

Assignment 2 Understanding transfer learning and fine tuning

```
In [9]: import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torch.utils.data import Dataset
import torch.nn.functional as F
import matplotlib.pyplot as plt
from PIL import Image
from torchvision import transforms
from torch.utils.data import random_split
from torchvision import datasets
import os
import numpy as np
from typing import Literal
from torchvision.models import resnet18, ResNet18_Weights
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score
from torch.amp import autocast, GradScaler

print(f"[INFO] Torch infos: {torch.__version__}")

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"[INFO] Using device: {device}")
```

```
[INFO] Torch infos: 2.10.0+cu126
```

```
[INFO] Using device: cuda
```

We need transform from resnet18 and its weights

```
In [10]: transform = ResNet18_Weights.DEFAULT.transforms()
```

pick a dataset you can import from

```
from torchvision import datasets
```

Easy picks:

- Food101
- Flowers102
- DTD
- FGVAircraft

Other picks:

```
full_train = datasets.OxfordIIITPet(root="data", split="trainval", download=True, transform=transform)
test_ds     = datasets.OxfordIIITPet(root="data", split="test", download=True, transform=transform)
```

Split the data

In [11]: *# Oxford pet dataset*

```
full_train = datasets.OxfordIIITPet(root="data", split="trainval", download=True, transform=transform)
test_ds     = datasets.OxfordIIITPet(root="data", split="test", download=True, transform=transform)

# Train/val split
val_ratio = 0.2
val_size = int(len(full_train) * val_ratio)
train_size = len(full_train) - val_size
train_ds, val_ds = random_split(full_train, [train_size, val_size])

print(f"Train size:      {len(train_ds)}")
print(f"Validation size: {len(val_ds)}")
print(f"Test size:         {len(test_ds)}")
```

```
Train size:      2944
Validation size: 736
Test size:       3669
```

load the train, validation and test split

```
In [12]: # Configurations
NUM_CLASSES = 37
EPOCHS = 15
PATIENCE = 3
BATCH_SIZE = 128
SCALER = GradScaler()
```

create dataloader handler from the dataset

```
In [13]: # Trying out num_workers for more efficiency
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, num_workers=4, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE, num_workers=4, shuffle=False)
test_loader = DataLoader(test_ds, batch_size=BATCH_SIZE, num_workers=4, shuffle=False)

print(f"Train batches: {len(train_loader)}")
print(f"Val batches: {len(val_loader)}")
print(f"Test batches: {len(test_loader)}")
```

Train batches: 23

Val batches: 6

Test batches: 29

now create the cnn, use the resnet18 as backbone, and ResNet18_Weights as initial weights.

- create the backbone
- create a classifier
- based on your dataset add the correct number of output classes
- the classifier have to be trainable.

Question 1?

- Why we freeze the backbone? Why not the classifier?

Answer

- We freeze backbone because it was already trained on ImageNet general purpose patterns (edges, textures, shapes). Freezing it means those weights are not updated during training, which:
 1. Saves computation
 2. Prevents destroying the learned patterns

```
In [14]: # Model for transfer learning From class power point example
class ResNet18Transfer(nn.Module):
    def __init__(self,
                  num_classes: int,
                  freeze_backbone: bool = True,
                  dense_units: list[int] = [256],
                  dropout_probabilities: float = 0.3,
                  weights=ResNet18_Weights.DEFAULT):
        super().__init__()
        self.weights = weights

        # Create a feature extractor
        backbone = resnet18(weights=weights)
        self.features = nn.Sequential(*list(backbone.children())[:-1])
        self.backbone_out = 512 # 512 for ResNet18

        # Freeze the layers
        if freeze_backbone:
            for p in self.features.parameters():
                p.requires_grad = False

        # Classifier
        mlp = []
        cur = self.backbone_out
        for h in dense_units:
            mlp += [
```

```
        nn.Linear(cur, h),
        nn.ReLU(inplace=True),
        nn.Dropout(p=dropout_probabilities),
    ]
    cur = h  # Update current size

    # Add final classifier
    self.classifier = nn.Sequential(*mlp)
    self.final_classifier = nn.Linear(cur, num_classes)

    # Ensure head trainable
    for p in self.classifier.parameters():
        p.requires_grad = True
    for p in self.final_classifier.parameters():
        p.requires_grad = True

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)

        return self.final_classifier(x)

    # Helper to unfreeze in phase2
    def unfreeze_layer4(self):
        # features[7] is Layer4 in ResNet18
        for p in self.features[7].parameters():
            p.requires_grad = True

model = ResNet18Transfer(num_classes=NUM_CLASSES, freeze_backbone=True, dropout_probabilities=0.3).to(device)
print(model)
```

```

ResNet18Transfer(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (5): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
)

```

```

(6): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(7): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(8): AdaptiveAvgPool2d(output_size=(1, 1))
)

