

# 深度學習：基礎及應用

## Pima Indians Diabetes Database-Predict the onset of diabetes based on diagnostic measures

106753020 資科碩三 李俊毅

108753205 資科碩一 蔡宗諺

108753208 資科碩一 葉冠宏

### 1. Introduction

India had an estimated 31,705,000 diabetics in the millennium year which is estimated to grow by over 100% to 79,441,000 by 2030. According to the International Diabetes Federation Atlas 2015, an estimated 69.2 million Indians are diabetic, which as per the WHO assessment, stood at 63 million in the year 2013.

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

The datasets consist of several medical predictor variables and one target variable, Outcome. Predictor variables include the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

We build an improved deep learning model to accurately predict whether or not the patients in the dataset have diabetes. The eight variables are listed below.

- Pregnancies: Number of times pregnant
- Glucose: Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- BloodPressure: Diastolic blood pressure (mm Hg)
- SkinThickness: Triceps skin fold thickness (mm)
- Insulin: 2-Hour serum insulin ( $\mu$ U/ml)
- BMI: Body mass index ( $\text{weight in kg}/(\text{height in m})^2$ )

- DiabetesPedigreeFunction: Diabetes pedigree function
- Age: Age (years)
- Outcome: Class variable (0 or 1) 268 of 768 are 1, the others are 0

## 2. Related Work

- **Single-layer learning revisited: a stepwise procedure for building and training a neural network**

A stepwise procedure for building and training a neural network intended to perform classification tasks, based on single layer learning rules, is presented. This procedure breaks up the classification task into subtasks of increasing complexity in order to make learning easier. The network structure is not fixed in advance: it is subject to a growth process during learning. Therefore, after training, the architecture of the network is guaranteed to be well adapted for the classification problem.

- **Classification ability of single hidden layer feedforward neural networks**

Multilayer perceptrons with hard-limiting (signum) activation functions can form complex decision regions. It is well known that a three-layer perceptron (two hidden layers) can form arbitrary disjoint decision regions and a two-layer perceptron (one hidden layer) can form single convex decision regions. This paper further proves that single hidden layer feedforward neural networks (SLFN) with any continuous bounded non-constant activation function or any arbitrary bounded (continuous or not continuous) activation function which has unequal limits at infinities (not just perceptrons) can form disjoint decision regions with arbitrary shapes in multidimensional cases, SLFN with some unbounded activation function can also form disjoint decision regions with arbitrary shapes.

- **Sales forecasting using extreme learning machine with applications in fashion retailing**

Sales forecasting is a challenging problem owing to the volatility of demand which depends on many factors. This is especially prominent in fashion retailing where a versatile sales forecasting system is crucial. This study applies a novel neural network technique called extreme learning machine (ELM) to investigate the relationship between sales amount and some significant factors which affect demand (such as design factors). Performances of our models are evaluated by using real data from a fashion retailer in Hong Kong. The experimental results demonstrate that our

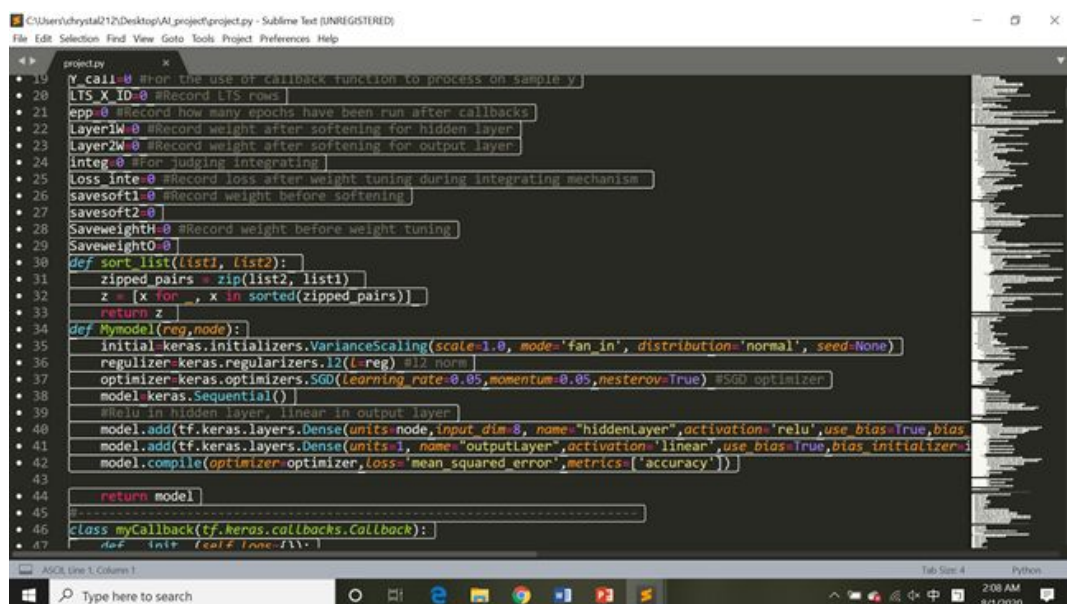
proposed methods outperform several sales forecasting methods which are based on backpropagation neural networks.

