# Hackathon 3

# Group 16

In this hackathon, we train models that predict the strength and shape of interactions between the nuclear spins from simulated time-dependent magnetization curves. Two models are made depending upon the type of interaction function between the nuclear spins: Gaussian and Ruderman-Kittel-Kasuya-Yosida (RKKY) function. Our main goal here is to build a regression model to predict the three parameters: coupling strength ($\alpha$), coupling length ($\boldsymbol{\xi}$), and the decay time (**d**).

First, we built a Multi-layer Perceptron (MLP) model and trained it using the Gaussian function. We used GridsearchCV from the SciKit-Learn library to find the best hyperparameters, such as: optimizer (Adam), weight initialization (he_uniform), activation function (relu), batch size, number of epochs, and the number of neurons in the hidden layers. After finding the optimal hyperparameters, we trained the model on truncated data (which is centered roughly at the echo) and its performance was evaluated on the test dataset. We also trained the model on the entire time-axis data, and a prediction is made on the evaluation dataset. In both cases, our model's performance didn't vary much. The test loss remained around 0.0131.

Next, for the RKKY model prediction we just used the imaginary part of the whole magnetization M(t) curves. For a pyramid shape structure of the Neural Network (NN) layers we defined a function to calculate the number of nodes in subsequent layers. Then using the RandomizedGridSearchCV we found the best NN for RKKY model prediction is 5 dense layers with the batch normalization in between. The selected activation function was 'relu' and for the loss function 'huber_loss' performs the best. In addition, for the kernel initializer the grid search recommended 'lecun_uniform'. We used the early stopping callbacks and hence set up the epoch at a high value of 500. We also used the callback 'ReduceLROnPlateu' to reduce learning rate close to the optimal solution. The model generated a slightly higher loss for this function (test loss of 0.0372) as compared to the loss for the Gaussian function trained on the full time-axis data.

Note: Some intermediate pages of training output were removed in the PDF, but are available for viewing in the .ipynb file.

# Hack3-Gauss-16

April 12, 2021

# 1 Hackathon 3

# 2 Group 16

## 2.1 Load and view the simulated data

[1]:
```
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

r = requests.get("https://docs.google.com/uc?
 ↪export=download&id=1N1wVk5C64p2fy7kxx7fGpvQA8--Bq38W",allow_redirects=True)
open(mat_file, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?
 ↪export=download&id=1__SGeKUwQCXLZa83-nKH1Twhh99Lu7tb",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?
 ↪export=download&id=1kRYLhoi1ClSKQbKBnp9asI5_hOoST_Hd",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=1IWaUbkaLh4XbK8CWrx-VZ78RteKBcwVj",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
```

```
r = requests.get("https://docs.google.com/uc?
↪export=download&id=18N_p6aCJJp_xoYkws5vFX_-mOxqZDkaG",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=1wFOrJB-JpSYohH8MEV-a4E-uw5R5Dxd4",allow_redirects=True)
open(mat_file, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?
↪export=download&id=1bagiHH3-bGAbQIpZalBSPWxg4AAczfpP",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?
↪export=download&id=1PvgRwdlJaDpsqElyU8oebfoaV2t13w35",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=1OCd91DR4qzFCqWonkjvJ9ZhhHphJOi7q",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?
↪export=download&id=1Wrab6Dk9IgRKPuzeEUiB-C5xEiVoFynr",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
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import requests
```

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

[177]:
```
f_prefix = "gauss"; # Gaussian functional between nuclei
#f_prefix = "RKKY"; # RKKY functional between nuclei
```

### 2.1.2 Now load the data and format it correctly

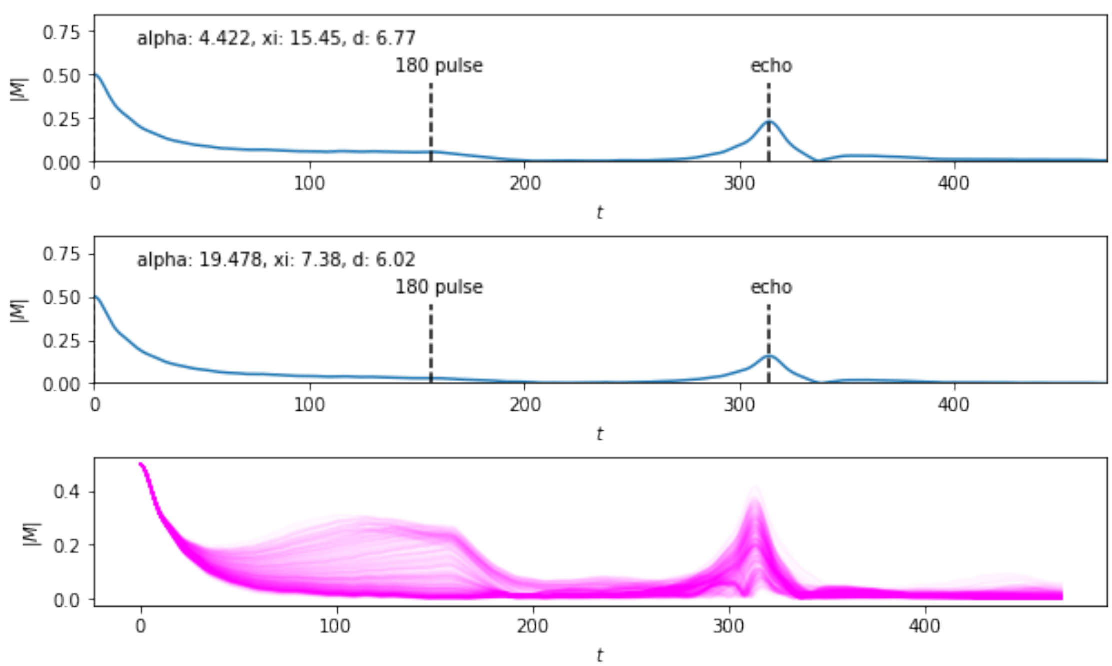
[178]:
```
mat_file = f_prefix+"_mat_info_model.txt"
M_file_r = f_prefix+"_echos_model_r.txt" # real part of echos
M_file_i = f_prefix+"_echos_model_i.txt" # imaginary part of echos

print("Loading into numpy arrays...")
# settings of each simulated material:
# format:  |      |      |   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
```

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

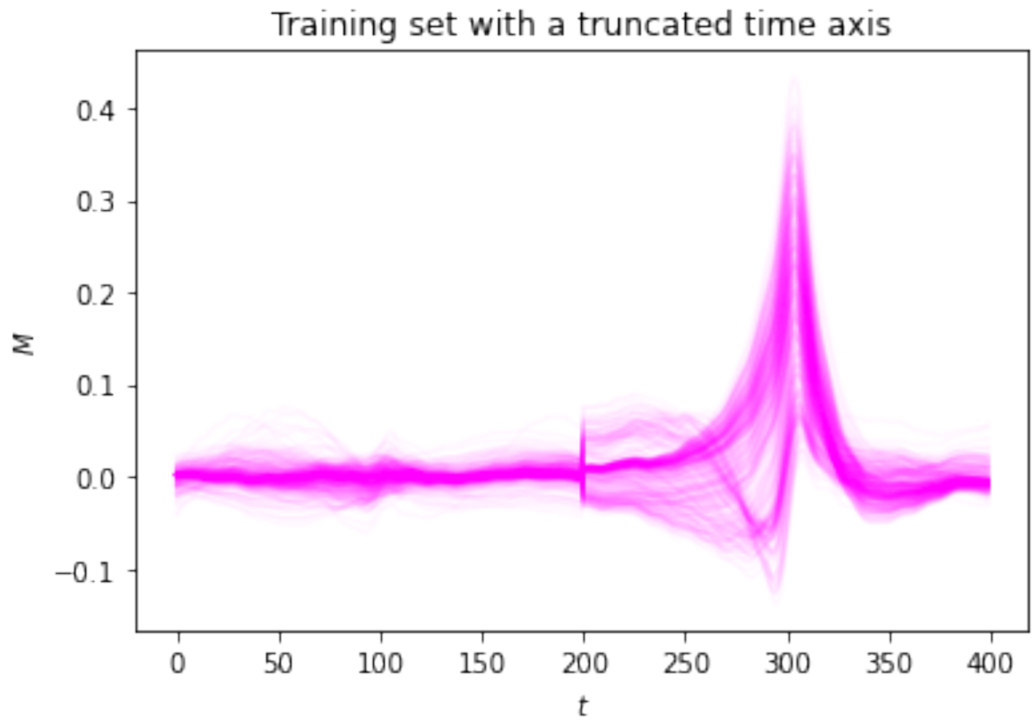
[179]:
```
fig1, ax1 = plt.subplots(3,1, figsize=(10,6));

# 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_i[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='magenta', alpha=0.025 );
ax1[2].set(ylabel="$|M|$", xlabel="$t$");

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



### 2.1.4 Truncate, scale, and partition the training/testing sets

[180]:
```
# 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='magenta', alpha=0.025);
plt.xlabel("$t$")
plt.ylabel("$M$")
plt.title("Training set with a truncated time axis");
```



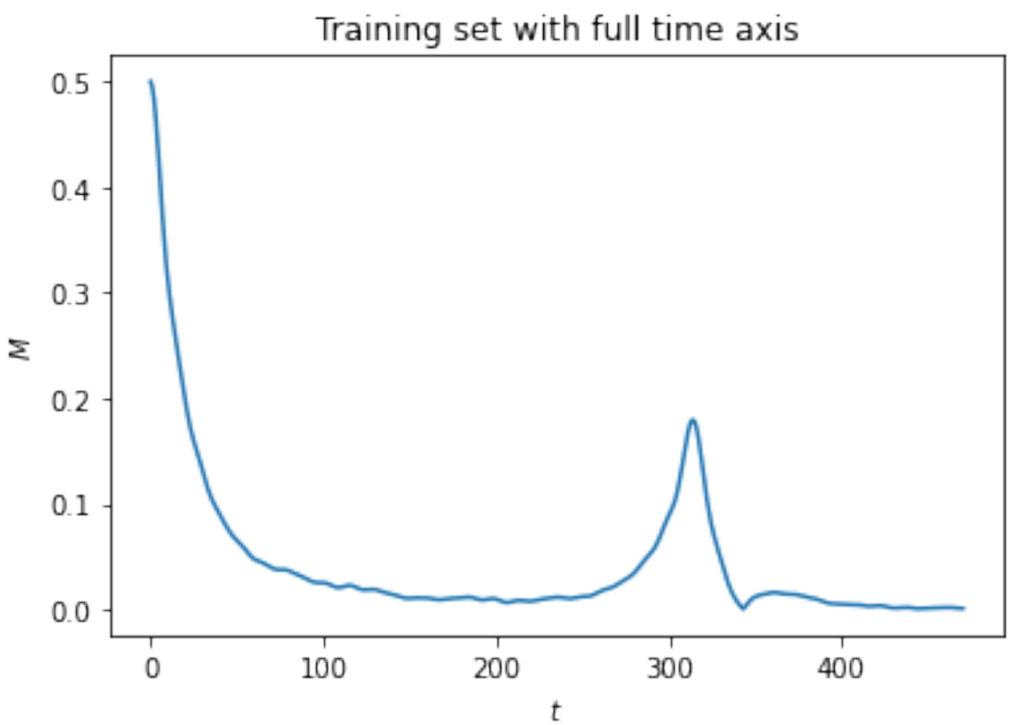

[111]:
```
M_train.shape
```

[111]: (5400, 400)

### 2.2 To use all of the curve

[181]:
```
# 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 !!!

# split into real and imaginary
#M_uncomplex = np.concatenate((np.real(M), np.imag(M)),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_full, M_test_full, mat_train_full, mat_test_full = train_test_split(M,␣
↪mat_info_scaled, test_size=0.1)

# plot the fist 500 elements of the training set, for visualizing variations in␣
↪the data
plt.plot((abs(M_train_full[1,:])));
plt.xlabel("$t$")
plt.ylabel("$M$")
plt.title("Training set with full time axis");
```




[174]:
```
import tensorflow
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, InputLayer, BatchNormalization
from tensorflow.keras.optimizers import Adam, Nadam, SGD, Adamax # gradient␣
↪descent optimizer
from sklearn.model_selection import GridSearchCV

