

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

$$p_{lt} = (\lfloor q_x \rfloor, \lfloor q_y \rfloor) \quad (\text{left top}) \quad (1)$$

$$p_{rt} = (\lceil q_x \rceil, \lfloor q_y \rfloor) \quad (\text{right top}) \quad (2)$$

$$p_{lb} = (\lfloor q_x \rfloor, \lceil q_y \rceil) \quad (\text{left bottom}) \quad (3)$$

$$p_{rb} = (\lceil q_x \rceil, \lceil q_y \rceil) \quad (\text{right bottom}) \quad (4)$$

.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: $3 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 256$
- Spatial reduction: $32 \rightarrow 16 \rightarrow 8 \rightarrow 4$
- FC layers: $4096 \rightarrow 512 \rightarrow 256 \rightarrow 100$

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: $3 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 256$
- Spatial: $32 \rightarrow 16 \rightarrow 8 \rightarrow 4$
- FC: $4096 \rightarrow 512 \rightarrow 256 \rightarrow 100$

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: $0.8 * \text{len}(\text{dataset})$

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

- $\text{delta_y} = \text{delta}[n, 2*k, \text{h_out}, \text{w_out}]$
- $\text{delta_x} = \text{delta}[n, 2*k+1, \text{h_out}, \text{w_out}]$

"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: $3 \rightarrow ___ \rightarrow ___ \rightarrow ___ \rightarrow ___$
- CNN spatial: $32 \rightarrow ___ \rightarrow ___ \rightarrow ___$
- 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: $3 \rightarrow _ \rightarrow _ \rightarrow _ \rightarrow _$
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:

Challenge 4: Debugging Scenario

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. $(1 - p_x - q_x) * (1 - p_y - q_y)$ 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.