ECE 1513: Introduction to Machine Learning

Lecture 7: Neural Networks I

Ali Bereyhi

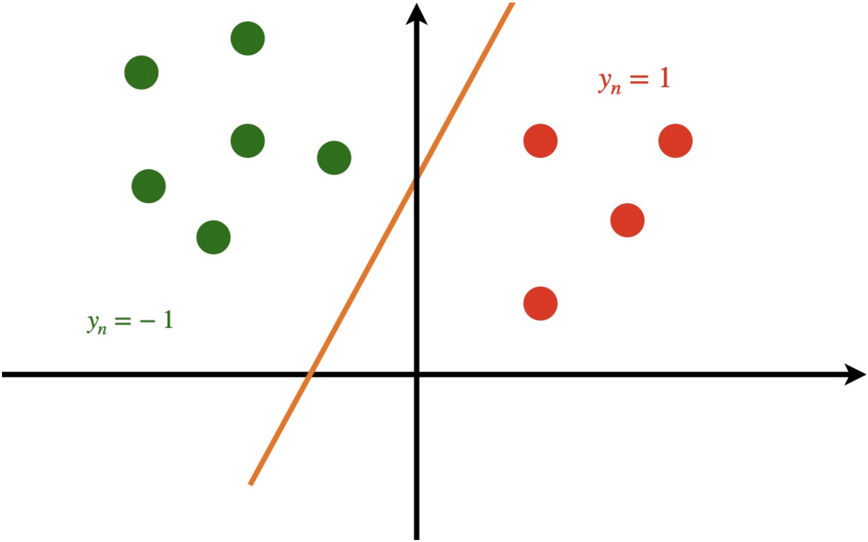
ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

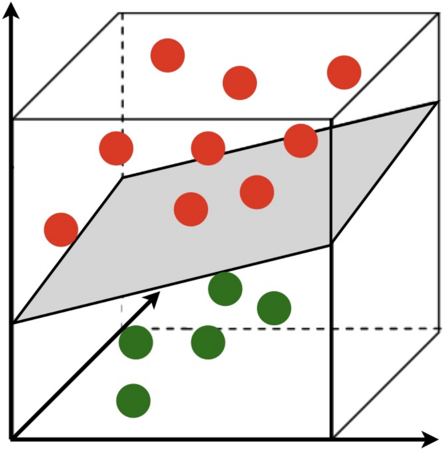
University of Toronto

Winter 2025

Quick Recap: *Linear Classifier*



Quick Recap: *Support Vector Machine*



Today's Agenda: *Support Vector Machine*

Today, we study a class of nonlinear models which is very powerful, i.e.,

Neural Networks

In this way, we discuss the following topics

- *Representing a binary function*
- *Neural networks as universal representation*
- *Expressive power of neural networks*
  - ↳ *Universal Approximation Theorem*

Learning Binary Functions

samples labels

We are given with all four cases of two ~~binary variables~~ and ~~their OR~~, i.e.,

$\mathbf{x} = [x_1, x_2]$ $$\mathbb{D} = \{([0,0],0),([0,1],1),([1,0],1),([1,1],1)\}$$

$v = x_1 \vee x_2$

Question: *Can we learn this binary function by a linear classifier from* $\mathbb{D}$*?*

$$y = \mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b \rightsquigarrow v = \begin{cases} 1 & y \geqslant 0 \\ 0 & y < 0 \end{cases}$$

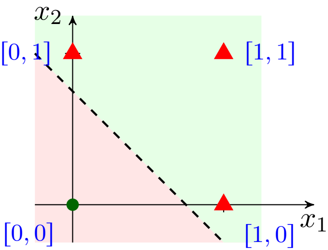
Example: *OR Function*

The OR of two binary variables $x_1, x_2 \in \{0,1\}$ is

$$x_1 \vee x_2 = \begin{cases} 1 & \text{if } x_1 \text{ or } x_2 = 1 \\ 0 & \text{if } x_1 = x_2 = 0 \end{cases}$$

OR Function: *Binary Classification*

Let's show data-points with label $y = 1$ by ▲ and those with label $y = 0$ by •



$\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$

It works!

Learning XOR Function

Let's now consider the XOR example

$$\mathbf{x} = [x_1, x_2] \quad \mathbb{D} = \{([0,0],0),([0,1],1),([1,0],1),([1,1],0)\}$$

$$v = x_1 \oplus x_2$$

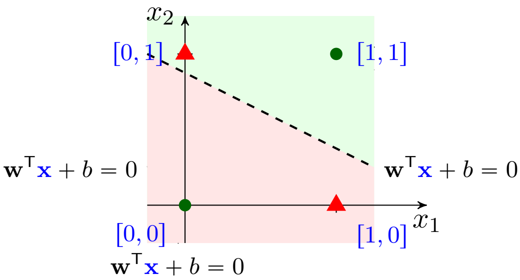
Reminder: *XOR Function*

The XOR of two binary variables $x_1, x_2 \in \{0,1\}$ is

$$x_1 \oplus x_2 = \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{if } x_1 = x_2 \end{cases}$$

Learning XOR Function: Binary Classification

Let's show data-points with label $y = 1$ by ▲ and those with label $y = 0$ by ●

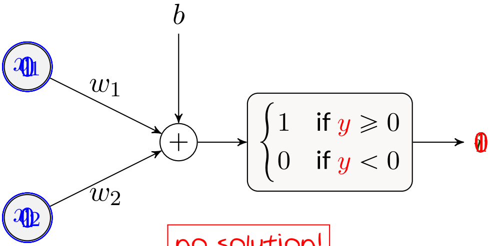


It seems to be impossible!
Well! We can show it

and look for $\mathbf{w}^{\mathsf{T}}\mathbf{x} + b = 0$ *that separates the labels*

Learning XOR Function: Binary Classification

Let's check it $\mathbb{D} = \{([0,0],0)\ ,\ ([0,1],1)\ ,\ ([1,0],1)\ ,\ ([1,1],0)\}$



no solution!