```

```
(classifier): Sequential(
  (0): Linear(in_features=512, out_features=256, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.3, inplace=False)
)
(final_classifier): Linear(in_features=256, out_features=37, bias=True)
)
```

add the train evaluation, and predict functions here

```
In [15]: # Training for one epoch
def train_one_epoch(model, loader, criterion, optimizer, scaler):

    model.train()
    total_loss = 0.0
    all_preds, all_labels = [], []

    for images, labels in loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()

        with autocast(device_type="cuda"):
            outputs = model(images)
            loss = criterion(outputs, labels)

        # Trying scaler to better performance
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()
        preds = outputs.argmax(dim=1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

    avg_loss = total_loss / len(loader)
    acc = accuracy_score(all_labels, all_preds)
    return avg_loss, acc
```



```
# Evaluating function for model
def evaluate(model, loader, criterion, device):

    model.eval()
    total_loss = 0.0
    all_preds, all_labels = [], []

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)

            with autocast(device_type="cuda"):
                outputs = model(images)
                loss = criterion(outputs, labels)

            total_loss += loss.item()
            preds = outputs.argmax(dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    avg_loss = total_loss / len(loader)
    acc = accuracy_score(all_labels, all_preds)
    return avg_loss, acc

# Training function for model, with stopping early
def train(model, train_loader, val_loader, criterion, optimizer, epochs, scaler, patience=PATIENCE):

    train_losses, val_losses = [], []
    train_accs, val_accs = [], []

    best_val_loss = float('inf')
    patience_counter = 0
    best_model_state = None

    for epoch in range(1, epochs + 1):
        train_loss, train_acc = train_one_epoch(model, train_loader, criterion, optimizer, scaler)
        val_loss, val_acc = evaluate(model, val_loader, criterion, device)

        train_losses.append(train_loss)
        val_losses.append(val_loss)
        train_accs.append(train_acc)
```

```

val_accs.append(val_acc)

# Tryin early stopping
if val_loss < best_val_loss:
    best_val_loss = val_loss
    patience_counter = 0
    best_model_state = model.state_dict().copy()
else:
    patience_counter += 1

if patience_counter >= patience:
    print(f"\n[EARLY STOP] No improvement for {patience} epochs. Stopping at epoch {epoch}.")
    model.load_state_dict(best_model_state)
    break

# If results stumble 5 consecutive epochs stop learning
if epoch % 5 == 0 or epoch == 1:
    print(f"Epoch {epoch:3d}/{epochs} | "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} | "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

return train_losses, val_losses, train_accs, val_accs

# Prediction function for model
def predict(model, loader, device):

    model.eval()
    all_preds, all_labels = [], []

    with torch.no_grad():
        for images, labels in loader:
            images = images.to(device)

            with autocast(device_type="cuda"):
                outputs = model(images)

            preds = outputs.argmax(dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.numpy())

    return np.array(all_preds), np.array(all_labels)

```

Phase 1: Transfer learning

freeze all layers except the classifier.

- train and evaluate the model for 50 epochs
- remember to save val loss and train loss

```
In [16]: # Freeze backbone parameters
for param in model.features.parameters():
    param.requires_grad = False

# Only the classifier parameters will be updated, because it the new added layer
optimizer = optim.Adam(model.classifier.parameters(), lr=1e-4)
criterion = nn.CrossEntropyLoss()

# Train for 50 epochs using the train() function
phase1_train_losses, phase1_val_losses, phase1_train_accs, phase1_val_accs = train(
    model, train_loader, val_loader, criterion, optimizer, scaler=SCALER, epochs=EPOCHS, patience=PATIENCE
)
```

```
Epoch  1/15 | Train Loss: 3.5374, Train Acc: 0.0642 | Val Loss: 3.4125, Val Acc: 0.1739
Epoch  5/15 | Train Loss: 2.7046, Train Acc: 0.4698 | Val Loss: 2.5787, Val Acc: 0.6304
Epoch 10/15 | Train Loss: 1.8614, Train Acc: 0.7052 | Val Loss: 1.7684, Val Acc: 0.7840
Epoch 15/15 | Train Loss: 1.3640, Train Acc: 0.7799 | Val Loss: 1.3038, Val Acc: 0.8220
```

Plot accuracy and loss

```
In [17]: epochs_range = range(1, EPOCHS + 1)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Loss curve
axes[0].plot(epochs_range, phase1_train_losses, label="Train Loss")
axes[0].plot(epochs_range, phase1_val_losses, label="Val Loss")
axes[0].set_title("Phase 1 - Loss")
axes[0].set_xlabel("Epoch")
```

