```
import numpy as np
from sklearn.neural_network import MLPClassifier

# Define training data for digits 0–9
digits = [
    [[1, 1, 1],
     [1, 0, 1],
     [1, 0, 1],
     [1, 0, 1],
     [1, 1, 1]],  # 0

    [[0, 1, 0],
     [1, 1, 0],
     [0, 1, 0],
     [0, 1, 0],
     [1, 1, 1]],  # 1

    [[1, 1, 1],
     [0, 0, 1],
     [1, 1, 1],
     [1, 0, 0],
     [1, 1, 1]],  # 2

    [[1, 1, 1],
     [0, 0, 1],
     [0, 1, 1],
     [0, 0, 1],
     [1, 1, 1]],  # 3

    [[1, 0, 1],
     [1, 0, 1],
     [1, 1, 1],
     [0, 0, 1],
     [0, 0, 1]],  # 4

    [[1, 1, 1],
     [1, 0, 0],
     [1, 1, 1],
     [0, 0, 1],
     [1, 1, 1]],  # 5

    [[1, 1, 1],
     [1, 0, 0],
     [1, 1, 1],
     [1, 0, 1],
     [1, 1, 1]],  # 6
```

```
    [[1, 1, 1],
     [0, 0, 1],
     [0, 1, 0],
     [1, 0, 0],
     [1, 0, 0]],  # 7

    [[1, 1, 1],
     [1, 0, 1],
     [1, 1, 1],
     [1, 0, 1],
     [1, 1, 1]],  # 8

    [[1, 1, 1],
     [1, 0, 1],
     [1, 1, 1],
     [0, 0, 1],
     [1, 1, 1]]   # 9
]

# Convert to numpy array
X_train = np.array(digits)

# Flatten and reshape training data
X_train_flat = X_train.reshape(len(X_train), -1)

# Define labels
y_train = list(range(10))  # [0,1,2,3,4,5,6,7,8,9]

# Initialize and train MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(10,), activation='relu', max_iter=2000, random_state=42)
clf.fit(X_train_flat, y_train)

# Define test data (same as training for simplicity here)
X_test = np.array(digits)

# Flatten and reshape test data
X_test_flat = X_test.reshape(len(X_test), -1)

# Predict using trained model
predictions = clf.predict(X_test_flat)

# Output
for i, pred in enumerate(predictions):
    print(f"Test {i}: Predicted number is {pred}")
 explain this code line by line
```

Of course! I'll explain your code **line by line**, clearly:

```python
import numpy as np
from sklearn.neural_network import MLPClassifier
```

- **Import libraries**:
  - `numpy` (as `np`) for array operations.
  - `MLPClassifier` from `sklearn.neural_network`, which is a **Multi-Layer Perceptron** (a type of artificial neural network) for classification tasks.

```python
# Define training data for digits 0–9
digits = [
    [[1, 1, 1],
     [1, 0, 1],
     [1, 0, 1],
     [1, 0, 1],
     [1, 1, 1]],  # 0
    ...
]
```

- **Create input data**:
  - `digits` is a list of 10 patterns, each representing the numbers 0–9.
  - Each pattern is a **5×3 grid** (`1` for "on" pixels and `0` for "off" pixels).
  - Example: The first pattern looks like a "0".

```python
# Convert to numpy array
X_train = np.array(digits)
```

- **Convert the `digits` list into a NumPy array**.
  - This makes it easier to process the data numerically.

```python
# Flatten and reshape training data
X_train_flat = X_train.reshape(len(X_train), -1)
```

- **Flatten each 5×3 grid into a 1D array**.
  - `reshape(len(X_train), -1)` means:
    - There are `len(X_train)` = 10 samples.
    - `-1` automatically calculates the number of features (5×3=15 features for each digit).
  - Result: `X_train_flat` becomes a **10×15** matrix (10 digits, each described by 15 numbers).

```python
# Define labels
y_train = list(range(10))  # [0,1,2,3,4,5,6,7,8,9]
```

- **Create labels**:
  - Each pattern gets a label (0 to 9).
  - So digit 0 is labeled as 0, digit 1 as 1, and so on.

```python
# Initialize and train MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(10,), activation='relu', max_iter=2000, random_state=42)
```

- **Initialize the neural network**:
  - **hidden_layer_sizes=(10,)** → 1 hidden layer with 10 neurons.
  - **activation='relu'** → Using **ReLU** (Rectified Linear Unit) activation function.
  - **max_iter=2000** → Train up to 2000 iterations (epochs) for better convergence.
  - **random_state=42** → Fix randomness for reproducibility (you get the same results every time you run).

```python
clf.fit(X_train_flat, y_train)
```

- **Train the neural network**:
  - Inputs: `X_train_flat` (features)
  - Outputs: `y_train` (labels)
- The model learns patterns that map each 5×3 grid to the correct digit.

```python
# Define test data (same as training for simplicity here)
X_test = np.array(digits)
```

- **Prepare test data**:
  - Here, for simplicity, you use the same `digits` for testing.

```python
# Flatten and reshape test data
X_test_flat = X_test.reshape(len(X_test), -1)
```

- **Flatten the test data**:

- Like before, reshape into a **10×15** array.

```python
# Predict using trained model
predictions = clf.predict(X_test_flat)
```

- **Make predictions**:
  - The model predicts the digit for each test sample.

```python
# Output
for i, pred in enumerate(predictions):
    print(f"Test {i}: Predicted number is {pred}")
```

- **Print the results**:
  - For each test sample, display the predicted number.

In short:

- You create **simple digit images (grids)**.
- Flatten the grids into **feature vectors**.
- Train a **small neural network** to classify them.
- Then **test** it (on the same samples) and **print predictions**.

Would you also like me to show a **diagram of the flow** or **visualize the digits** for better understanding? 🚀✨
It will make it even easier!

