What is Deep Learning?

Deep Learning

The subfield of machine learning that deals exclusively with a single type of predictive model: **(Deep) Artificial Neural Networks (NNs)**. Due to the vast number of different versatile architectures (specific NN designs) that have been developed (and are still being invented), NNs are extremely powerful predictive models that excel at (essentially) all kinds of tasks. At its core, deep learning refers to a series of machine learning models that transform input data in a way that help identify composite patterns and complex relationships through a series of function compositions (referred to as "layers" in the context of NNs). The word "deep" in the name "deep learning" refers to the number of layers that are present in the network architecture.

Types of Neural Architectures & Their Applications

- **Multi-Layered Perceptrons:** The most classical of NNs. Typically used on tabular datasets that consist of feature-label pairs, much like other supervised learning models.
- **Convolutional Networks:** A spatial-oriented architecture that utilizes convolutions and kernels to obtain "feature maps" of matrix-type data (like images or other data that has some type of spatial relationship).
- **Recurrent Networks:** A sequence-oriented architecture that aims at identifying predictions in a sequence chain by adding more weight to the "most recent" data in the sequence (such as text data or time-series data).
- **Transformer Networks:** A series of powerful sequence-oriented architectures that consist of either embedding data into abstract representations (Encoder), generating new data from an existing embedding (Decoder), or obtain embeddings from some data followed by generating new different type of data (Encoder-Decoder).

Some Reflections on Biology

Intuitively, artificial neural networks were inspired, and receive their name, from neuron cells found in the brain.

A **Biological Neuron** consists of 4 main components:
- **Dendrites:** The input channels through which the neuron receives incoming electrical impulses.
- **Nucleus:** The core of the neuron.
- **Axon:** The terminal through which the neuron sends information in the form of electrical pulses to other neurons.
- **Terminals:** The "gateways" to other neurons and contain synapses which permits an electrical pulse to be received by another neuron.

Dendrites

Nucleus

Axon Terminals

Axon
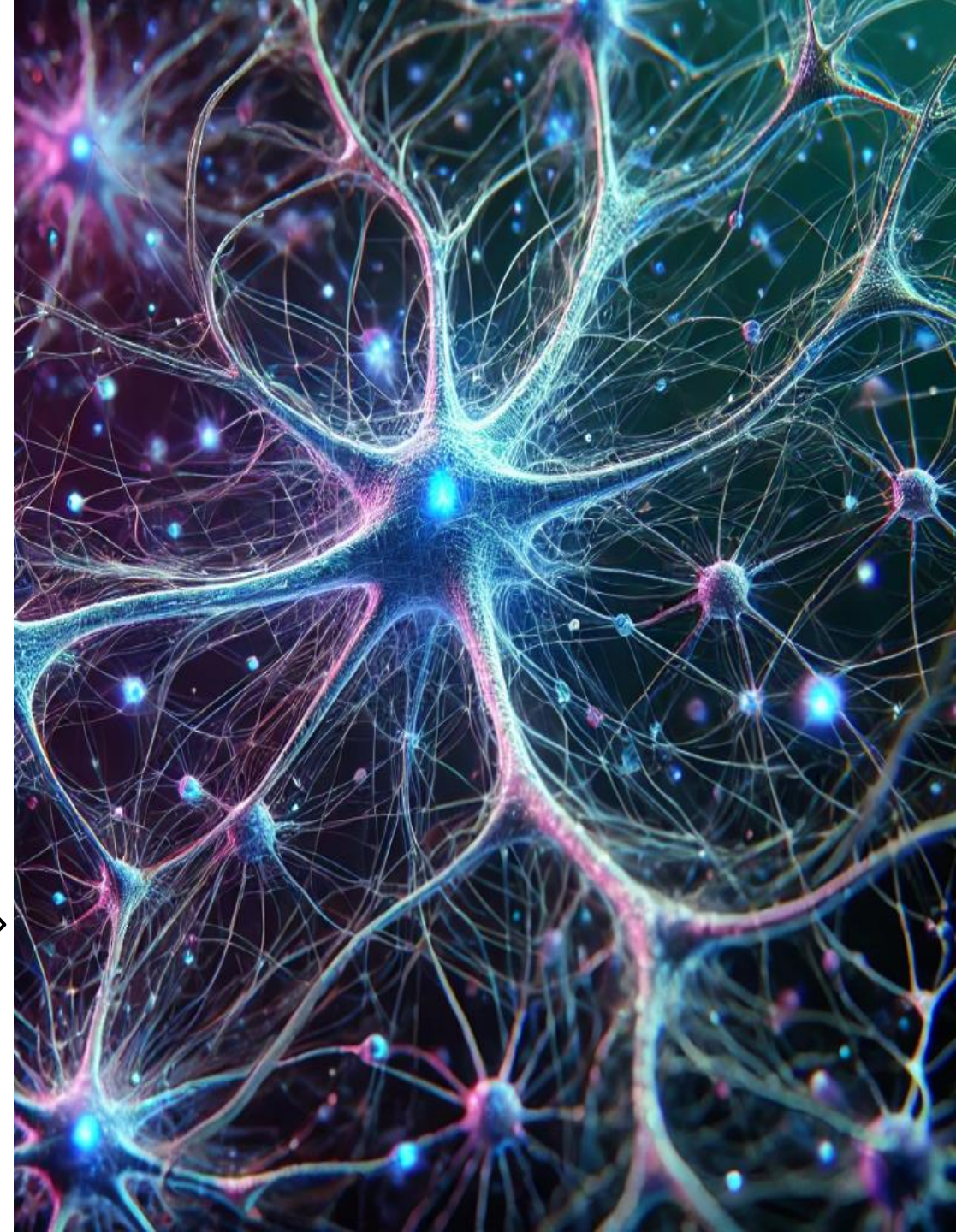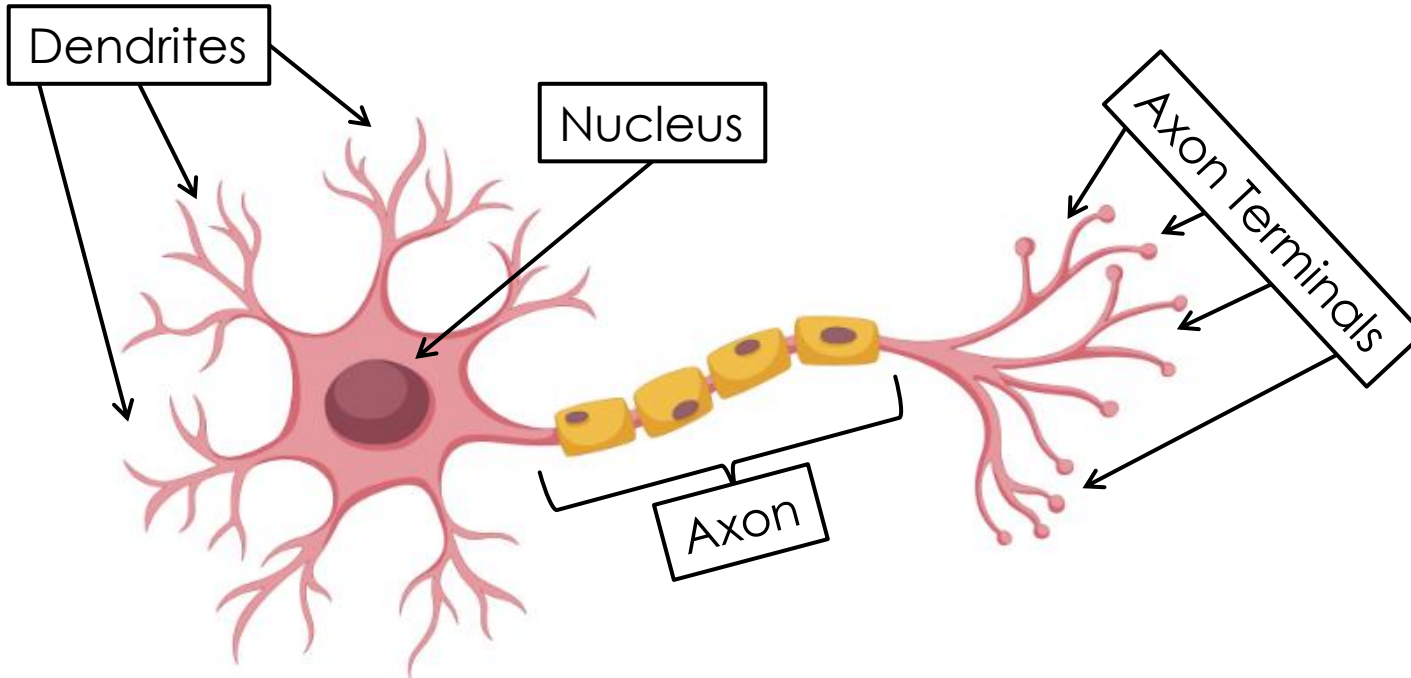
The Perceptron

The **origins** of Artificial Neural Networks can be traced back to **1958** when Frank Rosenblatt published a model that he termed as the **Perceptron** in a famous paper titled "*The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*"; a simple predictive model that ultimately serves as the fundamental building block of modern day NNs.

The Perceptron

The most foundational building block of a neural network is the **Perceptron**. This is a function that takes as input some vector $x \in \mathbb{R}^n$, performs a linear transformation $w^T x + b$ (for some **weight vector** $w \in \mathbb{R}^n$ and intercept (or **bias term**) $b \in \mathbb{R}$), and returns the output of some **activation function** $g: \mathbb{R} \to \mathbb{R}$. The **perceptron model** can be written formally as

$$h_\theta(x) = g(w^T x + b),$$

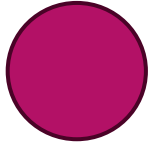where $\theta := [w^T, b]^T$.

- Further, the output of a perceptron is also referred to as a "**neuron**" or a "**node**". When we introduce the concept of a multi-layered perceptron, this can be represented as a network graph that consists of a series of connections between layers and layers of nodes.
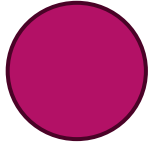
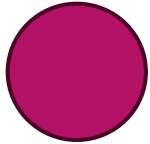## A Connection to <u>Linear</u> & <u>Logistic</u> Regression

- When using a single perceptron, it will typically take on the form of a **linear** or **logistic regression model**.
- For regression, typically $g = I$ (identity function), meaning that $h_\theta(x) = w^T x + b$ (a **linear model**).
- For classification, typically $g = \sigma$ (sigmoid function), meaning that $h_\theta(x) = \sigma(w^T x + b)$ (a **logistic model**).
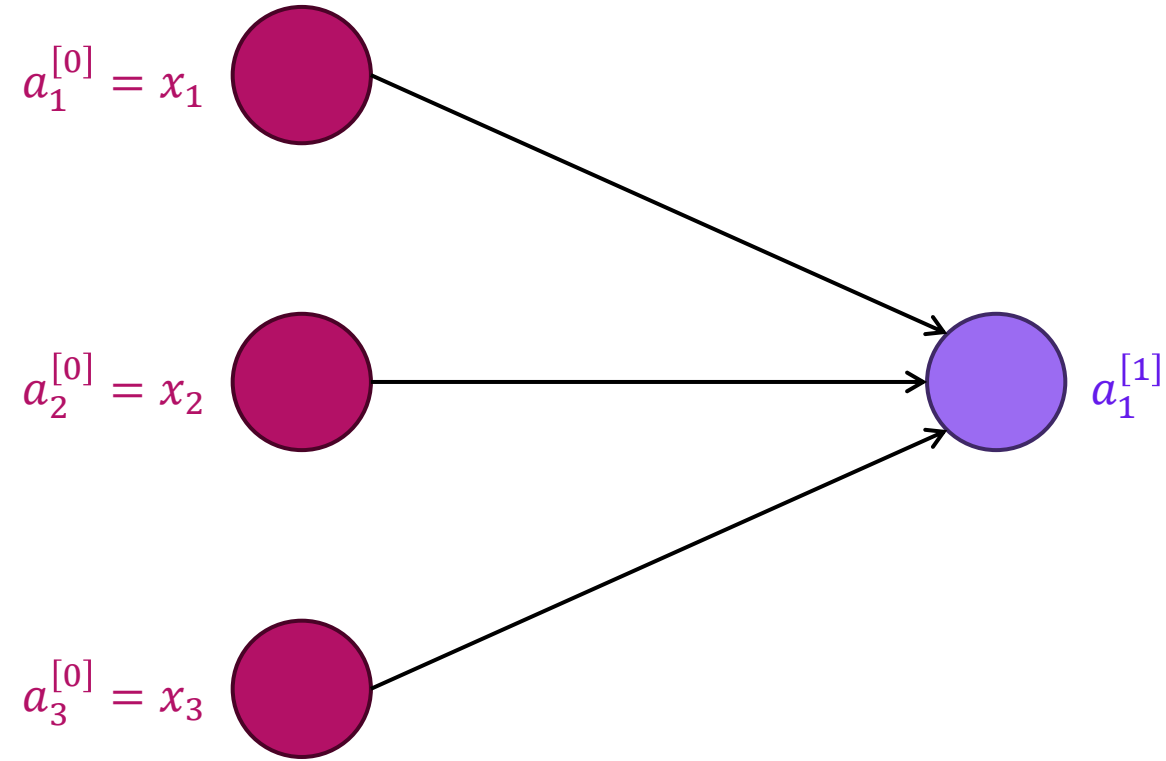
The Perceptron (Illustration)

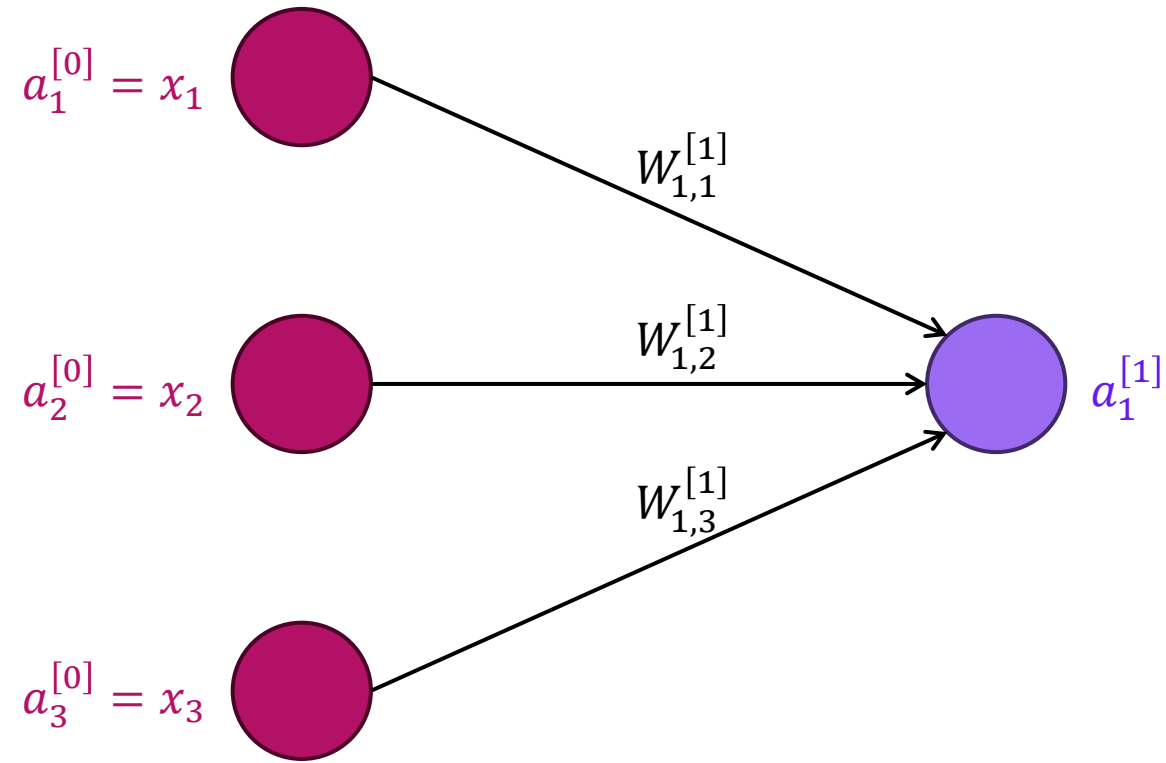$$a_1^{[0]} = x_1$$

$$a_2^{[0]} = x_2$$

$$a_3^{[0]} = x_3$$

The Perceptron (Illustration)

The Perceptron (Illustration)

The Perceptron (Illustration)

$$W^{[1]} = \begin{bmatrix} W_{1,1}^{[1]} & W_{1,2}^{[1]} & W_{1,3}^{[1]} \end{bmatrix} \in \mathbb{R}^{1 \times 3}$$
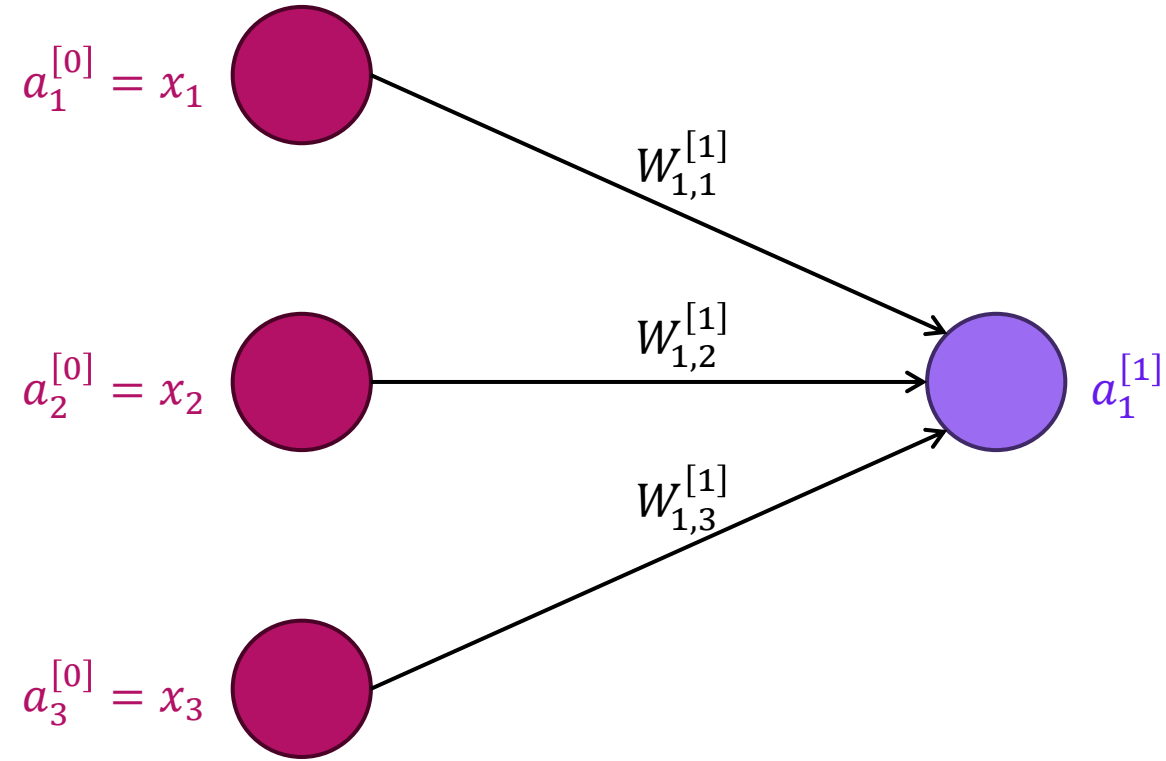
The Perceptron (Illustration)

