Lecture 24: (A Very Brief) Introduction to Artificial Neural Networks

MEGR 7080/8090: Dynamic System Learning and Estimation

Artur Wolek

Department of Mechanical Engineering and Engineering Science
UNC Charlotte

Fall 2022

Neural Networks (NNs) for Regression: Problem Setup

- Consider a static non-linear mapping from input $\boldsymbol{x} \in \mathbb{R}^n$ to output $\boldsymbol{y} \in \mathbb{R}^m$

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) \tag{1}$$

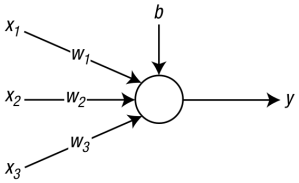
- Regression NN Goal: learn (1) given *training data* (input/output pairs):

$$\mathcal{T} = \{(\boldsymbol{x}_1, \boldsymbol{t}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{t}_N)\} \tag{2}$$

  where $\boldsymbol{x}_i$ is an input with corresponding (correct) output $\boldsymbol{t}_i$, and N is the number of training data points.
- Data may be synthetic, experimental, etc.

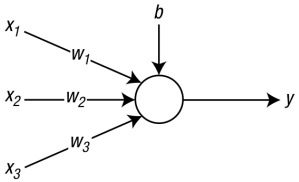
Architecture: Node

- Node: receives inputs (vector) and produces a single output (scalar)
- Consider three inputs $\boldsymbol{x} = [x_1, x_2, x_3]^{\mathrm{T}}$ and scalar output y.
- The simplest one is a *linear transformation*: multiply inputs by weights w_1, w_2, w_3, respectively, and the result is summed and biased by a factor b:



Architecture: Node

- Node: receives inputs (vector) and produces a single output (scalar)
- Consider three inputs $\boldsymbol{x} = [x_1, x_2, x_3]^{\mathrm{T}}$ and scalar output y.
- The simplest one is a *linear transformation*: multiply inputs by weights w_1, w_2, w_3, respectively, and the result is summed and biased by a factor b:



In matrix form:

$$y = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b = \boldsymbol{A}\boldsymbol{x} + b \tag{3}$$

Nodes

- If instead there are two node, then there will be two outputs $\boldsymbol{y} = [y_1, y_2]^{\mathrm{T}}$ then the transformation that describes both nodes would be

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4}$$

Nodes

- If instead there are two node, then there will be two outputs $\boldsymbol{y} = [y_1, y_2]^{\mathrm{T}}$ then the transformation that describes both nodes would be

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4}$$

- In general, any such transformation can be represented by the matrix equation

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} \tag{5}$$

  also called a *affine transformation*.
- The arrangement of inputs/nodes/outputs is arranged as set of *layers*.
- In the above example, the layer is parameterized by $(\boldsymbol{A}, \boldsymbol{b})$

- Terminology: input layer, output layer, and intermediate (hidden) layers
- Each column of nodes is called a layer
- The output from one layer becomes the input to the next.

