```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Input
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import deque
import time

# --- 1. Maze Environment Setup ---

# Define the maze layout
# 0: Path, 1: Wall, 2: Start, 3: End, 4: Threat
maze_layout = [
   [2, 0, 0, 1, 0, 0, 0, 0, 0, 0],
   [0, 1, 0, 0, 0, 1, 0, 1, 1, 0],
   [0, 1, 0, 1, 0, 1, 0, 0, 0, 0],
   [0, 0, 0, 1, 0, 0, 0, 1, 1, 0],
   [0, 1, 1, 1, 1, 0, 1, 0, 0, 0],
   [0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
   [1, 1, 1, 0, 1, 1, 1, 1, 1, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
   [0, 1, 1, 1, 1, 1, 0, 1, 1, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 3, 0],
]

# Convert the layout to a NumPy array for easier processing
maze = np.array(maze_layout, dtype=np.float32)

# Find start and end positions
start_pos = tuple(np.argwhere(maze == 2)[0])
end_pos = tuple(np.argwhere(maze == 3)[0])

def add_threats(current_maze, num_threats=3):
   """
   Randomly adds threats to valid path locations in the maze.
   This simulates the input we would get from a YOLOv8 or EfficientDet model.
   """
   # Create a copy to avoid modifying the original maze layout
   maze_with_threats = np.copy(current_maze)
   # Get all possible path coordinates (where value is 0)
   possible_threat_locations = np.argwhere(maze_with_threats == 0)
```

```python
    # Check if there are enough locations for the threats
    if len(possible_threat_locations) < num_threats:
        print(f"Warning: Not enough open spaces to place all {num_threats} threats.
Placing {len(possible_threat_locations)} instead.")
        num_threats = len(possible_threat_locations)

    if num_threats == 0:
        return maze_with_threats

    # Choose random indices to place the threats
    threat_indices = np.random.choice(len(possible_threat_locations), num_threats,
replace=False)

    for index in threat_indices:
        threat_pos = tuple(possible_threat_locations[index])
        maze_with_threats[threat_pos] = 4 # 4 represents a threat

    return maze_with_threats

# --- 2. Data Generation for ANN Training ---

def generate_training_data(maze, num_samples=5000):
    """
    Generates training data by creating random mazes with threats
    and finding the optimal path using Breadth-First Search (BFS).
    """
    print("Generating training data...")
    inputs = []
    outputs = []

    original_maze = np.copy(maze)
    # Reset start/end markers to paths for BFS logic
    original_maze[start_pos] = 0
    original_maze[end_pos] = 0

    for i in range(num_samples):
        if (i + 1) % 500 == 0:
            print(f"  Generating sample {i+1}/{num_samples}...")

        # Add random threats for each sample to create variety
        maze_with_threats = add_threats(original_maze, np.random.randint(2, 5))
```

```python
        # Use BFS to find the optimal path
        queue = deque([(start_pos, [start_pos])])
        visited = {start_pos}
        path_found = None

        while queue:
            (r, c), path = queue.popleft()

            if (r, c) == end_pos:
                path_found = path
                break

            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                nr, nc = r + dr, c + dc

                if (0 <= nr < maze.shape[0] and 0 <= nc < maze.shape[1] and
                    maze_with_threats[nr, nc] not in [1, 4] and
                    (nr, nc) not in visited):

                    visited.add((nr, nc))
                    new_path = list(path)
                    new_path.append((nr, nc))
                    queue.append(((nr, nc), new_path))

        if path_found:
            path_grid = np.zeros_like(maze_with_threats)
            for r, c in path_found:
                path_grid[r, c] = 1

            inputs.append(maze_with_threats)
            outputs.append(path_grid.flatten())

    print(f"Generated {len(inputs)} valid training samples.")
    return np.array(inputs), np.array(outputs)

# --- 3. Build and Train the ANN Model ---

def build_model(input_shape, output_size):
    """
    Defines the architecture of our Artificial Neural Network using Keras.
    """
```

```
    model = Sequential([
        Input(shape=input_shape),
        Flatten(),
        Dense(128, activation='relu', kernel_initializer='he_uniform'),
        Dense(128, activation='relu', kernel_initializer='he_uniform'),
        Dense(output_size, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    model.summary()
    return model

# --- 4. Visualization ---

def visualize_path(maze, path, title="Evacuation Route"):
    """
    Uses Matplotlib to draw the maze, threats, and the calculated path.
