BAYLOR UNIVERSITY

Eigenvalue Prediction from Convolutional Neural Networks

Travis Whyte

December 15, 2019

Abstract

In this work, we employ two different CNN architectures in order to predict eigenvalues of two classes of real symmetric matrices. It is found that the models are able to predict eigenvalues, but to very low numerical precision. We discuss previous work used to attain similar goals, as well as the results of training and validating the model. Also discussed is potential extensions and applications of this work to other fields.

Key words: Deep Learning, Convolutional Neural Networks, Eigenvalues, Real symmetric matrices

1 INTRODUCTION

Eigenvalues play a very important role in many areas of science and engineering. For example, in Lattice Quantum Chromodynamics (LQCD), the eigenvalues of the Dirac Operator can be used for noise reduction techniques in disconnected loop trace estimations[1]. While Krylov subspace methods are readily available for the computation of eigenvalues, it can sometimes be prohibitive to calculate them if the matrix is ill conditioned or very large. Faster methods are needed in order to accurately calculate the spectra of ill conditioned and very large matrices.

Deep Neural Networks (DNNs) have become the go to resource for many areas of science and engineering, such as medical image segmentation[8], calculating combinatorial atomic sets for compound analysis [7], and even mastering the game of Go [10]. Given the success of DNNs for these complex tasks, it is natural to ask if DNNs can be used for the prediction of eigenspectra from only the operators alone. Recurrent Neural Networks have been used in a limited capacity to compute eigenvalues and eigenvectors, but usually only for a small part of the spectra (See [13] where the smallest and largest eigenvalues of a real symmetric matrix are computed). Only recently have Convolutional Neural Networks (CNNs) been used to compute eigenvalues. In [3], a CNN was used to calculate the eigenvalues corresponding to phonons within a lattice, and in [4], the ten lowest lying eigenvalues of the Coulomb matrix were calculated. However, at the time of this writing, there are no indications in the literature that a DNN has been used to calculate the full spectra of a general matrix. In this work, we attempt to calculate the full spectra of real symmetric matrices using a CNN with limited success.

The layout of this work is thus: In section 2, we outline the methodology of this study, including the network architecture. In section 3, we present the results for the prediction of eigenvalues. In section 4, we present the summary of this work, potential improvements to the model, and the applicability of this work to other fields. In section 5, we discuss future work the author wishes to explore. The appendix contains information on failed attempts at using CNN to replicate the matrix inverse.

2 METHODOLOGY

Due to the success of CNNs with pattern recognition, a CNN was chosen to observe if the network could be trained to observe patterns within matrices that lead to accurate prediction of eigenvalues. The architecture was created with Keras 2.2.4 [11] using Python 3.7. In this study, two test cases were chosen: random uniform diagonal matrices and random real symmetric tridiagonal matrices. The diagonal components of the latter were chosen such that the spectra remained positive definite. Initial testing was performed on real symmetric matrices without a general structure, however the CNN was not able to predict eigenvalues in this case. It is believed that since these types of matrices are lacking a distinct structure, the CNN could not recognize a pattern. In both cases, the matrices are of size $16x16$, however the

architecture can admit matrices of all sizes greater than the convolution size. In the course of this study, it was found that no one architecture was amenable to both test cases, so different architectures were formed for each case. Both architectures loosely follow that of reference [3].

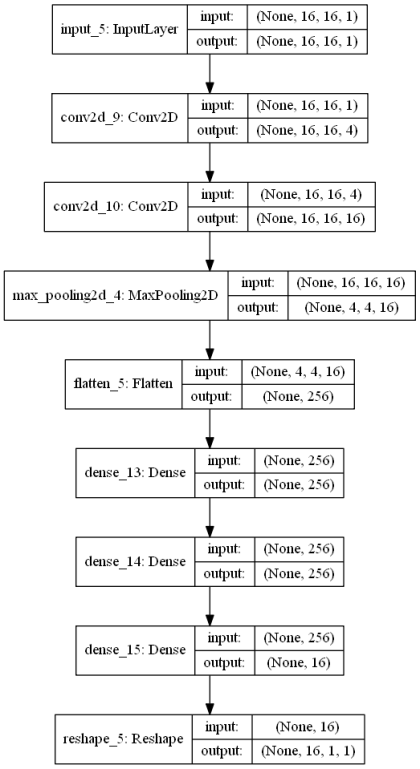


Figure 2.1: The created architecture for the real diagonal test case

2.1 Commonalities Between Test Cases

In both cases, the CNNs were trained with 500 training samples, using a 80/20 validation split. The small number of training samples was used to keep the training time to a reasonable amount. The optimizer employed was the Adam optimizer[6] with a learning rate of 10^{-3}. A custom loss function using the Keras backend was employed for this study, which calculated the error between the predicted eigenvalues and the true eigenvalues. Explicitly, the loss function is given by:

