Insights about ELM

Extreme Learning Machines

Agenda

1. Scikit-ELM and HPELM
2. What is a model?
3. Idea of ELM
4. Parts of ELM
5. Two-stage computation
   and the tricks we could do with it
6. ELM best practices

1. Scikit-ELM and HPELM

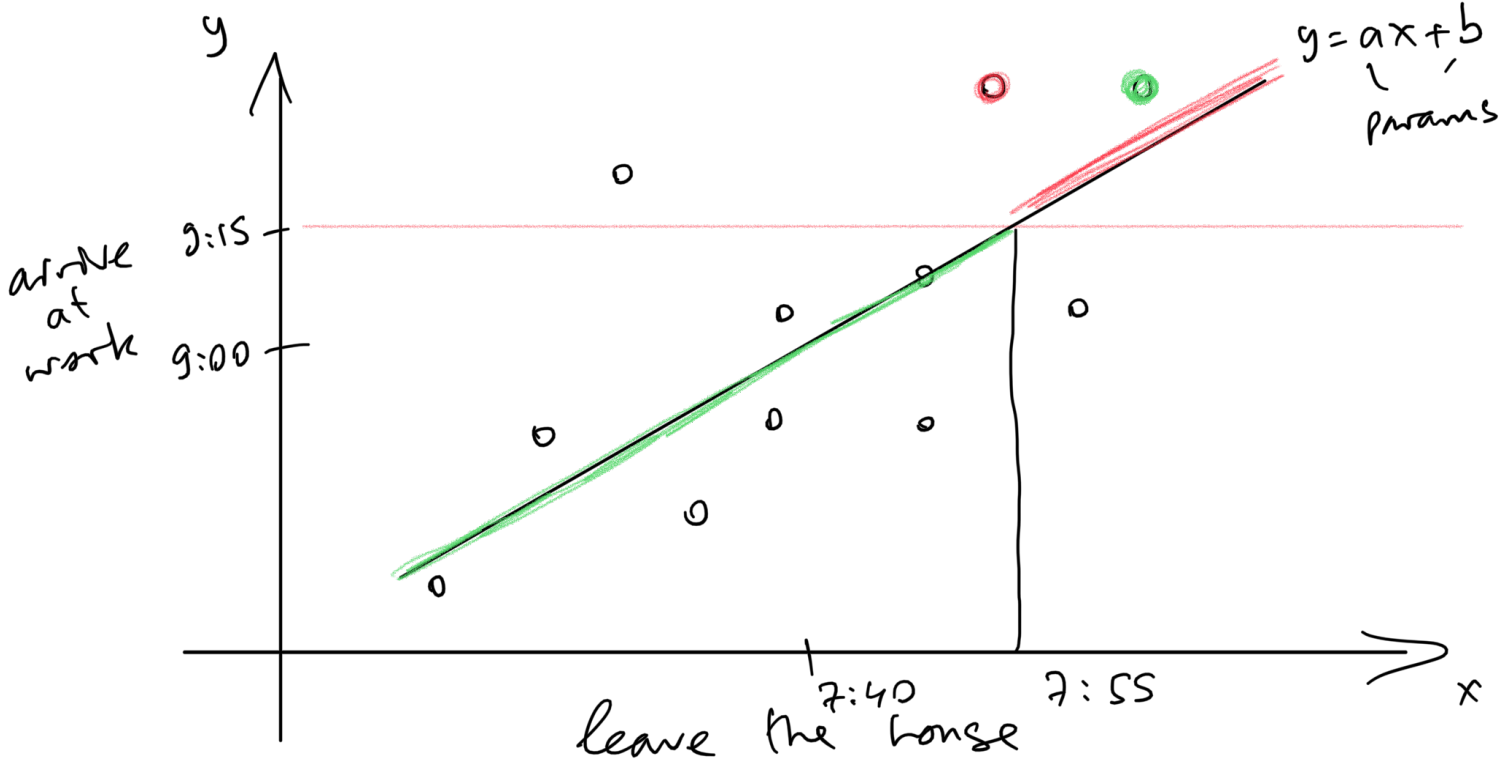
https://github.com/akusok/scikit-elm

- API documentation
  https://scikit-elm.readthedocs.io/en/latest/generated/skelm.ELMRegressor.html#skelm.ELMRegressor
- Workshop tutorials
  https://github.com/akusok/scikit-elm/blob/master/workshop/2021-12%20ELM%20Workshop.ipynb

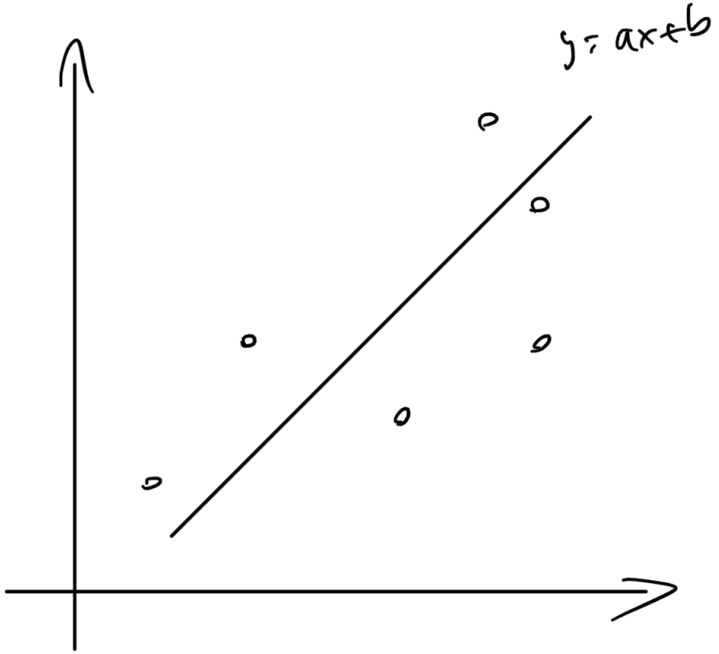
https://github.com/akusok/hpelm

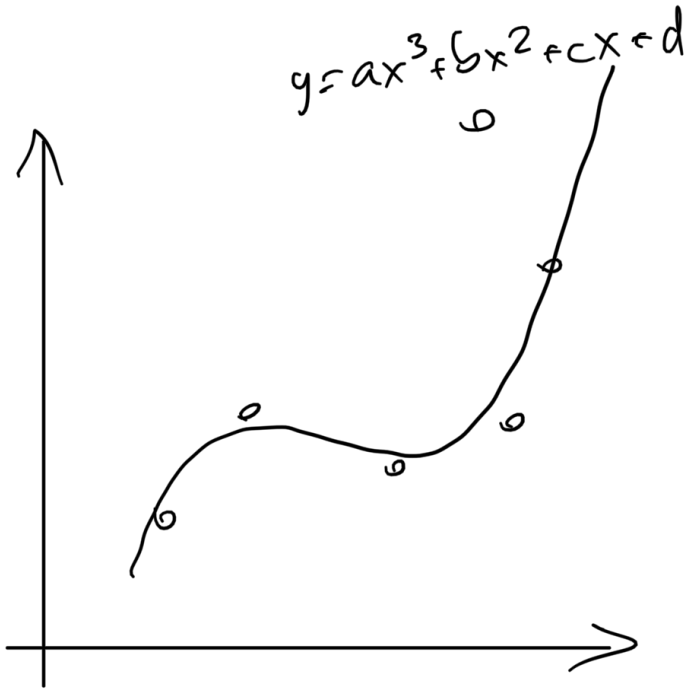
- API documentation
  https://hpelm.readthedocs.io/en/latest/api/hpelm.html
- Running HPELM in parallel on multiple machines (CSC)
  https://hpelm.readthedocs.io/en/latest/parallel.html

