Modelling Competition

Kiju Kim
Arizona State University
Tempe, USA
kkim196@asu.edu

Hsiaoping Ni
Arizona State University
Tempe, USA
hsiaopin@asu.edu

Vincent Bevilacqua
Arizona State University
Tempe, USA
vbevilac@asu.edu

***Abstract*— The goal of the competition is to establish and train 3-parameter, 15-parameter, and 75-parameter models against historical electric and gas utility production data to obtain the lowest mean squared error. Best mean squared error (MSE) achieved was 19.616 with 75-parameter model without regularization applied.**

I. Model Features

Inspection of the dataset graphed over time reveals a general structure that can be used to inform our model construction. Firstly, it is apparent that the overall trend of data increases in a somewhat parabolic fashion as time advances. Furthermore, it is notable that the dispersion of datapoints increases in the latter portion of the dataset.

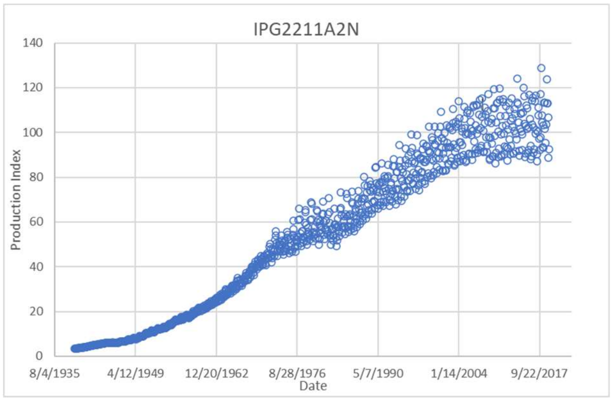
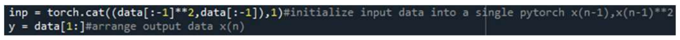


Figure 1 Raw Data Plotted Over Time

II. Data Preparation

Data was prepared for each network by scaling each input feature to normalize input values. This achieved more stable behavior during training and improved overall MSE in some cases. The scaling performed prior to forward propagation needed to be saved for testing and training.

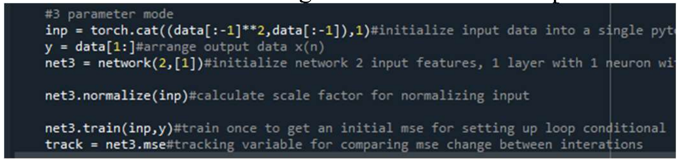
Input data was constructed by concatenating Pytorch tensors each consisting of a feature related to the network input. Example data preparation lines for 3-parameter model is shown below:



```
inp = torch.cat((data[:-1]**2,data[:-1]),1)#initialize input data into a single pytorch x(n-1),x(n-1)**2
y = data[1:]#arrange output data x(n)
```

Output data was prepared by removing 1 or 2 samples from the overall set as required (dependent on input features). Networks which took input features of two times steps, i.e. x(n-2), were afforded one less data sample in the training data. Due to the surplus of training data points, it was not deemed necessary to tr to restore the omitted sample via interpolation means.

Data was organized into Pytorch tensors such that each tensor row corresponded the all of the features needed for network activation. Because Pytorch operates on tensors, all samples are processed through the network simultaneously (in the form of different columns of the input matrix). This simplifies the error output calculation. Input features and outputs are passed as arguments as part of the training method for each network. See the figure below for an example:



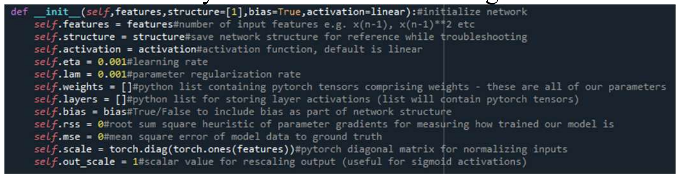
```
    #3 parameter mode
    inp = torch.cat((data[:-1]**2,data[:-1]),1)#initialize input data into a single pyt
    y = data[1:]#arrange output data x(n)
    net3 = network(2,[1])#initialize network 2 input features, 1 layer with 1 neuron wi

    net3.normalize(inp)#calculate scale factor for normalizing input

    net3.train(inp,y)#train once to get an initial mse for setting up loop conditional
    track = net3.mse#tracking variable for comparing mse change between interations
```

III. Model Construction

Each network is constructed as part of a class in python. The arguments to this class are:

- Features: Number of inputs (features) for activating the network
- Structure: Python list of integers defining the structure of the network. The number of entries in the list dictate the number of hidden layers and the value of each entry determines the width of each layer (number of neurons)
- Bias Boolean: Boolean to determine if a bias will be incorporated into the network structure. This is a binary choice and applies biases to all neurons if activated or none if not activates
- Activation function: Activation function applied to all neurons. Like the bias Boolean, this function is used for every neuron in the network.

