The Efficacy of Fully-Connected Neural Networks as Solutions to the Critical Disordered Quantum Ising Chain

In [3]:

```python
from google.colab import drive
drive.mount('mnt')
```

```
Mounted at mnt
```

In [4]:

```python
cd 'mnt/My Drive/DL_SDRG'
```

```
/content/mnt/My Drive/DL_SDRG
```

In [5]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
import tensorflow as tf
from matplotlib import pyplot
```

Background

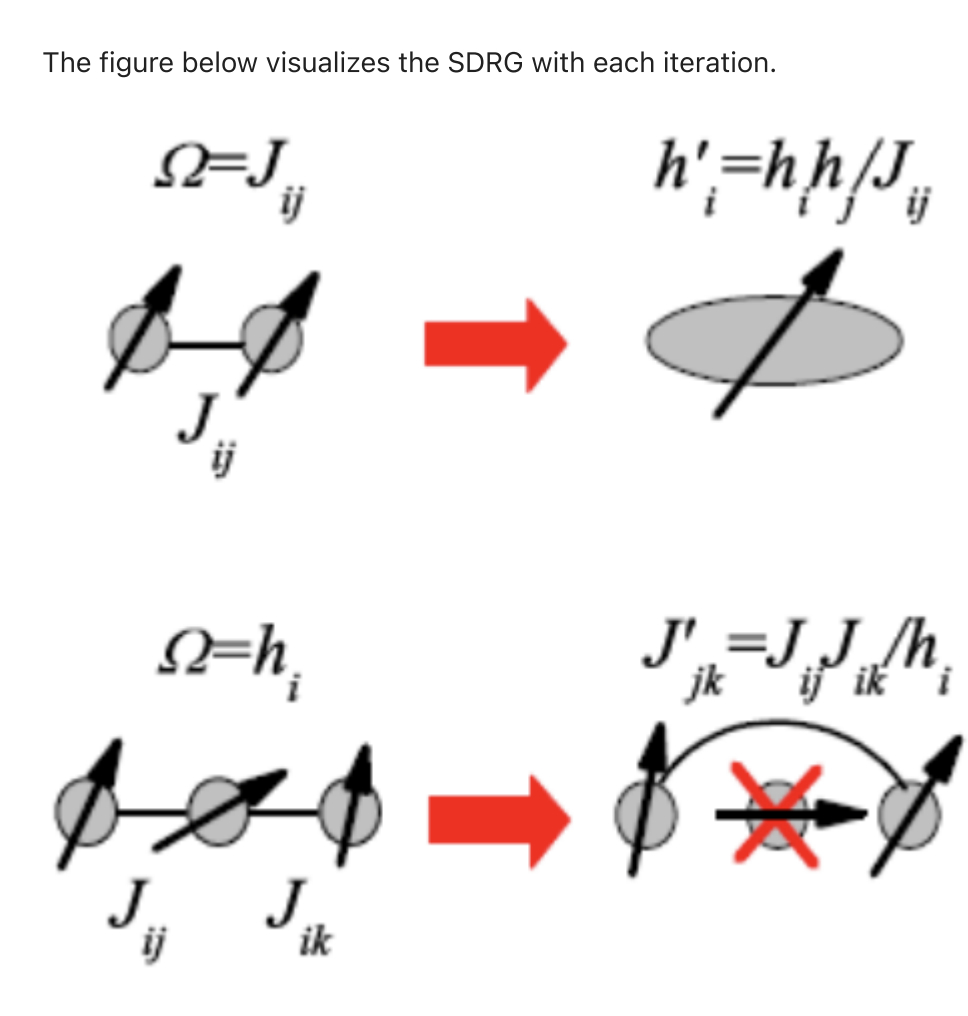
The quantum Ising model is described by the following Hamiltonian:

$$\mathcal{H} = -\sum_{\langle ij \rangle} J_{ij}\sigma_i^x\sigma_j^x - \sum_i h_i\sigma_i^z \;, \tag{1}$$

A Hamiltonian, in general, measures the total energy of the system, serving as a concise description of the model. Here, the Ising model Hamiltonian describes the interaction of entangled particles on a hypercubic lattice (e.g., a grid in two dimensions). The indices i, j indicate two neighboring particles, and $\sigma_{i,j}^{x,z}$, indicates the spin along the x or z direction of the ith or jth particle. Further, J_{ij} >0 is known as the coupling between particles i and j, aligning the spins, and h_i measures the strength of an external field applied to each particle. The solutions to this problem, namely the ways to optimize the particles arrangements to minimize the total energy, is highly nontrivial and falls under a broader class of optimization problems known as non-deterministic polynomial-time hard (NP-hard) problems.

The ground state of the RIM is conveniently determined by an efficient Strong Disorder Renormalization Group (SDRG) algorithm. During the SDRG method the largest local terms in the Hamiltonian are successively eliminated and new Hamiltonians are generated through perturbation calculation. The critical properties of the RIM are governed by an infinite disorder fixed point, in which the strength of disorder grows without limit during renormalization. Therefore, the SDRG results are %universal, i.e. independent of the original disorder distributions, as well as asymptotically exact in the vicinity of the critical point, which is indeed demonstrated both analytically and numerically. After decimating all degrees of freedom, the ground state of the RIM is found as a collection of independent ferromagnetic clusters of various sizes, each cluster being in a GHZ state $\frac{1}{\sqrt{2}}(|\uparrow\uparrow\cdots\uparrow\rangle + |\downarrow\downarrow\cdots\downarrow\rangle)$.

The figure below visualizes the SDRG with each iteration.



This project aims to explore and demonstrate the capability of machine learning applications for quantum mechanical treatments, namely the SDRG. Specifically, by training neural networks to learn the SDRG, it can not only demonstrate the ability of NNs to learn exotic physics, but since the SDRG is well understood, this can give insight toward the dynamics of NNs, and provide potential theoretical guarantees.

Generating Clusters

Below is the cluster-forming algorithm that was used for the SDRG:

In [ ]:

```python
def pbc(arr,i):  # Gives the system periodic boundary conditions (PBC)
    try:
        return arr[i]
    except IndexError:
        return arr[np.mod(i+np.size(arr), np.size(arr))]


def site_id(cluster):  # Gives site id format (explained below)
    id=np.array([cluster[0]])
    for index in range(np.size(cluster)-1):
        id=np.append(id, cluster[index+1]-cluster[index])
    return id


def fixed_h(size):  # Gives the fixed field with which we generate each sample
    chain=np.array([0, np.array([0])], dtype=object)
    for index in range(size):
        chain=np.vstack((chain, np.array([[1/np.e, np.array([index],dtype=int)]], dtype=object)))
    return np.delete(chain,(0),axis=0)


def gen_chain(size):  # Generates the coupling and field
    j=1-np.random.uniform(0,1,size)
    h=fixed_h(size)
    return j,h
```

In [ ]:

```python
def clust_length_larger(largest_size, clust_length):  # Checks to see whether a cluster has a larger size
                                                      #  than the so-called 'largest_size'
    if largest_size < clust_length:
        return True
    else:
        return False
```

In [ ]:

