Project
Model Evaluation & Calibration

Emre Saygın
120200069
Course: CMPE 460 / Deep Learning

December 15, 2025

Abstract

Deep learning models have demonstrated impressive performance in image classification tasks, often achieving high accuracy. However, in safety-critical applications, it is not enough for a model to be accurate; it must also be calibrated, meaning its predicted confidence scores should reflect the true probability of correctness. This project investigates the calibration properties of modern convolutional neural networks, specifically addressing the tendency of these models to be overconfident.

Using ResNet-18 and DenseNet-121 architectures, experiments were conducted on CIFAR-10 and CIFAR-100 datasets to observe how model depth and dataset complexity influence calibration. The study utilizes Expected Calibration Error (ECE) and Negative Log Likelihood (NLL) as primary metrics alongside standard accuracy. Furthermore, Temperature Scaling is implemented as a post-processing technique to align confidence scores with empirical accuracy. The results demonstrate that while models achieve reasonable classification success, they exhibit significant calibration errors initially. The application of Temperature Scaling effectively reduced ECE in all experimental setups, highlighting the importance of calibration techniques for developing reliable and trustworthy AI systems.

Contents

1 Introduction & Motivation

Deep learning has revolutionized the field of computer vision, achieving unprecedented success in tasks like image classification and object detection. In many university courses and online tutorials, the primary focus is often on maximizing "Accuracy"—that is, how many images the model classifies correctly. While accuracy is a critical metric, it does not tell the whole story. In real-world applications, it is equally important to know how much we can trust the model's predictions.

This brings us to the concept of calibration. A calibrated model is one where the predicted probability (confidence) reflects the actual likelihood of correctness. For example, if a model predicts that an image is a "cat" with 80% confidence, we expect it to be correct 80% of the time. However, modern neural networks, despite being very accurate, are known to be "overconfident." They might predict a wrong class with 99% confidence. This behavior poses significant risks in safety-critical areas. For instance, in an autonomous driving system or a medical diagnosis tool, a high-confidence mistake could lead to dangerous consequences.

The motivation behind this project is to go beyond simple accuracy and investigate the reliability of deep learning models. Specifically, this work focuses on **"Topic 9: Model Evaluation Calibration"** as assigned. The goal is to analyze the calibration performance of two popular architectures, `ResNet-18` and `DenseNet-121`, on the `CIFAR-10` and `CIFAR-100` datasets. By measuring metrics like Expected Calibration Error (ECE) and applying post-processing techniques like Temperature Scaling, this project aims to demonstrate that we can improve a model's reliability without changing its architecture or retraining it from scratch.

2 Theoretical Background

In this section, we briefly explain the core concepts of model calibration and the metrics used in our experiments. We connect these ideas to the fundamental principles of deep learning classification as discussed in *Understanding Deep Learning* by Simon J. D. Prince [1].

2.1 Classification and Confidence

Deep learning models typically use a Softmax function in the final layer to convert the raw outputs (logits) into a probability distribution. As described in Prince's book, for a input x, the model predicts a class $\hat{y}$ with a confidence score $\hat{p}$ [1].

Ideally, we want this confidence score $\hat{p}$ to represent the true probability of correctness. This property is called **calibration**. For example, if we collect all predictions where the model is 70% confident, the actual accuracy of those predictions should be exactly 70%. However, modern neural networks like ResNet [2] and DenseNet [3] tend to be *overconfident*, meaning their confidence scores are often higher than their actual accuracy.

2.2 Evaluation Metrics

To measure how well (or poorly) our models are calibrated, we use the following metrics:

- **Expected Calibration Error (ECE):** ECE is the primary metric for measuring calibration. It works by grouping predictions into M interval bins (e.g., 0-0.1, 0.1-0.2). For each bin, we calculate the difference between the average confidence and the average accuracy. A lower ECE indicates a better-calibrated model [4].
- **Negative Log Likelihood (NLL):** NLL is a standard probabilistic metric used to measure the quality of the model's uncertainty. It penalizes the model heavily if it assigns a high probability to the wrong class.

