Introduction to Deep Learning

UWM College of Engineering & Applied Science

What is Intelligence?

- What is intelligence?
- intelligence can be seen as the **information-processing capability that enables effective, informed, and adaptive decision-making**.

Real-World Example

- In humans, intelligence might mean noticing that it's going to rain, deciding to bring an umbrella, and learning to check the weather before leaving the house.
- In AI, intelligence might mean a recommendation algorithm analyzing user preferences and past behaviors to suggest the most relevant content or product.

What is Deep Learning?

- A sunset of Machine Learning that uses Neural Networks to process very large dataset and make decisions.

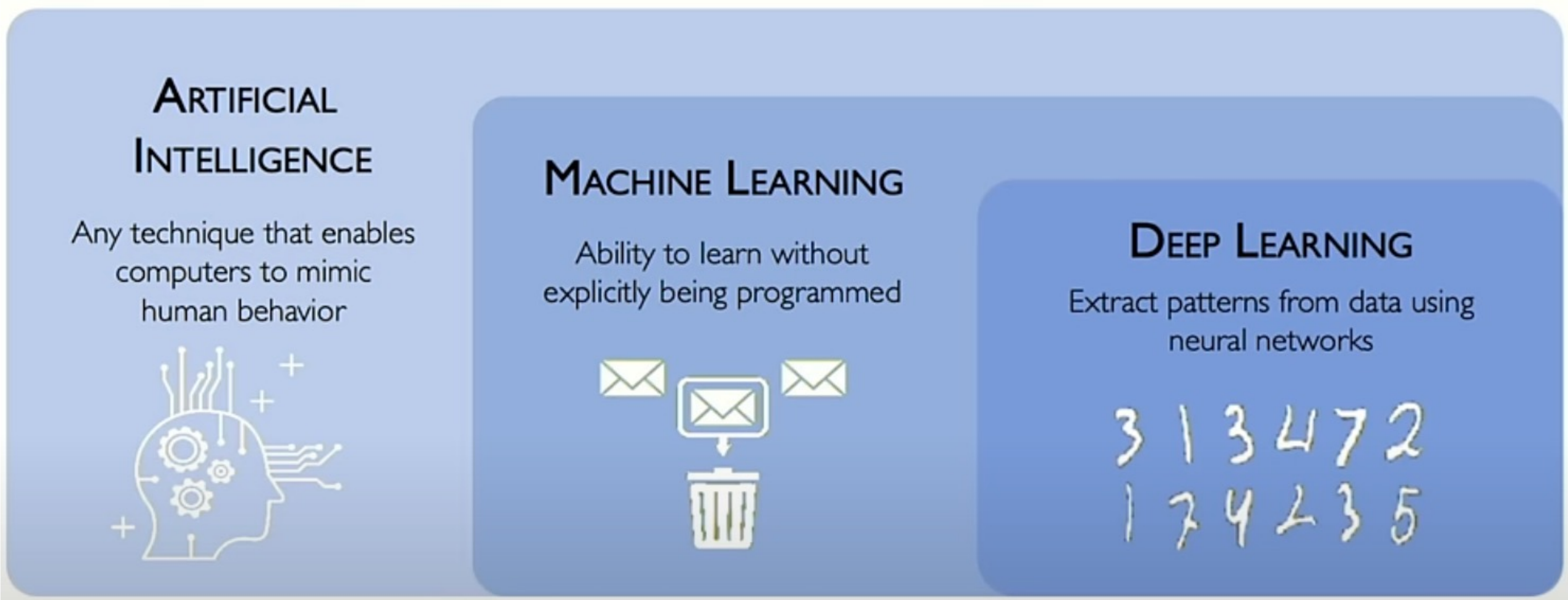


Image credit

What is Deep Learning?

- Deep learning is a subset of machine learning where neural networks with many layers learn from large amounts of data.
- These models are data hungry and computational hungry!
- **Purpose:** To enable machines to automatically learn complex patterns and representations from data.

- Neural networks vs. deep learning
- The word “deep” in deep learning is referring to the depth of layers in a neural network.
- A neural network that consists of more than three layers—which would be inclusive of the inputs and the output—can be considered a deep learning algorithm.

What is a Neural Network?

- Neural networks are computational models inspired by the human brain, used to recognize patterns and make decisions.