```python
def save_cluster_lin(coupling, field):
    largest_size = 0
    largest_cluster = np.array([])
    j=np.copy(coupling)
    h=np.copy(field)
    a=np.array([])
    for i in range(np.size(j)):
        h[i,0]=np.log(h[i,0])
    j=np.log(j)
    size_j=np.size(j)
    while size_j>2:
        for i in range(np.size(j)):  # We cycle through each site in the system
            if i>=np.size(j):
                break
            elif j[i]>h[i,0] and j[i]>pbc(h[:,0],i+1):  # There are various conditions on h given the site i
                                                        #  and the coupling j. When the condition of each of
                                                        #  the proceeding if statements is satisfied, we make
                                                        #  a new cluster, join the site to an existing
                                                        #  cluster, or put an "empty" label (0) on the site
                                                        #  (so it is not in a cluster). When a cluster fully
                                                        #  forms, we check whether it is larger than the
                                                        #  variable 'largest_size' (initialized at 0), and
                                                        #  if so, then that cluster replaces the largest
                                                        #  cluster. In this way, we can output the final
                                                        #  result of 'largest_cluster.'
                max_index=i
                next_index=np.mod(i+1, np.size(j))
                h[next_index, 0] = h[max_index, 0]+h[next_index, 0]-j[max_index]
                h[next_index, 1] = np.sort(np.append(h[max_index, 1], h[next_index, 1]))
                h=np.delete(h, max_index, 0)
                j=np.delete(j, max_index)
            elif h[i,0]>j[i] and h[i,0]>pbc(j, i-1):
                max_index=i
                prev_index=np.mod(i-1, np.size(j))
                if np.size(h[max_index, 1])>1:
                    clust_len = len(site_id(h[max_index, 1]))
                    if clust_length_larger(largest_size, clust_len):
                        largest_cluster = site_id(h[max_index, 1])
                        largest_size = clust_len
                j[prev_index] = j[prev_index]+j[max_index]-h[max_index, 0]
                h=np.delete(h, max_index, 0)
                j=np.delete(j, max_index)
            size_j=np.size(j)
            if size_j==2:
                break
    if np.max(h[:,0])>np.max(j):
        if np.size(h[np.argmax(h[:,0]),1])>1:
            clust_len = len(site_id(h[np.argmax(h[:,0]),1]))
            if clust_length_larger(largest_size, clust_len):
                largest_cluster = site_id(h[np.argmax(h[:,0]),1])
                largest_size = clust_len
        h=np.delete(h, np.argmax(h[:,0]), 0)
        if np.size(h[np.argmax(h[:,0]),1])>1:
            clust_len = len(site_id(h[np.argmax(h[:,0]),1]))
            if clust_length_larger(largest_size, clust_len):
                largest_cluster = site_id(h[np.argmax(h[:,0]),1])
                largest_size = clust_len
    else:
        cluster_1=h[np.argmax(h[:,0]),1]
        h=np.delete(h, np.argmax(h[:,0]), 0)
        if np.size(np.sort(np.append(h[np.argmax(h[:,0]),1], cluster_1)))>1:
            clust_len = len(site_id(np.sort(np.append(h[np.argmax(h[:,0]),1], cluster_1))))
            if clust_length_larger(largest_size, clust_len):
                largest_cluster = site_id(np.sort(np.append(h[np.argmax(h[:,0]),1], cluster_1)))
                largest_size = clust_len
    return largest_cluster


def site_id_to_array(site_ids, size):  # Changes output from a condensed format to a readable array
    arr = np.zeros((size))
    for index in np.arange(1, len(site_ids)):
        site_ids[index] += site_ids[index - 1]
    for site_id in site_ids:
        arr[site_id] = 1
    return arr
```

Below is an example of a coupling and field at L=32 that generates a system called 'largest_clust_test.' Note that only the largest cluster is shown -- not every cluster.

In [ ]:

```python
size_test = 32
j_test, h_test=gen_chain(size_test)
j_test
```

Out[ ]:

```
array([0.10894029, 0.76193948, 0.26892573, 0.61803792, 0.79814626,
       0.37620774, 0.73348718, 0.92004969, 0.59465111, 0.62666891,
       0.97301634, 0.07612448, 0.48421457, 0.65914935, 0.85498051,
       0.27731709, 0.67146939, 0.80777123, 0.18570787, 0.99761344,
       0.2711859 , 0.75858082, 0.70211352, 0.51392593, 0.85372044,
       0.56933118, 0.91206541, 0.84691119, 0.0922382 , 0.86932998,
       0.50656422, 0.32168971])
```

In [ ]:

```python
largest_clust_site_ids_test = save_cluster_lin(j_test, h_test)
largest_clust_test = site_id_to_array(largest_clust_site_ids_test, size_test)
largest_clust_test
```

Out[ ]:

```
array([0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0.])
```

Using the above code, 45,000 samples of SDRG systems were created for sizes L=8, 16, 32, 64, 128, and 256.

The Neural Network

Separate neural networks are trained for each system size. Only fully-connected neural networks (FCNN) are used to demonstrate the ability of achietectures. Relu is chosen as the activation function, and the final layer uses a sigmoid activation, which gives the proabbility of each site belonging to the largest clusetr. The models are trained on 50 epochs with 20% validation split, optimized by adam, using the binary cross entropy loss. Since the sigmoid gives probabilities, the accuracy metric cannot be used without rounding. Thus, MSE is chosen as a good approximation.

Each neural network is chosen to have the first and last layers correspond to the dimension of each element of the dataset to satisfy input and output sizes. The number of layers for each size remains the same, but the size of each layer is scaled linearly by input and output sizes. With the exception of the 256 NN, where the sizes of each layer is decreased due to our lack of computational resources.

In [ ]:

```python
j = np.loadtxt('data/8_j')
clust = np.loadtxt('data/8_arr')
X = np.array(j).reshape(j.shape[0], j.shape[1])
Y = np.array(clust).reshape(clust.shape[0], clust.shape[1])
model = Sequential()
model.add(Dense(8, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(8, activation='sigmoid'))
model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics='MSE')
history = model.fit(X, Y, validation_split=0.2, epochs=50, batch_size=30, verbose=0)
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'history8.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
!mkdir -p saved_model
model.save('saved_model/model8')
```

In [ ]:

```python
j = np.loadtxt('data/16_j')
clust = np.loadtxt('data/16_arr')
X = np.array(j).reshape(j.shape[0], j.shape[1])
Y = np.array(clust).reshape(clust.shape[0], clust.shape[1])
model = Sequential()
model.add(Dense(8*2, activation='relu'))
model.add(Dense(16*2, activation='relu'))
model.add(Dense(64*2, activation='relu'))
model.add(Dense(256*2, activation='relu'))
model.add(Dense(32*2, activation='relu'))
model.add(Dense(8*2, activation='sigmoid'))
model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics='MSE')
history = model.fit(X, Y, validation_split=0.2, epochs=50, batch_size=30, verbose=0)
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'history16.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
!mkdir -p saved_model
model.save('saved_model/model16')
```

In [ ]:

```python
j = np.loadtxt('data/32_j')
clust = np.loadtxt('data/32_arr')
X = np.array(j).reshape(j.shape[0], j.shape[1])
Y = np.array(clust).reshape(clust.shape[0], clust.shape[1])
model = Sequential()
model.add(Dense(8*2*2, activation='relu'))
model.add(Dense(16*2*2, activation='relu'))
model.add(Dense(64*2*2, activation='relu'))
model.add(Dense(256*2*2, activation='relu'))
model.add(Dense(32*2*2, activation='relu'))
model.add(Dense(8*2*2, activation='sigmoid'))
model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics='MSE')
history = model.fit(X, Y, validation_split=0.2, epochs=50, batch_size=30, verbose=0)
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'history32.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
!mkdir -p saved_model
model.save('saved_model/model32')
```

In [ ]:

```python
j = np.loadtxt('data/64_j')
clust = np.loadtxt('data/64_arr')
X = np.array(j).reshape(j.shape[0], j.shape[1])
Y = np.array(clust).reshape(clust.shape[0], clust.shape[1])
model = Sequential()
model.add(Dense(8*2*2*2, activation='relu'))
model.add(Dense(16*2*2*2, activation='relu'))
model.add(Dense(64*2*2*2, activation='relu'))
model.add(Dense(256*2*2*2, activation='relu'))
model.add(Dense(32*2*2*2, activation='relu'))
model.add(Dense(8*2*2*2, activation='sigmoid'))
model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics='MSE')
history = model.fit(X, Y, validation_split=0.2, epochs=50, batch_size=30, verbose=0)
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'history64.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
!mkdir -p saved_model
model.save('saved_model/model64')
```

In [ ]:

```python
j = np.loadtxt('data/128_j')
clust = np.loadtxt('data/128_arr')
X = np.array(j).reshape(j.shape[0], j.shape[1])
Y = np.array(clust).reshape(clust.shape[0], clust.shape[1])
model = Sequential()
model.add(Dense(8*2*2*2*2, activation='relu'))
model.add(Dense(16*2*2*2*2, activation='relu'))
model.add(Dense(64*2*2*2*2, activation='relu'))
model.add(Dense(256*2*2*2*2, activation='relu'))
model.add(Dense(32*2*2*2*2, activation='relu'))
model.add(Dense(8*2*2*2*2, activation='sigmoid'))
model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics='MSE')
history = model.fit(X, Y, validation_split=0.2, epochs=50, batch_size=30, verbose=0)
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'history128.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
!mkdir -p saved_model
model.save('saved_model/model128')
```

In [ ]:

```python
j = np.loadtxt('data/128_j')
clust = np.loadtxt('data/128_arr')
X = np.array(j).reshape(j.shape[0], j.shape[1])
Y = np.array(clust).reshape(clust.shape[0], clust.shape[1])
model = Sequential()
model.add(Dense(256, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(1048, activation='relu'))
model.add(Dense(4096, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(256, activation='sigmoid'))
model.compile(loss='BinaryCrossentropy', optimizer='adam', metrics='MSE')
history = model.fit(X, Y, validation_split=0.2, epochs=50, batch_size=30, verbose=0)
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'history128.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
!mkdir -p saved_model
model.save('saved_model/model128')
```

In [6]:

```python
history = pd.read_csv('history8.csv')
training_loss_8 = history['loss'].to_numpy()
training_acc_8 = history['MSE'].to_numpy()
testing_loss_8 = history['val_loss'].to_numpy()
testing_acc_8 = history['val_MSE'].to_numpy()
history = pd.read_csv('history16.csv')
training_loss_16 = history['loss'].to_numpy()
training_acc_16 = history['MSE'].to_numpy()
testing_loss_16 = history['val_loss'].to_numpy()
testing_acc_16 = history['val_MSE'].to_numpy()
history = pd.read_csv('history32.csv')
training_loss_32 = history['loss'].to_numpy()
training_acc_32 = history['MSE'].to_numpy()
testing_loss_32 = history['val_loss'].to_numpy()
testing_acc_32 = history['val_MSE'].to_numpy()
history = pd.read_csv('histor64.csv')
training_loss_64 = history['loss'].to_numpy()
training_acc_64 = history['MSE'].to_numpy()
testing_loss_64 = history['val_loss'].to_numpy()
testing_acc_64 = history['val_MSE'].to_numpy()
history = pd.read_csv('histor128.csv')
training_loss_128 = history['loss'].to_numpy()
training_acc_128 = history['MSE'].to_numpy()
testing_loss_128 = history['val_loss'].to_numpy()
testing_acc_128 = history['val_MSE'].to_numpy()
history = pd.read_csv('histor256.csv')
training_loss_256 = history['loss'].to_numpy()
training_acc_256 = history['MSE'].to_numpy()
testing_loss_256 = history['val_loss'].to_numpy()
testing_acc_256 = history['val_MSE'].to_numpy()
```

In [7]:

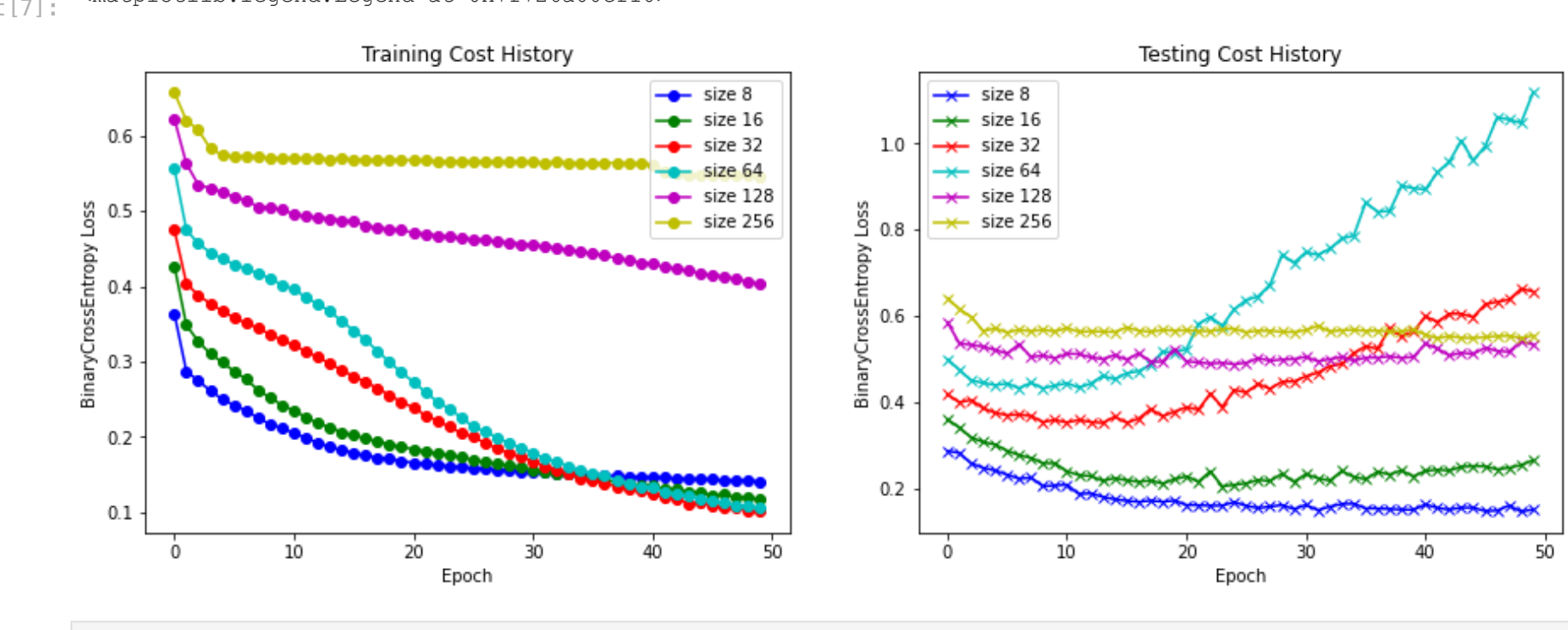
```python
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))

