Data Mining

Image Mining

https://data-mining.github.io/winter-2026/

CS 453/553 – Winter 2026
Yu Wang, Ph.D.
Assistant Professor
Computer Science
University of Oregon

Image Mining - Application

Conventional Image Mining Application

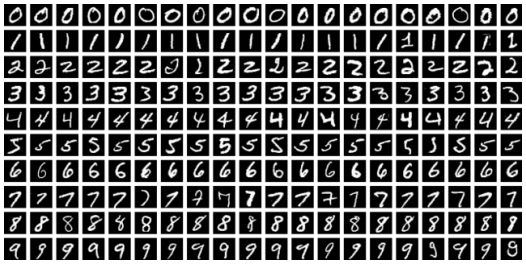


Image Classification

Global Biodiversity

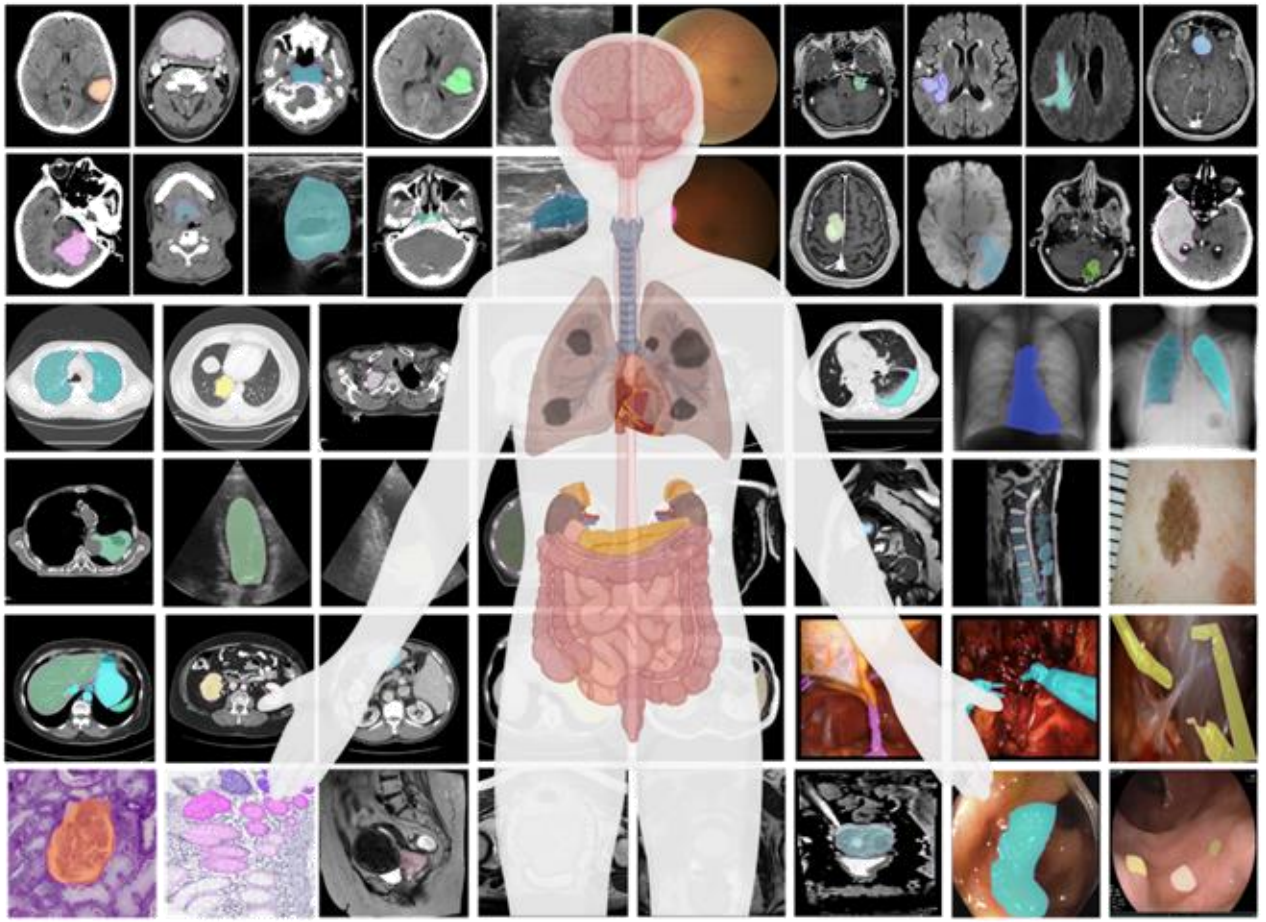


Fig. 1 | MedSAM is trained on a large-scale dataset that can handle diverse segmentation tasks. The dataset covers a variety of anatomical structures, pathological conditions, and medical imaging modalities. The magenta contours and mask overlays denote the expert annotations and MedSAM segmentation results, respectively.

Segment Anything

Image Mining - Application

GenAI Era

Editing Prompt: '*Let the person's hair turn pink*'.

CLIP-S=N/A | CLIP-S=0.091 | CLIP-S=0.103 | CLIP-S=0.118

(a) Source Image | **(b) Edited I** | **(c) Edited II** | **(d) Edited III**

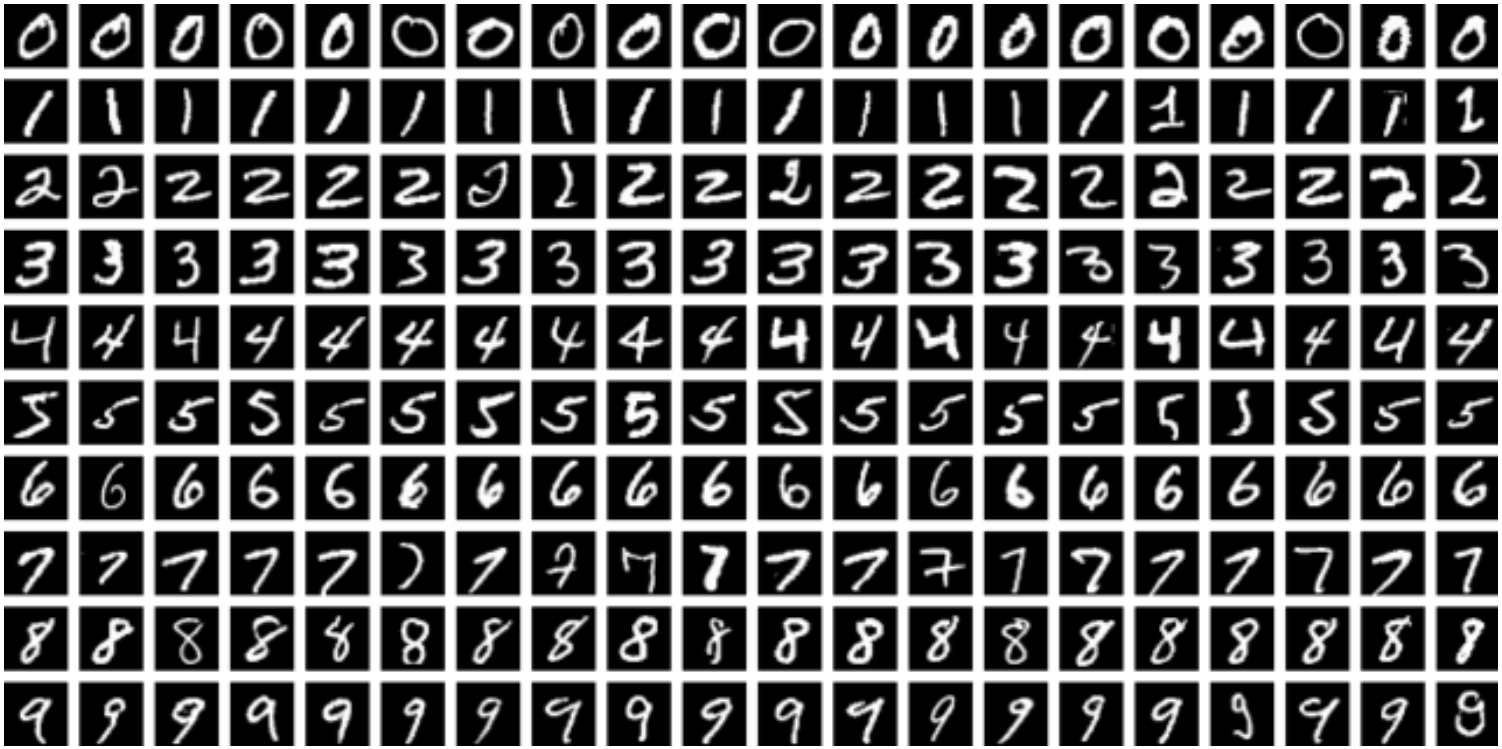
Editing Prompt: '*Let the person wear a police suit*'

Evaluation Metrics:

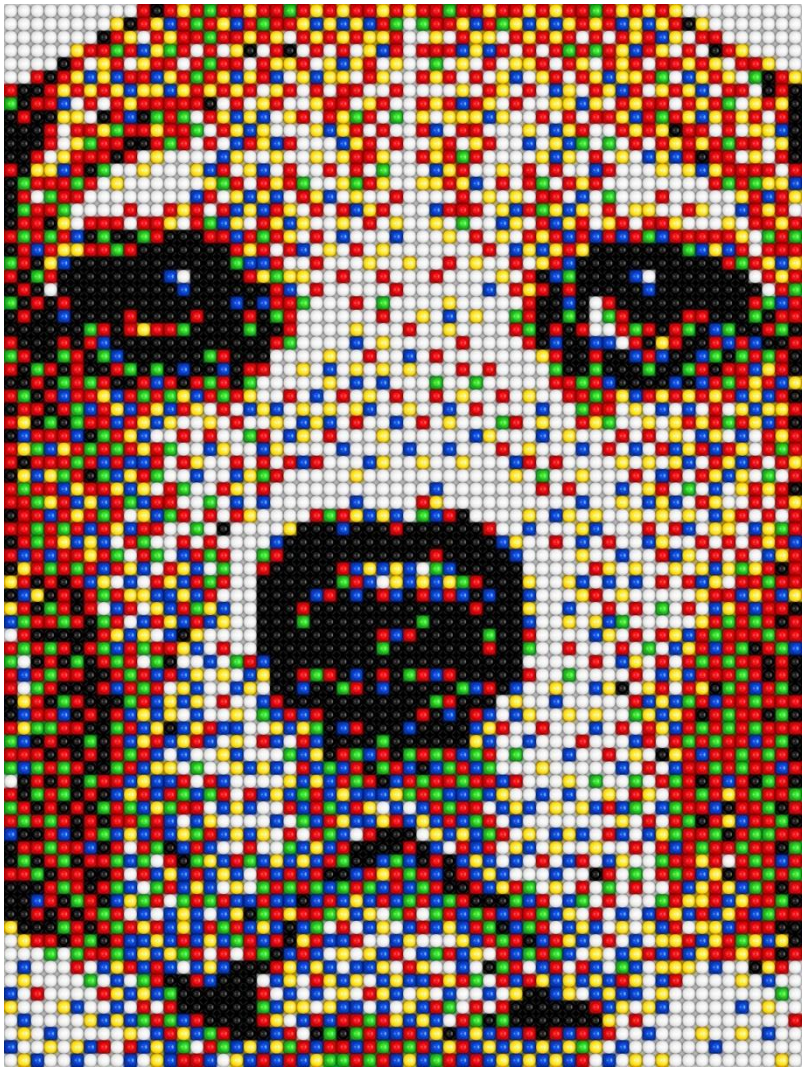
(a) Source Image	(b) Successful Case	(c) Under-Editing	(d) Over-Editing
	✓ Prompt Fidelity ✓ Image Integrity	✗ Prompt Fidelity ✓ Image Integrity	✓ Prompt Fidelity ✗ Image Integrity

