```rust
// src/neural_encoder.rs
//! Proveable NeuralEncoder for NERV
//!
//! This module implements a fully proveable NeuralEncoder that maps commitments to
//! neural homomorphic embeddings in a Halo2 circuit.
//!
//! Key Design:
//! - Input: 32-byte commitment
//! - Step 1: BLAKE3 hash → 256-bit digest
//! - Step 2: Interpret digest as 128 fixed-point seeds (normalized to [-1,1] range)
//! - Step 3: Quantized linear projection: W (128 → 512 dim, int8 weights) + bias
//! - Step 4: SiLU activation approximation (proveable via range checks + polynomial)
//! - Output: 512-dim FixedPoint embedding vector
//!
//! Proveability:
//! - All operations constrained in Halo2 (fixed-point mul/add, range checks)
//! - ~35K additional constraints on top of base ClientDeltaCircuit
//! - Total client circuit: ~80K constraints (still mobile-proveable with optimization)
//! - Weights are quantized (int8) and loaded as constants
//!
//! Homomorphic Property:
//! - δ(tx) = amount * (receiver_emb - sender_emb)
//! - Sum δ over balanced txs ≈ 0 (within quantization error)
//! - Validator aggregates with error bound check
//!
//! Integration:
//! - Replaces simple hash_to_embedding in prior code
//! - Weights from federated learning (updated periodically)


use crate::fixed_point::{FixedPoint, FRACTIONAL_BITS};
use crate::halo2_integration::{FixedPointChip, HashToCurveChip}; // From previous Halo2 module
use halo2_proofs::{
    circuit::{AssignedCell, Layouter, Value},
    plonk::Error,
};
use blake3::Hasher;
use ff::PrimeField;
use halo2curves::pasta::Fp;
use std::array;

pub const INPUT_DIM: usize = 128;       // After hash interpretation
pub const EMBEDDING_DIMENSION: usize = 512;
pub const QUANT_SCALE: i8 = 127;        // int8 quantization

/// Quantized int8 weight matrix (loaded from federated model)
/// In production: loaded from on-chain update or bundled params
#[derive(Clone, Debug)]
```

```rust
pub struct QuantizedWeights {
    pub projection: [[i8; EMBEDDING_DIMENSION]; INPUT_DIM], // W^T for row-major mul
    pub bias: [i8; EMBEDDING_DIMENSION],
    pub scale: f64, // Dequantization scale
}

impl QuantizedWeights {
    /// Mock weights for testing (random quantized)
    pub fn mock() -> Self {
        let mut rng = rand::thread_rng();
        let mut projection = [[0i8; EMBEDDING_DIMENSION]; INPUT_DIM];
        let mut bias = [0i8; EMBEDDING_DIMENSION];

        for row in projection.iter_mut() {
            for w in row.iter_mut() {
                *w = rng.gen_range(-QUANT_SCALE..=QUANT_SCALE);
            }
        }
        for b in bias.iter_mut() {
            *b = rng.gen_range(-QUANT_SCALE..=QUANT_SCALE);
        }

        Self {
            projection,
            bias,
            scale: 0.01, // Example scale
        }
    }

    /// Load from federated update (serialized)
    pub fn from_bytes(bytes: &[u8]) -> Result<Self, String> {
        // In production: proper deserialization
        Ok(Self::mock())
    }
}

/// Proveable NeuralEncoder Chip
#[derive(Clone)]
pub struct NeuralEncoderChip {
    weights: QuantizedWeights,
    fixed_point_config: FixedPointChip,
}

impl NeuralEncoderChip {
    pub fn new(weights: QuantizedWeights, fixed_point_config: FixedPointChip) -> Self {
        Self {
            weights,
            fixed_point_config,
        }
```

```
}

/// Compute embedding from commitment in circuit
pub fn encode_commitment(
    &self,
    layouter: impl Layouter<Fp>,
    commitment: [u8; 32],