# clear previous layers
tensorflow.keras.backend.clear_session()

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

## 2.3 Guassian function

### 2.3.1 Optimizer selection

[16]:
```
def build_model_optimizer(optimizer):
    model = Sequential([
        InputLayer(input_shape=N),
        Dense(200, activation='elu', kernel_initializer='he_normal'),
        Dense(100, activation='elu', kernel_initializer='he_normal'),
        Dense(50, activation='elu', kernel_initializer='he_normal'),
        Dense(3, activation='linear')
    ])

    model.compile(loss='mean_squared_error', optimizer=optimizer)
    return model

keras_reg_optimizer = keras.wrappers.scikit_learn.
 ↪KerasRegressor(build_model_optimizer)


param_distribs = {
             'optimizer':['sgd', 'adagrad', 'rmsprop', 'adam', 'nadam',␣
 ↪'adamax', 'adadelta']
}

optimizer_search_cv = GridSearchCV(keras_reg_optimizer, param_distribs, cv=3)

grid_result = optimizer_search_cv.fit(M_train, mat_train,␣
 ↪epochs=100,batch_size=64, verbose=2)

# summary results
print("Best: %f using %s" %(grid_result.best_score_, grid_result.best_params_))

means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" %(mean, stdev, param))
```

```
Train on 3600 samples
Epoch 1/100
3600/3600 - 1s - loss: 0.9492
Epoch 2/100
3600/3600 - 0s - loss: 0.9248
Epoch 3/100
3600/3600 - 0s - loss: 0.9091
Epoch 4/100
```

```
5400/5400 - 0s - loss: 0.0841
Epoch 92/100
5400/5400 - 0s - loss: 0.0859
Epoch 93/100
5400/5400 - 0s - loss: 0.1084
Epoch 94/100
5400/5400 - 0s - loss: 0.1007
Epoch 95/100
5400/5400 - 0s - loss: 0.0848
Epoch 96/100
5400/5400 - 0s - loss: 0.0736
Epoch 97/100
5400/5400 - 0s - loss: 0.1116
Epoch 98/100
5400/5400 - 0s - loss: 0.0745
Epoch 99/100
5400/5400 - 0s - loss: 0.1030
Epoch 100/100
5400/5400 - 0s - loss: 0.1097
Best: -0.147688 using {'optimizer': 'adam'}
-0.630895 (0.040662) with: {'optimizer': 'sgd'}
-0.631357 (0.047170) with: {'optimizer': 'adagrad'}
-0.350050 (0.095991) with: {'optimizer': 'rmsprop'}
-0.147688 (0.016441) with: {'optimizer': 'adam'}
-0.205244 (0.089067) with: {'optimizer': 'nadam'}
-0.238581 (0.027734) with: {'optimizer': 'adamax'}
-0.961784 (0.065228) with: {'optimizer': 'adadelta'}
```

### 2.3.2 Grid Search for Weight Initialization

[17]:
```
def build_model_optimizer(weight_initializer='uniform'):
    model = Sequential([
        InputLayer(input_shape=N),
        Dense(200, activation='elu', kernel_initializer=weight_initializer),
        Dense(100, activation='elu', kernel_initializer=weight_initializer),
        Dense(50, activation='elu', kernel_initializer=weight_initializer),
        Dense(3, activation='linear')
    ])

    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

keras_reg_weight_init = keras.wrappers.scikit_learn.
 ↪KerasRegressor(build_model_optimizer)
```

```
param_distribs = {
            'weight_initializer':['uniform', 'lecun_uniform', 'normal', 'zero',␣
 ↪'glorot_normal', 'glorot_uniform', 'he_normal', 'he_uniform']
}

weight_init_search_cv = GridSearchCV(keras_reg_weight_init, param_distribs,␣
 ↪cv=3)

grid_result_weight = weight_init_search_cv.fit(M_train, mat_train,␣
 ↪epochs=100,batch_size=64, verbose=2)

# summary results
print("Best: %f using %s" %(grid_result_weight.best_score_, grid_result_weight.
 ↪best_params_))

means = grid_result_weight.cv_results_['mean_test_score']
stds = grid_result_weight.cv_results_['std_test_score']
params = grid_result_weight.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" %(mean, stdev, param))
```

```
Train on 3600 samples
Epoch 1/100
3600/3600 - 1s - loss: 0.8980
Epoch 2/100
3600/3600 - 0s - loss: 0.6803
Epoch 3/100
3600/3600 - 0s - loss: 0.6448
Epoch 4/100
3600/3600 - 0s - loss: 0.6329
Epoch 5/100
3600/3600 - 0s - loss: 0.6258
Epoch 6/100
3600/3600 - 0s - loss: 0.6196
Epoch 7/100
3600/3600 - 0s - loss: 0.6176
Epoch 8/100
3600/3600 - 0s - loss: 0.6121
Epoch 9/100
3600/3600 - 0s - loss: 0.6079
Epoch 10/100
3600/3600 - 0s - loss: 0.5975
Epoch 11/100
3600/3600 - 0s - loss: 0.5796
Epoch 12/100
3600/3600 - 0s - loss: 0.5510
Epoch 13/100
```

```
5400/5400 - 0s - loss: 0.1200
Epoch 86/100
5400/5400 - 0s - loss: 0.1081
Epoch 87/100
5400/5400 - 0s - loss: 0.0832
Epoch 88/100
5400/5400 - 0s - loss: 0.0850
Epoch 89/100
5400/5400 - 0s - loss: 0.0857
Epoch 90/100
5400/5400 - 0s - loss: 0.0914
Epoch 91/100
5400/5400 - 0s - loss: 0.0879
Epoch 92/100
5400/5400 - 0s - loss: 0.0892
Epoch 93/100
5400/5400 - 0s - loss: 0.1065
Epoch 94/100
5400/5400 - 0s - loss: 0.0867
Epoch 95/100
5400/5400 - 0s - loss: 0.0900
Epoch 96/100
5400/5400 - 0s - loss: 0.1092
Epoch 97/100
5400/5400 - 0s - loss: 0.0954
Epoch 98/100
5400/5400 - 0s - loss: 0.0994
Epoch 99/100
5400/5400 - 0s - loss: 0.0785
Epoch 100/100
5400/5400 - 0s - loss: 0.0825
Best: -0.125546 using {'weight_initializer': 'he_uniform'}
-0.166495 (0.020828) with: {'weight_initializer': 'uniform'}
-0.145179 (0.029936) with: {'weight_initializer': 'lecun_uniform'}
-0.133503 (0.003732) with: {'weight_initializer': 'normal'}
-1.009452 (0.062361) with: {'weight_initializer': 'zero'}
-0.134905 (0.011377) with: {'weight_initializer': 'glorot_normal'}
-0.142354 (0.007820) with: {'weight_initializer': 'glorot_uniform'}
-0.168567 (0.022250) with: {'weight_initializer': 'he_normal'}
-0.125546 (0.009107) with: {'weight_initializer': 'he_uniform'}
```

### 2.3.3 Tuning Neuron Activation Function

[18]:
```
def build_model_activation(activation = 'relu',weight_initializer='he_uniform'):
    model = Sequential([
        InputLayer(input_shape=N),
```

```
        Dense(200, activation=activation, kernel_initializer=weight_initializer),
        Dense(100, activation=activation, kernel_initializer=weight_initializer),
         Dense(50, activation=activation, kernel_initializer=weight_initializer),
         Dense(3, activation='linear')
    ])

    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

keras_reg_activation = keras.wrappers.scikit_learn.
 ↪KerasRegressor(build_model_activation)


param_distribs = [{'activation':['softsign', 'relu', 'tanh', 'sigmoid',␣
 ↪'hard_sigmoid', 'linear','elu'],
             'weight_initializer':['he_uniform']}, {'activation':
 ↪['selu'],'weight_initializer':['lecun_normal']
             }]

activation_search_cv = GridSearchCV(keras_reg_activation, param_distribs, cv=3)

grid_result_activation = activation_search_cv.fit(M_train, mat_train,␣
 ↪epochs=100,batch_size=64, verbose=2)

# summary results
print("Best: %f using %s" %(grid_result_activation.best_score_,␣
 ↪grid_result_activation.best_params_))

means = grid_result_activation.cv_results_['mean_test_score']
stds = grid_result_activation.cv_results_['std_test_score']
params = grid_result_activation.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" %(mean, stdev, param))
```

```
Train on 3600 samples
Epoch 1/100
3600/3600 - 1s - loss: 0.8382
Epoch 2/100
3600/3600 - 0s - loss: 0.6614
Epoch 3/100
3600/3600 - 0s - loss: 0.6414
Epoch 4/100
3600/3600 - 0s - loss: 0.6166
Epoch 5/100
3600/3600 - 0s - loss: 0.6035
Epoch 6/100
3600/3600 - 0s - loss: 0.5812
```

```
Epoch 79/100
5400/5400 - 0s - loss: 0.0638
Epoch 80/100
5400/5400 - 0s - loss: 0.0688
Epoch 81/100
5400/5400 - 0s - loss: 0.0598
Epoch 82/100
5400/5400 - 0s - loss: 0.0686
Epoch 83/100
5400/5400 - 0s - loss: 0.0678
Epoch 84/100
5400/5400 - 0s - loss: 0.0720
Epoch 85/100
5400/5400 - 0s - loss: 0.0695
Epoch 86/100
5400/5400 - 0s - loss: 0.0600
Epoch 87/100
5400/5400 - 0s - loss: 0.0696
Epoch 88/100
5400/5400 - 0s - loss: 0.0759
Epoch 89/100
5400/5400 - 0s - loss: 0.0755
Epoch 90/100
5400/5400 - 0s - loss: 0.0620
Epoch 91/100
5400/5400 - 0s - loss: 0.0709
Epoch 92/100
5400/5400 - 0s - loss: 0.0515
Epoch 93/100
5400/5400 - 0s - loss: 0.0613
Epoch 94/100
5400/5400 - 0s - loss: 0.0713
Epoch 95/100
5400/5400 - 0s - loss: 0.0706
Epoch 96/100
5400/5400 - 0s - loss: 0.0555
Epoch 97/100
5400/5400 - 0s - loss: 0.0605
Epoch 98/100
5400/5400 - 0s - loss: 0.0673
Epoch 99/100
5400/5400 - 0s - loss: 0.0648
Epoch 100/100
5400/5400 - 0s - loss: 0.0828
Best: -0.079730 using {'activation': 'relu', 'weight_initializer': 'he_uniform'}
-0.108702 (0.017724) with: {'activation': 'softsign', 'weight_initializer':
'he_uniform'}
-0.079730 (0.010296) with: {'activation': 'relu', 'weight_initializer':
```

```
'he_uniform'}
-0.112754 (0.009486) with: {'activation': 'tanh', 'weight_initializer':
'he_uniform'}
-0.384742 (0.075484) with: {'activation': 'sigmoid', 'weight_initializer':
'he_uniform'}
-0.273885 (0.043225) with: {'activation': 'hard_sigmoid', 'weight_initializer':
'he_uniform'}
-0.427349 (0.038146) with: {'activation': 'linear', 'weight_initializer':
'he_uniform'}
-0.146191 (0.028081) with: {'activation': 'elu', 'weight_initializer':
'he_uniform'}
-0.134230 (0.015063) with: {'activation': 'selu', 'weight_initializer':
'lecun_normal'}
```

