Homework 4
CSC 277 / 477
End-to-end Deep Learning
Fall 2024

Tianyi Zhou - tzhou25@u.rochester.edu

Nov.12 2024

Deadline: See Blackboard

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. The relevant code should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

About Homework 4: In this assignment, we will study model calibration, conformal prediction. Copy and paste this template into an editor, e.g., `www.overleaf.com`, and then just type the answers in. You can use a math editor to make this easier, e.g., CodeCogs Equation Editor or MathType. You may use the AI (LLM) plugin for Overleaf for help you with LaTeXformatting.

Problem 1 - Model Calibration (22 points)

Model Calibration: In probabilistic modeling, a model is said to be calibrated if, for all instances where it predicts a probability p of an event occurring, the event actually occurs approximately $p \times 100\%$ many times out of 100 occurences. Formally, for a set of instances for which the model predicts a probability p, the ratio of those instances for which the event occurs should be close to p.

Real-life Example: Consider a medical tool that predicts a 90% likelihood of a specific disease. If 10 patients receive this 90% prediction, around 9 of them should genuinely have the disease for the tool to be reliable. If only 5 of them have the disease, the tool's predictions are not calibrated well.

Evaluating Model Calibration:

1. **Reliability Curve**: A graph where predicted probabilities are on the x-axis, and the actual outcomes are on the y-axis. A perfectly accurate model will match the diagonal line (where $y = x$).
2. **Expected Calibration Error (ECE)**: A metric that quantifies the divergence of predictions from the truth. A lower ECE indicates a more calibrated model.
3. **Maximum Calibration Error (MCE)**: The largest difference between predicted probabilities and real outcomes. A lower MCE indicates better calibration.

Problem Set:

For this homework's evaluation we will focus on reliability curves. For binary classification we provided you a modified version of CIFAR10 dataset which has only dog and cat classes (see the pyhton code). Use ResNet 18 model (you can start with pretrained weights). Do not forget to modify your network for binary classification.

Part 1: Platt Scaling (8 points)

Platt Scaling is a post-processing method designed to transform the continuous-valued predictions of a classifier into calibrated probabilities. Although it was originally conceived for support vector machines, it has since been applied to many other types of models, notably deep neural networks.

Given a binary classification task, let's assume the raw output (logit: the outputs of the final layer before applying the activation function.) of the model for an instance is denoted by z. Platt Scaling fits the subsequent sigmoid function to these outputs:

$$P(y = 1|z) = \frac{1}{1 + \exp(A \cdot z + B)}$$

In this equation, y represents the binary label, while A and B are parameters determined from a validation set. The objective is to minimize the negative log likelihood on this validation set in terms of A and B.

Pseudo-code for Platt Scaling:

1. Train the primary model (e.g., a neural network) using the training dataset.
2. Obtain the logits (pre-activation outputs) of the primary model on a validation set.
3. Determine the parameters A and B through logistic regression, using the logits as input features and the actual labels of the validation set as targets.
4. For any new data point, to obtain the calibrated probability:
   (a) Process the data point with the primary model to get the logit, z.
   (b) Calculate the probability using $P(y = 1|z) = \frac{1}{1+\exp(A \cdot z+B)}$.

Your Task:

1. Use the binary classification dataset and the model info we provided to build and train that model.
2. Draw reliability curve on the test data.
3. Implement Platt Scaling on the model's outputs on a validation set.
4. Draw reliability curve on the test data.
5. Assess the calibration using the two reliability curves you created in previous steps, comment on it.