$$W^{[1]} = \begin{bmatrix} W_{1,1}^{[1]} & W_{1,2}^{[1]} & W_{1,3}^{[1]} \end{bmatrix} \in \mathbb{R}^{1 \times 3} \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \end{bmatrix} \in \mathbb{R}^{1 \times 1}$$
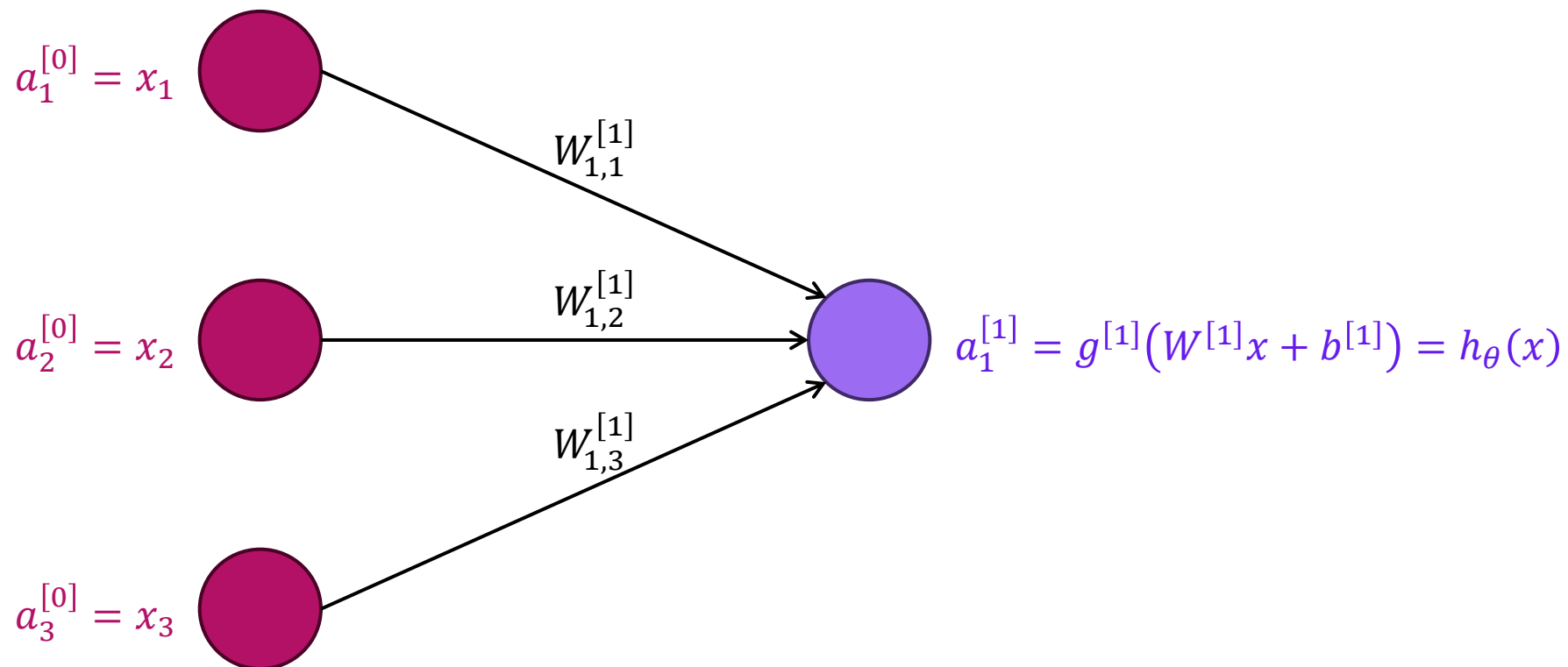

$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$

$a_3^{[0]} = x_3$

$W_{1,1}^{[1]}$

$W_{1,2}^{[1]}$

$W_{1,3}^{[1]}$

$a_1^{[1]} = g^{[1]}\big(W^{[1]}x + b^{[1]}\big) = h_\theta(x)$

Visualizing the Decision Boundary

$$a_1^{[0]} = x_1$$
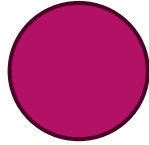
Input Data

$$a_2^{[0]} = x_2$$

Visualizing the Decision Boundary

$a_1^{[0]} = x_1$

Input Data

$W_{1,1}^{[1]}$

$a_1^{[1]}$

$W_{1,2}^{[1]}$

$a_2^{[0]} = x_2$

Visualizing the Decision Boundary

$a_1^{[0]} = x_1$

Input Data

$W_{1,1}^{[1]}$

$W_{1,2}^{[1]}$

$a_2^{[0]} = x_2$

$a_1^{[1]}$
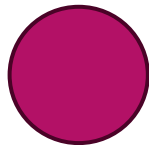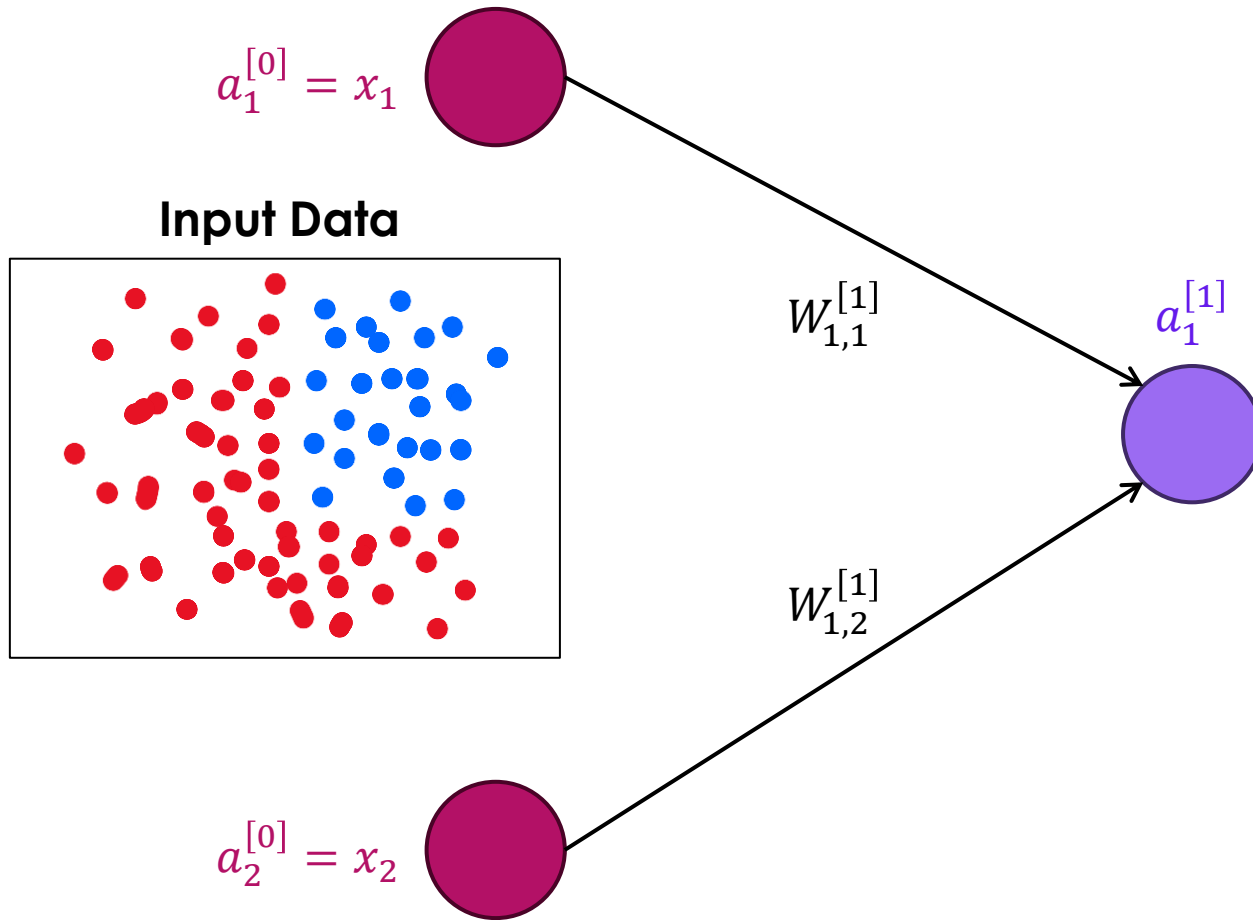
Linear Decision Boundary
$$a_1^{[1]} = g^{[1]}(W^{[1]}x + b^{[1]})$$
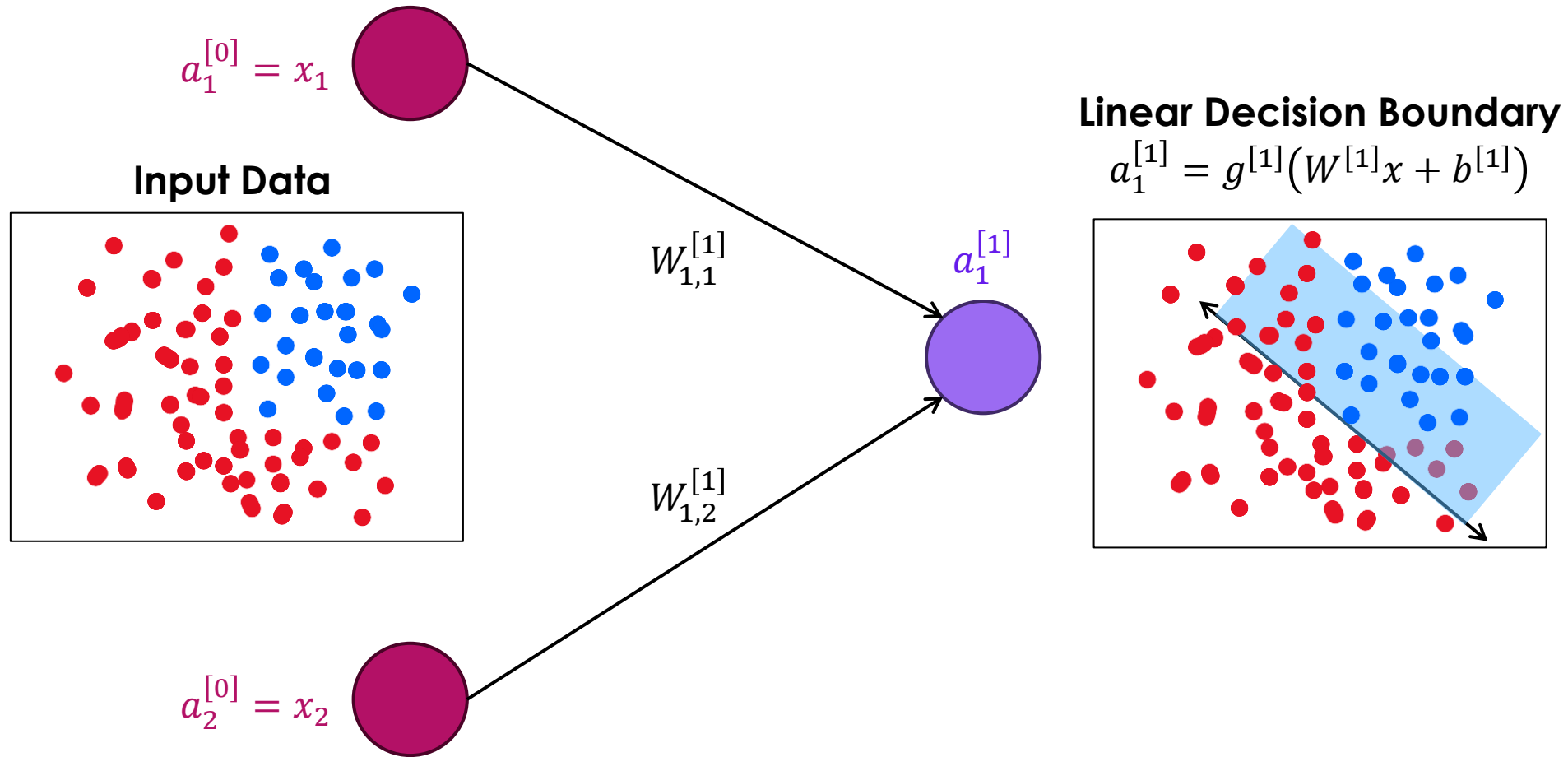
The Historic Problem That Killed Early "A.I." Hype

In 1969, Marvin Minsky and Seymour Papert highlighted an example of a very simple problem that illustrated the limitations of the basic perceptron due to its linear nature.
- This problem is known as the **XOR (Exclusive OR) problem** and consists of four datapoints with two target classes that lie diagonal of each other.
- Clearly, this problem (with a very simple pattern) can not be correctly solved with the perceptron.
- This problem drove research forward to develop the more-sophisticated MLP (multi-layer perceptron) that could indeed identify non-linear patterns
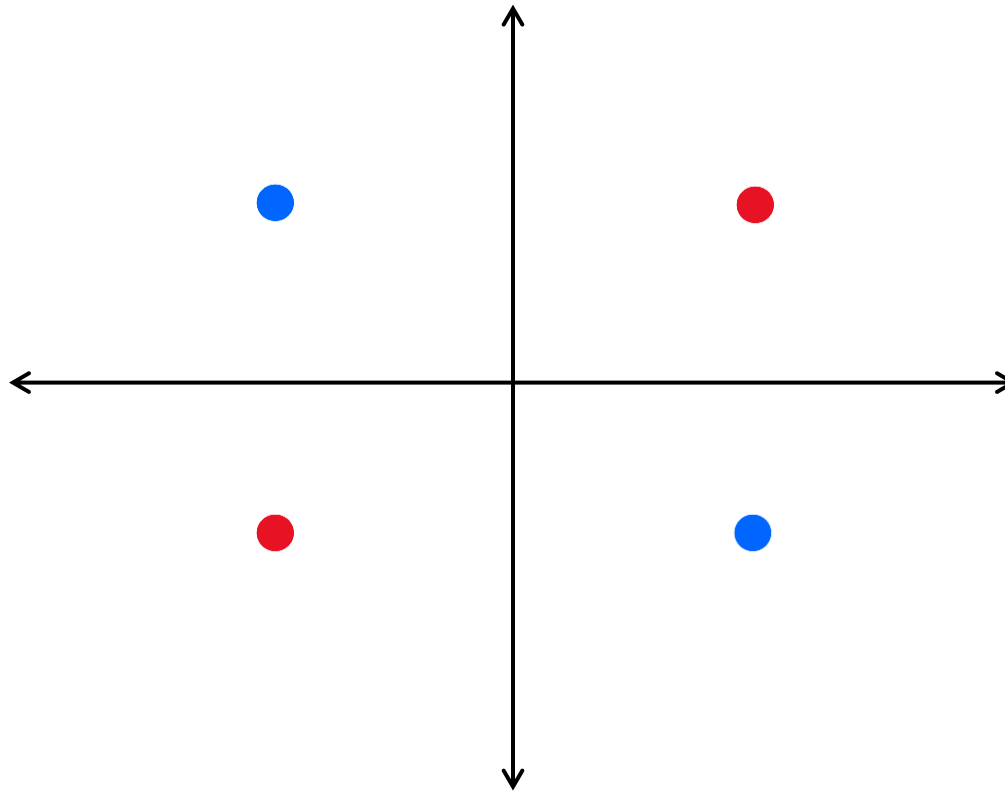
The Multi-Layered Perceptron

Using the perceptron model as a building block, we can define the **classical neural network** model (also known as a **fully-connected feed-forward NN** or a **multi-layered perceptron**) as a function composition defined by many layers of interconnected perceptrons.

Multi-Layered Perceptron (MLP)

The MLP (of feed-forward network) is the simplest form of a neural network and essentially consists of taking an input feature vector $x \in \mathbb{R}^n$ (referred to as the "**input-layer**") and feeding it through a series of intermediate perceptron models (each of which have their own activation functions), with outputs that are in-turn fed through further intermediate perceptrons (these intermediate steps make up what are referred to as the "**hidden layers**" of the network). Ultimately, this will result in a model that can be written as a **series of function compositions**, and which will result in a final "**output layer**" with values that will correspond to the network's prediction. Overall, one can write the general form of an MLP (sing a general and appropriate mathematical notation; more on this later) as a function $\mathcal{N}: \mathbb{R}^n \to \mathbb{R}$, defined as

$$\mathcal{N}(x; g, \theta \coloneqq \{W, b\}) \coloneqq g^{[L]}\left(W^{[L]}g^{[L-1]}\left(...\left(W^{[2]}g^{[1]}\left(W^{[1]}x + b^{[1]}\right) + b^{[2]}\right)...\right) + b^{[L]}\right).$$

In this equation, the parameters W and b are sets of matrices (or vectors) that define the weights and biases associated with each layer. Further, g denotes a series of activation functions for each of the layers and are the hyperparameters of the network. Lastly, we will sometimes use $h_\theta(x) \coloneqq \mathcal{N}(x; g, \theta)$.

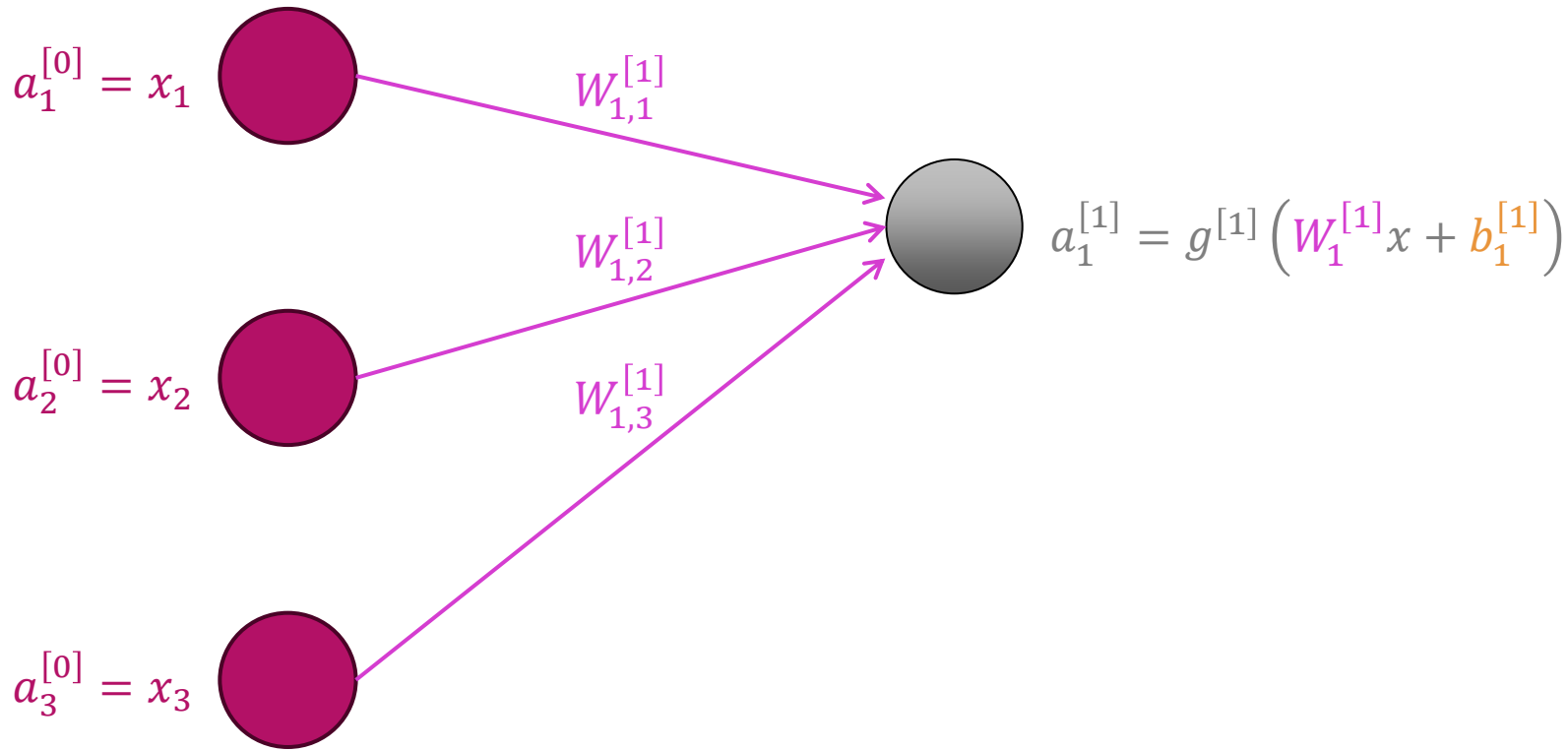## The Multi-Layered Perceptron (A Classical Neural Network)
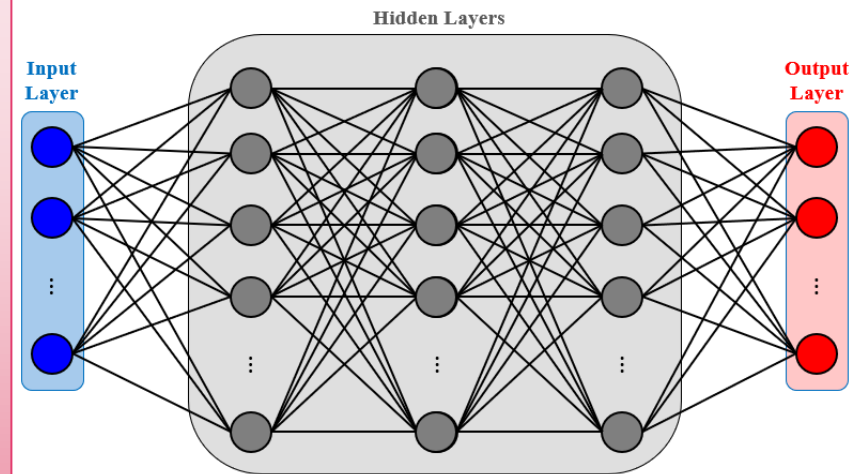
Mathematical Notation

- $L \in \mathbb{N}$: The **number of layers in the network** (including the output layer, but not including the input layer). Thus, if a network has a total of L layers, the output layer will be $\ell = L$, whereas the input layer would be layer $\ell = 0$.

