

```

1  import sys
2  import os
3  from pathlib import Path
4  import torch
5  import torchvision
6  from torch import nn
7  from torchvision import transforms, datasets
8  from torch.utils.data import DataLoader, Dataset
9  from torchinfo import summary
10 from timeit import default_timer as timer
11 from torch.utils.tensorboard import SummaryWriter
12
13 # Set device
14 device = "cuda" if torch.cuda.is_available() else "cpu"
15 print(f"Using device: {device}")
16
17 # Define Synthetic Dataset class
18 class SyntheticDataset(Dataset):
19     """A synthetic dataset for quick testing."""
20     def __init__(
21         self,
22         num_samples=1000,
23         img_size=224,
24         num_classes=1000
25     ):
26         self.num_samples = num_samples
27         self.img_size = img_size
28         self.num_classes = num_classes
29         self.data = torch.randn(
30             num_samples,
31             3,
32             img_size,
33             img_size
34         )
35         self.labels = torch.randint(
36             0,
37             num_classes,
38             (num_samples,)
39         )
40
41     def __len__(self):
42         return self.num_samples
43
44     def __getitem__(self, idx):
45         return self.data[idx], self.labels[idx]
46
47 # Define Vision Transformer components
48 class PatchEmbedding(nn.Module):
49     def __init__(
50         self,
51         in_channels: int = 3,
52         patch_size: int = 16,
53         embedding_dim: int = 768
54     ):
55         super().__init__()
56         self.patcher = nn.Conv2d(
57             in_channels=in_channels,
58             out_channels=embedding_dim,
59             kernel_size=patch_size,
60             stride=patch_size
61         )
62         self.flatten = nn.Flatten(start_dim=2)
63
64     def forward(self, x: torch.Tensor) -> torch.Tensor:
65         x = self.patcher(x)
66         x = self.flatten(x)
67         return x.permute(0, 2, 1) # [batch_size, num_patches, embedding_dim]
68
69 class MultiheadSelfAttention(nn.Module):
70     def __init__(
71         self,
72         embedding_dim: int = 768,
73         num_heads: int = 12,
74         attn_dropout: float = 0.0
75     ):

```

```

76         super().__init__()
77         self.layer_norm = nn.LayerNorm(
78             normalized_shape=embedding_dim
79         )
80         self.multihead_attn = nn.MultiheadAttention(
81             embed_dim=embedding_dim,
82             num_heads=num_heads,
83             dropout=attn_dropout,
84             batch_first=True
85         )
86
87     def forward(self, x):
88         x = self.layer_norm(x)
89         attn_output, _ = self.multihead_attn(
90             query=x,
91             key=x,
92             value=x,
93             need_weights=False
94         )
95         return attn_output
96
97 class MLPBlock(nn.Module):
98     def __init__(
99         self,
100         embedding_dim: int = 768,
101         mlp_size: int = 3072,
102         dropout: float = 0.1
103     ):
104         super().__init__()
105         self.layer_norm = nn.LayerNorm(
106             normalized_shape=embedding_dim
107         )
108         self.mlp = nn.Sequential(
109             nn.Linear(
110                 in_features=embedding_dim,
111                 out_features=mlp_size
112             ),
113             nn.GELU(),
114             nn.Dropout(p=dropout),
115             nn.Linear(
116                 in_features=mlp_size,
117                 out_features=embedding_dim
118             ),
119             nn.Dropout(p=dropout)
120         )
121
122     def forward(self, x: torch.Tensor) -> torch.Tensor:
123         x = self.layer_norm(x)
124         return self.mlp(x)
125
126 class TransformerEncoderBlock(nn.Module):
127     def __init__(
128         self,
129         embedding_dim: int = 768,
130         num_heads: int = 12,
131         mlp_size: int = 3072,
132         attn_dropout: float = 0.0,
133         mlp_dropout: float = 0.1
134     ):
135         super().__init__()
136         self.msa_block = MultiheadSelfAttention(
137             embedding_dim=embedding_dim,
138             num_heads=num_heads,
139             attn_dropout=attn_dropout
140         )
141         self.mlp_block = MLPBlock(
142             embedding_dim=embedding_dim,
143             mlp_size=mlp_size,
144             dropout=mlp_dropout
145         )
146
147     def forward(self, x: torch.Tensor) -> torch.Tensor:
148         x = self.msa_block(x) + x
149         x = self.mlp_block(x) + x
150         return x

```

```

151
152 class VisionTransformer(nn.Module):
153     def __init__(
154         self,
155         image_size: int = 224,
156         patch_size: int = 16,
157         num_transformer_layers: int = 12,
158         embedding_dim: int = 768,
159         num_heads: int = 12,
160         mlp_size: int = 3072,
161         num_classes: int = 1000
162     ):
163         super().__init__()
164         assert image_size % patch_size == 0, "Image size must be divisible by patch size."
165         self.num_patches = (image_size * image_size) // (patch_size * patch_size)
166         self.class_embedding = nn.Parameter(
167             torch.randn(1, 1, embedding_dim)
168         )
169         self.position_embeddings = nn.Parameter(
170             torch.randn(1, self.num_patches + 1, embedding_dim)
171         )
172         self.embedding_dropout = nn.Dropout(0.1)
173         self.patch_embedding = PatchEmbedding(
174             3,
175             patch_size,
176             embedding_dim
177         )
178         self.transformer_encoder = nn.Sequential(
179             *[
180                 TransformerEncoderBlock(
181                     embedding_dim,
182                     num_heads,
183                     mlp_size
184                 ) for _ in range(num_transformer_layers)
185             ]
186         )
187         self.classifier = nn.Sequential(
188             nn.LayerNorm(
189                 normalized_shape=embedding_dim
190             ),
191             nn.Linear(
192                 in_features=embedding_dim,
193                 out_features=num_classes
194             )
195         )
196
197     def forward(self, x: torch.Tensor) -> torch.Tensor:
198         batch_size = x.size(0)
199         class_token = self.class_embedding.expand(
200             batch_size,
201             -1,
202             -1
203         )
204         x = self.patch_embedding(x)
205         x = torch.cat((class_token, x), dim=1)
206         x = x + self.position_embeddings
207         x = self.embedding_dropout(x)
208         x = self.transformer_encoder(x)
209         return self.classifier(x[:, 0])
210
211 # Define training function with TensorBoard logging
212 def train_model(
213     model,
214     dataloader,
215     criterion,
216     optimizer,
217     epochs=5,
218     log_dir="logs/train"
219 ):
220     writer = SummaryWriter(log_dir=log_dir)
221     model.train()
222     for epoch in range(epochs):
223         epoch_loss = 0
224         correct = 0
225         total = 0

```

```

226         for batch_idx, (images, labels) in enumerate(dataloader):
227             images, labels = images.to(device), labels.to(device)
228             optimizer.zero_grad()
229             outputs = model(images)
230             loss = criterion(outputs, labels)
231             loss.backward()
232             optimizer.step()
233
234             epoch_loss += loss.item()
235             _, predicted = torch.max(outputs, 1)
236             correct += (predicted == labels).sum().item()
237             total += labels.size(0)
238
239             # Log batch loss to TensorBoard
240             writer.add_scalar("Batch Loss", loss.item(), epoch * len(dataloader) + batch_idx)
241
242         epoch_loss /= len(dataloader)
243         accuracy = correct / total * 100
244         print(
245             f"Epoch {epoch + 1}/{epochs}, Loss: {epoch_loss:.4f}, Accuracy: {accuracy:.2f}%"
246         )
247
248         # Log epoch metrics to TensorBoard
249         writer.add_scalar("Epoch Loss", epoch_loss, epoch)
250         writer.add_scalar("Epoch Accuracy", accuracy, epoch)
251
252     writer.close()
253
254     # Define evaluation function with TensorBoard logging
255     def evaluate_model(model, dataloader, criterion, log_dir="logs/eval"):
256         writer = SummaryWriter(log_dir=log_dir)
257         model.eval()
258         total_loss = 0
259         correct = 0
260         total = 0
261         with torch.no_grad():
262             for batch_idx, (images, labels) in enumerate(dataloader):
263                 images, labels = images.to(device), labels.to(device)
264                 outputs = model(images)
265                 loss = criterion(outputs, labels)
266                 total_loss += loss.item()
267                 _, predicted = torch.max(outputs, 1)
268                 correct += (predicted == labels).sum().item()
269                 total += labels.size(0)
270
271                 # Log batch loss to TensorBoard
272                 writer.add_scalar("Batch Loss", loss.item(), batch_idx)
273
274         avg_loss = total_loss / len(dataloader)
275         accuracy = correct / total * 100
276         print(f"Evaluation - Loss: {avg_loss:.4f}, Accuracy: {accuracy:.2f}%")
277
278         # Log evaluation metrics to TensorBoard
279         writer.add_scalar("Average Loss", avg_loss)
280         writer.add_scalar("Accuracy", accuracy)
281
282     writer.close()
283     return avg_loss, accuracy
284
285     # Retain pizza dataset functionality for later use
286     def load_pizza_data(custom_transform=None):
287         data_path = Path("vit/data/pizza_steak_sushi/")
288         train_dir = data_path / "train"
289         test_dir = data_path / "test"
290
291         train_data = datasets.ImageFolder(
292             train_dir,
293             transform=custom_transform
294         )
295         test_data = datasets.ImageFolder(
296             test_dir,
297             transform=custom_transform
298         )
299
300         train_dataloader = DataLoader(

