Ollscoil na Gaillimhe
University of Galway

Deep Learning

Topic 03: Fundamentals of Neural Networks, Part 2

Prof. Michael Madden

Chair of Computer Science
Head of Machine Learning Group
University of Galway

Learning Objectives – We Saw These Last Time

After successfully completing Topics 2 and 3,
you will be able to…

- Explain how logistic regression and stacked classifiers can be implemented as feed-forward neural networks
- Explain neural network notation and perform calculations for forward-propagation and backward-propagation
- Explain and implement the gradient descent algorithm
- Implement neural networks for supervised machine learning tasks, from first principles

In Topic 2, We Covered …

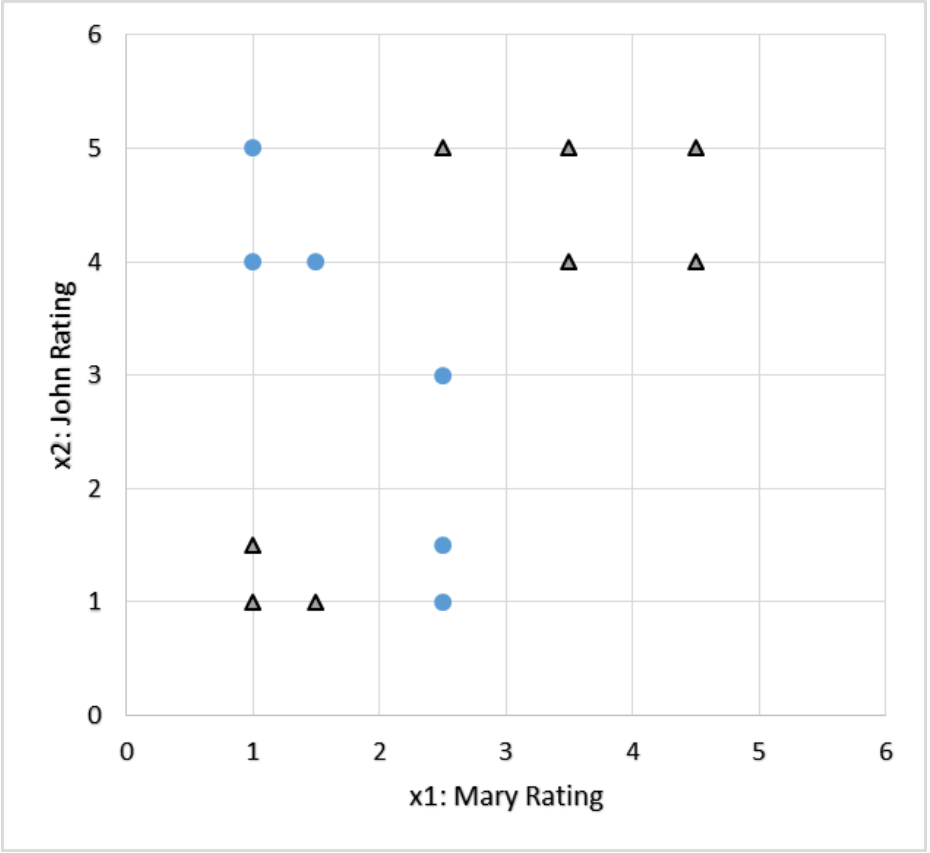
- Basic ideas of neural nets and classification
- A neural network approach to logistic regression
- Logistic Regression learning algorithm with gradient descent
- Overcoming limitations of Logistic Regression

In this Topic, We Will Cover …

- Basics of neural net calculations
- Full neural net training algorithm:
  - Overall structure of the algorithm
  - Activation functions and forward propagation
  - Backpropagation and gradient descent
- *Optional:* Explanation of partial derivative calculations
- Convergence; Design Issues; Limitations of Shallow NNs

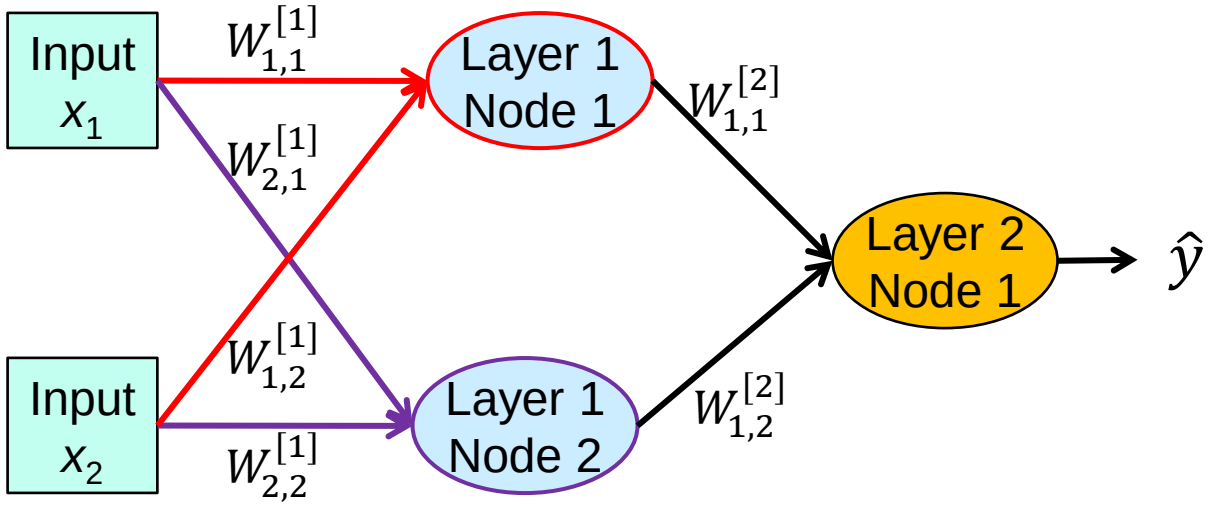
Last Time: Logistic Regression Limitations

- Classes must be linearly separable – what about this?
  - Added 3 more movies
  - I like them, but critics don't
- Can we use stacking to combine outputs of 2 separate LR classifiers?
  - Yes!
  - See "MoreMovies" tab of MovieRating.xlsx
- Essentially this is operating like a neural net with 1 hidden layer
- NOW: how to tackle this properly with NN learning.



