Assignment 1 Deep Learning and Sensor fusion 2026

Multi class Classification

Necessary iomport here

In [26]:
```
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
import os
import numpy as np
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split


# Problems with CUDA setup, test
print(f"[INFO] Torch infos: {torch.__version__}")
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"[INFO] Using device: {device}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")

DATASET_PATH = os.path.join("..", "Dataset", "BoneBreakClassification")
```

```
[INFO] Torch infos: 2.10.0+cu126
[INFO] Using device: cuda
GPU: NVIDIA GeForce RTX 3060 Ti
```

In [27]:
```
# some helpers
def load_data(dataset_path, class_names, type_of_data_to_load):
    # for each class name
    images, labels = list(), list()
```

```
    for el in class_names.keys():
        print(f"[WALKING] Walking into {el}")
        for images_name in os.listdir(os.path.join(dataset_path, el, type_of_data_to_load)):
            full_image_path = os.path.join(dataset_path, el, type_of_data_to_load, images_name)
            images.append(full_image_path)
            labels.append(class_names[el])
    return images, labels
```

set Dataset path

The path yopu see in the code point to my folder

```
.
├── Assignment_1
│   ├── Assignment_1_four_layers.ipynb
│   ├── Assignment_1_template.ipynb
│   ├── Assignment_1_three_layers.ipynb
│   └── cnn_ass_1.pt
├── Dataset
│   ├── BoneBreakClassification
```

If you put the data in a different place change it.

In [28]:
```
DATASET_PATH = os.path.join("..", "Dataset", "BoneBreakClassification")
```

(1) Load dataset

In [29]:
```
class_names = dict()

temp = os.listdir(DATASET_PATH)
temp = sorted(temp)

# prep up list and class number together
for i in range(len(temp)):
    class_names[temp[i]] = i
```

```
data_to_split, labels_to_split = load_data(dataset_path=DATASET_PATH,
                                                    class_names=class_names,
                                                    type_of_data_to_load="Train")

print(f"[DATA TO SPLIT] Data loaded complete {len(data_to_split)} Labeles are {len(labels_to_split)}")

# load test data
test_data_handler, test_labels_handler = load_data(dataset_path=DATASET_PATH,
                                                   class_names=class_names,
                                                   type_of_data_to_load="Test")

print(f"[TEST DATA LOADED] Data loaded complete {len(test_data_handler)} Labeles are {len(test_labels_handler)}")
```

```
[WALKING] Walking into Avulsion fracture
[WALKING] Walking into Comminuted fracture
[WALKING] Walking into Fracture Dislocation
[WALKING] Walking into Greenstick fracture
[WALKING] Walking into Hairline Fracture
[WALKING] Walking into Impacted fracture
[WALKING] Walking into Longitudinal fracture
[WALKING] Walking into Oblique fracture
[WALKING] Walking into Pathological fracture
[WALKING] Walking into Spiral Fracture
[DATA TO SPLIT] Data loaded complete 989 Labeles are 989
[WALKING] Walking into Avulsion fracture
[WALKING] Walking into Comminuted fracture
[WALKING] Walking into Fracture Dislocation
[WALKING] Walking into Greenstick fracture
[WALKING] Walking into Hairline Fracture
[WALKING] Walking into Impacted fracture
[WALKING] Walking into Longitudinal fracture
[WALKING] Walking into Oblique fracture
[WALKING] Walking into Pathological fracture
[WALKING] Walking into Spiral Fracture
[TEST DATA LOADED] Data loaded complete 140 Labeles are 140
```

(2) leave this cell because we need to create a val dataset and it is not there as default

In [30]:
```
# here you can split validation in stead of test so you can preserve intacted the test set.
train_data_handler, validation_data_handler, train_labels_handler, validation_labels_handler = train_test_split(np.ar

print(f"[TRAIN DATA] The train data is {train_data_handler.shape}  and its labels are {train_labels_handler.shape}")
print(f"[VALIDATION DATA] The validation data is {validation_data_handler.shape}  and its labels are {validation_labe

# my data loader need list yours? adapt it to your code
train_data_handler = list(train_data_handler)
train_labels_handler = list(train_labels_handler)

classes = len(class_names)

validation_data_handler = list(validation_data_handler)
validation_labels_handler = list(validation_labels_handler)
```

```
[TRAIN DATA] The train data is (791,)  and its labels are (791,)
[VALIDATION DATA] The validation data is (198,)  and its labels are (198,)
```

(3) Here ate two transforms one for training and one for evaluation

train transform need data augmentation:

- add random flip
- add random rotation
- add color jitter

both need need:

- resize
- toTensor handler
- Normalize (use the provided MEAN and STD)

Note that the evaluation transform does not need data augmentation.

