Run in Google Colab

Hackathon - NMR Challenge

Problem Statement

- Nuclear Magnetic Resonance (NMR) is an experimental technique that allows for the control and measurement of nuclear spins in crystals and molecules.
- A common "recipe" for NMR is called the spin echo: the spins start aligned, begin to disperse, and are then refocused. This creates a sharp peak, or "echo", in the net magnetization M of the material at a later time. When the spins interact with each other, this refocused echo can become highly distorted.
- Materials with strong electron-electron couplings have a variety of applications, from superconductivity to ferromagnetism. They also tend to enhance the nuclear spin-spin couplings, allowing NMR to act as a probe of these important systems.
- Design and train a model that predicts the strength and shape of interactions between the nuclear spins from simulated time-dependent magnetization curves, $M(t)$.

Before getting to any code, we first review the structure of this machine learning problem and introduce some of the details of the underlying physics we are trying to capture.

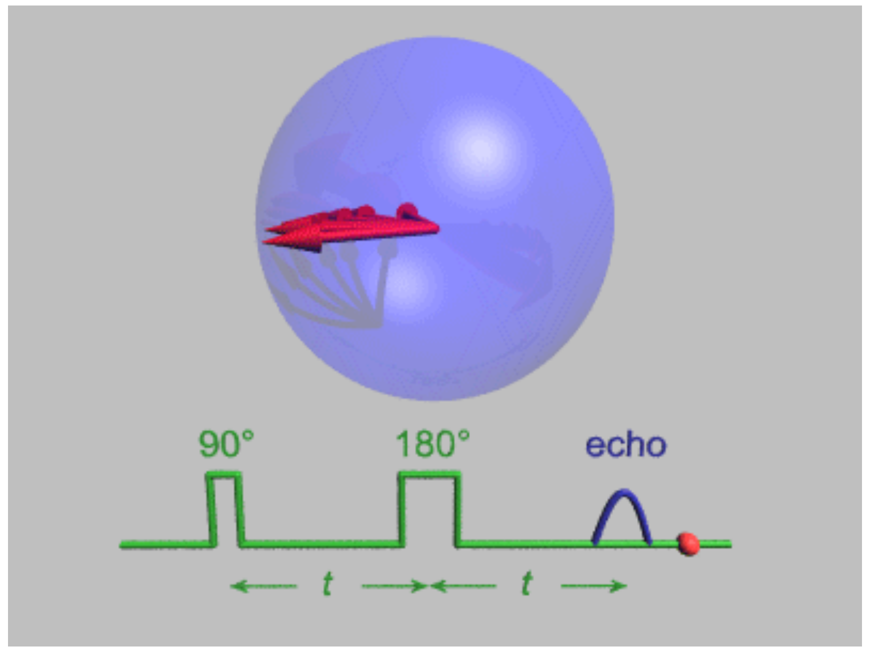
Quick description of the ML problem

Goal:

Predict three real numbers from an input vector of complex numbers.

Introduction to NMR and spin echos

Although the NMR "spin echo" technique may sound complicated, the following animation created by Gavin W Morley (by way of https://en.wikipedia.org/wiki/Spin_echo) makes it much clearer!



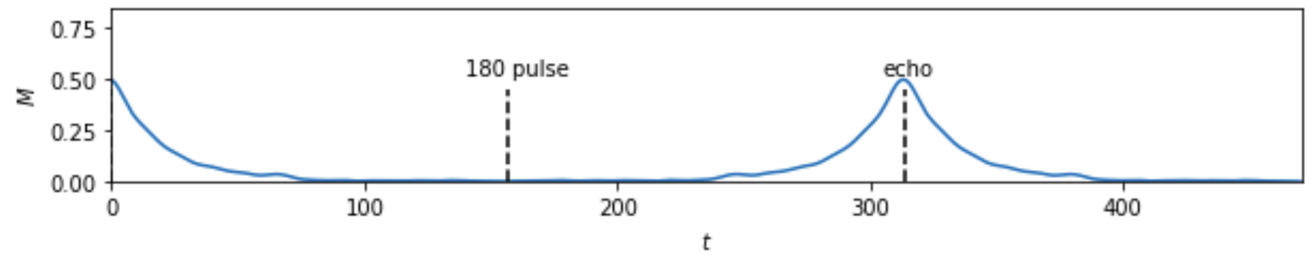
The red arrows in this animation represents the values of nuclear spins in the material. They all begin in the same direction (up), and then an applied magnetic field rotates them into the x-y plane (indicated by the 90° pulse). A constant external magnetic field in the z-direction did not affect the spins when they were pointing "up", but now that they lie in the x-y plane they begin to precess.

Because each nuclear spin sits in a slightly different magnetic environment, each one has a slightly different response to the background z-direction magnetic field, causing some to precess in a clockwise direction and others in a counterclockwise direction.

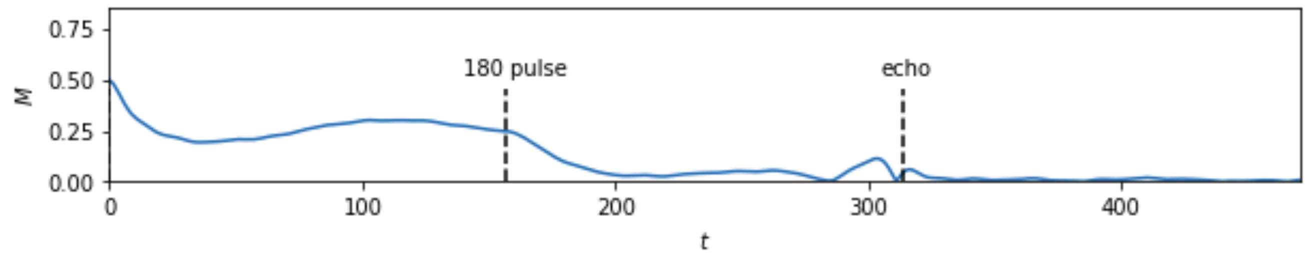
After a fixed amount of time, t in the above animation, a second magnetic pulse is applied and rotates each spin 180° in the x-y plane. After this, the spins continue to move as they did before, but because of the 180° pulse they are now effectively precessing "backwards" compared to the original motion! So after an additional time t passes, the variations in precession time is canceled out, causing a refocusing of the spins.