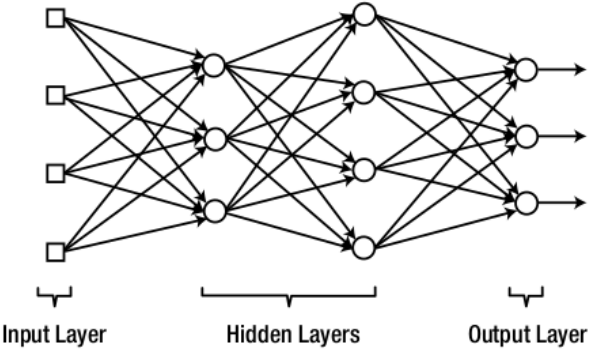


Image source: Kim 2017

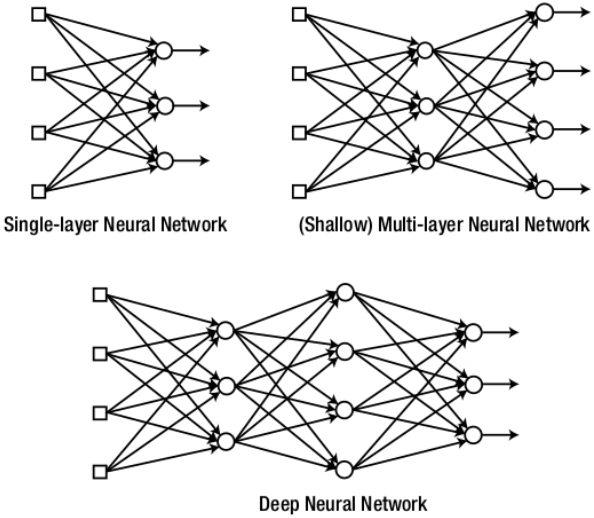


Image source: Kim 2017

Notation used here

- Consider a NN with d layers (including input/output layer)
- Input to each layer as $\boldsymbol{q}_i$ where $i \in \{1, \ldots, d\}$.

Notation used here

- Consider a NN with d layers (including input/output layer)
- Input to each layer as $\boldsymbol{q}_i$ where $i \in \{1, \ldots, d\}$.
- Input is the first layer $\boldsymbol{x} = \boldsymbol{q}_1 \in \mathbb{R}^n$
- Output is the last layer $\boldsymbol{y} = \boldsymbol{q}_d \in \mathbb{R}^m$.
- The intermediate $\boldsymbol{q}_i$'s are temp. variables of intermediate layers
- Note: layers can change size

Notation used here

- Consider a NN with d layers (including input/output layer)
- Input to each layer as $\boldsymbol{q}_i$ where $i \in \{1, \ldots, d\}$.
- Input is the first layer $\boldsymbol{x} = \boldsymbol{q}_1 \in \mathbb{R}^n$
- Output is the last layer $\boldsymbol{y} = \boldsymbol{q}_d \in \mathbb{R}^m$.
- The intermediate $\boldsymbol{q}_i$'s are temp. variables of intermediate layers
- Note: layers can change size
- Transition from one layer to next (for simple linear model):

$$\boldsymbol{q}_{i+1} = \boldsymbol{A}_i \boldsymbol{q}_i + \boldsymbol{b}_i \tag{6}$$

$$
\begin{aligned}
\text{Input} \qquad \boldsymbol{x} &= \boldsymbol{q}_1 \\
\text{Second layer} \qquad \boldsymbol{q}_2 &= \boldsymbol{A}_1\boldsymbol{q}_1 + \boldsymbol{b}_1 \\
&= \boldsymbol{A}_1\boldsymbol{x} + \boldsymbol{b}_1 \\
\text{Third layer} \qquad \boldsymbol{q}_3 &= \boldsymbol{A}_2\boldsymbol{q}_2 + \boldsymbol{b}_2 \\
&= \boldsymbol{A}_2(\boldsymbol{A}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \\
&= \boldsymbol{A}_2\boldsymbol{A}_1\boldsymbol{x} + (\boldsymbol{A}_2\boldsymbol{b}_1 + \boldsymbol{b}_2) \\
\text{Fourth layer} \qquad \boldsymbol{q}_4 &= \boldsymbol{A}_3\boldsymbol{q}_3 + \boldsymbol{b}_3 \\
&= \boldsymbol{A}_3(\boldsymbol{A}_2\boldsymbol{A}_1\boldsymbol{x} + (\boldsymbol{A}_2\boldsymbol{b}_1 + \boldsymbol{b}_2)) + \boldsymbol{b}_3 \\
&= \boldsymbol{A}_3\boldsymbol{A}_2\boldsymbol{A}_1\boldsymbol{x} + (\boldsymbol{A}_3\boldsymbol{A}_2\boldsymbol{b}_1 + \boldsymbol{A}_3\boldsymbol{b}_2 + \boldsymbol{b}_3) \\
&\vdots \\
\text{Output} \qquad \boldsymbol{y} = \boldsymbol{q}_d &= \boldsymbol{A}_{d-1}\boldsymbol{q}_{d-1} + \boldsymbol{b}_{d-1} \\
&= (\boldsymbol{A}_{d-1}\cdots\boldsymbol{A}_2\boldsymbol{A}_1)\boldsymbol{x} + (\boldsymbol{A}_{d-1}\cdots\boldsymbol{A}_2)\boldsymbol{b}_1 + (\boldsymbol{A}_{d-1}\cdots\boldsymbol{A}_3)\boldsymbol{b}_2 + \cdots + \boldsymbol{b}_{d-1}) \\
&= \prod_{i=1}^{d-1}\boldsymbol{A}_i\boldsymbol{x} + \sum_{i=1}^{d-1}\prod_{j=i}^{d-2}\boldsymbol{A}_j\boldsymbol{b}_i + \boldsymbol{b}_d
\end{aligned}
$$