$$loss = ||\theta_{true} - \theta_{pred}||_2$$

Where θ_{true} is the tensor containing the true eigenvalues and θ_{pred} is the tensor containing the predicted eigenvalues. Additionally, the accuracy metric is given by *1-loss*.

2.2 Diagonal Test Case

The diagonal test case architecture is illustrated in Figure 2.1. The input to the CNN is the matrix, where it enters two convolutional layers of size $2x2$, followed by a MaxPooling layer, also of size $2x2$. The layer is then flattened, and three dense layers with ReLu activation functions follow. The output is a $16x1$ tensor containing the predicted eigenvalues.

2.3 Real Symmetric Tridiagonal Test Case

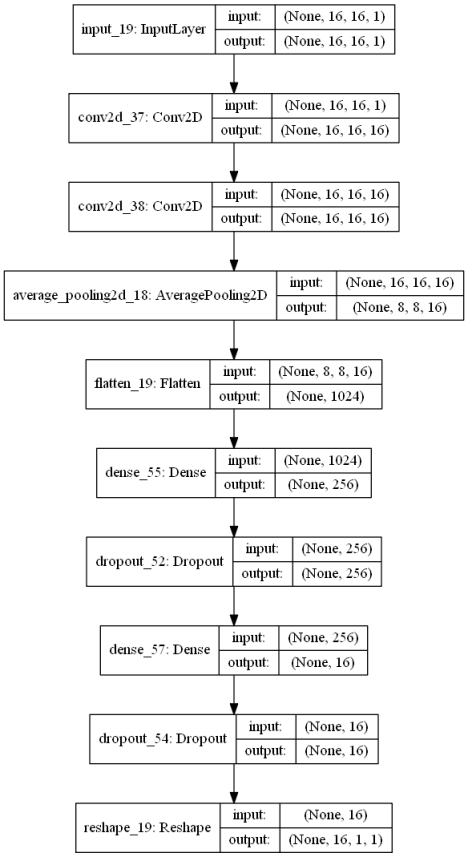


Figure 2.2: The created architecture for the real symmetric tridiagonal case

The architecture for the real symmetric tridiagonal test case is illustrated in Figure 2.2. This architecture follows the same basic structure as the diagonal test case with minor differences. It was observed throughout the course of this study that employing the same architecture for matrices having off diagonal components resulted in very slow training and large validation accuracy in comparison to training accuracy. In order to reduce the time to train and eliminate over training, dropout was implemented between layers, and L2 Regularization was employed on the convolutional layers, with $\lambda = 10^{-5}$, along with ReLu activation in the convolutional layers. It was also observed that Average Pooling in place of Max Pooling reduced

the loss per epoch. In addition, it was found that using ReLu activation functions in the last three dense layers did not result in an appropriately minimizing loss function, so linear activation was used.

3 Results

3.1 Diagonal Case

The CNN was able to able to predict the eigenvalues of a randomly generated uniform diagonal matrix to an accuracy upwards of 90%. Figure 3.1 (Left) displays the training loss and validation loss as a function of the number of training epochs. We see that the losses are well superimposed upon one another, and no over training is present. The loss approaches approximately 10% after 50 epochs. Figure 3.1 (Right) shows the validation loss and validation accuracy as a function of the number of epochs. After 50 epochs, the validation accuracy has reach approximately 90%.

