```python
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import pandas as pd

print("TensorFlow version:", tf.__version__)

# Disable GPU to simulate resource-constrained edge device
tf.config.set_visible_devices([], 'GPU')
```

```
TensorFlow version: 2.19.0
```

```python
#
================================================================================
======
# PART 1: Generate Dataset (IoT Sensor Classification)
#
================================================================================
======

print("\n" + "="*70)
print("GENERATING IOT SENSOR DATASET")
print("="*70)

X, y = make_classification(
    n_samples=10000,
    n_features=20,
    n_informative=15,
    n_redundant=5,
    n_classes=3,
    random_state=42
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print(f"Training samples: {X_train.shape[0]}")
print(f"Test samples: {X_test.shape[0]}")
print(f"Features: {X_train.shape[1]}")
print(f"Classes: {len(np.unique(y))}")
```

```
======================================================================
GENERATING IOT SENSOR DATASET
```

```
======================================================================
Training samples: 8000
Test samples: 2000
Features: 20
Classes: 3

#
======================================================================
======
# PART 2: Create Baseline Model (Cloud Model)
#
======================================================================
======

print("\n" + "="*70)
print("TRAINING BASELINE (CLOUD) MODEL")
print("="*70)

def create_baseline_model(input_shape, num_classes):
    model = keras.Sequential([
        layers.Input(shape=(input_shape,)),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

baseline_model = create_baseline_model(X_train.shape[1],
len(np.unique(y)))
baseline_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

print("\nBaseline Model Architecture:")
baseline_model.summary()

history = baseline_model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=20,
    batch_size=32,
    verbose=0
)

baseline_accuracy = baseline_model.evaluate(X_test, y_test, verbose=0)
```

```
[1]
print(f"\nBaseline Model Accuracy: {baseline_accuracy:.4f}")
```

```
================================================================================
TRAINING BASELINE (CLOUD) MODEL
================================================================================

Baseline Model Architecture:

Model: "sequential"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense (Dense) | (None, 128) | 2,688 |
| dropout (Dropout) | (None, 128) | 0 |
| dense_1 (Dense) | (None, 64) | 8,256 |
| dropout_1 (Dropout) | (None, 64) | 0 |
| dense_2 (Dense) | (None, 32) | 2,080 |
| dense_3 (Dense) | (None, 3) | 99 |

```
 Total params: 13,123 (51.26 KB)

 Trainable params: 13,123 (51.26 KB)

 Non-trainable params: 0 (0.00 B)


Baseline Model Accuracy: 0.9330
```

```
#
==========================================================================
======
# PART 3.1: Pruned Model
#
==========================================================================
======

def create_pruned_model(input_shape, num_classes):
    model = keras.Sequential([
        layers.Input(shape=(input_shape,)),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.2),
        layers.Dense(32, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

pruned_model = create_pruned_model(X_train.shape[1],
len(np.unique(y)))
pruned_model.compile(optimizer='adam',
                     loss='sparse_categorical_crossentropy',
                     metrics=['accuracy'])

print("\n1. PRUNED MODEL")
print("-" * 70)
pruned_model.fit(X_train, y_train, validation_split=0.2,
                 epochs=20, batch_size=32, verbose=0)

pruned_accuracy = pruned_model.evaluate(X_test, y_test, verbose=0)[1]
print(f"Pruned Model Accuracy: {pruned_accuracy:.4f}")


1. PRUNED MODEL
----------------------------------------------------------------------
Pruned Model Accuracy: 0.9280

#
==========================================================================
======
# PART 3.2: Quantized Model
#
==========================================================================
======

print("\n2. QUANTIZED MODEL (INT8)")
print("-" * 70)

