Final Mini-Project Info 3950

name: Sarah Benkoussa

netID: sb967

Using the pistachio images dataset found here: https://www.muratkoklu.com/datasets/

This project aims to explore and evaluate how different models and architectures can classify images of pistachio breeds.

Imports

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import cv2
import os
import random
import torchvision
from torchvision import datasets, models, transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader, random_split
import torch, torch.nn as nn, torch.nn.functional as F
```

Data

Data Exploration

In [2]:

```
from zipfile import ZipFile
# loading the temp.zip and creating a zip object
with ZipFile("BothPistachio.zip", 'r') as zObject:
    # Extracting all the members of the zip into a specific location.
    zObject.extractall(path="BothPistachios")
```

In [5]:

```
Kirmizi_img = []

for fname in os.listdir("BothPistachios/Kirmizi_Pistachio"):
    if fname.endswith('.jpg'):
        impath = os.path.join("Kirmizi_Pistachio", fname)
        tmp = cv2.imread(impath)
        Kirmizi_img.append(tmp)

Siirt_img = []

for fname in os.listdir("BothPistachios/Siirt_Pistachio"):
    if fname.endswith('.jpg'):
        impath = os.path.join("Siirt_Pistachio", fname)
        tmp = cv2.imread(impath)
        Siirt_img.append(tmp)
```

In [6]:

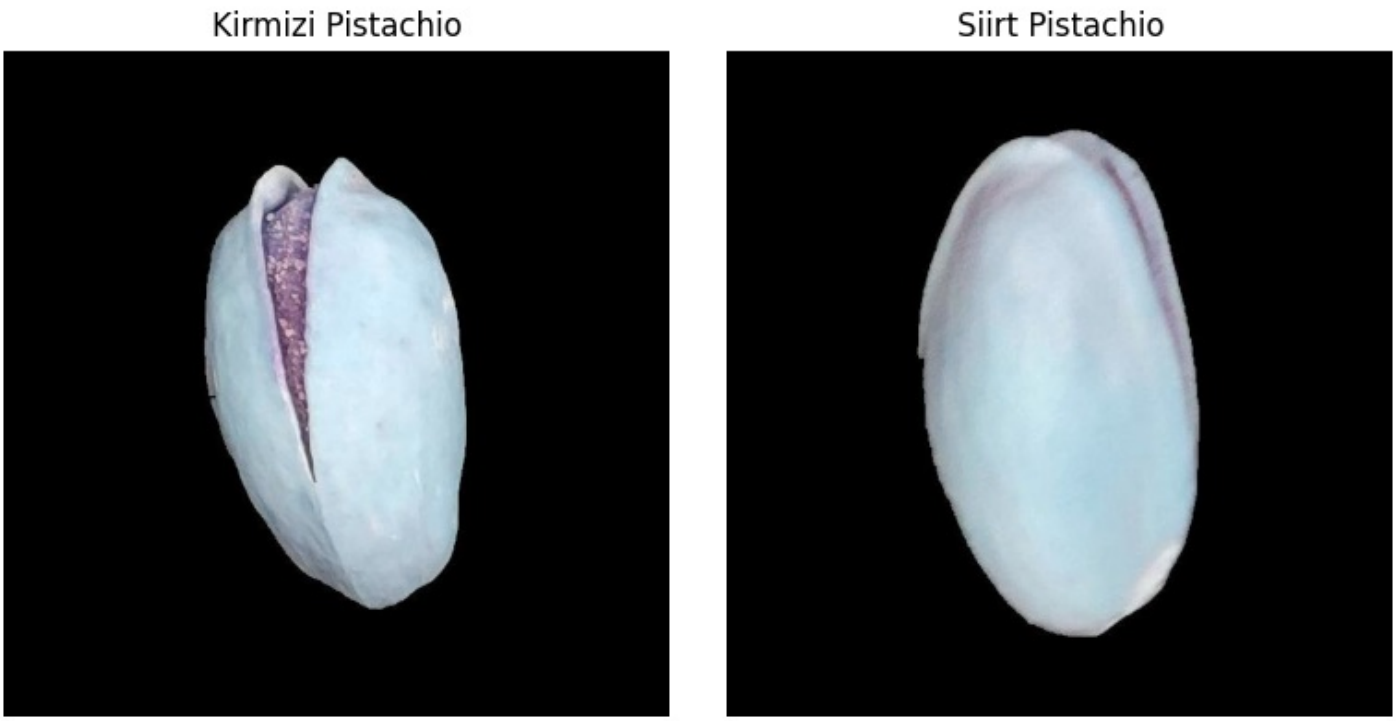
```
# How many images of each type we have
```

