2021-12 ELM Workshop

December 16, 2021

1 Welcome!

1.1 to Extreme Learning Machine Workshop

by: *Anton Akusok*

Let's go through a story of ELM in Python:
from basics to the cutting edge applications.

1.2 The Juicy Bits:

1. ELM tools and basics
2. Classification
3. Advanced Tuning

~ Break

5. High-Performance ELM
6. Driving an ELM with manual gearbox

~ Summary

2 ELM Code Tools

ELM relies on basic linear algebra. Easy to implement in Python.

- Numpy
- Scikit-Learn
- Scikit-ELM (`skelm`)
- High Performance ELM (`hpelm`)

Which tools do you use? - Matlab - Python + Numpy - Other language or math library - Scikit-Learn - hpelm - skelm - other ELM library

[1]:
```
# Example problem
from sklearn.datasets import load_diabetes

X, y = load_diabetes(return_X_y=True)
n, d = X.shape  # number of data samples and features

X.shape, y.shape
```

```
[1]: ((442, 10), (442,))
```

```
[2]: X[:3]
```

```
[2]: array([[ 0.03807591,  0.05068012,  0.06169621,  0.02187235, -0.0442235 ,
            -0.03482076, -0.04340085, -0.00259226,  0.01990842, -0.01764613],
           [-0.00188202, -0.04464164, -0.05147406, -0.02632783, -0.00844872,
            -0.01916334,  0.07441156, -0.03949338, -0.06832974, -0.09220405],
           [ 0.08529891,  0.05068012,  0.04445121, -0.00567061, -0.04559945,
            -0.03419447, -0.03235593, -0.00259226,  0.00286377, -0.02593034]])
```

```
[3]: y[:3]
```

```
[3]: array([151.,  75., 141.])
```

2.0.1 ELM implementation with different tools

Let's make a basic ELM model in different ways in Python

2.0.2 Numpy

`y = a*x` linear model
`y = B*h` in ELM notations

```
[4]: import numpy as np

     k = 10   # number of hidden neurons
     d = X.shape[1]   # number of data features
     W = np.random.randn(d, k)

     H = np.tanh(X@W)
     B = np.linalg.pinv(H)@y

     y_pred = np.tanh(X@W) @ B
     rmse = np.mean((y_pred - y)**2)**0.5
```

```
[5]: print("RMSE: {:.2f}".format(rmse))
```

```
RMSE: 161.60
```

2.0.3 Numpy with bias

```
y = B*h + bias
<==> y = B*h + bias*1
<==> y = [B bias] * [h 1]
```

```
[6]: H = np.tanh(X@W)
     H_bias = np.hstack([H, np.ones((n, 1))])
     B_bias = np.linalg.pinv(H_bias)@y
```

```
B, bias = B_bias[:-1], B_bias[-1]

y_pred = np.tanh(X@W) @ B + bias
rmse = np.mean((y_pred - y)**2)**0.5
```

[7]:
```
print("RMSE with bias: {:.2f}".format(rmse))
```

```
RMSE with bias: 53.38
```

2.0.4 Scikit-Learn

[8]:
```
from sklearn.random_projection import GaussianRandomProjection
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.base import BaseEstimator, TransformerMixin


class TanhTranformer(BaseEstimator, TransformerMixin):

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        return np.tanh(X)


model = make_pipeline(
    GaussianRandomProjection(n_components=k),
    TanhTranformer(),
    LinearRegression(fit_intercept=True)
)
```

[9]:
```
y_pred = model.fit(X, y[:, None]).predict(X)
rmse = np.mean((y_pred - y)**2)**0.5
```

[10]:
```
print("RMSE Scikit-Learn: {:.2f}".format(rmse))
```

```
RMSE Scikit-Learn: 94.87
```

2.0.5 Scikit-ELM

[11]:
```
!pip install scikit-elm
```

```
Requirement already satisfied: scikit-elm in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-packages (0.21a0)
Requirement already satisfied: scikit-learn in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-packages (from
scikit-elm) (0.24.2)
```

```
Requirement already satisfied: scipy in /Users/akusok/miniconda3/envs/scikit-
elm/lib/python3.9/site-packages (from scikit-elm) (1.6.2)
Requirement already satisfied: numpy in /Users/akusok/miniconda3/envs/scikit-
elm/lib/python3.9/site-packages (from scikit-elm) (1.20.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-packages (from
scikit-learn->scikit-elm) (2.1.0)
Requirement already satisfied: joblib>=0.11 in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-packages (from
scikit-learn->scikit-elm) (1.0.1)
```

[12]:
```
import skelm
from skelm import ELMRegressor
```

[13]:
```
model = ELMRegressor(n_neurons=k, ufunc='tanh')

y_pred = model.fit(X, y).predict(X)
rmse = np.mean((y_pred - y)**2)**0.5
```

[14]:
```
print("RMSE scikit-elm: {:.2f}".format(rmse))
```

```
RMSE scikit-elm: 53.47
```

2.0.6 HPELM

[15]:
```
!pip install hpelm
```

```
Requirement already satisfied: hpelm in /Users/akusok/Documents/hpelm (1.0.9)
Requirement already satisfied: numpy in /Users/akusok/miniconda3/envs/scikit-
elm/lib/python3.9/site-packages (from hpelm) (1.20.2)
Requirement already satisfied: scipy>=0.12 in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-packages (from
hpelm) (1.6.2)
Requirement already satisfied: tables in /Users/akusok/miniconda3/envs/scikit-
elm/lib/python3.9/site-packages (from hpelm) (3.6.1)
Requirement already satisfied: fasteners in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-
packages/fasteners-0.16.3-py3.9.egg (from hpelm) (0.16.3)
Requirement already satisfied: six in /Users/akusok/miniconda3/envs/scikit-
elm/lib/python3.9/site-packages (from hpelm) (1.16.0)
Requirement already satisfied: nose in /Users/akusok/miniconda3/envs/scikit-
elm/lib/python3.9/site-packages (from hpelm) (1.3.7)
Requirement already satisfied: numexpr>=2.6.2 in
/Users/akusok/miniconda3/envs/scikit-elm/lib/python3.9/site-packages (from
tables->hpelm) (2.7.3)
```

[16]:
```
from hpelm import HPELM
```

[17]:
```
model = HPELM(inputs=d, outputs=1)
model.add_neurons(k, 'tanh')

model.train(X, y)
y_pred = model.predict(X)
rmse = np.mean((y_pred - y)**2)**0.5
```

[18]:
```
print("RMSE hpelm: {:.2f}".format(rmse))
```

```
RMSE hpelm: 94.58
```

3 Math shared by all implementations

All implementations have the fundamental bits: - input matrix `X` of size [num_samples, num_features] - output matrix `y` of size [num_samples, num_outputs] - projection matrix `W` of size [num_features, num_neurons] - non-linear tranformation function `ufunc` like hyperbolic tangent (`np.tanh`), sigmoid, or any other - output weights `B` of size [num_neurons, num_outputs] - optional output bias/intercept of size [num_outputs,]

ELM formula: - Projection step `H = ufunc(X @ W)` - Solution step `B = solve(H, y)` - Prediction step `y_pred = ufunc(X_pred @ W) @ B`

Pair-wise ELM: - `H = ufunc(pairwise(X, W))` - `y_pred = ufunc(pairwise(X_pred, W)) @ B`

4 Improving the solution: Bias

Mathematical formula `Y = A*X` is an affine tranformation, a mix of scaling, rotation, and shear. It preserves the "center of mass" of the points. It cannot solve translation `A*X =/= X+1`.