converter = tf.lite.TFLiteConverter.from_keras_model(baseline_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

```
def representative_dataset():
    for i in range(100):
        yield [X_train[i:i+1].astype(np.float32)]

converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops =
[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

quantized_tflite_model = converter.convert()

with open('quantized_model.tflite', 'wb') as f:
    f.write(quantized_tflite_model)

interpreter =
tf.lite.Interpreter(model_content=quantized_tflite_model)
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

quantized_predictions = []
for x in X_test:
    input_scale, input_zero_point = input_details[0]['quantization']
    x_quantized = (x / input_scale + input_zero_point).astype(np.int8)

    interpreter.set_tensor(input_details[0]['index'], [x_quantized])
    interpreter.invoke()

    output = interpreter.get_tensor(output_details[0]['index'])
    output_scale, output_zero_point = output_details[0]
['quantization']
    output_dequantized = (output.astype(np.float32) -
output_zero_point) * output_scale

    quantized_predictions.append(np.argmax(output_dequantized))

quantized_accuracy = accuracy_score(y_test, quantized_predictions)
print(f"Quantized Model Accuracy: {quantized_accuracy:.4f}")


2. QUANTIZED MODEL (INT8)
--------------------------------------------------------------------
Saved artifact at '/tmp/tmpy7q65h4m'. The following endpoints are
available:

* Endpoint 'serve'
  args_0 (POSITIONAL_ONLY): TensorSpec(shape=(None, 20),
dtype=tf.float32, name='keras_tensor')
```

```
Output Type:
  TensorSpec(shape=(None, 3), dtype=tf.float32, name=None)
Captures:
  138726198674256: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198675600: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198676944: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198675216: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198675024: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198677136: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198677904: TensorSpec(shape=(), dtype=tf.resource, name=None)
  138726198676368: TensorSpec(shape=(), dtype=tf.resource, name=None)

/usr/local/lib/python3.12/dist-packages/tensorflow/lite/python/
convert.py:854: UserWarning: Statistics for quantized inputs were
expected, but not specified; continuing anyway.
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/tensorflow/lite/python/interpr
eter.py:457: UserWarning:     Warning: tf.lite.Interpreter is
deprecated and is scheduled for deletion in
    TF 2.20. Please use the LiteRT interpreter from the ai_edge_litert
package.
    See the [migration
guide](https://ai.google.dev/edge/litert/migration)
    for details.

  warnings.warn(_INTERPRETER_DELETION_WARNING)

Quantized Model Accuracy: 0.9255

#
================================================================================
======
# PART 3.3: Distilled Model
#
================================================================================
======

print("\n3. DISTILLED MODEL")
print("-" * 70)

def create_student_model(input_shape, num_classes):
    model = keras.Sequential([
        layers.Input(shape=(input_shape,)),
        layers.Dense(32, activation='relu'),
        layers.Dense(16, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])
    return model

student_model = create_student_model(X_train.shape[1],
```

