<div style="border:1px solid black;">

MATH5836: Data and Machine Learning

Week 3: Introduction to Neural Networks

Sarat Moka   *UNSW, Sydney*

</div>

Key Topics

<div style="border:1px solid black;">

Books:

(A) *Mathematical Engineering of Deep Learning* book by Liquet, Moka, and Nazarathy (2024). Click here for downloading chapters.

(B) Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Ed.) by Aurélien Géron (2019).

</div>

3.1 Tasks and Architectures

To get a feel for the models and methods covered, we now present an overview of the key tasks and architectures. Figure 3.1 presents schematics of different types of neural networks. We now briefly discuss about some of these networks, namely, feedforward fully connected neural network, convolutional neural network, recurrent neural network, generative models, and deep reinforcement learning.

> **Remark 3.1.1**
>
> Note that in this course we will cover feedforward fully connected networks in details. We will not have time cover other types networks in details. If you are interested, I suggest you to check out the book references provided.

Feedforward Fully Connected Neural Network

- The most basic deep neural network is the feedforward fully connected neural network; see Figure 3.1 (a).

- Simple special cases of this network are the linear model and logistic regression (sigmoid), both covered in Week 1.

- Mathematically, feedforward fully connected neural networks are simply combinations of affine (linear) transformations and non-linear activation functions, similar to logistic regression model.

- This enhancement gives the model, $f_\theta(\cdot)$, an incredible ability to express complex relationships, $y = f_\theta(x)$, while supporting an algorithmically tractable way of finding θ (fitting or training).

- Classically these models are also called multi-layer perceptron since they are descendants of the first ever neural network model, the perceptron, developed by Frank Rosenblatt in the late 1950's.

- This architecture is useful for tasks such as classification, regression, or feature extraction.

- Components of these networks, called fully connected layers, can also be components of more complex architectures such as convolutional networks, transformer models, and others.

- Understanding training of these feedforward fully connected architectures, where gradients are computed via the famous *backpropagation algorithm* is key to understanding the essence of deep learning.

Convolutional Neural Networks

- The VGG19 model is one example of a convolutional neural network. This class of models is illustrated in Figure 3.1 (c).
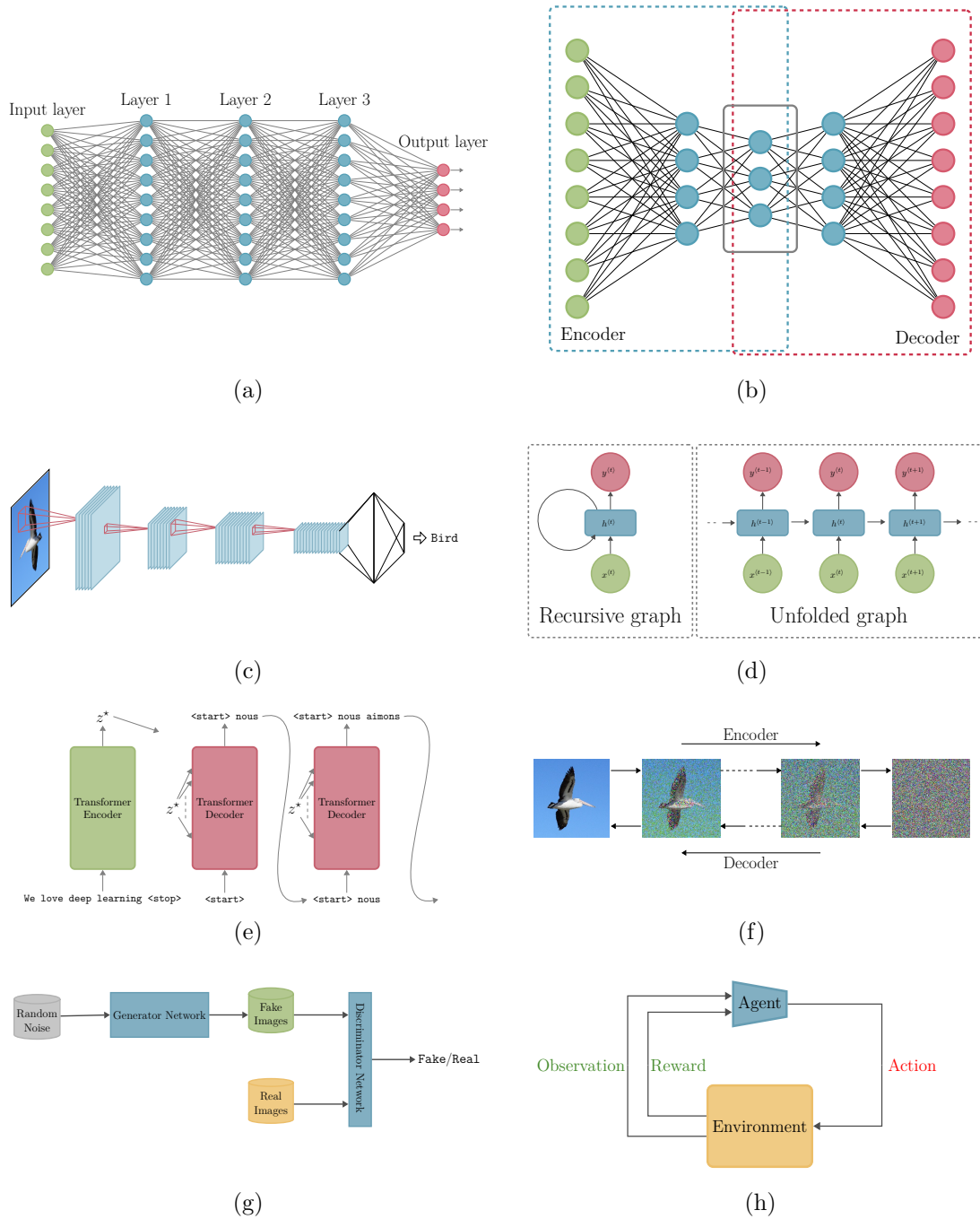
Figure 3.1: Illustrations of some common deep learning architectures and paradigms: (a) Feedforward fully connected neural networks. (b) Autoencoders with shallow versions. (c) Convolutional neural networks. (d) Recurrent neural networks. (e) Transformer architectures with an encoder and repeated application of a decoder. (f) Diffusion models for image generation. (g) Generative adversarial networks. (h) Reinforcement learning.