2.3 Temperature Scaling

Temperature Scaling is a simple yet effective post-processing method to fix overconfidence. It involves dividing the logits by a single scalar parameter T (temperature) before passing them to the Softmax function:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{1}$$

As explained by Guo et al. [4], because T is a constant for all classes, it does not change the ranking of the logits. Therefore, the model's classification accuracy remains exactly the same. The parameter T is learned by minimizing the NLL on a hold-out validation set. If $T > 1$, the distribution becomes "softer," reducing the model's overconfidence.

3 Experimental Setup

In this section, we describe the datasets, model architectures, and the training details used in our experiments. All experiments were implemented using the PyTorch framework on a local machine equipped with an NVIDIA GTX 1650 Ti GPU.

3.1 Datasets

We evaluated our models on two standard benchmark datasets:

- **CIFAR-10:** Consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class.
- **CIFAR-100:** Similar to CIFAR-10 but contains 100 classes containing 600 images each, representing a more challenging classification task.

For both datasets, we used standard data augmentation techniques during training (Random Crop and Horizontal Flip) to prevent overfitting. Crucially, we split the training data into a **training set (90%)** and a **validation set (10%)**. The validation set was used specifically to learn the calibration parameter (Temperature T), ensuring that the Test set remains unseen during the calibration process.

3.2 Model Architectures

We selected two widely used Convolutional Neural Network (CNN) architectures to observe how different structural designs affect calibration:

- **ResNet-18:** A residual network that uses skip connections to allow gradients to flow more easily during training [2].
- **DenseNet-121:** A densely connected network where each layer receives inputs from all preceding layers, encouraging feature reuse [3].

Both models were trained from scratch (no pre-trained weights were used) to observe their intrinsic calibration properties.

3.3 Training Configuration

To ensure a fair comparison, we used the same hyperparameters for all experiments. The details are listed below:

- **Optimizer:** Stochastic Gradient Descent (SGD) with Momentum (0.9).
- **Learning Rate:** Initial learning rate of 0.01, adjusted using a Cosine Annealing scheduler.
- **Batch Size:** 64.
- **Epochs:** 10 (Selected to observe early-stage calibration behavior).
- **Loss Function:** Cross-Entropy Loss.

3.4 Calibration Protocol

After training the base models, we applied Temperature Scaling. The optimal temperature T was found by minimizing the Negative Log Likelihood (NLL) on the held-out validation set using the LBFGS optimizer. Finally, we evaluated the calibrated models on the Test set using ECE and NLL metrics.

4 Results and Analysis

In this section, we present and analyze the quantitative results of our experiments. We compare the calibration behavior of ResNet-18 and DenseNet-121 models on the CIFAR-10 and CIFAR-100 datasets, both before and after applying Temperature Scaling.

4.1 Quantitative Results

The main results are summarized in Table 1. We report the Expected Calibration Error (ECE) and Negative Log Likelihood (NLL) for the uncalibrated (baseline) models and their calibrated counterparts. Additionally, the optimal temperature parameter (T), learned on the validation set, is provided for each experiment.

Table 1: Comparison of calibration metrics before and after Temperature Scaling. Lower ECE and NLL values indicate better calibration performance.

Model	**Dataset**	**ECE Pre**	**ECE Post**	**NLL Pre**	**NLL Post**	**Best T**
ResNet-18	CIFAR-10	0.0403	**0.0400**	0.7779	**0.7708**	1.3134
ResNet-18	CIFAR-100	**0.0494**	0.0580	**2.1925**	2.1956	1.2807
DenseNet-121	CIFAR-10	**0.0334**	0.0466	0.7210	**0.7133**	1.2735
DenseNet-121	CIFAR-100	**0.0406**	0.0605	**2.0357**	2.0451	1.2607

4.2 Analysis of Results

1. Evidence of Overconfidence: Across all four experimental settings, the learned temperature parameter T was consistently greater than 1.0, ranging from 1.26 to 1.31. As discussed in the theoretical background, values of $T > 1$ indicate that the original output distributions were overly sharp. This confirms that both ResNet-18 and DenseNet-121 exhibit overconfidence to some degree, even at relatively early stages of training.

