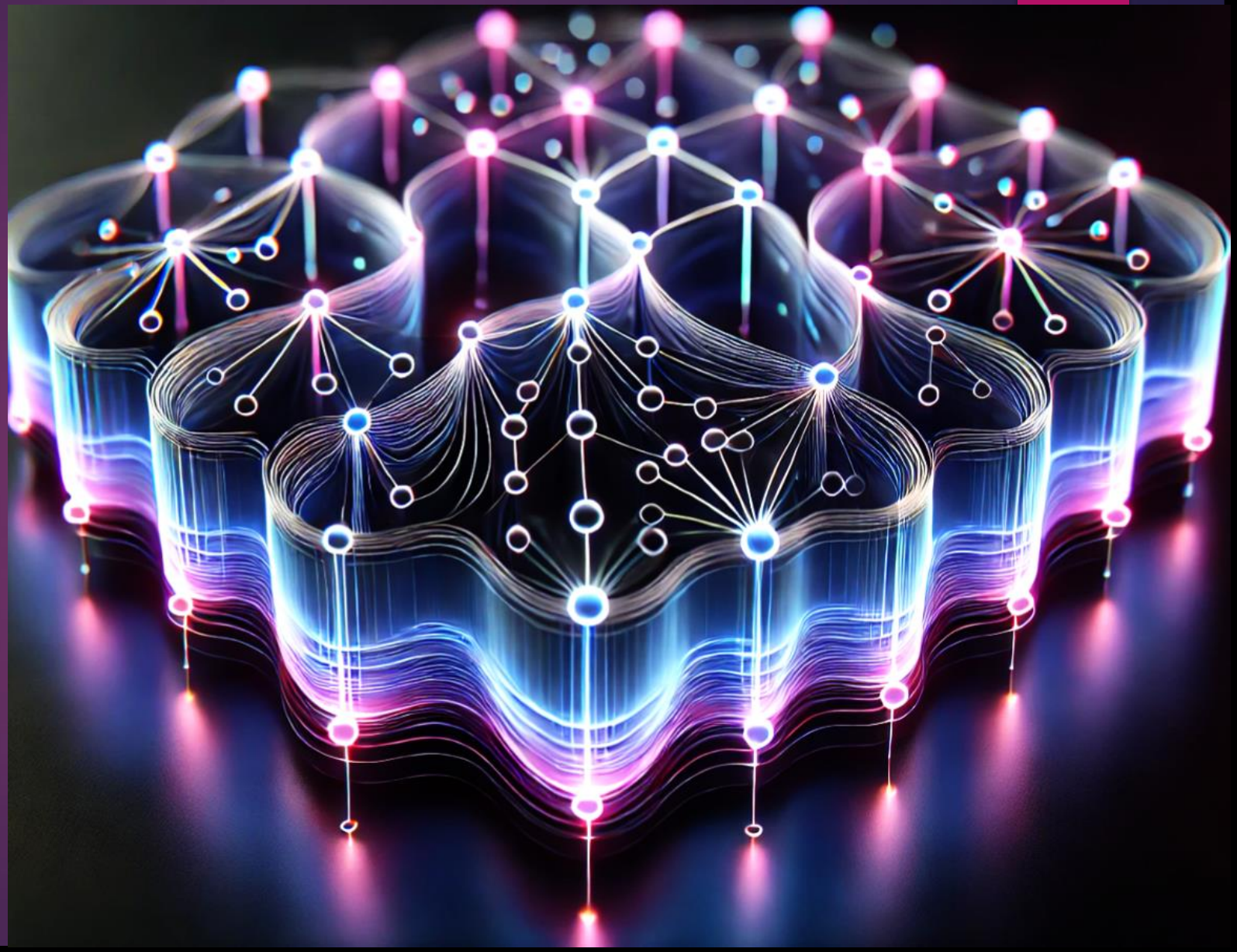
INTRODUCTION TO DEEP LEARNING

- ISE – 364 / 464
- DEPT. OF INDUSTRIAL & SYSTEMS ENGINEERING
- GRIFFIN DEAN KENT

LEHIGH UNIVERSITY.



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

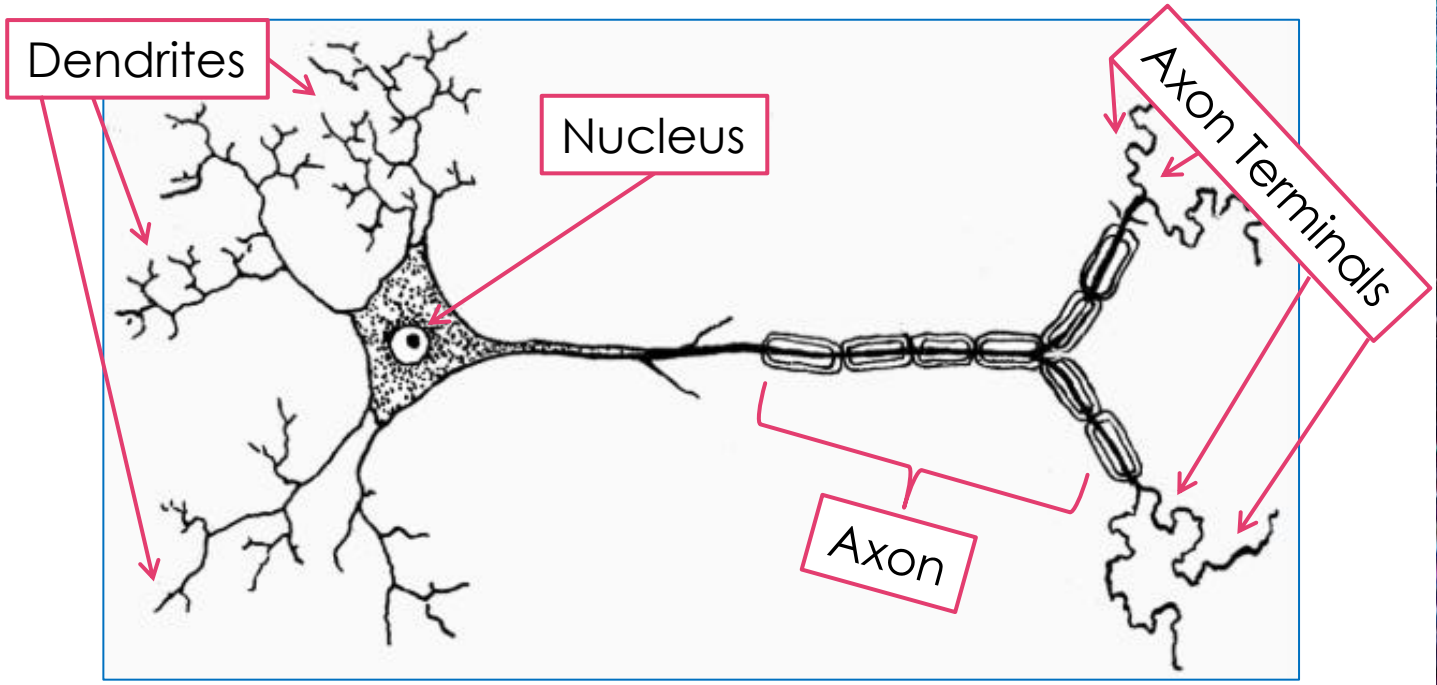
- **Multi-Layered Perceptrons:** The most classical of NNs. Typically used on tabular datasets that consist of feature-label pairs, much like other supervised learning models.
- **Convolutional Networks:** A spatial-oriented architecture that utilizes convolutions and kernels to obtain "feature maps" 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).

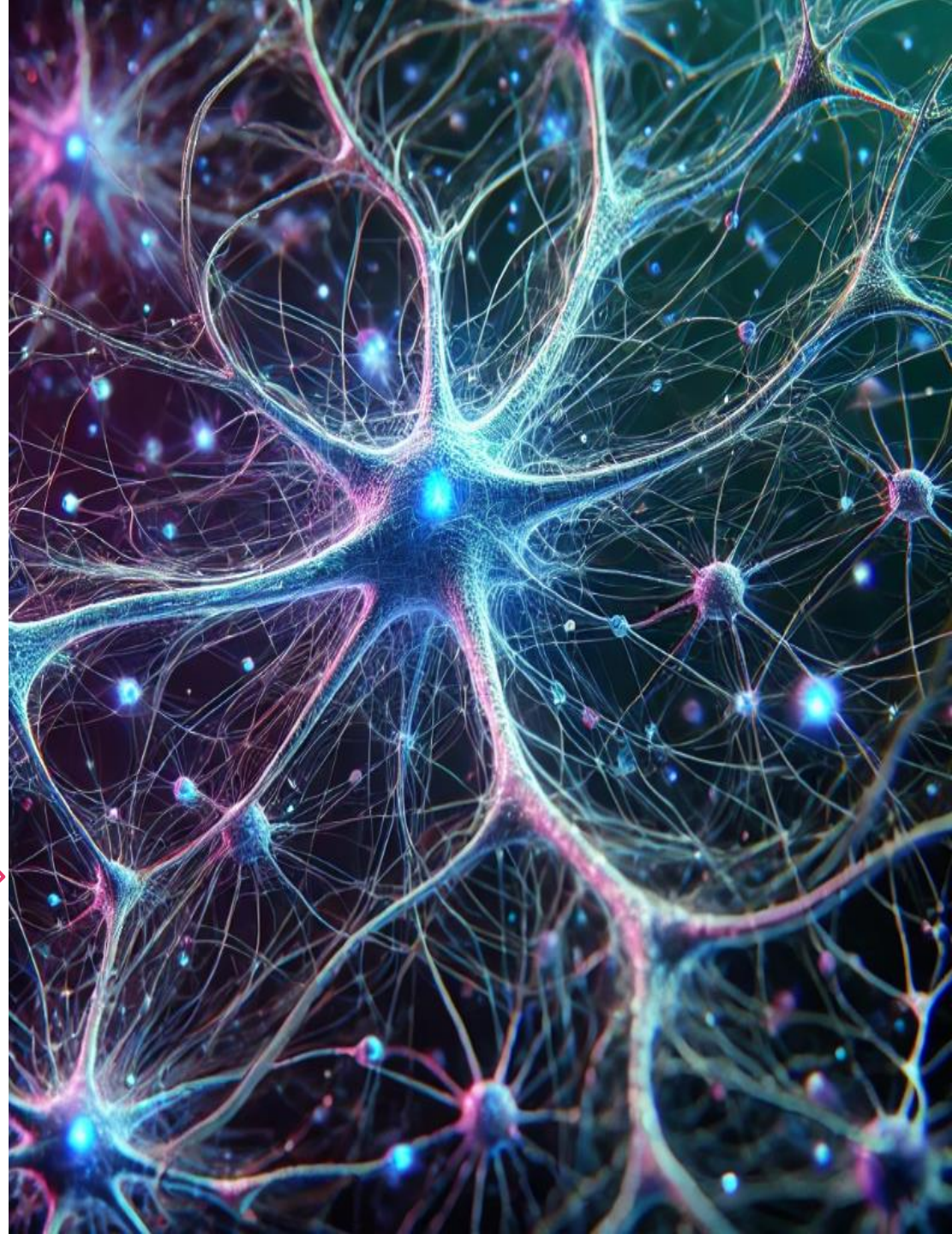
Some Reflections on Biology

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.





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

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

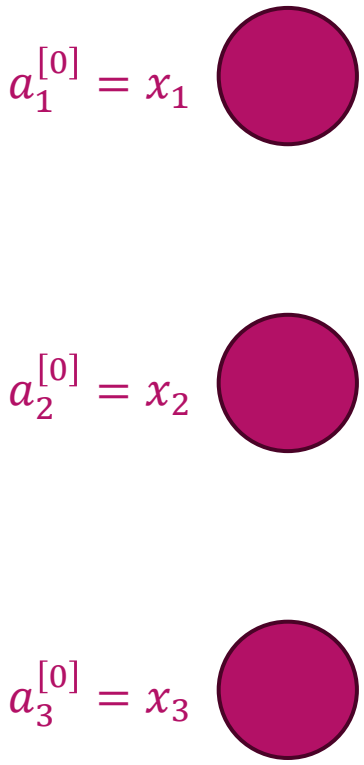
where $\theta := [w^T, b]^T$.

- 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.

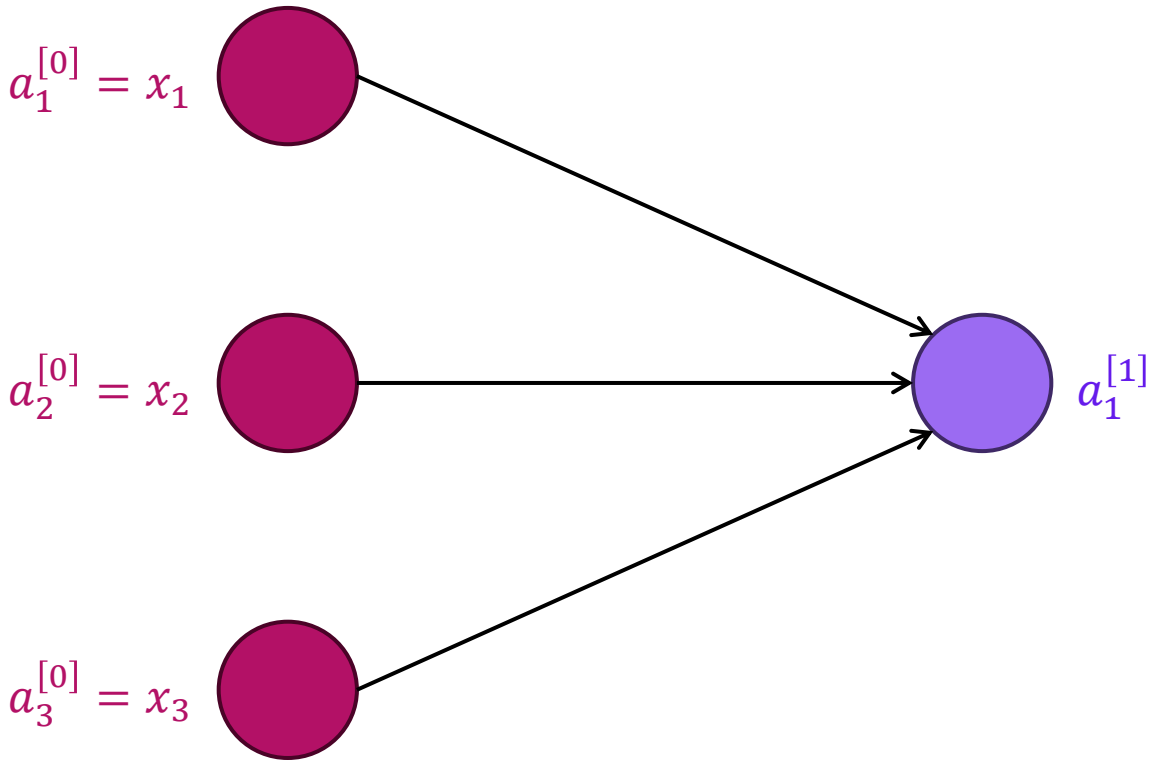
A Connection to Linear & Logistic Regression

- 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**).

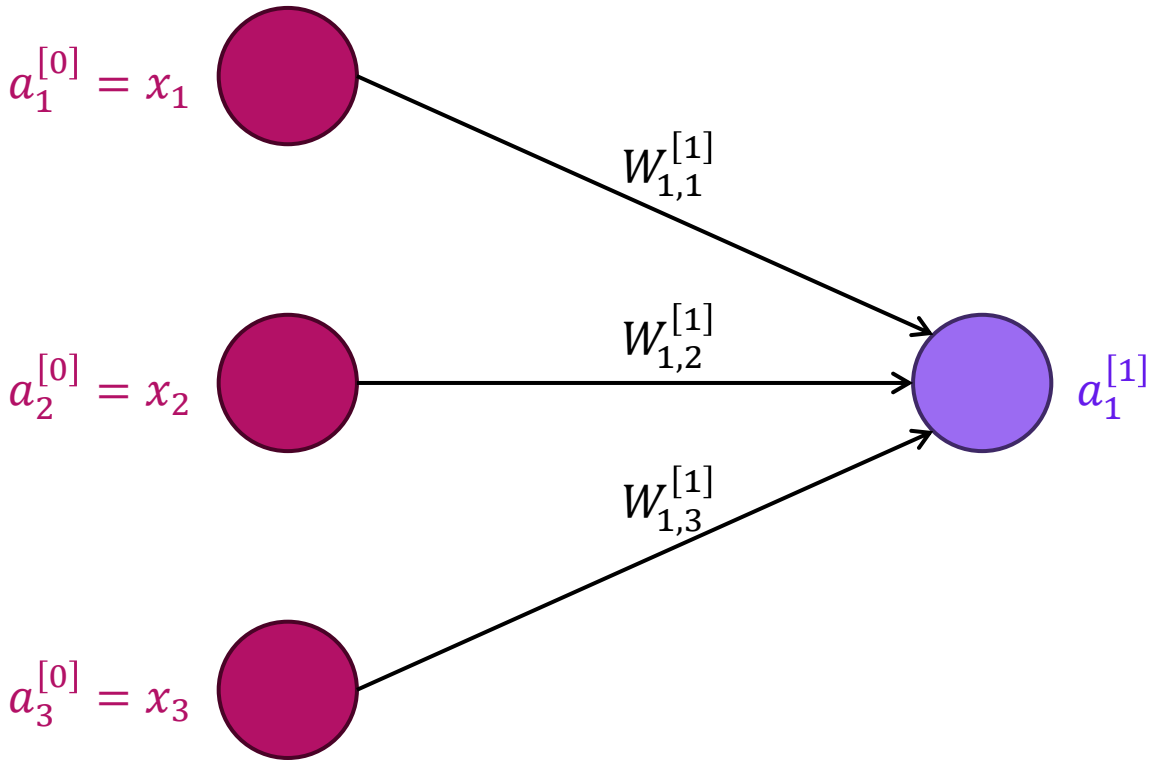
The Perceptron (Illustration)



The Perceptron (Illustration)

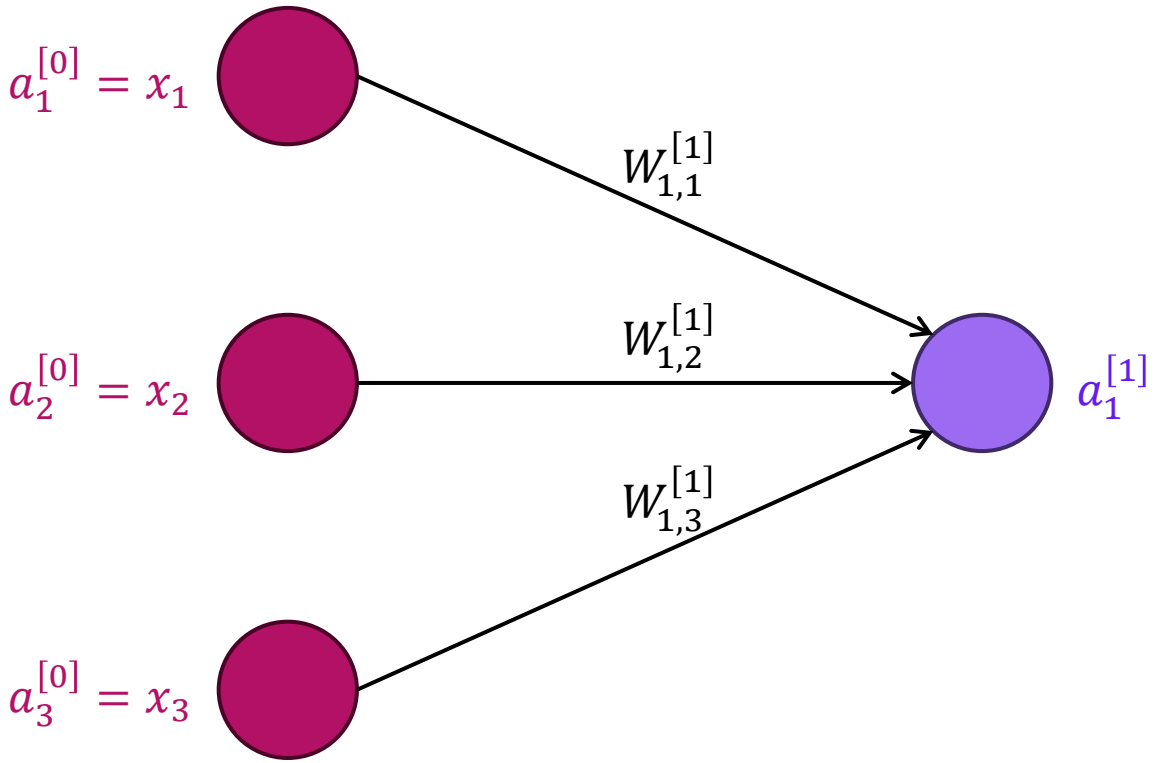


The Perceptron (Illustration)



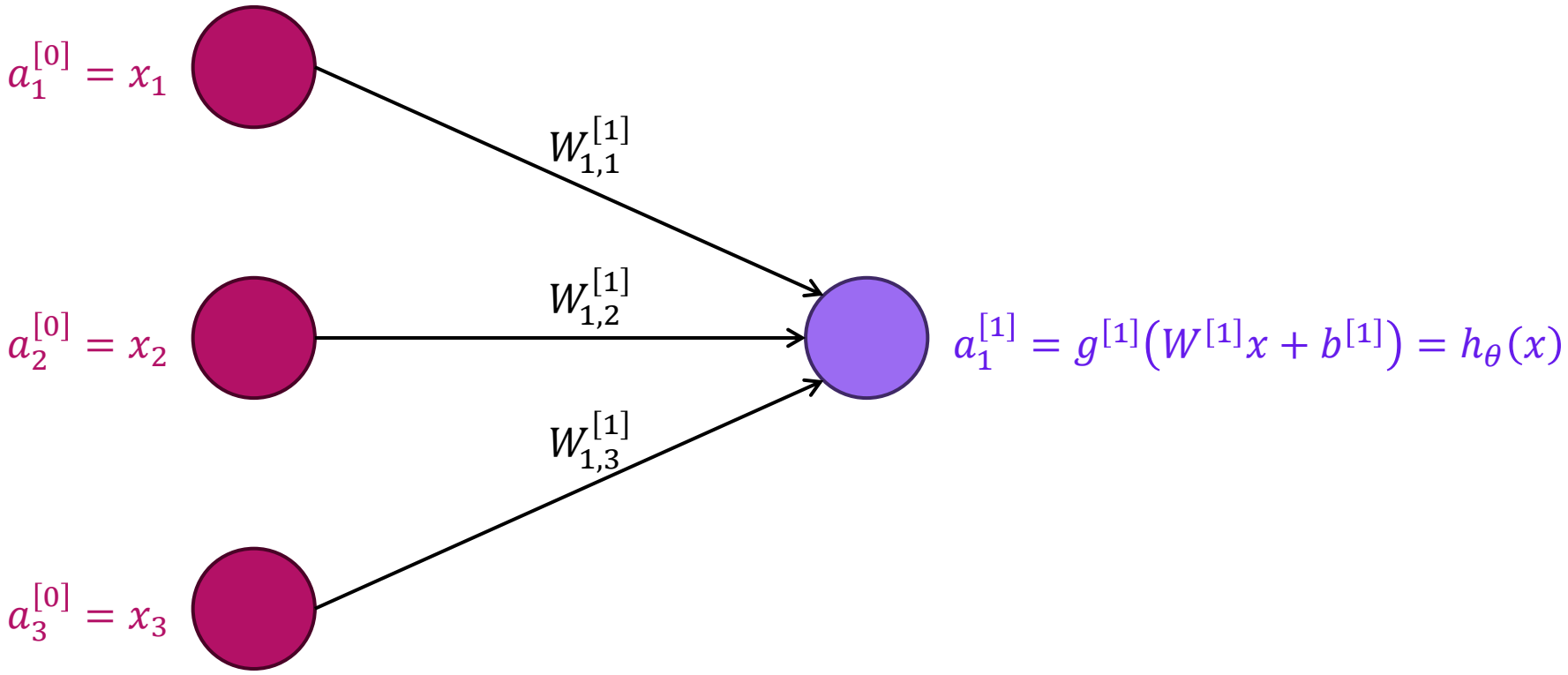
The Perceptron (Illustration)

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



The Perceptron (Illustration)

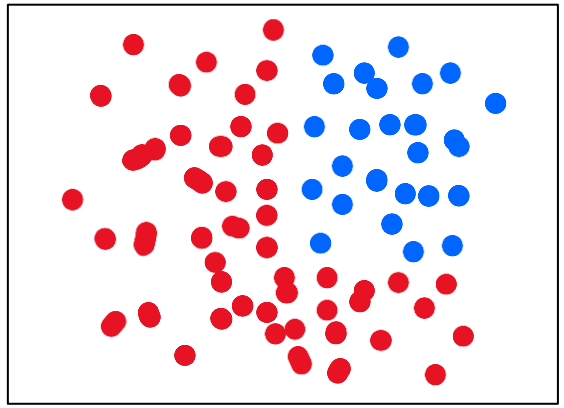
$$W^{[1]} = \begin{bmatrix} W_{1,1}^{[1]} & W_{1,2}^{[1]} & W_{1,3}^{[1]} \end{bmatrix} \in \mathbb{R}^{1\times 3} \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \end{bmatrix} \in \mathbb{R}^{1\times 1}$$



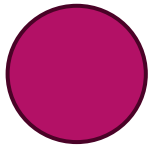
Visualizing the Decision Boundary

$a_1^{[0]} = x_1$

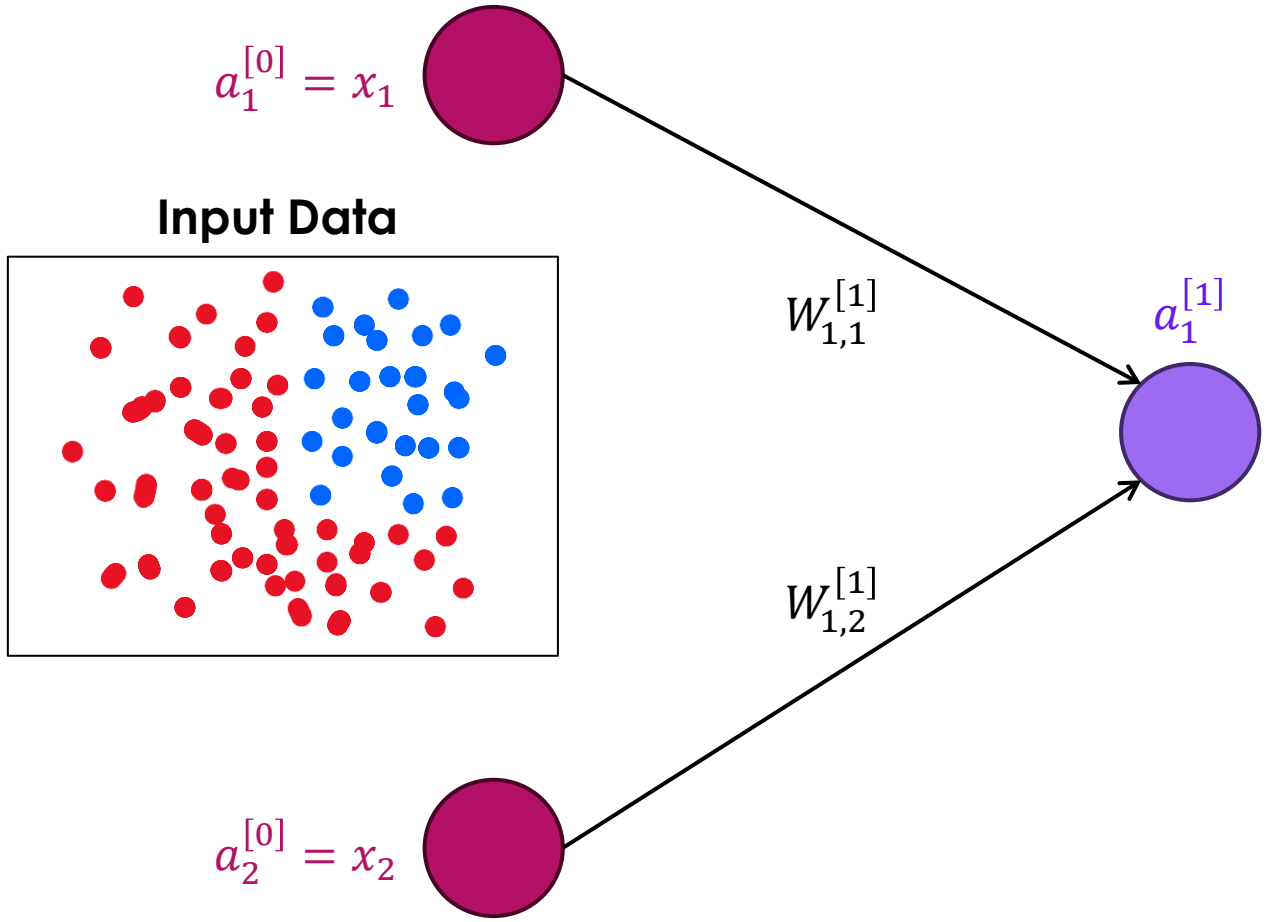
Input Data



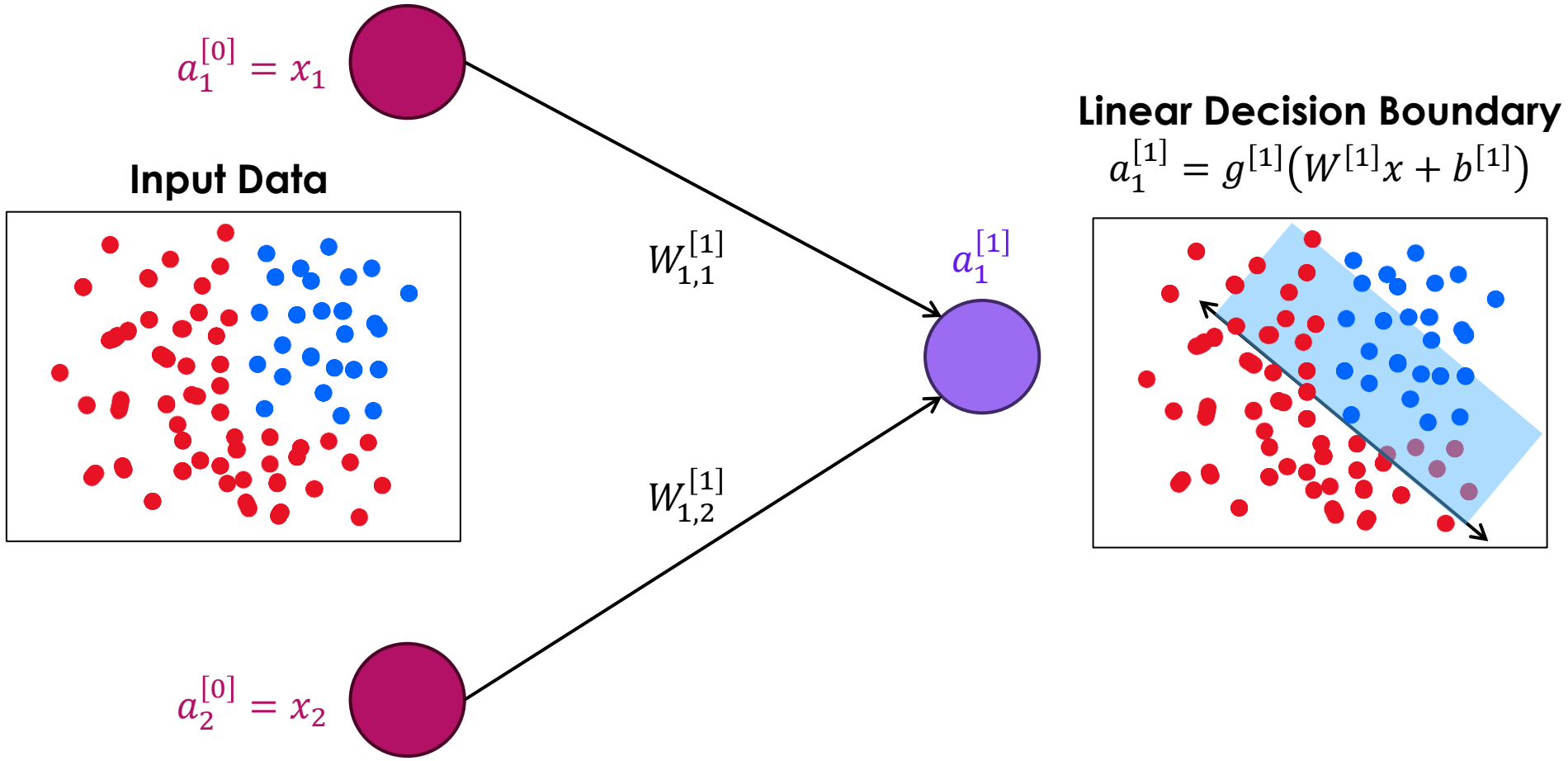
$a_2^{[0]} = x_2$



Visualizing the Decision Boundary



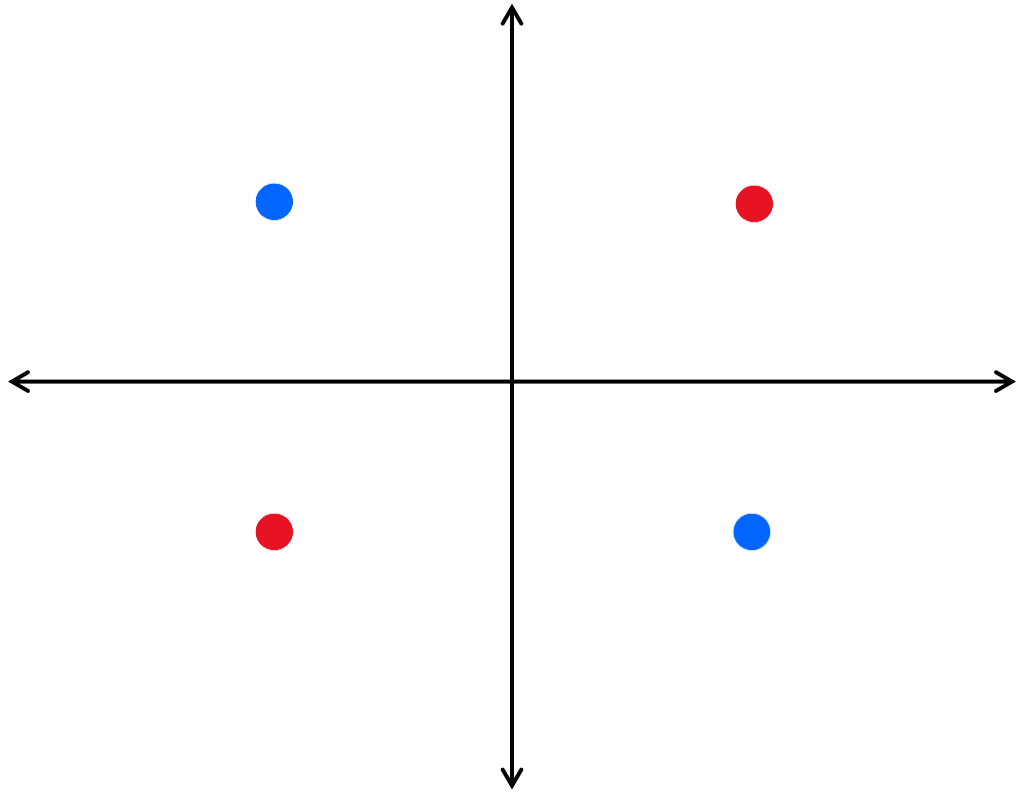
Visualizing the Decision Boundary



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.

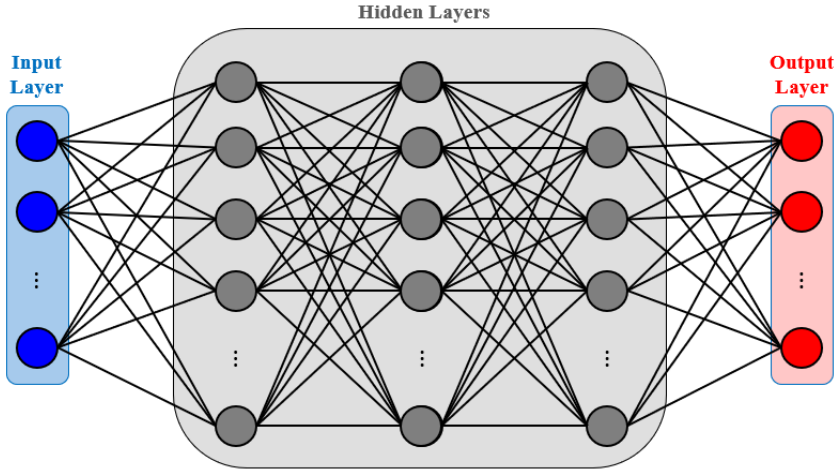
Multi-Layered Perceptron (MLP)

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

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

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 $h_\theta(x) := \mathcal{N}(x; g, \theta)$.

The Multi-Layered Perceptron (A Classical Neural Network)

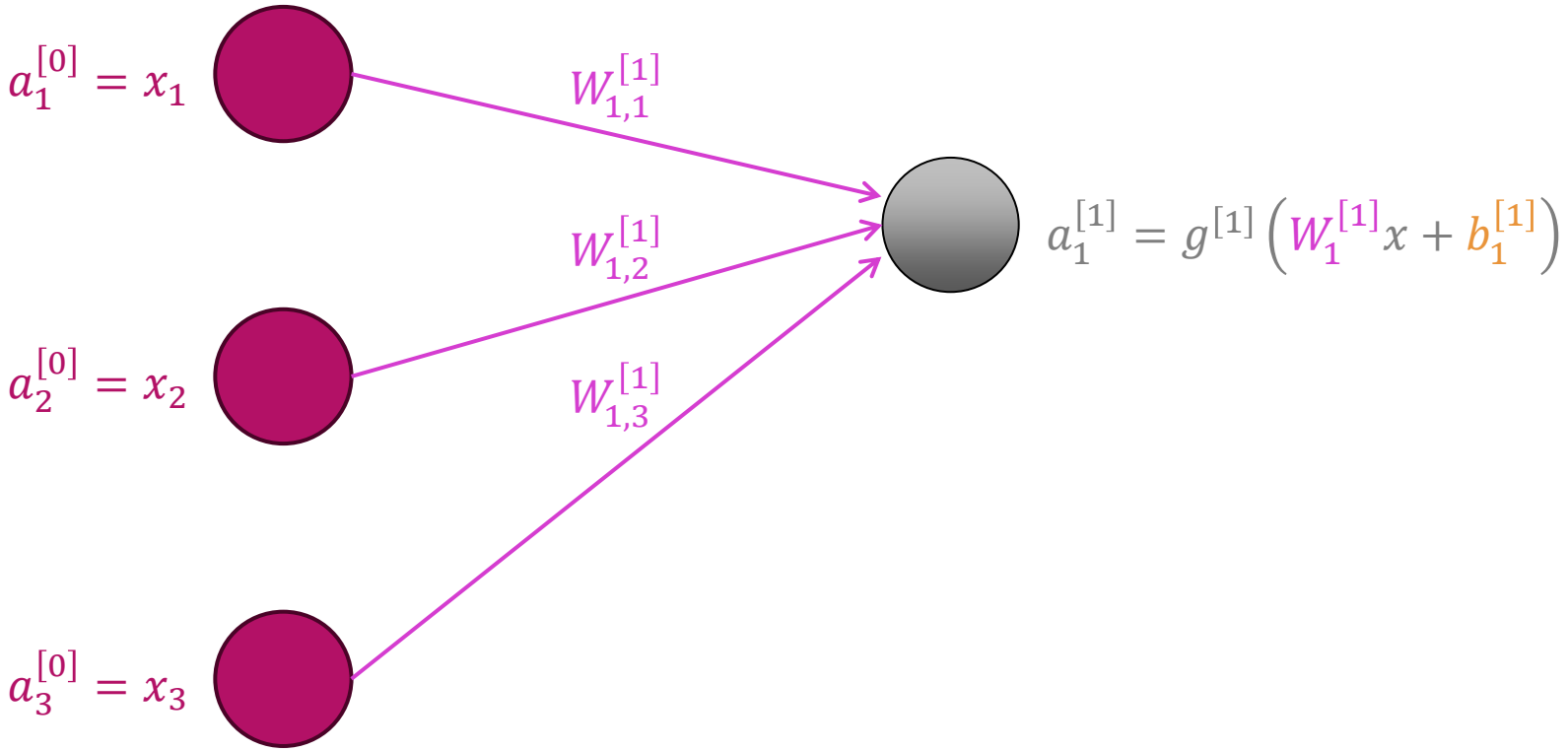


Mathematical Notation

- $\boldsymbol{L \in \mathbb{N}}$: 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 $\ell = 0$.
- $\boldsymbol{n[\ell] \in \mathbb{N}}$: The **number of neurons in the ℓ-th layer** of the network.
- $\boldsymbol{g^{[\ell]}: \mathbb{R} \to \mathbb{R}}$: 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 $g := \{g^{[\ell]}\}_{\ell=1}^{L}$.
- $\boldsymbol{z^{[\ell]} \in \mathbb{R}^{n[\ell]}}$: The vector of entries that will serve as **inputs into the activation function** $g^{[\ell]}$ to obtain the output vector $a^{[\ell]}$.
- $\boldsymbol{a^{[\ell]} \in \mathbb{R}^{n[\ell]}}$: The **vector of outputs from the ℓ-th layer**. Hence, each $a_j^{[\ell]}$, for all $j \in \{1,2,\dots,n[\ell]\}$, 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 $a^{[\ell](i)}$.
- $\boldsymbol{W_j^{[\ell]} \in \mathbb{R}^{n[\ell-1]}}$: The **vector of input weights for the j-th neuron of the ℓ-th layer**. One can stack all $j \in \{1,2,\dots,n[\ell]\}$ 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 $W := \{W^{[\ell]}\}_{\ell=1}^{L}$.
- $\boldsymbol{b_j^{[\ell]} \in \mathbb{R}}$: 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 $b := \{b^{[\ell]}\}_{\ell=1}^{L}$.

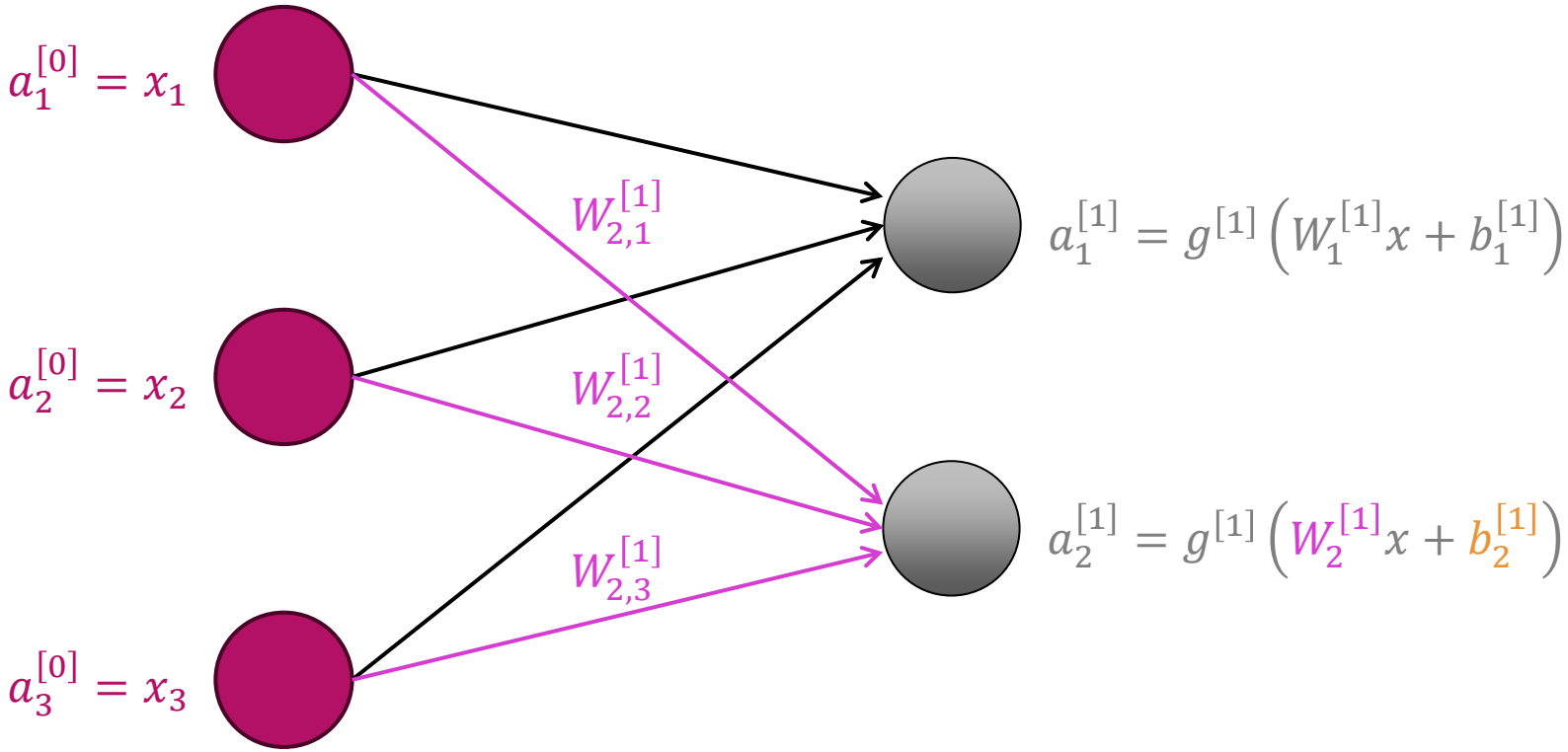
A Multi-Layered Perceptron (Illustration)

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



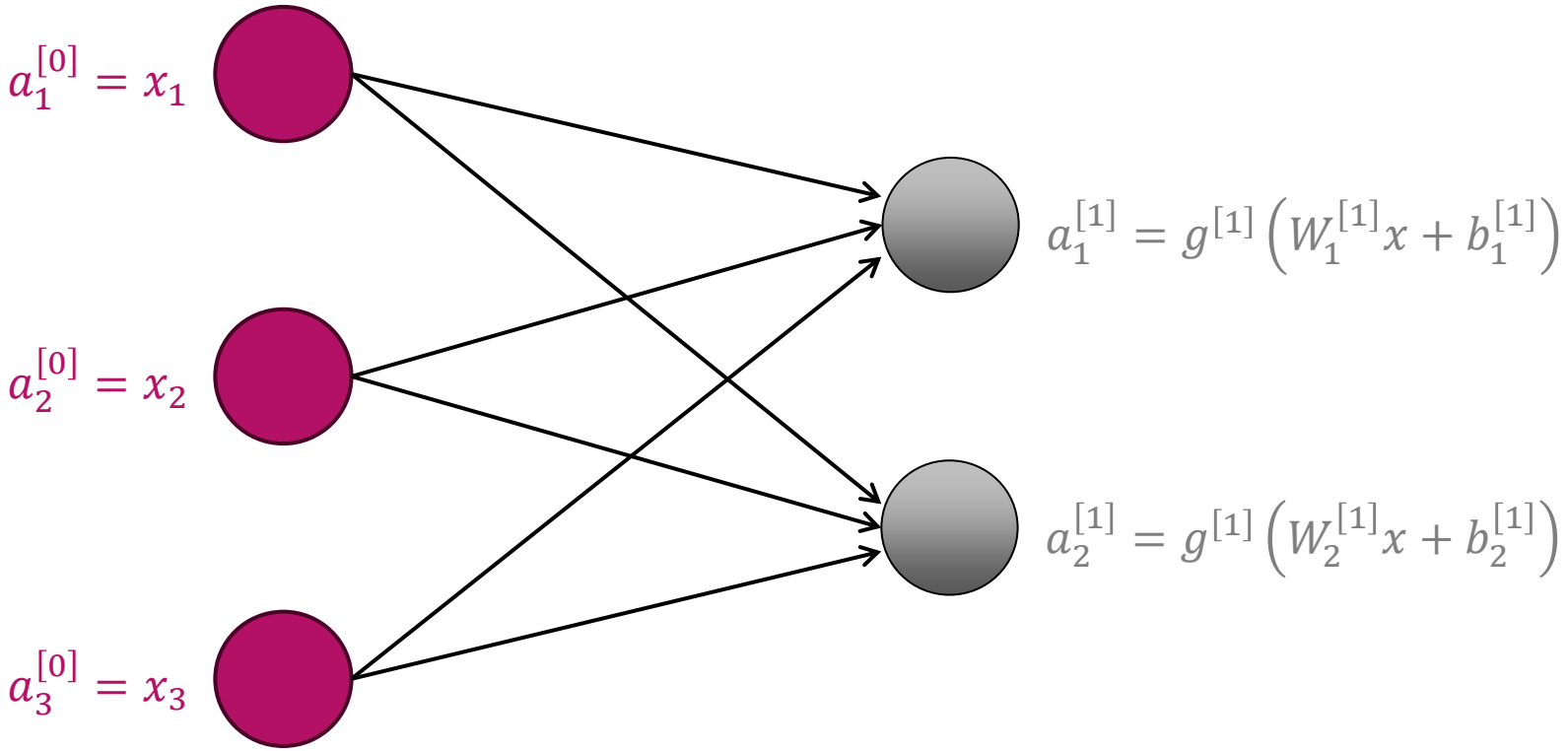
A Multi-Layered Perceptron (Illustration)

