

```

1
2 import argparse
3 import os
4 from datetime import datetime
5 from pathlib import Path
6 from collections import Counter, defaultdict
7 import json
8 import pickle
9
10 import numpy as np
11 import torch
12 import torchvision
13 import matplotlib.pyplot as plt
14 from torch import nn
15 from torchvision import transforms
16 from torchinfo import summary
17 from tqdm import tqdm
18 from scipy.stats import pearsonr, spearmanr
19 import torch.nn.functional as F
20 from torch.nn.functional import softmax
21 import numpy
22
23 # =====
24 # Parsing the arguments-----
25 # =====
26 parser = argparse.ArgumentParser()
27 parser.add_argument("--test_run", type=int, required=True, help="Experiment run
index")
28 parser.add_argument("--vit_ckpt_path", type=str, required=True, help="Path to saved
ViT checkpoint (.pth)")
29 parser.add_argument(
30     "--data_dir",
31     type=str,
32     required=True,
33     help="Path to dataset root containing train/, val/, and test/ folders"
34 )
35 args = parser.parse_args()
36
37 test_run = args.test_run
38 vit_ckpt_path = args.vit_ckpt_path
39
40 # =====
41 # Device configuration
42 # =====
43 device = "cuda" if torch.cuda.is_available() else "cpu"
44 print("#####")
45 print(f"Vision Transformer Conformal Prediction For Test Run: {test_run}")
46 print(f"The device for running the experiment is {device}")
47 print("#####")
48
49 # Basic version prints
50 print(f"torch version: {torch.__version__}")
51 print(f"torchvision version: {torchvision.__version__}")
52
53 # =====
54 # Dataset paths and summary
55 # =====
56 DATA_DIRECTORY = args.data_dir #f"/leonardo_work/IscrC_SKIDD-
AI/alitariqnagi_work/training_data_split_run_{test_run}"
57 main_folder = DATA_DIRECTORY

```

```

58 splits = ['train', 'test', 'val']
59
60 summary_dict = {}
61 for split in splits:
62     split_path = os.path.join(main_folder, split)
63     total_files = 0
64     summary_dict[split] = {}
65
66     if not os.path.exists(split_path):
67         print(f"Directory not found: {split_path}")
68         continue
69
70     for disease in os.listdir(split_path):
71         disease_path = os.path.join(split_path, disease)
72         if not os.path.isdir(disease_path):
73             continue
74
75         num_files = len([
76             f for f in os.listdir(disease_path)
77             if os.path.isfile(os.path.join(disease_path, f))
78         ])
79
80         summary_dict[split][disease] = num_files
81         total_files += num_files
82
83         print(f"\n{split.upper()} - Total files: {total_files}")
84         for disease, count in summary_dict[split].items():
85             print(f"    {disease}: {count} files")
86
87 train_dir = f"{DATA_DIRECTORY}/train"
88 val_dir   = f"{DATA_DIRECTORY}/val"
89 test_dir  = f"{DATA_DIRECTORY}/test"
90
91 # =====
92 # Data transforms and loaders
93 # =====
94 IMG_SIZE = 224
95 manual_transforms = transforms.Compose([
96     transforms.Resize((IMG_SIZE, IMG_SIZE)),
97     transforms.ToTensor(),
98 ])
99 print(f"Manually created transforms: {manual_transforms}")
100
101 from going_modular import data_setup
102 from going_modular.helper_functions import set_seeds
103 set_seeds()
104
105 BATCH_SIZE = 32
106
107 # Create an initial dataloader to get class_names
108 train_dataloader, _, class_names = data_setup.create_dataloaders(
109     train_dir=train_dir,
110     test_dir=val_dir,
111     transform=manual_transforms,
112     batch_size=BATCH_SIZE
113 )
114
115 print(f"The class names are: {class_names}")
116
117 # =====

