This python notebook was downloaded from:

[nanoHub]

Using neural networks to predict and classify crystal structures of elements

Outline:

1. Getting a dataset
2. Processing and Organizing Data
3. Creating the Model
4. Plotting

1. Getting a dataset

Datasets containing properties for the elements in the periodic table are available online; however, it would be thematic to create our own, using the tools from the first tutorial on MSEML Query_Viz. In this section we will query both Pymatgen and Mendeleev to get a complete set of properties per element. We will use this data to create the cases from which the model will train and test.

In this first snippet of code we will import all relevant libraries, the elements that will be turned into cases and the properties that will serve as the attributes for the cases. We will get 47 entries (which is a small dataset), but should give us a somewhat accurate prediction. It is important to note that more entries would move the prediction closer to the real value, and so would more attributes.

The elements listed were chosen because querying them for these properties yields a dataset with no unknown values, and because they represent the three most common crystallographic structures.

```
import tensorflow as tf
from tensorflow import keras
from keras import initializers
from keras.layers import Dense
from keras.models import Sequential
from tensorflow.keras import optimizers
import pymatgen.core as pymat
import mendeleev as mendel
import pandas as pd
import numpy as np
import random

%matplotlib inline
import matplotlib.pyplot as plt
import sys

fcc_elements = ["Ag", "Al", "Au", "Cu", "Ir", "Ni", "Pb", "Pd", "Pt", "Rh", "Th", "Yb"]
bcc_elements = ["Ba", "Ca", "Cr", "Cs", "Eu", "Fe", "Li", "Mn", "Mo", "Na", "Nb", "Rb", "Ta", "V", "W" ]
hcp_elements = ["Be", "Cd", "Co", "Dy", "Er", "Gd", "Hf", "Ho", "Lu", "Mg", "Re",
                "Ru", "Sc", "Tb", "Ti", "Tl", "Tm", "Y", "Zn", "Zr"]

elements = fcc_elements + bcc_elements + hcp_elements

random.Random(1).shuffle(elements)

#Note that different properties have been added from the two databases.
querable_mendeleev = ["atomic_number", "atomic_volume", "boiling_point", "en_ghosh",  "evaporation_heat", "heat_of_formation",
                      "lattice_constant", "melting_point", "specific_heat"]
querable_pymatgen = ["atomic_mass", "atomic_radius", "electrical_resistivity","molar_volume", "bulk_modulus", "youngs_modulus",
                      "average_ionic_radius", "density_of_solid", "coefficient_of_linear_thermal_expansion"]
querable_values = querable_mendeleev + querable_pymatgen
```