- $n[\ell] \in \mathbb{N}$: The **number of neurons in the ℓ-th layer** of the network.

- $g^{[\ell]} : \mathbb{R} \to \mathbb{R}$: The **activation function in the ℓ-th layer** of the network. The purpose of indexing g by the layer ℓ is due to the fact that different layers of the network can have different activation functions. Notice that we can define the collection of all the activation functions with the simple notation $g := \{g^{[\ell]}\}_{\ell=1}^{L}$.

- $z^{[\ell]} \in \mathbb{R}^{n[\ell]}$: The vector of entries that will serve as **inputs into the activation function** $g^{[\ell]}$ to obtain the output vector $a^{[\ell]}$.

- $a^{[\ell]} \in \mathbb{R}^{n[\ell]}$: The **vector of outputs from the ℓ-th layer**. Hence, each $a_j^{[\ell]}$, for all $j \in \{1, 2, \ldots, n[\ell]\}$, denotes the output of the j-th neuron in the ℓ-th layer (each $a_j^{[\ell]}$ can be thought of as a node in the network). Further, for the input layer, we denote the input vector of features $x \in \mathbb{R}^n$ by the vector $a^{[0]}$ with elements that correspond to each of the entries in x. It bears mentioning that the brackets in the superscript help to differentiate the layer of the network $[\ell]$ from the index (i) of the datapoint $(x^{(i)}, y^{(i)})$; hence, if we wish to distinguish between different datapoints, we can use $a^{[\ell](i)}$.

- $W_j^{[\ell]} \in \mathbb{R}^{n[\ell-1]}$: The **vector of input weights for the j-th neuron of the ℓ-th layer**. One can stack all $j \in \{1, 2, \ldots, n[\ell]\}$ of these vectors to obtain the weight matrix $W^{[\ell]} \in \mathbb{R}^{n[\ell] \times n[\ell-1]}$ for the ℓ-th layer (each weight vector $W_j^{[\ell]}$ can be thought of as corresponding to the connection between the j-th node in the ℓ-th layer and all nodes in the previous layer). The reason the weight matrix is of dimension $n[\ell] \times n[\ell-1]$ is because there are $n[\ell-1]$ nodes in the previous layer that connect to each of the $n[\ell]$ nodes in the current layer ℓ. Notice that we can define the collection of all the weight matrices with the simple notation $W := \{W^{[\ell]}\}_{\ell=1}^{L}$.

- $b_j^{[\ell]} \in \mathbb{R}$: The **bias (or intercept) term of the j-th neuron in the ℓ-th layer**. Further, these values can be concatenated into the vector of bias terms $b^{[\ell]} \in \mathbb{R}^{n[\ell]}$ of the ℓ-th layer. Notice that we can define the collection of all the bias vectors with the simple notation $b := \{b^{[\ell]}\}_{\ell=1}^{L}$.

A Multi-Layered Perceptron (Illustration)

$$b^{[1]} = \left[b_1^{[1]} \right] \in \mathbb{R}^{1 \times 1} \qquad W^{[1]} = \left[W_{1,1}^{[1]} \quad W_{1,2}^{[1]} \quad W_{1,3}^{[1]} \right] \in \mathbb{R}^{1 \times 3}$$

$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$

$a_3^{[0]} = x_3$

$W_{1,1}^{[1]}$

$W_{1,2}^{[1]}$

$W_{1,3}^{[1]}$

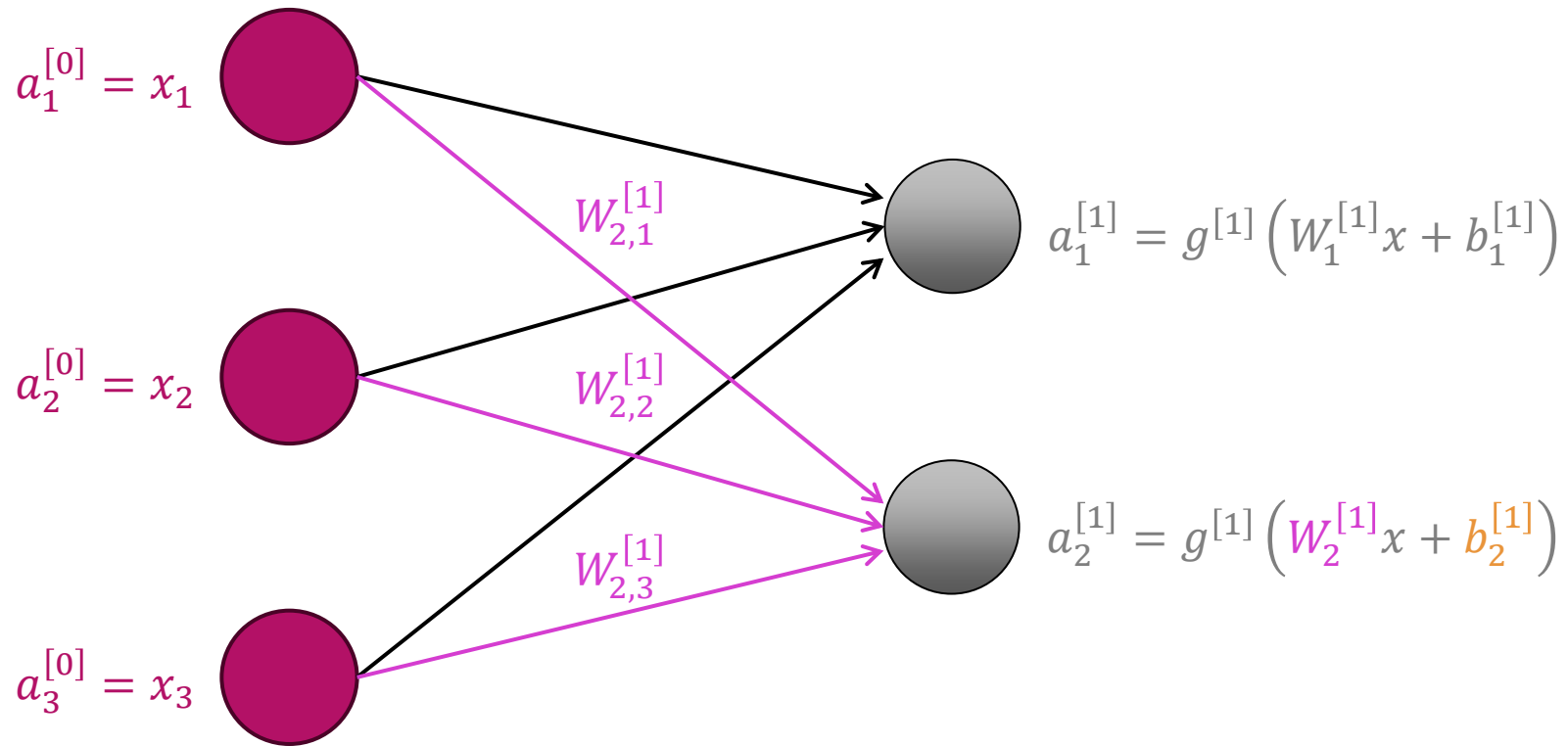$$a_1^{[1]} = g^{[1]} \left(W_1^{[1]} x + b_1^{[1]} \right)$$
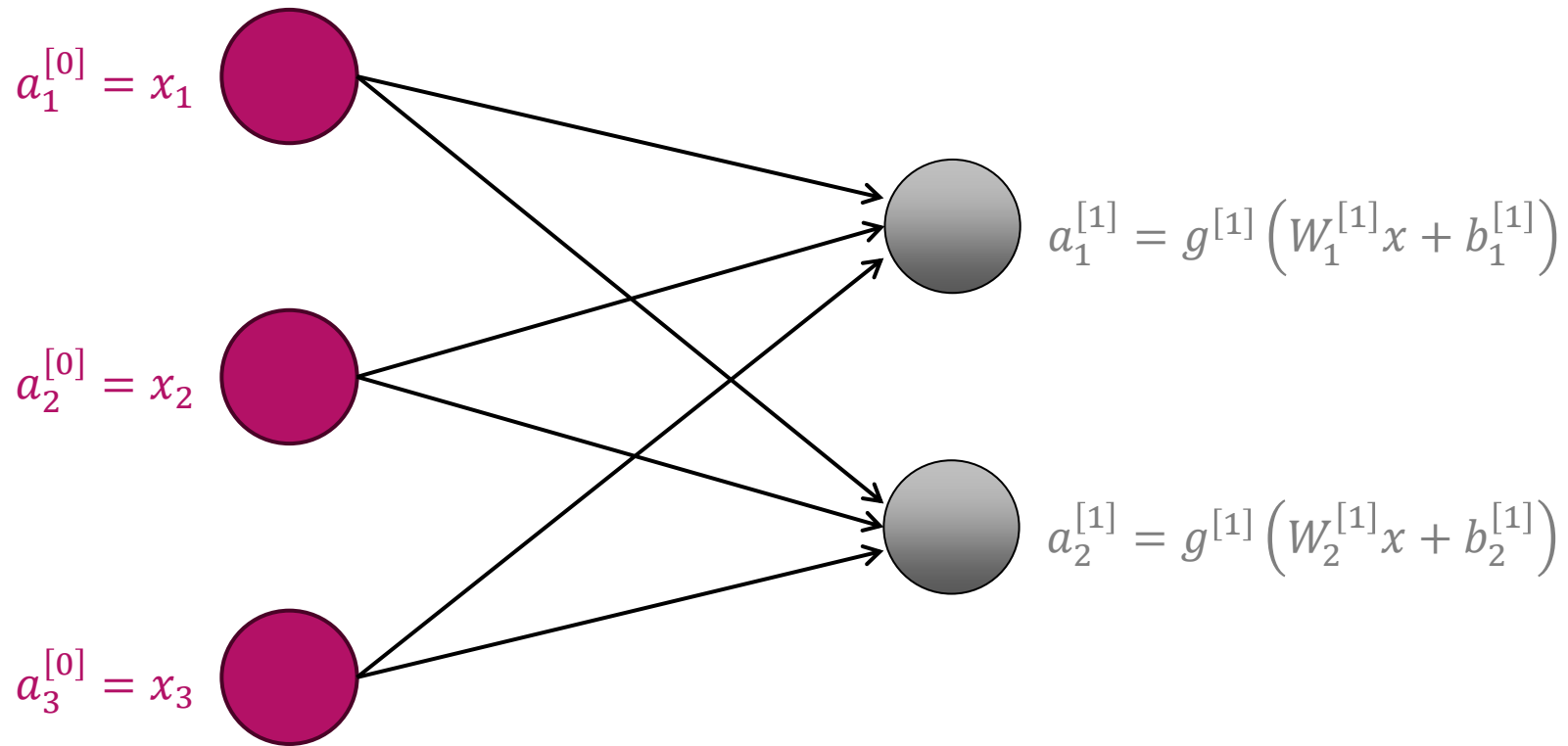
A Multi-Layered Perceptron (Illustration)

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix} \in \mathbb{R}^{2 \times 1} \qquad W^{[1]} = \begin{bmatrix} W_{1,1}^{[1]} & W_{1,2}^{[1]} & W_{1,3}^{[1]} \\ W_{2,1}^{[1]} & W_{2,2}^{[1]} & W_{2,3}^{[1]} \end{bmatrix} \in \mathbb{R}^{2 \times 3}$$

$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$

$a_3^{[0]} = x_3$

$W_{2,1}^{[1]}$

$W_{2,2}^{[1]}$

$W_{2,3}^{[1]}$

$a_1^{[1]} = g^{[1]} \left(W_1^{[1]} x + b_1^{[1]} \right)$

$a_2^{[1]} = g^{[1]} \left(W_2^{[1]} x + b_2^{[1]} \right)$
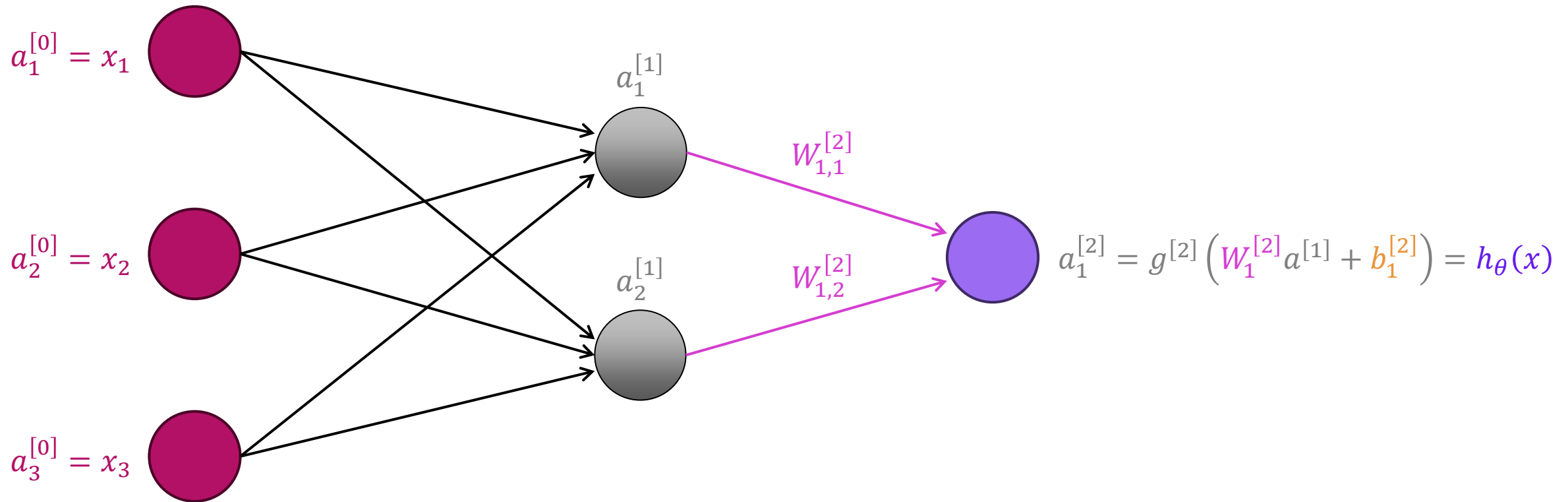
A Multi-Layered Perceptron (Illustration)

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix} \in \mathbb{R}^{2\times 1} \qquad W^{[1]} = \begin{bmatrix} W_{1,1}^{[1]} & W_{1,2}^{[1]} & W_{1,3}^{[1]} \\ W_{2,1}^{[1]} & W_{2,2}^{[1]} & W_{2,3}^{[1]} \end{bmatrix} \in \mathbb{R}^{2\times 3}$$
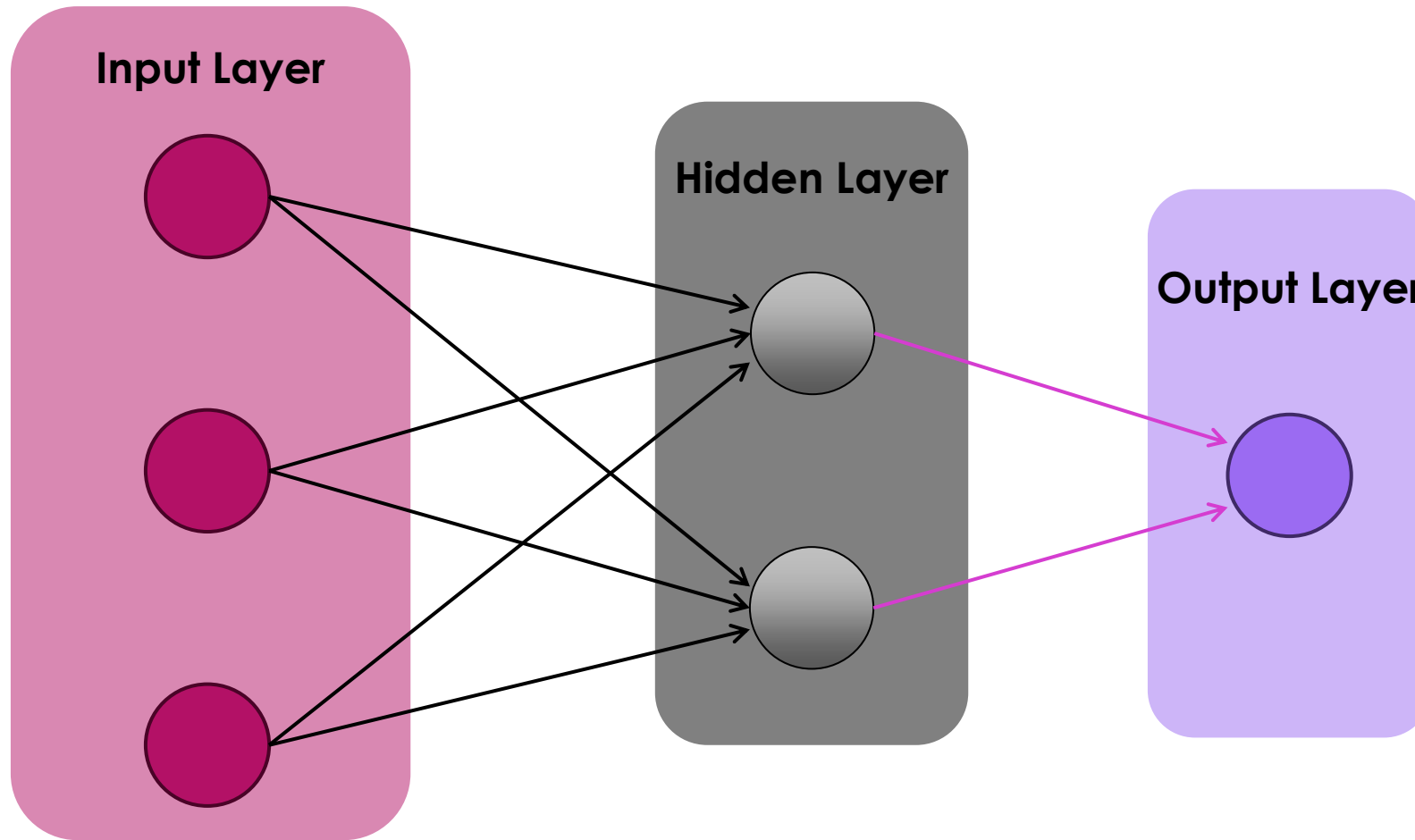