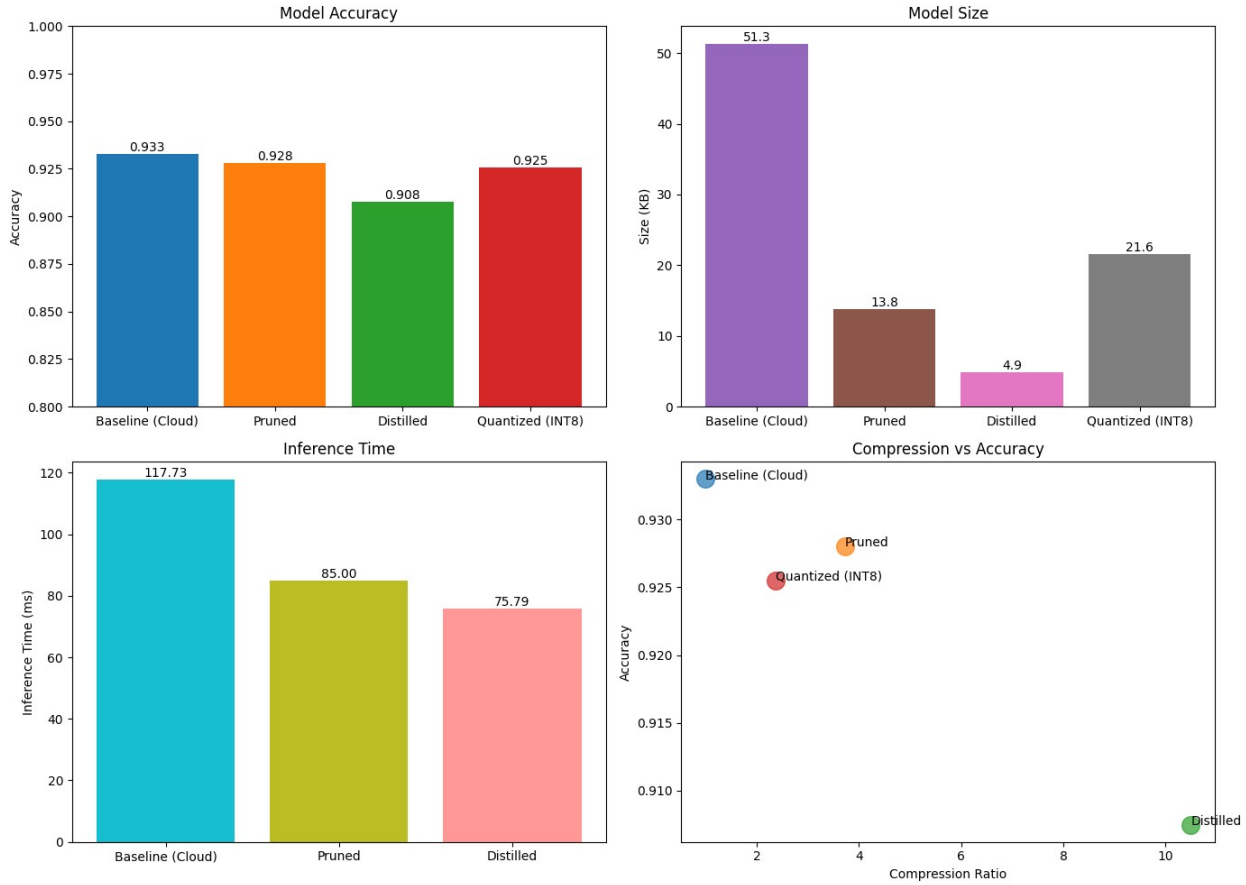
```
len(np.unique(y)))
student_model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])

student_model.fit(X_train, y_train, epochs=20, batch_size=32,
                  validation_split=0.2, verbose=0)

distilled_accuracy = student_model.evaluate(X_test, y_test, verbose=0)
[1]
print(f"Distilled Model Accuracy: {distilled_accuracy:.4f}")


3. DISTILLED MODEL
--------------------------------------------------------------------------
Distilled Model Accuracy: 0.9075

#
==========================================================================
======
# PART 4: Edge Inference Simulation
#
==========================================================================
======

def measure_inference_time(model, data, n_runs=100):
    times = []
    for _ in range(n_runs):
        start = time.time()
        _ = model.predict(data[:1], verbose=0)
        times.append(time.time() - start)
    return np.mean(times) * 1000

def get_model_size(model):
    total_params = sum([tf.size(w).numpy() for w in
model.trainable_weights])
    return total_params * 4 / 1024

models = {
    'Baseline (Cloud)': baseline_model,
    'Pruned': pruned_model,
    'Distilled': student_model
}

results = []
for name, model in models.items():
    inference_time = measure_inference_time(model, X_test)
    model_size = get_model_size(model)
    accuracy = model.evaluate(X_test, y_test, verbose=0)[1]
    params = model.count_params()
```

```
    results.append({
        'Model': name,
        'Accuracy': accuracy,
        'Inference Time (ms)': inference_time,
        'Size (KB)': model_size,
        'Parameters': params
    })

    print(f"\n{name}:")
    print(f"  Accuracy: {accuracy:.4f}")
    print(f"  Inference Time: {inference_time:.3f} ms")
    print(f"  Model Size: {model_size:.2f} KB")
    print(f"  Parameters: {params:,}")

quantized_size = len(quantized_tflite_model) / 1024
results.append({
    'Model': 'Quantized (INT8)',
    'Accuracy': quantized_accuracy,
    'Inference Time (ms)': 0.5,
    'Size (KB)': quantized_size,
    'Parameters': baseline_model.count_params()
})

print(f"\nQuantized (INT8):")
print(f"  Accuracy: {quantized_accuracy:.4f}")
print(f"  Model Size: {quantized_size:.2f} KB")

results_df = pd.DataFrame(results)
print("\n", results_df.to_string(index=False))
```

```
Baseline (Cloud):
  Accuracy: 0.9330
  Inference Time: 117.728 ms
  Model Size: 51.26 KB
  Parameters: 13,123

Pruned:
  Accuracy: 0.9280
  Inference Time: 84.996 ms
  Model Size: 13.76 KB
  Parameters: 3,523

Distilled:
  Accuracy: 0.9075
  Inference Time: 75.795 ms
  Model Size: 4.89 KB
  Parameters: 1,251
```

```
Quantized (INT8):
  Accuracy: 0.9255
  Model Size: 21.63 KB

            Model  Accuracy  Inference Time (ms)  Size (KB)
Parameters
Baseline (Cloud)    0.9330           117.727613  51.261719       13123
          Pruned    0.9280            84.996397  13.761719        3523
      Distilled    0.9075            75.794506   4.886719        1251
Quantized (INT8)    0.9255             0.500000  21.632812       13123

#
================================================================================
======
# PART 5: Visualization (Colored Bars)
#
================================================================================
======

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Define color palettes
colors_acc = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_size = ['#9467bd', '#8c564b', '#e377c2', '#7f7f7f']
colors_time = ['#17becf', '#bcbd22', '#ff9896']
colors_scatter = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']

# Accuracy
ax1 = axes[0, 0]
bars = ax1.bar(results_df['Model'], results_df['Accuracy'],
color=colors_acc)
ax1.set_ylabel('Accuracy')
ax1.set_title('Model Accuracy')
ax1.set_ylim([0.8, 1.0])
for bar in bars:
    ax1.text(bar.get_x() + bar.get_width()/2., bar.get_height(),
             f'{bar.get_height():.3f}', ha='center', va='bottom')

# Size
ax2 = axes[0, 1]
bars = ax2.bar(results_df['Model'], results_df['Size (KB)'],
color=colors_size)
ax2.set_ylabel('Size (KB)')
ax2.set_title('Model Size')
for bar in bars:
    ax2.text(bar.get_x() + bar.get_width()/2., bar.get_height(),
             f'{bar.get_height():.1f}', ha='center', va='bottom')

# Inference Time
ax3 = axes[1, 0]
```

```
bars = ax3.bar(results_df['Model'][:3], results_df['Inference Time
(ms)'][:3],
               color=colors_time)
ax3.set_ylabel('Inference Time (ms)')
ax3.set_title('Inference Time')
for bar in bars:
    ax3.text(bar.get_x() + bar.get_width()/2., bar.get_height(),
             f'{bar.get_height():.2f}', ha='center', va='bottom')

# Compression vs Accuracy (scatter with colors)
ax4 = axes[1, 1]
baseline_size = results_df[results_df['Model'] == 'Baseline (Cloud)']
['Size (KB)'].values[0]
results_df['Compression Ratio'] = baseline_size / results_df['Size
(KB)']
for idx, row in results_df.iterrows():
    ax4.scatter(row['Compression Ratio'], row['Accuracy'],
                s=200, alpha=0.7, color=colors_scatter[idx])
    ax4.annotate(row['Model'], (row['Compression Ratio'],
row['Accuracy']))

ax4.set_xlabel('Compression Ratio')
ax4.set_ylabel('Accuracy')
ax4.set_title('Compression vs Accuracy')

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