{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 12,
   "id": "29618668-81f5-4dae-8c97-fe2d2210ab9b",
   "metadata": {
    "tags": []
   },
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [

```
      "[[0.02291749]\n",
      " [0.94316892]\n",
      " [0.94423887]\n",
      " [0.07088367]]\n"
     ]
    }
   ],
   "source": [
    "import numpy as np\n",
    "\n",
    "# Set seed for reproducibility\n",
    "\n",
    "\n",
    "# Define sigmoid and its derivative\n",
    "def sigmoid(x):\n",
    "    return 1 / (1 + np.exp(-x))\n",
    "\n",
    "def sigmoid_derivative(x):\n",
    "    return x * (1 - x)\n",
    "\n",
    "# Training function\n",
    "def train_neural_network(X, y, learning_rate, epochs):\n",
    "    input_neurons = X.shape[1]\n",
    "    hidden_neurons = 4\n",
    "    output_neurons = y.shape[1]\n",
    "    \n",
    "    # Initialize weights with fixed random seed\n",
    "    hidden_weights = np.random.uniform(size=(input_neurons, hidden_neurons))\n",
    "    hidden_bias = np.random.uniform(size=(1, hidden_neurons))\n",
    "    output_weights = np.random.uniform(size=(hidden_neurons, output_neurons))\n",
    "    output_bias = np.random.uniform(size=(1, output_neurons))\n",
    "    \n",
    "    for i in range(epochs):\n",
    "        # Forward propagation\n",
    "        hidden_layer_activation = np.dot(X, hidden_weights) + hidden_bias\n",
    "        hidden_layer_output = sigmoid(hidden_layer_activation)\n",
    "\n",
    "        output_layer_activation = np.dot(hidden_layer_output, output_weights) + output_bias\n",
    "        predicted_output = sigmoid(output_layer_activation)\n",
    "\n",
    "        # Backpropagation\n",
    "        error = y - predicted_output\n",
    "        d_predicted_output = error * sigmoid_derivative(predicted_output)\n",
    "\n",
    "        error_hidden_layer = d_predicted_output.dot(output_weights.T)\n",
    "        d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)\n",
    "\n",
    "        # Update weights and biases\n",
    "        output_weights += hidden_layer_output.T.dot(d_predicted_output) * learning_rate\n",
    "        output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate\n",
    "        hidden_weights += X.T.dot(d_hidden_layer) * learning_rate\n",
```

```
    "        hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate\n",
    "    \n",
    "    return predicted_output\n",
    "\n",
    "# Example usage\n",
    "X = np.array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]])\n",
    "y = np.array([[0], [1], [1], [0]])\n",
    "\n",
    "# Train the Neural Network\n",
    "predicted_output = train_neural_network(X, y, learning_rate=0.1, epochs=10000)\n",
    "\n",
    "# Print the output\n",
    "print(predicted_output)\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "d0786c5b-6e46-458b-a88e-68504debcf13",
   "metadata": {},
   "outputs": [],
   "source": []
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "f16e4337-5d35-481e-afd9-51df4fae035e",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.11.5"
  }
 },
```

"nbformat": 4,
"nbformat_minor": 5
} ....explain this code line by line

Of course! Let's go **line-by-line** and explain your second code block carefully:

```python
import numpy as np
```

- **Import the numpy library** as `np`.
- Numpy helps with handling arrays and performing matrix operations easily.

```python
# Set seed for reproducibility
```

- **Comment**: A suggestion to fix random numbers for reproducibility (but **actual seed setting is missing**).
- Usually you would have something like `np.random.seed(42)` here.

```python
# Define sigmoid and its derivative
```

- **Comment**: Now you are defining the activation function (**sigmoid**) and its derivative.

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

- **Define the sigmoid activation function**.
- It "squashes" any real number to the range (0, 1).
- It is used for introducing non-linearity and for outputting probabilities.

```python
def sigmoid_derivative(x):
    return x * (1 - x)
```

- **Derivative of sigmoid function**.
- Useful for **backpropagation** (to update weights).

```python
# Training function
def train_neural_network(X, y, learning_rate, epochs):
```

- Define a function `train_neural_network` which will **train a simple neural network**.
- Inputs:
  - `X`: Training input data.
  - `y`: Target labels.
  - `learning_rate`: Step size for weight updates.
  - `epochs`: Number of times to repeat training.

```python
    input_neurons = X.shape[1]
    hidden_neurons = 4
    output_neurons = y.shape[1]
```

- **Set number of neurons**:
  - `input_neurons`: Based on features in `X` (here 3 inputs).
  - `hidden_neurons`: 4 hidden neurons (hardcoded).
  - `output_neurons`: Output neurons, based on label size (1 output here).

```python
    hidden_weights = np.random.uniform(size=(input_neurons, hidden_neurons))
    hidden_bias = np.random.uniform(size=(1, hidden_neurons))
    output_weights = np.random.uniform(size=(hidden_neurons, output_neurons))
    output_bias = np.random.uniform(size=(1, output_neurons))
```

- **Randomly initialize weights and biases** for:
  - **Hidden layer** and **output layer**.
- They are initialized with random values between 0 and 1.

```python
    for i in range(epochs):
```

- **Loop over number of epochs** to train the model many times.

```python
        hidden_layer_activation = np.dot(X, hidden_weights) + hidden_bias
```

- **Input to hidden layer** = (Input * Weights) + Bias.

```python
hidden_layer_output = sigmoid(hidden_layer_activation)
```

- **Apply sigmoid activation** on hidden layer.

```python
output_layer_activation = np.dot(hidden_layer_output, output_weights) + output_bias
```

- **Input to output layer** = (Hidden output * Weights) + Bias.

```python
predicted_output = sigmoid(output_layer_activation)
```

- **Apply sigmoid activation** to get **final predicted output**.

```python
# Backpropagation
error = y - predicted_output
```

- **Calculate error** between true label and predicted output.

```python
d_predicted_output = error * sigmoid_derivative(predicted_output)
```

- **Gradient at output layer** (how much to change the output layer weights).

```python
error_hidden_layer = d_predicted_output.dot(output_weights.T)
```

- **Error propagated back to hidden layer**.

```python
d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)
```

- **Gradient at hidden layer**.

```python
        output_weights += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
        output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
```

- **Update output layer weights and biases** using gradients.

```python
        hidden_weights += X.T.dot(d_hidden_layer) * learning_rate
        hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
```

- **Update hidden layer weights and biases** using gradients.

```python
    return predicted_output
```

- After training, **return the final predicted output**.

```python
# Example usage
X = np.array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
y = np.array([[0], [1], [1], [0]])
```

- Define **input samples** `X` and corresponding **labels** `y`.
- This looks like a basic **XOR problem** dataset.

```python
# Train the Neural Network
predicted_output = train_neural_network(X, y, learning_rate=0.1, epochs=10000)
```

- **Call the training function**.
- Train the neural network with:
  - learning rate = 0.1
  - epochs = 10,000

```python
# Print the output
print(predicted_output)
```

- **Print the final predicted output** after training.
- The output will be an array of values close to 0 or 1 (but not exactly due to sigmoid activation).

OpenAI

Example Final Output:

```lua
[[0.02]
 [0.94]
 [0.94]
 [0.07]]
```

- Close to expected outputs:
  - (0,1,1,0)

Summary:

You manually implemented a **simple 1-hidden-layer neural network** from scratch:

- Used **Sigmoid Activation**.
- Did **Forward Propagation**.
- Did **Backpropagation** to update weights and biases.
- No external library like `sklearn` used for training.
- Simple and excellent for understanding **how neural networks learn internally**.