axs[0].set_title(f'Training Cost History')
axs[0].plot(np.array(range(50)), training_loss_8, 'b', marker = 'o', label='size 8')
axs[0].plot(np.array(range(50)), training_loss_16, 'g', marker = 'o', label='size 16')
axs[0].plot(np.array(range(50)), training_loss_32, 'r', marker = 'o', label='size 32')
axs[0].plot(np.array(range(50)), training_loss_64, 'c', marker = 'o', label='size 64')
axs[0].plot(np.array(range(50)), training_loss_128, 'm', marker = 'o', label='size 128')
axs[0].plot(np.array(range(50)), training_loss_256, 'y', marker = 'o', label='size 256')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('BinaryCrossEntropy Loss')
axs[0].legend()

axs[1].set_title(f'Testing Cost History')
axs[1].plot(np.array(range(50)), testing_loss_8, 'b', marker = 'x', label='size 8')
axs[1].plot(np.array(range(50)), testing_loss_16, 'g', marker = 'x', label='size 16')
axs[1].plot(np.array(range(50)), testing_loss_32, 'r', marker = 'x', label='size 32')
axs[1].plot(np.array(range(50)), testing_loss_64, 'c', marker = 'x', label='size 64')
axs[1].plot(np.array(range(50)), testing_loss_128, 'm', marker = 'x', label='size 128')
axs[1].plot(np.array(range(50)), testing_loss_256, 'y', marker = 'x', label='size 256')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('BinaryCrossEntropy Loss')
axs[1].legend()
```

Out[7]:

```
<matplotlib.legend.Legend at 0x7f726a00cf10>
```



In [8]:

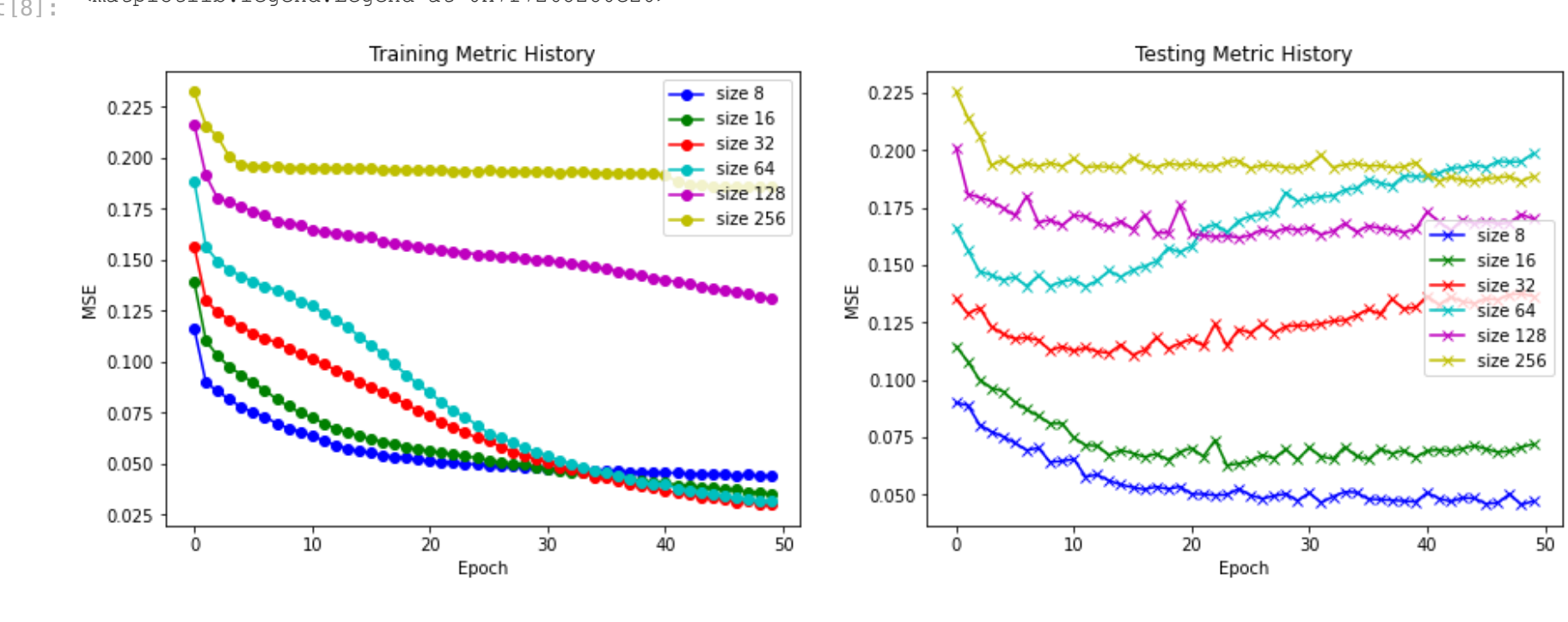
```python
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))

axs[0].set_title(f'Training Metric History')
axs[0].plot(np.array(range(50)), training_acc_8, 'b', marker = 'o', label='size 8')
axs[0].plot(np.array(range(50)), training_acc_16, 'g', marker = 'o', label='size 16')
axs[0].plot(np.array(range(50)), training_acc_32, 'r', marker = 'o', label='size 32')
axs[0].plot(np.array(range(50)), training_acc_64, 'c', marker = 'o', label='size 64')
axs[0].plot(np.array(range(50)), training_acc_128, 'm', marker = 'o', label='size 128')
axs[0].plot(np.array(range(50)), training_acc_256, 'y', marker = 'o', label='size 256')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('MSE')
axs[0].legend()

axs[1].set_title(f'Testing Metric History')
axs[1].plot(np.array(range(50)), testing_acc_8, 'b', marker = 'x', label='size 8')
axs[1].plot(np.array(range(50)), testing_acc_16, 'g', marker = 'x', label='size 16')
axs[1].plot(np.array(range(50)), testing_acc_32, 'r', marker = 'x', label='size 32')
axs[1].plot(np.array(range(50)), testing_acc_64, 'c', marker = 'x', label='size 64')
axs[1].plot(np.array(range(50)), testing_acc_128, 'm', marker = 'x', label='size 128')
axs[1].plot(np.array(range(50)), testing_acc_256, 'y', marker = 'x', label='size 256')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('MSE')
axs[1].legend()
```

Out[8]:

```
<matplotlib.legend.Legend at 0x7f7268280e20>
```



Results

In *Table 1* and *Figure 1*, we show the average accuracy ($\text{accuracy} = 1 - \frac{\text{number of misclassifications}}{\text{number of sites}}$) for 10,000 samples of the training set and 9,000 samples of the testing set (the entire testing set):

Table 1a: Average accuracy for the training and testing sets. As size increases, we see that the training and testing accuracies decrease. Also notice that the accuracies are very similar for L=256.

Table 1b: Average number of misclassifications per sample. Again we see a general increas with size, with the number of misclassificatinos being approximately equal between training and testing samples at L=256.

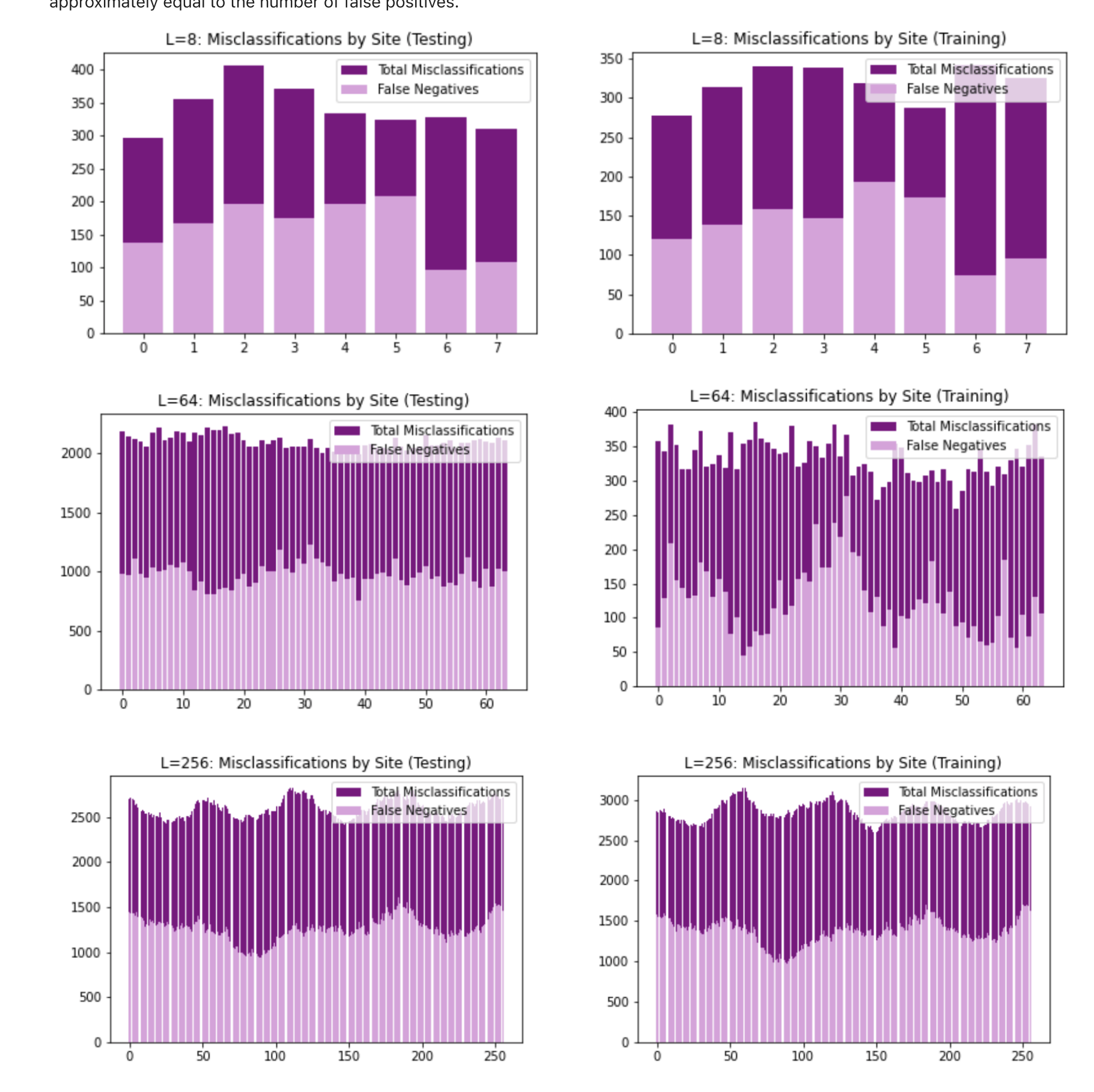
Average Accuracy

	Training Set (10000 samples)	Testing Set (9000 samples)
8	0.9683	0.9621
16	0.9598	0.9202
32	0.9621	0.8319
64	0.9668	0.7667
128	0.8102	0.7550
256	0.7145	0.7111

Average Number of Misclassifications / Sample

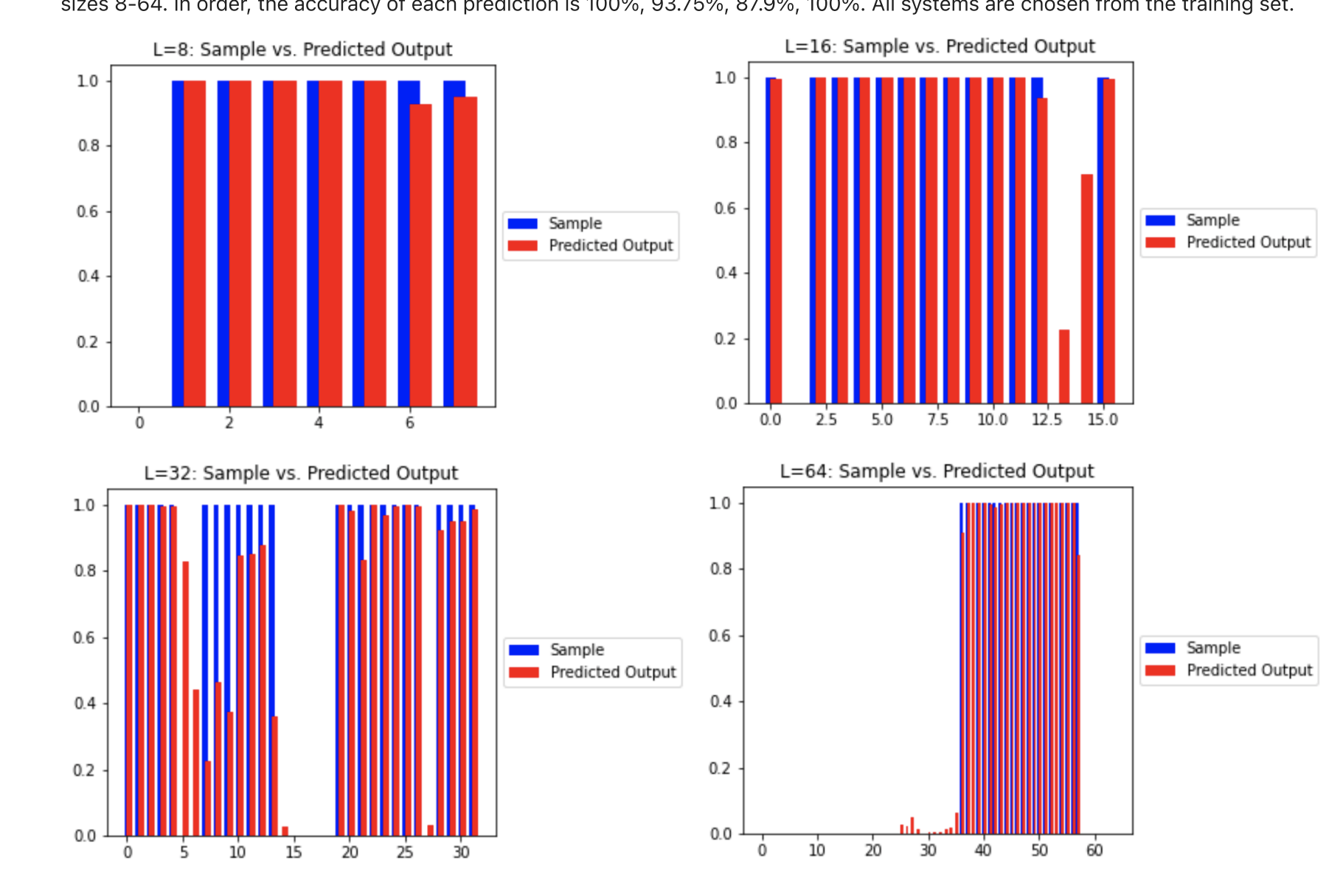
	Training Set (10000 samples)	Testing Set (9000 samples)
8	0.2540	0.3031
16	0.6436	1.2763
32	1.2135	5.3808
64	2.1230	14.9301
128	24.2969	31.3609
256	73.0961	73.9461