Conclusion: The multi-layer *linear* network can be replaced with an equivalent single-layer network with weights and bias vectors given by

$$\tilde{\boldsymbol{A}} = \prod_{i=1}^{d} \boldsymbol{A}_i \quad \text{and} \quad \tilde{\boldsymbol{b}} = \sum_{i=1}^{d} \prod_{j=i}^{d-1} \boldsymbol{A}_j \boldsymbol{b}_i + \boldsymbol{b}_d$$

so that the output becomes $\boldsymbol{y} = \boldsymbol{q}_{d+1} = \tilde{\boldsymbol{A}}\boldsymbol{x} + \tilde{\boldsymbol{b}}$.

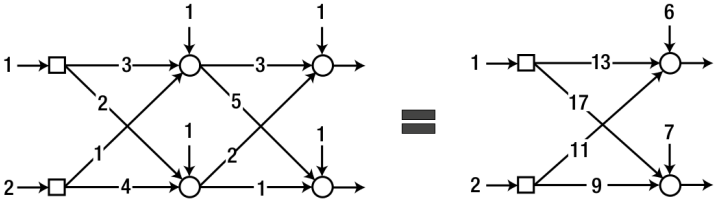


Image source: Kim 2017

Activation Functions

- Output of each node passes through a nonlinear *activation function* $\phi(\cdot)$:

$$y_k = \phi(\boldsymbol{A}\boldsymbol{x} + b) \tag{7}$$

- Allows a richer set of models to be constructed.

$$y_k = \phi(\boldsymbol{A}\boldsymbol{x}) \tag{8}$$

Activation Functions

- Output of each node passes through a nonlinear *activation function* $\phi(\cdot)$:

$$y_k = \phi(\boldsymbol{A}\boldsymbol{x} + b) \tag{7}$$

- Allows a richer set of models to be constructed.

$$y_k = \phi(\boldsymbol{A}\boldsymbol{x}) \tag{8}$$

- Note: Bias parameters can be absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$

Activation Functions

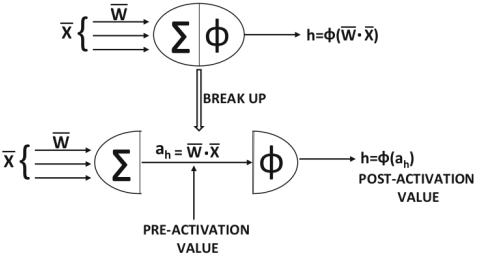
- Output of each node passes through a nonlinear *activation function* $\phi(\cdot)$:

$$y_k = \phi(\boldsymbol{A}\boldsymbol{x} + b) \tag{7}$$

- Allows a richer set of models to be constructed.

$$y_k = \phi(\boldsymbol{A}\boldsymbol{x}) \tag{8}$$

- Note: Bias parameters can be absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$



- Continuous or piece-wise differentiable (important for optimization)
- Examples:

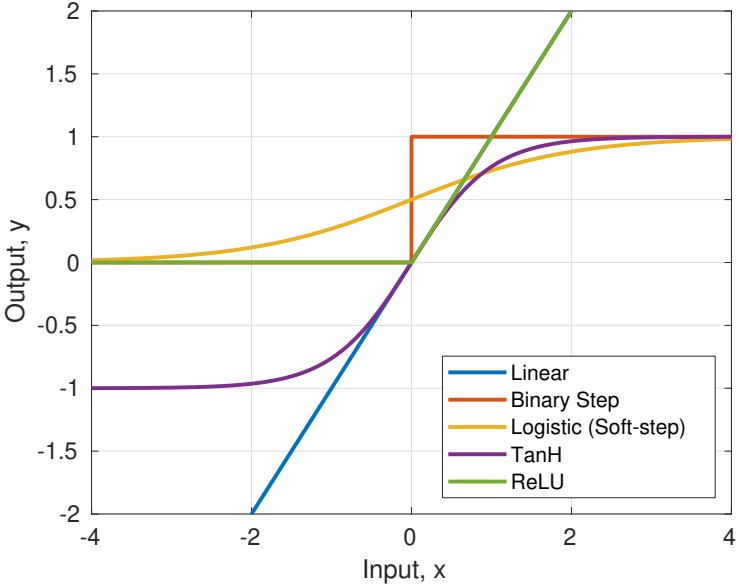
$$\phi(x) = x \qquad \text{(linear)} \tag{9}$$

