Notebook

November 26, 2025

1 Attn-KNN: Learned Attention over Neighbors

1.1 Core Claim

Learned attention over neighbors improves calibration and robustness versus uniform and distance-weighted kNN, at modest compute.

1.2 Experiment Structure

1. **Baselines**: Uniform kNN, Distance-weighted kNN, CNN classifier (upper bound)
2. **Robustness**: Label noise (10-30%), Long-tailed imbalance (CIFAR-LT)
3. **k-Sweep**: Accuracy/F1/NLL/ECE across k in {1,3,5,10,20,50}
4. **Efficiency**: Latency/memory profiling, FAISS index comparison
5. **Theory Link**: Error vs k, embedding norm stability

[ ]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
os.environ['OMP_NUM_THREADS'] = '1'
os.environ['MKL_NUM_THREADS'] = '1'

import json
import time
import random
from pathlib import Path
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass, field

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Subset
from torchvision import transforms, datasets, models
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, f1_score, log_loss
from tqdm.auto import tqdm
import faiss
```

```
device = torch.device('cuda' if torch.cuda.is_available() else ('mps' if torch.
↪backends.mps.is_available() else 'cpu'))
print(f'Device: {device}')
```

```
Device: mps
```

[ ]:
```
@dataclass
class Config:
    seed: int = 42
    embed_dim: int = 256
    num_heads: int = 4
    batch_size: int = 128
    epochs: int = 50
    lr: float = 1e-4
    k_values: List[int] = field(default_factory=lambda: [1, 3, 5, 10, 20, 50])
    noise_rates: List[float] = field(default_factory=lambda: [0.0, 0.1, 0.2, 0.
↪3])
    imbalance_ratios: List[float] = field(default_factory=lambda: [1.0, 0.1, 0.
↪01])
    mixup_alpha: float = 0.2
    label_smoothing: float = 0.1

cfg = Config()
DATA_ROOT = Path('/Users/taher/Projects/attn_knn_repo/data')
RESULTS_DIR = Path('results')
RESULTS_DIR.mkdir(exist_ok=True)

def set_seed(seed: int) -> None:
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)

set_seed(cfg.seed)
```

1.3 Models

[ ]:
```
class Embedder(nn.Module):
    """ResNet18 embedder with ImageNet pretrained weights."""
    def __init__(self, embed_dim: int = 256) -> None:
        super().__init__()
        backbone = models.resnet18(weights=models.ResNet18_Weights.
↪IMAGENET1K_V1)
        backbone.conv1 = nn.Conv2d(3, 64, 3, 1, 1, bias=False)
        backbone.maxpool = nn.Identity()
        in_f = backbone.fc.in_features
        backbone.fc = nn.Identity()
        self.backbone = backbone
        self.proj = nn.Sequential(nn.Linear(in_f, 512), nn.BatchNorm1d(512), nn.
 ↪ReLU(), nn.Linear(512, embed_dim))
        self.embed_dim = embed_dim

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return F.normalize(self.proj(self.backbone(x)), dim=1)


class MultiHeadNeighborAttention(nn.Module):
    """Multi-head attention over k neighbors with distance-aware bias."""
    def __init__(self, embed_dim: int = 256, num_heads: int = 4) -> None:
        super().__init__()
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = self.head_dim ** -0.5
        self.q_proj = nn.Linear(embed_dim, embed_dim)
        self.k_proj = nn.Linear(embed_dim, embed_dim)
        self.log_tau = nn.Parameter(torch.zeros(num_heads))
        self.dist_bias = nn.Sequential(nn.Linear(1, 32), nn.ReLU(), nn.
 ↪Linear(32, num_heads))

    def forward(self, q: torch.Tensor, neighbors: torch.Tensor, dists:␣
 ↪Optional[torch.Tensor] = None) -> torch.Tensor:
        B, K, D = neighbors.shape
        H, d = self.num_heads, self.head_dim
        qp = self.q_proj(q).view(B, 1, H, d).transpose(1, 2)
        kp = self.k_proj(neighbors).view(B, K, H, d).transpose(1, 2)
        tau = torch.exp(self.log_tau).clamp(0.01, 10).view(1, H, 1, 1)
        attn = (qp @ kp.transpose(-2, -1)) * self.scale / tau
        if dists is not None:
            bias = self.dist_bias(dists.unsqueeze(-1)).permute(0, 2, 1).
 ↪unsqueeze(2)
            attn = attn + bias
        return F.softmax(attn, dim=-1).squeeze(2).mean(dim=1)  # (B, K)


class AttnKNN(nn.Module):
    """Full Attn-KNN model."""
    def __init__(self, embed_dim: int = 256, num_heads: int = 4, num_classes:␣
 ↪int = 10) -> None:
        super().__init__()
        self.embedder = Embedder(embed_dim)
        self.attention = MultiHeadNeighborAttention(embed_dim, num_heads)
        self.classifier = nn.Sequential(nn.Linear(embed_dim, embed_dim), nn.
 ↪ReLU(), nn.Dropout(0.2), nn.Linear(embed_dim, num_classes))
        self.num_classes = num_classes
    def forward(self, x: torch.Tensor, neigh_emb: torch.Tensor, dists: torch.
↪Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        q = self.embedder(x)
        attn = self.attention(q, neigh_emb, dists)
        logits = self.classifier(q)
        return logits, attn


class CNNClassifier(nn.Module):
    """CNN baseline (accuracy upper bound)."""
    def __init__(self, num_classes: int = 10) -> None:
        super().__init__()
        backbone = models.resnet18(weights=models.ResNet18_Weights.
↪IMAGENET1K_V1)
        backbone.conv1 = nn.Conv2d(3, 64, 3, 1, 1, bias=False)
        backbone.maxpool = nn.Identity()
        backbone.fc = nn.Linear(backbone.fc.in_features, num_classes)
        self.model = backbone

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.model(x)


print('Models defined')
```

```
Models defined
```

1.4 Memory Bank and kNN Methods

```
[ ]: class MemoryBank:
         def __init__(self, dim: int, index_type: str = 'L2') -> None:
             self.dim = dim
             self.index_type = index_type
             self.index: Optional[faiss.Index] = None
             self.embeddings: Optional[np.ndarray] = None
             self.labels: Optional[np.ndarray] = None

         def build(self, emb: np.ndarray, lab: np.ndarray) -> None:
             self.embeddings = emb.astype('float32')
             self.labels = lab.astype('int64')
             if self.index_type == 'IP':
                 self.index = faiss.IndexFlatIP(self.dim)
             elif self.index_type == 'HNSW':
                 self.index = faiss.IndexHNSWFlat(self.dim, 32)
             else:
                 self.index = faiss.IndexFlatL2(self.dim)
```