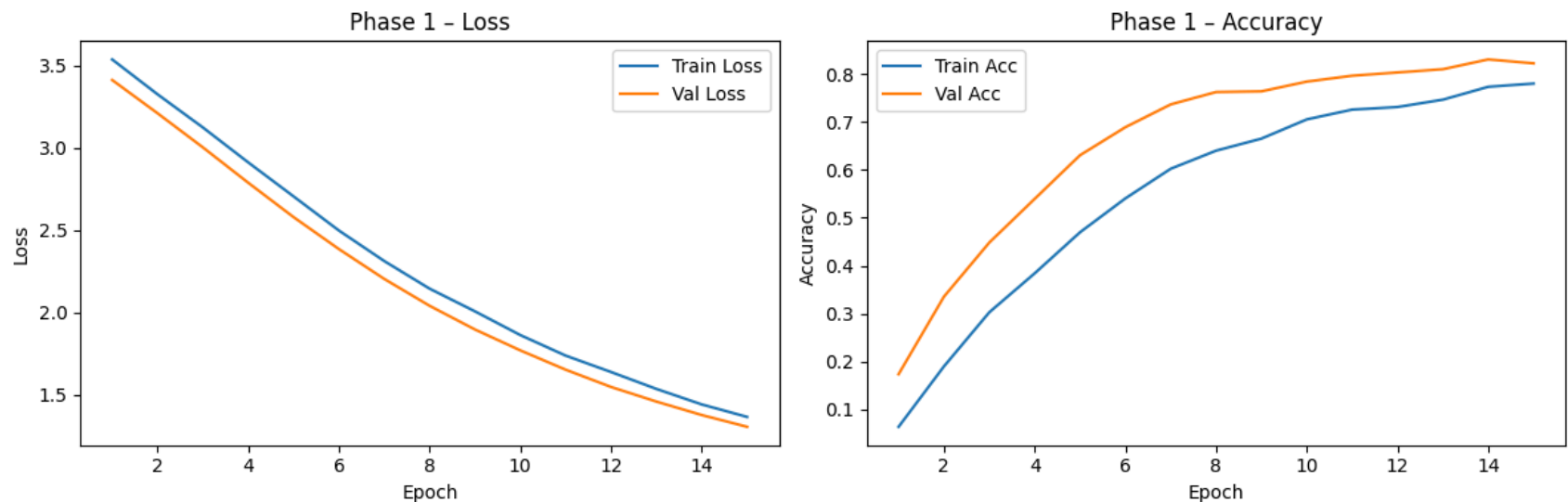
```

axes[0].set_ylabel("Loss")
axes[0].legend()

# Accuracy curve
axes[1].plot(epochs_range, phase1_train_accs, label="Train Acc")
axes[1].plot(epochs_range, phase1_val_accs, label="Val Acc")
axes[1].set_title("Phase 1 - Accuracy")
axes[1].set_xlabel("Epoch")
axes[1].set_ylabel("Accuracy")
axes[1].legend()

plt.tight_layout()
plt.show()

```



plot predictions

```

In [18]: # Grid of guesses for visualization, same type used in assignment 1
class_names = test_ds.classes # List of 37 breed names

# Grab one batch from the test loader
images_batch, labels_batch = next(iter(test_loader))
outputs = model(images_batch.to(device))
preds_batch = outputs.argmax(dim=1).cpu()

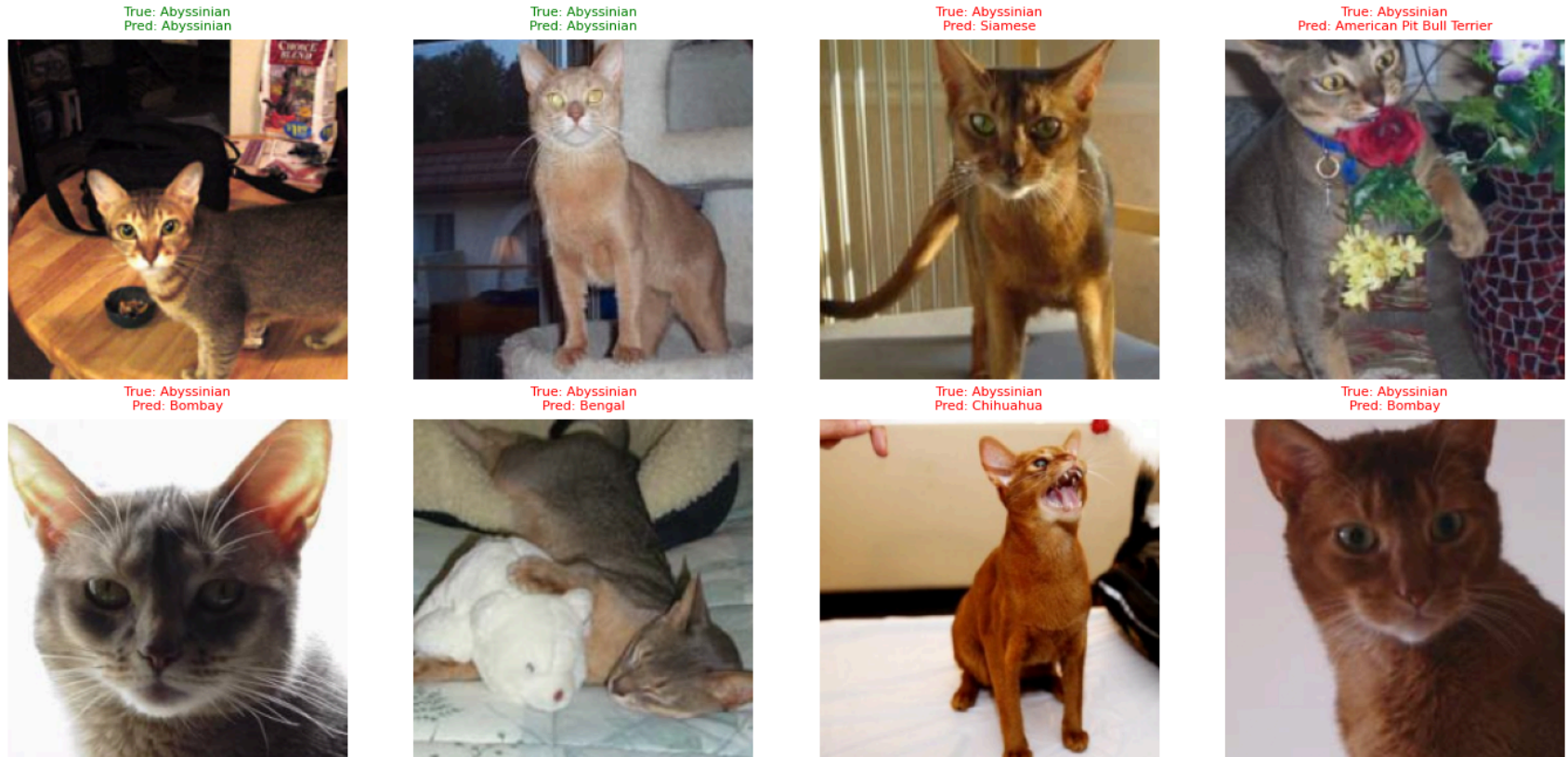
```

```
# ImageNet normalisation used by ResNet - we reverse it for display
mean = torch.tensor([0.485, 0.456, 0.406])
std  = torch.tensor([0.229, 0.224, 0.225])

fig, axes = plt.subplots(2, 4, figsize=(14, 7))
for i, ax in enumerate(axes.flat):
    img = images_batch[i].permute(1, 2, 0) * std + mean
    img = img.clamp(0, 1).numpy()
    ax.imshow(img)
    true_name = class_names[labels_batch[i]]
    pred_name = class_names[preds_batch[i]]
    color = "green" if labels_batch[i] == preds_batch[i] else "red"
    ax.set_title(f"True: {true_name}\nPred: {pred_name}", color=color, fontsize=8)
    ax.axis("off")

plt.suptitle("Phase 1 Predictions (green = correct, red = wrong)", fontsize=12)
plt.tight_layout()
plt.show()
```