$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \qquad \text{(binary step)} \tag{10}$$

$$\phi(x) = \frac{1}{1 + \exp(-x)} \qquad \text{(logistic (soft step))} \tag{11}$$

$$\phi(x) = \tanh(x) \qquad \text{(TanH)} \tag{12}$$

$$\phi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \qquad \text{(rectified linear unit (ReLU))} \tag{13}$$


2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Output, y
-4
-2
0
2
4
Input, x
Linear
Binary Step
Logistic (Soft-step)
TanH
ReLU

Layer output with activiation functions

The output vector at layer $i+1$ is determined by output at layer i:

$$\begin{bmatrix} (\boldsymbol{q}_{i+1})_1 \\ (\boldsymbol{q}_{i+1})_2 \\ \vdots \\ (\boldsymbol{q}_{i+1})_{N_{i+1}} \end{bmatrix} = \begin{bmatrix} \phi(\boldsymbol{A}_i^{(1)}\boldsymbol{q}_i) \\ \phi(\boldsymbol{A}_i^{(2)}\boldsymbol{q}_i) \\ \vdots \\ \phi(\boldsymbol{A}_i^{(N_i)}\boldsymbol{q}_i) \end{bmatrix} \tag{14}$$

Let $\boldsymbol{\Phi} : \mathbb{R}^{N_i} \to \mathbb{R}^{N_{i+1}}$. We can represent the transition of the intermediate $\boldsymbol{q}$ vectors through the affine transformations and activation functions as

$$\boldsymbol{q}_{i+1} = \boldsymbol{\Phi}_i(\boldsymbol{q}_i) \tag{15}$$

The parameters of the ith layer are $\boldsymbol{w}_i = \{w_{jk}\}$ for all j nodes in layer $(i-1)$ and k nodes in layer i. Weights determine $\boldsymbol{A}_i^{(k)}$ matrices.

As before:

$$
\begin{aligned}
\text{First (Input) Layer } \boldsymbol{x} &= \boldsymbol{q}_1 \\
\text{Second Layer } \boldsymbol{q}_2 &= \boldsymbol{\Phi}_1(\boldsymbol{q}_1) \\
\text{Third Layer } \boldsymbol{q}_3 &= \boldsymbol{\Phi}_2(\boldsymbol{q}_2) = \boldsymbol{\Phi}_3(\boldsymbol{\Phi}_2(\boldsymbol{q}_1)) \\
\text{Fourth Layer } \boldsymbol{q}_4 &= \boldsymbol{\Phi}_4(\boldsymbol{q}_3) = \boldsymbol{\Phi}_4(\boldsymbol{\Phi}_3(\boldsymbol{\Phi}_3(\boldsymbol{q}_1))) \\
&\vdots \\
\text{Output Layer } \boldsymbol{q}_{d+1} &= \boldsymbol{\Phi}_d(\boldsymbol{q}_d) \\
\implies \boldsymbol{y} = \boldsymbol{q}_{d+1} &= \boldsymbol{\Phi}_d(\boldsymbol{\Phi}_{d-1}(\cdots(\boldsymbol{\Phi}_2(\boldsymbol{\Phi}_1(\boldsymbol{x}))))))
\end{aligned}
$$

Neural Network Weights

- The NN with nonlinear activation functions is

$$\begin{aligned}\hat{\boldsymbol{y}} &= \hat{\boldsymbol{f}}(\boldsymbol{x};\boldsymbol{w}) \\ &= \boldsymbol{\Phi}_d(\boldsymbol{\Phi}_{d-1}(\cdots(\boldsymbol{\Phi}_2(\boldsymbol{\Phi}_1(\boldsymbol{x};\boldsymbol{w}_1);\boldsymbol{w}_2)\cdots);\boldsymbol{w}_{d-1});\boldsymbol{w}_d)\end{aligned}$$

