Machine Learning Discussion Meeting - Interactive Notebook

In this notebook, we will implement some of the various techniques discussed in the talk "Machine Learning: Concepts and Applications".

To install the requirements, please follow the instructions in the readme.

This notebook is composed of 4 parts:

1. We will build a deep convolutional neural network to classify handwritten digits.
2. Next, we will build a deep convolutional autoencoder to build a compression algorithm for the handwritten digits.
3. We will use a previously generated dataset to train a deep learning model to solve the Schrödinger Equation and build the total energy as a functional of the external potential for one-electron model systems.
4. Finally, the same model will be used to build the total energy as a functional of the charge density to compared with part 3.

1. Deep Convolutional Neural Network to Classify Handwritten Digits.

First, we will import the MNIST dataset of handwritten digits. This dataset consists of 70,000 28x28 grayscale images, along with the value in the image (0-9).

We will then split this into 60,000 training examples, 5000 validation examples, and 5000 test examples.

As we want to predict probabilities that the image contains each digit, we will convert the value to a caegorical value using one-hot encoding. e.g 7 -> [0,0,0,0,0,0,0,1,0,0]

The input data to the network should be normalised, we will normalise the images by dividing by 255 (the maximum value of 8-bit pixel data).maximum

We can then print the shapes of the data to make sure everything seems correct.

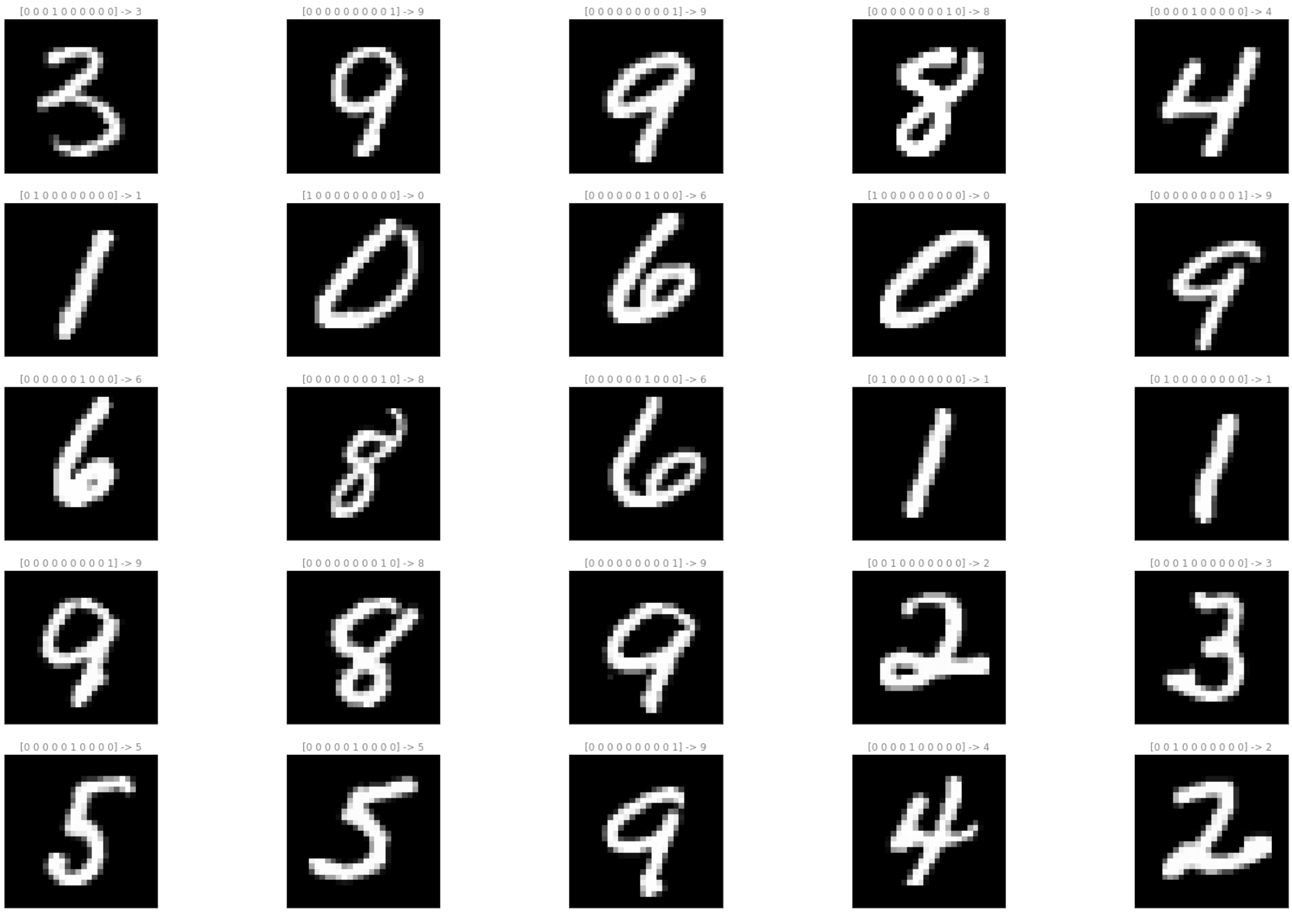
In [1]:
```
# Imports.
import pickle
import numpy as np
import tensorflow as tf
import matplotlib
import matplotlib.pyplot as plt


# # Set up GPU. [Uncomment this section if you have set up TensorFlow with the GPU on your system.]
# gpus = tf.config.experimental.list_physical_devices('GPU')
# if gpus:
#     try:
#         for gpu in gpus:
#             tf.config.experimental.set_memory_growth(gpu, True)
#     except RuntimeError as e:
#         print(e)


# Load in the dataset.
dataset = tf.keras.datasets.mnist


# Split the dataset into training, validation and testing data.
(x_train, y_train), (x_test, y_test) = dataset.load_data()
x_train = x_train.reshape((x_train.shape[0], x_train.shape[1], x_train.shape[2], 1))
x_test = x_test.reshape((x_test.shape[0], x_test.shape[1], x_test.shape[2], 1))
x_validation = x_test[:5000,...]
x_test = x_test[5000:,...]
y_validation = y_test[:5000,...]
y_test = y_test[5000:,...]


#  Convert the y values to categorical.
y_train = tf.keras.utils.to_categorical(y_train)
y_validation = tf.keras.utils.to_categorical(y_validation)
y_test = tf.keras.utils.to_categorical(y_test)


# Normalise the data.
x_train = x_train / 255.0
x_validation = x_validation / 255.0
x_test = x_test / 255.0


# Print the shapes of the data
print(x_train.shape, x_validation.shape, x_test.shape, y_train.shape, y_validation.shape, y_test.shape)
```

```
(60000, 28, 28, 1) (5000, 28, 28, 1) (5000, 28, 28, 1) (60000, 10) (5000, 10) (5000, 10)
```

