ABSTRACT

This project draws on a range of Python libraries (numpy, matplotly, jax) to implement a five layer neural network to project randomly generated data onto the surface of a unit radius sphere. The neural network uses the Adam stochastic optimisation method (derived from adaptive moment estimation) [1], which only requires first-order moments (mean). The normal hyper-parameters are left consistent as the trained neural network is exposed to novel test data across four test conditions to compare performance. Minibatch parameters are varied and show that running more individual training passes results in higher processing times, with only small improvements in training and test loss.

Index Terms— Neural networks, Gaussian, feedforward, stochastic gradient descent, ADAM

1. INTRODUCTION

Neural networks have become a staple machine learning (ML) tool and are subject of much attention [2]. Using a range of modern ML packages and libraries, available across multiple programming languages, designing and implementing neural networks is becoming easier, but it remains important for data scientists to understand what their networks are doing in order to get the most out of that improved functionality. This paper touches briefly on neural network architecture, describing the mathematical basis of a feed-forward network: the neuron, the layer, weights and biases, and the non-linearity function. Both deterministic and probabalistic approaches to training a network are defined and stochastic gradient descent is discussed as a useful method for escaping local minima.

2. NEURAL NETWORK THEORY

Feedforward neural networks are made up of multiple layers of neurons or perceptrons [3] combining to approximate some function f^*, that maps an output y based on some input x.

2.1. Feedforward architecture

Feedforward networks take input values x, perform some function and feed the output of that function through the next layer to form a chain of functions. The implementation used here for example, has an input layer x with three nodes, three hidden layers, z_1, z_2, z_3 each wih 16 nodes, and one output layer y, with three nodes. This network maps x, through a series of functions, to y: $z^1 = f^1(x)$, $z^2 = f^2(z^1)$, $z^3 = f^3(z^2)$, and $y = f^4(z^3)$.

2.2. Representing layers and neurons

Each layer comprises a set of nodes or neurons, so each neuron j within the layer i takes an output from the previous layer, modifies it with learned weights and bias, and applies a non-linear function to arrive at an output value. Each neuron can be represented with the following notation where g is an element-wise, non-linear function (in this implementation, a rectfied linear unit: g(z)=max0,z), W is a matrix (of learned weights) and c_i is a scalar:

The output of each neuron j in layer i can be represented:

$$\begin{aligned} f_j^{(i)}(z^{(i-1)}) &= g(W_{j:}z^{(i-1)} + c_i) \\ &= g(W_{j:}\begin{bmatrix} z^{(i-1)} \\ 1 \end{bmatrix}) \end{aligned}$$

And a layer i can be represented:

$$f^{(i)}(z^{(i-1)}) = \begin{bmatrix} f_1^i(z^{(i-1)}) \\ \vdots \\ f_p^i(z^{(i-1)}) \end{bmatrix}$$

2.3. Minimum Mean Square Error and Maximum Likelihood Estimation

A feedforward neural network assumes that some function $f*$ exists and attempts to find the $f(x;\theta)$ that maps $x \in \mathbb{R}^{d_2}$ to the output $y \in \mathbb{R}^{d_1}$, based on the observed data. We can use Minimum Mean Square Error (MMSE) to determine $f(x;\theta)$, that is θ^* by iterating through the database D:

$$\begin{aligned} D &= (x^{(n)}, y^{(n)})_{n=1,\dots,N} \\ \theta^* &= argmin_\theta \sum_{n \in A} ||y^{(n)} - f(x^{(n)};\theta||^2 \end{aligned}$$

We can also take a probabilistic approach to training the optimal function, rather than choosing a set of parameters $\theta = \mu, \Sigma$, using Maximum Likelihood Estimation (MLE). Most modern neural networks take this approach [@Goodfellow-et-al-2016], which allows for automatically determining a cost function - the negative log-likelihood or cross-entropy between training data and the distribution of the model:

Assuming $Y \overset{\text{iid}}{\sim} N(\mu, \Sigma)$, then $\Sigma = \sigma^2 I$

$$\begin{aligned} p(y|\theta) &= cexp(-\frac{1}{2}(y-\mu)^T\Sigma^{-1}(y-\mu)) \\ &= cexp(-\frac{1}{2}(\frac{1}{\sigma^2}(y-\mu)^T I(y-\mu))) \\ &= cexp(-\frac{1}{2\sigma^2}||(y-\mu)||_2^2) \\ &= cexp(-\frac{1}{2\sigma^2}||y-\mu||_2^2 \\ LL(\mu, data) &= constant - \frac{\sigma^2}{2}\sum_{n=1}^{N}||y(n) - \mu||_2^2 \end{aligned}$$

