

## Chapter 2.

# ARTIFICIAL NEURAL NETWORKS

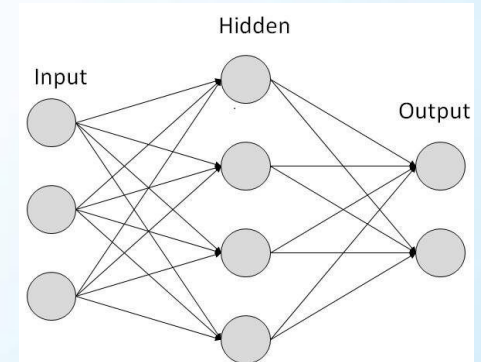
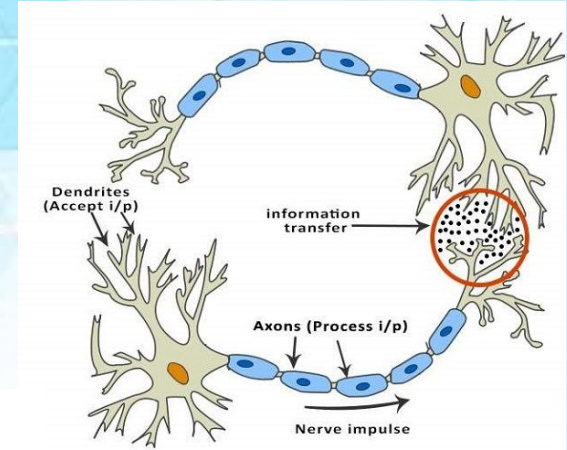
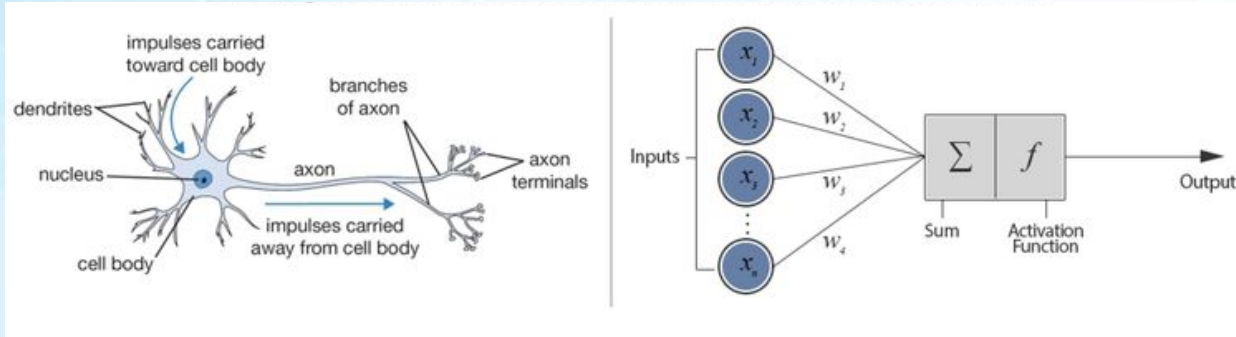
PhD. Nguyễn Thị Khánh Tiên  
tienntk@ut.edu.vn



# Introduction to Artificial Neural Networks (ANNs)

**Artificial Neural Networks (ANNs)** are inspired by the human brain, which consists of billions of interconnected neurons, and aim to mimic their learning and decision-making processes using artificial neurons and connections.

- ANNs are composed of multiple **nodes**, which imitate biological neurons of human brain.
- The output at each node is called its **activation** or **node value**.
- Each link is associated with **weight**. ANNs are capable of learning, which takes place by altering weight values.



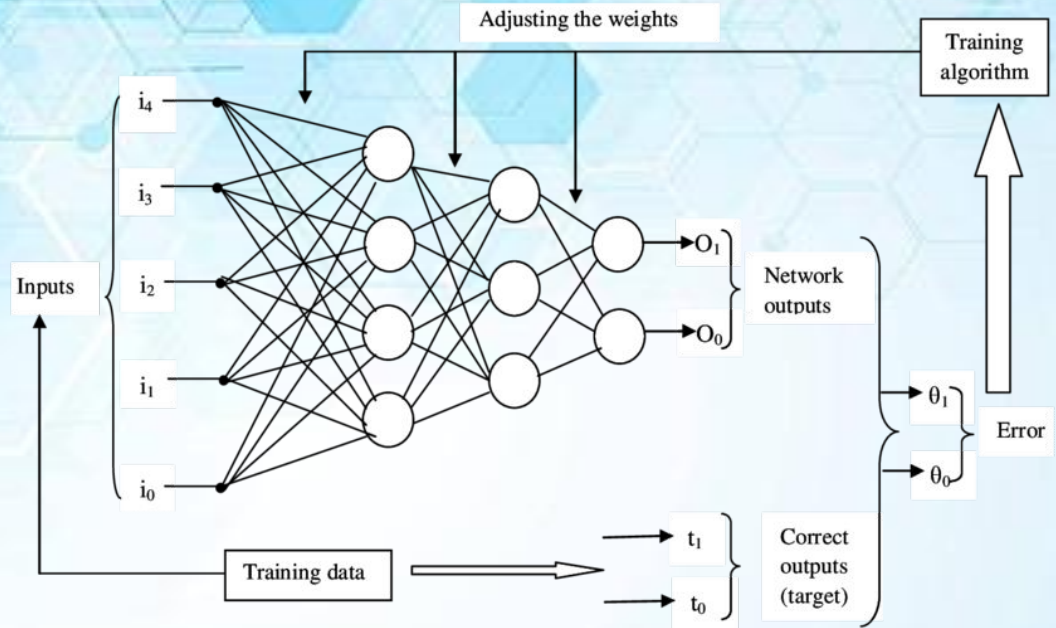
# How do neural networks work?

The nodes process input data through mathematical calculations, similar to how biological neurons transmit information.

Here's how it works:

1. **Input Layer:** initial data for the neural network. Received raw data - images or text.
2. **Hidden Layers:** These layers process the input data, extracting features and patterns.
3. **Output Layer:** Produces the final result, like a classification or prediction.
- 4.

During training, the network adjusts the connections between nodes (weights) to minimize errors in its predictions. This process, called backpropagation, allows the network to learn and improve its performance over time.



Each individual node as its own linear regression model, composed of input data, weights, a bias (or threshold), and an output.

$$\sum w_i x_i + \text{bias} = w_1 x_1 + w_2 x_2 + w_3 x_3 + \text{bias}$$

$$\text{output} = f(x) = 1 \text{ if } \sum w_l x_l + b \geq 0; 0 \text{ if } \sum w_l x_l + b < 0$$

# Back Propagation Algorithm

**Backpropagation** is a fundamental algorithm used to train neural networks. It's a supervised learning technique that adjusts the weights and biases of the network to minimize the difference between the predicted output and the actual target output.

**Here's a simplified breakdown of how it works:**

1. **Forward Pass:**

- Input data is fed into the neural network.
- The data is processed through each layer, with each neuron performing a weighted sum of its inputs and applying an activation function.
- The output of the final layer is compared to the desired output, and an error is calculated.

2. **Backward Pass:**

- The error is propagated backward through the network, layer by layer.
- At each layer, the error is used to calculate the gradient of the error with respect to the weights and biases of that layer.
- These gradients indicate the direction and magnitude of the weight and bias adjustments needed to reduce the error.

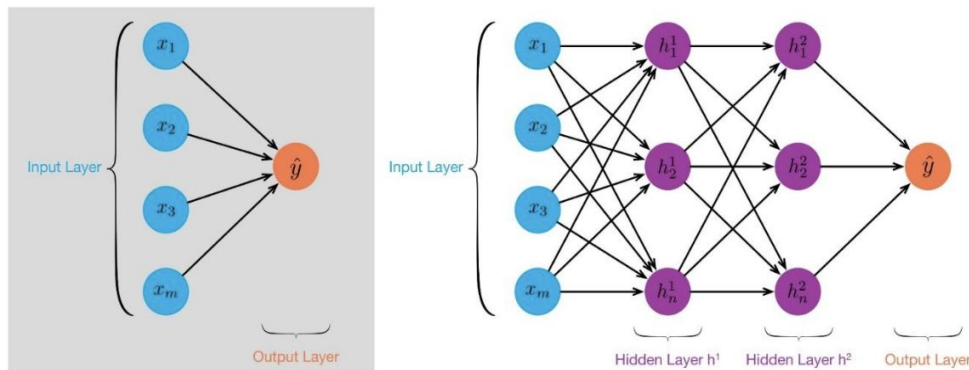
3. **Weight and Bias Update:**

- The weights and biases are updated using an optimization algorithm like gradient descent.
- This process is repeated for multiple iterations, gradually reducing the error and improving the network's accuracy.

**Key Concepts:**

- **Gradient Descent:** An optimization algorithm that iteratively adjusts parameters to minimize a cost function.
- **Learning Rate:** A parameter that controls the step size during gradient descent.
- **Activation Function:** A function that introduces non-linearity into the network, allowing it to learn complex patterns.
- **Loss Function:** A measure of how well the network is performing, used to calculate the error.

# Forward propagation (Forward pass)



First, if we have  $m$  input data ( $x_1, x_2, \dots, x_m$ ), we call this  $m$  **features**. A feature is just one variable we consider as having an influence to a specific outcome.

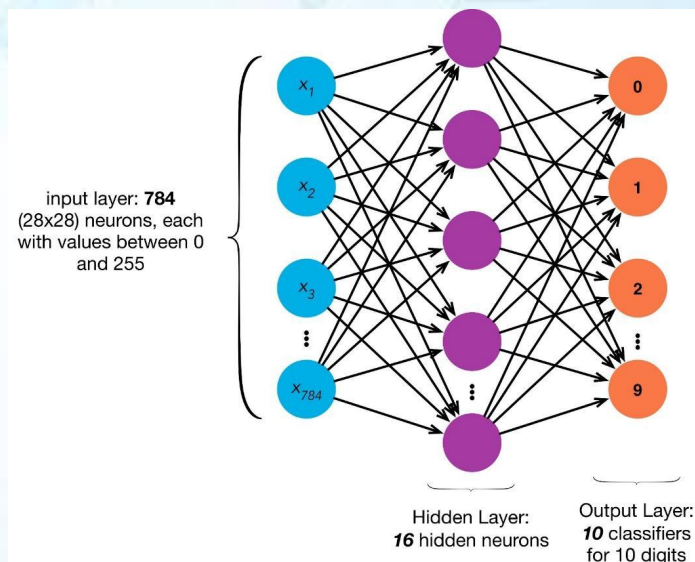
Secondly, when we multiply each of the  $m$  features with a weight ( $w_1, w_2, \dots, w_m$ ) and sum them all together, this is a **dot product**:

$$W \cdot X = w_1x_1 + w_2x_2 + \dots + w_mx_m = \sum_{i=1}^m w_i x_i$$

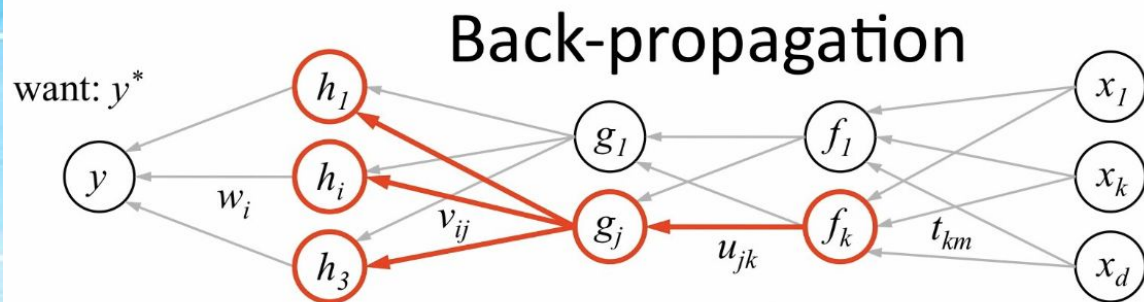
So here are the takeaways for now:

1. With  $m$  **features** in input  $X$ , you need  $m$  weights to perform a dot product
2. With  $n$  hidden neurons in the hidden layer, you need  $n$  sets of weights ( $W_1, W_2, \dots, W_n$ ) for performing dot products
3. With 1 hidden layer, you perform  $n$  dot products to get the hidden output  $h$ : ( $h_1, h_2, \dots, h_n$ )
4. Then it's just like a single-layer perceptron, we use hidden output  $h$ : ( $h_1, h_2, \dots, h_n$ ) as input data that has  $n$  **features**, perform dot product with 1 set of  $n$  weights ( $w_1, w_2, \dots, w_n$ ) to get your final output  $y_{\text{hat}}$ .

Example of A Multi-Layer Neural Network with 784 input neurons, 16 hidden neurons, and 10 output neurons.



# Backward propagation (Backward pass/Backpropagation)



1. receive new observation  $\mathbf{x} = [x_1 \dots x_d]$  and target  $y^*$
2. **feed forward:** for each unit  $g_j$  in each layer  $1 \dots L$   
compute  $g_j$  based on units  $f_k$  from previous layer:  $g_j = \sigma \left( u_{j0} + \sum_k u_{jk} f_k \right)$
3. get prediction  $y$  and error  $(y - y^*)$
4. **back-propagate error:** for each unit  $g_j$  in each layer  $L \dots 1$

(a) compute error on  $g_j$

$$\underbrace{\frac{\partial E}{\partial g_j}}_{\text{should } g_j \text{ be higher or lower?}} = \sum_i \underbrace{\sigma'(h_i)}_{\text{how } h_i \text{ will change as } g_j \text{ changes}} \underbrace{v_{ij}}_{\text{was } h_i \text{ too high or too low?}} \frac{\partial E}{\partial h_i}$$

(b) for each  $u_{jk}$  that affects  $g_j$

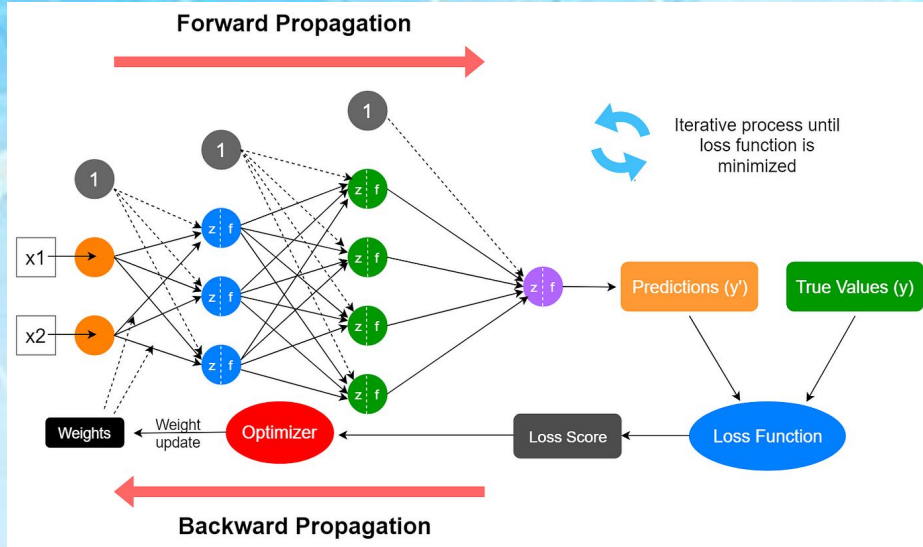
(i) compute error on  $u_{jk}$

$$\frac{\partial E}{\partial u_{jk}} = \underbrace{\frac{\partial E}{\partial g_j}}_{\text{do we want } g_j \text{ to be higher/lower}} \underbrace{\sigma'(g_j) f_k}_{\text{how } g_j \text{ will change if } u_{jk} \text{ is higher/lower}}$$

(ii) update the weight

$$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$$

# Calculation of the loss function



## 3 Key Loss Functions

- Mean Squared Error Loss Function
- Cross-Entropy Loss Function
- Mean Absolute Percentage Error

### 1. Mean Squared Error Loss Function

Mean squared error (MSE) loss function is the sum of squared differences between the entries in the prediction vector  $\vec{y}$  and the ground truth vector  $\vec{y}_{\text{hat}}$ .

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=0}^N (y_i - \hat{y}_i)^2$$

$y_i$  : entries in the prediction vector  $\vec{y}$   
 $\hat{y}_i$  : entries in the ground truth label  $\vec{y}_{\text{hat}}$

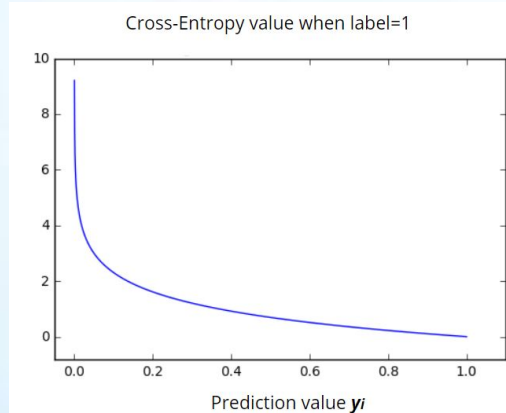
### 2. Mean Absolute Percentage Error

$$MAPE = \frac{100\%}{N} \sum_{i=0}^N \frac{|y_i - \hat{y}_i|}{\hat{y}_i}$$

### 3. Cross-Entropy Loss Function

$$\mathcal{L}(\theta) = - \sum_{i=0}^N \hat{y}_i \cdot \log(y_i)$$

$y_i$  : entries in the prediction vector  $\vec{y}$   
 $\hat{y}_i$  : entries in the ground truth label  $\vec{y}_{\text{hat}}$



# Multilayer Perceptron (MLP)

## Training Algorithms for MLPs

One of the most crucial aspects of MLPs is the training process, which involves adjusting the weights and biases of the network to minimize the error between the predicted output and the actual output.

The most widely used algorithm for training MLPs is backpropagation.

## Backpropagation

### 1. Forward Pass:

- Input data is fed into the network.
- The data propagates through the layers, with each neuron computing its activation function.
- The output layer produces a prediction.

### 2. Backward Pass:

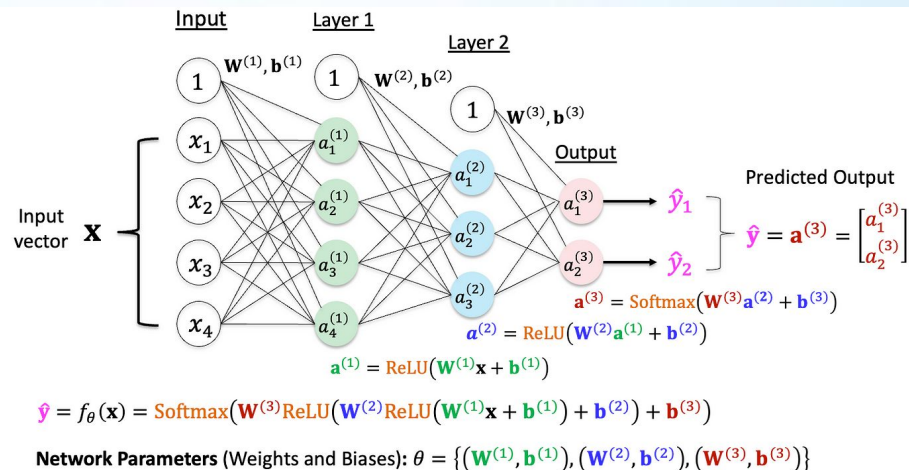
- The error between the predicted output and the actual output is calculated.
- This error is propagated backward through the network, layer by layer.
- The weights and biases of each neuron are adjusted using gradient descent to minimize the error.

## MLP example

→ [https://github.com/rcassani/mlp-example/blob/master/mlp\\_examples.ipynb](https://github.com/rcassani/mlp-example/blob/master/mlp_examples.ipynb)

**Regularization Techniques.** To mitigate overfitting, regularization techniques can be employed:

- **L1 Regularization:** Adds the absolute values of the weights to the loss function.
- **L2 Regularization (Weight Decay):** Adds the squared values of the weights to the loss function.
- **Dropout:** Randomly drops out neurons during training to prevent co-adaptation.



# Multilayer Perceptron (MLP)

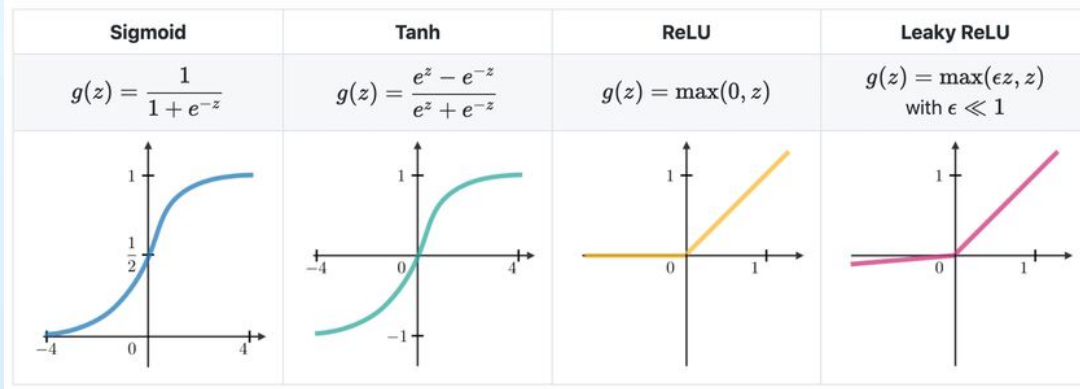
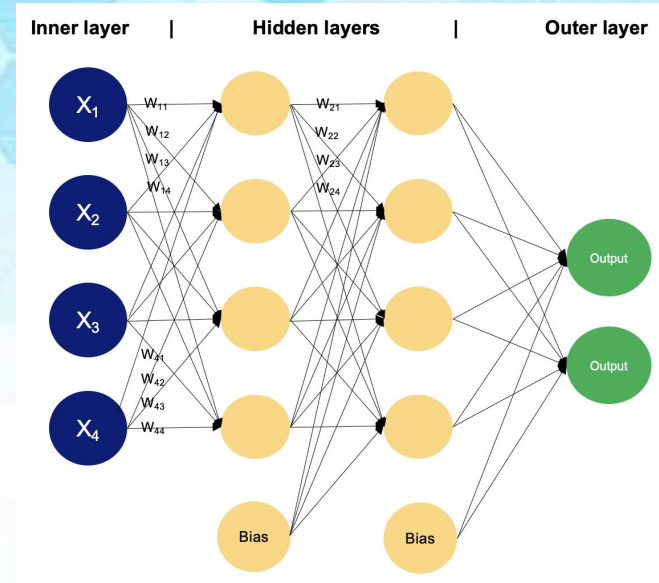
**MLPs** are a specific type of feedforward neural network that consists of multiple layers of interconnected neurons. Each neuron in a layer is connected to every neuron in the next layer. The multiple layers allow the network to extract features from the input data and gradually build up a representation of the underlying patterns.

## Key Components of an MLP:

1. **Input Layer** receives input data and passes it to the hidden layer(s).
2. **Hidden Layer(s)** process the input data and pass the results to the output layer. These layers introduce non-linearity to the network, enabling it to learn complex patterns.
3. **Output Layer** produces the final output of the network.

**Activation Functions.** Some common activation functions used in MLPs include:

- **Sigmoid:** Squashes the input to a value between 0 and 1.
- **Tanh:** Similar to sigmoid, but outputs values between -1 and 1.
- **ReLU (Rectified Linear Unit):** Outputs the maximum of 0 and the input.



$$\text{Weighted Sum} = \sum_{i=1}^n (w_i * x_i) + b$$

# Neural Networks for Classification

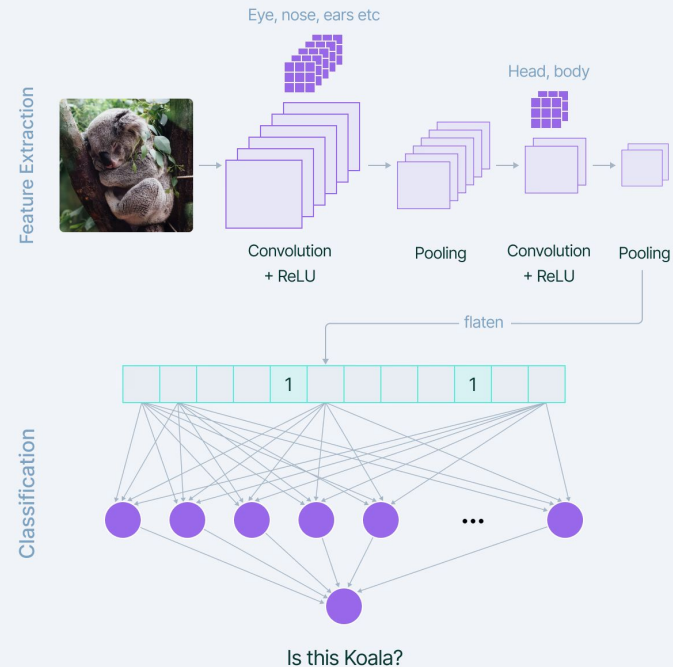
**Classification** is a type of supervised learning where the goal is to assign input data points to predefined categories or classes. Examples include:

- **Image Classification:** Determining if an image contains a cat, dog, bird, etc.
- **Spam Detection:** Classifying emails as "spam" or "not spam."
- **Sentiment Analysis:** Classifying text as "positive," "negative," or "neutral."
- **Medical Diagnosis:** Classifying a patient's condition as "disease present" or "disease absent."

**Neural networks** are particularly well-suited for classification tasks because:

- **Non-linearity:** They can learn complex, non-linear relationships between input features and output classes. This is crucial for real-world data which is often not linearly separable.
- **Feature Learning:** Deep neural networks can automatically learn relevant features from raw data, reducing the need for manual feature engineering.
- **High Accuracy:** When trained properly on sufficient data, neural networks can achieve state-of-the-art accuracy on many classification problems.
- **Scalability:** They can handle large datasets and high-dimensional input features.
- **Versatility:** The same basic neural network architecture can be adapted for various types of classification tasks (binary, multi-class, multi-label).

## Feature Extraction & Classification



# Core Components of a Neural Network for Classification

**Input Layer:** Receives the input features (the data you're using to make the classification).

- The number of neurons in the input layer corresponds to the number of features in your dataset.
- For example, if you're classifying images, the input could be the pixel values of the image (flattened into a vector).

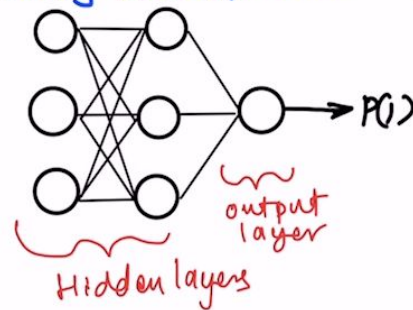
**Hidden Layers (One or More):** Perform computations on the input data through a series of interconnected neurons.

- Each neuron in a hidden layer typically does the following:
  - **Weighted Sum:** Takes the outputs from the previous layer, multiplies them by weights, and adds biases.
  - **Activation Function:** Applies a nonlinear function (like ReLU, sigmoid, tanh) to the weighted sum. This introduces non-linearity and enables the network to learn complex patterns.
- Multiple hidden layers allow the network to learn hierarchical representations of the data (deeper features).

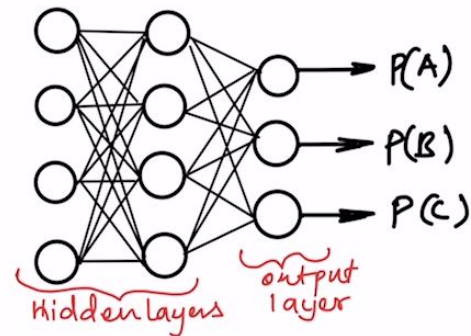
**Output Layer:** Produces the classification output.

- Number of Neurons in the output layer depends on the type of classification:
  - **Binary Classification (2 classes):** Often 1 neuron with a sigmoid activation function, outputting a probability between 0 and 1 (e.g., probability of being class '1').
  - **Multi-class Classification (more than 2 classes):** Typically as many neurons as there are classes, often with a softmax activation function. Softmax outputs a probability distribution over all classes (probabilities sum to 1).
- Activation Function in Output Layer:
  - **Sigmoid:** For binary classification, outputs a probability between 0 and 1.
  - **Softmax:** For multi-class classification, outputs a probability distribution over classes.
  - **Linear (or None):** Sometimes used if you are doing regression-like classification (though less common).

Binary classification



Multi-class classification



# Core Components of a Neural Network for Classification

## Weights and Biases:

- **Weights:** Determine the strength of the connections between neurons in different layers.
- **Biases:** Allow each neuron to have a baseline activation even when the input is zero.

## Activation Functions (Within Hidden and Output Layers):

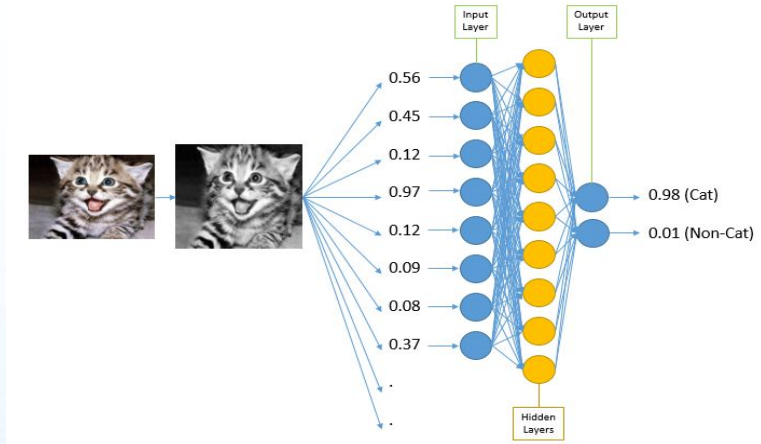
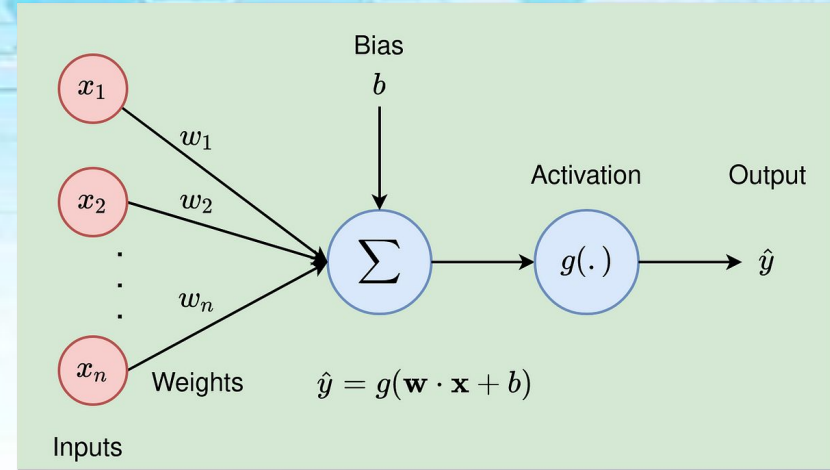
- Introduce non-linearity, enabling the network to learn complex patterns.
- Common activation functions:

**ReLU (Rectified Linear Unit):**  $f(x) = \max(0, x)$  - Simple and widely used in hidden layers.

**Sigmoid:**  $f(x) = 1 / (1 + e^{-x})$  - Outputs values between 0 and 1, often used in the output layer for binary classification.

**Tanh (Hyperbolic Tangent):**  $f(x) = \tanh(x)$  - Outputs values between -1 and 1, sometimes used in hidden layers.

**Softmax:** Used in the output layer for multi-class classification to produce probability distributions.



# Core Components of a Neural Network for Classification

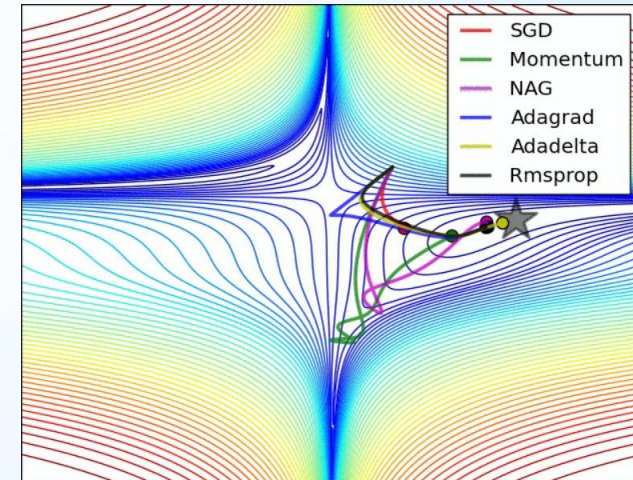
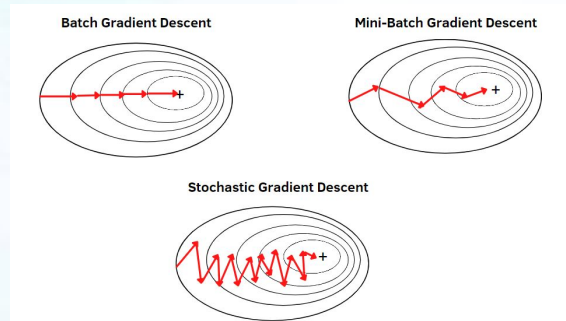
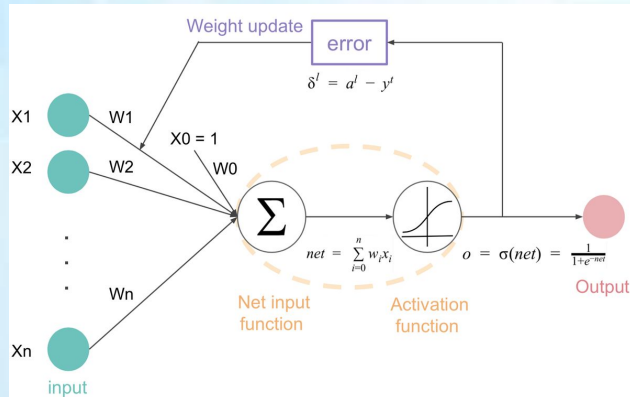
## Loss Function (Cost Function):

- Measures how well the neural network's predictions match the actual target labels during training.
- The goal of training is to minimize this loss function.
- Common loss functions for classification:
  - Binary Cross-Entropy (Log Loss),
  - Categorical Cross-Entropy (when labels are one-hot encoded),
  - Sparse Categorical Cross-Entropy (when labels are integer encoded),
  - Hinge Loss (Support Vector Machine Loss)

**Optimizer:** An algorithm that adjusts the weights and biases of the network to minimize the loss function.

Common optimizers:

- Gradient Descent (GD): Basic optimization algorithm.
- Stochastic Gradient Descent (SGD): Uses random subsets of the data (batches) to update weights, making training faster and often escaping local minima.
- Adam, RMSprop, Adagrad, etc.: More advanced optimizers that often converge faster and better than SGD in practice.

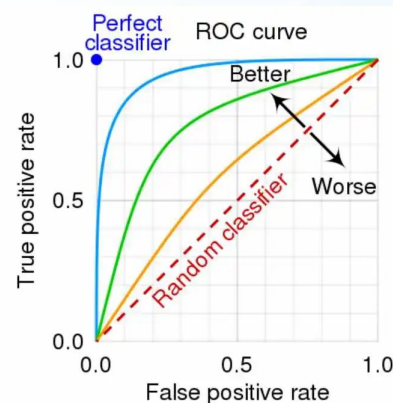


# Evaluation Metrics for Classification

After training, you need to evaluate the performance of your classification neural network on a separate test dataset (data the network hasn't seen during training). Common metrics include:

- **Accuracy:** The percentage of correctly classified instances. (Good for balanced datasets, can be misleading for imbalanced datasets).
- **Precision:** Out of all instances predicted as positive, what proportion was actually positive? (Important when minimizing false positives is crucial).
- **Recall (Sensitivity):** Out of all actual positive instances, what proportion was correctly predicted as positive? (Important when minimizing false negatives is crucial).
- **F1-Score:** The harmonic mean of precision and recall. Provides a balanced measure of precision and recall.
- **Area Under the ROC Curve (AUC-ROC):** For binary classification, measures the ability of the classifier to distinguish between the two classes across different thresholds. (Good for imbalanced datasets and when you want to understand performance across thresholds).
- **Confusion Matrix:** A table showing the counts of true positives, true negatives, false positives, and false negatives. Provides a detailed breakdown of classification performance per class.