We can now plot some of the test examples, to help visualise the data.

In [2]:
```
# Plot some results.
plt.rcParams['figure.figsize'] = [30, 20]
plt.rcParams['text.color'] = 'gray'
fig, axs = plt.subplots(5, 5)
k = 0
for i in range(axs.shape[0]):
    for j in range(axs.shape[1]):
        axs[i, j].imshow(x_test[k, :, :, :].reshape((28, 28)), cmap='gray')
        axs[i, j].set_title('{0} -> {1}'.format(y_test[k].astype(int), y_test[k].argmax()))
        axs[i, j].set_yticklabels([])
```

```
        axs[i, j].set_xticklabels([])
        axs[i, j].set_yticks([])
        axs[i, j].set_xticks([])
        k = k + 1
plt.savefig('digits.pdf')
```



We will next define the following structure of the neural network:

The input will be the image, the output will be the 10 probabilities.

In [3]:

```
# Build the neural network.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(filters=6, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=100, activation='relu'))
model.add(tf.keras.layers.Dense(units=84, activation='relu'))
model.add(tf.keras.layers.Dense(units=10, activation='softmax'))
model.summary()


# Complile the model.
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 6)         60        
                                                                 
 max_pooling2d (MaxPooling2D  (None, 13, 13, 6)        0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 11, 11, 16)        880       
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 5, 5, 16)         0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (None, 400)               0         
                                                                 
 dense (Dense)               (None, 100)               40100     
                                                                 
 dense_1 (Dense)             (None, 84)                8484      
                                                                 
 dense_2 (Dense)             (None, 10)                850       
                                                                 
=================================================================
Total params: 50,374
Trainable params: 50,374
Non-trainable params: 0
_________________________________________________________________
```

Next, we will train the model.

The number of **epochs** defines how many times the model will see the *whole* dataset. The **batch size** defines how many examples will be used to update the model each step. *In principle we would like the whole dataset, but we are limited by the memory on the GPU.* And we can chose to **shuffle** the data with each epoch.

In [4]:

```
# Train the model.
training = model.fit(x_train, y_train, validation_data=(x_validation, y_validation), epochs=5, batch_size=250, shuffle=True)
```

```
Epoch 1/5
240/240 [==============================] - 3s 4ms/step - loss: 0.4627 - accuracy: 0.8676 - val_loss: 0.1790 - val_accuracy: 0.
9450
Epoch 2/5
240/240 [==============================] - 1s 3ms/step - loss: 0.1151 - accuracy: 0.9653 - val_loss: 0.1249 - val_accuracy: 0.
9590
Epoch 3/5
240/240 [==============================] - 1s 3ms/step - loss: 0.0831 - accuracy: 0.9743 - val_loss: 0.0958 - val_accuracy: 0.
9680
Epoch 4/5
240/240 [==============================] - 1s 3ms/step - loss: 0.0666 - accuracy: 0.9799 - val_loss: 0.0766 - val_accuracy: 0.
9754
Epoch 5/5
240/240 [==============================] - 1s 3ms/step - loss: 0.0578 - accuracy: 0.9822 - val_loss: 0.0835 - val_accuracy: 0.
9738
```

As we will **not further modify the model**, we can apply it to the test set.

In [5]:

```
# Evaluate the model.
model.evaluate(x_test, y_test, verbose=2)
```

```
157/157 - 1s - loss: 0.0311 - accuracy: 0.9904 - 662ms/epoch - 4ms/step
```