```
        self.index.add(self.embeddings)

    def search(self, q: np.ndarray, k: int) -> Tuple[np.ndarray, np.ndarray, np.
↪ndarray]:
        d, i = self.index.search(q.astype('float32'), k)
        return d, i, self.labels[i]

    def get_emb(self, idx: np.ndarray) -> np.ndarray:
        return self.embeddings[idx]


def probs_uniform(neigh_lab: np.ndarray, nc: int, k: int) -> np.ndarray:
    """Uniform kNN: 1/k weight per neighbor."""
    B = neigh_lab.shape[0]
    p = np.zeros((B, nc), dtype=np.float32)
    np.add.at(p, (np.arange(B)[:, None], neigh_lab), 1.0 / k)
    return p


def probs_distance(neigh_lab: np.ndarray, dists: np.ndarray, nc: int, tau:␣
↪float = 1.0) -> np.ndarray:
    """Distance-weighted kNN: softmax(-distance/tau)."""
    w = np.exp(-dists / tau)
    w = w / w.sum(axis=1, keepdims=True)
    B = neigh_lab.shape[0]
    p = np.zeros((B, nc), dtype=np.float32)
    np.add.at(p, (np.arange(B)[:, None], neigh_lab), w)
    return p


def probs_attention(neigh_lab: np.ndarray, weights: np.ndarray, nc: int) -> np.
↪ndarray:
    """Attention-weighted kNN."""
    B = neigh_lab.shape[0]
    p = np.zeros((B, nc), dtype=np.float32)
    np.add.at(p, (np.arange(B)[:, None], neigh_lab), weights)
    return p


print('Memory bank and kNN methods defined')
```

```
Memory bank and kNN methods defined
```

1.5 Metrics

[ ]:
```
def ece(probs: np.ndarray, labels: np.ndarray, n_bins: int = 15) -> float:
    """Expected Calibration Error."""
    confs = probs.max(axis=1)
    preds = probs.argmax(axis=1)
    bins = np.linspace(0, 1, n_bins + 1)
    e = 0.0
    for i in range(n_bins):
        m = (confs > bins[i]) & (confs <= bins[i + 1])
        if m.sum() > 0:
            acc = (preds[m] == labels[m]).mean()
            conf = confs[m].mean()
            e += (m.sum() / len(labels)) * abs(acc - conf)
    return float(e)


def compute_metrics(probs: np.ndarray, y: np.ndarray) -> Dict[str, float]:
    pred = probs.argmax(axis=1)
    pc = np.clip(probs, 1e-9, 1.0)
    pc = pc / pc.sum(axis=1, keepdims=True)
    return {
        'accuracy': float(accuracy_score(y, pred)),
        'f1_macro': float(f1_score(y, pred, average='macro', zero_division=0)),
        'nll': float(log_loss(y, pc)),
        'ece': ece(probs, y)
    }


print('Metrics defined')
```

```
Metrics defined
```

1.6 Robustness Utilities

[ ]:
```
def inject_noise(labels: np.ndarray, rate: float, nc: int) -> np.ndarray:
    """Symmetric label noise."""
    if rate <= 0:
        return labels.copy()
    noisy = labels.copy()
    n_flip = int(rate * len(labels))
    idx = np.random.choice(len(labels), n_flip, replace=False)
    for i in idx:
        noisy[i] = np.random.choice([c for c in range(nc) if c != labels[i]])
    return noisy


def create_imbalanced(dataset, ratio: float, nc: int) -> Subset:
    """Long-tailed imbalanced subset (exponential decay)."""
    if ratio >= 1.0:
        return dataset
    labels = np.array(dataset.targets)
    max_n = np.bincount(labels).max()
    indices = []
    for c in range(nc):
        c_idx = np.where(labels == c)[0]
        factor = ratio ** (c / (nc - 1))
        n_keep = max(1, int(max_n * factor))
        indices.extend(np.random.choice(c_idx, min(n_keep, len(c_idx)),␣
↪replace=False).tolist())
    return Subset(dataset, indices)


print('Robustness utilities defined')
```

```
Robustness utilities defined
```

1.7 Training

```
[ ]: def mixup(x: torch.Tensor, y: torch.Tensor, alpha: float) -> Tuple[torch.
↪Tensor, torch.Tensor, torch.Tensor, float]:
    lam = np.random.beta(alpha, alpha) if alpha > 0 else 1.0
    idx = torch.randperm(x.size(0), device=x.device)
    return lam * x + (1 - lam) * x[idx], y, y[idx], lam


def train_epoch(model: AttnKNN, memory: MemoryBank, loader: DataLoader, opt:␣
↪torch.optim.Optimizer, k: int = 20) -> float:
    model.train()
    criterion = nn.CrossEntropyLoss(label_smoothing=cfg.label_smoothing)
    total = 0.0
    for xb, yb in tqdm(loader, leave=False, desc='Train'):
        xb, yb = xb.to(device), yb.to(device)
        xm, ya, yb_m, lam = mixup(xb, yb, cfg.mixup_alpha)
        opt.zero_grad()
        with torch.no_grad():
            q = model.embedder(xm)
            if device.type == 'mps': torch.mps.synchronize()
        d, i, nl = memory.search(q.cpu().numpy(), k)
        ne = torch.from_numpy(memory.get_emb(i)).to(device)
        dt = torch.from_numpy(d).to(device)
        logits, attn = model(xm, ne, dt)
        loss = lam * criterion(logits, ya) + (1 - lam) * criterion(logits, yb_m)
        ent = -(attn * torch.log(attn + 1e-9)).sum(1).mean()
        (loss - 0.01 * ent).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        opt.step()
        total += loss.item()
    return total / len(loader)


def build_memory(model: AttnKNN, loader: DataLoader) -> MemoryBank:
    model.eval()
    embs, labs = [], []
    with torch.no_grad():
        for xb, yb in tqdm(loader, leave=False, desc='Memory'):
            z = model.embedder(xb.to(device))
            if device.type == 'mps': torch.mps.synchronize()
            embs.append(z.cpu().numpy())
            labs.append(yb.numpy())
    mem = MemoryBank(model.embedder.embed_dim)
    mem.build(np.vstack(embs), np.concatenate(labs))
    return mem


def train_cnn(model: CNNClassifier, loader: DataLoader, epochs: int = 50) ->␣
↪CNNClassifier:
    model.to(device).train()
    opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, epochs)
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
    for ep in range(epochs):
        for xb, yb in tqdm(loader, leave=False, desc=f'CNN {ep+1}'):
            xb, yb = xb.to(device), yb.to(device)
            opt.zero_grad()
            loss = criterion(model(xb), yb)
            loss.backward()
            opt.step()
        sched.step()
    return model


print('Training functions defined')
```

```
Training functions defined
```

1.8 Evaluation

[ ]:
```
def evaluate_knn(model: AttnKNN, memory: MemoryBank, loader: DataLoader, k:␣
↪int, method: str = 'attention', tau: float = 1.0) -> Tuple[Dict, np.ndarray,␣
↪np.ndarray]:
    model.eval()
    nc = model.num_classes
    all_p, all_y = [], []
    with torch.no_grad():
        for xb, yb in tqdm(loader, leave=False, desc=f'Eval {method}'):
            q = model.embedder(xb.to(device))
            if device.type == 'mps': torch.mps.synchronize()
            d, i, nl = memory.search(q.cpu().numpy(), k)
            if method == 'uniform':
                p = probs_uniform(nl, nc, k)
            elif method == 'distance':
                p = probs_distance(nl, d, nc, tau)
            else:
                ne = torch.from_numpy(memory.get_emb(i)).to(device)
                dt = torch.from_numpy(d).to(device)
                _, attn = model(xb.to(device), ne, dt)
                p = probs_attention(nl, attn.cpu().numpy(), nc)
            all_p.append(p)
            all_y.append(yb.numpy())
    probs = np.vstack(all_p)
    y = np.concatenate(all_y)
    return compute_metrics(probs, y), probs, y