In ELM problem, an average of `H` is around zero because of non-linear function. If an average of `y` is around zero too, an ELM works great.
But if an average of `y` is far from zero, ELM may struggle to find a solution.

Bias term learn the translation, and solves the problem of different averages of `H` and `y`.

5 Improving the solution: L2-regularisation

Models can overfit and underfit, depending on ratio between model complexity and data complexity. Both reduce predictive performance.

ELM complexity is changed by the number of neurons. We can find an optimal number of neurons.

Linear models have also numerical problems. Output weights `B` can have large magnitude numbers or small magnitude numbers.
Both produce same solution. They look like:
- `y = 3`
- `x = [1, 1, 1]` - `y = 1*x[0] - 1*x[1] + 3*x[2]` - small magnitude `B`, correct answer
- `y = 1,000,000*x[0] - 1,000,000*x[1] + 3*x[2]` - large magnitude `B`, correct answer

Large magnitude of `B` values make ELM unstable: small changes in input features completely destroy predictions.

L2-regularization term reduces the magnitude of `B` *AND* reduces model complexity.

An alternative approach to ELM complexity balance: - Increase complexity by taking more neurons than needed - Decrease complexity *and* magnitude of `B` values by increasing L2-regularisation term

Get an optimal complexity model, with good stability from small magnitude `B` values.

How to implement L2-regularization?

- Numpy approach:
  - very simple code trick; on next slides
- Scipy approach:
  - replace `LinearRegression` with `Ridge` that is a linear regression with L2-parameter `alpha`
- Scikit-ELM, HPELM:
  - use `alpha` parameter

[19]:
```python
# Numpy without L2-regularization

B = np.linalg.pinv(H) @ y
B
```

[19]:
```
array([  594.23603143,  -347.19803279,    85.03919374,  -495.24567633,
       -1093.29018554,   585.25115157,   441.26710671,   177.18285223,
         106.3216512 ,   598.75718248])
```

[20]:
```python
# Numpy with L2-regularization
# HB = y  <==>  H'HB = H'y

HH = H.T@H
Hy = H.T@y[:, None]

alpha = 1000
HH_with_L2 = HH + alpha*np.eye(k)  # add `alpha` to diagonal of H'H

B = np.linalg.pinv(HH_with_L2) @ Hy
B.ravel()
```

[20]:
```
array([-1.04695474,  1.51977718,  0.41858134, -3.97798104, -1.78069866,
       -1.94954697,  0.61296103,  2.59370597,  0.29933982, -1.92710606])
```

[21]:
```python
# in-place diagonal increase
HH_L2 = HH.copy()
HH_L2.ravel()[::k+1] += alpha

B = np.linalg.pinv(HH_L2) @ Hy
B.ravel()
```

```
[21]: array([-1.04695474,  1.51977718,  0.41858134, -3.97798104, -1.78069866,
              -1.94954697,  0.61296103,  2.59370597,  0.29933982, -1.92710606])
```

6 Classification

Let's look at basics of ELM classification.

We actually saw regression in the previous chapter.

```
[22]: # Very similar problem - what changed?
      from sklearn.datasets import load_iris

      X, y = load_iris(return_X_y=True)
      n, d = X.shape  # number of data samples and features

      X.shape, y.shape
```

```
[22]: ((150, 4), (150,))
```

```
[23]: X[:3]
```

```
[23]: array([[5.1, 3.5, 1.4, 0.2],
             [4.9, 3. , 1.4, 0.2],
             [4.7, 3.2, 1.3, 0.2]])
```

```
[24]: y
```

```
[24]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
              1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
              1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
              2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
              2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Convert classes into multiple binary outputs

```
[25]: y_2d = y[:, None]
      Y = np.hstack([y_2d==i for i in range(3)]).astype(int)
      Y
```

```
[25]: array([[1, 0, 0],
             [1, 0, 0],
             [1, 0, 0],
             [1, 0, 0],
             [1, 0, 0],
             [1, 0, 0],
             [1, 0, 0],
             [1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[1, 0, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
[0, 0, 1],
       [0, 0, 1]])
```

6.0.1 Multi-output ELM for classification

Multi-output ELM is multiple ELMs with a shared hidden layer, each having an independent output solution.

[26]:
```
k = 10  # number of hidden neurons
d = X.shape[1]  # number of data features
W = np.random.randn(d, k)

H = np.tanh(X@W)
H_bias = np.hstack([H, np.ones((n, 1))])
B_bias = np.linalg.pinv(H_bias)@Y

B, bias = B_bias[:-1], B_bias[-1]

Y_raw = np.tanh(X@W) @ B + bias
y_pred = Y_raw.argmax(axis=1)

accuracy = np.mean(y_pred == y)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 94.67%
```

[27]:
```
# Each output feature is an independent problem

B2_bias_parts = [np.linalg.pinv(H_bias)@Y[:, i:i+1] for i in range(3)]
B2_bias = np.hstack(B2_bias_parts)

B, bias = B2_bias[:-1], B2_bias[-1]

Y_raw = np.tanh(X@W) @ B + bias
y_pred = Y_raw.argmax(1)

accuracy = np.mean(y_pred == y)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 94.67%
```

[28]:
```
from sklearn.linear_model import Ridge

model = make_pipeline(
    GaussianRandomProjection(n_components=k),
    TanhTranformer(),
    Ridge(fit_intercept=True, alpha=10)
)

Y_raw = model.fit(X, Y).predict(X)
```

```
y_pred = Y_raw.argmax(1)

accuracy = np.mean(y_pred == y)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 82.00%
```

6.0.2 Train/test split

So far we looked at training error - a measure of how well a model can extract information from known data.

Prediction problems needs to know how well a model predicts information for new coming, previously unseen data.

Test set helps approximating the prediction performance.

[29]:
```
from sklearn.model_selection import train_test_split

