CENG403 - Spring 2025: Homework set THE-2 Study Guide

Your Name

TASK 1: DEFORMABLE CNN MEMORIZATION GUIDE

.1

Problem 1.1

What are the 4 corner positions for bilinear interpolation given fractional position $q = (q_x, q_y)$? **Pattern to Remember:** Floor-Floor, Ceil-Floor, Floor-Ceil, Ceil-Ceil

$$p_{lt} = (\lfloor q_x \rfloor, \lfloor q_y \rfloor) \quad \text{(left top)} \tag{1}$$
$$p_{rt} = (\lceil q_x \rceil, \lfloor q_y \rfloor) \quad \text{(right top)} \tag{2}$$
$$p_{lb} = (\lfloor q_x \rfloor, \lceil q_y \rceil) \quad \text{(left bottom)} \tag{3}$$
$$p_{rb} = (\lceil q_x \rceil, \lceil q_y \rceil) \quad \text{(right bottom)} \tag{4}$$

.2

Problem 1.2

Complete the bilinear interpolation weight formula:

$$G(p, q) = (1 - |p_x - q_x|) \cdot (1 - |p_y - q_y|)$$

.3

Problem 1.3

Fill in the missing code for bilinear interpolation bounds checking:

```
def get_pixel_value(img, y, x):
    if 0 <= y < H and 0 <= x < W:
        return img[y, x]
    else:
        return ___  # What goes here?
```

Answer: `0.0` (zero padding for out-of-bounds)

.4

Problem 1.4

What is the correct order for bilinear interpolation calculation? **Memory Pattern:** "First X, then Y"

1. Get 4 corner values: $v_{00}, v_{01}, v_{10}, v_{11}$
2. Calculate fractional parts: $dx = q_x - x_0$, $dy = q_y - y_0$
3. Interpolate along X: $v_0 = v_{00}(1 - dx) + v_{01} \cdot dx$
4. Interpolate along X: $v_1 = v_{10}(1 - dx) + v_{11} \cdot dx$
5. Interpolate along Y: $out = v_0(1 - dy) + v_1 \cdot dy$

.5

Problem 1.5

In deformable convolution, how do you extract the y and x offsets from the delta tensor? **Critical Pattern - PyTorch stores Y first, then X:**

```
delta_y = delta[n, 2 * k, h_out, w_out]      # y offset
delta_x = delta[n, 2 * k + 1, h_out, w_out]  # x offset
```

.6

Problem 1.6

What is the deformable convolution sampling position formula?

```
sample_y = h_start + kh * dilation + _____
sample_x = w_start + kw * dilation + _____
```

Answer: `delta_y` and `delta_x`

TASK 2: CNN PYTORCH MEMORIZATION GUIDE

.1

Problem 2.1

What are the CIFAR-100 normalization values you must memorize? **Critical Constants:**

```
mean=[0.5071, 0.4867, 0.4408]  # CIFAR100 mean
std=[0.2675, 0.2565, 0.2761]   # CIFAR100 std
```

.2

Problem 2.2

Complete the data augmentation transforms for training:

```
transform_train = transforms.Compose([
    transforms.______(32, padding=4),  # What goes here?
    transforms.______(),               # What goes here?
    transforms.ToTensor(),
    transforms.Normalize(mean=[...], std=[...])
])
```

Answer: `RandomCrop` and `RandomHorizontalFlip`

.3

Problem 2.3

How do you split CIFAR-100 training data into 80/20 train/validation? **Pattern to Remember:**

```
train_size = int(0.8 * len(full_train_set))
val_size = len(full_train_set) - train_size
train_set, val_set = random_split(full_train_set, [train_size, val_size])
```

.4

Problem 2.4

What is the CNN architecture pattern for the CustomCNN class? **Layer Sequence Pattern:**

1. Conv2d(3, 32) → Conv2d(32, 64) → MaxPool2d
2. Conv2d(64, 128) → MaxPool2d
3. Conv2d(128, 256) → MaxPool2d
4. Flatten → FC(256*4*4, 512) → FC(512, 256) → FC(256, 100)

.5

Problem 2.5

Complete the forward pass activation pattern:

```
x = F.relu(self.conv1(x))
x = F.relu(self.conv2(x))
x = self.pool(x)  # 32x32 -> 16x16
x = F.relu(self.conv3(x))
x = self.pool(x)  # 16x16 -> 8x8
x = F.relu(self.conv4(x))
x = self.pool(x)  # 8x8 -> 4x4
x = x.view(x.size(0), -1)  # Flatten
x = F.relu(self.fc1(x))
x = self.dropout(x)
x = F.relu(self.fc2(x))
x = self.dropout(x)
x = self.fc3(x)  # No activation on final layer!
```

.6

Problem 2.6

What loss function and optimizer setup is standard for CIFAR-100?

```
loss_function = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(),
                      lr=0.01,
                      momentum=0.9,
                      weight_decay=5e-4)
```

.7

Problem 2.7

Complete the top-1 and top-5 accuracy calculation:

```
# Top-1 accuracy
_, top1_pred = outputs.topk(1, dim=1, largest=True, sorted=True)
top1_correct = top1_pred.eq(labels.view(-1, 1)).sum().item()

# Top-5 accuracy
_, top5_pred = outputs.topk(5, dim=1, largest=True, sorted=True)
top5_correct = top5_pred.eq(labels.view(-1, 1).____(____))).sum().item()
```

Answer: `expand_as(top5_pred)`

.8

Problem 2.8

What is the training loop structure pattern? **Memory Pattern - "Zero, Forward, Backward, Step":**

```
optimizer.zero_grad()    # Clear gradients
outputs = model(images)  # Forward pass
loss = loss_function(outputs, labels)  # Compute loss
loss.backward()          # Backward pass
optimizer.step()         # Update weights
```

.9

Problem 2.9

How do you add BatchNorm2d to the CNN architecture? **Pattern - After each Conv2d:**

```
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
self.bn1 = nn.BatchNorm2d(32)  # Same number as conv output
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
self.bn2 = nn.BatchNorm2d(64)  # Same number as conv output
```

TASK 3: RNN MEMORIZATION GUIDE

.1

Problem 3.1

How do you create character vocabulary and mappings? **Standard Pattern:**

```
chars = sorted(list(set(text)))
char2idx = {ch: i for i, ch in enumerate(chars)}
idx2char = {i: ch for i, ch in enumerate(chars)}
```

.2

Problem 3.2

How do you create input and target sequences for character prediction?

```
input_seq = text[:-1]  # All except last
target_seq = text[1:]  # All except first
```

.3

Problem 3.3

Complete the one-hot encoding function:

```
def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[___] = 1.0
    return vec
```

Answer: `idx`

.4

Problem 3.4

What are the RNN weight matrix dimensions? **Dimension Memory Pattern:**

```
W_xh = torch.randn(H, V, requires_grad=True) * 0.1  # (H, V)
W_hh = torch.randn(H, H, requires_grad=True) * 0.1  # (H, H)
b_xh = torch.zeros(H, requires_grad=True)           # (H,)
b_hh = torch.zeros(H, requires_grad=True)           # (H,)
W_hy = torch.randn(V, H, requires_grad=True) * 0.1  # (V, H)
b_y = torch.zeros(V, requires_grad=True)            # (V,)
```

.5

Problem 3.5

Complete the RNN forward pass equations:

```
# Hidden state update
h = torch.tanh(W_xh @ x_t + b_xh + W_hh @ h + b_hh)
```

```
# Output logits
s_t = _____ @ h + _____
```

Answer: W_hy and b_y

.6

Problem 3.6

How do you compute gradients explicitly with torch.autograd?

```
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=True)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=True)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=True)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=True)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=True)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=True)[0]
```

CRITICAL MEMORIZATION PATTERNS

Patterns

Problem Patterns

What are the key patterns you must memorize?

1. Bilinear Interpolation Pattern:

- Get 4 corners (floor/ceil combinations)
- Interpolate X first, then Y
- Use fractional parts: $dx = q_x - x_0$, $dy = q_y - y_0$

2. CNN Architecture Pattern:

- Conv-Conv-Pool, Conv-Pool, Conv-Pool structure
- Channel progression: 3→32→64→128→256
- Spatial reduction: 32→16→8→4
- FC layers: 4096→512→256→100

3. Training Loop Pattern:

- "Zero-Forward-Backward-Step"
- Always move tensors to device
- Use torch.no_grad() for validation

4. RNN Equations Pattern:

- $h_t = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$
- $s_t = W_{hy}h_t + b_y$
- Always remember matrix dimensions

5. Data Preprocessing Patterns:

- CIFAR-100: mean=[0.5071, 0.4867, 0.4408], std=[0.2675, 0.2565, 0.2761]
- 80/20 split: train_size = int(0.8 * len(dataset))
- Character sequences: input = text[:-1], target = text[1:]

COMMON MISTAKES TO AVOID

Mistakes

Problem Mistakes

What are the most common implementation mistakes?

1. Deformable CNN:

- Forget PyTorch stores Y offset first, then X offset
- Wrong bilinear interpolation order (do X first, then Y)
- Forget zero padding for out-of-bounds pixels

2. CNN:

- Forget to move tensors to device
- Wrong flatten calculation: x.view(x.size(0), -1)
- Forget activation on hidden layers, add activation on output layer

3. RNN:

- Wrong weight matrix dimensions
- Forget requires_grad=True for parameters
- Use retain_graph=True for multiple gradient computations

RAPID-FIRE ACTIVE RECALL QUIZ

Speed Round 1: Fill the Blanks

Problem Speed Round 1: Fill the Blanks

Complete these critical code snippets:

Q1: Bilinear interpolation corners:

```
y0 = int(np.______(q_y))
x0 = int(np.______(q_x))
y1 = y0 + ___
x1 = x0 + ___
```

Q2: Deformable conv offset extraction:

```
delta_y = delta[n, ___ * k, h_out, w_out]
delta_x = delta[n, ___ * k + ___, h_out, w_out]
```

Q3: CNN flatten operation:

```
x = x.view(x.____(__), ___)
```

Q4: Top-5 accuracy calculation:

```
_, top5_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top5_correct = top5_pred.eq(labels.view(-1, 1).______(_____)).sum().item()
```

Q5: RNN hidden state update:

```
h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)
```

Speed Round 2: True/False

Problem Speed Round 2: True/False

Mark T/F for these statements:

1. In bilinear interpolation, you interpolate Y direction first, then X direction. **[T/F]**
2. PyTorch stores Y offset before X offset in deformable convolution. **[T/F]**
3. CIFAR-100 has 100 classes, so the final FC layer outputs 100 values. **[T/F]**
4. You should apply ReLU activation to the final output layer in classification. **[T/F]**
5. In RNN, W_xh has dimensions (V, H). **[T/F]**
6. For validation, you need to call optimizer.zero_grad(). **[T/F]**
7. BatchNorm2d should be applied before the activation function. **[T/F]**
8. The input sequence for RNN is text[1:] and target is text[:-1]. **[T/F]**

Problem Speed Round 3: Memory Palace

Associate these concepts with memorable phrases:

Bilinear Interpolation: "Four corners, X then Y, fractional magic"

- 4 corners: lt, rt, lb, rb
- X interpolation: top_edge, bottom_edge
- Y interpolation: final result

CNN Architecture: "3 to 32, double-double-double, then shrink to 100"

- Channels: 3→32→64→128→256
- Spatial: 32→16→8→4
- FC: 4096→512→256→100

Training Loop: "Zero-Forward-Backward-Step dance"

- optimizer.zero_grad()
- outputs = model(images)
- loss.backward()
- optimizer.step()

RNN Weights: "Input-Hidden-Hidden, Hidden-Hidden-Hidden, Hidden-Vocab-Vocab"

- W_xh: (H, V) - maps input to hidden
- W_hh: (H, H) - maps previous hidden to current hidden
- W_hy: (V, H) - maps hidden to output vocabulary

LAST-MINUTE CHECKLIST

Pre-Exam Checklist

Problem Pre-Exam Checklist

Before the exam, ensure you can write from memory:

Critical Constants:

- CIFAR-100 mean: [0.5071, 0.4867, 0.4408]
- CIFAR-100 std: [0.2675, 0.2565, 0.2761]
- Train/val split: 0.8 * len(dataset)

Key Formulas:

- Bilinear weight: $(1 - |p_x - q_x|) \cdot (1 - |p_y - q_y|)$
- RNN hidden: $h_t = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$
- RNN output: $s_t = W_{hy}h_t + b_y$

Critical Code Patterns:

- Device transfer: `tensor.to(device)`
- Gradient computation: `torch.autograd.grad(loss, param, retain_graph=True)[0]`
- Top-k accuracy: `outputs.topk(k, dim=1, largest=True, sorted=True)`
- One-hot encoding: `vec[idx] = 1.0`

Architecture Patterns:

- CNN: Conv→BN→ReLU→Pool pattern
- RNN: Input→Hidden→Output with recurrence
- Training: Zero→Forward→Backward→Step

CODING BLOCKS MEMORIZATION

Code Block 1: Bilinear Interpolation Core

Problem Code Block 1: Bilinear Interpolation Core

Complete this bilinear interpolation function - focus on the mathematical pattern:

```
def bilinear_interpolate(a_l, q_y, q_x):
    H, W = a_l.shape

    # Step 1: Get integer positions
    y0 = int(np.______(q_y))
    x0 = int(np.______(q_x))
    y1 = y0 + ___
    x1 = x0 + ___

    # Step 2: Get values with bounds checking
    def get_pixel_value(img, y, x):
        if 0 <= y < H and 0 <= x < W:
            return img[y, x]
        else:
            return ____  # Out of bounds value

    # Step 3: Get four corner values
    v_00 = get_pixel_value(a_l, y0, x0)  # ___-___
    v_01 = get_pixel_value(a_l, y0, x1)  # ___-___
    v_10 = get_pixel_value(a_l, y1, x0)  # ___-___
    v_11 = get_pixel_value(a_l, y1, x1)  # ___-___

    # Step 4: Calculate fractional parts
    dy = q_y - ___
    dx = q_x - ___

    # Step 5: Interpolate X first, then Y
    v_0 = v_00 * (1 - ___) + v_01 * ___  # top edge
    v_1 = v_10 * (1 - ___) + v_11 * ___  # bottom edge
    out = v_0 * (1 - ___) + v_1 * ___    # final Y interpolation

    return out
```

Code Block 2: Deformable Conv Key Loop

Problem Code Block 2: Deformable Conv Key Loop

This is the heart of deformable convolution - memorize the offset extraction pattern:

```
for n in range(N):  # batch
    for c_out in range(C_out):  # output channels
        for h_out in range(H_out):  # height
            for w_out in range(W_out):  # width
                h_start = h_out * ____
                w_start = w_out * ____
                value = 0.0

                for kh in range(K_h):
                    for kw in range(K_w):
```

```
                    k = kh * K_w + kw

                    # CRITICAL: PyTorch offset order
                    delta_y = delta[n, ___ * k, h_out, w_out]
                    delta_x = delta[n, ___ * k + ___, h_out, w_out]
                    m_k = mask[n, k, h_out, w_out]

                    # Sampling position
                    sample_y = h_start + kh * dilation + ____
                    sample_x = w_start + kw * dilation + ____

                    for c_in in range(C_in):
                        interpolated = bilinear_interpolate(
                            a_l[n, c_in, :, :], sample_y, sample_x
                        )
                        value += weight[c_out, c_in, kh, kw] * ___ * ____

                out[n, c_out, h_out, w_out] = value
```

Code Block 3: CNN Architecture Constructor

Problem Code Block 3: CNN Architecture Constructor

Memorize the channel progression and layer naming pattern:

```
class CustomCNN(nn.Module):
    def __init__(self, norm_layer=None):
        super(CustomCNN, self).__init__()

        # Conv layers - memorize the channel progression
        self.conv1 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(___, ___, kernel_size=3, padding=1)

        # Pooling layer
        self.pool = nn.______(2, 2)

        # FC layers - calculate the input size
        self.fc1 = nn.Linear(___ * ___ * ___, 512)
        self.fc2 = nn.Linear(___, ___)
        self.fc3 = nn.Linear(___, ___)  # CIFAR-100 classes

        self.dropout = nn.Dropout(____)
```

Code Block 4: CNN Forward Pass Pattern

Problem Code Block 4: CNN Forward Pass Pattern

Memorize the activation and pooling pattern:

```
def forward(self, x):
    # Block 1: Conv-Conv-Pool
    x = F.____(self.conv1(x))
    x = F.____(self.conv2(x))
    x = self.pool(x)  # 32x32 -> ____
```

```
    # Block 2: Conv-Pool
    x = F.____(self.conv3(x))
    x = self.pool(x)  # 16x16 -> ____

    # Block 3: Conv-Pool
    x = F.____(self.conv4(x))
    x = self.pool(x)  # 8x8 -> ____

    # Flatten
    x = x.view(x.____(__), ___)

    # FC layers with dropout
    x = F.____(self.fc1(x))
    x = self.______(x)
    x = F.____(self.fc2(x))
    x = self.______(x)
    x = self.fc3(x)  # No activation here!

    return x
```

Code Block 5: Training Loop Core

Problem Code Block 5: Training Loop Core

The sacred training loop pattern - memorize the order:

```
def train(model, train_loader, optimizer, loss_function, device):
    model.____()  # Set to training mode

    for batch in train_loader:
        images, labels = batch
        images = images.to(____)
        labels = labels.to(____)

        # The sacred four steps:
        optimizer.____()         # Step 1: Clear gradients
        outputs = model(______)  # Step 2: Forward pass
        loss = loss_function(outputs, labels)  # Step 3: Compute loss
        loss.______()            # Step 4: Backward pass
        optimizer.____()         # Step 5: Update weights

        # Accuracy calculation
        _, top1_pred = outputs.topk(___, dim=1, largest=True, sorted=True)
        top1_correct = top1_pred.eq(labels.view(___, ___)).sum().item()
```

Code Block 6: BatchNorm CNN Constructor

Problem Code Block 6: BatchNorm CNN Constructor

Pattern for adding BatchNorm after each conv layer:

```
class CustomCNNwithBN(nn.Module):
    def __init__(self, norm_layer=None):
        super(CustomCNNwithBN, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(____)  # Same as conv1 output
```

```
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(____)  # Same as conv2 output

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(____)  # Same as conv3 output

        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(____)  # Same as conv4 output

    def forward(self, x):
        # Pattern: Conv -> BN -> ReLU
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = self.pool(x)

        x = F.relu(self.bn4(self.conv4(x)))
        x = self.pool(x)
        # ... rest of forward pass
```

Code Block 7: RNN Parameter Initialization

Problem Code Block 7: RNN Parameter Initialization

Memorize the weight dimensions and initialization pattern:

```
# RNN parameters - memorize the dimensions!
W_xh = torch.randn(___, ___, requires_grad=True) * 0.1  # Input to hidden
W_hh = torch.randn(___, ___, requires_grad=True) * 0.1  # Hidden to hidden
b_xh = torch.zeros(___, requires_grad=True)             # Hidden bias 1
b_hh = torch.zeros(___, requires_grad=True)             # Hidden bias 2
W_hy = torch.randn(___, ___, requires_grad=True) * 0.1  # Hidden to output
b_y = torch.zeros(___, requires_grad=True)              # Output bias

# Remember: V = vocab size, H = hidden size
```

Code Block 8: RNN Forward Pass

Problem Code Block 8: RNN Forward Pass

The RNN equations in code form:

```
logits_list = []
h = torch.zeros(H)

for t in range(seq_len):
    x_t = inputs[t]

    # Hidden state update equation
    h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)

    # Output logits equation
    s_t = _____ @ h + _____
```

```
    logits_list.append(s_t)

logits = torch.stack(logits_list)
log_probs = F.log_softmax(logits, dim=1)
loss_manual = F.nll_loss(log_probs, targets)
```

Code Block 9: Gradient Computation

Problem Code Block 9: Gradient Computation

Pattern for explicit gradient computation:

```
# Compute gradients explicitly
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=____)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=____)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=____)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=____)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=____)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=____)[0]

# Why retain_graph=True? Because we compute multiple gradients from same loss
```

Code Block 10: Character Processing

Problem Code Block 10: Character Processing

Standard pattern for character-level RNN preprocessing:

```
text = "Deep Learning"

# Step 1: Create vocabulary
chars = sorted(list(set(____)))
char2idx = {ch: i for i, ch in enumerate(____)}
idx2char = {i: ch for i, ch in enumerate(____)}

# Step 2: Create sequences
input_seq = text[____]   # All except last
target_seq = text[____]  # All except first

# Step 3: Convert to tensors
inputs = [one_hot(char2idx[ch], V) for ch in ____]
targets = torch.tensor([char2idx[ch] for ch in ____], dtype=torch.long)

def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[____] = 1.0
    return vec
```

MEMORIZATION MNEMONICS

Memory Aids

Problem Memory Aids

Use these phrases to remember key patterns:

"Floor-Ceil-Four-Corners": Bilinear interpolation corners

- lt: floor-floor, rt: ceil-floor, lb: floor-ceil, rb: ceil-ceil

"Y-before-X-in-PyTorch": Deformable conv offset ordering

- delta_y = delta[n, 2*k, h_out, w_out]
- delta_x = delta[n, 2*k+1, h_out, w_out]

"3-32-64-128-256": CNN channel progression

- Each layer doubles the channels (except first)

"32-16-8-4": Spatial dimension reduction

- Each MaxPool2d(2,2) halves the spatial dimensions

"Zero-Forward-Backward-Step": Training loop mantra

- Never forget the order!