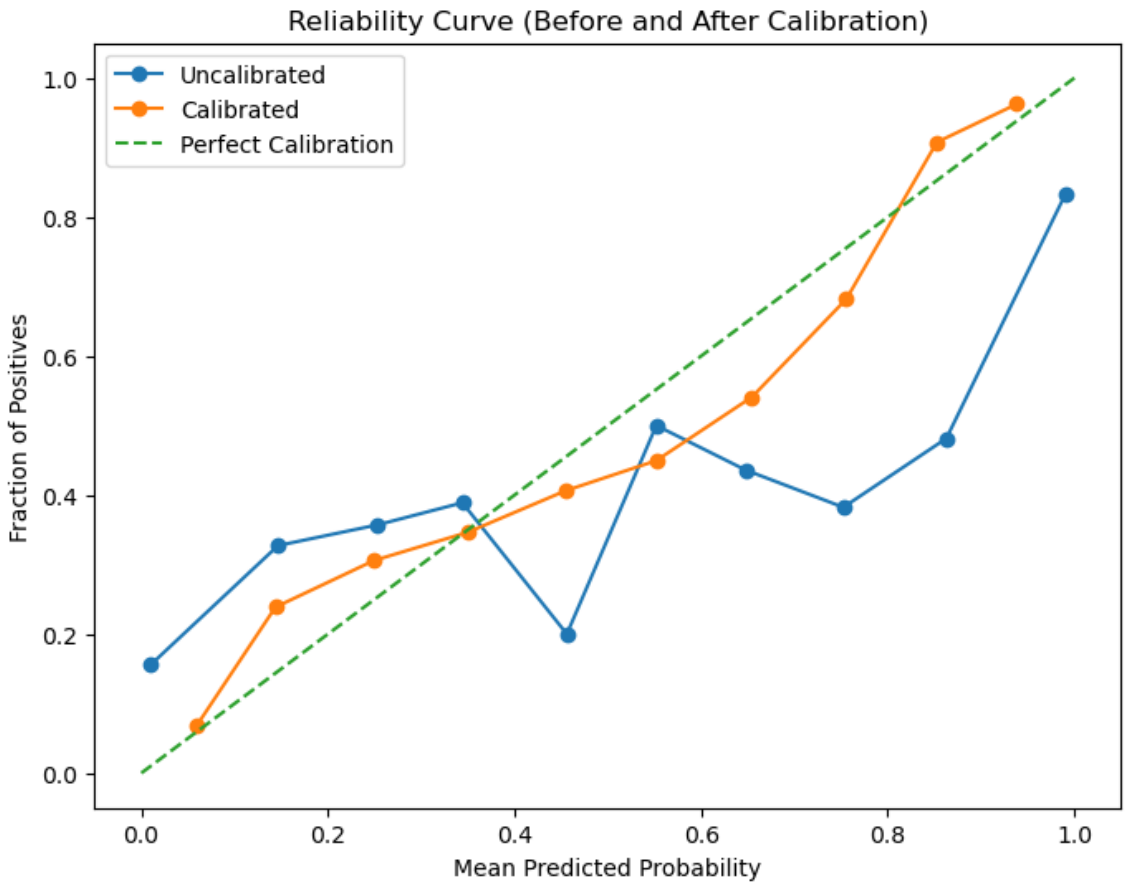


Figure 1: Reliability curves before and after Platt Scaling

Analysis:
A reliability curve (calibration curve) depicts correlation between predicted probabilities by the model and the observed frequencies of the predicted outcome. The chart shows two reliability curves, line in blue is the curve before platt scaling and line in orange is the curve after platt scaling. We can observe that after model calibration using platt scaling, the curve fit more to the perfect calibration line (in green).

Part 2: Improving Calibration with Label Smoothing (14 points)

Label smoothing is a regularization technique in which the hard 0 and 1 labels are replaced with smoothed values. This can prevent overconfidence in predictions and may lead to better-calibrated models.

Your Task:

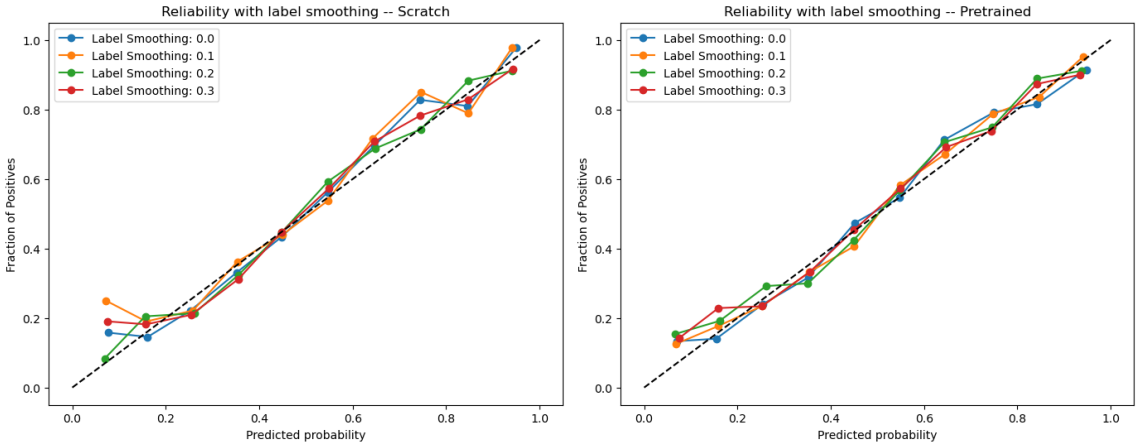
1. Fine-tune the pretrained model from part 1 using label smoothing. Experiment with different smoothing values (e.g., 0.1, 0.2, 0.3) and evaluate the calibration of each model using a reliability curve. Compare them with each other and results from part 1 verbally (no extra plots needed).(4 points)
2. Train the model from scratch using label smoothing. Experiment with different smoothing values (e.g., 0.1, 0.2, 0.3) and evaluate calibration with a reliability curve. Compare them with each other and results from part 1 verbally (no extra plots needed).(4 points)
3. Apply Plat-Scaling on models from Part 1, item(1) and item(2) with smooth labels. Draw reliability curves and comment on results. (6 points)

Deliverables:

1. Provide your answers to what is asked in the question. (both verbal comparisons and diagrams)
2. Provide your code for credit.

Comments:

- **Finetune with label smoothing:** At first glance, I cannot draw obvious distinction between curves. Label smoothing with value 0.2 is the closest to the diagonal line while the curve seems to be overcorrected when value increase to 0.3.
- **Train from scratch using label smoothing:** The difference between curves are also subtle. However, it's comparatively clearer to observe a converging trends as smoothing values increase (curves fitting closer to the diagonal line).
- **Curves after Platt-Scaling:**



(a) models trained from scratch with platt scaling (b) pretrained models with platt scaling

Here, smoothing value 0.0 represents models from part 1. I plotted two graphs, one is base models trained with different smoothing values and the other is pre-trained models with different values, both are applied with platt scaling. On this run, we can see pretrained resnet18 after platt scaling is more calibrated given predicted probability is either very low $(0.0 - 0.2)$ or very high $(0.8 - 1.0)$. Overall, we can see the curves fit more (more calibrated) as smooth values increases and pretrained model fit slightly better.