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



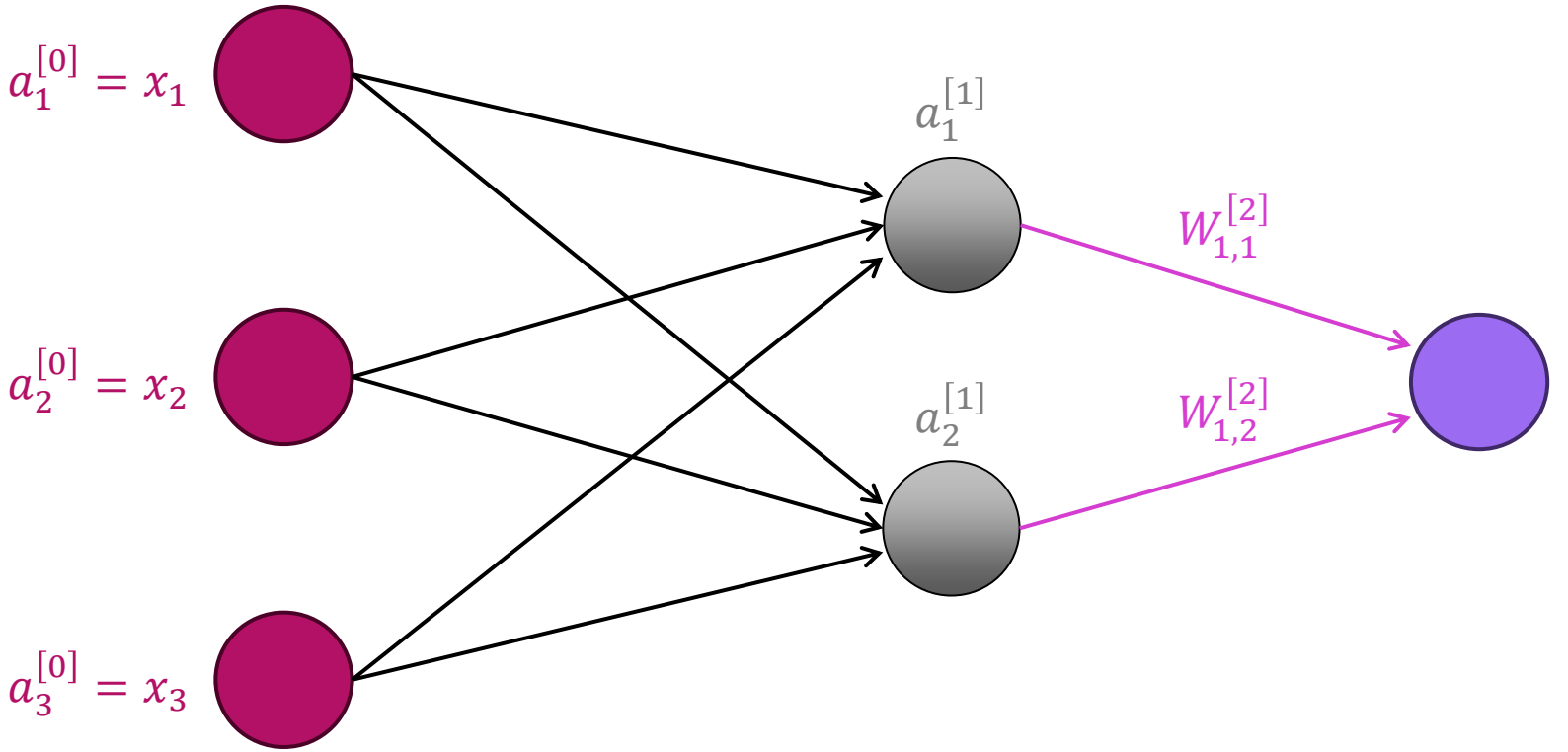
A Multi-Layered Perceptron (Illustration)

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



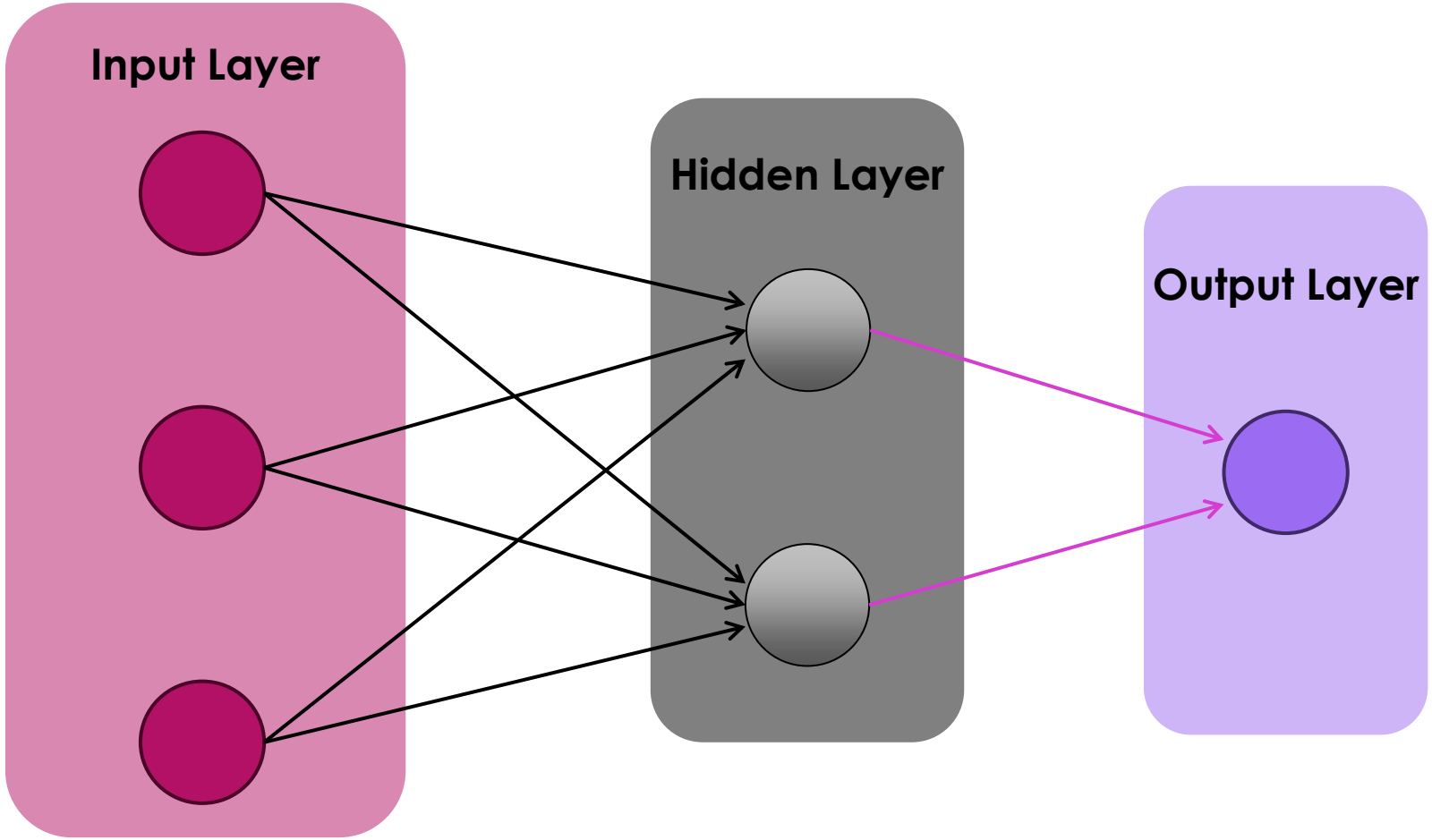
A Multi-Layered Perceptron (Illustration)

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



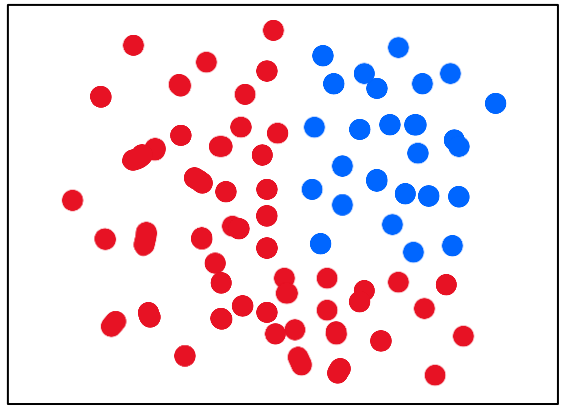
$$a_1^{[2]} = g^{[2]}\left(W_1^{[2]} a^{[1]} + b_1^{[2]}\right) = h_\theta(x)$$

A Multi-Layered Perceptron (Illustration)



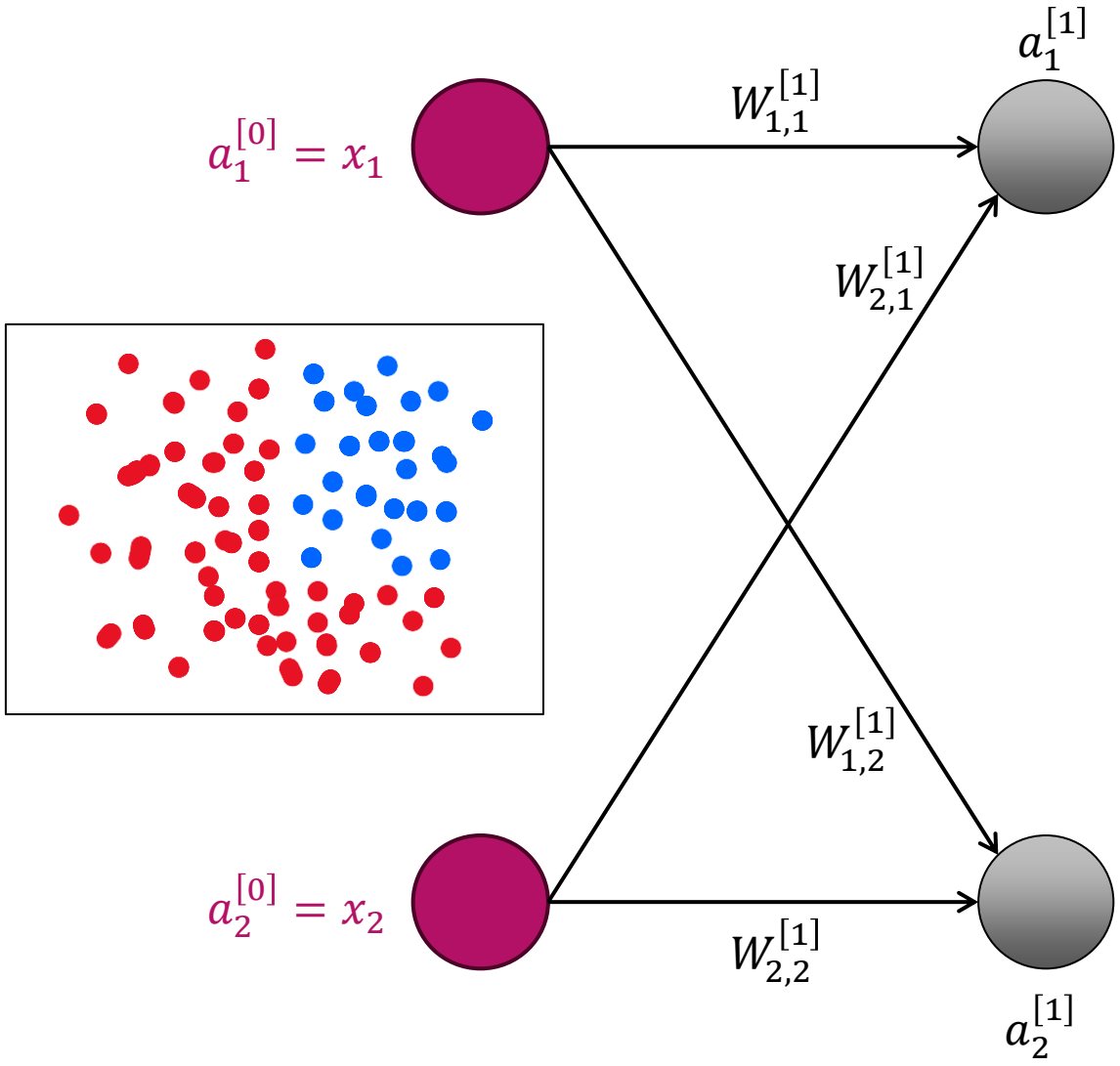
Visualizing the Decision Boundary

$a_1^{[0]} = x_1$

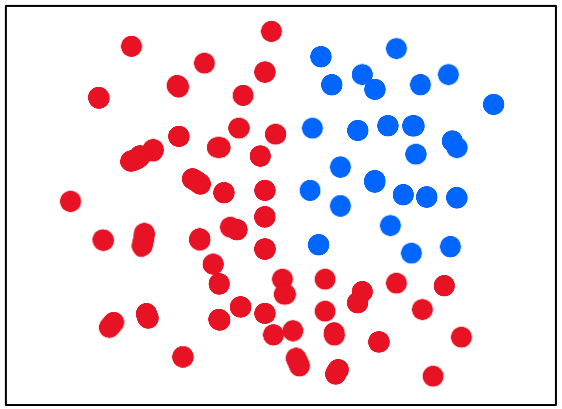


$a_2^{[0]} = x_2$

Visualizing the Decision Boundary



Visualizing the Decision Boundary



$a_1^{[0]} = x_1$

$a_2^{[0]} = x_2$

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

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

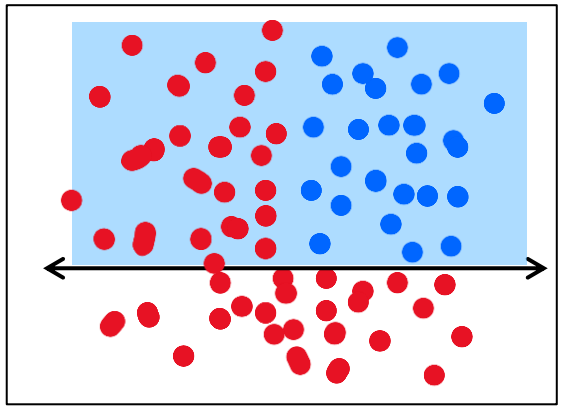
$W_{1,2}^{[1]}$

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

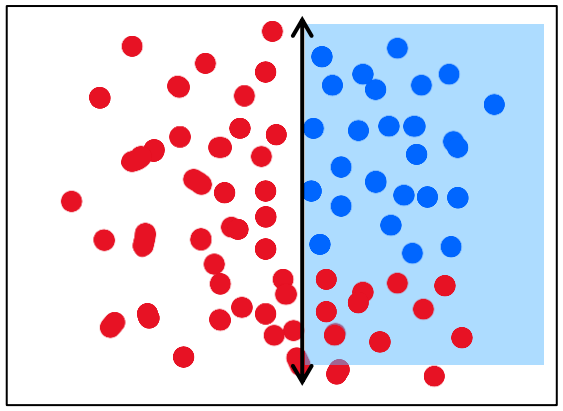
$a_1^{[1]}$

$a_2^{[1]}$

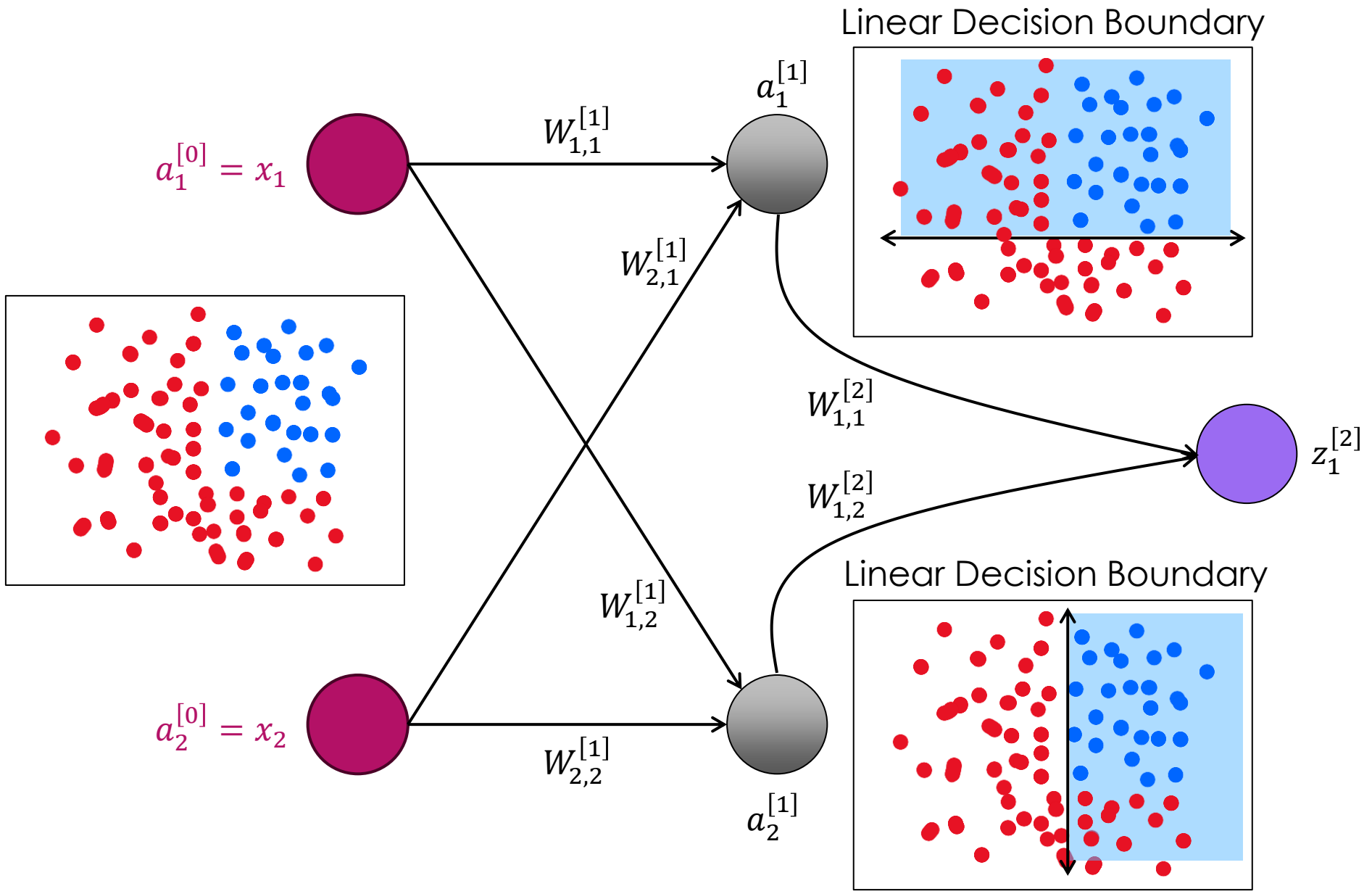
Linear Decision Boundary



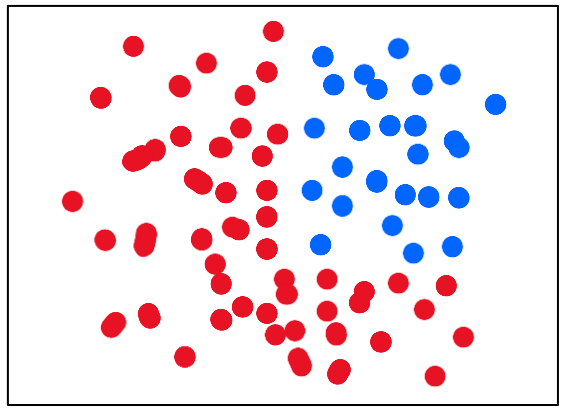
Linear Decision Boundary



Visualizing the Decision Boundary



Visualizing the Decision Boundary



$a_1^{[0]} = x_1$

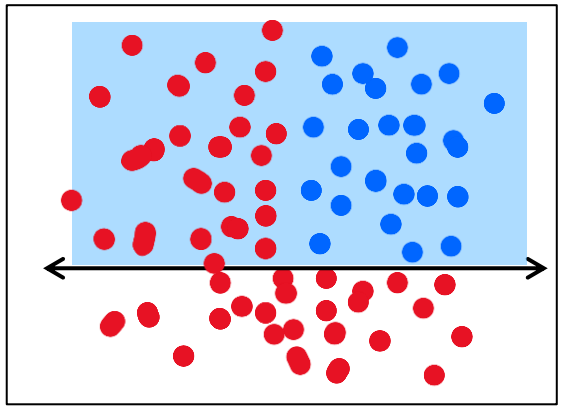
$a_2^{[0]} = x_2$

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

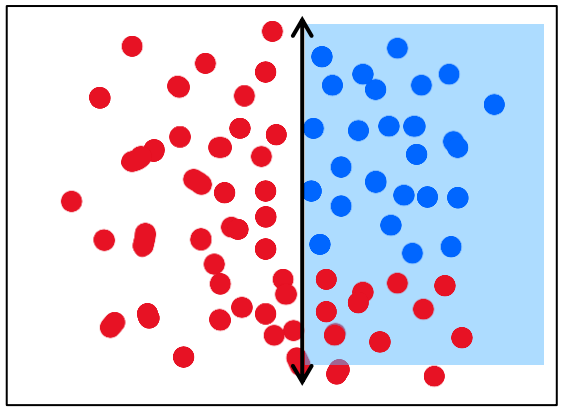
$a_1^{[1]}$ $a_2^{[1]}$

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

Linear Decision Boundary

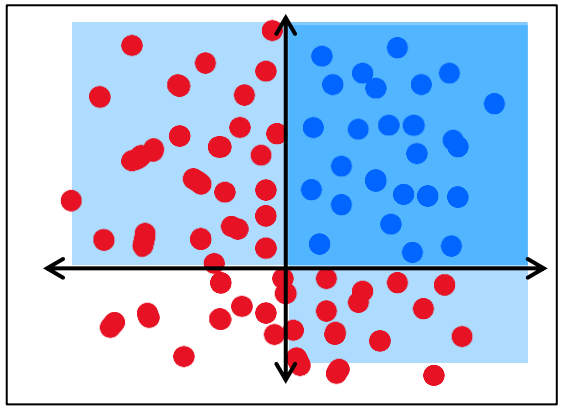


Linear Decision Boundary

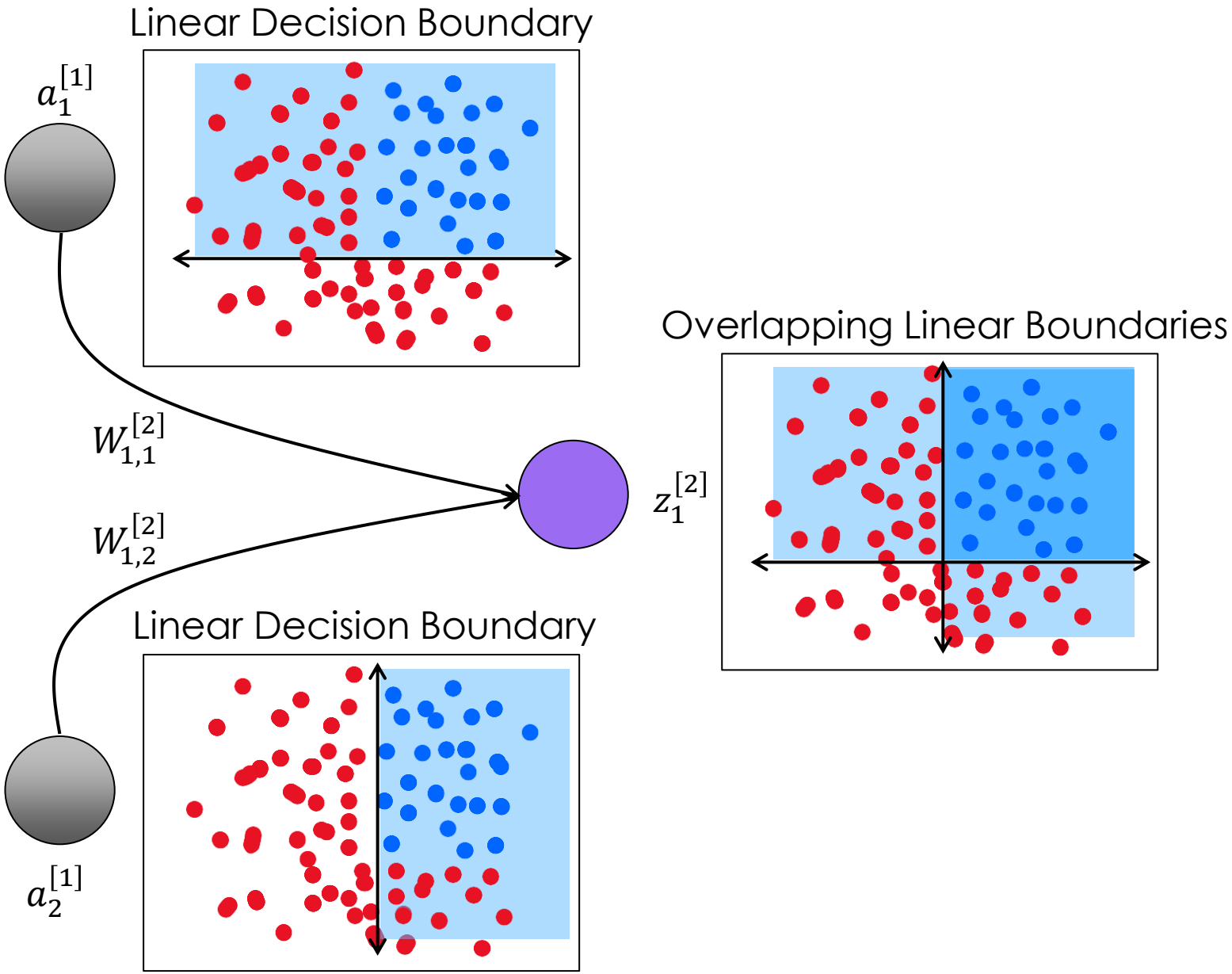


Overlapping Linear Boundaries

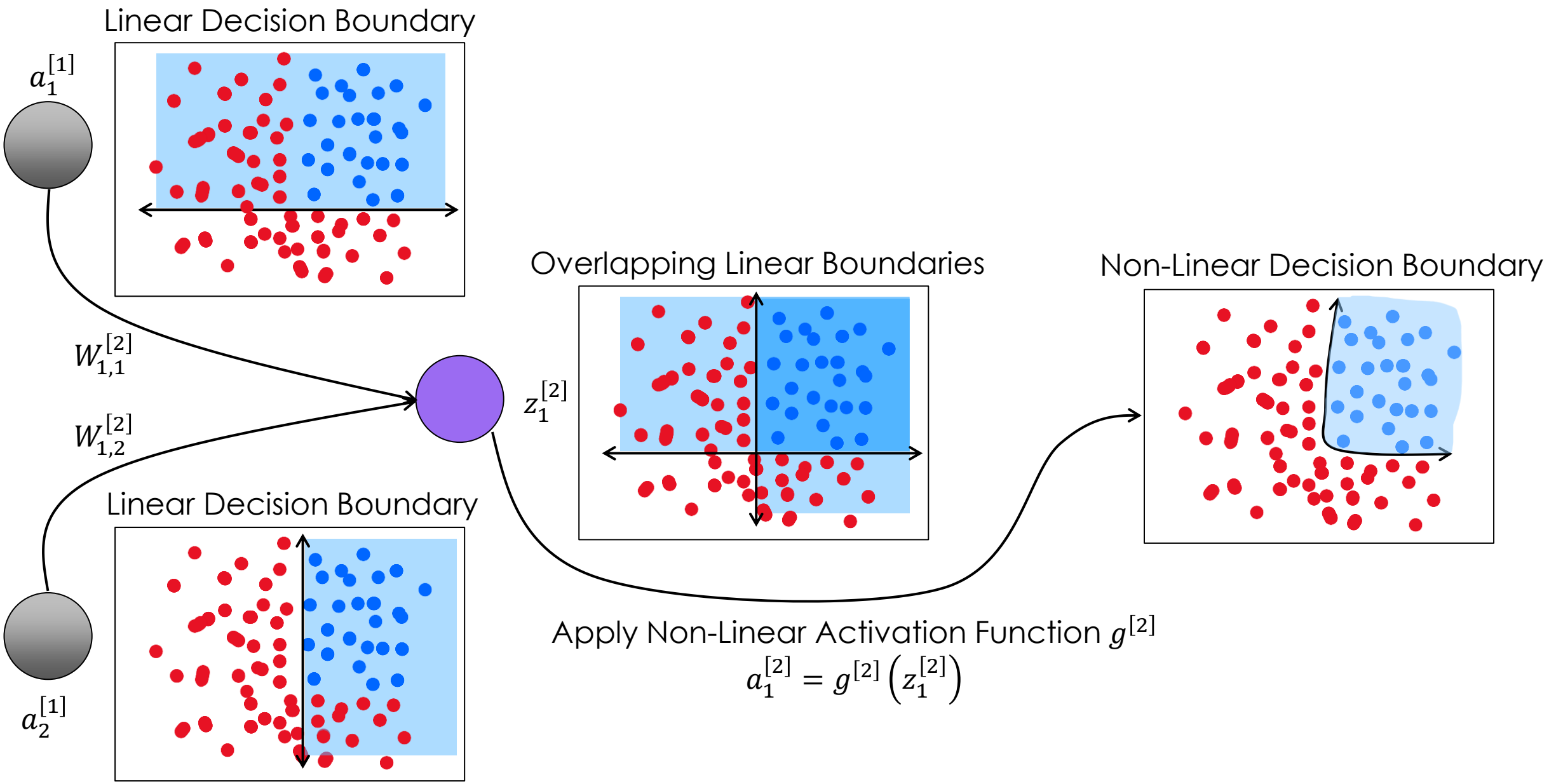
$z_1^{[2]}$



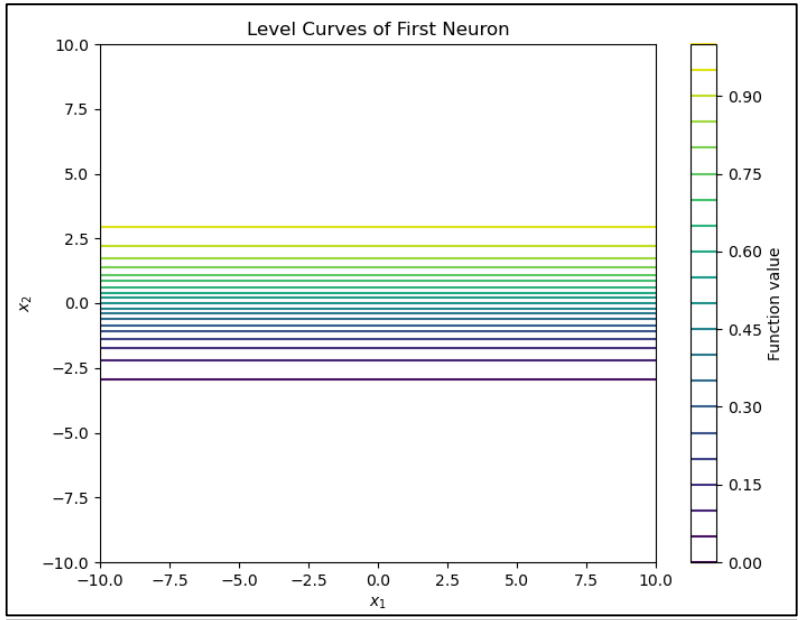
Visualizing the Decision Boundary

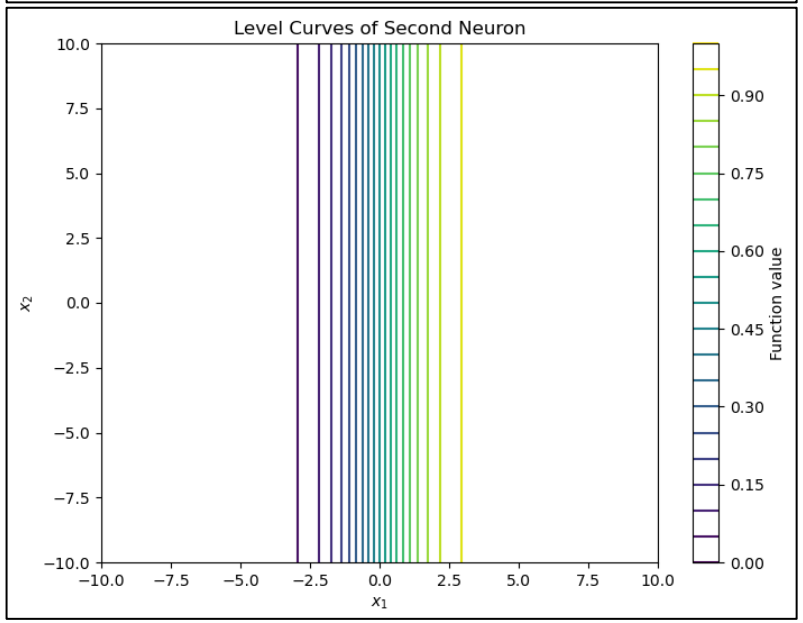


Visualizing the Decision Boundary

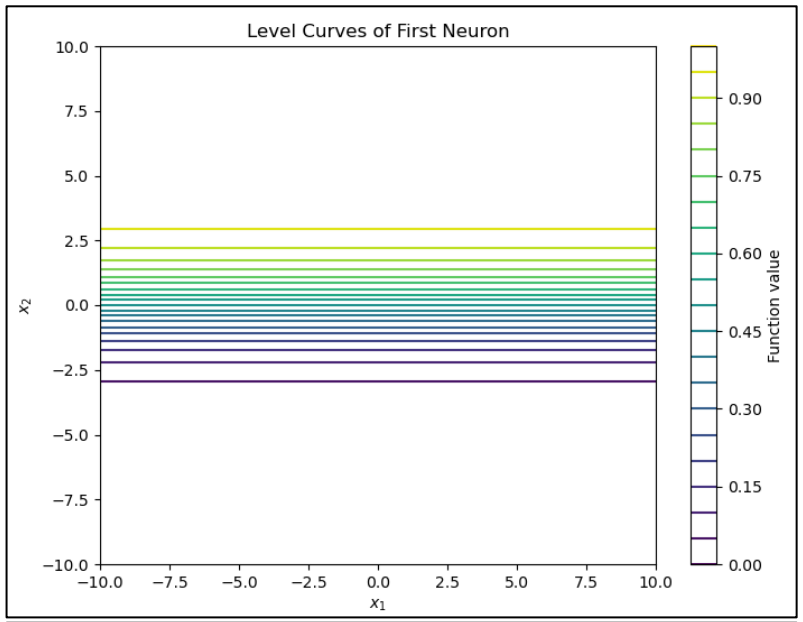


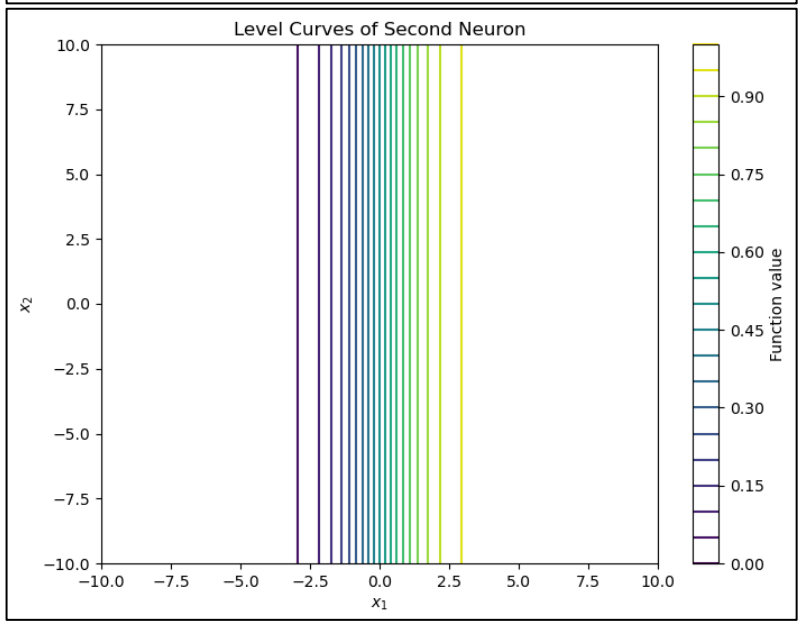
Level-Curves of Neurons

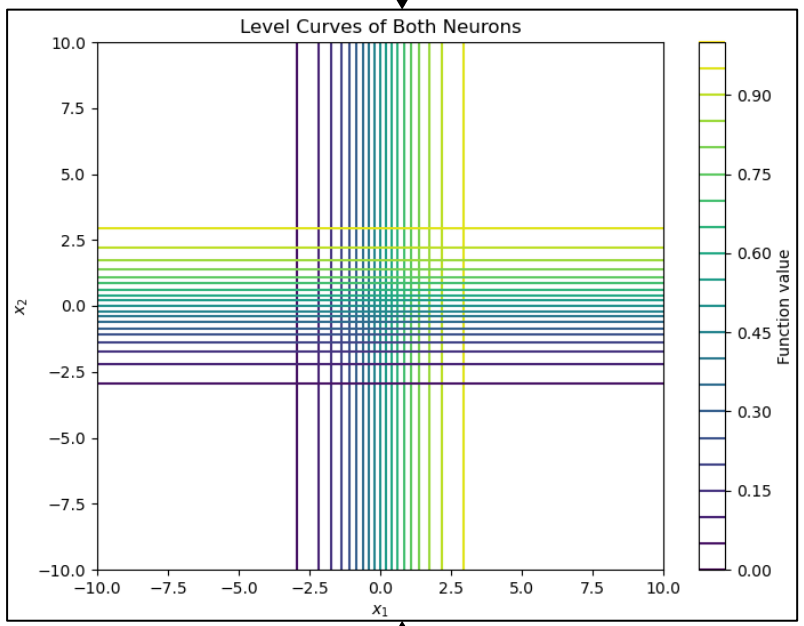




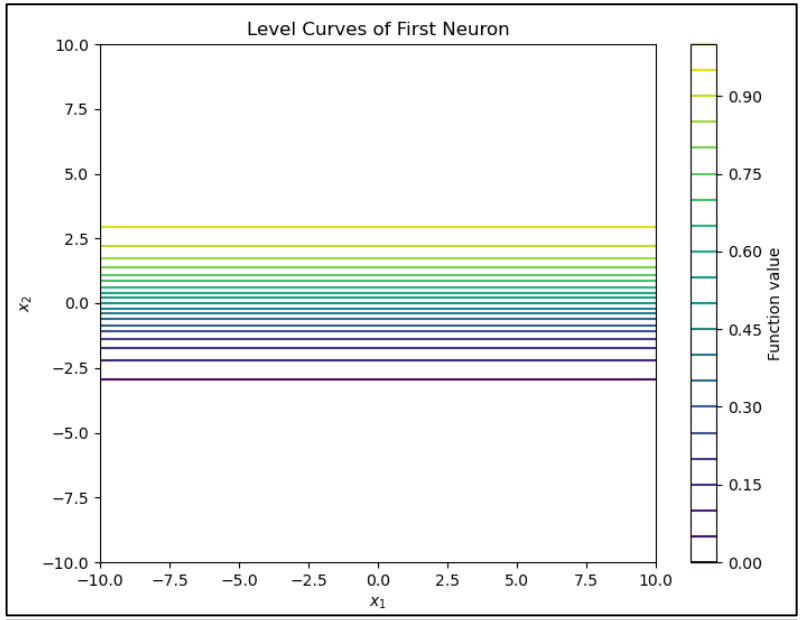
Level-Curves of Neurons

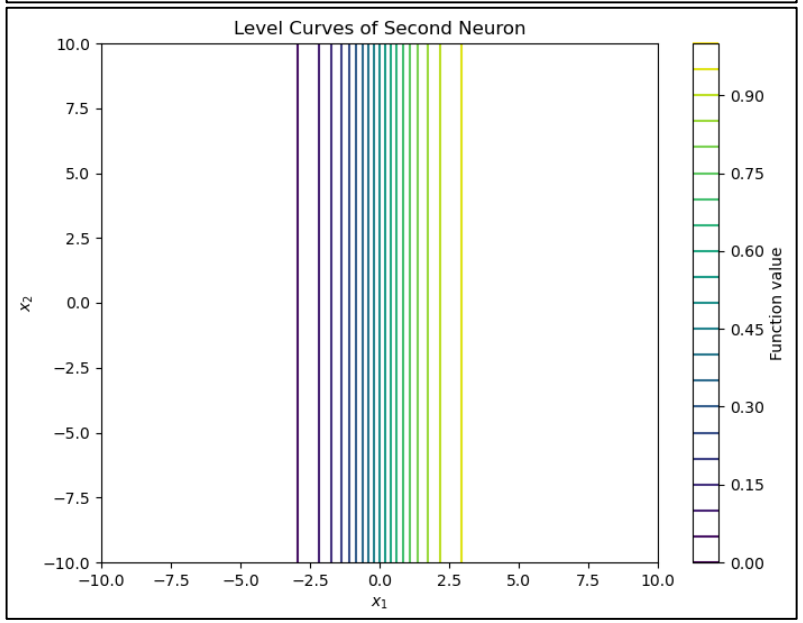


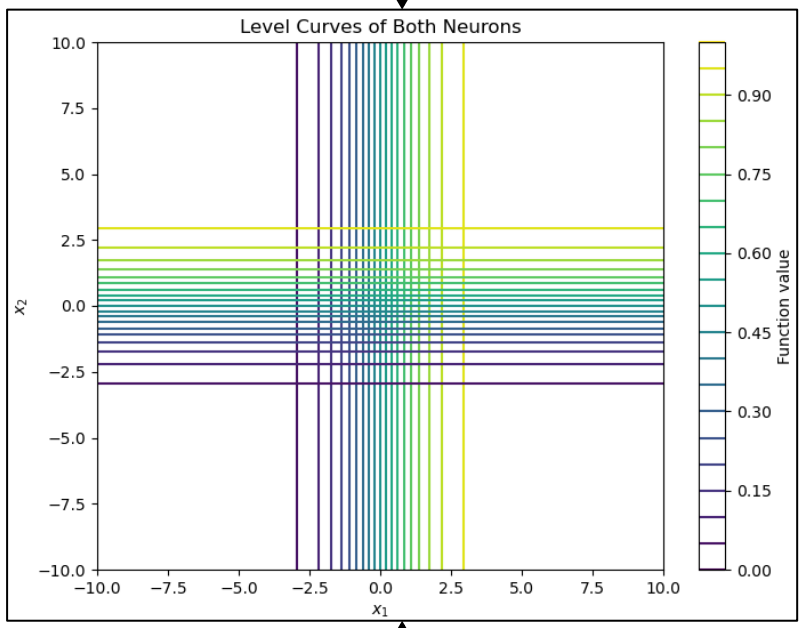


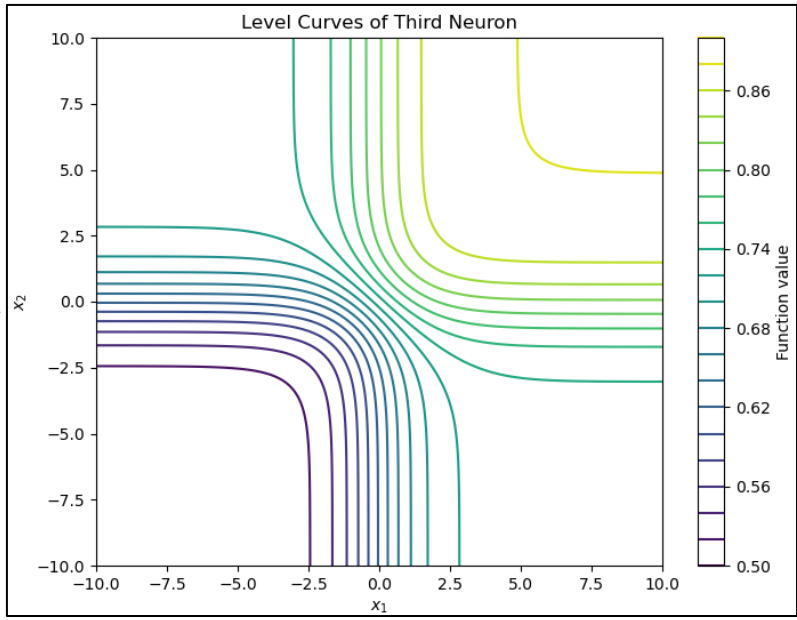


Level-Curves of Neurons









Forward-Pass (Single Datapoint)

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 $a^{[0]} = x^{(i)}$)

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

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

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

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 $A^{[0]} = X^T$)

$$\boldsymbol{A^{[\ell]} = g^{[\ell]}(Z^{[\ell]}) = g^{[\ell]}(W^{[\ell]} A^{[\ell-1]} + b^{[\ell]})},$$

for all $\ell \in \{1, 2, \ldots, L\}$. Here, we define the new matrices $Z^{[\ell]} := \left[z^{[\ell](1)}, z^{[\ell](2)}, \ldots, z^{[\ell](m)}\right] \in \mathbb{R}^{n[\ell] \times m}$ and $A^{[\ell]} := \left[a^{[\ell](1)}, a^{[\ell](2)}, \ldots, a^{[\ell](m)}\right] \in \mathbb{R}^{n[\ell] \times m}$, 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 **entirety of the internet**…) 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

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

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

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

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

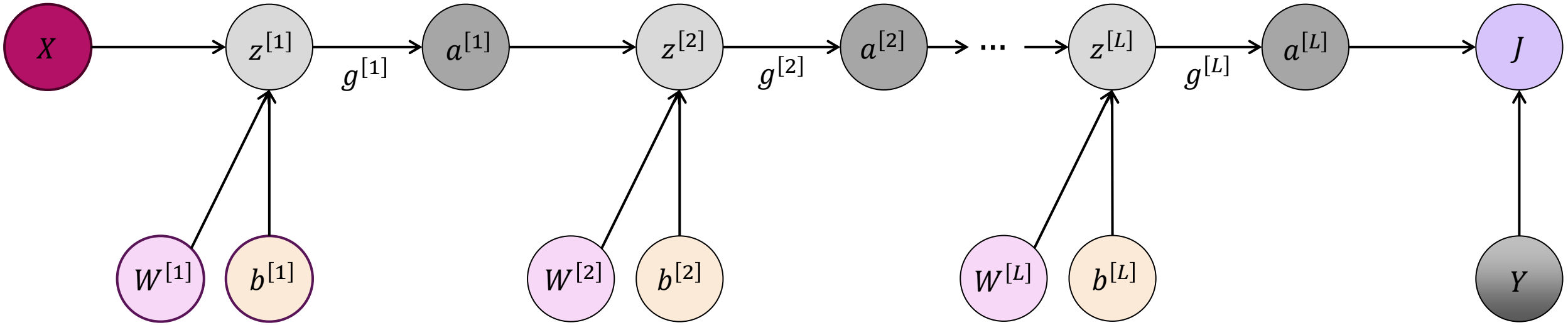
$$\boldsymbol{J(\theta; x^{(i)}, y^{(i)}) = -\sum_{j=1}^{k} \log\left(h_\theta(x^{(i)})_j \right) \mathbb{I}[y^{(i)} = j].}$$

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

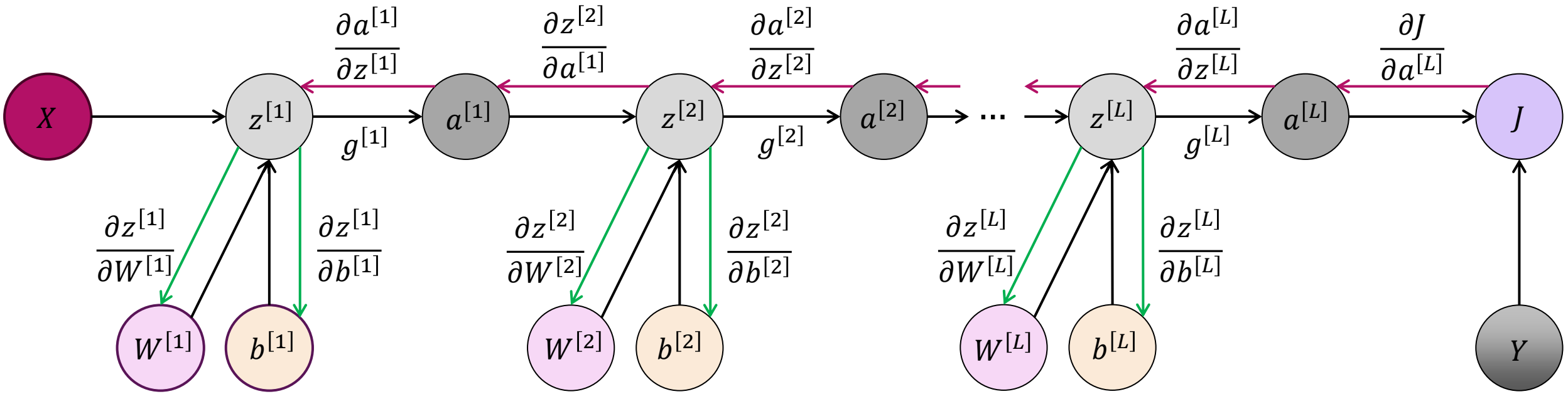
Given a datapoint (or batch of datapoints), one needs to compute the values of the activations corresponding to the input data layer-by-layer in the order $\ell \in \{1,2,\ldots,L\}$. This will yield the loss value J associated with this data. This is referred to as **forward propagation** (or the **forward-pass**).



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

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

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

- Recall that the vector $a^{[\ell]} \in \mathbb{R}^{n[\ell]}$ is obtained from the equation $a^{[\ell]} = g^{[\ell]}\left(z^{[\ell]}\right)$, 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 $\left(g^{[\ell]}\right)'\left(z^{[\ell]}\right)$ is the element-wise derivative of $g^{[\ell]}$ evaluated at $z^{[\ell]}$) (where $\otimes$ denotes the element-wise multiplication operator)

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

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

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

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

- 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 $\frac{\partial J}{\partial z^{[\ell]}}$)

$$\underbrace{\frac{\partial J}{\partial b^{[\ell]}}}_{(n[\ell]\times 1)} = \underbrace{\frac{\partial J}{\partial z^{[\ell]}}}_{(n[\ell]\times 1)} \cdot \underbrace{\frac{\partial z^{[\ell]}}{\partial b^{[\ell]}}}_{(n[\ell]\times 1)} = \underbrace{\frac{\partial J}{\partial z^{[\ell]}}}_{(n[\ell]\times 1)}$$

Gradients of a Neural Network (Example)

Consider an MLP $h_\theta(x) \coloneqq \mathcal{N}(x; g, \theta \coloneqq \{W, b\})$ 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

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

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

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

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

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

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

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

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

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).

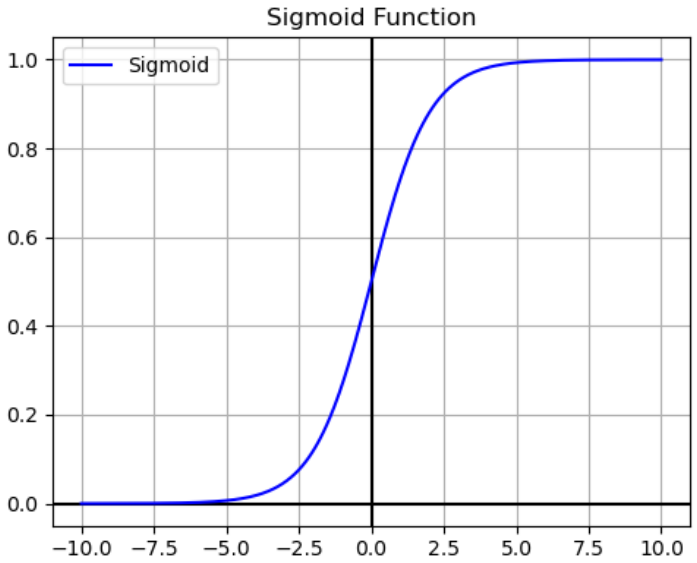
End For

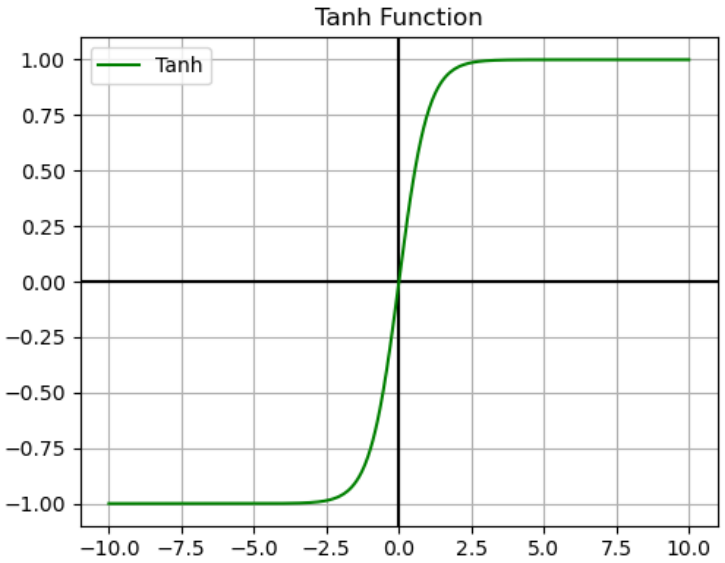
Return the trained ML $\mathcal{N}$.