As before, we will use the database queries to populate lists which can be displayed by the Pandas library in a user-friendly table with the properties as the column headers.

```
all_values = [] # Values for Attributes
all_labels = [] # Crystal structure labels (0 = fcc, 1 = bcc, 2 = hcp)

for item in elements:
    element_values = []

    # This section queries Mendeleev
    element_object = mendel.element(item)
    for i in querable_mendeleev:
        element_values.append(getattr(element_object,i))

    # This section queries Pymatgen
    element_object = pymat.Element(item)
    for i in querable_pymatgen:
```

```
        element_values.append(getattr(element_object,i))

    all_values.append(element_values) # All lists are appended to another list, creating a List of Lists

    if (item in fcc_elements):
        all_labels.append([1, 0, 0]) # The crystal structure labels are assigned here
    elif (item in bcc_elements):
        all_labels.append([0, 1, 0]) # The crystal structure labels are assigned here
    elif (item in hcp_elements):
        all_labels.append([0, 0, 1]) # The crystal structure labels are assigned here

#print("What is inside element_values")
#print(element_values)
#print("What is inside all_labels")
#print(all_labels)

# Pandas Dataframe
df = pd.DataFrame(all_values, columns=querable_values)

# We will patch some of the values that are not available in the datasets.

# Value for the CTE of Cesium
index_Cs = df.index[df['atomic_number'] == 55]
df.iloc[index_Cs, df.columns.get_loc("coefficient_of_linear_thermal_expansion")] = 0.000097
# Value from: David R. Lide (ed), CRC Handbook of Chemistry and Physics, 84th Edition. CRC Press. Boca Raton, Florida, 2003

# Value for the CTE of Rubidium
index_Rb = df.index[df['atomic_number'] == 37]
df.iloc[index_Rb, df.columns.get_loc("coefficient_of_linear_thermal_expansion")] = 0.000090
# Value from: https://www.azom.com/article.aspx?ArticleID=1834

# Value for the Evaporation Heat of Ruthenium
index_Ru = df.index[df['atomic_number'] == 44]
df.iloc[index_Ru, df.columns.get_loc("evaporation_heat")] = 595 # kJ/mol
# Value from: https://www.webelements.com/ruthenium/thermochemistry.html

# Value for the Bulk Modulus of Zirconium
index_Zr = df.index[df['atomic_number'] == 40]
df.iloc[index_Zr, df.columns.get_loc("bulk_modulus")] = 94 # GPa
# Value from: https://materialsproject.org/materials/mp-131/

df.head
```

```
/usr/local/lib/python3.10/dist-packages/pymatgen/core/periodic_table.py:212: UserWarning:

No data available for coefficient_of_linear_thermal_expansion for Cs

/usr/local/lib/python3.10/dist-packages/pymatgen/core/periodic_table.py:212: UserWarning:

No data available for coefficient_of_linear_thermal_expansion for Rb

/usr/local/lib/python3.10/dist-packages/pymatgen/core/periodic_table.py:212: UserWarning:

No data available for bulk_modulus for Zr
```

```
pandas.core.generic.NDFrame.head
def head(n: int=5) -> NDFrameT
```

```
Viewing the first 5 lines

>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey

Viewing the first `n` lines (three in this case)

>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon

For negative values of `n`

>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

2. Processing and Organizing Data

We again normalize the data and organize it into training and testing sets as before.

SETS

We have 47 elements for which the crystal structure is known and we will use 40 of these as a training set and the remaining 7 as testing set.

NORMALIZATION

We will again use the Standard Score Normalization, which subtracts the mean of the feature and divide by its standard deviation.

$\frac{X-\mu}{\sigma}$

While our model might converge without feature normalization, the resultant model would be difficult to train and would be dependent on the choice of units used in the input.

```
# SETS

all_values = [list(df.iloc[x]) for x in range(len(all_values))]

# List of lists are turned into Numpy arrays to facilitate calculations in steps to follow (Normalization).
all_values = np.array(all_values, dtype = float)
print("Shape of Values:", all_values.shape)
all_labels = np.array(all_labels, dtype = int)
print("Shape of Labels:", all_labels.shape)

# Training Set
train_values = all_values[:40, :]
train_labels = all_labels[:40, :]

# Testing Set
test_values = all_values[-7:, :]
test_labels = all_labels[-7:, :]

# NORMALIZATION

mean = np.nanmean(train_values, axis = 0) # mean, np.nanmean is a version of mean function in numpy that throws errors if all slices of data are NaN.

std = np.nanstd(train_values, axis = 0) # standard deviation

train_values = (train_values - mean) / std # input scaling
test_values = (test_values - mean) / std # input scaling

print(train_values[0]) # print a sample entry from the training set
#print(train_labels[0])
```

```
Shape of Values: (47, 18)
Shape of Labels: (47, 3)
[-0.80084167 -0.75983551  0.02340813 -0.40732945  0.15599373  0.16654528
 -1.09549525  0.09167774 -0.03493069 -0.82400017 -0.80570946 -0.67799461
 -0.75661221  0.70972845  0.6516648  -0.77257498  0.11409173 -0.3075323 ]
```

3. Creating the Model

For this classification, we will use a simple sequential neural network with one densely connected hidden layer. We will try many optimizers.

To learn more about Root Mean Squared Propagation, click here.

The key difference between the regression model and the classification model is our metric to measure network performance. While we used mean squared error (between the true outputs and the network's predicted output) for the regression task, we use categorical cross entropy (click here to learn more about it), using classification accuracy as a metric where higher accuracy implies a better network.

```
# DEFINITION OF THE MODEL

# The weights of our neural network will be initialized in a random manner, using a seed allows for reproducibility
kernel_init = initializers.RandomNormal(seed=14)

model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(train_values.shape[1],), kernel_initializer=kernel_init))
#model.add(Dense(16, activation='relu', kernel_initializer=kernel_init))
model.add(Dense(3, activation=tf.nn.softmax))  # Output Layer

# DEFINITION OF THE OPTIMIZER

#optimizer = optimizers.RMSprop(0.002) # AdaM Optimizer. 0.002 is the learning rate.
#optimizer = optimizers.Adam(0.002) # AdaM Optimizer. 0.002 is the learning rate.
optimizer = optimizers.SGD(0.002) # AdaM Optimizer. 0.002 is the learning rate.