- Convolutional models are primarily used for image analysis and it is fair to say that their recent success has shuffled the cards in the broad field of image processing.

- Beyond images, these models can be adapted for other domains such as radiology data or audio.

- In the context of images, there are multiple related tasks including classification, semantic segmentation, and localization, and all of this can be handled via convolutional neural networks.

- Convolutional neural networks can be viewed as adaptations of fully connected networks, where the action of each layer is not based on the full connections between activations but rather on smaller trainable convolutions.

- Such a setup significantly reduces the number of parameters, enables deeper architectures, and most importantly capitalizes on spatial relationships present in the input.

- This results in training that is much more efficient and the model is more efficient in production as well.

Recurrent Neural Networks, LSTMs, and GRUs

- Figure 3.1 (d) illustrates a recurrent neural network where on the left side of (d) we see the basic architectural components and on the right side we see what is called an unfolded representation of the network, illustrating its recursive operation.

- While key advances during 2010–2015 were in the convolutional domain focusing on images, the second half of that decade witnessed deep learning becoming an integral part of *natural language processing (NLP)*.

- By now, automatic translation engines, language generation models, and other solutions for tasks associated with text almost always involve deep neural networks or are entirely based on deep learning models. The use of recurrent neural networks is the most rudimentary modeling paradigm for such purposes.

- There are multiple variations of recurrent neural networks where the most basic one is illustrated in Figure 3.1 (d).

- However, the internal structure can vary and some popular and powerful variations include long short term memory (LSTM) models and gated recurrent unit (GRU) models.

- In addition to NLP, there are many other domains where such models for sequence data is a natural choice including genomic sequencing, multivariable time series, audio, and even video.

Generative Models

- A generative model in deep learning refers to a type of model that learns to generate new data points that resemble a given dataset.

- There are mainly two popular deep learning generative models: *diffusion models* and *generative adversarial network (GAN)*.

- In Figure 3.1 (f) we see a schematic of a diffusion model which here simply appears as a process of either adding noise to an image in an encoder or alternatvily removing noise from an image with a decoder.

- The overarching idea of a diffusion model is to learn not just how to add noise, but also how to create an image out of noise. With this, a trained decoder can generate realistic looking images that are actually random.

- Diffusion models and their generalizations are probabilistic in nature.

- A *generative adversarial network (GAN)* (GAN) architecture is illustrated in Figure 3.1 (g).

- Like diffusion models, GANs are very useful for creating random data that is realistic in nature.

- The rise and popularity of GANs predated that of diffusion models and today GANs and diffusion models compete for the state of the art in artificial data generation. GANs and diffusion models differ in their architecture and analysis.

- While diffusion models are probabilistic, the analysis and study of GANs is close to the field of game theory.

- The key idea of a GAN is to simultaneously train two deep neural networks, a generator and a discriminator. The former generates fake data, while the latter attempts to determine if the data is `fake` or `real`. As the training of both of these networks progresses, the generator is ultimately able to fool the discriminator and as a consequence, it also creates "real looking" data.

Deep Reinforcement Learning

- In Figure 3.1 (h) we illustrate the paradigm of reinforcement learning.

- Here the basic setup is that a system, or environment, is controlled by an agent.

- For example, one may think of the environment as a home, and the agent as a cleaning robot traversing and cleaning the home. As time progresses, the agent makes decisions in the form of actions, for example `move right 5 cm`, and these are interfaced with the environment. The agent in turn receives reward from the environment as well as observations, where the reward is a mechanism that helps to drive towards better goals, for example "cleaning in a quick and energy efficient manner", and the observations can include sensory input.

- In general, the goal of reinforcement learning is to develop meaningful ways for the agent to choose actions.

- One of the great leaps of AI during the second decade of this century is in the game of Go. This strategic board game was long considered much more difficult "to program" in comparison to other games such as Chess[1].

- Yet in 2015 a team from DeepMind through a series of advances and competitions designed a system called AlphaGo which beat the world's best Go players.

- This highly publicized achievement made the dream of artificial intelligence a bit more concrete by showing the ability of neural networks to solve complicated tasks. The key ideas of this achievement are from the field of reinforcement learning.

3.2   Neuron: Motivation for neural networks

- Brains are composed of (biological) neurons that are interconnected in unstructured ways; see Figure 3.2 where display (a) illustrates a single biological neuron and display (c) illustrates an interconnected network of biological neurons. A human brain has an estimated 85 billion neurons.

- A single human action such as movement of an arm may induce the firing of around 80 million such neurons, whereas the identification of a visual object may use the bulk of the estimated 150 million neurons that are in the visual cortex.

- Deep neural network models are neither brains nor attempts to create artificial brains. Nevertheless, the development of these models is highly motivated by the biological structure of the brain.

- The basic building block of a deep neural network model is the (artificial) neuron abstracting the synapse connection between neurons via a single number called an activation value.

- See display (b) of Figure 3.2 for a single (artificial) neuron and display (d) which presents a combination of multiple neurons as part of a feedforward (artificial) neural network.

- Pioneering and landmark work in AI research was inspired by neuroscience since brains are essentially the only complete proof we have for the existence of what we call "general intelligence".

- Further, many tasks of deep learning models involve the mimicking of human level (or animal level) tasks such as understanding images or conversational tasks. Thus for example one of the most well-known benchmarks in the world of artificial intelligence is the Turing test, originally named the imitation game when introduced by Alan Turing in 1950.

- It is essentially a test to see if a computer can engage in long conversation with a human, without another observing human distinguishing between the computer and the human.

---

[1]Computers have shown their superiority in the game of Chess since the mid 1990's with a notable victory of the Deep Blue Chess playing expert system defeating the champion Garry Kasparov over a six-game match in 1996.