def evaluate_cnn(model: CNNClassifier, loader: DataLoader) -> Dict:
    model.eval()
    all_p, all_y = [], []
    with torch.no_grad():
        for xb, yb in loader:
            logits = model(xb.to(device))
            all_p.append(F.softmax(logits, dim=1).cpu().numpy())
            all_y.append(yb.numpy())
    return compute_metrics(np.vstack(all_p), np.concatenate(all_y))


print('Evaluation functions defined')
```

```
Evaluation functions defined
```

1.9 Visualization

[ ]:
```
def plot_reliability(probs: np.ndarray, y: np.ndarray, title: str, path:␣
↪Optional[str] = None) -> None:
    confs = probs.max(axis=1)
    preds = probs.argmax(axis=1)
    correct = (preds == y).astype(float)
    bins = np.linspace(0, 1, 11)
    accs = [correct[(confs > bins[i]) & (confs <= bins[i+1])].mean() if ((confs␣
 ↪> bins[i]) & (confs <= bins[i+1])).sum() > 0 else np.nan for i in range(10)]
    fig, ax = plt.subplots(figsize=(6, 5))
    ax.bar(0.5*(bins[:-1]+bins[1:]), accs, width=0.08, alpha=0.7,␣
 ↪color='steelblue', edgecolor='black')
    ax.plot([0,1], [0,1], 'k--')
    ax.set_xlabel('Confidence'); ax.set_ylabel('Accuracy'); ax.set_title(title)
    ax.set_xlim(0,1); ax.set_ylim(0,1)
    if path: plt.savefig(path, dpi=150, bbox_inches='tight')
    plt.show()


def plot_k_sweep(results: Dict, metric: str, path: Optional[str] = None) ->␣
 ↪None:
    fig, ax = plt.subplots(figsize=(8, 5))
    colors = {'uniform': 'gray', 'distance': 'blue', 'attention': 'red'}
    for method in ['uniform', 'distance', 'attention']:
        ks = sorted([int(k) for k in results.keys()])
        vals = [results[str(k)][method][metric] for k in ks]
        ax.plot(ks, vals, 'o-', color=colors[method], label=method, linewidth=2)
    ax.set_xlabel('k'); ax.set_ylabel(metric.upper()); ax.legend(); ax.
 ↪grid(alpha=0.3)
    ax.set_title(f'{metric.upper()} vs k')
    if path: plt.savefig(path, dpi=150, bbox_inches='tight')
    plt.show()


def plot_noise_robustness(results: Dict, metric: str, path: Optional[str] =␣
 ↪None) -> None:
    fig, ax = plt.subplots(figsize=(8, 5))
    colors = {'uniform': 'gray', 'distance': 'blue', 'attention': 'red'}
    for method in ['uniform', 'distance', 'attention']:
        rates = sorted([float(r) for r in results.keys()])
        vals = [results[str(r)][method][metric] for r in rates]
        ax.plot([r*100 for r in rates], vals, 'o-', color=colors[method],␣
 ↪label=method, linewidth=2)
    ax.set_xlabel('Label Noise (%)'); ax.set_ylabel(metric.upper()); ax.
 ↪legend(); ax.grid(alpha=0.3)
    ax.set_title(f'{metric.upper()} vs Label Noise')
    if path: plt.savefig(path, dpi=150, bbox_inches='tight')
    plt.show()


def to_latex(results: Dict, caption: str, label: str) -> str:
    lines = ['\\begin{table}[h]', '  \\centering', '  \\begin{tabular}{lrrrr}',␣
 ↪'    \\toprule',
            '    Method & Acc & F1 & NLL & ECE \\\\', '    \\midrule']
    for n, m in results.items():
        lines.append(f"    {n} & {m['accuracy']*100:.2f} & {m['f1_macro']*100:.
↪2f} & {m['nll']:.3f} & {m['ece']:.4f} \\\\")
    lines += ['    \\bottomrule', '  \\end{tabular}', f' ␣
↪\\caption{{{caption}}}', f'  \\label{{{label}}}', '\\end{table}']
    return '\n'.join(lines)


print('Visualization functions defined')
```

```
Visualization functions defined
```

1.10 Efficiency Profiling

```
[ ]: def profile_index(n_samples: int = 50000, dim: int = 256, k: int = 10,␣
↪n_queries: int = 1000) -> Dict:
    """Profile FAISS index types."""
    data = np.random.randn(n_samples, dim).astype('float32')
    queries = np.random.randn(n_queries, dim).astype('float32')
    results = {}

    for idx_type in ['L2', 'IP', 'HNSW']:
        if idx_type == 'L2':
            index = faiss.IndexFlatL2(dim)
        elif idx_type == 'IP':
            index = faiss.IndexFlatIP(dim)
        else:
            index = faiss.IndexHNSWFlat(dim, 32)

        t0 = time.time()
        index.add(data)
        build_time = time.time() - t0

        t0 = time.time()
        index.search(queries, k)
        search_time = time.time() - t0

        results[idx_type] = {
            'build_time': build_time,
            'search_time': search_time,
            'search_per_query_ms': (search_time / n_queries) * 1000
        }

    return results
```

```
print('Efficiency profiling defined')
```

```
Efficiency profiling defined
```

1.11 Data Setup

[ ]:
```
print('='*70)
print('CIFAR-10 EXPERIMENT')
print('='*70)

norm = transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
train_tf = transforms.Compose([transforms.RandomCrop(32, 4), transforms.
 ↪RandomHorizontalFlip(),
                                transforms.ColorJitter(0.2, 0.2, 0.2),␣
 ↪transforms.ToTensor(), norm])
test_tf = transforms.Compose([transforms.ToTensor(), norm])

data_root = str(DATA_ROOT) if DATA_ROOT.exists() else './data'
train_ds = datasets.CIFAR10(data_root, True, train_tf, download=True)
test_ds = datasets.CIFAR10(data_root, False, test_tf, download=True)
train_ds_clean = datasets.CIFAR10(data_root, True, test_tf)

train_loader = DataLoader(train_ds, cfg.batch_size, True, num_workers=2)
train_loader_clean = DataLoader(train_ds_clean, cfg.batch_size, False,␣
 ↪num_workers=2)
test_loader = DataLoader(test_ds, cfg.batch_size, False, num_workers=2)

NC = 10
print(f'Train: {len(train_ds)}, Test: {len(test_ds)}')
```

```
======================================================================
CIFAR-10 EXPERIMENT
======================================================================
Train: 50000, Test: 10000
```

1.12 Train Attn-KNN

[ ]:
```
print('\n--- Training Attn-KNN ---')
model = AttnKNN(cfg.embed_dim, cfg.num_heads, NC).to(device)
memory = build_memory(model, train_loader_clean)

opt = torch.optim.AdamW([
    {'params': model.embedder.backbone.parameters(), 'lr': cfg.lr * 0.1},
    {'params': model.embedder.proj.parameters(), 'lr': cfg.lr},
    {'params': model.attention.parameters(), 'lr': cfg.lr},
    {'params': model.classifier.parameters(), 'lr': cfg.lr}
], weight_decay=1e-4)
sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, cfg.epochs)
```

