July 29, 2024

DEMONSTRATION OF A SIMPLE FULLY CONNECTED FEED-FORWARD NEURAL NETWORK (FCNN) MAPPING 3D VECTORS TO A SPHERE

Maria DaRocha

VUW, AIML 425: Neural Networks and Deep Learning

1. INTRODUCTION & THEORY

This discussion of theory and the FCNN demonstration that follows are my own work. Link to source code↗.

A fully connected feed-forward neural network (FCNN) is a type of artificial neural network where each neuron in one layer is connected to every neuron in the subsequent layer. The primary goal of an FCNN is to approximate some function $f^*(x)$ for x input data. For an FCNN, $f^{(i)}$: $R^{d_{i-1}} \rightarrow R^{d_i}$ maps each layer $(i-1)$ into layer i and $f^{(i)}$: $z_{i-1} \mapsto z_i$ for latent (intermediary) variable z. Each layer within the network can be expressed as: $f^i(z^{i+1}) = g(Wz^{i-1} + b^i)$, where g is a scalar nonlinearity (activation function) operating on an inner product between a weight matrix (W) and the vector output of the previous layer (z^{i-1}) plus an offset (bias, b^i). Parameters W and b define the relationship between the input and output of a layer. Further, all elements of W and b must be learned by the network from examples (i.e., training). The nonlinearity operator g facilitates matching of a nonlinear f^*; its selection is often based on empirical results. One common choice for g is the Rectified Linear Unit (ReLU): $g(W_z) = max\ (0, W_z)$. This structure allows the network to learn complex (non-linear) representations of data by transforming inputs through multiple layers of interconnected neurons. While "depth" (i.e., networks with two or more hidden layers) facilitates learning, it is not necessary for universal approximation – as demonstrated by the example that follows.

Supervised learning (for regression) consists of learning to predict an output $y \in R^{d_1}$ given an $x \in R^{d_2}$. To this end, the objective of training is to find the best $f(x;\theta)$ that best fits the observations $D = \{(x^{(n)}, y^{(n)})\}_{n \epsilon A}$, where n is an observation belonging to a set of indices A. To find the optimal θ by deterministic reasoning, we minimize a distance (average) where the distance between the *desired* output and the *actual* input are independent. That is, $\theta^* = argmin_\theta \mathcal{L}(\theta)$ can be found by obtaining the minimum mean square error over the database, as expressed by the loss $\mathcal{L}(\theta) = \sum_{n \epsilon A} ||y^{(n)} - f(x^{(n)};\theta)||^2$. Computing the 2-norm between the desired output and the output of the neural network as a function of theta summed over all entries in the database results in a minimized loss and optimized theta.

In many circumstances, it is preferable to use probabilistic reasoning to resolve optimal theta. For this application, the data has the probability density $y \in Y$ is Gaussian (multi-variate normal). This means that the probabilistic method of maximizing the log likelihood of the observation of outcome y is equivalent to deriving the minimum mean square error.

In the example that follows, the probability density for random variable Y is normally distributed (i.e., $Y \sim N(\mu, \sum)$, given $\Phi = \{\mu, \sum\}$). More precisely, data are vectors with a covariance matrix $\sum = \delta^2 I$ and all noise elements are independent (i.e., a diagonal matrix). Then $p(y|\Phi) = c * \exp\ (-\frac{1}{2}(y-\mu)^T \Sigma^{-1}(y-\mu))$. Given the covariance matrix is known, this resolves to $p(y|\Phi) = c * \exp\ (-\frac{1}{2\delta^2}||y-\mu||^2)$. In this application, the log likelihood of μ for a single observation y is expressed by the equation $LL(\mu|y) = p(y|\{\mu\Sigma\}) = constant - \frac{\delta^2}{2}||y-\mu||^2$. We aim to maximize μ to make the computation more negative, and so $\mu^* = argmax_\mu LL(\mu|data) = argmin_\mu \sum_{n=1}^{N} ||y^{(n)} - \mu||^2$. Thus, for the demonstration of an FCNN that follows and under these conditions generally, the probabilistic derivation of the maximum likelihood is equivalent to the deterministic minimization of the mean square error.

Supervised training of theta for regression aims to maximize the log likelihood (i.e., minimizes the loss function). In doing so, we use the gradient to walk the loss function down-slope by applying differentiation across batches (subsets) from our sample to improve training (learning) performance and computation speed. The general expression for gradient differentiation at each point is $\theta^{k+1} = \theta^k + \varepsilon \nabla_\theta LL(\theta|B^k)$, where the learning rate epsilon (ε) is often scheduled by an optimizer (e.g., Adam) and B^k is minibatch k for the k^{th} random subset of database *D*. Theta is updated after each minibatch and stochastic (probabilistic) descent is often the preferred approach for descent computation and training, as it introduces random noise to reduces the likeliness of getting stuck in a local minima.