$$0w_1 + 0w_2 + b < 0 \rightsquigarrow \boxed{b < 0} \qquad\qquad -2b \leqslant w_1 + w_2 < -b$$

$$0w_1 + 1w_2 + b \geqslant 0 \rightsquigarrow w_2 + b \geqslant 0$$

$$1w_1 + 0w_2 + b \geqslant 0 \rightsquigarrow w_1 + b \geqslant 0$$

$$\rightsquigarrow \boxed{-2b \leqslant w_1 + w_2}$$

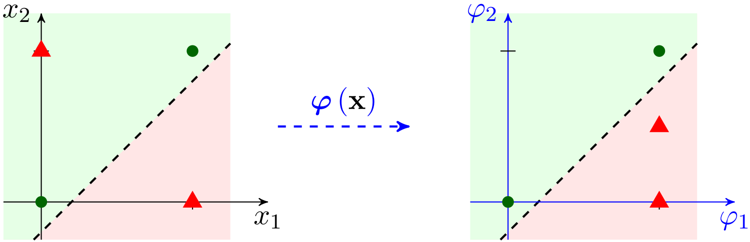
$$1w_1 + 1w_2 + b < 0 \rightsquigarrow w_1 + w_2 + b < 0 \rightsquigarrow \boxed{w_1 + w_2 < -b}$$

Learning XOR Function: Simple Remedy

So, we can conclude that

XOR function is not linearly separable

How can we solve this problem? *SVM, and no need for higher dimensions!*



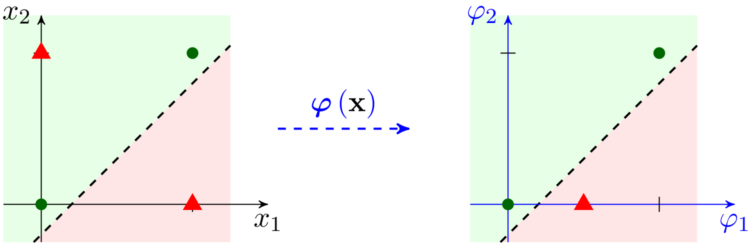
Then, we could learn XOR perfectly from $\varphi(\mathbf{x})$ via a linear model

Learning XOR Function: Simple Remedy

Consider the following transform

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow{\varphi(\mathbf{x})} \varphi(\mathbf{x}) = \begin{bmatrix} \varphi_1 \\ \varphi_2 \end{bmatrix} = \begin{bmatrix} 0.5 \max\{x_1 + x_2, 0\} \\ \max\{x_1 + x_2 - 1, 0\} \end{bmatrix}$$

Let's apply the transform on our data points

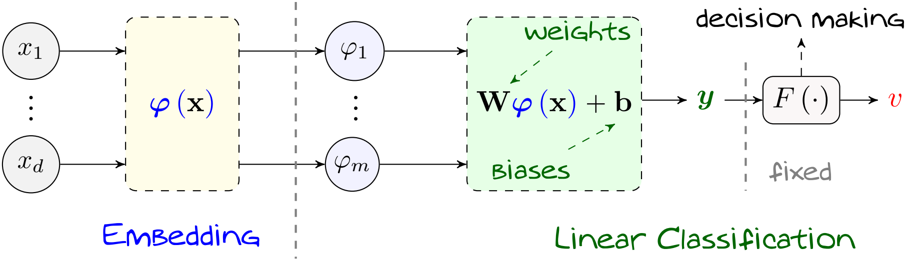


Applying binary classification on $\varphi(\mathbf{x})$, we can now learn XOR correctly

Visualizing the Nonlinear Model

We can visualize this as follows

How can we find the right embedding?



Learning Kernel

Attempt 1: *Engineering*

Engineer the feature, i.e., *find a good kernel by hand (trial and error)* ✖

Attempt 2: *Representation Learning*

Why not learning the embedding itself? Agree on a $\varphi(\mathbf{x};\omega)$, e.g.,

$$\varphi(\mathbf{x};\omega_0,\omega_1,\omega_2) = \omega_0 + \omega_1\mathbf{x} + \omega_2\mathbf{x}^2$$

Then try to learn ω along with the linear model, i.e., solve

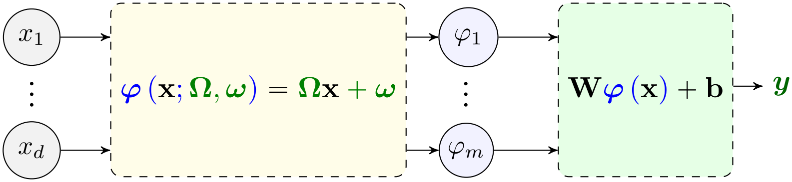
$$\min_{\mathbf{W},\mathbf{b}} \hat{R}(\mathbf{W},\mathbf{b}) \rightsquigarrow \min_{\mathbf{W},\mathbf{b},\omega} \hat{R}(\mathbf{W},\mathbf{b},\omega)$$

Basic Property of Embedding

A Basic Question

What kind of property should the embedding have?

Let's try a naive choice, i.e., *a linear embedding*



In this embedding,

- dimension of $\boldsymbol{\omega}$, i.e., m, is a hyper-parameter
- $\boldsymbol{\Omega}$ and $\boldsymbol{\omega}$ are learnable

Basic Property of Kernels

Well, let's see how y looks like

$$y = \mathbf{W}\varphi(\mathbf{x}) + \mathbf{b} = \mathbf{W}\left(\boldsymbol{\Omega}\mathbf{x} + \boldsymbol{\omega}\right) + \mathbf{b} = \underbrace{\cancel{\mathbf{W}\boldsymbol{\Omega}}}^{\tilde{\mathbf{W}}}\mathbf{x} + \overbrace{\cancel{\mathbf{W}\boldsymbol{\omega} + \mathbf{b}}}^{\tilde{\mathbf{b}}}$$

$$= \tilde{\mathbf{W}}\mathbf{x} + \tilde{\mathbf{b}}$$

linear embedding doesn't do anything!

A Basic Question

What kind of property should the embedding have?

Simple Answer

It should definitely contain some non-linearity

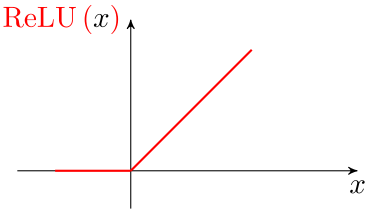
Wait a minute! Was it the case in the XOR example?!