Listing 1: Code Implementation of P1P2

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import models, transforms
from torchvision.models import ResNet18_Weights

from Model_calibration_dataset import get_cifar10_classes
import matplotlib.pyplot as plt
from sklearn.calibration import calibration_curve
from sklearn.linear_model import LogisticRegression
#%%
# Retrieve datasets and dataloaders
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010))
])
```

```
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010))
])
train_dataset, val_dataset, test_data = get_cifar10_classes(
    transform_train, transform_test)

# Create dataloader
batch_size=128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle
    =True, num_workers=2)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=
    False, num_workers=2)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=
    False, num_workers=2)

# Define the model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#%%
# Common functions
class LabelSmoothingBCE(nn.Module):  # CrossEntropyLoss has label
    smoothing, BCE does not
    def __init__(self, smoothing=0.0):
        super(LabelSmoothingBCE, self).__init__()
        assert 0.0 <= smoothing < 1.0
        self.smoothing = smoothing
        self.loss_fn = nn.BCEWithLogitsLoss()

    def forward(self, logits, target):
        with torch.no_grad():
            target = target * (1 - self.smoothing) + (self.smoothing /
                2)
        return self.loss_fn(logits, target)


def train_model(lr, train_loader, smoothing, finetune=False, num_epoch
    =10):
    if finetune:
        model = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
    else:
        model = models.resnet18(weights=ResNet18_Weights.DEFAULT)
    model.fc = nn.Linear(model.fc.in_features, 1)  # binary
        classification
    model.to(device)
    criterion = LabelSmoothingBCE(smoothing)
    optimizer = optim.SGD(model.parameters(), lr=lr)
    for epoch in range(num_epoch):
        model.train()
```

```
            running_loss = 0.0
            for i, (images, labels) in enumerate(train_loader):
                images, labels = images.to(device), labels.to(device)
                optimizer.zero_grad()
                outputs = model(images).squeeze()
                loss = criterion(outputs, labels.float())
                loss.backward()
                optimizer.step()
                running_loss += loss.item() * images.size(0)
            epoch_loss = running_loss / len(train_loader.dataset)
            print(f'Epoch {epoch+1}/{num_epoch}, Loss: {epoch_loss:.4f}')
        return model


    def evaluate(model, dataloader, label=''):
        model.eval()
        val_logits = []
        val_labels = []
        with torch.no_grad():
            for images, labels in dataloader:
                images = images.to(device)
                outputs = model(images).cpu().numpy().flatten()
                val_logits.extend(outputs)
                val_labels.extend(labels.numpy())
        val_logits = np.array(val_logits).reshape(-1, 1)
        val_labels = np.array(val_labels)
        return val_logits, val_labels


    def evaluate_calibration_scaling(model, platt_model, test_loader):
        model.eval()
        test_logits = []
        test_labels = []
        with torch.no_grad():
            for image, labels in test_loader:
                image = image.to(device)
                outputs = model(image)
                test_logits.extend(outputs.detach().cpu().numpy().flatten()
                    )
                test_labels.extend(labels.numpy())
            test_logits = np.array(test_logits).reshape(-1, 1)
            calibrated_probs = platt_model.predict_proba(test_logits)[:, 1]
                 # similar to 1 / (1 + np.exp(-A * test_logits - B))
        return test_labels, calibrated_probs
    # Apply platt scaling on models from previous two parts (pretrained &
        scratch) and the one from part 1 (temperature 0.0)
    def platt_scaling(finetune=False, label=''):
        smoothing_values = [0.0, 0.1, 0.2, 0.3]
        results = {}
        for alpha in smoothing_values:
```

```
            if finetune:
                print(f'Finetuning with alpha value: {alpha}')
            else:
                print(f'Training from scratch with alpha value: {alpha}')
            plt.figure(figsize=(8, 6))
            model = train_model(0.001, train_loader, alpha, finetune=
                finetune)
            # Apply platt scaling
            val_logits, val_labels = evaluate(model, val_loader)
            platt_model = LogisticRegression(solver='lbfgs')
            platt_model.fit(val_logits, val_labels)
            test_labels, test_logits = evaluate_calibration_scaling(model,
                platt_model, test_loader)
            prob_true_cal, prob_pred_cal = calibration_curve(test_labels,
                test_logits, n_bins=10)
            results[int(alpha * 10)]={
                'prob_true': prob_true_cal,
                'prob_pred': prob_pred_cal,
            }