Xt, Xs, yt, ys = train_test_split(X, y, test_size=0.3)

ys
```

[29]:
```
array([0, 2, 1, 0, 2, 2, 1, 0, 0, 1, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 1, 0,
       2, 2, 0, 2, 1, 1, 2, 1, 0, 0, 0, 0, 1, 1, 0, 0, 2, 0, 2, 2, 0, 2,
       1])
```

[30]:
```
Yt = np.hstack([yt[:, None]==i for i in range(3)]).astype(int)

Y_raw = model.fit(Xt, Yt).predict(Xs)
y_pred = Y_raw.argmax(1)

accuracy = np.mean(y_pred == ys)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 86.67%
```

Beware of cutting data in two - both training and test sets must get a similar mix of data

[31]:
```
Xt, Xs = X[:100], X[100:]
yt, ys = y[:100], y[100:]
```

[32]:
```
yt
```

[32]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

[33]:
```
ys
```

[33]:
```
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2])
```

[34]:
```
Yt = np.hstack([yt[:, None]==i for i in range(3)]).astype(int)

Y_raw = model.fit(Xt, Yt).predict(Xs)
y_pred = Y_raw.argmax(1)

accuracy = np.mean(y_pred == ys)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 0.00%
```

Correct split by randomly shuffle data before it

[35]:
```
index = np.arange(n)
np.random.shuffle(index)

X_shuffle = X[index]  # SAME index for X and y
y_shuffle = y[index]

Xt, Xs = X_shuffle[:100], X_shuffle[100:]
yt, ys = y_shuffle[:100], y_shuffle[100:]
```

[36]:
```
yt
```

[36]:
```
array([2, 2, 1, 0, 2, 0, 1, 2, 0, 2, 0, 1, 0, 0, 2, 1, 1, 0, 2, 0, 1, 1,
       0, 0, 2, 2, 2, 0, 1, 1, 0, 2, 1, 1, 0, 2, 2, 0, 2, 1, 1, 2, 1, 2,
       1, 0, 2, 1, 0, 2, 0, 2, 1, 2, 0, 0, 1, 1, 2, 0, 0, 2, 2, 1, 0, 2,
       1, 0, 1, 1, 2, 0, 0, 0, 1, 0, 2, 2, 2, 2, 2, 1, 2, 2, 0, 0, 0, 2,
       0, 0, 1, 2, 0, 0, 2, 2, 2, 0, 0, 1])
```

[37]:
```
ys
```

[37]:
```
array([1, 0, 1, 1, 0, 1, 1, 0, 2, 2, 2, 2, 2, 1, 1, 2, 0, 1, 2, 1, 1, 0,
       0, 1, 1, 1, 1, 2, 0, 2, 1, 0, 0, 1, 0, 0, 1, 2, 1, 1, 0, 1, 2, 0,
       1, 2, 1, 1, 2, 0])
```

[38]:
```
Yt = np.hstack([yt[:, None]==i for i in range(3)]).astype(int)

Y_raw = model.fit(Xt, Yt).predict(Xs)
y_pred = Y_raw.argmax(1)

accuracy = np.mean(y_pred == ys)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 54.00%
```

7 Scaling

Random projection in ELM is a simple mathematical formula that includes addition.

If feature A has 100x larger magnitude than feature B, it will be 100x more important after the addition. Small magnitude features are almost ignored.

Non-linear function also requires a specific range of values. Hyperbolic tangent `tanh` is linear in [-0.1, 0.1] and crops to -1 at [-inf, -3] and +1 at [3, +inf]. Best range of `X@W` values are within [-3, -0.1] U [0.1, 3].

[39]:
```
X[:3]
```

[39]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2]])
```

Simple scaling

[40]:
```
X_norm = (X - X.mean(0)) / X.std(0)

X_norm[:3]
```

[40]:
```
array([[-0.90068117,  1.01900435, -1.34022653, -1.3154443 ],
       [-1.14301691, -0.13197948, -1.34022653, -1.3154443 ],
       [-1.38535265,  0.32841405, -1.39706395, -1.3154443 ]])
```

Automatic scaling

[41]:
```
from sklearn.preprocessing import RobustScaler

X_robust = RobustScaler().fit_transform(X)

X_robust[:3]
```

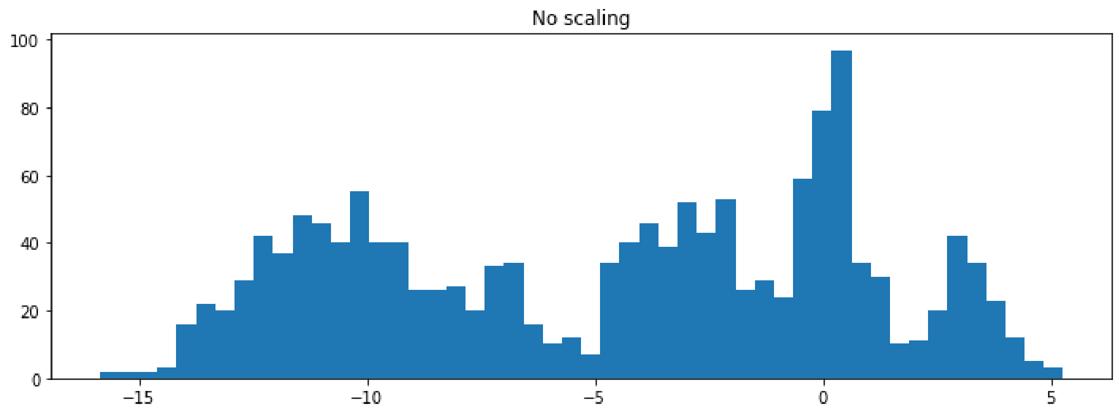
[41]:
```
array([[-0.53846154,  1.        , -0.84285714, -0.73333333],
       [-0.69230769,  0.        , -0.84285714, -0.73333333],
       [-0.84615385,  0.4       , -0.87142857, -0.73333333]])
```

Let's examine values of `H` that go into non-linear function of ELM

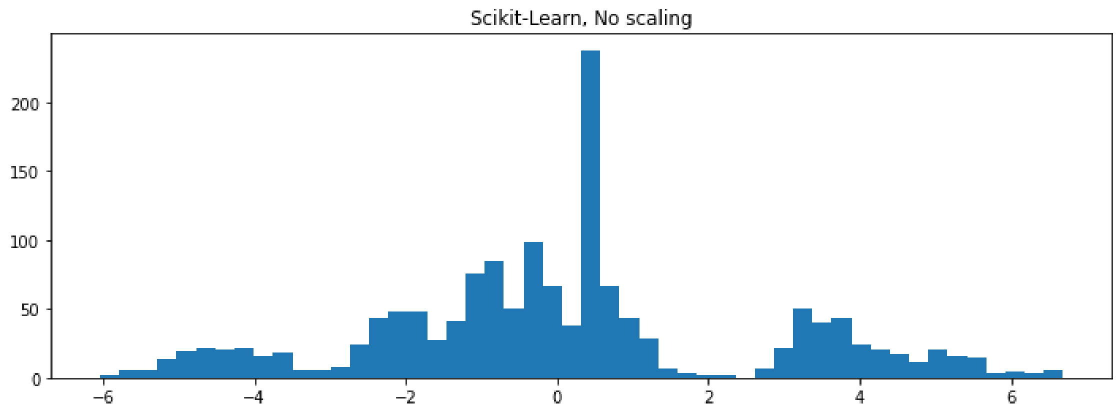
[42]:
```
from matplotlib import pyplot as plt
%matplotlib inline

plt.rcParams["figure.figsize"] = (12, 4)
```

