`

11.1 ANN: An Introduction

Artificial neural networks (ANNs) are nonlinear empirical models which can capture complex relationships between input-output variables via supervised learning or recognize data patterns via unsupervised learning. Architecturally, ANNs were inspired by human brain and are a complex network of interconnected neurons as shown in Figure 11.1. An ANN consists of an input layer, a series of hidden layers, and an output layer. The basic unit of the network, neuron, accepts a vector of inputs from the source input layer or the previous layer of the network, takes a weighted sum of the inputs, and then performs a nonlinear transformation to produce a single real-valued output. Each hidden layer can contain any number of neurons. Infact, an ANN with just 1 hidden layer has the universal approximation property, i.e., it can approximate any continuous function with any given accuracy.

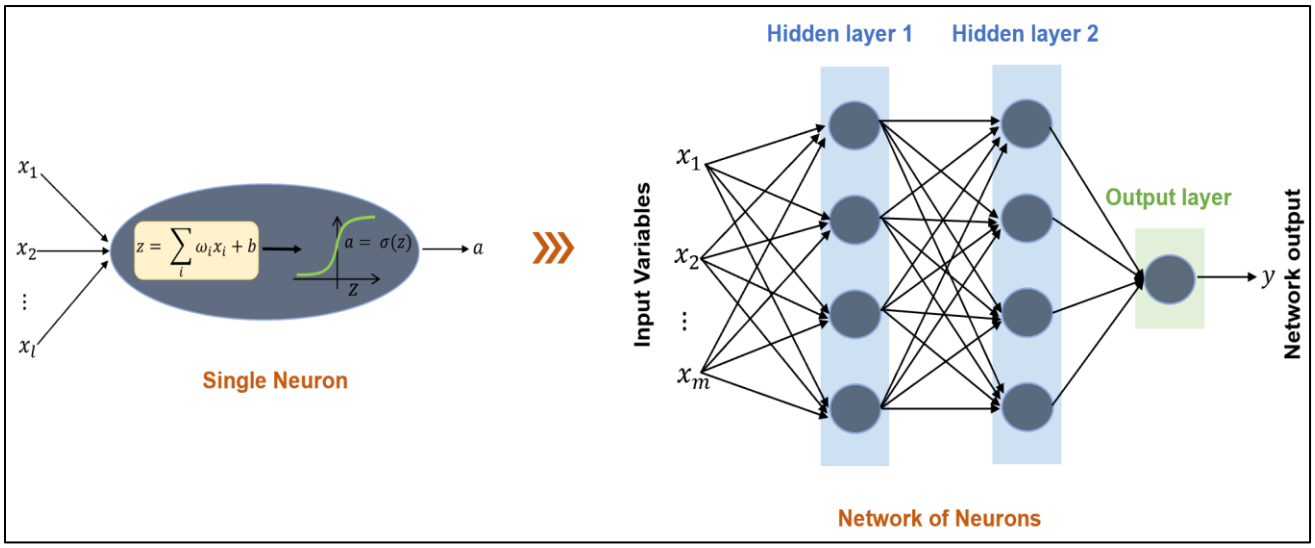


Figure 11.1: Architecture of a single neuron and feedforward neural network with 2 hidden layers for a MISO (multiple input, single output) system

The network shown in Figure 11.1 is an example of fully-connected feed-forward neural networks(FFNN), the most common type of ANN. In FFNN, signals flow in only one direction, from the input layer to the output layer via hidden layers, as opposed to the RNNs which contain feedback loops. In fully-connected networks, neurons between consecutive layers are connected fully pairwise and neurons within a layer are not connected.

While there are different types of ANN architectures (and ML researchers keep coming up with new ones!), FFNNs, RNNs, and CNNs are the most common architectures. While we will study RNNs in the next chapter, in this chapter we will focus on FFNNs. CNNs are primarily employed for image processing and are not covered in this book.

`

Deep learning

The recent popularity of ANNs can be attributed to the successes with deep learning. Deep learning simply refers to machine learning with deep neural networks (DNNs) which are ANNs with two or more hidden layers (see Figure 11.2). While shallow networks, with large number of neurons, can theoretically model any complex function, deep networks need much fewer model parameters and hence, are theoretically faster to train. Moreover, DNNs enable bypassing manual feature engineering; the first few hidden layers implicitly generate features that are utilized by downstream layers.

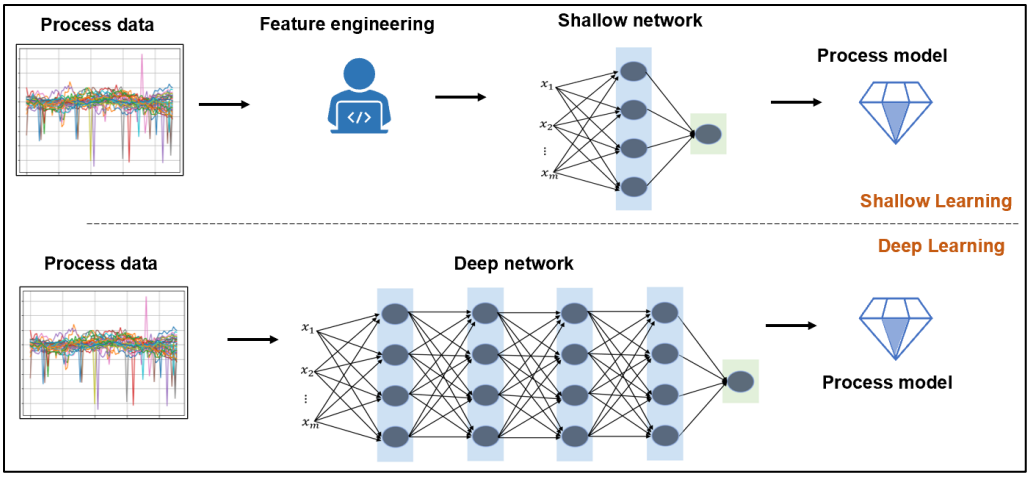


Figure 11.2: Shallow vs deep learning

Although DNNs seem superior to shallow networks, until recently, there weren't many applications of DNNs. DNNs suffered from the issue of vanishing and exploding gradients during model training. However, in recent years, several technical innovations have produced better nonlinear transformers (activation functions), initialization and regularization schemes, and, learning/optimization algorithms. These innovations have largely overcome the model training issues for DNNs and combined with the availability of cheap computing power and large amount of data have resulted in the AI revolutions that we are witnessing today.

A few words of caution: The power of DNNs should not make you undermine the importance of feature engineering. Explicit feature engineering contextualizes process data by considering the specific characteristics of the underlying process system. For example, for a time-series vibration data, feature engineering via frequency-domain transformation and binning can greatly help subsequent network training; without this, a DNN would require substantially large amount of training data to implicitly learn the features.

`

TensorFlow

For classical ML implementations, we utilized the packages available in Scikit-learn. For ANNs, other specialized libraries are commonly used which make it very easy to build neural net models. TensorFlow (by Google) and PyTorch (by Facebook) are the two most popular deep learning frameworks. These frameworks provide specialized algorithms for efficient training of ANNs.

Direct application of TensorFlow (TF) is not very straightforward and involves a steep learning curve. Modelers often utilized Keras, a high-level API built on top of TF, for ANN modeling. However, the latest version of TF released by Google in 2019 has integrated Keras directly into TF. This has made it possible to define, train, and evaluate ANN models in just a few lines of code. We will use TensorFlow Keras API in this book. The next section shows a quick application for modeling a combined cycle power plant to illustrate the ease of usage of these deep learning libraries.