Would you also like me to draw a small **diagram** of how the data flows between input, hidden, and output layers for better clarity? 🎯 (It'll be simple and super helpful!)
Just let me know! 🚀

{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 2,
   "id": "c45b559e",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "Input:\n",
      "[[0 0]\n",
      " [0 1]\n",
      " [1 0]\n",
      " [1 1]]\n",
      "Output:\n",
      "[[0.50001248]\n",

```
      " [0.49999867]\n",
      " [0.50000133]\n",
      " [0.49998752]]\n",
      "Expected Output:\n",
      "[[0]\n",
      " [1]\n",
      " [1]\n",
      " [0]]\n"
     ]
    }
   ],
   "source": [
    "import numpy as np\n",
    "\n",
    "# Define the parameters of the network\n",
    "input_neuron = 2   # Number of input neurons\n",
    "hidden_neuron = 4   # Number of hidden neurons\n",
    "output_neuron = 1   # Number of output neurons\n",
    "learning_rate = 0.1\n",
    "epochs = 10000\n",
    "\n",
    "# Define the training data\n",
    "X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])\n",
    "Y = np.array([[0], [1], [1], [0]])\n",
    "\n",
    "# Weights from input to hidden layer\n",
    "W1 = np.random.uniform(size=(input_neuron, hidden_neuron))\n",
    "# Weights from hidden to output layer\n",
    "W2 = np.random.uniform(size=(hidden_neuron, output_neuron))\n",
    "\n",
    "# Bias for hidden layer\n",
    "b1 = np.random.uniform(size=(1, hidden_neuron))\n",
    "# Bias for output layer\n",
    "b2 = np.random.uniform(size=(1, output_neuron))\n",
    "\n",
    "# Define the sigmoid activation function and its derivative\n",
    "def sigmoid(x):\n",
    "    return 1 / (1 + np.exp(-x))\n",
    "\n",
    "def sigmoid_derivative(x):\n",
    "    return x * (1 - x)\n",
    "\n",
    "# Train the network using backpropagation\n",
    "for i in range(epochs):\n",
    "    # Forward pass\n",
    "    hidden_layer_input = np.dot(X, W1) + b1\n",
    "    hidden_layer_output = sigmoid(hidden_layer_input)\n",
    "    output_layer_input = np.dot(hidden_layer_output, W2) + b2\n",
    "    output_layer_output = sigmoid(output_layer_input)\n",
    "\n",
    "    # Backward pass\n",
```

```
    "    output_error = Y - output_layer_output\n",
    "    output_delta = output_error * sigmoid_derivative(output_layer_output)\n",
    "\n",
    "    hidden_error = output_delta.dot(W2.T)\n",
    "    hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)\n",
    "\n",
    "    # Update weights and biases\n",
    "    W2 += np.dot(hidden_layer_output.T, output_delta) * learning_rate\n",
    "    b2 += np.sum(output_delta, axis=0, keepdims=True) * learning_rate\n",
    "    W1 += np.dot(X.T, hidden_delta) * learning_rate\n",
    "    b1 += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate\n",
    "\n",
    "# Test the network with some example inputs\n",
    "x_test = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])\n",
    "y_test = np.array([[0], [1], [1], [0]])\n",
    "\n",
    "hidden_layer_input = np.dot(x_test, W1) + b1\n",
    "hidden_layer_output = sigmoid(hidden_layer_input)\n",
    "output_layer_input = np.dot(hidden_layer_output, W2) + b2\n",
    "output_layer_output = sigmoid(output_layer_input)\n",
    "\n",
    "print(\"Input:\")\n",
    "print(x_test)\n",
    "print(\"Output:\")\n",
    "print(output_layer_output)\n",
    "print(\"Expected Output:\")\n",
    "print(y_test)\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "9e717fe5",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
```

```
  "name": "python",
  "nbconvert_exporter": "python",
  "pygments_lexer": "ipython3",
  "version": "3.11.5"
 }
},
"nbformat": 4,
"nbformat_minor": 5
}                    ..................explain this code line by line
```

Of course! Let's **explain your second code cell line-by-line** clearly:

Import Libraries

```python
import numpy as np
```

- Import the **NumPy** library, which helps in working with arrays, matrices, and numerical operations.

Define Network Parameters

```python
input_neuron = 2
hidden_neuron = 4
output_neuron = 1
learning_rate = 0.1
epochs = 10000
```

- **`input_neuron = 2`**: Two inputs (because XOR input has 2 values, e.g., [0,1]).
- **`hidden_neuron = 4`**: You are using **4 neurons** in the hidden layer (a choice; could be more or fewer).
- **`output_neuron = 1`**: Only **one output** (either 0 or 1 for XOR).
- **`learning_rate = 0.1`**: How fast the network learns (small changes in weight updates).
- **`epochs = 10000`**: How many **iterations** (training cycles) to train.

Define Training Data (XOR Problem)

```python
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
Y = np.array([[0], [1], [1], [0]])
```

- **`X`**: Inputs - standard XOR truth table inputs.
- **`Y`**: Expected Outputs for XOR:

- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0

Initialize Weights and Biases

```python
W1 = np.random.uniform(size=(input_neuron, hidden_neuron))
W2 = np.random.uniform(size=(hidden_neuron, output_neuron))
b1 = np.random.uniform(size=(1, hidden_neuron))
b2 = np.random.uniform(size=(1, output_neuron))
```

- `W1`: Random weights from **input layer → hidden layer**.
- `W2`: Random weights from **hidden layer → output layer**.
- `b1`: Bias for hidden layer.
- `b2`: Bias for output layer.

(Weights and biases are initialized randomly)

Define Activation Functions

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

- The **sigmoid function** "squashes" values between **0 and 1**.

```python
def sigmoid_derivative(x):
    return x * (1 - x)
```

- The **derivative** of the sigmoid function, used in **backpropagation** to adjust weights.

Training the Network (Forward + Backward Pass)

```python
for i in range(epochs):
```

- Loop through **10000 times** to train the network.

Forward Propagation

```python
hidden_layer_input = np.dot(X, W1) + b1
hidden_layer_output = sigmoid(hidden_layer_input)
output_layer_input = np.dot(hidden_layer_output, W2) + b2
output_layer_output = sigmoid(output_layer_input)
```

- **hidden_layer_input**: Multiply inputs and weights, then add bias.
- **hidden_layer_output**: Apply **sigmoid** activation to the hidden layer.
- **output_layer_input**: Multiply hidden outputs and second set of weights, add output bias.
- **output_layer_output**: Apply **sigmoid** again for final prediction.

Backward Propagation

```python
output_error = Y - output_layer_output
output_delta = output_error * sigmoid_derivative(output_layer_output)
```

- Calculate the **error**: (Expected output - Predicted output).
- Calculate **output delta**: How much change is needed at output.

```python
hidden_error = output_delta.dot(W2.T)
hidden_delta = hidden_error * sigmoid_derivative(hidden_layer_output)
```

- Find how much error is at the hidden layer (by back-propagating).
- Calculate **hidden delta**: How much change is needed at hidden.

Update Weights and Biases

```python
W2 += np.dot(hidden_layer_output.T, output_delta) * learning_rate
b2 += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
W1 += np.dot(X.T, hidden_delta) * learning_rate
b1 += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

- Adjust `W2`, `b2`, `W1`, `b1` according to calculated deltas multiplied by learning rate.
- **This is how the model "learns"** by minimizing the error!

Testing the Network After Training

```python
x_test = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

```
y_test = np.array([[0], [1], [1], [0]])
```

- Inputs and expected outputs for testing (same as training data).

Forward Pass for Testing

```python
hidden_layer_input = np.dot(x_test, W1) + b1
hidden_layer_output = sigmoid(hidden_layer_input)
output_layer_input = np.dot(hidden_layer_output, W2) + b2
output_layer_output = sigmoid(output_layer_input)
```

- Same forward propagation but now for testing data.

Print Results

```python
print("Input:")
print(x_test)
print("Output:")
print(output_layer_output)
print("Expected Output:")
print(y_test)
```

- Prints:
  - Test Inputs.
  - Predicted Outputs.
  - Actual Expected Outputs.