        # Plot after scaling
        plt.figure(figsize=(8, 6))
        for alpha in smoothing_values:
            prob_true = results[int(alpha * 10)]['prob_true']
            prob_pred = results[int(alpha * 10)]['prob_pred']
            plt.plot(prob_pred, prob_true, marker='o', label=f'Label
                Smoothing: {alpha}')
        plt.plot([0, 1], [0, 1], linestyle='--', color='black')  # Draw
            diagonal
        plt.title(f'Reliability with label smoothing -- {label}')
        plt.xlabel('Predicted probability')
        plt.ylabel('Fraction of Positives')
        plt.legend()
        plt.show()
    #%%
    # Execute
    platt_scaling(False, 'Scratch')
    platt_scaling(True, 'Pretrained')
```

Problem 2 - Conformal Prediction (18 points)

Conformal prediction (CP) is a framework used in machine learning and statistical prediction to provide reliable and well-calibrated measures of uncertainty for individual predictions. It was developed to address the limitations of traditional confidence intervals and prediction intervals, which may not always accurately capture the true uncertainty associated with a prediction.

In CP, instead of providing a single point prediction, a set of predictions is generated for each data point. These prediction sets are constructed in such a way that they have a guaranteed level of coverage (often set by the user) over a set of future unseen data points. This means that the prediction sets are designed to capture the true target value with a certain probability.

CP accomplishes this by using a calibration set (with length n), which contains examples that the model has not seen during training, to estimate the uncertainty associated with different prediction outcomes. It computes prediction sets $\tau(X_{n+1})$ that are expected to contain the true target value Y_{n+1} with a specified level of confidence $(1 - \alpha)$.

$$1 - \alpha \leq \mathbb{P}\left[Y_{n+1} \in \tau(X_{n+1})\right] \leq 1 - \alpha + \frac{1}{n+1} \tag{1}$$

The coverage theorem (Eq.1) in conformal prediction is a fundamental theorem that provides a guarantee of the reliability of prediction intervals generated by a conformal predictor. The coverage theorem relies on certain assumptions and conditions, such as exchangeability (the order of the data points should not affect the result of the data and correctness of the conformal predictor), and correctness (the conformal predictor should be calibrated properly).

CP can be applied to any machine learning or statistical model, making it a versatile tool for quantifying uncertainty in various domains, including classification, regression, and anomaly detection .

Algorithms Overview

In conformal prediction for any type of input (X) and output (Y), we start by estimating how uncertain our model is using a pre-trained model. We define a score function $(s(X, Y))$ that tells us how well the model's predictions match the actual outcomes (higher scores mean worse predictions). Next, we calculate a special value called the "quantile" $(\hat{q})$ based on these scores. This $\hat{q}$ helps us create prediction sets for new examples, giving us a way to understand how confident or uncertain our model is in its predictions.

Problem Set:

To ease the your burden, we trained a ResNet model on the Oxford-IIIT Pet Dataset on your behalf. By using that model we generated you softmax outputs and correct classes of the test dataset. Also, we provided a small set of images and their softmax outputs ("model_outputs.npy"), class id to name map ("idx2cls.npy"), and gt labels ("gt_classes.npy") for visualization purposes.

For this part we want you to implement both naive and adaptive prediction sets algorithms.

Part 1: Naive Algorithm

Score function:

$$s(X, Y) = \hat{f}(X)_Y \tag{2}$$

$$1 - \hat{q} = Quantile(s_1, \cdots, s_n; 1 - \frac{\lceil (n+1)(1-\alpha) \rceil}{n}) \tag{3}$$

Use Eq. 3 to calculate 1-quantile

$$\tau(X) = \{\pi_1(X), \cdots, \pi_k(X)\}, \text{ where } k = sup\{m : \hat{f}(X)_Y < 1 - \hat{q}\} + 1 \tag{4}$$

Use Eq.4 to construct prediction set.

Your Task: Implement the Naive algorithm (5 points)

1. Read 'softmax_outputs.npy' and 'correct_classes.npy' files. These files contain softmax outputs and the correct class label of the same output on the same array index.
2. Split the given data into calibration and validation datasets by putting the first $n = 2000$ elements in the calibration dataset and the rest into the validation dataset.
3. Use the calibration data to set up the naive algorithm.