Activation Functions

Sigmoid

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



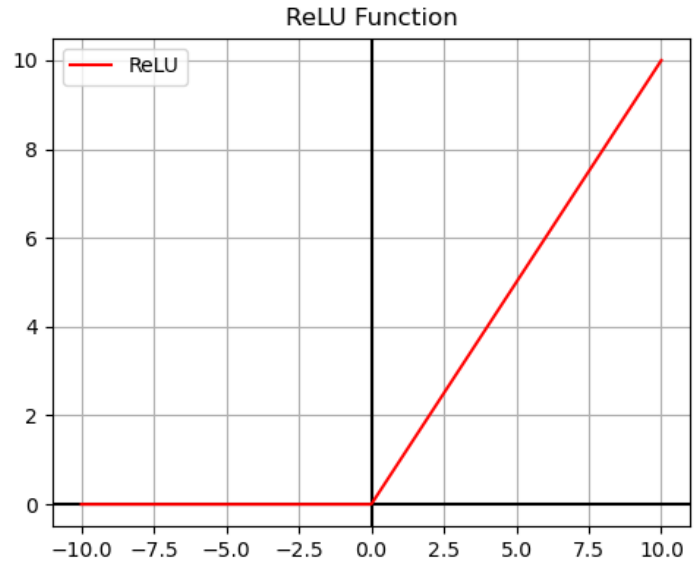


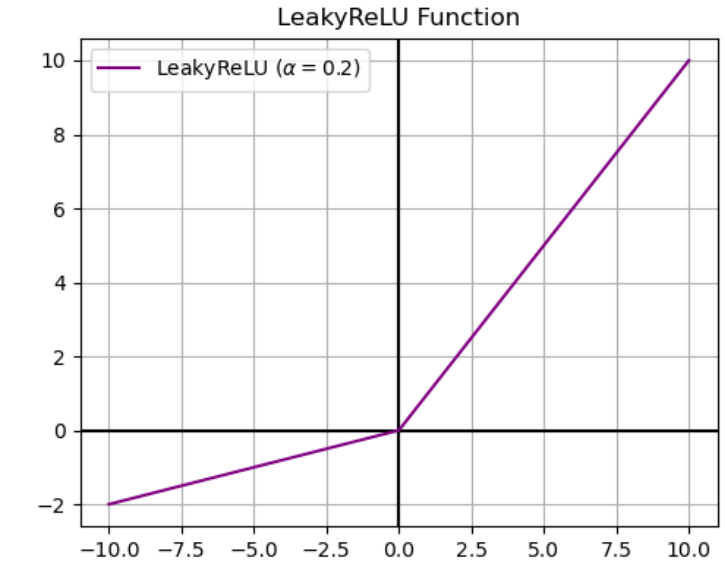
Hyperbolic Tangent

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

ReLU

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





LeakyReLU

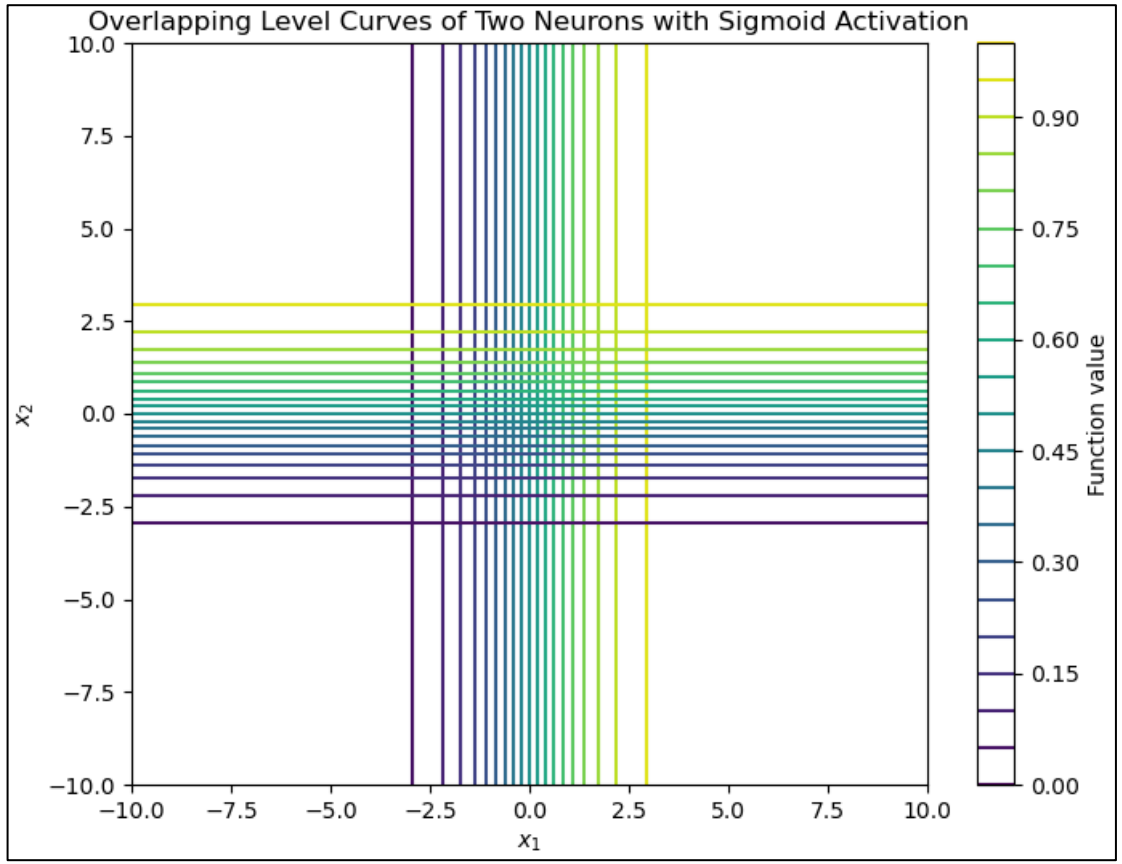
$$f(z;\alpha) = \begin{cases} z, \forall z > 0 \\ \alpha z, \text{otherwise} \end{cases}$$

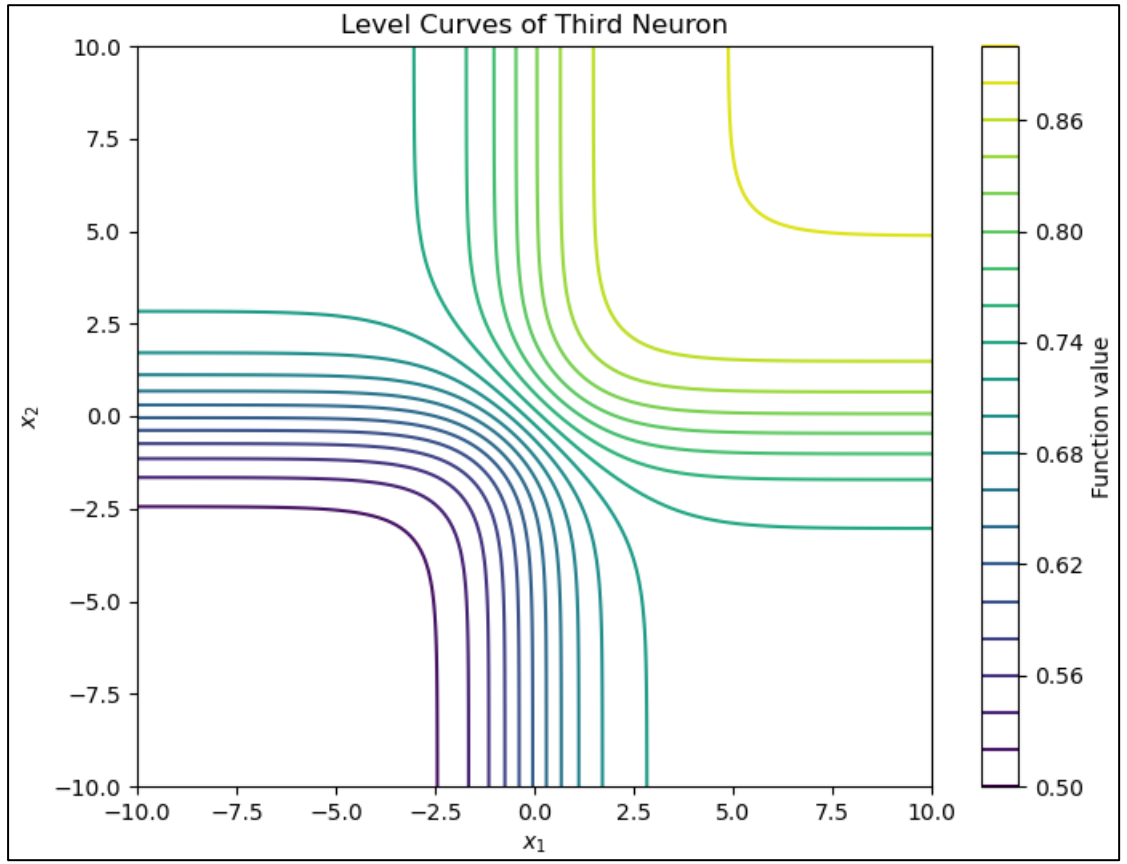
Activation Function Trade-Offs

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.

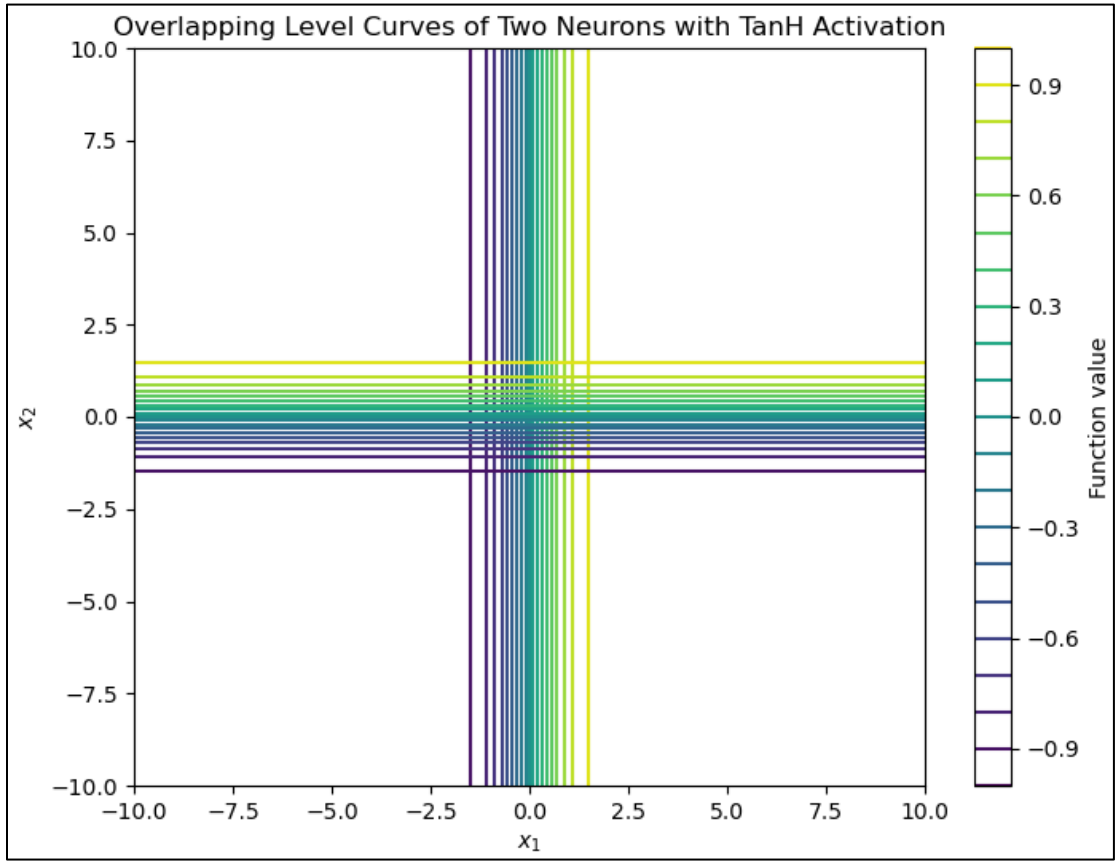
- **Sigmoid Function:** Outputs probabilities and is used as the output for binary classification problems (similarly, the softmax is used as the output for multi-class classification problems).
  - **Pros:** Smooth gradient (differentiable) and bounded range values (preventing extreme values).
  - **Cons:** 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.

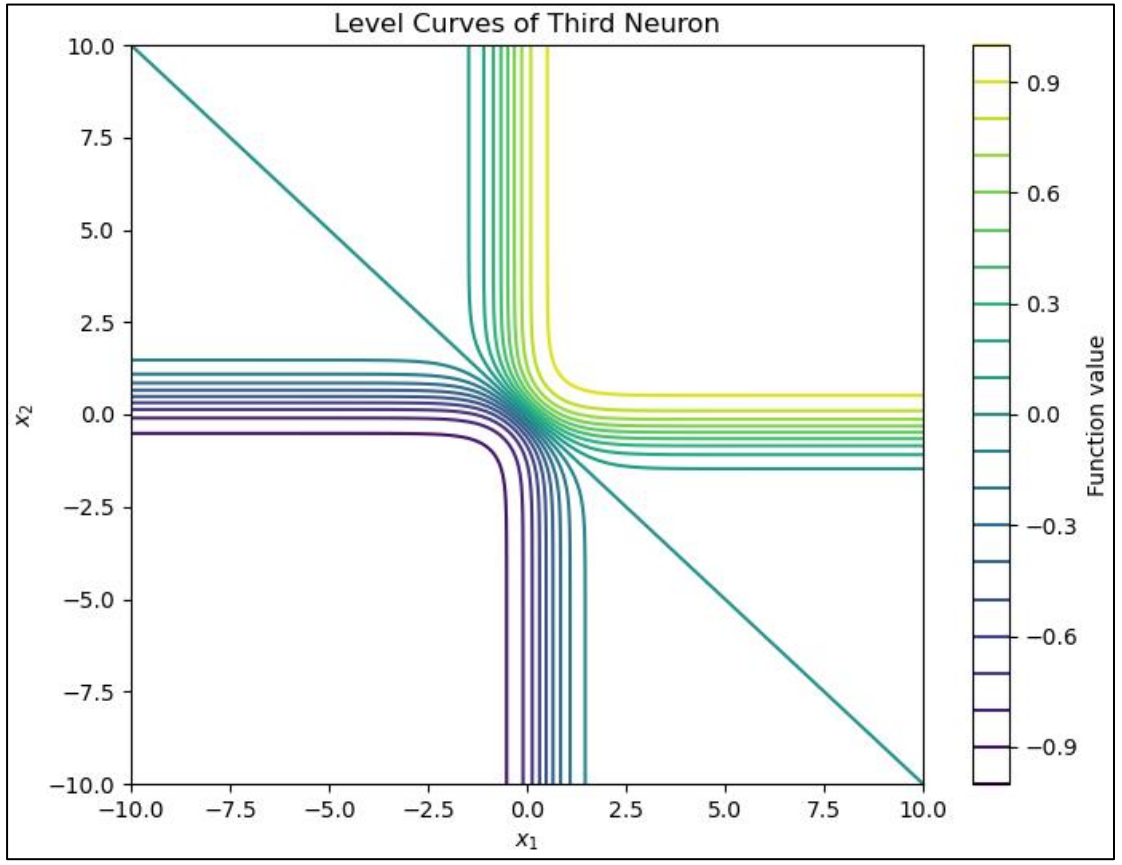
Sigmoid Decision Boundary



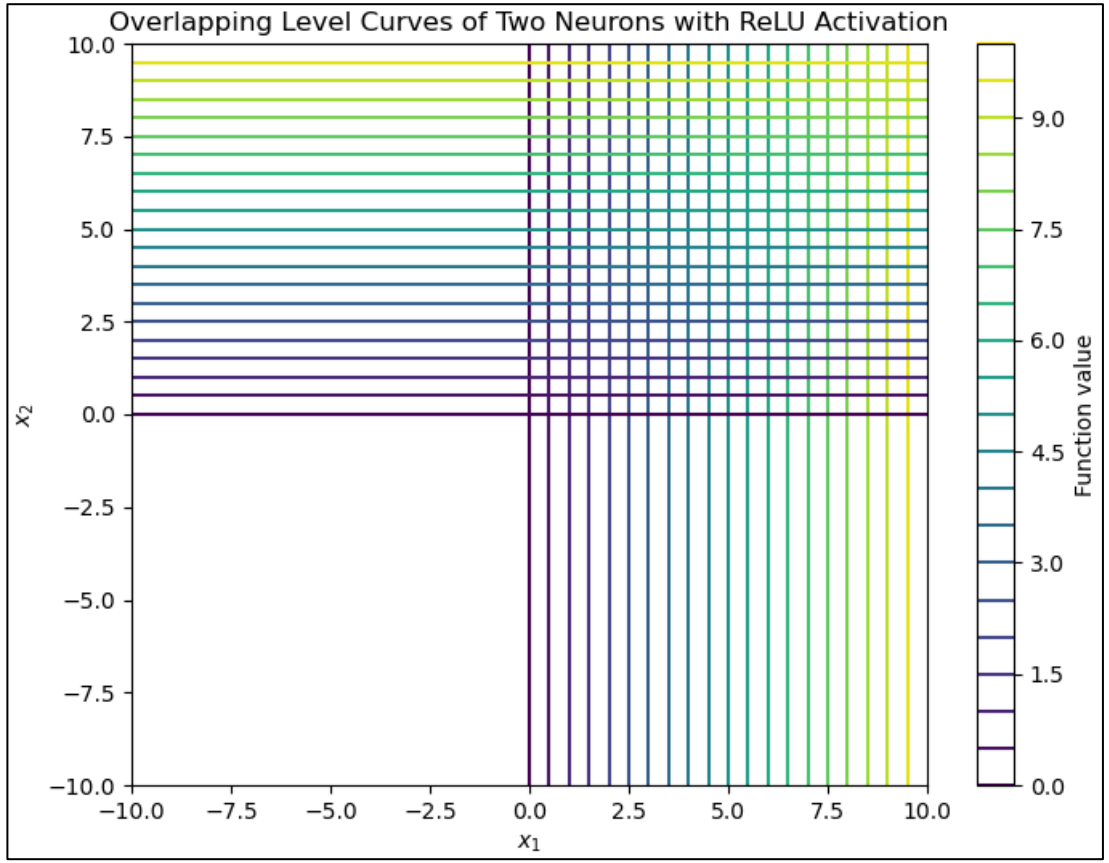


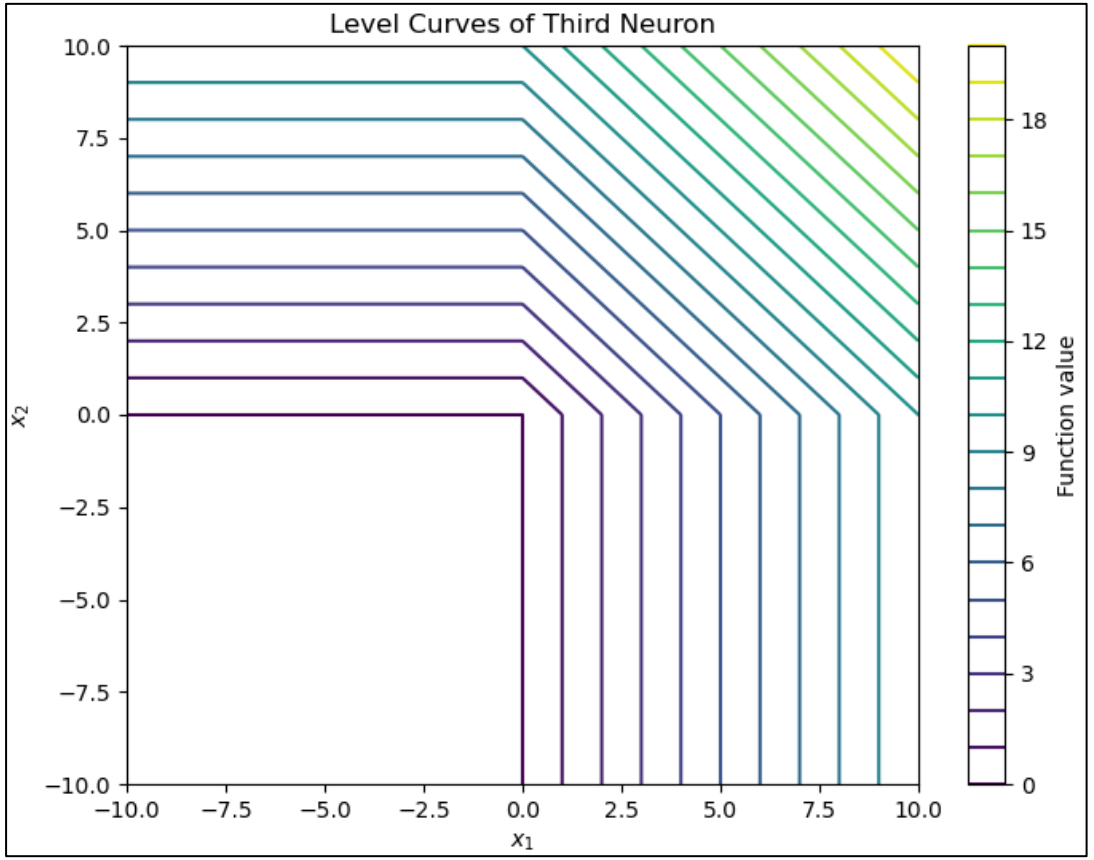
Hyperbolic Tangent Decision Boundary



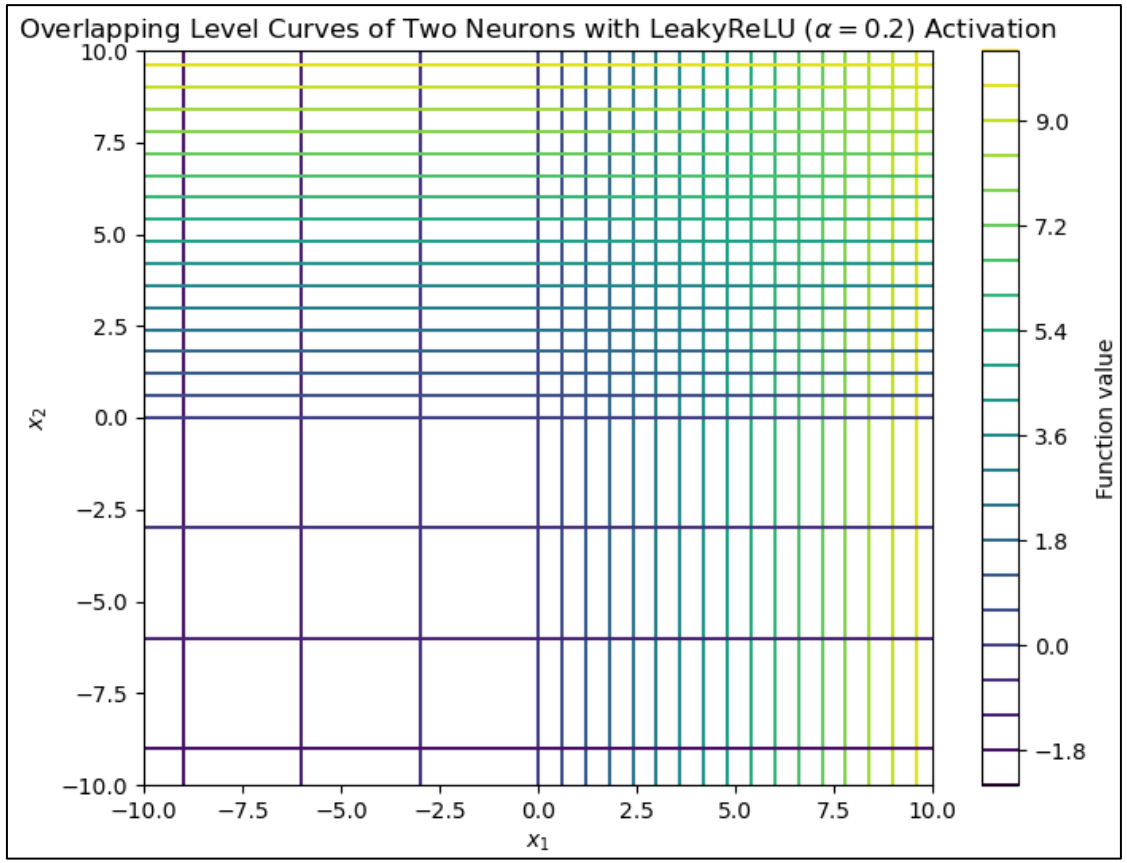


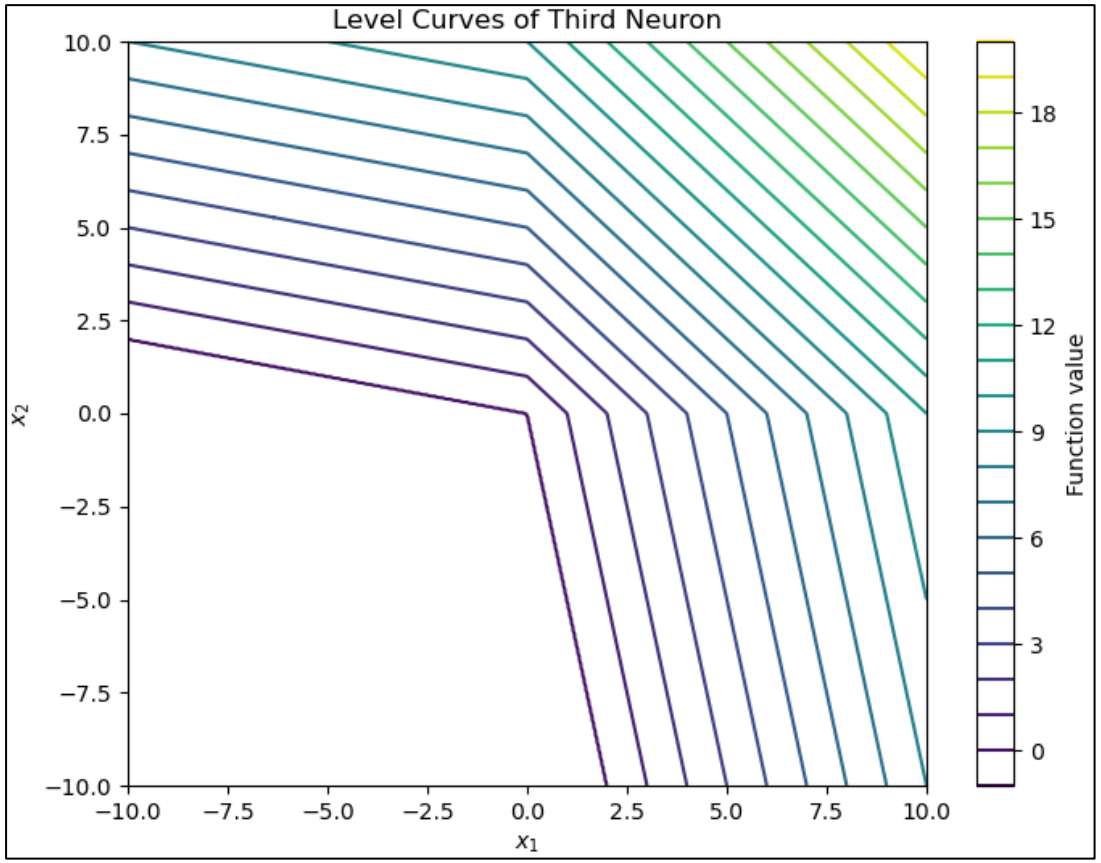
ReLU Decision Boundary





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

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

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

* This research was supported in part by NSF Grant DCR-619103, 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.

- 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

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

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-

White's participation was supported by a grant from the Guggenheim Foundation and by National Science Foundation Grant SES-8806990. The authors are grateful for helpful suggestions by the referees.

Requests for reprints should be sent to Halbert White, Department of Economics, D-008, UCSD, La Jolla, CA 92093.

Spatial-Driven Neural Architectures

INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS

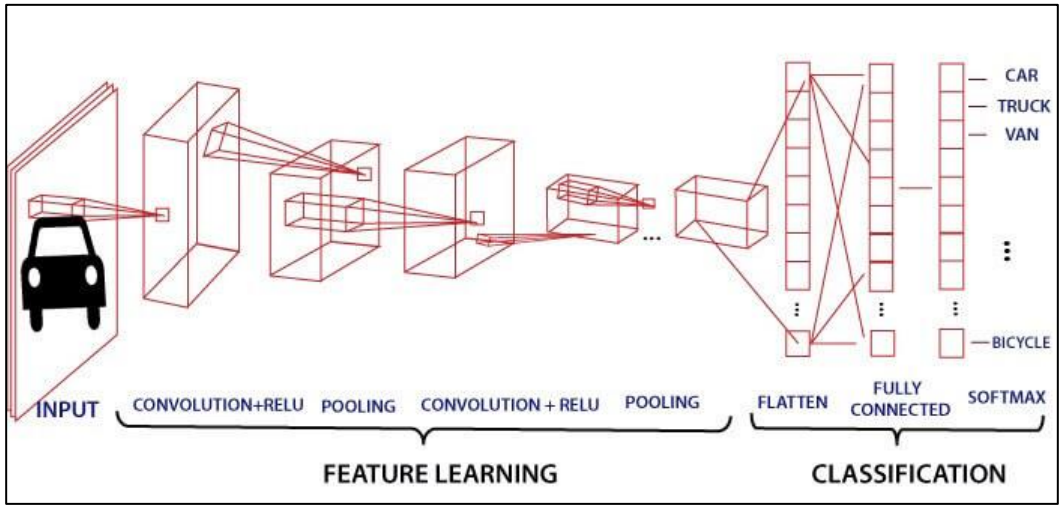
Overview of Convolutional Neural Networks

Convolutional Neural Networks (CNNs)

Convolutional neural networks are NNs with a specialized architecture, designed to excel at identifying patterns in spatially structured data (like images). These CNNs consist of a series of convolution layers and pooling layers.

- **Convolution Layers** can be thought of as identifying specific patterns in an image (lines, edges, textures, circles, colors, etc.) by outputting feature maps.
- **Pooling Layers** can be thought of as "summarizing" information within regions of an image, reducing its size by "zooming out", also making the network more robust and efficient.

Lastly, the output tensor of the convolutional and pooling layers is flattened out and fed into an MLP to ultimately generate a target prediction. As such, CNNs are highly effective at tasks like image recognition, object detection, and image segmentation.



See References

Kernel Matrices

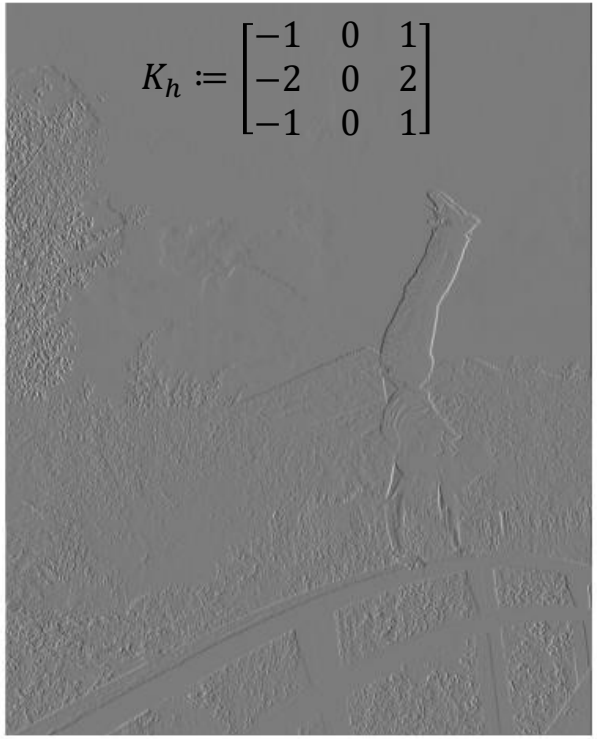
Kernel Matrix

In the context of convolutional neural networks, kernels are **small matrices** with elements that consist of learnable "**weights**". They are used on convolution operations to **detect certain patterns** such as horizontal or vertical edges, textures, or other different shapes in an input matrix. A kernel is applied to an image by "sliding" it over the input, pixel-by-pixel, and performing a dot-product at each position. There are different well-known kernel matrices that detect certain understood patterns, such as the horizontal and vertical Sobel kernels below, denoted by K_h and K_v, respectively.

Original Image

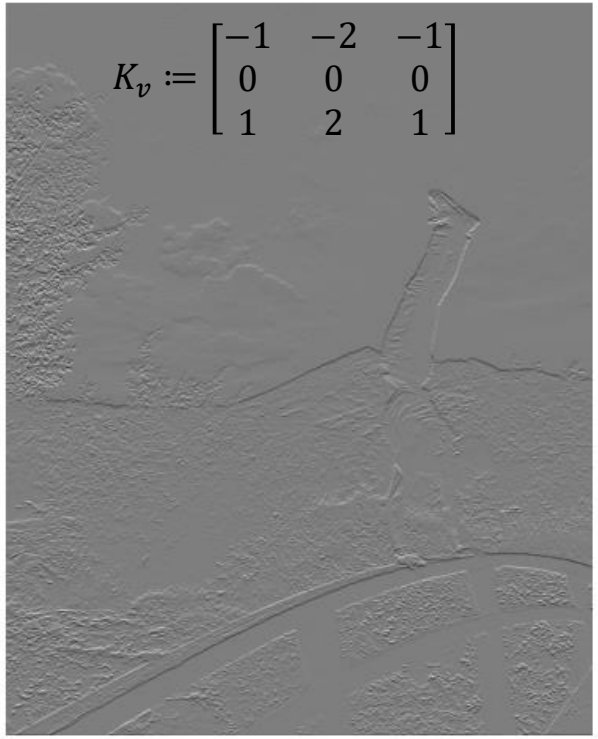
Horizontal Sobel Kernel

$$K_h := \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

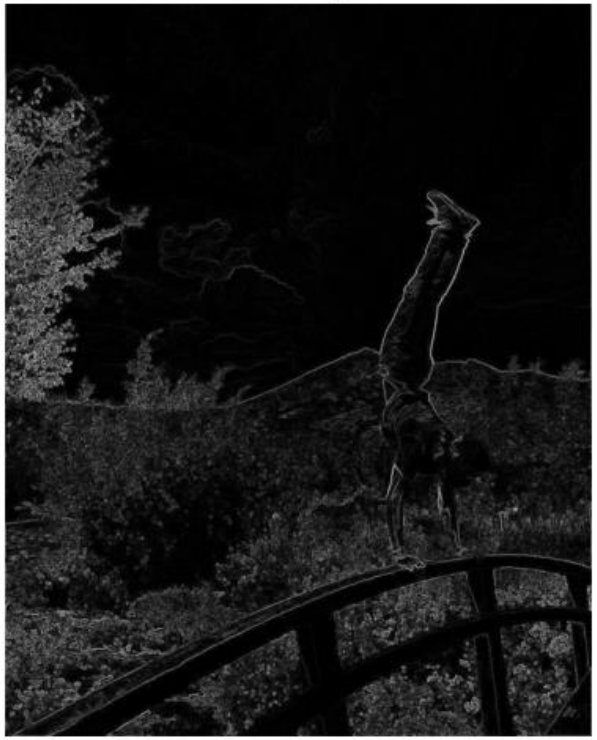


Vertical Sobel Kernel

$$K_v := \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$



Gradient Magnitude



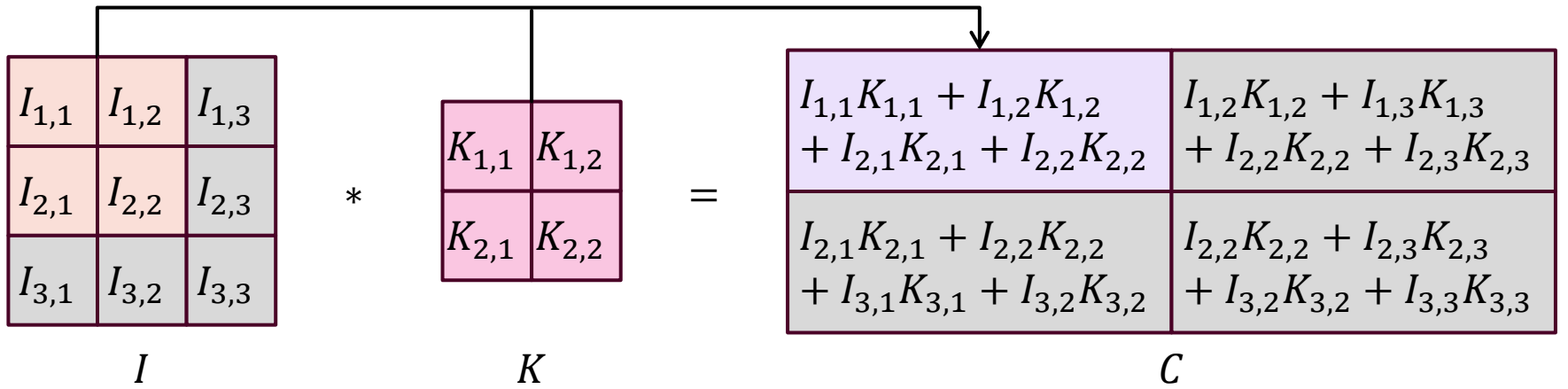
The Convolution Operation

Basic Convolution Operation (2×2)

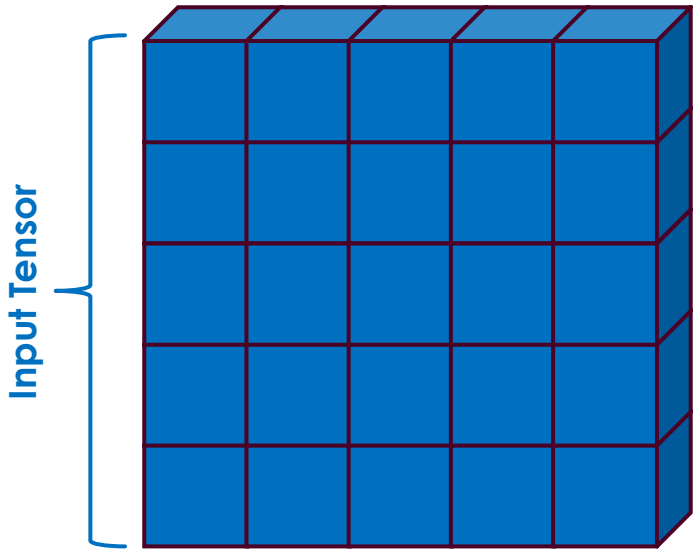
For the 2-dimensional case, when given an input matrix $I \in \mathbb{R}^{h \times w}$ (an image) with elements (pixel intensities) denoted by $I_{i,j}$ (for all $i \in \{1,2,\dots,h\}$ and $j \in \{1,2,\dots,w\}$), and a **kernel matrix** $K \in \mathbb{R}^{\hat{h} \times \hat{w}}$ with elements (kernel weights) denoted by $K_{\hat{\imath},\hat{\jmath}}$ (for all $\hat{\imath} \in \{1,2,\dots,\hat{h}\}$ and $\hat{\jmath} \in \{1,2,\dots,\hat{w}\}$), then the convolution operation output is denoted by the "**feature map**" matrix $C \in \mathbb{R}^{(h-\hat{h}) \times (w-\hat{w})}$ with elements $C_{i,j}$ that are computed by the convolution operation

$$C_{i,j} = \sum_{\hat{\imath}=1}^{\hat{h}} \sum_{\hat{\jmath}=1}^{\hat{w}} K_{\hat{\imath},\hat{\jmath}} I_{i+\hat{\imath},j+\hat{\jmath}} \,.$$

This is referred to as a **convolution** and the entire operation is sometimes expressed as $C = I * K$. The output C can then be fed through an activation function $g(\cdot)$ (the same functions as in MLPs) to learn non-linear patterns.



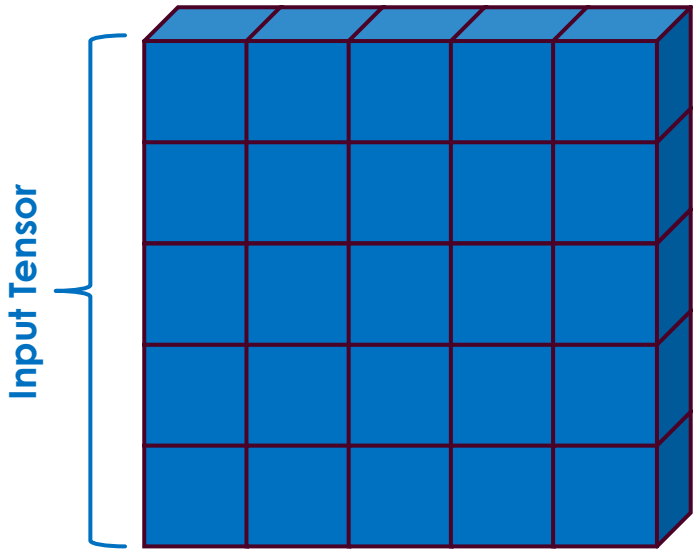
Convolution Operation Illustration

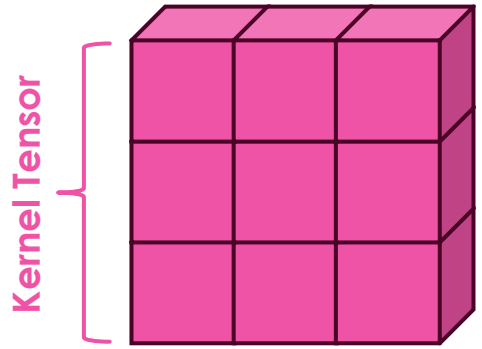


Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

Convolution Operation Illustration





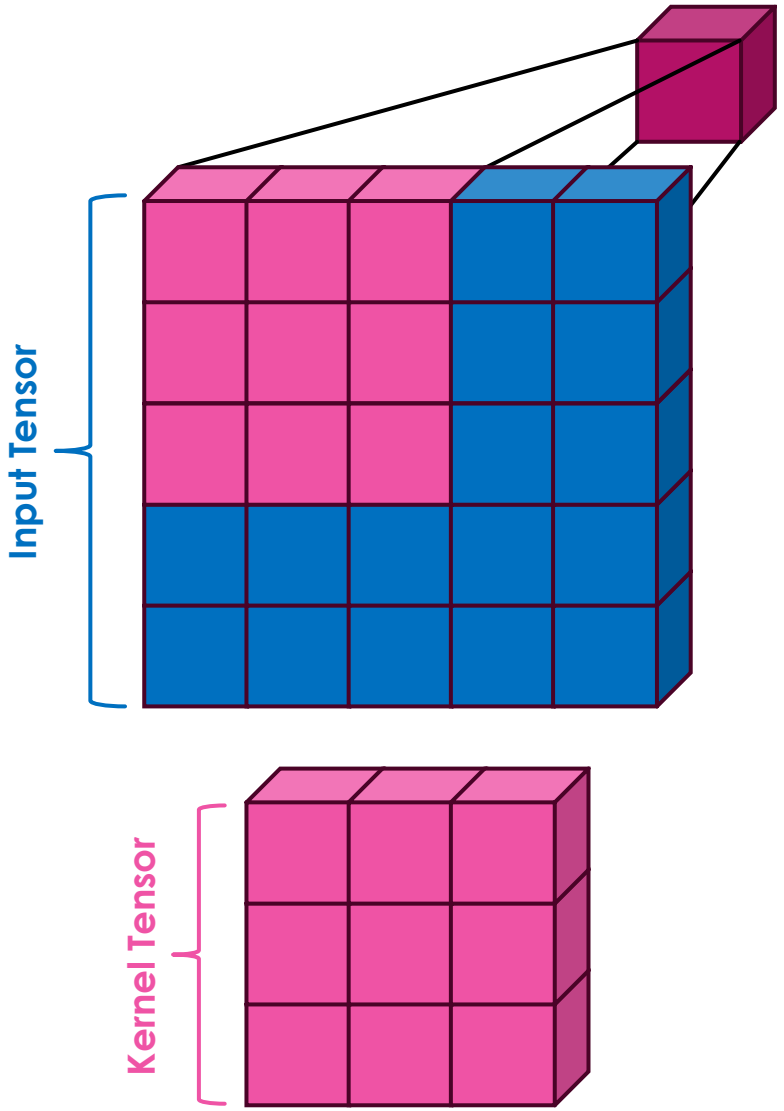
Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$c_{1,1} = 2 \cdot 1 + 0 \cdot 0 + 1 \cdot (-1) + 0 \cdot 0 + (-1) \cdot 2 + 2 \cdot 0 + (-2) \cdot 0 + 3 \cdot 1 + 0 \cdot 2 = 2$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

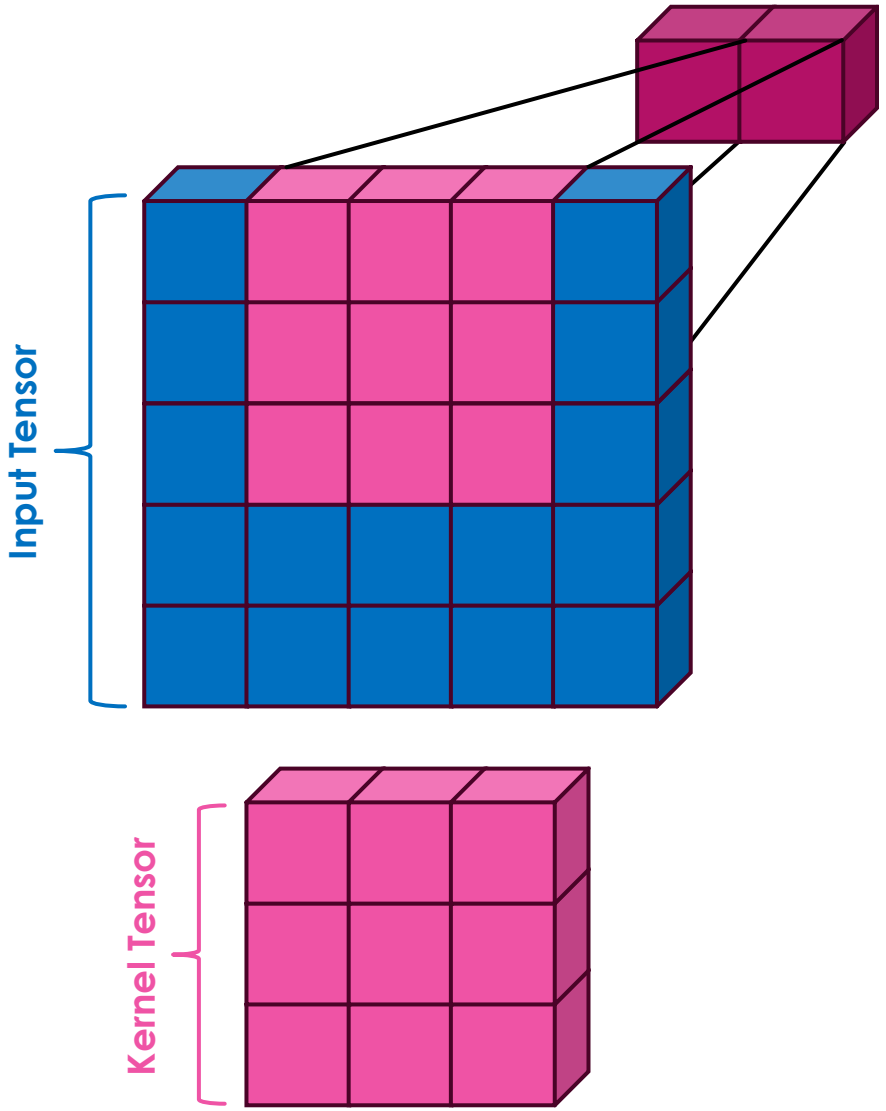
$=$ 2

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{1,2} = 0 \cdot 1 + 1 \cdot 0 + 1 \cdot (-1) + (-1) \cdot 0 + 2 \cdot 2 + 2 \cdot 0 + 3 \cdot 0 + 0 \cdot 1 + 0 \cdot 2 = 3$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

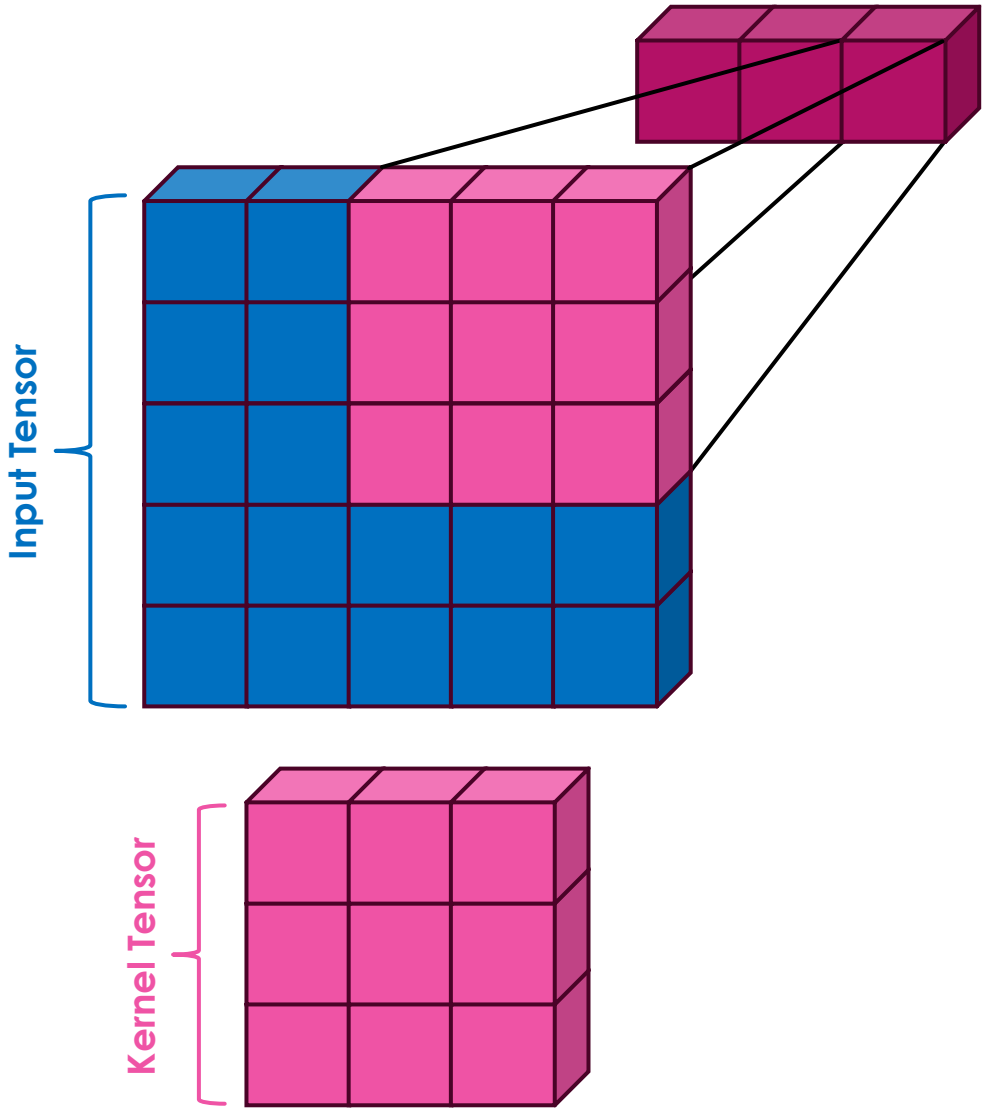
2	3

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{1,3}$$
$$= 1 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1) + 2 \cdot 0 + 2 \cdot 2 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 + 1 \cdot 2$$
$$= 4$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

