Quadratic Neural Network Architecture

As Evaluated Relative to Linear Neural Network Architecture

Reid Taylor, Ben Moses, JJ Satti

December 6, 2021

1 Introduction

Neural Networks of a traditional structure are proven to be capable of approximating omnipotence of arbitrary nonlinearity [1]. Traditional structure here implies that layers perform "linear" operations on input data vectors by taking the inner product of said input vector and a weight matrix, and then summing the result with a bias vector. The resulting vector is then activated through a nonlinear function as chosen by the scientist in use. This definition on the j^{th} layer is shown in figure 1 [3].

$$h_j = \sigma_j(W_j \cdot \vec{x}_{j-1} + \vec{b}_j)$$

Figure 1: The definition of an arbitrary layer in a traditional neural network

From this example, note that the activation function, σ, the weight matrix, W, and the bias vector, $\vec{b}$, are all unique to the layer in which they are applied. See also the input vector, $\vec{x}$, is denoted to mean the output of the previous layer. In the case j=1, the input vector is meant as the input data to the network itself.

Many evolutions of this structure exist to fit specific needs.

For example, convolutional neural networks are feed forward neural networks with at least one layer where the weight matrix and input vector inner product is instead given by a convolution. In addition to the requirement that the inner product is given by a convolution of functions, the weight matrix is also a sparse matrix, often of a tridiagonal structure, such that the weight matrix is nearly reduced to a structure of an eigenvalue decomposition. Generally, these neural networks also feature the same weight matrix used in each layer, restricting the number of parameters dramatically and reducing computational costs proportionally.

Furthermore, recurrent neural networks exist as an evolution of neural networks such that both the weight matrix and the bias vector are shared across all layers of the neural network, and optionally with the ability to implement new features at any given layer of the network, and to dually draw conclusions from any layer of the network. This allows for a great deal of flexibility in the neural network itself, and requires very few parameters as compared to "vanilla" neural networks. However, eigenvalue decomposition is experienced within recurrent neural networks of numerous layers.

As they exist today, multiple neural network architectures exist to meet certain requirements, though each has certain disadvantages which must be weighed when determining what architecture would best serve a particular need.

This paper introduces a novel form of neural network architecture in which a new term is added to the definition of each layer, taking advantage of higher ordered tensors to provide interactions among the values of the input data vector. The new definition is given in figure 2 [3].

$$h_j = \sigma_j(\vec{x}_{j-1} \cdot V_j \cdot \vec{x}_{j-1} + W_j \cdot \vec{x}_{j-1} + \vec{b}_j)$$

Figure 2: The definition of an arbitrary layer in a quadratic neural network

2 Implementation

In order to construct a novel neural network structure, it is necessary to do so without using existing architecture to ensure complete control over the interactions within and between layers. As such, this research is completed in Python with the use only of the NumPy package for use in defining algebraic structures and operations. Only the fundamentals needed from the package are used, the rest is developed as needed.

An object oriented programming approach is used for best practice to allow generality in implementation and for possible reuse in future cases. The neural network is developed with distinct layer objects amassed into a network object, each of these object (and all contained classes therein) with different properties and functions.

2.1 A Granular Examination of Layers

2.1.1 The Layer Archetype

The neural network's construction is instantiated at the most base level: the layer. The layer is implemented as an extension of a "Layer" object, imbued with input and output properties, a method for forward propagation and one for backward propagation. All layers contain these properties and methods, while each builds upon them uniquely as required by the network and as supported by traditional neural network architecture.

The input and the output properties are type set to be vectors, or, more specifically, NumPy arrays with dimensions 1 by n.

The forward propagation method is defined to take as its argument a vector to conform to the parent class input property mentioned above.

The backward propagation method takes as arguments the output error value and a learning rate. The output error value is a vector of equal dimensions to the layer's output vector. This vector details the magnitude by which each of the layer's respective nodes varies from the "true" value, the value which would contribute to a perfectly accurate prediction. The learning rate is a scalar commonly restricted to be an element of [0, 1) which scales the amount by which individual corrections are applied to each node within the network.

2.1.2 The Fully Connected Layer

The first child class of the Layer Archetype is a fully connected layer. This layer has a weights and a bias property, and the inherited forward propagation and

backward propagation methods. The weights and bias properties represent the nodes unique interactions to data, as these words imply in the context of data science and machine learning.