### 3. Methods

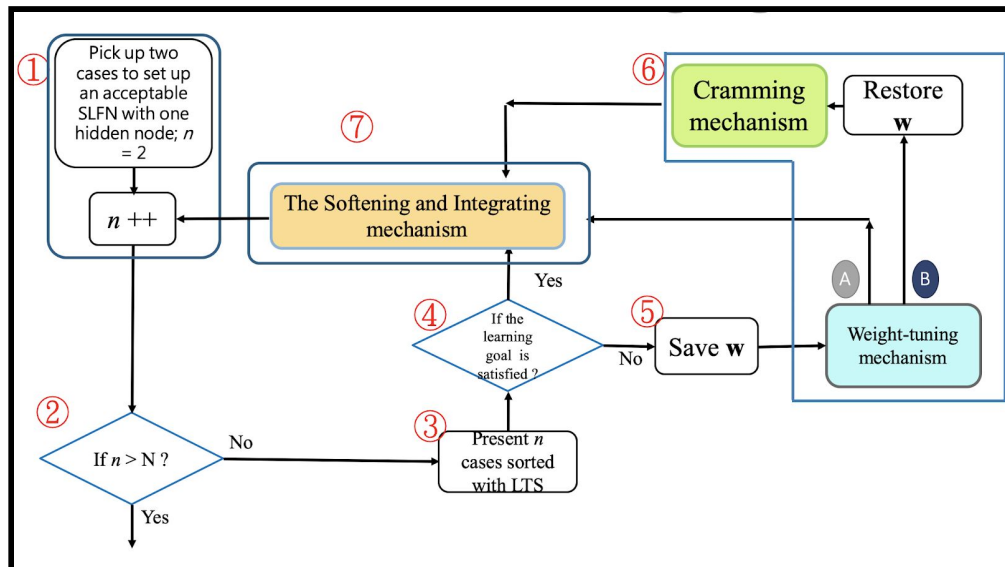
Many people use basic logistic regression algorithms or other classification algorithms to predict the outcome. In our model, we use a modified deep learning model to train our data.

Traditionally, deep learning models have multiple layers. In our model, we use a single hidden layer feedforward neural network(SLFN) to predict the data. Here, we use ReLu function for the first layer and linear function for the output layer. We also set our initial amount of hidden node as 1 since we start training data from only 2 samples and we want to minimize the amount of hidden nodes as many as possible.

We use Keras, Tensorflow 2.0, and Python to implement our experiment. Here are some of our initial settings. We set our regularizers as l2 norm with initial rate of 0.0001. And the dimension of the first layer is 8 since we have eight features in our training set. Besides, we use the optimizer SGD, and set our learning rate as 0.05, momentum rate as 0.05. In the 'compile' setting. We set our loss learning goal as minimizing the mean squared error.



```
19 project.py
20 # For the use of callback function to process on sample y
21 LTS X ID 0 #Record LTS row
22 epp 0 #Record how many epochs have been run after callbacks
23 Layer1W 0 #Record weight after softening for hidden layer
24 Layer2W 0 #Record weight after softening for output layer
25 integ 0 #For judging integrating
26 Loss inte 0 #Record loss after weight tuning during integrating mechanism
27 savesoft1 0 #Record weight before softening
28 savesoft2 0
29 SaveweightH 0 #Record weight before weight tuning
30 SaveweightO 0
31 def sort_list(list1, list2):
32     zipped_pairs = zip(list2, list1)
33     z = [(x, y) for x, y in sorted(zipped_pairs)]
34     return z
35 def MyModel(reg,node):
36     initial_keras.initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='normal', seed=None)
37     regularizer=keras.regularizers.l2(L=reg) #l2 norm
38     optimizer=keras.optimizers.SGD(learning_rate=0.05, momentum=0.05, nesterov=True) #SGD optimizer
39     model=keras.Sequential()
40     #Relu in hidden layer, linear in output layer
41     model.add(tf.keras.layers.Dense(units=node, input_dim=8, name="hiddenLayer", activation='relu', use_bias=True, bias_initializer='zeros'))
42     model.add(tf.keras.layers.Dense(units=1, name="outputLayer", activation='linear', use_bias=True, bias_initializer='zeros'))
43     model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['accuracy'])
44     return model
45
46 class myCallback(tf.keras.callbacks.Callback):
47     def on_train_begin(self, logs=None):
```

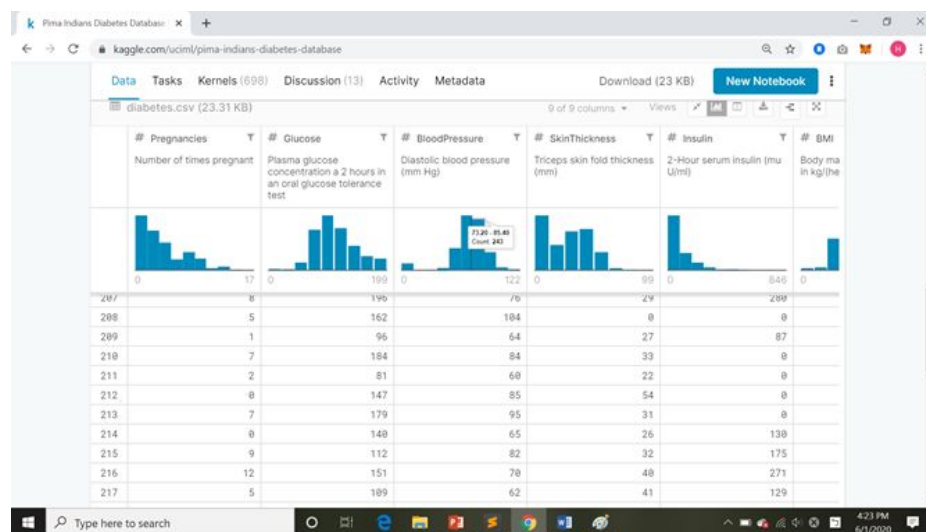


## SLFN Network Flow chart

### Step1:Data preprocessing and initial setting

We use MinMaxScaler to preprocess the data. And we get the initial weight by generating a random weight to train only two data sets to get 100% prediction rate.

$$X_{\text{new}} = \frac{X_i - \min(X)}{\max(x) - \min(X)}$$



### Step 2:Find out the LTS sample

We predict all N samples first, calculate their squared error and sort them. And then we select n samples of least squared error as our LTS sample at each stage to train our model. The initial LTS sample n is 2. We increment 1 sample to train until n>=N.