This shows up as a measurable "echo" in the average spin magnetization of the material, and can be measured in experiments. This is an important technique because the average spin magnetization is hard to measure during an applied "pulse", but there is no external pulse during the "echo", allowing for accurate measurement of the peak value and decay shape.

Here is a typical curve for the time-dependent magnetization $M(t)$ for a spin-echo in most materials:



Sometimes, a more complicated curve can occur, such as:



This more complicated structure has been caused by spin-spin interactions between the precessing nuclear spins. Normally, each spin precesses in a uniform way irrespective of the rest of the nuclei in the material. In this coupled case, however, the nuclear magnetization that occurs near the "echo" influences the spins' motions, modifying the shape of the observed echo.

Electronic and nuclear spins

Most materials can be classified by their electronic properties into three categories: metal, insulator, and semiconductor. These terms are based on a semi-classical description of the electrons in a crystal. The electrons are treated as a collection of classical particles, with energies that depend on their momentum in a way determined by the atomic structure of the crystal.

However, there are other electronic phases of matter that are truly "quantum" and cannot be described accurately with a classical analogy. In these scenarios, complicated structures in the electron states can give rise to large electronic spin density or strong electron-electron coupling. Because of these strong couplings between electrons, they are often hard to probe experimentally.

Luckily, electrons can interact with the nuclar spins of a material (by way of the hyperfine-interaction). If the electron-nuclear coupling becomes strong enough, then a non-neglible two-step process can couple the nuclei with each other throughout the material. That two-step process is when a nuclear spin couples to an electron and changes its motion, and then that electron later "scatters" off another nuclear spin elsewhere in the material.

We represent this two-step scattering prcoess by way of an effective spin-spin coupling between a nuclei at position r_j and r_i . **There are two datasets, "gauss" and "RKKY", and thus you will have to generate TWO models and hand in two models.**

The first is a simple gaussian function ("gauss"):

$$T_1(i,j) = \alpha \exp\left[\left(\frac{-|r_j - r_i|}{\xi}\right)^2\right]$$

And the second is the traditional Ruderman–Kittel–Kasuya–Yosida function ("RKKY"):

$$T_2(i,j) = \alpha x^{-4}\left(x \cos x - \sin x\right)$$

with $x = 2\frac{|r_j - r_i|}{\xi}$

For both function, α is the coupling strength and ξ is the coupling length Generally, α and ξ will depend on the details of the nuclear-electron coupling and the quantum state of the electrons, but here we will sample them randomly to see if the spin-echo experiment can provide enough information to accurately "reverse engineer" these values from a single $M(t)$ curve.

Our simulations also include dissipation of the nuclear spins: due to couplings with the environment the spin information can be "lost". This occurs at a time scale $T_{\text{decay}} \simeq \Gamma^{-1/2}$, with Γ given by:

$$\Gamma = 10^{-d}$$

Our goal is to develop two models, one for each function, that accurately determine the above variables (α, ξ, and d) from a single $M(t)$ curve. Note that RKKY is a harder problem.

Load and view the simulated data

Three datafiles will be used for the training of both models. Each file has 6000 lines, representing 6000 simulated $M(t)$ curves for different choices of the three material parameters:

- <model name>_echos_model_r.txt : Real part of the time-dependent magnetization, $\mathrm{Re}(M(t))$.
- <model name>_echos_model_i.txt : Imaginary part of the time-dependent magnetization, $\mathrm{Im}(M(t))$.
- <model name>_mat_info_model.txt : The three material parameters (α,ξ,d) introduced above.

Where <model name> is either "gauss" or "RKKY".

We also load two other echo files, which give an additional 6000 $M(t)$ curves. These will be used to judge the quality of your final models:

- <model name>_echos_eval_r.txt
- <model name>_echos_eval_i.txt

```
import numpy as np
import matplotlib.pyplot as plt
import requests

print("Downloading files off google drive...")

f_prefix = "gauss"

# data for model creation
mat_file = f_prefix+"_mat_info_model.txt"
M_file_r = f_prefix+"_echos_model_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_model_i.txt" # imaginary part of echos

asdf = "https://drive.google.com/uc?export=download&id="

r = requests.get("https://drive.google.com/uc?export=download&id=1J8CcJVQRpzSwue1vuHV9uB0bngdDrKCY",allow_redirects=True)
open(mat_file, "wb").write(r.content)
r = requests.get("https://drive.google.com/uc?export=download&id=1lBWcwF--1rrB8KCyCd0-5ZnPIjRrWkHg",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://drive.google.com/uc?export=download&id=1O7KKL-SW3vHePoRNk8YfLzX82wf2Z5ul",allow_redirects=True)
open(M_file_i, "wb").write(r.content)

# data for submission of final model
M_file_r = f_prefix+"_echos_eval_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_eval_i.txt" # imaginary part of echos

r = requests.get("https://drive.google.com/uc?export=download&id=1prIrtO7XJs3PBe1MZiWUxK3VUkrChVvz",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://drive.google.com/uc?export=download&id=1vbKcuxe6z8cRGQdTqj_Q2u5Oow0D9hbU",allow_redirects=True)
open(M_file_i, "wb").write(r.content)

# now repeat, but for RKKY type function

f_prefix = "RKKY"

# data for model creation
mat_file = f_prefix+"_mat_info_model.txt"
M_file_r = f_prefix+"_echos_model_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_model_i.txt" # imaginary part of echos
```

```
r = requests.get("https://docs.google.com/uc?export=download&id=1lS9AJ3sUFI4cfM5jQj618x4shoaJMXVo",allow_redirects=True)
open(mat_file, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1J21bKy8FTjoaGzHVdLXlWAao2UiWO7ml",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1nf3Y_FcJJEWXJbjwREAkgcnVz2tDA__I",allow_redirects=True)
open(M_file_i, "wb").write(r.content)

# data for submission of final model
M_file_r = f_prefix+"_echos_eval_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_eval_i.txt" # imaginary part of echos

r = requests.get("https://docs.google.com/uc?export=download&id=1Q46o_RnYZFWEjMVVF5m1VBI9HCltspyY",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1-z2ADFrBlEhXN5Z_LHiRLA4Nds_9uvQq",allow_redirects=True)
open(M_file_i, "wb").write(r.content)


print("Done with file downloads")
```

```
    Downloading files off google drive...
    Done with file downloads
```

Change the following "f_prefix" variable to select a different model to load and train on

```
f_prefix = "gauss"; # Gaussian functional between nuclei
# f_prefix = "RKKY"; # RKKY functional between nuclei
```

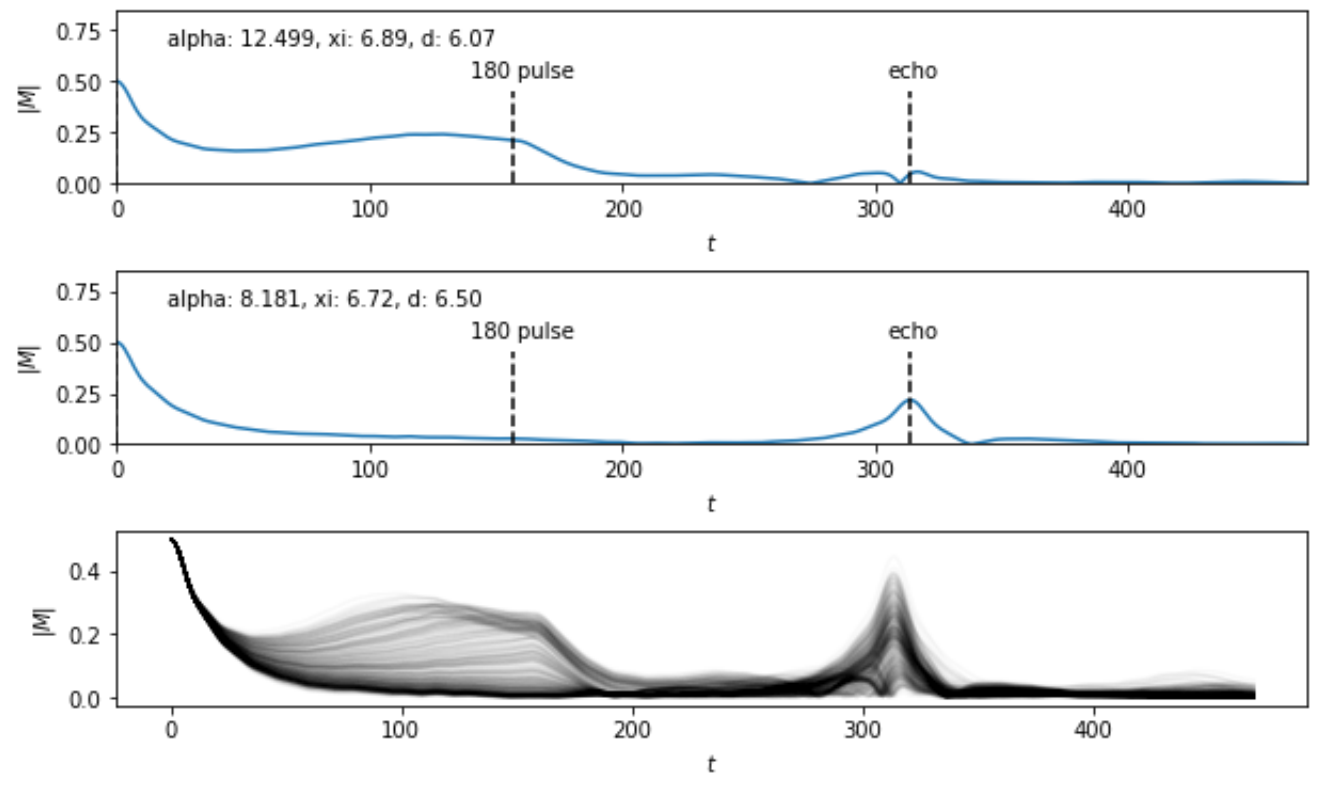
Now load the data and format it correctly

```
mat_file = f_prefix+"_mat_info_model.txt"
M_file_r = f_prefix+"_echos_model_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_model_i.txt" # imaginary part of echos

print("Loading into numpy arrays...")
# settings of each simulated material:
# format:  |   α  |   ξ  |   d  |
mat_info = np.loadtxt(mat_file, comments="#", delimiter=None, unpack=False);

# M(t) curve for each simulation, model:
M_r = np.loadtxt(M_file_r, comments="#", delimiter=None, unpack=False);
M_i = np.loadtxt(M_file_i, comments="#", delimiter=None, unpack=False);
M = M_r + 1j*M_i;

# M(t) curve for each simulation, eval:
M_file_r = f_prefix+"_echos_eval_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_eval_i.txt" # imaginary part of echos

M_r_eval = np.loadtxt(M_file_r, comments="#", delimiter=None, unpack=False);
M_i_eval = np.loadtxt(M_file_i, comments="#", delimiter=None, unpack=False);
M_eval = M_r_eval + 1j*M_i_eval;

print("Done with numpy loads")
```

```
    Loading into numpy arrays...
    Done with numpy loads
```

