CSCI323 Lab 1 Assignment (2025)

Content modified by: Cher Lim (cherl@uow.edu.au)

Name: Jeslyn Ho Ka Yan

7-digit UOW ID:8535383

Instructions:

1. There are three sections (A, B, C) in this notebook.
2. Run the code per section and review the computation results.
3. Answer the questions **supported by your computation results**. Provide **explanation** to your answers to demonstrate your understanding of the relevant concepts.
4. You will print the entire notebook (answers, computation results) into **PDF and submit to Moodle**.

A. Exploring the dataset (2 marks)

Review the t-SNE visualization outputs in Section A.

1. List 4 pairs of classes that are heavily overlapped. (1 mark)
2. The t-SNE visualization gives us a 'starting point' view of our data. What can you implement in a basic CNN model to better separate these overlapping classes? Give 5 suggestions. (1 mark)

Write your answers here:

1. List 4 pairs of classes that are heavily overlapped:

- cat and deer, 1.72
- cat and bird, 1.89
- bird and dog, 1.67
- cat and dog, 2.11

2. 5 suggestions to implement in a basic CNN model to better separate these overlapping classes

- **Data Augmentation** : Apply transformations like rotations, flips, and color jitter to expose the model to more variations.
- **Use Batch Normalization**: Helps stabilize and accelerate training, improving class separation.
- **Add More Convolutional Layers**: Capture more complex patterns to distinguish subtle features.
- **Use Dropout Regularization**: Prevents overfitting, encouraging generalization.
- **Use Class Weights or Focal Loss**: Emphasize hard-to-separate or minority classes during training.

B. Training the model (2 marks)

You are provided with a dataset of N samples. You split it into training and validation set with a ratio of 70:30. You then set the batch_size = k for your training.

1. How many times does the model update its weights? (1 mark)
2. How does batch size affect the model performance? (1 mark)

Write your answers here:

1. How many times does the model update its weights?

I had trained for 2 epochs, and your printouts show updates of:

[1, 2000] loss: ...
...
[2, 12000] loss: ...

12,000 batches × 2 epochs= 24,000 updates

The model updates its weights once per batch. Since the training set was processed in 12,000 batches per epoch, and training ran for 2 epochs, the model updated its weights **24,000 times**.

2. How does batch size affect the model performance? (1 mark)

- A **small batch size** (e.g., 32) makes training slower but often leads to **better generalization**. It introduces more noise in the gradients, which can help the model escape local minima.
- A **large batch size** (e.g., 128 or more) speeds up training by processing more data at once, but it may cause the model to **overfit** or converge to **sharp minima**, resulting in **poorer performance** on unseen data.

Answer: Smaller batch sizes improve generalization but slow down training. Larger batch sizes train faster but may reduce model accuracy on new data.

C. Role of activation function (6 marks)

You will train 3 CNN models:

- A standard CNN model with no activation function at the output layer.
- A non-standard CNN model with ReLU at the output layer.
- A non-standard CNN model with Tanh at the output layer.

Answer the following questions. Your answers must be supported by the **computation results** of your codes and the relevant concepts.

1. For each model above, discuss the impact on the output distribution and model performance. (3 marks)
2. Should an activation function be placed at the output layer prior to SoftMax? Explain your answer. (3 marks)

1. Impact on Output Distribution and Model Performance
Three CNN models were trained:

- **Standard CNN (no activation at output layer):** This model produced raw logits, with both positive and negative values (e.g., from around -1.29 to 2.88). It achieved the best performance with the lowest final loss of approximately 1.263. The outputs were well-distributed and suitable for CrossEntropyLoss, which expects raw logits.
- **CNN with ReLU at output layer:** This model clipped all negative outputs to zero. As a result, many class scores were limited, and the output distribution became skewed toward positive values. This reduced the model's ability to express uncertainty. The final loss was higher, around 1.351, and the performance was slightly worse than the standard model.
- **CNN with Tanh at output layer:** The Tanh function squeezed all outputs into a narrow range between -1 and 1. This made it difficult for the model to produce strong class distinctions, leading to lower confidence in predictions. The final loss was the highest, around 1.597, and the model performed the worst.

2. Should an activation function be placed at the output layer before SoftMax?

No, an activation function like ReLU or Tanh should not be placed at the output layer before SoftMax, especially when using CrossEntropyLoss in PyTorch. This is because CrossEntropyLoss already applies LogSoftmax internally and expects raw, unactivated logits as input.

Adding an activation before this can distort the logits:

- **ReLU** removes all negative values, limiting the model's ability to represent certain classes.
- **Tanh** compresses the values, reducing the model's confidence and affecting class separation.

As shown in the training results, the standard model without any output activation had the best performance. Therefore, using no activation before SoftMax is the correct approach for classification tasks using CrossEntropyLoss.

Learning objectives:

1. Extracts insights from a database.
2. Translate insights into a game plan.
3. Understanding the architecture of a basic CNN.
4. Interpret the training process of a neural network model.
5. Investigate the impact of activation functions at the OUTPUT layer of a CNN model.

A. Exploring the dataset

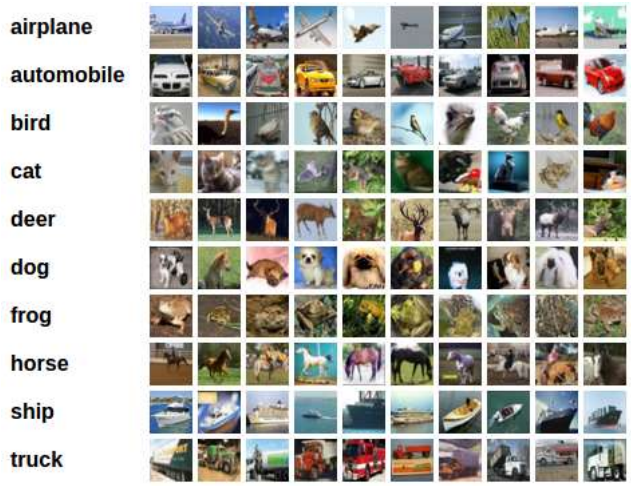
Generally, when you have to deal with image, text, audio or video data, you can use standard python packages that load data into a numpy array. Then you can convert this array into a `torch.*Tensor`.