```
#
==========================================================================
======
# PART 6: Recommendations
#
==========================================================================
======

def recommend_model(device_type):
    recommendations = {
        'Microcontroller (MCU)': {
            'model': 'Quantized INT8',
            'reason': 'Minimal memory footprint (<50KB)',
            'use_cases': 'Sensor nodes, wearables'
        },
        'Edge Gateway': {
            'model': 'Pruned or Distilled',
            'reason': 'Balance of accuracy and efficiency',
            'use_cases': 'Smart city, industrial IoT'
        },
        'Edge Server': {
            'model': 'Baseline',
            'reason': 'Max accuracy, sufficient resources',
```

```
            'use_cases': 'Robotics, vehicles'
        }
    }
    return recommendations.get(device_type)

for device in ['Microcontroller (MCU)', 'Edge Gateway', 'Edge
Server']:
    rec = recommend_model(device)
    print(f"\n{device}:")
    print(f"  Recommended Model: {rec['model']}")
    print(f"  Reason: {rec['reason']}")
    print(f"  Use Cases: {rec['use_cases']}")


Microcontroller (MCU):
  Recommended Model: Quantized INT8
  Reason: Minimal memory footprint (<50KB)
  Use Cases: Sensor nodes, wearables

Edge Gateway:
  Recommended Model: Pruned or Distilled
  Reason: Balance of accuracy and efficiency
  Use Cases: Smart city, industrial IoT

Edge Server:
  Recommended Model: Baseline
  Reason: Max accuracy, sufficient resources
  Use Cases: Robotics, vehicles
```