In [2]:
```
# Import necessary packages, download each package as needed if you do not already hav
import torch
import torchvision
import random
import numpy as np
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
import collections
import torch.nn as nn
import torch.nn.functional as F
import os


# Set the seed for reproducibility
SEED = 123456
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

# Data augmentation transformations, used to make the data harder to read by the model
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize image
])

# Path to the data folder, need to download this set to properly run the code
data = '/Users/tugge/Downloads/DATA375dataset'

# Load the dataset
full_dataset = datasets.ImageFolder(root=data, transform=transform)

# Define the split ratios of my data into a training set, a validation set, and a test
train_ratio = 0.7
val_ratio = 0.15
test_ratio = 0.15

# Sizes for each split
train_size = int(train_ratio * len(full_dataset))
val_size = int(val_ratio * len(full_dataset))
test_size = len(full_dataset) - train_size - val_size  # Remaining data goes to test

# Set seed for reproducibility when splitting the dataset, but the split itself is sti
torch.manual_seed(SEED)
train_dataset, val_dataset, test_dataset = random_split(full_dataset, [train_size, val

# Create DataLoader for training, validation, and test data, where the data is always
batch_size = 4
trainloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_worke
valloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True, num_workers=2
testloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True, num_workers
```

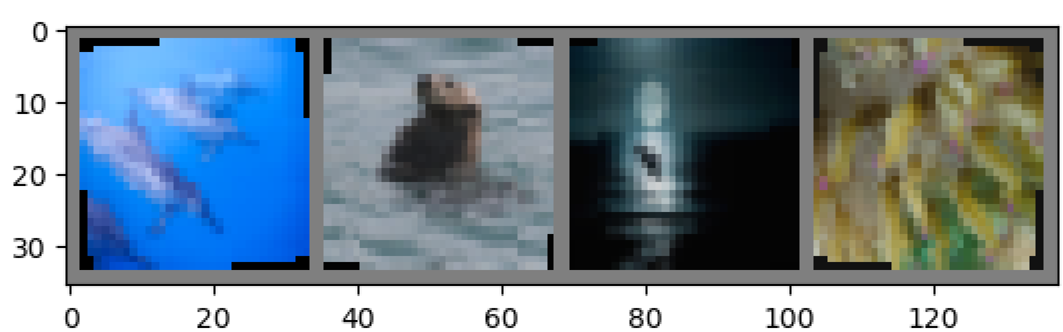
```
# Show the classes in the dataset
classes = full_dataset.classes
print("Classes:", classes)

# Print the size of each input in the dataset, should be a batch of 4 with a image of
for inputs, labels in trainloader:
    print(inputs.size(), labels.size())
    break

# Print the size of each dataset split,should be the same as the ratios I set before
print(f"Train size: {len(train_dataset)}")
print(f"Validation size: {len(val_dataset)}")
print(f"Test size: {len(test_dataset)}")
```

```
Classes: ['Clams', 'Corals', 'Crabs', 'Dolphin', 'Eel', 'Fish', 'Jelly Fish', 'Lobste
r', 'Nudibranchs', 'Octopus', 'Otter', 'Penguin', 'Puffers', 'Sea Rays', 'Sea Urchin
s', 'Seahorse', 'Seal', 'Sharks', 'Shrimp', 'Squid']
torch.Size([4, 3, 32, 32]) torch.Size([4])
Train size: 7482
Validation size: 1603
Test size: 1604
```

This function was directly taken from the Pytorch tutorial online on image classification

In [5]:
```
# This function just shows one batch and its associated image
def imshow(img):
    """Function to unnormalize and show the image."""
    img = img / 2 + 0.5  # Unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Get a batch of training data
data_iter = iter(trainloader)
images, labels = next(data_iter)

# Loads the names of the classes
label_names = [classes[label] for label in labels]

# Print the labels
print("Class names:", label_names)

# Tada
imshow(torchvision.utils.make_grid(images))
```

```
Class names: ['Dolphin', 'Otter', 'Dolphin', 'Shrimp']
```



In [7]:
```
# This was the first model I tried, inspired by the pytorch tutorial
class First_Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 20)
        self.dropout = nn.Dropout(0.6)  # Add dropout with a 60% probability
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = First_Model()
```

Chatgpt helped write this function below and was used to try to make my model better

In [3]:
```
import torch
import torch.nn as nn
# For this function,
class EnhancedModel(nn.Module):
    def __init__(self):
        super(EnhancedModel, self).__init__()

        # First convolution layer with 32 filters, kernel size 3, padding 1,
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        # Second convolution layer with 64 filters, kernel size 3, padding 1
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        # Third convolution layer with 128 filters, kernel size 3, padding 1
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)

        # Pooling layers to reduce spatial size
        self.pool = nn.MaxPool2d(2, 2)  # Reduces spatial size by a factor of 2 every

        # After 3 pooling layers, the size will be reduced to 4x4 spatial dimension.
        self.fc1 = nn.Linear(128 * 4 * 4, 512) # Is a 1D vector of size 2048, and take
        self.fc2 = nn.Linear(512, 20) # Same as above, but reduces the size to the 20

        # Dropout to prevent overfitting, drops have of the activations from fc1, maki
        self.dropout = nn.Dropout(0.5)

        # ReLU activation, makes all negative values 0's at every step in function
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        # Apply the first convolution and pooling layer
        x = self.pool(self.relu(self.conv1(x)))  # Output: 32x32x32 -> 16x16x32
        # Apply the second convolution and pooling layer
        x = self.pool(self.relu(self.conv2(x)))  # Output: 16x16x64 -> 8x8x64
```

```
        # Apply the third convolution and pooling layer
        x = self.pool(self.relu(self.conv3(x)))  # Output: 8x8x128 -> 4x4x128

        # Flatten the output for fully connected layers
        x = x.view(-1, 128 * 4 * 4)  # Flatten to a 1D vector of size 128*4*4 = 2048

        # Fully connected layers with ReLU activations
        x = self.relu(self.fc1(x))
        x = self.dropout(x)  # Apply dropout for regularization
        x = self.fc2(x)  # Final output layer

        return x
```