2. What is a model?

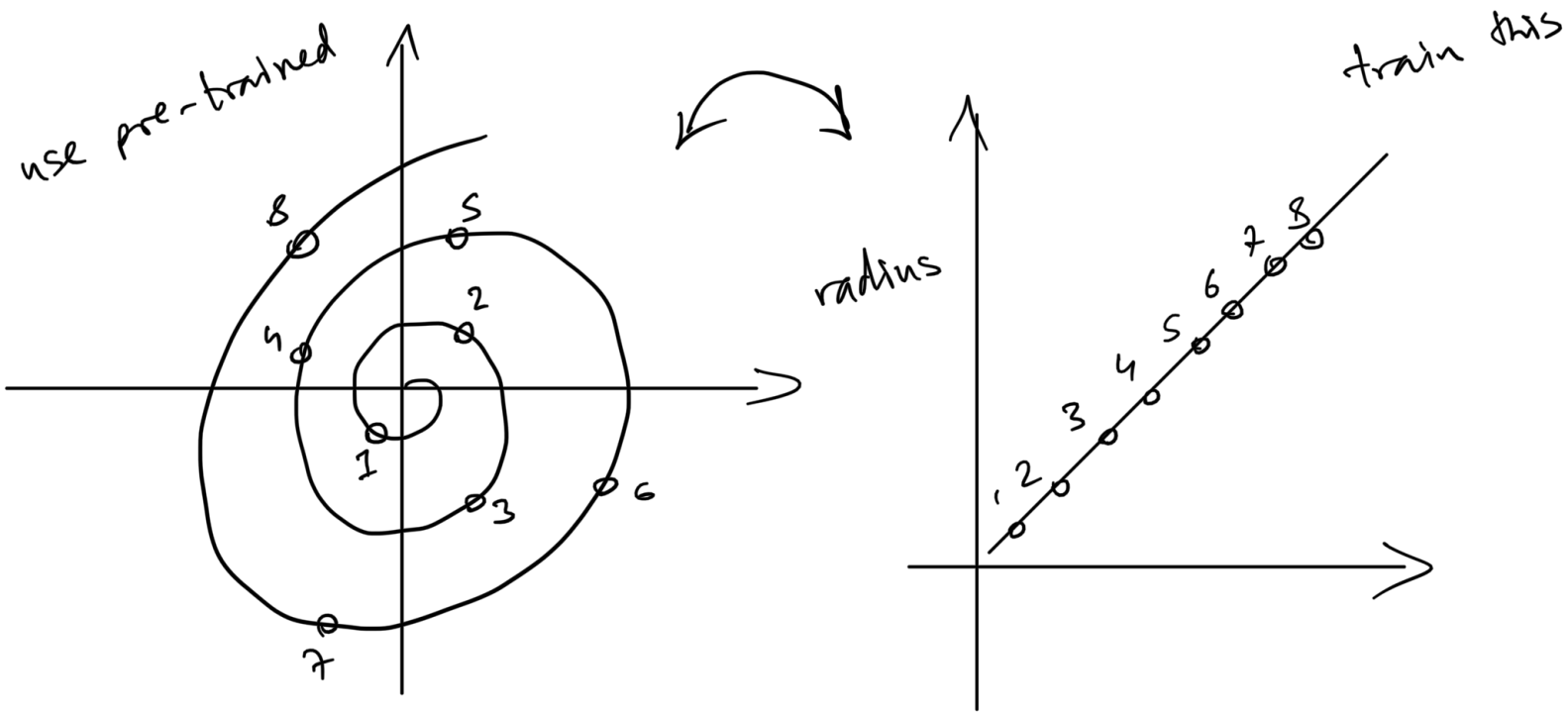


2.1 Linear model vs non-linear model

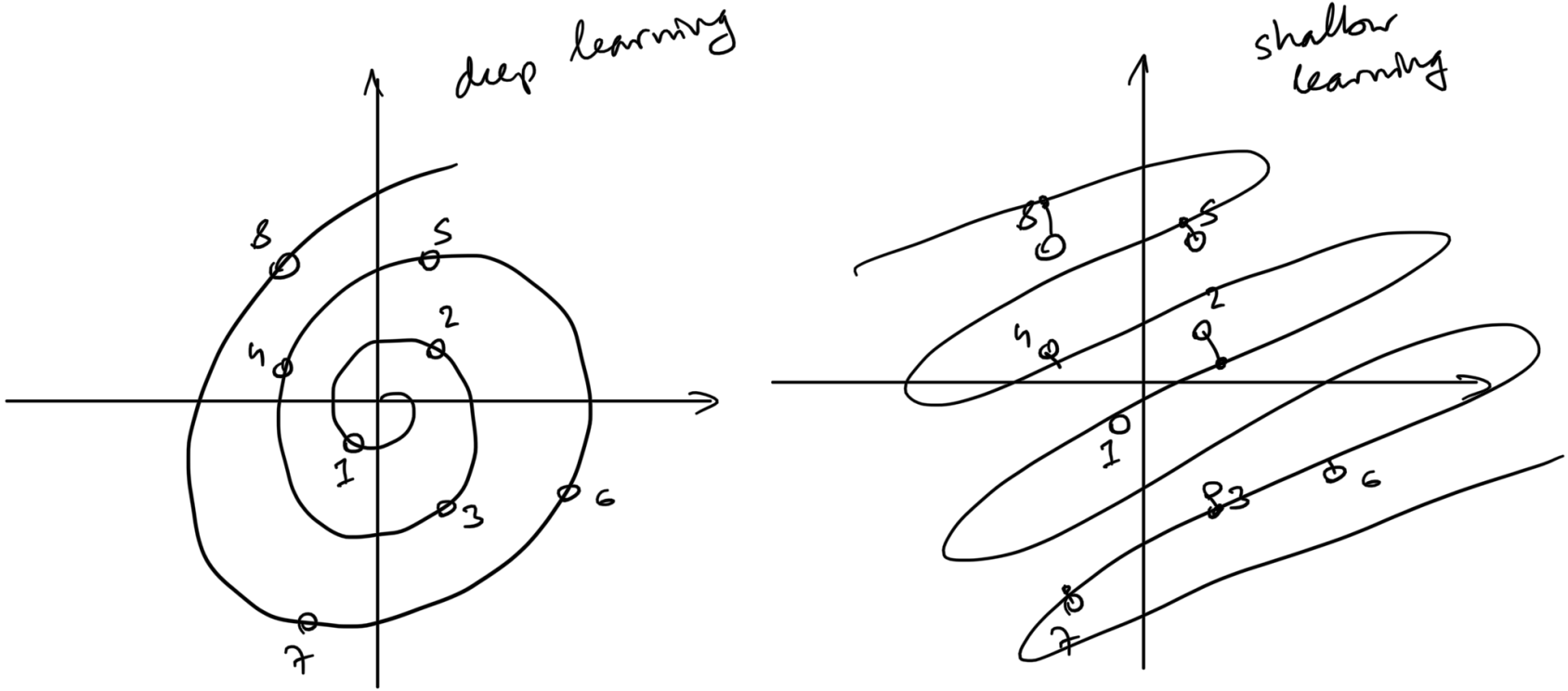




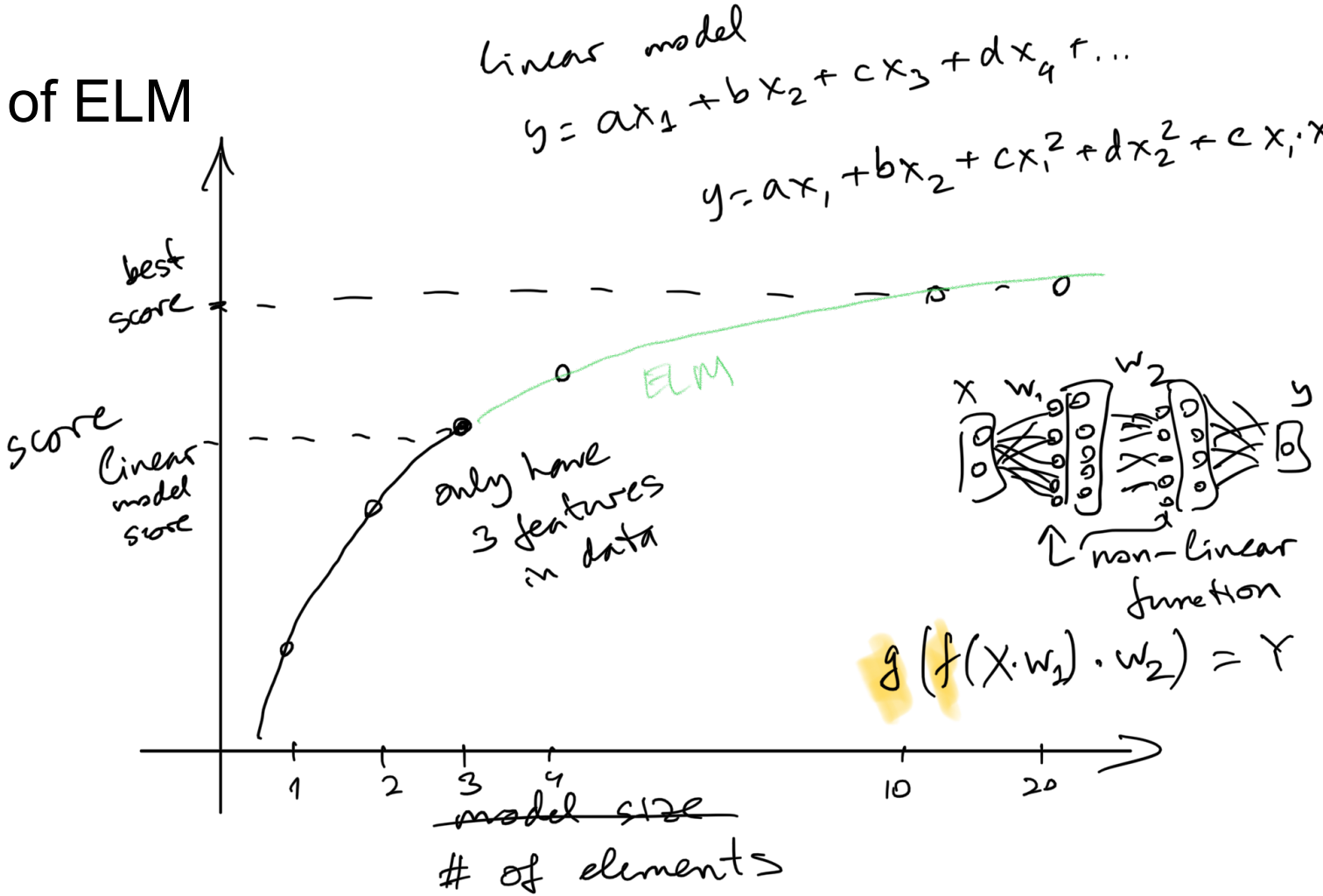
2.2 Deep Learning ~~model~~ transformation



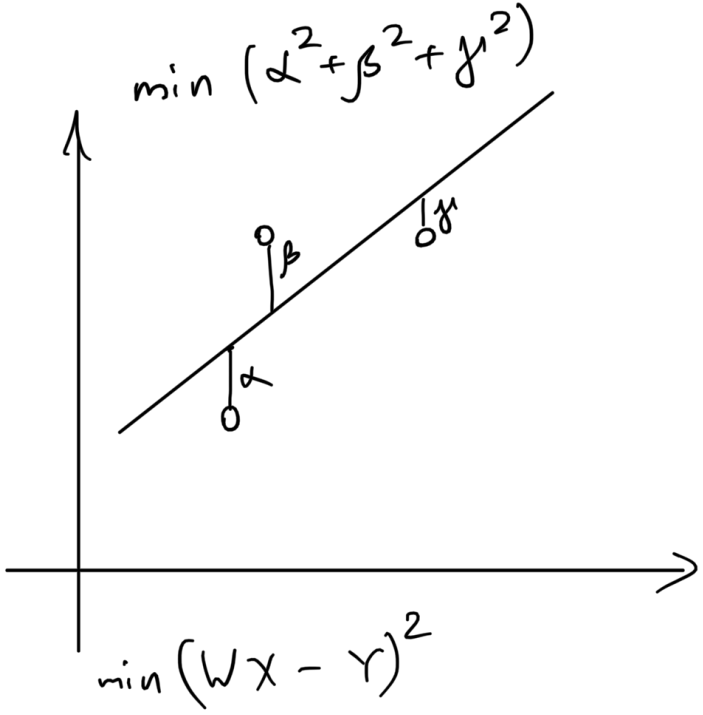
2.3 Deep vs Shallow Learning



3. Idea of ELM


linear model
$y = ax_1 + bx_2 + cx_3 + dx_4 + \ldots$
$y = ax_1 + bx_2 + cx_1^2 + dx_2^2 + e x_1 \cdot x_2 + \ldots$
best score
score
linear model score
ELM
only have 3 features in data
X
w_1
w_2
y
non-linear function
$g(f(X \cdot w_1) \cdot w_2) = Y$
1
2
3
4
10
20
model size
# of elements