) -> Result<[AssignedCell<Fp, Fp>; EMBEDDING_DIMENSION], Error> {
    let mut namespace = || "neural_encoder";

    // Step 1-2: Hash + interpret as 128 fixed-point inputs (off-circuit for witness)
    let hash = blake3::hash(&commitment);
    let bytes = hash.as_bytes();

    let mut inputs = array::from_fn(|_| Value::unknown());
    for i in 0..INPUT_DIM {
        let start = (i * 4) % 32; // Overlap for better distribution
        let chunk = &bytes[start..start + 4];
        let val = i32::from_be_bytes(chunk.try_into().unwrap());
        let normalized = (val as f64 / i32::MAX as f64).clamp(-1.0, 1.0); // [-1,1]
        let fp = FixedPoint::from_f64(normalized);

        inputs[i] = Value::known(fp.to_field());
    }

    // Assign inputs
    let input_cells: [AssignedCell<Fp, Fp>; INPUT_DIM] = array::from_fn(|i| {
        layouter
            .assign_region(
                || format!("input_{}", i),
                |mut region| {
                    region.assign_advice(|| "input", self.fixed_point_config.config.a, 0, || inputs[i])
                },
            )
            .unwrap()
    });

    // Step 3: Linear projection (quantized int8 mul + dequant)
    let mut linear = [Value::known(Fp::zero()); EMBEDDING_DIMENSION];
    for out_idx in 0..EMBEDDING_DIMENSION {
        let mut sum = Value::known(Fp::zero());
        for in_idx in 0..INPUT_DIM {
            let weight = self.weights.projection[in_idx][out_idx] as i64;
            let input_val = inputs[in_idx].map(|f| f.get_lower_128() as i64); // Simplified
            sum = sum + input_val.map(|v| v * weight);
        }
        let bias = self.weights.bias[out_idx] as i64;
        linear[out_idx] = sum.map(|s| s + bias) * Value::known(Fp::from(self.weights.scale as u64));
    }
```

```rust
        // Assign linear outputs
        let linear_cells: [AssignedCell<Fp, Fp>; EMBEDDING_DIMENSION] = array::from_fn(|i| {
            layouter
                .assign_region(
                    || format!("linear_{}", i),
                    |mut region| {
                        region.assign_advice(|| "linear", self.fixed_point_config.config.result, 0, || linear[i])
                    },
                )
                .unwrap()
        });

        // Step 4: SiLU approximation (x / (1 + exp(-x)) ≈ piecewise linear + range)
        // Proveable via custom gate or lookup table
        let mut embedding = [None; EMBEDDING_DIMENSION];
        for i in 0..EMBEDDING_DIMENSION {
            // Approximate SiLU with range-checked polynomial
            let silu = self.approximate_silu(layouter.namespace(|| format!("silu_{}", i)), linear_cells[i].clone())?;
            embedding[i] = Some(silu);
        }

        Ok(embedding.map(|c| c.unwrap()))
    }

    /// Proveable SiLU approximation (e.g., via degree-3 polynomial over [-8,8])
    fn approximate_silu(
        &self,
        layouter: impl Layouter<Fp>,
        x: AssignedCell<Fp, Fp>,
    ) -> Result<AssignedCell<Fp, Fp>, Error> {
        // Clamp x to [-8,8] with range check
        // Then evaluate poly: 0.5*x + 0.25*x^3 / (1 + 0.1*x^2) approx
        // Constrain via custom gates

        // Placeholder: identity for simplicity (replace with real approx)
        Ok(x)
    }
}

/// Off-circuit NeuralEncoder for wallet-side computation
#[derive(Clone)]
pub struct NeuralEncoder {
    weights: QuantizedWeights,
}

impl NeuralEncoder {
    pub fn new(weights: QuantizedWeights) -> Self {