Back to the XOR Example

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow{\boldsymbol{\varphi}(\mathbf{x})} \boldsymbol{\varphi}(\mathbf{x}) = \begin{bmatrix} \varphi_1 \\ \varphi_2 \end{bmatrix} = \begin{bmatrix} \max\{0.5(x_1 + x_2), 0\} \\ \max\{x_1 + x_2 - 1, 0\} \end{bmatrix}$$

Here, the non-linearity comes from the rectified linear unit (ReLU) function

$$\boxed{\mathrm{ReLU}(x) = \max\{x, 0\}}$$



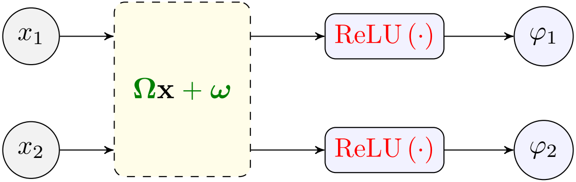
Back to the XOR Example

Defining $\mathrm{ReLU}\left(\mathbf{x}\right)$ to apply entry-wise on $\mathbf{x}$, we can write

$$\varphi\left(\mathbf{x}\right) = \begin{bmatrix} \mathrm{ReLU}(0.5x_1 + 0.5x_2) \\ \mathrm{ReLU}(x_1 + x_2 - 1) \end{bmatrix} = \mathrm{ReLU}\left(\begin{bmatrix} 0.5x_1 + 0.5x_2 \\ x_1 + x_2 - 1 \end{bmatrix}\right)$$

$$= \mathrm{ReLU}\left(\begin{bmatrix} 0.5 & 0.5 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right) = \mathrm{ReLU}\left(\boldsymbol{\Omega}\mathbf{x} + \boldsymbol{\omega}\right)$$

This is a linear embedding with only a tiny bit of non-linearity!



Deep Learning by Cascading Basic Blocks

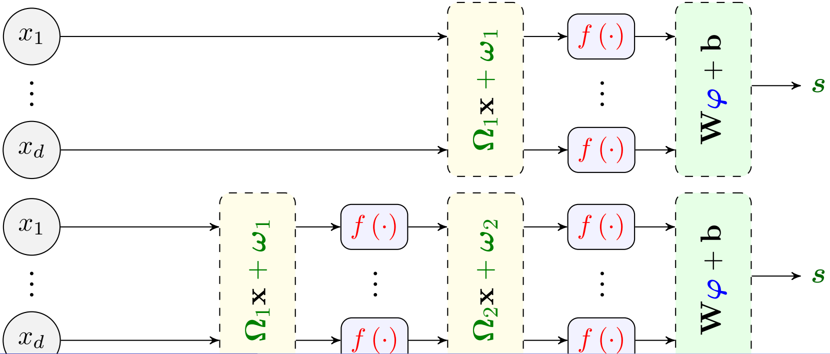
Moral of Story

Learnable linear mapping cascaded by a nonlinear function can be embedding

For simple learning problems like learning XOR function

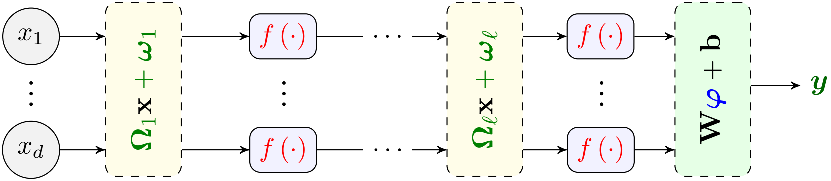
For more complicated problems we can add more layers

The more complicated the problem gets, the deeper we could go!



FNN Basics

FNN with ℓ layers between inputs and outputs looks like this:



Let's polish this diagram using some notions in the literature

Artificial Neuron

An artificial neuron (neural unit) is a basic computation unit

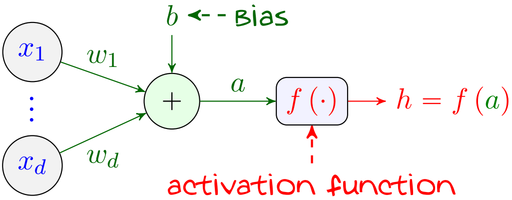
with multiple inputs and a single output

that does the following items:

- *It determines a linear combination of its inputs*

$$a = \sum_{i=1}^{d} w_i x_i + b$$

- *It sets a function of the linear combination as the output*



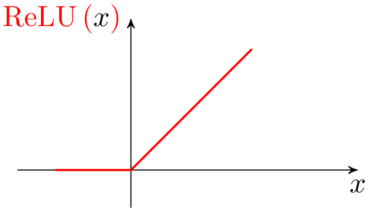
Activation

$\boxed{\text{Activation function is a } \textit{nonlinear} \text{ transform } f(\cdot) : \mathbb{R} \mapsto \mathbb{R}}$

We have already seen the example of ReLU. Other classical choices are

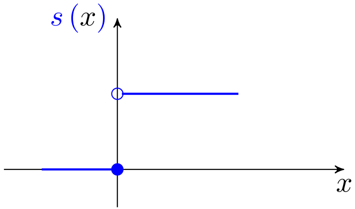
Rectified Linear Unit

$$\boxed{\mathrm{ReLU}(x) = \max\{x, 0\}}$$



Step Function

$$\boxed{s(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leqslant 0 \end{cases}}$$



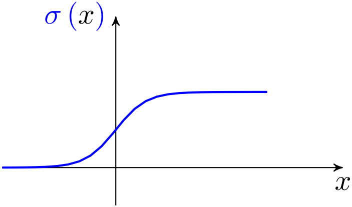
FNN Basics: Neuron

$$\boxed{\text{Activation function is a nonlinear transform } f(\cdot) : \mathbb{R} \mapsto \mathbb{R}}$$