		Predicted condition		
		Positive	Negative	
True condition	Total accidents (TA)			
	Positive	True positive (TP)	False Negative (FN)	$TPR, Recall = \frac{\sum TP}{\sum TP + FN}$
	Negative	False Positive (FP)	True negative (TN)	$FPR = \frac{\sum FP}{\sum FP + TN}$
$Accuracy = \frac{\sum TP + TN}{\sum TA}$		$Precision = \frac{\sum TP}{\sum TP + FP}$	$F - score = \frac{\sum 2TP}{\sum 2TP + FP + FN}$	



# Practical Considerations and Architectures NN for Classification

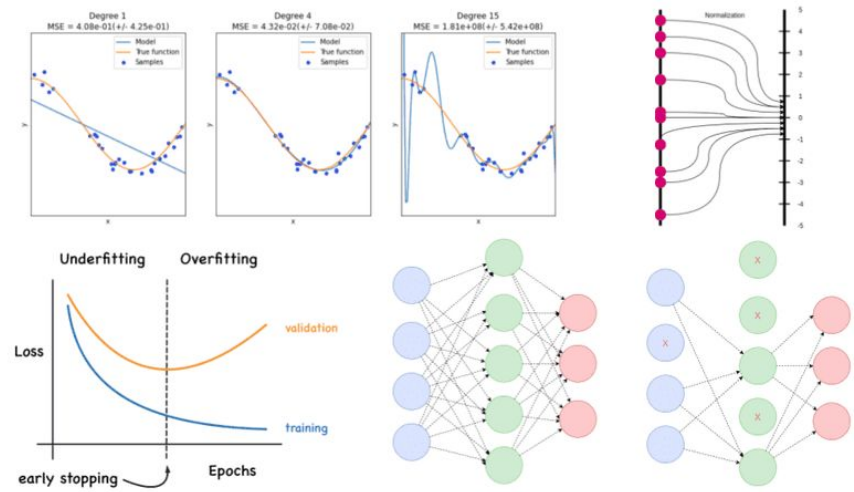
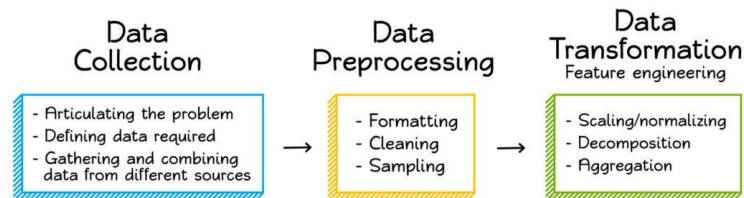
**Data Preprocessing:** Essential for good performance. Includes:

- **Normalization/Scaling:** Scaling input features to a similar range (e.g., 0-1 or -1 to 1).
- **Handling Categorical Features:** One-hot encoding, embedding layers.
- **Handling Missing Values:** Imputation techniques.

**Overfitting and Regularization:** Neural networks can easily overfit to the training data. Techniques to prevent overfitting include:

- **Regularization (L1, L2):** Adding penalties to the loss function based on the magnitude of weights.
- **Dropout:** Randomly dropping out neurons during training.
- **Early Stopping:** Monitoring performance on a validation set and stopping training when performance starts to degrade.
- **Data Augmentation:** Creating more training data by applying transformations to existing data (e.g., image rotations, flips).

## Data Preparation Process



# Neural networks for image classification

When discussing multilayer neural networks in the context of image classification, it's essential to understand the role of Multi-Layer Perceptrons (MLPs) and how they relate to the more commonly used Convolutional Neural Networks (CNNs).

## Multi-Layer Perceptrons (MLPs)

- **Basic Structure:**

- An MLP consists of an input layer, one or more hidden layers, and an output layer.
- It's a "fully connected" network, meaning each neuron in one layer is connected to every neuron in the next layer.

- **Image Processing with MLPs:**

- To use an MLP for image classification, the image's pixel data is typically "flattened" into a one-dimensional vector. This means the 2D image structure is lost.
- The flattened vector is then fed into the input layer of the MLP.
- The network processes this data through its hidden layers, applying weighted sums and activation functions, and finally produces a classification output.

- **Limitations for Images:**

- MLPs can be used for image classification, but they have significant limitations compared to CNNs.
- **Loss of Spatial Information:** Flattening the image disregards the spatial relationships between pixels, which are crucial for recognizing visual patterns.
- **High Computational Cost:** For high-resolution images, the flattened vector becomes very large, leading to a massive number of connections and parameters in the MLP, making training computationally expensive.
- **Inefficiency in feature extraction:** MLPs do not have built in methods of feature extraction like CNN's do.

→ MLPs can technically be used for image classification, but they are not well-suited for the task due to the loss of spatial information and high computational cost.

→ CNNs are the dominant architecture for image classification because they are designed to effectively process visual data and learn hierarchical representations of images

# Neural Network for Regression

Neural networks offer a powerful tool for regression problems, especially when dealing with complex, non-linear relationships in data. They learn complex relationships between input and output variables, making them suitable for predicting continuous values.