Last Time: Combining Logistic Regressors

Simple Example of 2-Layer NN



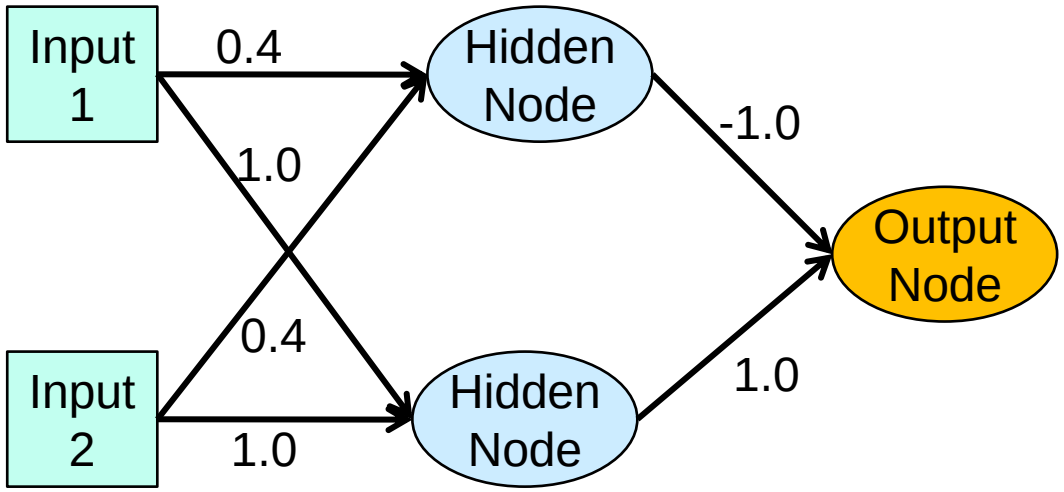
Note: No bias inputs in this simple example.

Activation of output node:

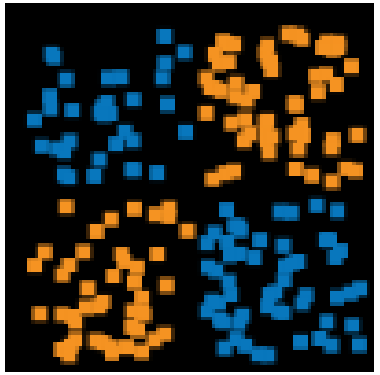
$$\hat{y} = a_1^{[2]} = f\left(W_{1,1}^{[2]}\, a_1^{[1]} + W_{1,2}^{[2]}\, a_2^{[1]}\right)$$

$$= f\left(W_{1,1}^{[2]}\, f\left(W_{1,1}^{[1]} x_1 + W_{1,2}^{[1]} x_2\right) + W_{1,2}^{[2]}\, f\left(W_{2,1}^{[1]} x_1 + W_{2,2}^{[1]} x_2\right)\right)$$

Simple Example: XOR Function

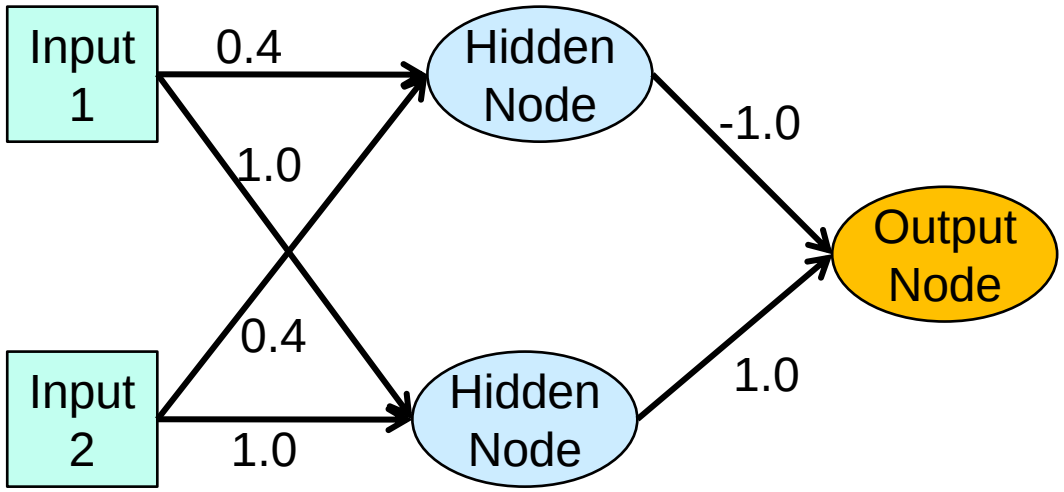


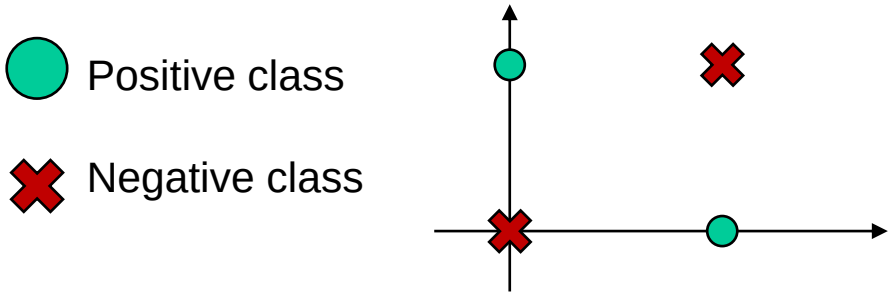
A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0



