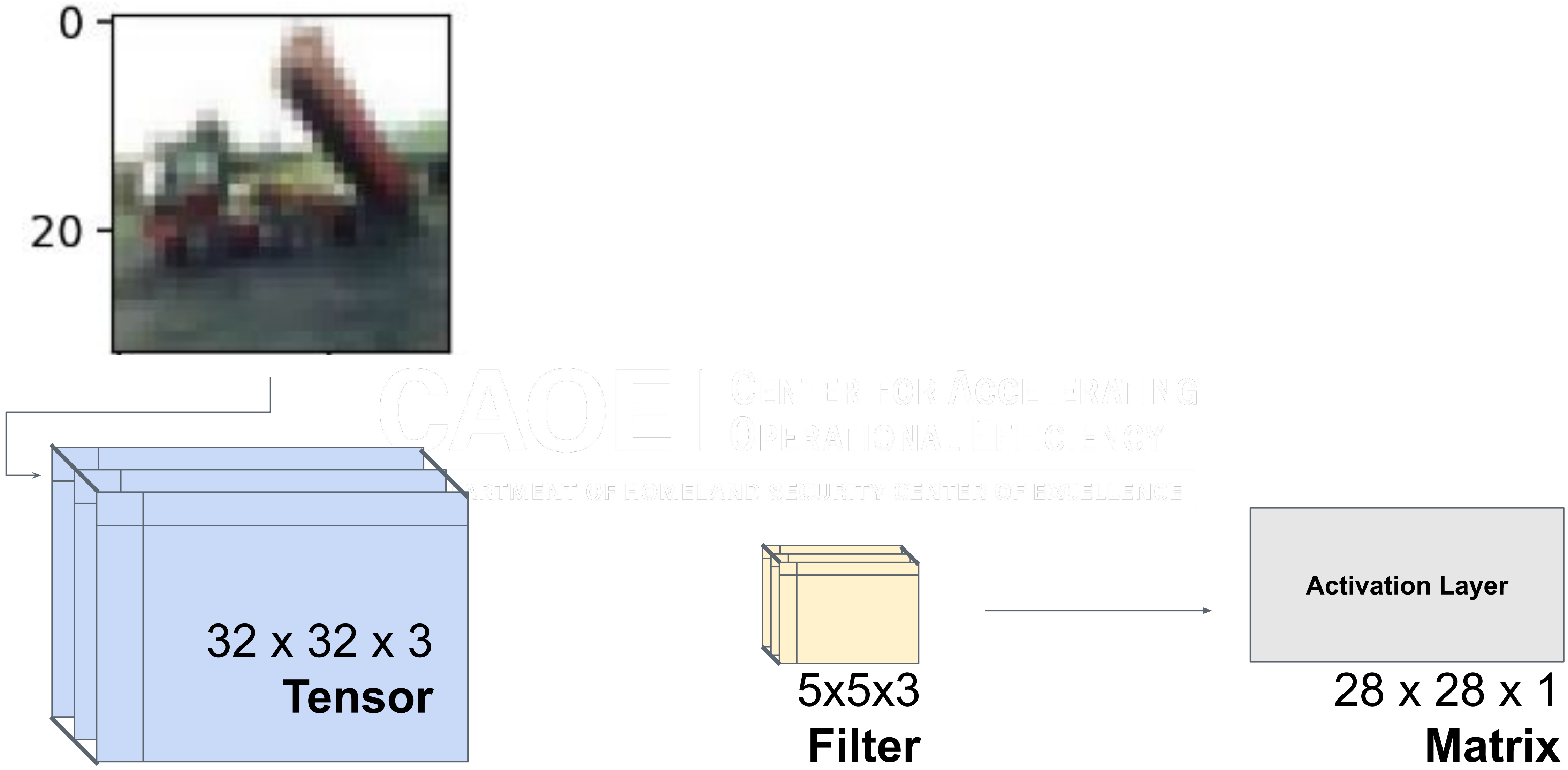
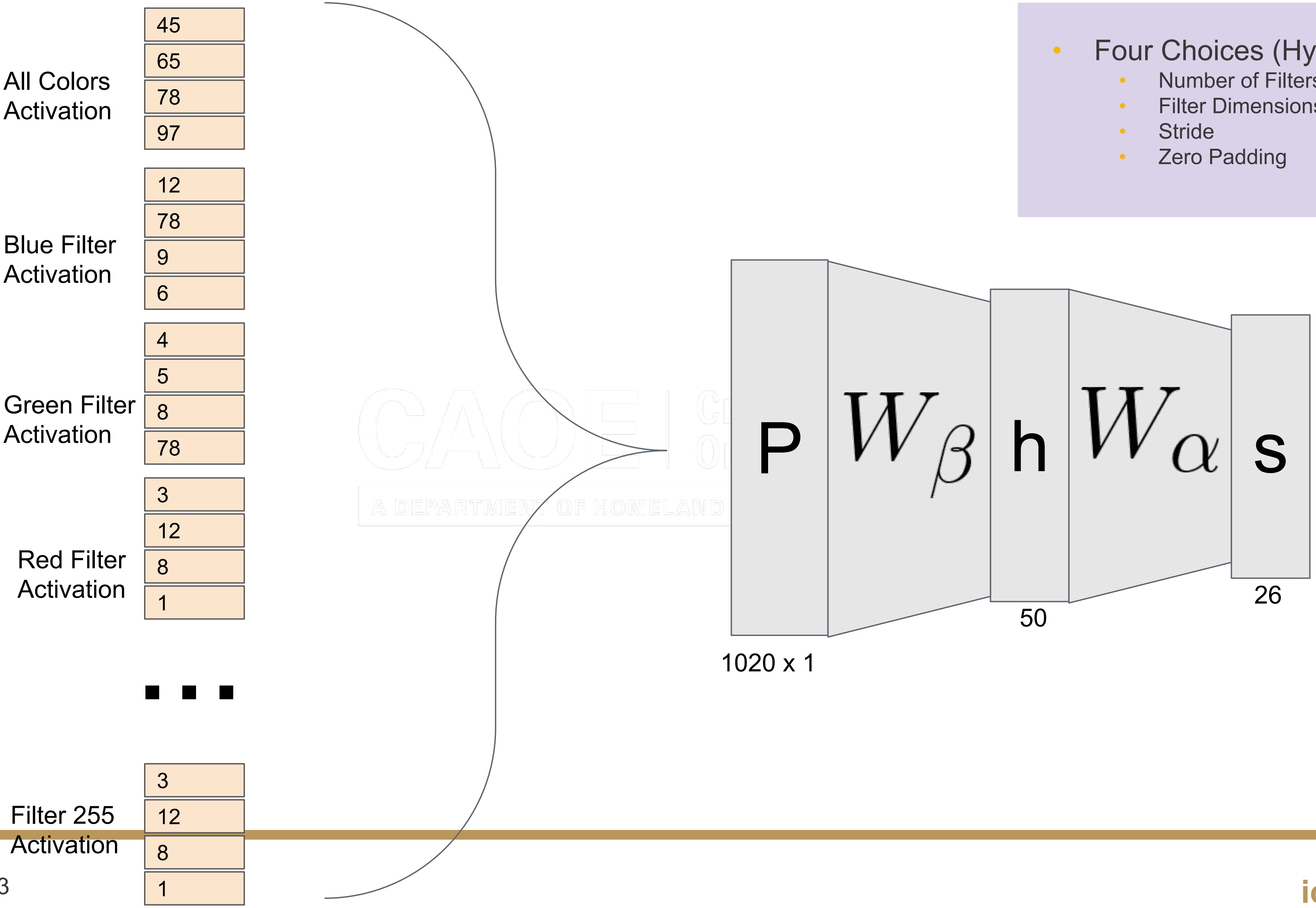
# DATA 442: Neural Networks & Deep Learning

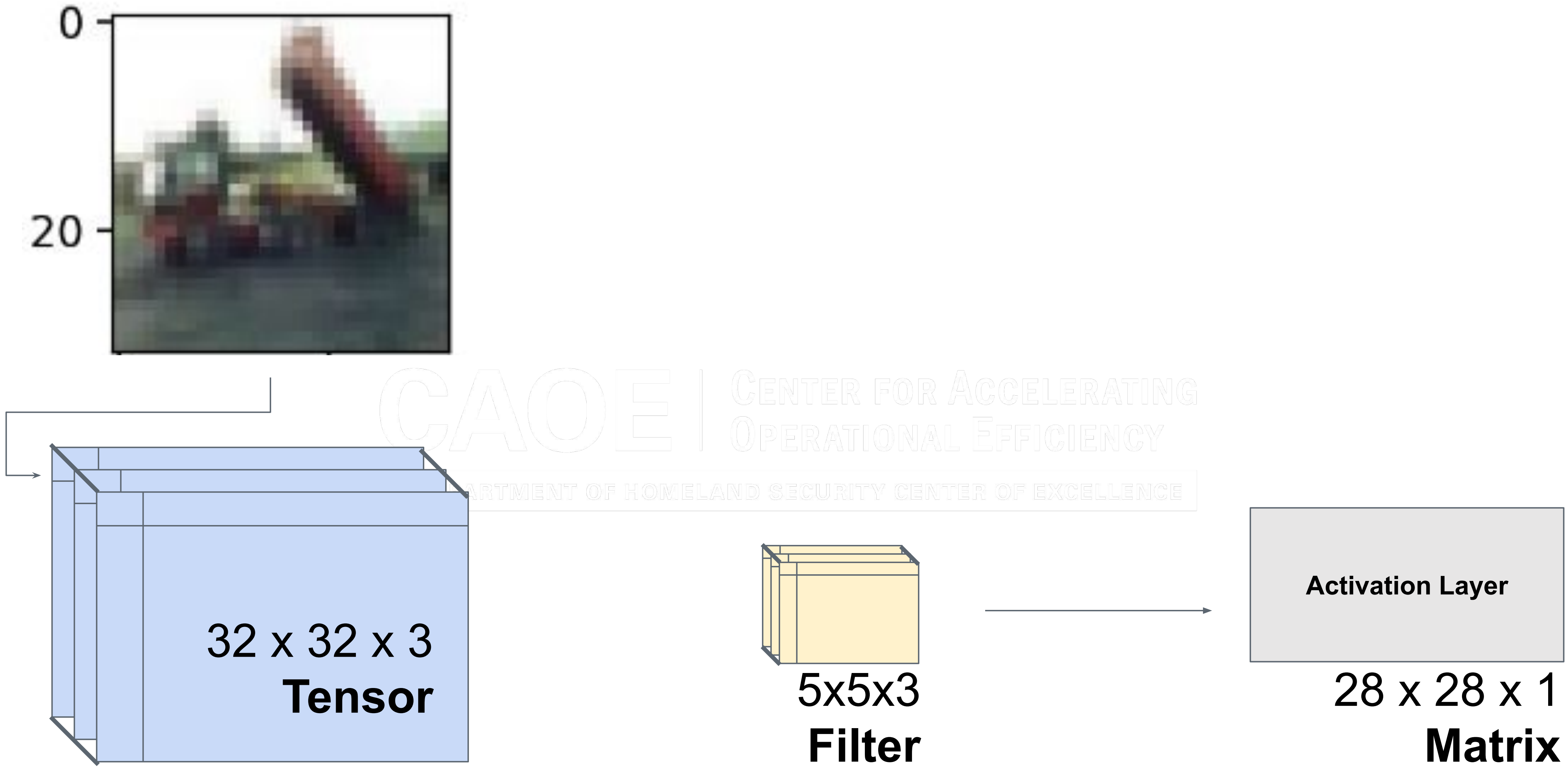
Dan Runfola – danr@wm.edu

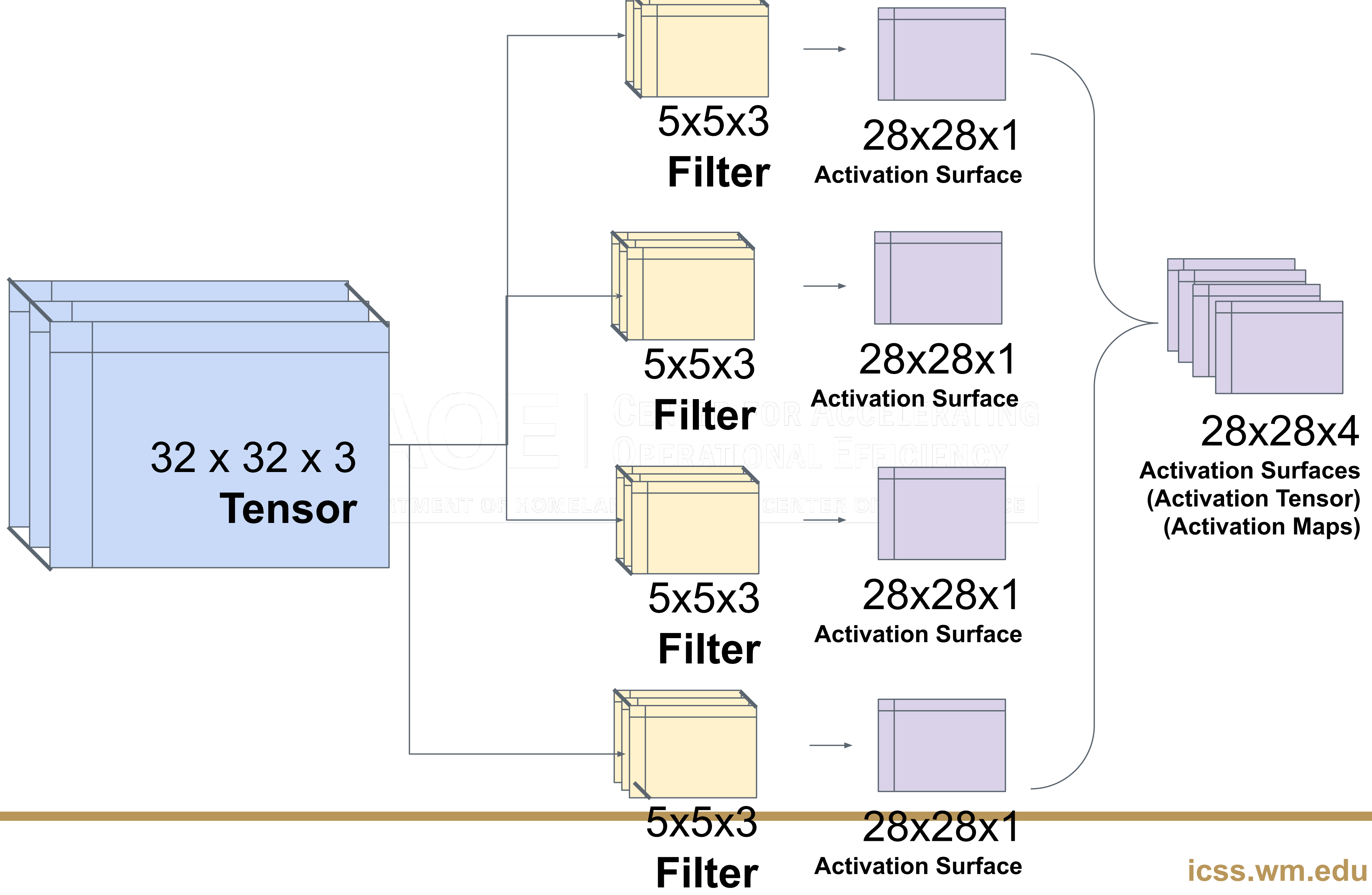
icss.wm.edu/data442/


0
20
32 x 32 x 3
Tensor
5x5x3
Filter
Activation Layer
28 x 28 x 1
Matrix

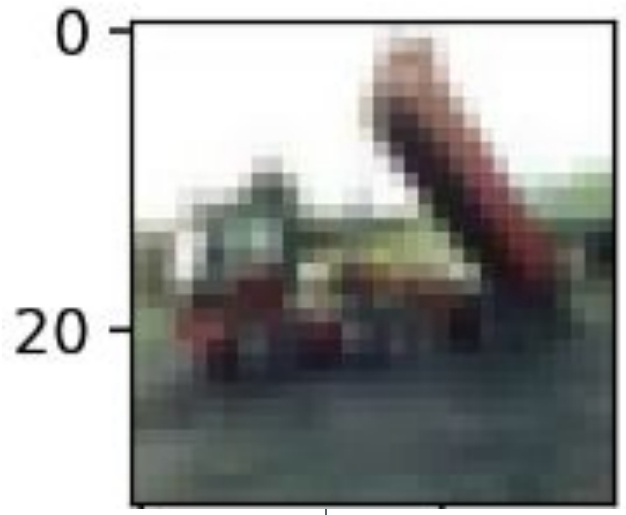
- Four Choices (Hyperparameters)
  - Number of Filters
  - Filter Dimensions
  - Stride
  - Zero Padding




0
20
Activation Layer
32 x 32 x 3
Tensor
5x5x3
Filter
28 x 28 x 1
Matrix



5x5x3
Filter
28x28x1
Activation Surface
5x5x3
Filter
28x28x1
Activation Surface
32 x 32 x 3
Tensor
5x5x3
Filter
28x28x1
Activation Surface
5x5x3
Filter
28x28x1
Activation Surface
28x28x4
Activation Surfaces
(Activation Tensor)
(Activation Maps)

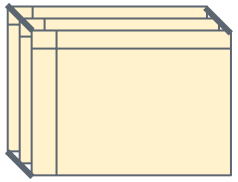



Convolution (5x5x3 Filter)

Activation Layer (28 x 28 x 1 Matrix)

Input Activation Array

784 × 1 ● Weights for each Pixel, for each of 10 CIFAR classes (784 × 10) = Output Scores Array (10 × 1)



5x5x3
**Filter**

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 1 | 8 | 6 |
| 9 | 8 | 5 | 4 | 3 |
| 1 | 6 | 4 | 3 | 1 |
| 1 | 5 | 8 | 3 | 1 |
| 1 | 2 | 2 | 1 | 1 |

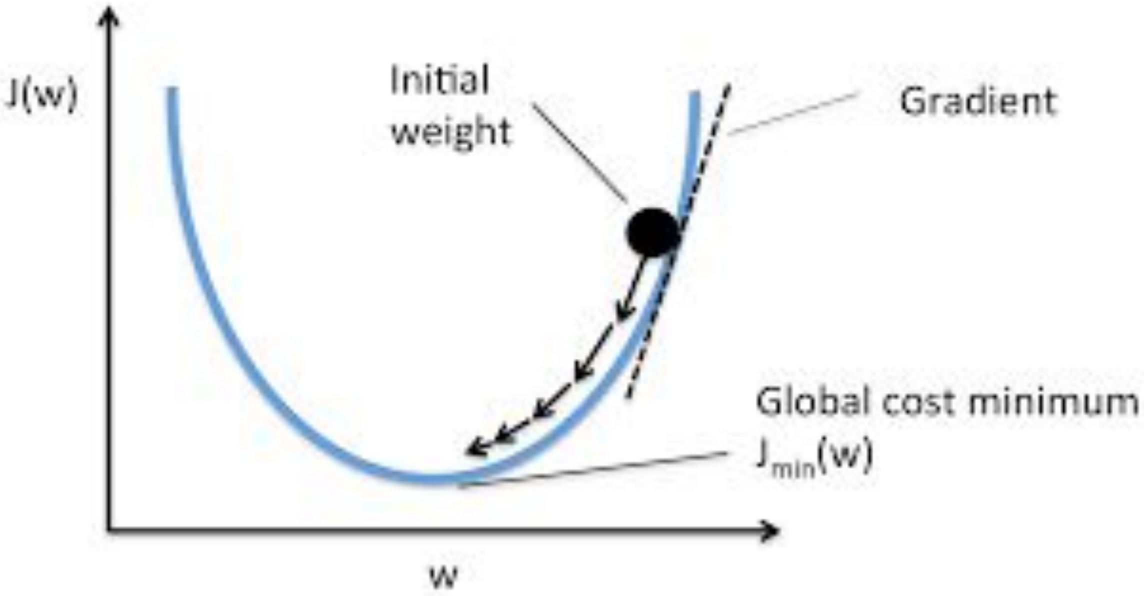
| | | | | |
|---|---|---|---|---|
| 5 | 8 | 4 | 2 | 0 |
| 6 | 6 | 4 | 5 | 1 |
| 9 | 8 | 9 | 1 | 1 |
| 7 | 6 | 5 | 0 | 3 |
| 5 | 1 | 1 | 0 | 2 |

| | | | | |
|---|---|---|---|---|
| 5 | 6 | 8 | 7 | 7 |
| 2 | 1 | 1 | 1 | 4 |
| 2 | 5 | 2 | 5 | 5 |
| 3 | 5 | 2 | 3 | 5 |
| 4 | 5 | 2 | 3 | 5 |

# Optimization

**Goal**: Find the best weights parameters to minimize a loss function.

**Approaches we've discussed**: Gradient Descent, Stochastic Gradient Descent, Mini-batch SGD.




# Optimization

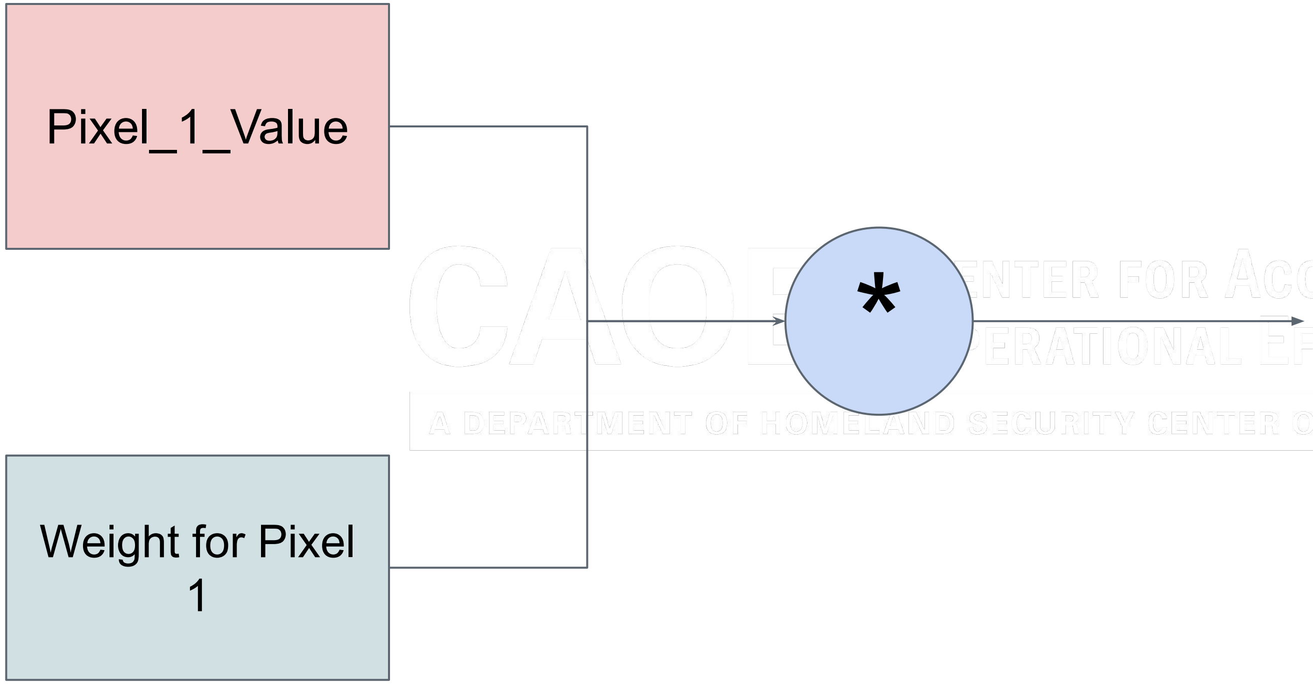
**Example (Mini-batch SGD):**

1. Sample your data (batch size)
2. Run a forward propagation through your network.
3. Calculate your loss
4. Backpropogate to calculate gradients of weights with respect to loss.
5. Update weights using the gradient.
6. Repeat until some threshold is reached (i.e., number of iterations).

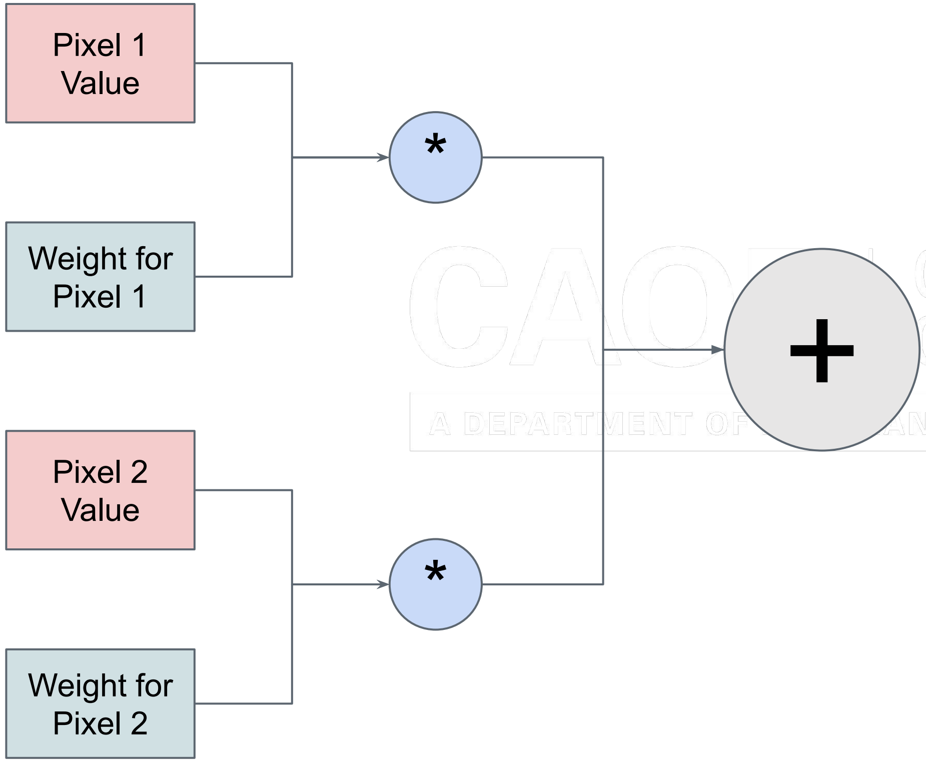
# Building and Optimizing a Neural Network

- Define Network Architecture (Computational Graph)
- Train / Optimize the Network
- Evaluation