- For images, packages such as Pillow, OpenCV are useful
- For audio, packages such as scipy and librosa
- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful

Specifically for vision, we have created a package called `torchvision`, that has data loaders for common datasets such as ImageNet, CIFAR10, MNIST, etc. and data transformers for images, viz., `torchvision.datasets` and `torch.utils.data.DataLoader`.

This provides a huge convenience and avoids writing boilerplate code.

For this tutorial, we will use the CIFAR10 dataset. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.



A-1. Load and normalize CIFAR10

Using `torchvision`, it's extremely easy to load CIFAR10.

```
import torch
import torchvision
import torchvision.transforms as transforms
```

Normalize dataset images during loading

Normalizing dataset images during loading is a *standard practice in deep learning*. **It converts original pixel values range from [0,1] to [-1,1].**

Benefits include:

- Centers the data around zero
- Makes the input distribution more symmetrical.
- Makes the data more compatible with activation functions.

To normalise the image during loading:

1. ToTensor(): Converts images from PIL format (0-255 range) to torch tensors with values in [0, 1]
2. Normalize(): Applies the formula: **(x - mean) / std** to each channel

Given:

- mean = (0.5, 0.5, 0.5)
- std = (0.5, 0.5, 0.5)

Using the formula in (2), normalized_x = (x - 0.5) / 0.5.

Hence, **normalized_x = 2x - 1**

Original pixel values range [0, 1] now becomes [-1, 1].

To display the images using Matplotlib, we need to **unnormalize** the image. That means we are solving for x.

x = (normalized_x + 1) / 2

Hence, x = (normalized_x / 2) + 0.5 (formula used in the code).

Let's test: normalized_x range [-1,1], x range [0,1]. So that's correct!

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Each batch has 4 samples. Hence, 12,500 (=50k/4) batches per epoch.
# Model updates its weights once per batch, so 12,500 updates per epoch.
batch_size = 4


trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

A-2. Check out the dataset

```
from collections import Counter
```

```
# 1. Dataset size
print(f"Training set size: {len(trainset)} images")
print(f"Test set size: {len(testset)} images")
print(f"Total dataset size: {len(trainset) + len(testset)} images")

# 2. Image dimensions
sample_image, _ = trainset[0]
print(f"Image dimensions: {sample_image.shape}")
print(f"Image type: {sample_image.dtype}")

# 3. Class distribution
train_labels = [label for _, label in trainset]
label_counts = Counter(train_labels)

print("\nClass distribution in training set:")
for class_idx, count in label_counts.items():
    print(f"Class {classes[class_idx]}: {count} images")
```

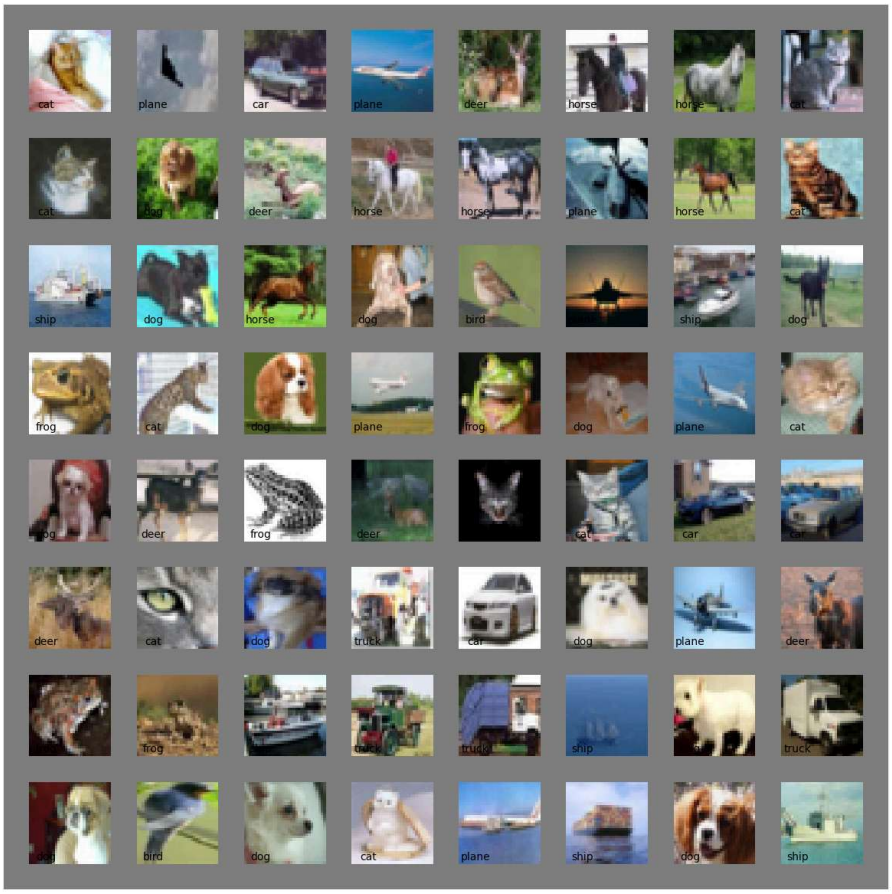
```
Training set size: 50000 images
Test set size: 10000 images
Total dataset size: 60000 images
Image dimensions: torch.Size([3, 32, 32])
Image type: torch.float32

Class distribution in training set:
Class frog: 5000 images
Class truck: 5000 images
```

```
Class deer: 5000 images
Class car: 5000 images
Class bird: 5000 images
Class horse: 5000 images
Class ship: 5000 images
Class cat: 5000 images
Class dog: 5000 images
Class plane: 5000 images
```