```

```

118 # ViT model loading checkpoint
119 # =====
120 pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT
121 pretrained_vit =
122     torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)
123 # Replace classifier head
124 pretrained_vit.heads = nn.Linear(in_features=768,
125     out_features=len(class_names)).to(device)
126
127 summary(model=pretrained_vit,
128     input_size=(32, 3, 224, 224),
129     col_names=["input_size", "output_size", "num_params", "trainable"],
130     col_width=20,
131     row_settings=["var_names"])
132 # Get transforms from ViT weights
133 pretrained_vit_transforms = pretrained_vit_weights.transforms()
134 print(pretrained_vit_transforms)
135
136 from going_modular.data_setup import create_dataloaders_no_shuffle
137 val_dataloader_pretrained, test_dataloader_pretrained, class_names =
138     create_dataloaders_no_shuffle(
139         train_dir=val_dir,
140         test_dir=test_dir,
141         transform=pretrained_vit_transforms,
142         batch_size=32
143     )
144 # =====
145 # Load saved ViT checkpoint
146 # =====
147 print(f"[INFO] Loading pretrained ViT weights from: {vit_ckpt_path}")
148
149 # Handling DataParallel if multiple GPUs
150 if torch.cuda.device_count() > 1:
151     print(f"[INFO] Using DataParallel on {torch.cuda.device_count()} GPUs")
152     pretrained_vit = torch.nn.DataParallel(pretrained_vit)
153
154 pretrained_vit.to(device)
155
156 state_dict = torch.load(vit_ckpt_path, map_location=device)
157 try:
158     pretrained_vit.load_state_dict(state_dict)
159 except RuntimeError as e:
160     print("[WARNING] Error loading state_dict, possibly DataParallel/non-DataParallel
161     mismatch.")
162     print("Error was:", e)
163     raise
164
165 pretrained_vit.eval()
166 print("[INFO] ViT checkpoint loaded successfully.\n")
167 # =====
168 # Conformal prediction functions (unchanged)
169 # =====
170
171 # alpha will be overridden with different values in the loop
172 alpha = 0.1
173

```

```

174 def calibration_scores(model, data_loader, device):
175     model.eval()
176     scores_calibration_set = []
177
178     with torch.no_grad():
179         for X, y in tqdm(data_loader,
180                         desc="#####Proceeding with the Calculation of
Calibration Scores#####"):
181             X = X.to(device)
182             y = y.to(device)
183
184             y = y.cpu().numpy()
185             pred_logits = model(X)
186             cal_smx = softmax(pred_logits, dim=1).cpu().numpy()
187             scores_per_batch = 1 - cal_smx[np.arange(y.shape[0]), y]
188             scores_calibration_set.extend(scores_per_batch)
189
190     scores_calibration_set = numpy.array(scores_calibration_set)
191     return scores_calibration_set
192
193
194 def predict_with_conformal_sets(model, data_loader, calibration_scores_array, alpha,
device):
195     model.eval()
196     prediction_sets = []
197     true_labels = []
198
199     n = len(calibration_scores_array)
200     q_level = np.ceil((n + 1) * (1 - alpha)) / n
201     qhat = np.quantile(calibration_scores_array, q_level, method='higher')
202
203     with torch.no_grad():
204         for X, y in tqdm(data_loader,
205                         desc="#####Proceeding with the Calculation of
Testing Scores#####"):
206             X = X.to(device)
207             y = y.cpu().numpy()
208             pred_logits = model(X)
209
210             val_smx = softmax(pred_logits, dim=1).cpu().numpy()
211             masks = val_smx >= (1 - qhat)
212
213             for m in masks:
214                 prediction_sets.append(np.where(m)[0].tolist())
215
216             true_labels.extend(y.tolist())
217
218     return prediction_sets, true_labels
219
220
221 def evaluate(prediction_sets, labels):
222     number_of_correct_predictions = 0
223     for pred_set, true_label in zip(prediction_sets, labels):
224         if true_label in pred_set:
225             number_of_correct_predictions += 1
226     empirical_test_coverage = number_of_correct_predictions / len(labels)
227     average_prediction_set_size = np.mean([len(s) for s in prediction_sets])
228     return empirical_test_coverage, average_prediction_set_size
229
230