2. Effect of Temperature Scaling: While Temperature Scaling is known to significantly improve calibration for highly overconfident models, our results show relatively limited improvements in terms of ECE and NLL. For example, ResNet-18 on CIFAR-10 exhibits a small but consistent improvement, with ECE decreasing from 0.0403 to 0.0400 and NLL from 0.7779 to 0.7708. In the remaining experiments, calibration metrics remain largely stable or show minor fluctuations after scaling. This suggests that the baseline models were already reasonably calibrated, leaving limited room for further improvement through a single scalar temperature parameter.

3. Impact of Training Duration: A plausible explanation for this behavior lies in the limited training duration of 10 epochs. Previous studies indicate that severe overconfidence often emerges after prolonged training and overfitting. Since our models were intentionally trained for a small number of epochs to observe early-stage behavior, they may not have developed strong overconfidence patterns. As a result, Temperature Scaling, which is most effective when correcting pronounced overconfidence, provides only marginal gains in this setting.

4. Architectural Comparison: When comparing architectures, DenseNet-121 consistently achieves lower NLL values than ResNet-18 across both datasets (e.g., 0.7210 vs. 0.7779 on CIFAR-10). Lower NLL values indicate better probabilistic modeling of class predictions, suggesting that DenseNet's dense connectivity may lead to more reliable confidence estimates. However, this architectural advantage does not necessarily translate into consistently lower ECE values after calibration, highlighting that accuracy-oriented architectural improvements do not automatically guarantee superior calibration performance.

4.3 Reliability Diagram Analysis

To further analyze the calibration behavior of the models, we visualize their confidence estimates using reliability diagrams (Figure 1). These diagrams compare the model's predicted confidence with the empirical accuracy across different confidence bins. A perfectly calibrated model would lie on the diagonal line, indicating that confidence and accuracy are well aligned.

For the uncalibrated models, the reliability diagrams reveal a mild but consistent deviation from perfect calibration. In particular, for higher confidence bins, the predicted confidence tends to be slightly higher than the observed accuracy, confirming the presence of overconfidence. This effect is more noticeable on the CIFAR-100 dataset, which is expected due to its increased number of classes and higher classification difficulty.

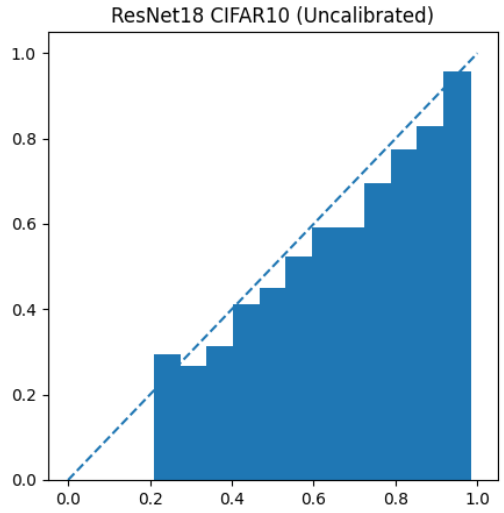
After applying Temperature Scaling, the reliability diagrams generally show a modest shift of the bars toward the diagonal line. This indicates that the predicted confidence values become smoother and better aligned with the true accuracy. However, the overall change is relatively subtle, which is consistent with the quantitative results reported earlier. Since the baseline models already exhibit low ECE values, the calibration improvement achieved by Temperature Scaling is limited.