4. Measure the empirical coverage of the produced prediction sets on the validation data. Report the coverage.
5. Check the algorithm on the example dataset which has images and .npy files with the same format. Show images with their labels and prediction sets; include this part in the report.

Report:

Empirical Coverage and Predictions on Example Images Using Naive Prediction:

By setting the significance level α to 0.05. Below is the report of empirical coverage of the produced prediction sets on the validation data and model's "guaranteed" prediction on the example images B0, B1, B2, B3, B15, B17, and B19.

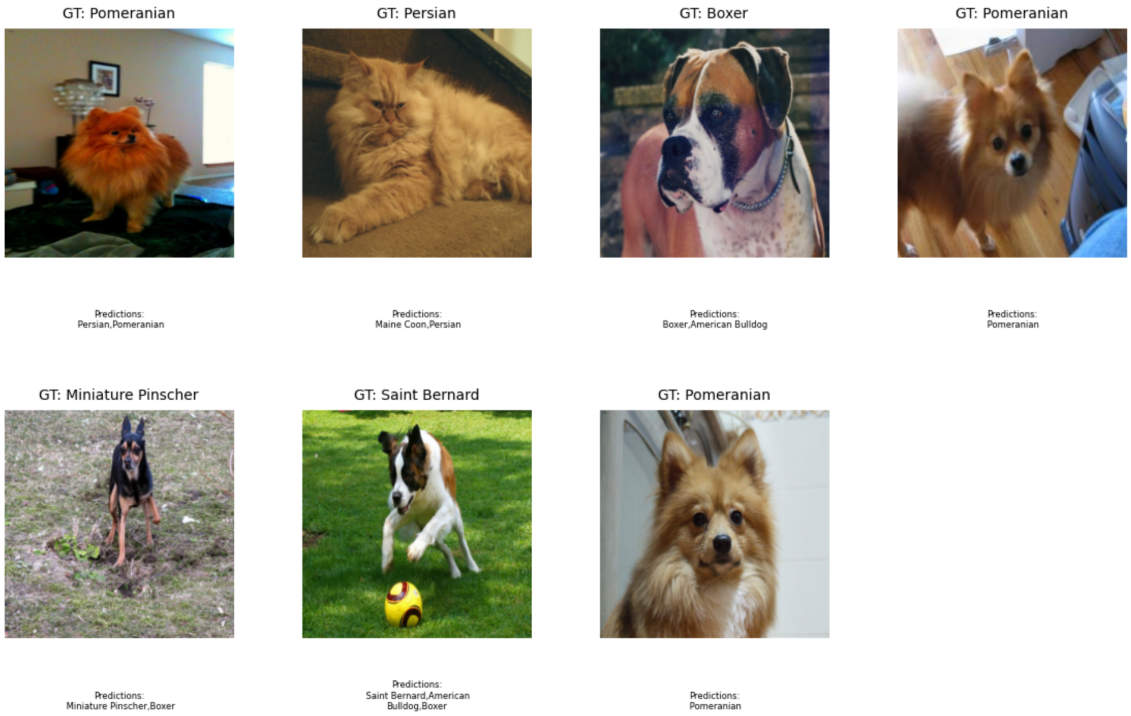


Figure 3: Coverage and Prediction Sets Using Naive Prediction

Part 2: Adaptive Prediction Sets Algorithm

Score function:

$$s(X, Y) = \sum_{j=1}^{k} \hat{f}(X)_{\pi_j(X)}, \text{ and } \pi(x)_k = Y \tag{5}$$

where $\pi(x)$ is the permutation of $\{1, ..., K\}$ that sorts $\hat{f}(X)$ from most likely to least likely.

$$\hat{q} = Quantile(s_1, \cdots, s_n; \frac{\lceil (n+1)(1-\alpha) \rceil}{n}) \tag{6}$$

Use Eq. 6 to calculate quantile

$$\tau(X) = \{\pi_1(X), \cdots, \pi_k(X)\}, \text{ where } k = sup\{m : \sum_{m}^{j=1} \hat{f}(X)_{\pi_j(X)} < \hat{q}\} + 1 \tag{7}$$

Use Eqs.7 to construct prediction set.

Your Task: Implement the adaptive prediction sets algorithm (5 points)

1. Read 'softmax_outputs.npy' and 'correct_classes.npy' files. These files contain softmax outputs and the correct class label of the same output on the same array index.
2. Split the given data into calibration and validation datasets by putting the first $n = 2000$ elements in the calibration dataset and the rest into the validation dataset.
3. Use the calibration data to set up the adaptive prediction sets algorithm.
4. Measure the empirical coverage of the produced prediction sets on the validation data. Report the coverage.
5. Check the algorithm on the example dataset which has images and .npy files with the same format. Show images with their labels and prediction sets; include this part in the report.

Report:

Empirical Coverage and Predictions on Example Images Using Adaptive Prediction:

By setting the significance level α to 0.1. Below is the report of empirical coverage of the produced prediction sets on the validation data and model's "guaranteed" prediction on the example images B0, B1, B2, B3, B15, B17, and B19.

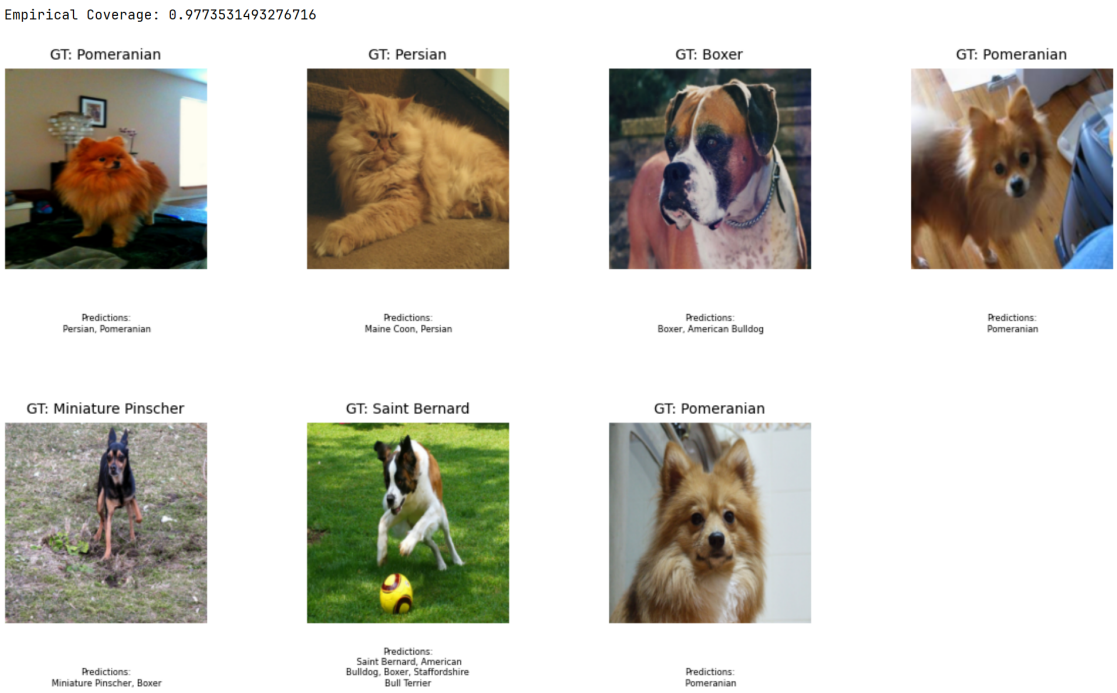


Figure 4: Coverage and Prediction Sets Using Adaptive Prediction

Part 3: Comparison of Part 1 and Part 2 Results (3ponts)

Compare visual results, and comment on algorithms; discuss potential improvements to the process .

Answer:

I ran a few experiments and found that only notable different is observed when using alpha value equals 0.5 (other values I've tried are $[0.1, 0.01]$). Difference in visual results is still small. Among picked example images, only Saint Bernard made the two prediction methods vary. Specifically, Adaptive Prediction produced more possible outputs which can be regards as incompetence in the softmax outputs. There are a few ways I think to improve the process:

1. Tune the hyperparameter α to better limit the size of my prediction set
2. We can refine the scoring method such as using weighted cumulative sum or perform calibration directly onto the prediction set to limit the size
3. Consider adding uncertainty estimate as part of the score function.

Part 4: (5 points)

For this exercise, you will explore how the nature of prediction sets changes based on the heuristic scores used to generate them.

1. Instead of using the model's softmax scores, generate random scores for each class for each input in your validation dataset. (Just replace validation data with a random array with values between [0,1].) Use these random scores as your heuristic and generate prediction sets. Evaluate these prediction sets and note down the coverage.
2. Compare the results from the all approaches. Discuss the differences in prediction set quality, coverage, and any other observed phenomena. What do these results tell you about the importance of the heuristic function in generating prediction sets?

Report your code and answers to verbal questions.

Report:

Despite the various alpha value tested, two methods' coverage are both nearly zero. Suggesting extremely low accuracy in generated prediction sets. The result clearly illustrate the importance of a meaningful heuristic score function: since random score do not carry any meaningful information and thus resulted in a high quantile threshold.