Phase 1 Predictions (green = correct, red = wrong)



calculate TEST accuracy score

```
In [19]: phase1_preds, phase1_labels = predict(model, test_loader, device)
phase1_test_acc = accuracy_score(phase1_labels, phase1_preds)
print(f"Phase 1 Test Accuracy: {phase1_test_acc:.4f}")
```

Phase 1 Test Accuracy: 0.8228

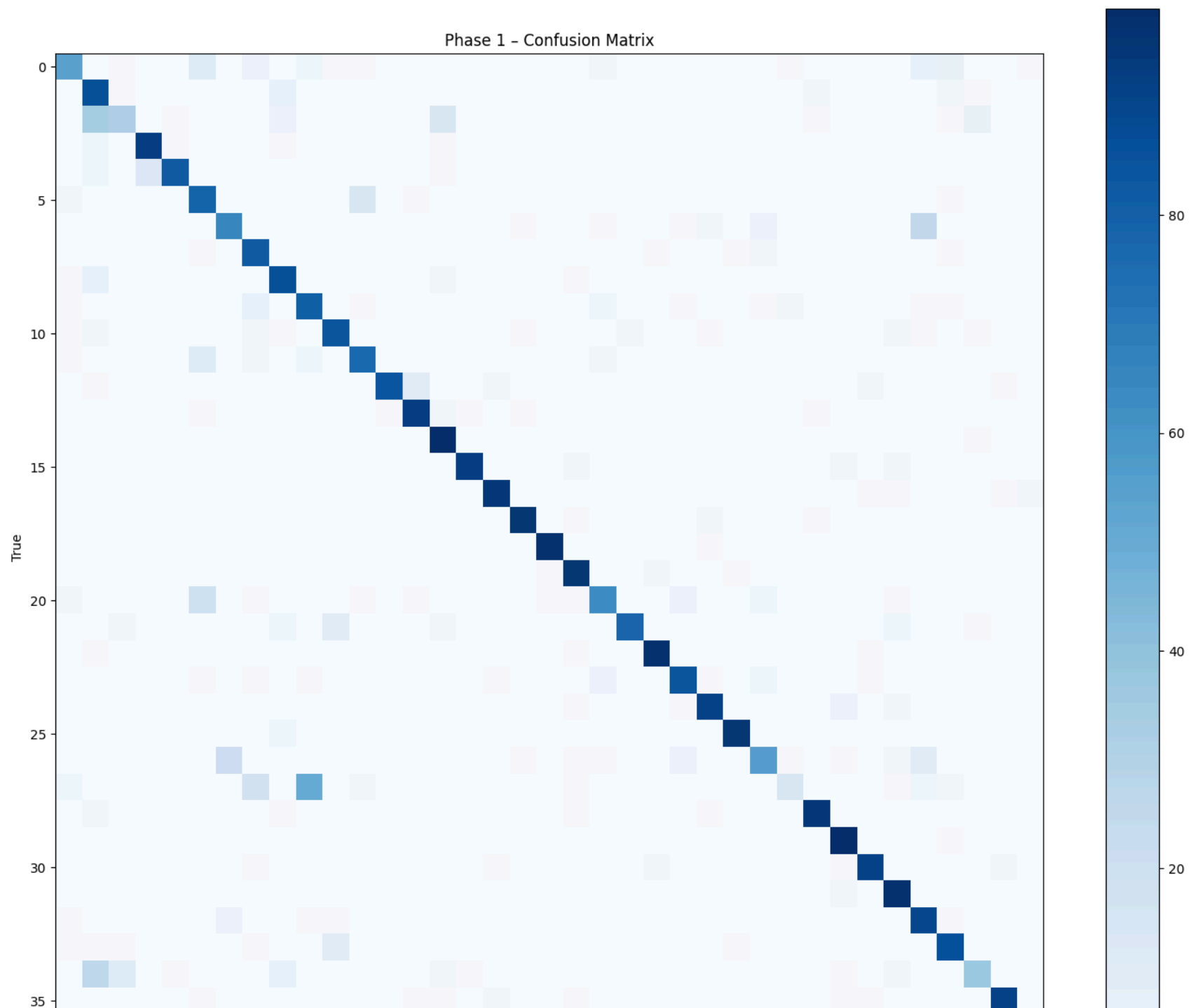
Calculate confusion matrices precision and recall