The best training method, as demonstrated in the example that follows, requires that the training database be split into training data (proper) and validation data to ensure over-fitting doesn't occur and to facilitate early stopping.

With these theoretical foundations [1], we will proceed to observe an application of an FCNN that addresses a "real" problem.

2. METHOD & EXPERIMENT

The feed-forward FCNN in question was trained to receive a three-dimensional vector of normally distributed data as input and displace (map) said vector onto the surface of a sphere.

A visualization of sample input data (i.e., before training) can be seen in **Figure A** (*Appendix*) and the output that we aim to have the network "learn" can be seen in **Figure B** (*Appendix*). The architecture of the neural network was specified by the problem as having four fully-connected layers that are 3 (input), 20, 20 and 3 (output) wide, requiring a nonlinear activation for all but the last layer (as expected for a regression problem) (see **Figure C**, *Appendix*).

First, training data was generated according to the problem specification. That is, data batches were generated to produce a number of samples in which each sample represented a 3D vector drawn from a normal distribution. To project the vector onto a unit sphere, it was necessary to calculate the magnitude of the vector using extended Pythagorean theorem then divide each component of the vector by its respective magnitude (**Figure D**, *Appendix*).

The FCNN itself was defined using Python Jax (Flax) libraries for their performant parallel processing.

The model weights were initialized using Xavier/Glorot initialization drawn from a distribution with a mean of 0 and a variance of $Var(W) = \frac{2}{n_{in}+n_{out}}$ to help maintain the variance of the activations throughout the layers of the network. Using this method for activation initialization helped to avoid explosion or vanishing gradients [2].

Network training was performed using mini-batch gradient descent, where the dataset was divided into smaller batches to update the model parameters iteratively. The optimizer used for training was the Adam optimizer, which combines the benefits of both Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). Adam optimization adapted the learning rate for each parameter, leading to more efficient training and faster convergence [3].

For each epoch, the parameters were updated by calculating the gradient of the loss function with respect to the current parameters. This process involved the following steps:

1. **Forward Pass:** The input data batch was passed through the network to obtain predictions.
2. **Loss Computation:** The mean squared error between the predictions and the target values was computed.
3. **Backpropagation:** The gradients of the loss function with respect to the network parameters were calculated using automatic differentiation.
4. **Parameter Update:** The optimizer applied the gradients to update the parameters.

K-fold cross validation was used to gain insight into how well the model generalized to unseen data. This technique divided the dataset into k subsets, or "folds." For each iteration, one fold was reserved as validation data while the remaining $(k-1)$ folds were used to train the model. The validation loss was calculated after each epoch by evaluating the network on this separate (validation) data. The use of a validation set helped to monitor the model's performance and detect overfitting, while the k-fold approach to cross validation ensured that the data split didn't disproportionately favor one section of the dataset. While the k-fold approach was not entirely necessary, given the synthetic nature of the data, it was still applied as a form of best-practice. The training and validation loss histories were plotted to visualize the learning and convergence of the model (see **Figure E** & **Table 1**, *Appendix)*.

Finally, the model's performance was evaluated on the test dataset. The same steps used during training were applied, but without updating the model parameters. The test loss was computed to provide a quantitative measure of the model's accuracy and the test predictions were visualized to qualitatively assess the performance of the network's projection of input vectors onto the unit sphere (**Figure F**, Appendix).

3. RESULTS

The results showed that the network successfully learned to displace the 3D projections onto the unit sphere, as evidenced by the low test loss and alignment of predicted values with expected outputs (**Figure E** and **Figure F**, *Appendix*). Further, the model was highly performant (for complete results and summary see **Table 1** & **Table 2**, Appendix).

4. CONCLUSION

This implementation successfully demonstrated training and validation for a simple feed-forward fully connected neural network (FCNN). Xavier/Glorot initialization provided stable weight initialization and the Adam optimizer facilitated efficient parameter updates. K-fold cross-validation supported the model's ability to generalize, thus confirming its ability to learn and accurately map 3D vectors onto the unit sphere. Visualization of these outputs and loss histories validated the network's performance. The final result of the experiment was a model that demonstrated FCNNs' ability to learn complex (nonlinear) transformations.