```
best_acc = 0.0
for ep in range(cfg.epochs):
    loss = train_epoch(model, memory, train_loader, opt)
    sched.step()
    if (ep + 1) % 5 == 0:
        memory = build_memory(model, train_loader_clean)
        m, _, _ = evaluate_knn(model, memory, test_loader, 10, 'attention')
        print(f'Epoch {ep+1}: Loss={loss:.4f}, Acc={m["accuracy"]*100:.2f}%')
        if m['accuracy'] > best_acc:
            best_acc = m['accuracy']
            torch.save(model.state_dict(), RESULTS_DIR / 'best_model.pt')

model.load_state_dict(torch.load(RESULTS_DIR / 'best_model.pt'))
memory = build_memory(model, train_loader_clean)
print(f'Best accuracy: {best_acc*100:.2f}%')
```

```
--- Training Attn-KNN ---

Memory:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Memory:   0%|          | 0/391 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Epoch 5: Loss=1.3002, Acc=78.26%

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Memory:   0%|          | 0/391 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Epoch 10: Loss=1.1263, Acc=84.45%

Train:   0%|          | 0/391 [00:01<?, ?it/s]

Train:   0%|          | 0/391 [00:01<?, ?it/s]
```

```
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 15: Loss=1.0574, Acc=86.83%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:02<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 20: Loss=1.0194, Acc=88.12%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 25: Loss=0.9851, Acc=89.10%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 30: Loss=0.9724, Acc=89.19%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
```

```
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 35: Loss=0.9427, Acc=89.31%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:02<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 40: Loss=0.9828, Acc=89.59%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 45: Loss=0.9379, Acc=89.62%
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Train:   0%|          | 0/391 [00:01<?, ?it/s]
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
Epoch 50: Loss=0.9398, Acc=89.70%
Memory:   0%|          | 0/391 [00:01<?, ?it/s]
Best accuracy: 89.70%
```

1.13 Train CNN Baseline (Upper Bound)

```
[ ]: print('\n--- Training CNN Baseline ---')
     cnn = train_cnn(CNNClassifier(NC), train_loader, epochs=50)
     cnn_metrics = evaluate_cnn(cnn, test_loader)
     print(f'CNN Accuracy: {cnn_metrics["accuracy"]*100:.2f}%')
```

```
--- Training CNN Baseline ---
CNN 1:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 2:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 3:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 4:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 5:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 6:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 7:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 8:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 9:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 10:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 11:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 12:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 13:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 14:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 15:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 16:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 17:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 18:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 19:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 20:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 21:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 22:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 23:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 24:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 25:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 26:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 27:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 28:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 29:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 30:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 31:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 32:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 33:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 34:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 35:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 36:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 37:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 38:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 39:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 40:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 41:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 42:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 43:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 44:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 45:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 46:   0%|          | 0/391 [00:02<?, ?it/s]
CNN 47:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 48:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 49:   0%|          | 0/391 [00:01<?, ?it/s]
CNN 50:   0%|          | 0/391 [00:01<?, ?it/s]
CNN Accuracy: 94.94%
```

1.14 Main Results

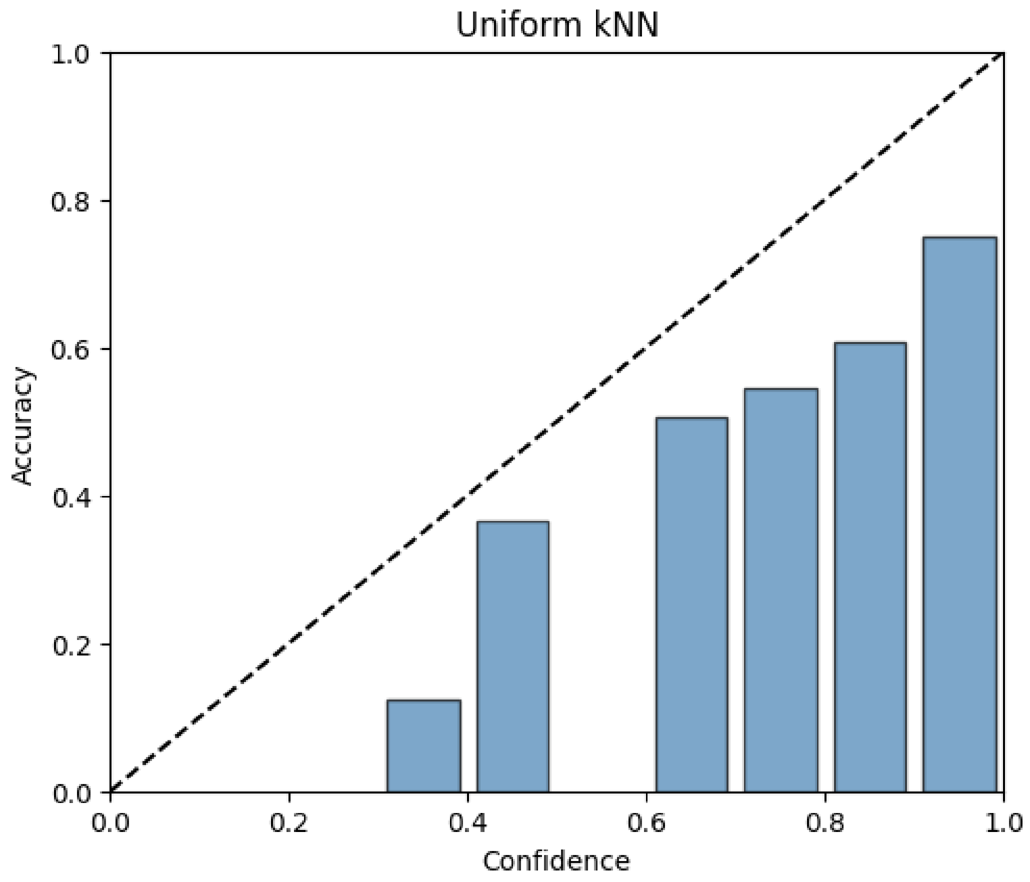
[ ]:
```python
print('\n--- Main Results (k=10) ---')
K = 10
main_results = {}