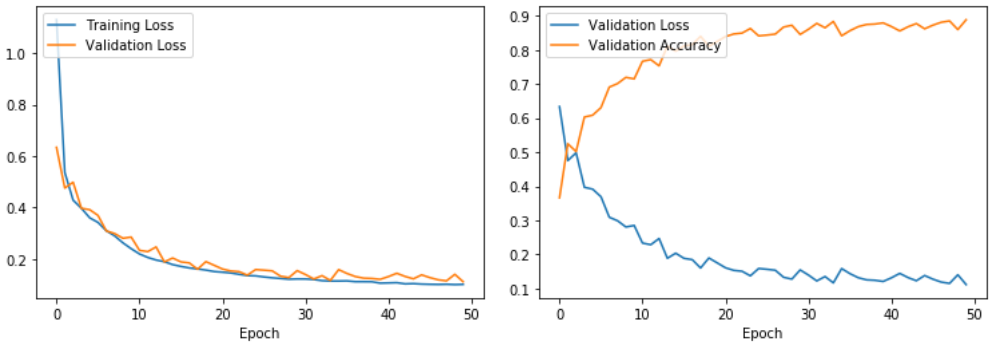


Figure 3.1: Losses and Accuracy for both training and validation for the diagonal test case.

The training and validation accuracies are not the only metric to determine the effectiveness of network in determining the eigenvalues of a matrix. Figure 3.2 shows the predicted eigenvalues plotted against the true eigenvalues of a test matrix after training the network. The red dashed line shows the line that the eigenvalue spectra would lie upon if the predicted values were exactly the true values. We can see that values are scattered about this line, but do not fall exactly upon it. Essentially, the neural network has determined that the eigenvalues of a diagonal matrix are *similar* to the diagonal entries of the matrix, but it has not determined that they are equal.

3.2 Tridiagonal Case

Despite the measures taken to decrease the training loss, the loss was still relatively high for the tridiagonal case. Figure 3.3 displays the training and validation losses as a function of the number of training epochs. We see (Left) that the training loss and validation loss are approximately equal after 100 epochs of training. However, despite the large number of epochs

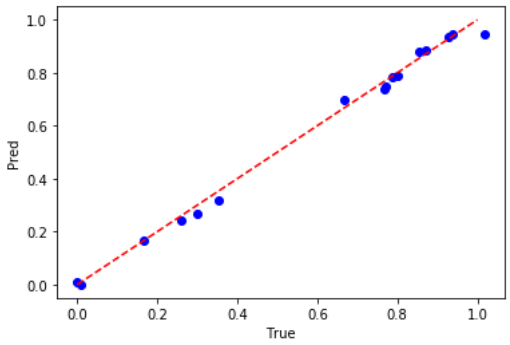


Figure 3.2: The regression for the predicted eigenvalues from a randomly uniform diagonal matrix used to test the model

used for training, the validation accuracy (Right) is still relatively low, reaching approximately 60% at its maximum. We remark that the accuracy may have been improved in this case given more training epochs, but found that it was prohibitively expensive for this work.

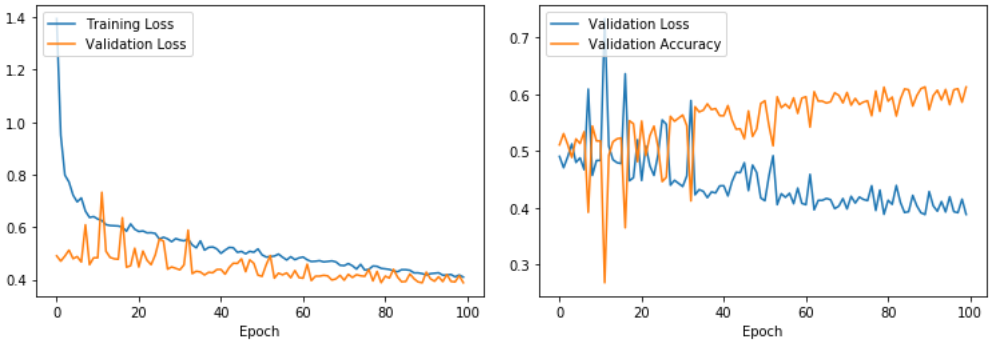


Figure 3.3: Losses and Accuracy for both training and validation for the tridiagonal test case.