Neural Nets for Process Systems: Brief History

The deep learning revolution has been a shot in the arm of the ML researchers in PSE community who are now exploring 'daring' applications of ANNs such as replacing model predictive controllers (MPCs) with ANNs by 'learning' MPC's optimal policies[+]. However, neural nets are not new for PSE community. Pre-deep learning era saw several interesting ANN applications for process systems as well. Venkatasubramanian[++] and Himmelbalu[+++] provide excellent accounts on the history of ANNs in PSE.

The 90s was a relatively high-activity period which witnessed applications of ANNs for fault detection, soft-sensing, data reconciliation[*], etc. This was followed by a decline in the interest in ANNs due to computational limitations and lack of big data. Today, these limitations having largely been overcome, ANNs are poised to make significant impact in PSE.

[+]Kumar et. al., Industrial, large-scale model predictive control with structured neural networks, Computers and Chemical Engineering, 2021

[++]Venkat Venkatasubramanian, The promise of artificial intelligence in chemical engineering: is it here, finally? AIChE Journal, 2018

[+++]David Himmelblau, Accounts of experiences in the application of artificial neural networks in chemical engineering, Industrial Engineering & Chemistry Research, 2008

[*]Karjarla, Dynamic data rectification via recurrent neural networks, Ph.D. Thesis, The University of Texas at Austin, 1995

`

11.2 Process Modeling via FFNN for Combined Cycle Power Plant (CCPP)

We will use data from a CCPP to illustrate the ease with which deep learning models can be built using TF Keras. The dataset (see Appendix for details) was collected over a period of 6 years and contains hourly average values of ambient temperature (AT), ambient pressure (AP), relative humidity (RH), and exhaust vacuum (V). These variables influence the net hourly electrical energy output (also provided in the dataset) of the plant operating at full load and will be the target variable in our ANN model. Let us first explore the dataset.

```
# import required packages and read data
import numpy as np, pandas as pd
import matplotlib.pyplot as plt

data = pd.read_excel('Folds5x2_pp.xlsx', usecols = 'A:E').values
X y = data[:,0:4], data[:,4][:,np.newaxis]

# plot input vs output for each input
plt.figure(), plt.plot(X[:,0], y, '*'), plt.title('AT vs EP')
plt.figure(), plt.plot(X[:,1], y, '*'), plt.title('V vs EP')
plt.figure(), plt.plot(X[:,2], y, '*'), plt.title('AP vs EP')
plt.figure(), plt.plot(X[:,3], y, '*'), plt.title('RH vs EP')
```

Figure 11.3 clearly indicates the impact of input variables on the electrical power (EP).

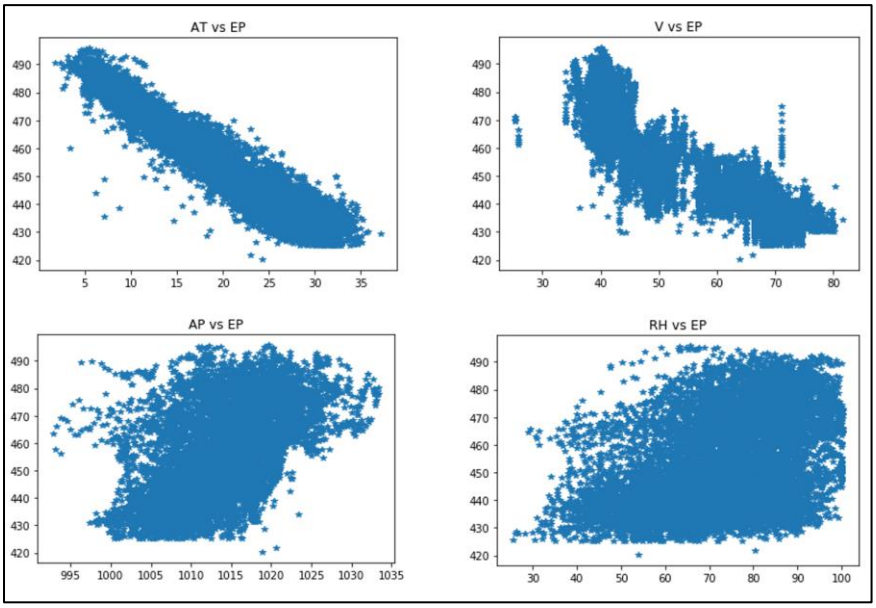


Figure 11.3: Plots of influencing variables (on x-axis) vs Electrical Power (on y-axis)

`

There is also a hint of nonlinear relationship between exhaust vacuum and power. While it may seem that AP and RH do not influence power strongly, it is a known fact that power increases with increasing AP and RH individually[59]. Let us now build a FFNN model with 2 hidden layers to predict power. We first split the dataset into training and test data, and then scale the variables.

```
# separate train and test data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)

# scale data
from sklearn.preprocessing import StandardScaler
X_scaler = StandardScaler()
X_train_scaled = X_scaler.fit_transform(X_train)
X_test_scaled = X_scaler.transform(X_test)

y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train)
y_test_scaled = y_scaler.transform(y_test)
```

To build FFNN model, we will import relevant Keras libraries and add different layers of the network sequentially. The Dense library is used to define a layer that is fully-connected to the previous layer.

```
# import Keras libraries
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# define model
model = Sequential()
model.add(Dense(8, activation='relu', kernel_initializer='he_normal', input_shape=(4,)))
# 8 neurons in 1st hidden layer
model.add(Dense(5, activation='relu', kernel_initializer='he_normal'))
# 5 neurons in 2nd layer
model.add(Dense(1))
# 1 neuron in output layer
```

The above 4-line code completely define the structure of the FFNN. Do not worry about the activation and kernel_initializer parameters; we will study them in more detail in later sections.

[59] Pinar Tufekci, Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods, Electrical Power and Energy Systems, 2014

`

Next, we will compile and fit the model. Again, do not worry about the optimizer, epochs, and batch_size parameters right now.

```
# compile model
model.compile(loss='mse', optimizer='Adam') # mean-squared error is to be minimized

# fit model
model.fit(X_train_scaled, y_train_scaled, epochs=25, batch_size=50)

# predict y_test
y_test_scaled_pred = model.predict(X_test_scaled)
y_test_pred = y_scaler.inverse_transform(y_test_scaled_pred)
```

The above lines are all it takes to a build FFNN and make predictions. Quite convenient, isn't it? Figure 11.4 compares actual vs predicted power for the test data.