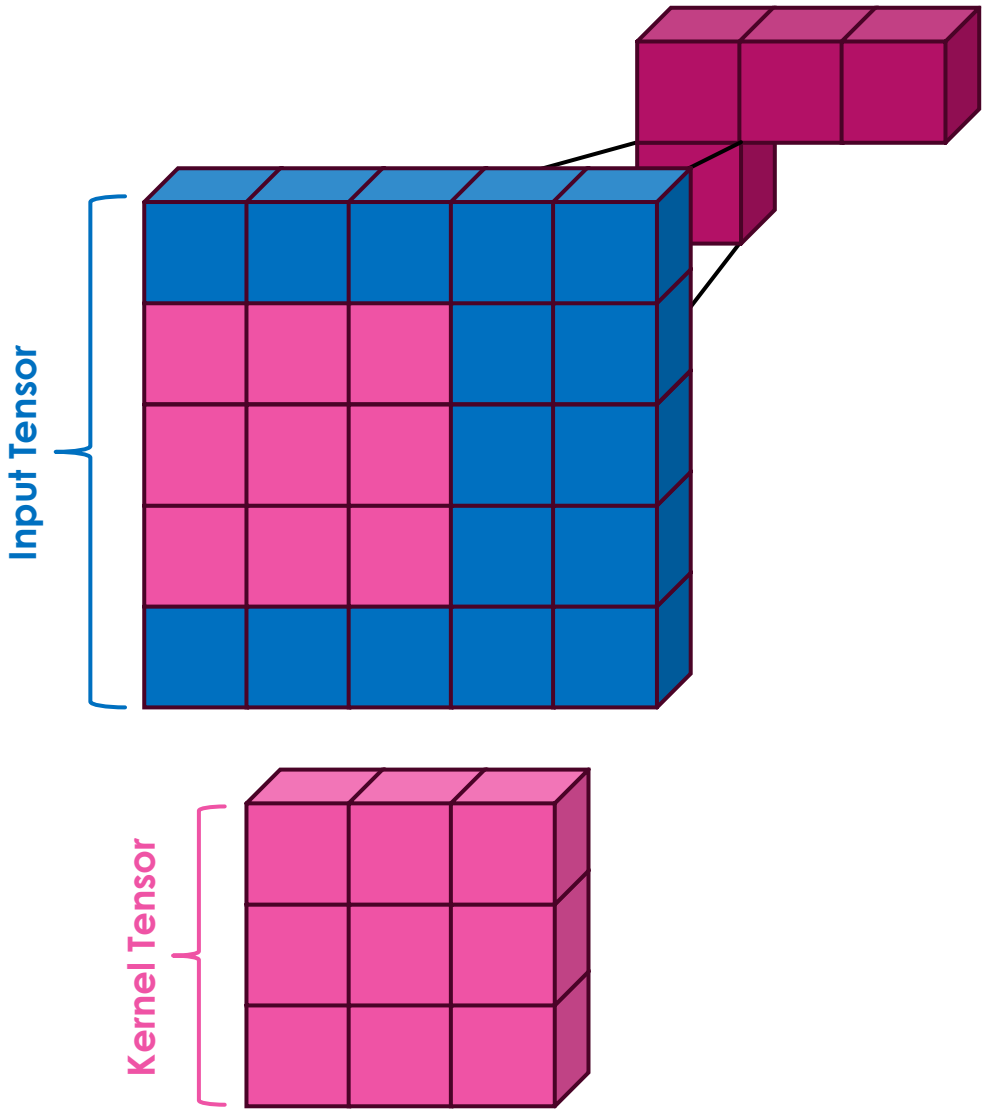
2	3	4

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{2,1}$$
$$= \mathbf{0} \cdot \mathbf{1} + (-\mathbf{1}) \cdot \mathbf{0} + \mathbf{2} \cdot (-\mathbf{1}) + (-\mathbf{2}) \cdot \mathbf{0} + \mathbf{3} \cdot \mathbf{2} + \mathbf{0} \cdot \mathbf{0} + \mathbf{0} \cdot \mathbf{0} + \mathbf{1} \cdot \mathbf{1} + (-\mathbf{3}) \cdot \mathbf{2}$$
$$= -\mathbf{1}$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

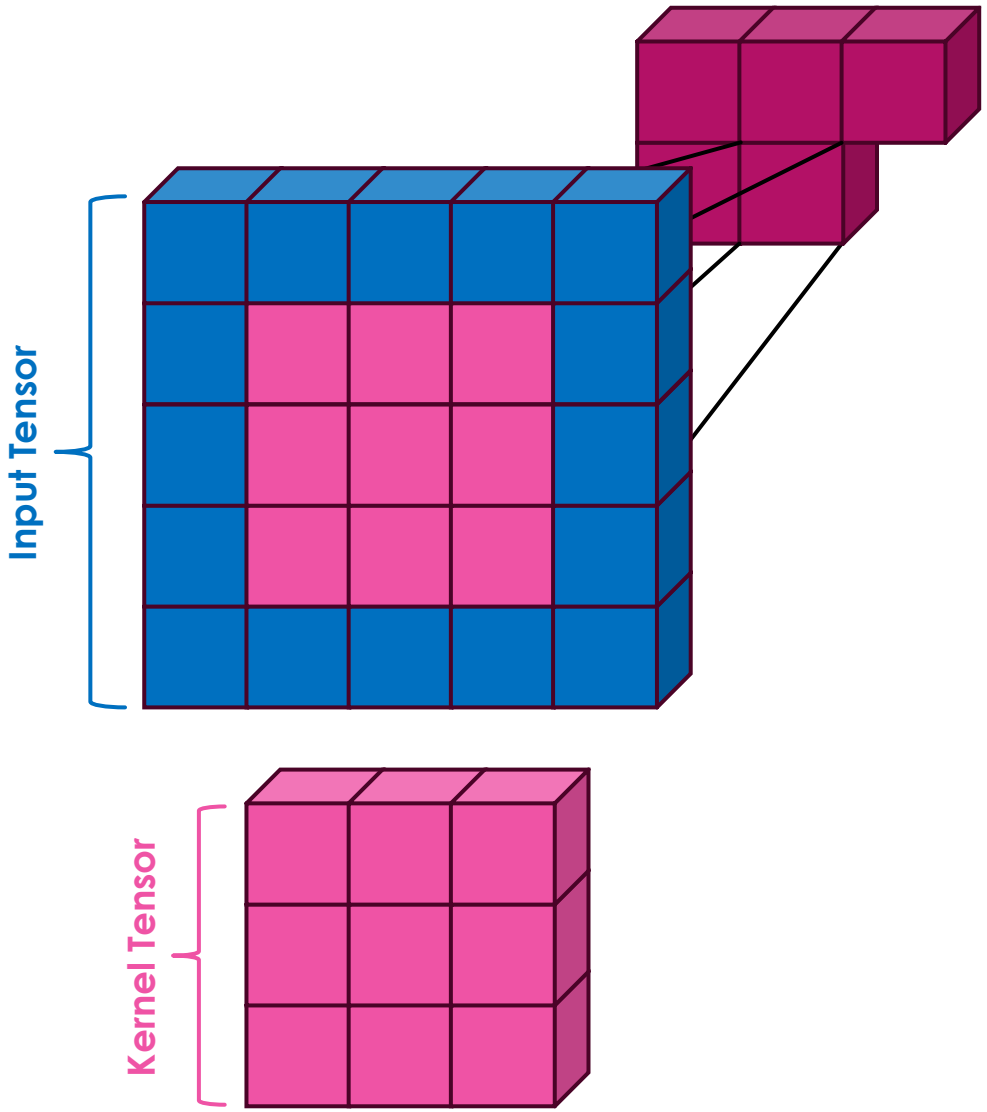
2	3	4
−1		

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{2,2}$$
$$= (-1) \cdot 1 + 2 \cdot 0 + 2 \cdot (-1) + 3 \cdot 0 + 0 \cdot 2 + 0 \cdot 0 + 1 \cdot 0 + (-3) \cdot 1 + 2 \cdot 2$$
$$= -2$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

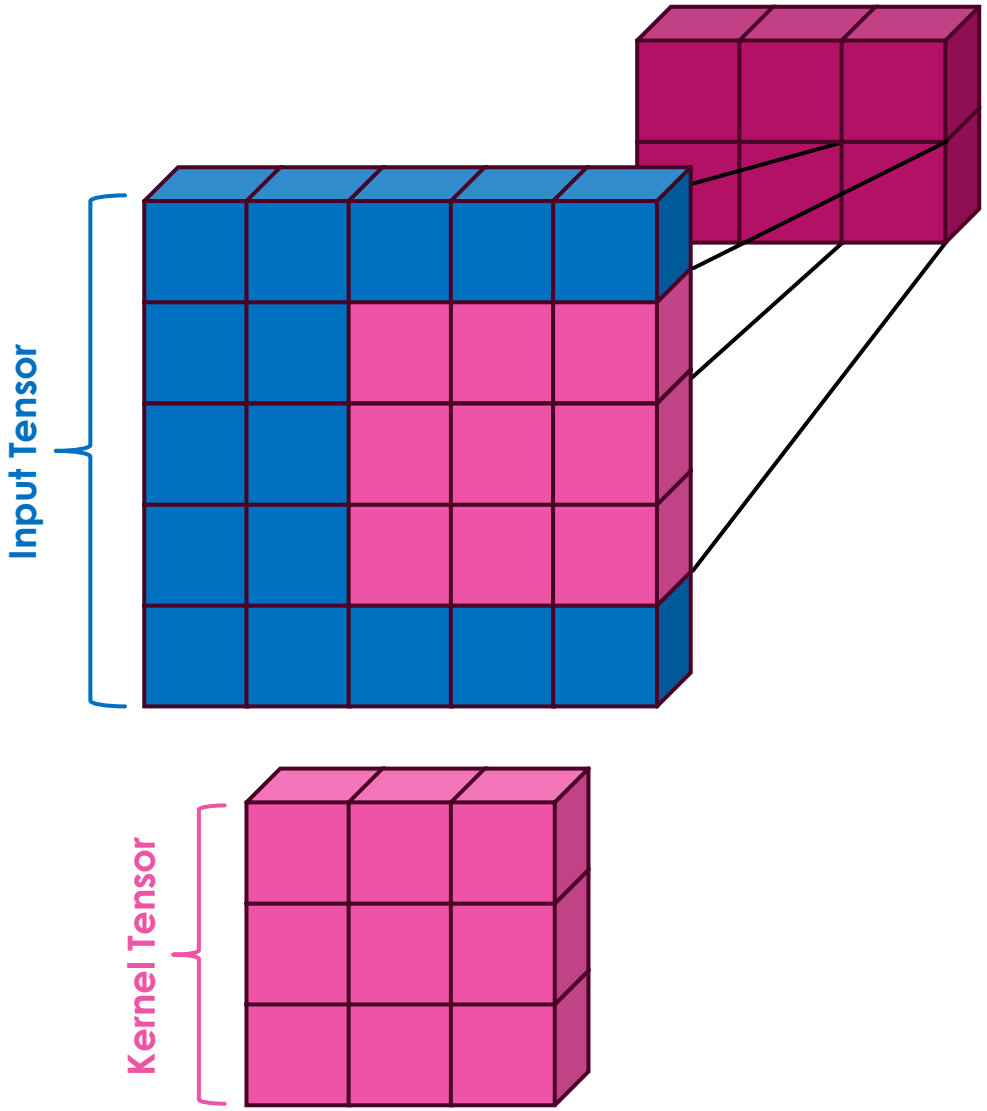
2	3	4
−1	−2	

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{2,3}$$
$$= 2 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1) + 0 \cdot 0 + 0 \cdot 2 + 1 \cdot 0 + (-3) \cdot 0 + 2 \cdot 1 + 3 \cdot 2$$
$$= 10$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

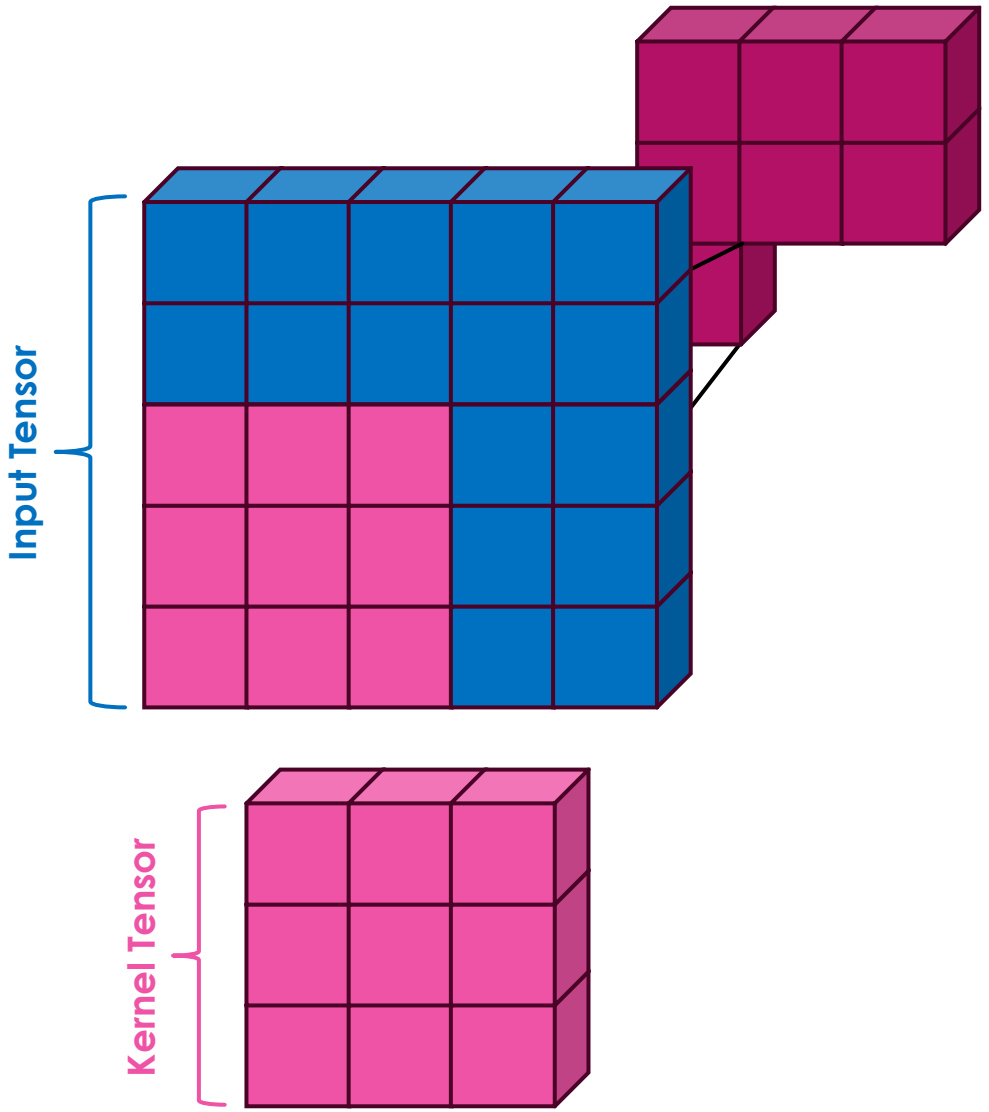
2	3	4
−1	−2	10

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{3,1}$$
$$= (-2) \cdot 1 + 3 \cdot 0 + 0 \cdot (-1) + 0 \cdot 0 + 1 \cdot 2 + (-3) \cdot 0 + 1 \cdot 0 + 1 \cdot 1 + (-2) \cdot 2$$
$$= -3$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

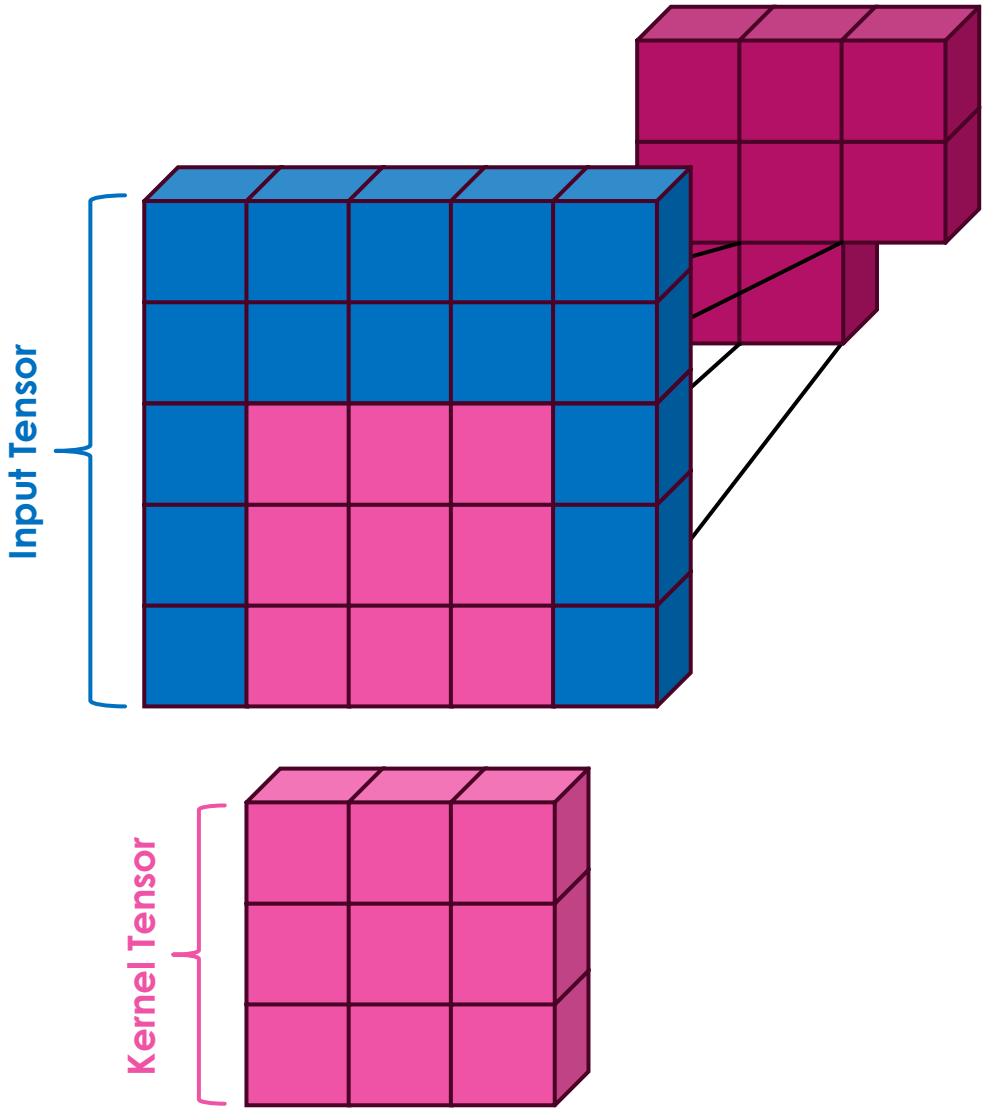
2	3	4
−1	−2	10
−3		

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{3,2}$$
$$= 3 \cdot 1 + 0 \cdot 0 + 0 \cdot (-1) + 1 \cdot 0 + (-3) \cdot 2 + 2 \cdot 0 + 1 \cdot 0 + (-2) \cdot 1 + 0 \cdot 2$$
$$= -5$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

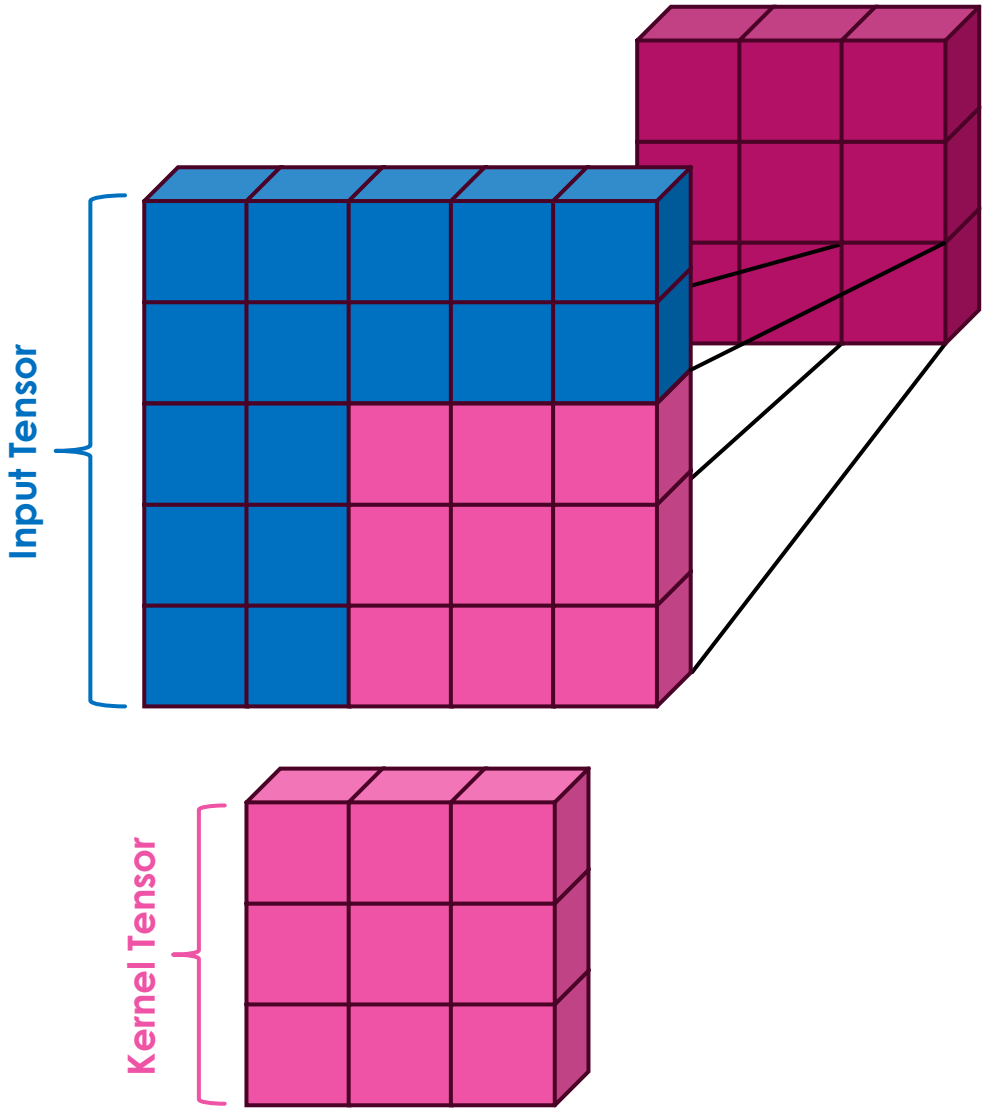
2	3	4
−1	−2	10
−3	−5	

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



$$C_{3,3}$$
$$= \mathbf{0} \cdot 1 + \mathbf{0} \cdot 0 + \mathbf{1} \cdot (-1) + (-\mathbf{3}) \cdot 0 + \mathbf{2} \cdot 2 + \mathbf{3} \cdot 0 + (-\mathbf{2}) \cdot 0 + \mathbf{0} \cdot 1 + (-\mathbf{1}) \cdot 2$$
$$= \mathbf{1}$$

Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$=$

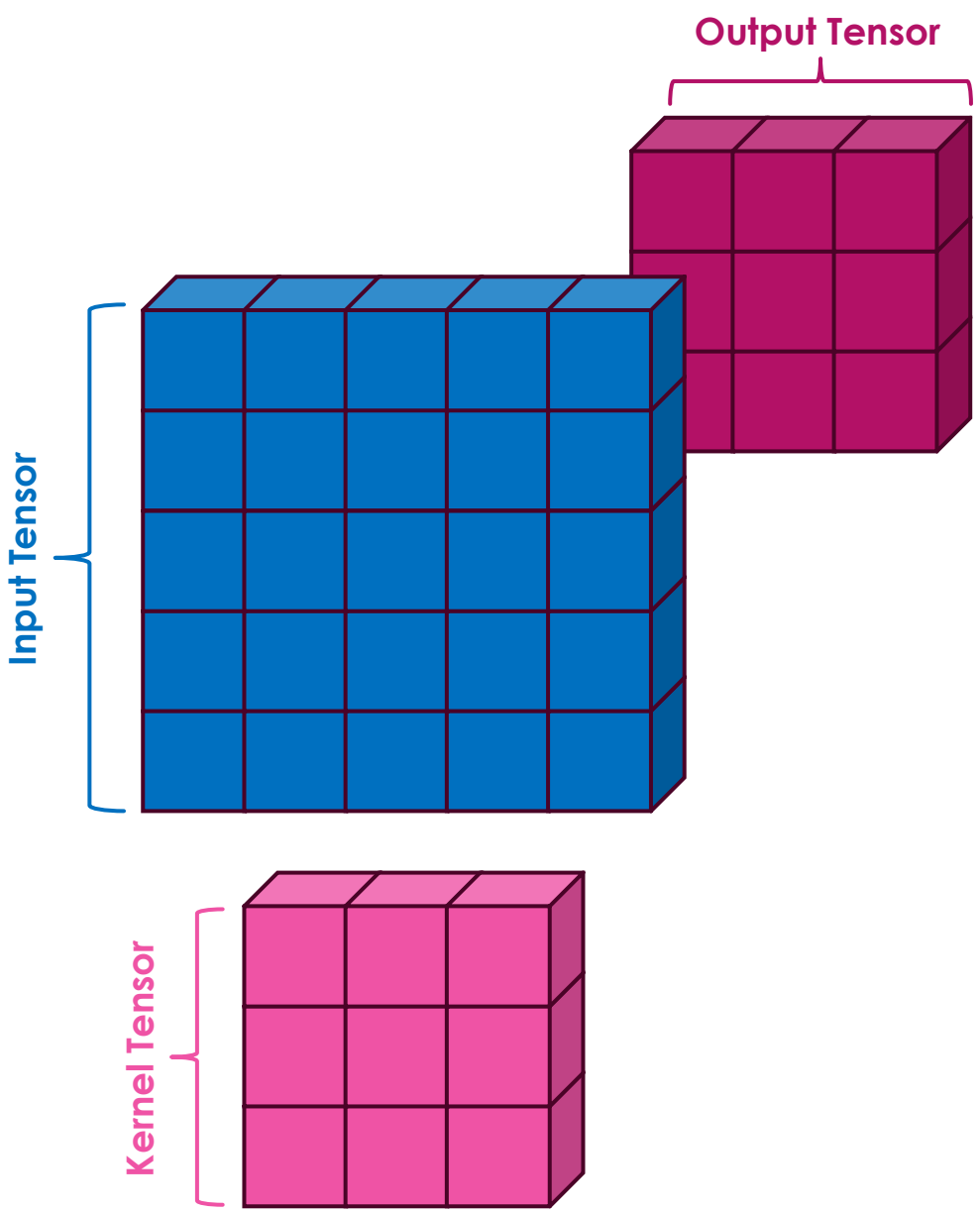
2	3	4
−1	−2	10
−3	−5	1

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

Convolution Operation Illustration



Input Tensor

2	0	1	1	3
0	−1	2	2	0
−2	3	0	0	1
0	1	−3	2	3
1	1	−2	0	−1

$*$

Kernel Tensor

1	0	−1
0	2	0
0	1	2

$=$

Output Tensor

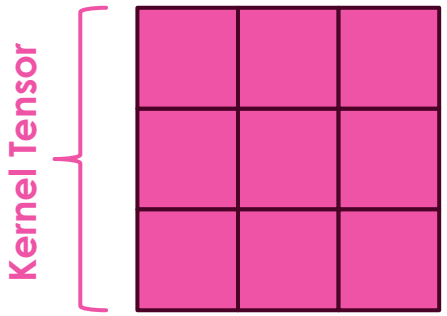
2	3	4
−1	−2	10
−3	−5	1

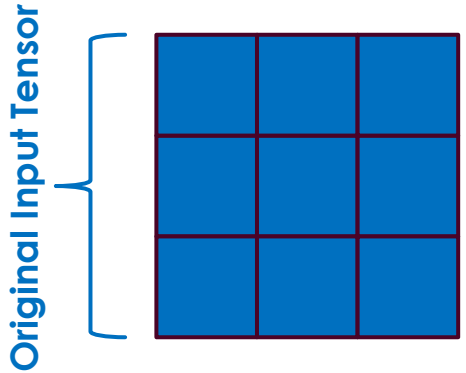
Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.



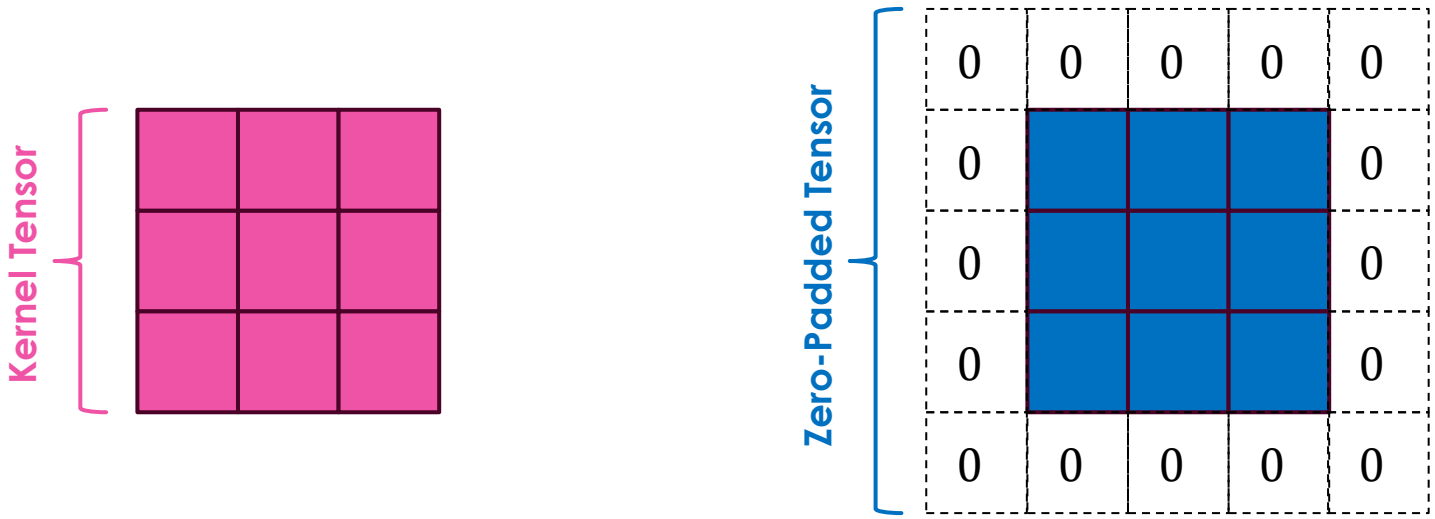


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

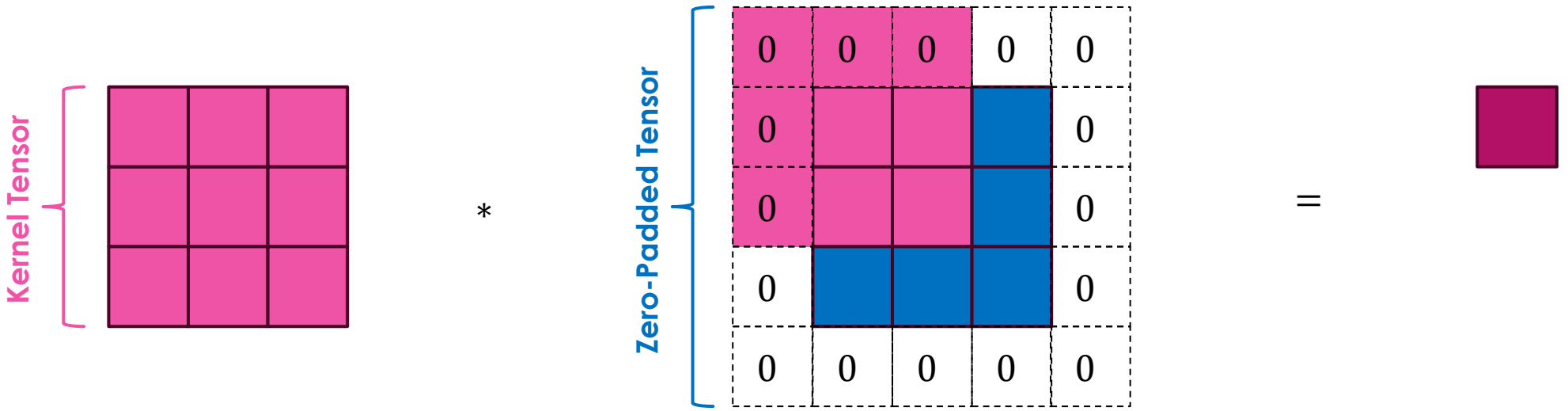


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

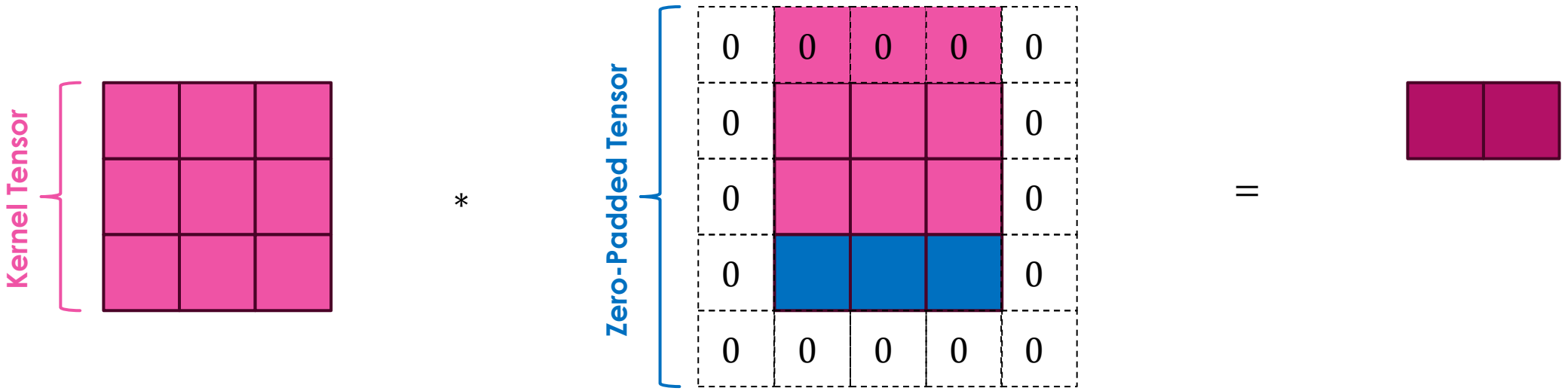


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

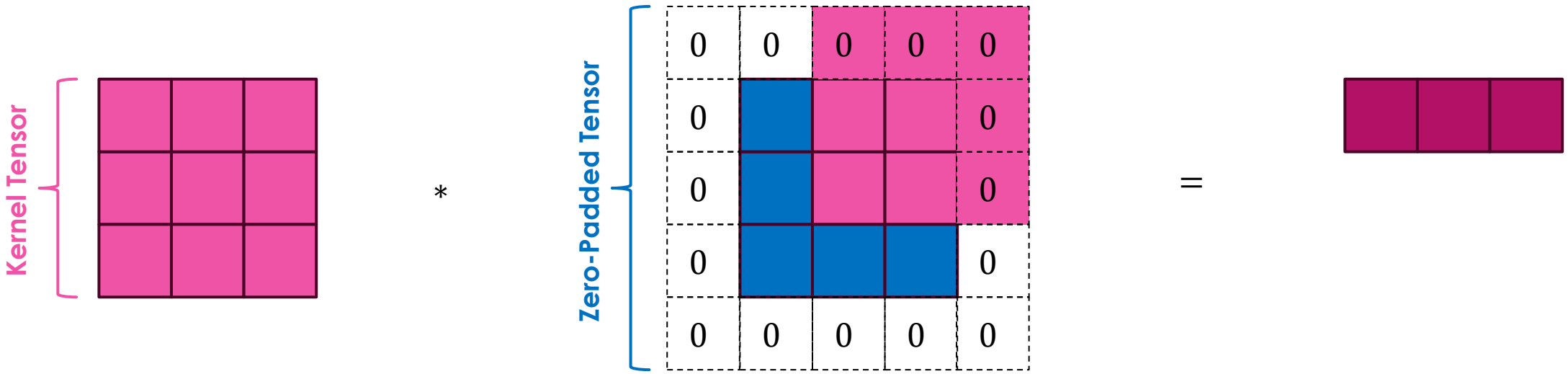


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

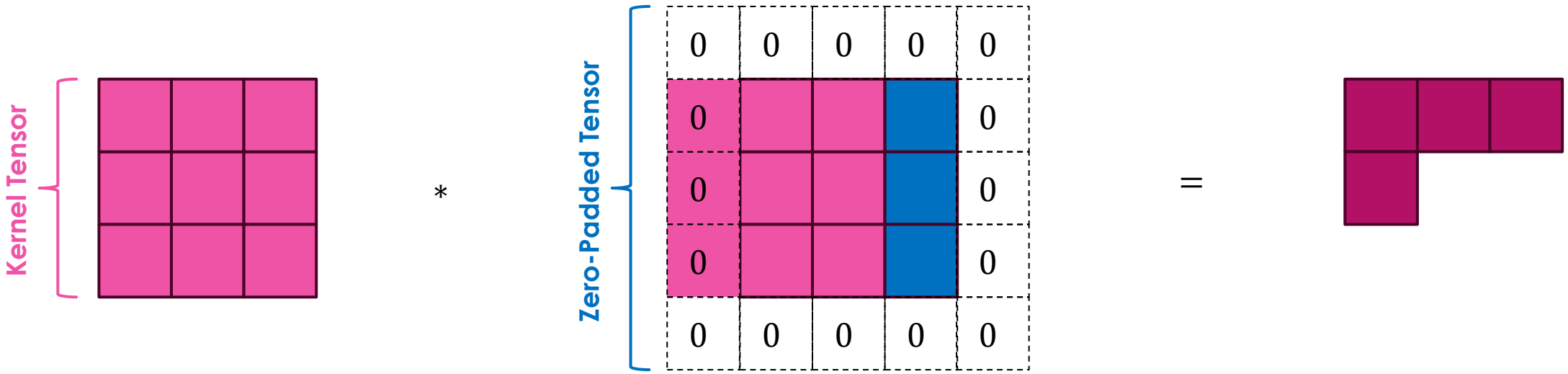


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

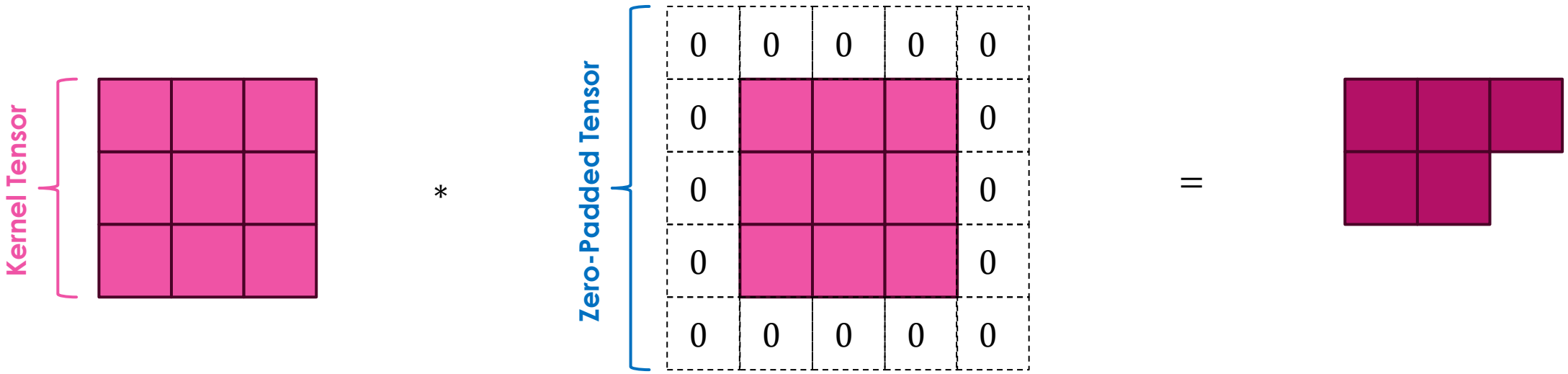


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

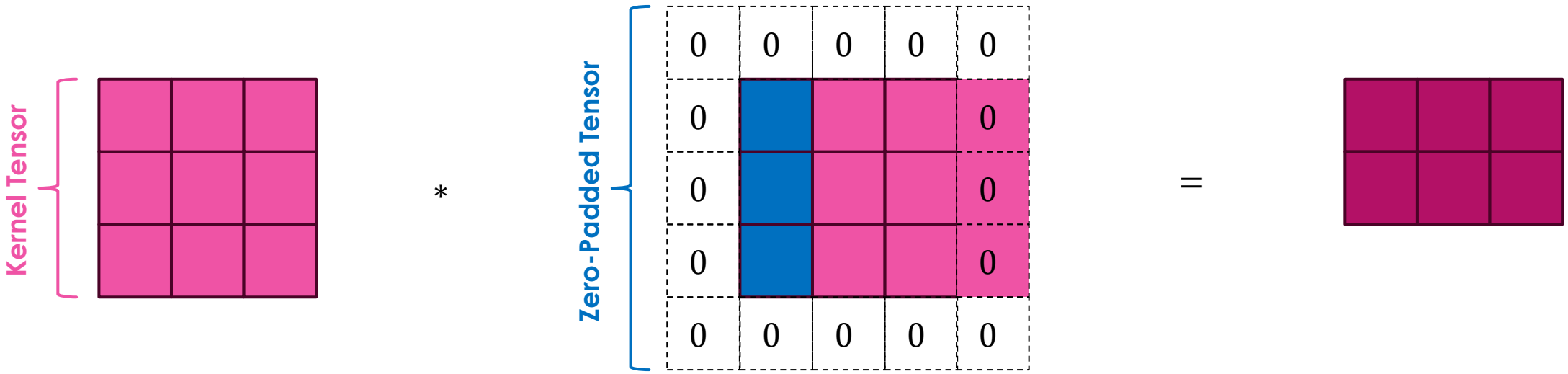


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

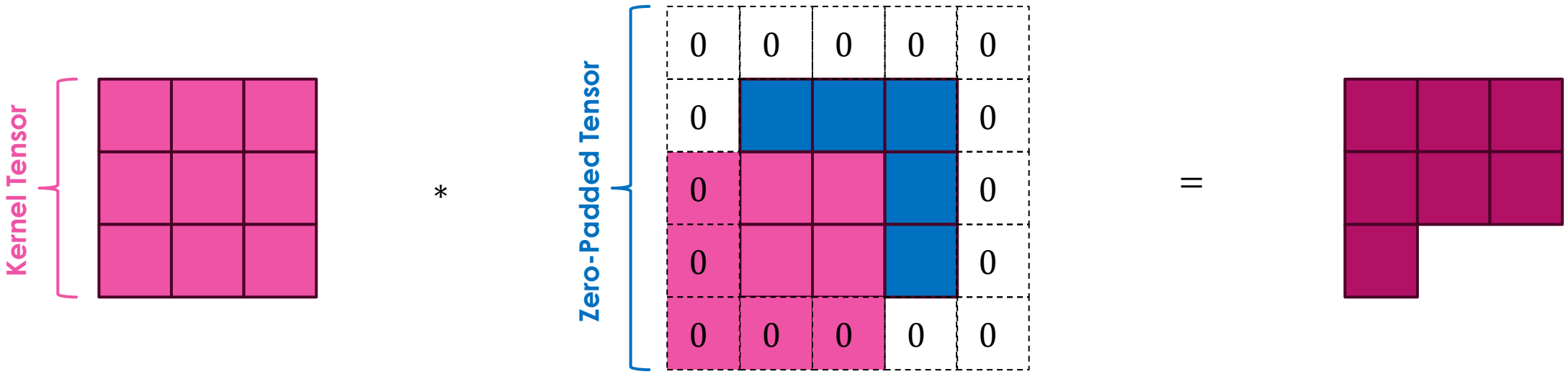


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

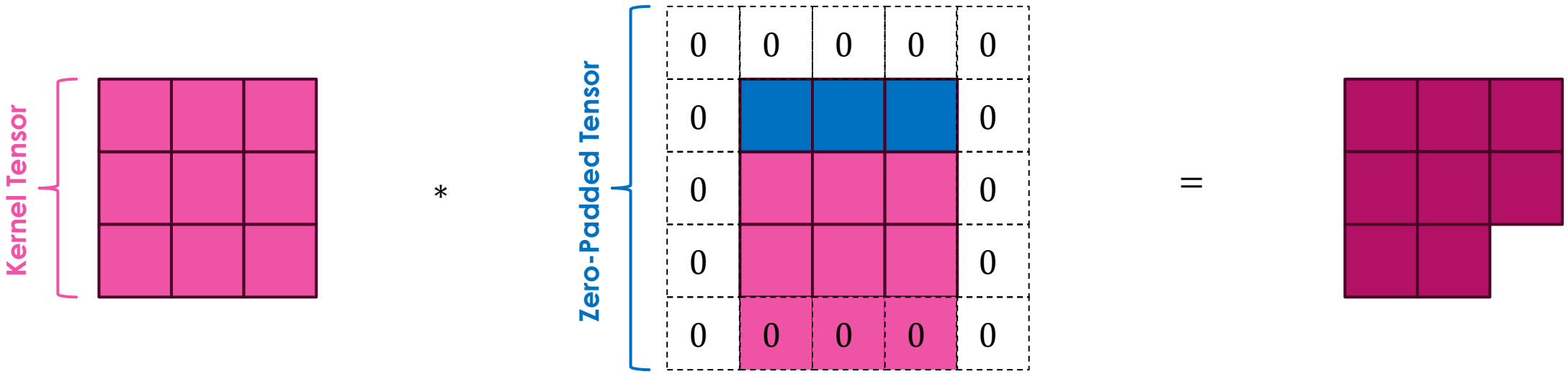


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

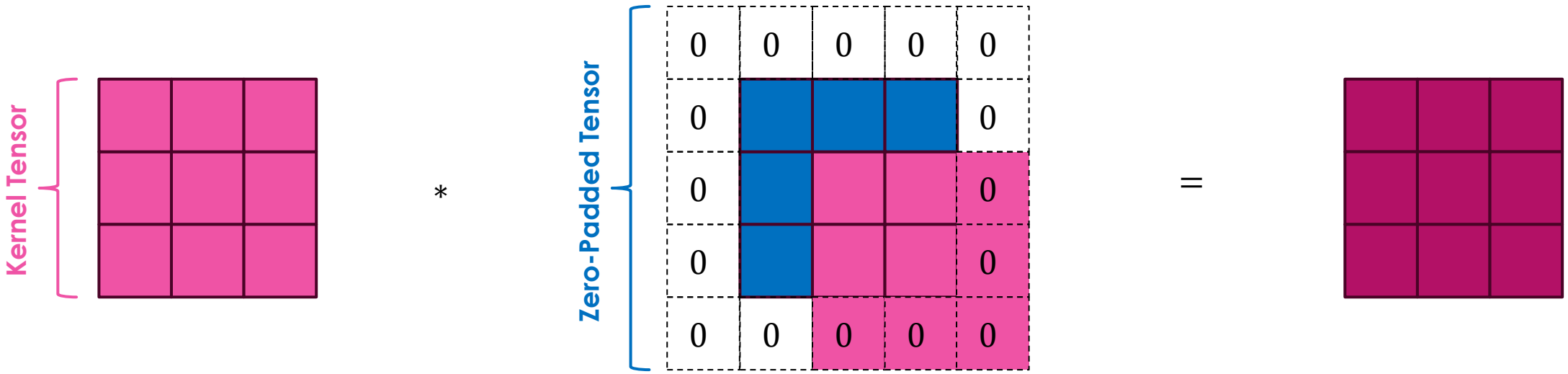


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

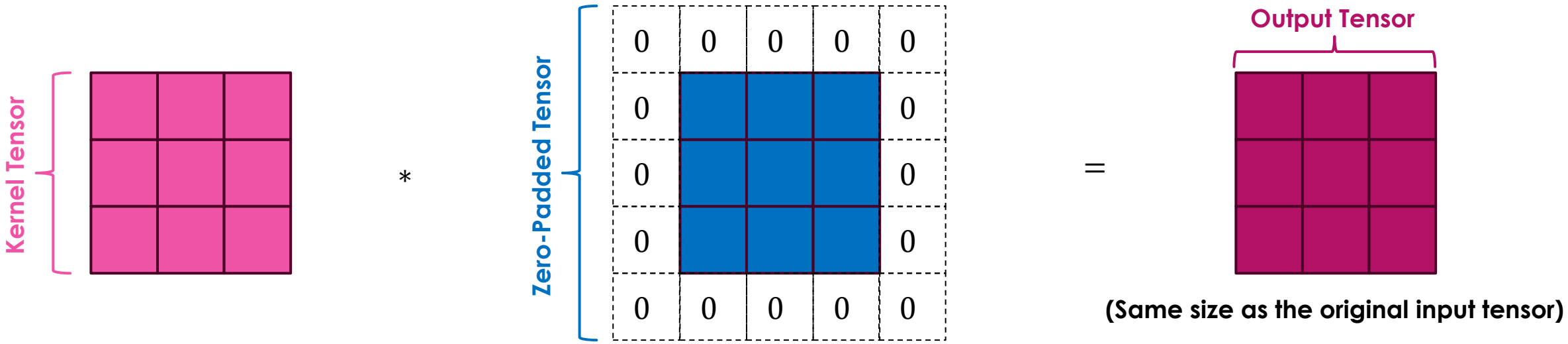


Padding

Often, when performing a convolution, one may wish for the output feature map to retain the same dimensions as the input. A common procedure to do this is padding.

Padding

Padding is a hyperparameter $p \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by adding p rows & columns to the edges of I (resulting in an image matrix of dimensions $(h + 2p) \times (w + 2p)$. These padded pixels can take on a particular value; however, the most common practice is to simply use what is called "**zero-padding**" which simply gives the padded pixels values of 0.