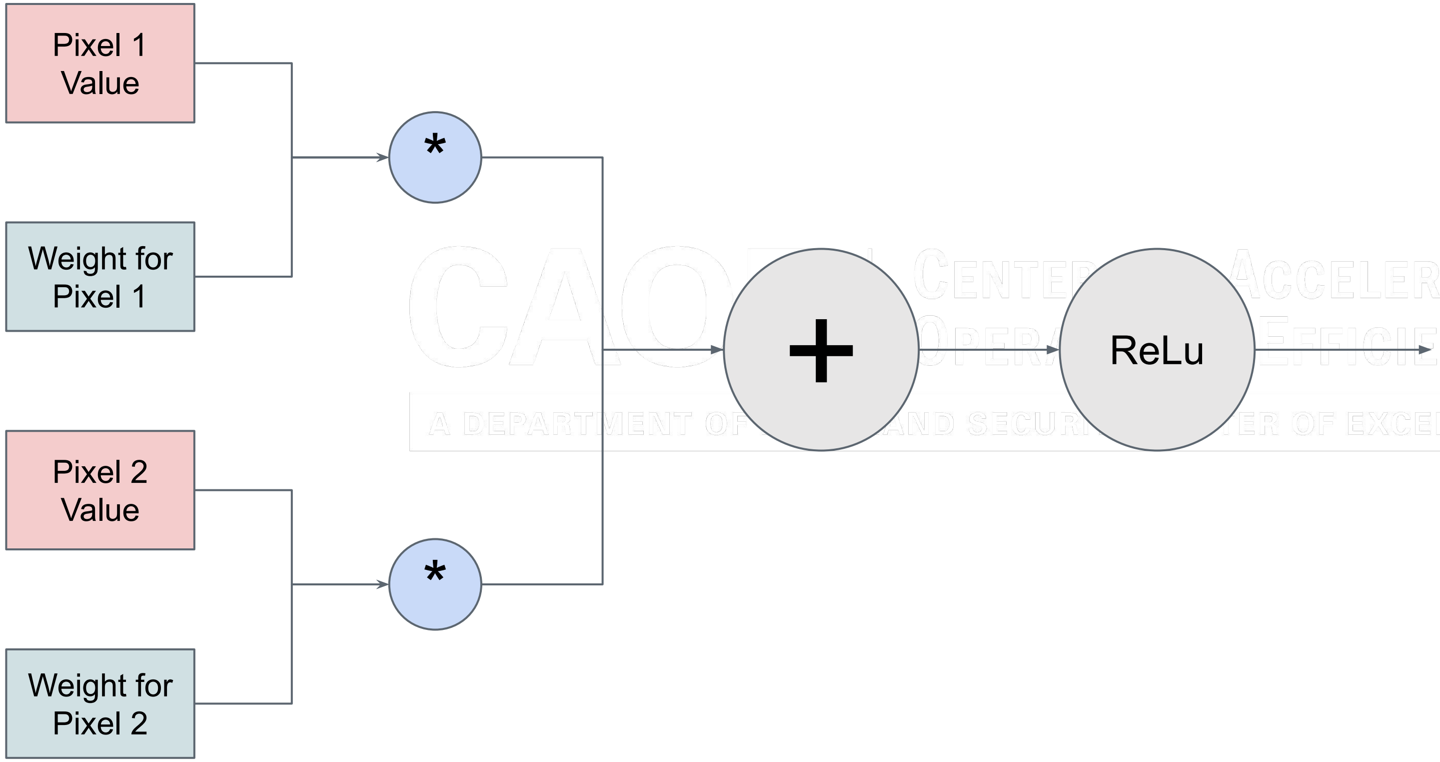
# Network Architecture: Fundamentals



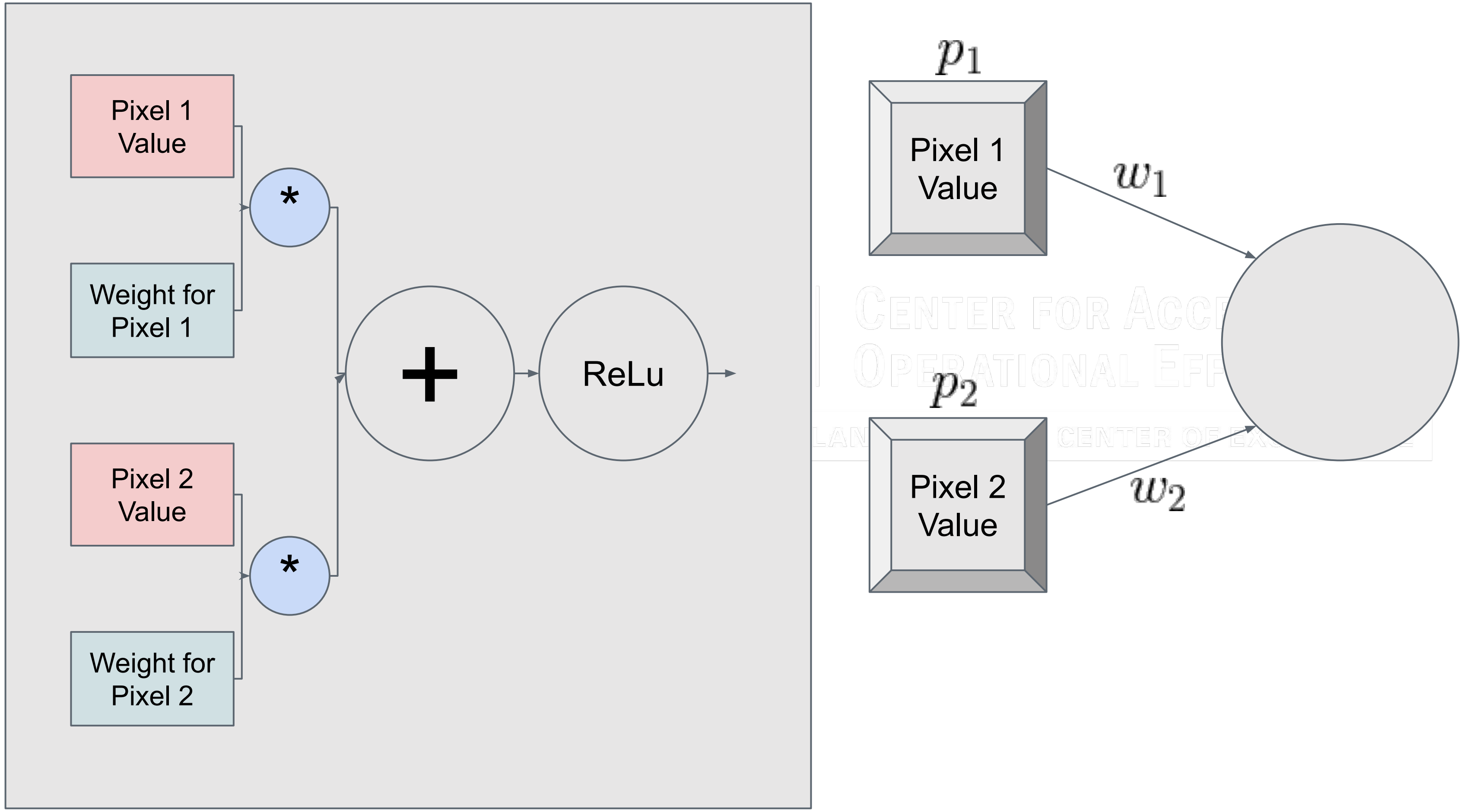
# Network Architecture: Fundamentals



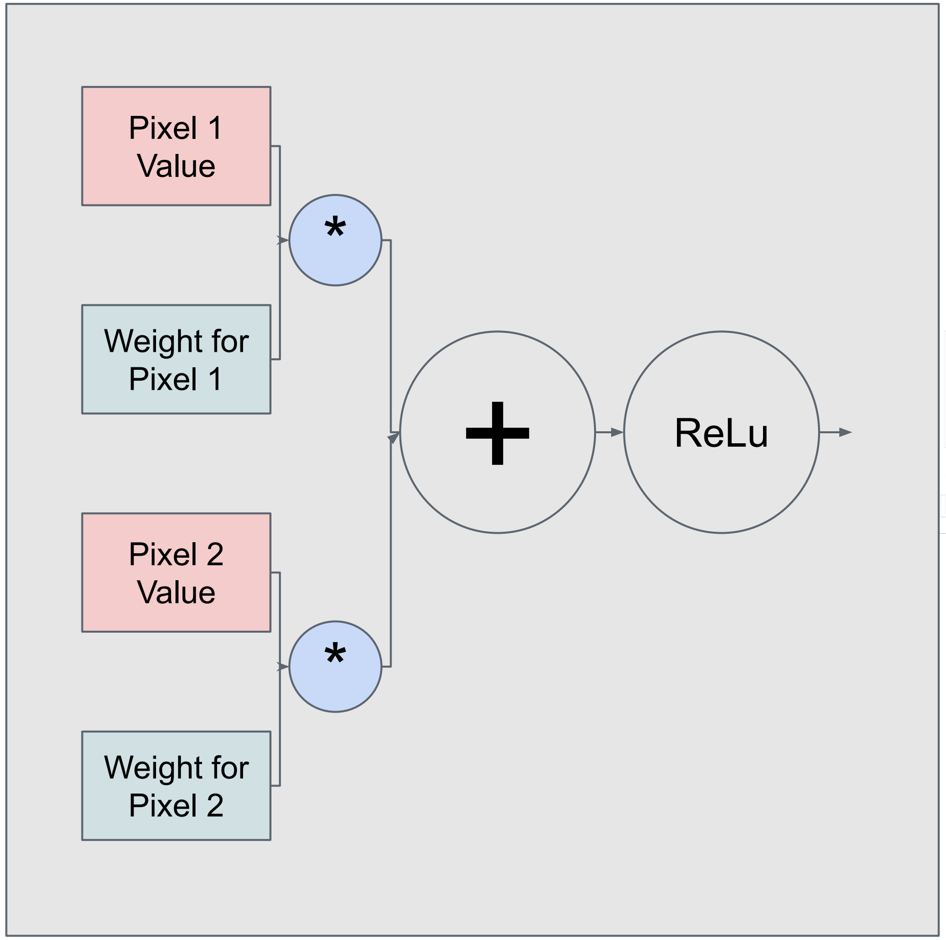
# Network Architecture: Fundamentals

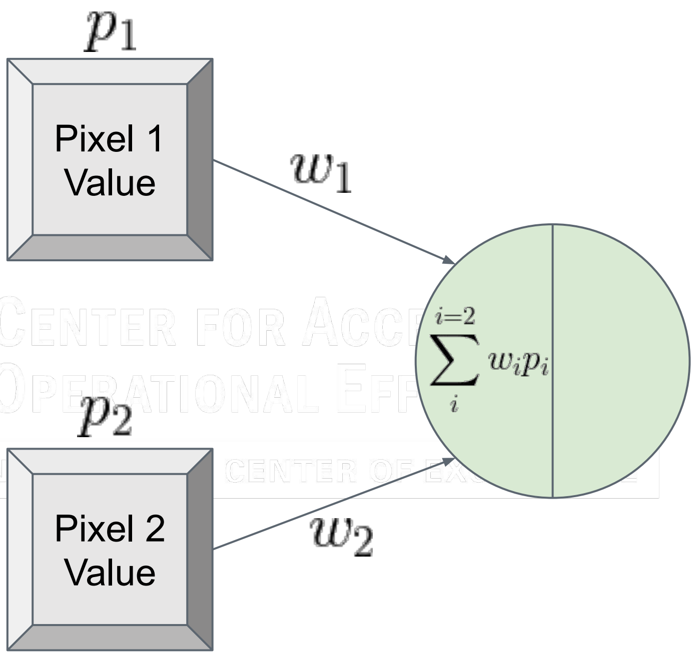


# Network Architecture: Fundamentals

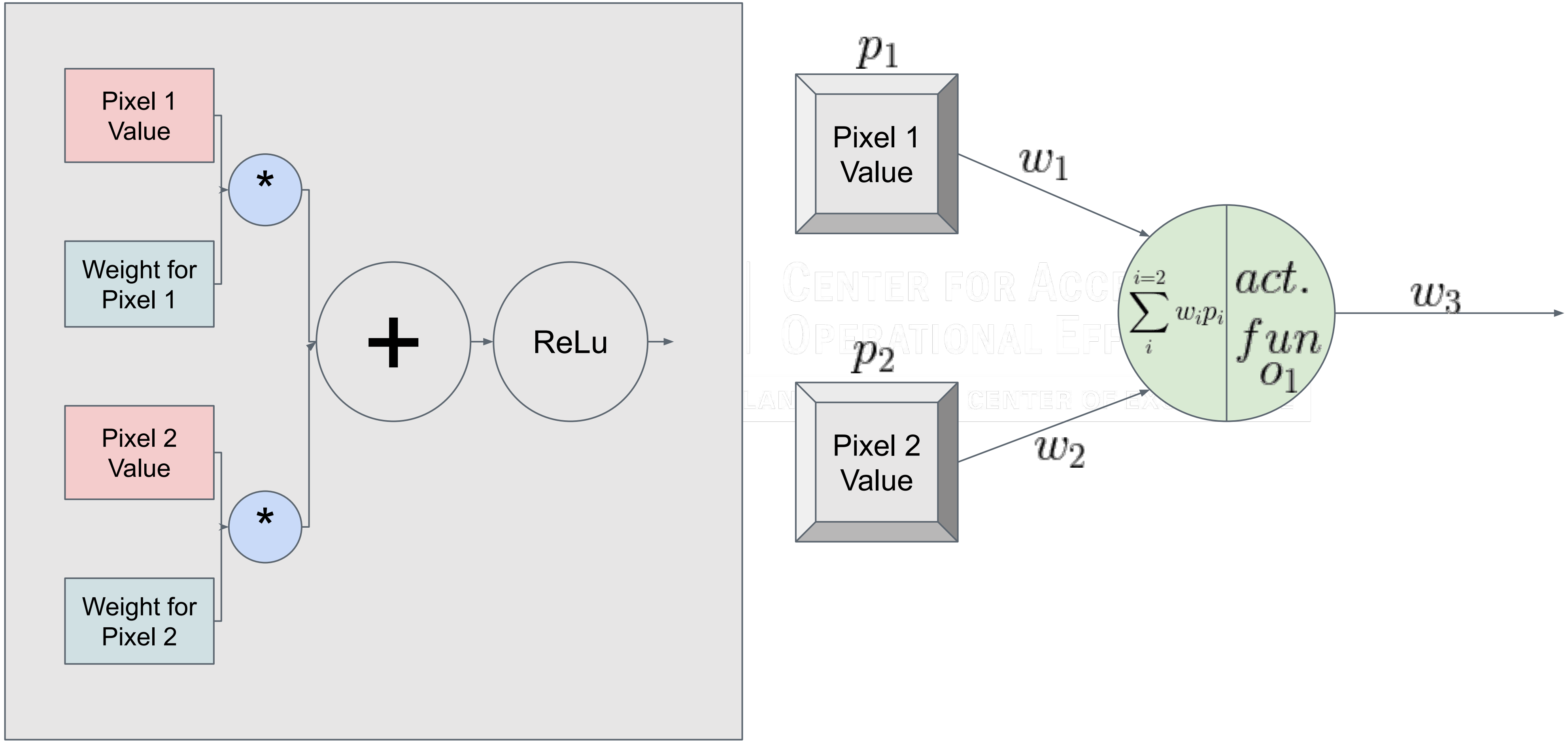


# Network Architecture: Fundamentals

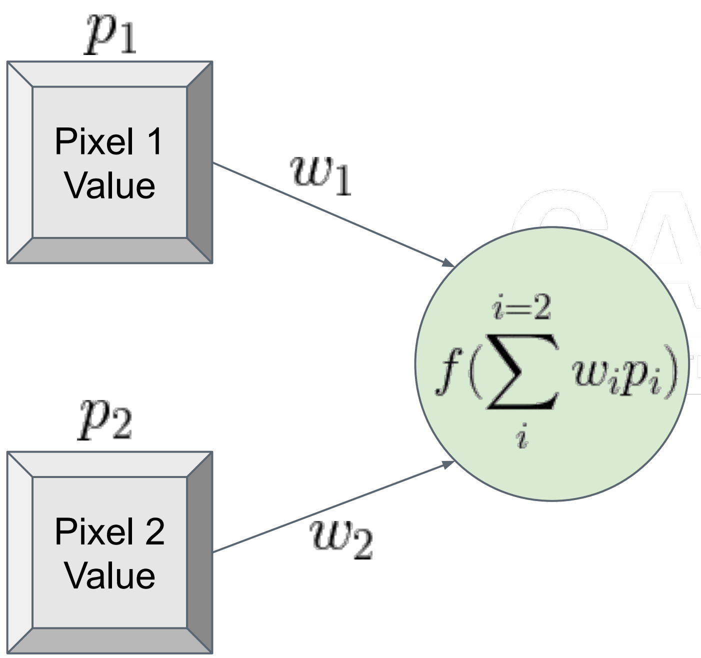


# Network Architecture: Fundamentals



# Network Architecture: Activation Function

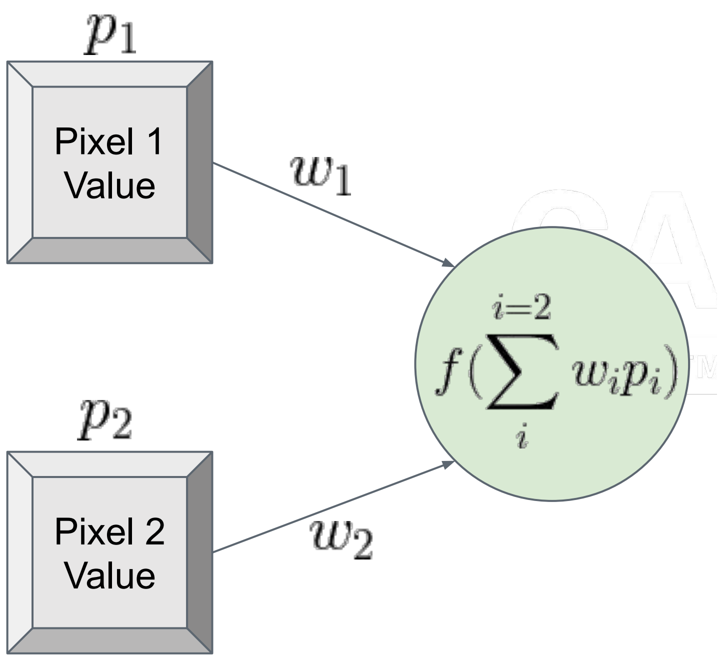


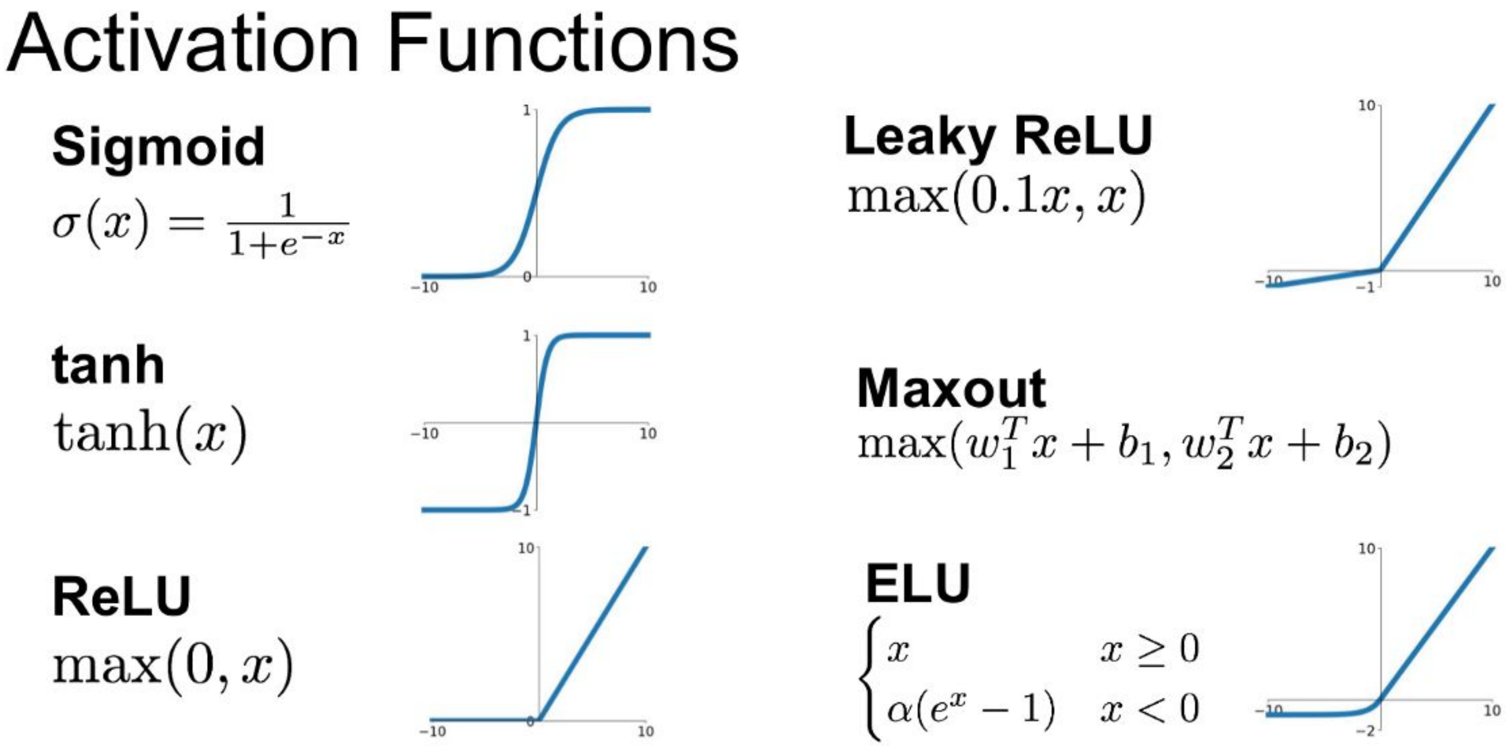

$$f(\sum_{i}^{i=2} w_i p_i)$$



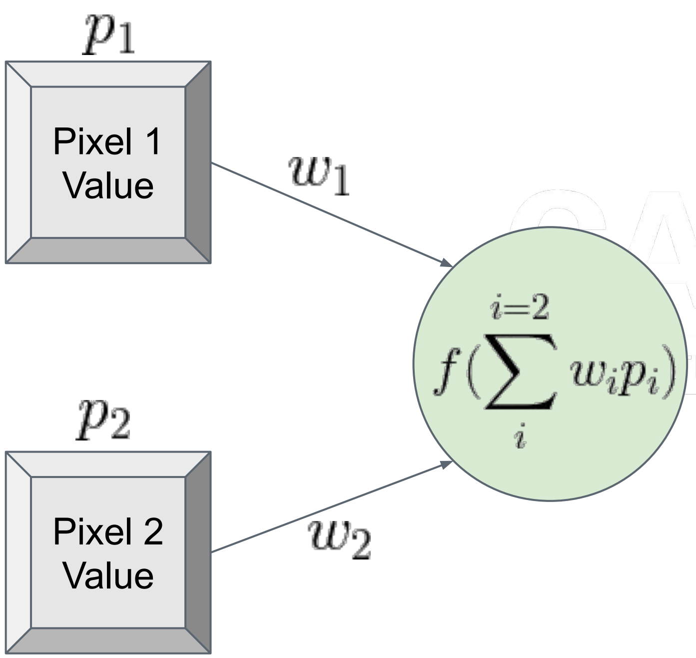
```
def activationFunction(input):
    return(input * 1)
```