Other classical choices are

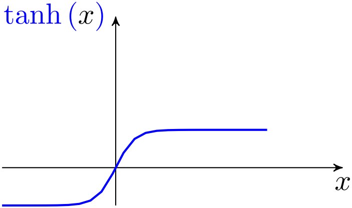
Sigmoid

$$\boxed{\sigma(x) = \frac{1}{1 + \exp\{-x\}}}$$



Hyperbolic Tangent

$$\boxed{\tanh(x) = \frac{\exp\{x\} - \exp\{-x\}}{\exp\{x\} + \exp\{-x\}}}$$



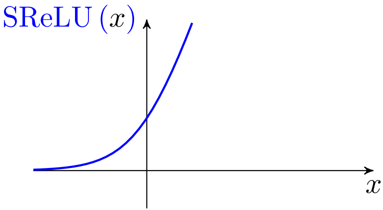
FNN Basics: Neuron

$$\boxed{\text{Activation function is a nonlinear transform } f(\cdot) : \mathbb{R} \mapsto \mathbb{R}}$$

Other classical choices are

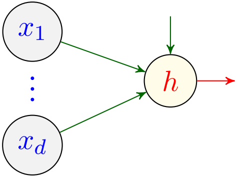
Soft ReLU

$$\boxed{\mathrm{SReLU}(x) = \log(1 + \exp\{x\})}$$



Artificial Neuron

We use the following shortened diagram to represent a neuron



Let's make an agreement

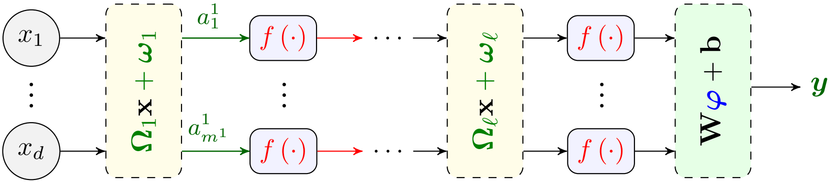
- *green edges are learnable: we find them by minimizing empirical risk*
- *red edge denotes the value we get after applying activation*

Why do we call this block *Neuron*?

The appellation is inspired by our understanding of biological neurons

Building FNNs via Neurons

Now, let us use the neuron block to polish the FNN

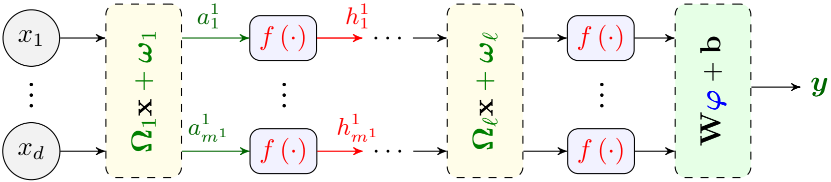


We start with first layer: let m^1 denote the number of features in this layer

$$\begin{bmatrix} a_1^1 \\ \vdots \\ a_{m^1}^1 \end{bmatrix} = \boldsymbol{\Omega}_1 \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} + \boldsymbol{\omega}_1 = \begin{bmatrix} \Omega_1^1[1] & \dots & \Omega_1^1[d] \\ \vdots & & \vdots \\ \Omega_{m^1}^1[1] & \dots & \Omega_{m^1}^1[d] \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} \omega_1^1 \\ \vdots \\ \omega_{m^1}^1 \end{bmatrix}$$

FNNs as Networks of Neurons

Now, let us use the neuron block to polish the FNN



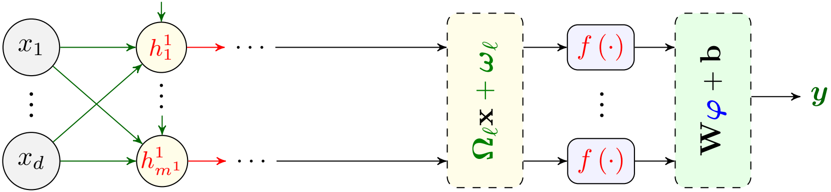
Therefore, a particular pair of a_j^1 *and* h_j^1 *for* $j \in \{1, \dots, m^1\}$ *are given by*

$$a_j^1 = \sum_{i=1}^{d} \Omega_j^1 [i] \, x_i + \omega_j^1 \qquad \text{and} \qquad h_j^1 = f\left(a_j^1\right)$$

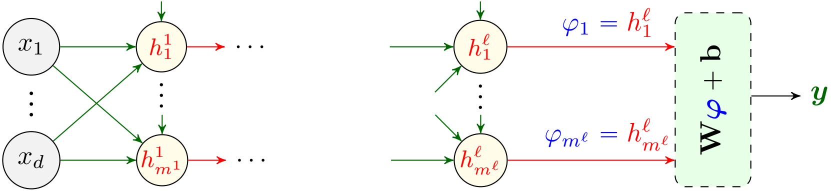
This is a neuron with inputs $x_1, \dots, x_d$ *and output* h_j^1

FNNs as Networks of Neurons

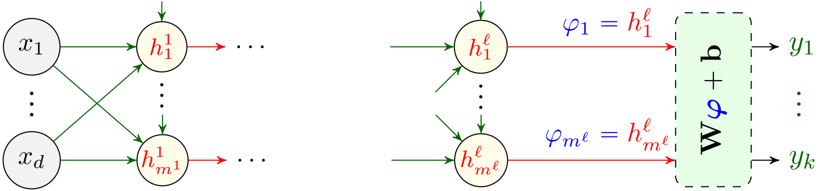
So, we could represent the first layer as



We can do the same for all the subsequent layers



FNNs for Regression



Let's denote the number of entries in $\boldsymbol{y}$ by k; we can then write

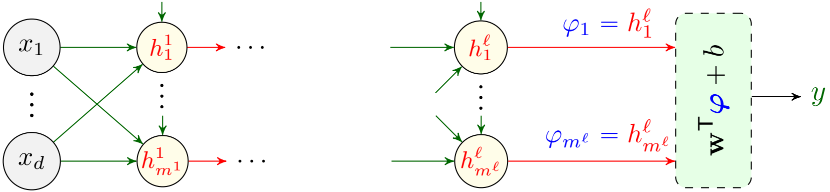
entry at row j and column i of $\mathbf{W}$

entry j of $\boldsymbol{y}$ $\leftarrow$ - - - $$y_j = \sum_{i=1}^{m^\ell} w_j[i]\,\varphi_i + b_j$$ - - $\rightarrow$ entry j of $\mathbf{b}$

$\boldsymbol{y}$ is a nonlinear function of $\mathbf{x}$!