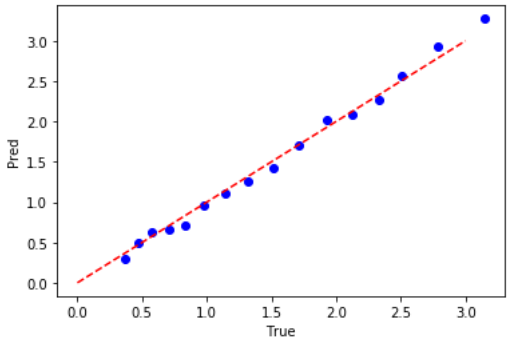


Figure 3.4: The regression for the predicted eigenvalues from a tridiagonal matrix used to test the model

Figure 3.4 displays the regression for the predicted eigenvalues for a test tridiagonal matrix. We observe that the regression for the tridiagonal case is better than that for the diagonal case, despite the lower accuracy while training. In this case, the CNN was better able to predict patterns within the matrix itself in order to predict eigenvalues. It is theorized that the more complex structure improved the true accuracy of the prediction, despite the lower training and validation accuracy. This result, while not numerically precise, is a good platform to begin development of more advanced CNNs that are capable of calculating the full spectra of any input matrix. We remark that our results presented here are of low statistical merit. Many more tests of the model are needed in order to have enough data for reliable statistics.

4 Summary

Two different CNN architectures were used in an attempt to calculate accurate eigenvalues from a general matrix of prescribed structure. For both test cases, it was found that eigenvalues were accurately predicted, but were not numerically precise. In most scientific pursuits, eigenvalues are needed that have a precision of at least 10^{-6}. Much more work is needed in order to predict eigenvalues to a similar precision. We remark that a more complex architecture might be able to alleviate this problem, such as more convolutional layers and batch normalization. However, the disadvantage to doing so would be more trainable parameters within the network and thus more training needed. We also point out that the fact that two different architectures were needed for two very similar problems highlights the fact that very carefully crafted and specific architectures are needed for a specific matrix. It was found that using the same architecture between test cases resulted in very low training accuracy for the test case the architecture was not designed for. This is both good and bad news. In applications where eigenvalue equations arise where the matrix does not have a precise structure, it would be more prudent to stick with tried and true iterative methods. However, in fields where many different matrices of the same structure are available, CNNs might be able to make a difference for eigenvalue prediction in the future.

5 Future Work

There are very many areas and ideas that have stemmed from the author's work on this project, and through a review of current literature. The ideas that have stemmed from the work on this small project are:

- Expansion of the CNN architecture to accurately predict *precise* eigenvalues and corresponding eigenvectors
- Expansion of the CNN architecture to allow for complex matrices
- The use of CNNs as a preconditioner for solving linear equations in LQCD

The first two are simple extensions of this work, but the last area deserves some more elaboration. In [5], the authors used a DNN in order to find block structures within an input

matrix so that a Block-Jacobi Preconditioner could be constructed to precondition GMRES. In [9], the authors formed a more general preconditioner for the Conjugate Gradient iterative solver. Preconditioning is a technique used heavily in LQCD in order to speed up convergence of linear equations. However, the size of the Dirac Operator ($n = O(10^9)$) stemming from modern lattices is prohibitively large for DNNs. It would take an unfeasible amount of time in order to train a decent preconditioner, when successful preconditioning techniques are readily available. However, adaptive multigrid [2] has been implemented successfully in LQCD for a number of Dirac operator discretizations. Adaptive multigrid creates a hierarchy of successively smaller grids from the original operator. In essence, the linear equations are solved on the coarsest grid, and then propogated back up to the highest grid level (the original operator). It has been shown to reduce the computational cost of the calculation, however the authors of [12] found that the computational cost on the coarse grid can make up the most significant part of the total cost. Since the number of grids can be made arbitrary, and the grid size deceases with each successive coarsening, it is possible to implement a CNN preconditioner to reduce the cost of the coarse grid calculation. The CNN can then be trained on the coarsest grid operator which is much smaller and would reduce the time and cost of training. Additionally, since a partial solve to very large residual norms on the coarsest grid can be implemented, it is possible that a CNN preconditioner may entirely remove the cost of the coarse grid if it is effective enough. The training for the CNN could be efficiently implemented during Hybrid Monte Carlo used to form the gauge fields, where the Dirac Operator must be inverted for every Molecular Dynamics (MD) step within the Monte Carlo (MC) trajectory. Since at least ten MD steps are usually needed per MC trajectory, and sometimes thousands of MC trajectories are needed, many training samples are efficiently available to form a preconditioner for the coarse grid Dirac Operator.

REFERENCES

[1] S. Baral, T. Whyte, W. Wilcox, and R. B. Morgan. Disconnected loop subtraction methods in lattice qcd. *Computer Physics Communications*, 241:64 – 79, 2019.

[2] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge. Adaptive algebraic multigrid. *SIAM J. Scientific Computing*, 27:1261–1286, 01 2006.