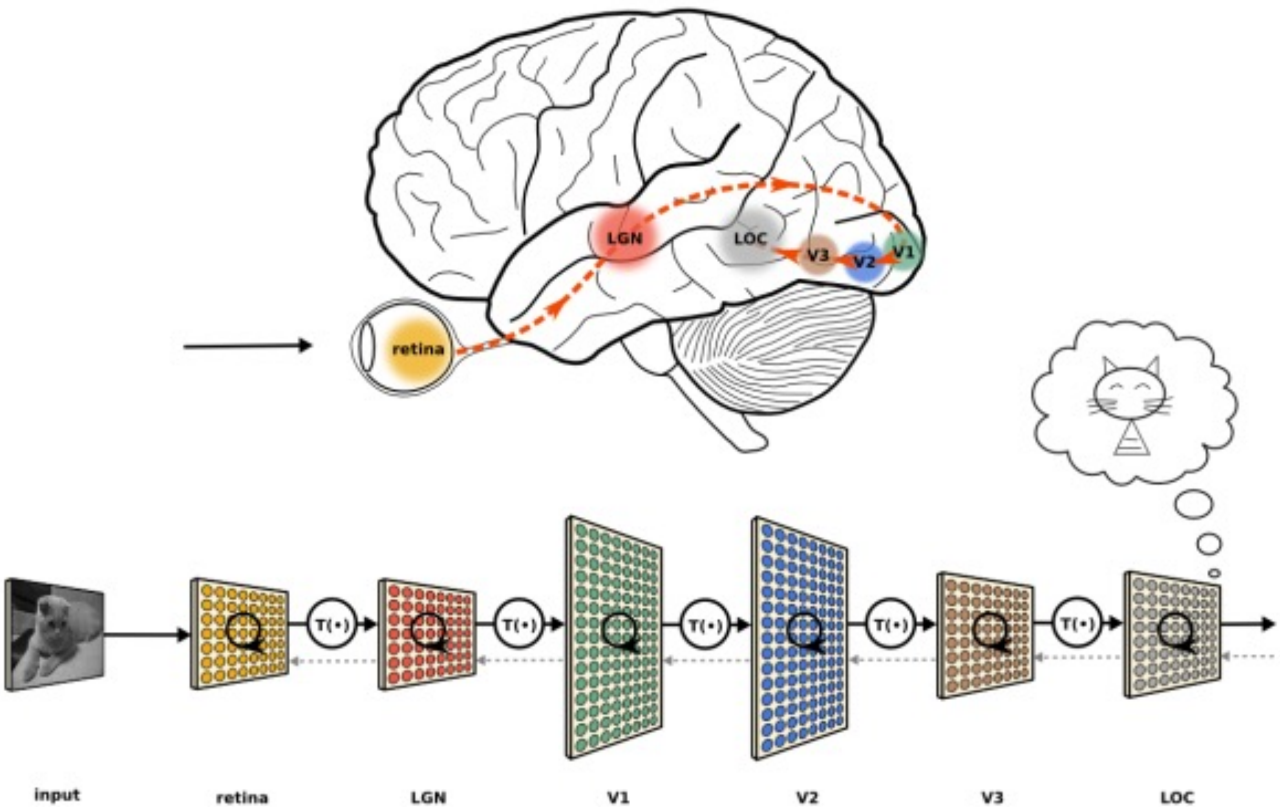


Image credit

What is a Neural Network?

In the Human Brain:

1. **Retina**: The eye captures the image and sends it to the brain as raw visual data.
2. **Basic Features**: The first layer in the brain (V1) detects simple things like edges and lines.
3. **More Complex Shapes**: As the data moves through deeper layers (V2, V3, etc.), the brain starts recognizing shapes and patterns, like a curve or part of an ear.
4. **Full Object Recognition**: In the final layers, the brain combines all the parts into a full picture and recognizes it as a "cat."

In a Deep Learning Model:

1. **Input Layer**: The model receives the image as raw pixel data.
2. **Basic Patterns**: The first layers detect edges and textures, like the outline of a face or whiskers.
3. **Complex Features**: Middle layers recognize more specific parts of the cat, such as an eye or ear.
4. **Object Recognition**: The final layers combine everything learned to recognize the full image as a "cat."