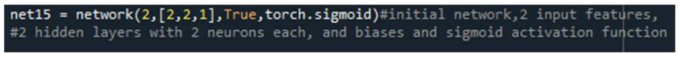
A detailed listed of class variables are enumerated in the comments in the Python code shown in the figure below:



```
def __init__(self,features,structure=[1],bias=True,activation=linear):#initialize network
    self.features = features#number of input features e.g. x(n-1), x(n-1)**2 etc
    self.structure = structure#save network structure for reference while troubleshooting
    self.activation = activation#activation function, default is linear
    self.eta = 0.001#learning rate
    self.lam = 0.001#parameter regularization rate
    self.weights = []#python list containing pytorch tensors comprising weights - these are all of our parameters
    self.layers = []#python list for storing layer activations (list will contain pytorch tensors)
    self.bias = bias#True/False to include bias as part of network structure
    self.rss = 0#root sum square heuristic of parameter gradients for measuring how trained our model is
    self.mse = 0#mean square error of model data to ground truth
    self.scale = torch.diag(torch.ones(features))#pytorch diagonal matrix for normalizing inputs
    self.out_scale = 1#scalar value for rescaling output (useful for sigmoid activations)
```

Figure 2 Class Variables and Arguments

Networks are constructed by calling the network class with the arguments enumerated above, see figure below for an example:



```
net15 = network(2,[2,2,1],True,torch.sigmoid)#initial network,2 input features,
#2 hidden layers with 2 neurons each, and biases and sigmoid activation function
```

Figure 3 15-Parameter Network Initialization

A. 3-Parameter Model

Figure 4 shows the structure of the 3-parameter network

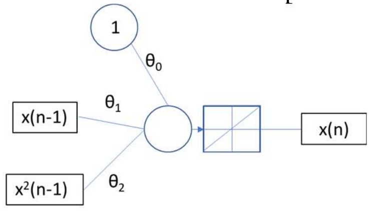


Figure 4. 3-Parameter Network Structure

The hypothesis for this network structure is as follows:

$$x(n-1) = \theta_2 x(n-1)^2 + \theta_1 x(n-1) + \theta_0$$

B. 15 Parameter Model

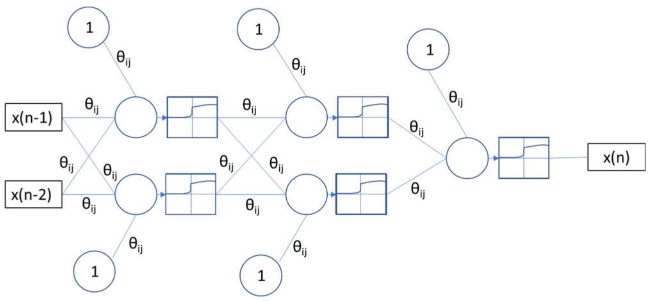


Figure 5 15-Parameter Network Structure

The hypothesis for this network structure:

$$x(n) = \phi(v_k)$$
$$v_k = \Sigma\big(\theta_{ik} y_{ij}\big) + \theta_{0k}$$
$$y_{ij} = \phi\big(v_{ij}\big)$$
$$v_{ij} = \Sigma\big(\theta_{ij} y_{ih}\big) + \theta_{0j}$$
$$y_{ih} = \phi(v_{ih})$$
$$v_{ih} = \Sigma\big(\theta_{ih} y_{ig}\big) + \theta_{0h}$$

C. 75-Parameter Model

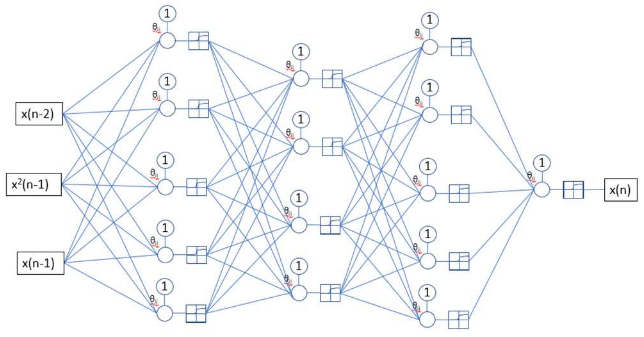


Figure 6 75-Parameter Network Structure

The hypothesis for this network structure is as follows:

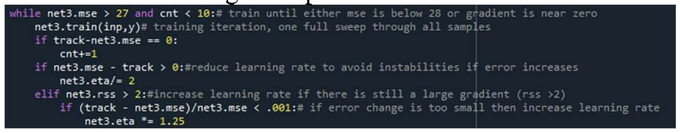
$$x(n) = \phi(v_k)$$
$$v_k = \Sigma\big(\theta_{ik} y_{ij}\big) + \theta_{0k}$$
$$y_{ij} = \phi\big(v_{ij}\big)$$
$$v_{ij} = \Sigma\big(\theta_{ij} y_{ih}\big) + \theta_{0j}$$
$$y_{ih} = \phi(v_{ih})$$
$$v_{ih} = \Sigma\big(\theta_{ih} y_{ig}\big) + \theta_{0h}$$
$$y_{ig} = \phi\big(v_{ig}\big)$$
$$v_{ig} = \Sigma\big[w_{ig} x(n-2) + w_{ig} x(n-1)^2 + w_{ig} x(n-1)\big] + w_{ig}$$

IV. Training Procedure

1) Hyper Parameters

a) Learning Rate

Learning rate for the 3 parameter network was initialized to 0.001 by default but as training progresses, the learning rate is scaled up or down to optimize training speed and stability. This is accomplished by halving the learning rate when the MSE error is seen to be increasing and increasing the learning rate by 25% when the change in MSE is less than 0.1% of the overall MSE. The below figure shows the training loop where learning rate updates are evaluated:



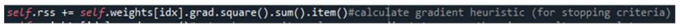
```
while net3.mse > 27 and cnt < 10:# train until either mse is below 28 or gradient is near zero
    net3.train(inp,y)# training iteration, one full sweep through all samples
    if track-net3.mse == 0:
        cnt+=1
    if net3.mse - track > 0:#reduce learning rate to avoid instabilities if error increases
        net3.eta/= 2
    elif net3.rss > 2:#increase learning rate if there is still a large gradient (rss >2)
        if (track - net3.mse)/net3.mse < .001:# if error change is too small then increase learning rate
            net3.eta *= 1.25
```

Figure 7 3-Parameter Training Loop and Learning rate update

This update is performed only if the gradient heuristic is calculated less than 2.

b) Gradient Heuristic

To improve training speed and stability, a gradient heuristic was calculated for measuring the proximity of the network to a local minimum. This heuristic is calculated by taking the sum of the squares of the partial derivatives of the parameters in the network. This is achieved conveniently through Pytorch's autograd functionality as Pytorch is automatically calculates gradients via its `backward()` method. The python code used for calculating the gradient heuristic is shown in the line below:



```
self.rss += self.weights[idx].grad.square().sum().item()#calculate gradient heuristic (for stopping criteria)
```

Figure 8 Gradient Heuristic

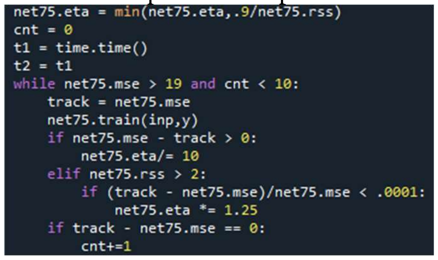
This gradient heuristic is used with learning rate updates (as described above).

c) Regularization Term

The 75-parameter network is trained both with and without parameter regularization. The regularization variable λ was initialized to 0.001. During training it was determined, that both model stability and speed were not affected largely by modulation of this hyperparameter. There the regularization term was kept at 0.001 for the final training attempt.

V. Stopping Criteria

By experimentation, it was noted which MSE each model could reasonably expect to achieve. This was done by running the training for each model and watching the point at which both the change in MSE became very small and the overall measure of the gradient heuristic was deemed very small (less than 1.0). With these metrics established we could set a goal for MSE for each model. Additionally, a counter was established to count how many times during training that the change in MSE was equal to zero. Training was run until either the minimum MSE for each mode was reached for the counter equaled or surpassed 10.



```
net75.eta = min(net75.eta,.9/net75.rss)
cnt = 0
t1 = time.time()
t2 = t1
while net75.mse > 19 and cnt < 10:
    track = net75.mse
    net75.train(inp,y)
    if net75.mse - track > 0:
        net75.eta/= 10
    elif net75.rss > 2:
        if (track - net75.mse)/net75.mse < .0001:
            net75.eta *= 1.25
    if track - net75.mse == 0:
        cnt+=1
```

Figure 9 Portion of Code that shows training condition and MSE tracking via a counter

A. 3-Parameter Nework

B. 15-Parameter Network

C. 75-Parameter Networks

VI. Results

Input space plots show the relationship between x(n-1) and x(n) and the relative performance of the networks as pertinent to the data.