FNNs for Classification

We can predict the class from $\boldsymbol{y}$



Example: *Binary Classification*

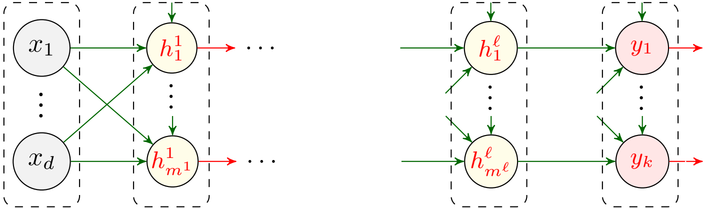
Say we get a single output, i.e., $k = 1$*. We can classify as*

$$\hat{v} = s(y) = \begin{cases} 1 & \text{if } y \geqslant 0 \\ 0 & \text{if } y < 0 \end{cases}$$

Or we can set $\Pr\{v = 1 | \mathbf{x}\} = \sigma(y)$

FNNs: Some Definitions

FNN is represented as a network of neurons - -> reason of appellation



1   # ℓ

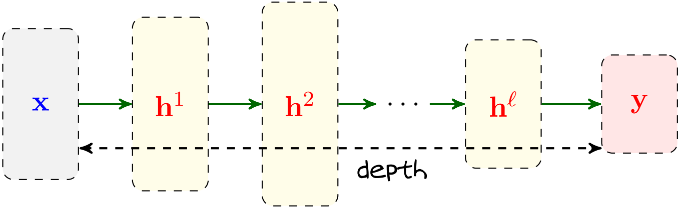
input layer   hidden layers   output layer

The network has three types of layers

- *input layer that takes the data-point* $\mathbf{x}$ *as the input*
- *output layer the uses a linear model to learn from the features*
- *hidden layers between input and output that extract the features*

FNNs: Some Definitions

Let's now represent FNN a bit compactly

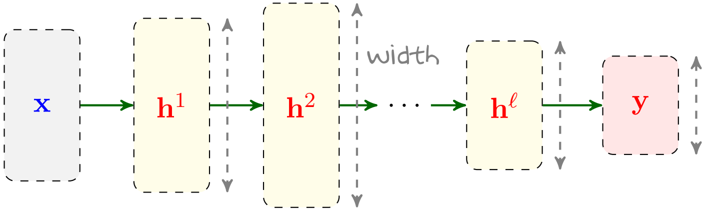


Depth of Neural Network

Number of layers including output layer, excluding input layer, i.e., $\ell + 1$

FNNs: Some Definitions

Let's now represent FNN a bit compactly



Width of a Layer

Number of neurons in the layer, i.e., m^j for hidden layer # j and k for output

In general, the width *changes from a layer to another*

Sometimes we refer to the maximum width as width of the network

FNNs: Few Extra Terminologies

Feedforward Neural Networks

FNNs represent *directed acyclic graph*, i.e., there is no cycle

We have only considered FNNs with *Fully-Connected* Layers

Fully-Connected Layer

Each neuron in the layer is connected to all outputs of the previous layer[1]

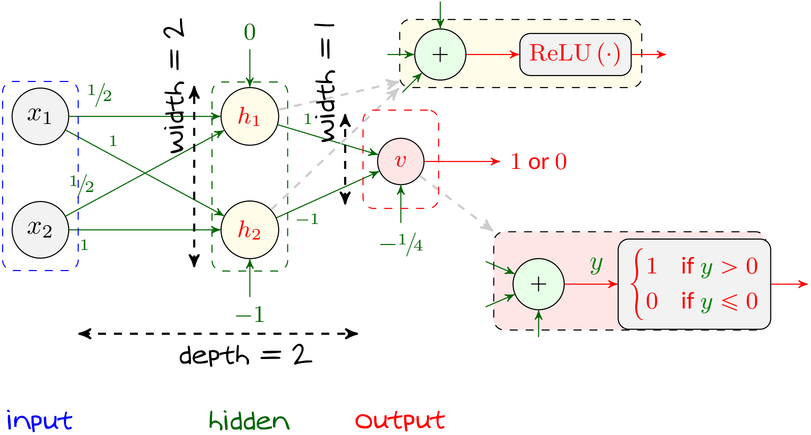
Fully-Connected FNNs are also called Multilayer Perceptrons (MLPs)

- This appellation is considered a misnomer (*check Wikipedia*)

[1] We will see also convolutional layers later on in the course

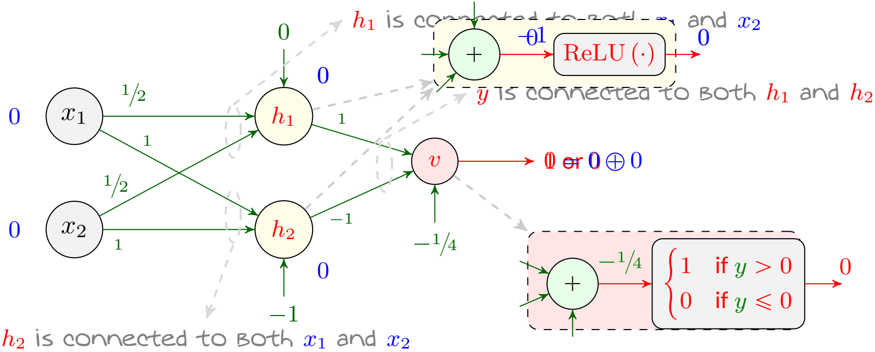
FNNs: XOR Example Revisited

Let's apply our knowledge on the simple XOR example



FNNs: XOR Example Revisited

Let's apply our knowledge on the simple XOR example



We can hence say

- First layer is fully-connected
- Second layer is fully-connected

Expressive Power of Models

Key Question

How expressive NNs are?

Let's start with a basic form of this question

- ? *How many binary functions we can represent by an NN?*

Binary Function $\equiv$ *Truth Table*

x_1	$\cdots$	x_d	$f(\boldsymbol{x})$
0	$\cdots$	0	$0/1$
0	$\cdots$	1	$0/1$
	$\vdots$		$\vdots$
1	$\cdots$	1	$0/1$

Complexity of Binary Space

x_1	$\cdots$	x_d	$f(\boldsymbol{x})$
0	$\cdots$	0	0/1
0	$\cdots$	1	0/1
	$\vdots$		$\vdots$
1	$\cdots$	1	0/1

Number of Possible Binary Functions

In general we have 2^d input combinations

↳ *each can have a binary output*

We have 2^{2^d} possible functions!