Seed Dance

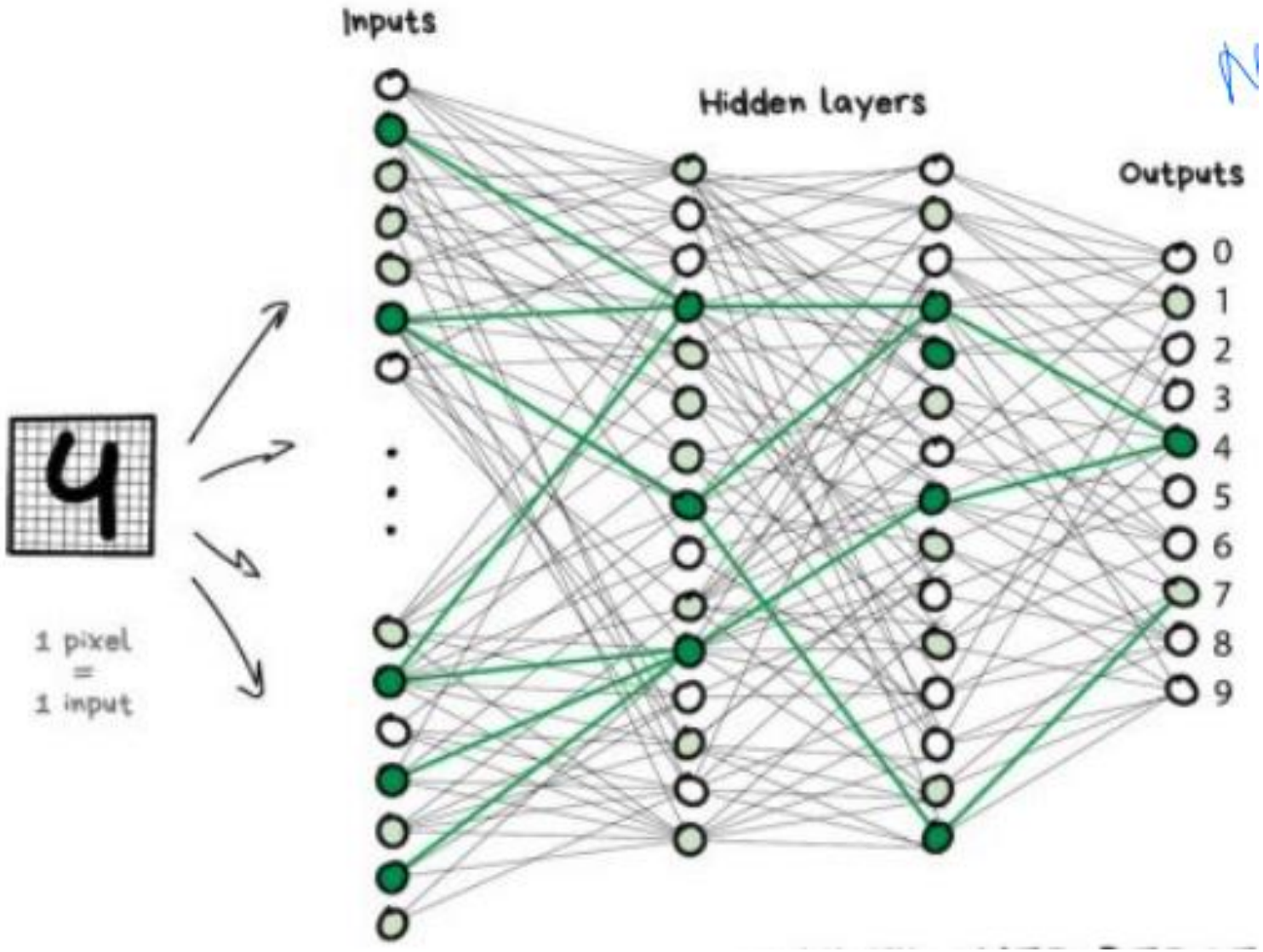
Image



Image



MLP for Image Classification



MLP for Image Classification

Load the MINST dataset

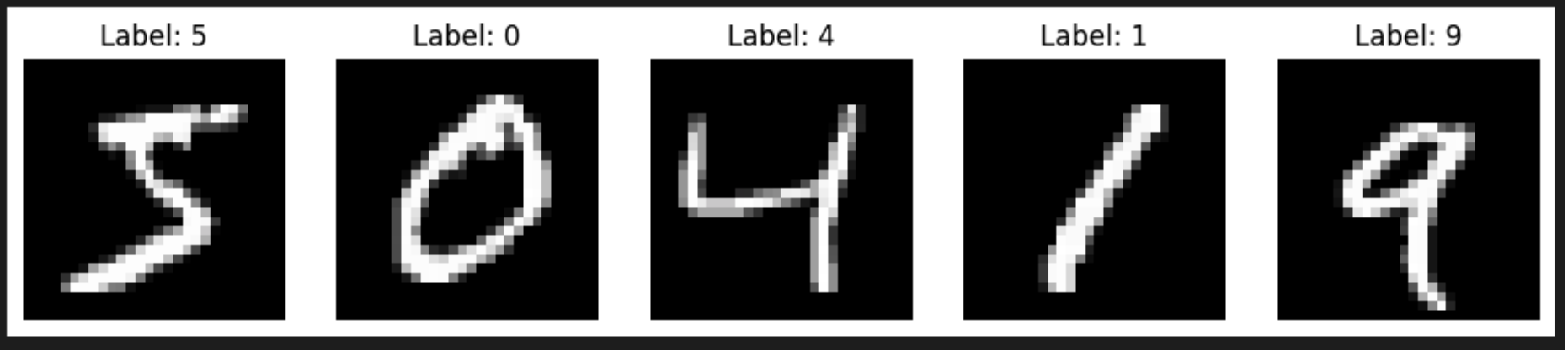
```
import torch
from torchvision import datasets, transforms

# Define a transform to convert PIL images to tensors (and normalize if desired)
transform = transforms.ToTensor()  # This will scale pixel values to [0.0, 1.0]

# Download and load the training and test datasets
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset  = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

print(f"Number of training samples: {len(train_dataset)}")
print(f"Number of test samples: {len(test_dataset)}")
```

32]

```
Number of training samples: 60000
Number of test samples: 10000
```

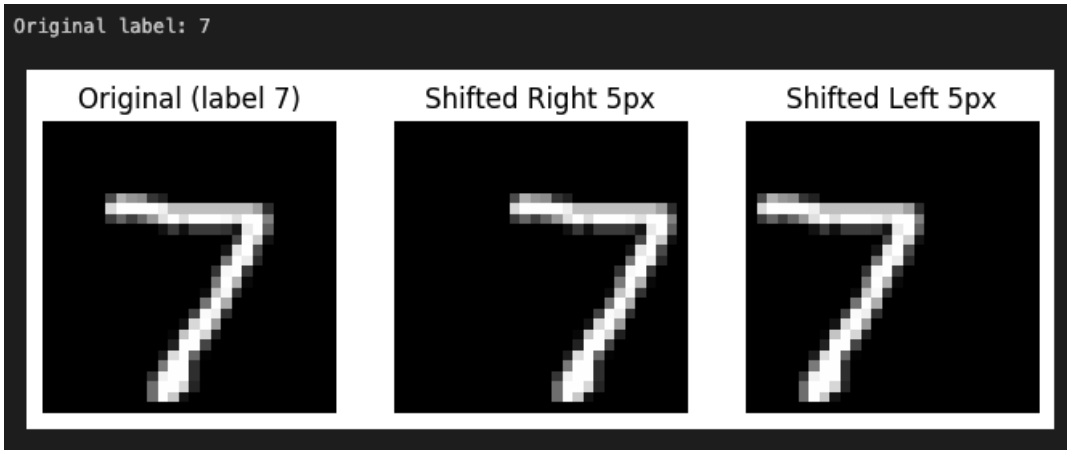
MLP for Image Classification

```python
import matplotlib.pyplot as plt

# Function to show a grid of images
def show_images(dataset, indices):
    """Display images from the dataset given a list of indices."""
    plt.figure(figsize=(12, 3))
    for i, idx in enumerate(indices):
        image, label = dataset[idx]
        image = image.squeeze(0)  # remove the channel dimension (1x28x28 -> 28x28)
        plt.subplot(1, len(indices), i+1)
        plt.imshow(image, cmap='gray')
        plt.title(f"Label: {label}")
        plt.axis('off')
    plt.show()

# Display 5 random training images
sample_indices = [0, 1, 2, 3, 4]  # for example, first 5 images
show_images(train_dataset, sample_indices)
```



MLP for Image Classification

```python
def shift_image(img_tensor, shift):
    """
    Shift a 28x28 image tensor horizontally by 'shift' pixels.
    img_tensor: torch.Tensor of shape (1, 28, 28) or (28, 28) for the image.
    shift: int, positive for right shift, negative for left shift.
    Returns a new torch.Tensor of shape (1, 28, 28) with the shifted image.
    """
    # Ensure img is 2D (28x28) for easier indexing
    if img_tensor.dim() == 3:
        img = img_tensor[0]  # shape (28, 28) if a channel dimension is present
    else:
        img = img_tensor
    img = img.clone()  # clone to avoid modifying original

    # Create an empty image filled with 0s
    shifted_img = torch.zeros_like(img)

    if shift > 0:
        # Shift right: original [:, 0:28-shift] -> new [:, shift:28]
        shifted_img[:, shift:] = img[:, :-shift]
    elif shift < 0:
        # Shift left: original [:, |shift|:28] -> new [:, 0:28-|shift|]
        s = abs(shift)
        shifted_img[:, :-s] = img[:, s:]
    else:
        shifted_img = img  # no shift

    # Add channel dimension back if it was present
    return shifted_img.unsqueeze(0)
```



MLP for Image Classification

```python
# Training loop
epochs = 5
model.train()  # set model to training mode
for epoch in range(1, epochs+1):
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()           # reset gradients
        outputs = model(images)         # forward pass
        loss = criterion(outputs, labels)  # compute loss
        loss.backward()                 # backpropagate
        optimizer.step()                # update weights
        running_loss += loss.item()
    avg_loss = running_loss / len(train_loader)
    print(f"Epoch {epoch}/{epochs}, Training Loss: {avg_loss:.4f}")
```

```
Epoch 1/5, Training Loss: 0.3457
Epoch 2/5, Training Loss: 0.1570
Epoch 3/5, Training Loss: 0.1102
Epoch 4/5, Training Loss: 0.0831
Epoch 5/5, Training Loss: 0.0665
```

Evaluate MLP on Shifted Images

```python
# Function to compute accuracy on the test set for a given horizontal shift
def evaluate_on_shift(shift):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            # apply the shift to each image in the batch
            # images is of shape (batch_size, 1, 28, 28)
            shifted_images = torch.stack([shift_image(img, shift) for img in images])  # apply shift_image to each
            outputs = model(shifted_images)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total

# Evaluate accuracy for shifts from -5 to +5
shifts = list(range(-5, 6))  # [-5, -4, ..., 0, ..., +5]
shift_accuracies = []
for s in shifts:
    acc = evaluate_on_shift(s)
    shift_accuracies.append(acc)
    print(f"Shift {s:+d}: Accuracy = {acc*100:.2f}%")
```

```
Shift -5: Accuracy = 30.02%
Shift -4: Accuracy = 39.81%
Shift -3: Accuracy = 55.80%
Shift -2: Accuracy = 79.28%
Shift -1: Accuracy = 95.01%
Shift +0: Accuracy = 97.41%
Shift +1: Accuracy = 95.78%
Shift +2: Accuracy = 82.15%
Shift +3: Accuracy = 55.46%
Shift +4: Accuracy = 34.11%
Shift +5: Accuracy = 20.22%
```

MLP for Image Classification

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

def optimize_mask(model, image, target_label, num_iters=300, lr=0.1):
    """
    Optimizes a mask matrix to highlight important pixels for a given image.