## Literature Review

## Least Trimmed Squares (LTS)

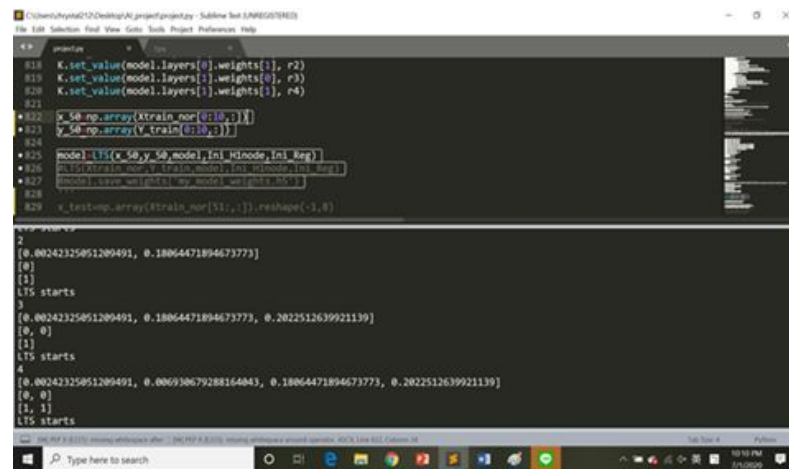
(Tsaih & Cheng, 2009; Tsaih, et al., 2018)

- Pick up the **first**  $n$  reference observations  $\{(x^c, y^c)\}$  which are **sorted** by **all**  $N$  reference observations' **squared residuals** in **ascending** order.

$$(e^{[1]})^2 \leq (e^{[2]})^2 \leq \dots \leq (e^{[n]})^2, \text{ in which } (e^c)^2 = (f(x^c, w) - y^c)^2$$

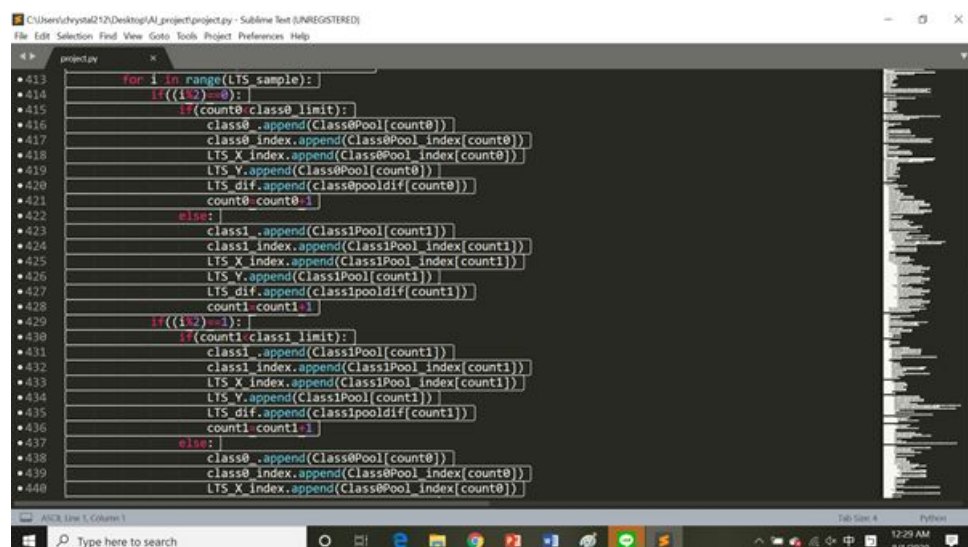
- Detect the potential outliers in (Tsaih & Cheng, 2009; Tsaih, et al., 2018)
- Accelerate the learning process and mimic the human learning

The result below shows the ascending order of squared error at each stage and the result of classification.



```
project.py
818 K.set_value(model.layers[0].weights[1], r2)
819 K.set_value(model.layers[1].weights[0], r3)
820 K.set_value(model.layers[1].weights[1], r4)
821
822 y_5d=np.array(X[train_nor[0:10,2]])
823 y_5d=np.array(y_train[0:10,2])
824
825 model.fit(x_5d,y_5d,model,[1],x_node,[1],reg)
826 print(X[train_nor[0:10,2]],model,[1],x_node,[1],reg)
827 model.save_weights('lts_5d_model_weights.h5')
828
829 x_test=np.array(X[train_nor[11,2]]).reshape(-1,8)
830
831 print(x_test)
832
833 [0.00242325051209491, 0.18064471894673773]
834 [0]
835 [1]
836 LTS starts
837 [0.00242325051209491, 0.18064471894673773, 0.2022512639921139]
838 [0, 0]
839 [1]
840 LTS starts
841 [0.00242325051209491, 0.006930679288164043, 0.18064471894673773, 0.2022512639921139]
842 [0, 0]
843 [1, 1]
844 LTS starts
```