> **Remark 3.2.1**
>
> An interesting aspect of biological neurons is their ability to generate complex electrical signals known as action potentials or spikes. These spikes play a crucial role in neural communication and information processing in the brain. Unlike artificial neurons in deep learning, which typically operate using continuous activation functions and numerical computations, biological neurons exhibit discrete, nonlinear firing patterns characterized by the generation and propagation of action potentials along their axons.
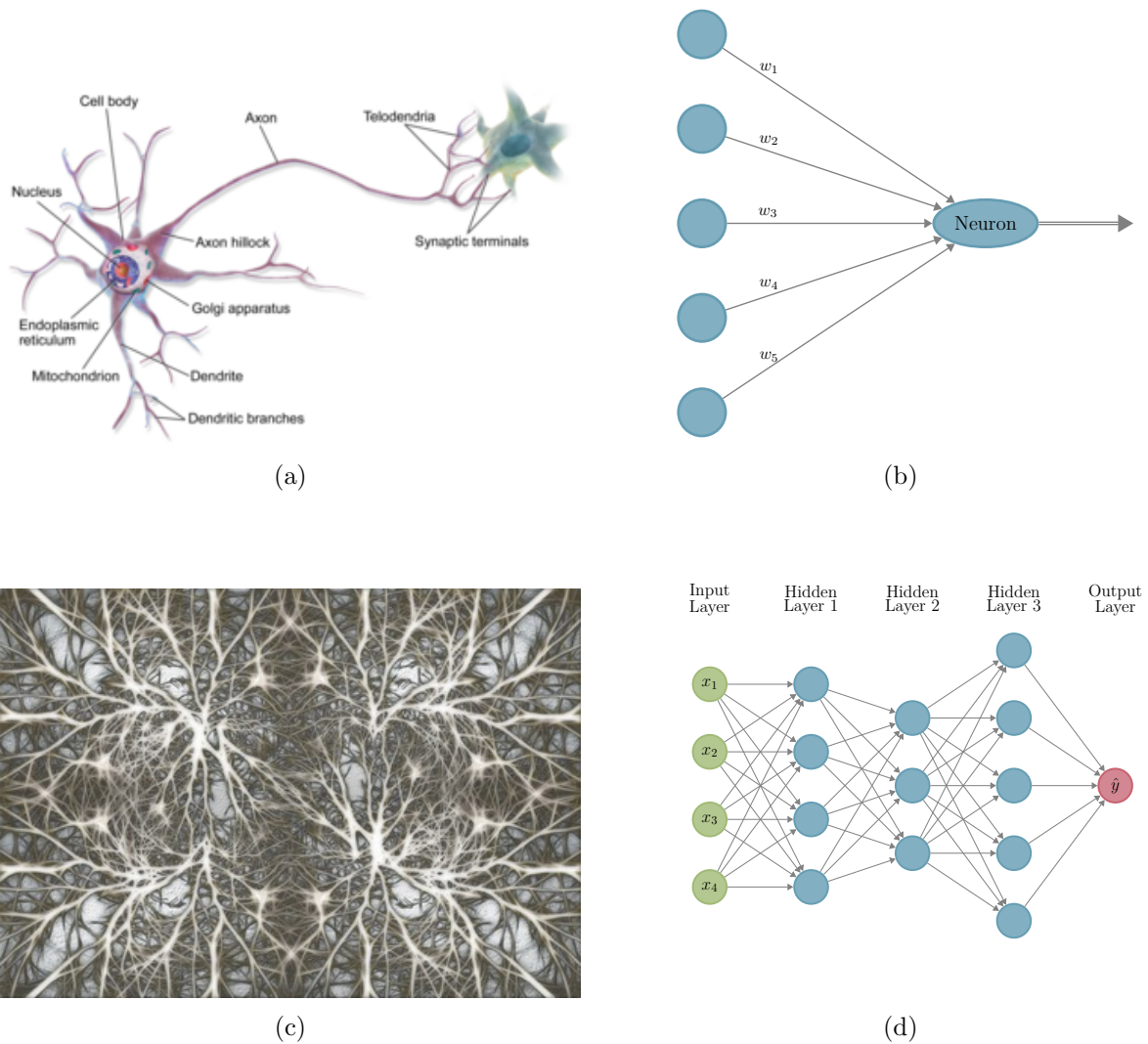

(a)

(b)

(c)

(d)

Figure 3.2: Biological and artificial neurons and networks.[2] (a) A single biological neuron. (b) A neuron in an artificial neural network. (c) Connection of multiple biological neurons in a brain. (d) An artificial neural network connecting multiple neurons in a feedforward structured manner.

3.3 General Feedforward Neural Networks

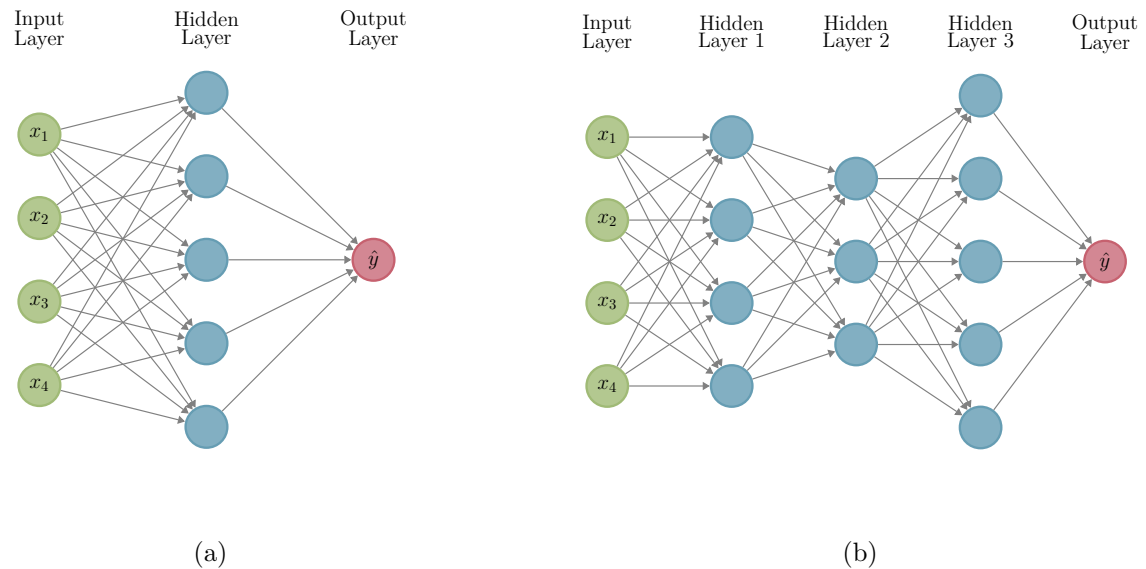(a)   (b)

Figure 3.3: Fully connected feedforward neural networks. (a) A network with a single hidden layer. (b) A deep neural network with multiple hidden layers.

- In the previous section, we briefly illustrated *feedforward fully connected neural networks*. Such a model can also be called a *fully connected network*, a *feedforward network*, or a *dense network* where each of these terms may also be augmented with the phrases "deep", "neural", and "general". Another name for such a model is a *multi-layer perceptron* (MLP) or a *multi-layer dense network*.

- Schematic illustrations of this architecture are presented in Figure 3.3. In that figure, each circle may be called a *neuron* or *unit* and each vertical set of neurons is a *layer*.

- The left most layer is called the *input layer* and it has the input features vector $\boldsymbol{x}$.

- The right most or *output layer* contains the output neurons (just one in this example).

- The layers in the middle are called *hidden layer*, since the neurons in these layers are neither inputs nor outputs. When using the network in production for prediction (regression or classification), we do not directly observe what goes on in the hidden layers, but rather observe network outputs resulting from inputs.

A Model Based on Function Composition

- The goal of a feedforward network is to approximate some function $f^* : \mathbb{R}^p \longrightarrow \mathbb{R}^q$.

- A feedforward network model defines a mapping $f_\theta : \mathbb{R}^p \longrightarrow \mathbb{R}^q$ and learns the value of the parameters θ that ideally result in

$$f_\theta(x) \approx f^*(x).$$

- The function f_θ is recursively composed via a chain of functions:

$$f_\theta(x) = f_{\theta[L]}^{[L]}(f_{\theta[L-1]}^{[L-1]}(\dots (f_{\theta[1]}^{[1]}(x))\dots)), \tag{3.1}$$

  where $f_{\theta[\ell]}^{[\ell]} : \mathbb{R}^{N_{\ell-1}} \longrightarrow \mathbb{R}^{N_\ell}$ is associated with the ℓ-th layer which depends on parameters $\theta^{[\ell]} \in \Theta^{[\ell]}$, where $\Theta^{[\ell]}$ is the parameter space for the ℓ-th layer.

- The *depth* of the network is L. We have that $N_0 = p$ (the number of features) and $N_L = q$ (the number of output variables).

- The number of neurons in the network is $\sum_{\ell=1}^{L} N_\ell$.

- Note that in case of networks used for classification we typically have $q = K$, the number of classes.

Affine Transformations Followed by Activations

- In deep learning, the function $f_{\theta[\ell]}^{[\ell]}$ is generally defined by an affine transformation followed by an activation function.

- Activation functions are the means of introducing non-linearity into the model. We have seen one such activation function in case logistic regression, where it is called sigmoid function; see Figure 3.4.
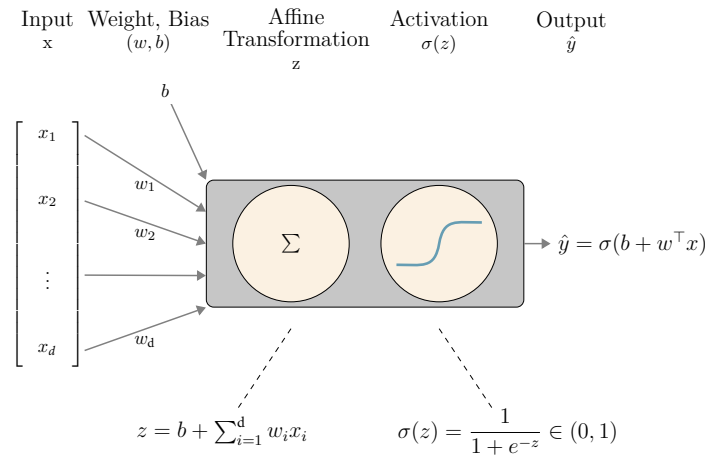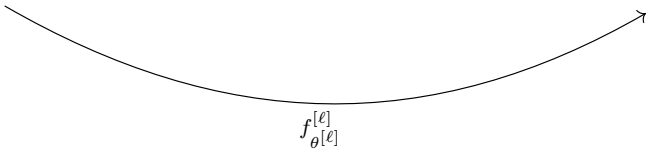

Figure 3.4: Logistic regression represented with neural network terminology as a shallow neural network. The gray box represents an artificial neuron composed of an affine transformation to create z and an activation $\sigma(z)$.

- The output of layer ℓ in feedforward network is represented by the vector $a^{[\ell]}$ and the intermediate result of the affine transformation is represented by the vector $z^{[\ell]}$.

- We typically denote the output of the model via $\hat{y}$ and hence,

$$\hat{y} = a^{[L]} = f_\theta(x).$$

- The action of $f_{\theta^{[\ell]}}^{[\ell]}$ can be schematically represented as follows,

$$a^{[\ell-1]} \xrightarrow{\overset{\text{Affine}}{\text{Transformation}}} z^{[\ell]} := W^{[\ell]}a^{[\ell-1]} + b^{[\ell]} \xrightarrow{\text{Activation}} a^{[\ell]} := S^{[\ell]}(z^{[\ell]}), \qquad (3.2)$$

$$f_{\theta^{[\ell]}}^{[\ell]}$$

where $a^{[0]} = x$. Hence the parameters of the ℓ-th layer, $\theta^{[\ell]}$, are given by the $N_\ell \times N_{\ell-1}$ *weight matrix* $W^{[\ell]} = \left(w_{i,j}^{[\ell]} \right)$ and the N_ℓ dimensional *bias vector* $b^{[\ell]} = (b_i^{[\ell]})$. Thus the parameter space of the layer is $\Theta^{[\ell]} = \mathbb{R}^{N_\ell \times N_{\ell-1}} \times \mathbb{R}^{N_\ell}$.

- The activation function $S^{[\ell]} : \mathbb{R}^{N_\ell} \longrightarrow \mathbb{R}^{N_\ell}$ is a non-linear multivalued function. For $\ell = 1, \ldots, L-1$ it is generally of the form

$$S^{[\ell]}(z) = \left[\sigma^{[\ell]}(z_1) \ \ldots \ \sigma^{[\ell]}(z_{N_\ell}) \right]^\top, \qquad (3.3)$$

where $\sigma^{[\ell]} : \mathbb{R} \to \mathbb{R}$ is typically an activation function common to all hidden layers.

- For the output layer, $\ell = L$, it is often of a different form depending on the task at hand.

- In the popular case of multi-class classification, a *softmax* function is used, or more specifically for binary classification, a *sigmoid* function is typically used as in the case of logistic regression.

- Thus, in such a classification framework, the output of the network is a vector of probability values determining class membership.

- In order to get a class label prediction one can convert the predicted probability scores into a class label using a chosen *threshold* as discussed in Week 1.

- Next section presents most poular activation functiosn used in deep learning.

The Forward Pass

- The *forward pass* equation, (3.1), of a deep neural network can be expanded out as follows,

$$
\text{Layer 1} \left\{ \begin{array}{rcl} z^{[1]} & = & W^{[1]} \overbrace{x}^{\text{Input}} + b^{[1]} \\ a^{[1]} & = & S^{[1]}(z^{[1]}) \end{array} \right.
$$

$$
\text{Layer 2} \left\{ \begin{array}{rcl} z^{[2]} & = & W^{[2]} a^{[1]} + b^{[2]} \\ a^{[2]} & = & S^{[2]}(z^{[2]}) \end{array} \right. \tag{3.4}
$$

$$
\vdots
$$

$$
\text{Layer L} \left\{ \begin{array}{rcccl} & & z^{[L]} & = & W^{[L]} a^{[L-1]} + b^{[L]} \\ \underbrace{\hat{y}}_{\text{output}} & = & a^{[L]} & = & S^{[L]}(z^{[L]}). \end{array} \right.
$$

- Thus, for a given input x, when computing $f_\theta(x)$, we sequentially execute the affine transformations and activation functions from layer 1 to layer L.

- The computational cost of such a forward pass is at an order of the total number of weights.

A General Approximation Result

- The expressive power of feedforward neural networks is highlighted by the following theorem, known as *universal approximation theorem*, which shows that we can approximate any continuous function on a convex set with the feedforward neural network with a single hidden layer ($L = 2$) and identity activations in the second layer.

Theorem 3.3.1

Consider a continuous function $f^* : \mathcal{K} \to \mathbb{R}^q$ where $\mathcal{K} \subseteq \mathbb{R}^p$ is a compact set. Then for any non-polynomial activation function $\sigma^{[1]}(\cdot)$ and any $\varepsilon > 0$, there exists an N_1 and parameters $W^{[1]} \in \mathbb{R}^{N_1 \times p}$, $b^{[1]} \in \mathbb{R}^{N_1}$, and $W^{[2]} \in \mathbb{R}^{q \times N_1}$, such that the function

$$
f_\theta(x) = W^{[2]} S^{[1]}(W^{[1]} x + b^{[1]}), \qquad \text{with} \qquad S^{[1]} \quad \text{as in (3.3),}
$$

satisfies $||f_\theta(x) - f^*(x)|| < \varepsilon$ for all $x \in \mathcal{K}$.

- Hence, this theorem states that essentially all functions can be approximated to arbitrary precision dictated via ε.

- Practically for complicated functions $f^*(\cdot)$ and small ε one may need large N_1. Yet, the theorem states that it is always possible.

- Note also that the tanh, sigmoid, and ReLU activation functions described in the next section are some of the valid activation functions for this result.

The Strength of a Hidden Layer

- As we saw in Week 1, shallow neural networks such as logistic regression can be used to create classifiers with linear decision boundaries.

- However, the expressiveness of models with a single hidden layer (or more), as introduced in the current chapter, can yield a versatile alternative to such shallow networks.

- Consider Figure 3.5 (a) for a classification task based on two inputs $x \in \mathbb{R}^2$ using logistic regression.

- By adding a single hidden layer with 4 neurons (*sigmoid* activation function is used for all units), we can move beyond the linear boundaries to obtain Figure 3.5 (b).

- Then, by increasing the number of neurons in the hidden layer from 4 to 10 units, the model further refines the decision boundary as in Figure 3.5 (c).
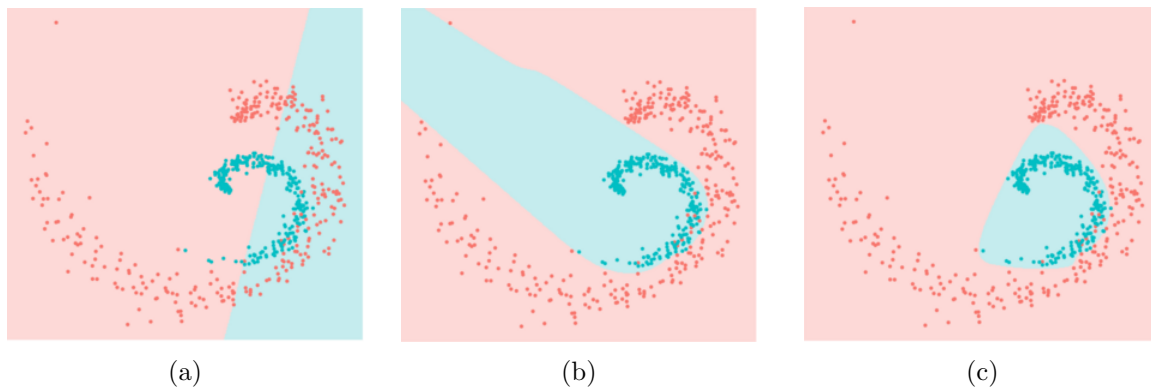

(a)   (b)   (c)

Figure 3.5: Binary classification example with $x \in \mathbb{R}^2$, moving from a shallow neural network to a model with a hidden layer and then increasing the number of neurons. (a) Sigmoid model ($L = 1$). (b) One-hidden layer ($L = 2$) $N_1 = 4$ neurons. (c) One-hidden layer ($L = 2$) $N_1 = 10$ neurons.

3.4 Activation Functions

- As presented in (3.2), each layer ℓ of the network incorporates an activation function which is generally a non-linear transformation of $z^{[\ell]}$ to arrive at $a^{[\ell]}$.