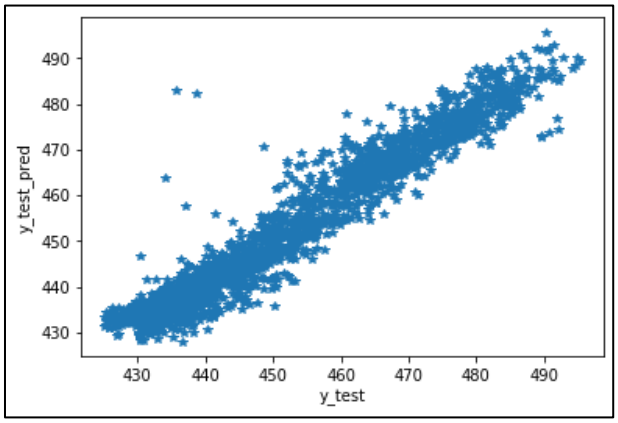


Figure 11.4: Actual vs predicted target for CCPP dataset (obtained R^2 = 0.93)

You can use the model.summary command to visualize the structure of the network and check the number of model parameters as shown in Figure 11.5. For the relatively simple CCPP dataset, we can obtain a reasonable model with just 1 hidden layer with 2 neurons. Nonetheless, this example has now familiarized us with the process of creating a DNN.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 40
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 45
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 6
=================================================================
Total params: 91
Trainable params: 91
Non-trainable params: 0
_________________________________________________________________
```

Figure 11.5: Structure of FFNN model with number of model parameters

`

11.3 Mathematical Background

Let's consider a sample input $\boldsymbol{x} \in R^4$ from the CCPP dataset and trace its path as it moves through the layers of the network to generate the output. In this forward pass (also called forward propagation), the inputs are first processed by the neurons of the first hidden layer. In the j*th* neuron of this layer, the weighted sum of the inputs are non-linearly transformed via an activation function, *g*

$$a_j = g(\boldsymbol{w}_j^T \boldsymbol{x} + b_j)$$

$\boldsymbol{w_j} \in R^4$ are the weights applied to the inputs and b_j is the bias added to the sum. Thus, each neuron has 5 parameters (4 weights and a bias) leading to 40 parameters for all the 8 neurons of the 1st layer that need to be estimated. Outputs of all the 8 neurons form vector $\boldsymbol{a}^{(1)} \in R^8$

$$\boldsymbol{a}^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(\mathbf{1})})$$

where each row of $\boldsymbol{W}^{(1)} \in R^{8\times4}$ contains the weights of a neuron. The same activation function is used by all the neurons of a layer. $\boldsymbol{a}^{(1)}$ becomes the input to the 2nd hidden layer.

$$\boldsymbol{a}^{(2)} \in R^5 = g^{(2)}(\boldsymbol{W}^{(2)}\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(\mathbf{2})})$$

where $\boldsymbol{W}^{(2)} \in R^{5\times8}$. Each neuron in the 2nd layer has 8 weights and a bias parameter, leading to 45 parameters in the layer. The final output layer had a single neuron with 6 parameters and no activation function, giving the network output as follows

$$\boldsymbol{a}^{(3)} = \hat{y} = (\boldsymbol{w}^{(3)})^T\boldsymbol{a}^{(2)} + b^{(\mathbf{3})}$$

where $\boldsymbol{w}^{(3)} \in R^5$.

Activation functions

Activation functions (AFs) are what impart non-linear capabilities to neural nets. We used ReLU activation function for CCPP modeling. While ReLU (rectified linear unit) is the preferred AF now-a-days, pre-deep learning era used to employ sigmoid activation function. Illustration below shows the form of sigmoid AF, which transforms pre-activation, z (weighted sum of inputs plus bias), into activation a $\in (0, 1)$. The saturation of sigmoid functions at 1 (or 0) for large positive (or negative) value of pre-activation results in the issue of vanishing/zero gradients for deep networks and is the reason why sigmoid AFs are not favored anymore for hidden layers.

`

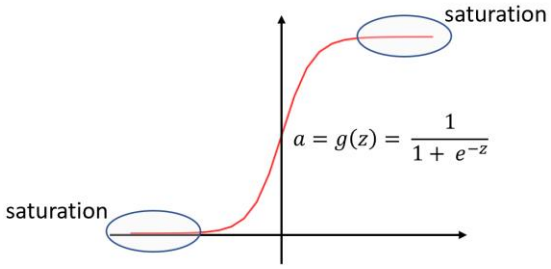
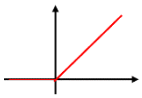
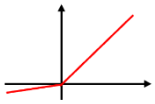
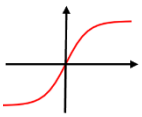


Table below lists other commonly employed activation functions. We will study in a later section how these functions impart non-linearity to ANN models.

Function	Equation	Graph	Remarks
ReLU	$g(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$		• works very well and preferred AF • computationally cheap • "dead" neurons for z <0 (not a big issue if 'learning-rates' are properly set)
Leaky ReLU	$g(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$		• alternative if 'dead' ReLU neurons cause issues • may not perform better than ReLU
Hyperbolic tangent (tanh)	$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$		• Range: (-1, 1) • prone to vanishing gradient issue and computationally expensive

Activation functions for output layer:

The choice of activation function in the output layer depends on the task at hand. For regression problems, no activation function (equivalently, *g(z)* = *z* linear function) is used. For classification tasks where the classes are mutually exclusive, softmax function is employed. Figure 11.6 shows a network for 3 exclusive classes with softmax activation in the output layer.

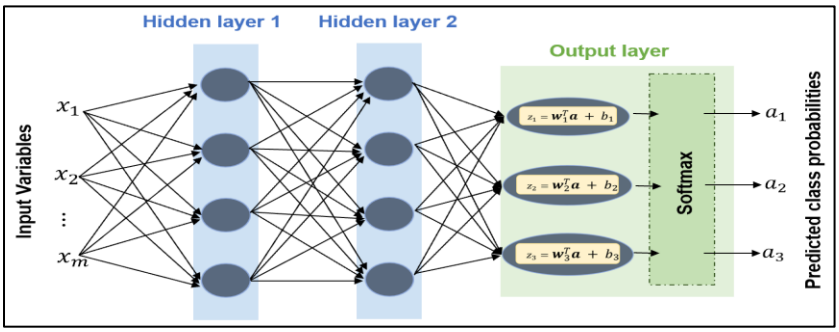


Figure 11.6: FFNN with softmax activation function. (Subscripts denoting layers are not shown for clarity)

`

Softmax is an exponential function that generates normalized activations so that they sum up to 1. In Figure 11.6, activation a_j ($j \in [1,2,3]$) is generated as follows

$$a_j = g_{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{3} e^{z_k}}$$

In ANN world, the pre-activations (z_j) that are fed as inputs to the softmax function as also called logits. The softmax activations lie between 0 and 1, and hence, they are interpreted as class-membership probabilities. The predicted class label is taken as the class with maximum probability (or activation)

$$\hat{y} = argmax\,(a_j, j \in [1,2,3])$$

As you will see in next subsection, the predicted class probabilities (and not the predicted class label) are directly used during model training. For classification tasks where the classes are not mutually exclusive, sigmoid function is employed in the neurons of the output layer. Sigmoid function is also used for binary mutually exclusive classification problems with a single neuron in the output layer.

Loss functions & cost functions

ANNs optimize their model parameters by minimizing the mismatch between measured and predicted values of the target variable(s). Loss function quantifies this mismatch for a single sample. The average of loss function over the samples of the training dataset that is minimized during model training is called cost function. For regression problems, the commonly employed loss function is mse (mean squared error)

$$MSE\ Loss = (y_{sample} - \hat{y}_{sample})^2$$

The corresponding cost function becomes

$$MSE\ Cost = \frac{1}{\#\ of\ samples} \times \sum_{samples} (y_{sample} - \hat{y}_{sample})^2$$

Note that the above expression is for a scalar output; for multi-dimensional output, summation is also carried over different dimensions. Other popular loss metric for regression is mae (mean absolute error) which measures the absolute difference $|y_{sample} - \hat{y}_{sample}|$. MAE cost is more robust to outliers compared to MSE as large errors are not heavily penalized through squaring. Mae metric, however, is less efficient due to the discontinuous nature. A good compromise can be achieved by rmse (root mean squared error) metric.

`

For binary-class classification problems, binary cross-entropy is the default loss function. Let *y* (can take value 0 and 1) be the true label for a data sample and *p* (or $\hat{\mathbf{y}}$) be the predicted probability (of *y = 1*) obtained from sigmoid output layer. The cross-entropy loss is given by

$$Cross-entropy\ Loss\ =\ -y*log(p)\ -\ (1-y)*log(1-p)\ =\ \begin{cases}-log(1-p), & y=0\\ -log(p), & y=1\end{cases}$$

The above expression is generalized as follows for a multiclass classification, where overall loss is sum of separate losses for each class label

$$Multi-class\ cross-entropy\ Loss\ =\ -\sum_{c=1}^{\#\ of\ classes} y_c log(p_c)$$

where, y_c and p_c are binary indicator and predicted probability of a sample belonging to class *c*, respectively. Note that for multi-class classification, the target variable will be in one-hot encoded form.

Gradient descent optimization

Once the cost/objective function ($J(\boldsymbol{\theta})$) has been defined, the model parameters ($\boldsymbol{\theta}$) are optimized to minimize the cost function. The most common optimization approach used by deep learning libraries is gradient descent where model parameters are estimated in multiple iterations, and in each iteration, parameters are updated in the direction of steepest descent (negative of the gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$). The expression below shows the update mechanism for the i^{th} model parameter

$$\boldsymbol{\theta}_{i,next} = \boldsymbol{\theta}_{i,current}\ -\ \eta\frac{\partial}{\partial\boldsymbol{\theta}_i}J(\boldsymbol{\theta})$$

where η (learning rate), a hyperparameter, determines the magnitude of update in each iteration. For illustration, Figure 11.7 shows the progression of cost function towards its minima through multiple iterations for simple systems (one and two model parameters).

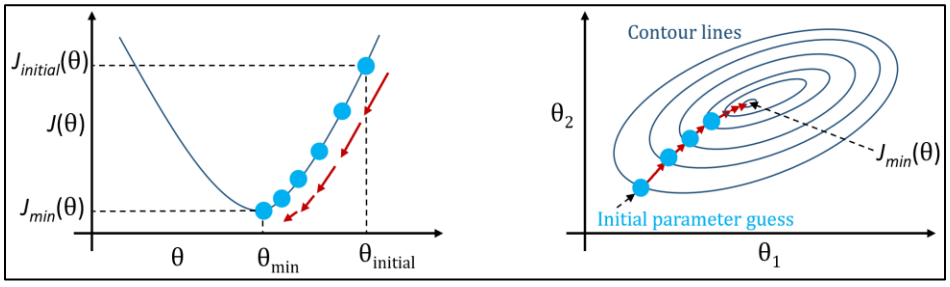


Figure 11.7: Gradient descent iterations for cost function minimization

`

The learning rate influences the speed of optimization convergence. Large learning rates during initial iterations help to quickly reach close to the minima point; however, large values can also lead divergence instead of convergence. On the other hand, too small learning rates leads to very slow convergence. To deal with this trade-off, learning rate is often adjusted during training via learning rate schedule (reducing η over iterations) or in-built mechanism in modern optimizers (more common approach now-a-days).

A drawback of the classical (or vanilla) gradient descent is that the optimizer is slow and often gets stuck in local minima. As a remedy, several variations of the vanilla approach have been devised which have been shown to provide better performance. These variations include Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and Adam optimization. Among these, Adam (adaptive moment estimation) optimization, which combines the ideas of momentum optimization and RMSProp, is recommended as the default optimizer to use.

Like with everything, Adam may not give good performance for some cases; in those cases, other optimizers should be tried. With Adam optimization you don't have to worry about learning rate scheduling as it's an adaptive learning rate algorithm, although the initial learning rate may need adjustments for some problems. Adam does have a few hyperparameters of its own but usually the default values work fine.

The superior performance of Adam optimization over gradient descent stems from several algorithmic innovations which includes, amongst others, using exponentially decaying average of past gradients to determine update direction. You are encouraged to check out this article[60] for an intuitive explanation of how these optimizers work.

Epochs & batch-size

In vanilla gradient descent, the entire training dataset is used for computing the gradient in each iteration. Due to this, for large datasets, each iteration can become very slow. This approach, aka batch gradient descent, is also not suitable for updating models online, i.e., updating models quickly with new data samples. Mini-batch gradient descent (MGD) and stochastic gradient descent (SGD) are two alternative approaches.

In mini-batch variation, for each parameter update iteration, only a subset of the training dataset goes into gradient computation. The number of samples in a subset is called batch-size. Batch sizes commonly range between 32 and 256. MGD results in faster convergence

[60] Sebastian Ruder, An overview of gradient descent optimization algorithms, arXiv, 2017

`

and noisier/oscillating gradients (different mini-batches can generate very different update directions), as shown in Figure 11.8. An advantage of noisier gradients is that it can help the optimizer to escape local minima. Stochastic gradient descent (not common now) is the extreme case of mini-batch variant where batch-size is just one! Although not technically correct, practitioners employ the term SGD also when mini-batches are used.

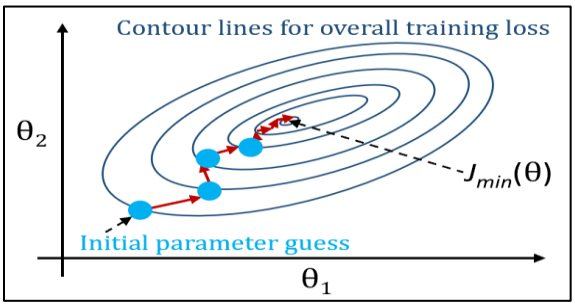


Figure 11.8: Mini-batch & stochastic gradient descents have noisier gradients compared to batch gradient descent

In MGD, several iterations (equal to $\frac{\#\text{ of training samples}}{batch\ size}$) are needed for the optimizer to go through the entire training dataset, also referred to as completion of one epoch. At the start of every epoch, training dataset is usually shuffled (to prevent optimizer getting stuck in cycles), divided into mini-batches, and then update iterations continue. Number of epochs is another crucial hyperparameter. Training and validation accuracies are plotted against epochs rather than iterations to keep track of model fitting. More epochs will lead to better training accuracy but can result in worse validation accuracy. We will learn more about it later. A simple thumb-rule is to use more epochs for higher batch sizes.

Backpropagation

Backpropagation computes the gradients of the cost function with respect to the model parameters efficiently through smart usage of chain rule of derivative computation. For conceptual understanding of the mechanism, let's focus on SGD (for ease of explanation) and assume that we are interested in computing the gradient with respect to ω_1 as highlighted in Figure 11.9.

`

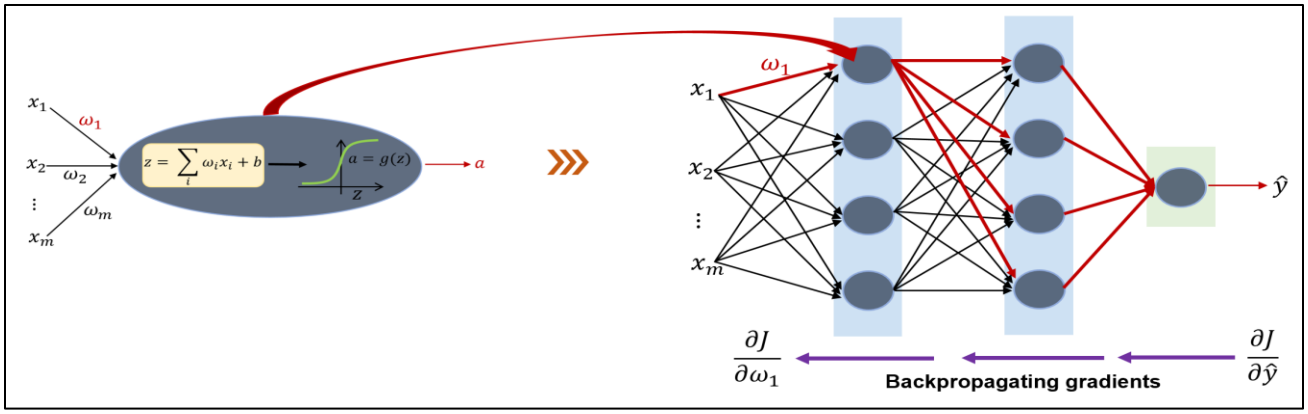


Figure 11.9: Illustration of backpropagation mechanism

Chain rule gives us the following expressions

$$\frac{\partial J}{\partial \omega_1} = \frac{\partial J}{\partial a} \times \frac{\partial a}{\partial \omega_1}$$
$$= \frac{\partial J}{\partial a} \times \left(\frac{\partial a}{\partial z} \times \frac{\partial z}{\partial \omega_1}\right)$$

While $\frac{\partial z}{\partial \omega_1}$ is simply x_1 and $\frac{\partial a}{\partial z}$ is derivative of the activation function, $\frac{\partial J}{\partial a}$ is the unknown element here. However, it is easy to show that this unknown part can be computed if partial derivatives of cost function with respect to the activations of the 2nd hidden layer are known, which themselves need the derivative of cost with respect to output layer activation or $\hat{y}$. This final missing element is given by

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} (y - \hat{y})^2 = \frac{1}{2}(y - \hat{y})$$