[43]:
```
XW = X@W
```

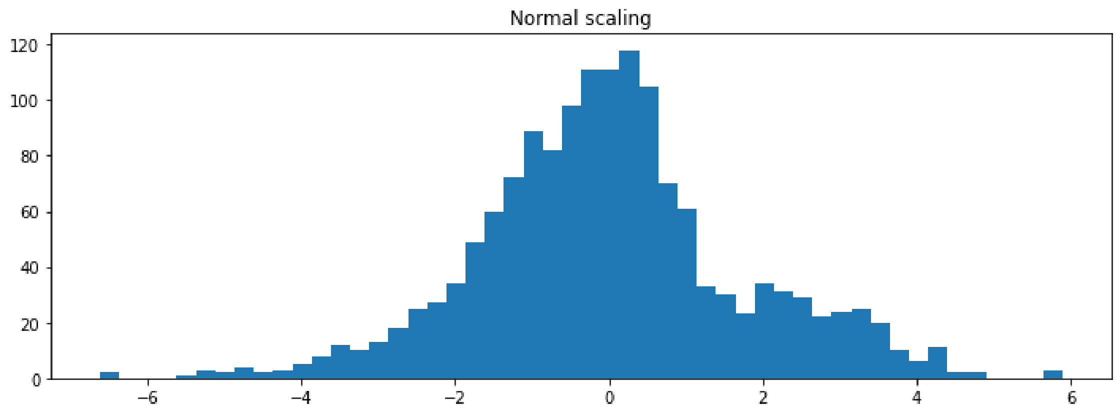
```
plt.hist(XW.ravel(), 50)
plt.title("No scaling")
plt.show()
```



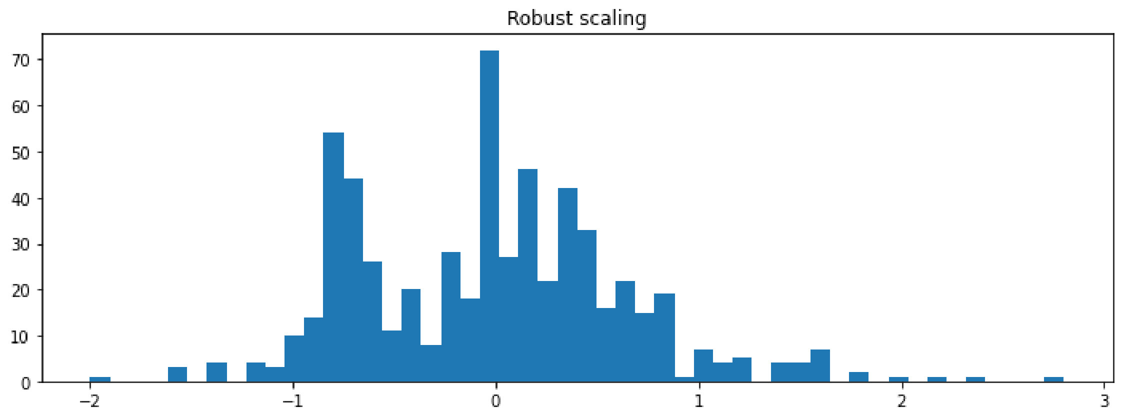
[44]:
```
XW_sklearn = GaussianRandomProjection(n_components=k).fit_transform(X)

plt.hist(XW_sklearn.ravel(), 50)
plt.title("Scikit-Learn, No scaling")
plt.show()
```



[45]:
```
X_norm = (X - X.mean(0)) / X.std(0)
XW_norm = X_norm@W

plt.hist(XW_norm.ravel(), 50)
plt.title("Normal scaling")
plt.show()
```



```
[46]: X_robust = RobustScaler().fit_transform(X)
      XW_robust = X_robust@W

      plt.hist(X_robust.ravel(), 50)
      plt.title("Robust scaling")
      plt.show()
```



Fixing scaling in Scikit-Learn

```
[47]: model = make_pipeline(
          RobustScaler(),
          skelm.ELMClassifier(n_neurons=k)
          #GaussianRandomProjection(n_components=k),
          #TanhTranformer(),
          #Ridge(fit_intercept=True, alpha=0.01)
      )

      Y_raw = model.fit(X, Y).predict(X)
```

```
y_pred = Y_raw.argmax(1)

accuracy = np.mean(y_pred == y)
print("Accuracy: {:.2f}%".format(100 * accuracy))
```

```
Accuracy: 85.33%
```

8 Advanced Tuning

We can do even better with our models.

```
[48]: import pandas as pd

      fortune1k = pd.read_csv("fortune 1000 companies in dec2021.csv").iloc[:, 3:].
       ↪replace('-', '0')
      fortune1k.head()
```

```
[48]:     Revenue revenue(% change) profits in millions profits % change    assets   \
     0  $559,151             0.067             $13,510           -0.092  $252,496
     1  $386,064             0.376             $21,331            0.841  $321,195
     2  $274,515             0.055             $57,411            0.039  $323,888
     3  $268,706             0.046              $7,179            0.082  $230,715
     4  $257,141             0.062             $15,403            0.113  $197,289

        market value change in rank in 1000  employees change in rank(500 only)  \
     0    $382,642.8                      0    2300000                        0
     1  $1,558,069.6                      0    1298000                        0
     2  $2,050,665.9                      1     147000                        1
     3     $98,653.2                      1     256500                        1
     4      $351,725                      2     330000                        2

       measure_up_rank
     0              20
     1              11
     2             188
     3              57
     4              25
```

```
[49]: for col in fortune1k.columns:
          if fortune1k[col].dtype == int:
              continue
          fortune1k[col] = fortune1k[col].str.replace("[$,]", "", regex=True).
       ↪astype(float)
```

```
[50]: fortune1k.head()
```

```
[50]:     Revenue  revenue(% change)  profits in millions  profits % change  \
      0  559151.0              0.067              13510.0            -0.092
      1  386064.0              0.376              21331.0             0.841
      2  274515.0              0.055              57411.0             0.039
      3  268706.0              0.046               7179.0             0.082
      4  257141.0              0.062              15403.0             0.113

           assets  market value  change in rank in 1000  employees  \
      0  252496.0      382642.8                     0.0    2300000
      1  321195.0     1558069.6                     0.0    1298000
      2  323888.0     2050665.9                     1.0     147000
      3  230715.0       98653.2                     1.0     256500
      4  197289.0      351725.0                     2.0     330000

         change in rank(500 only)  measure_up_rank
      0                       0.0             20.0
      1                       0.0             11.0
      2                       1.0            188.0
      3                       1.0             57.0
      4                       2.0             25.0
```