### 2.3.4 Batch size and no of epochs tuning

[35]:
```
def build_model_batch(batch_size = 64, epoch=100):
    model = Sequential([
        InputLayer(input_shape=N),
        Dense(200, activation='relu', kernel_initializer='he_uniform'),
        Dense(100, activation='relu', kernel_initializer='he_uniform'),
        Dense(50, activation='relu', kernel_initializer='he_uniform'),
        Dense(3, activation='linear')
    ])


    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

keras_reg_lr = keras.wrappers.scikit_learn.KerasRegressor(build_model_batch)

param_distribs = {
             'batch_size': [16,32,64,128,256],
             'epoch': [50, 100, 150, 200, 250, 300]
              }

batch_grid_search_cv = GridSearchCV(keras_reg_lr, param_distribs, cv=3)

batch_grid_result = batch_grid_search_cv.fit(M_train, mat_train, verbose=2)

# summary results
print("Best: %f using %s" %(batch_grid_result.best_score_, batch_grid_result.
 ↪best_params_))

means = batch_grid_result.cv_results_['mean_test_score']
stds = batch_grid_result.cv_results_['std_test_score']
```

```python
params = batch_grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" %(mean, stdev, param))
```

```
Train on 3600 samples
3600/3600 - 1s - loss: 0.7556
1800/1800 [==============================] - 0s 188us/sample - loss: 0.6598
Train on 3600 samples
3600/3600 - 1s - loss: 0.8339
1800/1800 [==============================] - 0s 165us/sample - loss: 0.5705
Train on 3600 samples
3600/3600 - 1s - loss: 0.7989
1800/1800 [==============================] - 0s 166us/sample - loss: 0.5579
Train on 3600 samples
3600/3600 - 1s - loss: 0.7806
1800/1800 [==============================] - 0s 164us/sample - loss: 0.6742
Train on 3600 samples
3600/3600 - 1s - loss: 0.8370
1800/1800 [==============================] - 0s 163us/sample - loss: 0.5530
Train on 3600 samples
3600/3600 - 1s - loss: 0.8424
1800/1800 [==============================] - 0s 168us/sample - loss: 0.5627
Train on 3600 samples
3600/3600 - 1s - loss: 0.7720
1800/1800 [==============================] - 0s 172us/sample - loss: 0.6512
Train on 3600 samples
3600/3600 - 1s - loss: 0.8278
1800/1800 [==============================] - 0s 171us/sample - loss: 0.5565
Train on 3600 samples
3600/3600 - 1s - loss: 0.8022
1800/1800 [==============================] - 0s 171us/sample - loss: 0.5637
Train on 3600 samples
3600/3600 - 1s - loss: 0.7472
1800/1800 [==============================] - 0s 170us/sample - loss: 0.6288
Train on 3600 samples
3600/3600 - 1s - loss: 0.8114
1800/1800 [==============================] - 0s 164us/sample - loss: 0.5576
Train on 3600 samples
3600/3600 - 1s - loss: 0.7695
1800/1800 [==============================] - 0s 170us/sample - loss: 0.5416
Train on 3600 samples
3600/3600 - 1s - loss: 0.7690
1800/1800 [==============================] - 0s 167us/sample - loss: 0.6646
Train on 3600 samples
3600/3600 - 1s - loss: 0.8145
1800/1800 [==============================] - 0s 160us/sample - loss: 0.5630
Train on 3600 samples
```

```
3600/3600 - 0s - loss: 0.9400
1800/1800 [==============================] - 0s 52us/sample - loss: 1.0323
Train on 3600 samples
3600/3600 - 0s - loss: 0.9947
1800/1800 [==============================] - 0s 52us/sample - loss: 0.8839
Train on 3600 samples
3600/3600 - 0s - loss: 0.9942
1800/1800 [==============================] - 0s 58us/sample - loss: 0.9114
Train on 3600 samples
3600/3600 - 0s - loss: 0.9334
1800/1800 [==============================] - 0s 55us/sample - loss: 1.0233
Train on 3600 samples
3600/3600 - 0s - loss: 1.0217
1800/1800 [==============================] - 0s 54us/sample - loss: 0.9145
Train on 3600 samples
3600/3600 - 0s - loss: 0.9940
1800/1800 [==============================] - 0s 60us/sample - loss: 0.9106
Train on 3600 samples
3600/3600 - 0s - loss: 0.9460
1800/1800 [==============================] - 0s 52us/sample - loss: 1.0408
Train on 3600 samples
3600/3600 - 0s - loss: 1.0110
1800/1800 [==============================] - 0s 51us/sample - loss: 0.8998
Train on 3600 samples
3600/3600 - 0s - loss: 1.0087
1800/1800 [==============================] - 0s 50us/sample - loss: 0.9329
Train on 3600 samples
3600/3600 - 0s - loss: 0.9317
1800/1800 [==============================] - 0s 57us/sample - loss: 1.0117
Train on 3600 samples
3600/3600 - 0s - loss: 0.9966
1800/1800 [==============================] - 0s 52us/sample - loss: 0.8741
Train on 3600 samples
3600/3600 - 0s - loss: 0.9917
1800/1800 [==============================] - 0s 55us/sample - loss: 0.9083
Train on 5400 samples
5400/5400 - 3s - loss: 0.7397
Best: -0.576023 using {'batch_size': 16, 'epoch': 200}
-0.596059 (0.045362) with: {'batch_size': 16, 'epoch': 50}
-0.596598 (0.054987) with: {'batch_size': 16, 'epoch': 100}
-0.590494 (0.043055) with: {'batch_size': 16, 'epoch': 150}
-0.576023 (0.037882) with: {'batch_size': 16, 'epoch': 200}
-0.601790 (0.044815) with: {'batch_size': 16, 'epoch': 250}
-0.591613 (0.052726) with: {'batch_size': 16, 'epoch': 300}
-0.658885 (0.013461) with: {'batch_size': 32, 'epoch': 50}
-0.635084 (0.057803) with: {'batch_size': 32, 'epoch': 100}
-0.657650 (0.055116) with: {'batch_size': 32, 'epoch': 150}
-0.684548 (0.069764) with: {'batch_size': 32, 'epoch': 200}
```

```
-0.679852 (0.048055) with: {'batch_size': 32, 'epoch': 250}
-0.643875 (0.053776) with: {'batch_size': 32, 'epoch': 300}
-0.780423 (0.053293) with: {'batch_size': 64, 'epoch': 50}
-0.802147 (0.083389) with: {'batch_size': 64, 'epoch': 100}
-0.803490 (0.056949) with: {'batch_size': 64, 'epoch': 150}
-0.772382 (0.087724) with: {'batch_size': 64, 'epoch': 200}
-0.778947 (0.033937) with: {'batch_size': 64, 'epoch': 250}
-0.780335 (0.035654) with: {'batch_size': 64, 'epoch': 300}
-0.908431 (0.056251) with: {'batch_size': 128, 'epoch': 50}
-0.901729 (0.051258) with: {'batch_size': 128, 'epoch': 100}
-0.891181 (0.063491) with: {'batch_size': 128, 'epoch': 150}
-0.890768 (0.047462) with: {'batch_size': 128, 'epoch': 200}
-0.901196 (0.058456) with: {'batch_size': 128, 'epoch': 250}
-0.878940 (0.061387) with: {'batch_size': 128, 'epoch': 300}
-0.941948 (0.057752) with: {'batch_size': 256, 'epoch': 50}
-0.943363 (0.055194) with: {'batch_size': 256, 'epoch': 100}
-0.942565 (0.064442) with: {'batch_size': 256, 'epoch': 150}
-0.949439 (0.052259) with: {'batch_size': 256, 'epoch': 200}
-0.957848 (0.060197) with: {'batch_size': 256, 'epoch': 250}
-0.931377 (0.058517) with: {'batch_size': 256, 'epoch': 300}
```

### 2.3.5 Number of neurons and hidden layer

[37]:
```
def build_model_neurons(no_neurons=100):
    model = Sequential([
        InputLayer(input_shape=N),
        Dense(no_neurons, activation='relu', kernel_initializer='he_uniform'),
        Dense(no_neurons/2, activation='relu', kernel_initializer='he_uniform'),
        Dense(50, activation='relu', kernel_initializer='he_uniform'),
        Dense(3, activation='linear')
    ])


    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

keras_reg_neurons = keras.wrappers.scikit_learn.
 ↪KerasRegressor(build_model_neurons)

param_distribs = {
              'no_neurons': [100, 150, 200, 250, 300, 350, 400],
               }

neurons_grid_search_cv = GridSearchCV(keras_reg_neurons, param_distribs, cv=3)
```

```
neurons_grid_result = neurons_grid_search_cv.fit(M_train, mat_train, batch_size␣
↪= 16, epochs = 200, verbose=2)

# summary results
print("Best: %f using %s" %(neurons_grid_result.best_score_,␣
↪neurons_grid_result.best_params_))

means = neurons_grid_result.cv_results_['mean_test_score']
stds = neurons_grid_result.cv_results_['std_test_score']
params = neurons_grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" %(mean, stdev, param))
```

```
Train on 3600 samples
Epoch 1/200
3600/3600 - 1s - loss: 0.8169
Epoch 2/200
3600/3600 - 1s - loss: 0.5632
Epoch 3/200
3600/3600 - 1s - loss: 0.5078
Epoch 4/200
3600/3600 - 1s - loss: 0.4579
Epoch 5/200
3600/3600 - 1s - loss: 0.4274
Epoch 6/200
3600/3600 - 1s - loss: 0.3848
Epoch 7/200
3600/3600 - 1s - loss: 0.3116
Epoch 8/200
3600/3600 - 1s - loss: 0.2635
Epoch 9/200
3600/3600 - 1s - loss: 0.2168
Epoch 10/200
3600/3600 - 1s - loss: 0.2011
Epoch 11/200
3600/3600 - 1s - loss: 0.1934
Epoch 12/200
3600/3600 - 1s - loss: 0.1867
Epoch 13/200
3600/3600 - 1s - loss: 0.1640
Epoch 14/200
3600/3600 - 1s - loss: 0.1709
Epoch 15/200
3600/3600 - 1s - loss: 0.1600
Epoch 16/200
3600/3600 - 1s - loss: 0.1630
Epoch 17/200
```

```
5400/5400 - 1s - loss: 0.0543
Epoch 189/200
5400/5400 - 1s - loss: 0.0431
Epoch 190/200
5400/5400 - 1s - loss: 0.0566
Epoch 191/200
5400/5400 - 1s - loss: 0.0528
Epoch 192/200
5400/5400 - 1s - loss: 0.0503
Epoch 193/200
5400/5400 - 1s - loss: 0.0455
Epoch 194/200
5400/5400 - 1s - loss: 0.0542
Epoch 195/200
5400/5400 - 1s - loss: 0.0496
Epoch 196/200
5400/5400 - 1s - loss: 0.0630
Epoch 197/200
5400/5400 - 1s - loss: 0.0520
Epoch 198/200
5400/5400 - 1s - loss: 0.0473
Epoch 199/200
5400/5400 - 1s - loss: 0.0490
Epoch 200/200
5400/5400 - 1s - loss: 0.0432
Best: -0.066506 using {'no_neurons': 150}
-0.071915 (0.011263) with: {'no_neurons': 100}
-0.066506 (0.013355) with: {'no_neurons': 150}
-0.087943 (0.008150) with: {'no_neurons': 200}
-0.071696 (0.007430) with: {'no_neurons': 250}
-0.114646 (0.054477) with: {'no_neurons': 300}
-0.088568 (0.023491) with: {'no_neurons': 350}
-0.131287 (0.071479) with: {'no_neurons': 400}
```

### 2.3.6 Training using tuned parameter

[65]:
```
tensorflow.keras.backend.clear_session()

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

# define the net
nn = Sequential()
nn.add(InputLayer(input_shape=N))
nn.add(Dense(400, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(150, activation='relu', kernel_initializer='he_uniform'))
```

```python
nn.add(BatchNormalization())
nn.add(Dense(75, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(50, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(3, activation='linear'))

batch_size = 128
s = 200*len(M_train)//batch_size
learning_schedule = keras.optimizers.schedules.ExponentialDecay(0.001, s, 0.1)

nn.compile(loss='huber_loss', optimizer=Adam(learning_schedule))
```

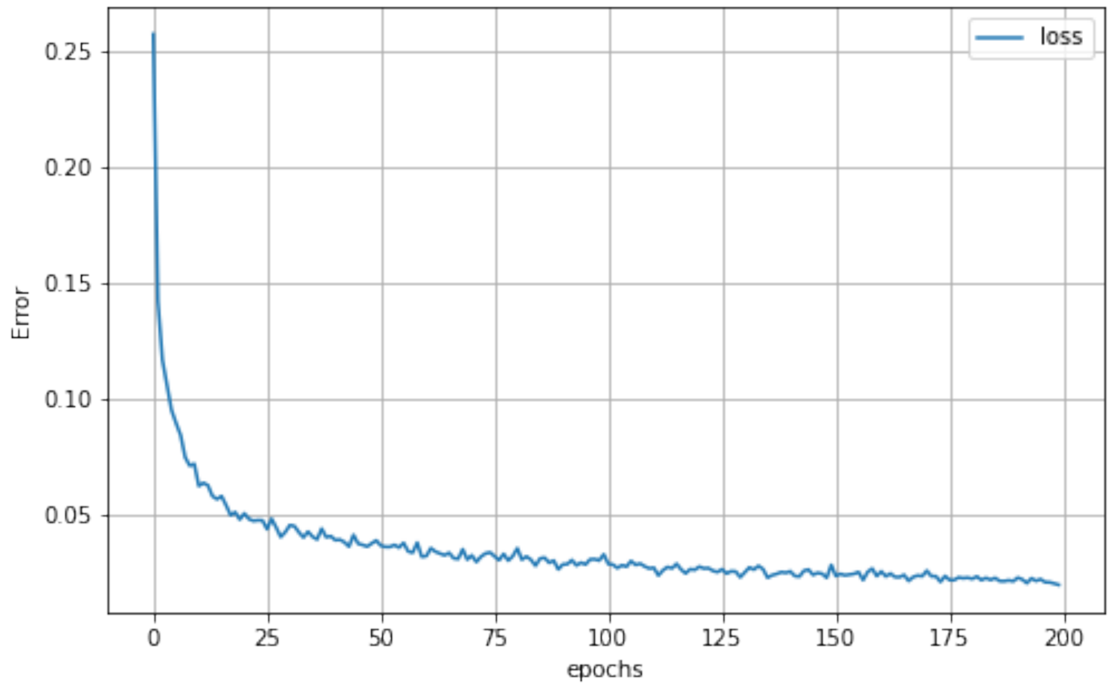
[66]:
```python
history = nn.fit(M_train, mat_train, epochs=200,batch_size=128, verbose=2)
```

```
Train on 5400 samples
Epoch 1/200
5400/5400 - 2s - loss: 0.2574
Epoch 2/200
5400/5400 - 0s - loss: 0.1421
Epoch 3/200
5400/5400 - 0s - loss: 0.1166
Epoch 4/200
5400/5400 - 0s - loss: 0.1059
Epoch 5/200
5400/5400 - 0s - loss: 0.0949
Epoch 6/200
5400/5400 - 0s - loss: 0.0892
Epoch 7/200
5400/5400 - 0s - loss: 0.0841
Epoch 8/200
5400/5400 - 0s - loss: 0.0745
Epoch 9/200
5400/5400 - 0s - loss: 0.0709
Epoch 10/200
5400/5400 - 0s - loss: 0.0717
Epoch 11/200
5400/5400 - 0s - loss: 0.0621
Epoch 12/200
5400/5400 - 0s - loss: 0.0634
Epoch 13/200
5400/5400 - 0s - loss: 0.0624
Epoch 14/200
5400/5400 - 0s - loss: 0.0577
Epoch 15/200
5400/5400 - 0s - loss: 0.0564
Epoch 16/200
```

```
5400/5400 - 0s - loss: 0.0224
Epoch 185/200
5400/5400 - 0s - loss: 0.0215
Epoch 186/200
5400/5400 - 0s - loss: 0.0224
Epoch 187/200
5400/5400 - 0s - loss: 0.0210
Epoch 188/200
5400/5400 - 0s - loss: 0.0210
Epoch 189/200
5400/5400 - 0s - loss: 0.0213
Epoch 190/200
5400/5400 - 0s - loss: 0.0209
Epoch 191/200
5400/5400 - 0s - loss: 0.0225
Epoch 192/200
5400/5400 - 0s - loss: 0.0217
Epoch 193/200
5400/5400 - 0s - loss: 0.0201
Epoch 194/200
5400/5400 - 0s - loss: 0.0222
Epoch 195/200
5400/5400 - 0s - loss: 0.0211
Epoch 196/200
5400/5400 - 0s - loss: 0.0218
Epoch 197/200
5400/5400 - 0s - loss: 0.0205
Epoch 198/200
5400/5400 - 0s - loss: 0.0205
Epoch 199/200
5400/5400 - 0s - loss: 0.0199
Epoch 200/200
5400/5400 - 0s - loss: 0.0194
```