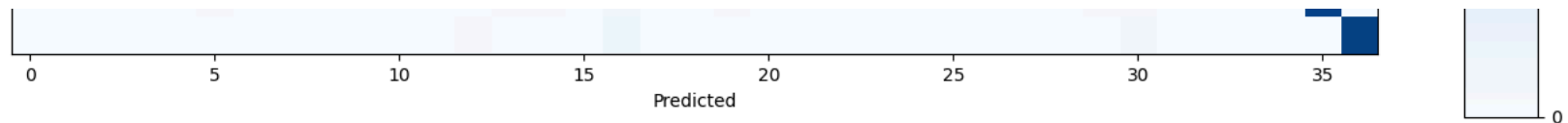
```
In [20]: # Confusion matrix
cm = confusion_matrix(phase1_labels, phase1_preds)

fig, ax = plt.subplots(figsize=(14, 12))
im = ax.imshow(cm, cmap="Blues")
plt.colorbar(im, ax=ax)
ax.set_title("Phase 1 - Confusion Matrix")
ax.set_xlabel("Predicted")
ax.set_ylabel("True")
plt.tight_layout()
plt.show()

# Precision recall matrix (macro average for a single summary number)
precision_macro = precision_score(phase1_labels, phase1_preds, average="macro", zero_division=0)
recall_macro    = recall_score(phase1_labels, phase1_preds,    average="macro", zero_division=0)

print(f"Phase 1 Macro Precision: {precision_macro:.4f}")
print(f"Phase 1 Macro Recall:    {recall_macro:.4f}")
```





Phase 1 Macro Precision: 0.8334

Phase 1 Macro Recall: 0.8219

Phase 2: Freeze layer 4 - Fine tuning

from the frozen cnn unfreeze the layer4.

```
In [21]: # Unfreezing Layer4
model.unfreeze_layer4()

# Model overfitted heavily with earlier Learning rates:
# Using a lower Learning rate for the unfrozen backbone layers to avoid
# overwriting the pretrained features too aggressively
optimizer = optim.Adam([
    {"params": model.features[7].parameters(), "lr": 1e-5},      # layer4
    {"params": model.classifier.parameters(), "lr": 5e-5},      # MLP head
    {"params": model.final_classifier.parameters(), "lr": 5e-4}, # final layer
])

scaler = GradScaler() # Reset scaler
print("layer4 and classifier are now trainable.")
```

layer4 and classifier are now trainable.

Train for 50 epochs

```
In [25]: # Train with early stopping
print("\n---Phase 2: Fine-Tuning---")
phase2_train_losses, phase2_val_losses, phase2_train_accs, phase2_val_accs = train(
    model, train_loader, val_loader, criterion, optimizer, scaler=SCALER, epochs=15, patience=PATIENCE
)
```

---Phase 2: Fine-Tuning---

Epoch	1/15		Train Loss: 0.6656,	Train Acc: 0.8675		Val Loss: 0.6370,	Val Acc: 0.8750
Epoch	5/15		Train Loss: 0.3563,	Train Acc: 0.9280		Val Loss: 0.4403,	Val Acc: 0.8899
Epoch	10/15		Train Loss: 0.1902,	Train Acc: 0.9633		Val Loss: 0.3665,	Val Acc: 0.8913
Epoch	15/15		Train Loss: 0.1105,	Train Acc: 0.9830		Val Loss: 0.3373,	Val Acc: 0.8981