```
Quantile (q_hat) using random scores: 0.9817
Empirical Coverage (Naive with Random Scores): 0.0354
Prediction Set Size (Naive with Random Scores): 2
Empirical Coverage (Adaptive with Random Scores): 0.0354
Prediction Set Size (Adaptive with Random Scores): 2
```

Figure 5: Quantile, Coverage, and Prediction Set Size comparison

Discussion:

Listing 2: Code Implementation for P2P4

```python
import numpy as np

# Data preparation
softmax_outputs = np.load('softmax_outputs.npy')
correct_classes = np.load('correct_classes.npy')

# Data Split
split = 2000
calibration_softmax = softmax_outputs[:split]
calibration_labels = correct_classes[:split]
validation_softmax = softmax_outputs[split:]
validation_labels = correct_classes[split:]

random_scores = np.random.rand(*validation_softmax.shape)

def construct_naive_pred_set(random_output, q_hat):
    sorted_indices = np.argsort(random_output)[::-1]
    cumulative_sum = 0
    pred_set = []
    for idx in sorted_indices:
        cumulative_sum += random_output[idx]
        pred_set.append(idx)
        if cumulative_sum >= q_hat:
            break
    return pred_set

def construct_adaptive_pred_set(random_output, q_hat):
    sorted_indices = np.argsort(random_output)[::-1]
    cumulative_sum = 0
    pred_set = []
    for idx in sorted_indices:
        cumulative_sum += random_output[idx]
```

```
            pred_set.append(idx)
            if cumulative_sum >= q_hat:
                break
        return pred_set

    # Compute Random Scores
    random_calibration_scores = []
    for i in range(len(calibration_softmax)):
        random_output = np.random.rand(calibration_softmax.shape[1])
        sorted_probs = np.sort(random_output)[::-1]
        cumulative_sum = np.cumsum(sorted_probs)
        score = cumulative_sum[np.argmax(sorted_probs)]
        random_calibration_scores.append(score)

    # Compute quantile
    alpha = 0.5
    q_hat_random = np.quantile(random_calibration_scores, 1 - alpha)
    print(f"Quantile (q_hat) using random scores: {q_hat_random:.4f}")

    # Evaluate the empirical coverage
    correct_naive = 0
    correct_adaptive = 0
    for i in range(len(validation_softmax)):
        random_output = random_scores[i]

        naive_pred_set = construct_naive_pred_set(random_output,
            q_hat_random)
        if validation_labels[i] in naive_pred_set:
            correct_naive += 1

        adaptive_pred_set = construct_adaptive_pred_set(random_output,
            q_hat_random)
        if validation_labels[i] in adaptive_pred_set:
            correct_adaptive += 1