```
print("Kirmizi_Pistachio counts: ", len(Kirmizi_img))
print("Siirt_Pistachio counts: ", len(Siirt_img))
```

```
Kirmizi_Pistachio counts:  1232
Siirt_Pistachio counts:  916
```

In [16]:

```
# show image of random pistachio from each category

fig, axes = plt.subplots(1, 2, figsize=(8, 4))

axes[0].imshow(Kirmizi_img[random.randrange(len(Kirmizi_img))])
axes[0].set_title("Kirmizi Pistachio")
axes[0].axis('off')

axes[1].imshow(Siirt_img[random.randrange(len(Siirt_img))]) #(image * 0.5) + 0.5
axes[1].set_title("Siirt Pistachio")
axes[1].axis('off')

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



To be honest, I'm not sure I would be able to distinguish one type of pistachio from another. Surely the models can do it.

Data Loading

In [7]:

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])
# Load dataset with ImageFolder
dataset = ImageFolder(root='BothPistachios', transform=transform)

# Split the dataset into training and test sets
train_size = int(0.75 * len(dataset))  # 75% for training
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

# Create data loaders for training and test sets
```

```
batch_size = 64    # 32?
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Model 1: simple CNN

For the first iteration of modeling, I chose to create a very minimal/uncomplicated CNN to preemptively prevent overfitting on the training data (since the dataset is quite small). Evaluating this simpler CNN gives a better idea of how complex the model may need to be later on.

Build Model 1

In [20]:

```
model1 = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Flatten(),

    nn.Linear(394272, 2)
)

learning_rate = 1e-3;
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model1.parameters(), lr=learning_rate)

model1
```

Out[20]:

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Flatten(start_dim=1, end_dim=-1)
  (4): Linear(in_features=394272, out_features=2, bias=True)
)
```

Train Model 1

In [21]:

```
epochs = 10 # 20
scores1 = []

for t in range(epochs):
    train_loss = []
    train_accuracy = []
    print(f'Epoch {t+1}: ', end='')


    for i,(x, y) in enumerate(train_loader):
        # Compute prediction and loss
        pred = model1(x)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
        train_accuracy.append((pred.argmax(1) == y).sum().item() / len(pred))


    test_loss = []
    test_accuracy = []
```

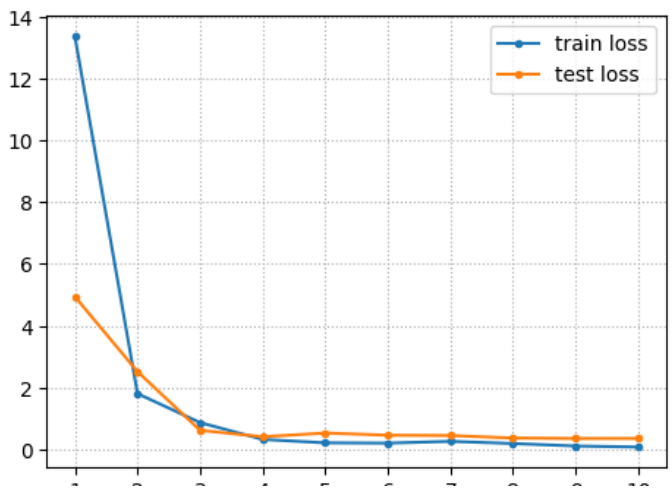
```
    for i, (x, y) in enumerate(test_loader):
        pred = model1(x)
        loss = loss_fn(pred, y)
        test_loss.append(loss.item())
        test_accuracy.append((pred.argmax(1) == y).sum().item() / len(pred))