Complexity of Binary Space: *Example*

With $d = 2$ we have 16 different cases!

x_1	x_2	$f_1(\boldsymbol{x})$
0	0	0
0	1	0
1	0	0
1	1	0

x_1	x_2	$f_2(\boldsymbol{x})$
0	0	1
0	1	0
1	0	0
1	1	0

x_1	x_2	$f_3(\boldsymbol{x})$
0	0	0
0	1	1
1	0	0
1	1	0

x_1	x_2	$f_4(\boldsymbol{x})$
0	0	1
0	1	1
1	0	0
1	1	0

$\cdots$

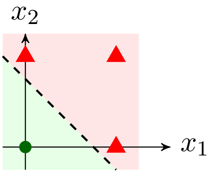
x_1	x_2	$f_{16}(\boldsymbol{x})$
0	0	1
0	1	1
1	0	1
1	1	1

Thresholding Binary Functions

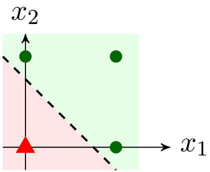
Key Observation

Many of these functions are linearly separable

x_1	x_2	$f_2(\boldsymbol{x})$
0	0	1
0	1	0
1	0	0
1	1	0



x_1	x_2	$f_{15}(\boldsymbol{x})$
0	0	0
0	1	1
1	0	1
1	1	1



Thresholding Binary Functions

Key Observation

Many of these functions are linearly separable

We can realize these functions via a single artificial neuron

- ! *Recall that a neuron is a linear classifier*

Nonlinear Forms

But some of these functions are not linearly separable

- ! *We have already seen XOR!*
  - ↳ *Maybe we can represent them by NNs!*

More Complex Binary Functions

If a function is not linearly separable

we can decompose it as sum of linearly separable functions

Example

x_1	x_2	$f_{10}(\boldsymbol{x})$
0	0	1
0	1	0
1	0	0
1	1	1

=

x_1	x_2	$f_2(\boldsymbol{x})$
0	0	1
0	1	0
1	0	0
1	1	0

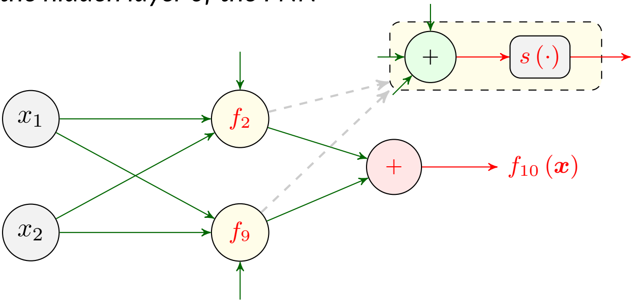
+

x_1	x_2	$f_9(\boldsymbol{x})$
0	0	0
0	1	0
1	0	0
1	1	1

More Complex Binary Functions

We can realize each linearly separable by a separate neuron

↳ *this is the hidden layer of the FNN*



We finally add them up at the output layer

General Binary Functions

? *How wide the hidden layer should be if we have d inputs?*

Worst-Case Function

Worst-case binary function returns 1 for half of combinations and 0 for the rest

Example

This is indeed XOR!

x_1	x_2	$f_{10}(\boldsymbol{x})$
0	0	1
0	1	0
1	0	0
1	1	1

General Binary Functions

? *How wide the hidden layer should be if we have d inputs?*

Worst-Case Function

Worst-case binary function returns 1 for half of combinations and 0 for the rest

We can decompose any binary function with d input as

$$f(\boldsymbol{x}) = \sum_{i=1}^{W} g_i(\boldsymbol{x})$$

with each g_i being *linearly separable*,where W is *at most*

$$W \leqslant \frac{\text{\# of Possible Combinations}}{2} = \frac{2^d}{2} = 2^{d-1}$$

General Binary Functions

? *How wide the hidden layer should be if we have d inputs?*

We can realize each g_i by a separate neuron in a hidden layer

$$f(\boldsymbol{x}) = \sum_{i=1}^{W} g_i(\boldsymbol{x})$$

f is then realized by adding them all at the output layer

Conclusion

Any binary function with d inputs can be realized by a shallow FNN whose hidden layer has 2^{d-1} neurons

Shallow FNN $\equiv$ only one hidden layer

General Binary Functions

Conclusion

Any binary function with d inputs can be realized by a shallow FNN whose hidden layer has 2^{d-1} neurons

? *Aren't we thinking of very large d?! Then the FNN is super wide!*

! *Yes indeed!*

Nested Decomposition of Binary Functions

! *Maybe better to decompose a function in a nested form*

$$f\left(x_1, x_2, \ldots, x_d\right) = g_0\left(g_1\left(x_1, \ldots x_{d/2}\right), g_2\left(x_{d/2+1}, \ldots x_d\right)\right)$$

Example

Say $d = 4$ and we have the XOR function

$$\begin{aligned} f\left(\boldsymbol{x}\right) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 &= \underbrace{\left(x_1 \oplus x_2\right)}_{g_1(x_1, x_2)} \oplus \underbrace{\left(x_3 \oplus x_4\right)}_{g_2(x_3, x_4)} \\ &= g_1\left(x_1, x_2\right) \oplus g_2\left(x_3, x_4\right) \end{aligned}$$

Nested Decomposition of Binary Functions

Say we have g_1 and g_2 already computed: *using*

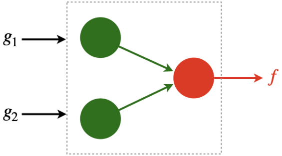
$$f(x_1, x_2, \dots, x_d) = g_0(g_1, g_2)$$

we can realize f from g_1 and g_2 via a

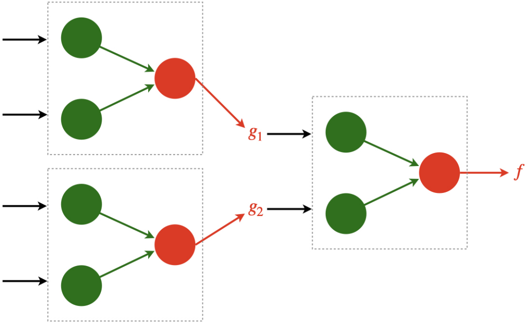
shallow FNN with 2 hidden neurons and a single output neuron