9 Log Scale: the missing mean

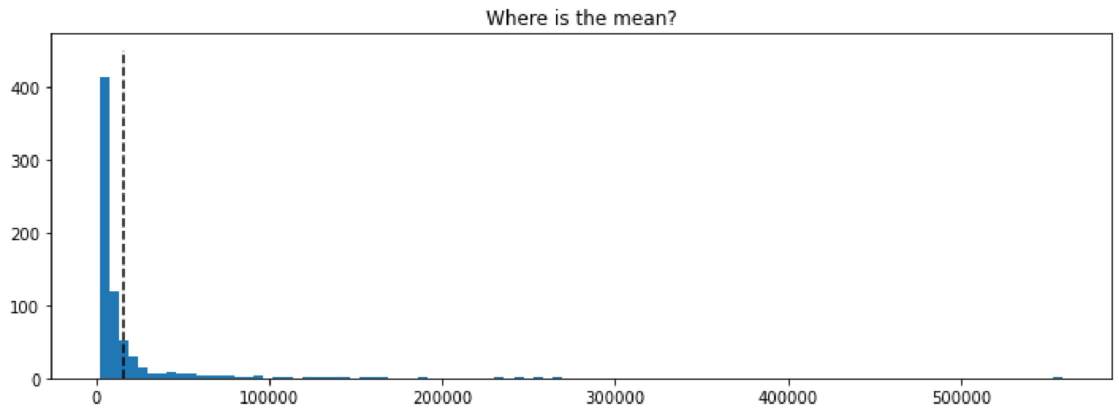
```
[51]: from sklearn.metrics import r2_score
```

```
[52]: X, y = fortune1k.iloc[:, 1:], fortune1k.iloc[:,0]
      Xt, Xs, yt, ys = train_test_split(X, y, train_size=0.3)

      model = skelm.ELMRegressor().fit(Xt, yt)
      ys_pred = model.predict(Xs)

      score = r2_score(y_true=ys, y_pred=ys_pred)
      print("R2 score:", score)
```

```
R2 score: -65.05610344828138
```

9.0.1 Let's examine the mean values

```
[53]: plt.hist(ys, 100)
      m = np.mean(ys)
      plt.plot([m, m], [0, 450], '--k')
      plt.title("Where is the mean?")
      plt.show()
```



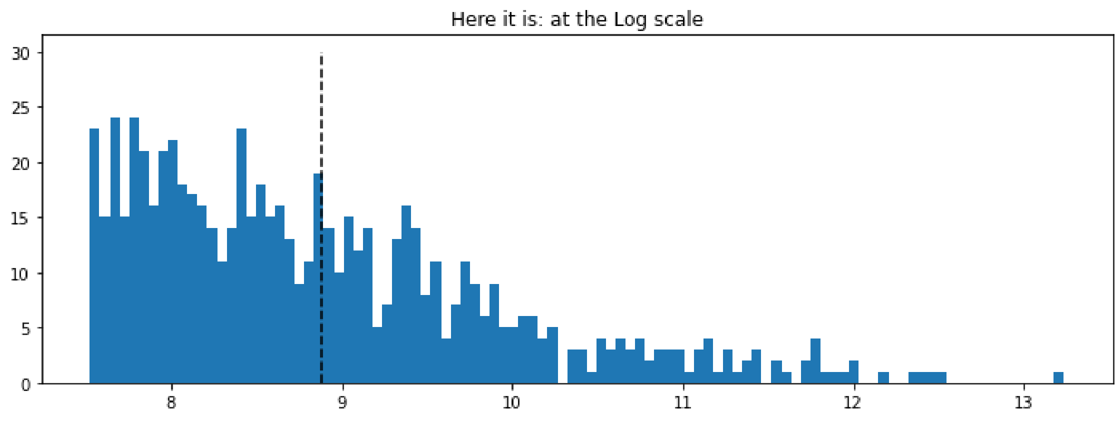
Logarithmically distributed variables have no concept of `mean`.

Very common in real world - income, followers, noise and light levels, etc.

Taking `log(value)` fixes the missing mean problem.

(This dataset is "top-1000" companies. It looks like a tail of normal distribution because it is a tail of normal distribution.)

[54]:
```
plt.hist(np.log(ys), 100)
m = np.mean(np.log(ys))
plt.plot([m, m], [0, 30], '--k')
plt.title("Here it is: at the Log scale")
plt.show()
```



9.0.2 Scale input features

```
[55]: X, y = fortune1k.iloc[:, 1:], fortune1k.iloc[:,0]
      for col in ['profits in millions', 'assets', 'market value',
                  'employees', 'measure_up_rank']:
          val = X[col]
          X[col] = np.sign(val) * np.log(np.abs(val) + 1)  # "negative" log values

      Xt, Xs, yt, ys = train_test_split(X, y, train_size=0.3)

      model = skelm.ELMRegressor().fit(Xt, yt)
      ys_pred = model.predict(Xs)

      score = r2_score(y_true=ys, y_pred=ys_pred)
      print("R2 score:", score)
```

```
R2 score: -0.483488160515708
```

Did not really help...

9.0.3 Scale *output* features too!

```
[56]: fortune1k_log = fortune1k.copy()
      for col in ['Revenue', 'profits in millions', 'assets', 'market value',
                  'employees', 'measure_up_rank']:
          val = fortune1k_log[col]
          fortune1k_log[col] = np.sign(val) * np.log(np.abs(val) + 1)
```

```
[57]: X, y = fortune1k_log.iloc[:, 1:], fortune1k_log.iloc[:,0]
      Xt, Xs, yt, ys = train_test_split(X, y, train_size=0.3)

      model = skelm.ELMRegressor(n_neurons=20).fit(Xt, yt)
      ys_pred = model.predict(Xs)

      score = r2_score(y_true=ys, y_pred=ys_pred)
      print("R2 score:", score)
```

```
R2 score: 0.3984951169667912
```

It worked!

ELM output is a linear model.

Linear model is a mapping between two values: `HB = Y`.
It is fully reversible: `H = YB^-1`.

If *either* of the values `H`, `Y` is poorly scaled, the model does not work.
Scale *both* inputs and outputs.

10 ~ Break | Questions?