In [31]:
```
TARGET_H = 224
TARGET_W = 224
IMAGE_MEAN = [0.485,0.456,0.406]
IMAGE_STD = [0.229,0.224,0.225]
```

```
# create transform
train_transform = transforms.Compose([
    transforms.Resize((TARGET_H, TARGET_W)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=8),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.05),
    transforms.ToTensor(),
    transforms.Normalize(mean=IMAGE_MEAN, std=IMAGE_STD)
])

# evaluation transform without data augmentation
evaluation_transform = transforms.Compose([
    transforms.Resize((TARGET_H, TARGET_W)),
    transforms.ToTensor(),
    transforms.Normalize(mean=IMAGE_MEAN, std=IMAGE_STD)
])
```

In [32]:

```
class MyDataset(Dataset):
    def __init__(self, list_of_pathes, list_of_classes, transform=None) -> None:
        super().__init__()
        self.list_of_pathes = list_of_pathes
        self.list_of_classes = list_of_classes
        self.transform = transform

    def __len__(self):
        return len(self.list_of_pathes)

    def __getitem__(self, index):
        # select the image path at index
        image_path = self.list_of_pathes[index]
        # select the label at index
        label = self.list_of_classes[index]
        # open the image using PIL
        image = Image.open(image_path).convert('RGB')
        # apply transform
        if self.transform:
            image_as_tensor = self.transform(image)
        else:
            image_as_tensor = transforms.ToTensor()(image)
        # return image and label
        return image_as_tensor, label
```

(4) Create data loder from the MyDataset class

In [33]:
```
BATCH_SIZE = 32
train_dataset = MyDataset(list_of_pathes=train_data_handler,
                          list_of_classes=train_labels_handler,
                          transform=train_transform)

validation_dataset = MyDataset(list_of_pathes=validation_data_handler,
                               list_of_classes=validation_labels_handler,
                               transform=evaluation_transform)

test_dataset = MyDataset(list_of_pathes=test_data_handler,
                          list_of_classes=test_labels_handler,
                          transform=evaluation_transform)

# put those here
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
validation_loader = DataLoader(validation_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=0)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=0)

print(f"[INFO] Train batches: {len(train_loader)}")
print(f"[INFO] Validation batches: {len(validation_loader)}")
print(f"[INFO] Test batches: {len(test_loader)}")
```

```
[INFO] Train batches: 25
[INFO] Validation batches: 7
[INFO] Test batches: 5
```

Instance of the CNN

Use the Sequential API it is faster

- This cnn has two hidden layers (conv2d -> proper activation -> max pool 2d) one has 16 untis and the second has 32 units
- Add a flatten layer
- Classifier need 128 Linear unit proper activation and set dropdown at 0.5
- The last layer of the classifier how many units need?

In [ ]:
```
#

model_2_layers = nn.Sequential(
    # First convolution layer
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Second convolution layer
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Flatten
    nn.Flatten(),

    # Classifier
    # After two max poolings: 224/4 = 56, so 56*56*32 = 100352 features combined to one layer to produce 128 features
    nn.Linear(56 * 56 * 32, 128),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(128, classes) # output layer with number of classes as output features
)

model_2_layers = model_2_layers.to(device)
print(model_2_layers)
print(f"\nTotal parameters: {sum(p.numel() for p in model_2_layers.parameters())}")
```

```
Sequential(
  (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): ReLU()
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Flatten(start_dim=1, end_dim=-1)
  (7): Linear(in_features=100352, out_features=128, bias=True)
  (8): ReLU()
  (9): Dropout(p=0.5, inplace=False)
  (10): Linear(in_features=128, out_features=10, bias=True)
)

Total parameters: 12851562
```

(5) Add proper optimizer and loss function set training epochs to 50

In [35]:
```
EPOCHS = 50
LEARNING_RATE = 0.0001

"learning rate 0.001 peformed poorly"

criterion = nn.CrossEntropyLoss()
optimizer_2_layers = optim.Adam(model_2_layers.parameters(), lr=LEARNING_RATE)

print(f"Training for {EPOCHS} epochs with learning rate {LEARNING_RATE}")
```

```
Training for 50 epochs with learning rate 0.0001
```

(6) add here the train and evaluation loop collect loss and accuracy, we want to see curves here

In [ ]:
```
# Training function
def train_one_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    total_loss = 0.0
    correct = 0
    total = 0
```

```
    for images, labels in dataloader:
        # Tensors to GPU
        images = images.to(device)
        labels = labels.to(device)

        # Reset gradient for each batch. Forward pass feeds images to the model. Backward pass computes gradient. Opt
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Storing to total loss and accuracy calculation
        total_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    avg_loss = total_loss / len(dataloader)
    accuracy = 100 * correct / total
    return avg_loss, accuracy

# Evaluation function
def evaluate(model, dataloader, criterion, device):
    model.eval()
    total_loss = 0.0
    correct = 0
    total = 0

    # no_grad saves computation while evaluating
    with torch.no_grad():
        for images, labels in dataloader:
            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)

            total_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
```