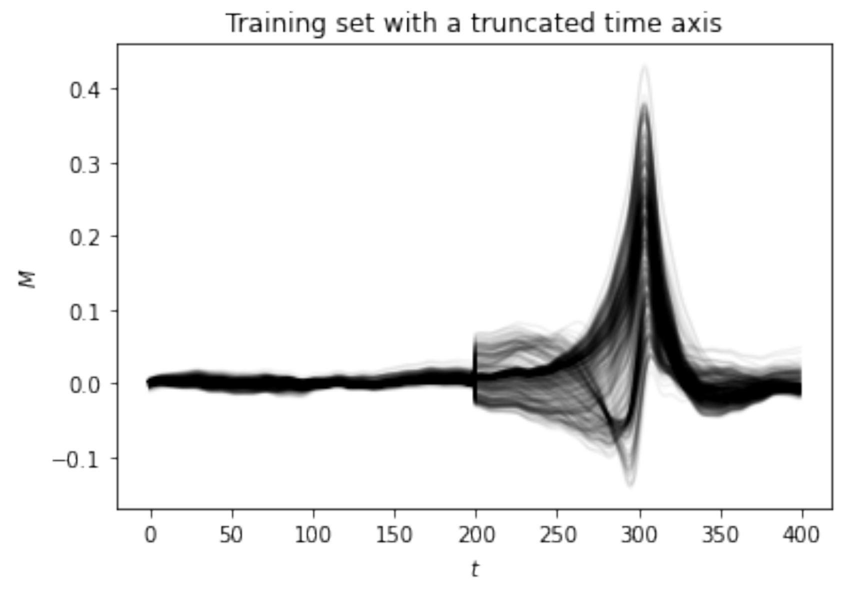
View the data with three plots, two with a specific curve and one with a lot of curves

```
fig1, ax1 = plt.subplots(3,1, figsize=(10,6));

# change the following to see different curves
plot_idx1 = 0; # weak spin-spin coupling
plot_idx2 = 10; # strong spin-spin coupling

# string format for material parameter plotting
mat_format = "alpha: %.3f, xi: %.2f, d: %.2f";

# view the selected curve, with a label of the material data
ax1[0].plot(abs(M[plot_idx1,:]));
ax1[0].text(20,0.68, mat_format % tuple(mat_info[plot_idx1,:]) );
ax1[0].plot([0, 0],[0, .45],'--k')
ax1[0].plot([157, 157],[0, .45],'--k')
ax1[0].text(140,0.52,"180 pulse")
ax1[0].text(305,0.52,"echo")
ax1[0].plot([2*157, 2*157],[0, .45],'--k')
ax1[0].axis([0, 471, 0, 0.85])
ax1[0].set(ylabel="$|M|$", xlabel="$t$");

# view the selected curve, with a label of the material data
ax1[1].plot(abs(M[plot_idx2,:]));
ax1[1].text(20,0.68, mat_format % tuple(mat_info[plot_idx2,:]) );
ax1[1].plot([0, 0],[0, .45],'--k')
ax1[1].plot([157, 157],[0, .45],'--k')
ax1[1].text(140,0.52,"180 pulse")
ax1[1].text(305,0.52,"echo")
ax1[1].plot([2*157, 2*157],[0, .45],'--k')
ax1[1].axis([0, 471, 0, 0.85])
ax1[1].set(ylabel="$|M|$", xlabel="$t$");

ax1[2].plot(abs(M[1:500,:]).T,color=(0,0,0,.025));
ax1[2].set(ylabel="$|M|$", xlabel="$t$");

fig1.subplots_adjust(hspace=.5)
```



Truncate, scale, and partition the training/testing sets

```
# number of M(t) curves
N_data = np.shape(M)[0]
# truncate time points
# !!! NOTE: May want to use all of the curve, takes longer to train though !!!
time_keep = range(210,410); # centered roughly at the echo
M_trunc = M[:,time_keep];
# split into real and imaginary
M_trunc_uncomplex = np.concatenate((np.real(M_trunc), np.imag(M_trunc)),axis=1)

# rescale data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

mat_info_scaled = sc.fit_transform(mat_info);

# partition data into a training and testing set using a random partition
from sklearn.model_selection import train_test_split
M_train, M_test, mat_train, mat_test = train_test_split(M_trunc_uncomplex, mat_info_scaled, test_size=0.1)

# plot the fist 500 elements of the training set, for visualizing variations in the data
plt.plot((M_train[1:500,:]).T,color=(0,0,0,.05));
plt.xlabel("$t$")
plt.ylabel("$M$")
plt.title("Training set with a truncated time axis");
```



```
M_eval_complex = M_eval[:,time_keep];
M_eval = np.concatenate((np.real(M_eval_complex), np.imag(M_eval_complex)),axis=1)
M_eval.shape
```

```
    (6000, 400)
```

Example solution: a simple neural net (NN)

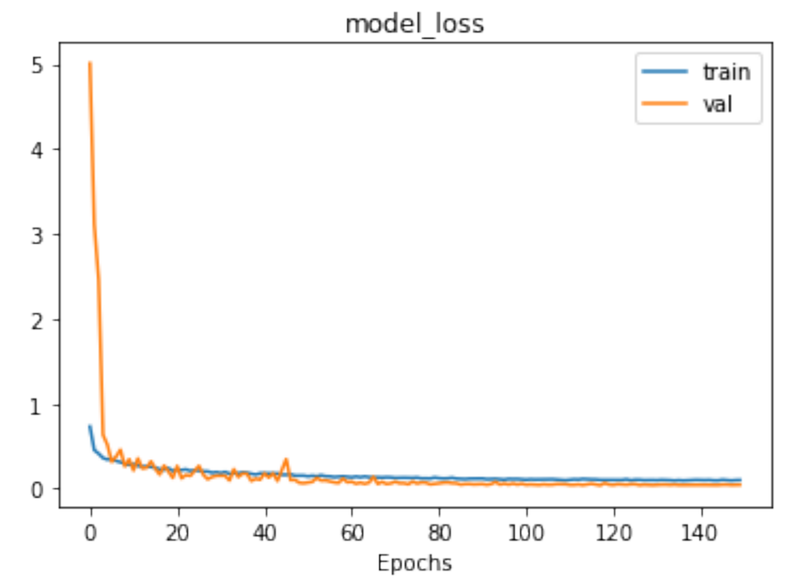
- Our input nodes are the vector $[\mathrm{Re}(M(t)), \mathrm{Im}(M(t))]$, which is a few hundred elements.
- Our output nodes are the three material parameters.
- We will use a standard NN predict the material properties from $M(t)$.

```
from keras.layers.normalization.batch_normalization import BatchNormalization
import tensorflow
import keras
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.optimizers import Nadam # gradient descent optimizer

# first we build the model

N = np.shape(M_train[0])[0] # number of input values from M(t) curve

# define the net
gauss = Sequential()
gauss.add(BatchNormalization())
gauss.add(Dense(200,input_dim=N, activation='relu', kernel_initializer="he_normal
gauss.add(BatchNormalization())
gauss.add(Dense(200, activation='relu', kernel_initializer="he_normal"))
gauss.add(BatchNormalization())
gauss.add(Dense(200, activation='selu',kernel_initializer="he_normal"))
gauss.add(BatchNormalization())
gauss.add(Dense(200, activation='tanh', kernel_initializer="he_normal"))
gauss.add(BatchNormalization())
gauss.add(Dense(200, activation='relu', kernel_initializer="he_normal"))
gauss.add(BatchNormalization())
gauss.add(Dense(200, activation='selu', kernel_initializer="he_normal"))
gauss.add(BatchNormalization())
gauss.add(Dense(200, activation='relu', kernel_initializer="he_normal"))
gauss.add(BatchNormalization())
gauss.add(Dense(3, activation='linear'))

gauss.compile(loss='mean_squared_error', optimizer='Nadam' )
# from keras.layers.normalization.batch_normalization import BatchNormalization
# import tensorflow
# import keras
# from keras.models import Sequential
# from keras.layers import Dense
# from tensorflow.keras.optimizers import Nadam # gradient descent optimizer

# # first we build the model

# N = np.shape(M_train[0])[0] # number of input values from M(t) curve

# # define the net
# RKKY = Sequential()
# # Let's try N -> 100 -> 40 -> 3, e.g. 2 hidden layers
# #keras.layers.BatchNormalization()
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(200,input_dim=N, activation='selu', kernel_initializer="he_normal"))
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(200, activation='relu', kernel_initializer="he_normal"))
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(200,input_dim=N, activation='selu', kernel_initializer="he_normal"))
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(200, activation='relu', kernel_initializer="he_normal"))
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(200, activation='selu', kernel_initializer="he_normal"))
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(200, activation='relu', kernel_initializer="he_normal"))
# RKKY.add(BatchNormalization())
# RKKY.add(Dense(3, activation='linear'))

# RKKY.compile(loss='mean_squared_error', optimizer='Nadam' )


from keras.callbacks import ReduceLROnPlateau
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.75, patience=5, min_delta=0.00001, cooldown=1, min_lr=1e-10, mode='auto')
checkpoint_cb = tensorflow.keras.callbacks.ModelCheckpoint("gauss.h5", save_best_only=True)
#early_stopping_cb = tensorflow.keras.callbacks.EarlyStopping(patience=5, restore_best_weights = True)
history=gauss.fit(M_train, mat_train,\
        batch_size=32,\
        epochs=150,\
        validation_data=(M_test,mat_test),\
        callbacks=[ reduce_lr, checkpoint_cb],# early_stopping_cb],\
        verbose=1, shuffle=True, initial_epoch=0
        )
```

```
    169/169 [==============================] - 3s 20ms/step - loss: 0.1061 - val_loss: 0.0439 - lr: 3.1676e-05
    Epoch 122/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1051 - val_loss: 0.0530 - lr: 3.1676e-05
    Epoch 123/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1030 - val_loss: 0.0499 - lr: 3.1676e-05
    Epoch 124/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1109 - val_loss: 0.0465 - lr: 2.3757e-05
    Epoch 125/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.0987 - val_loss: 0.0527 - lr: 2.3757e-05
    Epoch 126/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1058 - val_loss: 0.0536 - lr: 2.3757e-05
    Epoch 127/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1054 - val_loss: 0.0423 - lr: 2.3757e-05
    Epoch 128/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1006 - val_loss: 0.0474 - lr: 2.3757e-05
    Epoch 129/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1022 - val_loss: 0.0455 - lr: 1.7818e-05
    Epoch 130/150
    169/169 [==============================] - 4s 21ms/step - loss: 0.1021 - val_loss: 0.0422 - lr: 1.7818e-05
    Epoch 131/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1001 - val_loss: 0.0459 - lr: 1.7818e-05
    Epoch 132/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1042 - val_loss: 0.0477 - lr: 1.7818e-05
    Epoch 133/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1035 - val_loss: 0.0482 - lr: 1.7818e-05
    Epoch 134/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1004 - val_loss: 0.0449 - lr: 1.3363e-05
    Epoch 135/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.0958 - val_loss: 0.0485 - lr: 1.3363e-05
    Epoch 136/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1005 - val_loss: 0.0437 - lr: 1.3363e-05
    Epoch 137/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.0960 - val_loss: 0.0442 - lr: 1.3363e-05
    Epoch 138/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.0990 - val_loss: 0.0448 - lr: 1.3363e-05
    Epoch 139/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.0999 - val_loss: 0.0434 - lr: 1.0023e-05
    Epoch 140/150
    169/169 [==============================] - 4s 21ms/step - loss: 0.1040 - val_loss: 0.0446 - lr: 1.0023e-05
    Epoch 141/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1023 - val_loss: 0.0453 - lr: 1.0023e-05
    Epoch 142/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1012 - val_loss: 0.0448 - lr: 1.0023e-05
    Epoch 143/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.1034 - val_loss: 0.0429 - lr: 1.0023e-05
    Epoch 144/150
    169/169 [==============================] - 3s 20ms/step - loss: 0.0980 - val_loss: 0.0435 - lr: 7.5169e-06
    Epoch 145/150
    169/169 [==============================] - 3s 18ms/step - loss: 0.0973 - val_loss: 0.0432 - lr: 7.5169e-06
    Epoch 146/150
    169/169 [==============================] - 3s 18ms/step - loss: 0.1058 - val_loss: 0.0472 - lr: 7.5169e-06
    Epoch 147/150
    169/169 [==============================] - 3s 18ms/step - loss: 0.1011 - val_loss: 0.0479 - lr: 7.5169e-06
    Epoch 148/150
    169/169 [==============================] - 3s 17ms/step - loss: 0.0964 - val_loss: 0.0440 - lr: 7.5169e-06
    Epoch 149/150
    169/169 [==============================] - 3s 18ms/step - loss: 0.1008 - val_loss: 0.0445 - lr: 5.6377e-06
    Epoch 150/150
    169/169 [==============================] - 3s 18ms/step - loss: 0.1023 - val_loss: 0.0461 - lr: 5.6377e-06
```