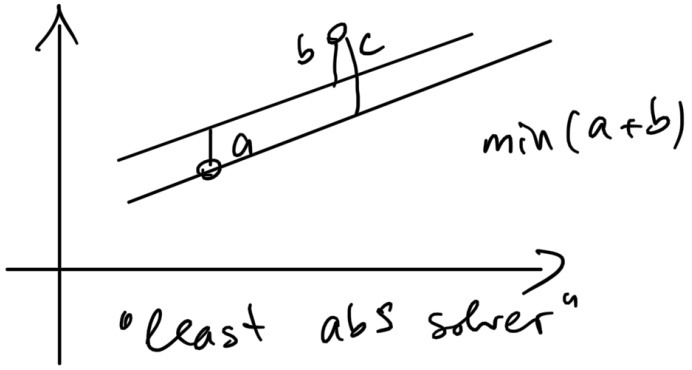
3.1 Least Squares solver



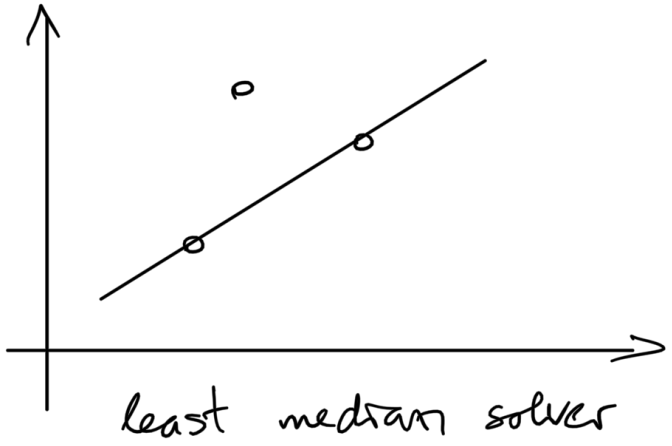
$\min (Wx - Y)^2$

$2W x^2 - 2 xY = 0$

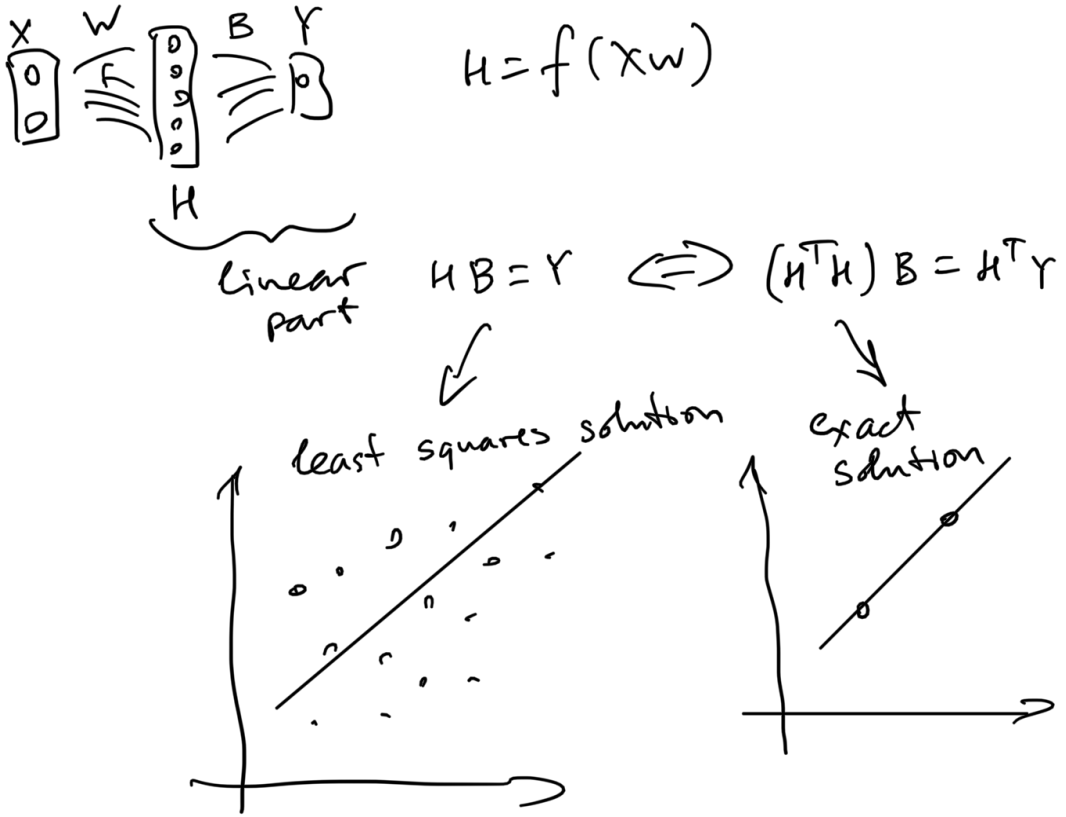
$W = x/Y$



$a+b = 0+c$



3.2 H^TH trick


X
W
B
Y
H
$H = f(XW)$
Linear part
$HB = Y \rightleftharpoons (H^TH)B = H^TY$
least squares solution
exact solution