$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$

$a_3^{[0]} = x_3$

$a_1^{[1]} = g^{[1]}\left(W_1^{[1]}x + b_1^{[1]}\right)$

$a_2^{[1]} = g^{[1]}\left(W_2^{[1]}x + b_2^{[1]}\right)$

A Multi-Layered Perceptron (Illustration)

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix} \in \mathbb{R}^{2 \times 1} \qquad W^{[1]} = \begin{bmatrix} W_{1,1}^{[1]} & W_{1,2}^{[1]} & W_{1,3}^{[1]} \\ W_{2,1}^{[1]} & W_{2,2}^{[1]} & W_{2,3}^{[1]} \end{bmatrix} \in \mathbb{R}^{2 \times 3} \qquad b^{[2]} = \begin{bmatrix} b_1^{[2]} \end{bmatrix} \in \mathbb{R}^{1 \times 1} \qquad W^{[2]} = \begin{bmatrix} W_{1,1}^{[2]} & W_{1,2}^{[2]} \end{bmatrix} \in \mathbb{R}^{1 \times 2}$$

$a_1^{[0]} = x_1$
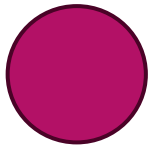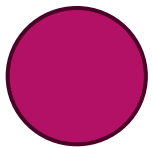
$a_2^{[0]} = x_2$

$a_3^{[0]} = x_3$

$a_1^{[1]}$

$a_2^{[1]}$

$W_{1,1}^{[2]}$

$W_{1,2}^{[2]}$

$a_1^{[2]} = g^{[2]}\left(W_1^{[2]} a^{[1]} + b_1^{[2]} \right) = h_\theta(x)$

A Multi-Layered Perceptron (Illustration)
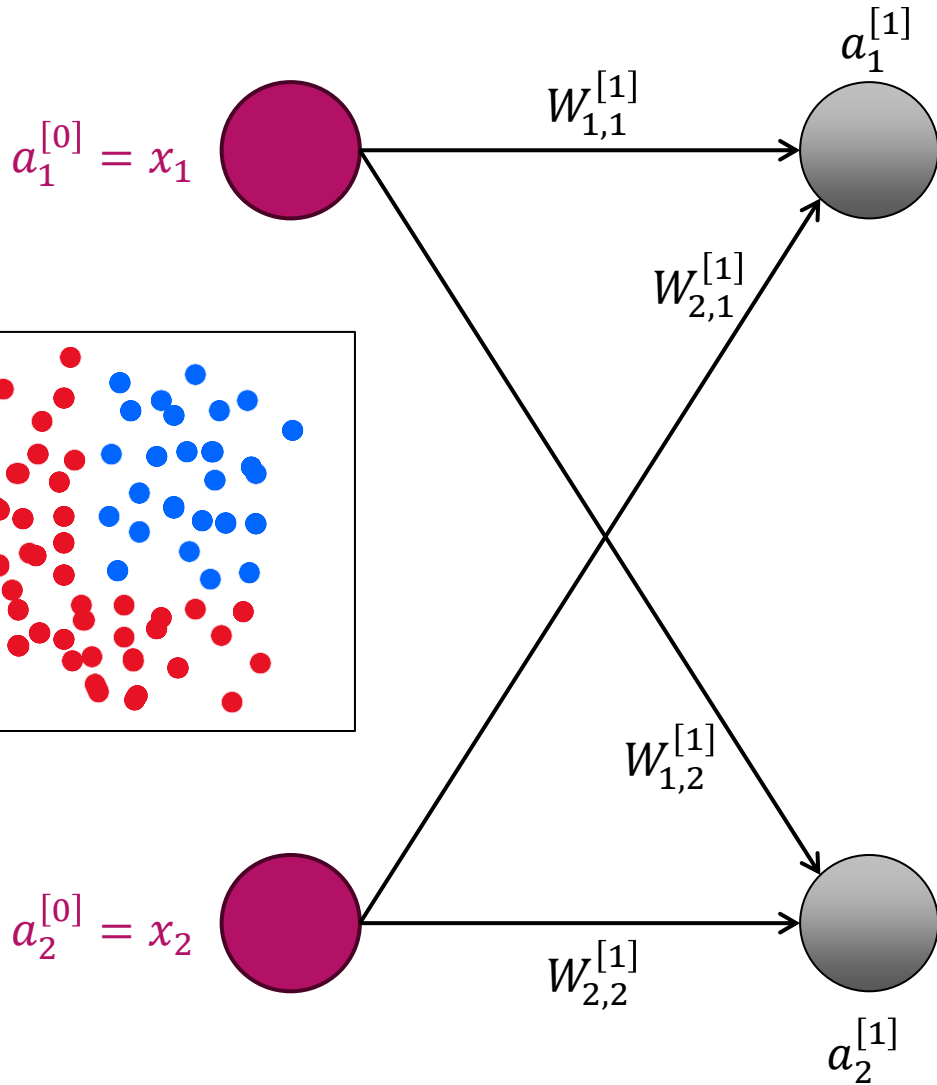
$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$
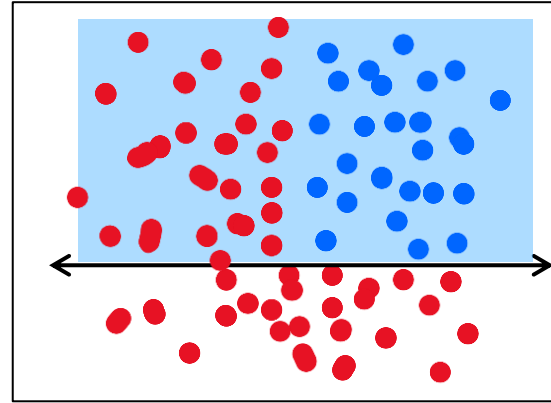
Visualizing the Decision Boundary

Visualizing the Decision Boundary

Linear Decision Boundary

Linear Decision Boundary

$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$

$W_{1,1}^{[1]}$

$W_{2,1}^{[1]}$
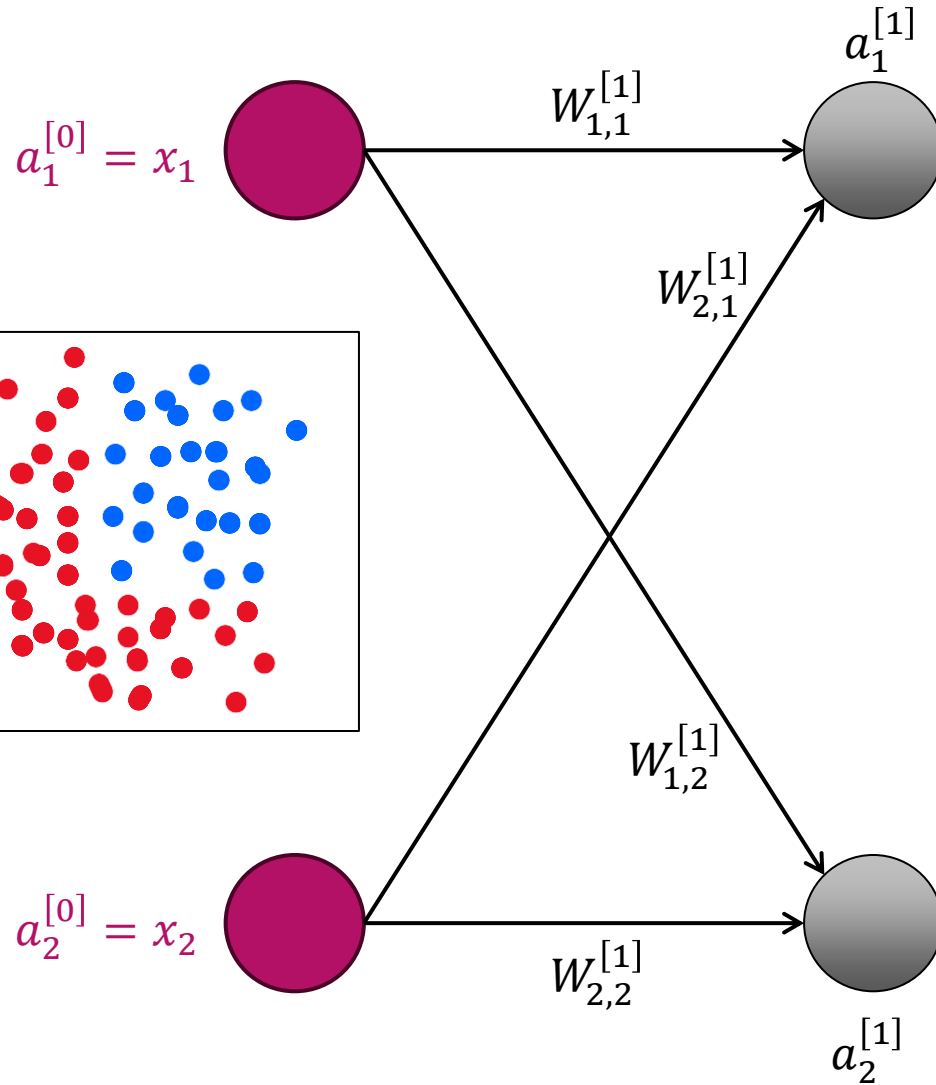
$W_{1,2}^{[1]}$

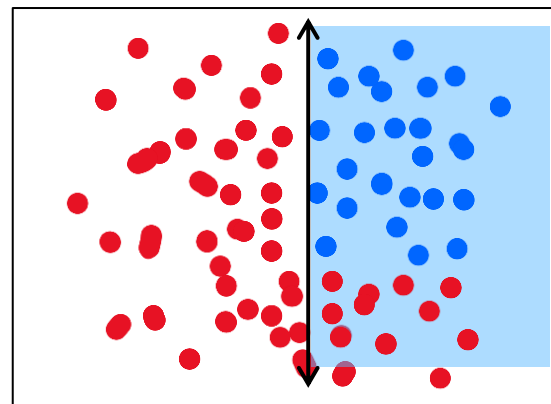$W_{2,2}^{[1]}$

$a_1^{[1]}$

$a_2^{[1]}$

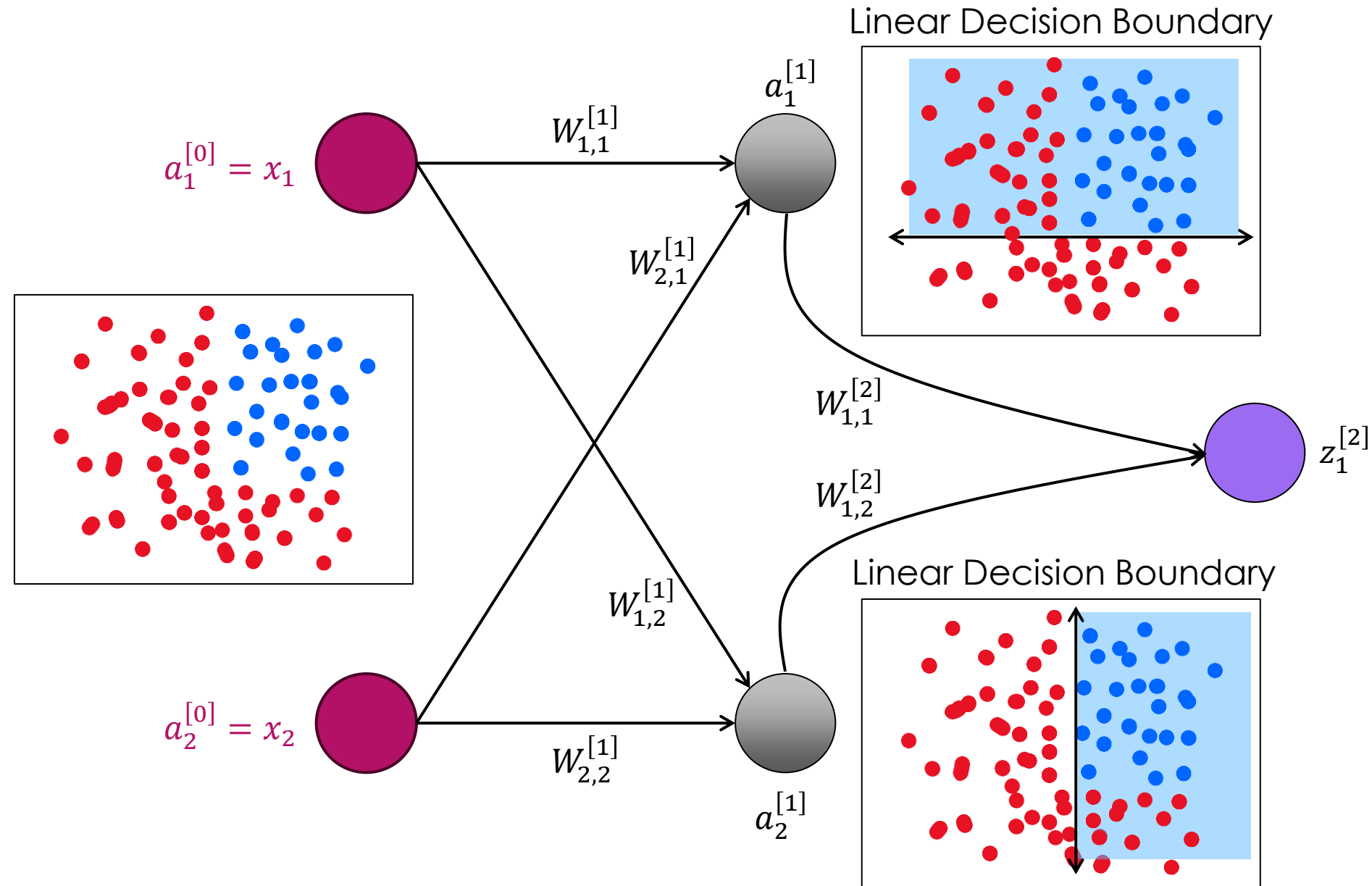# Visualizing the Decision Boundary

Linear Decision Boundary

Linear Decision Boundary

Visualizing the Decision Boundary

$a_1^{[0]} = x_1$

$W_{1,1}^{[1]}$

$a_1^{[1]}$

Linear Decision Boundary

$W_{2,1}^{[1]}$

$W_{1,1}^{[2]}$

$W_{1,2}^{[2]}$

$W_{1,2}^{[1]}$

$z_1^{[2]}$

Overlapping Linear Boundaries

$a_2^{[0]} = x_2$

$W_{2,2}^{[1]}$

Linear Decision Boundary

$a_2^{[1]}$

Visualizing the Decision Boundary

Linear Decision Boundary

$a_1^{[1]}$

$W_{1,1}^{[2]}$

$W_{1,2}^{[2]}$

$z_1^{[2]}$

$a_2^{[1]}$

Linear Decision Boundary

Overlapping Linear Boundaries

Visualizing the Decision Boundary

Linear Decision Boundary

$a_1^{[1]}$

$W_{1,1}^{[2]}$
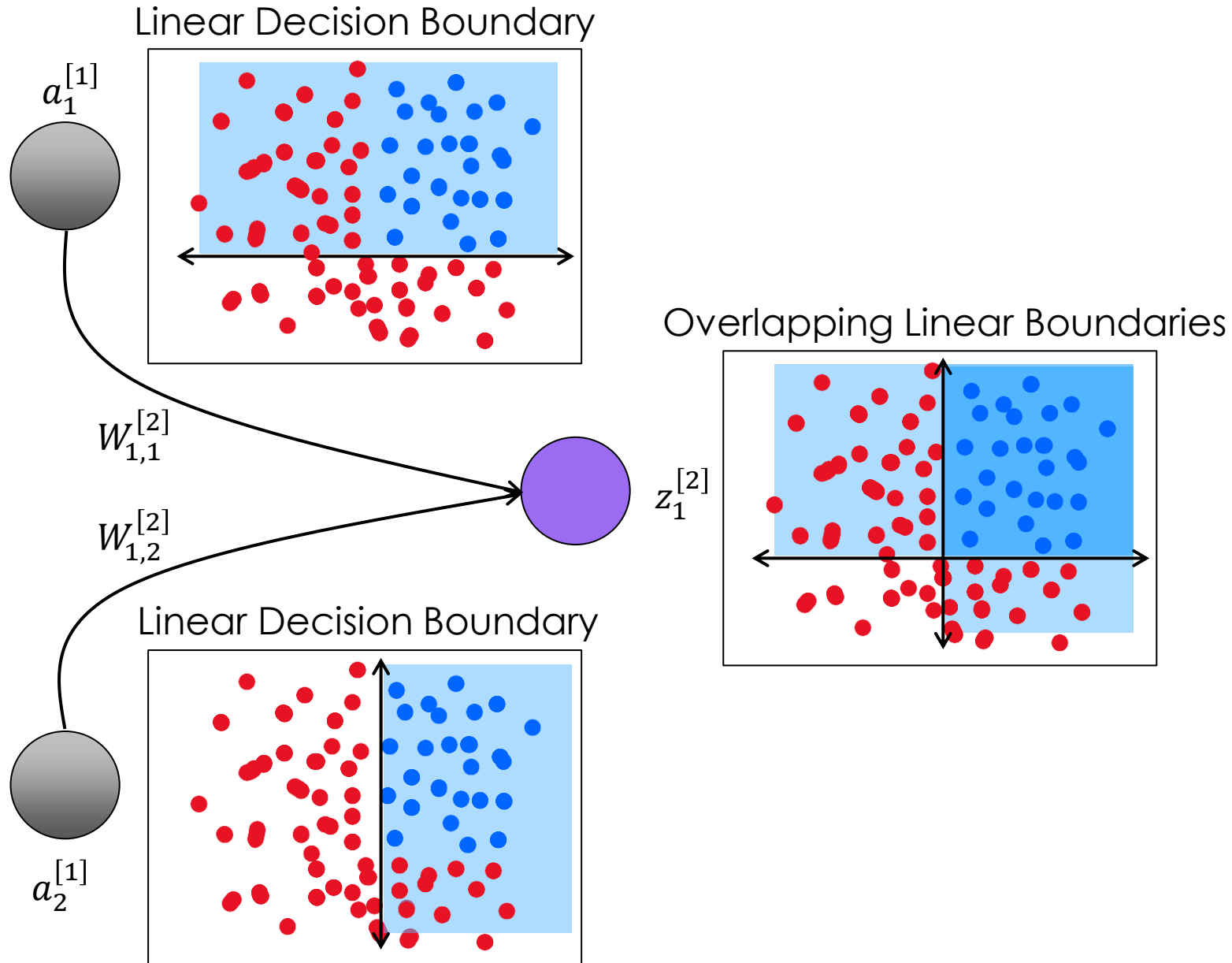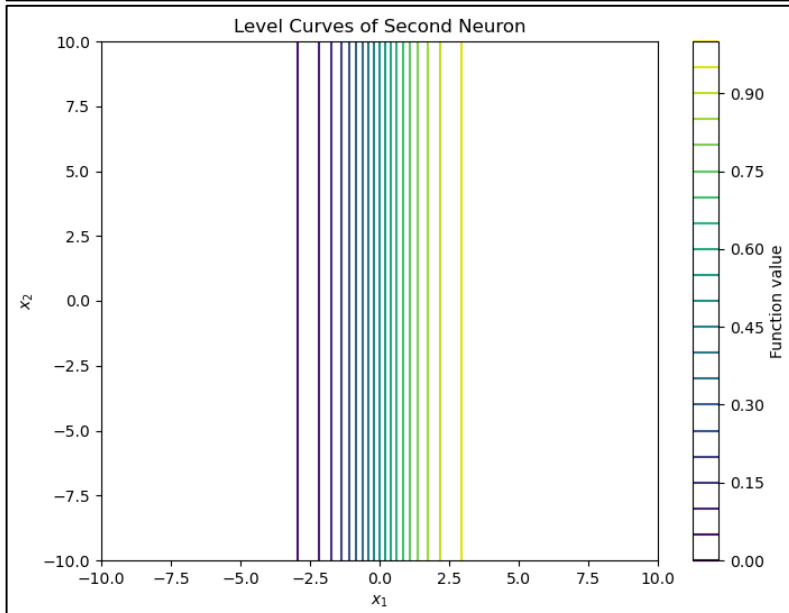
$W_{1,2}^{[2]}$