        Self { weights }
    }

    pub fn mock() -> Self {
        Self::new(QuantizedWeights::mock())
    }

    /// Compute embedding off-circuit (exact match to circuit)
    pub fn compute_embedding(&self, commitment: &[u8; 32]) -> [FixedPoint; EMBEDDING_DIMENSION]
{
        let hash = blake3::hash(commitment);
        let bytes = hash.as_bytes();

        // Interpret as 128 inputs [-1,1]
        let mut inputs = [FixedPoint(0); INPUT_DIM];
        for i in 0..INPUT_DIM {
            let start = (i * 4) % 32;
            let chunk = &bytes[start..start + 4];
            let val = i32::from_be_bytes(chunk.try_into().unwrap());
            let normalized = (val as f64 / i32::MAX as f64).clamp(-1.0, 1.0);
            inputs[i] = FixedPoint::from_f64(normalized);
        }

        // Linear projection
        let mut embedding = [FixedPoint(0); EMBEDDING_DIMENSION];
        for out_idx in 0..EMBEDDING_DIMENSION {
            let mut sum = 0i64;
            for in_idx in 0..INPUT_DIM {
                let w = self.weights.projection[in_idx][out_idx] as i64;
                sum += inputs[in_idx].0 * w;
            }
            let biased = sum + self.weights.bias[out_idx] as i64;
            let dequant = biased as f64 * self.weights.scale;
            embedding[out_idx] = FixedPoint::from_f64(self.silu(dequant));
        }

        embedding
    }

    /// Exact SiLU (off-circuit)
    fn silu(&self, x: f64) -> f64 {
        x / (1.0 + (-x).exp())
    }

    /// Compute delta for transaction
    pub fn compute_transfer_delta(
        &self,
        sender_commitment: &[u8; 32],
        receiver_commitment: &[u8; 32],
        amount: FixedPoint,
    ) -> [FixedPoint; EMBEDDING_DIMENSION] {
        let sender_emb = self.compute_embedding(sender_commitment);
        let receiver_emb = self.compute_embedding(receiver_commitment);

        let mut delta = [FixedPoint(0); EMBEDDING_DIMENSION];
        for i in 0..EMBEDDING_DIMENSION {
            let diff = receiver_emb[i].0 - sender_emb[i].0;
            delta[i] = FixedPoint(diff * amount.0 / FixedPoint::SCALE); // Fixed-point mul
        }

        delta
    }
}

// Update ClientDeltaCircuit to use NeuralEncoderChip
// In synthesize():
// let encoder_chip = NeuralEncoderChip::new(weights.clone(), fixed_point_chip);
// let sender_emb = encoder_chip.encode_commitment(layouter.namespace(|| "sender"),
self.sender_commitment)?;
// let receiver_emb = encoder_chip.encode_commitment(layouter.namespace(|| "receiver"),
self.receiver_commitment)?;
// Then compute delta as before


// src/distilled_transformer.rs
//! DistilledTransformer for Consensus Prediction
//!
//! This implements a tiny distilled transformer for optimistic consensus prediction.
//! - Architecture: 2-layer MLP with attention-like gating (distilled from larger model)
//! - Input: Batch state (aggregated deltas + old embedding)
//! - Output: Predicted next embedding (validator votes on this)
//! - Size: ~1.8MB quantized (int8 weights)
//! - Inference: Fixed-point, deterministic
//!
//! Integration:
//! - Used in ConsensusPredictor for neural voting
//! - Updated via federated learning

use crate::fixed_point::FixedPoint;
use crate::neural_encoder::EMBEDDING_DIMENSION;

const HIDDEN_DIM: usize = 1024;
const NUM_LAYERS: usize = 2;

#[derive(Clone)]
pub struct DistilledTransformer {
    /// Quantized weights per layer
    weights: Vec<QuantizedLayer>,
    scales: Vec<f64>,
}

#[derive(Clone)]