```

```

301         train_data,
302         batch_size=32,
303         shuffle=True
304     )
305     test_dataloader = DataLoader(
306         test_data,
307         batch_size=32,
308         shuffle=False
309     )
310
311     return train_dataloader, test_dataloader
312
313 def pretrained_vit_model():
314     pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT
315     pretrained_vit_transforms = pretrained_vit_weights.transforms()
316     pretrained_vit_model = torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)
317     for param in pretrained_vit_model.parameters():
318         param.requires_grad = False
319     torch.manual_seed(42)
320     torch.cuda.manual_seed(42)
321     pretrained_vit_model.heads = nn.Linear(in_features=768, out_features=3).to(device)
322     summary(
323         model=pretrained_vit_model,
324         input_size=(32, 3, 224, 224),
325         col_names=["input_size", "output_size", "num_params", "trainable"],
326         col_width=20,
327         row_settings=["var_names"]
328     )
329     return pretrained_vit_model, pretrained_vit_transforms
330
331 # Test with pizza dataset
332 if __name__ == "__main__":
333     print("Testing with pizza dataset...")
334
335     # Prompt user to choose between manual training and pretrained model fine-tuning
336     choice = input("Choose training mode: 'manual' for manual training or 'pretrained' for fine-tuning pretrained model:").strip().lower()
337
338     if choice == 'manual':
339         # Manual training with VisionTransformer
340         print("Using manual training with VisionTransformer...")
341         summary(
342             model=VisionTransformer(),
343             input_size=(32, 3, 224, 224),
344             col_names=["input_size", "output_size", "num_params", "trainable"],
345             col_width=20,
346             row_settings=["var_names"]
347         )
348         train_dataloader, test_dataloader = load_pizza_data()
349
350         vit_model = VisionTransformer(
351             image_size=224,
352             patch_size=16,
353             num_transformer_layers=12,
354             embedding_dim=768,
355             num_heads=12,
356             mlp_size=3072,
357             num_classes=3
358         ).to(device)
359         criterion = nn.CrossEntropyLoss()
360         optimizer = torch.optim.Adam(
361             vit_model.parameters(),
362             lr=3e-4
363         )
364
365         train_model(
366             vit_model,
367             train_dataloader,
368             criterion,
369             optimizer,
370             epochs=5,
371             log_dir="logs/manual_train"
372         )
373
374     print("Evaluating the manually trained model...")

```

```
375     evaluate_model(vit_model, test_dataloader, criterion, log_dir="logs/manual_eval")
376
377     # Save the manually trained model
378     MODEL_PATH = Path("torch_models") / "manual_vit_model.pth"
379     MODEL_PATH.parent.mkdir(parents=True, exist_ok=True)
380     torch.save(vit_model.state_dict(), MODEL_PATH)
381     print(f"Manually trained model saved to: {MODEL_PATH}")
382
383 elif choice == 'pretrained':
384     # Fine-tuning pretrained VisionTransformer
385     print("Using pretrained VisionTransformer for fine-tuning...")
386     vit_model, vit_transforms = pretrained_vit_model()
387     train_dataloader, test_dataloader = load_pizza_data(vit_transforms)
388
389     criterion = nn.CrossEntropyLoss()
390     optimizer = torch.optim.Adam(
391         vit_model.heads.parameters(), # Only train the classification head
392         lr=3e-4
393     )
394
395     print("Training the pretrained model...")
396     train_model(
397         vit_model,
398         train_dataloader,
399         criterion,
400         optimizer,
401         epochs=20,
402         log_dir="logs/pretrained_train"
403     )
404
405     print("Evaluating the fine-tuned pretrained model...")
406     evaluate_model(vit_model, test_dataloader, criterion, log_dir="logs/pretrained_eval")
407
408     # Save the fine-tuned model
409     MODEL_PATH = Path("torch_models") / "fine_tuned_vit_modelv1.pth"
410     MODEL_PATH.parent.mkdir(parents=True, exist_ok=True)
411     torch.save(vit_model.state_dict(), MODEL_PATH)
412     print(f"Fine-tuned model saved to: {MODEL_PATH}")
413
414 else:
415     print("Invalid choice. Please choose 'manual' or 'pretrained'.")
```