- For most layers of the network, the activation function $S^{[\ell]}(\cdot)$ is composed of a sequence of identical scalar valued activation functions as in (3.3).

- That is, the ℓ-th layer incorporates a scalar activation function $\sigma^{[\ell]} : \mathbb{R} \to \mathbb{R}$ and it is applied to each of the N_ℓ coordinates of $z^{[\ell]}$ separately.

- In some models, all the scalar activation functions across all layers will be of the same form, while in other models different layers will sometimes incorporate different forms of scalar activation functions.

Scalar Activations and their Derivatives

- There are different scalar activation functions used in deep learning. Choice of a function is often not based on theory, but rather based on practice and experience over the years. Here we outline the key activation functions.

- At the onset of the development of deep learning, in the late 1950's, the *step* scalar activation function was used. That is,

$$\sigma_{\text{Step}}(u) = \begin{cases} -1 & u < 0, \\ +1 & u \geq 0. \end{cases} \tag{3.5}$$

- However, σ_{Step} is not used in modern neural network models, primarily because its derivative $\dot{\sigma}_{\text{Step}}(u) = 0$ for all $u \neq 0$.

- Indeed the derivative of activation functions is important since it is used in the backpropagation algorithm (see the next section) to compute the gradient of the loss function with respect to the model parameters.

- The *sigmoid* activation function (also known as the *logistic* function), which we studied in Week 1, is a much more popular choice.

- Note also that its derivative $\dot{\sigma}_{\text{Sig}}$ can be expressed in terms of the function σ_{Sig} itself,

$$\sigma_{\text{Sig}}(u) = \frac{e^u}{1 + e^u} = \frac{1}{1 + e^{-u}}, \qquad \text{with} \qquad \dot{\sigma}_{\text{Sig}}(u) = \sigma_{\text{Sig}}(u)\big(1 - \sigma_{\text{Sig}}(u)\big).$$

- A similar popular function is *hyperbolic tangent*, denoted *tanh*,

$$\sigma_{\text{Tanh}}(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}, \qquad \text{with} \qquad \dot{\sigma}_{\text{Tanh}}(u) = 1 - \sigma_{\text{Tanh}}(u)^2.$$

- Both σ_{Sig} and σ_{Tanh} share similar qualitative properties as σ_{Step}. They are non-decreasing and bounded. At $u \to \infty$ both functions converge to unity just like σ_{Step} and at $u \to -\infty$ the sigmoid function converges to 0 while the tanh function converges to -1 like σ_{Step}.

- In practice when the output $\hat{y}$ is a probability in $[0, 1]$ using σ_{Sig} is much more common.

- See Figure 3.6 for plots of several activation functions.

- In earlier applications of deep learning, it was common to choose between models or layers that use σ_{Sig}, σ_{Tanh}, or similar forms.
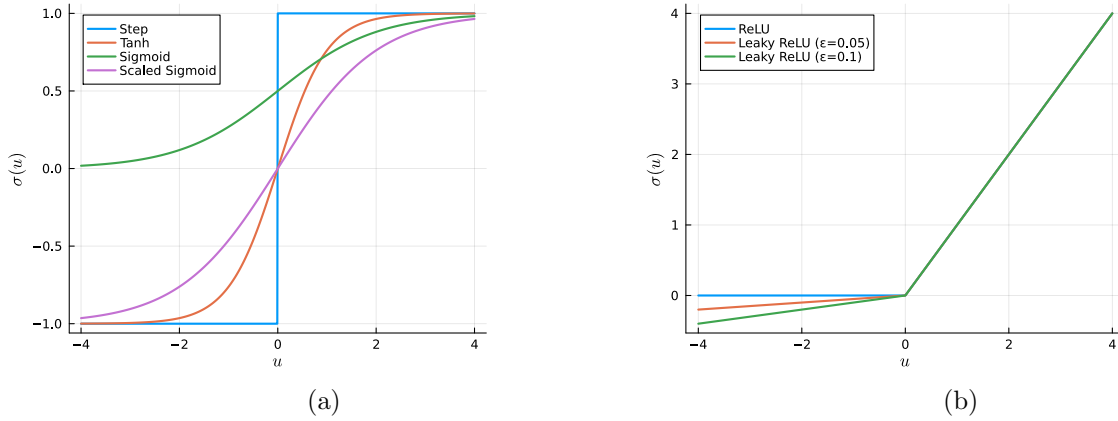
(a)   (b)

Figure 3.6: Several common scalar activation functions. (a) The step and tanh function have a range of $(-1, 1)$ while the sigmoid function has a range of $(0, 1)$. We also plot a scaled sigmoid to the range $(-1, 1)$ so it can be compared to tanh. (b) The ReLU activation function and the leaky ReLU variant with different leaky ReLU parameters.

- However in more recent years, a completely different type of scalar activation function became popular, namely the *rectified linear unit*ReLU [3] or *ReLU*,

$$\sigma_{\text{ReLU}}(u) = \max(0, u) = \begin{cases} 0 & u < 0, \\ u & u \geq 0, \end{cases} \qquad \text{with,} \qquad (3.6)$$

$$\dot{\sigma}_{\text{ReLU}}(u) = \mathbf{1}\{u \geq 0\} = \begin{cases} 0 & u < 0, \\ 1 & u \geq 0. \end{cases}$$

- A related activation function, *leaky ReLU*, parameterized by a fixed small $\varepsilon \geq 0$ (e.g., $\varepsilon = 0.01$) and defined via,

$$\sigma_{\text{LeakyReLU}}(u) = \max(0, u) + \min(0, \varepsilon u) = \begin{cases} \varepsilon u & u < 0, \\ u & u \geq 0, \end{cases} \qquad \text{with,}$$

$$\dot{\sigma}_{\text{LeakyReLU}}(u) = \mathbf{1}\{u \geq 0\} + \varepsilon \mathbf{1}\{u < 0\} = \begin{cases} \varepsilon & u < 0, \\ 1 & u \geq 0. \end{cases}$$

Observe that when $\varepsilon = 0$, this is just the ReLU activation function.

- Another variant called *PReLU* (parametric ReLU) considers the leaky ReLU parameter ε as a learned parameter. That is, the gradient based optimization for the parameters of the network also includes improvement steps of ε, incorporating it as part of the parameters for ℓ-th layer, $\theta^{[\ell]}$.