    empirical_coverage_naive_random = correct_naive / len(
        validation_softmax)
    empirical_coverage_adaptive_random = correct_adaptive / len(
        validation_softmax)
    print(f"Empirical Coverage (Naive with Random Scores): {
        empirical_coverage_naive_random:.4f}")
    print(f'Prediction Set Size (Naive with Random Scores): {len(
        naive_pred_set)}')
    print(f"Empirical Coverage (Adaptive with Random Scores): {
        empirical_coverage_adaptive_random:.4f}")
    print(f'Prediction Set Size (Adaptive with Random Scores): {len(
        adaptive_pred_set)}')
```

1 Appendix

Listing 3: Code Implementation for P1P1

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import models, transforms
from torchvision.models import ResNet18_Weights

from Model_calibration_dataset import get_cifar10_classes
import matplotlib.pyplot as plt
from sklearn.calibration import calibration_curve
from sklearn.linear_model import LogisticRegression
#%%
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010))
])
train_dataset, val_dataset, test_data = get_cifar10_classes(
    transform_train, transform_test)

# Create dataloader
batch_size=128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle
    =True, num_workers=2)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=
    False, num_workers=2)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=
    False, num_workers=2)

# Define the model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = models.resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
model.fc = nn.Linear(model.fc.in_features, 1)  # binary classification
model.to(device)
#%%
# Train
lr = 0.001
criterion = nn.BCEWithLogitsLoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=lr)
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images).squeeze()
        loss = criterion(outputs, labels.float())
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)
    epoch_loss = running_loss / len(train_loader.dataset)
    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}')
#%%
# Manually implement platt scaling

#%%
# Draw reliability curve on the test data before scaling
model.eval()
all_probs = []
all_labels = []
with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        outputs = model(images)
        probs = torch.sigmoid(outputs).cpu().numpy().flatten()
        all_probs.extend(probs)
        all_labels.extend(labels.numpy())
all_probs = np.array(all_probs)
all_labels = np.array(all_labels)
#%%

prob_true, prob_pred = calibration_curve(all_labels, all_probs, n_bins
    =10)
plt.figure(figsize=(8, 6))
plt.plot(prob_pred, prob_true, marker='o', label='Uncalibrated')
plt.plot([0, 1], [0, 1], linestyle='--', label='Perfect Calibration')
plt.title('Reliability Curve (Before Calibration)')
plt.xlabel('Mean Predicted Probability')
plt.ylabel('Fraction of Positives')
plt.legend()
plt.show()
#%%
# Implement Platt Scaling on outputs on val set
model.eval()
val_logits = []
val_labels = []
with torch.no_grad():
```

```
        for images, labels in val_loader:
            images = images.to(device)
            outputs = model(images).cpu().numpy().flatten()
            val_logits.extend(outputs)
            val_labels.extend(labels.numpy())
    val_logits = np.array(val_logits).reshape(-1, 1)
    val_labels = np.array(val_labels)

    platt_model = LogisticRegression(solver='lbfgs')
    platt_model.fit(val_logits, val_labels)
    A, B = platt_model.coef[0][0], platt_model.intercept_[0]  # Retrieve
        values of A and B
    #%%
    # Draw reliability curve on test data after scaling
    test_logits = []
    with torch.no_grad():
        for images, _ in test_loader:
            images = images.to(device)
            outputs = model(images)
            test_logits.extend(outputs.detach().cpu().numpy().flatten())
    test_logits = np.array(test_logits).reshape(-1, 1)
    calibrated_probs = platt_model.predict_proba(test_logits)[:, 1]  #
        similar to 1 / (1 + np.exp(-A * test_logits - B))

    prob_true_cal, prob_pred_cal = calibration_curve(all_labels,
        calibrated_probs, n_bins=10)
    plt.figure(figsize=(8, 6))
    plt.plot(prob_pred_cal, prob_true_cal, marker='o', label='Calibrated')
    plt.plot([0, 1], [0, 1], linestyle='--', label='Perfect Calibration')
    plt.title('Reliability Curve (After Platt Scaling)')
    plt.xlabel('Mean Predicted Probability')
    plt.ylabel('Fraction of Positives')
    plt.legend()
    plt.show()
    #%%

    #%%
    # Draw a combined curve
    plt.figure(figsize=(8, 6))
    plt.plot(prob_pred, prob_true, marker='o', label='Uncalibrated')
    plt.plot(prob_pred_cal, prob_true_cal, marker='o', label='Calibrated')
    plt.plot([0, 1], [0, 1], linestyle='--', label='Perfect Calibration')
    plt.title('Reliability Curve (Before and After Calibration)')
    plt.xlabel('Mean Predicted Probability')
    plt.ylabel('Fraction of Positives')
    plt.legend()
    plt.show()
```

Listing 4: Code Implementation for P2P1

```
import os
import numpy as np
from Cryptodome.PublicKey.DSA import construct
from PIL import Image
from matplotlib import pyplot as plt
import textwrap
#%%
## Part 1 -- Use Naive Prediction

# Data preparation
softmax_outputs = np.load('softmax_outputs.npy')
correct_classes = np.load('correct_classes.npy')

# Data Split
split = 2000
calibration_softmax = softmax_outputs[:split]
calibration_labels = correct_classes[:split]
validation_softmax = softmax_outputs[split:]
validation_labels = correct_classes[split:]

# Get score
scores = np.array([calibration_softmax[i, calibration_labels[i]] for i
    in range(split)])  # Retrieve the softmax score of the correct class
     for each example

# Calculate 1-quantile (one-tailed q?)
alpha = 0.5
q_hat = np.quantile(scores, 1-alpha)

# Construct prediction set that have cumulative score >= q_hat
def construct_prediction_set(softmax_output, q_hat):
    stored_indices = np.argsort(softmax_output)[::-1]  # Sort class by
        scores
    cumulative_score = 0
    predictions = []
    for idx in stored_indices:
        cumulative_score += softmax_output[idx]
        predictions.append(idx)
        if cumulative_score >= q_hat:
            break
    return predictions

# Evaluate empirical coverage
cor_preds = 0
for i in range(len(validation_softmax)):
    prediction_set = construct_prediction_set(validation_softmax[i],
        q_hat)
    if validation_labels[i] in prediction_set:
        cor_preds += 1
empirical_coverage = cor_preds / len(validation_softmax)
```

```
print(f'Empirical Coverage: {empirical_coverage}')

# Check naive prediction on example images
folder_path = 'example_images'
example_outputs = np.load(os.path.join(folder_path, 'model_outputs.npy'
    ))
gt_classes = np.load(os.path.join(folder_path, 'gt_classes.npy'))
idx2cls = np.load(os.path.join(folder_path, 'idx2cls.npy'),
    allow_pickle=True)
cls2idx = {v: k for k, v in enumerate(idx2cls)}

selected_images = ['B0.png', 'B1.png', 'B2.png', 'B3.png', 'B15.png', '
    B17.png', 'B19.png']
selected_indices = [0,1,2,3,15,17,19]
# Recalculate q_hat
example_scores = np.array([example_outputs[i, cls2idx[gt_classes[i]]]
    for i in range(len(example_outputs))])
#q_hat = np.quantile(example_scores, 1-alpha)
plt.figure(figsize=(14, 8))
for i, (image_name, idx) in enumerate(zip(selected_images,
    selected_indices)):
    image = Image.open(os.path.join(folder_path, image_name))
    gt_label = gt_classes[idx]
    example_output = example_outputs[idx]
    # Construct prediction set
    pred_set = construct_prediction_set(example_output, q_hat)
    pred_set_names = [idx2cls[i] for i in pred_set]

    # Plot image
    plt.subplot(2, 4, i+1)
    plt.imshow(image)
    plt.axis('off')
    plt.text(0.5, 1.05, f'GT: {gt_label}', fontsize=10, ha='center',
        wrap=True, transform=plt.gca().transAxes)
    pred_texts = ','.join(pred_set_names)
    wrapped_text = "\n".join(textwrap.wrap(pred_texts, 30))
    plt.text(0.5, -0.3, f'Predictions:\n {wrapped_text}', fontsize=6,
        ha='center', wrap=True, transform=plt.gca().transAxes)

plt.subplots_adjust(wspace=0.3, hspace=0.5)
plt.show()
```