```

```

231 def feature_stratified_coverage_by_class(prediction_sets, true_labels, alpha=0.1,
class_names=None):
232     class_to_indices = defaultdict(list)
233     for idx, label in enumerate(true_labels):
234         class_to_indices[label].append(idx)
235
236     class_coverages = {}
237     for cls, indices in class_to_indices.items():
238         covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
239         coverage = covered / len(indices)
240         class_coverages[cls] = coverage
241
242     print("\n Feature-Stratified Coverage (by True Class Label):")
243     for cls, cov in sorted(class_coverages.items()):
244         name = class_names[cls] if class_names else str(cls)
245         print(f" Class {name:<20}: Coverage = {cov:.3f} (Target ≥ {1 - alpha:.2f})")
246
247     fsc_metric = min(class_coverages.values())
248     print(f"\n FSC Metric (minimum class-wise coverage): {fsc_metric:.3f}")
249     return fsc_metric, class_coverages
250
251
252 def plot_class_wise_coverage(class_coverages, alpha=0.1, class_names=None,
title="Class-wise Coverage (FSC)":
253     classes = list(class_coverages.keys())
254     coverages = [class_coverages[c] for c in classes]
255     labels = [class_names[c] if class_names else str(c) for c in classes]
256
257     plt.figure(figsize=(12, 6))
258     bars = plt.bar(labels, coverages, color="pink", edgecolor="black")
259
260     target = 1 - alpha
261     plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage
({target:.2f})")
262
263     for bar, cov in zip(bars, coverages):
264         if cov < target:
265             bar.set_color('salmon')
266
267     plt.xticks(rotation=45, ha="right")
268     plt.ylim(0, 1.05)
269     plt.ylabel("Coverage")
270     plt.title(title)
271     plt.legend()
272     plt.tight_layout()
273     filename = f"Class_wise_Coverage_(FSC)_{test_run}_{timestamp}.png"
274     plt.savefig(os.path.join(results_directory, filename))
275     plt.show()
276
277
278 def plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=0.1,
class_names=None,
279                                     title="Coverage vs Class Frequency"):
280     class_indices = sorted(class_coverages.keys())
281     coverages = [class_coverages[c] for c in class_indices]
282
283     total = len(true_labels)
284     counts = Counter(true_labels)
285     frequencies = [counts[c] / total for c in class_indices]
286

```

```

287 labels = [class_names[c] if class_names else str(c) for c in class_indices]
288 x = np.arange(len(class_indices))
289
290 fig, ax1 = plt.subplots(figsize=(12, 6))
291
292 bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
293 edgecolor='black')
294 ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
295 alpha:.2f})")
296 ax1.set_ylim(0, 1.05)
297 ax1.set_ylabel("Coverage", color='pink')
298 ax1.tick_params(axis='y', labelcolor='pink')
299
300 for bar, cov in zip(bars1, coverages):
301     if cov < (1 - alpha):
302         bar.set_color('salmon')
303
304 ax2 = ax1.twinx()
305 bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
306 color='gray', alpha=0.5)
307 ax2.set_ylabel("Class Frequency", color='gray')
308 ax2.tick_params(axis='y', labelcolor='gray')
309
310 plt.xticks(x, labels, rotation=45, ha="right")
311 plt.title(title)
312 fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
313 plt.tight_layout()
314 filename = f"plot_coverage_vs_class_frequency_{test_run}_{timestamp}.png"
315 plt.savefig(os.path.join(results_directory, filename))
316 plt.show()
317
318 def plot_coverage_vs_class_frequency_with_correlation(class_coverages, true_labels,
319 alpha=0.1, class_names=None,
320
321 title="Coverage vs Class
322 Frequency"):
323     class_indices = sorted(class_coverages.keys())
324     coverages = [class_coverages[c] for c in class_indices]
325
326     total = len(true_labels)
327     counts = Counter(true_labels)
328     frequencies = [counts[c] / total for c in class_indices]
329
330     labels = [class_names[c] if class_names else str(c) for c in class_indices]
331     x = np.arange(len(class_indices))
332
333     pearson_corr, pearson_p = pearsonr(frequencies, coverages)
334     spearman_corr, spearman_p = spearmanr(frequencies, coverages)
335
336     print(f"\n Correlation between class frequency and coverage:")
337     print(f" - Pearson r = {pearson_corr:.3f}, p = {pearson_p:.3e}")
338     print(f" - Spearman p = {spearman_corr:.3f}, p = {spearman_p:.3e}")
339
340     fig, ax1 = plt.subplots(figsize=(12, 6))
341
342     bars1 = ax1.bar(x - 0.2, coverages, width=0.4, label="Coverage", color='pink',
343 edgecolor='black')
344 ax1.axhline(1 - alpha, color='red', linestyle='--', label=f"Target Coverage ({1 -
345 alpha:.2f})")
346 ax1.set_ylim(0, 1.05)