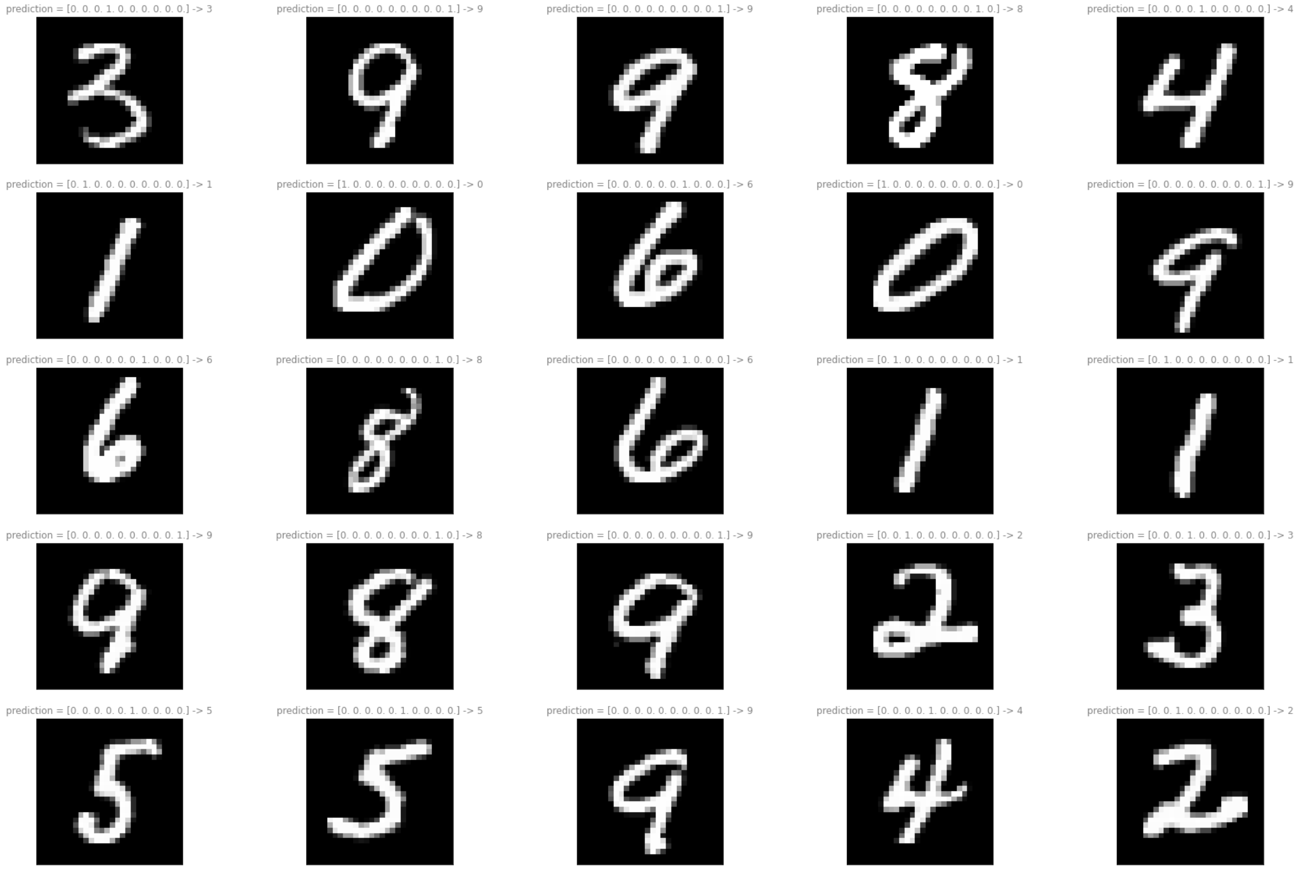
Out[5]:

```
[0.031061163172125816, 0.9904000163078308]
```

We can now visualise the predictions made by the model on the test set.

In [6]:

```
# Predict using the model.
N = 25 # Number of test images to predict
y_pred = model.predict(x_test[:N, :, :])


# Plot some results.
plt.rcParams['figure.figsize'] = [30, 20]
fig, axs = plt.subplots(5, 5)
k = 0
for i in range(axs.shape[0]):
    for j in range(axs.shape[1]):
        axs[i, j].imshow(x_test[k, :, :, :].reshape((28, 28)), cmap='gray')
        axs[i, j].set_title('prediction = {0} -> {1}'.format(y_pred[k].round(2), y_pred[k].argmax()))
        axs[i, j].set_yticklabels([])
        axs[i, j].set_xticklabels([])
        axs[i, j].set_yticks([])
        axs[i, j].set_xticks([])
        k = k + 1
plt.savefig('predictions.pdf')
```



2. Deep Convolutional Autoencoder

In order to build a custom compression algorithm for these images, we train a deep convolutional autoencoder:

Play around with the values of the hyperparameters and the structure of the model to see how it affects the performance. And how small you can make the latent space (compressed image size).

It is worth noting that we split the model into two parts, the encoder and decoder. This enables us to compress and decompress the images seperatly.

In [7]:
```
# Hyperparameters
optimizer = 'adadelta'
learning_rate = 1.0
epochs = 30
batch_size = 128
loss = 'mse'
shuffle = True


# Build the model.
input_img = tf.keras.layers.Input(shape=(28, 28, 1))
encoded = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
encoded = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(encoded)
encoded = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
encoded = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(encoded)
encoded = tf.keras.layers.Conv2D(2, (3, 3), activation='relu', padding='same')(encoded)
encoded = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(encoded)
latent_shape = (int(encoded.shape[1]), int(encoded.shape[2]), int(encoded.shape[3]))
print(latent_shape)
decoded = tf.keras.layers.Conv2D(2, (3, 3), activation='relu', padding='same')(encoded)
decoded = tf.keras.layers.UpSampling2D((2, 2))(decoded)
decoded = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(decoded)
decoded = tf.keras.layers.UpSampling2D((2, 2))(decoded)
decoded = tf.keras.layers.Conv2D(16, (3, 3), activation='relu')(decoded)
decoded = tf.keras.layers.UpSampling2D((2, 2))(decoded)
decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(decoded)
autoencoder = tf.keras.models.Model(input_img, decoded)


# Print the model architecture.
autoencoder.summary()


# Define the encoder.
encoder = tf.keras.models.Model(input_img, encoded)


# Define the decoder.
encoded_input = tf.keras.layers.Input(shape=latent_shape)
decoder_layers = len(autoencoder.layers) - len(encoder.layers)
for i in range(decoder_layers, 0, -1):
    decoder_layer = autoencoder.layers[-i]
    if i == decoder_layers:
```

```
        nest = decoder_layer(encoded_input)
    else:
        nest = decoder_layer(nest)
decoder = tf.keras.models.Model(encoded_input, nest)


# Compile the autoencoder.
if optimizer == 'adam':
    opt = tf.keras.optimizers.Adam(lr=learning_rate)
if optimizer == 'sgd':
    opt = tf.keras.optimizers.SGD(learning_rate=learning_rate, momentum=0.1)
if optimizer == 'adadelta':
    opt = tf.keras.optimizers.Adadelta(learning_rate=learning_rate)
autoencoder.compile(optimizer=opt, loss=loss)
latent_shape = tuple(encoder.layers[-1].output_shape[1:4])
```

```
(4, 4, 2)
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d_2 (Conv2D)           (None, 28, 28, 16)        160

 max_pooling2d_2 (MaxPooling  (None, 14, 14, 16)       0
 2D)

 conv2d_3 (Conv2D)           (None, 14, 14, 8)         1160

 max_pooling2d_3 (MaxPooling  (None, 7, 7, 8)          0
 2D)

 conv2d_4 (Conv2D)           (None, 7, 7, 2)           146

 max_pooling2d_4 (MaxPooling  (None, 4, 4, 2)          0
 2D)

 conv2d_5 (Conv2D)           (None, 4, 4, 2)           38

 up_sampling2d (UpSampling2D  (None, 8, 8, 2)          0
 )

 conv2d_6 (Conv2D)           (None, 8, 8, 8)           152

 up_sampling2d_1 (UpSampling  (None, 16, 16, 8)        0
 2D)

 conv2d_7 (Conv2D)           (None, 14, 14, 16)        1168

 up_sampling2d_2 (UpSampling  (None, 28, 28, 16)       0
 2D)

 conv2d_8 (Conv2D)           (None, 28, 28, 1)         145

=================================================================
Total params: 2,969
Trainable params: 2,969
Non-trainable params: 0
_________________________________________________________________
```