# This line matches the optimizer to the model and states which metrics will evaluate the model's accuracy
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 32)                608

 dense_3 (Dense)             (None, 3)                 99

=================================================================
Total params: 707 (2.76 KB)
Trainable params: 707 (2.76 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
kernel_init = initializers.RandomNormal(seed=14)

model1 = Sequential()
model1.add(Dense(32, activation='relu', input_shape=(train_values.shape[1],), kernel_initializer=kernel_init))
#model.add(Dense(16, activation='relu', kernel_initializer=kernel_init))
model1.add(Dense(3, activation=tf.nn.softmax))  # Output Layer

# DEFINITION OF THE OPTIMIZER

#optimizer = optimizers.RMSprop(0.002) # AdaM Optimizer. 0.002 is the learning rate.
optimizer1 = optimizers.Adam(0.002) # AdaM Optimizer. 0.002 is the learning rate.
#optimizer = optimizers.SGD(0.002) # AdaM Optimizer. 0.002 is the learning rate.

# This line matches the optimizer to the model and states which metrics will evaluate the model's accuracy
model1.compile(loss='categorical_crossentropy', optimizer=optimizer1, metrics=['accuracy'])
model1.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 32)                608

 dense_5 (Dense)             (None, 3)                 99

=================================================================
Total params: 707 (2.76 KB)
Trainable params: 707 (2.76 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
kernel_init = initializers.RandomNormal(seed=14)

model2 = Sequential()
model2.add(Dense(32, activation='relu', input_shape=(train_values.shape[1],), kernel_initializer=kernel_init))
#model.add(Dense(16, activation='relu', kernel_initializer=kernel_init))
model2.add(Dense(3, activation=tf.nn.softmax))  # Output Layer

# DEFINITION OF THE OPTIMIZER

optimizer2 = optimizers.RMSprop(0.002) # AdaM Optimizer. 0.002 is the learning rate.
#optimizer = optimizers.Adam(0.002) # AdaM Optimizer. 0.002 is the learning rate.
#optimizer = optimizers.SGD(0.002) # AdaM Optimizer. 0.002 is the learning rate.

# This line matches the optimizer to the model and states which metrics will evaluate the model's accuracy
model2.compile(loss='categorical_crossentropy', optimizer=optimizer2, metrics=['accuracy'])
model2.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 dense_6 (Dense)             (None, 32)                608

 dense_7 (Dense)             (None, 3)                 99

=================================================================
Total params: 707 (2.76 KB)
Trainable params: 707 (2.76 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

TRAINING

This model is trained for 500 epochs, and we record the training accuracy in the history object. This way, by plotting "history" we can see the evolution of the "learning" of the model, that is the decrease of the Mean Absolute Error. Models in Keras are fitted to the training set using the **fit** method.

One **Epoch** occurs when you pass the entire dataset through the model. One **Batch** contains a subset of the dataset that can be fed to the model at the same time. A more detailed explanation of these concepts can be found in this blog. As we have a really small dataset compared to the ones that are usually considered to be modeled by these neural networks, we are feeding all entries at the same time, so our batch is the entire dataset, and an epoch occurs when the batch is processed.