11 High-Performance ELM: hpelm

An older interface model that does not play well with Scikit-Learn.

Insane power, GPU, low compute overhead.

[58]:
```
from sklearn.datasets import fetch_california_housing

X, y = fetch_california_housing(return_X_y=True)

X.shape, y.shape
```

[58]:
```
((20640, 8), (20640,))
```

[59]:
```
n, d = X.shape
d_out = 1
k = 1000
```

[60]:
```
model = HPELM(d, d_out, precision='single', norm=1e-5)

model.add_neurons(number=k//2, func="tanh")
model.add_neurons(number=k//2, func="rbf_l2")
```

[61]:
```
model.train(X, y)
```

```
Covariance matrix is not full rank; solving with SVD (slow)
This happened because you have duplicated or too many neurons
```

```
/Users/akusok/Documents/hpelm/hpelm/nnets/slfn_python.py:65: FutureWarning:
`rcond` parameter will change to the default of machine precision times ``max(M,
N)`` where M and N are the input matrix dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
  B = np.linalg.lstsq(HH, HT)[0]
```

[62]:
```
yh = model.predict(X)

r2_score(y, yh)
```

[62]:
```
0.14323439672290872
```

11.0.1 Sensible to data normalization

[63]:
```
X = (X - X.mean(0)) / X.std(0)
y = y - y.mean()  # no bias at the moment
```

[64]:
```
model.train(X, y)
```

```
Covariance matrix is not full rank; solving with SVD (slow)
This happened because you have duplicated or too many neurons
```

```
/Users/akusok/Documents/hpelm/hpelm/nnets/slfn_python.py:65: FutureWarning:
`rcond` parameter will change to the default of machine precision times ``max(M,
N)`` where M and N are the input matrix dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
  B = np.linalg.lstsq(HH, HT)[0]
```

[65]:
```
yh = model.predict(X)

r2_score(y, yh)
```

[65]:
```
-0.6618510799351098
```

11.0.2 HP-ELM Classification

[66]:
```
from sklearn.datasets import fetch_covtype

X, y = fetch_covtype(return_X_y=True)
```

[67]:
```
X.shape, y.shape
```

[67]:
```
((581012, 54), (581012,))
```

[68]:
```
set(y)
```

[68]:
```
{1, 2, 3, 4, 5, 6, 7}
```

Transform outputs into one-hot encoding

[69]:
```
y_2d = y[:, None]
Y = np.hstack([y_2d==i for i in (1,2,3,4,5,6,7)]).astype(int)
Y.shape
```

[69]:
```
(581012, 7)
```

[70]:
```
n, d = X.shape
d_out = Y.shape[1]
k = 200
```

Classification types: - 'c' for classification - 'wc' for weighted classification - 'ml' for multi-label classification

[71]:
```
model = HPELM(d, d_out, classification='c', norm=1e-5)

model.add_neurons(number=k, func="tanh")
```

[72]:
```
model.train(X, Y)
```

```
[73]: from sklearn.metrics import f1_score

      Yh = model.predict(X)
      yh = Yh.argmax(1) + 1
```

```
[74]: f1_score(y, yh, average='micro')
```

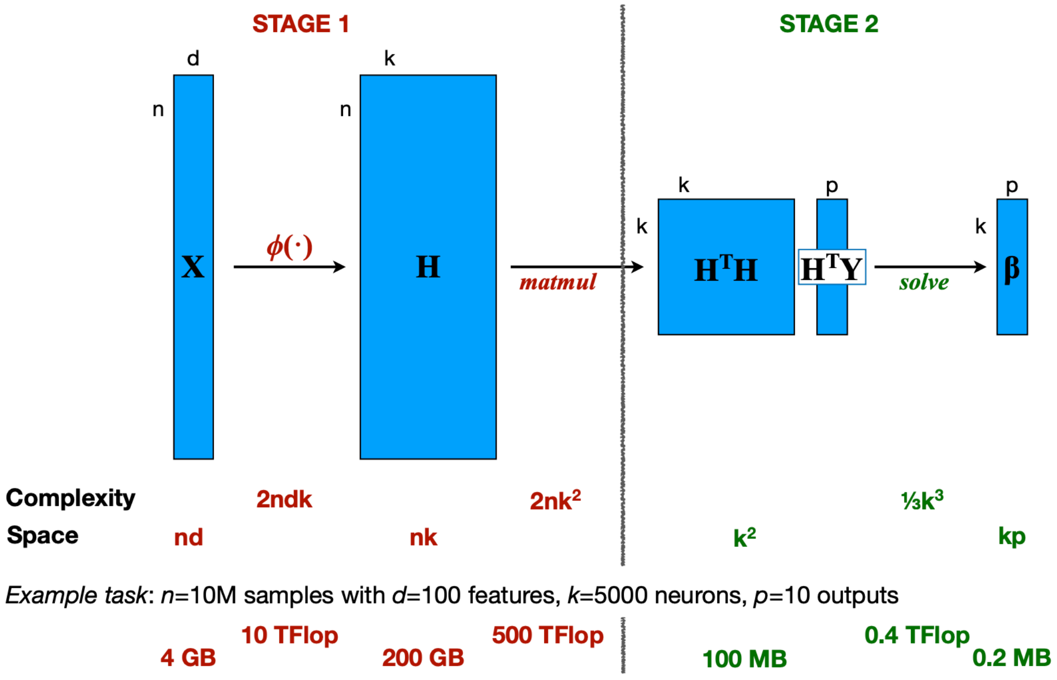
```
[74]: 0.543047303670148
```

12 ELM in manual gearbox

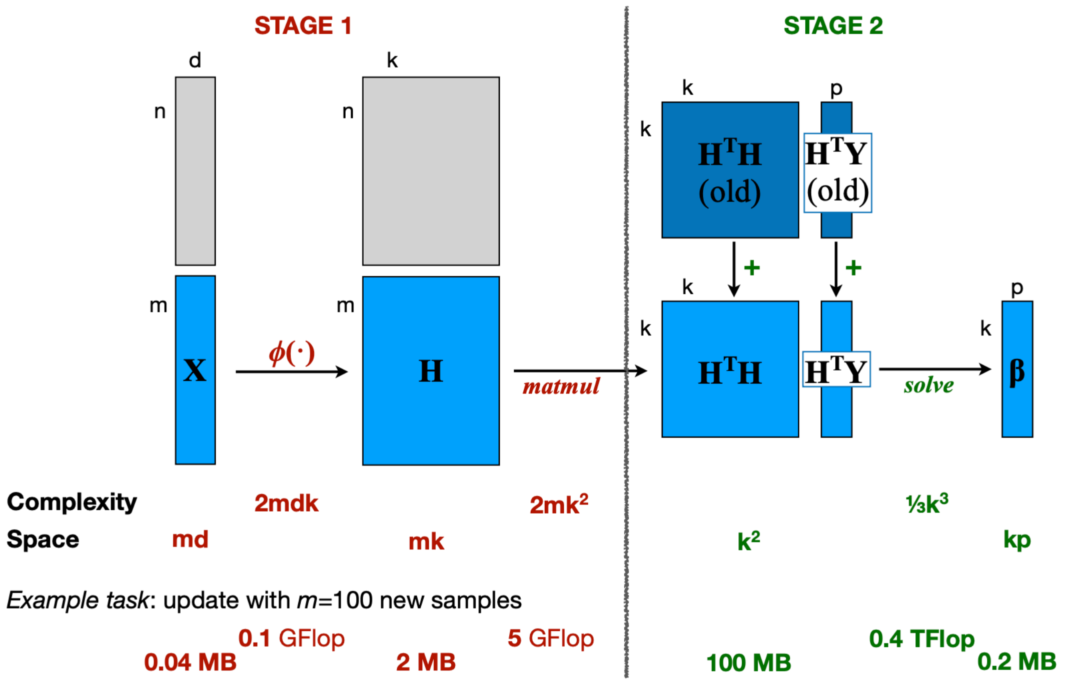
12.0.1 Two-stage solution with H'H matrix