We can now train the autoencoder, by using the images as the input and output of the model.

In [8]:

```
# Train the autoencoder.
training = autoencoder.fit(x_train, x_train, validation_data=(x_validation, x_validation), epochs=epochs, batch_size=batch_siz
```

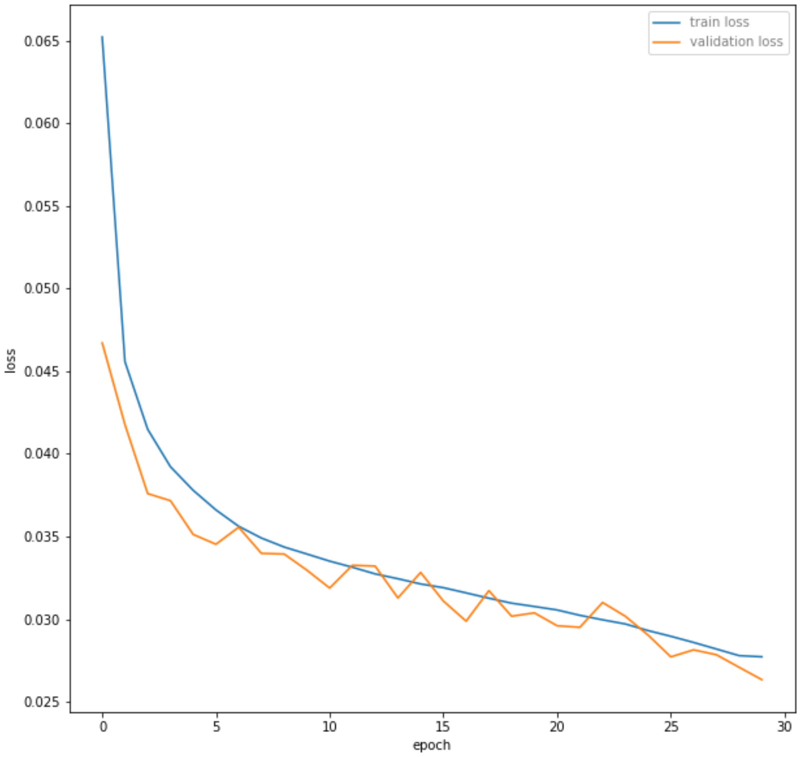
```
Epoch 1/30
469/469 [==============================] - 5s 7ms/step - loss: 0.0652 - val_loss: 0.0467
Epoch 2/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0456 - val_loss: 0.0418
Epoch 3/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0415 - val_loss: 0.0376
Epoch 4/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0392 - val_loss: 0.0372
Epoch 5/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0378 - val_loss: 0.0351
Epoch 6/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0366 - val_loss: 0.0345
Epoch 7/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0356 - val_loss: 0.0356
Epoch 8/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0349 - val_loss: 0.0340
Epoch 9/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0344 - val_loss: 0.0339
Epoch 10/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0339 - val_loss: 0.0330
Epoch 11/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0335 - val_loss: 0.0319
Epoch 12/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0331 - val_loss: 0.0333
Epoch 13/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0327 - val_loss: 0.0332
Epoch 14/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0324 - val_loss: 0.0313
Epoch 15/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0321 - val_loss: 0.0328
Epoch 16/30
469/469 [==============================] - 3s 5ms/step - loss: 0.0319 - val_loss: 0.0311
Epoch 17/30
469/469 [==============================] - 3s 5ms/step - loss: 0.0316 - val_loss: 0.0299
Epoch 18/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0313 - val_loss: 0.0317
Epoch 19/30
469/469 [==============================] - 2s 5ms/step - loss: 0.0310 - val_loss: 0.0302
```

```
Epoch 20/30
469/469 [==============================] - 3s 5ms/step - loss: 0.0308 - val_loss: 0.0304
Epoch 21/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0306 - val_loss: 0.0296
Epoch 22/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0302 - val_loss: 0.0295
Epoch 23/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0300 - val_loss: 0.0310
Epoch 24/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0297 - val_loss: 0.0302
Epoch 25/30
469/469 [==============================] - 3s 5ms/step - loss: 0.0293 - val_loss: 0.0290
Epoch 26/30
469/469 [==============================] - 3s 5ms/step - loss: 0.0290 - val_loss: 0.0277
Epoch 27/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0286 - val_loss: 0.0281
Epoch 28/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0282 - val_loss: 0.0278
Epoch 29/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0278 - val_loss: 0.0271
Epoch 30/30
469/469 [==============================] - 3s 6ms/step - loss: 0.0277 - val_loss: 0.0263
```