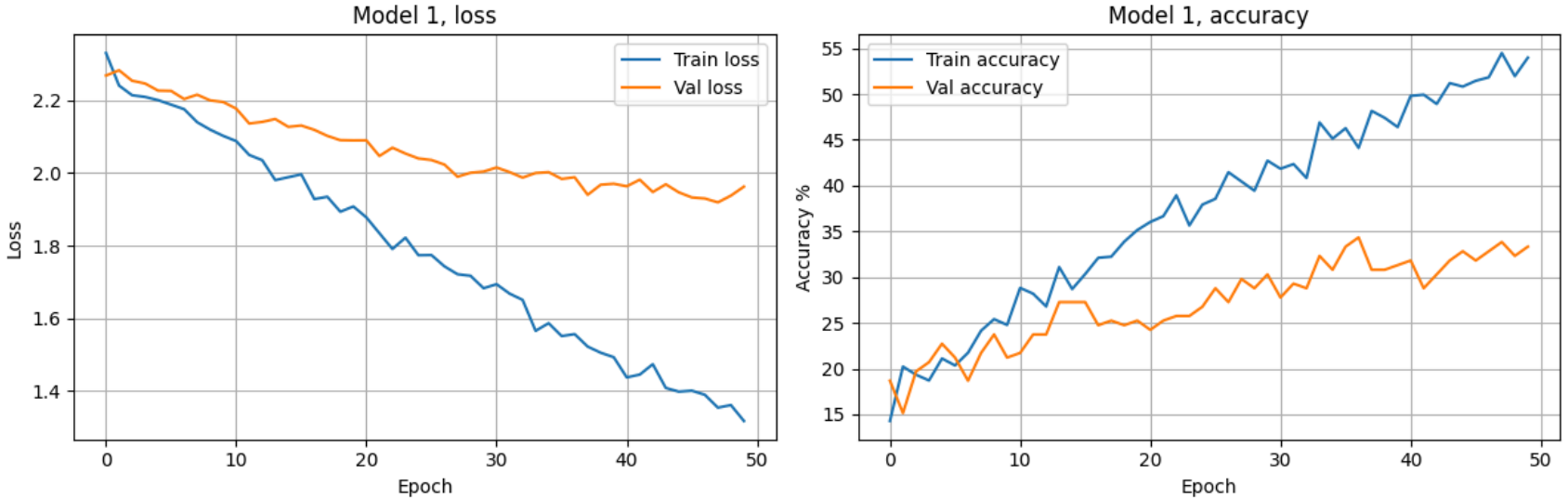
```
            correct += (predicted == labels).sum().item()

    avg_loss = total_loss / len(dataloader)
    accuracy = 100 * correct / total
    return avg_loss, accuracy
```

In [37]:
```
# Training loop for Model 1
train_losses_2 = []
train_accs_2 = []
val_losses_2 = []
val_accs_2 = []

print("Training for model 1 (2 layers)...\n")

for epoch in range(EPOCHS):
    train_loss, train_acc = train_one_epoch(model_2_layers, train_loader, criterion, optimizer_2_layers, device)
    train_losses_2.append(train_loss)
    train_accs_2.append(train_acc)

    val_loss, val_acc = evaluate(model_2_layers, validation_loader, criterion, device)
    val_losses_2.append(val_loss)
    val_accs_2.append(val_acc)

    # Print progress every 10 epochs to see if model still learns/works
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}/{EPOCHS}: Train loss={train_loss:.4f}, Train acc={train_acc:.2f}%, Val loss={val_loss

print("\nTraining complete!")
```

```
Training for model 1 (2 layers)...

Epoch 10/50: Train loss=2.1022, Train acc=24.78%, Val loss=2.1954, Val acc=21.21%
Epoch 20/50: Train loss=1.9080, Train acc=35.15%, Val loss=2.0897, Val acc=25.25%
Epoch 30/50: Train loss=1.6830, Train acc=42.73%, Val loss=2.0039, Val acc=30.30%
Epoch 40/50: Train loss=1.4931, Train acc=46.40%, Val loss=1.9704, Val acc=31.31%
Epoch 50/50: Train loss=1.3176, Train acc=53.98%, Val loss=1.9624, Val acc=33.33%

Training complete!
```