# Network Architecture: Activation Function

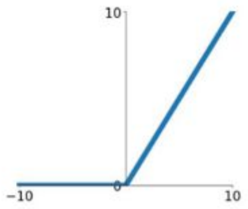


# Network Architecture: Activation Function



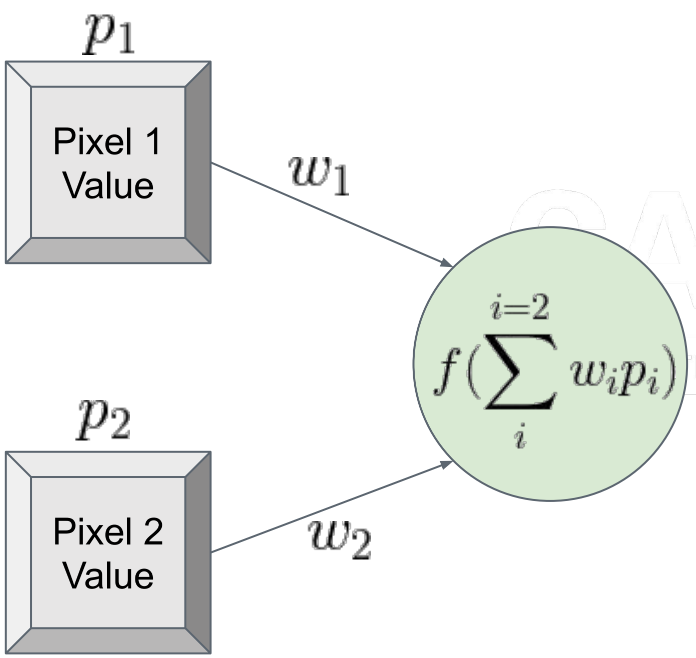

**ReLU**
$\max(0, x)$

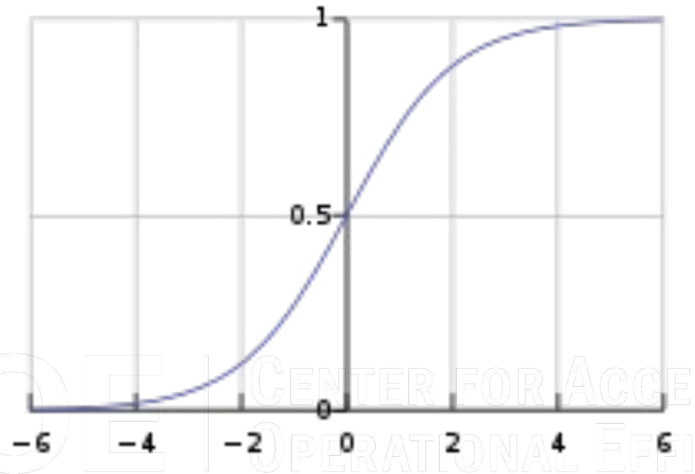





```
def activationFunction(input):
    return(max(input,0))
```

# Sigmoid Activation Function



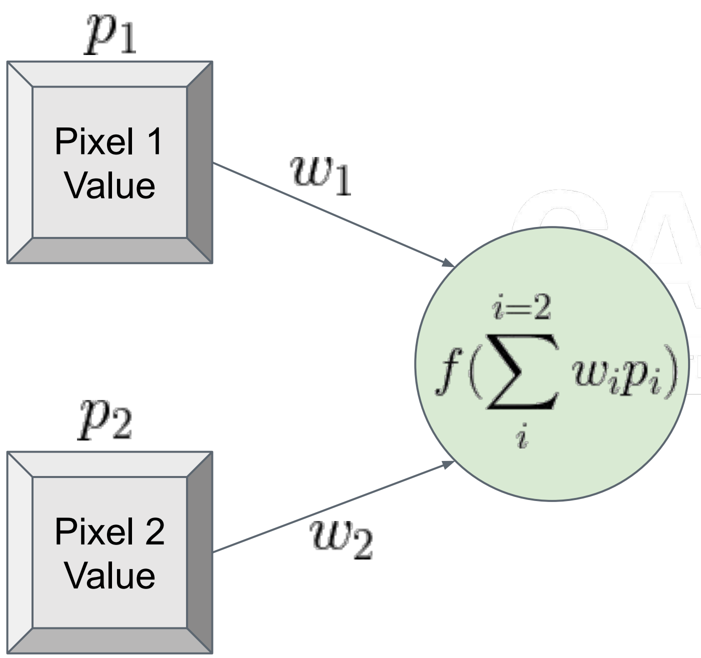


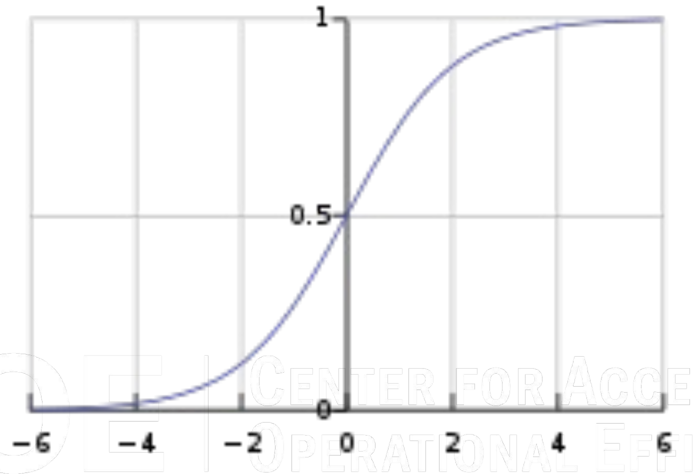

$$\sigma(x) = \frac{1}{(1 + e^{-x})}$$

## Features

- Output of function falls between 0 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.

# Sigmoid Activation Function




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

## Features

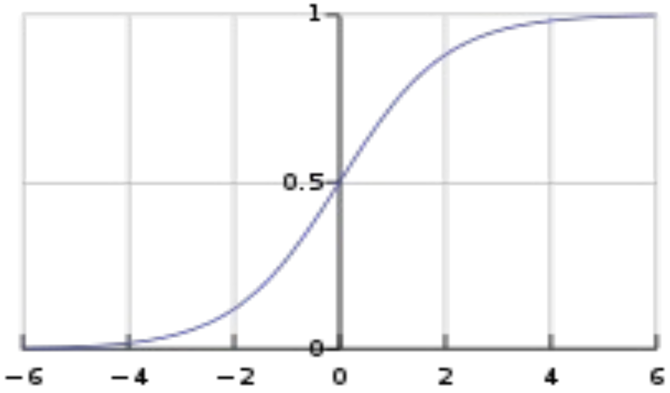
- Output of function falls between 0 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.