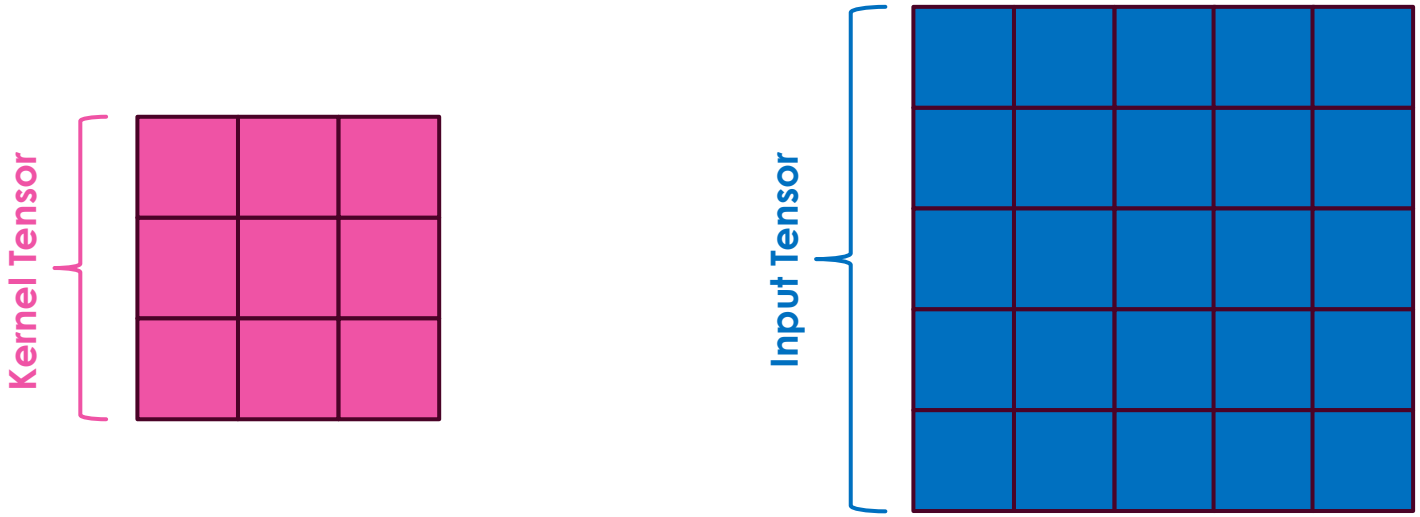


Stride

Opposing the idea of trying to preserve the size of a tensor when performing a convolution, often one wishes to reduce the size of the output. One way of doing this is by incorporating stride in the convolution operation.

Stride

Stride is a hyperparameter $s \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by "skipping" over s inputs when sliding the kernel matrix across the input. By default, stride is assumed to be $s = 1$. Increasing stride is one way of substantially decreasing the size of the output feature map.



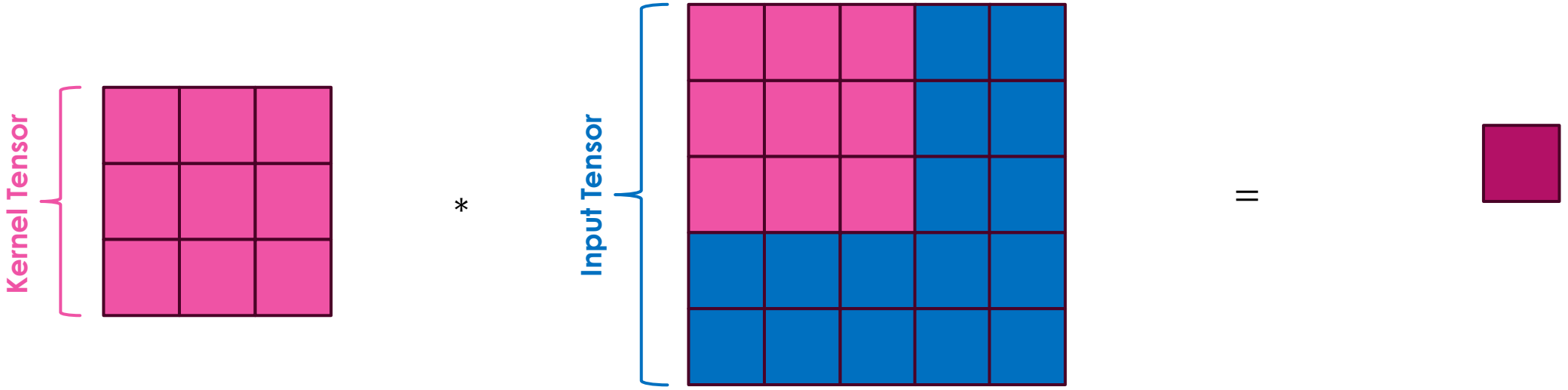
Stride

Opposing the idea of trying to preserve the size of a tensor when performing a convolution, often one wishes to reduce the size of the output. One way of doing this is by incorporating stride in the convolution operation.

Stride

Stride is a hyperparameter $s \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by "skipping" over s inputs when sliding the kernel matrix across the input. By default, stride is assumed to be $s = 1$. Increasing stride is one way of substantially decreasing the size of the output feature map.

(Stride $s = 2$)



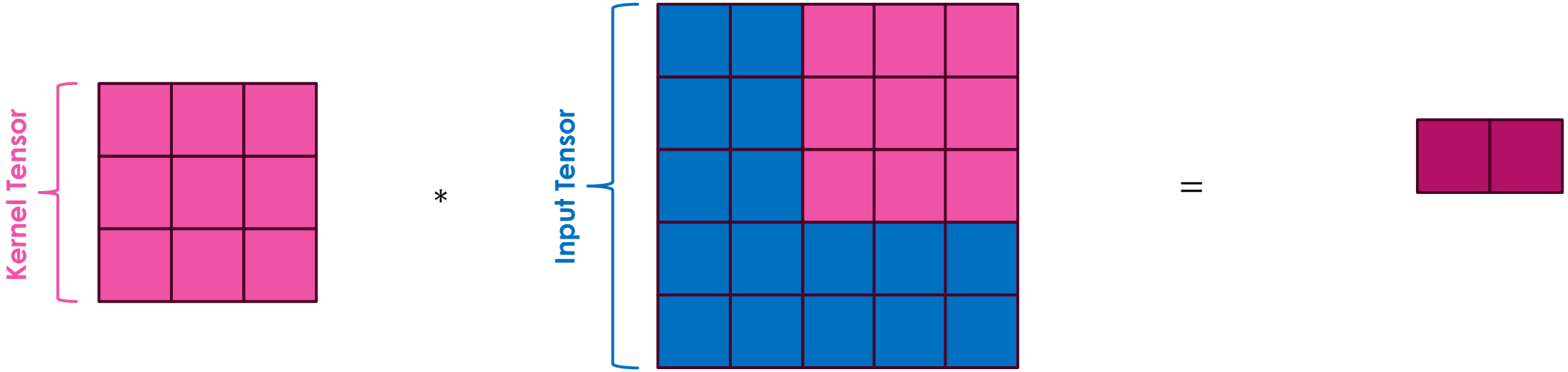
Stride

Opposing the idea of trying to preserve the size of a tensor when performing a convolution, often one wishes to reduce the size of the output. One way of doing this is by incorporating stride in the convolution operation.

Stride

Stride is a hyperparameter $s \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by "skipping" over s inputs when sliding the kernel matrix across the input. By default, stride is assumed to be $s = 1$. Increasing stride is one way of substantially decreasing the size of the output feature map.

(Stride $s = 2$)



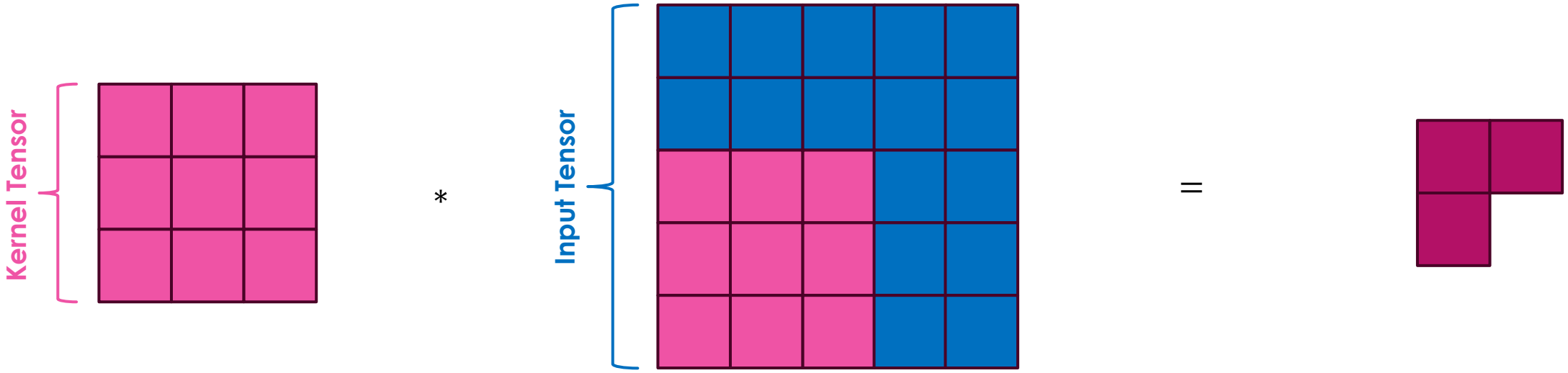
Stride

Opposing the idea of trying to preserve the size of a tensor when performing a convolution, often one wishes to reduce the size of the output. One way of doing this is by incorporating stride in the convolution operation.

Stride

Stride is a hyperparameter $s \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by "skipping" over s inputs when sliding the kernel matrix across the input. By default, stride is assumed to be $s = 1$. Increasing stride is one way of substantially decreasing the size of the output feature map.

(Stride $s = 2$)



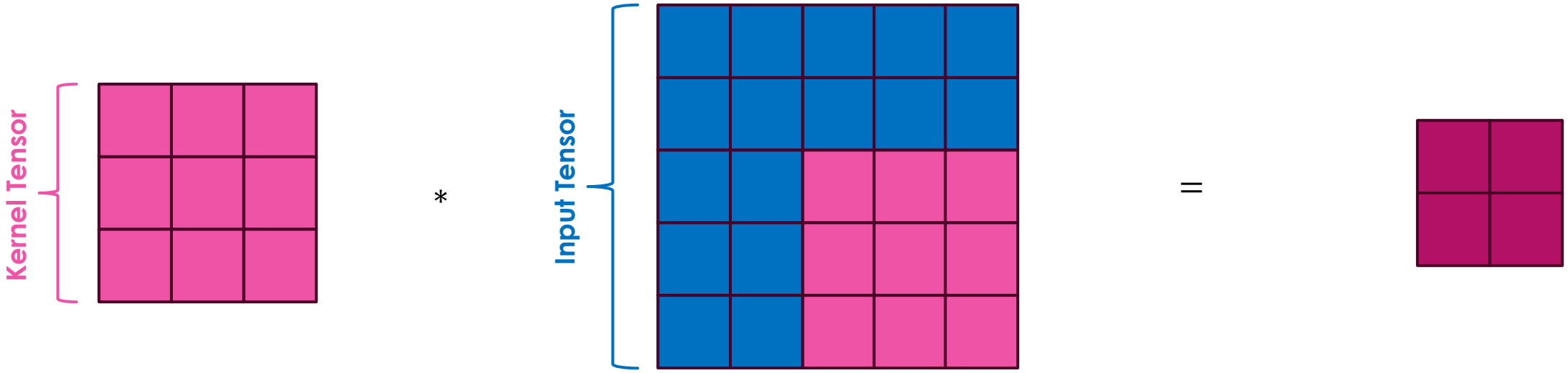
Stride

Opposing the idea of trying to preserve the size of a tensor when performing a convolution, often one wishes to reduce the size of the output. One way of doing this is by incorporating stride in the convolution operation.

Stride

Stride is a hyperparameter $s \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by "skipping" over s inputs when sliding the kernel matrix across the input. By default, stride is assumed to be $s = 1$. Increasing stride is one way of substantially decreasing the size of the output feature map.

(Stride $s = 2$)

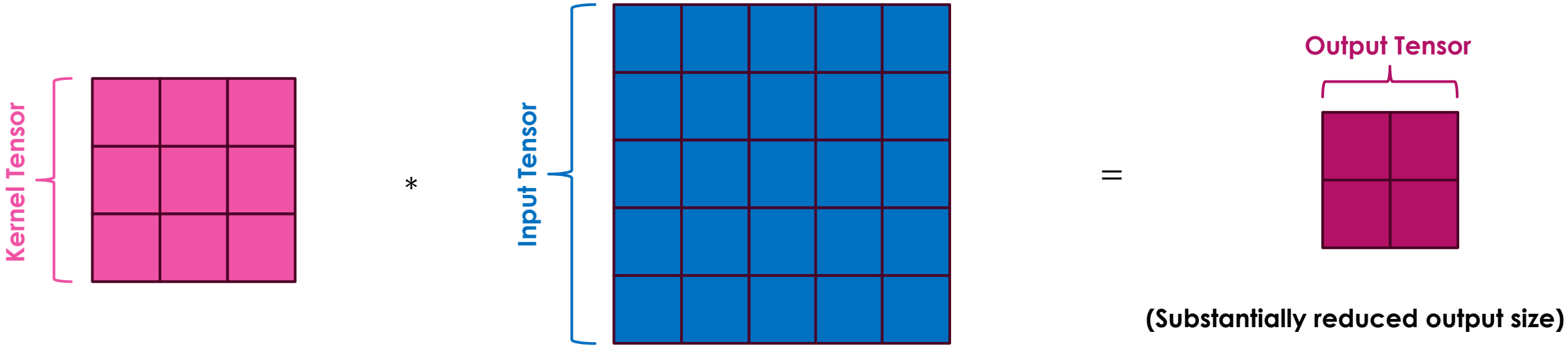


Stride

Opposing the idea of trying to preserve the size of a tensor when performing a convolution, often one wishes to reduce the size of the output. One way of doing this is by incorporating stride in the convolution operation.

Stride

Stride is a hyperparameter $s \in \mathbb{N}$ that can be incorporated in a convolution operation, and it acts by "skipping" over s inputs when sliding the kernel matrix across the input. By default, stride is assumed to be $s = 1$. Increasing stride is one way of substantially decreasing the size of the output feature map.



Generalized Convolution Operations

Now that we have defined the concepts of padding and stride for convolutions, we can restate our definition of the convolution operation in a general way that incorporates information of these two hyperparameters. The resulting definition is the typical form of the convolution that is utilized in practice (i.e., PyTorch, etc.).

Generalized Convolution Operation (2×2)

For the 2-dimensional case, when given an input matrix $I \in \mathbb{R}^{h \times w}$, a convolution "**filter**" $K \in \mathbb{R}^{\hat{h} \times \hat{w}}$, along with some padding $p \in \mathbb{N}$ added rows & columns to the edges of I and/or utilizing a stride of $s \in \mathbb{N}$, then the convolution operation output is denoted by the matrix $C \in \mathbb{R}^{\tilde{h} \times \tilde{w}}$ with elements $C_{i,j}$ that are computed by the convolution operation

$$C_{i,j} = \sum_{\hat{\imath}=1}^{\hat{h}} \sum_{\hat{\jmath}=1}^{\hat{w}} K_{\hat{\imath},\hat{\jmath}} I_{s \cdot i + \hat{\imath},\, s \cdot j + \hat{\jmath}} \,.$$

This is simply the generalized form of a convolution operation that incorporates padding and stride as possible hyperparameters. Further, the entire operation can still be denoted by

$$C = I * K.$$

Lastly, notice that the dimensions of the resulting feature map C is $\tilde{h} \times \tilde{w}$, where

$$\tilde{h} := \left\lfloor \frac{h + 2p - \hat{h}}{s} \right\rfloor + 1 \qquad \text{and} \qquad \tilde{w} := \left\lfloor \frac{w + 2p - \hat{w}}{s} \right\rfloor + 1.$$

The output C can then be fed through an activation function $g(\cdot)$ to learn non-linear patterns.

Pooling Operations

Another way of reducing the size of the feature map (and more commonly used than stride) is to utilize pooling layers. It bears mention that these are **not learnable layers**.

Pooling Operations

Pooling operations are another way of reducing the size of a feature map by localizing and condensing information of the regions of an input tensor. The two most common forms of pooling are as follows:

- **Max Pooling:** Returns the maximum value among all entries of a designated "window" (defined by the dimension of the kernel) on an input tensor.
- **Average Pooling:** Returns the average value among all entries within a designated "window" (defined by the dimension of the kernel) on an input tensor.

Max Pooling
(Stride $s = 2$, Kernel size of 2×2)

Input Tensor

2	0	1	1
0	−1	2	2
−2	3	0	0
0	1	−3	2

Average Pooling
(Stride $s = 2$, Kernel size of 2×2)

Input Tensor

2	0	1	1
0	−1	2	2
−2	3	0	0
0	1	−3	2

Pooling Operations

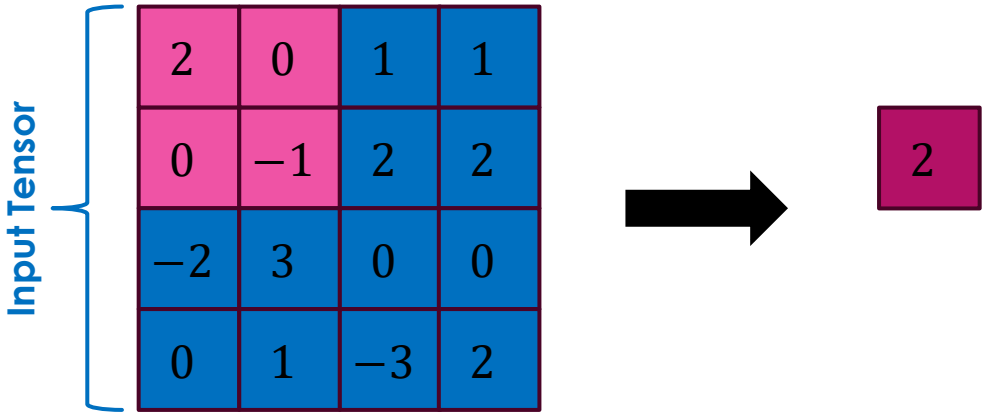
Another way of reducing the size of the feature map (and more commonly used than stride) is to utilize pooling layers. It bears mention that these are **not learnable layers**.

Pooling Operations

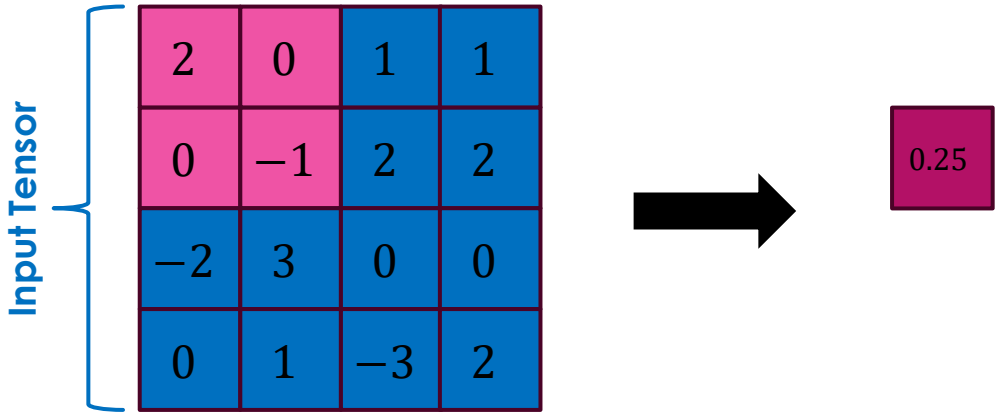
Pooling operations are another way of reducing the size of a feature map by localizing and condensing information of the regions of an input tensor. The two most common forms of pooling are as follows:

- **Max Pooling:** Returns the maximum value among all entries of a designated "window" (defined by the dimension of the kernel) on an input tensor.
- **Average Pooling:** Returns the average value among all entries within a designated "window" (defined by the dimension of the kernel) on an input tensor.

Max Pooling
(Stride $s = 2$, Kernel size of 2×2)



Average Pooling
(Stride $s = 2$, Kernel size of 2×2)



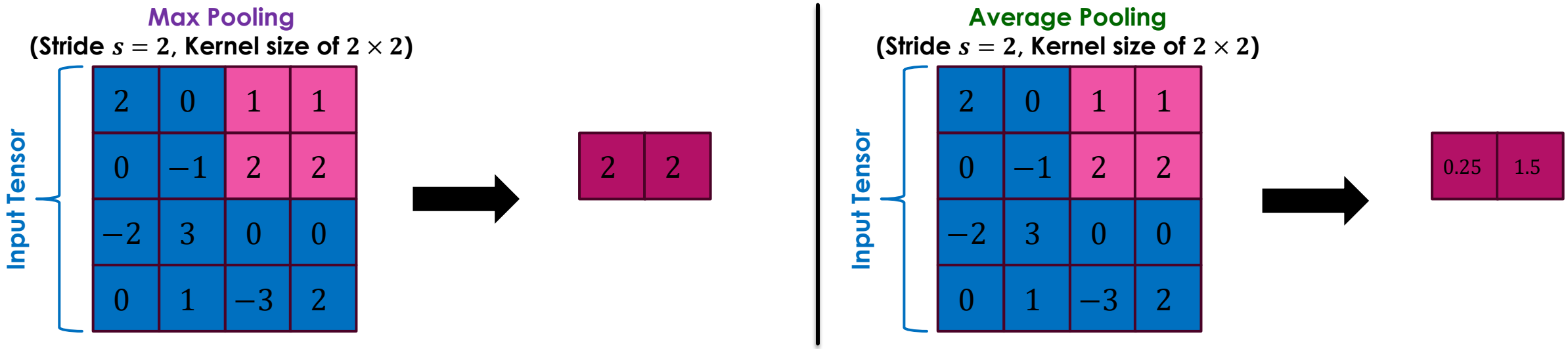
Pooling Operations

Another way of reducing the size of the feature map (and more commonly used than stride) is to utilize pooling layers. It bears mention that these are **not learnable layers**.

Pooling Operations

Pooling operations are another way of reducing the size of a feature map by localizing and condensing information of the regions of an input tensor. The two most common forms of pooling are as follows:

- **Max Pooling:** Returns the maximum value among all entries of a designated "window" (defined by the dimension of the kernel) on an input tensor.
- **Average Pooling:** Returns the average value among all entries within a designated "window" (defined by the dimension of the kernel) on an input tensor.



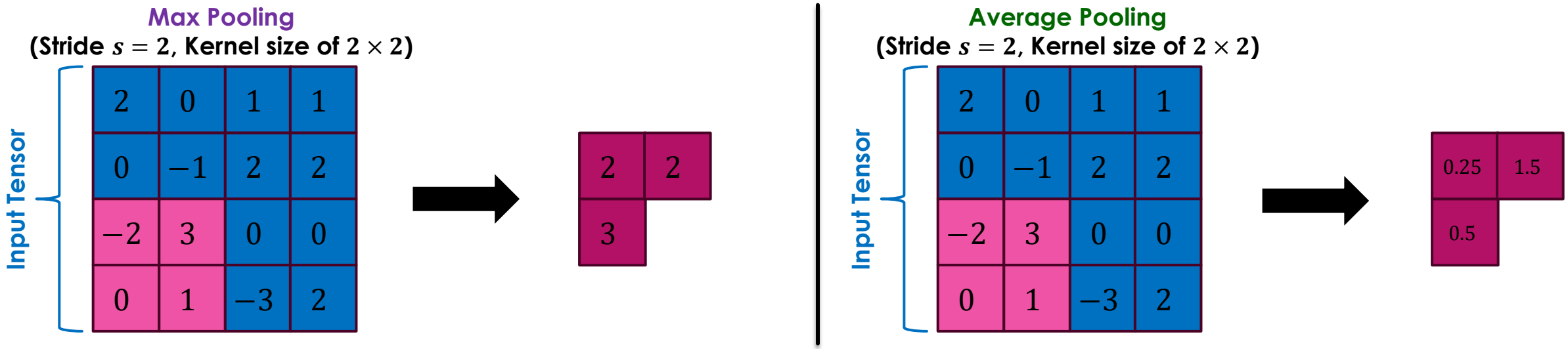
Pooling Operations

Another way of reducing the size of the feature map (and more commonly used than stride) is to utilize pooling layers. It bears mention that these are **not learnable layers**.

Pooling Operations

Pooling operations are another way of reducing the size of a feature map by localizing and condensing information of the regions of an input tensor. The two most common forms of pooling are as follows:

- **Max Pooling:** Returns the maximum value among all entries of a designated "window" (defined by the dimension of the kernel) on an input tensor.
- **Average Pooling:** Returns the average value among all entries within a designated "window" (defined by the dimension of the kernel) on an input tensor.



Pooling Operations

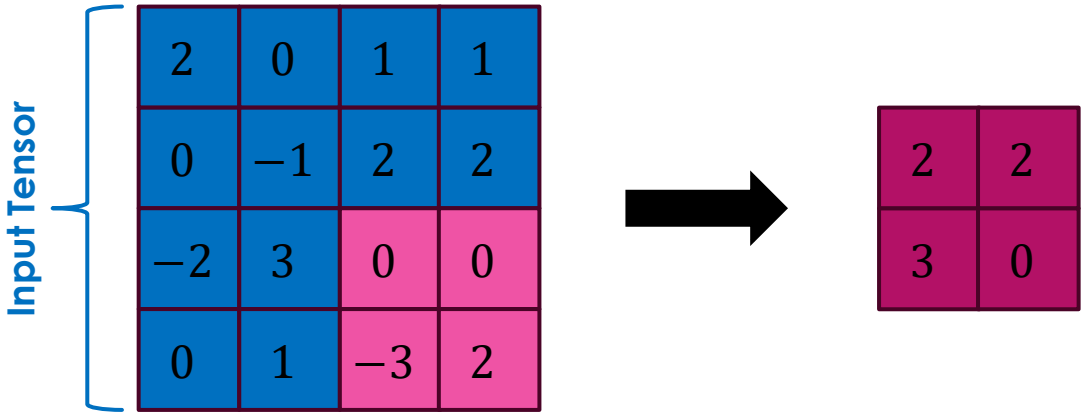
Another way of reducing the size of the feature map (and more commonly used than stride) is to utilize pooling layers. It bears mention that these are **not learnable layers**.

Pooling Operations

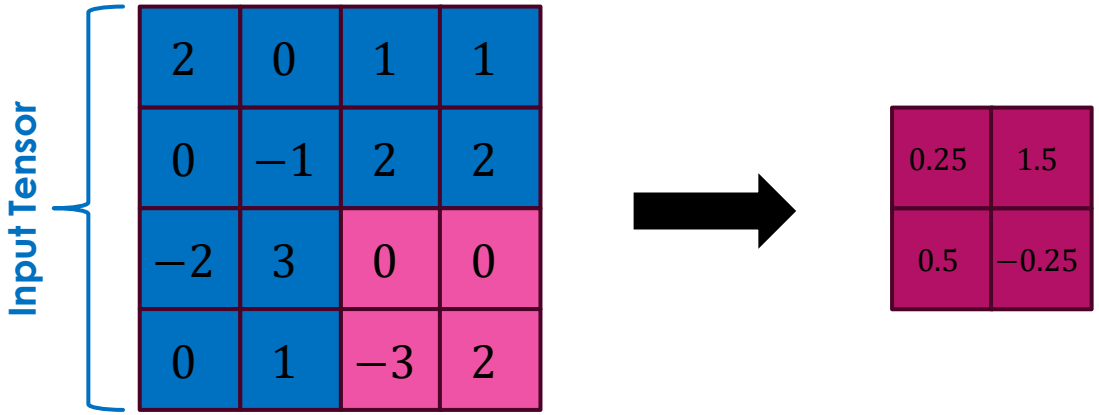
Pooling operations are another way of reducing the size of a feature map by localizing and condensing information of the regions of an input tensor. The two most common forms of pooling are as follows:

- **Max Pooling:** Returns the maximum value among all entries of a designated "window" (defined by the dimension of the kernel) on an input tensor.
- **Average Pooling:** Returns the average value among all entries within a designated "window" (defined by the dimension of the kernel) on an input tensor.

Max Pooling
(Stride $s = 2$, Kernel size of 2×2)



Average Pooling
(Stride $s = 2$, Kernel size of 2×2)



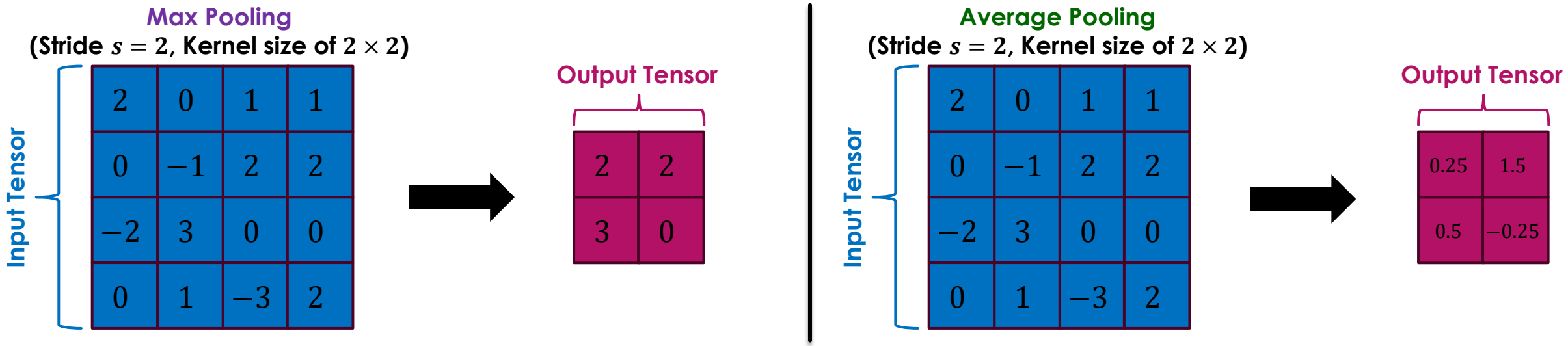
Pooling Operations

Another way of reducing the size of the feature map (and more commonly used than stride) is to utilize pooling layers. It bears mention that these are **not learnable layers**.

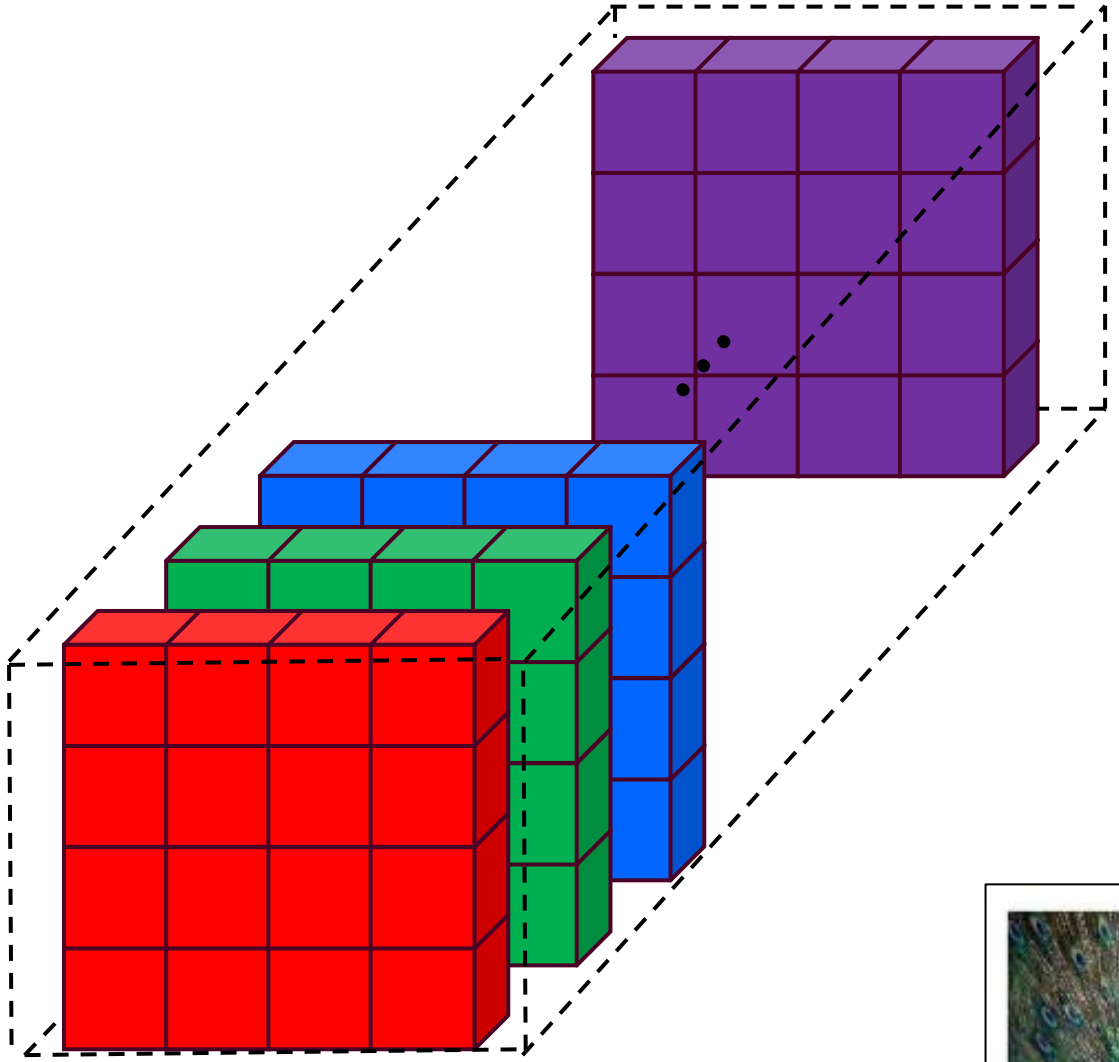
Pooling Operations

Pooling operations are another way of reducing the size of a feature map by localizing and condensing information of the regions of an input tensor. The two most common forms of pooling are as follows:

- **Max Pooling:** Returns the maximum value among all entries of a designated "window" (defined by the dimension of the kernel) on an input tensor.
- **Average Pooling:** Returns the average value among all entries within a designated "window" (defined by the dimension of the kernel) on an input tensor.



Colors and Channels



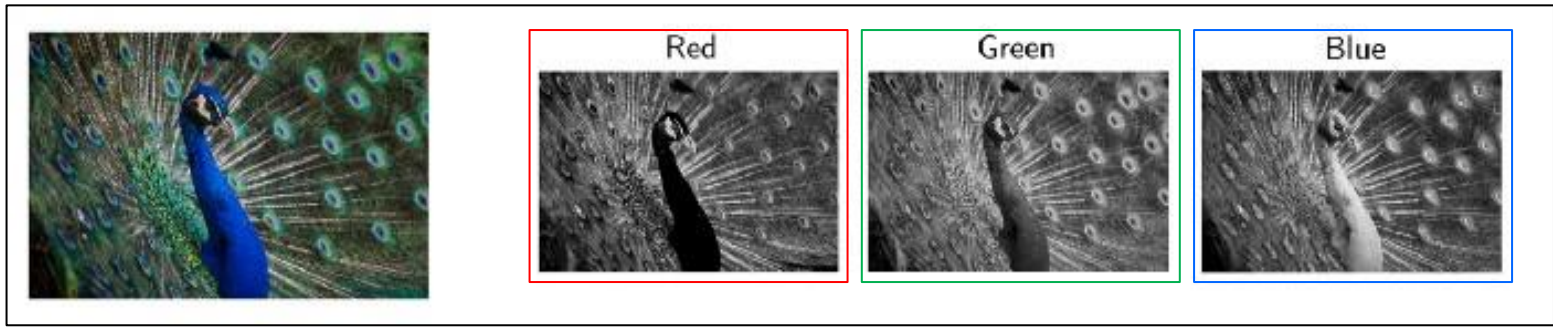
Channels

Up until this point, we have been discussing convolution and pooling operations in the context of a single input matrix (a single channel). However, in most applications there are typically several input **channels**, which correspond to different things. Here are some typical examples of different input channels:

- **(1 channel):** A single matrix that typically indicates a **greyscale** image.
- **(3 channel):** A tensor (or a collection of three matrices) that typically indicates a **full-color** image, where each of the channels (dimension of the tensor) represents one of the three main colors RGB.

However, it bears mentioning that although color is one thing that channels can represent, they can also take on different meanings. Typically, the number of channels only increases as you go deeper into a network. It may be more intuitive to think of a channel as **one of many "views"** of some image, that when combined given the original image. In this way, a channel can define other characteristics of an image other than color.

Illustration of the three standard color channels (R,G,B) (*See References*)

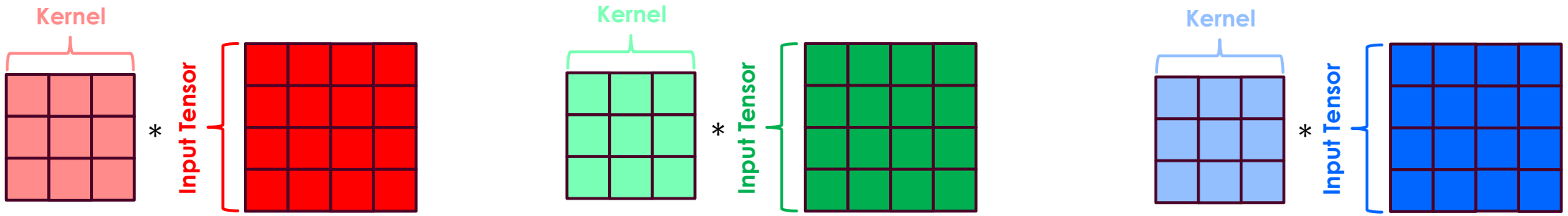


Filters

Filters

A **filter** is simply a **collection of kernel matrices**, where each kernel corresponds to a single input channel in the layer. Thus, each filter is used to perform its own convolution operation and will yield a single output matrix. Similarly, the number of output channels for a layer is equivalent to the number of filters that are used in that layer. It bears mentioning that **one can designate more than one kernel per channel**, which will lead to more output channels. In the 1-channel case, the terms "kernel" and "filter" are interchangeable.

- **Example:** Suppose you have a one input channel of $\mathbb{R}^{n \times m}$. You can obtain an output tensor of $\mathbb{R}^{t \times n \times m}$ by utilizing t kernels.

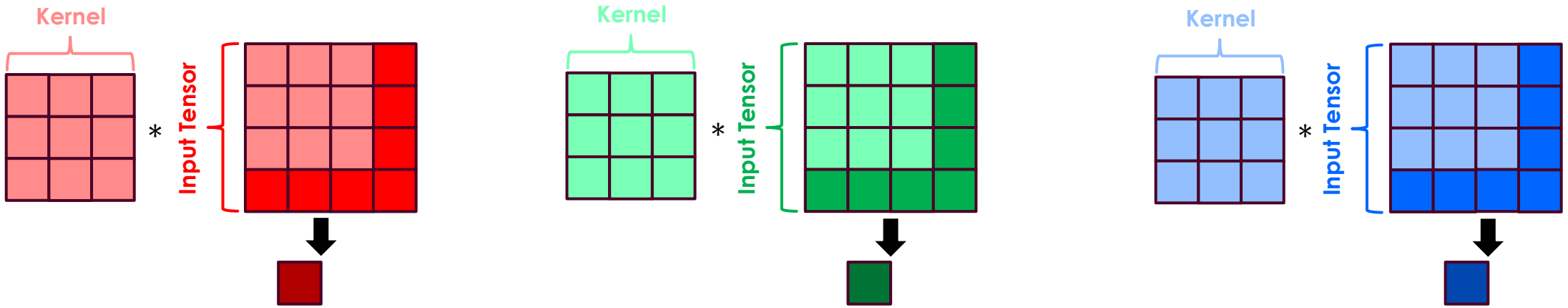


Filters

Filters

A **filter** is simply a **collection of kernel matrices**, where each kernel corresponds to a single input channel in the layer. Thus, each filter is used to perform its own convolution operation and will yield a single output matrix. Similarly, the number of output channels for a layer is equivalent to the number of filters that are used in that layer. It bears mentioning that **one can designate more than one kernel per channel**, which will lead to more output channels. In the 1-channel case, the terms "kernel" and "filter" are interchangeable.

- **Example:** Suppose you have a one input channel of $\mathbb{R}^{n \times m}$. You can obtain an output tensor of $\mathbb{R}^{t \times n \times m}$ by utilizing $t \in \mathbb{N}$ kernels.

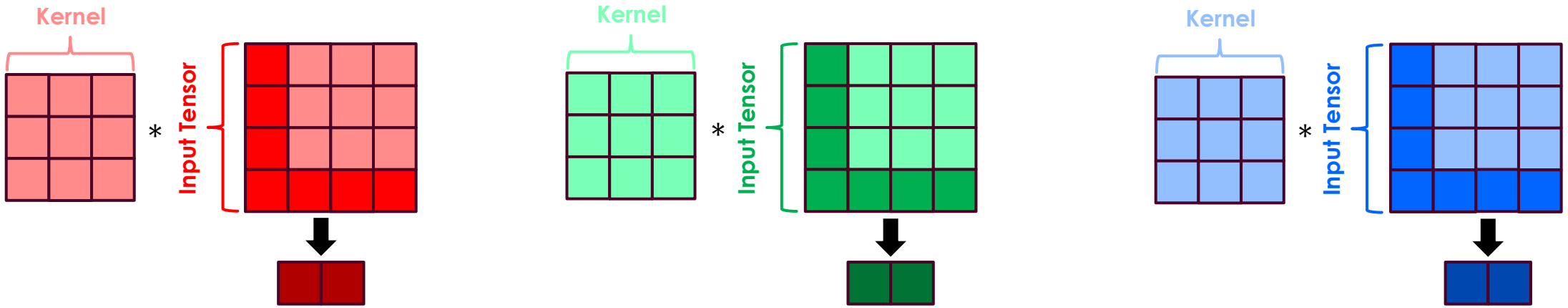


Filters

Filters

A **filter** is simply a **collection of kernel matrices**, where each kernel corresponds to a single input channel in the layer. Thus, each filter is used to perform its own convolution operation and will yield a single output matrix. Similarly, the number of output channels for a layer is equivalent to the number of filters that are used in that layer. It bears mentioning that **one can designate more than one kernel per channel**, which will lead to more output channels. In the 1-channel case, the terms "kernel" and "filter" are interchangeable.

- **Example:** Suppose you have a one input channel of $\mathbb{R}^{n \times m}$. You can obtain an output tensor of $\mathbb{R}^{t \times n \times m}$ by utilizing $t \in \mathbb{N}$ kernels.

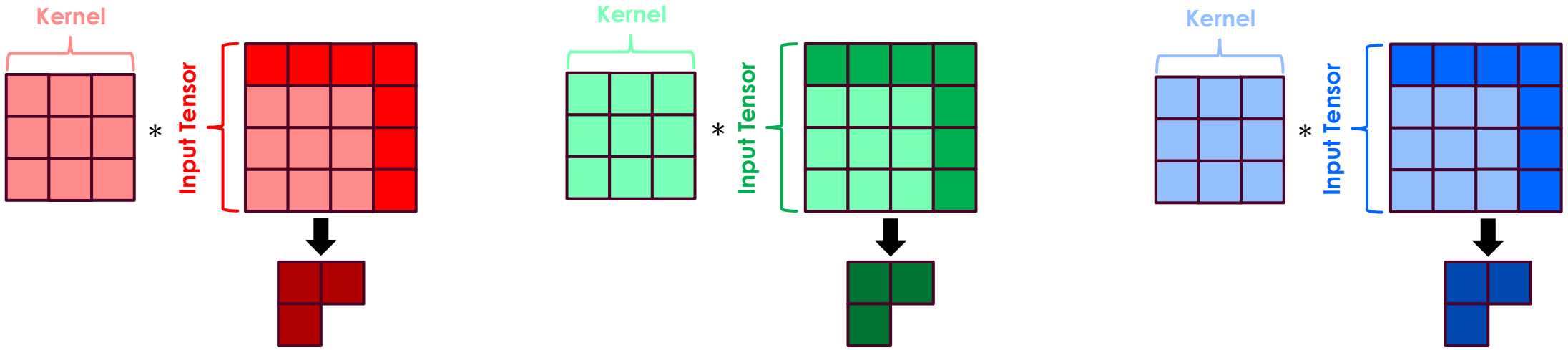


Filters

Filters

A **filter** is simply a **collection of kernel matrices**, where each kernel corresponds to a single input channel in the layer. Thus, each filter is used to perform its own convolution operation and will yield a single output matrix. Similarly, the number of output channels for a layer is equivalent to the number of filters that are used in that layer. It bears mentioning that **one can designate more than one kernel per channel**, which will lead to more output channels. In the 1-channel case, the terms "kernel" and "filter" are interchangeable.

- **Example:** Suppose you have a one input channel of $\mathbb{R}^{n \times m}$. You can obtain an output tensor of $\mathbb{R}^{t \times n \times m}$ by utilizing $t \in \mathbb{N}$ kernels.

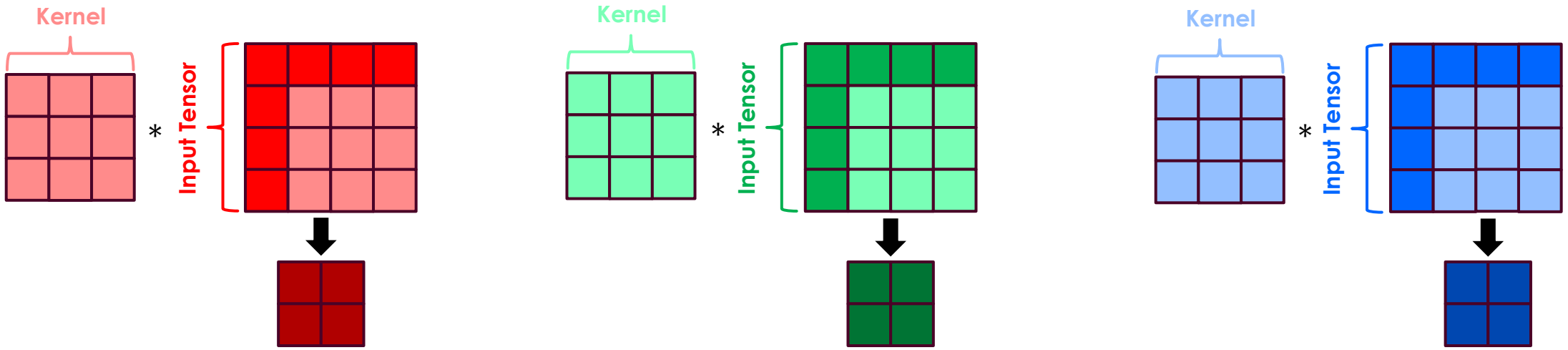


Filters

Filters

A **filter** is simply a **collection of kernel matrices**, where each kernel corresponds to a single input channel in the layer. Thus, each filter is used to perform its own convolution operation and will yield a single output matrix. Similarly, the number of output channels for a layer is equivalent to the number of filters that are used in that layer. It bears mentioning that **one can designate more than one kernel per channel**, which will lead to more output channels. In the 1-channel case, the terms "kernel" and "filter" are interchangeable.

- **Example:** Suppose you have a one input channel of $\mathbb{R}^{n \times m}$. You can obtain an output tensor of $\mathbb{R}^{t \times n \times m}$ by utilizing $t \in \mathbb{N}$ kernels.

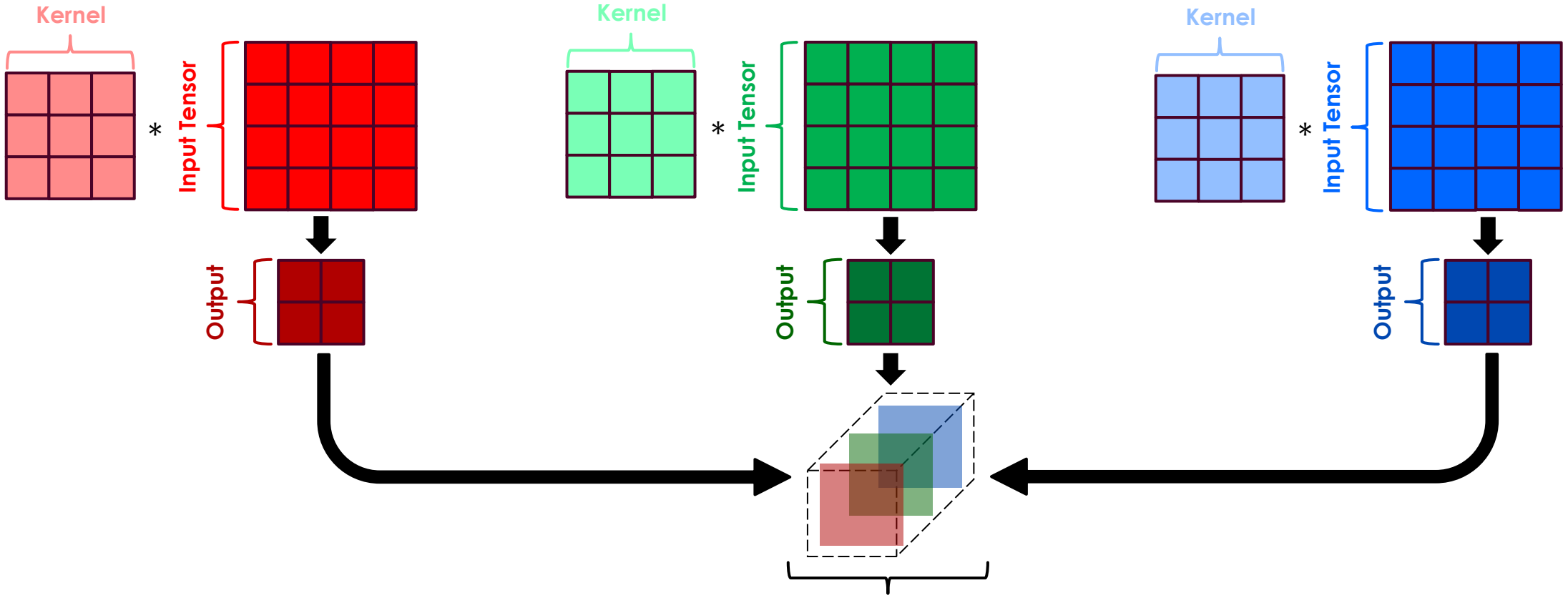


Filters

Filters

A **filter** is simply a **collection of kernel matrices**, where each kernel corresponds to a single input channel in the layer. Thus, each filter is used to perform its own convolution operation and will yield a single output matrix. Similarly, the number of output channels for a layer is equivalent to the number of filters that are used in that layer. It bears mentioning that **one can designate more than one kernel per channel**, which will lead to more output channels. In the 1-channel case, the terms "kernel" and "filter" are interchangeable.

- **Example:** Suppose you have a one input channel of $\mathbb{R}^{n \times m}$. You can obtain an output tensor of $\mathbb{R}^{t \times n \times m}$ by utilizing $t \in \mathbb{N}$ kernels.