```

```

340 ax1.set_ylabel("Coverage", color='pink')
341 ax1.tick_params(axis='y', labelcolor='pink')
342
343 for bar, cov in zip(bars1, coverages):
344     if cov < (1 - alpha):
345         bar.set_color('salmon')
346
347 ax2 = ax1.twinx()
348 bars2 = ax2.bar(x + 0.2, frequencies, width=0.4, label="Class Frequency",
color='gray', alpha=0.5)
349 ax2.set_ylabel("Class Frequency", color='gray')
350 ax2.tick_params(axis='y', labelcolor='gray')
351
352 plt.xticks(x, labels, rotation=45, ha="right")
353 plt.title(title)
354 fig.legend(loc='upper right', bbox_to_anchor=(0.85, 0.85))
355 plt.tight_layout()
356 filename =
f"plot_coverage_vs_class_frequency_with_correlation_{test_run}_{timestamp}.png"
357 plt.savefig(os.path.join(results_directory, filename))
358 plt.show()
359
360
361 def size_stratified_coverage(prediction_sets, true_labels, alpha=0.1,
362                             bins=[[1], [2], [3, 4, 5], list(range(6, 100))]):
363     bin_groups = defaultdict(list)
364
365     for i, pred_set in enumerate(prediction_sets):
366         set_size = len(pred_set)
367         for bin_range in bins:
368             if set_size in bin_range:
369                 bin_groups[str(bin_range)].append(i)
370                 break
371
372     bin_coverages = {}
373     for bin_key, indices in bin_groups.items():
374         covered = sum(true_labels[i] in prediction_sets[i] for i in indices)
375         coverage = covered / len(indices) if indices else 0
376         bin_coverages[bin_key] = coverage
377
378     print("\n Size-Stratified Coverage (SSC):")
379     for bin_key, cov in bin_coverages.items():
380         print(f" Set Size {bin_key:10s}: Coverage = {cov:.3f} (Target ≥ {1 -
alpha:.2f})")
381
382     ssc = min(bin_coverages.values())
383     print(f"\n SSC Metric (min bin-wise coverage): {ssc:.3f}")
384     return ssc, bin_coverages
385
386
387 def plot_ssc_coverage(bin_coverages, alpha=0.1, title="Size-Stratified Coverage"):
388     bin_labels = list(bin_coverages.keys())
389     coverages = [bin_coverages[k] for k in bin_labels]
390
391     plt.figure(figsize=(10, 5))
392     bars = plt.bar(bin_labels, coverages, color='pink', edgecolor='black')
393
394     target = 1 - alpha
395     plt.axhline(target, color='red', linestyle='--', label=f"Target Coverage
({target:.2f})")