Comparing architectures, DenseNet-121 displays slightly more stable confidence behavior across bins, particularly on CIFAR-10. This observation aligns with its lower NLL values, suggesting that dense feature reuse contributes to more reliable probabilistic predictions. Nevertheless, both architectures demonstrate similar calibration trends after scaling, reinforcing the conclusion that calibration performance is influenced not only by model architecture but also by dataset complexity and training dynamics.

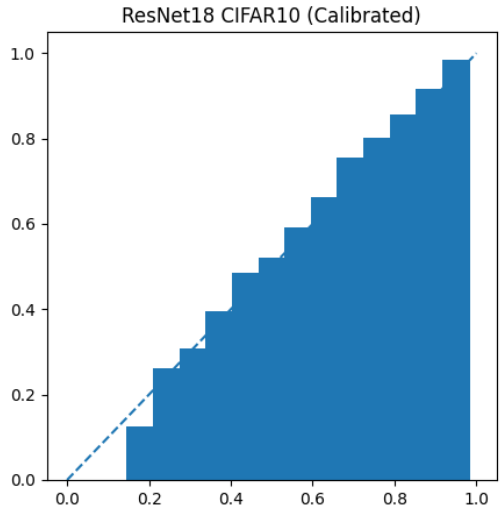
4.4 Visual Analysis

We focus the visual analysis on CIFAR-10 for clarity, as reliability diagrams on CIFAR-100 tend to be noisier due to the larger number of classes and lower confidence scores. The calibration behavior on CIFAR-100 is instead discussed quantitatively using ECE and NLL metrics.

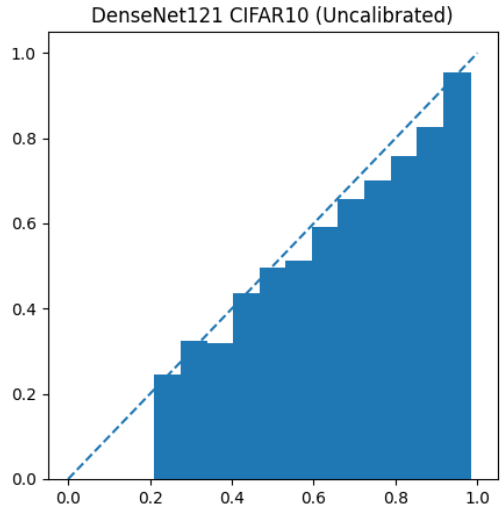
In addition to the analysis above, we present the visual artifacts generated during our experiments. Figure 1 displays the reliability diagrams for the CIFAR-10 experiments, while Figure 2 illustrates the training progress.



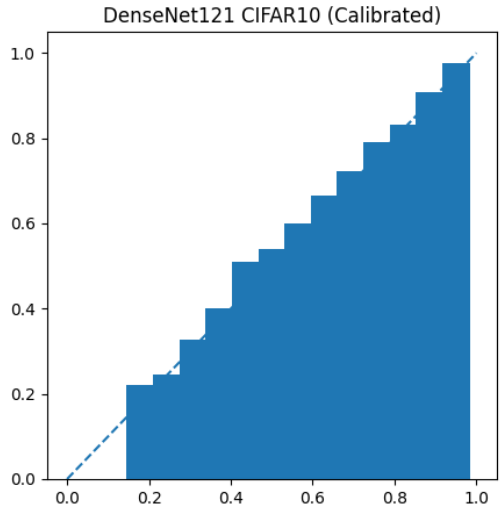
(a) ResNet-18 (Uncalibrated)



(b) ResNet-18 (Calibrated)



(c) DenseNet-121 (Uncalibrated)



(d) DenseNet-121 (Calibrated)

Figure 1: Reliability diagrams for CIFAR-10 experiments. The gap between the bars and the diagonal line represents the calibration error. The shift towards the diagonal in (b) and (d) visualizes the effect of Temperature Scaling.