- **?** *How can we compute g_1 and g_2?*
- **!** *We can repeat this nested decomposition on them!*
  - ↳ *Decompose g_1 and g_2 as f*
  - ↳ *Realize them with a shallow network via their decomposition*
  - ↳ *Keep going backward till we get to inputs*

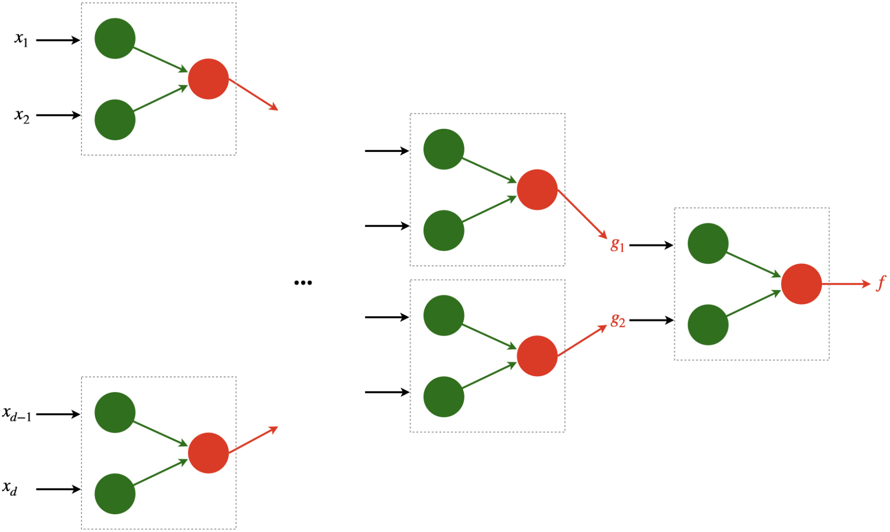
Nested Decomposition of Binary Functions



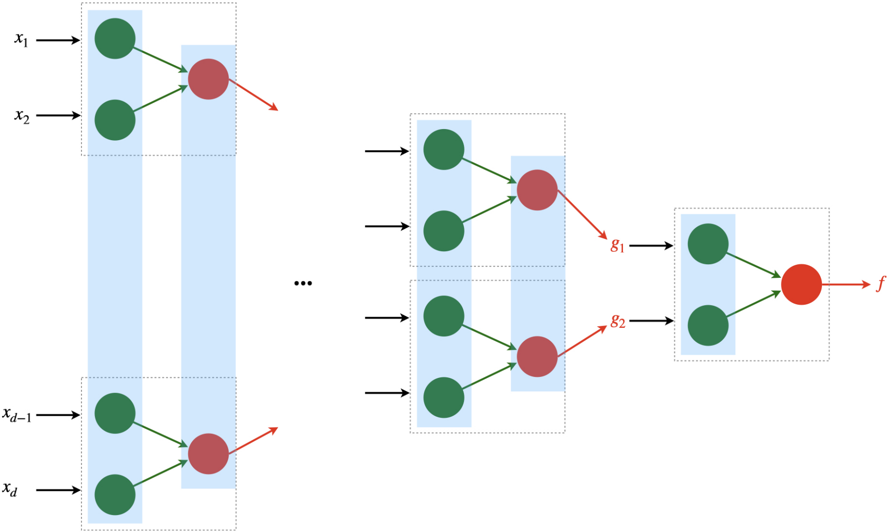
Nested Decomposition of Binary Functions



Nested Decomposition of Binary Functions



Nested Decomposition of Binary Functions



General Binary Functions

? *How many neurons do we use now?*

Conclusion

Any binary function with d inputs can be realized by a deep FNN of depth $2 \log d$ with only $3(d-1)$ neurons

Deep FNN $\equiv$ more than one hidden layer

? *That is a super reduction!*

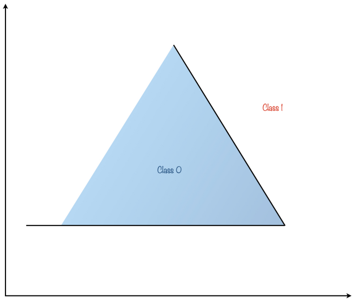
! *This is why everybody likes deep learning!*

Deep Learning $\equiv$ when we solve learning task with a deep NN

Classifications via Boolean Arguments

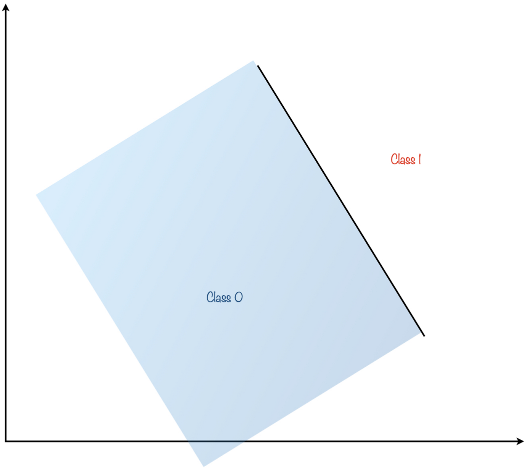
- **?** *But at the end of the day we want to do classification!*
- **!** *We can realize classification via a sequence of boolean arguments*