(7) Plot training curves

In [38]:
```
# Plot loss and accuracy curves
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(train_losses_2, label='Train loss')
ax1.plot(val_losses_2, label='Val loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.set_title('Model 1, loss')
ax1.legend()
ax1.grid(True)

ax2.plot(train_accs_2, label='Train accuracy')
ax2.plot(val_accs_2, label='Val accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy %')
ax2.set_title('Model 1, accuracy')
ax2.legend()
ax2.grid(True)

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

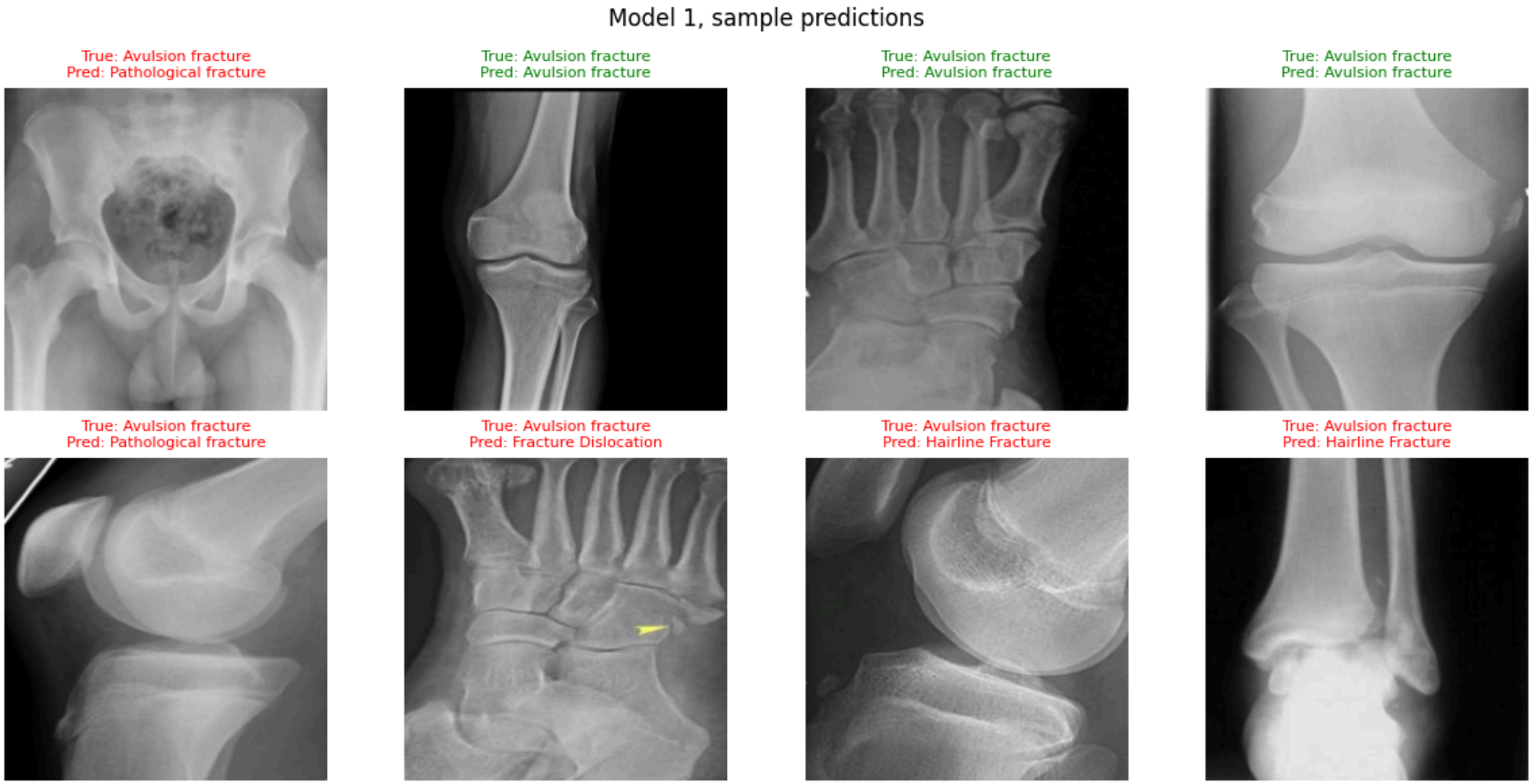


(8) Make predictions and show some predicted images

In [ ]:
```python
# Helper function to denormalize images, otherwise they look weird in plots. Range(3) for RGB channels.
def denormalize(img, mean, std):
    img_copy = img.clone()
    for i in range(3):
        img_copy[i] = img_copy[i] * std[i] + mean[i]
    return img_copy

# Create reverse mapping
idx_to_class = {v: k for k, v in class_names.items()}
```

In [ ]:
```python
# Get predictions from some images
model_2_layers.eval()
test_iter = iter(test_loader)
images, labels = next(test_iter)

images = images.to(device)
with torch.no_grad():
    outputs = model_2_layers(images)
    _, predictions = torch.max(outputs, 1)

# For visualization, move data back to CPU
images = images.cpu()
predictions = predictions.cpu()

# Plot sample pictures with true and predicted labels
fig, axes = plt.subplots(2, 4, figsize=(12, 6))
axes = axes.flatten()

for idx in range(8):
    img = denormalize(images[idx], IMAGE_MEAN, IMAGE_STD)
    img = img.permute(1, 2, 0).numpy()
    img = np.clip(img, 0, 1)

    axes[idx].imshow(img)
    axes[idx].axis('off')