The above discussion shows that once the activations, predicted output, and error are computed by forward propagation, backward propagation is employed to compute required gradients one layer at a time starting from the last layer. For MGD, through careful bookkeeping, a compact vector-matrix form of backpropagation can be obtained just like we did for forward pass. Note that technically backpropagation only refers to the mechanism for gradient computation, but the term is often used to refer to the entire model fitting algorithm.

Vanishing/Exploding gradients

In the past, the issue of vanishing or exploding gradients posed great difficulty in model fitting for deep networks. In backpropagation, we saw that gradients are propagated from the output layer towards the inner layers. Some of these gradients were derivatives of activation functions, which, for sigmoid and tanh functions, become very small for high pre-activations.

`

In the case of vanishing gradient scenario, gradients of inner layer parameters become very small due to products of small gradients propagating from the outer layers. This causes virtual halt of inner parameter updates leading to slow convergence and poor model fitting. On the other hand, sometimes, the problem arises due to gradients becoming very large due to large weights. Due to backpropagation the problem gets compounded and causes training failure.

The paper by Glorot and Bengio (titled 'Understanding the difficulty of training deep feedforward neural networks') provides excellent investigation on these issues. The usage of sigmoid activation function and poor initialization schemes were found to be some of the culprits. Today, these issues have mostly been curtailed through usage of ReLU functions, batch normalization, correct initialization schemes, etc.

11.4 Non-linearity in Neural Nets (width vs depth)

While designing a neural network for complex nonlinear models, one of the most crucial decisions that you will make is to decide between adding more neurons in a layer (increasing network width) and adding more hidden layers (increasing network depth). Both network width and depth influence network performance, albeit in different manners. Before we delve further into this 'width vs depth' question, let us analyze the role of ReLU activation functions in some more details – this will provide some clues to help us make the above decision.

Consider an absolute function that takes a scalar input and outputs the absolute value. This nonlinear function can be perfectly modeled via a FFNN with 1 hidden layer containing 2 nodes with ReLU activation function as shown in Figure 11.10.

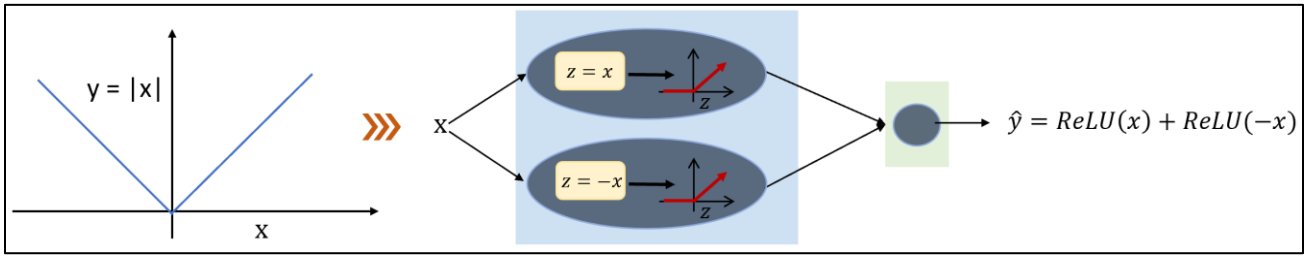


Figure 11.10: Modeling absolute function via FFNN

We were able to model the nonlinear absolute function perfectly because the ReLU-based neurons effectively bifurcated the input space into 2 regions (x <=0 and x>=0) where linear relationships hold; essentially, the absolute function got modeled via two piecewise linear functions. Continuing this logic, now assume that the scalar input-output system follows a

`

nonlinear (quadratic) relationship. This system can again be approximated by using several piecewise linear functions as shown in Figure 11.11. Infact, the approximation shown in the figure was obtained using 1 hidden layer with 5 ReLU neurons.

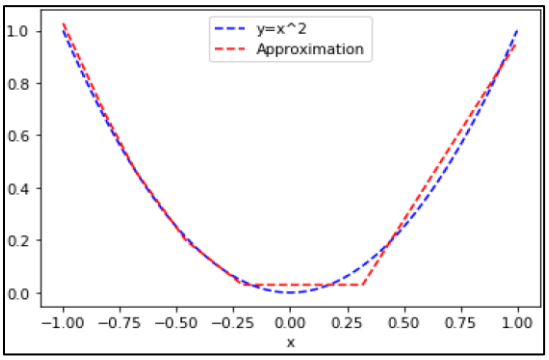


Figure 11.11: Approximation of quadratic function using piece-wise linear functions

What we learn from the above illustrations is that having several neurons in a single layer helps a neural network divide the input space into several local regions and approximate a complex function via several piecewise simpler functions. Consequently, a very nonlinear function can be modeled with a very wide network (large number of nodes in one hidden layer). However, training a very wide network can become problematic.

For a network of fixed width, another way to increase nonlinear capability is to add more depth/hidden layers to the network, enabling the network to estimate more complicated features. This is because the successive hidden layers build upon the nonlinear features generated from the inner layers to impart more nonlinearity to the overall operation. For any activation function, in a wide network with 1 hidden layer, activations of all neurons have similar degree of non-linearity. But, in a deep network, activations from different layers exhibit varied degrees of nonlinearity, with outermost layer exhibiting most nonlinearity. However, as mentioned before, deep networks are prone to vanishing/exploding gradient issues.

You can see that the answer to the ‘more width or more depth’ question is not very straightforward. Similar gain in performance can be gained by increasing network width or depth. A few words of caution here: higher network depth may fail to give adequate performance if network width is inadequately low. For example, irrespective of the network depth, absolute function cannot be accurately modeled if network width is just one. Therefore, both width and depth are important. A thumb-rule is that, for a system with low degree of nonlinearity, a high width-low depth network should be used, while for a very complex system high depth network with adequate width should be used.

`

For modeling process systems 2 to 3 hidden layers are usually sufficient for both regression and classification tasks. Going deeper helps if you are working with inputs which exhibit hierarchical structure such as images (complex shapes can be broken down into edges, corners, etc.).

11.5 Neural Net Hyperparameter Optimization

We have seen that before executing the model.fit command, several hyperparameters have to be specified. These include, amongst others, network depth, network width, number of epochs, minibatch size, learning rates, regularization penalty, activation function type. While general recommendations exist for most of the hyperparameters, specifying the number of hidden layers and the number of neurons in every layer requires some work as these depend largely on the specific problem at hand.

Common approaches for tuning the hyperparameters are based on manual trials and automated grid (or random) searches. In manual approach there are two strategies. You can choose a complex network structure (more layers and neurons than you 'feel' necessary) and use regularization techniques and early stopping (these are discussed in next section) to avoid overfitting. Alternatively, you can start with a simple structure (one hidden layer and reasonable number of neurons) and gradually increase the network depth and number of neurons until the overfitting begins.

The automated approach uses GridSearchCV from sklearn. A KerasRegressor wrapper is utilized to enable passing Keras model to GridSearchCV. The code below shows the implementation details where we search for the best specifications for network depth and width, regularization penalty, and learning rate. You will notice that we divide the training data into estimation and validation set to pass to GridSearchCV for cross-validation. Also, the FFNN model is defined via a function which we pass to KerasRegressor wrapper. The best model obtained uses one hidden layer with nine neurons. A R^2 value of 0.931 is obtained.