    - model: Trained MLP model
    - image: Input image (1x28x28)
    - target_label: True label of the image
    - num_iters: Number of optimization iterations
    - lr: Learning rate for optimization
    """
    model.eval()
    image = image.unsqueeze(0)  # Add batch dimension
    image = image.detach()  # Ensure no gradients flow into the original image

    # Initialize learnable mask with small random values
    mask = torch.zeros_like(image, requires_grad=True)

    # Define optimizer (only optimizing the mask)
    optimizer = optim.Adam([mask], lr=lr)

    for i in range(num_iters):
        optimizer.zero_grad()

        # Pass image through sigmoid mask to keep values between [0,1]
        masked_image = image * torch.sigmoid(mask)

        # Forward pass
        output = model(masked_image)
        confidence = output[0, target_label]  # Confidence for the correct class

        # Loss: Maximize confidence (sharpen prediction)
        loss = -confidence
        loss.backward()
        optimizer.step()

        # Optional: Print progress every 50 iterations
        if i % 50 == 0:
            print(f"Iteration {i}, Confidence: {confidence.item():.4f}")

    # Return optimized mask (after sigmoid activation)
    return torch.sigmoid(mask).detach()
```

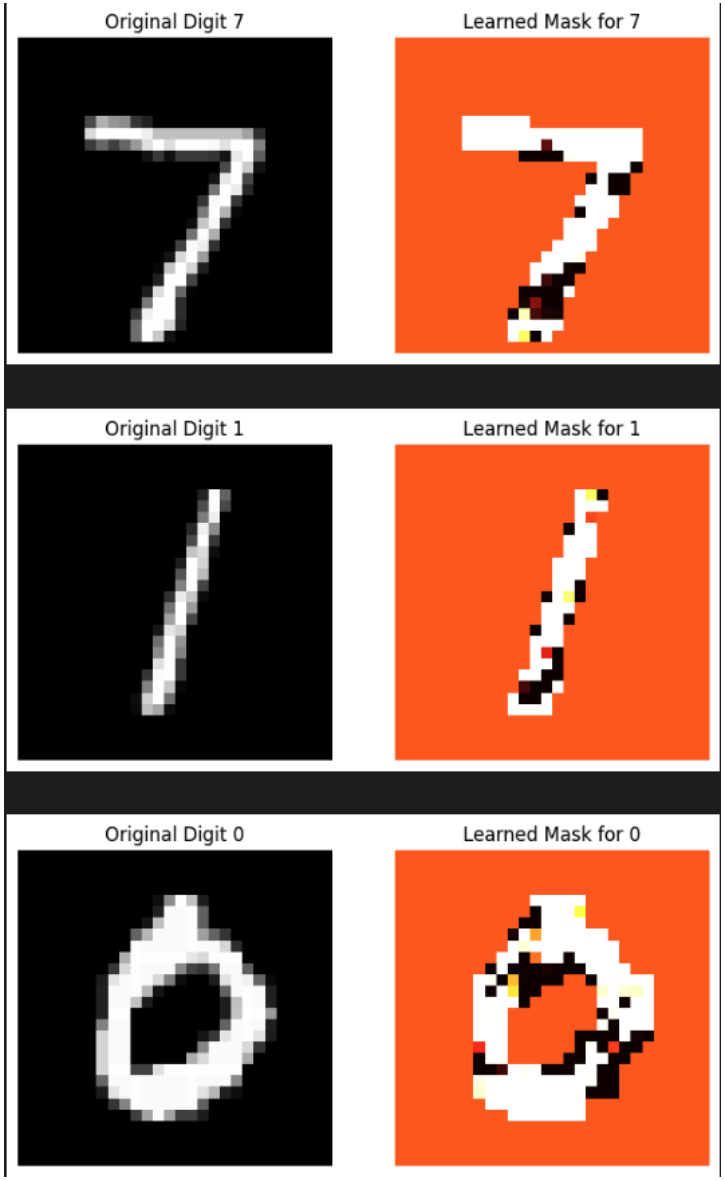
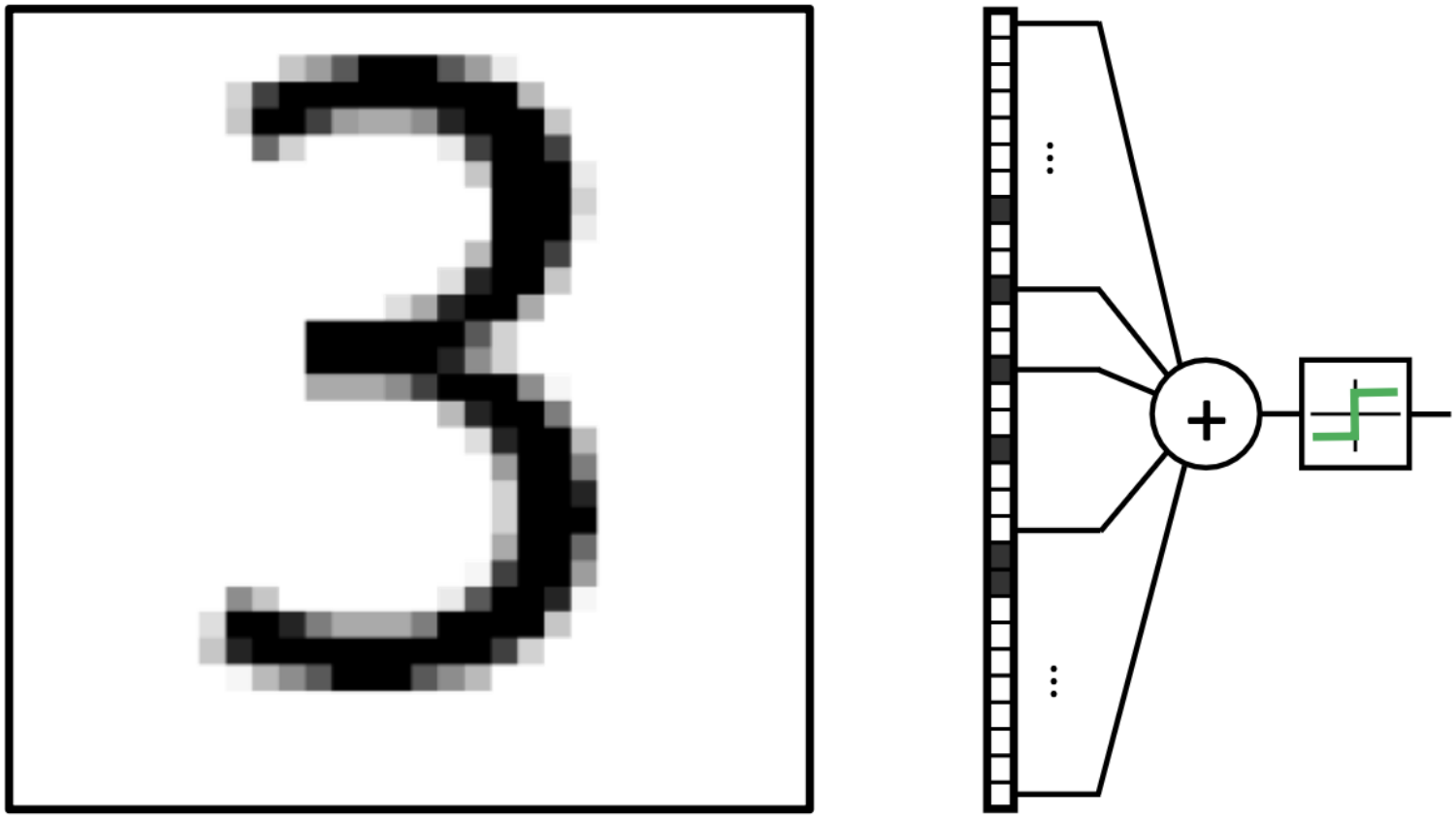


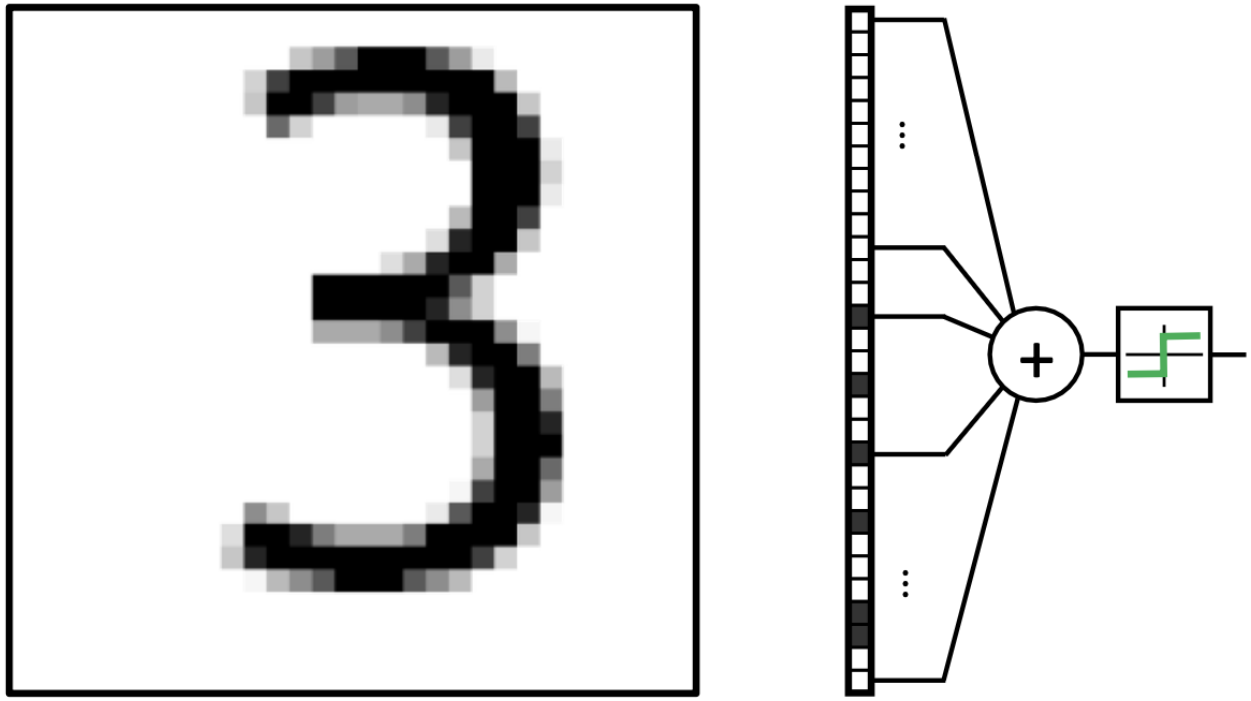
Image Mining



input image

input vector

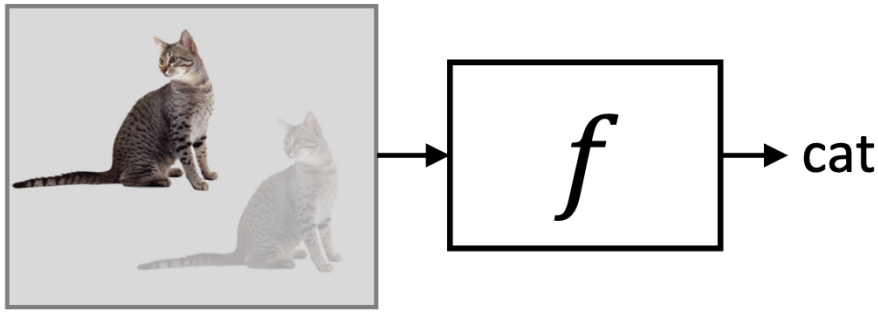
Image Mining



input image input vector

must learn shift invariance from data!

$\mathfrak{G}$-invariance $f(\rho(\mathrm{g})x) = f(x)$



$\mathfrak{G}$-equivariance $f(\rho(\mathrm{g})x) = \rho(\mathrm{g})f(x)$

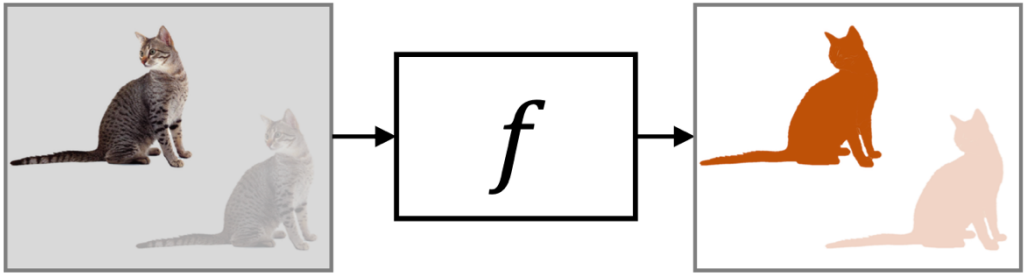
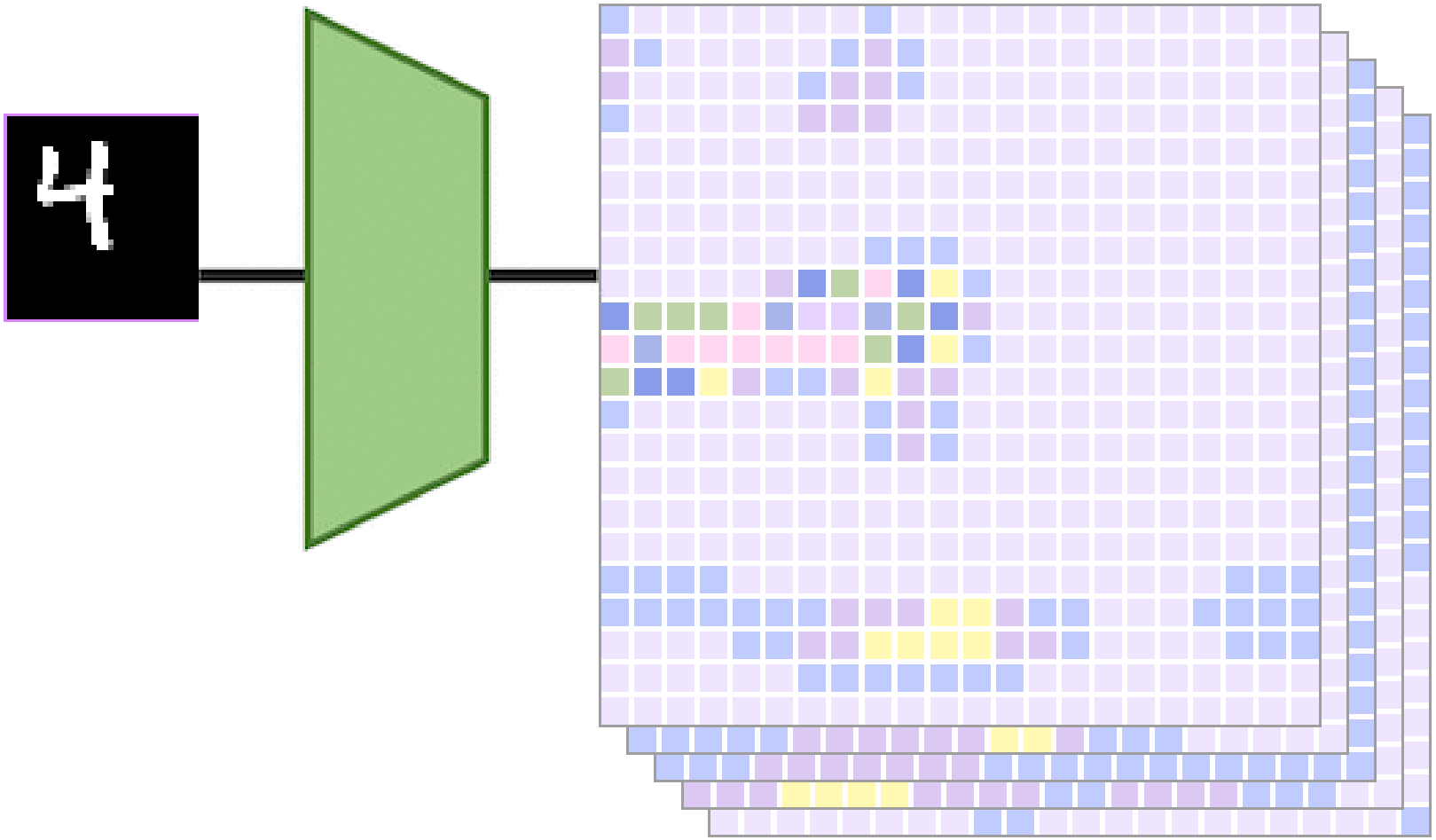
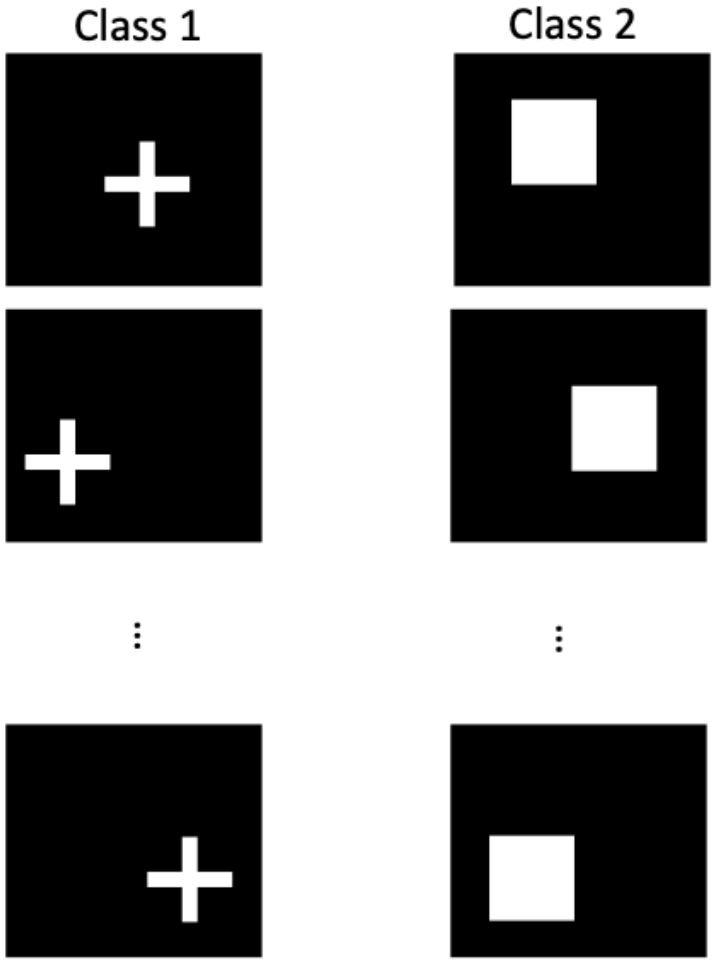


Image Mining

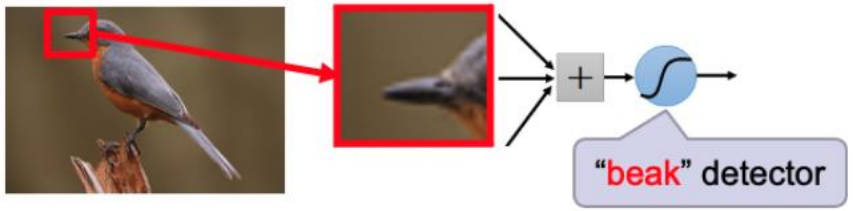


CNN for Image Classification

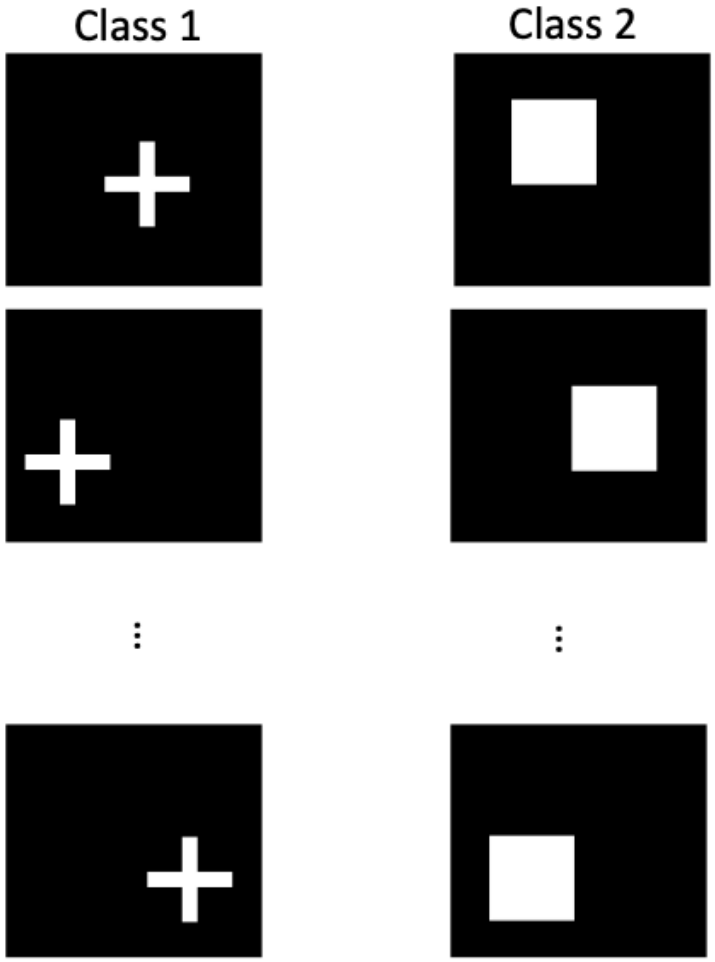


- The features of these classes are spatially local.
- Translation does not change the identity of the classes, i.e., we require a translation equivariant model.
- MLP is not a good match to this problem

Can represent a small region with fewer parameters

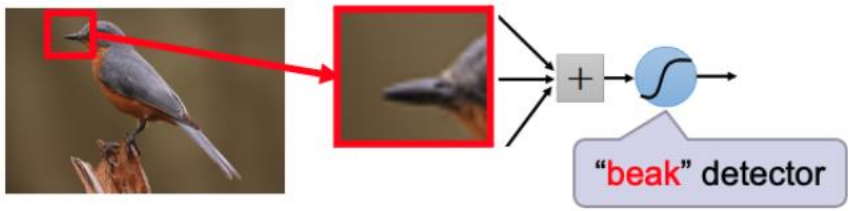


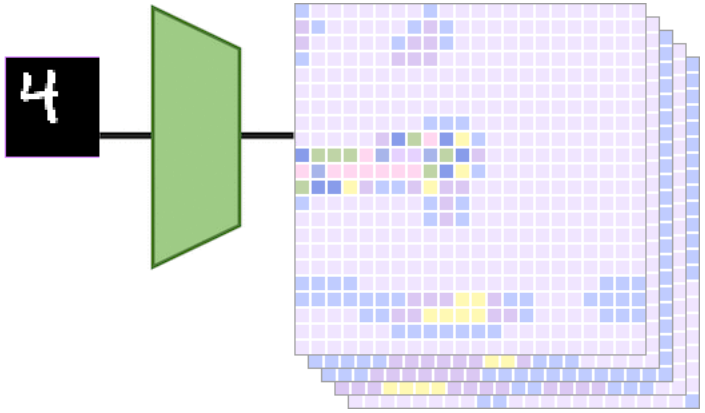
CNN for Image Classification



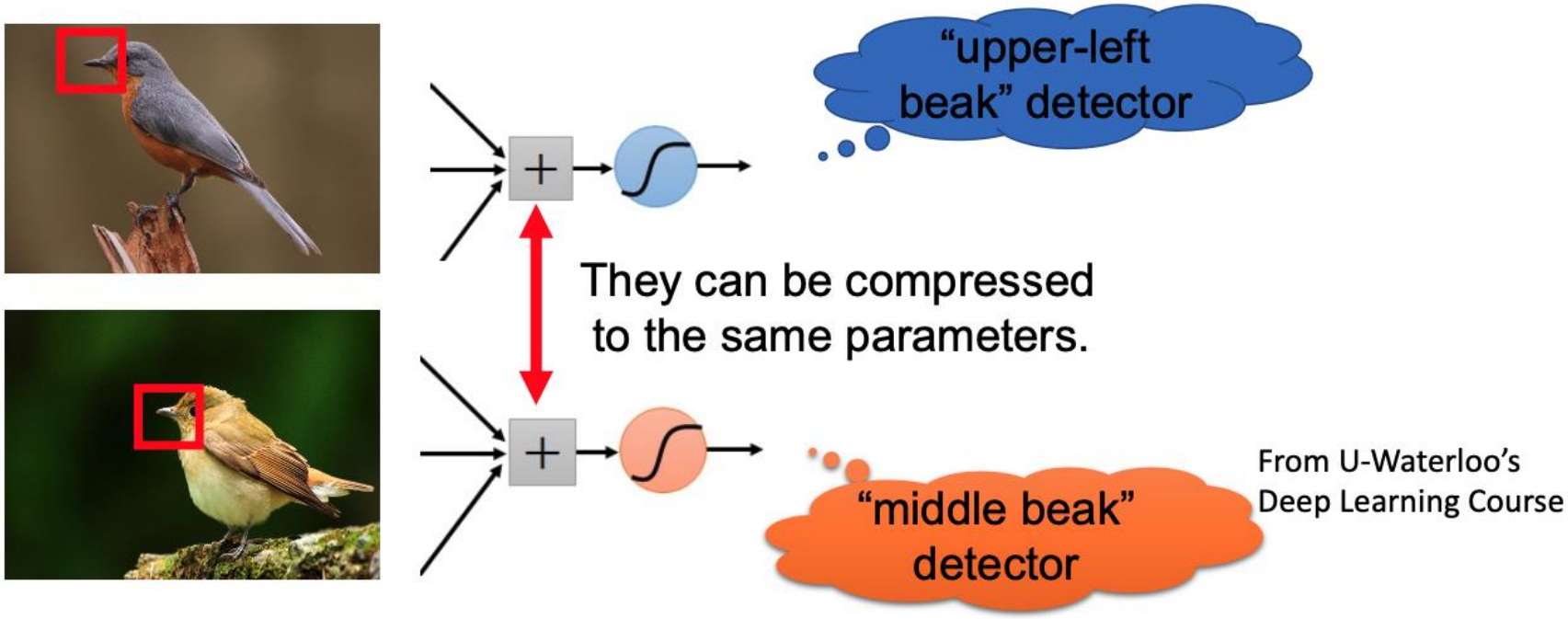
- The features of these classes are spatially local.
- Translation does not change the identity of the classes, i.e., we require a translation equivariant model.
- MLP is not a good match to this problem

Can represent a small region with fewer parameters



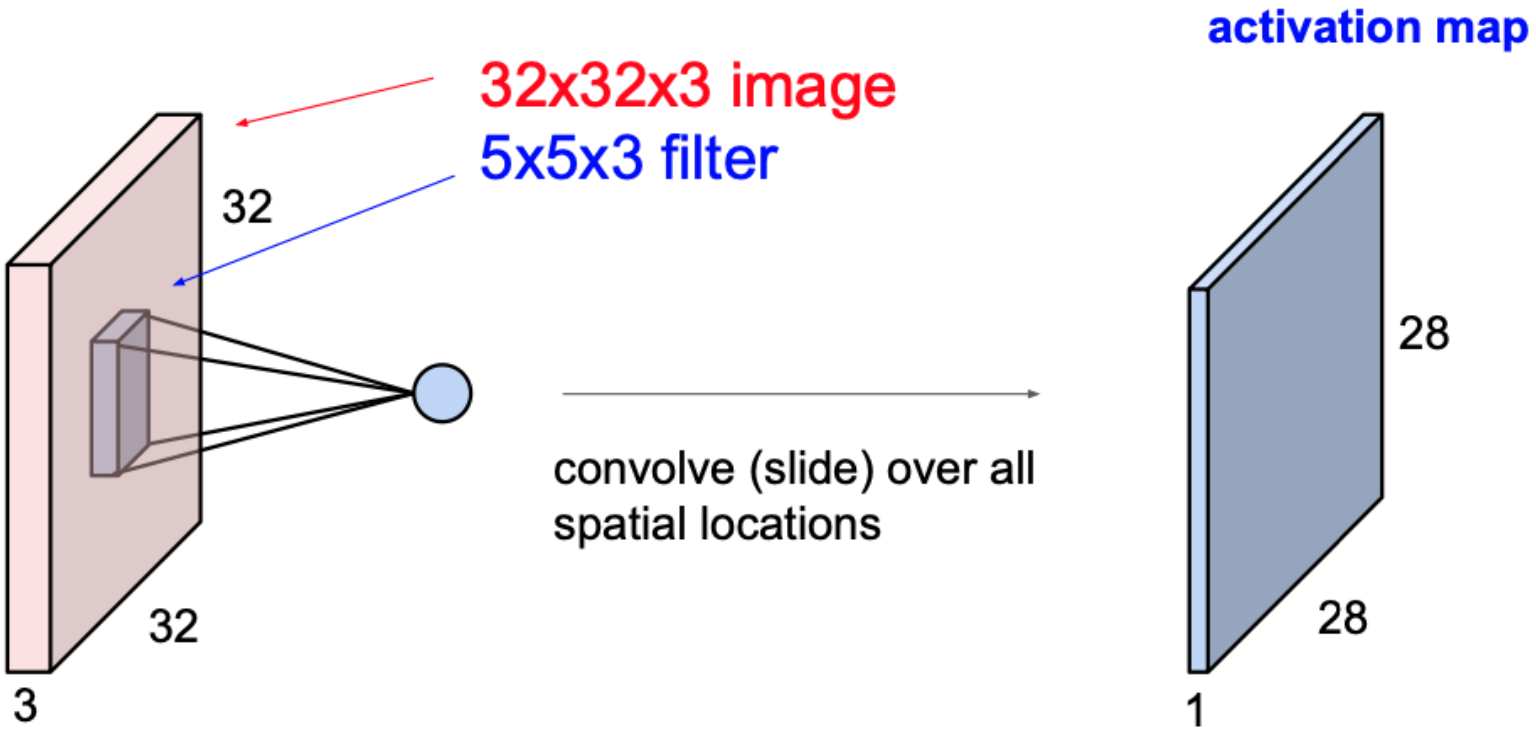


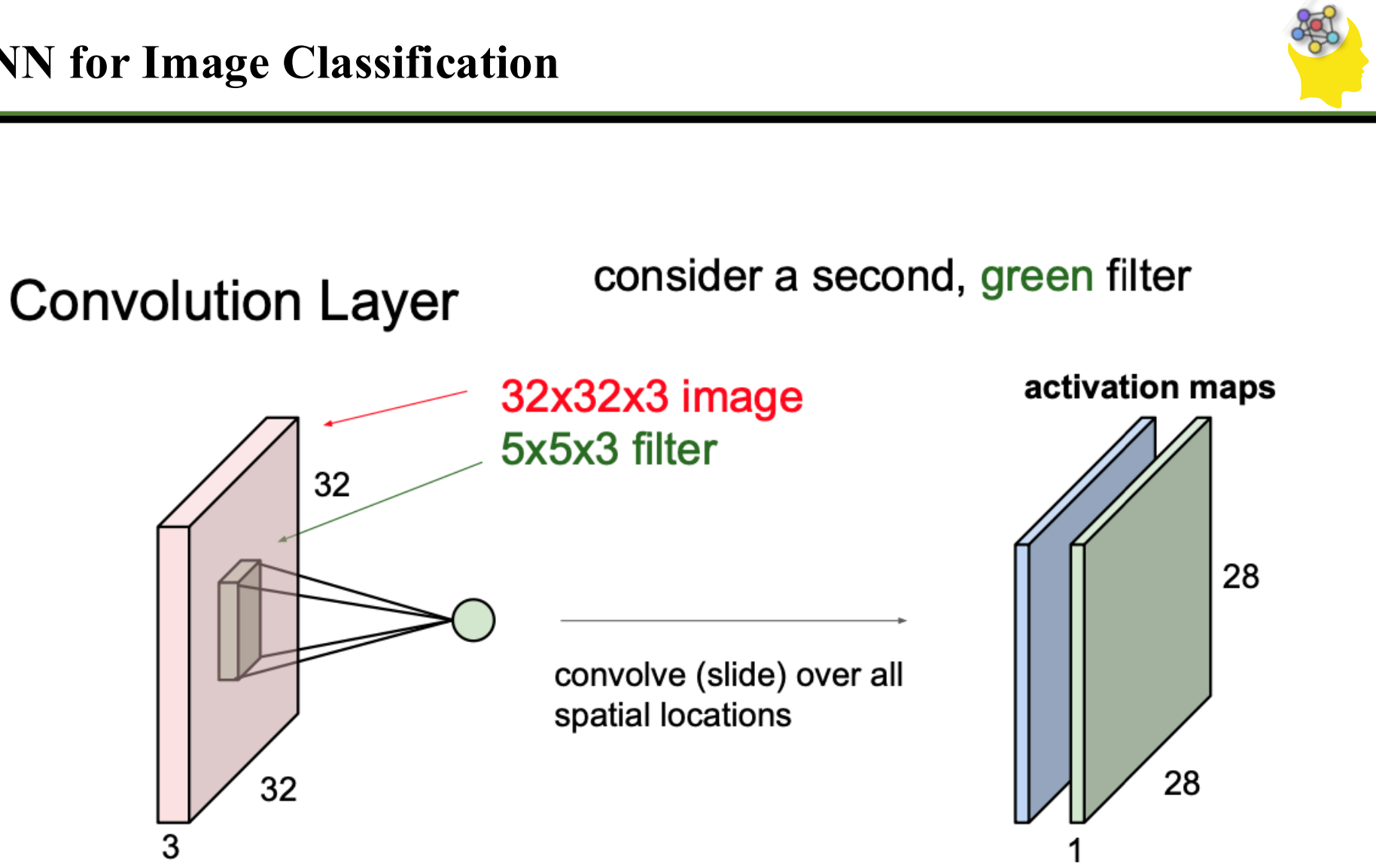
CNN for Image Classification



What about training a lot of such "small" detectors and each detector must "move around".

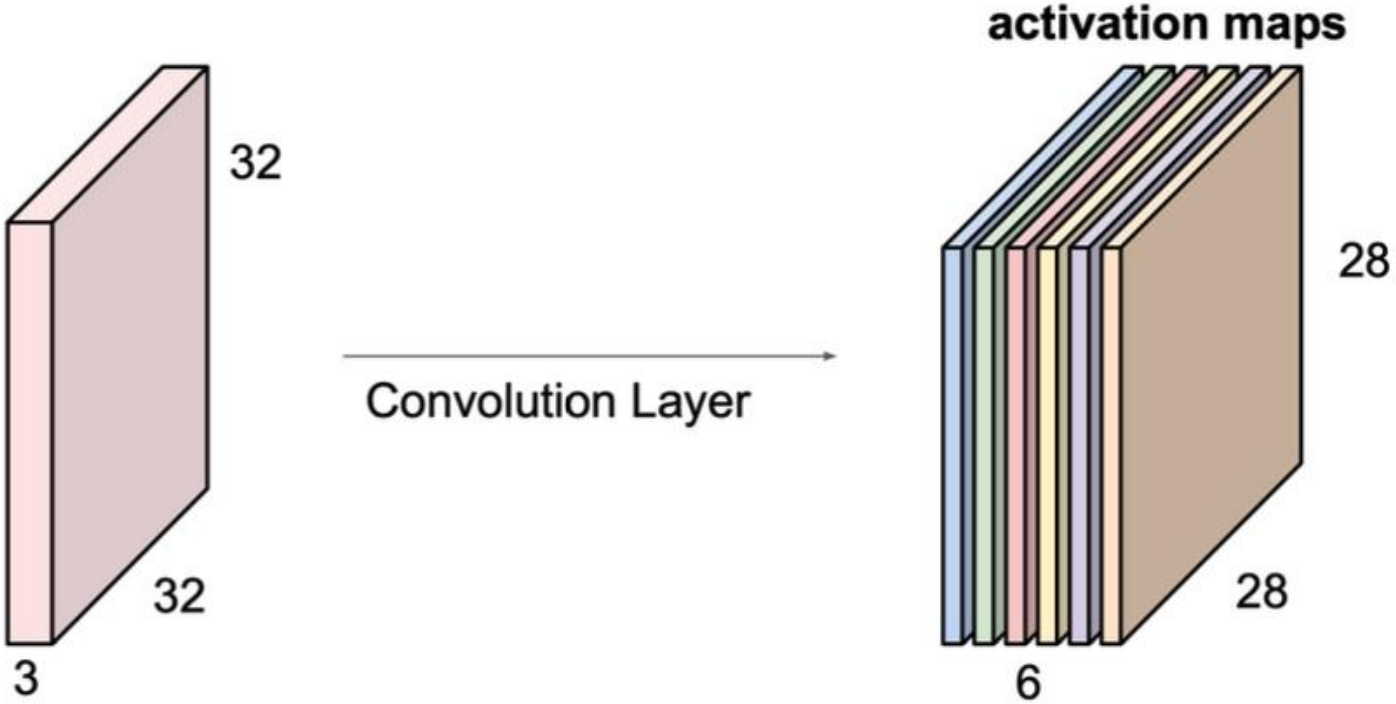