A. 3-Parameter Network

The final MSE for the 3-parameter network was 28.43. This was the worst performing network as expected.

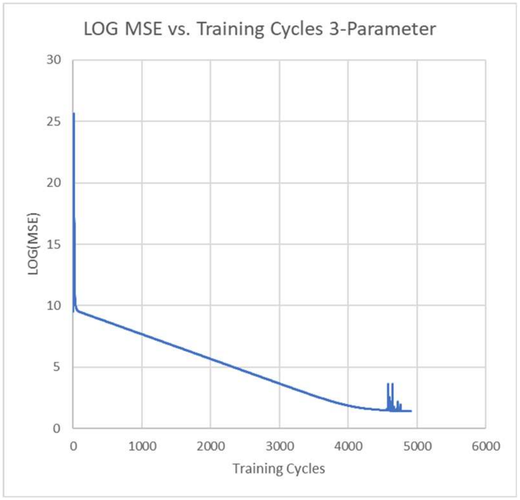


Figure 10 MSE of Network Per Training Iteration

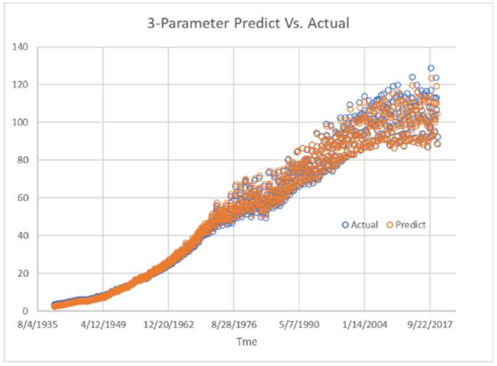


Figure 11 Predict vs. Actual Plotted Over Time

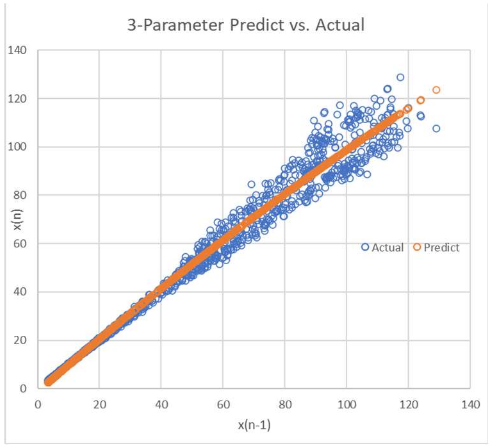


Figure 12 Predict vs. Actual in Input Space

B. 15-Parameter Network

Final MSE for the 15-parameter network was 21.245.