```
# separate training, validation, test data
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)
X_est, X_val, y_est, y_val = train_test_split(X_train, y_train, test_size = 0.3, random_state = 100)

# scale data
from sklearn.preprocessing import StandardScaler
```

`

```
X_scaler = StandardScaler()
X_est_scaled = X_scaler.fit_transform(X_est)
X_val_scaled = X_scaler.transform(X_val)
X_test_scaled = X_scaler.transform(X_test)

y_scaler = StandardScaler()
y_est_scaled = y_scaler.fit_transform(y_est)
y_val_scaled = y_scaler.transform(y_val)
y_test_scaled = y_scaler.transform(y_test)

# import packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers
from tensorflow.keras.optimizers import Adam

# model function
def FFNN_model(hidden_layers, layer_size, regularizationValue, learningRate):
    model = Sequential()
    model.add(Dense(layer_size, kernel_regularizer=regularizers.L1(regularizationValue),
                    activation='relu', kernel_initializer='he_normal', input_shape=(4,)))

    for _ in range(hidden_layers-1):
        model.add(Dense(layer_size, kernel_regularizer=regularizers.L1(regularizationValue),
                        activation='relu', kernel_initializer='he_normal'))

    model.add(Dense(1))
    model.compile(loss='mse', optimizer=Adam(learning_rate=learningRate))

    return model

# KerasRegressor
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
model = KerasRegressor(build_fn=FFNN_model, epochs=25, batch_size=50)

# gridSearchCV
from sklearn.model_selection import GridSearchCV

param_grid={
    "hidden_layers":[1, 2],
    "layer_size":np.arange(1,10),
    "regularizationValue": [0.001, 0.01, 0.1],
    "learningRate":[0.05, 0.01, 0.1]}

grid_searchcv = GridSearchCV(model, param_grid)
```

`

```
grid_searchcv.fit(X_est_scaled, y_est_scaled, validation_data=(X_val_scaled, y_val_scaled))

print("The best parameters obtained are:", grid_searchcv.best_params_)

>>> The best parameters obtained are: {'hidden_layers': 1, 'layer_size': 9, 'learningRate': 0.01,
'regularizationValue': 0.001}

# best model
model = grid_searchcv.best_estimator_.model
```

11.6 Strategies for Improved Network Training

Training ANN models is not easy and there are various issues you may encounter. Maybe your model is overfitting or underfitting, or you are getting stuck in local minima, or your model is not converging. Keeping track of the training process can help to obtain some glimpse into how well the model is fitting and provide hints about potential hyperparameter setting adjustments. One way to keep track of how the model fitting is progressing is to draw plot of training set and validation set costs w.r.t the epochs. Several strategies have been devised to deal with ANN training issues that are frequently encountered. We will study these strategies in this section.

Early stopping

As optimizer iterates for more epochs, training cost goes down; however, the validation cost may begin to increase. This is an indication of model overfitting. Figure 11.12 shows an example (generated with the Kamyr digester dataset). To prevent this, early stopping is adopted, where optimizer aborts training when validation cost begins to increase. To implement this, validation set and early stopping specifications are supplied as shown in the code below. Note the usage of the ‘history’ object which holds record of losses during training for drawing validation plot.

```
# define model
def FFNN_model():
    model = Sequential()
    model.add(Dense(20, activation='tanh', kernel_initializer='he_normal', input_shape=(19,)))
    model.add(Dense(5, activation='tanh', kernel_initializer='he_normal'))
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='Adam')
    return model
```

`

```
# fit model with early stopping
from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', patience=15)

history = FFNN_model().fit(X_est_scaled, y_est_scaled, epochs=250, batch_size=32,
validation_data=(X_val_scaled, y_val_scaled), callbacks=es)