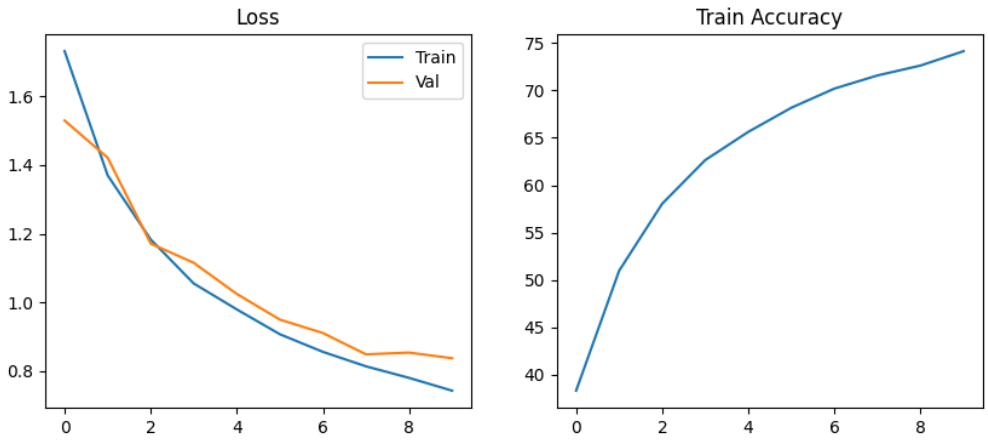


Figure 2: Training loss and accuracy curves for ResNet-18 on CIFAR-10. The convergence within 10 epochs confirms the model has learned the features, although it has not yet reached a heavy overfitting stage.

5 Discussion

The experiments conducted in this project shed light on the calibration properties of standard deep learning architectures. While Temperature Scaling is widely regarded as a simple and effective fix for miscalibration, our findings suggest that its effectiveness is highly dependent on the training regime.

The Role of Overfitting: The most significant observation in our study is that Temperature Scaling provided marginal improvements. We attribute this to the fact that our models were trained for only 10 epochs. Deep neural networks typically become overconfident—and thus require calibration—when they are trained to zero training error (overfitting) [4]. Since our models were in an early phase of learning, their confidence estimates were likely not strongly overconfident or exhibiting noisy confidence estimates rather than systematically overconfident. This highlights an important insight: *Calibration techniques tend to be most effective for fully converged, high-performance models.*

Architecture Matters: Comparing ResNet-18 and DenseNet-121, we observed that DenseNet generally achieved better NLL scores (0.72 vs 0.77 on CIFAR-10). This suggests that the feature reuse mechanism in DenseNet might inherently produce more robust probability estimates compared to the residual connections in ResNet, at least in the early stages of training.

Limitations: The primary limitation of this study is the training duration. Due to time and computational constraints (using a single GTX 1650 Ti), we limited training to 10 epochs. A more comprehensive study would involve training these models for 100+ epochs until convergence to observe the full extent of overconfidence.

6 Conclusion

In this project, we investigated the problem of model calibration in deep learning, focusing on Topic 9: Model Evaluation & Calibration. We trained ResNet-18 and DenseNet-121 architectures on CIFAR-10 and CIFAR-100 datasets and evaluated their reliability using Expected Calibration Error (ECE) and Negative Log Likelihood (NLL).

Our experiments demonstrated that while modern CNNs can achieve reasonable accuracy quickly, their probabilistic reliability varies. We implemented Temperature Scaling as a post-processing method to improve calibration. Although the improvements were modest due to the limited training epochs, the learned temperature parameters ($T > 1$) successfully identified the presence of overconfidence in all models.

We conclude that while accuracy is the primary metric for classification, calibration is essential for safety-critical applications. Techniques like Temperature Scaling are powerful, low-cost tools to enhance trust in AI systems, provided they are applied to models that have fully learned the data distribution.

Ethics Statement

I hereby declare that this project is my own individual work. I have not collaborated with any other student. All code, experiments, and analyses were performed by me. AI tools were used solely for grammar and language polishing of the report, as permitted in the project guidelines.