Linear Decision Boundary

$a_2^{[1]}$

$z_1^{[2]}$

Overlapping Linear Boundaries

Non-Linear Decision Boundary
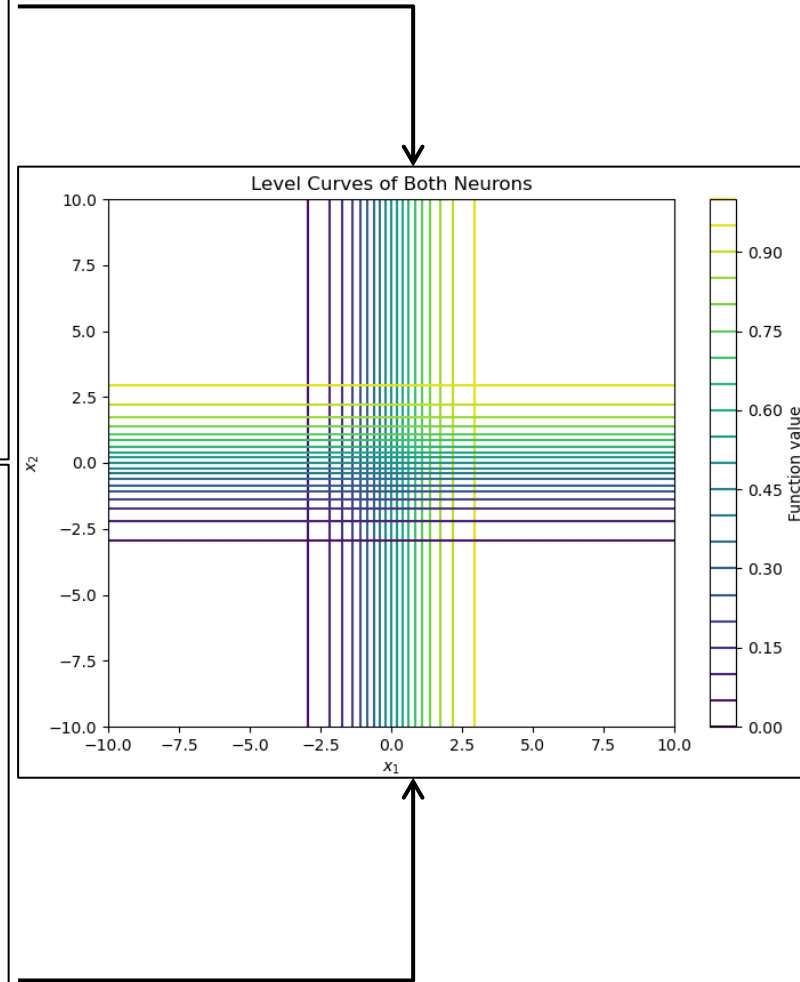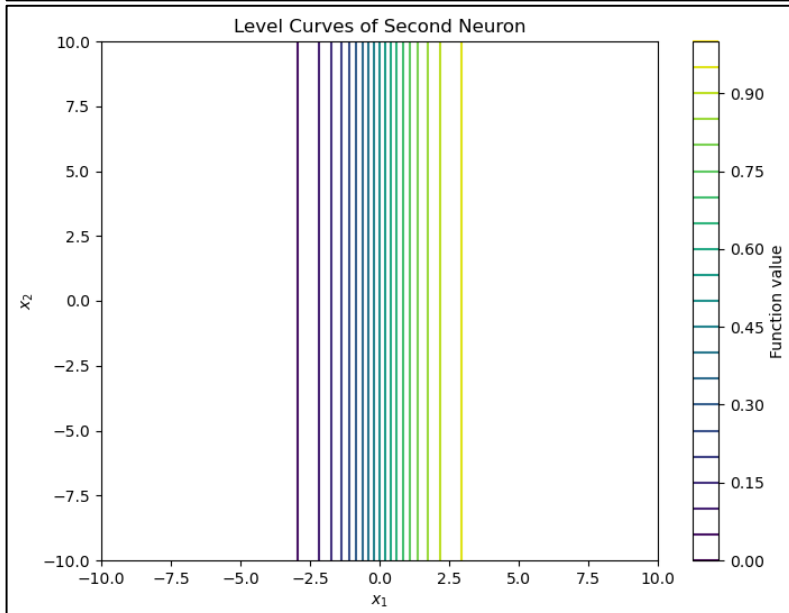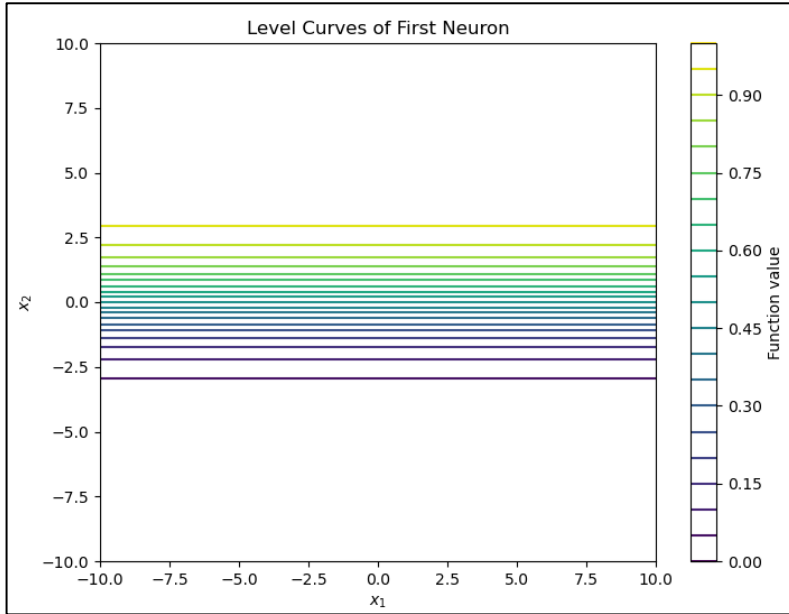
Apply Non-Linear Activation Function $g^{[2]}$

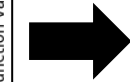$$a_1^{[2]} = g^{[2]}\left(z_1^{[2]}\right)$$
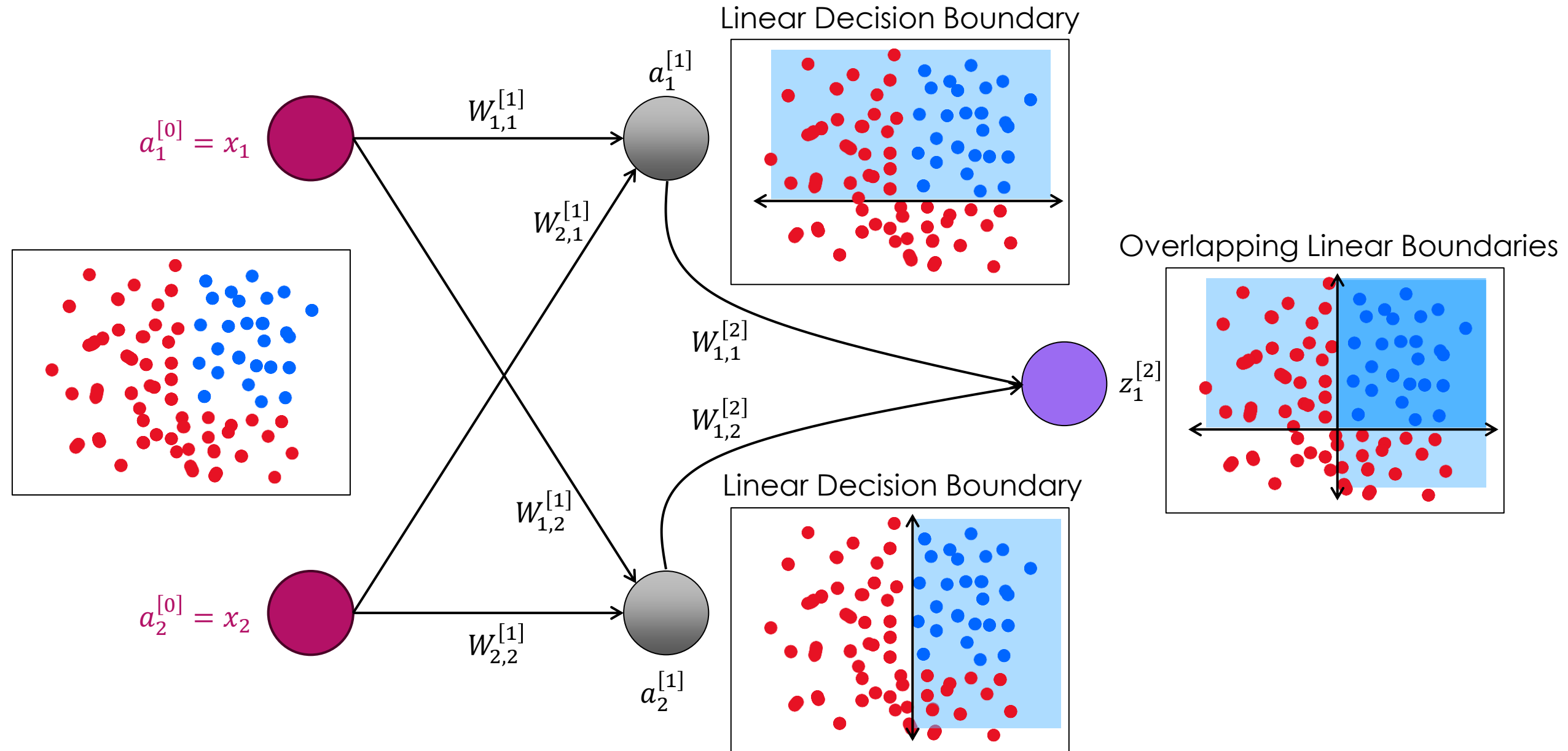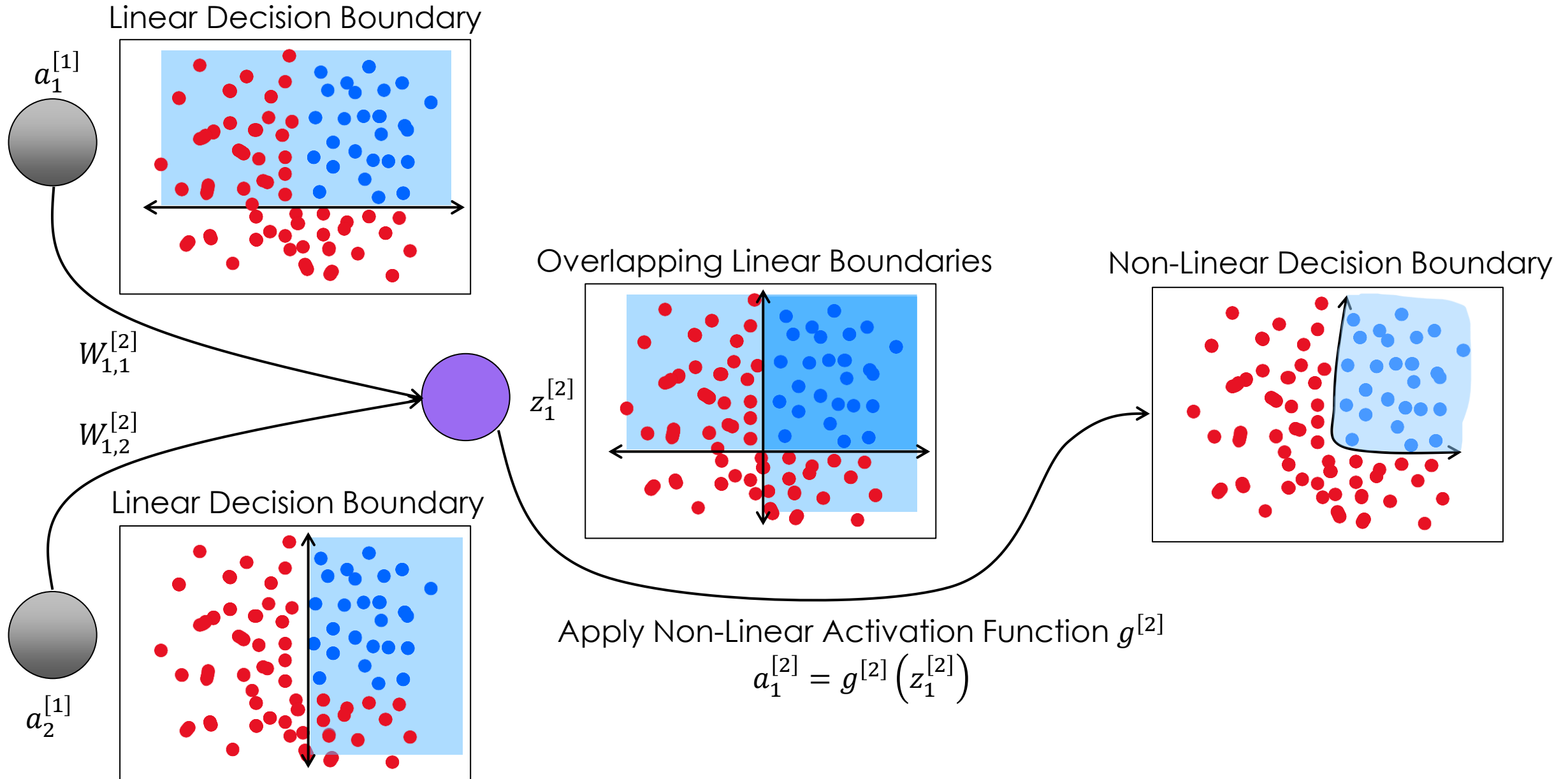
Level-Curves of Neurons
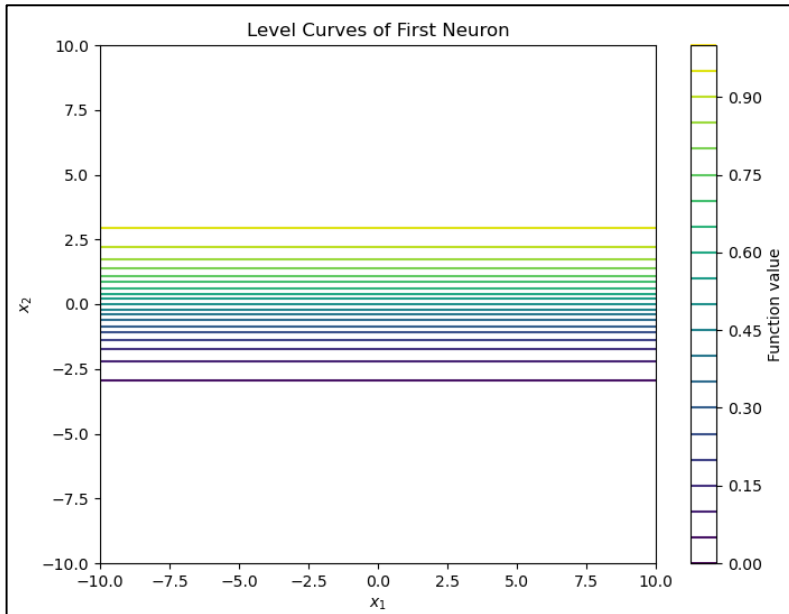
Level-Curves of Neurons