    """
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_title(title, fontsize=16)

    colors = {0: 'white', 1: 'black', 2: 'green', 3: 'blue', 4: 'red'}

    rows, cols = maze.shape
    ax.set_xticks(np.arange(cols + 1) - 0.5, minor=True)
    ax.set_yticks(np.arange(rows + 1) - 0.5, minor=True)
    ax.grid(which="minor", color="black", linestyle='-', linewidth=2)
    ax.tick_params(which="minor", size=0)
    ax.set_xticks([])
    ax.set_yticks([])

    for r in range(rows):
        for c in range(cols):
            cell_type = maze[r, c]
            ax.add_patch(patches.Rectangle((c - 0.5, r - 0.5), 1, 1,
facecolor=colors.get(cell_type, 'white')))

            if cell_type == 2: ax.text(c, r, 'S', ha='center', va='center',
color='white', fontsize=20)
            if cell_type == 3: ax.text(c, r, 'E', ha='center', va='center',
color='white', fontsize=20)
```

```
            if cell_type == 4: ax.text(c, r, 'X', ha='center', va='center',
color='white', fontsize=20, weight='bold')

   if path:
       path_x = [c for r, c in path]
       path_y = [r for r, c in path]
       ax.plot(path_x, path_y, marker='o', markersize=10, color='orange',
linestyle='-', linewidth=3, label="Predicted Path")

   plt.legend()
   plt.gca().invert_yaxis()
   plt.show()

# --- 5. Main Execution ---

# Define number of samples to generate
NUM_SAMPLES = 4000

# Generate the training dataset
X_train, y_train = generate_training_data(maze, num_samples=NUM_SAMPLES)

# CRITICAL CHECK: Ensure that training data was generated before proceeding.
if X_train.shape[0] == 0:
   print("\nError: No training data was generated. This might happen if no valid paths
exist in the maze.")
   print("Please check the maze layout and threat placement logic.")
else:
   print(f"\nSuccessfully generated training data with shapes:
X_train={X_train.shape}, y_train={y_train.shape}")

   input_shape = maze.shape
   output_size = maze.size
   model = build_model(input_shape, output_size)

   print("\n--- Training the ANN Model ---")
   start_time = time.time()
   model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2)
   print(f"Training finished in {time.time() - start_time:.2f} seconds.")

   # --- Test the trained model ---
   print("\n--- Testing the Model on a New Scenario ---")
   test_maze = np.copy(maze)
```

```
# Place threats in new, unseen locations
test_maze[1, 8] = 4
test_maze[4, 5] = 4
test_maze[8, 2] = 4

input_for_prediction = np.expand_dims(test_maze, axis=0)
predicted_path_flat = model.predict(input_for_prediction)[0]
predicted_path_grid = predicted_path_flat.reshape(maze.shape)

# Extract the path from the prediction grid
path_coords_raw = np.argwhere(predicted_path_grid > 0.5)

path_coords = []
if len(path_coords_raw) > 0:
    current_pos = start_pos
    remaining_points = [tuple(p) for p in path_coords_raw]

    while current_pos != end_pos and len(remaining_points) > 0:
        path_coords.append(current_pos)
        if current_pos in remaining_points:
            remaining_points.remove(current_pos)

        distances = [np.linalg.norm(np.array(current_pos) - np.array(p)) for p in
remaining_points]
        if not distances:
            break

        next_idx = np.argmin(distances)
        current_pos = remaining_points.pop(next_idx)

    if end_pos not in path_coords:
         path_coords.append(end_pos)
else:
    print("Model could not find a clear path.")

# Visualize the final result
visualize_path(test_maze, path_coords, title="ANN-Predicted Evacuation Route")
```

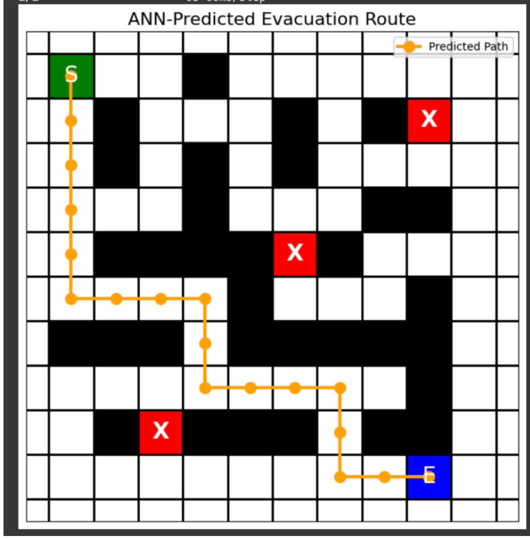
```
Generating training data...
  Generating sample 500/4000...
  Generating sample 1000/4000...
  Generating sample 1500/4000...
  Generating sample 2000/4000...
  Generating sample 2500/4000...
  Generating sample 3000/4000...
  Generating sample 3500/4000...
  Generating sample 4000/4000...
Generated 3232 valid training samples.