In [14]:

```
# Function to save checkpoint for future training of a given model
def save_checkpoint(epoch, model, optimizer, scheduler, best_val_loss, val_accuracy, f
    checkpoint = {
        'epoch': epoch,  # current epoch
        'model_state_dict': model.state_dict(),  # model weights
        'optimizer_state_dict': optimizer.state_dict(),  # optimizer state
        'scheduler_state_dict': scheduler.state_dict(),  # scheduler state
        'val_loss': best_val_loss,  # best validation loss so far
        'val_accuracy': val_accuracy,  # best validation accuracy so far
    }
    torch.save(checkpoint, file_path)

# Check if a checkpoint exists to resume training or start a new model
checkpoint_file = "C:/Users/tugge/Downloads/DATA375 Model Iterations.pth24_good"
best_val_loss = float('inf')  # Initialize
best_val_accuracy = 0.0  # Initialize

# Start new or Load from checkpoint
if os.path.exists(checkpoint_file):
    # Load checkpoint
    checkpoint = torch.load(checkpoint_file, weights_only=True)

    # Restore the model, optimizer, and scheduler states
    net = EnhancedModel() # Get the right model
    net.load_state_dict(checkpoint['model_state_dict'])  # Load parameters

    # Re-initialize optimizer and scheduler
    optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=1e-4)
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])  # Load optimizer st

    scheduler = StepLR(optimizer, step_size=4, gamma=0.7)
    scheduler.load_state_dict(checkpoint['scheduler_state_dict'])  # Loadscheduler sta

    # Restore other parameters
    start_epoch = checkpoint['epoch'] + 1
    best_val_loss = checkpoint['val_loss']  # Best validation loss
    best_val_accuracy = checkpoint['val_accuracy']  # Best validation accuracy
    print(f"Resuming from epoch {start_epoch} with best validation loss: {best_val_los
else:
    # Start training
    net = EnhancedModel()
    optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=1e-4)
    scheduler = StepLR(optimizer, step_size=5, gamma=0.7)
    start_epoch = 0
    print("Starting fresh training.")
```

```
# Initialize patience parameter to stop training if not getting better
patience = 5  # Number of epochs with no improvement, training then stops
epochs_without_improvement = 0  # Counter for tracking lack of improvement

# Initialize learning rate
for param_group in optimizer.param_groups:
    param_group['lr'] = 0.01

scheduler.last_epoch = start_epoch - 1  # Set scheduler to the correct epoch

#Model to training mode
net.train()

# Initialize deque to store the last 5 validation losses
last_5_val_losses = collections.deque([5.0] * 5, maxlen=5)  # Start with 5.0 for the f

num_epochs = 25  # Define the number of epochs you want to run for training

for epoch in range(start_epoch, num_epochs):
    running_loss = 0.0
    net.train()
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()  # Zero the gradients
        outputs = net(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Calculate loss
        loss.backward()  # Backprop
        optimizer.step()  # Update weights
        running_loss += loss.item()

    print(f'Epoch {epoch+1}, Training Loss: {running_loss / len(trainloader):.4f}')

    net.eval()  # Set model to evaluation mode
    val_loss = 0.0 # initialize loss
    correct = 0 # Count for correct
    total = 0 # Count of all
    with torch.no_grad():
        for data in valloader:
            inputs, labels = data
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    val_loss /= len(valloader)
    # Get accuracy
    val_accuracy = correct / total * 100
    print(f'Epoch {epoch+1}, Validation Loss: {val_loss:.4f}, Validation Accuracy: {va

    if val_loss < min(last_5_val_losses): # Get condition for whether do add a loss to
        epochs_without_improvement = 0  # Reset the counter
    else:
        epochs_without_improvement += 1 # Add if no improvement is found

    # Early stopping if validation loss doesn't improve
    if epochs_without_improvement >= patience:
        print(f"Early stopping: No improvement in validation loss for {patience} epoch
        break
```

```
    # Update the learning rate scheduler
    scheduler.step()
```

```
Resuming from epoch 20 with best validation loss: 2.2162795530590333 and accuracy: 2
9.881472239550845%
Epoch 21, Training Loss: 2.5849
Epoch 21, Validation Loss: 2.5936, Validation Accuracy: 21.83%
Epoch 22, Training Loss: 2.5299
Epoch 22, Validation Loss: 2.4762, Validation Accuracy: 22.77%
Epoch 23, Training Loss: 2.5058
Epoch 23, Validation Loss: 2.4494, Validation Accuracy: 26.51%
Epoch 24, Training Loss: 2.5271
Epoch 24, Validation Loss: 2.4447, Validation Accuracy: 24.70%
Epoch 25, Training Loss: 2.5297
Epoch 25, Validation Loss: 2.4618, Validation Accuracy: 25.33%
```

In [12]:

```
checkpoint = torch.load("C:/Users/tugge/Downloads/DATA375 Model Iterations.pth24_good"
net.load_state_dict(checkpoint['model_state_dict'])  # Load model parameters
net.eval()  # Set the model to evaluation mode

# Make predictions on the test set
with torch.no_grad():
    correct = 0
    total = 0
    for data in testloader:
        inputs, labels = data
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Test Accuracy: {accuracy:.2f}%')
```

```
Test Accuracy: 28.74%
```

In [ ]: