Developing a Neural Network for Predicting W' Spring 2023 Recap

Laney Goldman and Becca Verghese
Prof Lynn Quantum Optics Lab

Over the course of the semester, we developed a new neural network model for detecting the next set of measurements (W') for a quantum state that has not been detected by W. Our data set consisted of randomly generated states and their 12 measurements for W. We then created a custom loss function for predicting the most favorable triplet based on a weighting system that favors more negative values but does not severely punish less negative values. We found that this model performed only slightly better than the analytical model created by Eritas and the previous iteration of the neural network.

1 Main Changes to Model

We worked on improving the success rate of our neural network. We tried to do this in multiple ways: changing the input parameters, changing the type of classification, and changing the loss function. For our input parameters, we switched from using the 12 original witness values (witness 1's max and min value, and so on) to using only the 12 measurement probabilities (HH, HV, VH, etc). In terms of changing the type of classification, we switched to a softmax classification which outputs 3 numbers, one for each triplet which shows the confidence of the model for the triplet prediction. Finally, we created a custom loss function instead of a typical Binary or Multi-label cross-entropy loss for classification models as we wanted to define our success metric. In particular, the old loss functions tended to punish the model for "errors" which would still result in a negative witness value or detection.

1.1 Data Set. We computationally generated 100,000 random entangled states that were not detected by our original 6 witnesses using the standard method of 2-qubit state generation. We created two Keras models, one trained on 40,000 states (model_qual.json and model_qual.h5) and another model trained on 80,000 states (model_qual_v2.json and model_qual_v2.h5). Upon seeing the model performances of both models, we decided to wait before creating a new model with a larger training set.

1.1.1 Random Generation Method. We used the standard method of 2-qubit state generation used over the summer. We discussed also training the models on the random generation method, which involves tracing out a 2-qubit state from a 4-qubit pure state. However, we never got around to adding this method of random generation to our training set.

1.1.2 Information Included in Dataset. Our dataset included the 12 measurements for W, as well as the population values in three bases (HV, DA, and RL). We had a concurrence threshold of 0.

1.2 model arcitecture. The model trained on 40,000 states had three layers. The layers contain a small number of nodes (3 to 7). We used a relu activation and a softmax activation for the last layer. We had a small number of layers and nodes to prevent over-fitting. For the model trained on more states, we increased the layers and the number of nodes per layer as there was less chance of over-fitting.

Fig. 1 The model architecture for the model trained on 40,000 states.

```
# get the model, in this case there n_inputs = 12, n_outputs = 3
def get_model(n_inputs, n_outputs):
    model = Sequential()
    model.add(Dense(7, input_dim=n_inputs, kernel_initializer='he_uniform', activation='relu'))
    model.add(Dense(5, activation = 'softmax'))
    model.add(Dense(n_outputs))
    opt = keras.optimizers.Adam(clipnorm=1.0)
    model.compile(loss='witness_loss_fn', optimizer=opt)
    return model
```

1.3 Loss Function.

```
def witness_loss_fn(y_true, y_pred):
    # y_pred is a 1 by batch size tensor
    with all the predicted values for each state
    # y_true is a 3 by batch size tensor,
    with the three ground truth outputs per state

    batchsize = y_true.shape[0]


    # making y_pred 1 output when evaluating loss
    y_pred_xy = y_pred[0]
    # print('y_pred_xy', str(y_pred_xy))
    y_pred_yz = y_pred[1]
    # print('y_pred_yz', str(y_pred_yz))
    y_pred_xz = y_pred[2]
    # print('y_pred_xz', str(y_pred_xz))

    # for y_true, getting labels
    xy_qual = y_true[0]
    yz_qual = y_true[1]
    xz_qual = y_true[2]

    loss = y_pred_xy*xy_qual + y_pred_yz*yz_qual
           + y_pred_xz*xz_qual

    return tf.reduce_mean(loss, axis=-1)
```

Above, is our current code for our custom loss function. This loss function uses the witness values found in our dataset, y_true, to drive the model to favor the triplet predictions that resulted in a lower witness value. We came up with this loss function after talking to Prof Talvitie in the CS department.
We note that xy_qual, yz_qual, and xz_qual are simply the most negative witness value we get from the triplet. Since a more negative value means that the entanglement is "witnessed more," these witness values are the quality metric. Our model predicts "pseudo

Version 1.18, May 4, 2023

probabilities" for each triplet. The output/prediction for each triplet is between 0 and 1. The triplet with the highest "probability" is what we choose to be our prediction. y_pred_xy, y_pred_yz, and y_pred_xz, are the model's predicted psuedo probabilities. If the minimum witness value for a triplet i is w_i and the predicted "pseudo probability" of that triplet is $P(w_i)$, we define our loss to be

$$\sum_i P(w_i)w_i.$$

We minimize this loss in hopes to predict the triplet with the most negative witness value.

2 Model Performance

We defined the success rate as

$$\frac{N - F_m}{N - F_{wp}}$$

where N is the total number of states we tested on (that are not detected by W), F_m is the number of states that our model failed on, and F_{wp} is the number of states that the W's do not detect.

Model	Total States	Model Fail	W_p Fail	Success Rate
Version 1	10000	2899	2070	0.89546
Version 2	10000	2893	2063	0.89543

We tested our models on 10,000 randomly generated entangled states to find the overall success rates. Our models were tested on different sets of 10,000 randomly generated entangled states, but the values should still indicate that the performance of both models is nearly identical. Version 1 corresponds to the model trained on 40,000 states and Version 2 corresponds to the model trained on 80,000 states.

3 Exploration: Other Method for Random Generation

We explored another random generation method for two-qubit states. This random geenration method involves generating a 4-qubit pure state and then tracing out the 2-qubit state. This method was described in chapter 3 of this paper. We used this paper for generating the pure state. We used this github repository for computing the partial trace, among other operations. The Matlab code we wrote for this is called "random_state_v2.m".

We found that for one-qubit states, the second random generation method had a more even distribution of purity. However, for the 2-qubit state, we found that the first random generation method had a more even distribution and thus was more suited for our purposes.

4 Comparing the Model with the Analytical Method

When comparing the overall success rates of both the models and the analytical method, we found the success rate was extremely similar where all had a success rate of around 0.89. Thus, we were interested in seeing if the different models were actually different. Thus, we tested our version 2 model and the analytical population method together on 1,000 entangled pure states not detected by the original 6 witnesses. We found that the methods predicted different triplets for 130 states, but that both methods succeeded for all of these states. We also saved each of these states in this document. Below is an example ket in which the analytical method predicted Triplet 2 and the model predicted Triplet 3. Both triplets detected this state so both methods succeeded.

$$|\psi\rangle = \begin{bmatrix} 0.401 \\ -0.205 + 0.386i \\ 0.272 - 0.442i \\ -0.042 - 0.613i \end{bmatrix} \quad (1)$$

Fig. 2 The purity distribution of 10,000 2-qubit states using the tracing-out method for random generation

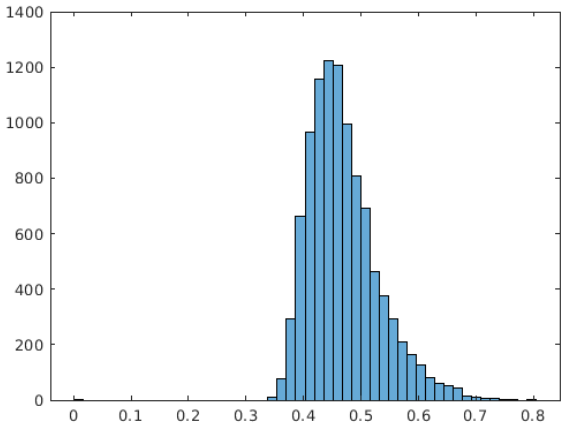


Fig. 3 The distribution of 1,000 1-qubit states using the standard method (first method) for random generation.

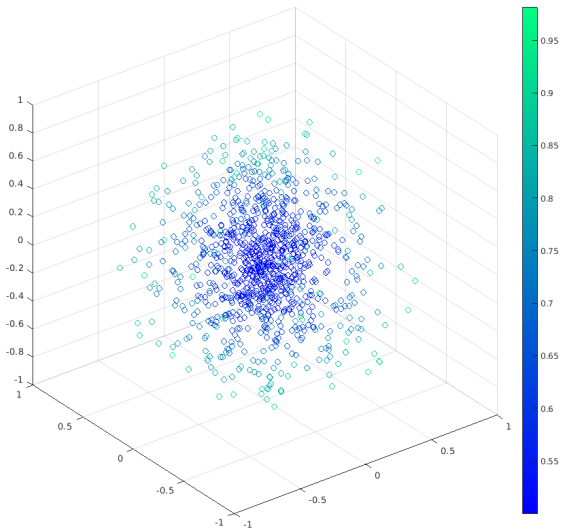
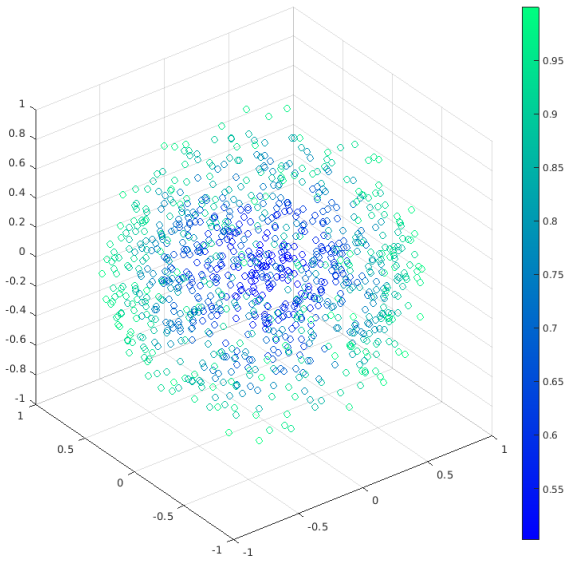


Fig. 4 The distribution of 1,000 1-qubit states using the tracing-out method (second method)for random generation.



We think it might be interesting to potentially recreate some of these states in the lab or see if there is a pattern within these states. We are planning on comparing the triplet predictions and the differences between successful states on mixed entangled states as well in the future. We expect there to be cases where one method fails and the other method succeeds which would show that the methods are in fact different.

5 Future Work

In the future, we would like to further explore the bounds of our system and figure out how well a potential model can perform given just the 12 measurements as input. We hope to explore this through a more theoretical information theory-based approach.

Furthermore, we would like to greater explore the differences between the analytical and model predictions. Based on the performance of both methods, it would seem at first glance that they are essentially the same method. However, upon some preliminary research, it seems that they predict different triplets at a rate higher than we anticipated. We aim to do this by comparing the predictions on 10,000 entangled states not detected by the original 6 witnesses and storing the states where the methods differ. We hope that this way would allow us to see if our methods cover/succeed in different sets of entangled states.

Finally, we would like to add approximately 20,000-50,000 states generated with the second method of random state generation to incorporate more mixed states into our training state. We hope that this would improve the method's success rate on extremely mixed states. After incorporating these states, we hope to do a hyperparameter sweep.

6 Code

Our code and Matlab Files All of our files can be found on the lab GitHub repository. Our Matlab files can be found in this Google Drive folder.

List of Figures

List of Tables