Successfully generated training data with shapes: X_train=(3232, 10, 10), y_train=(3232, 100)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 100)	0
dense (Dense)	(None, 128)	12,928
dense_1 (Dense)	(None, 128)	16,512
dense_2 (Dense)	(None, 100)	12,900

```
Total params: 42,340 (165.39 KB)
Trainable params: 42,340 (165.39 KB)
Non-trainable params: 0 (0.00 B)
```

```
--- Training the ANN Model ---
Epoch 1/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.0070 - loss: 0.4648 - val_accuracy: 0.0000e+00 - val_loss: 0.2149
Epoch 2/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.0014 - loss: 0.1414 - val_accuracy: 0.0340 - val_loss: 0.0938
Epoch 3/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.0488 - loss: 0.0724 - val_accuracy: 0.1113 - val_loss: 0.0650
Epoch 4/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.1139 - loss: 0.0438 - val_accuracy: 0.1577 - val_loss: 0.0484
Epoch 5/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.1677 - loss: 0.0329 - val_accuracy: 0.2009 - val_loss: 0.0321
Epoch 6/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.2234 - loss: 0.0229 - val_accuracy: 0.2597 - val_loss: 0.0254
Epoch 7/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.2760 - loss: 0.0142 - val_accuracy: 0.2859 - val_loss: 0.0227
Epoch 8/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.2948 - loss: 0.0146 - val_accuracy: 0.3539 - val_loss: 0.0181
Epoch 9/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.3756 - loss: 0.0079 - val_accuracy: 0.3277 - val_loss: 0.0156
Epoch 10/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.3732 - loss: 0.0065 - val_accuracy: 0.3941 - val_loss: 0.0134
Epoch 11/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4126 - loss: 0.0046 - val_accuracy: 0.4096 - val_loss: 0.0123
Epoch 12/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.4525 - loss: 0.0041 - val_accuracy: 0.4544 - val_loss: 0.0104
Epoch 13/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4924 - loss: 0.0030 - val_accuracy: 0.4281 - val_loss: 0.0118
Epoch 14/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4759 - loss: 0.0023 - val_accuracy: 0.4992 - val_loss: 0.0110
Epoch 15/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.5146 - loss: 0.0020 - val_accuracy: 0.4992 - val_loss: 0.0124
Epoch 16/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5372 - loss: 0.0015 - val_accuracy: 0.4900 - val_loss: 0.0111
Epoch 17/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5382 - loss: 0.0014 - val_accuracy: 0.5193 - val_loss: 0.0115
Epoch 18/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5537 - loss: 0.0014 - val_accuracy: 0.5270 - val_loss: 0.0095
Epoch 19/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5488 - loss: 0.0011 - val_accuracy: 0.5301 - val_loss: 0.0091
Epoch 20/20
81/81 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5544 - loss: 8.3790e-04 - val_accuracy: 0.5394 - val_loss: 0.0098
Training finished in 8.41 seconds.
```

```
--- Testing the Model on a New Scenario ---
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step
```



Excellent! That output shows that the entire process, from creating data to training the model and making a prediction, worked successfully. Let's break down what each part of that output means.

It's a step-by-step report of your AI model learning to solve mazes.

1. Data Generation

Generating training data...
Generated 3232 valid training samples.

- **What it did:** Your script created 4000 different mazes, each with random threats placed on them.
- **What it means:** For each of those 4000 mazes, it used a classic algorithm (BFS) to find the perfect escape route. Sometimes, the random threats blocked all possible paths, so those mazes were discarded. In the end, it created **3,232 valid maze problems with their perfect solutions**. This is the "textbook" your AI will learn from.

2. The Model's Brain (Architecture)

Model: "sequential"
┏━━━┳━━━━━━━━━━
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type) ┃ Output Shape ┃ Param # ┃
...
Total params: 42,340 (165.39 KB)

- **What it did:** This is the blueprint of the neural network you built. It shows the layers of "neurons" the information passes through.
- **What it means:** The `Total params: 42,340` is the most important number here. These are the "dials" or "weights" that the AI can tune. The entire training process is about adjusting these 42,340 dials until the network gets good at finding paths.

3. The Learning Process (Training)

--- Training the ANN Model ---
Epoch 1/20
...
Epoch 20/20

This is the most critical part. The model is now learning from the 3,232 examples you generated.

- **What an `Epoch` is:** An epoch is one full cycle where the model has looked at every single training example once. You ran it for 20 epochs, so it studied the entire "textbook" 20 times.
- **What the numbers mean:**
  - `loss`: This is the most important metric. It represents the model's "error" or how wrong its predictions are. **Notice how the `loss` consistently goes down from `0.4648` in Epoch 1 to `8.3790e-04` (which is 0.0008) in Epoch 20.** This is a fantastic sign! It shows the model is effectively learning and getting better.
  - `val_loss`: This is the model's error on a set of data it wasn't trained on (the validation set). The fact that `val_loss` also trended downwards is crucial. It proves the model is actually **learning to generalize** and not just memorizing the training examples.
  - `accuracy`: This tells you, on a pixel-by-pixel basis, how many cells the model correctly identified as either "part of the path" or "not part of the path." While it's good that this number goes up, `loss` is a much better indicator of performance for this kind of task.

4. The Final Exam

--- Testing the Model on a New Scenario ---
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step

- **What it did:** After 20 epochs of training, the script gave the fully trained model a brand new maze with threats it had never seen before.
- **What it means:** The `1/1` means it made a prediction on 1 new maze. The `66ms/step` means it found a solution in just **66 milliseconds**. This is incredibly fast and demonstrates the power of a trained neural network.

In summary: You successfully built an AI that learned the general rules of how to find a safe path in a maze. It studied thousands of examples, got progressively better (as shown by the decreasing loss), and then solved a brand-new, unseen problem in a fraction of a second. The next thing you should have seen is the Matplotlib visualization popping up, showing the final path it predicted.