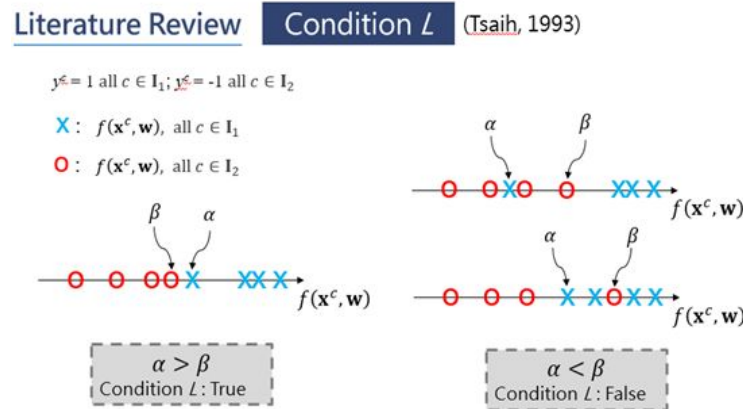
We select  $n$  samples from each side of the class pool. Otherwise, one sample class may be empty during the training process because  $n$  samples with least squared error may all come from the same class in  $N$ . (Condition L fails)



```
project.py
413 for i in range(LTS_sample):
414     if((i%2)==0):
415         if(count0>class0_limit):
416             class0.append(Class0Pool[count0])
417             class0_index.append(Class0Pool_index[count0])
418             LTS_X_index.append(Class0Pool_index[count0])
419             LTS_Y.append(Class0Pool[count0])
420             LTS_dif.append(Class0Pool_dif[count0])
421             count0=count0+1
422         else:
423             class1.append(Class1Pool[count1])
424             class1_index.append(Class1Pool_index[count1])
425             LTS_X_index.append(Class1Pool_index[count1])
426             LTS_Y.append(Class1Pool[count1])
427             LTS_dif.append(Class1Pool_dif[count1])
428             count1=count1+1
429     if((i%2)==1):
430         if(count1>class1_limit):
431             class1.append(Class1Pool[count1])
432             class1_index.append(Class1Pool_index[count1])
433             LTS_X_index.append(Class1Pool_index[count1])
434             LTS_Y.append(Class1Pool[count1])
435             LTS_dif.append(Class1Pool_dif[count1])
436             count1=count1+1
437         else:
438             class0.append(Class0Pool[count0])
439             class0_index.append(Class0Pool_index[count0])
440             LTS_X_index.append(Class0Pool_index[count0])
```

**Step 3: Check whether condition L satisfies**

If we assign the minimum value of samples in LTS sample class 1 as  $\alpha$ , and maximum value of samples in LTS sample class 0 as  $\beta$ . Condition L satisfies when  $\alpha$  is larger than  $\beta$ . That is, all  $y$  values in LTS sample class 1 is 1, and all  $y$  values in LTS sample class 0 is 0. Classification succeeds. Otherwise, it fails.



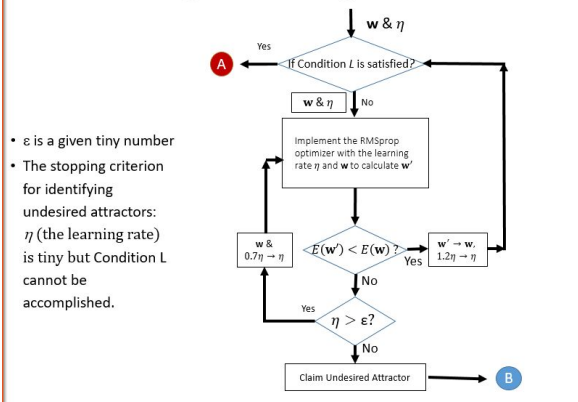
**Step4: Do weight tuning if condition L doesn't satisfy. Otherwise, it proceeds to softening and integrating stage.**

If it proceeds to the stage of weight-tuning, we implement gradient descent by setting our learning rate as 0.05, momentum rate as 0.05. We adapt our learning rate and momentum rate value based on your loss at each epoch compared with the previous epoch. If your loss at the current epoch is less than the previous epoch, you increase your learning rate and momentum rate by 1.2 times to speed up the training. Otherwise, we set our weight at the best weight trained so far and then minimize the learning rate and momentum rate by 0.7 times to gauge the appropriate further step.

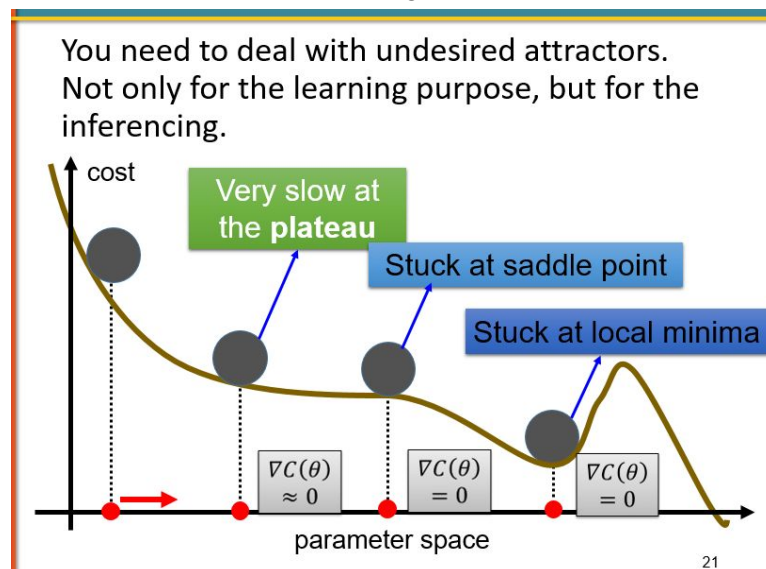
During the process, we will predict the LTS sample at each epoch and judge whether the condition L satisfies. If it does, we stop our weight-tuning process, record our classification result, and flag any information we need( Exits at A in the graph below). When we gauge the step, if the learning rate and momentum rate keeps minimizing until it reaches below our setting threshold, which is 0.0001, we stop the weight tuning process, return some necessary information, and flag our process as 'undesired attractor'.