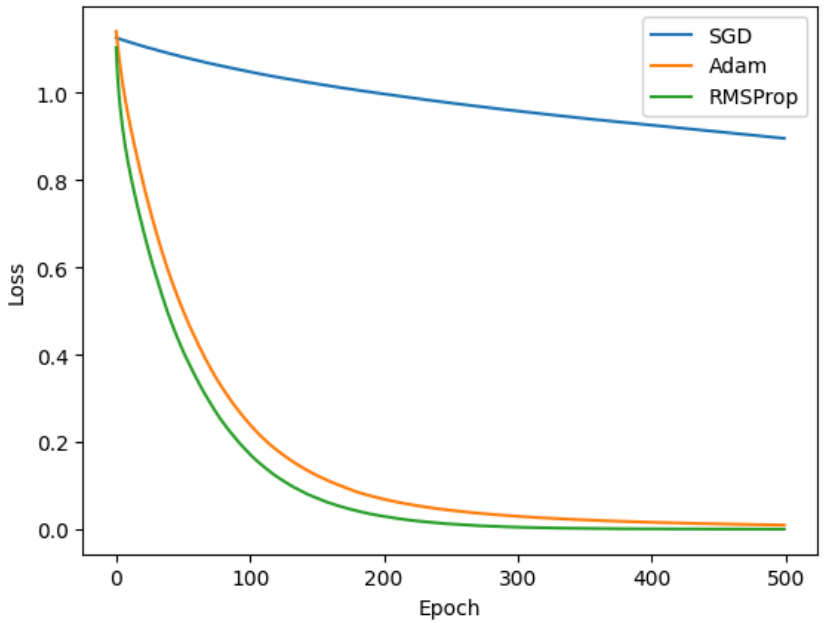
```
class PrintEpNum(keras.callbacks.Callback): # This is a function for the Epoch Counter
    def on_epoch_end(self, epoch, logs):
        sys.stdout.flush()
        sys.stdout.write("Current Epoch: " + str(epoch+1) + '\r') # Updates current Epoch Number


EPOCHS = 500 # Number of EPOCHS

# HISTORY Object which contains how the model learned
history = model.fit(train_values, train_labels, batch_size=train_values.shape[0], \
                    epochs=EPOCHS, validation_split=0.1, verbose = False, callbacks=[PrintEpNum()])
history1 = model1.fit(train_values, train_labels, batch_size=train_values.shape[0], \
                    epochs=EPOCHS, validation_split=0.1, verbose = False, callbacks=[PrintEpNum()])
history2 = model2.fit(train_values, train_labels, batch_size=train_values.shape[0], \
                    epochs=EPOCHS, validation_split=0.1, verbose = False, callbacks=[PrintEpNum()])
```

```
# PLOTTING HISTORY USING MATPLOTLIB

plt.figure()
plt.xlabel('Epoch')
plt.ylabel('Loss')
#plt.ylim(0.98,1.05)
plt.plot(history.epoch, np.array(history.history['loss']),label='SGD')
plt.plot(history1.epoch, np.array(history1.history['loss']),label='Adam')
plt.plot(history2.epoch, np.array(history2.history['loss']),label='RMSProp')
#plt.plot(history.epoch, np.array(history.history['val_accuracy']),label = 'Validation Accuracy')
plt.legend()
plt.show()
```



TESTING

Models in Keras are tested using the method **evaluate**. This method returns the classification accuracy on the training and the testing sets.

```
loss, acc = model.evaluate(train_values, train_labels, verbose=1)

print("Training Set Accuracy: %f" %(acc))

loss, acc = model.evaluate(test_values, test_labels, verbose=1)

print("Testing Set Accuracy: %f" %(acc))
```

```
2/2 [==============================] - 0s 8ms/step - loss: 0.8977 - accuracy: 0.7500
Training Set Accuracy: 0.750000
1/1 [==============================] - 0s 95ms/step - loss: 0.9273 - accuracy: 0.7143
Testing Set Accuracy: 0.714286
```

```
loss, acc = model1.evaluate(train_values, train_labels, verbose=1)

print("Training Set Accuracy: %f" %(acc))

loss, acc = model1.evaluate(test_values, test_labels, verbose=1)

print("Testing Set Accuracy: %f" %(acc))
```

```
2/2 [==============================] - 0s 7ms/step - loss: 0.0165 - accuracy: 1.0000
Training Set Accuracy: 1.000000
1/1 [==============================] - 0s 23ms/step - loss: 0.8197 - accuracy: 0.5714
Testing Set Accuracy: 0.571429
```

```
loss, acc = model2.evaluate(train_values, train_labels, verbose=1)

print("Training Set Accuracy: %f" %(acc))

loss, acc = model2.evaluate(test_values, test_labels, verbose=1)

print("Testing Set Accuracy: %f" %(acc))
```

```
2/2 [==============================] - 0s 6ms/step - loss: 0.0013 - accuracy: 1.0000
Training Set Accuracy: 1.000000
1/1 [==============================] - 0s 24ms/step - loss: 1.7161 - accuracy: 0.7143
Testing Set Accuracy: 0.714286
```