# validation plot
plt.figure()
plt.title('Validation Curves')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.plot(history.history['loss'], label='training')
plt.plot(history.history['val_loss'], label='validation')
plt.legend()
```

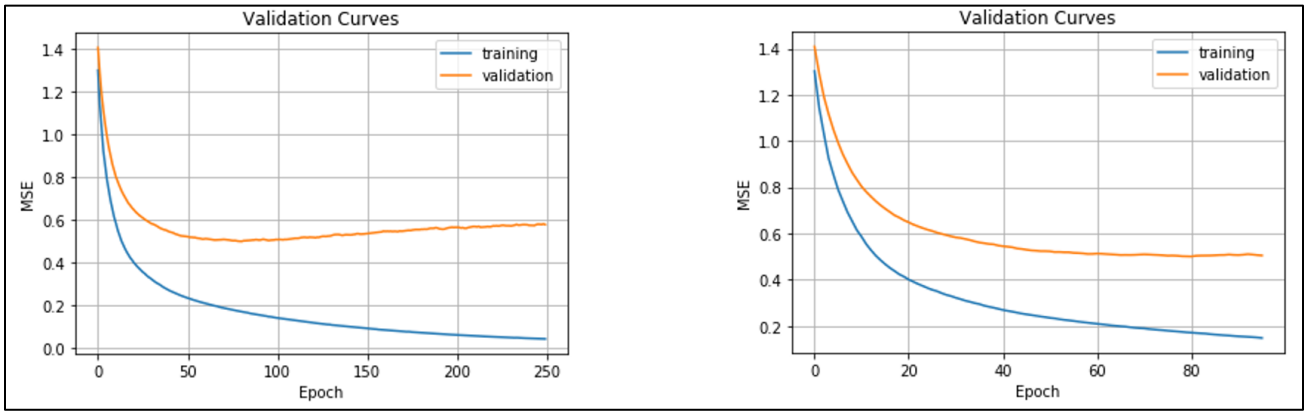


Figure 11.12: Validation plots for Kamyr digester dataset without (left) and with (right) early stopping

With early stopping, we can specify the number of epochs to a large value and not worry about overfitting as the training will automatically stop at the right time. The early stopping callback has several parameters which can be used to alter its behavior. For example, the 'patience' parameter that we used specifies the number of epochs to allow with no improvement on validation set after which the training is stopped. You should check out the official Tensorflow documentation[61] for details on other early stopping parameters.

[61] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

`

Regularization

As we have learnt before, regularization is a way to constraint model's weight parameters to avoid overfitting (large gap between training and validation accuracies). As one of the best practices, you should always employ regularization for your ANN model fitting. Accordingly, the regularization penalty becomes one of the important hyperparameter. Keras makes specifying regularization (L1, L2, or both) very easy as was shown in the hyperparameter optimization example where L1 regularization was used while defining the Dense layers. Keras allows three types of regularizers, namely, kernel_regularizer, bias_regularizer, activity_regularizer, to regularize the weight parameters, bias parameters, and neuron activations; of these kernel_regularizer is the recommended default choice.

For deep networks, another form of regularization is very popular and is called 'dropout'. In this strategy, in every training iteration, some neurons are 'dropped out', i.e., their activations are not computed in forward pass and weight updates are not performed. Note that dropped neurons in current iteration can become active in the next iteration. Dropout has been shown to provide superior generalization[62] as it forces the network to avoid relying too much on just a few neurons. In Keras, dropout is specified layer-wise. Two schemes are illustrated below. In scheme 1, dropout layer is added between the input and first hidden layer and dropout rate (fraction of neurons to drop) is set to 0.2; therefore, one out of every 5 input variables are randomly excluded in an iteration. In scheme 2, dropout layer is added between two hidden layers; here again, one out of every 5 neurons in the first hidden layer are randomly excluded in an iteration. Note that dropout is only done during model training.

Dropout on input layer	Dropout on hidden layer
model = Sequential() model.add(Dropout(0.2, input_shape=(19,)) model.add(Dense(20),…) model.add(Dense(1))	model = Sequential() model.add(Dense(20),…) model.add(Dropout(0.2)) model.add(Dense(1))

Initialization

Before the first optimization iteration is performed, model parameters need to be initialized with some initial values. Not until long ago, a trivial approach would be to initialize all weights to zero or same value. It was however realized that this trivial approach resulted in poor training. Specialized initialization schemes have now been devised. If you are using ReLU activation functions, He initialization scheme should be the default preference; here the weights parameters for neurons in a layer are initialized by drawing samples from a normal

[62] Srivastava et. al., Dropout: A simple way to prevent neural networks from overfitting, The journal of machine learning research, 2014

`

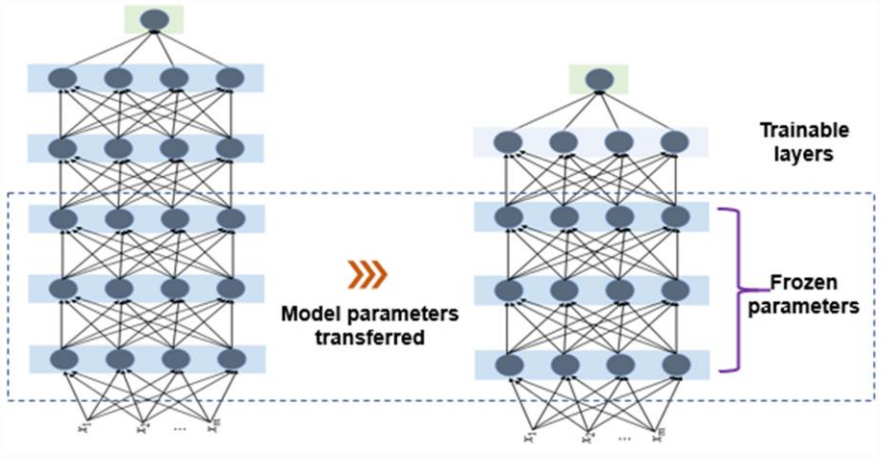
distribution centered at 0 with standard deviation = sqrt(2/n_inputs), where n_inputs is number of input connections (which equals number of neurons in previous layer for FFNN). For tanh activation functions, Xavier initialization scheme is found to be more suited.

Batch normalization

Batch normalization is another neat trick that have been devised to overcome the vanishing/exploding gradient issue for deep networks. The initialization schemes ensure that the weights within a layer are distributed appropriately. However, as training progresses and model parameters are update, the weights may lose their 'nice' distributions. This negatively impacts model training. With batch normalization strategy, at each training iteration with a mini-batch, the inputs to a layer are standardized. For deep networks, this strategy has been shown to provide significant training improvements. Implementing this in Keras is very straightforward wherein we add (`model.add(BatchNormalization())`) a batch normalization layer just before the layer whose inputs we wish to standardize.

Transfer Learning

Imagine that you have trained a deep network which provides good performance for some complex tasks, such as fault detection. Now you would like to replicate this to another similar site within your organization. However, you don't have a lot of training data at this second site – maybe the second site was commissioned recently and hasn't experienced many process faults yet! In such situations, you may want to adopt the popular strategy of 'transfer learning' wherein model parameters of inner layers of a pretrained model is transferred to a new model as shown in illustration below.



As alluded to before, inner layers in a DNN 'learn' the low-level features from process data and transfer learning simply transfers this learnt logic from one model to another for related tasks. This leads to faster convergence with limited data*.

*Li et. al., Transfer learning for process fault diagnosis: Knowledge transfer from simulation to physical processes, Computers & Chemical Engineering, 2020

`

11.7 Soft Sensing via FFNN for Debutanizer Column in a Petroleum Refinery

Let us employ all the knowledge we have gained regarding training FFNN models to develop a soft sensor for a very nonlinear system. For this, we will re-use the debutanizer column dataset that we saw in Chapter 7 (see the Dataset Description section in appendix for system details). The soft sensor is required to predict C4 content in bottoms product using other process data around the column. The dataset contains 2394 samples of input-output process values. The sampling time is 15 minutes. Seven process (pressures, temperatures, flows around debutanizer column) variables are used as inputs. Figure 11.13 shows that dataset has decent process variability.

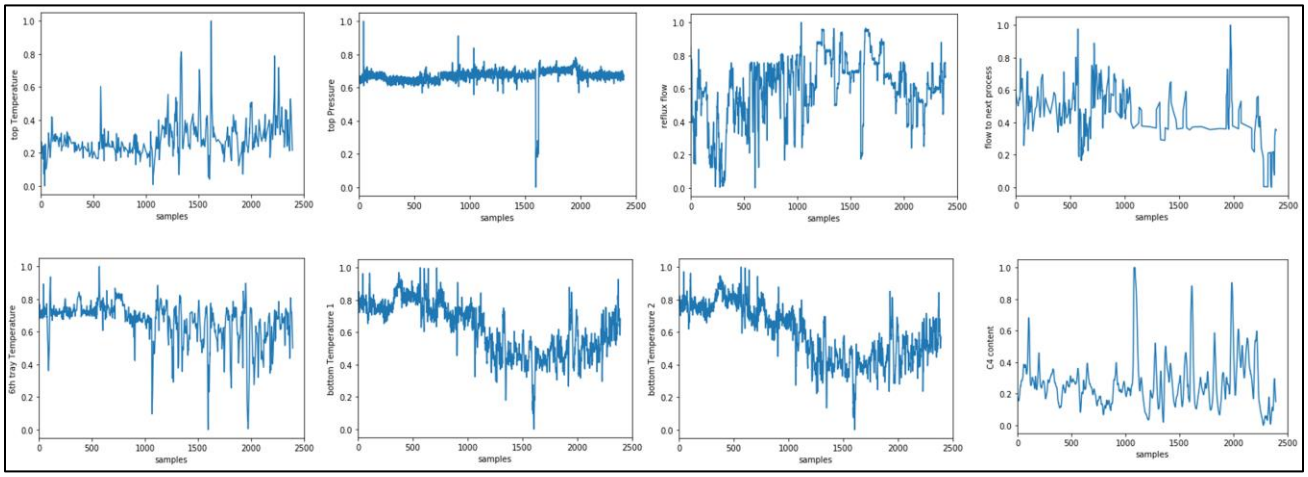


Figure 11.13: Plots of input and output (lower-right plot) variables for the debutanizer column

Before we build the FFNN model, a PLS model is built to serve as a reference for modeling accuracy assessment. Figure 11.14 suggests that the PLS model is grossly inadequate for predicting C4 content and hints the presence of strong non-linearities in the system. Specifically, the linear trend in residual plot clearly shows that the PLS model exhibits poor performance for high and low values of C4 content.

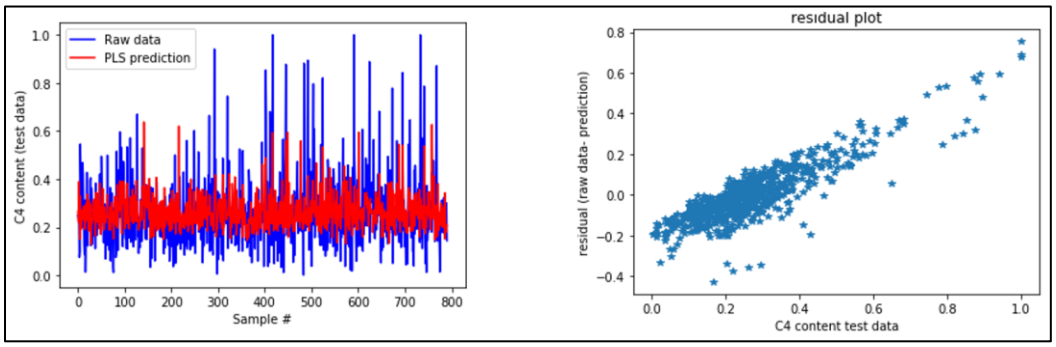


Figure 11.14: Comparison of test data with PLS predictions and residual plot

`

After a few trials, a network with 2 hidden layers with 60 and 30 neurons was obtained that gave 70% accuracy on test data. The code below shows the values of the rest of the hyperparameters. Figure 11.15 confirms the superior performance of FFNN model over PLS model. The residual plot does not show any significant trend indicating that the FFNN model is able to adequately capture the nonlinear relationships between the input and output variables. This case study demonstrates the power capabilities of ANN models to model complex process systems.