Listing 5: Code Implementation for P2P2

```
import os
import textwrap

import matplotlib.pyplot as plt
import numpy as np
from functorch.dim import softmax
```

```
from PIL import Image
#%%
## Part 2 -- Use Adaptive Prediction
# Data preparation
softmax_outputs = np.load('softmax_outputs.npy')
correct_classes = np.load('correct_classes.npy')

# Data Split
split = 2000
calibration_softmax = softmax_outputs[:split]
calibration_labels = correct_classes[:split]
validation_softmax = softmax_outputs[split:]
validation_labels = correct_classes[split:]

def calculate_scores(model_outputs, true_labels, gt_class=None):
    scores = []
    for i in range(len(model_outputs)):
        softmax_output = model_outputs[i]
        if gt_class is not None:
            true_class_prob = softmax_output[true_labels[gt_class[i]]]
        else:
            true_class_prob = softmax_output[true_labels[i]]
        sorted_probs = np.sort(softmax_output)[::-1]
        cumulative_sum = np.cumsum(sorted_probs)
        k = np.argmax(sorted_probs == true_class_prob)
        score = cumulative_sum[k]
        scores.append(score)
    return scores

def construct_adaptive_pred_set(softmax_output, q_hat):
    sorted_indices = np.argsort(softmax_output)[::-1]
    cumulative_sum = 0
    pred_set = []
    for idx in sorted_indices:
        cumulative_sum += softmax_output[idx]
        pred_set.append(idx)
        if cumulative_sum >= q_hat:
            break
    return pred_set

# Get coverage
alpha=0.5
calibration_scores = calculate_scores(calibration_softmax,
    calibration_labels)
q_hat = np.quantile(calibration_scores, 1-alpha)
cor_preds = 0
for i in range(len(validation_softmax)):
    pred_set = construct_adaptive_pred_set(validation_softmax[i], q_hat
        )
    if validation_labels[i] in pred_set:
```

```
            cor_preds += 1
    empirical_coverage = cor_preds / len(validation_softmax)
    print(f'Empirical Coverage: {empirical_coverage}')

    # Get prediction from example images
    folder_path = 'example_images'
    example_outputs = np.load(os.path.join(folder_path, 'model_outputs.npy'
        ))
    gt_classes = np.load(os.path.join(folder_path, 'gt_classes.npy'))
    idx2cls = np.load(os.path.join(folder_path, 'idx2cls.npy'),
        allow_pickle=True)
    cls2idx = {v: k for k, v in enumerate(idx2cls)}

    # Selected images and indices
    selected_images = ['B0.png', 'B1.png', 'B2.png', 'B3.png', 'B15.png', '
        B17.png', 'B19.png']
    selected_indices = [0, 1, 2, 3, 15, 17, 19]

    example_scores = calculate_scores(example_outputs, cls2idx, gt_classes)
    q_hat = np.quantile(example_scores, 1-alpha)

    # Plot
    plt.figure(figsize=(14, 8))
    for i, (idx, image_name) in enumerate(zip(selected_indices,
        selected_images)):
        softmax_output = example_outputs[idx]
        true_class = gt_classes[idx]
        image = Image.open(os.path.join(folder_path, image_name))
        pred_set = construct_adaptive_pred_set(softmax_output, q_hat)
        pred_class_names = [idx2cls[idx] for idx in pred_set]
        pred_texts = ', '.join(pred_class_names)
        wrapped_pred_texts = '\n'.join(textwrap.wrap(pred_texts, 30))

        ax = plt.subplot(2, 4, i+1)
        ax.imshow(image)
        ax.axis('off')
        plt.text(0.5, 1.05, f'GT: {true_class}', fontsize=10, ha='center',
            wrap=True, transform=plt.gca().transAxes)
        plt.text(0.5, -0.3, f'Predictions:\n {wrapped_pred_texts}',
            fontsize=6, ha='center', wrap=True, transform=plt.gca().
            transAxes)

    plt.subplots_adjust(hspace=0.3, wspace=0.5)
    plt.show()
```