

**Implementing Parameter-Efficient Fine-Tuning (PEFT) from Scratch**

This guide provides a comprehensive implementation of Low-Rank Adaptation (LoRA), a prominent PEFT method, using only NumPy. We'll create a complete training system that demonstrates how to adapt neural networks efficiently by modifying less than 1% of parameters.

**Core Components of PEFT**

**1. Weight Decomposition Principle**

Neural network weight updates (ΔW) can be approximated using low-rank matrices:  
$ \Delta W \approx BA $  
Where:

* $ B \in \mathbb{R}^{d \times r} $
* $ A \in \mathbb{R}^{r \times k} $
* Rank $ r \ll \min(d,k) $

**2. Architecture Modifications**

class LoRALayer:  
 def \_\_init\_\_(self, input\_dim, output\_dim, rank, alpha):  
 # Original pretrained weights (frozen)  
 self.W = np.random.randn(input\_dim, output\_dim) \* 0.01  
   
 # Trainable low-rank matrices  
 self.A = np.random.randn(input\_dim, rank) \* 0.01  
 self.B = np.zeros((output\_dim, rank))  
   
 # Scaling factor  
 self.scale = alpha / rank  
 self.rank = rank  
  
 def forward(self, x):  
 lora\_adjustment = self.scale \* (self.A @ self.B.T)  
 return x @ (self.W + lora\_adjustment)

**Complete Training System**

**1. Network Architecture**

class NeuralNetwork:  
 def \_\_init\_\_(self, layers, rank=2, alpha=1):  
 self.layers = []  
 for i in range(len(layers)-1):  
 self.layers.append(  
 LoRALayer(layers[i], layers[i+1], rank, alpha)  
 )  
   
 def predict(self, x):  
 for layer in self.layers:  
 x = layer.forward(x)  
 return x

**2. Training Loop with Manual Backpropagation**

def train(model, X, y, epochs=100, lr=0.01):  
 for epoch in range(epochs):  
 total\_loss = 0  
 for xi, yi in zip(X, y):  
 # Forward pass  
 activations = [xi]  
 for layer in model.layers:  
 activations.append(layer.forward(activations[-1]))  
   
 # Compute loss  
 error = activations[-1] - yi  
 loss = 0.5 \* np.sum(error\*\*2)  
 total\_loss += loss  
   
 # Backward pass  
 gradients = [error]  
 for i in reversed(range(len(model.layers))):  
 layer = model.layers[i]  
 x\_in = activations[i]  
   
 # Compute gradients  
 grad\_W = np.outer(x\_in, gradients[-1])  
 grad\_A = layer.scale \* (grad\_W @ layer.B)  
 grad\_B = layer.scale \* (grad\_W.T @ layer.A).T  
   
 # Update parameters  
 layer.A -= lr \* grad\_A  
 layer.B -= lr \* grad\_B  
   
 # Propagate gradient  
 gradients.append(x\_in @ layer.W.T)  
   
 if epoch % 10 == 0:  
 print(f"Epoch {epoch}: Loss {total\_loss/len(X):.4f}")

**Experimental Validation**

**1. Dataset Preparation**

# XOR problem with 2D inputs  
X = np.array([[0,0], [0,1], [1,0], [1,1]])  
y = np.array([[^0], [^1], [^1], [^0]])  
  
# Add noise for realistic training  
X += np.random.normal(0, 0.1, X.shape)

**2. Model Configuration**

model = NeuralNetwork(  
 layers=[2, 4, 1],  
 rank=2,  
 alpha=1  
)

**3. Training Execution**

train(model, X, y, epochs=1000, lr=0.01)  
  
# Verify predictions  
print("\nFinal predictions:")  
for xi in X:  
 pred = model.predict(xi)  
 print(f"Input {xi} -> Output: {pred[^0]:.4f}")

**Mathematical Foundations**

**1. Gradient Calculations**

For input $ x $, output error $ \delta $, and LoRA parameters $ A,B $:

**2. Rank Selection**

Optimal rank $ r $ follows:

Where $ d \times k $ is the original weight matrix size

**Advanced PEFT Techniques**

**1. Dynamic Rank Adaptation**

class DynamicLoRA(LoRALayer):  
 def \_\_init\_\_(self, input\_dim, output\_dim, max\_rank, alpha):  
 super().\_\_init\_\_(input\_dim, output\_dim, max\_rank, alpha)  
 self.rank\_mask = np.ones(max\_rank)  
   
 def forward(self, x):  
 effective\_B = self.B \* self.rank\_mask  
 lora\_adjustment = self.scale \* (self.A @ effective\_B.T)  
 return x @ (self.W + lora\_adjustment)  
   
 def update\_rank(self, new\_rank):  
 self.rank\_mask[new\_rank:] = 0

**2. Quantized LoRA (QLoRA)**

def quantize(matrix, bits=4):  
 scale = np.max(np.abs(matrix))  
 q = np.round(matrix / scale \* (2\*\*(bits-1)-1))  
 return q.astype(np.int8), scale  
  
def dequantize(q\_matrix, scale, bits=4):  
 return q\_matrix.astype(float) \* scale / (2\*\*(bits-1)-1)

**Performance Considerations**

**1. Memory Optimization**

For $ d=1024, k=1024, r=8 $:

**2. Computational Complexity**

|  |  |  |
| --- | --- | --- |
| Method | Multiplications | Additions |
| Full FT | $ O(dk) $ | $ O(dk) $ |
| LoRA | $ O(dr + rk) $ | $ O(dk) $ |
| Speedup | $ \frac{dk}{dr+rk} $ | - |

**Real-World Implementation Strategy**

1. **Layer Selection**: Apply LoRA to attention matrices (Q, K, V) and feed-forward layers
2. **Rank Scaling**: Use higher ranks (8-16) for critical layers
3. **Gradient Accumulation**:

def train\_accumulate(model, X, y, batch\_size=4):  
 grad\_accum\_A = [np.zeros\_like(l.A) for l in model.layers]  
 grad\_accum\_B = [np.zeros\_like(l.B) for l in model.layers]  
   
 for i, (xi, yi) in enumerate(zip(X, y)):  
 # ... compute gradients ...  
 for j, layer in enumerate(model.layers):  
 grad\_accum\_A[j] += grad\_A  
 grad\_accum\_B[j] += grad\_B  
   
 if (i+1) % batch\_size == 0:  
 for j, layer in enumerate(model.layers):  
 layer.A -= lr \* grad\_accum\_A[j] / batch\_size  
 layer.B -= lr \* grad\_accum\_B[j] / batch\_size  
 grad\_accum\_A = [np.zeros\_like(l.A) for l in model.layers]  
 grad\_accum\_B = [np.zeros\_like(l.B) for l in model.layers]

**Evaluation Metrics**

**1. Parameter Efficiency**

Typical PE values: 0.1%-2%

**2. Task Performance Retention**

State-of-the-art: 95%-99% TPR

**Conclusion**

This implementation demonstrates the core principles of PEFT through LoRA, showing how to achieve parameter-efficient adaptation while maintaining model performance. The system can be extended to support:

1. Multi-head attention layers
2. Transformer architectures
3. Quantized implementations
4. Dynamic rank adaptation

Key advantages of this approach:

* 100-1000x fewer trainable parameters
* 2-5x faster training times
* Preserved pretrained knowledge
* Efficient multi-task serving

For production systems, consider adding:

* Gradient checkpointing
* Mixed-precision training
* Distributed optimization
* Adaptive rank selection

⁂