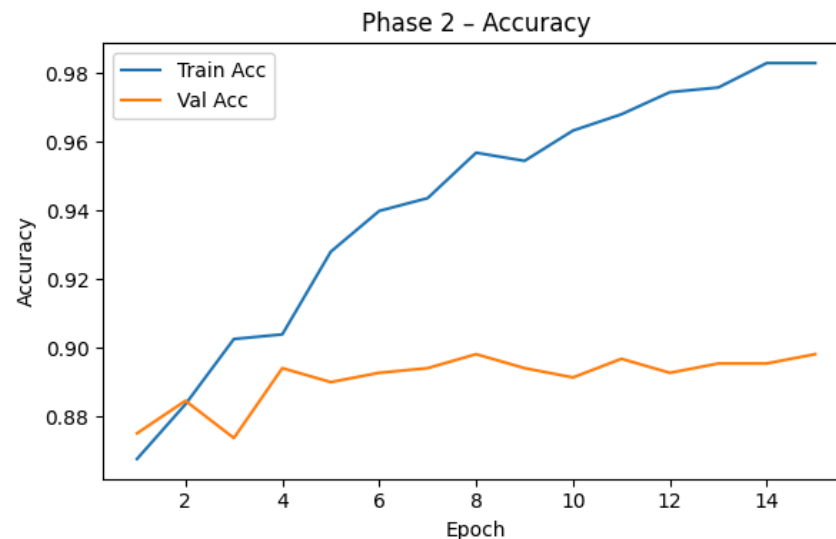
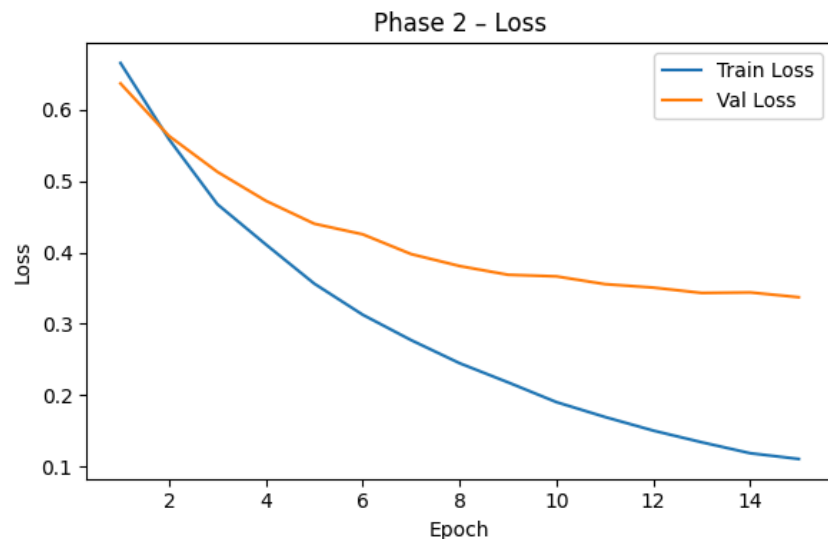
Plot curves

```
In [26]: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
epochs_range2 = range(1, len(phase2_train_losses) + 1)
axes[0].plot(epochs_range2, phase2_train_losses, label="Train Loss")
axes[0].plot(epochs_range2, phase2_val_losses, label="Val Loss")
axes[0].set_title("Phase 2 - Loss")
axes[0].set_xlabel("Epoch")
axes[0].set_ylabel("Loss")
axes[0].legend()

axes[1].plot(epochs_range2, phase2_train_accs, label="Train Acc")
axes[1].plot(epochs_range2, phase2_val_accs, label="Val Acc")
axes[1].set_title("Phase 2 - Accuracy")
axes[1].set_xlabel("Epoch")
axes[1].set_ylabel("Accuracy")
axes[1].legend()

plt.tight_layout()
plt.show()
```

```
Exception ignored while calling deallocator <function _MultiProcessingDataLoaderIter.__del__ at 0x000001C8FC4292D0>:
Traceback (most recent call last):
  File "c:\Users\Arde\miniconda3\envs\deep_learning_course_2026\Lib\site-packages\torch\utils\data\dataloader.py", line 1707, in __del__
    self._shutdown_workers()
  File "c:\Users\Arde\miniconda3\envs\deep_learning_course_2026\Lib\site-packages\torch\utils\data\dataloader.py", line 1665, in _shutdown_workers
    if self._persistent_workers or self._workers_status[worker_id]:
AttributeError: '_MultiProcessingDataLoaderIter' object has no attribute '_workers_status'
```