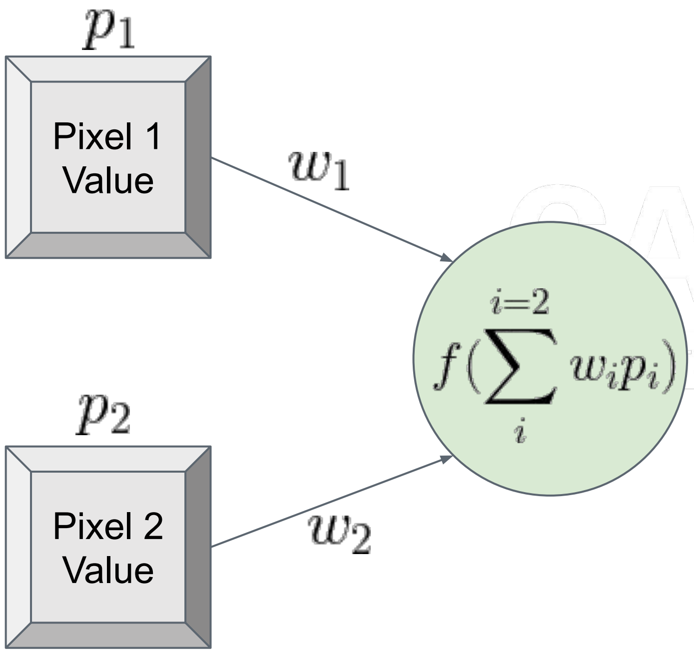
## Challenges

- Gradient Decay & Saturation
- Not Zero-Centered & Unidirectional Gradient Solutions

# Sigmoid Activation Function

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




Consider if pixel 1 value and pixel 2 value are both 1, and both weights are 10.

What would a change in the weight of -1 do to the sigmoid activation function?

## Features

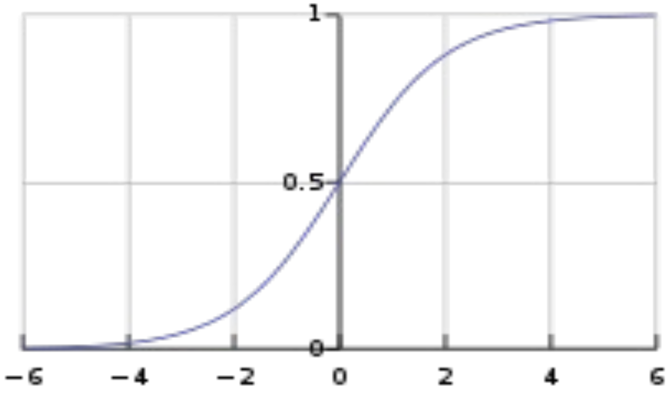
- Output of function falls between 0 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.

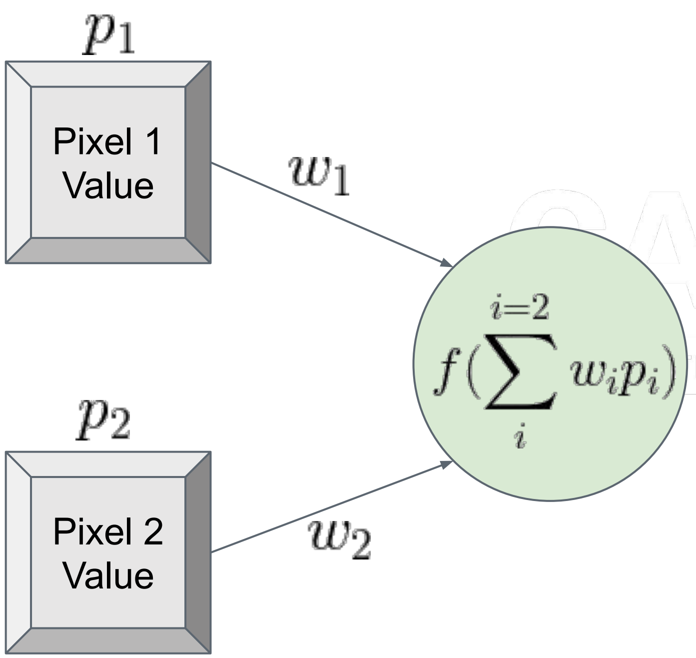
## Challenges

- Gradient Decay & Saturation
- Not Zero-Centered & Unidirectional Gradient Solutions

# Sigmoid Activation Function

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






Consider if pixel 1 value and pixel 2 value are both 1, and both weights are 10.

What would a change in the weight of -1 do to the sigmoid activation function?

**Nothing**
**(the gradient would be 0)**

## Features

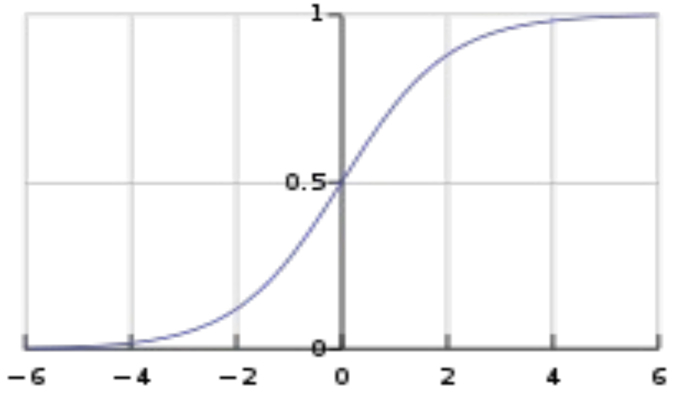
- Output of function falls between 0 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.

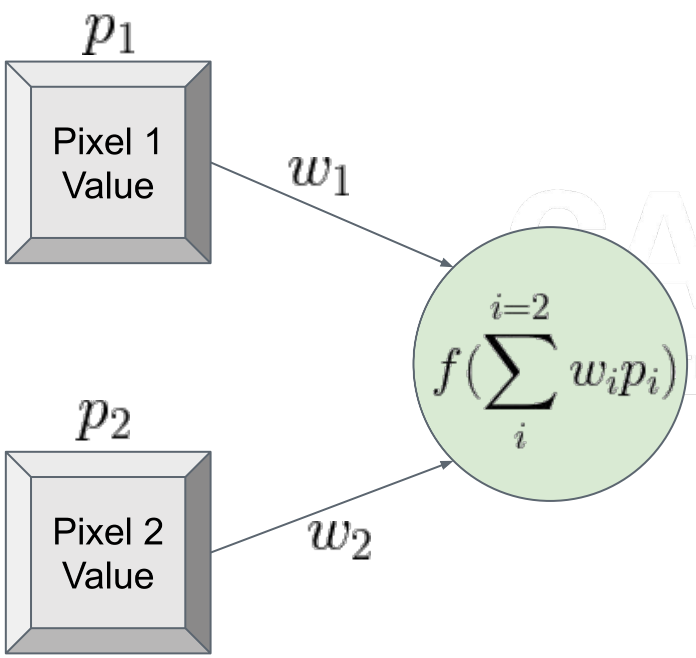
## Challenges

- Gradient Decay & Saturation
- Not Zero-Centered & Unidirectional Gradient Solutions

# Sigmoid Activation Function

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




Now consider the directionality of the gradient. If all of your inputs into a given neuron are positive, then gradients will all always be positive or negative - no mixing of positive and negative gradients during back propagation.

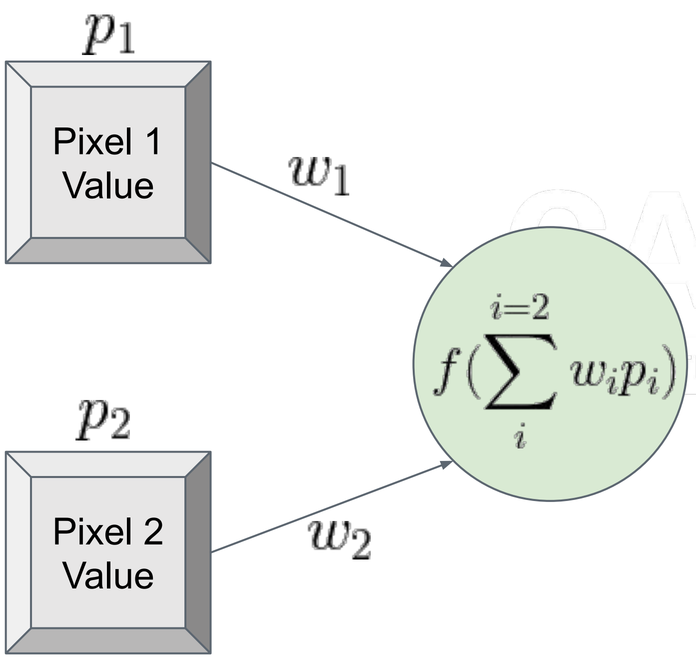
## Features

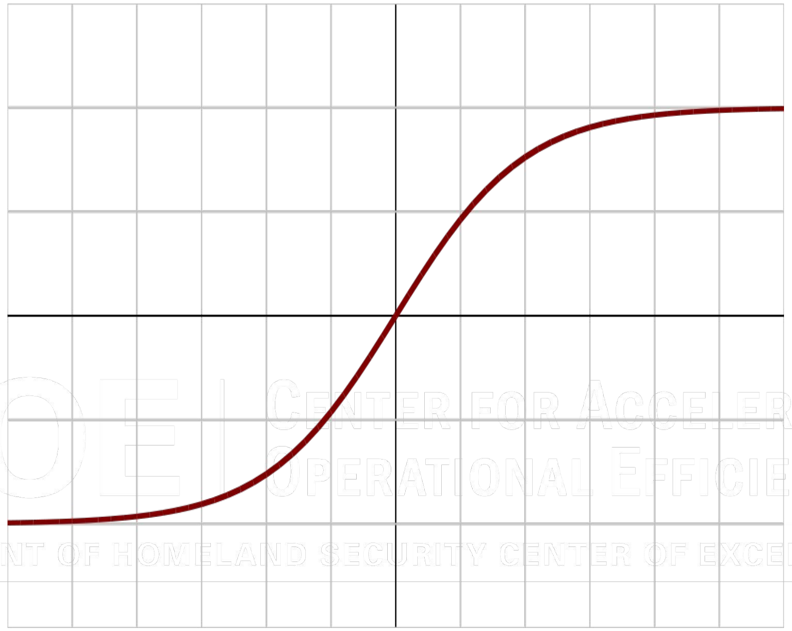
- Output of function falls between 0 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.

## Challenges

- Gradient Decay & Saturation
- Not Zero-Centered & Unidirectional Gradient Solutions

# tanh Activation Function





## Features

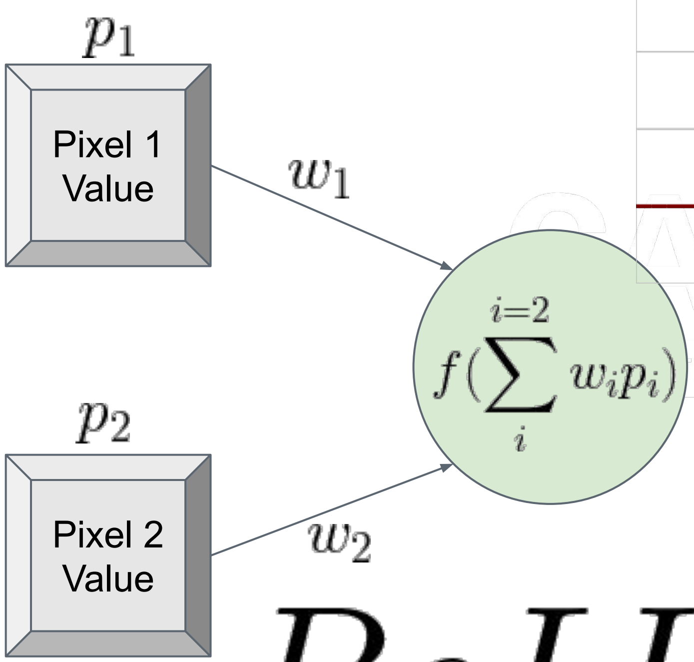
- Output of function falls between -1 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.
- Zero Centered