Structure of Neural Networks

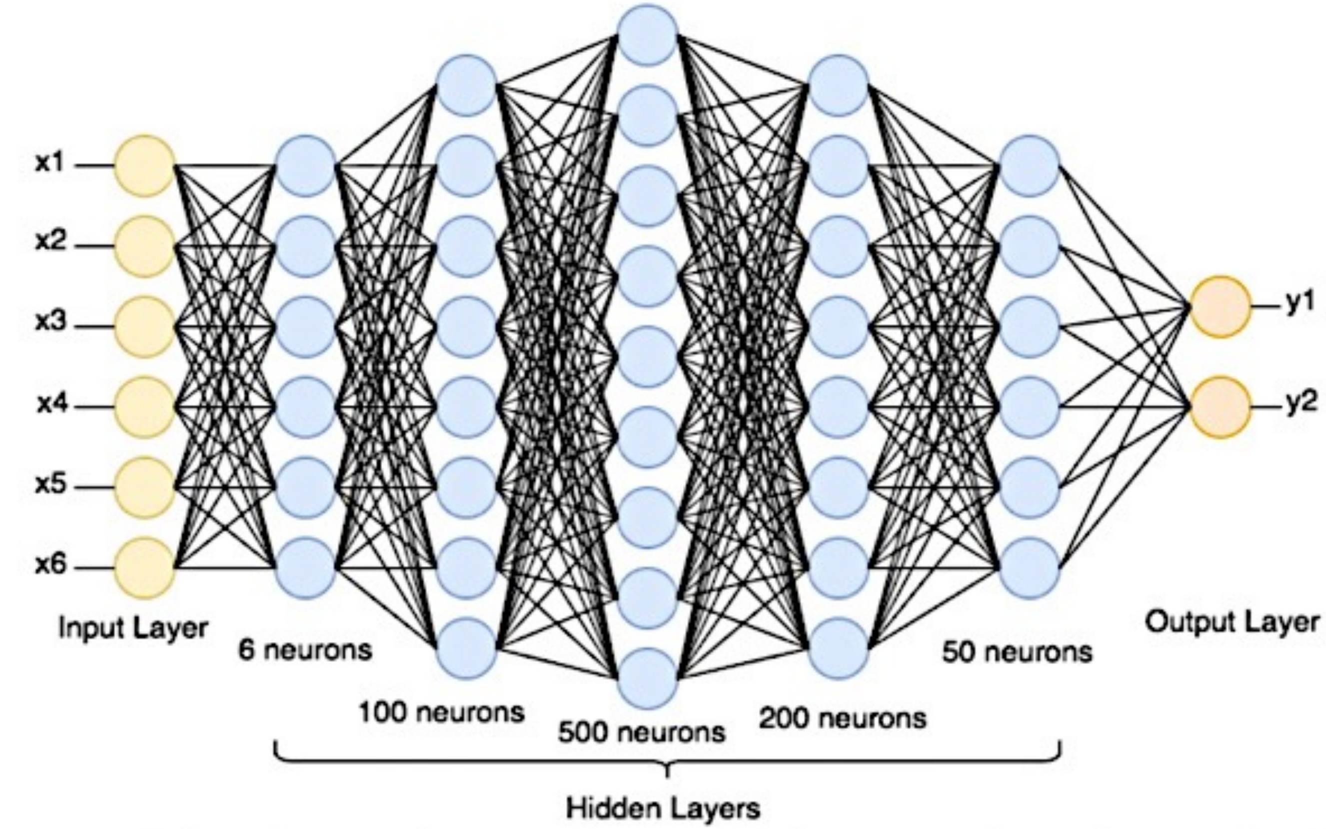


Image credit

UWM College of Engineering & Applied Science

Perceptron

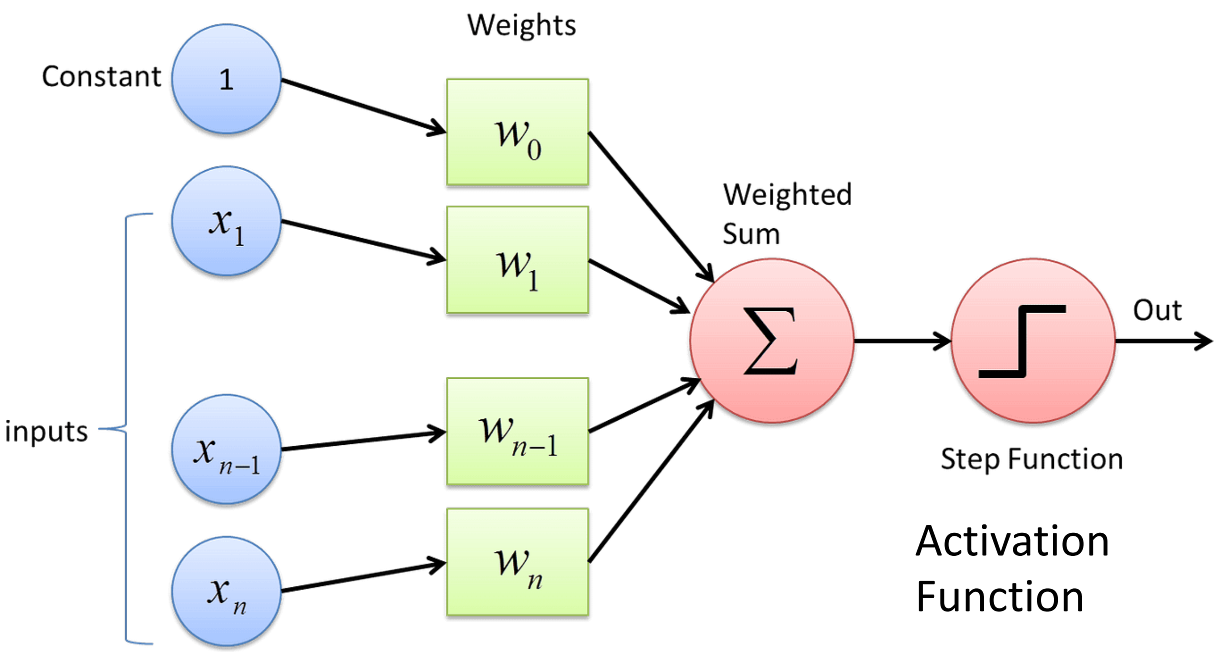
- Perceptron: A perceptron is the simplest form of a neural network. A building block!

- Developed by Frank Rosenblatt in the 1950s
- **Goal:** Classify data as 0 or 1 (binary classification)
- **Structure:** Takes inputs, multiplies them by weights, adds a bias, and produces an output
- **Formula:**

$$f(x) = 1 \text{ if } (w \cdot x + b) > 0, \text{ otherwise } 0$$

  - b is bias

Perceptron



UWM College of Engineering & Applied Science

Activation Function

- Purpose: Adds non-linearity, enabling the network to learn complex patterns
- Common Functions: Common functions include ReLU, Sigmoid, Softmax, and Tanh.
- Threshold: Neurons are activated if the output exceeds a certain threshold.