Interpreting the Layers of CNNs

Some Notes on Convolutional Layers

- A **convolution** operation is a **linear operation** as it is essentially computing the **dot-product** of the input window and the kernel matrix (think about flattening out the matrices into vectors and then computing the dot-product). By the nature of the dot-product, if the output "pixel" of a convolution is large, this means that that region of the image was very "**similar**" to the kernel matrix, and hence indicates that the pattern that the kernel matrix is looking for is present in that location of the image.
- This linear nature of the convolution operation is why we still utilize the same types of **activation functions** that are used in standard MLPs: to **introduce non-linearity**.
- In the earliest layers of a network, the convolution filters typically act as identifying simple patterns in the image (e.g., vertical lines, horizontal lines, etc.) and then deeper into the network can identify more complex non-linear shapes (round objects, waves, overall non-linear patterns, etc.).
- The output of convolution filters is also referred to as **feature maps**. This is essentially how convolutions function: to produce relevant features (which represent different views of different regions of an image) that will ultimately be used as inputs into feed-forward layers of the network (at the end).

Some Notes on Pooling Layers

- Pooling operations can be thought of as "zooming out" on an image, effectively reducing the spatial dimensions of a feature map by summarizing regions.
- **Max pooling** essentially summarizes a region of colors by outputting the most intense pixel value, whereas **average pooling** does this by outputting the average pixel intensity within a region.
- Reducing the dimensions of a feature map is useful to do because it lowers computational cost, and it helps prevent overfitting. This also helps make the network more invariant to small translations or distortions in the inputs.

Visualizing the Layer Outputs of a CNN

Original Input Image to the CNN

Conv. Layer with 6 Kernel filters (6 output channels)

Max Pooling Layer

Conv. Layer with 16 Kernel filters (16 output channels)

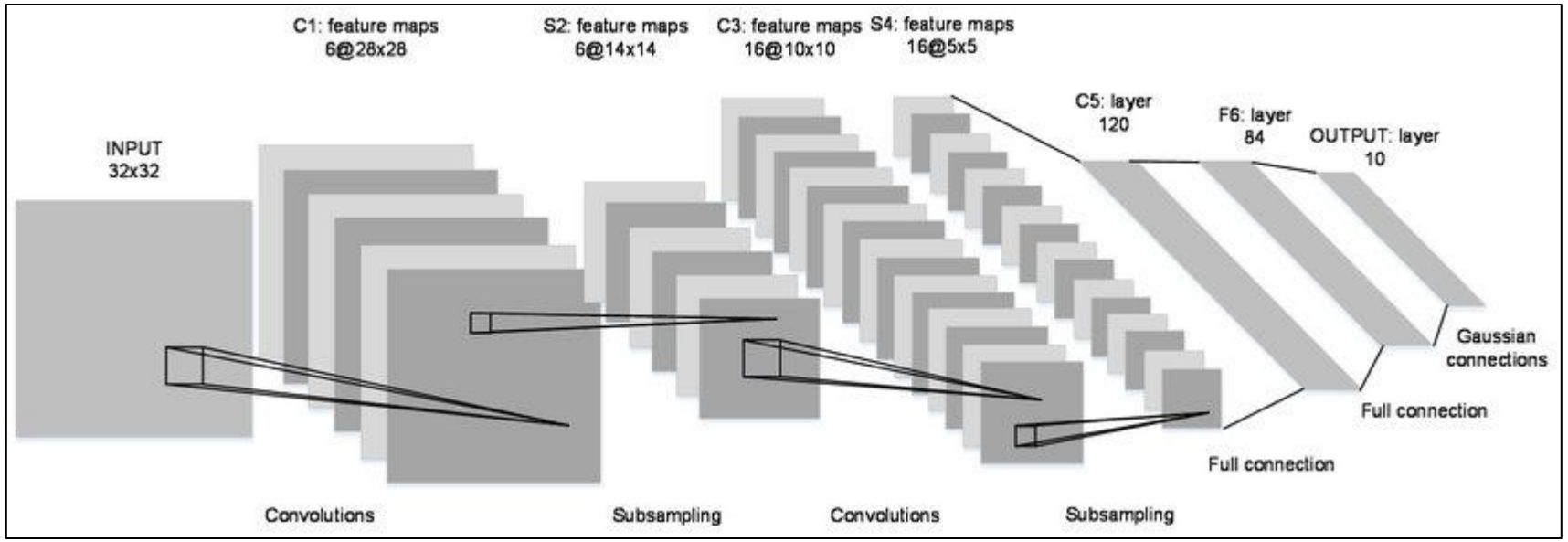
Average Pooling Layer

Common Convolutional Architectures

The LeNet Architecture

This was one of the first successful convolutional neural network architectures originally developed in the late 1980s and was formally proposed in the seminal paper "*Gradient-Based Learning Applied to Document Recognition*" by LeCun, Bottou, Bengio, and Haffner in (and leading up to) 1998. This architecture was designed to identify images of hand-written digits, and it demonstrated the efficacy of using convolution filters, pooling layers, as well as backpropagation for learning image recognition tasks. The principles laid out in this architecture served as the foundation for modern CNNs and drove the direction of computer vision today.

The LeNet-5 architecture (*See References*)

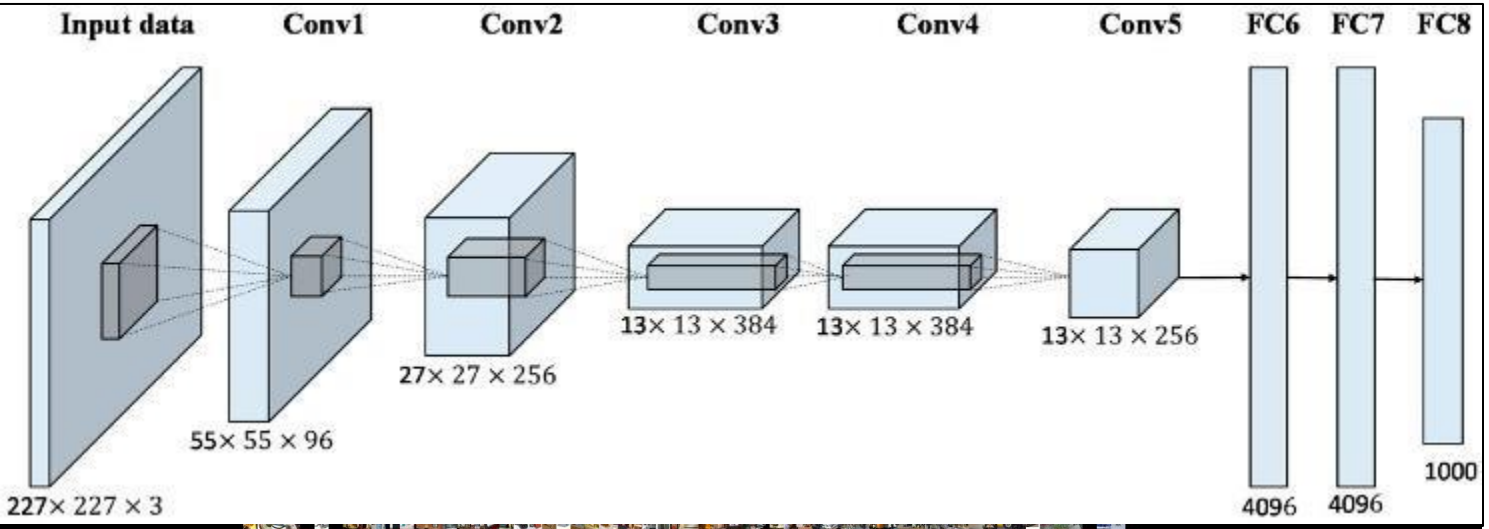


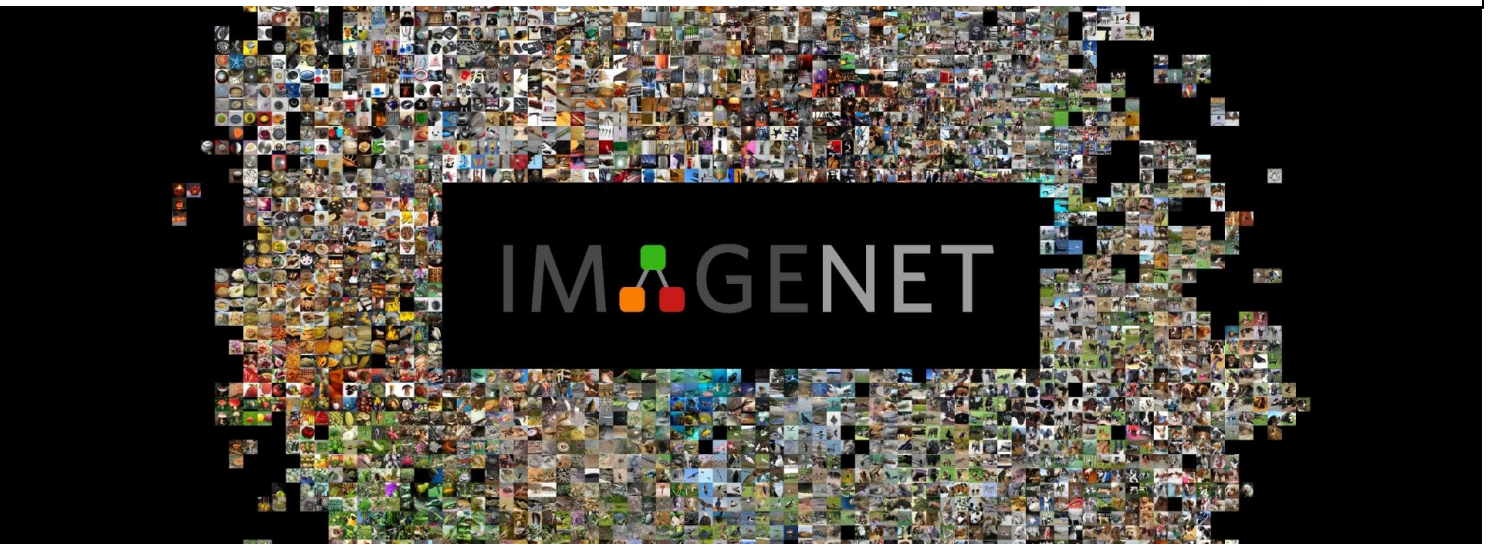
The MNIST dataset of hind-written digits

See References

Common Convolutional Architectures

The AlexNet architecture (*See References*)





See References

The AlexNet Architecture

The AlexNet architecture was proposed in the famous paper "ImageNet Classification with Deep Convolutional Neural Networks" written by Krizhevsky, Sutskever, and Hinton in 2012. This CNN was developed for the ImageNet dataset.

- **ImageNet** is a classic benchmarking dataset that is often used to evaluate the performance of advanced neural architectures. It consists of more than **14 million images** with labeled target classes as well as having more than **20,000 target class categories**.

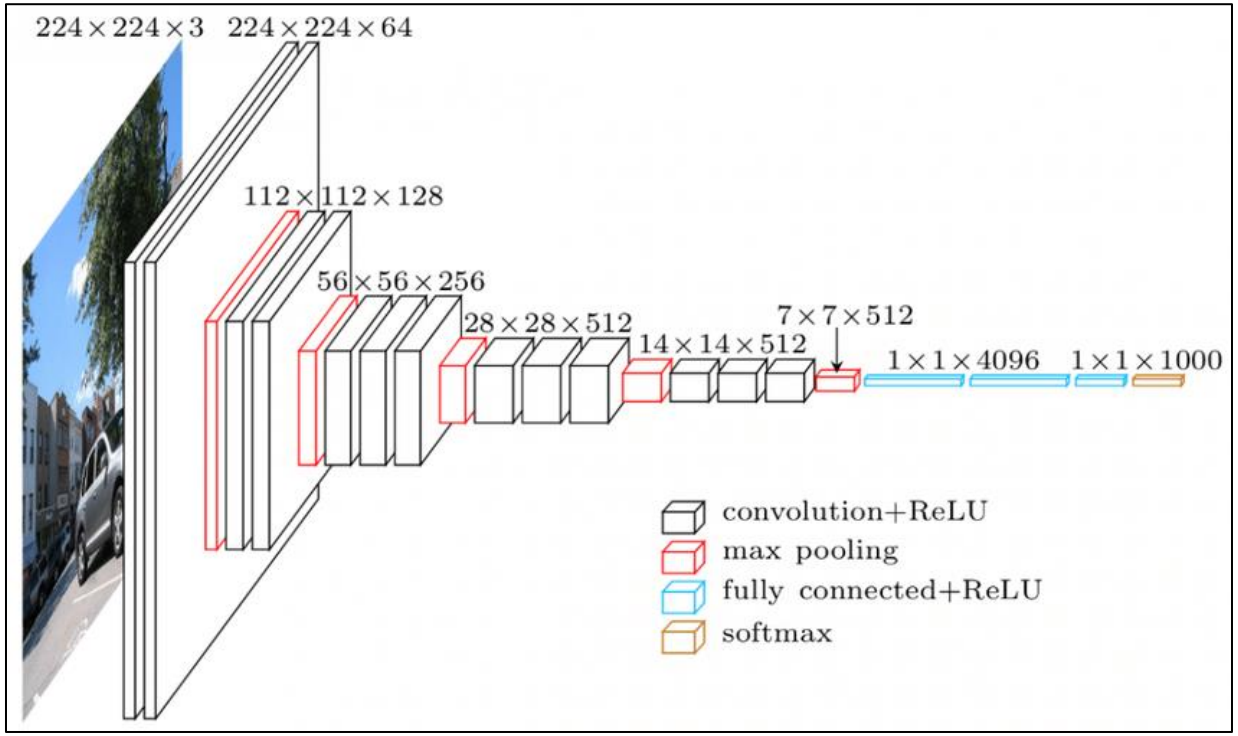
AlexNet revolutionized the field of deep learning by demonstrating the power of CNNs on large-scale datasets. It utilized ReLU activation functions and parallel processing from GPUs to achieve the top accuracy (at the time) in image classification. This marked the beginning of the boom in deep neural networks.

Common Convolutional Architectures

The Visual Geometry Group (VGG) Architecture

The general VGG architecture was developed in the paper "*Very Deep Convolutional Networks for Large-Scale Image Recognition*" by Simonyan and Zisserman in 2014. In terms of novel contributions, the VGG architecture was not necessarily the most revolutionary. However, it did demonstrate that by using simple 3x3 kernel convolution matrices for feature extraction paired with simply making the network deeper was able to yield better performances that had been seen prior. Its simplicity and uniformity, combined with its depth, was able to yield significant improvements on image classification tasks.

An example of a VGG-16 architecture (*See References*)

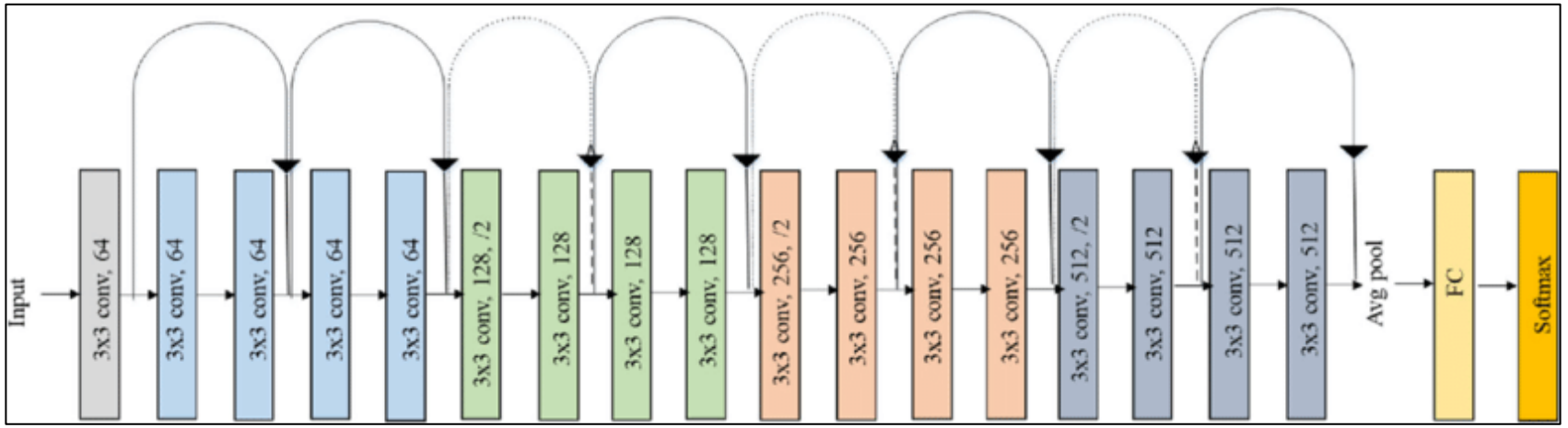


Common Convolutional Architectures

The Residual Network (ResNet) Architecture

The ResNet architecture was originally proposed in the paper "*Deep Residual Learning for Image Recognition*" written by K. He, X. Zhang, S. Ren, and J. Sun in 2015. The ResNet architecture introduced the concept of "**skip**" connections (otherwise known as **residual** connections) to help **prevent the vanishing gradient problem**. This simply amounted to allowing the inputs of the earlier layers also serve as inputs into the deeper layers; this way, when performing backpropagation, the gradients with respect to the earlier layers would not have vanished by the time they were computed. This allowed networks to be much deeper, which subsequently yielded an increase in performance on a variety of benchmarking datasets. Today, ResNet architectures are typical "go-to" models when performing vision tasks, and the ideas of the architecture have become a cornerstone in deep learning which have impacted many future architectures.

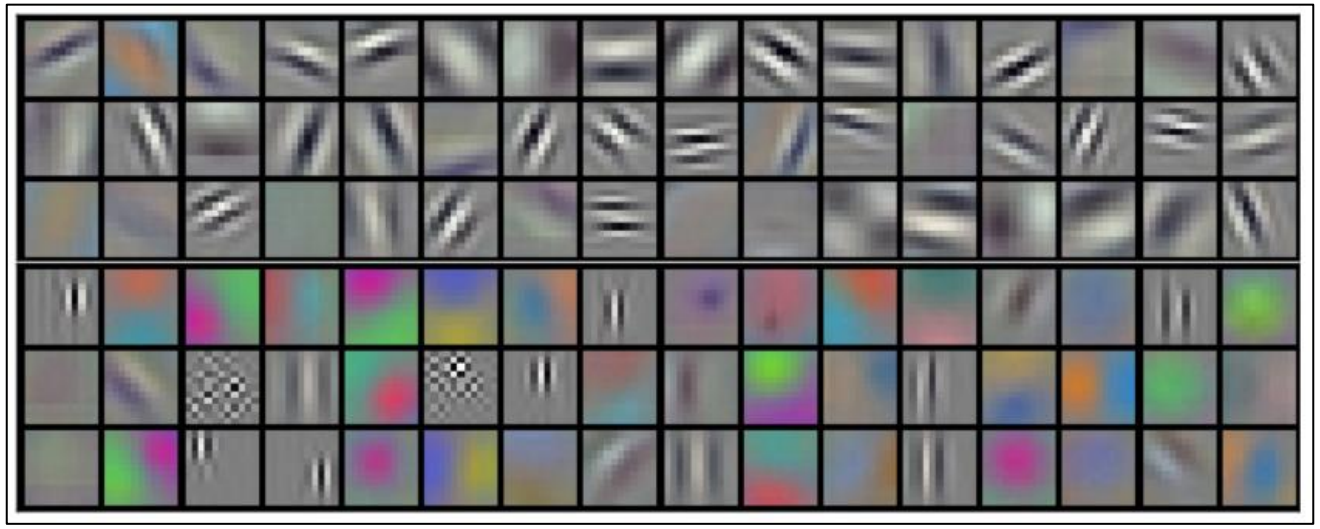
The original ResNet-18 architecture (*See References*)

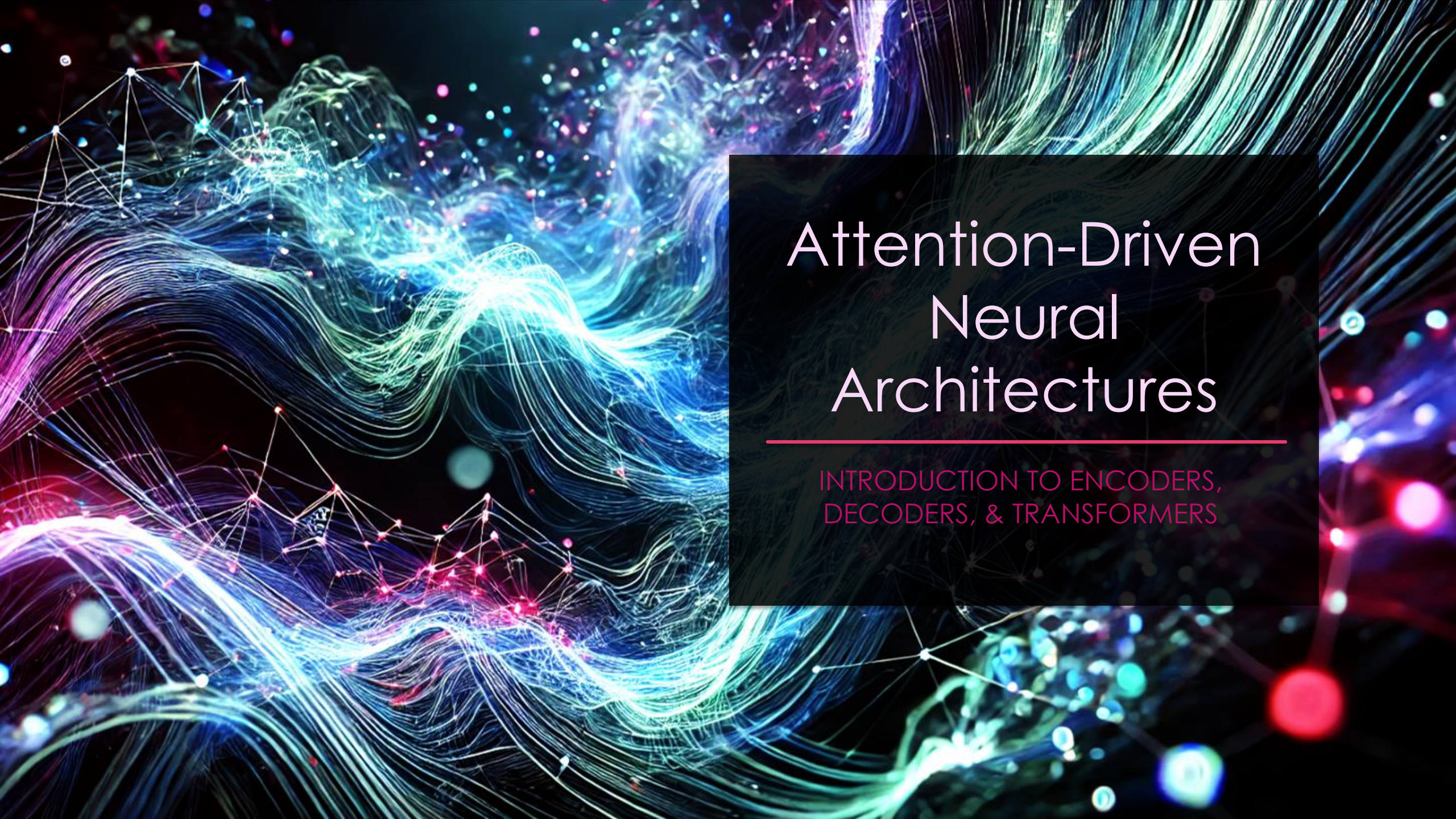


Convolution Feature Maps for MLPs

- In all the CNN architectures we have seen (and all CNNs as a whole), there are some number of convolution / pooling layers that output **feature maps** (in the form of a tensor with many channels), followed by ultimately "**flattening**" them out and feeding them into a multi-layer perceptron.
- How is this different from simply taking an input image, flattening it out into a single large vector of all its pixels, and then feeding it into an MLP?
- The **answer** is that the layers of convolution and pooling operations capture "**localized**" (or **spatial**) information by "**zooming out**" of the image. Thus, each pixel in the resulting output tensor is a value that corresponds to some larger region of the original image as opposed to simply being a single pixel of the original image.
- This is why CNNs are so much better for image tasks. **CNNs preserve spatial information of the image**.

Feature maps obtained from the first convolution layer of AlexNet (*See References*)




Attention-Driven Neural Architectures
INTRODUCTION TO ENCODERS, DECODERS, & TRANSFORMERS

Overview of Transformers

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

†Work performed while at Google Brain.

‡Work performed while at Google Research.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

Transformer Architectures

Proposed in the paper "***Attention is All You Need***" that was published in 2017, the general transformer architecture for neural networks has **revolutionized** the field of deep learning by replacing recurrent networks and convolutional networks with the "**attention**" mechanism for sequence modeling. At a high-level view, attention "transforms" some input data into different embedding spaces, where similarities within the input data impact the locations of their numeric locations in the embedded space.

- A transformer network then essentially consists of alternating between generating embeddings and feeding the output into Multi-Layer Perceptrons to identify nonlinear patterns within the embedded spaces. The transformer's **parallelizable architecture** significantly improved computational efficiency and scalability, which has allowed it to achieve state-of-the-art performance on large array of different kinds of tasks.

Embeddings

- In many machine learning applications and especially in modern ay deep learning, the concept of embedded spaces is of paramount importance.

Embeddings

An embedding is a way of **representing some type of complex data** (such as images, words, or sounds) as a **numerical representation** in some **vector space**. These embeddings are typically representations that "encode" some sort of key features, patterns, or relationships of the original data in a way that reduces its dimensionality. There are several different standard dimensionality reduction techniques such as PCA, TSNE, UMAP, etc., (refer to the Dimensionality Reduction slides). At a fundamental level, good embeddings are vector spaces with spatial regions that **represent some sort of semantic content**.

Illustration of Embedded Spaces

Dataset (Words, Images, etc.)



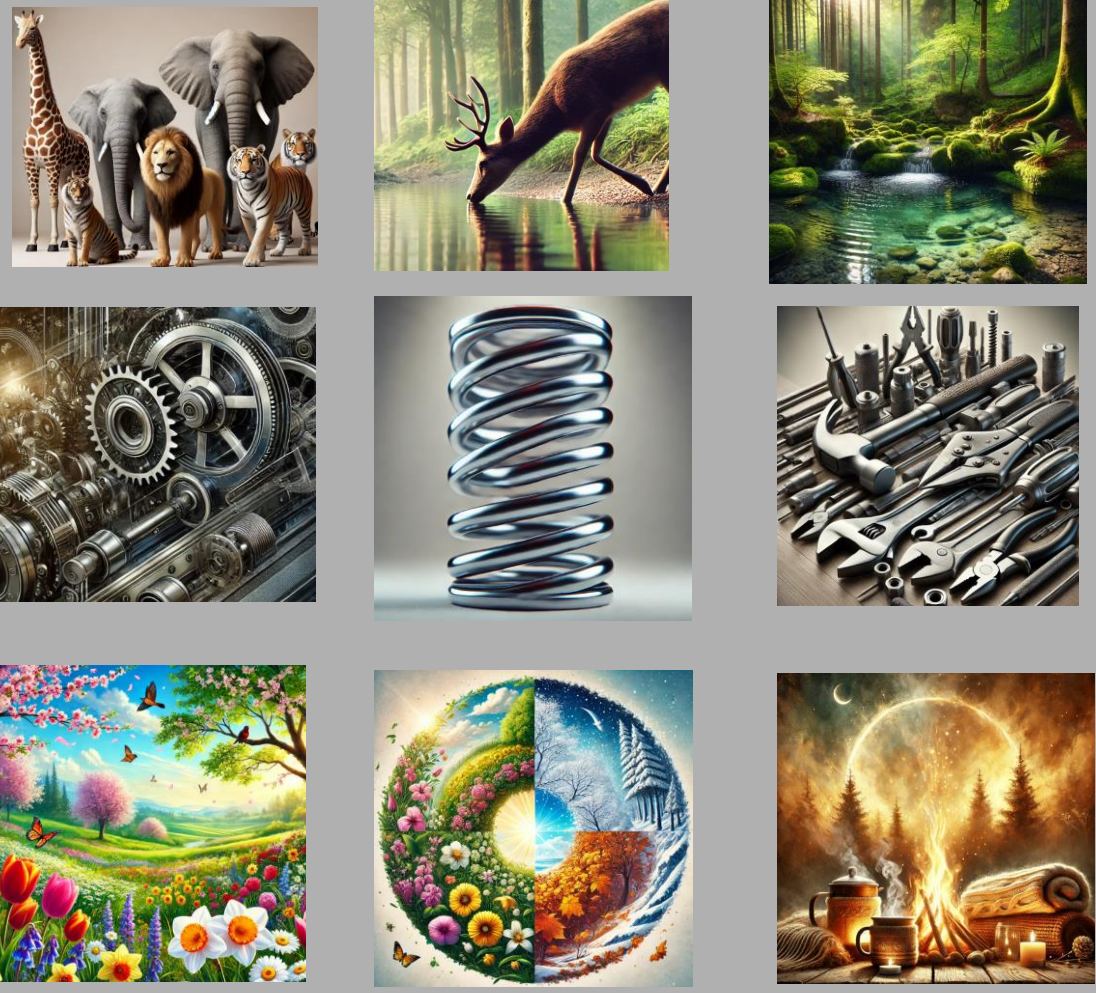






Illustration of Embedded Spaces

Dataset (Words, Images, etc.)



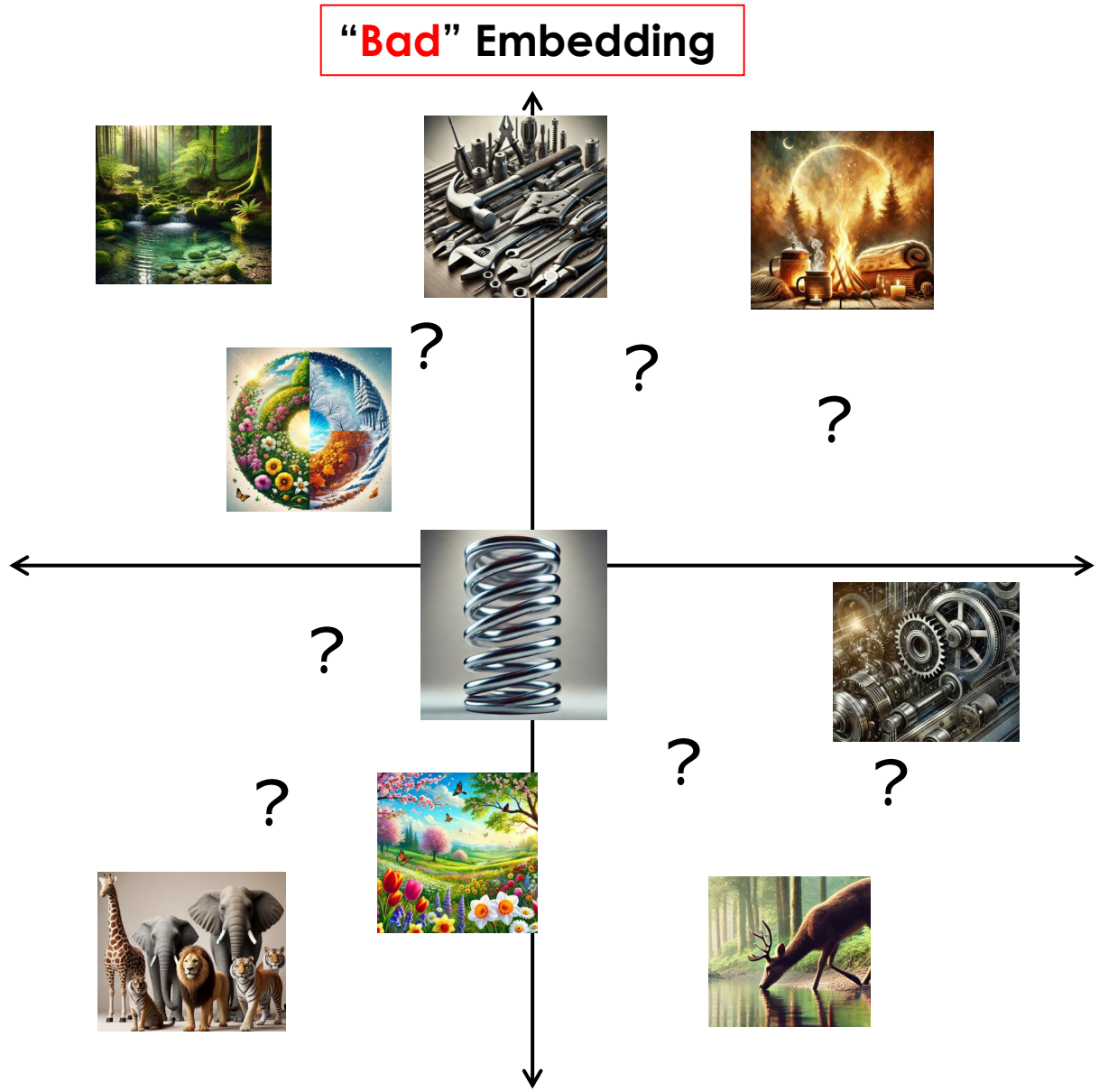


Illustration of Embedded Spaces

Dataset (Words, Images, etc.)



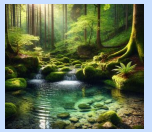






"Good" Embedding





Biological

Weather


Hardware



Similarity Metrics

- Based on our acknowledgement that we want to generate "good" embeddings, how do we define this measure of goodness?
- We begin by utilizing mathematical measures of "similarity" between vectors.

Similarity Scores

Similarity scores are a **mathematical way of measuring how similar or "close" two vectors are**. There are MANY different scores that exist (such as distance metrics like the norms, etc.). However, we list here those that are most-commonly used in DNNs. Naturally, these metrics **revolve around dot-products**, sue to the natural inclusion of the angle and magnitude of the vectors. The larger (positive) the score S, the more similar the vectors are.

- **Dot-Product:** Given two vectors $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$, then the most basic similarity score is simply the dot-product, given by

$$S(a, b) \coloneqq a^T b.$$

- **Cosine-Similarity:** Given two vectors $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$ with an angle θ between them, the cosine-similarity is given by

$$S(a, b) \coloneqq \cos(\theta) = \frac{a^T b}{\|a\| \cdot \|b\|}.$$

Notice that when $\|a\| = \|b\| = 1$ (i.e., the vectors are normalized), then the cosine similarity and the dot-product are equivalent.

- **Scaled Dot-Product:** Given two vectors $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$, the scaled dot-product is given by (this is the similarity score that was used in the original attention mechanism and is useful when the dimension of the vectors is very large)

$$S(a, b) \coloneqq \frac{a^T b}{\sqrt{n}}.$$

Overview of Attention

The Attention Mechanism (Overview & Summary)

The attention mechanism is the fundamental component of the transformer architecture and was proposed in the famous paper "*Attention is All You Need*" that was published in 2017. The **fundamental idea** of what the attention mechanism is doing can be summarized as **performing a series of linear transformations** on an **initial embedding** (of sequential data) to **different embedded spaces**, ending with a **new embedded space** that **adds more "context" to the tokens** within a sequence of data.

Attention can be described as consisting of four stages:

- **(1)** Take an **initial embedding** of some type of complex **sequential data** (images, words, passages, time-series, etc.).
- **(2) Transform** the embedded tokens into a **new embedded space** known as the **query/key space**. This is done by utilizing two different matrices W_Q and W_K (the matrices of query and key weights, respectively). Thus, the tokens are transformed twice, once using the matrix W_Q which will yield a series of "**queries**" (which can be thought of as **asking some form of question** about the context surrounding the token), and then second by using the matrix W_K which will yield a series of "**keys**" (which can be thought of as **answering the question** posed by the queries).
- **(3)** Compute the **similarity scores** between each query vector and every key vector to identify how well each key vector matches (or "answers") the query vectors.
- **(4) Transform** the original embedded tokens into the **value space** by using the matrix of value weights W_V.
- **(5)** Using the similarity scores in (3), **shift** the vectors in the value space in a direction that more closely aligns the query vectors with their similar key vectors. This is done by performing a linear transformation using the value matrix V. This new step uses the similarities that were computed in step (3) by **shifting the tokens in the value space in a direction that more closely aligns with the tokens that achieved high similarity scores in the query/key space**.

Attention – Formal Definition

The Attention Mechanism

Suppose that you have a dataset (or batch of datapoints) $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, where $x^{(i)}$ is some type of complex data (such as images, passages, words, etc.) and $y^{(i)}$ is some target variable, for all $i \in \{1,2,\dots,m\}$. Further, assume that each $x^{(i)}$ has some high-dimensional **embedding** (**token**) vector $E_i \in \mathbb{R}^n$ associated with it. Then, given a "**Query**" matrix $W_Q \in \mathbb{R}^{s\times n}$, a "**Key**" matrix $W_K \in \mathbb{R}^{s\times n}$, and a "Value" matrix $W_V \in \mathbb{R}^{d\times n}$, one can compute the "**Queries**" Q_i, "**Keys**" K_i, and "**Values**" V_i corresponding to all m tokens in the following way:

$$Q_i \coloneqq W_Q E_i \qquad and \qquad K_i \coloneqq W_K E_i \qquad and \qquad V_i \coloneqq W_V E_i, \qquad \forall i \in \{1,2,\dots,m\}.$$

- Notice that the matrices W_Q and W_K map each token E_i into a new space (the **Query/Key Space**); i.e., the resulting $Q_i \in \mathbb{R}^s$ and $K_i \in \mathbb{R}^s$ vectors will be in the same space (which similarity metrics can now be use on). Then, if the similarity score $S(Q_i, K_j)$ between a query Q_i and a key K_j is large, we would say that **the key K_j "attends" to the query Q_i**. Similarly, W_V maps each token E_i into the **Value** space in $\mathbb{R}^d$.

These new query, key, and value vectors are then concatenated into Query, Key, and Value Matrices $Q \in \mathbb{R}^{s\times m}$, $K \in \mathbb{R}^{s\times m}$, and $V \in \mathbb{R}^{d\times m}$ as

$$Q \coloneqq \begin{bmatrix} | & | & & | \\ Q_1 & Q_2 & \cdots & Q_m \\ | & | & & | \end{bmatrix} \quad \text{and} \quad K \coloneqq \begin{bmatrix} | & | & & | \\ K_1 & K_2 & \cdots & K_m \\ | & | & & | \end{bmatrix} \quad \text{and} \quad V \coloneqq \begin{bmatrix} | & | & & | \\ V_1 & V_2 & \cdots & V_m \\ | & | & & | \end{bmatrix}.$$

Therefore, one can **efficiently compute all the similarity scores** (using the scaled dot-product) between <u>each</u> query-key pair by **computing the outer-product** $G \coloneqq (QK^T)/\sqrt{s} \in \mathbb{R}^{m\times m}$, where the (i,j)th element of G denotes the similarity score given by

$$G_{i,j} \coloneqq S(Q_i, K_j) = \frac{Q_i^T K_j}{\sqrt{s}}, \qquad \forall i \in \{1,2,\dots,m\}, \qquad \forall j \in \{1,2,\dots,m\}.$$

The columns of G will then correspond to the queries and the rows will correspond to the keys. Since we don't want the similarity scores to get too large in absolute value, we convert each key corresponding to a single query (i.e., all the keys in the column G_i which correspond to the query Q_i) to **probabilities by utilizing the softmax** function column-wise. Lastly, the value embeddings are shifted in space by the similarity scores, moving a vector that had a high query-key similarity in a direction toward that vector (that corresponded to the key) by multiplying by the value matrix $V^T \in \mathbb{R}^{m\times d}$, yielding the final matrix of **attention embeddings** in $\mathbb{R}^{m\times d}$ given by

$$\text{Attention}(Q,K,V) \coloneqq \text{softmax}\left(\frac{QK^T}{\sqrt{s}}\right)V^T.$$

- The resulting matrix is a collection of the m **value vectors**, each in $\mathbb{R}^d$, that have been spatially shifted to a location closer to the tokens that yielded large similarity scores. This way, the value space **encodes the context within the original sequential data**.

Illustration of Attention for the Word “Spring”

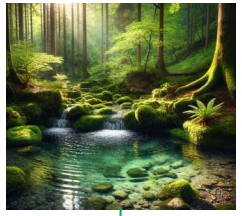
Input Data Sequence

The animals drank from the spring.

$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$

Illustration of Attention for the Word “Spring”





Input Data Sequence

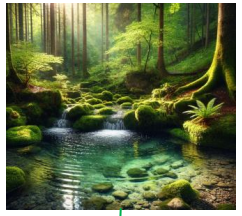
The animals drank from the spring.

$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$

Illustration of Attention for the Word "Spring"

In this example, we will illustrate the attention mechanism with regards to the word "**Spring**".





Input Data Sequence

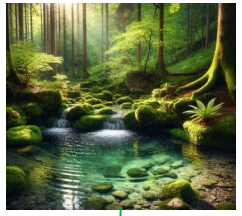
The animals drank from the **spring**.

$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$

Illustration of Attention for the Word "Spring"

In this example, we will illustrate the attention mechanism with regards to the word "**Spring**".





Input Data Sequence

The animals drank from the **spring**.

$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$

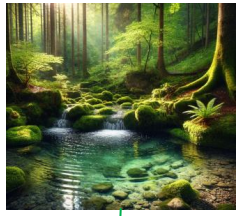
Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Illustration of Attention for the Word "Spring"

In this example, we will illustrate the attention mechanism with regards to the word "**Spring**".





Input Data Sequence

The animals drank from the **spring**.

$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Initial High-Dimensional Embedded Space

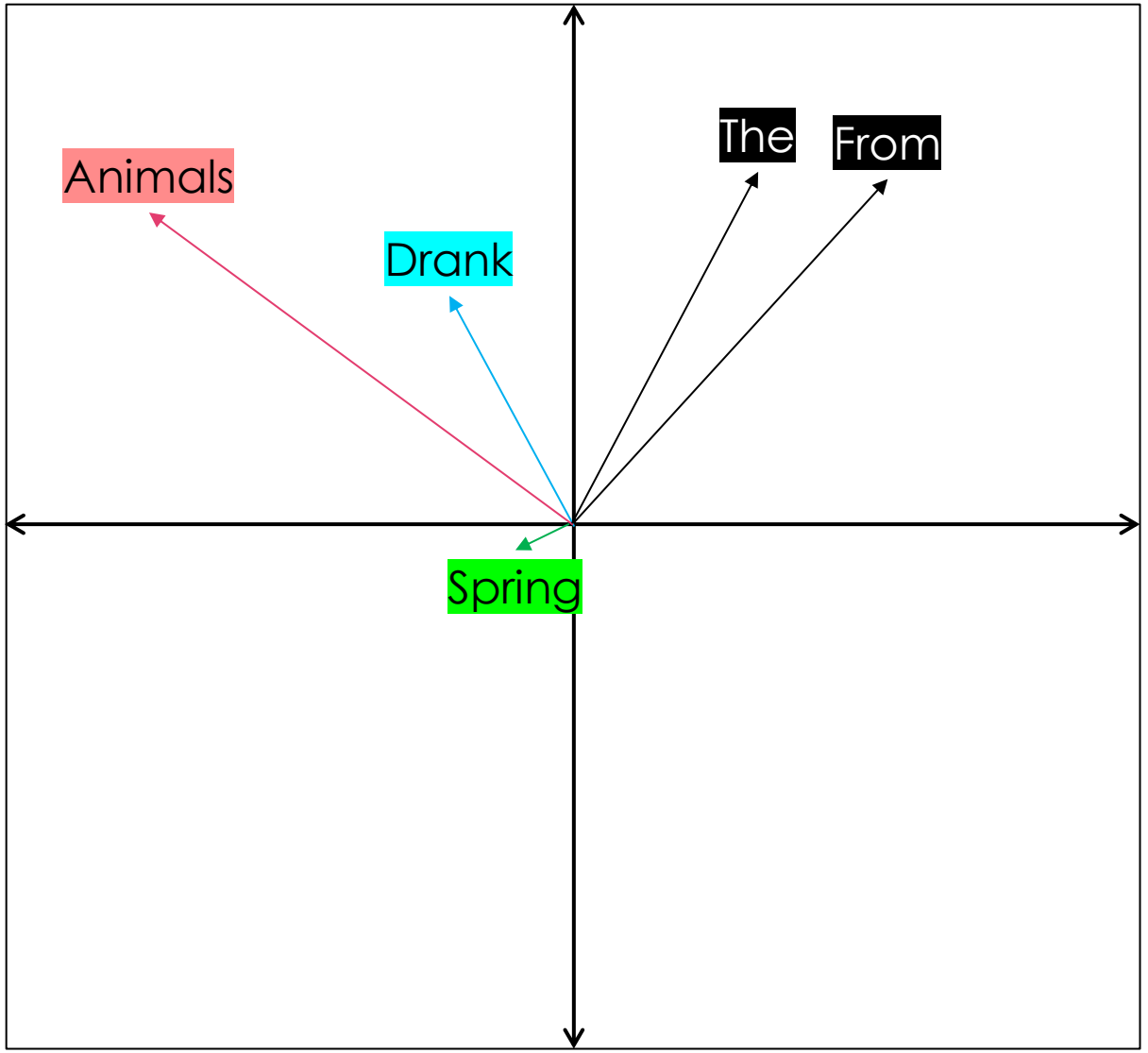


Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Currently, we are interested in the embedding of "**spring**".

Initial High-Dimensional Embedded Space

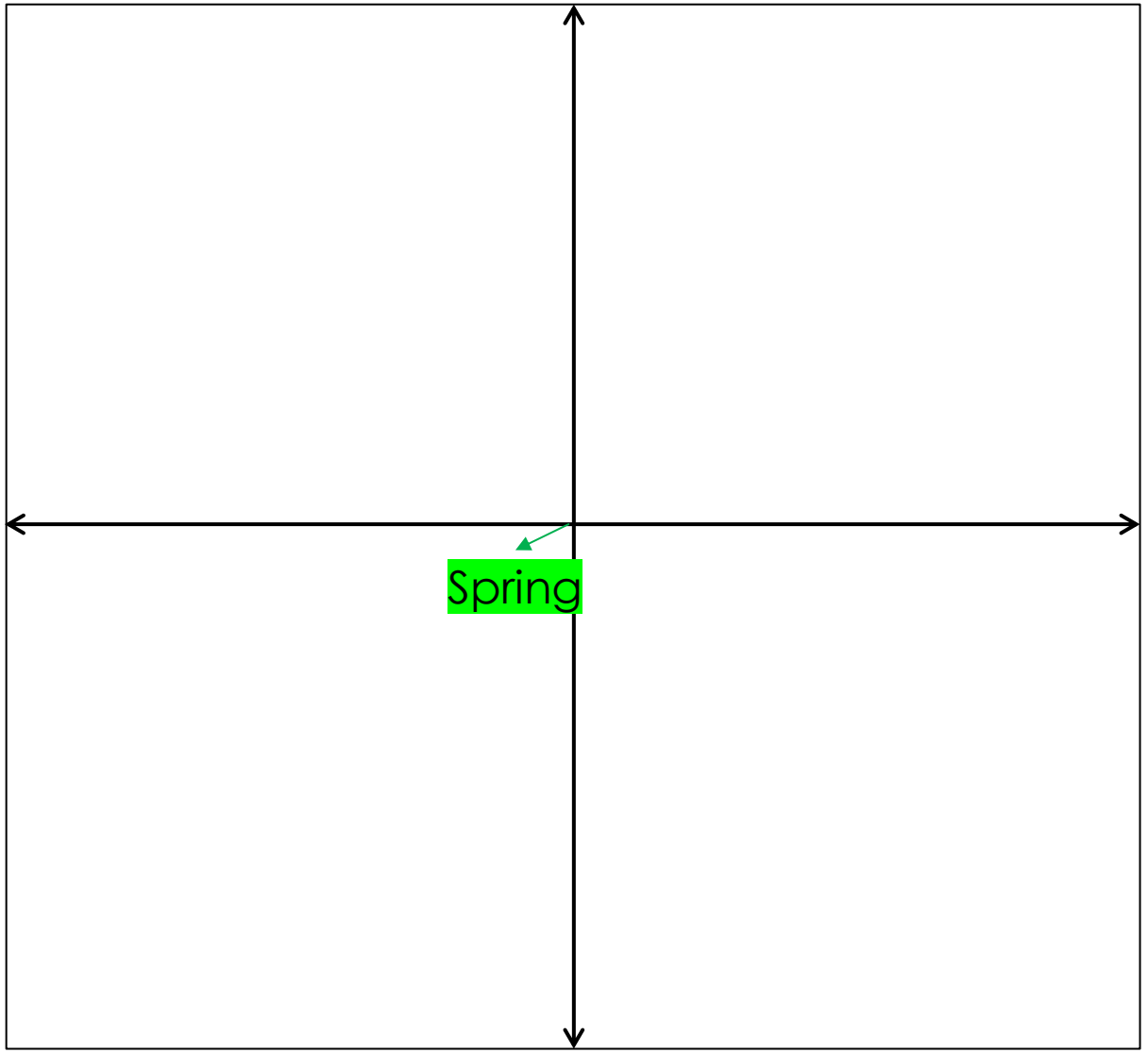


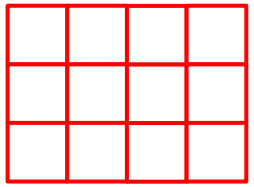
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Query Weights

W_Q



Using the matrix of query weights W_Q, map the initial embedding of "**spring**" (i.e., E_6) into the **Query embedded space**, yielding the query vector Q_6.

Spring: $W_Q E_6 = Q_6$

Query/Key Embedded Space

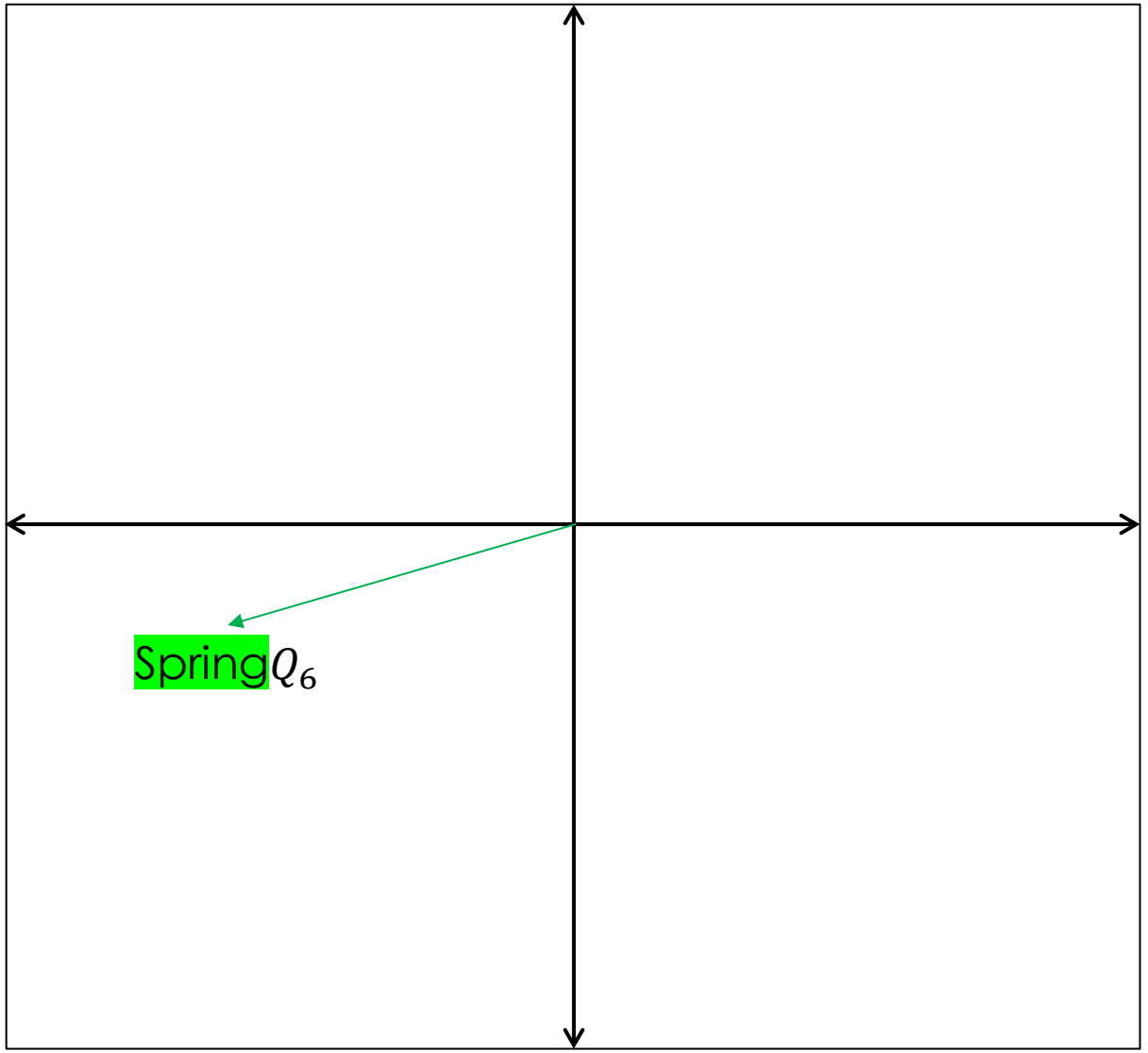


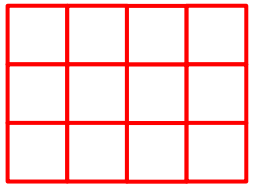
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Query Weights

W_Q



Using the matrix of query weights W_Q, map the initial embedding of "**spring**" (i.e., E_6) into the **Query embedded space**, yielding the query vector Q_6.