```
# read data
data = np.loadtxt('debutanizer_data.txt', skiprows=5)

# separate training, validation, and test data
from sklearn.model_selection import train_test_split
X, y = data[:,0:-1], data[:,-1][:,np.newaxis]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 100)
X_est, X_val, y_est, y_val = train_test_split(X_train, y_train, test_size = 0.25, random_state = 100)

# import packages
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam

# define model
model = Sequential()
model.add(Dense(60, kernel_regularizer=regularizers.L1(0.0000001), activation='relu',
kernel_initializer='he_normal', input_shape=(7,)))
model.add(Dense(30, kernel_regularizer=regularizers.L1(0.0000001), activation='relu',
kernel_initializer='he_normal'))
model.add(Dense(1, kernel_regularizer=regularizers.L1(0.0000001)))

# compile model
model.compile(loss='mse', optimizer=Adam(learning_rate=0.005))

# fit model
es = EarlyStopping(monitor='val_loss', patience=200)
history = model.fit(X_est, y_est, epochs=2000, batch_size=32, validation_data=(X_val, y_val),
callbacks=es)

# predict y
y_test_pred = model.predict(X_test)
```

`

```
# plot of raw and predicted data
plt.figure()
plt.plot(y_test, 'b', label=’Raw data’)
plt.plot(y_test_pred, 'r', label='FFNN prediction')
plt.ylabel('C4 content (test data)'), plt.xlabel('Sample #'), plt.legend()

# residual plot
plt.figure()
plt.plot(y_test, y_test-y_test_pred, '*')
plt.xlabel('C4 content test data'), plt.ylabel('residual (raw data- prediction)')
plt.title('residual plot')

# metrics
from sklearn.metrics import r2_score
print('R2 for test dataset:', r2_score(y_test, y_test_pred))

>>> R2 for test dataset: 0.707
```

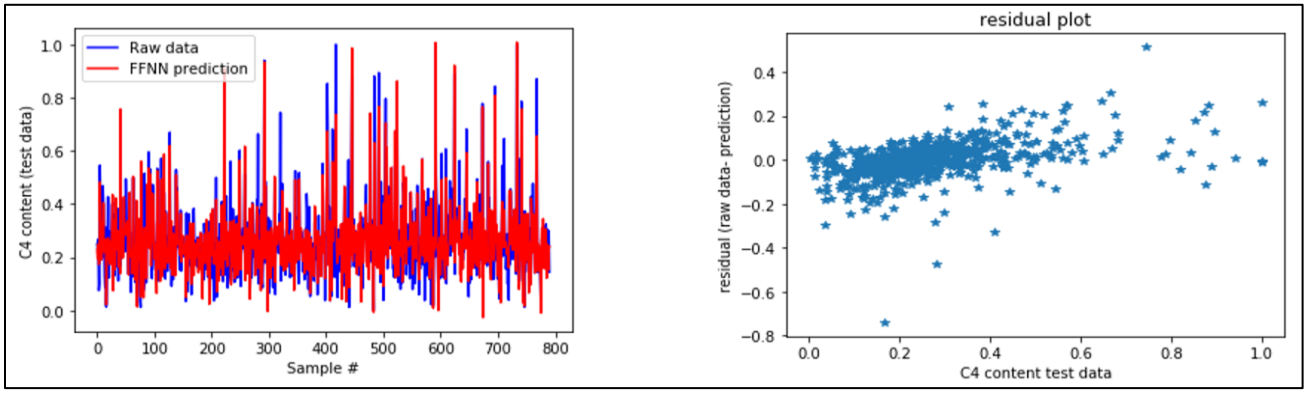


Figure 11.15: Comparison of test data with FFNN predictions and residual plot

`

FFNN Modeling Guidelines

ANN modeling is a very broad topic and with the plethora of tutorials on ANN modeling available out there, it is easy to get overwhelmed. However, we have seen in this chapter that ANNs are not as daunting as it may seem if we pay careful attention to a few key concepts. If you are looking for some quick guidelines on how to setup the ANN hyperparameters for process systems modeling, then the following suggestions will serve you well.

- **Number of hidden layers**: 1 for simple systems, 2 or 3 for complex systems/tasks
- **Activation function**: ReLU
- **Initialization scheme**: He initialization
- **Optimizer**: Adam
- **Regularization**: L1 or L2 kernel regularization
- **Learning rate schedule**: Not needed with Adam optimizer
- **Number of epochs**: Use a reasonably large value with early stopping
- **Mini-batch size**: 32

You will be lucky if you end up finding a good ANN model with the default settings in the very first attempt. Often the following adjustments may need to be made.

- If validation accuracy is much lower than training accuracy, then increase regularization penalty
- If optimizer is getting stuck in local minima, then adjust Adam's initial learning rate. If it does not help, then try adding more neurons or hidden layers
- If loss vs epoch curve is very noisy, then increase mini-batch size

Summary

Phew! that was a loaded chapter. We covered several techniques that are at our disposal to train ANN models efficiently. There are a lot more behind each of the several aspects that we touched upon, and we have only scratched the surface. However, you now are familiar with the core concepts of ANNs and have hands-on-experience with modeling the debutanizer column and combined cycle power plants. You are now well-aware of what it takes to develop a good ANN model for process systems. ANNs are going to be an important weapon in your arsenal which you will find yourself employing very often. In the next chapter we will learn how ANNs are deployed for dynamic or temporal data.