    print(f'epoch: {t}, train loss: {np.mean(train_loss):.5f}, test loss: {np.mean(test_
loss):.5f}, ' +\
          f'train_score: {np.mean(train_accuracy):.1%}, test_score: {np.mean(test_accura
cy):.1%}')
    scores1.append((np.mean(train_loss),np.mean(test_loss),np.mean(train_accuracy),np.me
an(test_accuracy)))
```

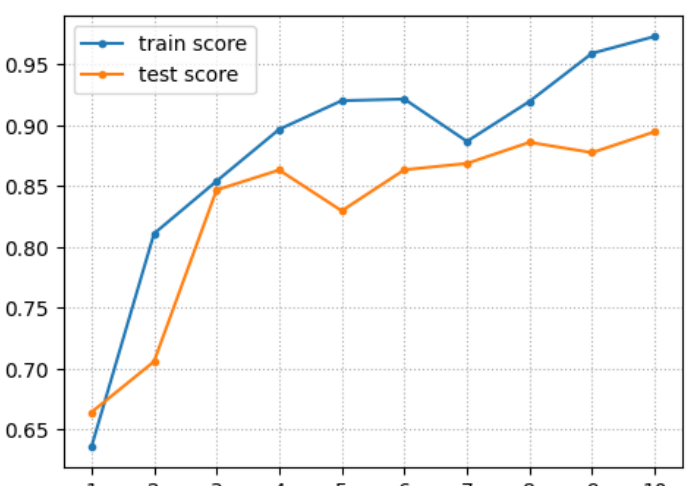
```
Epoch 1: epoch: 0, train loss: 13.37265, test loss: 4.94246, train_score: 63.6%, test_sco
re: 66.4%
Epoch 2: epoch: 1, train loss: 1.80219, test loss: 2.51440, train_score: 81.1%, test_scor
e: 70.6%
Epoch 3: epoch: 2, train loss: 0.86265, test loss: 0.61255, train_score: 85.4%, test_scor
e: 84.7%
Epoch 4: epoch: 3, train loss: 0.31773, test loss: 0.40270, train_score: 89.7%, test_scor
e: 86.3%
Epoch 5: epoch: 4, train loss: 0.20677, test loss: 0.52494, train_score: 92.0%, test_scor
e: 83.0%
Epoch 6: epoch: 5, train loss: 0.19756, test loss: 0.45407, train_score: 92.1%, test_scor
e: 86.3%
Epoch 7: epoch: 6, train loss: 0.25483, test loss: 0.44281, train_score: 88.7%, test_scor
e: 86.9%
Epoch 8: epoch: 7, train loss: 0.18469, test loss: 0.36291, train_score: 92.0%, test_scor
e: 88.6%
Epoch 9: epoch: 8, train loss: 0.10410, test loss: 0.34755, train_score: 95.9%, test_scor
e: 87.8%
Epoch 10: epoch: 9, train loss: 0.07105, test loss: 0.34885, train_score: 97.3%, test_sco
re: 89.5%
```

Evaluating Model 1

In [23]:

```
scores1n = np.array(scores1)
plt.figure(figsize=(12,4))

plt.subplot(121)
plt.plot(scores1n.T[0], '.-', label='train loss')
plt.plot(scores1n.T[1], '.-', label='test loss')
plt.xticks(range(10), range(1,11))
plt.legend()
plt.grid(ls=':');