[67]:

```
# visualing the train and validation losses
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.xlabel("epochs")
plt.ylabel('Error')
plt.show()
```

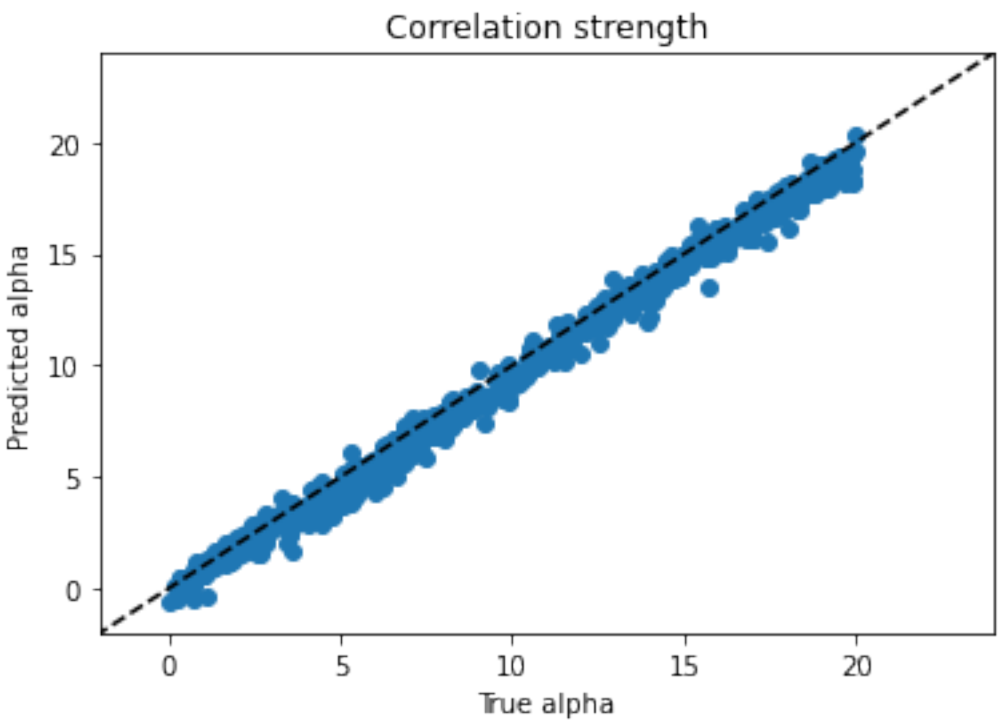


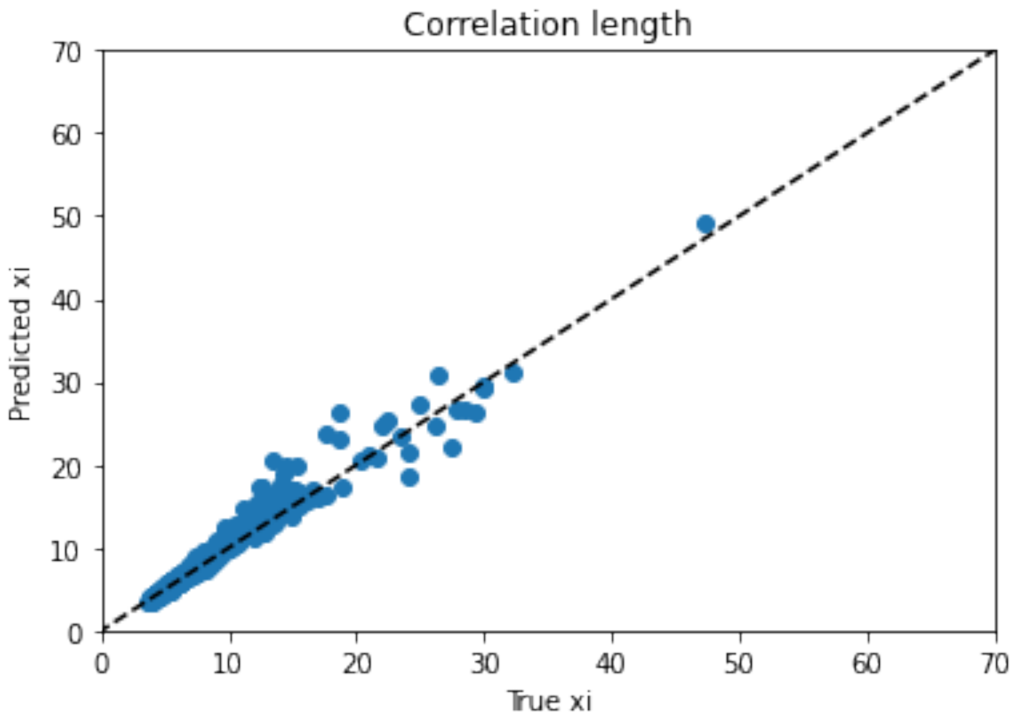

[69]:
```
# check results on test set

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

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]);
```

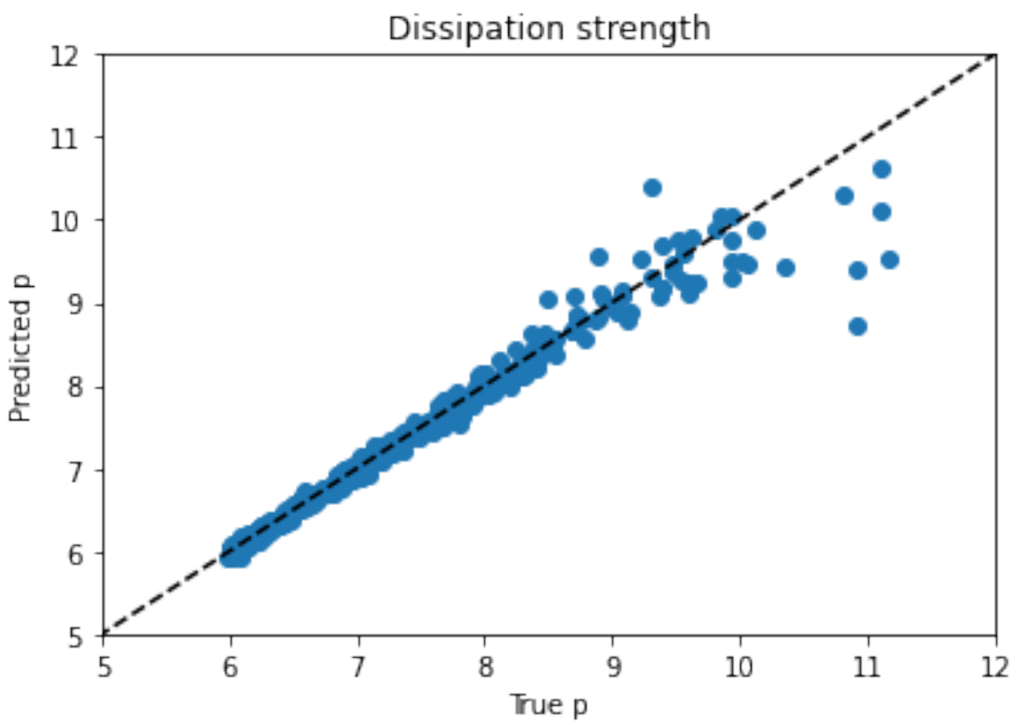
```
plt.plot([-100, 100],[-100, 100],"--k")
plt.xlabel("True p");
plt.ylabel("Predicted p");
plt.axis([5, 12, 5, 12])
plt.title("Dissipation strength")
```

```
600/600 [==============================] - 0s 123us/sample - loss: 0.0131
test loss: 0.013058506498734156
```

[69]:
```
Text(0.5, 1.0, 'Dissipation strength')
```




Correlation length
Predicted xi
True xi
0
10
20
30
40
50
60
70


Dissipation strength
Predicted p
True p
5
6
7
8
9
10
11
12

### 2.3.7 Using whole curve to train the model and making predictions on Evaluation data and saving as .txt file

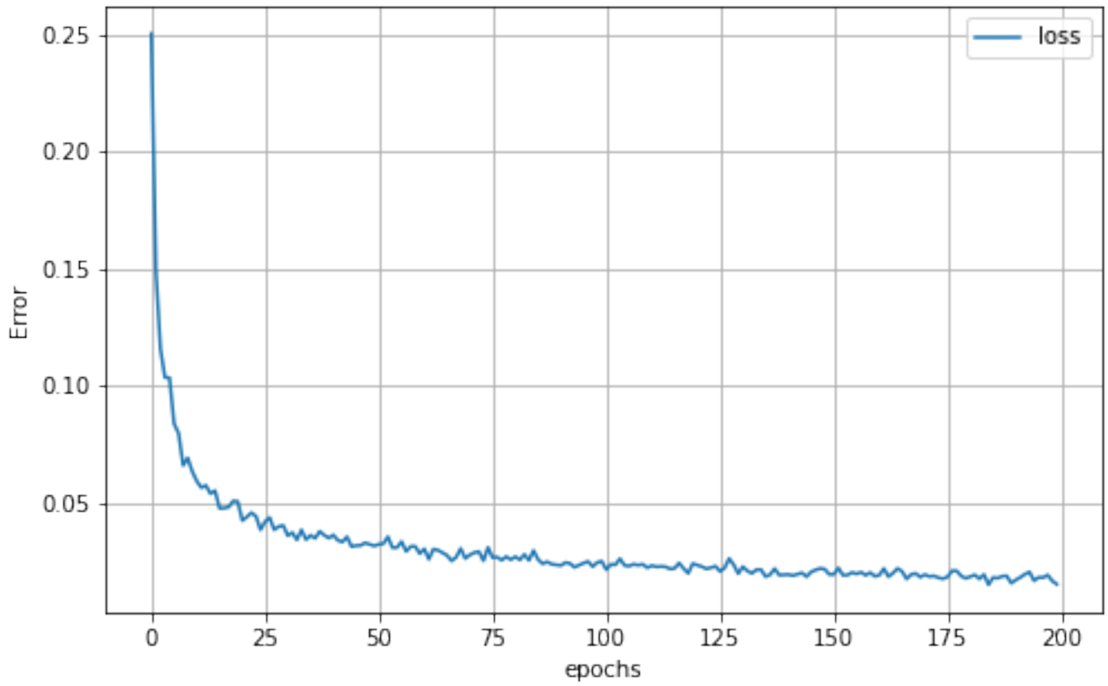
[91]:
```
tensorflow.keras.backend.clear_session()

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

# define the net
nn = Sequential()
nn.add(InputLayer(input_shape=N))
nn.add(Dense(400, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(150, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(75, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(50, activation='relu', kernel_initializer='he_uniform'))
nn.add(BatchNormalization())
nn.add(Dense(3, activation='linear'))

batch_size = 128
s = 200*len(M_train)//batch_size
learning_schedule = keras.optimizers.schedules.ExponentialDecay(0.001, s, 0.1)

nn.compile(loss='huber_loss', optimizer=Adam(learning_schedule))
```

[92]:
```
history1 = nn.fit(M_train_full, mat_train_full, epochs=200,batch_size=128,␣
↪verbose=2)
```

```
Train on 5400 samples
Epoch 1/200
5400/5400 - 2s - loss: 0.2502
Epoch 2/200
5400/5400 - 0s - loss: 0.1492
Epoch 3/200
5400/5400 - 0s - loss: 0.1157
Epoch 4/200
5400/5400 - 0s - loss: 0.1034
Epoch 5/200
5400/5400 - 0s - loss: 0.1033
Epoch 6/200
5400/5400 - 0s - loss: 0.0837
Epoch 7/200
5400/5400 - 0s - loss: 0.0795
Epoch 8/200
5400/5400 - 0s - loss: 0.0658
Epoch 9/200
```

[93]:

```python
# visualing the train and validation losses
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history1.history).plot(figsize=(8,5))
plt.grid(True)
plt.xlabel("epochs")
plt.ylabel('Error')
plt.show()
```



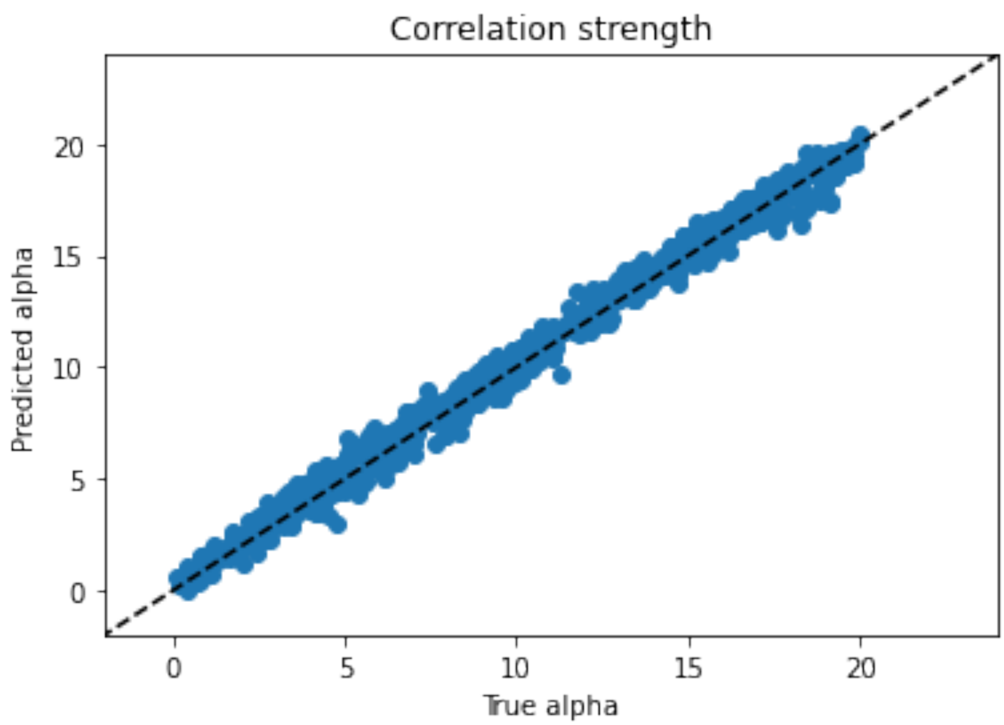

[95]:

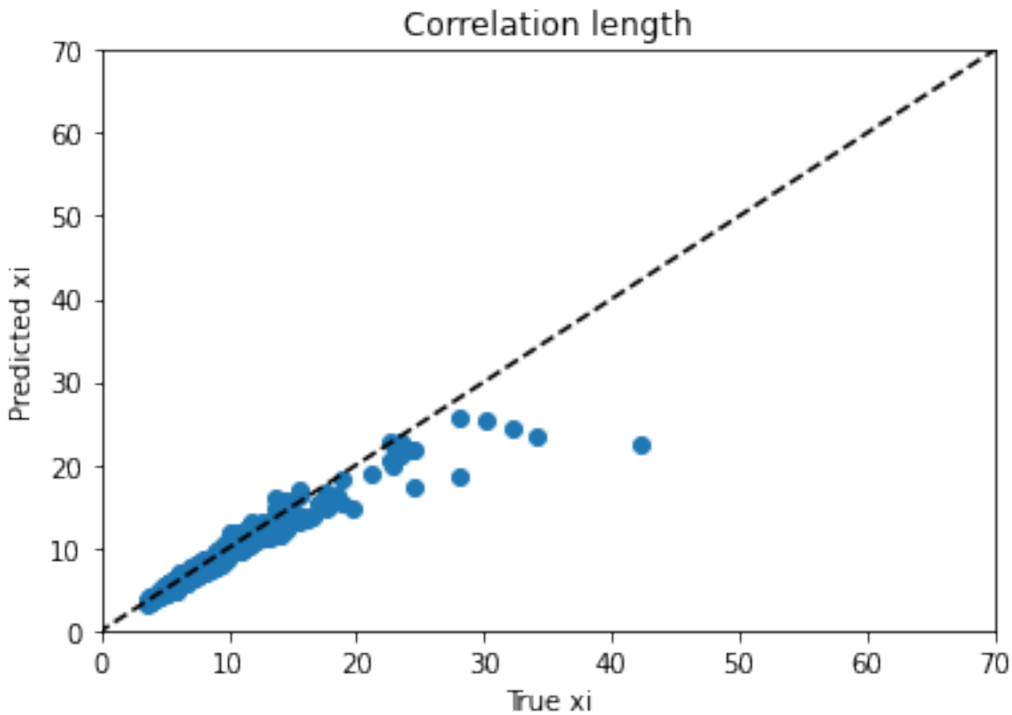
```python
# check results on test set

results = nn.evaluate(M_test_full,mat_test_full, batch_size=32);
print("test loss:", results)
nn_test_sc = sc.inverse_transform(nn.predict(M_test_full));
mat_test_sc = sc.inverse_transform(mat_test_full);

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")
```

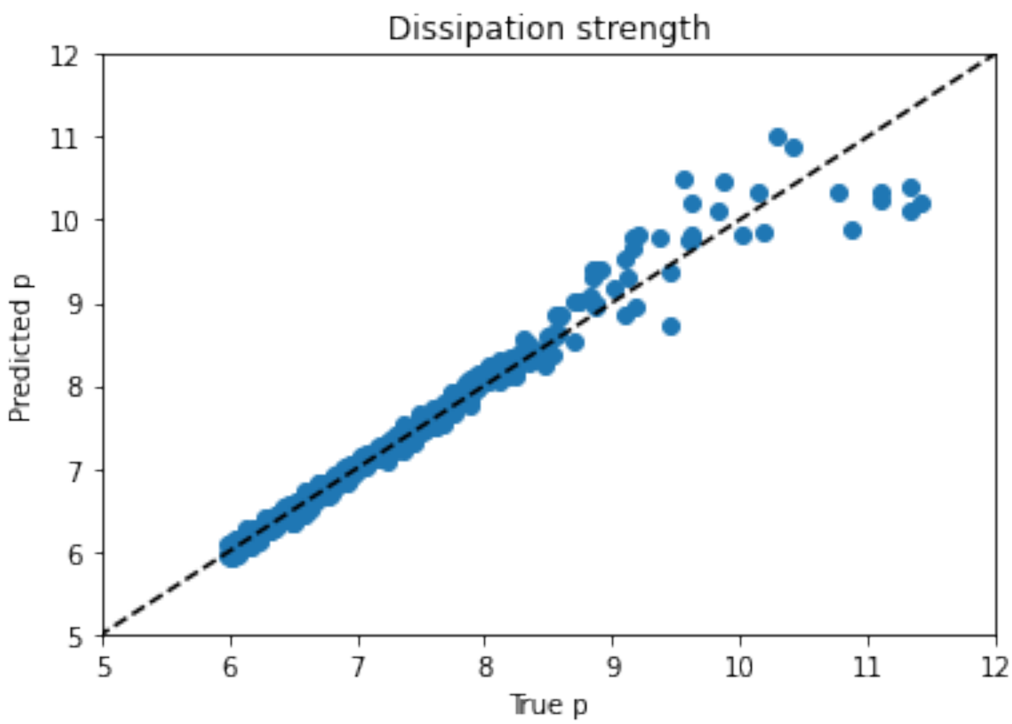
```
600/600 [==============================] - 0s 394us/sample - loss: 0.0131
test loss: 0.013097868983944256
```

[95]: Text(0.5, 1.0, 'Dissipation strength')





Correlation length
Predicted xi
True xi
0
10
20
30
40
50
60
70

Dissipation strength
Predicted p
True p
5
6
7
8
9
10
11
12

### 2.3.8 Prediction on evaluation data

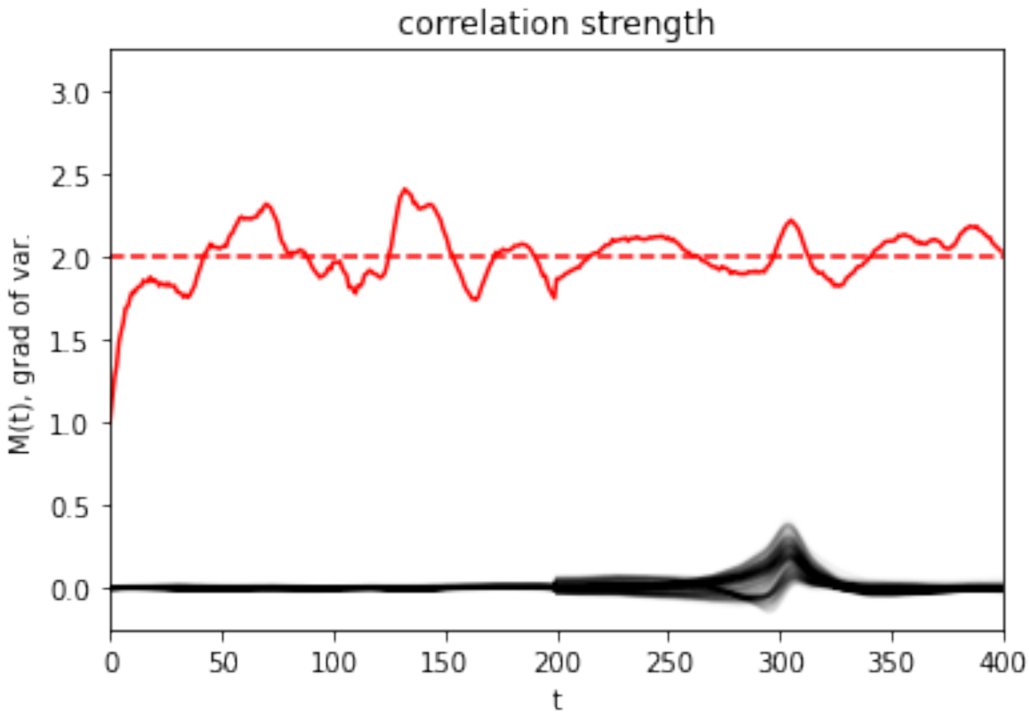
[100]:
```
y_eval = sc.inverse_transform(nn.predict(M_eval))
eval_file = f_prefix+"_mat_info_eval.txt"
np.savetxt(eval_file, y_eval, comments='#')
```

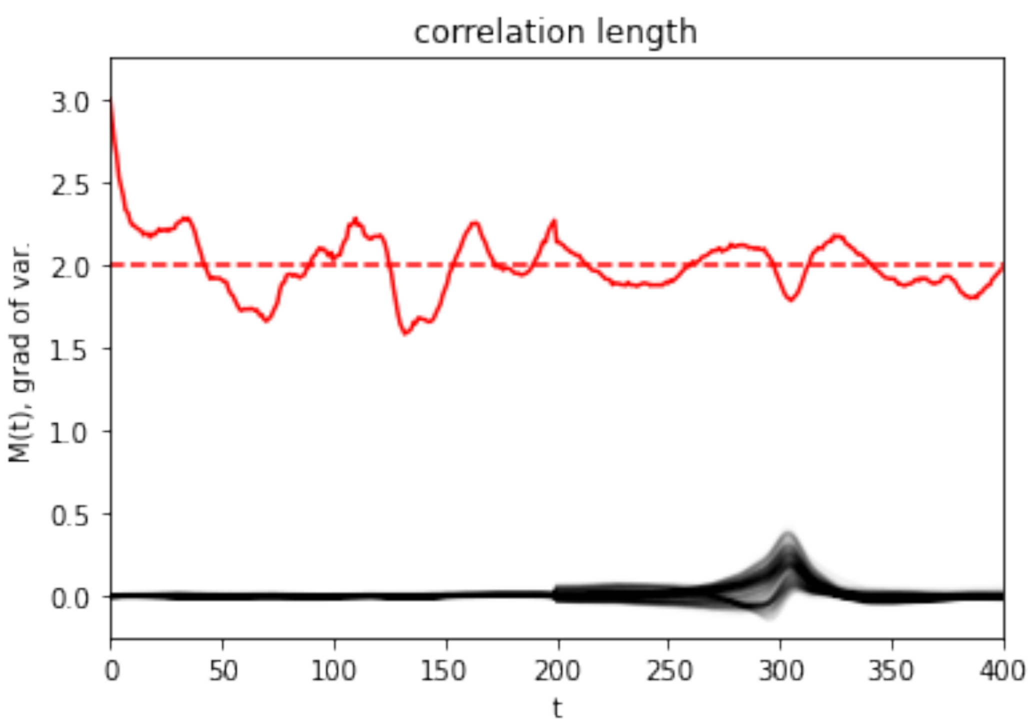
## 2.4 Heatmap of important features in the time domain

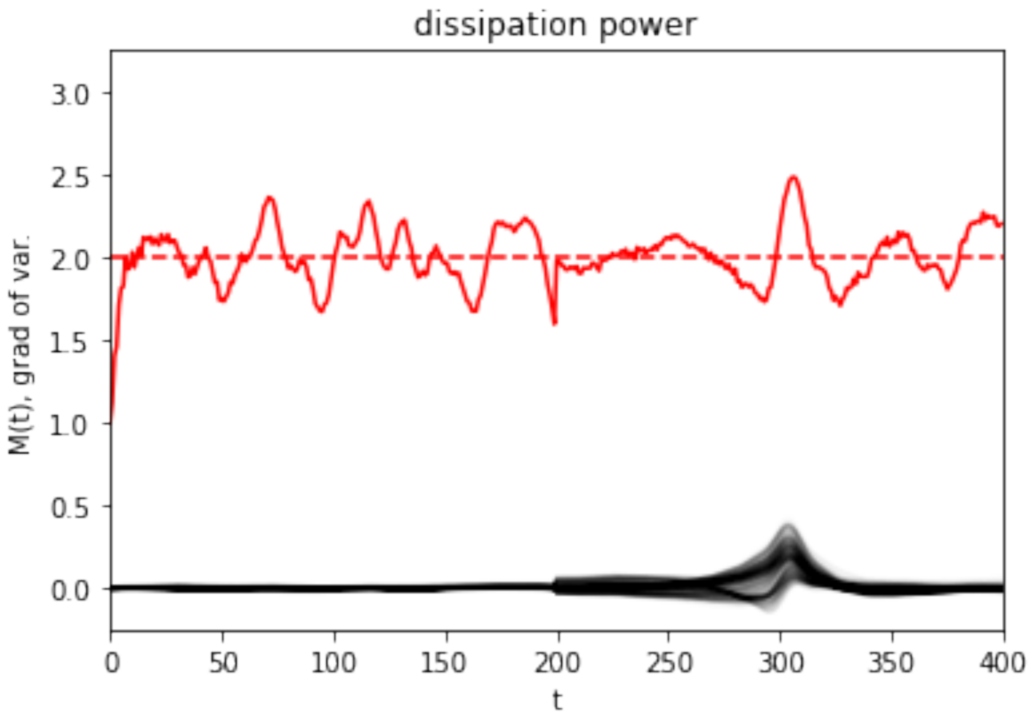
[345]:
```
# heatmap of feature importance in the time domain
from keras import backend as k
import tensorflow as tf

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(nn(in_tensor), tar_var, axis=1) # keep track of␣
↪the tar_var output

    grads = t.gradient(tar_output, in_tensor).numpy() # comput gradient using␣
↪tensorflow
    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.')
```

```
WARNING:tensorflow:Layer dense_1141 is casting an input tensor from dtype
float64 to the layer's dtype of float32, which is new behavior in TensorFlow 2.
The layer has dtype float32 because it's dtype defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this
warning. If in doubt, this warning is likely only an issue if you are porting a
TensorFlow 1.X model to TensorFlow 2.

To change all layers to have dtype float64 by default, call
`tf.keras.backend.set_floatx('float64')`. To change just this layer, pass
dtype='float64' to the layer constructor. If you are the author of this layer,
```

you can disable autocasting by passing autocast=False to the base Layer constructor.