Level-Curves of Neurons
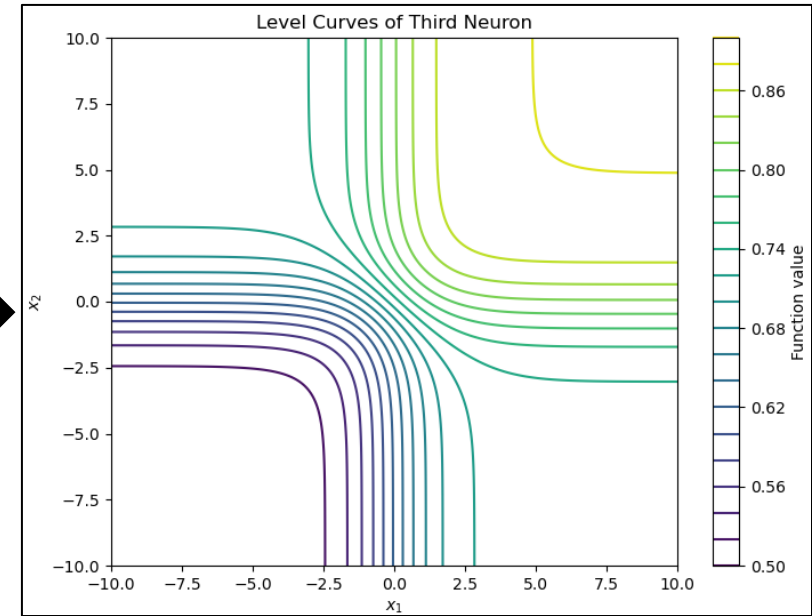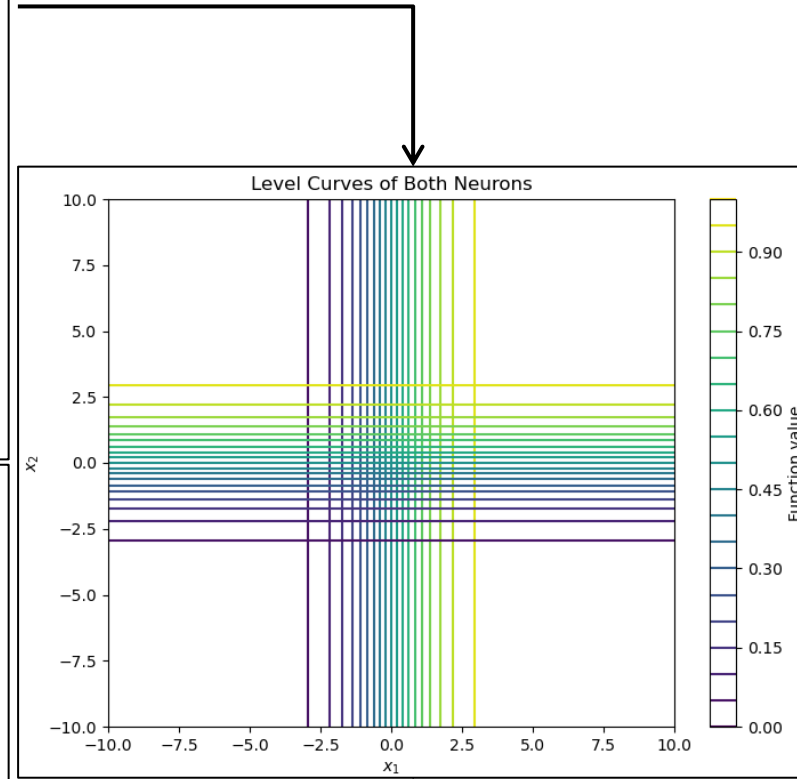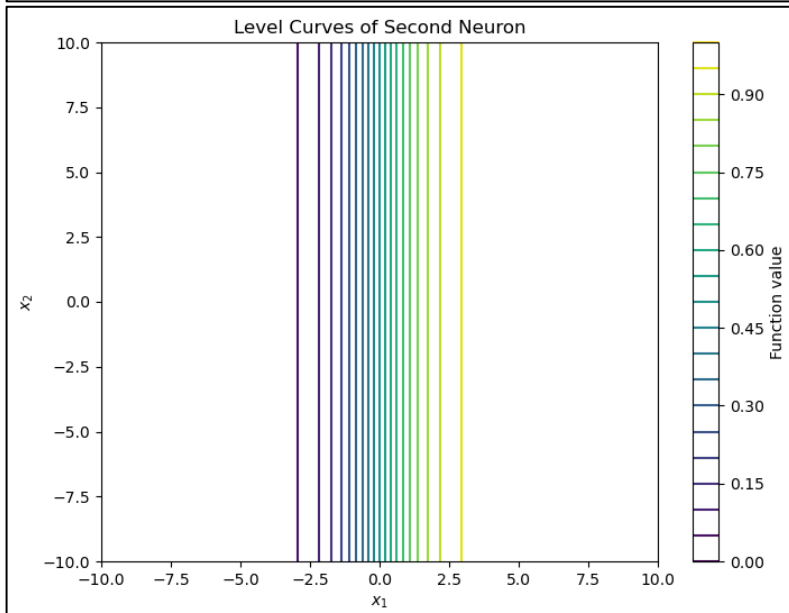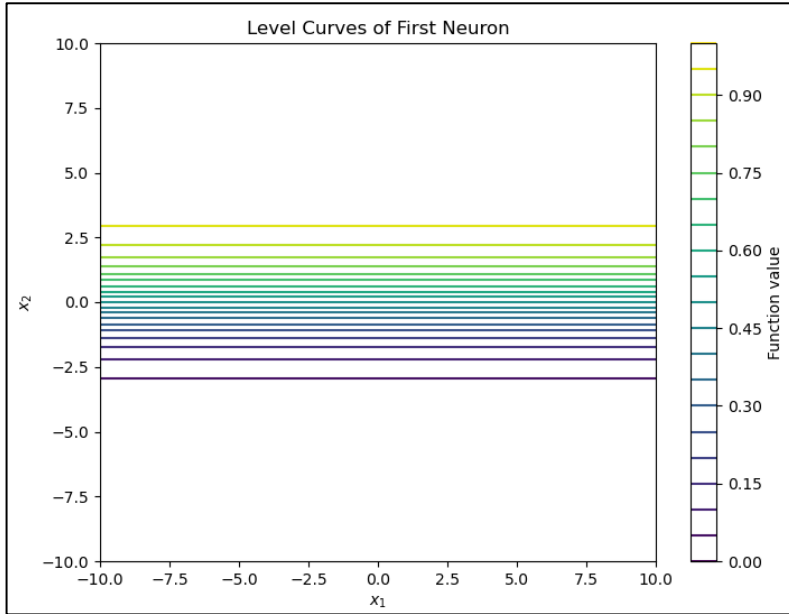
Forward Propagation (Single Datapoint)

Let $\mathcal{N}$ be a feed-forward neural network that consists of L layers, parameters $\theta := \{W, b\}$, where $W := \{W^{[\ell]}\}_{\ell=1}^{L}$ and $b := \{b^{[\ell]}\}_{\ell=1}^{L}$, and a pre-defined series of activation functions g for each layer, given by $g := \{g^{[\ell]}\}_{\ell=1}^{L}$. The **activation nodes of each neuron**, for a **single datapoint** $x^{(i)}$, are then computed layer-by-layer by the following equation (where $a^{[0]} = x^{(i)}$)

$$a_j^{[\ell]} = g^{[\ell]}\left(z_j^{[\ell]}\right) = g^{[\ell]}\left(W_j^{[\ell]} a^{[\ell-1]} + b_j^{[\ell]}\right),$$

for all $j \in \{1, 2, \dots, n[\ell]\}$ and $\ell \in \{1, 2, \dots, L\}$.

However, this requires two for-loops: (1) loop through the layers and then (2) loop through the neurons within each layer.

We can speed up computation by writing all the outputs in **vector notation**. This can be achieved by applying the activation function $g(\cdot)$ element-wise to the resulting vector $z^{[\ell]}$, yielding the equation

$$a^{[\ell]} = g^{[\ell]}(z^{[\ell]}) = g^{[\ell]}(W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}),$$

for all $\ell \in \{1, 2, \dots, L\}$.

Forward-Pass (Batch)

Typically, we are interested in computing the forward-pass on multiple datapoints at once (as is the case for either full gradient descent or stochastic gradient descent).

Forward Propagation (Batch)

Let $\mathcal{N}$ be a feed-forward neural network that consists of L layers, parameters $\theta := \{W, b\}$, where $W := \{W^{[\ell]}\}_{\ell=1}^{L}$ and $b := \{b^{[\ell]}\}_{\ell=1}^{L}$, and a pre-defined series of activation functions g for each layer, given by $g := \{g^{[\ell]}\}_{\ell=1}^{L}$. The **activation nodes of each neuron**, for a **batch of datapoints $X \in \mathbb{R}^{m \times n}$**, are then computed layer-by-layer by the following equation (where $A^{[0]} = X^T$)
$$\boldsymbol{A}^{[\ell]} = \boldsymbol{g}^{[\ell]}(\boldsymbol{Z}^{[\ell]}) = \boldsymbol{g}^{[\ell]}(\boldsymbol{W}^{[\ell]}\boldsymbol{A}^{[\ell-1]} + \boldsymbol{b}^{[\ell]}),$$
for all $\ell \in \{1, 2, \dots, L\}$. Here, we define the new matrices $Z^{[\ell]} := [z^{[\ell](1)}, z^{[\ell](2)}, \dots, z^{[\ell](m)}] \in \mathbb{R}^{n[\ell] \times m}$ and $A^{[\ell]} := [a^{[\ell](1)}, a^{[\ell](2)}, \dots, a^{[\ell](m)}] \in \mathbb{R}^{n[\ell] \times m}$, which hold the m output vectors that correspond to each of the m datapoints. Notice that the addition of the $\mathbb{R}^{n[\ell]}$ column vector of bias terms $b^{[\ell]}$ is interpreted as "adding the vector $b^{[\ell]}$ to each of the columns of the resulting matrix $W^{[\ell]}A^{[\ell-1]}$" (This is also referred to as "broadcasting").

Training Deep Neural Networks

BACK-PROPAGATION AND THE CHAIN RULE

Overview of Training Neural Networks

As technology has increased over the years, advancing the development of computer hardware, data warehouses, and overall power, the sizes of neural network models that can be trained is vast and large.
- "**Small**" NNs typically always contain parameters in the **hundreds of thousands**.
- "**Medium**" sized NNs can range anywhere between having **millions to billions** of parameters.
- "**Large**" modern day NNs can have **hundreds of billions** and even more than **trillions** of parameters…

To train models this large, one typically needs an **exorbitant amount of data** (for modern day network training, start thinking of the **<u>entirety of the internet</u>**…) to learn the best possible patterns. Considering all of this, it immediately becomes apparent that one needs to implement efficient strategies when training these models. The first is the use of **Stochastic Gradient Descent**.

Summary of SGD and its Variants for Deep Learning

As discussed in the "*Numerical Optimization*" lecture slides, SGD differs from regular gradient descent in terms of its practical implementation by simply utilizing "**mini-batches**" of datapoints to compute approximations of the gradient (i.e., **stochastic gradients**) instead of using the entire dataset (which would yield the true gradient). As such, this is why SGD is the choice of optimization algorithm for training deep NNs; because it is **computationally efficient** and still **has convergence guarantees**.
- Specifically, there are many different types of variants of SGD that have been developed for training NNs that are often used. Some of these methods include Adam, AdamW, Nesterov Accelerated SGD (a method that incorporates "momentum"), AdaGrad, RMSprop, etc.

Typical Loss Functions for Neural Networks

Choice of Loss Function for Training a Neural Network

Consider some general neural network $h_\theta(x) := \mathcal{N}(x; g, \theta)$. Then, depending on the type of problem that one is trying to solve (i.e., regression, classification, etc.) given some dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, one will choose an appropriate loss function $J(\theta)$ to minimize via some optimization procedure. Here are some examples of loss functions that are typically used for certain tasks.

- **Regression:** When $y \in \mathbb{R}$, it is common to utilize the **squared error loss function** given by

$$J(\theta; x^{(i)}, y^{(i)}) = \frac{1}{2} \left\| h_\theta(x^{(i)}) - y^{(i)} \right\|_2^2.$$

- **Binary Classification:** When $y \in \{0,1\}$, it is common to utilize the **cross-entropy loss function** (where $g^{[L]} = \sigma(\cdot)$) given by

$$J(\theta; x^{(i)}, y^{(i)}) = -\left[y^{(i)} \log\left(h_\theta(x^{(i)}) \right) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)}) \right) \right].$$

- **Multi-Class Classification:** When $y \in \{1, 2, \ldots, k\}$, it is common to utilize the **cross-entropy loss function for the general $k \in \mathbb{N}$** scenario (where $g^{[L]} = \text{softmax}(\cdot)$) given by

$$J(\theta; x^{(i)}, y^{(i)}) = -\sum_{j=1}^{k} \log\left(h_\theta(x^{(i)})_j \right) \mathbb{I}\left[y^{(i)} = j \right].$$

Notice that the $h_\theta(x^{(i)})_j$ denotes the j-th element of the softmax function in this expression.

The Network Computation Graph

Forward Propagation

Given a datapoint (or batch of datapoints), one needs to compute the values of the activations corresponding to the input data layer-by-layer in the order $\ell \in \{1, 2, \dots, L\}$. This will yield the loss value J associated with this data. This is referred to as **forward propagation** (or the **forward-pass**).
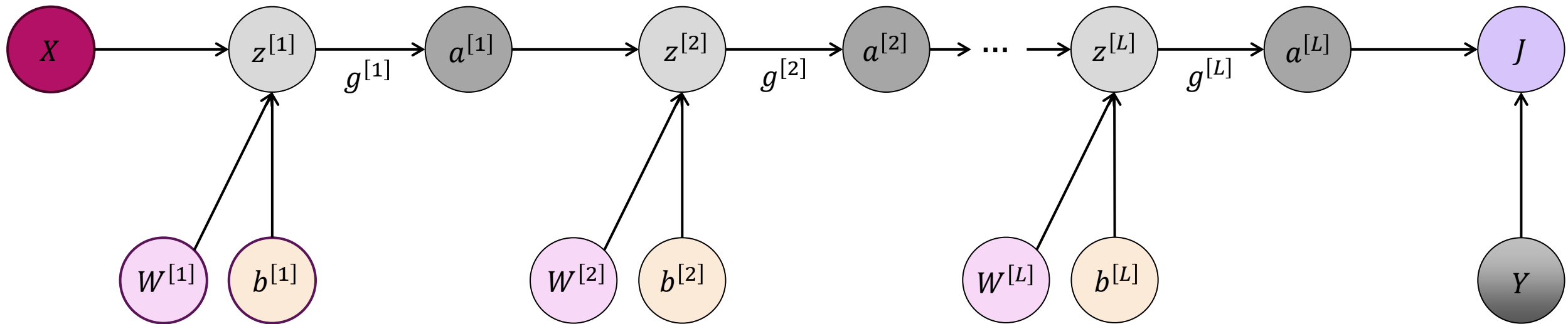
The Network Computation Graph
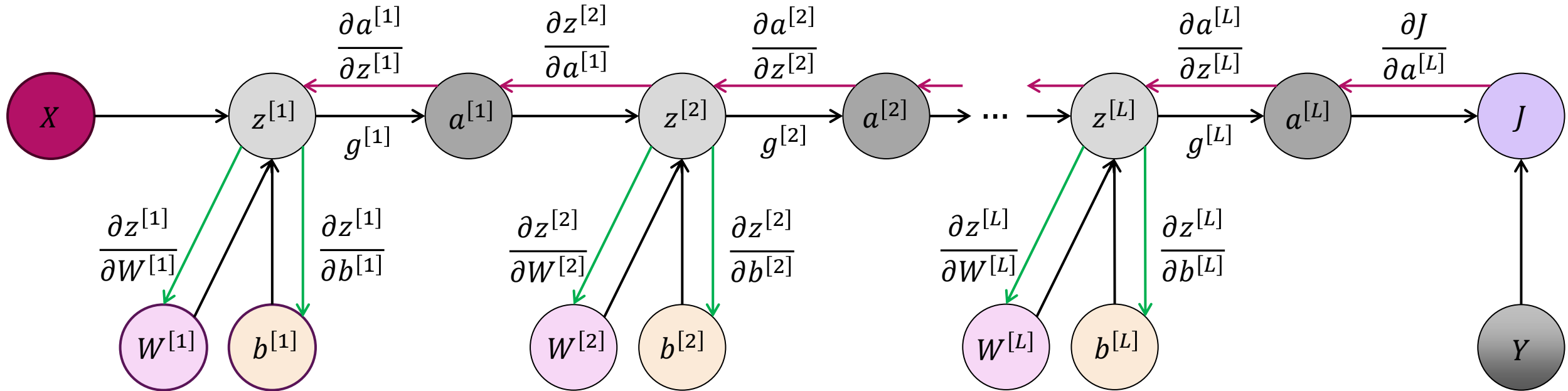
Forward Propagation

Given a datapoint (or batch of datapoints), one needs to compute the values of the activations corresponding to the input data layer-by-layer in the order $\ell \in \{1, 2, \ldots, L\}$. This will yield the loss value J associated with this data. This is referred to as **forward propagation** (or the **forward-pass**).

Backward Propagation

Once the forward-pass has computed the loss of some data, the gradients of the network parameters can be computed in a "backward" fashion layer-by-layer in the order $\ell \in \{L, L-1, \ldots, 2, 1\}$. These gradients are computed by consecutively **applying the chain rule**. This is referred to as **backward propagation** (or **back-propagation**).

Gradients of the Last Layer L

- Recall that we define an MLP $\mathcal{N}$ as
$$h_\theta(x) \coloneqq \mathcal{N}(x; g, \theta \coloneqq \{W, b\}) \coloneqq g^{[L]}\left(W^{[L]}g^{[L-1]}\left(\ldots\left(W^{[2]}g^{[1]}\left(W^{[1]}x + b^{[1]}\right) + b^{[2]}\right)\ldots\right) + b^{[L]}\right).$$
- Further, we can rewrite this in terms of the network's final layer L as
$$h_\theta(x) = a^{[L]} = g^{[L]}\left(z^{[L]}\right) = g^{[L]}\left(W^{[L]}a^{[L-1]} + b^{[L]}\right).$$
- We can continue expanding this expression out layer-by-layer until we obtain the expression above.
- We wish to derive the partial derivatives of the loss function $J(\theta; x, y)$ with respect to $W^{[\ell]}$ and $b^{[\ell]}$ for all layers $\ell \in \{1, 2, \ldots, L\}$.
- Notice that the gradient of the parameters in the ℓ-th layer (for any $\ell < L$) is dependent on the gradient of the parameters in the $(\ell + 1)$-th layer, all the way up to the final layer L. To illustrate this point, we can utilize the chain rule to derive expressions for the partial derivatives of the terms relating to the final layer L. The terms are given in the order that they are required to be computed:

$$(1): \frac{\partial J}{\partial a^{[L]}} \in \mathbb{R}^{n[L]},$$

$$(2): \frac{\partial J}{\partial z^{[L]}} = \frac{\partial J}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} \in \mathbb{R}^{n[L]},$$

$$(3): \frac{\partial J}{\partial W^{[L]}} = \frac{\partial J}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial W^{[L]}} = \frac{\partial J}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial W^{[L]}} \in \mathbb{R}^{n[L] \times n[L-1]},$$

$$(4): \frac{\partial J}{\partial b^{[L]}} = \frac{\partial J}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial b^{[L]}} = \frac{\partial J}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial b^{[L]}} \in \mathbb{R}^{n[L]},$$

$$(5): \frac{\partial J}{\partial a^{[L-1]}} = \frac{\partial J}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial a^{[L-1]}} = \frac{\partial J}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial a^{[L-1]}} \in \mathbb{R}^{n[L-1]}.$$

- All these terms are in-turn required for the partial derivatives corresponding to the earlier layers.

Gradients of $a^{[\ell]}$ and $z^{[\ell]}$ for a General Layer ℓ

- We wish to derive general expressions for the gradient of the parameters for each layer $\ell \in \{1, 2, \dots, L\}$.
- Based on the forward-propagation, we know that mapping from $a^{[\ell]} \in \mathbb{R}^{n[\ell]}$ to $z^{[\ell+1]} \in \mathbb{R}^{n[\ell+1]}$ is a linear function given by $z^{[\ell+1]} = W^{[\ell+1]} a^{[\ell]} + b^{[\ell+1]}$, where $W^{[\ell+1]} \in \mathbb{R}^{n[\ell+1] \times n[\ell]}$. The partial derivative $\frac{\partial J}{\partial a^{[\ell]}} \in \mathbb{R}^{n[\ell]}$ is given by the equation (where we assume that the partial $\frac{\partial J}{\partial z^{[\ell+1]}}$ has been computed)

$$\boxed{\frac{\partial J}{\partial a^{[\ell]}}} = \boxed{\frac{\partial z^{[\ell+1]}}{\partial a^{[\ell]}}} \cdot \boxed{\frac{\partial J}{\partial z^{[\ell+1]}}} = \boxed{\left(W^{[\ell+1]}\right)^T} \boxed{\frac{\partial J}{\partial z^{[\ell+1]}}}$$

$(n[\ell] \times 1)$ $(n[\ell] \times n[\ell+1])$ $(n[\ell+1] \times 1)$ $(n[\ell] \times n[\ell+1])$ $(n[\ell+1] \times 1)$

- Recall that the vector $a^{[\ell]} \in \mathbb{R}^{n[\ell]}$ is obtained from the equation $a^{[\ell]} = g^{[\ell]}(z^{[\ell]})$, where the activation function $g^{[\ell]}$ is applied element-wisely to the vector $z^{[\ell]} \in \mathbb{R}^{n[\ell]}$. The partial derivative $\frac{\partial J}{\partial z^{[\ell]}} \in \mathbb{R}^{n[\ell]}$ is given by the equation (where $\left(g^{[\ell]}\right)'(z^{[\ell]})$ is the element-wise derivative of $g^{[\ell]}$ evaluated at $z^{[\ell]}$) (where $\otimes$ denotes the element-wise multiplication operator)

$$\boxed{\frac{\partial J}{\partial z^{[\ell]}}} = \boxed{\frac{\partial a^{[\ell]}}{\partial z^{[\ell]}}} \otimes \boxed{\frac{\partial J}{\partial a^{[\ell]}}} = \boxed{\left[\left(g^{[\ell]}\right)'(z^{[\ell]})\right]} \otimes \boxed{\left[\left(W^{[\ell+1]}\right)^T \frac{\partial J}{\partial z^{[\ell+1]}}\right]}$$

$(n[\ell] \times 1)$ $(n[\ell] \times 1)$ $(n[\ell] \times 1)$ $(n[\ell] \times 1)$ $(n[\ell] \times 1)$

Gradients of $W^{[\ell]}$ and $b^{[\ell]}$ for a General Layer ℓ

- With the knowledge of the partial derivatives derived on the previous slide, one can now derive expressions for the partial derivatives with respect to the parameters of interest $W^{[\ell]}$ and $b^{[\ell]}$ for a general layer $\ell \in \{1, 2, \dots, L\}$.