Convolution Layer




Convolution Layer
consider a second, green filter
32x32x3 image
5x5x3 filter
32
32
3
convolve (slide) over all
spatial locations
activation maps
28
28
1

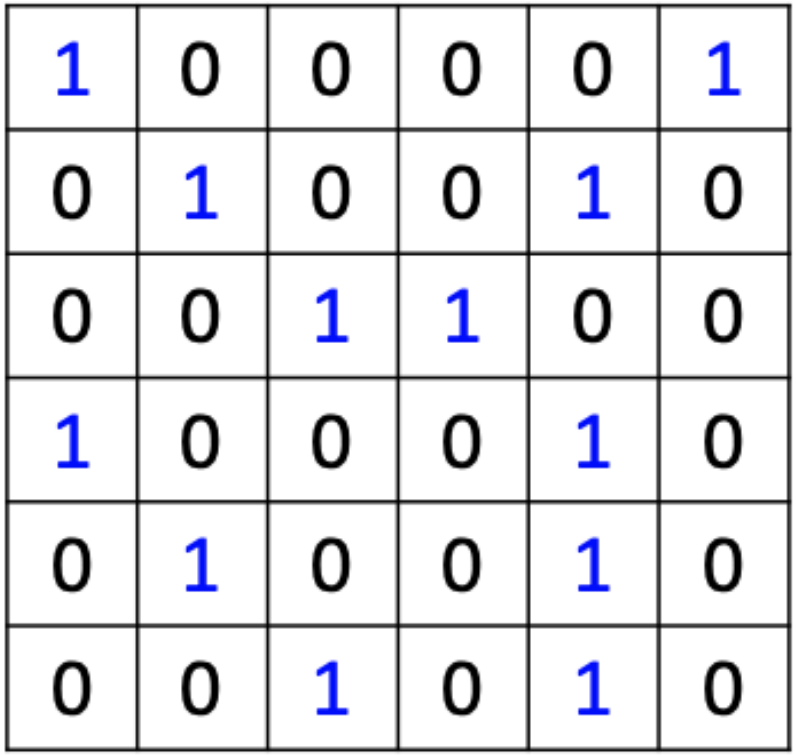
CNN for Image Classification

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



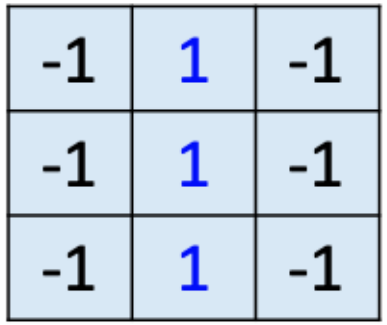
We stack these up to get a "new image" of size 28x28x6!

CNN for Image Classification



6 x 6 image

Filter 1

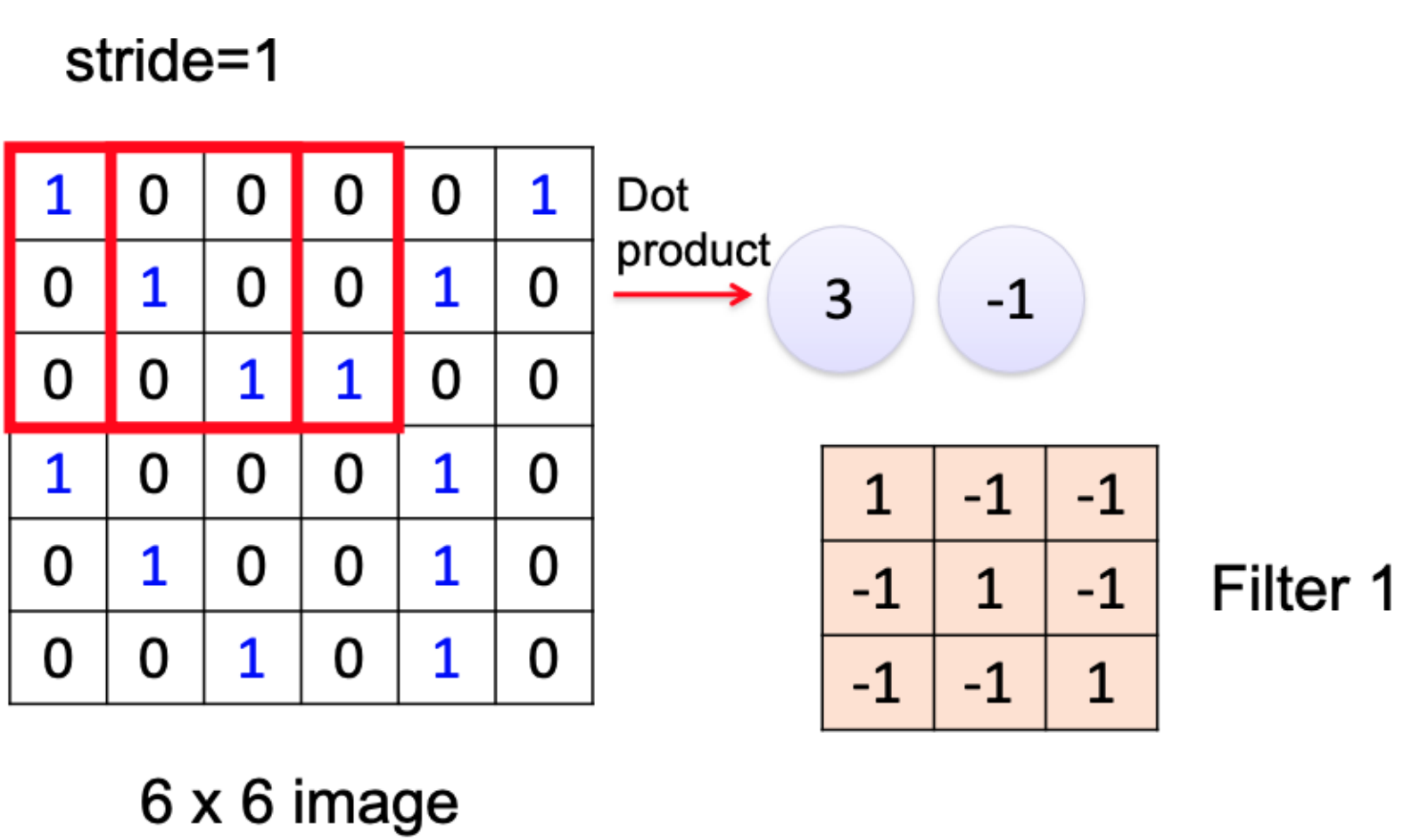


Filter 2

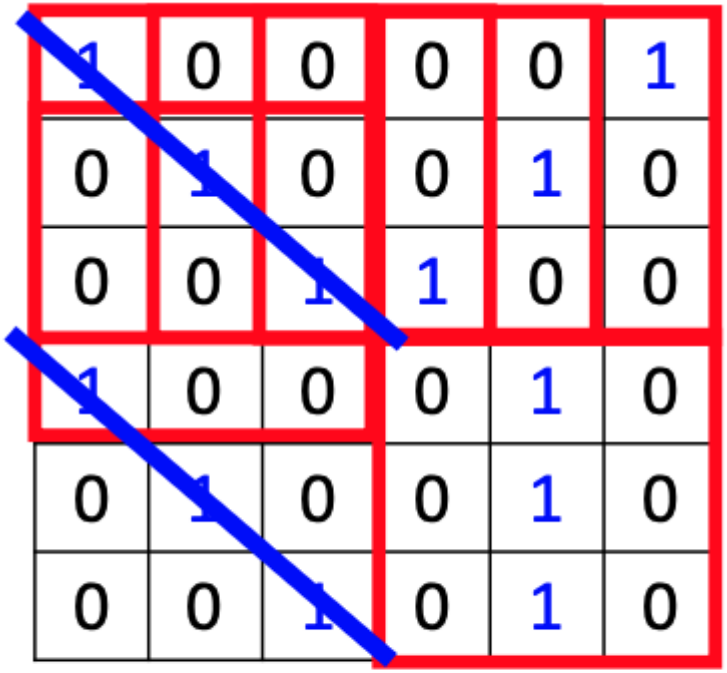
⋮ ⋮

Each filter detects a small pattern (3 x 3).

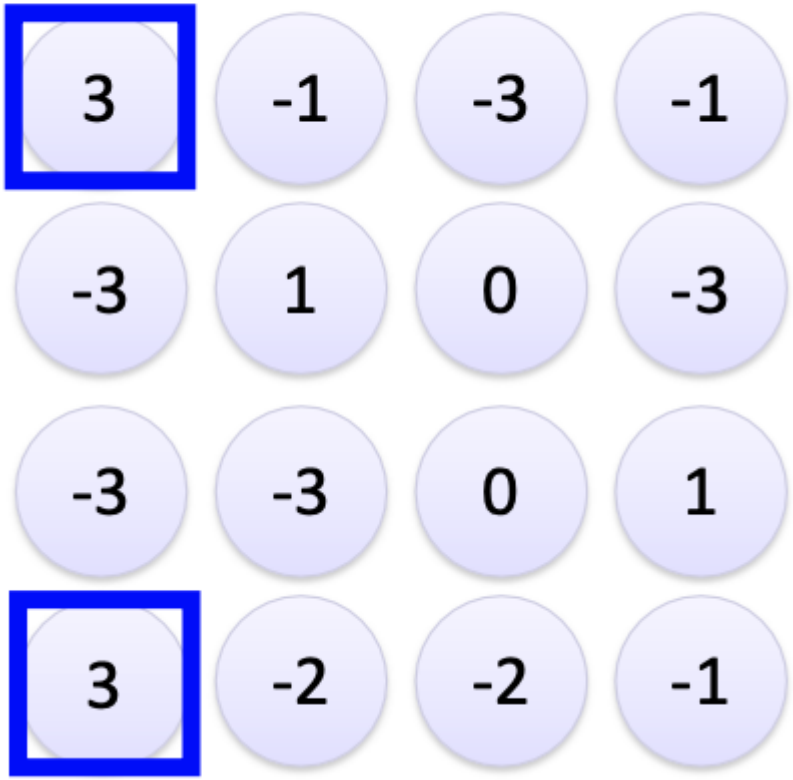
CNN for Image Classification



CNN for Image Classification



6 x 6 image



CNN for Image Classification

Convolutions (typically with *prespecified* filters) are a common operation in many computer vision applications

Original image z

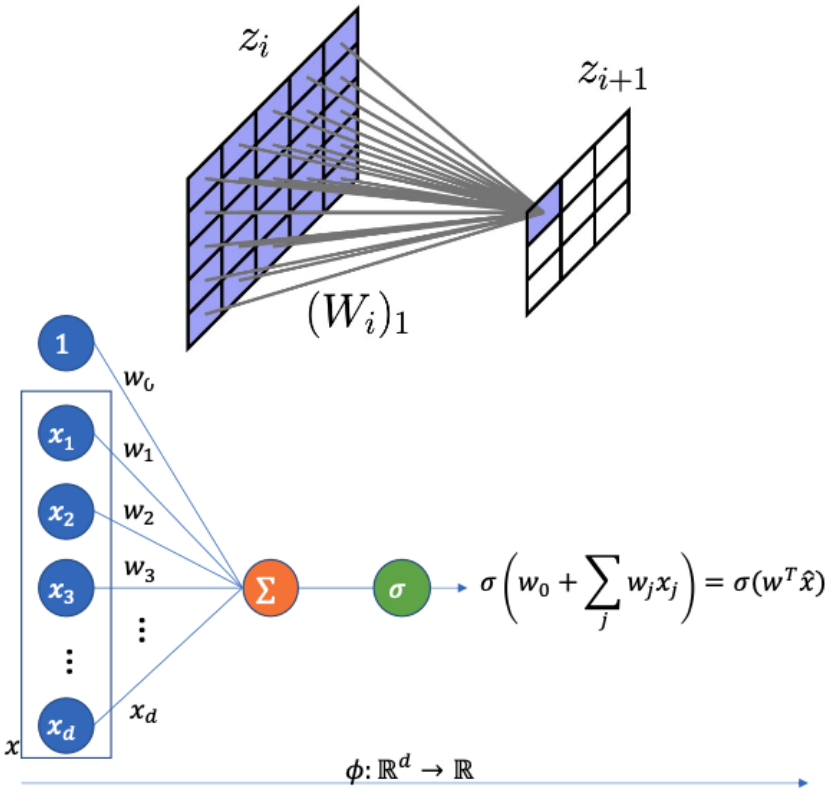
Gaussian blur