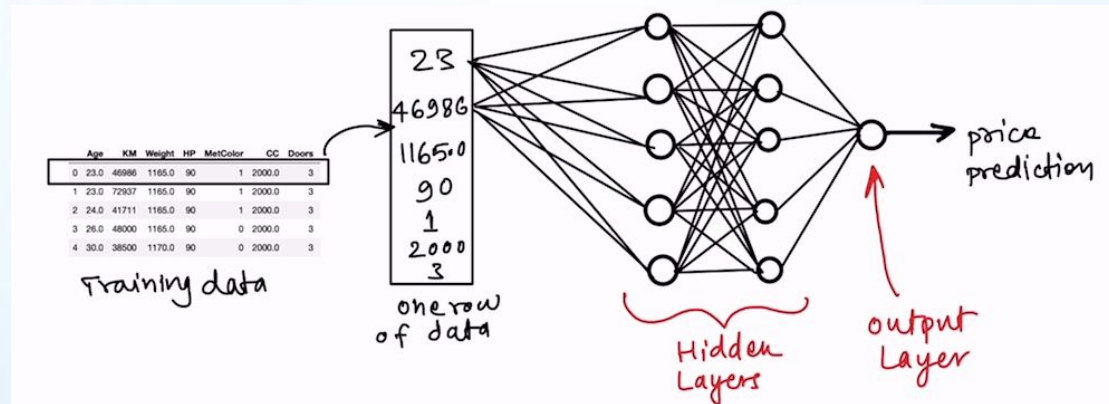
**Architecture:** Regression neural networks typically have an input layer, one or more hidden layers, and an output layer with a single node. The number of nodes in the input layer corresponds to the number of input features, and the single output node predicts the continuous target variable.

**Activation Functions:** Hidden layers commonly use activation functions like ReLU (Rectified Linear Unit) or sigmoid to introduce non-linearity, enabling the network to learn complex patterns. The output layer in regression tasks usually has a linear activation function or no activation function to allow for a wide range of output values.

**Loss Function:** Mean Squared Error (MSE) is the most common loss function for regression neural networks. It measures the average squared difference between the predicted and actual values. Other loss functions like Mean Absolute Error (MAE) can also be used.

**Training Process:** The network learns by adjusting its weights and biases during training to minimize the chosen loss function. Optimization algorithms like Adam or SGD (Stochastic Gradient Descent) are used for this process.

**Applications:** Regression neural networks are used in various fields, such as **Predicting stock prices**, **Sales forecasting**, **House price prediction**, **Demand forecasting**: Predicting future demand for products or services, **Energy consumption prediction**: Forecasting energy usage in buildings or systems.



# Introduction to a complete ML workflow implemented in PyTorch

## 0. Quickstart

## 1. Tensors

## 2. Datasets and DataLoaders

## 3. Transforms

## 4. Build Model

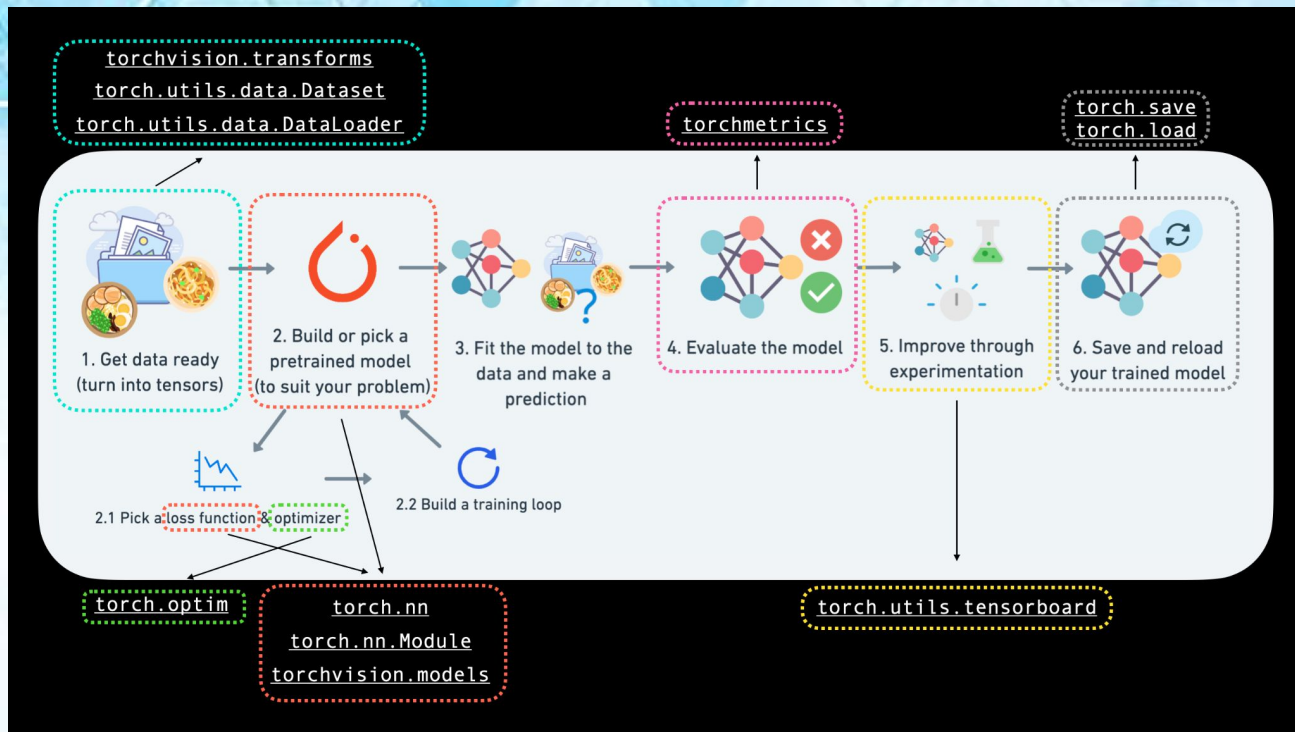
## 5. Automatic Differentiation

## 6. Optimization Loop

## 7. Save, Load and Use Model

Tutorial:

<https://pytorch.org/tutorials/beginner/basics/intro.html>



# Practice 1

## Exercise: PyTorch FashionMNIST Classification

1. **Learn:** Study PyTorch Tensors, Datasets/Loaders, Transforms, Model Building, Autograd, Optimization, and Model Saving/Loading.
2. **Implement:**
  - Load FashionMNIST dataset.
  - Apply transforms.
  - Build a neural network.
  - Create a training loop (forward, loss, backward, optimize).
  - Evaluate model accuracy.
  - Save and load the model.
3. **Tasks:**
  - Experiment with network/hyperparameters.
  - Visualize loss.
  - Display predicted vs actual images.
4. **Deliver:** Python code, brief report, loss graphs, image displays.
5. **Use:** PyTorch docs/tutorials.