plt.subplot(122)
plt.plot(scores1n.T[2], '.-', label='train score')
plt.plot(scores1n.T[3], '.-', label='test score')
plt.xticks(range(10), range(1,11))
plt.legend()
plt.grid(ls=':');
```





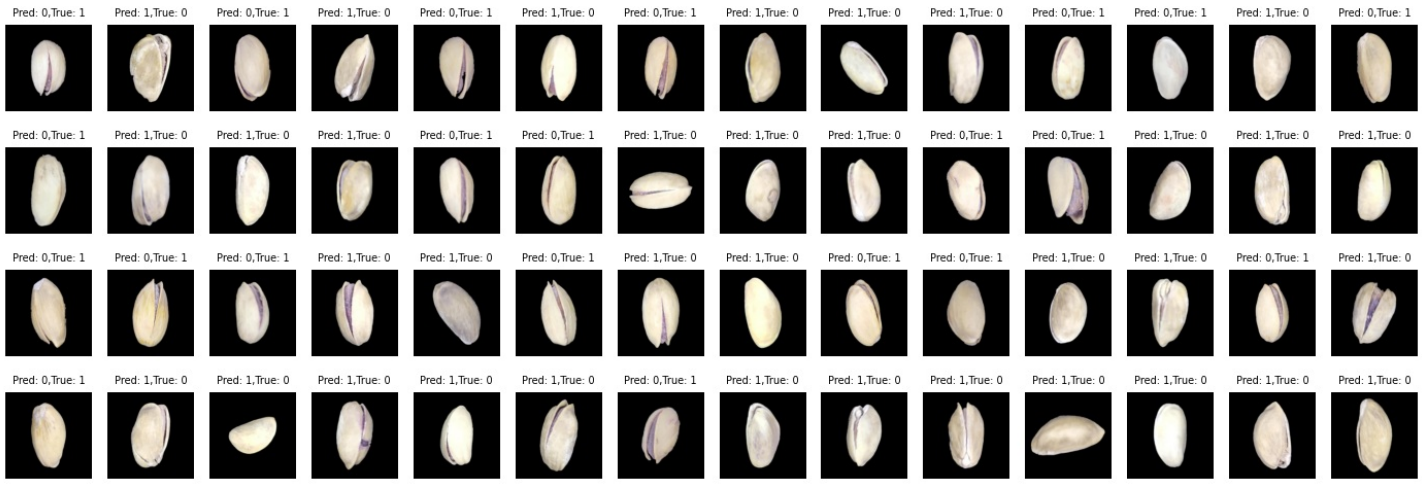
In [33]:

```
misclassified_images = []
misclassified_labels = []
misclassified_predictions = []

# Iterate over the test dataset
for images, labels in test_loader:
    # predict
    outputs = model1(images)
    predicted_labels = torch.argmax(outputs, dim=1)
    # find incorrectly classified images
    misclassified_mask = predicted_labels != labels
    misclassified_images.extend(images[misclassified_mask])
    misclassified_labels.extend(labels[misclassified_mask])
    misclassified_predictions.extend(predicted_labels[misclassified_mask])

print("number of incorrectly classified images with Model 1: ", len(misclassified_images)
)
```

```
number of incorrectly classified images with Model 1:  56
```

In [49]:

```
# where 0 = "Kirmizi", 1 = "Siirt"

# Display misclassified images
fig, axes = plt.subplots(4, 14, figsize=(14, 5))
axes = axes.flatten()

for i in range(len(misclassified_images)):
    image = misclassified_images[i].cpu().numpy().transpose(1, 2, 0)  # Convert tensor t
o NumPy array
    image = (image * 0.5) + 0.5  # Unnormalize image
    axes[i].imshow(image)
    axes[i].axis('off')
    axes[i].set_title(f"Pred: {misclassified_predictions[i]},True: {misclassified_labels[
i]}", fontsize=7)

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



Train and test accuracies are generally not too far apart from each other, though test scores never seem to break into the 90's or above.

Let's see if we can increase the complexity of the model just enough to gain a better test score without overfitting on the train data.

Model 2: CNN with more complex architecture

The simple model couldn't raise test accuracy to be over 90%. Perhaps introducing more layers will make it

The simple model couldn't raise test accuracy to be over 90%. Perhaps introducing more layers will make it possible by creating more complex features. However, adding more layers also risks overfitting the data, so will it be successful?

Build Model 2

In [42]:

```
model2 = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),

    nn.Conv2d(32, 32, kernel_size=3),
    nn.ReLU(),
    nn.BatchNorm2d(32),

    nn.Flatten(),

    nn.Linear(380192, 37),
    nn.ReLU(),
    nn.BatchNorm1d(37),

    nn.Linear(37, 2)
)

learning_rate = 1e-3;
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model2.parameters(), lr=learning_rate)

model2
```

Out[42]:

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
  (4): ReLU()
  (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): Flatten(start_dim=1, end_dim=-1)
  (7): Linear(in_features=380192, out_features=37, bias=True)
  (8): ReLU()
  (9): BatchNorm1d(37, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (10): Linear(in_features=37, out_features=2, bias=True)
)
```

Train Model 2

In [43]:

```
epochs = 10 # 20
scores2 = []

for t in range(epochs):
    train_loss = []
    train_accuracy = []
    print(f'Epoch {t+1}: ', end='')


    for i,(x, y) in enumerate(train_loader):
        # Compute prediction and loss
        pred = model2(x)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
```

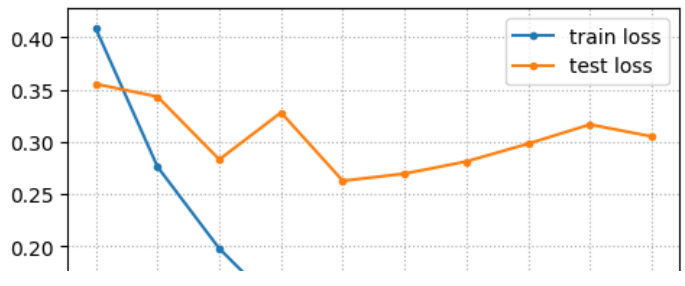
```
        optimizer.step()
        train_loss.append(loss.item())
        train_accuracy.append((pred.argmax(1) == y).sum().item() / len(pred))


    test_loss = []
    test_accuracy = []
    for i, (x, y) in enumerate(test_loader):
        pred = model2(x)
        loss = loss_fn(pred, y)
        test_loss.append(loss.item())
        test_accuracy.append((pred.argmax(1) == y).sum().item() / len(pred))

    print(f'epoch: {t}, train loss: {np.mean(train_loss):.5f}, test loss: {np.mean(test_
loss):.5f}, ' +\
          f'train_score: {np.mean(train_accuracy):.1%}, test_score: {np.mean(test_accura
cy):.1%}')
    scores2.append((np.mean(train_loss),np.mean(test_loss),np.mean(train_accuracy),np.me
an(test_accuracy)))
```

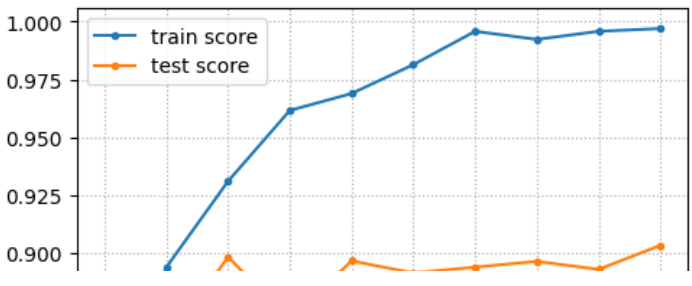
```
Epoch 1: epoch: 0, train loss: 0.40838, test loss: 0.35536, train_score: 82.2%, test_scor
e: 86.3%
Epoch 2: epoch: 1, train loss: 0.27611, test loss: 0.34321, train_score: 89.4%, test_scor
e: 86.0%
Epoch 3: epoch: 2, train loss: 0.19792, test loss: 0.28313, train_score: 93.1%, test_scor
e: 89.8%
Epoch 4: epoch: 3, train loss: 0.13919, test loss: 0.32804, train_score: 96.2%, test_scor
e: 87.1%
Epoch 5: epoch: 4, train loss: 0.10947, test loss: 0.26271, train_score: 96.9%, test_scor
e: 89.6%
Epoch 6: epoch: 5, train loss: 0.07742, test loss: 0.26958, train_score: 98.1%, test_scor
e: 89.1%
Epoch 7: epoch: 6, train loss: 0.04262, test loss: 0.28122, train_score: 99.6%, test_scor
e: 89.4%
Epoch 8: epoch: 7, train loss: 0.03842, test loss: 0.29821, train_score: 99.2%, test_scor
e: 89.6%
Epoch 9: epoch: 8, train loss: 0.03041, test loss: 0.31661, train_score: 99.6%, test_scor
e: 89.3%
Epoch 10: epoch: 9, train loss: 0.02230, test loss: 0.30518, train_score: 99.7%, test_sco
re: 90.3%
```

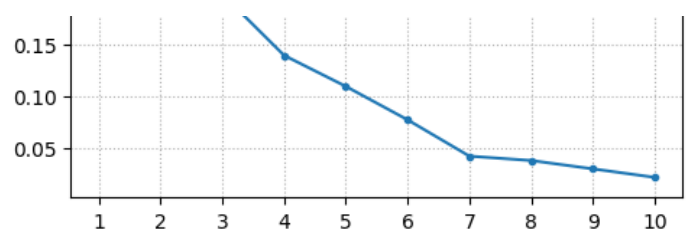
Evaluating Model 2

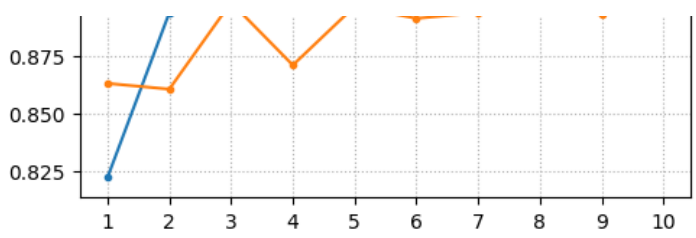
In [47]:

```
scores2n = np.array(scores2)
plt.figure(figsize=(12,4))

plt.subplot(121)
plt.plot(scores2n.T[0], '.-', label='train loss')
plt.plot(scores2n.T[1], '.-', label='test loss')
plt.xticks(range(10), range(1,11))
plt.legend()
plt.grid(ls=':');

plt.subplot(122)
plt.plot(scores2n.T[2], '.-', label='train score')
plt.plot(scores2n.T[3], '.-', label='test score')
plt.xticks(range(10), range(1,11))
plt.legend()
plt.grid(ls=':');
```









The increased complexity actually did not impact the test accuracy very significantly. While model 2's test scores are marginally higher, it is clear that the biggest impact is seen in the train score improvements. This means that the added complexity did indeed end up unnecessarily overfitting the training data. Model 1's test scores are essentially the same as Model 2, despite being much simpler in terms of its architecture.

In [48]:

```
misclassified_images2 = []
misclassified_labels2 = []
misclassified_predictions2 = []

# Iterate over the test dataset
for images, labels in test_loader:
    # predict
    outputs = model2(images)
    predicted_labels = torch.argmax(outputs, dim=1)
    # find incorrectly classified images
    misclassified_mask = predicted_labels != labels
    misclassified_images2.extend(images[misclassified_mask])
    misclassified_labels2.extend(labels[misclassified_mask])
    misclassified_predictions2.extend(predicted_labels[misclassified_mask])

print("number of incorrectly classified images with Model 2: ", len(misclassified_images2
))
```

```
number of incorrectly classified images with Model 2:  48
```

In [50]:

```
# where 0 = "Kirmizi", 1 = "Siirt"

# Display misclassified images
fig, axes = plt.subplots(4, 12, figsize=(14, 5))
axes = axes.flatten()

for i in range(len(misclassified_images2)):
    image = misclassified_images2[i].cpu().numpy().transpose(1, 2, 0)  # Convert tensor
to NumPy array
    image = (image * 0.5) + 0.5  # Unnormalize image
    axes[i].imshow(image)
    axes[i].axis('off')
    axes[i].set_title(f"Pred: {misclassified_predictions2[i]}, True: {misclassified_label
s2[i]}", fontsize=7)

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