- It bears mentioning that in the full attention mechanism, query vectors would be computed for all the words, but we are only interested in the word "spring" for this example.

Spring: $W_Q E_6 = Q_6$

You can think of this query vector as **asking a question** about the **context** surrounding it.

Query/Key Embedded Space

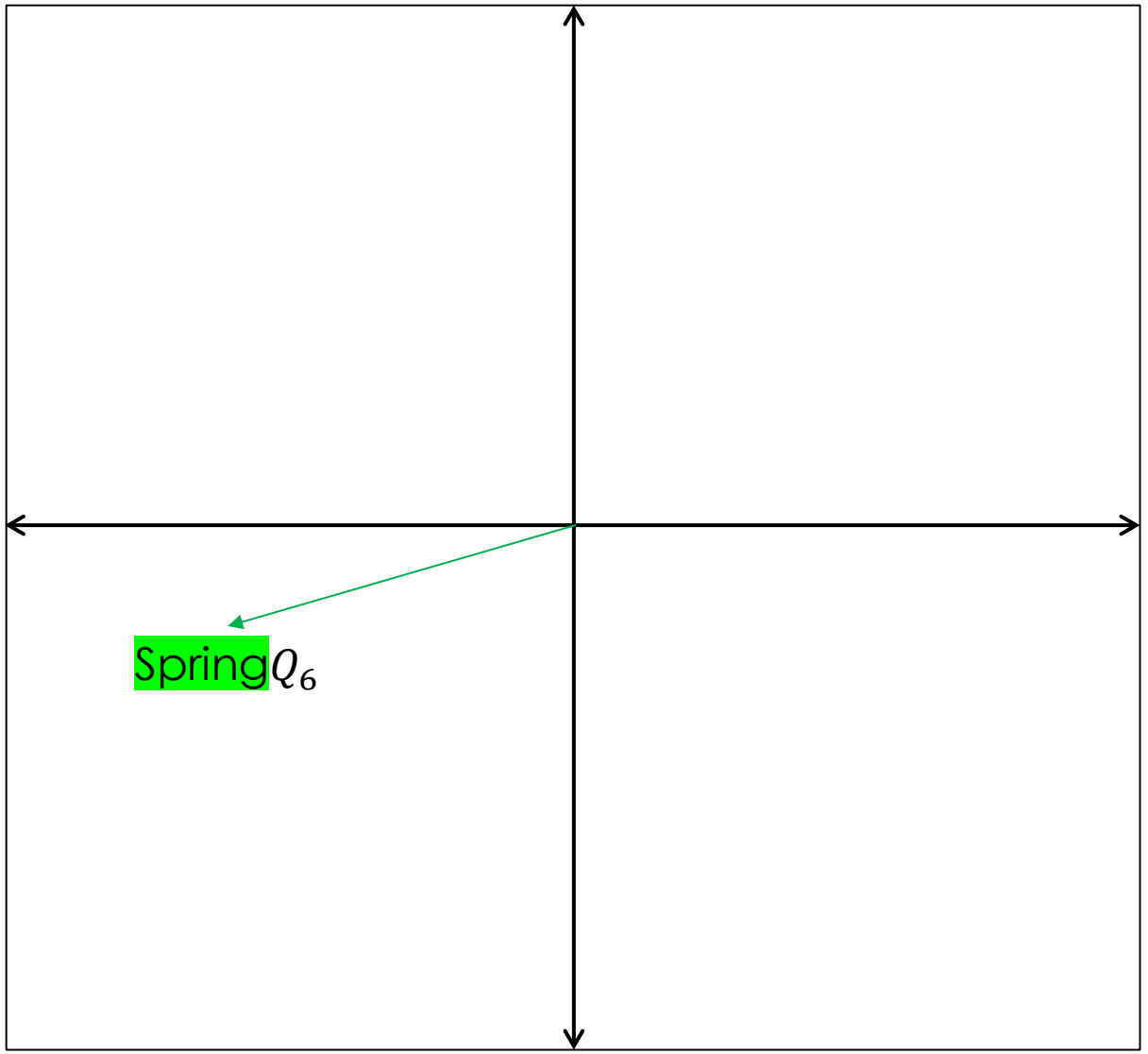


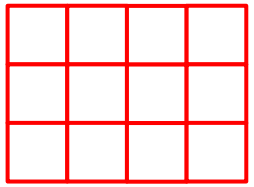
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

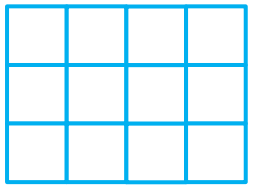
Query Weights

W_Q



Key Weights

W_K



Now, using the matrix of key weights W_K, compute the embeddings of all the other words into this same space.

The: $W_K E_1 = K_1$

Query/Key Embedded Space

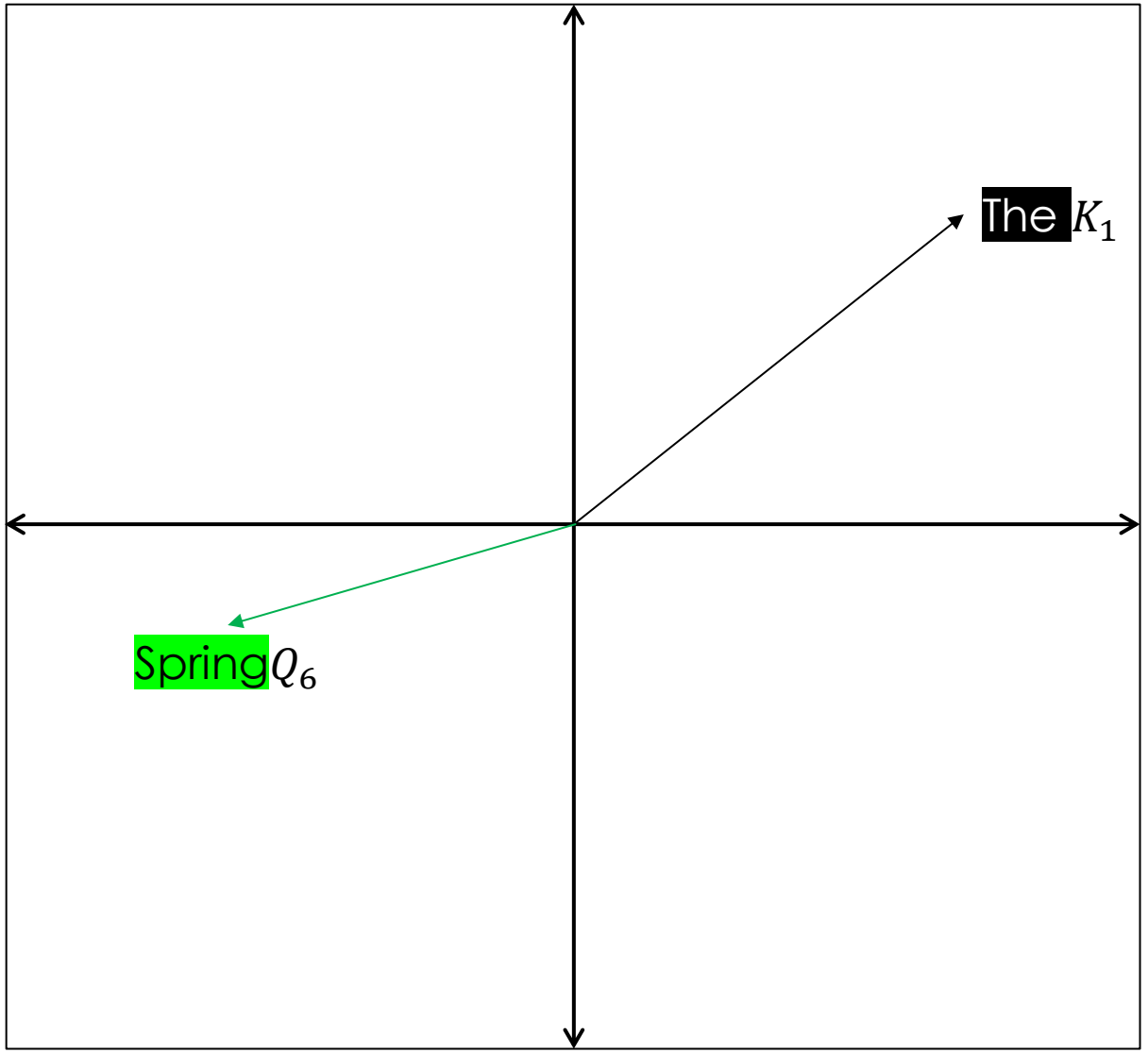


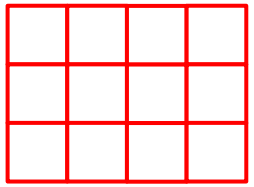
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

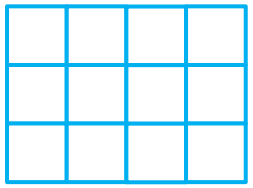
Query Weights

W_Q



Key Weights

W_K



Now, using the matrix of key weights W_K, compute the embeddings of all the other words into this same space.

The: $W_K E_1 = K_1$

Animals: $W_K E_2 = K_2$

Query/Key Embedded Space

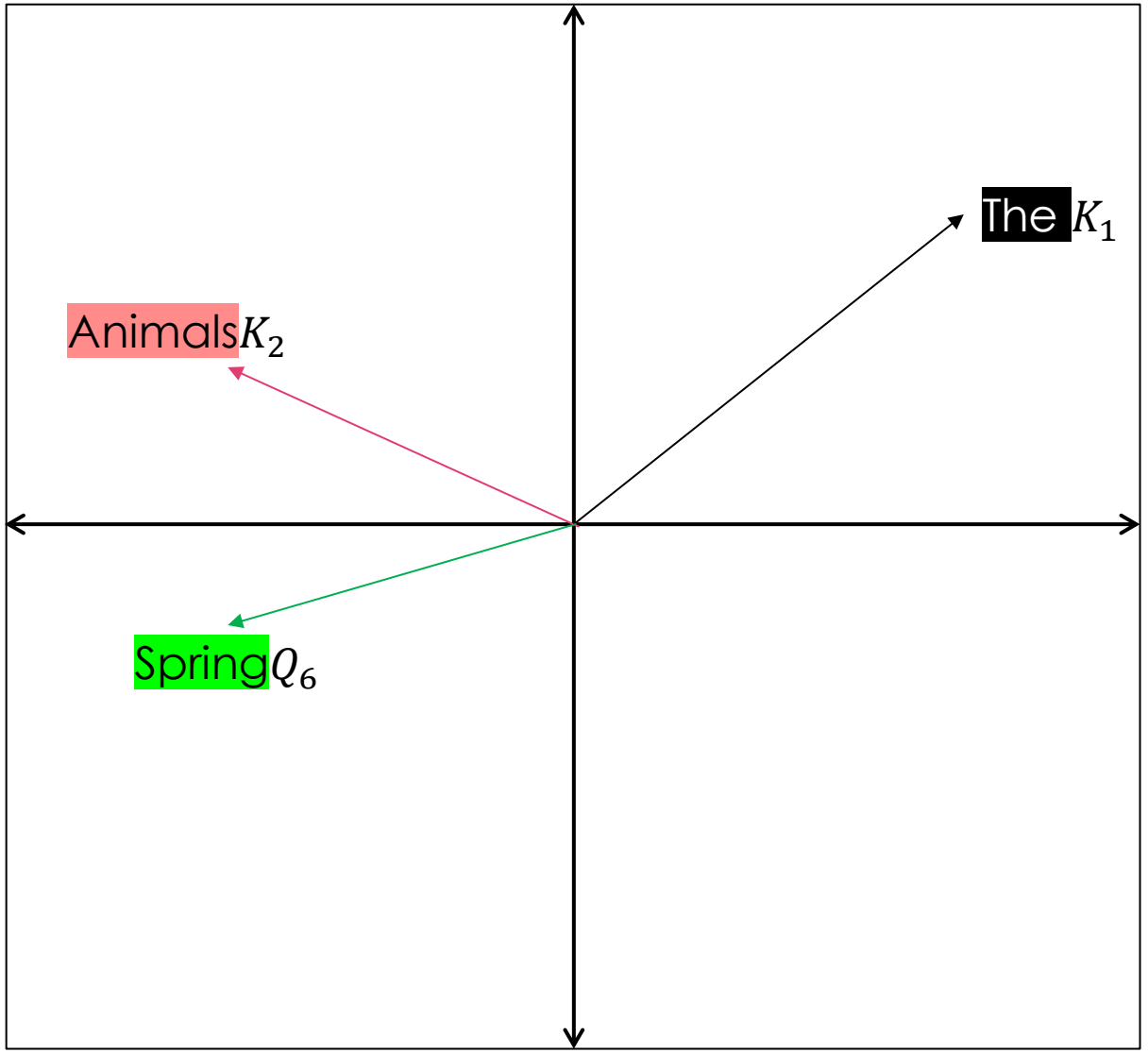


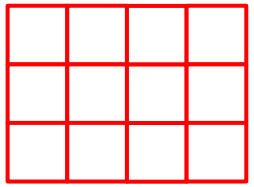
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

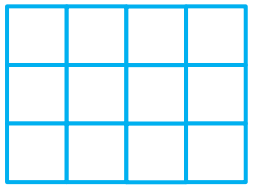
Query Weights

W_Q



Key Weights

W_K



Now, using the matrix of key weights W_K, compute the embeddings of all the other words into this same space.

The: $W_K E_1 = K_1$

Animals: $W_K E_2 = K_2$

Drank: $W_K E_3 = K_3$

Query/Key Embedded Space

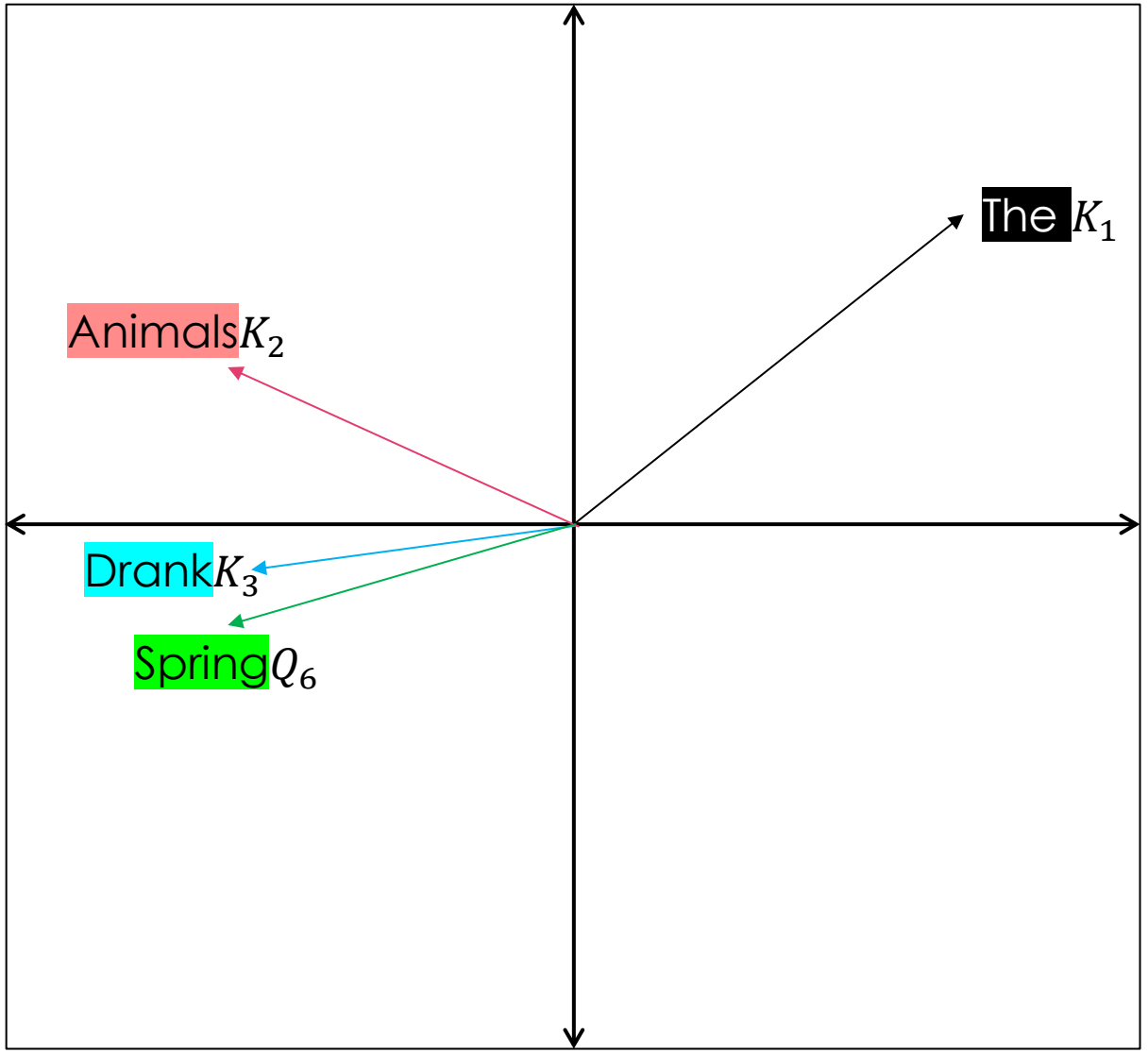


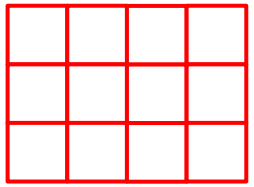
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

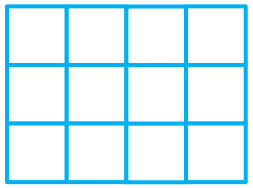
Query Weights

W_Q



Key Weights

W_K



Now, using the matrix of key weights W_K, compute the embeddings of all the other words into this same space.

The: $W_K E_1 = K_1$

Animals: $W_K E_2 = K_2$

Drank: $W_K E_3 = K_3$

From: $W_K E_4 = K_4$

Query/Key Embedded Space

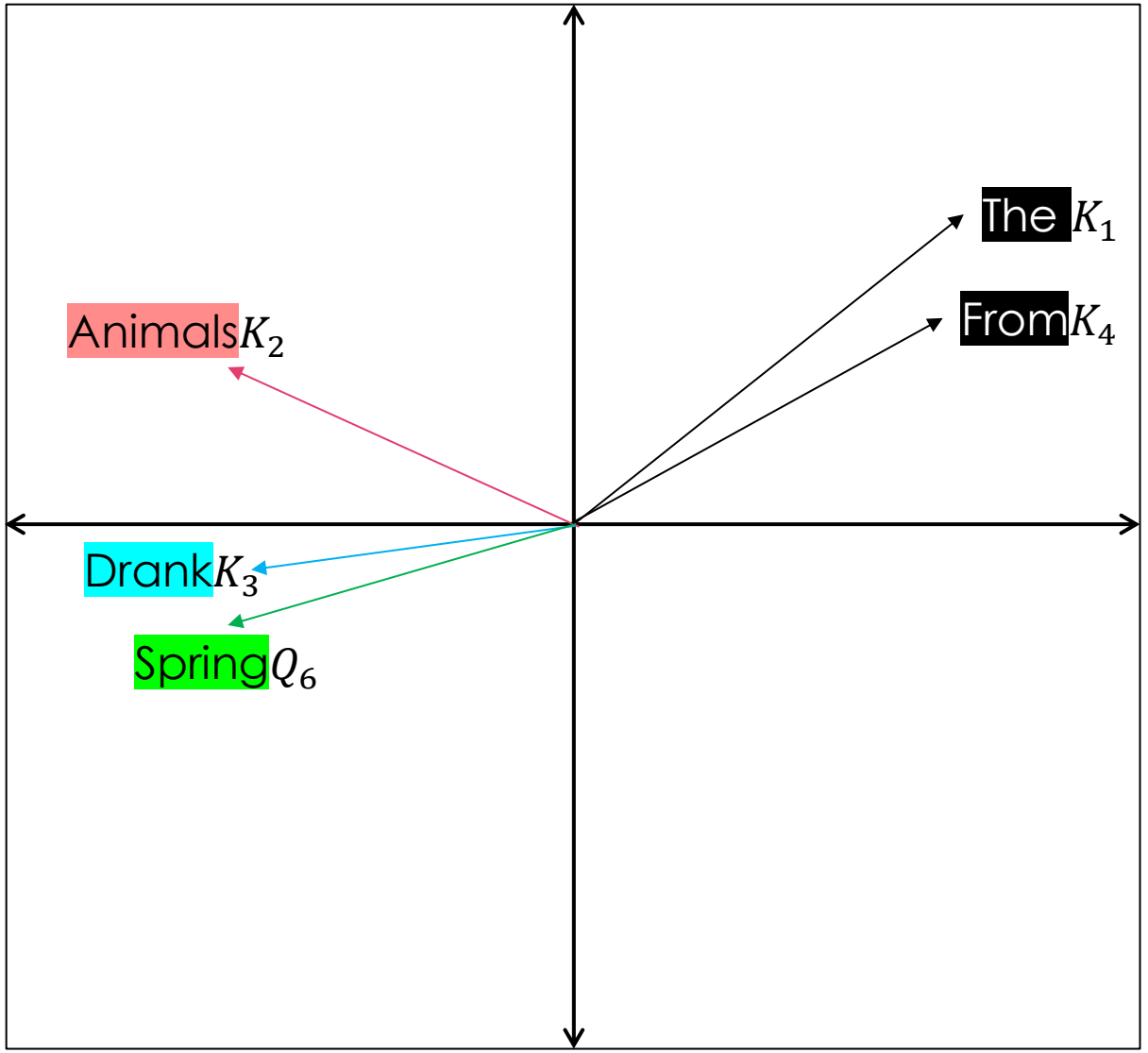


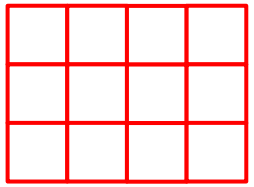
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

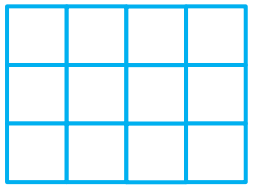
Query Weights

W_Q



Key Weights

W_K



Now, using the matrix of key weights W_K, compute the embeddings of all the other words into this same space.

The:	$W_K E_1 = K_1$
Animals:	$W_K E_2 = K_2$
Drank:	$W_K E_3 = K_3$
From:	$W_K E_4 = K_4$
The:	$W_K E_5 = K_5$
Spring:	$W_K E_6 = K_6$

You can think of this these key vectors as **answering the questions** posed by the **query vectors**.

Query/Key Embedded Space

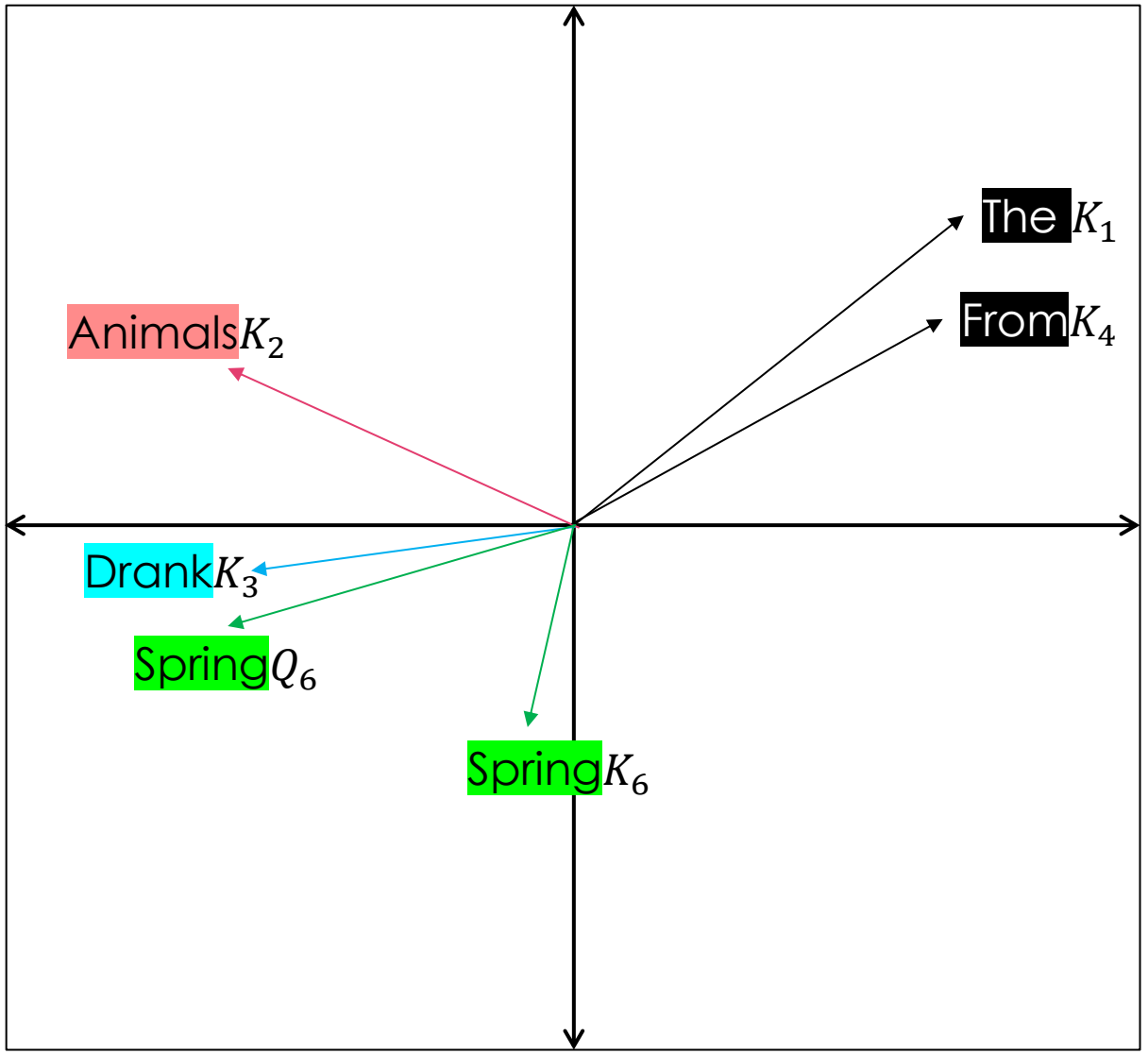


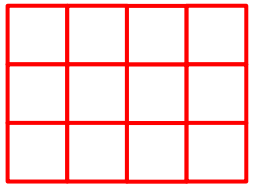
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

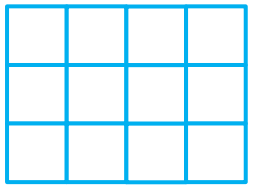
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$S(Q_6, K_1) = G_{6,1}$$

Query/Key Embedded Space

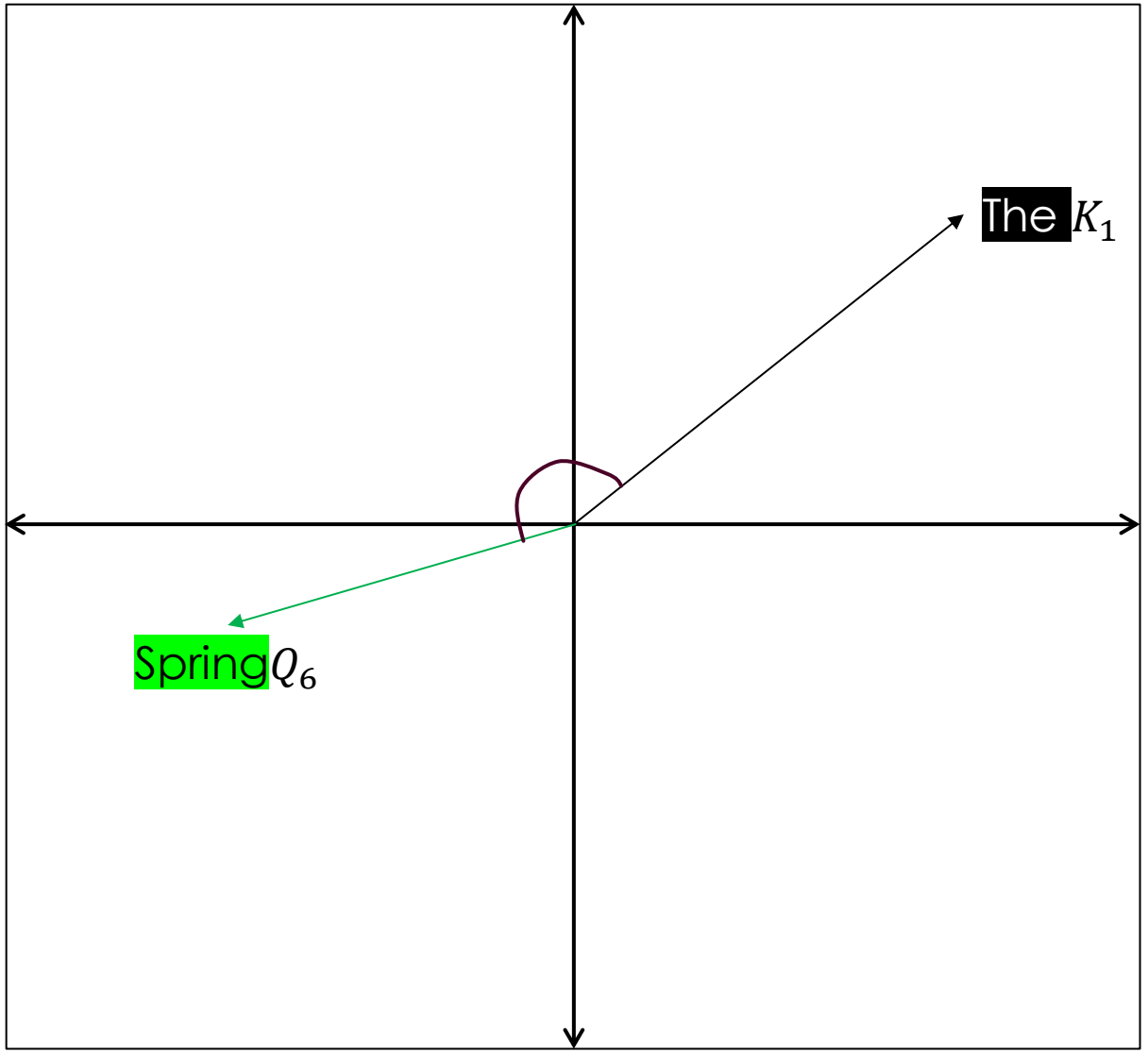


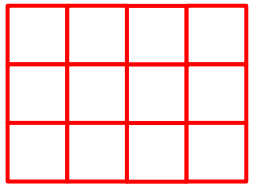
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

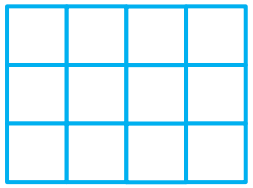
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$S(Q_6, K_1) = G_{6,1}$$
$$S(Q_6, K_2) = G_{6,2}$$

Query/Key Embedded Space

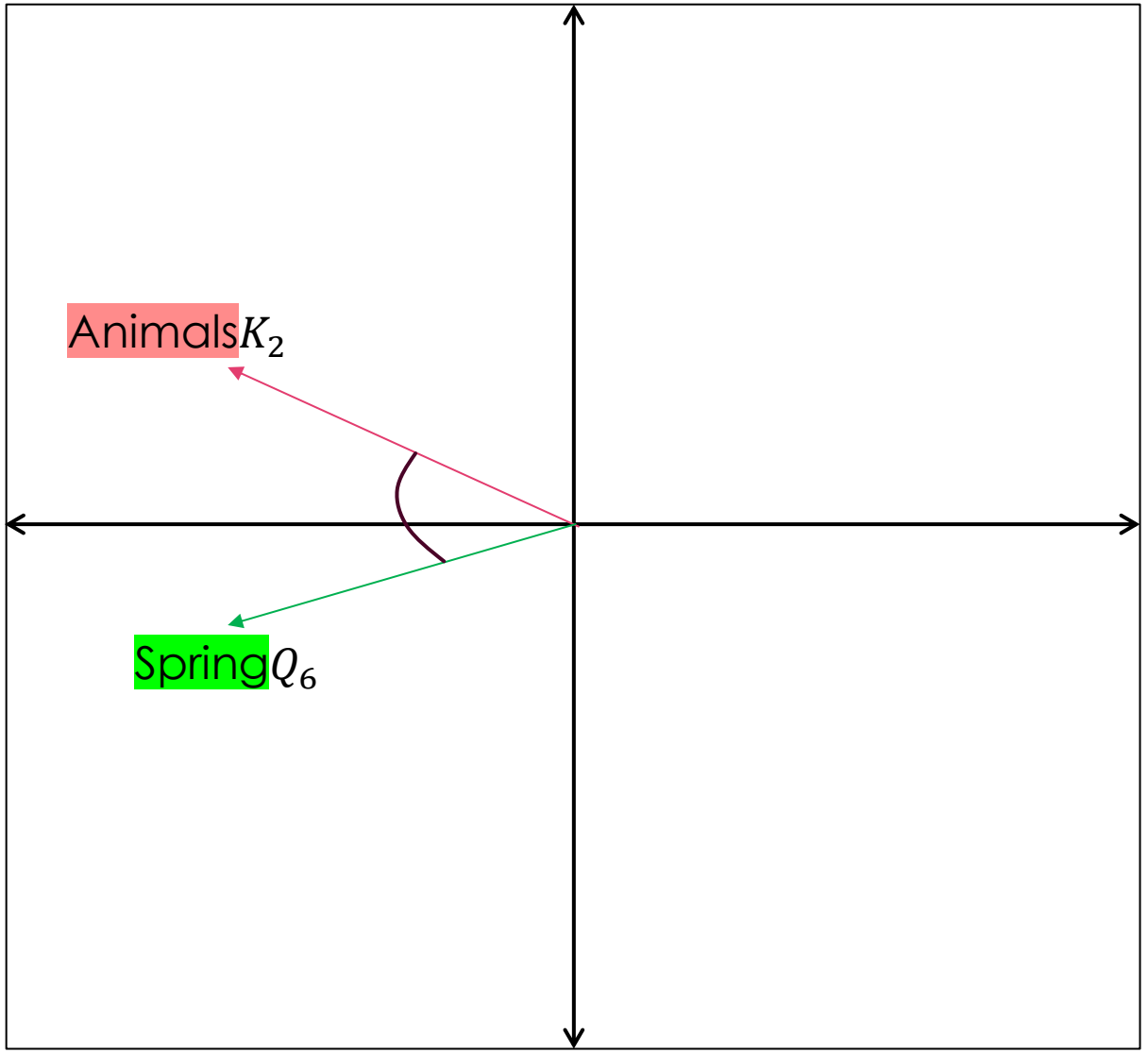


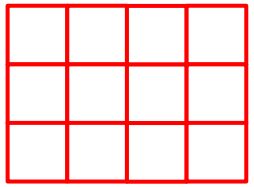
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

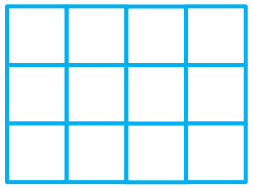
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$S(Q_6, K_1) = G_{6,1}$$
$$S(Q_6, K_2) = G_{6,2}$$
$$S(Q_6, K_3) = G_{6,3}$$

Query/Key Embedded Space

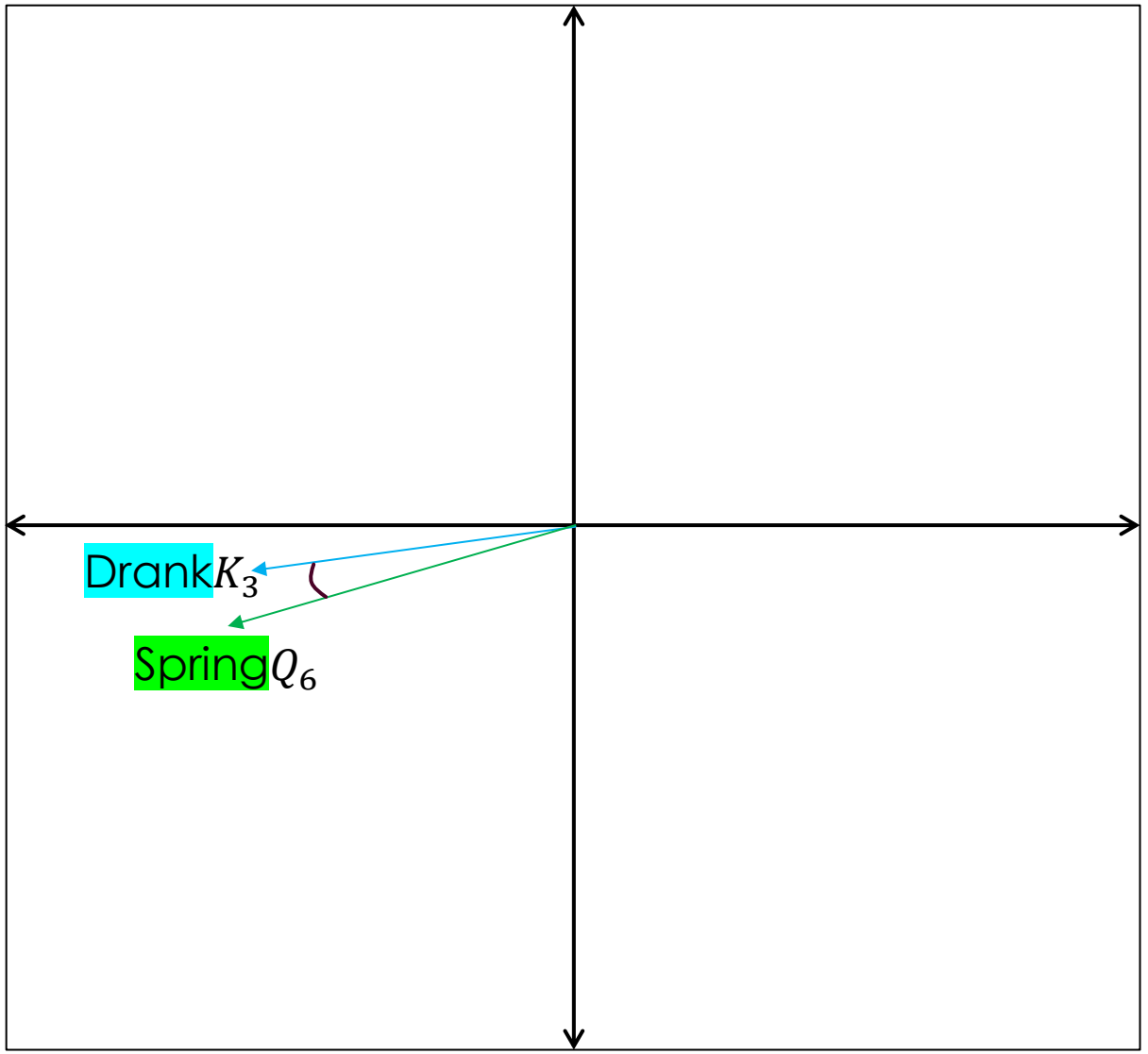


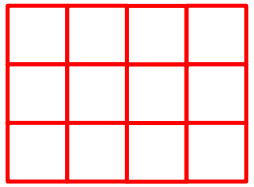
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

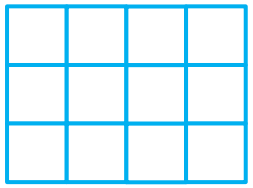
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$S(Q_6, K_1) = G_{6,1}$$
$$S(Q_6, K_2) = G_{6,2}$$
$$S(Q_6, K_3) = G_{6,3}$$
$$S(Q_6, K_4) = G_{6,4}$$

Query/Key Embedded Space

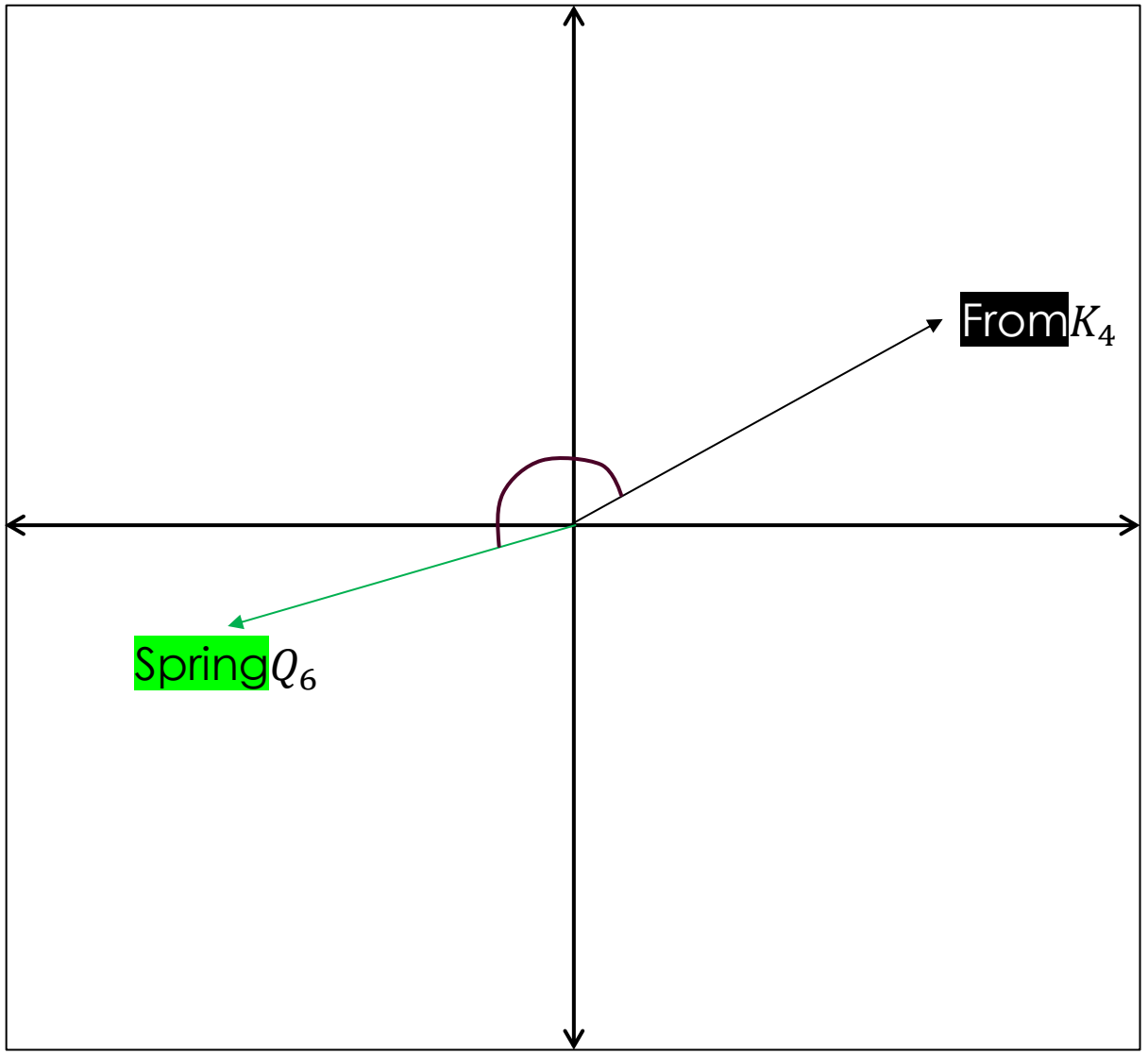


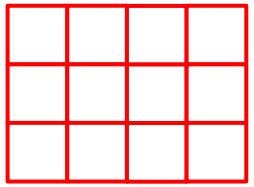
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

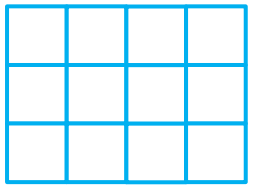
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$S(Q_6, K_1) = G_{6,1}$$
$$S(Q_6, K_2) = G_{6,2}$$
$$S(Q_6, K_3) = G_{6,3}$$
$$S(Q_6, K_4) = G_{6,4}$$
$$S(Q_6, K_5) = G_{6,5}$$
$$S(Q_6, K_6) = G_{6,6}$$

Query/Key Embedded Space

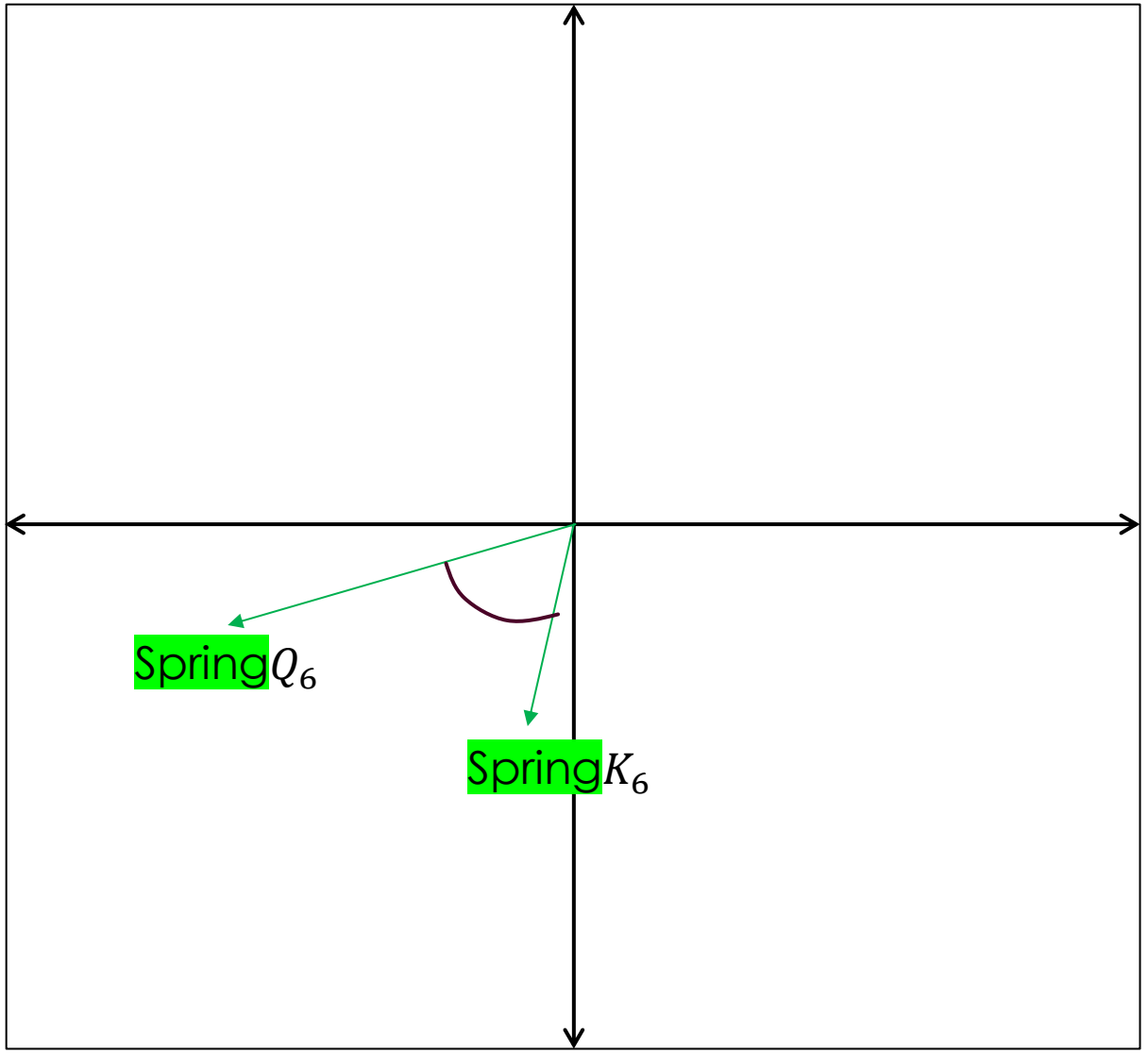


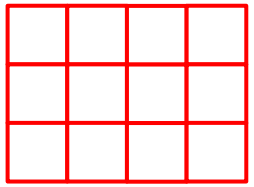
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

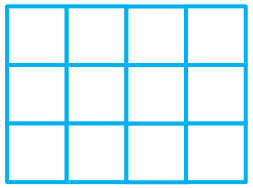
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$S(Q_6, K_1) = G_{6,1}$$
$$S(Q_6, K_2) = G_{6,2}$$
$$\boldsymbol{S(Q_6, K_3) = G_{6,3}}$$
$$S(Q_6, K_4) = G_{6,4}$$
$$S(Q_6, K_5) = G_{6,5}$$
$$S(Q_6, K_6) = G_{6,6}$$

We can see that the word "**Drank**" has the **largest** similarity score, meaning that K_3 most-closely "**answers**" Q_6.