    true_label = idx_to_class[labels[idx].item()]
    pred_label = idx_to_class[predictions[idx].item()]
    color = 'green' if labels[idx] == predictions[idx] else 'red'
    axes[idx].set_title(f'True: {true_label}\nPred: {pred_label}', color=color, fontsize=8)
```

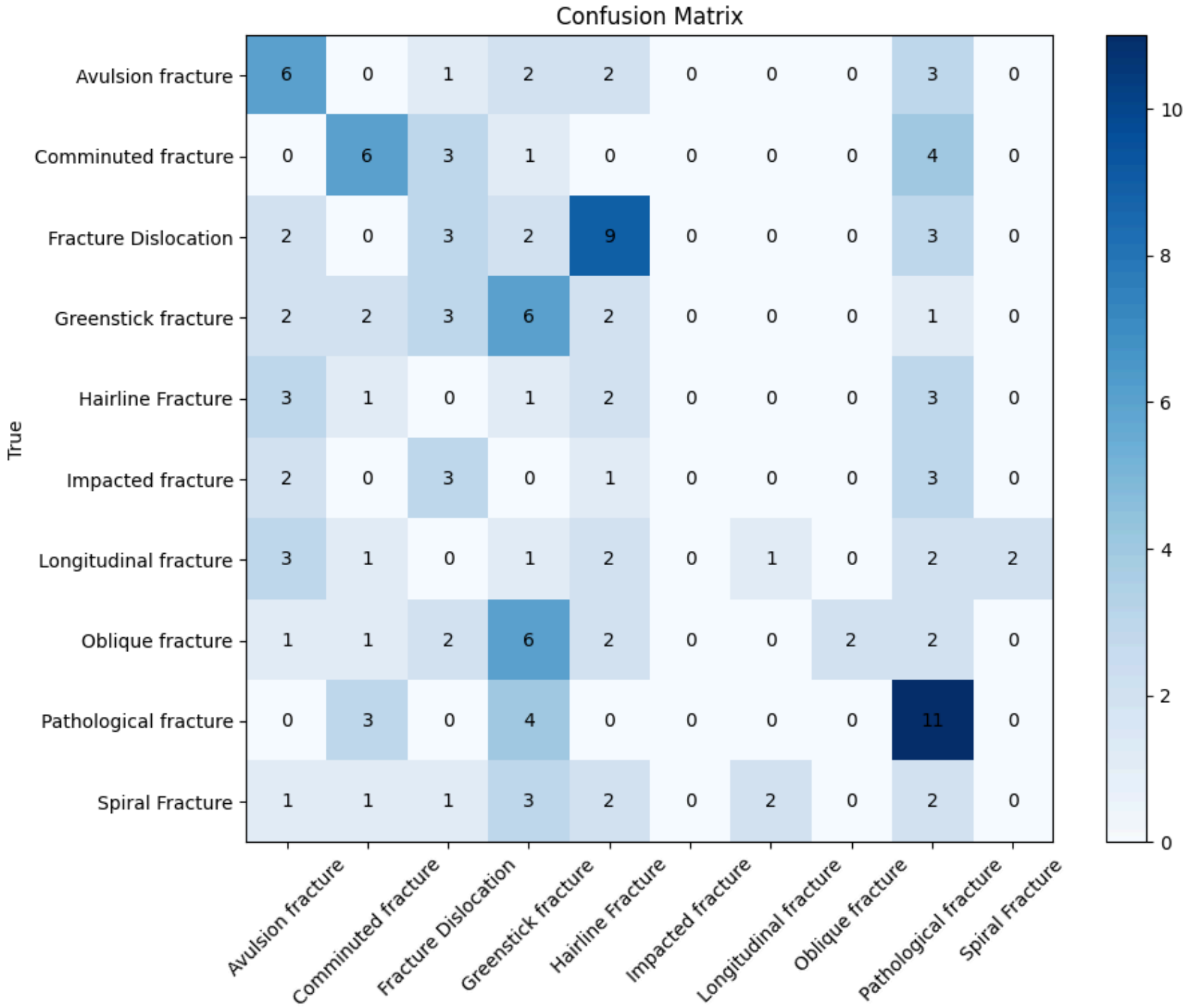
```
plt.suptitle('Model 1, sample predictions')
plt.tight_layout()
plt.show()
```



(9) Confusion Matrix

In [ ]:
```
# Get all predictions on test set
model_2_layers.eval()
all_predictions = []
all_labels = []

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

```
        outputs = model_2_layers(images)
        _, predictions = torch.max(outputs, 1)

        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(labels.numpy())

# Compute confusion matrix
cm = confusion_matrix(all_labels, all_predictions)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
plt.imshow(cm, cmap='Blues')
plt.colorbar()

# Add numbers in cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center')

# Add labels
plt.xticks(range(len(class_names)), class_names.keys(), rotation=45)
plt.yticks(range(len(class_names)), class_names.keys())
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()

# Calculate test accuracy
test_loss, test_acc = evaluate(model_2_layers, test_loader, criterion, device)
print(f"\n[RESULTS] Model 1 test loss: {test_loss:.4f}, test accuracy: {test_acc:.2f}%")
```


Confusion Matrix
True
Avulsion fracture
Comminuted fracture
Fracture Dislocation
Greenstick fracture
Hairline Fracture
Impacted fracture
Longitudinal fracture
Oblique fracture
Pathological fracture
Spiral Fracture
6 0 1 2 2 0 0 0 3 0
0 6 3 1 0 0 0 0 4 0
2 0 3 2 9 0 0 0 3 0
2 2 3 6 2 0 0 0 1 0
3 1 0 1 2 0 0 0 3 0
2 0 3 0 1 0 0 0 3 0
3 1 0 1 2 0 1 0 2 2
1 1 2 6 2 0 0 2 2 0
0 3 0 4 0 0 0 0 11 0
1 1 1 3 2 0 2 0 2 0
0
2
4
6
8
10

Predicted

```
[RESULTS] Model 1 test loss: 2.5983, test accuracy: 26.43%
```

Model 2: Three Hidden Layers CNN

In [42]:
```python
# Instance of the CNN with 3 hidden layers
# Conv layers: 16 units, 32 units, 64 units
# Classifier: 128 units with dropout 0.5

model_3_layers = nn.Sequential(
    # First convolutional block
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Second convolutional block
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Third convolutional block
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Flatten
    nn.Flatten(),

    # Classifier
    # After three max poolings: 224/8 = 28, so 28*28*64 = 50176
    nn.Linear(28 * 28 * 64, 128),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(128, classes)
)

model_3_layers = model_3_layers.to(device)
print(model_3_layers)
print(f"\nInfo, total parameters: {sum(p.numel() for p in model_3_layers.parameters())}")
```

```
Sequential(
  (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): ReLU()
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (7): ReLU()
  (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (9): Flatten(start_dim=1, end_dim=-1)
  (10): Linear(in_features=50176, out_features=128, bias=True)
  (11): ReLU()
  (12): Dropout(p=0.5, inplace=False)
  (13): Linear(in_features=128, out_features=10, bias=True)
)

Info, total parameters: 6447530
```