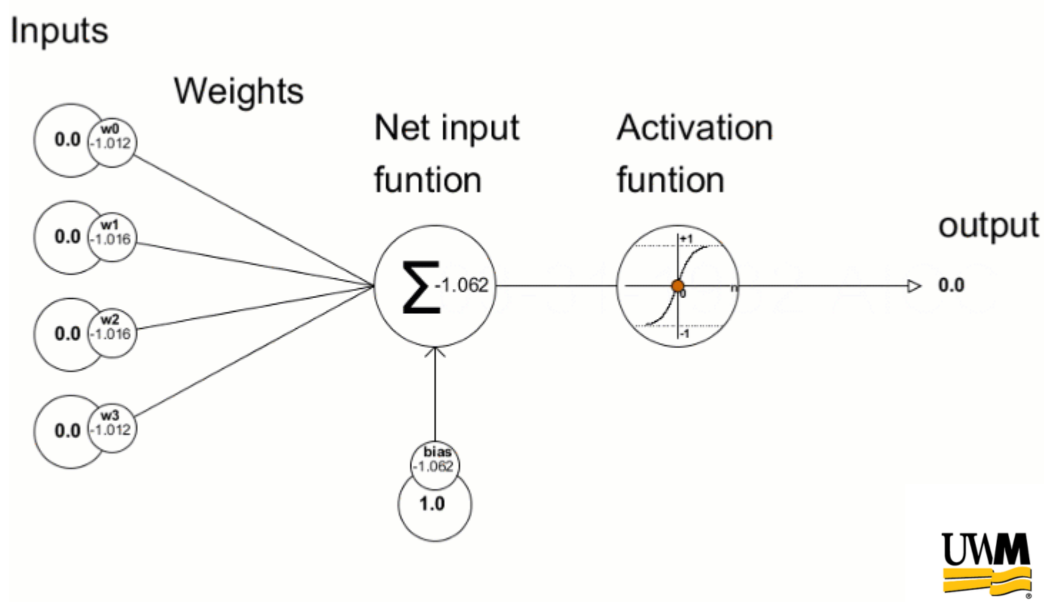


Image credit

Activation Function

Activation Function	Output Range	Common Use Case
Sigmoid	0 to 1	Binary classification output layer
Tanh	-1 to 1	Hidden layers for centered data
ReLU	0 to $+\infty$	Hidden layers in deep networks
Leaky ReLU	$-\infty$ to $+\infty$	Hidden layers with small negative values allowed
Softmax	0 to 1 (sum = 1)	Multi-class classification output
Linear	$-\infty$ to $+\infty$	Regression output layer

Perceptron

```
import numpy as np

class Perceptron(object):

    def __init__(self, no_of_inputs, threshold=100,
learning_rate=0.01):
        self.threshold = threshold
        self.learning_rate = learning_rate
        self.weights = np.zeros(no_of_inputs + 1)  # Initialize
weights to zeros

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) +
self.weights[0]  # Include bias term
        activation = 1 if summation > 0 else 0  # Simplified
activation calculation
        return activation

    def train(self, training_inputs, labels):
        for _ in range(self.threshold):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                # Update weights including bias term
                update = self.learning_rate * (label -
prediction)
                self.weights[1:] += update * inputs
                self.weights[0] += update
```

UWM College of Engineering & Applied Science

Limitation of a Single Perceptron

- **Only works for linearly separable data**
- **Example:** XOR problem (exclusive OR) - cannot separate with a straight line

The XOR Problem and the Need for Multi-Layer Perceptrons

- **The XOR Problem**:
  - We have four points in a 2D space: **(0,0), (0,1), (1,0), and (1,1)**.
  - The goal: Separate the points where the XOR of the coordinates equals 1 from those where it equals 0.
- **Key Challenge**:
  - **XOR is Not Linearly Separable**: A single perceptron can only create a straight line boundary, but XOR data needs a non-linear boundary.
- **Why This Matters**:
  - A single perceptron **cannot solve the XOR problem** because it can only classify l separable data.

Multi-Layer Perceptron

- **What is MLP?** To overcome the limitations of a single perceptron and further enhance its accuracy, we can use a multi-layer perceptron (MLP).

- **Structure:** Multiple layers - input, hidden, and output
  Each of these layers consists of multiple neurons, and each neuron in a layer is connected to every neuron in the next layer.

UWM | College of Engineering & Applied Science

Multi-Layer Perceptron

Example MLP Code with Keras

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# Placeholder data (replace with your actual data)
X = np.random.rand(100, 8)
y = np.random.randint(2, size=(100, 1))