[3] D. Finol, Y. Lu, V. Mahadevan, and A. Srivastava. Deep convolutional neural networks for eigenvalue problems in mechanics. *International Journal for Numerical Methods in Engineering*, 118(5):258–275, 2019.

[4] M. Giulini and R. Potestio. A deep learning approach to the structural analysis of proteins, 2019.

[5] M. GÃűtz and H. Anzt. Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation, 11 2018.

[6] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

[7] R. Ramakrishnan and O. A. von Lilienfeld. Machine learning, quantum mechanics, and chemical compound space, 2015.

[8] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[9] J. Sappl, L. Seiler, M. Harders, and W. Rauch. Deep learning of preconditioners for conjugate gradient solvers in urban water related problems, 2019.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[11] K. Team. Keras. `https://github.com/keras-team/keras`.

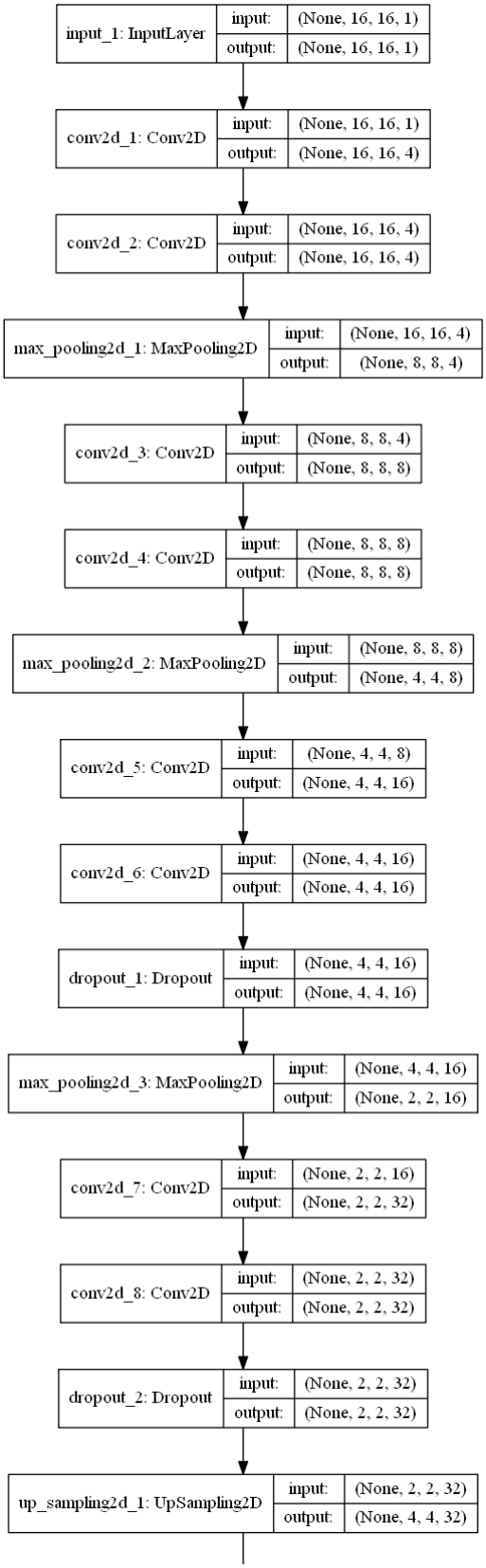
[12] T. Whyte, W. Wilcox, and R. B. Morgan. Deflated gmres with multigrid for lattice qcd, 2019.

[13] Z. Yi, Y. Fu, and H. J. Tang. Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix. *Computers Mathematics with Applications*, 47(8):1155 – 1164, 2004.

6 APPENDIX

The initial goal of this project was to produce of CNN architecture that could accurately predict the solution to linear systems arising from matrix equations. However, it was discovered that this would not be a feasible goal to accomplish. The initial architecture used was that of U-Net [8] without the tensor concatention with the output of the same hierarchy in the CNN. This was done in the hopes that the CNN would be able to train an approximate inverse as the ouput.