3.3 Why we want a bias

$H = f(xW + bias_1)$

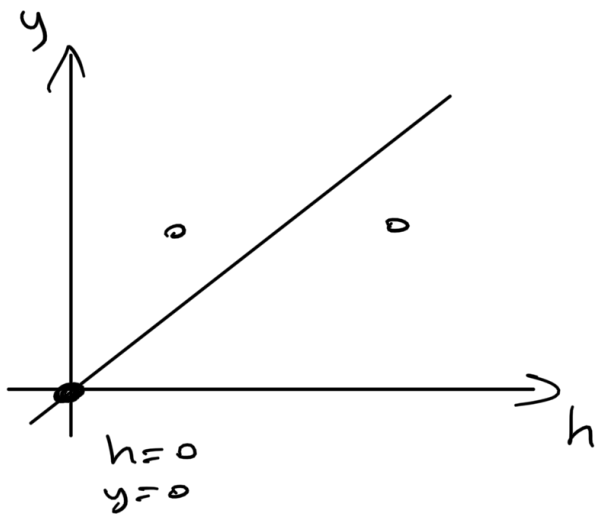
$HB + bias_2 = Y$

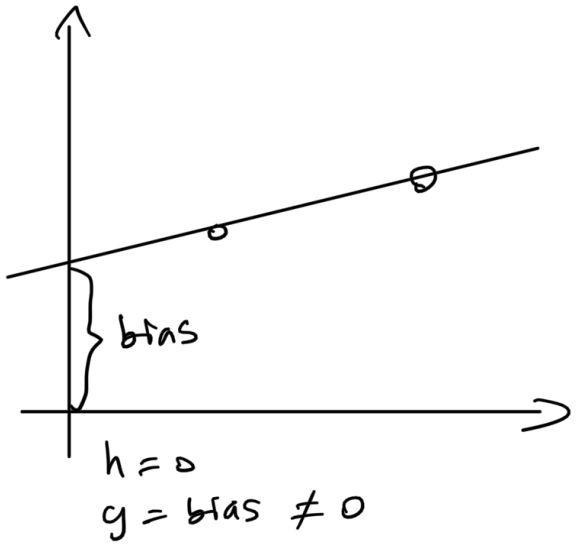
$HB = Y$

$HB + bias = Y$

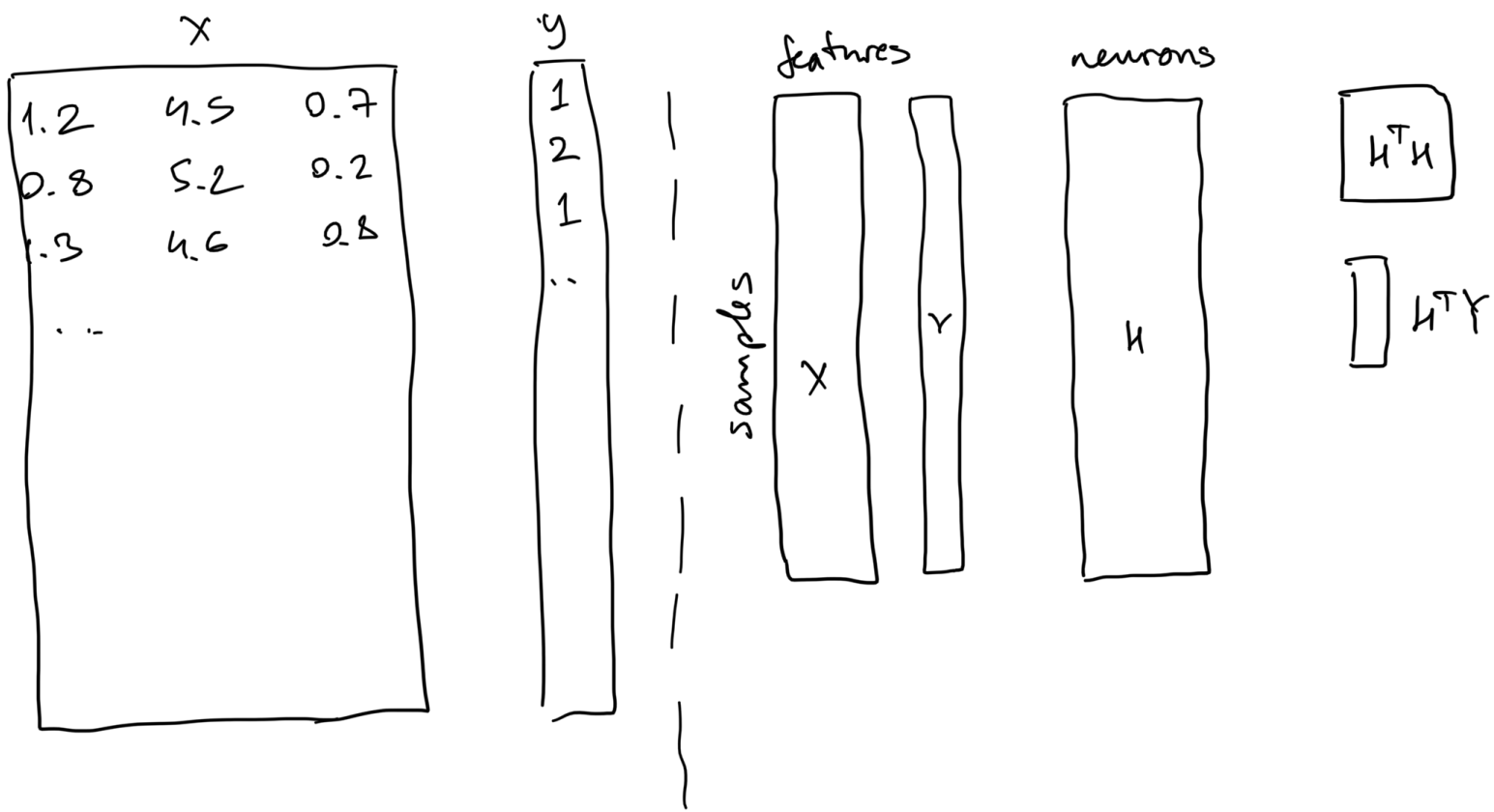
$h = 0$

$y = 0 \leftarrow$ has to be

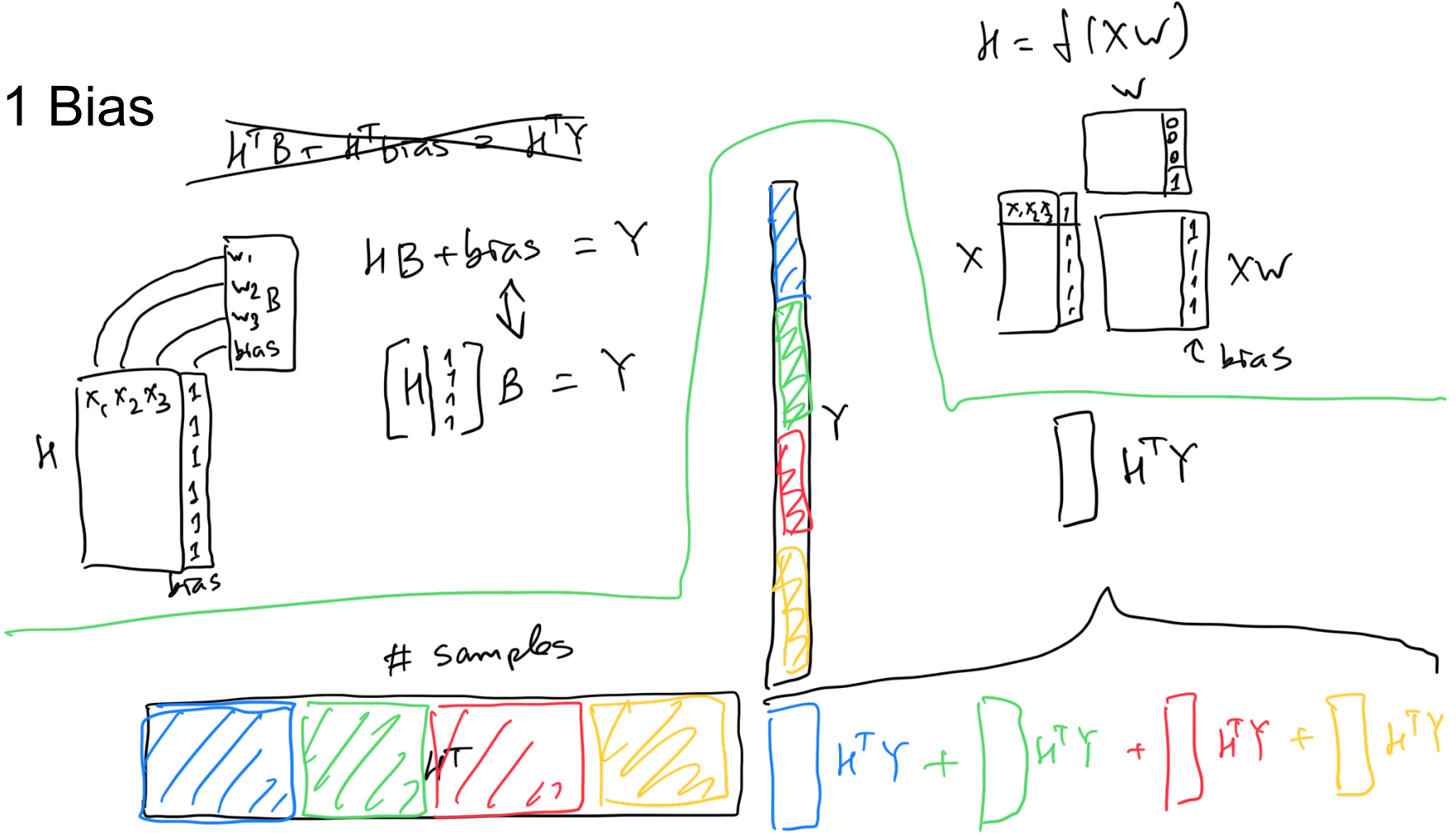




4. Parts of ELM



4.1 Bias


H = f(XW)
W
X
XW
bias
H
bias
HB + bias = Y
[H | 1 1 1 1] B = Y
Y
$H^T Y$
samples
H^T
$H^T Y + H^T Y + H^T Y + H^T Y$

4.2 Solvers

2.5 Fast Computation of Output Weights

Several practical methods for computing output weights $\boldsymbol{\beta}$ are described below. Note that other methods are available, see for instance the matrix solver selection algorithm[7] of MATLAB™.

Through matrix inversion as on equation (11). This approach is not recommended as ELM never needs an explicit inverse matrix $(\mathbf{H}^T\mathbf{H})^{-1}$ but the inverse is a slow operation, and it adds numerical instability to the solution.

$$\boldsymbol{\beta} = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{Y} \tag{11}$$