---

[3]Note that source of the name is from electrical engineering where a rectifier is a device that converts alternating current to direct current.

Non-Scalar Activations and their Derivatives

- Some layers also use non-scalar activation functions. That is, $S^{[\ell]} : \mathbb{R}^{N_\ell} \to \mathbb{R}^{N_\ell}$ is a vector to vector function that cannot be decomposed as in (3.3).

- The most common example of this is the softmax activation function, typically used for classification (with K classes) in the last layer $\ell = L$. In that case, $N_L = K$ and

$$a^{[L]} = S_{\text{Softmax}}(z^{[L]}), \tag{3.7}$$

which is defined as

$$S_{\text{Softmax}}(z) = \frac{1}{\sum_{i=1}^{K} e^{z_i}} \begin{bmatrix} e^{z_1} & \cdots & e^{z_K} \end{bmatrix}^\top,$$

3.5 Backpropagation Algorithm

- Now we focus on computation of the gradient of the loss function with respect to the parameters so as to facilitate learning using variants of gradient descent method discussed in Week 1.

- A key algorithm is the *backpropagation algorithm* which implements backward mode *automatic differentiation.*

- We state this algorithm for general recursive model.

- It is instructive to first consider a general recursive feedforward model as appearing in (3.1). For such a model, the recursive step is of the form $a^{[\ell]} = f_{\theta^{[\ell]}}^{[\ell]}(a^{[\ell-1]})$.

- However, for our discussion, it is convenient to use notation that treats the function $f^{[\ell]}$ separately as a function of $a^{[\ell-1]}$ and of the parameter $\theta^{[\ell]}$:

$$f^{[\ell]}(\cdot \, ; \theta^{[\ell]}) : \mathbb{R}^{N_{\ell-1}} \longrightarrow \mathbb{R}^{N_\ell}, \qquad \text{and} \qquad f^{[\ell]}(a^{[\ell-1]} \, ; \cdot) : \Theta^{[\ell]} \longrightarrow \mathbb{R}^{N_\ell}.$$

- Using this notation, the recursive step is,

$$a^{[\ell]} = f^{[\ell]}(a^{[\ell-1]} \, ; \theta^{[\ell]}), \qquad \text{for} \qquad \ell = 1, \ldots, L, \tag{3.8}$$

where $a^{[0]} = x$ and $\hat{y} = a^{[L]}$.

- Given a single data sample (x, y) we assume there is a loss function which depends on the given parameters θ, on the label value y, and on the output of the model, $a^{[L]}$. We denote this loss via $C(a^{[L]}, y \, ; \theta)$.

- Our goal is to optimize the loss with respect to $\theta = (\theta^{[1]}, \ldots, \theta^{[L]})$.

- For this, we require the gradient with respect to θ and we denote its components via,

$$g_\theta^{[\ell]} := \frac{\partial C(a^{[L]}, y \, ; \theta)}{\partial \theta^{[\ell]}}. \tag{3.9}$$

- Further, we need

$$\dot{C}(u) := \frac{\partial C(u, y\,;\,\theta)}{\partial u}, \quad \dot{f}_a^{[\ell]}(u) := \frac{\partial f^{[\ell]}(u\,;\,\theta^{[\ell]})}{\partial u}, \quad \dot{f}_\theta^{[\ell]}(u) := \frac{\partial f^{[\ell]}(a^{[\ell-1]}\,;\,u)}{\partial u}. \tag{3.10}$$

- The derivative $\dot{C}(u)$ is typically a gradient and thus vector valued with length N_L.

- The derivative $\dot{f}_a^{[\ell]}(u)$ is typically an $N_{\ell-1} \times N_\ell$ matrix obtained via a transpose of a Jacobian and thus matrix valued.

- Finally, the derivative $\dot{f}_\theta^{[\ell]}(u)$ may take on various shapes depending on the form of θ.
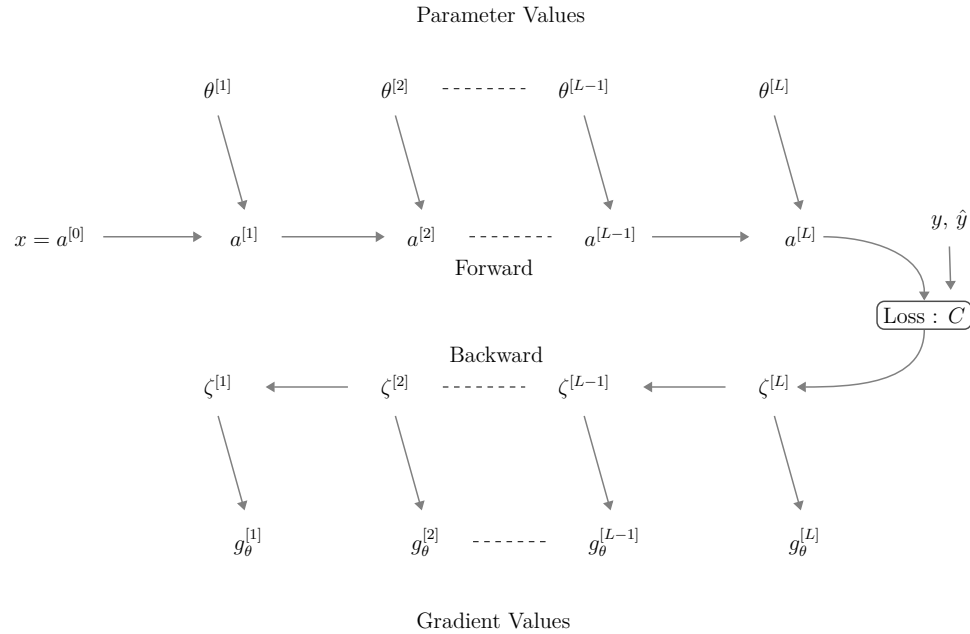

Figure 3.7: The variables and flow of information in the backpropagation algorithm for the general recursive model.

- En-route to compute the desired gradients (3.9), we require intermediate gradients of the loss with respect to the activation values $a^{[1]}, \ldots, a^{[L]}$.