Say we want to learn the following classifier



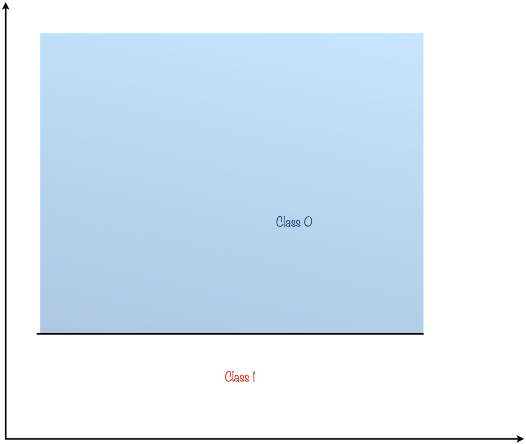
Classifications via Boolean Arguments

We can represent it by a boolean argument of the following linear classifier



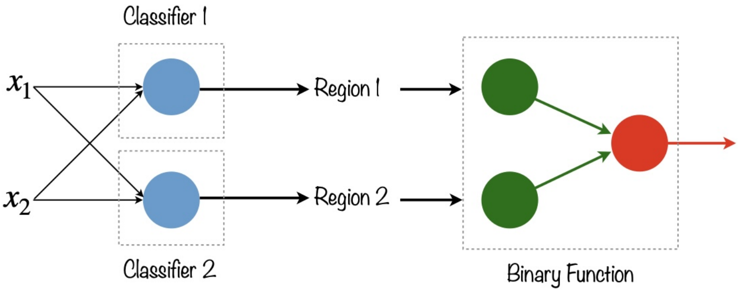
Classifications via Boolean Arguments

with this other linear classifier



Classifications via Boolean Arguments

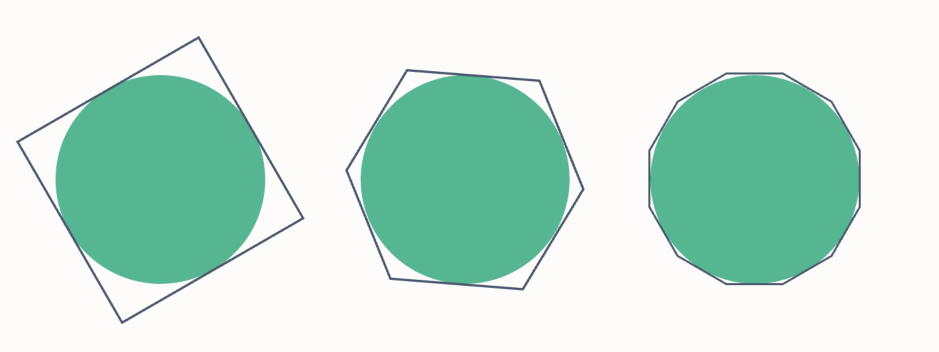
- *Each linear classifier is realized by a separate neuron*
- *The boolean argument is also realized by an NN*
  - ↳ *This is a binary function*



Classifications via Boolean Arguments

? *What of the classification regions are curved?*

! *We can approximate them accurate enough*



Universal Approximation Theorem: *Classification Use*

NNs as Universal Classifiers

For a generic classification task

- *We can always approximate the classifier via a NN*
  - ↳ *We can make approximation as accurate as we want!*
- *For a given accuracy we need to have enough neurons in the NN*
- *The deeper the NN gets, the less would be the number of required neurons*
  - ↳ *The difference could be in exponential order*

Moral of Story

We can use deep NN for pretty much any classification task!

Approximating Real Functions

? *What about regression then?!*

! *We can approximate any surface using the classifier!*

Say we have a real-valued function $f(\boldsymbol{x})$: we can do the followings

1. *Partition the space of $\boldsymbol{x}$ into M small grids $\mathbb{G}_1, \dots \mathbb{G}_M$*
2. *In grid m, take the average value of the function*

$$\bar{f}_m = \frac{\int_{\mathbb{G}_m} f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}{\int_{\mathbb{G}_m} \mathrm{d}\boldsymbol{x}}$$

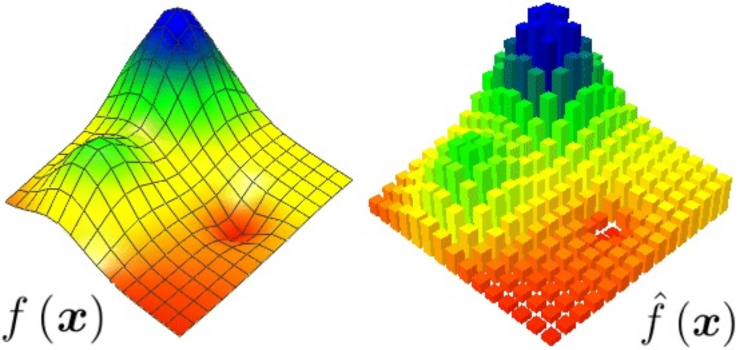
3. *For grid m, we can use an NN to build the following classifier*

$$g_m(\boldsymbol{x}) = \begin{cases} 1 & \boldsymbol{x} \in \mathbb{G}_m \\ 0 & \boldsymbol{x} \notin \mathbb{G}_m \end{cases}$$

Approximating Real Functions

We can then approximate $f(\boldsymbol{x})$ *as*

$$\hat{f}(\boldsymbol{x}) = \sum_{m=1}^{M} \bar{f}_m g_m(\boldsymbol{x})$$



Universal Approximation Theorem: *General Form*

Universal Approximation Theorem *(informal)*

For a given function $f(\cdot)$ *and* $\varepsilon > 0$, *there exists a neural network* $\hat{f}(\cdot)$ *that*

$$\sup_{\boldsymbol{x}} \| f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}) \| \leqslant \varepsilon$$

This indicates that NNs have a huge expressive power

- *They can represent any classifier*
  - ↳ *If trained well, they can learn any classification task*
- *They can approximate any real-valued function*
  - ↳ *If trained well, they can do complicated regression tasks*

Deep Learning: *Why So Late?*

The above results are known for quite some time

- *Different forms for universal approximation came out*
  - ↳ *They proved how expressive neural networks are*
  - ↳ *Some pioneer work was done by George Cybenko a UofT graduate* ☺
- *People got hope that this could solve complicated tasks*
- *The trouble was to train a neural network*
  - ↳ *We know that they can represent pretty much any function*
  - ↳ *But, how can we find the right weights?*
  - ↳ *Many concluded that "it's just a nice theory!"*
- *Some tried to minimize empirical risk with gradient descent (GD)*
  - ↳ *Backpropagation got developed to compute gradients*
  - ↳ *People did not believe backpropagation and GD would work in practice*
  - ↳ *Here at UofT, we proved them wrong in 2012* ☺

We learn how to train an NN in the next lecture

Further Read

- Bishop
  - ↳ Chapter 5: *Section 5.1* ***NNs***
- ESL
  - ↳ Chapter 11: *Sections 11.1 and 11.3* ***NNs***
- Goodfellow
  - ↳ Chapter 6: *Section 6.1 and 6.3* ***FNNs***