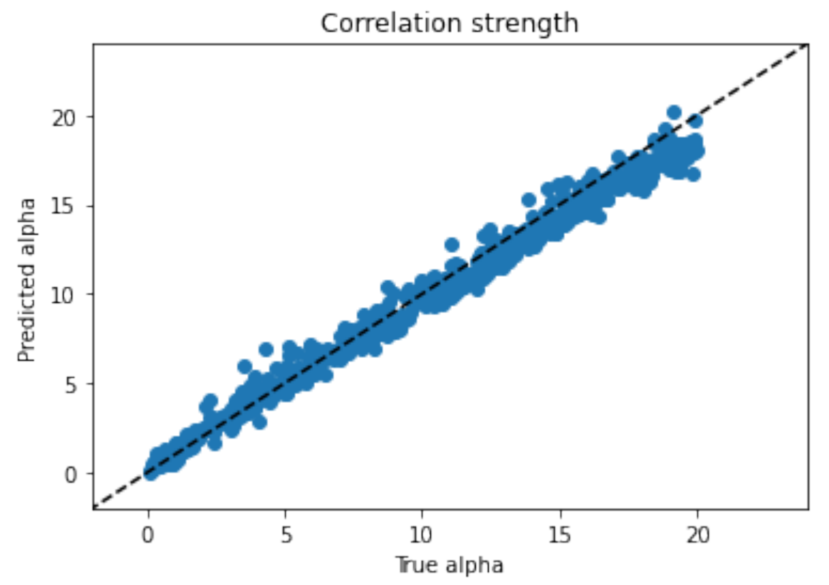
```
gauss = tensorflow.keras.models.load_model("gauss.h5")
```

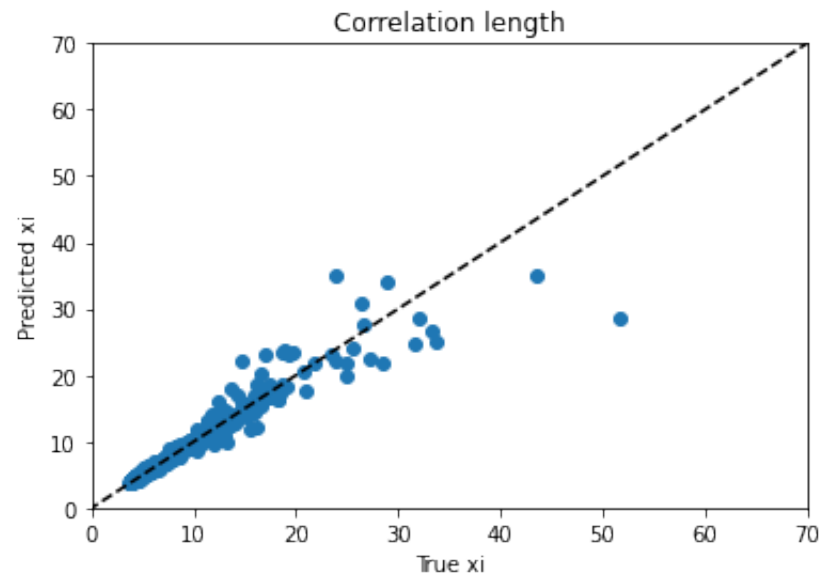
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model_loss')
plt.xlabel('Epochs')
plt.legend(['train','val'], loc='upper right')
plt.show()
```

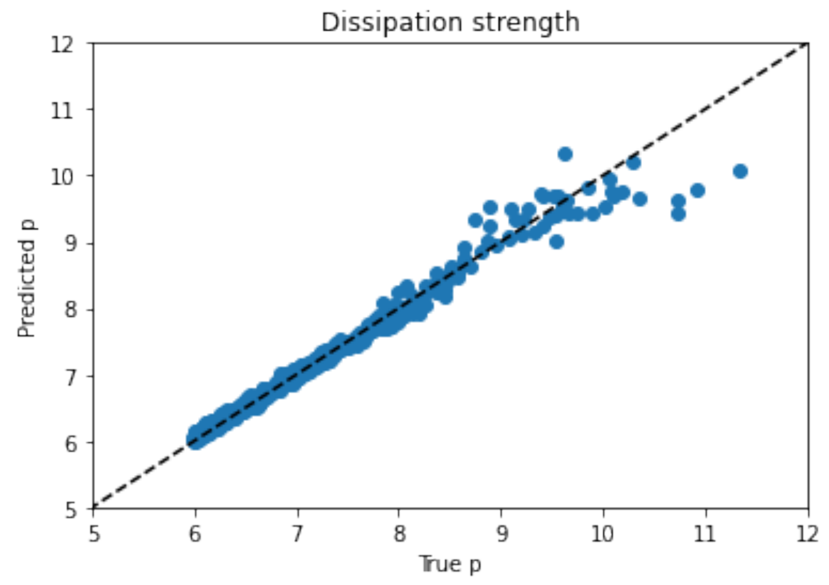


