CAP 4630 – Neural Networks

Instructor: Aakash Kumar

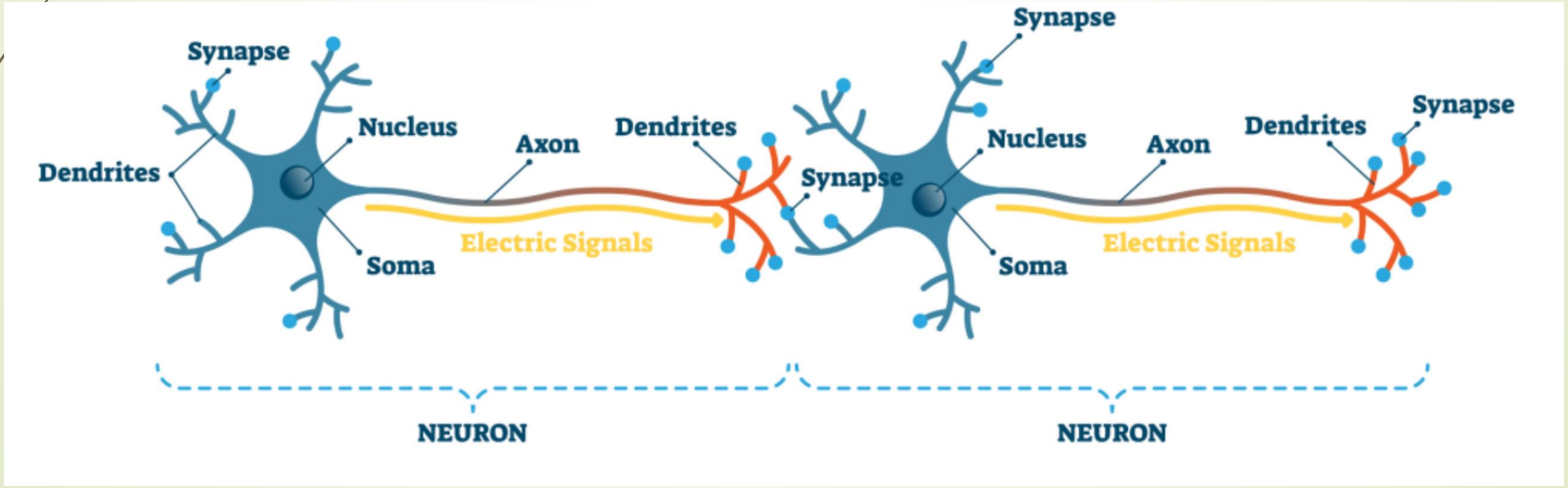
University of Central Florida

Brief History of Neural Networks

- The idea of building intelligent systems by drawing inspiration from the human brain has a longstanding history.
- Specifically, this concept aims to model and simulate neurons — the primary cells involved in animal nervous systems.
- The term "neuron" is often used for branding purposes, aligning with a narrative that emphasizes biological inspiration.
- This narrative has frequently been leveraged in company positioning to highlight innovative, "brain-like" approaches in AI.

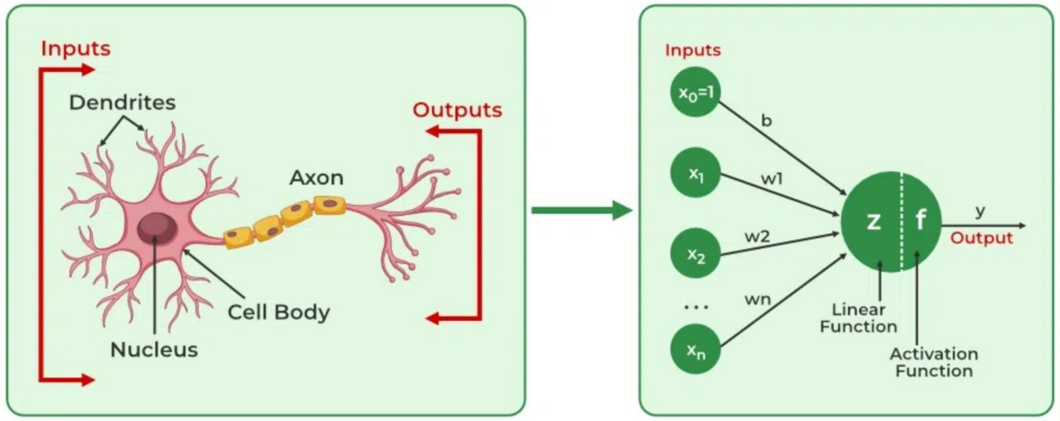
Biological Inspiration for Neural Networks

- Dendrites receive input signals.
- Axon transmits electric signals to other neurons.
- Synapses facilitate communication between neurons.



Compassion

- Dendrites receive input signals.
- The cell body processes the signals.
- The axon transmits electric impulses to other neurons.
- Activation: Neurons "fire" when the signal is strong enough to surpass a threshold.