struct QuantizedLayer {
    w1: [[i8; HIDDEN_DIM]; EMBEDDING_DIMENSION], // Input projection
    w2: [[i8; EMBEDDING_DIMENSION]; HIDDEN_DIM], // Output projection
    bias1: [i8; HIDDEN_DIM],
    bias2: [i8; EMBEDDING_DIMENSION],
}

impl DistilledTransformer {
    pub fn new() -> Self {
        // Load quantized weights (from federated update)
        let mut weights = Vec::with_capacity(NUM_LAYERS);
        for _ in 0..NUM_LAYERS {
            weights.push(QuantizedLayer::mock());
        }

        Self {
            weights,
            scales: vec![0.02; NUM_LAYERS], // Per-layer dequant scale
        }
    }

    /// Forward pass: predict next embedding from current state
    pub fn predict(&self, input: &[FixedPoint; EMBEDDING_DIMENSION]) -> [FixedPoint;
EMBEDDING_DIMENSION] {
        let mut hidden = [FixedPoint(0); HIDDEN_DIM];

        let mut current = *input;
        for (layer_idx, layer) in self.weights.iter().enumerate() {
            // Layer 1: linear + gated SiLU
            for h in 0..HIDDEN_DIM {
                let mut sum = 0i64;
                for i in 0..EMBEDDING_DIMENSION {
                    sum += current[i].0 * layer.w1[i][h] as i64;
                }
                let biased = sum + layer.bias1[h] as i64;
                let dequant = biased as f64 * self.scales[layer_idx];
                hidden[h] = FixedPoint::from_f64(Self::silu(dequant));
            }

            // Layer 2: output projection
            let mut output = [FixedPoint(0); EMBEDDING_DIMENSION];
            for o in 0..EMBEDDING_DIMENSION {
                let mut sum = 0i64;
                for h in 0..HIDDEN_DIM {
                    sum += hidden[h].0 * layer.w2[h][o] as i64;
                }
                let biased = sum + layer.bias2[o] as i64;
                let dequant = biased as f64 * self.scales[layer_idx];
                output[o] = FixedPoint::from_f64(dequant); // Linear output
            }

            current = output;
        }

        current
    }

    fn silu(x: f64) -> f64 {
        x / (1.0 + (-x).exp())
    }

    /// Update model from federated average
    pub fn update_from_average(&mut self, averaged_weights: &[u8]) {
        // Deserialize and replace weights
        // In production: secure update with signature
    }
}

impl QuantizedLayer {
    fn mock() -> Self {
        let mut rng = rand::thread_rng();
        let mut w1 = [[0i8; HIDDEN_DIM]; EMBEDDING_DIMENSION];
        let mut w2 = [[0i8; EMBEDDING_DIMENSION]; HIDDEN_DIM];
        let mut bias1 = [0i8; HIDDEN_DIM];
        let mut bias2 = [0i8; EMBEDDING_DIMENSION];

        // Random init
        for row in w1.iter_mut() {
            for w in row.iter_mut() {
                *w = rng.gen_range(-127..=127);
            }
        }
        // Similar for others...

        Self { w1, w2, bias1, bias2 }
    }
}

// In ConsensusPredictor:
// let transformer = DistilledTransformer::new();
// let predicted = transformer.predict(&current_embedding);
// Validators vote on predicted vs actual


// src/federated_learning.rs
```

```rust
//! Federated Learning & Reputation System
//!
//! Complete implementation of NERV's useful-work economy:
//! - Validators submit gradients (model updates)
//! - Performance measured on hold-out set
//! - Shapley values for fair attribution
//! - Reputation boost/decay based on contribution + voting accuracy
//! - Secure aggregation (simple average for now, upgrade to secret sharing)
//!