Through SVD decomposition as on equation (12) - slow but numerically stable, suitable for ill-conditioned matrix $\mathbf{H}^T\mathbf{H}$

$$\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T = \mathbf{H}^T\mathbf{H} \tag{12}$$

scipy.linear.svd()

$$\boldsymbol{\beta} = \mathbf{V}\boldsymbol{\Sigma}^+\mathbf{U}^T\mathbf{H}^T\mathbf{Y} \tag{13}$$

where $\boldsymbol{\Sigma}^+$ is a diagonal matrix with all its elements replaced by their inverse values.

Through Cholesky decomposition as on equation (14), solving $\boldsymbol{\beta}$ by double back-substitution with triangular matrices. This approach is very fast and has a block form for out-of-core learning. Cholesky decomposition fails on ill-conditioned matrix $\mathbf{H}^T\mathbf{H}$, that can be fixed by a large amount of L2 regularization.

scipy.linear.cho_solve()

$$\mathbf{L}\mathbf{L}^T = \mathbf{H}^T\mathbf{H} \tag{14}$$

$$\mathbf{L}(\mathbf{L}^T\boldsymbol{\beta}) = \mathbf{H}^T\mathbf{Y} \tag{15}$$

Through QR decomposition as on equation (16), solving $\boldsymbol{\beta}$ by back-substitution with a triangular matrix $\mathbf{R}$.

scipy.linear.solve()