The weights property is a matrix of dimensions n by m, where n represents the dimension of the data used as input to the fully connected layer, and m represents the dimension of the data destined as output from the fully connected layer. This property serves as the matrix W in 1 and in 2; however, it is important to note that this layer's structure is modeled by 1 exclusively, as there are no quadratic interactions present in this fully connected layer. At instantiation, the weight matrix is constructed of appropriate dimensions with each entry a random float value selected from the interval [-0.5, 0.5)

The bias property is a vector of dimension m, with m likewise representing the output vector dimensions. At instantiation, the bias vector is constructed of the appropriate dimensions as a row vector with each entry set to zero.

The inherited forward propagation function is designed to calculate the layer's output vector. The calculation is as shown in figure 3.

$$(W_j \cdot \vec{x}_{j-1} + \vec{b}_j)$$

Figure 3: The forward propagation calculation for a fully connected layer's output.

The inherited backward propagation function is designed to calculate by what magnitude each of the layer's contained nodes should adjust in response to the cost function of the neural network to be discussed in **??**. The calculation is as shown in figures 4 and 5, for the weights and bias adjustments, respectively.

$$W_{t+1} = W_t - lr \cdot \vec{x}^T \cdot y$$

Figure 4: The backward propagation calculation for a fully connected layer's weights. W_{t+1} represents the new weight matrix based on the backward propagation, while W_t represents the weight matrix at the time of calculation of the layer's output. lr denotes the learning rate property defined as a hyper-parameter in 2.1.1. $\vec{x}^T$ denotes the transpose of the layer's input vector. y denotes the error calculated to originate from the layer's output.

$$W_j \cdot \vec{x}_{j-1} + \vec{b}_j$$

Figure 5: The backward propagation calculation for a fully connected layer's bias.

Observe that a fully connected layer of input dimension x and output dimension y contains at most $(x \cdot y + y)$ unique parameters.

2.1.3 The Quadratic Layer

The next child class of the Layer Archetype is the newly designed quadratic layer. This layer has the standard properties as defined for the fully connected layer, of congruent dimensions and instantiation as the fully connected layer denotes. In addition to these, however, the quadratic layer additionally has a 3-Dimensional tensor denoted as V, the quadratic weights. This quadratic layer inherits the forward and backward propagation functions from the Layer Archetype.

The tensor V is defined as of dimensions n by m by n, with n and m still representing the dimensions of the layer's input and output, respectively. At instantiation, this tensor is filled with values still from the interval [-0.5,0.5).

The forward propagation function for a quadratic layer takes a different form than that of the fully connected layer in accommodating the tensor V, though the essence of the calculation is preserved and is as shown in figure 6.

$$\vec{x}_{j-1} \cdot V_j \cdot \vec{x}_{j-1} + W_j \cdot \vec{x}_{j-1} + \vec{b}_j$$

Figure 6: The forward propagation calculation for a quadratic layer.

The backward propagation function for a quadratic layer inherits from the fully connected layer the calculations for the weight matrix's adjustments and the bias vector's adjustments. The backward propagation function introduces a quadratic weight error calculation, as shown in figure 7.

$$V_{t+1} = V_t - lr \cdot \vec{x}^T \cdot y \cdot \vec{x}$$

Figure 7: The backward propagation calculation for a quadratic layer's weights. V_{t+1} represents the new quadratic weight tensor based on the backward propagation, while V_t represents the weight tensor at the time of calculation of the layer's output. lr denotes the learning rate property defined as a hyper-parameter in 2.1.1. $\vec{x}^T$ denotes the transpose of the layer's input vector. y denotes the error calculated to originate from the layer's output.

Observe here that a quadratic layer of input dimension x and output dimension y contains at most $(x \cdot y \cdot x + x \cdot y + y)$ unique parameters. That is, this layer contains precisely $(x \cdot y \cdot x)$ more parameters than a fully connected layer does.

2.1.4 The Activation Layer

The last child class of the Layer Archetype is the activation layer. This layer is unique in that it does not have individual weights nor biases, but instead has an activation and an activation prime property, each of which is represented as a function. This class also has a forward and backward propagation function, though these too are unique from the other layers discussed. In some sense, it would be fair to state that the activation layer is an extension of the layer that immediately preceded it in that the activation functions it represents are applied directly to the output of the either fully connected or quadratic layer which came before, and the neural network definition of a layer given in figures 1 and 2 is not complete until the activation layer is compounded upon the layer before it.

The activation function is to be chosen from an available list of defined functions, including the standard activation functions used in other libraries of machine learning: *tanh*, *ReLU*, *Sigmoid*, and the *identity* functions. The activation prime function is simply the respective gradient of each of these activation functions.