# Create a Sequential model
model = Sequential()

# Add an input layer and a hidden layer
model.add(Dense(32, input_dim=8, activation='relu'))

# Add an output layer
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Fit the model
model.fit(X, y, epochs=150, batch_size=10)
```

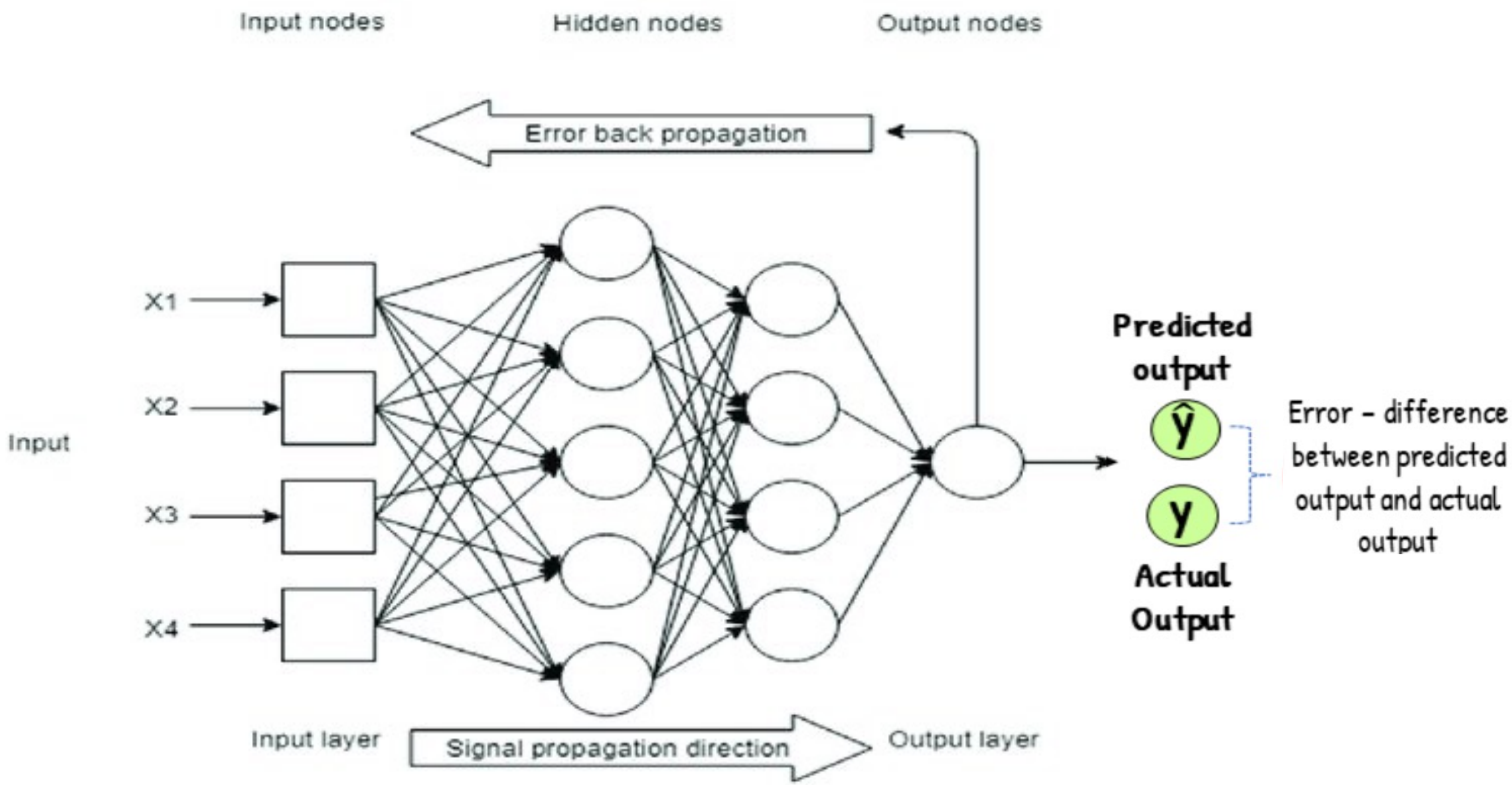
Training a Neural Network

How NN works?

1. Forward Propagation:
   – The network takes an input and computes the predicted output.

2. Loss Calculation:
   – calculated after forward propagation
3. Backpropagation:
   – Using the error calculated from the loss function, the network computes gradients with respect to each weight and bias.
   – These gradients are then used to update the weights and biases to reduce the error in future predictions.

Training a Neural Network



Training a Neural Network
Forward Propagation

More details for Forward Propagation:

1. input Layer:
   - The input data is fed into the network.
2. Hidden Layers:
   - The input data is transformed by the neurons in the hidden layers using weights, biases, and activation functions.
   - For each neuron, compute: $z = \sum(w \cdot x) + b$

     where w is the weight, x is the input, and b is the bias.
   - Apply an activation function (e.g., ReLU, sigmoid) to z:

$$a = \text{activation}(z)$$

3. Output Layer:
   - The final layer transforms the activations of the last hidden layer into the output predictions.

Training a Neural Network
Loss Function

Loss Calculation:

- The loss function is used to calculate the error between the predicted output and the actual target values.

- This step happens after forward propagation and before backpropagation.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

y_i is the actual target value for the i-th sample.

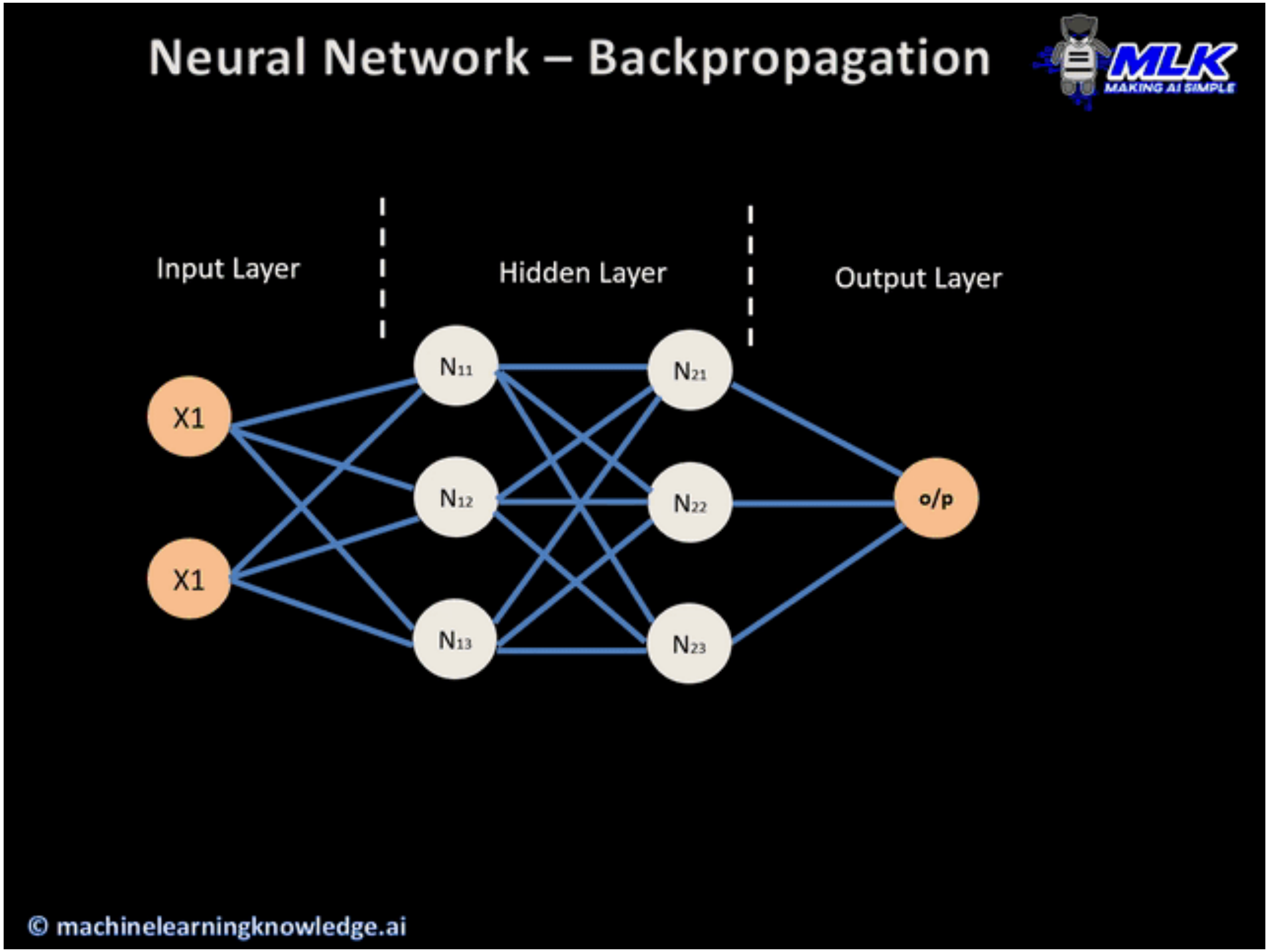
$\hat{y}_i$ is the predicted value for the i-th sample.

n is the number of samples.

- The calculated loss value is used to quantify how well the network is performing.
- There are other functions like Binary Cross-Entropy and Categorical Cross-Entropy for calculating loss.

Training a Neural Network

Neural networks learn by making mistakes, learning from them, and adjusting their parameters repeatedly to improve and achieve more accurate predictions.



Training a Neural Network
Back Propagation

More details on Backpropagation:

- Output Layer Gradient:
  - Calculate the gradient of the loss with respect to the output.
- Hidden Layer Gradients:
  - Propagate the gradient back through the network to calculate the gradients with respect to the hidden layers.
- Weight and Bias Updates:
  - Update the weights and biases using the computed gradients and a learning rate.

Training a Neural Network