- Passing in training data (2) through (**??**) gives predicted outputs:

$$\hat{\mathcal{T}} = \{(\boldsymbol{x}_1, \hat{\boldsymbol{y}}_1), \ldots, (\boldsymbol{x}_N, \hat{\boldsymbol{y}}_N)\}\ . \tag{16}$$

- Ideally, predictions $\{\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_N\}$ closely match correct outputs $\{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_N\}$
- A common cost (loss) function J is a sum of squared errors

$$J(\boldsymbol{w}) = \sum_{i=1}^{N} ||\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i||_2 = \sum_{i=1}^{N} ||\boldsymbol{y}_i - \hat{\boldsymbol{f}}(\hat{\boldsymbol{x}}_i, \boldsymbol{w})||_2 \tag{17}$$

where $||\cdot||_2$ denotes the l^2-norm (i.e., $||\boldsymbol{x}||_2 = \sqrt{\sum_{k=1}^{n} |x_k|^2}$ for $\boldsymbol{x} = [x_1, x_2, \ldots, x_k]^{\mathrm{T}}$.

Training a Neural Network

- Training a NN involves adjusting the weights $\boldsymbol{w}$ to minimize loss (cost)
- Supervised learning: we know the correct outputs
- Once we have optimized the NN to fit the training data we can see how well it does on a *validation* data set that was not part of $\mathcal{T}$
- Validaton data can also be included in training

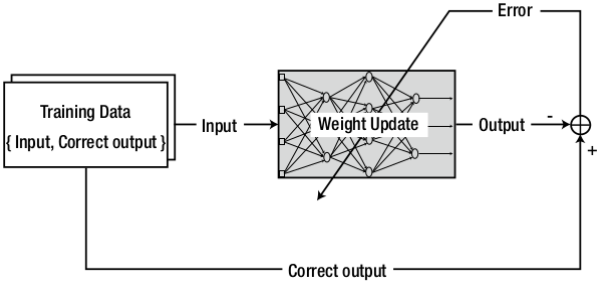


Image source: Kim 2017

Gradient Descent

At each new iteration $\tau + 1$: update the parameters

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E(\boldsymbol{w}^{(\tau)}) \tag{18}$$

where $\eta > 0$ is known as the learning rate.

Figure 5.5 Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.

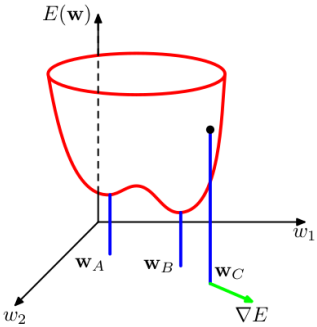


Image source: Bishop 2006

Stochastic Gradient Descent

The "batch" error over all training data is

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} E_n(\boldsymbol{w}) \tag{19}$$

To speed up computation we can consider only a subset of the data points at each iteration.

- Sequential Gradient Descent: cycle through all data points
- Stochastic Gradient Descent: select data points at random

SGD can also help escape local minima

How can we calculate the gradient of the error-function?

Consider error for just a single training data point

$$E_n = \frac{1}{2}\sum_{k=1}^{N_d}(y_k - t_k)^2 \tag{20}$$

where

- n: identifier for the particular training data input
- N_d: number of nodes in the output layer d
- y_k: predicted output for data input n at output node k
- t_k: correct output for data input n at output node k

Note: predictions/outputs correspond to data index n

The output q_k of an arbitrary node k in the network is a function of previous node outputs and weights

$$q_k = \phi\left(\sum_{i}^{k\text{th node's inputs}} w_{ki} q_i\right) \tag{21}$$

Let a_k denote the sum value prior to activation

$$a_k = \left(\sum_{i}^{k\text{th node's inputs}} w_{ki} q_i\right) \tag{22}$$

then

$$q_k = \phi(a_k) \tag{23}$$

Useful fact: Let

$$a_j = \left(\sum_{i}^{j\text{th node's inputs}} w_{ji} q_i \right) \quad \text{and} \quad a_k = \left(\sum_{m}^{k\text{th node's inputs}} w_{km} q_m \right) \tag{24}$$

Suppose output of node j feeds into node k for a particular value $m = j$ so that

$$q_{m=j} = \phi(a_j) \tag{25}$$

No other q_m depend on a_j except for the case $m = j$. Then

$$\implies \frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial q_j} \frac{\partial q_j}{\partial a_j} = w_{kj} \phi'(a_j) \tag{26}$$