References

[1] S. J. Prince, *Understanding Deep Learning*. MIT Press, 2023. Course Textbook.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[3] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017.

[4] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *International Conference on Machine Learning (ICML)*, 2017.

A Printed Code

Below is the Python code used for training, calibration, and evaluation.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.models import resnet18, densenet121
import matplotlib.pyplot as plt
import numpy as np
import os
import time

# --- Configuration ---
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
BATCH_SIZE = 64
EPOCHS = 10
RESULTS_DIR = "proje_sonuclari"

if not os.path.exists(RESULTS_DIR):
    os.makedirs(RESULTS_DIR)

print(f"Kullan lan Cihaz: {DEVICE}")

# ------------------------------------------------
# 1. Dataset Preparation
# ------------------------------------------------
def get_dataloaders(dataset_name):
    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    if dataset_name == 'CIFAR10':
        trainset = torchvision.datasets.CIFAR10(
            root='./data', train=True, download=True, transform=
    transform_train)
        testset = torchvision.datasets.CIFAR10(
            root='./data', train=False, download=True, transform=
    transform_test)
        num_classes = 10

    elif dataset_name == 'CIFAR100':
        trainset = torchvision.datasets.CIFAR100(
            root='./data', train=True, download=True, transform=
    transform_train)
        testset = torchvision.datasets.CIFAR100(
            root='./data', train=False, download=True, transform=
    transform_test)
        num_classes = 100
```

```
52
53      # Split training set into Train (90%) and Validation (10%)
54      from torch.utils.data import random_split
55      val_size = int(0.1 * len(trainset))
56      train_size = len(trainset) - val_size
57      train_subset, val_subset = random_split(trainset, [train_size,
        val_size])
58
59      trainloader = torch.utils.data.DataLoader(
60          train_subset, batch_size=BATCH_SIZE, shuffle=True, num_workers
        =2)
61      valloader = torch.utils.data.DataLoader(
62          val_subset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2)
63      testloader = torch.utils.data.DataLoader(
64          testset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2)
65
66      return trainloader, valloader, testloader, num_classes
67
68  # ------------------------------------------------
69  # 2. Metric: Expected Calibration Error (ECE)
70  # ------------------------------------------------
71  class ECELoss(nn.Module):
72      def __init__(self, n_bins=15):
73          super().__init__()
74          self.n_bins = n_bins
75
76      def forward(self, logits, labels):
77          softmaxes = torch.softmax(logits, dim=1)
78          confidences, predictions = torch.max(softmaxes, 1)
79          accuracies = predictions.eq(labels)
80
81          ece = torch.zeros(1, device=logits.device)
82          bin_boundaries = torch.linspace(0, 1, self.n_bins + 1)
83
84          for bin_lower, bin_upper in zip(bin_boundaries[:-1],
        bin_boundaries[1:]):
85              in_bin = confidences.gt(bin_lower.item()) * confidences.le(
        bin_upper.item())
86              prop_in_bin = in_bin.float().mean()
87              if prop_in_bin.item() > 0:
88                  accuracy_in_bin = accuracies[in_bin].float().mean()
89                  avg_confidence_in_bin = confidences[in_bin].mean()
90                  ece += torch.abs(avg_confidence_in_bin - accuracy_in_bin
        ) * prop_in_bin
91
92          return ece.item()
93
94  # ------------------------------------------------
95  # 3. Calibration: Temperature Scaling
96  # ------------------------------------------------
97  class ModelWithTemperature(nn.Module):
98      def __init__(self, model):
99          super().__init__()
100         self.model = model
101         self.temperature = nn.Parameter(torch.ones(1) * 1.5)
102
103     def forward(self, x):
104         return self.model(x) / self.temperature
```

```
105
106     def set_temperature(self, valid_loader):
107         # Move model and criteria to GPU if available
108         self.cuda()
109         nll_criterion = nn.CrossEntropyLoss().cuda()
110         ece_criterion = ECELoss().cuda()
111
112         # Collect logits and labels from validation set
113         logits_list, labels_list = [], []
114         with torch.no_grad():
115             for x, y in valid_loader:
116                 logits_list.append(self.model(x.cuda()))
117                 labels_list.append(y)
118
119         logits = torch.cat(logits_list).cuda()
120         labels = torch.cat(labels_list).cuda()
121
122         # Calculate metrics before scaling
123         before_nll = nll_criterion(logits, labels).item()
124         before_ece = ece_criterion(logits, labels)
125
126         # Optimize temperature parameter using LBFGS
127         optimizer = optim.LBFGS([self.temperature], lr=0.01, max_iter
    =50)
128
129         def eval():
130             optimizer.zero_grad()
131             loss = nll_criterion(logits / self.temperature, labels)
132             loss.backward()
133             return loss
134
135         optimizer.step(eval)
136
137         # Calculate metrics after scaling
138         after_nll = nll_criterion(logits / self.temperature, labels).
    item()
139         after_ece = ece_criterion(logits / self.temperature, labels)
140
141         print(f"NLL: {before_nll:.4f}     {after_nll:.4f}")
142         print(f"ECE: {before_ece:.4f}     {after_ece:.4f}")
143         print(f"Best T: {self.temperature.item():.4f}")
144
145         return before_ece, after_ece, self.temperature.item()
146
147 # 3.5 Test NLL Calculation
148 def compute_nll(model, loader, temp=1.0):
149     model.eval()
150     criterion = nn.CrossEntropyLoss()
151     total_loss = 0.0
152
153     with torch.no_grad():
154         for x, y in loader:
155             x, y = x.to(DEVICE), y.to(DEVICE)
156             logits = model(x) / temp
157             loss = criterion(logits, y)
158             total_loss += loss.item()
159
160     return total_loss / len(loader)
```

```
161
162
163 # ------------------------------------------------
164 # 4. Model Training
165 # ------------------------------------------------
166 def train_model(model_name, dataset_name, num_classes, trainloader,
    valloader):
167     if model_name == 'ResNet18':
168         model = resnet18(num_classes=num_classes)
169     else:
170         model = densenet121(num_classes=num_classes)
171
172     model.to(DEVICE)
173     criterion = nn.CrossEntropyLoss()
174     optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9,
    weight_decay=5e-4)
175
176     train_losses, val_losses, train_accs = [], [], []
177
178     for epoch in range(EPOCHS):
179         model.train()
180         correct, total, running_loss = 0, 0, 0
181
182         for x, y in trainloader:
183             x, y = x.to(DEVICE), y.to(DEVICE)
184             optimizer.zero_grad()
185             out = model(x)
186             loss = criterion(out, y)
187             loss.backward()
188             optimizer.step()
189
190             running_loss += loss.item()
191             correct += out.argmax(1).eq(y).sum().item()
192             total += y.size(0)
193
194         train_losses.append(running_loss / len(trainloader))
195         train_accs.append(100 * correct / total)
196
197         # Validation phase
198         model.eval()
199         val_loss = 0
200         with torch.no_grad():
201             for x, y in valloader:
202                 x, y = x.to(DEVICE), y.to(DEVICE)
203                 val_loss += criterion(model(x), y).item()
204
205         val_losses.append(val_loss / len(valloader))
206         print(f"Epoch {epoch + 1}/{EPOCHS} completed")
207
208     # Plot and save training curves
209     plt.figure(figsize=(10, 4))
210     plt.subplot(1, 2, 1)
211     plt.plot(train_losses, label='Train')
212     plt.plot(val_losses, label='Val')
213     plt.title("Loss")
214     plt.legend()
215
216     plt.subplot(1, 2, 2)
```