We set our maximum epochs to run as 500 epochs. If it runs to the end before approaching to the ideal result (Exit at A), we treat it as an undesired attractor.

## The weight-tuning mechanism



The explanation of the undesired attractor is in the graph below.



As you can see, when we proceed to the LTS sample 36, the nth data doesn't satisfy the condition L. The nth number 0 is in our LTS sample class 1. Hence, we run the weight tuning mechanism.



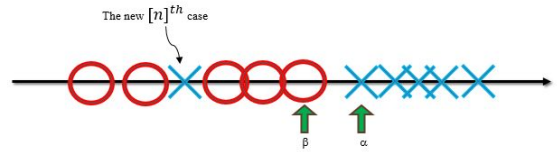




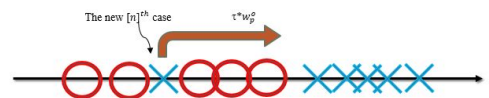
# The Cramming Mechanism

using the LTS

When we encounter with a new case that **cannot be learnt well** with the current SLFN and the weight-tuning mechanism, an extra hidden node is used to **cram** this case.



## The Cramming Mechanism



- Via adding an extra hidden node, we would like to pull this new case into the right place with other cases **staying still at the same place** (for instance, in the above scenario).
- set up the  $\tau$  value that renders  $w_p^g \tau > \max_{u \in I_2(n)} \sum_{i=1}^{p-1} w_i^g a_i^u > \sum_{i=1}^{p-1} w_i^g a_i^{[n]}$  be true.

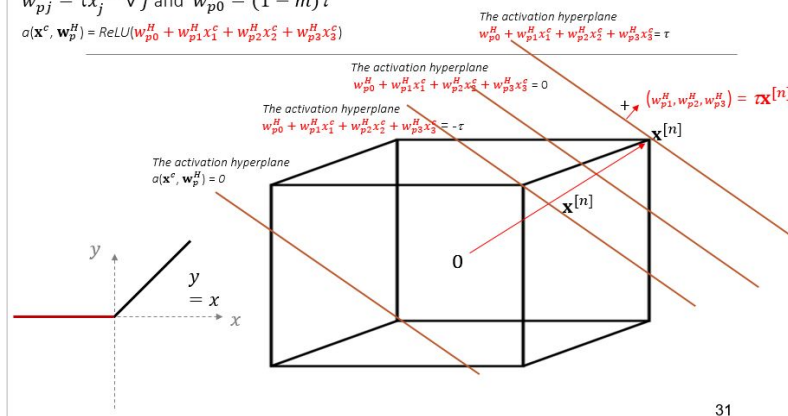
A hyper-parameter

Below is the mathematical explanation of cramming. Basically, what we do is to try to add up some new weight to fit the unfamiliar data to the model. The newly added weight doesn't have any effect on the first  $n-1$  data which already has been trained well. The newly added weight makes first  $n-1$  data negative, which it becomes 0 after the effect of ReLU. Only the unfamiliar  $n$ th data is positive, so that it can move from the wrong class to the right class. We create 3 hidden nodes in each cramming mechanism so as to 'clamp' the data. In the process, we set the number  $caci$  as  $1e-4$ , and generate random weight that can fit in the equation.

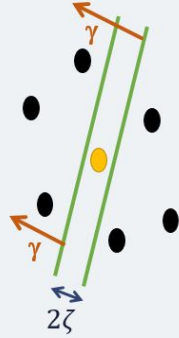
## The $\{-1,1\}^m$ space

$$w_{pj}^H = \tau x_j^{[n]} \forall j \text{ and } w_{p0}^H = (1-m)\tau$$

$$\sigma(\mathbf{x}^c, \mathbf{w}_p^H) = \text{ReLU}(w_{p0}^H + w_{p1}^H x_1^c + w_{p2}^H x_2^c + w_{p3}^H x_3^c)$$



## The Cramming Mechanism



Case of  $y^{[n]} = 1.0$

Step 1: Let  $\zeta$  be a given small number. Find an  $m$ -vector  $\gamma$  of length one such that  $\gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]}) \neq 0$  all  $c \in \mathcal{I}(n) - \{[n]\}$

AND  $(\zeta + \gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})) * (\zeta - \gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})) < 0$  all  $c \in \mathcal{I}(n) - \{[n]\}$ .