Figure 6.1 on page 11 and 12 displays the architecture used in pursuit of this goal. This model takes both the matrix as input and the right hand side of the equation $Ax = b$. Not displayed in the architecture is a concatenation with the result "x8" with the right hand side input before the flatten and dense layers. This was done in order to incorporate the information provided from the right hand side. Ultimately, this project was scrapped as the losses during training were of $O(10^3)$. However, comparing to results from reference [9], this is not unreasonable. Despite this, training would still have taken an excessive amount of time for this work. It is possible that earlier incorporation of the information within the right hand side to the model would make this more successful. It is the authors wish to expand upon this work with different architectures and refinements to observe if this could be used as a preconditioner within multigrid for applications within LQCD.


input_1: InputLayer
input: (None, 16, 16, 1)
output: (None, 16, 16, 1)
conv2d_1: Conv2D
input: (None, 16, 16, 1)
output: (None, 16, 16, 4)
conv2d_2: Conv2D
input: (None, 16, 16, 4)
output: (None, 16, 16, 4)
max_pooling2d_1: MaxPooling2D
input: (None, 16, 16, 4)
output: (None, 8, 8, 4)
conv2d_3: Conv2D
input: (None, 8, 8, 4)
output: (None, 8, 8, 8)
conv2d_4: Conv2D
input: (None, 8, 8, 8)
output: (None, 8, 8, 8)
max_pooling2d_2: MaxPooling2D
input: (None, 8, 8, 8)
output: (None, 4, 4, 8)
conv2d_5: Conv2D
input: (None, 4, 4, 8)
output: (None, 4, 4, 16)
conv2d_6: Conv2D
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
dropout_1: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
max_pooling2d_3: MaxPooling2D
input: (None, 4, 4, 16)
output: (None, 2, 2, 16)
conv2d_7: Conv2D
input: (None, 2, 2, 16)
output: (None, 2, 2, 32)
conv2d_8: Conv2D
input: (None, 2, 2, 32)
output: (None, 2, 2, 32)
dropout_2: Dropout
input: (None, 2, 2, 32)
output: (None, 2, 2, 32)
up_sampling2d_1: UpSampling2D
input: (None, 2, 2, 32)
output: (None, 4, 4, 32)

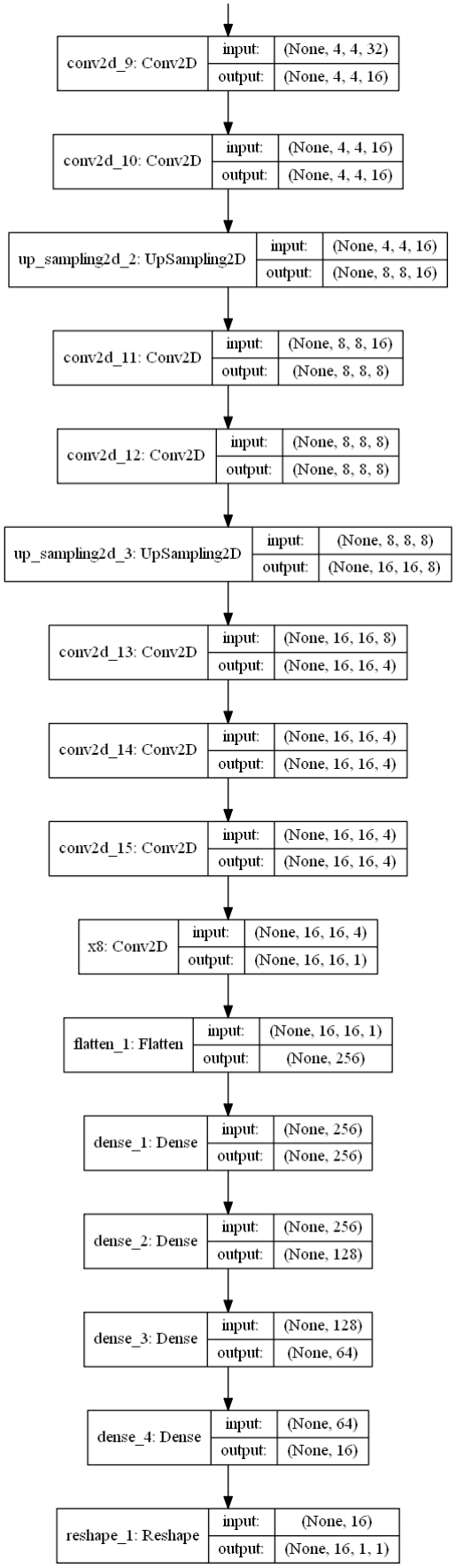


Figure 6.1: The network architecture for the attempted linear solve