"Input-Hidden-Hidden": RNN weight dimensions

- W_xh: (H,V), W_hh: (H,H), W_hy: (V,H)

"Tanh-Hidden-Linear-Output": RNN computation flow

- Hidden uses tanh, output is linear

ANSWERS TO CODING BLOCKS

Code Block 1: floor, floor, 1, 1, 0.0, top-left, top-right, bottom-left, bottom-right, y0, x0, dx, dx, dx, dx, dy, dy

Code Block 2: stride, stride, 2, 2, 1, delta_y, delta_x, m_k, interpolated

Code Block 3: 3, 32, 32, 64, 64, 128, 128, 256, MaxPool2d, 256, 4, 4, 512, 256, 256, 100, 0.5

Code Block 4: relu, relu, 16x16, relu, 8x8, relu, 4x4, size(0), -1, relu, dropout, relu, dropout

Code Block 5: train, device, device, zero_grad, images, backward, step, 1, -1, 1

Code Block 6: 32, 64, 128, 256

Code Block 7: H, V, H, H, H, H, V, H, V

Code Block 8: W_xh, b_xh, W_hh, b_hh, W_hy, b_y

Code Block 9: True, True, True, True, True, True

ADVANCED CODING SCENARIOS

Problem 1

Scenario 1: Debugging Deformable Conv If your deformable convolution gives wrong results, what are the most likely bugs?

```
# Common Bug 1: Wrong offset extraction
delta_y = delta[n, k, h_out, w_out]        # WRONG - missing factor of 2
delta_x = delta[n, k + 1, h_out, w_out]    # WRONG - should be 2*k+1

# Correct version:
delta_y = delta[n, ____ * k, h_ut, w_out]
delta_x = delta[n, ____ * k + ____, h_out, w_out]

# Common Bug 2: Wrong bilinear interpolation order
# WRONG: Interpolate Y first
v_y = v_00 * (1 - dy) + v_10 * dy
v_final = v_y * (1 - dx) + v_01 * dx

# Correct: Interpolate X first, then Y
v_0 = v_00 * (1 - ____) + v_01 * ____  # top edge
v_1 = v_10 * (1 - ____) + v_11 * ____  # bottom edge
out = v_0 * (1 - ____) + v_1 * ____    # Y interpolation
```

Scenario 2: CNN Architecture Variations

Problem Scenario 2: CNN Architecture Variations

If asked to modify the CNN, remember these patterns:

```
# Adding more conv layers - maintain the pattern
class CustomCNNDeep(nn.Module):
    def __init__(self):
        super().__init__()
        # Pattern: start with 3 channels, double each time
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)  # Same channels
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)  # Double
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)  # Same
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1) # Double

    def forward(self, x):
        # Pattern: Conv-Conv-Pool, Conv-Conv-Pool
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)  # After every 2 conv layers

        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)

        # Calculate new flatten size: 128 * 8 * 8 = ____
```

Scenario 3: Validation vs Training Mode

Problem Scenario 3: Validation vs Training Mode

Critical differences between training and validation:

```
# Training mode
def train_epoch():
    model.____()  # Enable dropout and batch norm training mode

    for batch in train_loader:
        optimizer.____()  # Clear gradients
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.____()       # Compute gradients
        optimizer.____()  # Update weights

# Validation mode
def validate():
    model.____()  # Disable dropout, batch norm in eval mode

    with torch.____():  # Disable gradient computation
        for batch in val_loader:
            # NO optimizer.zero_grad() here!
            # NO loss.backward() here!
            # NO optimizer.step() here!
            outputs = model(images)
            loss = criterion(outputs, labels)
```

Scenario 4: RNN with Different Sequence Lengths

Problem Scenario 4: RNN with Different Sequence Lengths

If given a different text, adapt the RNN code:

```
# Original: "Deep Learning"
text = "Deep Learning"
input_seq = text[:-1]   # "Deep Learnin"
target_seq = text[1:]   # "eep Learning"

# New text: "Hello World"
text = "Hello World"
input_seq = text[____]  # "Hello Worl"
target_seq = text[____] # "ello World"

# Vocabulary size changes!
chars = sorted(list(set(text)))
V = len(chars)  # This will be different!

# All weight matrices need to be reinitialized with new V
W_xh = torch.randn(H, ____, requires_grad=True) * 0.1
W_hy = torch.randn(____, H, requires_grad=True) * 0.1
b_y = torch.zeros(____, requires_grad=True)
```

Scenario 5: Hyperparameter Grid Search Pattern

Problem Scenario 5: Hyperparameter Grid Search Pattern

Standard grid search implementation:

```
learning_rates = [0.0001, 0.001]
```

```
optimizers = [torch.optim.Adam, torch.optim.SGD]
model_classes = [CustomCNN, CustomCNNwithBN]

best_accuracy = 0
best_params = None

for model_class in model_classes:
    for optimizer_class in optimizers:
        for lr in learning_rates:
            # CRITICAL: Reinitialize model each time
            model = model_class().to(device)

            # Initialize optimizer based on type
            if optimizer_class == torch.optim.SGD:
                optimizer = optimizer_class(
                    model.parameters(),
                    lr=lr,
                    momentum=____,
                    weight_decay=____
                )
            else:  # Adam
                optimizer = optimizer_class(
                    model.parameters(),
                    lr=lr,
                    weight_decay=____
                )

            # Train and validate...
            val_acc = train_and_validate(model, optimizer)

            if val_acc > best_accuracy:
                best_accuracy = val_acc
                best_params = (model_class.__name__, optimizer_class.__name__, lr)
```

Scenario 6: Top-K Accuracy Calculation Variations

Problem Scenario 6: Top-K Accuracy Calculation Variations

Different ways to calculate accuracy:

```
# Top-1 accuracy (most common)
_, predicted = torch.max(outputs, 1)
correct = (predicted == labels).sum().item()
accuracy = correct / labels.size(0) * 100

# Top-K accuracy using topk
_, top_k_pred = outputs.topk(____, dim=1, largest=True, sorted=True)
top_k_correct = top_k_pred.eq(labels.view(-1, 1).expand_as(____)).sum().item()

# Alternative top-K calculation
values, indices = torch.topk(outputs, k=5, dim=1)
correct_mask = indices == labels.unsqueeze(1)
top_k_accuracy = correct_mask.sum().float() / labels.size(0) * 100
```

EXAM SIMULATION QUESTIONS

Quick Code Writing

Problem Quick Code Writing

Write these functions from memory in 2 minutes each:

1. Write the bilinear interpolation weight calculation:

```
def calculate_bilinear_weight(p_x, p_y, q_x, q_y):
    # Your code here - calculate G(p,q)
    return ____
```

2. Write the RNN hidden state update:

```
def rnn_step(x_t, h_prev, W_xh, W_hh, b_xh, b_hh):
    # Your code here - compute new hidden state
    return ____
```

3. Write the CNN forward pass for one block:

```
def cnn_block_forward(x, conv1, conv2, pool):
    # Your code here - conv-conv-pool pattern
    return ____
```

4. Write the training step:

```
def training_step(model, optimizer, criterion, images, labels):
    # Your code here - complete training step
    return loss
```

5. Write character to one-hot conversion:

```
def char_to_onehot(char, char2idx, vocab_size):
    # Your code here - convert character to one-hot vector
    return ____
```

FINAL MEMORY CHECK

Problem 1

Last Minute Review Before the exam, quickly verify you remember:

Constants (write from memory):

- CIFAR-100 mean: [____, ____, ____]
- CIFAR-100 std: [____, ____, ____]
- Train/val split ratio: ____

- Dropout probability: ____
- Weight decay: ____
- SGD momentum: ____

Dimensions (write from memory):

- CNN channels: 3→____→____→____→____
- CNN spatial: 32→____→____→____
- RNN W_xh: (____, ____)
- RNN W_hh: (____, ____)
- RNN W_hy: (____, ____)

Key Equations (write from memory):

- Bilinear weight: G(p,q) = __________
- RNN hidden: h_t = __________
- RNN output: s_t = __________

CENG403 - Spring 2025: Homework set THE-2 Study Guide

Your Name

TASK 1: DEFORMABLE CNN MEMORIZATION GUIDE

.1

Problem 1.1

What are the 4 corner positions for bilinear interpolation given fractional position $q = (q_x, q_y)$? **Pattern to Remember:** Floor-Floor, Ceil-Floor, Floor-Ceil, Ceil-Ceil

$$p_{lt} = (\lfloor q_x \rfloor, \lfloor q_y \rfloor) \quad \text{(left top)} \tag{1}$$
$$p_{rt} = (\lceil q_x \rceil, \lfloor q_y \rfloor) \quad \text{(right top)} \tag{2}$$
$$p_{lb} = (\lfloor q_x \rfloor, \lceil q_y \rceil) \quad \text{(left bottom)} \tag{3}$$
$$p_{rb} = (\lceil q_x \rceil, \lceil q_y \rceil) \quad \text{(right bottom)} \tag{4}$$

.2

Problem 1.2

Complete the bilinear interpolation weight formula:

$$G(p, q) = (1 - |p_x - q_x|) \cdot (1 - |p_y - q_y|)$$

.3

Problem 1.3

Fill in the missing code for bilinear interpolation bounds checking:

```
def get_pixel_value(img, y, x):
    if 0 <= y < H and 0 <= x < W:
        return img[y, x]
    else:
        return ___  # What goes here?
```

Answer: `0.0` (zero padding for out-of-bounds)

.4

Problem 1.4

What is the correct order for bilinear interpolation calculation? **Memory Pattern:** "First X, then Y"

1. Get 4 corner values: $v_{00}, v_{01}, v_{10}, v_{11}$
2. Calculate fractional parts: $dx = q_x - x_0$, $dy = q_y - y_0$
3. Interpolate along X: $v_0 = v_{00}(1 - dx) + v_{01} \cdot dx$
4. Interpolate along X: $v_1 = v_{10}(1 - dx) + v_{11} \cdot dx$
5. Interpolate along Y: $out = v_0(1 - dy) + v_1 \cdot dy$

.5

Problem 1.5

In deformable convolution, how do you extract the y and x offsets from the delta tensor? **Critical Pattern - PyTorch stores Y first, then X:**

```
delta_y = delta[n, 2 * k, h_out, w_out]      # y offset
delta_x = delta[n, 2 * k + 1, h_out, w_out]  # x offset
```

.6

Problem 1.6

What is the deformable convolution sampling position formula?

```
sample_y = h_start + kh * dilation + _____
sample_x = w_start + kw * dilation + _____
```

Answer: `delta_y` and `delta_x`

TASK 2: CNN PYTORCH MEMORIZATION GUIDE

.1

Problem 2.1

What are the CIFAR-100 normalization values you must memorize? **Critical Constants:**

```
mean=[0.5071, 0.4867, 0.4408]  # CIFAR100 mean
std=[0.2675, 0.2565, 0.2761]   # CIFAR100 std
```

.2

Problem 2.2

Complete the data augmentation transforms for training:

```
transform_train = transforms.Compose([
    transforms.______(32, padding=4),  # What goes here?
    transforms.______(),               # What goes here?
    transforms.ToTensor(),
    transforms.Normalize(mean=[...], std=[...])
])
```

Answer: `RandomCrop` and `RandomHorizontalFlip`

.3

Problem 2.3

How do you split CIFAR-100 training data into 80/20 train/validation? **Pattern to Remember:**

```
train_size = int(0.8 * len(full_train_set))
val_size = len(full_train_set) - train_size
train_set, val_set = random_split(full_train_set, [train_size, val_size])
```

.4

Problem 2.4

What is the CNN architecture pattern for the CustomCNN class? **Layer Sequence Pattern:**

1. Conv2d(3, 32) → Conv2d(32, 64) → MaxPool2d
2. Conv2d(64, 128) → MaxPool2d
3. Conv2d(128, 256) → MaxPool2d
4. Flatten → FC(256*4*4, 512) → FC(512, 256) → FC(256, 100)

.5

Problem 2.5

Complete the forward pass activation pattern:

```
x = F.relu(self.conv1(x))
x = F.relu(self.conv2(x))
x = self.pool(x)  # 32x32 -> 16x16
x = F.relu(self.conv3(x))
x = self.pool(x)  # 16x16 -> 8x8
x = F.relu(self.conv4(x))
x = self.pool(x)  # 8x8 -> 4x4
x = x.view(x.size(0), -1)  # Flatten
x = F.relu(self.fc1(x))
x = self.dropout(x)
x = F.relu(self.fc2(x))
x = self.dropout(x)
x = self.fc3(x)  # No activation on final layer!
```

.6

Problem 2.6

What loss function and optimizer setup is standard for CIFAR-100?

```
loss_function = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(),
                      lr=0.01,
                      momentum=0.9,
                      weight_decay=5e-4)
```

.7

Problem 2.7

Complete the top-1 and top-5 accuracy calculation:

```
# Top-1 accuracy
_, top1_pred = outputs.topk(1, dim=1, largest=True, sorted=True)
top1_correct = top1_pred.eq(labels.view(-1, 1)).sum().item()

# Top-5 accuracy
_, top5_pred = outputs.topk(5, dim=1, largest=True, sorted=True)
top5_correct = top5_pred.eq(labels.view(-1, 1).____(____))).sum().item()
```

Answer: `expand_as(top5_pred)`

.8

Problem 2.8

What is the training loop structure pattern? **Memory Pattern - "Zero, Forward, Backward, Step":**

```
optimizer.zero_grad()    # Clear gradients
outputs = model(images)  # Forward pass
loss = loss_function(outputs, labels)  # Compute loss
loss.backward()          # Backward pass
optimizer.step()         # Update weights
```

.9

Problem 2.9

How do you add BatchNorm2d to the CNN architecture? **Pattern - After each Conv2d:**

```
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
self.bn1 = nn.BatchNorm2d(32)  # Same number as conv output
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
self.bn2 = nn.BatchNorm2d(64)  # Same number as conv output
```

TASK 3: RNN MEMORIZATION GUIDE

.1

Problem 3.1

How do you create character vocabulary and mappings? **Standard Pattern:**

```
chars = sorted(list(set(text)))
char2idx = {ch: i for i, ch in enumerate(chars)}
idx2char = {i: ch for i, ch in enumerate(chars)}
```

.2

Problem 3.2

How do you create input and target sequences for character prediction?

```
input_seq = text[:-1]  # All except last
target_seq = text[1:]  # All except first
```

.3

Problem 3.3

Complete the one-hot encoding function:

```
def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[___] = 1.0
    return vec
```

Answer: `idx`

.4

Problem 3.4

What are the RNN weight matrix dimensions? **Dimension Memory Pattern:**

```
W_xh = torch.randn(H, V, requires_grad=True) * 0.1  # (H, V)
W_hh = torch.randn(H, H, requires_grad=True) * 0.1  # (H, H)
b_xh = torch.zeros(H, requires_grad=True)           # (H,)
b_hh = torch.zeros(H, requires_grad=True)           # (H,)
W_hy = torch.randn(V, H, requires_grad=True) * 0.1  # (V, H)
b_y = torch.zeros(V, requires_grad=True)            # (V,)
```

.5

Problem 3.5

Complete the RNN forward pass equations:

```
# Hidden state update
h = torch.tanh(W_xh @ x_t + b_xh + W_hh @ h + b_hh)
```

```
# Output logits
s_t = _____ @ h + _____
```

Answer: `W_hy` and `b_y`

.6

Problem 3.6

How do you compute gradients explicitly with torch.autograd?

```
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=True)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=True)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=True)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=True)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=True)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=True)[0]
```

CRITICAL MEMORIZATION PATTERNS

Patterns

Problem Patterns

What are the key patterns you must memorize?

1. Bilinear Interpolation Pattern:

- Get 4 corners (floor/ceil combinations)
- Interpolate X first, then Y
- Use fractional parts: $dx = q_x - x_0$, $dy = q_y - y_0$

2. CNN Architecture Pattern:

- Conv-Conv-Pool, Conv-Pool, Conv-Pool structure
- Channel progression: 3→32→64→128→256
- Spatial reduction: 32→16→8→4
- FC layers: 4096→512→256→100

3. Training Loop Pattern:

- "Zero-Forward-Backward-Step"
- Always move tensors to device
- Use torch.no_grad() for validation

4. RNN Equations Pattern:

- $h_t = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$
- $s_t = W_{hy}h_t + b_y$
- Always remember matrix dimensions

5. Data Preprocessing Patterns:

- CIFAR-100: mean=[0.5071, 0.4867, 0.4408], std=[0.2675, 0.2565, 0.2761]
- 80/20 split: train_size = int(0.8 * len(dataset))
- Character sequences: input = text[:-1], target = text[1:]

COMMON MISTAKES TO AVOID

Mistakes

Problem Mistakes

What are the most common implementation mistakes?