Figure 1a-f: Number of misclassifications for various sizes among the testing/training sets. Across all, the number of misclassifications is roughly uniform across all sites, indicating that **our models learned periodic boundary conditions** (i.e., we don't see a greater number of misclassifications near the ends of the systems). Additionally, the number of false negatives is approximately equal to the number of false positives.



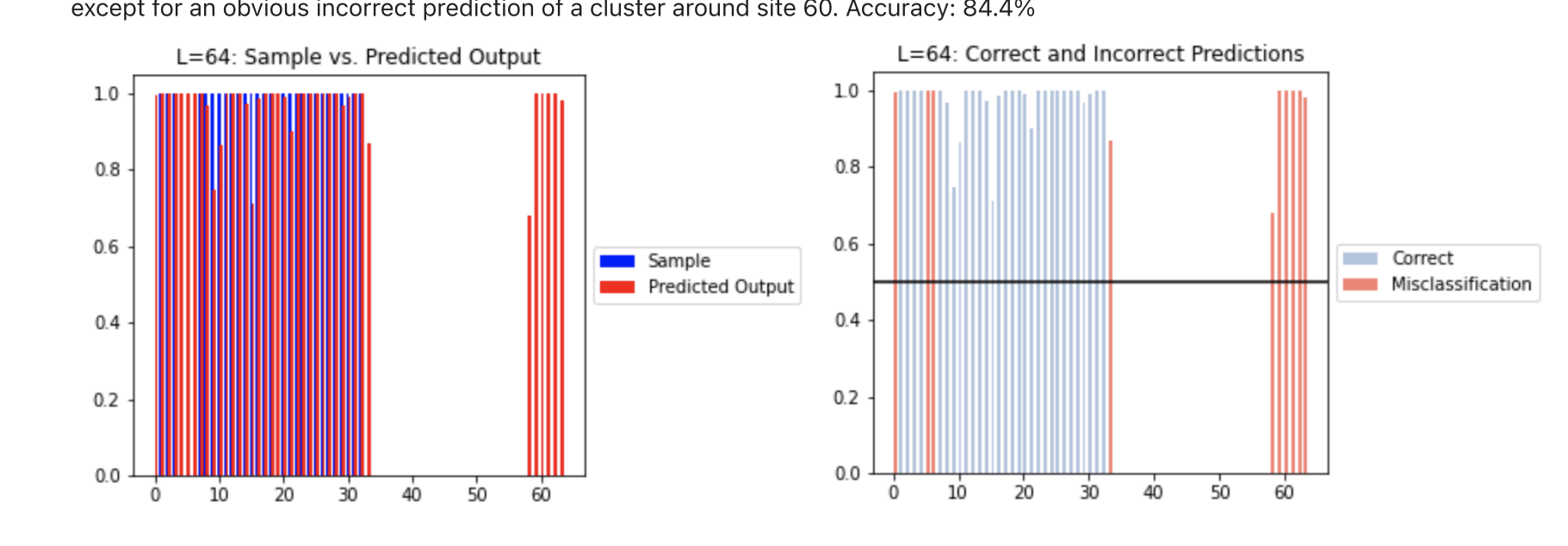
We can site-by-site compare our predicted models with the actual systems for various sizes.

Figures 2-4: Comparing predicted output to actual system site-by-site, we see great agreement between the model and system for sizes 8-64. In order, the accuracy of each prediction is 100%, 93.75%, 87.9%, 100%. All systems are chosen from the training set.



Additionally, we can perform an analysis on the number of misclassifications made by the model. This is shown in Figure 5 on a sample from the L=64 testing set.

Figure 5a-b: Misclassification analysis on a training set sample at L=64. We see general agreement between model and system except for an obvious incorrect prediction of a cluster around site 60. Accuracy: 84.4%



The misclassification analysis of *Figure 5b* demonstrates that the predicted output can be interpreted as the probability that a 1 exists at a particular site. Below we perform this analysis for several training and testing samples at L=128 and L=256 (where the accuracy of the prediction is expected to be much lower than for smaller sizes, see *Table 1a-b*).