Let us show some of the training images, for fun.

```
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image
def show_image_grid(images, labels, nrow=8):  # Changed nrow to 5 for a 5x5 grid
    img_grid = torchvision.utils.make_grid(images, nrow=nrow, padding=10)
    img_grid = img_grid / 2 + 0.5  # unnormalize
    npimg = img_grid.numpy()

    plt.figure(figsize=(15, 15))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')

    # Add class labels as text
    for i, label in enumerate(labels[:len(images)]):
        row = i // nrow
        col = i % nrow
        plt.text(col * (32 + 10) + 16, row * (32 + 10) + 40,
                 classes[label], ha='center')

    plt.show()

# Usage with 25 images
images_list = []
labels_list = []
for i in range(64):  # Get 25 images
    if i % 4 == 0:
        dataiter = iter(trainloader)
    img, lbl = next(dataiter)
    images_list.append(img[0])
    labels_list.append(lbl[0])

show_image_grid(images_list, labels_list)
```



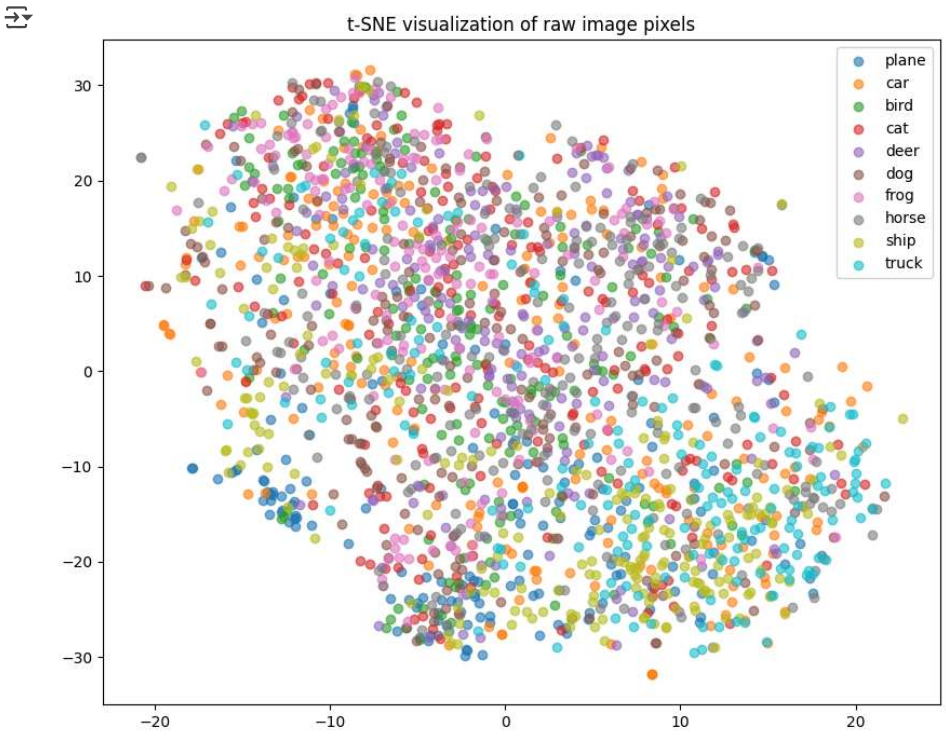
A-3. Run t-SNE on raw pixel values

When you run t-SNE on raw pixel values, each image in CIFAR-10 starts as a high-dimensional vector (**3,072 dimensions for 32×32×3 RGB images**).

The t-SNE algorithm then finds a way to project these high-dimensional points onto a 2D plane while trying to preserve the relative distances between points. (Similar to the idea of PCA).

The x and y axes in the resulting plot are simply the two dimensions that t-SNE creates to represent the data.

```
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Get a subset of images (for speed)
subset_size = 2000
subset_indices = np.random.choice(len(trainset), subset_size, replace=False)

# Extract raw images and labels
images = []
labels = []
for idx in subset_indices:
    img, label = trainset[idx]
    # Flatten the image to a vector
    images.append(img.view(-1).numpy())
    labels.append(label)

images = np.array(images)
labels = np.array(labels)

# Apply t-SNE
tsne = TSNE(n_components=2, random_state=42)
images_tsne = tsne.fit_transform(images)

# Plot
plt.figure(figsize=(10, 8))
for i in range(10):
    indices = labels == i
    plt.scatter(images_tsne[indices, 0], images_tsne[indices, 1], label=classes[i], alpha=0.6)
plt.legend()
plt.title('t-SNE visualization of raw image pixels')
plt.show()
```



A-4. Calculate the Euclidean distance between each pair of classes

```
from sklearn.metrics import pairwise_distances
import pandas as pd

centroids = []
for i in range(10):
    class_points = images_tsne[labels == i]
    centroid = class_points.mean(axis=0)
    centroids.append(centroid)

# Calculate pairwise distances
dist_matrix = pairwise_distances(centroids)

# Create labeled DataFrame
dist_df = pd.DataFrame(dist_matrix, index=classes, columns=classes)

# Print the matrix (rounded for clarity)
print("\nPairwise class centroid distances in t-SNE space (rounded):\n")
print(dist_df.to_string(formatters={col: '{:.2f}'.format for col in dist_df.columns}))
```

```
Pairwise class centroid distances in t-SNE space (rounded):

      plane   car  bird   cat  deer   dog  frog horse  ship truck
plane  0.00 11.82 15.41 15.73 16.45 13.90 20.47 13.39  3.27  8.12
car   11.82  0.00  4.26  3.97  4.67  2.59  9.23  2.98 10.76 10.13
bird  15.41  4.26  0.00  1.89  3.60  1.67  5.10  5.68 14.79 14.37
cat   15.73  3.97  1.89  0.00  1.72  2.11  5.50  4.34 14.72 13.68
deer  16.45  4.67  3.60  1.72  0.00  3.60  6.15  3.83 15.11 13.51
dog   13.90  2.59  1.67  2.11  3.60  0.00  6.73  4.42 13.16 12.72
frog  20.47  9.23  5.10  5.50  6.15  6.73  0.00  9.76 19.89 19.18
horse 13.39  2.98  5.68  4.34  3.83  4.42  9.76  0.00 11.65  9.69
ship   3.27 10.76 14.79 14.72 15.11 13.16 19.89 11.65  0.00  4.87
truck  8.12 10.13 14.37 13.68 13.51 12.72 19.18  9.69  4.87  0.00
```

B. Define a Convolutional Neural Network

1. The following section consists of **two identical CNN models** except one has ReLU after the final layer.
2. You will train both models and tracks their loss curves Evaluates their performance on the test set
3. You will visualise:
    - Training loss comparison
    - Output value distributions (showing how ReLU truncates negative values)
    - Example outputs for specific classes with and without ReLU
4. You will compare accuracy overall and for each class

```
import torch.nn as nn
import torch.nn.functional as F

# Define standard model without ReLU at final layer
class NetStandard(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, 10)

    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)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)  # No ReLU here
        return x
```

```
# Define model with ReLU at final layer
class NetWithFinalReLU(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, 10)
        #self.leaky_relu = nn.LeakyReLU(negative_slope=0.01)

    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)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))  # ReLU added here
        #x = self.leaky_relu(self.fc3(x))  # LeakyReLU applied here
        #x = torch.tanh(self.fc3(x))  # Tanh applied here
        return x
```

CNN with Tanh at the output layer

```
class NetWithTanh(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, 10)

    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)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = torch.tanh(self.fc3(x))  # ✅ Tanh applied only once here
        return x
```

```
# Training function
def train_model(net, trainloader, epochs=2):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    losses = []
    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % 2000 == 1999:
                avg_loss = running_loss / 2000
                losses.append(avg_loss)
                print(f'[{epoch + 1}, {i + 1:5d}] loss: {avg_loss:.3f}')
                running_loss = 0.0

    print('Finished Training')
    return losses
```

```
# Function to get example outputs
def get_sample_outputs(net, testloader):
    # Get a batch from the testloader
    dataiter = iter(testloader)
    images, labels = next(dataiter)

    # Get outputs
    outputs = net(images)

    return outputs, labels
```

B-1. Training the model

```
# Train both models
print("Training standard model (without final ReLU)...")
net_standard = NetStandard()
losses_standard = train_model(net_standard, trainloader)

print("\nTraining model with final ReLU...")
net_with_relu = NetWithFinalReLU()
losses_with_relu = train_model(net_with_relu, trainloader)

print("Training model with Tanh at final layer...")
net_tanh = NetWithTanh()
losses_tanh = train_model(net_tanh, trainloader)
```

```
Training standard model (without final ReLU)...
[1,  2000] loss: 2.187
[1,  4000] loss: 1.881
[1,  6000] loss: 1.643
[1,  8000] loss: 1.560
[1, 10000] loss: 1.500
[1, 12000] loss: 1.466
[2,  2000] loss: 1.402
[2,  4000] loss: 1.368
[2,  6000] loss: 1.327
[2,  8000] loss: 1.330
[2, 10000] loss: 1.305
[2, 12000] loss: 1.263
Finished Training

Training model with final ReLU...
[1,  2000] loss: 2.301
[1,  4000] loss: 2.258
[1,  6000] loss: 2.178
```

```
[1,  8000] loss: 2.011
[1, 10000] loss: 1.758
[1, 12000] loss: 1.596
[2,  2000] loss: 1.513
[2,  4000] loss: 1.459
[2,  6000] loss: 1.433
[2,  8000] loss: 1.374
[2, 10000] loss: 1.367
[2, 12000] loss: 1.351
Finished Training
Training model with Tanh at final layer...
[1,  2000] loss: 2.187
[1,  4000] loss: 1.945
[1,  6000] loss: 1.840
[1,  8000] loss: 1.784
[1, 10000] loss: 1.731
[1, 12000] loss: 1.704
[2,  2000] loss: 1.667
[2,  4000] loss: 1.661
[2,  6000] loss: 1.625
[2,  8000] loss: 1.638
[2, 10000] loss: 1.614
[2, 12000] loss: 1.597
Finished Training
```

B-2. Save the models

```
PATH_S = './cifar_net_standard.pth'
torch.save(net_standard.state_dict(), PATH_S)
PATH_R = './cifar_net_with_relu.pth'
torch.save(net_with_relu.state_dict(), PATH_R)
```

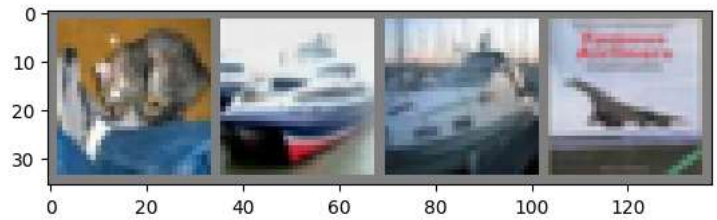
Next, let's load back in our saved model (note: saving and re-loading the model wasn't necessary here, we only did it to illustrate how to do so):

```
#net = net_standard()
net_standard.load_state_dict(torch.load(PATH_S, weights_only=True))
net_with_relu.load_state_dict(torch.load(PATH_R, weights_only=True))
```

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

Okay, now let us see what the neural network thinks these examples above are:

```
dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
```




```
GroundTruth:  cat   ship  ship  plane
```

```
outputs_samples = net_standard(images)
```

```
_, predicted = torch.max(outputs_samples, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))
```

```
Predicted:  dog   car   car   ship
```

Compare Outputs or Plot Losses

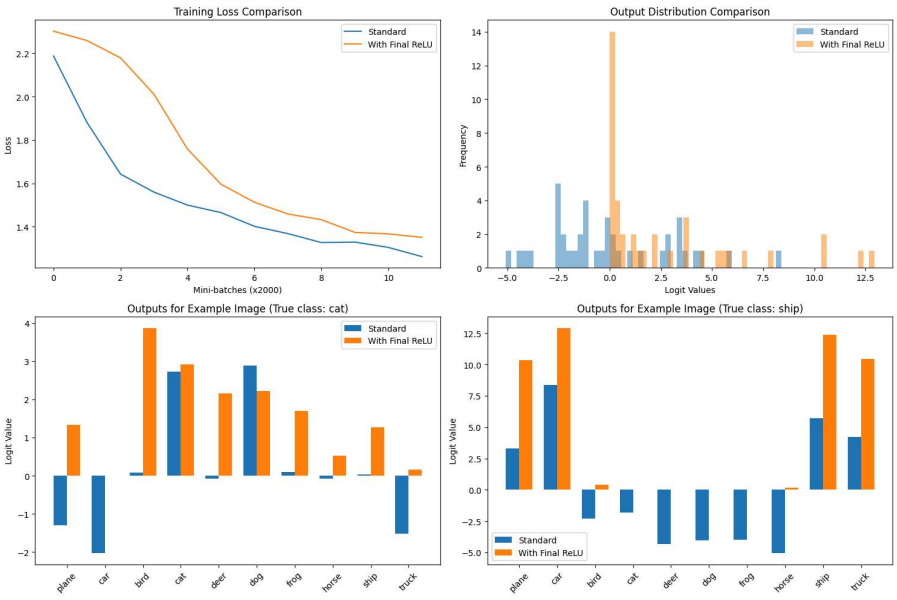
```
out_std, _ = get_sample_outputs(net_standard, testloader)
out_relu, _ = get_sample_outputs(net_with_relu, testloader)
out_tanh, _ = get_sample_outputs(net_tanh, testloader)

print("Standard Output:", out_std[0])
print("ReLU Output:", out_relu[0])
print("Tanh Output:", out_tanh[0])