- Starting from the expression $\frac{\partial J}{\partial z^{[\ell]}}$, we can obtain the expression of the partial derivative $\frac{\partial J}{\partial W^{[\ell]}}$ as

$$\frac{\partial J}{\partial W^{[\ell]}} = \frac{\partial J}{\partial z^{[\ell]}} \cdot \frac{\partial z^{[\ell]}}{\partial W^{[\ell]}} = \frac{\partial J}{\partial z^{[\ell]}} \cdot \left(a^{[\ell-1]}\right)^T$$

$$(n[\ell] \times n[\ell-1]) \quad (n[\ell] \times 1) \quad (1 \times n[\ell-1]) \quad (n[\ell] \times 1) \quad (1 \times n[\ell-1])$$

- In a similar fashion, one can obtain the expression of the partial derivative $\frac{\partial J}{\partial b^{[\ell]}}$ as (notice that the result follows because $\frac{\partial z^{[\ell]}}{\partial b^{[\ell]}} = \hat{1} \in \mathbb{R}^{n[\ell]}$, where $\hat{1}$ denotes a vector of 1s in $\mathbb{R}^{n[\ell]}$, and as a result the partial derivative $\frac{\partial J}{\partial b^{[\ell]}}$ can simply be written as the term $\frac{\partial J}{\partial z^{[\ell]}}$)

$$\frac{\partial J}{\partial b^{[\ell]}} = \frac{\partial J}{\partial z^{[\ell]}} \cdot \frac{\partial z^{[\ell]}}{\partial b^{[\ell]}} = \frac{\partial J}{\partial z^{[\ell]}}$$

$$(n[\ell] \times 1) \quad (n[\ell] \times 1) \quad (n[\ell] \times 1) \quad (n[\ell] \times 1)$$

Gradients of a Neural Network (Example)

Consider an MLP $h_\theta(x) := \mathcal{N}(x; g, \theta := \{W, b\})$ that is utilized to predict a binary target variable $y \in \{0,1\}$. Then, assume that the activation function of the final layer is he sigmoid function, i.e., $g^{[L]} = \sigma$, and further that

$$h_\theta(x) = a^{[L]} = \sigma(z^{[L]}) = \sigma(W^{[L]}a^{[L-1]} + b^{[L]}).$$

To train this network to predict the binary target y, we wish to minimize the cross-entropy loss function

$$J(h_\theta(x), y) = -y\log(h_\theta(x)) - (1-y)\log(1 - h_\theta(x)).$$

To do this, one we need to compute the gradients of this function with respect to the network parameters $W^{[\ell]}$ and $b^{[\ell]}$ for every layer of the network $\ell \in \{1, 2, \dots, m\}$. To that end, we have

$$\frac{\partial J}{\partial a^{[L]}} = -\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}.$$

$$\frac{\partial J}{\partial z^{[L]}} = \frac{\partial J}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = \left(-\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}\right)\left(\sigma(z^{[L]})\sigma(-z^{[L]})\right) = \left(-\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}\right)\left(a^{[L]}(1 - a^{[L]})\right) = a^{[L]} - y.$$

$$\frac{\partial J}{\partial W^{[L]}} = \frac{\partial J}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial W^{[L]}} = (a^{[L]} - y)(a^{[L-1]})^T.$$

$$\frac{\partial J}{\partial b^{[L]}} = \frac{\partial J}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial b^{[L]}} = a^{[L]} - y.$$

The gradients of the remaining layers can be obtained by starting with these expressions, using the general equations that were derived on the previous two slides. Again, once one obtains the general expressions for the gradients of this problem, they are obtained via the backpropagation algorithm.

Training an MLP

Now that we have discussed how the gradients of the network parameters are computed, we can finally introduce the general outline of the procedure for training a neural network.

Algorithm   Training an MLP

Input: The dataset $\mathcal{D} := \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{m}$, some MLP $\mathcal{N}$ with randomly initialized weights and biases, some loss function J, some sequence of learning rates $\{\alpha_j\}_{j=1}^{s}$, mini-batch size $B \in \mathbb{N}$, and the maximum number of iterations T.

For $t = 1, 2, \ldots, T$ **do**

    a) Sample a Batch. Randomly sample a mini-batch of size B from the dataset $\mathcal{D}$.

    b) Forward Propagation. Pass the mini-batch through the MLP via forward propagation to compute the activations and the loss.

    c) Backward Propagation. Compute the gradient of the loss with respect to the weights and biases of the network layer-by-layer, in the order $\ell \in \{L, L - 1, ..., 2, 1\}$, via back-propagation.

    d) Update Parameters. Using the gradients computed, update the network parameters via dome form of gradient descent (or stochastic gradient descent).
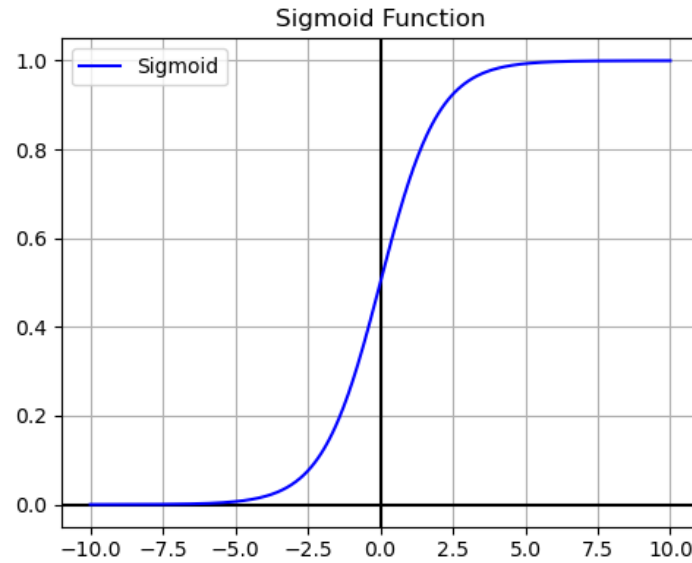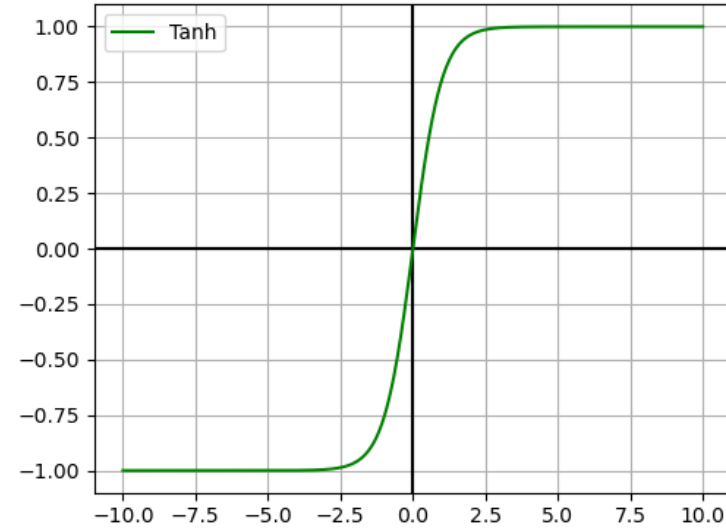
End For

Return the trained ML $\mathcal{N}$.

Activation Functions

Sigmoid

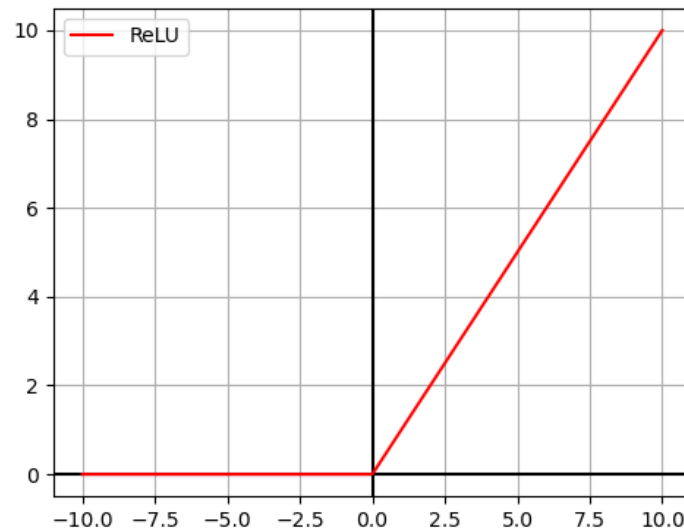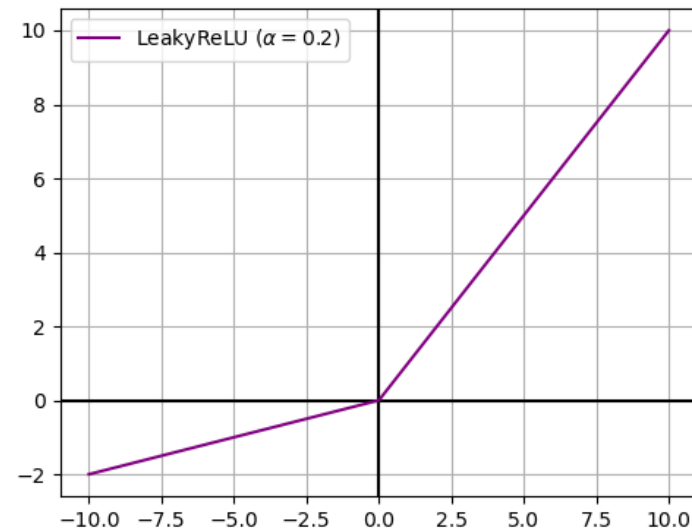$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Hyperbolic Tangent

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

ReLU

$$f(z) = \begin{cases} z, \forall z > 0 \\ 0, \text{otherwise} \end{cases}$$

LeakyReLU

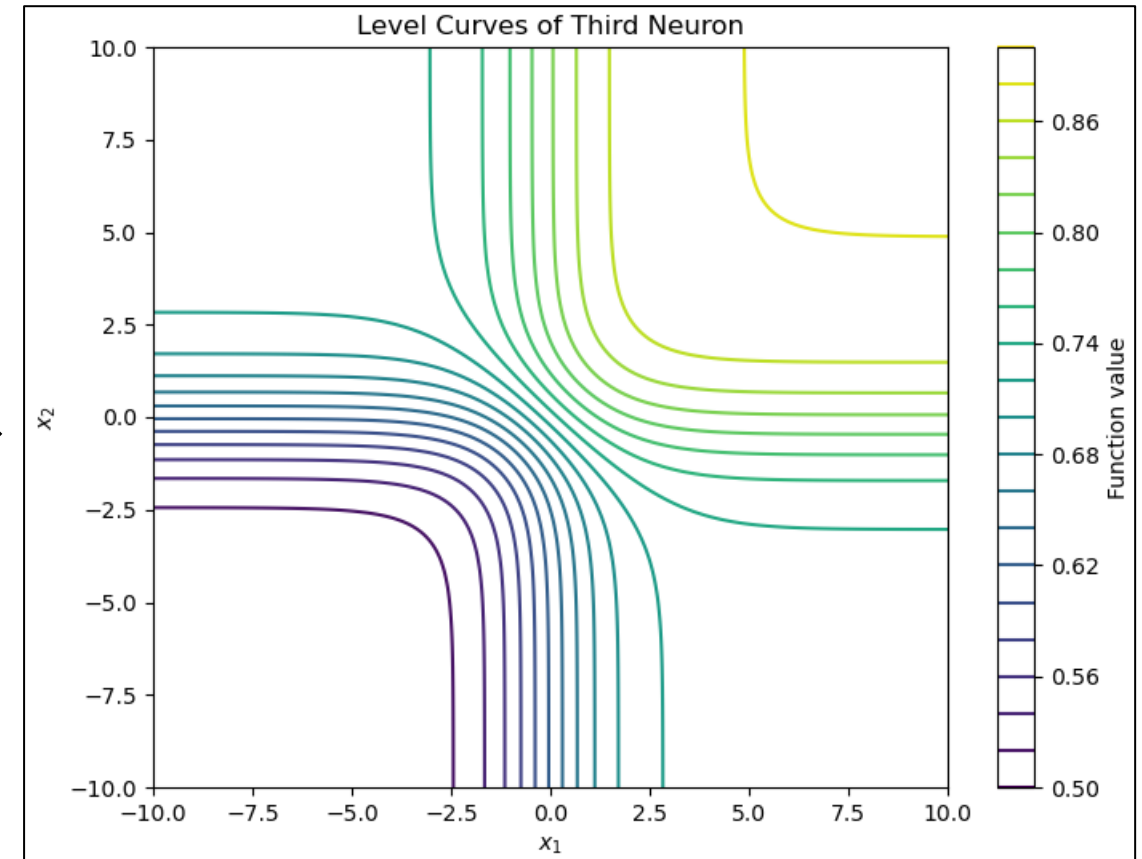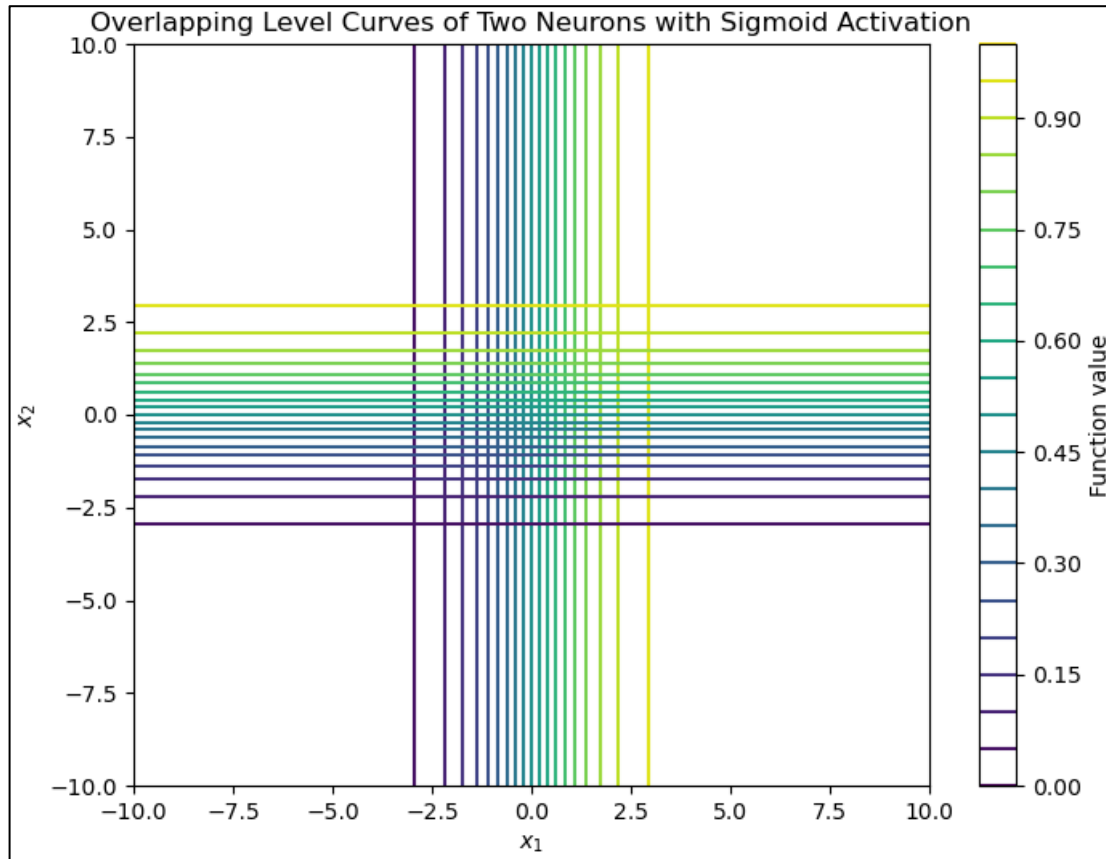$$f(z; \alpha) = \begin{cases} z, \forall z > 0 \\ \alpha z, \text{otherwise} \end{cases}$$