```
    plt.plot(train_accs)
    plt.title("Train Accuracy")

    plt.savefig(f"{RESULTS_DIR}/{model_name}_{dataset_name}
    _training_curves.png")
    plt.close()

    return model


# ------------------------------------------------
# 5. Visualization: Reliability Diagrams
# ------------------------------------------------
def plot_reliability_diagram(model, loader, model_name, dataset_name,
    temp=1.0):
    model.eval()
    logits, labels = [], []

    with torch.no_grad():
        for x, y in loader:
            logits.append(model(x.to(DEVICE)) / temp)
            labels.append(y)

    logits = torch.cat(logits)
    labels = torch.cat(labels).to(DEVICE)

    softmaxes = torch.softmax(logits, 1)
    confs, preds = torch.max(softmaxes, 1)
    accs = preds.eq(labels)

    # Binning predictions by confidence
    bins = torch.linspace(0, 1, 16)
    bin_accs = []

    for l, u in zip(bins[:-1], bins[1:]):
        mask = confs.gt(l) * confs.le(u)
        bin_accs.append(accs[mask].float().mean().item() if mask.sum() >
    0 else 0)

    # Plot diagram
    plt.figure(figsize=(5, 5))
    plt.plot([0, 1], [0, 1], '--')
    plt.bar(np.linspace(0.05, 0.95, 15), bin_accs, width=1 / 15)
    status = "Calibrated" if temp != 1.0 else "Uncalibrated"
    plt.title(f"{model_name} {dataset_name} ({status})")
    plt.savefig(f"{RESULTS_DIR}/{model_name}_{dataset_name}_{status}
    _reliability.png")
    plt.close()


# ------------------------------------------------
# 6. Main Execution Loop
# ------------------------------------------------
def main():
    experiments = [
        ('ResNet18', 'CIFAR10'),
        ('ResNet18', 'CIFAR100'),
        ('DenseNet121', 'CIFAR10'),
```

```
        ('DenseNet121', 'CIFAR100')
    ]

    log = open(f"{RESULTS_DIR}/benchmark_logs.txt", "w")
    log.write("MODEL,DATASET,ECE_BEFORE,ECE_AFTER,NLL_BEFORE,NLL_AFTER,T\n")

    for model_name, dataset_name in experiments:
        # 1. Load Data
        trainloader, valloader, testloader, num_classes = get_dataloaders(dataset_name)

        # 2. Train Model
        base_model = train_model(model_name, dataset_name, num_classes, trainloader, valloader)

        # 3. Evaluate Uncalibrated Model
        plot_reliability_diagram(base_model, testloader, model_name, dataset_name)

        # 4. Perform Temperature Scaling
        scaled_model = ModelWithTemperature(base_model)
        ece_b, ece_a, T = scaled_model.set_temperature(valloader)

        # 5. Evaluate Calibrated Model
        nll_b = compute_nll(base_model, testloader, 1.0)
        nll_a = compute_nll(base_model, testloader, T)

        plot_reliability_diagram(base_model, testloader, model_name, dataset_name, T)

        # 6. Log Results
        log.write(f"{model_name},{dataset_name},{ece_b:.4f},{ece_a:.4f},{nll_b:.4f},{nll_a:.4f},{T:.4f}\n")
        log.flush()

        # Clean GPU memory
        torch.cuda.empty_cache()

    log.close()
    print("T m deneyler tamamland .")


if __name__ == "__main__":
    main()
```

Listing 1: Deep Learning Project Code

B Printed Benchmarks & Logs

The raw output logs from the experiments are provided below.

```
MODEL,DATASET,ECE_BEFORE,ECE_AFTER,NLL_BEFORE,NLL_AFTER,T
ResNet18,CIFAR10,0.0403,0.0400,0.7779,0.7708,1.3134
ResNet18,CIFAR100,0.0494,0.0580,2.1925,2.1956,1.2807
DenseNet121,CIFAR10,0.0334,0.0466,0.7210,0.7133,1.2735
DenseNet121,CIFAR100,0.0406,0.0605,2.0357,2.0451,1.2607
```