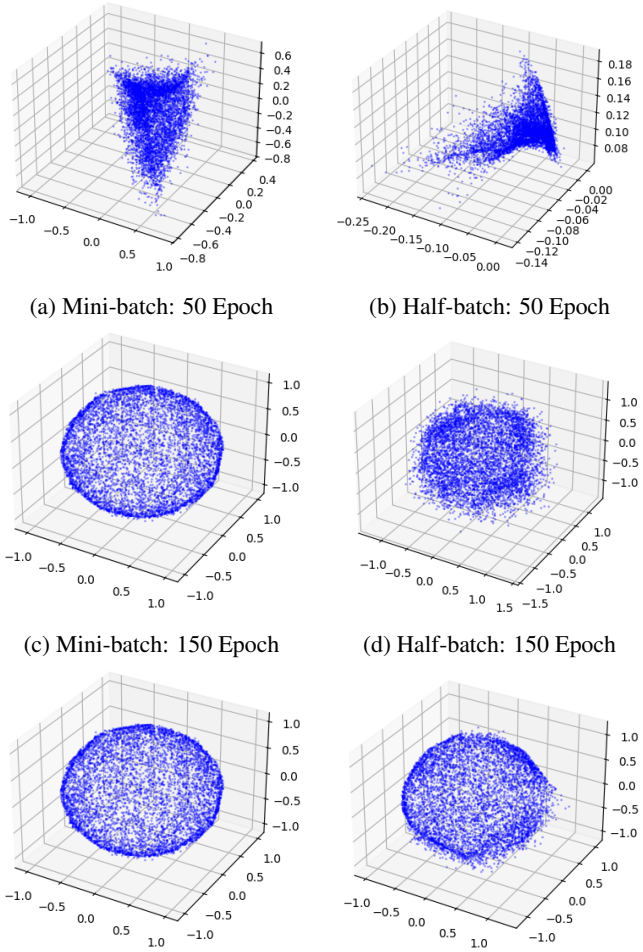
Appendix



(a) Multivariate Normal Distribution: 500 samples

(b) Projection of unit-radius sphere: 500 samples

Fig. 1: Training Data Visualisation



(a) Mini-batch: 50 Epoch

(b) Half-batch: 50 Epoch

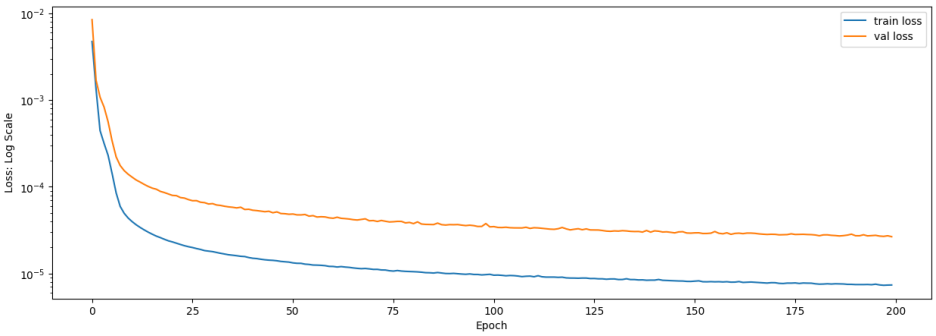
(c) Mini-batch: 150 Epoch

(d) Half-batch: 150 Epoch

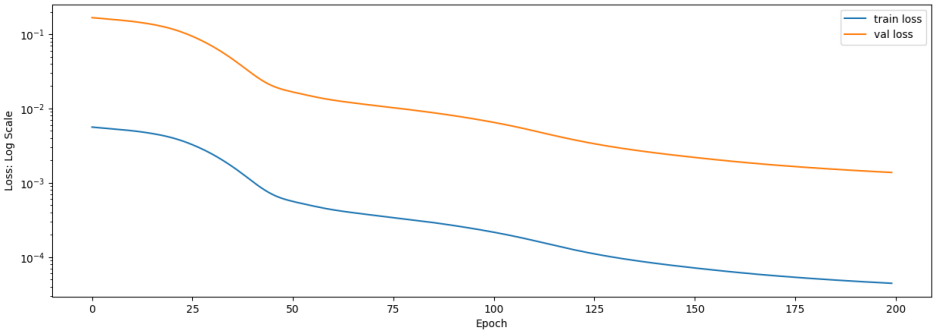
(e) Mini-batch: 200 Epoch

(f) Half-batch: 200 Epoch

Fig. 2: Training Progress Visualisation

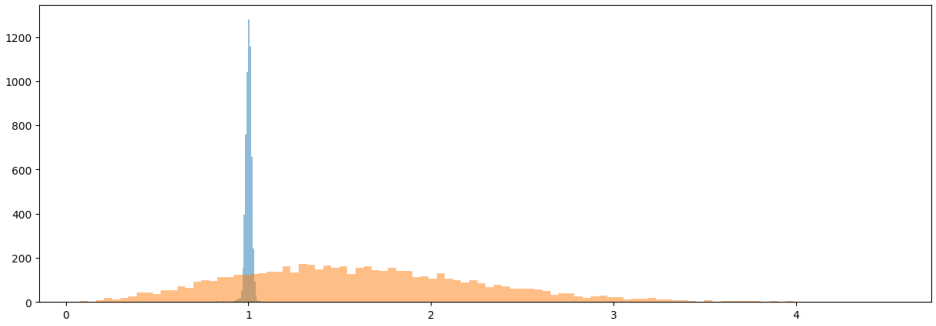


(a) Mini Batch: Loss Graph

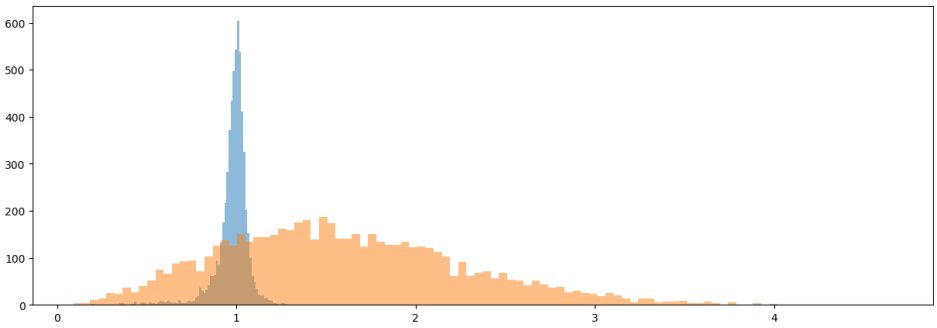


(b) Half Batch: Loss Graph

Fig. 3: Training Data Visualisation



(a) Mini Batch: Vector Graph



(b) Half Batch: Vector Graph

Fig. 4: Histogram Visualisation: Vector lengths pre(orange) and post(blue) train