$$\mathbf{Q}\mathbf{R} = \mathbf{H}^T\mathbf{H} \tag{16}$$

$$\mathbf{R}\boldsymbol{\beta} = \mathbf{Q}^T\mathbf{H}^T\mathbf{Y} \tag{17}$$

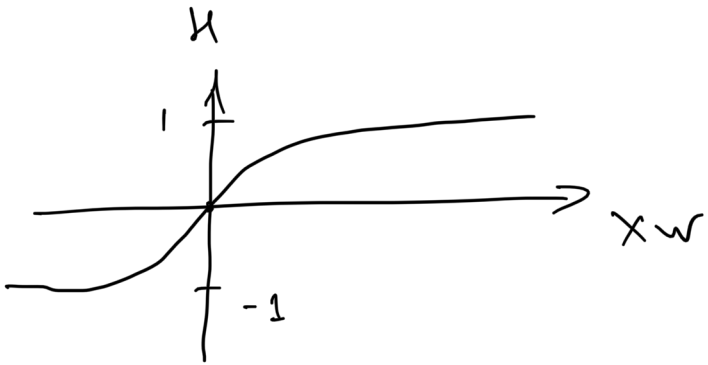
$(H^TH)\beta = H^TY$

~~$(H^TH)^{-1}(H^TH)\beta = (H^TH)^{-1}H^TY$~~

1

never use

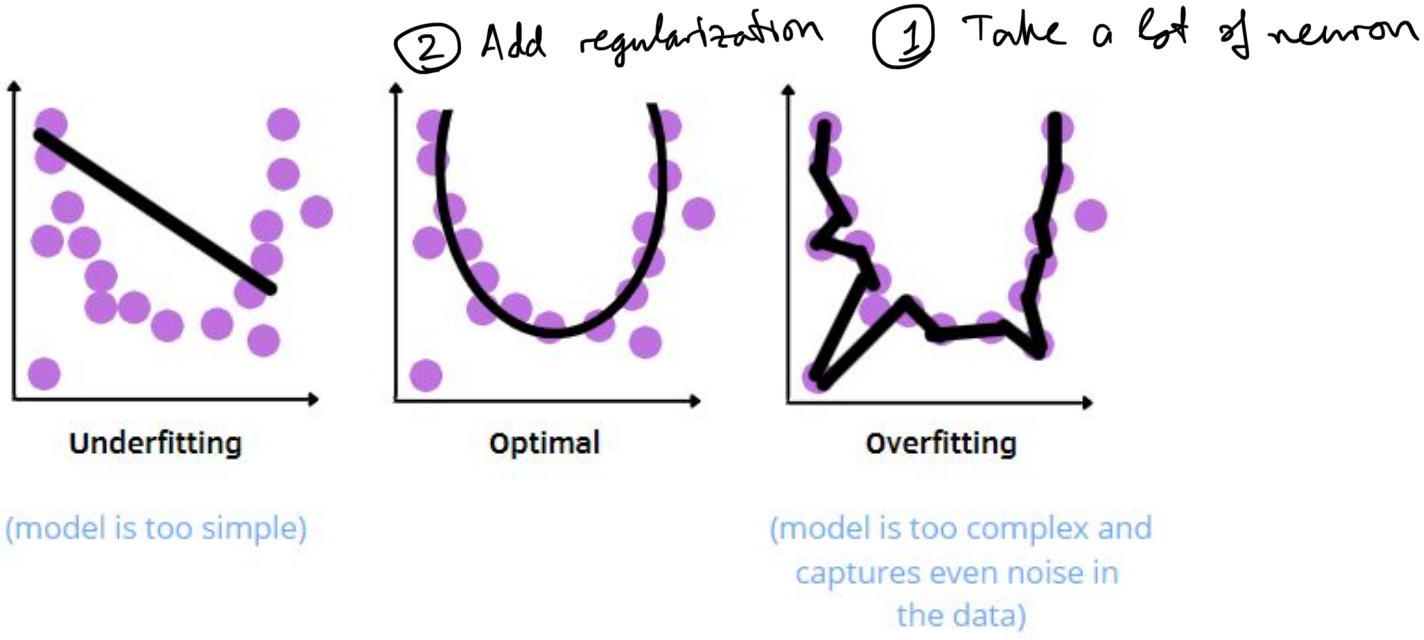
4.3 Hidden neurons



4.4 L2 regularization

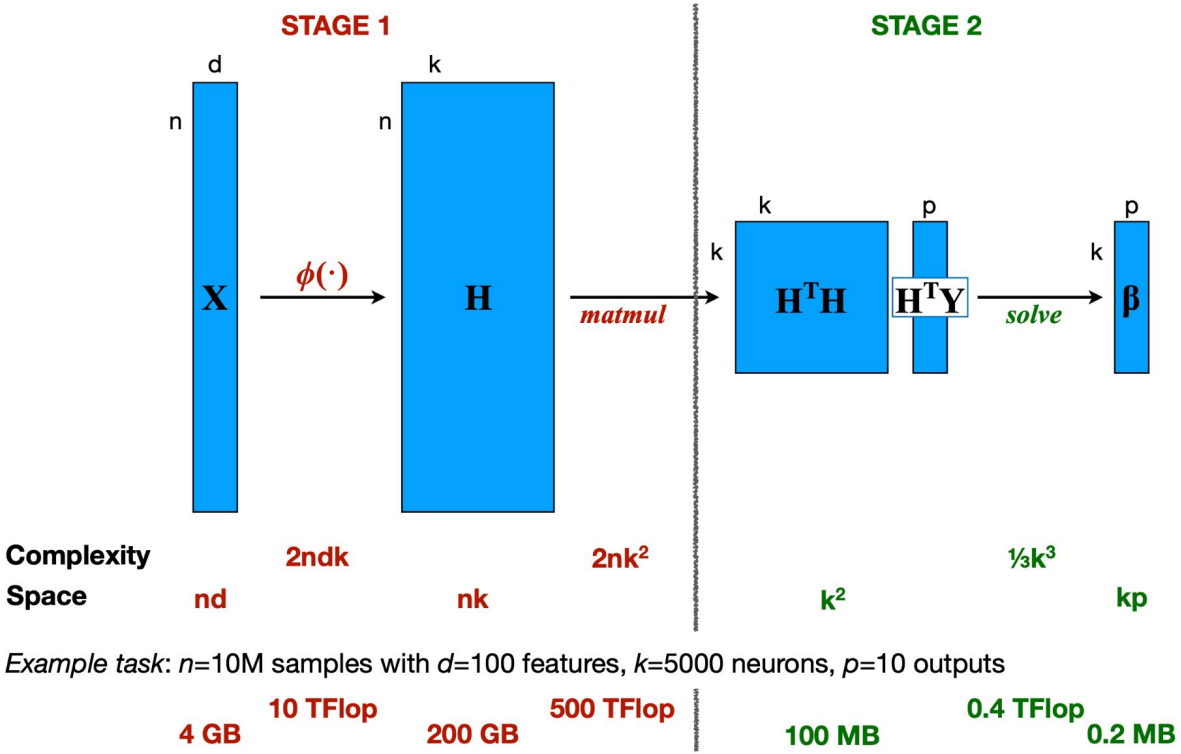
L1 regularization for reference
https://hpelm.readthedocs.io/en/latest/api/mrsr.html

Comparison of L1 and L2 regularization	
L1 regularization	**L2 regularization**
Sum of absolute value of weights	Sum of square of weights
Sparse solution	Non-sparse solution
Multiple solutions	One solution
Built-in feature selection	No feature selection
Robust to outliers	Not robust to outliers (due to the square term)