Step 2: Let  $p+3 \rightarrow p$  and add three new hidden nodes  $p-2^{\text{th}}$ ,  $p-1^{\text{th}}$  and  $p^{\text{th}}$  to the existing SLFN with

$$\square w_{p-2,0}^H = \zeta - \gamma^T \mathbf{x}^{[n]}, w_{p-2}^H = \gamma,$$

$$\square w_{p-1,0}^H = -\gamma^T \mathbf{x}^{[n]}, w_{p-1}^H = \gamma,$$

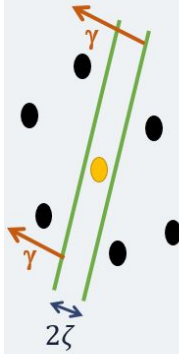
$$\square w_{p0}^H = -\zeta - \gamma^T \mathbf{x}^{[n]}, w_p^H = \gamma, \text{ and}$$

$$\square w_{p-2}^o = -0.5w_{p-1}^o = w_p^o = 1.1 \left( \frac{\max_{u \in \mathcal{I}_2(n)} \sum_{i=1}^{p-3} w_i^o a_i^u - \sum_{i=1}^{p-3} w_i^o a_i^{[n]}}{\zeta} \right)$$

$\tau = 1.1$

37

## The Cramming Mechanism



Case of  $y^{[n]} = -1.0$

Step 1: Let  $\zeta$  be a given small number. Find an  $m$ -vector  $\gamma$  of length one such that  $\gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]}) \neq 0$  all  $c \in \mathcal{I}(n) - \{[n]\}$

AND  $(\zeta + \gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})) * (\zeta - \gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})) < 0$  all  $c \in \mathcal{I}(n) - \{[n]\}$ .

Step 2: Let  $p+3 \rightarrow p$  and add four new hidden nodes  $p-2^{\text{th}}$ ,  $p-1^{\text{th}}$  and  $p^{\text{th}}$  to the existing SLFN with

$$\square w_{p-2,0}^H = \zeta - \gamma^T \mathbf{x}^{[n]}, w_{p-2}^H = \gamma,$$

$$\square w_{p-1,0}^H = -\gamma^T \mathbf{x}^{[n]}, w_{p-1}^H = \gamma,$$

$$\square w_{p0}^H = -\zeta - \gamma^T \mathbf{x}^{[n]}, w_p^H = \gamma, \text{ and}$$

$$\square w_{p-2}^o = -0.5w_{p-1}^o = w_p^o = 1.1 \left( \frac{\min_{u \in \mathcal{I}_1(n)} \sum_{i=1}^{p-3} w_i^o a_i^u - \sum_{i=1}^{p-3} w_i^o a_i^{[n]}}{\zeta} \right)$$

$\tau = 1.1$

38

*Contribution of three extra hidden nodes to the output*

- $\mathbf{x}^c$ : whose  $\gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})$  is either greater than  $\zeta$  or less than  $-\zeta$
- $\Delta f = w_p^o * [\text{ReLU}(\zeta + \gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})) - 2\text{ReLU}(\gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]})) + \text{ReLU}(-\zeta + \gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]}))]$

$\gamma^i(x^c - x^{[n]})$	$(p-2)^{\text{th}}$	$-2(p-1)^{\text{th}}$	$p^{\text{th}}$	$\Delta f$
$-2\zeta$	0	0	0	0
$-\zeta$	0	0	0	0
$-0.7\zeta$	$0.3\zeta$	0	0	$0.3\zeta w_p^o$
$-0.5\zeta$	$0.5\zeta$	0	0	$0.5\zeta w_p^o$
$-0.3\zeta$	$0.7\zeta$	0	0	$0.7\zeta w_p^o$
0	$\zeta$	0	0	$\zeta w_p^o$
$0.3\zeta$	$1.3\zeta$	$-0.6\zeta$	0	$0.7\zeta w_p^o$
$0.5\zeta$	$1.5\zeta$	$-1.0\zeta$	0	$0.5\zeta w_p^o$
$0.7\zeta$	$1.7\zeta$	$-1.4\zeta$	0	$0.3\zeta w_p^o$
$\zeta$	$2\zeta$	$-2\zeta$	0	0
$2\zeta$	$3\zeta$	$-4\zeta$	$\zeta$	0

44

The algorithm for creating an  $m$ -vector  $\gamma$  of length one such that  $\gamma^T(\mathbf{x}^c - \mathbf{x}^{[n]}) \neq 0$  all  $c \in \mathbf{I}(n) - \{[n]\}$

- Assume  $x^i \neq x^j$  when  $i \neq j$ .

Step 1: Set  $\beta_1 = 1$  and let  $k = 2$ .

Step 2: Let  $C_k \equiv \{c : c \in I(n) - \{[n]\} \text{ AND } x_j^c = x_j^{[n]} \text{ all } j = 1, \dots, k\}$ . Considering  $\beta_k$  as the unknown and  $\beta_j, j = 1, \dots, k-1$ , as previously determined, set  $\beta_k$  = the smallest integer that is greater than or equal to 1 and  $\sum_{j=1}^k \beta_j (x_j^c - x_i^{[n]}) \neq 0$  all  $c \in I(n) - \{[n]\} - C_k$ .

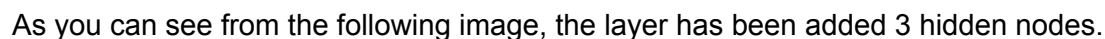
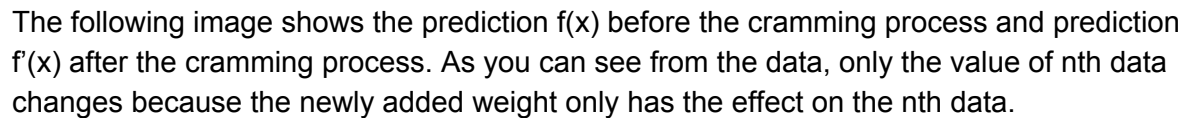
Step 3:  $k+1 \rightarrow k$ . If  $k \leq m$ , go to Step 2;

Step 4: Set  $\gamma_j = \frac{\beta_j}{\sqrt{\sum_{j=1}^m \beta_j^2}} \forall j = 1, \dots, m$  and STOP.