Activation Function Trade-Offs

Choosing nonlinear activation functions for the output of the neurons in a network is what allows NNs to learn highly-complex nonlinear patterns. As such, there are a variety of choices of activation functions that are most used for MLPs, each of which has their own pros and cons.
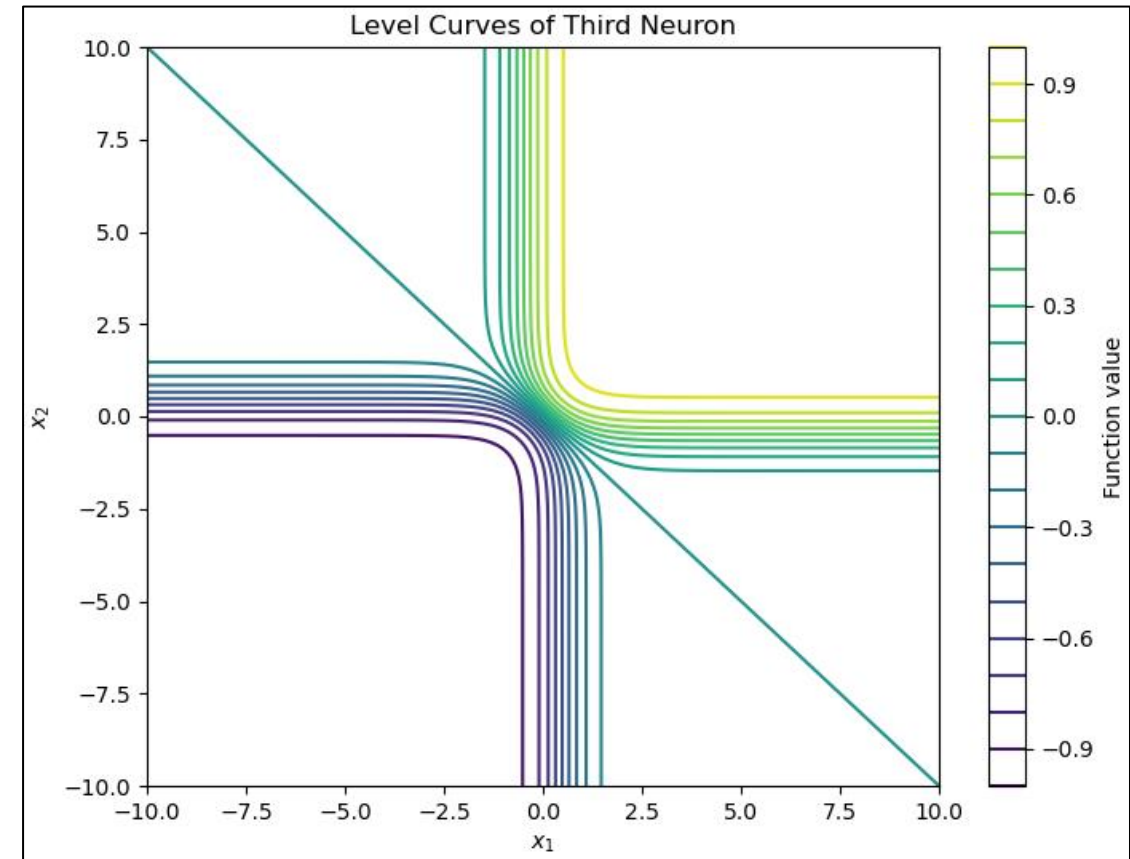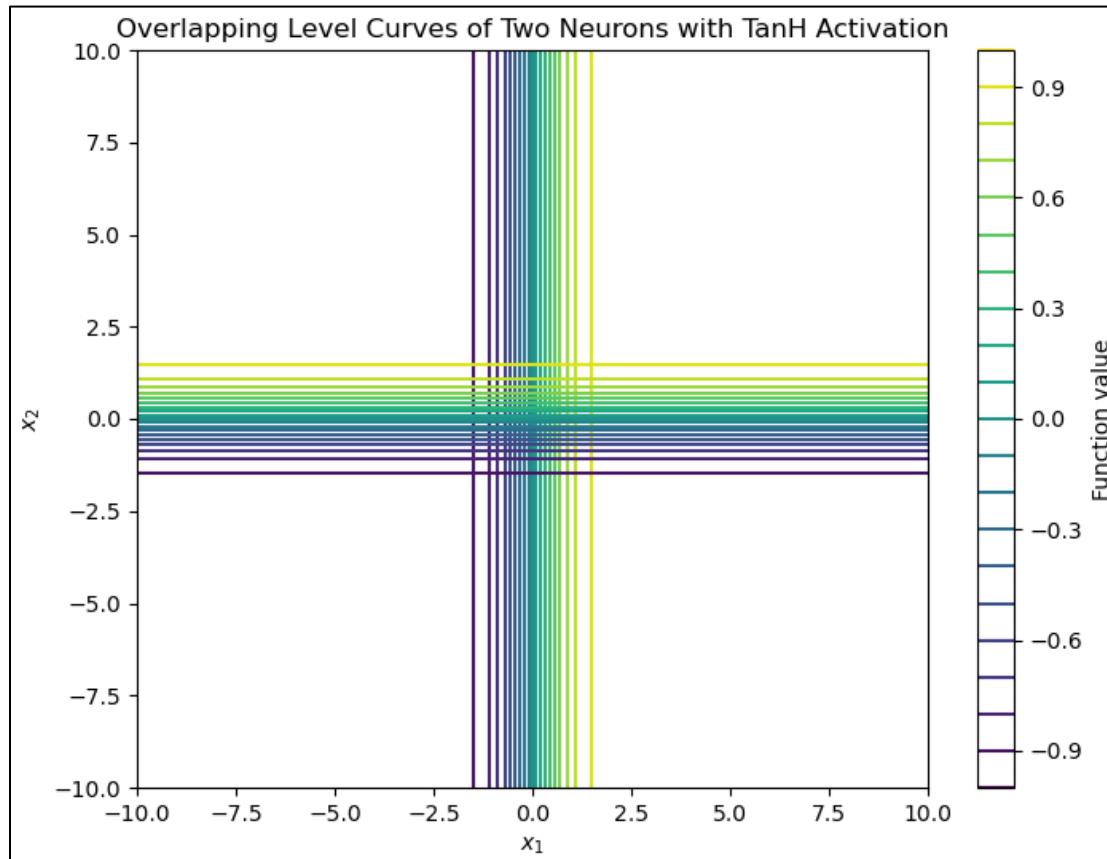
- **Sigmoid Function:** Outputs probabilities and is used as the output for binary classification problems (similarly, the softmax is used as the output for multi-class classification problems).
  - **Pros:** Smooth gradient (differentiable) and bounded range values (preventing extreme values).
  - **Cons:** For very large positive or negative inputs, the gradient is very close to 0, making convergence to a solution very slow (known as the Vanishing Gradient Problem). For this reason, the sigmoid function is not typically used as the activation function for the hidden layers.
- **Hyperbolic Tangent Function:** Similar to the sigmoid function but has a range of $(-1,1)$. Is typically used in hidden layers of recurrent neural networks.
  - **Pros:** Smooth gradient (differentiable), bounded, and more sensitive to changes near 0.
  - **Cons:** Still suffers from the vanishing gradient problem and is more computationally expensive.
- **Rectified Linear Unit (ReLU):** Introduces sparsity in NNs by setting negative values to 0. The default activation function for the hidden layers in most implementations of Deep NNs.
  - **Pros:** Avoids the vanishing gradient problem (faster convergence) and computationally efficient.
  - **Cons:** Neurons can become inactive if they return negative values too often (Dying ReLU Problem) since gradients will become 0; a problem that happens in deeper NNs.
- **LeakyReLU:** A simple modification of ReLU to allow for small, non-zero gradients for negative inputs.
  - **Pros:** Mitigates the Dying ReLU problem that often develops in deep NNs and efficient.
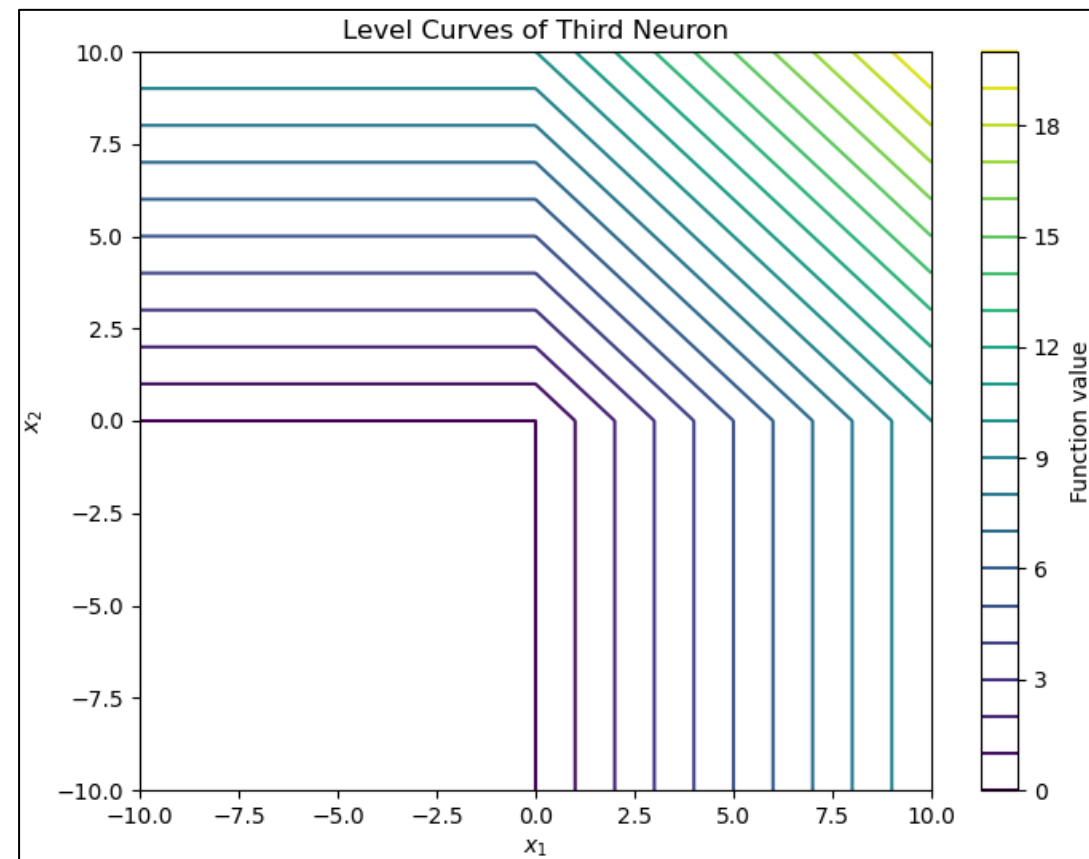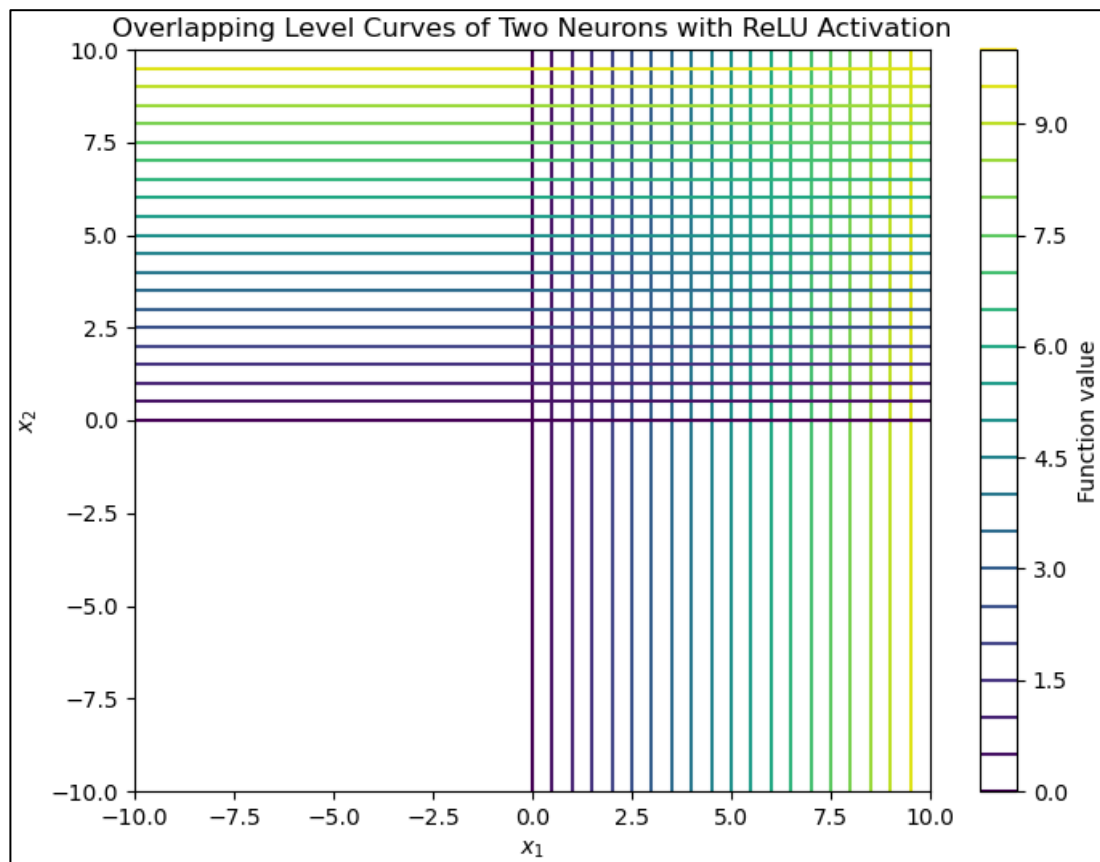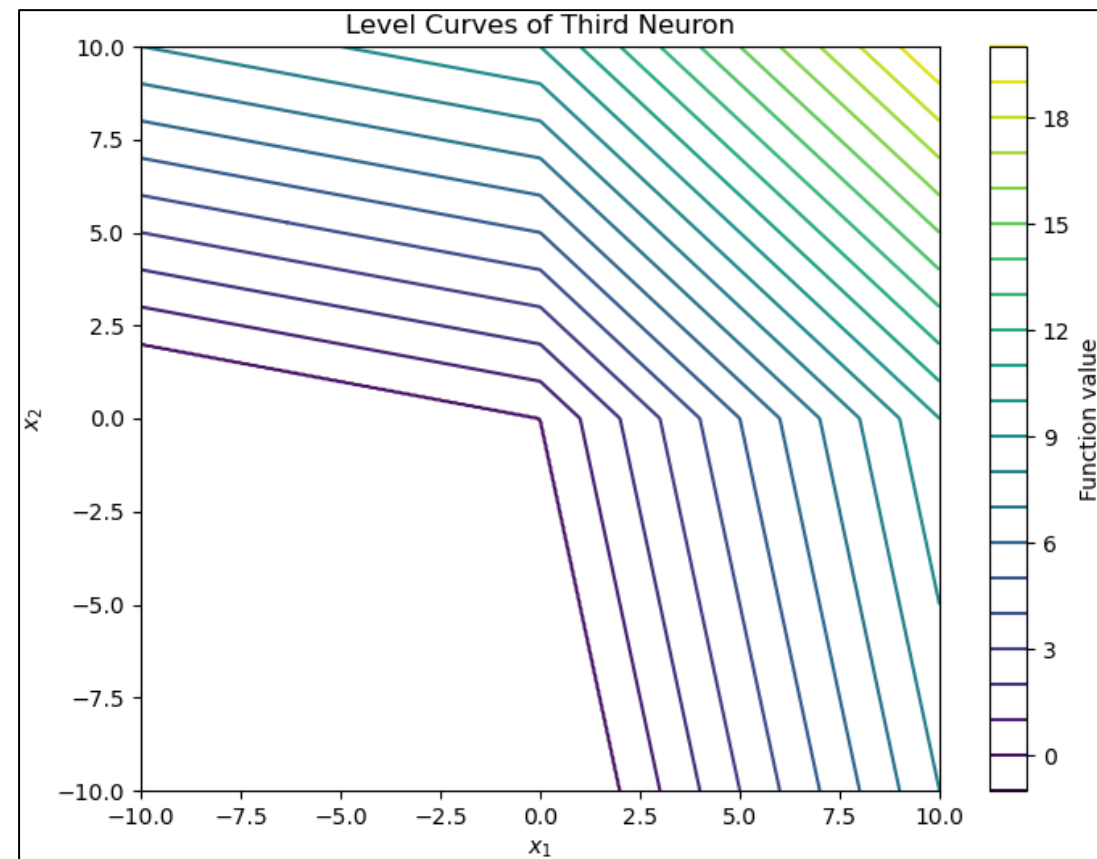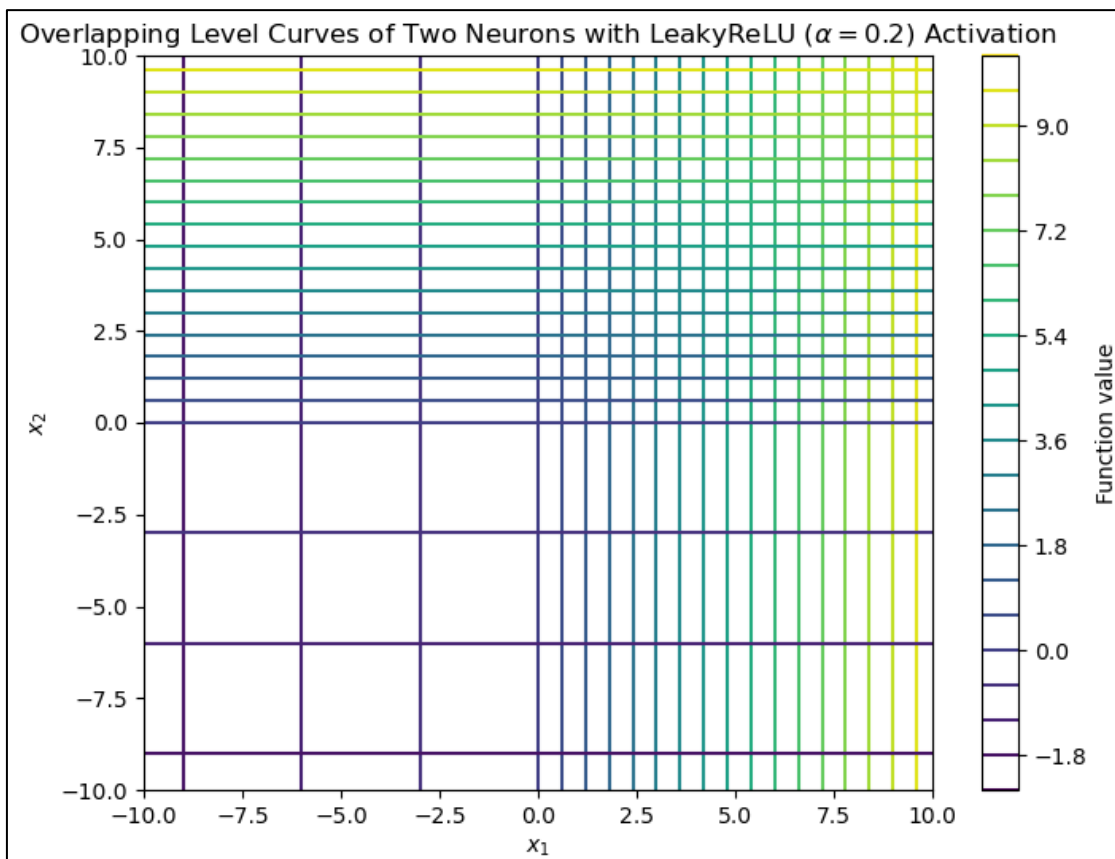 - **Cons:** Requires hyperparameter tuning α.

Sigmoid Decision Boundary

Hyperbolic Tangent Decision Boundary

Overlapping Level Curves of Two Neurons with TanH Activation

Level Curves of Third Neuron

ReLU Decision Boundary

LeakyReLU Decision Boundary

Neural Networks as Universal Approximators

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

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

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 $y^\mathsf{T} x$ 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:

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

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

- One remarkable property of neural networks that was proven in the late 1980s was that NNs are "**Universal Approximators**".

- This means that no matter what type of function you are trying to approximate (which is what the entire goal of training a machine learning model is), as you add more neurons and hidden layers in a neural network (i.e., as the complexity of the network approaches infinity), the network will essentially learn the function that it is trying to approximate perfectly.

- This is one of the reasons that NNs are one of the most promising directions toward AGI (artificial general intelligence).

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE

University of California, San Diego

Abstract—This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

1. INTRODUCTION

It has been nearly twenty years since Minsky and Papert (1969) conclusively demonstrated that the simple two-layer perceptron is incapable of usefully representing or approximating functions outside a very narrow and special class. Although Minsky and Papert left open the possibility that multilayer networks might be capable of better performance, it has only been in the last several years that researchers have begun to explore the ability of multilayer feedforward networks to approximate general mappings from one finite dimensional space to another. Recently, this research has virtually exploded with impressive successes across a wide variety of applications. The scope of these applications is too broad to mention useful specifics here; the interested reader is referred to the proceedings of recent IEEE Conferences on Neural Networks (1987, 1988) for a sampling of examples.

The apparent ability of sufficiently elaborate feedforward networks to approximate quite well nearly any function encountered in applications leads one to wonder about the ultimate capabilities of such networks. Are the successes observed to date reflective of some deep and fundamental approximation capability, or are they merely flukes, resulting from selective reporting and a fortuitous choice of problems? Are multilayer feedforward networks in fact inherently limited to approximating only some fairly special class of functions, albeit a class somewhat larger than the lowly perceptron? The purpose of this paper is to address these issues. We show that multilayer feedforward networks with as few as one hidden layer are indeed capable of universal approximation in a very precise and satisfactory sense.

Advocates of the virtues of multilayer feedforward networks (e.g., Hecht-Nielsen, 1987) often cite Kolmogorov's (1957) superposition theorem or its more recent improvements (e.g., Lorentz, 1976) in support of their capabilities. However, these results require a *different* unknown transformation (g in Lorentz's notation) for each continuous function to be represented, while specifying an exact upper limit to the number of intermediate units needed for the representation. In contrast, quite specific squashing functions (e.g., logistic, hyperbolic tangent) are used in practice, with necessarily little regard for the function being approximated and with the number of hidden units increased *ad libitum* until some desired level of approximation accuracy is reached. Al-