5. APPENDIX: TABLES & FIGURES

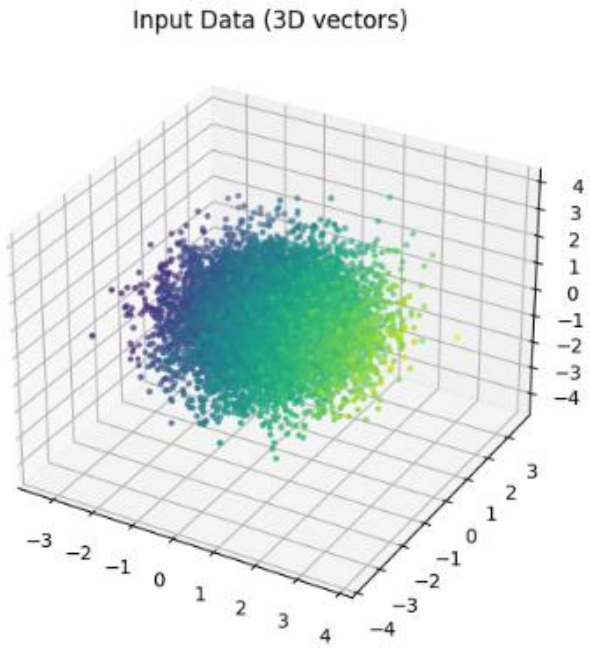

Figure A: Sample Input Data Prior to FCNN Projection

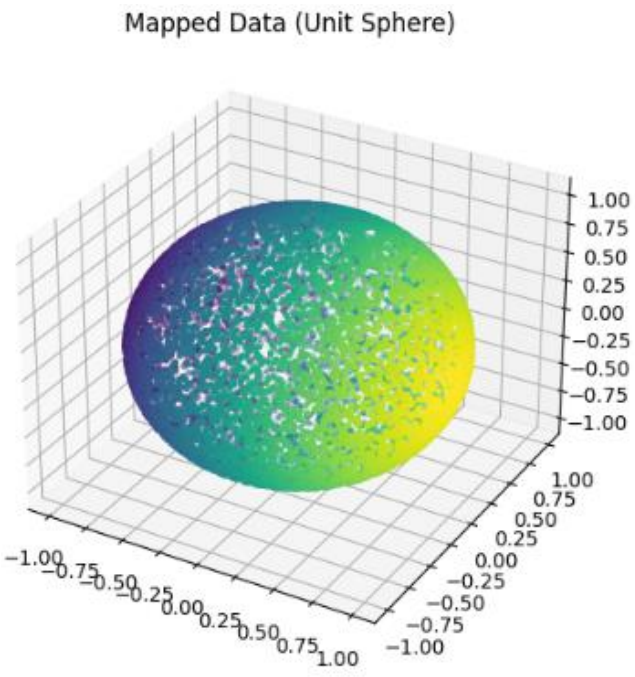

Figure B*:* Example of Desired Output and/or Data Provided for Model Training

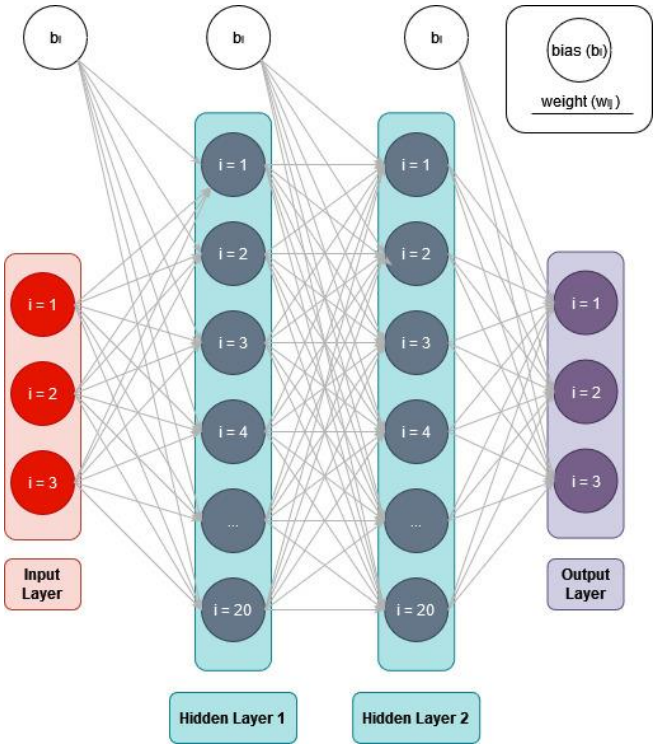

Figure C: Simplified Diagram of FCNN Architecture for 3D Sphere Projection

$$y_i = \frac{x_i}{r_i} = \left(\frac{x_1}{r_i}, \frac{x_2}{r_i}, \frac{x_3}{r_i}\right)$$
$$= \left(\frac{x_1}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_2}{\sqrt{x_1^2 + x_2^2 + x_3^2}}, \frac{x_3}{\sqrt{x_1^2 + x_2^2 + x_3^2}}\right)$$