The following image shows our classification before cramming. As you can see, the nth sample doesn't satisfy the condition L . The nth y number 0 is in the LTS sample class 1.

[illegible]

As you can see from the following image, after cramming mechanism, the nth y number 0 that was originally in LTS sample class 1 moves to the right class, LTS sample class 0.



In this process, we first do the softening mechanism. The goal of softening mechanism is to try if we can further minimizing our weight by increasing our regularization rate. Here, we set our regularization rate as 1.2 times of the original. During the process, if any of the epoch fails to satisfy the condition L, we stop softening procedure, and restore the weight to the best weight trained so far. If it satisfies during the process, we keep increasing our

momentum rate and learning rate by 1.2 times of the original. After increasing the rate, we'll see if our current loss is less than the previous. If it doesn't, we first set our weight to the previous best weight and try to see if we can move further by minimizing our learning rate and momentum rate by 0.7 times. If the learning rate and the momentum rate is less than our threshold. We stop the softening process and restore weight to the best weight trained so far.

In the integrating mechanism, we try our model to see if we can cut 1 hidden node by temporarily ignoring the  $i$ th hidden node and it still satisfies the learning goal 'Condition L'. If the learning goal is satisfied, we first store the model and try if cutting the other  $i$ th node also works. If it works, we compare the loss with the reduced model that has been stored. And if it is better, we replace it with the newly testing model of reduced weight. After each successful integrating process, we will do the softening process of the reduced model to see if we can further minimize our weight.

### Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

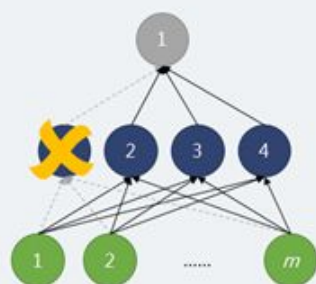
$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

- $R(w_1) = 1$
- $R(w_2) = 0.25$

## The Integrating Mechanism



The 1<sup>st</sup> hidden node is irrelevant and can be pruned

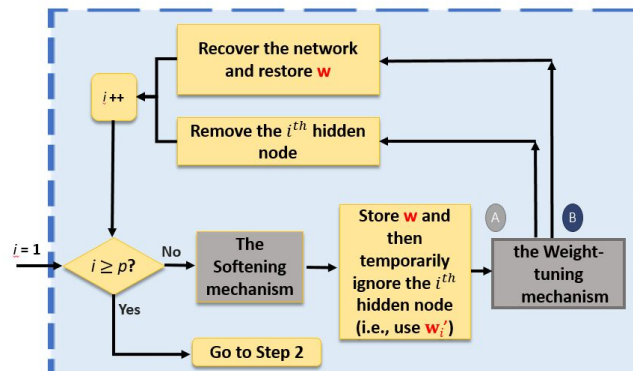
Case: Condition L

	$i=1$	$i=2$	$i=3$	$i=4$
$\alpha'_i$	= 0.2	= -0.1	= -0.2	= 0.05
$\beta'_i$	= -0.1	= 0.3	= 0.71	= 0.08
$g'_i$	= 0.3	= -0.4	= -0.91	= -0.03

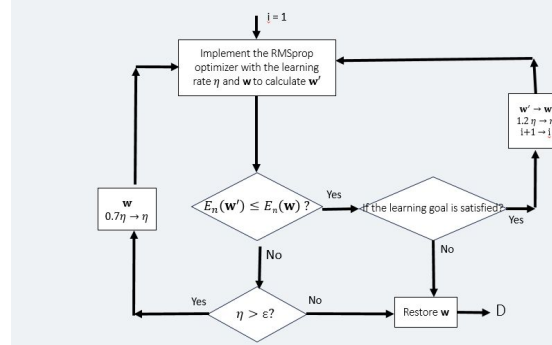
$$\max_{1 \leq k \leq p} g'_k > 0$$



## The Softening and Integrating mechanism (I) (sequentially check all hidden nodes)



## The softening mechanism with the regularization term



As you can see from the following image, because the softening process doesn't successfully reduce the weight, we restore the weight to the weight before softening.

```

C:\Users\stayed\Documents\project\project.py - Sublime Text [UNREGISTERED]
File Edit Selection Find View Goto Tools Project Preferences Help

#114  x = np.linspace(0, 1, 100)
#115  y = np.array(y_train[0:50,:])
#116
#117  model = TS(x, y, 50, model, ini_hidden, ini_reg)

soft starts
[array([[ 0.6551119,  0.4984872,  0.4984872,  0.4984872 ],
        [ 0.09545052, -0.4628568, -0.4628568, -0.4628568 ],
        [ 0.02118529,  0.3450561,  0.3450561,  0.3450561 ],
        [ 0.34802292,  0.1678315,  0.1678315,  0.1678315 ],
        [ 0.99883555,  0.37587473,  0.37587473,  0.37587473 ],
        [ 0.33855994,  0.3385552,  0.3385552,  0.3385552 ],
        [ 0.82845074,  0.28320444,  0.28320444,  0.28320444 ],
        [ 0.49274734, -0.23241463, -0.23241463, -0.23241463 ]],
      dtype=float32), array([ 0.12430776, -0.09920378, -0.09930378, -0.09940378], dtype=float32)]
initial soft not satisfy
soft over
[array([[ 0.6551119,  0.4984872,  0.4984872,  0.4984872 ],
        [ 0.09545052, -0.4628568, -0.4628568, -0.4628568 ],
        [ 0.02118529,  0.3450561,  0.3450561,  0.3450561 ],
        [ 0.34802292,  0.1678315,  0.1678315,  0.1678315 ],
        [ 0.99883555,  0.37587473,  0.37587473,  0.37587473 ],
        [ 0.33855994,  0.3385552,  0.3385552,  0.3385552 ],
        [ 0.82845074,  0.28320444,  0.28320444,  0.28320444 ],
        [ 0.49274734, -0.23241463, -0.23241463, -0.23241463 ]],
      dtype=float32), array([ 0.12430776, -0.09920378, -0.09930378, -0.09940378], dtype=float32)]
ini soft end
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
inte start
  