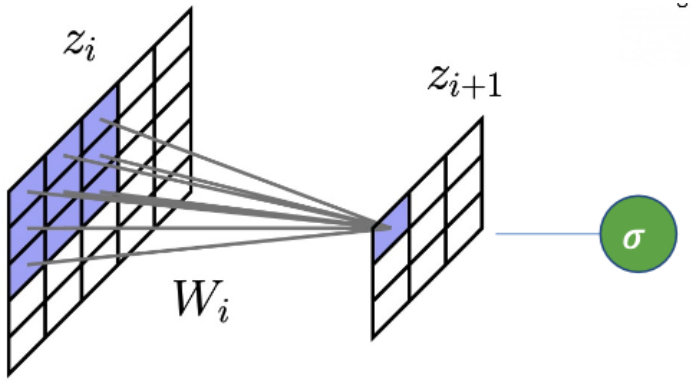
Image gradient

$$z * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 4 & 4 & 1 \end{bmatrix} /273$$

$$\left(\left(z * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \right)^2 + \left(z * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \right)^2 \right)^{\frac{1}{2}}$$

CNN for Image Classification





Convolution is a linear operator

We need to follow convolution with a nonlinearity (e.g., ReLU) to get nonlinear functions.

CNN for Image Classification

bird

bird

1	-1	-1
-1	1	-1
-1	-1	1

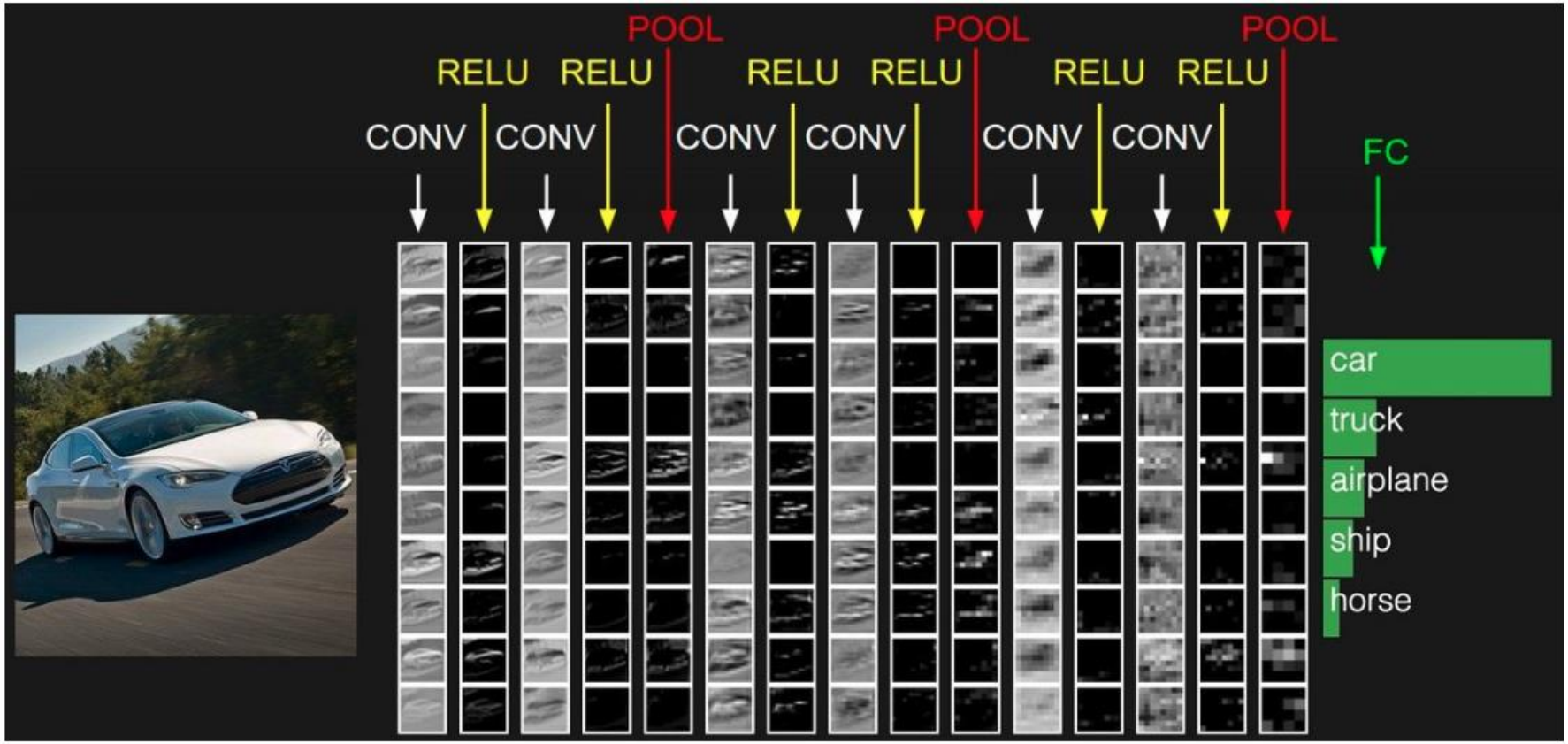
Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

3	-1	-3	-1	-1	-1	-1	-1
-3	1	0	-3	-1	-1	-2	1
-3	-3	0	1	-1	-1	-2	1
3	-2	-2	-1	-1	0	-4	3

CNN for Image Classification

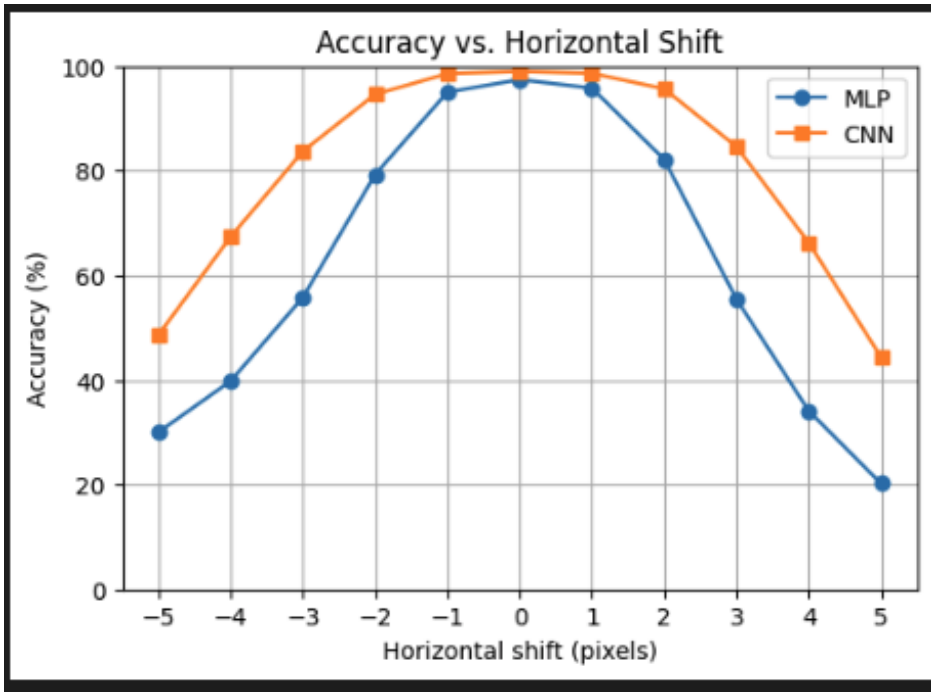


CNN for Image Classification

Define your CNN

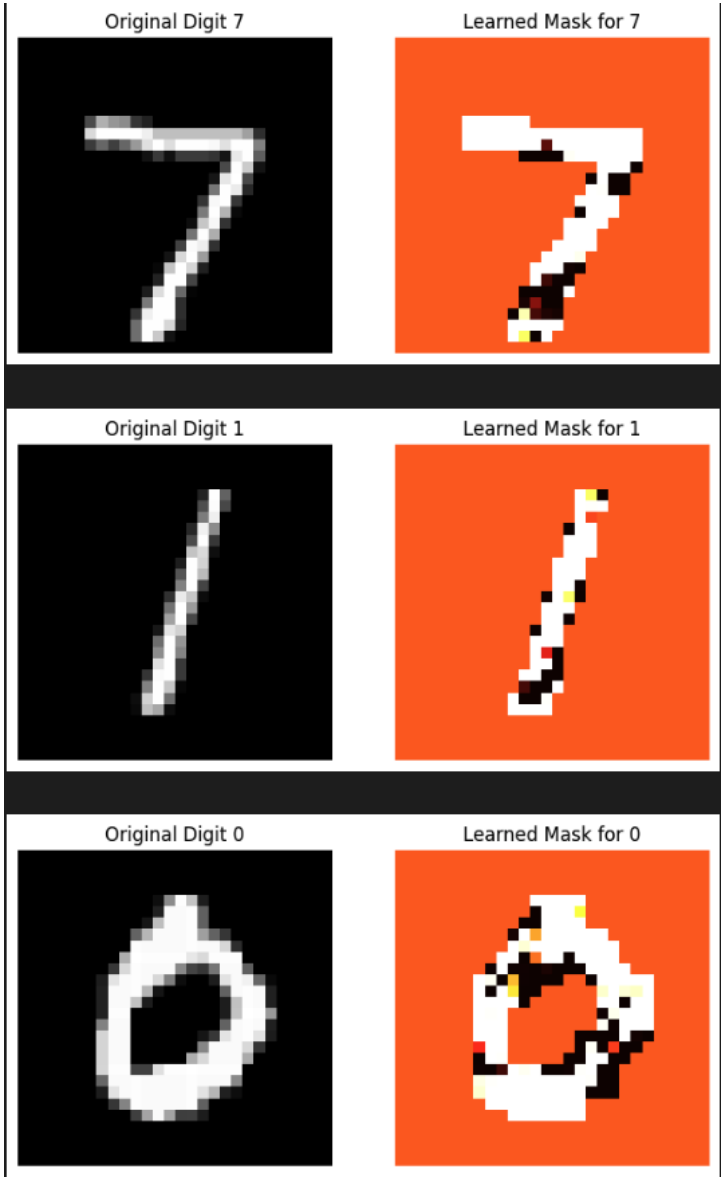
```python
import torch.nn as nn
import torch.optim as optim