- Inputs: 0 or 1
- Output: 0 if inputs match, 1 otherwise
- Hard threshold:
  f(z) = 0 if z < 0.5
  = 1 otherwise

XOR Network: Examine the Computation





$$\hat{y} = a_1^{[2]} = f\left(W_{1,1}^{[2]}\, a_1^{[1]} + W_{1,2}^{[2]}\, a_2^{[1]}\right)$$

$$= f\left(W_{1,1}^{[2]}\, f\left(W_{1,1}^{[1]} x_1 + W_{1,2}^{[1]} x_2\right) + W_{1,2}^{[2]}\, f\left(W_{2,1}^{[1]} x_1 + W_{2,2}^{[1]} x_2\right)\right)$$

Let's compute the output value for some 0/1 inputs …

These weights were set manually: How can weights be learned?

Overall Gradient Descent with Backpropagation Algorithm for Training a Neural Network

Initialisation Step:

- Choose learning rate and other hyperparameters
- Set all ***W*** and *b* to small random values

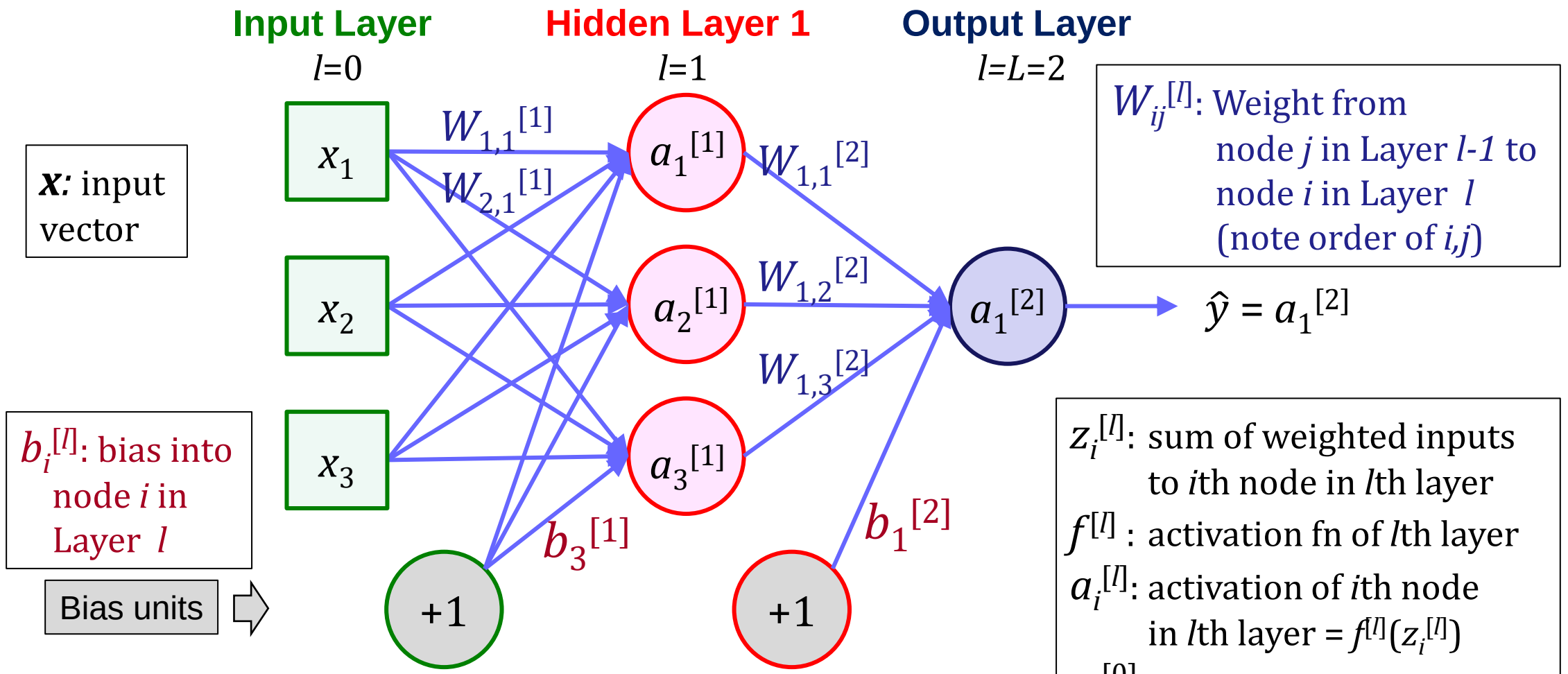
Repeat until convergence or max number of iterations:

- Select one training case at random (stochastic gradient descent)
- Forward Propagation Step:
  - Calculate output(s) for the training case
- Back Propagation Step:
  - Propagate the output errors back through the network
  - Numerically calculate the derivatives of the cost fn w.r.t. ***W*** and *b*
- Stochastic Gradient Descent Update Step:
  - Adjust ***W*** and *b* values in a direction to reduce cost as given by derivatives, by an amount controlled by the learning rate

Training a Neural Network: Terminology

- One **Epoch** = one forward pass and one backward pass of *all* the training examples
- **Batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- Number of **iterations** = number of passes, each pass using [batch size] number of examples.
  - Note: one pass = forward pass + backward pass
  - Do **not** count the forward and backward passes as two different passes

Training a Neural Network: Notation



Training Algorithm: Initialisation Step for Network Weights and Biases

- The training algorithm starts by making random guesses for all ***W*** and *b* values, and from there it tries to improve them
- As a result, when you re-run training, you can converge to different weight configurations that work equally well, and convergence can take different lengths of time
- We initialise ***W*** and *b* to **small** **random** values:
  - **Random** because if all parameters start with identical values, all the hidden layer units will end up learning the same function of the input
  - **Small** because activation functions like sigmoid "saturate" at large positive/negative values: changes to their input barely affect their output
  - Often use normally distributed random values with mean=0, stdev=0.01

Training Algorithm: Forward Propagation Step

- Forward propagation:
  - Starting with a set of **x** values, propagate them forward through the neural network to get the output value(s)
  - The result depends on the current weights and biases

- A fundamental calculation:
  - When training a neural net, it is the first step before updating weights
  - When a neural net is deployed into an application, the outputs of the net from forward propagation are its predictions.

Forward Propagation Calculations

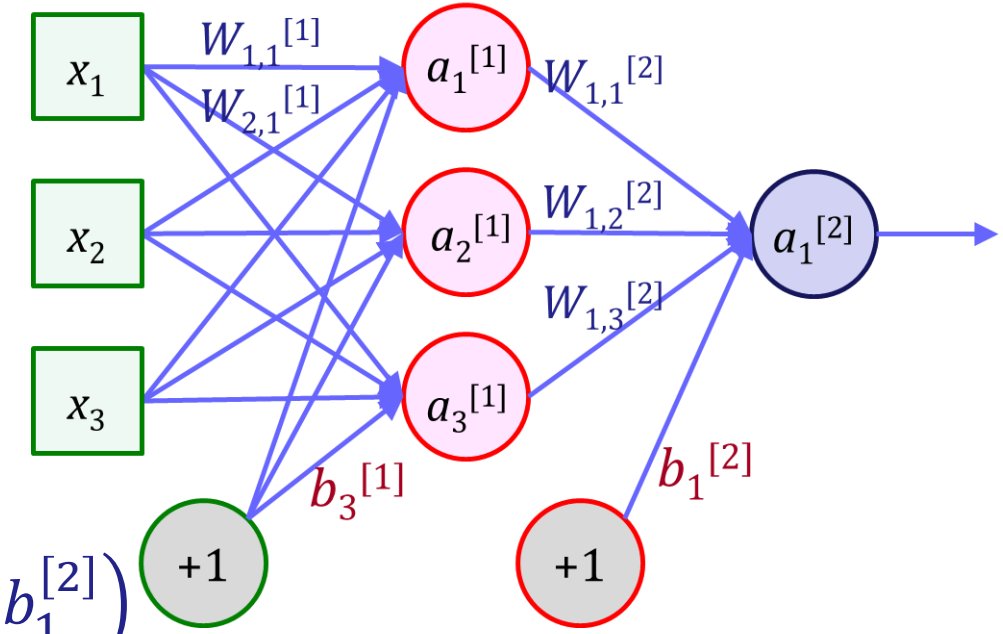
Computation performed by the neural net shown previously:

$$a_1^{[1]} = f^{[1]}\left(W_{1,1}^{[1]}x_1 + W_{1,2}^{[1]}x_2 + W_{1,3}^{[1]}x_3 + b_1^{[1]}\right)$$

$$a_2^{[1]} = f^{[1]}\left(W_{2,1}^{[1]}x_1 + W_{2,2}^{[1]}x_2 + W_{2,3}^{[1]}x_3 + b_2^{[1]}\right)$$

$$a_3^{[1]} = f^{[1]}\left(W_{3,1}^{[1]}x_1 + W_{3,2}^{[1]}x_2 + W_{3,3}^{[1]}x_3 + b_3^{[1]}\right)$$

$$\hat{y} = a_1^{[2]} = f^{[2]}\left(W_{1,1}^{[2]}\,a_1^{[1]} + W_{1,2}^{[2]}\,a_2^{[1]} + W_{1,3}^{[2]}a_3^{[1]} + b_1^{[2]}\right)$$



Let's see the general case of these …

Forward Propagation Calculations: General Case

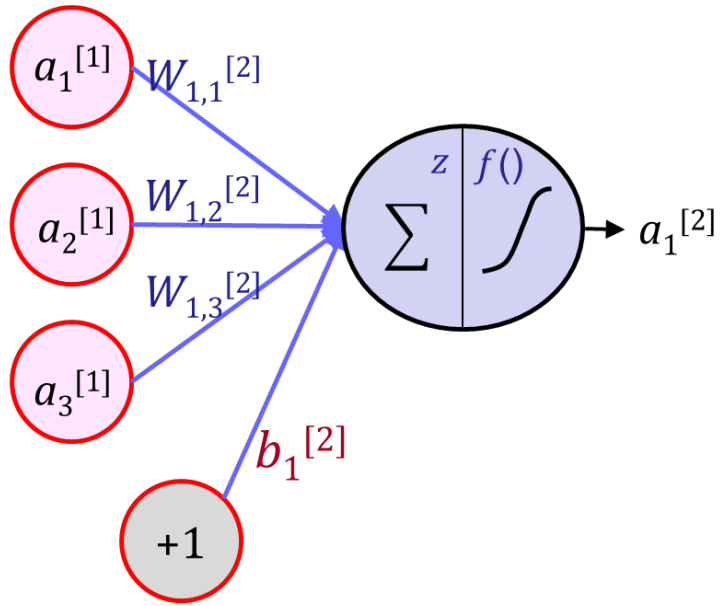
- We define $z_i^{[l]}$ as sum of inputs into node i in layer l:

$$z_i^{[l]} = \sum_{j=1}^{n} W_{ij}^{[l]} a_j^{[l-1]} + b_i^{[l]}$$

- Then: $a_i^{[l]} = f^{[l]}(z_i^{[l]})$

  is the output (activation) of node i in layer l

  Where $f^{[l]}()$ is the activation function for layer l



- These equations apply to all layers:
  - We treat the inputs as layer number 0, so $a_i^{[0]} \equiv x_i$
  - Output values are $\hat{y}$, and output layer number is L, so $a_i^{[L]} = \hat{y}_i$

Common Activation Functions

Logistic: $f(z) = \frac{1}{1+e^{-z}}$ $\quad f'(z) = f(z)\big(1-f(z)\big)$

Tanh: $f(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ $\quad f'(z) = 1-f(z)^2$

ReLU: $f(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$ $\quad f'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$

Leaky ReLU: $f(z) = \begin{cases} 0.01z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$ $\quad f'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$

Cost Function for Gradient Descent

- For classification tasks, we will use the same **average log loss** cost function that we saw last week for logistic regression:

$$J_{\boldsymbol{W},b}(y,\hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right)$$

Where $\hat{y} = a_1^{[L]}$, the output from the single node in the final layer, and *N* is the number of training cases

- Since we are using the Stochastic Gradient Descent algorithm, which operates on one training case at a time, this simplifies to:

$$J_{\boldsymbol{W},b}(y,\hat{y}) = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

Training Algorithm: A Single Backpropagation Step and Gradient Descent Update Step

In the forward propagation step of this iteration, have already calculated $a_i^{[l]}$ and $z_i^{[l]}$ for all nodes in all layers.

Backpropagation Step - Calculate the following gradients, where terms such as Δz denote partial derivatives of cost function w.r.t. z: $\Delta z = \partial J / \partial z$

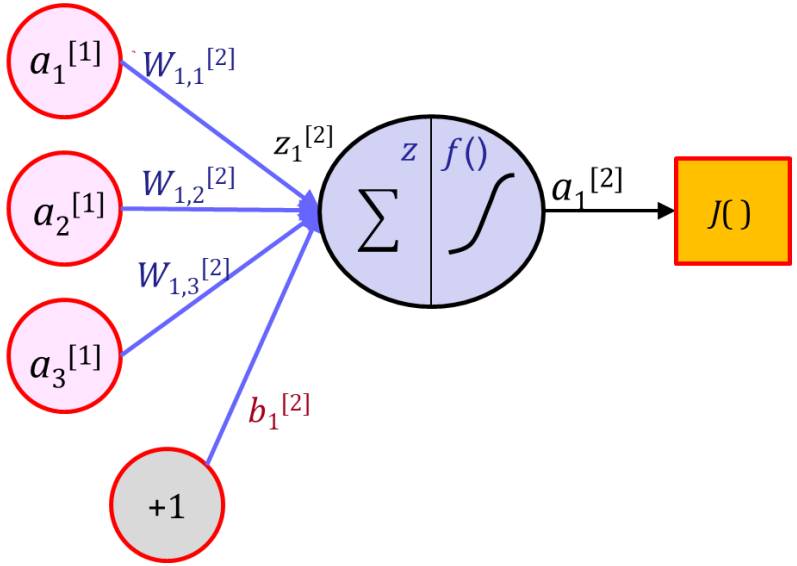
> Output Layer: First: $\Delta z_i^{[L]} = a_i^{[L]} - y_i$

Then: $\Delta W_{1,i}^{[L]} = \Delta z_1^{[L]} \, a_i^{[L-1]}$ and $\Delta b_1^{[L]} = \Delta z_1^{[L]}$

> Hidden Layers: Loop backwards from L-1 to 1:

First: $\Delta z_i^{[l]} = f'\left(z_i^{[l]}\right) \sum_j \left(\Delta z_j^{[l+1]} \, W_{j,i}^{[l+1]}\right)$

Then: $\Delta W_{j,i}^{[l]} = \Delta z_j^{[l]} \, a_i^{[l-1]}$ and $\Delta b_j^{[l]} = \Delta z_j^{[l]}$



Gradient Descent Update Step using the gradients:

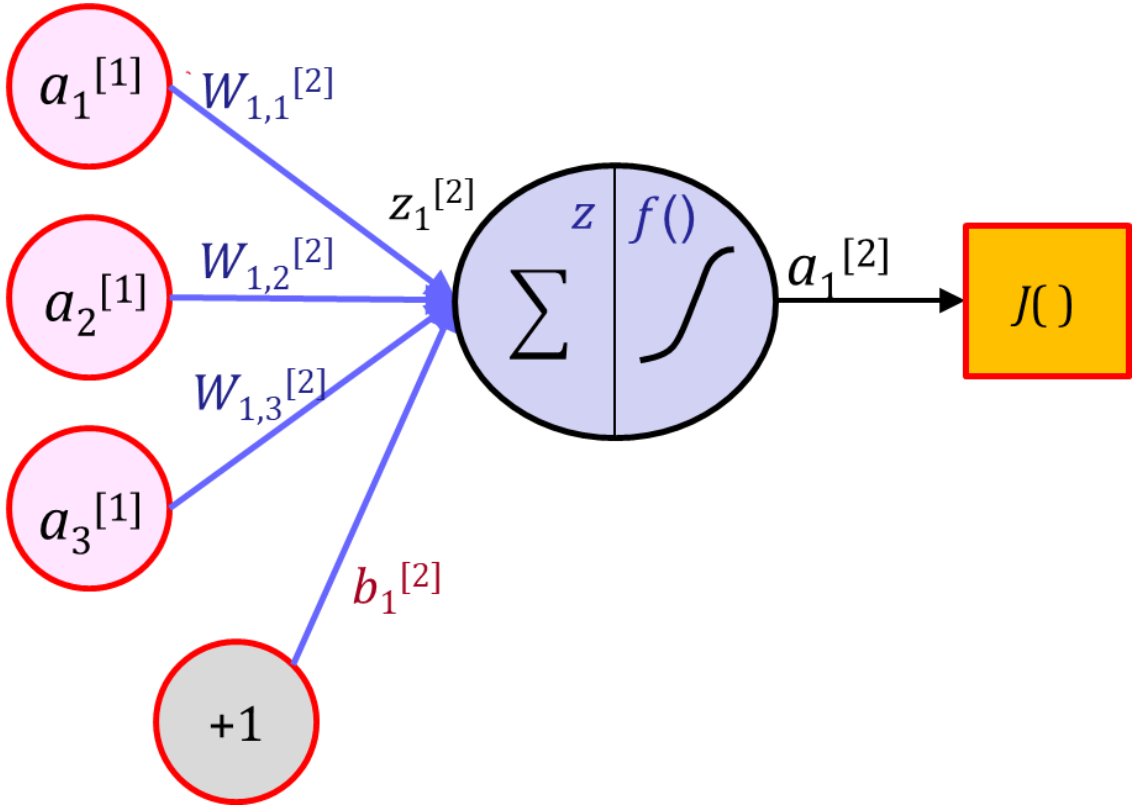
Loop over all weights and all biases:

$W_{j,i}^{[l]} \mathrel{-}= \alpha \, \Delta W_{j,i}^{[l]}$ and $b_j^{[l]} \mathrel{-}= \alpha \, \Delta b_j^{[l]}$

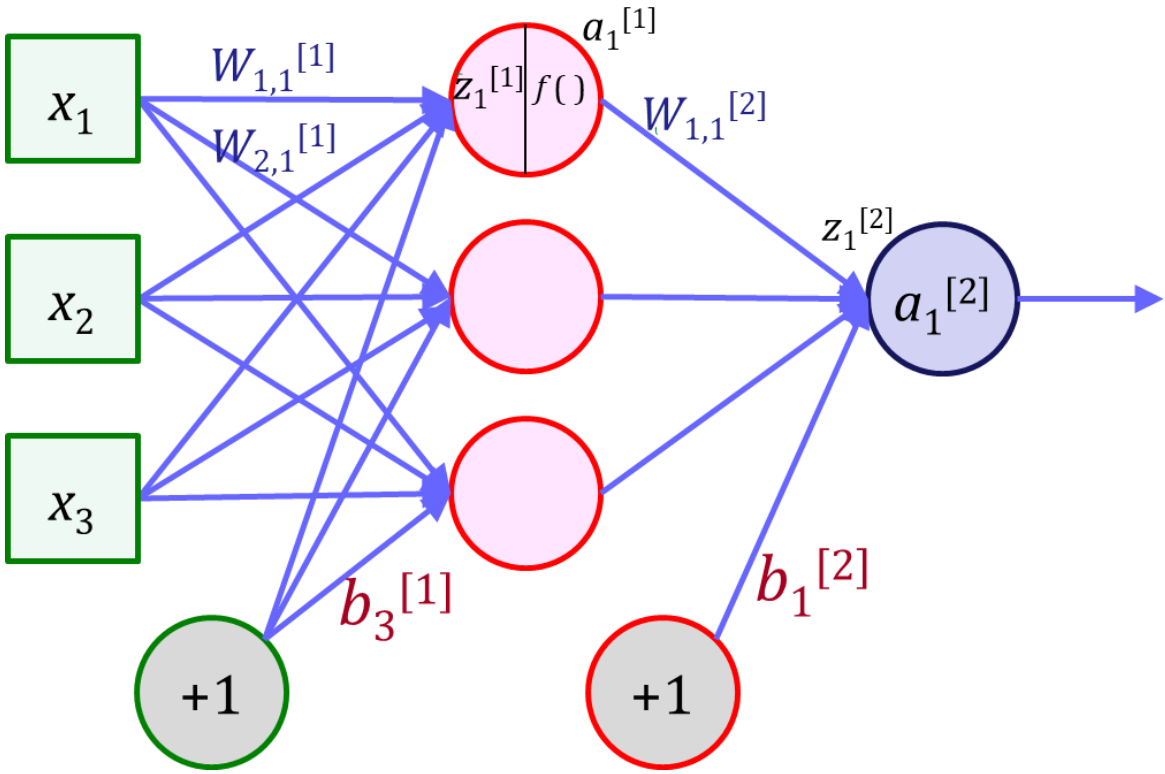
i is index of nodes in layer *l*
j is index in layer *l*+1

Optional: Explanation of Partial Derivative Calculations – Output Layer

$$J\left(a_1^{[L]}, y\right) = -\left(y \log\left(a_1^{[L]}\right) + (1-y)\log\left(1-a_1^{[L]}\right)\right)$$



