Report: Construction and Optimization of Spiking Neural Networks (SNNs)

1. Baseline SNN Network Construction

We implemented a simple Spiking Neural Network (SNN) using the Leaky Integrate-and-Fire (LIF) neuron model, trained on the MNIST dataset. The architecture and implementation were carried out in PyTorch with the help of the SpikingJelly framework.

1.1 Network Architecture

- **Input Layer:** 784 Poisson-encoded neurons (28x28 pixel grayscale images)
- **Hidden Layer:** 400 LIF neurons
- **Output Layer:** 10 LIF neurons
- **Time Steps:** 100 ms per image
- **Neuron Model:** LIF with membrane decay and reset
- **Encoding:** Poisson rate coding

1.2 Training Details

- **Optimizer:** Adam
- **Loss Function:** Cross-entropy (based on spike count)
- **Learning Rate Scheduler:** StepLR
- **Gradient Clipping:** Enabled
- **Training Epochs:** 30
- **Accuracy Achieved:** ~97% on MNIST test set

1.3 Code Snapshot

The training routine is implemented as follows:

```
spike_input = encoder.encode(images)
outputs, hidden_spikes = net(spike_input)
loss = criterion(outputs, labels) + firing_rate_regularization(hidden_s
pikes)
loss.backward()
torch.nn.utils.clip_grad_norm_(net.parameters(), 1.0)
optimizer.step()
```

2. Optimized SNN with Dynamic Pruning

To improve the network's efficiency, we integrated a dynamic pruning strategy during training. This pruning focuses on eliminating unimportant synaptic connections by monitoring spiking activity and weight magnitudes.

2.1 Pruning Strategy

- **Criteria:** Low activity and small absolute weight
- **Granularity:** Neuron-wise and connection-wise
- **Trigger:** After every 10 epochs
- **Mask Update:** Pruned weights are set to zero and excluded from updates

2.2 Results

- **Parameter Reduction:** ~40%
- **Accuracy:** ~96.3% (minor drop)
- **Sparsity:** Enhanced post-training weight sparsity
- **Energy Implication:** Less MAC operations, more suitable for FPGA deployment

2.3 Code Snapshot

The following code is used for activity-based pruning:

```
avg_fr = total_fr / len(dataloader)
prune_idx = (avg_fr < threshold).nonzero(as_tuple=True)[0]
net.fc1.weight[:, prune_idx] = 0
net.fc2.weight[:, prune_idx] = 0
```

3. Auto-Tuned SNN with Adaptive Hyperparameters

In addition to manual pruning, we implemented an auto-tuned training pipeline using a hyperparameter optimization library. The system dynamically adjusted learning rate, regularization strength, and dropout rate to improve convergence and robustness.

3.1 Tuning Framework

- **Tool:** Optuna (Tree-structured Parzen Estimator)
- **Search Space:**
    - Learning rate: [1e-4, 1e-2]
    - Dropout rate: [0.1, 0.5]
    - Firing rate penalty: [0.001, 0.1]

3.2 Best Configuration Found

- Learning rate: 0.0018

- Dropout rate: 0.25
- Firing rate penalty (lambda_reg): 0.02

3.3 Performance

- **Accuracy:** 96.7%
- **Convergence Speed:** ~20% faster than baseline
- **Sparsity:** Similar to pruned model (~40%)
- **Remarks:** Stable training with fewer overfitting signs

4. Additional Optimizations and Comparisons

We compared several optimization techniques applied individually and in combination:

Optimization Technique	Accuracy (%)	Model Size Reduction	Notes
Baseline	97.0	1x	High accuracy, full connectivity
+ Dynamic Pruning	96.3	~0.6x	Reduced compute and parameters
+ Auto Tuning	96.7	~0.6x	Tuned hyperparameters, stable
+ Dropout (p=0.2)	96.8	1x	Regularizes firing patterns
+ Batch Normalization	96.9	1x	Stabilizes training
+ Firing Rate Regularization	96.6	1x	Reduces excessive spike activity
+ All Combined	96.1	~0.55x	Best trade-off for hardware use

4.1 Firing Rate Regularization

We added a regularization term to the loss function to penalize neurons that fire excessively:

```
loss = cross_entropy + lambda_reg * total_spike_rate
```

This encourages sparser spike activity, which is energy-efficient for neuromorphic hardware.

4.2 BatchNorm and Dropout

- BatchNorm is applied between layers to normalize the input spike tensor across the batch.
- Dropout with a rate of 0.2–0.25 is used during training to prevent overfitting.

5. Summary and Conclusion

Through a combination of dynamic pruning, auto-tuning, and software-level regularization techniques, we were able to construct a high-performing and resource-efficient SNN model. The optimized versions retain above 96% accuracy while offering significant parameter sparsity and lower computational load, making them highly suitable for FPGA or other neuromorphic hardware deployment.

Next steps will involve quantization-aware training, latency evaluation, and full fixed-point conversion for RTL synthesis.

Appendix: Code Resources

This report is backed by the following implementations:

- **with_prune.py**: Complete dynamic pruning + training pipeline
- **test.py**: Baseline and auto-tuned network with regularization
- **para.py**: Quantized model metrics (params, sparsity, MACs, INT8 size, latency)

These scripts demonstrate advanced SNN optimization and are instrumented with TensorBoard for visualization.

```
# Poisson encoder
class PoissonEncoder:
    def __init__(self, T, scale=0.7):
        self.T = T
        self.scale = scale
    def encode(self, images):
        B, C, H, W = images.shape
        images = images.view(B, -1).unsqueeze(0).repeat(self.T, 1, 1)
        return (torch.rand_like(images) < images * self.scale).float()

# Training with dynamic pruning
multi_round_pruning_finetune(net, train_loader, test_loader, encoder,
    device=device, max_rounds=5, min_neurons=50,
    threshold=0.01, finetune_epochs=5)

# Evaluation metrics (MACs, sparsity, latency, size)
macs = compute_macs(net)
sparsity = compute_params_and_sparsity(net)
gpu_latency = measure_gpu_latency(net)
size_int8 = estimate_quantized_size(net, 8)
```