(5) Setup optimizer and loss for Model 2

In [43]:
```
optimizer_3_layers = optim.Adam(model_3_layers.parameters(), lr=LEARNING_RATE)
print(f"Info, training {EPOCHS} epochs with learning rate {LEARNING_RATE}")
```

```
Info, training 50 epochs with learning rate 0.0001
```

(6) Training Loop for Model 2

In [44]:
```
# Training loop for Model 2
train_losses_3 = []
train_accs_3 = []
val_losses_3 = []
val_accs_3 = []

print("Training for model 2 (3 layers)...\n")

for epoch in range(EPOCHS):
    train_loss, train_acc = train_one_epoch(model_3_layers, train_loader, criterion, optimizer_3_layers, device)
    train_losses_3.append(train_loss)
    train_accs_3.append(train_acc)
```

```
    val_loss, val_acc = evaluate(model_3_layers, validation_loader, criterion, device)
    val_losses_3.append(val_loss)
    val_accs_3.append(val_acc)

    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}/{EPOCHS}: Train loss={train_loss:.4f}, Train acc={train_acc:.2f}%, Val loss={val_loss

print("\nTraining complete!")
```

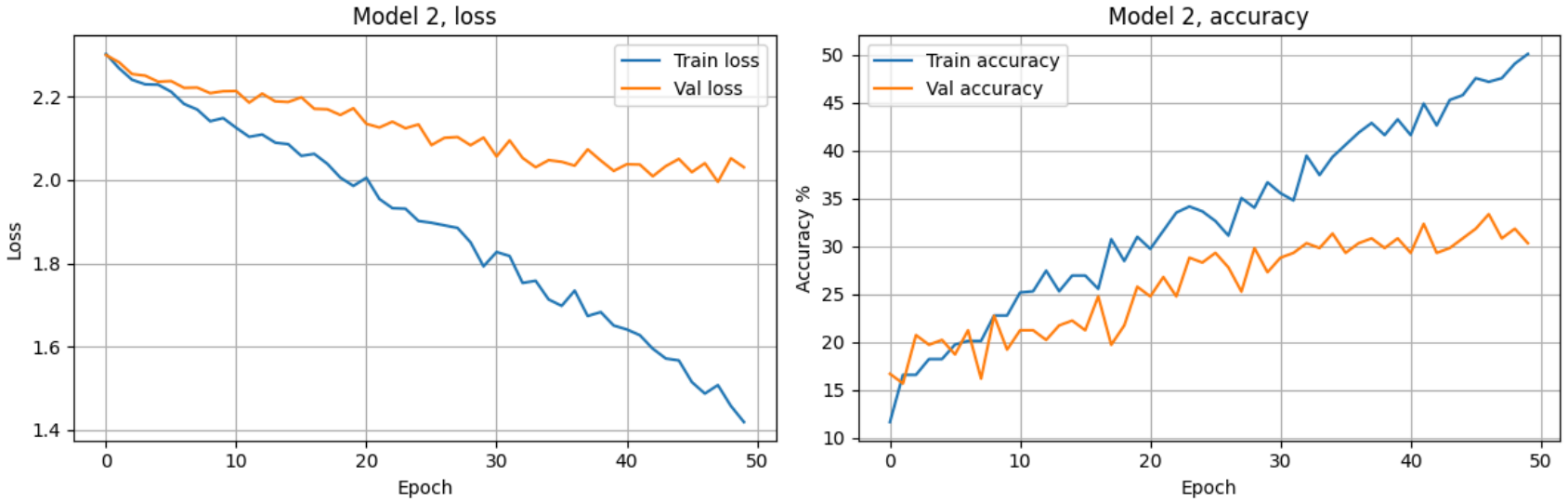
```
Training for model 2 (3 layers)...

Epoch 10/50: Train loss=2.1481, Train acc=22.76%, Val loss=2.2124, Val acc=19.19%
Epoch 20/50: Train loss=1.9854, Train acc=30.97%, Val loss=2.1716, Val acc=25.76%
Epoch 30/50: Train loss=1.7925, Train acc=36.66%, Val loss=2.1013, Val acc=27.27%
Epoch 40/50: Train loss=1.6506, Train acc=43.24%, Val loss=2.0214, Val acc=30.81%
Epoch 50/50: Train loss=1.4192, Train acc=50.06%, Val loss=2.0301, Val acc=30.30%

Training complete!
```

(7) Plot training curves for Model 2

In [45]:

```
# Plot loss and accuracy curves
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(train_losses_3, label='Train loss')
ax1.plot(val_losses_3, label='Val loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.set_title('Model 2, loss')
ax1.legend()
ax1.grid(True)

ax2.plot(train_accs_3, label='Train accuracy')
ax2.plot(val_accs_3, label='Val accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy %')
ax2.set_title('Model 2, accuracy')
ax2.legend()
ax2.grid(True)
```