```
# check results on test set

results = gauss.evaluate(M_test,mat_test, batch_size=32);
print("test loss:", results)
nn_test_sc = sc.inverse_transform(gauss.predict(M_test));
mat_test_sc = sc.inverse_transform(mat_test);

# creating outut file and writing predicted value on it

out_file = f_prefix+"_mat_info_eval.txt"
predictions = gauss.predict(M_eval)
myfile = open(out_file, 'w')
for i in range(len(predictions)):
    a = np.str(predictions[i,0]) + "\t" + np.str(predictions[i,1]) + "\t" + np.str(predictions[i,2]) +"\n"
    myfile.write(a)
myfile.close()



plt.scatter(mat_test_sc[:,0],nn_test_sc[:,0]);
plt.plot([-100,100],[-100, 100],"--k")
plt.xlabel("True alpha");
plt.ylabel("Predicted alpha");
plt.axis([-2, 24, -2, 24])
plt.title("Correlation strength")

plt.figure()
plt.scatter(mat_test_sc[:,1],nn_test_sc[:,1]);
plt.plot([-100, 100],[-100, 100],"--k")
plt.xlabel("True xi");
plt.ylabel("Predicted xi");
plt.axis([0, 70, 0, 70])
plt.title("Correlation length")
plt.figure()
plt.scatter(mat_test_sc[:,2],nn_test_sc[:,2]);
plt.plot([-100, 100],[-100, 100],"--k")
plt.xlabel("True p");
plt.ylabel("Predicted p");
plt.axis([5, 12, 5, 12])
plt.title("Dissipation strength")
```

```
19/19 [==============================] - 0s 5ms/step - loss: 0.0396
test loss: 0.03956524655222893
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:14: DeprecationWarning: `np.str` is a deprecated alias for the builtin `str`. To silence this warning, use `str` by itself. Doing
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations

Text(0.5, 1.0, 'Dissipation strength')
```







Heatmap of important features in the time domain

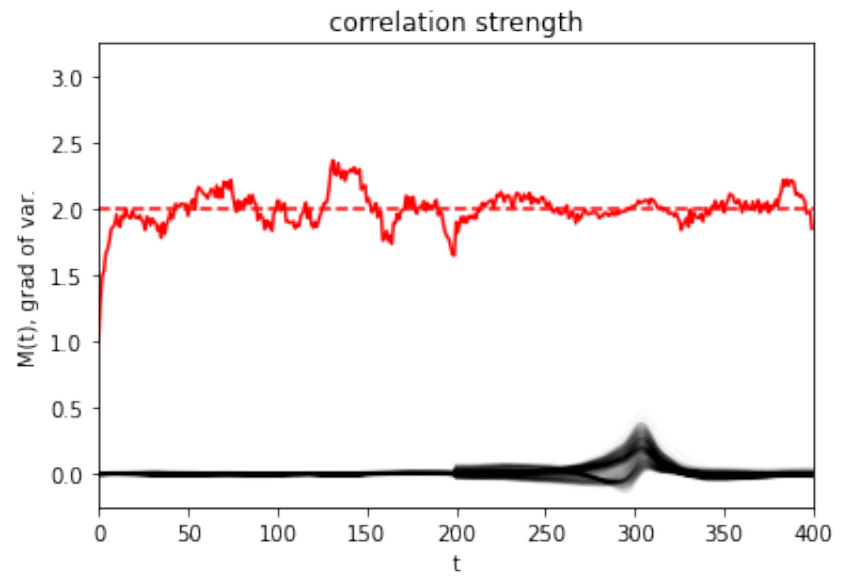
It can be helpful to visualize the relative "importance" of each input element by back-propagating gradients of the output. For example, at a given t, we are computing for α:

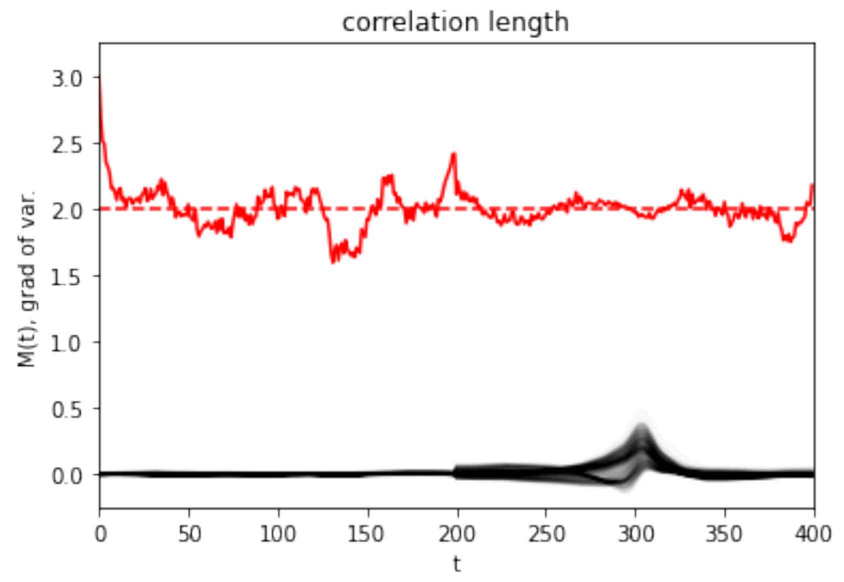
$$G_\alpha(t) = \sum_i \frac{d\alpha}{dM_i(t)}$$

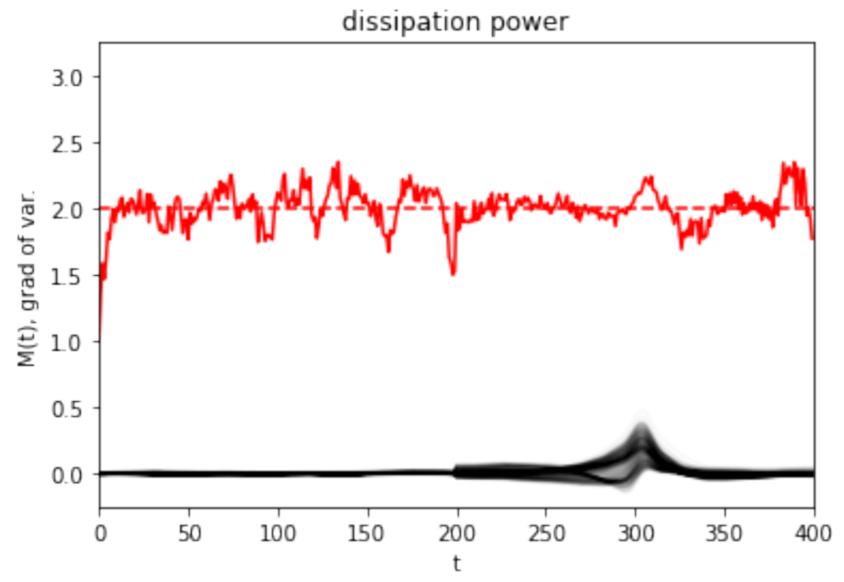
and displaying it in red in the first plot below.

Does the real part of the data (first half of the x-axis) seem important?

```
# heatmap of feature importance in the time domain
from keras import backend as k
import tensorflow as tf

var_names = ["correlation strength", "correlation length", "dissipation power"]

for tar_var in range(3):
    in_tensor = tf.convert_to_tensor(M_test) # we will track gradients w.r.t. M(t
    with tf.GradientTape() as t:
        t.watch(in_tensor)
        tar_output = tf.gather(gauss(in_tensor), tar_var, axis=1) # keep track of

    grads = t.gradient(tar_output, in_tensor).numpy() # comput gradient using ten
    grad_sum = np.sum((grads),axis=0) # sum along all testing curves

    plt.figure()
    plt.plot((M_train[1:500,:]).T,color=(0,0,0,.025))
    plt.plot(2+grad_sum/np.max(np.abs(grad_sum)),'r')
    plt.plot([0, 400],[2, 2],'--r')
    plt.title(var_names[tar_var])
    plt.xlabel('t')
    plt.axis([0, 400, -.25, 3.25])
    plt.ylabel('M(t), grad of var.')
```


correlation strength
M(t), grad of var.
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
50
100
150
200
250
300
350
400
t


correlation length
M(t), grad of var.
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
50
100
150
200
250
300
350
400
t


dissipation power
M(t), grad of var.
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
50
100
150
200
250
300
350
400
t

Submission format

We ask you to submit two models.

When we downloaded the data for the model development, we also downloaded spin echos for evaluation of our models, like "gauss_echos_eval_r.txt" and "RKKY_echos_eval_i.txt". These share the same format as the "<model_type>_echos_model_*.txt," used for the model training above.

Use your model to predict the three spin-interaction variables from the echos, and submit your results for **each model** in a tab delimited .txt file of dimensions 6000 x 3 matching the "<model_type>_mat_info_model.txt" format.

That is, the columns should be:

| α | ξ | d |

and there should be 6000 rows.

Name this file "<model_type>_mat_info_eval.txt"

The quality of the model will be judged by the minimization of normalized mean-square error:

$$\mathrm{Err} = \sum_{\upsilon=1}^{3} \sum_{i=i}^{6000} \left(\tilde{\upsilon}^i_{\mathrm{model}} - \tilde{\upsilon}^i_{\mathrm{true}}\right)^2$$

where υ^i is one of the three spin-interaction variables for echo number i, and the tilde represents normalization of each variable (using the StandardScalar() object used above).

Your submission should include:

- Your ipython notebook (`.ipynb`),
- A PDF copy of your notebook together with a description of what you have done,
- Your model's evaluation of the Gaussian data ("gauss_mat_info_eval.txt"),
- Your model's evaluation of the RKKY data ("RKKY_mat_info_eval.txt").