```

As you can see in the following image, we print out the combinations of the reduced different hidden nodes.





```

•523 #print('before cram')
•524 #print(LIS_sample)
•525 #print(class0)
•526 #print(class1)
•527 #
•528 #
•529 #print('weight before cram')
•530 #print(model.layers[0].get_weights())
•531 #
•532 #before cram pre=model.predict(LIS_X)
•533 #print('before cram pre')
•534 #
•535 #Hinode_HINode-1

```

```

weight tune waits
22
weight tune waits
23
weight tune waits
24
weight tune waits
25
weight tune waits
26
undesired attractor
still no
precision
0.5656565656565657
[Finished in 582.4s]

```

As you can see in the image below, a lot of hidden nodes are added, so it needs more time to test in the integrating process.

```

•824
•825 model.LIS(x_50,y_50,model,ini_Hinode,ini_Reg)
•826 #LIS(Xtrain nor,Y train,model,ini_Hinode,ini_Reg)
•827 model.save_weights('my_model_weights.h5')
•828
•829 x_test=np.array(Xtrain nor[51:,:]).reshape(-1,0)
•830 y_test=np.array(Y train[51:,:]).reshape(-1,1)
•831 p_test=model.predict(x_test).reshape(-1,1)
•832
•833 p_test_trans=np.where(p_test>0.5,1,0)
•834 p_test_trans=p_test_trans.reshape(-1,1)
•835
•836 diff=np.absolute(p_test_trans-y_test).reshape(-1,1)

```

```

0.40944305, 0.40944305, 0.40944305, -0.6601472, -0.6601472,
-0.6601472 ], dtype=float32), array([ 0.12430776, 0.65718466, 0.65708464, 0.6569846, -0.28518912,
-0.2851891, -0.28518913, -0.20466235, -0.20476235, -0.20486236,
0.10009964, 0.09999964, 0.09989963, 0.21247786, 0.21237788,
0.21227787, -0.00694886, -0.00704886, -0.00714886, 0.87791234,
0.8778123, 0.8777123, 0.15819001, 0.15809001, 0.15799001,
0.44086856, 0.44076854, 0.44066855, 0.5557202, 0.55562025,
0.55552024, 0.3242295, 0.32412952, 0.3240295, 0.07114402,
0.07104402, 0.07094403, -0.3785015, -0.37860152, -0.3787015,
-0.66598946, -0.6660895, -0.6661895, 0.16799164, 0.16789164,
0.16779163], dtype=float32))
ini soft end
[(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,

```

## 5. Discussion / Comparison

### Advantage of the model:

- It enables us to tune the weight based on each sample.
- It tunes the learning rate and momentum rate during the process of training dynamically.
- The model lets us increase and decrease hidden nodes of the layer based on each circumstances
- It can deal with outliers or special cases in the training process.

### Concern:

- If the original features are too many, you need to preprocess the data and select the best features to train. Otherwise, it may need more computing resources to complete the experiment.
- It is hard to set the benchmark of  $f(x)$  to classify the prediction. It may not be 0.5 all the time. You may need the domain knowledge of the data.
- Three nodes will be added in Cramming. If we need to delete more than 2 hidden nodes, we need to consider multiple possibilities, which may require a lot of computing resources.
- The quality or the sequence of the training data may largely affect the training process. If you encounter many special cases at the beginning, you may have to add many hidden nodes, while the model may not need that many nodes if you reverse the sequence of the dataset.

## 6. Conclusions

The model lets us train the sample one by one to adjust the weight of the model more subtly. We can flexibly increase and decrease hidden nodes of the weight during the process of the training. Hence, the model could be more flexible.

In the experiment, we don't see a great result during the process in terms of the amount of hidden nodes even though we get good results for the classification of training data.

In the future, we will examine if we can modify our experiment process or our preprocessing procedure to make the model better.

## 7. Reference

- S. Knerr, L. Personnaz and G. Dreyfus, "Single-layer learning revisited: a stepwise procedure for building and training a neural network", *Neurocomputing*. Heidelberg : Springer Berlin Heidelberg, 1990.
- Guang-Bin Huang, Yan-Qiu Chen and H. A. Babri, "Classification ability of single hidden layer feedforward neural networks," in *IEEE Transactions on Neural Networks*, vol. 11, no. 3, pp. 799-801, May 2000.
- Knerr, Stefan, Léon Personnaz, and Gérard Dreyfus. "Single-layer learning revisited: a stepwise procedure for building and training a neural network." *Neurocomputing*. Springer, Berlin, Heidelberg, 1990. 41-50.
- Sun, Zhan-Li, et al. "Sales forecasting using extreme learning machine with applications in fashion retailing." *Decision Support Systems* 46.1 (2008): 411-419.
- <https://www.kaggle.com/uciml/pima-indians-diabetes-database>
- 國立政治大學 資訊管理學系 蔡瑞煌 特聘教授