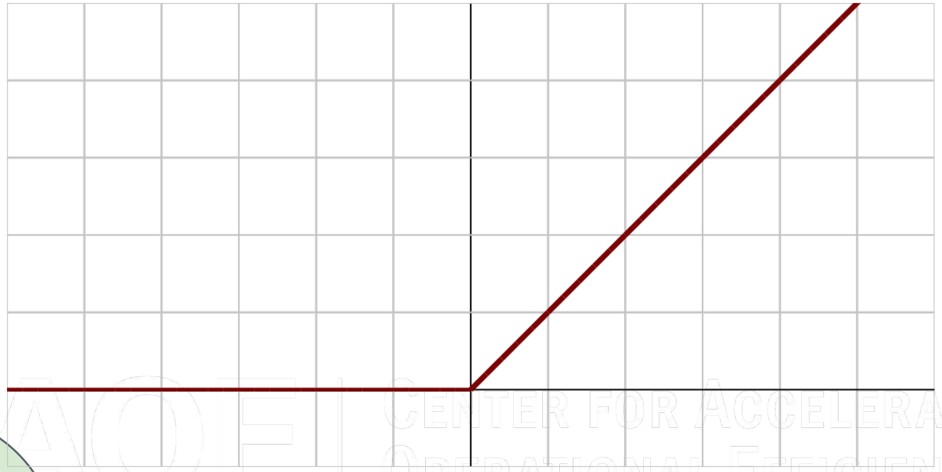
## Challenges

- Gradient Decay & Saturation

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

# Rectified Linear Unit (ReLU) Activation Function



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

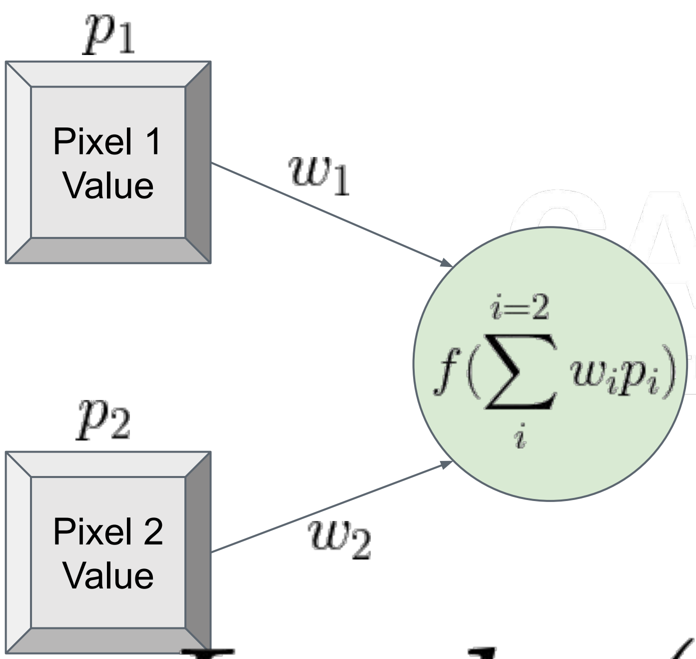
## Features

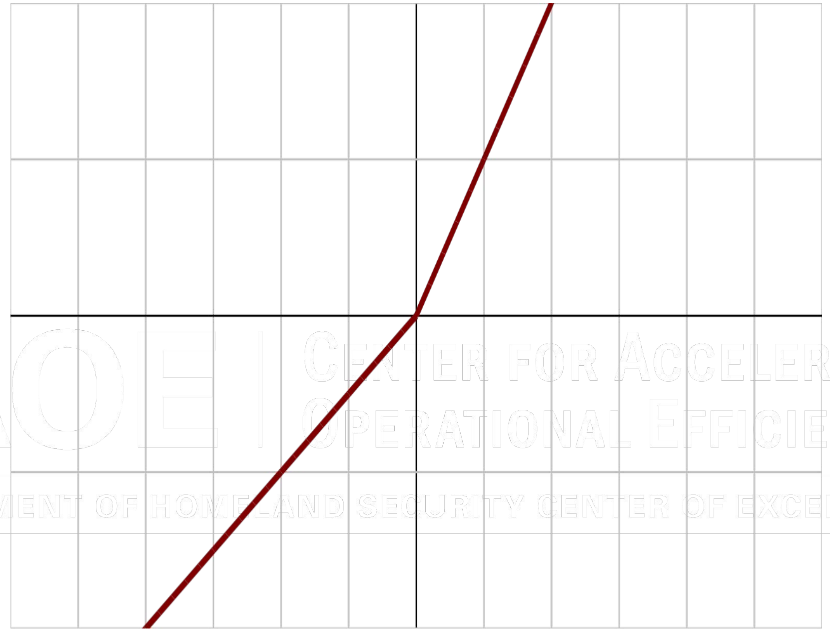
- No saturation in positive direction
- Very, very simple (and, thus, computationally efficient)
- Roughly approximates how a neuron works - 0 values until 0 is reached, then x.

## Challenges

- Not zero-centered
- Gradient Decay / Saturation if X < 0

# Leaky ReLU Activation Function



## Features

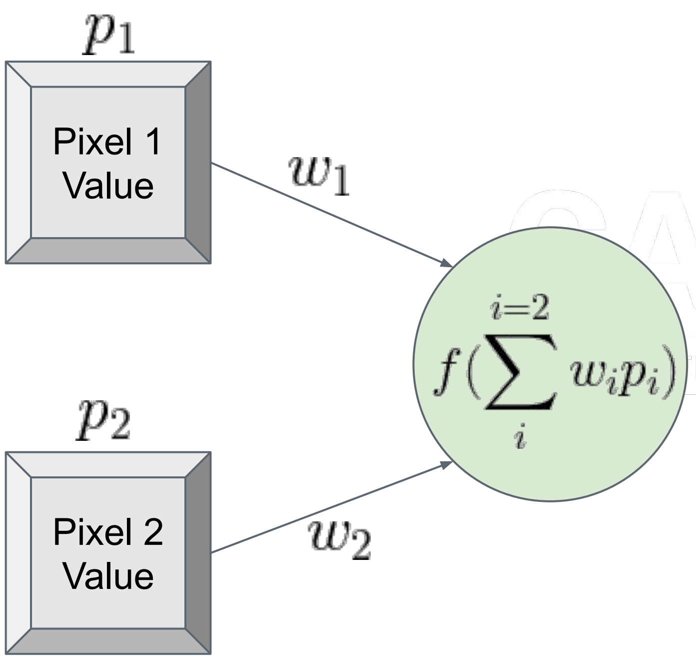
- No saturation (and, thus, no ReLU “death”).
- Still very simple (and, thus, computationally efficient)
- Roughly approximates how a neuron works - small values until 0 is reached, then x.

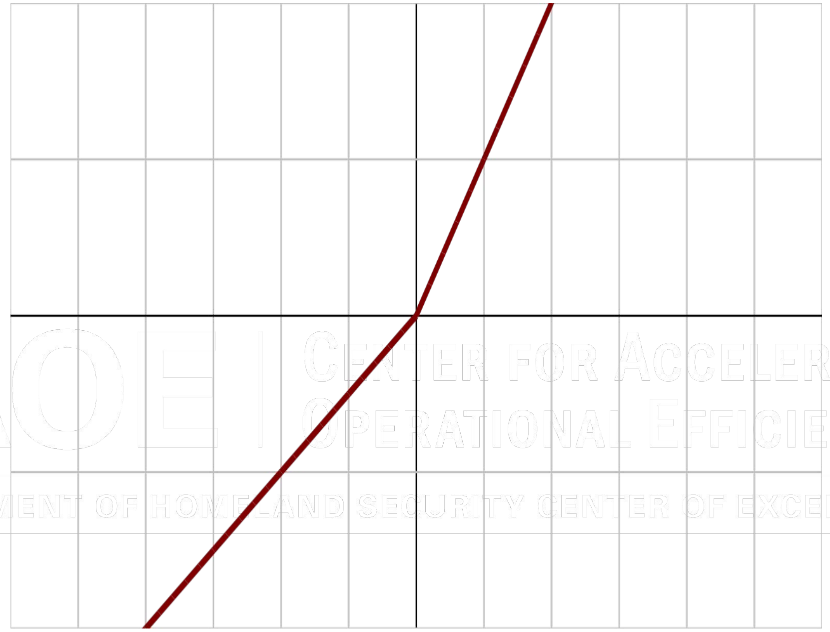
## Challenges

- Not zero-centered

$$Leaky(x) = max\{0.01X, X\}$$

# Parametric ReLU Activation Function





## Features

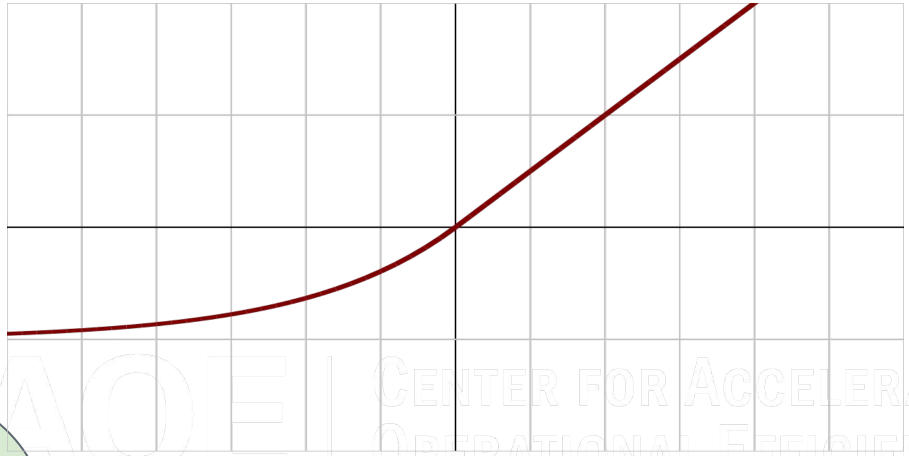
- No saturation (and, thus, no ReLU "death").
- Still very simple (and, thus, computationally efficient)
- Roughly approximates how a neuron works - small values until 0 is reached, then x.
- Parameterized, and can be fit during optimization.

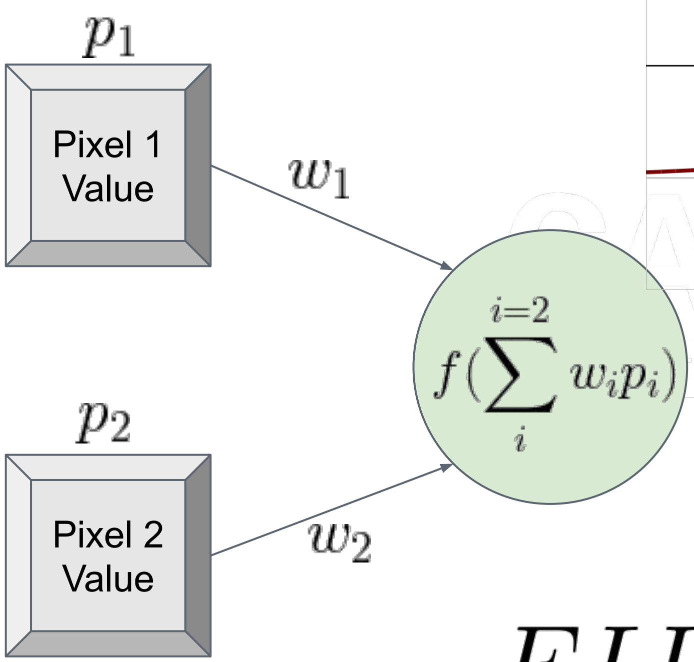
## Challenges

- Not zero-centered

$$PReLU(x) = max\{\alpha * X, X\}$$

# Exponential Linear Units (ELU) Activation Function








## Features

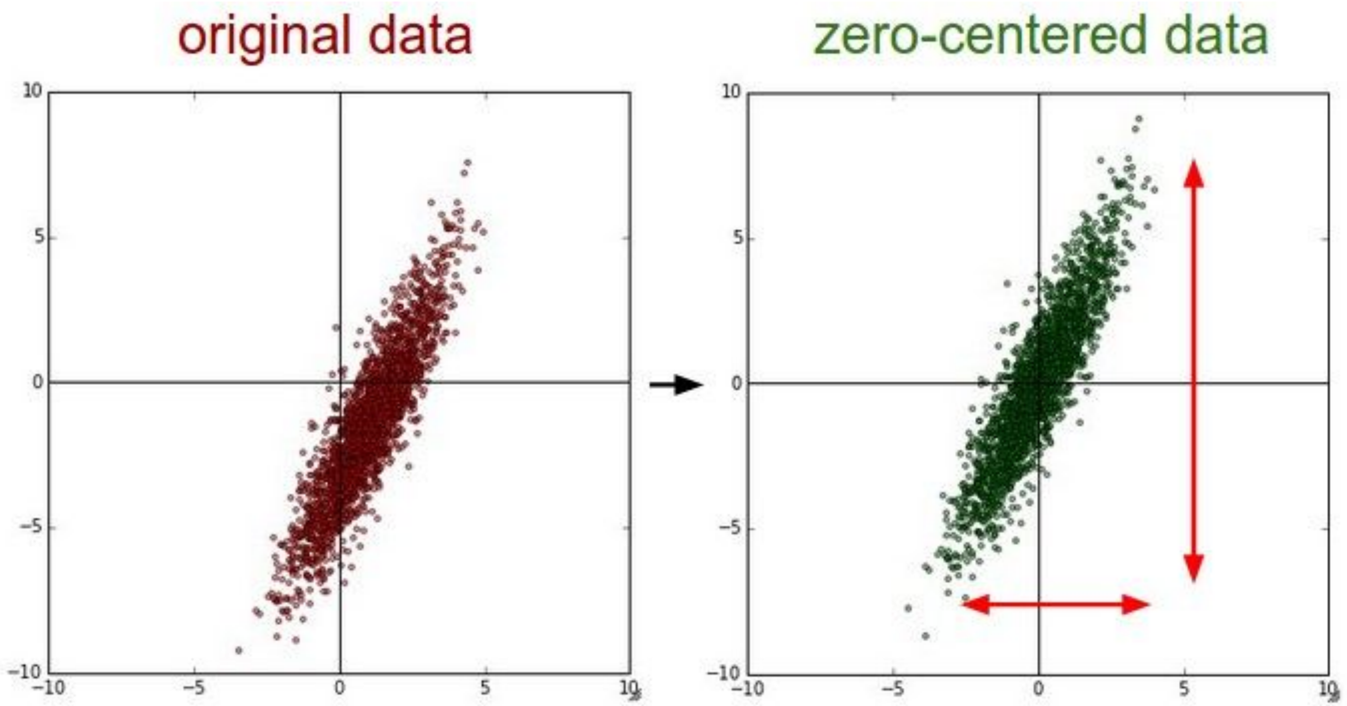
- No saturation if x > 0
- Roughly approximates how a neuron works - small values until 0 is reached, then x.
- Parameterized, and can be fit during optimization.
- Close to mean centered.

## Challenges

- Potential for saturation if x < 0.
- Not actually zero-centered, though it is much closer.

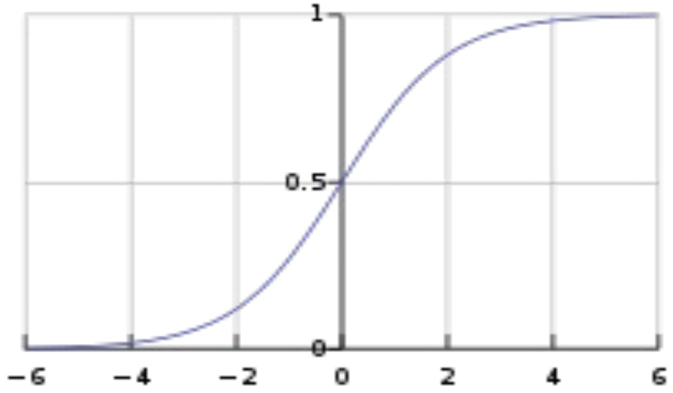
$$ELU(x) = \begin{cases} x & \text{if x} > 0 \\ \alpha(exp(x) - 1) & \text{if x} \leq 0 \end{cases}$$

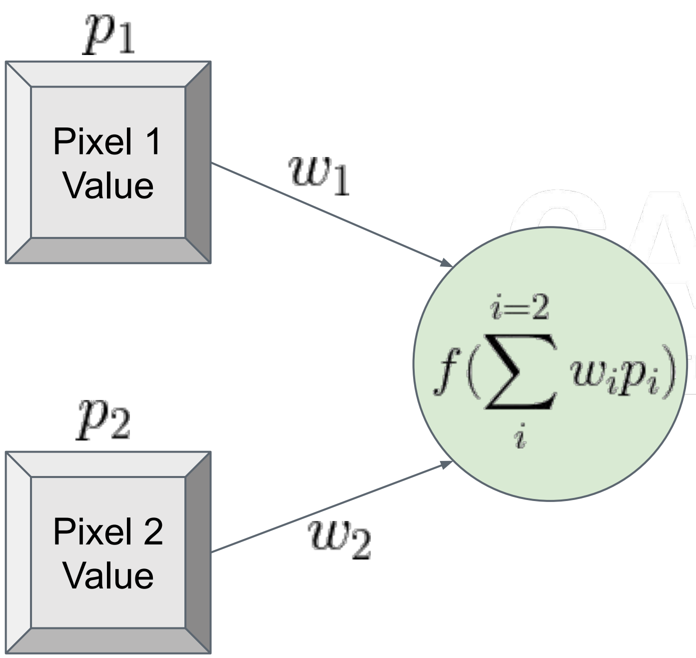
# Network Architecture: Data Preprocessing



# Sigmoid Activation Function

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






Now consider the directionality of the gradient. If all of your inputs into a given neuron are positive, then gradients will all always be positive or negative - no mixing of positive and negative gradients during back propagation.

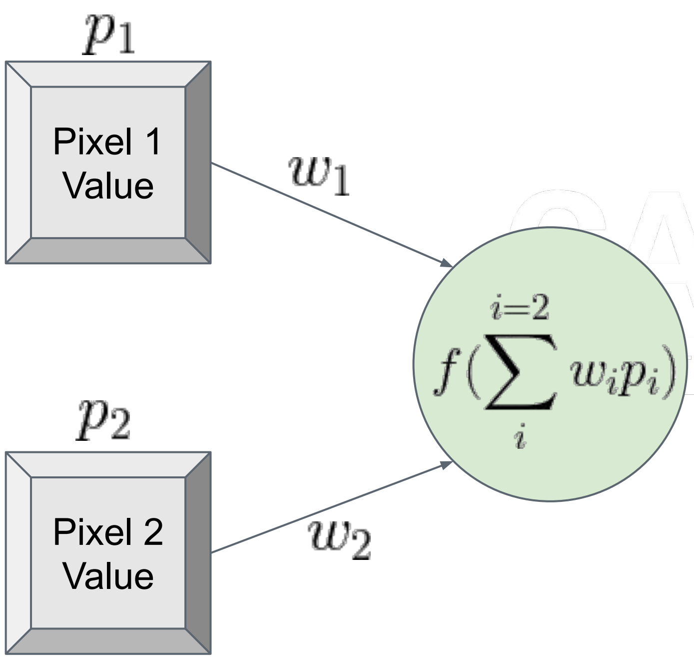
## Features

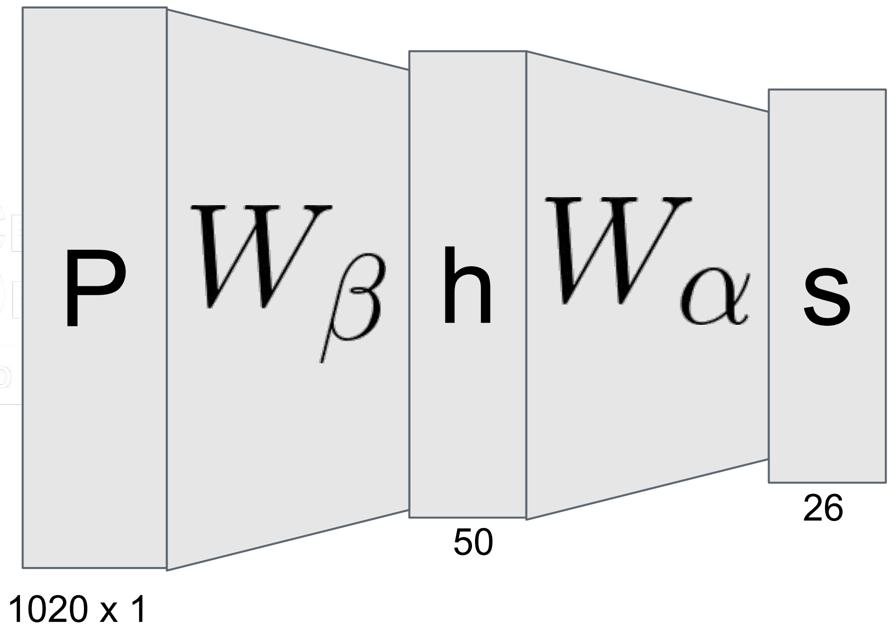
- Output of function falls between 0 and 1.
- Roughly approximates how a neuron works - 0 values until some threshold is reached, then 1.

## Challenges

- Gradient Decay & Saturation
- Not Zero-Centered & Unidirectional Gradient Solutions

# Network Optimization: Weight Initialization



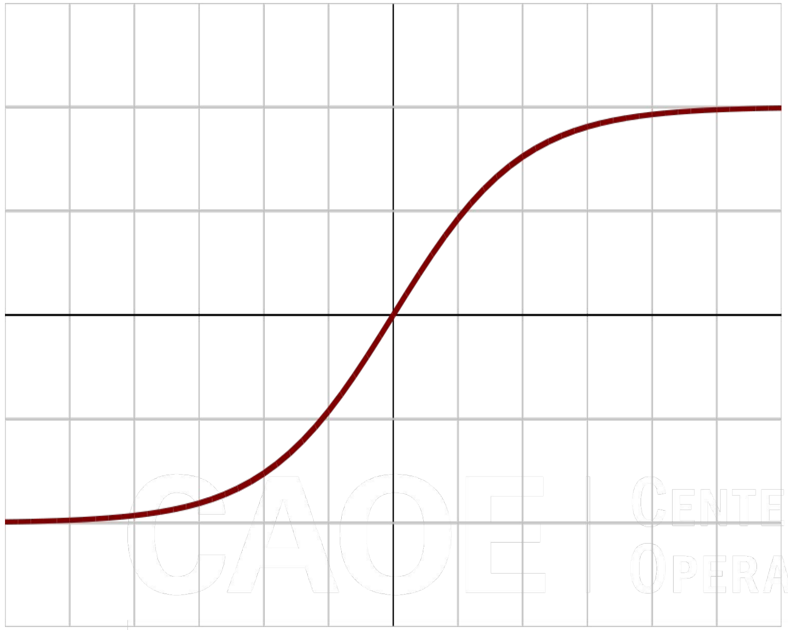
# Network Optimization: Weight Initialization

```
W = np.random.randn(3072, 10) * .0001
```

# Network Optimization: Weight Initialization

**Idea:** Big numbers!

W = np.random.randn(3072, 10) * 10





$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

# Network Optimization: Weight Initialization

**Idea:** ...medium numbers!

(Ok, wait a minute, this is harder than it seemed).

# Xavier Initialization

**Initial weights should be based on model complexity.**

Measurement of complexity: How many inputs and outputs your network has.

# Xavier Initialization

**Original**:

W = np.random.randn(3072, 10) * .0001



**Xavier**:

W = np.random.randn(3072, 10) / np.sqrt(3072)

# Xavier Initialization

**<u>Original:</u>**

**W = np.random.randn(3072, 10) * .0001**

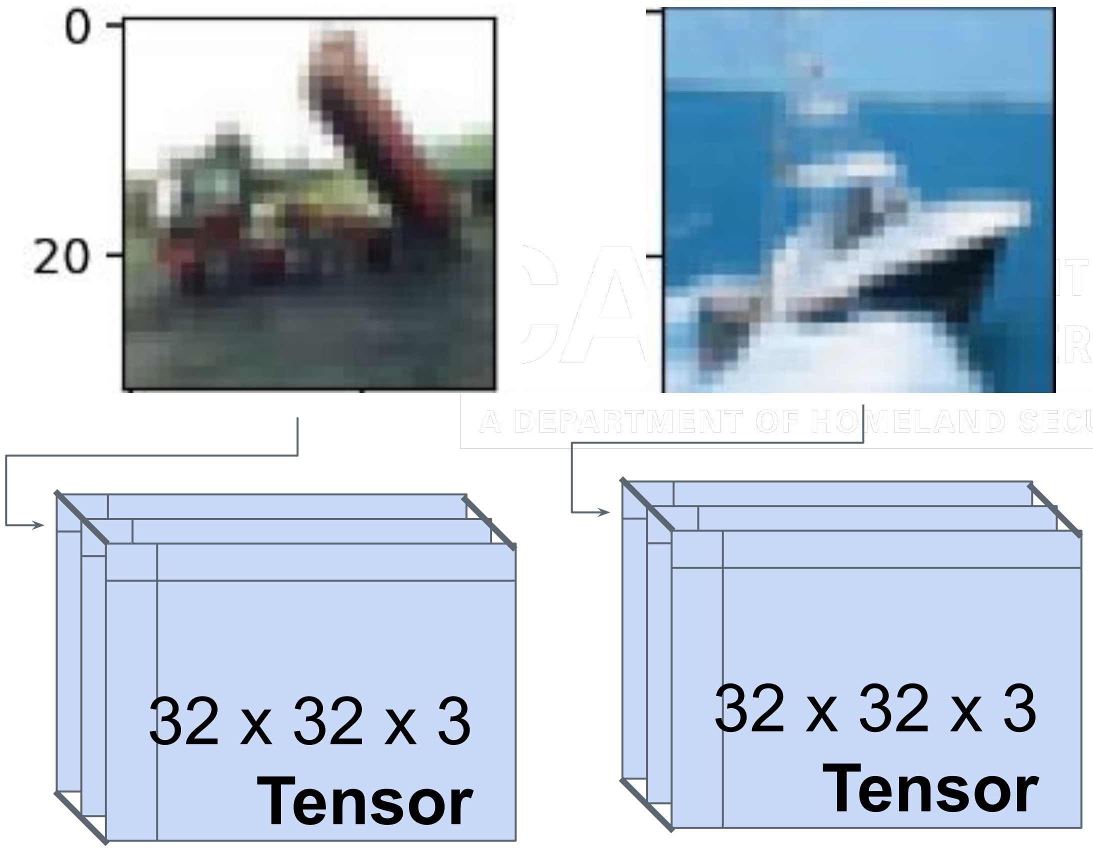
**<u>Xavier:</u>**

**W = np.random.randn(3072, 10) / np.sqrt(3072)**

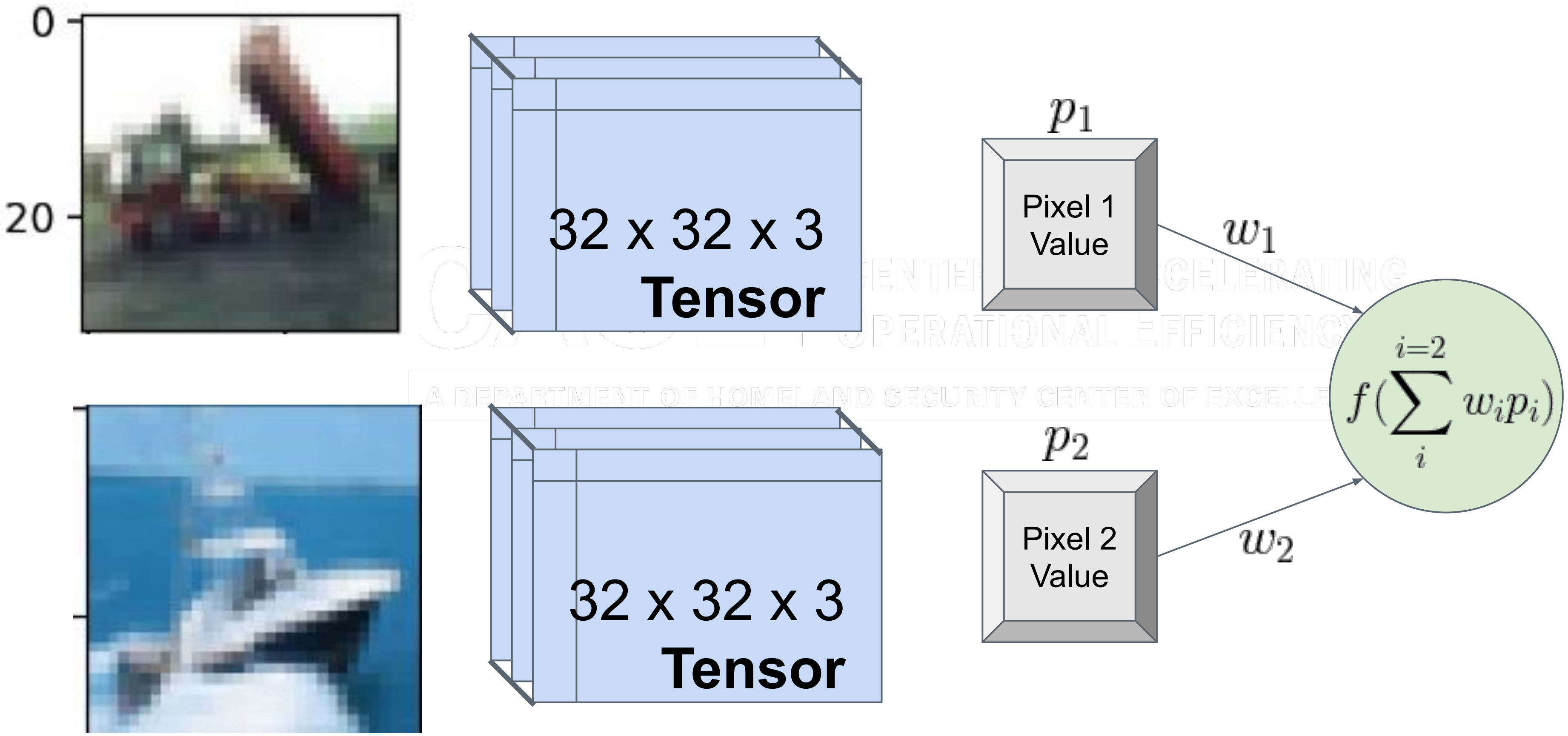
**<u>He:</u>**

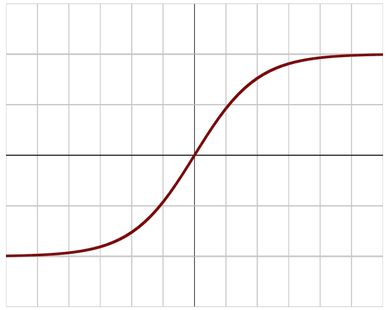
**W = np.random.randn(3072, 10) / np.sqrt(3072 / 2)**