```

```

396
397     for bar, cov in zip(bars, coverages):
398         if cov < target:
399             bar.set_color('grey')
400
401     plt.xticks(rotation=45, ha='right')
402     plt.ylim(0, 1.05)
403     plt.ylabel("Coverage")
404     plt.title(title)
405     plt.legend()
406     plt.tight_layout()
407     filename = f"plot_ssc_coverage_{test_run}_{timestamp}.png"
408     plt.savefig(os.path.join(results_directory, filename))
409     plt.show()
410
411
412 def plot_prediction_set_size_by_class(prediction_sets, true_labels, class_names=None,
413 results_dir="", test_run="",
414                                     timestamp=""):
415     size_per_class = defaultdict(list)
416     for pred_set, true_label in zip(prediction_sets, true_labels):
417         size_per_class[true_label].append(len(pred_set))
418
419     sorted_classes = sorted(size_per_class.keys())
420     means = [np.mean(size_per_class[c]) for c in sorted_classes]
421     stds = [np.std(size_per_class[c]) for c in sorted_classes]
422     labels = [class_names[c] if class_names else str(c) for c in sorted_classes]
423
424     plt.figure(figsize=(12, 6))
425     plt.bar(labels, means, yerr=stds, capsize=5, color="lightgreen",
426 edgecolor="black")
427     plt.ylabel("Avg Prediction Set Size")
428     plt.xlabel("Class")
429     plt.title("Average Prediction Set Size by Class")
430     plt.xticks(rotation=45, ha="right")
431     plt.grid(axis="y", linestyle="--", alpha=0.5)
432     plt.tight_layout()
433
434     filename = f"prediction_set_size_by_class_{test_run}_{timestamp}.png"
435     plt.savefig(os.path.join(results_directory, filename))
436     plt.show()
437
438
439 def plot_calibration_score_distribution(calibration_scores, results_dir="",
440 test_run="", timestamp=""):
441     plt.figure(figsize=(10, 5))
442     plt.hist(calibration_scores, bins=30, color="pink", edgecolor="black", alpha=0.8)
443     plt.xlabel("Nonconformity Score (1 - P(True Class))")
444     plt.ylabel("Frequency")
445     plt.title("Calibration Score Distribution")
446     plt.grid(axis="y", linestyle="--", alpha=0.6)
447     plt.tight_layout()
448
449     filename = f"calibration_score_distribution_{test_run}_{timestamp}.png"
450     plt.savefig(os.path.join(results_directory, filename))
451     plt.show()
452
453 # =====
454 # Multiple alpha values conformal evaluation

```



```

453 # =====
454 timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
455
456 base_results_root =
457     f"Results_normal_Conformal_Prediction_experiment_run_{test_run}_{timestamp}"
458 os.makedirs(base_results_root, exist_ok=True)
459 print("Base conformal results directory:", base_results_root)
460
461 print("\n===== Computing calibration scores on validation set =====\n")
462 results_directory = base_results_root
463 scores_calibration = calibration_scores(
464     model=pretrained_vit,
465     data_loader=val_data_loader_pretrained,
466     device=device
467 )
468 np.save(
469     os.path.join(base_results_root,
470         f"calibration_scores_experiment_run_{test_run}_{timestamp}.npz"),
471     scores_calibration
472 )
473 with open(
474     os.path.join(base_results_root,
475         f"scores_calibration_run_{test_run}_{timestamp}.pkl"),
476     "wb"
477 ) as f:
478     pickle.dump(scores_calibration, f)
479
480 plot_calibration_score_distribution(
481     calibration_scores=scores_calibration,
482     results_dir=base_results_root,
483     test_run=test_run,
484     timestamp=timestamp
485 )
486
487 coverage_levels = np.arange(0.65, 0.95 + 1e-9, 0.05)
488 all_alpha_metrics = []
489
490 for coverage_target in coverage_levels:
491     alpha = 1.0 - coverage_target
492
493     print("\n=====")
494     print(f" Target coverage: {coverage_target:.2f} | alpha = {alpha:.2f}")
495     print("=====")
496
497     cov_tag = int(round(coverage_target * 100))
498     results_directory = os.path.join(base_results_root, f"coverage_{cov_tag}")
499     os.makedirs(results_directory, exist_ok=True)
500
501     prediction_sets, true_labels = predict_with_conformal_sets(
502         model=pretrained_vit,
503         data_loader=test_data_loader_pretrained,
504         calibration_scores_array=scores_calibration,
505         alpha=alpha,
506         device=device
507     )
508     test_set_coverage, avg_size = evaluate(prediction_sets=prediction_sets,
509     labels=true_labels)
510
511     log_path = os.path.join(