Query/Key Embedded Space

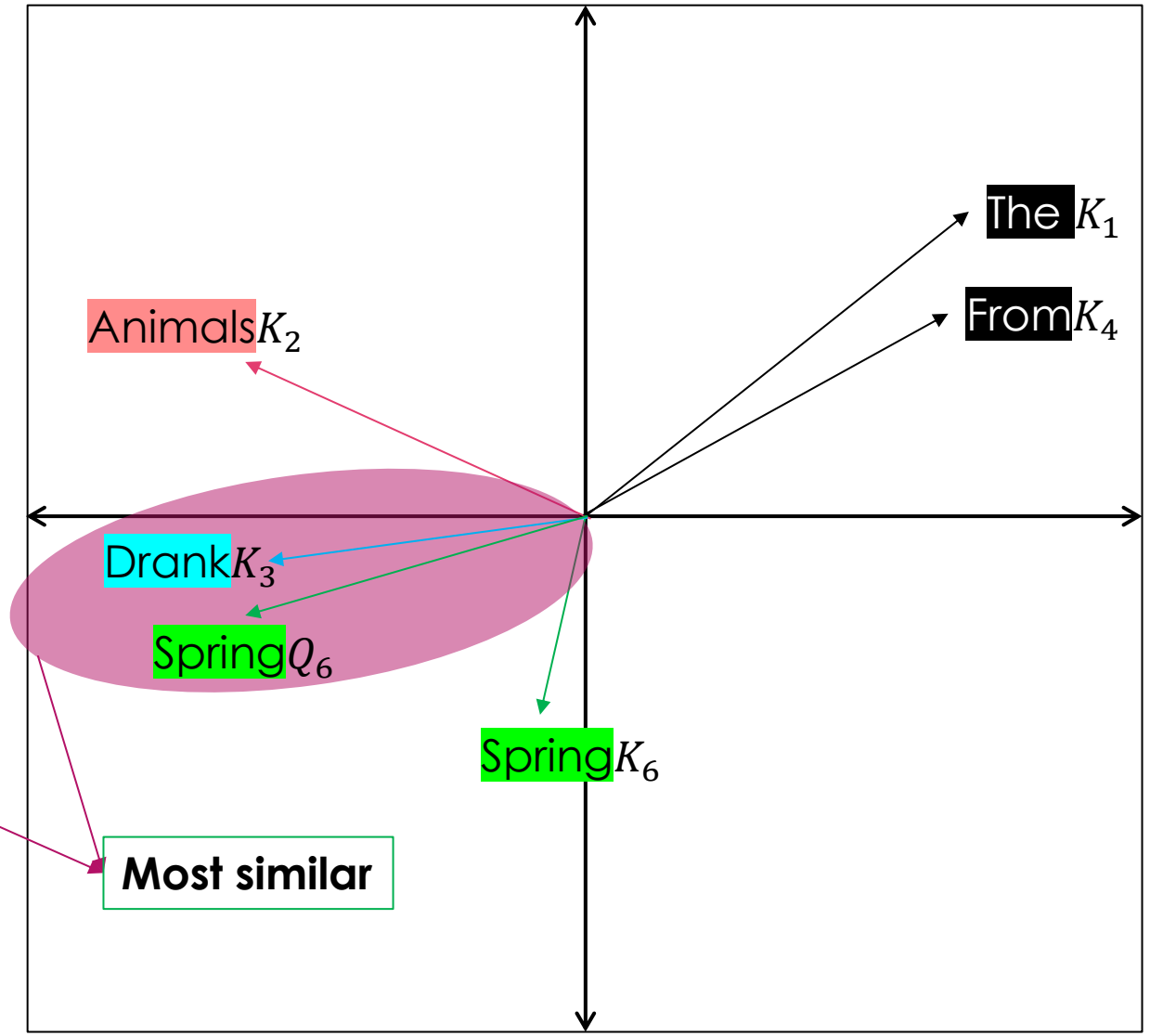


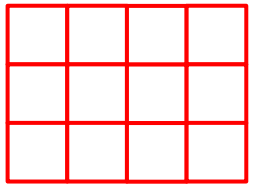
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

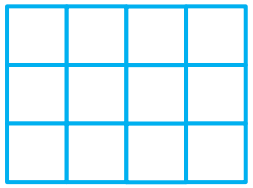
Query Weights

W_Q



Key Weights

W_K



Now, to determine which key vectors "**answer**" the **question** that was asked by the query vector Q_6, compute the **similarity scores** (using the scaled dot-product) between the query vector Q_6 and the key vectors.

$$\left.\begin{aligned} S(Q_6, K_1) &= G_{6,1} \\ S(Q_6, K_2) &= G_{6,2} \\ \boldsymbol{S(Q_6, K_3) = G_{6,3}} &\\ S(Q_6, K_4) &= G_{6,4} \\ S(Q_6, K_5) &= G_{6,5} \\ S(Q_6, K_6) &= G_{6,6} \end{aligned}\right\} G_6 \in \mathbb{R}^6$$

We can see that the word "**Drank**" has the **largest** similarity score, meaning that K_3 most-closely "**answers**" Q_6.

Query/Key Embedded Space

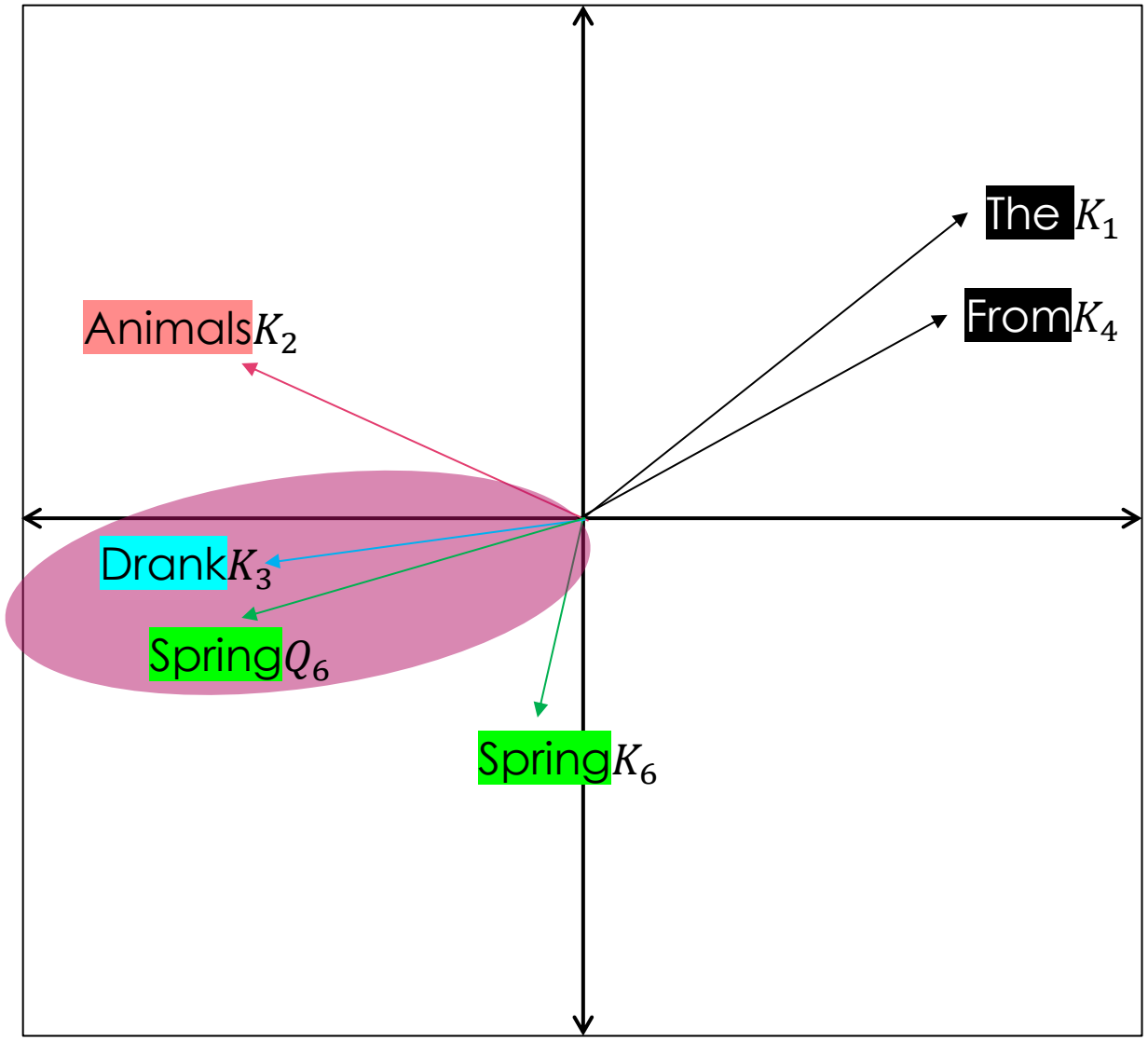


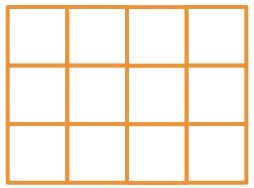
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Value Weights

W_V



Using the value weight matrix W_V, compute the value embedding for "spring" as

$$W_V E_6 = V_6.$$

Now, in order to actually update the position of the embeddings of the words, we transform the query/key space into the value space (which in this example is simply the same as the original embedding space for clarity).

Value/Initial High-Dimensional Embedded Space

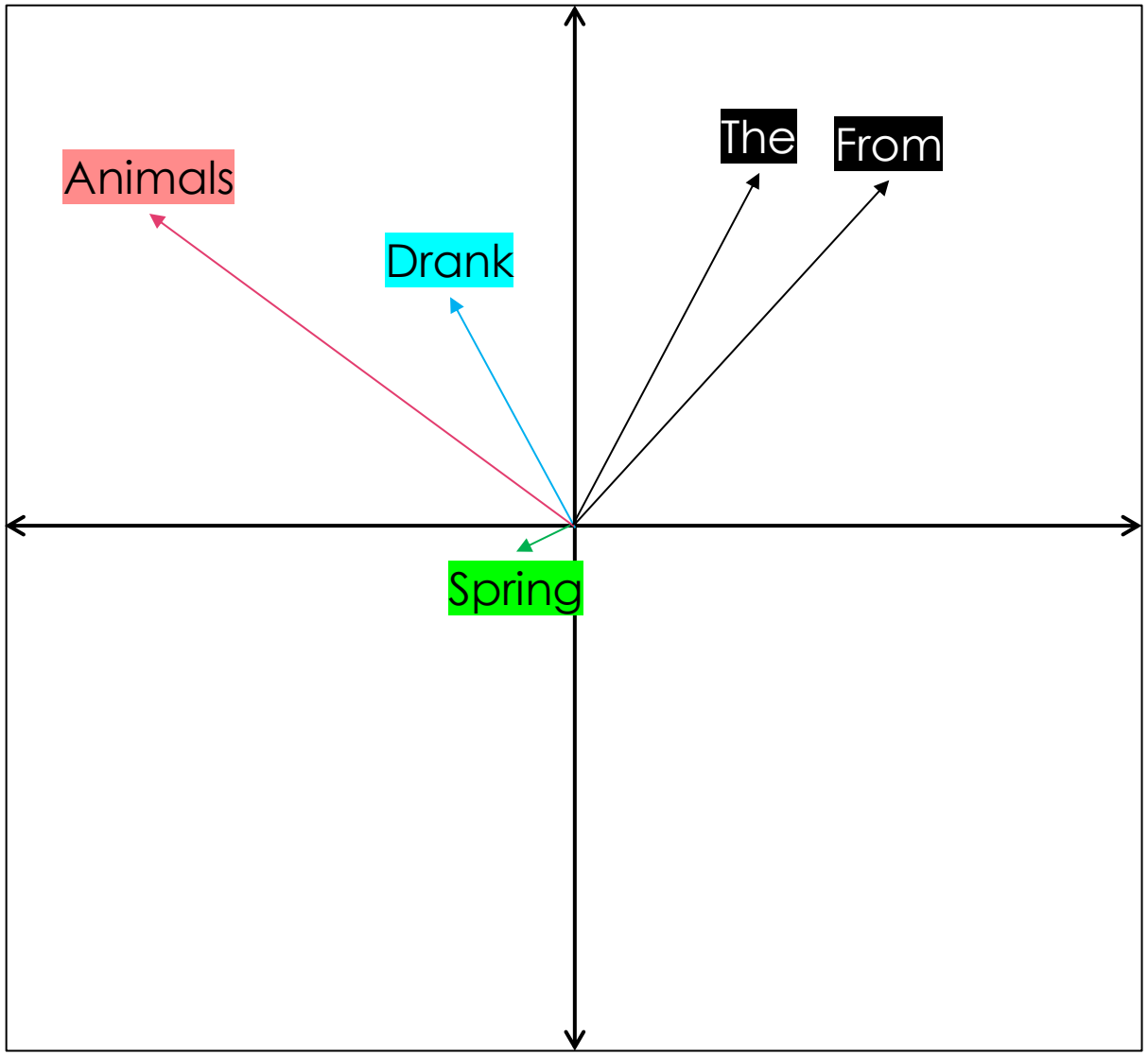


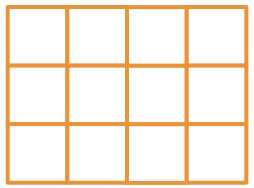
Illustration of Attention for the Word "Spring"

Initial Embeddings

$$E_1 = \begin{bmatrix} 0.2 \\ 5 \\ \vdots \\ -0.15 \end{bmatrix} \quad E_2 = \begin{bmatrix} 1 \\ 0.8 \\ \vdots \\ 23 \end{bmatrix} \quad E_3 = \begin{bmatrix} 0 \\ -2 \\ \vdots \\ 0.1 \end{bmatrix} \quad E_4 = \begin{bmatrix} -1 \\ 0.9 \\ \vdots \\ 2 \end{bmatrix} \quad E_5 = \begin{bmatrix} 24 \\ 1.2 \\ \vdots \\ 0.7 \end{bmatrix} \quad E_6 = \begin{bmatrix} 10 \\ -4 \\ \vdots \\ 0 \end{bmatrix}$$

Value Weights

W_V



Using the value weight matrix W_V, compute the value embedding for "spring" as

$$W_V E_6 = V_6.$$

Now, in order to actually update the position of the embeddings of the words, we transform the query/key space into the value space (which in this example is simply the same as the original embedding space for clarity).

$$\text{Attention}(W_Q, W_K, W_V)_6 = V_6^T \text{softmax}(G_6)$$

Thus, performing the **mapping from the query/key space into the value space** will **shift the words in space** to **closer represent the correct semantic context**.

Value/Initial High-Dimensional Embedded Space

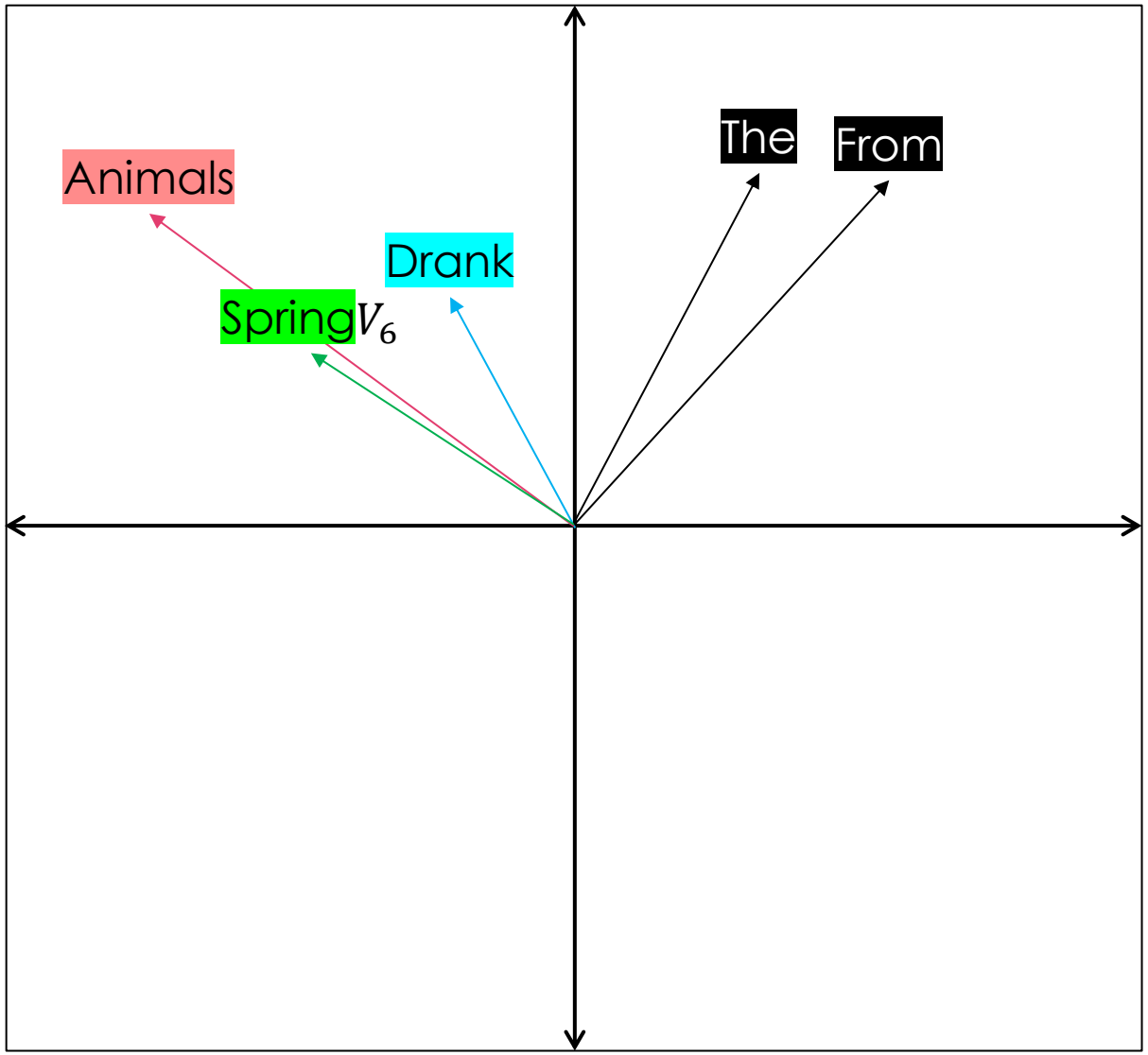


Illustration of Attention for the Word “Spring”

Input Data Sequence (1)

The animals drank from the **spring**.

Embedded Space (Effect of Attention on “Spring”)

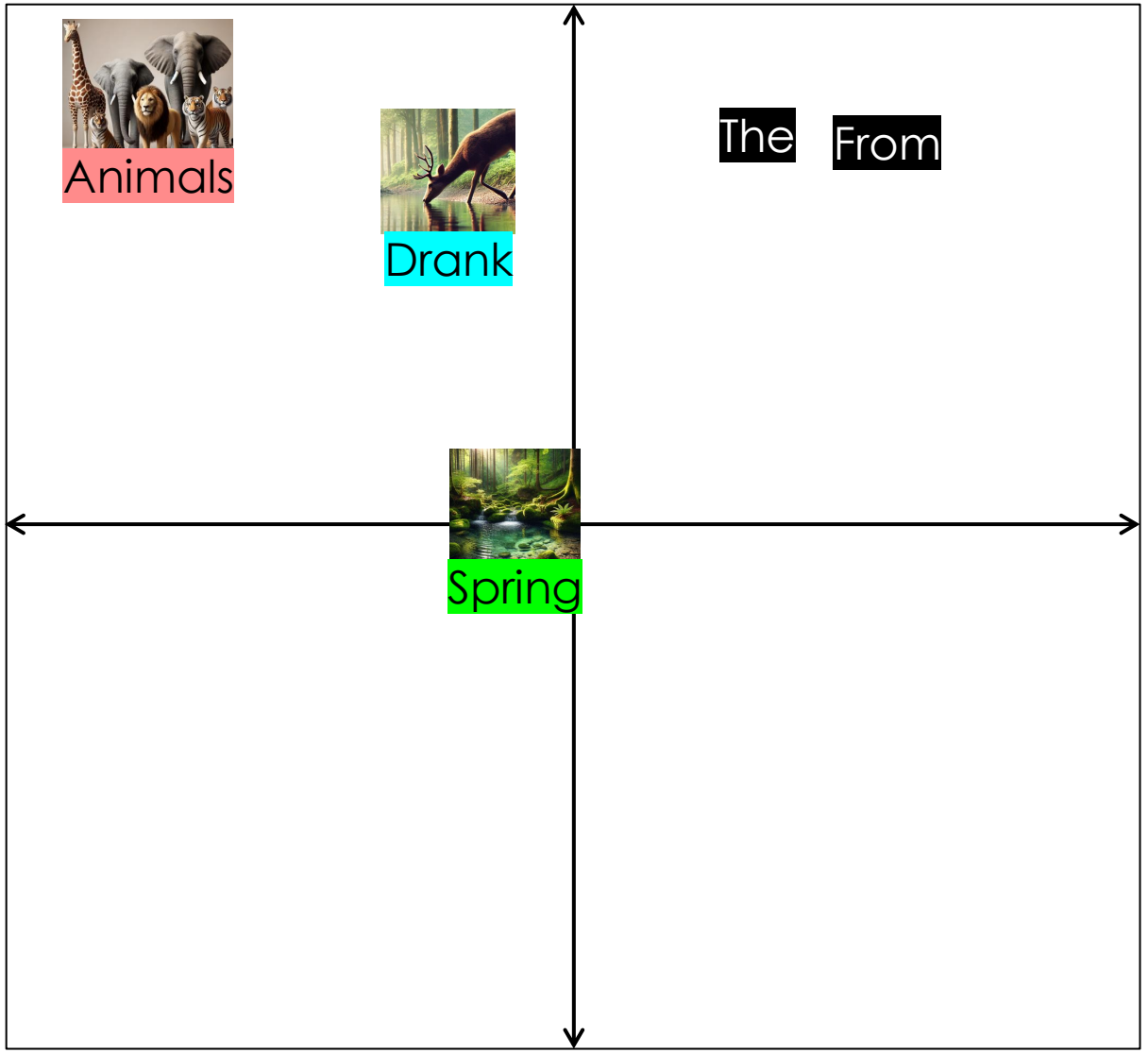


Illustration of Attention for the Word “Spring”

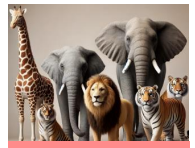
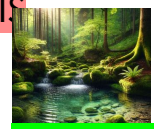


Input Data Sequence (1)

The animals drank from the **spring**.

$\text{Attention}(\text{Spring}(1))$

Embedded Space (Effect of Attention on “Spring”)


Animals

Spring

Drank

The From

Illustration of Attention for the Word “Spring”



Input Data Sequence (1)

The animals drank from the **spring**.



Input Data Sequence (2)

The mechanical **spring** is a tool.

Embedded Space (Effect of Attention on “Spring”)

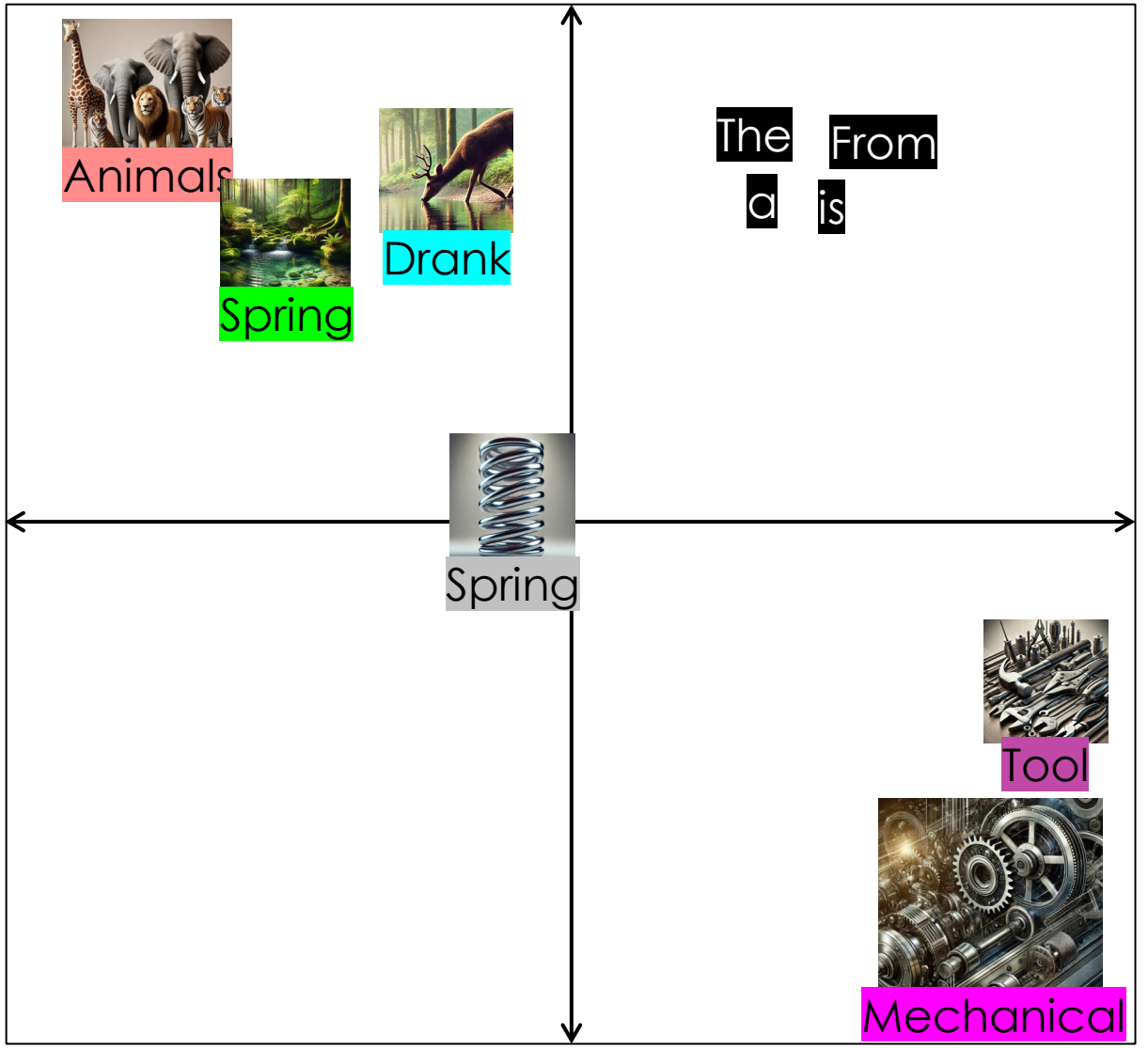


Illustration of Attention for the Word “Spring”



Input Data Sequence (1)

The animals drank from the **spring**.

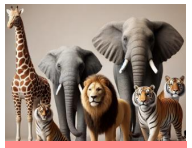


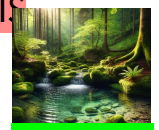
Input Data Sequence (2)

The mechanical **spring** is a tool.

Attention(Spring(2))

Embedded Space (Effect of Attention on “Spring”)


Animals


Spring


Drank

The

From

a

is


Tool


Spring

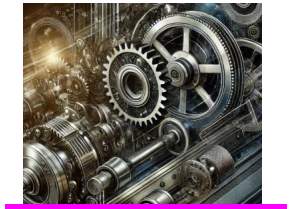

Mechanical

Illustration of Attention for the Word “Spring”

Input Data Sequence (1)

The animals drank from the **spring**.

Input Data Sequence (2)

The mechanical **spring** is a tool.

Input Data Sequence (3)

The **spring** season was warm.

Embedded Space (Effect of Attention on “Spring”)

Illustration of Attention for the Word “Spring”

Input Data Sequence (1)

The animals drank from the **spring**.

Input Data Sequence (2)

The mechanical **spring** is a tool.

Input Data Sequence (3)

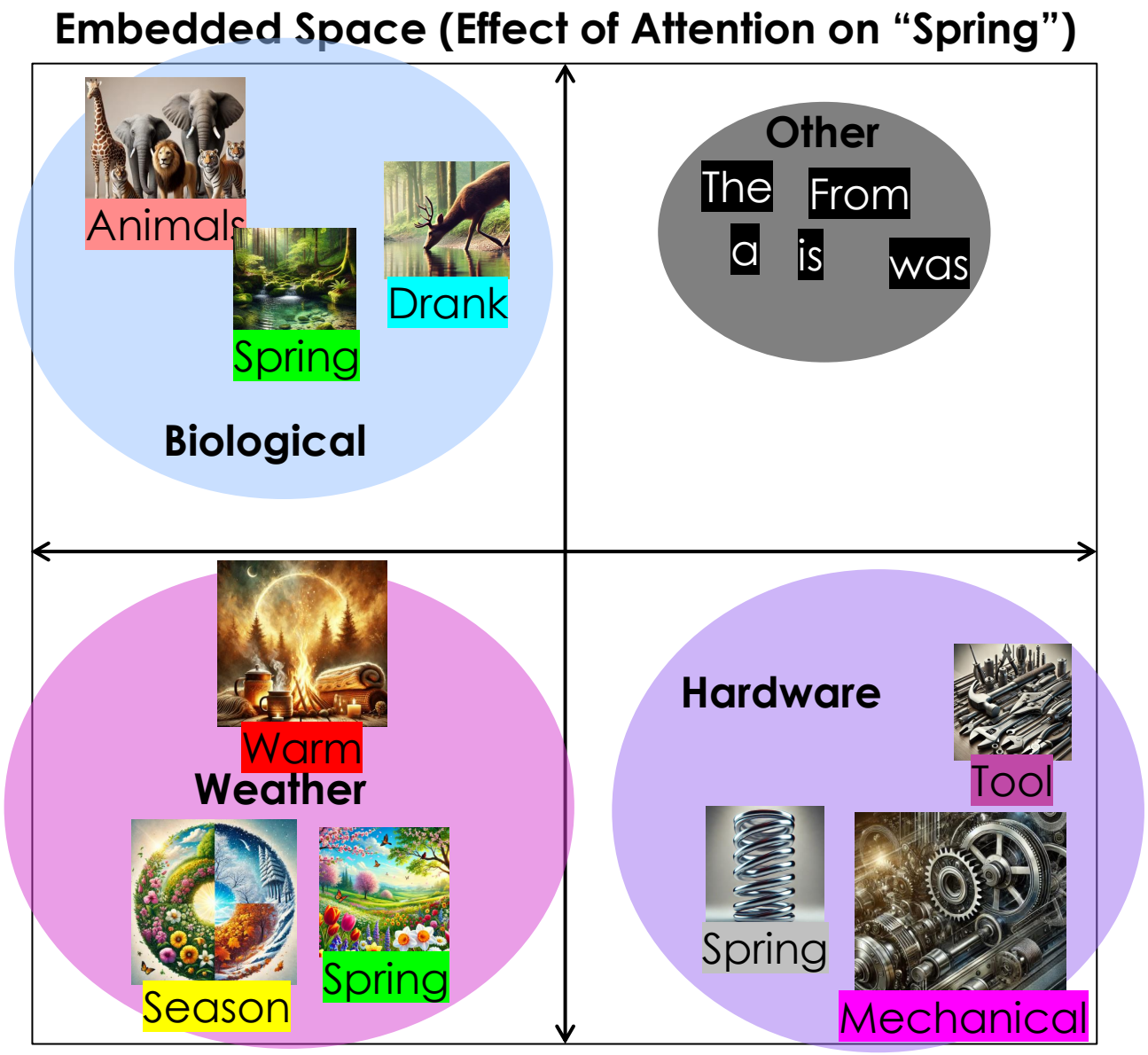
The **spring** season was warm.

$\text{Attention}(\text{Spring}(3))$

Embedded Space (Effect of Attention on “Spring”)

Illustration of Attention for the Word “Spring”

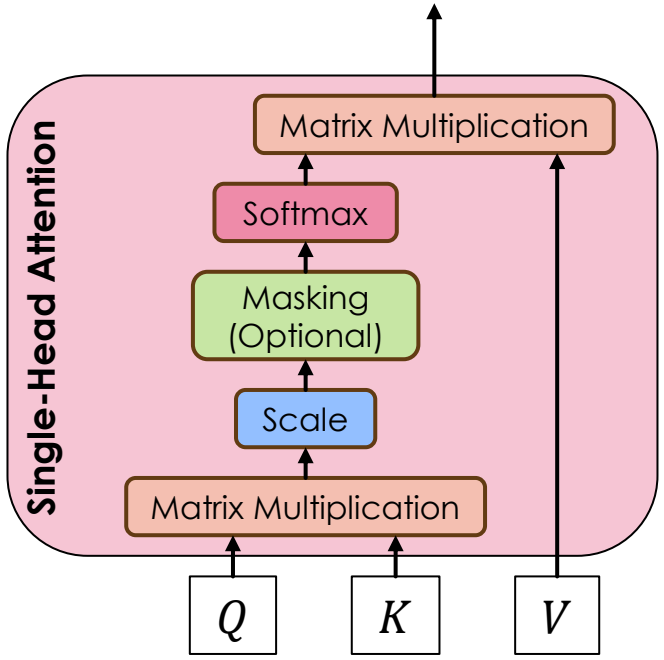
The fundamental **purpose of attention** is to **generate better embeddings** that can **capture more complete semantic context** within data.



Single-Head Vs. Multi-Headed Attention

The Single-Head Attention Mechanism

We have already formally introduced the complete definition of the attention mechanism (which is referred to as "**single-head attention**" or "**self-attention**"). As stated earlier, a single attention operation essentially takes an initial embedding as input and transforms it into a space that better encodes contextual and semantic content.



- Notice that the attention operation can optionally be modified to include an extra internal operation known as "**masking**". This is something that is performed to the similarity matrix $G \in \mathbb{R}^{m \times m}$ before the softmax function. Specifically, the attention operation is modified as

$$\mathrm{A}(Q,K,V) \coloneqq \mathrm{s.max}\left(\frac{QK^T}{\sqrt{s}} + M\right)V,$$

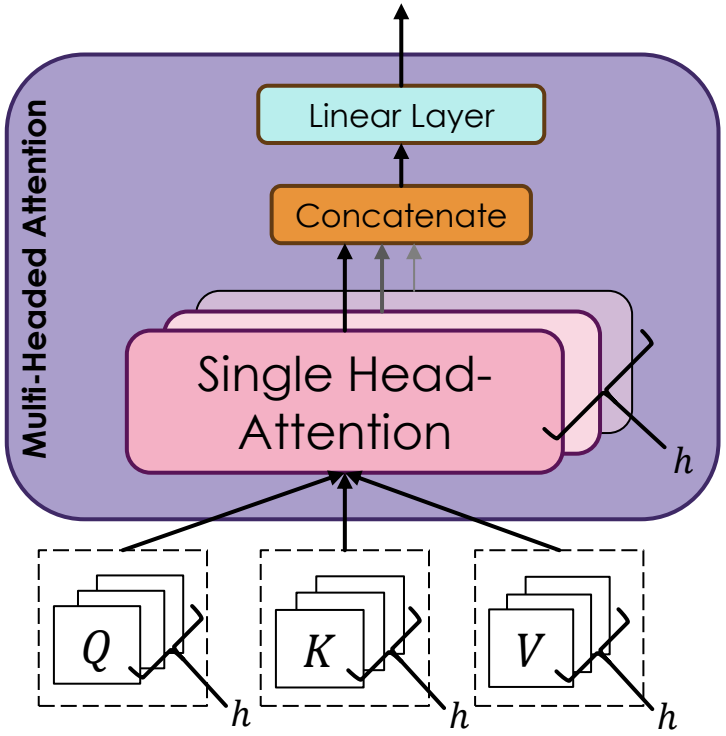
where $M \in \mathbb{R}^{m \times m}$ is the masking matrix.

- The elements of $M_{i,j}$ are either 0 if $i \leq j$ or $-\infty$ if $i > j$. This prevents similarity scores of "future" tokens from impacting the prediction of "past" tokens.

The Multi-Headed Attention Mechanism

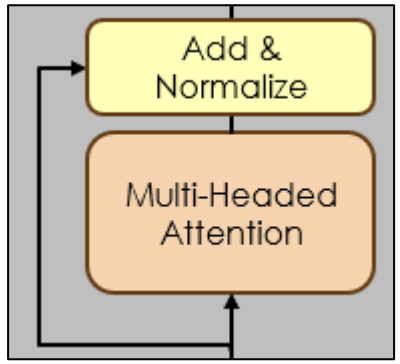
The **multi-headed** attention operation is exactly as it sounds; it simply incorporates multiple single-head attention operations and is what is used in transformer architectures. Specially, this will compute some $h \in \mathbb{N}$ total attention embeddings (each requiring their own W_Q, W_K, and V matrices, yielding a total of h of each), followed by a linear fully-connected layer.

- The new embeddings from all the attention layers are **concatenated** into a very large high-dimensional matrix. This is where the **linear layer** comes in; it can be thought of as "**choosing**" which embedded dimensions are the "**best**" by performing a linear combination of them, resulting in a new **lower-dimensional embedded space**.



Addition & Normalization Operation

(Snapshot from a Transformer Block)



Addition & Layer Normalization Operations

These are operations that add the **resulting embedding of the attention block** (which we can denote here as $\text{SubLayer}(x) \in \mathbb{R}^d$) to the **original embedding before the attention** (which we can denote as the input vector $x \in \mathbb{R}^d$), followed by "**layer normalization**". These two embeddings are added together to aid with **combatting the vanishing gradient problem** that can occur in the training process, and it can be viewed as helping keep the structure of the original data present throughout the network. Similarly, the normalization is performed to help ensure numerical stability in the training process by ensuring that the range of values does not get too large or small.

Therefore, the output of the "**Add & Normalize**" block is defined as

$$\text{Output} = \text{LayerNorm}\big(x + \text{SubLayer}(x)\big),$$

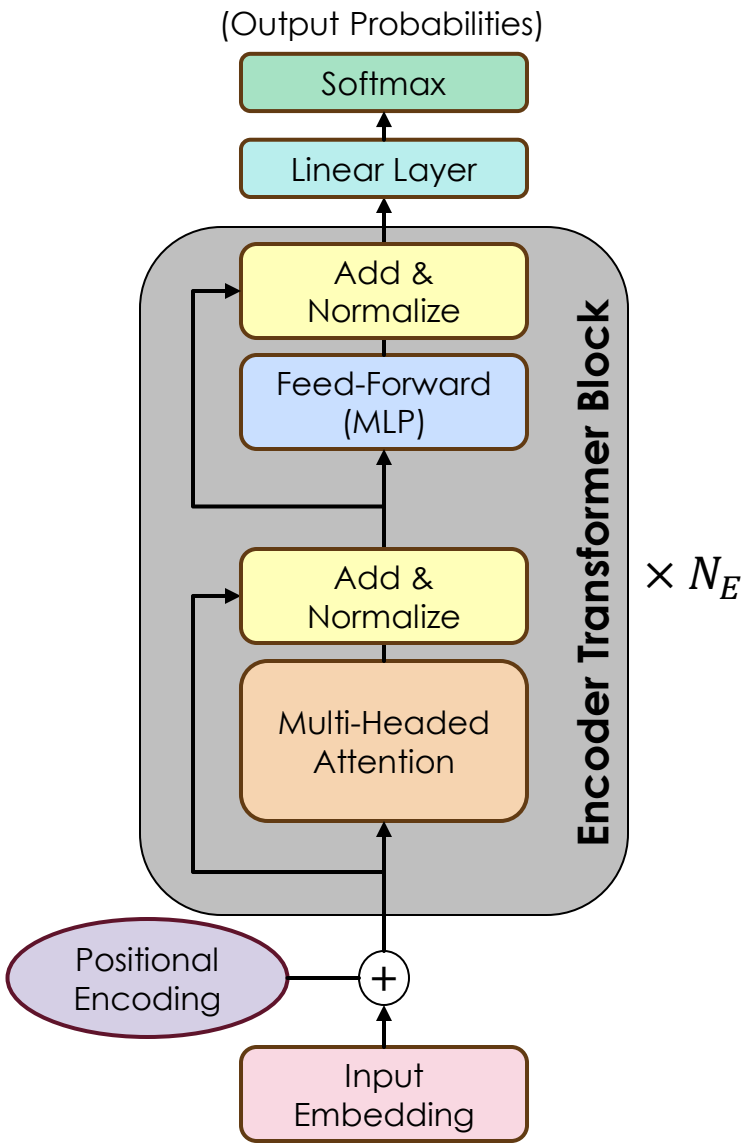
where the **layer normalization** operation is defined as

$$\text{LayerNorm}(z) = \frac{z - \mu}{\sigma} \cdot \gamma + \beta,$$

where $z \in \mathbb{R}^d$ is the input vector to normalize, $\mu \in \mathbb{R}$ is the mean of z, $\sigma \in \mathbb{R}$ is the standard deviation, and $\gamma \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are trainable scaling and offset parameters (to allow for less-restrictive and more-nuanced normalizing).

Incorporating the input vector x in the layer normalization of the output of $\text{SubLayer}(x)$ is referred to as a "**Residual Connection**" or a "**Skip Connection**" and are typically incorporated to help **enable better propagation of the loss to the entire network** when computing gradients (this helps eliminate the vanishing gradient problem).

Encoder-Only Transformer Architectures



The Encoder-Only Architecture

The first of the three main transformer architectures is the **encoder-only architecture**. It consists of three stages: **(1)** given some input data, properly embed it into some vector space that incorporates positional information (with regards to sequential data), **(2)** some number $N_E \in \mathbb{N}$ of encoder blocks, **(3)** followed by an output layer that returns predicted probabilities (or numeric values).

The Encoder Transformer Block

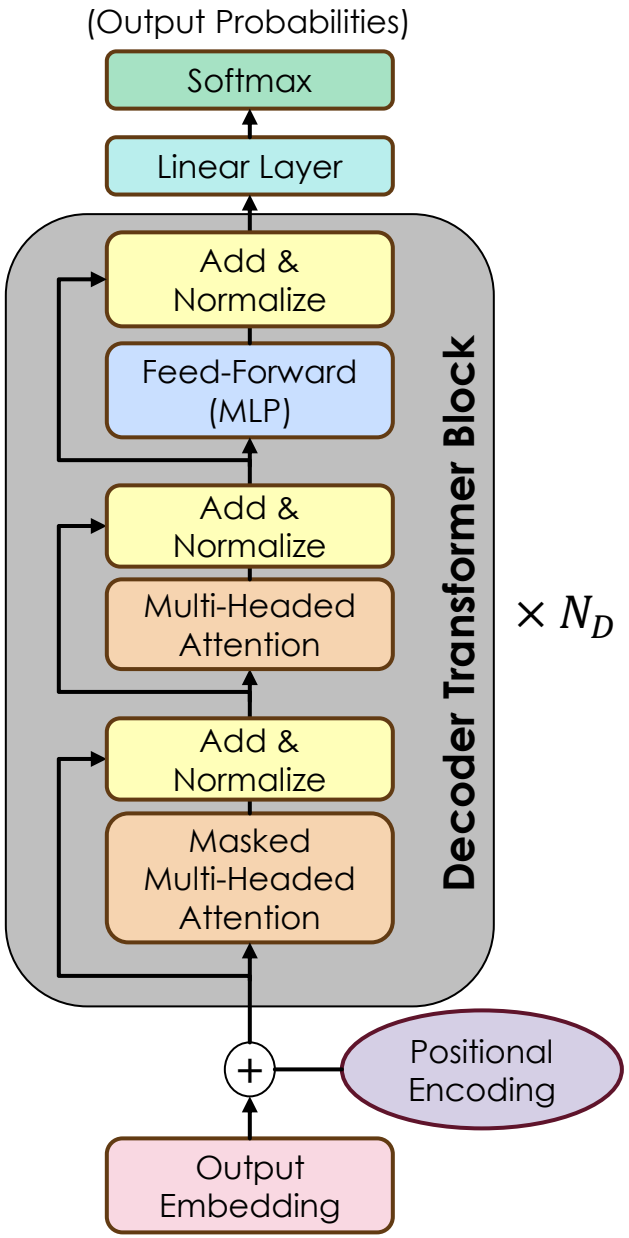
This block is the thing that differentiates the transformer architectures from one another. Specifically, the encoder block simply consists of a multi-headed attention layer followed by a general MLP, each interspersed with residual normalization blocks to aid with numerical stability. These blocks are then "stacked" for a total of N_E times to yield the complete encoder-only model, which generates predictions by feeding the output of the encoder blocks through a final linear fully-connected layer.

Interpretation & Use Cases

Encoder-Only models are essentially performing a series of transformations from an initial input to different embedded spaces, each of which consecutively captures some type of semantic context within the data, which ultimately yields better and more refined "clusterings". The MLP layers are incorporated to enable the model to identify nonlinear patterns to enrich the attention embeddings further.

- **Encoder-only** architectures perform quite well on **typical supervised learning classification / regression problems** (i.e., predicting if an email is spam, image classification problems, sentiment classification of a text, predicting numeric atmospheric parameters of celestial objects, etc.). One of the most famous encoder-only model is the **Bidirectional Encoder Representations from Transformers (BERT)** model (which has many variations: **RoBERTa**, **DistilBERT**, **ALBERT**, etc.).

Decoder-Only Transformer Architectures



The Decoder-Only Architecture

The second of the three main transformer architectures is the **decoder-only architecture**. It again consists of three stages: **(1)** given some input data, properly embed it into some vector space that incorporates positional information (with regards to sequential data), **(2)** some number $N_D \in \mathbb{N}$ of decoder blocks, **(3)** followed by an output layer that returns predicted probabilities (or numeric values).

The Decoder Transformer Block

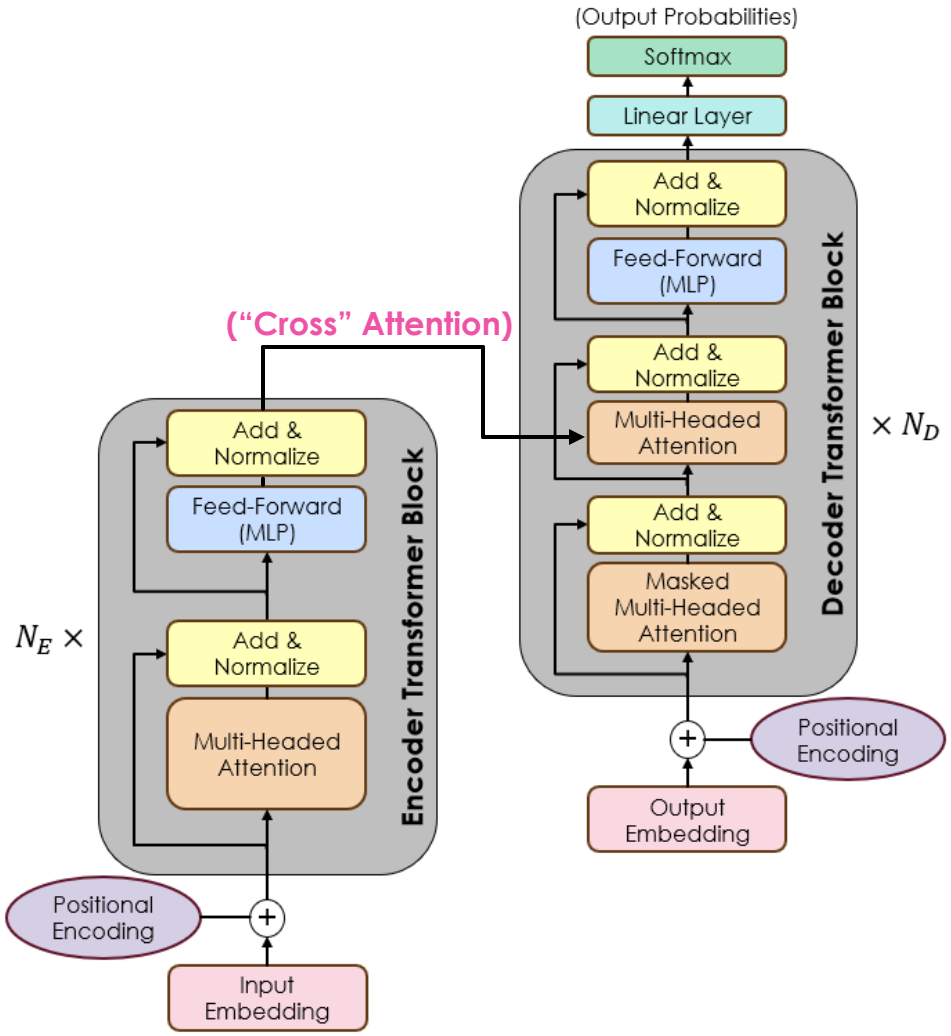
This block is the thing that differentiates the transformer architectures from one another. It consists of the same architecture as in the encoder-only model, only there is a **Masked Multi-Headed Attention** block (this is just the typical multi-headed operation which includes masking of the sequence data) incorporated at the start of the decoder block.

Interpretation & Use Cases

The interpretation of decoder-only models is essentially the exact same as that of encoder-only models.

- The only difference is that, because of the extra masked attention block, **decoder-only** models are preferred (and excel) for **sequence generation problems** (since the model is trained by hiding the future tokens when performing the attention) (i.e., text generation, time-series generations like market predictions, etc.). Famous examples of decoder-only models consist of OpenAI's **ChatGPT** and Google's **Gemini**.

The Encoder-Decoder Transformer Architecture



The Encoder-Decoder Architecture

This architecture was the original "transformer" architecture that was first proposed in the paper "*Attention is All You Need*" and essentially just incorporates some number $N_E \in \mathbb{N}$ of encoder blocks which are then fed into some number $N_D \in \mathbb{N}$ of decoder blocks. This architecture is also referred to as a **sequence-to-sequence** architecture since these models excel at converting sequences of one data type (like a phrase in English) into corresponding sequences of a different data type (like a phrase in Spanish). Thus, you can think of the encoder blocks as dealing with one data type (English inputs) and the decoder blocks as dealing with another type of data (Spanish inputs) with the goal of performing English-to-Spanish conversion. This is achieved through the "cross" attention connection.

- **"Cross" Attention Operation:** The exact same thing as a typical attention operation, only the key and value matrices K and V come from the encoder block and the query matrix Q comes from the decoder block. This allows the data type from the decoder to ask questions about its context, which will be informed by the answers of the keys from the encoder block. This allows the decoder to attend to the encoded representations of the input sequence.

Interpretation & Use Cases

Interpreting an encoder-decoder model is only slightly different from their counterparts. The main idea is that the model is working with two different types of data in tandem and seeing how one relates to the context of the other through cross-attention.

- Encoder-decoder models excel at **sequence conversion problems** (i.e., English-to-Spanish translation, audio-to-text, etc.). Famous examples of encoder-decoder models include Meta's **BART** and Google's **T5**.

References

- Loukadakis, Cano, et al. "Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures", 2018.
- Simonyan, Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition", 2014.
- Vaswani, Shazeer, et al. "Attention is All You Need". arXiv, 2017, https://doi.org/10.48550/arXiv.1706.03762.
- Lecun, Bottou, et al. "Gradient-Based Learning Applied to Document Recognition". 1998, doi: 10.1109/5.726791.
- Krizhevsky, Sutskever, et al. "ImageNet Classification with Deep Convolutional Neural Networks". 2015, https://doi.org/10.1145/3065386.
- He, Zhang, et al. "Deep Residual Learning for Image Recognition". arXiv, 2015, https://doi.org/10.48550/arXiv.1512.03385
- Cybenko. "Approximation by Superpositions of a Sigmoidal Function". 1989.
- Hornik, Stinchcombe, et al. "Multilayer feedforward networks are universal approximators". 1989.
- https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1
- https://www.javatpoint.com/pytorch-convolutional-neural-network
- LeCun, Cortes, et al. "The MNIST Database of handwritten digits." https://yann.lecun.com/exdb/mnist/