Gradient Descent

What is Gradient Descent?

- A widely-used optimization algorithm to minimize the error or cost function.

Goal: Iteratively move in the direction of steepest descent (negative of the gradient) to find optimal parameters.

How it Works:

- **Error Function**: Measures the difference between predicted and actual values.
- **Gradient**: Calculates the slope of the error function with respect to model weights.

Training a Neural Network
Gradient Descent

- **Update Rule**:

$$w = w - \alpha \cdot \nabla J(w)$$

  - w: Current weights
  - α: Learning rate (controls step size)
  - $\nabla J(w)$: Gradient of the error function with respect to w

Purpose in Neural Networks:

- Used during training to adjust weights in each layer to minimize prediction errors and improve accuracy.

Training a Neural Network
Variants of Gradient Descent

1. **Batch Gradient Descent**:
   - Calculates the gradient using the entire training dataset.
   - **Pros**: Smooth convergence, accurate updates.
   - **Cons**: Computationally expensive for large datasets.
2. **Stochastic Gradient Descent (SGD)**:
   - Calculates the gradient using one training example at a time.
   - **Pros**: Faster updates, more flexible with large datasets.
   - **Cons**: Updates can be noisy, leading to less stable convergence.
3. **Mini-Batch Gradient Descent**:
   - Compromise between batch and stochastic gradient descent.
   - Uses a small batch of examples for each update.
   - **Pros**: Faster than batch gradient descent with less noise than SGD.
   - **Common Choice**: Often used in deep learning for its balance of efficiency and stability.

Learning Rate in NN

- **Definition**: The learning rate is a critical hyperparameter in neural networks that controls the step size during weight updates.
- **Purpose**: Determines how quickly or slowly the model moves toward minimizing the error (loss function).

Key Points

- **High Learning Rate**:
  - Pros: Faster convergence.
  - Cons: Risk of overshooting the minimum, leading to suboptimal solutions.
- **Low Learning Rate**:
  - Pros: More precise convergence.
  - Cons: Slower learning, which can make training time-consuming.

Learning Rate in NN

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

# Create a Sequential model
model = Sequential()

# Add an input layer and a hidden layer
model.add(Dense(32, input_dim=8, activation='relu'))

# Add an output layer
model.add(Dense(1, activation='sigmoid'))

# Define the optimizer with a learning rate of 0.01
sgd = SGD(lr=0.01)

# Compile the model
model.compile(loss='binary_crossentropy', optimizer=sgd,
metrics=['accuracy'])

# Fit the model
model.fit(X, y, epochs=150, batch_size=10)
```