In [9]:
```
# Encode and then decode some test data.
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

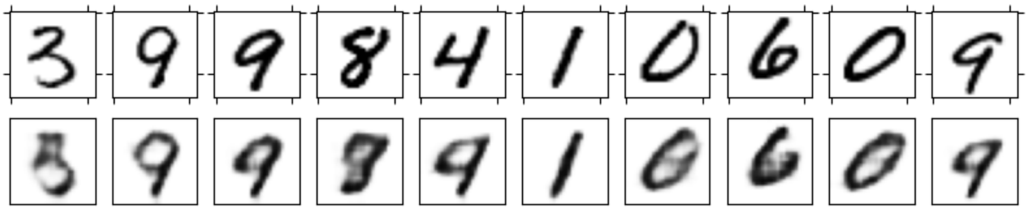
Plotting the training we can see that the model is not overfitting, and that further epochs would improve model performance.

In [10]:
```
# plotting the training
train_loss = np.array(training.history['loss'])
test_loss = np.array(training.history['val_loss'])
plt.rcParams['figure.figsize'] = [10, 10]
plt.plot(train_loss, label='train loss')
plt.plot(test_loss, label='validation loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.savefig('training.pdf')
```



We can now investigate how well the decompressed images recover the original image for some test examples:

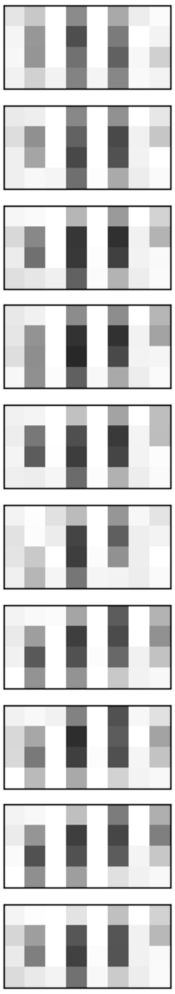
In [11]:
```
# Plot the decoded test data.
n = 10
vmax = np.max(np.abs(x_test))
vmin = -np.max(np.abs(x_test))
plt.figure(figsize=(10, 2), dpi=100)
for i in range(10):
    ax = plt.subplot(2, n, i+1)
    plt.imshow(x_test[i,:,:,0], norm=plt.Normalize(0,vmax), cmap=matplotlib.cm.binary, interpolation='hamming')
    plt.tick_params(axis='both', which='both', bottom='off', top='off', labelbottom='off', right='off', left='off', labelleft=
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax = plt.subplot(2, n, i+n+1)
    plt.imshow(decoded_imgs[i,:,:,0], norm=plt.Normalize(0,vmax), cmap=matplotlib.cm.binary, interpolation='hamming')
    plt.tick_params(axis='both', which='both', bottom='off', top='off', labelbottom='off', right='off', left='off', labelleft=
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])
plt.savefig('auto_predictions.pdf')
```



We can also visualise the compressed images for each case.

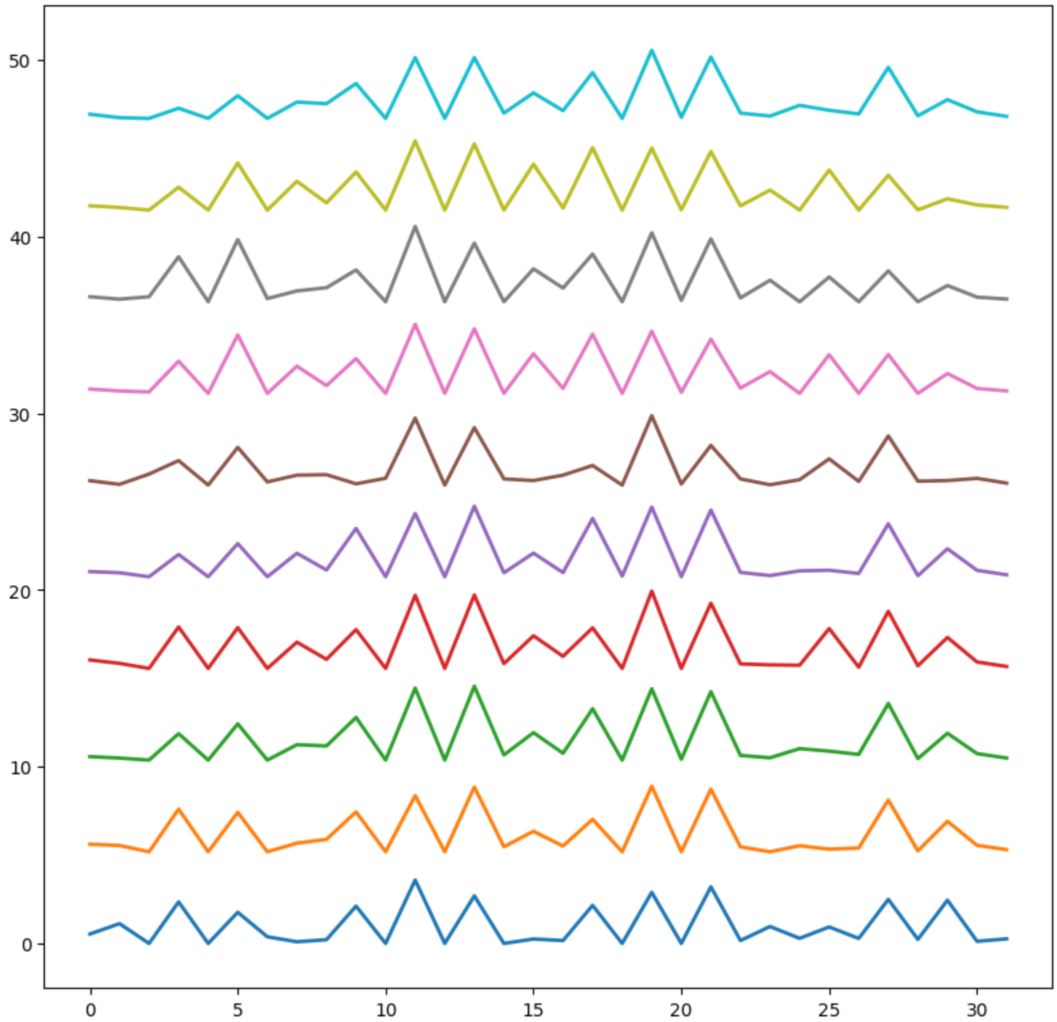
In [12]:

```
# Plot the latent spaces.
n = 10
vmax = np.max(np.abs(encoded_imgs))
vmin = -np.max(np.abs(encoded_imgs))
plt.figure(figsize=(10, 10), dpi=100)
for i in range(10):
    ax = plt.subplot(n, 1, i + 1)
    plt.imshow(np.reshape(encoded_imgs[i,:,:,:],(latent_shape[0],latent_shape[1]*latent_shape[2])), norm=plt.Normalize(0,vmax)
    plt.tick_params(axis='both', which='both', bottom='off', top='off', labelbottom='off', right='off', left='off', labelleft=
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.set_xticks([])
    ax.set_yticks([])
plt.savefig('latent.pdf')
```



In [13]:

```
# Plot the flat latent spaces.
n = 10
shift = np.max(encoded_imgs)
plt.figure(figsize=(10, 10), dpi=100)
for i in range(10):
    plt.plot(np.reshape(encoded_imgs[i,:,:,:],(latent_shape[0]*latent_shape[1]*latent_shape[2]))+i*shift, label='${0}$'.format
plt.savefig('latent_flat.pdf')
```



3. Solving the Schrödinger Equation using Deep Learning

In the following code, we use the dataset of 100,000 one electron systems to train a deep neural network to solve the 1D one electron Schrödinger Equation for the ground state energy.

The dataset contains the ground-state charge density (black) and total energy (blue line) for 100,000 randomly generated external potentials (red) on a grid containing 64 points:

In the following code we load in the data, perform the usual train, validation and testing split.

Then we build a convolutional (1D) network to predict the total energy from the external potential $E[V_\mathrm{ext}(x)]$

In [14]:
```
# Load in the dataset.
V = pickle.load(open('V.db','rb'))
n = pickle.load(open('density.db','rb'))
E = pickle.load(open('E.db','rb'))


# Split the dataset into training, validation and testing data.
V = V.reshape(V.shape + (1,))
n = n.reshape(n.shape + (1,))
E = E.reshape(E.shape + (1,))
V_train = V[:50000,...]
V_validation = V[50000:75000,...]
V_test = V[75000:,...]
n_train = n[:50000,...]
n_validation = n[50000:75000,...]
n_test = n[75000:,...]
E_train = E[:50000,...]
E_validation = E[50000:75000,...]
E_test = E[75000:,...]
print(V.shape, n.shape, E.shape)


# Build the neural network.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv1D(filters=6, kernel_size=(3), activation='relu', input_shape=(64, 1)))
model.add(tf.keras.layers.MaxPooling1D())
model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=(3), activation='relu'))
model.add(tf.keras.layers.MaxPooling1D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=120, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))
```

```
model.add(tf.keras.layers.Dense(units=50, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(units=1, activation='linear'))


# Compile the model.
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001), loss='mse')


# Train the model.
training = model.fit(V_train, E_train, validation_data=(V_validation, E_validation), epochs=50, batch_size=250, shuffle=True)