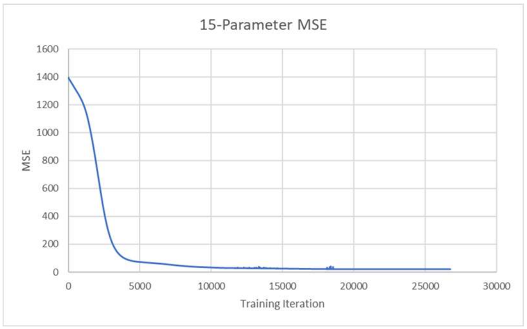


Figure 13 MSE of Network Per Training Iteration

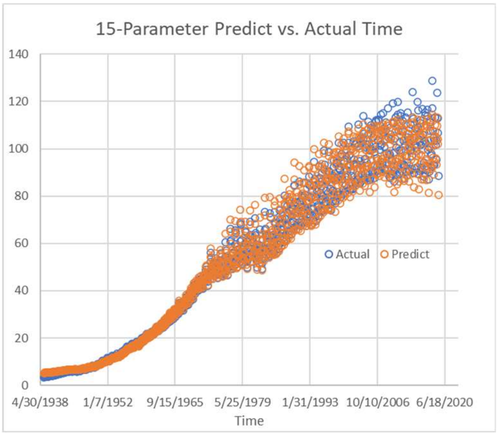


Figure 14 Predict vs. Actual Plotted over Time

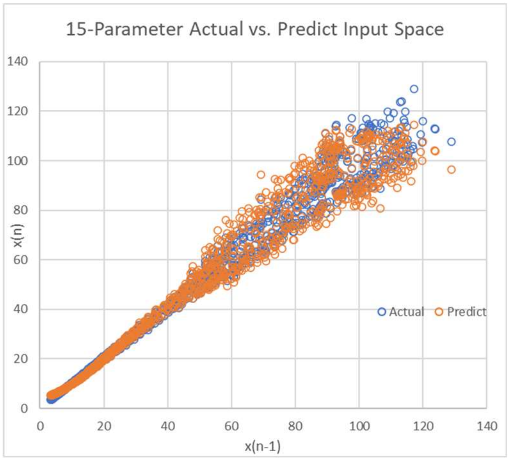


Figure 15 Predict vs Actual in Input Space

C. *75-Parameter Network*

Final MSE for the 75-parameter network was 19.616. This was the best performance of all models examined. The 75-parameter network with regularization performed very slightly worse with and MSE of 19.623.

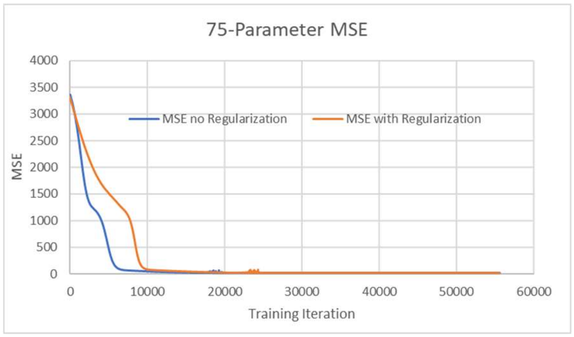


Figure 16 MSE of Each Network Per Training Iteration

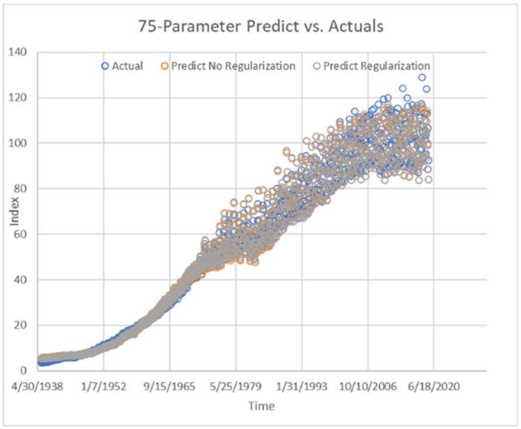


Figure 17 Predict Vs. Actual Plotted Over Time

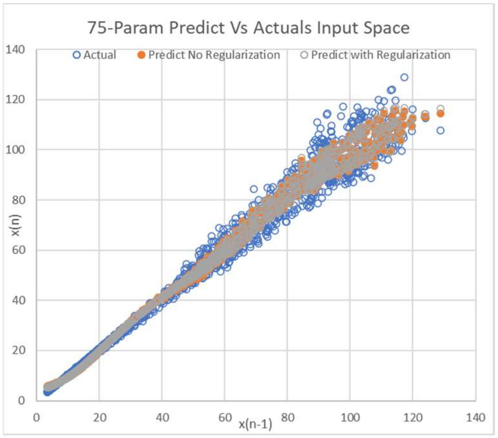
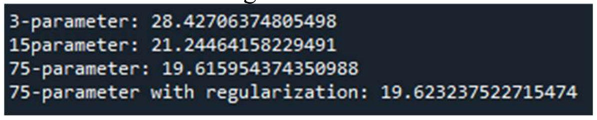


Figure 18 Predict vs Actual in Input Space

A summary of all performance can be seen in the python output displaying the MSEs of each network as shown in the figure below:



```
3-parameter: 28.42706374805498
15parameter: 21.24464158229491
75-parameter: 19.615954374350988
75-parameter with regularization: 19.623237522715474
```

Figure 19 Network MSEs

References

[1] S. Torabian, "Industrial Production: Electric and gas utilities", Kaggle, 2021. Available: https://www.kaggle.com/sadeght/industrial-production-electric-and-gas-utilities

Credit Report:

Vincent Bevilacqua: 33% - Organize and gather all the information into the paper and instruct the model for the team.
Hsiaoping Ni: 33% - Analyze the mathematic structure in the neural network and help with the model code.
Kiju Kim: 33% - Offer help to the model and provide the aspects of possible alternatives for the sound modelling.