+ Code + Text

MAKING PREDICTIONS

The last step in a Regression Model is to make predictions for values not in the training set, which are determined by the method **predict**. In the following cell we print the Elements in the testing set, the real values for their Young's Moduli and the predictions generated by the Machine Learning model.

```
train_predictions = model.predict(train_values)
test_predictions = model.predict(test_values)

all_labels = np.vstack((train_labels, test_labels))
all_predictions = np.vstack((train_predictions, test_predictions))

predicted_labels = []
true_labels = []

for i in range(all_predictions.shape[0]):
    if (np.argmax(all_predictions[i]) == 0): # np.argmax returns the index of maximum value along an axis.
        # Here we are looking for the value
        predicted_labels.append("FCC")
    if (np.argmax(all_labels[i]) == 0):
        true_labels.append("FCC")
    if (np.argmax(all_predictions[i]) == 1):
        predicted_labels.append("BCC")
    if (np.argmax(all_labels[i]) == 1):
        true_labels.append("BCC")
    if (np.argmax(all_predictions[i]) == 2):
        predicted_labels.append("HCP")
    if (np.argmax(all_labels[i]) == 2):
        true_labels.append("HCP")

predicted_labels = np.array(predicted_labels).reshape((-1, 1))
true_labels = np.array(true_labels).reshape((-1, 1))
headings = ["Atomic number", "True crystal structure", "Predicted crystal structure"]

atomic_number_array = np.array(df.iloc[:, 0]).reshape((-1, 1))
plot_table = np.concatenate((atomic_number_array, true_labels, predicted_labels), axis=1)

plot_df = pd.DataFrame(plot_table, columns=headings)
```

```
2/2 [==============================] - 0s 5ms/step
1/1 [==============================] - 0s 21ms/step
```

```
plot_df
```

	Atomic number	True crystal structure	Predicted crystal structure
0	27	HCP	HCP
1	69	HCP	HCP
2	39	HCP	HCP
3	75	HCP	HCP
4	28	FCC	HCP
5	67	HCP	HCP
6	79	FCC	FCC
7	21	HCP	HCP
8	45	FCC	HCP
9	74	BCC	HCP
10	64	HCP	HCP
11	65	HCP	HCP
12	72	HCP	HCP
13	70	FCC	BCC
14	55	BCC	BCC
15	30	HCP	HCP
16	56	BCC	BCC
17	25	BCC	BCC
18	26	BCC	HCP
19	42	BCC	HCP
20	11	BCC	BCC
21	71	HCP	HCP
22	90	FCC	HCP
23	29	FCC	HCP
24	3	BCC	BCC
25	81	HCP	HCP
26	23	BCC	HCP
27	37	BCC	BCC
28	40	HCP	HCP
29	24	BCC	BCC

30	41	BCC	HCP
31	47	FCC	FCC
32	4	HCP	HCP
33	44	HCP	HCP
34	13	FCC	HCP
35	22	HCP	HCP
36	82	FCC	FCC
37	20	BCC	BCC
38	73	BCC	HCP
39	66	HCP	HCP
40	48	HCP	FCC
41	68	HCP	HCP
42	46	FCC	FCC
43	63	BCC	BCC
44	77	FCC	HCP
45	12	HCP	BCC
46	78	FCC	HCP

Next steps: Generate code with `plot_df` View recommended plots

```
crystal_structures = ["FCC", "BCC", "HCP"]
FCC_prediction = []
BCC_prediction = []
HCP_prediction = []

for item in range(len(all_predictions)):
    FCC_prediction.append(all_predictions[item].tolist()[0])
    BCC_prediction.append(all_predictions[item].tolist()[1])
    HCP_prediction.append(all_predictions[item].tolist()[2])

# ------------------------------------------------------------

# This block will be used to sort the elements by their atomic number

atomic_number = list(df.iloc[:, 0]) # From the Pandas Dataset
order = np.argsort(atomic_number) # Sorting Indexes

# Sorting the lists by the indexes
# elements = [elements[x] for x in order]
# FCC_prediction = [FCC_prediction[x] for x in order]
# BCC_prediction = [BCC_prediction[x] for x in order]
# HCP_prediction =[HCP_prediction[x] for x in order]

# # ------------------------------------------------------------
```