```

```
Standard Output: tensor([-1.2936, -2.0344,  0.0881,  2.7281, -0.0833,  2.8849,  0.1034, -0.0720,
         0.0342, -1.5195], grad_fn=<SelectBackward0>)
ReLU Output: tensor([1.3330, 0.0000, 3.8738, 2.9142, 2.1602, 2.2187, 1.6903, 0.5272, 1.2618,
        0.1521], grad_fn=<SelectBackward0>)
Tanh Output: tensor([-0.9927, -0.9999, -0.7903,  0.9990, -1.0000,  0.0562,  0.9958, -1.0000,
        -0.8878, -0.9997], grad_fn=<SelectBackward0>)
```

C. Investigate the impact of activation functions at the OUTPUT layer of a CNN model

C-1. Visualize the training loss and Logit values of both models

```
# Get sample outputs
outputs_standard, labels = get_sample_outputs(net_standard, testloader)
outputs_with_relu, _ = get_sample_outputs(net_with_relu, testloader)

# Convert to numpy for plotting
outputs_standard_np = outputs_standard.detach().numpy()
outputs_with_relu_np = outputs_with_relu.detach().numpy()
labels_np = labels.numpy()

# Visualize
plt.figure(figsize=(15, 10))

# Plot loss curves
plt.subplot(2, 2, 1)
plt.plot(losses_standard, label='Standard')
plt.plot(losses_with_relu, label='With Final ReLU')
plt.title('Training Loss Comparison')
plt.xlabel('Mini-batches (x2000)')
plt.ylabel('Loss')
plt.legend()

# Plot output distributions (histogram of logits)
plt.subplot(2, 2, 2)
plt.hist(outputs_standard_np.flatten(), bins=50, alpha=0.5, label='Standard')
plt.hist(outputs_with_relu_np.flatten(), bins=50, alpha=0.5, label='With Final ReLU')
plt.title('Output Distribution Comparison')
plt.xlabel('Logit Values')
plt.ylabel('Frequency')
plt.legend()

# Example comparison for specific images
plt.subplot(2, 2, 3)
example_idx = 0  # Select first image from batch
# Plot the output values for this example
standard_values = outputs_standard_np[example_idx]
relu_values = outputs_with_relu_np[example_idx]
true_label = labels_np[example_idx]

x = np.arange(10)
width = 0.35

plt.bar(x - width/2, standard_values, width, label='Standard')
plt.bar(x + width/2, relu_values, width, label='With Final ReLU')
plt.title(f'Outputs for Example Image (True class: {classes[true_label]})')
plt.xticks(x, classes, rotation=45)
plt.ylabel('Logit Value')
plt.legend()

# Add another example with a different class
plt.subplot(2, 2, 4)
for i in range(1, len(labels_np)):
    if labels_np[i] != labels_np[0]:  # Find example with different class
        example_idx = i
        break
else:
    example_idx = 1  # Fallback if all same class

standard_values = outputs_standard_np[example_idx]
relu_values = outputs_with_relu_np[example_idx]
true_label = labels_np[example_idx]

plt.bar(x - width/2, standard_values, width, label='Standard')
plt.bar(x + width/2, relu_values, width, label='With Final ReLU')
plt.title(f'Outputs for Example Image (True class: {classes[true_label]})')
plt.xticks(x, classes, rotation=45)
plt.ylabel('Logit Value')
plt.legend()

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




```
# Get outputs from the model
outputs = net_with_relu(images)
print("Sample logits (first sample):", outputs[0])
print(f"Min logit: {outputs.min().item():.4f}")
print(f"Max logit: {outputs.max().item():.4f}")

# Convert to NumPy array for analysis
outputs_np = outputs.detach().numpy().flatten()

# Count total and negative logits
total_logits = outputs_np.size
num_negatives = np.sum(outputs_np < 0)

print(f"Total number of logit values: {total_logits}")
print(f"Number of negative logit values: {num_negatives}")
print(f"Percentage of negative values: {100 * num_negatives / total_logits:.2f}%")

```

```
Sample logits (first sample): tensor([1.3330, 0.0000, 3.8738, 2.9142, 2.1602, 2.2187, 1.6903, 0.5272, 1.2618,
        0.1521], grad_fn=<SelectBackward0>)
Min logit: 0.0000
Max logit: 12.9223
Total number of logit values: 40
```

```
Number of negative logit values: 0
Percentage of negative values: 0.00%
```

C-2. Evaluate the models on test data

```python
# Evaluate both models
def evaluate_model(net, testloader):
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = net(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy on test images: {accuracy:.2f}%')
    return accuracy

# Evaluate and print accuracy comparison
print("\nEvaluating standard model (without final ReLU)...")
accuracy_standard = evaluate_model(net_standard, testloader)

print("\nEvaluating model with final ReLU...")
accuracy_with_relu = evaluate_model(net_with_relu, testloader)

print("\nAccuracy Comparison:")
print(f"Standard Model: {accuracy_standard:.2f}%")
print(f"Model with Final ReLU: {accuracy_with_relu:.2f}%")
```

```
Evaluating standard model (without final ReLU)...
Accuracy on test images: 52.83%