The forward propagation function for the activation layer applies the chosen activation function to the layer's input vector in an element-wise operation. The exact implementation of this varies from each type of activation function, though it is represented in a generalized sense in figures 1 and 2, with respect to the class of layer which preceded this activation layer.

Likewise, the backward propagation function applies the derivative of the activation function to the input of the activation layer, and then scales this result by the layer's output's error, which is a scalar in this context.

2.2 A Granular Examination of the Network

The network object is constructed with multiple properties and functions intrinsic to it. Among these properties are the layers, the loss function and the associated derivative of the loss function. The functions include those necessary to add layers to the neural network, to set the loss functions of the network, and to train the network, as well as to use the network to predict values given input.

The layers are implementations of the layers described above, and therefore will be discussed no further in the context of the network. The loss function is open to customization by the scientist; however, for the experiments performed, the standard mean squared error loss function is implemented.

2.2.1 The Training Function

The training function takes as input the training data along with the true outcomes of that data. In addition, it takes in the number of epochs to run over and the learning rate to implement. For each epoch, every sample of the input data is fed through the network through each sequential layer's forward propagation method, and then the final output of this process is used in the calculation of the loss function. The value returned by the loss function is used in calculating the error of the model, and by instead returning a value from the derivative of the loss function we gain insight into what changes to the model parameters must be implemented through the backward propagation method. This process is completed and the model's parameters are corrected before the next epoch is run. The error recorded is saved and then reported from each training epoch to provide relative time series data as to the performance of the model's backpropagation method.

2.2.2 The Prediction Function

The prediction function takes as input the value to predict for. This function does not evaluate, nor alter, the parameters of the model as input data is fed through the network. Only a value representing the output of the model is returned.

3 Results

In order to properly assess the neural network structure in comparison to a traditional network, both created models were limited to a shallow network design, only with an input layer and a fully connected layer following. The quadratic structure is only applied to the latter.

Both networks are limited to identical hyper-parameters, including: the number of layers, the number of nodes within each layer, the learning rate, and the activation functions in each respective layer, as well as the number of epochs in use during the training stage. Beyond this, experimental control also requires that the same data is used to train and score the networks.

In adherence to the requirements of this research, the networks are evaluated in response to each of the following:

- A constant function
- A simple linear function, the identity function
- A simple quadratic function
- A simple exponential function
- A simple sinusoidal function

In each of these, the keyword "simple" indicates that the function takes the standard form, without any translations, dilations, or rotations. With respect to the above list, the explicit functions for testing are as follows:

- $f(x) = 1$
- $f(x) = x$
- $f(x) = x^2$
- $f(x) = e^x$
- $f(x) = \sin(x)$

In evaluating the results of the study, proper experimental design requires appropriate criteria by which to determine an objectively superior model architecture: each model, relative to the other, will be evaluated on three distinct criteria:

- The accuracy of each neural network model after training is complete

- The number of significant nonzero parameters with respect to all parameters in each respective model
- The distribution of all parameters, with evaluations on relevant measures of variability

In the order with which each function to be tested was given, the following are the experimental results for both neural network structures for each tested function.

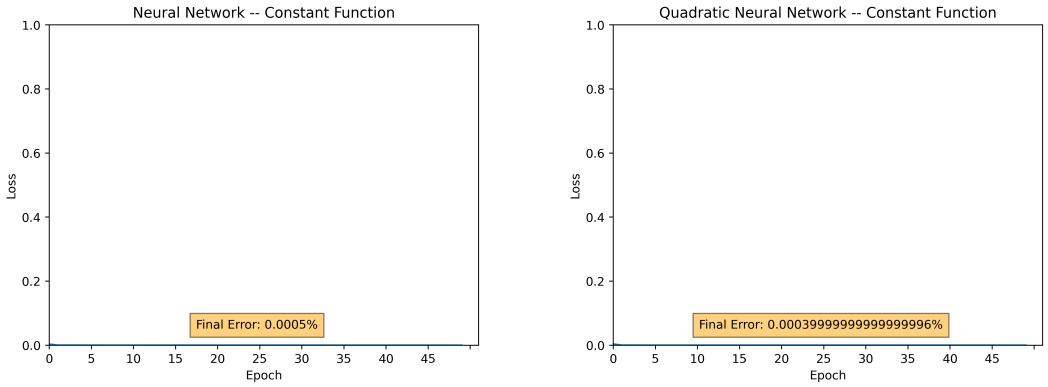

Figure 8: Experimental results displaying the loss of each neural network structure as a function of the epoch of training after which it was scored; this graph corresponds to the constant function tested.