In Short:

✅ This code trains a **simple 2-4-1 Neural Network** to learn the **XOR** function using **sigmoid activation** and **backpropagation**.

✅ Initially, outputs will not match perfectly, but after 10,000 epochs, they should be close to **0** or **1**.

Would you also like me to draw a **diagram** showing the architecture (input → hidden → output) to visualize it better? 🚀 ✨
(Just say "yes" if you want!)

import numpy as np

class ART1:

```
    def __init__(self, input_size, num_categories, vigilance):
        self.input_size = input_size
        self.num_categories = num_categories
        self.vigilance = vigilance

        # Weight initialization (set to 1s, classical ART1)
        self.weights = np.ones((num_categories, input_size * 2))  # for complement coding

    def complement_code(self, input_vector):
        # Add complement to input vector (1s and 0s assumed)
        return np.concatenate([input_vector, 1 - input_vector])

    def match_category(self, input_vector):
        # Calculate match (activation) as sum of minimums
        return np.array([np.sum(np.minimum(input_vector, w)) for w in self.weights])

    def vigilance_test(self, input_vector, category_index):
        # Match quality (fuzzy match) test
        numerator = np.sum(np.minimum(input_vector, self.weights[category_index]))
        denominator = np.sum(input_vector)
        return numerator / denominator >= self.vigilance

    def train(self, input_patterns):
        for pattern in input_patterns:
            input_vector = self.complement_code(pattern)

            while True:
                match_scores = self.match_category(input_vector)

                # Choose best matching category
                chosen = np.argmax(match_scores)

                # If it passes vigilance, learn
                if self.vigilance_test(input_vector, chosen):
                    self.weights[chosen] = np.minimum(self.weights[chosen], input_vector)
                    break
                else:
                    # Disable this category for current input
                    self.weights[chosen] = np.zeros_like(self.weights[chosen])

    def predict(self, input_vector):
        input_vector = self.complement_code(input_vector)
        match_scores = self.match_category(input_vector)
        return np.argmax(match_scores)

# --- Example usage ---
inputs = np.array([
    [1, 0, 0, 0],
    [0, 1, 0, 1],
    [1, 1, 0, 0],
    [0, 0, 1, 1]
```

```
])

art = ART1(input_size=4, num_categories=4, vigilance=0.7)
art.train(inputs)

print("\nPredictions:")
for input_vector in inputs:
    category = art.predict(input_vector)
    print(f"Input: {input_vector}, Predicted Category: {category}")            ........explain this code line
by line
```

Of course! Let's go **line-by-line** through your ART1 neural network code:

```python
import numpy as np
```

- Imports the **NumPy** library for array operations (needed for fast mathematical calculations).

```python
class ART1:
```

- Defines a **class** called `ART1`, which implements the **Adaptive Resonance Theory 1** algorithm (unsupervised clustering for binary input).

```python
    def __init__(self, input_size, num_categories, vigilance):
```

- **Constructor** (`__init__` function) that runs when you create an ART1 object.
- It receives:
    - `input_size`: Number of input features.
    - `num_categories`: Maximum number of clusters/categories.
    - `vigilance`: A threshold to determine how strict matching should be (between 0 and 1).

```python
        self.input_size = input_size
        self.num_categories = num_categories
        self.vigilance = vigilance
```

- Saves the input parameters inside the object for later use.

```python
        self.weights = np.ones((num_categories, input_size * 2))  # for complement coding
```

- Initializes the **weights** as a 2D array full of ones.
- Size = (number of categories) × (input size × 2), because **complement coding** is used (more on this soon).

```python
    def complement_code(self, input_vector):
```

- Defines a function to create **complement coded** input vectors.
- Complement coding is a technique that helps ART1 preserve information better by adding `` `(1 - input)` `` for each input feature.

```python
        return np.concatenate([input_vector, 1 - input_vector])
```

- Takes the input vector (like `` `[1, 0]` ``) and adds its complement (`` `[0, 1]` ``), making it `` `[1, 0, 0, 1]` ``.

```python
    def match_category(self, input_vector):
```

- Defines a function to find **how well** the input matches **each category**.

```python
        return np.array([np.sum(np.minimum(input_vector, w)) for w in self.weights])
```

- For each weight vector (category):
  - Calculates the **element-wise minimum** between the input and the weight.
  - Sums these minimum values.
  - The higher the sum, the better the match.

```python
    def vigilance_test(self, input_vector, category_index):
```

- Defines a function to **check if** a chosen category **passes** the vigilance test.

```python
        numerator = np.sum(np.minimum(input_vector, self.weights[category_index]))
        denominator = np.sum(input_vector)
```

- Calculates:
  - **Numerator**: How much of the input is actually matched by the category.
  - **Denominator**: Total active features in input.

```python
        return numerator / denominator >= self.vigilance
```

- If the match is **good enough** (ratio ≥ vigilance), returns **True**.
- Otherwise, returns **False** (category fails vigilance check).

```python
    def train(self, input_patterns):
```

- Defines a **training function** that takes multiple input patterns and tries to cluster them.

```python
        for pattern in input_patterns:
```

- Loop through each input pattern.

```python
            input_vector = self.complement_code(pattern)
```

- Apply **complement coding** to the pattern.

```python
            while True:
```

- Keeps trying until the input is assigned to a category.

```python
                match_scores = self.match_category(input_vector)
```

- Calculate how well the input matches every category.

```python
                chosen = np.argmax(match_scores)
```

- **Choose** the category with the **highest** match score.

```python
                if self.vigilance_test(input_vector, chosen):
```

- **Check** if the chosen category **passes** the vigilance test.

```python
                    self.weights[chosen] = np.minimum(self.weights[chosen], input_vector)
```

- **Learn**: Update the category's weights by taking the **element-wise minimum** between the current weight and the input.
- Makes the category more specialized to the pattern.

```python
                    break
```

- Successfully assigned; exit the loop.

```python
                else:
```

- If vigilance test **fails**:

```python
                    self.weights[chosen] = np.zeros_like(self.weights[chosen])
```

- **Disable** the category by setting all its weights to 0 (so it can't be selected again for this pattern).

```python
    def predict(self, input_vector):
```

- Defines a **prediction function**.

```python
        input_vector = self.complement_code(input_vector)
```

- Apply **complement coding** to the input.

```python
        match_scores = self.match_category(input_vector)
```

- Calculate match scores with all categories.

```python
        return np.argmax(match_scores)
```

- Return the index (category number) with the highest match.

Example usage

```python
inputs = np.array([
    [1, 0, 0, 0],
    [0, 1, 0, 1],
    [1, 1, 0, 0],
    [0, 0, 1, 1]
])
```

- Defines 4 binary input patterns.

```python
art = ART1(input_size=4, num_categories=4, vigilance=0.7)
```

- Create an ART1 network:
  - 4 input features
  - 4 maximum categories
  - Vigilance threshold of 0.7

```python
art.train(inputs)
```

- Train the ART1 model on the given inputs.

```python
print("\nPredictions:")
for input_vector in inputs:
    category = art.predict(input_vector)
    print(f"Input: {input_vector}, Predicted Category: {category}")
```

- Loop over all input patterns again.
- Predict which category each pattern belongs to.
- Print the input and its predicted category.

🔥 In short:

- **ART1** clusters binary input patterns into categories.
- It **learns adaptively** and ensures new inputs match well enough (**vigilance**), otherwise creates new categories.
- **Complement coding** helps in stability and discrimination.
- **Weights are reduced** (element-wise minimum) to maintain the best matching features only.

Would you like me to also create a **simple visual flowchart** or **example run** for easier understanding? 🚀
(If yes, just say!)

```
import numpy as np

class HopfieldNetwork:
    def __init__(self):
        self.weights = None

    def train(self, patterns):
        num_neurons = patterns.shape[1]
        self.weights = np.zeros((num_neurons, num_neurons))

        for p in patterns:
            self.weights += np.outer(p, p)

        np.fill_diagonal(self.weights, 0)  # Remove self-connections
        self.weights /= patterns.shape[0]  # Normalize

    def activation(self, x):
        return np.where(x >= 0, 1, -1)

    def recall(self, pattern, steps=5):
        output = pattern.copy()
        for _ in range(steps):
```

```
        output = self.activation(np.dot(self.weights, output))
    return output

# --- Example usage ---
if __name__ == "__main__":
    # Define binary patterns to store (-1 and 1)
    patterns = np.array([
        [1, 1, -1, -1],
        [-1, -1, 1, 1],
        [1, -1, 1, -1],
        [-1, 1, -1, 1]
    ])

    hopfield_net = HopfieldNetwork()
    hopfield_net.train(patterns)                                        print("Testing recall
on original patterns:")
    for i, p in enumerate(patterns):
        output = hopfield_net.recall(p)
        print(f"Pattern {i+1}:")
        print("Input:  ", p)
        print("Output: ", output)
        print()

    # Test with noisy input
    noisy_input = np.array([1, 1, -1, 1])  # Flip one bit of first pattern
    output = hopfield_net.recall(noisy_input)

    print("Testing with noisy input:")
    print("Noisy input: ", noisy_input)
    print("Recovered:   ", output)
........explain this code line by line
```

Of course! Let’s go **line-by-line** and explain your Hopfield network code clearly:

```python
import numpy as np
```

- Import the **NumPy** library to handle matrices and arrays efficiently.

```python
class HopfieldNetwork:
```

- Define a **class** named `HopfieldNetwork` that models a **Hopfield neural network** (an associative memory system).

```python
    def __init__(self):
        self.weights = None
```

- **Constructor** (`__init__` method) initializes the `weights` attribute to `None`.
- `weights` will later store the **connection strengths** between neurons.

```python
    def train(self, patterns):
```

- Define a **train** method to **store** (memorize) patterns into the network.

```python
        num_neurons = patterns.shape[1]
```

- Find out how many **neurons** (features) each pattern has.
- `patterns` is a 2D array: rows = patterns, columns = neurons.

```python
        self.weights = np.zeros((num_neurons, num_neurons))
```

- Create a **weight matrix** initialized to all zeros.
- Size = (number of neurons × number of neurons).

```python
        for p in patterns:
            self.weights += np.outer(p, p)
```

- Loop over each pattern `p`.
- Update the weights by **adding** the outer product of `p` with itself.
  - The outer product builds a full matrix based on how each neuron should influence others.

```python
        np.fill_diagonal(self.weights, 0)  # Remove self-connections
```

- Set the **diagonal** of the weight matrix to 0.
- No neuron should influence itself.

```python
        self.weights /= patterns.shape[0]  # Normalize
```

- **Normalize** the weights by dividing by the number of patterns.
- This prevents the weights from becoming too large as you store more patterns.

```python
    def activation(self, x):
```

- Define an **activation function**.

```python
        return np.where(x >= 0, 1, -1)
```

- For each neuron:
  - If the input is **≥ 0**, output **1**.
  - If the input is **< 0**, output **-1**.
- This ensures binary-like output values (-1 or 1).

```python
    def recall(self, pattern, steps=5):
```

- Define a **recall** method:
  - It tries to recover (retrieve) a stored pattern starting from an input.
  - It will update the pattern for a fixed number of steps.

```python
        output = pattern.copy()
```

- Copy the input pattern to `` `output` `` (so as not to modify the original input).

```python
        for _ in range(steps):
            output = self.activation(np.dot(self.weights, output))
```

- For each step:
  - Multiply the weights with the current output (`` `dot` `` product).

   - Apply the activation function to decide new neuron states.

---

python

```python
        return output
```

- After updating the neurons for `steps` times, return the final recalled pattern.

Now the example usage:

python

```python
if __name__ == "__main__":
```

- Run this part **only if** the file is executed directly (not imported as a module).

---

python

```python
    patterns = np.array([
        [1, 1, -1, -1],
        [-1, -1, 1, 1],
        [1, -1, 1, -1],
        [-1, 1, -1, 1]
    ])
```

- Define 4 **training patterns**.
- Each pattern is a vector of -1s and 1s.

---

python

```python
    hopfield_net = HopfieldNetwork()
    hopfield_net.train(patterns)
```

- Create an instance of the Hopfield network.
- Train it on the given patterns (store them into memory).

---

python

```python
    print("Testing recall on original patterns:")
    for i, p in enumerate(patterns):
        output = hopfield_net.recall(p)
        print(f"Pattern {i+1}:")
        print("Input:  ", p)
        print("Output: ", output)
        print()
```

- Loop over each stored pattern:
   - Test if the network can recall (retrieve) the pattern correctly.

- Print input and recalled output.

```python
noisy_input = np.array([1, 1, -1, 1])  # Flip one bit of first pattern
```

- Create a **noisy version** of the first pattern by flipping one bit.

```python
output = hopfield_net.recall(noisy_input)
```

- Try to **recall** the original pattern from the noisy input.

```python
print("Testing with noisy input:")
print("Noisy input: ", noisy_input)
print("Recovered:   ", output)
```

- Print the noisy input and the **recovered** pattern.
- **Goal:** See if the Hopfield network can correct the noise and retrieve the correct pattern.

🔥 In short:

- Hopfield networks can **store patterns** and **retrieve them** even if the input is noisy.
- It uses a **weight matrix** learned by **outer products** of the patterns.
- Updates the neurons in parallel for a few steps.
- Uses simple binary threshold activation (1 or -1).