# Another Strategy: Batch Normalization



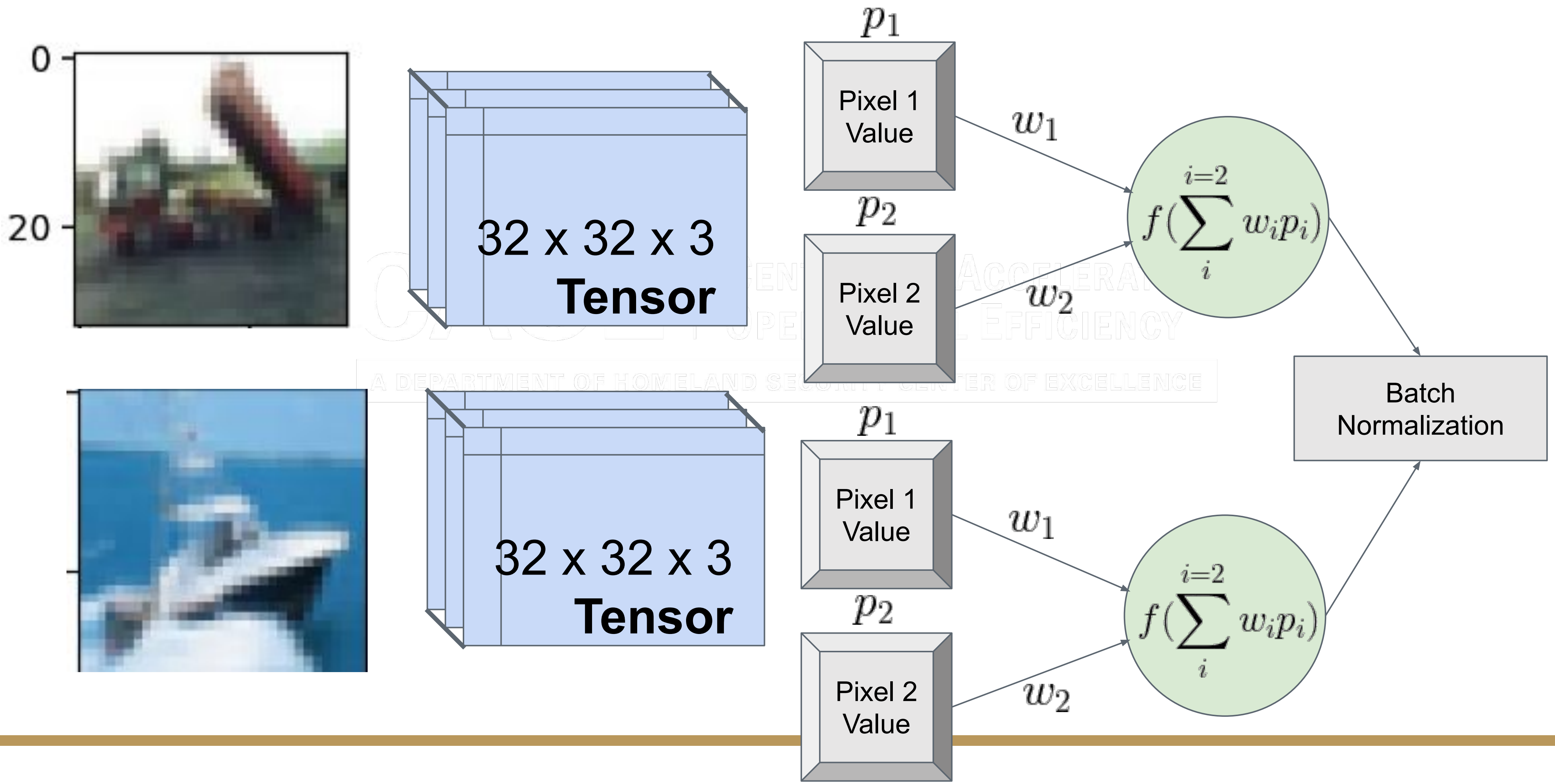
# Another Strategy: Batch Normalization



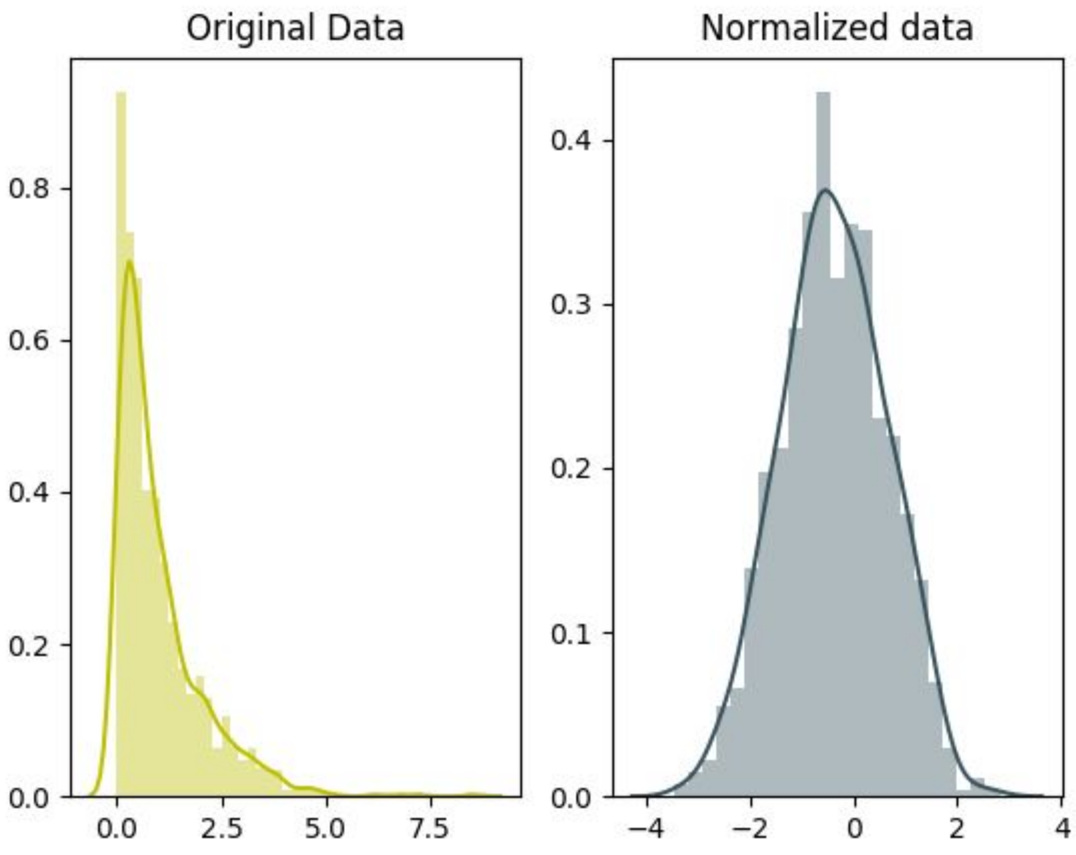



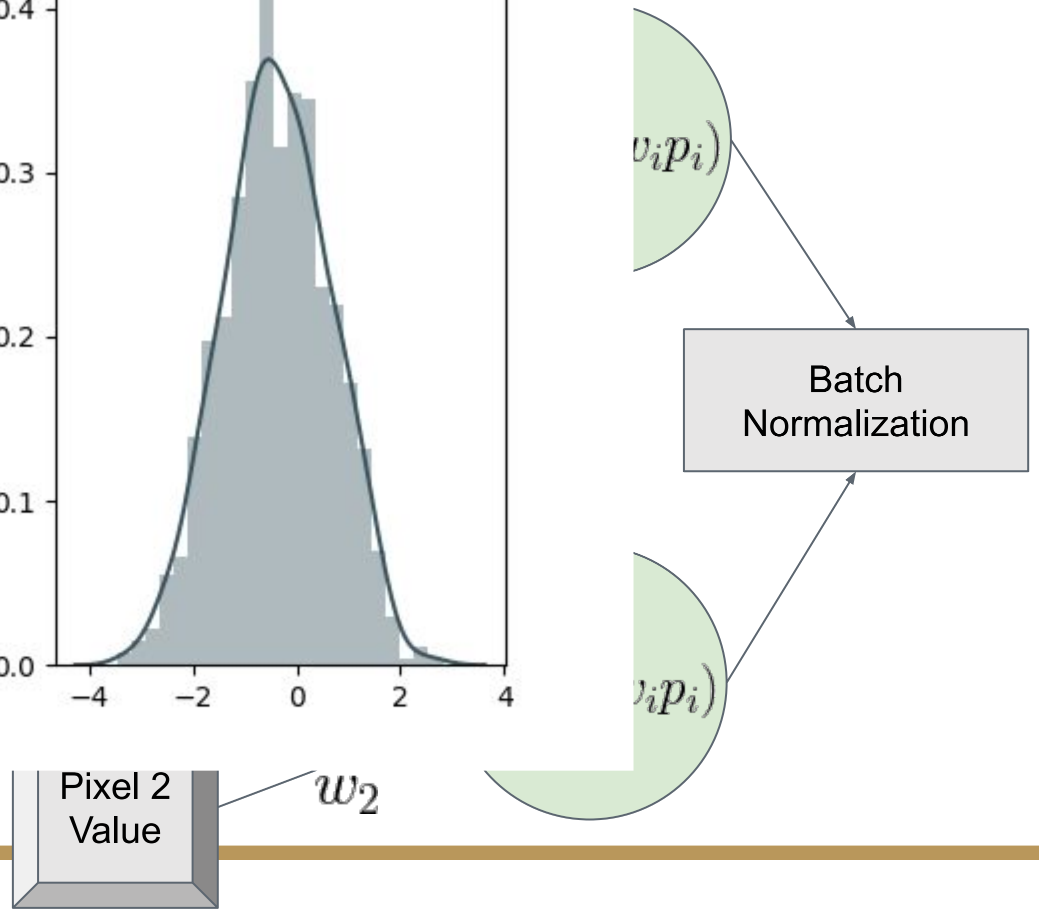
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

# Another Strategy: Batch Normalization

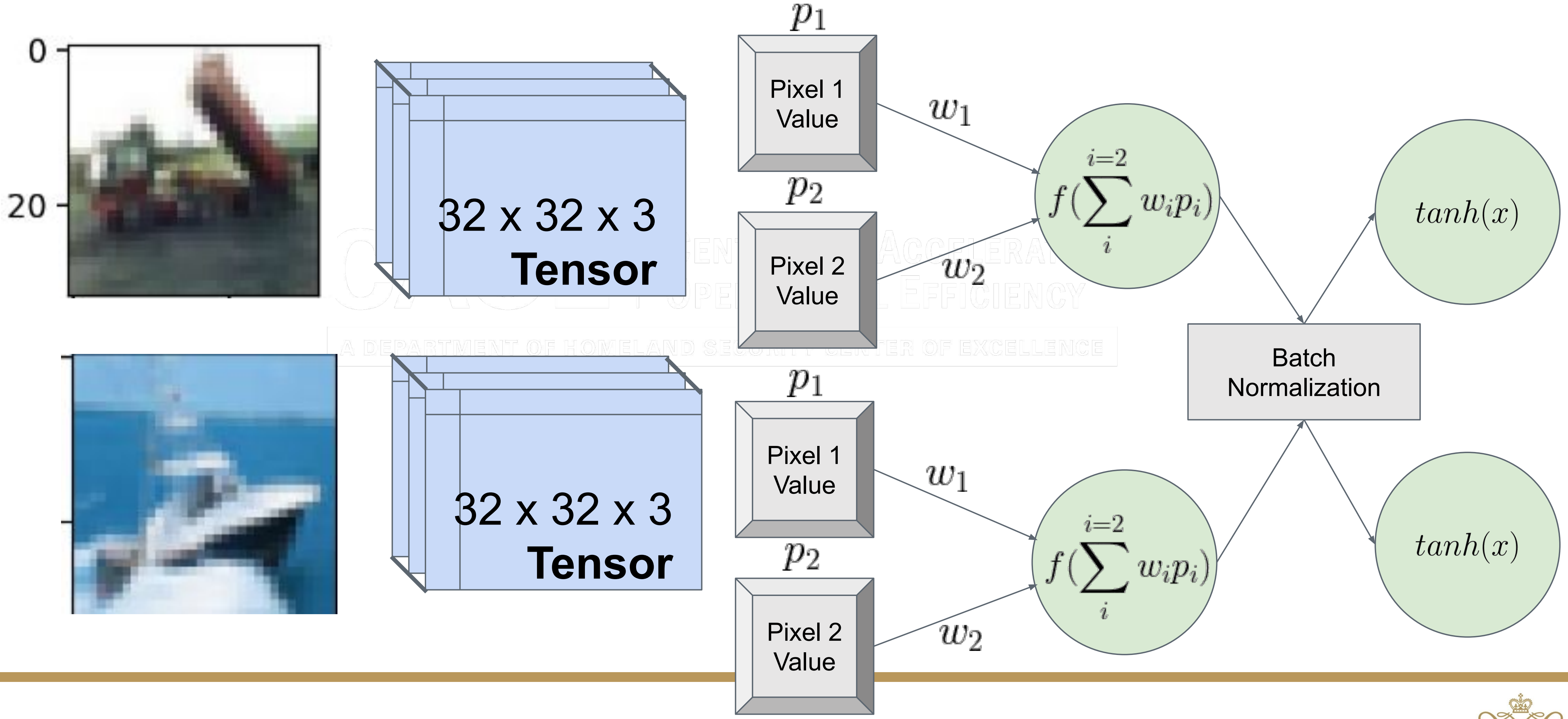


# Another Strategy: Batch Normalization



# Another Strategy: Batch Normalization



# Where we are now

1. Define network architecture (number of hidden layers, inputs, outputs, batch normalizations, activations, etc).
2. Define data preprocessing pipeline (zero-mean standardization).
3. Define weight initializations strategy.