Optional: Explanation of Partial Derivative Calculations – Hidden Layer



Training Algorithm Convergence

- Unlike the linear case such as logistic regression, this is **not a convex optimisation problem**
  - Gradient descent converges to local minimum: not necessarily global
  - For new tasks, often watch progression of training and manually monitor for convergence
- Signs of convergence:
  - Little or no change in weight values after multiple iterations
  - Loss function computed on all training data stops decreasing
  - Loss function computed on held-out data may even start to increase
- Important if using Stochastic or Mini-Batch GD:
  - Change in loss between two iterations, or change in weight values, can be erratic
  - Aggregate statistics over one epochs' worth of iterations

Training Algorithm Convergence

- To monitor the progress of learning, plot a Training Curve: Error vs. Epochs
  - When error plateaus, it's time to stop
  - On held-out data, it may get worse, indicating overfitting
- Experiment with learning rate:
  - if too low or too high, convergence will be affected
  - Some approaches adjust learning rate during training

Training Algorithm: More Details

- Andrew Ng's team in Stanford put together good tutorials on this:
  - Neural net notation and Backprop algorithm: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/
  - Advice on error checking & debugging backprop: http://ufldl.stanford.edu/tutorial/supervised/DebuggingGradientChecking/
- If you find other sources of information that help you understand these concepts, please use the discussion forum to share them with other students!

What Functions Can Feed-Forward NN Represent?

- No hidden layers (Perceptron)
  - Linearly separable functions: AND, OR, NOT
  - Cannot represent XOR [Minsky & Papert, 1969]
- One hidden layer [with **unlimited number of nodes**]:
  - All continuous functions can be approximated with arbitrarily small error
  - Provided that the activation function is non-linear
- Two hidden layers: All functions
- Using deeper networks can have advantages
  - Able to represent functions of the same complexity with a network of smaller overall size
  - Other advantages, as will be discussed in next topic

To represent any n-input Boolean Function: $2^n/n$ hidden nodes

Slide from CS188 Intro to AI at UC Berkeley, used with permission.

Universal Function Approximation Theorem

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j), \qquad (1)$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^T is the transpose of y so that $y^T x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

Neural Networks, Vol. 4, pp. 251–257, 1991
Printed in the USA. All rights reserved.

0893-6080/91 $3.00 + .00

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Technische Universität Wien, Vienna, Austria

(*Received* 30 *January* 1990; *revised and accepted* 25 *October* 1990)

Abstract—*We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to $L^p(\mu)$ performance criteria, for arbitrary finite input environment measures μ, provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonconstant, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.*

Keywords—Multilayer feedforward networks, Activation function, Universal approximation capabilities, Input environment measure, $L^p(\mu)$ approximation, Uniform approximation, Sobolev spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural network architectures have recently been investigated by many authors, including Carroll and Dickinson (1989), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989, 1990), Irie and Miyake (1988), Lapedes and Farber (1988), Stinchcombe and White (1989, 1990). (This list is by no means complete.)

If we think of the network architecture as a rule for computing values at l output units given values at k input units, hence implementing a class of mappings from R^k to R^l, we can ask how well arbitrary mappings from R^k to R^l can be approximated by the network, in particular, if as many hidden units as required for internal representation and computation may be employed.

How to measure the accuracy of approximation depends on how we measure closeness between functions, which in turn varies significantly with the specific problem to be dealt with. In many applications, it is necessary to have the network perform *simultaneously* well on all input samples taken from some compact input set X in R^k. In this case, closeness is measured by the uniform distance between functions on X, that is,

$$\rho_{\infty,X}(f, g) = \sup_{x \in X} |f(x) - g(x)|.$$

In other applications, we think of the inputs as random variables and are interested in the *average performance* where the average is taken with respect to the input environment measure μ, where $\mu(R^k) < \infty$. In this case, closeness is measured by the $L^p(\mu)$ distances

$$\rho_{p,\mu}(f, g) = \left[\int_{R^k} |f(x) - g(x)|^p \, d\mu(x) \right]^{1/p},$$

$1 \le p < \infty$, the most popular choice being $p = 2$, corresponding to mean square error.

Of course, there are many more ways of measuring closeness of functions. In particular, in many applications, it is also necessary that the *derivatives* of the approximating function implemented by the network closely resemble those of the function to be approximated, up to some order. This issue was first taken up in Hornik et al. (1990), who discuss the sources of need of smooth functional approximation in more detail. Typical examples arise in robotics (learning of smooth movements) and signal processing (analysis of chaotic time series); for a recent application to problems of nonparametric inference in statistics and econometrics, see Gallant and White (1989).

All papers establishing certain approximation ca-

Requests for reprints should be sent to Kurt Hornik, Institut für Statistik und Wahrscheinlichkeitstheorie, Technische Universität Wien, Wiedner Hauptstraße 8-10/107, A-1040 Wien, Austria.