2.4. Supervised training of θ

Supervised training of θ involves finding θ^* using MLE, where our model is $p(y|x;\theta) = cexp(-\frac{1}{2\sigma^2}||y - f(x;\theta)||^2)$, and we fit f using:

$$
\begin{aligned}
D &= x^{(n)}, y^{(n)})_{n \in A} \\
\theta^* &= argmax_\theta LL(\theta, D) \\
&= argmin_\theta \sum_{n \in A} ||y^{(n)} - f(x^{(n)};\theta)||^2
\end{aligned}
$$

So θ is updated on each pass, stepping away from the direction of the steepest gradient (negative log likelihood estimate) at the learning rate ϵ: $\theta^{k+1} = \theta^k + \epsilon \nabla_\theta LL(\theta, D)$. This gradient descent function computes over the entire database and is dependent on the learning rate, too high a rate risks overshooting the minima, too low and the function may never reach the minima [4].

2.5. ADAM: Adaptive moment estimation and stochastic gradient descent

Stochastic gradient descent introduces randomness to the gradient descent process and brings the benefit of breaking out of local minima and improving the peformance of the network by calculating the gradient of a sample of the data rather than of the entire data set [5]. In this implementation, the neural network is optimised using the Adaptaive moment estimation (ADAM) method, "an algorithm for first-order gradient-based optimisation of stochastic objective functions" [1].

2.6. Comparing minibatch parameters

This paper implements a neural network with x, z_1, z_2, z_3, y layers and $(3, 16, 16, 16, 3)$ nodes, using the adam optimizer. The experiment trains a neural network to project input data X onto the surface of a unit-radius sphere and then uses trained parameters to project novel test data onto a sphere. The learning rate is constant throughout, and four different minibatch paramaters are compared: baseline, more batches, more epochs, and bigger batches.

3. RESULTS

Implementing a neural network with dimensions [3, 16, 16, 16, 3] and using the adam optimiser produces the results in Fig. 1. In the baseline neural network the minibatch parameters batchsize = 50, batchno = 1001, epochs=1, resulting in processing time of 70 seconds, training loss of 0.0014 and test loss 0.002. When using bigger batches (batchsize = 5000), processing time reduces to 47s, training and test loss both improve to 0.0013. When using more batches (batchno = 5001), processing time increases to 196s with improved training and test loss of 0.001. Finally, when running more epochs (epoch = 10) processing time grows to 583s, bringing the best performance with training loss at 0.0005 and test loss at 0.0008.

index	Obs	batch	epoch	total batches
baseline	50	1001	1	1000
bigger batches	5000	1001	1	1000
more batches	50	5001	1	5000
more epochs	50	1001	10	10009

(a) minibatch parameters

index	time	trng loss	test loss
baseline	70.0	0.0014	0.002
bigger batches	47.0	0.0013	0.0013
more batches	196.0	0.001	0.001
more epochs	583.0	0.0005	0.0008

(b) minibatch performance results

Fig. 1. Minibatch parameters and performance results

4. CONCLUSION

The implemented neural network showed fairly consistent performance for a [3, 16, 16, 16, 3] network, with varying batch parameters. Although loss was lowest with more training cycles (loss ¡0.001, with 10,000 batches), comparable performance was achieved (with lower processing times) when using fewer total batches even when batch size was increased by an order of magnitude. It is unclear whether the ADAM optimizer was any more or less effective than other optimizers, so future experiments could look at running comparisons between optimizers on the same datasets.

The implementation of the network itself was somewhat constrained by task requirements (in terms of network architecture and initial hyperparameters) but that also provided greater certainty and comparability of experiment results. With a five layer neural network it would be interesting to review what effect changes in the learning rate would have - meaning changes by orders of magnitude because small incremental changes to the learning rate are likely to vanish over a deep, wide net. It would also be interesting to see what effect adding or removing layers/ nodes will have on the network's performance, especially considering the principles driving the ADAM optimizer - i.e. is there a lower limit to the network architecture where this optimizer's performance (computation vs loss ratio) converges with that of a classic gradient descent model.

5. REFERENCES

[1] D.P. Kingma and J.L. Ba, "Adam: A method for stochastic optimization," in *Published as a conference paper at ICLR 2015*. ICLR, 2015, pp. 1–15.

[2] Wojciech Samek, Gregoire Montavon, Sebastian Lapuschkin, Christopher J Anders, and Klaus-Robert Muller, "Explaining deep neural networks and beyond: A review of methods and applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press, 2016, http://www.deeplearningbook.org.

[4] Leon Bottou and Olivier Bousquet, "The tradeoffs of large scale learning," *Advances in neural information processing systems*, vol. 20, 2007.

[5] Leon Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*, pp. 421–436. Springer, 2012.

link to google colab for code in commented notebook. To run the full code set takes about 15-20 minutes.