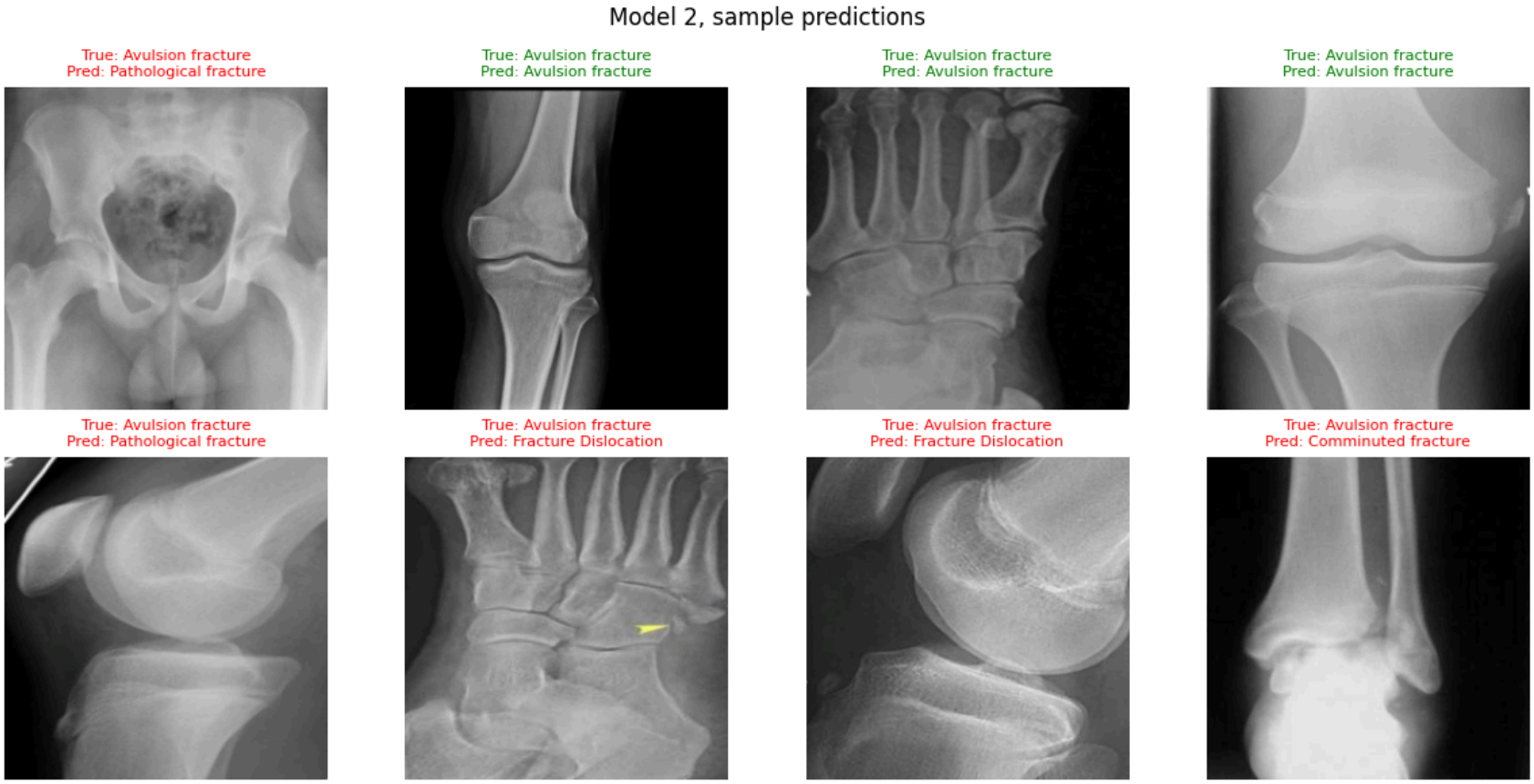
```
plt.tight_layout()
plt.show()
```



(8) Make predictions and show some predicted images - Model 2

In [46]:
```
# Get predictions on some test images
model_3_layers.eval()
test_iter = iter(test_loader)
images, labels = next(test_iter)

images = images.to(device)
with torch.no_grad():
    outputs = model_3_layers(images)
    _, predictions = torch.max(outputs, 1)

images = images.cpu()
predictions = predictions.cpu()

# Plot 8 sample predictions
fig, axes = plt.subplots(2, 4, figsize=(12, 6))
axes = axes.flatten()

for idx in range(8):
    img = denormalize(images[idx], IMAGE_MEAN, IMAGE_STD)
```