1. Deformable CNN:

- Forget PyTorch stores Y offset first, then X offset
- Wrong bilinear interpolation order (do X first, then Y)
- Forget zero padding for out-of-bounds pixels

2. CNN:

- Forget to move tensors to device
- Wrong flatten calculation: x.view(x.size(0), -1)
- Forget activation on hidden layers, add activation on output layer

3. RNN:

- Wrong weight matrix dimensions
- Forget requires_grad=True for parameters
- Use retain_graph=True for multiple gradient computations

RAPID-FIRE ACTIVE RECALL QUIZ

Speed Round 1: Fill the Blanks

Problem Speed Round 1: Fill the Blanks

Complete these critical code snippets:

Q1: Bilinear interpolation corners:

```
y0 = int(np.______(q_y))
x0 = int(np.______(q_x))
y1 = y0 + ___
x1 = x0 + ___
```

Q2: Deformable conv offset extraction:

```
delta_y = delta[n, ___ * k, h_out, w_out]
delta_x = delta[n, ___ * k + ___, h_out, w_out]
```

Q3: CNN flatten operation:

```
x = x.view(x.____(__), ___)
```

Q4: Top-5 accuracy calculation:

```
_, top5_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top5_correct = top5_pred.eq(labels.view(-1, 1).______(_____)).sum().item()
```

Q5: RNN hidden state update:

```
h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)
```

Speed Round 2: True/False

Problem Speed Round 2: True/False

Mark T/F for these statements:

1. In bilinear interpolation, you interpolate Y direction first, then X direction. **[T/F]**
2. PyTorch stores Y offset before X offset in deformable convolution. **[T/F]**
3. CIFAR-100 has 100 classes, so the final FC layer outputs 100 values. **[T/F]**
4. You should apply ReLU activation to the final output layer in classification. **[T/F]**
5. In RNN, W_xh has dimensions (V, H). **[T/F]**
6. For validation, you need to call optimizer.zero_grad(). **[T/F]**
7. BatchNorm2d should be applied before the activation function. **[T/F]**
8. The input sequence for RNN is text[1:] and target is text[:-1]. **[T/F]**

Speed Round 3: Memory Palace

Problem Speed Round 3: Memory Palace

Associate these concepts with memorable phrases:

Bilinear Interpolation: "Four corners, X then Y, fractional magic"

- 4 corners: lt, rt, lb, rb
- X interpolation: top_edge, bottom_edge
- Y interpolation: final result

CNN Architecture: "3 to 32, double-double-double, then shrink to 100"

- Channels: 3→32→64→128→256
- Spatial: 32→16→8→4
- FC: 4096→512→256→100

Training Loop: "Zero-Forward-Backward-Step dance"

- optimizer.zero_grad()
- outputs = model(images)
- loss.backward()
- optimizer.step()

RNN Weights: "Input-Hidden-Hidden, Hidden-Hidden-Hidden, Hidden-Vocab-Vocab"

- W_xh: (H, V) - maps input to hidden
- W_hh: (H, H) - maps previous hidden to current hidden
- W_hy: (V, H) - maps hidden to output vocabulary

LAST-MINUTE CHECKLIST

Pre-Exam Checklist

Problem Pre-Exam Checklist

Before the exam, ensure you can write from memory:

Critical Constants:

- CIFAR-100 mean: [0.5071, 0.4867, 0.4408]
- CIFAR-100 std: [0.2675, 0.2565, 0.2761]
- Train/val split: 0.8 * len(dataset)

Key Formulas:

- Bilinear weight: $(1 - |p_x - q_x|) \cdot (1 - |p_y - q_y|)$
- RNN hidden: $h_t = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$
- RNN output: $s_t = W_{hy}h_t + b_y$

Critical Code Patterns:

- Device transfer: `tensor.to(device)`
- Gradient computation: `torch.autograd.grad(loss, param, retain_graph=True)[0]`
- Top-k accuracy: `outputs.topk(k, dim=1, largest=True, sorted=True)`
- One-hot encoding: `vec[idx] = 1.0`

Architecture Patterns:

- CNN: Conv→BN→ReLU→Pool pattern
- RNN: Input→Hidden→Output with recurrence
- Training: Zero→Forward→Backward→Step

CODING BLOCKS MEMORIZATION

Code Block 1: Bilinear Interpolation Core

Problem Code Block 1: Bilinear Interpolation Core

Complete this bilinear interpolation function - focus on the mathematical pattern:

```
def bilinear_interpolate(a_l, q_y, q_x):
    H, W = a_l.shape

    # Step 1: Get integer positions
    y0 = int(np.______(q_y))
    x0 = int(np.______(q_x))
    y1 = y0 + ___
    x1 = x0 + ___

    # Step 2: Get values with bounds checking
    def get_pixel_value(img, y, x):
        if 0 <= y < H and 0 <= x < W:
            return img[y, x]
        else:
            return ____  # Out of bounds value

    # Step 3: Get four corner values
    v_00 = get_pixel_value(a_l, y0, x0)  # ___-___
    v_01 = get_pixel_value(a_l, y0, x1)  # ___-___
    v_10 = get_pixel_value(a_l, y1, x0)  # ___-___
    v_11 = get_pixel_value(a_l, y1, x1)  # ___-___

    # Step 4: Calculate fractional parts
    dy = q_y - ___
    dx = q_x - ___

    # Step 5: Interpolate X first, then Y
    v_0 = v_00 * (1 - ___) + v_01 * ___  # top edge
    v_1 = v_10 * (1 - ___) + v_11 * ___  # bottom edge
    out = v_0 * (1 - ___) + v_1 * ___    # final Y interpolation

    return out
```

Code Block 2: Deformable Conv Key Loop

Problem Code Block 2: Deformable Conv Key Loop

This is the heart of deformable convolution - memorize the offset extraction pattern:

```
for n in range(N):  # batch
    for c_out in range(C_out):  # output channels
        for h_out in range(H_out):  # height
            for w_out in range(W_out):  # width
                h_start = h_out * ____
                w_start = w_out * ____
                value = 0.0

                for kh in range(K_h):
                    for kw in range(K_w):
```

```
                        k = kh * K_w + kw

                        # CRITICAL: PyTorch offset order
                        delta_y = delta[n, ___ * k, h_out, w_out]
                        delta_x = delta[n, ___ * k + ___, h_out, w_out]
                        m_k = mask[n, k, h_out, w_out]

                        # Sampling position
                        sample_y = h_start + kh * dilation + ____
                        sample_x = w_start + kw * dilation + ____

                        for c_in in range(C_in):
                            interpolated = bilinear_interpolate(
                                a_l[n, c_in, :, :], sample_y, sample_x
                            )
                            value += weight[c_out, c_in, kh, kw] * ___ * ____

                out[n, c_out, h_out, w_out] = value
```

Code Block 3: CNN Architecture Constructor

Problem Code Block 3: CNN Architecture Constructor

Memorize the channel progression and layer naming pattern:

```
class CustomCNN(nn.Module):
    def __init__(self, norm_layer=None):
        super(CustomCNN, self).__init__()

        # Conv layers - memorize the channel progression
        self.conv1 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(___, ___, kernel_size=3, padding=1)

        # Pooling layer
        self.pool = nn.______(2, 2)

        # FC layers - calculate the input size
        self.fc1 = nn.Linear(___ * ___ * ___, 512)
        self.fc2 = nn.Linear(___, ___)
        self.fc3 = nn.Linear(___, ___)  # CIFAR-100 classes

        self.dropout = nn.Dropout(____)
```

Code Block 4: CNN Forward Pass Pattern

Problem Code Block 4: CNN Forward Pass Pattern

Memorize the activation and pooling pattern:

```
def forward(self, x):
    # Block 1: Conv-Conv-Pool
    x = F.____(self.conv1(x))
    x = F.____(self.conv2(x))
    x = self.pool(x)  # 32x32 -> ____
```

```
    # Block 2: Conv-Pool
    x = F.____(self.conv3(x))
    x = self.pool(x)  # 16x16 -> ____

    # Block 3: Conv-Pool
    x = F.____(self.conv4(x))
    x = self.pool(x)  # 8x8 -> ____

    # Flatten
    x = x.view(x.____(__), ___)

    # FC layers with dropout
    x = F.____(self.fc1(x))
    x = self.______(x)
    x = F.____(self.fc2(x))
    x = self.______(x)
    x = self.fc3(x)  # No activation here!

    return x
```

Code Block 5: Training Loop Core

Problem Code Block 5: Training Loop Core

The sacred training loop pattern - memorize the order:

```
def train(model, train_loader, optimizer, loss_function, device):
    model.____()  # Set to training mode

    for batch in train_loader:
        images, labels = batch
        images = images.to(____)
        labels = labels.to(____)

        # The sacred four steps:
        optimizer.____()         # Step 1: Clear gradients
        outputs = model(______)  # Step 2: Forward pass
        loss = loss_function(outputs, labels)  # Step 3: Compute loss
        loss.______()            # Step 4: Backward pass
        optimizer.____()         # Step 5: Update weights

        # Accuracy calculation
        _, top1_pred = outputs.topk(___, dim=1, largest=True, sorted=True)
        top1_correct = top1_pred.eq(labels.view(___, ___)).sum().item()
```

Code Block 6: BatchNorm CNN Constructor

Problem Code Block 6: BatchNorm CNN Constructor

Pattern for adding BatchNorm after each conv layer:

```
class CustomCNNwithBN(nn.Module):
    def __init__(self, norm_layer=None):
        super(CustomCNNwithBN, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(____)  # Same as conv1 output
```

```
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(____)  # Same as conv2 output

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(____)  # Same as conv3 output

        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(____)  # Same as conv4 output

    def forward(self, x):
        # Pattern: Conv -> BN -> ReLU
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = self.pool(x)

        x = F.relu(self.bn4(self.conv4(x)))
        x = self.pool(x)
        # ... rest of forward pass
```

Code Block 7: RNN Parameter Initialization

Problem Code Block 7: RNN Parameter Initialization

Memorize the weight dimensions and initialization pattern:

```
# RNN parameters - memorize the dimensions!
W_xh = torch.randn(___, ___, requires_grad=True) * 0.1  # Input to hidden
W_hh = torch.randn(___, ___, requires_grad=True) * 0.1  # Hidden to hidden
b_xh = torch.zeros(___, requires_grad=True)             # Hidden bias 1
b_hh = torch.zeros(___, requires_grad=True)             # Hidden bias 2
W_hy = torch.randn(___, ___, requires_grad=True) * 0.1  # Hidden to output
b_y = torch.zeros(___, requires_grad=True)              # Output bias

# Remember: V = vocab size, H = hidden size
```

Code Block 8: RNN Forward Pass

Problem Code Block 8: RNN Forward Pass

The RNN equations in code form:

```
logits_list = []
h = torch.zeros(H)

for t in range(seq_len):
    x_t = inputs[t]

    # Hidden state update equation
    h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)

    # Output logits equation
    s_t = _____ @ h + _____
```

```
    logits_list.append(s_t)

logits = torch.stack(logits_list)
log_probs = F.log_softmax(logits, dim=1)
loss_manual = F.nll_loss(log_probs, targets)
```

Code Block 9: Gradient Computation

Problem Code Block 9: Gradient Computation

Pattern for explicit gradient computation:

```
# Compute gradients explicitly
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=____)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=____)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=____)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=____)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=____)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=____)[0]

# Why retain_graph=True? Because we compute multiple gradients from same loss
```

Code Block 10: Character Processing

Problem Code Block 10: Character Processing

Standard pattern for character-level RNN preprocessing:

```
text = "Deep Learning"

# Step 1: Create vocabulary
chars = sorted(list(set(____)))
char2idx = {ch: i for i, ch in enumerate(____)}
idx2char = {i: ch for i, ch in enumerate(____)}

# Step 2: Create sequences
input_seq = text[____]   # All except last
target_seq = text[____]  # All except first

# Step 3: Convert to tensors
inputs = [one_hot(char2idx[ch], V) for ch in ____]
targets = torch.tensor([char2idx[ch] for ch in ____], dtype=torch.long)

def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[____] = 1.0
    return vec
```

MEMORIZATION MNEMONICS

Memory Aids

Problem Memory Aids

Use these phrases to remember key patterns:

"Floor-Ceil-Four-Corners": Bilinear interpolation corners

- lt: floor-floor, rt: ceil-floor, lb: floor-ceil, rb: ceil-ceil

"Y-before-X-in-PyTorch": Deformable conv offset ordering

- delta_y = delta[n, 2*k, h_out, w_out]
- delta_x = delta[n, 2*k+1, h_out, w_out]

"3-32-64-128-256": CNN channel progression

- Each layer doubles the channels (except first)

"32-16-8-4": Spatial dimension reduction

- Each MaxPool2d(2,2) halves the spatial dimensions

"Zero-Forward-Backward-Step": Training loop mantra

- Never forget the order!

"Input-Hidden-Hidden": RNN weight dimensions

- W_xh: (H,V), W_hh: (H,H), W_hy: (V,H)

"Tanh-Hidden-Linear-Output": RNN computation flow

- Hidden uses tanh, output is linear

ANSWERS TO CODING BLOCKS

Code Block 1: floor, floor, 1, 1, 0.0, top-left, top-right, bottom-left, bottom-right, y0, x0, dx, dx, dx, dx, dy, dy

Code Block 2: stride, stride, 2, 2, 1, delta_y, delta_x, m_k, interpolated

Code Block 3: 3, 32, 32, 64, 64, 128, 128, 256, MaxPool2d, 256, 4, 4, 512, 256, 256, 100, 0.5

Code Block 4: relu, relu, 16x16, relu, 8x8, relu, 4x4, size(0), -1, relu, dropout, relu, dropout

Code Block 5: train, device, device, zero_grad, images, backward, step, 1, -1, 1

Code Block 6: 32, 64, 128, 256

Code Block 7: H, V, H, H, H, H, V, H, V

Code Block 8: W_xh, b_xh, W_hh, b_hh, W_hy, b_y

Code Block 9: True, True, True, True, True, True

ADVANCED CODING SCENARIOS

Scenario 1: Debugging Deformable Conv

Problem Scenario 1: Debugging Deformable Conv

If your deformable convolution gives wrong results, what are the most likely bugs?

```
# Common Bug 1: Wrong offset extraction
delta_y = delta[n, k, h_out, w_out]        # WRONG - missing factor of 2
delta_x = delta[n, k + 1, h_out, w_out]    # WRONG - should be 2*k+1

# Correct version:
delta_y = delta[n, ____ * k, h_out, w_out]
delta_x = delta[n, ____ * k + ____, h_out, w_out]

# Common Bug 2: Wrong bilinear interpolation order
# WRONG: Interpolate Y first
v_y = v_00 * (1 - dy) + v_10 * dy
v_final = v_y * (1 - dx) + v_01 * dx

# Correct: Interpolate X first, then Y
v_0 = v_00 * (1 - ____) + v_01 * ____  # top edge
v_1 = v_10 * (1 - ____) + v_11 * ____  # bottom edge
out = v_0 * (1 - ____) + v_1 * ____    # Y interpolation
```

Scenario 2: CNN Architecture Variations

Problem Scenario 2: CNN Architecture Variations

If asked to modify the CNN, remember these patterns:

```
# Adding more conv layers - maintain the pattern
class CustomCNNDeep(nn.Module):
    def __init__(self):
        super().__init__()
        # Pattern: start with 3 channels, double each time
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)  # Same channels
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)  # Double
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)  # Same
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1) # Double

    def forward(self, x):
        # Pattern: Conv-Conv-Pool, Conv-Conv-Pool
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)  # After every 2 conv layers

        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)

        # Calculate new flatten size: 128 * 8 * 8 = ____
```

Scenario 3: Validation vs Training Mode

Problem Scenario 3: Validation vs Training Mode

Critical differences between training and validation:

```
# Training mode
def train_epoch():
    model.____()  # Enable dropout and batch norm training mode

    for batch in train_loader:
        optimizer.____()  # Clear gradients
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.____()       # Compute gradients
        optimizer.____()  # Update weights

# Validation mode
def validate():
    model.____()  # Disable dropout, batch norm in eval mode

    with torch.____():  # Disable gradient computation
        for batch in val_loader:
            # NO optimizer.zero_grad() here!
            # NO loss.backward() here!
            # NO optimizer.step() here!
            outputs = model(images)
            loss = criterion(outputs, labels)
```

Scenario 4: RNN with Different Sequence Lengths

Problem Scenario 4: RNN with Different Sequence Lengths

If given a different text, adapt the RNN code:

```
# Original: "Deep Learning"
text = "Deep Learning"
input_seq = text[:-1]   # "Deep Learnin"
target_seq = text[1:]   # "eep Learning"

# New text: "Hello World"
text = "Hello World"
input_seq = text[____]  # "Hello Worl"
target_seq = text[____] # "ello World"

# Vocabulary size changes!
chars = sorted(list(set(text)))
V = len(chars)  # This will be different!

# All weight matrices need to be reinitialized with new V
W_xh = torch.randn(H, ____, requires_grad=True) * 0.1
W_hy = torch.randn(____, H, requires_grad=True) * 0.1
b_y = torch.zeros(____, requires_grad=True)
```

Scenario 5: Hyperparameter Grid Search Pattern

Problem Scenario 5: Hyperparameter Grid Search Pattern

Standard grid search implementation:

```
learning_rates = [0.0001, 0.001]
optimizers = [torch.optim.Adam, torch.optim.SGD]
model_classes = [CustomCNN, CustomCNNwithBN]

best_accuracy = 0
best_params = None

for model_class in model_classes:
    for optimizer_class in optimizers:
        for lr in learning_rates:
            # CRITICAL: Reinitialize model each time
            model = model_class().to(device)

            # Initialize optimizer based on type
            if optimizer_class == torch.optim.SGD:
                optimizer = optimizer_class(
                    model.parameters(),
                    lr=lr,
                    momentum=____,
                    weight_decay=____
                )
            else:  # Adam
                optimizer = optimizer_class(
                    model.parameters(),
                    lr=lr,
                    weight_decay=____
                )

            # Train and validate...
            val_acc = train_and_validate(model, optimizer)

            if val_acc > best_accuracy:
                best_accuracy = val_acc
                best_params = (model_class.__name__, optimizer_class.__name__, lr)
```

Scenario 6: Top-K Accuracy Calculation Variations

Problem Scenario 6: Top-K Accuracy Calculation Variations

Different ways to calculate accuracy:

```
# Top-1 accuracy (most common)
_, predicted = torch.max(outputs, 1)
correct = (predicted == labels).sum().item()
accuracy = correct / labels.size(0) * 100

# Top-K accuracy using topk
_, top_k_pred = outputs.topk(____, dim=1, largest=True, sorted=True)
top_k_correct = top_k_pred.eq(labels.view(-1, 1).expand_as(____)).sum().item()

# Alternative top-K calculation
values, indices = torch.topk(outputs, k=5, dim=1)
correct_mask = indices == labels.unsqueeze(1)
top_k_accuracy = correct_mask.sum().float() / labels.size(0) * 100
```

EXAM SIMULATION QUESTIONS

Quick Code Writing

Problem Quick Code Writing

Write these functions from memory in 2 minutes each:

1. Write the bilinear interpolation weight calculation:

```
def calculate_bilinear_weight(p_x, p_y, q_x, q_y):
    # Your code here - calculate G(p,q)
    return ____
```

2. Write the RNN hidden state update:

```
def rnn_step(x_t, h_prev, W_xh, W_hh, b_xh, b_hh):
    # Your code here - compute new hidden state
    return ____
```

3. Write the CNN forward pass for one block:

```
def cnn_block_forward(x, conv1, conv2, pool):
    # Your code here - conv-conv-pool pattern
    return ____
```

4. Write the training step:

```
def training_step(model, optimizer, criterion, images, labels):
    # Your code here - complete training step
    return loss
```

5. Write character to one-hot conversion:

```
def char_to_onehot(char, char2idx, vocab_size):
    # Your code here - convert character to one-hot vector
    return ____
```

FINAL MEMORY CHECK

Last Minute Review

Problem Last Minute Review

Before the exam, quickly verify you remember:

Constants (write from memory):

- CIFAR-100 mean: [____, ____, ____]
- CIFAR-100 std: [____, ____, ____]

- Train/val split ratio: ____
- Dropout probability: ____
- Weight decay: ____
- SGD momentum: ____

Dimensions (write from memory):

- CNN channels: 3→____→____→____→____
- CNN spatial: 32→____→____→____
- RNN W_xh: (____, ____)
- RNN W_hh: (____, ____)
- RNN W_hy: (____, ____)

PRACTICE EXAM - NO PEEKING AT ANSWERS!

Practice 1: Bilinear Interpolation Implementation

Problem Practice 1: Bilinear Interpolation Implementation

Complete this bilinear interpolation function:

```
def bilinear_interpolate(a_l, q_y, q_x):
    H, W = a_l.shape

    y0 = int(np.______(q_y))
    x0 = int(np.______(q_x))
    y1 = y0 + ___
    x1 = x0 + ___

    def get_pixel_value(img, y, x):
        if 0 <= y < H and 0 <= x < W:
            return img[y, x]
        else:
            return ____

    v_00 = get_pixel_value(a_l, y0, x0)
    v_01 = get_pixel_value(a_l, y0, x1)
    v_10 = get_pixel_value(a_l, y1, x0)
    v_11 = get_pixel_value(a_l, y1, x1)

    dy = q_y - ___
    dx = q_x - ___

    v_0 = v_00 * (1 - ___) + v_01 * ___
    v_1 = v_10 * (1 - ___) + v_11 * ___
    out = v_0 * (1 - ___) + v_1 * ___

    return out
```

Practice 2: Deformable Conv Offset Extraction

Problem Practice 2: Deformable Conv Offset Extraction

Fill in the correct offset extraction pattern:

```
for kh in range(K_h):
    for kw in range(K_w):
        k = kh * K_w + kw

        delta_y = delta[n, ___ * k, h_out, w_out]
        delta_x = delta[n, ___ * k + ___, h_out, w_out]
        m_k = mask[n, k, h_out, w_out]

        sample_y = h_start + kh * dilation + ____
        sample_x = w_start + kw * dilation + ____
```

```
            # Apply weight and mask
            value += weight[c_out, c_in, kh, kw] * ___ * ____
```

Practice 3: CNN Architecture

Problem Practice 3: CNN Architecture

Complete the CustomCNN constructor with correct channel progression:

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

        self.conv1 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(___, ___, kernel_size=3, padding=1)

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

        self.fc1 = nn.Linear(___ * ___ * ___, 512)
        self.fc2 = nn.Linear(___, ___)
        self.fc3 = nn.Linear(___, ___)

        self.dropout = nn.Dropout(____)
```

Practice 4: CNN Forward Pass

Problem Practice 4: CNN Forward Pass

Complete the forward pass with correct activations:

```
def forward(self, x):
    x = F.____(self.conv1(x))
    x = F.____(self.conv2(x))
    x = self.pool(x)  # 32x32 -> ____

    x = F.____(self.conv3(x))
    x = self.pool(x)  # 16x16 -> ____

    x = F.____(self.conv4(x))
    x = self.pool(x)  # 8x8 -> ____

    x = x.view(x.____(__), ___)

    x = F.____(self.fc1(x))
    x = self.______(x)
```

```
    x = F.____(self.fc2(x))
    x = self.______(x)
    x = self.fc3(x)

    return x
```

Practice 5: Training Loop Pattern

Problem Practice 5: Training Loop Pattern

Complete the sacred training loop:

```
def train_step(model, optimizer, criterion, images, labels, device):
    images = images.to(____)
    labels = labels.to(____)

    optimizer.____()
    outputs = model(______)
    loss = criterion(outputs, labels)
    loss.______()
    optimizer.____()

    return loss.item()
```

Practice 6: Top-K Accuracy

Problem Practice 6: Top-K Accuracy

Complete the top-1 and top-5 accuracy calculation:

```
# Top-1 accuracy
_, top1_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top1_correct = top1_pred.eq(labels.view(___, ___)).sum().item()

# Top-5 accuracy
_, top5_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top5_correct = top5_pred.eq(labels.view(-1, 1).____(____))).sum().item()
```

Practice 7: RNN Weight Initialization

Problem Practice 7: RNN Weight Initialization

Initialize RNN parameters with correct dimensions:

```
V = len(chars)  # Vocabulary size
H = 16          # Hidden size

W_xh = torch.randn(___, ___, requires_grad=True) * 0.1
W_hh = torch.randn(___, ___, requires_grad=True) * 0.1
b_xh = torch.zeros(___, requires_grad=True)
b_hh = torch.zeros(___, requires_grad=True)
W_hy = torch.randn(___, ___, requires_grad=True) * 0.1
b_y = torch.zeros(___, requires_grad=True)
```

Practice 8: RNN Forward Equations

Problem Practice 8: RNN Forward Equations

Complete the RNN forward pass:

```
for t in range(seq_len):
    x_t = inputs[t]

    # Hidden state update
    h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)

    # Output logits
    s_t = _____ @ h + _____

    logits_list.append(s_t)
```

Practice 9: Character Processing

Problem Practice 9: Character Processing

Complete the character-level preprocessing:

```
text = "Deep Learning"

chars = sorted(list(set(____)))
char2idx = {ch: i for i, ch in enumerate(____)}
idx2char = {i: ch for i, ch in enumerate(____)}

input_seq = text[____]
target_seq = text[____]

def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[____] = 1.0
    return vec
```

```
inputs = [one_hot(char2idx[ch], V) for ch in ____]
targets = torch.tensor([char2idx[ch] for ch in ____], dtype=torch.long)
```

Practice 10: Gradient Computation

Problem Practice 10: Gradient Computation

Complete the explicit gradient computation:

```
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=____)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=____)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=____)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=____)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=____)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=____)[0]
```

Practice 11: BatchNorm Integration

Problem Practice 11: BatchNorm Integration

Add BatchNorm to CNN architecture:

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

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(____)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(____)

    def forward(self, x):
        x = F.relu(self.______(self.conv1(x)))
        x = F.relu(self.______(self.conv2(x)))
```

Practice 12: CIFAR-100 Constants

Problem Practice 12: CIFAR-100 Constants

Fill in the CIFAR-100 normalization values:

```
transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize(mean=[____, ____, ____],
                         std=[____, ____, ____])
])
```

Practice 13: Data Splitting

Problem Practice 13: Data Splitting

Complete the 80/20 train/validation split:

```
full_train_set = CIFAR100(root='./data', train=True, download=True)

train_size = int(____ * len(full_train_set))
val_size = len(full_train_set) - ____

train_set, val_set = random_split(full_train_set, [____, ____])
```

Practice 14: Loss and Optimizer Setup

Problem Practice 14: Loss and Optimizer Setup

Set up loss function and optimizer for CIFAR-100:

```
loss_function = nn.______()

optimizer = optim.SGD(model.parameters(),
                     lr=____,
                     momentum=____,
                     weight_decay=____)
```

Practice 15: Quick Recall

Problem Practice 15: Quick Recall

Answer these without looking:

1. What function rounds down to nearest integer? __________
2. In PyTorch deformable conv, Y offset comes at index ____ and X offset at ____
3. The bilinear interpolation weight formula is: G(p,q) = ____________________

4. CNN channel progression in our model: 3 → ___ → ___ → ___ → ___
5. RNN hidden state uses which activation function? __________
6. In training loop, what comes after loss.backward()? __________
7. For validation, you wrap the loop with torch.__________():
8. The flatten operation uses x.______(x.______(0), ____):
9. Top-5 accuracy uses outputs.______(___, dim=1):
10. RNN weight W_hy has dimensions (___, ___):

CHALLENGE PROBLEMS

Challenge 1: Complete Function Implementation

Problem Challenge 1: Complete Function Implementation

Write the complete bilinear interpolation function from scratch:

Challenge 2: Complete RNN Step

Problem Challenge 2: Complete RNN Step

Write a complete RNN forward step function:

Challenge 3: Training vs Validation

Problem Challenge 3: Training vs Validation

Write the key differences between training and validation loops:

Problem Challenge 4: Debugging Scenario

Your deformable convolution outputs don't match PyTorch. List 5 most likely bugs:

1. ____
2. ____
3. ____
4. ____
5. ____

Challenge 5: Architecture Design

Problem Challenge 5: Architecture Design

Design a deeper CNN with 6 conv layers following the same pattern:

ANSWER BANK

Practice 1 Answers:

Problem Practice 1 Answers:

floor, floor, 1, 1, 0.0, y0, x0, dx, dx, dx, dx, dy, dy

Practice 2 Answers:

Problem Practice 2 Answers:

2, 2, 1, delta_y, delta_x, m_k, interpolated

Practice 3 Answers:

Problem Practice 3 Answers:

3, 32, 32, 64, 64, 128, 128, 256, MaxPool2d, 256, 4, 4, 512, 256, 256, 100, 0.5

Practice 4 Answers:

Problem Practice 4 Answers:

relu, relu, 16x16, relu, 8x8, relu, 4x4, size, 0, -1, relu, dropout, relu, dropout

Practice 5 Answers:

Problem Practice 5 Answers:

device, device, zero_grad, images, backward, step

Practice 6 Answers:

Problem Practice 6 Answers:

1, True, True, -1, 1, 5, True, True, expand_as(top5_pred)

Practice 7 Answers:

Problem Practice 7 Answers:

H, V, H, H, H, H, V, H, V

Practice 8 Answers:

Problem Practice 8 Answers:

W_xh, b_xh, W_hh, b_hh, W_hy, b_y

Practice 9 Answers:

Problem Practice 9 Answers:

text, chars, chars, [:-1], [1:], idx, input_seq, target_seq

Practice 10 Answers:

Problem Practice 10 Answers:

True, True, True, True, True, True

Practice 11 Answers:

Problem Practice 11 Answers:

32, 64, bn1, bn2

Practice 12 Answers:

Problem Practice 12 Answers:

0.5071, 0.4867, 0.4408, 0.2675, 0.2565, 0.2761

Practice 13 Answers:

Problem Practice 13 Answers:

0.8, train_size, train_size, val_size

Practice 14 Answers:

Problem Practice 14 Answers:

CrossEntropyLoss, 0.01, 0.9, 5e-4

Practice 15 Answers:

Problem Practice 15 Answers:

1. np.floor 2. 2*k, 2*k+1 3. (1-—px-qx—)*(1-—py-qy—) 4. 32, 64, 128, 256 5. tanh 6. optimizer.step() 7. no_grad 8. view, size, -1 9. topk, 5 10. V, H

Challenge Problem Answers:

Problem Challenge Problem Answers:

Challenge 1: Complete bilinear function with all bounds checking, corner calculations, and interpolation steps.

Challenge 2: RNN step with tanh activation: h = torch.tanh(W_xh @ x + b_xh + W_hh @ h_prev + b_hh)

Challenge 3: Training: model.train(), optimizer steps, gradients enabled. Validation: model.eval(), torch.no_grad(), no optimizer steps.

Challenge 4: 1. Wrong offset indexing (2*k), 2. Wrong bilinear order (X first), 3. Missing zero padding, 4. Wrong sampling position calculation, 5. Incorrect weight/mask application.

Challenge 5: Conv layers: $3 \rightarrow 32 \rightarrow 32 \rightarrow 64 \rightarrow 64 \rightarrow 128 \rightarrow 128$, with pooling after every 2 conv layers.

CENG403 - Spring 2025: Homework set THE-2 Study Guide

Your Name

TASK 1: DEFORMABLE CNN MEMORIZATION GUIDE

.1

Problem 1.1

What are the 4 corner positions for bilinear interpolation given fractional position $q = (q_x, q_y)$? **Pattern to Remember:** Floor-Floor, Ceil-Floor, Floor-Ceil, Ceil-Ceil

$$p_{lt} = (\lfloor q_x \rfloor, \lfloor q_y \rfloor) \quad \text{(left top)} \tag{1}$$
$$p_{rt} = (\lceil q_x \rceil, \lfloor q_y \rfloor) \quad \text{(right top)} \tag{2}$$
$$p_{lb} = (\lfloor q_x \rfloor, \lceil q_y \rceil) \quad \text{(left bottom)} \tag{3}$$
$$p_{rb} = (\lceil q_x \rceil, \lceil q_y \rceil) \quad \text{(right bottom)} \tag{4}$$

.2

Problem 1.2

Complete the bilinear interpolation weight formula:

$$G(p, q) = (1 - |p_x - q_x|) \cdot (1 - |p_y - q_y|)$$

.3

Problem 1.3

Fill in the missing code for bilinear interpolation bounds checking:

```
def get_pixel_value(img, y, x):
    if 0 <= y < H and 0 <= x < W:
        return img[y, x]
    else:
        return ___  # What goes here?
```

Answer: `0.0` (zero padding for out-of-bounds)

.4

Problem 1.4

What is the correct order for bilinear interpolation calculation? **Memory Pattern:** "First X, then Y"

1. Get 4 corner values: $v_{00}, v_{01}, v_{10}, v_{11}$
2. Calculate fractional parts: $dx = q_x - x_0$, $dy = q_y - y_0$
3. Interpolate along X: $v_0 = v_{00}(1 - dx) + v_{01} \cdot dx$
4. Interpolate along X: $v_1 = v_{10}(1 - dx) + v_{11} \cdot dx$
5. Interpolate along Y: $out = v_0(1 - dy) + v_1 \cdot dy$

.5

Problem 1.5

In deformable convolution, how do you extract the y and x offsets from the delta tensor? **Critical Pattern - PyTorch stores Y first, then X:**

```
delta_y = delta[n, 2 * k, h_out, w_out]      # y offset
delta_x = delta[n, 2 * k + 1, h_out, w_out]  # x offset
```

.6

Problem 1.6

What is the deformable convolution sampling position formula?

```
sample_y = h_start + kh * dilation + _____
sample_x = w_start + kw * dilation + _____
```

Answer: `delta_y` and `delta_x`

TASK 2: CNN PYTORCH MEMORIZATION GUIDE

.1

Problem 2.1

What are the CIFAR-100 normalization values you must memorize? **Critical Constants:**

```
mean=[0.5071, 0.4867, 0.4408]  # CIFAR100 mean
std=[0.2675, 0.2565, 0.2761]   # CIFAR100 std
```

.2

Problem 2.2

Complete the data augmentation transforms for training:

```
transform_train = transforms.Compose([
    transforms.______(32, padding=4),  # What goes here?
    transforms.______(),               # What goes here?
    transforms.ToTensor(),
    transforms.Normalize(mean=[...], std=[...])
])
```

Answer: `RandomCrop` and `RandomHorizontalFlip`

.3

Problem 2.3

How do you split CIFAR-100 training data into 80/20 train/validation? **Pattern to Remember:**

```
train_size = int(0.8 * len(full_train_set))
val_size = len(full_train_set) - train_size
train_set, val_set = random_split(full_train_set, [train_size, val_size])
```

.4

Problem 2.4

What is the CNN architecture pattern for the CustomCNN class? **Layer Sequence Pattern:**

1. Conv2d(3, 32) → Conv2d(32, 64) → MaxPool2d
2. Conv2d(64, 128) → MaxPool2d
3. Conv2d(128, 256) → MaxPool2d
4. Flatten → FC(256*4*4, 512) → FC(512, 256) → FC(256, 100)

.5

Problem 2.5

Complete the forward pass activation pattern:

```
x = F.relu(self.conv1(x))
x = F.relu(self.conv2(x))
x = self.pool(x)  # 32x32 -> 16x16
x = F.relu(self.conv3(x))
x = self.pool(x)  # 16x16 -> 8x8
x = F.relu(self.conv4(x))
x = self.pool(x)  # 8x8 -> 4x4
x = x.view(x.size(0), -1)  # Flatten
x = F.relu(self.fc1(x))
x = self.dropout(x)
x = F.relu(self.fc2(x))
x = self.dropout(x)
x = self.fc3(x)  # No activation on final layer!
```

.6

Problem 2.6

What loss function and optimizer setup is standard for CIFAR-100?

```
loss_function = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(),
                     lr=0.01,
                     momentum=0.9,
                     weight_decay=5e-4)
```

.7

Problem 2.7

Complete the top-1 and top-5 accuracy calculation:

```
# Top-1 accuracy
_, top1_pred = outputs.topk(1, dim=1, largest=True, sorted=True)
top1_correct = top1_pred.eq(labels.view(-1, 1)).sum().item()

# Top-5 accuracy
_, top5_pred = outputs.topk(5, dim=1, largest=True, sorted=True)
top5_correct = top5_pred.eq(labels.view(-1, 1).____(____))).sum().item()
```

Answer: `expand_as(top5_pred)`

.8

Problem 2.8

What is the training loop structure pattern? **Memory Pattern - "Zero, Forward, Backward, Step":**

```
optimizer.zero_grad()    # Clear gradients
outputs = model(images)  # Forward pass
loss = loss_function(outputs, labels)  # Compute loss
loss.backward()          # Backward pass
optimizer.step()         # Update weights
```

.9

Problem 2.9

How do you add BatchNorm2d to the CNN architecture? **Pattern - After each Conv2d:**

```
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
self.bn1 = nn.BatchNorm2d(32)  # Same number as conv output
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
self.bn2 = nn.BatchNorm2d(64)  # Same number as conv output
```

TASK 3: RNN MEMORIZATION GUIDE

.1

Problem 3.1

How do you create character vocabulary and mappings? **Standard Pattern:**

```
chars = sorted(list(set(text)))
char2idx = {ch: i for i, ch in enumerate(chars)}
idx2char = {i: ch for i, ch in enumerate(chars)}
```

.2

Problem 3.2

How do you create input and target sequences for character prediction?

```
input_seq = text[:-1]  # All except last
target_seq = text[1:]  # All except first
```

.3

Problem 3.3

Complete the one-hot encoding function:

```
def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[___] = 1.0
    return vec
```

Answer: `idx`

.4

Problem 3.4

What are the RNN weight matrix dimensions? **Dimension Memory Pattern:**

```
W_xh = torch.randn(H, V, requires_grad=True) * 0.1  # (H, V)
W_hh = torch.randn(H, H, requires_grad=True) * 0.1  # (H, H)
b_xh = torch.zeros(H, requires_grad=True)           # (H,)
b_hh = torch.zeros(H, requires_grad=True)           # (H,)
W_hy = torch.randn(V, H, requires_grad=True) * 0.1  # (V, H)
b_y = torch.zeros(V, requires_grad=True)            # (V,)
```

.5

Problem 3.5

Complete the RNN forward pass equations:

```
# Hidden state update
h = torch.tanh(W_xh @ x_t + b_xh + W_hh @ h + b_hh)
```

```
# Output logits
s_t = _____ @ h + _____
```

Answer: `W_hy` and `b_y`

.6

Problem 3.6

How do you compute gradients explicitly with torch.autograd?

```
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=True)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=True)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=True)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=True)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=True)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=True)[0]
```

CRITICAL MEMORIZATION PATTERNS

Patterns

Problem Patterns

What are the key patterns you must memorize?

1. Bilinear Interpolation Pattern:

- Get 4 corners (floor/ceil combinations)
- Interpolate X first, then Y
- Use fractional parts: $dx = q_x - x_0$, $dy = q_y - y_0$

2. CNN Architecture Pattern:

- Conv-Conv-Pool, Conv-Pool, Conv-Pool structure
- Channel progression: 3→32→64→128→256
- Spatial reduction: 32→16→8→4
- FC layers: 4096→512→256→100

3. Training Loop Pattern:

- "Zero-Forward-Backward-Step"
- Always move tensors to device
- Use torch.no_grad() for validation

4. RNN Equations Pattern:

- $h_t = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$
- $s_t = W_{hy}h_t + b_y$
- Always remember matrix dimensions

5. Data Preprocessing Patterns:

- CIFAR-100: mean=[0.5071, 0.4867, 0.4408], std=[0.2675, 0.2565, 0.2761]
- 80/20 split: train_size = int(0.8 * len(dataset))
- Character sequences: input = text[:-1], target = text[1:]

COMMON MISTAKES TO AVOID

Mistakes

Problem Mistakes

What are the most common implementation mistakes?

1. Deformable CNN:

- Forget PyTorch stores Y offset first, then X offset
- Wrong bilinear interpolation order (do X first, then Y)
- Forget zero padding for out-of-bounds pixels

2. CNN:

- Forget to move tensors to device
- Wrong flatten calculation: x.view(x.size(0), -1)
- Forget activation on hidden layers, add activation on output layer

3. RNN:

- Wrong weight matrix dimensions
- Forget requires_grad=True for parameters
- Use retain_graph=True for multiple gradient computations

RAPID-FIRE ACTIVE RECALL QUIZ

Speed Round 1: Fill the Blanks

Problem Speed Round 1: Fill the Blanks

Complete these critical code snippets:

Q1: Bilinear interpolation corners:

```
y0 = int(np.______(q_y))
x0 = int(np.______(q_x))
y1 = y0 + ___
x1 = x0 + ___
```

Q2: Deformable conv offset extraction:

```
delta_y = delta[n, ___ * k, h_out, w_out]
delta_x = delta[n, ___ * k + ___, h_out, w_out]
```

Q3: CNN flatten operation:

```
x = x.view(x.____(__), ___)
```

Q4: Top-5 accuracy calculation:

```
_, top5_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top5_correct = top5_pred.eq(labels.view(-1, 1).______(_____)).sum().item()
```

Q5: RNN hidden state update:

```
h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)
```

Speed Round 2: True/False

Problem Speed Round 2: True/False

Mark T/F for these statements:

1. In bilinear interpolation, you interpolate Y direction first, then X direction. **[T/F]**
2. PyTorch stores Y offset before X offset in deformable convolution. **[T/F]**
3. CIFAR-100 has 100 classes, so the final FC layer outputs 100 values. **[T/F]**
4. You should apply ReLU activation to the final output layer in classification. **[T/F]**
5. In RNN, W_xh has dimensions (V, H). **[T/F]**
6. For validation, you need to call optimizer.zero_grad(). **[T/F]**
7. BatchNorm2d should be applied before the activation function. **[T/F]**
8. The input sequence for RNN is text[1:] and target is text[:-1]. **[T/F]**

Speed Round 3: Memory Palace

Problem Speed Round 3: Memory Palace

Associate these concepts with memorable phrases:

Bilinear Interpolation: ”Four corners, X then Y, fractional magic”

- 4 corners: lt, rt, lb, rb
- X interpolation: top_edge, bottom_edge
- Y interpolation: final result

CNN Architecture: ”3 to 32, double-double-double, then shrink to 100”

- Channels: 3→32→64→128→256
- Spatial: 32→16→8→4
- FC: 4096→512→256→100

Training Loop: ”Zero-Forward-Backward-Step dance”

- optimizer.zero_grad()
- outputs = model(images)
- loss.backward()
- optimizer.step()

RNN Weights: ”Input-Hidden-Hidden, Hidden-Hidden-Hidden, Hidden-Vocab-Vocab”

- W_xh: (H, V) - maps input to hidden
- W_hh: (H, H) - maps previous hidden to current hidden
- W_hy: (V, H) - maps hidden to output vocabulary

LAST-MINUTE CHECKLIST

Pre-Exam Checklist

Problem Pre-Exam Checklist

Before the exam, ensure you can write from memory:

Critical Constants:

- CIFAR-100 mean: [0.5071, 0.4867, 0.4408]
- CIFAR-100 std: [0.2675, 0.2565, 0.2761]
- Train/val split: 0.8 * len(dataset)

Key Formulas:

- Bilinear weight: $(1 - |p_x - q_x|) \cdot (1 - |p_y - q_y|)$
- RNN hidden: $h_t = \tanh(W_{xh}x_t + b_{xh} + W_{hh}h_{t-1} + b_{hh})$
- RNN output: $s_t = W_{hy}h_t + b_y$

Critical Code Patterns:

- Device transfer: `tensor.to(device)`
- Gradient computation: `torch.autograd.grad(loss, param, retain_graph=True)[0]`
- Top-k accuracy: `outputs.topk(k, dim=1, largest=True, sorted=True)`
- One-hot encoding: `vec[idx] = 1.0`

Architecture Patterns:

- CNN: Conv→BN→ReLU→Pool pattern
- RNN: Input→Hidden→Output with recurrence
- Training: Zero→Forward→Backward→Step

CODING BLOCKS MEMORIZATION

Code Block 1: Bilinear Interpolation Core

Problem Code Block 1: Bilinear Interpolation Core

Complete this bilinear interpolation function - focus on the mathematical pattern:

```
def bilinear_interpolate(a_l, q_y, q_x):
    H, W = a_l.shape

    # Step 1: Get integer positions
    y0 = int(np.______(q_y))
    x0 = int(np.______(q_x))
    y1 = y0 + ___
    x1 = x0 + ___

    # Step 2: Get values with bounds checking
    def get_pixel_value(img, y, x):
        if 0 <= y < H and 0 <= x < W:
            return img[y, x]
        else:
            return ____  # Out of bounds value

    # Step 3: Get four corner values
    v_00 = get_pixel_value(a_l, y0, x0)  # ___-___
    v_01 = get_pixel_value(a_l, y0, x1)  # ___-___
    v_10 = get_pixel_value(a_l, y1, x0)  # ___-___
    v_11 = get_pixel_value(a_l, y1, x1)  # ___-___

    # Step 4: Calculate fractional parts
    dy = q_y - ___
    dx = q_x - ___

    # Step 5: Interpolate X first, then Y
    v_0 = v_00 * (1 - ___) + v_01 * ___  # top edge
    v_1 = v_10 * (1 - ___) + v_11 * ___  # bottom edge
    out = v_0 * (1 - ___) + v_1 * ___    # final Y interpolation

    return out
```

Code Block 2: Deformable Conv Key Loop

Problem Code Block 2: Deformable Conv Key Loop

This is the heart of deformable convolution - memorize the offset extraction pattern:

```
for n in range(N):  # batch
    for c_out in range(C_out):  # output channels
        for h_out in range(H_out):  # height
            for w_out in range(W_out):  # width
                h_start = h_out * ____
                w_start = w_out * ____
                value = 0.0

                for kh in range(K_h):
                    for kw in range(K_w):
```

```
                k = kh * K_w + kw

                # CRITICAL: PyTorch offset order
                delta_y = delta[n, ___ * k, h_out, w_out]
                delta_x = delta[n, ___ * k + ___, h_out, w_out]
                m_k = mask[n, k, h_out, w_out]

                # Sampling position
                sample_y = h_start + kh * dilation + ____
                sample_x = w_start + kw * dilation + ____

                for c_in in range(C_in):
                    interpolated = bilinear_interpolate(
                        a_l[n, c_in, :, :], sample_y, sample_x
                    )
                    value += weight[c_out, c_in, kh, kw] * ___ * ____

            out[n, c_out, h_out, w_out] = value
```

Code Block 3: CNN Architecture Constructor

Problem Code Block 3: CNN Architecture Constructor

Memorize the channel progression and layer naming pattern:

```
class CustomCNN(nn.Module):
    def __init__(self, norm_layer=None):
        super(CustomCNN, self).__init__()

        # Conv layers - memorize the channel progression
        self.conv1 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(___, ___, kernel_size=3, padding=1)

        # Pooling layer
        self.pool = nn.______(2, 2)

        # FC layers - calculate the input size
        self.fc1 = nn.Linear(___ * ___ * ___, 512)
        self.fc2 = nn.Linear(___, ___)
        self.fc3 = nn.Linear(___, ___)  # CIFAR-100 classes

        self.dropout = nn.Dropout(____)
```

Code Block 4: CNN Forward Pass Pattern

Problem Code Block 4: CNN Forward Pass Pattern

Memorize the activation and pooling pattern:

```
def forward(self, x):
    # Block 1: Conv-Conv-Pool
    x = F.____(self.conv1(x))
    x = F.____(self.conv2(x))
    x = self.pool(x)  # 32x32 -> ____
```

```
    # Block 2: Conv-Pool
    x = F.____(self.conv3(x))
    x = self.pool(x)  # 16x16 -> ____

    # Block 3: Conv-Pool
    x = F.____(self.conv4(x))
    x = self.pool(x)  # 8x8 -> ____

    # Flatten
    x = x.view(x.____(__), ___)

    # FC layers with dropout
    x = F.____(self.fc1(x))
    x = self.______(x)
    x = F.____(self.fc2(x))
    x = self.______(x)
    x = self.fc3(x)  # No activation here!

    return x
```

Code Block 5: Training Loop Core

Problem Code Block 5: Training Loop Core

The sacred training loop pattern - memorize the order:

```
def train(model, train_loader, optimizer, loss_function, device):
    model.____()  # Set to training mode

    for batch in train_loader:
        images, labels = batch
        images = images.to(____)
        labels = labels.to(____)

        # The sacred four steps:
        optimizer.____()         # Step 1: Clear gradients
        outputs = model(______)  # Step 2: Forward pass
        loss = loss_function(outputs, labels)  # Step 3: Compute loss
        loss.______()            # Step 4: Backward pass
        optimizer.____()         # Step 5: Update weights

        # Accuracy calculation
        _, top1_pred = outputs.topk(___, dim=1, largest=True, sorted=True)
        top1_correct = top1_pred.eq(labels.view(___, ___)).sum().item()
```

Code Block 6: BatchNorm CNN Constructor

Problem Code Block 6: BatchNorm CNN Constructor

Pattern for adding BatchNorm after each conv layer:

```
class CustomCNNwithBN(nn.Module):
    def __init__(self, norm_layer=None):
        super(CustomCNNwithBN, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(____)  # Same as conv1 output
```

```
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(____)  # Same as conv2 output

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(____)  # Same as conv3 output

        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(____)  # Same as conv4 output

    def forward(self, x):
        # Pattern: Conv -> BN -> ReLU
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = self.pool(x)

        x = F.relu(self.bn4(self.conv4(x)))
        x = self.pool(x)
        # ... rest of forward pass
```

Code Block 7: RNN Parameter Initialization

Problem Code Block 7: RNN Parameter Initialization

Memorize the weight dimensions and initialization pattern:

```
# RNN parameters - memorize the dimensions!
W_xh = torch.randn(___, ___, requires_grad=True) * 0.1  # Input to hidden
W_hh = torch.randn(___, ___, requires_grad=True) * 0.1  # Hidden to hidden
b_xh = torch.zeros(___, requires_grad=True)             # Hidden bias 1
b_hh = torch.zeros(___, requires_grad=True)             # Hidden bias 2
W_hy = torch.randn(___, ___, requires_grad=True) * 0.1  # Hidden to output
b_y = torch.zeros(___, requires_grad=True)              # Output bias

# Remember: V = vocab size, H = hidden size
```

Code Block 8: RNN Forward Pass

Problem Code Block 8: RNN Forward Pass

The RNN equations in code form:

```
logits_list = []
h = torch.zeros(H)

for t in range(seq_len):
    x_t = inputs[t]

    # Hidden state update equation
    h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)

    # Output logits equation
    s_t = _____ @ h + _____
```

```
    logits_list.append(s_t)

logits = torch.stack(logits_list)
log_probs = F.log_softmax(logits, dim=1)
loss_manual = F.nll_loss(log_probs, targets)
```

Code Block 9: Gradient Computation

Problem Code Block 9: Gradient Computation

Pattern for explicit gradient computation:

```
# Compute gradients explicitly
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=____)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=____)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=____)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=____)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=____)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=____)[0]

# Why retain_graph=True? Because we compute multiple gradients from same loss
```

Code Block 10: Character Processing

Problem Code Block 10: Character Processing

Standard pattern for character-level RNN preprocessing:

```
text = "Deep Learning"

# Step 1: Create vocabulary
chars = sorted(list(set(____)))
char2idx = {ch: i for i, ch in enumerate(____)}
idx2char = {i: ch for i, ch in enumerate(____)}

# Step 2: Create sequences
input_seq = text[____]   # All except last
target_seq = text[____]  # All except first

# Step 3: Convert to tensors
inputs = [one_hot(char2idx[ch], V) for ch in ____]
targets = torch.tensor([char2idx[ch] for ch in ____], dtype=torch.long)

def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[____] = 1.0
    return vec
```

MEMORIZATION MNEMONICS

Memory Aids

Problem Memory Aids

Use these phrases to remember key patterns:

"Floor-Ceil-Four-Corners": Bilinear interpolation corners

- lt: floor-floor, rt: ceil-floor, lb: floor-ceil, rb: ceil-ceil

"Y-before-X-in-PyTorch": Deformable conv offset ordering

- delta_y = delta[n, 2*k, h_out, w_out]
- delta_x = delta[n, 2*k+1, h_out, w_out]

"3-32-64-128-256": CNN channel progression

- Each layer doubles the channels (except first)

"32-16-8-4": Spatial dimension reduction

- Each MaxPool2d(2,2) halves the spatial dimensions

"Zero-Forward-Backward-Step": Training loop mantra

- Never forget the order!

"Input-Hidden-Hidden": RNN weight dimensions

- W_xh: (H,V), W_hh: (H,H), W_hy: (V,H)

"Tanh-Hidden-Linear-Output": RNN computation flow

- Hidden uses tanh, output is linear

ANSWERS TO CODING BLOCKS

Code Block 1: floor, floor, 1, 1, 0.0, top-left, top-right, bottom-left, bottom-right, y0, x0, dx, dx, dx, dx, dy, dy

Code Block 2: stride, stride, 2, 2, 1, delta_y, delta_x, m_k, interpolated

Code Block 3: 3, 32, 32, 64, 64, 128, 128, 256, MaxPool2d, 256, 4, 4, 512, 256, 256, 100, 0.5

Code Block 4: relu, relu, 16x16, relu, 8x8, relu, 4x4, size(0), -1, relu, dropout, relu, dropout

Code Block 5: train, device, device, zero_grad, images, backward, step, 1, -1, 1

Code Block 6: 32, 64, 128, 256

Code Block 7: H, V, H, H, H, H, V, H, V

Code Block 8: W_xh, b_xh, W_hh, b_hh, W_hy, b_y

Code Block 9: True, True, True, True, True, True

ADVANCED CODING SCENARIOS

Scenario 1: Debugging Deformable Conv

Problem Scenario 1: Debugging Deformable Conv

If your deformable convolution gives wrong results, what are the most likely bugs?

```
# Common Bug 1: Wrong offset extraction
delta_y = delta[n, k, h_out, w_out]        # WRONG - missing factor of 2
delta_x = delta[n, k + 1, h_out, w_out]    # WRONG - should be 2*k+1

# Correct version:
delta_y = delta[n, ____ * k, h_out, w_out]
delta_x = delta[n, ____ * k + ____, h_out, w_out]

# Common Bug 2: Wrong bilinear interpolation order
# WRONG: Interpolate Y first
v_y = v_00 * (1 - dy) + v_10 * dy
v_final = v_y * (1 - dx) + v_01 * dx

# Correct: Interpolate X first, then Y
v_0 = v_00 * (1 - ____) + v_01 * ____  # top edge
v_1 = v_10 * (1 - ____) + v_11 * ____  # bottom edge
out = v_0 * (1 - ____) + v_1 * ____    # Y interpolation
```

Scenario 2: CNN Architecture Variations

Problem Scenario 2: CNN Architecture Variations

If asked to modify the CNN, remember these patterns:

```
# Adding more conv layers - maintain the pattern
class CustomCNNDeep(nn.Module):
    def __init__(self):
        super().__init__()
        # Pattern: start with 3 channels, double each time
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)  # Same channels
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)  # Double
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)  # Same
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1) # Double

    def forward(self, x):
        # Pattern: Conv-Conv-Pool, Conv-Conv-Pool
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool(x)  # After every 2 conv layers

        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool(x)

        # Calculate new flatten size: 128 * 8 * 8 = ____
```

Scenario 3: Validation vs Training Mode

Problem Scenario 3: Validation vs Training Mode

Critical differences between training and validation:

```
# Training mode
def train_epoch():
    model.____()  # Enable dropout and batch norm training mode

    for batch in train_loader:
        optimizer.____()  # Clear gradients
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.____()       # Compute gradients
        optimizer.____()  # Update weights

# Validation mode
def validate():
    model.____()  # Disable dropout, batch norm in eval mode

    with torch.____():  # Disable gradient computation
        for batch in val_loader:
            # NO optimizer.zero_grad() here!
            # NO loss.backward() here!
            # NO optimizer.step() here!
            outputs = model(images)
            loss = criterion(outputs, labels)
```

Scenario 4: RNN with Different Sequence Lengths

Problem Scenario 4: RNN with Different Sequence Lengths

If given a different text, adapt the RNN code:

```
# Original: "Deep Learning"
text = "Deep Learning"
input_seq = text[:-1]   # "Deep Learnin"
target_seq = text[1:]   # "eep Learning"

# New text: "Hello World"
text = "Hello World"
input_seq = text[____]  # "Hello Worl"
target_seq = text[____] # "ello World"

# Vocabulary size changes!
chars = sorted(list(set(text)))
V = len(chars)  # This will be different!

# All weight matrices need to be reinitialized with new V
W_xh = torch.randn(H, ____, requires_grad=True) * 0.1
W_hy = torch.randn(____, H, requires_grad=True) * 0.1
b_y = torch.zeros(____, requires_grad=True)
```

Scenario 5: Hyperparameter Grid Search Pattern

Problem Scenario 5: Hyperparameter Grid Search Pattern

Standard grid search implementation:

```python
learning_rates = [0.0001, 0.001]
optimizers = [torch.optim.Adam, torch.optim.SGD]
model_classes = [CustomCNN, CustomCNNwithBN]

best_accuracy = 0
best_params = None

for model_class in model_classes:
    for optimizer_class in optimizers:
        for lr in learning_rates:
            # CRITICAL: Reinitialize model each time
            model = model_class().to(device)

            # Initialize optimizer based on type
            if optimizer_class == torch.optim.SGD:
                optimizer = optimizer_class(
                    model.parameters(),
                    lr=lr,
                    momentum=____,
                    weight_decay=____
                )
            else:  # Adam
                optimizer = optimizer_class(
                    model.parameters(),
                    lr=lr,
                    weight_decay=____
                )

            # Train and validate...
            val_acc = train_and_validate(model, optimizer)

            if val_acc > best_accuracy:
                best_accuracy = val_acc
                best_params = (model_class.__name__, optimizer_class.__name__, lr)
```

Scenario 6: Top-K Accuracy Calculation Variations

Problem Scenario 6: Top-K Accuracy Calculation Variations

Different ways to calculate accuracy:

```python
# Top-1 accuracy (most common)
_, predicted = torch.max(outputs, 1)
correct = (predicted == labels).sum().item()
accuracy = correct / labels.size(0) * 100

# Top-K accuracy using topk
_, top_k_pred = outputs.topk(____, dim=1, largest=True, sorted=True)
top_k_correct = top_k_pred.eq(labels.view(-1, 1).expand_as(____)).sum().item()

# Alternative top-K calculation
values, indices = torch.topk(outputs, k=5, dim=1)
correct_mask = indices == labels.unsqueeze(1)
top_k_accuracy = correct_mask.sum().float() / labels.size(0) * 100
```

EXAM SIMULATION QUESTIONS

Quick Code Writing

Problem Quick Code Writing

Write these functions from memory in 2 minutes each:

1. Write the bilinear interpolation weight calculation:

```
def calculate_bilinear_weight(p_x, p_y, q_x, q_y):
    # Your code here - calculate G(p,q)
    return ____
```

2. Write the RNN hidden state update:

```
def rnn_step(x_t, h_prev, W_xh, W_hh, b_xh, b_hh):
    # Your code here - compute new hidden state
    return ____
```

3. Write the CNN forward pass for one block:

```
def cnn_block_forward(x, conv1, conv2, pool):
    # Your code here - conv-conv-pool pattern
    return ____
```

4. Write the training step:

```
def training_step(model, optimizer, criterion, images, labels):
    # Your code here - complete training step
    return loss
```

5. Write character to one-hot conversion:

```
def char_to_onehot(char, char2idx, vocab_size):
    # Your code here - convert character to one-hot vector
    return ____
```

FINAL MEMORY CHECK

Last Minute Review

Problem Last Minute Review

Before the exam, quickly verify you remember:

Constants (write from memory):

- CIFAR-100 mean: [____, ____, ____]
- CIFAR-100 std: [____, ____, ____]

- Train/val split ratio: ____
- Dropout probability: ____
- Weight decay: ____
- SGD momentum: ____

Dimensions (write from memory):

- CNN channels: 3→____→____→____→____
- CNN spatial: 32→____→____→____
- RNN W_xh: (____, ____)
- RNN W_hh: (____, ____)
- RNN W_hy: (____, ____)

PRACTICE EXAM - NO PEEKING AT ANSWERS!

Practice 1: Bilinear Interpolation Implementation

Problem Practice 1: Bilinear Interpolation Implementation

Complete this bilinear interpolation function:

```
def bilinear_interpolate(a_l, q_y, q_x):
    H, W = a_l.shape

    y0 = int(np.______(q_y))
    x0 = int(np.______(q_x))
    y1 = y0 + ___
    x1 = x0 + ___

    def get_pixel_value(img, y, x):
        if 0 <= y < H and 0 <= x < W:
            return img[y, x]
        else:
            return ____

    v_00 = get_pixel_value(a_l, y0, x0)
    v_01 = get_pixel_value(a_l, y0, x1)
    v_10 = get_pixel_value(a_l, y1, x0)
    v_11 = get_pixel_value(a_l, y1, x1)

    dy = q_y - ___
    dx = q_x - ___

    v_0 = v_00 * (1 - ___) + v_01 * ___
    v_1 = v_10 * (1 - ___) + v_11 * ___
    out = v_0 * (1 - ___) + v_1 * ___

    return out
```

Practice 2: Deformable Conv Offset Extraction

Problem Practice 2: Deformable Conv Offset Extraction

Fill in the correct offset extraction pattern:

```
for kh in range(K_h):
    for kw in range(K_w):
        k = kh * K_w + kw

        delta_y = delta[n, ___ * k, h_out, w_out]
        delta_x = delta[n, ___ * k + ___, h_out, w_out]
        m_k = mask[n, k, h_out, w_out]

        sample_y = h_start + kh * dilation + ____
        sample_x = w_start + kw * dilation + ____
```

```
        # Apply weight and mask
        value += weight[c_out, c_in, kh, kw] * ___ * ____
```

Practice 3: CNN Architecture

Problem Practice 3: CNN Architecture

Complete the CustomCNN constructor with correct channel progression:

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

        self.conv1 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(___, ___, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(___, ___, kernel_size=3, padding=1)

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

        self.fc1 = nn.Linear(___ * ___ * ___, 512)
        self.fc2 = nn.Linear(___, ___)
        self.fc3 = nn.Linear(___, ___)

        self.dropout = nn.Dropout(____)
```

Practice 4: CNN Forward Pass

Problem Practice 4: CNN Forward Pass

Complete the forward pass with correct activations:

```
def forward(self, x):
    x = F.____(self.conv1(x))
    x = F.____(self.conv2(x))
    x = self.pool(x)  # 32x32 -> ____

    x = F.____(self.conv3(x))
    x = self.pool(x)  # 16x16 -> ____

    x = F.____(self.conv4(x))
    x = self.pool(x)  # 8x8 -> ____

    x = x.view(x.____(__), ___)

    x = F.____(self.fc1(x))
    x = self.______(x)
```

```
    x = F.____(self.fc2(x))
    x = self.______(x)
    x = self.fc3(x)

    return x
```

Practice 5: Training Loop Pattern

Problem Practice 5: Training Loop Pattern

Complete the sacred training loop:

```
def train_step(model, optimizer, criterion, images, labels, device):
    images = images.to(____)
    labels = labels.to(____)

    optimizer.____()
    outputs = model(______)
    loss = criterion(outputs, labels)
    loss.______()
    optimizer.____()

    return loss.item()
```

Practice 6: Top-K Accuracy

Problem Practice 6: Top-K Accuracy

Complete the top-1 and top-5 accuracy calculation:

```
# Top-1 accuracy
_, top1_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top1_correct = top1_pred.eq(labels.view(___, ___)).sum().item()

# Top-5 accuracy
_, top5_pred = outputs.topk(___, dim=1, largest=___, sorted=___)
top5_correct = top5_pred.eq(labels.view(-1, 1).____(____))).sum().item()
```

Practice 7: RNN Weight Initialization

Problem Practice 7: RNN Weight Initialization

Initialize RNN parameters with correct dimensions:

```
V = len(chars)  # Vocabulary size
H = 16          # Hidden size

W_xh = torch.randn(___, ___, requires_grad=True) * 0.1
W_hh = torch.randn(___, ___, requires_grad=True) * 0.1
b_xh = torch.zeros(___, requires_grad=True)
b_hh = torch.zeros(___, requires_grad=True)
W_hy = torch.randn(___, ___, requires_grad=True) * 0.1
b_y = torch.zeros(___, requires_grad=True)
```

Practice 8: RNN Forward Equations

Problem Practice 8: RNN Forward Equations

Complete the RNN forward pass:

```
for t in range(seq_len):
    x_t = inputs[t]

    # Hidden state update
    h = torch.tanh(_____ @ x_t + _____ + _____ @ h + _____)

    # Output logits
    s_t = _____ @ h + _____

    logits_list.append(s_t)
```

Practice 9: Character Processing

Problem Practice 9: Character Processing

Complete the character-level preprocessing:

```
text = "Deep Learning"

chars = sorted(list(set(____)))
char2idx = {ch: i for i, ch in enumerate(____)}
idx2char = {i: ch for i, ch in enumerate(____)}

input_seq = text[____]
target_seq = text[____]

def one_hot(idx, size):
    vec = torch.zeros(size)
    vec[____] = 1.0
    return vec
```

```
inputs = [one_hot(char2idx[ch], V) for ch in ____]
targets = torch.tensor([char2idx[ch] for ch in ____], dtype=torch.long)
```

Practice 10: Gradient Computation

Problem Practice 10: Gradient Computation

Complete the explicit gradient computation:

```
grad_W_xh = torch.autograd.grad(loss_manual, W_xh, retain_graph=____)[0]
grad_W_hh = torch.autograd.grad(loss_manual, W_hh, retain_graph=____)[0]
grad_b_xh = torch.autograd.grad(loss_manual, b_xh, retain_graph=____)[0]
grad_b_hh = torch.autograd.grad(loss_manual, b_hh, retain_graph=____)[0]
grad_W_hy = torch.autograd.grad(loss_manual, W_hy, retain_graph=____)[0]
grad_b_y = torch.autograd.grad(loss_manual, b_y, retain_graph=____)[0]
```

Practice 11: BatchNorm Integration

Problem Practice 11: BatchNorm Integration

Add BatchNorm to CNN architecture:

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

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(____)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(____)

    def forward(self, x):
        x = F.relu(self.______(self.conv1(x)))
        x = F.relu(self.______(self.conv2(x)))
```

Practice 12: CIFAR-100 Constants

Problem Practice 12: CIFAR-100 Constants

Fill in the CIFAR-100 normalization values:

```
transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize(mean=[____, ____, ____],
                         std=[____, ____, ____])
])
```

Practice 13: Data Splitting

Problem Practice 13: Data Splitting

Complete the 80/20 train/validation split:

```
full_train_set = CIFAR100(root='./data', train=True, download=True)

train_size = int(____ * len(full_train_set))
val_size = len(full_train_set) - ____

train_set, val_set = random_split(full_train_set, [____, ____])
```

Practice 14: Loss and Optimizer Setup

Problem Practice 14: Loss and Optimizer Setup

Set up loss function and optimizer for CIFAR-100:

```
loss_function = nn.______()

optimizer = optim.SGD(model.parameters(),
                      lr=____,
                      momentum=____,
                      weight_decay=____)
```

Practice 15: Quick Recall

Problem Practice 15: Quick Recall

Answer these without looking:

1. What function rounds down to nearest integer? __________
2. In PyTorch deformable conv, Y offset comes at index ____ and X offset at ____
3. The bilinear interpolation weight formula is: G(p,q) = ____________________

4. CNN channel progression in our model: 3 → ___ → ___ → ___ → ___

5. RNN hidden state uses which activation function? __________

6. In training loop, what comes after loss.backward()? __________

7. For validation, you wrap the loop with torch.__________():

8. The flatten operation uses x.______(x.______(0), ____):

9. Top-5 accuracy uses outputs.______(___, dim=1):

10. RNN weight W_hy has dimensions (___, ___):

CHALLENGE PROBLEMS

Challenge 1: Complete Function Implementation

Problem Challenge 1: Complete Function Implementation

Write the complete bilinear interpolation function from scratch:

Challenge 2: Complete RNN Step

Problem Challenge 2: Complete RNN Step

Write a complete RNN forward step function:

Challenge 3: Training vs Validation

Problem Challenge 3: Training vs Validation

Write the key differences between training and validation loops:

Problem Challenge 4: Debugging Scenario

Your deformable convolution outputs don't match PyTorch. List 5 most likely bugs:

1. ____
2. ____
3. ____
4. ____
5. ____

Problem Challenge 5: Architecture Design

Design a deeper CNN with 6 conv layers following the same pattern:

ANSWER BANK

Practice 1 Answers:

Problem Practice 1 Answers:

floor, floor, 1, 1, 0.0, y0, x0, dx, dx, dx, dx, dy, dy

Practice 2 Answers:

Problem Practice 2 Answers:

2, 2, 1, delta_y, delta_x, m_k, interpolated

Practice 3 Answers:

Problem Practice 3 Answers:

3, 32, 32, 64, 64, 128, 128, 256, MaxPool2d, 256, 4, 4, 512, 256, 256, 100, 0.5

Practice 4 Answers:

Problem Practice 4 Answers:

relu, relu, 16x16, relu, 8x8, relu, 4x4, size, 0, -1, relu, dropout, relu, dropout

Practice 5 Answers:

Problem Practice 5 Answers:

device, device, zero_grad, images, backward, step

Practice 6 Answers:

Problem Practice 6 Answers:

1, True, True, -1, 1, 5, True, True, expand_as(top5_pred)

Practice 7 Answers:

Problem Practice 7 Answers:

H, V, H, H, H, H, V, H, V

Practice 8 Answers:

Problem Practice 8 Answers:

W_xh, b_xh, W_hh, b_hh, W_hy, b_y

Practice 9 Answers:

Problem Practice 9 Answers:

text, chars, chars, [:-1], [1:], idx, input_seq, target_seq

Practice 10 Answers:

Problem Practice 10 Answers:

True, True, True, True, True, True

Practice 11 Answers:

Problem Practice 11 Answers:

32, 64, bn1, bn2

Practice 12 Answers:

Problem Practice 12 Answers:

0.5071, 0.4867, 0.4408, 0.2675, 0.2565, 0.2761

Practice 13 Answers:

Problem Practice 13 Answers:

0.8, train_size, train_size, val_size

Practice 14 Answers:

Problem Practice 14 Answers:

CrossEntropyLoss, 0.01, 0.9, 5e-4

Practice 15 Answers:

Problem Practice 15 Answers:

1. np.floor 2. 2*k, 2*k+1 3. (1-—px-qx—)*(1-—py-qy—) 4. 32, 64, 128, 256 5. tanh 6. optimizer.step() 7. no_grad 8. view, size, -1 9. topk, 5 10. V, H

Challenge Problem Answers:

Problem Challenge Problem Answers:

Challenge 1: Complete bilinear function with all bounds checking, corner calculations, and interpolation steps.

Challenge 2: RNN step with tanh activation: h = torch.tanh(W_xh @ x + b_xh + W_hh @ h_prev + b_hh)

Challenge 3: Training: model.train(), optimizer steps, gradients enabled. Validation: model.eval(), torch.no_grad(), no optimizer steps.

Challenge 4: 1. Wrong offset indexing (2*k), 2. Wrong bilinear order (X first), 3. Missing zero padding, 4. Wrong sampling position calculation, 5. Incorrect weight/mask application.

Challenge 5: Conv layers: 3→32→32→64→64→128→128, with pooling after every 2 conv layers.

CENG403 - Spring 2025: Homework set Write Code Blocks Practice

Your Name

TASK 1: DEFORMABLE CNN - WRITE CODE BLOCKS

.1: Floor Operation

Problem 1.1: Floor Operation

Write code to get the floor of a float value q_y and store it in y0:

.2: Ceiling Operation

Problem 1.2: Ceiling Operation

Write code to get the ceiling of a float value q_x and store it in x1:

.3: Kernel Index Calculation

Problem 1.3: Kernel Index Calculation

Write code to calculate linear kernel index from 2D position (kh, kw) with kernel width K_w:

.4: Y Offset Extraction

Problem 1.4: Y Offset Extraction

Write code to extract Y offset from delta tensor for batch n, kernel index k, output position (h_out, w_out):

.5: X Offset Extraction

Problem 1.5: X Offset Extraction

Write code to extract X offset from delta tensor for batch n, kernel index k, output position (h_out, w_out):

.6: Base Position Calculation

Problem 1.6: Base Position Calculation

Write code to calculate base sampling position h_start from output position h_out and stride:

.7: Final Sampling Position

Problem 1.7: Final Sampling Position

Write code to calculate final sampling position sample_y from h_start, kh, dilation, and delta_y:

.8: Bounds Check Condition

Problem 1.8: Bounds Check Condition

Write code to check if coordinates (y, x) are within image bounds (H, W):

.9: Safe Pixel Access

Problem 1.9: Safe Pixel Access

Write code to get pixel value at (y, x) from image img, returning 0.0 if out of bounds:

.10: Fractional Part Calculation

Problem 1.10: Fractional Part Calculation

Write code to calculate fractional part dx from q_x and its floor x0:

.11: Linear Interpolation

Problem 1.11: Linear Interpolation

Write code to linearly interpolate between v_left and v_right using weight dx:

.12: Bilinear Weight Calculation

Problem 1.12: Bilinear Weight Calculation

Write code to calculate bilinear interpolation weight for points p and q:

.13: Value Accumulation

Problem 1.13: Value Accumulation

Write code to accumulate weighted interpolated value to current_value using weight, mask, and interpolated:

.14: Corner Position Calculation

Problem 1.14: Corner Position Calculation

Write code to calculate all four corner positions (y0, x0), (y0, x1), (y1, x0), (y1, x1) from q_y and q_x:

.15: Mask Application

Problem 1.15: Mask Application

Write code to apply modulation mask m_k to an interpolated value:

TASK 2: CNN PYTORCH - WRITE CODE BLOCKS

.1: Conv Layer Definition

Problem 2.1: Conv Layer Definition

Write code to define a Conv2d layer with 32 input channels, 64 output channels, kernel size 3, padding 1:

.2: BatchNorm Layer Definition

Problem 2.2: BatchNorm Layer Definition

Write code to define a BatchNorm2d layer for 128 channels:

.3: MaxPool Layer Definition

Problem 2.3: MaxPool Layer Definition

Write code to define a MaxPool2d layer that halves spatial dimensions:

.4: Linear Layer Definition

Problem 2.4: Linear Layer Definition

Write code to define a Linear layer with 4096 inputs and 512 outputs:

.5: Dropout Layer Definition

Problem 2.5: Dropout Layer Definition

Write code to define a Dropout layer with probability 0.5:

.6: ReLU Activation

Problem 2.6: ReLU Activation

Write code to apply ReLU activation to variable x:

.7: Tensor Flattening

Problem 2.7: Tensor Flattening

Write code to flatten tensor x while preserving batch dimension:

.8: Device Transfer

Problem 2.8: Device Transfer

Write code to move tensor images to device:

.9: Optimizer Zero Grad

Problem 2.9: Optimizer Zero Grad

Write code to clear gradients from optimizer:

.10: Forward Pass

Problem 2.10: Forward Pass

Write code to perform forward pass through model with input images:

.11: Loss Calculation

Problem 2.11: Loss Calculation

Write code to calculate loss using criterion with outputs and labels:

.12: Backward Pass

Problem 2.12: Backward Pass

Write code to perform backward pass on loss:

.13: Optimizer Step

Problem 2.13: Optimizer Step

Write code to update model parameters using optimizer:

.14: Top-1 Prediction

Problem 2.14: Top-1 Prediction

Write code to get top-1 predictions from outputs:

.15: Accuracy Calculation

Problem 2.15: Accuracy Calculation

Write code to calculate accuracy from predicted and labels tensors:

.16: Model Training Mode

Problem 2.16: Model Training Mode

Write code to set model to training mode:

.17: Model Evaluation Mode

Problem 2.17: Model Evaluation Mode

Write code to set model to evaluation mode:

.18: No Gradient Context

Problem 2.18: No Gradient Context

Write code to create context where gradients are disabled:

.19: Dataset Size Calculation

Problem 2.19: Dataset Size Calculation

Write code to calculate training size as 80% of total dataset size:

.20: Random Split

Problem 2.20: Random Split

Write code to split dataset into train_set and val_set with sizes train_size and val_size:

TASK 3: RNN - WRITE CODE BLOCKS

.1: Character Set Creation

Problem 3.1: Character Set Creation

Write code to create sorted list of unique characters from text:

.2: Character to Index Mapping

Problem 3.2: Character to Index Mapping

Write code to create dictionary mapping each character to its index:

.3: Index to Character Mapping

Problem 3.3: Index to Character Mapping

Write code to create dictionary mapping each index to its character:

.4: Input Sequence Creation

Problem 3.4: Input Sequence Creation

Write code to create input sequence (all characters except last) from text:

.5: Target Sequence Creation

Problem 3.5: Target Sequence Creation

Write code to create target sequence (all characters except first) from text:

.6: One-Hot Vector Creation

Problem 3.6: One-Hot Vector Creation

Write code to create one-hot vector of given size with 1 at given index:

.7: Input-to-Hidden Weight Initialization

Problem 3.7: Input-to-Hidden Weight Initialization

Write code to initialize W_xh weight matrix for RNN with hidden size H and vocab size V:

.8: Hidden-to-Hidden Weight Initialization

Problem 3.8: Hidden-to-Hidden Weight Initialization

Write code to initialize W_hh weight matrix for RNN with hidden size H:

.9: Hidden Bias Initialization

Problem 3.9: Hidden Bias Initialization

Write code to initialize bias vector b_xh for hidden layer with size H:

.10: Output Weight Initialization

Problem 3.10: Output Weight Initialization

Write code to initialize W_hy weight matrix from hidden to output with vocab size V and hidden size H:

.11: Hidden State Initialization

Problem 3.11: Hidden State Initialization

Write code to initialize hidden state vector with zeros of size H:

.12: Input Contribution Calculation

Problem 3.12: Input Contribution Calculation

Write code to calculate input contribution to hidden state using W_xh and x_t:

.13: Hidden Recurrence Calculation

Problem 3.13: Hidden Recurrence Calculation

Write code to calculate recurrent contribution to hidden state using W_hh and previous hidden state h:

.14: Hidden State Update

Problem 3.14: Hidden State Update

Write code to update hidden state using tanh activation with all contributions and biases:

.15: Output Logits Calculation

Problem 3.15: Output Logits Calculation

Write code to calculate output logits s_t from hidden state h using W_hy and b_y:

.16: Character Index Lookup

Problem 3.16: Character Index Lookup

Write code to get index of character 'e' from char2idx dictionary:

.17: Input List Creation

Problem 3.17: Input List Creation

Write code to convert input sequence to list of one-hot vectors:

.18: Target Tensor Creation

Problem 3.18: Target Tensor Creation

Write code to convert target sequence to tensor of character indices:

.19: Logits Stacking

Problem 3.19: Logits Stacking

Write code to stack list of logits tensors into single tensor:

.20: Log Softmax Calculation

Problem 3.20: Log Softmax Calculation

Write code to calculate log softmax of logits along vocabulary dimension:

.21: NLL Loss Calculation

Problem 3.21: NLL Loss Calculation

Write code to calculate negative log likelihood loss from log_probs and targets:

.22: Gradient Calculation

Problem 3.22: Gradient Calculation

Write code to calculate gradient of loss with respect to W_xh:

DATA PREPROCESSING - WRITE CODE BLOCKS

.1: CIFAR-100 Mean Values

Problem 4.1: CIFAR-100 Mean Values

Write code to define CIFAR-100 normalization mean values:

.2: CIFAR-100 Std Values

Problem 4.2: CIFAR-100 Std Values

Write code to define CIFAR-100 normalization standard deviation values:

.3: ToTensor Transform

Problem 4.3: ToTensor Transform

Write code to create ToTensor transform:

.4: Normalization Transform

Problem 4.4: Normalization Transform

Write code to create Normalize transform with CIFAR-100 mean and std:

.5: Random Crop Transform

Problem 4.5: Random Crop Transform

Write code to create RandomCrop transform with size 32 and padding 4:

.6: Random Flip Transform

Problem 4.6: Random Flip Transform

Write code to create RandomHorizontalFlip transform:

.7: Transform Composition

Problem 4.7: Transform Composition

Write code to compose multiple transforms into single transform:

.8: CIFAR-100 Dataset Loading

Problem 4.8: CIFAR-100 Dataset Loading

Write code to load CIFAR-100 training dataset with transform:

.9: DataLoader Creation

Problem 4.9: DataLoader Creation

Write code to create DataLoader with batch size 128 and shuffle=True:

OPTIMIZATION AND TRAINING - WRITE CODE BLOCKS

.1: CrossEntropy Loss Definition

Problem 5.1: CrossEntropy Loss Definition

Write code to define CrossEntropy loss function:

.2: SGD Optimizer Definition

Problem 5.2: SGD Optimizer Definition

Write code to define SGD optimizer with learning rate 0.01 and momentum 0.9:

.3: Adam Optimizer Definition

Problem 5.3: Adam Optimizer Definition

Write code to define Adam optimizer with learning rate 0.001:

.4: Model Parameter Count

Problem 5.4: Model Parameter Count

Write code to count total number of parameters in model:

.5: Learning Rate Update

Problem 5.5: Learning Rate Update

Write code to multiply learning rate by 0.1 for all parameter groups:

.6: Model State Save

Problem 5.6: Model State Save

Write code to save model state dictionary to file 'model.pth':

.7: Model State Load

Problem 5.7: Model State Load

Write code to load model state dictionary from file 'model.pth':

.8: Gradient Clipping

Problem 5.8: Gradient Clipping

Write code to clip gradients to maximum norm of 1.0:

3cm

.9: Top-5 Accuracy

Problem 5.9: Top-5 Accuracy

Write code to calculate top-5 accuracy from outputs and labels:

Problem 5.10: Loss Item Extraction

Write code to extract scalar value from loss tensor:

DEBUGGING AND UTILITIES - WRITE CODE BLOCKS

.1: Tensor Shape Check

Problem 6.1: Tensor Shape Check

Write code to print shape of tensor x:

.2: Tensor Device Check

Problem 6.2: Tensor Device Check

Write code to check which device tensor x is on:

.3: Model Device Transfer

Problem 6.3: Model Device Transfer

Write code to move entire model to GPU:

.4: Gradient Existence Check

Problem 6.4: Gradient Existence Check

Write code to check if parameter has gradients:

.5: Memory Usage Check

Problem 6.5: Memory Usage Check

Write code to check CUDA memory usage:

.6: Random Seed Setting

Problem 6.6: Random Seed Setting

Write code to set PyTorch random seed to 42:

.7: Numpy Seed Setting

Problem 6.7: Numpy Seed Setting

Write code to set numpy random seed to 42:

.8: Model Summary

Problem 6.8: Model Summary

Write code to print model architecture:

.9: Batch Dimension Check

Problem 6.9: Batch Dimension Check

Write code to get batch size from tensor x:

Problem 6.10: Tensor Type Conversion

Write code to convert tensor x to float type:

ANSWER BANK

Task 1 - Deformable CNN Answers:

Problem Task 1 - Deformable CNN Answers:

1.1: `y0 = int(np.floor(q_y))`

1.2: `x1 = int(np.ceil(q_x))`

1.3: `k = kh * K_w + kw`

1.4: `delta_y = delta[n, 2 * k, h_out, w_out]`

1.5: `delta_x = delta[n, 2 * k + 1, h_out, w_out]`

1.6: `h_start = h_out * stride`

1.7: `sample_y = h_start + kh * dilation + delta_y`

1.8: `if 0 <= y < H and 0 <= x < W:`

1.9: `value = img[y, x] if (0 <= y < H and 0 <= x < W) else 0.0`

1.10: `dx = q_x - x0`

1.11: `result = v_left * (1 - dx) + v_right * dx`

1.12: `weight = (1 - abs(p_x - q_x)) * (1 - abs(p_y - q_y))`

1.13: `current_value += weight * mask * interpolated`

1.14: `y0, x0 = int(np.floor(q_y)), int(np.floor(q_x))`
`y1, x1 = y0 + 1, x0 + 1`

1.15: `modulated_value = m_k * interpolated_value`

Task 2 - CNN PyTorch Answers:

Problem Task 2 - CNN PyTorch Answers:

2.1: `self.conv = nn.Conv2d(32, 64, kernel_size=3, padding=1)`

2.2: `self.bn = nn.BatchNorm2d(128)`

2.3: `self.pool = nn.MaxPool2d(2, 2)`

2.4: `self.fc = nn.Linear(4096, 512)`

2.5: `self.dropout = nn.Dropout(0.5)`

2.6: `x = F.relu(x)`

2.7: `x = x.view(x.size(0), -1)`

2.8: `images = images.to(device)`

2.9: `optimizer.zero_grad()`

2.10: `outputs = model(images)`

2.11: `loss = criterion(outputs, labels)`

2.12: `loss.backward()`

2.13: `optimizer.step()`

2.14: `_, predicted = torch.max(outputs, 1)`

2.15: `accuracy = (predicted == labels).sum().item() / labels.size(0) * 100`

2.16: `model.train()`

2.17: `model.eval()`

2.18: `with torch.no_grad():`

2.19: `train_size = int(0.8 * len(dataset))`

2.20: `train_set, val_set = random_split(dataset, [train_size, val_size])`

Task 3 - RNN Answers:

Problem Task 3 - RNN Answers:

3.1: `chars = sorted(list(set(text)))`

3.2: `char2idx = {ch: i for i, ch in enumerate(chars)}`

3.3: `idx2char = {i: ch for i, ch in enumerate(chars)}`

3.4: `input_seq = text[:-1]`

3.5: `target_seq = text[1:]`

3.6: `vec = torch.zeros(size)`
`vec[idx] = 1.0`

3.7: `W_xh = torch.randn(H, V, requires_grad=True) * 0.1`

3.8: `W_hh = torch.randn(H, H, requires_grad=True) * 0.1`

3.9: `b_xh = torch.zeros(H, requires_grad=True)`

3.10: `W_hy = torch.randn(V, H, requires_grad=True) * 0.1`

3.11: `h = torch.zeros(H)`

3.12: `input_contrib = W_xh @ x_t`

3.13: `hidden_contrib = W_hh @ h`

3.14: `h = torch.tanh(W_xh @ x_t + b_xh + W_hh @ h + b_hh)`

3.15: `s_t = W_hy @ h + b_y`

3.16: `idx = char2idx['e']`

3.17: `inputs = [one_hot(char2idx[ch], V) for ch in input_seq]`

3.18: `targets = torch.tensor([char2idx[ch] for ch in target_seq], dtype=torch.long)`

3.19: `logits = torch.stack(logits_list)`

3.20: `log_probs = F.log_softmax(logits, dim=1)`

3.21: `loss = F.nll_loss(log_probs, targets)`

3.22: `grad_W_xh = torch.autograd.grad(loss, W_xh, retain_graph=True)[0]`

Additional Sections Answers:

Problem Additional Sections Answers:

4.1: `mean = [0.5071, 0.4867, 0.4408]`

4.2: `std = [0.2675, 0.2565, 0.2761]`

4.3: `transform = transforms.ToTensor()`

4.4: `normalize = transforms.Normalize(mean=[0.5071, 0.4867, 0.4408], std=[0.2675, 0.2565, 0.2761])`

4.5: `crop = transforms.RandomCrop(32, padding=4)`

4.6: `flip = transforms.RandomHorizontalFlip()`

4.7: `transform = transforms.Compose([transform1, transform2, ...])`

4.8: `dataset = CIFAR100(root='./data', train=True, transform=transform)`

4.9: `loader = DataLoader(dataset, batch_size=128, shuffle=True)`

5.1: `criterion = nn.CrossEntropyLoss()`

5.2: `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)`

5.3: `optimizer = optim.Adam(model.parameters(), lr=0.001)`

5.4: `total_params = sum(p.numel() for p in model.parameters())`

5.5: `for param_group in optimizer.param_groups:  param_group['lr'] *= 0.1`

5.6: `torch.save(model.state_dict(), 'model.pth')`

5.7: `model.load_state_dict(torch.load('model.pth'))`

5.8: `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)`

5.9: `_, top5_pred = outputs.topk(5, dim=1)`
`top5_acc = top5_pred.eq(labels.view(-1, 1).expand_as(top5_pred)).sum().item()`

5.10: `loss_value = loss.item()`

6.1: `print(x.shape)`

6.2: `print(x.device)`

6.3: `model = model.to('cuda')`

6.4: `if param.grad is not None:`

6.5: `print(torch.cuda.memory_allocated())`

6.6: `torch.manual_seed(42)`

6.7: `np.random.seed(42)`

6.8: `print(model)`

6.9: `batch_size = x.size(0)`

6.10: `x = x.float()`

Additional Answers

Problem Additional Answers

C1: 32 x 32, C2: 8 x 8, C3: 4096, C4: 73856, C5: 256

S1: numpy, torch.nn, F, transforms

S2: nn.Module, super

S3: def forward(self, x)

S4: nn.CrossEntropyLoss()

S5: optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

M1: x = self.dropout(x) if self.training else x

M2: for param_group in optimizer.param_groups: param_group['lr'] *= 0.1

M3: torch.save(model.state_dict(), 'model.pth')

M4: outputs = model(images.to(device))

ADVANCED DEFORMABLE CNN OPERATIONS

D1

Problem D1

Write code to calculate output dimensions after deformable convolution:

D2

Problem D2

Write code to pad input tensor with zeros on all sides by padding amount:

D3

Problem D3

Write code to initialize output tensor for deformable convolution with correct shape:

D4

Problem D4

Write code to extract four corner values for bilinear interpolation given positions:

D5

Problem D5

Write code to apply dilation to kernel position in deformable convolution:

D6

Problem D6

Write code to accumulate convolution result across all input channels:

D7

Problem D7

Write code to validate that sampling position is fractional:

D8

Problem D8

Write code to compute weighted sum for deformable convolution at one position:

ADVANCED CNN OPERATIONS

A1

Problem A1

Write code to calculate receptive field size after multiple conv layers:

A2

Problem A2

Write code to implement skip connection (residual connection):

A3

Problem A3

Write code to apply different transforms for training vs validation:

A4

Problem A4

Write code to calculate model memory usage:

A5

Problem A5

Write code to freeze specific layers during training:

A6

Problem A6

Write code to implement learning rate scheduling:

A7

Problem A7

Write code to calculate class-wise accuracy:

A8

Problem A8

Write code to implement early stopping condition:

A9

Problem A9

Write code to apply different dropout rates for different layers:

A10

Problem A10

Write code to implement gradient accumulation for large batches:

ADVANCED RNN OPERATIONS

R1

Problem R1

Write code to handle variable length sequences in RNN:

R2

Problem R2

Write code to implement bidirectional RNN forward pass:

R3

Problem R3

Write code to calculate perplexity from RNN loss:

R4

Problem R4

Write code to implement teacher forcing during training:

R5

Problem R5

Write code to generate text using trained RNN:

R6

Problem R6

Write code to implement attention mechanism for RNN:

R7

Problem R7

Write code to handle padding in RNN sequences:

R8

Problem R8

Write code to implement LSTM cell from scratch:

R9

Problem R9

Write code to calculate gradient flow through time steps:

R10

Problem R10

Write code to implement sequence-to-sequence mapping:

ERROR HANDLING AND DEBUGGING

E1

Problem E1

Write code to check if tensors are on the same device:

E2

Problem E2

Write code to handle CUDA out of memory error:

E3

Problem E3

Write code to validate input tensor shapes before processing:

E4

Problem E4

Write code to check for NaN values in gradients:

E5

Problem E5

Write code to log training progress every N steps:

E6

Problem E6

Write code to handle empty batch gracefully:

E7

Problem E7

Write code to validate model output dimensions:

E8

Problem E8

Write code to catch and handle gradient explosion:

E9

Problem E9

Write code to verify learning rate is positive:

E10

Problem E10

Write code to check if model is in correct mode for evaluation:

MODEL EVALUATION AND METRICS

V1

Problem V1

Write code to calculate precision for multi-class classification:

V2

Problem V2

Write code to calculate recall for specific class:

V3

Problem V3

Write code to compute F1-score from precision and recall:

V4

Problem V4

Write code to create confusion matrix:

V5

Problem V5

Write code to calculate mean average precision:

V6

Problem V6

Write code to implement cross-validation split:

V7

Problem V7

Write code to calculate model inference time:

V8

Problem V8

Write code to compute per-class accuracy:

V9

Problem V9

Write code to calculate balanced accuracy:

V10

Problem V10

Write code to evaluate model on subset of classes:

DATA LOADING VARIATIONS

L1

Problem L1

Write code to create weighted sampler for imbalanced dataset:

L2

Problem L2

Write code to implement custom collate function:

L3

Problem L3

Write code to handle corrupted data samples:

L4

Problem L4

Write code to implement data augmentation pipeline:

L5

Problem L5

Write code to create stratified train/val split:

L6

Problem L6

Write code to implement multi-scale image loading:

L7

Problem L7

Write code to balance dataset using oversampling:

L8

Problem L8

Write code to implement k-fold cross validation data split:

L9

Problem L9

Write code to create data loader with custom worker init:

L10

Problem L10

Write code to implement online data augmentation:

OPTIMIZATION TECHNIQUES

O1

Problem O1

Write code to implement cosine annealing learning rate:

O2

Problem O2

Write code to add L1 regularization to loss:

O3

Problem O3

Write code to implement momentum SGD from scratch:

O4

Problem O4

Write code to apply different learning rates to different layers:

O5

Problem O5

Write code to implement AdamW optimizer setup:

O6

Problem O6

Write code to implement linear warmup schedule:

O7

Problem O7

Write code to add noise to gradients:

O8

Problem O8

Write code to implement cyclical learning rates:

O9

Problem O9

Write code to calculate effective learning rate:

O10

Problem O10

Write code to implement gradient centralization:

TENSOR OPERATIONS

T1

Problem T1

Write code to reshape tensor while preserving total elements:

T2

Problem T2

Write code to concatenate tensors along specific dimension:

T3

Problem T3

Write code to split tensor into equal chunks:

T4

Problem T4

Write code to transpose last two dimensions:

T5

Problem T5

Write code to compute element-wise maximum of two tensors:

T6

Problem T6

Write code to select top-k elements along dimension:

T7

Problem T7

Write code to create mask for padding tokens:

T8

Problem T8

Write code to compute pairwise distances between vectors:

T9

Problem T9

Write code to normalize tensor to unit length:

T10

Problem T10

Write code to apply sliding window operation:

MEMORY AND PERFORMANCE

P1

Problem P1

Write code to enable mixed precision training:

P2

Problem P2

Write code to clear GPU cache:

P3

Problem P3

Write code to profile memory usage:

P4

Problem P4

Write code to implement checkpointing for memory efficiency:

P5

Problem P5

Write code to use torch.no$_grad()forinference$:

P6

Problem P6

Write code to pin memory for faster data loading:

P7

Problem P7

Write code to set number of threads for CPU operations:

P8

Problem P8

Write code to benchmark model inference speed:

P9

Problem P9

Write code to implement gradient checkpointing:

P10

Problem P10

Write code to optimize model for inference:

EXTENDED ANSWER BANK

Advanced Deformable CNN Answers:

Problem Advanced Deformable CNN Answers:

D1: `H_out = (H_in + 2*padding - dilation*(K_h-1) - 1) // stride + 1`

D2: `padded_input = F.pad(input, (padding, padding, padding, padding))`

D3: `output = np.zeros((N, C_out, H_out, W_out), dtype=np.float32)`

D4: `v_00, v_01 = get_pixel(img, y0, x0), get_pixel(img, y0, x1)`
`v_10, v_11 = get_pixel(img, y1, x0), get_pixel(img, y1, x1)`

D5: `dilated_pos_y = h_start + kh * dilation`

D6: `for c_in in range(C_in):  value += weight[c_out, c_in, kh, kw] * interpolated[c_in]`

D7: `assert sample_y != int(sample_y) or sample_x != int(sample_x)`

D8: `result = sum(w[k] * m[k] * bilinear_interp(input, pos[k]) for k in range(K))`

Advanced CNN Answers:

Problem Advanced CNN Answers:

A1: `receptive_field = ((kernel_size - 1) * dilation + 1)`

A2: `x = F.relu(self.conv(x) + x)` # residual connection

A3: `transform = train_transform if self.training else val_transform`

A4: `memory_usage = sum(p.numel() * p.element_size() for p in model.parameters())`

A5: `for param in model.layer.parameters():  param.requires_grad = False`

A6: `scheduler.step(); current_lr = scheduler.get_last_lr()[0]`

A7: `per_class_acc = [(pred==i).sum()/(labels==i).sum() for i in range(num_classes)]`

A8: `if val_loss > best_loss + patience_delta:  stop_training = True`

A9: `self.dropout1 = nn.Dropout(0.2); self.dropout2 = nn.Dropout(0.5)`

A10: `if (step + 1) % accumulation_steps == 0:  optimizer.step(); optimizer.zero_grad()`

Advanced RNN Answers:

Problem Advanced RNN Answers:

R1: `packed_seq = nn.utils.rnn.pack_padded_sequence(x, lengths, batch_first=True)`

R2: `h_forward = rnn_forward(x); h_backward = rnn_backward(x[:,::-1])`

R3: `perplexity = torch.exp(loss)`

R4: `decoder_input = target[:-1] if training else previous_output`

R5: `with torch.no_grad():  output = model.generate(start_token, max_length)`

R6: `attention_weights = F.softmax(torch.matmul(query, keys.T), dim=-1)`

R7: `mask = (sequence != pad_token).float().unsqueeze(-1)`

R8: `f_gate = torch.sigmoid(W_f @ x + U_f @ h + b_f)`

R9: `grad_h = torch.autograd.grad(loss, hidden_states, retain_graph=True)`

R10: `decoder_output = decoder(encoder_output, target_sequence)`

Additional Sections Available Upon Request...

Problem Additional Sections Available Upon Request...

CENG403 - Spring 2025: Homework set Write Code Blocks Practice

Your Name

TASK 1: DEFORMABLE CNN - WRITE CODE BLOCKS

.1: Floor Operation

Problem 1.1: Floor Operation

Write code to get the floor of a float value q_y and store it in y0:

.2: Ceiling Operation

Problem 1.2: Ceiling Operation

Write code to get the ceiling of a float value q_x and store it in x1:

.3: Kernel Index Calculation

Problem 1.3: Kernel Index Calculation

Write code to calculate linear kernel index from 2D position (kh, kw) with kernel width K_w:

.4: Y Offset Extraction

Problem 1.4: Y Offset Extraction

Write code to extract Y offset from delta tensor for batch n, kernel index k, output position (h_out, w_out):

.5: X Offset Extraction

Problem 1.5: X Offset Extraction

Write code to extract X offset from delta tensor for batch n, kernel index k, output position (h_out, w_out):

.6: Base Position Calculation

Problem 1.6: Base Position Calculation

Write code to calculate base sampling position h_start from output position h_out and stride:

.7: Final Sampling Position

Problem 1.7: Final Sampling Position

Write code to calculate final sampling position sample_y from h_start, kh, dilation, and delta_y:

.8: Bounds Check Condition

Problem 1.8: Bounds Check Condition

Write code to check if coordinates (y, x) are within image bounds (H, W):

.9: Safe Pixel Access

Problem 1.9: Safe Pixel Access

Write code to get pixel value at (y, x) from image img, returning 0.0 if out of bounds:

.10: Fractional Part Calculation

Problem 1.10: Fractional Part Calculation

Write code to calculate fractional part dx from q_x and its floor x0:

.11: Linear Interpolation

Problem 1.11: Linear Interpolation

Write code to linearly interpolate between v_left and v_right using weight dx:

.12: Bilinear Weight Calculation

Problem 1.12: Bilinear Weight Calculation

Write code to calculate bilinear interpolation weight for points p and q:

.13: Value Accumulation

Problem 1.13: Value Accumulation

Write code to accumulate weighted interpolated value to current_value using weight, mask, and interpolated:

.14: Corner Position Calculation

Problem 1.14: Corner Position Calculation

Write code to calculate all four corner positions (y0, x0), (y0, x1), (y1, x0), (y1, x1) from q_y and q_x:

.15: Mask Application

Problem 1.15: Mask Application

Write code to apply modulation mask m_k to an interpolated value:

TASK 2: CNN PYTORCH - WRITE CODE BLOCKS

.1: Conv Layer Definition

Problem 2.1: Conv Layer Definition

Write code to define a Conv2d layer with 32 input channels, 64 output channels, kernel size 3, padding 1:

.2: BatchNorm Layer Definition

Problem 2.2: BatchNorm Layer Definition

Write code to define a BatchNorm2d layer for 128 channels:

.3: MaxPool Layer Definition

Problem 2.3: MaxPool Layer Definition

Write code to define a MaxPool2d layer that halves spatial dimensions:

.4: Linear Layer Definition

Problem 2.4: Linear Layer Definition

Write code to define a Linear layer with 4096 inputs and 512 outputs:

.5: Dropout Layer Definition

Problem 2.5: Dropout Layer Definition

Write code to define a Dropout layer with probability 0.5:

.6: ReLU Activation

Problem 2.6: ReLU Activation

Write code to apply ReLU activation to variable x:

.7: Tensor Flattening

Problem 2.7: Tensor Flattening

Write code to flatten tensor x while preserving batch dimension:

.8: Device Transfer

Problem 2.8: Device Transfer

Write code to move tensor images to device:

.9: Optimizer Zero Grad

Problem 2.9: Optimizer Zero Grad

Write code to clear gradients from optimizer:

.10: Forward Pass

Problem 2.10: Forward Pass

Write code to perform forward pass through model with input images:

.11: Loss Calculation

Problem 2.11: Loss Calculation

Write code to calculate loss using criterion with outputs and labels:

.12: Backward Pass

Problem 2.12: Backward Pass

Write code to perform backward pass on loss:

.13: Optimizer Step

Problem 2.13: Optimizer Step

Write code to update model parameters using optimizer:

.14: Top-1 Prediction

Problem 2.14: Top-1 Prediction

Write code to get top-1 predictions from outputs:

.15: Accuracy Calculation

Problem 2.15: Accuracy Calculation

Write code to calculate accuracy from predicted and labels tensors:

.16: Model Training Mode

Problem 2.16: Model Training Mode

Write code to set model to training mode:

.17: Model Evaluation Mode

Problem 2.17: Model Evaluation Mode

Write code to set model to evaluation mode:

.18: No Gradient Context

Problem 2.18: No Gradient Context

Write code to create context where gradients are disabled:

.19: Dataset Size Calculation

Problem 2.19: Dataset Size Calculation

Write code to calculate training size as 80% of total dataset size:

.20: Random Split

Problem 2.20: Random Split

Write code to split dataset into train_set and val_set with sizes train_size and val_size:

TASK 3: RNN - WRITE CODE BLOCKS

.1: Character Set Creation

Problem 3.1: Character Set Creation

Write code to create sorted list of unique characters from text:

.2: Character to Index Mapping

Problem 3.2: Character to Index Mapping

Write code to create dictionary mapping each character to its index:

.3: Index to Character Mapping

Problem 3.3: Index to Character Mapping

Write code to create dictionary mapping each index to its character:

.4: Input Sequence Creation

Problem 3.4: Input Sequence Creation

Write code to create input sequence (all characters except last) from text:

.5: Target Sequence Creation

Problem 3.5: Target Sequence Creation

Write code to create target sequence (all characters except first) from text:

.6: One-Hot Vector Creation

Problem 3.6: One-Hot Vector Creation

Write code to create one-hot vector of given size with 1 at given index:

.7: Input-to-Hidden Weight Initialization

Problem 3.7: Input-to-Hidden Weight Initialization

Write code to initialize W_xh weight matrix for RNN with hidden size H and vocab size V:

.8: Hidden-to-Hidden Weight Initialization

Problem 3.8: Hidden-to-Hidden Weight Initialization

Write code to initialize W_hh weight matrix for RNN with hidden size H:

.9: Hidden Bias Initialization

Problem 3.9: Hidden Bias Initialization

Write code to initialize bias vector b_xh for hidden layer with size H:

.10: Output Weight Initialization

Problem 3.10: Output Weight Initialization

Write code to initialize W_hy weight matrix from hidden to output with vocab size V and hidden size H:

.11: Hidden State Initialization

Problem 3.11: Hidden State Initialization

Write code to initialize hidden state vector with zeros of size H:

.12: Input Contribution Calculation

Problem 3.12: Input Contribution Calculation

Write code to calculate input contribution to hidden state using W_xh and x_t:

.13: Hidden Recurrence Calculation

Problem 3.13: Hidden Recurrence Calculation

Write code to calculate recurrent contribution to hidden state using W_hh and previous hidden state h:

.14: Hidden State Update

Problem 3.14: Hidden State Update

Write code to update hidden state using tanh activation with all contributions and biases:

.15: Output Logits Calculation

Problem 3.15: Output Logits Calculation

Write code to calculate output logits s_t from hidden state h using W_hy and b_y:

.16: Character Index Lookup

Problem 3.16: Character Index Lookup

Write code to get index of character 'e' from char2idx dictionary:

.17: Input List Creation

Problem 3.17: Input List Creation

Write code to convert input sequence to list of one-hot vectors:

.18: Target Tensor Creation

Problem 3.18: Target Tensor Creation

Write code to convert target sequence to tensor of character indices:

.19: Logits Stacking

Problem 3.19: Logits Stacking

Write code to stack list of logits tensors into single tensor:

.20: Log Softmax Calculation

Problem 3.20: Log Softmax Calculation

Write code to calculate log softmax of logits along vocabulary dimension:

.21: NLL Loss Calculation

Problem 3.21: NLL Loss Calculation

Write code to calculate negative log likelihood loss from log_probs and targets:

.22: Gradient Calculation

Problem 3.22: Gradient Calculation

Write code to calculate gradient of loss with respect to W_xh:

DATA PREPROCESSING - WRITE CODE BLOCKS

.1: CIFAR-100 Mean Values

Problem 4.1: CIFAR-100 Mean Values

Write code to define CIFAR-100 normalization mean values:

.2: CIFAR-100 Std Values

Problem 4.2: CIFAR-100 Std Values

Write code to define CIFAR-100 normalization standard deviation values:

.3: ToTensor Transform

Problem 4.3: ToTensor Transform

Write code to create ToTensor transform:

.4: Normalization Transform

Problem 4.4: Normalization Transform

Write code to create Normalize transform with CIFAR-100 mean and std:

.5: Random Crop Transform

Problem 4.5: Random Crop Transform

Write code to create RandomCrop transform with size 32 and padding 4:

.6: Random Flip Transform

Problem 4.6: Random Flip Transform

Write code to create RandomHorizontalFlip transform:

.7: Transform Composition

Problem 4.7: Transform Composition

Write code to compose multiple transforms into single transform:

.8: CIFAR-100 Dataset Loading

Problem 4.8: CIFAR-100 Dataset Loading

Write code to load CIFAR-100 training dataset with transform:

.9: DataLoader Creation

Problem 4.9: DataLoader Creation

Write code to create DataLoader with batch size 128 and shuffle=True:

OPTIMIZATION AND TRAINING - WRITE CODE BLOCKS

.1: CrossEntropy Loss Definition

Problem 5.1: CrossEntropy Loss Definition

Write code to define CrossEntropy loss function:

.2: SGD Optimizer Definition

Problem 5.2: SGD Optimizer Definition

Write code to define SGD optimizer with learning rate 0.01 and momentum 0.9:

.3: Adam Optimizer Definition

Problem 5.3: Adam Optimizer Definition

Write code to define Adam optimizer with learning rate 0.001:

.4: Model Parameter Count

Problem 5.4: Model Parameter Count

Write code to count total number of parameters in model:

.5: Learning Rate Update

Problem 5.5: Learning Rate Update

Write code to multiply learning rate by 0.1 for all parameter groups:

.6: Model State Save

Problem 5.6: Model State Save

Write code to save model state dictionary to file 'model.pth':

.7: Model State Load

Problem 5.7: Model State Load

Write code to load model state dictionary from file 'model.pth':

.8: Gradient Clipping

Problem 5.8: Gradient Clipping

Write code to clip gradients to maximum norm of 1.0:

3cm

.9: Top-5 Accuracy

Problem 5.9: Top-5 Accuracy

Write code to calculate top-5 accuracy from outputs and labels:

Problem 5.10: Loss Item Extraction

Write code to extract scalar value from loss tensor:

DEBUGGING AND UTILITIES - WRITE CODE BLOCKS

.1: Tensor Shape Check

Problem 6.1: Tensor Shape Check

Write code to print shape of tensor x:

.2: Tensor Device Check

Problem 6.2: Tensor Device Check

Write code to check which device tensor x is on:

.3: Model Device Transfer

Problem 6.3: Model Device Transfer

Write code to move entire model to GPU:

.4: Gradient Existence Check

Problem 6.4: Gradient Existence Check

Write code to check if parameter has gradients:

.5: Memory Usage Check

Problem 6.5: Memory Usage Check

Write code to check CUDA memory usage:

.6: Random Seed Setting

Problem 6.6: Random Seed Setting

Write code to set PyTorch random seed to 42:

.7: Numpy Seed Setting

Problem 6.7: Numpy Seed Setting

Write code to set numpy random seed to 42:

.8: Model Summary

Problem 6.8: Model Summary

Write code to print model architecture:

.9: Batch Dimension Check

Problem 6.9: Batch Dimension Check

Write code to get batch size from tensor x:

Problem 6.10: Tensor Type Conversion

Write code to convert tensor x to float type:

ANSWER BANK

Task 1 - Deformable CNN Answers:

Problem Task 1 - Deformable CNN Answers:

1.1: `y0 = int(np.floor(q_y))`

1.2: `x1 = int(np.ceil(q_x))`

1.3: `k = kh * K_w + kw`

1.4: `delta_y = delta[n, 2 * k, h_out, w_out]`

1.5: `delta_x = delta[n, 2 * k + 1, h_out, w_out]`

1.6: `h_start = h_out * stride`

1.7: `sample_y = h_start + kh * dilation + delta_y`

1.8: `if 0 <= y < H and 0 <= x < W:`

1.9: `value = img[y, x] if (0 <= y < H and 0 <= x < W) else 0.0`

1.10: `dx = q_x - x0`

1.11: `result = v_left * (1 - dx) + v_right * dx`

1.12: `weight = (1 - abs(p_x - q_x)) * (1 - abs(p_y - q_y))`

1.13: `current_value += weight * mask * interpolated`

1.14:
```
y0, x0 = int(np.floor(q_y)), int(np.floor(q_x))
y1, x1 = y0 + 1, x0 + 1
```

1.15: `modulated_value = m_k * interpolated_value`

Task 2 - CNN PyTorch Answers:

Problem Task 2 - CNN PyTorch Answers:

2.1: `self.conv = nn.Conv2d(32, 64, kernel_size=3, padding=1)`

2.2: `self.bn = nn.BatchNorm2d(128)`

2.3: `self.pool = nn.MaxPool2d(2, 2)`

2.4: `self.fc = nn.Linear(4096, 512)`

2.5: `self.dropout = nn.Dropout(0.5)`

2.6: `x = F.relu(x)`

2.7: `x = x.view(x.size(0), -1)`

2.8: `images = images.to(device)`

2.9: `optimizer.zero_grad()`

2.10: `outputs = model(images)`

2.11: `loss = criterion(outputs, labels)`

2.12: `loss.backward()`

2.13: `optimizer.step()`

2.14: `_, predicted = torch.max(outputs, 1)`

2.15: `accuracy = (predicted == labels).sum().item() / labels.size(0) * 100`

2.16: `model.train()`

2.17: `model.eval()`

2.18: `with torch.no_grad():`

2.19: `train_size = int(0.8 * len(dataset))`

2.20: `train_set, val_set = random_split(dataset, [train_size, val_size])`

Task 3 - RNN Answers:

Problem Task 3 - RNN Answers:

3.1: `chars = sorted(list(set(text)))`

3.2: `char2idx = {ch: i for i, ch in enumerate(chars)}`

3.3: `idx2char = {i: ch for i, ch in enumerate(chars)}`

3.4: `input_seq = text[:-1]`

3.5: `target_seq = text[1:]`

3.6: `vec = torch.zeros(size)`
`vec[idx] = 1.0`

3.7: `W_xh = torch.randn(H, V, requires_grad=True) * 0.1`

3.8: `W_hh = torch.randn(H, H, requires_grad=True) * 0.1`

3.9: `b_xh = torch.zeros(H, requires_grad=True)`

3.10: `W_hy = torch.randn(V, H, requires_grad=True) * 0.1`

3.11: `h = torch.zeros(H)`

3.12: `input_contrib = W_xh @ x_t`

3.13: `hidden_contrib = W_hh @ h`

3.14: `h = torch.tanh(W_xh @ x_t + b_xh + W_hh @ h + b_hh)`

3.15: `s_t = W_hy @ h + b_y`

3.16: `idx = char2idx['e']`

3.17: `inputs = [one_hot(char2idx[ch], V) for ch in input_seq]`

3.18: `targets = torch.tensor([char2idx[ch] for ch in target_seq], dtype=torch.long)`

3.19: `logits = torch.stack(logits_list)`

3.20: `log_probs = F.log_softmax(logits, dim=1)`

3.21: `loss = F.nll_loss(log_probs, targets)`

3.22: `grad_W_xh = torch.autograd.grad(loss, W_xh, retain_graph=True)[0]`

Additional Sections Answers:

Problem Additional Sections Answers:

4.1: `mean = [0.5071, 0.4867, 0.4408]`

4.2: `std = [0.2675, 0.2565, 0.2761]`

4.3: `transform = transforms.ToTensor()`

4.4: `normalize = transforms.Normalize(mean=[0.5071, 0.4867, 0.4408], std=[0.2675, 0.2565, 0.2761])`

4.5: `crop = transforms.RandomCrop(32, padding=4)`

4.6: `flip = transforms.RandomHorizontalFlip()`

4.7: `transform = transforms.Compose([transform1, transform2, ...])`

4.8: `dataset = CIFAR100(root='./data', train=True, transform=transform)`

4.9: `loader = DataLoader(dataset, batch_size=128, shuffle=True)`

5.1: `criterion = nn.CrossEntropyLoss()`

5.2: `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)`

5.3: `optimizer = optim.Adam(model.parameters(), lr=0.001)`

5.4: `total_params = sum(p.numel() for p in model.parameters())`

5.5: `for param_group in optimizer.param_groups:  param_group['lr'] *= 0.1`

5.6: `torch.save(model.state_dict(), 'model.pth')`

5.7: `model.load_state_dict(torch.load('model.pth'))`

5.8: `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)`

5.9: `_, top5_pred = outputs.topk(5, dim=1)`
`top5_acc = top5_pred.eq(labels.view(-1, 1).expand_as(top5_pred)).sum().item()`

5.10: `loss_value = loss.item()`

6.1: `print(x.shape)`

6.2: `print(x.device)`

6.3: `model = model.to('cuda')`

6.4: `if param.grad is not None:`

6.5: `print(torch.cuda.memory_allocated())`

6.6: `torch.manual_seed(42)`

6.7: `np.random.seed(42)`

6.8: `print(model)`

6.9: `batch_size = x.size(0)`

6.10: `x = x.float()`

Additional Answers

Problem Additional Answers

C1: 32 x 32, C2: 8 x 8, C3: 4096, C4: 73856, C5: 256

S1: numpy, torch.nn, F, transforms

S2: nn.Module, super

S3: def forward(self, x)

S4: nn.CrossEntropyLoss()

S5: optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

M1: x = self.dropout(x) if self.training else x

M2: for param_group in optimizer.param_groups: param_group['lr'] *= 0.1

M3: torch.save(model.state_dict(), 'model.pth')

M4: outputs = model(images.to(device))

M5: correct = (predicted == labels).sum().item()