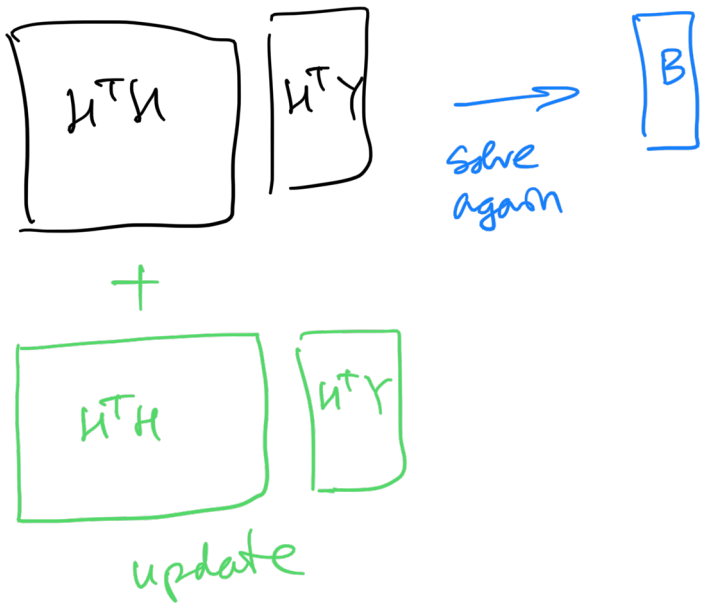


5. Two-stage computation on ELM

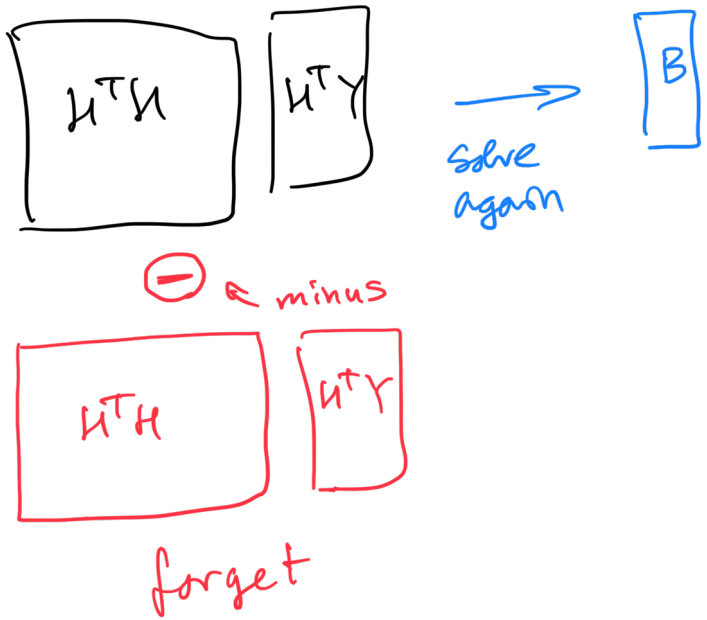
5. Two-stage computation on ELM



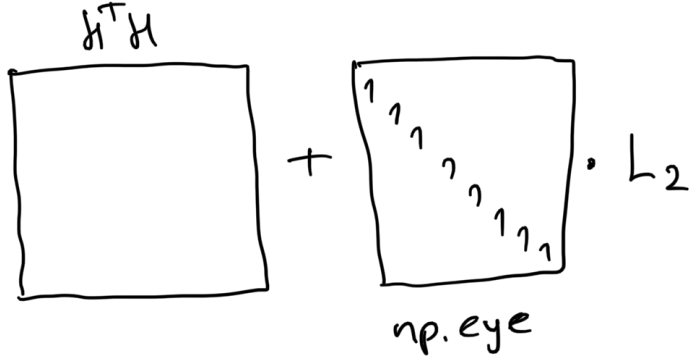
5.1 Adding data to trained ELM model

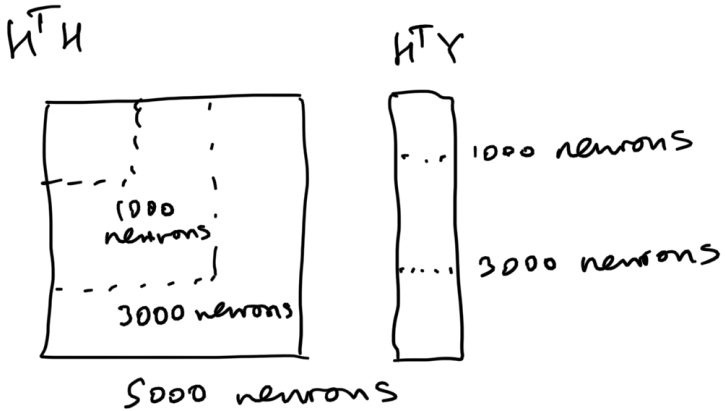


5.2 **Forgetting** data in trained ELM model



5.3 Post-training model tuning





Post-training model tuning

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

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.

```
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
```

Post-training model downsizing

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

```
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
```

5.4 Distributed ELM computation

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.

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

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

load model with same # of inputs

```
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")
```

```
import pickle
```

heaviest computation part

save to disk

```
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)
```

combine $H^TH = H_1^TH_1 + H_2^TH_2 + H_3^TH_3$

```
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
```

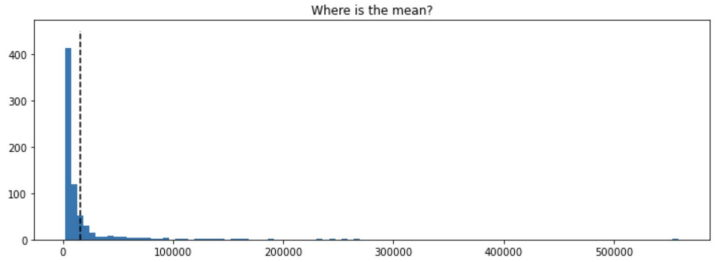
```
B = np.linalg.lstsq(HH, HY, rcond=-1)[0]

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
```

6. ELM best practices

Let's examine the mean values

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



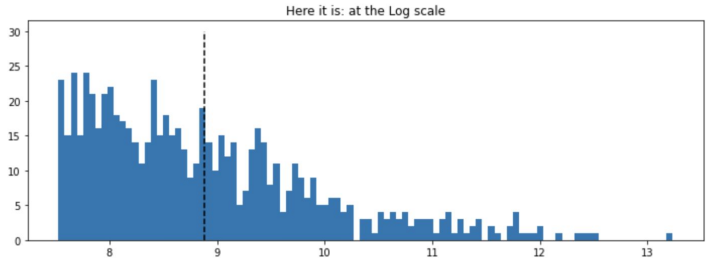
Logarithmically distributed variables have no concept of `mean`.

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.)

```
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()
```



6.1 Data scaling

```
from sklearn.preprocessing import RobustScaler
```

```
rs = RobustScaler().fit(X_train)
X_train = rs.transform(X_train)
X_test = rs.transform(X_test)
```

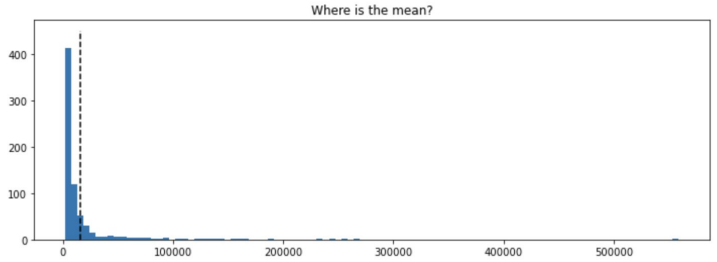
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.RobustScaler

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder

6.2 Log data scaling

Let's examine the mean values

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



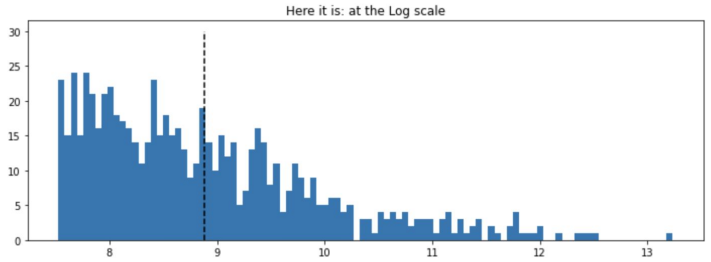
Logarithmically distributed variables have no concept of `mean`.

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.)

```
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()
```



6.3 Save/load model

```
import sys
sys.path.append("../")
```

```
import numpy as np
import pickle
from skelm import ELMRegressor
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
```

```
X, y = load_diabetes(return_X_y=True)
x1, x2, y1, y2 = train_test_split(X, y)
```

```
model = ELMRegressor(n_neurons=10, random_state=0)
model.partial_fit(x1, y1)

with open("ELM_model.pkl", "wb") as elm_file:
    pickle.dump(model, elm_file, pickle.HIGHEST_PROTOCOL)
```

```
with open("ELM_model.pkl", "rb") as elm_file:
    new_model = pickle.load(elm_file)

model.partial_fit(x2, y2)
yh = model.predict(X)
```

```
model_compare = ELMRegressor(n_neurons=10, random_state=0)
model_compare.fit(X, y)
yh2 = model_compare.predict(X)
```

```
np.allclose(yh, yh2)
```

6.3 Save/load model to Keras and **iOS**