Learning Rate in NN

Here is an example of the output of the code:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/150
60000/60000 [==============================] - 2s 33us/sample -
loss: 0.6558 - accuracy: 0.5782 - val_loss: 0.6045 -
val_accuracy: 0.6224
Epoch 2/150
60000/60000 [==============================] - 2s 33us/sample -
loss: 0.5949 - accuracy: 0.6344 - val_loss: 0.5752 -
val_accuracy: 0.6318
...
```

Explanation

- The learning rate can significantly affect model performance. Fine-tuning this parameter is often necessary to balance training speed and accuracy.
- **Output Expectations**:
  - A well-chosen learning rate improves convergence and accuracy.
  - Example output: Training accuracy might be high, but tuning the learning rate and using regularization helps generalize to test data.

Choosing the Right Optimizer

Optimizers control how weights are updated during backpropagation. They help improve the model's performance and convergence speed.

Types of Optimizers

1. **Momentum**: Speeds up gradient descent by incorporating past updates to move faster in the correct direction.

2. **Nesterov Accelerated Gradient (NAG)**: Adds momentum but looks ahead to adjust the direction more intelligently.

3. **Adagrad**: Adapts learning rates based on frequency of parameter updates, useful for sparse data.

UWM College of Engineering & Applied Science

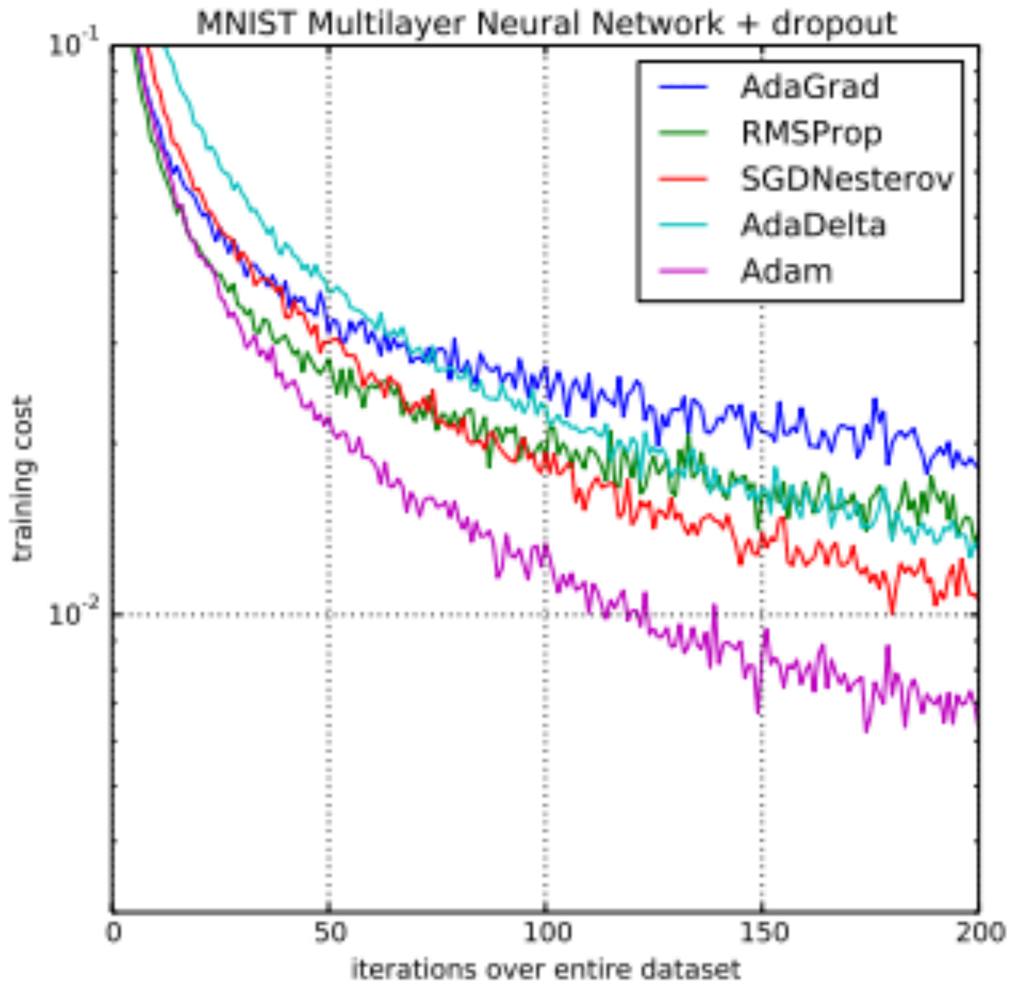
Advanced Optimizers in NN

- **RMSprop**: Controls oscillations by adjusting learning rates based on recent gradient magnitudes, ideal for deep networks.

- **Adam (Adaptive Moment Estimation)**:
  - Combines momentum and RMSprop advantages.
  - Offers faster convergence and bias-correction for stable learning.