The gradient of E_n with respect to a weight w_{ki}

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{1}{2}\frac{\partial}{\partial w_{ki}}\left(\underbrace{y_k}_{=\phi(a_k)} - t_k\right)^2 \tag{27}$$

$$= (y_k - t_k)\frac{\partial \phi}{\partial a_k}\frac{\partial a_k}{\partial w_{ki}} \tag{28}$$

$$= \underbrace{(y_k - t_k)\frac{\partial \phi}{\partial a_k}}_{\delta_k := \partial E_n / \partial a_k} q_i \tag{29}$$

$$= \delta_k q_i \tag{30}$$

Delta terms are called "errors"

Backpropagation

All of the q_i values are known because they were required to compute the output in the first "forward propagation" of the input

$$\frac{\partial E_n}{\partial w_{ki}} = \delta_k \underbrace{q_i}_{\text{from forward prop}} \tag{31}$$

The delta terms are easy to compute at the output layer

$$\delta_k := \frac{\partial E_n}{\partial a_k} = \left(\underbrace{y_k}_{\text{NN prediction}} - \underbrace{t_k}_{\text{correct output}} \right) \underbrace{\frac{\partial \phi}{\partial a_k}}_{\text{known derivative evaluated at } a_k} \tag{32}$$

The challenge is to find the delta terms at intermediate (hidden) nodes. Solution is known as *backpropagation*: use deltas at output to find deltas at previous layer and repeat.

Backpropagation argument:

- Variations in a_j give rise to variations in the error function E_n through variations in a_ks in the nex layer (recall node j connects to several nodes k in the forward direction)

$$\delta_j := \frac{\partial E_n}{\partial a_j} = \sum_k^{\text{next layer}} \underbrace{\frac{\partial E_n}{\partial a_k}}_{\delta_k} \underbrace{\frac{\partial a_k}{\partial a_j}}_{w_{kj}\phi'(a_j)} \tag{33}$$

$$\implies \delta_j = \phi'(a_j) \sum_k^{\text{next layer}} w_{kj}\delta_k \tag{34}$$

Allows us to compute δ_j from downstream deltas in the next layer

Backpropagation Summary

1. Apply an input vector $\boldsymbol{x}_n$ to the network and forward propagate to produce the outputs. Record the outputs q_k and summation values a_k of all hidden nodes.
2. Evaluate δ_k for all of the output nodes (in the final layer)
3. Use backprop to find δ_j for the second-to-last layer. Continue recursively working backwards one layer at a time to the input layer.
4. Assemble the error derivative $\nabla E(\boldsymbol{w}^{(\tau)})$ from $\frac{\partial E_n}{\partial w_{ki}} = \delta_k q_i$
5. Update weights
$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E(\boldsymbol{w}^{(\tau)}) \tag{35}$$
6. Go back to Step 1. Repeat.

When to stop?

Overfitting

Principle of Parsimony

"with competing theories or explanations, the simpler one, for example a model with fewer parameters, is to be preferred"

(Source: Wikipedia)

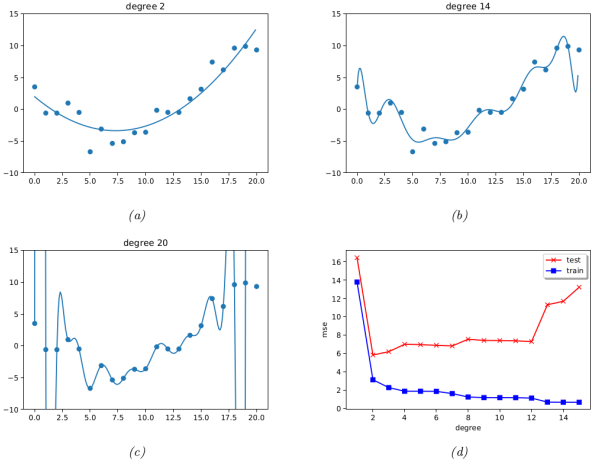


Figure 1.7: (a-c) Polynomials of degrees 2, 14 and 20 fit to 21 datapoints (the same data as in Figure 1.5). (d) MSE vs degree. Generated by code.probml.ai/book1/1.7.