Figure D: Calculation to Project 3D Vector onto Unit Sphere

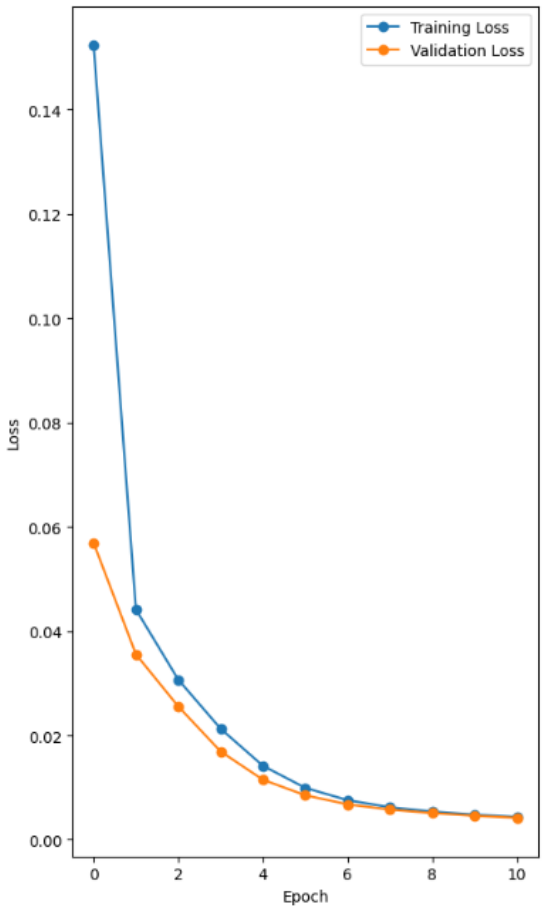

Figure E: Training Loss to Validation Loss Over Epochs

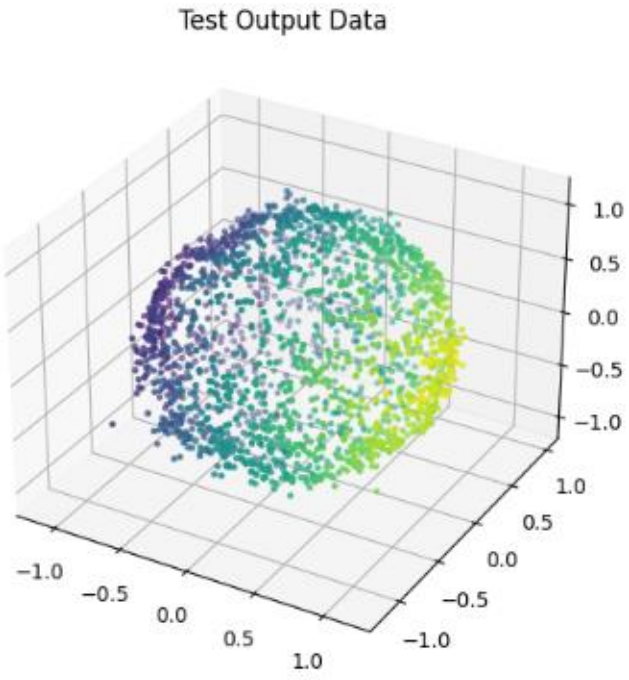

Figure F: Output Projection Produced by the FCNN

Table 1: Training Loss to Validation Loss Over Epochs

Epoch	Batch size	Training Loss	Validation Loss
0	100	0.152	0.056
1	100	0.044	0.036
2	100	0.031	0.026
3	100	0.021	0.017
4	100	0.014	0.011
5	100	0.010	0.009
6	100	0.007	0.007
7	100	0.006	0.006
8	100	0.005	0.005
9	100	0.005	0.005
10	100	0.004	0.004

Table 2: Results Summary Table

Total Samples	Execution Time (seconds)	Final Training Loss	Final Test Loss
10,000	0.68	0.0043	0.00042

6. ORIGINAL CODE

LINK:
THIS CODE IS MY OWN. https://github.com/marianette/a1-aiml425/blob/main/a1-fcnn-unit%20sphere-jaxflax-implementation.ipynb https://colab.research.google.com/drive/1arjiadyu0imk76htvcuinh_alnam-dro?usp=sharing

7. REFERENCES

[1] B. Kleijn, “Basic Deep Learning Regressor & Gradient Descent and Context: AIML425 [Lecture 3 & 4 Notes],” *Master of Artificial Intelligence Year One, Victoria University of Wellington*, Aug. 2024, Accessed: Jul. 29, 2024. [Online]. Available: https://ecs.wgtn.ac.nz/foswiki/pub/Courses/AIML425_2024T2/LectureSchedule/DLintro.pdf

[2] K. Wong, R. Dornberger, and T. Hanne, “An analysis of weight initialization methods in connection with different activation functions for feedforward neural networks,” *Evol Intell*, vol. 17, no. 3, pp. 2081–2089, 2024, doi: 10.1007/s12065-022-00795-y.

[3] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv preprint*, vol. arXiv: 1412, no. 6980, 2014.