```
from keras.optimizers import Adam
adam = Adam(lr=0.01)
```

Choosing the Right Optimizer



Hyperparameter Tuning

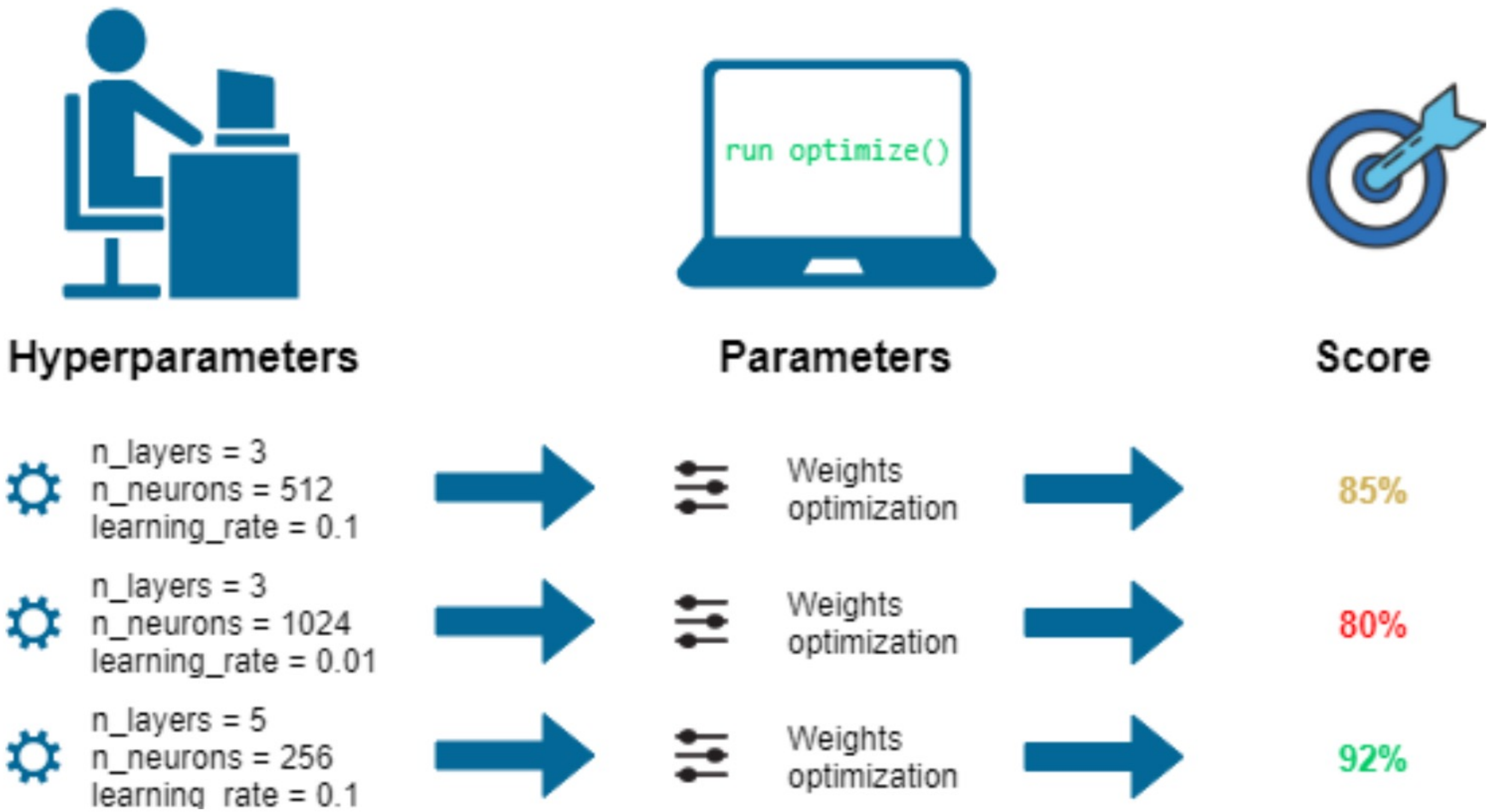
- **Definition**: The process of finding the best set of hyperparameters (e.g., learning rate, optimizer type, layer sizes) for optimal model performance.

Common Tuning Techniques

1. **Grid Search**: Exhaustive search over a specified parameter grid; very thorough but computationally expensive.
2. **Random Search**: Randomly selects parameter combinations, quicker and effective for large search spaces.
3. **Bayesian Optimization**: Builds a probabilistic model to choose promising hyperparameters iteratively, balancing exploration and exploitation.

Visual: Simple diagrams for each technique, showing their search process over a parameter grid.

Hyperparameter Tuning



source

UWM College of Engineering & Applied Science