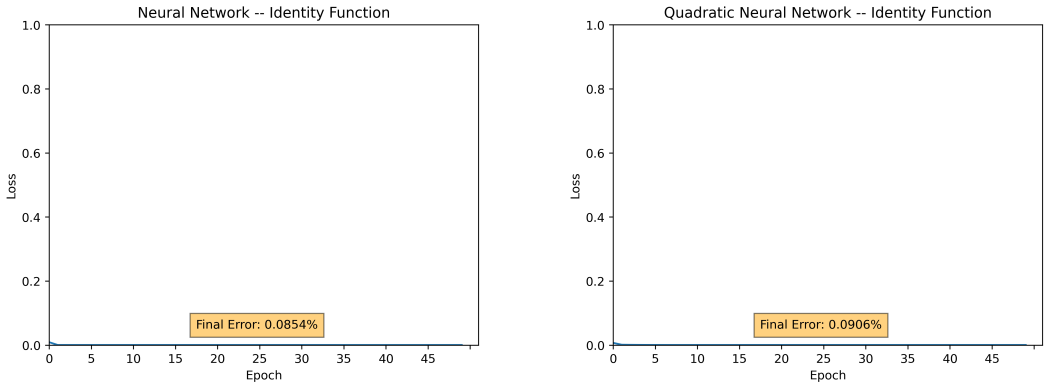

Figure 9: Experimental results displaying the loss of each neural network structure as a function of the epoch of training after which it was scored; this graph corresponds to the identity function tested.

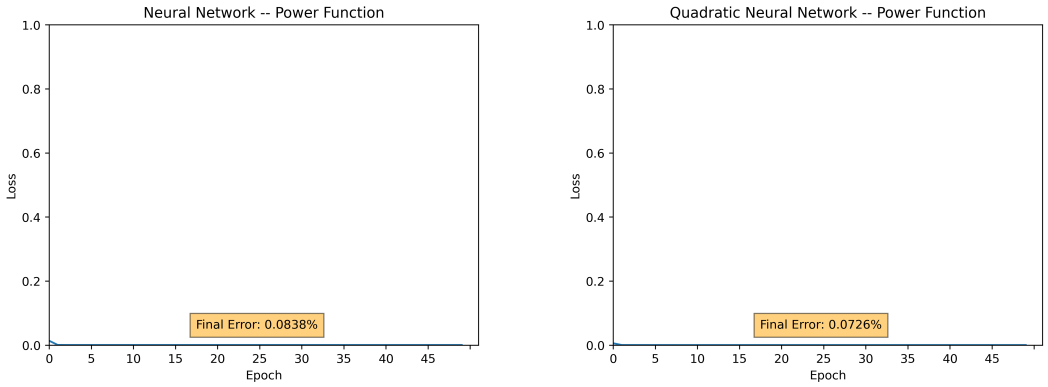

Figure 10: Experimental results displaying the loss of each neural network structure as a function of the epoch of training after which it was scored; this graph corresponds to the power function tested.

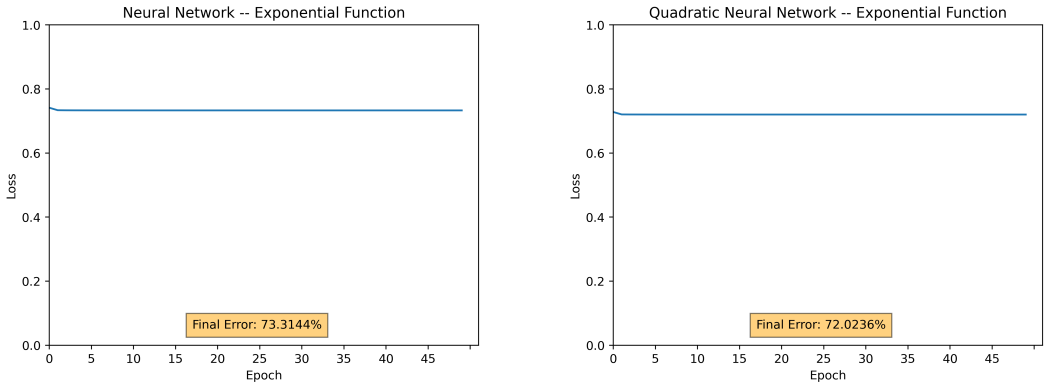

Figure 11: Experimental results displaying the loss of each neural network structure as a function of the epoch of training after which it was scored; this graph corresponds to the exponential function tested.