Evaluating model with final ReLU...
Accuracy on test images: 51.75%

Accuracy Comparison:
Standard Model: 52.83%
Model with Final ReLU: 51.75%
```

C-3. Accuracy per class in standard CNN model (without ReLU at output layer)

```python
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# checking the CNN without ReLU at output
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net_standard(images) # checking the CNN without ReLU at output
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: plane is 45.5 %
Accuracy for class: car   is 90.2 %
Accuracy for class: bird  is 37.8 %
Accuracy for class: cat   is 33.7 %
Accuracy for class: deer  is 37.9 %
```

```
Accuracy for class: dog   is 61.3 %
Accuracy for class: frog  is 59.3 %
Accuracy for class: horse is 51.4 %
Accuracy for class: ship  is 64.6 %
Accuracy for class: truck is 46.6 %
```

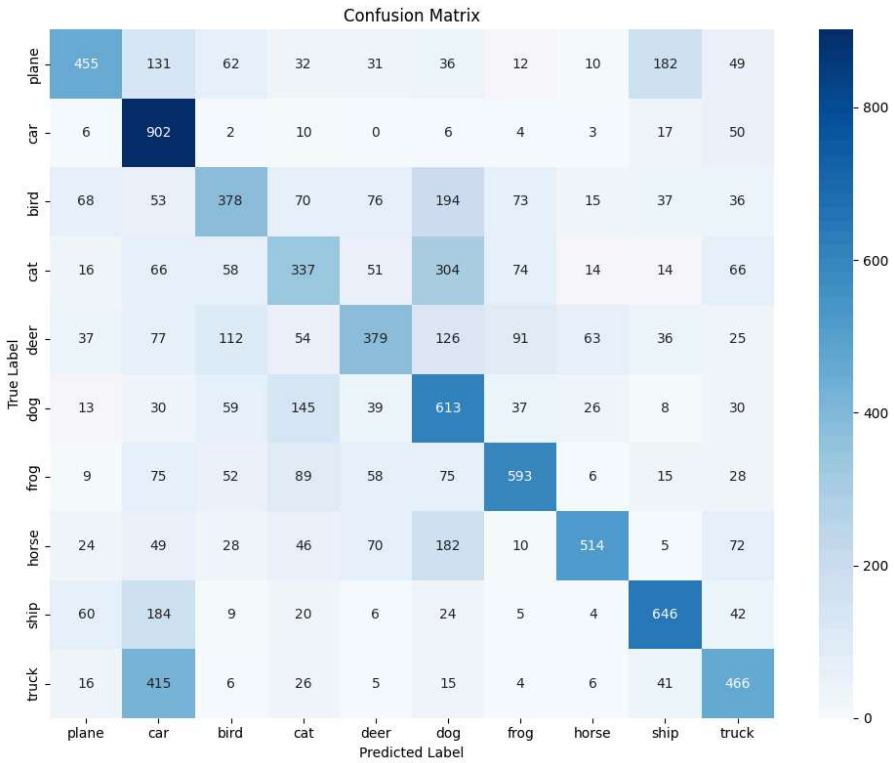
C-4. Accuracy per class in CNN model with ReLU at output layer

```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# checking the CNN without ReLU at output
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net_with_relu(images) # checking the CNN with ReLU at output
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1
# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: plane is 45.6 %
Accuracy for class: car   is 64.7 %
Accuracy for class: bird  is 51.1 %
Accuracy for class: cat   is 22.1 %
Accuracy for class: deer  is 39.5 %
Accuracy for class: dog   is 51.6 %
Accuracy for class: frog  is 41.3 %
Accuracy for class: horse is 62.9 %
Accuracy for class: ship  is 78.1 %
Accuracy for class: truck is 60.6 %
```

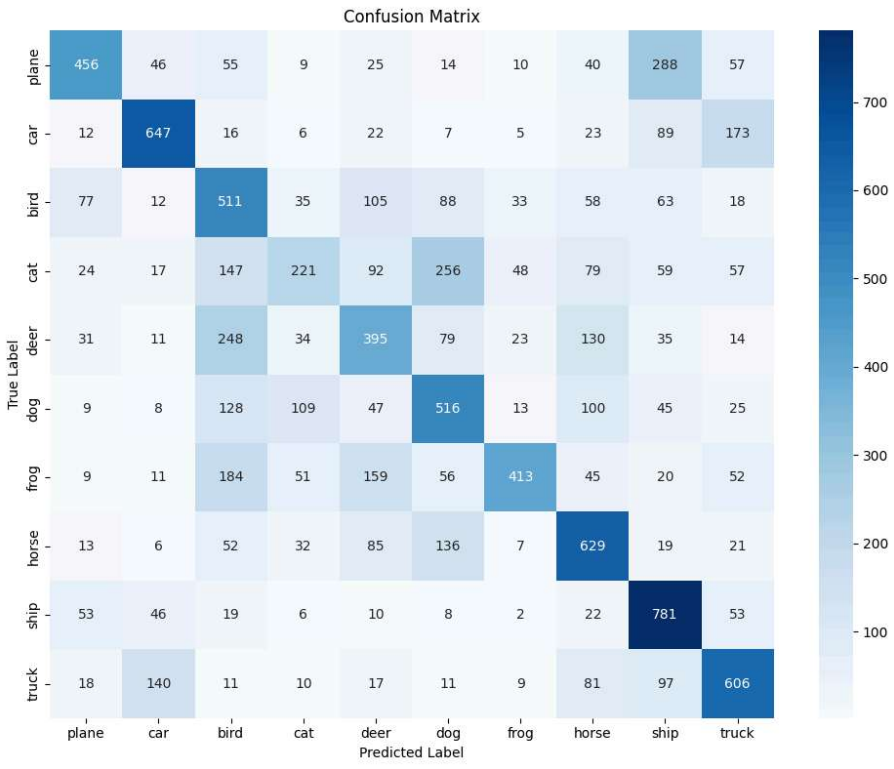
C-5. Confusion matrix for standard CNN model (without ReLU at output layer)