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

[ ]:

Hack3-RKKY-16

April 12, 2021

# 1 Hackathon 3

# 2 Group 16

## 2.1 Load and view the simulated data

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[3]:
```
cd drive/MyDrive/Colab\ Notebooks/
```

```
/content/drive/MyDrive/Colab Notebooks
```

[ ]:
```
!mkdir Hack3 # No need to run for recompile the code
```

```
mkdir: cannot create directory ‘Hack3’: File exists
```

[4]:
```
cd Hack3/
```

```
/content/drive/MyDrive/Colab Notebooks/Hack3
```

[5]:
```
!pwd
```

```
/content/drive/MyDrive/Colab Notebooks/Hack3
```

[6]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
# No need to run since all are saved in my google drive
import requests

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

f_prefix = "gauss"
```

```python
# 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=1N1wVk5C64p2fy7kxx7fGpvQA8--Bq38W",allow_redirects=True)
open(mat_file, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1__SGeKUwQCXLZa83-nKH1Twhh99Lu7tb",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1kRYLhoi1ClSKQbKBnp9asI5_hOoST_Hd",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=1IWaUbkaLh4XbK8CWrx-VZ78RteKBcwVj",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=18N_p6aCJJp_xoYkws5vFX_-m0xqZDkaG",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=1wF0rJB-JpSYohH8MEV-a4E-uw5R5Dxd4",allow_redirects=True)
open(mat_file, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1bagiHH3-bGAbQIpZalBSPWxg4AAczfpP",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?export=download&id=1PvgRwdlJaDpsqElyU8oebfoaV2t13w35",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=10Cd91DR4qzFCqWonkjvJ9ZhhHphJ0i7q",allow_redirects=True)
open(M_file_r, "wb").write(r.content)
r = requests.get("https://docs.google.com/uc?
↪export=download&id=1Wrab6Dk9IgRKPuzeEUiB-C5xEiVoFynr",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
```

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

[7]:
```
#f_prefix = "gauss"; # Gaussian functional between nuclei
f_prefix = "RKKY"; # RKKY functional between nuclei
```

### 2.1.2 Now load the data and format it correctly

[8]:
```
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:  |      |      |   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
```

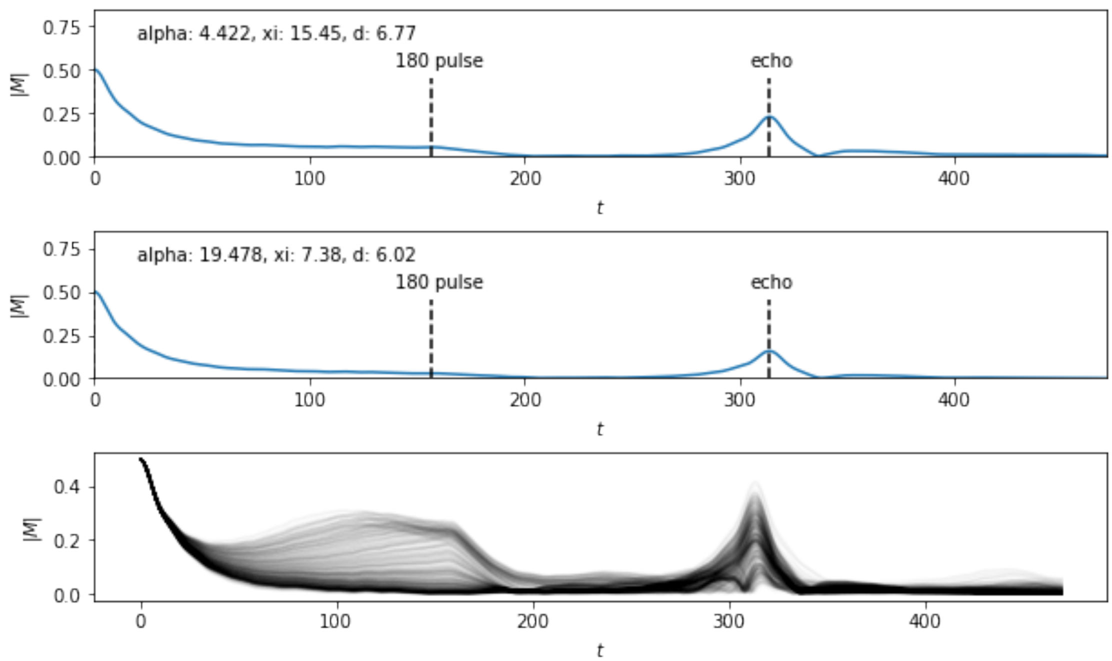
### 2.1.3 View the data with three plots, two with a specific curve and one with a lot of curves

[9]:
```
M.shape # 6000 different curves and 471 points in each curve?
```

[9]: (6000, 471)

[ ]:
```
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="$t$");
```

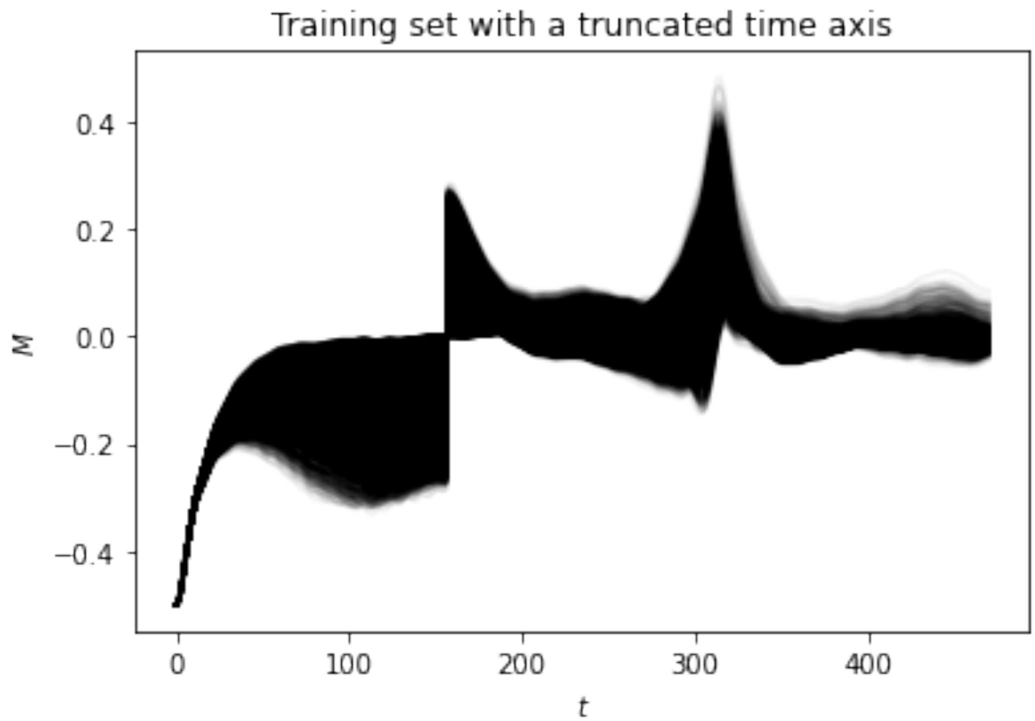
```
fig1.subplots_adjust(hspace=.5)
```




### 2.1.4 Truncate, scale, and partition the training/testing sets

[10]:
```
# 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
time_keep = range(0,471)
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)
#taking only the imaginry part only
M_trunc_imaginary = np.imag(M_trunc)
# 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_imaginary,␣
 ↪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[:,:]).T,color=(0,0,0,.05));
plt.xlabel("$t$")
plt.ylabel("$M$")
plt.title("Training set with a truncated time axis");
```




## 3 RKKY Model

[76]:
```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
from keras.optimizers import SGD # gradient descent optimizer
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from scipy.stats import reciprocal
import math
```

[81]:
```
def calculate_layer_nodes(n_layers, first_layer_nodes, last_layer_nodes):
  nodes_in_layers=[]
  nodes_difference = (first_layer_nodes-last_layer_nodes)/(n_layers-1)
  nodes_ = first_layer_nodes
  for i in range(n_layers):
    nodes_in_layers.append(math.ceil(nodes_))
    nodes_=nodes_ - nodes_difference
  return nodes_in_layers
```

[82]:
```
def create_model(n_layers=4, first_layer_nodes=100, last_layer_nodes=3,␣
↪activation='relu',
                loss='mean_squared_error', kernel_initializer='he_normal'):
  nodes_in_layers = calculate_layer_nodes(n_layers, first_layer_nodes,␣
↪last_layer_nodes)
  nn = Sequential()
  for i in range(n_layers-1):
    if i==0:
      nn.add(Dense(first_layer_nodes,input_dim=M_train.shape[1],␣
↪activation=activation, kernel_initializer=kernel_initializer))
      nn.add(BatchNormalization())
    else:
      nn.add(Dense(nodes_in_layers[i], activation=activation,␣
↪kernel_initializer=kernel_initializer))
      nn.add(BatchNormalization())
  nn.add(Dense(last_layer_nodes,activation='linear')) #relu or softplus will do␣
↪too
  nn.compile(loss=loss, optimizer=SGD(lr=0.01, momentum = 0.95) )
  return nn
```

[83]:
```
model = KerasRegressor(build_fn=create_model)
param_grid = {"n_layers" : [2,3,4,5], "first_layer_nodes" : [100,200,300,400],
    "activation" : ['relu','selu','elu','gelu','exponential','softplus'],
    "kernel_initializer" :␣
↪['he_normal','he_uniform','glorot_normal','glorot_uniform','lecun_normal','lecun_uniform'],
    "loss" : ['mean_squared_error','mean_absolute_error','huber_loss'],
    "batch_size" : [32,64,96,128]}
grid = RandomizedSearchCV(model, param_distributions=param_grid, cv=3,␣
↪n_iter=100)
```

[84]:
```
#call backs set up
early_stopping_cb = keras.callbacks.EarlyStopping(patience=20,␣
↪restore_best_weights=True)
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=10)
callbacks = [early_stopping_cb,lr_scheduler]
```

[17]:

```
#train the grid
history = grid.fit(M_train, mat_train, epochs=500,  verbose=2,␣
↪callbacks=callbacks, validation_split=0.2)
```

```
Streaming output truncated to the last 5000 lines.
Epoch 22/500
45/45 - 0s - loss: 0.2494 - val_loss: 0.2745
Epoch 23/500
45/45 - 0s - loss: 0.2517 - val_loss: 0.2643
Epoch 24/500
45/45 - 0s - loss: 0.2484 - val_loss: 0.2668
Epoch 25/500
45/45 - 0s - loss: 0.2530 - val_loss: 0.3085
Epoch 26/500
45/45 - 0s - loss: 0.2513 - val_loss: 0.2789
Epoch 27/500
45/45 - 0s - loss: 0.2536 - val_loss: 0.2813
Epoch 28/500
45/45 - 0s - loss: 0.2497 - val_loss: 0.2941
Epoch 29/500
45/45 - 0s - loss: 0.2491 - val_loss: 0.2831
Epoch 30/500
45/45 - 0s - loss: 0.2450 - val_loss: 0.2650
Epoch 31/500
45/45 - 0s - loss: 0.2465 - val_loss: 0.2679
Epoch 32/500
45/45 - 0s - loss: 0.2448 - val_loss: 0.2791
Epoch 33/500
45/45 - 0s - loss: 0.2507 - val_loss: 0.2676
Epoch 34/500
45/45 - 0s - loss: 0.2469 - val_loss: 0.2629
Epoch 35/500
45/45 - 0s - loss: 0.2427 - val_loss: 0.2773
Epoch 36/500
45/45 - 0s - loss: 0.2427 - val_loss: 0.2572
Epoch 37/500
45/45 - 0s - loss: 0.2439 - val_loss: 0.2565
Epoch 38/500
45/45 - 0s - loss: 0.2403 - val_loss: 0.2545
Epoch 39/500
45/45 - 0s - loss: 0.2391 - val_loss: 0.2645
Epoch 40/500
45/45 - 0s - loss: 0.2415 - val_loss: 0.2527
Epoch 41/500
45/45 - 0s - loss: 0.2459 - val_loss: 0.2780
Epoch 42/500
45/45 - 0s - loss: 0.2452 - val_loss: 0.2560
```

```
30/30 - 0s - loss: 0.5082 - val_loss: nan
Epoch 15/500
30/30 - 0s - loss: 0.4539 - val_loss: nan
Epoch 16/500
30/30 - 0s - loss: 0.4620 - val_loss: nan
Epoch 17/500
30/30 - 0s - loss: 0.4828 - val_loss: nan
Epoch 18/500
30/30 - 0s - loss: 0.4153 - val_loss: nan
Epoch 19/500
30/30 - 0s - loss: 0.4270 - val_loss: nan
Epoch 20/500
30/30 - 0s - loss: 0.4148 - val_loss: nan
Epoch 1/500
```