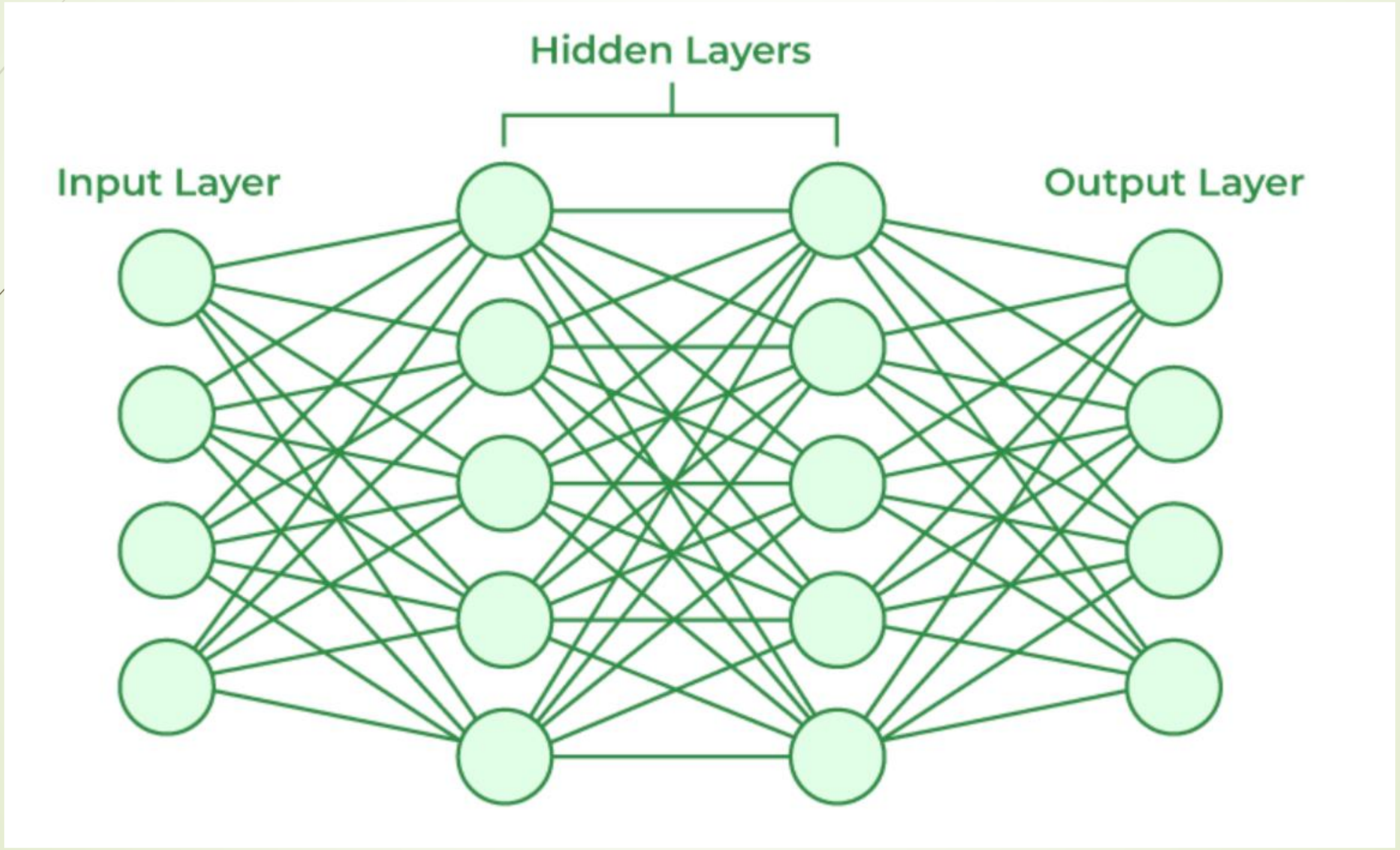


Biological neurons to Artificial neurons

- Inputs are combined with weights (w1, w2, etc.) and a bias term (b).
- A linear function computes the weighted sum.
- The activation function maps the input to an output, determining the "activation" level.

Ref: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/

Artificial Neural Network



Ref: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/

"Neural Networks" vs. Biological Neurons

- Limited Biological Inspiration: Despite the name, modern neural networks are not directly modeled after the human brain. They don't function like biological neurons, and neuroscientific feedback was not used in their development.
- Key Differences:
  - Matrix Multiplications: Artificial neural networks rely on matrix operations that require full connectivity, which is not how biological neural networks are structured.
  - Backpropagation: In artificial networks, backpropagation requires bidirectional connections to adjust weights. Biological neurons, however, only transmit signals in one direction.
  - Activation Representation: Artificial networks represent values as continuous activation strengths. In contrast, biological neurons operate using spikes, where the signal is either present or absent.

Neuromorphic or Biomimetic Computing

- Inspired by Biological Computation: Animal brains perform operations like convolutions (e.g., in the retina) that are replicated in some areas of artificial intelligence.
- Research Focus: Neuromorphic and biomimetic computing aim to create systems that more closely model the functioning of biological neurons, moving beyond the traditional neural network structure.
- Objectives:
  - Often focused on understanding the human brain and its mechanisms.
  - Performance optimization is typically a secondary goal, with the primary emphasis on biological accuracy.
  - Potential of Human-Equivalent Computation: There's an implicit belief that, if implemented effectively, neuromorphic systems could approach human-like cognitive capabilities.
  - Challenges: Despite advancements, the exact mechanisms and algorithms used by the human brain to perform complex tasks remain largely unknown.

Brief History (1940-1970)

- Early Development:
  - The Perceptron architecture was first introduced by McCullough and Pitts in 1943, later described in detail by Frank Rosenblatt.
  - It functioned as a linear classifier with a threshold output, accompanied by a unique learning algorithm.
- Rise of Hype:
  - The Perceptron sparked significant excitement and optimism, with researchers and funders hopeful about its potential.
- The Book "Perceptrons" (1969):
  - In 1969, Marvin Minsky and Seymour Papert published Perceptrons, highlighting limitations of the linear perceptron, especially its inability to learn non-linear functions.
  - This critique led to a sharp decline in funding and interest, stalling research in neural networks for nearly a decade.

Brief history (1980s)

- Introduction of Hidden Layers:
  - Neural networks gained renewed popularity in the mid-1980s, particularly after the influential work of Rumelhart, Hinton, and Williams (1986).
- Key Innovations:
  - Backpropagation: Introduced the concept of training by backpropagating errors, enabling more effective learning.
  - Non-linear Activation Functions: Placed non-linear functions between layers, allowing neural networks to represent complex, non-linear relationships and arbitrary functions.
- New Hype Cycle:
  - This period marked a resurgence in interest and optimism about neural networks.
- The Connectionist Debate:
  - Emerged as a central topic in AI and psychology, with connectionist models (neural networks) competing against symbolic and computational models, sparking debates about the nature of intelligence and cognitive modeling.

Brief History (1990s–2000s)

- Slow Progress and Setbacks:
  - The era saw limited advancement in neural networks, partly due to misinterpretations of a theorem suggesting that a network with a single hidden layer could represent any function, discouraging exploration of deeper networks.
  - The popularity of the sigmoid activation function added challenges, as it led to slow convergence during training.
- Notable Innovations Amidst Challenges:
  - Convolutional Neural Networks (CNNs): Developed by Yann LeCun and Leon Bottou with the LeNet architecture, CNNs were designed for image processing but remained overshadowed by other computer vision techniques of the time.
  - Max Pooling: Introduced to enhance CNNs by downsampling features, reducing computation, and improving generalization.
  - Long Short-Term Memory (LSTM): Invented by Sepp Hochreiter and Jürgen Schmidhuber, LSTMs addressed issues with learning long-term dependencies in sequential data but lacked immediate practical applications.
- Explorations in Training Multi-Layer Networks:
  - Researchers experimented with various methods to improve training for deeper networks, laying groundwork for future advancements.

Brief History (2010s–2020s)

- The 2010s saw a surge in breakthroughs, as deep learning techniques began to surpass traditional methods across multiple domains.
- Key Milestones:
  - 2012: AlexNet — Introduced by Krizhevsky, Sutskever, and Hinton, this CNN-based model revolutionized computer vision, making prior techniques obsolete.
  - 2014: Seq2Seq with LSTM — Enabled effective neural machine translation, surpassing traditional methods in language translation.
  - 2016: AlphaGo — Developed by DeepMind, this system, combining CNNs and deep reinforcement learning (RL), defeated a world champion in the complex game of Go, marking a milestone in AI.
  - 2017: Transformers — A new architecture that shifted the paradigm in natural language processing (NLP), gradually replacing LSTMs for sequence tasks.
  - 2018: BERT — Google's Bidirectional Encoder Representations from Transformers, the first large language model (LLM) based on transformers, achieved remarkable progress in understanding human text.
  - 2022: GPT-3.5 — OpenAI's LLM, GPT-3.5, became the first broadly available model, bringing advanced language generation capabilities to the general public.

Problem Setup

- Supervised Learning: We approach this problem in the context of supervised learning, where the goal is to learn a mapping from input features to a target output.
- Input Data:
  - Each training example consists of a set of features $x^{(i)}$, where i denotes the i-th training example.
  - The feature vector x contains n features that describe various attributes of the data.

$$x^{(i)} = [x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}]$$

- Output Data:
  - For each training example, we have an output label $y^{(i)}$,which is a scalar (a single value) representing the target or ground truth associated with $x^{(i)}$.

Hypothesis Function

- Formulation:
  - To simplify notation, we define our hypothesis function as a weighted sum of the input features:

$$h(x) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

  - But Wait... Isn't This Linear Regression?
    - Yes! This setup is identical to the formulation of linear regression.
    - In linear regression, we aim to find the weights w that minimize the error between the predicted output h(x) and the actual output y.

Multiple Outputs

- Expanding the Hypothesis Function:
  - Now, let's consider a scenario where we have a vector of outputs

$$\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m].$$

  - To achieve this, we introduce a **weight matrix** W of size m×n, where each row corresponds to a different output.
- Mathematical Representation:
  - For each output y^j, we have:

$$\hat{y}_j = \sum_i w_{ij} x_i$$

  - This equation represents a linear combination of inputs for each output, using the corresponding weights from the matrix W.
- But Wait... Isn't This Still Linear Regression?
  - Yes! This setup can be seen as m independent linear regressions packaged together.
  - Each output y_j is computed as a linear regression on the input features, making this setup essentially a multi-output linear regression model.

Multiple Layers

- Introduction to Multiple Layers:
  - Neural networks can consist of multiple layers, each with its own weight matrix $W^{(1)}, W^{(2)}, \dots, W^{(k)}$ allowing for complex representations.
- Hidden Layers:
  - We introduce hidden layers $z^{(c)}$, where c=0,1,…, Each layer $z^{(c)}$ has a size n^c (number of neurons in that layer).
  - $z^{(0)}$ represents the input x.
  - $z^{(k)}$ represents the output y.
- Layer-by-Layer Transformation:
  - Each layer is computed as:

$$z^{(c+1)} = W^{(c)} z^{(c)}$$

- Output Prediction:

$$\hat{y} = W^{(k)} W^{(k-1)} \dots W^{(0)} x$$

But, Wait...

- Can’t We Just Multiply the Matrices W?
- By multiplying all the weight matrices, we can reduce the network to a single linear transformation:

$$W = W^{(k)} W^{(k-1)} \dots W^{(0)}$$

- This simplifies the hypothesis function to:

$$\hat{y} = f(x, \theta) = Wx$$

Limitations of a Purely Linear Model:

- Still Linear Regression: Despite using multiple layers, the output remains a linear combination of the input.
- No Gain in Expressivity: We don't achieve any new capability to represent complex, non-linear relationships.
- Inability to Solve Non-Linear Problems: Without non-linear transformations, the model can't capture non-linear patterns in the data.
- Excessive Parameters: We end up with a large number of parameters W that contribute nothing beyond a single-layer linear regression.
- Historical Context:
  - This limitation formed the basis of Minsky's critique of perceptrons, highlighting that purely linear architectures cannot solve complex, non-linear tasks.

Nonlinearity

- To overcome the limitations of purely linear transformations, we introduce a nonlinear activation function g(·), and define each layer transformation as:

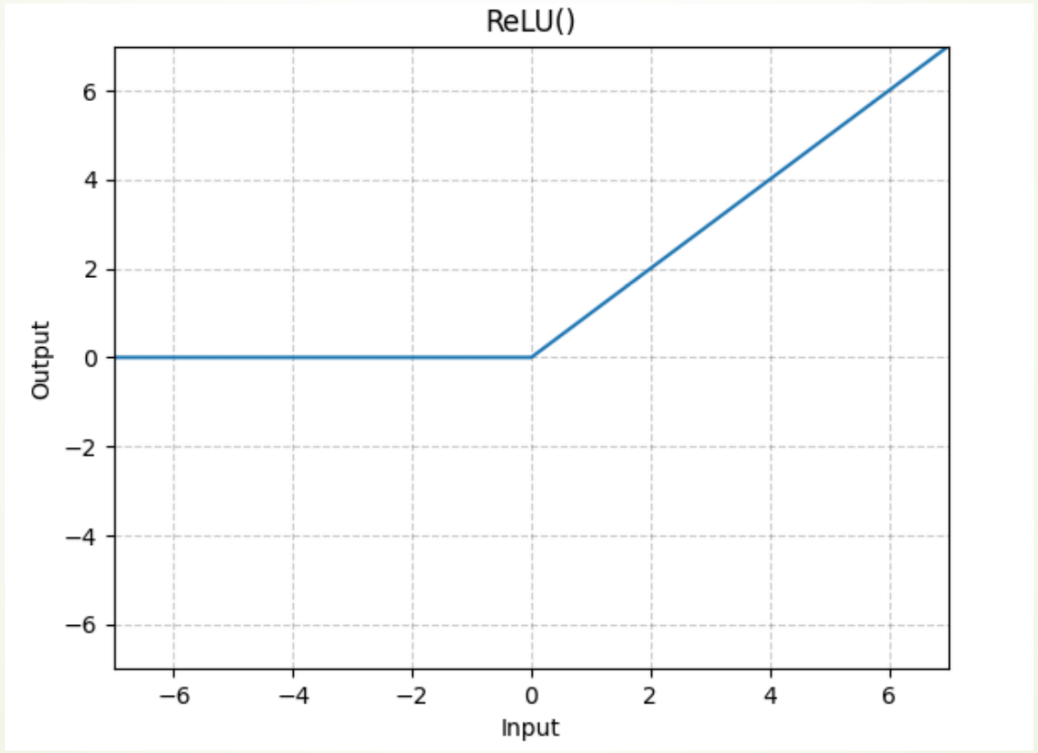
$$z^{(c+1)} = g\left(W^{(c)} z^{(c)}\right)$$

- Applying Nonlinearity:
  - The function g is applied element-wise to each component of the vector introducing nonlinearity into the network
- Impact on the Network:
  - With nonlinearity, we can no longer combine the weight matrices into a single product.
  - The full network output now involves applying g at each layer:

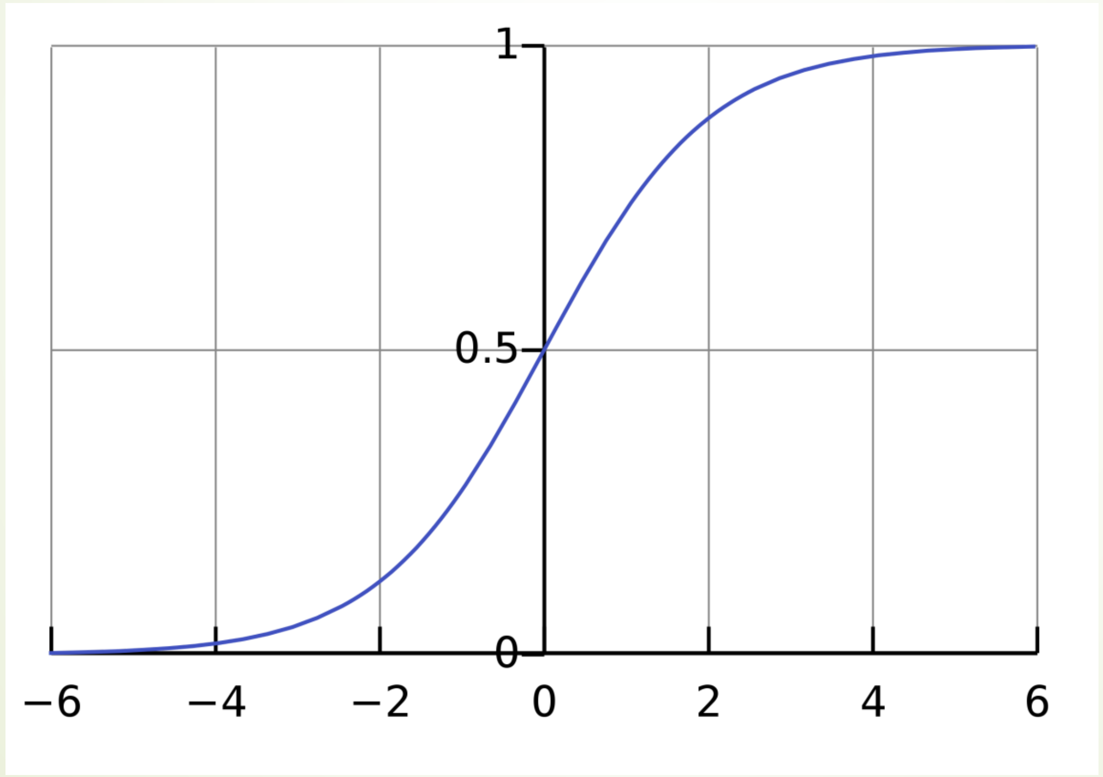
$$\hat{y} = g^{(k)}\left(W^{(k)} \cdot g^{(k-1)}\left(W^{(k-1)} \dots g^{(0)}\left(W^{(0)} x\right) \dots\right)\right)$$

- Why Nonlinearity?
  - Nonlinear activation functions allow the network to learn complex patterns and solve non-linear problems, vastly increasing its expressive power.

Nonlinearity: ReLU



Nonlinearity: sigmoid



Fully Connected Network (Multi-Layer Perceptron)

- Definition:
  - A fully connected neural network is one where every node in a layer is connected to every node in the preceding and succeeding layers (if they exist).
  - This means each node in a layer receives inputs from all nodes in the previous layer and sends outputs to all nodes in the next layer.
- Terminology:
  - Also known as a multi-layer perceptron (MLP).
  - MLPs are foundational architectures in neural networks and are often used as a baseline model for classification and regression tasks.
- Characteristics:
  - Fully connected layers allow for complex mappings between inputs and outputs but can lead to high computational costs and many parameters.
  - Suitable for general-purpose learning but may struggle with high-dimensional data like images, where specialized architectures (e.g., CNNs) are preferred.

Did We Gain Anything in Expressivity?

- Yes!!!
  - A series of theorems known as the Universal Approximation Theorems demonstrate that neural networks with a single hidden layer can approximate arbitrary functions to an arbitrary degree of precision.
- Requirements:
  - This approximation is possible with only mild conditions on the nonlinear activation function g.
  - However, achieving this approximation may require an infinitely large hidden layer, which is impractical for real-world applications.
- Practical Nonlinearities:
  - Commonly used nonlinear activation functions (like ReLU, sigmoid, tanh) meet the requirements specified in these theorems, enabling neural networks to model complex, non-linear relationships in practice.

Understanding a Single Neuron in a Neural Network

- Understanding a Single Neuron in a Neural Network (Without Bias)
- Concept:
  - A neuron (or perceptron) can be thought of as a basic unit in a neural network that combines inputs with weights and applies an activation function.
- Inputs and Weights:
  - Suppose we have two input features, x1 and x2.
  - Each input has an associated weight: w_1 for x_1 and w_2 for x_2.
  - These weights control how strongly each input contributes to the neuron's output.
- Weighted Sum (Linear Combination):
- The neuron computes a weighted sum of the inputs:

$$z = x_1 \cdot w_1 + x_2 \cdot w_2$$

- This linear combination z is the input to the activation function.

Understanding a Single Neuron in a Neural Network Conti…

- Activation Function:
  - To introduce non-linearity, we apply an activation function σ(z). Common choices include:
    - Sigmoid: Compresses the output to a range between 0 and 1, suitable for probabilities.

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

    - ReLU (Rectified Linear Unit): Outputs z if z>; otherwise, 0,

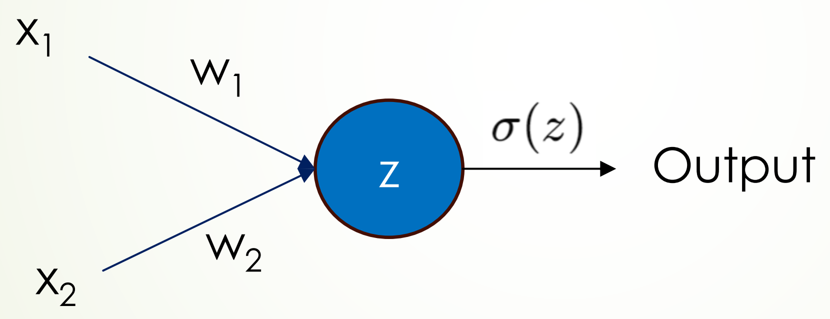
$$\sigma(z) = \max(0, z)$$

- Output:
  - The final output of the neuron is:

$$\text{output} = \sigma(z) = \sigma(x_1 \cdot w_1 + x_2 \cdot w_2)$$

  - This output might represent a prediction (e.g., a probability if using sigmoid) or be passed to another layer.

Single Neuron



$$\text{output} = \sigma(z) = \sigma(x_1 \cdot w_1 + x_2 \cdot w_2)$$

Forward Propagation in a Neural Network

- Network Configuration
  - Input Layer: 2 neurons (features x_1 and x_2)
  - Hidden Layer: 4 neurons
  - Output Layer: 2 neurons (outputs y_1 and y_2)

Initialize Inputs and Weights

- Inputs: x_1 and x_2 (from the dataset)
- Weights:
  - Input to Hidden Layer

    Weights $w_{11}, w_{12}, w_{13}, w_{14}$ for x_1 to each hidden neuron

    Weights $w_{21}, w_{22}, w_{23}, w_{24}$ for x_2 to each hidden neuron
  - Hidden to Output Layer:

    Weights $w'_{11}, w'_{12}, w'_{13}, w'_{14}$ for connections from hidden to output neurons

Calculate Hidden Layer Activations

- For each hidden neuron

$$h_j = \sigma(x_1 \cdot w_{1j} + x_2 \cdot w_{2j})$$

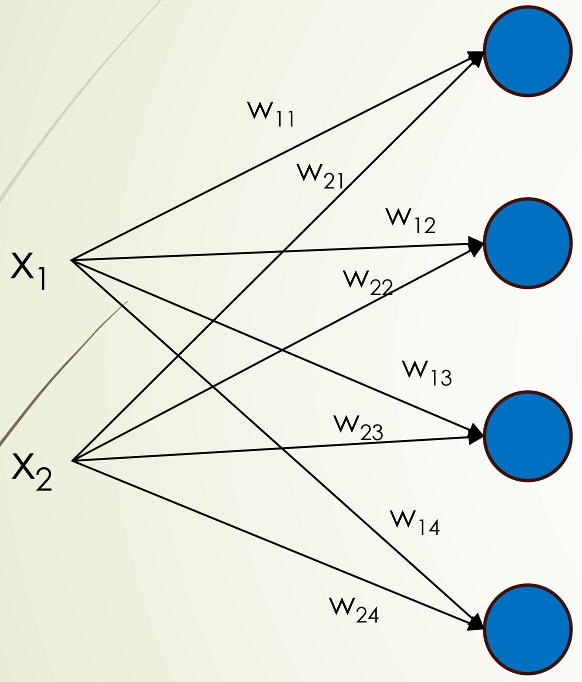
- Example Calculations:

Hidden Neuron 1: $h_1 = \sigma(x_1 \cdot w_{11} + x_2 \cdot w_{21})$

Hidden Neuron 2: $h_2 = \sigma(x_1 \cdot w_{12} + x_2 \cdot w_{22})$

Hidden Neuron 3: $h_3 = \sigma(x_1 \cdot w_{13} + x_2 \cdot w_{23})$

Hidden Neuron 4: $h_4 = \sigma(x_1 \cdot w_{14} + x_2 \cdot w_{24})$



$$h_1 = \sigma(x_1 \cdot w_{11} + x_2 \cdot w_{21})$$

$$h_2 = \sigma(x_1 \cdot w_{12} + x_2 \cdot w_{22})$$

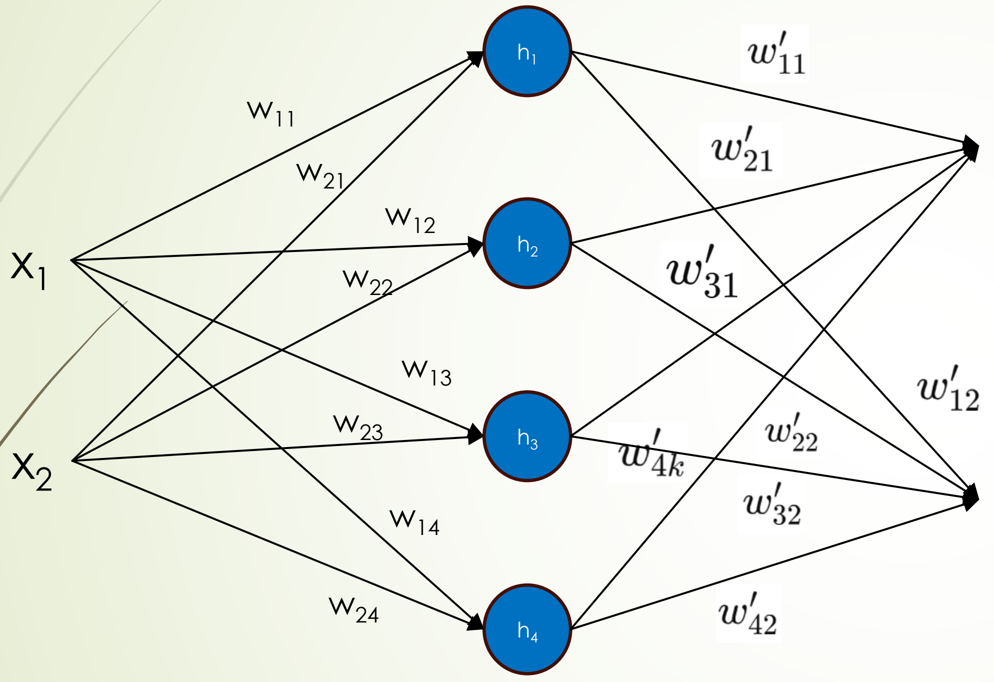
$$h_3 = \sigma(x_1 \cdot w_{13} + x_2 \cdot w_{23})$$

$$h_4 = \sigma(x_1 \cdot w_{14} + x_2 \cdot w_{24})$$

Calculate Output Layer Activations

Output Neuron 1: $y_1 = \sigma(h_1 \cdot w'_{11} + h_2 \cdot w'_{21} + h_3 \cdot w'_{31} + h_4 \cdot w'_{41})$

Output Neuron 2: $y_2 = \sigma(h_1 \cdot w'_{12} + h_2 \cdot w'_{22} + h_3 \cdot w'_{32} + h_4 \cdot w'_{42})$



$$y_1 = \sigma(h_1 \cdot w'_{11} + h_2 \cdot w'_{21} + h_3 \cdot w'_{31} + h_4 \cdot w'_{41})$$

$$y_2 = \sigma(h_1 \cdot w'_{12} + h_2 \cdot w'_{22} + h_3 \cdot w'_{32} + h_4 \cdot w'_{42})$$

The Role of Bias in Neural Networks

- Purpose: The bias term shifts the activation function, adding flexibility in learning patterns, like an intercept in linear regression.
- Function: Each neuron has its own bias term, adjusted during training to fit the data better.
- Why It Matters: Biases help improve understanding of network flexibility and the learning process.

Calculating with Bias

- Hidden Layer Activations:

Hidden Neuron 1: $h_1 = \sigma(x_1 \cdot w_{11} + x_2 \cdot w_{21} + b_1)$

Hidden Neuron 2: $h_2 = \sigma(x_1 \cdot w_{12} + x_2 \cdot w_{22} + b_2)$

Hidden Neuron 3: $h_3 = \sigma(x_1 \cdot w_{13} + x_2 \cdot w_{23} + b_3)$

Hidden Neuron 4: $h_4 = \sigma(x_1 \cdot w_{14} + x_2 \cdot w_{24} + b_4)$

- Output Layers

Output Neuron 1: $y_1 = \sigma(h_1 \cdot w'_{11} + h_2 \cdot w'_{21} + h_3 \cdot w'_{31} + h_4 \cdot w'_{41} + b'_1)$

Output Neuron 2: $y_2 = \sigma(h_1 \cdot w'_{12} + h_2 \cdot w'_{22} + h_3 \cdot w'_{32} + h_4 \cdot w'_{42} + b'_2)$

The Softmax Function

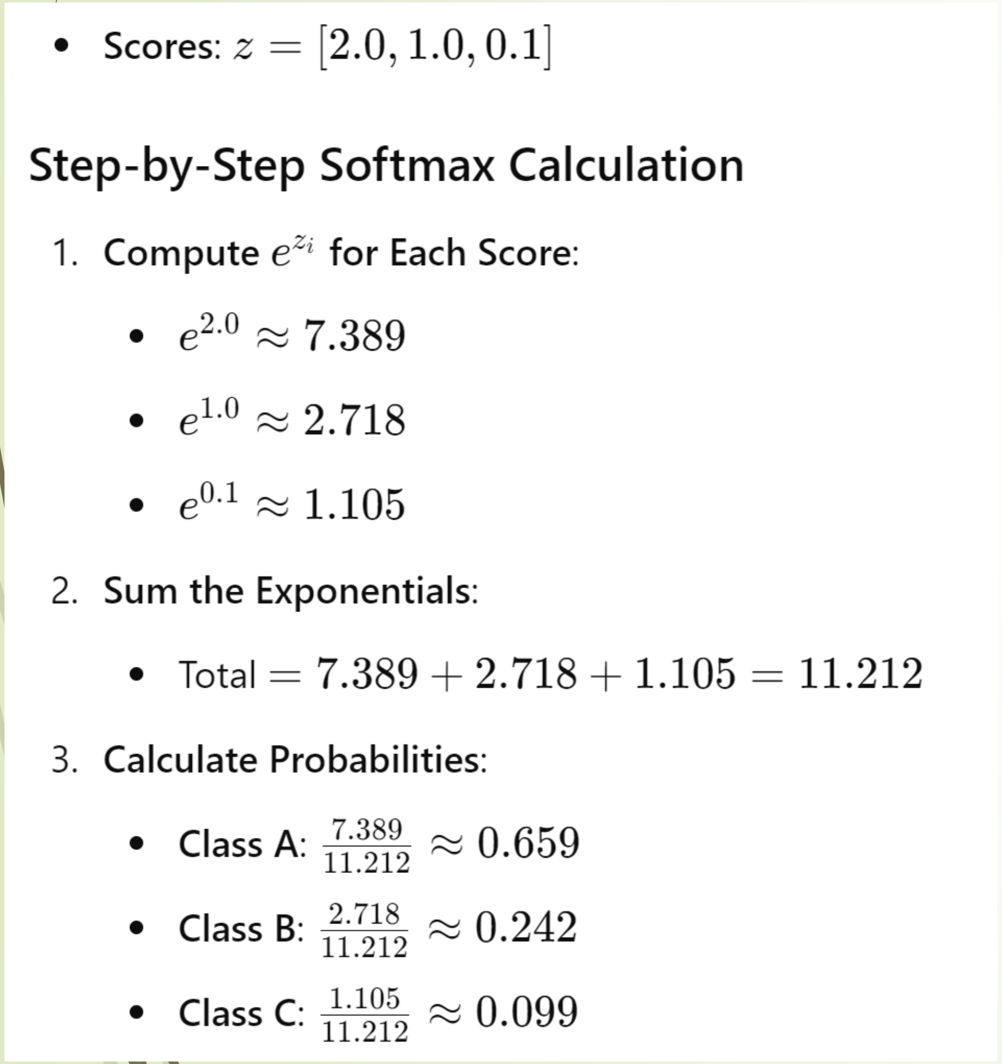
- Purpose:
  - The softmax function is commonly used in the output layer of a neural network for multi-class classification tasks.
  - It transforms raw output scores into probabilities, assigning each class a probability between 0 and 1.
- Definition:
  - For a set of output scores corresponding z to n classes, the softmax function calculates the probability for each class i as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

The Softmax Function Continue..

- Properties:
  - Probabilities Sum to 1: The outputs of the softmax function for all classes sum to 1, making it suitable for probabilistic interpretation.
  - Amplifies Differences: Larger scores become more dominant in the output, highlighting the most likely class.

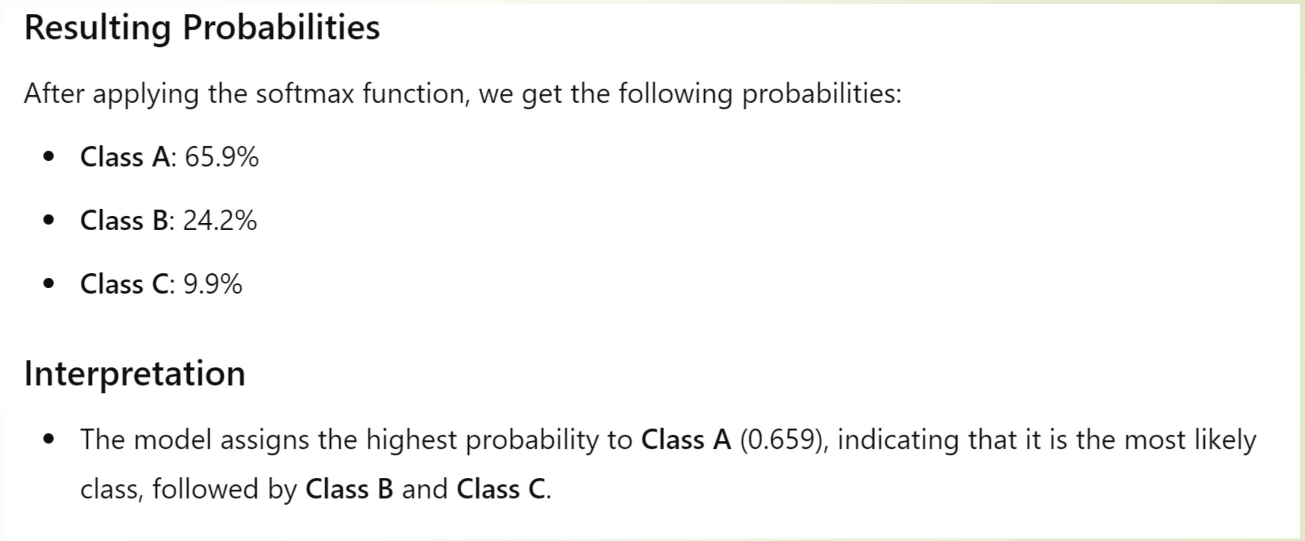
Softmax Computation



- Scores: $z = [2.0, 1.0, 0.1]$

Step-by-Step Softmax Calculation

1. **Compute e^{z_i} for Each Score:**
   - $e^{2.0} \approx 7.389$
   - $e^{1.0} \approx 2.718$
   - $e^{0.1} \approx 1.105$
2. **Sum the Exponentials:**
   - Total $= 7.389 + 2.718 + 1.105 = 11.212$
3. **Calculate Probabilities:**
   - Class A: $\frac{7.389}{11.212} \approx 0.659$
   - Class B: $\frac{2.718}{11.212} \approx 0.242$
   - Class C: $\frac{1.105}{11.212} \approx 0.099$



Resulting Probabilities

After applying the softmax function, we get the following probabilities:

- **Class A**: 65.9%
- **Class B**: 24.2%
- **Class C**: 9.9%

Interpretation

- The model assigns the highest probability to **Class A** (0.659), indicating that it is the most likely class, followed by **Class B** and **Class C**.

How Do We Train a Neural Network?

- Architecture:
  - Defined by factors like the number of layers, size of each layer, and choice of non-linear activation functions.
  - Engineered, Not Trained: Architecture choices are based on experience and intuition rather than training.
  - These are hyperparameters that we set before training begins.
- Parameters:
  - Represented by the weights W across all layers.
  - These are learned during training, adjusting to minimize the network's error on the data.
  - Training Method: Typically optimized using Stochastic Gradient Descent (SGD) or other variants, similar to linear regression but more complex.

Number of Parameters in a Neural Network (Including Bias)

- Weights: The product of the number of neurons between consecutive layers.
- Biases: Each neuron (except in the input layer) has an individual bias term.
- Input to First Hidden Layer:
  - Weights: input size × first hidden layer size
  - Biases: first hidden layer size
- Between Hidden Layers:
  - Weights: size of previous hidden layer × size of next hidden layer
  - Biases: size of next hidden layer
- Last Hidden Layer to Output Layer:
  - Weights: size of last hidden layer × output size
  - Biases: output size
- Total Parameters:
  - Sum up all weight matrices and biases for each layer.
- Scaling with Complexity:
  - As input dimensions, hidden layer sizes, and number of layers increase, the number of parameters w grows rapidly.

Taking the Gradient

- Goal: Compute the gradient for each parameter to minimize the loss.\

$$\nabla\mathcal{L} = \left[\frac{\partial\mathcal{L}}{\partial\theta_i}\right]$$

- Challenges:
  - Non-Differentiable Points:
    - Some activation functions, like ReLU, are not differentiable at certain points
    - Solution: Approximate by setting the derivative at non-differentiable points
  - Complexity with Millions of Parameters:
    - Calculating gradients manually for a large network is impractical.
    - Solution: Use automatic differentiation tools provided by libraries like PyTorch and TensorFlow.
- Efficient Computation:
  - Solution: Use backpropagation, which calculates gradients efficiently by moving backward through the network using the chain rule.

Global and Unique Solution

- Linear Regression:
  - For linear regression with least squares, the loss surface is convex.
  - This means there is a unique global minimum, and gradient descent will reliably find it.
- Neural Networks:
  - In fully connected neural networks, the loss function is non-convex with many local minima.
- Multiple Optimal Solutions:
  - This also introduces numerous local minima and saddle points.
- Challenges with Gradient Descent:
  - Despite the complex landscape, gradient descent works surprisingly well in practice.
  - However, there are no theoretical guarantees that it will find the global optimum.

Introduction to Overfitting

- Definition:
  - Overfitting occurs when a neural network learns the noise and specifics of the training data rather than the general patterns, leading to poor generalization on new data.
- Symptoms:
  - High accuracy on the training set but low accuracy on the validation/test set.
- Goal:
  - We want the model to perform well on new, unseen data, not just on the training data.

Regularization Techniques – L2 and L1 Regularization

- L2 Regularization (also called Weight Decay):
  - Adds a penalty term to the loss function based on the sum of the squares of the weights.
  - Effect: Reduces the magnitude of weights, encouraging simpler models that generalize better.

$$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda \sum_i W_i^2$$

- L1 Regularization:
  - Adds a penalty term based on the sum of the absolute values of the weights.
  - Effect: Encourages sparsity, potentially setting some weights to zero.

$$\mathcal{L}_{\text{total}} = \mathcal{L} + \lambda \sum_i |W_i|$$

Dropout Regularization

- Definition:
  - Dropout is a technique where, during training, randomly selected neurons are "dropped out" (set to zero) in each forward pass.
- Purpose:
  - Dropout helps address co-adaptation—a phenomenon where neurons rely on each other's output values rather than learning unique patterns from the data.
  - By regularly changing which neurons are active, dropout pushes each neuron to learn features that are independently valuable, fostering more diverse and distributed feature representations.
- How it Works:
  - During each forward pass, each neuron has a probability p of being kept (often p=0.5 for hidden layers).

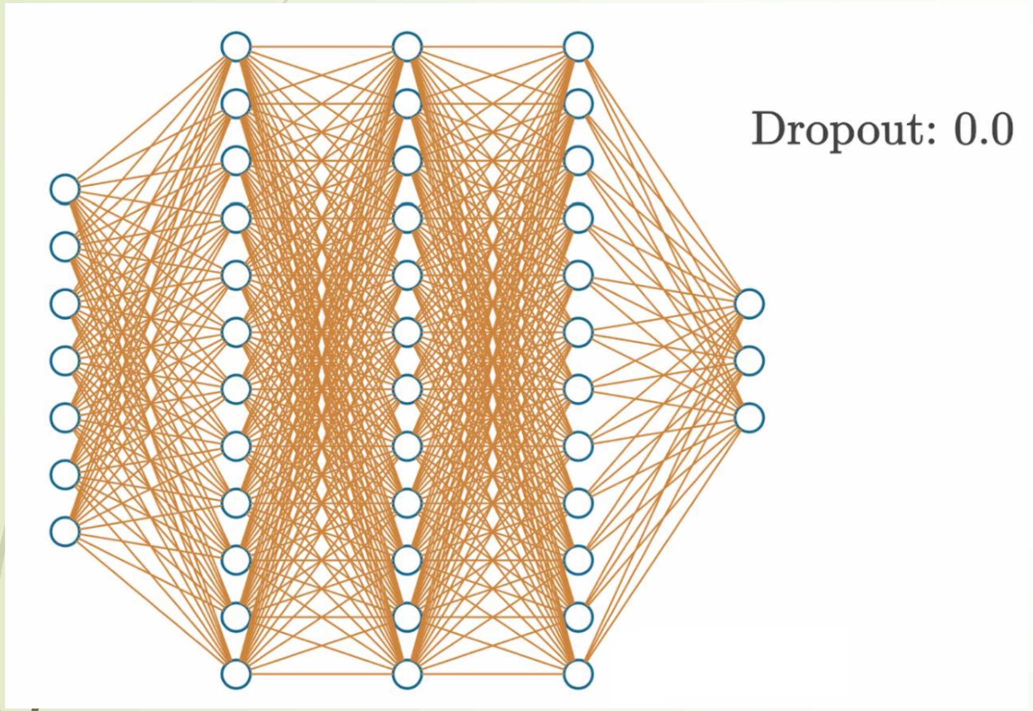
Dropout Regularization Conti...

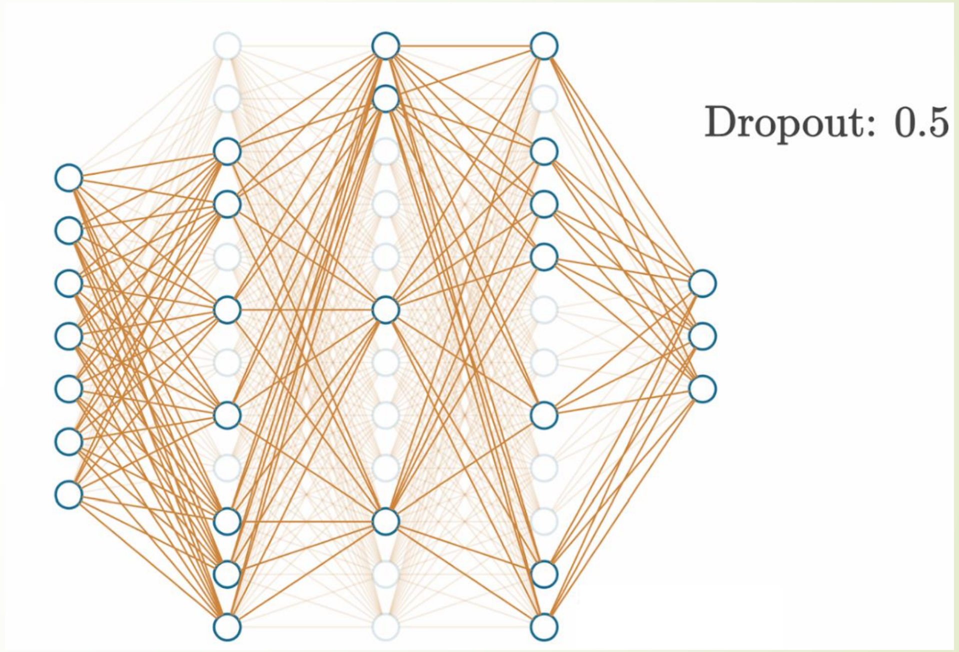
- Preventing Overfitting:
  - Dropout mitigates overfitting by preventing the network from memorizing specific samples or relying too heavily on certain neurons for particular data points.
  - As neurons are disabled, the model is forced to rely on the collective behavior of multiple neurons, which leads to better generalization on new data.

Dropout During Training

- Training Phase
  - During training, dropout randomly disables a certain percentage of neurons in each forward pass (the exact percentage is determined by the dropout rate, such as 0.5 or 50%).
  - This dropout process makes the network more robust by forcing it to learn redundantly across different neurons. Each neuron needs to be more flexible and learn features on its own, rather than relying on the presence of specific other neurons.
  - As a result, during training, the network is effectively operating as if it were an “ensemble” of different subnetworks, each using a subset of neurons to make predictions.

Dropout During Training





Ref: https://wikidocs.net/182302

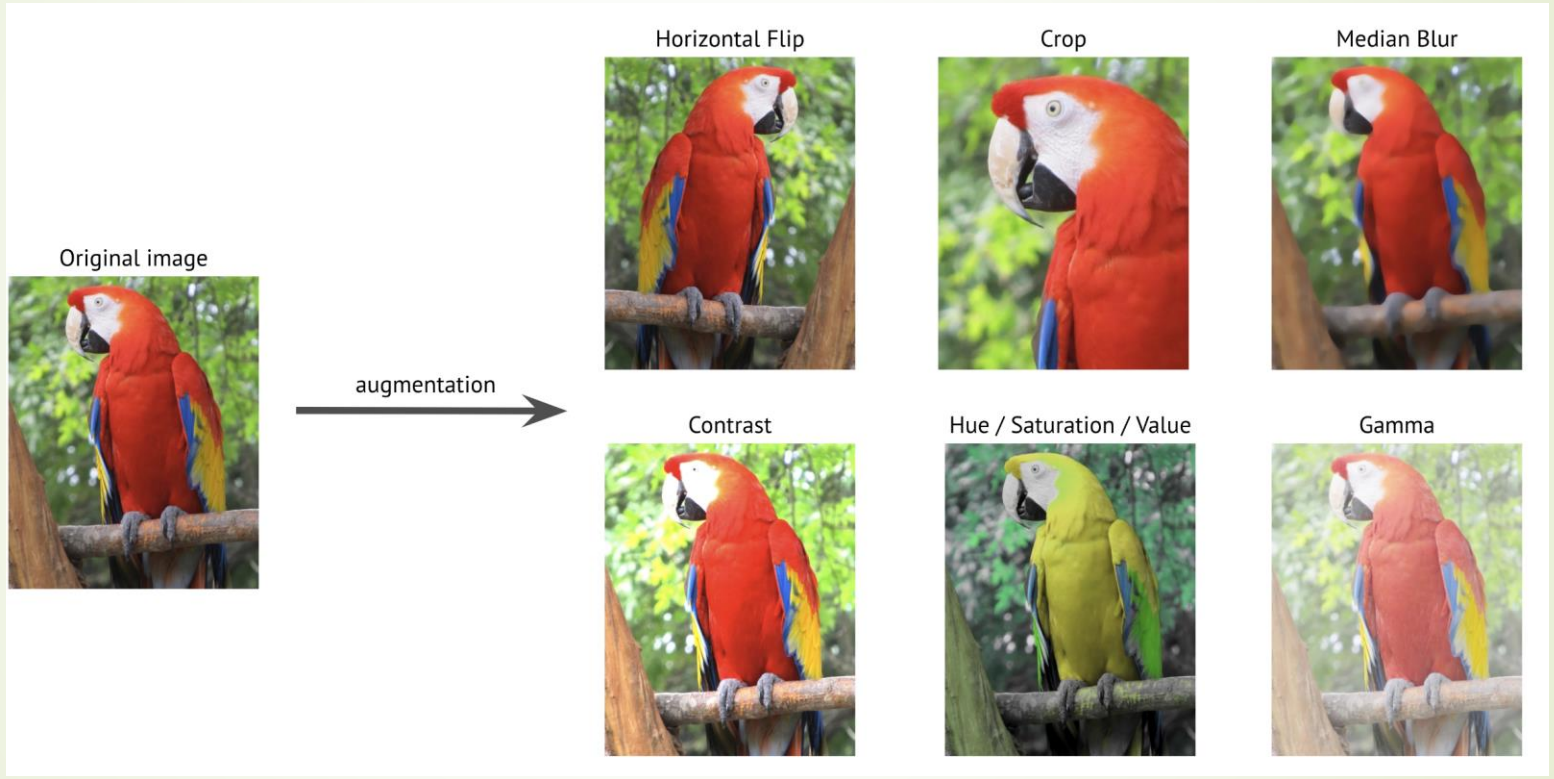
Dropout During Inference

- During inference (or testing), we want the full power of the network to be utilized—meaning, we want all neurons to be active and contribute to the prediction.
- So, dropout is turned off during inference, allowing all neurons to participate in each forward pass.
- Scaling at Inference:
  - During inference, when all neurons are active, the contributions to the output would be higher than during training if we didn't adjust for dropout.
  - To balance this, we scale the weights by the dropout rate. For a dropout rate of 0.5, for instance, we scale each neuron's weights by 0.5 during inference.
  - This scaling ensures that the activations are similar to what they were during training, even though now all neurons are contributing.

Data Augmentation

- Definition:
  - Data augmentation involves creating new training samples by applying random transformations to existing data (e.g., flipping, rotating, or cropping images).
- Effect:
  - Increases the diversity of the training data, which helps the model generalize better by reducing sensitivity to specific data characteristics.
- Examples:
  - For images: Random cropping, rotation, flipping, brightness adjustments.
  - For text: Synonym replacement, slight paraphrasing.

Data Augmentation



Ref: https://albumentations.ai/docs/introduction/image_augmentation/

Early Stopping

- Concept:
  - Stop training when the model's performance on a validation set starts to degrade, even if training accuracy continues to improve.
- How It Works:
  - Track the validation loss during training. When validation loss stops decreasing (or starts increasing), stop training.
- Benefits:
  - Prevents the model from overfitting by halting training before it starts to learn the noise in the training data.

Using a Validation Set

- Purpose:
  - A validation set helps to monitor the model's performance on unseen data during training.
- How It Helps:
  - By regularly evaluating on validation data, you can detect overfitting early and make adjustments, like stopping training or tuning hyperparameters.
- Best Practice:
  - Split the data into training, validation, and test sets to have separate data for tuning and final evaluation.

Reducing Model Complexity

- Simplify the Architecture:
  - Use fewer layers or fewer neurons in each layer to reduce model capacity.
- Effect:
  - A smaller model is less likely to memorize specific details in the training data, thus generalizing better to new data.
- When to Use:
  - When you observe overfitting, consider if the model's capacity is too large for the dataset.

Batch Normalization

- Definition:
  - Batch normalization normalizes the inputs of each layer to have a mean of 0 and a standard deviation of 1, which helps to stabilize and accelerate training.
- Benefit for Overfitting:
  - Acts as a regularizer by introducing noise due to mini-batch statistics, making it harder for the model to overfit.
- How to Apply:
  - Place batch normalization layers between the linear and activation layers in your network.

References

- Ref: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/
- https://www.cs.ucf.edu/~lboloni/Teaching/CAP4611_Fall2023/
- Ref: https://wikidocs.net/182302
- Ref: https://albumentations.ai/docs/introduction/image_augmentation/