m_uni, p_uni, y_true = evaluate_knn(model, memory, test_loader, K, 'uniform')
main_results['Uniform kNN'] = m_uni
```

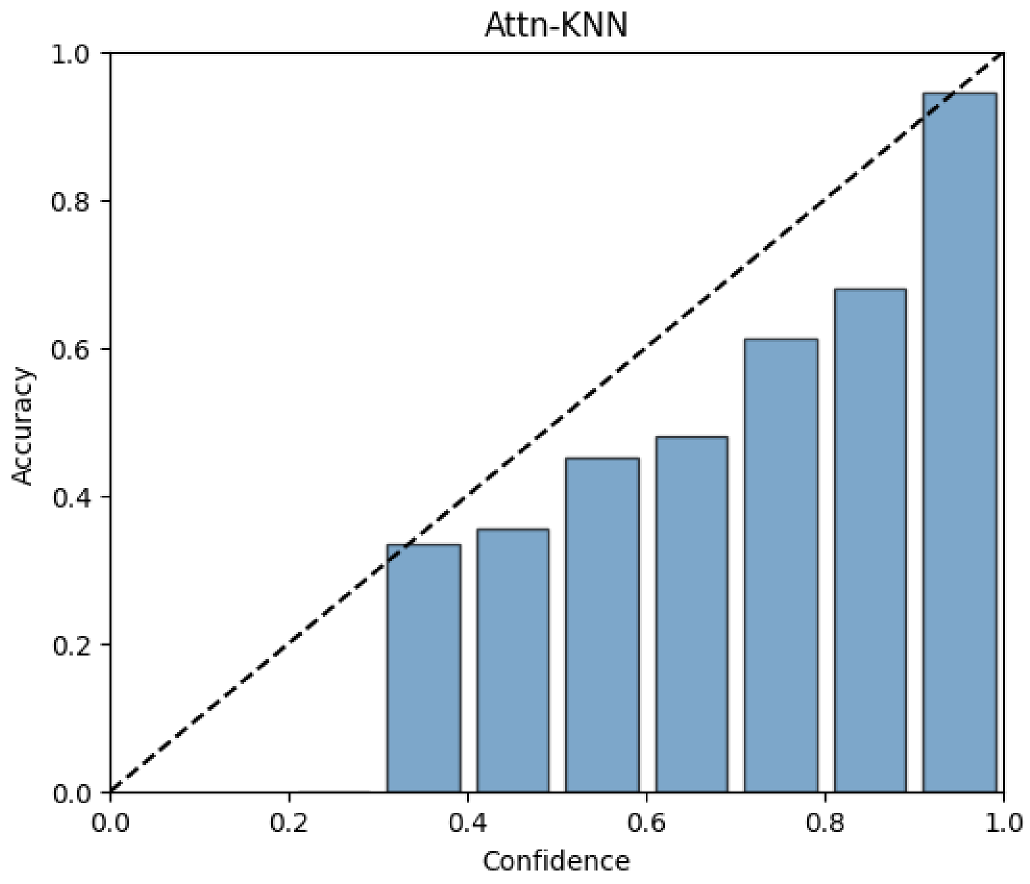
```
m_dist, p_dist, _ = evaluate_knn(model, memory, test_loader, K, 'distance',␣
  ↪tau=1.0)
main_results['Distance kNN'] = m_dist

m_attn, p_attn, _ = evaluate_knn(model, memory, test_loader, K, 'attention')
main_results['Attn-KNN (Ours)'] = m_attn

main_results['CNN (Upper Bound)'] = cnn_metrics

print('\n' + '='*70)
print(f'{"Method":<20} {"Acc":>8} {"F1":>8} {"NLL":>8} {"ECE":>8}')
print('-'*70)
for n, m in main_results.items():
    print(f'{n:<20} {m["accuracy"]*100:>7.2f}% {m["f1_macro"]*100:>7.2f}%␣
  ↪{m["nll"]:>8.4f} {m["ece"]:>8.4f}')
print('='*70)
```

```
--- Main Results (k=10) ---

Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]


======================================================================
Method                    Acc       F1      NLL      ECE
----------------------------------------------------------------------
Uniform kNN            89.69%   89.66%   0.9636   0.0243
Distance kNN           89.70%   89.66%   0.9636   0.0566
Attn-KNN (Ours)        89.70%   89.66%   0.9636   0.0578
CNN (Upper Bound)      94.94%   94.93%   0.2577   0.0718
======================================================================
```

```
[ ]: print('\n--- Reliability Diagrams ---')
plot_reliability(p_uni, y_true, 'Uniform kNN', str(RESULTS_DIR /␣
  ↪'reliability_uniform.png'))
plot_reliability(p_attn, y_true, 'Attn-KNN', str(RESULTS_DIR /␣
  ↪'reliability_attn.png'))
```

```
--- Reliability Diagrams ---
```


Uniform kNN
Accuracy
Confidence
0.0
0.2
0.4
0.6
0.8
1.0

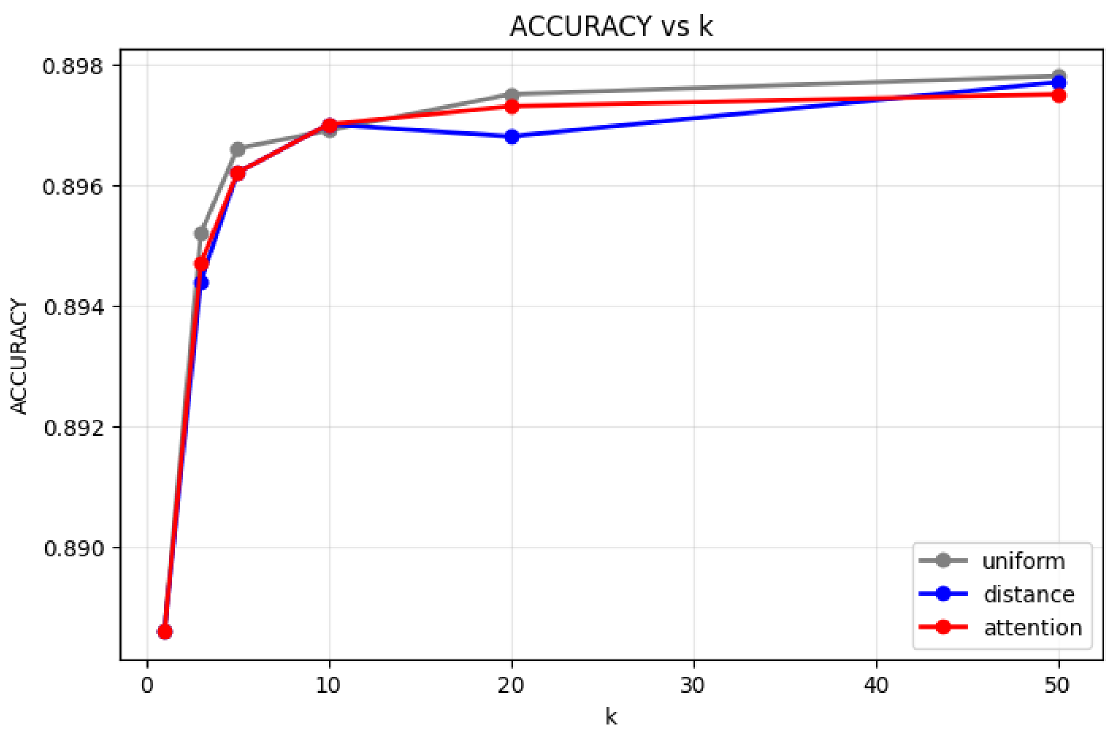


1.15 k-Sweep Experiment

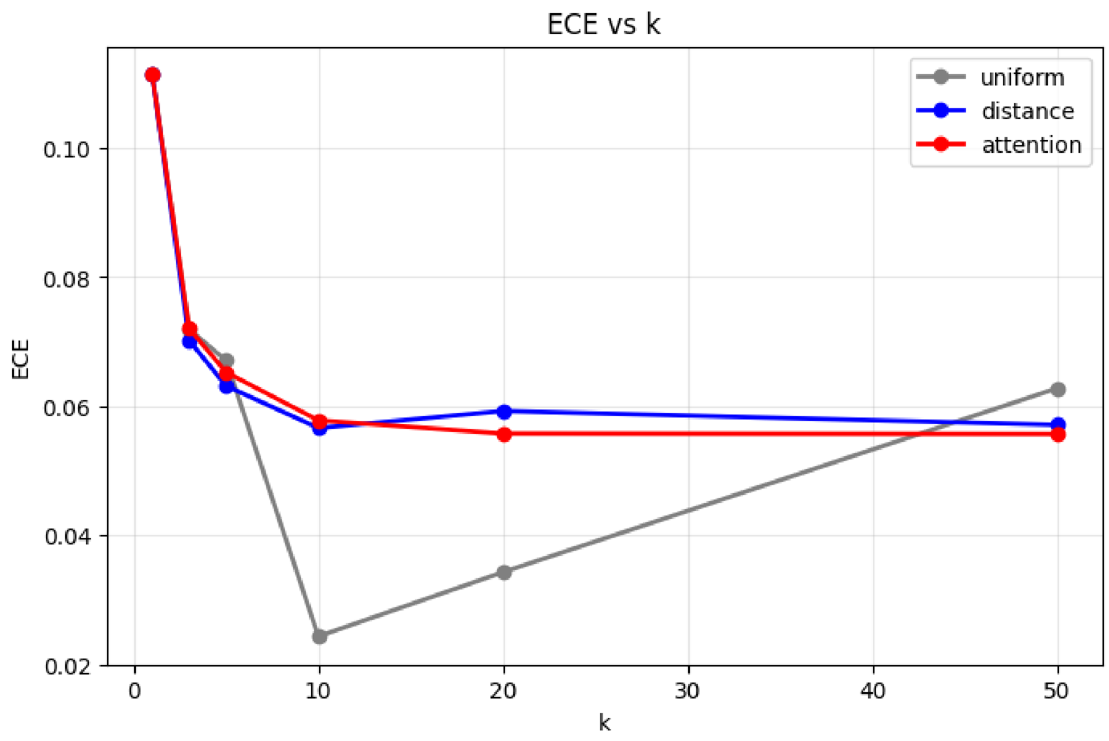
```
[ ]: print('\n--- k-Sweep ---')
     k_results = {}
     for k in cfg.k_values:
         print(f'k={k}')
         k_results[str(k)] = {
             'uniform': evaluate_knn(model, memory, test_loader, k, 'uniform')[0],
             'distance': evaluate_knn(model, memory, test_loader, k, 'distance')[0],
             'attention': evaluate_knn(model, memory, test_loader, k, 'attention')[0]
         }