```
/usr/local/lib/python3.7/dist-
packages/sklearn/model_selection/_validation.py:536: FitFailedWarning: Estimator
fit failed. The score on this train-test partition for these parameters will be
set to nan. Details:
TypeError: object of type 'NoneType' has no len()

  FitFailedWarning)
```

```
Streaming output truncated to the last 5000 lines.
45/45 - 0s - loss: 0.4974 - val_loss: 0.5193
Epoch 119/500
45/45 - 0s - loss: 0.4979 - val_loss: 0.5540
Epoch 120/500
45/45 - 0s - loss: 0.4997 - val_loss: 0.5045
Epoch 121/500
45/45 - 0s - loss: 0.5079 - val_loss: 0.5404
Epoch 122/500
45/45 - 0s - loss: 0.5172 - val_loss: 0.4757
Epoch 123/500
45/45 - 0s - loss: 0.4976 - val_loss: 0.5268
Epoch 124/500
45/45 - 0s - loss: 0.4955 - val_loss: 0.5199
Epoch 125/500
45/45 - 0s - loss: 0.4995 - val_loss: 0.4708
Epoch 126/500
45/45 - 0s - loss: 0.4999 - val_loss: 0.4616
Epoch 127/500
45/45 - 0s - loss: 0.5015 - val_loss: 0.5116
Epoch 128/500
45/45 - 0s - loss: 0.5084 - val_loss: 0.4695
Epoch 129/500
45/45 - 0s - loss: 0.5010 - val_loss: 0.5514
Epoch 130/500
45/45 - 0s - loss: 0.4936 - val_loss: 0.5746
```

```
68/68 - 0s - loss: 0.0907 - val_loss: 1.4179
Epoch 32/500
68/68 - 0s - loss: 0.0953 - val_loss: 0.9198
Epoch 33/500
68/68 - 0s - loss: 0.0888 - val_loss: 0.7600
Epoch 34/500
68/68 - 0s - loss: 0.0963 - val_loss: 0.1655
Epoch 35/500
68/68 - 0s - loss: 0.0936 - val_loss: 0.5278
Epoch 36/500
68/68 - 0s - loss: 0.0942 - val_loss: 0.1620
Epoch 37/500
68/68 - 0s - loss: 0.0883 - val_loss: 0.2267
Epoch 38/500
68/68 - 0s - loss: 0.0834 - val_loss: 0.1144
Epoch 39/500
68/68 - 0s - loss: 0.0792 - val_loss: 0.4520
Epoch 40/500
68/68 - 0s - loss: 0.0773 - val_loss: 0.1156
Epoch 41/500
68/68 - 0s - loss: 0.0790 - val_loss: 0.1431
Epoch 42/500
68/68 - 0s - loss: 0.0795 - val_loss: 0.5620
Epoch 43/500
68/68 - 0s - loss: 0.0784 - val_loss: 0.0908
Epoch 44/500
68/68 - 0s - loss: 0.0778 - val_loss: 0.3193
Epoch 45/500
68/68 - 0s - loss: 0.0806 - val_loss: 0.1957
Epoch 46/500
68/68 - 0s - loss: 0.0756 - val_loss: 0.1733
```

[18]:
```
grid.best_score_
```

[18]:
```
-0.04847385982672373
```

[19]:
```
grid.best_params_
```

[19]:
```
{'activation': 'relu',
 'batch_size': 64,
 'drop_rate': 0.2,
 'first_layer_nodes': 400,
 'kernel_initializer': 'lecun_uniform',
 'loss': 'huber_loss',
 'n_layers': 5}
```

[20]:
```
grid.cv_results_
```

```
       0.00472993, 0.00291992, 0.0019318 , 0.00185217, 0.00175276,
       0.00104955, 0.00377488, 0.00171169, 0.00340907, 0.00223419,
       0.00271592, 0.00105025, 0.00059921, 0.00236257, 0.00569641,
       0.00318412, 0.0016357 , 0.00298635, 0.00594486, 0.00744274,
       0.03743035, 0.00307801, 0.00387476, 0.00086066, 0.00161959,
       0.0042237 , 0.0084399 , 0.00081697, 0.00245124, 0.0080435 ,
       0.00176692, 0.00143641, 0.00128422, 0.00132806, 0.00082137,
       0.00383163, 0.00414593, 0.00320754, 0.0026638 , 0.00339059,
       0.00338032, 0.0040446 , 0.00286569, 0.00339056, 0.00538244]),
 'std_test_score': array([1.85985135e-02, 9.02704731e-03, 1.71700876e-02,
3.37409264e-02,
                  nan, 1.99984576e-03,            nan, 1.41332360e-02,
       1.68412121e-02, 2.73453853e-02, 1.23027219e-01, 1.34944897e-01,
       1.07977120e-02,            nan, 4.20759126e-03, 2.03513613e-01,
       4.07470991e-03,            nan, 2.54307481e-02,            nan,
       3.85116505e-02, 2.69665112e-03, 5.40585931e-02,            nan,
       3.66016831e-03, 2.33576151e-01, 1.23884098e-02, 3.40346379e-01,
       5.41644436e-02, 6.48470579e-03, 8.44942936e-02, 7.41465584e-02,
       9.58055725e-02, 2.56318025e-02, 2.00306907e-01, 2.23099748e-02,
       1.30276133e-01, 2.48573212e-01, 1.07595679e-01, 2.90841784e-02,
       7.93857563e-03, 4.00075459e-02, 3.88635705e+17, 1.22257467e-01,
       3.23991211e-02, 5.24635563e-02, 1.64385833e-02, 1.11575136e-02,
       9.63003363e-02, 1.11351142e-02, 8.96885050e-03, 7.76378711e-02,
       5.71672117e-02, 9.28190141e-02, 1.22793872e-02, 5.95576913e-02,
       1.31080735e-01, 2.10408074e-01, 7.43081489e-03, 5.14152173e-03,
       7.10651523e-03, 2.02955110e-02, 2.55317170e-02, 8.63277552e-03,
       3.76690328e-03, 6.09035595e-03, 1.84148250e-03, 9.01036728e-03,
       8.46493913e-03, 1.04462521e-01, 1.70118475e-02, 1.02493177e-02,
       1.44080761e-01, 1.23649946e-02, 8.48391483e-03,            nan,
       6.68070317e-03, 1.31490289e-01, 3.31190011e-02, 4.42176541e-03,
       2.26728028e+28, 5.84783001e-02, 2.68331105e-03, 1.39916679e-02,
       7.05863390e-03, 1.61208579e-02, 4.90997073e-02, 1.81067409e-01,
       1.57867612e-01, 9.72487813e-02, 1.95154648e-01, 1.50807102e-02,
       5.09049260e-03,            nan, 5.25886862e-01, 1.14496546e-01,
       1.50164512e-01, 2.26902864e-02, 1.25006205e-02, 5.83468363e-02])}
```

[90]:
```
nn = create_model(n_layers=5, first_layer_nodes=400, activation='relu',␣
 ↪loss='huber_loss', kernel_initializer='lecun_uniform')
```

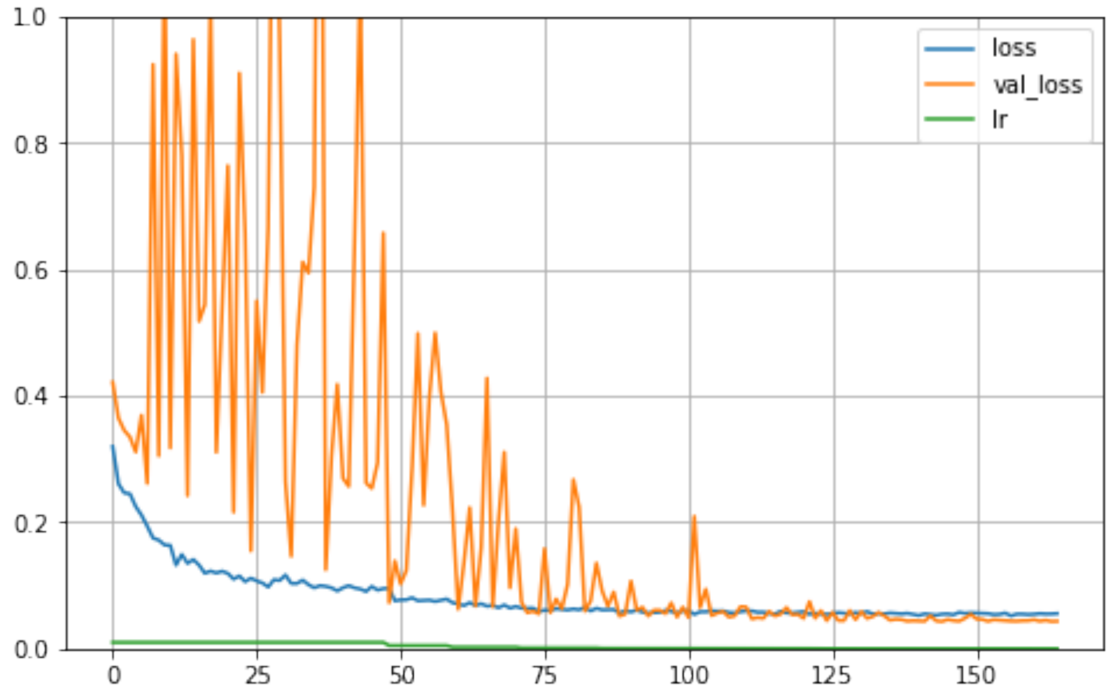
[91]:
```
history = nn.fit(M_train, mat_train, epochs=500,batch_size=64, verbose=2,␣
 ↪callbacks=callbacks, validation_split=0.2)
```

```
Epoch 1/500
68/68 - 1s - loss: 0.3199 - val_loss: 0.4217
Epoch 2/500
68/68 - 0s - loss: 0.2608 - val_loss: 0.3649
Epoch 3/500
```

```
68/68 - 0s - loss: 0.0537 - val_loss: 0.0445
Epoch 148/500
68/68 - 0s - loss: 0.0575 - val_loss: 0.0436
Epoch 149/500
68/68 - 0s - loss: 0.0559 - val_loss: 0.0482
Epoch 150/500
68/68 - 0s - loss: 0.0569 - val_loss: 0.0545
Epoch 151/500
68/68 - 0s - loss: 0.0563 - val_loss: 0.0471
Epoch 152/500
68/68 - 0s - loss: 0.0560 - val_loss: 0.0464
Epoch 153/500
68/68 - 0s - loss: 0.0556 - val_loss: 0.0434
Epoch 154/500
68/68 - 0s - loss: 0.0540 - val_loss: 0.0459
Epoch 155/500
68/68 - 0s - loss: 0.0544 - val_loss: 0.0451
Epoch 156/500
68/68 - 0s - loss: 0.0565 - val_loss: 0.0441
Epoch 157/500
68/68 - 0s - loss: 0.0519 - val_loss: 0.0438
Epoch 158/500
68/68 - 0s - loss: 0.0548 - val_loss: 0.0436
Epoch 159/500
68/68 - 0s - loss: 0.0548 - val_loss: 0.0443
Epoch 160/500
68/68 - 0s - loss: 0.0541 - val_loss: 0.0445
Epoch 161/500
68/68 - 0s - loss: 0.0546 - val_loss: 0.0459
Epoch 162/500
68/68 - 0s - loss: 0.0556 - val_loss: 0.0436
Epoch 163/500
68/68 - 0s - loss: 0.0550 - val_loss: 0.0451
Epoch 164/500
68/68 - 0s - loss: 0.0551 - val_loss: 0.0432
Epoch 165/500
68/68 - 0s - loss: 0.0557 - val_loss: 0.0435
```

[92]:
```
import pandas as pd
```

[93]:
```
pd.DataFrame(history.history).plot(figsize=(8,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.show()
```

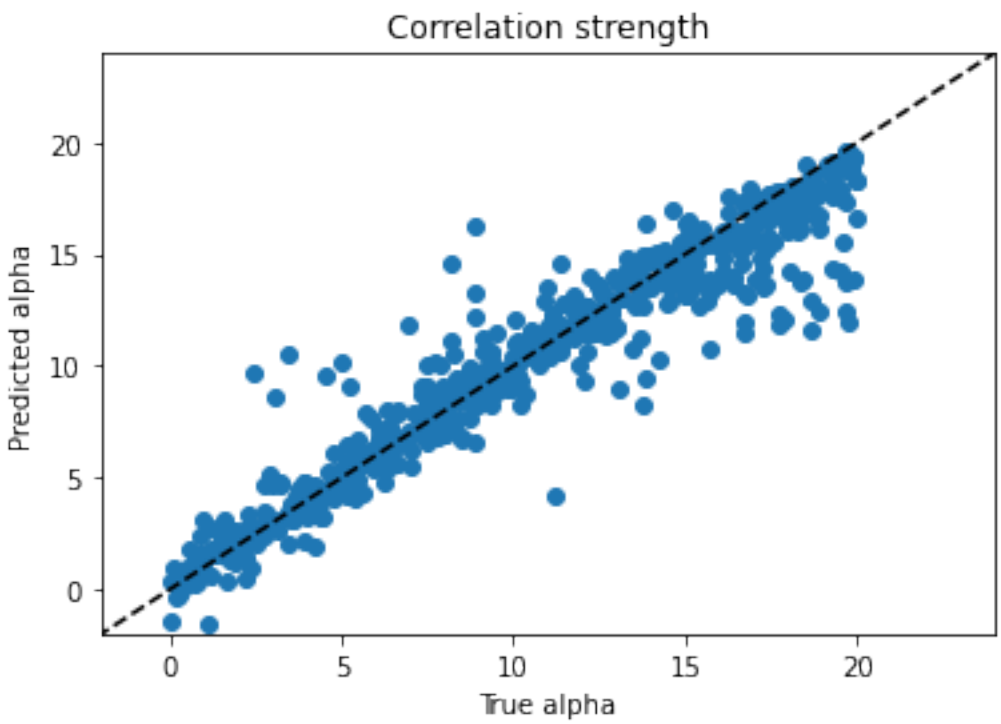


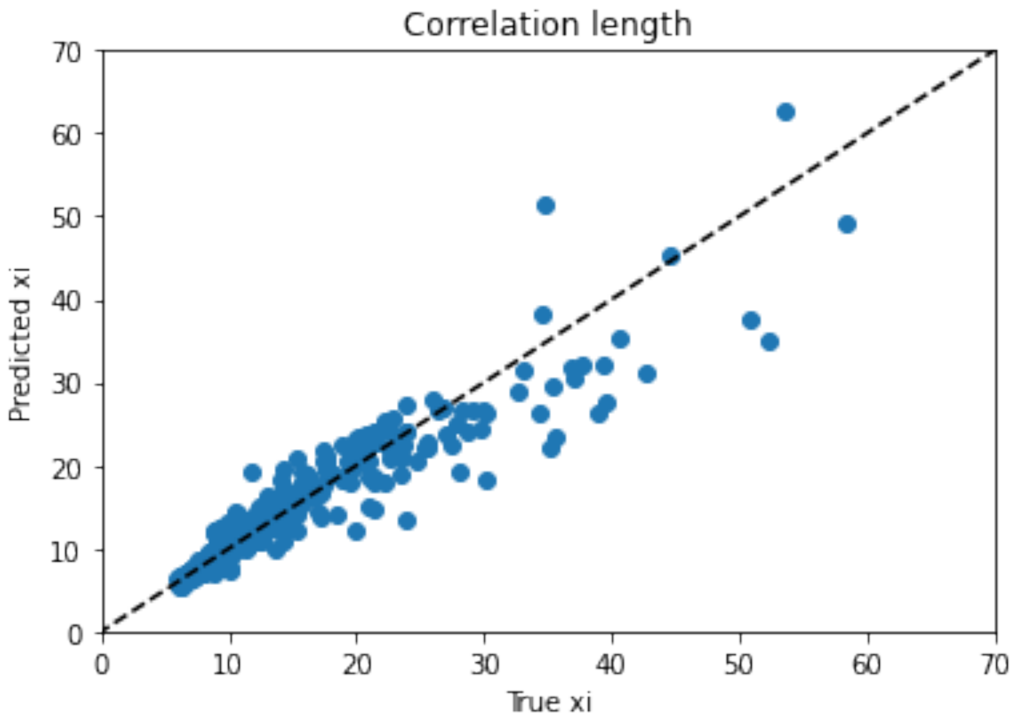

[94]:
```
# check results on test set