# Evaluate the model.
model.evaluate(V_test, E_test, verbose=2)


# Predict using the model.
N = 25 # Number of test images to predict
E_pred = model.predict(V_test[:N, :, :])
```

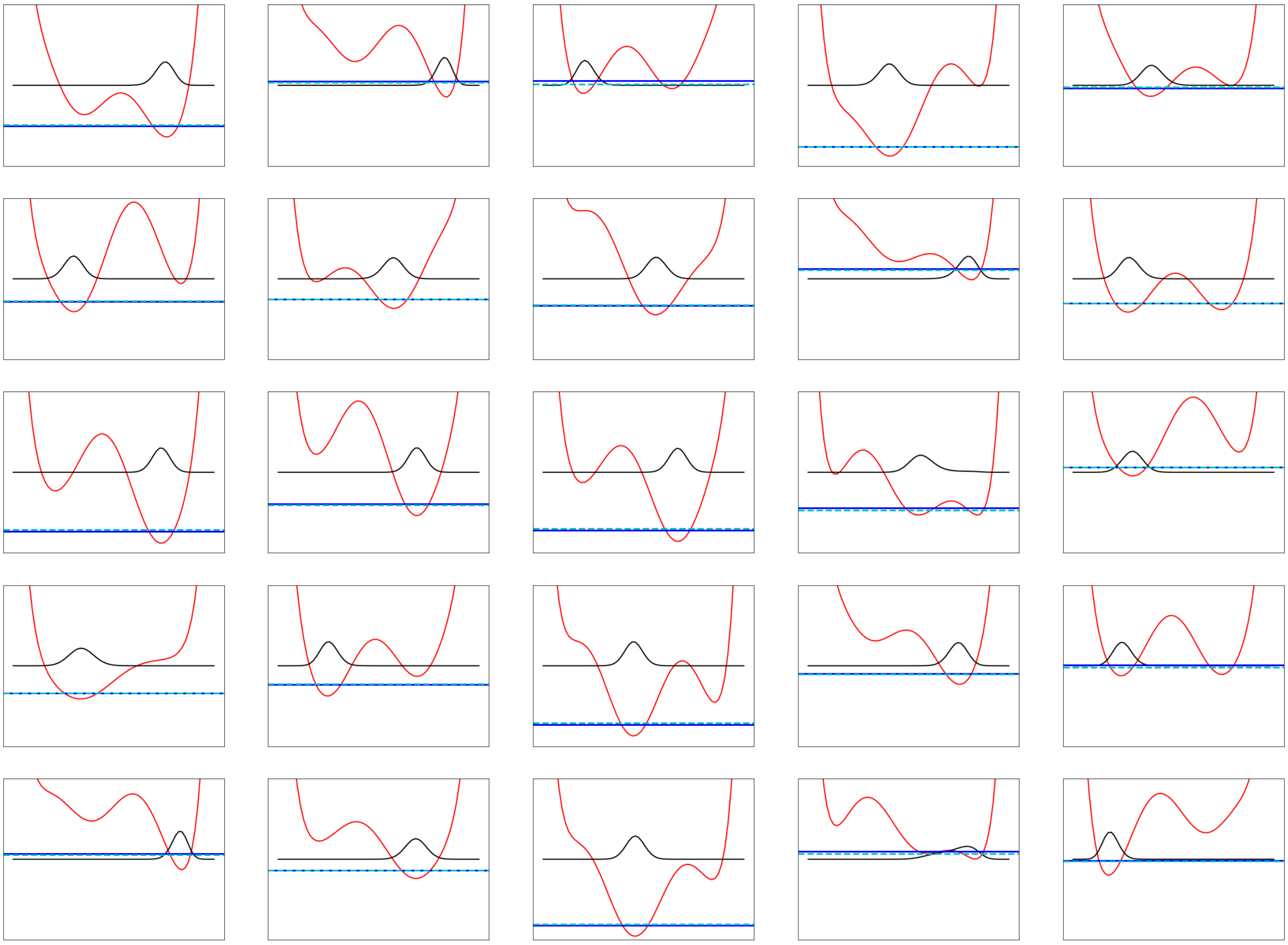
```
(100000, 64, 1) (100000, 64, 1) (100000, 1)
Epoch 1/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0280 - val_loss: 0.0043
Epoch 2/50
200/200 [==============================] - 1s 4ms/step - loss: 0.0112 - val_loss: 0.0015
Epoch 3/50
200/200 [==============================] - 1s 4ms/step - loss: 0.0089 - val_loss: 0.0016
Epoch 4/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0076 - val_loss: 0.0011
Epoch 5/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0068 - val_loss: 0.0012
Epoch 6/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0064 - val_loss: 0.0011
Epoch 7/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0059 - val_loss: 8.9242e-04
Epoch 8/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0058 - val_loss: 5.8070e-04
Epoch 9/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0056 - val_loss: 5.5265e-04
Epoch 10/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0055 - val_loss: 4.6677e-04
Epoch 11/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0055 - val_loss: 4.0945e-04
Epoch 12/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0052 - val_loss: 3.5810e-04
Epoch 13/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0052 - val_loss: 7.1015e-04
Epoch 14/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0052 - val_loss: 6.8156e-04
Epoch 15/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0052 - val_loss: 7.3587e-04
Epoch 16/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0051 - val_loss: 5.0488e-04
Epoch 17/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0051 - val_loss: 0.0011
Epoch 18/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0050 - val_loss: 6.9003e-04
Epoch 19/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0049 - val_loss: 4.1594e-04
Epoch 20/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0049 - val_loss: 3.2133e-04
Epoch 21/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0049 - val_loss: 6.2494e-04
Epoch 22/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0049 - val_loss: 2.9447e-04
Epoch 23/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0049 - val_loss: 4.6939e-04
Epoch 24/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0049 - val_loss: 4.5574e-04
Epoch 25/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0048 - val_loss: 5.3806e-04
Epoch 26/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0048 - val_loss: 8.1652e-04
Epoch 27/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0047 - val_loss: 4.8099e-04
Epoch 28/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0047 - val_loss: 5.7752e-04
Epoch 29/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0048 - val_loss: 3.4765e-04
Epoch 30/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0047 - val_loss: 3.4241e-04
Epoch 31/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0047 - val_loss: 7.2721e-04
Epoch 32/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0048 - val_loss: 5.5564e-04
Epoch 33/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0046 - val_loss: 4.7157e-04
Epoch 34/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0046 - val_loss: 5.1059e-04
Epoch 35/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0047 - val_loss: 5.1871e-04
Epoch 36/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0047 - val_loss: 2.8154e-04
Epoch 37/50
200/200 [==============================] - 0s 3ms/step - loss: 0.0046 - val_loss: 6.0481e-04
Epoch 38/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 3.1901e-04
Epoch 39/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 5.9210e-04
Epoch 40/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 3.2698e-04
Epoch 41/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 2.9606e-04
Epoch 42/50
```

```
200/200 [==============================] - 0s 2ms/step - loss: 0.0046 - val_loss: 3.2364e-04
Epoch 43/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0047 - val_loss: 4.7431e-04
Epoch 44/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0047 - val_loss: 5.5140e-04
Epoch 45/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 2.8999e-04
Epoch 46/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 6.8349e-04
Epoch 47/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0046 - val_loss: 2.7814e-04
Epoch 48/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 4.0985e-04
Epoch 49/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 7.3741e-04
Epoch 50/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0046 - val_loss: 2.4619e-04
782/782 - 1s - loss: 2.4407e-04 - 1s/epoch - 2ms/step
```

After training the model, we can plot the predictions (dotted cyan) against the true values of the energy (blue) for 25 of the test systems:

In [15]:

```
# Plot some results.
plt.rcParams['figure.figsize'] = [40, 30]
fig, axs = plt.subplots(5, 5)
k = 0
for i in range(axs.shape[0]):
    for j in range(axs.shape[1]):
        axs[i, j].plot(V_test[k, :, :].reshape(64,), 'r-', linewidth=2.0)
        axs[i, j].plot(n_test[k, :, :].reshape(64,), 'k-', linewidth=2.0)
        axs[i, j].axhline(E_test[k, 0], color='b', linewidth=3.0)
        axs[i, j].axhline(E_pred[k, 0], color='c', linestyle='--', linewidth=3.0)
        axs[i, j].set_ylim([-1,1])
        axs[i, j].set_yticklabels([])
        axs[i, j].set_xticklabels([])
        axs[i, j].set_yticks([])
        axs[i, j].set_xticks([])
        k = k + 1
plt.savefig('v_predictions.pdf')
```



4. Learning the functional of the charge density.

We can now begin to investigate in a simple way, **of which quantity is it best to build a given functional from?**. We can take the *identical model* from above, that we used to train $E[V_\mathrm{ext}(x)]$, and use it to instead train $E[n(x)]$. Does this model yield better results? Is it harder to train?

In [16]:

```
# Build the neural network.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv1D(filters=6, kernel_size=(3), activation="relu", input_shape=(64, 1)))
model.add(tf.keras.layers.MaxPooling1D())
model.add(tf.keras.layers.Conv1D(filters=16, kernel_size=(3), activation="relu"))
model.add(tf.keras.layers.MaxPooling1D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=120, activation="relu"))
```

```python
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(units=50, activation="relu"))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(units=1, activation="linear"))


# Complile the model.
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss="mse")


# Train the model.
training = model.fit(n_train, E_train, validation_data=(n_validation, E_validation), epochs=50, batch_size=250, shuffle=True)


# Evaluate the model.
model.evaluate(n_test, E_test, verbose=2)


# Predict using the model.
N = 25 # Number of test images to predict
E_pred = model.predict(n_test[:N, :, :])
```

```
Epoch 1/50
200/200 [==============================] - 1s 3ms/step - loss: 0.1051 - val_loss: 0.0791
Epoch 2/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0877 - val_loss: 0.0779
Epoch 3/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0851 - val_loss: 0.0771
Epoch 4/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0827 - val_loss: 0.0760
Epoch 5/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0808 - val_loss: 0.0752
Epoch 6/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0797 - val_loss: 0.0742
Epoch 7/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0780 - val_loss: 0.0730
Epoch 8/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0765 - val_loss: 0.0716
Epoch 9/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0751 - val_loss: 0.0703
Epoch 10/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0733 - val_loss: 0.0691
Epoch 11/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0724 - val_loss: 0.0682
Epoch 12/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0713 - val_loss: 0.0676
Epoch 13/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0706 - val_loss: 0.0665
Epoch 14/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0701 - val_loss: 0.0664
Epoch 15/50
200/200 [==============================] - 1s 4ms/step - loss: 0.0694 - val_loss: 0.0659
Epoch 16/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0689 - val_loss: 0.0659
Epoch 17/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0686 - val_loss: 0.0653
Epoch 18/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0683 - val_loss: 0.0650
Epoch 19/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0681 - val_loss: 0.0649
Epoch 20/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0677 - val_loss: 0.0651
Epoch 21/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0676 - val_loss: 0.0647
Epoch 22/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0676 - val_loss: 0.0647
Epoch 23/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0674 - val_loss: 0.0647
Epoch 24/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0674 - val_loss: 0.0646
Epoch 25/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0672 - val_loss: 0.0648
Epoch 26/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0670 - val_loss: 0.0647
Epoch 27/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0672 - val_loss: 0.0646
Epoch 28/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0670 - val_loss: 0.0644
Epoch 29/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0669 - val_loss: 0.0645
Epoch 30/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0667 - val_loss: 0.0644
Epoch 31/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0668 - val_loss: 0.0644
Epoch 32/50
200/200 [==============================] - 1s 3ms/step - loss: 0.0667 - val_loss: 0.0644
Epoch 33/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0666 - val_loss: 0.0643
Epoch 34/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0649
Epoch 35/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0666 - val_loss: 0.0651
Epoch 36/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0666 - val_loss: 0.0644
Epoch 37/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0666 - val_loss: 0.0643
Epoch 38/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0666 - val_loss: 0.0644
Epoch 39/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0643
Epoch 40/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0645
Epoch 41/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0646
Epoch 42/50
```

```
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0644
Epoch 43/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0643
Epoch 44/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0662 - val_loss: 0.0645
Epoch 45/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0662 - val_loss: 0.0647
Epoch 46/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0664 - val_loss: 0.0642
Epoch 47/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0663 - val_loss: 0.0642
Epoch 48/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0662 - val_loss: 0.0642
Epoch 49/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0662 - val_loss: 0.0647
Epoch 50/50
200/200 [==============================] - 0s 2ms/step - loss: 0.0662 - val_loss: 0.0649
782/782 - 1s - loss: 0.0643 - 1s/epoch - 1ms/step
```

We can see that building a funtional of the density is more difficult for the machine to learn for an equivelent mode, why is this? (For example, a shift in the potential shifts the energy, but has no effect on the density.)

Task: Can you modify the code above the produce a more accurate functional when applied to the test systems. Consider modifying the parameters of the model, such as the kernals of the convolutions, the number of dense layers, the learning rate, number of epochs ect...

In [17]:

```python
# Plot some results.
plt.rcParams['figure.figsize'] = [40, 30]
fig, axs = plt.subplots(5, 5)
k = 0
for i in range(axs.shape[0]):
    for j in range(axs.shape[1]):
        axs[i, j].plot(V_test[k, :, :].reshape(64,), 'r-', linewidth=2.0)
        axs[i, j].plot(n_test[k, :, :].reshape(64,), 'k-', linewidth=2.0)
        axs[i, j].axhline(E_test[k, 0], color='b', linewidth=3.0)
        axs[i, j].axhline(E_pred[k, 0], color='c', linestyle='--', linewidth=3.0)
        axs[i, j].set_ylim([-1,1])
        axs[i, j].set_yticklabels([])
        axs[i, j].set_xticklabels([])
        axs[i, j].set_yticks([])
        axs[i, j].set_xticks([])
        k = k + 1
plt.savefig('n_predictions.pdf')
```