Figure 6a-b: A training sample at size L=128. Notice that the predicted output can be interpreted as a smooth approximation of the actual system. Also, we see that the most disagreement tends to occur near the boundaries of clusters (this is generally found to be the case). Accuracy: 76.6%

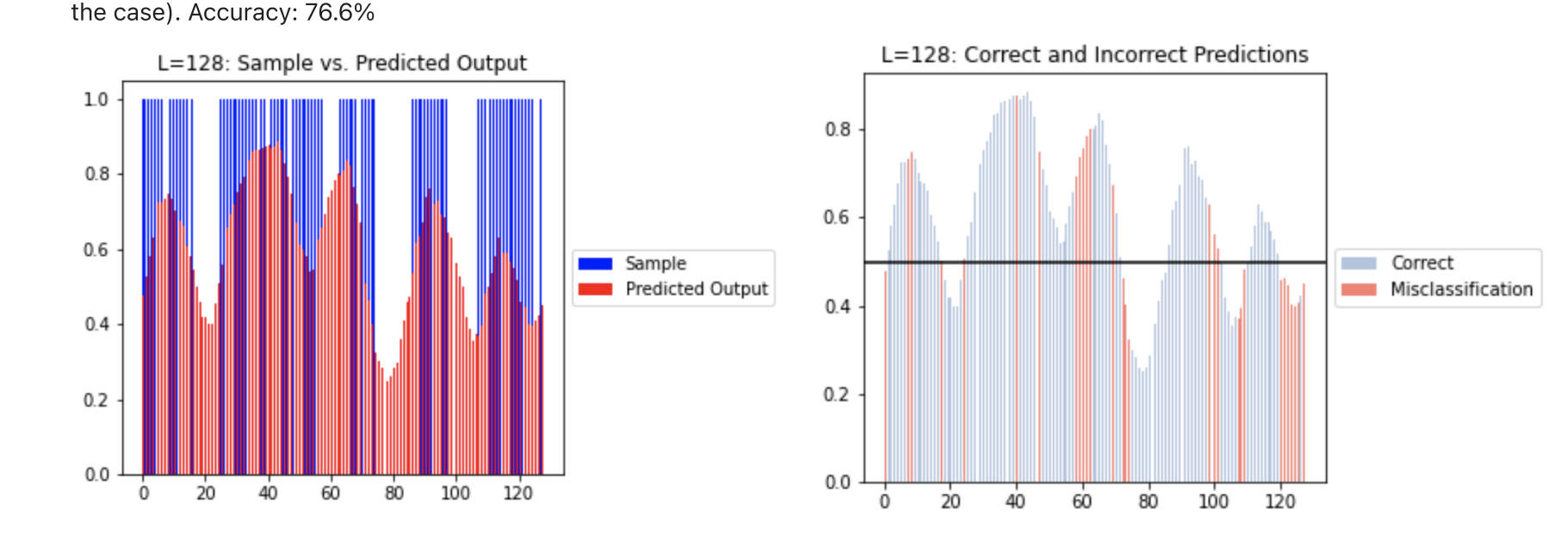


Figure 7a-b: A testing sample at size L=128. Notice that the spikes in the predicted output around sites 60-80 and 120 do not meet the 0.5 threshold to be considered misclassifications (i.e., they are rounded down to 0). Accuracy: 93.0%

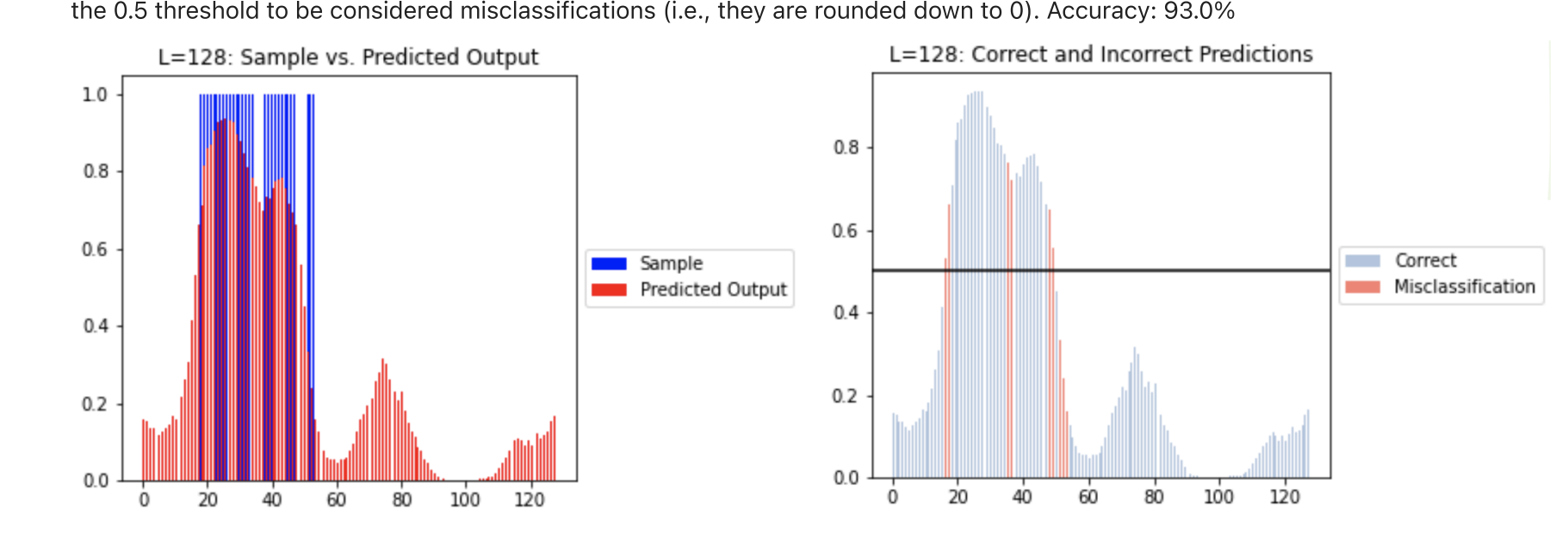


Figure 8a-b: A training sample at size L=256. The predicted output is approximately Gaussian around the large central cluster. Accuracy: 85.2%

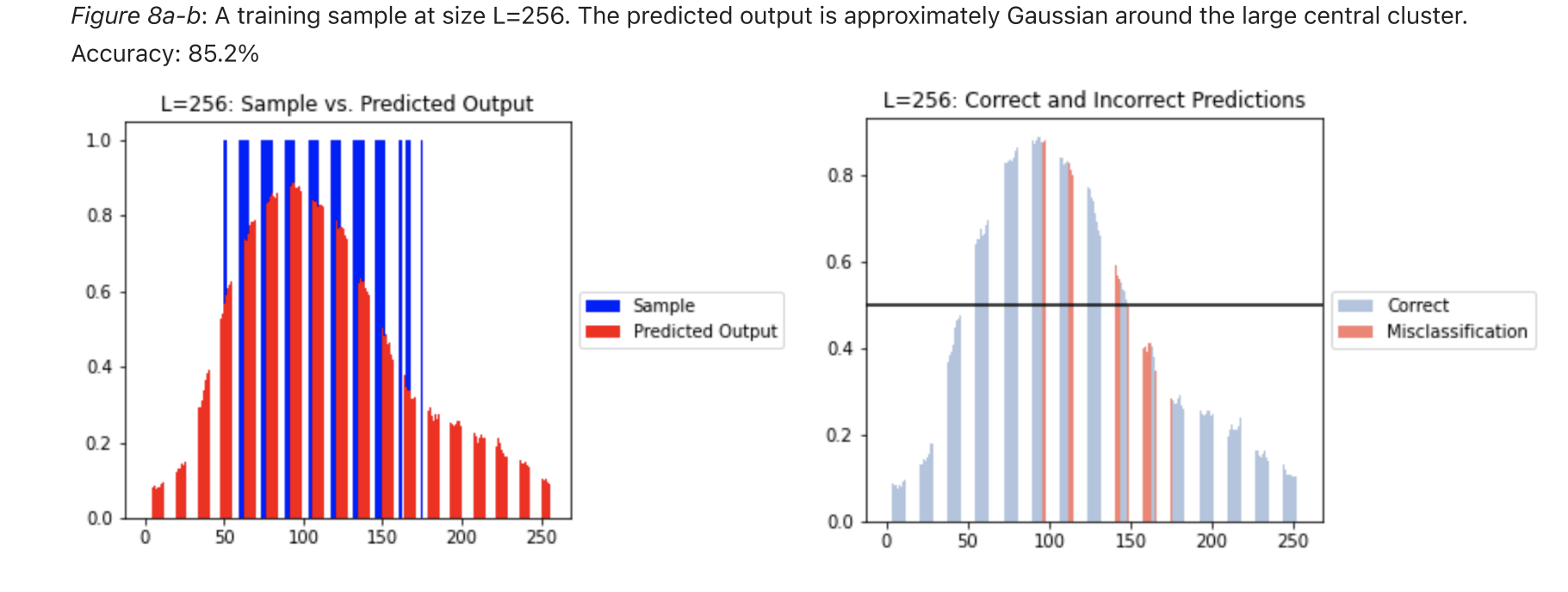


Figure 9a-b: A testing sample at size L=256. The high accuracy of the predicted output implies that the model learned PBC. Accuracy: 85.2%

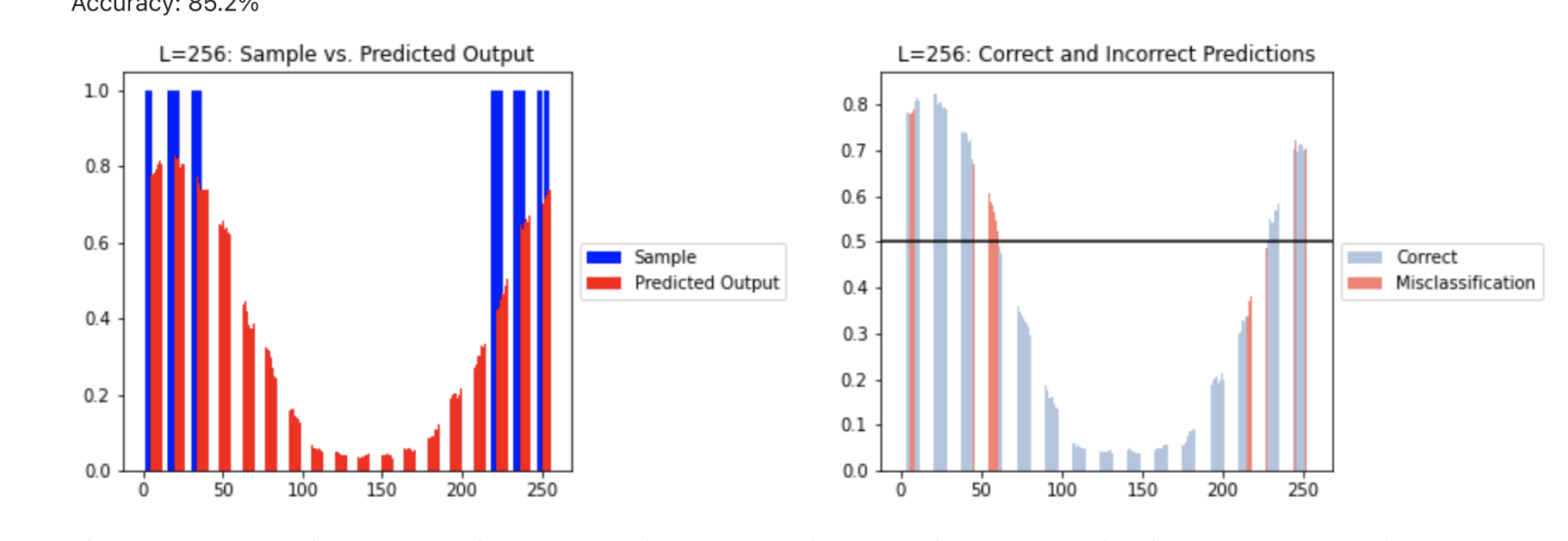
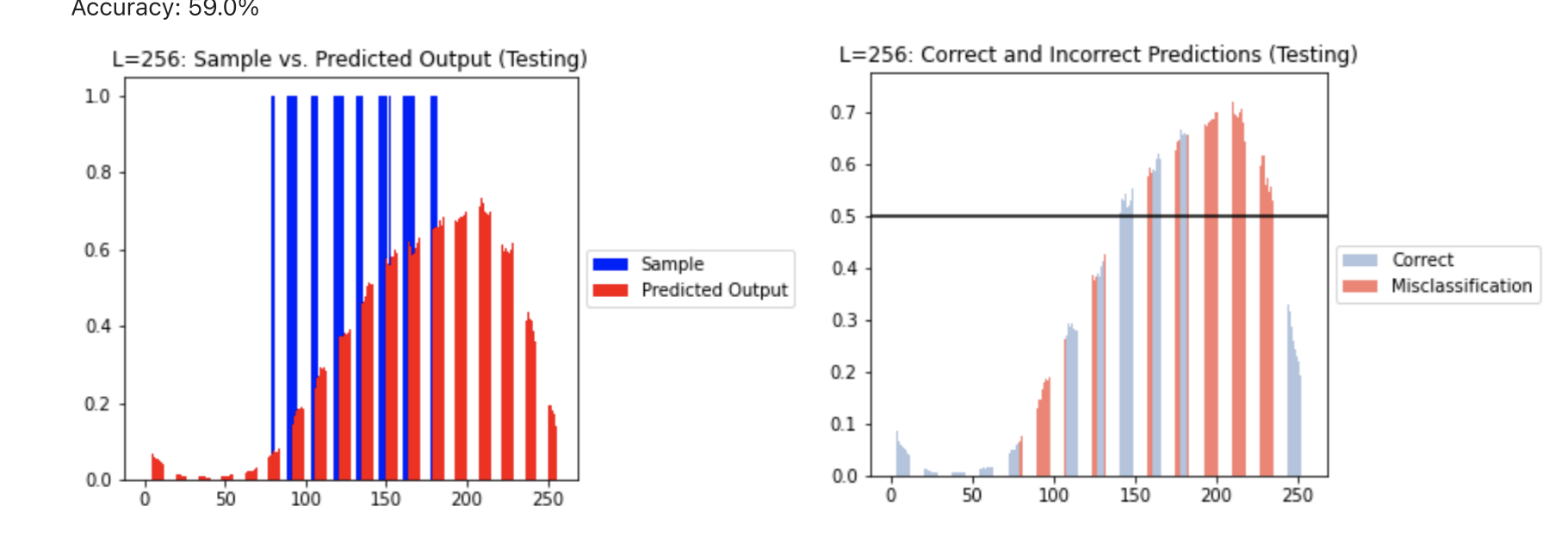


Figure 10a-b: A more inaccurate testing sample at size L=256. Again we see disagreement primarily along the edges of the cluster. Accuracy: 59.0%



Conclusions and Next Steps

We demonstrated the ability to use deep learning to gain meaningful results about SDRG systems. In fact, even naive archietectures (FCNNs) are able to demonstrate good results. Next steps would include allocating more computational resources toward larger sizes, such as increasing the number of layers, neurons, and epochs. We expect better convergence for traing and testing costs and metrics.

Also, we tend to see misclassifications aroung the edges of clusters, so using different activation functions, such as the hard-sigmoid would be a logical next step in improving our current models.

Additional analytic work should be studied. Specifically, since the SDRG is well studied, comparing the weights of each neuron to that of the SDRG will give insight on the dynamics of FCNNs. An intuitive place to start is to have the same number of layers and neurons as the number of SDRG steps and remaining sites, and see if the two algorithms behave similarly.

More exotic neural networks should also be explored. Specifically, there is a natural connection between neural networks and complex networks, so studying the performance of graph neural networks would be interesting. Graph neural networks will also support SDRG in higher dimensions with less resources, whereas FCNNs will likely face difficulties. Ultimately, this study should serve as the basis for understanding NNs, as well as using NNs to generate new results in the realm of quantum information through reinforcement learning.