Deploying ANN on low-cost platforms

1. Introduction

Deploying artificial neural networks (ANNs) on low-cost platforms is a critical step in transitioning machine learning (ML) models from research to real-world applications. These platforms include devices like Raspberry Pi, NVIDIA Jetson Nano, and microcontrollers such as ESP32 or Arduino with additional hardware. Despite their limited computational power and memory, they enable cost-effective solutions for edge computing, IoT (Internet of Things), and embedded AI systems.

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the sources of data, such as IoT devices, sensors, and users. Instead of relying on a central cloud server, processing is done at the "edge" of the network, closer to where the data is generated and needed.

Key characteristics of edge computing:

1. **Decentralized processing:** Computation happens on devices or servers located at the edge of the network, like routers, gateways, or even the devices themselves.
2. **Reduced latency:** By processing data locally, edge computing significantly reduces the delay (latency) in getting results. This is crucial for applications that require real-time responses, such as autonomous vehicles, industrial automation, and online gaming.
3. **Improved bandwidth efficiency:** Processing data at the edge reduces the amount of data that needs to be sent to the cloud, freeing up bandwidth and reducing network congestion.
4. **Increased security and privacy:** Keeping sensitive data at the edge can enhance security and privacy by reducing the risk of data breaches during transmission to and from the cloud.
5. **Enhanced reliability and resilience:** Edge computing systems can be designed to operate even when the connection to the cloud is disrupted, ensuring continued functionality in remote or challenging environments.

Benefits of edge computing:

- **Real-time insights:** Edge computing enables real-time analysis of data, allowing for immediate responses and actions.
- **Improved application performance:** Reduced latency leads to faster and more responsive applications.
- **Cost savings:** By reducing data transmission and cloud storage needs, edge computing can lower operational costs.
- **Enhanced security:** Edge computing can provide better security for sensitive data by keeping it closer to the source.
- **Increased scalability:** Edge computing systems can be easily scaled by adding more edge devices or servers.

Examples of edge computing:

- **Self-driving cars:** Autonomous vehicles need to make real-time decisions based on sensor data, making edge computing essential for their operation.
- **Industrial automation:** Edge computing is used in factories to monitor and control machines, improving efficiency and safety.
- **Smart cities:** Edge computing helps analyze data from sensors in smart cities to manage traffic, optimize energy consumption, and improve public safety.

- **Healthcare:** Edge computing is used in healthcare to analyze patient data in real-time, enabling faster diagnoses and treatments.

Platform Descriptions:

- **Raspberry Pi:** A small, affordable computer with GPIO pins, widely used for prototyping and deployment of AI applications. Models like Raspberry Pi 4 offer up to 8GB RAM, making them suitable for lightweight neural network inference.
- **NVIDIA Jetson Nano:** A compact AI computing device with a GPU for accelerating neural network inference, suitable for tasks like computer vision and robotics. It supports TensorFlow and PyTorch for efficient deployment.
- **Arduino with Neural Networks:** Arduino boards, particularly with hardware like Arduino Nano 33 BLE Sense, can be used for deploying simple ANNs in applications like signal processing and anomaly detection. They are best suited for low-power environments where small-scale models suffice.

2. Challenges and Advantages of Low-Cost Platforms

Challenges:

- **Limited Resources:** Low-cost platforms often have restricted computational power, memory, and storage.
- **Energy Efficiency:** Power consumption is a crucial factor for battery-operated devices.
- **Latency Constraints:** Ensuring real-time inference is challenging with limited hardware capabilities.
- **Compatibility:** Running Python-based Keras models on resource-constrained devices requires compatibility adjustments.

Advantages:

- **Affordability:** Low-cost platforms reduce deployment costs, making AI accessible.
- **Portability:** Compact and lightweight devices are ideal for remote or embedded applications.
- **Scalability:** Easy to replicate for distributed systems or IoT networks.

3. Optimizing Neural Networks for Deployment

To deploy neural networks effectively on low-cost platforms, certain optimization techniques are necessary:

a) Model Quantization:

Convert weights and activations from 32-bit floating-point to 8-bit integers. This reduces the model size and improves inference speed with minimal accuracy loss.

b) Pruning:

Remove less significant connections or neurons from the network to reduce computational complexity and memory usage.

Pruning is a technique used to compress and optimize deep learning models by removing unnecessary connections (weights) in the neural network. It's like trimming a tree to remove dead branches and make it more efficient.

How it works:

1. **Identify less important connections:** During training, pruning algorithms identify weights with relatively small magnitudes (close to zero). These weights contribute less to the overall prediction of the model and can be removed without significantly affecting accuracy.
2. **Remove connections (set weights to zero):** The identified connections are removed by setting their weights to zero, effectively "pruning" them from the network.
3. **Fine-tuning:** After pruning, the model is usually fine-tuned for a few epochs to adjust the remaining weights and recover any potential loss in accuracy due to the removal of connections.

Pruning and Quantization

```
1
2   import tensorflow_model_optimization as tfmot
3
4   # Load the .keras model
5   My_model = keras.models.load_model('My_regression_model.keras', compile=False)
6
7   # --- Pruning first ---
8   # Define pruning parameters
9   prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude
10  pruning_params = {
11      'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.50,
12                                                               final_sparsity=0.80,
13                                                               begin_step=0,
14                                                               end_step=1000)
15  }
16
17  # Apply pruning
18  pruned_model = prune_low_magnitude(My_model, **pruning_params)
19
20  # Compile and fine-tune the pruned model
21  pruned_model.compile(optimizer='adam',
22                       loss=keras.losses.MeanSquaredError(),
23                       metrics=['mean_squared_error'])
24
25  # Add the UpdatePruningStep callback to the fit method call
26  callbacks = [tfmot.sparsity.keras.UpdatePruningStep()]  # Create the callback
27  pruned_model.fit(X_train, y_train, epochs=10, callbacks=callbacks)  # Use it during training
```

```python
# --- Quantization after Pruning ---
# Create a TFLiteConverter instance
converter = tf.lite.TFLiteConverter.from_keras_model(pruned_model)

# Set optimization for quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# For 8-bit integer quantization, set inference type to tf.int8
converter.inference_type = tf.int8

# Convert the model to quantized TensorFlow Lite
final_tflite_model = converter.convert()

# Save the final quantized and pruned model
with open('My_model_quantized_pruned.tflite', 'wb') as f:
    f.write(final_tflite_model)
```

c) Model Compression:

Compress the model by reducing redundant weights or using efficient encoding techniques.

d) Knowledge Distillation:

Use a smaller "student" model to mimic the behavior of a larger "teacher" model.

e) Framework-Specific Tools:

- TensorFlow Lite (TFLite) for Keras models.
- ONNX (Open Neural Network Exchange) for interoperability.
- Edge Impulse for low-power devices.

4. Deployment Workflow

Step 1: Model Training and Export

Train the neural network in Python using Keras. Save the trained model in HDF5 or SavedModel format.

```
from tensorflow.keras.models import load_model
model = load_model('model.keras')
model.save('saved_model')
```

Step 2: Model Conversion

Convert the trained model to a format compatible with the target platform, such as TensorFlow Lite (TFLite).

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model('saved_model')
tflite_model = converter.convert()
with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Step 3: Deployment on Platform

Copy the model to the target device and run it using a lightweight inference engine. For example, deploy on Arduino:

1. Use TensorFlow Lite Micro for Arduino.
2. Convert and upload the model as a `.h` file to the Arduino board.

```
xxd -i model.tflite > model.h
```

3. Include the model in the Arduino sketch and load it for inference using TensorFlow Lite Micro.

```
#include "model.h" // Include the converted model
#include <TensorFlowLite.h> // TensorFlow Lite Micro library

// Example setup
void setup() {
  Serial.begin(115200);
  // Load model and allocate memory for inference
  tflite::MicroInterpreter interpreter(...);
  interpreter.AllocateTensors();
}

void loop() {
  // Example inference logic
}
```

Step 4: Optimize Runtime Performance

- Utilize hardware acceleration where possible (e.g., Arduino Portenta H7 with a neural network accelerator).
- Use efficient algorithms for pre-processing and inference.
- Keep the model architecture simple to ensure compatibility with the microcontroller.

5. Examples of Low-Cost Deployments

a) Raspberry Pi:

Use a Raspberry Pi for edge AI tasks such as image classification.

- Hardware: Raspberry Pi 4 (4GB RAM).
- Example: Deploy a MobileNetV2-based model for object detection.

Raspberry Pi 5/8GB: 90 EUR

b) NVIDIA Jetson Nano:

Utilize Jetson Nano for accelerated inference in applications such as robotics or real-time video analytics.

- Hardware: NVIDIA Jetson Nano (128-core Maxwell GPU).
- Example: Deploy a YOLOv4 model for real-time object detection.

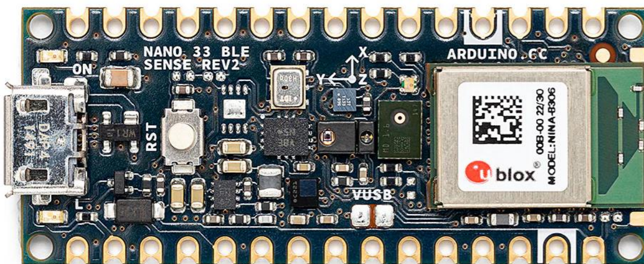
NVIDIA Jetson Orin Nano 8GB Development Kit,   480 EUR

c) Arduino for Conventional ANNs:

Deploy simple conventional ANNs on Arduino boards for sensor data processing.

- Hardware: Arduino Nano 33 BLE Sense.
- Example: Detect temperature anomalies using a simple ANN model deployed with TensorFlow Lite Micro.

Arduino Nano 33 BLE Sense Rev2   46 EUR



6. Practical Tips and Best Practices

- **Profiling:** Use profiling tools to identify bottlenecks.
- **Batch Size:** Use smaller batch sizes to fit memory constraints.
- **Lightweight Libraries:** Use frameworks tailored for low-cost platforms (e.g., MicroTensor, TensorFlow Lite).
- **Testing:** Thoroughly test the deployment for latency, accuracy, and stability.

7. Conclusion

Deploying neural networks on low-cost platforms bridges the gap between AI research and real-world applications. While these platforms present challenges, careful optimization and appropriate tools make them viable for numerous use cases. By leveraging techniques like quantization, pruning, and model conversion, developers can ensure efficient deployment on constrained devices.

8. Further Reading and Resources

- TensorFlow Lite Documentation: https://www.tensorflow.org/lite
- Raspberry Pi Official Guide: https://www.raspberrypi.org/documentation
- Edge Impulse: https://www.edgeimpulse.com
- Jetson Nano Resources: https://developer.nvidia.com/embedded/jetson-nano
- Arduino with TensorFlow Lite Micro: https://www.arduino.cc/pro/tutorials/tensorflow-lite