# Define a simple CNN model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)  # 28x28 -> 28x28
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1) # 28x28 -> 28x28
        self.pool = nn.MaxPool2d(2, 2)  # 28x28 -> 14x14
        # Fully connected layers
        self.fc1 = nn.Linear(64 * 14 * 14, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = self.pool(x)  # Downsample
        x = x.view(x.size(0), -1)  # Flatten
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)  # Logits for 10 classes
        return x

# Initialize CNN, loss function, and optimizer
cnn_model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_model.parameters(), lr=0.001)
```

```python
# DataLoader for training and testing
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1000, shuffle=False)

# Train CNN model
epochs = 5
cnn_model.train()  # Set to training mode
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()  # Reset gradients
        outputs = cnn_model(images)  # Forward pass
        loss = criterion(outputs, labels)  # Compute loss
        loss.backward()  # Backpropagation
        optimizer.step()  # Update weights
        running_loss += loss.item()

    avg_loss = running_loss / len(train_loader)
    print(f"Epoch {epoch+1}/{epochs}, Training Loss: {avg_loss:.4f}")
```

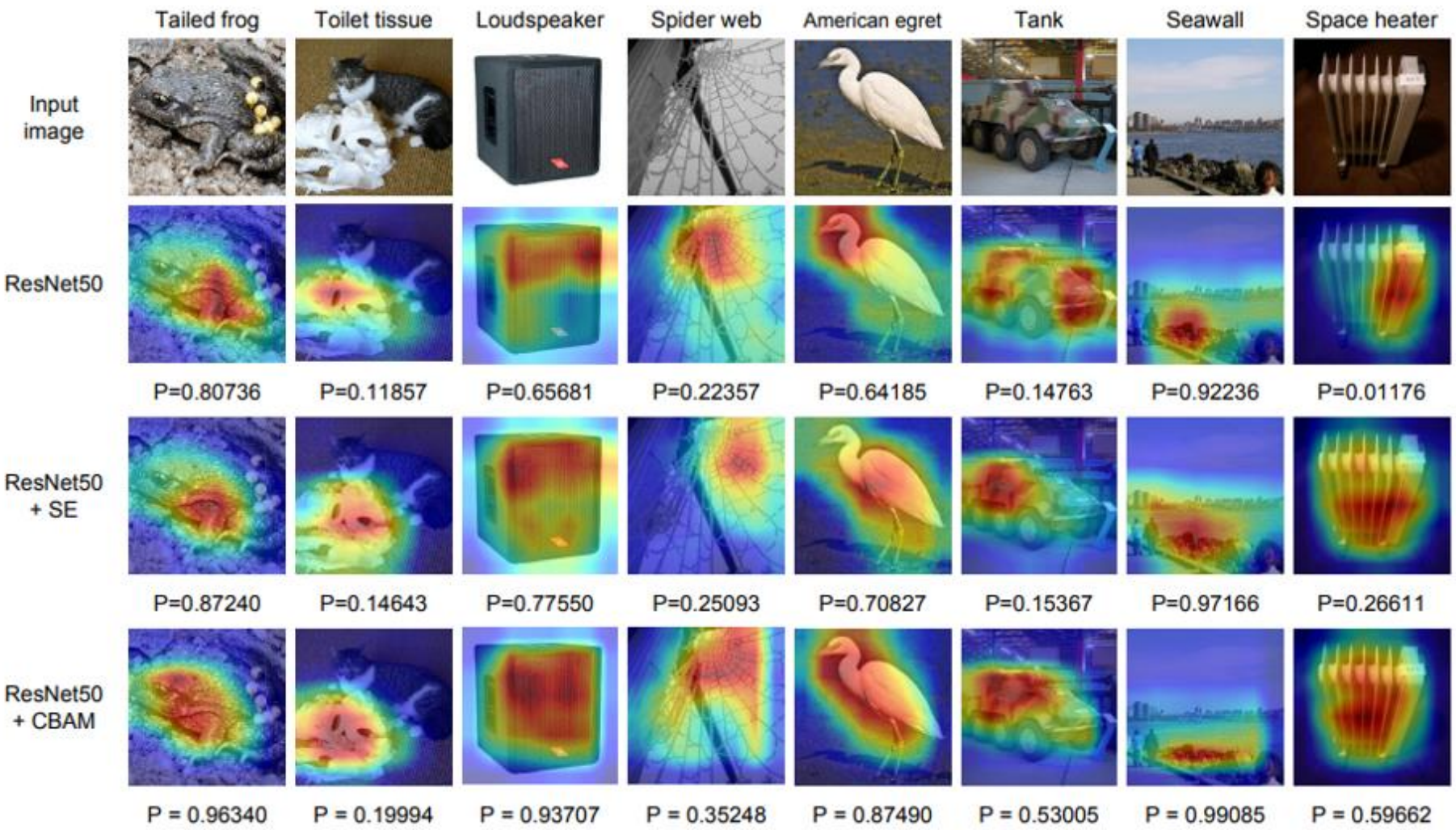


MLP for Image Classification

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

def optimize_mask(model, image, target_label, num_iters=300, lr=0.1):
    """
    Optimizes a mask matrix to highlight important pixels for a given image.

    - model: Trained MLP model
    - image: Input image (1x28x28)
    - target_label: True label of the image
    - num_iters: Number of optimization iterations
    - lr: Learning rate for optimization
    """
    model.eval()
    image = image.unsqueeze(0)  # Add batch dimension
    image = image.detach()  # Ensure no gradients flow into the original image

    # Initialize learnable mask with small random values
    mask = torch.zeros_like(image, requires_grad=True)

    # Define optimizer (only optimizing the mask)
    optimizer = optim.Adam([mask], lr=lr)

    for i in range(num_iters):
        optimizer.zero_grad()

        # Pass image through sigmoid mask to keep values between [0,1]
        masked_image = image * torch.sigmoid(mask)

        # Forward pass
        output = model(masked_image)
        confidence = output[0, target_label]  # Confidence for the correct class

        # Loss: Maximize confidence (sharpen prediction)
        loss = -confidence
        loss.backward()
        optimizer.step()

        # Optional: Print progress every 50 iterations
        if i % 50 == 0:
            print(f"Iteration {i}, Confidence: {confidence.item():.4f}")

    # Return optimized mask (after sigmoid activation)
    return torch.sigmoid(mask).detach()
```

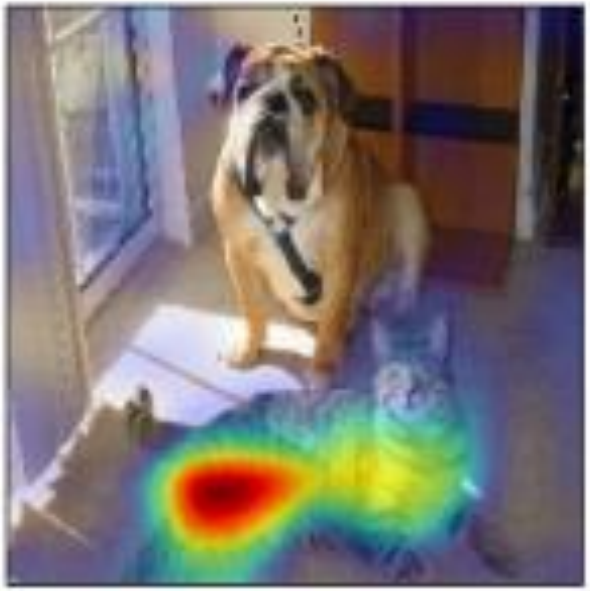


MLP for Image Classification



MLP for Image Classification

Original Image



Grad-CAM 'Cat'

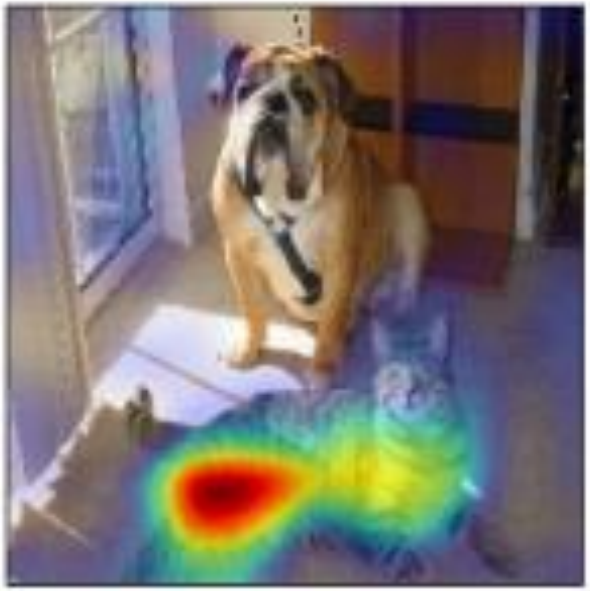
Grad-CAM 'Dog'

$$y = f_\theta(x) \qquad \frac{\partial y}{\partial x}$$

How much change of each input of unit will change the output

MLP for Image Classification

Original Image



Grad-CAM 'Cat'

Grad-CAM 'Dog'

$$y = f_\theta(M \odot x)$$

Can we learn a mask so that the prediction can become more confident or prediction can become less right?

Question Time!