Would you like me to also show you a **simple small visual animation** of how noisy recall works step-by-step? 🚀 (It will make it even easier!)
Just let me know! 📈

import tensorflow as tf
import numpy as np
import cv2

# Load the pre-trained model
model = tf.saved_model.load('D:\\faster_rcnn_resnet50_v1_640x640_coco17_tpu-8\\saved_model')

# Load the label map
label_map = {1: 'person', 3: 'car', 18: 'dog'} # Replace with your label map

```
# Load and preprocess the input image
image = cv2.imread("C:/Users/athar/OneDrive/Desktop/Practical/ANNPrac/image.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert to uint8 (Fix for TypeError)
image = tf.convert_to_tensor(image, dtype=tf.uint8)

# Expand dimensions for batch processing
image = tf.expand_dims(image, axis=0)

# Run object detection
detections = model(image)

# Extract results
boxes = detections['detection_boxes'][0].numpy()
scores = detections['detection_scores'][0].numpy()
classes = detections['detection_classes'][0].numpy().astype(int)

# Set confidence threshold
threshold = 0.5
filtered_boxes = boxes[scores >= threshold]
filtered_classes = classes[scores >= threshold]

# Convert tensor back to numpy for OpenCV drawing
image_np = image.numpy()[0]

# Draw bounding boxes and labels
for box, cls in zip(filtered_boxes, filtered_classes):
    ymin, xmin, ymax, xmax = box
    h, w, _ = image_np.shape
    ymin, xmin, ymax, xmax = int(ymin * h), int(xmin * w), int(ymax * h), int(xmax * w)

    label = label_map.get(cls, 'Unknown')
    cv2.rectangle(image_np, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)
    cv2.putText(image_np, label, (xmin, ymin - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)

# Display the image with detections
cv2.imshow('Object Detection', image_np)
cv2.waitKey(0)
cv2.destroyAllWindows()                    .........expalin this code line by line
```

Of course! Let's go **line by line** and explain your TensorFlow object detection code clearly:

```python
import tensorflow as tf
import numpy as np
import cv2
```

- Import necessary libraries:
  - `tensorflow` for running the detection model,
  - `numpy` for numerical operations,
  - `cv2` (OpenCV) for image processing and visualization.

```python
# Load the pre-trained model
model = tf.saved_model.load('D:\\faster_rcnn_resnet50_v1_640x640_coco17_tpu-8\\saved_model')
```

- Load a **pre-trained Faster R-CNN ResNet50** model from a saved path.
- `tf.saved_model.load` loads the entire trained model into memory.

```python
# Load the label map
label_map = {1: 'person', 3: 'car', 18: 'dog'}  # Replace with your label map
```

- Create a **label map** that maps **class IDs** (like 1, 3, 18) to **human-readable labels** (like 'person', 'car', 'dog').
- Normally you would load the full COCO label map, but here you manually defined a small one.

```python
# Load and preprocess the input image
image = cv2.imread("C:/Users/athar/OneDrive/Desktop/Practical/ANNPrac/image.jpg")
```

- Load an image from your computer using OpenCV.
- Image is loaded in **BGR** color format by default.

```python
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

- Convert the image from **BGR** to **RGB** because TensorFlow models expect RGB input.

```python
# Convert to uint8 (Fix for TypeError)
image = tf.convert_to_tensor(image, dtype=tf.uint8)
```

- Convert the image from NumPy array to TensorFlow tensor of type **unsigned 8-bit integer** (`uint8`).
- Required format for TensorFlow models.

```python
# Expand dimensions for batch processing
image = tf.expand_dims(image, axis=0)
```

- Add an extra dimension to the image to represent a **batch** of size 1.
- Deep learning models expect inputs like `` `[batch_size, height, width, channels]` ``.

```python
# Run object detection
detections = model(image)
```

- Pass the image through the model to get **detections**.
- `` `detections` `` is a dictionary containing outputs like:
  - `` `detection_boxes` ``
  - `` `detection_scores` ``
  - `` `detection_classes` ``
  - etc.

```python
# Extract results
boxes = detections['detection_boxes'][0].numpy()
scores = detections['detection_scores'][0].numpy()
classes = detections['detection_classes'][0].numpy().astype(int)
```

- Extract specific parts from the detections:
  - `` `boxes` ``: bounding box coordinates for each detected object.
  - `` `scores` ``: confidence scores (how sure the model is about the detection).
  - `` `classes` ``: class IDs for each detected object.
- Use `` `[0]` `` because there's only **one image** in the batch.
- Convert tensors to NumPy arrays for further processing.

```python
# Set confidence threshold
threshold = 0.5
```

- Set a **threshold**: only keep detections with **confidence score >= 50%**.

```python
filtered_boxes = boxes[scores >= threshold]
filtered_classes = classes[scores >= threshold]
```

- Filter the bounding boxes and classes based on the threshold.
- Only keep the high-confidence detections.

```python
# Convert tensor back to numpy for OpenCV drawing
image_np = image.numpy()[0]
```

- Convert the tensor image back to a NumPy array for **OpenCV drawing**.
- Again `[0]` because batch size = 1.

```python
# Draw bounding boxes and labels
for box, cls in zip(filtered_boxes, filtered_classes):
```

- Loop through each filtered detection (bounding box and class).

```python
    ymin, xmin, ymax, xmax = box
    h, w, _ = image_np.shape
    ymin, xmin, ymax, xmax = int(ymin * h), int(xmin * w), int(ymax * h), int(xmax * w)
```

- Each bounding box is normalized (values between 0 and 1).
- Multiply by image height `h` and width `w` to get **pixel coordinates**.

```python
    label = label_map.get(cls, 'Unknown')
```

- Find the **label name** for the detected class.
- If class ID not found in `label_map`, default to `'Unknown'`.

```python
    cv2.rectangle(image_np, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)
```

- Draw a **green bounding box** around the detected object.

```python
    cv2.putText(image_np, label, (xmin, ymin - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
```

- Draw the **label text** (e.g., "dog", "person") above the bounding box.

```python
# Display the image with detections
cv2.imshow('Object Detection', image_np)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- **Display** the resulting image with drawn boxes and labels in a window.
- `cv2.waitKey(0)` waits indefinitely for any key press.
- `cv2.destroyAllWindows()` closes the window.

🔥 Summary:

- Load model ✔️
- Load and prepare the image ✔️
- Detect objects ✔️
- Filter based on confidence ✔️
- Draw bounding boxes and labels ✔️
- Display the final annotated image ✔️

Would you also like me to explain **how you could improve it** (like dynamic label loading or using TensorFlow's `object_detection` API visualization tools)? 🚀
It would make your project even more professional! 🎯

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
import os

# Load and preprocess dataset (CIFAR-10 for demonstration)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert labels to categorical format
y_train, y_test = keras.utils.to_categorical(y_train, 10), keras.utils.to_categorical(y_test, 10)

# Data Augmentation
data_augmentation = keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
```

```
    layers.RandomZoom(0.1),
])

# Build CNN Model
def build_model():
    model = models.Sequential([
        data_augmentation,
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    return model

# Compile Model
model = build_model()
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the Model
history = model.fit(x_train, y_train, epochs=20, batch_size=64, validation_data=(x_test, y_test))

# Plot Accuracy & Loss
fig, axs = plt.subplots(2, 1, figsize=(10, 8))

# Accuracy Graph
axs[0].plot(history.history['accuracy'], label='Train Accuracy')
axs[0].plot(history.history['val_accuracy'], label='Validation Accuracy')
axs[0].set_title('Model Accuracy')
axs[0].set_ylabel('Accuracy')
axs[0].set_xlabel('Epoch')
axs[0].legend()

# Loss Graph
```

axs[1].plot(history.history['loss'], label='Train Loss')
axs[1].plot(history.history['val_loss'], label='Validation Loss')
axs[1].set_title('Model Loss')
axs[1].set_ylabel('Loss')
axs[1].set_xlabel('Epoch')
axs[1].legend()

plt.tight_layout()
plt.show()

Evaluate Model
final_loss, final_acc = model.evaluate(x_test, y_test)
print(f"Final Test Accuracy: {final_acc:.4f}")
.........explain this code line by line

Of course! Let’s **explain your code line by line** so you fully understand it:

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
import os
```