     plot_k_sweep(k_results, 'accuracy', str(RESULTS_DIR / 'k_sweep_accuracy.png'))
     plot_k_sweep(k_results, 'ece', str(RESULTS_DIR / 'k_sweep_ece.png'))
     plot_k_sweep(k_results, 'nll', str(RESULTS_DIR / 'k_sweep_nll.png'))
```

```
--- k-Sweep ---
k=1
```

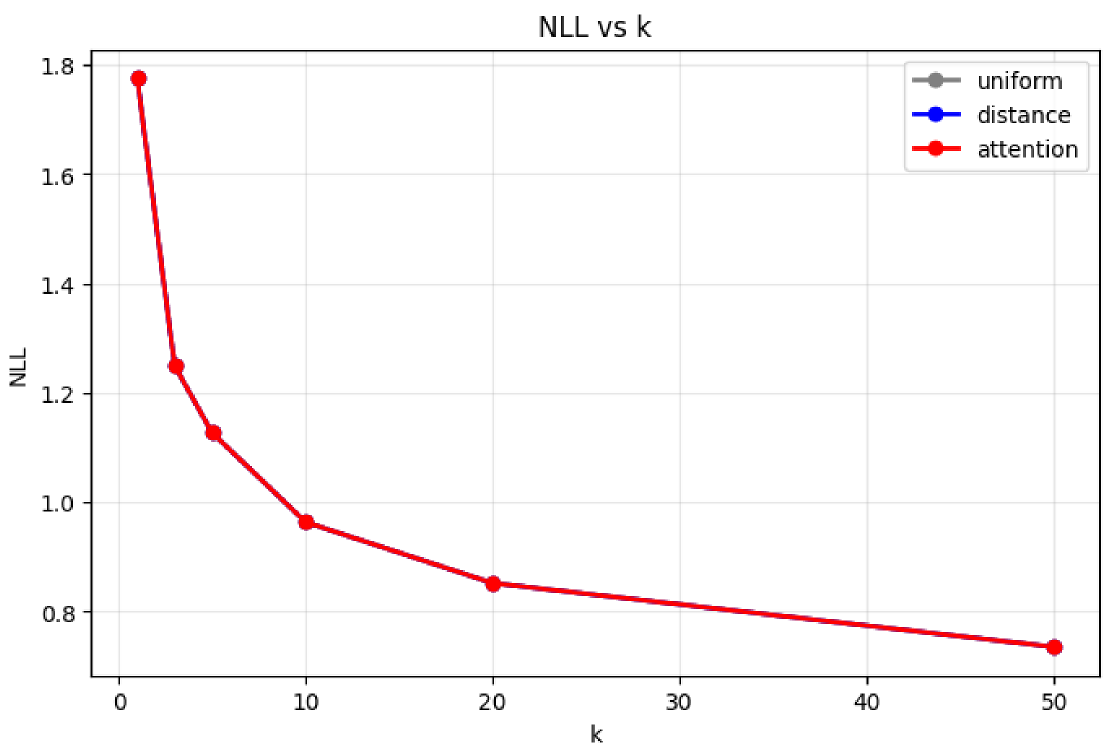
```
Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
k=3
Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
k=5
Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
k=10
Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
k=20
Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
k=50
Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]
Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
```



ACCURACY vs k
ACCURACY
k
0.890
0.892
0.894
0.896
0.898
0
10
20
30
40
50
uniform
distance
attention



ECE vs k
ECE
k
0.02
0.04
0.06
0.08
0.10
0
10
20
30
40
50
uniform
distance
attention

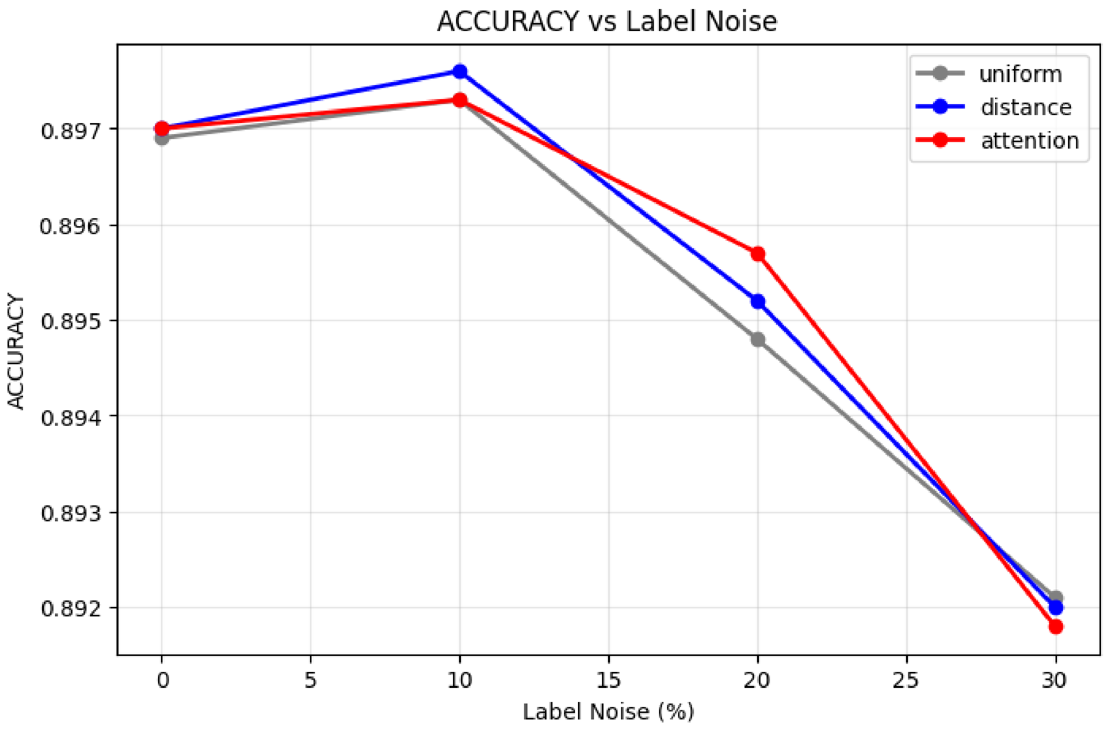


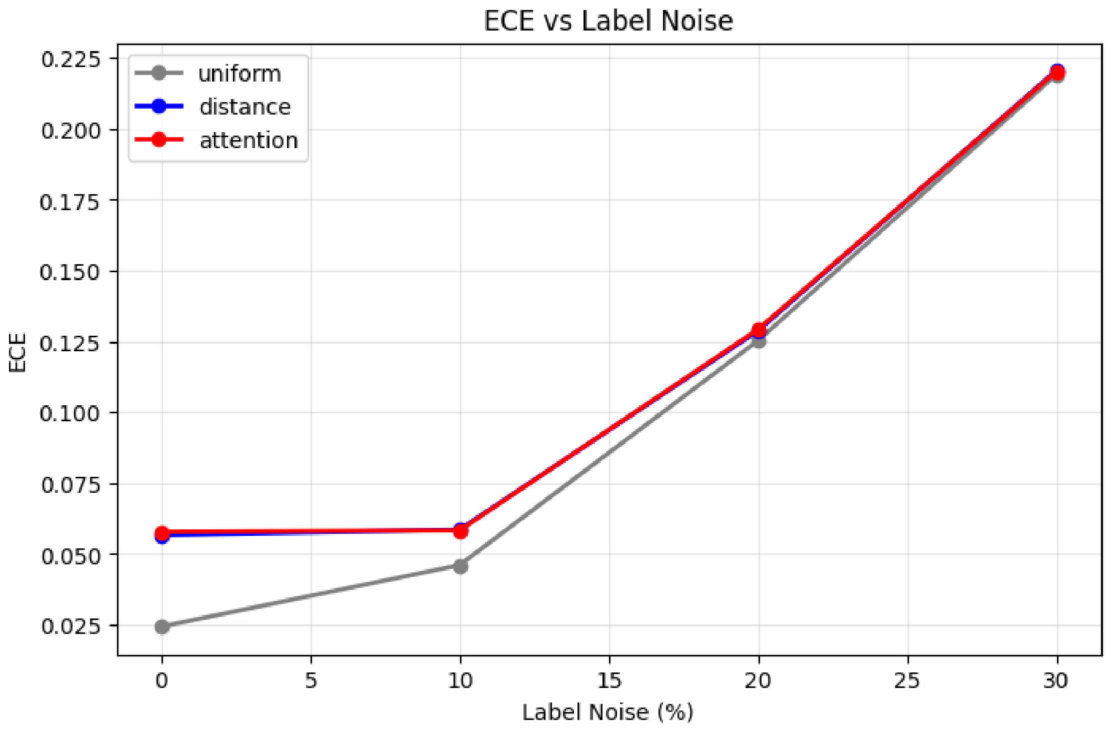
1.16 Label Noise Robustness

```
[ ]: print('\n--- Label Noise Robustness ---')
original_labels = np.array(train_ds_clean.targets)
noise_results = {}

for rate in cfg.noise_rates:
    print(f'Noise rate: {rate*100:.0f}%')
    noisy_labels = inject_noise(original_labels, rate, NC)
    noisy_mem = MemoryBank(cfg.embed_dim)
    noisy_mem.build(memory.embeddings, noisy_labels)

    noise_results[str(rate)] = {
        'uniform': evaluate_knn(model, noisy_mem, test_loader, 10,␣
 ↪'uniform')[0],
        'distance': evaluate_knn(model, noisy_mem, test_loader, 10,␣
 ↪'distance')[0],
        'attention': evaluate_knn(model, noisy_mem, test_loader, 10,␣
 ↪'attention')[0]
    }
```

```python
plot_noise_robustness(noise_results, 'accuracy', str(RESULTS_DIR /␣