Image source: Murphy 2022

Simple approach: Early Stopping

- Stop training when validation error begins to increase

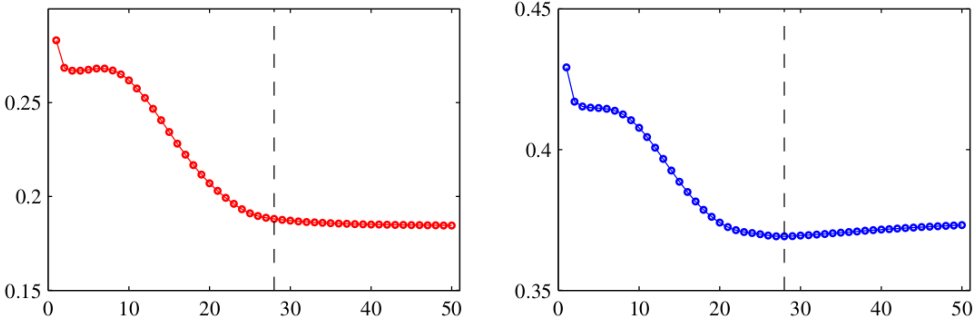


Figure 5.12 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

Image source: Bishop 2006

Training Issues

- Regularization tackles overfitting: add term $\lambda C(\boldsymbol{w})$ to penalize model complexity where λ is a regularization parameter
- How to choose λ?
  - Grid search (to maximize performance on validation data)
  - k-folds cross-validation $\rightarrow$ optimize $\lambda^* \rightarrow$ train with all data $\rightarrow \boldsymbol{w}^*$
- Another approach: dropout (randomly remove nodes during training)
- Tackling poor performance
  - Adjust network architecture
  - Adjust training parameters
  - Collect more data
  - Is the problem well posed?

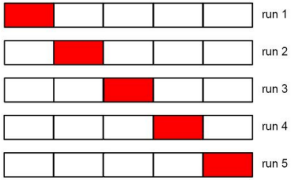


Figure 4.6: Schematic of 5-fold cross validation.

Image source: Murphy 2022

Neural Network Implementation

	Keras	TensorFlow	PyTorch
Level of API	high-level API[1]	Both high & low level APIs	Lower-level API[2]
Speed	Slow	High	High
Architecture	Simple, more readable and concise	Not very easy to use	Complex[3]
Debugging	No need to debug	Difficult to debugging	Good debugging capabilities
Dataset Compatibility	Slow & Small	Fast speed & large	Fast speed & large datasets
Popularity Rank	1	2	3
Uniqueness	Multiple back-end support	Object Detection Functionality	Flexibility & Short Training Duration
Created By	Not a library on its own	Created by Google	Created by Facebook[4]
Ease of use	User-friendly	Incomprehensive API	Integrated with Python language
Computational graphs used	Static graphs	Static graphs	Dynamic computation graphs[5]

Image Source: Yu-Cheng Kuo (Analytics Vidhya) [Link]

Regression network in MATLAB

1) Randomly select initial conditions

```
x0Mat = ((rand([Nic 2])-0.5)*2)*3; % generate random ICs
```

2) Simulate the ODE, gather state-rate data

```
for i = 1:1:size(x0Mat,1)
    [t,X] = ode45(@(t,x) double_well(t,x,params), tspan, x0Mat(i,:) , options);
    tt = [tt; t];
    Xt = [Xt; X];
end
for i = 1:1:length(tt)
    dxdt = double_well(tt(i),Xt(i,:)',params);
    Xdot(i,1) = dxdt(1);
    Xdot(i,2) = dxdt(2);
end
```

3) Arrange into a table. Split into training/validation sets.

```
Xtrain = array2table([tt Xt Xdot],'VariableNames',
{'t','x1','x2','x1dot','x2dot'});
c = cvpartition(size(Xtrain,1),"Holdout",0.30);
trainingIndices = training(c);
validationIndices = test(c);
tblTrain = Xtrain(trainingIndices,:);
tblValidation = Xtrain(validationIndices,:);
```

4) Train

```
x1dot_mdl = fitrnet(Xtrain,'x1dot','PredictorNames',{'t','x1','x2'},
"Standardize",true,"ValidationData",tblValidation,"LayerSizes",layerSize);
```

5) Predict (within ode45)

```
dxdt(1,1) = predict(x1dot_mdl, [t xx']);
```

10 ICs, 4 Layers, 30 nodes each, default settings