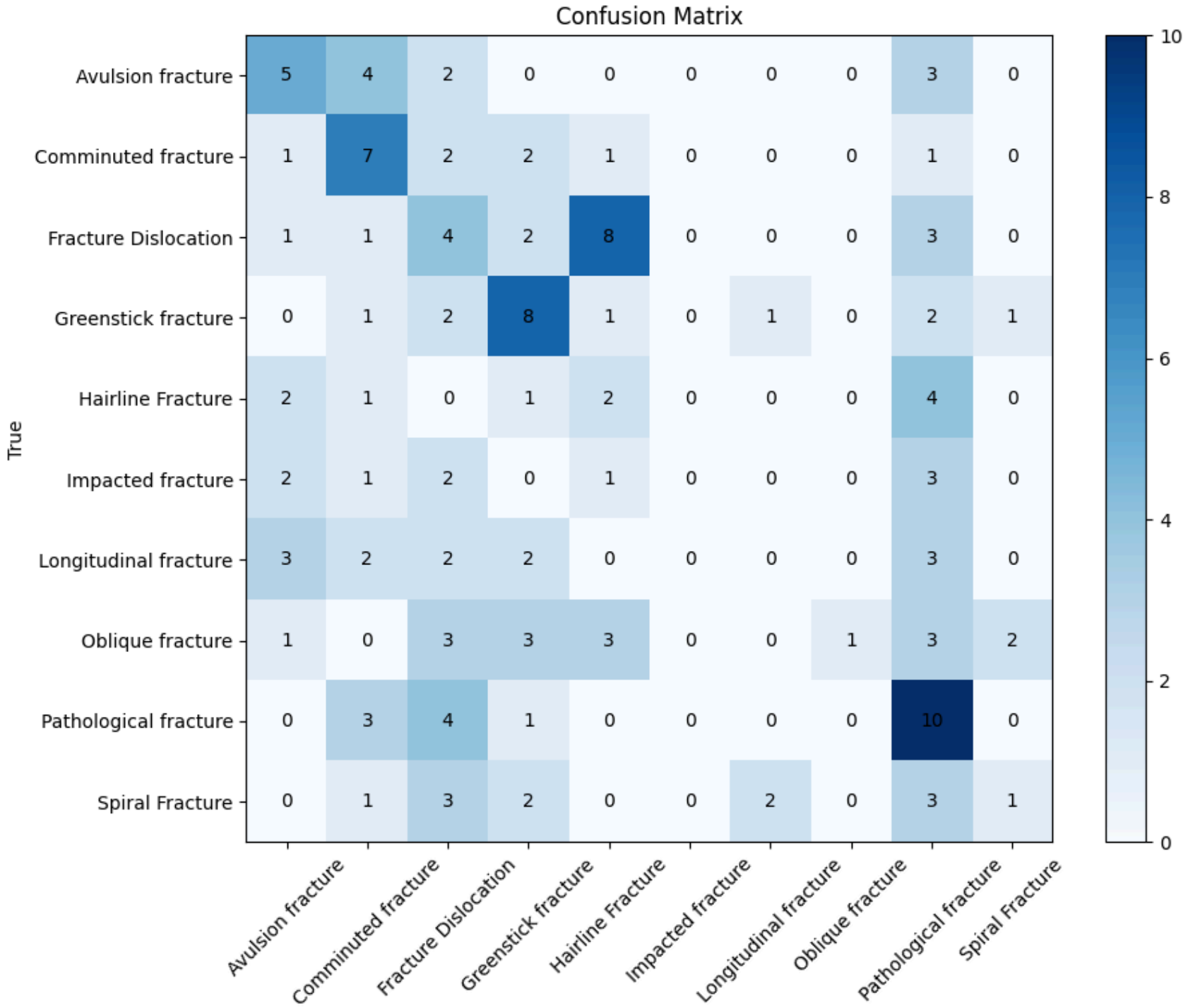
```
    img = img.permute(1, 2, 0).numpy()
    img = np.clip(img, 0, 1)

    axes[idx].imshow(img)
    axes[idx].axis('off')

    true_label = idx_to_class[labels[idx].item()]
    pred_label = idx_to_class[predictions[idx].item()]
    color = 'green' if labels[idx] == predictions[idx] else 'red'
    axes[idx].set_title(f'True: {true_label}\nPred: {pred_label}', color=color, fontsize=8)

plt.suptitle('Model 2, sample predictions')
plt.tight_layout()
plt.show()
```



(9) Confusion Matrix - Model 2

In [47]:
```
# Get all predictions on test set
model_3_layers.eval()
all_predictions = []
all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        outputs = model_3_layers(images)
        _, predictions = torch.max(outputs, 1)

        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(labels.numpy())

# Compute confusion matrix
cm = confusion_matrix(all_labels, all_predictions)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
plt.imshow(cm, cmap='Blues')
plt.colorbar()

# Add numbers in cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, str(cm[i, j]), ha='center', va='center')

# Add labels
plt.xticks(range(len(class_names)), class_names.keys(), rotation=45)
plt.yticks(range(len(class_names)), class_names.keys())
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()

# Calculate test accuracy
```

```
test_loss, test_acc = evaluate(model_3_layers, test_loader, criterion, device)
print(f"\n[RESULTS] Model 2 test loss: {test_loss:.4f}, test accuracy: {test_acc:.2f}%")
```


Confusion Matrix
True
Avulsion fracture
Comminuted fracture
Fracture Dislocation
Greenstick fracture
Hairline Fracture
Impacted fracture
Longitudinal fracture
Oblique fracture
Pathological fracture
Spiral Fracture
5 4 2 0 0 0 0 0 3 0
1 7 2 2 1 0 0 0 1 0
1 1 4 2 8 0 0 0 3 0
0 1 2 8 1 0 1 0 2 1
2 1 0 1 2 0 0 0 4 0
2 1 2 0 1 0 0 0 3 0
3 2 2 2 0 0 0 0 3 0
1 0 3 3 3 0 0 1 3 2
0 3 4 1 0 0 0 0 10 0
0 1 3 2 0 0 2 0 3 1
10
8
6
4
2
0

Predicted

```
[RESULTS] Model 2 test loss: 2.4188, test accuracy: 27.14%
```

Save Both Models

In [48]:
```python
# Save both models
torch.save(model_2_layers.state_dict(), 'cnn_2_layers.pt')
torch.save(model_3_layers.state_dict(), 'cnn_3_layers.pt')
print("[INFO] Models saved successfully!")
```

```
[INFO] Models saved successfully!
```