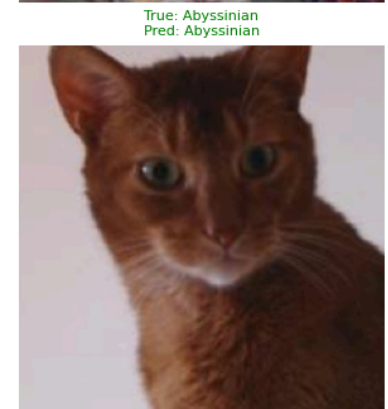
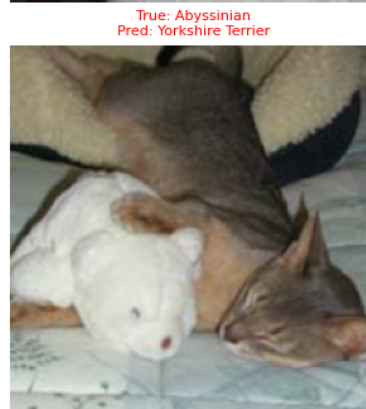
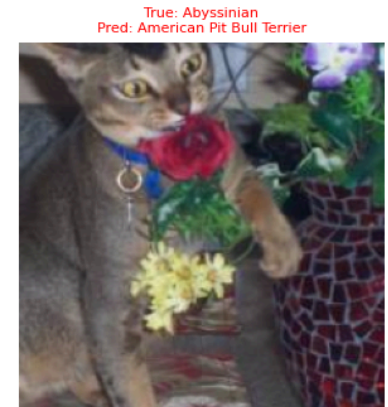
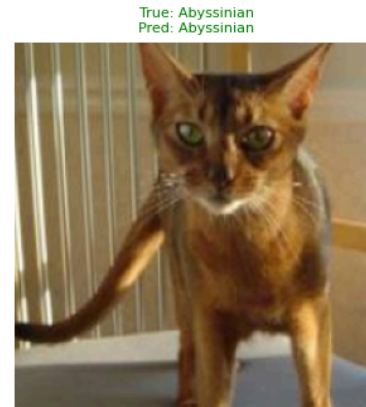
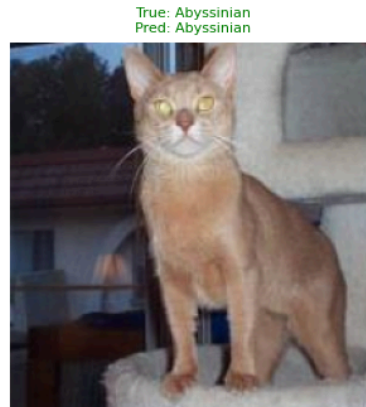
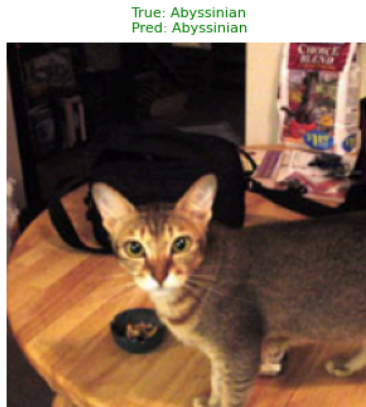
visualize prediction

```
In [27]: # Reuse the same batch from phase 1
outputs = model(images_batch.to(device))
preds_batch = outputs.argmax(dim=1).cpu()

fig, axes = plt.subplots(2, 4, figsize=(14, 7))
for i, ax in enumerate(axes.flat):
    img = images_batch[i].permute(1, 2, 0) * std + mean
    img = img.clamp(0, 1).numpy()
    ax.imshow(img)
    true_name = class_names[labels_batch[i]]
    pred_name = class_names[preds_batch[i]]
    color = "green" if labels_batch[i] == preds_batch[i] else "red"
    ax.set_title(f"True: {true_name}\nPred: {pred_name}", color=color, fontsize=8)
    ax.axis("off")

plt.suptitle("Phase 2 Predictions (green = correct, red = wrong)", fontsize=12)
plt.tight_layout()
plt.show()
```

Phase 2 Predictions (green = correct, red = wrong)



Calculate test accuracy score

```
In [29]: phase2_preds, phase2_labels = predict(model, test_loader, device)
phase2_test_acc = accuracy_score(phase2_labels, phase2_preds)
print(f"Phase 1 Test Accuracy: {phase1_test_acc:.4f}")
print(f"Phase 2 Test Accuracy: {phase2_test_acc:.4f}")
print(f"Improvement: {phase2_test_acc - phase1_test_acc:+.4f}")
```