- Keeping in mind that $a^{[L]} = \hat{y}$ is a function of these activation values, the intermediate gradients are denoted,

$$\zeta^{[\ell]} := \frac{\partial C(a^{[L]}, y\,;\,\theta)}{\partial a^{[\ell]}}, \qquad \ell = 1, \ldots, L. \tag{3.11}$$

- Now based on the multivariate chain rule, the recursive step (3.8), and the definitions above, we observe,

$$\zeta^{[\ell]} = \frac{\partial a^{[\ell+1]}}{\partial a^{[\ell]}} \frac{\partial C}{\partial a^{[\ell+1]}} = \dot{f}_a^{[\ell+1]}(a^{[\ell]})\,\zeta^{[\ell+1]}, \qquad g_\theta^{[\ell]} = \frac{\partial a^{[\ell]}}{\partial \theta^{[\ell]}} \frac{\partial C}{\partial a^{[\ell]}} = \dot{f}_\theta^{[\ell]}(\theta^{[\ell]})\,\zeta^{[\ell]}. \qquad (3.12)$$

- Hence once the activation values $a^{[1]}, \ldots, a^{[L]}$ are populated via forward propagation of (3.8), backward computation can be carried out via,

$$\zeta^{[\ell]} = \begin{cases} \dot{C}(a^{[L]}), & \ell = L, \\ \dot{f}_a^{[\ell+1]}(a^{[\ell]})\,\zeta^{[\ell+1]}, & \ell = L-1, \ldots, 1, \end{cases} \qquad (3.13)$$

  and at each step the gradient can be obtained via $g_\theta^{[\ell]} = \dot{f}_\theta^{[\ell]}(\theta^{[\ell]})\,\zeta^{[\ell]}$.

- This process backpropagation is illustrated in Figure 3.7, and summarized in Algorithm 1.

Algorithm 1: Backpropagation for the general recursive model

Input: Dataset $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)})\}$,
      loss function $C(\cdot)$, and
      parameter values $\theta = (\theta^{[1]}, \ldots, \theta^{[L]})$
Output: gradients of the loss $g_\theta^{[1]}, \ldots, g_\theta^{[L]}$

Compute $a^{[\ell]}$ for $\ell = 1, \ldots, L$ using (3.8) (Forward pass)
Compute $\zeta^{[L]} = \dot{C}(a^{[L]})$
Compute $g_\theta^{[L]} = \dot{f}_\theta^{[L]}(\theta^{[L]})\,\zeta^{[L]}$

for $\ell = L-1, \ldots, 1$ **do**
    compute $\zeta^{[\ell]} = \dot{f}_a^{[\ell+1]}(a^{[\ell]})\,\zeta^{[\ell+1]}$
    compute $g_\theta^{[\ell]} = \dot{f}_\theta^{[\ell]}(\theta^{[\ell]})\,\zeta^{[\ell]}$
end

3.6 Number of Hidden Layers

- Despite the fact that the universal approximation theorem states that almost any function can be approximated using a neural network model with a single hidden layer, practice and research has shown that to gain high expressive power, this model might require a very large number of units (N_1 needs to be very large).

- Hence gaining significant expressive power may require a very large number of parameters. The power of deep learning then arises via repeated composition of non-linear activations functions via an increase of depth (an increase of L).

- Note first that if the identity activation function is used in each hidden layer, then the network reduces to a shallow neural network,

$$f_\theta(x) = S^{[L]}(\tilde{W}x + \tilde{b}),$$

where,[4]

$$\tilde{W} = W^{[L]}W^{[L-1]} \cdot \ldots \cdot W^{[1]}, \qquad \text{and} \qquad \tilde{b} = \sum_{\ell=1}^{L} \Big(\prod_{\tilde{\ell}=\ell+1}^{L} W^{[\tilde{\ell}]} \Big) b^{[\ell]}.$$

- In the case where the identity function is also used for the output layer, the model reduces to be a linear (affine) model. Thus, we have no gain by going deeper and adding multiple layers with identity activations.

- Thus, non-linearity obtained by activations in each layer is crucial in building deep neural networks.

- The expressivity of the neural network comes from the composition of non-linear activation functions. The repeated compositions of such functions has significant expressive power and can reduce the number of units needed in each layer in comparison to a network with a single hidden layer. A consequence is that the parameter space is reduced as well.

3.7 One Hot Encoding

In deep learning, especially in tasks like natural language processing (NLP) and classification, one-hot encoding is a common technique used to represent categorical variables numerically. It involves converting categorical variables into binary vectors, where each vector has a length equal to the number of categories and contains a single '1' value at the index corresponding to the category, while all other elements are '0'.

- Denote a categorical variable with K categories as $c_1, c_2, \ldots, c_K$.

- The one-hot encoding of this variable would result in binary vectors $x_1, x_2, \ldots, x_K$, each with dimension K, where:

$$x_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

where j is the index of the category c_j in the original categorical variable.

- This means that the one-hot encoding of the category c_i results in a vector where the ith element is '1' and all other elements are '0'.

- **Example:** Consider a categorical variable `Animal` with $K = 3$ categories:

---

[4]Note that in the expression $\prod_{\tilde{\ell}=\ell+1}^{L} W^{[\tilde{\ell}]}$ we assume that the product multiplies the matrices in the left to right order $W^{[L]}W^{[L-1]}W^{[L-2]} \ldots$. Further, for $\ell = L$ the product is taken as the identity matrix.

- Categories: $\{\text{Dog}, \text{Cat}, \text{Bird}\}$
- Index mapping: $c_1 = \text{Dog}$, $c_2 = \text{Cat}$, $c_3 = \text{Bird}$

The one-hot encoding produces:

Original	Encoded Vector		
	x_1	x_2	x_3
Dog	1	0	0
Cat	0	1	0
Bird	0	0	1

- In deep learning applications, one-hot encoding is commonly used as input representations for categorical variables in neural networks, particularly in tasks such as multi-class classification, where the output layer typically employs a softmax activation function. This encoding ensures that the neural network can effectively learn to predict probabilities across multiple classes.

- Despite its simplicity, one-hot encoding has some drawbacks, such as high dimensionality when dealing with a large number of categories and the inability to capture ordinal relationships between categories. However, it remains a fundamental technique in deep learning for handling categorical variables efficiently.