In [1]:

```
import os
import torch
import yaml
import glob
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import torchvision
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
from PIL import Image
import numpy as np
import seaborn as sns
import pandas as pd
```

In [2]:

```
def set_seed(seed):
    torch.manual_seed(seed)
    np.random.seed(seed)

    # for cuda
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
```

In [3]:

```
set_seed(0)
```

In [4]:

```
def extract_files():
    import google.colab
    import zipfile

    google.colab.drive.mount('/content/drive')
    PROJECT_DIR = "/content/drive/MyDrive/thesis/data/"

    zip_ref = zipfile.ZipFile(PROJECT_DIR + "fiveK.zip", 'r')
    zip_ref.extractall(".")
    zip_ref.close()
```

In [5]:

```
if 'google.colab' in str(get_ipython()):
```

```
  extract_files()
  config_path = "/content/drive/MyDrive/thesis/config.yaml"
else:
  config_path = "../../config.yaml"
```

```
Mounted at /content/drive
```

In [6]:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cpu
```

In [7]:

```
try:
    # Load configuration
    with open(config_path, 'r') as config_file:
        config = yaml.safe_load(config_file)
except:
    raise FileNotFoundError(f"Config file not found at path: {config_path}")
```

In [8]:

```
base_checkpoint_path = config['paths']['checkpoints']
```

In [9]:

```
def load_best_checkpoint(checkpoint_dir):
    # Check if the directory exists
    if not os.path.exists(base_checkpoint_path):
        print(f"No directory found: {checkpoint_dir}")
        return None
      # Get a list of all checkpoint files in the directory
    checkpoint_files = glob.glob(os.path.join(checkpoint_dir, f'{os.path.basename(checkpoint_dir)}_*.pth'))

    # Check if any checkpoint files are present
    if not checkpoint_files:
        print(f"No checkpoints found in the directory: {checkpoint_dir}")
        return None

    best_accuracy = 0.0
    accuracies = []
    epochs = []
    lossess = []
    for checkpoint_file in checkpoint_files:
        checkpoint = torch.load(checkpoint_file, map_location=torch.device(device))
        accuracies.append(checkpoint['accuracy'].cpu())
```

```
        epochs.append(checkpoint['epoch'])
        lossess.append(checkpoint['loss'])
        if best_accuracy < checkpoint['accuracy']:
            best_accuracy = checkpoint['accuracy']
            best_checkpoint = checkpoint

    return best_checkpoint, accuracies, epochs, lossess
```

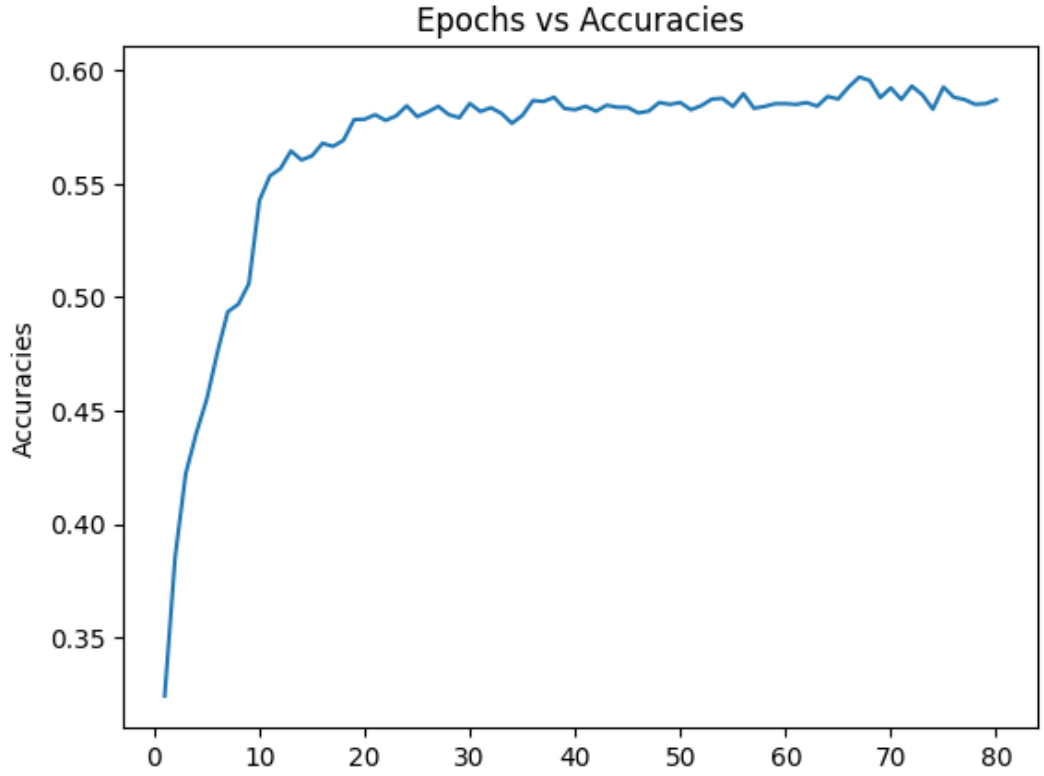
In [10]:

```
checkpoint, accuracies, epochs, losses = load_best_checkpoint(base_checkpoint_path)
```

In [11]:

```
# Draw the plot between epochs and accuracies
print(type(epochs[0]))
import matplotlib.pyplot as plt
plt.plot(epochs, accuracies)
plt.xlabel('Epochs')
plt.ylabel('Accuracies')
plt.title('Epochs vs Accuracies')
plt.show()
```

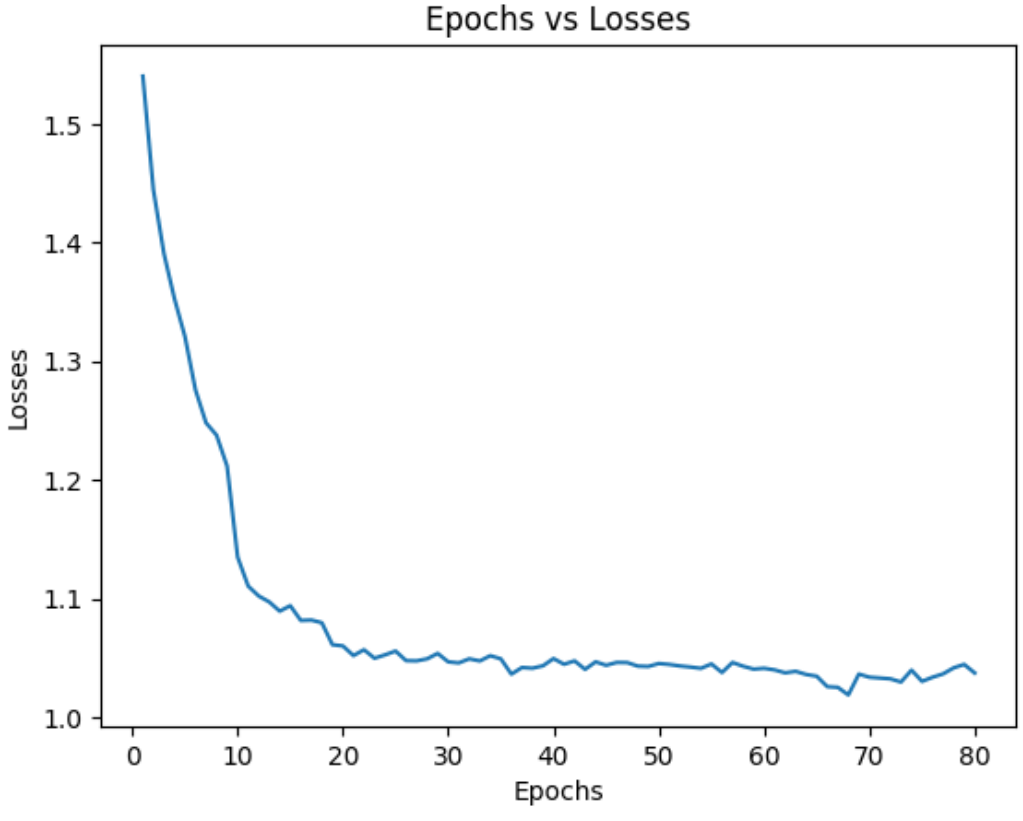
```
<class 'int'>
```



Epochs

In [12]:

```
# Draw the plot between epochs and accuracies
import matplotlib.pyplot as plt
plt.plot(epochs, losses)
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.title('Epochs vs Losses')
plt.show()
```



In [13]:

```
print(checkpoint['accuracy'])
```

```
tensor(0.5972, dtype=torch.float64)
```

In [14]:

```
model_name = config['model']['name']
```

```
if not model_name.startswith('resnet'):
    raise ValueError("Model name must start with 'resnet'")
```

In [15]:

```
if config['model']['type'] == 'FEATURE_EXTRACTOR':
    model = torchvision.models.__dict__[model_name](weights='IMAGENET1K_V1')
    # Freeze all layers except the fully connected layers
    for param in model.parameters():
        param.requires_grad = False
elif config['model']['type'] == 'FINE_TUNING':
    model = torchvision.models.__dict__[model_name](weights='IMAGENET1K_V1')
elif config['model']['type'] == 'TRAIN_FROM_SCRATCH':
    model = torchvision.models.__dict__[model_name](weights=None)
else:
    raise ValueError(f"Unknown model type: {config['model']['type']}")

num_ftrs = model.fc.in_features

model.fc = nn.Linear(num_ftrs, config['model']['num_classes'])

# change the first convolution to accept 6 channels
model.conv1 = nn.Conv2d(6, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

model = model.to(device)
```

In [16]:

```
model.load_state_dict(checkpoint['state_dict'])
```

Out[16]:

```
<All keys matched successfully>
```

In [17]:

```
print(checkpoint['epoch'])
```

```
67
```

In [18]:

```
# List of class directories
class_directories = ['expA', 'expB', 'expC', 'expD', 'expE']
# raw data directory
raw_dir = "raw"
```

In [19]:

```
class CustomDataset(Dataset):
```

```
    def __init__(self, data_dir, raw_data_dir, filename, transform=None):
        super().__init__()
        self.filename = filename
        self.transform = transform

        self.classname = self._extract_class_name(data_dir)
        self.encode = {k: i for i, k in enumerate(class_directories)}


        # Read the train.txt file and store the image paths
        with open(self.filename) as f:
            img_paths= []
            raw_img_paths = []
            for line in f:
                line = line.strip()
                img_paths.append(os.path.join(data_dir, line))
                raw_img_paths.append(os.path.join(raw_data_dir, line))

            self.image_paths = img_paths
            self.raw_image_paths = raw_img_paths

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

    def __getitem__(self, index):
        image_path = self.image_paths[index]
        raw_image_path = self.raw_image_paths[index]
        image = Image.open(image_path)
        raw_image = Image.open(raw_image_path)
        image = np.dstack((np.array(image), np.array(raw_image)))
        label = self.encode[self.classname]
        if self.transform is not None:
            image = self.transform(image)
        return image, label

    def _extract_class_name(self, root_dir):
        # Extract the class name from the root directory
        class_name = os.path.basename(root_dir)
        return class_name
```

In [20]:

```
data_folder = config['paths']['data']
test_file = config['paths']['test']
```

In [21]:

```
def read_dataset(data_folder, txt_file, trasform=None):
    # Create separate datasets for each class
    datasets = []
```

```
    for class_dir in class_directories:
        class_train_dataset = CustomDataset(
            data_dir=os.path.join(data_folder, class_dir),
            raw_data_dir=os.path.join(data_folder, raw_dir),
            filename=os.path.join(txt_file),
            transform=trasform
        )
        datasets.append(class_train_dataset)
    return datasets
```

In [22]:

```
test_tr = transforms.Compose([
        transforms.ToTensor(),
        transforms.CenterCrop(160),
        transforms.Normalize([0.4397, 0.4234, 0.3911, 0.2279, 0.2017, 0.1825], [0.2306, 0.2201, 0.2327, 0.1191, 0.1092, 0.1088])
    ])
```

In [23]:

```
test_dataset = torch.utils.data.ConcatDataset(read_dataset(data_folder, test_file, test_tr))
```

In [24]:

```
bs = 64
```

In [25]:

```
test_dataloader = DataLoader(test_dataset, batch_size=bs*2, shuffle=False)
```

In [26]:

```
def test_accuracy(model, data_loader, device):
    model.eval()
    correct = 0
    total = 0

    nb_classes = len(class_directories)
    confusion_matrix = np.zeros((nb_classes, nb_classes))

    with torch.no_grad():
        for data in data_loader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            for t, p in zip(labels, predicted):
```

```
                confusion_matrix[t.long(), p.long()] += 1

    return round(100 * correct / total, 2), confusion_matrix
```

In [27]:

```
accuracy, confusion_matrix = test_accuracy(model, test_dataloader, device)
```

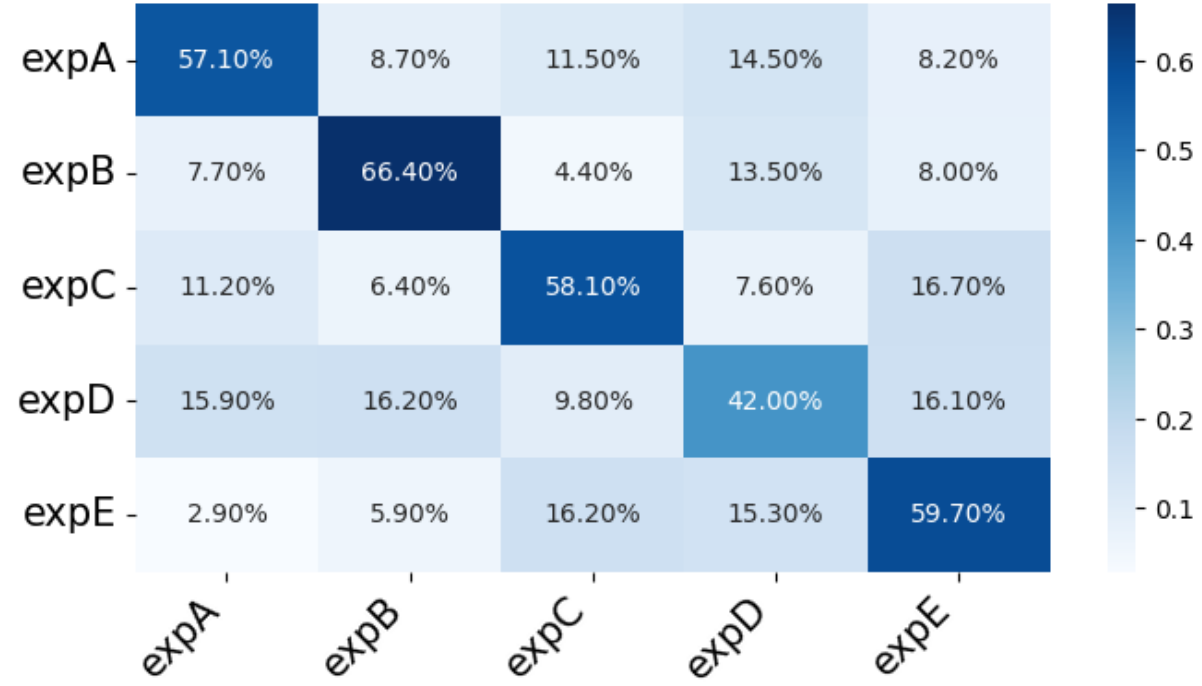
In [28]:

```
print(f"Accuracy: {accuracy}%")
```

```
Accuracy: 56.66%
```

In [29]:

```
plt.figure(figsize=(8,4))

class_names = list(class_directories)
confusion_matrix = confusion_matrix / confusion_matrix.sum(axis=1, keepdims=True)
df_cm = pd.DataFrame(confusion_matrix, index=class_names, columns=class_names).astype(float)
heatmap = sns.heatmap(df_cm, annot=True, fmt='.2%',  cmap='Blues')

heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right',fontsize=15)
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha='right',fontsize=15)
plt.show()
```



In [30]:

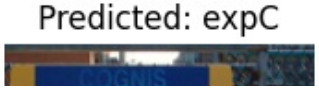
```
def imshow(inp, title=None):
    """Display image for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    org_img = inp[:,:,:3]
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    org_img = std * org_img + mean
    org_img = np.clip(org_img, 0, 1)
    plt.imshow(org_img)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated
```

In [31]:

```
def visualize_model_predictions(model,img_path, raw_img_path):
    was_training = model.training
    model.eval()

    img = Image.open(img_path)
    raw = Image.open(raw_img_path)
    img = np.dstack((np.array(img), np.array(raw)))
    img = test_tr(img)
    img = img.unsqueeze(0)

    img = img.to(device)

    with torch.no_grad():
        outputs = model(img)
        _, preds = torch.max(outputs, 1)

        ax = plt.subplot(2,2,1)
        ax.axis('off')
        ax.set_title(f'Predicted: {class_directories[preds[0]]}')
        imshow(img.cpu().data[0])

        model.train(mode=was_training)
```

In [32]:

```
org_img_path = os.path.join(config['paths']['data'], 'expC', '4100.png')
raw_img_path = os.path.join(config['paths']['data'], 'raw', '4100.png')
visualize_model_predictions(model, img_path=org_img_path, raw_img_path=raw_img_path)
```



In [ ]:

```
import time
time.sleep(5)  # Sleep for 5 seconds to let the system cool down
from google.colab import runtime
runtime.unassign()
```

In [ ]:

```

```