- Import essential libraries:
    - `tensorflow`, `keras`, and `layers/models` for building and training the CNN.
    - `matplotlib.pyplot` for plotting graphs.
    - `numpy` for numerical operations.
    - `os` (not used in this code, but often for file paths).

```python
# Load and preprocess dataset (CIFAR-10 for demonstration)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
```

- Load the **CIFAR-10 dataset** (60,000 small color images, 10 classes like airplane, car, cat, etc.).
- It automatically splits into:
    - `x_train`, `y_train`: training data and labels
    - `x_test`, `y_test`: testing data and labels

```python
# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0
```

- Images have pixel values between 0-255.
- Normalize them to range **[0,1]** by dividing by 255 to speed up training and improve convergence.

```python
# Convert labels to categorical format
y_train, y_test = keras.utils.to_categorical(y_train, 10), keras.utils.to_categorical(y_test, 10)
```

- Labels (originally integers like 0,1,2...) are **one-hot encoded**.
  - Example: Label `3` → `[0,0,0,1,0,0,0,0,0,0]`
- Required for classification using `softmax` at the output layer.

```python
# Data Augmentation
data_augmentation = keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
])
```

- **Data augmentation** to artificially increase training data by modifying images:
  - **Random flip** horizontally,
  - **Small rotation** (10% of 360°),
  - **Small zoom**.
- Helps improve model's ability to generalize.

```python
# Build CNN Model
def build_model():
```

- Define a **function** `build_model()` to create and return the CNN.

```python
    model = models.Sequential([
```

- Start a **Sequential model** (stacking layers one after another).

```python
        data_augmentation,
```

- Apply **data augmentation** layer first (only during training).

```python
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
```

- 2D Convolution Layer:
  - 32 filters of size 3×3,
  - `ReLU` activation,
  - Input shape is (32x32 pixels, 3 color channels).

```python
        layers.BatchNormalization(),
```

- Normalize activations from the previous layer to make training **faster and more stable**.

```python
        layers.MaxPooling2D((2, 2)),
```

- Reduce image size by taking the maximum over a 2×2 area (**downsampling**).

```python
        layers.Dropout(0.25),
```

- Randomly set 25% of activations to zero during training (helps prevent **overfitting**).

```python
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),
```

- 2nd **Convolution block**:
  - More filters (64),
  - Same pattern: conv → batchnorm → pool → dropout.

```python
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),
```

- 3rd **Convolution block**:

- Even more filters (128) for capturing complex patterns.

```python
        layers.Flatten(),
```

- **Flatten** output into a 1D vector to feed into Dense layers.

```python
        layers.Dense(256, activation='relu'),
```

- Fully connected (dense) layer with 256 neurons, `ReLU` activation.

```python
        layers.BatchNormalization(),
        layers.Dropout(0.5),
```

- Normalize and apply a **stronger dropout (50%)** for further regularization before output.

```python
        layers.Dense(10, activation='softmax')
```

- Final output layer:
  - 10 neurons (for 10 classes),
  - `softmax` activation to output **probability distribution** over classes.

```python
    ])
    return model
```

- Close the model and return it.

```python
# Compile Model
model = build_model()
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- Build and **compile** the model:
  - **Adam** optimizer with learning rate 0.001,

- **Categorical crossentropy** loss (for multi-class classification),
- Monitor **accuracy** during training.

```python
# Train the Model
history = model.fit(x_train, y_train, epochs=20, batch_size=64, validation_data=(x_test, y_test))
```

- **Train** the model for **20 epochs** with a **batch size of 64**.
- Use **`x_test`**, **`y_test`** for **validation** after every epoch.
- Save training progress into **`history`**.

```python
# Plot Accuracy & Loss
fig, axs = plt.subplots(2, 1, figsize=(10, 8))
```

- Create two subplots (one for accuracy, one for loss) with size 10×8 inches.

```python
# Accuracy Graph
axs[0].plot(history.history['accuracy'], label='Train Accuracy')
axs[0].plot(history.history['val_accuracy'], label='Validation Accuracy')
axs[0].set_title('Model Accuracy')
axs[0].set_ylabel('Accuracy')
axs[0].set_xlabel('Epoch')
axs[0].legend()
```

- Plot the training and validation **accuracy** over epochs.
- Add labels, title, and legends.

```python
# Loss Graph
axs[1].plot(history.history['loss'], label='Train Loss')
axs[1].plot(history.history['val_loss'], label='Validation Loss')
axs[1].set_title('Model Loss')
axs[1].set_ylabel('Loss')
axs[1].set_xlabel('Epoch')
axs[1].legend()
```

- Plot the training and validation **loss** over epochs similarly.

```python
plt.tight_layout()
plt.show()
```

- Adjust layout and **display the graphs**.

```python
# Evaluate Model
final_loss, final_acc = model.evaluate(x_test, y_test)
print(f"Final Test Accuracy: {final_acc:.4f}")
```

- **Evaluate** the final model performance on the **test set**.
- Print the final **accuracy** (formatted to 4 decimal places).

✨ Summary:

- Load CIFAR-10 dataset ✔️
- Normalize and augment images ✔️
- Build a CNN model ✔️
- Train and validate the model ✔️
- Plot training progress ✔️
- Evaluate final performance ✔️

Would you like me to also give you a **diagram** showing the CNN structure you built? 🎯 (It would be helpful for your notes or presentation!)
Let me know! 🚀

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize

# Reshape for CNN (adding channel dimension)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# Define CNN model
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')  # Output layer for 10 classes
```

])

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

# Train model
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))

Evaluate model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}")
........explain this code line by line

Of course! Let's **explain your second code line by line**:

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

- Import libraries:
  - `` `tensorflow` `` and `` `keras` `` for building deep learning models.
  - `` `layers` `` for adding CNN layers easily.

```python
# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

- Load the **MNIST dataset**:
  - Handwritten digits (0–9).
  - 60,000 training images and 10,000 testing images.
  - Each image is 28×28 pixels in grayscale.

```python
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize
```

- **Normalize** pixel values from `` `[0, 255]` `` to `` `[0, 1]` ``.
- Helps the neural network train faster and more accurately.

```python
# Reshape for CNN (adding channel dimension)
x_train = x_train.reshape(-1, 28, 28, 1)
```

```
x_test = x_test.reshape(-1, 28, 28, 1)
```

- CNNs expect 4D input: (batch_size, height, width, **channels**).
- MNIST is grayscale, so **channels=1**.
- `-1` means **"infer batch size automatically"** (it keeps the number of samples the same).

```python
# Define CNN model
model = keras.Sequential([
```

- Start defining a **Sequential model** — simple layer-by-layer model.

```python
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
```

- 1st **Convolution Layer**:
  - 32 filters (kernels) of size 3×3,
  - Activation: **ReLU**,
  - Input size: 28×28×1.

```python
    layers.MaxPooling2D((2, 2)),
```

- **Max Pooling Layer**:
  - Downsample (reduce) the feature maps by selecting maximum value from 2×2 regions.
  - Helps in **reducing computation** and **controlling overfitting**.