```
from sklearn.metrics import confusion_matrix
import seaborn as sns


# Assuming: net is your trained model, testloader is your DataLoader, and classes is a list of class names
net_standard.eval()

all_preds = []
all_labels = []

with torch.no_grad():
    for inputs, labels in testloader:
        outputs = net_standard(inputs)
        _, predicted = torch.max(outputs, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

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

# Plot as heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()
```



C-6. Confusion matrix for CNN model with ReLU at output layer

```
# Assuming: net is your trained model, testloader is your DataLoader, and classes is a list of class names
net_with_relu.eval()

all_preds = []
all_labels = []

with torch.no_grad():
    for inputs, labels in testloader:
        outputs = net_with_relu(inputs)
        _, predicted = torch.max(outputs, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

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

# Plot as heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()
```



C-7. Saliency maps using GradCam

A saliency map highlights which regions of an image most influenced the model's classification decision. The **warmer colors (red, yellow) indicate areas of high importance**, while cooler colors (blue, green) show less influential regions.

When generating saliency maps, we're **visualizing the gradient flow back** through these activation functions to the input image. Different activation functions create different gradient landscapes.

Study the saliency map for the different models (standard CNN vs non-standard CNN). **What do you observe?**

```
!pip install grad-cam
```

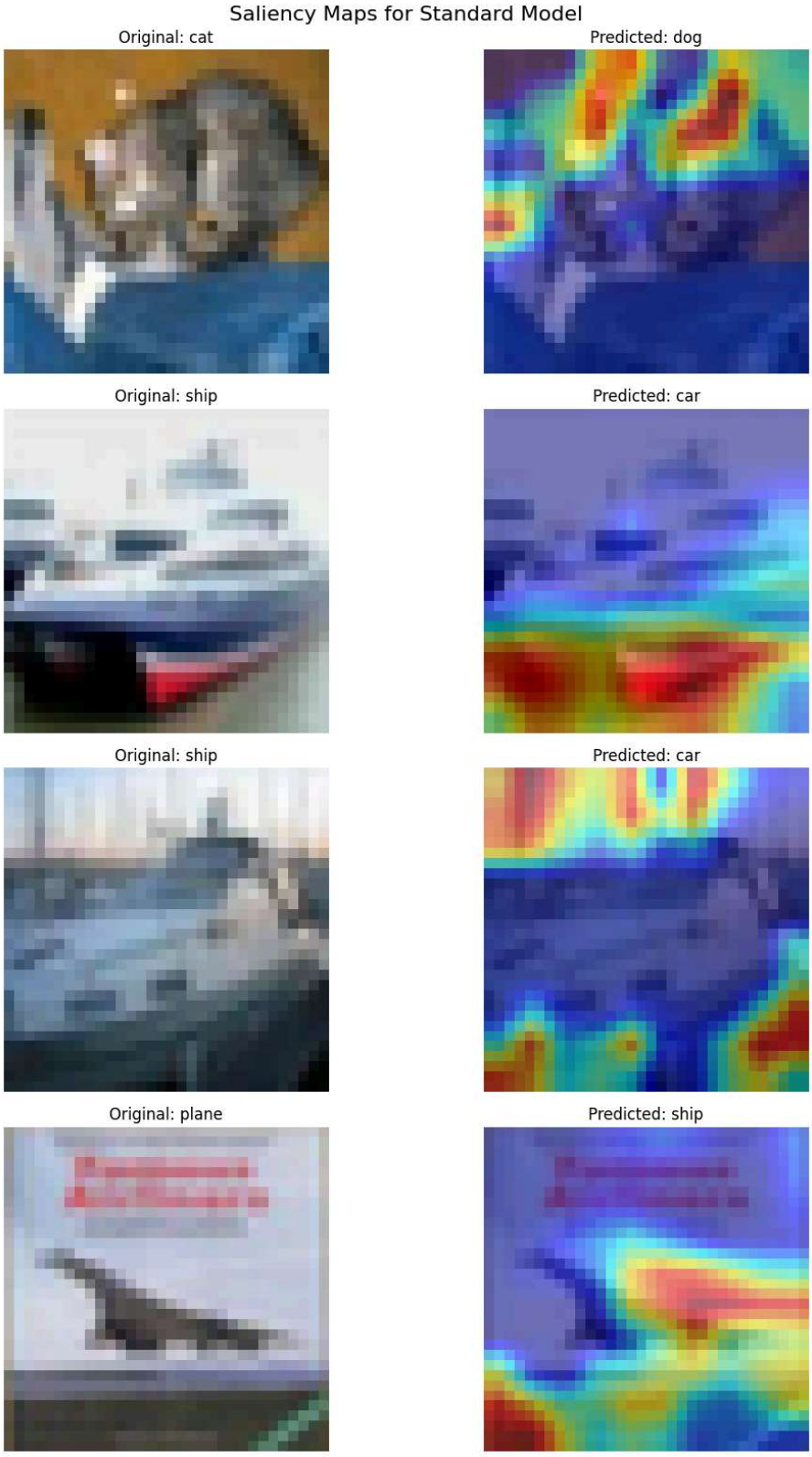
```
Collecting grad-cam
  Downloading grad-cam-1.5.5.tar.gz (7.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 51.5 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from grad-cam) (2.0.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.11/dist-packages (from grad-cam) (11.1.0)
Requirement already satisfied: torch>=1.7.1 in /usr/local/lib/python3.11/dist-packages (from grad-cam) (2.6.0+cu124)
Requirement already satisfied: torchvision>=0.8.2 in /usr/local/lib/python3.11/dist-packages (from grad-cam) (0.21.0+cu124)
Collecting ttach (from grad-cam)
  Downloading ttach-0.0.3-py3-none-any.whl.metadata (5.2 kB)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from grad-cam) (4.67.1)
Requirement already satisfied: opencv-python in /usr/local/lib/python3.11/dist-packages (from grad-cam) (4.11.0.86)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (from grad-cam) (3.10.0)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (from grad-cam) (1.6.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (3.18.0)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (4.
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (2025.3.2)
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (2.2
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (1
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch>=1.7.1->grad-cam)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch>=1.7.1->grad-cam) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch>=1.7.1->grad-
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (4.57.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (24.2)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib->grad-cam) (2.8.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->grad-cam) (1.14.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->grad-cam) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->grad-cam) (3.6.0)
```

