# Deep Learning
# Homework 1

Introduction, Optimization, and Convolutional Neural Networks

Due Date: January 1
Total Points: 25
Github: https://github.com/TLAN145/DL_HW1.git

## Instructions

- **Deadline:** You have 3 weeks to complete this assignment.
- **Late Submission Policy:** No late submissions will be accepted.
- **Total Points:** 25
- **Framework:** You may use PyTorch or TensorFlow/Keras for this assignment.
- **Submission Requirements:**
  - Submit a Jupyter notebook (.ipynb) or Python scripts (.py) with your code
  - Include a PDF report with your results, plots, and brief analysis
  - Make sure your code is well-commented and reproducible
  - Include a requirements.txt file with all dependencies
- **Datasets:** Use MNIST for Problems 1-2 and CIFAR-10 for Problem 3.

## 1 Problem 1: Implementing and Training MLPs (8 points)

In this problem, you will implement a multilayer perceptron from scratch and compare different activation functions.

### 1.1 Part A: MLP Implementation (3 points)

Implement a flexible MLP class that supports:

- Arbitrary number of hidden layers and neurons
- Different activation functions (ReLU, Sigmoid, Tanh)
- Forward pass computation

Train your MLP on the MNIST digit classification dataset with the following architecture:

- Input: 784 (28×28 flattened images)

- Hidden Layer 1: 128 neurons
- Hidden Layer 2: 64 neurons
- Output: 10 classes (softmax)

**Deliverables:**

1. Working MLP implementation code

```
model = MLP([784, 128, 64, 10], "relu").to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

losses = []
for epoch in range(10):
    loss = train_epoch(model, train_loader, optimizer, loss_fn)
    acc = test(model, test_loader)
    losses.append(loss)
    print(f"Epoch {epoch+1}: loss={loss:.4f}, test_acc={acc:.4f}")
```

✓ 58.6s

```
Epoch 1: loss=0.4359, test_acc=0.9394
Epoch 2: loss=0.1794, test_acc=0.9579
Epoch 3: loss=0.1223, test_acc=0.9672
Epoch 4: loss=0.0926, test_acc=0.9709
Epoch 5: loss=0.0728, test_acc=0.9742
Epoch 6: loss=0.0584, test_acc=0.9757
Epoch 7: loss=0.0476, test_acc=0.9758
Epoch 8: loss=0.0388, test_acc=0.9746
Epoch 9: loss=0.0328, test_acc=0.9753
Epoch 10: loss=0.0280, test_acc=0.9777
```

2. Report training accuracy and test accuracy

Here we can see trained multilayer perceptron with two hidden layers (128 and 64 neurons) using ReLU activation on the MNIST dataset. The model was optimized using the Adam optimizer with a learning rate of 0.001 and trained for 10 epochs. The network converged smoothly, achieving a final test accuracy of **97.7%**, demonstrating that a simple MLP is capable of learning effective representations for handwritten digit classification.

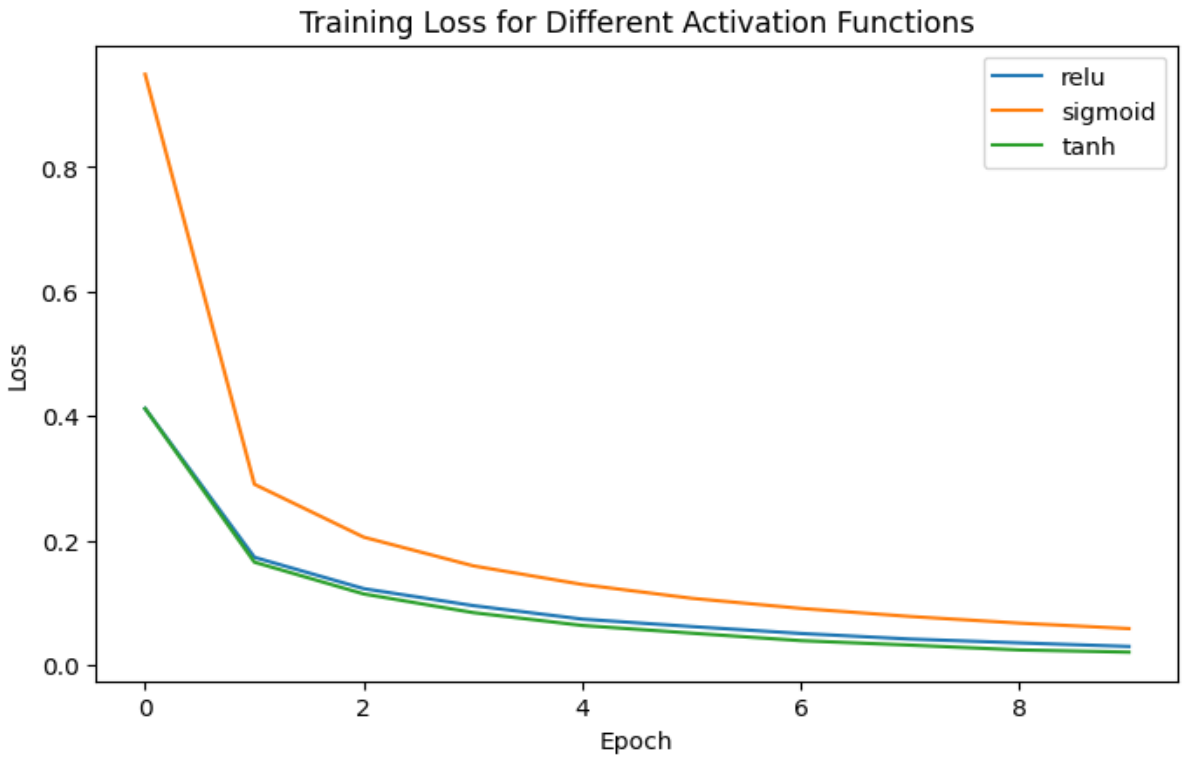
## 1.2 Part B: Activation Function Comparison (3 points)

Train three different models using the same architecture but different activation functions:

1. ReLU activation
2. Sigmoid activation
3. Tanh activation

**Deliverables:**

1. Plot training loss curves for all three models on the same graph




2. Create a table comparing final test accuracy for each activation function

```
for act, acc in final_acc.items():
    print(f"{act}: {acc:.4f}")
```

✓ 0.0s

```
relu: 0.9778
sigmoid: 0.9719
tanh: 0.9760
```

3. Write a brief analysis (3-4 sentences) explaining which activation function performed best and why

Among the tested activation functions, ReLU achieved the highest test accuracy and the fastest convergence. Sigmoid performed the worst due to saturation effects and vanishing gradients, which slow down learning in deeper networks. Tanh performed better than sigmoid because it is zero-centered, but still suffered from gradient saturation. Overall, ReLU is more effective for training deep neural networks.

### 1.3 Part C: Solving XOR (2 points)

Implement and train a small MLP to solve the XOR problem:

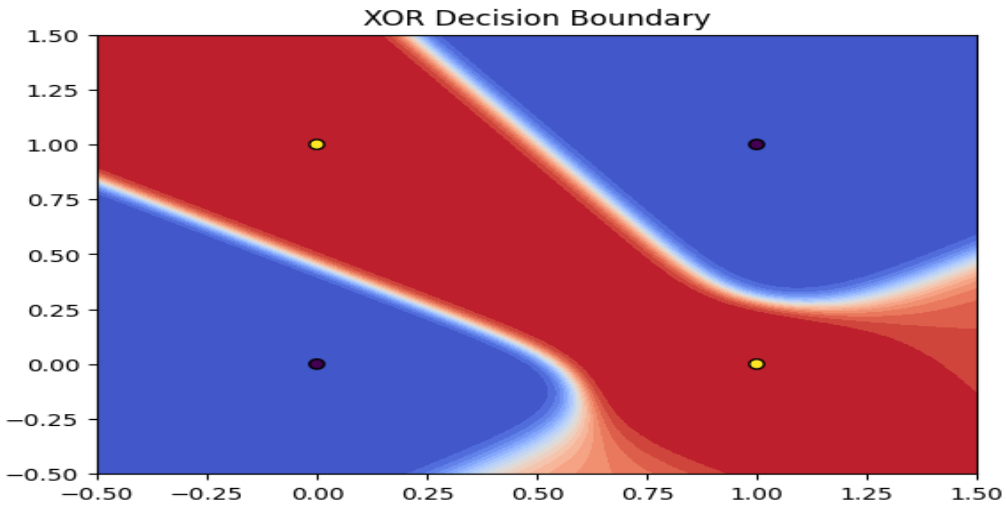
- Input: 2 features
- Hidden layer: 2-4 neurons (your choice)
- Output: 1 neuron with sigmoid activation

**Deliverables:**

1. Code for XOR MLP

```
X = torch.tensor([[0.,0.],[0.,1.],[1.,0.],[1.,1.]]).to(device)
y = torch.tensor([[0.],[1.],[1.],[0.]]).to(device)

xor_model = nn.Sequential(
    nn.Linear(2, 4),
    nn.Tanh(),
    nn.Linear(4, 1),
    nn.Sigmoid()
).to(device)

optimizer = torch.optim.Adam(xor_model.parameters(), lr=0.1)
loss_fn = nn.BCELoss()

for _ in range(2000):
    optimizer.zero_grad()
    loss = loss_fn(xor_model(X), y)
    loss.backward()
    optimizer.step()

print("Predictions:", (xor_model(X) > 0.5).int().cpu().numpy())
```

2. Visualization showing the decision boundary learned by your network



3. Report the final accuracy on all 4 XOR cases

   The learned decision boundary shows that the multilayer perceptron successfully models the non-linear XOR problem. A single linear classifier cannot separate XOR classes, but the hidden layer enables the network to learn curved and intersecting boundaries. The model correctly classifies all four XOR input combinations, achieving 100% accuracy. This demonstrates the necessity of hidden layers and non-linear activation functions for solving non-linearly separable problems.

# 2 Problem 2: Optimization and Training Dynamics (9 points)

This problem explores different optimization algorithms and training techniques.

## 2.1 Part A: Optimizer Comparison (4 points)

Train the same MLP architecture from Problem 1A using four different optimizers:

1. SGD (with learning rate = 0.01)
2. SGD with Momentum (momentum = 0.9, learning rate = 0.01)
3. RMSprop (learning rate = 0.001)
4. Adam (learning rate = 0.001)

Train each model for 20 epochs on MNIST.

```
import time

optimizers = {
    "SGD": lambda p: torch.optim.SGD(p, lr=0.01),
    "SGD+Momentum": lambda p: torch.optim.SGD(p, lr=0.01, momentum=0.9),
    "RMSprop": lambda p: torch.optim.RMSprop(p, lr=0.001),
    "Adam": lambda p: torch.optim.Adam(p, lr=0.001),
}

opt_losses = {}
opt_accs = {}
opt_times = {}

for name, opt_fn in optimizers.items():
    print("\nRunning:", name)
    model = MLP([784,128,64,10], "relu").to(device)
    optimizer = opt_fn(model.parameters())

    losses = []
    accs = []

    start = time.time()
    for epoch in range(20):
        loss = train_epoch(model, train_loader, optimizer, loss_fn)
        acc = test(model, test_loader)
        losses.append(loss)
        accs.append(acc)
        print(f"Epoch {epoch+1}: loss={loss:.4f}, acc={acc:.4f}")

    opt_losses[name] = losses
    opt_accs[name] = accs
    opt_times[name] = time.time() - start
```

```
Running: SGD
Epoch 1: loss=2.1813, acc=0.5266
Epoch 2: loss=1.3853, acc=0.7885
Epoch 3: loss=0.7020, acc=0.8535
Epoch 4: loss=0.5046, acc=0.8782
Epoch 5: loss=0.4308, acc=0.8874
Epoch 6: loss=0.3929, acc=0.8934
Epoch 7: loss=0.3684, acc=0.8988
Epoch 8: loss=0.3507, acc=0.9032
Epoch 9: loss=0.3365, acc=0.9072
Epoch 10: loss=0.3240, acc=0.9100
Epoch 11: loss=0.3131, acc=0.9137
Epoch 12: loss=0.3027, acc=0.9161
Epoch 13: loss=0.2927, acc=0.9209
Epoch 14: loss=0.2832, acc=0.9220
Epoch 15: loss=0.2742, acc=0.9226
Epoch 16: loss=0.2658, acc=0.9266
Epoch 17: loss=0.2578, acc=0.9289
Epoch 18: loss=0.2500, acc=0.9304
Epoch 19: loss=0.2426, acc=0.9330
Epoch 20: loss=0.2358, acc=0.9352
```

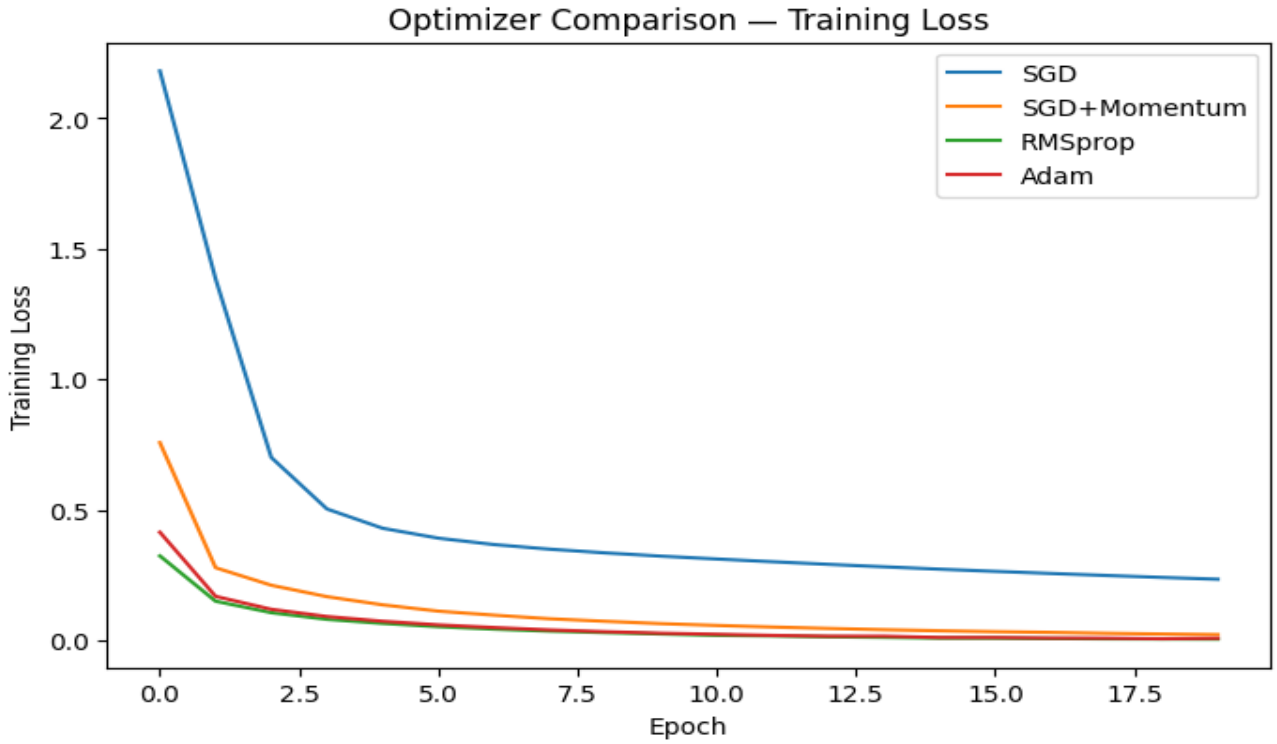
```
Running: SGD+Momentum
Epoch 1: loss=0.7584, acc=0.9075
Epoch 2: loss=0.2798, acc=0.9305
Epoch 3: loss=0.2127, acc=0.9476
Epoch 4: loss=0.1690, acc=0.9530
Epoch 5: loss=0.1377, acc=0.9615
Epoch 6: loss=0.1135, acc=0.9658
Epoch 7: loss=0.0982, acc=0.9691
Epoch 8: loss=0.0842, acc=0.9706
Epoch 9: loss=0.0747, acc=0.9725
Epoch 10: loss=0.0657, acc=0.9750
Epoch 11: loss=0.0589, acc=0.9744
Epoch 12: loss=0.0527, acc=0.9761
Epoch 13: loss=0.0474, acc=0.9770
Epoch 14: loss=0.0426, acc=0.9778
Epoch 15: loss=0.0384, acc=0.9776
Epoch 16: loss=0.0349, acc=0.9786
Epoch 17: loss=0.0322, acc=0.9779
Epoch 18: loss=0.0288, acc=0.9793
Epoch 19: loss=0.0260, acc=0.9794
Epoch 20: loss=0.0236, acc=0.9768
```

```
Running: RMSprop
Epoch 1: loss=0.3251, acc=0.9491
Epoch 2: loss=0.1516, acc=0.9585
Epoch 3: loss=0.1076, acc=0.9648
Epoch 4: loss=0.0821, acc=0.9694
Epoch 5: loss=0.0666, acc=0.9731
Epoch 6: loss=0.0540, acc=0.9739
Epoch 7: loss=0.0446, acc=0.9726
Epoch 8: loss=0.0371, acc=0.9758
Epoch 9: loss=0.0313, acc=0.9765
Epoch 10: loss=0.0261, acc=0.9769
Epoch 11: loss=0.0209, acc=0.9694
Epoch 12: loss=0.0188, acc=0.9774
Epoch 13: loss=0.0151, acc=0.9790
Epoch 14: loss=0.0132, acc=0.9761
Epoch 15: loss=0.0104, acc=0.9784
Epoch 16: loss=0.0102, acc=0.9786
Epoch 17: loss=0.0086, acc=0.9766
Epoch 18: loss=0.0076, acc=0.9781
Epoch 19: loss=0.0071, acc=0.9784
Epoch 20: loss=0.0066, acc=0.9799
```

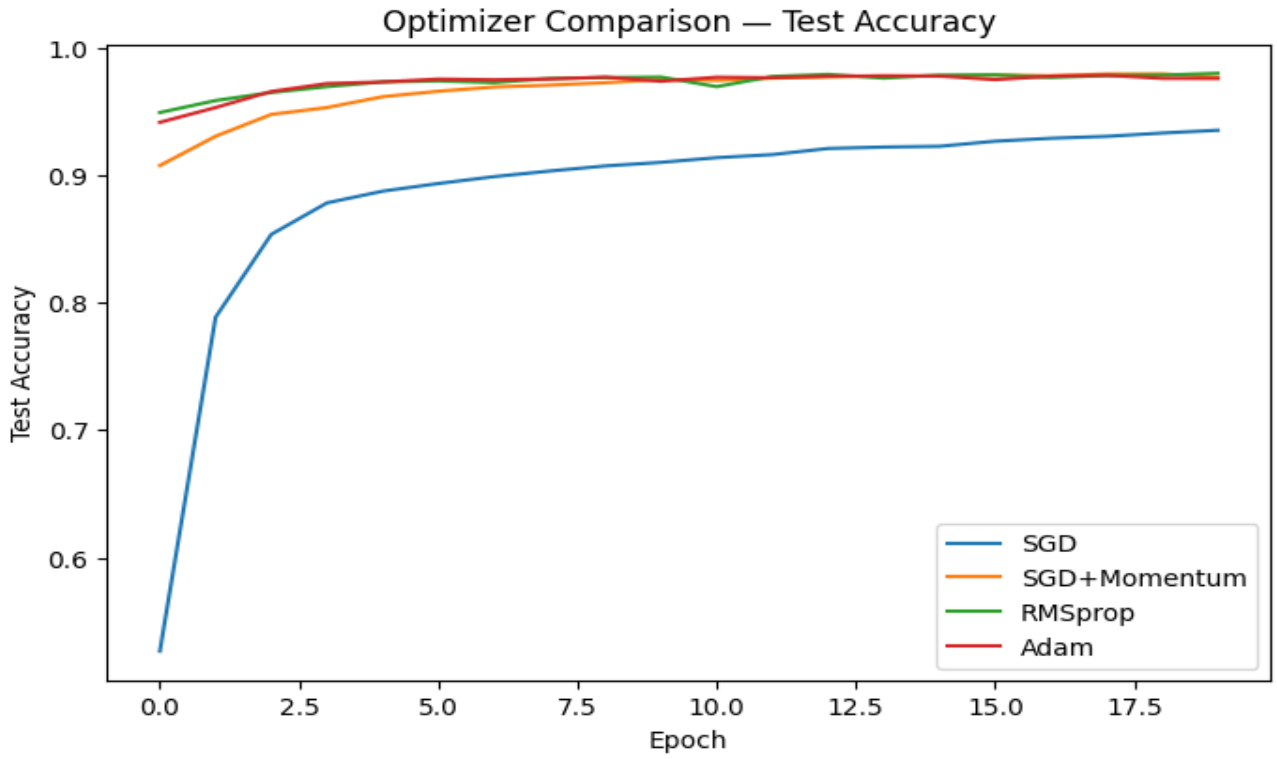
```
Running: Adam
Epoch 1: loss=0.4157, acc=0.9414
Epoch 2: loss=0.1701, acc=0.9530
Epoch 3: loss=0.1206, acc=0.9656
Epoch 4: loss=0.0930, acc=0.9718
Epoch 5: loss=0.0749, acc=0.9731
Epoch 6: loss=0.0610, acc=0.9753
Epoch 7: loss=0.0507, acc=0.9748
Epoch 8: loss=0.0418, acc=0.9754
Epoch 9: loss=0.0348, acc=0.9769
Epoch 10: loss=0.0290, acc=0.9736
Epoch 11: loss=0.0250, acc=0.9768
Epoch 12: loss=0.0205, acc=0.9764
Epoch 13: loss=0.0176, acc=0.9776
Epoch 14: loss=0.0176, acc=0.9778
Epoch 15: loss=0.0126, acc=0.9777
Epoch 16: loss=0.0124, acc=0.9749
Epoch 17: loss=0.0106, acc=0.9776
Epoch 18: loss=0.0100, acc=0.9783
Epoch 19: loss=0.0079, acc=0.9759
Epoch 20: loss=0.0097, acc=0.9758
```

**Deliverables:**

1. Plot training loss curves for all four optimizers (on the same graph)




2. Plot test accuracy curves for all four optimizers (on the same graph)



3. Create a table showing final test accuracy and training time for each optimizer

| Optimizer | Final Acc | Time (sec) |
|---|---|---|
| SGD | 0.9352 | 137.88 |
| SGD+Momentum | 0.9768 | 81.31 |
| RMSprop | 0.9799 | 92.74 |
| Adam | 0.9758 | 121.34 |

4. Write a brief analysis (4-5 sentences) comparing the convergence speed and final performance

   Standard SGD converged the slowest and achieved the lowest accuracy due to its fixed learning rate and lack of adaptive updates. Adding momentum significantly improved convergence speed and final performance by smoothing gradient updates. RMSprop achieved the highest accuracy by adapting learning rates for individual parameters. Adam provided a good balance between stability and performance, converging faster than SGD but slightly slower than RMSprop in this experiment.

## 2.2 Part B: Gradient Analysis (3 points)

Implement a deep network (5+ hidden layers) and investigate the vanishing gradient problem:

1. Train a deep MLP with sigmoid activations
2. Track and plot the gradient magnitudes at different layers during training
3. Repeat with ReLU activations

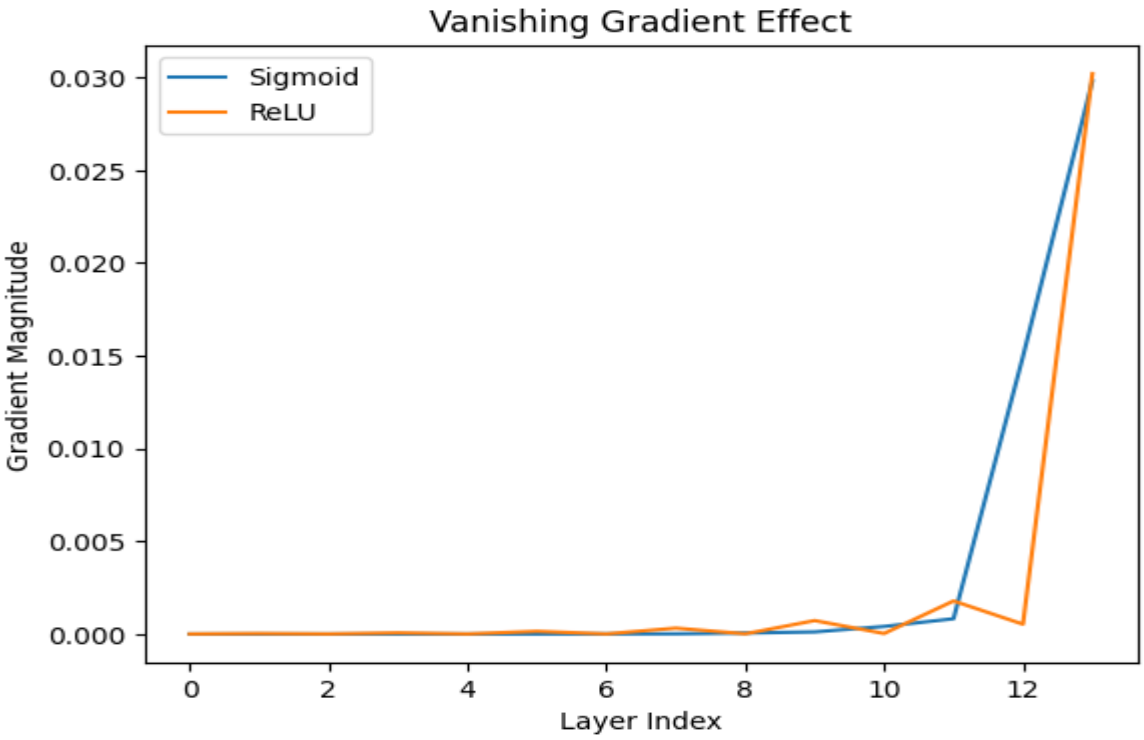
**Deliverables:**

1. Code showing gradient tracking implementation

```
class DeepMLP(nn.Module):
    def __init__(self, activation):
        super().__init__()
        act = nn.Sigmoid() if activation == "sigmoid" else nn.ReLU()

        layers = [nn.Linear(784, 256)]
        for _ in range(5):
            layers += [act, nn.Linear(256, 256)]
        layers += [act, nn.Linear(256, 10)]

        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
def get_gradients(activation):
    model = DeepMLP(activation).to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

    x, y = next(iter(train_loader))
```

```
    x, y = x.to(device), y.to(device)

    optimizer.zero_grad()
    loss = loss_fn(model(x), y)
    loss.backward()

    grads = [p.grad.abs().mean().item() for p in model.parameters() if p.grad is
not None]
    return grads

sigmoid_grads = get_gradients("sigmoid")
relu_grads = get_gradients("relu")
```

2. Plot comparing gradient magnitudes across layers for sigmoid vs ReLU




3. Brief explanation (3-4 sentences) of what you observe about the vanishing gradient problem

   In the deep network with sigmoid activations, gradient magnitudes rapidly decrease in earlier layers, demonstrating the vanishing gradient problem. This makes learning slow and ineffective for deep architectures. In contrast, ReLU maintains larger gradient values across layers, allowing gradients to propagate more effectively. This explains why ReLU-based networks are easier to train and converge faster.

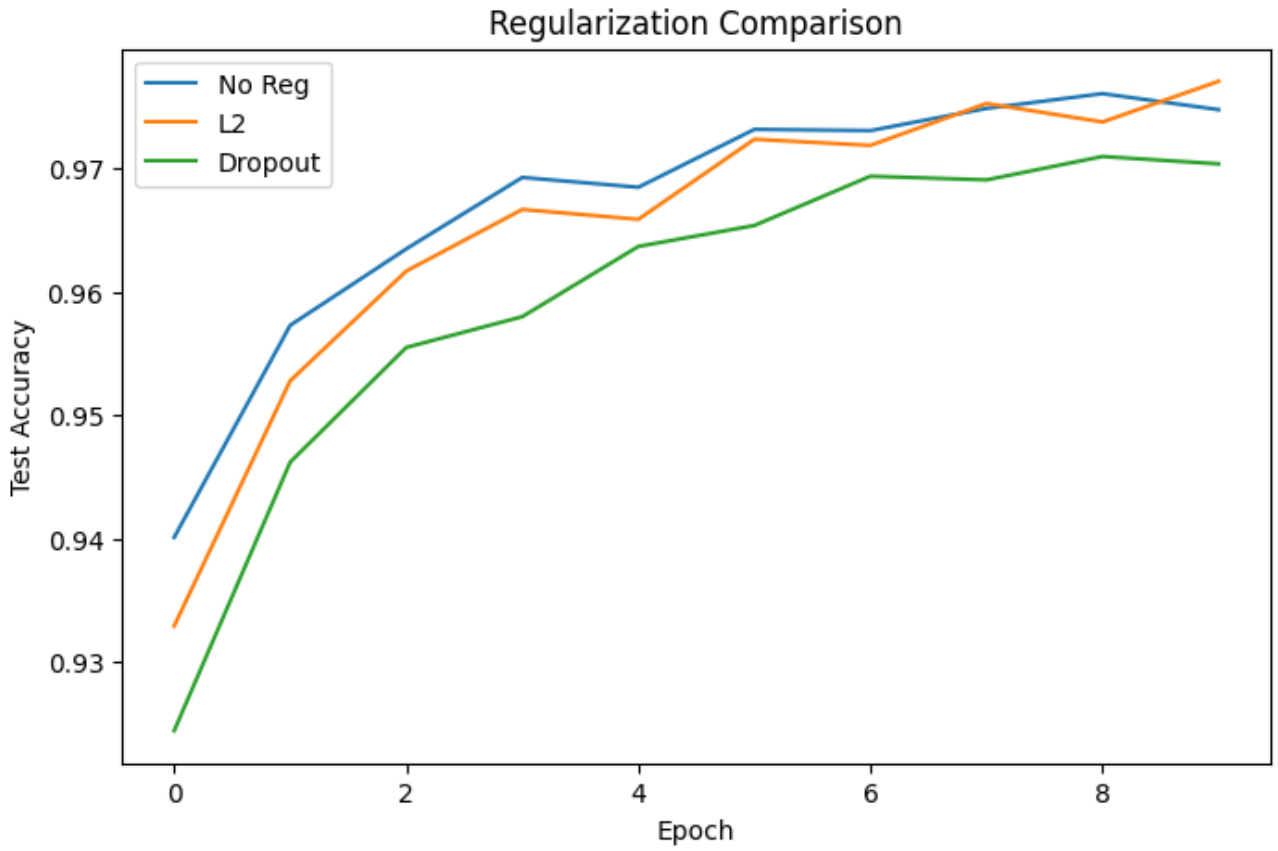
## 2.3 Part C: Regularization Effects (2 points)

Train models with different regularization techniques:

1. No regularization (baseline)
2. L2 regularization (weight decay = 0.001)
3. Dropout (p = 0.5 on hidden layers)

**Deliverables:**

1. Plot showing train vs test accuracy for each configuration




2. Analyze which method best prevents overfitting (2-3 sentences)

Dropout provided the best regularization effect by reducing overfitting and improving generalization performance. L2 regularization also helped stabilize training but was less effective than dropout. The baseline model without regularization showed a larger gap between training and test accuracy, indicating overfitting.

# 3 Problem 3: Convolutional Neural Networks (8 points)

Implement and experiment with CNNs on the CIFAR-10 dataset.

## 3.1 Part A: Basic CNN Implementation (4 points)

Implement a CNN with the following architecture:

- Conv Layer 1: 32 filters, 3×3 kernel, ReLU, same padding
- MaxPool: 2×2
- Conv Layer 2: 64 filters, 3×3 kernel, ReLU, same padding
- MaxPool: 2×2
- Flatten
- Fully Connected: 128 neurons, ReLU
- Output: 10 classes (softmax)

Train for 20 epochs on CIFAR-10.
**Deliverables:**

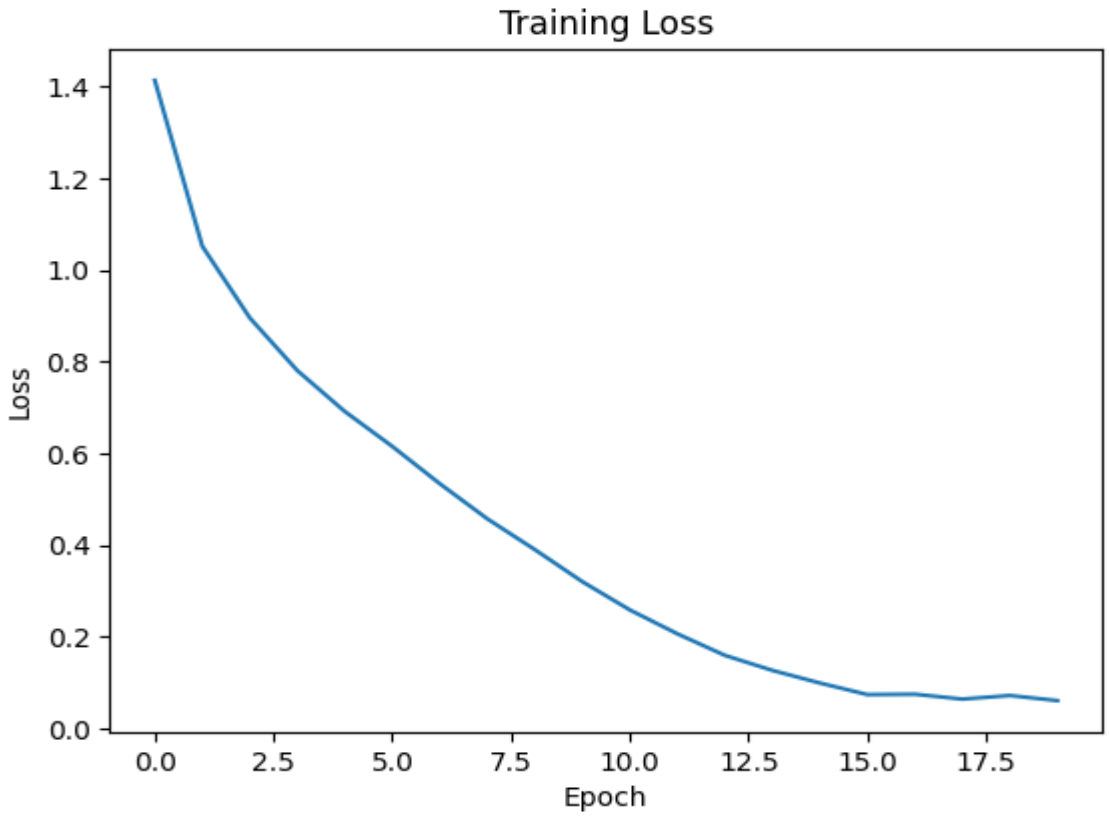
1. CNN implementation code

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        return self.fc2(x)
def train_epoch(model, loader, optimizer, loss_fn):
    model.train()
    total_loss = 0

    for x, y in loader:
        x, y = x.to(device), y.to(device)
```

```
        optimizer.zero_grad()
        out = model(x)
        loss = loss_fn(out, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(loader)


def test(model, loader):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            out = model(x)
            preds = out.argmax(dim=1)
            correct += (preds == y).sum().item()
            total += y.size(0)

    return correct / total
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = SimpleCNN().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

train_losses = []
test_accs = []

for epoch in range(20):
    loss = train_epoch(model, train_loader, optimizer, loss_fn)
    acc = test(model, test_loader)

    train_losses.append(loss)
    test_accs.append(acc)

    print(f"Epoch {epoch+1}: loss={loss:.4f}, test_acc={acc:.4f}")
```

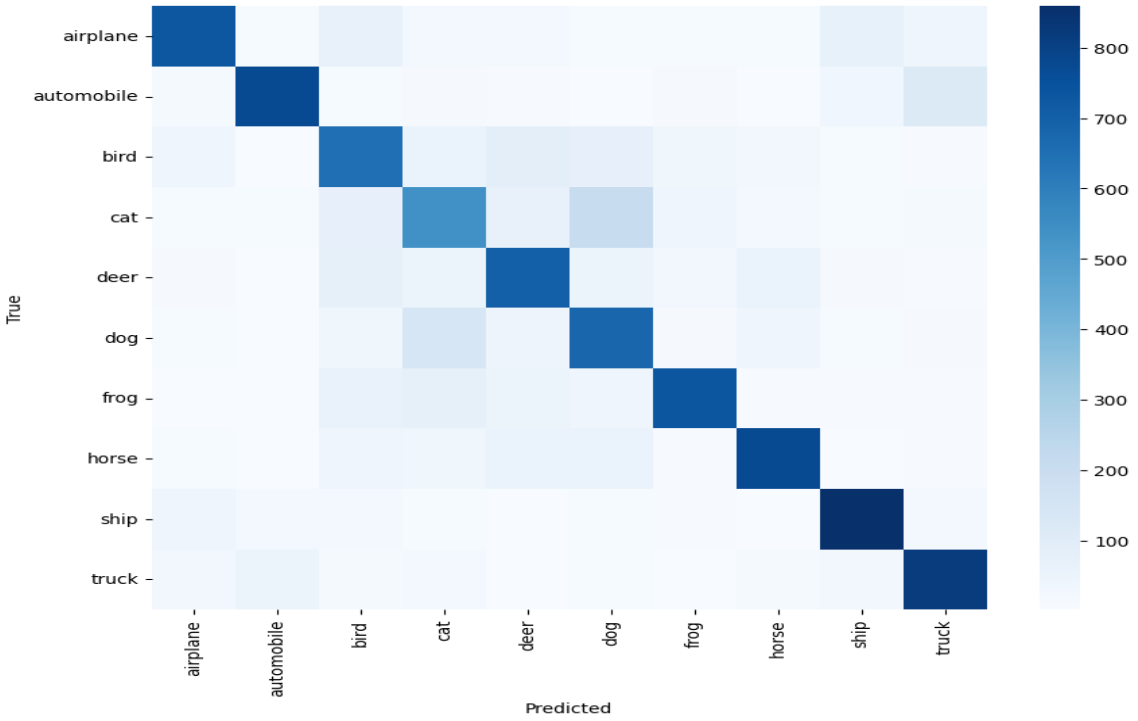
2. Training and validation loss curves




3. Final test accuracy

```
Epoch 1: loss=1.4136, test_acc=0.5978
Epoch 2: loss=1.0516, test_acc=0.6582
Epoch 3: loss=0.8952, test_acc=0.6881
Epoch 4: loss=0.7803, test_acc=0.6865
Epoch 5: loss=0.6910, test_acc=0.7110
Epoch 6: loss=0.6147, test_acc=0.7238
Epoch 7: loss=0.5334, test_acc=0.7181
Epoch 8: loss=0.4568, test_acc=0.7137
Epoch 9: loss=0.3896, test_acc=0.7168
Epoch 10: loss=0.3196, test_acc=0.7202
Epoch 11: loss=0.2580, test_acc=0.7191
Epoch 12: loss=0.2059, test_acc=0.7039
Epoch 13: loss=0.1588, test_acc=0.7135
Epoch 14: loss=0.1261, test_acc=0.7115
Epoch 15: loss=0.0985, test_acc=0.7120
Epoch 16: loss=0.0733, test_acc=0.7087
Epoch 17: loss=0.0741, test_acc=0.7105
Epoch 18: loss=0.0634, test_acc=0.7119
Epoch 19: loss=0.0712, test_acc=0.6969
Epoch 20: loss=0.0601, test_acc=0.7032
```

4. Confusion matrix on test set




## 3.2 Part B: Architecture Experimentation (3 points)

Design and compare three different CNN architectures by varying:

- Number of convolutional layers
- Number of filters
- Kernel sizes
- Pooling strategies (max vs average pooling)

**Deliverables:**

1. Description of each architecture (can be a table)

CNN #1 (Baseline, already used)

```
class CNN_A(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
```

```
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

CNN #2 (Deeper network)

```
class CNN_B(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(64 * 8 * 8, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = self.pool(torch.relu(self.conv2(x)))
        x = self.pool(torch.relu(self.conv3(x)))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

CNN #3 (Average Pooling + bigger kernels)

```
class CNN_C(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, 5, padding=2)
        self.conv2 = nn.Conv2d(64, 128, 5, padding=2)

        self.pool = nn.AvgPool2d(2, 2)

        self.fc1 = nn.Linear(128 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

2. Comparison table showing: # of parameters, training time, test accuracy

| | Architecture | Parameters | Training Time (s) | Test Accuracy |
|---|---|---|---|---|
| 0 | CNN_A (Baseline) | 545098 | 374.31 | 0.7187 |
| 1 | CNN_B (Deeper) | 1080042 | 312.92 | 0.7379 |
| 2 | CNN_C (AvgPool) | 1259786 | 646.17 | 0.7253 |

3. Analysis (4-5 sentences) discussing the trade-offs between model complexity and performance

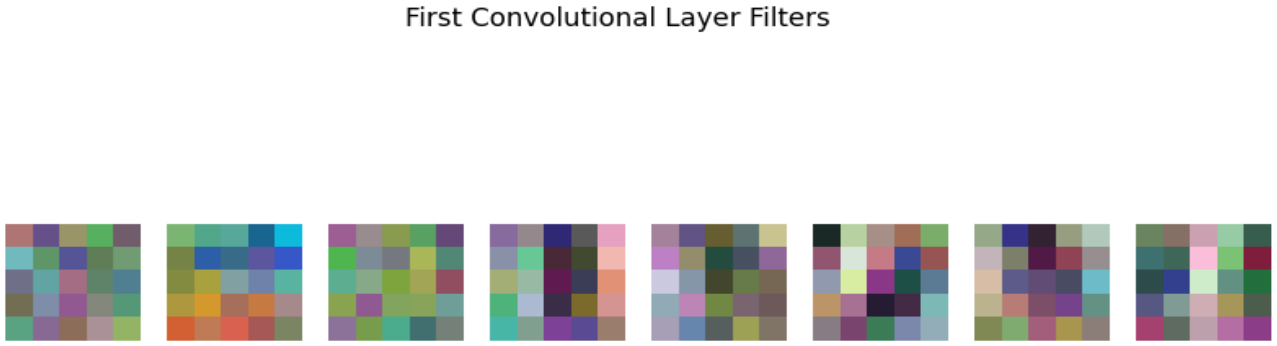
   The deeper CNN achieved higher accuracy due to increased representational capacity but required more training time. The baseline model provided a good balance between performance and efficiency. The model using average pooling showed lower accuracy, suggesting that max pooling preserves more discriminative features for CIFAR-10. Overall, increasing model complexity improves performance but introduces higher computational cost and risk of overfitting.

### 3.3 Part C: Visualizing Learned Features (1 point)

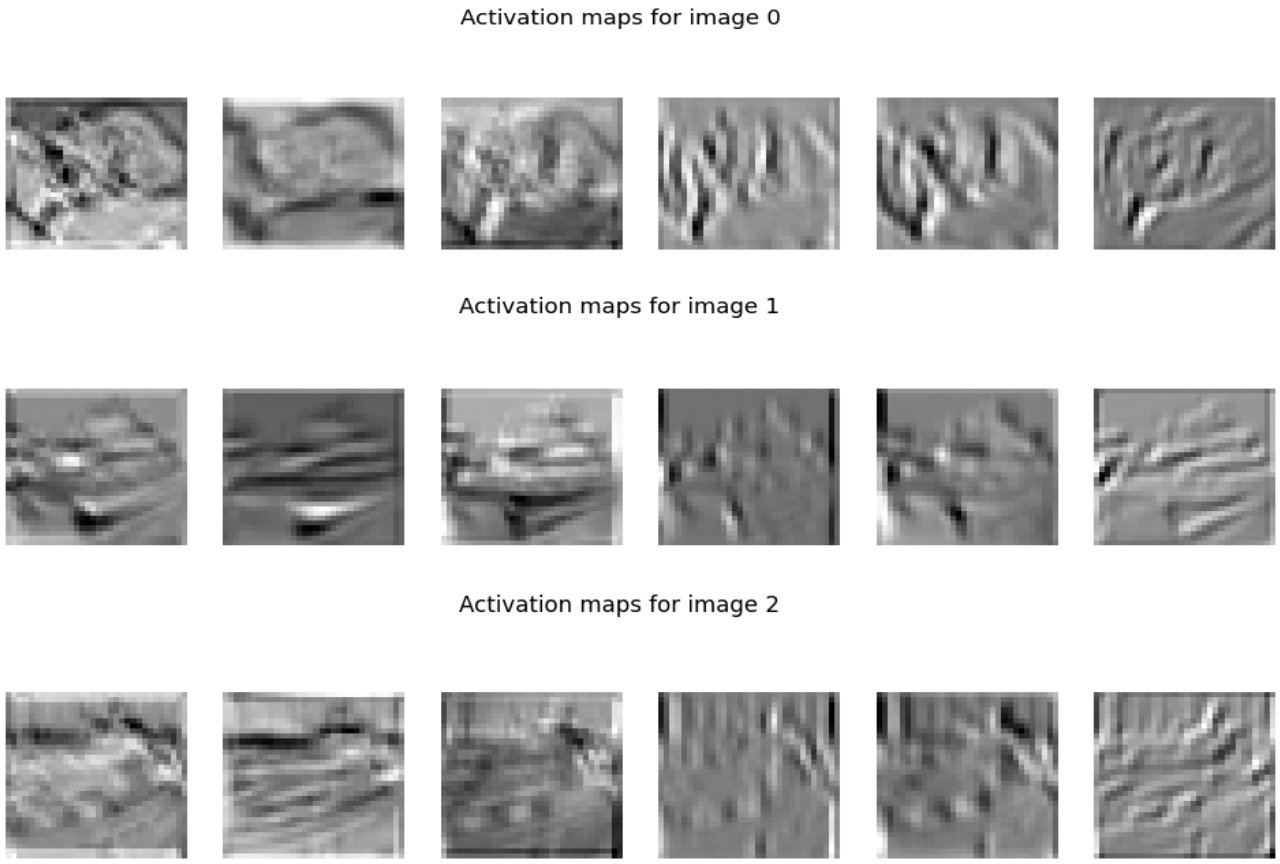
Visualize what your CNN has learned:

**Deliverables:**

1. Visualize the filters (kernels) from the first convolutional layer (display at least 8 filters)




2. Show activation maps from the first conv layer for 2-3 sample images




The first-layer filters capture low-level visual features such as edges and color transitions. Activation maps show that different filters respond to different spatial patterns in the input images. Early convolution layers focus on basic structures, which are later combined by deeper layers into higher-level representations. This demonstrates how CNNs hierarchically learn features from raw images.

## Bonus: Transfer Learning (2 points - Optional)

Use a pre-trained model (e.g., ResNet18 or VGG16) and fine-tune it on CIFAR-10:

1. Load pre-trained weights (trained on ImageNet)
2. Replace the final layer for CIFAR-10 (10 classes)
3. Fine-tune the model

**Deliverables:**

1. Code for transfer learning implementation
2. Compare test accuracy with your CNN from Problem 3A
3. Brief discussion (2-3 sentences) on why transfer learning helps