```

```

511     results_directory,
512     f"logs_experiment_run_{test_run}_alpha_{alpha:.2f}_{timestamp}.txt"
513 )
514 with open(log_path, "w") as f:
515     f.write("##### Results
#####\n")
516     f.write(f"test_run {test_run}\n")
517     f.write(f"Target coverage: {coverage_target:.3f}\n")
518     f.write(f"Alpha: {alpha:.3f}\n")
519     f.write(f"Coverage: {test_set_coverage:.3f}\n")
520     f.write(f"Average Prediction Set Size for Test Set: {avg_size:.3f}\n")
521
522 f.write("#####\n")
523
524 print("#####Results:#####")
525 print(f"test_run {test_run}")
526 print(f"Target coverage: {coverage_target:.3f}")
527 print(f"Alpha: {alpha:.3f}")
528 print(f"Coverage: {test_set_coverage:.3f}")
529 print(f"Average Prediction Set Size for Test Set: {avg_size:.3f}")
530 print("#####")
531
532 with open(
533     os.path.join(results_directory,
534                   f"prediction_data_experiment_run_{test_run}__{timestamp}.pkl"),
535     "wb"
536 ) as f:
537     pickle.dump(
538         {
539             "prediction_sets": prediction_sets,
540             "true_labels": true_labels
541         },
542         f
543     )
544
545 # FSC & related plots
546 fsc_metric, class_coverages = feature_stratified_coverage_by_class(
547     prediction_sets,
548     true_labels,
549     alpha=alpha,
550     class_names=class_names
551 )
552 plot_class_wise_coverage(class_coverages, alpha=alpha, class_names=class_names)
553 plot_coverage_vs_class_frequency(class_coverages, true_labels, alpha=alpha,
class_names=class_names)
554 plot_coverage_vs_class_frequency_with_correlation(
555     class_coverages=class_coverages,
556     true_labels=true_labels,
557     alpha=alpha,
558     class_names=class_names
559 )
560
561 # SSC & related plots
562 ssc_metric, bin_coverages = size_stratified_coverage(
563     prediction_sets,
564     true_labels,
565     alpha=alpha,
566     bins=[[1], [2], [3], list(range(4, 100))]
567 )
568 plot_ssc_coverage(bin_coverages, alpha=alpha)

```

```

568
569 # Prediction set size by class
570 plot_prediction_set_size_by_class(
571     prediction_sets=prediction_sets,
572     true_labels=true_labels,
573     class_names=class_names,
574     results_dir=results_directory,
575     test_run=test_run,
576     timestamp=timestamp
577 )
578
579 metrics = {
580     "test_run": test_run,
581     "timestamp": timestamp,
582     "alpha": alpha,
583     "target_coverage": coverage_target,
584     "test_set_coverage": test_set_coverage,
585     "average_prediction_set_size": avg_size,
586     "FSC": fsc_metric,
587     "SSC": ssc_metric,
588     "class_coverages": {str(k): float(v) for k, v in class_coverages.items()},
589     "bin_coverages": bin_coverages
590 }
591
592 with open(os.path.join(results_directory,
593     f"metrics_summary_{test_run}_alpha_{alpha:.2f}_{timestamp}.json"), "w") as f:
594     json.dump(metrics, f, indent=4)
595
596 all_alpha_metrics.append(metrics)
597
598 with open(os.path.join(base_results_root,
599     f"metrics_summary_all_alphas_{test_run}_{timestamp}.json"),
600     "w") as f:
601     json.dump(all_alpha_metrics, f, indent=4)
602
603 print("\nFinished multi-alpha conformal evaluation for coverage 0.65-0.95 (step
604     0.05).")
605 print("All conformal results saved under:", base_results_root)

```