Phase 1 Test Accuracy: 0.8228

Phase 2 Test Accuracy: 0.8899

Improvement: +0.0670

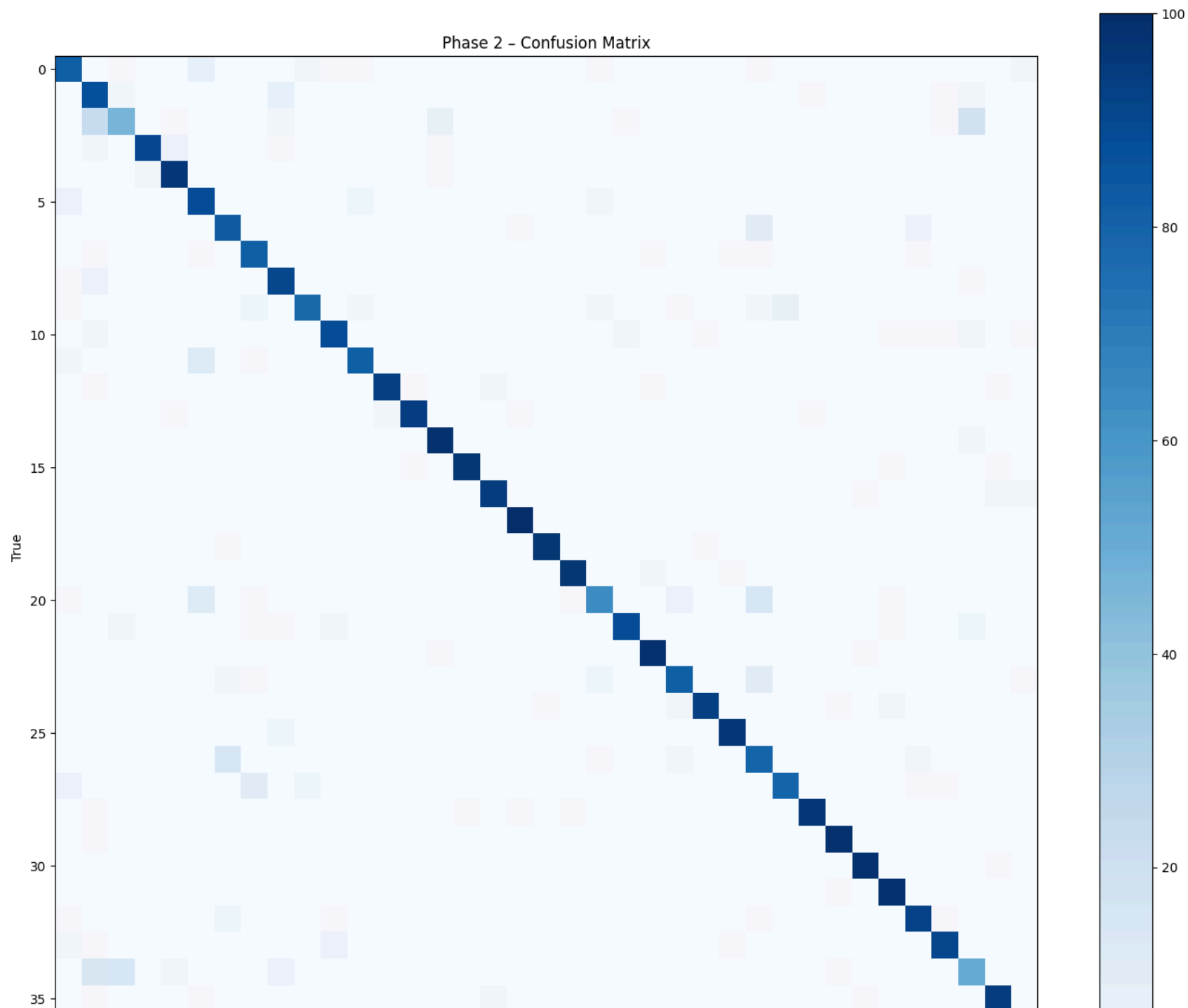
calculate confusion matrix precision and recall

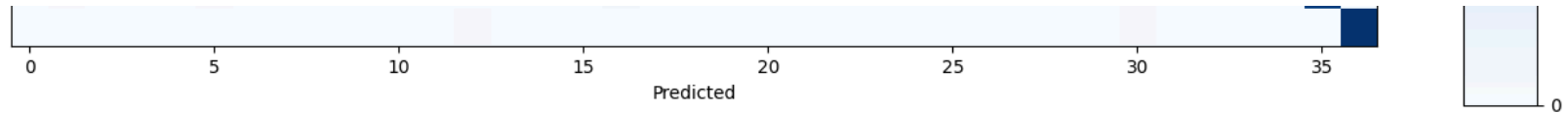
```
In [30]: cm2 = confusion_matrix(phase2_labels, phase2_preds)

fig, ax = plt.subplots(figsize=(14, 12))
im = ax.imshow(cm2, cmap="Blues")
plt.colorbar(im, ax=ax)
ax.set_title("Phase 2 - Confusion Matrix")
ax.set_xlabel("Predicted")
ax.set_ylabel("True")
plt.tight_layout()
plt.show()

precision_macro2 = precision_score(phase2_labels, phase2_preds, average="macro", zero_division=0)
recall_macro2    = recall_score(phase2_labels, phase2_preds,    average="macro", zero_division=0)

print(f"Phase 2 Macro Precision: {precision_macro2:.4f}")
print(f"Phase 2 Macro Recall:    {recall_macro2:.4f}")
```





Phase 2 Macro Precision: 0.8930

Phase 2 Macro Recall: 0.8891

Question 2 – What is the difference between transfer learning and fine tuning?

Transfer learning In phase1 we used pre-trained model and used its feature extraction layers as they were trained. We only trained an add-on classification layer to have fast results with new data. On other words we "froze" pretrained layers.

Fine tuning In phase2 we fine tuned the layer4 of the pre-trained model to have better performance on the new data we want to train it on. Risk is to over train and ruin the models performance but this is where fine tuning needs to be done well. Smaller learning rates should result in more delicate learning and keep underlying pre-trained performance in tact.

Key takeaway: Transfer learning will get some of the results if the pre-trained model is suitable. Therefore we can use large models without actually training it from scratch. Fine tuning comes handy when we want to have better results. This task was already quite computationally heavy and I had to tune down the model for better computational performance.