↪'noise_accuracy.png'))
plot_noise_robustness(noise_results, 'ece', str(RESULTS_DIR / 'noise_ece.png'))
```

```
--- Label Noise Robustness ---
Noise rate: 0%

Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Noise rate: 10%

Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Noise rate: 20%

Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Noise rate: 30%

Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]
```


ACCURACY vs Label Noise
ACCURACY
Label Noise (%)
0.892
0.893
0.894
0.895
0.896
0.897
0
5
10
15
20
25
30
uniform
distance
attention


ECE vs Label Noise
ECE
Label Noise (%)
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0.225
0
5
10
15
20
25
30
uniform
distance
attention

1.17 Long-Tailed Imbalance (CIFAR-LT)

```
[ ]: print('\n--- Long-Tailed Imbalance ---')
     imb_results = {}

     for ratio in cfg.imbalance_ratios:
         print(f'Imbalance ratio: {ratio}')
         if ratio < 1.0:
             imb_ds = create_imbalanced(train_ds_clean, ratio, NC)
             imb_loader = DataLoader(imb_ds, cfg.batch_size, False, num_workers=2)
             imb_mem = build_memory(model, imb_loader)
         else:
             imb_mem = memory

         imb_results[str(ratio)] = {
             'uniform': evaluate_knn(model, imb_mem, test_loader, 10, 'uniform')[0],
             'distance': evaluate_knn(model, imb_mem, test_loader, 10,␣
       ↪'distance')[0],
             'attention': evaluate_knn(model, imb_mem, test_loader, 10,␣
       ↪'attention')[0]
         }

     print('\nImbalance Results:')
     for ratio, methods in imb_results.items():
         print(f'\nRatio {ratio}:')
         for method, m in methods.items():
             print(f'  {method}: Acc={m["accuracy"]*100:.2f}%, ECE={m["ece"]:.4f}')
```