MULTILAYER FEEDFORWARD NETWORKS WITH NON-POLYNOMIAL ACTIVATION FUNCTIONS CAN APPROXIMATE ANY FUNCTION

by

Moshe Leshno
Faculty of Management
Tel Aviv University
Tel Aviv, Israel 69978

and

Shimon Schocken
Leonard N. Stern School of Business
New York University
New York, NY 10003

September 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-26

Appeared previously as *Working Paper No. 21/91* at The Israel Institute Of Business Research

Cybenko (1989): "Approximations by superpositions of sigmoidal functions"
Hornik (1991): "Approximation Capabilities of Multilayer Feedforward Networks"
Leshno and Schocken (1991): "Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function"

Design Issues: Topology

- How many hidden layers and nodes/layer?
  - Depends on target function complexity (not much help!)
  - Common first guess: one layer with ½ nodes of input layer
  - Specialised topologies and error measurements may be appropriate to problem domain: better performance
- Too few layers/nodes: cannot approximate target well
- Too many: risk of overfitting
  - Network may memorise training data
  - Particularly if more parameters than training examples
  - Less of a problem with large data sets
- Should network be fully connected?
  - Sparse networks can have advantages

Some Problems with Classic Feed-Forward Neural Nets

- High number of "degrees of freedom"
  - Easy to get stuck in local minimum when training
  - Re-running learning can lead to v. different outcomes
- Large nets need lots of training
  - Training can take a very long time
- More layers can reduce overall number of nodes
  - But backprop runs into problems in deep architectures
  - Credit not always assigned fully back through all layers
  - Early layers in particular tend to converge to 1/0 weights
- Need lots of data
  - Labelled data can be expensive to come by
  - Traditionally, people were used to dealing with relatively small problems

Coming Up Next …

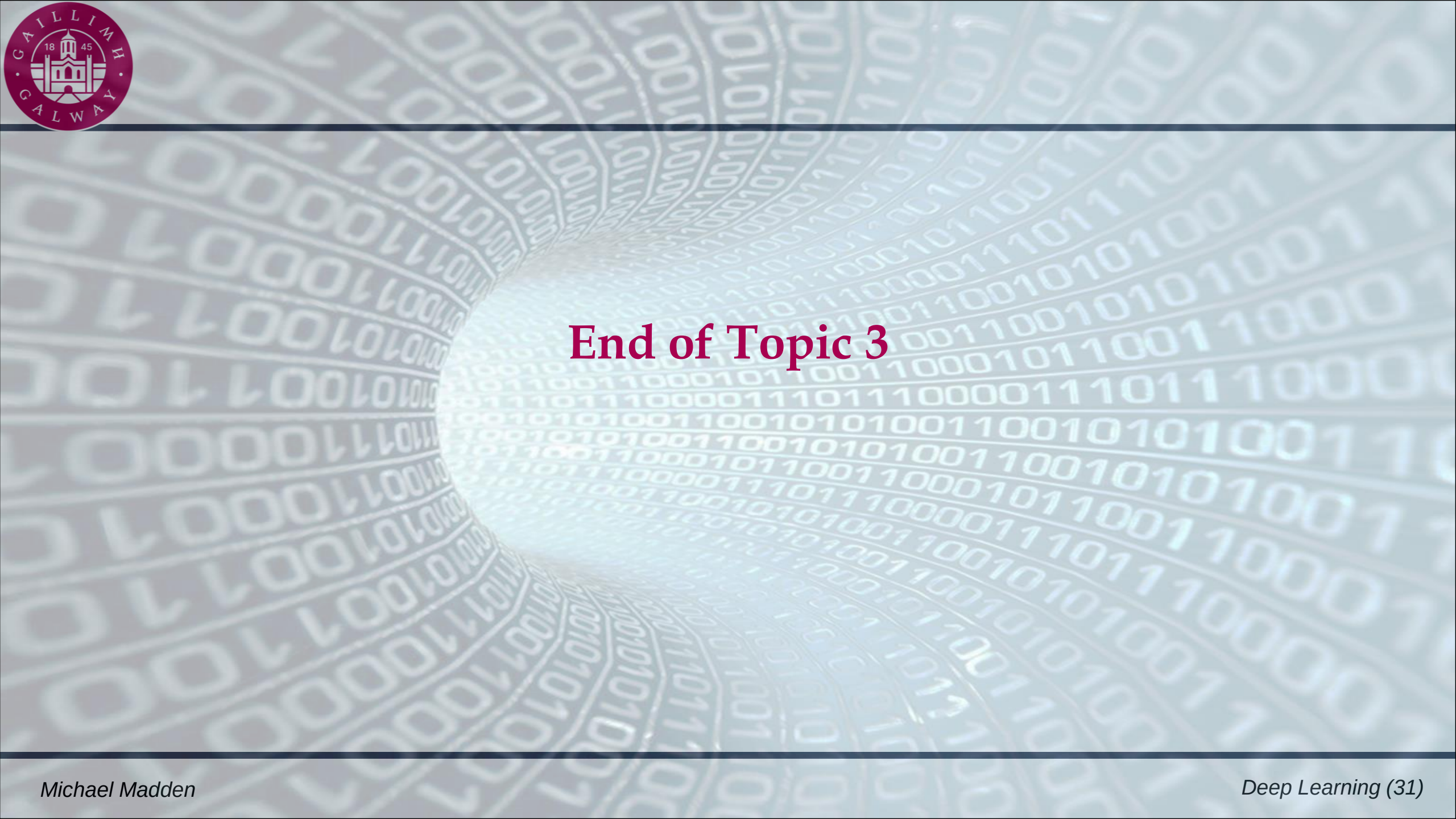
- Neural networks and the backprop algorithm form the basis of Deep Neural Networks, which we will discuss in the next topic.

- They include a variety of advances that overcome many of the problems identified on the previous slide.

Review of Learning Objectives

After having successfully completing Topics 2 and 3, you should now be able to …

- Explain how logistic regression and stacked classifiers can be implemented as feed-forward neural networks
- Explain neural network notation and perform calculations for forward-propagation and backward-propagation
- Explain and implement the gradient descent algorithm
- Implement neural networks for supervised machine learning tasks, from first principles



End of Topic 3