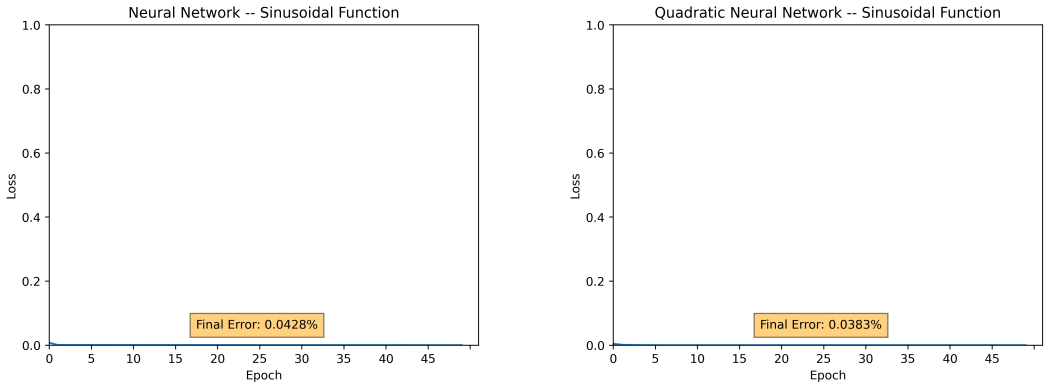

Figure 12: Experimental results displaying the loss of each neural network structure as a function of the epoch of training after which it was scored; this graph corresponds to the sinusoidal function tested.

Observe that the final error, and much of the error through the entire training stage, is nearly equal between neural network structure for all functions tested. Additionally, all functions tested by each model exhibit the model's error rate converging to zero for all cases, except for the exponential function. This function converges to approximately 72% error, as shown, and repeatedly does so in multiple tests. While this value nears the inverse value of the complex golden ratio raised to the power of $\frac{2}{\pi}$, this seems to be more likely due to chance, and not a true association between the exponential function and a complex gold ratio mathematical constant. More likely is the hypothesis that this is due to the limitations of the neural network structures and the associated activation functions in predicting functions of exponential growth.

Indeed, in all cases it is observed that model performance did not significantly differ between traditional neural network architecture and the posited quadratic neural network structure. Note that, despite equal performance, the quadratic model suffers from significantly more parameters required, and more computation through the processes of forward and backward propagation as a result.

These results prove that equal performance is achievable, yet there exist significant differences in performance. As these models currently stand, these "vanilla" iterations of them favor the traditional neural network structure over quadratic neural network structure for computational efficiency.

4 Conclusion

It is important to note that, the results of this study aside, there are multiple factors which influence the efficiency of each model tested. For one, these models are vanilla in design, in that only the most fundamental operations are used to build the model, while neglecting many possible avenues of further growth. Many of the tools and methods of implementation of neural networks were not made available in this experimental design; these include, but are not limited to:

- Randomized dropout of neural connections
- Wide scale model pooling
- Convolutional or Recurrent structuring
- Custom loss functions
- LSTM model style control gates
- Back propagation through calculations other than steepest descent gradient approach
- Data batching and mini batching
- Other, possibly more appropriate, activation functions
- Adaptive learning rates

Exploration into these avenues of model refinement, and others, may prove fruitful in attaining even greater accuracy in the quadratic neural network architecture as given in this paper, and may even provide a way to reduce the number of parameters necessary to achieve such results, thereby favoring quadratic neural network architecture for problems of regression.

As technology advances further and further, we encroach upon the horizon of our current knowledge and capabilities to understand the knowledge we have access to. Quadratic neural network architecture is a promising way to better interpret and identify trends of high dimension data, and will certainly become a valuable tool for data science and machine learning in the coming years as its use is better researched, understood, and implemented in common avenues of the industry.

References

[1] A.N. Gorban, D.C. Wunsch (1998) *The General Approximation Theorem*, IEEE, DOI: 10.1109/IJCNN.1998.685957

[2] F. Fan, W. Cong, G. Wang (2017) *Generalized Backpropagation Algorithm for Training Second-Order Neural Networks*, International Journal for Numerical Methods in Biomedical Engineering, DOI: 10.1002/CNM.2956

[3] P. Mantini, S. Shah (2020) *CQNN: Convolutional Quadratic Neural Networks*, Research Gate, DOI: 10.13140/RG.2.2.35842.71367