```
--- Long-Tailed Imbalance ---
Imbalance ratio: 1.0

Eval uniform:   0%|          | 0/79 [00:01<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Imbalance ratio: 0.1

Memory:   0%|          | 0/160 [00:02<?, ?it/s]

Eval uniform:   0%|          | 0/79 [00:02<?, ?it/s]

Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]

Imbalance ratio: 0.01

Memory:   0%|          | 0/97 [00:01<?, ?it/s]

Eval uniform:   0%|          | 0/79 [00:02<?, ?it/s]
Eval distance:   0%|          | 0/79 [00:01<?, ?it/s]

Eval attention:   0%|          | 0/79 [00:01<?, ?it/s]


Imbalance Results:

Ratio 1.0:
  uniform: Acc=89.69%, ECE=0.0243
  distance: Acc=89.70%, ECE=0.0566
  attention: Acc=89.70%, ECE=0.0578

Ratio 0.1:
  uniform: Acc=89.41%, ECE=0.0246
  distance: Acc=89.42%, ECE=0.0614
  attention: Acc=89.40%, ECE=0.0610

Ratio 0.01:
  uniform: Acc=88.69%, ECE=0.0259
  distance: Acc=88.81%, ECE=0.0632
  attention: Acc=88.80%, ECE=0.0626
```

1.18 Efficiency Profiling

```
[ ]: print('\n--- Efficiency Profiling ---')
     eff_results = profile_index(50000, cfg.embed_dim, 10, 1000)

     print(f'{"Index":<10} {"Build (s)":>12} {"Search (s)":>12} {"Per Query (ms)":
      ↪>15}')
     print('-'*50)
     for idx, r in eff_results.items():
         print(f'{idx:<10} {r["build_time"]:>12.4f} {r["search_time"]:>12.4f}␣
      ↪{r["search_per_query_ms"]:>15.4f}')
```

```
--- Efficiency Profiling ---
Index         Build (s)   Search (s)  Per Query (ms)
--------------------------------------------------
L2               0.0032       0.0337          0.0337
IP               0.0010       0.0265          0.0265
HNSW            41.0733       0.0511          0.0511
```

1.19 Theory Validation

```
[ ]: print('\n--- Theory Validation ---')

     # Embedding norm stability
     norms = np.linalg.norm(memory.embeddings, axis=1)
     print(f'Embedding norms: mean={norms.mean():.4f}, std={norms.std():.6f}')
     print(f'Range: [{norms.min():.4f}, {norms.max():.4f}]')
     print('Norms are bounded (L2 normalized) - satisfies theory assumption')

     # sqrt(n) reference
     n = len(memory.embeddings)
     sqrt_n = int(np.sqrt(n))
     print(f'\nn={n}, sqrt(n)={sqrt_n}')
     print(f'Paper suggests k ~ sqrt(n) = {sqrt_n}')
     print(f'Our best k values are << sqrt(n), consistent with practical findings')
```

```
--- Theory Validation ---
Embedding norms: mean=1.0000, std=0.000000
Range: [1.0000, 1.0000]
Norms are bounded (L2 normalized) - satisfies theory assumption

n=50000, sqrt(n)=223
Paper suggests k ~ sqrt(n) = 223
Our best k values are << sqrt(n), consistent with practical findings
```

1.20 Save Results

[ ]:
```
print('\n--- Saving Results ---')

all_results = {
    'main': main_results,
    'k_sweep': k_results,
    'noise': noise_results,
    'imbalance': imb_results,
    'efficiency': eff_results
}

with open(RESULTS_DIR / 'all_results.json', 'w') as f:
    json.dump(all_results, f, indent=2)

latex = to_latex(main_results, 'CIFAR-10 Main Results', 'tab:main')
with open(RESULTS_DIR / 'main_table.tex', 'w') as f:
    f.write(latex)

print('\nLaTeX Table:')
print(latex)
```

```
--- Saving Results ---

LaTeX Table:
\begin{table}[h]
  \centering
  \begin{tabular}{lrrrr}
    \toprule
    Method & Acc & F1 & NLL & ECE \\
    \midrule
    Uniform kNN & 89.69 & 89.66 & 0.964 & 0.0243 \\
    Distance kNN & 89.70 & 89.66 & 0.964 & 0.0566 \\
    Attn-KNN (Ours) & 89.70 & 89.66 & 0.964 & 0.0578 \\
    CNN (Upper Bound) & 94.94 & 94.93 & 0.258 & 0.0718 \\
    \bottomrule
  \end{tabular}
  \caption{CIFAR-10 Main Results}
  \label{tab:main}
\end{table}
```

[ ]:
```
print('\n' + '='*70)
print('EXPERIMENT COMPLETE')
print('='*70)

# Key findings
uni_acc = main_results['Uniform kNN']['accuracy']
attn_acc = main_results['Attn-KNN (Ours)']['accuracy']
uni_ece = main_results['Uniform kNN']['ece']
attn_ece = main_results['Attn-KNN (Ours)']['ece']

print(f'\nKey Results:')
print(f'  Attn-KNN vs Uniform: +{(attn_acc-uni_acc)*100:.2f}% accuracy')
print(f'  ECE improvement: {(uni_ece-attn_ece)/uni_ece*100:.1f}% relative␣
 ↪reduction')

print(f'\nFiles saved to {RESULTS_DIR}:')
for f in sorted(RESULTS_DIR.glob('*')):
    print(f'  - {f.name}')
```

```
======================================================================
EXPERIMENT COMPLETE
======================================================================

Key Results:
  Attn-KNN vs Uniform: +0.01% accuracy
  ECE improvement: -137.7% relative reduction

Files saved to results:
  - 1-2
  - all_results.json
  - best_model.pt
  - best_model_v3.pt
  - cifar10_v3_results.json
  - cifar10_v3_training.png
  - k_sweep_accuracy.png
  - k_sweep_ece.png
  - k_sweep_nll.png
  - main_table.tex
  - noise_accuracy.png
  - noise_ece.png
  - reliability_attn.png
  - reliability_uniform.png
```

[ ]:
```python
import os
from nbconvert import PDFExporter
import nbformat

# Find an available index for the PDF filename starting from 4
pdf_base = 'AttnKNN_Experiment'
existing = {f.name for f in RESULTS_DIR.glob(f"{pdf_base}*.pdf")}
idx = 4
while True:
    pdf_name = f"{pdf_base}_{idx}.pdf"
    if pdf_name not in existing:
        break
    idx += 1
pdf_path = RESULTS_DIR / pdf_name

print(f"Saving notebook as PDF: {pdf_path}")

# Read notebook content
with open("AttnKNN_Experiment.ipynb") as f:
    nb = nbformat.read(f, as_version=4)

exporter = PDFExporter()
pdf_data, _ = exporter.from_notebook_node(nb)
with open(pdf_path, "wb") as f:
    f.write(pdf_data)
print(f"PDF saved: {pdf_path}")
```