```
# Import necessary libraries
from pytorch_grad_cam import GradCAM
from pytorch_grad_cam.utils.image import show_cam_on_image
import matplotlib.pyplot as plt
import numpy as np
import torch

```

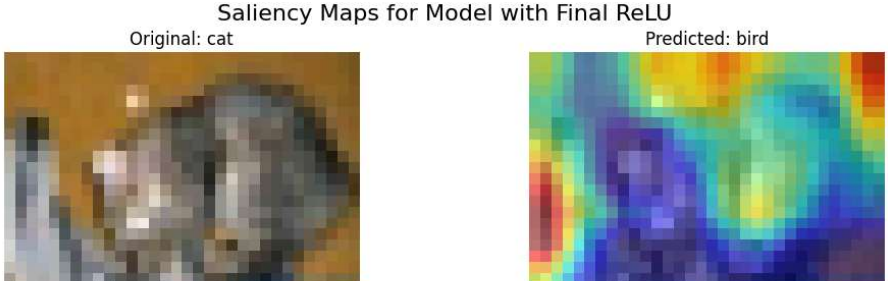
```
def visualize_saliency_maps(net, name):
    """
    Generate and display saliency maps for a given model

    Args:
        net: The trained model
        name: Name of the model for display purposes
    """
    # Select the target layer for visualization (last convolutional layer)
    target_layer = [net.conv2]

    # Initialize GradCAM
    cam = GradCAM(model=net, target_layers=target_layer)

    # Get a batch of test images
    dataiter = iter(testloader)
    images, labels = next(dataiter)

    # Get model predictions
    outputs = net(images)
    _, predicted = torch.max(outputs, 1)

    # Create a figure to display images and their saliency maps
    fig, axes = plt.subplots(4, 2, figsize=(12, 16))
    fig.suptitle(f'Saliency Maps for {name}', fontsize=16)

    for i in range(4):  # Process first 4 images of the batch
        # Original image
        img = images[i].cpu().numpy().transpose(1, 2, 0)
        img = img / 2 + 0.5  # unnormalize

        # Create saliency map
        input_tensor = images[i].unsqueeze(0)  # Add batch dimension
        grayscale_cam = cam(input_tensor=input_tensor, targets=None)
        grayscale_cam = grayscale_cam[0, :]

        # Overlay saliency map on original image
        visualization = show_cam_on_image(img, grayscale_cam, use_rgb=True)

        # Display original image
        axes[i, 0].imshow(img)
        axes[i, 0].set_title(f'Original: {classes[labels[i]]}')
        axes[i, 0].axis('off')

        # Display saliency map
        axes[i, 1].imshow(visualization)
        axes[i, 1].set_title(f'Predicted: {classes[predicted[i]]}')
        axes[i, 1].axis('off')

    plt.tight_layout()
    plt.subplots_adjust(top=0.95)
    plt.show()

```

```
# Generate saliency maps for both models
print("Generating saliency maps for standard model (without final ReLU)...")
visualize_saliency_maps(net_standard, "Standard Model")

print("\nGenerating saliency maps for model with final ReLU...")
visualize_saliency_maps(net_with_relu, "Model with Final ReLU")

```

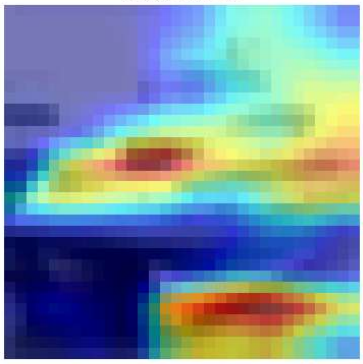
```
Generating saliency maps for standard model (without final ReLU)...
```



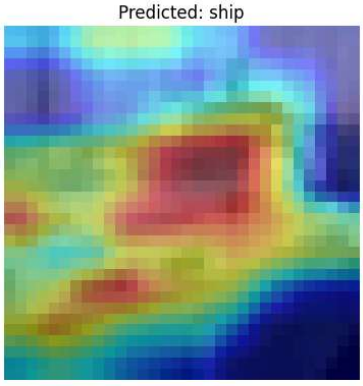

```
Generating saliency maps for model with final ReLU...
```



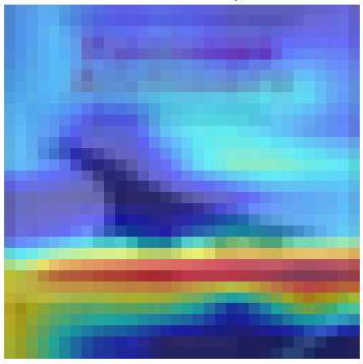
Original: ship Predicted: car



Original: ship Predicted: ship



Original: plane Predicted: ship



```
# Let's also create a direct comparison between the two models
def compare_saliency_maps():
    """
    Generate and compare saliency maps between the two models
    """
    # Get a batch of test images
    dataiter = iter(testloader)
    images, labels = next(dataiter)

    # Create a figure for comparison
    fig, axes = plt.subplots(4, 3, figsize=(15, 16))
    fig.suptitle('Saliency Map Comparison: Standard vs. ReLU at Output', fontsize=16)

    # Target layer for visualization
    target_layer_std = [net_standard.conv2]
    target_layer_relu = [net_with_relu.conv2]

    # Initialize GradCAM for both models
    cam_std = GradCAM(model=net_standard, target_layers=target_layer_std)
    cam_relu = GradCAM(model=net_with_relu, target_layers=target_layer_relu)

    # Get predictions
    outputs_std = net_standard(images)
    outputs_relu = net_with_relu(images)
    _, predicted_std = torch.max(outputs_std, 1)
    _, predicted_relu = torch.max(outputs_relu, 1)

    for i in range(4):  # Process first 4 images
        # Original image
        img = images[i].cpu().numpy().transpose(1, 2, 0)
        img = img / 2 + 0.5  # unnormalize

        # Generate saliency maps for both models
        input_tensor = images[i].unsqueeze(0)

        # Standard model saliency
        grayscale_cam_std = cam_std(input_tensor=input_tensor, targets=None)
```