A trick to large-scale ELM is to remove data sample size `n` variable from the calculations.

Then we don't care how much data there is.



This trick allows us to update previously trained model!



```
[75]: model = HPELM(d, d_out, classification='c', norm=1e-5)

      model.add_neurons(number=k, func="tanh")
```

```
[76]: model.add_data(X[0::3], Y[0::3])
      model.nnet.HH[:3, :3]
```

```
[76]: array([[193427.4071176 ,  30353.55352168, -12373.00075164],
             [     0.        , 193487.67781409,  -3321.91147406],
             [     0.        ,      0.        , 193470.68255472]])
```

```
[77]: model.add_data(X[1::3], Y[1::3])
      model.add_data(X[2::3], Y[2::3])
      model.nnet.HH[:3, :3]
```

```
[77]: array([[580258.63412699,  91799.98327675, -37096.26449528],
             [     0.        , 580515.87596593, -10206.85281667],
             [     0.        ,      0.        , 580423.04466254]])
```

```
[78]: model.nnet.solve()
```

```
[79]: Yh = model.predict(X)
      yh = Yh.argmax(1) + 1

      f1_score(y, yh, average='micro')
```

```
[79]: 0.5421867362464114
```

12.0.2 HH, HY matrices in scikit-elm

[80]:
```
model = skelm.ELMClassifier(n_neurons=k, ufunc='tanh')

model.partial_fit(X[0::3], Y[0::3], compute_output_weights=False)
```

[80]:
```
ELMClassifier(n_neurons=200)
```

[81]:
```
model.hidden_layers_
```

[81]:
```
(HiddenLayer(n_neurons=200),)
```

[82]:
```
layer = model.hidden_layers_[0]

layer.transform, layer.ufunc_
```

[82]:
```
(<bound method HiddenLayer.transform of HiddenLayer(n_neurons=200)>,
 <ufunc 'tanh'>)
```

[83]:
```
H = layer.ufunc_(layer.transform(X[0::3]))
HH = H.T @ H
HY = H.T @ Y[0::3]
```

[84]:
```
HH[:3, :3]
```

[84]:
```
array([[110900.63208837,  25750.44242839, -16304.4179474 ],
       [ 25750.44242839, 112310.60792288, -48153.17632853],
       [-16304.4179474 , -48153.17632853, 111589.56669432]])
```

[85]:
```
HY[:3]
```

[85]:
```
array([[ 13776.31683254,   9829.7808262 ,   4599.13926558,
           480.5205682 ,    511.35032502,   1738.40172467,
          1606.7939097 ],
       [ 53594.48847965,  70914.10413657,   9053.831326  ,
           712.09053582,   2301.28370537,   4418.76929286,
          5235.19822804],
       [-22074.80939298, -22607.92147265,  -8830.15815199,
          -712.09053553,  -1902.11720931,  -4310.14209332,
         -1574.08740123]])
```

12.0.3 Difference in solvers: fast & failing solvers, finite iterative solvers

ELM output layer needs a least squares solution of a linear system.

Basic equation `B = pinv(H) @ y` is computationally inefficient, and has numerical problems.

Solvers can compute `B` from `H, y` without inverting matrix `H` directly.

[86]:
```
k = 500
model = skelm.ELMClassifier(n_neurons=k, ufunc='tanh')

model.partial_fit(X[:1000], Y[:1000], compute_output_weights=False)

layer = model.hidden_layers_[0]
```

[87]:
```
H = layer.ufunc_(layer.transform(X[:1000]))
HH = H.T @ H
HY = H.T @ Y[:1000]
```

[88]:
```
import scipy

%time B = scipy.linalg.solve(HH, HY)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<timed exec> in <module>

~/miniconda3/envs/scikit-elm/lib/python3.9/site-packages/scipy/linalg/basic.py␣
 ↪in solve(a, b, sym_pos, lower, overwrite_a, overwrite_b, debug, check_finite,␣
 ↪assume_a, transposed)
    212                                           (a1, b1))
    213         lu, ipvt, info = getrf(a1, overwrite_a=overwrite_a)
--> 214         _solve_check(n, info)
    215         x, info = getrs(lu, ipvt, b1,
    216                         trans=trans, overwrite_b=overwrite_b)

~/miniconda3/envs/scikit-elm/lib/python3.9/site-packages/scipy/linalg/basic.py␣
 ↪in _solve_check(n, info, lamch, rcond)
     27                          '.'.format(-info))
     28     elif 0 < info:
---> 29         raise LinAlgError('Matrix is singular.')
     30 
     31     if lamch is None:

LinAlgError: Matrix is singular.
```

Solve numerical problems by adding L2-regularization

[89]:
```
%time B = scipy.linalg.solve(HH + 10000*np.eye(k), HY)
```

```
CPU times: user 21.8 ms, sys: 6.21 ms, total: 28 ms
Wall time: 17.5 ms
```

Other solvers can handle numerical instability

[90]:
```
%time B = np.linalg.lstsq(HH, HY, rcond=-1)[0]
```

```
CPU times: user 189 ms, sys: 21.1 ms, total: 210 ms
Wall time: 86 ms
```

L2-regularisation helps with failing solution

```
[91]: %time B = scipy.linalg.cho_solve(scipy.linalg.cho_factor(HH), HY)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<timed exec> in <module>

~/miniconda3/envs/scikit-elm/lib/python3.9/site-packages/scipy/linalg/
 ↪decomp_cholesky.py in cho_factor(a, lower, overwrite_a, check_finite)
    150
    151     """
--> 152     c, lower = _cholesky(a, lower=lower, overwrite_a=overwrite_a,␣
 ↪clean=False,
    153                          check_finite=check_finite)
    154     return c, lower

~/miniconda3/envs/scikit-elm/lib/python3.9/site-packages/scipy/linalg/
 ↪decomp_cholesky.py in _cholesky(a, lower, overwrite_a, clean, check_finite)
     35     c, info = potrf(a1, lower=lower, overwrite_a=overwrite_a,␣
 ↪clean=clean)
     36     if info > 0:
---> 37         raise LinAlgError("%d-th leading minor of the array is not␣
 ↪positive "
     38                           "definite" % info)
     39     if info < 0:

LinAlgError: 5-th leading minor of the array is not positive definite
```

```
[92]: %time B = scipy.linalg.cho_solve(scipy.linalg.cho_factor(HH + 1000*np.eye(k)),␣
 ↪HY)
```

```
CPU times: user 12.5 ms, sys: 2.45 ms, total: 15 ms
Wall time: 3.86 ms
```

Naive approach

```
[93]: %time B = np.linalg.pinv(HH) @ HY
```

```
CPU times: user 193 ms, sys: 15.3 ms, total: 208 ms
Wall time: 61.3 ms
```

Different solver algorithms have different time requirements, and limitations: - pseudo-inverse: 78ms - lstsq (SVD-based solver): 44ms - scipy.solve (LU-based, can fail): 6ms - scipy cholesky: 4ms

Different solvers may be faster at extremely large tasks. LSQR is a finite iterative solver that converges faster on large or poorly determined problems.

```
[94]: from scipy.sparse.linalg import lsqr
```

```
[95]: %time b = lsqr(HH, HY[:,0])[0]
```

```
CPU times: user 970 ms, sys: 48 ms, total: 1.02 s
Wall time: 316 ms
```

Why do we want such a slow solver?

When crossing >10,000 neurons and >1hour solution, it is actually faster.

Also *MUCH* faster on poorly defined problems - gets to a bad solution quickly.

13 Post-training model tuning

ELM allows us to change model parameters *after* the heaviest computations are finished.

13.0.1 Post-training L2-regularization

L2 regularization term `alpha` is a constant added to the diagonal of H'H

We can add different L2-regularizations and re-solve ELM only looking at training data once.

```
[96]: for alpha in [0.01, 1, 100]:
          B = np.linalg.lstsq(HH + alpha*np.eye(k), HY, rcond=-1)[0]

          Yh = H @ B
          yh = Yh.argmax(1) + 1

          score = f1_score(y[:1000], yh, average='micro')
          print(f"L2 alpha={alpha}, ELM training score: {score}")
```

```
L2 alpha=0.01, ELM training score: 0.8399999999999999
L2 alpha=1, ELM training score: 0.845
L2 alpha=100, ELM training score: 0.747
```

13.0.2 Post-training model downsizing

Number of neurons in ELM can be reduced after training by taking part of HH, HY matrices.

```
[97]: for k in [200, 100, 50, 10, 3]:
          B = np.linalg.lstsq(HH[:k,:k], HY[:k], rcond=-1)[0]

          Yh = H[:, :k] @ B
          yh = Yh.argmax(1) + 1

          score = f1_score(y[:1000], yh, average='micro')
          print(f"Neurons={k}, ELM training score: {score}")
```

```
Neurons=200, ELM training score: 0.756
Neurons=100, ELM training score: 0.702
Neurons=50, ELM training score: 0.661
Neurons=10, ELM training score: 0.592
Neurons=3, ELM training score: 0.585
```

13.0.3 Distributed computations + combine results

Distributed computations are done by computing parts of `HH, HY` matrices on different machines, then summing them up, and solving the model.

Must use the same model on all machines - generate one model, save it, and re-load.

[98]:
```
X1, X2, X3 = X[0::3], X[1::3], X[2::3]
Y1, Y2, Y3 = Y[0::3], Y[1::3], Y[2::3]
```

[99]:
```
model = HPELM(inputs=d, outputs=d_out)
model.add_neurons(k, 'tanh')
model.save("model.pkl")
```

[100]:
```
m1 = HPELM(inputs=d, outputs=d_out)
m1.load("model.pkl")

m2 = HPELM(inputs=d, outputs=d_out)
m2.load("model.pkl")

m3 = HPELM(inputs=d, outputs=d_out)
m3.load("model.pkl")
```

[101]:
```
import pickle
```

[102]:
```
m1.add_data(X1, Y1)
with open("data1.pkl", "wb") as f:
    pickle.dump([m1.nnet.HH, m1.nnet.HT], f)

m2.add_data(X2, Y2)
with open("data2.pkl", "wb") as f:
    pickle.dump([m2.nnet.HH, m2.nnet.HT], f)

m3.add_data(X3, Y3)
with open("data3.pkl", "wb") as f:
    pickle.dump([m3.nnet.HH, m3.nnet.HT], f)
```

[103]:
```
HH = np.zeros((k,k))
HY = np.zeros((k, d_out))

with open("data1.pkl", "rb") as f:
    HH_part, HY_part = pickle.load(f)
    HH += HH_part
    HY += HY_part

with open("data2.pkl", "rb") as f:
    HH_part, HY_part = pickle.load(f)
    HH += HH_part
    HY += HY_part

with open("data3.pkl", "rb") as f:
    HH_part, HY_part = pickle.load(f)
    HH += HH_part
    HY += HY_part
```

[104]:
```
B = np.linalg.lstsq(HH, HY, rcond=-1)[0]

Yh = model.project(X) @ B
yh = Yh.argmax(1) + 1

score = f1_score(y, yh, average='micro')
print(f"ELM training score: {score}")
```

```
ELM training score: 0.48996406270438475
```

14 Summary

ELM is a wonderful thing: it lets us get inside and tune its inner workings to whatever task we face. No other method ever allows the same freedom.

14.0.1 Key takeaway points

- ELM has simple math notation: use it!
- scaling of inputs *and* outputs is important
- use distributed computing and custom solvers to push limits of huge tasks

14.1 Thanks!

15 Questions

Presenter: Anton Akusok

— **ELM Code Tools**
ELM implementation with different tools
Math shared by all implementations
Improving the solution: Bias
Improving the solution: L2-regularisation
How to implement L2-regularization?

— **Classification**
Multi-output ELM for classification Train/test split

Scaling
Range of `H` values

— **Advanced Tuning**
Log Scale: the missing mean
Scale input features
Scale output features too!

— **High-Performance ELM: hpelm**
Sensible to data normalization
HP-ELM Classification

— **ELM in manual gear**
Two-stage solution with H'H matrix
HH, HY matrices in scikit-elm
Difference in solvers: fast & failing solvers, finite iterative solvers
Post-training model tuning
Distributed computations + combine results