```python
    layers.Conv2D(64, (3, 3), activation='relu'),
```

- 2nd **Convolution Layer**:
  - 64 filters of size 3×3,
  - `ReLU` activation.

```python
    layers.MaxPooling2D((2, 2)),
```

- Again **Max Pooling** to further reduce spatial size.

```python
    layers.Conv2D(64, (3, 3), activation='relu'),
```

- 3rd **Convolution Layer**:
  - 64 filters again,
  - No pooling after this — directly going to flatten.

```python
    layers.Flatten(),
```

- **Flatten** the 3D output (height × width × channels) into a **1D vector**.
- Ready to feed into dense (fully connected) layers.

```python
    layers.Dense(64, activation='relu'),
```

- Fully connected **Dense Layer**:
  - 64 neurons,
  - `ReLU` activation.

```python
    layers.Dense(10, activation='softmax')  # Output layer for 10 classes
```

- **Output Layer**:
  - 10 neurons (one for each digit 0–9),
  - `softmax` to output a **probability distribution** over the 10 classes.

```python
# Compile model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

- **Compile** the model:
  - Optimizer: **Adam** (efficient and adaptive),
  - Loss: **Sparse categorical crossentropy** (because labels are integers, not one-hot encoded),
  - Metric: **Accuracy** (to monitor model performance).

```python
# Train model
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```

- Train the model:
    - For **5 epochs** (5 passes over the training data),
    - Also evaluate on **test data** after each epoch (validation).

```python
# Evaluate model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}")
```

- After training, **evaluate** final performance on the test set.
- Print the **test accuracy** rounded to 4 decimal places.

✨ Summary:

- Load MNIST handwritten digits ✔️
- Normalize and reshape ✔️
- Build a small CNN with 3 Conv layers ✔️
- Train for 5 epochs ✔️
- Evaluate and report accuracy ✔️

Would you also like me to draw the architecture diagram (block diagram) of this CNN? 📈🔵 It would be useful for understanding how the data flows!
Let me know! 🚀

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Load the MNIST dataset, a collection of handwritten digits
mnist = keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
Normalize pixel values to be in the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0
# Build the neural network model
model = keras.models.Sequential([
layers.Flatten(input_shape=(28, 28)),
layers.Dense(128, activation='relu'),
layers.Dropout(0.2),

```
layers.Dense(10, activation='softmax')
])
# Compile the model
model.compile(optimizer='adam',

loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)
# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print("\nTest accuracy:", test_acc)
# Save the model
model.save("handwritten_digit_recognition_model.h5")
import tensorflow as tf
from tensorflow.keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt                          # Load the saved model
model = load_model("handwritten_digit_recognition_model.h5")
# Load an example image (you can replace this with your own image)
image_path = "D:/number.jpg"
image = tf.keras.preprocessing.image.load_img(image_path,
                          color_mode="grayscale",
                          target_size=(28, 28))

input_data = tf.keras.preprocessing.image.img_to_array(image)
input_data = input_data / 255.0 # Normalize the pixel values
# Make a prediction on the input data
prediction = model.predict(np.array([input_data]))
# Get the predicted digit (the class with the highest probability)
predicted_digit = np.argmax(prediction)
# Display the image and the predicted digit
plt.imshow(input_data.reshape(28, 28), cmap="binary")
plt.show()                ........expalin this code line ny line
```

Of course! Let's **explain your entire code line by line**:

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

- Import TensorFlow and Keras modules.
- `layers` is for easily creating neural network layers.

```python
# Load the MNIST dataset, a collection of handwritten digits
mnist = keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- Load **MNIST dataset**.
- `x_train` and `x_test`: Images (28×28 grayscale digits).
- `y_train` and `y_test`: Labels (0–9).

```python
# Normalize pixel values to be in the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0
```

- Normalize pixel values from 0–255 to 0–1.
- Helps model training become faster and stable.

```python
# Build the neural network model
model = keras.models.Sequential([
```

- Start building a **Sequential model** (layers added one by one).

```python
layers.Flatten(input_shape=(28, 28)),
```

- **Flatten** the 2D (28×28) image into a 1D vector (784 values).
- Prepares the image for fully connected (Dense) layers.

```python
layers.Dense(128, activation='relu'),
```

- **Dense layer** with 128 neurons.
- Activation function: **ReLU** (introduces non-linearity).

```python
layers.Dropout(0.2),
```

- **Dropout layer**: Randomly sets 20% of neurons to zero during training.
- Helps prevent **overfitting**.

```python
layers.Dense(10, activation='softmax')
])
```

- **Output Dense layer**:
  - 10 neurons (for digits 0–9).
  - **Softmax** activation outputs probabilities that sum to 1.

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

- Compile the model:
  - **Optimizer**: `adam` (adaptive and fast optimizer),
  - **Loss function**: `sparse_categorical_crossentropy` (for integer labels),
  - **Metric**: `accuracy`.

```python
# Train the model
model.fit(x_train, y_train, epochs=5)
```

- Train the model for **5 epochs** (5 passes over the training set).

```python
# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print("\nTest accuracy:", test_acc)
```

- Evaluate how well the model performs on the **test set**.
- Print the final test accuracy.

```python
# Save the model
model.save("handwritten_digit_recognition_model.h5")
```

- Save the trained model into an **H5 file** (`.h5`) format.
- You can reuse or deploy this model later without retraining.

Now second part: Loading the saved model and making a prediction

```python
import tensorflow as tf
from tensorflow.keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
```

- Import required libraries for loading model, handling arrays, and plotting.

```python
# Load the saved model
model = load_model("handwritten_digit_recognition_model.h5")
```

- Load the previously saved model.

```python
# Load an example image (you can replace this with your own image)
image_path = "D:/number.jpg"
image = tf.keras.preprocessing.image.load_img(image_path,
                                               color_mode="grayscale",
                                               target_size=(28, 28))
```

- Load an image (`` `number.jpg` ``) from the specified path.
- Convert it to **grayscale** and resize it to **28×28** (to match MNIST format).

```python
input_data = tf.keras.preprocessing.image.img_to_array(image)
input_data = input_data / 255.0 # Normalize the pixel values
```

- Convert the image to a NumPy array.
- Normalize pixel values between 0 and 1, just like during training.

```python
# Make a prediction on the input data
prediction = model.predict(np.array([input_data]))
```

- Predict the probabilities for each digit (0–9).
- Wrap `` `input_data` `` in a list (`` `np.array([input_data])` ``) to make it a batch (batch_size=1).

```python
# Get the predicted digit (the class with the highest probability)
predicted_digit = np.argmax(prediction)
```

- `` `np.argmax` `` returns the index of the highest probability → **predicted digit**.

```python
# Display the image and the predicted digit
plt.imshow(input_data.reshape(28, 28), cmap="binary")
plt.show()
```

- Show the input image (reshaped back to 28×28) in **binary (black-and-white)** color map.

🚀 Final Summary:

- You trained a simple neural network to recognize handwritten digits.
- Saved the model to disk (`.h5`).
- Loaded it later.
- Made a prediction on a new external image.
- Visualized the input image.

Would you also like me to show **how you can print the predicted digit on the image** itself using `plt.text`? 📷🔢 (it looks very nice!)
Let me know!