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

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]);
```

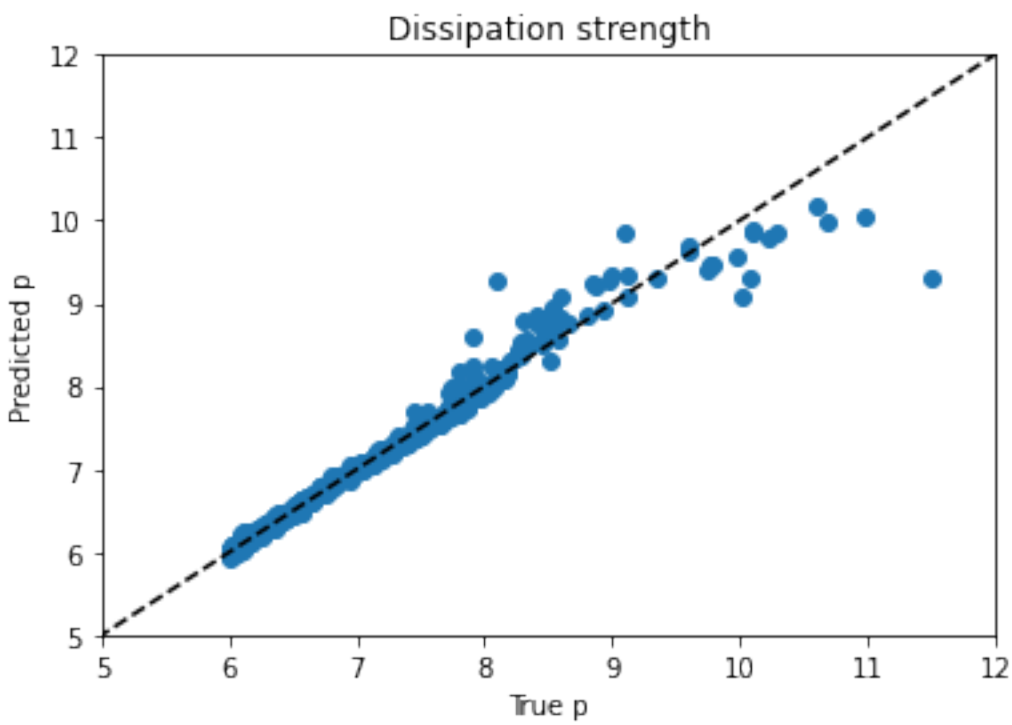
```
plt.plot([-100, 100],[-100, 100],"--k")
plt.xlabel("True p");
plt.ylabel("Predicted p");
plt.axis([5, 12, 5, 12])
plt.title("Dissipation strength")
```

```
19/19 [==============================] - 0s 1ms/step - loss: 0.0372
test loss: 0.03717200458049774
```

[94]: Text(0.5, 1.0, 'Dissipation strength')




Correlation length
Predicted xi
True xi
0
10
20
30
40
50
60
70


Dissipation strength
Predicted p
True p
5
6
7
8
9
10
11
12

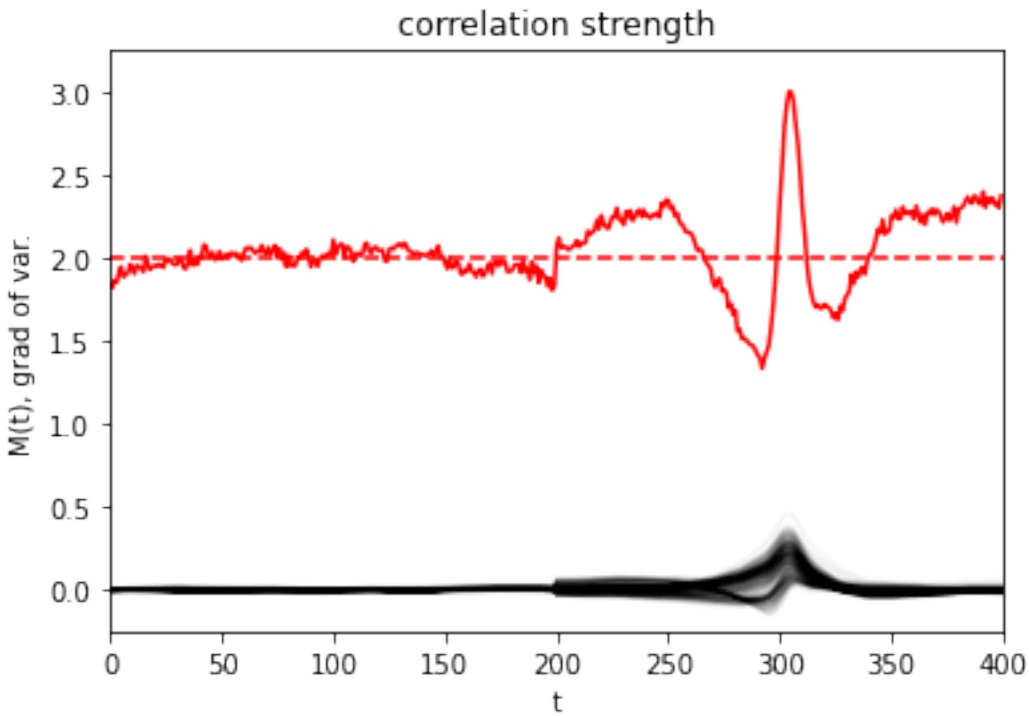
[95]:

```
M_eval_imaginary = np.imag(M_eval)
mat_eval = sc.inverse_transform(nn.predict(M_eval_imaginary))
filename = f_prefix + "_mat_info_eval.txt"
np.savetxt(filename,mat_eval,delimiter='\t')
```

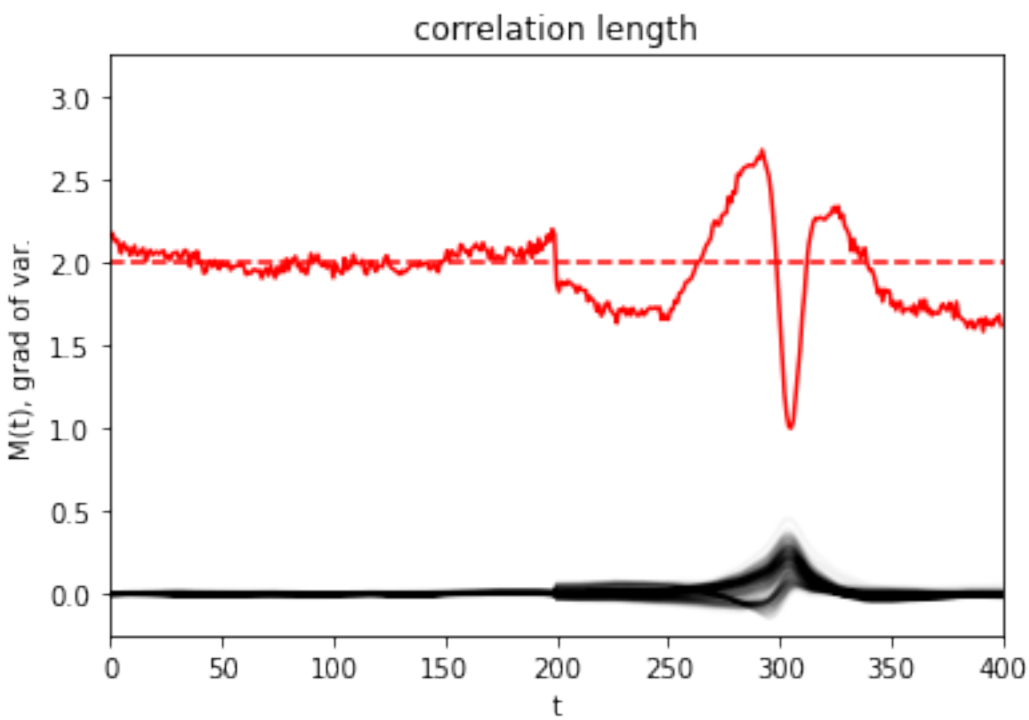
### 3.1 Heatmap of important features in the time domain

[ ]:

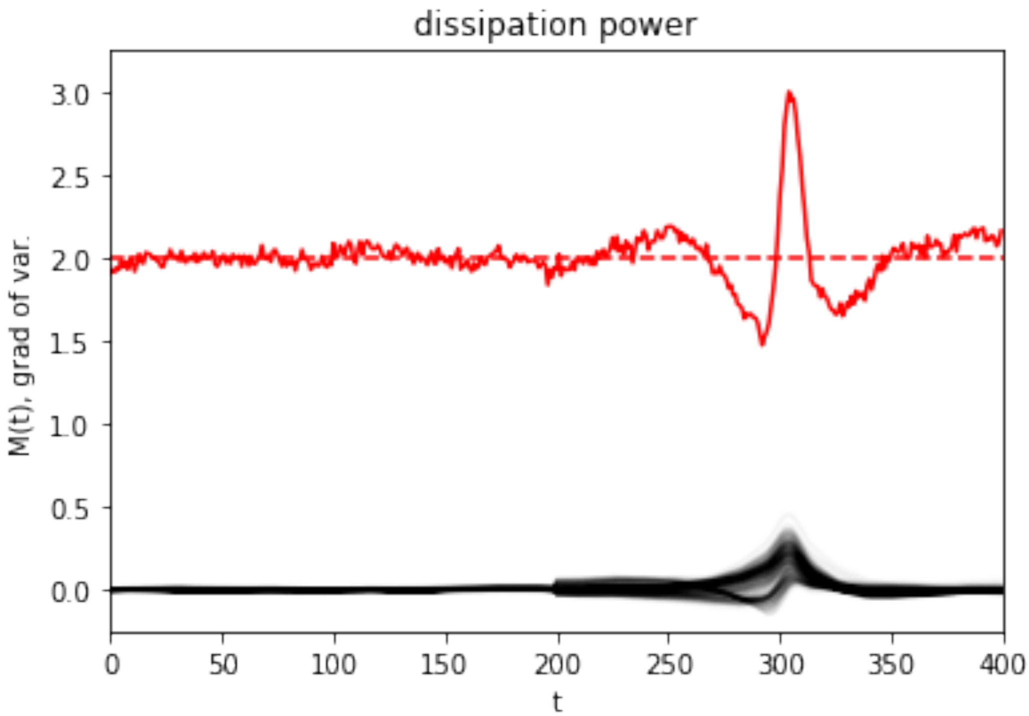
```
# heatmap of feature importance in the time domain
from keras import backend as k
import tensorflow as tf

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(nn(in_tensor), tar_var, axis=1) # keep track of␣
 ↪the tar_var output

    grads = t.gradient(tar_output, in_tensor).numpy() # comput gradient using␣
 ↪tensorflow
    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.
t

correlation length
M(t), grad of var.
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