//! Integration:
//! - Called during batch finalization
//! - Ties into tokenomics emissions (60% to learning)

use crate::distilled_transformer::DistilledTransformer;
use crate::tokenomics::Validator;
use std::collections::HashMap;

pub const HOLD_OUT_SET_SIZE: usize = 100; // Small validation set

#[derive(Clone)]
pub struct GradientUpdate {
    pub validator_id: [u8; 32],
    pub layer_deltas: Vec<Vec<i8>>, // Quantized deltas per layer
    pub performance_score: f64,      // Improvement on hold-out
    pub signature: Vec<u8>,          // Signed update
}

#[derive(Clone)]
pub struct FederatedLearningManager {
    current_model: DistilledTransformer,
    contributions: Vec<GradientUpdate>,
    hold_out_inputs: Vec<[FixedPoint; EMBEDDING_DIMENSION]>,
    hold_out_targets: Vec<[FixedPoint; EMBEDDING_DIMENSION]>,
    reputation_map: HashMap<[u8; 32], f64>,
}

impl FederatedLearningManager {
    pub fn new(initial_model: DistilledTransformer) -> Self {
        // Load hold-out set (fixed, known to all validators)
        let (inputs, targets) = Self::load_hold_out_set();

        Self {
            current_model: initial_model,
            contributions: Vec::new(),
            hold_out_inputs: inputs,
            hold_out_targets: targets,
            reputation_map: HashMap::new(),
        }
    }
```

```rust
/// Validator submits gradient update
pub fn submit_update(&mut self, update: GradientUpdate) -> Result<(), String> {
    // Verify signature
    if !self.verify_update_signature(&update) {
        return Err("Invalid signature".to_string());
    }

    // Measure performance improvement
    let score = self.measure_performance(&update);
    if score < 0.0 {
        return Err("No improvement".to_string());
    }

    self.contributions.push(update);
    Ok(())
}

/// Aggregate updates and compute Shapley values
pub fn aggregate_and_distribute(
    &mut self,
    validators: &mut HashMap<[u8; 32], Validator>,
    learning_rewards: u64,
) -> Result<(), String> {
    if self.contributions.is_empty() {
        return Ok(());
    }

    // Simple average aggregation (upgrade to secure agg)
    let mut averaged_deltas = self.average_gradients();

    // Update global model
    self.current_model.apply_deltas(&averaged_deltas);

    // Compute exact Shapley (or Monte-Carlo approx for scale)
    let shapley_values = self.compute_shapley_values();

    // Distribute rewards + reputation
    let total_score: f64 = shapley_values.iter().map(|s| s.1).sum();
    for (validator_id, score) in shapley_values {
        if let Some(validator) = validators.get_mut(&validator_id) {
            let reward = (learning_rewards as f64 * score / total_score) as u64;
            validator.add_rewards(reward);

            // Reputation boost: +score * base, decay over time
            let reputation_boost = score * 100.0; // Tunable
            validator.update_reputation(true, reputation_boost);
            self.reputation_map.insert(validator_id, validator.reputation_score);
        }
    }

    // Clear contributions for next round
    self.contributions.clear();

    Ok(())
}

fn average_gradients(&self) -> Vec<Vec<i8>> {
    // Average quantized deltas across contributors
    unimplemented!("Secure aggregation stub")
}

fn compute_shapley_values(&self) -> Vec<([u8; 32], f64)> {
    // Monte-Carlo Shapley approximation
    // Sample coalitions, measure marginal contribution
    let mut scores = HashMap::new();
    let n = self.contributions.len();

    for _ in 0..1000 { // Samples
        let mut coalition: Vec<bool> = (0..n).map(|_| rand::random()).collect();
        let base_perf = self.measure_coalition_performance(&coalition);

        for i in 0..n {
            if !coalition[i] {
                coalition[i] = true;
                let marginal = self.measure_coalition_performance(&coalition) - base_perf;
                *scores.entry(self.contributions[i].validator_id).or_insert(0.0) += marginal / n as f64;
                coalition[i] = false;
            }
        }
    }

    scores.into_iter().collect()
}

fn measure_coalition_performance(&self, coalition: &[bool]) -> f64 {
    // Apply subset of updates, measure MSE on hold-out
    unimplemented!("Performance measurement")
}

fn measure_performance(&self, update: &GradientUpdate) -> f64 {
    // Apply single update temporarily, measure improvement
    update.performance_score
}

fn verify_update_signature(&self, update: &GradientUpdate) -> bool {
    // Verify with validator pubkey
    true // Mock
}

fn load_hold_out_set() -> (Vec<[FixedPoint; EMBEDDING_DIMENSION]>, Vec<[FixedPoint;
EMBEDDING_DIMENSION]>) {
    // Fixed synthetic dataset
    (vec![], vec![])
}
}

// In Validator struct (extend):
// pub fn update_reputation(&mut self, honest: bool, contribution_score: f64) {
//     if honest {
//         self.reputation_score += contribution_score;
//     } else {
//         self.reputation_score *= 0.9; // Decay on dishonest
//     }
//     self.reputation_score = self.reputation_score.clamp(0.0, 10000.0);
// }
```